Ryan Armstrong
Dr. Phillips
CSCI 4350/5350
October 25, 2018

Open Lab 2: Local Search Algorithms Report

Both the greedy and simulated annealing local search algorithms were implemented in this lab to maximize a sum of Gaussians function, which is a function whose $f(X_0, X_1, \ldots, X_n)$ value is calculated to be the sum of any number of bell curves at a given n-dimensional point. An example of this problem in one dimension is illustrated below:

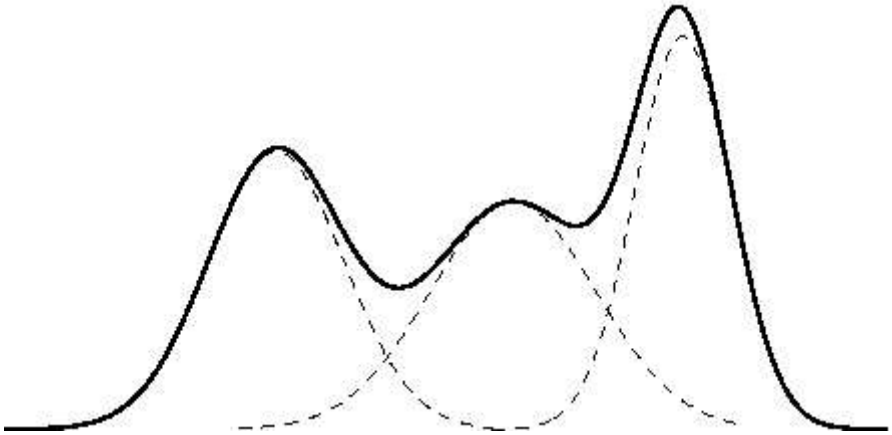

Fig. 1: A graph of three one-dimensional Gaussian distributions summed together.

Two separate programs were created to perform the searches: greedy and sa, with greedy performing the greedy local search and sa performing the simulated annealing search. Both programs take three command-line arguments: an integer seed to initialize the random number generators, an integer D representing the number of dimensions, and an integer N representing the number of Gaussian distributions to sum to create the function. Both programs also, once these three arguments are provided, generate a random N-dimensional starting position X at which the algorithms will begin searching for the global maximum. Neither algorithm guarantees that a global maximum will be found; however, both algorithms will at least find a local maximum.

In the greedy program, we keep track of two doubles: curVal, which is initialized to the value of the function at the randomly-generated starting position X and represents the current value at any subsequent position, and prevVal, which is initialized to NaN and represents the value of the function at the last position on the previous iteration. Once these values are initialized, we step toward a local or global maximum until the difference between curVal and prevVal is considered insignificant (10^{-8} tolerance) or is negative, ensuring that we are climbing to a maximum rather than moving toward a minimum. On each iteration, we calculate the partial derivatives of the function at the current point X and increase each member of X by a step size of $0.01 * (dG(X) / dX)_i$, where dG(X) / dX is a vector containing the partial derivatives of G(), the vector we're attempting to maximize, along each axis of the D-dimensional frame and $(dG(X) / dX)_i$ is the i-th component of this vector. This step size ensures that we take bigger steps given steeper slopes and smaller steps given more gradual slopes and allows us to more efficiently reach the peak of a hill.

The sa program, however, introduces a temperature value T that controls the amount of "bad", or value-decreasing, moves that are made while searching for the global maximum.

While T is high, more bad moves are accepted, whereas they would more likely be rejected with a low value of T. T starts at a high value (1.0 in the program) and decreases to 10^{-8} as iteration continues (these specific values will be explained later). The rate at which T decreases is controlled by an annealing schedule, which in this case is a trigonometric additive cooling function that can be calculated as follows: $T = T_n + 0.5(T_0 - T_n)(1 + \cos(k\pi / n))$, in which T_n is the final temperature, T_0 is the initial temperature, k is the current cycle, and n is the total number of cycles.

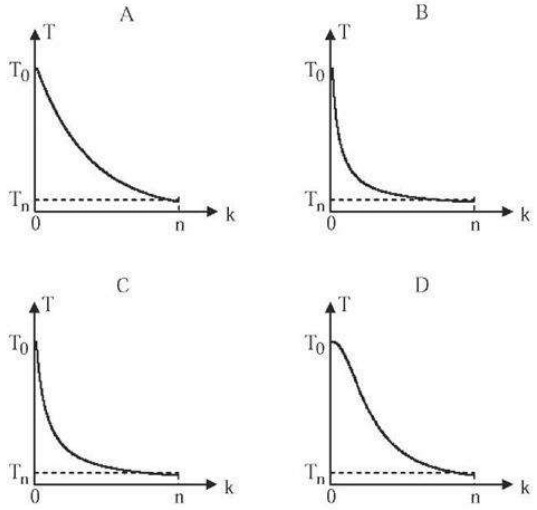

Fig. 2: Additive cooling curves. The bottom right is trigonometric.

After the initial position is generated in the sa program, we perform exactly 100,000 iterations with T decreasing each time. On every iteration we generate a uniform random step size between –0.01 and 0.01 for each element of our D-dimensional position. If the value of the function at the new position is greater than the value at the previous position, we accept the new position; however, if it isn't greater, then we calculate a probability with which to accept the new position. We then generate a random uniform number between 0.0 and 1.0, accepting the lower-valued position if the number is less than or equal to the calculated probability and rejecting it otherwise. The probability that a bad move will be accepted is calculated as follows: $e^{(G(Y) - G(X)) / T}$, where Y is the new position. This probability will always be in the range [0, 1] since G(Y) - G(X) should always ≤ 0, as $G(Y) \leq G(X)$. T has an upper-bound of 1 since we want to increase the absolute value of the exponent in the probability function as T decreases, meaning that the actual value of the exponent will approach $-\infty$, thus resulting in a probability that approaches 0 as T decreases. A lower bound of 10^{-8} is placed on the temperature so that we don't divide by 0 and receive NaN as a result when approaching the 100,000th iteration.

Each program was run with a random seed from 1 to 100 for all combinations of D = {1, 2, 3, 4, 5} and N = {5, 10, 50, 100, 500, 1000} and the number of times each algorithm found a higher value than the other was recorded. Below is a table of the compiled results (for some reason each combination of D and N was run 101 times):

	N = 5	N = 10	N = 50	N = 100	N = 500	N = 1000
D = 1	Greedy: 12 SA: 10 Ties: 79	Greedy: 5 SA: 16 Ties: 80	Greedy: 5 SA: 13 Ties: 83	Greedy: 4 SA: 6 Ties: 91	Greedy: 1 SA: 10 Ties: 90	Greedy: 0 SA: 15 Ties: 86

D = 2	Greedy: 43 SA: 19 Ties: 39	Greedy: 36 SA: 25 Ties: 40	Greedy: 22 SA: 20 Ties: 59	Greedy: 31 SA: 27 Ties: 43	Greedy: 6 SA: 17 Ties: 78	Greedy: 6 SA: 13 Ties: 82
D = 3	Greedy: 48 SA: 28 Ties: 25	Greedy: 60 SA: 14 Ties: 27	Greedy: 52 SA: 23 Ties: 26	Greedy: 59 SA: 29 Ties: 13	Greedy: 60 SA: 26 Ties: 15	Greedy: 50 SA: 32 Ties: 19
D = 4	Greedy: 38 SA: 12 Ties: 51	Greedy: 51 SA: 21 Ties: 29	Greedy: 62 SA: 36 Ties: 3	Greedy: 63 SA: 31 Ties: 7	Greedy: 59 SA: 35 Ties: 7	Greedy: 49 SA: 47 Ties: 5
D = 5	Greedy: 16 SA: 9 Ties: 76	Greedy: 27 SA: 12 Ties: 62	Greedy: 64 SA: 23 Ties: 14	Greedy: 74 SA: 24 Ties: 3	Greedy: 69 SA: 32 Ties: 0	Greedy: 67 SA: 32 Ties: 2

The simulated annealing algorithm performed better than the greedy algorithm 657 of 3,535 times, or 18.5% of total runs. The greedy algorithm outperformed the simulated annealing algorithm performed better than the simulated annealing algorithm 1,139 of 3,535 times, or 32.2% of total runs. Both algorithms performed similarly 1,739 of 3,535 times, or 49.2% of the time.

The simulated annealing algorithm performed better than the greedy algorithm when D = 1 for N = 10 to N = 1000 and when D = 2 for N = 500 and N = 1000. This is possibly due to two reasons:

1. When N is high, there is more of a chance that a bad move will place the search position on a hill with a higher local maximum value than the previous since there are more hills closer together.
2. When D is low, the simulated annealing algorithm has fewer directions in which a bad move can be made; therefore, there is a higher chance that making a bad move will place the search on a slope that leads to a higher maximum value.

As D increases, the amount of flat (or near-zero-value) surface area between Gaussians increases for the same value of N. The simulated annealing algorithm, in this case, has more of a chance to move off a hill and into this flat area, meaning that the search will kind of just skitter along the surface before the temperature decreases enough that it starts to move back toward a local maximum; however, by this time, there are not enough cycles remaining for the algorithm to reach the hill again. The greedy algorithm, on the other hand, will simply move up the hill and won't touch the flat area unless the random starting position placed it there.

If I were to hazard an educated guess as to why my simulated annealing algorithm performed worse than the greedy algorithm for many of the runs in which there was not a tie, I would have to say that my annealing schedule allowed the temperature to decrease too quickly. This, I believe, would result in the search being thrown off, due to a bad move, the hill leading to a maximum higher than that of its neighbors without allowing for an adequate number of subsequent bad moves for the search to move around the function's range. This would also explain why the simulated annealing algorithm performed worse on average in higher dimensions than the greedy algorithm. A solution to this would perhaps be to increase the amount of cycles in the simulated annealing algorithm, allowing the temperature to decrease more gradually. Alternatively, a larger random step size could potentially circumvent the problem introduced by a suboptimal annealing schedule: the simulated annealing algorithm would be able to explore more of the space before settling on a hill to climb and the algorithm

would perform better than the current implementation for high values of D where there is a large amount of flat surface area between Gaussians.

Works Cited

Joshi, Prateek. "Multimodal." Perpetual Enigma. June, 2016. https://prateekvjoshi.com/2013/06/29/gaussian-mixture-models/

"A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence)." *Whatwhenhow RSS*, what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

Unknown. Depiction of four additive cooling curves. Whatwhenhow. http://lh6.ggpht.com/_1wtadqGaaPs/TH_Q-K8CQZI/AAAAAAAAXGE/g44Pbt_3osY/s1600-h/tmp44175_thumb3.jpg