Artficial Intelligence (CS F407)

Programming Assignment 1

Report

Name: Mehul Gupta

Roll No: 2019A7PS0033G

Introduction

The task demonstrated in this report involved building a Genetic Algorithm to solve the NP-hard problem of finding an assignment of logical values for variables that satisfies a 3-CNF statement.

Algorithm

The pseudo-code for the improved-Genetic Algorithm implemented for this assignment is as follows:

```
function Genetic-Algorithm( cnf_statement) returns an individual
    initialize population
    repeat
        weights ← Weighted-by( population, fitness)
        population2 ← population
        for i=1 to Size(population) do
            parent1, parent2 ← Weighted-Random-Choice(population, weight)
            child ← Reproduce(parent1, parent2)
            if  (small random probability) then child ← Mutate(child)
            add child to population2

        population = Tournament-Selection( population2, Size(population), epoch)
    until ( fitness of individual is 100 OR best fitness has stagnated for many epochs OR time
        limit has been crossed)
    return the best individual in population according to fitness
```

As can be seen from the algorithm, it is similar to a naive-Genetic Algorithm as mentioned in the Artificial Intelligence, Russel and Norvig. Each run of the main loop in the algorithm involves taking the population, adding new children to the population and selecting new population based on a Tournament Selection (3) with respect to the fitness of individuals. Unlike the naive algorithm, in this even the old generation is considered in the new population (3) that participates in the tournament selection which allows fitter old generations to leak into the next generation.

The main differences in this new improved algorithm are encapsulated in the various algorithms that are used in the main algorithm mentioned above. They are as follows

```
function Reproduce(parent1, parent2) returns an individual
    n ← Length( parent1)
    c ← random number form 1 to n
    return  Append( Substring( parent1,1,c) , Substring(parent2, c+1, n) )

function Mutate(child) returns mutated_child
    for i=1 to length(child) do
        if (small random probability) flip Truth Value of variable \i in child
    return child
```

This Mutate algorithm allows mutation in more than one position of the child (3). The genetic algorithm implemented above has been selected to be highly selective so that it can converge on a good fitness individual as fast as possible so as to allow it to perform efficiently under time constraints. This often results in the model getting stuck in a sub-optimal optima in the state-space depending on the initial conditions (3). To fix this, mutation has been modified to allow mutation in each location (3) to allow a higher degree of exploration as compared to sticking to single mutation per individual (3). This was observed to give better results.

```
Function Tournament-Selection( population, pop_size, epoch)   returns population2
    population2 = []
    parameter alpha, beta                    where  (alpha < 1)
    sort population in decreasing order wrt to fitness scores
    for k=1 to pop_size do
        p ← alpha ^ ( epoch/ beta)
```

```
        if (random probability < alpha)
                population2.add( random individual from population)
        else
                population2.add ( population[i] )
    return population2
```

Tournament Selection implements a Fittest man survives policy where the most fit individuals are selected. It also takes random individuals by random probability which is kept high initially but exponentially decays with epoch run (3). Using a constant probability for this resulted in high variability in fitness even in the later stages and did not yield optimal conditions (3) so the probability was made to decrease exponentially. This allows more diverse populations in the initial stages of the algorithm and slowly makes it highly selective as it approaches the end of it's run.

Experiment

Experiments to gauge the performance of the new Genetic Algorithm was designed as follows:

3-CNF statements were randomly generated for a fixed number of 50 literals with number of clauses ranging from 100 to 300 with step sizes of 20. For each case, 10 runs were made over different CNF statements to estimate the average Fitness achieved by the model and the average time taken to run. Fitness was measured as the percentage of clauses satisfied in the CNF statement.

The terminal conditions used for stopping an individual run was either if the model achieved perfect satisfiability, stagnated at the same fitness for longer than 3000 epochs or ran for more than 45 seconds. Each population size is restricted to 10 per generation.

The results for the algorithm displayed below involved a lot of parameter tuning before arriving on the optimal parameters. The approach used from designing the algorithm to fine-tuning the parameters has been as follows, aggressive mutations with higher probabilities and random explorations in the initial stages paired with a highly selective tournament stage to explore diverse states and then reduce variations in the states to converge to an optimum as quickly as possible to meet the time constraint (3).

The results for the achieved are demonstrated by the graphs below.

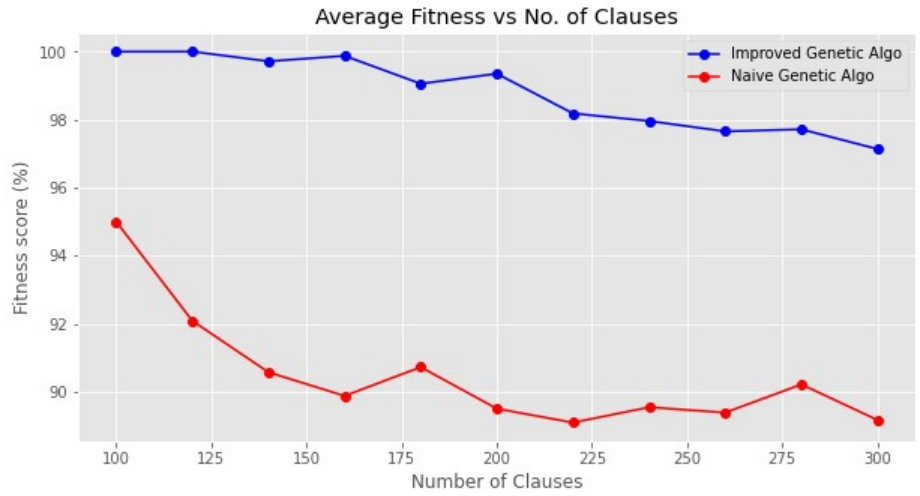

Fig: Average Fitness achieved vs Number of Clauses in 3-CNF (1)

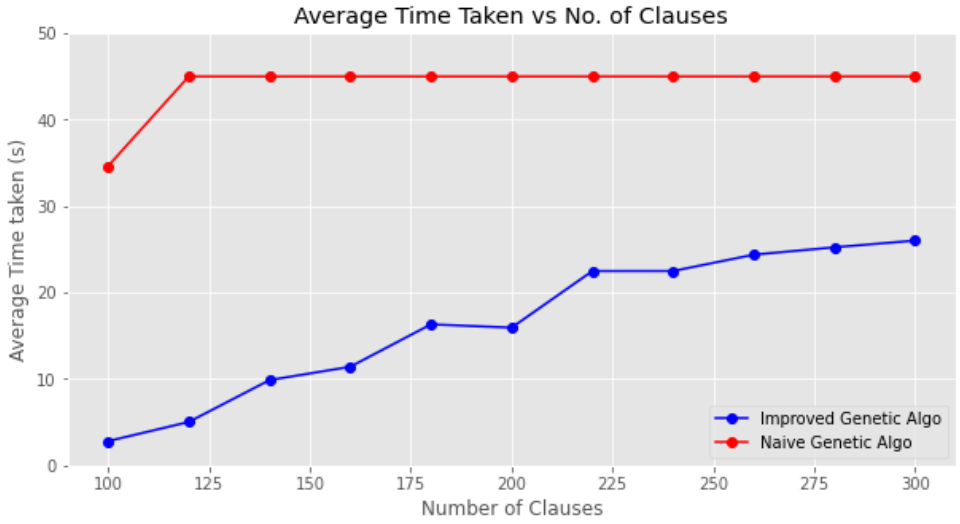

Fig: Average Time taken for the Genetic Algorithm vs number of clauses in 3-CNF (2)

As can be seen, the improved algorithm manages to converge extremely quickly, with cnf statements with 100 clauses taking merely 2.5 seconds and even the worst case scenario of 300 clauses capped under 30 seconds, compared to the naive-algorithm, which failed to even converge to a value beyond 120 clause CNF-statements, and managing accuracies over 97%, way over the best fitness achieved by the Naive genetic Algorithm.

Observations

From the graphs above, its easy to identify certain general trends:

1) Average fitness goes down as the number of clauses increase

2) Time taken for the Genetic Algorithm to converge increases with number of clauses.

It is easy to see that the overall difficulty of solving the 3-CNF problem increases as the number of clauses increases. This can be expected, as keeping the number of literals constant, as the number of clauses increases, the number of valid assignments for the CNF are extremely likely to go down. This will make even converging to a single solution extremely hard as assignments satisfying one clauses is increasingly likely to not satisfy another clause. Also considering that many functions have O(N) time complexity where N is the number of clauses, the time taken to perform a single run goes up significantly (4, 5)