Jigsaw Puzzle Solver

A Major Qualifying Project
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
degree of Bachelor of Science

Submitted by:
Matthew Aguiar
Jack Ayvazian
Jadon Laforest
Winnie Ly
Ryan Wheeler

Advised by:
Professor Jacob Whitehill

Date: March 4th, 2022

This report represents work of one or more WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review.

Abstract

Solving jigsaw puzzles presents a unique challenge for artificial intelligence through the application of computer vision. We explored different puzzle matching techniques by considering piece edges, internal features, and color to design a hybrid system that serves as a puzzle solving user aid. By uploading images of the scattered pieces and the completed puzzle, the matching algorithm will attempt to find the proper locations for each piece and provide visual cues to the user. The application was tested on a sample of varied puzzles and sizes, where the accuracy was dependent on the complexity of the puzzle features and declined linearly as the puzzle was split into more pieces.

Table of Contents

Abstract 1

Table of Contents 2

Table of Figures 4

1. Introduction 6

2. Background 8
- Edge Detection 9
- Representing Edges as Splines 12
- Bezier Curves 13
- Subsequence Matching 14
- Dynamic Time Warping 15
- SIFT 17
- Homography 22
- Color Histograms 22
- Prior Work on Jigsaw Puzzle Matching 23

3. Methods 27
- Overview 27
- Splines and Edge Detection 27
  - Representation 28
  - Visualization 30
  - Subsequence Matching 31
- SIFT 34
  - Segmentation 35
  - Matching 37
  - Sorting 39
  - Limitations 42
- Color Histograms 42
- UI 45
  - Workflow 45
  - Streamlit 47
  - Flask and Alternatives 48
- Experimentation 48
  - Constraints 49
  - Procedure 50

4. Results and Analysis **52**
Edge Detection 52
SIFT 54
Puzzle Image Analysis 55
Color Histograms 59
5. Conclusion **61**
Future Work: Improving Practicality 62
Robust Masking 62
Error Checking 62
Other Approaches 64
Lessons Learned 65
References **66**
Image References 68
Appendix **69**
Appendix A: Flask User Interface Design **69**
Appendix B: Streamlit User Interface Design **70**
Appendix C: Puzzle Testing Data **71**

Table of Figures

Figure 1: Puzzle examples solved by different methods 6

Figure 2: Inspiration for envisioned application interface 7

Figure 3: Puzzle solving approaches 9

Figure 4: Puzzle piece and its edges 10

Figure 5: Types of edges 10

Figure 6: Puzzle edge points into vectors 13

Figure 7: Puzzle piece outline with linear splines 15

Figure 8: Implementation of DTW without Window Constraint 16

Figure 9: Implementation of DTW with Window Constraint 17

Figure 10: Example of SIFT keypoint detection and matching 18

Figure 11: Keypoint with descriptors 21

Figure 12: Homography example 22

Figure 13: Normalized Color Histograms 23

Figure 14: Example of a puzzle piece spline 28

Figure 15: Spline matching procedure 29

Figure 16: a.Ranked matches between puzzle pieces 33
Figure 16: b. Pair of matching pieces

Figure 17: Flowchart for the SIFT puzzle piece matching process 35

Figure 18: Sample puzzle image and corresponding mask 36

Figure 19: SIFT keypoint matching 38

Figure 20: Three consecutive matches from the user interface 40

Figure 21: Misaligned grid of puzzle piece center coordinates 41

Figure 22: Division of puzzle grid 43

Figure 23. OpenCV's compareHist() function 44

Figure 24 a. Spline points stopping at bottom left 53
Figure 24 b.Spline points going all the way around a piece

Figure 25: Average matching accuracy line graph for SIFT + Color Histograms 55

Figure 26: Puzzles with minimum and maximum SIFT matching accuracy 56

Figure 27: Stacked bar chart showing the SIFT matching accuracy dropoff 57

Figure 28: Puzzle images with notable jumps in accuracy between puzzle sizes 58

Figure 29: Distorted contour of matched piece 63

1. Introduction

During the COVID-19 pandemic, classic at-home activities such as jigsaw puzzles saw a surge in demand, with sales up 300% to 400% (Bodenheimer, 2020). Jigsaw puzzles are a great mind exercise for all ages which require careful observation and concentration. Depending on the puzzle size and difficulty, this matching process can take numerous hours of trial and error. However, what if a computer could analyze a picture of all the puzzle pieces and determine the matches in seconds? While this may defeat the purpose of solving a puzzle, it could help in alleviating moments of frustration when stuck on a certain part or provide a starting point for a seemingly overwhelming task. Additionally, a user may be curious about the capabilities of an Artificial Intelligence (AI) puzzle solver compared to a human.

Puzzle piece matching presents a task for AI, made possible through applying Computer Vision. The implicit way humans analyze puzzles would be mimicked and broken down into three separate types of information: piece edges, internal features, and color.

Figure 1: From left to right, puzzles that would be solved by-looking solely at piece edges (Habicht), puzzle features (Lamamour), or colors (Signals)

When considering these types of information individually, matching accuracy is limited and largely dependent on the puzzle; attempting to match pieces from a

monochromatic puzzle solely based on features or color would be ineffective. That's why the goal for this project was to combine these different types of information to create a robust, hybrid matching algorithm for puzzles. While there are libraries already in place which can extract points of interest (features) from images, there are several other key components necessary to make a full scale jigsaw puzzle solving application that is user-friendly. The envisioned product would allow a user to simply input images of the puzzle pieces scattered and of the completed puzzle usually provided on the box. The app would then output individual steps of matching pieces to guide the user towards completion, similar to Figure 2 below.

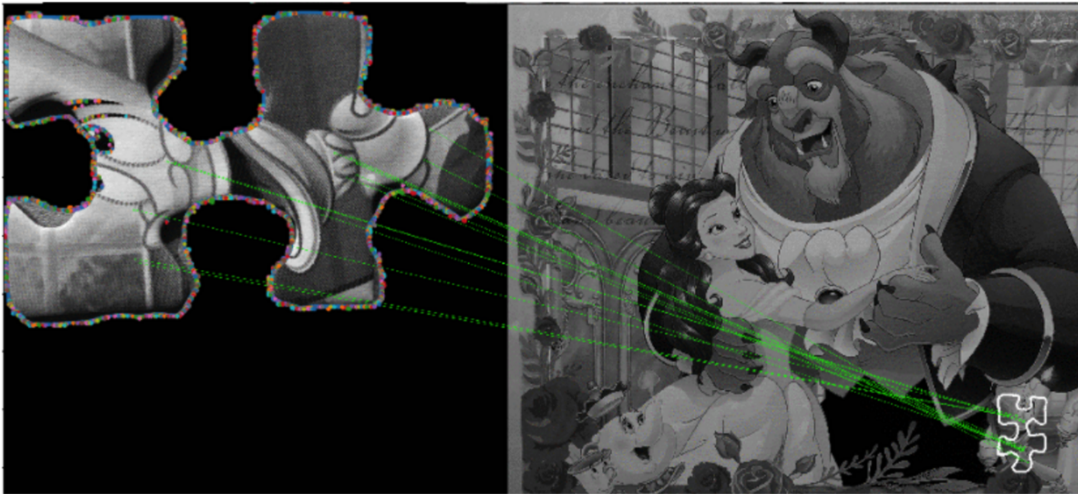


Figure 2: Inspiration for our envisioned product, where the matching location of a puzzle piece would be easy to visualize (github.com/whitcrrd).

In the following background section, the concepts behind edge detection, matching algorithms, feature recognition, and color histograms will be explored to gain a foundation of what is happening behind the scenes of these computer vision techniques.

2. Background

Jigsaw puzzles are a popular hobby for all ages. The first jigsaw puzzle was created in 1762 when John Splisbury printed a map onto a piece of wood and carved out the countries to help teach young children geography (The History of Jigsaw Puzzles, 2019). Today jigsaw puzzles come in a variety of shapes and sizes, being made out of different materials, most commonly wood and cardboard. Puzzles are made to be solved by putting all the individual pieces that were cut out from a large image back together, but what if assembling a puzzle required no thinking or puzzle skills at all? We want to create a system that can do the solving work for a puzzle builder, providing clear steps for putting the puzzle together.

Computer vision is a vital part of our project, allowing our program to visualize the puzzle pieces in order to solve them. Computer vision is a field that aims to imitate the human visual system by extracting meaningful information from digital images or videos, which can be applied towards automating certain tasks. Solving jigsaw puzzles presents a unique challenge for a computer vision application, as high precision is required to model and match pieces. A hybrid of color and shape information used to solve puzzles can be divided into two separate computer vision processes: analyzing piece edges and their internal features. When solving puzzles, a “relative” piece-to-piece matching approach is typically used, however there is an alternative “absolute” approach where pieces are mapped to the completed picture of the puzzle independently. Both of these computer vision puzzle solving strategies will be explored in the following chapter.

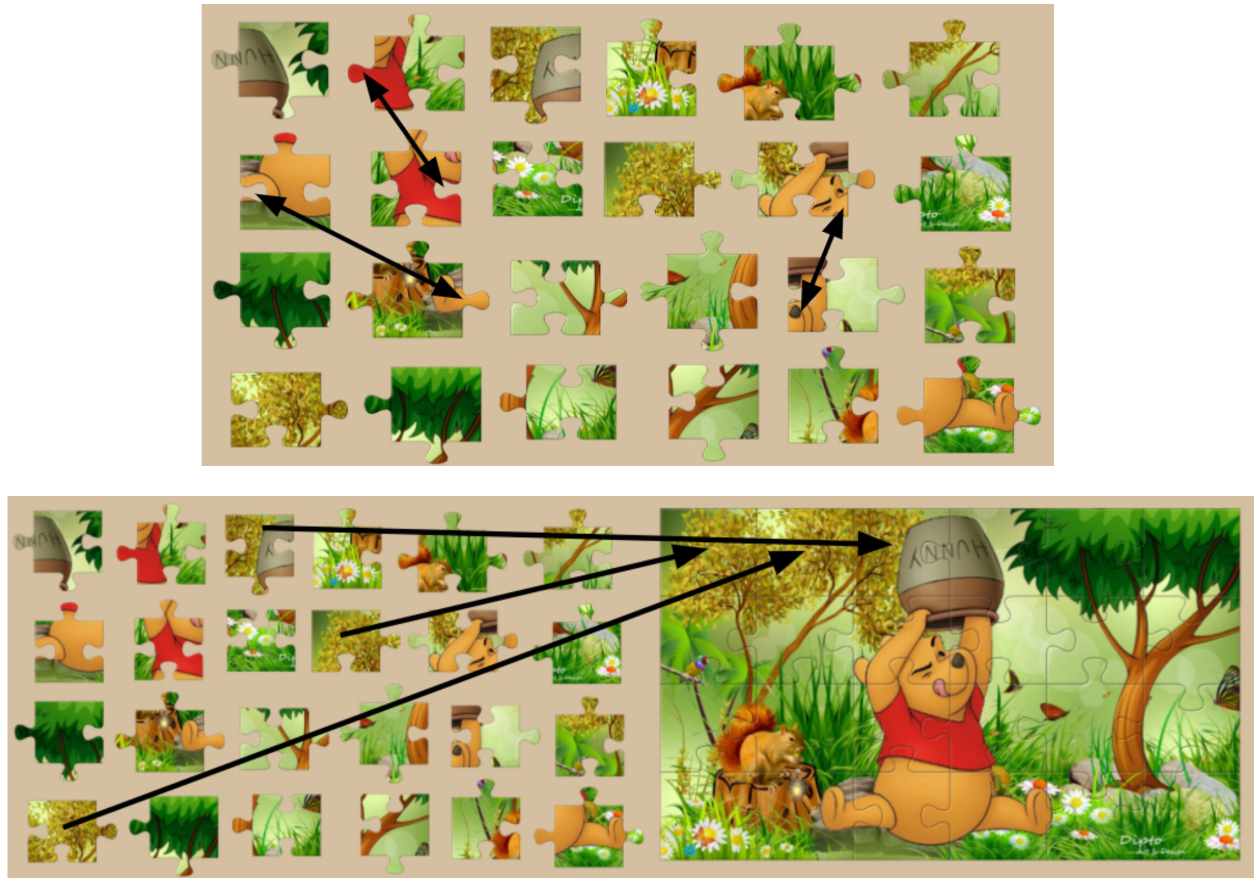


Figure 3: The top illustrates a piece-to-piece solving approach. The bottom shows an absolute approach as pieces are matched left to right, top to bottom.

Edge Detection

Edge detection is an image analysis technique that uses changes in intensity of pixels to determine the boundaries of objects in an image (MathWorks, n.d.). Edge detection attempts to convert an image into only its edges, which can then be used in a multitude of applications including machine learning, machine vision, and image processing.

An edge is represented by an abrupt change in the intensity of local pixels. This discontinuity in intensity is ideally represented by either step discontinuity or line discontinuity (Jain, 1995). A step discontinuity is an abrupt change in the pixel intensity from one side of the discontinuity to another. An example of this would be a white puzzle piece against a black background.

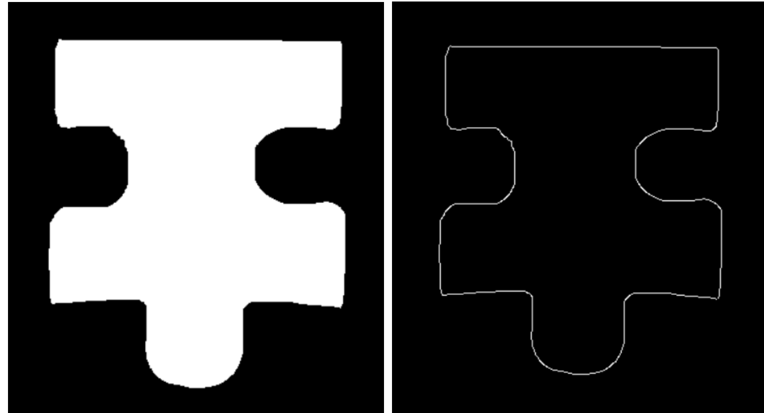


Figure 4: Example of a white puzzle piece and it's edges.

The abrupt change between the white piece and black background would be represented as an edge with step discontinuity. A line discontinuity is where the intensity abruptly changes but then returns back to its original state within some short distance. An example of this would be a thin black dotted line drawn on a white piece of paper. These types of discontinuity are rare in real images because the change in intensity is often not so drastic. The two more common types of discontinuity are ramp edges and roof edges. Ramp edges are in place of step edges where the intensity changes over a small, finite distance rather than immediately from pixel to pixel. Roof edges are where the intensity changes then returns more gradually than a line edge.

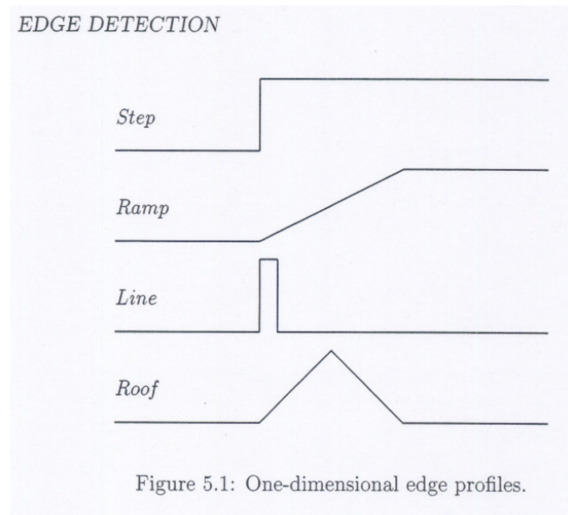


Figure 5: Types of edges (Jain, R., et al., 1995)

A common edge detection algorithm used is Canny Edge Detection. This was created by John F Canny in 1986 who concluded that this method of edge detection had very good detection, being able to reduce the possibility of false positives and negatives when detecting edges, as well as good localization. Good localization in the algorithm uses guaranteed edges to make accurate and educated assumptions of less obvious edges that are detected by the program. Canny Edge Detection runs in four stages: noise reduction, calculating the intensity gradient, suppressing non-maximum edge pixels, and hysteresis thresholding (HIPR, n.d).

The first step of Canny Edge Detection is to remove the noise of the image. Noise in an image is caused by random variation in image intensity, which can be seen as film grain and variations to the pixel level in digital images (Noise , n.d.). This causes many undesirable effects in an image such as blurry objects and unrealistic edges (Boyat, & Joshi, 2015). Noise degrades the effectiveness of the edge detection, so a Gaussian filter is applied to reduce the noise of the image before any edge detection is run. By applying a gaussian filter, the image is slightly blurred, removing details and noise that would otherwise lead to more false positives and negatives when detecting the edges. A 5x5 filter is most commonly used for this step (OpenCV, n.d.).

The next step is to calculate the intensity gradient of the image. The image is filtered to get the first derivative in both the horizontal [G_x] and vertical directions [G_y]. These values then can be used to find the edge gradient, as well as the angle of direction that the edge is going at a certain pixel with the formulas shown below. Gradient direction (angle) is rounded to one of the four directions, either horizontal, vertical and two diagonals (0, 45, 90, 135) (OpenCV, n.d.).

$$Edge_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

The third step in the process is suppressing non-maximum edge pixels. This step is where all pixels are checked and ones that are guaranteed to not be edges

are suppressed. Pixels are checked to see if it is a local maximum of the other pixels that are in its neighborhood in the direction of the gradient (given by the previous step). Local maximums are considered for the next step in the process (OpenCV, n.d.).

The fourth and last step in the Canny Edge Detection process is hysteresis thresholding. This step takes all the edges found through the previous step and determines whether or not it is truly an edge using two threshold values, a max threshold and min threshold. If an edge has an intensity gradient that is larger than the max threshold, then it is considered a definite edge. In the same fashion, edges that have an intensity gradient below the min threshold are not considered as edges. If the intensity gradient of an edge falls between the two values, then what determines if it will be counted as an edge are its neighbors. If the edge is connected to a definite edge then it is also considered an edge, and if it is not connected to an edge it is suppressed. This allows the algorithm to make assumptions about pixels that other edge detection methods would otherwise not be able to make (OpenCV, n.d.).

Representing Edges as Splines

Once edges of a puzzle piece are determined, the next step is to represent them as splines. The edge points detected can be transformed into a spline to create an accurate representation of a curve. A spline is a mathematical representation of a sequence of points, displayed as complex curves and surfaces (12). Splines are a polynomial curve with the general form $y = a + bx + cx^2 + dx^3 + \dots$. There are two types of curves that a spline can have. There is an interpolating curve, in which the line of the curve passes through the control points. The second type is an approximating curve, where the line passes near the control points. In our puzzle we will be using an interpolating curve for our spline and the control points are the pixel x and y coordinates of the pixels along the puzzle piece edges.

Because of the points along the edge of each puzzle piece, we have to represent the edges as linear splines rather than cubic splines to simplify calculations. Three points along the edge of a piece can be turned into two vectors, which then the angle between the two can be calculated with the following formula

$$\theta = \arccos[(x_a * x_b + y_a * y_b) / (\sqrt{(x_a^2 + y_a^2)} * (\sqrt{(x_b^2 + y_b^2)}))]$$

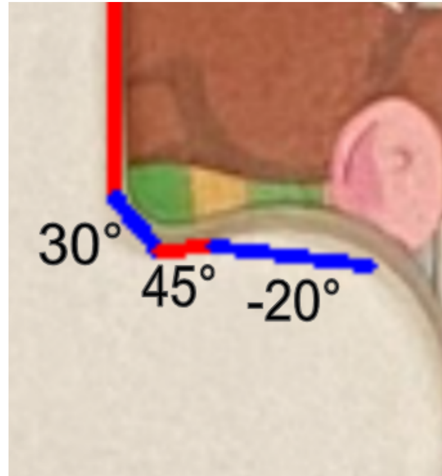


Figure 6: Three points along the edge of a piece turned into vectors / linear splines.

Bezier Curves

A Bezier curve is used to draw curves that are defined by a set of control points. In its most simple form, the linear bezier curve represented by points P_0 and P_1 from $0 \leq t \leq 1$ is given by

$$B(t) = (1-t)P_0 + tP_1$$

T represents the proportion of how far the point of the curve is from each control point. For the linear bezier curve at t = 0.5 for example, the curve passes through a point that is 50% along a line that is drawn between anchor points. Higher order bezier curves can be found with the following formula

$$B(t_k) = \sum_{i=0}^{n} P_i b_{i,n}(t_k)$$

with

$$b_{i,n}(t_k) = \frac{n!}{i!(n-i)!} {t_k}^i (1 - t_k)^{n-i}$$

This returns the bezier curve for N number of control points (Yu, et al., 2012).

For comparing two bezier curves we will have to utilize the arc length of segments along the curve. By storing these in an array for each piece, the curves will be comparable helping determine if two puzzle pieces are a good match. We should also try using arc angles between these points to calculate the angle. These could be stored in an array for each piece and compared similarly to the arc length data.

To calculate the angle between two points along our edge the points a = (x_1,y_1) and b = (x_2,y_2) can be represented as vectors A = $[x_a, y_a]$ and B = $[x_b, y_b]$. The angle can be calculated buy the following formula

$$\theta = \arccos[(x_a * x_b + y_a * y_b) / (\surd(x_a^2 + y_a^2) * (\surd(x_b^2 + y_b^2))]$$

Subsequence Matching

Sequence matching is a way to process and produce a pattern sequence from the input data given. There are two categories for sequence matching, one of which is whole sequence matching while the other one is subsequence matching. With whole sequence matching, it requires the sequence entries to all be the same length as the query length as it finds matches in the given dataset. For subsequence matching, it finds all the subsequences that occur in a longer dataset that matches the query regardless of its length pertaining to the query or to the dataset. Whole sequence matching suffers in comparison to subsequence matching as it is susceptible to non-Euclidean measures meaning that it is sensitive to irregular changes in the dataset (Han et al., n.d.). In particular, computing the Longest Common Subsequence (LCS) tends to be more robust than whole sequence matching as it encounters noise in the dataset.

The Longest Common Subsequence (LCS) is an algorithm in which given two datasets, both with varying lengths, the resulting LCS would consist of a sequence that was found in both datasets. LCS is effective for matching data types such as strings, however when handling data that do not produce exact matches such as user data, there needs to be an established threshold to handle variances in data. With a threshold, it allows for matches to be true even if the matches are not exact, as long as their difference is below the specified threshold (Bellogin et al., n.d.).

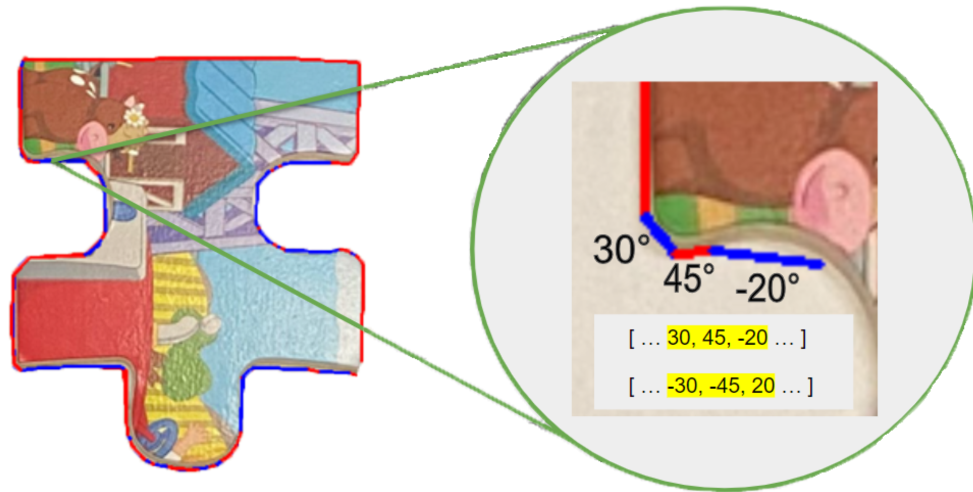


Figure 7: A puzzle piece outlined in linear splines.

Dynamic Time Warping

Dynamic Time Warping (DTW) is an algorithm that calculates and measures the distance and similarity between two sequences that can either be of the same length or different length as they vary in speed. When comparing the distances between two sequences, DTW is more effective in calculating the distance as it conducts one-to-many matches and there are no left-outs that would occur as everything is mapped to each other. When implementing DTW, there are rules that must be followed in order to calculate the similarity and distances between the two sequences. These rules fall under the categories of: boundary, monotonicity, and continuity constraints. The boundary constraint states that the start index of the first sequence must match to the start index of the second sequence but it can be

mapped to another index on the sequence. This constraint is also applied to the ending index of both sequences. The monotonicity constraint states that the mapping of the indices from the first sequence and second sequence must be monotonically increasing and vice-versa. The continuity constraint states that when the mapping of indices happens along with monotonicity, it eliminates the possibility of cross matching and having no indices being left out (Zhang, 2020 & Deriso, 2019).

Along with those three main constraints, there are others that can improve the DTW algorithm further. The window constraint helps limit the number of elements a sequence can match to on another sequence (Zhang, 2020). The slope constraint helps avoid extreme movements in one direction when calculating the warping path (Alizadeh, 2020).

To get the DTW between two sequences, first take the cost which is calculated from taking the absolute differences for each matched pair of indices. After calculating the cost, it is then added to the minimum cost of arrays with varying lengths: (i – 1, j), (i , j – 1), and (i – 1, j – 1). Figure 8 demonstrates the implementation of the algorithm as it would generate a matrix that would show the cost calculated from each index on the two sequences, disregarding the window constraint for now.

```
int DTWDistance(s: array [1..n], t: array [1..m]) {
    DTW := array [0..n, 0..m]

    for i := 1 to n
        for j := 1 to m
            DTW[i, j] := infinity
    DTW[0, 0] := 0

    for i := 1 to n
        for j := 1 to m
            cost := d(s[i], t[j])
            DTW[i, j] := cost + minimum(DTW[i-1, j  ],    // insertion
                                        DTW[i  , j-1],    // deletion
                                        DTW[i-1, j-1])    // match

    return DTW[n, m]
}
```

Figure 8: Implementation of DTW without Window Constraint (Zhang, 2020)

The last result in the matrix is the calculated distance between the two sequences. When adding in the window constraint shown in Figure 9, the implementation is adjusted slightly as it is accommodating the range in which the elements match to each other (Zhang, 2020).

```
int DTWDistance(s: array [1..n], t: array [1..m], w: int) {
    DTW := array [0..n, 0..m]

    w := max(w, abs(n-m)) // adapt window size (*)

    for i := 0 to n
        for j:= 0 to m
            DTW[i, j] := infinity
    DTW[0, 0] := 0

    for i := 1 to n
        for j := max(1, i-w) to min(m, i+w)
            DTW[i, j] := 0

    for i := 1 to n
        for j := max(1, i-w) to min(m, i+w)
            cost := d(s[i], t[j])
            DTW[i, j] := cost + minimum(DTW[i-1, j  ],    // insertion
                                        DTW[i  , j-1],    // deletion
                                        DTW[i-1, j-1])    // match

    return DTW[n, m]
}
```

Figure 9: Implementation of DTW with Window Constraint (Zhang, 2020)

SIFT

An alternative technique applicable to solving jigsaw puzzles with computer vision is to locate and match internal areas of interest between images. In 2004, David Lowe presented the Scale Invariant Feature Transform (SIFT) algorithm which identifies features of an image (keypoints) and neighboring descriptors.

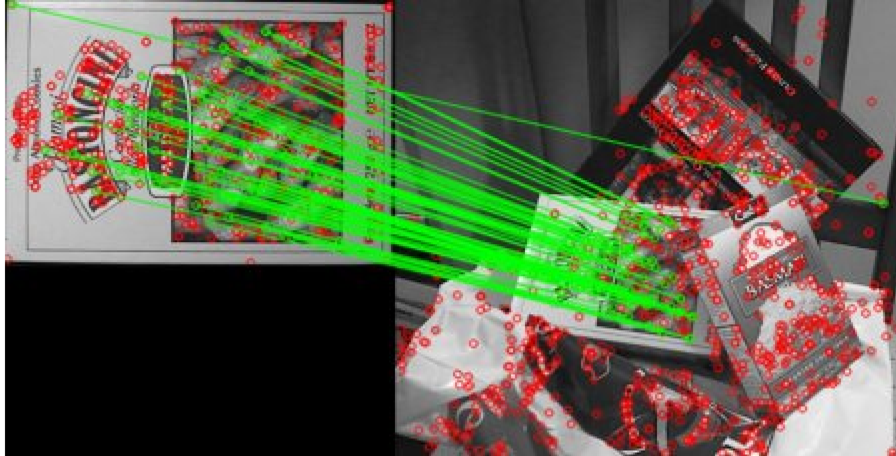


Figure 10: Example of SIFT in action. Each red circle on the images represents an individual keypoint. The green lines represent a match between the left and right image's respective keypoints (OpenCV, 2022).

An example of SIFT and the subsequent keypoint matching is shown above in Figure 10. As the name suggests, this algorithm is invariant to the image scale and orientation, and it operates in four major stages: Scale-space extrema detection, keypoint localization, orientation assignment, and keypoint descriptor (Lowe, 2004). Subsequently, keypoint matching is the final step to connect the filtered keypoints between images.

The first stage of SIFT searches all image locations and scales to identify potential points of interest. As the interpretation of real world objects depends on their scale, a multi-scale representation is essential to extract meaningful information about objects. The notion of scale-space addresses this concern, by considering all scales simultaneously with the use of the Gaussian function. The scale-space of an image can be described as a function produced from the convolution of a Gaussian kernel with an input image:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

Where L is the scale-space, G is the Gaussian Blur operator, and I is the image, with parameters x and y representing location coordinates and σ as the "scale" (Lowe, 2004 & Lindeberg, 1994). This convolution operation blurs the image, and the

difference of Gaussian D can be computed from the difference of two images separated by a gaussian blurring factor k:

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y)$$
$$= L(x, y, k\sigma) - L(x, y, \sigma)$$

Taking the D(x, y, σ) calculated, each potential keypoint is compared to its eight neighbor pixels, as well as the nine neighbors at the scale-space above and below it. The pixel is considered a keypoint if it is the largest or smallest value of all these 26 neighbors. This short process typically eliminates a lot of the egregious potential keypoints generated.

The second stage of SIFT, Keypoint Localization, where the potential points from the first stage are filtered and refined to only the most accurate keypoints. Keypoints with low contrast or ones located along the edge of the image are typically the first ones eliminated. This process uses a Taylor expansion of the scale-space function to fit a 3D quadratic function to the local points:

$$D(x) = D + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x$$

In this equation, D and its derivatives are evaluated at the sample point and x = (x, y, σ)^2 is the offset from this point. To find the extremum, the derivative of the function is taken with respect to x and setting it to zero:

$$\hat{x} = -\frac{\partial^2 D^{-1}}{\partial x^2} \frac{\partial D}{\partial x}$$

These equations determine the offset, which if determined to be less than some threshold value (e.g., 0.3) in any direction indicates the location has low contrast, and should not be used as a potential keypoint.

By nature of the Gaussian function used in SIFT's first stage, points found along the edge of an image are very common. Edge points are not well defined since they have less neighbor points and therefore less to compare to, leading to more keypoints than that of a typical "middle" point. Using more complex but lightweight mathematical operations, SIFT naturally filters out edge points that are not likely to be accurate.

The third stage of SIFT, Orientation Assignment, makes the algorithm rotation invariant by producing it at various different rotations and orientations. When used in the matching step, it helps to account for varying positions of the keypoint from source to destination. After the second stage, we are left with only the most stable keypoints. For each of these keypoints, an orientation histogram is created, with 36 bins covering 360 degrees. The gradient is taken from the keypoint and all neighboring pixels. The magnitude of the gradient is then put into the bin of the respective gradient direction (angle). The highest peak in the histogram is then considered the main orientation and any peaks greater than 80% of the main peak are also considered. All these orientations are then made into their own keypoints with the exact same location and scale, but with different direction vectors (Tyagi, 2019).

Up until this point, each identified keypoint has a location, scale, and orientation. The fourth phase of SIFT is where a Keypoint Descriptor is computed which is how the keypoint is stored and displayed. This process is done in a specific manner to ensure each keypoint is as invariant as possible to allow for matching without regard for illumination, orientation, or 3D viewpoint. A 4x4 array of histograms around each keypoint is created. Within each histogram is an 8 bin orientation histogram. This comes together for a 128 dimensional vector that represents the area around each keypoint.

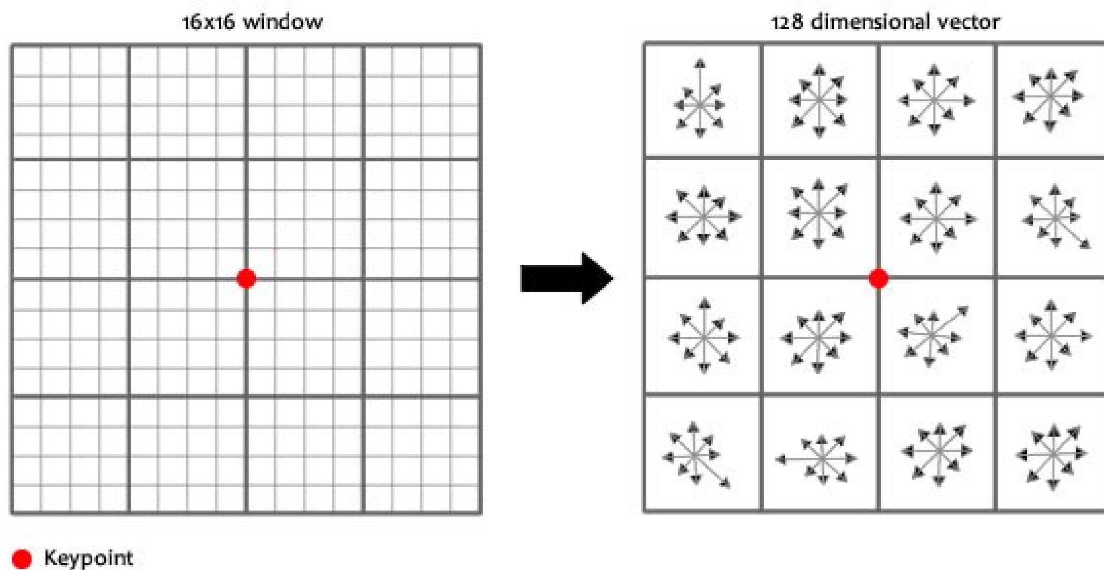


Figure 11: Keypoint with descriptors (Tyagi, D, 2020).

A quick vector normalization step eliminates the chance for illumination variance and the keypoint is in its final form.

The keypoint descriptors calculated from SIFT can be applied towards matching keypoints between two images. Each keypoint of the "query" image is matched to its nearest neighbor keypoint of the "training" image based on the minimum Euclidean distance of their descriptors. However, many keypoints will not have a correct match if they were not detected in the training image, or if they arose from background noise in the image. To filter out these features, Lowe proposed performing a "ratio-test", which he found to eliminate 90% of false matches while discarding 5% of correct matches (Lowe, 2004). This ratio test compares the distance between the closest neighbor match to the second closest neighbor, and rejects matches where the distance ratio is greater than 0.8. The logic behind this test is that "good" matches have to be distinct and significantly closer than the closest incorrect match. If the second closest match is in close proximity to the first, then the feature is too ambiguous to yield a reliable match.

Homography

The keypoint matches determined by the minimum Euclidean distance of descriptors can help identify where the two processed images intersect. Locating this image overlap can be especially useful for solving jigsaw puzzles, as the goal is to find where images of individual pieces are located with respect to the whole puzzle image. One method of transforming one image to another in this fashion is using homography, which is a 3x3 matrix used to map points between images on the same planar surface (Snavely, 2006). To determine the correct homography matrix among a mix of accurate and inaccurate keypoints, estimation algorithms like Random sample consensus (RANSAC) are utilized. RANSAC iterates through random combinations of keypoint sets and computes the number of inliers for the resulting homography matrix. In a least-squares approach, the homography containing the most inliers is taken as the final result (Snavely, 2006).

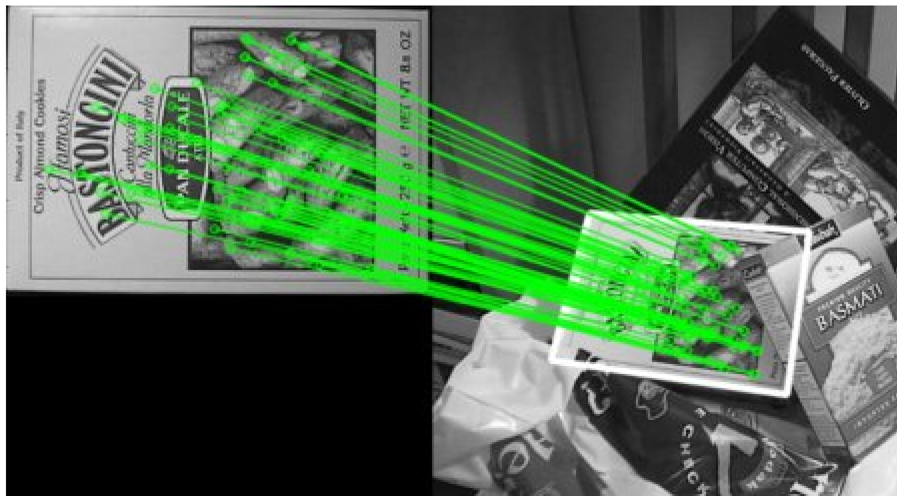


Figure 12: Example of homography applied to SIFT keypoints. The white box around the object on the right is the area with the most concentrated keypoint match inliers. (OpenCV, 2022).

Color Histograms

Histograms in general are a type of bar graph that groups numeric data into bins (MATLAB, n.d.). More specifically, color histograms are the distribution of

colors in the RGB space. The earliest use of color histograms were modeled as a gaussian cluster in a color space. This meant that an object could be modeled by a certain color characteristic, and the more a pixel deviated from this characteristic, the less likely it would be that object (Novak, 1992). Now histograms are able to represent homogeneous objects in the three dimensional RGB space. This allows for more complicated computer vision and deeper knowledge for analyzing images. Typically, color histograms are split into their three distinct color channels before doing individual analysis. An example of a few images' color histograms are shown below:

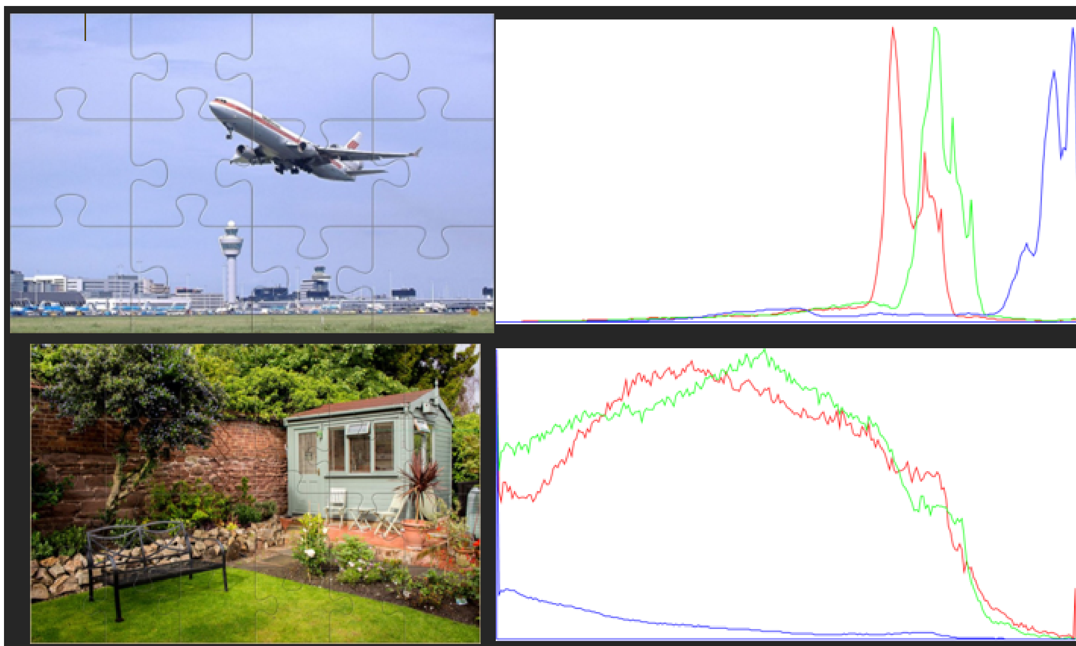


Figure 13: Two examples of normalized color histograms generated on different images

Because color is such a vital part of not only solving puzzles, but computer vision in general, color histograms can play a large role.

Prior Work on Jigsaw Puzzle Matching

The idea of using a computer to solve a jigsaw puzzle has been explored in the past, all using a multitude of different strategies and techniques. Rather than

the standard jigsaw puzzle, people have worked on irregular shapes and even perfect square pieces to put a creative twist on this problem. Some used strictly edge data, strictly image data, or a creative combination of both.

One jigsaw puzzle solver created by Travis Allen of Stanford University uses a combination of piece category, side length, and the curve of a side to assign a score to a match (Allen, 2016). It does not use the picture of the completed puzzle on the box at all. This project was also created entirely in Matlab and created for potential smartphone applications. Their algorithm utilized a Gaussian filter to blur the image, as well as a green screen to create better edge detection. The green screen acted as a solid background to more easily mask the pieces and distinguish them from the background and one another. Then puzzle pieces were categorized, and the sides of each piece are also categorized as either a flat side, a head, or a hole. The creator of this project broke the assembly into a global and local, distinguishing the edges from the inner pieces. Construction begins with an edge piece as a starting piece, and once each match is given a score the algorithm moves to the best match and calls that the new starting piece and continues. This method is used on pieces categorized as edges to assemble the border of the puzzle. The middle is solved from the open, uppermost left slot and solves left to right and downwards. This program found success with 12 to 16 piece puzzles, but struggled with 24 piece puzzles. In a few edge cases, color was found to not be helpful because the color drastically changes along the edge of a match. While this program had many flaws with larger puzzles, it was successfully able to solve smaller puzzles purely by looking at the disassembled pieces with no reference to the final image (Allen, 2016).

Unlike the previous puzzle solver, one created by Cris Zanoci and Jim Andress of Stanford University does not use edges at all, but rather attempts to solve puzzles where the pieces are all square, using image data alone. Similar to the other puzzle solver, this solver gives matches a score based on criteria that is then used to assemble the puzzle. One of the criteria used is a dissimilarity metric, which measures how similar the gradient distribution is between two puzzle piece edges. Dissimilarity is calculated using the Mahalanobis distance, which measures the

distance between a point and a probability distribution. The pieces were reconstructed by organizing them in a graph, specifically a minimum spanning tree. The accuracy of the program was measured with a direct comparison metric, which measured the percent of pieces that were in the correct position in comparison with the original image, and a neighbor comparison metric, which measured the percent of edges with are adjacent to their neighbor in the original image (in other words, pieces that fit together regardless of if they are in the right location). The mahalanobis gradient compatibility was found to perform really well when matching pieces based on image data alone. This algorithm was able to successfully solve a 2856-piece puzzle. Their algorithm also had a drawback where if a mistake was made there was no way to backtrack and correct the error. Overall the ability to solve such a high piece puzzle makes their methods worth investigating further for our own application (Zanoci & Andress, 2016).

This next puzzle solver was created by M. Makridis, N. Papamarkos, and C.Chamzas of Democritus University of Thrace, Greece. Unlike the previous two reviewed, this puzzle solver combines both shape and color information from the pieces to solve the puzzle. This method consists of five stages: Corner detection, color segmentation, comparing, iteration of previous stages, and a matching stage. This solver also uses a green screen to easily separate the background from the pieces. For comparison, this algorithm takes four points along one edge and compares the angle with four points from another piece's edge. This was designed in order to speed up the process and not check every single point. They also checked the luminescence at these points and compared it to the luminescence of the potential match. Then they compared the Euclidean distance between the connected sequence of points from one piece to the second piece. This is repeated with a different number of points in order to catch the errors that could have been missed the first time around. This in turn kept the algorithm fast and attempted to increase accuracy as well. The pieces were assembled using reference points and a shift and rotation formula. The puzzles tested by this algorithm were between three to ten pieces, all of which were cut into irregular shapes with multiple edges. This

differed a bit from the standard four sides we are used to when we think of a puzzle piece. (Makridis et al., 2005)

A different puzzle solving algorithm by Liang Liang and Zhongkai Liu utilizes the completed puzzle image and performs feature detection to solve jigsaw puzzles. The steps of this algorithm were to downsample the image and reduce noise, patch segment extraction, feature detection with SURF (Speeded-Up Robust Features), feature descriptor matching, geometric consistency check, and create the image. They decided to use SURF because through their preliminary testing they found that SURF was faster than SIFT. Note that SURF is still patented, so its use requires payment and is not included in default OpenCV packages. The feature points of a puzzle piece are generated using this algorithm and then are compared with the template image. Its location is estimated based on this comparison. Then its geometric consistency (i.e. location and rotation) is checked using a Matlab tool called RANSAC, also known as random sample consensus. Finally the piece is overlaid on the template image to see if the piece is in the correct location. This algorithm heavily depended on the image of the puzzle. They were able to produce a 91.7% detection success rate for one 24 piece puzzle, but for different 24 piece puzzles that had a cartoon image with large lines and large chunks of color they only achieved a success of 13/24 pieces (Liang & Liu, n.d.).

Overall, the most promising solver was Cris Zanoci and Jim Andress's- one that used solely image data. This was able to produce successful solutions at high piece counts. The solution by Liang and Liu seemed to produce somewhat promising results and had a different approach to others, using feature detection via SURF. Algorithms that were based on only edge data seemed to produce inconsistent results at much lower piece counts. The three main approaches- feature detection, image data, and edge data- were all potentially beneficial for detecting piece matches and were worth investigating more as we began developing our own algorithmic approaches.

3. Methods

Overview

The goal of this project was to design an application to help users solve jigsaw puzzles through three different matching approaches: edges, features, and color. The methods to achieve this goal are as followed:

1. Apply techniques for splines with subsequence matching algorithms to match one piece with another based on their respective shapes.
2. Match puzzle pieces based on their appearance or texture by utilizing SIFT (Scale-Invariant Feature Transform).
3. Implement color histogram correlation matching by splitting the RGB channels of puzzle pieces and sections of the puzzle image.
4. Create a suitable user interface (UI) for the application which follows a logical workflow.

This chapter details the development process for each puzzle matching approach and the user interface, as well as the procedures for testing the application accuracy.

Splines and Edge Detection

One of the approaches the team took in matching puzzle pieces together is through the combination of splines and edge detection. For initial edge detection, we used Canny edge detection through Python's OpenCV library. We determined that it returned more quality results to that of Sobel edge detection. Canny edge detection produced more and smoother edges because it used the non-maximum suppression and hysteresis steps which Sobel edge detection does not have. As stated in the background, a spline is a mathematical representation that follows a sequence of points. Combining edge detection with this idea of splines, we were able to

represent the individual splines surrounding each puzzle piece by using the points found via edge detection.

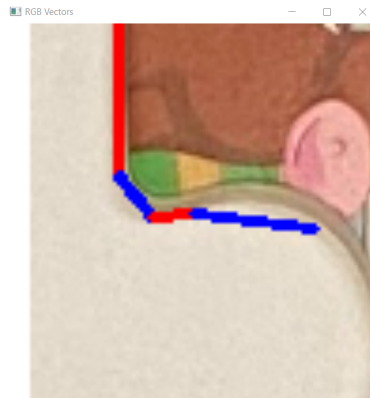


Figure 14: Example of a puzzle piece spline

Representation

To begin using the spline and edge detection approach, getting the spline representation of each individual jigsaw puzzle piece was the first step. To do this, an image of a puzzle piece needs to be read into the program which was done through OpenCV. That image is then converted to grayscale. This conversion of the image to grayscale allows the edge detection to pick up on the edges more easily as it detects the intensity of the pixels which is then used to calculate a gradient threshold. This threshold is then used to determine whether the pixels belong to an edge of the image or not. After the conversion of the image to grayscale, the image is masked in white while keeping the background black which helps even further when doing edge detection on the image. Most importantly, the mask makes sure that few edges will be detected within the piece itself. Canny edge detection is then performed on the masked image, giving the outline of the puzzle piece and ignoring the contents or inside of the piece. The resulting image then becomes the basis of the spline of the puzzle piece.

From the resulting image that displays the edges of the puzzle piece, the next step is to find the points that form the spline of the puzzle piece. This is done through the OpenCV library in Python as it provides a function that finds the contours of an image. The parameters of this function consists of an image, the

contour retrieval mode, and the contour approximation method. The contour retrieval mode parameter tells the function what mode to get the contours of the image. In this project, the mode that was used was RETR_EXTERNAL which gets the extreme outer contours of the image which in this case is the outline of the puzzle piece. The other parameter, the contour approximation method, is getting the approximation of the contours when looking at the image and for this approach, CHAIN_APPROX_SIMPLE was used to get the contour approximation.

After finding the contours of a puzzle piece, the next step was calculating the angle at every contour point to get the cumulative angle which was used to do the spline matching. By using the dot product operation, it allowed the project team to calculate the angle at every contour point by evaluating the vectors that were produced from the given points. After getting the angle values at every contour point and normalizing it, the project team calculated the cumulative angle over a specified arc length (currently is set to 96 but is adjustable), which was used to do the spline matching between puzzle pieces and declaring if they were matches or not. Figure 15 shows a flow chart of the previously described procedure.

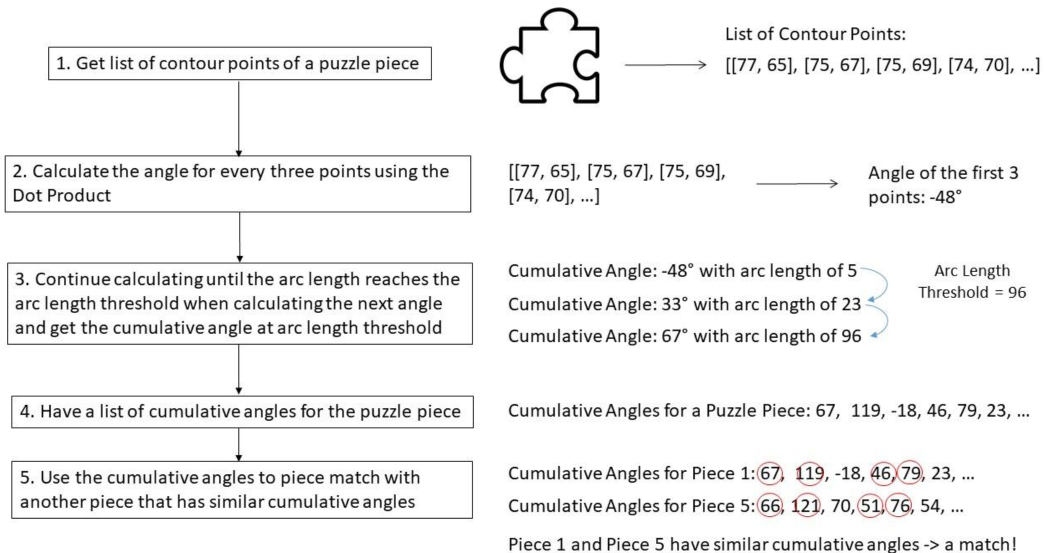


Figure 15: The procedure of using splines to do piece matching

Visualization

When visualizing the splines on a puzzle piece, the factors that affect how the spline is computed are the kernel size, the morphological operator, and the parameters of the Canny Edge Detection function. After converting the image of the puzzle piece into grayscale, the next step was to fill in the image with white so that the contents of the puzzle would not show up when masking the image to get the edges. Both the kernel size and the closing morphological operation helps remove false positives, or other pixels in the contents of the image that are not white, that are still present in the image and fill it with white pixels creating a mask of the puzzle piece. Adjusting the kernel size affects the noise of the image as the bigger the hyperparameter gets, the more the contents of the image would be filled with white pixels. This causes parts of the puzzle piece to become part of the mask (i.e., the notches of the piece). Once a suitable kernel size was established, Canny Edge Detection is applied to the image as it takes in the image, a minimum, and a maximum as parameters and with this, detects which pixels in the image are a “sure-edge”. The resulting image from applying the edge detection gives the edges or the outline of the puzzle piece which is used for the spline representation.

After getting the spline representation of the puzzle piece, the next task was to determine the direction of the spline when it is being traversed and the associated angle. From the previous section, the dot product was used to find the cumulative angle over a specified arc length. A challenge that first occurred when getting the cumulative angle was how to normalize the angles. Without normalizing the angles, the resulting cumulative angle that was calculated at the specified arc length was a large positive value which does not give a lot of information or purpose when performing spline matching. By taking the first vector that was calculated and flattening it as the x-axis for the second vector, the angle given from the second vector determines the sign and the direction the spline. For example, if the second vector is above the x-axis, it gives a negative angle and the spline is making a left turn. Each turn made by the spline is shown in a different color: red signifies that

the angle is positive and the spline is making a right turn while blue signifies that the angle is negative and the spline is making a left turn (See Figure 14). This normalization of the angles allowed the cumulative angle that was calculated at the specified arc length to be more accurate and meaningful when doing the spline matching.

Subsequence Matching

After obtaining the splines and the visualization of them, to identify which puzzle perfectly matches with another puzzle piece, subsequence matching was used to take the cumulative angles of two pieces and check if there is a common sequence of angles when comparing them with a given angle threshold and a subsequence threshold. The angle threshold determines if the angles being compared is common between the two puzzle pieces. The subsequence threshold is used to determine if the number of common angles reaches to that threshold or greater, then it is considered a match. As an example, if the list of cumulative angles for the first puzzle piece is [300, 198, 209, 98, 100] and the list of cumulative angles for the second puzzle piece is [300, 200, 198, 56, 98, 102] with an angle threshold of 2 and a sequence threshold of 4, when comparing the two pieces by their list of cumulative angles, then the two pieces are considered a match since they have four common angles.

However, when running the program with subsequence matching on real-life puzzle pieces, some of the cumulative angles were either opposites of each other or were going in the reverse direction. For example, one puzzle piece has cumulative angles [179, -91, 67, 203, -15] and the other puzzle piece has cumulative angles [-14, -199, 68, 90, 177]. From this example, the angles of the second piece are going in the reverse direction and the signs are sometimes switched. This was observed to be happening to some of the puzzle pieces that were supposed to match to each other. To reduce this occurrence, we added a way to check if the list of cumulative angles is matching in one direction or in the reverse direction to ensure that potential matches were not lost when doing subsequence matching. Checking if the angles

were opposites of each other or not was also implemented to have less matches lost when running the program (e.g., 15 and -15 is considered a match).

After revising the subsequence matching technique, when running the program with an image of the puzzle pieces scattered evenly and segmenting each piece, it produced a variety of results that were dependent on the values that were set for the angle and subsequence thresholds. When increasing the angle threshold and keeping the subsequence threshold constant, the number of matches increased as well however it is not helpful data as it matches puzzle pieces that are not supposed to match together. When tightening the subsequence threshold and keeping the angle threshold constant, it reduced the total number of matches found as the common sequences found from two puzzle pieces must reach that threshold to be considered a match. From observing the changes in the results, the values of the two thresholds were set to a standard that produces a reasonable amount of matches. After getting the matches for each puzzle piece, each puzzle piece was ranked by the number of matches that were found to know which puzzle piece matches the other more than another piece. For example, if puzzle piece 0 matches with puzzle piece 11 with a match count of 5 and puzzle piece 0 matches with puzzle piece 5 with a match count of 2, when comparing the two pairs, puzzle piece 0 is most likely a potential match with puzzle piece 11 than if it were to match with puzzle piece 5. Figure 16 shows an example of the results with a 12-piece puzzle ranked by the number of matches.

```
Pieces 0 and 4 are a potential match with  11 matches
Pieces 0 and 11 are a potential match with  9 matches
Pieces 0 and 3 are a potential match with  8 matches
Pieces 0 and 7 are a potential match with  7 matches
Pieces 0 and 9 are a potential match with  7 matches
Pieces 0 and 1 are a potential match with  6 matches
Pieces 0 and 2 are a potential match with  6 matches
Pieces 0 and 5 are a potential match with  6 matches
Pieces 0 and 8 are a potential match with  6 matches
Pieces 1 and 2 are a potential match with  6 matches
Pieces 1 and 3 are a potential match with  6 matches
Pieces 1 and 4 are a potential match with  6 matches
Pieces 1 and 5 are a potential match with  6 matches
Pieces 1 and 6 are a potential match with  6 matches
Pieces 1 and 11 are a potential match with  6 matches
Pieces 2 and 4 are a potential match with  6 matches
Pieces 2 and 6 are a potential match with  6 matches
Pieces 2 and 9 are a potential match with  6 matches
Pieces 3 and 4 are a potential match with  9 matches
Pieces 3 and 9 are a potential match with  8 matches
Pieces 3 and 10 are a potential match with  7 matches
```

Figure 16a: Ranked matches between puzzle pieces

Figure 16b: Example of a pair of pieces matching

To test this matching, we ran our edge matching algorithm on the 12 piece wooden farm puzzle. In order to check if this algorithm performs any better than random matching we conducted a simulation, running 30 iterations of randomly matching the pieces to each other. We chose to make 27 random matches since this was the number of matches made by the edge detection.

To check for any bugs in the matching algorithm we set up two tests with artificial "perfect" pieces to check if they came up as a match. Since the matching algorithm had two parts, the cumulative angles and then subsequence matching we ran tests for both. To see if the subsequence matching worked, we constructed two arrays of perfect angle matches. Then we tested two series of points along an X,Y axis that would produce the same cumulative angles and run matching on those. This was to ensure that there was no error in the matching portion of our algorithm.

SIFT

As an alternative approach to matching puzzle pieces based on their edges, the internal features of each individual piece can be independently compared with those of the completed puzzle. By matching these features, the piece image can be mapped to its proper location and orientation within the puzzle image. Python's OpenCV library provided many tools for this process, including the SIFT algorithm for locating keypoints and corresponding descriptors. While SIFT formed the basis of the system's functionality, there were a number of necessary steps in between to process the user input and resulting matches. See Figure 17 below for a flowchart

depicting this process.

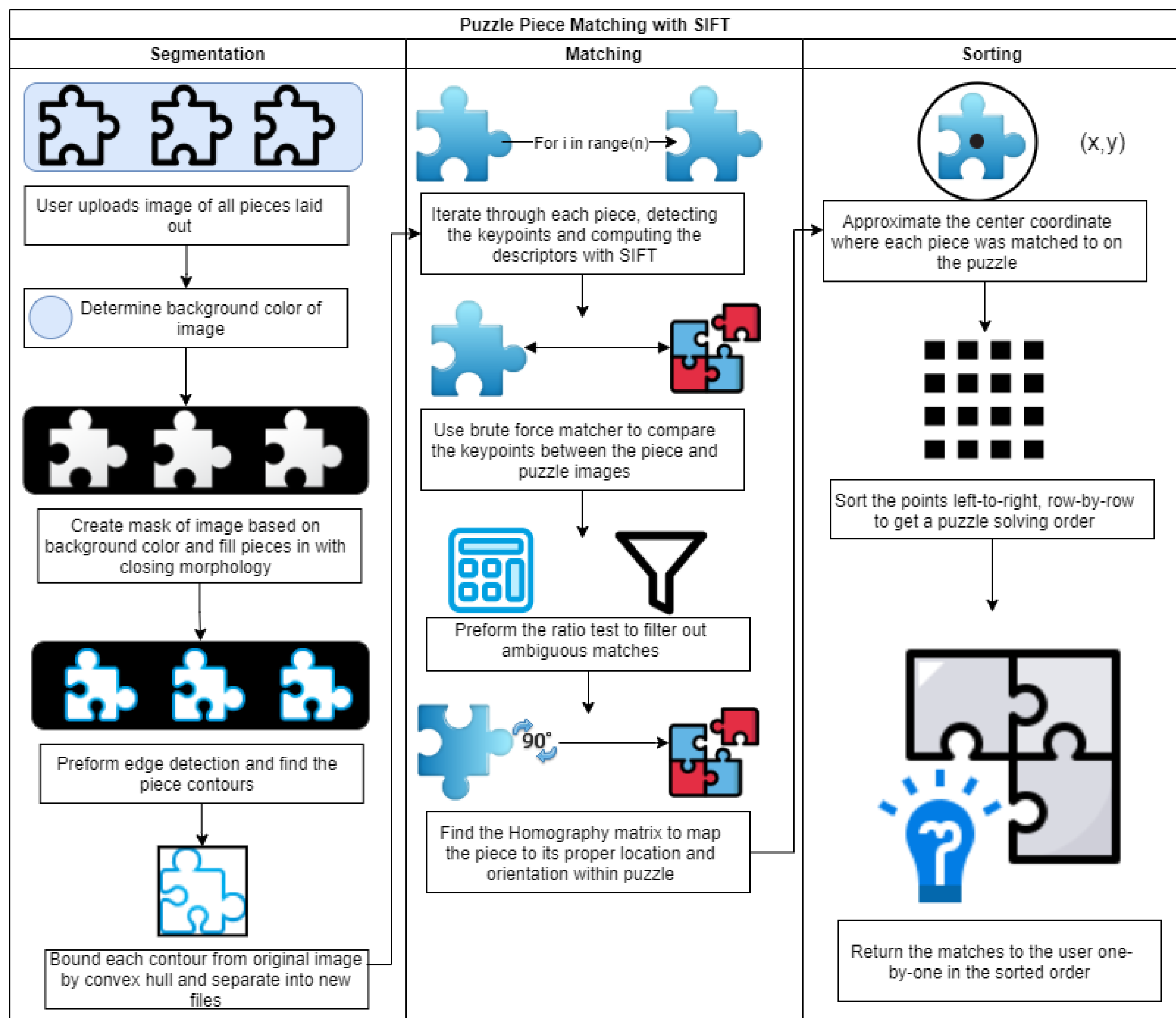


Figure 17: Flowchart for the SIFT puzzle piece matching process, including segmentation, matching, and sorting

Segmentation

In order to analyze the features of each puzzle piece, they first need to be separated out. While a user could upload individual images of each puzzle piece, this is both inconvenient and impractical for larger puzzles. Instead, an image that contains all of the pieces spread out can be processed to automatically separate the pieces based on their edges. For best results, users are asked to lay the pieces out on

a solid colored background with proper lighting and quality so the features can be extracted precisely.

Piece segmentation relies on having an image with a solid colored background in order to decipher the piece pixels from the background. Once the image of pieces is inputted, the background color is taken as the most common color value (blue, green, red) along the perimeter. While this approach carries an assumption about the image, it was just as effective yet more lightweight than having to find the most common value throughout the entire image.

After determining the background color, a binary mask of the image is created to turn the pixel values into strictly black or white. Pixels within an adjustable ± range of the background color would be considered the image background, with everything outside the range as the pieces. It is expected that some pieces have areas that match the background color, so a morphology closing operation is performed to enclose the piece in solid white. This operation is essentially a combination of dilation followed by erosion, which helps to fill in small gaps in objects. Depending on the spacing of the pieces, the morphology kernel size may need to be adjusted by the user. If the pieces are closer together, a smaller kernel size would be needed to ensure that the closing operation does not connect two pieces. A sample puzzle image with the resulting mask can be seen in Figure 18.

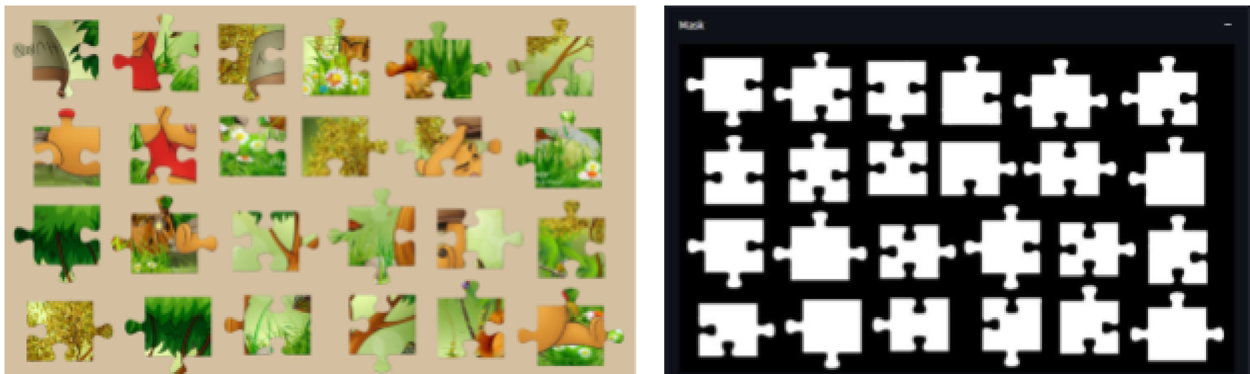


Figure 18: Sample puzzle image and corresponding mask

Once the solid puzzle pieces are distinguished from the background, Canny Edge Detection is applied to extract only the piece edges. The reason we could not immediately perform edge detection on the input image is because the internal features of a piece would also have been captured; creating the mask image first helped eliminate the noise from piece features. With all of the piece contours identified, the last remaining step is to bound each one separately and create new files for them. While a simple bounding rectangle or circle could do the job, edges of other pieces would often overlap this boundary depending on how far apart the pieces were spread out in the image. To avoid this issue, a convex hull boundary was used instead on the contours (which are concave) to bound them more tightly.

Matching

The next major step in the puzzle piece matching process uses SIFT to find the main areas of interest in each piece, and compare them with features of the whole puzzle. OpenCV provides a convenient method for creating a SIFT detector, which can detect and compute the keypoints and descriptors of an image. Having segmented all the pieces into separate files, the resulting descriptors of each piece can be matched to the puzzle descriptors one at a time by iterating through a loop.

A brute force matcher is used to find the puzzle feature with the minimum Euclidean distance to each piece feature. However, this approach often leaves an abundance of ambiguous and false matches which dilutes the set of precise matches. To mitigate this problem, for every keypoint in one image (the piece), the two closest matches are calculated and Lowe's Ratio Test is performed. If the two best matches calculated are too close together (within the 0.75 threshold), then the match is discarded entirely; the closer match would be deemed too ambiguous to confidently say it is correct when the second best option is in its vicinity.

If at least four "good" matches are identified between the piece and the puzzle after filtering through the ratio test (although more is preferred), then a homography can be calculated to transform the piece to its location and orientation within the puzzle. This homography function takes RANSAC as an approximation

method parameter to find the transformation which contains the most inliers matches. By extracting the piece contour in a similar fashion as the segmentation process, the contour can be projected onto a copy of the puzzle image based on the homography transformation. An arrow points from the piece's original location in the image taken by the user to the matching location found on the puzzle to provide a clear visual cue for the user. A puzzle image copy is created for each piece to show the respective matches individually, so the user can follow along one step at a time. However, before outputting the results to the user, the matches need to be in a presentable format with a sequential ordering. An example image showing a piece match with the raw keypoint matching lines can be seen in Figure 19.

Figure 19: Puzzle piece match displayed by highlighting the location of the piece from the user inputted image and the matching location on the puzzle. The matching keypoints contributing to this match are connected with lines

Sorting

With the “absolute” SIFT approach to matching, each piece was individually compared to the puzzle with no knowledge of where the other pieces were matched to. This could introduce trouble for the user when solving the puzzle if the matches relative to the puzzle image are given in a random order, since there would be no guarantee that consecutive matches would directly connect pieces together; the user would have to approximate the piece positions on the surface they are solving the puzzle. To make the puzzle solving more realistic, the matches would be presented in a consecutive order allowing pieces to be connected every step. The ordering chosen was to sort the matches starting in the top left corner of the puzzle, and going across left to right, row by row, ending at the bottom right corner of the puzzle as shown in Figure 20.

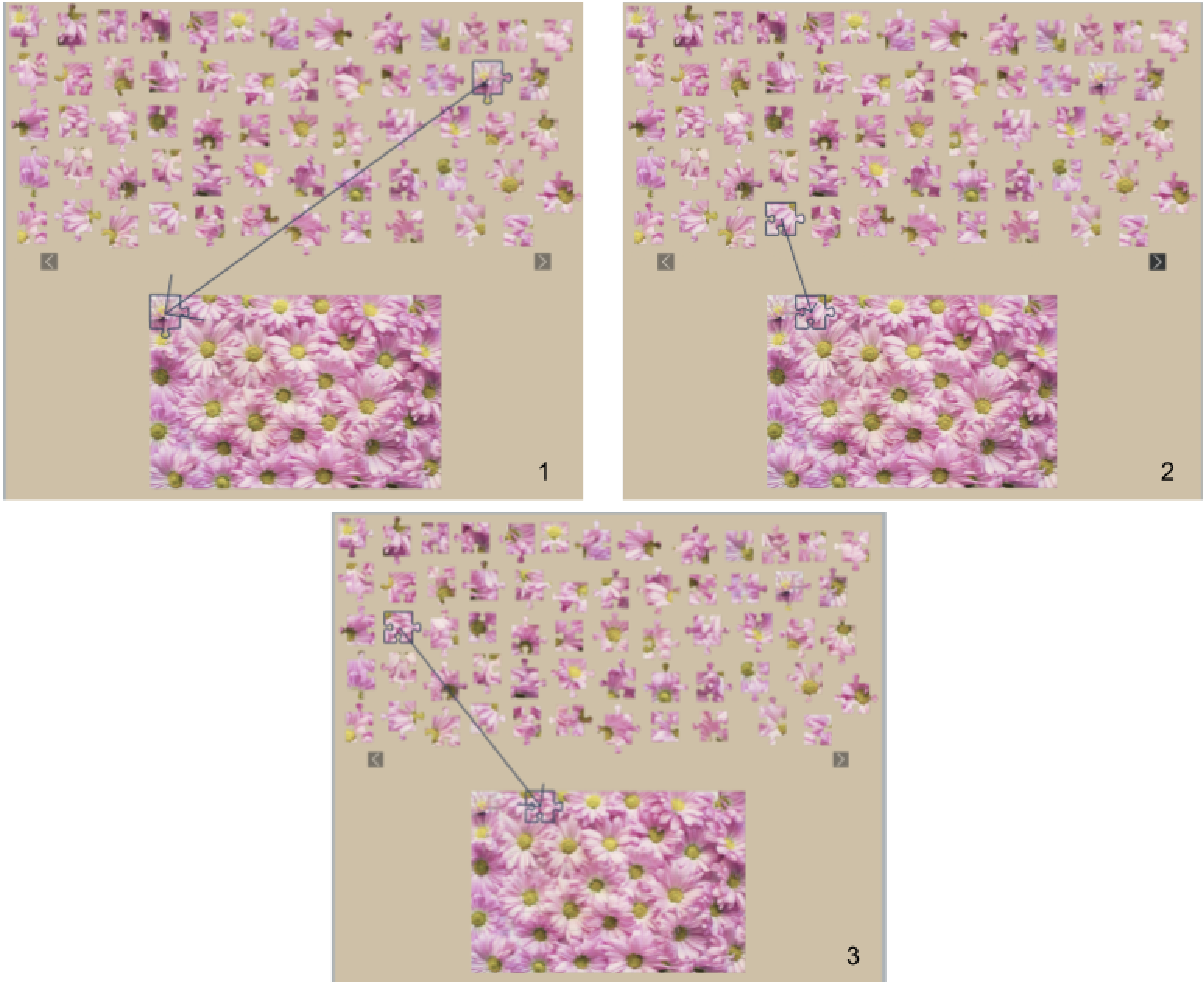


Figure 20: A sequence of three consecutive matches, starting from the top left of the puzzle

To generate a comparable value, points were taken from each projected contour on the puzzle image after the homography transformation. These points were defined as the center of the minimum enclosing circle of the contours. However, since the projected contours were merely approximations, each row of

points had variations in their x and y values; the collection of points reassembled an unaligned grid (Figure 21). A sorting algorithm was created to handle the misalignment of points to ultimately output the ordering from the top left of the puzzle to bottom right.

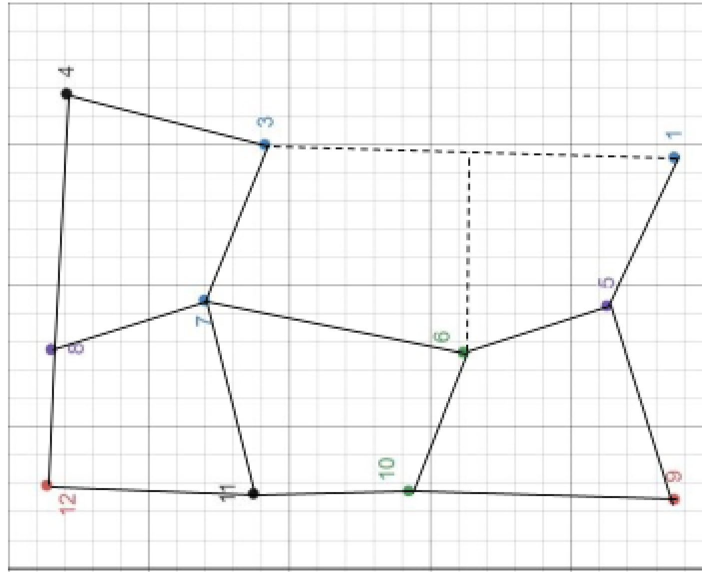


Figure 21: Example of a misaligned grid of points representing the center coordinate of each matching piece (Piece 2 was unmatched)

All the points were first sorted based on y value, which would group together the points within the same row, but would not guarantee correct ordering going across the row. Next, the difference in y values between each consecutive point was analyzed to find where one row breaks to another. While there may be some variance in y value between points in the same row, the start of a new row could be identified by a large change. Because the jump to a new row is much greater than the variance between points in the same row, any point greater than the average difference in y value between points was designated as a new row. While there may have been more robust methods of determining a threshold, this method was chosen due to its simplicity and effectiveness. Each row could not be assumed to have the same number of points, as pieces that did not have enough keypoints or could not be accurately matched were excluded from the ordering. Once each row of points was

identified, they were sorted individually by x value and recombined afterwards, giving the final sorted ordering for the user to consecutively match pieces together in a systematic fashion.

Limitations

One glaring limitation with SIFT matching is the dependence on having a distinct puzzle image, as only the internal features of puzzle pieces are considered without any regard to the edges. Many real puzzles created to be intentionally difficult are ones that have a low amount of distinguishing features on the pieces. Usually a person solving a puzzle would need to manually try connecting pieces that look similar to see if the edges fit together perfectly. As an extreme example, using SIFT on a monochromatic puzzle with no distinguishing features would be futile.

Color Histograms

Another approach to piece-to-puzzle matching is the use of color histograms. The general idea is to compare the ratios of the present colors between each piece and each spot on the full puzzle image. The higher the correlation between a piece and a spot, the more likely it is to be a match. In theory, this approach would excel in cases where the puzzle pieces resemble uniform colors without many features. This is because all features of the puzzle piece are lost when extracting only the RGB color concentrations. Introducing color histograms with SIFT would be complementary as SIFT matching relies on an abundance of features and is ineffective for pieces with blank space.

There is no way to already know which pieces go where before matching, so some approximations do need to be made for generating the “spots” on the full puzzle. To do this, a grid is first formed atop the puzzle (an example is shown in Figure 22), splitting the full puzzle image into equally sized rectangles- the same amount as the number of pieces. This grid is formed using the puzzle dimensions, which is something that must be hard-coded, provided, or dynamically determined.

In this case, hard-coded values provided the most consistent way to test this method and were used throughout the process. Once the grid is formed, we have access to an array of grid locations, which each correspond to a potential piece match location. This is obviously a rough estimate of a match location since each puzzle piece has unique cuts, holes, and extensions that don't lock it into a perfect rectangle. Nevertheless, these work well enough to generate some match results.

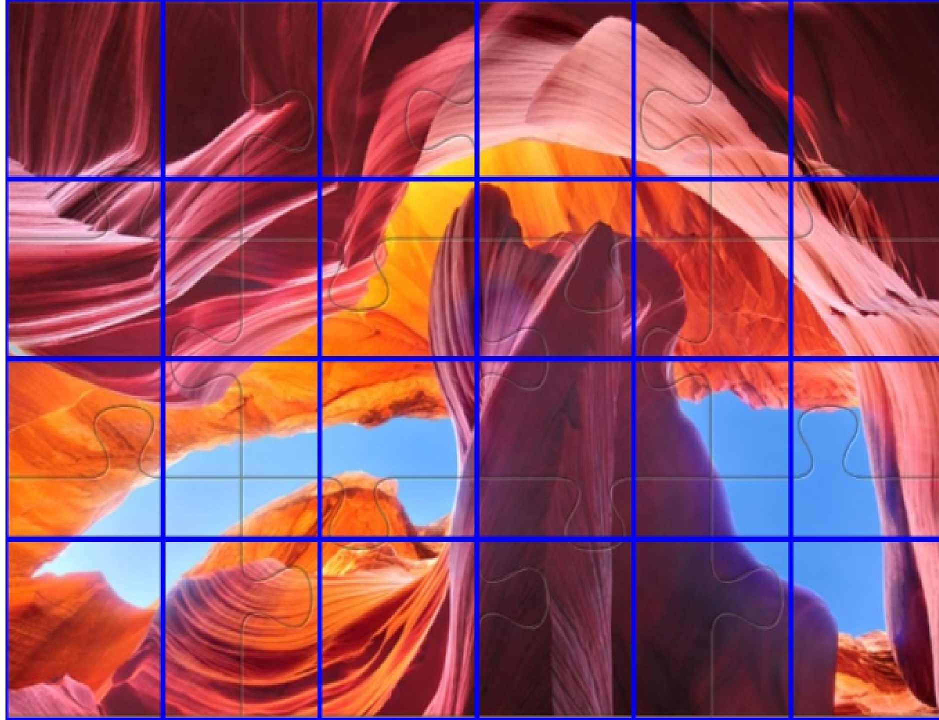


Figure 22: Example of a puzzle "grid" where it is divided into equal rectangles based on the number of pieces in each row and column

Next, using OpenCV, color histograms are generated for each piece and each grid location. For each piece and grid, three histograms are created- one for each of the color channels (Red, Green, and Blue). Each individual histogram is then normalized. This is done because in order to match pieces, we are more interested in the ratios of colors, not the raw amount of colors that a histogram naturally generates. In other words, for each color value, we rather know what *percent* of the piece has that color, not the amount of pixels that have that color.

With the normalized histograms for each piece and grid location, comparisons must then be made. Each piece is compared to each grid location. The comparison function used is OpenCV's compareHist() and the type of comparison used is correlation. The equation used will return a float between -1.0 and 1.0, where 1.0 would be returned for two identical histograms and -1.0 would be returned for two completely opposite ones. The function used in the backend is as follows:

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H}_1)(H_2(I) - \bar{H}_2)}{\sqrt{\sum_I (H_1(I) - \bar{H}_1)^2 \sum_I (H_2(I) - \bar{H}_2)^2}}$$

where N is the number of histogram bins, and for a given histogram H_k (k = 1 or 2):

$$\bar{H}_k = \frac{1}{N} \sum_J H_k(J)$$

Figure 23: OpenCV's compareHist() correlation comparison equation

The red, green, and blue channels of the piece and grid location are compared separately and are added together for a final correlation value that will be between -3.0 and 3.0. Once all comparisons have been completed, the grid location with the maximum correlation value is treated as a match.

Using color information to solve puzzles would not provide compelling results in isolation, as although it may help find the general area for a piece it would be difficult to pinpoint the precise location among adjacent pieces that have similar concentrations of color. However, the major upside with the color histogram approach is how there is always a maximum color correlation value between the pieces and puzzle grid to declare as the best match; SIFT would be incapable of producing any attempt at a complete match if there not enough keypoints were detected (a minimum of four is required for a homography transformation). By process of elimination, considering only the puzzle locations not matched by SIFT as a starting point for other methods improves matching odds in any case. While color

histograms were not the primary approach for matching, the hope was to fill in any gaps from SIFT and guide the user towards a complete puzzle solution.

UI

Workflow

When envisioning how the AI application could help users solve puzzles, two different approaches came to mind with varying degrees of involvement. On one extreme, the app would perform all of the work in matching the puzzle together, leaving the user with no control over the process. Because of the limitations with image analysis and matching for certain puzzles, we could not put total faith in the system to always report correct matches. Instead, the application's intention would be acting as a guidance tool to aid the user in completing the puzzle. A user should not assume the claimed matches will be correct 100% of the time, as ultimately only someone with the physical pieces can confirm if they match together. Additionally, there will often be puzzles with little to no matches found, in which case the user may have to rely less on the system to complete the puzzle.

Our workflow design for the user interface needed to be both functional and intuitive to use. The general use case path is outlined below:[1]

1. The user will first be presented with two file upload buttons for the necessary images of the puzzle they want solved. These images consist of an overhead of the disassembled puzzle on a solid color background, and one for the completed image usually provided on the puzzle box.
2. Once uploaded, the user will be prompted to submit the number of pieces the puzzle contains. This will help the matching algorithms confirm the correct number of piece contours are extracted.
3. A mask image of the disassembled puzzle will be displayed to the user to confirm that the pieces are accurately separated from the background. Two

[1] This user interface illustrating this workflow can be found in Appendix A.

adjustable sliders will be provided for the background color threshold and the morphology kernel size with default values. Depending on the image and spacing of the pieces, these parameters may need to be changed by the user to mask the pieces accurately. When either slider is changed, the mask image can be refreshed by clicking “check mask”, to give the user visual feedback on how the sliders affect the mask image. Once the user is satisfied with the masked image, a button to “submit” will initiate the segmentation and matching program.

4. A new image will display combining the user’s input images vertically with the disassembled pieces on top and the completed puzzle on the bottom. One piece from the top portion will be outlined, with its corresponding destination to the completed puzzle also outlined. An arrow will be displayed connecting these locations to give a clear cue for the user as to where the piece should go.
5. Using the interface as a guide, the user will try placing their puzzle piece in the location recommended by the program to confirm the accuracy.
6. The user may continue to the next step in the puzzle order (going top left to bottom right) by clicking the right arrow on the image carousel, and left to go back one.
7. The user will repeat this process until all the matches are reviewed.

Rather than computing matches piece by piece on the fly, our system computes matches for the entire puzzle initially to speed up navigation between steps. Another design choice for the system was to not provide the user with freedom to select pieces to match, where a predetermined order was used instead. As the SIFT algorithm matches pieces based on the complete puzzle picture, there would be no guarantee that user-selected pieces would link together; pieces placed in relation to the puzzle image would form gaps if done in a random ordering.

Depending on the image, SIFT may find conflicting or a lack of keypoints to match the piece. In this case, our system will move these pieces to the end of the ordering and will be run through color histogram matching. The hope is that if

there were only a few pieces that could not be matched, the majority of the puzzle would already be solved. With any remaining puzzle gaps, the user should have an easier time figuring out for themselves where the unmatched pieces should go with help from the histogram matching. However, there would certainly be cases where there are too many unmatched pieces, in which case our system is limited in how much assistance it can provide the user.

Streamlit

The Streamlit framework was used to create the initial prototype for our user interface (UI). Launched in 2019, this open-source python framework is primarily intended for data science and machine learning web applications. While this was not our area of focus, Streamlit was chosen because of its simplistic widgets, abundant and clear documentation, modern looking design, and easy backend integration.

Behind the scenes, Streamlit reruns the python script from top to bottom after every change in state or when the screen needs to update. This concept facilitated part of the workflow and removed duplicate code. For example, when the image masking slider is adjusted by the user, the screen automatically refreshes with a recomputed image based on the new parameters. Additionally, when clicking the “next” or “back” buttons to navigate through the matching steps, the new step will be displayed by simply incrementing the state of a counter variable; the change in state will trigger a rerun of the script, and the code segment that displays the image will use the updated variable value instead.

The Streamlit concept of rerunning the script upon any change also introduced a few inconveniences to workaround. Normal variables would be reset to their initial values unless a session state variable was used. Extra session variables were also required to track when things were run for the first time in order to avoid rerunning expensive operations like SIFT. Submit buttons that were intended to only be clicked once throughout the duration of the app would be reset after a page refresh, causing any subsequent logic to be unreachable without the use of

additional variables to track progress. The major downside of using Streamlit for displaying images was how the screen kept refreshing when going to the next step. This was disorienting as the image would disappear and be replaced by a new one, without a smooth transition. Because of these limitations, a different approach with Flask was later pursued to allow for more flexibility.[2]

Flask and Alternatives

We looked into multiple different Python UI frameworks when deciding on how to display our results. Most simple frameworks like PyGUI and PySimpleGUI provided nearly identical potential to Streamlit. They all worked with widgets to help users build quick applications without having to start from scratch. Since we had already run our initial tests in Streamlit, we decided that there was no need to change to a different widget-based framework as they all came with similar limitations.

Contrary to widget based frameworks, Flask provided a flexible alternative to integrate the backend Python code with frontend HTML and CSS. This provided the most amount of flexibility for the app design, but it was also more involved. Using Flask required defining routes for POST requests coming from HTML forms, and handling the redirects between pages. In addition, a lot more styling was required to make the HTML look appealing. To mitigate the issues of having a page refresh in between steps, a carousel was created with Bootstrap to seamlessly transition between the images. While it may have taken longer to implement initially, using Flask was beneficial in the long run because of the increased possibilities and control over the design.

Experimentation

In designing both the backend and frontend of the application, only a select few puzzles were used as sample input files. While these few 12 and 24 piece

---

[2] Screenshots of the Streamlit design can be found in Appendix B.

puzzles showcased a near perfect level of matching, this sample size was inadequate to investigate the accuracy of the system. Therefore, a testing framework was designed in order to consistently evaluate the system with a larger sample of puzzle images and sizes. Quantitative data would be collected for the number of correct pieces matched for a given set of images to allow for an objective comparison of puzzles through statistical analysis. The results of this experimentation would help uncover key properties about puzzles which affect the success of our system, and point out areas for improvement.

Constraints

The quality of an image played an important role in our puzzle solver. By following certain constraints when taking images of a puzzle, we were able to produce better results from our program.

To begin, our program requires an image of the completed puzzle, which can be provided by taking an image from the puzzle box. The user must also take a top-down image of the scattered pieces. The pieces must not be touching any other pieces and be placed on a solid color background, whose color is not common in the puzzle. The user is also recommended to crop the images to remove any portion of the picture that is not the solid color background or pieces. It is also recommended to use a dark color background rather than a light one. Users should also attempt to take pictures with minimal glare.

These constraints were derived after experimenting with different scenarios. The first test was done on a white background, with a few of the pieces touching. The masking missed a few pieces due to the light glare and the white background, along with the touching pieces segmenting into one piece. Another test was conducted on the white background and no pieces touching. No mask was able to be created from this image. The third test was conducted on a black background and a mask was able to be made, clearly distinguishing all the pieces from each other. All tests were conducted under the same lighting conditions.

Procedure

For the testing process, we selected a sample size of 50 puzzles; each project team-member tested ten puzzles from an online puzzle simulator. We chose to test on an online simulator for the abundance of available puzzles and to prevent factors like glare and poor lighting. For each puzzle, we conducted the test on piece numbers of 24, 60, and 96 to judge the performance of the puzzle solver as the pieces contain less and less of a puzzle's image.

The team collectively selected the puzzles based on guiding categories to ensure a diverse set was tested. The categories included puzzles that were anticipated to have a higher matching rate (high color variation with many distinguishing features), ones that would perform decently well (high color variation with not many distinguishing features and low color variation with many distinguishing features), and others that were expected to have a low matching rate (low color variation with not many distinguishing features). Other puzzles that were more unpredictable were chosen, such as ones with repeating patterns. By comparing the matching accuracy of different puzzles, it would help us to observe the characteristics which make an "ideal" puzzle and ones more difficult for the application.

To get the images from the online puzzle simulator to use in our program we followed this procedure:

1. Choose a puzzle from the selected sample
2. Spread out the pieces by dragging, making sure there is ample room in between pieces
3. Take a single screenshot of the puzzle pieces together as zoomed in as possible
4. Complete the puzzle and take another screenshot
5. Input the two images into the application and follow the steps entering piece count and mask

6. Record the number of claimed matches displayed out of the total number of pieces and visually inspect the arrow and contour locations to subtract out any incorrect matches to get a final number of “actual” matches.[3]

After these steps we observed the number of correct matches made to get an accuracy of our program on the puzzle. We then looked for similarities between high accuracy puzzles and the differences between puzzles that score a high accuracy vs a low accuracy. This also helps us determine the error of our program in order to understand how it can work better.

[3] The full data set collected can be found in Appendix C.

4. Results and Analysis

In this section we will discuss the results of our testing, the accuracy of the three methods explored for our puzzle solver, Edge detection, SIFT, and color histogram. We tested each method to get an accuracy to determine the usefulness of a particular strategy. This helped us decide on what to implement in our final product as well as determine next steps for any future development.

Edge Detection

Edge detection was the most unsuccessful matching strategy that we explored as it was not able to detect enough points along the edges of pieces. Reasons for this included the image passed in was not the right size to detect points (e.g., either too big or too small) or the real-world conditions affected the contents of the puzzle piece when taking a picture of it (e.g., too much lighting gave glare on the puzzle, the thickness of the piece, etc.). There were also instances where the points that were detected would go in one direction and then stop at a certain point and would loop backwards until it reaches the point that it previously stopped at due to the thickness of a piece causing double edges. From this example, some of the points detected were double counted and from our observation, this was most likely because of the thickness of the piece as shown in Figure 24.

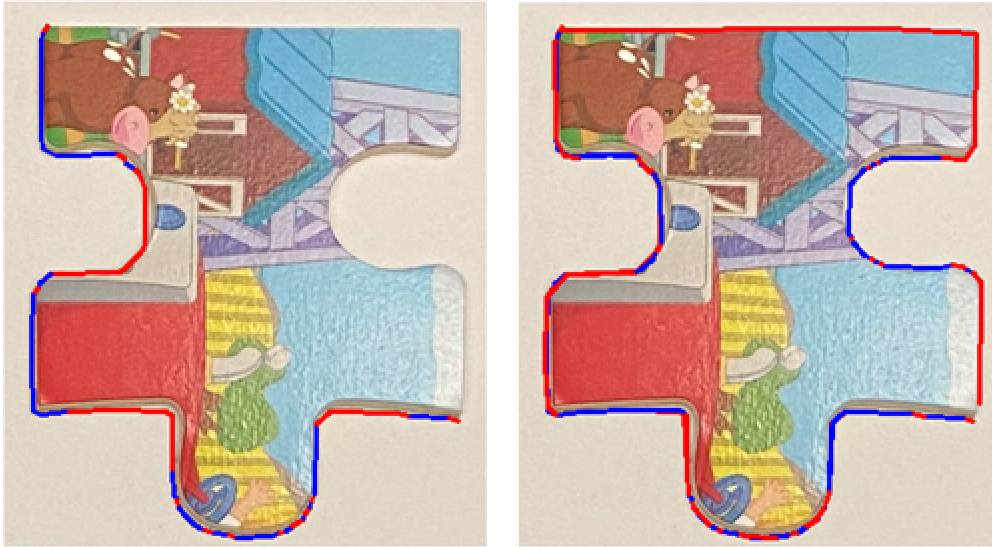


Figure [24.a]: Spline points traversing counter-clockwise order, stopped near the bottom-left

Figure [24.b]: Spline points traverse backwards from the stop point and end at the same point

From our testing of the random matches vs. our edge detection, random guesses made 6.93 matches on average, with a 25.6 average match percentage. Some trials performed better than our matching, getting 11 out of 27 matches, while others made 0 of 27 matches. When rounded to the nearest whole match, this random guessing performed identically to the spline matching. This led us to conclude that the spline matching we had implemented was not a sufficient strategy to find matches.

Testing the subsequence matching on two perfect sequences returned the expected true match. The test for calculating the cumulative angle of two perfect matching pieces then running subsequence matching also returned in a match. This led us to the conclusion that the detection of points along the edge was not sufficient enough to accurately represent the pieces by their angles.

We decided not to run this testing on the online puzzles we used for SIFT. This decision was made because the online puzzle pieces generated had the same

edges so even if we detected every point along a piece, it would come up with many false positives since the edge would match with numerous pieces who have the same shape.

Due to the low accuracy caused by an insubstantial amount of edge points detected, we decided to stop pursuing this approach to solving the puzzles. After testing we have no reason to believe that our current implementation of this method would produce any better results than random guessing, whose results were identical to our spline matching. We believe that if the spline detection had produced more data points along the edge of the pieces, then that would have been a more viable approach to solving puzzles. However, with an inadequate amount of points detected on the large 12 piece puzzle, the number of points would only decrease as the puzzle piece count increased, so we determined spline matching was not something we wanted to pursue further.

SIFT

Of the three main puzzle piece facets explored in this project -- edges, features, and colors, matching based on features with SIFT proved most effective. The average matching accuracy over the 50 puzzles tested can be seen in Figure 25, where the blue line represents matches from SIFT alone, with the red line including matches from color histograms in addition to SIFT. The average percentage of pieces matched with SIFT alone started at 91% with 24 piece puzzles and went down to 73.4% and 54% at 60 and 96 pieces respectively. The standard deviation for these averages increased as the puzzle size increased, starting from roughly 18% at 24 pieces going up to 26.6% at 96 pieces. These standard deviation values were expected as the matching accuracy was largely dependent on the puzzle image; the sample of 50 puzzles were hand selected by the team to guarantee a variety such that some were predicted to perform well and others not so well.

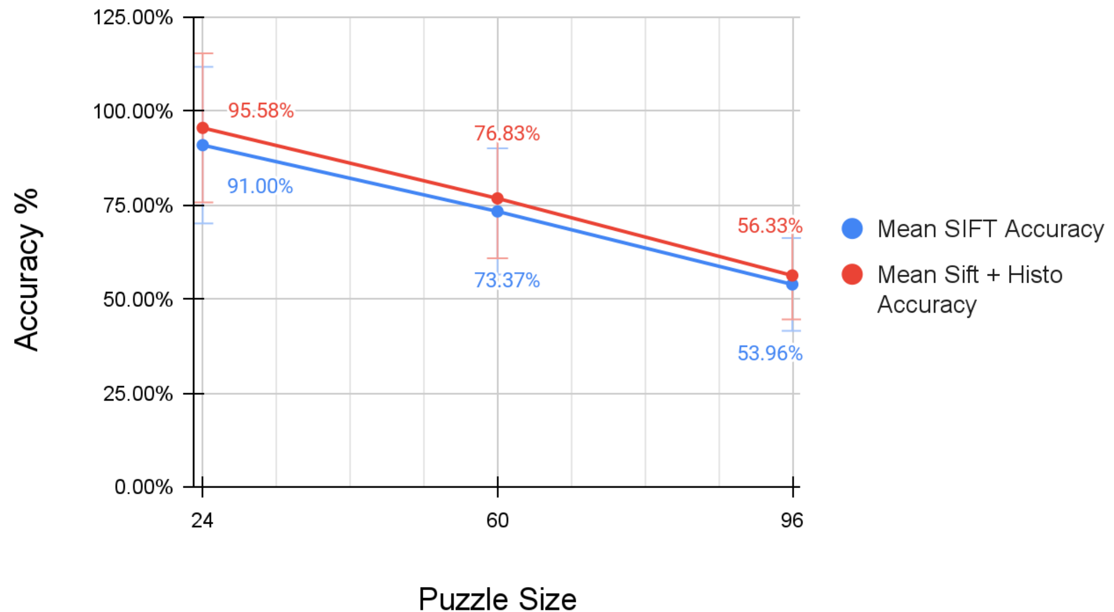


Figure 25: Line graph depicting the average matching accuracy for different puzzles sizes for SIFT and SIFT + Color Histogram methods

Puzzle Image Analysis

One evident finding from the collected data is how much the matching accuracy varied based on the puzzle image, ranging all the way from 0% to 100% at 96 pieces. This leads us to the question: what makes one puzzle yield a higher accuracy than another? We can speculate by examining the puzzle images which performed the best and worst and apply our knowledge of how SIFT works. The effectiveness of SIFT matching heavily depends on the number of available keypoints on an image, where more keypoint matches increases the odds for outputting a match for the piece. Therefore, while we can't perceive exactly what the keypoints are, we can broadly judge the intricacy of an image to help predict which puzzles will have the best matching accuracy.

Figure 26 shows the puzzle images in which SIFT matched no pieces for all puzzle sizes (left) and matched all pieces across all sizes (right). The right image has

hundreds of small flowers with all sorts of colors, extending to all four edges. The unique arrangement and variation of colors in the image along with how there is no particular spot that is missing features explains how sift was able to match everything through 96 pieces. The puzzle that SIFT did not match any pieces is a little more surprising as there is decent color variation which would suggest at least some keypoints were detected. The soft edges of the triangles as well as the gradual color gradient change between neighboring triangles likely led to little to no keypoints being detected.

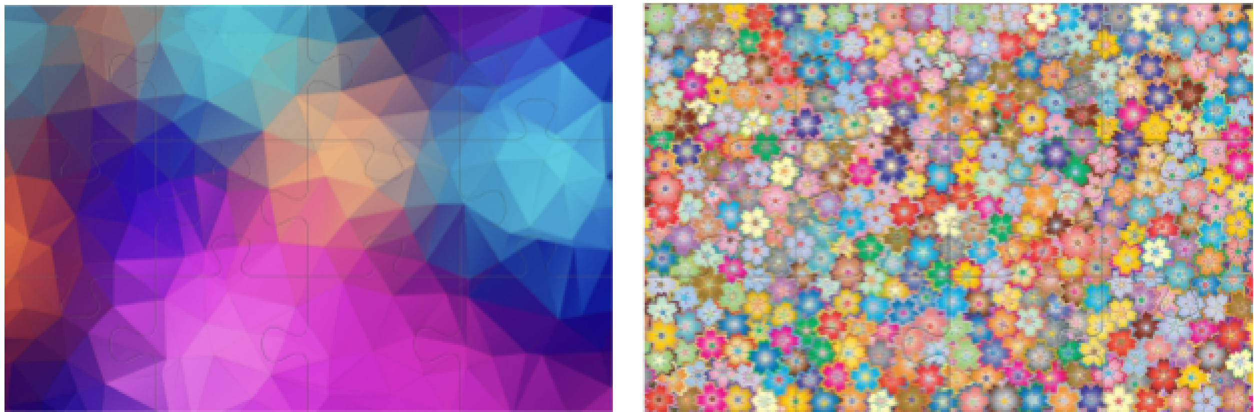


Figure 26: Puzzle images with the minimum and maximum SIFT matching accuracy

Another outcome to examine from the data is how the accuracy declines in a linear fashion for the same puzzle by increasing the puzzle size. Looking back at Figure 25, we observe this decrease in accuracy from the negative sloping lines. Naturally, as the number of pieces increases in a puzzle, the area of each piece diminishes. With less "content" on each piece, there are fewer keypoints to detect by SIFT, leading to a decrease in total matches. This average decline in matching, found by subtracting consecutive accuracy values, was 17.6% between 24 and 60 pieces (SIFT only) and 19.4% between 60 and 96 pieces. It was no mistake that the puzzle sizes chosen were equal increments of 36 from 24 to 60 and 60 to 96, because it would allow us to make a more equal comparison between the groups. The

roughly 2% difference in average decline between the two groups suggests mostly a linear decline.

To further analyze the accuracy decline with increasing puzzle size, Figure 27 shows data for every individual puzzle tested. The tops of the yellow, red, and blue bars in a single column represent the matching percentage at 24, 60, and 96 pieces for a single puzzle respectively. Things to notice about this graph in particular are the ranges of the yellow and red bars; the span of the yellow bar shows the decline in accuracy from 24 pieces to 60, and the span of the red bar shows the decline from 60 to 96 pieces. The puzzles are sorted in ascending order of the 96 piece accuracy percentage, which is represented by the blue bar.

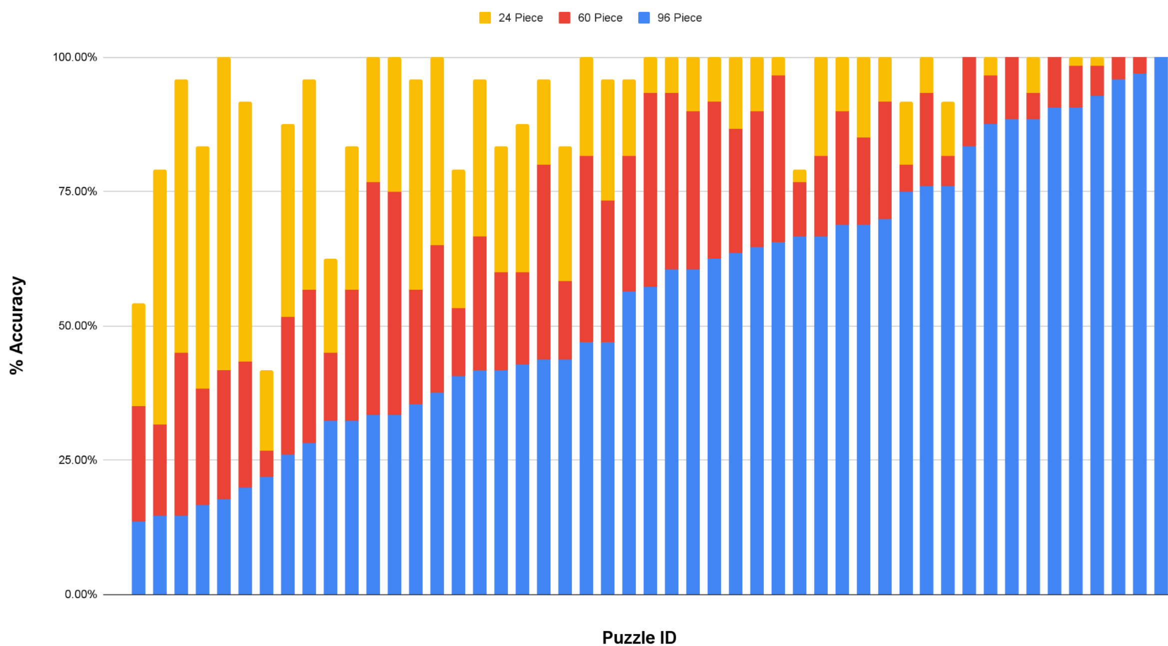


Figure 27: Stacked bar chart showing the SIFT matching accuracy dropoff for all puzzles tested when increasing puzzle size. The yellow bars show the decline in accuracy from 24 pieces to 60, and the red bar shows the decline from 60 to 96 pieces

There were a couple of outliers in the data from Figure 27, the first being the 7th bar over from the left, which has a total height much lower than the surrounding values. The actual image corresponding to this entry is shown in Figure 28 on the left side. This particular puzzle had a much lower initial accuracy of 10/24 pieces and had a smaller decline as the puzzle size was increased; the red bar portion from 60 to 96 pieces only saw a 4.8% decrease from 26.7% to 21.9%, compared to the average which was 19.4% for this category. By examining the puzzle image, one can notice the amount of empty sky that is occupying a large area of the puzzle. The pieces occupying that space would have very few if any keypoints as there are no distinguishing features. Therefore, even at 24 pieces the SIFT matching percentage would be quite low compared to other puzzles which do not have as much featureless space. As the puzzle size is increased beyond 24 pieces, the amount of pieces which represent the foreground and airplane will stay roughly proportional to those that represent the sky; Once a piece from a 24 piece puzzle does not find a match, than surely a smaller piece from the same area would also not find a match. This insight helps explain why there is a smaller decline in matching accuracy for puzzles which have a low initial matching percentage at 24 pieces.

Figure 28: Puzzle images with notable jumps in accuracy between puzzle sizes

In the opposite direction, one puzzle which had a large decline in accuracy is shown on the right side of Figure 28. This colorful umbrella picture had all 24/24

matches in the first set, but then only matched 25/60 when the size was increased, dropping a total of 58.3% between these two sizes compared to the average of 17.6% for this category across all puzzles. With the puzzle split into 24 pieces, there are many different colored umbrellas that are visible in each piece which would allow SIFT to find complementary features in the same area. However, as the piece area diminishes, fewer colored umbrellas will exist on the same piece. For this specific image, we speculate the pronounced decrease in accuracy compared to other puzzles is due to the repetition of colors and orientation of the umbrellas. For example, upwards of eight pink umbrellas are present in this image, and there is not much variation in how they are arranged. Therefore, if only one umbrella is visible on a particular piece which is pink, SIFT may have a hard time distinguishing exactly which one it is within the puzzle. If the keypoints match to all the pink umbrellas in the image, there would be too many conflicting locations to output an actual match for the piece. Although this puzzle example is very specific having a repeating pattern, the same idea can be applied for other puzzles to a lesser degree; as pieces get smaller, less of the puzzle image is represented which narrows the available context of features to output a confident match.

Color Histograms

The color histogram results from puzzle testing did marginally increase overall matching percentage approximately 4.5% at 24 pieces to 2.4% at 96 pieces as seen in Figure 25, but it did not fully reach the level of accuracy hoped for. As the matching from color histograms was performed only on the pieces leftover from SIFT, the additional matches did not particularly follow the average consistently, but rather an all-or-nothing result was observed. The accuracy of the histogram matching did not decline in a linear fashion with piece count like the SIFT matching, but was more sporadic in its level of accuracy. For example, when only a few pieces were in contention, the color histogram matching did a good job of completing the puzzle fully, whereas when there were lots of pieces leftover from

SIFT the method had a hard time matching anything. Realistically, by the time a user finished reviewing the SIFT matches, they would likely be able to complete the puzzle themselves if there are only a few missing gaps with the help of the histogram results or not.

For a majority of the puzzles tested, the color histogram method would match to the same grid space for nearly all remaining pieces. This was an unfortunate downside with the histogram matching as although it would basically guarantee a single correct match to the particular grid space, sheer random guessing would have been just as effective. Upon investigating the RGB channel correlation values for many different pieces, the sum was in fact highest at the same particular grid space. This occurrenced more frequently with puzzles having lots of color variation and at higher piece counts. The concave and convex piece notches that were not represented in the puzzle grid of rectangles may have also skewed these results more than anticipated.

While the team considered excluding the histogram matching due to the high number of false matches, certain puzzles showed a different story. For more subtle color differences of a particular channel, the histogram matching performed surprisingly well especially when there were only a few unmatched pieces from SIFT that were considered. For example, one puzzle with a clear blue sky seemed to have an undistinguishable difference in color to the naked eye, however the color histogram was able to pinpoint the correct matches. With some positive results, we determined that there was no harm to include the histogram matching after all; there could only be additional matches added, as it would not detract from the initial SIFT matches.

Despite the poor accuracy in certain puzzles, the promising results in others indicates the potential this method may have if developed further. To increase consistency with puzzles of high color contrast, different histogram comparison parameters besides “cv.HISTCMP_CORREL” could be tested along with an HSV color model instead of an RGB one. In addition, the puzzle grid could be improved by considering the shape of the piece sides instead of having constant rectangles.

5. Conclusion

In this section, we will summarize the results from each technique implemented in our puzzle solver: edge detection, SIFT, and color histograms. Also, this section will reflect on what could be improved for future work and the lessons that our team has learned from doing this project.

We discovered that SIFT was a powerful tool to use in solving puzzles. With SIFT alone, our average accuracy for puzzles of piece count 24, 60, and 96 was 91.00%, 73.00%, and 53.96%, respectively. We discovered that images with many intricate details performed better than those with less features, as SIFT matching relies on having an abundance of detected keypoints. We also implemented color histogram matching for the remaining pieces not solved by SIFT, where the RGB channel values of each piece was compared with the remaining grid locations of the puzzle image. This increased the average accuracy of 24-piece puzzles by 4.58%, 60-piece puzzles by 3.86%, and 96-piece puzzles by 2.38%. In all piece counts, this increase in average accuracy made color histograms a worthwhile addition to our puzzle solver. The spline matching facet of the project was ultimately not included in the final app design. While we were able to apply subsequence matching on pseudo-pieces created, in real examples the splines of the pieces were too imprecise due to a lack of detected edge points.

The strategies of using image content, color, and shape to solve puzzles all change in importance from puzzle to puzzle. This leads us to conclude that there is no single method that would be ideal for every puzzle. A hybrid system can use all the strategies in tandem to fill in for where one may be lacking. With the system being so sensitive to the images passed in, we believed that it would be hard for practical use without the constraints laid out for taking pictures being followed with high precision.

Future Work: Improving Practicality

Robust Masking

The puzzle masking approach taken in this project sufficiently helped segment pieces, however the strict constraints were not quite practical for use outside of the online puzzle testing environment. While the image background was distinguished efficiently by considering just the outside border, it only holds for uniform backgrounds of a single color. As many everyday tables and desks have some surface variation, it may hinder user accessibility. In addition, it was difficult to take images of the puzzle pieces with proper lighting conditions; glare not only blurred the piece features, but it also interfered with the masking process. While the background color threshold slider did help to some degree, the system would need a more robust way of removing glare to expand its practicality.

Error Checking

As part of the experimentation procedure, the reported number of piece matches outputted by the system needed to be verified in order to be considered as actual matches. One observation from this process was how the piece contour displaying the match location became distorted and less precise as the number of pieces for a puzzle increased. For example, the piece contour would sometimes become stretched thin or wide. This could be explained by the loss of puzzle mask precision observed when having smaller pieces. Additionally, finding the homography transformation for smaller pieces became more difficult with the smaller sample size of keypoints available; it seemed the homography transformation would shear when groups of keypoints were in multiple areas, as it would try to account for both areas. While these distorted contours could oftentimes still be verified to be in the right spot, it became increasingly difficult when they were stretched over such a far region of the puzzle as shown in Figure 29.

Figure 29: Matching piece contour becoming distorted with smaller pieces and less precision in the mask

As a way of filtering out the matches with distorted contours, some error checking was implemented after calculating the homography transformation. A bounding box of the contour was taken, and if the area was almost zero or greater than a certain amount, the match was rejected. An alternative criteria attempted was examining the arclength of the contour for outlier values. This combination worked well together as some counters that were stretched completely thin would have an undetectable difference in it's bounding box area, but the arc length would be much greater than expected.

The major drawback to this error checking approach was how there was no definitive threshold to set for outliers. Each puzzle seemed to have varying degrees

of distortion in the contours for larger piece sizes, and there would likely be good matches thrown out in any case. For the bounding box area threshold, we took the pixel area of the puzzle image and divided it among how many pieces were in the puzzle, giving a rough approximation of the area of a single piece's contour. The maximum threshold was then set at a conservative three times this calculated area to account for larger pieces and extra space that may be present bordering the puzzle image. A more robust threshold could be discovered with more rigorous testing, but for our purposes this was sufficient.

While it was not a high priority, one functionality that was considered for the final design was a button to signal an incorrect match. While the claimed matches that SIFT produced were most often actual matches, there were occasional mismatches. As there is no guarantee a match is correct until a user attempts it with the actual puzzle, this proposed functionality would serve as a correction mechanism. Due to the consistency of SIFT, rerunning the matching on a certain piece would yield the same result; an alternative matching approach with edge detection or color histograms could be applied for such cases.

Other Approaches

There are other possible approaches to solving jigsaw puzzles with computer vision that were not included in our system. One such promising approach that was briefly explored was with Machine Learning, however ultimately it fell outside the project scope. Since matching pieces would have an overlapping region of common features, they could potentially yield similar values with an object recognition Machine Learning model. This method alone would probably struggle with similar puzzles that SIFT did, as it would be difficult to match pieces lacking internal features.

Lessons Learned

Upon reflecting as a team, some lessons learned through the MQP process was the importance of getting started with coding demos early on as it helped jumpstart the process rather than strictly focusing on the literature. When making design decisions for the application it was helpful to always keep the end user in mind. We could have stuck with our initial user interface design with streamlit, but we decided to start from scratch in a different direction with Flask in order to provide a better user experience for transitioning from one match to another. Along that note, it was important to keep an open mind to new ideas throughout the project duration. For example, we had not begun implementing color histograms until halfway through our MQP period. We also learned that while we may put time and effort into researching certain background or methods, we ultimately may not end up finding a use for them in our finished product. Examples of this were background research such as Bezier curves and Dynamic Time Warping (DTW), and more notably, spline detection and subsequence matching. Because we were researching how to solve this problem we learned to deal with some of the research providing no useful information in the end.

References

Alizadeh, E. (2020, Oct. 11). *An Illustration Introduction to Dynamic Time Warping.* Towards Data Science. https://towardsdatascience.com/an-illustrative-introduction-to-dynamic-time-warping-36aa98513b98. Retrieved Oct. 8, 2021.

Allen, T. V. (2016). *Using Computer Vision to Solve Jigsaw Puzzles.* 10.

Bellogin, Alejandro, and Pablo Sanchez. *"Collaborative Filtering Based on Subsequence Matching: A New Approach."* Information sciences 418-419 (2017): 432–446. Web.

Bodenheimer, R. (2020, Dec. 16). *What's behind the pandemic puzzle craze?* JSTOR Daily. https://daily.jstor.org/whats-behind-the-pandemic-puzzle-craze/

Boyat, A. K., & Joshi, B. K. (2015). *A Review Paper: Noise Models in Digital Image Processing. ArXiv:1505.03489 [Cs].* http://arxiv.org/abs/1505.03489

Deriso, D., & Boyd, S. (2019). *A General Optimization Framework for Dynamic Time Warping.*

Han, Tae Sik, Seung-Kyu Ko, and Jaewoo Kang. *"Efficient Subsequence Matching Using the Longest Common Subsequence with a Dual Match Index."* Machine Learning and Data Mining in Pattern Recognition. Berlin, Heidelberg: Springer Berlin Heidelberg. 585–600. Web.

HIPR. *Feature Detectors—Canny Edge Detector. (n.d.).* Retrieved September 6, 2021, from https://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm

Jain, R., Kasturi, R., & Schunck, B. G. (1995). *Machine vision.* New York: McGraw-Hill.

Liang, L., & Liu, Z. (n.d.). *A Jigsaw Puzzle Solving Guide on Mobile Devices*. 6.

Lindeberg, T. 1994. *Scale-space theory: A basic tool for analysing structures at different scales.* Journal of Applied Statistics, 21(2):224-270. http://kth.diva-portal.org/smash/get/diva2:457189/FULLTEXT01

Lowe, D. G. (2004). *Distinctive image features from scale-invariant keypoints.* International journal of computer vision, 60(2), 91-110. https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

Makridis, M., Papamarkos, N., & Chamzas, C. (2005). *An Innovative Algorithm for Solving Jigsaw Puzzles Using Geometrical and Color Features (Vol. 3773, p. 976).* https://doi.org/10.1007/11578079_99

MathWorks. (n.d.). *Edge Detection*. Retrieved September 5, 2021, from https://www.mathworks.com/discovery/edge-detection.html

Noise in photographic images | imatest. (n.d.). Retrieved September 6, 2021, from https://www.imatest.com/docs/noise/

OpenCV: Canny Edge Detection. (n.d.). Retrieved September 6, 2021, from https://docs.opencv.org/4.5.1/da/d22/tutorial_py_canny.html

Snavely, Noah. (2006). *Lecture 13: Homograpies and RANSAC.* MIT CSAIL. http://6.869.csail.mit.edu/fa12/lectures/lecture13ransac/lecture13ransac.pdf

The History of Jigsaw Puzzles. (2019, May 7). Wentworth Wooden Puzzles. Retrieved October 29, 2021, from https://www.wentworthpuzzles.com/us/2019/05/07/history-of-jigsaw-puzzles

Tyagi, Deepanshu. (2019 Mar. 16). Introduction to SIFT (Scale Invariant Feature Transform). Data Breach. https://medium.com/data-breach/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40

Yu, L., Kaijin, Q., & Kangli, Z. (2012). Achieving higher-order Bézier Curve based on the application of probability calculation method. Proceedings of 2012 International Conference on Measurement, Information and Control, 1, 493–497. https://doi.org/10.1109/MIC.2012.6273349

Zanoci, C., & Andress, J. (2016). *Making Puzzles Less Puzzling: An Automatic Jigsaw Puzzle Solver.* 7.

Zhang, J. (2020, Feb. 11). *Dynamic Time Warping*. Towards Data Science. https://towardsdatascience.com/dynamic-time-warping-3933f25fcdd. Retrieved Oct. 8, 2021.

Image References

Habicht, C. (n.d.). *1000 Colors Puzzle*. Uncommon goods. https://www.uncommongoods.com/product/1000-colors-puzzle. Retrieved February 23, 2022.

Lamamour. (n.d.). *R/mildlyinteresting - this puzzle I bought has 5 corners*. Reddit. https://www.reddit.com/r/mildlyinteresting/comments/ckpfsz/this_puzzle_i_bought_has_5_corners/. Retrieved February 23, 2022.

OpenCV. (2022). *Feature matching*. OpenCV. https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html. Retrieved February 25, 2022.

OpenCV. (2022). *Feature Matching + Homography to find Objects*. OpenCV. https://docs.opencv.org/3.4/d1/de0/tutorial_py_feature_homography.html. Retrieved February 25, 2022.

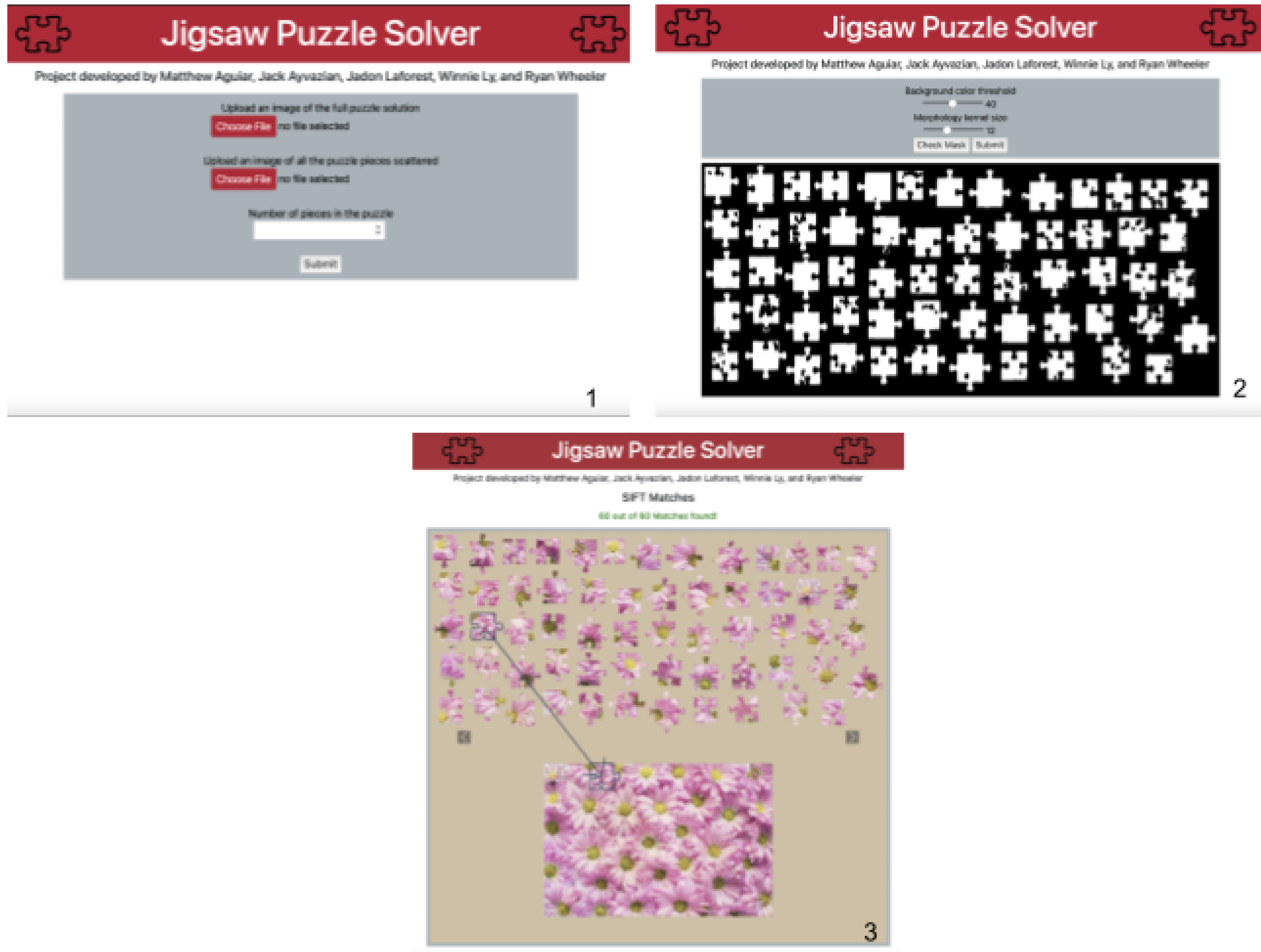
Signals. (n.d.). *A magical mystery tour of 100 beatles songs Jigsaw puzzle*. Signals. https://www.signals.com/HAG952.html. Retrieved February 23, 2022.

Tyagi, D. (2020, April 7). *Introduction to SIFT(scale invariant feature transform)*. Medium. https://medium.com/data-breach/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40. Retrieved February 25, 2022.

Whitcrrd, Reed. (Aug 21, 2019). Jigsaw Puzzle Piece Image Segmentation & Placement Prediction. https://github.com/whitcrrd/puzzle-image-segmentation

Appendix

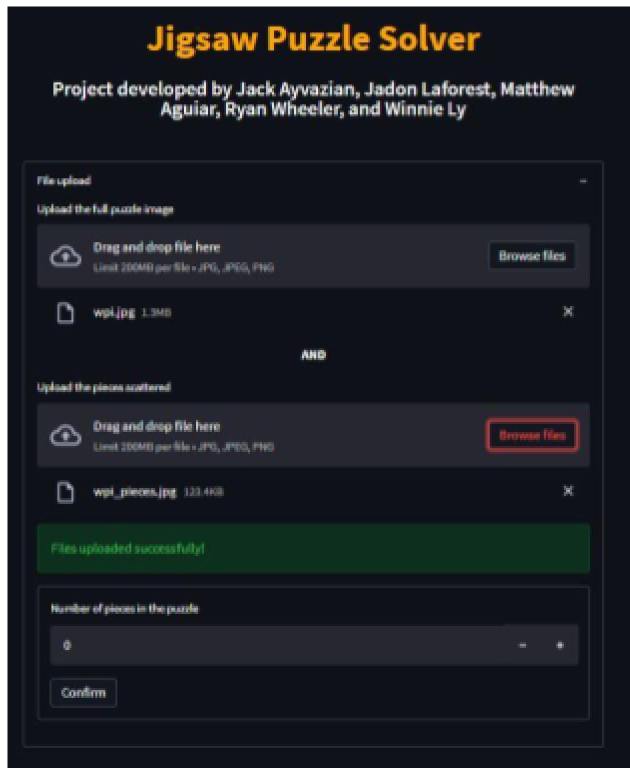
Appendix A: Flask User Interface Design

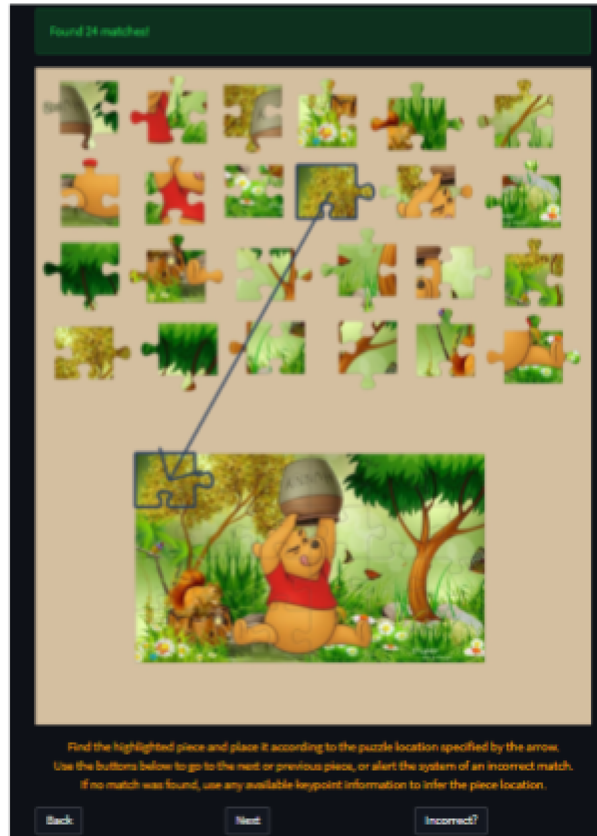


Workflow Steps:

1.
    a. Upload image files for full puzzle solution and all pieces scattered
    b. Input the number of pieces in the puzzle and click "submit"
2.
    a. Adjust the mask threshold sliders and click "check mask" to generate a new mask image
    b. Once the user is satisfied with the mask, click "submit" to begin matching
3. A carousel of image will display the matches for the pieces, where the left and right arrows navigate between them

Appendix B: Streamlit User Interface Design





Appendix C: Puzzle Testing Data

Puzzle ID	24 Piece (% Correct) SIFT	60 Piece (% Correct) SIFT	96 Piece (% Correct) SIFT	24 Piece (% Correct) SIFT+Histo	60 Piece (% Correct) SIFT+Histo	96 Piece (% Correct) SIFT + Histo
22	0.00%	0%	0%	25.00%	3.33%	2.08%
40	79.17%	31.67%	14.58%	100.00%	33.33%	15.63%
10	95.83%	45.00%	14.58%	100.00%	46.67%	16.67%
35	54.17%	35.00%	13.54%	79.17%	40.00%	16.67%
48	83.33%	38.33%	16.67%	87.50%	41.67%	17.71%
19	100.00%	41.67%	17.71%	100.00%	43.33%	19.79%
3	91.67%	43.33%	19.79%	100.00%	45.00%	22.92%
15	95.83%	56.67%	28.13%	100.00%	60.00%	29.17%
8	87.50%	51.67%	26.04%	100.00%	58.33%	31.25%
44	62.50%	45.00%	32.29%	70.83%	48.33%	33.33%
14	100.00%	76.67%	33.33%	100.00%	80.00%	34.38%
20	83.33%	56.67%	32.29%	91.67%	58.33%	35.42%
18	100.00%	75.00%	33.33%	100.00%	78.33%	36.46%
36	41.67%	26.67%	21.88%	62.50%	38.33%	36.46%
50	95.83%	56.67%	35.42%	100.00%	60.00%	39.58%
17	100.00%	65.00%	37.50%	100.00%	66.67%	40.63%
34	79.17%	53.33%	40.63%	83.33%	58.33%	42.71%
39	95.83%	66.67%	41.67%	100.00%	73.33%	42.71%
33	83.33%	60.00%	41.67%	95.83%	61.67%	43.75%
5	95.83%	80.00%	43.75%	100.00%	81.67%	44.79%
32	87.50%	60.00%	42.71%	91.67%	63.33%	45.83%
38	100.00%	81.67%	46.88%	100.00%	85.00%	48.96%
13	95.83%	73.33%	46.88%	100.00%	83.33%	50.00%
24	83.33%	58.33%	43.75%	95.83%	66.67%	52.08%
46	95.83%	81.67%	56.25%	100.00%	85.00%	57.29%
42	100.00%	93.33%	57.29%	100.00%	95.00%	59.38%
11	100.00%	93.33%	60.42%	100.00%	96.67%	60.42%
1	100.00%	90.00%	60%	100.00%	95.00%	61.46%
37	100.00%	91.67%	62.50%	100.00%	93.33%	63.54%
43	100.00%	86.67%	63.54%	100.00%	88.33%	64.58%
7	79.17%	76.67%	66.67%	100.00%	96.67%	67.71%
12	100.00%	90.00%	64.58%	100.00%	91.67%	67.71%
41	100.00%	96.67%	65.63%	100.00%	98.33%	68.75%
45	100.00%	81.67%	66.67%	100.00%	88.33%	68.75%
9	100.00%	90.00%	68.75%	100.00%	91.67%	70.83%
49	100.00%	85.00%	68.75%	100.00%	90.00%	70.83%
47	100.00%	91.67%	69.79%	100.00%	93.33%	71.88%
26	91.67%	80.00%	75.00%	95.83%	81.67%	76.04%
4	100.00%	93.33%	76.04%	100.00%	96.67%	77.08%
25	91.67%	81.67%	76.04%	100.00%	86.67%	77.08%
16	100.00%	100.00%	83.33%	100.00%	100.00%	86.46%
28	100.00%	100.00%	88.54%	100.00%	100.00%	89.58%
23	100.00%	93.33%	88.54%	100.00%	98.33%	90.63%
21	100.00%	96.67%	87.50%	100.00%	100.00%	92.71%
27	100.00%	100.00%	90.63%	100.00%	100.00%	92.71%
6	100.00%	98.33%	92.71%	100.00%	100.00%	93.75%
30	100.00%	98.33%	90.63%	100.00%	100.00%	93.75%
2	100.00%	100.00%	95.83%	100.00%	100.00%	96.88%
29	100.00%	100.00%	96.88%	100.00%	100.00%	97.92%
31	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Average:	**91.00%**	**73%**	**53.96%**	**95.58%**	**76.83%**	**56.33%**
Standard Dev:	**18.02%**	**24.04%**	**26.55%**	**12.75%**	**23.39%**	**26.06%**