Grey-scale Image Reconstruction

Evolutionary Algorithms Part 2. Final work.

Juan Carlos Recio Abad

Máster universitario de ingeniería del software e inteligencia artificial.

Universidad de Málaga

Abstract

The article describes how to build a image reconstruction *evolutionary algorithm* to obtain the original image from an unordered set of rows of a given image in grey-scale colors. A process using different fitness functions, parameters and operators is followed in order to find out the right combination to provide the best possible solution to the problem. The implementation is based on the java project included on the subject: *Evolutionary Algorithms* available in GitHub.

Keywords: *evolutionary algorithm; mutation; permutation; euclidean distance; java*

---

1. Introduction

The problem consists of reconstructing an image to its real form after have been altered by an external action that forces it to be in disorder.

2. Representation of the problem. Permutations.

An image in grey-scale color is represented by a 2D square matrix of integers within the range [0,255]. For the given test samples, these matrices have dimension 512x512. That is 262144 pixels for every possible image (combination of pixels). A straightforward representation using the matrix itself would require a lot of memory and a lot of computation as it would imply going through every pixel to analyse and make comparisons to find a solution. Doing that is totally unmanageable in a reasonable time. The definition of the problem states that only 1 dimension in the matrix is unordered while the other is in the right position. That allows to track only 1 dimension to represent solutions of the problem. For this, the best choice is to use *permutations* to represent combinations of rows on an image. Each item from the permutation stands for the order of each row in the image.

3. Fitness function

Given a proper representation, obtaining the fitness of any possible solution is much less time and memory consuming. In order to get the fitness and check how good a solution is we need to check how any combination of rows is close or not from the original image. Intuitively, for any given image, 2 adjacent rows of pixels would be very similar. The colors cannot (should not) change abruptly in every pixel position for a natural image, just the opposite, they change in more or less smooth transitions along the image. For this reason, calculating the quality of a solution should

be as simple as minimizing the average distance of every couple of adjacent rows in the image. To calculate the distance between 2 rows, the euclidean distance can be used as if they were normal N-dimensional vectors.

$$d(x_a, x_b) = \sqrt{x_a^2 - x_b^2}$$

So the fitness function for a permutation x can be defined as the sum of all the adjacent pair distances:

$$f(x) = \frac{1}{n} \sum_{i=1}^{n-1} d(x_i, x_{i+1}) = \frac{1}{n} \left(d(x_1, x_2) + \cdots + d(x_{n-1}, x_n) \right)$$

For simplicity and to avoid extra calculations, *division per n* is skipped as it is not relevant for the result as all values would be divided by same number (this will imply working with higher values).

$$f'(x) = \sum_{i=1}^{n-1} d(x_i, x_{i+1}) = d(x_1, x_2) + \cdots + d(x_{n-1}, x_n)$$

Good fitness will be lower values whereas bad fitness will be higher values. This is then a minimization problem, trying to obtain the lowest possible fitness for an image.

4. Population generation

The initial population is basically a method that generates n random permutations of 512 integers.

An alternative approach might be to also generate the same n random permutations plus the original disordered permutation which is implicit from the matrix that represents the image. This puts additional information into the problem as we can compute the fitness value of the current disordered image so we never go worse than the provided image. However, the rest of the population will not catch up with the quality of the given individual so the recombination would might make worse offspring all the time and the algorithm will get trapped in a local optima given by the original individual.

5. Operators

Several permutation operators have been combined differently to get and compare solutions, as well as to study how each converges to certain fitness values.

5.1. Recombination operators for permutations

5.1.1. Order 1

To apply this recombination of parents, there will be 2 positions selected randomly. From one parent the subset between the 2 edges will be copied into the offspring. Rest of the items will be copied in the remaining offspring positions in the same order that appear in the second parent without repeating what was copied from parent 1.

5.1.2. Edge Crossover

This operator is based on the concept of neighborhood. The idea is to prioritize the edges common to the parents when generating the offspring. It looks to be quite appropriate for this type of problem. The point is that rows should somehow always be interconnected to others based on the color similarities minimizing the *"paths between rows"*, this operator is particularly useful when adjacency is looked for, like in the case of the images.

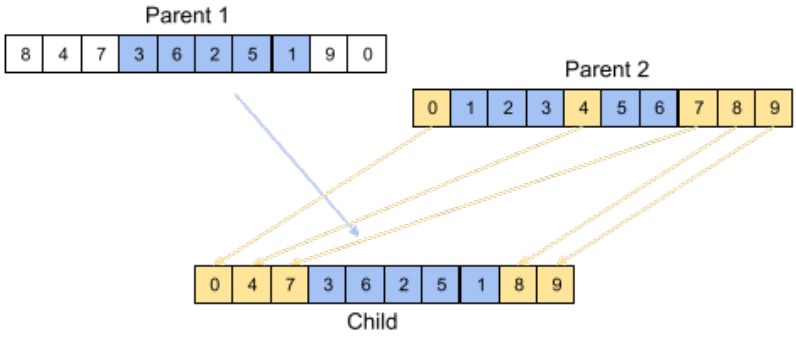

Figure 1. Recombination operator: Order 1

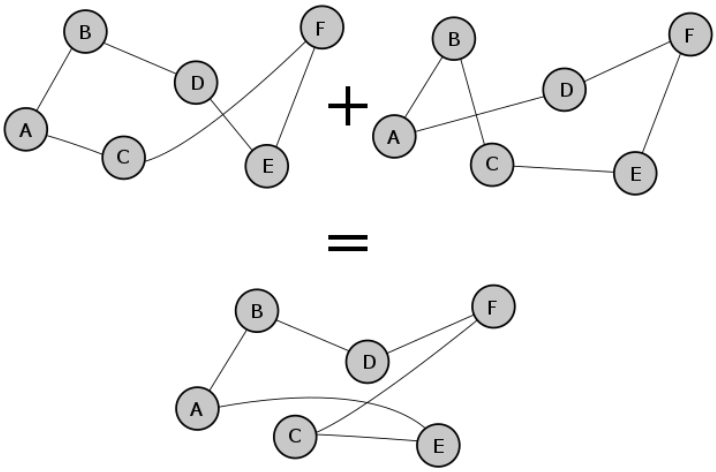

Figure 2. Edge Recombination operator. Source: Wikipedia.

5.2. Mutation operators for permutations

5.2.1. Simple swap mutation operator

This operator chooses randomly 2 rows to swap its order. The resulting solution with the swapped values represents the image offspring.

5.2.2. Swap mutation operator

This operator is a more sophisticated variation of the simple swap operator. It randomly chooses 2 rows, and all the intermediate rows are randomly swapped in pairs to produce the new image offspring.

5.2.3. Scramble mutation operator

It randomly chooses 2 rows from an individual and scrambles the intermediate values (slides one position to the left and it leaves the first element in the last position) to produce a new image offspring.

5.2.4. Inversion mutation operator

It chooses randomly 2 rows that act as edges of a subset of an individual and inverse the order of the intermediate subset to produce a new image offspring.

5.2.5. Random mutation operator

It chooses randomly 2 rows that act as edges of a subset of an individual and swap randomly all the items inside.

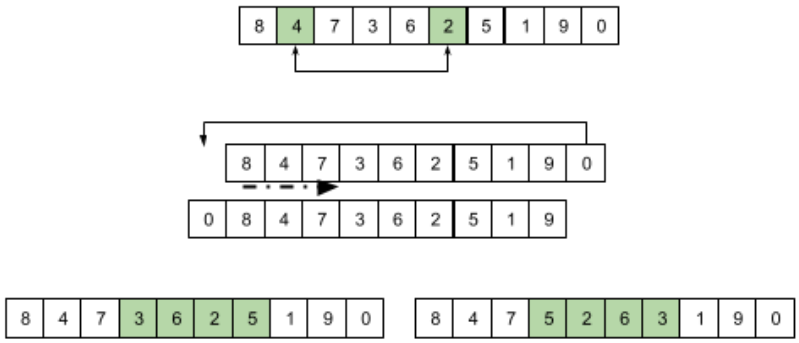

Figure 3. Swap, scramble and inversion mutation operators.

6. Algorithm Implementation

The algorithm has been implemented using Java. It is based on the code provided in the subject *"Algoritmos evolutivos"* from *"Máster Universitario de ingeniería del software e inteligencia artificial"*.
The code has been modified to allow parameterize all the features required to run the evolutionary algorithm.
The code is based on composition of different parameters and operators. The basic algorithm skeleton allows to pass in different recombination and mutation operators. In the same way, mutation probability, population size, number of function fitness evaluations and a seed for randomness.
The operators described in the above section have all been programmed in Java.

7. Algorithm execution and results

In order to test the algorithm, 4 different integer grey-scale images have been provided. These represent images whose rows have been altered so they are disordered. The disorder of each image is different.
The fitness function can be run against the images in order to test what is the current value of the disordered images.

Image	Fitness
Image 1	598680.298
Image 2	191550.169
Image 3	138463.703
Image 4	114329.390

Table 1. Fitness for the provided disordered images.

From the results obtained, we can conclude that the disorder of the images is descending. Image 1 is the most disordered image and 4 is the least disordered image as it can be seen in 4 and 5.

Some initial naïve experiment can be started to observe a bit the nature of the problem and how the algorithm behaves.
For a first experiment:

- *Order 1* Recombination operator.
- Scramble mutation operator with 0.1 of mutation probability.

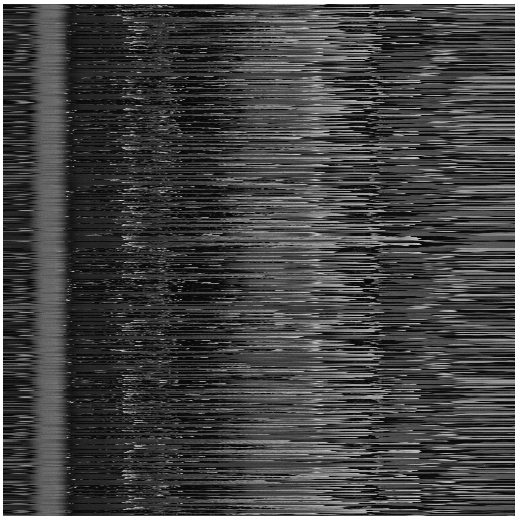

Figure 4. Image most disordered. Fitness = 598680.298

Figure 5. Image least disordered. Fitness = 114329.390

- 2000 Fitness function evaluations.
- Population size of 50 individuals.
- Image 1 as an input.
- Average of 20 executions.

This produces a poor result, only slightly above the fitness value of Image 1 which is the worst. The value obtained is 590864.799 so this approach barely gets any improvement on the order. Same experiment is repeated with several mutation operators and different mutation probabilities. Because probability of 0.1 was not escaping from a poor value, attempts with higher probabilities are made. Then the combinations tested are:

- *Order 1* Recombination operator.
- Scramble, swap, simple swap and inversion operators with probabilities between [0.1, 0.8] (testing from increments of 0.1)
- Fitness function evaluations from 2000 to 10000.

- Population size of 5 to 50 individuals.
- Image 1 as an input.
- Average of 20 executions.

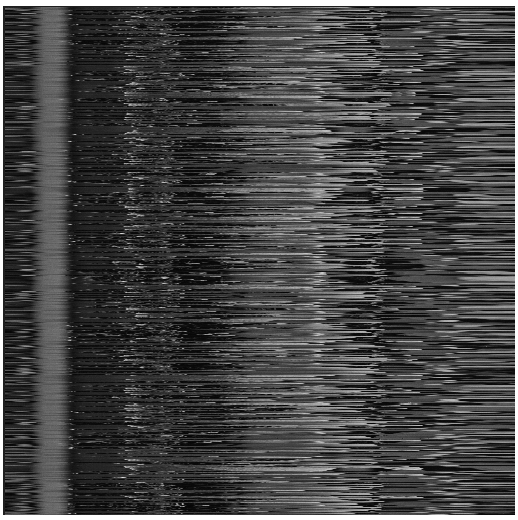

Figure 6. Image produced by Order 1 recombination operator.

The values obtained are still too bad, as their fitness range from 58000 to 60000. The explanation must be on the recombination operator used. Order 1 is not providing a good way to pass features in the offspring permutations, it cannot escape from the bad local optima it starts with.
Figure 3 shows that Order 1 recombination does not produce any reconstruction or visible pattern in the image, which was also indicative from the high fitness value.

Next recombination operator to use is Edge Crossover, as per the theory this operator is expected to produce better results on permutations and adjacency problems. The downside is its performance which is fairly slower than Order 1 as it needs to compute more things.
Rest of parameters are set the same as the previous execution for Order 1.

- *Edge Crossover* Recombination operator.
- Scramble mutation operator with 0.1 of mutation probability.
- 2000 Fitness function evaluations.
- Population size of 50 individuals.
- Image 1 as an input.
- Average of 20 executions.

The results are way better just by changing the recombination operator, obtaining an average fitness of 457232.549. Even if this still is a high value to get a right visible image, it is an improvement of 23% only by using a new operator. In Figure 4 it can be inferred that similar rows are now closer and somehow depicting certain patterns. However, the fitness still gets trapped in a local optima with a high fitness value. Figure 8 shows how this algorithm converges to a local optima after 50000 function evaluations.
The reason swap mutation has been used it was a way to escape from local optima (some isolated parts of the image already together might get stuck without improving to better solutions).

In order to avoid this, one option would be to use a mutation that allows to "move blocks" without impacting the adjacency already obtained. An attempt for this can be a variation of scramble,

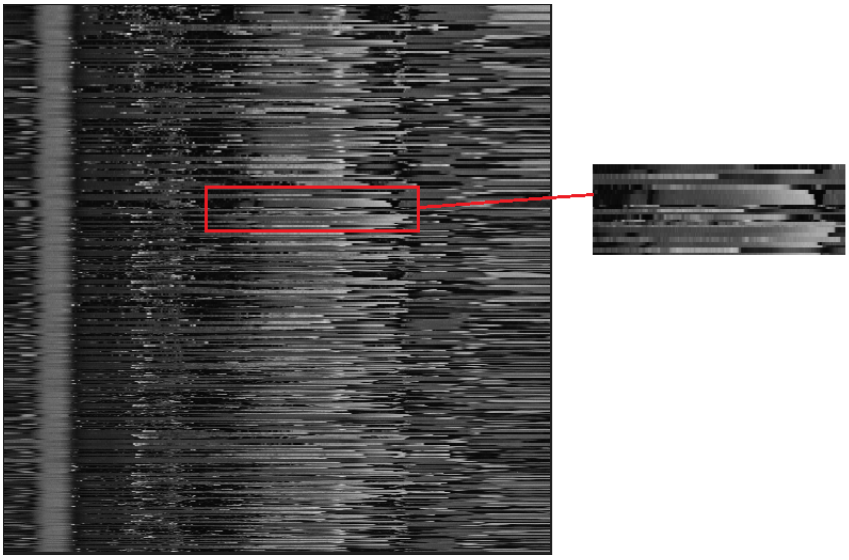

Figure 7. Image produced by Edge recombination operator with Fitness = 455231.26

that allows to slide more than one position.

Using this variation (from now on Scramble *) is observed that the algorithm converges extremely much faster (Figure 9) to an slightly better solution, but the fitness is still around 430000 440000. Another alternative used is to use variable mutation probability based on the number of evaluations, decreasing the value as long as the number of evaluations grow. The results are the same and the convergence is similar.

7.1. Results

Img	Evaluations	Mutation op	Mutation rate	Crossover	Population	Fitness
Image 1	10.000	Scramble	0.015	Order1	25	592745
Image 1	10.000	Scramble *	0.4	Order1	40	592929
Image 4	10.000	Swap	0.2	Order1	100	587337
Image 2	20.000	Inversion	0.02	Edge Crossover	15	444623
Image 2	8.000	Simple Swap	0.1	Edge Crossover	12	444254
Image 3	15.000	Scramble	0.35	Edge Crossover	200	452999
Image 4	25.000	Scramble *	0.15	Edge Crossover	500	458519
Image 1	50.000	Scramble *	0.09	Edge Crossover	250	431289

Table 2. Results for different executions.

8. Code usage

Next commands show how to use the algorithm:

run <file> <population size> <number of evaluations> <mutation probability> <mutation operator> <recombination operator>[includeInitialChromosome] [seed]

- file: Path (without spaces) to the file that contains the integer grey-scale matrix.
- population size: Number of individuals of the population
- number of evaluations: Maximum number of evaluations that the algorithm will execute.

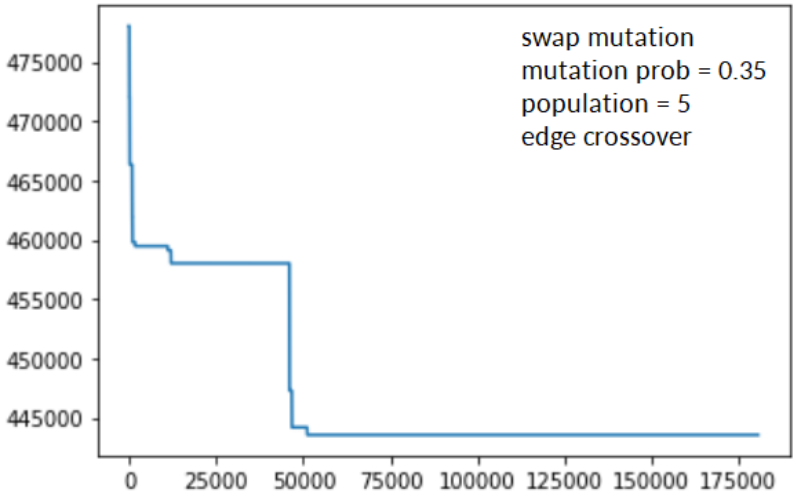

Figure 8. Convergence at 50.000 function evaluations

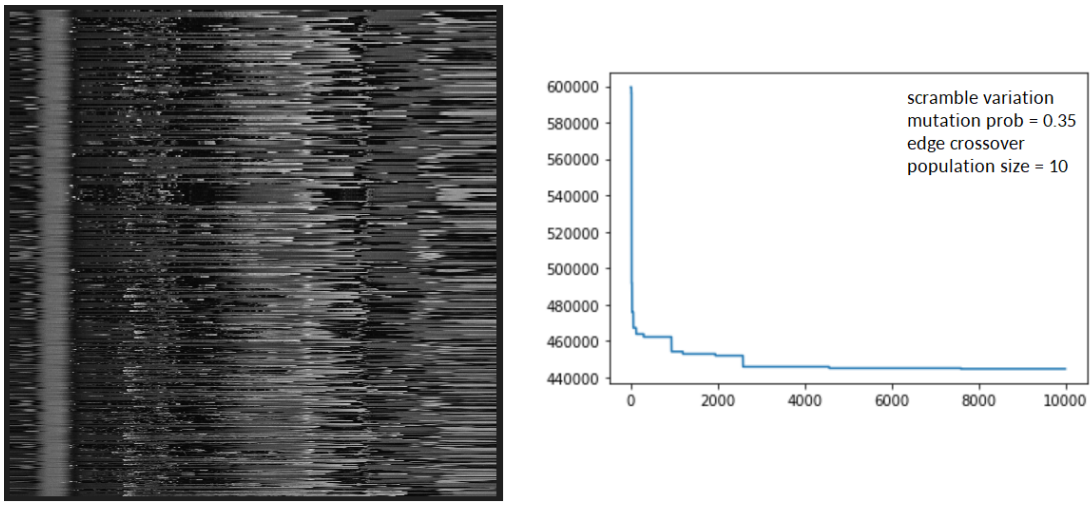

Figure 9. Faster convergence at 2.500 function evaluations

- mutation probability: Mutation probability
- mutation operator: Mutation operator to use: candidates: 'inverse', 'swap', 'simpleSwap', 'scramble' and 'scramble2' (variation of scramble)
- recombination operator: Recombination operator to use: candidates: 'edge' and 'ox1'
- includeInitialChromosome: (Optional) Whether the initial image order (disordered) is to be included
- seed: (Optional) Number to be used as a randomness seed

Example: *run img4.txt 3 100000 0.25 random edge*
display <file>

It will display the integer grey-scale matrix as an image.

After running the algorithm, a result image will be stored in the program folder.
The program contains an internal boolean flag on the Main class $ENABLE_LOGS$ that allows to store the results of an execution in CSV format that can be exported to plot with matplotlib or other plotting library. That is how the graphs were generated for this document.

9. Conclusion

The results are not satisfactory as the images were not reconstructed and the image has not been recognised. I might have overlooked some key factor in the algorithm that makes it converge too early in non good solutions, and it cannot escape from local optima always around a fitness value between [430000, 450000]. One assumption is that I did not find the right mutation operator plus the right parameters. (Edge crossover recombination has been a good improvement to get better solutions).

Another assumption is that I made some mistake in the code of the operators.
As I coded all the recombination and mutation operators reading the literature and I might have introduced some non-trivial bugs that make the algorithm not behave properly at certain moments.
For future works, given these results, it might be more reasonable to use already built-in libraries with all the operators instead of coding all of them from scratch.
In any case, the best combination found has been to use the Edge Crossover recombination operator, the Variation Scramble mutation operator and a mutation probability of 0.35. It is worth to mention that also using the random mutation operator gets similar results.

References

[1] Eiben, A. E., Smith, J. E. (2003). Introduction to Evolutionary Computing. Springer. ISBN: 978-3-662-05094-1

[2] Rubicite (2008), Genetic Algorithms, https://www.rubicite.com/Tutorials/GeneticAlgorithms.aspx

[3] Froese, Tom and Emmet Spier. "Convergence and crossover Froese and Spier 2 Convergence and crossover : The permutation problem revisited." (2008).

[4] Desport, Pierre & Basseur, Matthieu & Lardeux, Frédéric & Saubion, Frédéric & Goëffon, Adrien. (2015). Empirical Analysis of Operators for Permutation Based Problems.

[5] hieng, Hung Wahid, Noorhaniza. (2014). A Performance Comparison of Genetic Algorithm's Mutation Operators in n-Cities Open Loop Travelling Salesman Problem.