Optimization

Carlos Mougan

Genetic Monalisa

Submitted To:
Lluis Alseda
Optimization Professor
Mathematics Department

Submitted By :
Carlos Mougan
MSc in Modelling
CRM

Abstract

The goal of this project is drawing the Monalisa figure with the 50 best polynomials from different orders & searching the solution space with a genetic algorithm. The solution space takes into account the number of vertex, position and colour of each polygon. We have decided after some coding not to use a crossover for optimal performance. Different mutation techniques have been used to see the optimal learning rate of the algorithm. Any other image can be attempt to be paint with this algorithm.

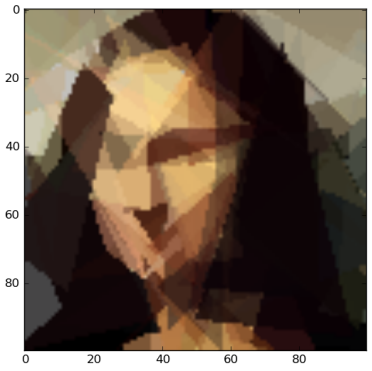


Figure 1: Final GA version

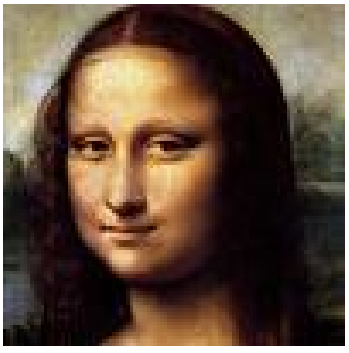


Figure 2: Real image

Contents

1 Genetic Algorithms

Genetic Algorithms (GAs) are adaptive methods which may be used to solve search and optimization problems. They are based on the genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and survival of the fittest", stated by Charles Darwin in The Origin of Species . By mimicking this process, genetic algorithms are able to *evolve* solution to real world problems.

GAs use a direct analogy of natural behaviour. They work with a population of **individuals**, each representing a possible solution to a given problem. Each individual is assigned a **fitness score** according to how good a solution to the problem is. The highly fitted individuals are given opportunities to reproduce, by **crossover** and **mutation** with other individuals in the population. A genetic algorithm is

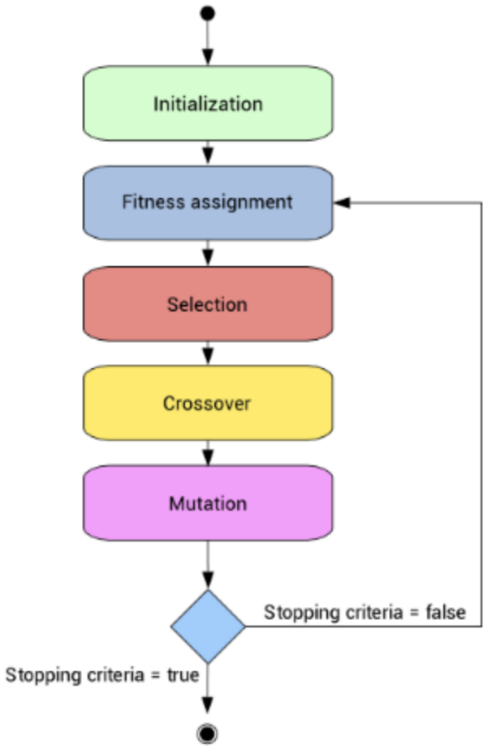


Figure 3: Graphical representation of genetic algorithms

expected to make evolve the population of individuals over generations, in such a way that the average fitness score tends to *the optimal fitness score*, so that the individuals tend to the optimal solution of the problem.

Fittest individuals are given chance to reproduce, passing useful genes to their offspring. Thus the gene pool constantly improves, i.e. evolution is converging to better and better solutions. There are very candidate solutions evaluated all the time. This is the behaviour that we try to roughly mimic with genetic algorithms.

Once we have understood what is the idea of GA, we can go deeper in the algorithm. Normally a GA works as it follows, [Uddalok Sarkar, 2010]:

- **1-** Start with a randomly generated population of n l−bit chromosomes (candidate solutions to a problem).
- **2-** Calculate the fitness function of each chromosome x in the population.
- **3-** Repeat the following steps until n offspring have been created:
  - **a)** Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.
  - **b)** With probability pc (the "crossover probability" or "crossover rate"), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, from two offspring that are exact copies of their respective parents. (Note that here the crossover rate is defined to be the probability that two parents will cross over in a single point. There are also "multi−point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place.)
  - **c)** Mutate the two offspring at each locus with probability pm (the mutation probability or mutation rate), and place the resulting chromosomes in the new population. If n is odd, one new population member can be discarded at random.
- **4-** Replace the current population with the new population.
- **5-** Go to step 2.

2 Code

The code is one python script ("monalisa.py") that can be run from the terminal just by calling

```
python monalisa.py
```

The image file has to be in the same folder than the python script. Now lets talk about what is inside of the file.

2.1 Global Variables

This variables are made public:

```
OFFSET = 10 #
POLYGONS = 50
POLY_MIN_POINTS = 3
POLY_MAX_POINTS = 5
color_mutation=90
shape_mutation=90
POPULATION_SIZE=2
Generations=100
```

The OFFSET is defined so the vertices don't fall inside the figure. POLYGONS is the number of geometric figures in each frame. POLI_POINTS refer to the minimum number of vertex that each polygon has. POPULATION_SIZE is the number of frames (individuals) of each population. And generations is the maximum number of generations that the algorithm will perform.

2.2 Population

After defining some functions that initiate a color, a polygon a vertex, let's define how is the individual going to be.

```
def generate_initial_individual():
    (width, height) = imagen.size
    aux_population_0,aux_population_1,aux_population_2=[],[],[]
    for i in range(POLYGONS):
        polygon_type = random.randrange(POLY_MIN_POINTS, POLY_MAX_POINTS + 1)
        points = []
        for j in range(polygon_type):
```

```
8                point = generate_point(width, height)
9                points.append(point)
10           colour =genereate_color()
11           aux_population_1.append(colour)
12           aux_population_2.append(points)
13       img = drawing( aux_population_1, aux_population_2, COLOUR_BLACK)
14       aux_population_0.append(img)
15       return aux_population_0, aux_population_1,aux_population_2
16  generate_initial_individual()
17  OUT:
18  ([<PIL.Image.Image image mode=RGB size=100x100 at 0x1A16A10D68>],
19   [(0, 0, 0, 255), (0, 0, 0, 255)],
20   [[(13, 43), (28, 0), (96, 71), (−10, 44), (109, 16)],
21    [(28, 107), (44, 88), (51, 5), (1, 83), (23, 10)]])
```

Listing 1: Initiate Individual

The population is just another list containing individuals of the length indicated in the initial global variables. The fitness function of the individual is made by comparing pixel by pixel the original and the individual. It's made measuring the distances between the colors in the RGB three dimensional space.

```
1   def fitness(img_1, img_2):
2       fitness = 0.0
3       for y in range(0, img_size[1]):
4           for x in range(0, img_size[0]):
5               r1, g1, b1 = img_1.getpixel((x, y))
6               r2, g2, b2 = img_2.getpixel((x, y))
7               d_r = r1 − r2
8               d_b = b1 − b2
9               d_g = g1 − g2
10              pixel_fitness = math.sqrt(d_r ∗ d_r + d_g ∗ d_g + d_b ∗ d_b )
11              fitness += pixel_fitness
12      return fitness
13  OUT: 245.3444
```

Listing 2: Fitness Function

2.3 Selection & Crossover

After careful thinking and implementing some typical crossover we decided not to do any crossover since it was not useful at all. The idea of crossover doesn't really suit

this problem since the polygons are transparent and just changing one will change the image entirely. Also the order has to be taken into consideration.

One way to check if making crossover had any sense was having 3 individual: 1 stays equal to the father, 1 mutates & 1 crossover, and choosing always the best individual out of this three. The crossover was almost never selected less than 0.1%.

After this considerations we decided not to include any crossover and just do an elitist selection of the best individual.

At first this seemed to be really exploratory and that we were forgetting the exploratory part, but we decided to let that in the mutation.

2.4 Mutation

For this algorithm we have spent a lot of time to develop the best mutation. Just to start there are two possible types of mutation.

Point Mutation Selection from one polygon one vertex and/or a dimension of RGB and changing it randomly.

Polygon Mutation Select a polygon from the frame and changing the type of polygon, all the vertex and/or the full color.

The Polygon Mutation seems more exploratory while the Point Mutation seems more exploratory. In the Listing 3 we can see the Polygon Mutation. There is two given mutations the color & shape that enables that one, both or none of the mutations take place.

```
1  def mutate(population)
2      rand_individual = random.randrange(0, POLYGONS)
3      (width, height) = imagen.size
4      for rand_polygon in range(POPULATION_SIZE):
5          if color_mutation>=random.random()*100:
6              population[1][rand_polygon][rand_individual] = generate_colour()
7          if shape_mutation>=random.random()*100:
8              polygon_type = random.randrange(POLY_MIN_POINTS, POLY_MAX_POINTS + 1)
9              points=[]
10             for j in range(polygon_type):
11                 point = generate_point(width, height)
12                 points.append(point)
13             population[2][rand_polygon][rand_individual] = points
14     return population
```

Listing 3: Polygon Mutation

For mutation we will also discuss:

- How many vertex is best to mutate?
- What is probability of mutation that performs better?
- Can we apply a Stochastic mutation?
- Does it make sense to apply an Heuristic Mutation?
- Is it possible to combine stochastics & heuristics in mutation?

For a more detailed description of the code see the attached Jupyter Notebook where one of the many versions run to optimize the genetic algorithm is shown.

3 Results & Discussion

For this section we will discuss different strategies used to tackle the problem. In order to see the performance of the different strategies we will set a fixed number of iterations and the the evolution rate of the population.

Lets define first a **baseline model** that will be with a low single & simple mutation, polygons between 3-5 and 100.000 iterations.

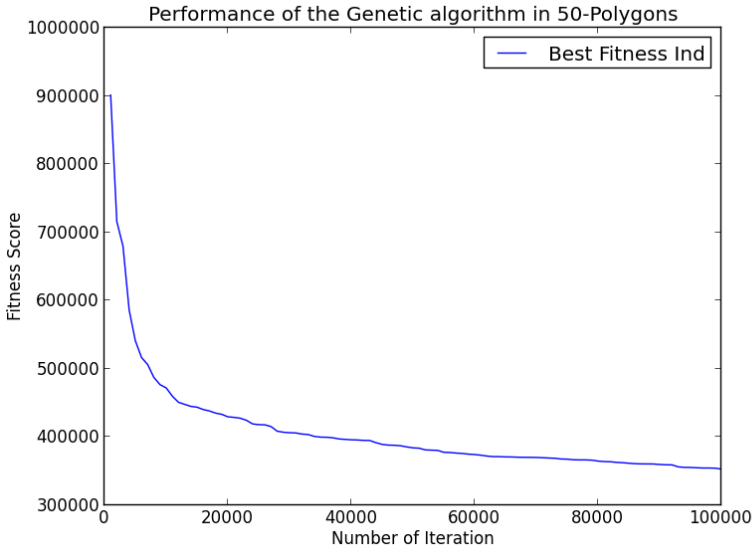


Figure 4: Performance evolution of the baseline model

We can notice that at the beginning the fitness drops drastically until 20k iterations where it starts behaving linearly and ends up with a fitness score of 350k.

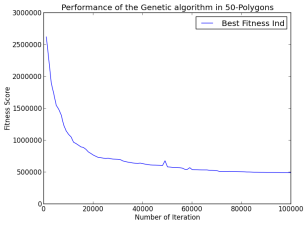
3.1 Polygons Vertex

We had the hypothesis where if the evolution of the algorithm was related to the number of vertex that each polygon could have so we decided to play with the parameters and check how does it affect the performance.

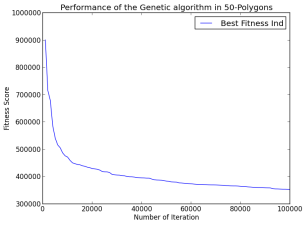
From this different parametrizations of the algorithm we can see that the figure (a) has slower decreasing rate after it starts being stable while figure (b) has it higher.

We can determine that smaller order polynomials are better for this case, so for optimal performance we will choose polygons of order 3 to 5.

Fitness evolution of changing the possible vertex of the polygons



(a) GA with vertex numbers in the range of 3-9



(b) GA only with triangles

3.2 Probability of Mutation

For this section we run an algorithm with a small probability of mutation for both mutation types.

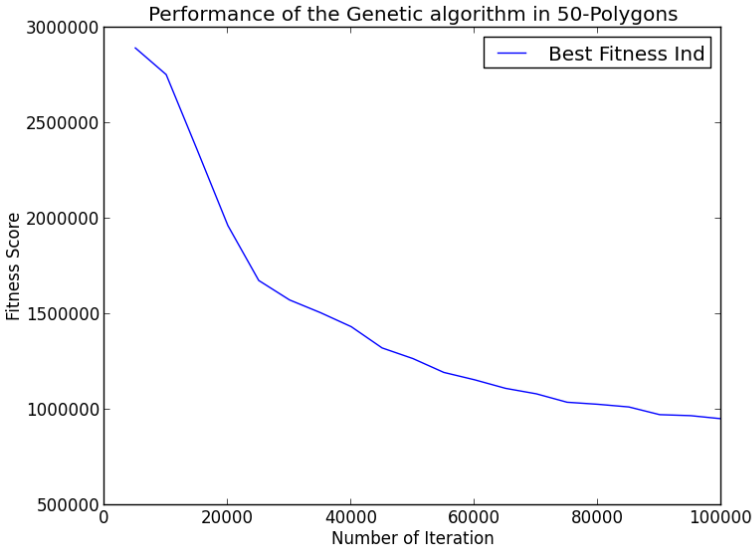


Figure 6: Performance evolution of the fitness value with a low mutation model

We can see that the learning rate is just smaller, so increasing the mutation makes the algorithm converge much faster. This actually make sense since the selection is made via tournament selection of the parent and child. A higher mutation just makes more possible improvements.

Also we define different mutations probabilities for different mutation types. Per example we tried $color_mutation = [0.1,' .25, 0.5, 0.9]$ & $shape_mutation = [0.1,' .25, 0.5, 0.9]$.

The optimal performance was with both probabilities set to 0.5

3.3 Mutation Type

The baseline model it's done with vertex mutation. In this section we will use polygonal mutation described as`Selecting a polygon from the frame and changing the type of polygon,all the vertex and/or the full color.`

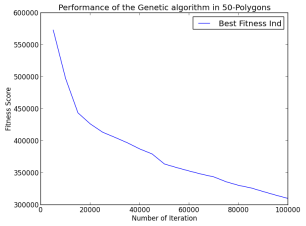
3.4 Stochastic mutation

Since we are only doing mutation we have to find a way to avoid that the algorithm doesn't get stock in a local minimima. Since now our mutation has been highly exploitatory.

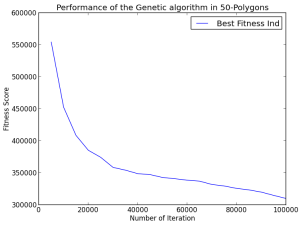
The idea for this section is to define a stochastic mutation that will allow our algorithm to mutate a random number of times. This will create a possibility that the polygon configuration can fully change.

We will try three different mutations changing the length of the stochastic vector from 5,10 & 15 in order to see the best performance.

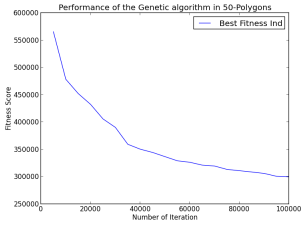
Fitness evolution with stochastic hyperparametrization



(a) GA with stochastic vector in range 0-5



(b) GA with stochastic vector in range 0-10



(c) GA with stochastic vector in range 0-15

For this algorithm we find a difference in the fitness evolution from others algorithms.

At the start the fitness doesn't drop as fast as other methods, which seems reasonable since it's a bit exploratory now. But afterwards the algorithm starts behaving linearly with a higher slope. At the end achieves the best performance so far. This algorithm is a candidate for a final long run where we will check the best result.

The difference between the different figures is that as the stochastic length increases the slope seems to be more linear. The best performing algorithm is the stochastic of 15 but it just performs slightly better than the others. The algorithm with the best final slope is (a).

4 Final Run

We have decided to run the best performing algorithm for a longer time and compile the images. The full evolution of images can be seen in my github account https://github.com/cmougan

The parameters for this run are:

- Point mutation since we have seen that the performance is better than the more general one.
- Stochastic length of 15, after seeing the graphs we noticed that the initial drop wasn't the best but after 100k iterations the slope was continuing increasing.
- Mutation probability of 0.5 for both kinds of mutation.
- 3 to 5 Polygon Vertex.

The best solution with 50 Polygons after 300 000 iterations for the Monalisa is

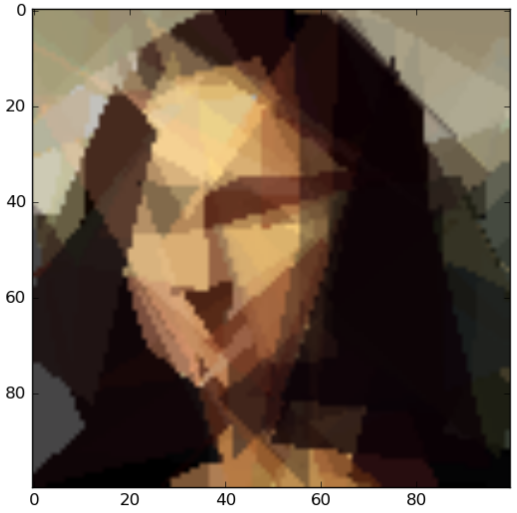


Figure 8: Optimal drawing of Monalisa

In the last images we could appreciate that the evolution didn't change at all, so we used an early stopping to stop the loop since it seemed that even afther a lot of time it wont improve much further.

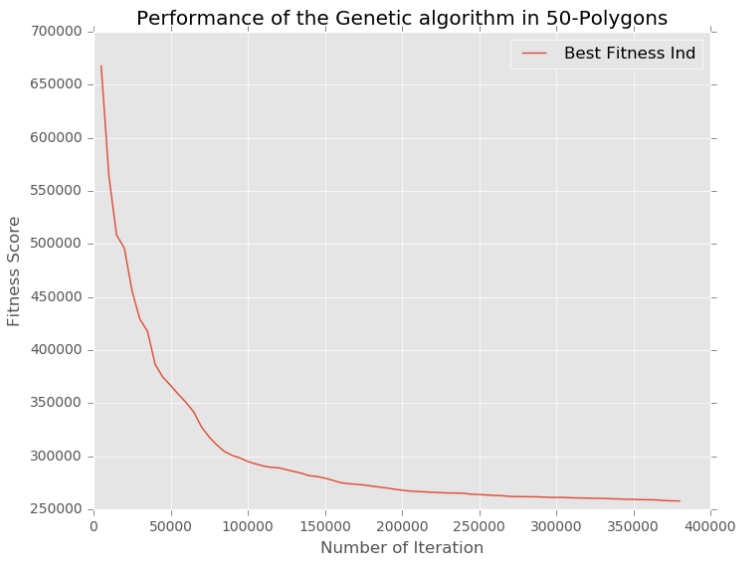


Figure 9: Evolution of the fitness function

For Figure 8 we didn't plot the initial points in order to be able to appreciate more precisely how the learning rate adjusts at the end. We can see how at the end the learning rate doesn't drop much and gets stable to a 250k fitness cost. This was the hyperparameter tunning that had the best performance.

5 Further Research

For this algorithm we have decided not to do any **crossover** since we didn't find any crossover that suit us. It would be possible that there is a crossover that will improve the algorithm efficiency.

One possible mutation/crossover will be to change the **order** in which polynomials are displayed. This is important due to the roll of transparency.

One of the best ideas to make the performance of the algorithm better will be to make an **heuristic mutation**. This way we could actually make the best mutation for a full polynomial or a vertex. With this philosophical argument we will make sure that the genome is converging in a extremely exploitatory way.

In case this is not enough we can define an **stochastic heuristic mutation** that could improve the algorithm efficiency since it will be exploitatory and exploratory at the same time, giving the opportunity to the algorithm to fast converge to a solution while avoiding the possibility of getting stuck in a local minima.

After all the discussion of the possible methods we have seen that some of the parameters of the algorithm make them converge faster in different parts of the algorithm. One possible idea here is to use **ensemble methods** from the different models to achieve an optimal performance. If we apply the optimal ensemble it will start droping faster at the beginning like polygon mutation, then it will start applying stochastic mutation of different lengths trying to find the fastest best solution possible.

6 Conclusion

In this project we have developed a genetic algorithm able to optimize the painting of an image with 50 polygons.

We have been able to solve the initial problem with different strategies. We have discuss the order of the polygons, the type of mutation, the probabilities, stochastic methods, heuristics & ensemblings. We can appreciate that the general idea of genetic algorithms is a powerful technique to optimize complex systems in a very intuitive way. For this case the computation cost is high since we were only able to appreciate the Monalisa after 45 minutes of computing time and best runs were more than 3 hours long.

One of the weakness of genetic algorithms is the tuning, some parameters can work for certain purposes but they won't for others. That's the reason why is important to understand Genetic Algorithms and their parameters, so we can build the solutions ad-hoc.

References

Mutation methods. URL https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm.

Luis Alseda. A list of genetic operations (crossover and mutation).

Sayan Nag Uddalok Sarkar. An adaptive genetic algorithm for solving n- 1. introduction: Queens problem. *Department of Electrical Engineering*, 2010.

Wikipedia. General information for genetic problems.