Benjamin Berube

10/05/2021

ELEC-494

HW 2: Genetic Algorithm

Summary:

The purpose of this assignment was to draft and implement a genetic algorithm, which is a local search algorithm used popularly within the artificial intelligence field. The basic functionality of the search algorithm is as follows:

1. An initial "population" of randomized data is created by a specified length and size
2. This population is analyzed by various methods and pre-established guidelines which determine the "fitness" of the population both as a whole as well as with individual members. Each member represents a solution. The fitness of a given member essentially refers to how successfully it can solve the problem/task at hand
3. Once analyzed, the two "least fit" members of the population are removed and replaced by two members who were randomly selected and then altered by either a "crossover" or "mutation" effect. These effects are not guaranteed to take place, and are determined by a number of parameters chosen by the designer.
4. This evolutionary process of replacement is repeated until the most optimal solution is approached

Development:

The development of this algorithm was broken down into multiple functions which each controlled specific measures of the algorithm. Some (but not all) of these functions included: randomGenome(), makePopulation(), fitness(), evaluateFitness(), crossover(), mutate(), selectPair(), replace(), runGA(). In addition to these functions, the algorithm kept track of results regarding the progress of each generation. These results were stored in a log file which is saved upon the completion of the algorithm. Overall the creation of this search algorithm was a success, with all requirements being met to the best of my knowledge, and the optimal solution consistently being generated.

Observations:

After successfully developing the desired algorithm, I began to systematically alter various parameters of the program to see how they would affect the ultimate output. I began by varying the population size, starting at a size of 50 and then increasing that number stepwise by 10 until it reached 100. It was found that as the population size increased, the average number of generations needed to successfully produce the optimal solution grew larger. At *size = 50*, the typical number of generations ranged between 175 and 215. At *size = 80*, it ranged from around 250 to 300, and finally at *size = 100*, the number of generations floated around 320 on average.

Next I began to vary the *crossover rate*, beginning with 0.1 and incrementing by that amount until it reached 1.0. What was noticed is when the *crossover rate* is low (0.1-0.3 range) the algorithm takes significantly longer to produce the desired output, often exceeding 450-500 generations. When the *crossover rate* sits at around 0.6-0.8, the most consistent and efficient solutions were produced (with all other variables held static). After the parameter exceeds 0.8, there is no noticeable change in the effectiveness of the algorithm.

When altering the *mutation rate* variable on a "range" basis, it was found that regardless if the *mutation rate* is very low (0.05), somewhere in the middle (~0.5), or very high (0.8), there is no major influence on the number of generations <u>if the *crossover rate* is high.</u> When the *crossover rate* is low, a <u>high</u> *mutation rate* proved to hardly ever produce an optimal solution under 500 generations, leaving the average fitness around 13.0 and the best fitness never exceeding 17. This is likely due to the fact that mutation alone does not vary the population significantly, leading to less change each generation and hence a slower climb to the optimal solution.

While the assignment initially estimated that the optimal solution would be achieved at or before 50 generations, it was found that the optimal solution for a system containing the default parameters (listed in instructions) to consistently produce an optimal solution floating just around 250 generations, with an average fitness of around 17.8 to 18.2. While there were many instances of outliers (with my record low number of generations being 66 and my record high surpassing 500), the majority of cases remained consistent.

One further observation was that when the algorithm produced the optimal solution after a very high number of generations, the average fitness was much higher than when the optimal solution was found quickly. This feature of the algorithm is important to take into consideration, for it can help a user cater solutions towards their specific needs. If they want one solution quickly, at the disadvantage of a less-fit total population, they should seek to keep the number of generations low. However, if they wish to build a "strong" population, perhaps they should let the algorithm run for a little while longer.