Luna McBride

**Assignment 2: Genetic Algorithms**

**Part I**

Note: For the purpose of displaying the stark differences various other parts could cause, I allowed the generations to go past 50. It will be changed back after this part

1. List of generations until best fitness:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Run | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Generations | 51 | 20 | 74 | 8 | 11 | 71 | 13 | 11 | 56 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Run | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| Generations | 37 | 14 | 88 | 30 | 24 | 19 | 32 | 6 | 30 |

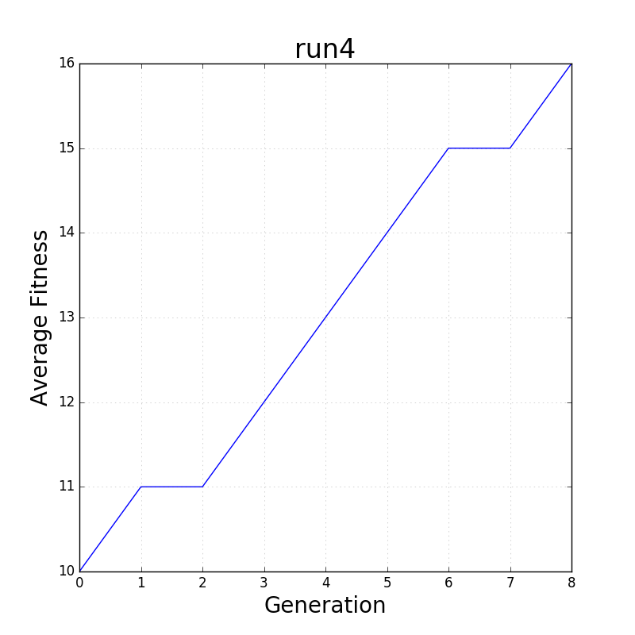
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Run | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| Generations | 13 | 13 | 9 | 31 | 22 | 15 | 9 | 54 | 69 |

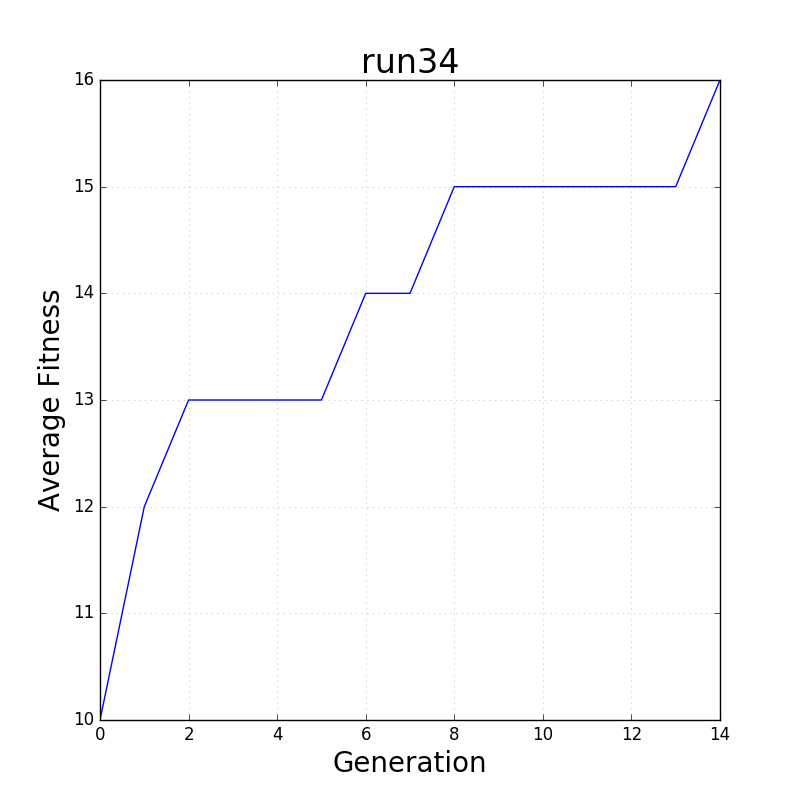
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Run | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| Generations | 35 | 9 | 16 | 126 | 24 | 49 | 14 | 50 | 14 |

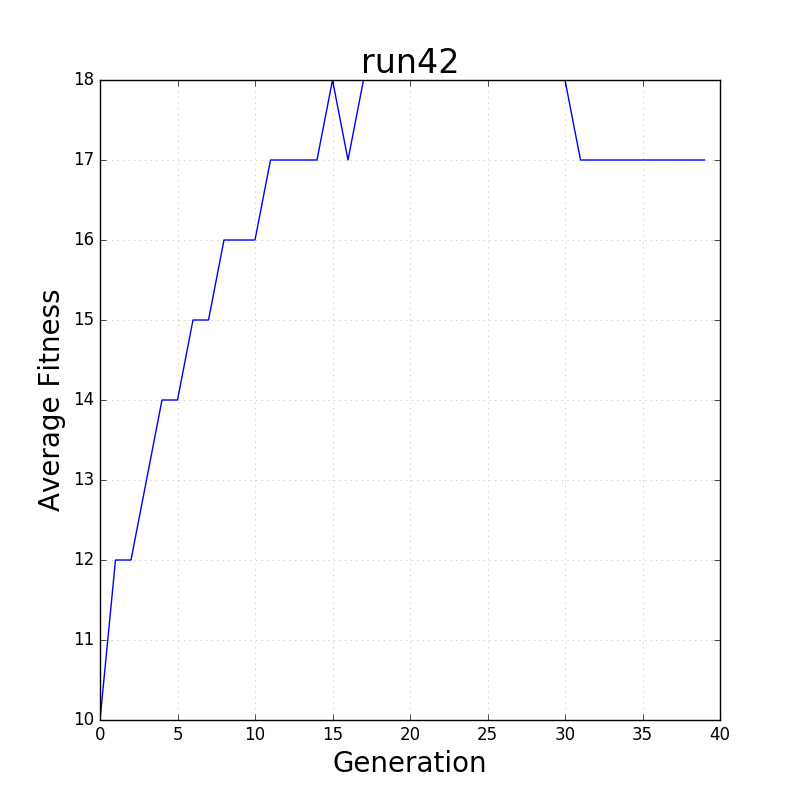
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Run | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 |
| Generations | 25 | 39 | 40 | 28 | 72 | 39 | 5 | 8 | 47 |

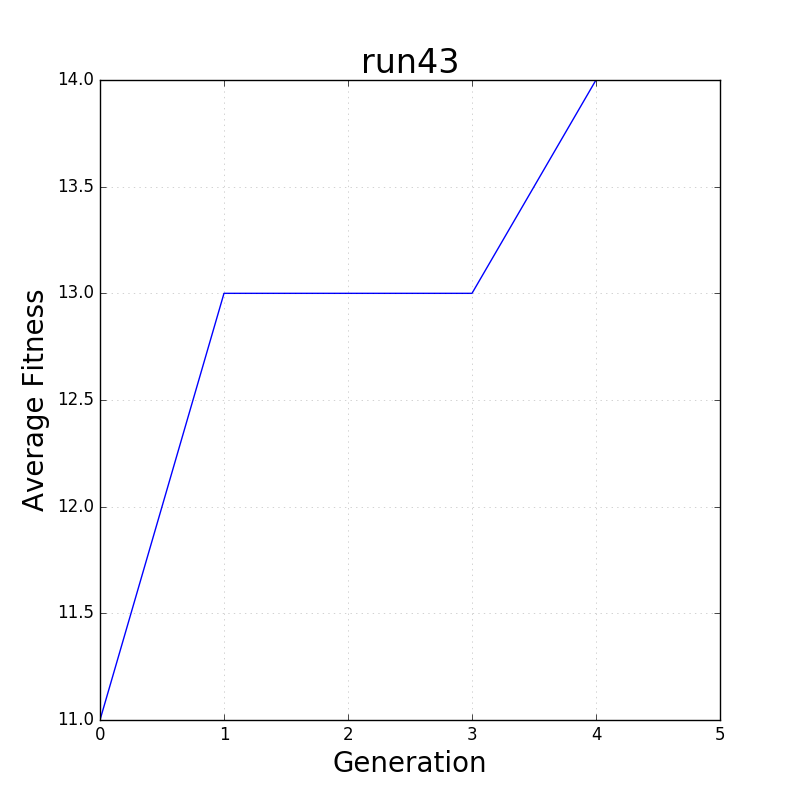
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Run | 46 | 47 | 48 | 49 | 50 |  |  |  |  |
| Generations | 15 | 53 | 25 | 21 | 41 |  |  |  |  |

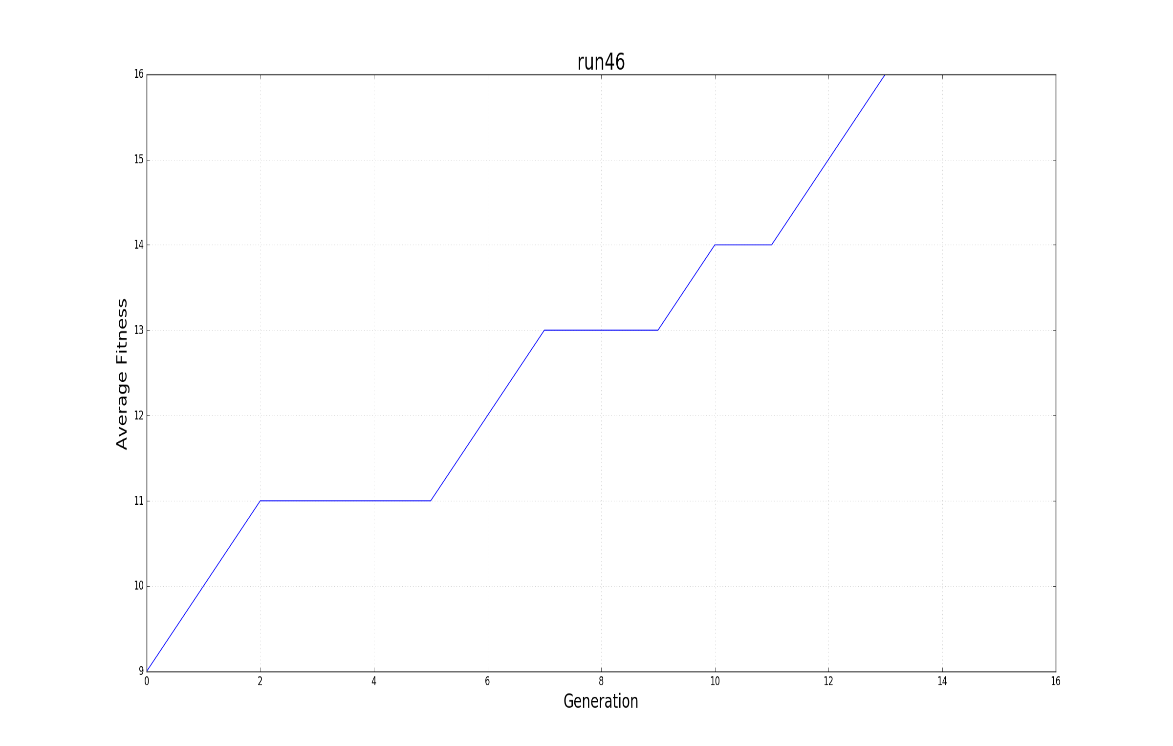
Average: 32.5, Minimum: 5 (#43), Maximum: 126 (#31)





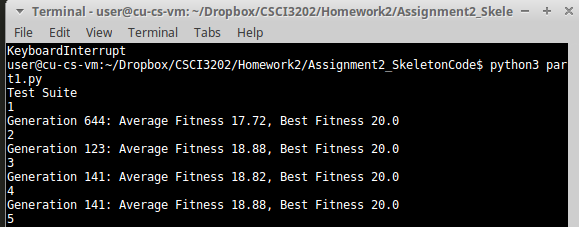




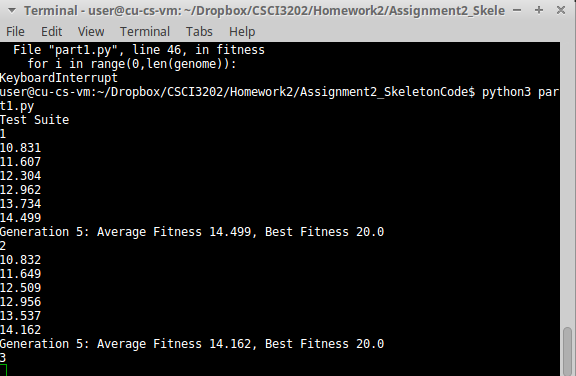


All five of these have a similar structure: The average fitness increases over time. It should be fair to note, however, that all averages were held as integers, so the lines are a bit jagged. However, all became better over time. They tended to stay at the highest fitness until the end, except for run 42 that likely was riding the line around 18, while being floored by being an integer. Interestingly, the highest fitness of these ranged from 14 to 18, which means the crossover and mutations could bring the final round of 1’s from large ranges of previous generations.

1. With no crossover, the algorithm takes so much longer, the majority of generations being over 100 (much more than the average for the 0.7 crossover of 32.5). In addition, average fitness by the end was also much higher by the end than the 0.7 crossover (most averages being 17-18 rather than the wide range). This is because the population was relying on mutations and parent choices without the actual “genetics” of the genetic algorithm. A new generation was not being created in the sexual way, rather, it was as if it were a bacterium relying on splitting itself over and over with mutations from replicating DNA. As we know however, bacteria change slower, but split so quickly that change seem relatively quick, but only by doing more generations in a shorter period of time.
2. Per individual rate:
   1. Population: The higher the population, the less generations it took per run; however, the slower the program ran. Of course, the more strings, the higher the chance there is to quickly get the full string of 1’s per generation. It also takes more resources to run through 1000 genomes rather than 100, explaining this behavior.

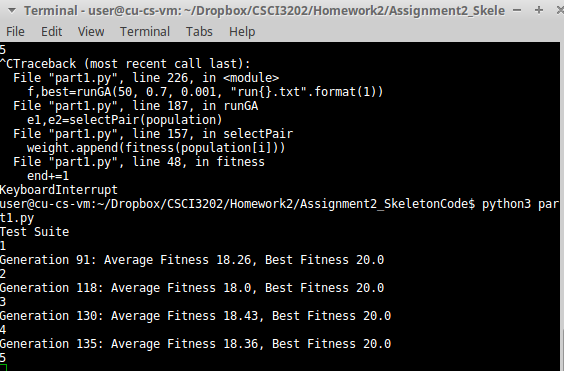


Low Population (50), high generation count

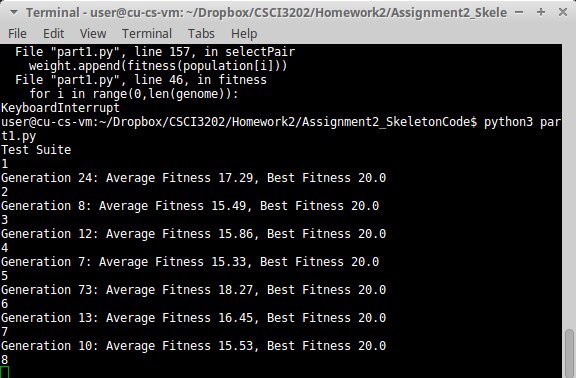


High Population (1000), low generation count (Note, its hard to show slowness, but these two runs took quite a while relative to regular runs)

* 1. Crossover: The program went through less generations the higher the crossover rate. Large chunks trade off in a crossover, opening up a higher possibility of the most fit giving the span of 1’s that negates any last 0’s in one child and vice versa with low fitness ones that happen to have 1’s at the end where needed. Of course, you can only go so fast and the variability was more visible at lower crossover values, but the difference was still there between 0.9 and 1.

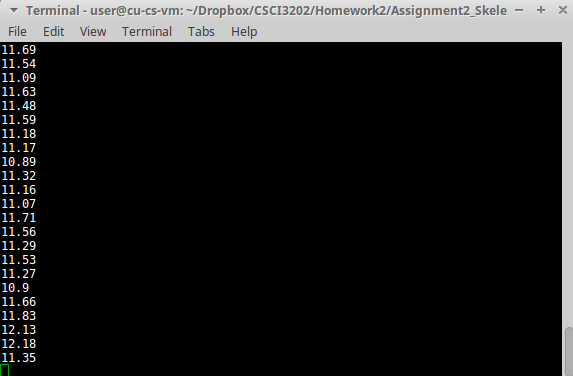


Low crossover rate (0.1), high generation count

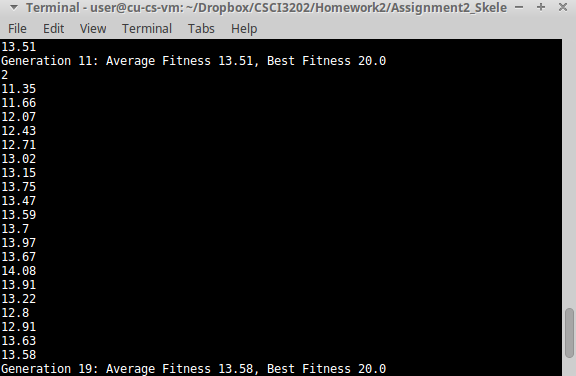


High crossover rate (1), lower generation count

* 1. Mutation: This one was the most interesting of the bunch. At sufficiently low values, the mutations sped up the generations, giving a lucky break to the genomes just one or two values away. In addition, higher values lead to a lower average fitness before finding the best outcome, meaning one just so happened to get lucky to get the specific configuration of mutations to get to 20. This was until about 0.05 or higher, where the mutation happened too fast and caused a near-infinite loop of averages wiggling around a single value. The mutations just become so cumbersome when happening too much that it actively becomes harder to get all 1’s without one of those becoming a 0 in a lot of cases, making near infinite loops when it is trying too hard.



Higher mutation rate (0.1). The generation count was bumped up to show this, but this is after about 8 seconds. The averages wiggle up and down, but very rarely actually get the solution.



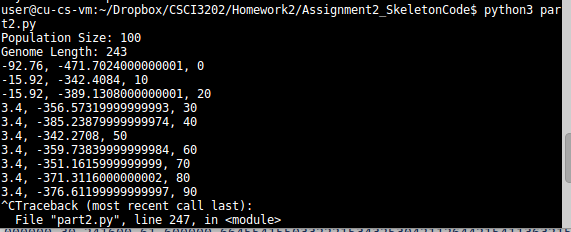
Higher population, but not too high (0.03). The jump from a lower average (13.5) to the proper solution is evident here.

**Part II**

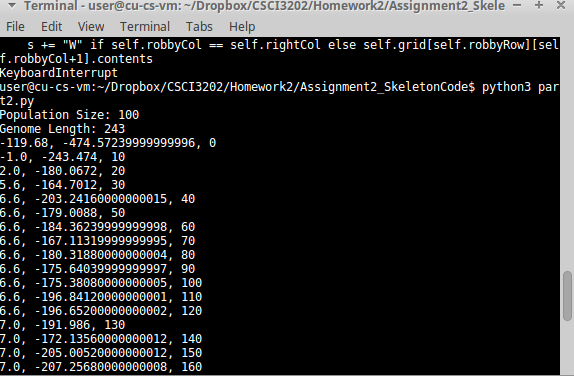
Of all the different strategies, there were some that made our robot better and some that slowly made it regress rather than improve. Of the following inputs, each had various effects:

1. Mutation: this one had the craziest effect, which needed to be explained first. The starting value was set to 0.15, which was driving me insane with how the averages stayed negative and actually got lower despite the actual code all seeming right. Going higher, like the TA said should help, made it even worse. However, remembering the infinite-esque state of higher values before, values were lower. These not only gave an actual fair solution, but progressed without plateau in the 300-generation set (going up, with a little oscillation as the nature of the algorithm allows for it). The best value tested was 0.01.

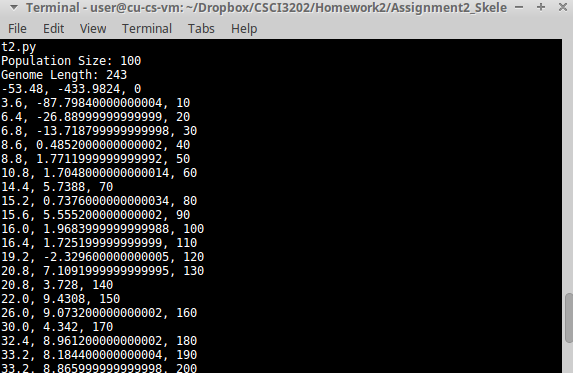
Structure: Best, Average, Generation



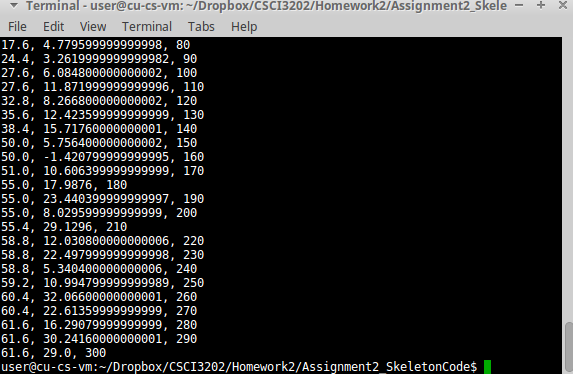
Mutation Rate 0.4. Terminated early, however, the crazy low averages exemplify how bad the mutation actually was.



Given Mutation Rate: 0.15. The averages stayed below -150, which is very low, which did not give the robot the chance to fully explore.

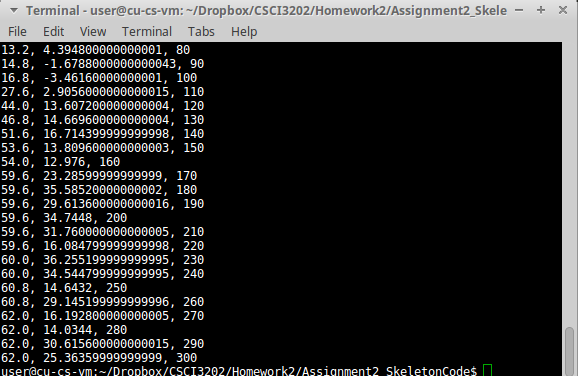


Mutation Rate: 0.005. Without a high value, this robot could actually learn and explore. Averages entered the positives for once, which also showing improvement over time.

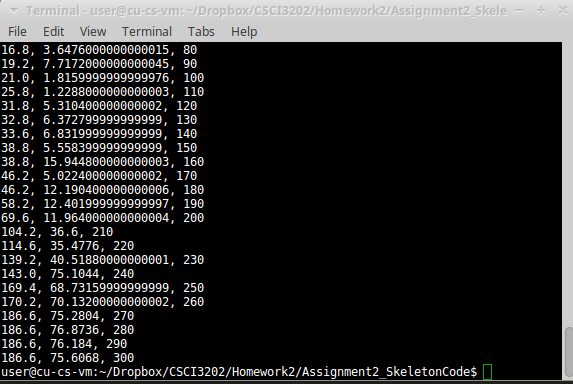


Mutation Rate: 0.01. This has the benefit of the small number without going over the hump displayed by the larger numbers. This one progresses and reaches an ultimate value of 61.6, higher than any other one has given. It will be used as the baseline to see how the others can improve it.

1. Crossover: This one turned out interesting as well. Going in, the expectation was higher crossover, faster conclusion like with part 1. The exact opposite turned out to be true. The crossover rate of 1 gave a best similar to the unchanged rate shown in the Mutation Rate 0.01 image, just a little higher, as variation can occur. The 0.1 rate, however, more than doubled that rate. It is likely because of the difference between 0-6 instead of 0-1 allowed values, as switching over is not as simple as on and off. Instead, it turned out better not to meddle too deeply; rather, the lower rate of crossover made one crossover more meaningful than if it were more constant, which reigns true in many facets of life.

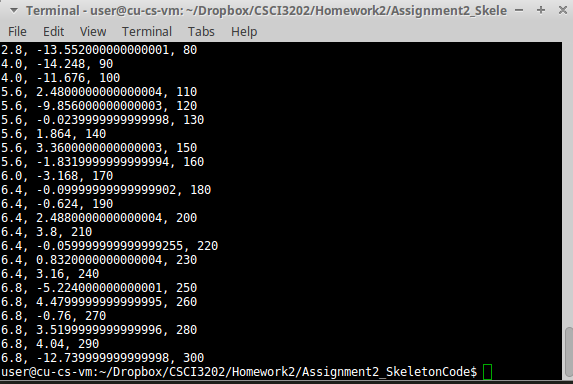


Crossover Rate: 1. Results are very similar to the Mutation Rate 0.01 image (Crossover 0.7), likely only surpassing it by variation that can easily occur in such an automated process.

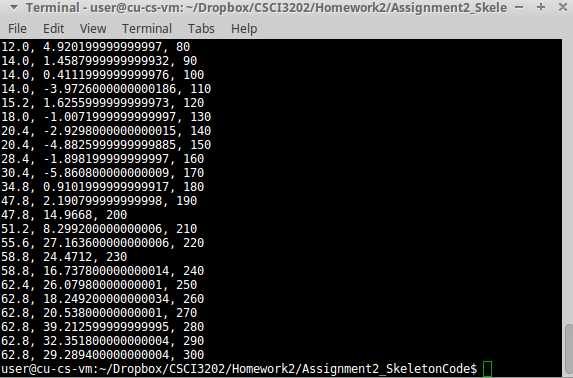


Crossover Rate: 0.1. The heightened growth given by the small number of crossovers was staggering, but sometimes only a touch of something good leads to a better result. This will be used in addition to test population.

1. Population: With the sample space size changing, of course fitness is going to change. A smaller population (10) did not give enough data to work with. There is not much of a population to give any variety, so of course it is not going to show off said variety. The super large (200) case was different, however. This one, under the same condition as the crossover 0.1 above, provided a lower fitness, coming similar in numbers to the crossover rate 1. This is likely because the larger population was too big to get a proper footing. In the crossover 0.1 with 100 population, there was a distinct point in which the algorithm got its footing and jumped to higher places, but the higher population could not do that within the time frame we allowed. Even so, it took a lot longer per generation to process, which emphasizes the importance of getting a sample size that both fits the type of data and does not strain computer fans too much.



Population: 10. Not much variation can come from a small population. Boy did it run fast, however.



Population: 200. The population could not get the proper footing to jump like with pop 100. Also, I got the dishes done before this could finish. Just saying.