Hao Zhe Kok

Mr. Pettit

CS 467.01

3/30/2015

Project 2 Reflection

Project 1, which is on Genetic Algorithms (GA's), had two distinct parts. One is adapting the problem domain of the Knapsack Problem and creating a GA to attempt to solve it. The second part is for us to pick a problem domain of our choice and creating a GA to solve that. I chose the domain of Scrabble. More precisely, given a string of characters, my GA aims to return an optimal list of words that is both valuable in terms of Scrabble points, as well as being legitimate words in the English language.

The first part was fairly straightforward, with instructions given by Mr. Pettit every step of the way. Candidate solutions are presented as strings of either 1's or 0's, indicating if an item is being placed in the Knapsack. In my initial design, parents are chosen strictly by rank, meaning that the first two candidate solutions are automatically chosen to produce offspring. The offspring is then created by taking the first half of the first parent and combining it with the second half of the second parent.

This approach presented a slew of issues. For one, it was very difficult to obtain much diversity by only using the same two parents, even if the child candidate had a chance to usurp either of their positions. I very quickly realized this and made the changes necessary to test ALL the candidates equally. However, my approach to strictly taking halves of each parent also made it difficult to create homogenous populations if the initial conditions were slightly awry. The added 10% mutation chance of flipping a bit helped create variation, but added to the second problem. I thus settled to initiate cataclysmic mutation before the population is completely homogenous.

Armed with the lessons learned from the first part, I tacked my second part a little more carefully. After much deliberation, I decided to represent candidate solutions as "Attempt" objects. Each Attempt has a list of words, the list of characters currently in use and not in use, as well as the current total Scrabble score of each of the words. It also has an "addToWords" function that adds a word to its list of words given the characters in those words are in the list of unused words, it then updates the lists accordingly. Finally, each Attempt has a "generateWords" function that attempts to randomly generate strings using the list of characters still available to it.

The initial population would be generated from attempts with at least one valid word in it (Scrabble score > 0). However, after the initial population, the offspring creation process quickly made more optimal answers. This time, each candidate solution is ranked in order by descending value, with the highest candidate given the highest weight. Therefore, the likelihood of a better candidate being chosen to be a parent would be higher, yet still giving a chance for others to be selected as well.

Crossover is performed at first by setting a limit of half the length of the list of words for each parent and taking the valid words up to the limit, within the unused character constraints of course. However, I realized quickly that this gave more preferences to the first parent, and adapted my approach slightly. Instead I now alternated word selection from both parents until the limit is reached. The resulting offspring (should) now has valid words from both parents (within character constraints),

without any of the illegitimate gibberish. However, it also still has a list of unused characters. This is where the generateWords function is used, in attempt to create more valid words. I view this as a form of mutation.

This process of adding offsprings is done until the entire population is homogenous (later iterations to within 5 units to homogeneity). At this point, a cataclysm mutation event occurs. The way I handled it is to have each word in an attempt (except the top ranked one) deconstruct itself with a 40% likelihood. What this means is that the words will be removed from the list of words, and its constituent characters be added to the unused list. The generateWords function is then called to try and create more words using the (now slightly larger) unused list, in an attempt to introduce more variation to the candidate solutions. The process of creating offsprings is also repeated until homogeneity.

Once cataclysmic mutation is performed 3 times, the final population is then displayed, with rather decent results and time efficiency.

Overall I've enjoyed this project, especially after identifying the pitfalls after the first part and designing the second part more intelligently. I was especially pleased that I was able to create a working program in the short timeframe we have for the second part. I've realized that the benefit of a GA is not to provide an exact answer, but rather to provide a decently-optimal answer in a fraction of the time it would take to brute-force a perfect answer.