**Introduction**

Randomized optimization methods are commonly used for problems where 1) the input space is intractably large, 2) the fitness space is not convex, and 3) the fitness function is not differentiable or continuous. We will explore the strengths of four randomized optimization methods (randomized hill climbing, simulated annealing, genetic algorithm, and Mutual Information-Maximizing Input Clustering) on optimizing the weights of a neural network, and 3 other optimization problems.

Randomized hill climbing (RHC) is an algorithm that repeatedly generates a neighbor by randomly varying the input parameters. If the neighbor’s fitness value is greater than the current fitness value, the algorithm replaces the current input with that neighbor. A clear disadvantage of this algorithm is the fact that it can become stuck in local optima.

Simulated annealing (SA) is an algorithm of a similar flavor, but instead of always rejecting neighbors that have a worse fitness value, there is some probability of accepting that neighbor as a replacement for the current input. That probability is related to the hyperparameter T, which decreases with each iteration. In the early iterations, T is high and the probability of accepting a bad neighbor is high. In the later iterations, T is low and the algorithm essentially becomes simple randomized hill climbing. This behavior of accepting bad neighbors gives the algorithm a chance to escape local optima and converge at a better optimum.

Genetic algorithms (GA) is an algorithm inspired from biology. A population of inputs are kept, and at each iteration pairs of the population are randomly “mated” together to create children. Each child will inherit a block of its parameters from one parent, and the remaining from the other parent. In addition, at each iteration a random portion of the population will have its parameters “mutated” – which is to say that a random parameter would be varied. At the end of each iteration, only the inputs with the best fitness values would be kept in the population. This mating behavior allows parameters that synergistically depend on each other to significantly improve the fitness values to persist through the iterations.

Instead of keeping the best individual inputs, Mutual Information-Maximizing Input Clustering (MIMIC) is an algorithm that iteratively generates a probability distribution of good solutions from a population of inputs. At each iteration, a new population of solutions is created by sampling from the probability distribution of the last iteration. This allows MIMIC to estimate regions in the input space that have good fitness values, and iteratively narrow down to the regions that give the best fitness values.

**Dataset**

The dataset used to train the neural network contains 1599 instances of red wine reviews and its physiochemical composition [1]. In particular, the red wines sampled were variants of the Portuguese “Vinho Verde” wine. There are many facets to consider when reviewing wine, including its grape variety, where it was grown, how long it had been aged, and so on. In this dataset, there are 11 objectively measured features, including acidity, amount of chlorides, amount of sulphates, and alcohol content. The features in the dataset were all normalized to zero mean and 1 standard deviation. The reviews were aggregated into two classes: good and bad. The class aggregation was based on the median of the reviews, which cut the data into roughly equal halves.

**Methods**

Since the MIMIC algorithm implemented only takes discrete values, we only used RHC, SA, and GA to train the neural network. We performed 5-fold randomized cross validation, randomly splitting 10% into a test set and training on the remaining 90%. We also applied all four optimization methods to 3 optimization problems: count ones, knapsack, and the traveling salesman problem.

For the methods with hyperparameters, we performed experiments while varying some of the appropriate hyperparameters. For SA, we varied the cooling rate. For GA and MIMIC, we varied the population size. In each experiment, we performed 5 trials for a fixed number of iterations. We then took the trial with the largest fitness at the end of the iterations to represent each experiment.

All algorithms were implemented in the ABAGAIL library, with slight modifications to count the number of function calls made [2].

**Results and Discussion**

**Neural Network Optimization**

In assignment 1, the network architecture that was determined to be most ideal was a network of 4 hidden layers, each with 25 hidden units. The training and testing accuracy with each training epoch is shown in figure 1. At the end of the training, the training accuracy was 0.989, and the testing accuracy was 0.767.

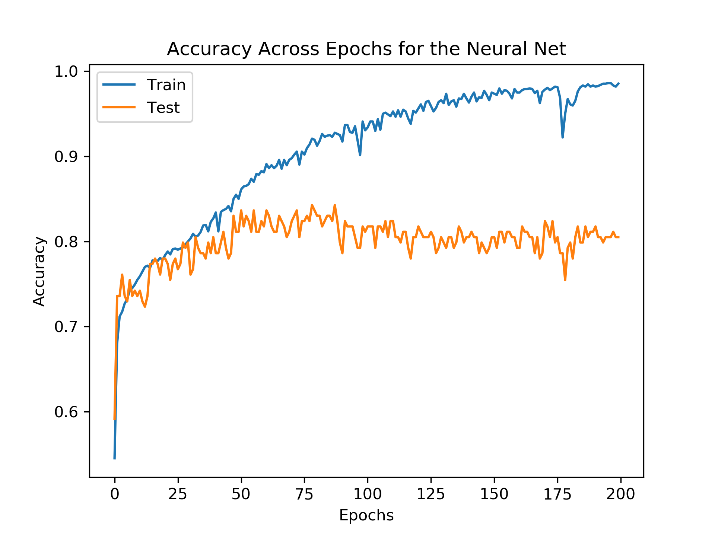


Figure 1:Training and testing accuracy using backpropagation for the neural network

Neural networks are typically trained through backpropagation, which calculates the gradient of the error at each layer and updates the weights through gradient descent. However, the weights can also be updated through randomized optimization.

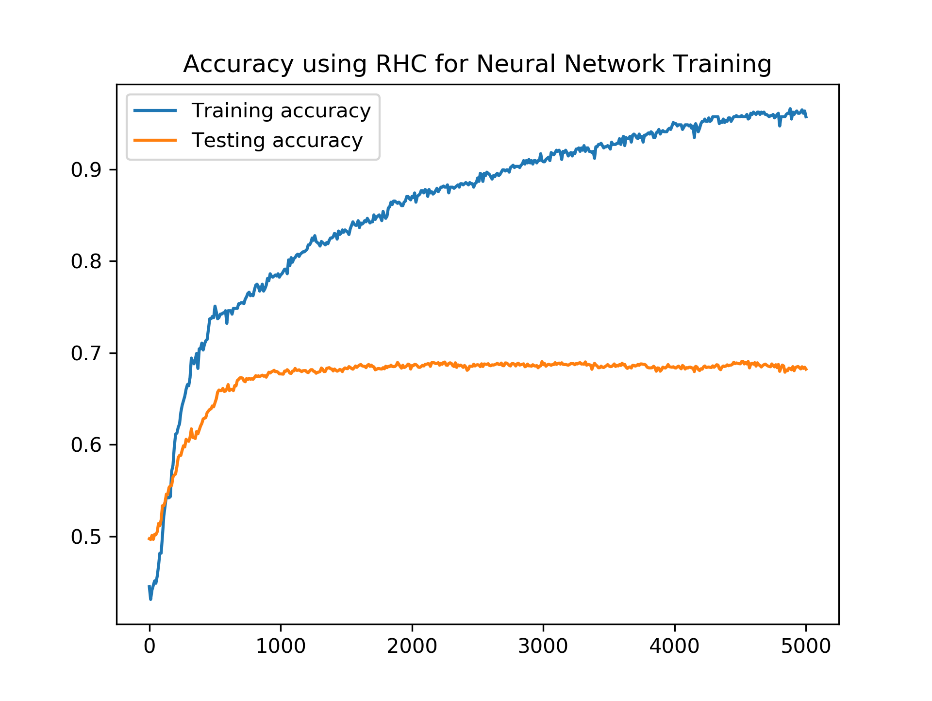


Figure 2:Training and testing accuracy using RHC

Figure 2 shows the training and testing accuracy of RHC. As the number of iterations grow, the training accuracy approaches 1, but the testing accuracy reaches a steady state of approximately 0.67.

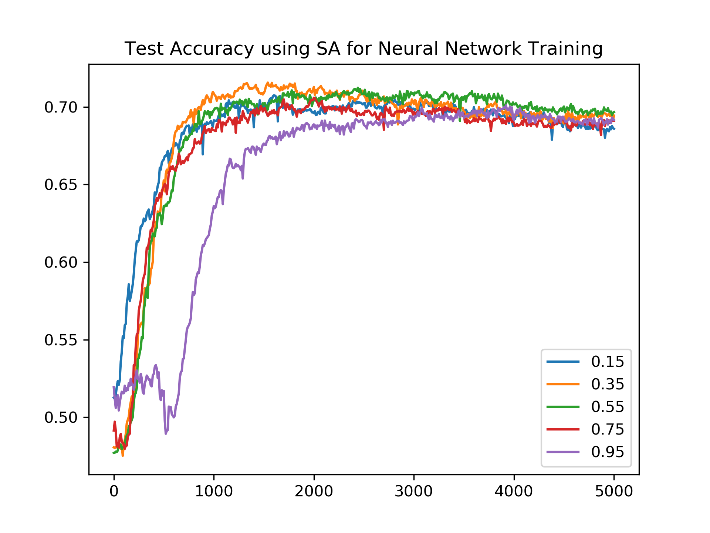
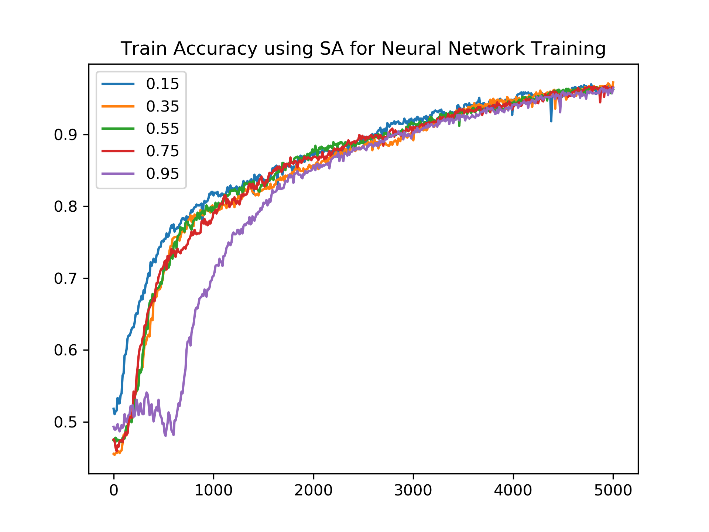


Figure 3: Training (left) and testing (right) accuracy using SA

Figure 3 shows the training and testing accuracy of SA for various cooling rates. We can see that for the slowest cooling rate of 0.95, there is a drop in training accuracy in the early iterations which is indicative of the algorithm accepting a worse neighbor. As number of iterations grows, the training accuracy approaches 1 just as in RHC, and the testing accuracy also reaches a peak of approximately 0.71.

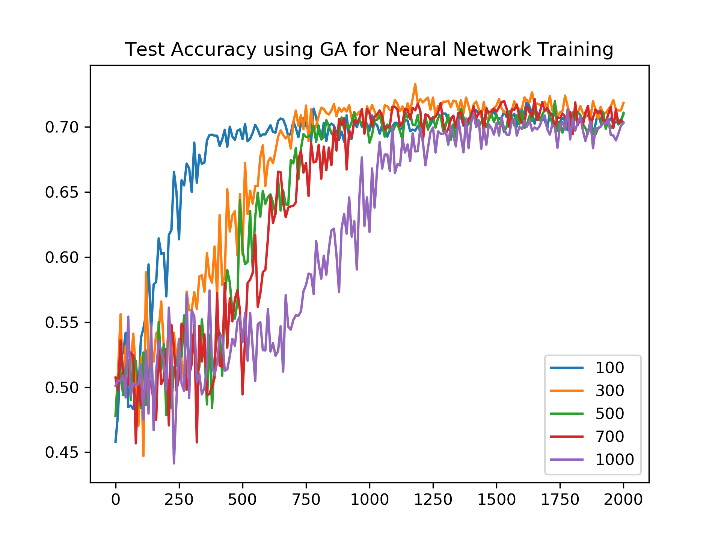
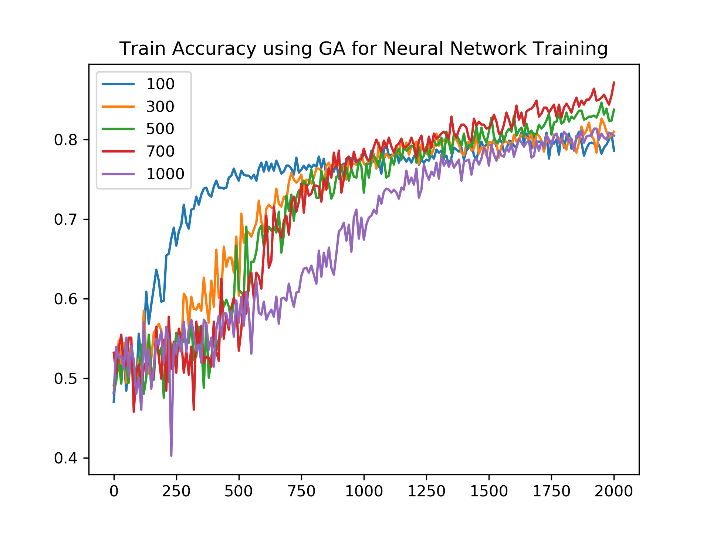


Figure 4: Training (left) and testing (right) accuracy using GA

Figure 4 shows the training and testing accuracy of GA for various population sizes. The training error does not completely approach 1, but does appear to be on the upward trajectory. On the other hand, the testing error has already reached a steady state around 0.7 accuracy.

The testing accuracy of all the optimization methods were not as good as backpropagation, which can be expected. Backpropagation makes use of gradient descent, which captures more information about the shape of the fitness space and feeds it back through the layers to update the weights accordingly. RHC and SA rely on randomly tuning the weights independently, and can easily get stuck in local optima. We can see that SA performs a little better than RHC, possibly due to the chance for SA to escape its local optima in the early iterations of the algorithm. GA works well when a set of parameters together improve fitness significantly, but it is not clear that the parameters have such dependency. The poor performance of GA supports the lack of this particular dependency. When training a neural network, it seems that backpropagation is the preferred method.

**Count Ones Problem**

*Description:*

The Count Ones problem is a simple one: given an array of bits x, how many 1’s are there in the array? In its optimization form, we are trying to find the maximum number of 1’s in the array by changing the values in each element of the array. We can see that 1) the parameters of the problem (the bits of the array) are independent of each other; 2) there is only one global optimum (the array where every bit is 1); 3) there are no local minima (for every combination of bits that’s not the global optimum, there is at least one way to flip a bit such that the fitness function will improve).

*Results:*

For this problem, the input was an array of size 100, and we can see from figure 5 below that all the algorithms converge at the optimal fitness. We can also see that MIMIC arrives at the best fitness in far fewer iterations than the other algorithms.

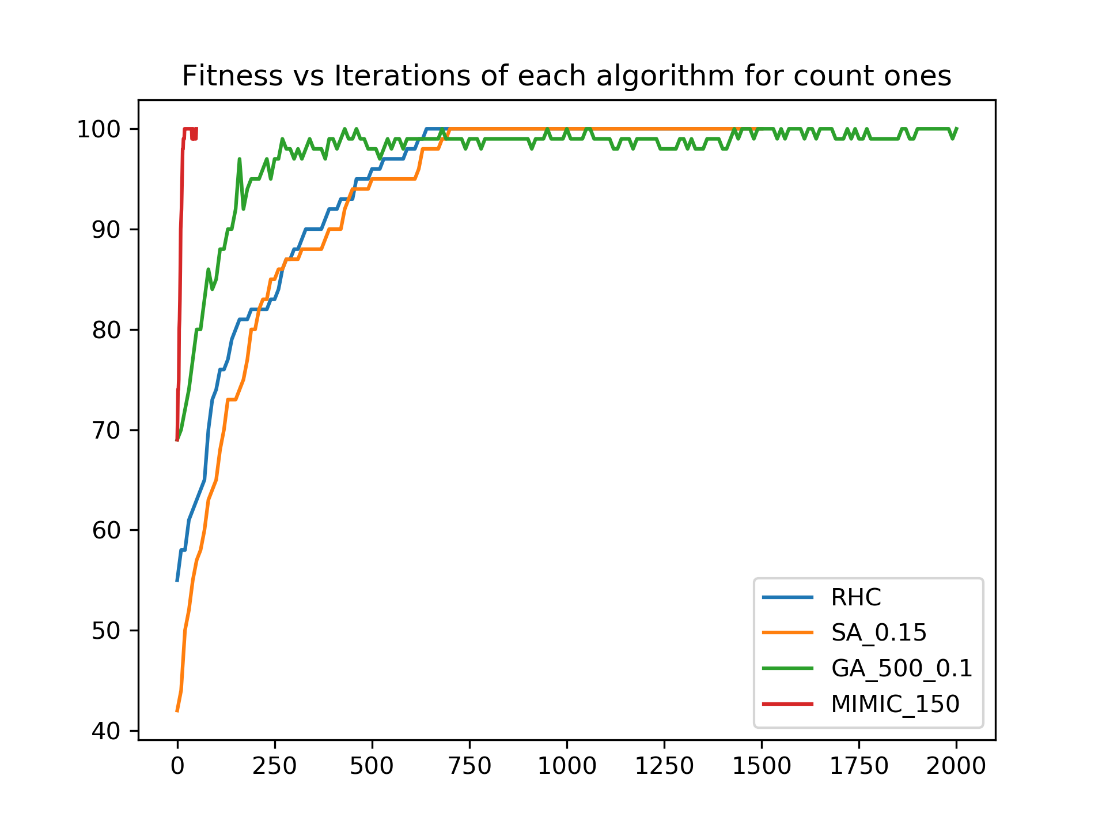


Figure 5: Fitness vs iterations for each algorithm on the Count Ones problem

However, the MIMIC algorithm computes a large number of fitness functions per iteration while performing sampling to generate its population. To get a better sense of how efficient the algorithm is, we can compare fitness by fitness function evaluations instead of iterations. In figure 6, we can see that RHC and SA converges at the optimal fitness after approximately 1000 function evaluations. At 1000 function evaluations, MIMIC still hovers around 70 fitness value. Due to the sampling rate, there is no data for GA at 1000 function evaluations, but it also hovers around 70 fitness value even after 8000 function evaluations.

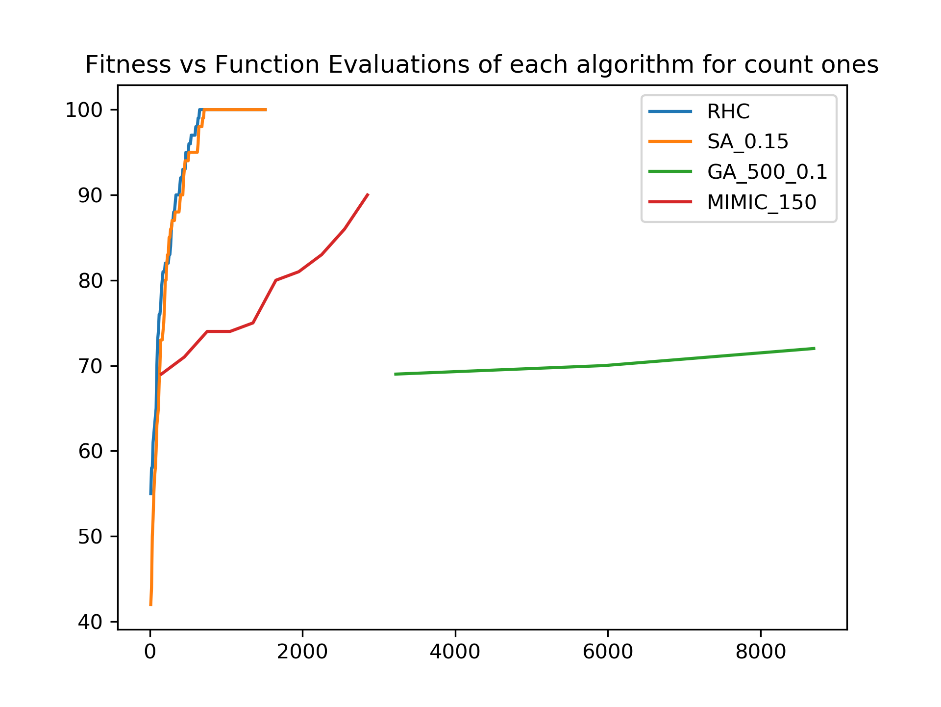


Figure 6: Fitness vs function evaluations for each algorithm on the Count Ones problem

Because of single global optimum and lack of local optima, we expect that any of the 4 algorithms would be able to find the global optimum. The weakness of RHC in possibly getting stuck in local optima is not relevant, since there are no local optima. In addition, the ability of SA to escape local optima in the early iterations becomes unnecessary.

The parameters of the input are also independent, there is no need for the inherent complexity of algorithms like GA and MIMIC that encode the assumption of some dependency between parameters. The computational effort in tracking multiple inputs, or generating probability distributions over regions of good inputs is unnecessary. For such a simple problem, a simple algorithm like RHC and SA work well to converge at the optimal solution in the least amount of time.

**Knapsack Problem**

*Description:*

The knapsack problem is interesting because its decision problem form is known to be NP-complete. There is no polynomial time solution for the knapsack problem, and any deterministic solution is usually considered to be intractable. Randomized optimization methods are frequently used as a heuristic. In the optimization form, we are given a set of items, each with some weight and some volume. We are also given a knapsack with some maximum volume, and we want to add items to our knapsack to maximize the total weight while not exceeding the maximum volume.

*Results:*

For this problem, the MIMIC algorithm produced a combination of items with the best fitness. We can see from figure 7 that the MIMIC algorithm using 1000 samples per iteration converged on a solution with approximately 18200 total weight, while the rest converged on a solution around 17000 weight.

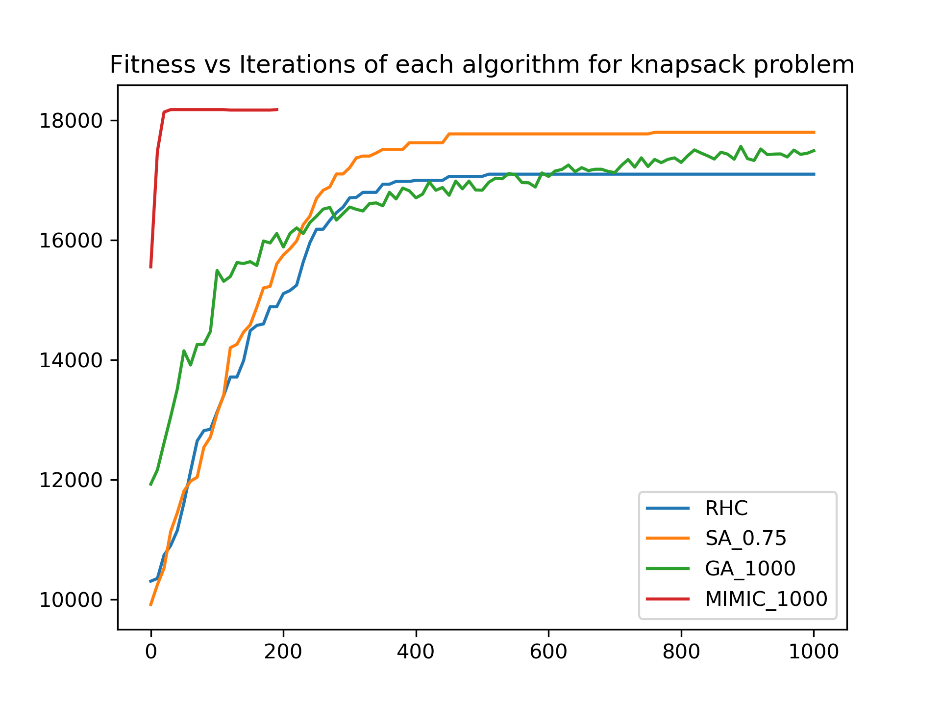


Figure 7: Fitness vs iterations for each algorithm for the knapsack problem

One possible reason for the performance of the MIMIC algorithm may be in the structure of the probability distribution that is generated at each iteration. A special case of a Bayesian network, a dependency tree, is used to represent the probability distribution. Each node of the tree represents an item selected to be in the knapsack, and a path from the root to leaf represents a combination of items. Selecting items one after another can be naturally represented by the dependency tree, and makes the estimation of regions with good solutions more accurate.

We can also expect that there would be many local optima in the fitness space (adding a large weight, large volume item would leave little remaining volume to add items). We can see that the SA algorithm performs better than the RHC algorithm, likely because it was able to escape a local optimum and converge into a better one.

GA performs reasonably well, falling between RHC and SA. A possible reason for not performing better than SA may be because GA tends to inherit the combinations of previous iterations. Instead of adding items when there is enough volume, it may be relying on crossover to fill up the volume.

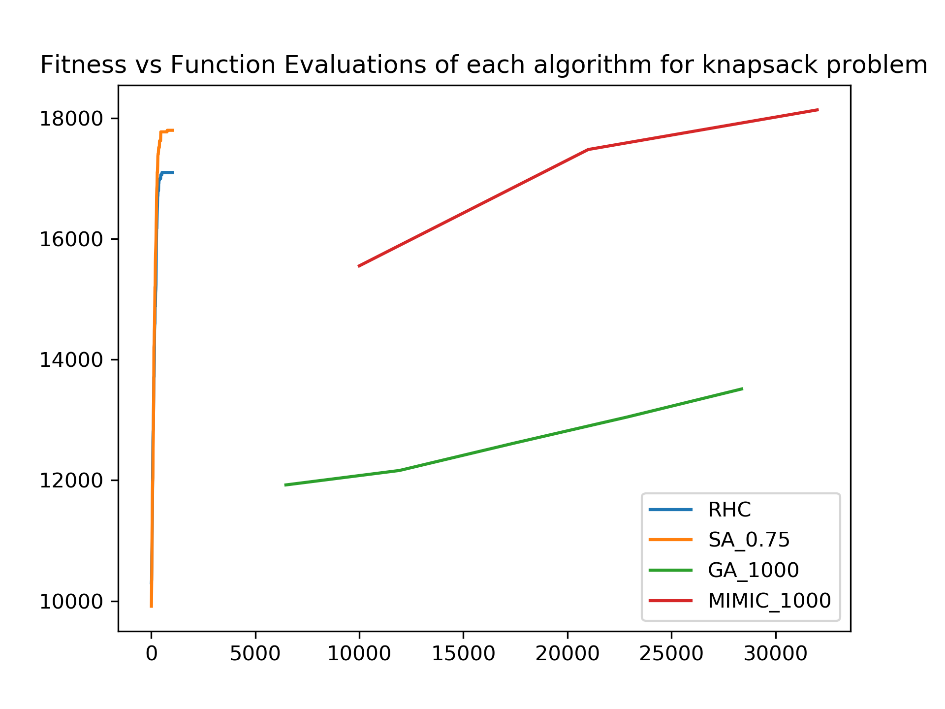


Figure 8:Fitness vs function evaluations for each algorithm for the knapsack problem

Despite the performance of MIMIC, it has the disadvantage of requiring a large number of function evaluations. In figure 8 we can see that RHC and SA converge after 1000 function evaluations, while GA and MIMIC require far more function evaluations. For MIMIC, the fitness seems to exceed the fitness of RHC and SA after 30,000 function evaluations. If the cost of doing a function evaluation is high, then it may still be more appropriate to use SA over MIMIC.

**Traveling Salesman Problem**

*Description:*

The traveling salesman problem (TSP) is also another interesting NP-complete problem. Given a graph with a set of vertices and non-negative weighted edges, we want to find the smallest weight cycle that touches every vertex. Like the knapsack problem, a deterministic solution is intractable, and randomized optimization is frequently used as a heuristic.

*Results:*

As seen in figure 9 the GA algorithm performed the best with this problem, converging at a path with 0.11 fitness (9 length). Once again SA performed better than RHC, with a fitness of 0.083 (12.1 length) compared to a fitness of 0.081 (12.3 length). The MIMIC algorithm performed the worst, with a fitness of 0.07 (15 length).

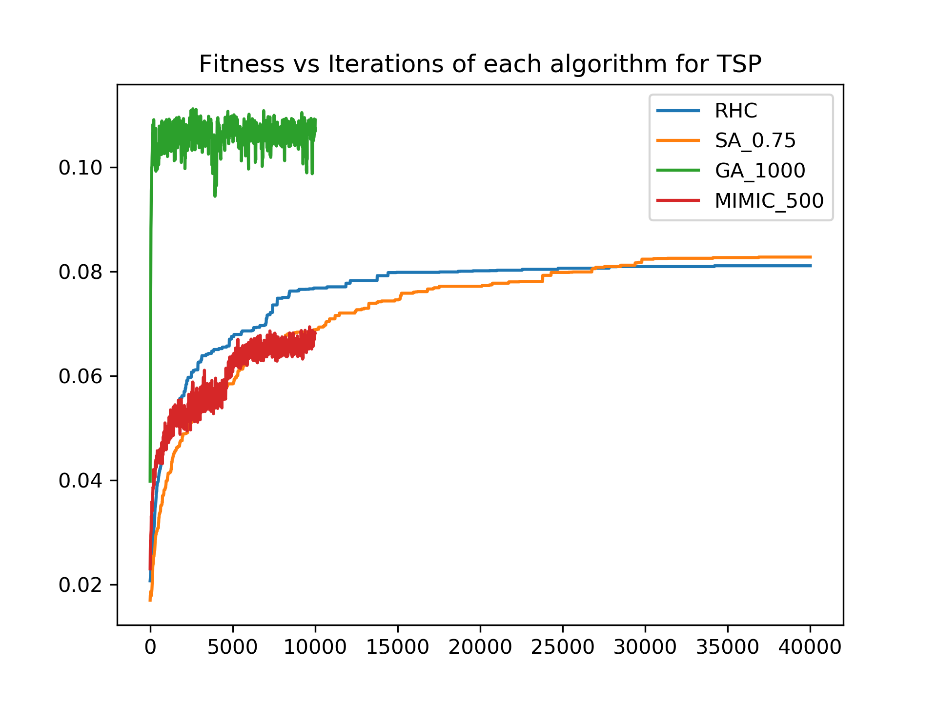


Figure 9:Fitness vs Iterations for each algorithm for the traveling salesman problem

One possible reason for the performance of the GA algorithm is that the population maintains blocks of parameters that perform well together from iteration to iteration. In terms of the TSP, these blocks of parameters correspond to a path through the graph. Once a good path is found, then that path is inherited through the iterations with few changes while it looks for a good partner to mate with.

The fitness space of the TSP appears to have many local optima and very narrow basins of attraction for good optima. We can get some intuition for this by assuming that we have a cycle through the graph. By doing a single swap of vertices, we can get drastically different path lengths. Only a very particular cycle gets a good length. This may be a reason why SA and RHC converges on similar fitness functions, and MIMIC does poorly. With so many local optima, SA and RHC converge quickly upon a suboptimal solution. With such a large input space and narrow basins of attraction, it is difficult for MIMIC to sample enough of the input space to get an idea of where the regions of good inputs are.

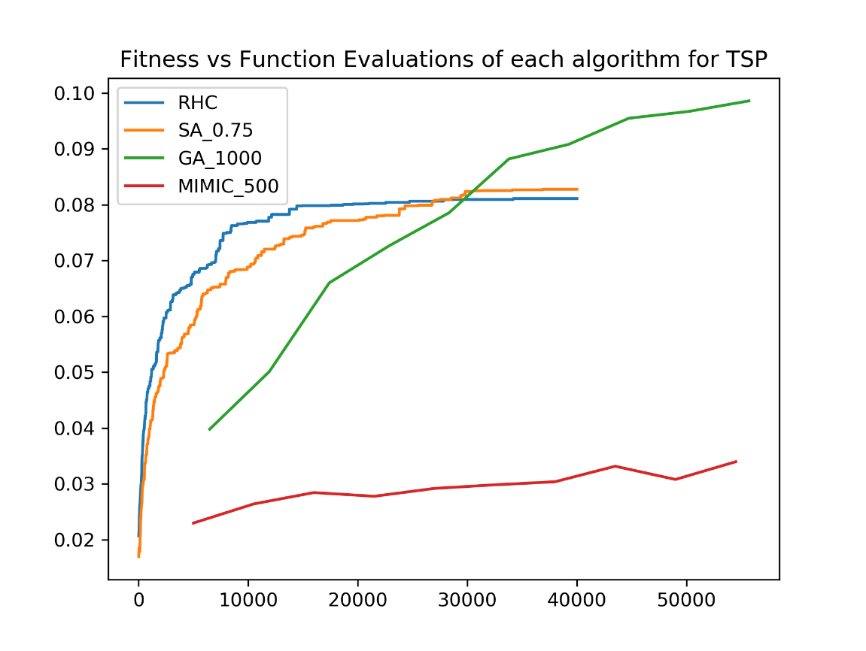
****

Figure 10: Fitness vs function evaluations for each algorithm for the traveling salesman problem

Unlike the past problems, the algorithm that gives the best fitness also does well in terms of number of fitness evaluations. At 30,000 function evaluations, RHC and SA converges, and GA exceeds the fitness values of all other algorithms (figure 10).

**Conclusion:**

The experiments above highlighted the strengths and weaknesses of the various randomized optimization methods. Training the neural network with RHC, SA, and GA made it clear that backpropagation through gradient descent is a better method, since it is able to get more information about the shape of the fitness space and use that to update the weights. Solving a simple problem of Count Ones demonstrated that RHC and SA outperforms complex methods like GA and MIMIC whenever there are no or few local optima. In the Knapsack problem, where the parameters are related to each other in a structure similar to the probability distribution created by MIMIC, MIMIC produces the best solution. However, MIMIC has the disadvantage of making many fitness calls while sampling from the distribution. GA worked far better than the other methods in the TSP, due to the benefit of keeping a good block of parameters unchanged through iterations. Depending on the structure of the problem, and the expected shape of the fitness space, different methods perform better than others.

**Sources:**

[1] Wine quality dataset: <https://archive.ics.uci.edu/ml/datasets/wine>

[2] ABAGAIL library: <https://github.com/pushkar/ABAGAIL>