Jacob Anderson

October 10th 2019

Dr. Phillips

OLA 2 Report

A Sum of Gaussian Random Variables is a Gaussian Random Variable. A basic result from the theory of random variables is that when you sum two independent random variables. This function when graphed will provide you with a bell-shaped line graph.

For this project I was to develop two programs (greedy.py and sa.py) that to find the maximum value of the Sum of Gaussian Function. To maximize this function, I used two different algorithms, Greedy-Local Search (Hill-Climbing) and Simulated Annealing.

The code developed for both projects takes in 3 arguments at the command line for the programs to know what to do. They both take in a random number seed, an integer for the number of dimensions for the SoG function, and an integer for the number of Gaussians for the SoG function. Both programs were designed to cut off after a max of 100,000 iterations

For the greedy.py I used the Greedy Local Search is a local search that continuously moves in the direction of increasing values to find the max value of the function. Since Greedy is a local search algorithm it only looks at the its immediate neighbors and calculates the value of the neighboring states and will move in the direction of the one with the highest value. Once it has picked a direction it will keep moving in that direction with a step size of (0.01*dG(X)/dX) until the value decreases from the max or when the values stop increasing at a 1e-8 tolerance . In this algorithm, you only have to maintain the current state, so you don't have to maintain the graph.

Simulated Annealing is another search algorithm that is used for solving constrained or unconstrained bound optimization problems. The algorithm gets its name because it mimics the process of heating metal and then slowly cooling it down. In this algorithm you start at a random location and at each iteration you me to a new location. The extent of the search is determined by the probability distribution with a scale that is proportional to the Temperature. Through each iteration of the algorithm you slowly decrease Temperature so that you are less likely to pick values that decrease. As Temperature gets closer to 0 the more it acts like Greedy Local search. This stops you from getting stuck in local maximums and helps you find the global maximum. While developing the code for Simulated Annealing I was to settle on an annealing schedule used to "cool" the temperature. The Annealing schedule that I found was the best was a decrement of T = T – (T * 0.09). I found that the other schedules I used would either cool the Temperature too quickly and the algorithm wouldn't be able to allow enough bad moves to eventually find a the rise in the graph, and the other algorithms I tried wouldn’t cool the Temperature fast enough making my program take too long to find the maximum of the function

These Algorithms can both be used to solve the same kind of problems, however the main difference between the two is that in Greedy you keep climbing until you hit a decrease or a plateau, but in Simulated Annealing you allow yourself to go through the decrease or the plateau so that you can find the local or global maximum.

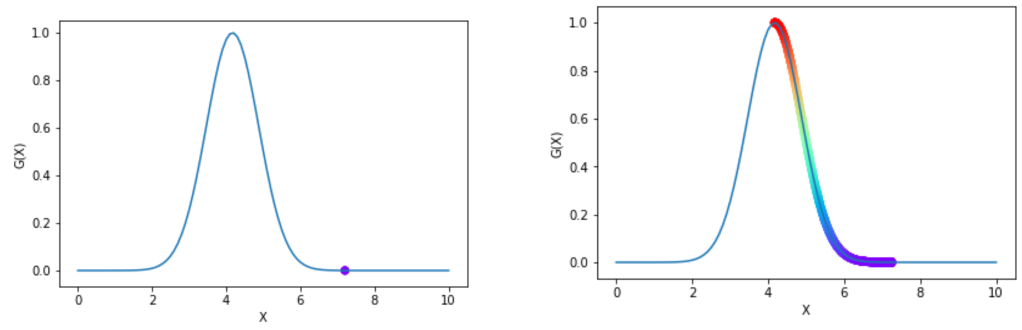

Figure 1 Results of Greedy Local Search (Left) and Results of Simulated Annealing (Right)

As you can see on the table in figure 1 the staring point for the Greedy Algorithm starts in a plateau and is unable to make it because the next location is either a slight decrease or the same value. The Simulated Annealing algorithm, however, allows you to make those bad moves and then eventually hits the rise of the graph and goes on to find the global maximum.

I found that while testing my code that the Greedy algorithm and the simulated annealing algorithm would produce similar results. However, through testing I found that the greedy algorithm struggled when the random starting point stared in a plateau on the graph. This was expected because the greedy algorithm can only move towards increasing values. Simulated annealing can avoid this problem by allowing random moves into decreasing or same values. I also found while testing that when you insert a higher number of gaussians that greedy.py will preform a faster search then my sa.py. I figure this is because there is higher number of gaussian that there is more for the sa.py to look for.