Moshe (Judah) Goldring
Professor Leff
Spring 2023
Design and Analysis

Arithmetic Puzzle Write-up:

In order to model a chromosome, I decided to use an ArrayList<Character> with each index corresponding to the number that character would be represented by in the possible solution.

To create a random instance of this chromosome, I created a set of all the letters involved in the puzzle and I inserted those letters in to the ArrayList<Character> at random. If the size of the set was less than 10, I added ' ' along with the letters to make the size of the ArrayList 10, since each letter had a possible value of 0-9. I then added that instance to list of lists of chromosomes.

To then create the entire initial population, I took my original chromosome that I made at random and created another ArrayList that was equivalent to the original ArrayList. I then shuffled that copy at random and added it to my list of lists of chromosmes. I continued doing this until my desired population size was reached.

The threshold for a correct solution was that the sum of the number representation of the addend and augend was exactly equivalent to the number representation of the sum. This was easily calculated using simple arithmetic -> addend + augend – sum = 0. Unless this threshold was reached a solution was not considered correct. A chromosome's fitness was calculated using the addend + augend – sum formula and a chromosome was considered more fit than other when its fitness was closer to the threshold.

For selection I selected chromosomes using either Tournament or Roulette selection until the size of the selected chromosmes was the same size as my initial population. I then performed crossover and mutation on that selected population at random depending on the crossover and mutation probabilities.

I found mutation to be the easiest element of the genetic algorithm to implement because all I had to do was get a random member of the population (a get from the list of lists using a random number generated between 0 and the size of the pop) and then shuffle that chromosome using Colections.shuffle. Since that method is completely random I found this to be a more than adequate implementation for mutation.

Although I found mutation to be quite easy, I found crossover to be much more complicated. Since the requirements stated a chromosome can't have any repeat letters and since a chromosome needed to contain all the letters it had before crossover, I couldn't just do a simple crossover. Instead, every time I did crossover, I had to make sure before children contained all their parents' letters and had no repeats. I decided to use OX crossover which essentially chose two chromosomes at random and then two random crossover points in both parent Chromosomes and copied the sub list between the two crossover points from the first parent to the first child

and from the second parent to the second child in opposite order. Then copies the remaining elements, that aren't already in the children, from the second list to the first child and the remaining elements from the first parent to the second child.

After running my Genetic Algorithm several times on many different inputs, I found that the best results were when crossover probability was between 0.6 and 0.8, mutation probability was between 0.2 and 0.4, and selection type was tournament. With a population size of 50 and max generations of 50, Roullete selection, with a crossover probability between 0.6-0.8 and mutation probability of 0.2-04, produced a correct result around 120 out of 150 times with an average of 25 generations required to produce that result. A smaller mutation and higher crossover allow the algorithm to converge on a result quicker than when crossover probability is lower, and mutation is higher. With a higher mutation more randomness is involved and even if your algorithm is converging on a specific result a higher mutation probability might affect those quality chromosomes. Whereas a higher crossover probability allows suitable candidates to mate meaning that they will producing children like them if not better allowing convergence to be reached earlier. The reason for Roulette selection performing better could be implementation based, but one reason I believe Roulette produced better results is because Tournament selection might've been converging too prematurely promoting a highly fit gene to be considered the optimum preventing the actual threshold from ever being reached. Since my algorithm selected an entire new population and then performed crossover on that newly selected population a chromosome could've emerged as a top candidate that wasn't really the correct one for the job and that chromosome could've kept being selected and crossed over because of its candidacy. Since genetic algorithms are entirely implementation specific different implementations could produce wildly different results even though they are all doing the same basic things. I.E. creating a random initial population, selecting the top individuals from that initial population at random, mating them at random sometimes, and mutating some individuals sometimes. I tried many different implementations including single point crossover for crossover, mutating just a single chromosome, and only selecting half the population and leaving over the other half. This implementation produced the best results out of them all and I think it's mainly due to the randomness of it. I found that the more random you made the genetic algorithm the stronger it performed. Because the last thing you want is premature convergence, I think the randomness along with specific techniques like crossover mutation and selection allow the algorithm to succeed. However, because of this randomness it is almost impossible to predict whether the algorithm will produce, and I found that I almost never got the same results twice.