CS 4641 Assignment 2: Randomized Optimization

Maddie Ravichandran
GTID: 903006897

Overview:

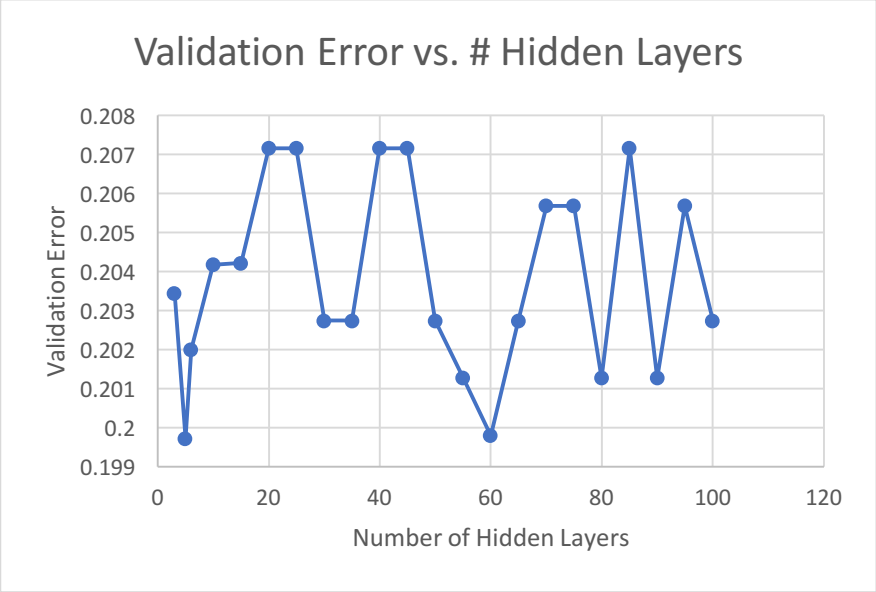
In this assignment, I will perform a detailed analysis of the Game of Thrones (GoT) dataset using three main optimized search algorithms: Hill-Climbing, Simulated Annealing, and Genetic Algorithm. I will use these randomized optimization algorithms to find the weights for the neural-network implemented in assignment 1, and compare these results with the backpropagation method. Furthermore, I will explore three different optimization problems to provide an in-depth analysis of the three optimized search algorithms. This paper will examine the advantages and disadvantages of each algorithm in these respective domains. Lastly, I will expand on an existing demonstration of a genetic algorithm on Flappy Bird to study how modification of the algorithm alters the gameplay. Overall, this assignment will provide a comprehensive understanding of the different randomized optimization algorithms we studied in class.

The dataset is in the testing folder but can also be found in the following link:
https://www.kaggle.com/mylesoneill/game-of-thrones

Part One:

To review from assignment 1, the GOT dataset is a collection of information about various characters in the Game of Thrones novels, in which I use the features, Prediction, Male, isNoble, isMarried and isPopular to determine if the characters will survive. The dataset has 1947 data points. Last time, several supervised learning algorithms were trained and tested on this dataset. For training, 70% of the data was separated. The last 30% of the data was used to for testing.

Figure 1. Finding the # of Hidden Layers



To first create a neural network (NN) to model the data, the network was tuned to find the optimal parameters. Through experimenting with ABAGAIL [1], and adapting code from MosDragon on Github [2], it was found that a neural net of 5 hidden layers provided the

greatest accuracy with back propagation with a minimum validation error of 19.973%. This is shown in figure 1 above. This number was used throughout the experiments for the randomized optimization algorithms. It is important to note that there is a preference for a short network, which was why 5 hidden layers was used instead of 60 layers even though they roughly estimated the same validation error. Shorter networks helped decrease the computational time to perform higher volumes of iterations. With a smaller degree of freedom, it makes the search space smaller allowing the randomized search algorithms to converge on some optima faster.

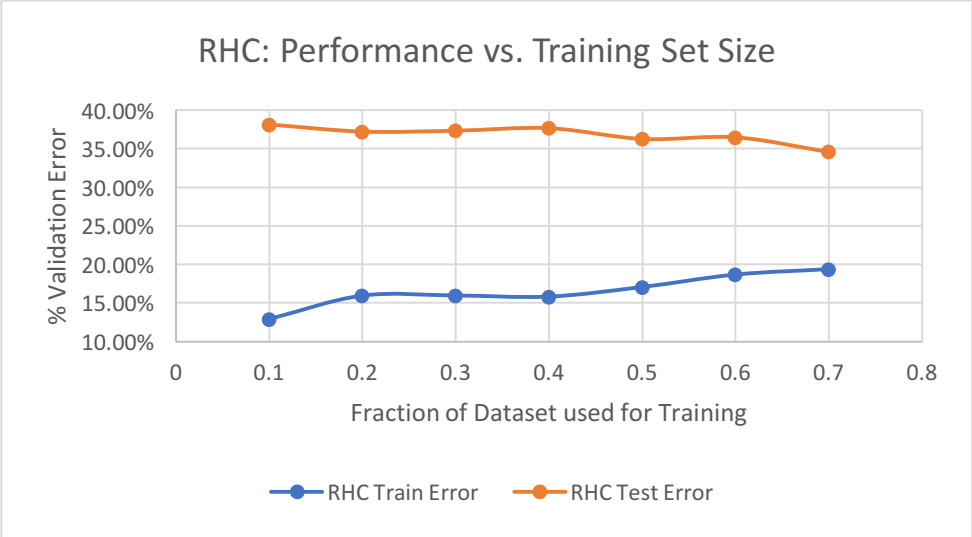
Cross validated (k =10) backpropagation was run using every combination of (k-1) folds for training to find the best weights for the neural network. The kth fold was used for validation at each iteration. This was run for 2000 iterations and the lowest validation set error was recorded and used as the 'best' model. This was a NN built using an input layer equal of 5, an output layer equal to 2, and a hidden layer with the number of nodes equal to 5. The initial learning rate was set to 0.1, the maximum learning rate was set to 50 and the minimum learning rate was set to 0.000001. The model was then trained and tested to find the error rate for both training and testing.

It was found that the best Neural Network after back propagation had a training error of 19.236% and testing error of 36.130% The total elapsed time was 3.274 seconds. This is expected behavior that the training error would be a lot lower than the testing error. However, this also implies that the set of weights found in training do not generalize well for new data. There may have also been some optimization issues in the error derivatives on individual data points that did not translate well into creating a good set of weights.

Randomized Hill Climbing (RHC):

RHC was the first random optimization algorithm that I used to find the weights for the neural network created from assignment one. The algorithm works by essentially starting at random points of the search space and moving in the direction of the optima.

Figure 2. RHC Performance vs Dataset size



Using the basic neural network described above, RHC was used to find the weights. First, I wanted to see well the algorithm ran with various amounts of input data. This is shown above

in Figure 2 where set sized was increased from 0.1 to 0.7. The remaining data was used for testing respectively. It is easy to see the general trend of increasing training error as the size of the set increases. This is due to the fact as more data is available for RHC to train on, the algorithm must search more area and it becomes harder to generalize.

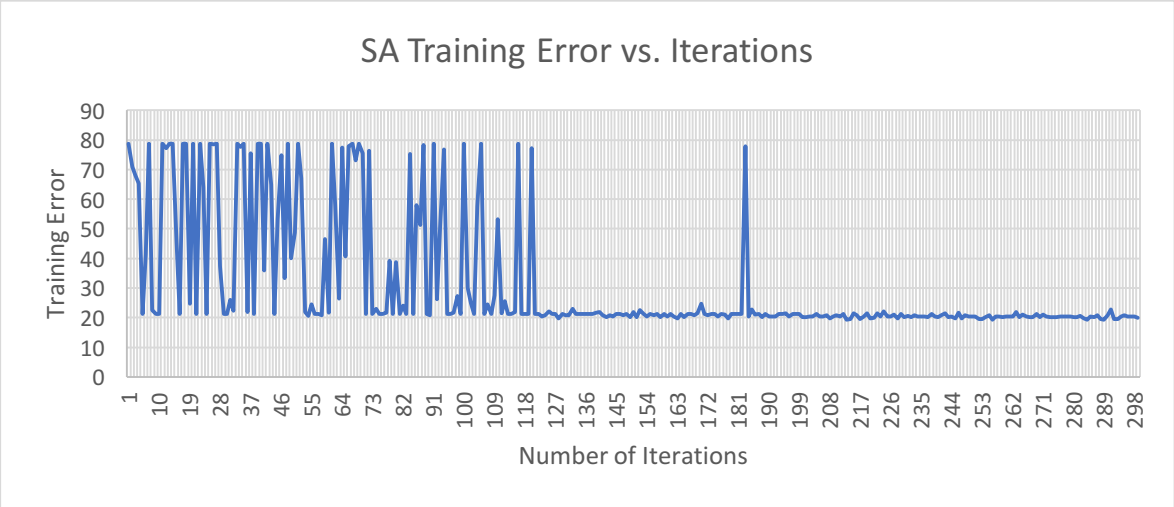
Moreover, the algorithm is more likely to find local minima as the dataset increases in size. However, random restarts were utilized as an effort to minimize the chances of this occurring. Implementing random restarts allows the algorithm to explore more areas and try new solutions if it get stuck at local optima. At a training set size of 0.7 with 200 iterations, the algorithm achieved a training error of 19.530% and a testing error rate of 35.788%. The algorithm was able to converge on a solution that performed better in testing in comparison to back propagation but not by a significant amount with a difference of 0.181. RHC was also able to find the set of weights 1.7 seconds faster than back propagation. The complete set of weights found by RHC is found in Appendix A.

Simulated Annealing (SA):

SA was the next optimization algorithm used to find weights for the GOT neural network. SA works from the metaphor used in physics where heating and cooling of metals in a controlled schedule will help increase the strength. In optimizing search, the algorithm essentially converges to some optima through a schedule. The algorithm combats getting stuck in local optima by using a temperature function that is slowly decreased overtime to decide whether the algorithm should continue searching. As temperature decreases, the algorithm is less likely to search worse areas and finds the best optima it could in its schedule.

To implement SA, we needed to first find the parameters of interest of starting temperature and cooling rate. The algorithm was trained using the same cross-validation settings to find the best number of hidden nodes in the back-propagated NN. Temperatures in the set A = {1e5, 1E8, 1E10, 1E12, 1E15, 1E20, 1E25}, and cooling rates in the set B = {0.6, 0.7, 0.8, 0.85, 0.9, 0.95, 0.99, 0.999} were cycles through to find the best parameter configuration.

Figure 3. Training Error vs. Iterations



While every cooling rate and temperature eventually converged in some optima, it was found that a temperature of 1E12, and a cooling rate of 0.6 gave the smallest validation error in 300 iterations. Figure 3 above shows the training and testing validation error as the number of iteration increases overtime for the SA modeled with the tuned parameters. In comparison to the other cooling rates and temperature, the tuned parameters converge earlier after 125 iterations. It is important to note that a temperature of 1E12 is large and makes the algorithm run slower. However, since the GOT dataset has fewer data points in comparison to other large datasets found on Kaggle, the time taken to find the optima was not an issue.

SA found a solution in 1.591 seconds which was faster than back propagation but slower than RHC. In training, SA had the same validation error percentage as back propagation with a value of 19.236%. It seems that SA was better than RHC at avoiding local optima as predicted since SA will attempt to explore more areas. With a testing error of 34.247%, SA were able to better generalize the weights than both the back propagation and RHC strategies. These weights are found in Appendix B.

Genetic Algorithms (GA):

Genetic algorithm is the last randomized optimization algorithm we used to generate weights for the neural network. GA is inspired from biology where the fittest individual, in this case weights, will be preferred to mate to generate the optimal population of weights. This natural selection is evaluated through a fitness function which was the validation error function in this case.

GA have three parameters that need to be tuned that can change how well the algorithms runs. The first is the Population Ratio, which determines how large our population is that we consider at each evaluation. From ratios of 0.10, 0.15, 0.20, and 0.25, it was found that the optimal ratio was 0.15 after 200 iterations. Considering too many candidates seems to increase the overall validation error. This makes sense as there would be so much variability in the population and the evaluation would have to compare all possible individuals at that point. While 0.15 is a smaller population ratio, it seems to achieve some diversity in each generation even though it converges slowly to the goal.

The second parameter is the Mate Ratio, which determines how much of the selected population should undergo 'reproduction,' to produce new weights that carry over characteristics of the parents. The same experiment was conducted with 0.02, 0.04, 0.06, and 0.08 mate ratios. With 200 iterations, it was found that the optimal ratio for mating was 0.02. This suggests that combining two optimal weights do not always produce the best offspring. Lastly, a Mutation Ratio was found by evaluating the GA with 0.02, 0.04, 0.06, and 0.08 mutation ratios. It was found that 0.04 was the optimal ratio which suggests that while some randomization of weights is good, too much leads to bad candidates.

GA are an interesting algorithm in that there seems to be a tradeoff with the number of iterations run versus the time it takes to converge. Table 1 below illustrates this exchange. Training and testing error were recorded for 20, 50, 100, 150 and 200 iterations. After each iteration, the time elapsed was also recorded. Initially with 20 iterations, the algorithm converges quickly with a training error of 21.29% and a testing error of 35.10%. In comparison to back propagation, RHC and SA, GA's error is only worse by a mean of 1.96% for training. In

testing with 20 iterations, the algorithm does better with a value of 35.103% which is smaller than back propagation's testing error of 36.130%. As the number of iteration increases, you can see that the training and testing error slowly decreases as the time increases, almost doubling at each iteration. It seems that with more generations, the algorithm is able to produce better candidates which reigns true with the theory behind natural selection.

Table 1. GA iterations versus Error and Time

Iterations	Train Error	Test Error	Time Elapsed
20	21.29%	35.10%	1.249s
50	21.29%	35.10%	2.863s
100	21.22%	34.42%	5.509s
150	20.93%	35.45%	8.241s
200	19.13%	34.08%	10.845s

Overall with 200 iterations, GA's produces the lowest training and testing error of 19.130% and 34.075% respectively. The time elapsed was the highest taking a total of 10.845 seconds, more than triple the amount of time taken to use back propagation on the initial neural network. The weights generated by the GA is listed in Appendix C.

Part 1 Conclusion:

To help generalize the findings found from each method, table 2 below summarizes the evaluation of each algorithm through the reported validation errors and the elapsed time. It seems that every randomized algorithm did better than back propagation when it came to minimizing testing error. RHC, SA and GA with 20 iterations were able to find these results faster than back propagation. However, back propagation was able to minimize training error better than RHC and GA with 20 iterations, while SA did equally well. It seems that RHC performed the worst out of all the randomized algorithms. This is expected since they are more likely to get stuck at local optima. GA were the best overall algorithm when they were run with 200 iterations. This algorithm reported the smallest training and testing error overall but with the highest time elapsed. This comes to show that if there is enough time permitted, GA's might be able to find a good solution. However, if you are trying to consider speed as well as accuracy, SA might be the best candidate to consider.

Table 2. Summary of Results

	Back prop	RHC	SA	GA – 20 iter.	GA – 200 iter.
Training Error	19.236%	19.530%	19.236%	21.291%	19.130%
Testing Error	36.130%	35.788%	34.247%	35.103%	34.075%
Time	3.274 s	1.574 s	1.591s	2.208s	10.845s

It is important to mention the biases for each algorithm since this also factors into how well the algorithm performs on a given data set. For RHC, there is a preference bias for local neighbors which are assumed to be 'good' if they head towards some local optima. RHC worked

well in our case because the weights for the 'best' neural network are probably similar in value. However, as shown in our experiment, RHCs can converge on a local optimum and increase their overall error. Similarly, SA prefer locality as well and have the same preference bias as RHC. However, due to the cooling and heating nature, SAs have a smaller preference to locality when compared to RHC. Lastly, GA do not have this local bias since the algorithm tries to increase genetic diversity which prevents the model from getting stuck a suboptimal solution.

It seems that unlike randomized optimization algorithms, back propagation uses knowledge of the actual output to make adjustments to the weights in the right direction. This differs from RHC, SA and GA which randomly changes weights to find the best solution thereby ignoring past information. In general, GA's will do probably do better in the long run to find the weights for a neural network, but back propagation can generalize well in a shorter time.

Part 2:

In this section of the paper, I will examine 3 different optimization problems to understand how each randomized optimization algorithm performs in different settings. I will be considering the classic Traveling salesman problem, the Knapsack problem and the Four-Peaks problem to find out the strengths and weaknesses of each algorithm in each situation.

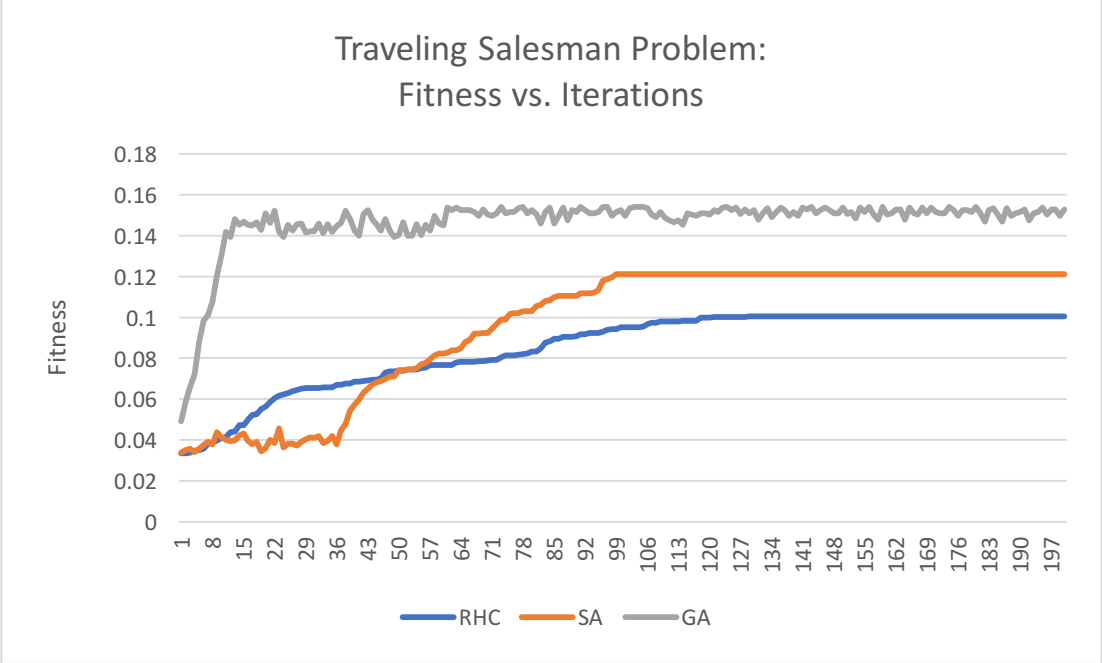
Traveling Salesman:

The traveling salesman is the most basic graph searching problem. It states that man is trying to visit all the cities exactly once in a country, but wants to do so by minimizing cost. In this case, cost refers to the distance traveled which in this case is inversed. This problem is often taught in classrooms as an NP-complete search problem since the number of paths to consider increase exponentially. However, randomized search algorithms can generally converge at a somewhat optimal solution in a relatively short amount of time.

Before running the algorithm, each model was tuned for optimal performance. RHC, Gas and SA were run with 200 iterations as the values seem to converge way before this value. GA were more computationally heavy in comparison to the other random optimization algorithms and took longer to run. The SA model was given a cooling rate of 0.95 and an initial temperature of 1E12. Lastly, the GA model used a selection population of 200, mating of 150 individuals and a mutation 20 individuals in each generation.

Using these parameters, each algorithm was run on the traveling salesman problem to compare how each function performed. Figure 4 below shows how each algorithm performed with fitness overall. It is clear to see that GA performs the best, converging on a higher fitness with a smaller number of iterations. In contrast, SA and RHC took over 100 iterations and quickly flattened out to the same fitness. This suggests that these algorithms settled on local optima and stopped considering other candidates after a certain point in time. This is more obvious when comparing the RHC and SA curves. RHC settles on a lower fitness whereas SA continues to consider other regions. SA performs better in comparison to RHC in this perspective. While it is clear to see that GA performed the best of the three algorithms in the given iterations, it is important to note that GA took longer to run than SA and RHC.

Figure 4. Traveling Salesman Fitness Curves for Each Algorithm



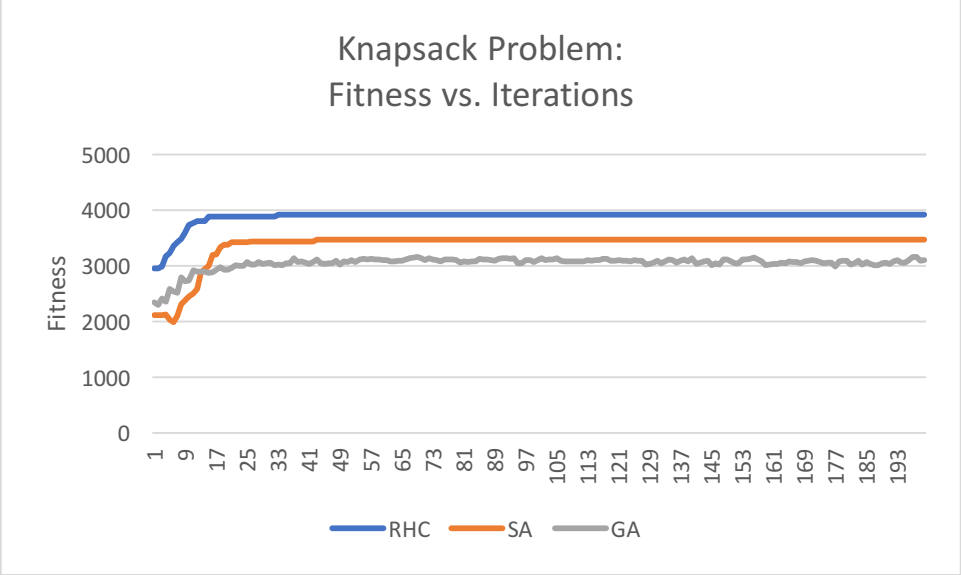
This brings us to the question of why GA outperformed the other two for the traveling salesman problem. As an NP-complete problem, the number of solutions to consider increases exponentially. However, GA has an advantage in that it performs in a non-deterministic approach and considers a variety of solutions as opposed to local solutions. This seems to be an optimal approach since over time, natural selection lends way to the best solution at each iteration. Thus, GA outperformed RHC and SA in this manner even though the other two converged on solutions faster. Thus, GA are the best way to calculate a path for the problem of Traveling salesman.

Knapsack Problem:

The knapsack problem is another classic NP-complete problem that many programmers learn about in college. Given N items with M values and K weights, the user must find a solution that maximizes the value of the knapsack while minimizing weights. There is also a restriction to how many copies of an item you can add to the knapsack. In this case, N was set to 40 with a restriction of 4 copies. The weight and value maximum were set to 50. There are many different solutions since there are usually no one solution that both has the maximum value and minimum weight. Fitness in this case if found by the maximum value with the smallest weight (or largest inverse weight).

RHC, SA and GA were run with 200 iterations to find the highest fitness over time. The same tuning parameters were used from the first problem. Figure 5 below shows the results of the experiment. All randomized optimization algorithms generally plateau after 30 iterations. But it is interesting to note that GA converges on the lowest fitness. SA does a little better with a fitness value of 3463. RHC has the maximum fitness overall with a value of 3911. In contrast to the traveling knapsack problem, GA performs the worst and RHC outperforms all other algorithms.

Figure 5. Knapsack Problem Fitness Evaluations for each algorithm



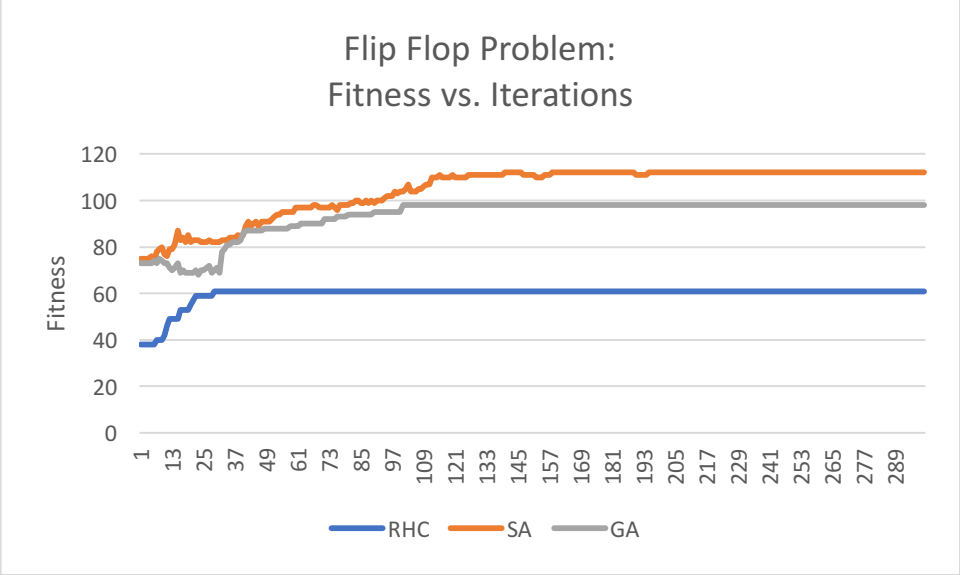
So why does RHC outperform all the other algorithms in this problem? To understand why, it is important to consider the solutions to knapsack problem. Both SA and RHC have preference bias towards locality with the idea that some answers are closer to the 'best' answer. Therefore, this bias worked out in favor for the two algorithms unlike GA which considers all possibilities. While RHC typically gets stuck at local optima, since it was able to search more of the hypothesis space with random restarts, RHC was able to surpass SA. It seems that using the knowledge learned from previous searches helped RHC find the best solution.

Flip-Flop Problem:

The flip-flop problem entails finding the number alternations in a bit string of length N. Alternating is defined as when the bit string goes from 0 to 1, or from 1 to 0. Unlike the traveling salesman and the knapsack problem, the solutions to this problem is pretty simple. The fitness in this case is when a bit-string has the largest number of alternations. A length of 80 is used to find solutions to this problem using the same tuning parameters described in the first two problems. Each algorithm was run with 300 iterations by which all functions converged on a fitness. Figure 6 below demonstrates the relationships amongst the different algorithms

In this case, SA was the best performer converging on a higher fitness value when compared to GA and RHC. GA was the second best where as RHC scored the worst. It is worth noting that RHC converged faster than the other two algorithms. This is probably due to the reason that RHC found a local minimum and stayed on this path. GA on the other hand only performed slightly worse than SA. These relationships can be seen in figure 6 below.

Figure 6. Flip-Flop Fitness Evaluations for each algorithm



To understand why SA performed better than GA, it is important to remember the distinction of the problem. SA was able to reach a higher fitness with more iterations in faster time. This is because GA are computationally heavy which already puts SA and RHC in an advantage. With the flip-flop problem, since the solutions are so simple, using GA is not going to give you a higher advantage. Locality was more important which is why SA outperformed GA. However, RHC should still be as fast. But with SA, unlike RHC, the algorithm did not settle in a local optima due to its heating and cooling rates. These factors of the problem helped SA perform better than the other random optimization algorithms.

Part 2 Conclusion:

Overall, it seems there are some clear tradeoffs between the three problems. The large space of the knapsack problem preferred RHC to other optimization algorithms. The simple nature of the Flip-Flop problem gave preference to SA since the algorithm searches more regions with a locality preference. Lastly, the large space search and non-deterministic characteristics of GA allowed it to outperform the others in the traveling salesman problem. The final rankings with respect to the problems for each algorithm are showed in table 3 below.

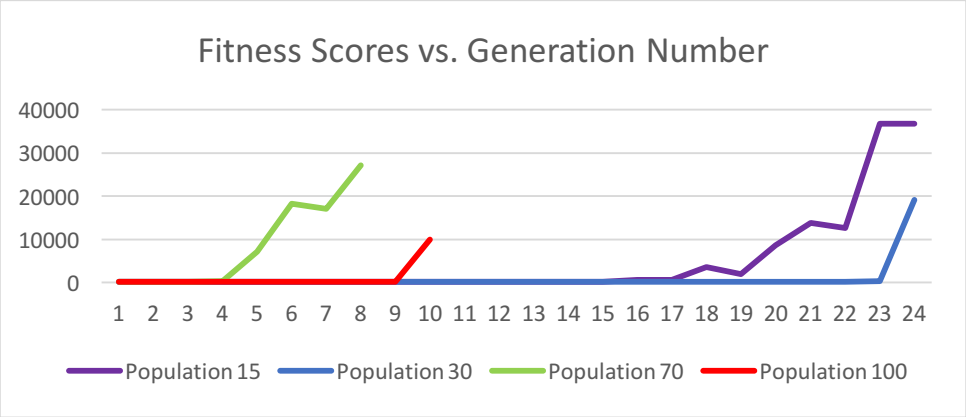
Table 3. Ranking of all Randomized Optimization Algorithms

	Traveling Salesman	Knapsack	Flip-Flop
RHC	3	1st	3
SA	2	2	1st
GA	1st	3	2

Part 3: Flappy Bird

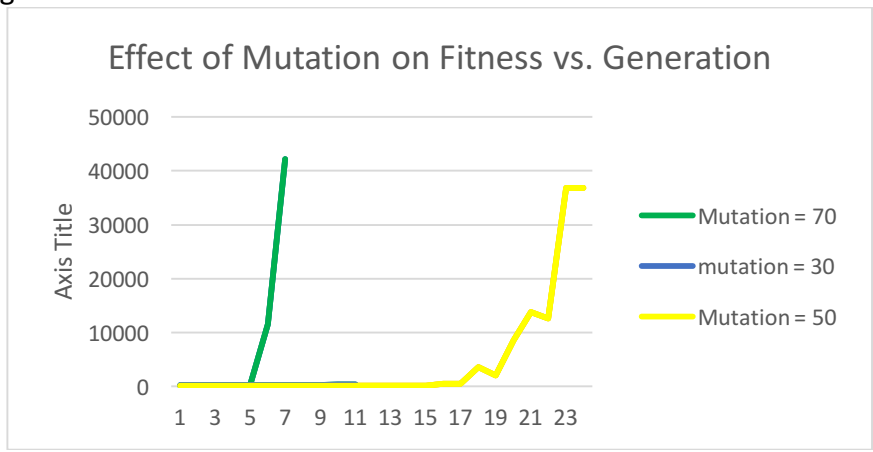
For the last part of this paper, I will conduct an analysis of a genetic algorithm written to play the game Flappy Bird given by the git repository from TsReaper. This was used as opposed to the repository given in class since the dependency 'tensorflow' would not work with my Mac's higher OS. Three main variables were altered to see how the gameplay was effected by each parameter. These variables are population size, mutation chance and survival rate. Data from each run is recorded in the excel sheet titled 'flappy' in the Assignment2 folder.

Figure 7. Population size as it affects Fitness Score versus Generation Number



The first variable adjusted was population size. Population size refers to the number of individuals created after each generation. Mutation chance and survival rate were kept at 50 and 30 respectively as population was increased from 15, 30, 70 and 100. Figure 7 above shows the level of fitness scores of the genetic algorithm after every generation. Once the program found a solution, it would never end. To save time, if the generation ran longer than 10 minutes, the run was stopped and the value was recorded. It is easy to see that a population of 15 had the higher fitness score but it reached its value slower and with more iterations. But a population of 70 converged at a solution faster since it considered more candidates. This is due to the fact that with more candidates, the GA was more likely to find a 'good' bird. However, with too many candidates as is the case with 100, the solution has too much options to consider.

Figure 8. Mutation as it affects Fitness Score versus Generation number



The second variable that was adjusted was mutation chance. Figure 8 above shows the effect of increasing mutation percentage at each generation. The population was kept at 70 to keep fast convergence and the survival rate was still set to 30%. It seems that a lower mutation led to homogeneous candidates so the level of fitness was very low. This is seen with a mutation chance of 30%. The higher the mutation chance, the faster the population found a 'good' candidate. It seems that initially, there is a huge need for mutation to increase diversity of the population considered. As time goes on, this can actually decrease fitness as can be seen with a 50% mutation chance.

Lastly, the variable survival rate had an overall predictive outcome that did not alter the fitness as much. In low survival ratings, the algorithm took forever to find an 'optimal' bird since there was a small number of candidates considered. The population became homogeneous a lot faster. As the survival rate was increased to a high, the GA was very slow to converge on an answer. Values within the middle range provided higher fitness overall.

In conclusion, variation of the three parameters greatly affected the gameplay. Smaller population sizes, higher mutation chances and a moderate survival resulted in higher fitness for the flappy bird game. Using these generalizations, an optimal solution was found to be population size of 15, mutation chance of 70% and a survival rate of 30%.

Works Cited

1. https://github.com/pushkar/ABAGAIL
2. https://gist.github.com/mosdragon/ad893f877a631260e3e8
3. https://github.com/TsReaper/AI-Plays-FlappyBird:
    a. https://tsreaper.github.io/AI-Plays-FlappyBird/

Appendices

A) RHC Weights for NN:

0.331907, -0.246290, 0.290881, -0.497371, 0.187767, 0.323700, -1.402268, -0.962836, 0.557026, 0.689451, -2.823096, -0.755356, -0.122276, -0.680624, 4.073774, 2.347666, -0.153548, -0.870833, 0.484188, -3.592221, -0.883058, -0.471724, 1.593502, 0.906931, 0.588834, 0.511061, -0.397304, -0.094136, -5.453136, -0.235691, -0.261133, -0.073662, 0.169131, 0.122631, 0.006256, -1.663339, -0.053554, -0.459834, 0.018332, -0.065050, 0.148585, 0.033324, -0.685448, 0.088712, 0.417468, 1.620015, 0.249221

B) SA Weights for NN:

-0.266675, -1.725611, -0.073969, -0.215968, 0.755925, -0.761852, -2.952334, -1.124087, -4.032363, -3.330193, -2.029161, -1.466784, -1.787783, -0.890505, -1.598088, -2.900152, 2.007804, -1.113678, 1.560530, -2.131306, 0.080957, 2.047145, 0.953199, -0.510286, -3.240890, 1.254855, 0.138868, -1.351291, -0.944408, 2.195479, 0.814393, 0.159571, 0.477226, -2.785692, 1.941488, -2.180443, 1.411187, 1.042300, 0.681623, 1.088645, 0.754582, 1.131819, -2.277459, -1.247596, -0.402075, -0.534943, -0.829382

C) GA Weights for NN:

0.018517, 0.114282, -0.355607, -0.262842, 0.483515, -0.476889, -0.339248, -0.459125, -0.439188, 0.025598, -0.353328, 0.313322, 0.702445, -0.186623, 0.434381, -0.453084, -0.354377, 0.306995, 0.432769, -0.079622, 0.224055, -0.164710, 0.246136, -0.105400, 0.458694, 0.046162, -0.121095, 0.217111, 0.020268, -0.484609, 0.235171, -0.081934, -0.073110, -0.347933, 0.258435, 0.147370, 0.362621, -0.156874, -0.199462, -0.169135, -0.060179, -0.513159, 0.075170, 0.325768, 0.416589, 0.185837, 0.491699