1 Overview

For my final project, I chose to attempt writing a program to solve jigsaw puzzles. The program would take image files of jigsaw puzzle pieces and determine how they should be placed to get the assembled puzzle.

I have three code files: \texttt{imgProcess.py}, \texttt{puzzProcess.py}, and \texttt{assembly.py}.

- \texttt{imgProcess.py}: contains most of the image processing code involved in my project. To see the program working, the DEBUG flag at the top of the file can be changed to True and image outputs will be displayed. To close a displayed image, hit any key on the keyboard.

- \texttt{puzzProcess.py}: contains the main function for looping through input pieces, and classes to store the information gained from image processing. This program can be run using the line

  \[
  \texttt{python puzzProcess.py pieces}
  \]

  where pieces is the name of a directory in the same directory as the program. The pieces directory contains the input images for the program.

  If the DEBUG flag of \texttt{imgProcess.py} is set to True, it is highly recommended to run the program on a subset of the puzzle pieces. The test directory can be used for this purpose.

- \texttt{assembly.py}: hardcoded puzzle piece data is used to run the assembly algorithm. Usage is

  \[
  \texttt{python assembly.py assemble}
  \]

  The assemble file contains the same images, but with special image names that I use for getting data about the side of each piece in the hardcoded version of my assembly algorithm.

In this section, I will define the problem constraints and the subtasks I broke it into. In later sections, I will elaborate on the process of solving each subproblem and the methods I used to do so.
1.1 Constraints

To solve any problem, it is necessary to first make some assumptions. In this case, making assumptions translated to setting constraints for the scope of the problem. The constraints I chose were as follows:

1. One puzzle piece per image
2. Solid white backdrop for the puzzle piece
3. Solid dark colored puzzle pieces
4. Puzzle piece roughly centered in the image

I would have preferred to use the image on the face of the puzzle piece, but in the end I decided to constrain the problem to deal only with the shape of the puzzle piece so that I would have an easier time extracting piece boundaries with edge detection.

1.2 Subproblems

1. Input data

This writeup discusses issues according to the logic of the program, and not in the order they were encountered. I bought a jigsaw puzzle to take pictures of for developing my project, but before I had the chance to do so I used a sample image from Google.

![Image of puzzle piece](image)

**Figure 1: Initial development image**

Note that this image is simple, with uniform colors and clear contrast between puzzle piece and background.

Later, I used a simple, nine piece jigsaw puzzle and photographed the pieces on a blank sheet of paper. These images were the inputs for my program. A good additional constraint would have been to use a mounted camera and take pictures from a consistent distance to ensure that all the puzzle pieces would have the same scale, but unfortunately this equipment was unavailable.
2. Identify the puzzle piece in the image

   The first task is to take the image file and find the puzzle piece in it. This is very easy for a human, but less easy for a computer program.

3. Recognize each side of the puzzle piece

   After task 1, I was able to get an array of points representing the contour of the puzzle piece object. The next task was to split up this array into the parts representing each side of puzzle piece. This proved to be more difficult than expected.

4. Analyze piece sides

   After getting arrays representing sides of the puzzle piece, I needed to be able to classify the sides of the puzzle pieces to determine whether the sides of two puzzle pieces fit together.

5. Assemble the puzzle

   Lastly, I used a greedy algorithm to assemble the puzzle starting with one corner.

6. Displaying output

   Finally, I had to display the output in a way that would be easily understood by humans.

2 Identifying the Puzzle Piece

The input images were too large, so the first step was to scale them down. Once I had an image that could be viewed on one screen, the next step was the begin processing.

To binarize the image, I first called OpenCV’s built in GaussianBlur function to remove noise, then used threshold with Otsu Binarization (see http://en.wikipedia.org/wiki/Otsu%27s_method). Otsu Binarization is a method which dynamically chooses the threshold value based on the input image. This helped account for variations in lighting for the different images.

Next, I used another OpenCV feature called findContours, which allowed me to extract shape boundaries from the binary image I had created. This worked beautifully on the inputs which binarized well, meaning that the puzzle piece was the only region that was white in the binarization. When I was using a computer generated image, there was no issue because the background was uniformly white. After I used my own pictures, I had some images with messy blobs in the binarized version, as below.
When I used `findContours`, these blobs were included in the list of contours returned. To resolve this, I calculated the centroid of each region outlined by a contour, and selected the contour corresponding to the centroid closest to the center of the image. This is why I needed the constraint that the puzzle pieces be roughly centered in their image. Finally, I had a contour representing the outline of a puzzle piece. If needed, I could have smoothed the contour using a polygonal approximation algorithm, but the contours I was finding turned out to be smooth enough to be used immediately.

Internally to the computer program, a contour is simply a numpy array containing the defining points of an outline. This understanding was very important when I was doing the next part of this project.

3 Recognizing Sides of a Piece

This was the most difficult part of the project. I had to take an array of points and divide the array into segments according to the sides of the puzzle piece. I tried corner detection first, but it didn’t work well so I used more complicated methods.

I found the minimum bounding box of the puzzle piece contour and chose a particular segment of the contour array to correspond to each side of the bounding box. This was done by iterating through every possible subcontour of the puzzle piece contour and taking the subcontour which minimized a heuristic value.

At first, my segment selection heuristic selected segments based on slope and length. “Slope” of a contour was calculated by finding the slope of the line between the start and end points of the contour segment. The heuristic was also set to assign a higher penalty to contour segments whose endpoints were closer together in the image. This meant it would try to maximize the length of the contour segment chosen, rather than choosing only part of the side of the piece.

To see whether I was dividing the contour correctly, I drew the bounding box of the image and drew a line between the start and end points of the contour segment that had been selected. If the run was successful, the yellow (and later green) lines drawn
should line up with the four sides of the puzzle piece. All contour points in the array which fell between the endpoints would be included in the contour segment chosen, so the shape of that side of the puzzle piece would have been extracted. Some early attempts are shown below.

Figure 3: Early attempt #1

Figure 4: Early attempt #2

It was difficult to achieve a balance of the properties in the heuristics since opti-
mizing one could damage to another, but still produce the best heuristic value overall.

For example, segments with bad slope sometimes had a lower line length penalty since the slanted line would be longer.

Later, when I modified my heuristic to penalize segments whose endpoints were farther away from the bounding box, I had an entire generation of outputs similar to the one below.

Figure 5: The left segment is close to the bounding box, but too short.

Eventually, I was able to find a stable heuristic that worked both for the simple image and a real image.
From the start, I knew that brute forcing every possible subcontour would be very slow. Halfway through making tweaks to the heuristic, I changed my program to use the convex hull of the contour rather than the original contour. The convex hull is just an estimate of the contour with much fewer points. I didn’t need all the details of the contour yet, and this gave far superior runtimes. I was able to get the convex hull points in terms of indices into the original contour array, so I could still take segments
of the original contour after identifying the points I wanted to use for my boundaries.

4 Analyze Piece Sides

When a human solves a puzzle, a frequently used strategy is to put together the border of the puzzle and then fill in the interior. I wanted to use this strategy in my program, and I thought it would be straightforward to determine whether a contour was a line or a jigsaw-shaped side.

I first attempted to use polygonal approximation. The polygonal approximation algorithm reduces the number of points in a curve, so for a contour which is already roughly a line, the result should be two or three points. But I found that this was not the case with my contours, and I wasn’t able to figure out why.

Next, I tried fitting a line to each of my contours and classifying a contour based on how closely it matched a line. However, my program output that the distance from the contour to the fit line was thousands of pixels for each of the contours. There is most likely a math error which I will investigate further in the future.

Next was curve matching. As future work, I hope to implement a real algorithm for determining whether or not two curves complement each other. But for this project, I was only able to identify whether a side had an extrusion or an indentation. I accomplished this by fitting a line to my contour and checking which side of the line the points of my contour fell on. An extrusion’s points would fall on the outside of the contour, and an indentation’s points would fall on the inside. This was actually sufficient for the puzzle I used, because the shapes of the pieces were not complex. However, the goal is to be able to run my program on more complicated inputs.

This section is the subproblem that still needs the most work. My classification of extrusion/indentation is unreliable for some pieces, and I wasn’t able to classify pieces by type. This made it impossible to test the next section, assembly, so I created another copy of my code with the classifications of edges hardcoded into the filenames. This file is called assembly.py. For future work, I will most likely modify assembly.py to call the functions in imgProcess.py. puzzProcess.py is more of a sandbox version of the code.

5 Assembling the Puzzle

assembly.py can be run to view the output of my assembly algorithm, even though the image processing doesn’t entirely work.

I named the puzzle image files according to the labels my image processing should have assigned each edge. The labelings start with the bottom edge and go clockwise. Rather than use image processing, this program parses the image filename to extract the type of each side.

The algorithm chooses a corner to start with, and puts the puzzle together working along each row. It assumes that any “In” side will complement an “Out” side. Based on this, it constructs an array of the type of edge needed for each side of any piece that fits, and greedily chooses the first piece it encounters that fits the constraints. To determine whether a piece fits, it attempts each rotation of the piece until the piece has the right sides in the right positions.
If the process reaches a dead end, or a point where no piece fits, it backtracks to a point where there were multiple possible pieces and tries a different piece from the one chosen the previous time.

The solution is accumulated as an array of piece objects. Once the solution has been found, a method is called to display the solution as one image tiled with each of the puzzle piece images. While solving the puzzle, information about the rotation of the piece was stored in the piece object, so each piece’s image can be rotated to the correct orientation for displaying. The working output is shown in the figure below.

I had some difficulty rotating and tiling the images. The dimensions of the images being tiled had to match, and keeping track of the proper rotations introduced some bugs which took a while to find. I also had to scale down each image before concatenating it with the others, or the final image would be too large for one screen.

6 Conclusions

In computer vision, tasks which seem trivial can take a great deal of time and effort to program. Often, there are not many optimizations that can be made and brute force is the way to go. For example, I attempted to optimize my contour segmentation procedure, and achieved poorer results because the points I needed would sometimes be skipped. The key idea is to extract only necessary information and disregard unnecessary details.
Computer vision algorithms are also difficult to debug, because simply printing all
the values provides very little useful information. OpenCV is powerful, but at times
it is too inflexible or too opaque and the developer can have trouble identifying subtle
issues with how the built in functions are being used.

On the other hand, computer vision is very rewarding when successful. This project
was only partially successful, but I would consider the results a good basis for continued
work on this project. There are dozens of ways to improve the image processing in this
program for the puzzle contours alone.

Computer vision projects seem to rarely have a clear finish line. I was able to learn
many of the basics of practical computer vision from this project, and this should allow
me to do more advanced work in the future.