PUZZLE SOLVER

Kristjan Kõiv[1]
Joosep Näks[1]
[1]Institute of Computer Science, University of Tartu

UNIVERSITY OF TARTU
1632

Introduction

This project aims to create an algorithm for solving jigsaw puzzles. It takes an image of a shuffled puzzle as input and outputs the solved image. The image of the puzzle is taken from an online jigsaw maker [1], so the background is monochromatic, pieces don't overlap and are always oriented correctly.

The problem can be divided into several subproblems.

1. Finding the puzzle pieces
2. Finding the edges and corners of the puzzle pieces
3. Finding the colour gradient along the edge of a puzzle piece
4. Comparing how well edges from separate puzzle pieces fit together
5. Finding a permutation of pieces that minimizes the errors from adjacent edges
6. Aligning the pieces after the correct permuation has been found

Proposed solution

Input

The input to the algorithm is an image that contains all of the puzzle pieces. In order for our algorithm to work, the input image needs to have the following constraints: the background needs to be monochromatic, puzzle pieces need to have the same orientation and be non-overlapping.

Finding individual pieces and associated metadata

Given an input image with those constraints, the next step is to extract the individual pieces. Piece extraction is done in 3 steps.

1. First we use OpenCV's findContours function to extract a set of plausible contours that describe the individual pieces
2. The previous step produces some false positives, so a filtering step is needed. First we filter out contours that encompass an area significantly smaller than other contours. This step is trivial thanks to OpenCV's contourArea function. Secondly, we remove contours that are entirely contained by another contour. We do this by comparing the bounding boxes of the contours.
3. These contours are useful for analyzing the shapes of the pieces, but they are not good for describing the colour gradient of a puzzle piece's edge, because the pixels underneath this contour are mostly the colour of the background. To additionally get the colour gradient of a piece's edge we perform the following operations:
   (a) We want to get a contour that is similar to the original, but a few pixels "inward".
   (b) For that we first copy the contours to a separate image, dilate them and find contours from the dilated image. This will always produce two contours: one inner and one outer. We then take the inner of these two.
   (c) Later on we will want to use the original contour for shape and the inner contour for colours, so a mapping is needed between the two. For this we simply map each pixel from the outer contour to the nearest pixel on the inner contour.
4. It will also be useful to know which pixels from the contour are the pixels representing the 4 corners of the piece. The final process that we came up with was as follows:
   (a) First for each corner of the bounding box, we measure the L-1 distance to each point on the contour.
   (b) From these measurements we extract all local minima and consider the 3 lowest local minima.
   (c) Then we calculate the centre of mass for the puzzle piece and consider vectors from the centre of mass to the coordinates of the local minima.
   (d) We then take as the corner, the local minima whose associated vector has a direction most similar to that corners corresponding diagonal vector
5. To finally find the set of pixels that make up the entire piece, a DFS is ran starting from the middle of the piece (defined as the intersection of diagonals connecting the 4 corners found earlier) and ending when encountering a pixel from the contour set.

Classifying pieces

In order to combine the pieces into the original image, a few more assumptions are made about the puzzle. Specifically, it is assumed that the underlying image is a rectangle and that the jigsaw pieces roughly divide the image into n by m rows and columns. Under these assumptions, it is good to first classify the pieces as corner, edge, or middle pieces (and also which corner and which edge they are from). This is also similar to how many humans solve jigsaw puzzles. The process for determining whether a piece is a corner or an edge piece is as follows.

1. For each piece, the top, bottom, left and right edges of that piece are defined by the pixels along the contour from one corner to the next
2. For a horizontal edge, we then consider the standard deviation of the edge's pixel's y coordinates and analogously, for a vertical edge, we consider the standard deviation of the edge's pixel's x coordinates.
3. We then look at the standard deviations over all pieces for each edge direction and use a custom clustering algorithm to classify the edges with a small standard deviation as straight.
4. The global position of a piece (centre, edge or middle) can then be inferred from whether (and which of) it's edges are straight.

Difference metric

Since earlier we made the assumption that the individual pieces form a grid, the task of combining the pieces is essentially finding which row and column the piece is from. To estimate how well a piece fits into some position on the grid, a metric is needed to find the similarity of two edges from different pieces. We use dynamic time warping to calculate this metric.

We consider both the shape of a piece's contour and the colours along that contour. To put it another way, the dynamic time warping algorithm compares two 5-dimensional vectors of points, where each point is defined by it's distance from the start of the edge along both the x and y axis, as well as it's colour, with each colour channel being considered separately. Because the values for colours are significantly larger than the differences in location, we also introduce a term by which the location differences are multiplied and which can be used to control how sensitive the metric is to either shape or colour data.

Loss function minimization

The task of combining the jigsaw puzzle can now be restated as two subtasks: finding the values for the difference metric for all possible edge pairs and then combining the edges so that the sum of the metric for adjacent edges is minimal.

Calculating the metric for all possible edge pairs is embarrassingly parallel, so we decided to implement it on a GPU. With this parallel metric calculation, other parts of the solution became the bottleneck instead. Combining the pieces to minimize the sum of adjacent edges is analogous to TSP, in fact, for a 1 by m puzzle, it is exactly TSP. In the case of an n by m puzzle, the task is still similar to TSP, so metastrategies that work for TSP are expected to work here. We decided to use a greedy approach that first fixes all corners and then starts adding pieces adjacent to fixed pieces. At each timestep, it considers each piece that can go to any adjacent empty space and selects the piece and location combination that has the lowest metric value for the edges that connect it to already fixed pieces.

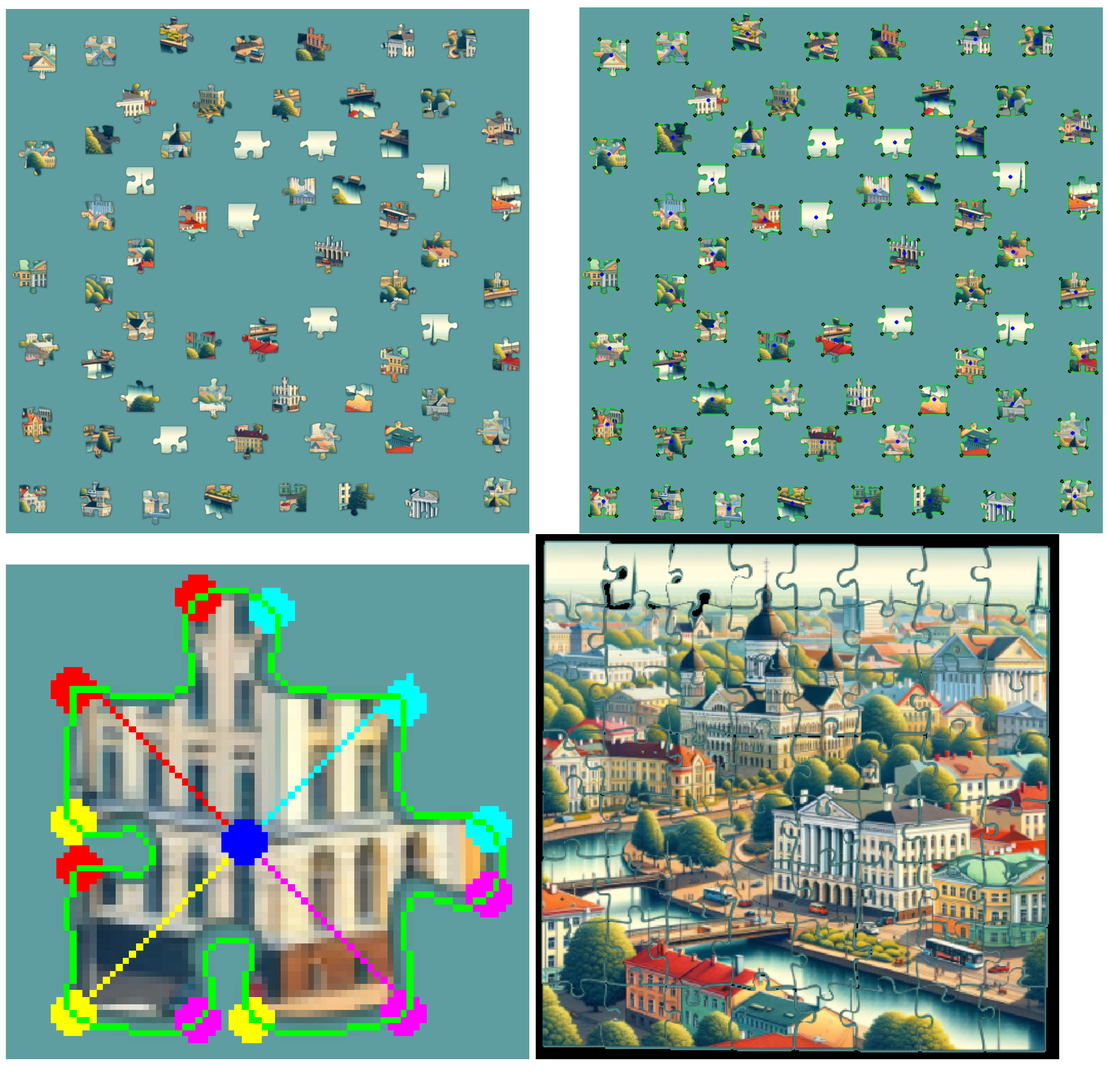
This solution exhibits interesting self-correcting properties as illustrated by a GIF in our repository [2]. The solution for the puzzle shown below places a piece at the top left incorrectly near the beginning, but because this mistake makes it hard to place additional pieces near it, the solution tends to first focus on other areas where it hasn't made a mistake. This stops any initial mistakes from propagating through the final solution.

Aligning the pieces visually

This part is trivial and left as an exercise to the reader.

Conclusions

We compared the solving times of various solutions and humans for the puzzle below. Humans were the slowest with 340 seconds, the CPU solution was slightly better at 300 seconds and the GPU solution was the fastest at 3.7 seconds.
Illustrated below are the various solution stages for an AI-generated [3] image of Tartu.



References

1. https://jiggie.fun
2. Link to repository: https://github.com/joosu77/puzzler
3. https://openai.com/dall-e-3