ARTIFICIAL INTELLIGENCE - CS F407

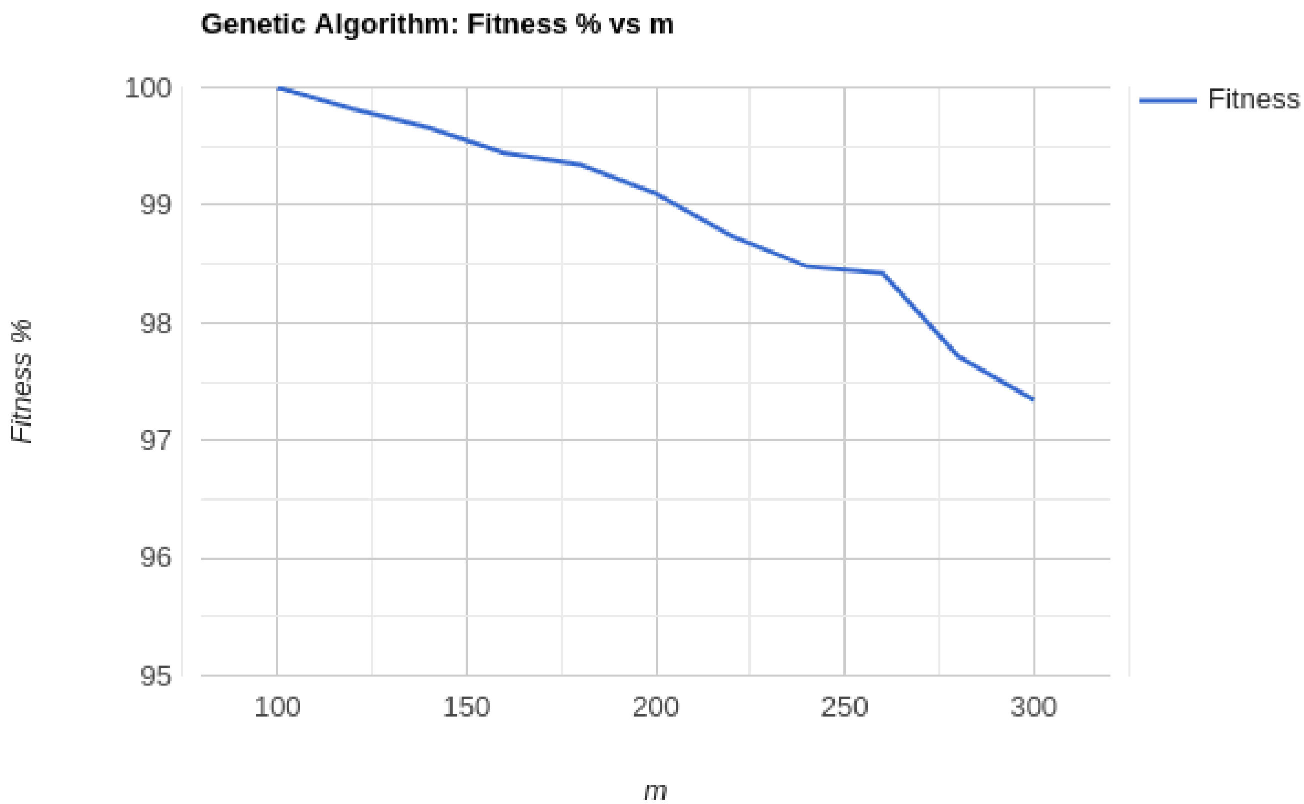
GENETIC ALGORITHM FOR 3-SAT Problem

A submission for **Assignment 1** made in partial fulfilment of course Artificial Intelligence - CS F407 by

KUSHAL JOSEPH VALLAMKATT

2019A7PS0135G

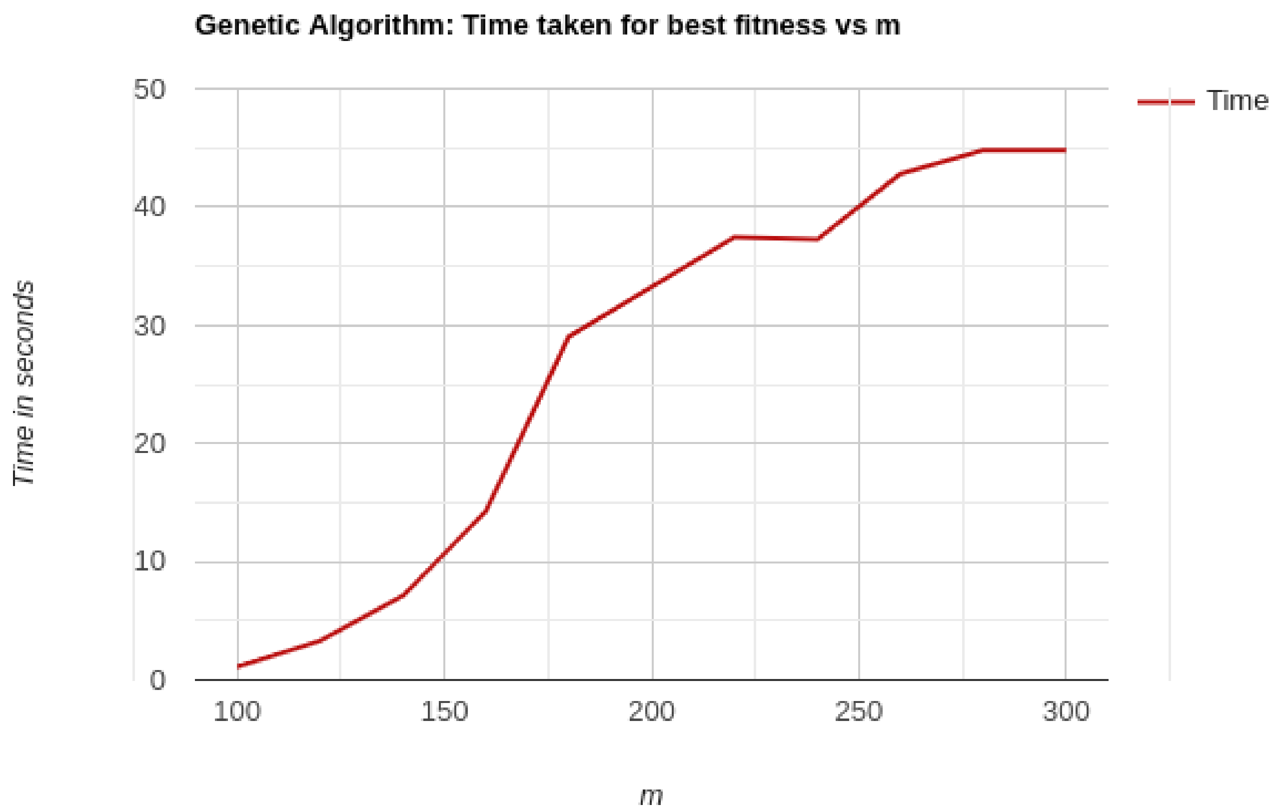
Q1: Graph of Fitness% value vs m, the number of clauses:



The fitness value is almost 100% for lower values of m, and it decreases to about 97.4% on average for m = 300

m	100	120	140	160	180	200	220	240	260	280	300
F %	100.0	99.81	99.66	99.44	99.24	99.10	98.74	98.48	98.42	97.71	97.34

Q2: Graph of Time taken for best Fitness% value vs m, the number of clauses:



The average time is greatly increased if GA gets stuck at a certain local optima. In this case, the average time is increased until the program is terminated or 45 seconds is completed.

m	100	120	140	160	180	200	220	240	260	280	300
Time	1.132	3.30	7.121	14.23	29.05	37.25	37.45	42.82	44.83	44.82	44.82

Note: I had auto-terminated the program completely at t = 44.8

Q3 How did I improve the GA Algorithm, and some failed approaches:

General Method, and population info:

The **members of the population** were taken to be Bit-Strings of length 50, since n remains constant = 50 throughout the experiment. If the Bit-String at index i == 0, it means that Variable #(i + 1) is set to negative (negation), and vice versa (positive) for value at index i == 1.

The **fitness function** used is direct, and measures the number of Clauses satisfied, given an assignment of variables (a bit-string). The population size is kept constant = 20, for all values of m, as this was found to be most efficient.

With the version of GA described in the Textbook, I was getting a moderate final fitness function value: I Was able to satisfy CNF, 100 clause-sentences, however, the value wasn't hitting even close to 98-99% for 100-120 clauses, and the time taken was in general, slow.

I made the following changes which helped improve GA Algorithm:

1. **Selecting best 2 from the population at every iteration to form the next generation**

   This method boosted up the running time of the algorithm by a great margin. Instead of assigning weights to each member on the basis of their fitness value, and hence choosing a member based on their weight (since the probability of a member with greater fitness to be picked is more), I picked the best 2 from the population to reproduce.

This process first involved ordering the population on the basis of fitness function values, then, taking only a certain fraction (elitism rate) of population onto the next generation. From this new generation, best 2 are picked and their offspring are added to the current generation to fill up for those members “killed” during transfer

2. **2-point crossover:**

   Instead of the general 1-point crossover, I introduced a better, 2-point crossover, for “reproducing” 2 members of the population. Here, we randomly select 2 points x, y (x and y represent the gene number, or the “variable” number in the chromosome) (x < y) and exchange the genetic material of these 2 parents in between x and y Only. The 2 parents would produce 2 children, 1 of which was similar to Parent1 in regions not belonging to (x, y) and the other was similar to Parent2 in regions not belonging to (x, y)

3. **Extensive Multi-bit mutation**:

   Upon experimentation, I found that extensive mutation helps to improve diversity of the population, so that if we proceed along a path that “looks correct”, but will not converge to a maximum value, i.e, the path would probably lead to a local optimum, we still add diversity by extensive mutation.

   Initially, I mutated only 1 bit in the entire child produced after crossover. This didn’t give great results in terms of final fitness%. So, I decided to mutate multiple bits (multi-bit mutation). The max_mutation_size hyperparameter is set, and the number of bits mutated for every child is a random number between 0 and max_mutation_size.

4. **Keeping mutated children:**

   Instead of mutating children with some "probability", I **always** mutated all children formed. This was done as mentioned in the previous point because mutation improves diversity and may help to steer us out of a path leading to local optima.

   So, each call to mutate() would return 2 children: mutated versions of the children formed after crossover. This way, I retained important and "good" genetic material, and also improved diversity in the population

5. **Include a hyperparameter "elitism":**

   Elitism was defined as a hyperparameter (0 - 1 range) that defines how much fraction of the population (ordered descending by fitness function value) would be transferred onto the next generation. Hence, (1 - elitism_rate) * population_size members (the last members with low fitness function values) are removed at every iteration, hence eliminating unfavourable genes.

6. **Early-Stopping after a certain number of iterations:**

   To improve the time taken by the program, I decided to terminate the program if the fitness doesn't improve over several generations (iterations). The value of this certain number of generations is a hyperparameter, "iterations_to_terminate", and its value depends on m. Because for larger m, even after 800-1000 generations of constant fitness%, I found the fitness may improve. For smaller m, the value required was smaller, for example, for m <= 120, I used the value 350.

Approaches that failed to improve GA:

1. Increasing population size:

   I tried to increase population size to large values like 100, 200, 400 etc. However, it didn’t have a great impact on the fitness%, at the same time, it considerably slowed down the program, because there are multiple sorting operations involved in the program. After experimentation, it was best to choose population size = 20, this value gave best fitness% values, at the same time, it took a moderate-low amount of time

2. Modifying elitism rate:
   - Making the elitism rate small:

     For elitism rate < 0.4, the model failed to converge quickly

   - Making the elitism rate big:

     For elitism rate > 0.9, the model again, did not perform well

   After multiple experiments, I found that elitism rate = 0.6-0.7 works best. I finally implemented GA with the rate = 0.7

3. Different forms of mutation:

   I tried various forms of mutation: Single point, Multiple point, FlipGA (iterate left to right, flipping only if fitness% would increase, then another iteration right to left), etc.

   However, after multiple tries in each of these methods, a simple Random-multiple point mutation worked best. This was explained previously under “Extensive multi-bit mutation”

4. Different forms of crossover:
   - Single point crossover: As mentioned in the Textbook
   - 2-point crossover: As explained under the section with the same name. This is the final method implemented
   - Copy-parent crossover: Only one child is created here. Those genes where parent1[i] == parent2[i] are directly copied, i.e child[i] = parent1[i], and the other bits, where parent1[i] != parent2[i] are randomly set to be either 0 or 1

   After multiple tries in all these methods, 2 point crossover worked best for me, in terms of both fitness% and efficiency in time

Q4 Where GA might find it difficult to find a good solution:

- Genetic algorithms are highly sensitive to the initial population used. As I noticed during the experimentation, even for lower values of m, sometimes the algorithm gets stuck at ((number of clauses) - 1), and fails to converge correctly within 45 seconds. Most other times, it would solve and find a correct variable assignment within 2 seconds, hence the solution is random and dependent on the initial population used. Since the initial population is created in a random fashion, the GA may fail sometimes, or perform extremely slowly.
- Usually GA and different evolutionary algorithms, like particle swarm optimization (PSO), have a big stochastic component. This means that one needs to find a statistical convergent solution after multiple simulations/experiments.
- **Local Optima**: It is possible that GA gets stuck in a local optimum, and even mutations and crossovers are unable to remove the algorithm from the local optima. In this case, it usually gets stuck here for a very large number of iterations, sometimes more than 1000, and fails to converge to the global maximum. From my experience, a lot of local optima occur quite close to 100% satisfiability, even for lower values of m.

Q5 What does the above graphs tell you about the difficulty of satisfying a 3-CNF sentence in n variables. When does a 3-CNF sentence become difficult to satisfy?

For given n = 50, GA was able to satisfy sentences with clauses upto around m ~= 200. After m = 220, GA was finding difficulties in converging quickly to 100% clauses satisfied. So, I believe, the difficulty of satisfying 3-CNF depends on the **ratio m/n**: The ratio of clauses to variables. It is easier to satisfy 3-CNF if this ratio is smaller, and vice versa

So, if m/n is high, in general, if m/n > ~(3 - 4), it becomes more difficult to satisfy the 3-CNF problem