Athreya Anand
CS 4641
6 November 2018

Randomized Optimization Deep Dive

Introduction

This analysis has been split accordingly into two parts. The first comprises of the implementation of three distinct random search algorithms within a neural network in order to find ideal weights. The second part consists of two optimization problems illustrating the efficiencies of the three aforementioned search algorithms. Overall, in this paper, I will be looking into the results of these two implementations and come to a conclusion that identifies which search algorithms are beneficial in select scenarios.

Dataset Summary

1. Nursery: This dataset comes from 1980's Slovenia where nursery schools received such an excessive amount of applications that most schools did not have the capacity to accept them all. Therefore, the government stepped in and started adding discrete quantitative values to applications in order to distinguish better applicants from worse ones. These numbers revolve around three primary classification categories: financial standing, family occupation, and overall family healthiness. As for the dataset itself, each of the three categories are composed of eight different subfeatures with three to five possible values to take on. The actual label can also take on five possible values ranging from "not recommended" to "special priority." The distribution consists of approximately 13,000 equally distributed instances.

   I personally decided to change my dataset from the ones used in the last analysis due to the fact that both of my previous datasets were focused around image recognition and did not have the greatest run time. In addition, I wanted to try a more straightforward classification problem versus complex sets with a plethora of attributes in order to see differences in performance and runtime. However, similar to my last choices, I wanted to find a set of data that was personally meaningful and had significant societal impact and believe I landed on a dataset that really lived up to those standards.

   In addition to using this dataset, I also did some manual manipulation to simplify the problem even further. Instead of assigning each input one of five labels, I decided to classify them as one of two labels. Since the upper two labels of the original five are called "priority" and "special priority" applications I decided to label every instance as either "no-priority" or "priority." I felt this would better clarify the differences in the algorithms due to the fact that the problem is now a binary classification.

Implementing Search Algorithms within a Neural Net

Library and Initial Prototyping

When experimenting and playing around with different libraries in order to incorporate the search algorithms within the neural network, I landed on ABAGAIL due to its ease of use and implementation. In order to use the different search algorithms I borrowed my neural network implementation from my supervised learning analysis and modified it in order to work with the new binary classification problem I created. Technology wise, I used the same sklearn library and the default genetic neural net classifier that came within it. However, after some initial fiddling, my immediate initial hypothesis was that the the go-to back propagation algorithm would be much better in both accuracy and runtime. This was simply because the backprop algorithm was stochastic gradient descent: a very optimized and highly efficient algorithm. In my mind, there was no way any of the to-be-implemented randomized search algorithms could surpass the two aforementioned qualities due to their lack of optimization.

Results

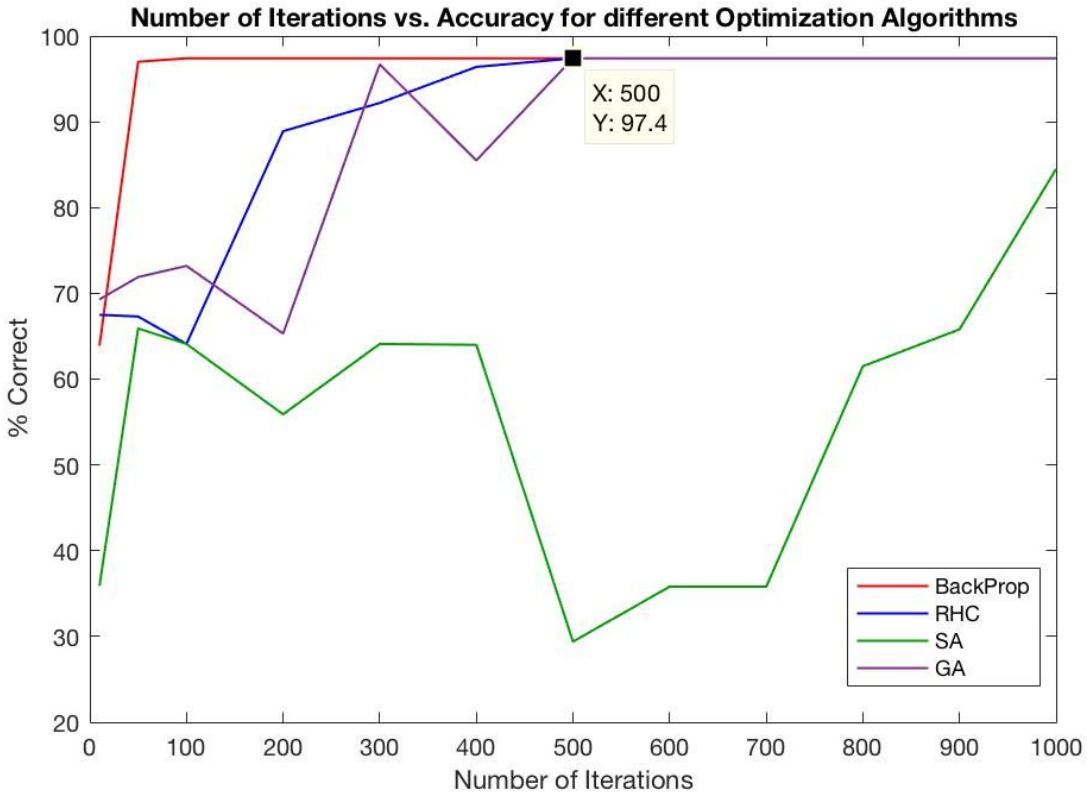

Figure 1: *iterations versus testing accuracy using implementation of search algorithms*

The consistency between the two graphs can be seen with the number of increasing iterations; all algorithms were tested with iterations from 10 to 1000 increasing around 100 each

time (10, 50, 100, 200, 300, etc.). After these tests, only three of the four search implementations were able to eventually reach an identical accuracy of our back propagation algorithm (97%). As expected, the stochastic gradient descent blew out all four algorithms by reaching the maximum accuracy the fastest at fifty iterations (*figure 1*). On the complete other hand, simulated annealing came in last with a max accuracy of eighty-five percent after all one thousand iterations. Let's take a look at each algorithm by itself:

Random Hill Climbing (RHC) took 500 iterations to reach its max accuracy. The random starts within RHC makes the algorithm the simplest out of the three analyzed but is still too slow when comparing it to the 50 iterations of our default back propagation algorithm. Looking at it deeper, the slower result could be due to several local minima the algorithm is getting stuck in within the problem; this would more than explain why it took 500 iterations instead of backprop's 50.

Simulated Annealing (SA) is similar to random hill climbing but has a unique temperature feature that allows more exploration; however, this feature results in SA being the poorest algorithm of the bunch. This is because the algorithm itself fails to reduce its temperature for the proper extremes and reveal more when trapped within the global minimum. Taking a deeper look, the lack of decisiveness could be due to the uniform nature of the binary dataset; since this distribution could cause a more flat graph with a plethora of shallow minima and optima, simulated annealing has a very high chance of being unable to find the true global optimum.

The Genetic Algorithm (GA) is similar to RHC in that they both reach the optimal 97 percent in around 500 iterations (*figure 1*); the accuracy graph of GA also looks very similar to that of RHC. This is due to the similarity in both algorithms since GA choses a random point and mutates a new set of solutions in order to pick the "fittest" and keeps producing and mutating off of the best of the population. Also again, the 500 iterations it took for GA to reach its ideal accuracy reiterates the abundance of local extremes within the dataset.

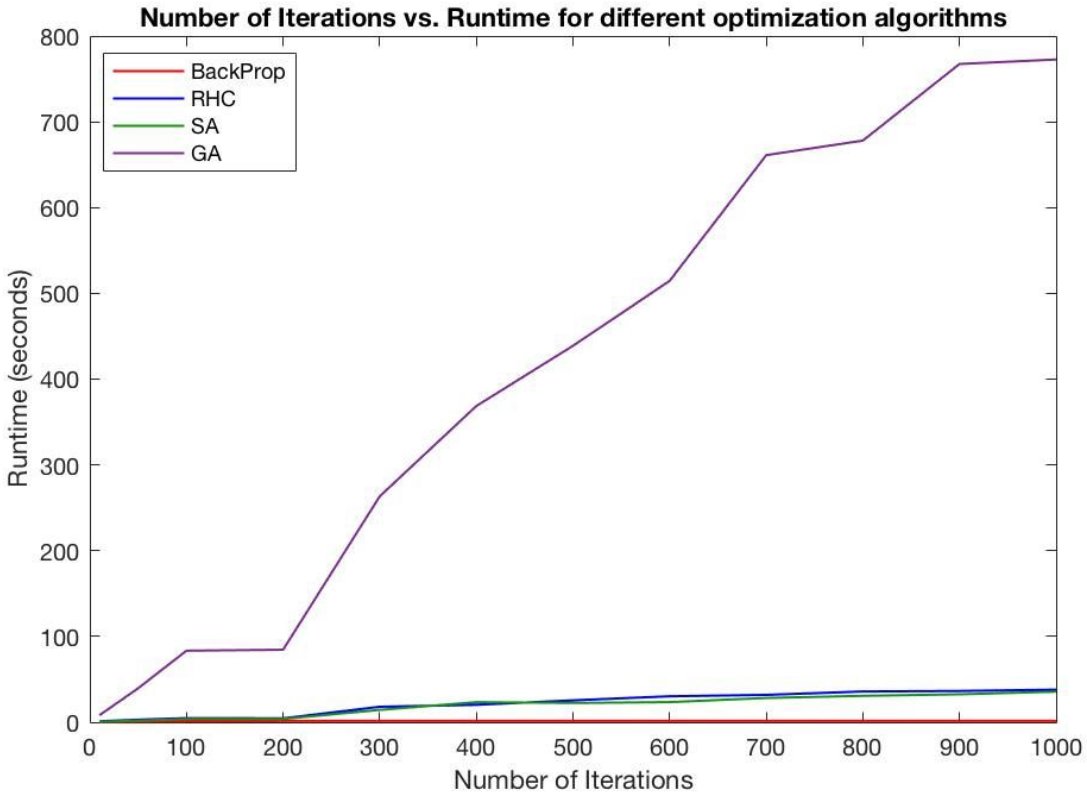

Figure 2: *iterations versus runtime using implementation of search algorithms*

Figure 2 reveals that the Genetic Algorithm is the king of runtime while our highly opti,ized backprop is the quickest. However, this huge discrepancy was to be expected. GA obviously takes the longest due to the nature of its process; by taking one single point and generating an entire population from it, the algorithm significantly increases its runtimes (more points = bigger runtime). The fact that this occurs on each iteration clearly explains why the algorithm can take a whopping 800 seconds on 1000 iterations. Backprop takes the shortest since it manages to immediately discover the perfect weight values causing the algorithm to return early immediately when the perfect classifier is produced.

Randomized Optimization Problems

Before I dive into the second part of the analysis, a brief introduction. Throughout the rest of the paper I take a look at specific optimization problems in which a certain algorithm out of the three (SA, GA, or RHC) performs better than its fellow brethren. However, when using ABAGAIL and going through the Udacity course, I found another plausible algorithm, did some research on it, and decided to add it into my experiments out of personal curiosity. Introducing MIMIC, a randomized optimization algorithm that is very similar to RHC but differs in that it has the ability to use previously visited data points and the results of such in order to better its focus. So, without further ado, let's take a look at the domain problems.

Traveling Salesman

The travelling salesman problem is a widely known problem and is actually the latest problem I used in a coding interview using dynamic programming. The problem itself focuses on a graph and tries to find the lowest costing path that touches/visits each of the nodes once and returns to the starting point. For this, I tested the domain with each of our now four algorithms and analyzed the results to find the best one. In order to see the efficiency, I decided to graph the inverse of the total travelling distance instead of minimizing distance; i took this approach so I could use maxima since minima are harder to work with for me personally and for the sake of this homework. All algorithms were tested on varying sized graphs ranging from five to a hundred nodes. The program would generate a graph with randomized node positions and weights between aforementioned nodes in which all four algorithms were tested on.

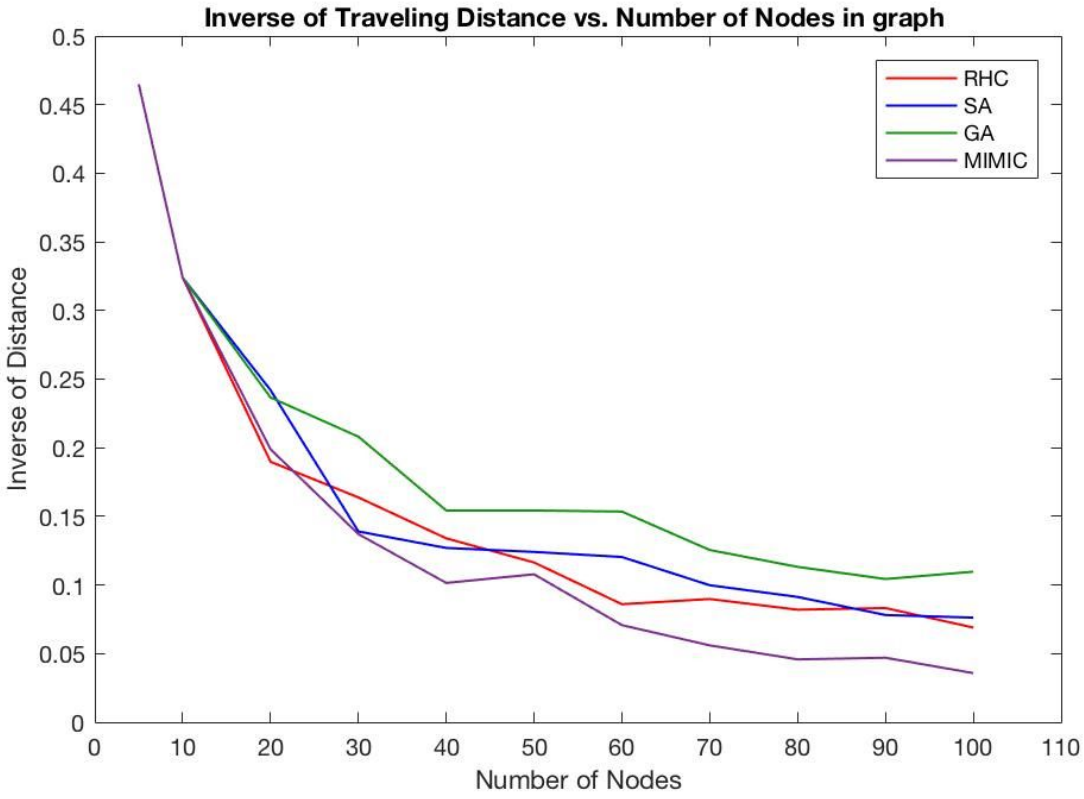

Figure 3: *inverse travelling distance versus nodes in the problem domain using search algorithms*

Figure 3 reveals the performance of each of our four optimization algorithms on a randomly generated "map" with N nodes. Immediately, the first thing that catched my eye was the fact that inverse distance decreases as the number of nodes increases. This is great since we expected for the distances to increase as the number of nodes increased. We can also see that from five to ten nodes, all four algorithms perform the same suggesting that they all find the global maxima successfully. However, after that, Genetic Algorithm takes the crown when

compared to the others. In addition to this, we know that the results of the experiment are not random since all four algorithms follow a pattern in reducing their performance as size increases. Not only this, but all four algorithms are able to keep stable positions relative to one another. The Genetic Algorithm does better due to the fact that it generates "populations" that are the most fit; in such a large solution space, this gives it the ability to not drift away from the global maxima by much. On the other hand MIMIC performs the worst since it tries to span the entire solution space resulting in the significant decrease of the probability of finding the correct path. This could be counteracted by possibly increasing MIMIC's sample size; however, the runtime was already significantly worse compared to the other three and this would not help that problem whatsoever.

Flip Flop

The flip flop evaluation function is a basic solution that discovers how ever many flips occur within a string composed of binary values. As a simple rule of thumb, the first value is automatically a flip since it changes from "nothing" and every subsequent bit change adds to the total. As an example. "0011" would return two since the first and third indices count as a flip. If we were to expand on that, "1010" would give us a value of four since a flip occurs at every index. The minimum value the test can result is one since the first index always counts; the largest possible value would be equal to the length of the string if each character was the opposite of the prior.

Each of the algorithms weill start with an identical randomized binary string and iterate through in order to convert it into an alternating bit string (largest score). Immediately, we can hypothesize that a plethora of local maxima/minima are evident since the neighboring bits of whatever state the algorithm is in could only either be identical or a different by a value of one.

After understanding the problem, I began testing the four different algorithms and tested their performance on varying lengths of bit strings; the testing started from strings comprised of fifty bits and went all the way up to five hundred in increments of fifty (50, 100, 150, … 500).

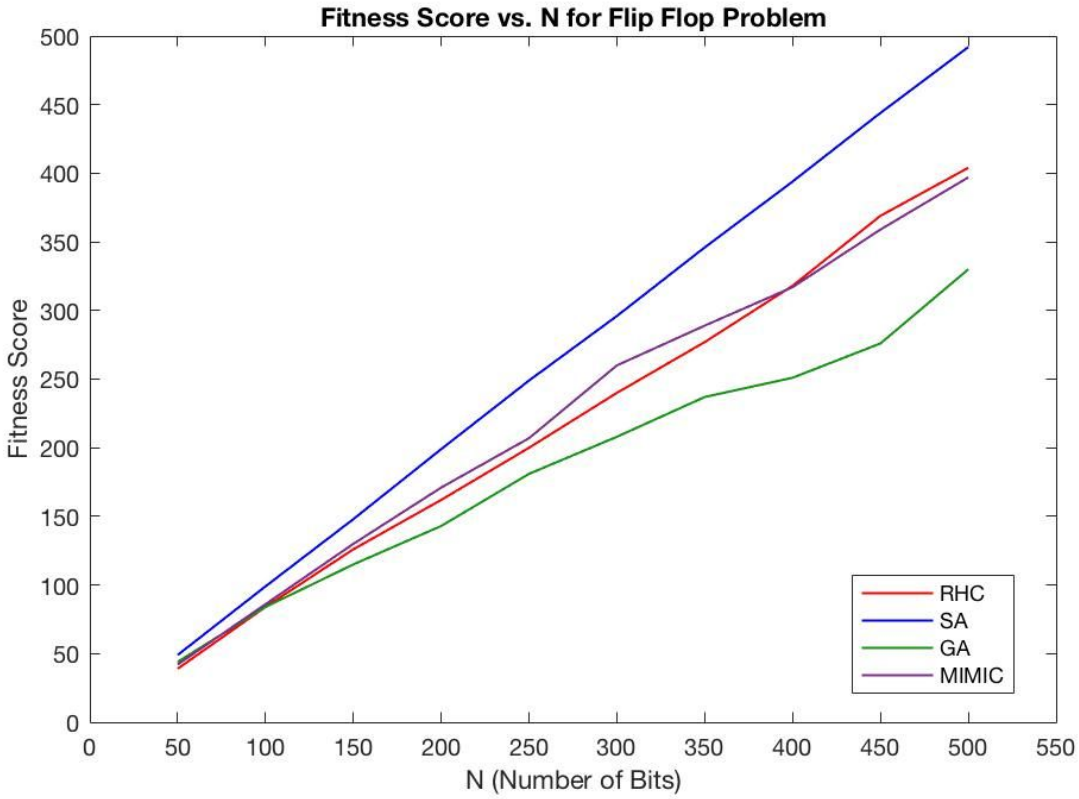

Figure 4: *number of bits versus fitness core in the problem domain using search algorithms*

Figure 4 reveals the performance of each of the four algorithms we tested in order to find the optimal algorithm for our flip flop problem. It instantly becomes clear that Simulated Annealing (SA) did the best job out of the bunch and is able to be on top of all algorithms for every bit count (N). But why? This is simply explained by my prior hypothesis; the fact that there are a plethora of local maxima and minima, it becomes easy for other algorithms to get stuck within these troughs. Simulated Annealing's initial high temperature allows the maximum exploration before narrowing down to a single maxima. This starting high exploration rate allows SA to find the correct spot the fastest.

On the complete other hand, we can see that the Genetic Algorithm performs the worst; this behavior occurs since the algorithm is generating new populations in an environment where local extrema are very close together. The close proximity of all these points merely makes the entire newly generated population random noise.

Random Hill Climbing and MIMIC are always somewhat equal in their performance. This illustrates that the ability to use previously visited data points and the results of such don't really prove too beneficial in the problem domain. However, we can conclude, since RHC is the exploration element of the Simulated Annealing algorithm, that the temperature and random exploration of simulated annealing is what increases its efficiency in the domain. This made me wonder what would occur with varying cooling rates for the Simulated Annealing algorithm; so let me reveal the findings.

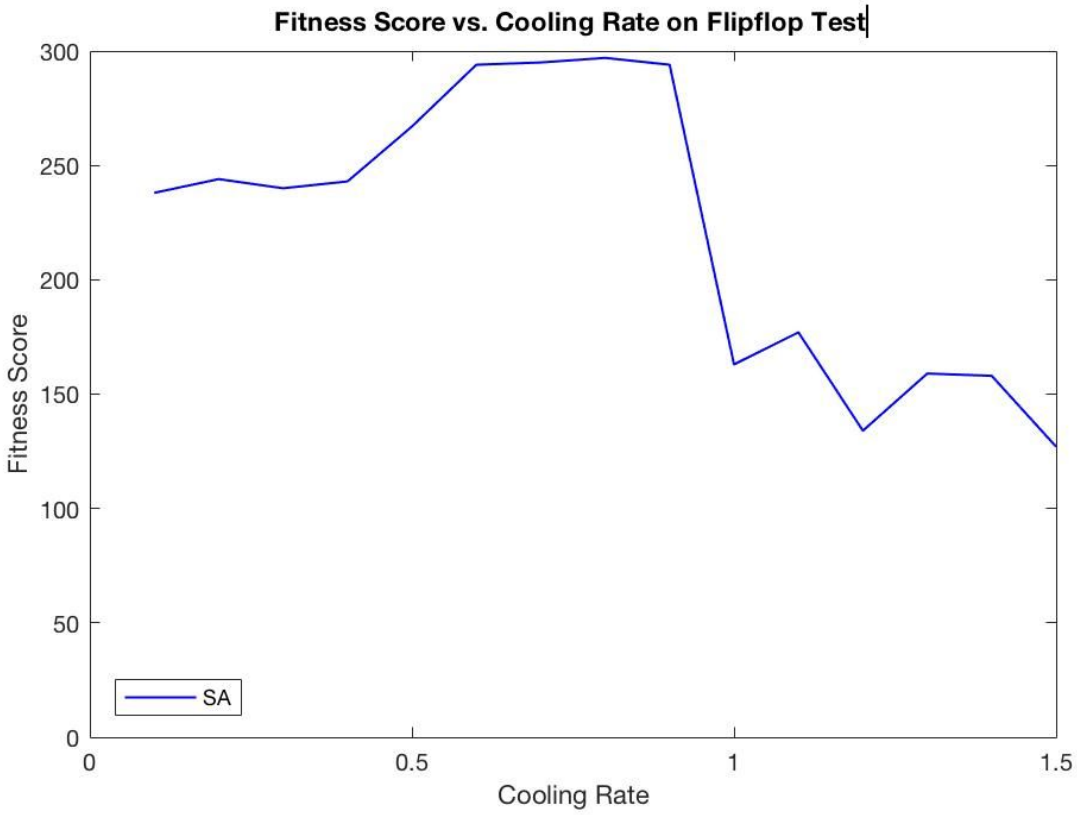

Figure 5: *performance of simulated annealing with varying cooling rates*

As we know, the cooling rate is what determines the amount of exploration simulated annealing does; the larger the rate, the less the algorithm explores the problem space. FIgure 5 reveals that the cooling rate is inefficient both when too low and too high. Making the cooling rate too high does not allow for enough exploration in order to discover the ideal solution; on the other hand, a smaller rate and increased exploration causes less exploitation and is not ideal to find the perfect solution. Like goldilocks, we see that simulated annealing prefers a cooling rate right in the middle of two extremes; any cooling rate between 0.5 and 1 allow for the algorithm to perform as efficiently as possible.

Overall, that concludes my deepdive into the two domains of the paper: neural net and search problem domain implementations. From the first part, we can conclude that random hill climbing, genetic algorithms and of course our optimized backprop algorithms result in the highest accuracy when implemented into our neural network. Unfortunately, simulated annealing's temperature and randomized exploration causes its demise and makes it the least efficient algorithm to implement into a neural network. In addition, we found that the default stochastic gradient descent backpropagation implementation is the best bet due to its simplicity and highly optimized nature; these features allowed the algorithm to have both the best accuracy and runtime. The second part of the paper revealed that Genetic Algorithms perform the best in

problems with a large solution space where there are well defined local and global maxima. It also revealed that Simulated Annealing is beneficial in problem domains in which several local maxima are close in proximity and exploration is required to find the most optimal solution. Not only that, but we also saw that the ideal cool rate is between 0.5 and 1 where anything greater or less takes away from the efficiency of the SA algorithm. In all we see that the amount of maxima and minima as well as the overall size of the solution space determines what search algorithm to use in a certain situation.