Understanding Optimization problems using Genetic Algorithms

Paulo Sousa Oliva da Silva 17011541

1. INTRODUCTION

Optimization problems are problems with finding the best solutions from all the possible solutions on a given problem. Genetic Algorithms (GA) on the other hand, are normally used to achieve solutions for optimization and search problems. It is a process inspired on biological operators such as the selection of the parents, the crossover of the selected parents and finally, the mutation in order to produce a better offspring with better fitness.

The aim of this assignment is to understand how optimization problems inspired by biological factors and natural occurrences can be used in computing to find a best solution to a given problem using different fitness functions.

2. BACKGROUND RESEARCH

Aguston Eiben (2007) states that "The main problem is that the description of a specific EA contains it components, such as choice of representation, selection, recombination, and mutation operators", the different methods of components will reflect how well or how fast a Genetic Algorithm can resolve the problem at hand.

Genetic Algorithms have been used in many different fields to find solutions to specific problems and find results. NASA used GA's to try to find the best shape an antenna should have in order to achieve the best radiation patterns possible within certain criteria.

Another example is in the field of Economics where *Shu-Heng* (1996) states that GA "can capture several features of the experimental behaviour of human subjects better than other simple learning algorithms" and later that " GA could be considered an appropriate choice to model learning agents in a multiagent system." GA was used on the Cobweb model which is an economic model that explains why prices can be affected by periodic fluctuations in certain markets.

GA's are normally used to solve very hard problems, and is able to search for optimal or close to optimal solutions. The main reason of selecting GA over other heuristic algorithm is due to the simplicity and yet powerful capability of GA to find a solution.

By using nature as an example, GA's can create and find the best solution within a certain population being created at random or manually chosen. As stated before, this is achieved by using different methods of evolution:

- Selection: a selection technique is used to choose the parents from a population.
- Crossover (or recombination): where the parents will swap their genetic code in order to create a better offspring.
- Mutation: it happens naturally to help the offspring survive the current environment.

These are the steps of evolution that the GA is based on as well as being the ones used for this assignment.

3. EXPERIMENTATION

3.1 Preparing the algorithm

Genes

The main feature of the GA is the gene or chromosome. This is what will be used to represent the data for each individual of the population. The goal of the assignment is to find the best solution to 3 different problems: Function 1 is a maximization problem and Function 2 and 3 are minimization problems.
For Function 1, each individual will have to possess a fitness between 0 and 255 meaning it will need to be represented in binary with 8 bits, i.e. 10101101.
In Function 2 however, each individual will need to have a fitness between -15 and 15 as well as two variables x and y. This means that it needs to be represented by 10bits 5 bits for x and 5bits for y.
Lastly, Function 3, in which each individual will need to have a fitness between -5.12 to 5.12. However, this Function requires real values with n = 10 or 20, so each individual will have n real values as their genes.

Initiation

The initial population will be randomly created using randomized genes for each Function, using n times according to the size of the population.

Tournament Selection

For the selection method, Tournament was the chosen one as it allows each individual to have the exact same probability to be picked from the pool of individuals. Tournament is also good to avoid getting stuck in a local maximum or minima and no fitter solutions available.
The selection works by selecting two individuals at random, checking their fitness and copying the one with the best fitness to a temporary population. The step is then repeated once again and the two parents are then selected and the other steps are performed, crossover and then mutation.

Crossover

Single point crossover was used in all functions and it works by having a random number generator (RNG) pick a point in which the two parents will then change their tails.

Mutation

Mutation applies a small change to the population and without it, the solution would most likely fall in a local best, which is not the goal. To avoid this, mutation comes into place and it mixes the genes of an individual to help push the solution towards the goal.

Evaluation

This will be used to find the fitness of each individual using the different Functions for this assignment. The best fitness and average will be recorded from each generation in order to record progress of the GA.

These steps will be used in all 3 Functions for this assignment, however, some parameters will be changed in order to understand their roles and what they affect overall. These changes will be mentioned where they take place.

The figures in this report will have a caption with their respective mutation probability (pm), probability of crossover (pc) and population size (pops). All graphs are an average of 5 runs, with labels on each line: Red for average, and Blue for best.

3.2 Function 1

Function 1 is a maximization problem. The fitness function that was used is: f(x) = x^2 with x being between 0 and 255. Each individual is represented with 8 bits.

Gene Encoding: each individual has randomly selected 8 bits of 0’s or 1’s i.e. 10001101 -> 141 after converting from binary to denary the value 141 is fed into the function to calculate the individuals fitness, 141^2 = 19881.

Fitness

The fitness of each individual is calculated by transforming the binary string into a denary number and getting the x value. This is then put into the fitness function and stored in the individual.

Results

The expected results are that the average fitness of the entire population will converge into the best fitness of the population. This happens due to Function 1 keeping always the best individuals as offspring so that the overall fitness of the entire population will increase over time.

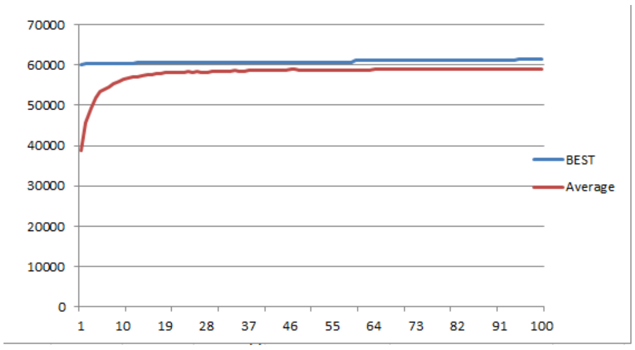

Figure 1 pm: 0.001 pc: 0.6 pops: 10

Figure 1 shows the expected the convergence of the average fitness of the population with the best fitness of each generation. We can also observe the fitness increase over the course of 100 where the fitness starts to increase rapidly but then seem to plateau. Over the 100 generations the average fitness is much higher than it was originally. This combination of pm and pc was also the one that created the most stable results.

Parameter Testing

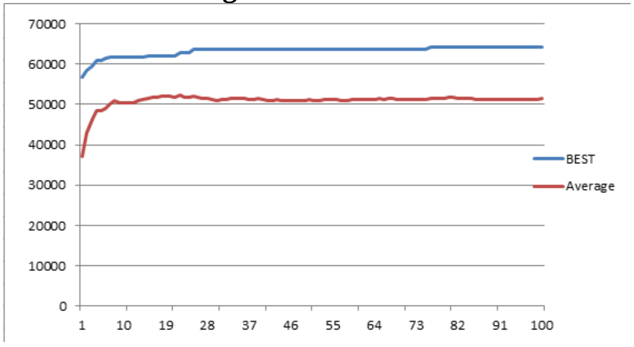

Figure 2 pm: 0.01 pc: 0.6 pops: 10

As can be observed in **Figure 2** with higher mutation probability the average fitness of the population still increases, however, there are some generations where the fitness decreases and demonstrates erratic behaviour.

In **Figure 2,** fitness still increased however, it can't be assured that it will always be due to mutation, which cannot assure if it will increase of decrease fitness. In addition, having a higher probability of this happening will not create stable results. From **Figure 1** we can observe a set of parameters that will constantly produce results and increase the average fitness over N generations.

3.3 Function 2

Function 2 is a minimization problem and the fitness function used was: *f(x,y)* = 0.26.(*x2* + *y2*) – 0.48.*x.y* where x and y have to be values between -15 and 15, so x will have 5 bits to be represented. To represent x and y in an individual, 10bits will be needed by using sign and magnitude.

Gene Encoding: each individual has randomly selected 10 bits of 0's or 1's i.e. 1000101101 -> the binary string then is converted into 2 values, x and y. So:

$$\begin{tikzcd}
10001 \arrow[d] & 01101 \arrow[d] \\
x \arrow[d] & y \arrow[d] \\
-1 & 13
\end{tikzcd}$$

Then x = -1 and y = 13 are fed into the function to calculate the individuals fitness.

Results

The expected results are that the average fitness of the entire population will converge into the best fitness of the population. However, in case of Function 2, we expect the fitness to decrease overtime as Function 2 is a minimization problem. This also happens due the function 2 keeping always the best solutions from each selection so that the overall fitness of the entire population will decrease over time.

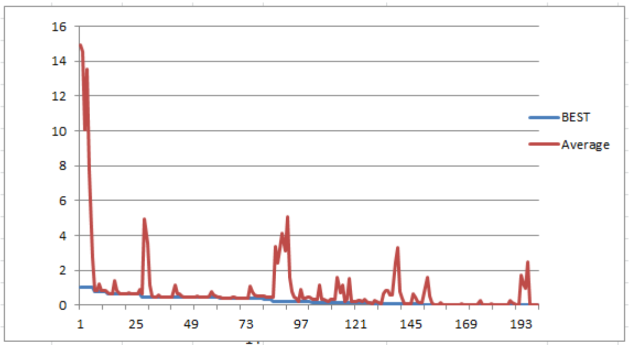

Figure 3 pm: 0.001 pc: 0.6 pops: 10

Figure 3 shows the convergence of the average fitness to the best fitness of each generation until it reaches a plateau. We can also observe that it took more than Function 1 to achieve this goal, for Function 2, it took 250 generations. Within 250 generations the population would always reach the goal, 0 fitness.

Just like in **Figure 1**, **Figure 3** set of parameters proved to be the most affective in order to get constant results that pushed the solution to its best solution.

Parameter Testing

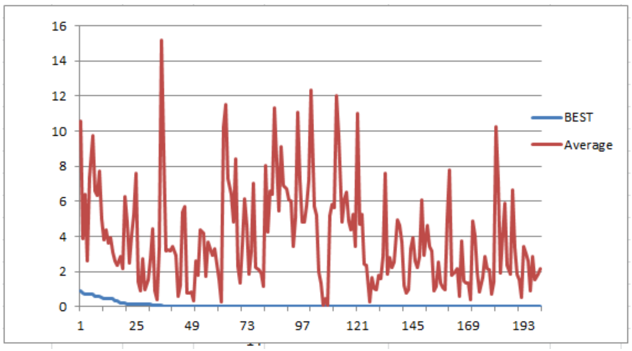

Figure 4 pm: 0.01 pc: 0.6 pops: 10

Figure 4 was made with the same parameters as **Figure 2.** However, we can observe how Function 2 is much more sensitive as it shows changes in mutation. We can also observe that the overall average fitness did not always go down, but would always converge back to the best solution. Mutation would make the average fitness spike quite often.

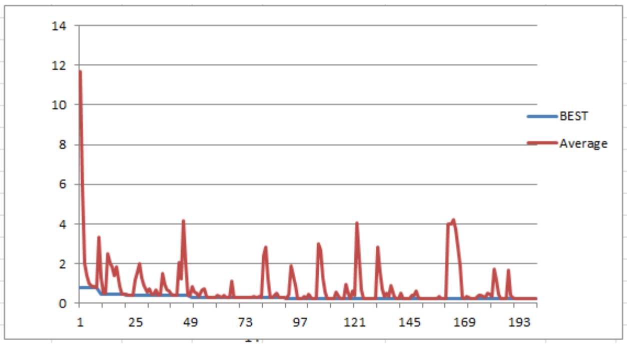

Figure 5 pm: 0.001 pc: 0.1 pops: 10

Figure 5 shows that with low crossover, the solution did not reach 0 which was the overall goal for this Function. However, it can be observed that the best fitness still went down in the beginning, which is mainly due to the use of elitism where we substitute the worst offspring with the best parent of a generation. This will allow for the solution to reach the goal in a much easier way.

Even with the low crossover probability, the average fitness still converged to the best fitness despite the solution not hitting its goal.

3.4 Function 3

Unlike the previous Functions were the data was represented using binary strings, Function 3 uses floating point number, ranging from -5.12 to 5.12.
To compensate for the floating-point numbers, a new way was required to represent the data within the gene.
Each gene within the individual will hold a value between -5.12 to 5.12 that was randomly generated using a RNG and by populating each gene of an individual at the start of each generation.

Function 3 is also a minimization problem and the fitness function used was: *f(x)* = 10*n* + sigma x^2–10.cos (2.pi.X).

Gene Encoding: each individual has randomly selected 10 floating number ranging from -5.12 to 5.12 i.e.: individual 1= {-1.20, 2.55, 3.58, -4.31, 5.00, 0.89, -2.41, 1.78, -4.30, 3.10}.

Fitness

Unlike the previous functions, the gene already has a decimal number so each gene can be fed into the function to calculate an individual's fitness.

Results

The expected results are that the average fitness of the entire population will converge into the best fitness of the population. As Function 3 is also a minimization problem, it is expected that the overall average fitness will decrease and converge to the best fitness.

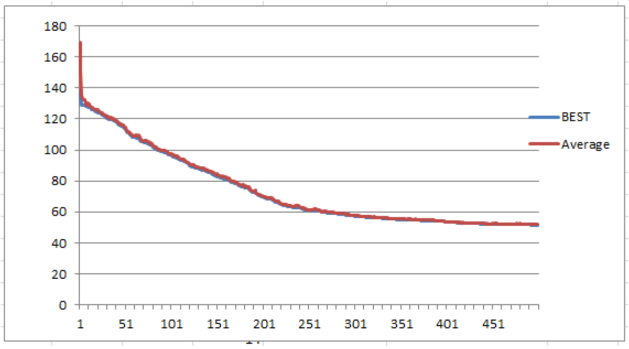

Figure 6 pm: 0.1 pc: 0.8 pops: 10 n: 10

Figure 6 shows that the average fitness does indeed decrease overall and converge to the best fitness of the solution. On Function 3, we have a new parameter to be explored which is the amount mutation affects the individual. In this case, the value ranges from -0.02 to 0.02 and this makes small but constant changes to the fitness, making it decrease overtime. This Function takes the most generation needed to start plateauing which is normally around the 400/450 generations.

We can also observe in **Figure 6** that pm is much higher than the previous Function, which in this case is needed.

Parameter Testing

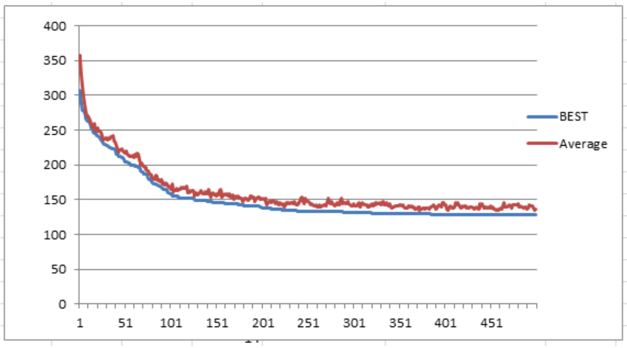

Figure 7 pm: 0.1 pc: 0.8 pops: 10 n: 20

Figure 7 shows a change in parameters on mutation and using n=20 instead of 10, while **Figure 6** shows the range in which mutation could change the genes, which was between -0.02, and 0.02. In **Figure 7** the value ranged between -0.1 and 0.1. This change also allows the overall average fitness to converge into the best fitness and overall decrease overtime. However, we can see the erratic behaviour just like **Figure 2** and **Figure 4.** It is also important to notice that the lower the mutations range the steadier the fitness would decrease overtime but taking considerably longer to achieve results. For example with a mutation range of -0.01 to 0.01 it would normally take around 800 generations for the fitness to plateau however if the range is -0.02 to 0.02, just like in **Figure 6** it takes around 450/500 generation to get the same result, saving computational power, while still producing steady results.

4. CONCLUSIONS

GA is an extremely powerful, easy to implement and overall a very reliable algorithm. The results achieved in this paper have shown that it is possible to get results and finding the solution is possible.

The result of this experiment has shown that optimization problems using GAs are possible. Having said this, it did also show that the parameters set at the beginning of the algorithm are necessary and tweaking them to achieve the desired results, using Adaptive Gas, could potentially solve this issue. However, it is also inefficient to determine the last mutation to find the global maximum. Parameter setting played a huge role on this assignment as it proved the reason why the program would produce results or not, however small the change in the parameters it can be very noticeable as shown in the various Figures.

If this project was to be carried out again in the future, more features of GA would be used on Function 3, especially Adaptive GA, to try to push the rest of the average fitness to the absolute minimum.

The production of this project was very beneficial as it give a clear overview of GA, how they can be used for optimization problems and how important parameter settings can be while setting up a GA.

REFERENCES

A. Eiben, Z. Michalewicz, M. Schoenauer and J. Smith (2007) *Parameter Control in Evolutionary Algorithms*

D. E. Goldberg (1989) *Genetic Algorithms in Search, Optimization & Machine Learning*

D. Photeinos, S. Giannareli and D. Tsahalis (2004) *Optimisation of the Parameters of a Back-Propagation Neural Network using Genetic Algorithms*

F.G. Lobo, C.F. Lima and Z. Michalewicz (2007) *Parameter Setting in Evolutionary Algorithms*

S. Das and P.N. Suganthan(2010) *Problem definitions and Evaluation Criteria for CEC 2011 Competition on Testing Evolurionary Algorithms on Real World Optimization Problems*

S.H. Chen and C.H Yeh (1996) *Genetic Programming Learning and the Cobweb Model.* Paper published in *Advances in Genetic Programming 2*

APPENDIX

1) All source code for the project can be found here: https://github.com/boapeca/BioComputation

2) Source code for assessing fitness for F1:

```
def parent_evaluate():  # f=x^2 for parent
    global totalPopFitness
    totalPopFitness = 0
    populationLength = len(population)

    for e in range(P):
        population[e].fitness = 0
        if population[e].gene[0] == 1:
            population[e].fitness += 128

        if population[e].gene[1] == 1:
            population[e].fitness += 64

        if population[e].gene[2] == 1:
            population[e].fitness += 32

        if population[e].gene[3] == 1:
            population[e].fitness += 16

        if population[e].gene[4] == 1:
            population[e].fitness += 8

        if population[e].gene[5] == 1:
            population[e].fitness += 4

        if population[e].gene[6] == 1:
            population[e].fitness += 2

        if population[e].gene[7] == 1:
            population[e].fitness += 1

        population[e].fitness = population[e].fitness **
2
        totalPopFitness += population[e].fitness
```

3) Source code for assessing fitness for F2:

```
def parent_evaluate():  # #fitness by f(x,y) = 0.26.( x2
+ y2 ) – 0.48.x.y for parent
    global totalPopFitness
    totalPopFitness = 0
    populationLength = len(population)

    for e in range(P):
        population[e].fitnessX = 0
        population[e].fitnessY = 0
        population[e].fitness = 0

        if population[e].gene[1] == 1:
            population[e].fitnessX += 8

        if population[e].gene[2] == 1:
            population[e].fitnessX += 4

        if population[e].gene[3] == 1:
            population[e].fitnessX += 2

        if population[e].gene[4] == 1:
            population[e].fitnessX += 1

        if population[e].gene[6] == 1:
            population[e].fitnessY += 8

        if population[e].gene[7] == 1:
            population[e].fitnessY += 4

        if population[e].gene[8] == 1:
            population[e].fitnessY += 2

        if population[e].gene[9] == 1:
            population[e].fitnessY += 1
```

```
        if population[e].gene[5] == 1:
            population[e].fitnessY *= -1

        if population[e].gene[0] == 1:
            population[e].fitnessX *= -1

        population[e].fitness = 0.26 *
((population[e].fitnessX ** 2) +

(population[e].fitnessY ** 2)) - \
            0.48 * population[e].fitnessX *
population[e].fitnessY
        totalPopFitness += population[e].fitness
```

4) Source code for assessing fitness for F3:

```
def parent_evaluate():  # Parent Fitness : 10N + sigma
X^2 -10 cos(2.pi.X)
    global totalPopFitness
    totalPopFitness = 0

    for e in range(P):
        population[e].fitness = 0
        for i in range(N):
            population[e].fitness +=
(population[e].gene[i] ** 2) - (10 * math.cos(2*math.pi
* population[e].gene[i]))
        population[e].fitness += (10 * N)
        #population[e].fitness =
round(population[e].fitness, 2)

        totalPopFitness += population[e].fitness
        totalPopFitness = round(totalPopFitness, 2)
```

5) Tournament Selection:

```
def tournament():
    global roundNum
    global tempPopulation1
    tempPopulation1 = copy.deepcopy(population)
    tempPopulation2 = copy.deepcopy(population)
    for i in range(0, P):
        check = 0
        check2 = 0
        while check != 1:
            parent1 = random.randint(0, P - 1)
            parent2 = random.randint(0, P - 1)
            if parent1 != parent2:
                check = 1
        # First parent is choosen
        if population[parent1].fitness <=
population[parent2].fitness:
            parent3 = parent1
        else:
            parent3 = parent2
        while check2 != 1:
            parent1 = random.randint(0, P - 1)
            parent2 = random.randint(0, P - 1)
            if parent1 != parent3 or parent2 != parent3:
                check2 = 1
        # Second parent is chosen
        if population[parent1].fitness <=
population[parent2].fitness:
            crossover(parent3, parent1)
        else:
            crossover(parent3, parent2)

        #  The index of the selected parents are sent to
the crossover function
```

6) Crossover:

```
def crossover(parent1cross, parent2cross):
    global Pc
    global crossPoint
    global offspringCheck
    global tempPopulation2

    crossPoint = random.randint(0, N)  # point of
```

```
crossover
    cross = random.uniform(0, 1)
    offspringCheck = len(tempPopulation1)

    if cross < Pc:
        for i in range(crossPoint, N):
            tempPopulation1[parent1cross].gene[i] =
tempPopulation1[parent2cross].gene[i]
    mutation(parent1cross)
```

7) Mutation for F1 and F2:

```
def mutation(tomutate):
    global Pm
    mutated = False
    global offPopCheck
    temp = 0
    mut = random.uniform(0, 1)
    for j in range(N):
        mut = random.uniform(0, 1)
        if mut <= Pm:
            if tempPopulation1[tomutate].gene[j] == 0:
                tempPopulation1[tomutate].gene[j] = 1
            elif tempPopulation1[tomutate].gene[j] == 1:
                tempPopulation1[tomutate].gene[j] = 0

    offspring.append(tempPopulation1[tomutate])
```

8) Mutation for F3:

```
def mutation(tomutate):
    global Pm
    mutated = False
    global offPopCheck

    for j in range(N):
        mut = random.uniform(0, 1)
        if mut <= Pm:
            num = random.uniform(-0.1, 0.1)
            tempPopulation1[tomutate].gene[j] += num

            if tempPopulation1[tomutate].gene[j] <= -
5.12:
                tempPopulation1[tomutate].gene[j] = -
5.12

            elif tempPopulation1[tomutate].gene[j] >=
5.12:
                tempPopulation1[tomutate].gene[j] = 5.12

    offspring.append(tempPopulation1[tomutate])
```

9) Elitism:

```
def elitism():
    worstIndex = 0
    bestIndex = 0
    global population
    worstTemp[0] = population[0]
    for i in range(P):
        if offspring[i].fitness > bestTemp[0].fitness:
            bestIndex = i
            bestTemp[0] = offspring[i]

        if population[i].fitness <=
worstTemp[0].fitness:
            worstIndex = i
            worstTemp[0] = population[i]

    # Elitism, copies worst parent in offspring pool to
become new parent pop
    offspring[bestIndex] = population[worstIndex]
    population = copy.deepcopy(offspring)
    off_evaluate()

    offspring.clear()
    tempPopulation1.clear()
    tempPopulation2.clear()
```

10) Main function:

```
def start():
    global totalPopFitness
    global totalOffSpringFitness
    global maxOffFitness
    global averageOffSpringFitness
    global minOffFitness

    # creates a graph for easier behaviour
interpretation of the pop
    plt.ylabel("Fitness")
    plt.xlabel("Generation")
    plt.axis([0, Maxgen, 0, 400])

    # Iterates through each generation
    for i in range(Maxgen):
        offPopCheck = 0
        totalPopFitness = 0
        totalOffSpringFitness = 0
        averageOffSpringFitness = 0
        maxOffFitness = 0
        minOffFitness = 0
        parent_evaluate()
        tournament()
        off_evaluate()
        elitism()
        print("Generation: ", i + 1)
        print("Min: ", minOffFitness, " Max: ",
maxOffFitness)
        print("Total Parent Fitness: ", totalPopFitness)
        print("Total offspring Fitness: ",
totalOffSpringFitness)
        print("Average: ", averageOffSpringFitness)
        y = averageOffSpringFitness
        x = minOffFitness
        line1 = plt.plot(i, y, 'go', label='Average')  #
Average Line
        line2 = plt.plot(i, x, 'r^', label='Best')  #
Best line
        plt.pause(0.01)
        file2write = open("MaxF3.txt", 'a')
        file2write.write(str(minOffFitness) + "\n ")
        file2write.close()
        file2write = open("AverageF3.txt", 'a')
        file2write.write(str(averageOffSpringFitness) +
"\n ")
        file2write.close()
        print("")
    plt.legend()
    plt.show()
```