Experiment Group Selection with Metaheuristics

Jarom Hulet – Spring 2020

DSA 5900 – Professional Practice – 4 Credit Hours

The University of Oklahoma

Faculty Supervisor – Dr. Randa Shehab

Project Sponsor – Toyota Motor Credit Corporation

Introduction

This project is for four credit hours and is intended to fulfill the practicum requirement for the non-thesis Data Science and Analytics Master program at the University of Oklahoma. The practicum is an unpaid research project sponsored by Toyota Motor Credit Corporation (TMCC). Celia Pan is a manager of data science at TMCC and will be supervising the project. She can be reached by email at celia.pan@toyota.com.

This project will help the sponsor improve market testing by researching a methodology for creating test groups given business constraints. More discussion on the nature and motivation of the project are discussed in the objectives section.

Objectives

Toyota Motor Credit Corporation uses experimentation to test new financial products and pricing strategies before national implementation. The gold standard for experimentation is to use randomization to eliminate the impact of potential confounding factors. In the case of TMCC's business, randomization would ideally be done at a loan application or dealer level. Unfortunately, such randomization is not possible due to business constraints. It is bad practice to change financial product offerings or pricing at an application level since it would likely cause confusion and frustration for the Finance and Insurance (F&I) departments at the dealerships. Changing offerings and pricing at a dealer level is also bad practice since dealers would likely become aware of offers given to other local franchises and may be disgruntled by the differential treatment. Additionally, the consumer finance sector is heavily regulated, and precise differential treatment could cause legal issues for TMCC. Due to these concerns, experimental grouping is typically done by state.

Since there are only 50 states, randomization of states is very unlikely to eliminate the influence of confounding factors in an experiment. In previous employment, I saw a three-month experiment thrown out because the selected state groups had widely different aggregated metrics. This made the results unreliable and the experiment a waste of valuable time and resources.

Splitting the 50 states into similar groups is a difficult problem. The number of possible state groupings is enormous and there is no way to check if a proposed solution is optimal. For example, the total number of possible solutions for putting 50 states into three groups is represented in the formula below:

$$\sum_{n=1}^{48} \sum_{m=1}^{48-n} \binom{48}{n} \binom{48-n}{m}$$

The number of possible solutions is much too large to do an exhaustive search. The best feasible solution is the use a metaheuristic to find a grouping as close to optimal as possible.

This project explored the use of four neighborhood-based metaheuristic algorithms to solve this problem. The four algorithms are (1) Hill Climb, (2) Tabu Search, (3) Simulated Annealing and (4) Variable Neighborhood search. Empirical research was done to test how well each algorithm performed compared to randomly sampling the solution space. There are many other metaheuristic algorithms that could be used to solve this problem. To allow for easier future research and experimentation, the code was developed so that other algorithms could easily be added to the framework.

In this project, the research was done for splitting all the states into three testing groups. However, the Python scripts and methodology developed can be applied to the general case of the problem. Specifically, any number of entities can be put into any number of groups (given the number of groups is less than the number of entities). This project is intended to solve any permutation of this grouping problem in the future.

The objective of the project is to create a Python script that uses a best performing meta heuristic to find state groupings with minimal intergroup differences. This is intended to decrease the likelihood that confounding factors will impact the conclusion of the experiment.

As a student, my learning objectives are to (1) learn to implement and experiment with multiple metaheuristic algorithms, (2) improve Python and object-oriented programming abilities and (3) improve communication skills by working closely with the project sponsor.

Plan

The objectives of this project will be met by writing Python code that creates a framework to execute four different meta heuristic algorithms. The target objective of the algorithms will be to minimize the sum of the total Euclidean distance between the state groupings (see figure 1 for calculation example) and the variance of the intragroup distances. Each of the four algorithms will be optimized by adding or removing common modifications (e.g. adding/removing random walk to the hill-climb algorithm) and by adjusting applicable hyper-parameters. Each of the four algorithms will be rated based on its performance through multiple iterations of testing. The algorithms' performance will be rated on (1) total Euclidean distance and (2) the variance of the intra-group Euclidean distance (see table 1 for metric calculation example). The best performing algorithm will be proposed as the algorithm to solve the grouping problem.

Table 1 – Metric Calculations

	Metric 1	Metric 2	Metric 3
Group 1	10	7	6
Group 2	12	2	6
Group 3	9	18	7

Euclidean Distances

Group 1, Group 2	$= (10-12)^2 + (7-2)^2 + (6-6)^2 = 29$
Group 1, Group 3	$= (10-9)^2 + (7-18)^2 + (6-7)^2 = 123$
Group 2, Group 3	$= (12-9)^2 + (2-18)^2 + (6-7)^2 = 266$

Total Euclidean Distance for Grouping $= 29 + 123 + 266 = 418$

Euclidean Distance Variance =Variance(29,123,266) = 9495

The four metaheuristic algorithms that will be compared are (1) Hill Climb, (2) Tabu Search, (3) Simulated Annealing and (4) Variable Neighborhood Search.

Non-proprietary state economic data was used for this project. The purpose of the dataset selection is to protect the data property of TMCC while still gaining valuable insights. The knowledge gained by this project on non-proprietary data will be easily transferred to private data by TMCC. The data used was a compiled data set from Data.Census.gov[1] and the Bureau of Economic Analysis[2].

Most of the skills required for this project are from the content of DSA 5113. All the elements of this project having to do with metaheuristics are taught in this course. In addition, the concepts of object-oriented programming taught in DSA 5005 were implemented in the development of the code.

Deliverables

The two primary deliverables of this project are (1) Python scripts that allow a user to easily apply and tune the four metaheuristic algorithms on a dataset and (2) a recommendation for the algorithm and tuning parameter values that performed best in the empirical experimentation.

The code was written using a hierarchal class inheritance structure. This structure allows for simple additions of new algorithms for future research and applications. The logical structure of the code can be seen in figure 1. The base class contains all methods needed to create a random starting solution and calculate distances of various solutions. The neighborhood class holds the function to create neighborhoods. The individual algorithms have their own classes that inherit from the neighborhood class. In the future, new classes can be made for new types of algorithms. This allows for clean and concise addition to code without disrupting the functionality of the original code base.

Below are details of some key functions in the code:

- startSol – This function from the base class creates a random starting solution. The starting solution is created by creating an empty list for each group. These lists are sequentially given a random state until all states have been assigned. This way of assigning states to groups results in groups that have close to the same number of states in them.
- Standardize – This function from the base class standardizes all of the metrics in the data so that the scale of the individual metrics have proportional weights in the distance calculations.

1 https://data.census.gov/cedsci/table?q=state%20demographics&g=0100000US.04000.001&hidePreview=true&table=CP05&tid=ACSCP1Y2018.CP05&lastDisplayedRow=93&vintage=2018&mode=

2 https://www.bea.gov/news/2019/gross-domestic-product-state-second-quarter-2019

- pairDist, groupMetrics, totalDist – These three functions from the base class are used in conjunction to calculate the total distance between the input groups. In this project, only Euclidean distance is used, but the code can calculate Manhattan and Cosine distance as well. Other distances can easily be added if deemed appropriate in the future.
- Sample_hist – This function in the base class creates a histogram of random neighbors. This can be used to see how well an algorithm performs against randomly selected solutions.
- createNbrhood – This function is in the neighborhood class. It takes a single solution as an input and creates a specified number of neighbors. It creates neighbors by randomly selecting n neighbors in a group and randomly swapping them with an adjacent group. In order to allow group sizes to fluctuate, a group will randomly swap an extra state based on a probability that the user inputs. Without the extra swap feature, the states in each group would change, but the size of each group would never change. This would represent a significant restriction in solution search space. Since the full-size neighborhoods are large (4,624 for a three-group neighborhood with 16, 17 and 17 for the sizes of group 1, group 2 and group 3 respectively) the createNbrhood method allows for the user to sample from the neighborhood to save on computational power.
- runNVS, runHillClimb, runTabu, runSimulatedAnnealing – All of these methods are in their own classes and run their respective metaheuristic algorithms.

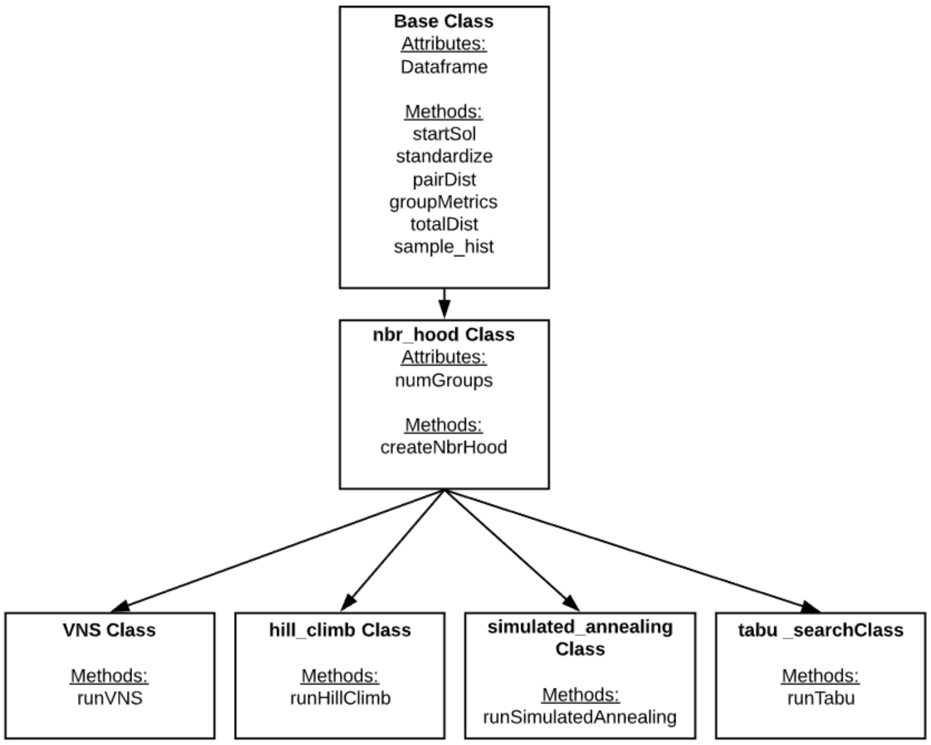


Figure 1

All code developed for the project can be viewed and cloned from this GitHub repository https://github.com/jaromhulet/DSA_5900.

Schedule

The schedule for the project will be as follows:

Jan 13th to Jan 17th – Submit proposal

Jan 17th to Feb 24th – Develop code

Feb 25th to Feb 29th – Test code

Mar 1st to Mar 15th – Initial algorithm runs and analysis

Mar 16th to Mar 31st – Tune parameters and experiment with algorithm variants

Apr 1st to Apr 13th – Prepare write-up to document methodology and recommendations

All due dates were met on-time or earlier. There were no delays in the schedule.

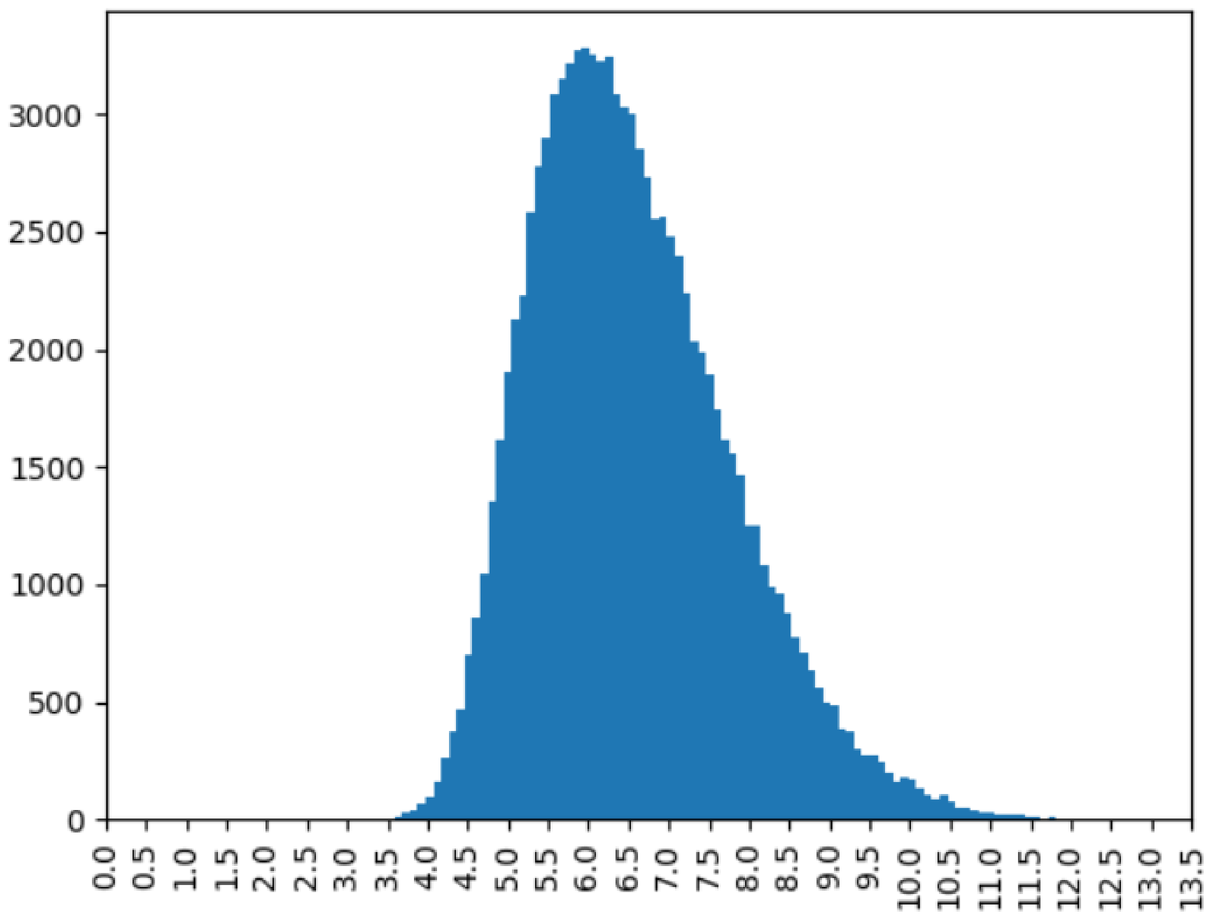
Results

As discussed previously, in order to solve this problem, four metaheuristic algorithms were deployed. Different skews and hyperparameters were used in the algorithms to get the algorithm as close to optimal performance as possible. The performance of the algorithms were compared against each other as well as against a benchmark of randomly sampled solutions.

It is important to note that this project was not able to experiment with every skew and every possible permutation of hyperparameters for these algorithms. The combinations are nearly unlimited while this project is bound by both time and computational power. During the experimentation, best judgement was used to explore the most promising skews and hyperparameter values.

Benchmark

The algorithms must be able to find better solutions than random sampling. Random sampling serves as the benchmark for this project. The histogram of the random samples also gives an idea of the underlying distribution of the performance metric in the solutions space. Figure 2 shows a histogram of the Euclidean distance of 100,000 randomly selected groups of three.



Five Number Summary

min	*25%*	*median*	*75%*	*max*
3.103	*5.632*	*6.390*	*7.310*	*12.794*

Figure 2

The initial proposal for this project said that the winning algorithm would be determined by consideration of the total Euclidean distance as well as the variance of the pairwise Euclidean distances. The weakness of this methodology is that the variance of the pairwise distances is not factored into the heuristic algorithms. Since the variance is not in the algorithms, it basically comes down to random chance whether a potential solution has high or low pairwise variance relative to other solutions. It is important that all the groups are similar to each other. Table 2 shows an example of how a solution with a lower total distance can be inferior to a solution with a larger total distance. In the example, solution 1 could cause problems with the experiment since Group 2 and Group 3 are much less similar than the other pairings. Solution 2, while higher in total distance would likely give experiment results with a lower bias since all pairwise groupings are approximately equally similar. Solution 2 has a much lower variance than solution 1.

Table 2 – Metric Calculations

Solution 1		Solution 2	
Pairwise Groups	Pairwise Distances	Pairwise Groups	Pairwise Distances
Group 1, Group 2	0.22	Group 1, Group 2	0.32
Group 1, Group 3	0.25	Group 1, Group 3	0.35
Group 2, Group 3	0.51	Group 2, Group 3	0.34
Total Distance	0.98	Total Distance	1.01
Variance of Distances	0.016956	Variance of Distances	0.000156

It is important to consider the variance of solutions in the metaheuristic algorithms. For the application of this project, the objective function was changed from the sum of the pairwise distances to the sum of the pairwise distances plus a multiplicative factor times the pairwise variance. The higher the factor, the more influence the variance will play in the search for an optimal solution.

Objective Function

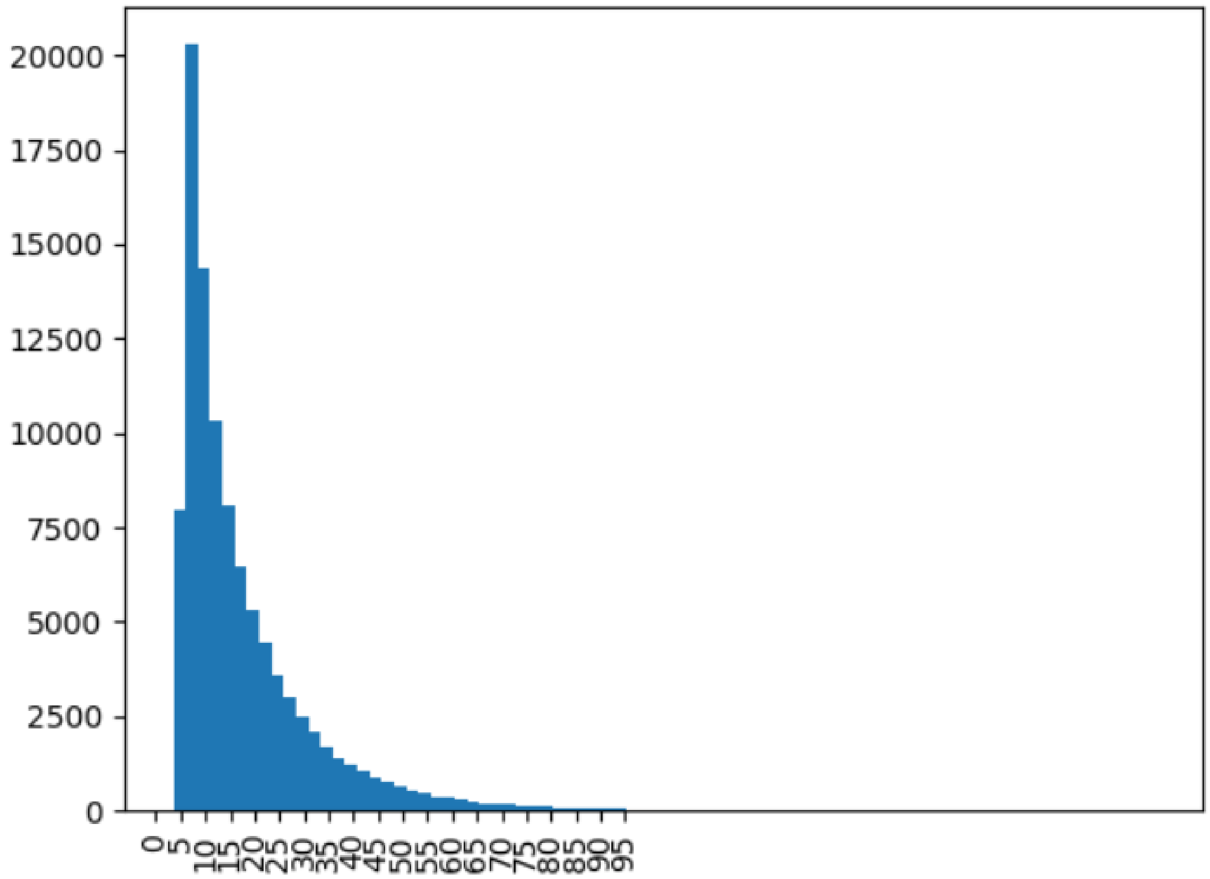
$$\left(\sum_{n=1}^{g} d_n\right) + \varepsilon\sigma^2$$

Where:

g = number of pairwise group sets
d_n = the pairwise distances for pair n
ε = weighting factor
σ^2 = variance of the pairwise distances

For this project, ε was set at a value so that the total distance and the variance were approximately equal in weight. This was accomplished by setting ε equal to the median of total distance divided by the median of the variance (6.39/0.173=36.85). The code base allows for ε to be set at any value. Future users of the scripts can adjust the importance of the variance as appropriate for the specific application. Figure 3 shows the distribution of the objective function that was used in this project.

Figure 3 is a distribution of 100,000 random samples using the new objective function. The distribution of the variances between group distance has a strong positive skew and is truncated at zero. When the distribution of the variances is added to the total Euclidean distance distribution, the result is a right skewed distribution. In business terms this means that there are a lot of pretty good solutions and a couple of very bad solutions. Note that the mean of the distance plus variance metric in figure 3 is approximately twice the size of the distance distribution in figure 2. This is evidence that about half of the weight of the second distribution comes from the total Euclidean distance and the other half comes from the variance times the weighting factor.



Five Number Summary

min	*25%*	*median*	*75%*	*max*
3.554	*8.052*	*12.722*	*22.082*	*201.996*

Figure 3

Hill-Climbing

The most basic algorithm that was explored is hill-climbing. The other three algorithms generally build on this fundamental methodology. The skews of hill-climbing that were explored are best accept, first accept and random walk. Random restarts will be used to help prevent stopping at poorly performing local minimums.

Best Accept

The best accept skew of the hill-climbing evaluates each neighbor in the sample of neighbors and always chooses the neighbor with the lowest metric. This is the most greedy of the Hill-Climb skews explored. The applicable inputs for hill-climbing with best accept are (1) the number of restarts and (2) the neighborhood type and sample size. This portion of the project explored two basic strategies. The first strategy is smaller neighborhoods with more restarts and the second is larger neighborhoods with fewer restarts. Clearly, larger neighborhoods with more restarts has the highest chance of yielding a better solution, but applying time and computation constraints help the project yield practical results that can be implemented in a corporate environment.

Initially, neighborhood samples from 10 to 150 were used and 50 restarts were applied. Figure 4 shows the results of the various runs and the total number of solutions that were examined per iteration.

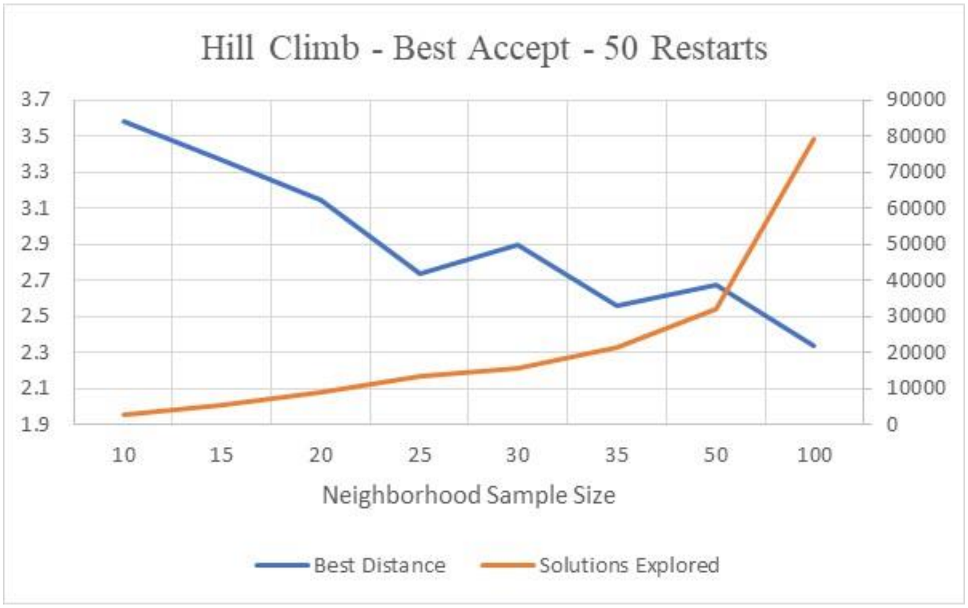


Figure 4

For the set of algorithm tests with larger neighborhoods and fewer restarts, neighborhood samples between 150 and 250 where used each with 10 restarts (fig. 5). The larger neighborhood size with fewer restarts is more effective than the smaller neighborhoods with more restarts. The larger neighborhoods find better solutions with less exploration.

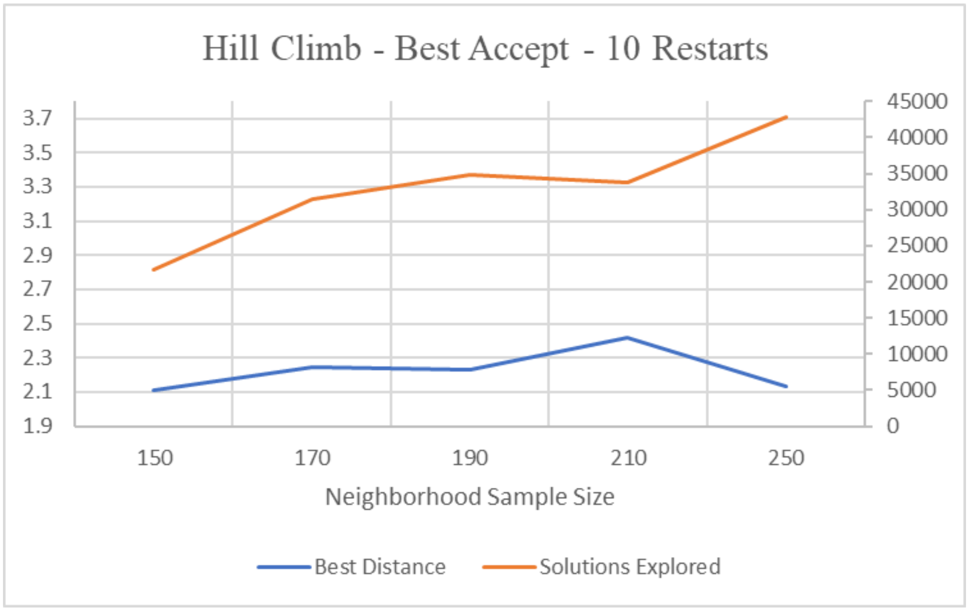


Figure 5

Hill-Climb with Best Accept is able to achieve relatively good solutions compared to most of the other algorithms, but at a very high cost. It is one of most computationally expensive algorithms explored in this project (rivaled only by Tabu search). Fisrt Accept gets better results with far fewer computations. This is evidence that Best Accept gets stuck at local minimums after exploring many neighbors.

First Accept

In the first accept skew of hill-climbing, the first solution that is better than the current solution is immediately accepted. This less greedy approach is intended to avoid local minimums and move more quickly through the search space. The same approach will be used for first accept as was used for best accept. Smaller neighborhoods with many restarts were explored (fig. 6), and then larger neighborhoods with few restarts were explored (fig. 7).

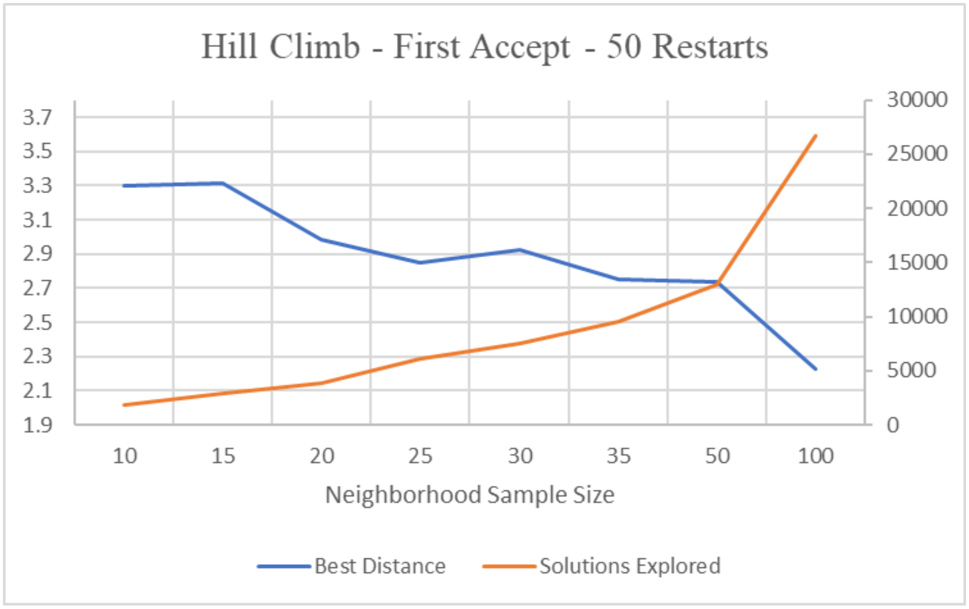


Figure 6

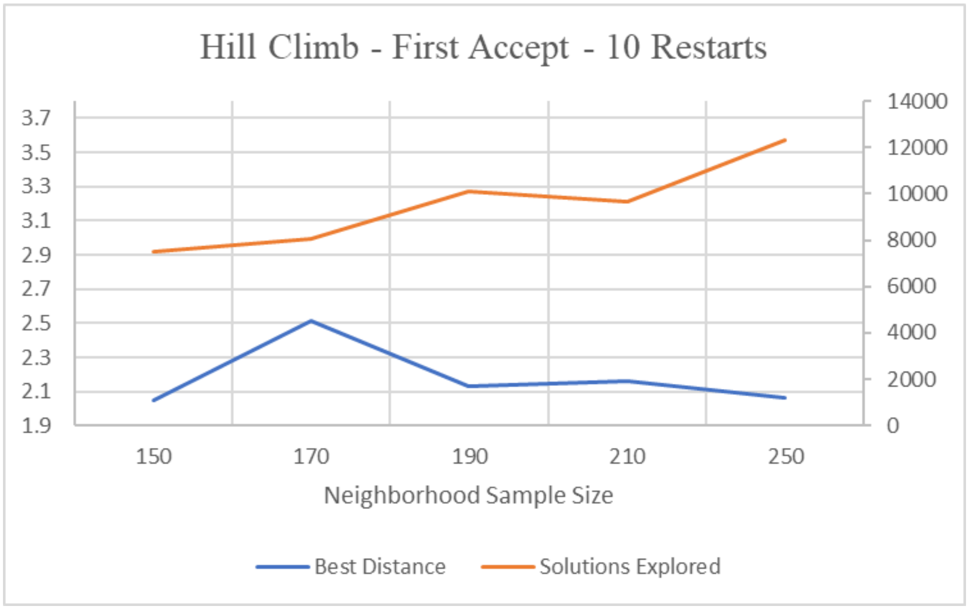


Figure 7

Fewer restarts with more neighbors is clearly superior for both Best Accept and First Accept. The starting analysis for the other algorithms were set to few restarts and more neighbors due to this finding.

Hill-Climb with First Accept is a top performing algorithm among the algorithms used in this project. It has relatively low computational cost and returns among the best solutions.

Random Accept

The random accept skew of hill-climbing is like best accept in that every neighbor in the sample of neighbors is examined. It is unique in that each neighbor gets a weighted probability based on the objective function score (see table 3 for calculation). The next solution is then selected stochastically from the probability distribution created. This skew takes size of neighborhood sample, number of restarts and a power as inputs. The power input controls what the probability distribution looks like. The higher the power, the less likely an inferior solution is to be accepted. For the initial run, the power of 1 was used with a neighborhood sample size of 150, but after over one hundred-thousand iterations, the algorithm was still on its first restart. It was concluded that in this application, 1 was too small and thus the algorithm was basically doing a random walk of the solution space. Higher powers were iteratively used but the algorithm would still not converge on a solution. Finally, the neighborhood sample size was reduced to give the algorithm fewer choices. This had limited success because it was difficult to find parameter combinations that converged in a reasonable amount of time.

Table 3 – Random Probability Calculation Example

	Original Value	Flipped Order Value	Apply Power of 2	Proportion of Value = Probability of Acceptance
Neighbor 1	18	3	9	1%
Neighbor 2	16	5	25	2%
Neighbor 3	15	10	100	9%
Neighbor 4	11	11	121	11%
Neighbor 5	10	15	225	21%
Neighbor 6	5	16	256	24%
Neighbor 7	3	18	324	31%

Most of the iterations attempted ran for hours without terminating. The iterations that did terminate had very bad solutions compared to the other algorithms (see table 4 for results). For these reasons Hill-Climb with Random Select was not recommended. It is important to note that there are many other ways to introduce randomness in the selection process and this algorithm only uses one of those methods. Other methods not explored in this project could yield better results.

Table 4 – Random Select Results

Random Select 25 Restarts

Power	Neighborhoorhood Sample Size	Solutions Explored	Best Distance
3	15	5730	3.768024852
3	20	10700	3.268480516
4	15	4485	3.40317048
5	20	3750	3.643964884

Variable Neighborhood Search

The variable neighborhood search algorithm uses multiple neighborhood definitions to search a solution space. In this case, two types of different neighborhoods were created. The first type was created by adding additional state swaps (e.g., 2-swap, 3-swap etc.). The second was created by increasing the neighborhood sample size (e.g., original neighborhood sample size is 10, second type of neighborhood has sample size of 20). The specific algorithm implemented in this project used the 1-swap neighbor to start out with (same definition as used in all other algorithms in this project) and then when a value that is better than anything seen so far is found, it switches to the second neighborhood type for one iteration. The main idea of this algorithm is to look harder when in a relatively good spot and save time when in relatively bad spots.

The VNS algorithm for this project uses the "best accept" logic when selecting solution space movements.

Six types of variable neighbors were explored, (1) 2-swap, (2) 3-swap, (3) 4-swap, (4) 2x neighborhood sample size, (5) 3x neighborhood sample size and (6) 4x neighborhood sample size. See VNS swap results in fig. 8 and VNS expand results in fig. 9.

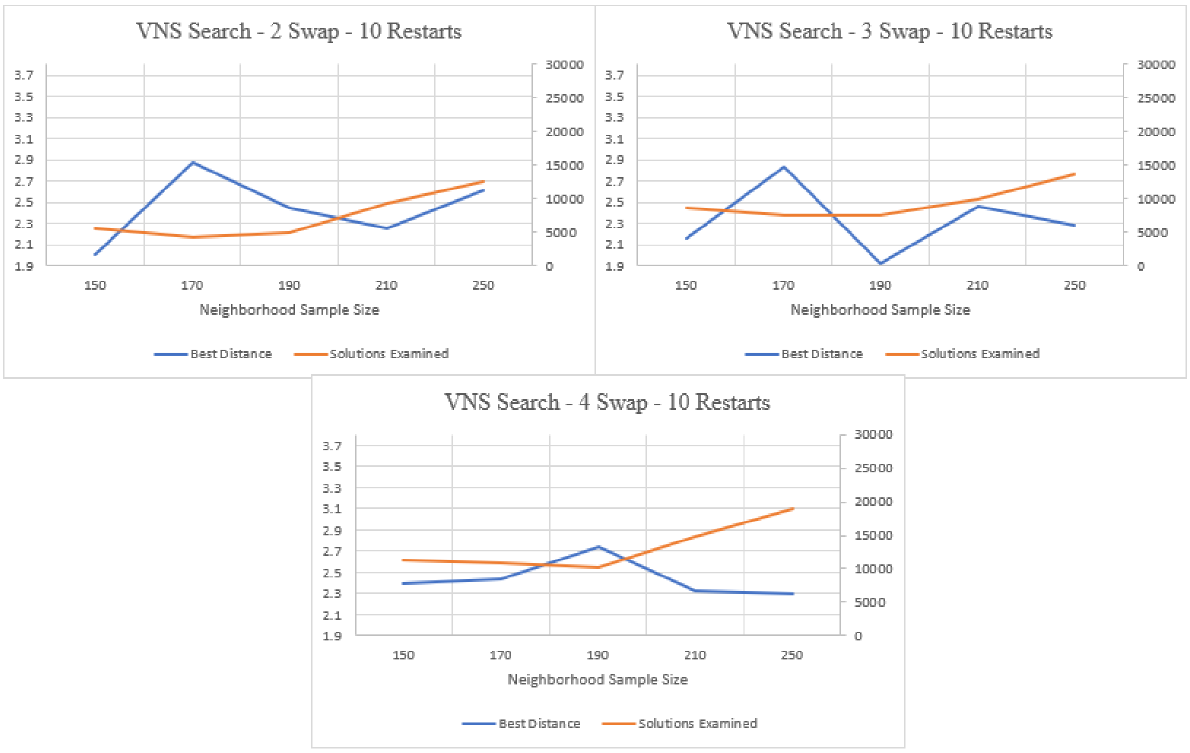


Figure 8

VNS swap has a larger range of variance in the value of the best solution across neighborhood sample sizes. The computational cost is low relative to other algorithms like Tabu search and best accept.

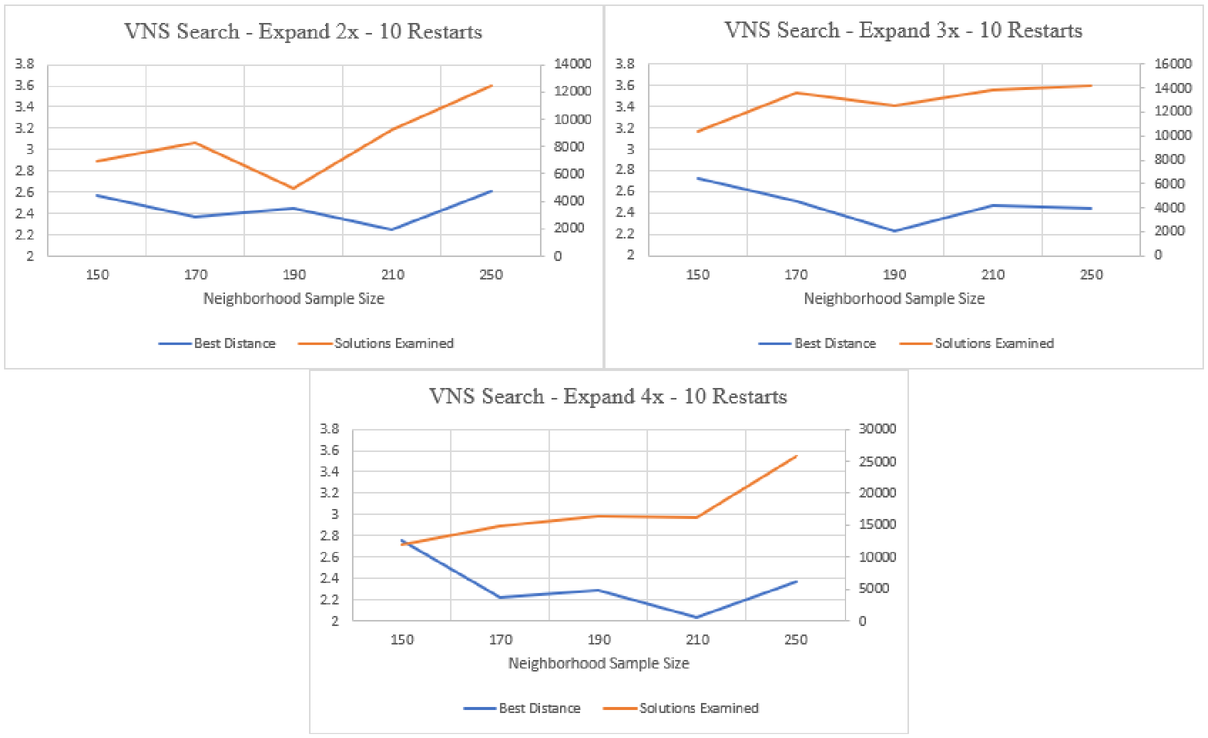


Figure 9

VNS expand has relatively low computational expense, but at the cost of performance. The value of the best solutions found is a little high compared to other algorithms like Hill-Climb with Best Accept and Hill-Climb with First Accept. It could be that VNS expand is too greedy. The expansion allows the algorithm to behave even more greedily than regular Hill-Climb Best Accept. The fact that VNS explores fewer solutions (even though the neighborhoods grow significantly larger) than Hill-Climb Best Accept is evidence that the greediness is stopping quickly at local minimums.

Tabu Search

Tabu search is an algorithm that uses memory to temporarily limit the search space based on previous searches. This technique is designed to avoid local minimums by forcing the algorithm to not be able to explore some parts of previously high scoring solutions. In this project's implementation of Tabu search, the states that were swapped to get to a better solution are set as tabu and cannot be removed for a certain number of iterations. The number of iterations that states are locked is referred to as tenure. Tenure and restarts are the only parameters that can be adjusted for this version of tabu search. Both were experimented with to get the below results in fig. 10.

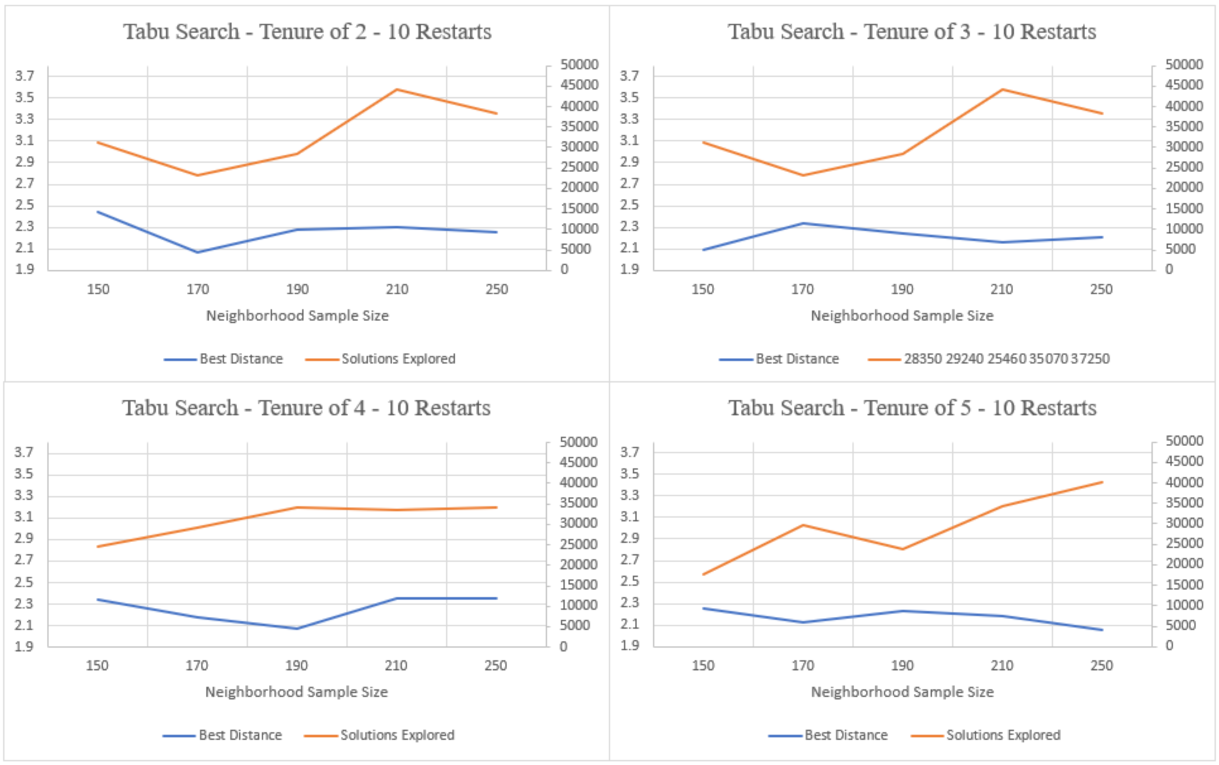


Figure 10

In this application of Tabu search, changing the tenure of the search did not appear to change the quality of the solution found or the number of solutions explored. Tabu search finds reasonably good solutions, but at a higher computational cost than any of the other algorithms.

Simulated Annealing

Simulated annealing is an algorithm that allows for some bad moves at the beginning of the run, but decreasingly allows for bad moves as the run grows older. Eventually it is very unlikely to accept bad moves. This is yet again another attempt at avoid local minimums. The time that it takes for the algorithm to move from allowing a lot of bad moves to allowing very few is called a cooling schedule. The cooling schedule along with the number of iterations at each temperature are the two inputs that will be tested in this project.

Two cooling schedules were tested in the project:

Schedule 1 = 0.15,0.125,0.1,0.075,0.05,0.025,0.01

Schedule 2 = 0.15,0.14,0.13,0.12,0.11,0.10,0.09,0.08,0.07,0.06,0.05,0.04,0.03,0.02,0.01

Figure 11 shows the results of the Simulated Annealing runs. The number of solutions explored is not included in the charts because it is deterministic for Simulated Annealing. The number of temperatures multiplied by the number of iterations at each temperature is the number solutions explored.

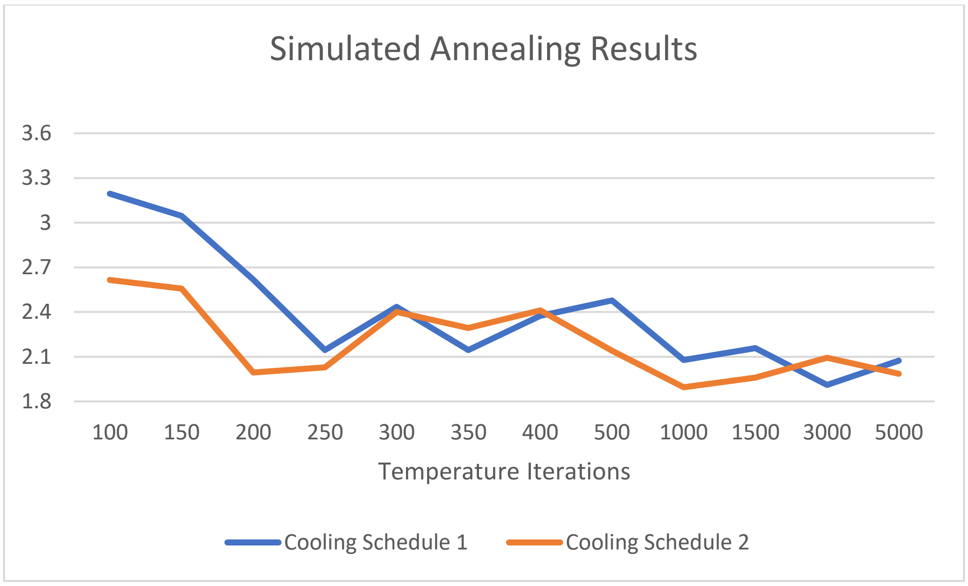


Figure 11

Simulated Annealing consistently finds good solutions at about 7,000 solutions explored which is fast compared to most of the other algorithms. Simulated Annealing also found that lowest score of the testing.

Further Analysis

The top performing algorithms are Hill-Climb with First Accept and Simulated Annealing. The two algorithms have similar performance. One additional test was run on both algorithms to gather additional information to help differentiate their performance. Both algorithms were set to run until a total of 100,000 observations were explored. This created a solutions-explored level comparison between the two algorithms and the original benchmark. The results of the tests are in table 5.

Table 5 – 100k Iteration Results

Algorithm	100,000 Observation Run
Benchmark	3.554
First Accept	1.976
Simulated Annealing	1.748

The results of the 100,000 solution test helped create the final recommendation in the next section.

Figure 12 shows the states included in the best state grouping found in all of the experimentation (found by Simulated Annealing in the 100,000-solution exploration test run).

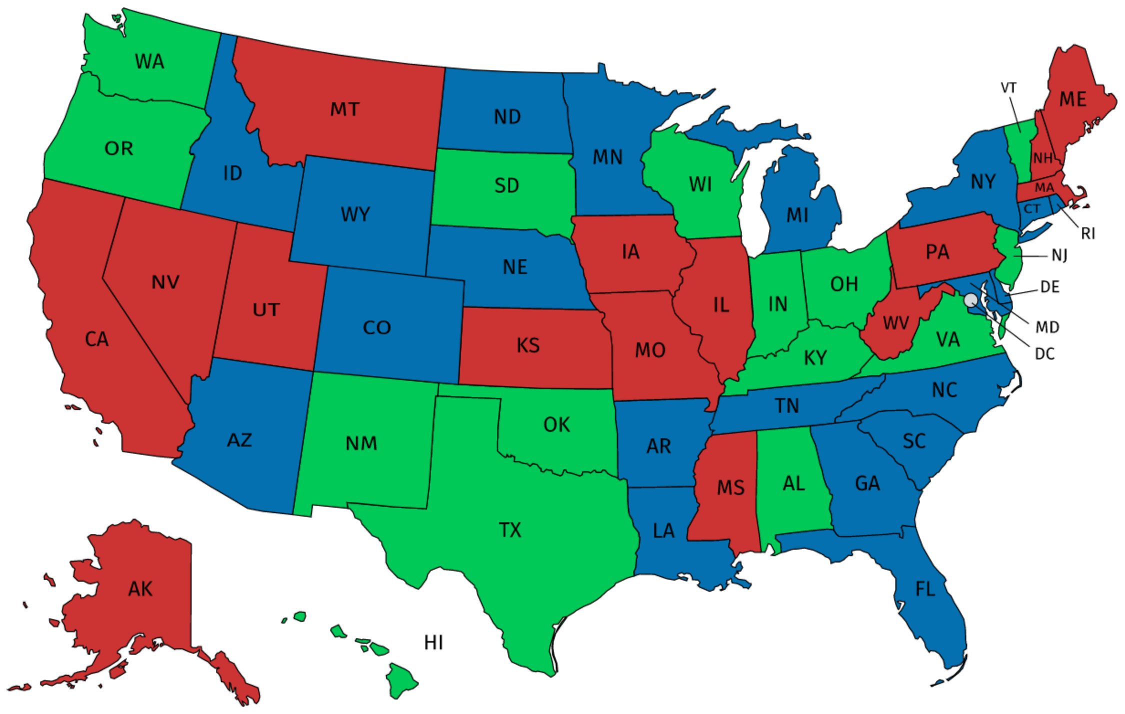


Figure 12

Algorithm Recommendation

Based on the empirical evidence gathered, the Simulated Annealing algorithm with cooling schedule 2 is recommended. It was clearly a top performer in the first set of tests and found the lowest objective valued solution in the secondary testing against Hill-Climb with First Accept. Simulated Annealing consistently found good solutions in very few iterations. It found the lowest valued solution of all tests in the first set of testing with schedule 2 and 1,000 temperature iterations (15,000 total solutions explored). Another benefit of the Simulated Annealing algorithm is that the number of solutions explored is deterministic. The user has exact control over the computational power that will be used. It is the only algorithm of the algorithms included in this project that has a deterministic number of explored solutions.

Self-Assessment

I was able to meet all of the project deadlines and handoff the two specified deliverables on time. I feel that I was very thorough in my exploration of the algorithms' capabilities. I feel that the code and analysis I delivered were thorough and well thought out.

The main challenge that I faced in the project was deciding what parameters to change and how much time to spend on each algorithm. I was able to partially overcome that challenge by using what I learned from the Best Accept and First Accept algorithms to decrease the scope of exploration with the other algorithms. I also addressed this problem by increasing my capacity to explore the algorithms by running the experiments on multiple EC2 instances on AWS simultaneously. Overall, the project spent over 250 computing hours to come up with the results presented here. While there will always be more to explore, I think I did enough to give good recommendations to TMCC. Also, if TMCC wants to do any more research or analysis, the code is written in a dynamic way where modifications or continuations of experiments is relatively easy.

This project has helped me (1) gain a much deeper understanding of the metaheuristic algorithms applied, (2) increase my knowledge and proficiency of the Python programming language and (3) better my communication skills by working with the project sponsor to communicate progress and final deliverables.

References

1. Kirkpatrick, S.; Gelatt Jr, C. D.; Vecchi, M. P. (1983). "Optimization by Simulated Annealing". *Science*. **220** (4598): 671–680. Bibcode:1983Sci...220..671K. CiteSeerX 10.1.1.123.7607. doi:10.1126/science.220.4598.671. JSTOR 1690046. PMID 17813860.
2. *Fred Glover (1989). "Tabu Search – Part 1". ORSA Journal on Computing.* ***1*** *(2): 182–203. doi:10.1287/ijoc.1.3.190*
3. Fred Glover (1990). "Tabu Search – Part 2". *ORSA Journal on Computing*. **2** (1): 3–32. doi:10.1287/ijoc.2.1.4.
4. Hansen, P.; Mladenovic, N.; Perez, J.A.M. (2010). "Variable neighbourhood search: methods and applications". *Annals of Operations Research*. **175**: 367–402. doi:10.1007/s10479-009-0657-6.