

COSC 420: BIOLOGICALLY-INSPIRED COMPUTATION
PROJECT 4 - “GENETIC ALGORITHMS”

CLARA NGUYEN

TABLE OF CONTENTS

1. Synopsis	5
2. Development	5
3. How it works	6
3.1. Algorithm	6
3.2. Crossovers and Mutations. A solution to a problem.	7
3.2.1. The Problem	7
3.2.2. Crossovers	7
3.2.3. Mutations	7
4. Hypotheses	8
4.1. Synopsis	8
4.2. List of Hypotheses to Test For	8
5. Simple Experiments	9
5.1. Experiment 1: The Default Test	10
5.1.1. Graphs	10
5.1.2. Discussion	11
5.2. Experiment 2: An increase of genes	12
5.2.1. Graphs	12
5.2.2. Discussion	13
5.3. Experiment 3: A test of more individuals	14
5.3.1. Graphs	14
5.3.2. Discussion	15
5.4. Experiment 4: A test of even more individuals	16
5.4.1. Graphs	16

5.4.2. Discussion	17
5.5. Experiment 5: Maximum Crossover	18
5.5.1. Graphs	18
5.5.2. Discussion	19
6. Complex Experiments with Unusual Results	20
6.1. Experiment 6: Flat-lining Best Fitness	21
6.1.1. Graphs	21
6.1.2. Discussion	22
6.2. Experiment 7: Maximum Mutation	23
6.2.1. Graphs	23
6.2.2. Discussion	24
7. Extreme Experiments	25
7.1. Extreme Test 1: 3000 people in 100 generations	26
7.1.1. Graphs	26
7.1.2. Discussion	27
7.2. Extreme Test 2: 50 genes per person	28
7.2.1. Graphs	28
7.2.2. Discussion	29
7.3. Extreme Test 3: 1000 generations	30
7.3.1. Graphs	30
7.3.2. Discussion	31
8. Observations	32
8.1. Hypotheses Results	32
8.2. Can we go even further beyond?	33

9. Project Files and Submission Information.....	34
9.1. Node.js Source Code Files	34
9.2. Shell Scripts	34
9.3. Excel Documents	35
9.4. CSV Experiment Files	35
9.5. Docx Master File.....	35

1. SYNOPSIS

In this project, I explored how to implement a basic genetic algorithm to simulate a civilization. The civilization consists of a number of individuals in a generation that have a number of genes. To go to the next step of the simulation, two individuals can mate and produce offspring. To make the civilization grow, crossover and mutation properties are added into the algorithm to produce more variety in the offspring that are born. The goal, overall, was to write a simulator that produced a population that has the most fit individuals.

2. DEVELOPMENT

The simulator was written in Node.js. It was written with an object-oriented aspect in mind. There is an object for an entire simulation, a generation, and an individual. This flexibility allows the end user to make multiple simulations within the same program easily, within a few lines of code. The application is also fully tweakable, accepting 5 parameters:

```
node src/ga.js genes population generations mutation_prob crossover_prob
```

For example, the default configuration consists of 20 genes, 30 individuals per generation, 10 generations, a 0.033 probability of mutating a gene, and a 0.6 probability of genes to crossover. Therefore, the following must be passed into the program:

```
node src/ga.js 20 30 10 0.033 0.6
```

The program will output a valid CSV file containing information about the “Generation”, “Average Fitness”, “Best Fitness Individual”, and the “Average Correct Bits”. Pipe the information into a file, and then you can generate a graph in another application. You can also execute it with this:

```
./run.sh 20 30 10 0.033 0.6
```

3. HOW IT WORKS

3.1. ALGORITHM

```

initialize data structures for keeping statistics
create experiment object
for i from G to number of runs
    calculateFitness()
    determineParentsAndMate()
    destroyCurrentCivilization()
    copyNewCivilizationIntoNewGeneration()
compute averages over number of generations
write data file

```

Figure 1. *Pseudo Code for the Simulator.*

The program's structure was written with pseudo code from Figure 1 in mind. The program abides by a mathematical equation that was given to us in a lab writeup and is shown in Figure 2. On top of that, the simulator uses bit logic of generating numbers by setting random bits to 1 or 0, and then converting it into a number. This number is used as the x value in the equation in Figure 2.

$$F(s) = \left(\frac{x}{2^\ell}\right)^{10}$$

Figure 2. *Fitness Equation.*

This equation is the only one used. After all individual fitness values are found, they are normalized so that, when added up, they result in 1. This is used for determining parents to mate and produce offspring for the next generation. To mate, 2 parents are chosen at random, and they cannot be the same individual. Each parent brings birth to two children. Their genes are determined by their parents, crossover, and mutations.

3.2. CROSSOVERS AND MUTATIONS. A SOLUTION TO A PROBLEM.

3.2.1. THE PROBLEM

Without crossover and mutation, the children who are born simply have their mother and father genes copied over. This means that the fitness of each new individual would remain the same, and their genes would as well. Society would never improve with this. This is where crossover and mutation comes into play.

3.2.2. CROSSOVERS

As shown in the Development section, the simulator takes a “crossover_probability” parameter. This allows for a chance for genes to crossover between parents at a random position. With this, we can have some variety in the individuals of the next generation. It isn’t a complete solution, however, as it just allows the genes to switch at a random point.

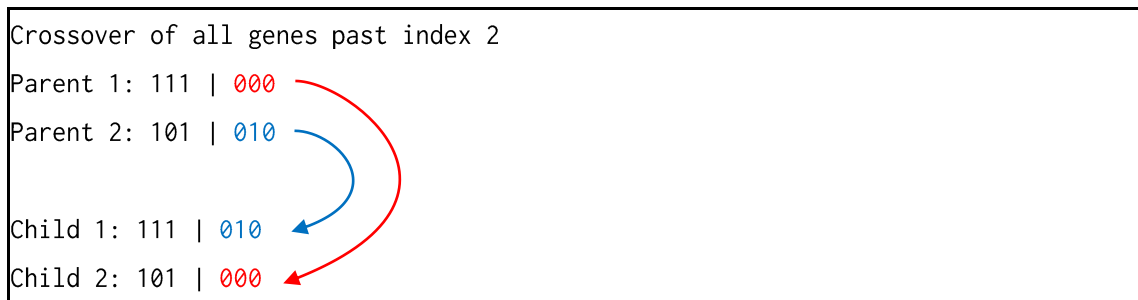


Figure 3. Basic Crossover of 2 individuals’ genes starting from index 3 on to the end.

3.2.3. MUTATIONS

Crossover allows for the redistribution of genes, but no data will be gained, nor lost. Because of this, a generation can not improve over time. The solution to this is to change data up through mutations. This simply involves going through the child’s genes and randomly switching the values if a random number is under the percentage given. This variety will cause generations to either improve to have more fit individuals or be their complete and utter downfall.

4. HYPOTHESES

4.1. SYNOPSIS

In running this simulator, there are a few hypotheses that can be analyzed just by tweaking the configurations of it. We are passing in 5 parameters: genes, population, generations, mutation probability, and crossover probability. And we know what each of them do. So, let's set up some hypotheses to try to figure out.

A lot of these hypotheses were carefully chosen to reflect what is possible to obtain via tweaking the simulator's parameters. They were also chosen after running the default configuration, so we at least know what kind of graphs to base our data off of and compare from.

4.2. LIST OF HYPOTHESES TO TEST FOR

1. What would happen if the genes were increased/decreased from the default configuration?
2. What would happen if the population was larger or smaller?
3. What would happen if the mutation probability was 100%?
4. What would happen if the crossover probability was 100%?
5. Is the algorithm capable of standing a test of time? Can it last an infinite amount of generations?

With these set up, let's begin our experiments. There are 10 experiments. 5 are normal ones, 2 are experiments that showed unusual results, and 3 of them are extreme tests that push the simulator to its limits.

5. SIMPLE EXPERIMENTS

Now we will get to experimentation. The procedure for testing goes as follows:

- Have a set of parameters to pass into a function
- Execute the simulation 5 times with those parameters
- Generate graphs comparing “Average Fitness”, “Best Fitness”, and “Average Correct Bits”.
- Show them and discuss. Compare to prior experiments.

As per our usual procedure, we will try to experiment with extreme cases as well to try to understand how this algorithm performs under brutal testing. In this case, it'll show whether or not a civilization stands a chance. The following experiments will be run:

#	Genes	Individuals	Generations	Mutation %	Crossover %
1	20	30	10	0.033	0.6
2	30	30	10	0.033	0.6
3	20	40	10	0.033	0.6
4	20	100	10	0.033	0.6
5	20	30	10	0.033	1
6	5	30	10	0.033	0.6
7	20	30	10	1	0.6
8	30	3000	100	0.033	0.6
9	50	3000	100	0.0001	1
10	50	3000	1000	0.0001	1

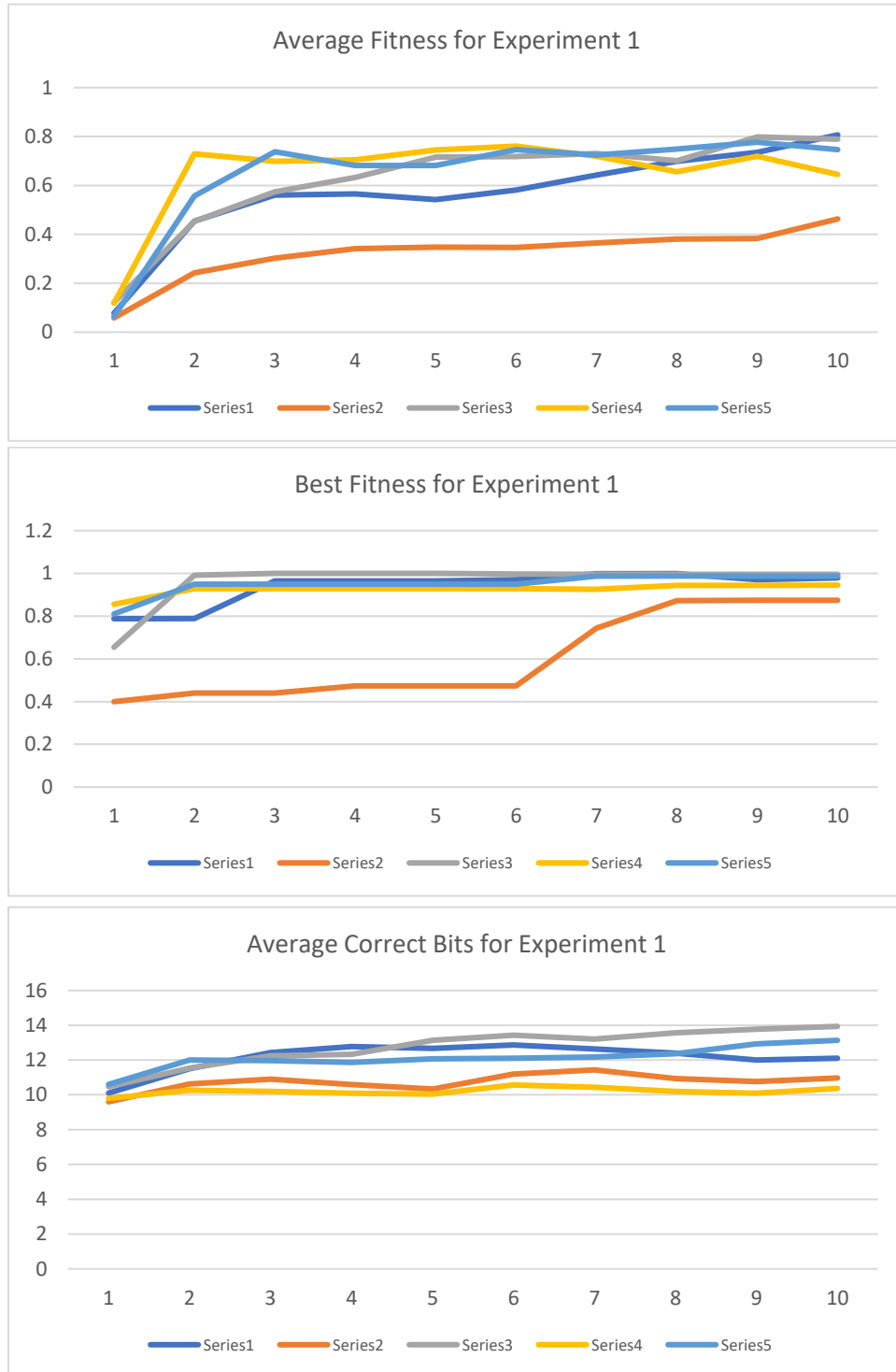
Table 1. Experiments to be run in simulation for analysis.

The experiments are minor adjustments of the parameters to demonstrate the effects of each one. Values marked in **blue** are at an abnormally low value, meanwhile **red** is for abnormally high values. The experiments were carefully chosen to demonstrate the alterations of every possible parameter that can be passed into the simulator.

5.1. EXPERIMENT 1: THE DEFAULT TEST

Parameters: 20 Genes, 30 Individuals, 10 Generations, 3.3% Mutation, 60% Crossover

5.1.1. GRAPHS



Figures 4 (top), 5 (middle), and 6 (bottom). Data for Experiment 1.

5.1.2. DISCUSSION

To start the experiment series, we will go with the suggested settings in the lab writeup. In this experiment, there are only 10 generations. But those 10 generations are enough to have the “Best Fitness Individual” come close to 1.0. We can set a goal to find settings to find the perfect individual, keeping these graphs in mind. The average fitness of the first generation is very pathetic, at an average of exactly 0.087695685. However, as the generations proceed, the value always goes up. The same is true for the individual of the best fitness and the average number of correct bits for each individual. Run 2, as shown in Figure 4 and Figure 5, has a slow start in mutating genes. As a result, it ends up being the weakest of the 5 civilizations generated.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.806694932	0.978916729	12.1
2	0.462792945	0.874017072	10.96666667
3	0.788704706	0.995070934	13.93333333
4	0.64528258	0.94488295	10.36666667
5	0.74644472	0.986606003	13.13333333

Table 2. Summaries of the runs in Experiment 1.



Figure 7. Graph comparing each run in Experiment 1.

5.2. EXPERIMENT 2: AN INCREASE OF GENES

Parameters: 30 Genes, 30 Individuals, 10 Generations, 3.3% Mutation, 60% Crossover

5.2.1. GRAPHS



Figures 8 (top), 9 (middle), and 10 (bottom). Data for Experiment 2.

5.2.2. DISCUSSION

An increase of genes hasn't changed the graphs by much. The only values that changed accordingly were the average correct bits, and that was to account for the increased number of genes added to each individual. If we added more, it can be speculated that the number would be even higher, and would likely increase at a linear rate. Due to hardware restrictions, the maximum genes a person can have is 64. In experiments such as Extreme Test 2 (6.2), we test 50 genes and get an average correct bit amount of around 30. Because the number increases due to the gene count, we can generate an equation to predict the amount of correct bits at the end of a simulation:

$$ACB(g) = \frac{9g}{15}$$

Figure 11. Equation to predict **ACB** (Average Correct Bits) from **g** (genes).

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.780522446	0.999037476	18.36666667
2	0.79968595	0.995102821	17.6
3	0.653577913	0.97977056	17.63333333
4	0.633699801	0.97504159	16.7
5	0.662797962	0.976428552	18.13333333

Table 3. Summaries of the runs in Experiment 2.

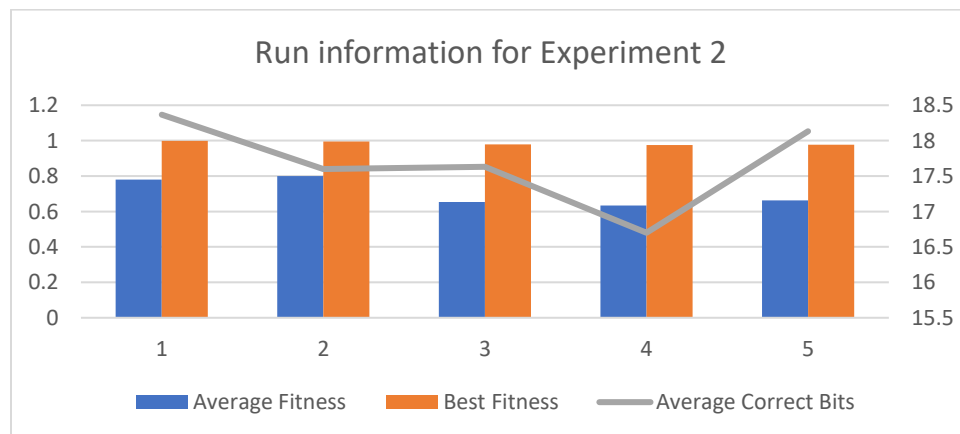


Figure 12. Graph comparing each run in Experiment 2.

5.3. EXPERIMENT 3: A TEST OF MORE INDIVIDUALS

Parameters: 20 Genes, 40 Individuals, 10 Generations, 3.3% Mutation, 60% Crossover

5.3.1. GRAPHS



Figures 13 (top), 14 (middle), and 15 (bottom). Data for Experiment 3.

5.3.2. DISCUSSION

As shown in Figure 11 in Experiment 2, because there are 20 genes per individual, we are able to predict the average number of correct bits to be around 12. The graphs agree with this estimate, as they range from 11.625 to 12.725 in the final generation. From the graphs in this experiment compared to the previous one, it does not appear that more individuals has affected any of the three graphs at all. The average fitness, best fitness, and average correct bits (though accounting for less genes in this simulation) are all similar. So the theory then is whether increasing the number of individuals actually affects the data in any meaningful way. Let's increase the count by a bit more.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.650112701	0.997114121	12.725
2	0.79343358	0.994188305	12.875
3	0.754308871	0.999818817	12.6
4	0.756151017	0.994121899	12.375
5	0.732642124	0.9891056	11.625

Table 4. Summaries of the runs in Experiment 3.

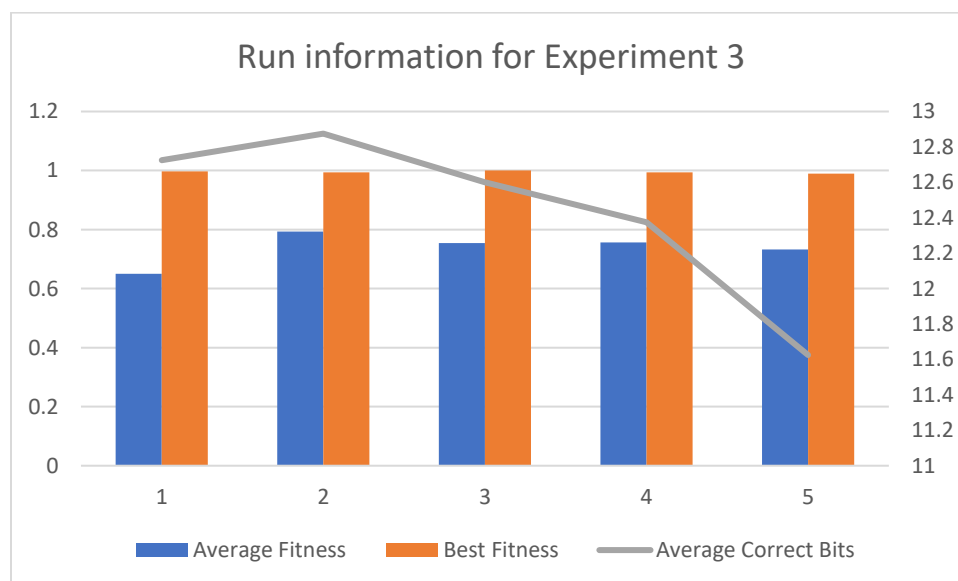


Figure 16. Graph comparing each run in Experiment 3.

5.4. EXPERIMENT 4: A TEST OF EVEN MORE INDIVIDUALS

Parameters: 20 Genes, 100 Individuals, 10 Generations, 3.3% Mutation, 60% Crossover

5.4.1. GRAPHS



Figures 17 (top), 18 (middle), and 19 (bottom). Data for Experiment 4.

5.4.2. DISCUSSION

It seems that our theory is proven wrong. The number of individuals does have an impact on the graphs, but it isn't a visible change unless the number of individuals changes dramatically. In this case, we tested 100 individuals, and the graphs look a lot more smoother and close together than the previous experiments' graphs. This is not on purpose. It seems that, with the more individuals we have, it is easier for a single individual to have a very high fitness value early on in the simulation. Every run in the simulation has a single individual with a fitness of over 0.9 at the second generation.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.738287587	0.993202114	13.3
2	0.747240652	0.99965673	12.47
3	0.790074064	0.998208538	12.54
4	0.740322875	0.999847423	11.85
5	0.773890523	0.990229914	13.09

Table 5. Summaries of the runs in Experiment 4.

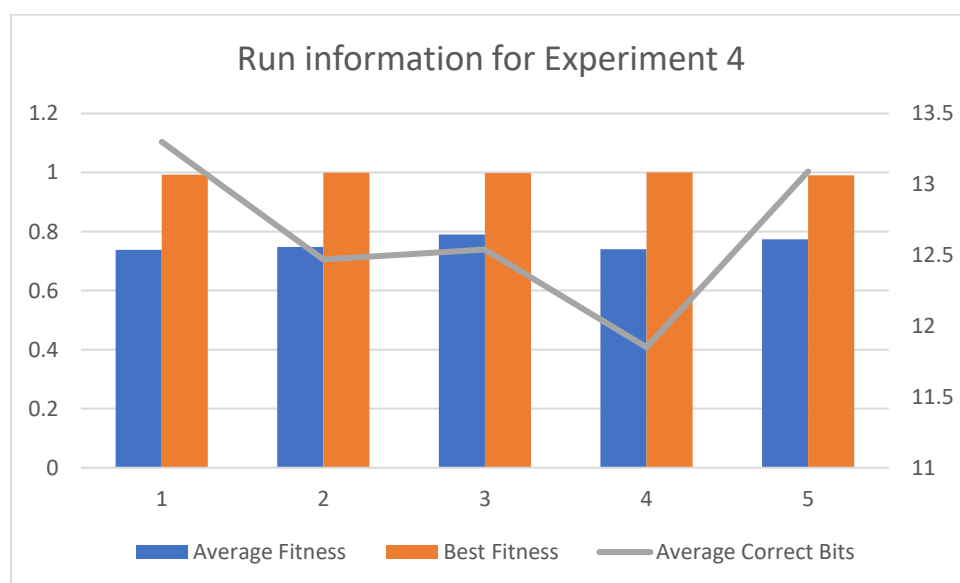
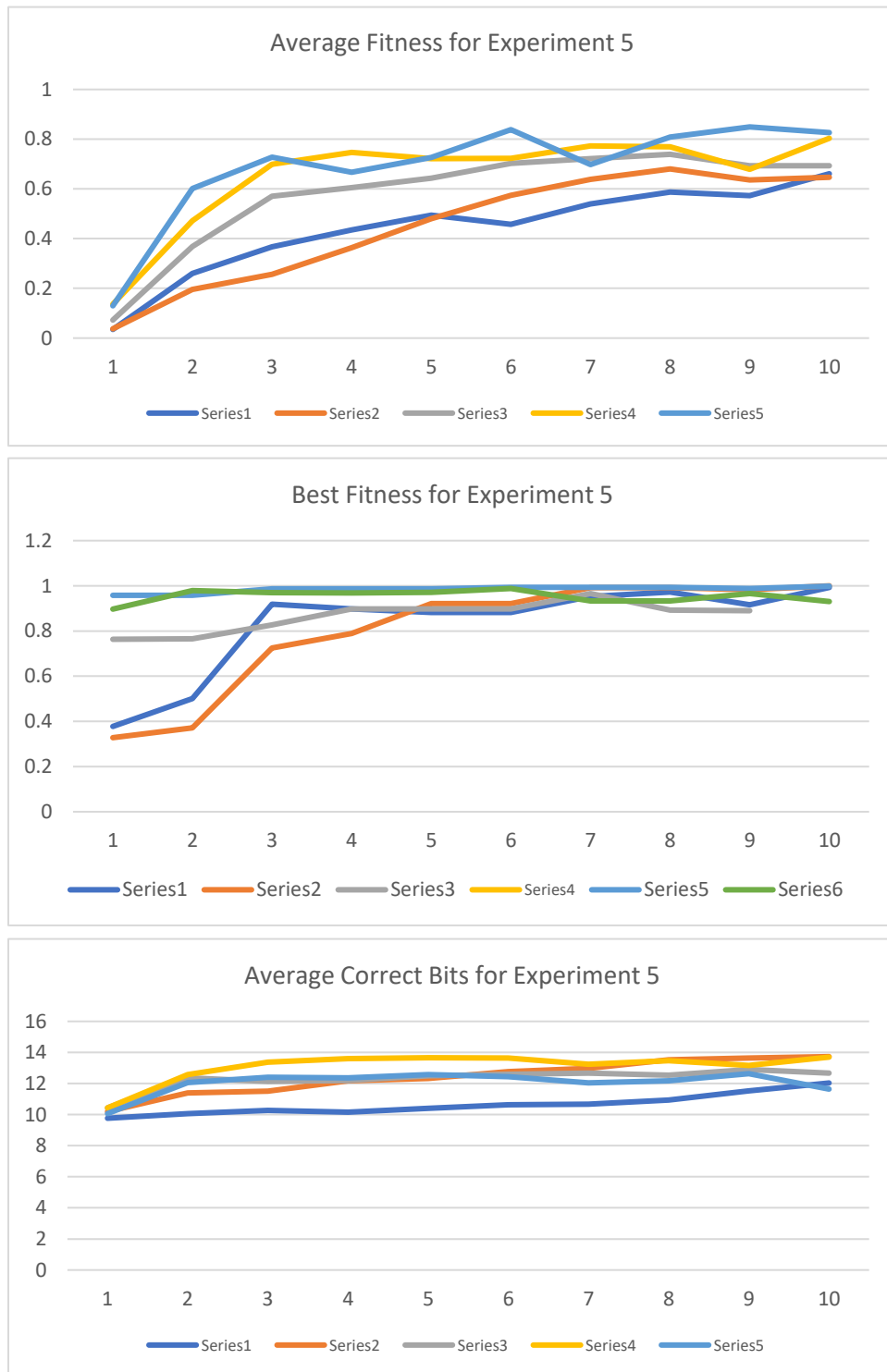


Figure 20. Graph comparing each run in Experiment 4.

5.5. EXPERIMENT 5: MAXIMUM CROSSOVER

Parameters: 20 Genes, 30 Individuals, 10 Generations, 3.3% Mutation, *100% Crossover*

5.5.1. GRAPHS



Figures 21 (top), 22 (middle), and 23 (bottom). Data for Experiment 5.

5.5.2. DISCUSSION

In theory, crossover shouldn't affect the overall performance of the final generation. The reason for this is because it just simply switches genes. No data is lost, nor gained. The simulator, when it decides to perform crossover, will choose a random index and switch all genes after that index between two children. The simulation is based on Experiment 1, except the crossover chances have been maxed out to 100%. The resulting graphs look the same. The results in Table 6 are also around the same.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.660875606	0.992709348	12.03333333
2	0.6467315	0.999876029	13.73333333
3	0.692937807	0.890287997	12.66666667
4	0.803545659	0.998541836	13.7
5	0.825800366	0.930563371	11.63333333

Table 6. Summaries of the runs in Experiment 5.

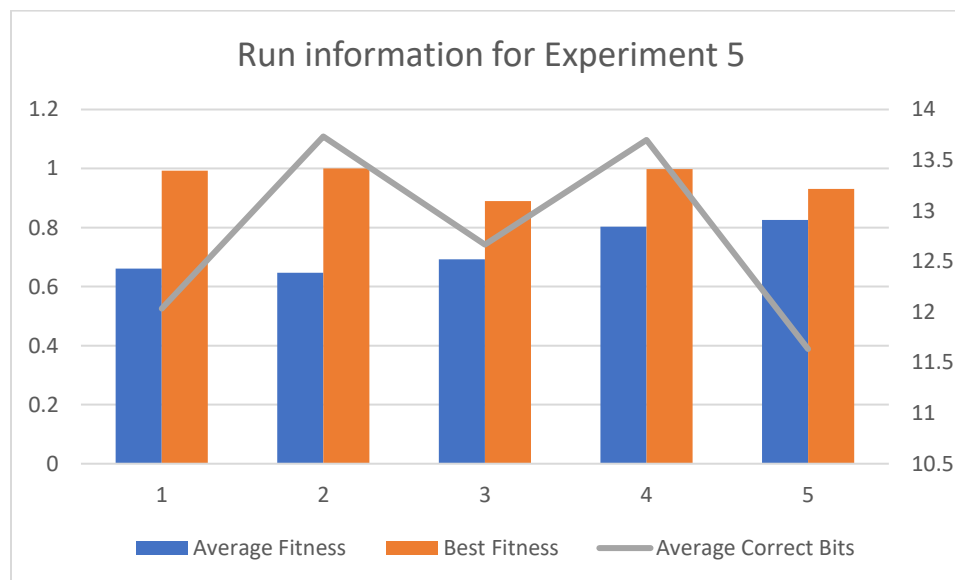


Figure 24. Graph comparing each run in Experiment 5.

6. COMPLEX EXPERIMENTS WITH UNUSUAL RESULTS

The previous experiments were simple manipulations of the simulation parameters to allow for some variety in the results. However, there are some unusual results that can spawn from the simulator as well. The next few experiments will cover unusual cases that were encountered throughout the simulation. The reasoning for these is explicitly because of how the algorithm used to generate the civilization works.

The following simulations produced unusual results with their civilizations:

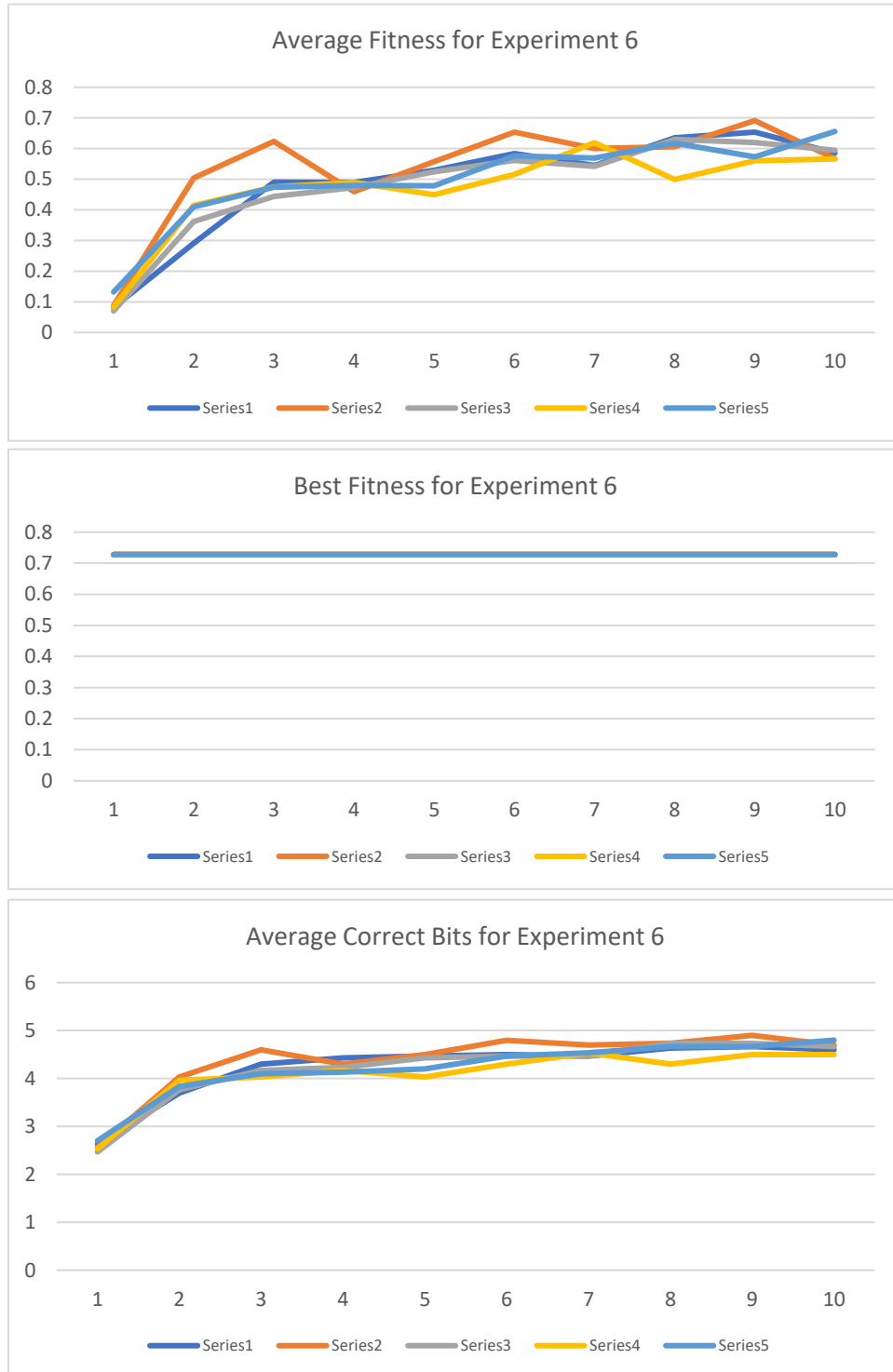
- **Experiment 6** – Best Fitness never improved. Stayed in one place.
- **Experiment 7** – Maximum mutation makes each generation alternate between being fit and the opposite.

These experiments are not including cases where a generation would suddenly shoot down to barely any fitness and stay there. Certain parameters will cause a simulation to tank and have all of the values extremely close to zero. For the sake of these simulations, we will not include those, as they are all pretty much the same results as each other and do not generate any interesting results at all. The only ones included are the ones that generated intriguing results that either abused the simulator's algorithm, or just gave an unusual result.

6.1. EXPERIMENT 6: FLAT-LINING BEST FITNESS

Parameters: 5 Genes, 30 Individuals, 10 Generations, 3.3% Mutation, 60% Crossover

6.1.1. GRAPHS



Figures 25 (top), 26 (middle), and 27 (bottom). Data for Experiment 6.

6.1.2. DISCUSSION

This experiment is an unusual case where the best fitness never improves for the generation. However, the average fitness and the average correct bits managed to increase a bit. The graphs are fairly similar to previous experiments. However, the average number of correct bits happens to break the equation listed in Experiment 2. This could mean that the relation between genes and the number of correct bits being a 9/15 ratio only applies whenever there are more than 5 genes per individual. The low number of genes is believed to be the reason for this behavior. After testing with 6-9 genes, this appears to be the case. An odd number of genes doesn't contribute to it.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.585465502	0.727976157	4.6
2	0.566887822	0.727976157	4.7
3	0.59480052	0.727976157	4.666666667
4	0.565709603	0.727976157	4.5
5	0.655937737	0.727976157	4.8

Table 7. Summaries of the runs in Experiment 6.

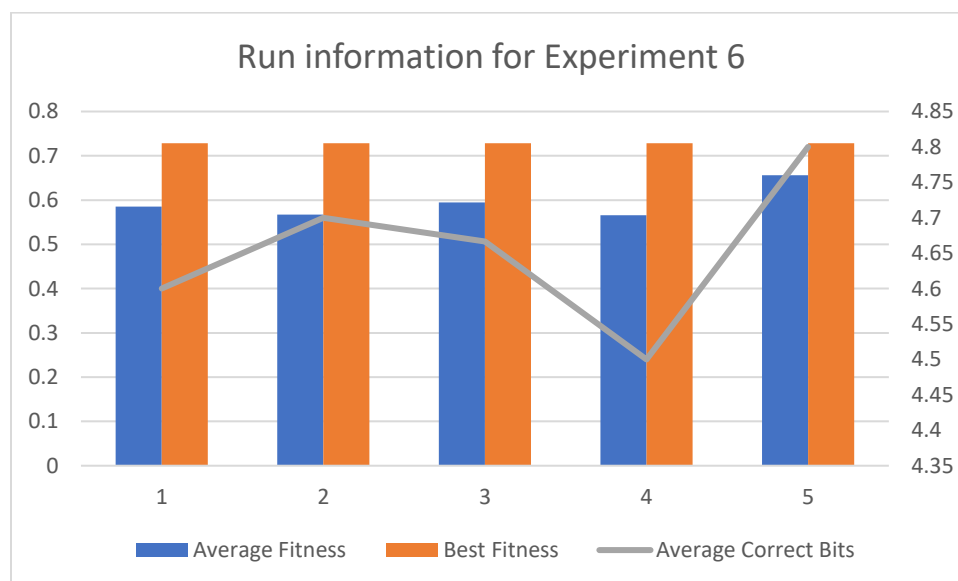
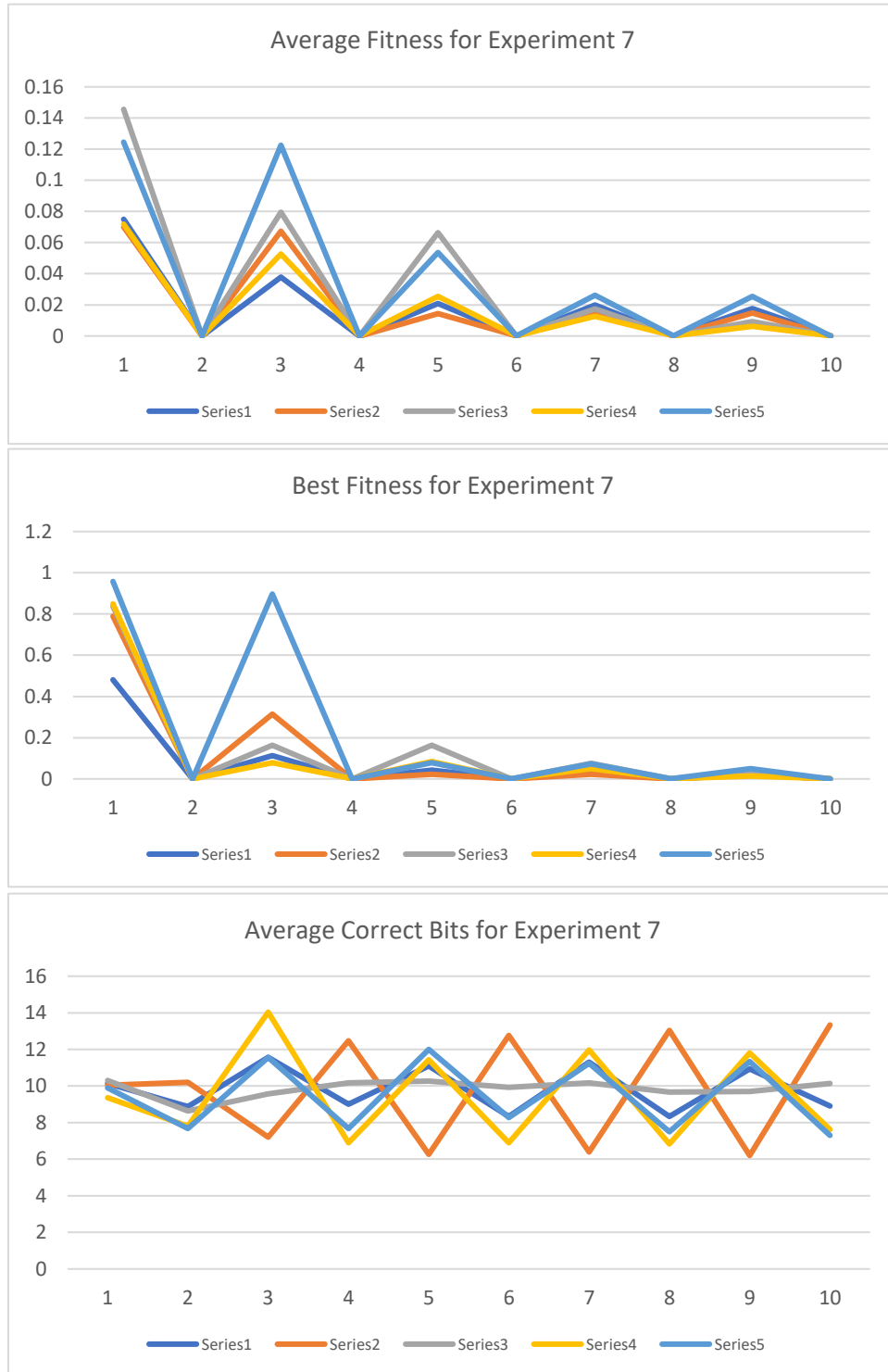


Figure 28. Graph comparing each run in Experiment 6.

6.2. EXPERIMENT 7: MAXIMUM MUTATION

Parameters: 20 Genes, 30 Individuals, 10 Generations, *100% Mutation*, 60% Crossover

6.2.1. GRAPHS



Figures 29 (top), 30 (middle), and 31 (bottom). Data for Experiment 7.

6.2.2. DISCUSSION

This simulation heavily relied on the mutation being 100%. The result is that the offspring's genes were flipped every time. Though, this also caused the individuals to advance very poorly. Table 8 shows that the average fitness of every individual after 10 generations was at almost 0. The graphs for the Average Fitness (mainly Figure 28) displays that the average fitness of each individual actually went down as time went by. The average correct bits were unstable, but never reached the 75% mark of being all "1"s. This experiment takes advantage of how the bits flip per generation if a random number is chosen that is below the probability to mutate. And since it's at 100%, it was guaranteed to flip everytime.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	1.83054E-05	3.38169E-05	8.9
2	2.55084E-05	3.59308E-05	13.33333333
3	5.05146E-05	0.000684127	10.13333333
4	9.50402E-05	0.000367108	7.633333333
5	1.41447E-06	1.45728E-06	7.3

Table 8. Summaries of the runs in Experiment 7.

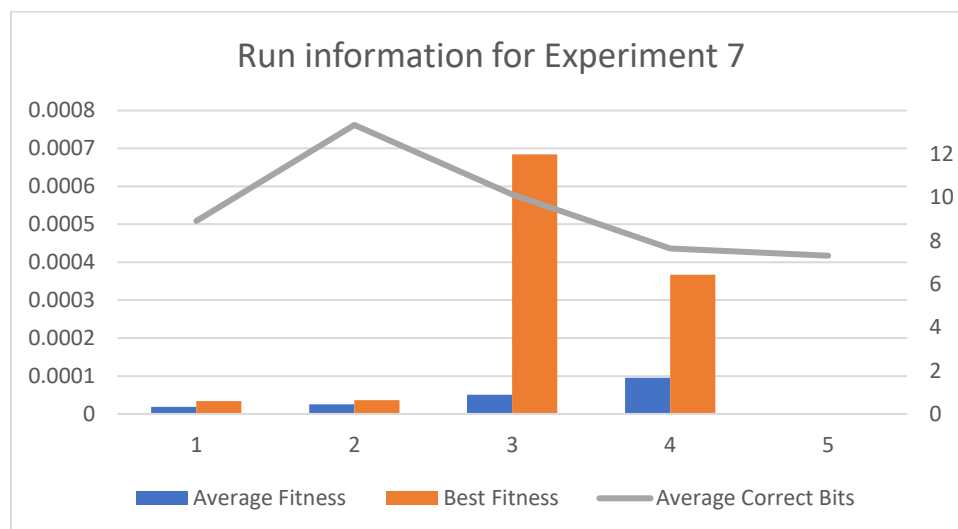


Figure 32. Graph comparing each run in Experiment 7.

7. EXTREME EXPERIMENTS

The previous experiments were tests of the parameters passed into the simulator to gather results. However, now that we are aware of the program's behavior, we can start to push the simulator to its limits and get results far beyond what was originally requested. The reasons for why this may be useful include:

- **Algorithm Consistency** – Whether the algorithm can stand the test of time and give similar results after a longer span of time.
- **Algorithm Instability** – Whether the algorithm will start producing unusual results after a longer span of time.
- **Curiosity** – Just out of curiosity of how the program will perform when tested with unusually high parameters being passed in.

More importantly, it's to understand how the algorithm works, and how we can use it to truly obtain the most fit individual. With these extreme experiments, we can set a goal of trying to find the perfect individual that has all “1”s in their bit string sequence.

The following experiments will be performed and analyzed:

#	Genes	Individuals	Generations	Mutation %	Crossover %
8	30	3000	100	0.033	0.6
9	50	3000	100	0.0001	1
10	50	3000	1000	0.0001	1

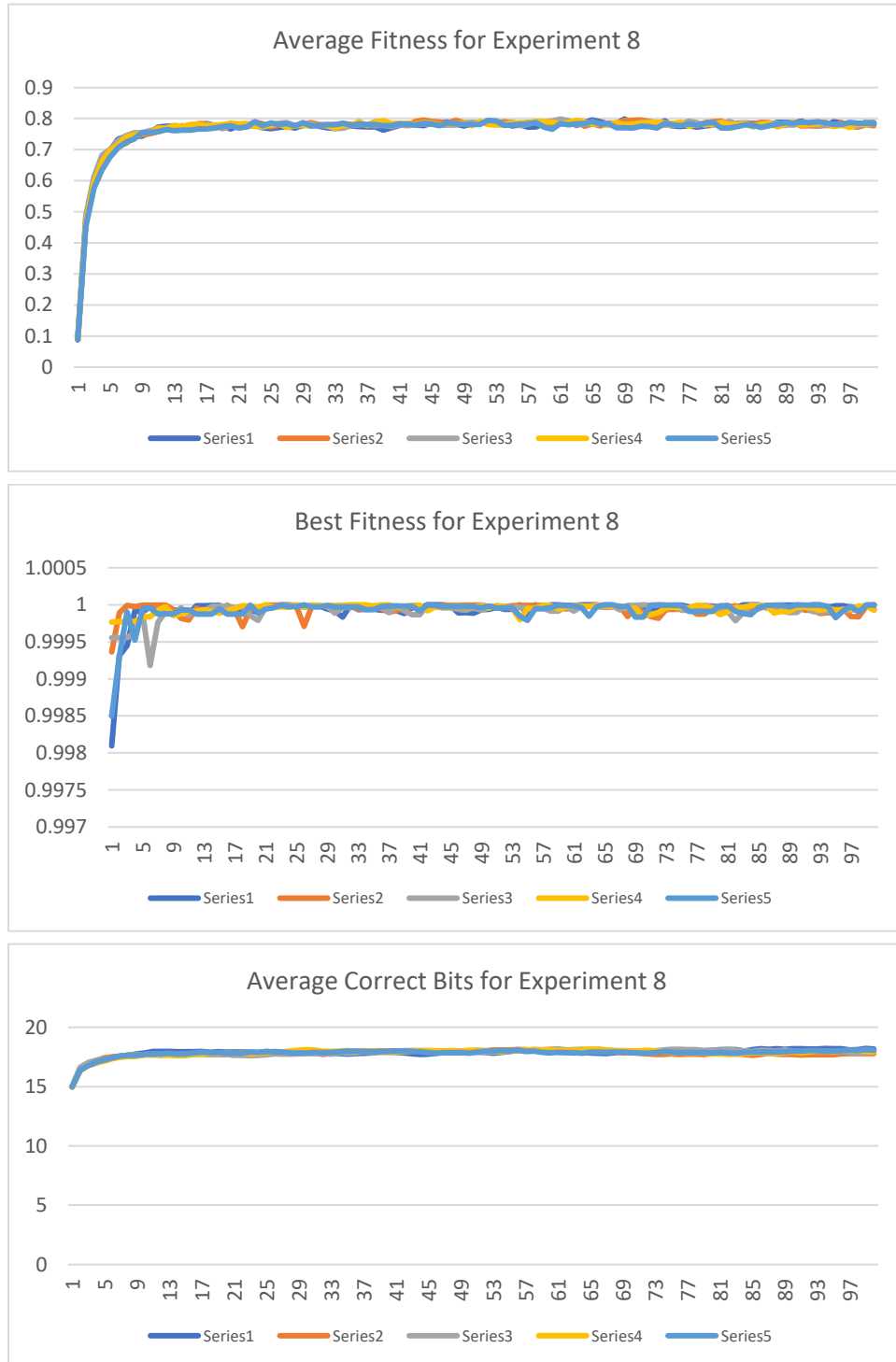
Table 9. Extreme Experiments to be run in this section.

They will be experimented in the same fashion as the previous experiments, where graphs will be posted and then discussed.

7.1. EXTREME TEST 1: 3000 PEOPLE IN 100 GENERATIONS

Parameters: 20 Genes, 3000 Individuals, 100 Generations, 0.033% Mutation, 60% Crossover

7.1.1. GRAPHS



Figures 33 (top), 34 (middle), and 35 (bottom). Data for Experiment 8.

7.1.2. DISCUSSION

This is where we start to hit the “Almost perfect” zone, where the best fitness is extremely close to being 1.0. In fact, according to the data in the Excel spreadsheet, these generations hit 0.998-0.999 extremely fast. The very first run had most of the individuals having mostly 1’s in their genes. Looking at the graphs, we can also confirm that the algorithm is stable and consistent, being able to run for a long amount of time and not give any sudden changes that result in a catastrophic underdevelopment in the future generations to come. The graph looks like an inverse logarithmic regression, though with a few bumps along the way.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.787305319	0.999940136	18.161
2	0.77782914	0.999999059	17.79133333
3	0.784700544	0.99999701	17.94233333
4	0.78243035	0.999927313	17.926
5	0.783507691	0.999995846	18.07466667

Table 10. Summaries of the runs in Experiment 8.

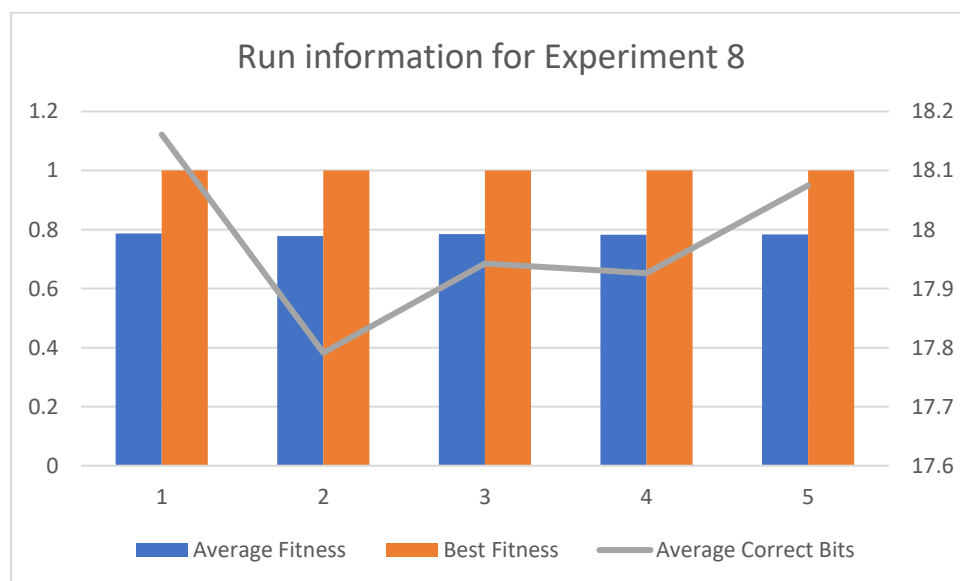
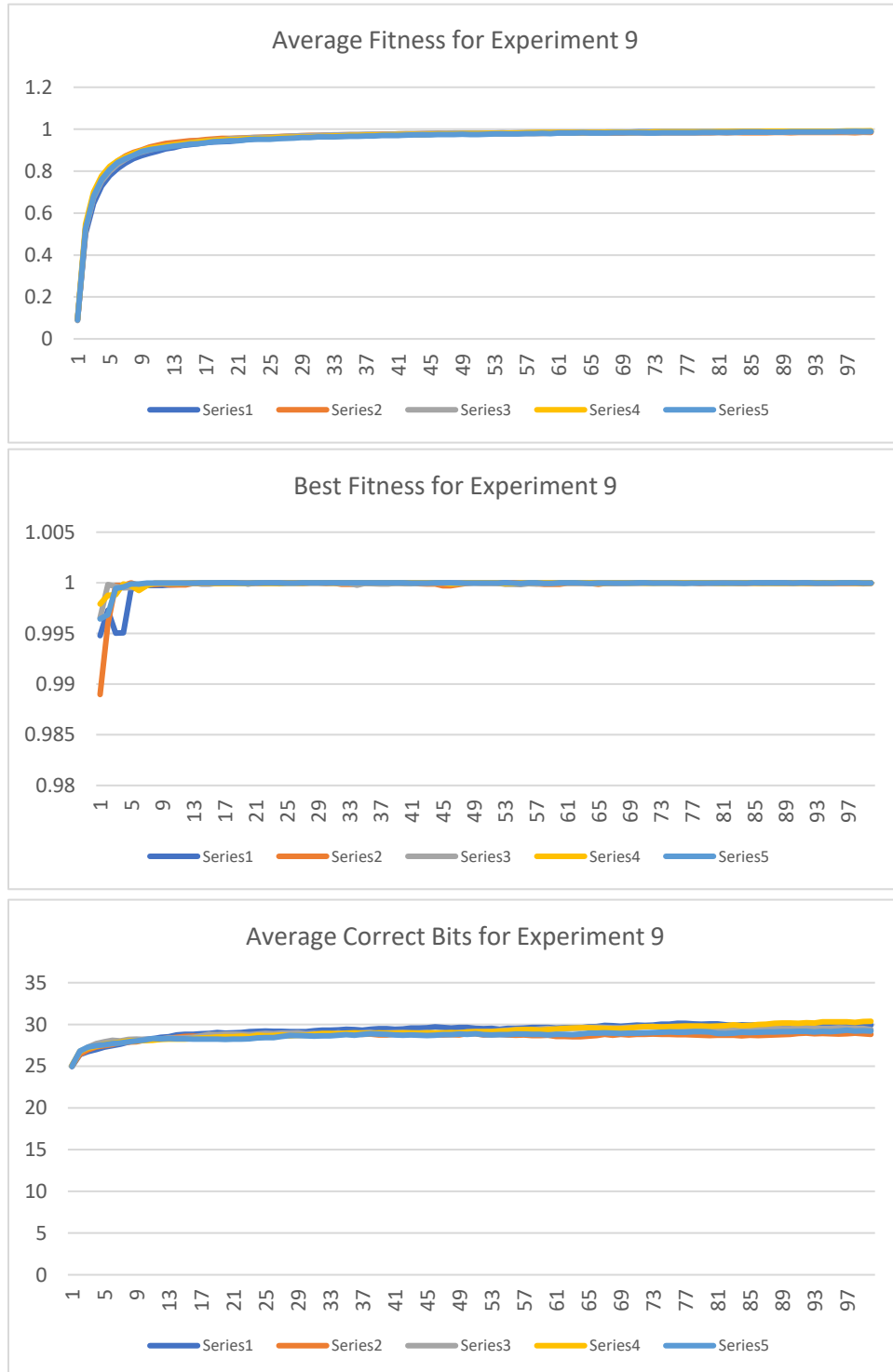


Figure 36. Graph comparing each run in Experiment 8.

7.2. EXTREME TEST 2: 50 GENES PER PERSON

Parameters: *50 Genes*, *3000 Individuals*, *100 Generations*, *0.01% Mutation*, *100% Crossover*

7.2.1. GRAPHS



Figures 37 (top), 38 (middle), and 39 (bottom). Data for Experiment 9.

7.2.2. DISCUSSION

Upping the number of genes to 50 caused the graph to become more stable and smooth. Figure 36 shows a nearly logarithmic-like graph. The effect also applies to the best fitness and average correct bit graphs, as they look much more smoother and consistent (with the only exception being the first 6-7 generations in Run 1 and Run 2 in the Best Fitness graph). Just like Experiment 8, the values in the Average Fitness and Best Fitness graphs are all approaching 1. Interestingly, despite there being 50 genes total, the average number of correct bits remains at around 30 bits per individual. It is likely because the lower-end bits are more likely to flip based on the implementation given to the simulator.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.989081719	0.999997589	29.90133333
2	0.985142151	0.999930897	28.819
3	0.988312915	0.999998784	29.386
4	0.992183064	0.999966567	30.37233333
5	0.988983649	0.999984659	29.188

Table 11. Summaries of the runs in Experiment 9.

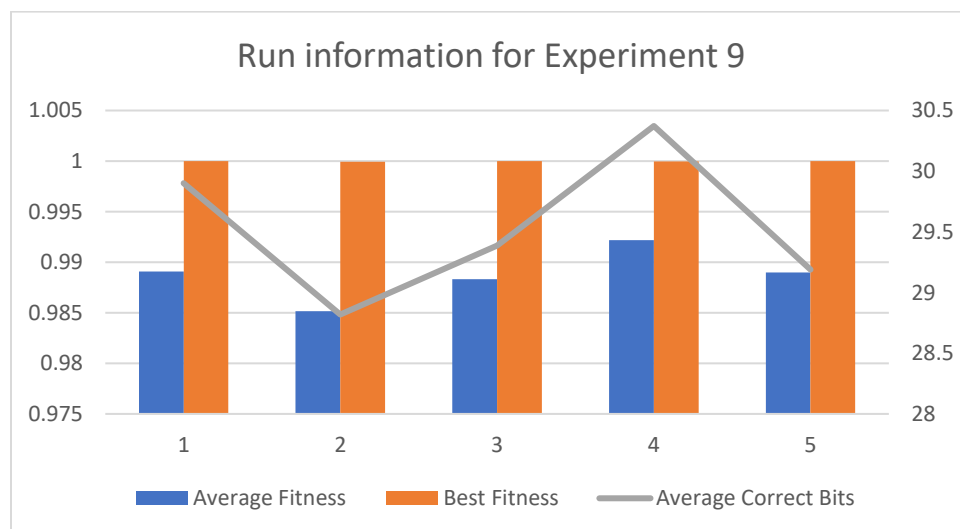
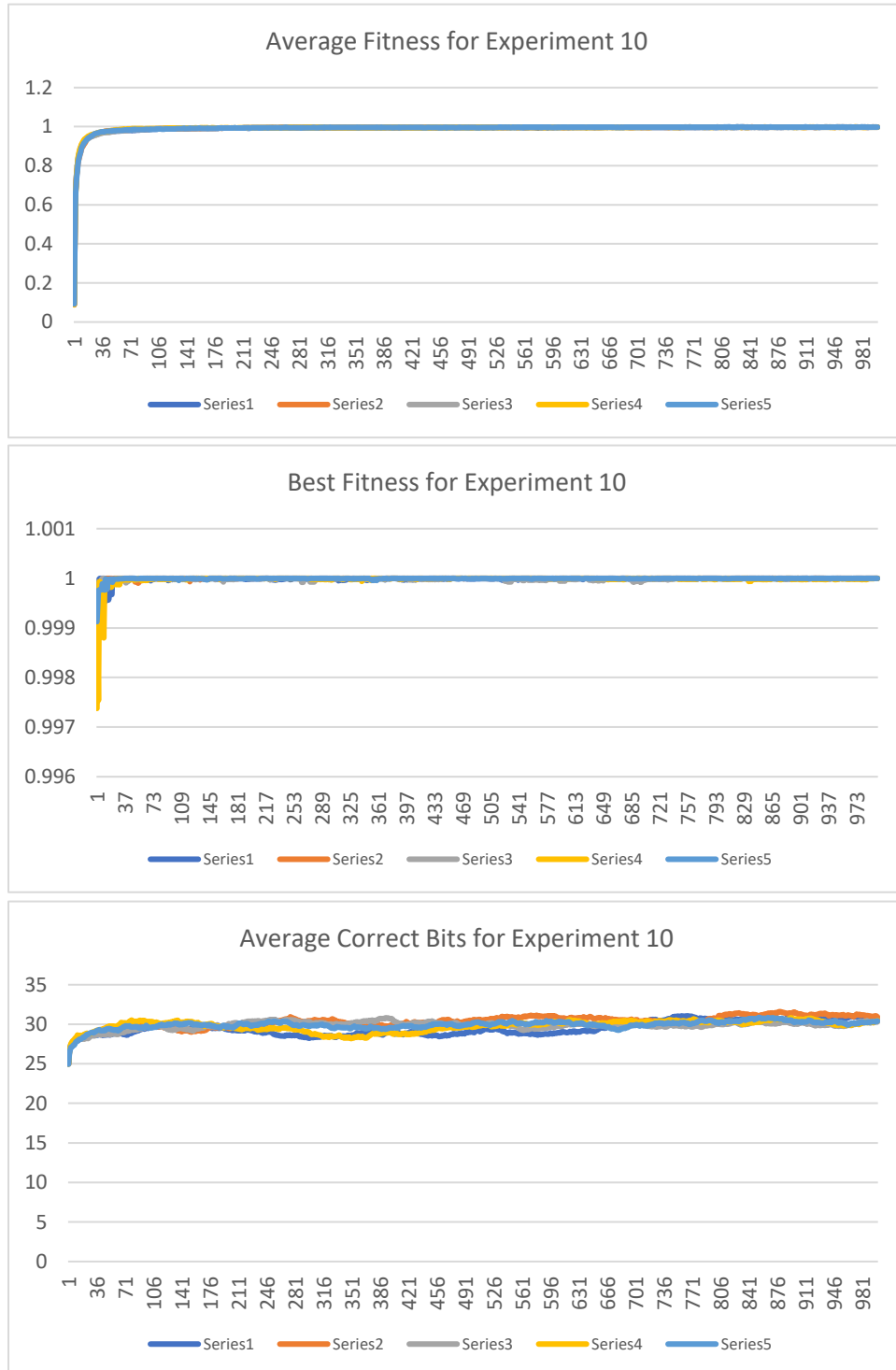


Figure 40. Graph comparing each run in Experiment 9.

7.3. EXTREME TEST 3: 1000 GENERATIONS

Parameters: 50 Genes, 3000 Individuals, 1000 Generations, 0.01% Mutation, 100% Crossover

7.3.1. GRAPHS



Figures 41 (top), 42 (middle), and 43 (bottom). Data for Experiment 10.

7.3.2. DISCUSSION

Finally, we reach the longest and most challenging test of them all. This simulation is a way to further challenge how the algorithm stands the test of time. But it is also a test of completely changing the parameters given to the program. Every parameter passed in is either a much higher value or a much lower value than the default values passed into Experiment 1. The effects are not as interesting as one would expect, unfortunately. The graph for the average correct bits is more unstable than the graph in Extreme Test 2. With a low mutation chance, the generations evolve/devolve slowly, allowing the generations to approach a 1.0 of best fitness after around 35 generations. Run 4 had a slow start to having the best fit individual, being at around 0.9975 compared to 0.9992 for the others.

Run #	Average Fitness	Best Fitness	Average Correct Bits
1	0.997242616	0.999997952	30.49666667
2	0.997881405	0.999999852	30.911
3	0.998303049	0.999998505	30.29933333
4	0.997454154	0.99999746	30.44733333
5	0.997550016	0.999997867	30.40266667

Table 12. Summaries of the runs in Experiment 10.

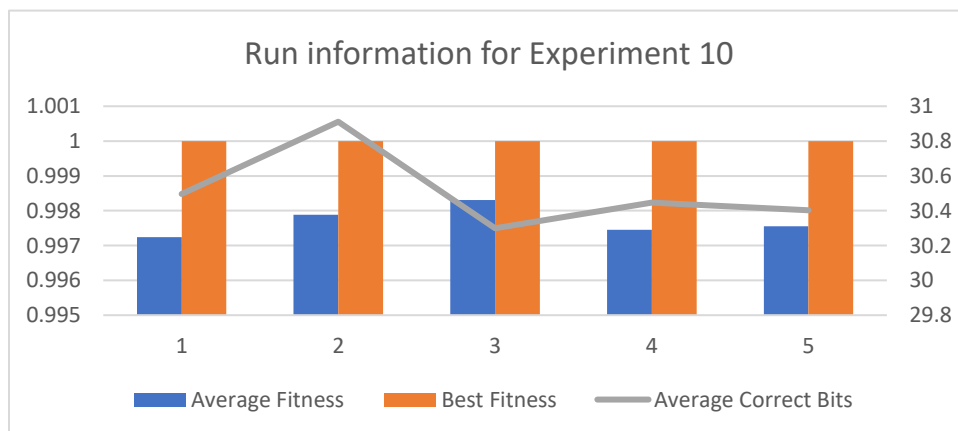


Figure 44. Graph comparing each run in Experiment 10.

8. OBSERVATIONS

Now that we have a lot of data and results, let's look at the hypotheses mentioned in Section 4.2.

8.1. HYPOTHESES RESULTS

1. What would happen if the genes were increased/decreased from the default configuration?

Genes are almost directly related to the average number of correct bits in each individual at the end of a simulation. This was discovered in Experiment 2, where the ratio of genes to the average number of correct bits can be mostly expressed with the expression below. A counter example was found in example 6, where a low number of genes throws this off.

$$ACB(g) = \frac{9g}{15}$$

Figure 45. Equation to predict **ACB** (Average Correct Bits) from **g** (genes).

2. What would happen if the population was larger or smaller?

Population, initially, didn't have much of an effect on the data. However, as shown in Experiment 4, It seems that, with the more individuals we have, it is easier for a single individual to have a very high fitness value early on in the simulation. The graphs were smoother and much closer together as well. So the results between runs were much more similar when the population was larger.

3. What would happen if the mutation probability was 100%?

Based on Experiment 7, because the genes were forced to flip each and every generation, the graphs produced had a weird zig-zag pattern. The average fitness and best fitness suffered immensely as a result of constant flipping.

4. What would happen if the crossover probability was 100%?

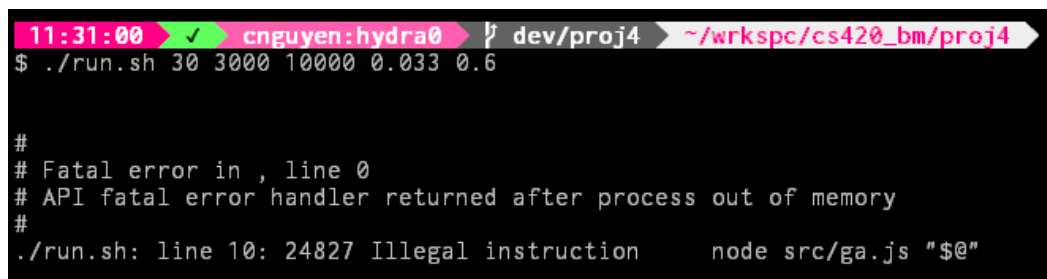
It has no major impact on the program. Based on Experiment 5, the parameters passed into the program were the exact same as in Experiment 1, with only the crossover changed. The results were similar, if not, the same.

5. Is the algorithm capable of standing a test of time? Can it last an infinite amount of generations?

Yes. Experiments 8, 9, and 10 (Extreme Tests 1, 2, and 3) demonstrated this. After running 1000 generations, it was concluded that it is stable and will either stay really close to 1.0, or approach 1.0 after a long period of time.

8.2. CAN WE GO EVEN FURTHER BEYOND?

No. The Node.js language has restricted what this simulator is capable of doing. We are allowed to go up to 64 genes per person, and run around 1500 generations maximum. It also can optionally store a family tree, keeping track of the parents of each offspring, and pointing to it as a reference. By default, it frees each generation after it creates a new generation. Regardless of this memory freeing, Node.js will crash on testing machines such as UTK's Hydra and Tesla if the parameters exceed those either due to memory or mark-sweeps.



```

11:31:00 ✓ cnguyen:hydra0 dev/proj4 ~/wrkspc/cs420_bm/proj4
$ ./run.sh 30 3000 10000 0.033 0.6

#
# Fatal error in , line 0
# API fatal error handler returned after process out of memory
#
./run.sh: line 10: 24827 Illegal instruction      node src/ga.js "$@"

```

Figure 46. Node.js crashing after attempting 10000 generations.

```

11:57:04 ✓ cnguyen:hydra0 dev/proj4 ~/wrkspc/cs420_bm/proj4
$ ./run.sh 100 3000 1000 0.0001 1

<--- Last few GCs --->
[28711:0x3dda350] 185369 ms: Mark-sweep 1452.5 (1754.9) -> 1452.5 (1754.9) MB, 704.5 / 0.0 ms  allocati
on failure scavenge might not succeed
[28711:0x3dda350] 186064 ms: Mark-sweep 1452.5 (1754.9) -> 1452.5 (1754.9) MB, 693.6 / 0.0 ms  allocati
on failure scavenge might not succeed
[28711:0x3dda350] 186741 ms: Mark-sweep 1452.5 (1754.9) -> 1452.5 (1754.9) MB, 676.3 / 0.0 ms  allocati
on failure scavenge might not succeed

<--- JS stacktrace --->
Cannot get stack trace in GC.
FATAL ERROR: MarkCompactCollector: semi-space copy, fallback in old gen Allocation failed - JavaScript he
ap out of memory
 1: node::Abort() [node]
 2: 0x102266c [node]
 3: v8::Utils::ReportOOMFailure(char const*, bool) [node]
 4: v8::internal::V8::FatalProcessOutOfMemory(char const*, bool) [node]
 5: v8::internal::EvacuateNewSpaceVisitor::Visit(v8::internal::HeapObject*, int) [node]
 6: v8::internal::FullEvacuator::RawEvacuatePage(v8::internal::Page*, long*) [node]
 7: v8::internal::Evacuator::EvacuatePage(v8::internal::Page*) [node]
    
```

Figure 47. Node.js crashing after attempting 100 genes per individual.

9. PROJECT FILES AND SUBMISSION INFORMATION

9.1. NODE.JS SOURCE CODE FILES

You can find the source code to the Node.js simulator in the “src” directory of this submission. Because Node.js is not a compiled language, but rather an interpreted language, it does not have to be compiled. It simply has to be run with “node src/ga.js”, as shown in Section 2. In the case that the Node.js executable is not found, there is a copy of it that “run.sh” uses. As long as you are on Hydra/Tesla, it will work.

9.2. SHELL SCRIPTS

The “src” directory contains shell scripts written by me to automate the experimentation and generation of data files. “do_experiments.sh” will execute the simulator 10 times with the 10 experiment parameters given in this paper. It will generate csv files that go into the “experiments” directory. The time it takes to run these experiments, despite the last 3 pushing it to its limits, was around 2 minutes.

9.3. EXCEL DOCUMENTS

In the “experiments” directory, there is an “Experiments.xlsx” file, which contains all data used in this report. It also contains all graphs and CSV data imported into a friendly format that can be read simply in Excel. The XLSX file also contains a table generator sheet, which took the experiment number, and generated a table and graph of data for the specific experiment.

9.4. CSV EXPERIMENT FILES

In the “experiments” directory, there are directories for each experiment run and mentioned in this paper, numbered from “01” to “10”. In each of these directories, there are 5 CSV files that represent a single run of the experiment at each configuration. Each experiment was run 5 times, for a total of 50 CSV files.

9.5. DOCX MASTER FILE

This document itself has a master file included in the “docs” directory under “4 - Genetic Algorithms.docx”. Of course, this file (“4 - Genetic Algorithms.pdf”) is in the root directory of this submission.