Programming Assignment I

Artificial Intelligence (CS F407)

Genetic algorithm for satisfying 3-CNF sentences

Varun Singh
2018B5A70869G

Data Representation

50-bit bitstrings (stored in 64-bit integers for efficiency).

For 3-CNF sentences in 50 variables with m clauses:

A clause input [1, 2, -3] meaning $(x_1 \vee x_2 \vee \neg x_3)$ becomes a pair of bitstrings:
P = 0x0000000000000003 = $\ldots 0011_2$
N = 0x0000000000000004 = $\ldots 0100_2$
where the set bits in P (N) denote positive (negative) literals in the clause.

A 3-CNF sentence with m clauses becomes a list of m such bitstring pairs.

A valuation (i.e. model) for a sentence is also naturally expressed in a similar way.
v = 0x0000000000000007 = $\ldots 0110_2$ means assigning $x_2 = x_3 = True, x_1 = False$.
Set bits in v denote which literals are assigned True.

Bit manipulation outperforms string processing. And it lets us translate the algorithm's operations to vectorized instructions on numpy arrays.

Fitness Function

Checking whether a valuation satisfies a sentence now becomes a matter of counting non-zero elements in the m-tuple $S = \{ (v \wedge P_i) \vee (\neg v \wedge N_i) \mid 0 \leq i < m \}$.

A non-zero value for the i[th] member of S implies v assigns True (False) to some positive (negative) literal in the i[th] clause.

$$Fitness(v) = CountNonZero(S)$$

which is proportional to percentage of clauses satisfied, but avoids floating-point arithmetic (division by m).

Population Size

Uniformly random 50-bit bitstrings can form the initial population of possible valuations for a given sentence.

$$\text{Population } = \{ v_i \mid 0 \leq v < 2^{50} \ \ and \ 0 \leq i < population\ size \}$$

Large populations tend to converge in fewer iterations (better exploration).
Small populations allow quicker iterations (fewer computations/iter involved).
Increasing population size involves a tradeoff: slowing down iterations vs. converging in fewer iterations.

Changing population size from 20 to 100 worked best on samples.

Seeding a schema to fix pure literals

Choosing uniformly random integers in the range $[0, 2^{50})$ for the initial population wastes information present in the sentence. Borrowing a notion of 'pure' literals from the DPLL algorithm, we can seed a schema for a better start.

Some literals appear *only* as positive or negative literals in the clauses of the sentence. Fixing them to be True (or False) makes the search space smaller. Changing them can only lead to a worse solution with unmet clauses.

Example: $(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$
a appears only as a positive literal. c appears only as a negative literal.
Search space can be restricted to valuations of the form 3'b0X1 where only b can vary.
3'b0X1 is a schema.

Fixing bits for pure literals in the starting population (combined with selective mutation described later) is very effective for shorter sentences.

Set bits in P_i represent positive literals in ith clause.
Set bits in N_i represent negative literals in ith clause.
Set bits in $A = \vee_0^{m-1} (P_i)$ tell which ones appear as positive literals in the sentence.
Set bits in $B = \vee_0^{m-1} (N_i)$ tell which ones appear as negative literals in the sentence.
Set bits in $(A \wedge \neg B)$ tell which ones appear *only* as positive literals in the sentence.
Set bits in $(\neg A \wedge B)$ tell which ones appear *only* as negative literals in the sentence.

Now,

$v \leftarrow \left(v \ \vee (A \wedge \neg B)\right)$	Set bits for pure positive literals.
$v \leftarrow \neg v$	Invert bits
$v \leftarrow \left(v \vee (\neg A \wedge B)\right)$	Set bits for pure negative literals.
$v \leftarrow \neg v$	Invert bits

ensures that v conforms to a schema that fixes pure literals.

These operations need not be repeated after seeding initial population if mutation ignores bits for literals which only appear in satisfied clauses.

Selective mutation of literals in failing clauses

When all valuations in the population have very high fitness, they become similar and improvement plateaus. The algorithm is essentially waiting to flip some critical bit by random chance. Changing literals that don't appear in an unmet clause doesn't affect fitness.

If mutation randomly picks a literal in some *failing* clause, only relevant bits are flipped and fitness almost certainly rises or falls.

This improvement comes cheap because failing clauses correspond to zero entries in S, which are easily found.

Elitism

Sometimes maximum fitness may decrease in successive generations because the fittest individual was lost.

Carrying the fittest individual to the next generation with no change ensures maximum fitness never drops.

Restart on plateau

If fitness doesn't improve for around 8 seconds, the algorithm starts afresh. This ensures it explores the search space instead of spending most of its 45 seconds exploiting a dead-end.

Performance of the improved algorithm

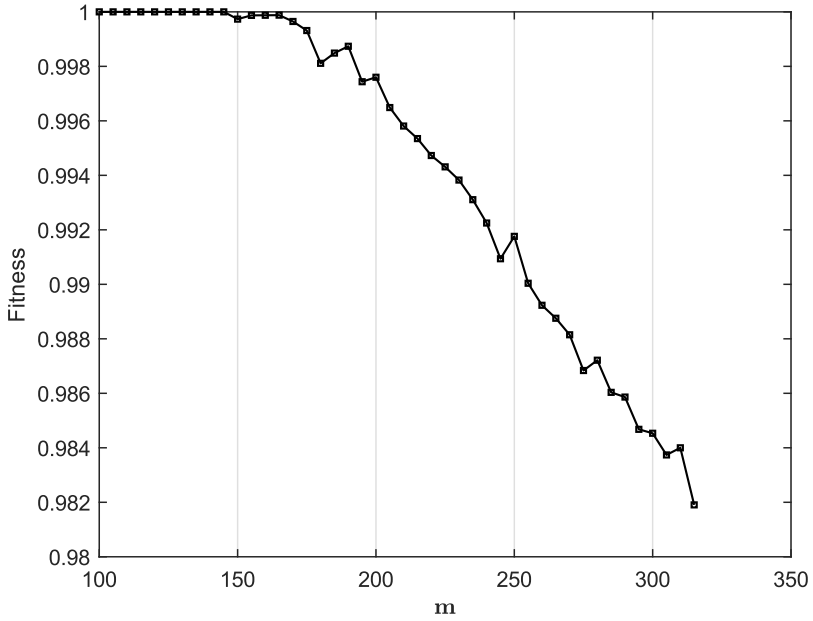


Fig. 1 : Average fitness of the best models found for 50 randomly generated 3-CNF sentences in 50 variables with m clauses.

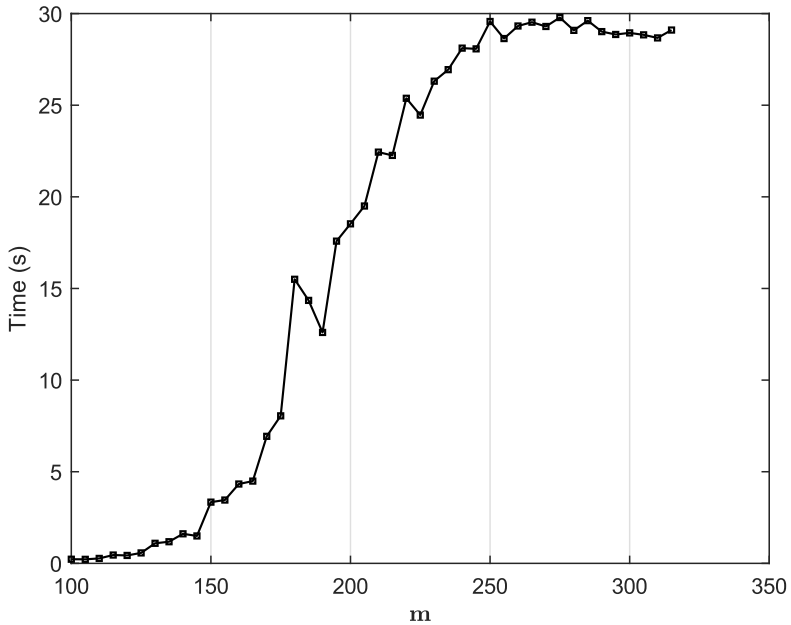


Fig. 2 : Average time elapsed (in seconds) to find the best models for 50 randomly generated 3-CNF sentences in 50 variables with m clauses.

Contradictory clauses are likely to arise with rise in $\frac{m}{n}$

For sentences with $m \leq 160$, all clauses were satisfied within seconds. (Fig. 1)
Shorter sentences have many solutions in the search space of 2^{50} bitstrings.
Fewer clauses allow many ways of satisfying the sentence.

Longer sentences may be unsatisfiable. GA times out with < 1 fitness. (Fig. 2)
In the same search space of 2^{50} bitstrings, high fitness solutions are rarer and may not be diverse. Clauses limit ways of satisfying the sentence..

GA finds it difficult to find a good solution in a large search space where solutions are sparse.

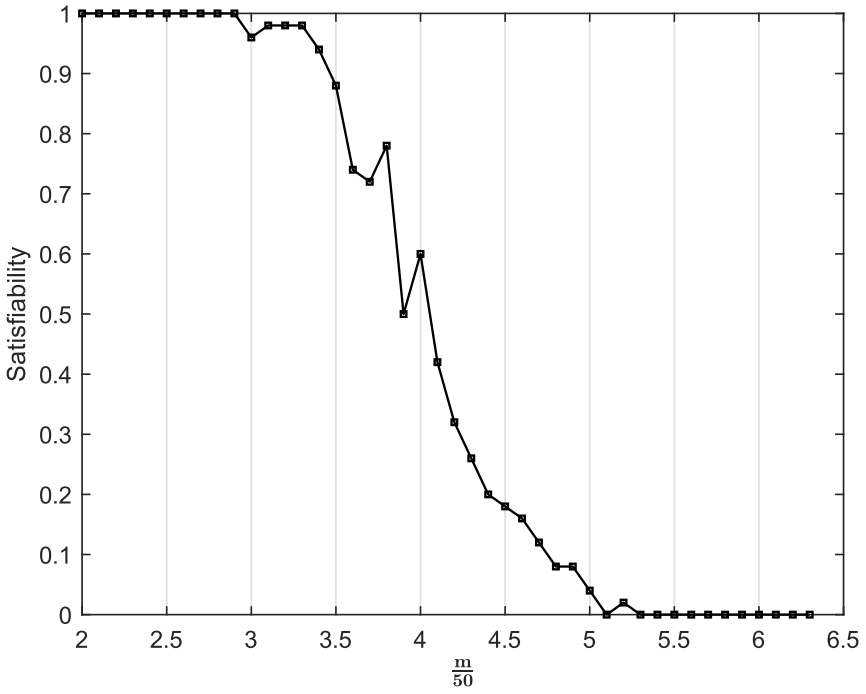


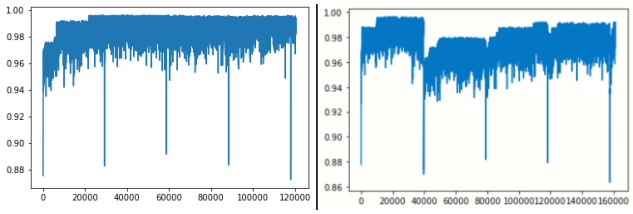
Fig. 3 : Fraction of sentences satisfied among 50 randomly generated 3-CNF sentences in 50 variables with m clauses.

Difficulty of satisfying a random 3-CNF sentence in *n* variables with *m* clauses rises sharply beyond a threshold of $\frac{m}{n} \approx 3.25$. And when this ratio crosses ~5, they are almost certainly unsatisfiable.

Failed approaches

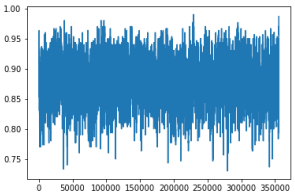
(All figures show maximum fitness plotted against iteration count.)

- Seeding fittest individual across restarts puts the algorithm back in the rut it just escaped.

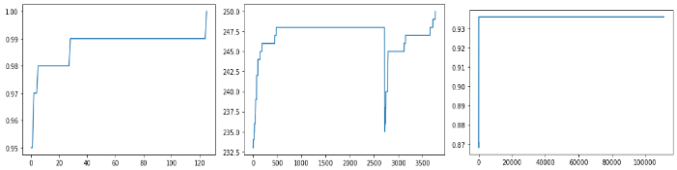


The sharp drops in fitness mark restarts. Notice how clean restarts lead to different fitness values in the second picture. On the left, restarts are wasted.

- Culling unfit individuals discouraged exploration of the search space.
- Very high mutation rates made convergence difficult.



- Very large population size makes each iteration take longer, which allows fewer generations in evolution.
- String representation of data may allow for elaborate crossover/mutation schemes, but string processing is slow.
- Selecting crossover point between two parents based on ratio of fitness effectively culls unfit individuals too early.
- Tried checking rolling mean of maximum fitness (with elitism) for cut-off criterion. This does not work because number of iterations a plateau may last for is unpredicatable. Fitness may abruptly improve, spike after a restart, or remain unchanged until 45 seconds elapse.



A simple time limit on duration for which no improvement in fitness is tolerated worked better.

References

- "Genetic Algorithm." *Wikipedia*, Wikimedia Foundation, 19 Sept. 2021, https://en.wikipedia.org/wiki/Genetic_algorithm.
- Russell, Stuart J., Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach.* 4th ed. pp. 129
- "Boolean Satisfiability Problem." *Wikipedia*, Wikimedia Foundation, 5 July 2021, https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.
- Janson, Svante, et al. "Bounding the Unsatisfiability Threshold of Random 3-Sat." Random Structures & Algorithms, 1 Sept. 2000, https://dl.acm.org/doi/10.5555/351830.351832.