James Helm

Project 2 – Genetic Algorithm

CS 461

Fall 2021

My genetic algorithm implementation has a simple structure, relying on three object classes: “item,” “trunk,” and “norms”. Two of these classes, “item” and “trunk,” are essentially used in place of the generic “gene” and “genome” respectively. The third class, “norms,” allows portability and encapsulation for the L2 normalization distribution calculations. This way, one object can hold all the weighted distribution functionality for an entire generation, instead of performing multiple calculations on the fly with multiple variables. Each class has various setters and getters. Their functionality is self-explanatory and as such they will not be elaborated upon here. Class objects were chosen instead of bitwise functions or Boolean strings as I found they would be the easiest to keep track of to manipulate or test.

An “item” class object is comprised of a utility, weight, and a Boolean as to whether it’s packed or not. The item’s weight and utility attributes are set at the beginning from the input data text file. These values are not changed during runtime. The Boolean “packed” defaults to false unless otherwise triggered by the member method “setPacked()” or a mutation trigger. This Boolean functions as a flag for whether the item should be counted towards trunk weight and utility. The class also contains two specific methods: “setMutation()” that allows the packed Boolean to flip based on mutation trigger chances; “setPacked()” is a simple setter to flip the packed Boolean when creating the initial population to be tested.

A “trunk” class object consists of a list of 400 of the aforementioned “item” class, a double for storing total fitness of the items in the trunk, and a double for storing the total weight of the items in the trunk. It also consists of a few methods specifically for crossover handling: “setItemsPacked(),” which allows replacement of the list of items, whether during the crossover procedure or mutation, and “setWeight()” and “setFitness(),” which handle the calculations of both combined weight and fitness of the trunk respectively.

The “norms” class consists of an integer “tally,” an integer “range,” and a double “weight”. The tally serves to count the number of instances of a value in the data set within the “range” for normalization calculation and weighting. The range sets the maximum value allowed of fitness of trunk to fall in that category for tallying. The weight is for the distribution of percent probability weight for each fitness range. The only special methods here involve “incrementTally()” which, as the name would suggest, increments the population tally for that range, and “squaredTally()” which calculates the squares of the tallies for L2 normalization later.

In the end, the program runs fairly well. The initial generation of 1000 trunks doesn’t take too long but the crossover process for them can take a few minutes per generation. This is probably due to having to randomly find a trunk by a certain fitness in the trunk list, but that had to be done to minimize the chances of the first matching item in the list dominating the gene pool. I wish I could have found a way to implement multithreading from the beginning, but I don’t feel that would have made much difference given the file structure and could have led to some unintended lockups on crossover or fitness functionality. In retrospect, perhaps keeping multiple sub-lists of trunks, sorted by the fitness ranges, would have made the search much quicker. In the end, with the current implementation the number of generations before the difference became less than 1% over ten generations was almost always 20, as the genomes tended to get stuck in a rut around 12 generations for reasons I don’t understand. I even upped the number of initial item selections from 20 to 40 with no real change to that trend. With 20 items initially picked, the average fitness peaked at around 350 during my tests, while 40 items only peaked at around 440.