Use Of Jigsaw Puzzle Solving Algorithms In The Real World

Luca Sartore

May 2023

Contents

1 Abstract

The jigsaw puzzle problem has been in the eye of computer scientists for a while, and some clever solutions have already been found. These algorithms are made to work with a "digital" jigsaw puzzle F1, but there aren't papers (at least not popular enough to be searchable) that try to apply the solution to a "real world" jigsaw puzzle F2.
The problem has been tackled by some small projects. But, as said earlier, the process and eventual challenges has never been documented by a full paper, this wants to be the first.
As a bonus the paper will also cover the creation of a user friendly app that will be open source and free to use.

2 Introduction

2.1 Classification

This paper will focus on type 2 puzzles. A type 2 puzzle is a puzzle where the position, and the orientation of each piece is unknown.

2.2 Digital vs Real-World Jigsaw Puzzles

There is another important distinction between different types of puzzles. They can be divided into "digital" and "real world" jigsaw puzzles.

Figure F1: An example of a "digital" jigsaw puzzle

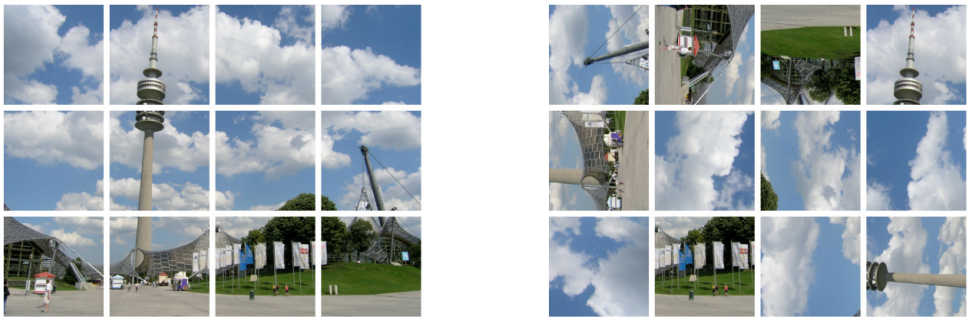
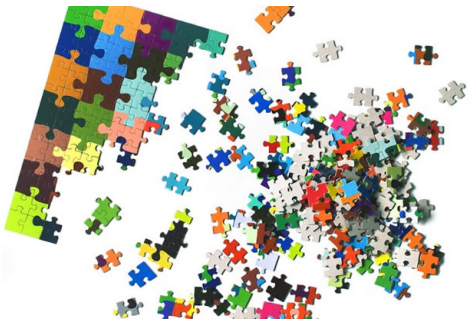


Figure F2: An example of a "real world" jigsaw puzzle



The reason this distinction is important is because, despite the generic concept of the puzzle not changing, obtaining accurate matches of a piece's characteristics is far easier with a digital puzzle, since there are far less things that can go wrong.

Figure F3: An example of what can go wrong when dealing with the real world

3 Previous Literature

This section will analyze 3 different algorithms that have been proposed as a solution of type 2 puzzles. The objective is to understand the strengths and the weaknesses of each one, to build up some knowledge that will be useful for the next sections.

3.1 General Structure Of The Algorithms

All the algorithms that will be analyzed are composed of 3 sub algorithms:

3.1.1 The Splitter:

This component takes as input one or more images containing all the pieces. It then split all the pieces from each other, 9and split each piece into his four sides.

3.1.2 The Comparator:

This component compares each side with all the others, in order to understand whether they match or not.
There are two distinct kinds of "Comparator" algorithms: The "Binary Comparison" and the "Non Binary Comparison". As the name suggests when comparing two sides with a "Binary Comparison" the result can either be 0 (they do not match) or 1 (they match). In contrast a "Non Binary Comparison" can give any value between 0 and 1. This allows states of uncertainty to be represented.

3.1.3 The Solver:

This component uses the information provided by the Comparator 3.1.2, and tries to find a solution (i.e. a position and an orientation for each piece) that is the most likely to be correct.

3.2 Solving Jigsaw Puzzles By The Graph Connection Laplacian [1]

This algorithm falls in the "binary comparison" category 3.1.2. And can be used to understand the strength and weaknesses of this approach.
The main benefit of this approach is speed, having a binary comparison allows some specific optimization, in particular the use of some graph theory techniques.
The algorithm has a time complexity of ruffly $O(N^2)$. Which is in line with the theoretical minimum for jigsaw puzzle solving, according to the study: "No easy puzzles: Hardness results for jigsaw puzzles [3]".
The negative aspect might be the accuracy, since by using only two states (match or not match) some informations are lost, compared to a non binary comparison. To quantify accuracy the paper used the "neighbors comparison" metric, which "calculates the percentage of pairs of image patches that are matched correctly".

3.3 A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles [2]

This paper applies a genetic algorithm to the jigsaw puzzle problem, with some promising results.
The algorithm seems to have a $O(N^2)$ time complexity, with time slightly lower than the previous example (But it is impossible to know for sure which one is faster, given that they didn't specify the hardware used).
The paper has used the same method to evaluate accuracy that the previous one used (neighbors comparison) so it is easy to compare them.
The algorithm falls in the "Non binary comparison" 3.1.2; this should give it an advantage, since it has more data to work with.
Unfortunately this advantage does not compensate for the worst precision of the algorithm itself, and the accuracy results are equivalent, if not slightly worse than the previous algorithm.
It is important to keep in mind that for the very nature of genetic algorithms, it might be possible that the accuracy would have been better if they had allowed it to run for more generations.

3.4 Computer Vision Powers Automatic Jigsaw Puzzle Solver [4]

The solutions 3.2 and 3.3 can be considered "state of the art" for digital jigsaw puzzles. But, as already introduced in section 2.2 and better explained in section TODO, appalling the solution to the real world can be challenging.
This solution is possibly a "state of the art" for real world puzzles.
In reality calling this a "solution" is incorrect, since all it does is find the best 8 matches and asks the user which one to place. Making the algorithm not autonomous.
Even ignoring this major detail, making an analysis of this solution has proven to be challenging, since the article is more of a publicity stunt than an academic paper. In fact, the code is not open source and the information about the time needed to solve the puzzle is missing.
The article only shows a single puzzle solved, with a 9×6 dimension.
Even with all of these problems this article still is one of the best example of a solution for real world puzzle.

4 The Objective Of This Paper

After reading some of the previous literature for this problem, some conclusions could be drawn: The problem of "digital jigsaw puzzle" has been solved many times, and is already well optimized.
On the other hand, the problem of "real world jigsaw puzzle" has mostly been ignored
by the academic community.

The objective of this paper is to fill this void, by answering the following questions:

- Can the current state of the art solutions for digital puzzles be used for real world puzzles, or the unpredictable nature of real world data makes it impossible?
- If the algorithms cannot be applied to the real world, what is a possible solution that is more redundant to imprecise data?

5 The Setup

To answer these questions, some data to start with, and an algorithm to analyze the data; in particular the necessary components are:

5.1 The Puzzle To Solve

The first step is finding a real world puzzle to solve; It has been decided to use a classic 500 piece puzzle, and use a printer's scanner to digitize the pieces.
The puzzle has been scanned in various sub-sizes to allow the testing of different configurations. A resolution

of 1200 ppi has been used to ensure a sharp and precise image of the piece, as seen in image F3. The puzzle has been scanned backward, since the comparator algorithm that has been used REF does not use colors, and having an uniform color would make splitting the pieces easier.

5.2 Splitting The Pieces

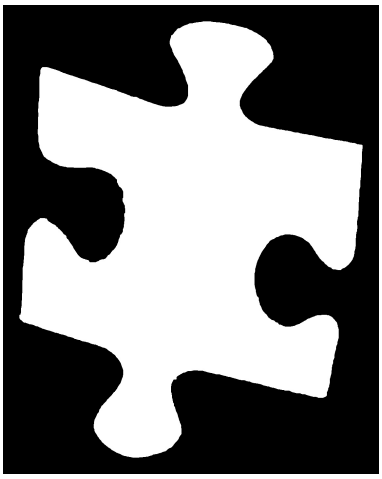
As introduced in section 3.1.1, the first step to solve a puzzle is to split an image composed of many pieces into individual ones. This is relatively straightforward, and can be done with the following steps:

- Apply a threshold and transform an RGB image into a binary mask.
- Find all individual Blobs (a Blob is just set of connected pixels with the same color) and for each one:
  - Calculate the area (the number of pixels the Blob is made of)
  - Add it to the set of pieces only if his area in within a certain range (to avoid small pieces of dust ending up counted as pieces)

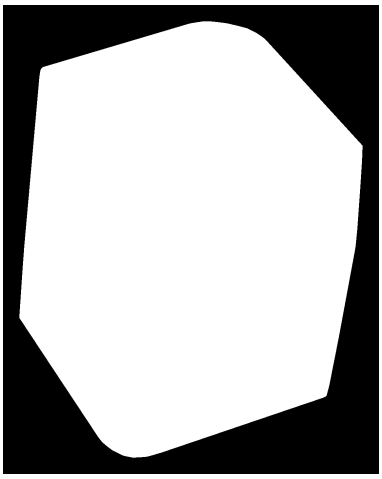
5.3 Splitting The Sides

The next step following the general template 3.1.1 is to split a pieces into his 4 sides, this could be done with the following steps:

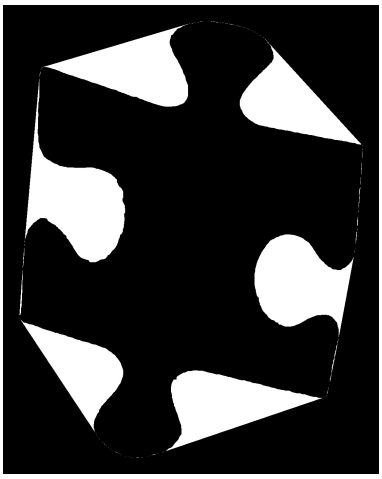
- **Remove the holes from the piece:**
  To do so the program computes the convex hull of the piece F4b, then calculates the difference between the convex hull and the original image, and obtains an image with the "filler areas" F4c. These filler areas have very different shapes, depending on what has generated them (a "hole" or a "knob" in the piece). So they can be classified, and added to the original piece only if they are generated by a hole, obtaining an image without holes F4d.
- **Remove the knobs from he piece**
  To do so the image without holes gets eroded F4e and then dilated F4f. This generates an image that resembles the original one, but has no knobs. Then the program finds the pixels that are white in the image without holes, but not in the image that has been eroded and dilated. This generates a set of Blobs F4g, but again, the shape of the blob is very different depending on what has generated it (an "angle" or a "knob"), so they can be classified. Then the program removes from the image with no holes all the pixels that are close to a Blob generated by a knob, obtaining an image without knobs and holes F4h.
- **Find the coordinates of the four corners**
  To find the coordinates of the corners, the program uses a tree steps approximation. It initially finds the minimum enclosing rectangle, and uses the coordinates of the corners of the rectangle as first approximation F4i.
  As a second approximation it takes, from the white pixels that are close to the first approximation, the pixel that has the most black pixels within a certain distance. Then it considers only the pixels that are within a certain distance of the second approximation, and finds the minimum enclosing triangle of sed pixels F4j.
  The corner of the triangle that is closest to the second approximation is the third and final approximation of the corner of the piece. The position of the four corners can be seen in image F4k.
- **Split each side using the previously found coordinates.**
  Once the program has found the four coordinates of the corners, to obtain each side. An example of a single side can be seen in image F4l.



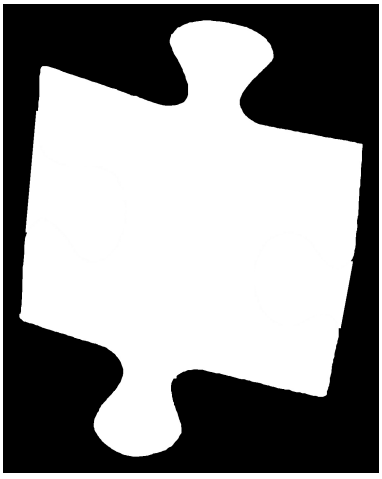
(a) Original binary mask of a piece.



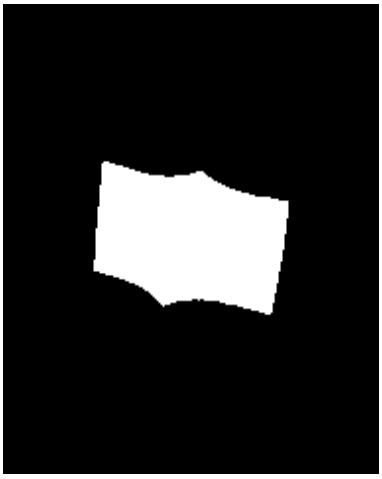
(b) Convex Hull of a piece.



(c) Difference: F4b − F4a.



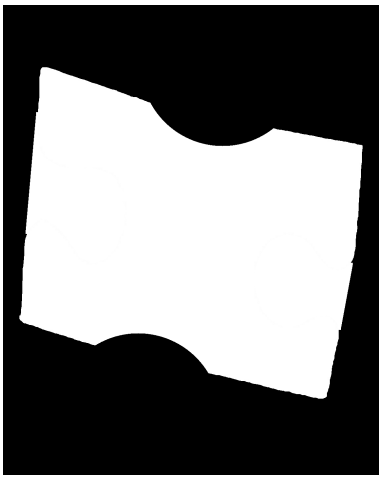
(d) Piece F4a with holes removed.



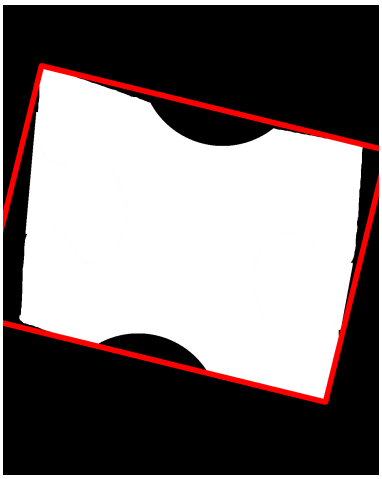
(e) Mask F4d eroded.

(f) Mask F4e dilated.

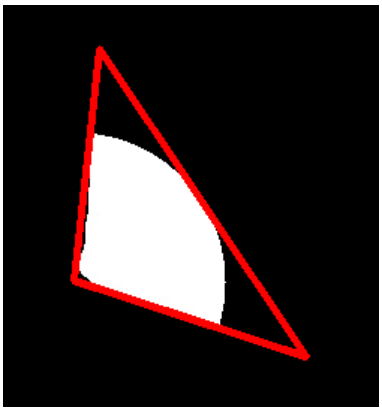
(g) Difference: F4d − F4e.



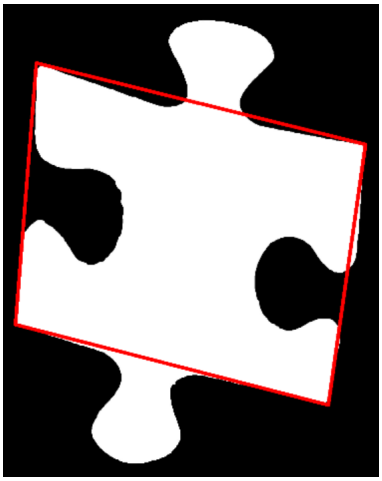
(h) Piece F4d with knobs removed.



(i) Min enclosing rectangle of F4h.



(j) Min enclosing triangle of the bottom right corner of F4h.



(k) The corners of the piece F4a.

(l) One of the sides of the piece F4a.

Figure F4: Steps needed to split an image into his four sides.

5.4 Comparing The Sides

The last step needed before it is possible to start working on a solver algorithm, is to compare the sides that have been generated by the previous step.
In order to generate a compatibility shore, the program compute the following steps:

- For each side, it considers only the border, and it makes the border thicker.
- It puts two sides attached to each other using the coordinates of the corners (In the same way a human would do to test if two pieces fit together).
- It calculates the "Or Area"; that is, the number of pixels that belong to the border of one side or the other.
- It calculates the "And Area"; that is, the number of pixels that belong to the border of both sides.
- It calculates the final shore, which is defined as: $Shore = \frac{AndArea}{OrArea}$, and goes from 0 to 1.

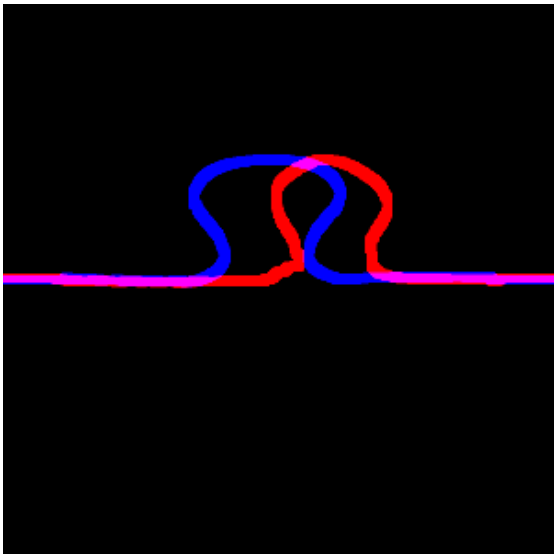


Figure F5:
In this image we can see two sides that have been compared, one side is drowned in red, the other one in blue. In this picture, the "Or Area" is represented by all non black pixels; The "And Area" is represented by all the purple pixels.

6 Apling Already Existing Algorithms To The Real World

Now that the setup is completed, it is possible to test the already existing algorithms with real world puzzles, to see if they can work. For this test, the fist algorithm 3.2 has been chosen, because the paper is full of detail. In particular it had some information about the accuracy of the algorithm in function to the corruption rate, which are essential for this analysis.

The tests that have been conducted have shown that our comparator algorithm 5.4 and the "Graph Connection Laplacian" solution 3.2 are not compatible with each other. A proof for this claim follows.

6.1 Proof

This section will provide a proof for the incompatibility of the two algorithms mentioned above.

The comparison algorithm deceived in section 5.4 outputs a compatibility shore that goes from 0 to 1.
But the algorithm 3.2 takes as input a binary value, which means that it is necessary to apply a threshold to the input data to convert them from float to bool.

6.1.1 Input Data

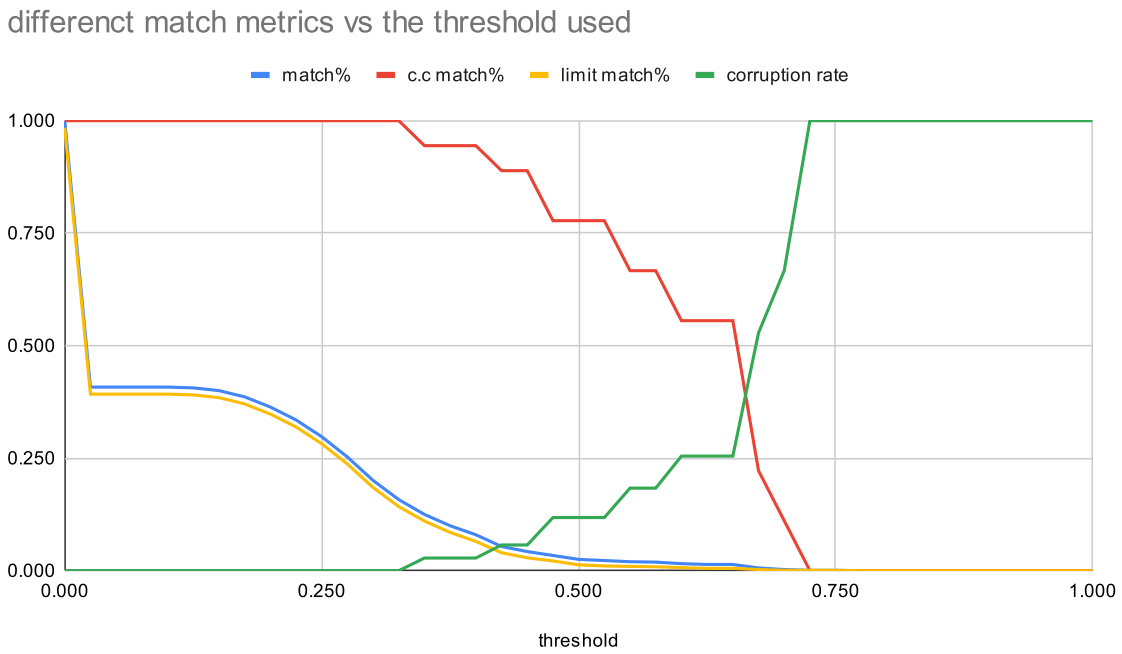
To find the optimal threshold a small 16 piece puzzle has been solved. And the following 4 values have been measured, in function of the threshold:

- **match%:**
  The probability that 2 random sides match.
- **correct combination match% (or c.c match%):**
  The probability that 2 sides match given that in the correct puzzle solution they actually are close to each other. ideally this should be 1.
- **limit match%:**
  Given that this measure has been taken on a 16 piece puzzle, the match% is highly influenced by the correct piece matching. With a bigger puzzle the match% would be lower. Limit match% is an approximation of what the match% would be in the limit for $N \to +\infty$ (where N is the number of pieces). This value is obtained by subtracting to match% the probability of choosing randomly the correct side, multiplied by the probability that the comparison returns true (which is c.c match%). ideally this should be almost zero.

$$limit\, math\% = math\% - \frac{c.c.\ mathc\%}{16 \times 4}$$

- **corruption rate:**
  Is defined as: $1 - \sqrt{c.c\, match\%}$. ideally this should be 0.

threshold	match%	c.c match%	limit match%	corruption rate
0.000	1.000	1.000	0.984	0.000
0.025	0.408	1.000	0.392	0.000
0.050	0.408	1.000	0.392	0.000
0.075	0.408	1.000	0.392	0.000
0.100	0.408	1.000	0.392	0.000
0.125	0.406	1.000	0.391	0.000
0.150	0.400	1.000	0.385	0.000
0.175	0.386	1.000	0.371	0.000
0.200	0.364	1.000	0.348	0.000
0.225	0.335	1.000	0.319	0.000
0.250	0.298	1.000	0.282	0.000
0.275	0.253	1.000	0.237	0.000
0.300	0.201	1.000	0.185	0.000
0.325	0.158	1.000	0.142	0.000
0.350	0.125	0.944	0.110	0.028
0.375	0.100	0.944	0.085	0.028
0.400	0.080	0.944	0.065	0.028
0.425	0.054	0.889	0.040	0.057
0.450	0.043	0.889	0.029	0.057
0.475	0.034	0.778	0.022	0.118
0.500	0.025	0.778	0.013	0.118
0.525	0.023	0.778	0.010	0.118
0.550	0.020	0.667	0.010	0.184
0.575	0.019	0.667	0.009	0.184
0.600	0.016	0.556	0.007	0.255
0.625	0.014	0.556	0.005	0.255
0.650	0.014	0.556	0.005	0.255
0.675	0.006	0.222	0.003	0.529
0.700	0.003	0.111	0.001	0.667
0.725	0.001	0.000	0.001	1.000
0.750	0.001	0.000	0.001	1.000
0.775	0.000	0.000	0.000	1.000
0.800	0.000	0.000	0.000	1.000
0.825	0.000	0.000	0.000	1.000
0.850	0.000	0.000	0.000	1.000
0.875	0.000	0.000	0.000	1.000
0.900	0.000	0.000	0.000	1.000
0.925	0.000	0.000	0.000	1.000
0.950	0.000	0.000	0.000	1.000
0.975	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	1.000



Assuming the algorithm has to solve a 500 piece puzzle, with a limit match% of 0.04 and a corruption rate of 0.057 (data taken from the line in red); what would this mean in terms of accuracy of the results?

6.1.2 Definitions

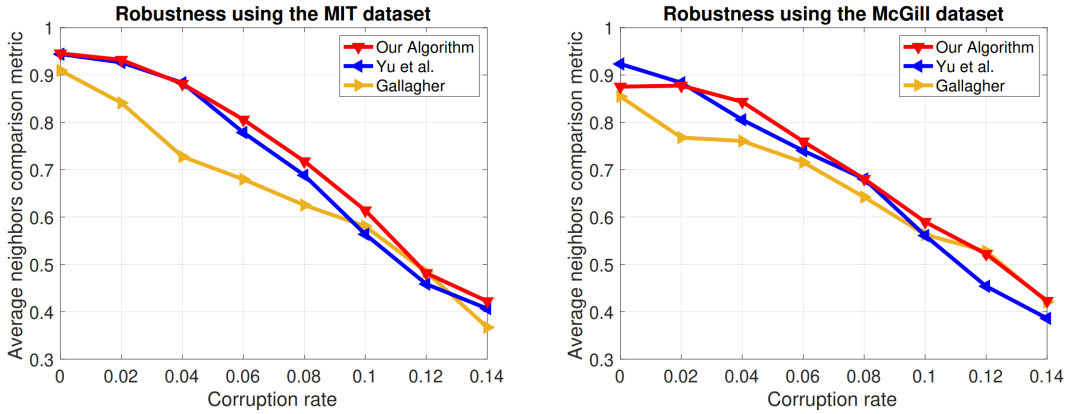
This set of definitions are useful to better understand the next section:

- **Connection:**
  When two sides of two different pieces are joined they form a connection. A solved puzzle with N pieces, (assuming it is squared) has around $2N$ total connections.
- **Possible Combinations:**
  A puzzle with N pieces can be put together in many different ways, (even if not all of them actually fit together). The total number of possible combinations is $N! \times 4^N$ where $N!$ represent all possible way to combinate the pieces in a particular position, and 4^N represents all possible orientations that each piece can have.
- **Possible Correct Combination:**
  Is one of the possible combinations where all the connections are considered a match by the comparing algorithm.
- **Actual Correct Combination:**
  An Actual Correct Combination is a Possible Combination that is also the correct solution of the original puzzle

6.1.3 Corruption Rate Analysis

From the graph F6, it can be observed that with a corruption rate of 0.057, a neighbors comparison value of about 80% can be expected. Which means that one in every 5 connections on average will be wrong. So in a puzzle of 500 pieces (which has approximately 1000 connections), 200 connections will be wrong on average. It is fair to say that this is not an ideal result.

Figure F6: Neighbors comparison vs Corruption rate of algorithm 3.2. Image taken from pater [1] page 34.



6.1.4 Limit Match% Analysis

As seen earlier, the limit match percentage is 4%.

Given that in a puzzle of N pieces there are around $2N$ connections, the probability of one random combination to be a possible correct combination is: 0.054^{2N}.

The total number of possible combinations are $N! \times 4^N$, which means that on average, on a puzzle with N pieces, there will be $0.054^{2N} \times N! \times 4^N$ possible correct combinations.

For $N = 500$ the expected number of possible combinations is around 1.5×10^{37}. Which makes the chance of finding the actual correct combination basically zero.

6.1.5 Conclusion

In this case both the match% and the corruption rate are bad enough to make a correct reconstruction impossible.

To lower the limit match% is necessary to increase the threshold, but this would increase the corruption rate.

To lower the corruption rate is necessary to decrease the threshold, but this would increase the limit match%.

This means that, regardless of the threshold chosen, our comparison algorithm 5.4 can not work with a state of the art solver 3.2.

6.2 Solutions For The Problem

There are two way to solve this deadlock:

6.2.1 Improve The Comparator

Improving the comparator can shortly be done. Some minor improvements can be made by spending some time tweaking the current algorithm. But it is clear that, for such a task, the optimal solution is probably to use machine learning.

6.2.2 Improving The Solver Algorithm

Different algorithms have different resilience to corrupted data. All the major algorithms that have been proposed since now, focus on speed. It is possible that an algorithm that focuses on accuracy over speed coud work with our comparator.

6.2.3 The Path Forward

Improving the comparator is probably the best long term solution, but it comes with a problem:
Manually testing if some pieces fit together, then making a scan, and finally label them to create a dataset would require hundreds if not thousands of hours.
On the other hand improving the solver is not a good solution long term, given that it will almost certainly have a worst time complexity, but could have some major short term benefit
In particular it does not require to create a huge dataset, and could be used to build one. In Fact it could solve hundreds of small puzzles, and automatically label the connections using the solution he created.

Given the two options, the author of this paper has chosen to focus on improving the solver algorithm. The choice was driven by the lack of the time needed to build a dataset, And the belief that creating a new algorithm, for then using that to build a dataset, would still be faster compared to building a dataset manually.

References

[1] Vahan Huroyan, Gilad Lerman and Hau-Tieng Wu, Solving Jigsaw Puzzles By The Graph Connection Laplacian, 2020. https://arxiv.org/pdf/1811.03188.pdf.

[2] Dror Sholomon, Omid David and Nathan S. Netanyahu, A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles, 2013. https://openaccess.thecvf.com/content_cvpr_2013/papers/Sholomon_A_Genetic_Algorithm-Based_2013_CVPR_paper.pdf.

[3] Michael Brand, No easy puzzles: Hardness results for jigsaw puzzles, 2015. https://www.sciencedirect.com/science/article/pii/S0304397515001607.

[4] AbtoSoftware, Computer Vision Powers Automatic Jigsaw Puzzle Solver, 2019. https://www.abtosoftware.com/blog/computer-vision-powers-automatic-jigsaw-puzzle-solver.