Using genetic algorithms to solve the satisfiability problem

- *Studenti: Hristodor Minu-Mihail, Căutișanu Andrei-Constantin*
- *Grupa și anul: II E2*
- *Tema T3*

Introduction

The boolean satisfiability problem's (SAT) input is a boolean formula F in conjunctive normal form, consisting of a set of n boolean variables $x_1 \dots x_n$ and m clauses $c_1 \dots c_m$. The clauses are disjunctions of two or more literals (a variable x_i or its negation). The formula is satisfiable if there exists a truth assignment to the variables which satisfies every clause. Our goal is to determine if such an assignment exists or (the SAT problem), if we cannot find it or it does not exist, determine a truth assignment that satisfies the maximum possible number of clauses (the MAX-SAT problem).

The satisfiability problem can be divided in classes named k-SAT, where each clause contains a maximum of k literals. 2-SAT is solvable in polynomial time, but for any k >= 3, the k-SAT problem is NP-complete, and therefore we cannot use a polynomial algorithm to solve it, meaning it is not feasible to use a deterministic method for large instances containing some hundreds of literals.

The evolutionary approach

To attempt to solve the problem in a more reasonable time, we will use an evolutionary algorithm, which, while faster than the traditional deterministic methods, does not guarantee finding the assignment that satisfies all the clauses, although it can get very close as we will see in the results of our benchmark instances.

A genetic algorithm takes set of encoded potential solutions called the "populations" as input and uses a "fitness" function to evaluate each candidate solution (genome). The algorithm simulates the process of natural selection, creating a new "generation" of the population by giving the better genomes a better chance to reproduce using crossover and mutation.

Implementing a genetic algorithm for the MAX-SAT problem

- **SOLUTION ENCODING** We encode our solution as an array of bits of length equal to the number of literals in the input formula. The value of each bit corresponds to the truth value assigned to the literal of the same index. (i.e. if sol[i] = 0, then x_i is false, if sol[i] = 1, then x_i is true). Each generation has a size of 100 individuals.
- **FITNESS AND GENERATIONS** In our algorithm, a solution's fitness is equal to a number in the interval [0, 1] equal to the ratio of clauses in the formula that solution satisfies. We established a limit of 2000 generations for each instance in our experiment.
- **SELECTION AND ELITISM** Each generation, we use elitism to transfer a certain number of the best individuals to the next generation without changing them. This is to ensure a quality standard of the parents in each generation, while also reducing the run time by not having to apply crossover and mutation on all the genomes. We experimented with various elitism rates, and we found that a rate of 50% was preferable for a steady increase in the early generations, reducing it to 30% and 20% after 250 and 500 generations,

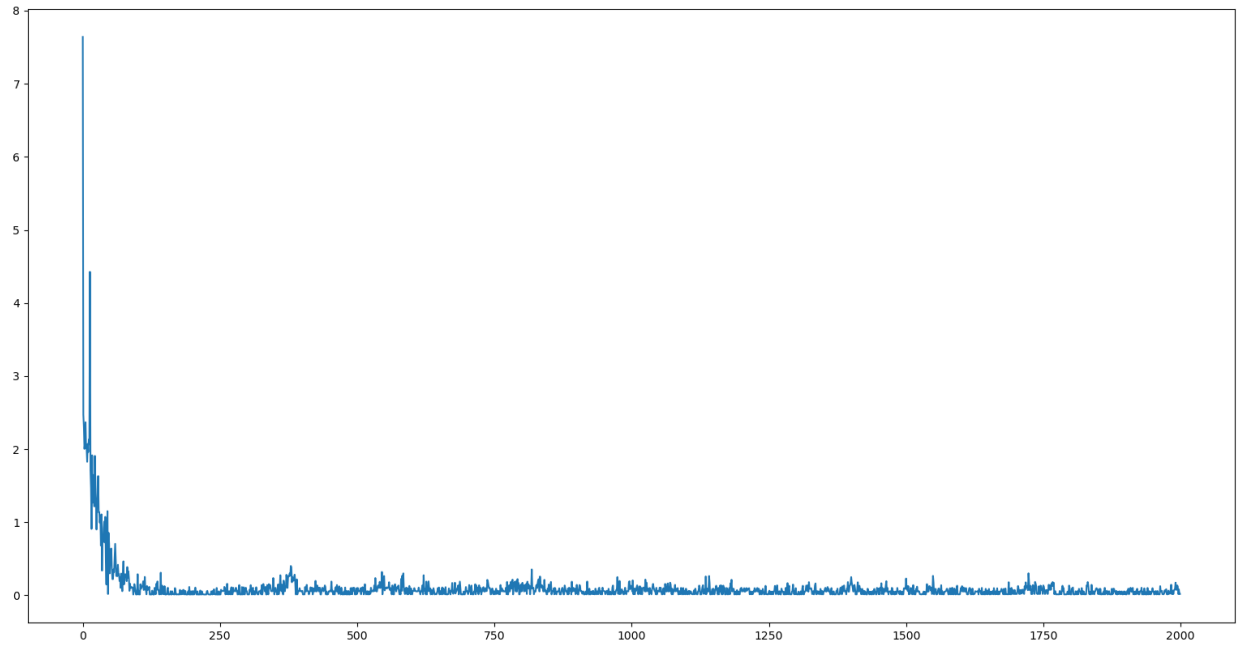
respectively. After transferring the best individuals, the rest are selected for reproduction using **roulette wheel selection**, where an individual's chance to be selected is proportional to its fitness. We select 2 parents, apply crossover and mutation according to our crossover and mutation rates, and the resulting offspring is transfered to the next generation. This process is repeated until we fill up the remaining places in the new generation after elitism.

- **CROSSOVER AND MUTATION** The crossover method we use is **Two point crossover**, where we randomly select two points between 0 and the length of the individual, and then exchange the bits within the limits of those two points between the individuals. The mutation flips one random bit in the individual if it was randomly selected according to our mutation rate. Our initial crossover rate is 70%, meaning we will apply it between the 2 selected parents 70% of the time, while leaving it open the other times. The initial mutation rate, which is the probability of an individual being selected for mutation is 1%. Both rates are variable, increasing as the algorithm progresses to expand the search space and to increase the chance of finding better solutions.

Generations	Mutation Rate	Crossover Rate	Elitism rate
0-250	1%	70%	50%
250-500	2%	75%	30%
500-1000	5%	90%	20%
1000-2000	10%	100%	20%

Table 1 - Variation of genetic parameters throughout the algorithm

- **FITNESS CACHE** Each fitness value we calculate is being stored in a dictionary structure as we calculate it for each individual. This way, should we encounter an individual that is the same as a previous one, we no longer have to calculate its fitness again, instead we directly refer to its stored value in the dictionary. This significantly improves runtime as we advance further in generations.



Graph 1 - evolution of the runtime (Y axis) of each generation (X axis) in seconds over one instance of the algorithm using the 450 literals formula

Our final algorithm looks like this:

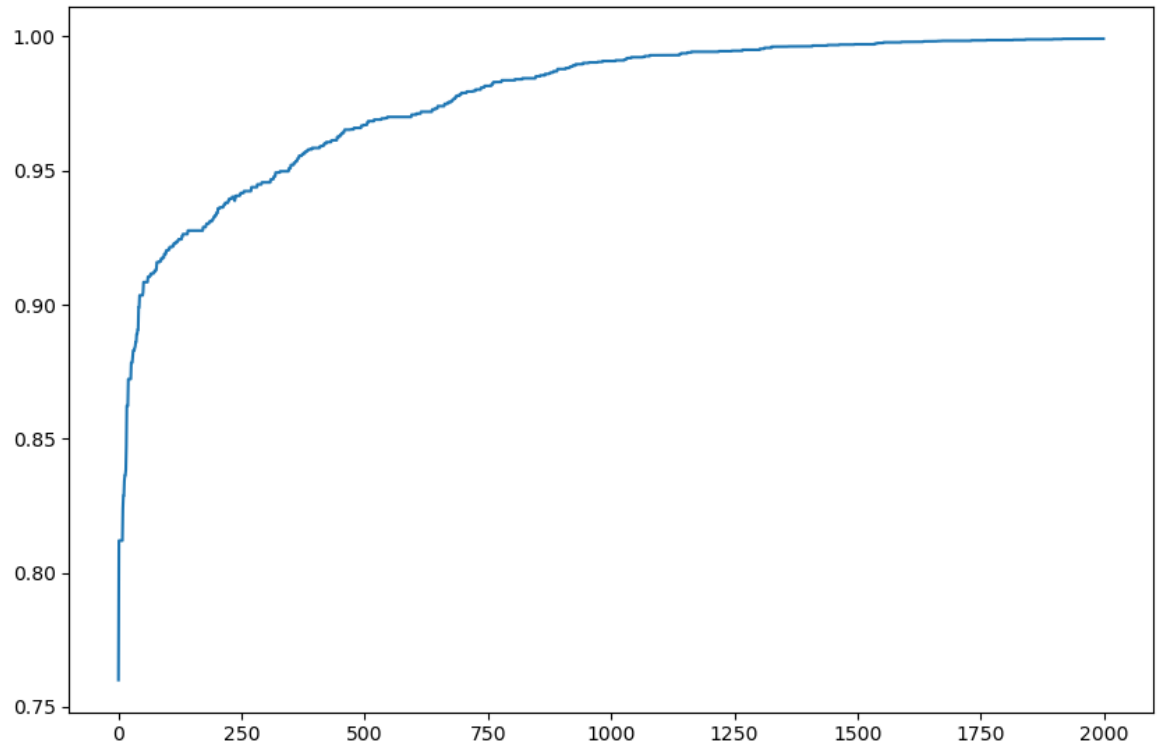
```
generate random population
set initial ELITISM_RATE, CROSSOVER_RATE, MUTATION_RATE;

for generation in [0, 2000] {
    sort the population by fitness;
    transfer the best POPULATION_SIZE * ELITISM_RATE individuals to next generation;

    while there are less than POPULATION_SIZE individuals in next generation {
        select 2 individuals using Roulette-Wheel;
        apply Crossover depending on CROSSOVER_RATE;
        apply Mutation depending on MUTATION_RATE;
        transfer the resulting children to next generation;
    }
}
```

RESULTS

We ran the algorithm on 8 different instances each with a different number of literals and clauses 30 times. We have also run a random search 30 times for each instance, generating 5000 random solutions each time. The results are represented by the best individual fitness in the 2000th generation (the fitness being the ratio of clauses satisfied by the solution).

No. of literals	No. of clauses	GA Average	GA St. Dev.	RS Average	RS St. Dev.
450	19 084	0.9990	0.0001	0.8046	0.0080
595	29 707	0.9951	0.0008	0.8007	0.0075
760	43 780	0.9940	0.0013	0.7928	0.0046
945	61 855	0.9877	0.0029	0.7911	0.0055
1150	84 508	0.9766	0.0028	0.7902	0.0042
1272	98 921	0.9713	0.0034	0.7872	0.0051
1400	114 668	0.9642	0.0061	0.7843	0.0049
1534	132 295	0.9607	0.0080	0.7827	0.0055

Table 2 - results. GA = genetic algorithm, RS = random search



Graph 2 - evolution of best fitness (Y axis) throughout the generations (X axis) one instance of the algorithm using the 450 literals formula

CONCLUSION

As we can see, the algorithm does very well in getting close to finding a solution that satisfy a very high percentage of the clauses in the given formulas. As such, it is certainly a viable alternative to the exponential time deterministic method for the MAX-SAT problem, as it is able to get good results in a rather short time. However it is not quite able to solve the satisfiability problem, as every formula we've used in this experiment is satisfiable, and yet the algorithm cannot find the truth assignment that satisfies every clause in the generation limit we've imposed.

REFERENCES

- http://sites.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm
- A. Bhattacharjee et al. / Advances in Science, Technology and Engineering Systems Journal Vol. 2, No. 4 https://www.astesj.com/publications/ASTESJ_020416.pdf
- J. Gottlieb et al. / Evolutionary Algorithms for the Satisfiability Problemhttp://www.cs.ru.nl/~elenam/fsat.pdf