Solving Problems by Local Search

This project contains a program for both a greedy local search and a simulated annealing search. These searches are used to attempt to obtain the maximum value of the Sum of Gaussians function. The Sum of Gaussians function is set up by giving the program three command-line arguments. These are as follows: one, the random number seed, two, the number of dimensions for the Sum of Gaussians function, and three, the number of Gaussians for the Sum of Gaussians function. The program uses the seed to set up a random number generator. This is used in creating the Sum of Gaussians function with the specified number of Gaussians and with the specified number of dimensions. The function has x values between [0,10] in each dimension.

The first algorithm I worked on was the greedy search. With the greedy search, the program starts in a random location. A vector of x values within the zero to ten range marks the location, one x value for each dimension. The program then calls the "Evaluate" function to find the value of the function at the x location. The function "Gradient" is then called to determine the steepness of the slope at the point. A move is then made by adding 0.01 * (the derivative of the function with respect to each x) to each x value individually. This allows for larger steps where the slope in steeper. The "Evaluate" function is then called again to determine the value of the function at the new location. This new value is compared to the previous value. When the value of the function no longer increases within 1e-8 tolerance, the greedy search function will end. Until this qualification has been met or after 100,000 iterations are complete, the function will continue to add the appropriate step size and compare the resultant value to the previous x value. At each iteration, the program prints the location (x values) and the value of the Sum of Gaussians function. With this approach, whether or not the algorithm will find the highest point is dependent on the location of the initial random point. It is easy for the algorithm to get stuck on a peak that is not the overall highest point.

The second algorithm I worked on was the simulated annealing search. This program also starts in a random location. A vector of x values within the zero to ten range marks the location, one x value for each dimension. The program then calls the "Evaluate" function to find the value of the function at the x location. The program differs from the greedy search in finding the direction for the next move. Rather than calling the function "Gradient" to determine the steepness of the slope, a random move in the range -0.05 to 0.05 is generated. If the move increases the value of the function, then it is always taken. However, if the move is not beneficial, it is only taken with a probability of e^(((new value)-(old value))/temperature). Since the new value for the function is less than (or equal to) the old one, (new value) - (old value) will never be above zero. Therefore, the probability will always be between zero and one. Temperature is the other main variation between this search algorithm and the greedy search algorithm. The temperature starts off as a high value and decreases as the program goes on. Consequently, bad moves have a greater probability of being taken at the beginning of the search and a low probability of being taken as time goes on. For 100,000 iterations, the function will continue this process of generating a random move and accepting it based on the appropriate probability. However, if the temperature is less than or equal to zero and the

function no longer increases within 1e-8 tolerance, the algorithm will end early. At each iteration, the program prints the location (x values) and the value of the Sum of Gaussians function. With this approach, the function finding the highest point is still not guaranteed. However, the exploration (in the form of bad moves) that the high temperature at the beginning of the program allows increases the chance of finding the overall highest point.

Initially, when I was trying to settle on an annealing schedule, I tried various start temperatures and various decrement values for lowering the temperature with different values for the dimension and examined the overall final value of the function. One weakness of this trial-and-error approach is that the sheer number of possible combinations makes it very likely I skipped over a better annealing schedule in my testing. After this experimenting, I used matplotlib.pyplot to graph some 1d and 2d problems to get a better grasp of what exploration took place. I realized that not as much exploration occurred as I had expected. While I initially thought this was due to my annealing schedule, I later realize this was due to an error in my code where I had a random step within a range smaller than [-0.05, 0.05]. After fixing this error, I continued using the graphs to test different schedules. While this method did not guarantee I chose the best possible option, I was able to get a better idea of how much the program was exploring for 1d and 2d problems. My biggest problem was finding an annealing schedule that allowed for exploration but also had enough iterations left by the end to reach the top of the peak. I would settle on one for my 1d problem, then realize it did not allow enough time near the end for hill climbing for my 2d problems (Figure 2). While decreasing the amount of exploration did allow the program to reach the top of a hill, my 1d problem usually did not have enough time to explore (Figure 1). I ran differing annealing schedules a hundred times with different seed values with 3 dimensions and 50 centers to try to get a better idea of which of my options worked better in dimensions I could not visualize. My schedule that did less exploring did better 76% of the time and did better than the greedy search 66% of the time. In the end, I settled for an annealing schedule with less exploration, but which generally allowed for the algorithm to reach the top of a peak. My final schedule had temperature = temp_max * (1-(iter_count/temp_limit)) for each iteration, where temp_max = 1000000000, temp_limit = 250, and iter_count is incrementing with each iteration.

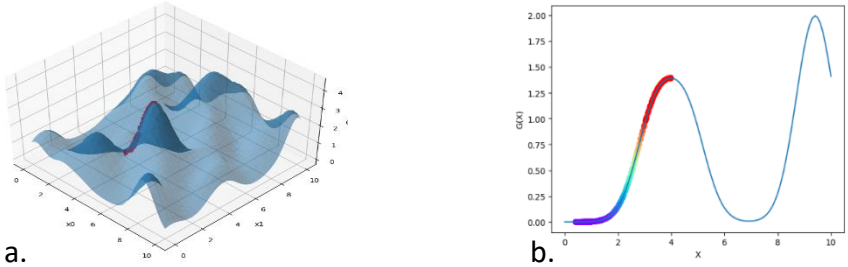

***Figure 1**: Here temperature = temp_max * (1-(iter_count/temp_limit)) for each iteration, where temp_max = 1000000000, temp_limit = 250, and iter_count is incrementing with each iteration. These are the Sum of Gaussians functions with the parameters: a) seed = 0, dimensions = 2, number of Gaussians = 50 and b) seed = 35, dimensions = 1, number of Gaussians = 4.*

***Figure 2**: Here temperature = temp_max * e^(-12*((2*iter_count)/max_iter)) for each iteration, where max_iter = 100000, temp_max = 1000000000, and iter_count is incrementing with each iteration. These are the Sum of Gaussians functions with the parameters: a) seed = 0, dimensions = 2, number of Gaussians = 50 and b) seed = 35, dimensions = 1, number of Gaussians = 4.*

After completing the two programs, I used them to perform an analysis of the strengths and weaknesses of each. Please refer to the tables below for the numbers reviewed in this paragraph. Using both my greedy search and simulated annealing programs, I searched for the maximum values of the Sum of Gaussians functions set up by using all combinations of Dimension = 1,2,3,5 = D and Number of Gaussians = 5,10,50,100 = N. I used 100 unique but corresponding seed values for each case to ensure the same problem using both algorithms. I then calculated the number of times that the simulated annealing program out-performed and/or tied the greedy search program for each condition (within 1e-8 tolerance). The general statistics for both algorithms (minimum, median, mean, maximum, and standard deviation) indicate that the algorithms generally did better at low dimensions and with a high number of Gaussians (Tables 1 and 2). When comparing the two, simulated annealing beat or tied greedy search 95% of the time in one dimension for N = 5 and N = 10 (Table 3). This number went down by six percent for N = 50 and N = 100. For 2d and 5d problems, simulated annealing did best when N = 10 (Table 3). Simulated annealing did worst in comparison to greedy search for 3d problems. In 3d, for all examples but N = 5, simulated annealing beat or tied greedy search less than 50% of the time, getting down to 26% for N = 50 (Table 3). Overall, simulated annealing outperformed or tied greedy search 1180 times out of 1600, or 73.75 percent of the time (Table 4).

Greedy Search	Minimum	Median	Mean	Maximum	Std Deviation
D = 1; N = 5	7.12575E-07	1.80697612	1.662314258	3.329364951	0.765424193
D = 1; N = 10	1.000004803	2.332922559	2.440086714	4.823644131	0.991237749
D = 1; N = 50	7.62023164	11.09565762	11.27503147	16.01982557	2.021615348
D = 1; N = 100	12.19463789	21.53209912	20.8684785	27.87137133	3.138478179
D = 2; N = 5	1.16585E-19	0.999999757	0.70916382	2.042973056	0.570288244
D = 2; N = 10	2.58333E-15	1.018815184	1.172696611	2.868328042	0.494793257
D = 2; N = 50	1.035794821	3.056632	2.970841	4.823645	0.843655
D = 2; N = 100	2.004249	4.966452	5.112496	8.923926	1.390051
D = 3; N = 5	6.02815E-27	4.26767E-07	0.322212746	1.20032146	0.472475866
D = 3; N = 10	7.03893E-22	0.999999765	0.623490848	1.405237655	0.504172424

D = 3; N = 50	8.75595E-05	1.061536355	1.242286111	2.666117758	0.432495367
D = 3; N = 100	1.001429	1.5725	1.677363	4.054828	0.594776
D = 5; N = 5	9.37016E-48	3.17709E-17	0.030000838	0.999999753	0.171446416
D = 5; N = 10	2.40057E-38	1.32391E-10	0.030004088	0.999999756	0.171445843
D = 5; N = 50	2.85298E-16	7.20751E-06	0.230130013	1.00966028	0.423155098
D = 5; N = 100	2.53378E-13	0.000152065	0.492955888	1.128792588	0.505641838

***Table 1**: Greedy search statistics.*

SA Search	Minimum	Median	Mean	Maximum	Std Deviation
D = 1; N = 5	1	1.834925411	1.754932996	3.556604949	0.700463431
D = 1; N = 10	1.000005045	2.438657896	2.481909757	4.823644211	1.028473314
D = 1; N = 50	6.934809638	11.09565766	11.22698262	16.01982562	2.050630828
D = 1; N = 100	9.79335576	21.5635983	20.82994041	27.87137134	3.283583412
D = 2; N = 5	4.61076E-18	1.0000001	0.858040214	2.042973323	0.505213796
D = 2; N = 10	6.42574E-14	1.051085233	1.221973188	2.868327964	0.439925532
D = 2; N = 50	1.035795081	3.07738917	2.981549722	4.823644907	0.819521645
D = 2; N = 100	2.004248656	4.951323059	5.104222778	8.923926143	1.375628864
D = 3; N = 5	5.17564E-32	0.999996935	0.544763975	1.236656997	0.506250335
D = 3; N = 10	6.83086E-22	0.999999812	0.821604794	1.653050532	0.421643019
D = 3; N = 50	1.000022732	1.06688831	1.243814562	2.512396766	0.347079466
D = 3; N = 100	1.001428085	1.653944234	1.714610027	4.054823727	0.615625128
D = 5; N = 5	4.30232E-46	3.62834E-17	0.099996366	1.000014428	0.301500343
D = 5; N = 10	4.48108E-36	2.62056E-11	0.169992247	0.999999696	0.377507411
D = 5; N = 50	1.80932E-17	0.9999604	0.66438806	1.276590473	0.480104797
D = 5; N = 100	2.6806E-16	1.000025602	0.869237892	1.409494754	0.35524812

***Table 2**: Simulated annealing statistics.*

Comparison	D = 1 N = 5	D = 1 N = 10	D = 1 N = 50	D = 1 N = 100	D = 2 N = 5	D = 2 N = 10	D = 2 N = 50	D = 2 N = 100
Outperformed/Tied	95	95	89	89	87	94	87	81
Comparison	D = 3 N = 5	D = 3 N = 10	D = 3 N = 50	D = 3 N = 100	D = 5 N = 5	D = 5 N = 10	D = 5 N = 50	D = 5 N = 100
Outperformed/Tied	63	43	26	34	88	92	66	51

***Table 3**: Number of times simulated annealing beat or tied greedy search for each combination.*

Outperformed/Tied	Total Runs	Percentage
1180	1600	73.75

***Table 4**: "Outperformed/Tied" is the number of times simulated annealing beat or tied greedy search, and "Percentage" is the overall percentage of times simulated annealing beat or tied greedy search.*