Maximizing a 3 Variable Function

Author: Rodolfo Croes

To maximize a function is to find a combination of variables that when put into a function returns to us the highest possible value of fitness given some constraints. The traditional way of achieving this mathematically is by using gradient ascent-based optimization techniques. These techniques, as described by Sean Luke on page 13-15, start at an initial random value, make some change to the value based on the derivative of the function and once the derivative is equal to 0, we can say that we have found a maximum result for one run [1]. This then can be put in a loop with a new random start point and run again until it finds another maximum result. After this you compare the result from this run to the previous run and whichever has a higher fitness value is stored as a global maximum. The problem with this method is that we assume that the derivatives can be calculated for every function. This assumption does not apply to most real word scenarios, so this is where we must introduce stochastic optimization techniques.

Stochastic optimization, as described by Sean Luke on page 9, is the type of algorithm that uses some form of randomness to find an optimal solution to difficult problems [1]. An example of these difficult problems is to find a maximum value of a complex function f(x,y,z) given 3 variables, named x, y and z. The given constraint of this problem is that x, y and z are real numbers and that they are greater than or equal to 0 while being less than or equal to 5. To visualize this problem is very difficult because it would require a 4-dimensional plot of the 3 variables and the fitness value, and a 4-dimensional plot is hard to code so we cannot use a visual method to look for the highest peak in the constraints given. Instead, when approaching this problem, we need to implement search algorithms to find possible maximum values of fitness.

One approach to this is a Random Search. This algorithm initializes a random value for x, y and z as the base case and from this finds a fitness value for these variables. After this the algorithm generates new values for x, y and z and the respective fitness value. If the new fitness value generated is greater than the current one, it replaces the x, y, z and fitness value with the better one. This process continues for the number of repetitions initialized and returns the best solution found. Seeing that the algorithm only returns one maximum after running it once, it does not show a trend of how the algorithm generalizes to one variable combination. This can be accounted for by running the algorithm many times and sorting all the results by highest fitness value. This has more of a chance to produce a better result. However, using this algorithm is very random because it produces many random fitness values without considering what values have been produced before. So, there is no way for the algorithm to learn from its previous runs on where a change in x, y and z is better or worse. For this reason, I used this algorithm to give me a rough idea of where possible maximums can be, but not for a definitive answer for this problem.

Another algorithm that can be used for this problem is a Hill-climbing algorithm. This algorithm also initializes a random set of variables for the base case and computes the fitness value of these variable combinations. Then it generates new values for x, y, and z by taking the current best x, y and z values and changes it by a step size which is randomly generated from a normal distribution with mean 0 and standard deviation of a given step size. These new variables are then used when calculating the new fitness value. The new fitness value is then compared to the current best fitness value and the greater fitness value is kept. This process continues until the number of steps of this hill climb is completed. The problem with this method is that the algorithm without running it multiple only gets one local maximum and not a global maximum. The way we get around this is by allowing multiple runs of hill climb at different starting variables. Another problem is that the step size is constant so you cannot get more accurate once you've reached the top of a hill. This is where we can modify the hill climb and make new algorithms that change the step size, accept new solutions even though they may not be optimal, add a way to stop a run early if no change has been done in a few runs, or a combination of all these methods.

After testing these 2 algorithms, I have decided to also test some modified versions of hill-climbing algorithms, these being the Iterative Local Search (ILS) and Simulated Annealing. The Iterative Local Search, as described by S. Luke on page 28, finds a local optimum and from there explores by either making a small change to the current optimum, or make a larger change in hope of finding a better local optimum [1]. This algorithm works well to find other local maximums that are near the current local maximum thus improving the result of the search. The Simulated Annealing introduces an element of sometimes taking a worst result in hopes of finding a better result later in the run. This is done by introducing a randomly generated probability of taking a worst result. This probability of having a worst result decrease with time thus it will take more worst answers early and less worst answers later resulting in hopefully a better result. The way we control the scalability of taking a worst result is with a temperature scalar where the higher it is, the worst answers the algorithm takes.

One more Algorithm that I tested is a Genetic Algorithm which is a population-based method of searching. This algorithm, as described by S. Luke on page 36-37, creates a random population of parents, and from these they assess the fitness of each parent and pick the best parent as the base case [1]. From there they take two random parents from the population and then they make a child from them by either applying a crossover or a mutation or both to the parents that they chose. A crossover is taking some features from both parents and setting that as the child. Most times a crossover makes the child into the average of the two parents. A mutation is done by taking the child's current value and changing it slightly. This process is repeated until a new set of children are made and then these are all evaluated and if a child is better than the current best answer, then the child replaces the current answer. The entire process described above is done in one generation and this continues for many generations until all generations are completed.

Running the 5 algorithms above many times such that the algorithms would go through roughly 100,000,000 iterations each, I have gotten their best results and compared them for the algorithm that produces the highest values consistently and frequently. From the resulting histograms plotting the frequency of answers returned by each algorithm, I can see that ILS, Simulated Annealing and Hill Climb are slightly skewed towards the right thus showing me that they produce high value answers more frequently than Random Search. The Genetic Algorithm did not perform the best. However, I have decided to keep using it along with ILS and Simulated Annealing given that these 3 have better ways of combating local optimum compared to the regular hill climb.

The maximum value for f(x,y,z) after running each algorithm for many iterations is f(x,y,z) = 10.61514344608478 given x = 3.6772267273422643, y = 4.96626754668021 and z = 4.744465258887172. This value was acquired by the hill climb algorithm. After this I printed the top 1% of all of the data we collected so far and set new bounds for x, y and z. The reason for this is to shorten the search space and lessen the steps to improve the f(x,y,z) by a few decimal points. The way I determined what to set the lower bound and upper bound for each variable is by taking the lowest and highest bound from the top 50 for each variable and setting it to those. So, for the next iterations of Simulated Annealing, ILS and Genetic Algorithm, I have determined the bounds to be x = [1.1612763902255776, 4.936620950448777], y = [2.6868321554379526, 4.969924646318895] and z = [0.5139418912723328, 4.750578509460287]. I then redefined the algorithms to account for different bounds for each variable and continue running it as long as I previously did for the first attempts, with less step sizes for ILS to account for the accuracy.

After the 2nd round of iterations, I did not get a better result than the one above, thus I have tightened the bounds again. This time taking the top 10 best results so far and setting the lower bounds the same way as I have done before. The upper bound I have set them the same way but with an added 0.1 to account for overfitting the upper bound. The new bounds that were set are: x = [3.29864043162981, 4.441237536488602], y = [4.1069927204672085, 4.9762675466802095] and z = [2.083239492479558, 4.756993976743159].

This process was continued until the top 5 variables and fitness value produced were equal to the 2nd decimal place to show consistency and a high value. This convergence occurred after 4 generations of running the ILS algorithm on smaller bounds and smaller step sizes. The highest fitness value and its variables found are: f(x,y,z) = 10.68211928313492 where x = 4.430876917754807, y = 4.811065841389389 and z = 4.745662545725095. While the fitness value did not converge to the 3rd decimal place, I am content with the value that I have acquired for the time I ran the algorithms. Some improvements I would make if I had more time is to allow the algorithms to automatically adjust the bounds after a certain number of iterations and I would modify my code to save space on the number of lists of large lengths I was saving. I would have also liked it if my Simulated Annealing and my Genetic Algorithm to perform as well as the ILS did. This would require some testing with the temperature scalar, mutation probability and crossover probability.

References

[1] S. Luke, 2013, “Essentials of Metaheuristics”, Lulu, second edition, available at http://cs.gmu.edu/~sean/book/metaheuristics/