Brian Trybus

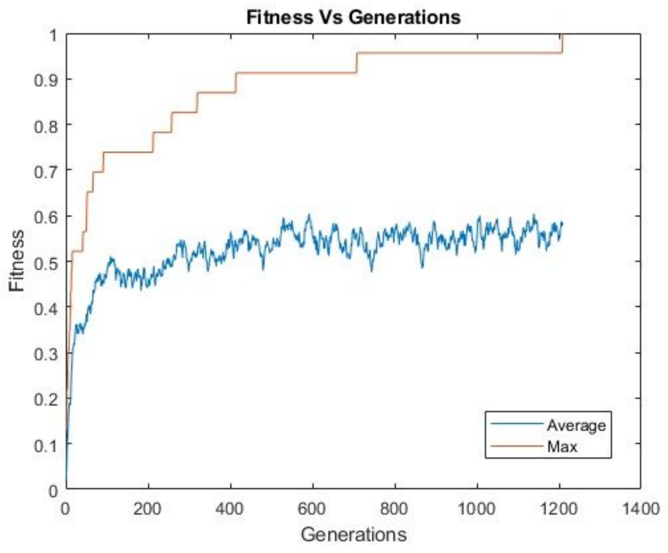
Profesor Maciej Zagrodzki

CSCI 1320

12/8/2019

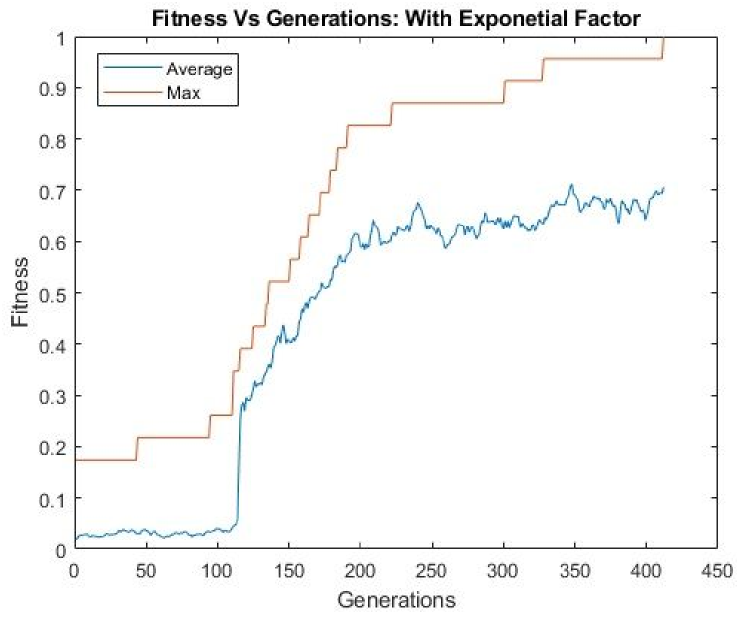
Final Project Writeup: Genetic Algorithm Results

The project objective was to create a basic genetic algorithm that is assessed on its time taken to figure out a target phrase. By using a text phrase we can easily use the individual characters as the DNA we will modify, and compare. The program starts by creating a random pool of 200 strings all the same length as the target phrase. Then all of the strings are given a fitness based off of their similarity to the target phrase, members with higher fitness are given a higher chance to breed when creating the next generation. The breeding mechanism picks 2 parents and mixes there charters to create a new string using the characters from the 2 parents. The children are then mutated at random to ensure genetic diversity at a rate of 4% chance per letter. The cycle of comparing fitness to mutating children is repeated tell the target value has been generated. To add to this, the program also combats regression keeping the champion by ensuring the highest scoring member will be passed on to the next generation. Without keeping the champion the program could regress back and reach the loop limit a majority of the time.



(Figure 1: Fitness over generations graph for base function)

At base the program performs fairly well getting the test phrase of "May your mountains rise" on an average of under 4000 generations, and taking less than a second to converge. To quantify how changes affect efficiency the program is run 100 times with each change and average time is calculated. Changes in the inputs and the methods used have major impacts on the program's performance as the most significant impact is changing the length of the target, going from 23 to 30 character target increased the run time 20 fold going from an average run time of 0.82 seconds to 20 seconds. Increasing the population size from 200 to 400 converges on the target in less generations but takes more time to process, doing this is known as overfitting where you give the program more resources than it needs so it has better performance at the cost of efficiency. By default the program only allows for letters and spaces but adding the ability to use numbers too only slightly slows down the run time bring it from .82 seconds average for 53 characters to 1 second with 63 characters. The breeding function has too different methods one where it takes a chunk of one parent and inverse of the other to create the child and another where at every letter it picks a parent to take from, both methods work but in practice the randomly alternating method proved to be fastest with .82 seconds vs 1.6 seconds. The test yielded the most impressive results was changing the fitness to exponential factor where before making the mating pool it squares all of the fitness scores to create a larger difference in scores giving higher scoring members an even higher advantage. When using the exponential factor method it improves the run time from .82 seconds average to a 0.13 second average.



(Figure 2: Fitness v Generation with Exponential Factor Fitness)

Currently the least efficient function is the fitness assessment function as it using 2 for loops to first get a fitness score, then get a fitness percentage instead of doing both steps in one loop. The function that takes the most time is the runGeneration as it contains a majority of the code, but the best way to improve its runtime would be to streamline the functions it is dependent on. Further testing on other ways to combat regression could increase the program's performance but overall it was a successful genetic algorithm.