A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles: Final Report

Jordan Davidson

Abstract

For this project, I implemented the genetic algorithm-based jigsaw puzzle solver described in "A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles" by Sholomon *et al* [1]. This method involves producing a "child" solution from two "parent" solutions by detecting, extracting, and combining correctly assembled puzzle pieces. The solver developed by the paper's team is able to accurately solve puzzles of up to 22,834 pieces in a reasonable time. My solver is not able to quickly solve puzzles of such size, but is able to solve smaller puzzles with high accuracy.

1. Introduction

For this project, I aimed to reimplement the jigsaw puzzle solver described in the paper "A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles" by Dror Sholomon, Omid David, and Nathan S. Netanyahu [1]. The paper details a method mimicking natural selection on how to take a "jigsaw", or image cut into small pieces, and, through many iterations of the proposed algorithm, reconstruct the original image. First, a "population" of scrambled images is generated. Then, for some number of generations, individuals in the population are evaluated according to a fitness function, and a new population is populated with children produced from a crossover function applied to two parents in the current population, where more fit parents are more likely to be selected. The solution is then the fittest child in the most recent generation. Many jigsaw puzzle solvers have been implemented previously, but none of them have solved the problem using a genetic algorithm, and none of them have been able to handle puzzles of up to 22,834 pieces, as this one has.

As the authors explain, jigsaw puzzles are widely known to many people from childhood, in which the player has to reconstruct an original image from an assortment of its pieces. Solutions to this problem might benefit biology, chemistry, literature, speech descrambling, archeology, image editing, and the recovery of shredded documents or photographs, and, besides that, deserves research simply for stirring pure interest [1].

2.1 Review of Previous Work

The first attempt at solving this problem was made by Freeman and Garder [2] in 1964. Their solution matched together pieces of different shapes and could handle up to nine pieces. Since then, research focus regarding the problem has shifted

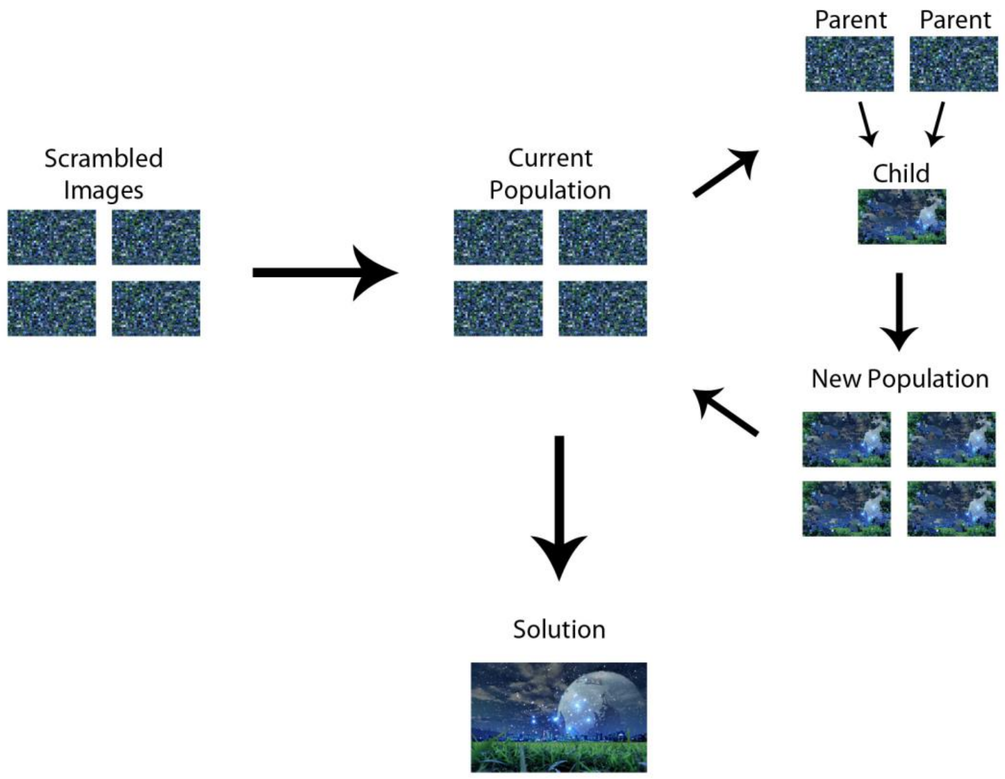


Figure 1: Overview of the genetic algorithm. First, a population of scrambled images is generated. Then, for 100 generations, a new population is formed from "child" images produced by two "parent" images in the current population. A solution is chosen from the most recent generation.

from shape-based to color-based solvers of square-tile puzzles. In 2010, Cho et al. [3] developed a probabilistic solver capable of handling up to 432 pieces. A year later, Yang et al. [4] improved upon these results with a particle filter-based puzzle solver, and in the same year, Pomeranz et al. [5] produced a solver capable of handling up to 3,000 piece puzzles.

2.2 Contributions of Method

The genetic algorithm-based jigsaw puzzle solver described in the paper by Sholomon *et al*[1] is the first time an effective genetic algorithm-based solver has been implemented. It should benefit research in the area of evolutionary computation by providing techniques that could be useful in solving additional problems with similar properties. Furthermore, the authors' implementation is able to handle up to 22,834-piece puzzles, more than any previous puzzle solver, with high accuracy, and can also solve smaller puzzles with higher accuracy than previous puzzle solvers. While my implementation is not able to handle puzzles with so many pieces in a reasonable time frame, it can handle up to 1440-piece puzzles with similar accuracy to the author's implementation. The performance provided by this solver could prove useful for solving more advanced

Algorithm 1

```
1. population ← generate 1000 random chromosomes
2. for generation_number = 1 → 100 do
3.         evaluate all chromosomes using the fitness function
4.         new_population ← NULL
5.         copy 4 best chromosomes to new_population
6.         while size(new_population) ≤ 1000 do
7.                 parent1 ← select chromosome
8.                 parent2 ← select chromosome
9.                 child ← crossover(parent1,parent2)
10.                add child to new_population
11.        end while
12.        population ← new_population
13. end for
```

Figure 2: Pseudocode of the framework for the genetic algorithm.

forms of the jigsaw puzzle problem, such as puzzles with missing pieces.

3.1 Technical Details: Summary

In order to solve the jigsaw problem, the algorithm considers a given selection of jigsaw puzzle pieces. First, an initial population of possible solutions, referred to as chromosomes, is randomly generated. Each chromosome represents a possible arrangement of the puzzle pieces into a complete image. Next, various operators inspired by natural selection, such as selection, reproduction, and mutation, are applied to the chromosomes.

A chromosome's reproduction rate is set proportional to its fitness, a score obtained by evaluating it according to a fitness function. The result of this function represents the quality of the chromosome as a solution. So, better solutions will produce more "offspring" through the reproduction operator, crossover. This operator attempts to take the best traits from both "parents" and use them to create a new chromosome that is a better solution than either of them.

Figure 2 represents the general framework for the algorithm [1]. 1000 random chromosomes are generated. Then, for 100 generations, every chromosome in the population is evaluated according to a fitness function, the four best chromosomes from the previous generations are retained, and the rest of the 1000 slots for the new generation are produced from the crossover of two parents. The probability that a chromosome is selected as a parent for a particular child is directly proportional to its fitness score.

3.2.1 Technical Details: Fitness Function

The fitness function works by measuring the sum of a dissimilarity measure between every neighboring piece in the chromosome. Piece edges are represented in L*a*b* color space by a $K \times K \times 3$ matrix, where K is the height and width of a piece in pixels.

Algorithm 2

1. If any available boundary meets the criterion of Phase 1 (both parents agree on a piece), place the piece there and goto (1); otherwise continue.
2. If any available boundary meets the criterion of Phase 2 (one parent contains a best-buddy piece), place the piece there and goto (1); otherwise continue.
3. Randomly choose a boundary, place the most compatible available piece there and goto (1).

Figure 2: Overview of the crossover function.

Considering two pieces, the dissimilarity between them in a specific spatial relation is equal to the square root of the sum of the squared L*a*b* value differences between adjacent pixels along their neighboring edges. The result for every pair of neighboring pieces is then summed and inversed into a cumulative fitness score. Higher scores indicate an individual with pieces that fit better together and, therefore, are more likely to resemble the original image.

Because the fitness of individuals, and therefore the dissimilarity between pieces in the individuals, is calculated many times, it is necessary to use a lookup table to minimize the run-time cost of the fitness function. The lookup table is of size 2 * $(N * M)^2$, where N is the number of pieces along the width of the puzzle, and M is the number of pieces along the height of the puzzle, containing all of the pairwise compatibilities for all pieces in both the right and up directions.

3.2.2 Technical Details: Crossover Function

The other consideration is the crossover function. Figure 3 outlines the function [1]. It starts with a single, randomly selected piece from the puzzle and continually adds new pieces onto itself until the child is formed. To accomplish this, the function repeats three phases until completion. First, it checks whether, for any available boundary, both parents have the same neighbor. If so, both pieces are probably in the correct position relative to each other, the piece is added to the collection of chosen pieces, called the kernel, and the first phase starts again. If not, then the function moves onto the next phase. In the second phase, the function checks whether, for any available boundary, a parent contains a "best-buddy" piece in the given spatial relation to the boundary. Two pieces are considered best-buddies if, for the given spatial relation, both pieces consider each other the best match (have lower dissimilarities together than with any other pieces). If a best-buddy is found, then the piece is added to the kernel, and the first phase begins again. If both the first and second phases fail to find a piece, then the third phase begins. In this phase, a boundary of the kernel is simply picked at random, the best available piece is attached to it, and phase 1 repeats. This process continues until the kernel has grown into a full-sized image.

4. Results

To test the accuracy of the solver, it was run on a set of 10 images divided into 360, 640, 1000, and 1444 pieces. The correctness of the generated solutions are evaluated by calculating the percentage of correct neighbors as well as by calculating a similarity score. This is the sum of the dissimilarities (as calculated earlier) between every pair of adjacent pieces in the original, unscrambled image divided by the same sum of dissimilarities in the generated solution. For both methods, we should expect higher scores to correspond to solutions that more closely resemble the original image.

The results of running the solver on each of the 10 images are shown in Tables 1 through 4, and the averages are shown in Table 5. As we can see, the solver assigned neighbors correctly 96.7081%, 95.6452%, 91.9620%, and 91.3605% of the time for 360-piece, 640-piece, 1000-piece, and 1440-piece puzzles, respectively. This is similar to the results of the authors' implementation, 95.70%, 95.38%, 95.85%, and 88.00% for 432-piece, 540-piece, 805-piece, and 2,360-piece puzzles, respectively. Unfortunately, the test images used by the authors are not available, but these results suggest that my and their implementations have similar performance.

The results show a similar trend for the similarity score. The generated solutions had an average similarity of 98.4724%, 95.0873%, 91.8486%, and 92.5804% for 360-piece, 640-piece, 1000-piece, and 1440-piece puzzles, respectively. Clearly, by both measures, the puzzle solver generates solutions strongly resembling the original image.

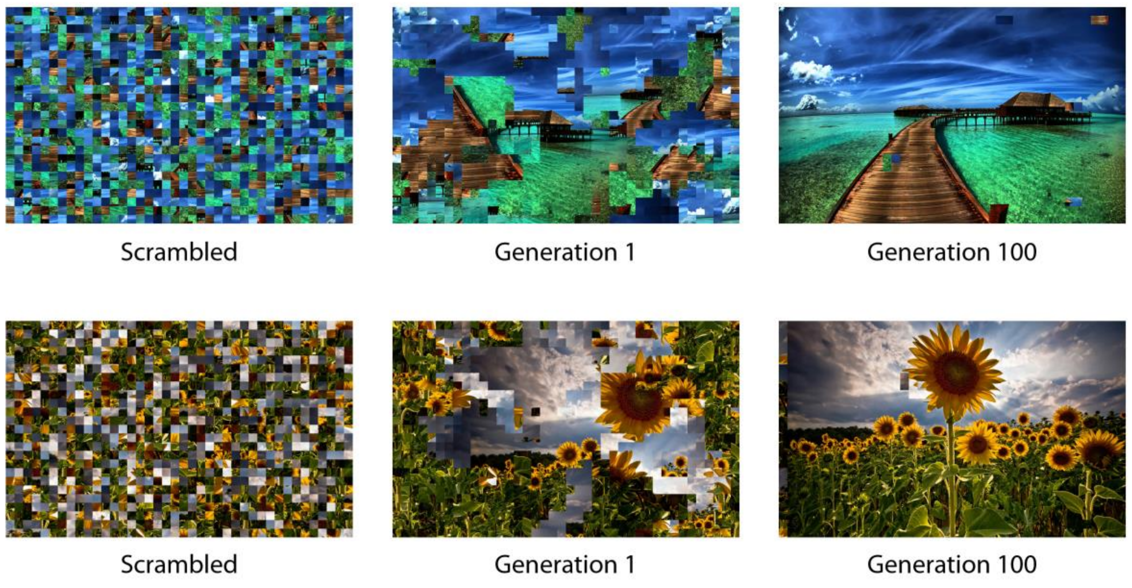


Figure 3: 2 test images after being scrambled, after 1 generation, and after 100 generations (the solution).

360 Piece Puzzle (Population Size: 1000, Generations: 100)

Image #	Percent Correct Neighbors	Percent Similarity	Run Time
1	88.9752%	97.0128%	63.7227 sec
2	100%	100%	63.5305 sec
3	99.2236%	99.9060%	63.7776 sec
4	89.5963%	95.1696%	62.2362 sec
5	97.5155%	92.7625%	62.0799 sec
6	95.9627%	99.9574%	62.6855 sec
7	100%	100%	63.1906 sec
8	100%	100%	63.6201 sec
9	98.9130%	99.9316%	62.3983 sec
10	96.8944%	99.9838%	61.6523 sec

Table 1: Results on 10 images divided into 360 pieces.

640 Piece Puzzle (Population Size: 1000, Generations: 100)

Image #	Percent Correct Neighbors	Percent Similarity	Run Time
1	98.7267%	99.6577%	167.229 sec
2	100%	100%	176.428 sec
3	100%	100%	170.588 sec
4	89.1341%	90.9676%	163.201 sec
5	70.5433%	60.4417%	151.228 sec
6	100%	100%	166.413 sec
7	100%	100%	170.293 sec
8	100%	100%	170.899 sec
9	100%	100%	162.330 sec
10	98.0475%	99.806%	163.773 sec

Table 2: Results on 10 images divided into 640 pieces.

1000 Piece Puzzle (Population Size: 1000, Generations: 100)

Image #	Percent Correct Neighbors	Percent Similarity	Run Time
1	81.7415%	84.5562%	352.868 sec
2	99.2521%	99.3802%	362.408 sec
3	83.7393%	85.5277%	357.783 sec
4	82.4444%	86.8303%	334.689 sec
5	96.4295%	88.2005%	344.799 sec
6	91.5470%	89.2697%	335.724 sec
7	99.1987%	99.3101%	355.690 sec
8	100%	100%	361.345 sec
9	98.9701%	98.1953%	342.709 sec
10	86.2970%	87.2155%	343.459 sec

Table 3: Results on 10 images divided into 1000 pieces.

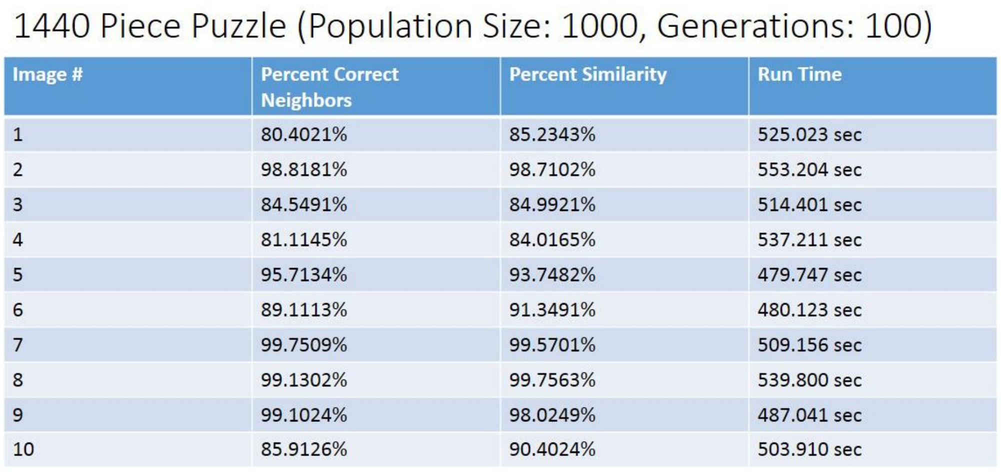
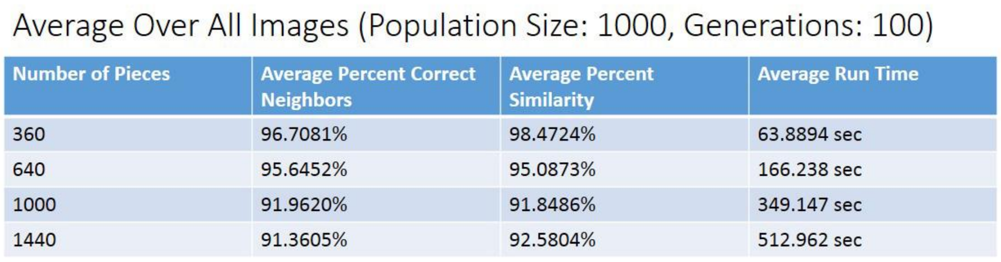

1440 Piece Puzzle (Population Size: 1000, Generations: 100)

Image #	Percent Correct Neighbors	Percent Similarity	Run Time
1	80.4021%	85.2343%	525.023 sec
2	98.8181%	98.7102%	553.204 sec
3	84.5491%	84.9921%	514.401 sec
4	81.1145%	84.0165%	537.211 sec
5	95.7134%	93.7482%	479.747 sec
6	89.1113%	91.3491%	480.123 sec
7	99.7509%	99.5701%	509.156 sec
8	99.1302%	99.7563%	539.800 sec
9	99.1024%	98.0249%	487.041 sec
10	85.9126%	90.4024%	503.910 sec

Table 4: Results on 10 images divided into 1440 pieces.


Average Over All Images (Population Size: 1000, Generations: 100)

Number of Pieces	Average Percent Correct Neighbors	Average Percent Similarity	Average Run Time
360	96.7081%	98.4724%	63.8894 sec
640	95.6452%	95.0873%	166.238 sec
1000	91.9620%	91.8486%	349.147 sec
1440	91.3605%	92.5804%	512.962 sec

Table 5: Results averaged over all 10 images for each size puzzle.

5. Conclusions

For this project, I successfully implemented a jigsaw puzzle solver based on the genetic-algorithm described in the paper and implemented by Sholomon *et al* [1], the first effective genetic algorithm-based solver. My solver is not able to solve puzzles with as many pieces as the authors' solver in a reasonable amount of time, but it is able to solve smaller puzzles with similar accuracy. This solver could be useful in solving more difficult variations of the jigsaw puzzle problem and is, itself, a solid contribution to the set of jigsaw puzzle solvers.

6. References

[1] D. Sholomon, O. David, N. Netanyahu. A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles. In *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1767-1774.

[2] H. Freeman and L. Garder. Apictorial Jigsaw Puzzles: The Computer Solution of a Problem in Pattern Recognition. *IEEE Transactions on Electronic Computers*, EC-13(2):118–127, 1964.

[3] T. Cho, S. Avidan, and W. Freeman. A Probabilistic Image Jigsaw Puzzle Solver. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 183-190, 2010.

[4] X. Yang, N. Adluru, and L. J. Latecki. Particle Filter with State Permutations for Solving Image Jigsaw Puzzles. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2873–2880. IEEE, 2011.

[5] D. Pomeranz, M. Shemesh, and O. Ben-Shahar. A Fully Automated Greedy Square Jigsaw Puzzle Solver. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 9-16, 2011.