Gordon Gao, Thomas Lourenco, Kevin Ouellette, Monika Danielewicz

Group 16

**CS4341 Assignment 2**

**Simulated Annealing:**

Our temperature schedules used for simulated annealing is see in table 1.

|  |  |  |
| --- | --- | --- |
| **Schedule Variant** | **Equation** | **Starting Heat** |
| Linear Decay Schedule | H = H - 0.01 | H = time\_limit |
| Time Based Decay Schedule | H = (1 - Elapsed Time/ Starting Heat) | H = 1 |
| Constant heat Schedule | H = 1 | H = 1 |

**Table 1**. Equations and starting heat values used for our testing. All testing was done with a sample size of 999.

The heat values from our trials are then input into a probability function that produces a value between one and zero. The probability function used is as follows:

e^((newScore - topScore - 1)/ heat)

In this case newScore is the score of the consider board state, topScore is the current state of the board, and heat is the heat value determined by our cooldown function.

Our probability function was generated by comparing the the difference in scores between the current state and the considered move. The worst the move, the higher the heat needs to be in order to make the move. A higher heat would lower the requirement for the move to be accepted.

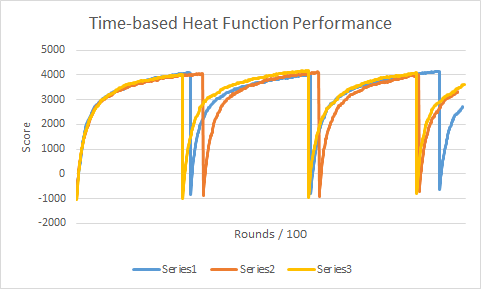
From this point we began testing each of our heat schedules with the provided probability function. We tested three different temperature schedules on a text board of size 999 under tune.txt with a time limit of 30 seconds for each trial. For this specific test we did 3 separate runs of the functions to ensure consistency.

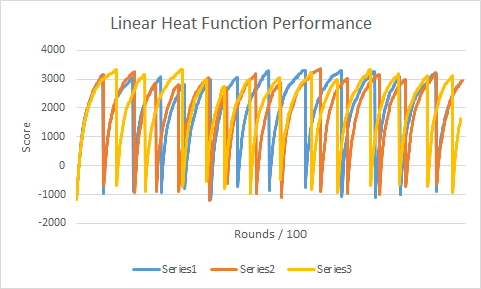
|  |  |  |  |
| --- | --- | --- | --- |
| **Temperature Schedules** | **Highest Score** | **Iterations** | **Rounds (Moves Considered)** |
| Constant Heat Schedule | 1st - 4365  2nd - 4285  3rd - 4305 | 1  2  1 | 76569  77798  77787 |
| Time Decay Schedule | 1st - 4158  2nd - 4134  3rd - 4188 | 3  3  3 | 79061  78058  79357 |
| Linear Decay Schedule | 1st - 3334  2nd - 3362  3rd - 3350 | 12  13  13 | 78250  79055  78684 |

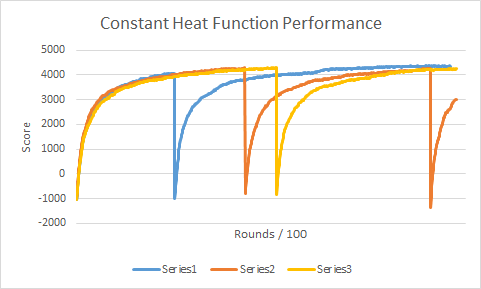
**Table 2.** 3 Trial runs showing scores, iterations (number of resets due to sideways/backwards moves) and all moves made for our three temperature schedules.

We made three graphs based off of the current score of the board throughout the cycle of the algorithm. Since the number of moves was so large, we only show the score at every 100 moves.

From this data we came to a conclusion that a time-based heat function or a constant heat would be the best for the list of three. The linear decay model unfortunately re-iterates too frequently and never reaches much higher of a score, as compared to the other two.



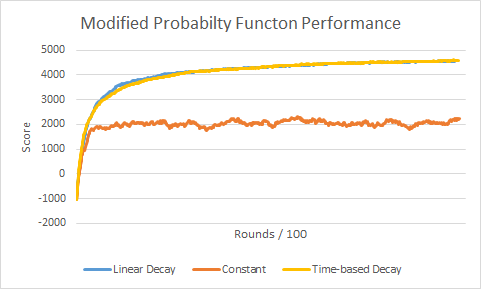




We also tested results with a changed probability function. With a changed probability function of e^((newScore - topScore) / heat) sideways moves are always taken, even if the the new calculated score is equal to the top score kept.

|  |  |  |  |
| --- | --- | --- | --- |
| **Temperature Schedules** | **Highest Score** | **Iterations** | **Rounds (Moves Considered)** |
| Constant Heat Schedule | 2255 | 0 | 78331 |
| Time Based Decay Schedule | 4608 | 0 | 78165 |
| Linear Decay Schedule | 4578 | 0 | 77383 |

**Table 3.** Table of revised probability functions. Shows the reduced amount of iterations and change in scores allowing moves that maintain the score.



For our submission we have decided to go with time based decay not allowing sideways moves. We did not choose to go with constant heat function because there is the chance you can head away from an optimal state. While it does perform better here, the chance to make a worse move should be lessened over time. A time based decay is more in the spirit of simulated annealing, by starting off really “hot” and then cooling down over time. Additionally, we opted to not use e^((newScore - topScore) / heat) as our probability function because in some cases it may get stuck in a local maxima, as it always takes sideways moves.

**Genetic Algorithm**

Tuning the Algorithm

The algorithm was tuned using the same tune.txt file used for simulated annealing, in which the input array was 999 numbers long. The data below was gathered to determine which population, level of elitism, and level of mutation would make the algorithm generate the highest score.

Population: Run with elitism = 10%, mutation = 1%, runtime = 10 seconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population | 50 | 100 | 150 | 200 | 250 | 500 | 1000 |
| Avg Score | -38 | 390 | 170 | 145 | 72 | -162 | -343 |

**Table 4.**This data shows that the ideal population for this genetic algorithm is 150. It is believed that a smaller population is more ideal because each generation can be generated faster, allowing for more generations to be created. This allows the algorithm to converge more effectively on the hypothetical best score. When a large population is used, fewer generations can be created, meaning the algorithm has to rely more on random chance to get higher scores.

Elitism: Run with population = 150, mutation = 1%, runtime = 10 seconds

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Elitism (%) | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 40 | 50 |
| Avg Score | 39 | 206 | 182 | 269 | 222 | 200 | 225 | 256 | 179 |

**Table 5.**This data shows that the ideal percentage of elitism is 15%. This decision was made despite 40% resulting in a similar average because 15% resulted in the highest single score, which was 369. This relatively small level of elitism makes sense. Though less elitism requires more time to converge to its maximum, the ceiling to which it can converge will often be higher because there will be more randomness in the algorithm.

Mutation: Run with population = 100, elitism = 10%

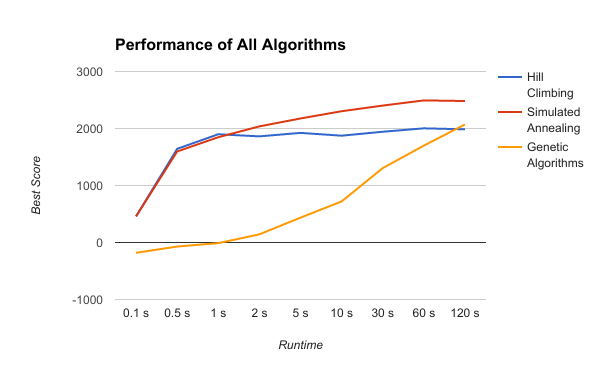
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mutation | 70% | 30% | 10% | 2.5% | 0% |
| Avg Score for 10s | 72.6 | 104.3 | 72 | 106.6 | 73.6 |
| Avg Score for 60s | 1024.3 | 1033.3 | 772 | 625.6 | 67 |

**Table 6.** This data shows that of the tested percentages, 30% mutation of the population is ideal. However, for shorter periods of time, 2.5% mutation is the best. This makes sense, as with less generations, there is a small pool of best solutions, which mutation just destroys. However, for a longer run time, a low level of mutation results in stagnation of the population and inhibits the seeking of better solutions. A mutation percentage that is way too high (70%) changes too many potentially good solutions, while a 0% mutation predictably does the worst.

**Testing:**

Below is a table and graph comparing the 3 types of searching used in optimize.py.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0.1 s** | **0.5 s** | **1 s** | **2 s** | **5 s** | **10 s** | **30 s** | **60 s** | **120 s** |
| **Hill Climbing** | 459 | 1643 | 1899 | 1862 | 1923 | 1874 | 1943 | 2003 | 1986 |
| **Simulated Annealing** | 458 | 1597 | 1847 | 2037 | 2176 | 2303 | 2402 | 2493 | 2482 |
| **Genetic Algorithms** | -181 | -71 | -11 | 143 | 436 | 720 | 1305 | 1700 | 2070 |



For the dataset provided in test.txt simulated annealing proved to be the best search algorithm. It is beaten by hill climbing in the first second of search, likely due to the larger amount of iterations and being able to complete them. From two seconds on, however, simulated annealing pulls ahead as it gives each iteration more time based on our time decay heat function. At no point in the range of 0.1s to 60s is the genetic algorithm better than either of the other two. Some members of the team believe that, given enough time, the genetic algorithm may generate better results than either of the other algorithms because the the genetic algorithm did not approach a noticeable asymptote when run on test.txt. This theory was tested for a 2 minute runtime, and the results are shown in the above table. This shows that the genetic algorithm was able to surpass the hillclimbing algorithm given enough time.