Project 2 – Supervised Learning Report

Part 1: Optimization Algorithms with a Neural Network

Introduction: The objective of this section is to find good weights for one of the neural networks from Assignment 1, using **Randomized Hill Climb (RHC)**, **Simulated Annealing (SA)** and **Genetic Algorithm (GA)** respectively. I have chosen to use the neural network I created for the UCI repository white wine quality dataset[1]:

To recap, the white wine quality dataset originally has 12 real attributes, with 4898 instances and an output class containing 11 members. In the course of attribute selection with WEKA's ***CfsSubSetEval*** attribute evaluator, 8 significant attributes were selected and normalized.

In Assignment 1, the resulting neural network for this dataset had 8 nodes in its input layer, 19 nodes in its hidden layer and 1 output node, and was trained in 500 iterations/epochs. To ensure unbiased comparison, I configured the neural network in this section to mirror all of the foregoing settings.

Implementation Methodology: All the experiments described in this report were carried out using the **ABAGAIL** Machine Learning Library[2], with a few slight modifications (see the *README.txt* file for details).

Specifically, ABAGAIL's ***AbaloneTest.java*** class was adapted for use with the white wine dataset. As previously stated, the neural network was set up with 8 input nodes, 19 hidden nodes and 1 output node. Also, the normalized dataset was split such that 75% of the data was used for training, and the remaining 25% was used as a test set.

The neural network was then trained with RHC, SA and GA in increasing numbers of iterations (from 1 to 5000) respectively, with the training times recorded for analysis. Each trained neural network was also tested, and the classification error rates were also recorded and analysed.

Analysis: **Figure 1** below shows a plot of both the accuracy of the test sets and the training time for the 3 algorithms against the number of iterations. From the accuracy plot, it is clear that at around 1000 iterations, all the algorithms had converged to about 84% accuracy. However, the manner in which the different algorithms converged is markedly different, and will be further explored below.

Also, the training time plot shows that both RHC and SA require almost identical times to train the neural, network, and do not take too long to train. For GA on the other hand, training time is very high, and grows rapidly as the number of iterations grows.

The following sections are a brief analysis of the performance of each algorithm:

- **Randomized Hill Climb (RHC)**: As the name implies, RHC attempts to locate global optima by randomly picking a point and checking if its output is greater than previously selected points. This randomization is aimed at reducing the chances of landing on and selecting local optima values. However, the dip in the RHC line in the left chart in **Figure 1** shows that RHC is still susceptible to hitting local optima. It appears that RHC found a local optima at around 100 iterations, and this would have led to poor neural network weight selection had training been limited to only 100 iterations

[1] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.
Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.
[2] Absolute Best Andrew Guillory Artificial Intelligence Library (ABAGAIL) by Andrew Guillory http://abagail.readthedocs.io/en/latest/index.html

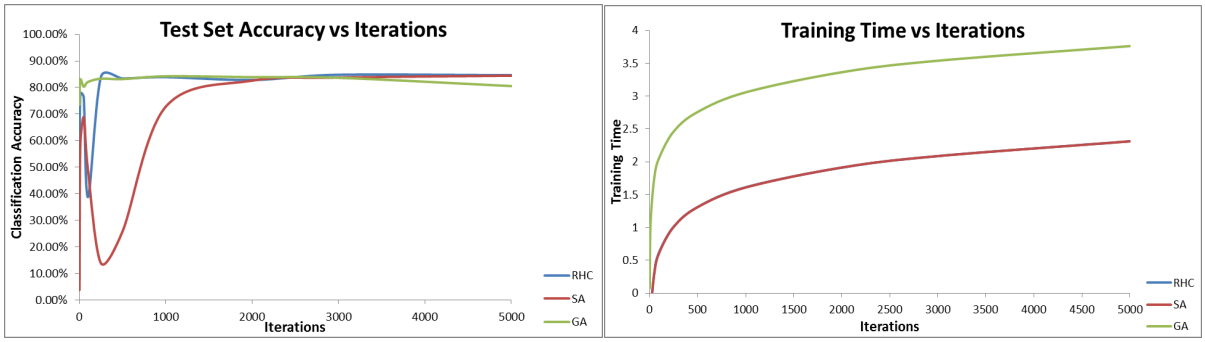

Figure 1: Classification Accuracy and Training Time of Optimization Algorithms vs. Number of Iterations

Also, it is worth noting that RHC is quite efficient it terms of training time, as is shown above (the RHC and SA curves overlap).

- **Simulated Annealing (SA)**: SA is similar to RHC, but instead of the "greedy exploitation" principle used by RHC, SA seeks to both "explore" and "exploit". SA not only searches in the direction indicated by its best previous results (exploiting), but also checks the direction of not-so-good results (exploration), in the hope of avoiding getting stuck in local maxima that appears promising. This mechanism, while effective, does not make SA completely invulnerable to selecting local optima, as is shown in **Figure 1** above. Similar to RHC, SA appears to have landed on a locally optimal set of weights.

  There is an interesting contrast to be drawn between SA and RHC here. **Figure 1** shows that RHC quickly "escaped" the local maxima, given more iterations. SA on the other hand, due to the exploratory nature of its operation, took a noticeably larger number of iterations to "recover" from the effects of the local maxima's basin of attraction. As such, SA converged in about 1000 iterations, while it took RHC only 250 iterations to converge. This behaviour can easily be remedied by reducing the temperature setting in the SA algorithm, but I thought that this was an interesting contrast worth highlighting.

  Similar to RHC, SA is also relatively efficient in terms of training time.

- **Genetic Algorithm**: The first glaring difference between GA and the previously discussed algorithms is that it appears to be much less susceptible to getting trapped in a local maxima than the previous 2 algorithms, as shown in **Figure 1**. At the point in the graph where RHC and SA classification accuracy dipped, GA hardly dipped at all.

  GA also converged after only 50 iterations. This is attributable to GA's mechanism of using crossover to select the best attributes from previously seen points, and combining them to improve its future results. The cost of this hyper-efficient convergence however, is the amount of time it takes to train the neural network with GA. To illustrate, it took GA the same amount of time to complete 250 iterations as it took the other algorithms to complete all 5000 iterations. This time complexity exhibited by GA is due to the intensive computations involved in its crossover calculations.

- **Comparison with Back Propagation and Conclusion**: **Figure 2** below shows the learning curve for the original neural network (using back propagation), vs. those of the 3 optimization algorithms. Varying sizes of training data was used to train the neural network, and the respective classification accuracies were recorded. Note that all the training was done at 500 iterations (as was the case in assignment 1), to allow for accurate comparison.

  Recall that the best results obtained in Assignment 1 for the wine set classification dataset was approximately 48%. It is clear from **Figure 2** that all the optimization algorithms produce much better

results than back propagation (at these settings). Only GA performs worse than back propagation, for reasons already explained above.

Also, the algorithms do not seem to be overfitting the data, as there is little or no appreciable effect when more training data is used to train the neural network.

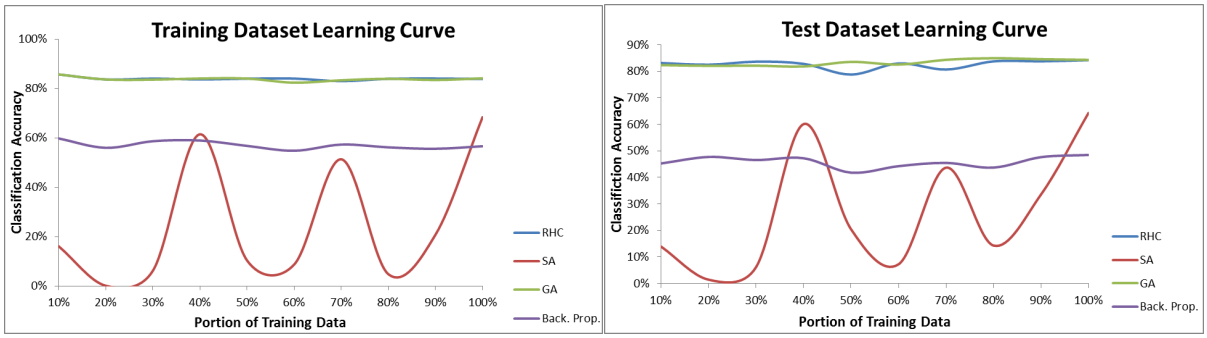

Figure 2: Learning Curves – Back Propagation vs. Randomized Optimization

Part 2: Analysis of Three Optimization Problems

Introduction: The objective of this section is to investigate the performance and relative strengths and weaknesses of the 3 algorithms we have already considered (RHC, SA and GA), and a fourth algorithm known as **Mutual-Information-Maximizing Input Clustering (MIMIC)**, on 3 optimization problems. My intention is to choose problems that show off the strengths of each algorithm, and to that end the optimization problems I chose are the **Knapsack** problem, the **Travelling Salesman** problem, and the **Continuous Peaks** problem. Fortunately each of these problems already exists in the ABAGAIL library, and I only had to make slight modifications to the existing classes to obtain the data analysed below.

The Knapsack Problem

- **Introduction**: The Knapsack problem is a classic NP-hard optimization problem. Given a knapsack with a maximum weight capacity, and an array of objects with different weights and values, the objective is to determine the combination of objects to put in the knapsack to maximize value, while staying at or below the maximum allowed knapsack weight.

- **Analysis**: ABAGAIL's ***KnapsackTest.java*** class was configured with 40 objects (with each object having 4 instances), and a knapsack with a maximum weight capacity of 3200. The program was then run in increasing numbers of iterations (from 1 to 5000), with all 4 algorithms. The resulting fitness values, runtime and number of function evaluations for each iteration/algorithm combination were then recorded and analysed.

  Figure 3 below shows plots of fitness values for the 4 algorithms against number of iterations, and also plots of the logarithm of execution times against number of iterations. From the graphs, it is clear that the most successful algorithm here (i.e. the algorithm with highest fitness value) is MIMIC, closely followed by GA, while SA and RHC have about the same level of performance. Also note that MIMIC requires fewer iterations than any of the other algorithms to converge. Furthermore, the time complexity chart shows that, as expected, MIMIC is far and away the most expensive algorithm in

terms of time, with GA not far behind. As experienced in the previous section, both RHC and SA are much more time efficient.

The reason that MIMIC is so successful with the Knapsack problem lies in the manner in which the MIMIC algorithm works. MIMIC estimates a probability density during every iteration, based on the probability density and results from the previous iteration. It then uses this new probability distribution to estimate the results for the current iteration, and so forth.

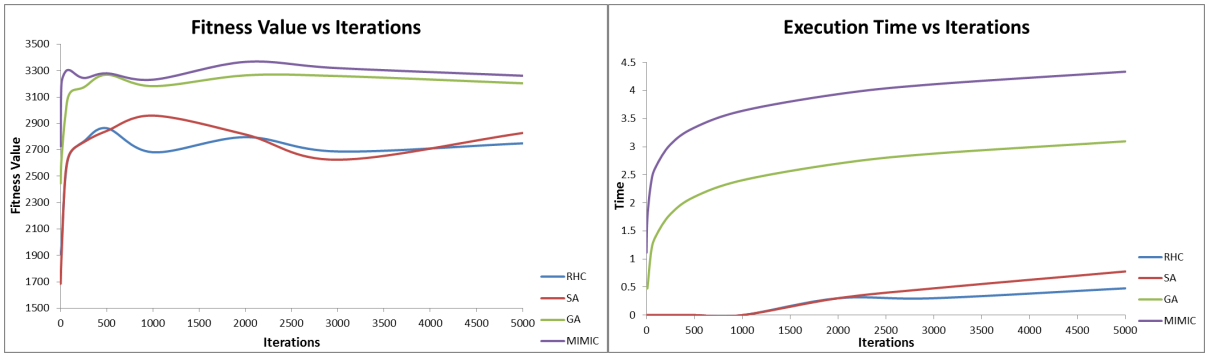

Figure 3: Fitness Values and Execution Time vs Iterations – Knapsack Problem

With its set of known objects and weights, the Knapsack problem basically comes with a built in probability distribution that plays perfectly to MIMICs strengths. In other words, MIMIC thrives on optimization problems that have a discernible order or structure, as it can leverage on this structure to build a relatively accurate probability distribution right from the first iteration.

The next question to consider is - why does MIMIC take so long to execute? The left half of **Figure 4** below shows a plot of the logarithm of the number of evaluation function calls for each algorithm against the number of iterations. The evaluation function of an optimization algorithm is the function that the algorithm calls to determine the suitability or otherwise of each potential solution that it considers. The number of calls to the evaluation function is therefore a good proxy to measure how many points each algorithm examines during each iteration.

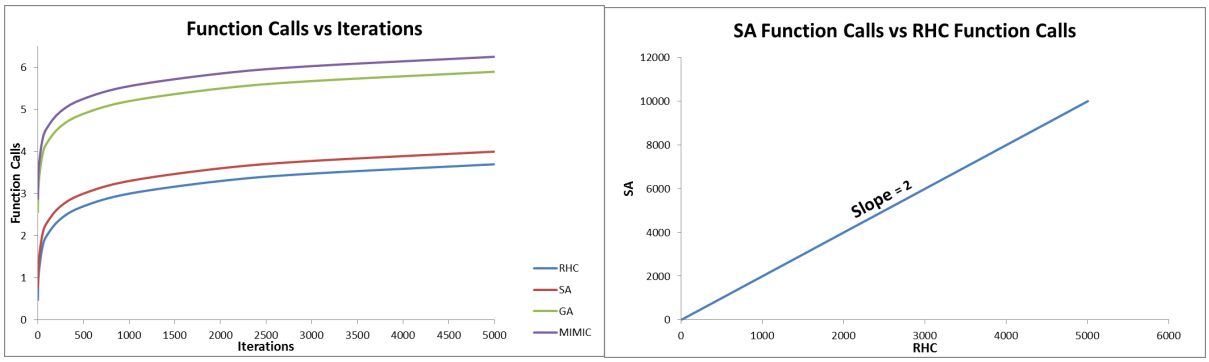

Figure 4: Evaluation Function Calls vs Iterations, & SA Function Calls vs RHC Function Calls – Knapsack Problem

As the chart shows, MIMIC evaluates far more potential solutions in each iteration than any of the other algorithms. This accounts for the time complexity of MIMIC. The number of function calls for the other algorithms also accounts for their respective time complexities.

Finally, the right half of **Figure 4** represents an interesting relationship that I believe is worth highlighting. It is a plot of the number of function calls for SA against those for RHC, for different numbers of iterations. The line in the chart passes through the origin, and has a slope of 2. This means that for any given number of iterations, SA performs twice as many evaluation function calls (and consequently examines twice as many potential solutions) as RHC.

Upon further consideration, this makes perfect sense. RHC randomly picks a single point in each iteration, checks if it provides a better output than its current best result, and then moves on to another point in the next iteration. SA on the other hand considers a point on both sides of its current point in each iteration, in order to fulfil its "explore and exploit" mandate. SA therefore examines twice as many points as RHC in each iteration.

The Travelling Salesman Problem

- **Introduction**: This is another classic NP-hard optimization problem. Given a scenario where a travelling salesman must travel between N cities and end up at the original city, and given that we do not care in what order the salesman visits the cities, the aim is to find the optimal sequence of visits such that the distance covered by the salesman is minimized.

- **Analysis**: ABAGAIL's ***TravellingSalesmanTest.java*** class was configured such that there are 100 cities to visit. The program was then run in increasing numbers of iterations (from 1 to 5000), using all 4 algorithms, and the fitness values, runtime and number of function evaluations recorded are analysed below.

  Figure 5 below shows plots of fitness values for the 4 algorithms against number of iterations, and also plots of the logarithm of execution times against number of iterations. It is clear from the graph on the left that GA outperforms the other algorithms in this problem. It converges quicker than the other algorithms, and also has by far the highest fitness values. The other 3 algorithms on the other hand appear to converge towards the same range of fitness values. MIMIC still has the greatest time complexity, but in spite of this it is unable to beat GA on this problem.

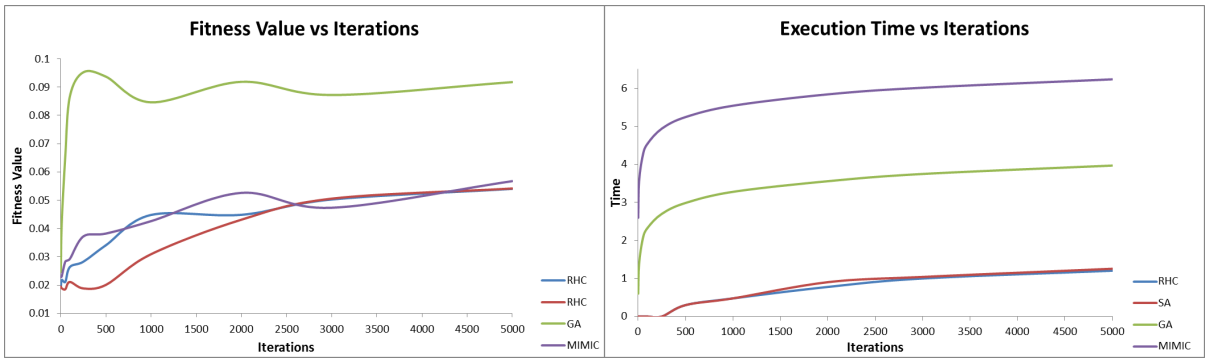

Figure 5: Fitness Values and Execution Time vs Iterations – Travelling Salesman Problem

From these charts and given our domain knowledge, it is clear that here is no implicit relationship or structure that binds the 100 destinations; hence MIMIC finds it difficult to make accurate probabilistic selections. GA on the other hand seems able to quickly converge towards an efficient ordering of the destinations by using the lessons it learnt during previous iterations.

The left chart in **Figure 6** below again shows the relationship between the number of evaluation function calls and iterations for each algorithm. The plotted function calls values are logarithmic, to accommodate the wide range of values. In this problem, GA evaluates more options proportionally than it did in the Knapsack problem, but MIMC still evaluates many more potential solutions, especially as the number of iterations increases.

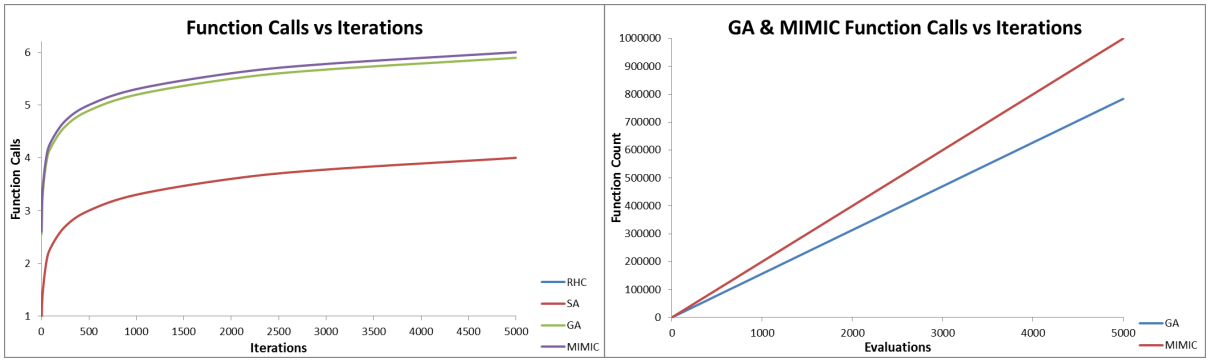

Figure 6: Evaluation Function Calls vs Iterations; GA and MIMIC Function Calls vs Iterations – Travelling Salesman Problem

Also, just like in the Knapsack problem, SA generates twice as many evaluation function calls as RHC, for the same reason as earlier describe. This relationship is therefore not plotted, as it is identical to that in the right-hand chart in **Figure 4**.

Instead, the right-hand chart in **Figure 6** depicts another interesting observation. It shows that for both MIMIC and GA, the number of points evaluated increases at a constant (albeit different) rate with the number of iterations respectively, regardless of how high the number of iterations goes. In other words, for both algorithms there is a linear relationship between the number of points evaluated and the number of iterations. In fact, this relationship holds true for all 4 algorithms in both the Knapsack and Traveling Salesman problems.

The Continuous Peaks Problem

- **Introduction**: The continuous peaks problem involves finding the global maxima among a collection of many local maxima. The primary objective here therefore is to avoid getting stuck in one of the many local optima, and arriving at the global optima as efficiently as possible.

- **Analysis**: ABAGAIL's ***ContinuousPeaksTest.java*** class was configured to generate 100 peaks, with one global maxima. The program was then run in increasing numbers of iterations (from 1 to 5000), using all 4 algorithms. The following is an analysis of the results obtained.

  Figure 7 shows the same graphs as were plotted for previous problems. At lower number of iterations, MIMIC performs better than the other algorithms. However, as the number of iterations approaches and exceeds 8000, MIMIC is overtaken by SA.

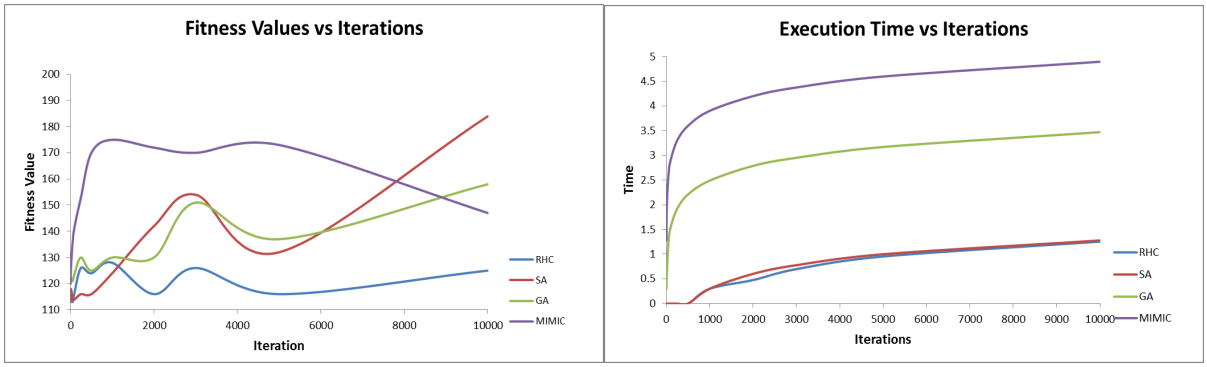

Figure 7: Fitness Values and Execution Time vs Iterations – Continuous Peaks Problem

I am unable to explain MIMICs initial dominance using my current understanding of MIMIC, as there is no underlying structure in this problem for MIMIC to take advantage of. However, at higher iterations, SA performs better than MIMIC, taking advantage of its exploratory abilities to escape falling into local maxima. It is also no surprise that MIMIC has the highest time complexity, followed by GA, with RHC and SA showing much better time efficiency (as usual).

Plotting the function call vs iterations graph as shown in **Figure 8,** we see that in this problem the number of options evaluated by each algorithm is much closer than for the previous 2 problems. Also in this problem I was surprised to discover that the previously established relationship between the number of function evaluation for RHC and those for SA is not obeyed. Instead, both SA and RHC have essentially the same number of function calls. Similarly, both MIMIC and GA do not obey the previously established linear relations between function calls and iterations.

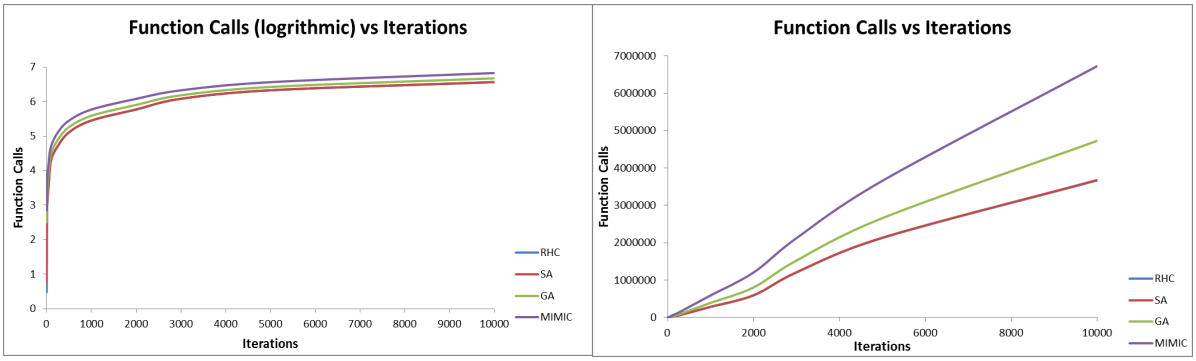

Figure 8: Logarithmic and Normal Evaluation Function Calls vs Iterations – Continuous Peaks Problem

Again, I am not sure what it is about the Continuous Peaks problem that causes this deviation from the behaviour of the previous problems.

Conclusion

I have taken away a lot from this assignment. For starters, I have a much better understanding of the strengths and weaknesses of the 4 algorithms that were examined. I also better understand what kinds of problems each algorithm is best suited to handle.

Table 1 below summarizes the performance of the 4 algorithms on the 3 different optimization problems, in terms of their fitness values. MIMIC performed quite well on all the problems, but is limited by the fact that is extremely expensive in terms of time. GA also performed well on the problems, and is also has a relatively high time complexity (although not as high as MIMIC).

Table 1: Summary of Results of Optimization Problems Investigation – Fitness Values

Position	Optimization Problem		
	Knapsack	Travelling Salesman	Continuous Peaks
1st	MIMIC	GA	SA
2nd	GA	MIMIC	MIMIC
3rd	SA	RHC	GA
4th	RHC	SA	RHC

Table 2 shows the performance of the algorithms in terms of time complexity. SA and RHC were consistently proven to be the most timely/quick of the 4 algorithms, with GA a distant third and MIMIC by far the most time consuming algorithm. However, considering time complexity in isolation is misleading, as being fast does not translate to being correct. Instead, one should base ones choice of algorithm on the accuracy requirements of the scenario, as well as the amount of time available. This trade-off between speed and accuracy is another important lesson I have learned from this assignment.

Finally, I was able to show some evidence of a relationship between the number of function calls incurred by RHC and SA. However, the results from the Continuous Peaks problem muddied the waters a bit, and given more time I would like to investigate this relationship further with more optimization problems.

Table 2: Summary of Results of Optimization Problems Investigation – Time Complexity

Position	Optimization Problem		
	Knapsack	Travelling Salesman	Continuous Peaks
1st	RHC	RHC	RHC
2nd	SA	SA	SA
3rd	GA	GA	GA
4th	MIMIC	MIMIC	MIMIC