Christopher Moran

Dr. MacLennan

CS 420

7 April 2020

Project 2: Genetic Algorithms

The purpose of this assignment was to give the students a chance at creating their own genetic algorithm. The general concept behind this can be derived from its namesake. That is to say, when writing genetic algorithms, the goal is to find the optimal model based on the most efficient and optimal models. It takes a sort of shotgun approach, as a large number of initial models are first created almost at random. Then, through a series of tests, the program will be able to determine which model is the most fit, or in other words, it determines which models are the best representation of whatever it is that is meant to be represented. Once the best models have been found, the program will make them reproduce, so to speak. What this means is that a number of new models will be created, all based on a combination of traits held by the two parent models. This process repeats for as long as is dictated by the program, and ideally ends when the optimal model has been produced.

The program created for this project is a relatively simple one, as it is entirely self-contained, and does not read or write, in other words communicate, with any other files, at least none that are required by the rubric. That being said, it does still require a number of input variables to run, and that affect the way the program operates. These variables are the number of genes in the genetic string, represented by l, the population size, represented as n, the mutation probability, represented by pm, the crossover probability, represented by pc, and the number of generations, represented by g.

Methodology

The program that was submitted runs a number of genetic algorithm tests based on a number of initial variables. For each set of initial variables, three tests were run. The initial variables were set up in the arrays shown below:

```
l = [20, 30, 10, 50, 50, 50]

n = [30, 20, 40, 50, 50, 50]

pm = [0.033, 0.1, 0.01, 0.5, 0.033, 0.02]

pc = [0.6, 0.3, 0.9, 0.5, 0.6, 0.4]

g = [10, 25, 5, 50, 50, 50]
```

Each epoch consists of the three tests run on each set of initial variables above.

That being said, each test begins with a randomly initialized population. This population is represented as a 2d array, with each sub array representing an individual. These individuals are identified with their genes, or bits, that are constructed within the program as an array of characters, randomly initialized with a 50/50 chance of each character being either 1 or 0. The population array is set to be n units long, which represents the number of individuals being tested.

Once the test has initialized its population, it begins to calculate the fitness of each individual. The first thing that must happen here is to convert the character array of 1s and 0s, which themselves represent a number written in binary, into its decimal form. Then, once that is complete, the individual's fitness can be calculated with this fitness function:

$$F(s)=(x/2^{l})^{10}$$

The individual's fitness is also used to calculate the total running sum of all individual's fitness, as it is just added to the total sum's counter as its fitness is calculated.

Then, using that running sum, each individual's normalized fitness can be found. The normalized fitness can be calculated as the individual's sum divided by the total running sum.

Then, the array of normalized fitness for each individual can be used to find a running normalized total. This means that an array is kept that records the normalized value of each individual added to all normalized values before that.

Once this is all done, the program must enter a loop that lasts for N/2 iterations. This is where the magic happens, and reproduction can commence. For each iteration, two offspring must be created, so that a total of N offspring are created after the end of the loop. However, before the offspring can be created, the parents must be chosen. In order to do this, the program generates two random numbers. Using the first random number, the program checks to see where that number would lie if overlaid with the total sum of all normalized values.

An example can be used to clarify this. Say that the running normalized sum contains the values …0.4, 0.5, and 0..6….. Then, assume that the first random number created was 0.55. In this example, 0.6 would be chosen as the first parent. This is the case because 0.55 lies between 0.5 and 0.6, and so the greater of these two will always be chosen. The second parent is slightly more complicated to find because it cannot be a duplicate of the first parent. This means that if the same individual is chosen to be the second parent as the first one, a new second random number must be generated, or perhaps even more than that, until a second, non-duplicate parent is found.

Then, once the parents have been found, they must reproduce. The first thing that happens here is a check that determines whether or not the offspring will represent a combination of their parents, or rather that offspring 1 will be a direct copy of parent 1 and so on. This is done with a comparison of the crossover probability and a random number, which means that a certain

percentage of offspring, this percentage being represented by the variable pc, will be made up with a combination of their parents traits, while the other offspring will represent their parents in their entirety, meaning that offspring 1 = parent 1 and that offspring 2 = parent 2.

The second step of creating the offspring is allow for them to mutate. This is done bit by bit, or in the case of this program, character by character. Each character has a certain percentage chance of mutating, given by the variable pm. This means that for each character, if it just so happens that it is chosen for mutation, it will change its value from 0 to 1, or 1 to 0. It becomes its opposite.

Once N offspring have been created, a generation has been completed. Each test runs until it has completed each of its generations. For each generation, the studied and modified population is given by the offspring population found in the generation before, except for the first generation, which was initialized randomly.

For each population/generation studied, a number of statistics must be accumulated. For each population, its average fitness, the fitness of its most fit individual, and the number of correct bits in that fit individual, must be recorded.

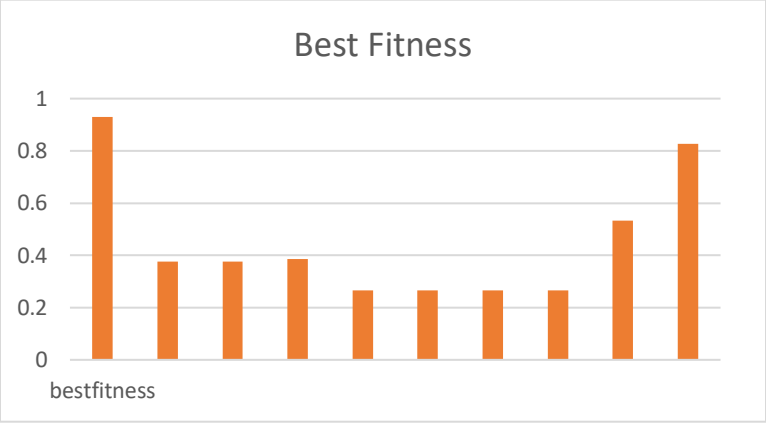
Then, for each test, arrays of these statistics must be recorded and then printed in a csv. For each set of arrays printed, the variables used, as well as the epoch and test number, are recorded and printed as well.

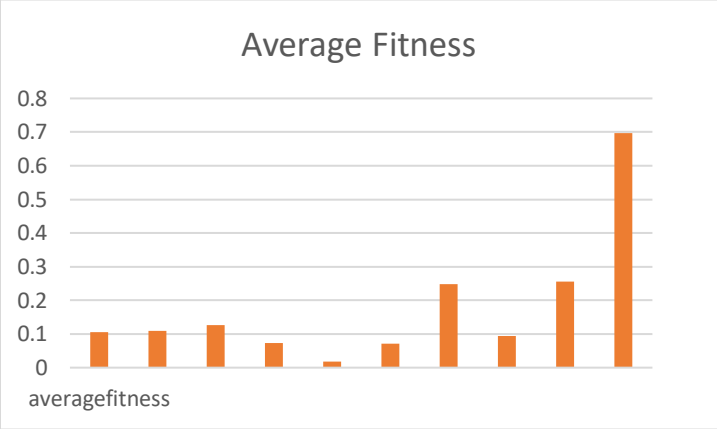
Results

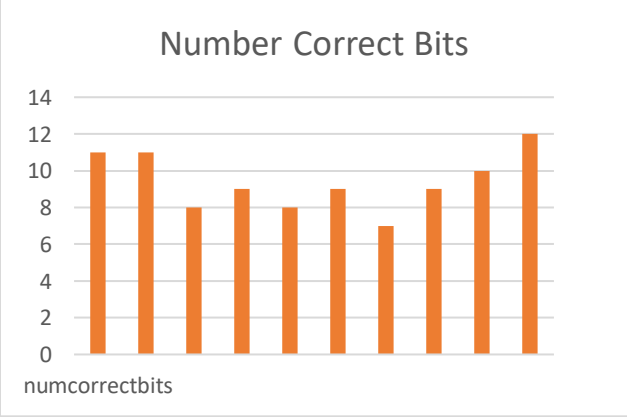
Epoch 1

Test 1

epoch number	test number	l	N	G	Pm	Pc
1	1	20	30	10	0.033	0.6

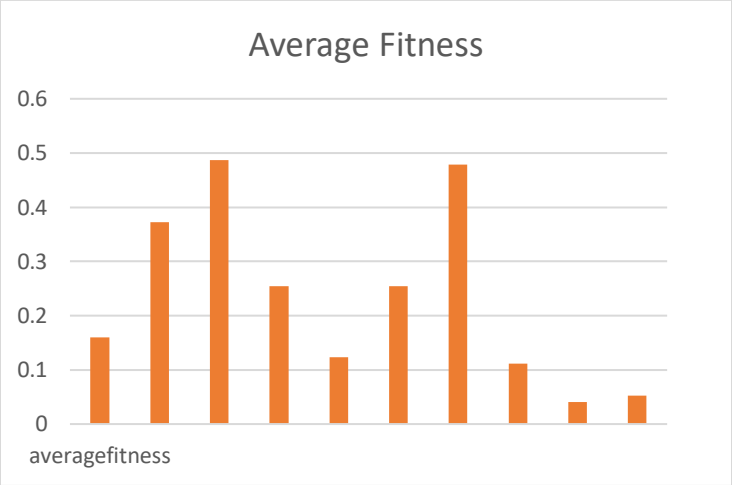


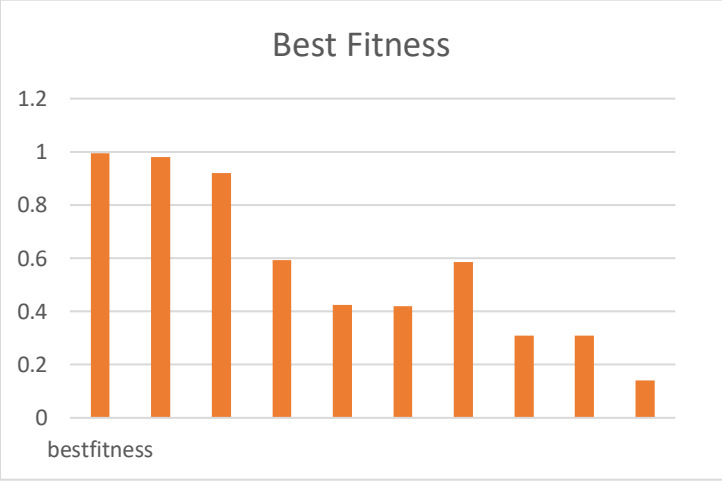


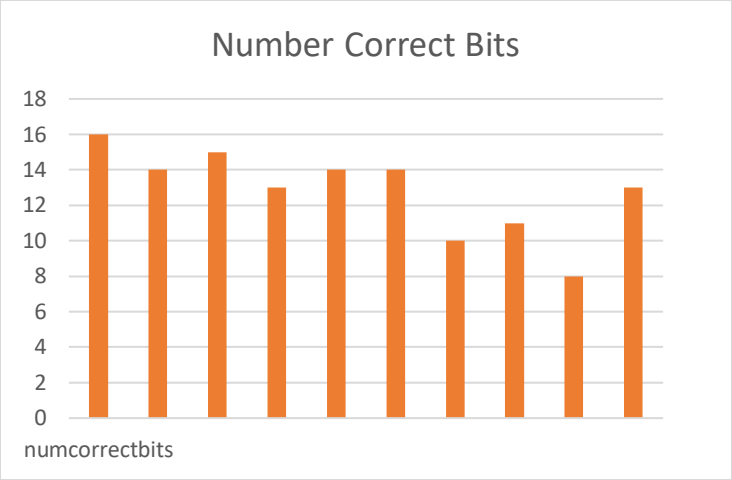


Test 2

epoch number	test number	l	N	G	Pm	Pc
1	2	20	30	10	0.033	0.6

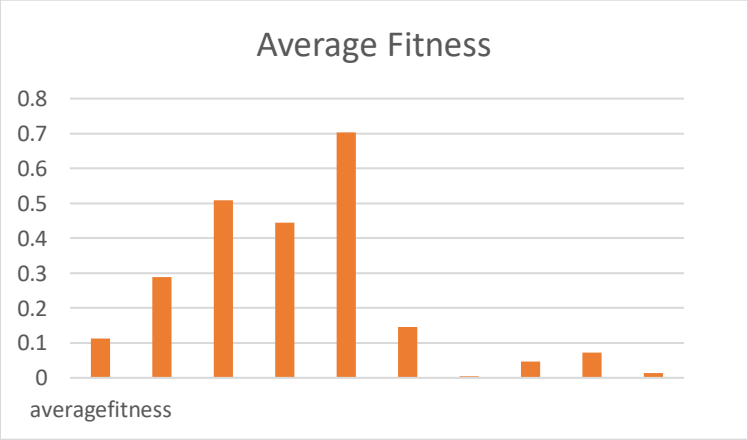


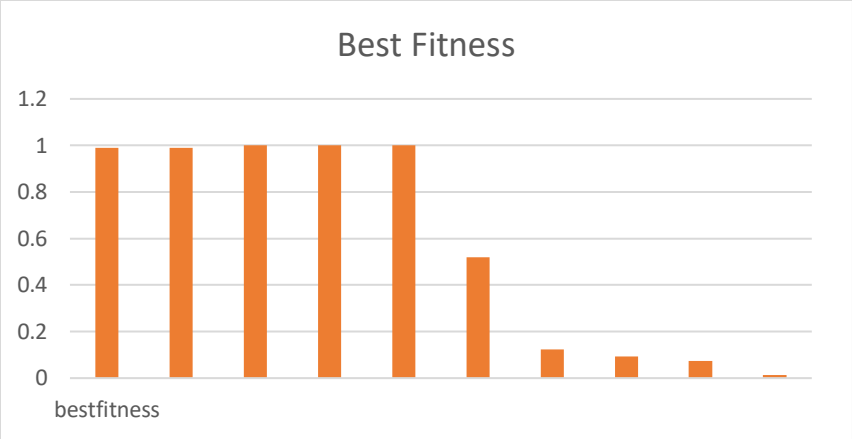


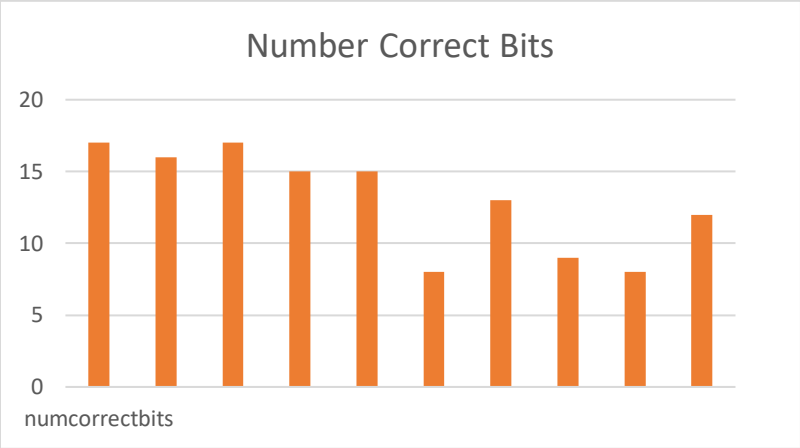


Test 3

epoch number	test number	l	N	G	Pm	Pc
1	3	20	30	10	0.033	0.6



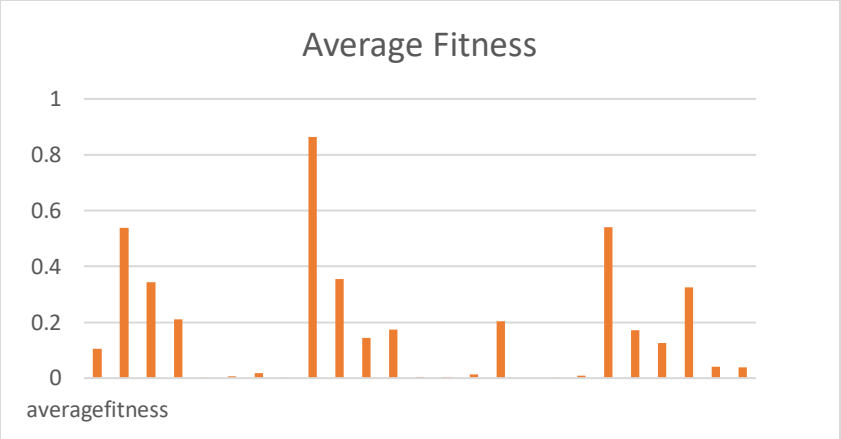


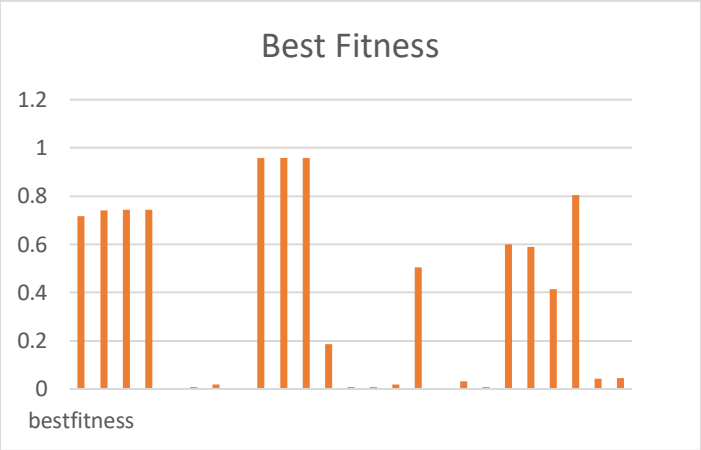


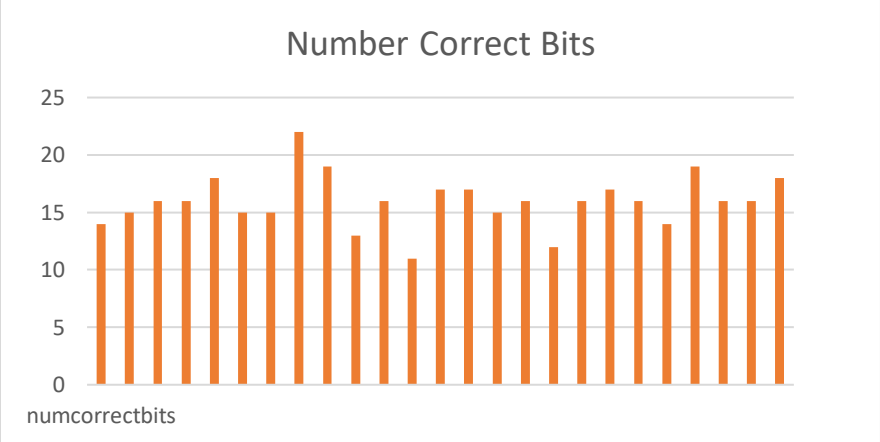
Epoch 2

Test 1

epoch number	test number	l	N	G	Pm	Pc
2	1	30	20	25	0.1	0.3

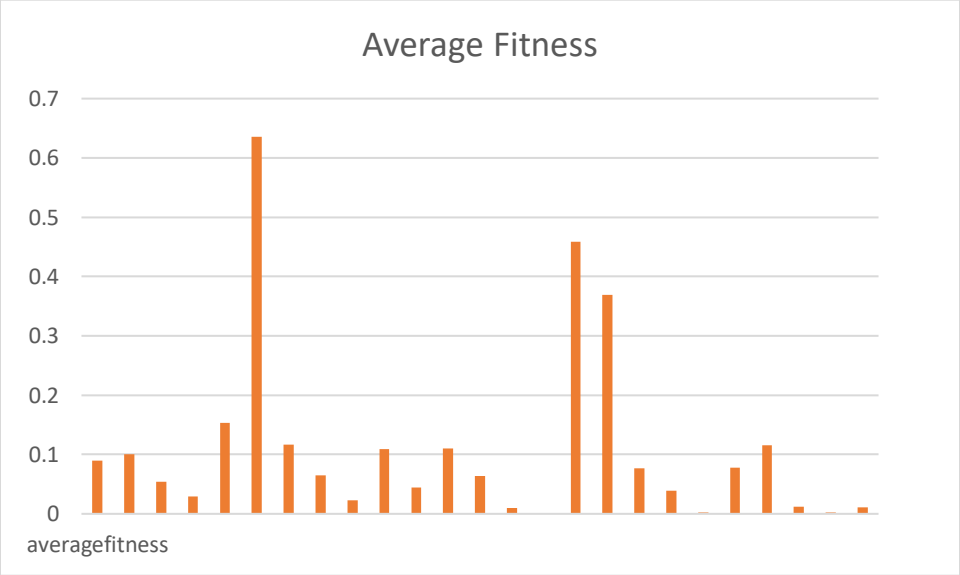

Average Fitness
1
0.8
0.6
0.4
0.2
0
averagefitness

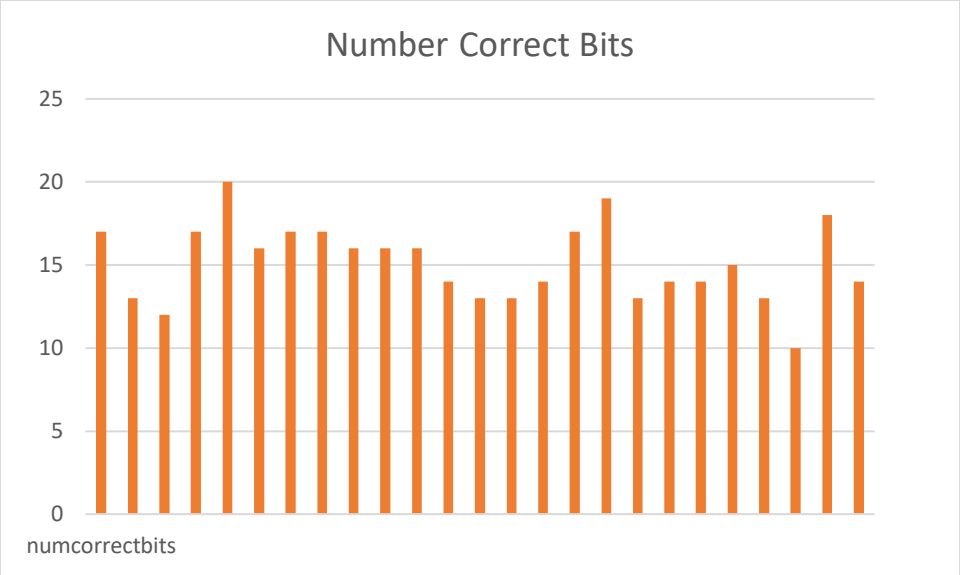

Best Fitness
1.2
1
0.8
0.6
0.4
0.2
0
bestfitness


Number Correct Bits
25
20
15
10
5
0
numcorrectbits

Test 2

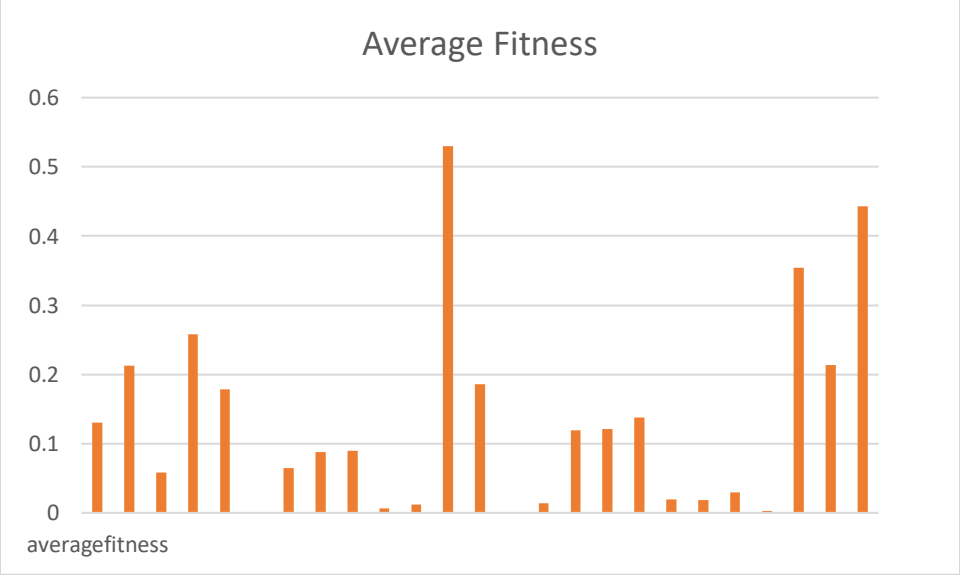
epoch number	test number	l	N	G	Pm	Pc
2	2	30	20	25	0.1	0.3

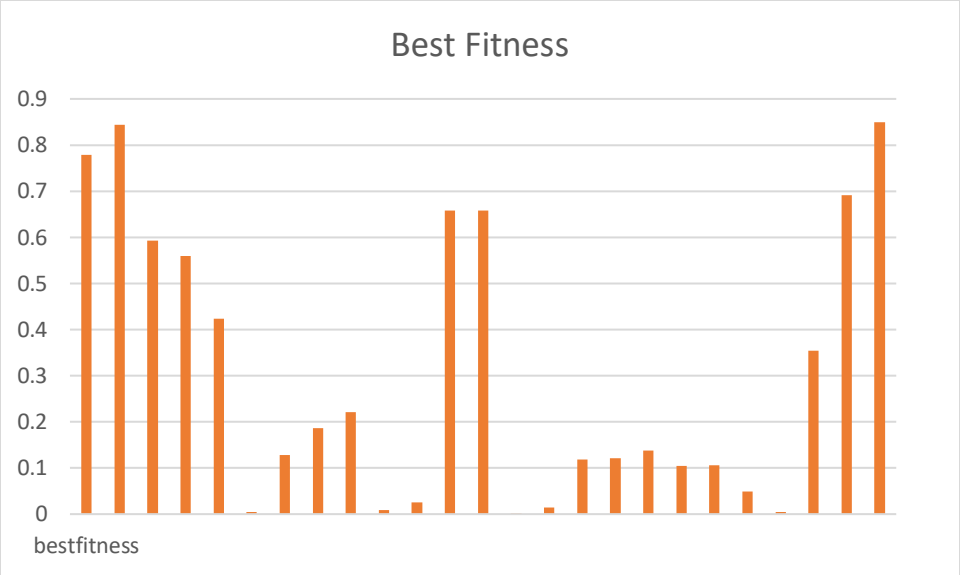


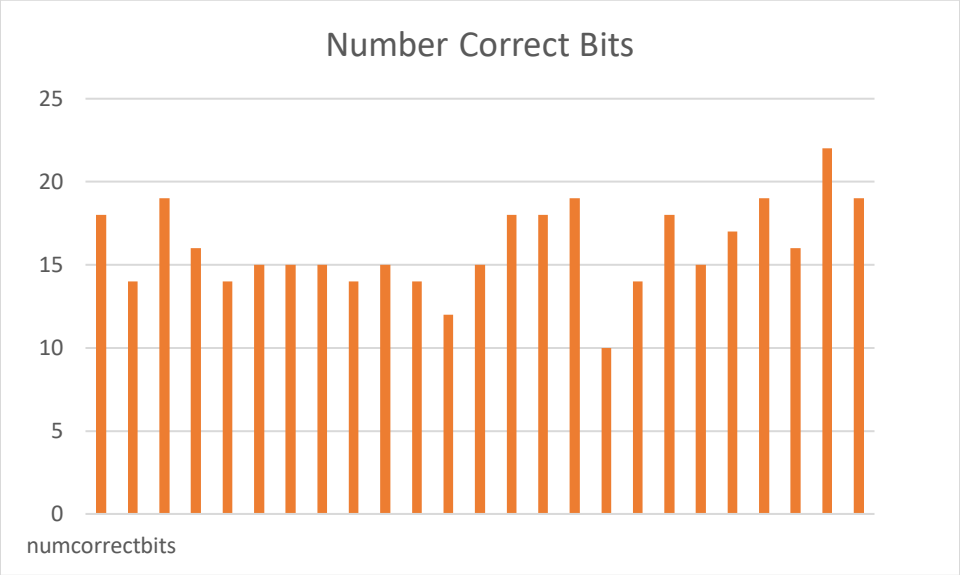


Test 3

epoch number	test number	l	N	G	Pm	Pc
2	3	30	20	25	0.1	0.3



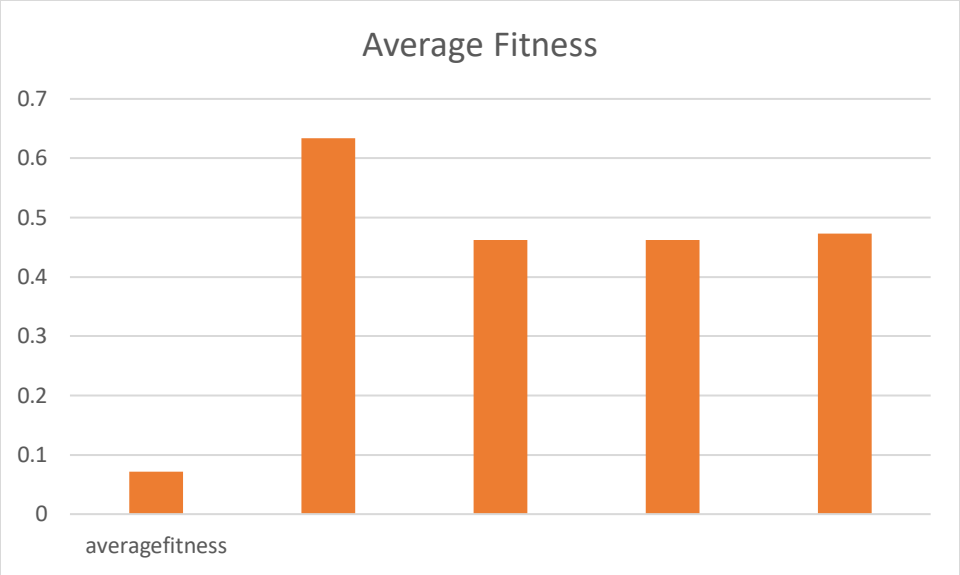


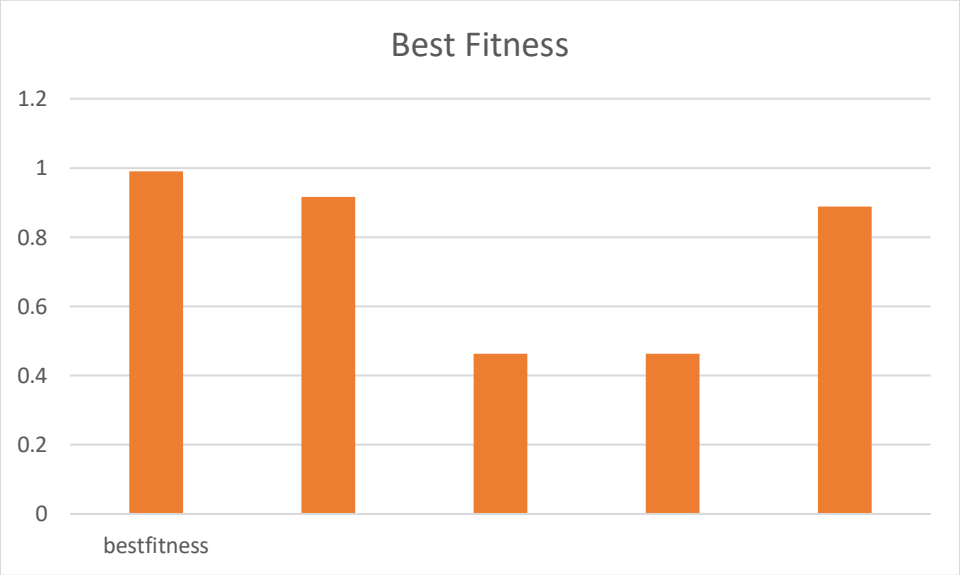


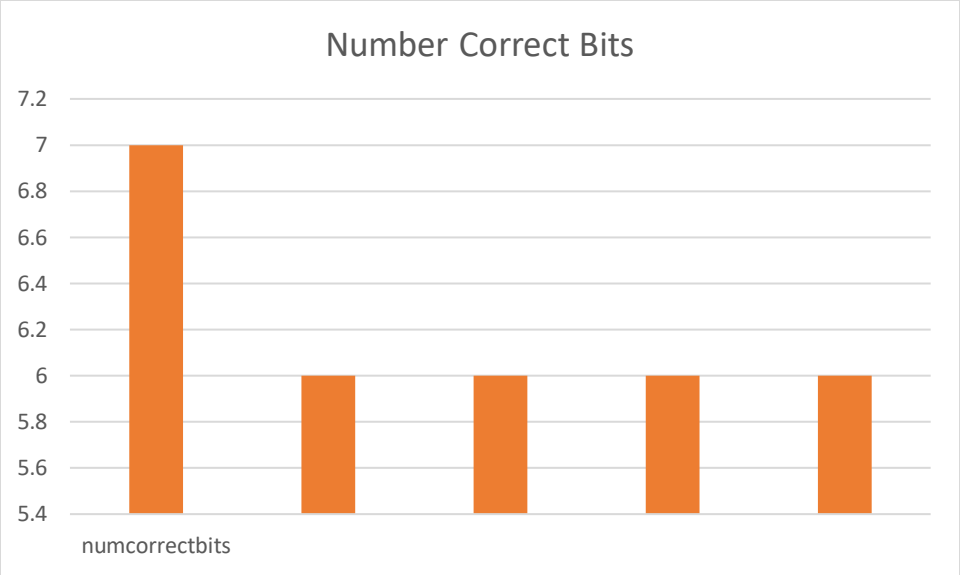
Epoch 3

Test 1

epoch number	test number	l	N	G	Pm	Pc
3	1	10	40	5	0.01	0.9

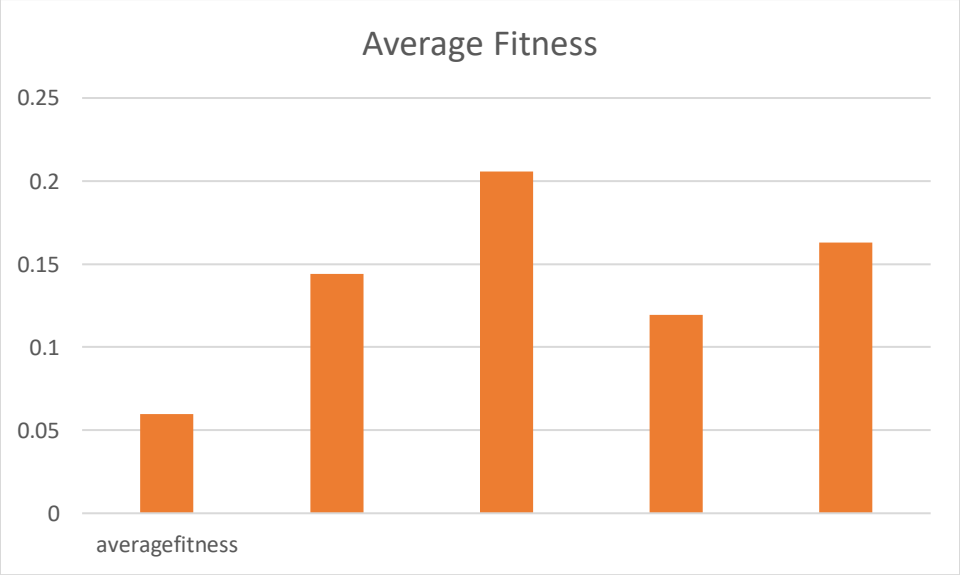

Average Fitness
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
averagefitness

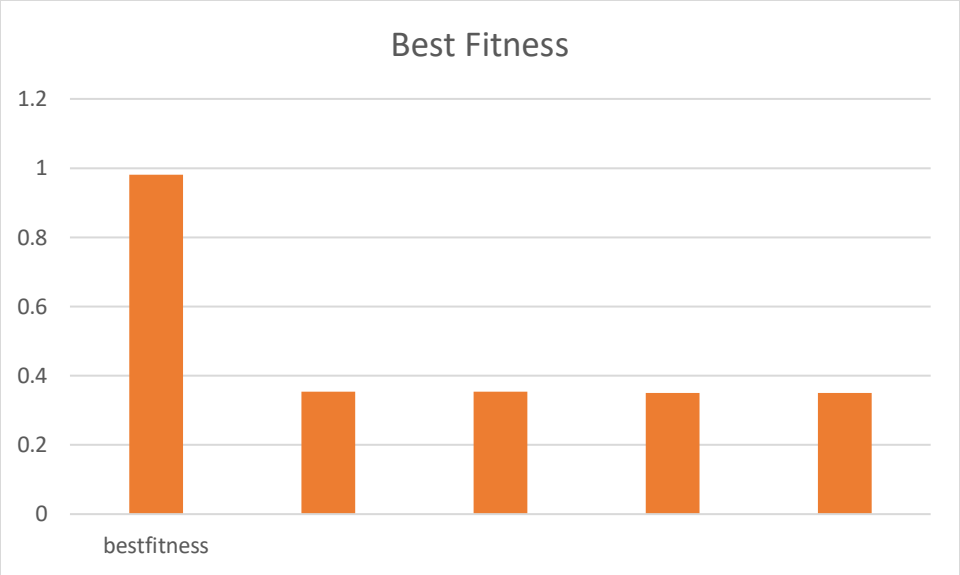

Best Fitness
1.2
1
0.8
0.6
0.4
0.2
0
bestfitness

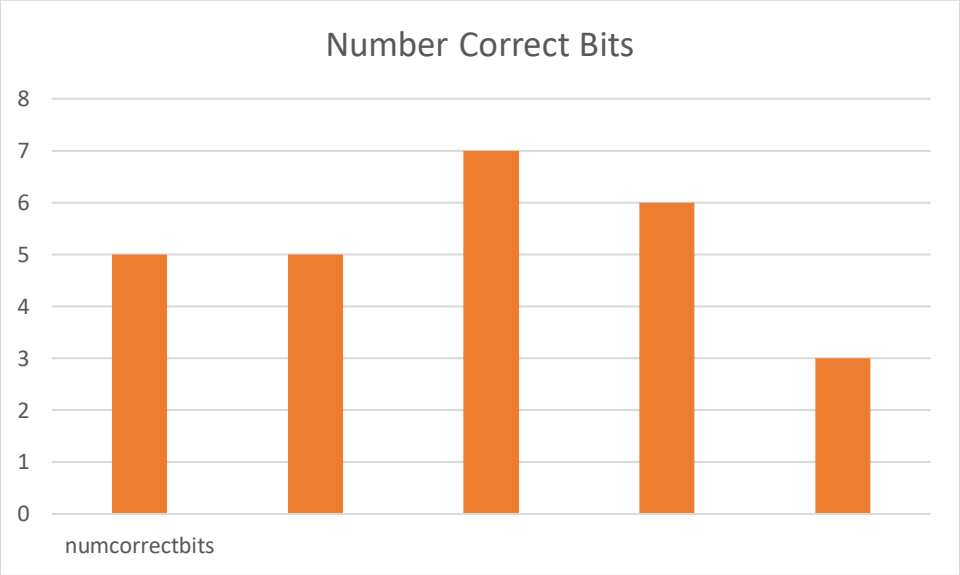


Test 2

epoch number	test number	l	N	G	Pm	Pc
3	2	10	40	5	0.01	0.9

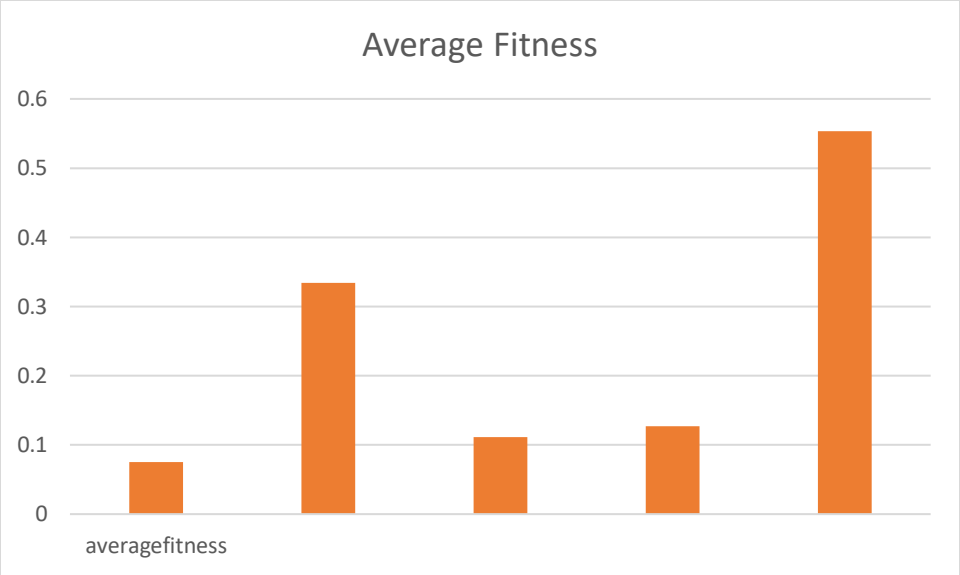


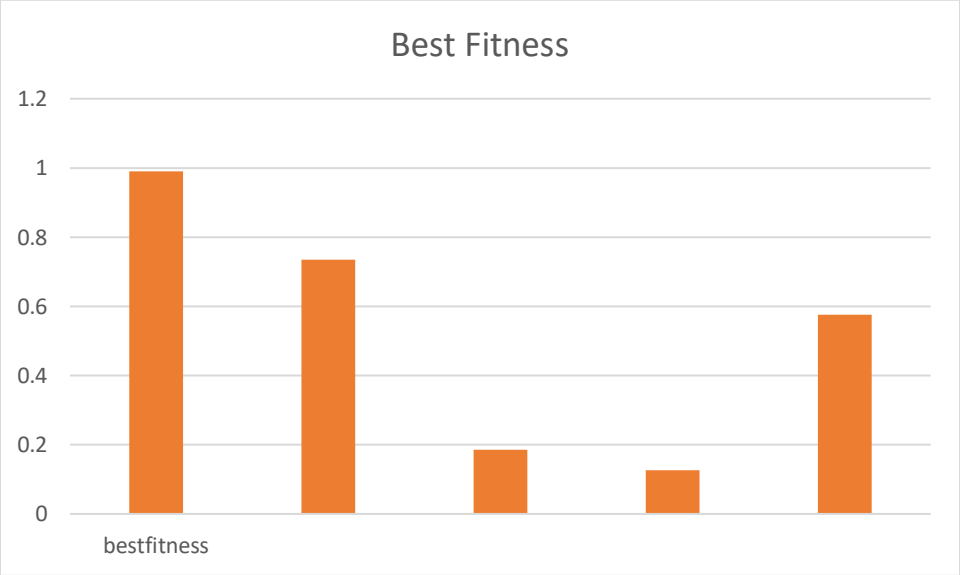


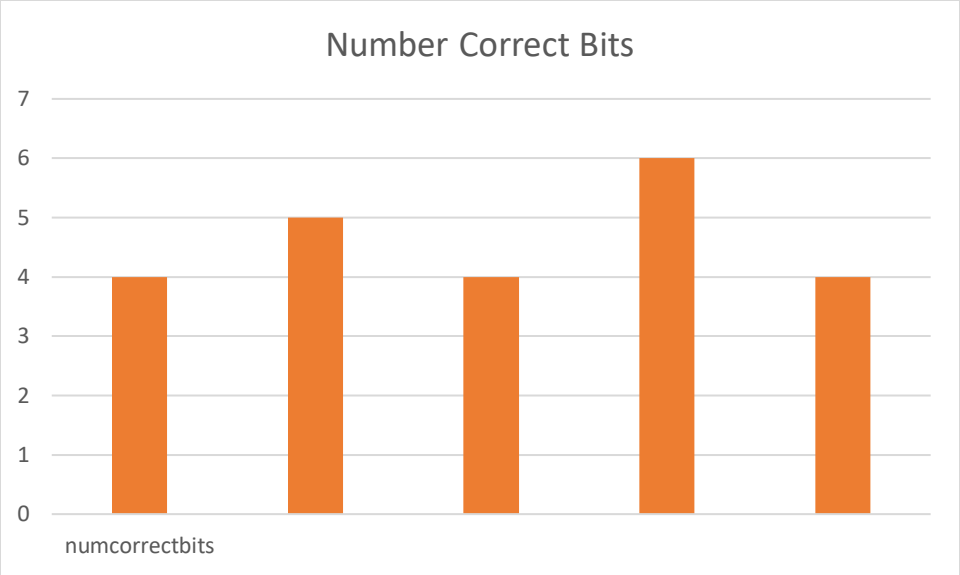


Test 3

epoch number	test number	l	N	G	Pm	Pc
3	3	10	40	5	0.01	0.9


Average Fitness
0.6
0.5
0.4
0.3
0.2
0.1
0
averagefitness

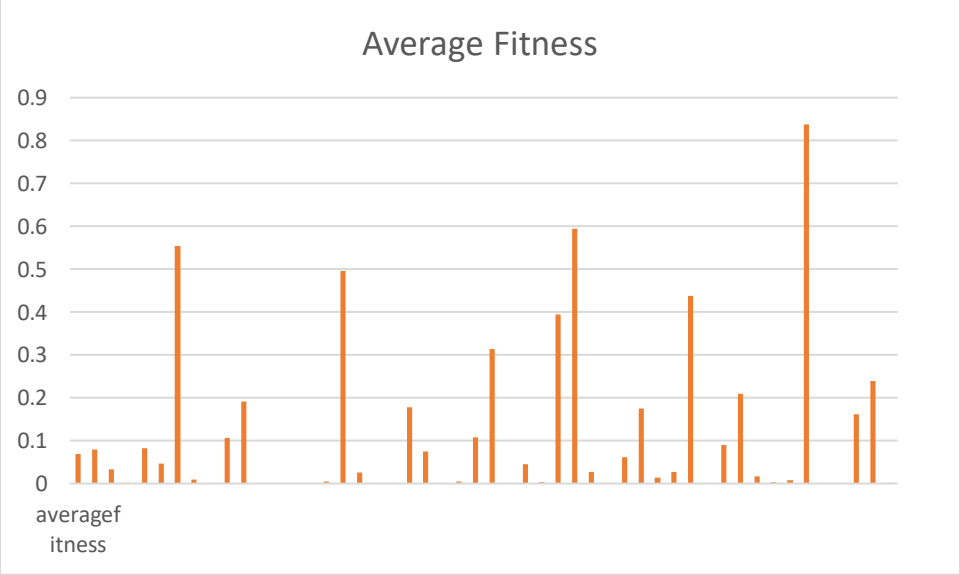

Best Fitness
1.2
1
0.8
0.6
0.4
0.2
0
bestfitness

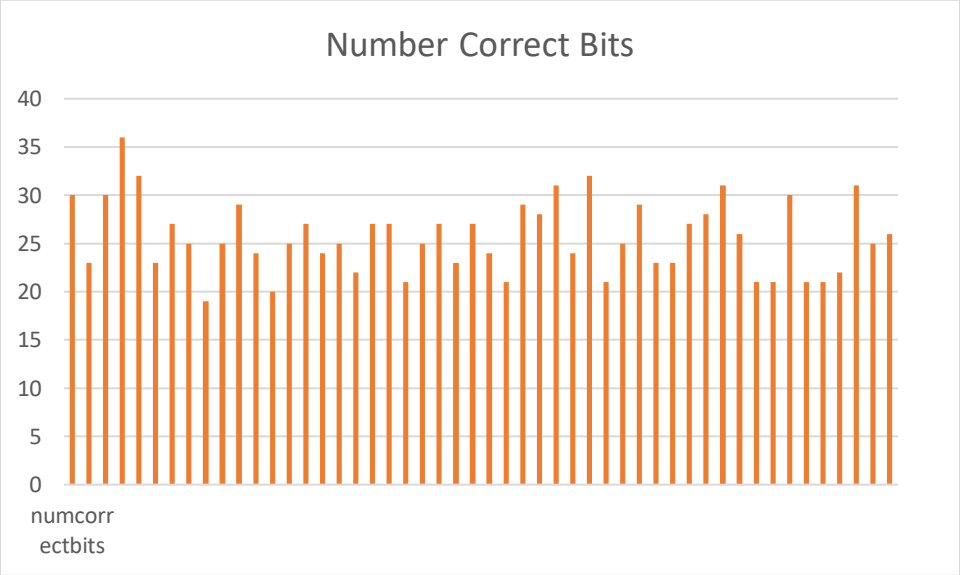


Epoch 4

Test 1

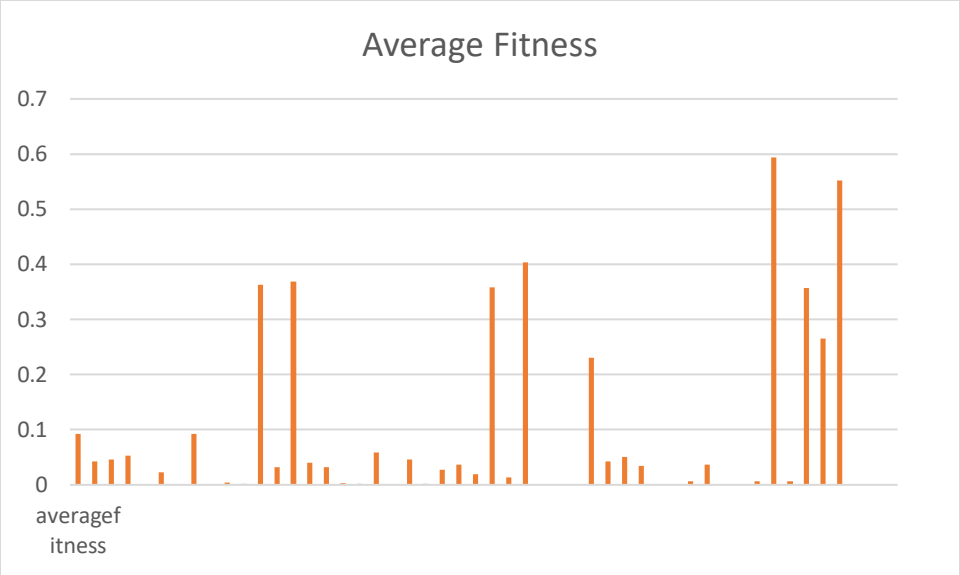
epoch number	test number	l	N	G	Pm	Pc
4	1	50	50	50	0.5	0.5



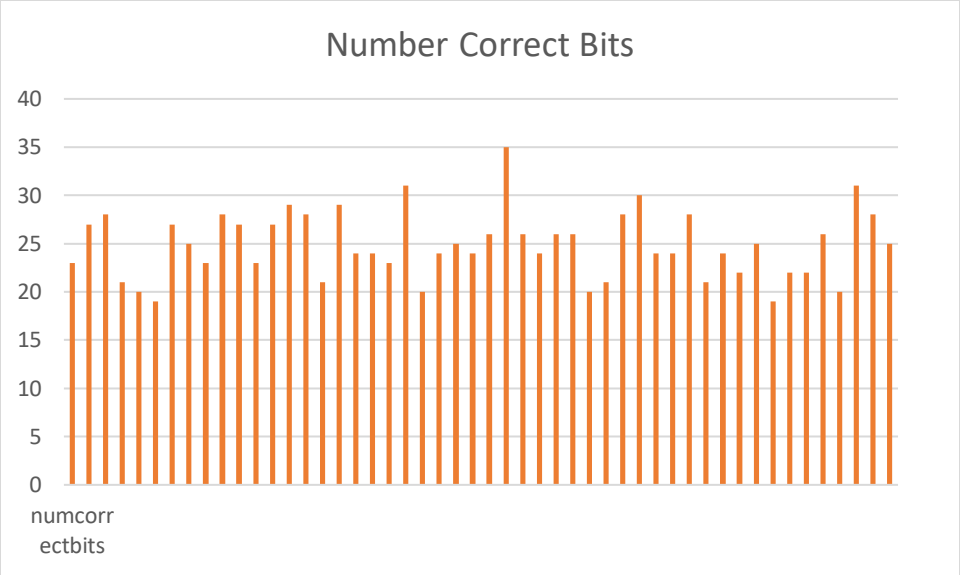


Test 2

epoch number	test number	l	N	G	Pm	Pc
4	2	50	50	50	0.5	0.5

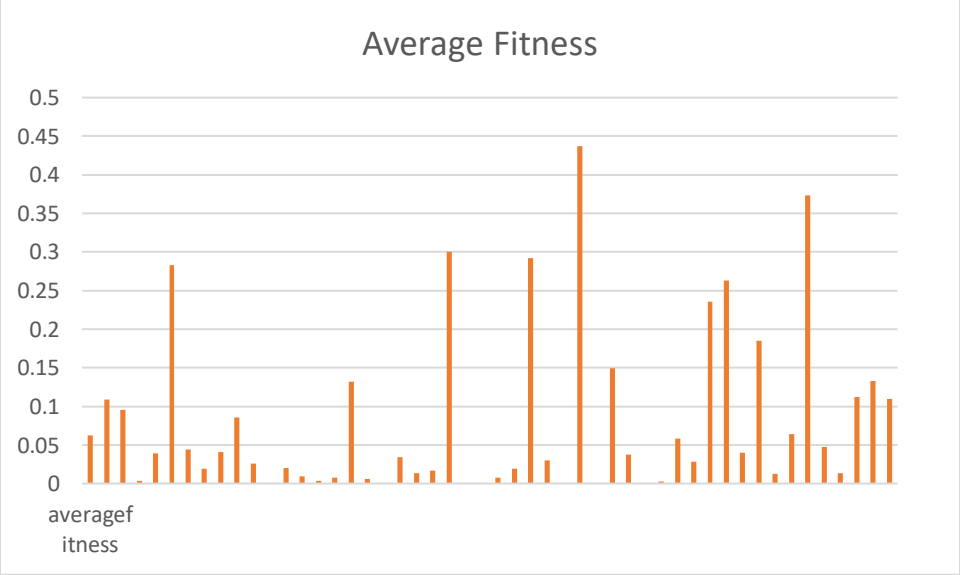

Average Fitness
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
averagef
itness

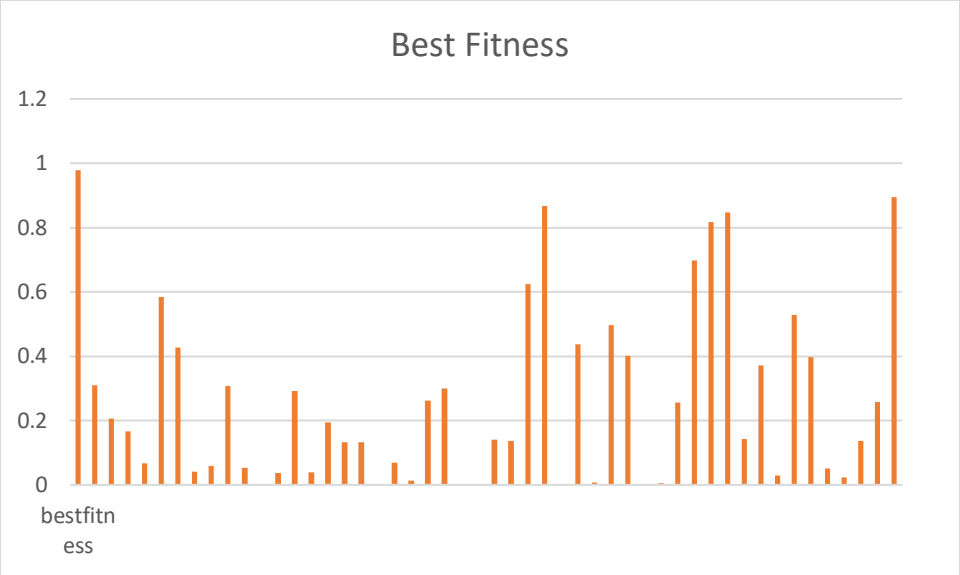

Best Fitness
1.2
1
0.8
0.6
0.4
0.2
0
bestfitn
ess

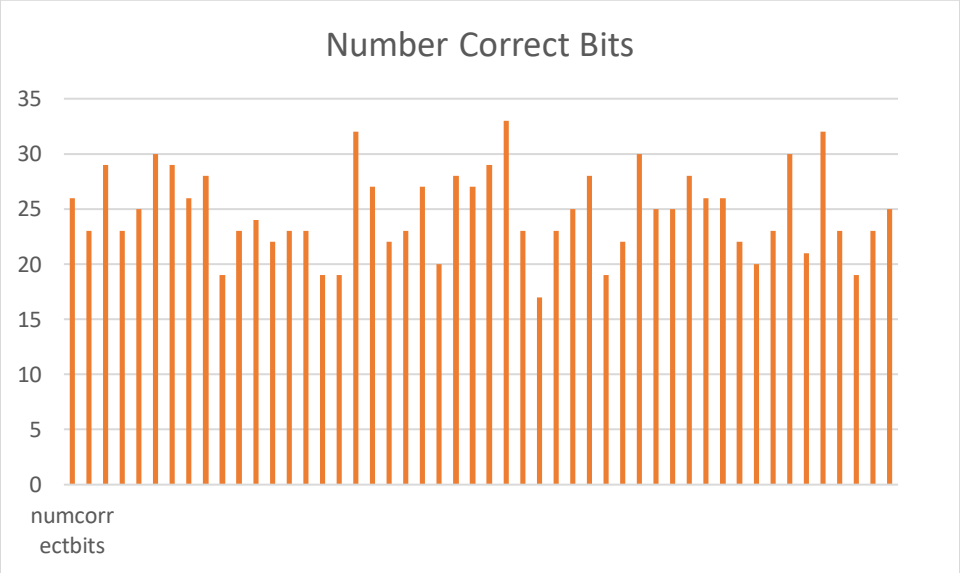


Test 3

epoch number	test number	l	N	G	Pm	Pc
4	3	50	50	50	0.5	0.5



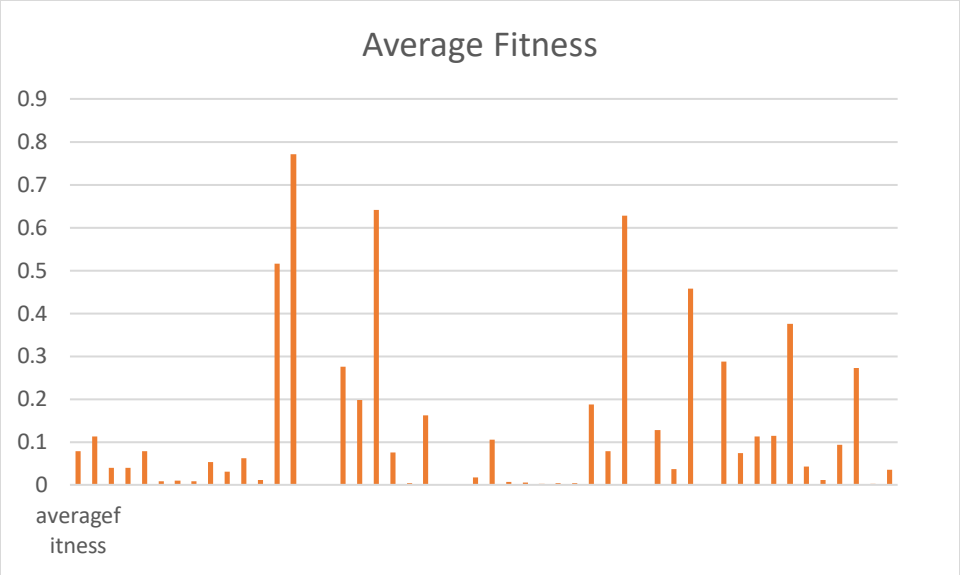


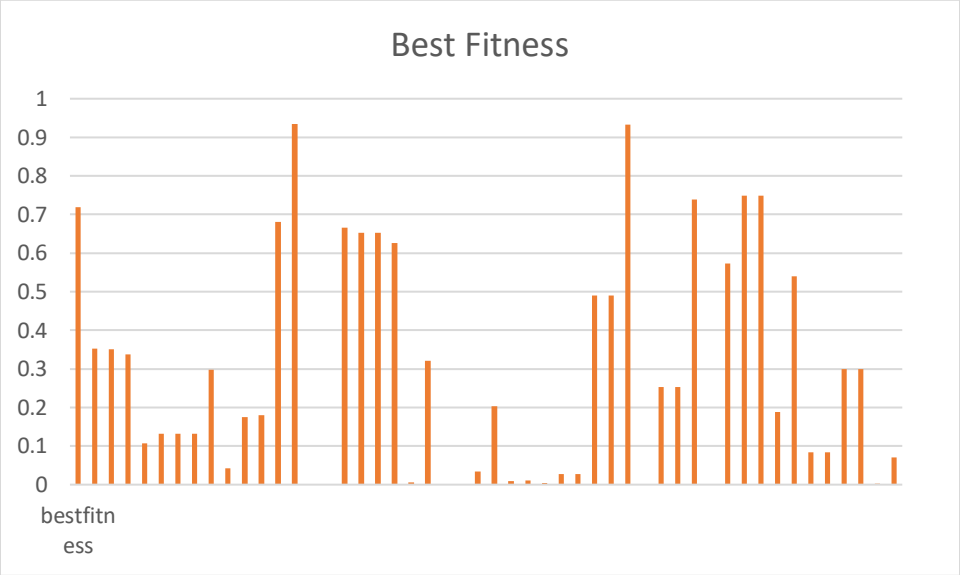


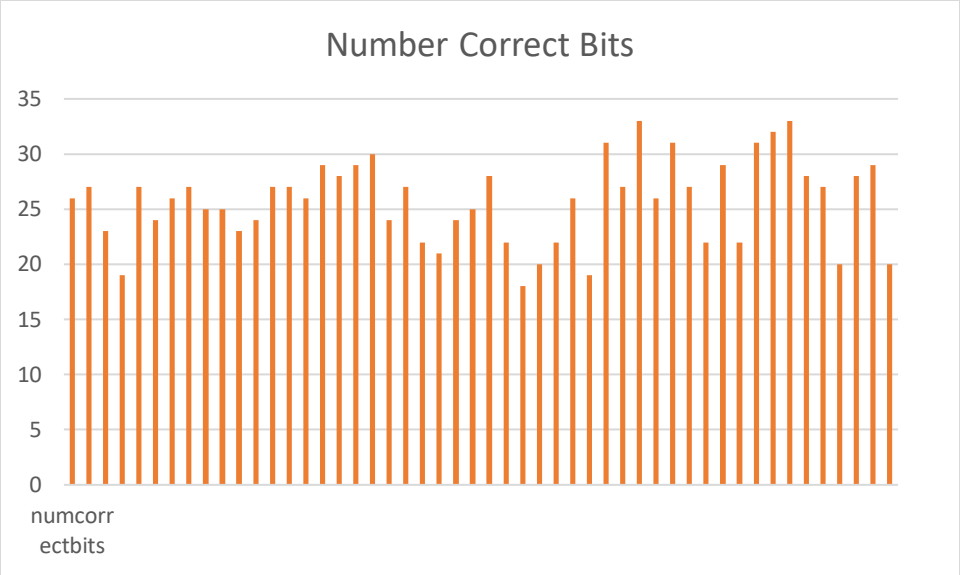
Epoch 5

Test 1

epoch number	test number	l	N	G	Pm	Pc
5	1	50	50	50	0.033	0.6

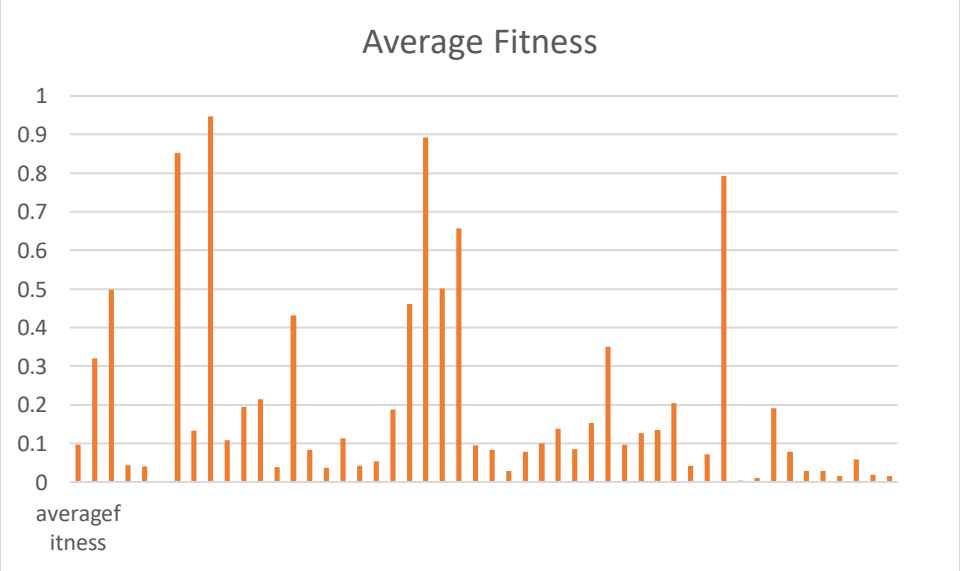

Average Fitness
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
averagef
itness

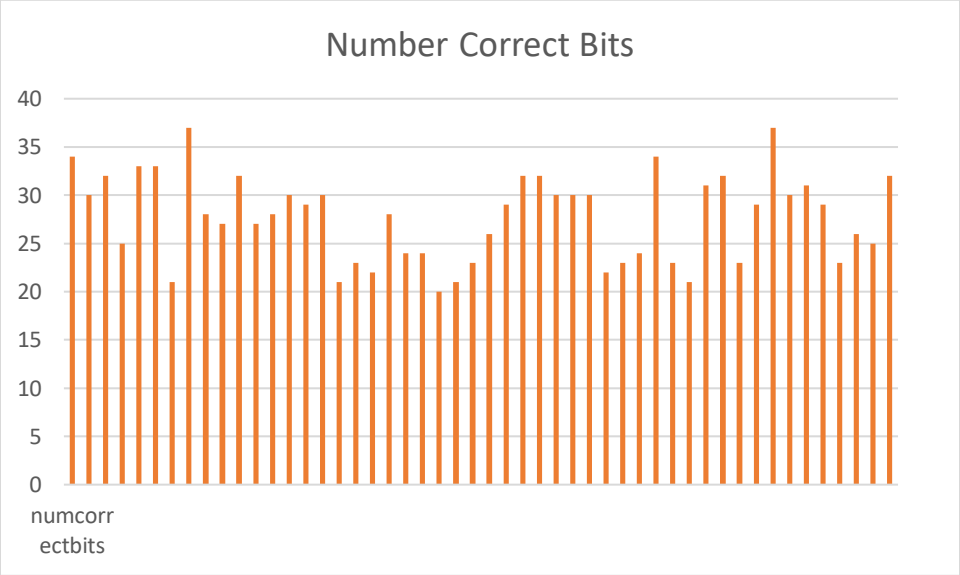

Best Fitness
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
bestfitn
ess



Test 2

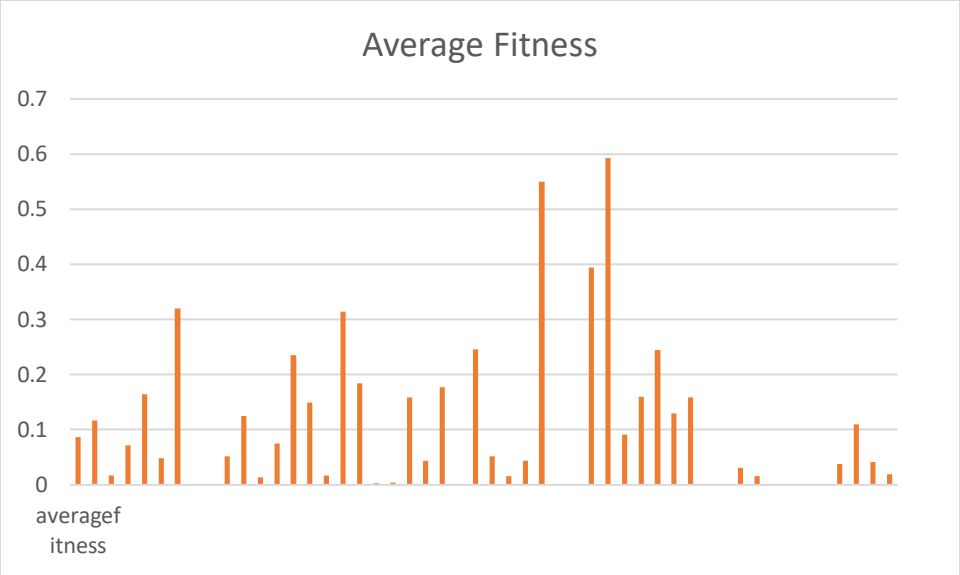
epoch number	test number	l	N	G	Pm	Pc
5	2	50	50	50	0.033	0.6

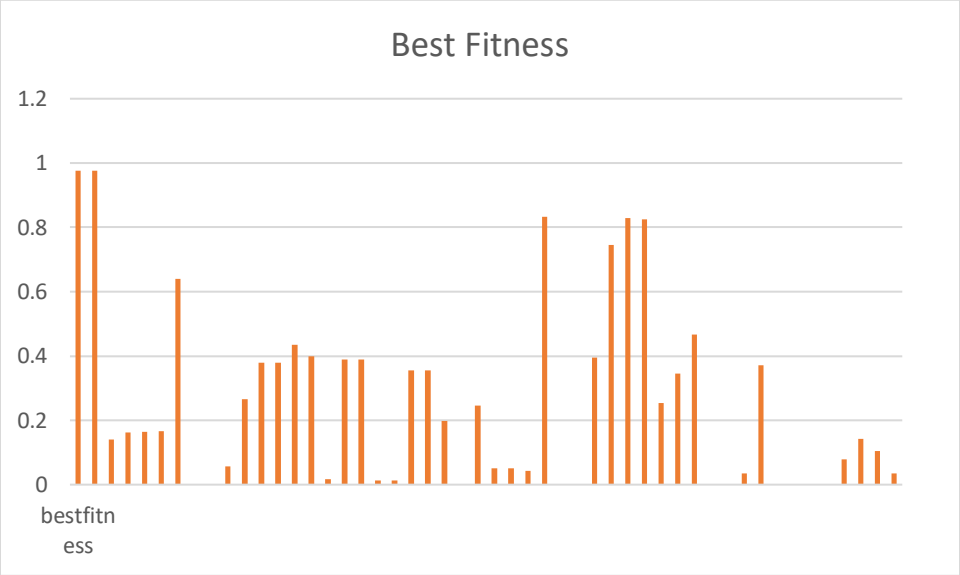


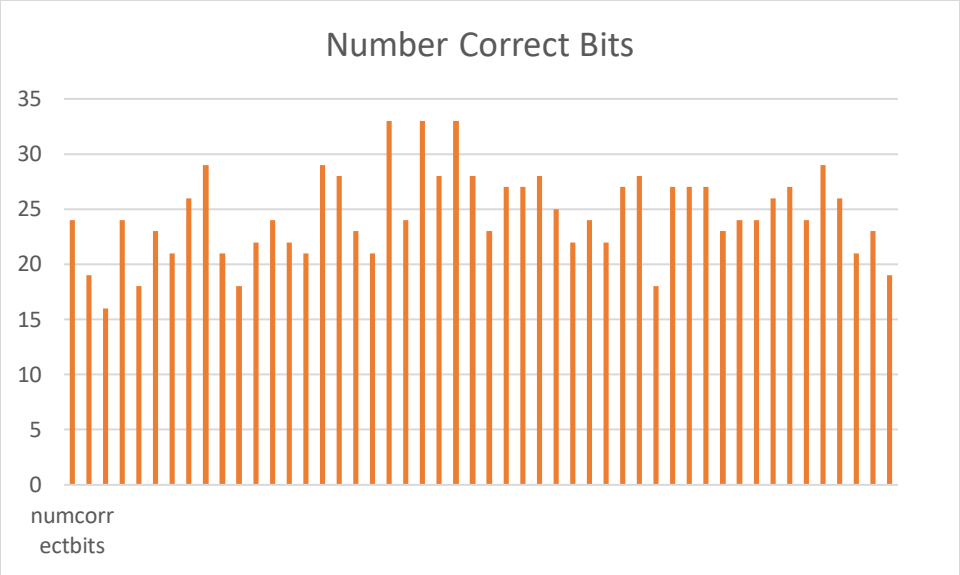


Test 3

epoch number	test number	l	N	G	Pm	Pc
5	3	50	50	50	0.033	0.6


Average Fitness
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
averagef
itness

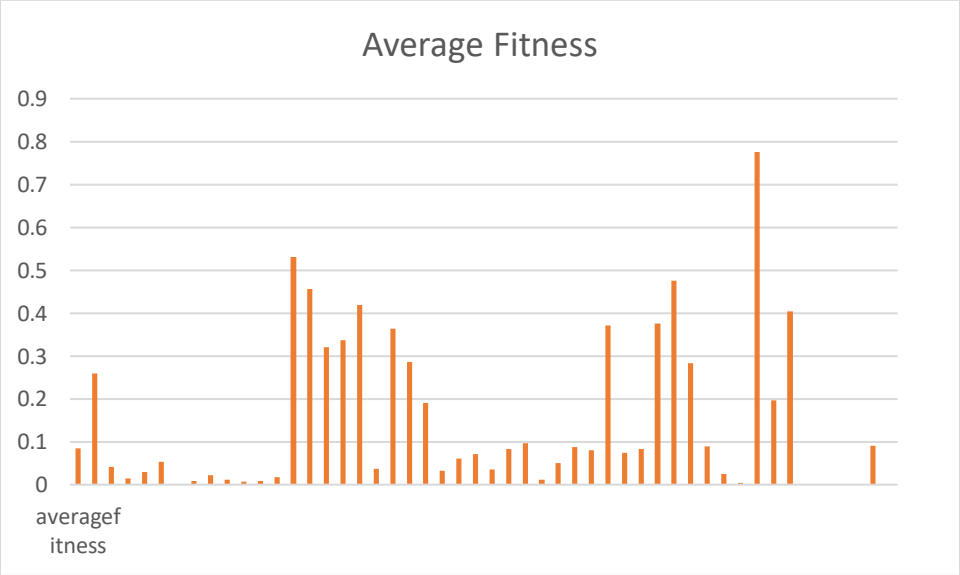

Best Fitness
1.2
1
0.8
0.6
0.4
0.2
0
bestfitn
ess

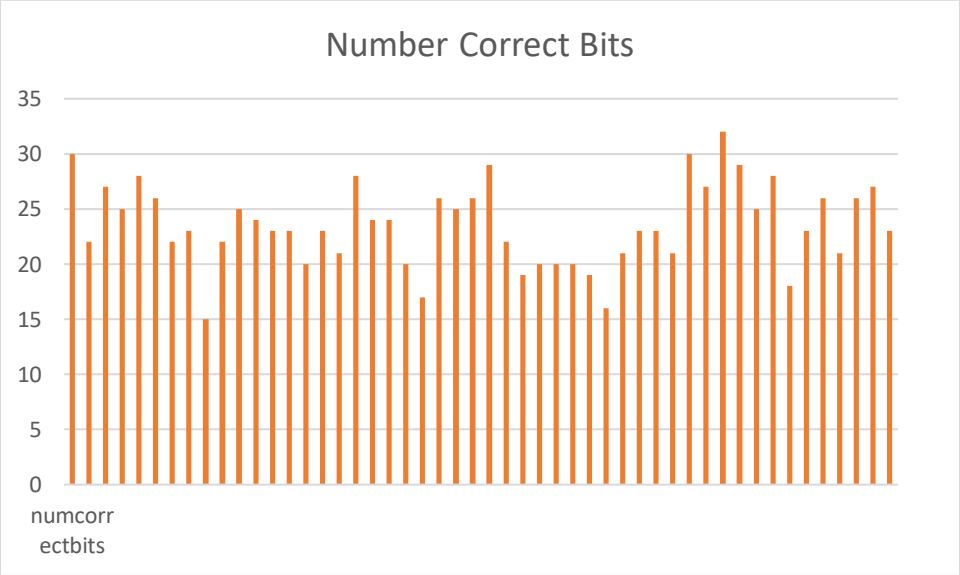


Epoch 6

Test 1

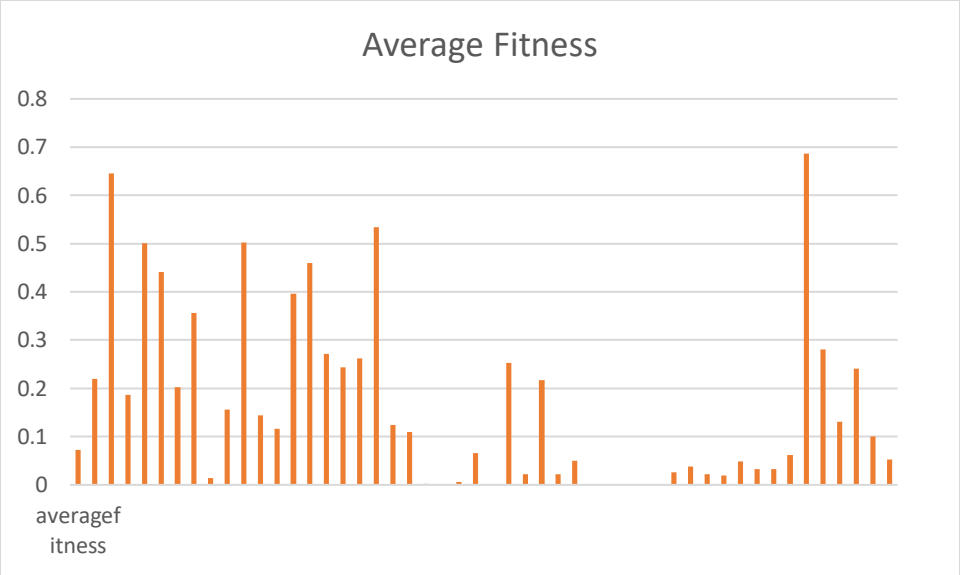
epoch number	test number	l	N	G	Pm	Pc
6	1	50	50	50	0.02	0.4

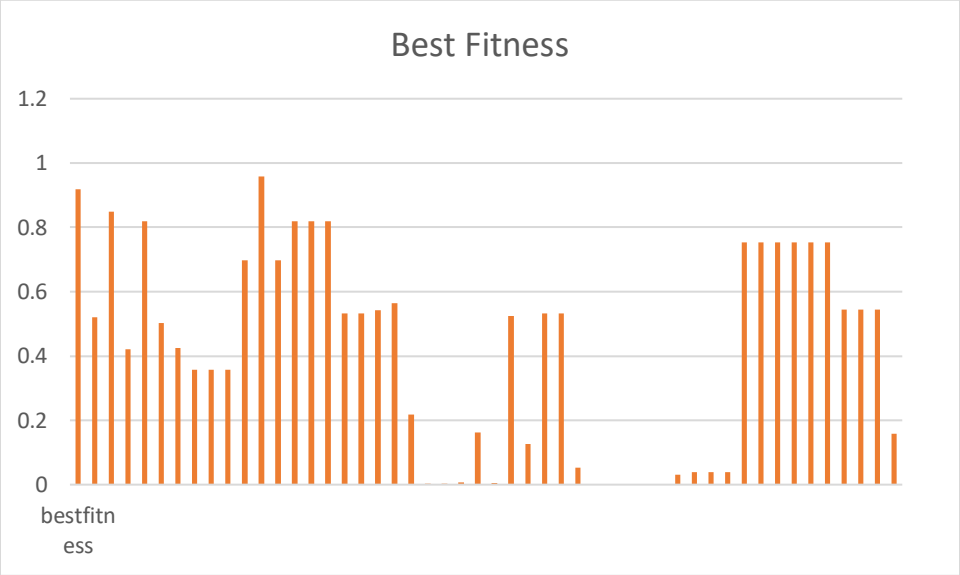


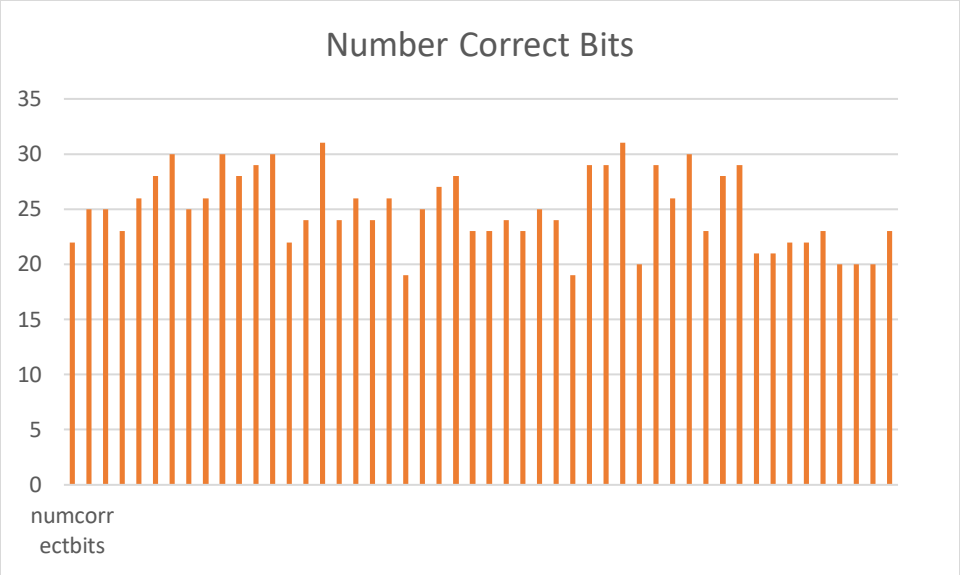


Test 2

epoch number	test number	l	N	G	Pm	Pc
6	2	50	50	50	0.02	0.4

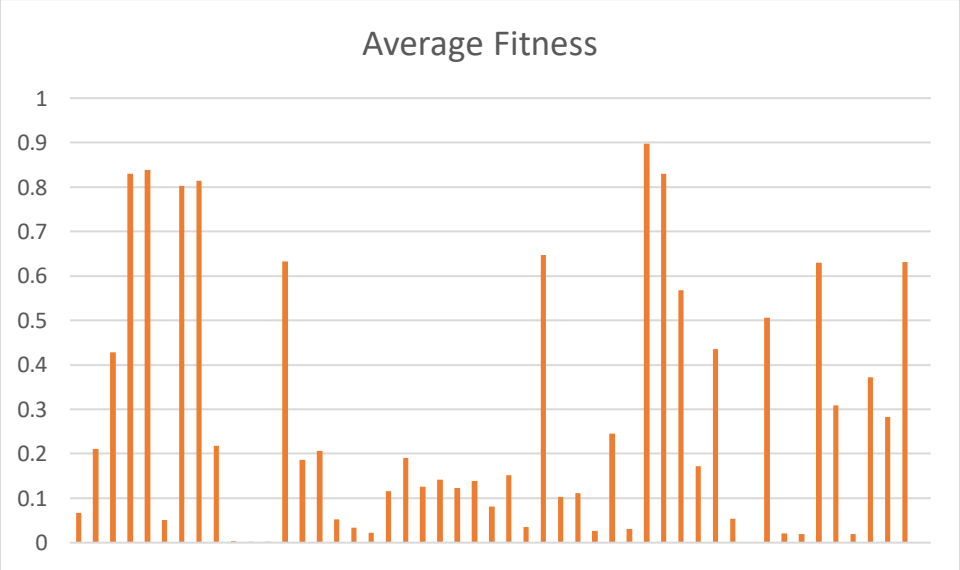

Average Fitness
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
averagef
itness

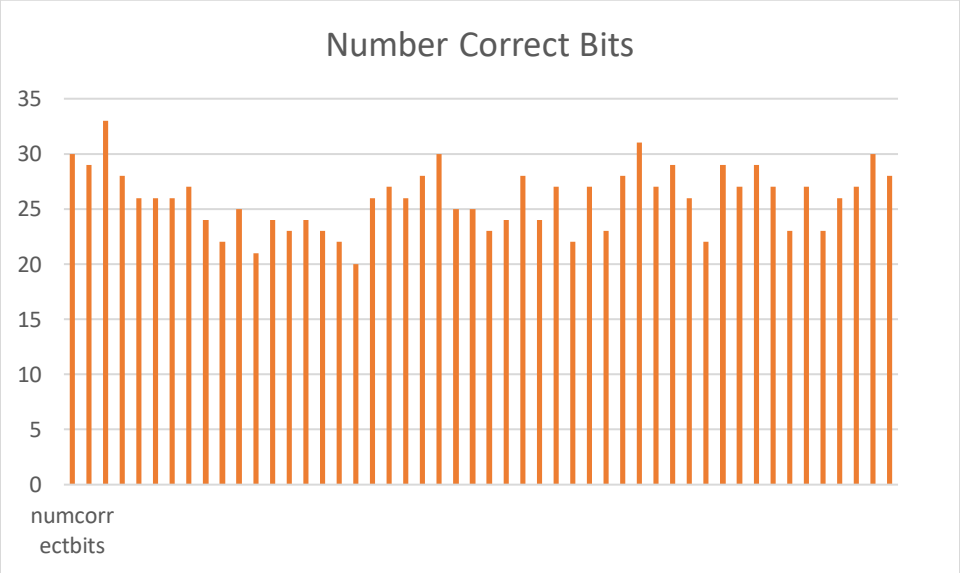

Best Fitness
1.2
1
0.8
0.6
0.4
0.2
0
bestfitn
ess



Test 3

epoch number	test number	l	N	G	Pm	Pc
6	3	50	50	50	0.02	0.4





Analysis

The data presented above correlates to the arrays shown in the methodology:

```
l = [20, 30, 10, 50, 50, 50]

n = [30, 20, 40, 50, 50, 50]

pm = [0.033, 0.1, 0.01, 0.5, 0.033, 0.02]
```

```
pc = [0.6, 0.3, 0.9, 0.5, 0.6, 0.4]

g = [10, 25, 5, 50, 50, 50]
```

An attempt was made in this project to increase the number of generations to far above the recommended amount of 10 to 50, which is the amount chosen for the last 3 epochs, and therefore 9 tests. Unfortunately, there were some unexpected results that occurred there that actually make sense upon further inspection. In those examples, it can be observed that the best fitness array routinely increases until approaching 1, at which point it crashes back down to almost 0. This is likely due to the way that the parents are chosen. Because there can be no duplicate parents chosen, a difficult situation is presented whenever one of the individuals has a very high fitness score. When this happens, it in effect takes value away from the other individuals, with whom the most fit must still mate with. This means that on occasion, the most fit has no choice but to mate with those less fit than it, creating the wonky, varying genetic code shown above.

Also, the number of correct bits would always hover around a value equivalent to half of the value of l, no matter the other parameters.

Pc and Pm affect how quickly the best fitness candidate cycles between high and low.

Increasing l seemed to have minimal affect at a large scale, so it can be assumed that it only increased the accuracy and grain of the other functions.

Increasing N seemed to have a similar affect to increasing l, as it simply introduced more grain.

Conclusion

This project was actually one of the main reasons I even signed up for this class, and for that reason I am happy that I was finally able to do it. I had seen youtube videos of people using

genetic algorithms to create wonderful things, and so I thought it would be neat to learn how to create one myself. This project almost seemed like a flawless execution of that action, and I feel as if I actually have a pretty good grasp on the subject now. Everything worked swimmingly, and so it now seems that I can implement a genetic algorithm without much fuss.