Using Computer Vision to Solve Jigsaw Puzzles

Travis V. Allen
Stanford University
CS231A, Spring 2016
tallen1@stanford.edu

Abstract

Many in the computer vision community have written papers on solving the "problem" of jigsaw puzzles for decades. Many of the former papers written cite the potential extension of this work to other more serious endeavors like reconstructing archaeological artifacts or fitting together scanned objects while others simply do it because it's an interesting application of computer vision concepts. This author falls in the latter camp. Several methods for constructing jigsaw puzzles from images of the pieces were considered from a theoretical standpoint before the computing power and the high-resolution image capturing devices necessary to employ these methods could be fully realized. More recently, many algorithms and methods tend toward disregarding piece shape as a discriminant entirely by using "square pieces" and rely instead on the underlying image properties to find a solution. The jigsaw puzzle solver described in this paper falls somewhere in between these two extremes. The author of this project paper describes the creation of an "automatic" jigsaw puzzle solving program that relies on multiple concepts from computer vision as well as past work in the area to assemble puzzles from a single image of the disassembled pieces. While the program is currently specifically tailored to solve rectangular puzzles with "canonical" puzzle pieces, concepts learned from this work can be used in concert with other computer vision advances to enhance the puzzle solver and make it more robust to varying piece and puzzle shape. The puzzle solver created for this purpose is fairly unique in that it uses a picture of the disassembled pieces as input, no reference to the original puzzle image, and is done using the Matlab Image Processing Toolbox. The solver created for this project was successfully used on five separate puzzles with different rectangular bounds and dissimilar puzzle images. These results are similar to others who have created similar puzzle solvers in the past. Ultimately, the author hopes that this work could lay the groundwork for a smart phone application.

1. Introduction

This project focuses on the creation of an end-to-end "automatic" jigsaw puzzle solving program that assembles a jigsaw puzzle using an image of the disassembled puzzle pieces. The solver does not use the picture of the assembled puzzle for matching purposes. Rather, the solver created for this project attempts to solve the lost the box conundrum by displaying the assembled puzzle to the user without the need for the reference image. Needless to say, this program was created with an eye toward a potential smart phone application in the future. It should be noted that the puzzle solver is created entirely in Matlab and heavily utilizes functions found in the Matlab Image Processing Toolbox.

Solving jigsaw puzzles using computer vision (CV) is an attractive problem as it benefits directly from many of the advances in the field while still proving to be both challenging and intellectually stimulating. Due to time constraints, manpower limitations, and the fact that I had little to no prior experience with many of these concepts and the Matlab Image Processing Toolbox when beginning this project, several assumptions about the problem were made up front in order to make it simple enough to solve in the time given. The primary assumptions and limitations are as follows:

1. The pieces in the source image do not overlap nor do they touch.

2. The source image is captured in a "top-down" manner with minimal perspective distortion of the pieces.

3. The pieces in the source image comprise one entire puzzle solution (you can not mix and match pieces from other puzzles).

4. The final puzzle is rectangular in shape and all pieces fit neatly into a grid.

5. The pieces of the puzzle are standard or "canonical" in shape – this means they are square with 4 distinct, sharp corners and four resulting sides.

6. All intersections of pieces in the puzzle will be at the corners of the puzzle pieces and all internal intersections will be at the corners of four pieces.

7. Each side is characterized by having a "head," a "hole," or by being "flat."

In the following paper, I will discuss the previous work that has been done in this area in section 2, emphasizing those papers that most influenced the methodology I followed for my puzzle solver. I will then describe my technical approach to the problem and how my puzzle solver works in section 3. In section 4 I will show some of the results obtained with the puzzle solver so far and discuss both the experimentation that has been conducted and areas where more experimentation could occur. Finally, I will wrap things up in a conclusion in section 5.

2. Related Work

As one can imagine, such a problem as solving jigsaw puzzles might attract a good number of people in the computer vision community. And indeed it has. The problem has been considered for decades, going back to H. Freeman, and L. Gardner [2] in 1964. They first looked at how to solve puzzles with shape alone. Then there's H. Wolfson *et al*. [4] who describes the very matching methodology that I use. He is able to assemble a 104 piece puzzle using his method in 1988. While I do not solve a 104 piece puzzle, their solution requires individual pictures of each piece, though he does solve it with shape alone. One area where his method is different than mine is that it can handle two intermixed puzzles, whereas mine can only handle the pieces from one at present. D. Goldberg *et al*. [3] expanded on Wolfson's work and developed an even more global approach to solving jigsaw puzzles – their method allowed for the solution of puzzles that did not necessarily intersect at corners. My inspiration for color matching across boundaries comes from D. Kosiba *et al*. [5] who propose methods for using color in addition to shape in 1994.

Some insight into how to use Matlab to help solve this problem came from some detailed student papers that were found with a Google search. A. Mahdi [8] from the University of Amsterdam and N. Kumbla [6] from the University of Maryland both attempt the problem using methods similar to what I end up using, though neither ends up with quite full and satisfactory solutions and both rely on high resolution and highly controlled inputs. Finally, my inspiration for creating a smart phone application comes from L. Liang and Z. Liu [7] from Stanford who do not use the same matching methodology (they use the actual image of the fully constructed puzzle and SURF/RANSAC), but who do try to implement their solution in a real-time smart phone application. This is a possibility in the future for my puzzle solver.

Lastly, a former student of CS231A, Jordan Davidson [1], did his project in this very area, though it was in a slightly different vein. He looked at a genetic algorithm that could solve large "jigsaw" puzzles with square pieces that used the information from the pieces to determine if it had found the correct match. While the algorithm is interesting and probably applicable on some level to my puzzle solver, it was not quite what I was looking to do for this project. Jordan's work appears to be in an area that is growing with others attempting to solve larger puzzles of this kind. Personally, I wanted to solve the puzzles as they are seen and manipulated in real life. Most of this area of research has other applications and was not quite what I was looking to do. Though, as I said, there is definite applicability of some of the algorithms to my ultimate solver and future iterations may look at something like the algorithm explored in Jordan's paper.

3. Puzzle Solver Technical Approach

Creating a program to construct a puzzle using an image of the pieces requires a number of steps, each of which can be executed in a number of ways. In this section, I will describe the methods I used in my final code, but will also discuss alternatives that were either attempted with suboptimal results or that were not used but could be in future iterations.

3.1. Image Capture and Segmentation

In order to capture the pieces to be assembled into a final puzzle, I used a fairly high resolution camera – a Canon Rebel T4i DSLR with 18.0 Megapixel resolution. The pieces were placed face up on an easily segmentable background (i.e. a "green screen") with great care taken to ensure they were not overlapping (see figure 1 for an example using the Wookie puzzle). The picture was taken from a "top-dead-center" position looking straight down onto the pieces in order to reduce perspective distortion. Lighting was kept as neutral as possible with consideration given to sources of glare and to the possible disproportionate lighting of some pieces over others. With more time, the ability to compensate for off-axis image capture of the pieces (i.e. rectification to the ground plane) could be built into the code, though that was not explored for the current incarnation of the solver.

As discussed earlier, with an eye toward an eventual smart phone application, the first step I take is to significantly reduce the resolution of the input image from that of the original in order to shrink the memory burden and make the resulting image more comparable to one that might be obtained with a smart phone. Once I have resized the image (960x1440 was the main resolution used for the test cases), I use a Gaussian filter (with $\sigma = 1$) to blend the edges prior to the actual segmentation.

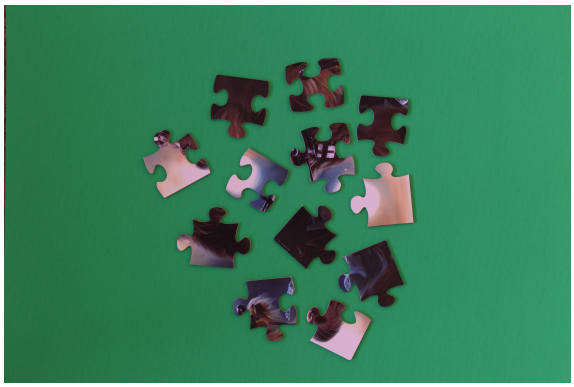

Figure 1. Input Image for the Wookie Puzzle

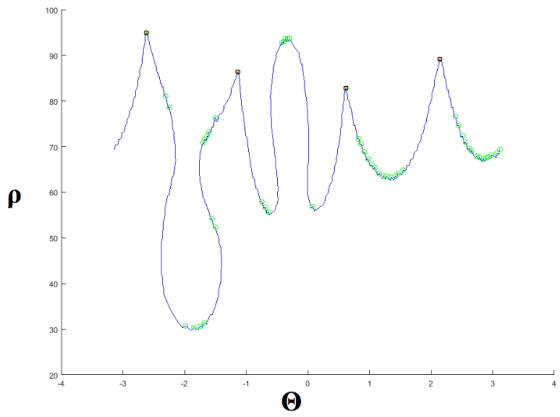

Figure 2. Polar Plot of Boundary for One Puzzle Piece

Since segmentation can be done in a multitude of ways, the approach I use is to put the pieces on a solid green screen background and segment based on color. This has been done for ease and efficiency since segmentation is not the entire focus of the project. However, as with the additional adaptation of the code to varying image capture angles, it would be entirely possible to segment the image using other means, such as the mean shift or normalized-cut methods used in class. These other methods have the potential to make the code more robust to varying backgrounds, such as wood table tops or off-axis image capture. These methods were considered, but were not completed since the puzzle assembly was considered more important to the overall program completion.

The primary result of the segmentation is a binary mask that we use to obtain the location and extent of the individual pieces using the `bwconncomp` and `bwregionprops` Matlab functions. There are provisions in the code for cleaning up somewhat imperfect masks that may have resulted from additional specks of dirt blocking the green background or when the green background does not fill the entire image and there is a segment along an edge where the true background (i.e. the floor or table) peeks through.

3.2. Puzzle Piece Characterization

Once the pieces have been segmented from the background and a binary mask created, the individual pieces are broken down further and many traits are extracted that will be used in later matching.

The first step is to define the corners. As per the assumptions about the puzzle pieces used, each piece has four, well-defined corners with an angle approximating 90 degrees. Several approaches for extracting either the corners or the sides were attempted before ultimately landing on the method used in the puzzle solver. One of the methods involved using the Hough transform to find the predominant direction of the points along the border while another found in [5] involved the conversion of the borders XY-coordinates to polar coordinates and then using this information to find the predominant direction of the points, taking the average of the intersection, and then taking the closest point on the boundary. While these methods may have eventually been adapted for my purposes, I had issues getting them to work consistently and instead landed on a method based on the one presented in [6] where I convert the border points to polar coordinates and use the Matlab function `imregionalmax` to find peaks in the data (other matlab functions were tried, but did not return as favorable of results). See Figure 2 for a visualization of what is returned for an individual piece. Unfortunately, since the border is somewhat noisy, there are many more peaks than actual corners or heads. Thus I wrote a script that steps through the peaks, consolidates the multiples to the true peak in the θ region, and tries to discriminate whether the peak is a corner or a head and, if a head, eliminates the peak. The script then goes in and uses the remaining peaks to determine the corners, which are two sets of peaks separated by 180 degrees that alternate.

Once the four corners of the piece are found, the sides of the piece are simply taken to be the points along the boundary between each set of two corners. The next most significant task is to assign each side of each piece as a head (+1), a hole (-1), or flat (0), see Figure 3 for a visual. This can be done in many ways. The original method I used would take the points along the edge in XY-coordinates, re-align them such that the two corners each lie on the x-axis, and then take the integral. If there was significant area either above or below the axis created by the two corner points, then the side could be declared either a head or hole, otherwise it was considered flat. However, there were issues with this method that were brought to light when experimenting with one of the puzzles. Specifically, there was difficulty setting a specific pixel integral threshold for all of the sides when the piece was more rectangular than square. One so-

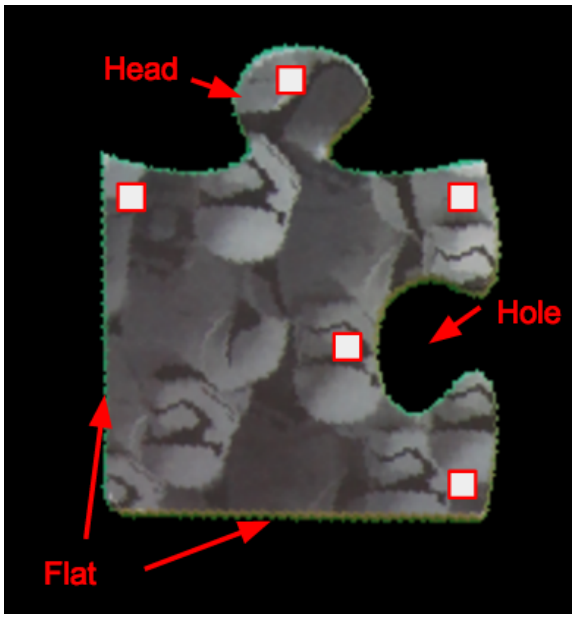

Figure 3. Example Piece with Labeled Sides and Example Color Patches

lution to this problem was to divide the resulting integral by the length of the side, but even that did not completely solve the issue. So instead I developed a new method that uses the "height" of the midpoint of the side relative to the x-axis as defined by the rotated side. If the height is beyond a certain threshold, it could be considered a head or hole. Otherwise it is considered a flat side.

Once the sides of the piece are determined and the information stored, the next step is to gather color information about the piece that will be used in the matching process. By using the mask created by the individual piece, I can use Matlab functions to back out intensity levels for individual color channels for the entire piece. At first I grabbed a lot of information, and much of it is still in the code. This includes average, minimum, and maximum intensity levels for each of the RGB color channels as well as each of the HSV color channels. After rather exhaustive experimentation with the matching algorithm, however, most of this information proved to be hit or miss when it came to true color coordination across pieces. Since this is information about the entire piece, it is primarily used to find regional likeness in the final puzzle image. After doing some research, one method for finding how similar two colors are in the spectrum is by calculating the ΔE, which requires the color values to be in CIE's L*a*b* color space. So, ultimately, the color values for each piece are translated to this color space and the average "intensity" in each of those channels is found and stored.

Not only do I store color information about the entire piece, but also color information along each edge. Using a method similar to [5], I identify small patches of pixels (I found three, 2x2 patches to work well) distributed evenly along the edge, grab the average L*a*b* color information in each of the patches, and store it for the matching process (see figure 3 for an example). Of course, I only do this along the edges with holes and heads – this is unnecessary along the flat edges. As with the piece information before, I also gather the HSV values for each of the patches, but no longer use them in the matching process after receiving mixed results.

3.3. Puzzle Assembly

When discussing the puzzle assembly, I have decided to break the process down first and foremost into "global" versus "local" assembly, and then break the "global" assembly down further into two distinct areas of the puzzle: the border pieces and the inner pieces. This is a method that is discussed and accomplished in most of the papers I reviewed, but specifically discussed in [4] and [3] and seems like a reasonable approach to the problem. It also seems like a logical approach given how one might go about solving a jigsaw puzzle in the physical world. As such, I will be breaking this section into subsections along those lines, beginning with the discussion on how two pieces are matched.

3.3.1 Local Assembly – Piece-to-Piece Matching

I am beginning the discussion on how the puzzle is assembled by describing the method by which the pieces are matched to one another. While the algorithm works slightly differently whether we're assembling the border or we're assembling the inner pieces, the specific match parameters remain the same. I will describe the general case of two pieces being matched together and then discuss the specific nuances for each of the two global situations.

What this algorithm is ultimately trying to find is a "match" between the edge of one piece and the edge of another piece. A "match" is defined as two pieces that "fit" together. Ideally, the two pieces that "fit" the best would also be the "correct match," so one would think that if the head of one piece is similar to the hole of another piece, they would fit and we could move on. However, when trying to do this process with little to no human input, finding the correct match is not as easy as finding two shapes that are simply the inverse of one another. Especially when one throws in measurement noise due to imperfect segmentation and image distortion. So instead, I try to use additional information about each piece in order to try and compute a "score" for a potential match.

After extensive experimentation, the factors that two neighboring pieces appear to have most in common are side length, color along the edge, and overall piece color. Additionally, I looked at the difference between the overlaid

curves along each edge to determine an average overlap (or gap) – obviously the smaller the overlap or gap, the better the match. With these factors in mind, here is how the matching algorithm works.

Once the algorithm determines the two pieces and the side of each piece to be matched, it first checks to make sure one is a head and one is a hole. If not, the match is discarded. It then compares the side lengths and the integrals along each edge. If the difference in side lengths is significant (beyond approximately 10 pixels for the standard resolution image), the match is discarded. The integral of the curve along each edge is compared. If they are not approximately equal and opposite, the match is discarded (the threshold for this varies based on image resolution – turns out it is not the best method for weeding out candidate matches so it is not the most strict). Next, the overlap is calculated by overlaying the two edges in XY-coordinate space and taking the difference (using `pdist`). The result is the overlap, or gap, between the two pieces. If the average, minimum, or maximum overlap is beyond specific thresholds based on image resolution, then the match is thrown out. Once we've looked at the basic shape discriminants for determining whether a match is likely correct, we then look at the color discriminants and determine the ΔE. We do this both from a regional perspective (piece to piece) and from an edge perspective (using the patches along the edge). ΔE is essentially the "distance" between two colors in the color spectrum and is determined using the following formula:

$$\Delta L^* = L^*_{1Avg} - L^*_{2_{Avg}} \quad (1)$$

$$\Delta a^* = a^*_{1Avg} - a^*_{2_{Avg}} \quad (2)$$

$$\Delta b^* = b^*_{1Avg} - b^*_{2_{Avg}} \quad (3)$$

$$\Delta E = \sqrt{\Delta L^{*2} + \Delta a^{*2} + \Delta b^{*2}} \quad (4)$$

The ΔE is found for each pair of patches along the edge and the average of those patch values is used. Obviously, the lower the ΔE, the closer the colors are in the spectrum.

Once we've computed and captured these shape and color comparisons and have weeded out obviously bad matches, we then compute a match "score." This score is found using experimentally determined weights that are multiplied by the four key matching criteria: side length difference, overlap difference, ΔE along the edge, and ΔE between the pieces. Since these values tend to vary widely between matches and puzzles, the standard range of values found for "correct" matches on one of the test puzzles was used to develop a set of weights that somewhat normalizes each parameter so that one specific criteria is not favored too much more than another. For most correct matches, when multiplied by the weights, the values should be no greater than 100. This means that, theoretically, a correct match could have a total score of up to approximately 400. However, in reality, most correct matches have low scores in at least two of the categories, so a score threshold of about 280 (again, experimentally determined) can be set in order to weed out incorrect matches.

Once the local matching algorithm has pared down to a final set of scored matches, it then returns these matches to the global algorithm in score-priority order. The primary nuances that differ between local matching for the border versus the inner pieces is the orientation of the piece. As I will soon discuss, the border pieces are aligned according to the flat edge, so the local matching only considers one potential edge for each piece. And, because the border is being matched in the absence of the rest of the puzzle, the local matching algorithm does not have to consider any additional sides from other pieces that may come into play. Not so for the inner pieces. For the inner portion of the puzzle, a piece is being matched to a slot in a grid, and that slot has neighboring sides. As will soon be discussed, the local matching algorithm will always have at least two sides for each internal piece, but could have upwards of three or four to consider depending on the location of the slot and what pieces have been matched so far. In the inner case, the local matching algorithm has to ensure that the heads and holes all line up first, as before, but then determines all of the matching metrics per side and takes the average over the number of sides. The major difference here is that a single piece could fit into a slot in multiple ways, so piece orientation must be accounted for and the piece must be rotated in all valid configurations before the algorithm returns a set of matches. A single piece could potentially have four possible "matches" to a single slot, depending on hole/head orientation. The match thresholding and scoring are the same across pieces and edges in the inner piece matching as in the border matching, the only true difference being that they are averaged across each piece/edge to which the piece is matched in the inner matching (which is unnecessary in the border case).

3.3.2 Global Assembly – the Border Pieces

It is logical to begin the global assembly with the border because the border pieces are distinct – they each have at least one flat side. Since the piece matching algorithm is not perfect and does not always return the "correct" match as the "best" match, a so-called "greedy" algorithm that simply places the "best" match between two pieces in the next available slot will not necessarily result in a coherent solution (i.e. one might get something that is non-rectangular or even nonsensical). In order to provide for this possibility while not resorting to a "brute-force" approach that runs through every possible combination of pieces, I decided to use a so-called "branch-and-bound" algorithm, normally used in the solution of the Traveling Salesman Problem (TSP). In the general description of the TSP, a salesman

needs to do business in a number of cities spread out over a region with defined distances between each. The salesman wants to find the shortest overall route that goes to every city only once and returns to his starting point – thus its a distance minimization problem.

Much like the TSP, each match made between two pieces along the puzzle border is given a score. Once a solution is found, the total of all of the match scores that make up that border solution should reflect how good the solution is. Ideally, the smallest overall score will be the best and correct solution. However, that turns out not always being the case, as will be discussed.

Since there are many ways of reaching a solution, we need a way of capturing a large number of possible solutions and then finding the best one of those potential solutions. One way of doing this is the branch-and-bound method. This algorithm will be discussed shortly, but first I will describe the greater methodology for how the border is constructed.

The general construction of the border begins with a corner piece. We orient the piece such that the counter-clockwise-most flat side is "down," with the other flat side to the "left" and a head or hole to the "right." Matching is then done to the "right" in a sequential manner. The local matching algorithm, then, receives a left piece and a list of possible right pieces, all with the flat side down. It returns a list of potential right pieces. We then choose one of the right pieces to be the new left piece and continue the process until we run out of possible right pieces. As we progress around the border, when we hit another corner, we rotate the entire puzzle and continue as if the flat side of each piece is "down."

Now, since the potential number of solutions is $(n-1)!$ where n is the number of border pieces, we want to constrain the number of solutions found through whatever means necessary. The local matching algorithm does a good job of weeding out very poor matches, but will still return multiple potential matches that could lead to a nonsensical full solution. In order to combat this we also place a set of side length constraints on the puzzle such that the border solution must have side lengths corresponding to a rectangular puzzle. If we've gone too long without a corner piece or if we get sides defined by corner pieces that do not equal one another, the solution is thrown out and we search for a new one.

After all of this preamble, I am now going to describe the basic branch-and-bound algorithm that is used for the border matching problem. The algorithm can be described in the following manner:

1. Choose a starting piece (usually a corner) and call it piece A.

2. Use piece A to find a set of potential matches – these can be considered "children" or "branches."

3. Since the local matching algorithm returns a rank-ordered list of potential matches, start with the first potential match as piece B.

4. Next, remove piece B from the list of pieces remaining to be matched and make piece B the new piece A.

5. Use the new piece A to find more potential matches.

6. This process continues until one of the following occurs:

   (a) We run out of pieces remaining – in this case we have found a solution or "leaf." We store this set of matches and their scores as a solution, back up, and see if we can find more solutions.

   (b) We run into a border constraint that isn't satisfied – in this case we back up and see if another match does satisfy the border constraint before moving on.

   (c) We do not get any potential matches for the current piece A – in this case we need to back up and see if we can find another path using a different piece from an older set of potential matches.

The algorithm either runs until it has exhausted the search space and found all possible solutions based on both the side constraints and the local matching thresholds, or until it has obtained the number of solutions requested of it. As can probably be surmised from the basic description above, one can visualize this approach as a tree with the first piece at the root and branches extending upward for each potential match. If we are able to make it all the way up a branch to a leaf, then we have found a solution to the problem. If we get stuck on a branch and can't expand, we come back down the tree until we find another path that looks fruitful. In this way we can reduce the total number of solutions tried to well below that which would be found through simple brute force.

As one can see, because the problem is being solved in a nonlinear fashion, the number of solutions that might be found before the "correct" solution is highly dependent on several factors, not least of which are the first piece chosen and how well the first few pieces match. If incorrect matches are made early in the process, it can take a long time (and a lot of matches) before the correct solution is found. And even when the correct solution is found, it may not be the "best" solution as per the scoring system. Ideally the "best" and "correct" would be the same, but that is not always the case.

3.3.3 Global Assembly – the Inner Pieces

Once a border solution is found, it is passed to the global assembly for the inner pieces. The global assembly algorithm assembles the border pieces into a grid and then grabs the "upper left" open slot as the new "piece A." This algorithm also creates a grid with relative orientations for each piece. When the pieces were first characterized, they were each oriented with "side 1" being "up." Once they are placed in the final puzzle grid, we create a second grid with the same dimensions that provides the relative 90 degree rotation from "up" for each piece (0-3). Because we know we have the border completed, we can be assured that that first slot will have at least two pieces along its edges. The more pieces along an edge, the more accurate and discriminating the score should be. The inner piece matching algorithm then uses a similar branch-and-bound algorithm as in the border case to find potential matches for this first slot. It then removes the piece from those remaining and moves across the puzzle filling in all available pieces from left to right and then top down (like reading a book). Unlike in the border case, however, the potential matches could involve the same piece, just oriented differently. For this algorithm, orientation is very important. As with the border assembly algorithm, once all of the pieces are found, that solution is stored and we then back up and see if we can find more until either we run out of solutions or we have found the number of solutions desired. Ideally, the solution with the lowest total score (aka the "best" solution) will also be the "correct" solution.

One last note about the global assembly algorithms: while it might make sense to have the two algorithms separate during developing and while trying to understand where each breaks down, the ideal case would be to combine these algorithms in order to weed out border solutions that do not provide for full puzzle solutions. This was thought about, though not implemented in the final code. Had there been more time, this would have helped to bring down the total number of end puzzle solutions. As it was, in the time allowed, I was simply able to get both of these algorithms working well enough to tweak the various variables to see how best to find matches. The next step would be to link these two algorithms and throw out border solutions that have no potential solutions based on all of the criteria discussed above.

3.4. Final Image Construction

Once we have assembled all of the pieces into a grid with their relative orientations, we now have the solution to the puzzle. The next step is displaying that solution to a user. Ideally, I would like to display a completed and fully stitched together puzzle image using the puzzle pieces as segmented from the original image. While this is most likely possible using the Matlab Image Processing Toolbox, it was not completed in a satisfactory manner by the end of this project. Instead, I use the functions `vision.AlphaBlender` and `step` along with the piece masks and the cropped segmented pieces from the original image to create a quasi-final image "grid" that shows the extracted pieces oriented per the solution. It's not ideal, but it at least shows how the final pieces should be laid out and arranged. For reference, see figure 4 for an example of the final solution.

Puzzle Name	Total Pieces	Border Pieces	Border Soln	Inner Soln
Wookie	12	10	1st	1st
Storm Troops	24	16	58th	5th
Droids	12	10	1st	1st
Speeder	16	12	1st	1st
Rey Finn	12	10	1st	1st
Kylo Ren	24	16	N/A	N/A

Table 1. Test Puzzle Results

4. Experimentation and Results

I ran my puzzle solver in Matlab on both a home PC with 4 year old hardware (6 GB RAM, Intel i5 processor, AMD Graphics Card) and a 13-inch MacBook Air, 2015 model with 4GB memory and Intel Graphics with little difficulty. It takes about a minute or so to run one of the test puzzles (it might take more than a minute for the 24 piece puzzles) from image segmentation through to final construction. I carried a good amount of information in memory throughout the process since I was doing a lot of experimentation and wanted the ability plug and play various modules for both fine-tuning and debugging. This could be pared down for a future implementation.

While the puzzle solver created for this project cannot be used on every puzzle (per the limitations noted earlier), for those it could be used on, I was able to experiment to find limitations and weaknesses. I also used this experimentation to find the best criteria for matching.

For this project I tested the puzzle solver on six Star Wars themed children's puzzles that I bought at Target. The overview of the results using the final matching parameters are in Table 1.

As one can see, most of the puzzles had 12 to 16 pieces, except for two that had 24 pieces. Of all the puzzles that had less than 24 pieces (four of the puzzles), the correct border solution was the best border solution returned, by score (hence the "1st" in the third column). Then, using the correct solution as the lead-in to the inner puzzle algorithm, those same puzzles found the correct solution to be the one with the best score.

For the Storm Trooper puzzle, there were two primary

factors that led to it not doing as well. First, there are more pieces and therefore more potential matches. Still, if the matches were registering scores that reflected the true "correctness" of the match, then one would expect the overall score of the completed border to be better than 58th. And, even when we fed the correct border solution to the inner puzzle algorithm the correct solution was 5th best by score. However, to put this into perspective, the global border solution algorithm returned in excess of 2000 potential border solutions for the storm trooper puzzle (of a mathematically potential $15!$, approximately 1.3 trillion, solutions with brute force), of which the correct one was 58th by score. Which isn't all that bad. Additionally, the storm trooper puzzle was by far and away the most homogeneous in terms of color of all the puzzles. It was very difficult to discriminate matches based on color using the methods I described before, especially because of the way the storm trooper line discontinuities happen to match up along the border of the pieces, making cross border color matching very difficult indeed. And finally, the pieces were fairly square, so all four sides were very even and comparable in length. If they were instead more elongated with one pair of sides longer than the other, the side length discriminant would have knocked down potential matches.

This was a case where the experimentally determined match scoring algorithm broke down. While it worked in the other test cases very well, one can see quite clearly that other methods would need to be pursued in order to get the storm trooper puzzle to be solved correctly.

The other puzzle in the table that was looked at as a test case but does not have a rank for a solution is the Kylo Ren puzzle. This puzzle highlighted the need for a better corner-finding or edge-finding process. While the code I developed to find the corners repeatedly on the other puzzles worked quite well, the pieces of the Kylo Ren puzzle were extremely elongated and many of the "heads" were so small as to be mistaken for corners. Needless to say, the automatic corner-finding was not able to find the corners, and without them the rest of the algorithm just doesn't work as is.

When experimentally determining the criteria to use for the piece matching, individual matches were observed with special attention paid to the values for correct matches. The Wookie puzzle was used as the baseline case and the values derived from this puzzle were applied to the others with general success (except for the Storm Trooper puzzle). Here were the typical values and the final weights applied:

- Side Distance Difference: 0-8 pixels (wt = 12.5)
- Overlap Average Difference: 0-14 pixels (wt = 7.0)
- ΔE Patches: 7-45 (wt = 2.9)
- ΔE Pieces: 4-35 (wt = 3.2)

While it would logically seem, and in most cases it would actually be, that matching the color patches across the boundaries should be one of the best ways to discriminate in order to find a true match, due to the variability in how the puzzle pieces are carved up, this was not always the case. For instance one piece was carved almost perfectly along Wookie's nose, which is dark, and just on the other side of the edge there was a bright background. The ΔE in this case was fairly large even though the match is a correct one. In fact, this was also a case where the puzzle pieces had a small ΔE between them, but the patch difference was much higher. This is not an expected result. And while one might begin to think that maybe color is too volatile and should not be considered at all since the perceived variability in the shape is much smaller among true matches per my above list, that is not entirely accurate. Due to noise and distortion, the actual length of the sides and the measured overlap is not exact. And while the correct match is always small, so are many other matches. These criteria are best for weeding out those pieces whose shape isn't even close to correct. It can also help when the head of one piece is bent in one direction while the hole of the other piece is expecting it to be bent in another – then the overlap will suffer. For the most part, however, the size helps get you close. Unfortunately, many of the pieces have differences that fall within the acceptable ranges above. That is why color is then used to help with the ranking of those potential matches. And in most cases, the color does help. There are just a few in every puzzle where there are large transitions in both the puzzle region or just along the border that cause the match scoring to return some interesting values.

Additional parameters that were used early on for color scoring were RGB and HSV channel averages. While in some cases there was clear correlation, in many others there didn't appear to be any correlation whatsoever. Color variance was also considered, though it was also disregarded because, after some thought, I could not see how it would return a marked improvement. The extreme variability in the color scoring led to a rethinking about how the colors were being compared and the eventual use of ΔE.

Additional methods for finding border matches that were considered but were unable to be implemented before this report were:

- Segmentation along the border (such as meanshift) – if we could determine there are a certain number of segments along one border that coincide with a certain number of segments along another, then maybe we could find a potential match.
- Find lines along a border that break at the border, then look for the continuation of these lines on the other side. Would have something to do with the flow of pixels – seems difficult to implement, though would

really help with the Storm Trooper puzzle.

- Grab features within the head piece or along the edge and build a Bag of Words model. Then try to find matches on the other side of the piece around the hole. This has potential, but is potentially computationally expensive.

While the matching algorithm used isn't perfect, it worked for the test cases considered. And while others may have been able to solve larger puzzles [4][3] with their algorithms, my code proved to be fairly robust and efficient at solving the puzzles provided. Since there appear to be no standardized "jigsaw puzzle metrics" against which to compare by puzzle solver for puzzles with irregular shapes, I cannot say exactly how my puzzle solver compares to others that have been developed. However, it is one of the few that I've seen that takes a raw image of all of the pieces at once and produces a fully constructed solution. Most of the puzzle solvers found in official papers and in student submissions cited earlier produce only partial or theoretical solutions, or solutions that require even greater initial constraints than my own (i.e. the pieces have to lie in a grid at the outset or each piece has to be scanned individually with a high resolution scanner). Still others rely on the original image to find the location of the pieces in the final image, which is not the problem I set out to solve. One example using the Wookie puzzle can be seen in with the original image in figure 1 and the final solution as found by my puzzle solver in figure 4.

5. Conclusion

I have created an end-to-end "automatic" jigsaw puzzle solver that uses Matlab and the Matlab Image Processing Toolbox to piece together a jigsaw puzzle using only an image of the pieces. I used this puzzle solver on six test puzzles and proved that it works on five of them quite reliably, but also found where there were weaknesses in the current implementation. Certain design decisions were made early on that simplified the problem such that I could complete the entire project by the deadline. Unfortunately, this also meant it was hard to go back and try a completely new method once I had begun going down a certain path.

I learned a lot over the course of this project. I learned about how to think about a 3-dimensional, physical world problem in terms of a 2-dimensional perception of that problem. I learned how to think about manipulating every ounce of information I could glean from a single photograph to help the computer "think" like a human and make matches that would result in a correct solution. I learned about many functions inherent within Matlab, especially the Matlab Image Processing Toolbox. As I developed the program, I learned new tools and tricks that, had I known them earlier, I may have approached certain parts of the project differently. This knowledge will certainly be helpful in the future and could be applied to improving the puzzle solver.

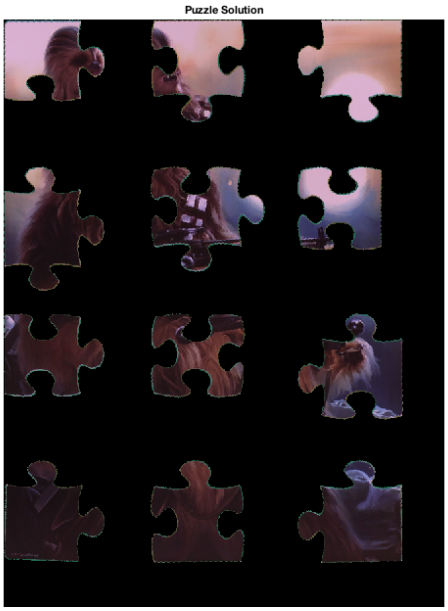

Figure 4. Solution Created by the Automatic Puzzle Solver

Figure 5. The "Truth" – A Picture of the Assembled Wookie Puzzle

Several of the papers I read where people have attempted

this problem in the past did not make much sense to me until I went and attempted it myself. I had believed it would be easier to extract the corners of the four-sided pieces, and therefore decided to go with canonical piece jigsaw puzzles. However, this then meant I was fairly limited in the types and numbers of puzzles to which my program could apply. It also meant that extraction of this information was absolutely essential to everything my program did afterward. Some of the other methods, like the use of fiducial points as in [3], may have proven more difficult at first, but could have paid dividends in its ability to scale.

If my original end goal was to create a program that began to explore the possibility of creating an automatic jigsaw puzzle solver smart phone application, which was the original idea, then I believe I have achieved a pretty great stride in that direction. However, my code is not yet robust enough to the kinds of inputs a smart phone might provide, nor is it efficient enough in both memory allocation and processor requirements to be feasible for that application. Many changes would have to be made before I can get to that end goal, which is something I realized about halfway through the project. While I believe I have created a solid and workable solution within the constraints of the problem as I originally set forth, I see many areas where it could be improved for future incarnations. All in all, I did what I set out to do, I learned a lot, and I enjoyed the process.

References

[1] J. Davidson. A genetic algorithm-based solver for very large jigsaw puzzles: Final report.

[2] H. Freeman and L. Garder. Apictorial jigsaw puzzles: The computer solution of a problem in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-13(2):118–127, April 1964.

[3] D. Goldberg, C. Malon, and M. Bern. A global approach to automatic solution of jigsaw puzzles. *Comput. Geom. Theory Appl.*, 28(2-3):165–174, June 2004.

[4] A. K. Y. L. H. Wolfson, E. Schonberg. Solving jigsaw puzzles by computer. *Annals of Operations Research*, 12:51–64, 1988.

[5] D. A. Kosiba, P. M. Devaux, S. Balasubramanian, T. L. Gandhi, and K. Kasturi. An automatic jigsaw puzzle solver. In *Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision amp; Image Processing., Proceedings of the 12th IAPR International Conference on*, volume 1, pages 616–618 vol.1, Oct 1994.

[6] N. Kumbla. An automatic jigsaw puzzle solver.

[7] L. Liang and Z. Liu. A jigsaw puzzle solving guide on mobile devices.

[8] A. Mahdi. Solving jigsaw puzzles using computer vision.