Using A Genetic Algorithm For Symbolic Regression

Jordan Lynn
University of Idaho
lynn8983@vandals.uidaho.edu

Abstract

Using a Genetic Algorithm (GA) to find a solution to a mystery function on a given set of data the GA was able to come up with with interesting results using a steady state model within an acceptable iteration count of roughly 150 cycles to settle on a solution. Using parsimony pressure to help keep tree growth to sain levels helped the program operate within the hardware constraints of the machine used to run the algorithm. Results show evolution of the trees taking place.

1. Introduction

A genetic algorithm is used by first initializing a group of individuals, depending on the problem size and the computer's hardware this number is decided. For our purposes the population size is fifty, this can be fine tuned with a "meta genetic algorithm" as we discussed in class. For this assignment however tuning the numbers manually works well. A steady state model is used in this program, each iteration after a full cycle of evaluation→tournament→anti-selection is referred to as a "generation". After testing on many different data-sets a generation count of 200 was settled on. Short enough for the GA to finish in a reasonable time but still have plenty of generational room to keep evolving a solution.
The algorithm begins by seeding an array (the size of the chosen population) of individuals with random trees sized from four at the minimum size to seven at the maximum. These individuals are evaluated and given a fitness.

2. Routines

The following section defines subroutines that are used in the program to perform evaluations, tournament, crossover, selection and anti-selection, and parsimony pressure.

2.1. Evaluation

Each individual has a routine named "eval()" which after parsing the data from the csv file, compares the given value to what the tree calculates for the same value of x_i. Once this value is found the root-mean-square error (RMS error) is calculated. This value is then added to the individual's fitness field.

2.2. Tournament

Once each individual has been given a fitness a second array of pointers that reference the top ranked individuals in the population this is how selection is performed. This array is then sorted with the best individual at index zero, second best at index 1, and so on. Then the odd numbered individual's crossover function, which is a routine within the individual class is called with it's even numbered neighbor as a passed in parameter.

2.3. Crossover

An individual crosses its own tree with a passed in other individual, this is done by randomly calculating a depth to proceed to first then advancing a pointer down the tree taking a random left or right branch (as long as those movements are valid branches) and after reaching a position within the tree the same is done

for the other individual. Then the nodes at this point are switched and pointers reassigned.

2.4. Anti-Selection

Anti-Selection, or how new individuals are placed back into the population. Is based off of the fitness of the new individuals. A similar array of pointers from before is used to rank the population from worst fitness to best. As the new individuals are compared to ones in the population if the fitness is lower than those in the population they are replaced. Making sure to only to replace an individual if it has a better fitness is important because it can significantly reduce computation times.

3. Results & Conclusions

The GA seems to be performing very well. Initially there was concern over tree size, but that was easily resolved by letting the parsimony pressure contribute to the fitness. For the sake of seeing the type of function produced here is a result of running with a population of 50, 200 generations, and a tournament size of 12 (so there are 6 crossovers): $((4.6 + ((6.6 + x_9) * (6.05 - x_8)) + (4.6((6.6 + x_9) * (6.05 - x_8))))$ while this function isn't an exact answer, getting the exact answer isn't really the point with a genetic algorithm. Getting an answer that works just like the original is considered apart of the goals. If you look at you can see the best individual improving over the generations. And eventually leveling out around generation 150.

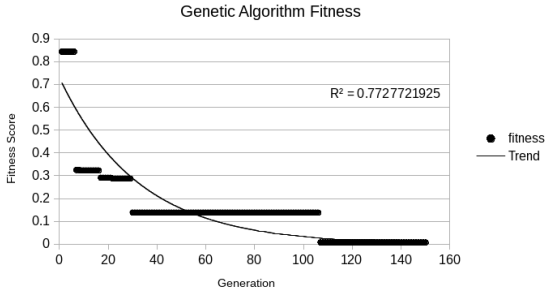

Figure 1: *graph of best individual*

The R^2 has been calculated and added to the graph to show the relation exponential decreasing of the fitness. This particular run came up with a $R^2 = 0.77$, other runs would come up with solutions even closer which can hint towards an evolutionary algorithm having diminishing rates of returns. For future projects involving a similar algorithm this should be taken into account when thinking about the run-time of the software.
The population as a whole also improved over time, this graph shown below shows all the individuals slowly improving.

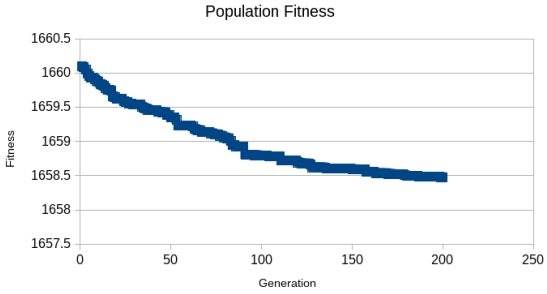

Figure 2: *graph of populations' fitness*

When comparing my algorithm to other students' algorithms in class that haven't implemented an anti-selection that only replaces when the fitness is better, there was a significant difference in computation time needed. Only allowing better individuals back into the population keeps the overall trees at a good fitness. Allowing badly crossed over or badly mutated children into the population sets the GA back and now it needs to overcome the bad mutations/crossovers. This small decision will affect what population size is chosen, mutation rates, generation counts and so on. Mainly what I'm illustrating here is that small choices when making a GA (such as anti-selection method) can have large detrimental or beneficial results. Comparing and discussing a GA with peers' GAs is important to compare/-contrast benefits and costs.