3-CNF SATISFIABILITY USING THE GENETIC ALGORITHM

By Praveen Sridhar - 2018A7PS0166G

In partial fulfillment of the requirements for the course

ARTIFICIAL INTELLIGENCE - CS F407

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCES, PILANI, K.K. BIRLA GOA CAMPUS

Introduction

The Genetic Algorithm is a powerful optimization technique for solving a discrete combinatorial search problem. This is particularly powerful for problems with a very large search space and problems which are not solvable in polynomial time (NP). The 3CNF problem fits the above conditions perfectly and would be the ideal problem that can be solved by the genetic algorithm. By bringing in an element of randomness and allowing the algorithm to make decisions which may not be the immediate best choice, we alleviate the local minima problem greatly. In this way, the Genetic Algorithm outperforms its traditional greedy counterparts which are almost guaranteed to get stuck in local optima and runs much faster than classic brute force.

The python file attached contains all the improvements that were attempted but weren't actually included because they did not perform as well as expected. This includes culling, elitism and the softmax activation function to model the weights (probabilities).

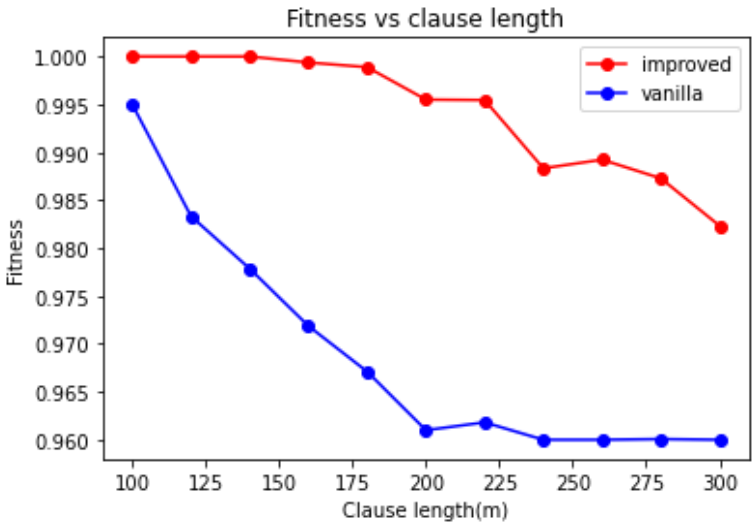
The functions called contain a set of default parameters disabling the above operations and keeping only the relevant ones.

The following variations were tried:

- The Vanilla Genetic Algorithm
- Genetic Algorithm with an improved combination function (best)
- Genetic Algorithm with a modified mutation function
- Genetic Algorithm with elitism
- Genetic Algorithm with culling
- Genetic Algorithm with softmax activation
- Genetic Algorithm with varied population size
- Genetic Algorithm with multiple combinations of the above possible hyperparameters

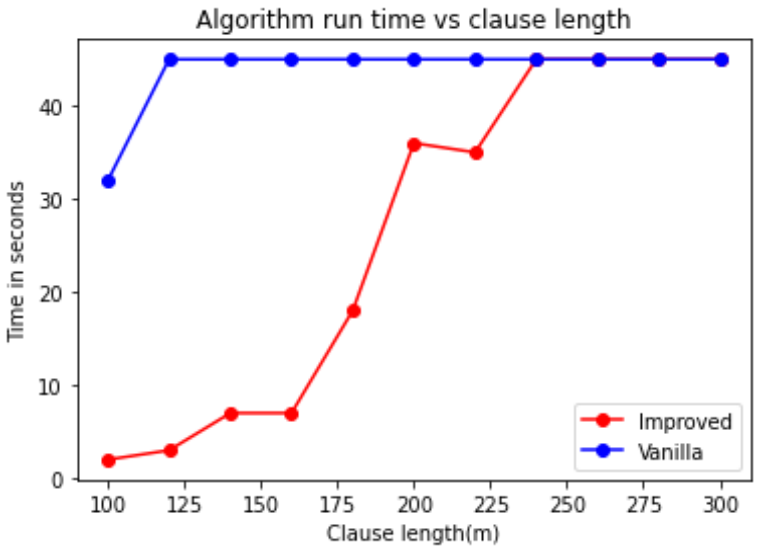
The fitness function is the percentage of clauses satisfied in the given sentence. The entire code is written following principles of Object Oriented Programming.

Question 1:



The graph plotted above represents the average performance of the genetic algorithm for 10 randomly generated sentences for a given sentence length. The modified genetic algorithm incorporates a custom reproduce() function where each combination of parents generates 2 children. The child with better fitness is returned. Additionally, there have been optimizations made in the state representation as well.
The decreasing trend of the fitness value and the increasing time taken (shown below) to find a solution shows that the sentences become increasingly harder to satisfy. Additionally, some sentences may be completely unsatisfiable at large sentence lengths as well.

Question 2:



The rapid increase in average time taken to find a satisfying assignment for the sentence with an increase in sentence length once again reiterates the point that the sentences become harder and harder to satisfy.

Question 3:

There are multiple areas which could be improved upon in the original genetic algorithm. The reproduction function, the mutation function, etc. We can achieve this through operations such as elitism, culling, etc and progressively generate a better population.

A lot of these above methods were tried and tested in conjunction with the others which led to a vast combination of hyperparameters tried out. We will start out with the hyperparameters which offer the best improvements in performance and work our way down to the ones which didn't help (and even hindered) the performance.

Approaches that worked the best:

Reproduce function:

This was the single greatest point of improvement over the vanilla algorithm. Traditionally, we select 2 parents based on a probability array which is proportional to their fitness value, select a random index, pick the left half of the state upto the index from parent1 and pick the right half of the index upto the end from parent 2 and concatenate them.

Here, do the above, but we generate a second child, this time choosing the left half from parent 2 and the right half from parent 1. We pick the child with the higher fitness value and return that. This simple optimization had the algorithm converging in a second while the vanilla didn't converge even after 45 seconds

Other optimizations such as including more parents only slowed down the algo further. Generating any further children per combination of parents didn't help either and in fact hindered performance. The added runtime associated with generating more children did not outweigh the reduced number of generations (which was a natural consequence of the increased time per generation).

The State Object:

The state class was designed keeping in mind the principles of object oriented programming. The idea here was to keep the fitness value tightly coupled with the actual assignment instead of having them in separate lists. This would especially be useful when performing operations such as culling or elitism and we want to sort based on a key value like fitness.

The state class consists of a single constructor, a fitness function method and 2 class level variables - one for the actual assignment and one for the fitness function. The assignment can be a numpy array or a list and must be passed as a parameter to the constructor of the state object. The CNF should also be passed to the constructor in order to calculate fitness.

At the time of object instantiation, the fitness value is called and stored in the class level variable. As a result, the algorithm has to never recompute the fitness function for any state. This gave a significant increase in the speed over recalculating the fitness function over and over. It also made performing operations like sorting much easier for culling and elitism. (for operations like culling, we don't need to perform costly removal operations or create a new list of only the best individuals. We can simply set the fitness value in the state to 0 and the state will never be selected again).

The numpy array State Representation:

Initially the state assignment was represented using lists and a lot of the operations were performed using list comprehension and for loops. Certain operations can be made dramatically faster by the use of numpy arrays because they are written in C++ and they have internal memory optimizations.

After extensively testing all the methods of numpy, it was found that operations like addition of 2 numpy arrays, concatenation, sum and mean were much much faster with numpy arrays - upto a 100 times faster. However the append operation on numpy arrays was really slow, about 20 times slower than regular lists. Therefore, the population was left as a list because of the large number of append operations. But the internal representation of the assignment in the state object was modified to be a numpy array.

While this offered some speed up, however it was not quite as pronounced as the previous 2. This gave anywhere between 100-150 more generations per run with a population size of 20. Early stopping wasn't implemented because the algorithm would often find a better solution after many generations of stagnation and this number was highly variable. This was the case with stopping based on time as well. Given that early stopping wasn't implemented, these extra 150 or so generations were found to make a difference on some runs and formulae.

Approaches that did not work well:

Culling:

Culling is a way to eliminate the weakest individuals (small fitness value) in the population. 2 implementations were tested but neither of them yielded good results.

- Threshold based approach: Here we set a threshold value for fitness or the probability beneath which the individual will not be selected. This threshold is identified by simply running the algorithm and finding the approximate range within which the probabilities of the individuals lie. For this algorithm, the culling was performed based on probability because as we vary the number of clauses, the expected fitness range would change quite a lot and this can result in too much culling and the population dying out. However, the probabilities will lie in approximately the same range due to the population size remaining constant and the closely clustered fitness values of the individuals themselves in the population. This approach offered no improvement in performance and performed almost exactly the same as the improved algorithm mentioned above.

- Surplus population approach: Here we generate a larger population of individuals than required and sort them based on their fitness. We then select the top individuals who fit our needs. This approach didn't improve performance either

Softmax activation:

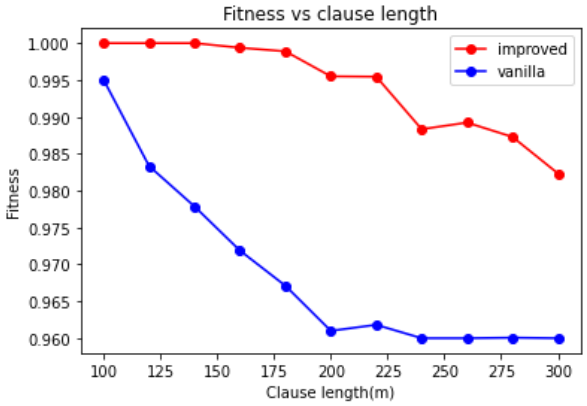
One way to make sure that better individuals are selected even more often is to use a softmax activation function on the fitness to model the probabilities. This is commonly used in neural networks to predict this output.

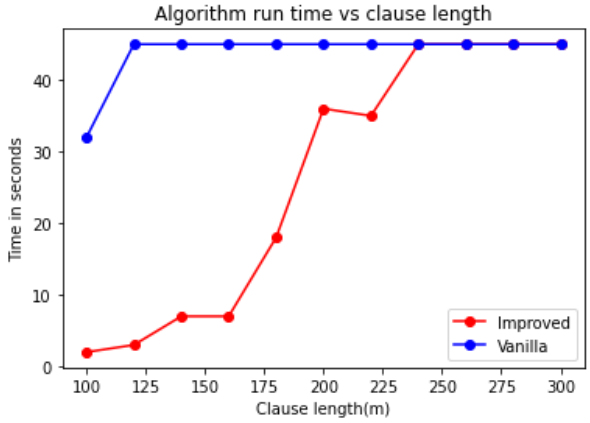
$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

The advantage of using this function is that it increases the weight given to the most fit individual as opposed to simple normalization by dividing by the sum. This approach didn't change much either.

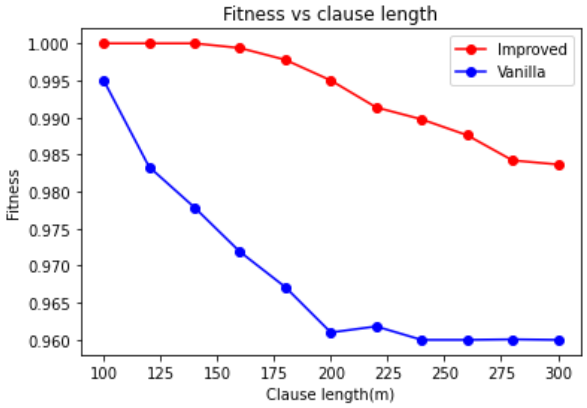
Modifying population size:

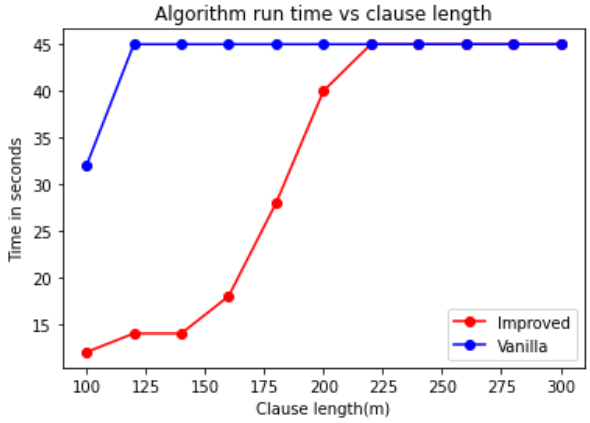
Population size 10:





Population size 100:





Modifying the population size had no effect on the actual performance for the range for formulae tested. However, it was found that smaller populations found the same solution a lot faster. Using a small population size makes the algorithm a little unstable as compared to having a larger population but given that at every step we store the best individual and that we have a larger number of generations, using a smaller population size was the way to go.

Approaches that degraded performance:

Elitism:

Elitism is a way by which the more fit individuals are retained in a population. This was again implemented in 2 ways.

- Retain best individuals: This involved concatenating the new population with the old population, sorting it and selecting the required number of individuals. This could potentially have the effect of retaining the entire parent population and none of the children if all the children were worse off than the parents.
- Retain a fixed number of best parents: This involved selecting the top best k parents from the population. For this we first sort the parent population and the child population separately. We pick the top k individuals from the parents and the remaining individuals from the child population. This would have a similar effect except that certain suboptimal individuals may enter the population for the next generation also

Both of the above approaches degraded performance pretty harshly with the algorithm not converging quite often even for a smaller number of clauses per sentence. This might be due to the fact that this step involves a sorting operation which can be costly. As a result, it is possible that the lesser number of generations hits the algorithm. Additionally, if we repeatedly retain the same individuals, the degree of homogenisation may become too high for the algorithm to explore the rest of the search space.

Mutation function:

The modified mutation function simply picked an index at random and flipped the bit at that index with a probability. This was done in order to help keep the population stable and prevent it from jumping around too much. However, this also degraded performance as compared to the original version where every bit would be flipped with a small probability. The explorative aspect of algorithm was compromised severely in favor of stability

Question 4:

From the above graphs, it can be seen that with increasing sentence length, the algorithm finds it increasingly difficult to find a good solution. This is because the landscape of the fitness function is expected to become complicated for a larger number of clauses and we are more likely to find local optima in our search space. It appeared to be the case that overall, the longer the algorithm ran and the more individuals it saw (either by means of a larger population size or a larger number of generations), the better the results were. This seemed to be a very large determinant of the performance of the algorithm itself.

Question 5:

The graphs tell us that the difficulty of finding a satisfying assignment for a 3CNF formula increases a lot with an increase in the number of clauses m for a fixed number of symbols n. The larger the value of the number of clauses m, the more likely the algorithm is getting stuck in local optima. It is quite likely that the sentence itself isn't satisfiable for a larger value of m.

Increasing the value of n exponentially increases the cardinality of the search space and is quite likely to result in the algorithm taking a much longer time to find a solution if it exists.