Tucker Haydon
Arthur Lockman
John Lomi
Jon Sawin

AI Project 2

3rd February 2017

OVERVIEW

For this assignment, we created a hill climbing, simulated annealing, and genetic algorithm. We tuned each algorithm when appropriate and tasked them with sorting numbers into 3 bins with the hope of maximizing the total score. The algorithms were tuned using a single tuning file then tested on another that had not been run before. The results and analysis of each of these algorithms can be found in their respective sections below.

Hill Climbing

The hill climbing algorithm that we designed looks at 100 possible moves and chooses the best one. To do this, a best new state is tracked as the algorithm generates 100 random swaps from the current state. If the algorithm finds that there is no move out of the 100 swaps generated that will increase the score, the bins are reshuffled and the search restarts. This process repeats until the time expires. Once the time expires, the best state found so far is returned and printed.

Simulated Annealing

We implemented simulated annealing as an extension of hill climbing. Whereas hill climbing randomly generates 100 possible paths and always chooses the best one, simulated annealing generates random paths and randomly chooses one. It randomly chooses one based on a probability distribution given by a function of the path's score:

$$\text{Pr} = e^{-max(0,\, best - new)/temperature}$$

Where *best* is the current best score, *new* is the score of the newly generated path, and *temperature* is a scaling factor. If a new path is discovered that is better than the current one, this probability function is 1 and the algorithm will take this path. If the new path is not better than the current path, it takes the path with some probability.

The temperature is a scaling factor that pushes the algorithm towards towards greedy decisions as time begins to run out. At the beginning of the algorithm, the temperature is set a large number (1 million in our case) and it produces probabilities near 1. Every random swap is given a near equal chance. As the algorithm progresses, the temperature is reduced towards 0 following a temperature schedule function. As the temperature lowers, the algorithm becomes more greedy and prefers swaps that result in higher scores.

Tuning Annealing

For our simulated annealing implementation, we had one hyperparameter: the temperature schedule function. This function determines how the temperature changes over time and has two requirements: a) The initial temperature must be the maximum temperature b) The final temperature must be 0 (or very close to it).

We experimented with 3 possible temperature schedules:

Linear decrease: $T = max\ temp * (time\ left / total\ time)$

Quadratic decrease: $T = max\ temp / (1 + 0.1 * (current\ time)^2)$

Geometric decrease: $T = current\ temp * 0.97$

To choose the best one, we ran the algorithm over a number of tuning sets and averaged (mean) the results. We created three tuning sets of size 600, 3000, and 9999 and five unique tuning files in each set. We used multiple data set sizes to see if the size of the data affected the efficiency of each algorithm; we used multiple files to prevent data overfitting (much like 5-fold validation tests for machine learning). The results of these tests are charted below.

	Lin	**Quad**	**Geo**
600	2384.2	2716	2719
3000	3774.8	4949.2	5083.2
9999	-6778.6	-5838.4	-5891

All implementations were run for 10 seconds each. Both quadratic and geometric perform significantly better than linear. In some cases, quadratic performed better than geometric and in other cases it was the opposite. We also ran the tuning tests on the 9999 data size for 30 seconds to see if the time affected the relative asymptotic behavior. Results shown below.

	Lin	**Quad**	**Geo**
999	-112.6	1081	1071.4

The relative asymptotic behavior did not appear to change with the increased running time; linear still performed significantly worse than the quadratic and geometric temperature schedules. Since the quadratic temperature schedule on average tended to squeak out a bit more performance than the geometric schedule on higher cardinality data sets, we chose that to be our default/best choice.

Genetic Algorithm

A genetic algorithm is very different from hill climbing or simulated annealing and because of this, a different approach was taken. To initialize the algorithm, a population of genomes to start with is generated by shuffling the bins and their contents. This population size is determined by the population size parameter. Then, the list of genomes is sorted by score and a percentage of them are kept for the next population. This percentage depends on the elitism parameter. The rest of the spots in this new population are filled by semi-randomly breeding two of the non-elites.

The genomes are given a weight, determined by their ranking in the list of all genomes, and this weight is used as a probability that the genome will be chosen to be bred. This process repeats until two genomes are found. These two genomes become parents and are bred to have two children which will be added to the new population.

The breeding process consists of choosing a cut point, splitting the parents at that point, swapping the right portions of the cut, and correcting for missing and extra values. The cut point is a randomly chosen even index. We decided that an even cut point would be best because it ensures that there will always be pairs of missing/extra values which is easier to correct for than an odd number. The correction process compares the number of instances of each number from the original right portion and the new right portion. Essentially, any values with extras are turned into values that are missing.

After the children are generated, mutation has a chance to occur. There is a probability that each genome will be mutated which is determined by the mutation rate parameter. If a genome gets mutated, a random swap in the genome occurs.

This process repeats until a new population is created with the elites and children. This new population is then run through the entire algorithm again and the top scoring genome is returned once the time runs out.

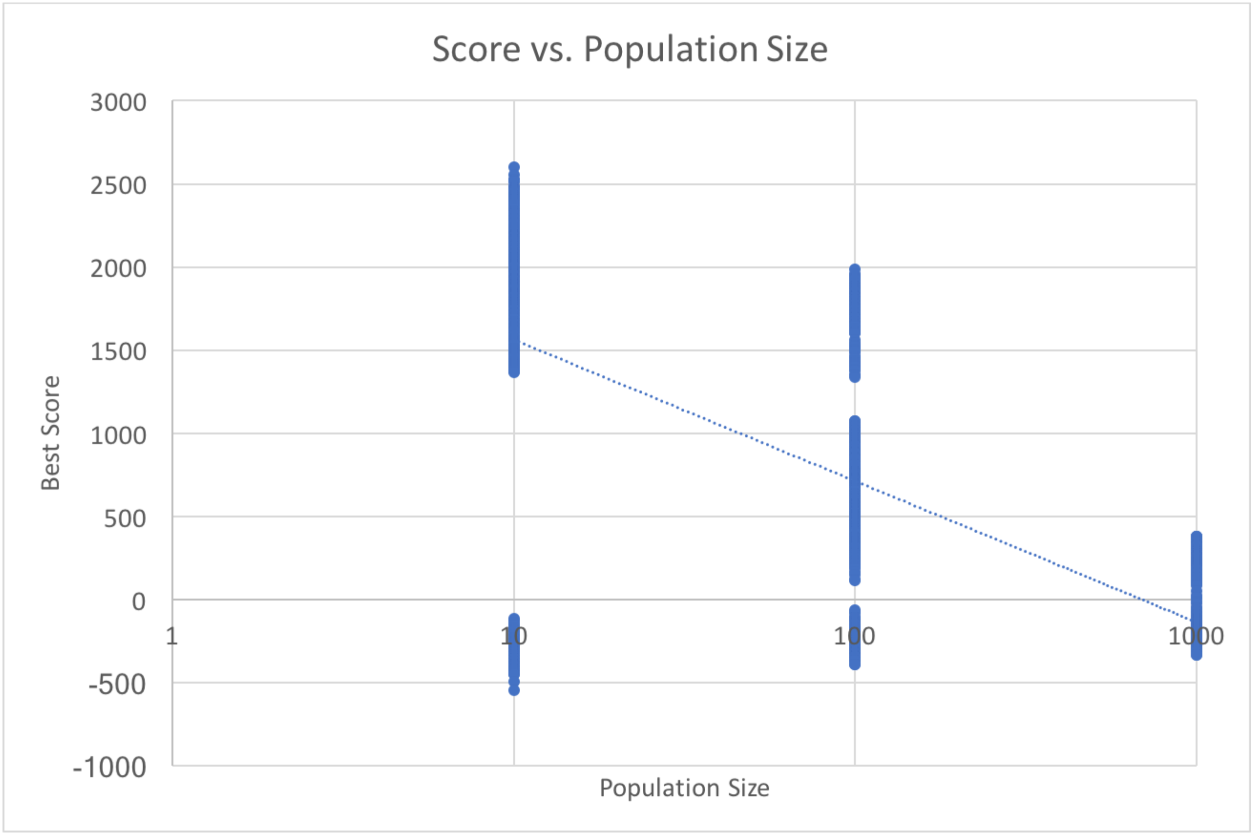
Tuning Genetic Algorithm

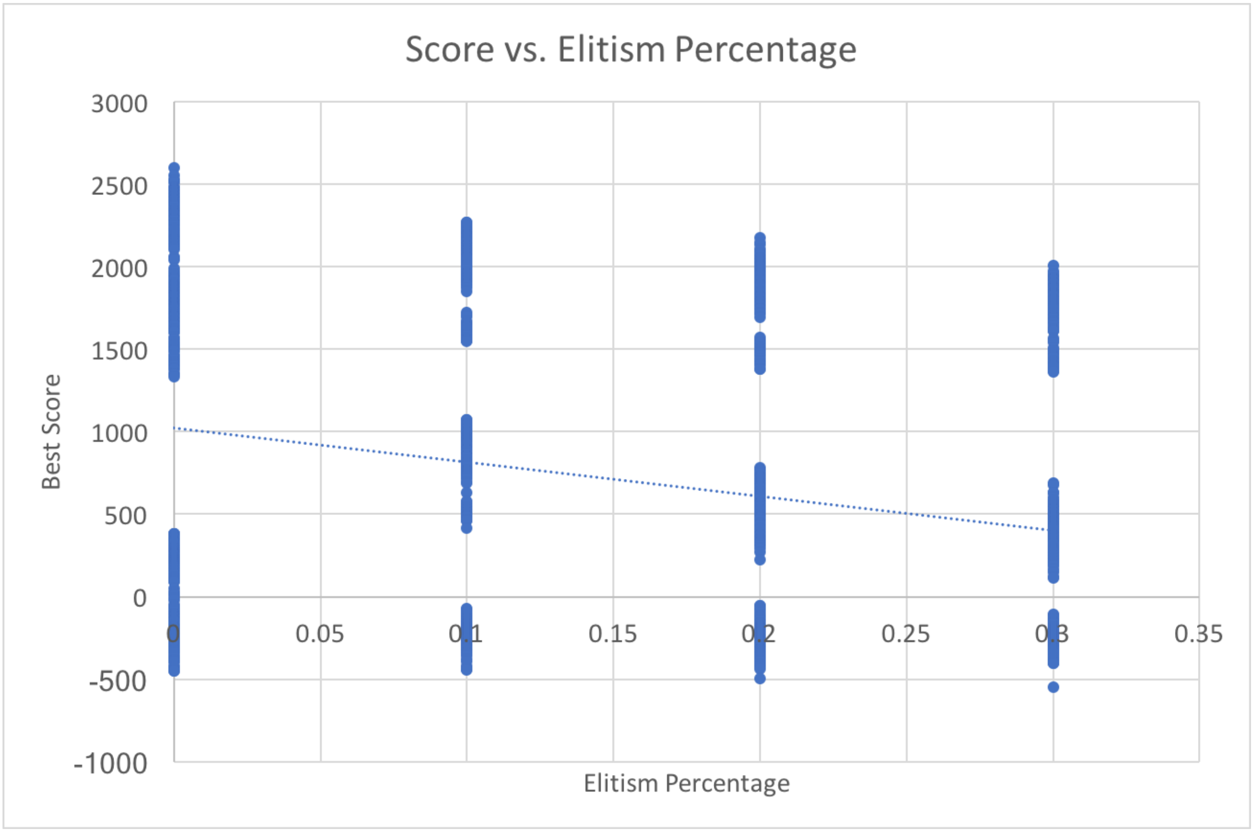
In order to tune the genetic algorithm, we optimized for three different parameters: population size, elitism percentage, and mutation rate. Each possible value of these were measured against all other values for five iterations each with 10 seconds of run time. The parameter values we tested were:

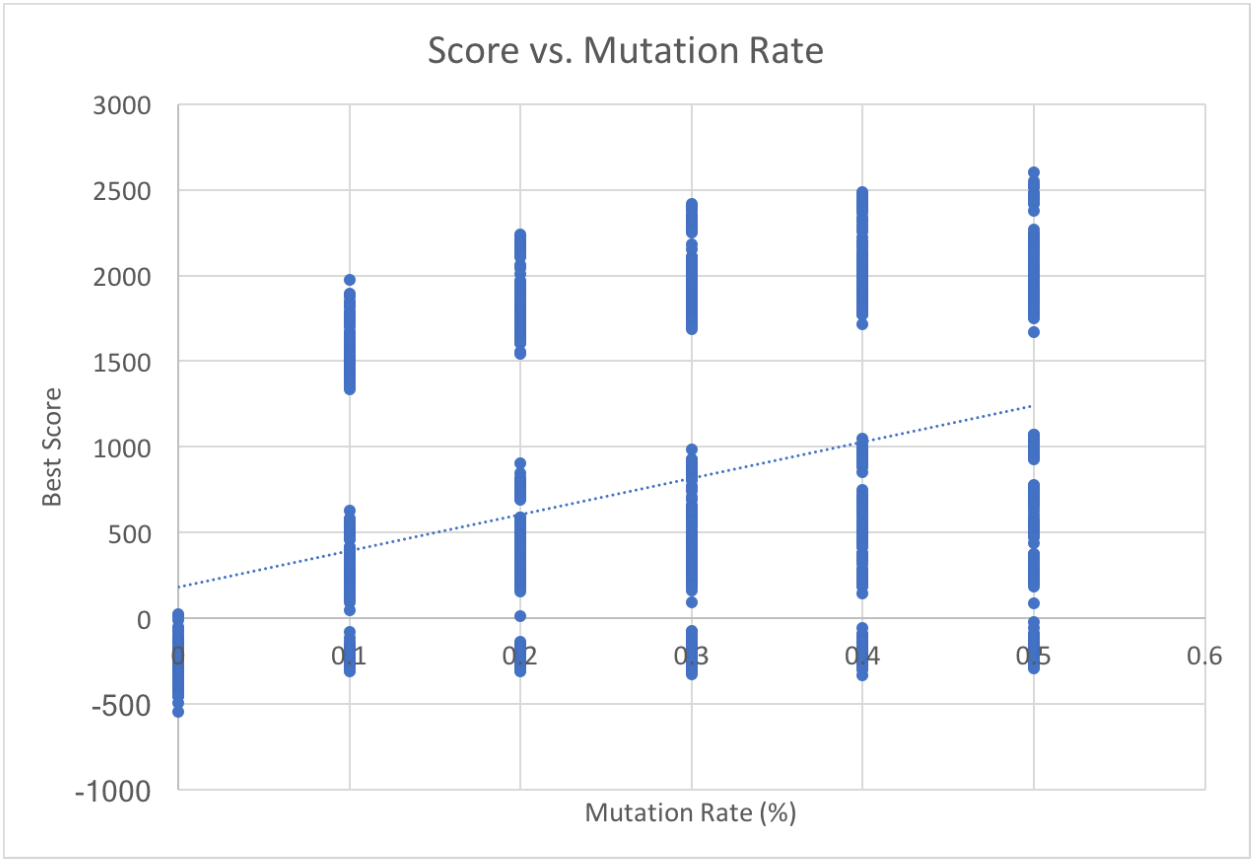
- Population size: 10, 100, and 1000
- Elitism percentage: 0%, 10%, 20%, and 30%
- Mutation rate: 10%, 20%, 30%, 40%, and 50%

The score resulting from each of the combinations of parameters was logged for analysis.

After the tuning test was performed, we looked at the .csv file that was created and analyzed the data to find our optimal parameters. The results of this tuning can be seen in the charts and tables below:




Score vs. Elitism Percentage
Best Score
Elitism Percentage
3000
2500
2000
1500
1000
500
0
-500
-1000
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35


Score vs. Mutation Rate
Best Score
Mutation Rate (%)
3000
2500
2000
1500
1000
500
0
-500
-1000
0
0.1
0.2
0.3
0.4
0.5
0.6

File	Population	Elitism %	Mutation %	Score
tuning_sets/tune_600_b.txt	10	0	0.5	2599
tuning_sets/tune_600_d.txt	10	0	0.5	2552
tuning_sets/tune_600_b.txt	10	0	0.5	2527
tuning_sets/tune_600_d.txt	10	0	0.5	2522
tuning_sets/tune_600_a.txt	10	0	0.5	2520
tuning_sets/tune_600_a.txt	10	0	0.5	2516
tuning_sets/tune_600_b.txt	10	0	0.4	2485
tuning_sets/tune_600_c.txt	10	0	0.5	2481
tuning_sets/tune_600_b.txt	10	0	0.5	2475
tuning_sets/tune_600_d.txt	10	0	0.5	2474

File	Population	Elitism %	Mutation %	Score
tuning_sets/tune_600_e.txt	100	0	0.4	1988
tuning_sets/tune_600_b.txt	100	0	0.5	1955
tuning_sets/tune_600_b.txt	100	0	0.4	1953
tuning_sets/tune_600_a.txt	100	0	0.4	1941
tuning_sets/tune_600_c.txt	100	0	0.5	1932

tuning_sets/tune_600_b.txt	100	0	0.4	1928
tuning_sets/tune_600_e.txt	100	0	0.5	1927
tuning_sets/tune_600_a.txt	100	0	0.4	1925
tuning_sets/tune_600_b.txt	100	0	0.5	1923
tuning_sets/tune_600_e.txt	100	0	0.5	1918

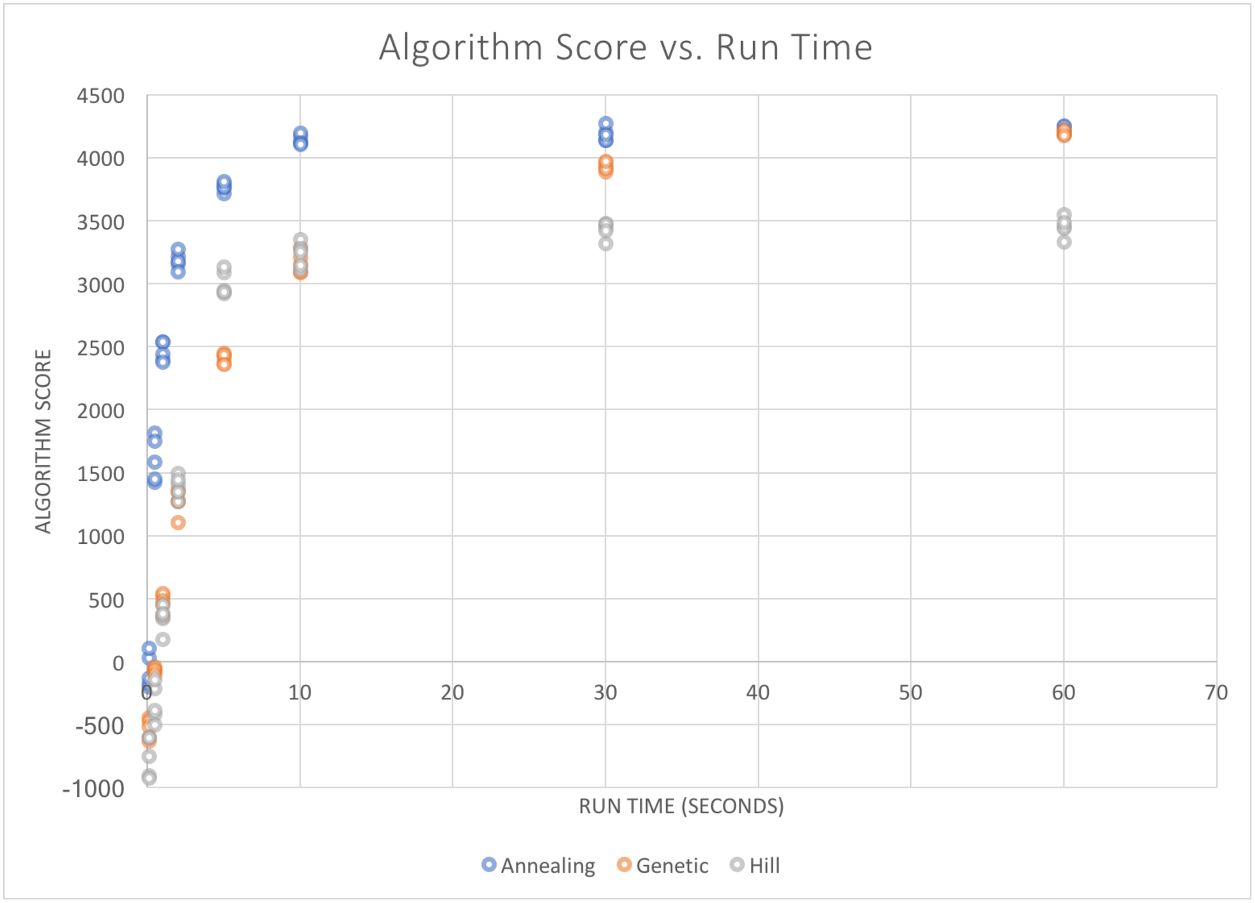
File	Population	Elitism %	Mutation %	Score
tuning_sets/tune_600_d.txt	1000	0	0.3	381
tuning_sets/tune_600_e.txt	1000	0	0.4	380
tuning_sets/tune_600_e.txt	1000	0	0.4	378
tuning_sets/tune_600_d.txt	1000	0	0.5	378
tuning_sets/tune_600_d.txt	1000	0	0.5	374
tuning_sets/tune_600_c.txt	1000	0	0.3	366
tuning_sets/tune_600_b.txt	1000	0	0.4	358
tuning_sets/tune_600_a.txt	1000	0	0.5	358

tuning_sets/tune_600_c.txt	1000	0	0.4	356
tuning_sets/tune_600_b.txt	1000	0	0.5	349

From the tuning data we collected, we determined that the optimal parameters were a population size of 10 with a mutation rate of 50%, and an elitism percentage of 0%.

Analysis

Running all three algorithms on the *test.txt* file yielded the results in the table below, summarized by the chart:



Algorithm	Time (s)	Score test.txt (average over 5 test runs)
Hill	0.1	-754
Hill	0.5	-330
Hill	1	347
Hill	2	1394
Hill	5	3006
Hill	10	3232
Hill	30	3428
Hill	60	3452
Annealing	0.1	-70
Annealing	0.5	1606
Annealing	1	2461
Annealing	2	3186
Annealing	5	3770
Annealing	10	4145
Annealing	30	4188
Annealing	60	4220
Genetic	0.1	-527
Genetic	0.5	-67
Genetic	1	473
Genetic	2	1276
Genetic	5	2406
Genetic	10	3167
Genetic	30	3933
Genetic	60	4197

As time extends to 60 seconds, all of the algorithms approach (or reach, in the case of hill climbing) their asymptotic performance limit. Hill climbing performs the worst of the three, followed by the genetic algorithm. The best performing algorithm seems to be the simulated annealing algorithm, though with a computation time of 60 seconds the genetic algorithm performs almost identically to the annealing algorithm. We believe from the chart that if we continued on and ran the genetic algorithm for longer than 60 seconds it would beat the other two algorithms by a few percentage points, considering it does not appear to hit an asymptotic limit in the tests we performed. Given a shorter computation time, one should use the simulated annealing algorithm, as it performs better at all steps than the other algorithms. With longer computation times (t > 60 seconds) the genetic algorithm would be better, as it will continue to improve if given more computation time.

Bonus Feature

We added multiple process support to our code. When run in multi-processor support mode, the program will detect the number of possible parallel processes that can run and run the algorithm multiple times in parallel using that number of processors. For example, a machine with 4 cores (with hyperthreading) can run 8 simultaneous processes. The program will run any algorithm 8 times in parallel and then take the maximum value. We squeeze out a bit of extra performance in the same amount of time. For example, the hill climbing algorithm might return a value between 2100-2400 for any individual run, but maximizing over 8 processes often yields a value around 2400, the optimal value. To run the program in multi-processing mode, simply add a '-mp' flag at the end of the command line.