Solving the Logical Satisfiability Problem (SAT) with heuristic methods

Robert Găină, David Viziteu

January, 4th 2020

Abstract

This report attemps to solve the Logical Satisfiability Problem (SAT) using two heuristic methods. In this paper we propose a Simulated Annealing aproach and different versions of Genetic Algorithms for solving the SAT problem. Results obtained are compared side by side outlining the efficiency of each other on a set of instances. The authors attempt to find the best suited technique for solving a SAT instance.

1 Introduction

The satisfiability problem is one of the NP-complete problems meaning that there is no known algorithm that efficiently(in polynomial time) solves any SAT instance. The most efficient algorithms known use local optimization, assigning truth values to variables until we get a conflict or the solution. After getting a conflict the algorithm backtracks changing its value, therefore the time-complexity of backtracking is not feasable for hundreds or thousands of variables. With general belief that no time efficient local optimization search algorithm exists, heuristic SAT-algorithms are the preffered approach in finding a solution as they can compute a result in a shorter time.

A boolean satisfiability problem, abbreaviated as SAT, involves a boolean formula consisting of a set of variables. Usually, the form is in conjunctive normal form (CNF) meaning that it is a conjunction of clauses, with each clause being a disjunction of literals. This formula is satisfiable if there exists a truth assignment to the variables such that each clause is satisfiable. The goal is to determine a permutation of values to the variables such that the formula given is satisfiable.

There are two approaches which will be used in this paper. The first one is Simulated Annealing and the other one being Genetic Algorithms.

2 Methods

2.1 Algorithm

1. **Simulated Annealing:** The concept of annealing comes from metallurgy; where metals are heated to a high temperature and cooled at a controled rate to avoid defects in the structure. The idea behind this implementation is that a high temperature allows jumps out of local maxima trading a better solution at a moment for a worse one that can be improved

more later in the evolution process. The algorithm will start with a random candidate solution and a temperature randomly generating a number of random candidates. There are two conditions that should be estabilished: the termination-condition(inner-loop) and the halting criterion(outer-loop). The halting criterion is usually related with the temperature; it starts with the initial value of the temperature that decreases at each iteration towards an equilibrium state. The three main elements of this loop are the initial temperature, the cooling rate and the stopping criterion. In the inner-loop a random neighbor of the solution is evaluated. If it is better it is saved as the new solution and if not we are evaluating its chances of replacing the current value. Usually the acceptance criteria is chosen such that the probability of accepting a worse solution decreases with the temperature.

2. **Genetic Algorithm:** Since genetic algorithms simulate the process of natural selection the solutions that can adapt to the desired environment are able to survive and reproduce to the next generation. Each generation is a population on individuals (chromosomes) and each individual represents a starting point in the search space of possible solutions. Therefore, the genetic algorithm initiates its search from a population of points, not a single one. Similar to natural genetics, these solutions must face improvements over time so these modifications consists of mutations and crossovers. In order to keep the survival of the test concept, to each individual we apply a fitness function to rate their quality. This fittness function must be chosen such that the value of this function is higher for the best solutions. This way, the better individuals are given a higher chance of selection over generations evolving better solutions until reaching a stopping criterion.

2.2 Implementation

The implementation of the two methods follow the guidelines from 2.1 section with the following additions:

2.2.1 Simulated Annealing:

- Binary format solutions – the length of the solution equals the number of variables in the formula.
- The notion of neighbour means neighbour in Hamming space of distance one.
- The initial temperature is 100.
- The halting condition is $temperature < 10e^{-8}$ for smaller instances while for big instances we are stopping after a number of 100 iterations without any improvement.
- Temperature decreases by 5%.
- Inner loop is an iteration of 10.000 steps.

2.2.2 Genetic Algorithm

- The population is a set of of fixed-length bitstrings equal with the number of variables in the formula.
- The quality of the chromozomes is evaluated based on the number of clauses satisfied.

- Selection proportional with the fitness value (Roulette-Wheel).
- Resulting chromosomes replace their parents.
- Different types of mutation and cross-over used.

The mutation functions used:
1. Flip multiple bits (chosen randomly) - iterate through all the bits of every chromosome of the population an flip each one with a certain probability
2. Greedy-Flip (Left-Right) - For each chromozome of the population iterate through each bit from left to right ONCE and flip each bit that improves the chromozome
3. Multi-Greedy-Flip (Left-Right) - For each chromozome of the population iterate through each bit from left to right and flip each bit that improves the chromozome until there is no more improvement possible.

Out of these functions only the first one is dependent on a mutation rate. The others are greedy approaches that will be applied to all the solutions.

Cross-over functions used:
1. Single-cut-point crossover - Randomly select a pivot point and exchange the substring from the pivot towards the end between two chromosomes.
2. Uniform-crossover - Exchange each pair of bits between two chromosomes with a fixed 0.5 probability.

3 Experiments

The experiments were conducted on 10 different instances ranging from 20 variables and 91 clauses up to 1534 variables and 132295 clauses. Out of those instances one was chosen to not be satisfiable. The experiment started with a simple Simulated Annealing and a simple Genetic Algorithm. Even if the classic Simulated Annealing approach proved to be rather efficient the Genetic Algorithm returned rather bad results ($\sim 20\%$ succes rate) so we have decided to use a greedier method of improving the chromosomes. In order to allow a wider exploration the uniform-crossover was chosen. The mutation probability (0.01) and cross-over probability (0.7) are fixed. The random number generator is the standard implementation of a Mersenne Twister.

Algorithm 1 Pseudocode of GA2

```
t=0
generate the starting population P(t)
evaluate P(t)
while not StoppingCondition do
    t=t+1
    select P(t) from P(t-1)
    uniformCrossover P(t)
    mutate P(t)
    multiGreedyFlip P(t)
    evaluate P(t)
end while
```

Where: StoppingCondition=1000 for GA1 and 100 for GA2, PopulationSize=100

Alg	**Sample**	**Vars**	**Clauses**	**CLU (bst)**	**% solved (avg)**	**CLU (avg)**	**Std dev**
SA	uf20-91	20	91	0	100%	0	0
	pigeon hole*	42	133	1	99.25%	1	0
	uf250-1065	250	1065	0	100%	0	0
	frb35-17-1	450	19084	2	99.99%	3.43	0.71
	frb40-19-1	760	43780	3	99.99%	3.93	0.86
	frb45-21-1	945	61855	3	99.99%	5	0.9
	frb50-23-1	1150	84508	4	99.99%	5.54	1.02
	frb53-24-1	1272	98921	4	99.99%	5.75	0.86
	frb56-25-1	1400	114668	5	99.99%	6.75	1.25
	frb59-26-1	1534	132295	3	99.99%	6.98	1.34
GA 1	uf20-91	20	91	0	100%	0	0
	pigeon hole*	42	133	1	99.25%	1	0
	uf250-1065	250	1065	0	100%	0	0
	frb35-17-1	450	19084	14900	22.93%	15025.75	550.98
	frb59-26-1	1534	132295	97761	22.82%	104671.28	7695.61
GA 2	uf20-91	20	91	0	100%	0	0
	pigeon hole*	42	133	1	99.25%	1	0
	uf250-1065	250	1065	0	100%	0	0
	frb35-17-1	450	19084	2	99.99%	2.87	0.53
	frb40-19-1	760	43780	1	99.99%	2.13	0.56
	frb45-21-1	945	61855	2	99.99%	2.49	0.63
	frb50-23-1	1150	84508	3	99.99%	2.93	0.43
	frb53-24-1	1272	98921	3	99.99%	2.87	0.45
	frb56-25-1	1400	114668	6	99.99%	7.54	1.57
	frb59-26-1	1534	132295	4	99.99%	5.15	0.91

*pigeon hole instance is not satisfiable

Table 1: Comparison of minimas obtained vs expected results (out of 30 runs each)

4 Observations

4.1 Corectitude of solutions for SAT

Since both Simulated Annealing and Genetic Algorithm are both approximation algorithms, they are most efficient in approximating solutions to optimization problems(in particular for NP-hard problems). However, when talking about the SAT problem the notion of partial/approximation solution is not enough. No matter how many clauses of our instance are satisfied, a solution that doesn't satisfy all of them is equally bad. Even if from our experiment at least 99% of the clauses were satisfiable all results below 100% are wrong for our SAT problem. Therefore heuristic methods are more useful in problems where an approximation of the solution is relevant. (e.g. TSP-problem, P-coloring...)

4.2 Existence of solutions for SAT

Another drawback of both algorithms is the fact that none is able to check the existence of a solution for a instance. Pigeon hole instance is an instance for SAT problem based on the Dirichlet's box principle, stating that given n object to be placed into k boxes ($k < n$), no matter how we arrange the objects at least one box contains $\lceil n/k \rceil$ objects. This statement can be modeled into a unsatisfiable formula in CNF form (meaning that there is no assignment that satisfies all clauses) , therefore could be solved with a SAT solver algorithm. Even if the number of variables and clauses were small a lot of computing power was wasted by both algorithms because none was able to check if a solution exists for the test given. For instance, when running the test uf250-1065 on our Greedy Genetic Algorithm the result was returned faster than pigeon hole even if the size of the input was bigger. Therefore, without a method of checking the existence of a solution for SAT problems both algorithms are wasting time with unsatisfiable instances.

4.3 Simulated Annealing

In the previous experiments that compared multiple heuristic algorithms, simulated annealing proved to return the best results, therefore it was chosen for this problem. From this experiment we can see that the simulated annealing yielded great results, because a permissive inner-loop of 10.000 iterations allowed for a high-exploration of solutions, especially useful for big instances. Another advantage of this method is the time increase is not liniar with the problem size.

4.4 GA1

The first genetic algorithm used was a simple one where the mutation was made with a fixed probability of 0.01 and then we applied a single-cutpoint crossover. This method proved to be unsuccessful with results similar with a random search on bigger instances. Since the number of clauses was big and the fitness function was proportional with the number of clauses, the pressure of selection was so low that the genetic operators were not able to improve the chromozomes over generations. This method is a fast way of approximating a solution when the number of clauses and variables is reduced.

4.5 GA2

Hence the first approach towards the genetic algorithm produced sub-par results for big instances we decided that the improvements over generations must be more well-defined. In order to obtain more visible differences between parent and child chromozomes we have decided that improvements simmilar to greedy picks must be implemented.

Since in the previous algorithm the chromozomes were not evolving fast enough the first decision was to flip each bit that improved the number of clauses satisfied for a solution instead of random mutations once. In the second attempt we tried to improve the chromozome until no more improvement was found. However, because this process of flipping was computationally expensive and the convergence towards solutions was rather fast we used the second flipping method with a smaller number of generations. In order to assure an extended search space we have concluded that a uniform crossover would be superior compared to the single cutpoint crossover.

4.6 Function evaluation

A common function that all three algorithm uses is the process of evaluating the number of satisfied clauses. As the solutions improve the time required for evaluating this function value decreases because there is a higher probability of finding that a clause is satisfiable from the first terms of the paranthesis. Therefore, iterations get faster as we progress in the algorithm and are directly influenced by this evaluation function.

4.7 Convergence

All three algorithms converge differently towards the solution. While simulated annealing tends to stabilize as the temperature decreases and the simple genetic algorithm (GA1) brings smaller improvements close to the initial solution, the Multi-Greedy-Flip version of the Genetic Algorithm quickly descents at over 99% satisfiability from the first generations. From figures (1),(2) and (3) we can see how all these functions converge on test frb35-17-1 (450 variables, 19084 clauses) .

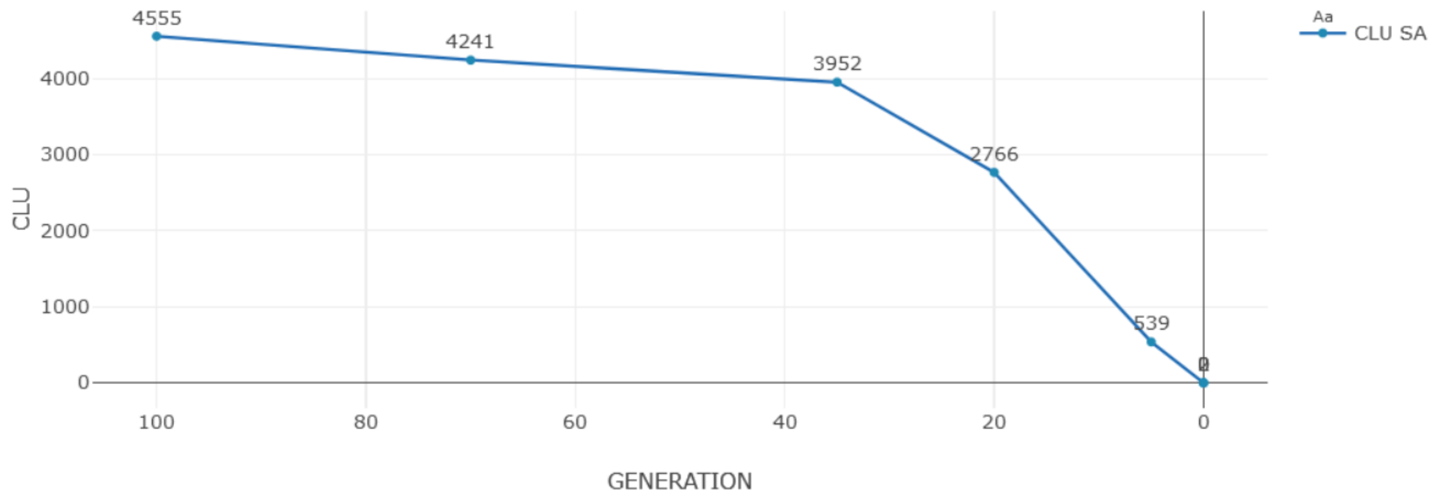

Figure 1: Convergence of Simulated Annealing

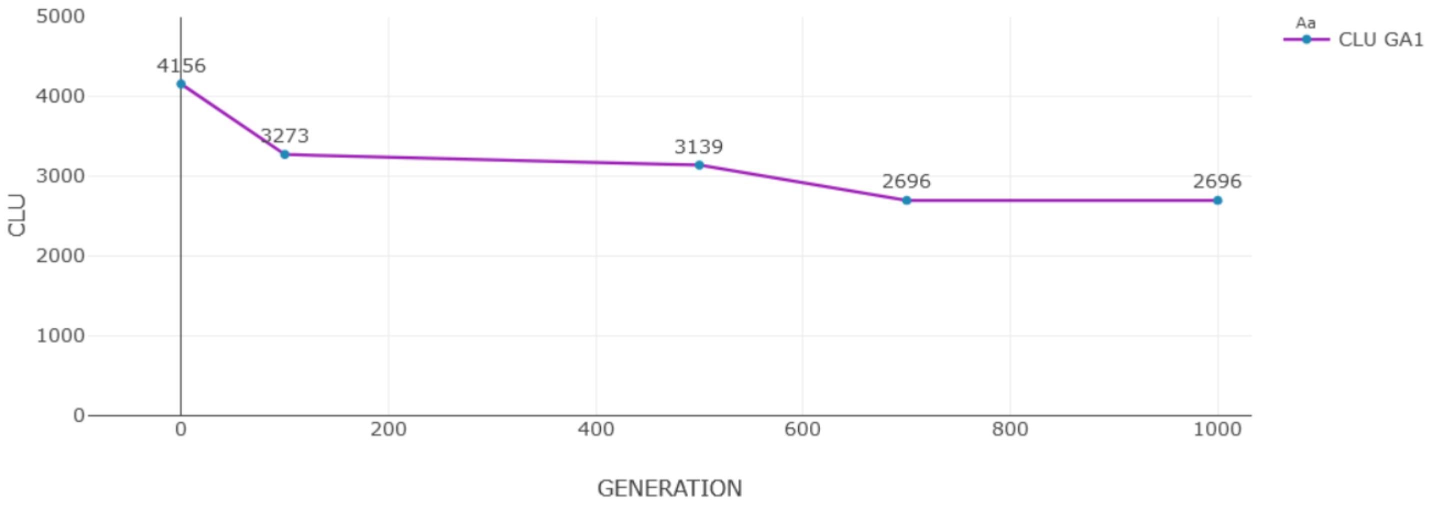

Figure 2: Convergence of GA1

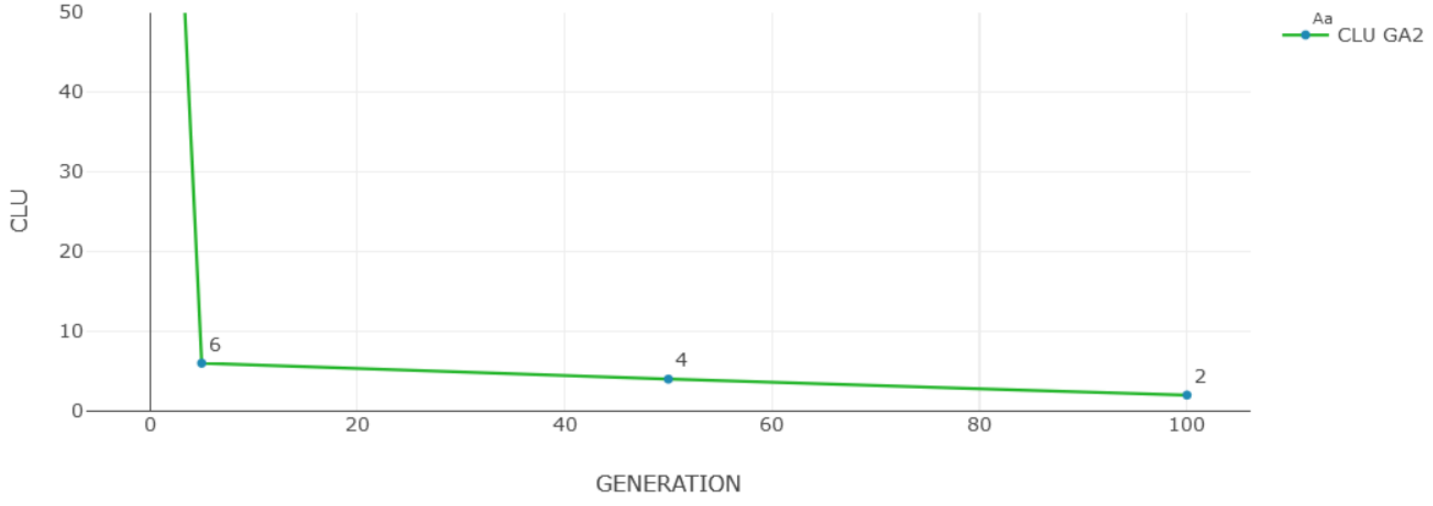

Figure 3: Convergence of GA2

5 Conclusion

In this paper, we presented 3 heuristic approaches to solving the SAT problem: Simulated Annealing, a classic Genetic Algorithm and a fine tuned version of the classical GA (GA2). The fine-tuned version includes a mutation operator which resembles a greedy improvement (mutate all bits as much as possible and then keep the best out of the chromosome) and a uniform crossover. These two processes are complementary and they allow GA2 to explore the search space and to exploit particular interesting areas in a more effective manner. Moreover, this combination of crossover and aggressive mutation ensures a sufficient diversity in the involved population. GA2 has been evaluated on a set of 10 instances and has been compared with SA and a classic GA. Experimental results show that GA2 is very competitive compared with the two other mentioned algorithms, SA and classical GA. Every algorithm is also providing very interesting results and two of them (SA and GA2) appears to be effective for the SAT problem.

Our future work will consist of developing a better version of GA2 by using adaptive parameters and trying to avoid unsolvable partial solutions.

References

[1] Genetic Algorithms implementation
https://profs.info.uaic.ro/~eugennc/teaching/ga/

[2] Genetic Algorithms Theoretical concepts
https://profs.info.uaic.ro/~eugennc/teaching/ga/res/gaSlidesOld.pdf

[3] ASTES Journal, Solving the SAT problem
https://www.astesj.com/publications/ASTESJ_020416.pdf

[4] An improved Genetic Algorithm for 3-SAT, Huimin Fu
https://www.atlantis-press.com/journals/ijcis/25888772/view

[5] Genetic Algorithms general informations
https://www.geeksforgeeks.org/genetic-algorithms/

[6] HillClimbing and Simulated Annealing report
https://drive.google.com/file/d/1zjJ7JFevY8Yxao5lTzfkHLIEWXPug7JD/view?usp=
sharing

[7] Fitness function computing methods
http://students.info.uaic.ro/~vladut.ungureanu/Algoritmi-genetici-ID.pdf

[8] Report structure
https://gitlab.com/eugennc/research/-/blob/master/2019SYNASC/
synasc2015-paper18.pdf

[9] Influence of Population on the Genetic Algorithm
https://annals-csis.org/Volume_1/pliks/167.pdf

[10] Roulette-Wheel Selection
https://medium.com/datadriveninvestor/genetic-algorithms-selection-5634cfc45d78

[11] Graph editor
http://mathcha.io

[12] Simulated Annealing vs Genetic Algorithms
https://www.seas.upenn.edu/~cis391/Lectures/informed-search-II.pdf

[13] Sorting based on probability
https://stackoverflow.com/questions/37368787/c-sort-one-vector-based-on-another-one/
46370189

[14] Mersene twister random number generator
https://www.guyrutenberg.com/2014/05/03/c-mt19937-example/

[15] When a Genetic Algorithm outperforms Simulated Annealing
https://bit.ly/3lkcj8b

[16] GASAT: a genetic local search algorithm for thesatisfiability problem
overwww.info.univ-angers.fr/~lardeux/papers/ECJ06.pdf