Calvin Zikakis

October 16th, 2019

CSCI 3202

Assignment 2 Report

Report: Part 1

1. The average amount of generations to achieve the generation of all ones was 21 generations. The lowest time to achieve all ones was 2 generations. The highest time to achieve the all ones was 50 generations.

2. In the below figure (figure 1) we can clearly see how the average fitness of the generations increase. Notice the slopes of the runs are not perfectly increasing, this is because the genetic algorithm can sometimes breed two parents that don't produce superior offspring. This is likely due to the roulette wheel sampling occasionally picking parents that are not the best fit. This could be also caused by mutation of the genes.

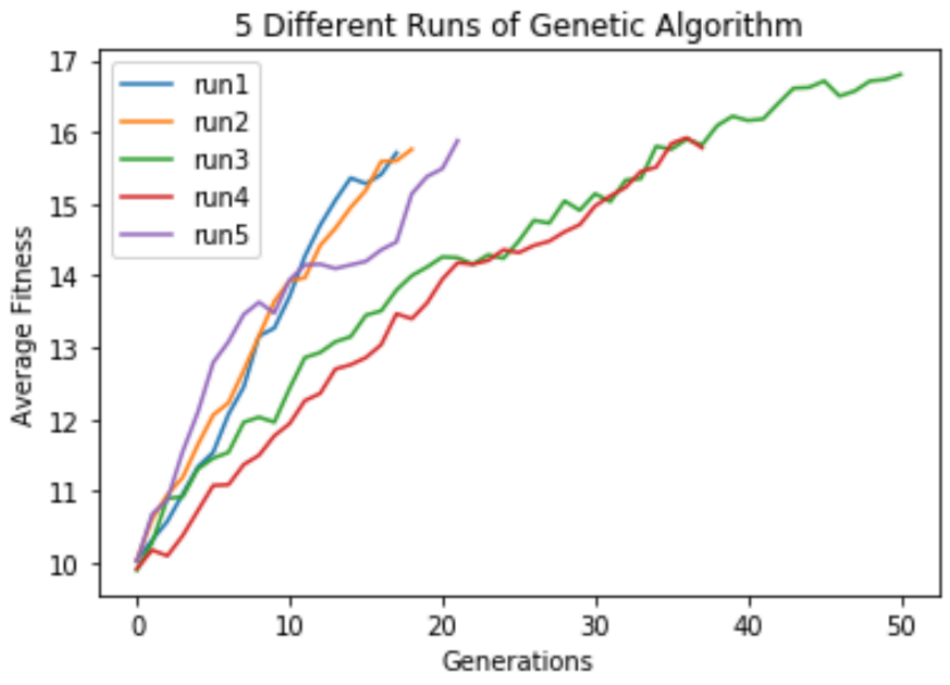


FIGURE 1: GRAPHING OF GENETIC ALGORITHM

3. Running the experiment without crossover leads to a genetic algorithm that is unable to find the optimal solution. Currently the only reason there is change in the algorithm is because of mutations. These mutations can sometimes hurt the algorithm as well as help it but it is a totally random process. Figure 2 below shows the effect of crossover being off.

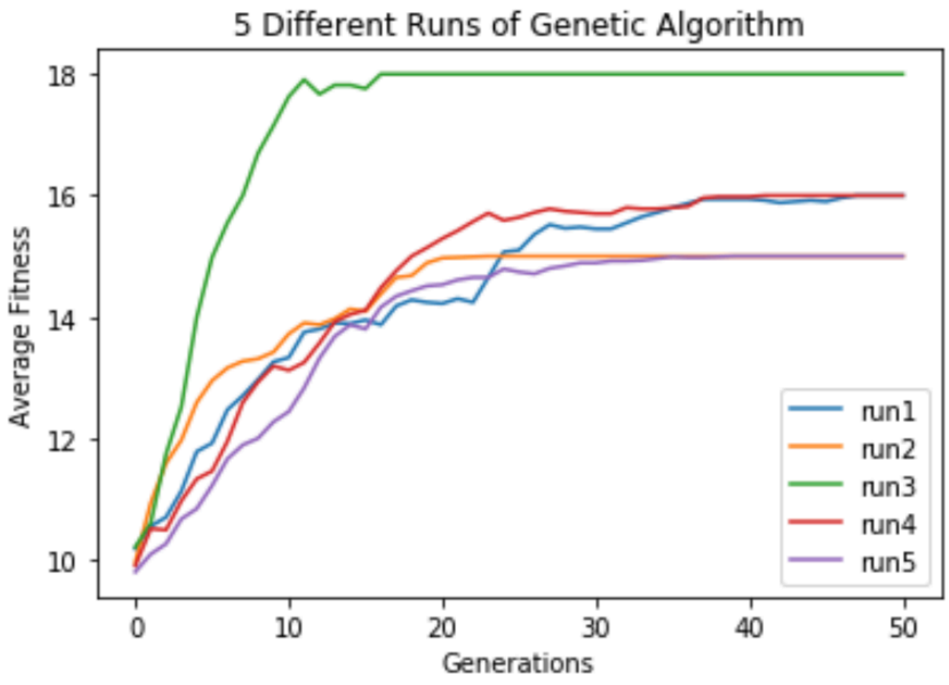


FIGURE 2: GENETIC ALGORITHM NO CROSSOVER

4.

```
Inital Conditions
Population = 100
Crossover rate = 80%
Mutation Rate = 1%
```

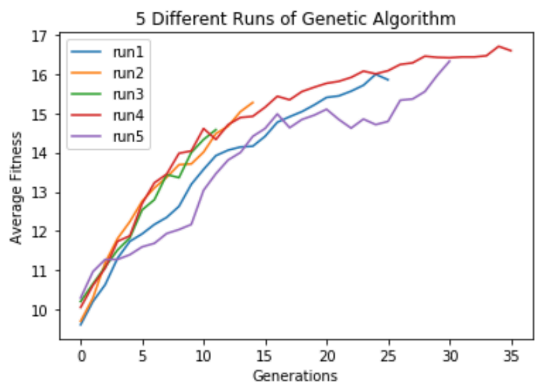


FIGURE 1: BASELINE

```
Inital Conditions
Population = 200
Crossover rate = 80%
Mutation Rate = 1%
```

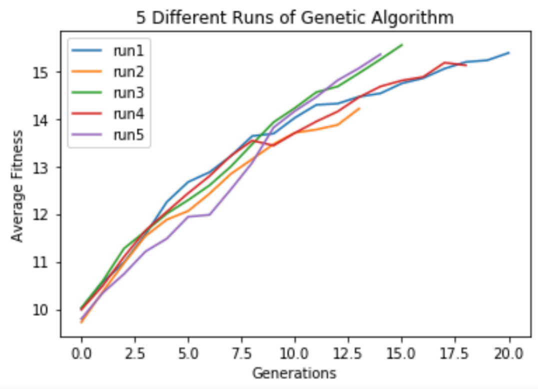


FIGURE 2: HIGH POPULATION

```
Inital Conditions
Population = 100
Crossover rate = 80%
Mutation Rate = 50%
```

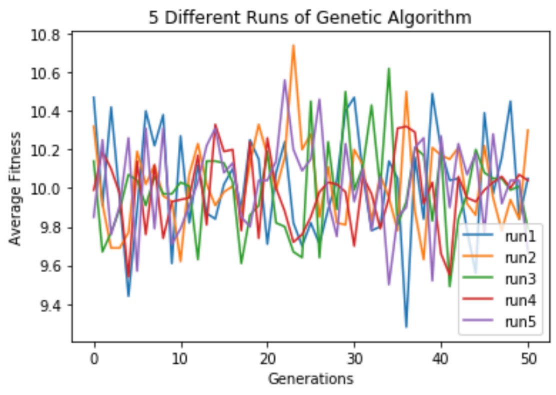


FIGURE 3: HIGH MUTATION RATE

```
Inital Conditions
Population = 500
Crossover rate = 80%
Mutation Rate = 1%
```

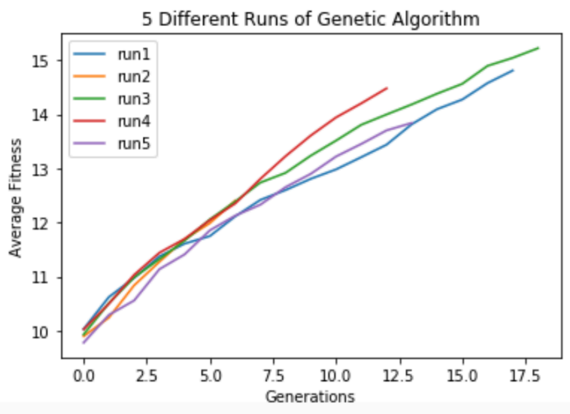


FIGURE 4: VERY HIGH POPULATION

In my experiments population amount had the largest effect on the speed at which the genetic algorithm found an optimal solution. I started my experiments with baseline parameters of population at 100, crossover rate at 80%, and mutation rate at 1% as seen in figure 1.

I next varied my population rate by doubling it from the initial conditions. This lead to the algorithm finding and optimal solution on average at generation 16. This is far faster than the baseline of 21 generations. The results of this experiment can be seen in figure 2. The higher population allows for the genetic algorithm to pick better fitting parents to breed as offsprings.

My next experiments involved varying the mutation rate to 50% from the baseline. This experiment can be seen in figure 3. As one can see, varying the mutation rate to 50% led to bad results. This high of a mutation rate makes it nearly impossible for the algorithm to find a best fit. Run 2 shows us that at generation 24, we spikes on an average population fitness but ended up mutating away from that fitness. Mutation rate needs to be small for this algorithm to work optimally.

Figure 4 shows my final experiment. This experiment bumped the population size up to 500 from 100. As one can clearly see in the figure, a higher population will result in faster convergence upon the optimal solution. The downside of increasing the population size is the decrease in performance of the algorithm. This was the most time intensive test but produced the best results.

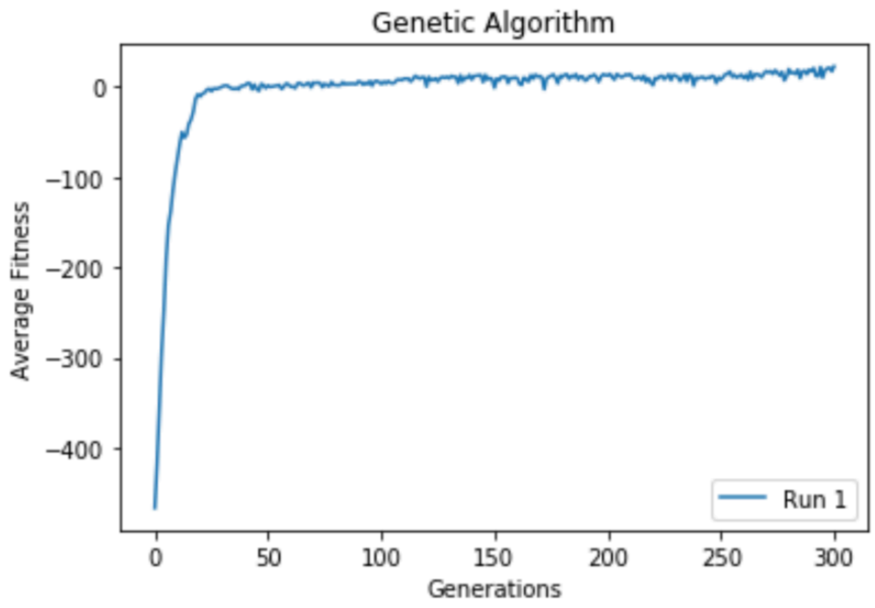
Report: Part 2

In part 2, I adapted the genetic algorithm from part 1 to help Robby navigate his world. The baseline parameters were a population size of 100, a crossover rate of 100%, mutation rate of 0.5%, and a generation size of 300 generations. This baseline strategy produced a peak fitness value of 60 with an average peak fitness value of 48.15. This generation produced a strategy of using the following string:

```
00660215643506315560452045121210515046253615615244012231524005002
44640522313250144150100261123440244050466664201152306464430224546
20015310630023540654225220304610166516150120623154132550230625444
153225061154524360566112641621161646402311325106
```

This string represents the moves Robby should make to maximize Robby picking up soda cans and navigating his world. Below is a graph of the average fitness per generation.

```
Inital Conditions of population = 100
Crossover rate = 100%
Mutation Rate = 0.05%
```



BASELINE

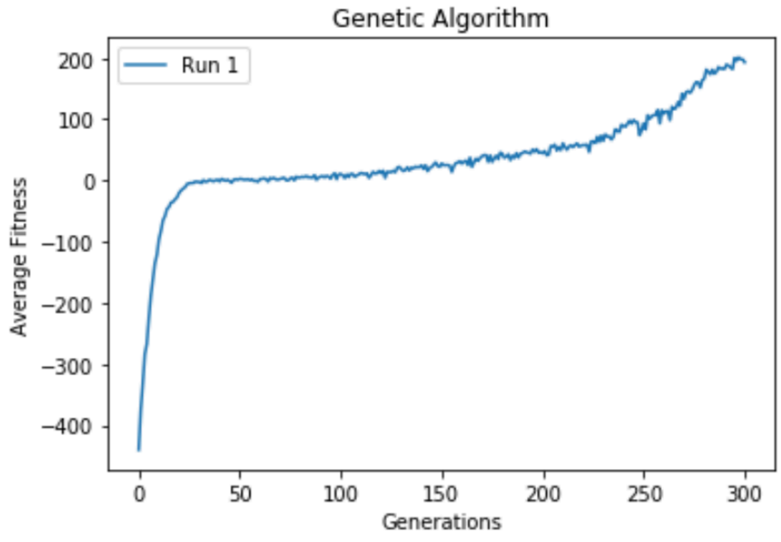
I next adjusted the crossover rate on the genetic algorithm from 100% down to 70% to test the efficiency of 100% crossover verses 70% crossover. At around generation 200, the best fitness was already up to 52 while the average was at 42.7. In the baseline test, at generation 200 the average was only at 15.3 while the best was 38. This shows me that a crossover rate of 100% makes the algorithm take longer to converge upon the best fitness. At generation 300, the best fit of this lower crossover rate was 54. The average fitness of this generation was sitting around 43. This lower crossover rate did not produce a higher peak value of fitness than the baseline but it converged to its peak range much faster and sat within +/- 5 fitness of this peak for over 130 generations. The best strategy of these conditions was:

611666152401435151341125354126103153211430153135330113424336001505662
401405401045605412002254243551124635020402623311146666153041043052123
000613020002025105504366636230154511465153410303335264446156022114154
152015542512060064104524061343603502

Next I adjusted the mutation rates which lead to similar results to part 1. The mutation rate takes a careful balance between enough to mutate and not enough so that every gene is being mutated and can never converge upon a single answer.

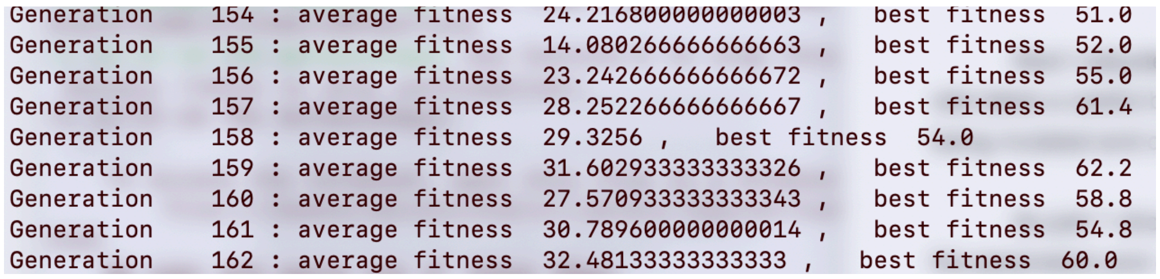
As part 1 showed me, population size has the biggest effect on fitness. Next I adjusted the parameters and decided on a mutation rate of 0.5%, a crossover rate of 80% (provided the best results in part 1), and a population size of 150. I would have liked to have the population higher but sadly increasing the population size slows down the algorithm massively. The graph below shows the average fitness per generation.

```
Inital Conditions of population = 150
Crossover rate = 80%
Mutation Rate = 0.05%
```



HIGHER POPULATION RESULTS IN BETTER FIT

Clearly this generation is preforms much better. At generation 157, this implementation already beat out the baseline peak for best fitness.



```
Generation    154 : average fitness  24.216800000000003 ,   best fitness  51.0
Generation    155 : average fitness  14.080266666666663 ,   best fitness  52.0
Generation    156 : average fitness  23.242666666666672 ,   best fitness  55.0
Generation    157 : average fitness  28.252266666666667 ,   best fitness  61.4
Generation    158 : average fitness  29.3256 ,   best fitness  54.0
Generation    159 : average fitness  31.602933333333326 ,   best fitness  62.2
Generation    160 : average fitness  27.570933333333343 ,   best fitness  58.8
Generation    161 : average fitness  30.789600000000014 ,   best fitness  54.8
Generation    162 : average fitness  32.48133333333333 ,  best fitness  60.0
```

GENERATION 157

The max value for fitness achieved with this larger population was 270.6 with an average value max of 201.35 in generation 297. Clearly increasing the population size results in a much better fitness. Our resulting best strategy was:

```
        650354252650030253005642564656336255651153054636044426605606466
        04062423423645516430500520540530660513515345250521130551542601
        14644165046225402124216002431135354514153351255255151255351336
        5161564541151521206113532652041445002556506423646042535 14
```

The genetic algorithm used could be much more efficient. As it stands, the baseline for this test takes around 20 minutes to fully finish its 300 generations. In order to make this algorithm more efficient, one could decrease the amount of for loops and attempt to rewrite the best fitness function. The data structure used to store the strategies could also be improved massively by using hash tables to store the data. The big O time of a hash table is significantly more efficient that the current implementation.

Below you will find all code used to make the graphs and analyze the data.