2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI) | 979-8-3503-2110-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/SACI58269.2023.10158589

Solving Jigsaw Puzzles Using Computer Vision and Curve Similarity Measures

Olivér Balogh
John von Neumann Faculty of Informatics
Óbuda University
Budapest, Hungary
NS5CC9@stud.uni-obuda.hu

Zoltán Vámossy
John von Neumann Faculty of Informatics
Óbuda University
Budapest, Hungary
vamossy.zoltan@nik.uni-obuda.hu

***Abstract*—Jigsaw puzzles are a popular form of entertainment. Solving them with the help of computers raises several interesting problems, and has been the subject of many published papers in the past. In this paper, an approach to creating a program that uses pictures of real puzzle pieces to reconstruct the full puzzle image is presented. A photography technique is described that results in consistently recognizable images, which are used to extract features of the puzzle pieces. These features are then compared using two similarity algorithms - Hausdorff distance and Dynamic Time Warping. Three different puzzle assembly strategies that use these comparisons are presented, along with additional logic rules to solve the full puzzle. Using these the final program is capable of solving two different, 25-piece jigsaw puzzles. A comparison of the different similarity measures and assembly algorithms in the scope of the problem is also presented.**

***Keywords*— jigsaw puzzle, Hausdorff distance, dynamic time warping, computer vision**

I. Introduction

The origin of jigsaw puzzles can be traced back to the 18th century. Today there are many different types of puzzles, the most common being flat, 2-dimensional ones where every piece has four sides. A side can either be concave, convex, or straight, with the latter meaning that a particular edge constitutes the outside border of the puzzle. In the context of this paper, we are only examining puzzles that fit this description.

The goal is to create a program that, using pictures taken of real puzzle pieces, is able to automatically assemble the whole puzzle. To do this, the program needs several components. First, it needs to recognize the puzzle pieces in the input images and extract as much relevant information about them as possible. Then, using this data, the program should be able to compare two given puzzle pieces and decide how good of a fit they are. Using these comparisons and an assembly algorithm then arranges the pieces in the correct order. Finally, based on this the expected output of the program is a graphical assembly of the input puzzle pieces.

Computer-based, automatic puzzle solving has been the subject of many papers and theses. It is a problem that is easy to grasp, but as outlined above, the solution is not trivial and requires many different techniques working together to accomplish. Furthermore, it is a problem that can be made analogous to many other ones, from a variety of fields.

II. Related Work

The first paper [1] on this subject outlines many basic principles that are to be considered when tackling this problem. It describes two different kinds of algorithms - pictorial, one where the image of the puzzle piece is also considered when finding matches, and apictorial, which solely focuses on the geometric shape of the puzzle pieces. The paper shows an apictorial algorithm, which is based on the existing problem of pattern matching. It also goes on to discuss the issues of puzzle pieces being similar to each other, which might result in incorrect matches and makes efforts to make the algorithm less sensitive to noise.

While this provides a good starting point, it does not discuss the acquisition of puzzle pieces, opting instead to use near-perfect descriptors of the shape of edges. Not only does this prevent the utilization of color as a potential property to find matches by, but perfect geometric information also should not be assumed when working with photographs of puzzle pieces. Acquiring the precise shape of pieces requires specific photography techniques, such as scanning or illuminating the pieces from below [2].

Systems that use pictures of puzzle pieces as input [3] require additional steps at the start. First, the puzzle piece needs to be recognized from the picture. This can be seen as a type of foreground detection, with the foreground being the puzzle piece, and everything else the background. This detection needs to be as precise as possible since inaccuracies during the detection of the border impact the later stages of the algorithm. With the border found, the puzzle piece can be segmented from the image. The next step is separating the border into individual edges [4]. This can be done by exploiting the pieces' unique property of always having four separate sides, each adjoined by two corners. If the corners are found, splitting the border is trivial.

Finding the corners of the puzzle piece can be done in multiple ways. This includes existing corner detection algorithms, like Harris' [5] that work with the contour of the puzzle piece, or one that converts the outline into chain code and finds corners using that. With the edges separated, they can be classified and stored along with features that are later to be used during comparisons.

In principle, comparing two puzzle edges is done with a distance function, the inner workings of which can vary greatly. It can compare geometric information, such as the shape, length, or other features, or pictorial data, like the color or intensity changes along the edges. Combining two or more measurements with different weighing is also possible [6].

Likewise, there are many ways of assembling the whole puzzle. Real-life approaches, such as sorting the pieces by type and assembling the frame first can be used, as well as strategies that are infeasible for humans but can be achieved using computers. These can include calculating all possible matches in advance, brute force or backtrack-based assemblies, as well as more modern, loop-based solutions [7].

III. Methodology

Our approach to the problem is based on input images that require as little post-processing as possible. Taking photos of the puzzle pieces already distorts them to a degree, and since we intend to use the shape of the edges for comparisons, avoiding any further modification to the images is vital. This is achieved by illuminating the pieces from the bottom using a screen. With this method, a high contrast, white background can be used which makes for precise thresholding, while also eliminating the shadows cast by the pieces that would result in misshapen edges during detection. Combined with this solution the pictures are also taken from a fixed height, with the camera lens parallel to the puzzle piece. This way, the detected pieces can be treated relatively to each other, while not needing any corrections regarding their angling. Lastly, we need to consider the resolution of the taken pictures. Increasing the image quality past a certain degree not only results in diminishing returns, but slows down the speed of detection and provides more opportunities for image errors.

Thanks to the efforts made during image acquisition, the processing of inputs is relatively simple. After applying a greyscale filter, a low threshold value can be applied to remove the background. The resulting image can be used as a mask on the original, as well as for the input of a boundary tracing algorithm to find the puzzle piece's contour. As this array of border points is the basis of further calculations, it is converted into two additional formats for ease of use, polar coordinate and chain code representation. The polar coordinate form uses the middle of the puzzle piece, calculated by averaging the border points, as its reference point.

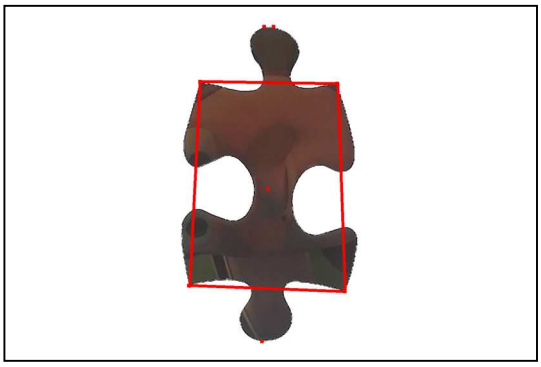

Fig. 1. Locating puzzle piece corners by rectangle approximation

The next step is identifying the four corners of the puzzle piece. We do this by first finding local maxima in the polar coordinate representation. Since this solution finds all corners as well as the tip of any tabs (convex puzzle side), it needs to be refined further. For this, we utilize the property of puzzle piece corners being approximately 90° from each other. This means that with some inaccuracy the four corners can be considered the corners of a rectangle, as seen in Fig. 1. Using this criterion we can reliably detect the corners.

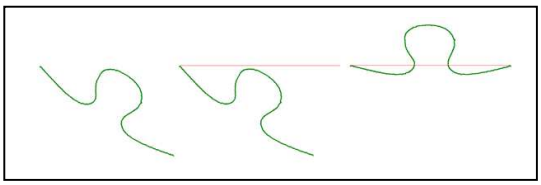

Fig. 2. Rotating edge segment with the help of a guide line

With the corners found, we can divide the contour into four sides. This is done by starting at a corner and adding subsequent points to the edge until the next corner is reached. The edges are rotated so that their starting and end points are at the same height, making all edges uniformly aligned. This process is shown on Fig. 2. The side is then grouped into one of three categories. A line is drawn between the first and last points of the edge, and the height difference of the edge points and the line is averaged. If this average is around zero, meaning the edge does not deviate much from the straight line, the side is considered straight. If the points tend to be above the line, it is considered a tab or convex side, otherwise, it is categorized as a blank or concave.

After the last step, we have access to all puzzle pieces and their important features. The assembly of the whole picture is done by two separate modules working in tandem. The simpler, more mathematical part conducts comparisons between puzzle pieces. It has no information about the whole picture, its only task is judging how good of a fit two edges make. The module responsible for the whole assembly is more complex, handling the assembly logic as well as assigning work to the comparer, using the results and a given strategy to assemble the whole puzzle.

Finding the pair of an edge involves taking a group of candidate edges, comparing each one to the original edge, and selecting the one with the best fitness. The number of candidates can be reduced by applying certain logical rules when selecting them. First, the pair of a convex edges can only ever be a concave one, and vice versa. A straight edge can never be the pair of any other edge, so they can be excluded from comparisons. Additionally, edges that already have a match can't be matched again, and an edge can never have a pair that is from the same puzzle piece as itself. These rules reduce the number of possible pairings greatly, but there are additional observations we can make, exploiting straight edges. Let us consider the neighbors of an edge and the two other edges that are directly next to it on a puzzle piece. For example, the left neighbor of the top edge is the left edge, and its right neighbor is the right edge. The following statements can be made:

- if an edge's left neighbor is straight, it can only be matched to an edge whose right neighbor is also straight
- if an edge's right neighbor is straight, it can only be matched to an edge whose left neighbor is also straight

These rules are based on the fact that jigsaw puzzles always have an unbroken, straight border. Similarly, the following can be stated regarding non-border edges:

- if an edge's left neighbor is not straight, it can only be matched to an edge whose right neighbor is also not straight

- if an edge's right neighbor is not straight, it can only be matched to an edge whose left neighbor is also not straight

Using these restrictions the number of comparisons can be reduced further, not only resulting in faster matching but potentially avoiding false positive pairs.

Our program is modular in design, meaning the type of comparison algorithm and assembly strategy can be swapped and combined in any manner. Regarding comparisons, we tested two similarity measures: Hausdorff distance and Dynamic Time Warping.

Hausdorff distance is a metric that can be used to compare two curves. It is ideal for puzzle edge comparisons, as the two shapes don't have to match in length. We use the following formula to calculate it, where X and Y are two sides being compared.

$$\max_{y \in Y} \min d(X, y)$$

The second comparison method tested is Dynamic Time Warping. While it's most commonly used for comparing time-based sequences, the warping aspect can be useful to compare edges that are similar in shape, but not in size.

The assembly module is responsible for positioning the puzzle pieces in the correct location and orientation. Since we do not provide any external information about the whole puzzle picture, the starting point for assembly is always a corner piece, which is placed and rotated as the top-left piece. It is worth noting that the corner piece is selected at random, so it is entirely possible that the initial puzzle piece is not the actual top-left piece but another corner. This does not affect assembly, and the only consequence is that the output image will be rotated.

The first implemented assembly strategy places pieces in row-major order. After the initial corner piece, a matching piece is placed to the right of it, which is repeated until the last right edge of the row is straight. The next row's starting piece is found and placed based on the bottom edge of the previous row's first piece. These steps are repeated until the whole puzzle is assembled.

The column-major order assembler works similarly, except for building the puzzle column-by-column.

The third assembly method is based on a popular approach when it comes to solving real-life jigsaw puzzles. The core idea is constructing the frame of the puzzle first, and then filling the inside. Since only pieces with a straight edge are part of the frame, those are the only pieces used for comparisons when constructing the border. Then, the remaining inside pieces are placed in row-major order.

Since these assemblies provide just a virtual arrangement of the pieces, the last module of the program uses masked pictures of puzzle pieces to graphically assemble the full image. These pieces are positioned on the picture regarding other pieces already placed, being rotated to fit better visually if necessary.

IV. Evaluation

Testing was conducted using two, 25-piece jigsaw puzzles. Each piece was placed and illuminated from the bottom using a tablet displaying a static white image, and the pictures were taken with a phone fixed above the capture area. The program was written in C#, using various features of the .NET and Accord.NET frameworks. All measurements are the result of multiple test runs, averaged. For testing, 1280 × 720 pixel resolution pictures were used which proved to be a good middle ground between processing speed and image clarity.

Generally, the most impactful step regarding to the final output is image acquisition and detection. If an input image is inadequate, the puzzle piece is not detected correctly, resulting in wrong descriptors. This makes it harder to find proper edge pairs, resulting in incorrect assemblies. Since any information lost during the input phase is irrecoverable later, it is important to gather as much and as precise data as possible. Detecting 25 input images took 44.7 seconds on average when done serialized, and 37.3 seconds when parallelized. Since detecting each piece is a separate task, it can be easily parallelized.

Fig. 3 and 4 show the final results of two datasets. The images presented are cropped and rotated for ease of viewing.

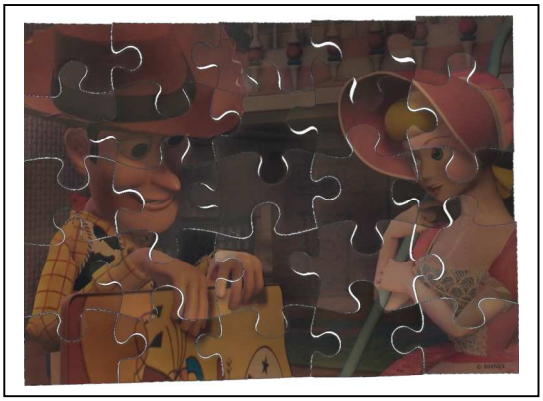

Fig. 3. Complete puzzle assembled by the program from the first dataset

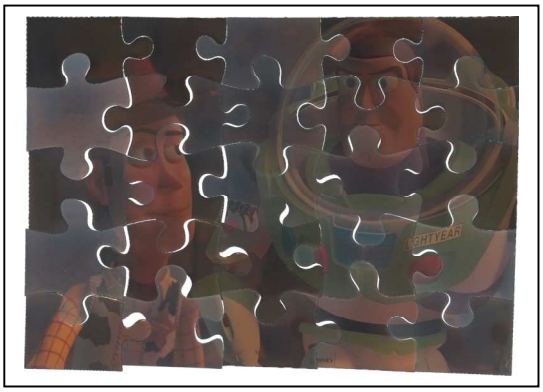

Fig. 4. Complete puzzle assembled by the program from the second dataset

Tables 1 and 2 show the respective runtimes of different assembly strategies combined with each comparison algorithm for each dataset.

TABLE 1 PUZZLE ASSEMBLY RUNTIMES FOR FIRST DATASET (MILLISECONDS)

	Hausdorff	Hausdorff, additional logic	Dynamic Time Warping	Dynamic Time Warping, additional logic
Row-major	6 610	2 800	47 619	20 001
Column-major	5 654	2 316	40 642	16 526
Frame	3 586	2 128	25 932	15 264

TABLE 2 PUZZLE ASSEMBLY RUNTIMES FOR SECOND DATASET (MILLISECONDS)

	Hausdorff	Hausdorff, additional logic	Dynamic Time Warping	Dynamic Time Warping, additional logic
Row-major	7 513	3 128	N/A[a]	22 747
Column-major	6 439	2 634	39 798	18 714
Frame	4 203	2 419	30 802	17 557

[a] Incorrectly determined match resulted in an error

These results show a few peculiarities. First, on average the second puzzle's assembly took more time than the first. While they are both 25-piece sets, edge segments from the second set on average have 743 points, while edges from the first puzzle have only 698. Since the comparisons go over each point, this difference adds up, resulting in the difference. Second, in both cases row-major assembly was slower than column-major assembly. This is for a similar reason – the puzzle pieces used for testing are elongated in shape, meaning they have two longer and two shorter sides. Since these strategies do comparisons on the same side repeatedly, one keeps comparing against longer sides, hence the difference in speed.

Based on our testing, Dynamic Time Warping is considerably slower than the Hausdorff comparison. This is to be expected – while Hausdorff distance is a relatively simple shape comparison algorithm using minimum and maximum searches, Dynamic Time Warping is more complicated, with the manipulation of curves before the comparison is very resource intensive. Among the different assembly strategies, constructing the frame first and then the inside proved to be the fastest. While filtering the pieces based on whether they have a straight edge or not introduces some overhead, it is negligible compared to the time saved by avoiding futile comparisons. The same can be said about introducing additional logic, as it works similarly – rejecting match candidates early based on their neighboring edges consists of four logic checks, which cuts down on the number of later comparisons.

As Table 2 shows, in one case our program failed to correctly assemble the puzzle. This happened due to an incorrectly matched piece after the initial corner piece was placed, resulting in an entirely wrong assembly. This issue was solved with the introduction of additional logic checks.

V. CONCLUSION

In this paper we presented our approach to building a computer-based jigsaw puzzle solver. The basis of this was a robust image acquisition method which results in input images that require minimal post-processing. We presented and tested Hausdorff distance and Dynamic Time Warping as possible puzzle edge comparison algorithms. Applying these along with different assembly strategies our program was able to solve two different, 25-piece puzzles.

Possible improvements include implementing more comparison algorithms and assembly strategies [8], as well as testing additional puzzle sets.

ACKNOWLEDGMENTS

The authors would like to thank both the GPGPU Programming Research Group of Óbuda University and the Hungarian National Talent Program (NTP-HHTDK-22) for their valuable support.

REFERENCES

[1] H. Freeman and L. Garder, "Apictorial Jigsaw Puzzles: The Computer Solution of a Problem in Pattern Recognition," in IEEE Transactions on Electronic Computers, vol. EC-13, no. 2, pp. 118-127, April 1964, DOI: 10.1109/PGEC.1964.263781.

[2] Á. Altsach, Jigsaw puzzle solver, Thesis, Óbuda University, John von Neumann Faculty of Informatics, 2013, p. 24

[3] D. A. Kosiba, P. M. Devaux, S. Balasubramanian, T. L. Gandhi and K. Kasturi, "An automatic jigsaw puzzle solver," Proceedings of 12th International Conference on Pattern Recognition, 1994, pp. 616-618 vol.1, DOI: 10.1109/ICPR.1994.576377.

[4] T. Ö. Onur, "Improved Image Denoising Using Wavelet Edge Detection Based on Otsu's Thresholding", in Acta Polytechnica Hungarica vol. 19. no. 2. 2022. pp. 79–92., DOI: 10.12700/APH.19.2.2022.2.5

[5] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in Proceedings of the 4th Alvey Vision Conference, 1988, pp. 147-151.

[6] Min Gyo Chung, M. M. Fleck and D. A. Forsyth, "Jigsaw puzzle solver using shape and color," ICSP '98. 1998 Fourth International Conference on Signal Processing (Cat. No.98TH8344), 1998, pp. 877-880 vol.2, DOI: 10.1109/ICOSP.1998.770751.

[7] K. Son, J. Hays and D. B. Cooper, "Solving Square Jigsaw Puzzle by Hierarchical Loop Constraints," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 41, no. 9, pp. 2222-2235, 1 Sept. 2019, DOI: 10.1109/TPAMI.2018.2857776.

[8] I. Lovas, "Fixed Point, Iteration-based, Adaptive Controller Tuning, Using a Genetic Algorithm", in Acta Polytechnica Hungarica vol. 19. no. 2. 2022. pp. 59–77., DOI: 10.12700/APH.19.2.2022.2.5