Genetic Algorithm for Mimicking Image

*Thomas Laverghetta*,   
Old Dominion University – Computational Modeling and Simulation Engineering

Please read over Assignment 2 System Requirements document (which accompanies this report) first to gain understanding of program overview.

# Algorithms

In this section, I will be discussing the *Genetic Algorithm* used to generate mimicking source images using triangles. A genetic algorithm is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution (Mallawaarachchi, 2017). The process of natural selection starts with fittest individuals having higher probability to survive, and therefore, will have offspring (inherit the characteristics of the parents and will be added to the next generation). Their offspring should be better than their parents and have better chances at survival. This process iterates until a steady state has been reached where the difference in fitness is minuet.

Algorithmically, the process of natural selection can be expressed into five phases: initial population generation, fitness function, selection, crossover, mutation. The initial population is the initial population to which natural selection can occur. Fitness function is a way of determining the likelihood of survival for individuals. Selection is the process of selecting individuals to have offspring. Crossover is the process of augmenting two individual’s chromosomes to generate a new chromosome (new offspring). Mutation is a random occurrence that modifies new offspring’s chromosome. This ensures that the population does not reach steady state too fast. The following is the algorithm for genetic algorithm:

*Genetic Algorithm ()*

*1. Generate Initial Population*

*2. Computer fitness scores for individuals in Population*

*3. while not converged {or number of generations created is less than max number of generations}:*

*4. for number of children per generation:*

*5. parent1, parent2 = Selection(Population)*

*6. child = Crossover(parent1, parent2)*

*7. Mutation(child)*

*8. Fitness(child)*

*9. end for*

*10. replace highest fitness scores with children generated*

*11. end while*

For this project, genes will be triangles, chromosome is a set of triangles to create image (individuals are images), and population is a set of images.

The following next sections will be pseudocode for fitness, selection, crossover, and mutation functions.

## Fitness Function

The fitness function, as stated before, calculates how fit an individual to survive (how close it is to the goal or source image for this application). The fitness score I will be using was taken from a genetic algorithm Peng Ding created and posted to GitHub (Ding, 2016). Since it is on GitHub, it is opensource. He calculated the root-mean-square difference between images. This method can also be found under example 19 of ProgramCreek’s Python PIL.ImageChops.difference() webpage (ProgramCreek, n.d.). The following is Ping Ding’s fitness that I used in my program (pseudocode mixed with Python syntax):

*Fitness (genetically generated image (img))*

*1. h = ImageChops.difference(source image, img).histogram()*

*2. fitness = sqrt(reduce(operator.add, map(lambda h, i: h \* (I ^2), h, list(range(256)) \* 3 / (area of source of image in pixels))*

*3. return fitness*

Note, img is equivalent to individual of the population.

## Selection Function

The selection function, as mentioned previously, selects two parents to mate to create new offspring. To select the new parents, I did a weighted random parent selector. Where individuals (parent candidates) with lower fitness score (less difference between source and generated image – desired individuals) will have higher probabilities. To do this, I used the cumulative density function (CDF) of inverse fitness scores on a sorted list of individuals (population). With this, random numbers can be calculated for parents and compared against CDF value until parent’s random number is less than CDF value. The algorithm is as follows:

*Selection (sorted list of individuals (population[]) by fitness)*

*1. get two uniform random numbers between 0 and 1 (rn1, rn2) for probability of two parents*

*2. create variable for cumulative probability values (CDF)*

*3. create two pointer variables to point at selected individuals (parent1 and parent2)*

*4. set parent1 & parent2 to null*

*5. create iteration variable (iter) for while-loop to iterate through list*

*6. set iter to zero*

*7. calculate sum of 1 / fitness for whole population and set it to totalInverseFitness*

*8. while parent1 or parent2 is null:*

*9. set CDF to 1 / (fitness of population[iter])*

*10. create variable p and set it to CDF / totalInverseFitness*

*11. if parent1 is not null and rn1 <= p:*

*12. set parent1 to point at population[iter]*

*13. end if*

*14. if parent2 is not null and rn2 <= p:*

*15. set parent2 to point at population[iter]*

*16. end if*

*17. end while*

*18. return parent1 and parent2*

## Crossover Function

The crossover function, as mentioned previously, is the process of augmenting two parents to generate a new chromosome (new offspring/individual). To do this, some genes from one parent will be added to child and some genes from another parent will be added until a child has the same number of genes as each parent (gene length). The distribution of genes from each parent is around 50%. Where the probability of getting a gene from a parent is 50%. Although, for this project, I decided to allow for user to make a skewed distribution. This allows for a higher probability of either parent one or two to pass their genes to the child. The pseudocode for crossover is the following:

*Crossover (parent1, parent2)*

*1. get crossover probability of parent1 (cp)*

*2. create child*

*3. for i = 0 : gene length:*

*4. if random number <= cp:*

*5. child gene i equals parent1’s gene i*

*6. else:*

*7. child gene i equals parent2’s gene i*

*8. end if*

*9. end for*

*10. return child*

## Mutation Function

The mutation function, as mentioned previously, is a random occurrence that modifies new offspring’s chromosome. To do this, a mutation probability is set which dictates the probability of mutation occurring on an individual gene, and a mutation is created to change the gene. The probability for mutation is user specified, so no constant value will be set. Mutations on individual genes will be a random rearrangement of triangle (changing position of triangle) and randomly selecting a new color for triangle. Do not forget, a triangle is a gene. From all this, the following was created:

*Crossover (child)*

*1. get mutation probability (mp)*

*3. for i = 0 : gene length:*

*4. if random number <= mp:*

*5. mutate child’s gene (triangle) i*

*6. end if*

*7. end for*

# Results

Taking from algorithms discussed in the previous section, I created a Python script which allows users to input an image then output a mimicking image generated by genetic algorithm. To test this script and the algorithm implementation, I created several test scenarios. First test is determining the level of complexity allowed from a source image to generate a somewhat acceptable mimicking image. This tests the algorithm’s ability to produce mimicking images from source image. For the second test, I choose an image with mid-complexity, and change genetic algorithm parameters to see if I can make the image look better.

## Complexity Analysis

For this test three images were chosen with different levels of complexity. Figure 1 shows images. The far-left image (Italian flag) is my mid-level complexity test, middle image (ODU logo) is my high-level complexity test, and lastly, far right image (black image) is low-level complexity test.

![Shape

Description automatically generated](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAZAAAAELBAMAAAAFMM1/AAAAFVBMVEX///8AkkbOKzf10tWY07QAjT7NJTJXwIMQAAAA8UlEQVR42u3PwQAAAAgEsBRSCCKS/FmCuO9msOrYVWwnJiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIikniFNUdjvhgMbwAAAABJRU5ErkJggg==)Logo, company name

Description automatically generatedA picture containing shape

Description automatically generated

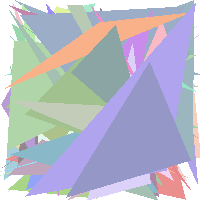
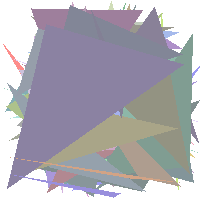
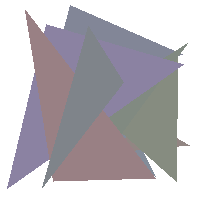
Figure 1. Input images

The images shown in figure 1 were inputted into the genetic algorithm. The algorithm parameters were population size set to 25-images, 100-triangles per individual, 5-children per generation, 0.5-probability for parent1 crossover, and 0.05-mutation probability.

To keep it simplistic, I will only be showing generations 0, 1000, and 100,000. The remaining data can be found within data folder (MSIM-480/Assignment2/Documentation/”Data Folder”). The data is formatted with “<image name>\_pop<population size>\_genelength<number of triangles>\_mut<mutation probability>\_cross<crossover probability>”.

Following is the output images:

* Black Image

* Italian Flag

* ODU Logo

## Sensitivity Analysis

For this test, I inputted the Italian flag from figure 1 (far left image) with different parameter values to determine performance. The parameter I changed were, population size, number of triangles per individual, probability of parent1 crossover, and mutation probability. These changes cumulated into five sample tests which were inputted into the system:

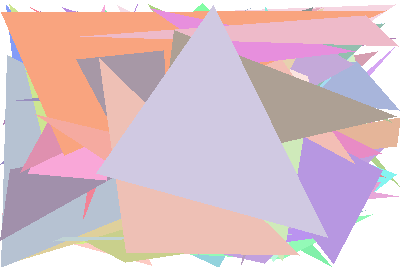
1. Population = 25, Number of triangles = 100, Probability of crossover = 0.5, and Mutation probability = 0.05
2. Population = 25, Number of triangles = 100, Probability of crossover = 0.5, and Mutation probability = 0.10
3. Population = 25, Number of triangles = 200, Probability of crossover = 0.5, and Mutation probability = 0.05
4. Population = 40, Number of triangles = 100, Probability of crossover = 0.5, and Mutation probability = 0.05
5. Population = 25, Number of triangles = 100, Probability of crossover = 0.7, and Mutation probability = 0.05

I inputted tests into the system and the following images were produced. Just in Complexity Analysis section, to keep it simplistic, I will only be showing generations 0, 1000, and 100,000, and the data can be found in the same place.

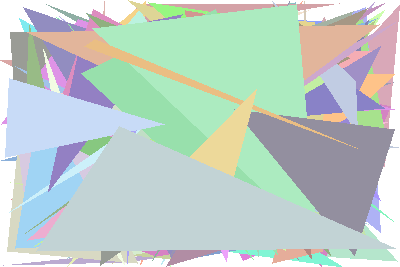
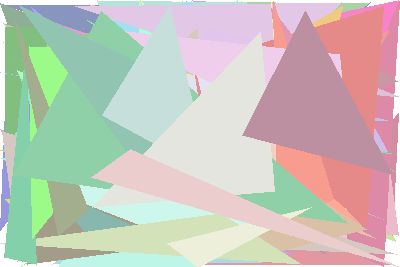
1. Population = 25, Number of triangles = 100, Probability of crossover = 0.5, and Mutation probability = 0.05

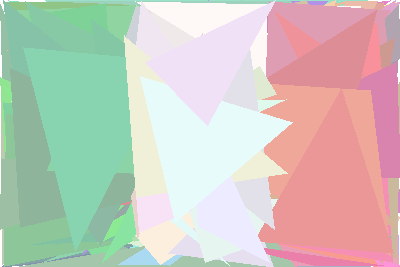
1. Population = 25, Number of triangles = 100, Probability of crossover = 0.5, and Mutation probability = 0.10

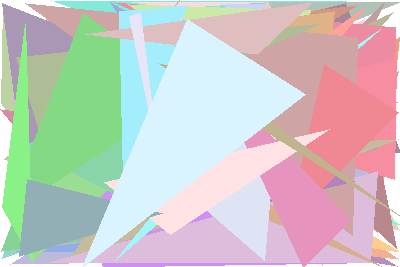
1. Population = 25, Number of triangles = 200, Probability of crossover = 0.5, and Mutation probability = 0.05

1. Population = 40, Number of triangles = 100, Probability of crossover = 0.5, and Mutation probability = 0.05

1. Population = 25, Number of triangles = 100, Probability of crossover = 0.7, and Mutation probability = 0.05

Based on the outputs for the five samples test, all produce outputs at generation 100,000 that resemble source image (Figure 1. Italian Flag); although, some produced slightly better images. Right off the bat, out of all tests, sample test 1 was the worst due to a purplish triangle in the center of the image. This purple triangle was produced at generation zero and stayed until generation 100,000. On like sample test 2, which had a similar purple triangle that disappeared between generation 1,000 and 100,000. Therefore, producing a more similar image to the source image. Although, sample test 2 did have problem of its own. Within the red domain of the flag, it had yellow and tan triangles that other sample tests were able to eliminate by having triangles that were closer to red color. For example, sample test 4 had dark purples mixed with red triangles to make the decolorization less noticeable. Other than those critiques, all the sample test produce similar images to the source image.