# OPTIMIZATION

## GENETIC PROGRAMMING – EVOLUTION OF MONALISA

Author: Lawrence Adu-Gyamfi

Date: 03/02/2019

## Table of Contents

# 1. INTRODUCTION

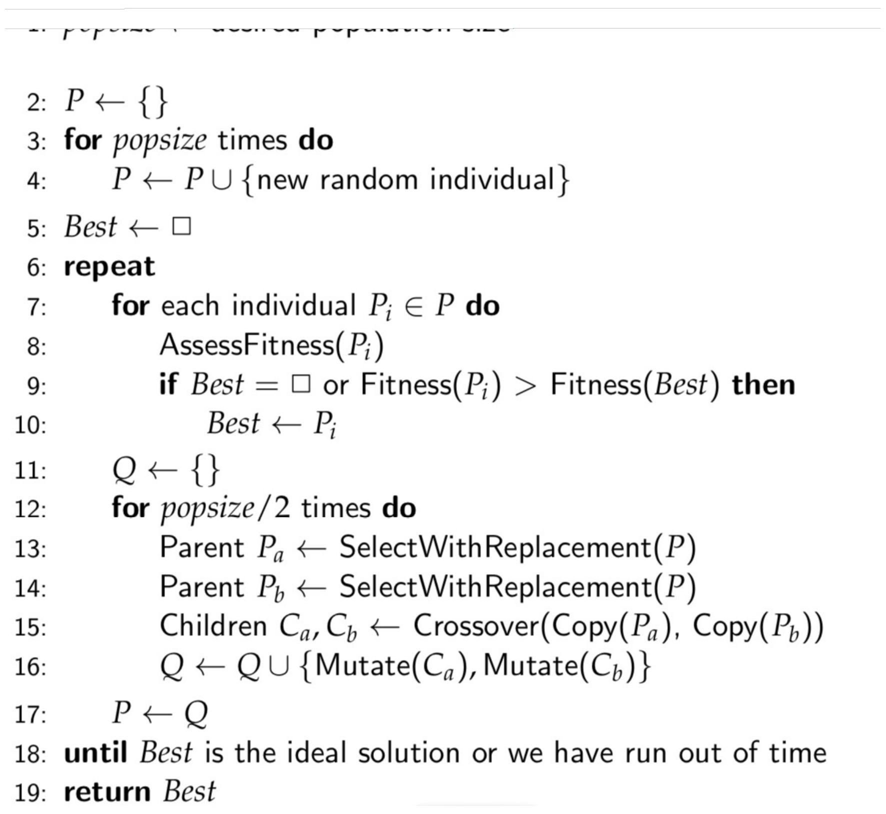
## 1.1. Genetic Algorithms Overview [5]

Genetic algorithms are part of a general bigger family of evolutionary algorithms that are useful for solving search and optimization problems that have large search spaces. It is able to arrive at a solution by continuously generating candidate solutions, evaluating how well the solutions fit the desired outcomes. The algorithm continues to improve the best solutions at every iteration until it arrives at the best solutions. This methodology is similar to the phenomenon of evolution based on the principles of natural selection.
A major underlining feature of genetic algorithms is the fitness function which serves as a feedback mechanism to the genetic engine and helps in making the better choice between two potential solutions. It combines the benefits of exploratory and exploitatory algorithms as part of its operation.

The typical working principle of genetic algorithms is outlined below:

1. Generate initial population of individuals based on the population size, with each representing a possible solution. This can be done randomly.
2. Assess the fitness of each individual in the population and assign each individual a fitness score based on how well they solve the problem.
3. From the parent population, generate a new population of children solutions of <u>**same population size**</u> as parents as follows:
    a. Select 2 parents from the parent population, with a bias towards fitter individuals, for reproduction. This is done with replacement.
    b. **Crossover** the two parents to produce two new children with a probability known as crossover rate. Normally this should more or less result in fitter children or better solutions.
    c. Perform a **mutation** operation on the two new children (solutions) based on a probability known as the mutation rate.
    d. Add these new children to the new population.
4. Replace the parent population with the new population. (which should normally be a population of fitter (better) solutions.
5. Repeat steps 2 -4 until the ideal solution is realized or the population converges (about 95% of the solutions in the population have the same makeup), or the maximum number of generations, if there is one, is reached.

A pseudocode of the algorithm is shown below: Ref. [2]:



```
1: $popsize$ ← desired population size
2: $P \leftarrow \{\}$
3: for $popsize$ times do
4:     $P \leftarrow P \cup \{$new random individual$\}$
5: $Best \leftarrow \square$
6: repeat
7:     for each individual $P_i \in P$ do
8:         AssessFitness($P_i$)
9:         if $Best = \square$ or Fitness($P_i$) > Fitness($Best$) then
10:            $Best \leftarrow P_i$
11:    $Q \leftarrow \{\}$
12:    for $popsize/2$ times do
13:        Parent $P_a \leftarrow$ SelectWithReplacement($P$)
14:        Parent $P_b \leftarrow$ SelectWithReplacement($P$)
15:        Children $C_a, C_b \leftarrow$ Crossover(Copy($P_a$), Copy($P_b$))
16:        $Q \leftarrow Q \cup \{$Mutate($C_a$), Mutate($C_b$)$\}$
17:    $P \leftarrow Q$
18: until $Best$ is the ideal solution or we have run out of time
19: return $Best$
```

## 1.2. Definition of Problem (Evolution of Monalisa Using Genetic Programming)

The report presents an implementation of genetic algorithm programming to regenerate the image of Monalisa. The task is to be able to generate the image of Monalisa (to a reasonable degree of resemblance) using a population of 50 polygons.

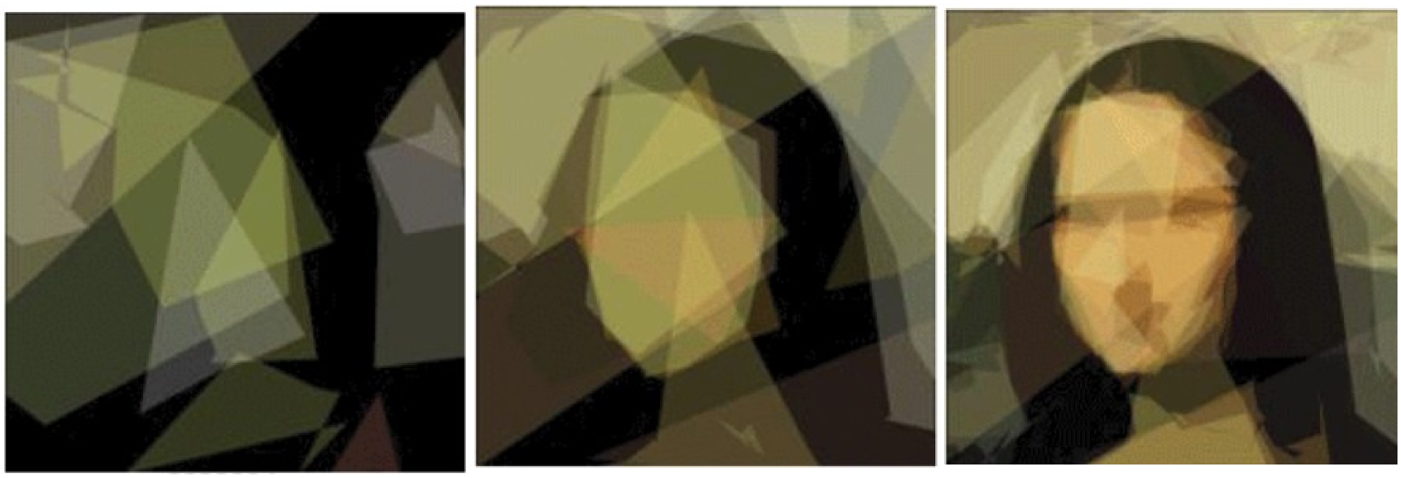


*Figure 1-1: Sample Solutions of the Problem. Ref. [6]*

## 2. METHODOLOGY

This section explains the methodology that has been implemented using the genetic algorithm to regenerate the image of Monalisa.
The solution has been implemented using the python programming language (python version 3.6) with additional packages from PIL, and numpy .

The execution of the program starts with a generation of the initial population which is made up of a number of images or drawings that are each the results of a combination of 50 randomly generated polygons.
A new generation of solutions is produced through the two (2) main evolutionary strategies (crossover and mutation) with the knowledge of the fitness of the best individual in the previous generation in mind. The new generation is created with the intention to improve on the previous solutions and thereby get closer to the target image. The closeness of each solution image to the target image is evaluated using a fitness function (described in detail in section 2.3)
The steps for the creation of the new generation and subsequent evaluation of the fitness of the individuals is repeated until the final solution is attained or for a certain number of iterations or generations.
The individual components of the algorithm are described in the following sections.

### 2.1. Individual

An individual or candidate solution in the implementation is represented as an image or drawing object that is made up of 50 polygons . The polygons which represent the genes in this case are identified individually by their specific colour and number of vertices. These are the attributes that can be perturbed during an evolutionary stage.
Each individual solution is generated using the target image as the template in terms of the size of the final image and the colour mode (RGBA) of the image.
The polygons (genes) that make up the image can have different possible number of sides ranging between 3 and 6 (can be adapted in the implementation), as well as a colour that includes a degree of transparency.
The following are some examples of typical solutions generated by the algorithm during the initial generations of the run.

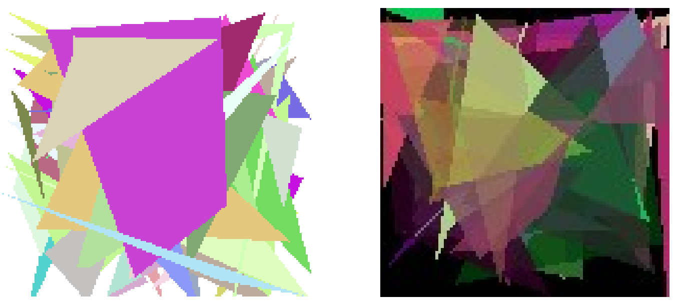


*Figure 2-2:Problem-specific Definition of Individual Solution*

## 2.2. Population

A population in this implementation of evolution of the Monalisa problem is represented by a set of individuals or solutions with each solution representing a combination of several polygons to form an image or drawing object as explained in section 2.1.

The initial population of the parents are generated randomly based on the population size specified and taking into account the restrictions of the genotype as described in section 2.1.

The size of the population has been considered as a hyperparameter in this implementation and has been altered as required to improve the performance of the algorithm.

For the final implementation, a population of two (2) individuals has been kept at the end of each generation. These individuals are selected from the sorted list of all individuals at the end of the evolution stages (crossover and mutation), meaning only the individuals with the best fitness are kept.

## 2.3. Fitness function

The fitness of each individual image or solution is defined in this implementation as how close the image is to the target solution which is the image of Monalisa.

This operation is done by comparing the generated solution to the target image (both of the same size) at per-pixel level. The sum of the absolute difference between the solution and target image for each pixel value at each colour channel (R, G, B, A) is measured and averaged over the number of individual pixels in the image.

The implementation makes use of a provision by the numpy package in python to quickly compute this multi-dimensional array operation.

Below is the code used for implementing this in the application:

```python
def fitness(image_1,image_2):
    """
    The fitness of each individual image or solution is defined in this implementation as how close the image
    is to the target solution which is the image of Monalisa.
    This operation is done by comparing the generated solution to the target image (both at the same size) at
    per-pixel level.The sum of the absolute difference between the solution and target image for each pixel value
    at each colour channel(R, G, B, A) is measured and averaged over the number of pixels. .
    """
    #Convert Image types to numpy arrays
    image1 = numpy.array(image_1,numpy.int16)
    image2 = numpy.array(image_2,numpy.int16)
    image_diff = numpy.sum(numpy.abs(image1-image2))
    return (image_diff / 255.0 * 100) / image1.size
```

## 2.4. Selection Methods

The parents of each population are selected for recombination (crossover and mutation) randomly based on a selection principle with more likelihood to favour the fittest individuals.

To make the algorithm exploitative, at the end of each evolutionary stage which involves a combination of crossover and mutation, the final population is sorted and only the top 2 individuals  with the best fitness values are carried over to the next generation. Automatically both individuals are chosen for recombination during the evolutionary stage of the generation.

## 2.5. Crossover Algorithm

As described in the previous section, individuals in the population are both chosen for the crossver or recombination stage. In this, the aim is to produce better solutions by combining parts of both individuals which are normally the best from the previous generation.
Crossover, however, is controlled using the crossover rate. For each generation, during crossover, the value of the rate is compared to a randomly generated number (between 0 and 1) and if the rate is greater, we proceed to generate a random integer to use as a crossover point. In order not to control the possibility of detoriorating the fitness of both parents, we limit the maximum value for the crossover to the number of polygons divided by 10.
The output of crossover is the new individual with the best fitness. This new individual is further compared to the fitness of the previous 2nd best parent and if better, we substitute the parent with the new child before proceeding to mutate them.

Below is the code used to implement the crossover in the application:

```
## -------------------------- CROSSOVER FUNCTION --------------------------------------##
"""
Funtion to perform crossover based on crossover rate, to produce new child from parents.
Crossover point is determined randomly but limited to a few polygons in order not to  change
the image so much.
Two children generated but only the best is kept.
"""

def crossover(parent_1,parent_1_fitness, parent_2, parent_2_fitness, crossover_rate):
    if crossover_rate > random.random():

        # Define crossover points relative to the number of polygons(max is 10% of number of polygons)
        crossover_pt = random.randint(0, len(parent_1.polygons)//10)
        temp = parent_1.polygons[:crossover_pt]

        #perform swap of polygons on parents
        parent_1.polygons[:crossover_pt] = parent_2.polygons[:crossover_pt]
        parent_2.polygons[:crossover_pt] = temp
        p1_fitness = fitness(parent_1.drawImage(), TARGET)
        p2_fitness = fitness(parent_2.drawImage(), TARGET)

        # check and keep only the best results
        if p1_fitness < p2_fitness:
            return (parent_1, p1_fitness)
        else:
            return (parent_2, p2_fitness)
    else:
        return(parent_2, parent_1_fitness)
```

## 2.6. Mutation

### 2.6.1. Mutation Algorithm

Mutation is implemented  for any solution or individual image in the application by perturbing the polygons that make up the image.
As described before, the parameters of a polygon that can be modified during each mutation stage includes :
- the resulting colour of the polygon. And for this, there is the opportunity to modify any of the individual RGBA values of the colour.
- The vertices of the polygon. This can be done  by modifying the individual points of the vertices, or changing the number of sides of the polygon by removing or adding vertices.

Below is the snippet of the code in the application that implements mutation:

```python
def mutate(self):
    # perform mutation only based on comparison of muation rate to a randomly generated number
    for polygon in self.polygons:
        if MUTATION_RATE > random.random():
            polygon.mutate()

    # Randomly add or remove a polygon based on the global respective rates
    if NEW_POL_RATE < random.random():
        self.polygons.append(Polygon(self.size))
    # check that there are still polygons in the image before proceeding to remove any polygons
    # based on the rate for removing polygons
    if len(self.polygons) > 0 and REMOVE_POL_RATE < random.random():
        self.polygons.remove(random.choice(self.polygons))
```

Mutation for an individual is initiated based on a mutation rate (between 0 and 1) which is specified as a global variable in the application. The mutation rate is compared to a randomly generated number and if greater than this number we proceed to make random choice between the parameters of the polygon, i.e. the number of sides, the position and the color, to modify.
Below is the code snippet as used in the implementation to mutate a polygon.

```python
# Mutation function for a polygon. This defines the details of how mutation is carried
# out for each polygon. A choice is made between the parameters. To control the exploitatory nature,
# a small mutation size is defined which is relative to each parameter
def mutate(self):
    mutation_size = max(1,int(round(random.gauss(15,4))))/100
    mutation_param =  random.choice(["sides","position", "color"])

    if mutation_param == "sides":
        self.sides = random.randint(MIN_POLGON_SIDES, MAX_POLYGON_SIDES)
    elif mutation_param == "position":
        self.pos = mutate_vertice(self.pos) # mutate_vertices explained in helper functions
    elif mutation_param == "color":
        r = min(max(0,random.randint(int(self.color[0]*(1-mutation_size)),int(self.color[0]*(1+mutation_size)))),255)
        g = min(max(0,random.randint(int(self.color[1]*(1-mutation_size)),int(self.color[1]*(1+mutation_size)))),255)
        b = min(max(0,random.randint(int(self.color[2]*(1-mutation_size)),int(self.color[2]*(1+mutation_size)))),255)
        a = min(max(0,random.randint(int(self.color[3]*(1-mutation_size)),int(self.color[3]*(1+mutation_size)))),200)
        self.color = tuple((r,g,b,a))
```

A further addition to the mutation to increase how exploitative the algorithm is, is the possibility to add or remove  random polygons to the solution being mutated. This is

controlled by two separate predefined rates which are compared a randomly generated number and effected as such depending on which side of the random number the value falls.
In order to explore several possibilities, the two parents are mutated for a number of times. This quantity has been considered as another hyperparameter. The final output of mutation is a list of new different individuals generated as a result of multiple mutations of the 2 previous individuals. This list is termed as the children population.
This implemented in the application with the following functions in code:

```python
def mutate_child(child):
    """
    Function to call mutation on a single image and return the result together with the new fitness
    """
    image = deepcopy(child)
    image.mutate()
    image_1 = image.drawImage()
    target = TARGET
    return (image, fitness(image_1,target))

def test_mutation(child,POP_TEST_SIZE):

    #Mutate child several times based test population size
    results = [mutate_child(child) for child in [child]*int(POP_TEST_SIZE)]
    return results
```

For the next generation this list of new individuals is sorted based on the fitness and only the top individuals are retained.

## 2.7. Convergence

Due to the nature of the problem and the way the solution has been drafted, the convergence of the algorithm has not been monitored in the usual way where the average fitness of the solutions is expected to converge towards that of the best solution.
However , the evolution of the best fitness is monitored and reported as well to show how it gradually reduces towards the optimal fitness over the generations. The results of this is reported in section 3.

## 2.8. Running The Application

The application can be run at the command line using the following code (supplying the location of the reference image as an argument:

```
$ python3 monalisa_evolution.py [location to target image file]
```

A folder (IMAGES) will be created to store the images every 10 generations (this can be modified). A results.txt file is generated in the same folder which logs the fitness of the best solution every generation.

The full code for the implementation is attached as part of submission of this report.

# 3. RESULTS

This section presents some results of running the implementation of the genetic algorithm towards solving the evolution of Monalisa problem.

## 3.1. Fitness Evolution over Generations

The following plot shows the gradual decline of the fitness of the best individual at the end of iteration over the generations. For the sake of the clarity, only results of up to the 10000th generation are shown, however running the application to about 92000 generations managed to achieve a minimum fitness of approximately **4.62 (down from 26.06) within 1 and half hours.**

On several occasions the evolution tended to get stuck on a particular solution especially as the solution got closer and closer to the optimum, however as a result of the implementation of the mutation and crossover, a new and better solution was realised, which helped the fitness to continue decreasing. This is as per the underlying philosophy of the genetic algorithms.

The parameters have been altered and modified on several occasions to improve the performance of the application. The results presented below are with crossover and mutation rates of 0.5 and 0.2 respectively. As explained in section 2.6 for the mutation philosophy, additional rates of addition or removal of polygons have been included. For the results presented below, these have been set at 0.3 and 0.2 respectively.

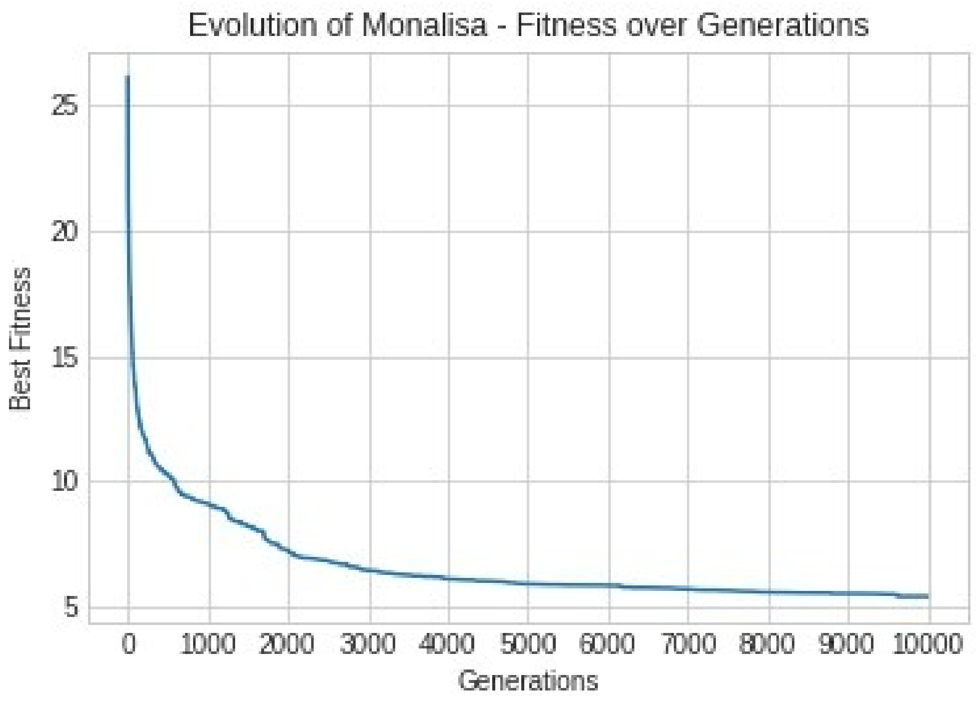


*Figure 2- 3: Fitness Evolution over  Generations*

## 3.2. Sample Solutions

Sample solutions of running the application are presented below showing the evolution of the solutions through the generations and as the fitness improved.

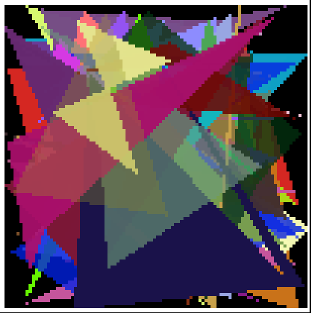
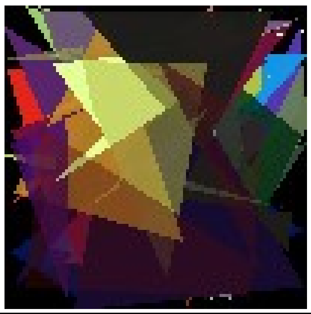
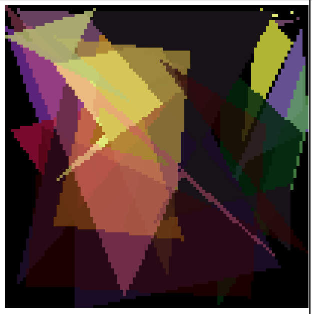
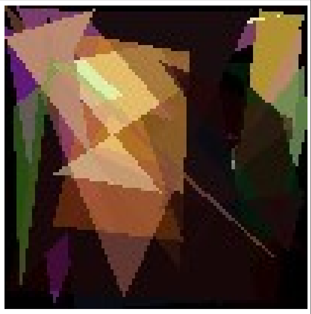
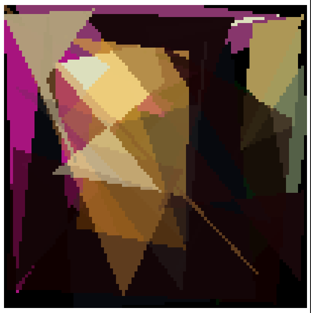
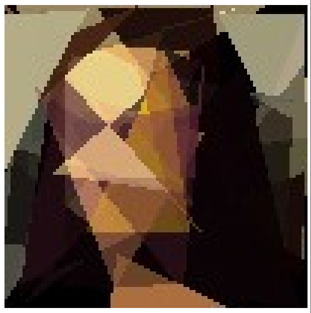
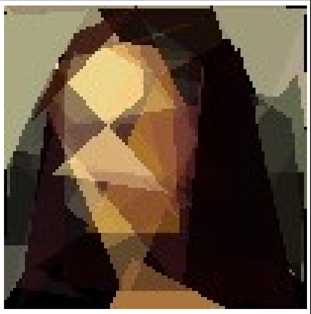
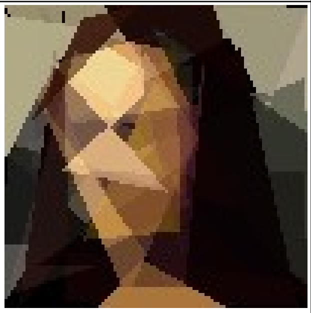
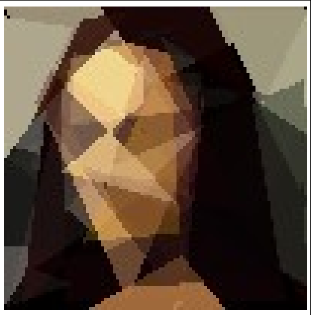
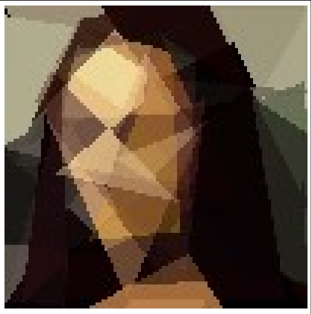
| | | | |
|---|---|---|---|
| Generation 1<br>Fitness = 26.06 |  | Generation 100<br>Fitness = 13.70 |  |
| Generation 200<br>Fitness = 11.77 |  | Generation 500<br>Fitness = 10.27 |  |
| Generation 1000<br>Fitness = 9.11 |  | Generation 5000<br>Fitness = 5.95 |  |
| Generation 10000<br>Fitness = 5.45 |  | Generation 20000<br>Fitness = 5.07 |  |
| Generation 50000<br>Fitness = 4.78 |  | Generation 90000<br>Fitness = 4.63 |  |

*Table 3-1: Sample Solutions*

## 4. CONCLUSION

As part of this project, the principles of the genetic algorithm have been implement to evolve towards a target image (the Monalisa in this case) starting from a canvas with 50 polygons. I have utilized the several principles of genetic algorithm including population definition and selection, recombination through crossover and mutation . In some situations, there have been minor additions to or deviations from the basic algorithms, however this is expected as these algorithms are meant to be redefined to suit the problem being solved.

The choice of the means for calculating the fitness did have an impact on the speed of convergence and eventually, an already existing package in python has been used to simplify the operation.

It was noticed that the definition of the crossover and mutation algorithms had a major impact on the behaviour of evolution. In some cases, not regulating how much the old points, colours or sides of polygons were modified during mutation tended to make it difficult for the fitness to converge quickly.

In this implementation, the effect of the  values chosen for the rate of crossover and mutations has had an impact on how quickly the algorithm progressed towards the optimum solution. This also affected how long the evolution got stuck at some particular solutions. In such cases, altering the rates accordingly tended to help increase the efficiency on the next run of the program. This confirms the notion that there is no hard and fast rule for choosing these parameters and several tries have to be made to determine the best working values.

## 5. REFERENCES

The following references have been used in one way or the other in preparation of this report. They have been referenced as and when applicable.

1. An Overview of Genetic Algorithms, *David Beasley*, *David R. Bull*, *Ralph R. Martin*
2. Introduction to Genetic Algorithms and their Applications (A lectures for the Autonomous University of Barcelona), *Jacek Dziedzic*
3. Essentials of Metaheuristics, A Set of Undergraduate Lecture Notes, *Sean Luke*
4. An Introduction to Genetic Algorithms, *Melanie Mitchel*
5. Research and Innovation – Genetic Algorithms – 8-Queens Problem, *Lawrence Adu-Gyamfi*, *Adria Fenoy*.
6. *https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/*