SOFTWARE REQUIREMENTS SPECIFICATION DOCUMENT

THE PUZZLE SOLVER

Version 1.0

**Prepared by : James Wheelock
Duncan Caruthers
Timothy Dodd
Jeffery Michaelis
Colin Woods**

**Submitted to : Senior Design Class
COSC 4950**

December 7, 2021

Contents

1 Introduction

1.1 Purpose

This document is intended for two audiences: potential users of our software and those interested in the details of its implementation. The first purpose of this Software Requirements Specifications document is to provide details to potential users concerning what is required to run and use our software. The second purpose of this document is to provide detailed information about how our software works, such that readers may be able to implement it themselves.

1.2 Scope

This software will be an automatic jigsaw puzzle solver named “The Puzzle Solver.” The software will allow users to input images of puzzle pieces spread out on a solid colored background and will output an image of the solved puzzle, with pieces labeled so the user can construct the solution themselves. The software will specifically be designed for puzzles with standard piece shapes, i.e. generally square pieces with clearly defined corners and locks. The software is also specifically designed for jigsaw puzzles with square boundaries, including four corner pieces and side pieces between them.

This puzzle solver will work by using computer vision to detect pieces and features about each piece. Then, the software will use this information to process the similarities between each piece. Finally, the solver will try to maximize the similarity between each piece in a solution, and return this solution as the most likely candidate.

1.3 Definitions, Acronyms, and Abbreviations

Any reference to ‘puzzle’ will strictly mean a standard jigsaw puzzle in the context of this document. A standard jigsaw is any square puzzle with four corners, any number of side pieces, and middle pieces with four edges, all standard shapes. Any reference to ‘piece’ refers to a singular puzzle piece. Groups of connected pieces should be called a ‘section’ of the puzzle.

Any reference to ‘corner piece’ will strictly mean a puzzle piece that has two straight edges, which are adjacent to one another. Any reference to ‘side piece’ will strictly mean a puzzle piece that has one straight edge.

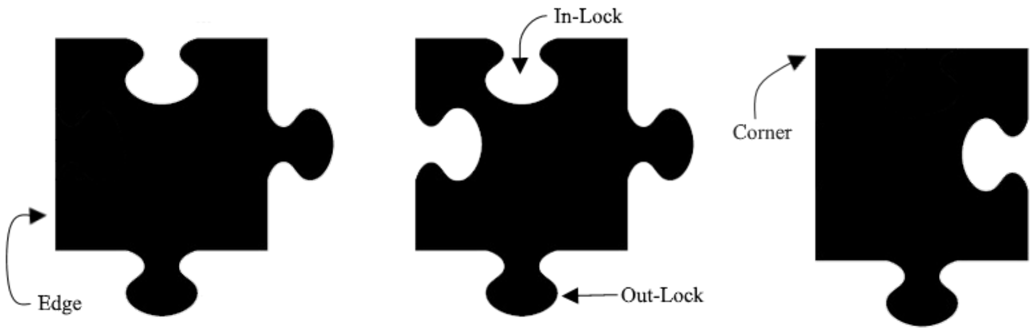

Figure 1.1: Examples of standard jigsaw puzzle pieces

Any reference to 'in-lock' and 'out-lock' will strictly mean the part of a puzzle piece that is referenced in figure 1. Any reference to 'lock' will mean either 'in-lock' or 'out-lock'. An example standard puzzle piece with these labels is shown in figure 1.1.

For this document, computer vision will refer to the computer use of digital images to perceive information about the world it sees. In our example, this includes the computer's processing of images of puzzle pieces to learn details about each piece.

Genetic algorithms refer to algorithms that build off of previous generations of solutions to create better solutions by combining details from multiple solutions together. For our software, we will be generating new layouts of pieces that fit certain constraints using previous layouts.

1.4 References

1. Allen, T. V. (n.d.). Using Computer Vision to Solve Jigsaw Puzzles [Scholarly project]. In Web.stanford.edu. Retrieved October 15, 2021, from https://web.stanford.edu/class/cs231a/prev_projects_2016/computer-vision-solve_-_1_.pdf

2. Computer Vision Powers Automatic Jigsaw Puzzle Solver. (2021, October 11). Retrieved November 15, 2021, from https://www.abtosoftware.com/blog/computer-vision-powers-automatic-jigsaw-puzzle-solver

3. Sholomon, D., David, O., & Netanyahu, N. S. (2013). A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles. 2013 IEEE Conference on Computer Vision and Pattern Recognition. doi:10.1109/cvpr.2013.231

1.5 Overview

This first section is intended to inform readers about the purpose of this document. In the second section, factors affecting the requirements are discussed, intended as background for later sections. This section is also for potential users of the software so that they can understand how the software operates without getting into detailed requirements. The third section discusses these requirements, intended for developers who want to understand the technical details of the software. The goal of this third section is to discuss requirements in such detail that interested readers would be able to implement our software design themselves.

2 Overall Description

2.1 Product Perspective

2.1.1 System Interfaces

We interact with the system's graphics, which will be accomplished through the use of TkInter. We will also interact with the system's file explorer, which will also be accomplished using TkInter.

2.1.2 Interfaces

Our software will provide a GUI for the user to interact with. The GUI will provide the user with a method for the input of files, and prompt them for information required to run our puzzle solver. The GUI will display puzzle solutions to the user, with labels on the pieces so that the user can construct the solution themselves. The GUI will not have any accessibility features in the current version.

2.1.3 Hardware Interfaces

This system relies on a scanner for the user to get images of their puzzle pieces. For testing of the software, we have been using the app CamScanner, without image filters. Users may use other scanning apps or scanning hardware to get images of their puzzle pieces.

2.1.4 Software Interfaces

Our program has a number of software interfaces. If the user runs our program as an executable, knowing these will not be relevant as they will be built in. If the user decides to run the program from the source code, then these will be relevant. These software interfaces are described below.

Name	Python Programming Language
Mnemonic	Python
Version	3.10.0
Source	www.python.org
Description	Python was used to write all code for our software, so it translates the code into something that can run on user's computers.

Name	Open Computer Vision
Mnemonic	OpenCV
Version	4.5.4
Source	Provided by Python Install Repository
Description	Almost all image processing is done using OpenCV. Image data is passed to one of their functions, and processed data is returned.

Name	TkInter
Mnemonic	Tk
Version	3.10.0
Source	Provided by Python Install Repository
Description	TkInter is used to creater a user interface for our software.

Name	Scientific Python
Mnemonic	SciPy
Version	1.7.1
Source	Provided by Python Install Repository
Description	SciPy provides useful functions for data processing that we use when finding corners and labelling pieces.

Name	Numerical Python
Mnemonic	NumPy
Version	1.21.2
Source	Provided by Python Install Repository
Description	NumPy allows us to do more complicated operations on lists in an efficient way, as it is written in C.

2.1.5 Communication Interfaces

As our program is intended to be run by a single user, without any connection to other users, there are no communication interfaces.

2.1.6 Memory Constraints

This software should be able to run on any system with more than a Gigabyte of memory.

2.1.7 Operations

The initial mode of operation will include the user uploading their files to the software. This mode will prompt the user for information about the images being uploaded, and allow them to change settings for the puzzle solver.

There will then be a period of unattended operation while the system solves the puzzle. The length of this time will be dependent on the size of the puzzle.

After the puzzle has been solved, the user will be able to interact with the solved image and save it to their computer.

2.1.8 Site Adaptation Requirements

Our software will only operate if given clear, high-quality images of the puzzle pieces. So, the software requires that users find suitable lighting, scanning software, or equipment to scan photos of their puzzle.

2.2 Product Functions

Our product has one primary function: to solve a jigsaw puzzle using photos of pieces uploaded by the user. Our software also should show the solution to the user and allow them to interact with it, but this is mostly an extension of the main goal.

To achieve this functionality, our software breaks the goal of solving a jigsaw puzzle into clearly defined tasks. These tasks are described in the following list.

1. **Extraction of piece information from images**
   Our software must extract information about puzzle pieces from each image uploaded by the user. This is done by identifying the background color, removing it, and identifying the remaining sections of the image as pieces. This means that it is important for pieces not to be overlapping, as each piece must be in its section to be identified properly.

2. **Labelling pieces**
   Our software assigns a set of numbers to each piece so that the user can easily find a piece in their collection once seeing the solution. Each piece is labeled according to its location in an image, and according to which image it comes from. Labeling is done by row and column in an image, so it may be helpful to lay out pieces in a grid.

3. **Solving the puzzle**
   - ❖ **Generating initial solutions**
     Once all pieces have been labeled and all information about them has been found, our software will begin solving the puzzle. Our solver begins with randomized solutions, placing puzzle pieces in a grid but randomly and without correct edges together. Many such random solutions are found, and each solution is given a score.
   - ❖ **Generating better solutions**
     Using the incorrect solutions already found, our solver creates new solutions that may be better. It does this by sampling information from the solutions with the best scores found previously. By compiling the information from pairs of solutions together, better solutions can be found.
   - ❖ **Generating the final solution**
     To generate the best solution, our solver begins with thousands of random solutions. As described above, new solutions are generated using these random solutions. Once a new group of solutions is found, these are plugged back into

the process, generating even better solutions. This process is repeated many times. At the end, the solution with the highest score is output and made visible to the user.

4. **Displaying the final solution**
   The final goal of our software is to be able to show users solutions to their jigsaw puzzles so that they can recreate them themselves. After completing the process shown above, an image is created which contains the final solution to the puzzle. On this image, the piece labels will be shown. The user should be able to move around in the image, look at each piece, and find the piece matching it in their collection. By doing this for every piece, users will be able to solve their jigsaw puzzles easily.

2.3 User Characteristics

We hope that virtually anyone would be able to use our software as designed. Users should not be required to be educated or have technical expertise, but they should be competent enough with computers to scan images of puzzle pieces, run an executable, and navigate a file system. Users should also have the ability to sort colored puzzle pieces onto contrasting backgrounds, as this will help get better results from computer vision.

Due to the nature of our software, many users may be attracted by how it works. So, it may be beneficial to include verbose options, allowing users to see details about what the program is doing at any moment. This will not be a default option though, as the software should remain open to anyone with or without computer science experience.

2.4 Constraints

Due to the nature of our problem statement, our software does not have any constraints that users would need to know about.

2.5 Assumptions and Dependencies

The largest factor that our software depends on to work correctly is that users have access to standard jigsaw puzzles. This is not always true, as there was a shortage in spring 2020, and it could happen again. Without a puzzle to scan, our software is useless. We also assume that users use standard-shaped puzzle pieces, and if non-standard shapes are used, our software will not work.

To run our software without an executable, we assume that users have a recent version of Python 3 installed on their machine. We also assume that users not using an executable have access to the libraries we use in our code. As we plan to compile this software into an executable, these will not be assumptions if the user chooses to download that version.

We assume that the user has access to either a scanner or an app that automatically scans documents. The specific scanner app that we are using for testing may not be available in the future, but similar apps should be.

We also assume that users have access to brightly colored paper, used when scanning the images. This may not be the case based on availability at their local stores.

When scanning the photos, users should maintain image resolution 1080p and 1440p so that the software has enough information to run, but not too much information as to slow it down.

2.6 Apportioning of Requirements

We have many ideas to add to the software in future versions. Some of these ideas may be simple, while others may be more complicated. This affects the likelihood of each feature being implemented. Some of these ideas may seem simple but would be significantly challenging to implement. Below is a list of possible future expansions to our software, with descriptions for each item.

❖ **Solving puzzles with missing pieces**
At the moment, our software uses several pieces to determine the piece dimensions of the puzzle (6x4, 13x24, etc). An alternative method could be found to work around the presence of missing pieces. Missing pieces may present unknown challenges in the workings of our current puzzle solver.

❖ **Creating and solving digital jigsaw puzzles**
To demonstrate the ability of our puzzle solver, it would be interesting to write an automatic digital jigsaw puzzle creator. This would take an image as input, and output an image containing standard jigsaw puzzle pieces, taken from the input image. These pieces should form the original image when put together. Solving such puzzles would not be difficult, but creating them would be challenging. This would mostly be useful for those curious to use the product without buying their jigsaw puzzles.

❖ **Solving jigsaw puzzles with non-standard pieces**
Many companies sell puzzles with non-standard pieces. These could include wavy edges on some middle pieces, flat edges on middle pieces, borders that are not square, and abstract-shaped pieces with more than four sides. We would like to expand the types of pieces used in our puzzles so that users can buy a puzzle to try out without fear of surprise non-standard puzzle pieces. This problem would be difficult to solve but is certainly feasible for specific types of puzzle pieces. An early extension of the types we allow will likely include wavy middle-piece edges, as found in many Buffalo brand puzzles.

❖ **Utilizing more complex image features**
At the moment, only the color values around the sides of the pieces are used. In the future, we would like to expand this to use features of the image as it appears

on the box of the puzzle. Users would be able to scan the cover of the box, and we would be able to extract useful information from these pictures. Using machine learning or simple heuristics, this information could be very valuable to us. This will only be added if it helps us solve puzzles with a high piece count.

- **Developing a more interactive and attractive user interface**
  We would like to encourage users to use our puzzle solver by developing a more attractive environment to display the puzzle solution in. Nothing is off the table when it comes to the user interface. We would like the process of solving the puzzle to be made transparent to the user by displaying the progress made after each generation. This could be done in a simple 2d environment, featuring options and settings for the user alongside the image of the best solution found so far. We could also render the pieces in 3d so that they can be displayed on a table as an actual puzzle would be.

  Instead of showing the labels for each piece on top of the image, we could only show the label on the piece which the user is hovering over with the mouse. This would clean up the appearance quite a bit.

- **Extending our work to more complex problems**
  Previous work has been done on solving generalized jigsaw puzzles (digital images cut into squares) with more complex requirements. One special extension includes double-sided puzzles, where an image is printed on both sides of any given piece. Another includes mixed puzzles, where there are two separate images constructed out of pieces that have been mixed. There are many more extensions that could be thought up, and we would like to try our hand at a few of these. Specifically, the mixed puzzle idea would be a fun problem to tackle next, using images of real jigsaw puzzles as the input.

3 Specific Requirements

3.1 External Interfaces

The primary mode of input for this software shall be photos of 1080p 1440p resolution puzzle pieces, laid out and separated on a solid, colorful, background. These photos are used as the primary data about the puzzle so that it can be solved by the software. The system shall prompt the user to upload photos of the pieces, provide the user with a file explorer to find the photos on their file system, and store these files for future use. The software should be able to handle .jpg or .png file formats.

The system shall also prompt the user for more information about the puzzle. This includes the dimensions of the puzzle and the number of pieces in each image uploaded to the system. The dimensions of the puzzle input by the user shall be positive float or integer numbers and shall be stored for future use. The number of pieces in each image uploaded shall be integers greater than zero. These values shall not be checked by the system for validity, as the specific user is the only one affected by incorrect input information.

The system shall provide a GUI to the user so that they can interact with the puzzle solver. This GUI shall provide methods to input the data mentioned above, as well as methods to interact with the solution output by the system. The user shall be presented with the option to save an image of the generated solution. This saved file should be output as a .jpg file, and its name should be distinct from all other existing files in the same location.

3.2 Objects and Classes

3.2.1 Class 1: Piece Edge

The class PieceEdge shall store all information about an edge of a piece, and provide the ability to compare PieceEdge objects to other such objects.

3.2.1.1 Attributes

- image
- number
- contour
- distance_array
- label
- color_array

3.2.1.2 Functional Requirements

The following functions shall be implemented in the PieceEdge class.

- $\langle constructor\rangle(number : int, image : numpy\_array, contour : numpy\_array)$
  This constructor shall create a new instance of this class, and assign its variables to those passed to the constructor. The distance array, color array, and label variables should be assigned values by calling the corresponding helper functions in that order.

- $findDistanceArray(void)$
  This function shall calculate and update the value of the object's distance_array variable. This variable shall contain the perpendicular distance to the line between corners for each point on the edge's contour. Pseudocode describing this calculation is shown in the following algorithm.

```
distance_array ← [ ]
$x1, y1$ ← contour[0]                                   // first corner
$x2, y2$ ← contour[-1]                                  // second corner
for $x3, y3$ in contour do
    /* Finds the perpendicular distance between x3, y3 on
       the edge's contour and the line between x1, y1 and
       x2, y2.  Appends to distance_arr.                    */
    $d = \frac{crossProd([x1,y1]-[x2,y2],[x3,y3]-[x1,y1])}{norm([x1,y1]-[x2,y2])}$
    distance_arr.append($d$)
end
```

❖ $findEdgeLabels(void)$

This function shall find the label for the edge object by using information from the edge's distance array. The object's label variable should be updated after this calculation. This is described in the pseudocode shown below.

```
$epsilon \leftarrow 30$                                  // Set empirically
$v\_extreme \leftarrow max(abs(\text{distance\_array}))$
/* Uses the value of the most extreme distance from the
   line between corners to determine the shape of the edge
   */
if $v\_extreme \leq epsilon$ then label $\leftarrow$ "flat"
if $v\_extreme = max(\text{distance\_array})$ then label $\leftarrow$ "outer"
if $v\_extreme = -min(\text{distance\_array})$ then label $\leftarrow$ "inner"
```

❖ $findEdgeColors(void)$

This function shall calculate the average color along the object's contour and store these colors in color_array. Colors should be stored in the same order as the points they correspond to in contour. This is described in the algorithm below.

```
/* Range of pixels from edge to average color over, set
   empirically                                              */
color_range $\leftarrow 3 \ldots 10$
color_array $\leftarrow$ [ ]
for $i$ in $1 \ldots len(\text{contour})$ do
    /* Find the unit vector perpendicular to the contour at
       each point                                           */
    $pL, p, pR \leftarrow \text{contour}[i-1, i, i+1]$
    $dx, dy = pL - pR$
    $perp = \frac{[dy, -dx]}{norm([dy, -dx])}$
    /* Go along the line perpendicular to the contour at
       point $p$ and average the colors along this line     */
    color_sum $\leftarrow$ [0, 0, 0]
    for $j$ in color_range do
        new_point $\leftarrow p + j * perp$
        $c \leftarrow \text{image}[new\_point]$
        color_sum += $c$
    end
    color_array.append(color_sum / $len$(color_range))
end
```

- ❖ *compare*(*other* : *PieceEdge*)
  This function shall compare the PieceEdge object passed as a variable to the function to the object calling the function. The result of this comparison will be an integer whose value denotes the difference between the two PieceEdge objects. If this number is high, the objects are very different and if the number is low, they are very similar. The calculation of this value is described in the pseudocode below.

```
Input: other
Output: difference
if one of the objects' labels is "flat" then return infinity
// Find the shorter and longer edges
if $len$(contour) $<$ $len$(other.contour) then
    short, long ← self, other
else
    short, long ← other, self
end
/* position shorter edge over longer edge, find position
   that minimizes difference                              */
minDistPos ← 0
for $s$ in $0 \ldots len$(long.contour) $-$ $len$(short.contour) do
    $e \leftarrow s + len$(short.contour)
    dist ← $sum$($abs$(short.distance_array - long.distance_array[$s : e$]))
    if dist < minDist then minDistPos ← $s$
end
/* Return the sum of the differences between the color and
   distance arrays, at the index that minimizes differences
   */
dist ← $sum$($abs$(short.distance_array -
 long.distance_array[$minDistPos : minDistPos + len$(short.contour)]))
color_dist ← $sum$($abs$($norm$(short.color_array -
 long.color_array[$minDistPos : minDistPos + len$(short.contour)])))
return dist + color_dist
```

3.2.1.3 Messages

PieceEdge objects shall be accessed from Piece objects when the edges are created and the object's data is initialized. The PieceEdge function "compare" shall be called from the solution class, but no other PieceEdge functions shall be used elsewhere.

3.2.2 Class 2: Piece

The Piece class shall store information about puzzle pieces in its objects.

3.2.2.1 Attributes

- label
- image
- contour
- corners
- edges
- type

3.2.2.2 Functional Requirements

- $\langle constructor\rangle(label : int, image : numpy\_array, contour : numpy\_array)$
  This constructor shall create a new instance of this class, and assign its variables to those passed to the constructor. The corners, edges, and type shall be defined by calling the corresponding functions, listed below.

- $findCorners(void)$
  This function shall set the value of the object's corner variable using the information from the contour. The corners should be stored so that they are in clockwise order with respect to their position in the object's contour. The entries added to the corners array shall contain the corners' indices in contour as well as their x and y pixel locations in the image. Pseudocode describing the process is shown below.

```
/* Translate the x, y values in the contour so that the
   center of the contour is at the origin                    */
center ← [mean(contour.x), mean(contour.y)]
centered_contour ← contour - center
// Convert cartesian to polar coordinates
polar ← cartToPol(centered_contour)
// Find indexes in contour of sharpest peaks in polar data
peaks ← findPeaks(polar)
sharpness ← sharpness(peaks)
sharpest ← peaks[argsort(sharpness)[-4:]]
for peak in sharpest do
    /* Store the corner's index and x, y position in the
       array                                                 */
    entry ← [peak].append(polToCart(contour[peak]))
    corners.append(entry)
end
```

- ❖ *findEdges(void)*
  This function shall set the value of the object's edges variable using information from its corners and contour variables. The edges should be stored so that they are in clockwise order with respect to the object's contour. Pseudocode describing this process is shown below.

```
/* Finds the positions along the contour where the current
   and previous corners are located, creates an edge using
   the section of the contour between them as an argument
   in its constructor                                    */
/* The first edge will go over the boundaries of the
   contour, so should include the concatenated sections of
   the contour instead                                   */
pos1 ← corners[−1][0]
pos2 ← corners[0][0]
new_edge ← PieceEdge(0, image, contour[pos1:]+ contour[:pos2])
edges.append(new_edge)
for $i$ in $1 \dots len$(corners) do
    pos1 ← corners[$i - 1$][0]
    pos2 ← corners[$i$][0]
    new_edge ← PieceEdge(i, image, contour[pos1 : pos2])
    edges.append(new_edge)
end
```

- ❖ *findType(void)*
  This function shall set the object's type variable using information from its edges. The type variable shall be set to "corner", "side", or "middle" dependent on the number of side edges on the piece. Pseudocode describing this process is shown below.

```
num_flat ← number of PieceEdges in edges with label "flat"
if num_flat = 0 then type ← "middle"
if num_flat = 1 then type ← "side"
if num_flat = 2 then type ← "corner"
```

- ❖ *getSubimage(edge_up : int)*
  This function shall return an image containing the section of the image containing all points in the contour. This function shall rotate the image so that the PieceEdge at index edge_up is displayed at the top of the piece so that the line between its corners is flat. Pseudocode describing this image manipulation is shown below:

```
Input: edge_up
Output: sub_image
/* Find the corners on either side of the PieceEdge at
   index edge_up                                          */
c1 ← corners[edge_up - 1]
c2 ← corneres[edge_up]
// Find the angle to rotate the image
$dx, dy$ ← c1.position - c2.position
angle ← $arctan(dy/dx)$
if $dx < 0$ then angle += 180

/* Find the minimum circle that encloses all points on the
   contour, use this to crop the image                    */
$x, y, r$ ← minEnclosingCircle($contour$)
sub_image ← image[$y - r : y + r$][$x - r : x + r$]
sub_image ← rotate(sub_image, angle)        // rotate the image
putText(sub_image, (x,y), label)             // Label the piece
return sub_image
```

3.2.2.3 Messages

Piece objects shall initialize only four PieceEdge objects each. Piece objects shall be initialized in the PieceCollection class, and used in the PieceCollection class, Solution class, and PuzzleSolver class. The Solution class uses Piece's getSubImage function. No other Piece functions shall be used outside of the Piece class.

3.2.3 Class 3: Piece Collection

The Piece Collection class shall provide methods and attributes to store information about puzzle pieces coming from multiple images. The Piece objects for each puzzle piece in the images shall be instantiated here.

3.2.3.1 Attributes

- images
- pieces
- num_pieces_array
- num_pieces_total

3.2.3.2 Functional Requirements

- $\langle constructor \rangle (void)$
  This constructor shall set all object variables to default values. This shall be the

empy list for images, pieces, and num_pieces_array, and zero for num_pieces_total.

- ❖ *addPieces*(*image* : *numpy_array*, *num_pieces* : *int*)
  This function shall append image to the object's array of images and append num_-pieces to the object's variable num_pieces_array. The variable num_pieces should then be added to num_pieces_total.

  Then, the function shall call getContours, passing image and num_pieces. After this, the function shall call getLabels, passing the contours from getContours. A new Piece object shall be created for each pair of labels and contours, passing the label, the object's image, and the contour to the constructor for Piece. This Piece object should then be appended to pieces.

- ❖ *getContours*(*image* : *numpy_array*, *num_pieces* : *int*)
  This function shall return a list of contours in the image, specifically a number of the largest contours equivalent to num_pieces. The contours shall be detected using openCV's inRange, dilate, and findContours functions. Pseudocode showing this process is shown below:

```
Input: image, num_pieces
Output: contours
/* Before finding contours, convert the image from BGR to
   HSV color format                                        */
image_HSV ← BGR2HSV(image)
/* Find the mode in the color data, to be used as a
   background to remove                                    */
mode ← mode(image_HSV)
/* Create a mask to remove this color                      */
delta ← 10                                 // set empirically
mask ← inRange(image, mins=mode - delta, maxs=mode + delta)
mask ← dilate(mask, kernel=(5,5), iterations=2)
/* Finally, do contour detection, sort by area, and return
   num_pieces largest ones                                 */
contours ← findContours(mask)
sorted ← sort(contours, key=contourArea, reverse=True)
return sorted[:num_pieces]
```

- ❖ *getLabels*(*contours* : *numpy_array*)
  This function shall return an array of tuples to be used as the label value in Piece objects. Each tuple shall contain three integers, representing the image that the piece is on and the approximate column and row that the contour is in with respect to the other contours.

  This function should use two iterations of hierarchical clustering, one using distance in the x direction to get column labels, and another in the y direction to get row labels. This function shall still output labels even if the pieces in the image are not laid out in neat columns and rows. This function should use SciPy's fclusterdata function to cluster the data. This function shall never assign the same label to two distinct contours, and shall always assign the same label to the same contours given as input. Pseudocode that finds this array is shown below.

```
Input: contours
Output: labels
// Find the centers of each contour in contours
centers ← [ ]
for contour in contours do
    center ← (mean(contour.$x$), mean(contour.$y$))
    centers.append(center)
end
/* Cluster according to the centers' $x$ coordinates to get
   column labels                                              */
x_clusters ← fclusterdata(centers.x)
x_centers ← clusterCenters(clusters)
// Make sure cluster labels increase as $x$ increases
ids ← argsort(x_centers)
lookup[ids] ← range($len$(x_centers))
x_labels ← lookup[clusters]
// Repeat for $y$
y_clusters ← fclusterdata(centers.y)
y_centers ← clusterCenters(clusters)
ids ← argsort(y_centers)
lookup[ids] ← range($len$(y_centers))
y_labels ← lookup[clusters]
/* Find tuple of labels for each contour and return           */
for $i$ in $1 \ldots len$(contours) do
    labels[$i$] ← ($len$(images) − 1, x_labels[$i$],y_labels[$i$])
end
return labels
```

3.2.3.3 Messages

PieceCollection objects shall be created only in the main class. PieceCollection shall be referenced only by the solutions class and the puzzle solver class. Variables of PieceCollection objects shall be accessed by these classes, but no functions should be used other than getter functions, not shown here.

3.2.4 Class 4: Solution

The solution class shall store information about a single potential solution to the puzzle. This class shall include methods for scoring potential solutions, generating new potential solutions from existing ones, generating an image of the potential solution, and verifying that new solutions are valid layouts of the puzzle pieces.

3.2.4.1 Attributes

- graph
- score
- pieces
- solution_dimensions

3.2.4.2 Functional Requirements

- Class 4.1: SolutionNode
  This inner class of Solution shall store information about nodes in Solution's graph. The attributes shall be piece and piece_edge, which are Piece and PieceEdge objects. There shall also be an attribute edge_up containing an index representing which edge on the piece is intended to be displayed on top. There shall also be attributes x and y, representing the spatial location of the piece once placed in the solution.

  The constructor for this class shall initialize piece and piece_edge to the corresponding values passed. The variables edge_up, x, and y should be initialized to zero.

- $\langle constructor \rangle (pieces : PieceCollection, dimensions : (int, int))$
  This constructor shall set the object variables pieces and solution_dimensions according to those passed to it. The score variable shall be initialized to zero.

  The object's graph shall be initialized by creating an adjacency list, containing a vertex for every PieceEdge object stored in the Piece objects in pieces. Each vertex should be a SolutionNode object, created by passing the specific Piece and PieceEdge objects to the constructor of SolutionNode.

- $crossover(other : Solution)$
  This function shall generate a new Solution object given information from this Solution object and the other object passed to the function. This new solution shall be valid, i.e. all edge placements in the graph will be valid, there will be no duplicate nodes in the graph, and all non-flat Piece object edges will be connected to others in the graph.

  The crossover between the two Solution objects shall add edges to the new solution based on the shared information in the two parent Solutions. If an edge in their graphs is contained in both solutions, this will be added to the new solution first. Otherwise, new edges will be added to the graph based on comparisons between PieceEdge objects. New edges shall be added to the graph from a kernel, starting at a random vertex. This means that only edges reachable from the initial starting vertex shall be considered when adding new edges to the graph. Pseudocode describing the crossover process is shown below.

```
Input: other, Output: new_solution
// Initialize new Solution object and kernel
new_solution ← Solution(pieces, solution_dimensions)
kernel ← emptyset, available_pieces ← set(pieces.pieces)
/* Choose a random vertex, add its edges to kernel with $x$,
   $y$ at the origin                                          */
$v$ ← choice(graph.vertices)
for $e$ in $v$.piece.edges do kernel.add(($v$.piece, $e$))
available_pieces -= $v$.piece
/* while there are available pieces left, find the best
   edge and add it to the new solution's graph               */
while $len$(available_pieces) $> 0$ do
    // get the best edge to add next
    $(v1, v2)$ ← getNextEdge(available_pieces, kernel, other)
    // If no valid edges, break
    if $v1$ = None then break
    // Update, edge_up, x, and y values for $v2$
    $d$ ← ($v1$.edge.number - $v1$edge_up) mod 4
    if d mod 2 = 1 then  $v2$.edge_up ← ($v2$.edge.number + $d$) mod 4
    else  $v2$.edge_up ← ($v2$.edge.number + $d - 2$) mod 4
    if $v2$.edge_up mod 2 = 0 then $v2$.y = $v1$.y + $v2$.edge_up - 1)
    else  $v2$.x = $v1$.x - ($v2$.edge_up - 2)
    // Add the edge to new_solution
    new_solution.graph[$v1$].add($v2$)
    new_solution.score += $v1$.edge.compare($v2$.$edge$)
    /* Update the kernel to include available edges from
       $v2$.piece and not $v1$.edge                            */
    kernel -= ($v1$.piece, $v1$.edge)
    for $e$ in $v2$.piece.edges do
        if $(v1, v2)$ not in new_solution.graph.edges then
            kernel.add(($v2$.piece, $e$))
        end
    end
    available_pieces -= $v2$.$piece$          // Update available_pieces
end
return new_solution
```

❖ $getNextEdge(available : set, kernel : set, other : Solution, new\_solution : Solution)$
This function shall find the best next edge to add to the graph given the list of available pieces to connect to, and a kernel of possible edges to connect from. This function shall prioritize based on presence in both this object's graph and the other Solution object's graph. If there are no edges present in both graphs, the lowest weight edge extending from kernel will be added. This function shall return a tuple of SolutionNode objects representing an edge. Pseudocode showing this search is shown below:

```
Input: available, kernel, other, new_solution
Output: new_edge
/* Loop over all possible edges, keeping track of minimum
   cost edge found                                      */
min ← infinity
minEdge ← (None, None)
for piece in available_pieces do
    for edge in piece.edges do
        for k in kernel do
            $v1$ ← new_solution.graph.vertices where (v.piece, v.edge) =
              (k.piece, k.edge)
            $v2$ ← new_solution.graph.vertices where (v.piece, v.edge) =
              (piece, edge)
            cost ← edge.compare(k.edge)
            if cost < min then  min, minEdge ← cost, $(v1, v2)$)
            // If edge is invalid, continue
            if cost = infinity or not isValid($v1, v2$, new_solution) then
                continue
            end
            // If edge in both solutions
            if $(v1, v2)$ in graph.edges and in other.graph.edges then
                return $(v1, v2)$
            end
        end
    end
end
/* If there were no edges that were in both of the graphs,
   return the lightest weight edge extending kernel      */
return minEdge
```

❖ $isValid(v1 : SolutionNode, v2 : SolutionNode, new\_solution : Solution)$
This function shall return whether or not the edge between $v1$ and $v2$ would be valid to add to new_solution.graph. This edge would be invalid if the bounds of the x and y coordinates of each piece were wider than solution_dimensions. If this is the case, then the function shall return False. Otherwise, it shall return True.

Pseudocode for this function is shown below:

```
Input: $v1, v2$, new_solution
Output: is_valid
/* Edges adjacent to the edges in question should line up
   if flat.  If not, this is not a valid edge to add.    */
adj1left, adj1right $\leftarrow$ adjacent edges to $v1$.edge in $v1$.piece
adj2left, adj2right $\leftarrow$ adjacent edges to $v2$.edge in $v2$.piece
if adj1left.label = "flat" xor adj2right.label = "flat" then
    return False
end
if adj1right.label = "flat" xor adj2left.label = "flat" then
    return False
end
/* Finds the total height and width of nodes reachable from
   the initial state, specifically looking at nodes that
   have piece type ``middle"                               */
if not $v2$.piece.type = "middle" then return True
middles $\leftarrow$ [new_solution.graph.vertices where piece.type = "middle"]
$w \leftarrow abs(max(\text{middles.x}) - min(\text{middles.x}))$
$h \leftarrow abs(max(\text{middles.y}) - min(\text{middles.y}))$
/* If two middle pieces are more than the puzzle height -
   2, or width - 2 away from each other, then the final
   solution will exceed the bounds                         */
max_w, max_h = solution_dimensions - 2
// Finds the direction that the new node will be added in
$d \leftarrow$ ($v1$.edge.number - $v1$edge_up) mod 4
// If up, down and adding will exceed bounds
if $d$ mod 2= 0 and $h + 1 >$ max_h then return False
// If left, right and adding will exceed bounds
if $d$ mod 2= 1 and $w + 1 >$ max_w then return False
// If none of these things have returned False
return True
```

- *randomize(void)*
  This function shall initialize the graph of this object to be a random assortment of pieces that still make up a valid layout. This function shall operate in exactly the same way as crossover( ), except when choosing an edge to add, a random choice is made from the available edges until a valid edge is chosen.

- *getSolutionImage(void)*
  This function shall return an image containing the sub-images from each Piece, laid out in an arrangement given by the object's graph. This means that if there is an sub-image next to another in the final image, then there should be an edge

between the corresponding vertices in the graph.

This function shall simply find the extreme x and y values in graph.vertices, loop over all possible x, y in those bounds, and find a vertex that has those specific coordinates. The function getSubimage should be called on each piece from each vertex, passing edge_up to the function call. This sub-image should be stored in an array of other such images, at an index specified by x and y. After all sub-images have been found, the sub-images in each row of the array can be combined together using numpy's hstack function. Then, the resulting images for each row can be combined into the final image using numpy's vstack function. Finally, this image can be returned.

3.2.4.3 Messages

Solution objects are created in the puzzleSolver class, which initializes the values passed into the constructor. The Solution class uses information from the pieceCollection class, the piece class, and the edge class in its functions and attributes.

3.2.5 Class 5: Puzzle Solver

The Puzzle Solver class shall contain all information required to perform a genetic algorithm, which shall result in an estimated optimal solution for the puzzle.

3.2.5.1 Attributes

- pieces
- dimensions
- num_generations
- generation_size
- generation_counter
- solutions

3.2.5.2 Functional Requirements

- $\langle constructor\rangle(pieces : PieceCollection, puzzle\_size\_inches : (float, float),$ $num\_generations : int, generation\_size : int)$
  This constructor shall set object variables to those passed to the constructor. The object's variable dimensions shall be initialized by calling the function findDimensions, passing puzzle_size_inches to the function. The variable generation_counter shall be initialized to zero. Then, the constructor shall call the function initializeFirstGeneration.

- *findDimensions*(*puzzle_size_inches* : (*float*, *float*))
  This function shall use the dimensions variable to calculate an estimate on the number of pieces in each row and column of the puzzle. The variable dimensions shall be set to a tuple containing the estimated number of pieces in each row and column. Pseudocode that estimates these values is shown below.

```
Input: puzzle_size_inches
num_pieces ← len(pieces.pieces)
w_in, h_in ← puzzle_size_inches
/* Assuming the pieces are relatively square, the ratio
   between the number of pieces in rows and columns should
   approximately equal the ratio between the puzzle width
   and puzzle height in inches, so we want to minimize the
   difference between these ratios, while maintaining the
   number of pieces                                      */
min_difference ← infinity
min_w ← 0
min_h ← 0
for w in 1 . . . num_pieces do
    h ← num_pieces / w
    // If h is not an integer, continue
    if h != floor(h) then continue
    // Find the difference of ratios
    difference ← abs((w/h) − (w_in/h_in))
    // Update min
    if difference < min_difference then
        min_difference ← difference
        min_w ← w
        min_h ← h
    end
end
dimensions ← (min_w, min_h)
```

- *initializeFirstGeneration*(*void*)
  This function shall initialize the variable solutions to be an array of generation_-size random solutions. This function shall create a new Solution object, call the function randomize on this object, and then append the object to an array. This should be done the number of times specified by generation_size, starting with an empty array. The variable solutions should be set to this array once all objects have been appended.

- *doOneGeneration*(*void*)
  This function shall generate an array of new solutions by calling the crossover function from the Solution class using two random Solution objects. Then, the

solution variable shall be set to this new array and the generation counter shall be incremented. The scores from each Solution object shall be used to weight the random choice of objects used in the crossover. Pseudocode that performs this operation is shown below.

```
new_solutions ← [ ]
scores ← [$s$.score for $s$ in solutions]    // scores for each solution
/* Normalize and invert, so that objects with low scores
   are more likely to be chosen.  Scores are based on total
   difference, so lower is better.                          */
scores ← 1 - normalize(scores)
/* Get random solutions, using weights, append new
   solutions to solution array, stop once enough have been
   found                                                    */
while $len$(new_solutions) < $len$(solutions) do
    $s1$ ← choice(solutions, weights=scores)
    $s2$ ← choice(solutions, weights=scores)
    new_solution ← $s1$.crossover($s2$)
    new_solutions.append(new_solution)
end
solutions ← new_solutions
generation_counter += 1
```

- ❖ *doRemainingGenerations(void)*
  This function shall repeatedly call the function doOneGeneration until generation_-counter is equal to num_generations.

- ❖ *getBestSolution(void)*
  This function shall return the Solution object in solutions that has the highest score value. This should be done by looping over every Solution object in solutions, keeping track of the minimum score and its object seen so far, and returning the object with the minimum score at the end.

3.2.5.3 Messages

A PuzzleSolver object is created in the main class. The puzzle solver class passes on piece information to the Solution class, and does not directly interact with any other classes.

3.2.6 main

This section will describe the required functionality of the main part of the software's code. This will be all code that is directly called from the main function, without passing through other objects. In python, this can be thought of as any code that will be called inside of “if __name__ == “__main__:””.

To begin, a PieceCollection object should be created. This object will initially have only default values for all of its variables.

Next, a Tk object from Tkinter shall be created, which will control our UI. This object may be configured in any way but should have its title set to be “Puzzle Solver” or similar. A button should be added prompting the user to upload their images. This button should call filedialog.askopenfilename from Tkinter if pressed. Once a file is selected, the filename shall be stored in an array. A pop-up window shall appear, with a text field available for the user to input the number of pieces in their image. This number should be stored in an array as well.

Another button shall be included with text asking the user if they are done uploading files. If this is pressed, all filenames stored thus far shall be opened using openCV’s imread function. The function addPieces shall be called on the PieceCollection object, passing the image from imread and the corresponding number of pieces stored previously.

Once all images have been read and their pieces have been stored in PieceCollection, the user should be prompted to enter the puzzle dimensions in inches. There should be text describing that the dimensions are found on the puzzle’s box. After this is done, a PuzzleSolver object shall be created, with the PieceCollection object and the puzzle dimensions passed to the constructor. The number of generations and generation size may be fixed at 30 and 1000, respectively. The user may also be prompted to enter other numbers for these variables, but with the default options shown.

The function doRemainingGenerations shall be called on the PuzzleSolver object. Then, the function getBestSolution() shall be called on the object. This will return an image array. The Tk object should be updated to include this image. The user shall be given the option to save the image, and be presented with a file explorer UI to select a location. After this, openCV’s imwrite function should be used to save the image at the selected location. Once the user closes the Tk window, the program shall end.

3.3 Performance Requirements

This specification gives no requirements on performance as performance is determined by both the size of the puzzle being solved and the computing ability of the host machine.

3.4 Design Constraints

Our current requirements do not specify any design constraints, as there are no relevant limitations, standards, or regulations put on our software.

3.5 Software Systems Attributes

3.5.1 Reliability

The system is reliable as it is available to be run as an executable whenever the user requests. The application can be launched any time that the host operating system allows it. In the event of a crash, the progress of solving the puzzle will be lost.

3.5.2 Availability

The availability of the program is on-demand. It will, however, require as much time as it takes to completely solve the puzzle.

3.5.3 Security

As our software does not deal with personal information, and there are not any malicious applications of our software, security is not an issue.

3.5.4 Maintainability

The project will be exported as an executable and thus will not need to be changed to work as a dependencies update.

3.5.5 Portability

Our software is as portable as the language python. It avoids using any host-dependent software.

4 Further Information

4.1 Change Management Process

If customers want to ask for changes, they will submit a formal email to our group, who will then log and review it and decide upon a plan of action regarding the change requested.

4.2 Document Approvals

The contents of this document have been monitored by the instructors of COSC 3950 at the University of Wyoming. Their signatures are requested below.

Dr. Ruben Gamboa ....

Dr. Mike Borowczak ....