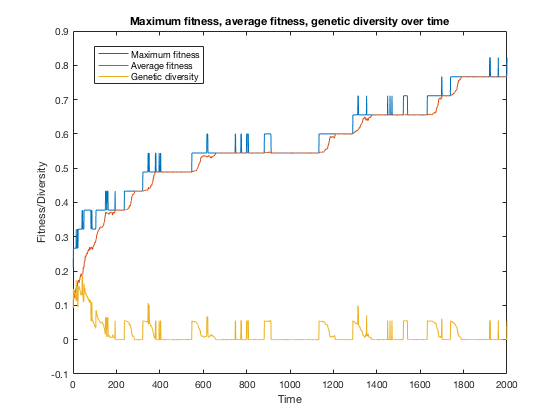
**Try to answer the questions below and motivate your choice for the most suitable value for each parameter:**

We start with the following set of parameters (random seed = 42):

* Target phrase = ‘To be or not to be’
* Target phrase length = 18
* Population size = 200
* Mutation rate = 0.01
* Max generations = 2000
* Mating factor = 10
* Breeding method = 0
* Kill factor = 0.1 (kill 10% of lowest fitness members)

The plot for maximum and average fitness and genetic diversity over generations for the above parameters:

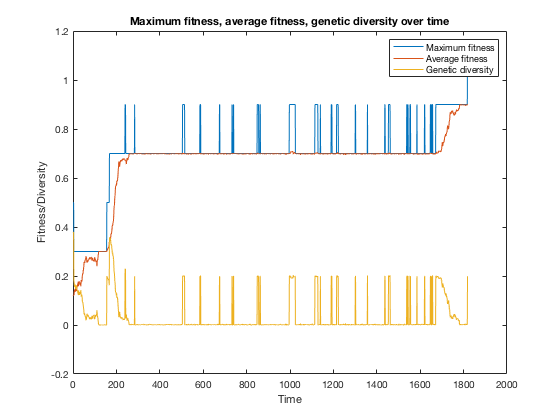


By the last generation, the maximum fitness is 0.7222 and average fitness is 0.6222 with a genetic diversity of 5.56e-2. The fittest string is “ToCbe oB not UoaOe”.

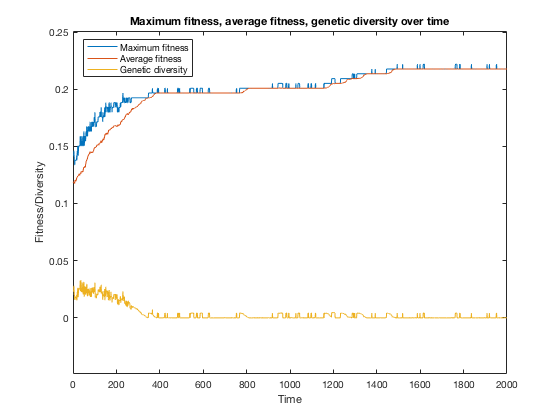
**1) Change the length of the target phrase**

- Target phrase = ‘Hello’; Target phrase length = 5

As a result, we converge to the fittest string much quicker with a short phrase. Over 1817 generations (with early stopping allowed), we obtained the max fitness of 1.0 and the fittest string is an exact match.



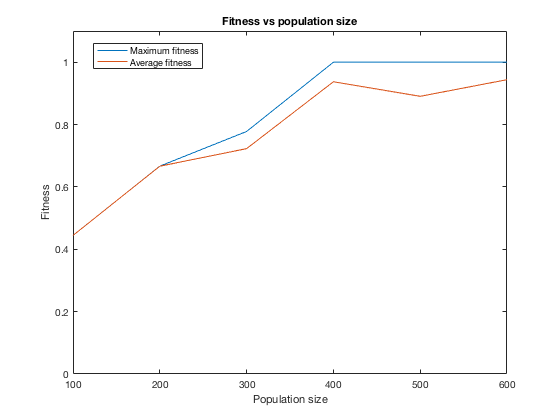
- Target phrase = ‘The universe we observe has precisely the properties we should expect if there is at bottom no design no purpose no evil and no good nothing but blind pitiless indifference DNA neither knows nor cares DNA just is And we dance to its music’; Target phrase length = 238 (Note: Punctuations are omitted to keep the DNA bits the same.)



As expected, longer phrase lengths like this one takes longer to evolve to the target phrase. After 2000 generations, we only have a max fitness of 0.22. For the remaining experiments, we will go with the phrase ‘To be or not to be’ as the target.

**2) Change the number of population members**

We test the following population size array: [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]. Fitness vs population size (at end of time) plot below shows that the higher the population the higher fitness we’ll get at the end of the generations.



**3) Change the mutation rate**

We test the following mutation rate array with varying magnitudes: [0.001, 0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2]. Fitness vs mutation rate (at end of time) plot below shows that up until the value of around 0.075, the higher the mutation rate the higher fitness we’ll get at the end of the generations. Interestingly, the peak fitness is at around mutation rate of around 0.025 for average fitness and 0.075 for max fitness, and increasing the rate from 0.075 on shows a decrease in fitness.

|  |  |  |
| --- | --- | --- |
| **Mutation rate** | **Maximum fitness at gen=2000** | **Average fitness at gen=2000** |
| **0.010** | **0.7222** | **0.6222** |
| **0.025** | **0.8889** | **0.7889** |
| **0.050** | **0.7778** | **0.6778** |
| **0.10** | **0.5556** | **0.4456** |
| **0.5** | **0.2778** | **0.1778** |
| **1.0** | **0.2222** | **0.1222** |

**4) Change the range of possible characters being considered (include numbers, etc)**

We test the ‘alphabet’ (aka range of DNA bits) to varying ASCII ranges:

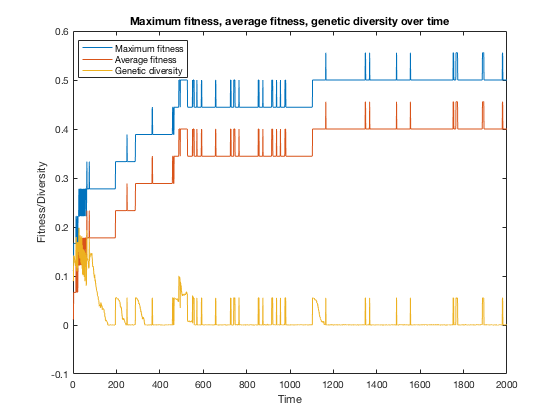
* ASCII vector [32:126] for space, special characters/punctuations, numbers, upper and lowercase characters: At the end of the generations, the max fitness is 0.4444 and the average fitness if 0.4439.
* ASCII vector [32, 48:57, 65:90, 97:122] for space, numbers, upper and lowercase characters: At the end of the generations, the max fitness is 0.6111 and the average fitness if 0.5992.
* ASCII vector [32, 65:90] for only space and uppercase characters: At the end of the generations, the max fitness is 0.3333 and the average fitness if 0.3331.
* ASCII vector [32, 97:122] for only space and uppercase characters: At the end of the generations, the max fitness is 0.8889 and the average fitness if 0.8797. Given the target phrase we have, this is the best range of possible characters since most of the target characters are lowercase and spaces.

**5) Try changing between the two breeding methods ‑ which one works better?**

Using the original set of parameters.

Method 0: See the plot at the beginning of this report. As above-mentioned, at the last generation, the maximum fitness is 0.8222 and average fitness is 0.7667 with a genetic diversity of 5.56e-2. The fittest string is “ToCbe oB not UoaOe”.

Method 1: At the last generation, the maximum fitness is 0.5 and average fitness is 0.4 with a genetic diversity of 2.11e-15. The fittest string is “Oo We Wv noTjQolbN”. This method needs more generations to evolve.



**6) Change the mating factor ‑ what benefit might we get from increasing this?**

**What’s a reasonable value for it?**

Using the original set of parameters.

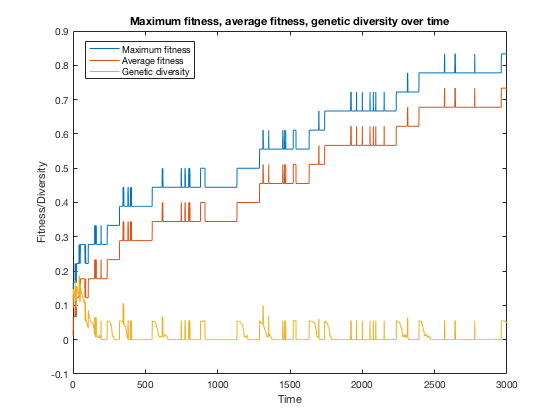
|  |  |  |
| --- | --- | --- |
| **Mating factor** | **Maximum fitness at gen=2000** | **Average fitness at gen=2000** |
| **10** | **0.7222** | **0.6222** |
| **20** | **0.7222** | **0.6222** |
| **50** | **0.7222** | **0.6222** |
| **100** | **0.7222** | **0.6222** |

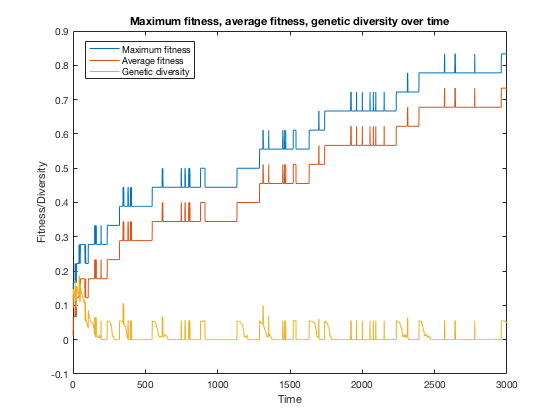
**7) Change the maximum generations ‑ what happens to fitness over time?**

Using the original set of parameters.

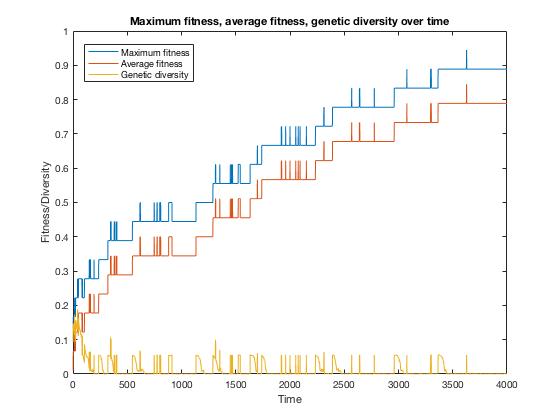
Since I started with max generations = 2000, I will increase this parameter. As expected, fitness will increase over time.

For 3000 generations, the final max fitness is 8333 and average fitness is 7333. The fittest string is ‘ToCbe or not toale’.





For 4000 generations, the final max fitness is 0.8889 and average fitness is 0.7889. The fittest string is ‘To be or not toXXe’.



For 5000 generations, as with 4000 generations, the final max fitness is 0.8889 and average fitness is 0.7889. The fittest string is ‘To be or not toXDe’.

For 10,000 generations, we have evolved the target string at generation 7210. the final max fitness is 1.0 and average fitness is 0.9. The fittest string is ‘To be or not to be’.

**8) Which function takes the longest to run? Can you improve its runtime at all?**

At first, calculateFitness() took 0.000898 seconds each time it is called because I used a for loop within for loop to compare each character in the organism and target strings to calculate the fitness score. I have replaced this with fitness\_score = sum(organism\_str == target\_str) / target\_len;.

Now,

buildPopulation() took 0.000463 seconds.

calculateFitness() took 0.000136 seconds.

buildMatingPool() took 0.000246 seconds.

breed() took 0.000478 seconds.

causeMutation() took 0.000294 seconds.

Plotting took 0.144744 seconds.

Writing data to text file took 0.029518 seconds.

After modifitications, out of the functions, buildPopulation() and breed() took longest. Overall, plotting took longest.

**To make our code much more efficient at selecting the best parents, we can force our population to favor slightly fitter members much more than everyone else. After calculating our fitness, which are numbers from 0 to 1, we can raise it to some power, let’s say 3. Now values which are close in fitness will be much easier to differentiate between.**

**Although the fitness values themselves got smaller, the relative difference between them became larger. Remembering that the mating pool normalizes fitnesses to the maximum fitness, this will now award 0.33 even more tickets in the raffle compared to 0.30 or 0.27. In your script, try raising your entire fitness vector to different values before the mating pool is built (we’ll call this the ‘exponential factor’). How high can you make this value before it stops becoming beneficial? How might you want to adjust your mating factor (from Task 1.3) after introducing this exponential factor? Write your observations in the report.**

Mating factor = 10

|  |  |  |
| --- | --- | --- |
| **Exponential**  **factor** | **Maximum fitness at gen=2000** | **Average fitness at gen=2000** |
| **3** | **0.4444** | **0.4444** |
| **4** | **0.5** | **0.5** |
| **5** | **0.3334** | **0.333** |