Use Of Jigsaw Puzzle Solving Algorithms In The Real World

Luca Sartore

May 2023

Contents

1 Abstract

This paper examines the Jigsaw puzzle problem, which has already been extensively studied in the literature. Several algorithms with efficient time complexity have been developed specifically for "digital" F1 jigsaw puzzles. However, the counterpart: the "real world' jigsaw puzzle, has received relatively little attention in the literature. While some small-scale projects have addressed this issue, we were unable to find any papers discussing it.

The puzzle problem can be divided into two separate components: the "Comparator" 3.1.2 whose objective is to determine whether two sides of two pieces fit together, and the "Solver" 3.1.3 whose objective is to use the information provided by the comparator to find the best way to fit some pieces together.

Solving the problem in the real world has proven to be more challenging compared to the digital realm. This is because all measurements in the real world are noisier than those in the digital world, making it challenging to obtain accurate data from the Comparator. The Solver poses its own set of challenges, as it is an example of a multiobjective optimization problem. Specifically, one objective is to have the fastest algorithm possible, while the other objective is to create an algorithm that performs well with imprecise data. The latter objective is particularly important when solving real-world puzzles.

In the first half of this paper, a Comparator is developed, and its accuracy is evaluated. The data are then used to demonstrate its incompatibility with the current state-of-the-art algorithm. At this juncture, two options remain to solve the problem: either create a superior Comparator or design a Solver that can operate with noisier data.

The most promising approach to improving the Comparator involves utilizing machine learning, but this requires a substantial amount of data. Conversely, creating a better Solver is a less straightforward task. However, if successful, it could potentially serve as a labeling tool to generate the dataset needed to train a superior Comparator.

The second part of this paper focuses on developing a new Solver algorithm specifically designed to work with noisy data. While the Solver created may not be considered state-of-the-art, as its time complexity ultimately ends up being worse than the current best algorithms, it could still play a fundamental role in solving the problem. As mentioned earlier, it could be employed to solve many small-sized puzzles and use the results to train a better Comparator.

In conclusion, this paper outlines the additional challenges that the real-world puzzle problem presents compared to its digital counterpart. Additionally, it lays the groundwork for a superior machine learning-based Comparator.

2 Introduction

2.1 Classification

This paper will focus on Type 2 puzzles. A Type 2 puzzle is one where the position and orientation of each piece are 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" F1 and "real world" F2 jigsaw puzzles. The difference between the two is quite self explanatory, and can be observed in the respective pictures.

Figure F1: An example of a “digital” jigsaw puzzle

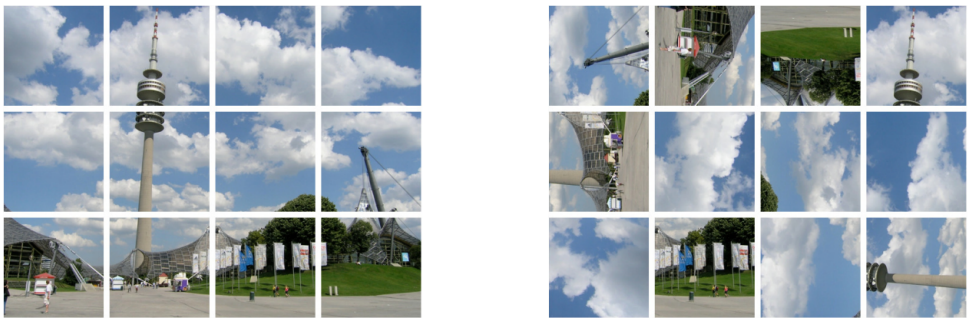
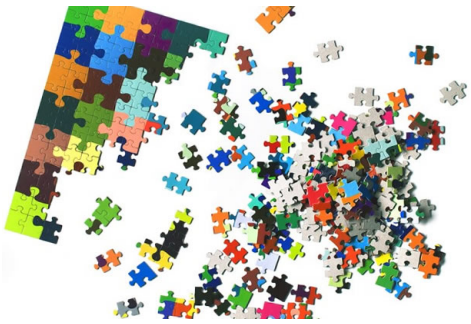


Figure F2: An example of a “real world” jigsaw puzzle



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 fewer things that can go wrong. For example, a piece’s shape may be distorted by a low-quality cut, as seen in picture F3

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

3 Previous Literature

This section will analyze three different algorithms that have been proposed as solutions for type 2 puzzles. The objective is to understand the strengths and weaknesses of each one, in order to build up 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 separates all the pieces from each other, and 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 utilizes the information provided by the Comparator 3.1.2, and attempts 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 within the "binary comparison" category 3.1.2. And can be used to understand the strength and weaknesses of this approach.
The primary advantage of this approach lies in its speed; the utilization of binary comparison permits specific optimizations, particularly leveraging graph theory techniques.
The algorithm exhibits a time complexity of approximately $O(N^2)$, a value consistent with the theoretical minimum for jigsaw puzzle solving, as outlined in the study "No easy puzzles: Hardness results for jigsaw puzzles [3]".
However, a potential drawback may arise in terms of accuracy. This stems from the fact that, with only two states (matching or not matching), certain details may be lost when compared to a non-binary comparison.
To quantify accuracy the paper used the "neighbors compariso" metric, which "calculates the percentage of pairs of image patches that are matched correctly". This will be useful later.

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

This paper employs genetic algorithms to address the jigsaw puzzle problem, yielding promising results.
The algorithm seems to have a $O(N^2)$ time complexity, with execution time slightly lower than the previous example (However, it remains uncertain which one is faster, as the papers did not specify the particular hardware configuration used).
The paper adopts the same evaluation method for accuracy as the previous example, namely neighbor comparison, facilitating direct comparison between the two approaches. This algorithm falls under the category of "Non binary comparison" 3.1.2 a distinction that theoretically grants it an advantage due to the increased granularity of input data. 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 the algorithm to run for more generations.

3.4 Computer Vision Powers Automatic Jigsaw Puzzle Solver [4]

Solutions 3.2 and 3.3 can be considered "state-of-the-art" for digital jigsaw puzzles. However, as already introduced in section 2.2, translating these solutions to the real world can be challenging.
This article represents one of the most comprehensive and well-documented implementations for real-world puzzles. However, it's important to note that referring to this approach as a "solution" may be misleading, as it primarily identifies the top eight matching pieces for a specific point and subsequently relies on user

input to make the final placement decision. This aspect renders the algorithm non-autonomous..
Even disregarding this significant detail, conducting an analysis of this solution has proven to be challenging, as the article leans more towards a publicity stunt than an academic paper. In fact, the code is not open source, and informations regarding the time needed to solve the puzzle are absent. The article only showcases a single puzzle solved, with dimensions of 9×6.
Despite these limitations, this article still stands as one of the best examples of a solution for real-world puzzles.

4 The Objective Of This Paper

After a review of the existing literature on this problem, some conclusions could be drawn: The "digital" jigsaw puzzle dilemma has been extensively addressed and is already well-optimized. Conversely, the "real-world" counterpart has largely been overlooked within the academic community.

The first objective for this paper is to fill this void, and been the first formal paper to analyze the real-world jigsaw puzzle problem.

The second objective is to use the forthcoming research to create a labeling tool.

This objective is motivated by the fact that the biggest challenge for real world puzzles is creating a reliable comparator 3.1.2, that could provide accurate information even with noisy input images, that are obtained from real-world measure.

It is easy to see how a machine learning based comparator would probably be the best solution, but this poses problems, in fact having to manually scan and label thousands of individual puzzle pieces to train the model would take ages. Instead an automatic tool would allow to scan a small 4×4 puzzle in one shot, solve it, and return the labeled data in just minutes.

5 The Setup

To fulfill the objectives of this paper, it is essential to have the necessary data for analysis. This section will encompass the input data utilized, as well as the Splitter 3.1.1 and Comparator 3.1.2 algorithms.

5.1 The Puzzle To Solve

The first step involves selecting a real-world puzzle for solving. It has been decided to utilize a classic 1000-piece puzzle measuring 66cm x 50cm. A printer's scanner is employed to digitize the individual pieces.

The puzzle has been scanned in various sub-sizes to facilitate the testing of different configurations. A resolution of 1200 pixels per inch (ppi) has been employed to ensure a sharp and precise depiction of the pieces, as seen in image F3. The puzzle has been scanned backward, using a black background.

This choice has been dictated by the fact that the scanner used has the flashlight slightly off-centered from the scanner's sensor. And this causes a shadows to be cast on one side of the pieces. A black background has been chosen to solve the problem of the shadow. And then the pieces has been flipped backward, since many of them where black, and it was difficult to separate them from the background. In the future it would be possible to scan the pieces in the correct orientation, by using a different scanner.

5.2 Splitting The Pieces

As introduced in section 3.1.1, the initial stage in puzzle solving involves splitting 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 defined as a 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

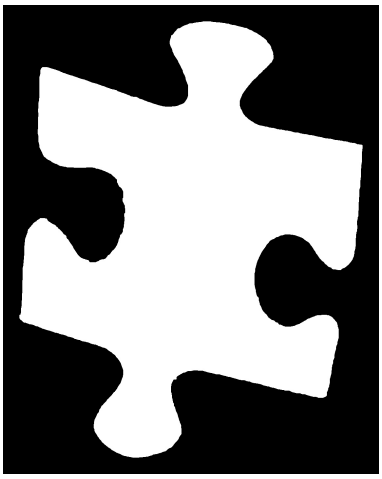
The next step following the general structure 3.1.1 is to split a piece into his 4 sides, this could be done with the following steps:

- **Remove the holes from the piece:**
  To remove the holes from the piece, the program employs a three-step process. Initially it computes the convex hull of the piece F4b. Subsequently it calculates the difference between the convex hull and the original image, resulting in an "image with filler areas" F4c. It can be noted that these filler areas have distinct shapes, depending on what has generated them (either a "hole" or a "knob"). Therefore they can be categorized. Finally the filler areas are added back to the original only if they originated from a hole. This results in an image where the holes have been filled F4d

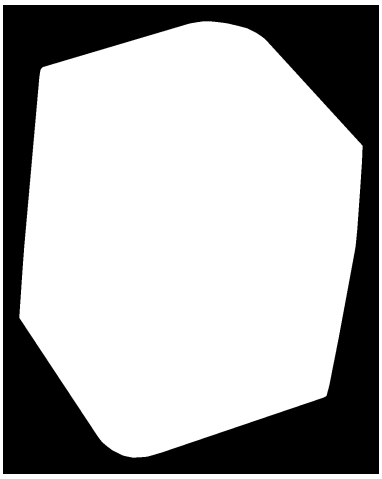
- **Remove the knobs from he piece:**
  In order to remove the knobs from the piece, the image without holes undergoes an erosion F4e followed by a dilation F4f. This process results in an image that resembles the original, but without any knobs. Subsequently, the program identifies the pixels that are white in the image without holes F4d, but not in the eroded and dilated version F4f. This leads to the formation of a set of blobs F4g. It is important to note that the shape of a blob can vary significantly depending on whether it originated from an "angle" or a "knob" and thus they can be classified accordingly. Following this, the program proceeds to eliminate from the hole-free image F4d all pixels that are in close proximity to a blob generated by a knob. This operation yields an image that is both devoid of knobs and holes F4h.

- **Find the coordinates of the four corners:**
  In order to determine the coordinates of the corners, the program employs a three-step approximation method. Initially, it identifies the minimum enclosing rectangle and utilizes the coordinates of its corners as the first approximation F4i.
  For the second approximation, the program selects, from the white pixels in close proximity to the first approximation, the pixel that contains the highest number of black pixels within a specified range. 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 nearest to the second approximation serves as the third and final approximation for the corner of the puzzle piece. The position of the four corners can be seen in image F4k.

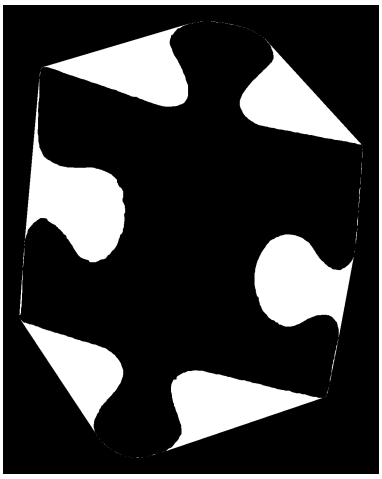
- **Split each piece into four sides using the previously found coordinates:**
  After the program has successfully identified the four corner coordinates, it can straightforwardly derive each of the four sides. An example of a single side can be seen in image . F4l.



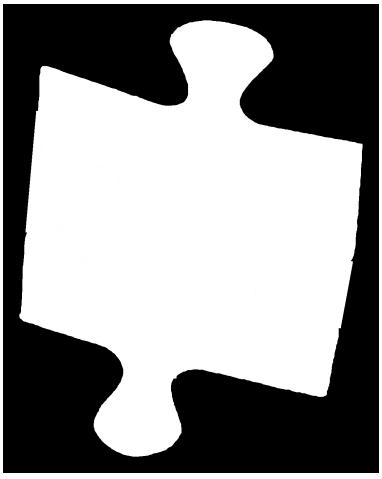
(a) Original binary mask of a piece.



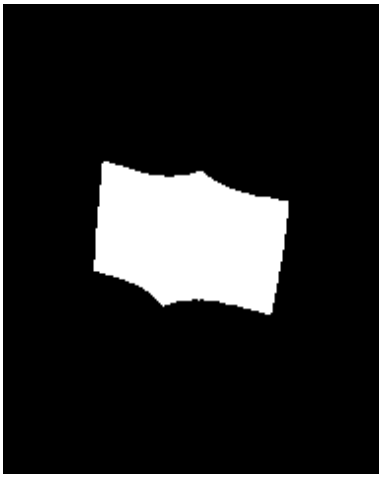
(b) Convex Hull of a piece.



(c) Difference: F4b − F4a.



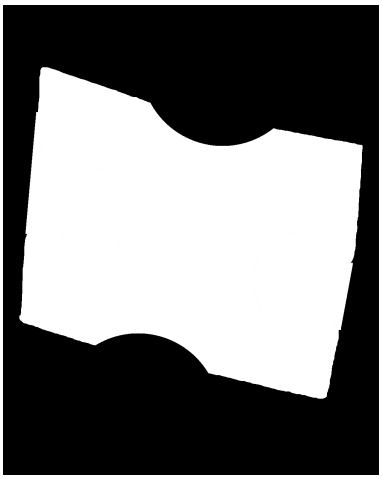
(d) Piece F4a with holes removed.



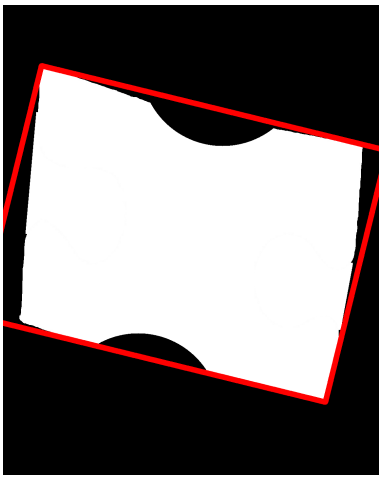
(e) Mask F4d eroded.

(f) Mask F4e dilated.

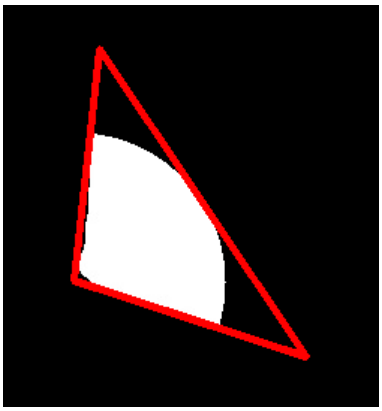
(g) Difference: F4d − F4e.



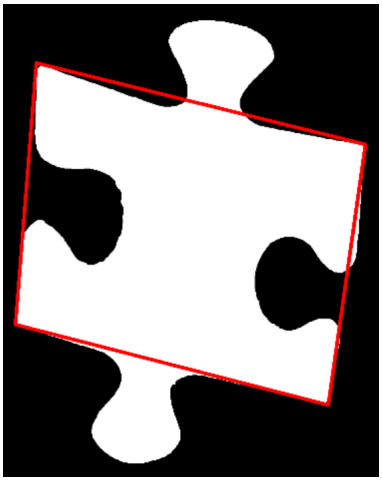
(h) Piece F4d with knobs removed.



(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 in the setup, is to compare the sides that have been generated before. In order to generate a compatibility shore, the program compute the following steps:

- For each side, it considers only the border, and makes it 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.

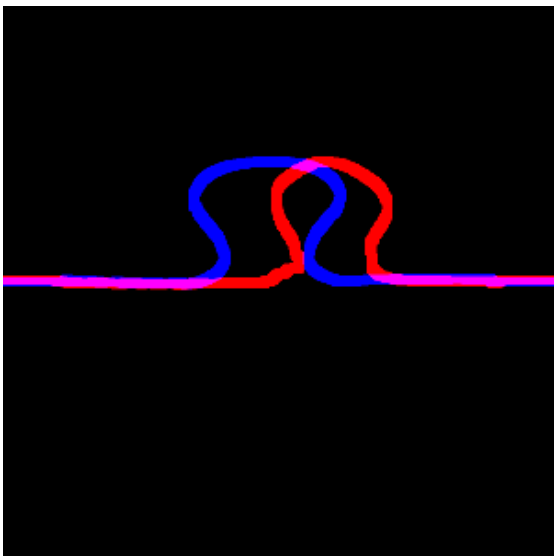


Figure F5:
In this image we can see two sides that have been compared, one side is drowned in red, the other one in blue. In this picture, the "Or Area" is represented by all non black pixels; The "And Area" is represented by all the purple pixels.

6 Applying Already Existing Algorithms To The Real World

With the initial setup completed, it is now possible to test the existing algorithms with real-world puzzle. For this test, the fist algorithm 3.2 is selected due to the comprehensive details provided in the paper. Notably, it contains information regarding the algorithm's accuracy in relation to the corruption rate, which is crucial for this analysis.

The conducted tests reveal that the Comparator developed in this paper 5.4 and the "Graph Connection Laplacian" solution 5.4 are not compatible with each other. A formal proof supporting this assertion is presented subsequently.

6.1 Proof

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:

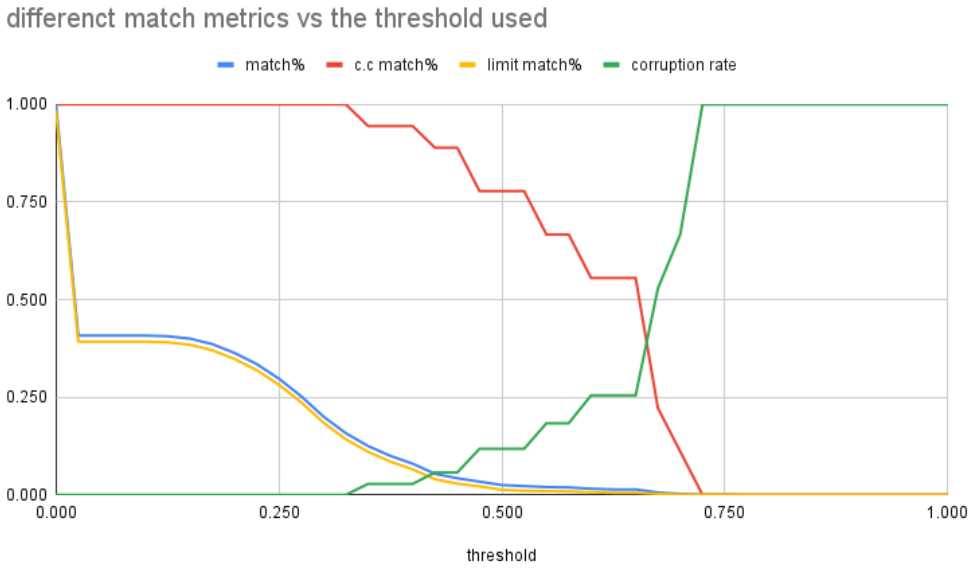
- **match%:**
  The probability that 2 randomly selected sides matches.
- **correct combination match% (or c.c match%):**
  The probability that 2 randomly selected sides matches given that in the correct puzzle solution they are actually close to each other. Ideally this should be 1.
- **limit match%:**
  Given that this measures has been taken on a 16 piece puzzle, the match% is highly influenced by the correct piece matching together. With a bigger puzzle the match% would be lower (Given that the amount of correct couple of sides scales with N, while the amount of possible couple of sides scales with N^2. Where N is the number of pieces).
  Limit match% is an approximation of what the match% would be in the limit for $N \to +\infty$. 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 limit match% 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

Figure F6: match metrics vs threshold used



Now, let's consider Algorithm 3.2 tasked with solving a 500-piece puzzle using our Comparator 5.4. A threshold of 0.425 is chosen; This implies a limit match% of 0.04 and a corruption rate of 0.057. What would this mean in terms of accuracy of the results?

6.1.2 Definitions

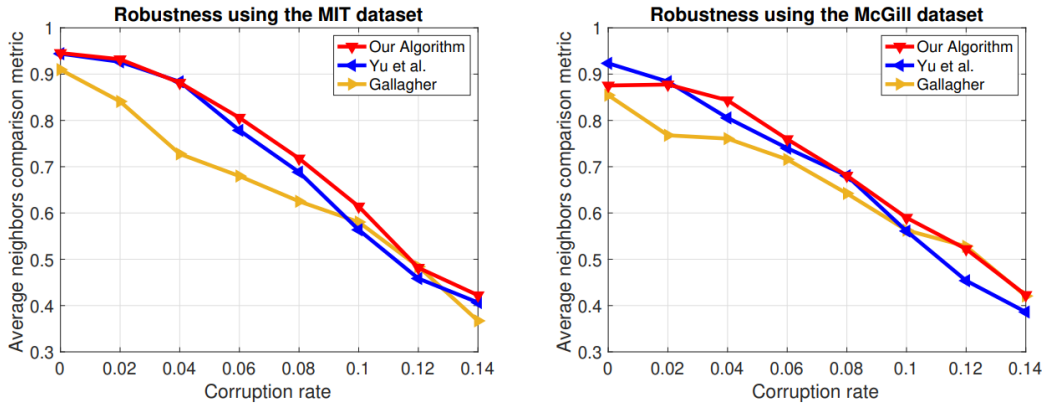
This set of definitions are useful to better understand the next section:

- **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

In Figure F7, it can be observed that at a corruption rate of 0.057, a neighbors comparison metric of roughly 80% can be anticipated. This implies that, on average, one out of every five connections will be incorrect. In the context of a puzzle comprising 500 pieces (equating to approximately 1000 connections), an estimated 200 connections will be inaccurate on average. It is reasonable to assert that this outcome is suboptimal.

Figure F7: Neighbors comparison vs Corruption rate of algorithm 3.2. Image taken from pater [1] page 34.



6.1.4 Limit Match% Analysis

Let's remember that $limit\_match\% = 0.04$.

Given that in a puzzle with N pieces there are around $2N$ connections, the probability of one random combination to be a possible correct combination is: 0.04^{2N}.

The total number of possible combinations are $N! \times 4^N$, which means that on average, in a puzzle with N pieces, there will be $0.04^{2N} \times N! \times 4^N$ 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 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 viable approaches to overcome this challenge. It is either necessary to enhance the Comparator while continuing to utilize the state-of-the-art solver, or it is possible to re-design the solver algorithm and create one specifically designed to operate with noisier data.

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. However, it is evident that for a task of this nature, the most viable solution would likely involve leveraging machine learning techniques.

6.2.2 Improving The Solver Algorithm

The puzzle dilemma is a "multiobjective optimization problem", where one objective is optimizing the time complexity, and the other is optimizing the accuracy of the algorithm in the presence of corrupted data. All the major algorithms proposed up to this point have prioritize speed. However, it is conceivable, and even probable, that an algorithm prioritizing accuracy over speed could prove effective with our comparator

6.2.3 The Path Forward

Improving the comparator is likely the most effective long-term solution, but it presents a significant challenge: manually testing if certain pieces fit together, followed by scanning and labeling, is a time-intensive process that could require hundreds, if not thousands, of hours.

On the other hand, improving the solver may not be the optimal long-term solution, as it is likely to result in a worse time complexity. However, it could yield significant short-term benefits.

Notably, this approach does not necessitate the creation of an extensive dataset and has the potential to facilitate its development. In fact, it could have the capability to solve numerous small puzzles and subsequently auto-label the connections based on the solutions it generates.

Given the two options, it has been decided to focus on improving the solver algorithm. This choice was driven by the belief that creating a new algorithm, and subsequently using it to generate a dataset, would still be a more time-efficient approach compared to manually constructing a dataset.

7 The Solver

The Solver that has been developed is not the primary focus of this paper and cannot be considered state-of-the-art when compared to the current best digital puzzle solvers. However, it may be considered as such when compared to existing "real-world" puzzle solvers.

7.1 Overview

The algorithm itself is relatively straightforward. It operates as a recursive procedure, initiating by recognizing all 2x2 sets of matching pieces through the combination of four individual pieces. Subsequently, it extends this process to identify all 4x4 sets of matching pieces by aggregating four 2x2 pieces, and this pattern continues recursively.

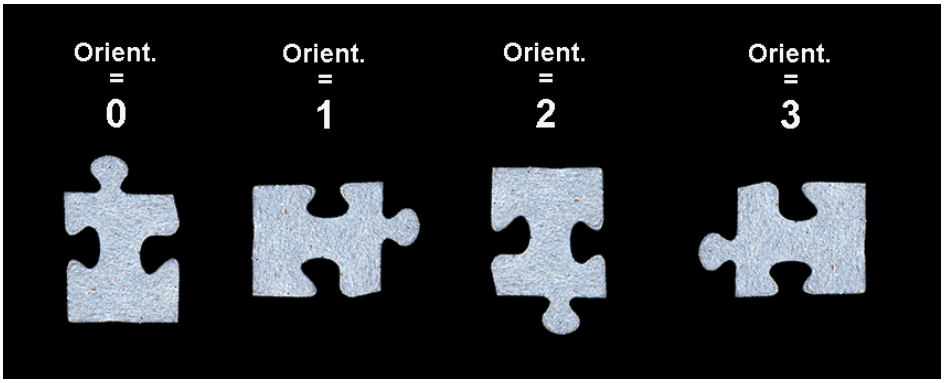
7.2 Basic Components

To implement the algorithm two fundamental object are defined

- SinglePiece: which contains information about a single puzzle piece.
- PieceGroup<T> where T can be either a PieceGroup or a SinglePiece
  This element encompasses four elements of type T, each with its own position (top right, top left, bottom left, bottom right).

Both SinglePiece and PieceGroup have an "orientation" attribute that goes from 0 to 3 F8.

Figure F8: orientation attribute of an object



7.2.1 The Levels

These objects are assigned levels:

- SinglePiece is at level 0, representing a 1x1 “solution” to a puzzle.
- PieceGroup<SinglePiece> is at level 1, representing a 2x2 solution to a puzzle.
- PieceGroup<PieceGroup<SinglePiece>> is at level 2, representing a 4x4 solution to a puzzle, and so forth.

7.2.2 The Behaviors

These basic objects also have the following shared behaviors:

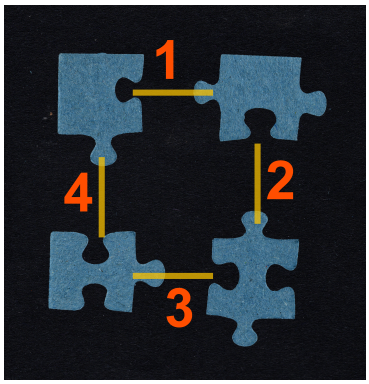
- 4 objects of level N can be combined to build an object of level N+1.
- One object of level N has 4 sides, each of which can be compared with a side of an element of level N. This results in a compatibility shore.
  - The comparing shore for level 0 objects is just the shore given by the Comparator 3.1.2.
  - The comparing shore for level N+1 objects is the average between two comparisons made with objects of level N

7.2.3 Combining four objects

When combining four objects into one, some comparisons are executed to ensure that the resulting piece is a valid combination.

- The right side of the object in the top left position is compared with the left side of the object in the top right position F9(1).
- The bottom side of the objection the top right position is compared with the top side of the object in the bottom right position F9(2).
- The left side of the objection the bottom right position is compared with the right side of the object in the bottom left position F9(3).
- The top side of the objection the bottom left position is compared with the bottom side of the object in the top left position F9(4).

Figure F9: Example of a combination of four objects in to one



In order for the created object to be considered valid all of the comparisons must have a shore greater than the constant MIN_SHORE_SINGLE_SIDE. And the average of the four comparisons must be greater than the constant MIN_SHORE_PIECE_GROUP.

The process also checks that all individual pieces inside a newly created object are different (because obviously is impossible to reuse the same piece twice to solve a puzzle)

7.3 The Algorithm

As mentioned earlier, the algorithm is implemented with a recursive function. The function is generic, and can take as input objects of any level.

A Function of level N takes as input a list of objects of level N, and creates all possible objects with level N+1. Then it uses the newly created objects to generate a recursive call on a version of itself with level N+1. This process is repeated until the desired size is reached.

```
# This function returns a list of object that match with 'piece_to_match'
# along the direction 'direction'.
# It also filters the data to return only pieces with an index greater than 'index_filter'
# This function is executed in constant time, since it relies on some pre-processing of the data
def get_matches(piece_to_match, direction, index_filter):
    ...

def solve<T>(pieces: List<T>):

    number_of_pieces = pieces.length()

    found_pieces = List::<PieceGroup<T>>::new()

    # this function compare all pieces with the others, and build some data structures
    # that are used from the function 'get_matches'
    # the time complexity is o(N^2)
    do_pre_processing(pieces)

    # test all possible combinations for the top left piece
    for top_left_index in 0..number_of_pieces:
        for top_left_orientation in 0..4:

        # get the top left piece
        top_left = get_piece(top_left_index, top_left_orientation)

        # test all possible combinations of top_right, bottom_right and bottom_left
        for top_right in get_matches(top_left, RIGHT, top_left_index):
            for bottom_right in get_matches(top_right, DOWN , top_left_index):
                for bottom_left in get_matches(bottom_right, LEFT, top_left_index):

                    # try to create the new piece group
                    new_element = PieceGroup::new(top_left, top_right, bottom_right,bottom_left)

                    if new_element.is_valid():
                        # if the 4 pieces can fit together the newly created piece
                        # is added to the list of found pieces
                        found_pieces.add_element(new_element)
                    else:
                        # otherwise the program check what the case of the
                        # invalid creation is, and generate a continue
                        # on the respective loop, to speed up the operations
                        continue_failure_cause_loop()

    # at the end a recursive call is generated
    return solve::<PieceGroup<T>>(found_pieces)
```

The procedure begins by testing all possible pieces and orientations as the top_right piece. Then continues by finding all the other 3 pieces in an efficient way thanks to get_matches. When the program has an hypothesis on the 4 pieces that can build a new piece it tries to create it.
The creation process does some checks, and if everything is successful it adds the piece to the list of found pieces. If the creation is not successful it figures out which piece has caused the failure, and moves to the

next iteration of the respective loop.
When all combinations have been found, the function makes a recursive call and the process is repeated.
The process ensures that the piece in the top_left is always the one with the lower index. This is done to avoid building the same piece 4 times with 4 different orientations.
One advantage of this algorithm is that the outer edge for loop can be easily parallelized to achieve better performances.

8 Testing Results

In this section, the solver proposed by this paper will be put to the test. This evaluation serves two key purposes:

- Gaining insight into the performance and limitations of the proposed algorithm.
- Conducting tests using both digital and real-world jigsaw puzzles to discern the distinctions between the two contexts.

8.1 Setup

The test sample consist in 5 different puzzles:

- a 4x4 real world puzzle
- a 8x8 real world puzzle
- a 4x4 digital puzzle
- a 8x8 digital puzzle
- a 16x16 digital puzzle

The real world puzzles have been obtained with a printer's scanner, and the digital puzzles have been obtained from https://puzzle.telegnom.org/.

The constants MIN_SHORE_SINGLE_SIDE and MIN_SHORE_PIECE_GROUP mentioned in section 7.2.3 were set at different values depending on the nature of the puzzle (real or digital).

The test has been performed on a ryzen 5 5600 CPU with 16 GB of RAM. The time measured in the test does not include the execution time of the Comparator 5.4, but only the one of the Solver 7.

8.2 Results

Size	Pieces	Execution time [s]	
		Digital puzzle	Real Puzzle
4x4	16	4	59
8x8	64	30	2851
16x16	256	377	N/A

8.3 Accuracy

It is crucial to mention that the project currently lacks an automated method for evaluating the accuracy of results. However, the accuracy of the two real-world puzzles has been manually confirmed, with both achieving a 100% correctness.

8.4 Analysis

This paper has asserted from the beginning that real-world puzzles pose greater challenges compared to their digital counterparts. While some intuitive explanations were provided to support this assertion, a proof hasn't been been presented until now. This test ultimately substantiates the claim, demonstrating that the performance of the same algorithm can vary significantly depending on the accuracy of the input data.

It is fair to point out that the performances of the algorithm are not amazing, but they are good enough to achieve the objectives of this paper 4. It is also worth remembering that the problem itself is extremely complex. Using the formula introduced in section 6.1.2 we can calculate that a 64 piece puzzle, like the one that has been solved before, has approximately $4 \cdot 10^{127}$ possible combinations.

$$C = N! \cdot 4^N = 64! \cdot 4^{64} \approx 4.32 \cdot 10^{127}$$

9 The Labeling tool

When the program runs, it also exports files that could be utilized in the future for training a machine-learning-based comparator. This section will list the available data and explain the format employed.

9.1 Basic notions

One piece is uniquely identified by his id piece_id. The id is an integer that goes from 0 to Number of pieces - 1. Each piece has 4 sides that can be identified by the side side_id, an integer that goes from 0 to 3.

9.2 Image of each piece

Inside the folder results/pieces the program will generate one image for each one of the pieces of the puzzle. The file will be named {piece_id}.jpeg. An example can be seen in F10

Figure F10

9.3 Image of each side

Inside the folder results/sides the program will generate one image for each side of each piece. The images will be named {piece_id}_{side_id}.jpeg. An example can be seen in F11

Figure F11

9.4 Corners of each piece

The file corners.json will be generated inside the folder results. This file will contain a list of pieces. Each piece has 4 sides, and each side has the coordinates of the two points that delimit itself.
An example of the generated json can be seen below.

```
[
  {
    "piece_id": 0,
    "side_0": {
      "p1": {"x":955, "y":1924},
      "p2": {"x":1024, "y":1016}
    },
    "side_1": {
      "p1": {"x":1024, "y":1016},
      "p2": {"x":1850, "y":1028}
    },
    "side_2": {
      "p1": {"x":1850, "y":1028},
      "p2": {"x":1814, "y":1992}
    },
    "side_3": {
      "p1": {"x":1814, "y":1992},
      "p2": {"x":955, "y":1924}
    }
  },
  {...},
  {...},
  ...
]
```

9.5 Match for each side

The file connections_result.json will be generated inside the folder results. This file will contain a list of pieces, each piece has 4 sides, and each side has a link to the side of the piece that matches it. The link can be null if the side is along the border.
An example of the generated json can be seen below.

```
[
  {
    "piece_id": 12,
    "side_0": {
        "piece": 5,
        "side": 0
    },
    "side_1": {
        "piece": 8,
        "side": 0
    },
    "side_2": {
        "piece": 0,
        "side": 0
    },
    "side_3": null
  },
  {...},
  {...},
  ...
]
```

10 Conclusions

In conclusion, this paper has successfully achieved its objectives. It has provided a comprehensive explanation of why real-world puzzles present greater challenges compared to their digital counterparts. Additionally, it has been shown that the same algorithm can have drastically worse performance when working with real-world puzzles compared to digital ones.

Furthermore, we have developed a highly effective labeling tool, which holds great promise for future applications in creating a machine learning-based comparator. Such a comparator has the potential to greatly facilitate puzzle-solving algorithms by providing more accurate data.

Even the solver we have created can be considered a success. While its time complexity may be less favorable when compared to state-of-the-art digital puzzle solvers, it stands as a strong contender in the realm of real-world puzzle-solving algorithms. In fact, it may even be considered one of the best available online for tackling real-world puzzles.

Figure F12: example of a result render of a digital puzzle

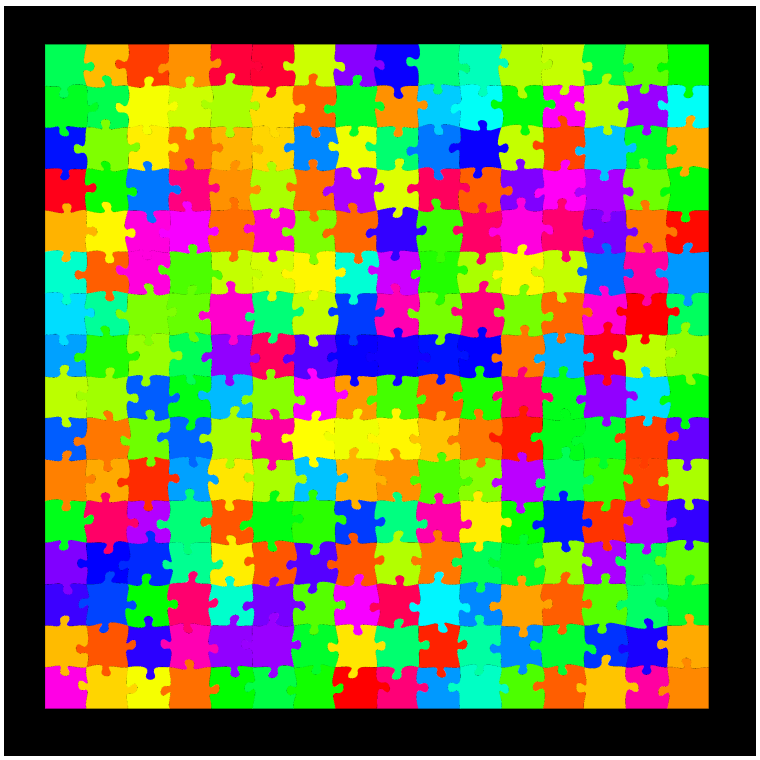
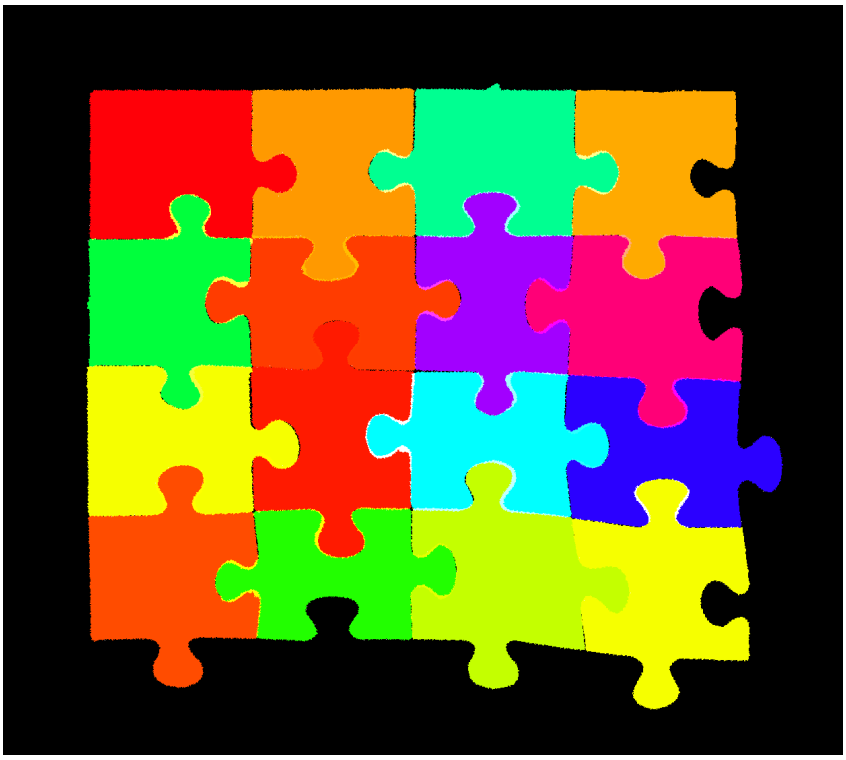


Figure F13: example of a result render of a real world puzzle



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.