UPPSALA
UNIVERSITET

Report for Natural Computation Methods for Machine Learning

Genetic Algorithm on SAT Problem

Group 06

Xiaoyue Chen Suling Xu Qinhan Hou

May 24, 2021

Abstract

abstract

Keywords: A, B, C

1 Introduction

The satisfiability problem (SAT) is a NP-complete problem [**cook˙1971**]. SAT problem consists a set of Boolean variables $x_1, \cdots, x_n$ and a Boolean formula $f : \mathbb{B}^n \to \mathbb{B} = \{0, 1\}$. The question is whether an assignment $x = (x_1, \cdots, x_n)$ exists such that $f(x) = 1$ [**gottlieb˙marchiori˙rossi˙2002**]. A wide range of problems can be expressed as SAT problems. This includes verification, planning, scheduling and combinatorial design [**biere2009handbook**]. Hence, a SAT solver has been a general purpose platform for solving various real-world problems and has obvious importance.

There are two types of SAT solver technologies: complete methods and incomplete methods. Complete methods can guarantee that it will eventually report a satisfying assignment or prove not satisfiable, if it run for long enough. Complete methods have an exponential time complexity, unless P = NP. On the other hand, incomplete methods has no such guarantee and generally use stochastic local search [**gomes˙kautz˙sabharwal˙selman˙2008**]. In many cases where the problem does not require a definitive answer, incomplete methods could outperform complete methods in terms of both speed and memory usage.

Genetic algorithms (GAs) are incomplete methods that have been applied to many NP-complete problems, including SAT [**gottlieb˙marchiori˙rossi˙2002**]. However, results suggest that classical GAs may not outperform local search algorithms [**de1989using**]. Nevertheless, recent results show that GAs can yield good results if combined with other methods [**gottlieb˙marchiori˙rossi˙2002**].

This paper proposes a genetic algorithm—GNTSAT, which is based on WalkSat [**selman1994noise**] and outperforms WalkSat in many benchmark instances. This paper introduces the GNTSAT algorithm, makes performance evaluations based on benchmarks, and discuss the possible reasons for the performance difference and their implications.

2 Backgrounds

3 Methods

We implemented a genetic solver in C++ using genetic algorithm with a local search improvement. The data we used for benchmarks is in DIMACS CNF format, which is a formula in conjunction (logical and) of a set of clauses. And we decided to use the randomly generated 3-SAT-instances that are satisfiable in this format.

A candidate solution is a string of bits whose length equals to the number of variables of the considered problem instance. To generate healthy population that could lead to the final solution, firstly we repeatedly generate random assignments for the instance until a certain number of population reached in the current generation. We use a population consists of 256 individuals. Then repeat actions on the whole candidate solutions: In each iteration, firstly select some best individuals. The fitness function we used to evaluate individuals is the number of satisfied clauses. Then crossover are performed on two individuals among the bests. Each generated child is improved individually using a local search process, and then new individuals are inserted and old individuals are removed in the population under some conditions. Iterate the process until a solution is generated, that is when the maximum number of satisfied clauses equals to the number of instance clauses. The general algorithm is illustrated in Fig1.

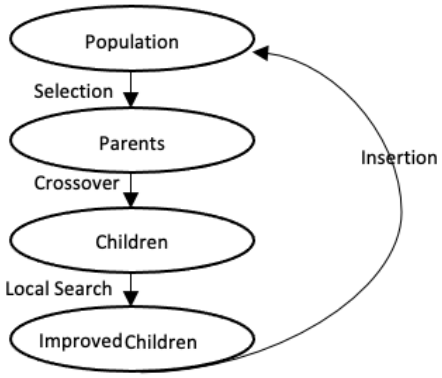


Figure 1: Algorithm Scheme

3.1 Selection Operation

For selection, we adapt part of the idea of tournament selection, which has small algorithm complexity of O(n) (where n is the tournament size). We randomly select a certain number of individuals and then selected the top few individuals in descending order of fitness. The number of individuals randomly selected as tournament size is set to 16 and the number of best individuals among the tournament size is set to 4.

3.2 Crossover Operators

The main goal of the crossover operator in our genetic solver is to create potentially promising new individuals. Using the randomly selected a pair of parents from the resulting selected individuals from the tournament selection, there are many ways for them to crossover, and we consider 5 types: CC (Corrective Clause) Crossover, F&F (Fleurent and Ferland) Crossover, Uniform Crossover, One-point Crossover and Two-point Crossover. We will introduce them respectively below.

CC (Corrective Clause) Crossover: For each clause c that is unsatisfiable for both parent_x and parent_y solutions and for each variables that appears in clause c, find the bit position using the literally absolute value i of the variable that produces the maximum improvement on two parents guided by the improvement evaluation function. The function equals to the number of false clauses which become true by flipping the ith bit in one of the solutions minus the number of satisfied clauses which become false. Assign the flipped value applied to one of the parents to the child in the bit position found, and all the bits with no value of child take the value in corresponding position of parent_x or parent_y with the equal probability.

F&F (Fleurent and Ferland) Crossover: For each clause c that is satisfiable for one parent and unsatisfiable for the other parent and for each variables that appears in clause c, the bits positions in the child, corresponding to the literally absolute value i of the variable in parents, are assigned values according to the parent satisfying the identified clause. All the bits with no value of the child take the value in corresponding position of parent_x or parent_y with the equal probability.

Uniform Crossover: With equal probability, each bits of the child is chosen from either parent. In that case, one new offspring is generated.

One-point Crossover: Randomly select one pivot point (ranging between 0 to length of the bits string) and exchange the substring from that bit point till the end of the string between the two individuals. In that case, two new offspring individual are generated.

Two-point Crossover: Randomly pick two pivot points and the bits in between the two points are swapped between the parent individuals. In that case, two new offspring individual are generated. mutation operators.

3.3 Local Search

We implement the idea of mutation in genetic algorithm using WalkSat local search method. It's a randomized local search algorithm. It attempts to determine the best move by randomly choosing a clause among those that are currently unsatisfied and selecting a variable to flip within it under some conditions. The number of tries (Maxtries) is set to

1000. The search stops when one solution is found.

3.4 Three Performance Measures

After all these processes to generate a solution, we compare our solver using different crossovers and with other existing solvers. Several runs are required for each benchmark instance under consideration and some measuring methods need to be taken for statistically meaningful and also relatively fair results. We take 10 runs and consider three measuring methods: SR (Success rate), AT (Average Time) and AFS (Average Flips to Solution). We will introduce them respectively below.

SR (Success rate): SR represents the percentage of runs where a solution has been found within a time limit. Since some runs where the time to get a solution is too long or the solver is stuck in the local optimal solution, we use SR to measure the quality of the solver. And the maximum time we set for a successful run is 120 seconds.

AT (Average Time): AT represents the average seconds taken in successful runs.

AFS (Average Flips to Solution): Relating the computational costs to the number of the basic moves in the search space for a solution has become the standard measure used for studying the cost of SAT algorithms[**hoos2005stochastic**]. AFS represents the average number of bit flips needed to find a solution in successful runs, which is by raised by Gottlieb and Voss (2000) used to compare EAs generating new solution candidates by single bit flips.

4 Experiments & Results

4.1 Experiments

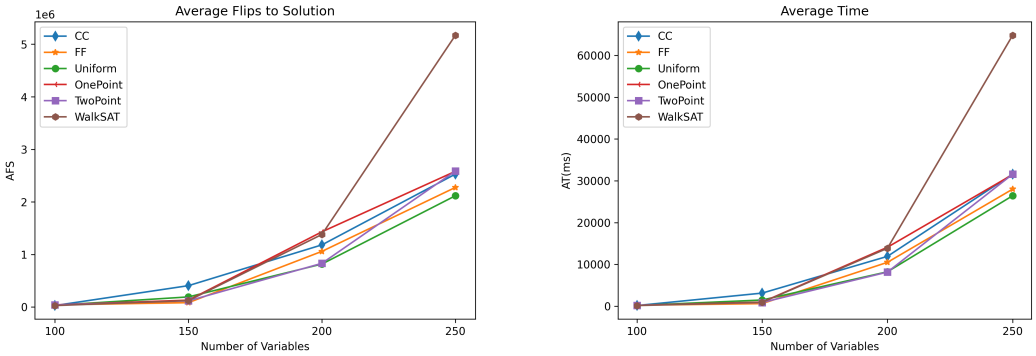
Here we performed our algorithms in different datasets with variant clause numbers to determine both its efficiency and effectiveness. We choose datasets with 100, 150, 200 and 250 variables, corresponding to 430, 645, 860 and 1075 clauses respectively. That is to say, the rates of a number of variables and clauses are fixed and equals to 4.3. The datasets we used are all generated randomly. Datasets with 100, 150 and 200 variables are downloaded from SATLIB, an online published by UBC[1]. Datasets with 250 variables are generated by ourselves. All of the datasets are satisfiable for 3-SAT problems. In order to reduce contingency, we run the genetic algorithm-based method and the WalkSAT method in each dataset 10 times, and then we computed the average time used and average flip times to measure the efficiency. In addition, we had set a two minutes timeout for the two algorithms, which means that failing to find an available solution in two minutes would seem a failure to solve the dataset. We would count the number of failures to determine the effectiveness of the algorithms. The detailed results could be found in Table.1.

In our experiments we tested five kinds of crossover methods for genetic-based algorithm, they were CC, FF, uniform, one-point and two-point crossover respectively. Besides we run WalkSAT on the same dataset as the contrast to find the difference between our algorithms and local search methods.

4.2 Results

According to Table.1, it is obvious that with the variables increasing, the AFS and AT of all the algorithms raised, but the SR decreased. For datasets with 100 and 150 variables, the SRs of all the algorithms were 100%, but algorithms began to fail solving problems when it came to 200 variables (for example, dataset UF-200-068). From Fig.2 it could be found that there were only slight differences between different methods while the number of variables was low. However, while the problems becoming more complex, the difference would show. For those datasets having 250 variables, the WalkSAT algorithm would take a long time to search the solution, and there would be a high possibility that it could not give a solution. Overall, the genetic-based algorithm using uniform crossover performed

[1] https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html



(a) AFS (b) AT

Figure 2: The performance on datasets with different numbers of variables. We computed the average AFS and AT on all the 5 datasets with the same number of variables and then plot them in the figures.

best on the complex 3-SAT problems, both in computation complexity and time consumption aspect. For other kinds of crossover methods, CC, one-point, and two-point crossover performed most the same, while FF had shown a little superiority over them.

5 Discussions

6 Conclusion

		Genetic Based-Algorithm															WalkSAT		
		CC			FF			Uniform			OP			TP					
		AFS	AT(ms)	SR	AFS	AT(ms)	SR	AFS	AT(ms)	SR	AFS	AT(ms)	SR	AFS	AT(ms)	SR	AFS	AT(ms)	SR
UF-100	0404	9177.4	53.4	1.0	9633.0	53.7	1.0	9529.4	55.4	1.0	14731.7	79.9	1.0	16413.6	88.2	1.0	7255.5	41.2	1.0
	0647	15930.1	87.8	1.0	24728.8	128.5	1.0	33237.3	177.2	1.0	25046.2	131.0	1.0	31038.8	161.2	1.0	28321.3	147.1	1.0
	0720	13826.8	76.9	1.0	27329.1	142.2	1.0	16999.4	93.6	1.0	19317.5	102.8	1.0	16049.8	86.1	1.0	11183.2	61.1	1.0
	0835	73864.9	386.2	1.0	120311.8	607.4	1.0	70826.9	371.2	1.0	135137.5	685.1	1.0	104195.1	528.3	1.0	73663.1	376.4	1.0
	0925	33489.4	178.5	1.0	28828.7	149.2	1.0	38156.2	202.4	1.0	19596.8	103.8	1.0	33367.7	172.9	1.0	24007.9	124.9	1.0
UF-150	016	21526.1	169.6	1.0	10625.7	82.3	1.0	20516.0	162.2	1.0	21600.0	165.5	1.0	22829.2	174.7	1.0	17521.0	136.4	1.0
	025	410056.8	3116.7	1.0	148867.1	1092.7	1.0	234302.9	1780.7	1.0	178405.0	1321.9	1.0	178955.2	1327.7	1.0	248282.3	1874.6	1.0
	027	1556158.2	11978.0	1.0	201949.8	1478.6	1.0	637603.0	4864.7	1.0	366749.4	2717.2	1.0	319983.2	2373.6	1.0	282071.3	2127.9	1.0
	089	34081.2	264.8	1.0	44668.5	328.2	1.0	57147.8	439.4	1.0	54248.0	405.8	1.0	36884.2	279.0	1.0	45971.1	350.7	1.0
	093	16817.7	134.0	1.0	14447.8	110.4	1.0	17689.0	140.8	1.0	37797.9	284.9	1.0	33640.0	254.2	1.0	24792.6	191.3	1.0
UF-200	018	371425.5	3706.8	1.0	314467.2	3002.8	1.0	383495.1	3840.5	1.0	270917.4	2641.1	1.0	505483.3	4927.6	1.0	279120.8	2803.0	1.0
	024	1337897.9	13435.2	1.0	907525.4	8847.3	1.0	1213634.6	12200.6	1.0	1362716.6	13389.9	1.0	975747.7	9542.9	1.0	2346452.8	23529.8	0.9
	051	1074383.1	10784.6	1.0	1365475.4	13591.9	1.0	935085.4	9385.3	1.0	1209651.5	11847.2	1.0	887628.7	8684.6	1.0	2396565.5	24034.6	1.0
	068	3074382.4	31095.2	0.5	2675314.1	26672.3	0.6	479945.1	14907.8	0.7	4232634.8	41823.3	0.6	1740682.9	17109.7	1.0	1802523.3	18111.5	0.6
	072	59105.7	593.8	1.0	37188.5	352.9	1.0	82337.7	827.5	1.0	83066.0	813.8	1.0	53628.1	527.8	1.0	82984.3	836.6	1.0
UF-250	004	886070.3	10958.0	1.0	820759.3	9799.9	1.0	844263.0	10432.6	1.0	994404.5	12008.9	1.0	682684.1	8227.3	1.0	4053239.0	50826.1	1.0
	005	1056567.4	13096.9	1.0	1046032.8	12579.3	1.0	1145335.5	14195.0	1.0	990259.0	11972.8	1.0	1382175.6	16753.1	1.0	5483059.7	68762.1	0.8
	008	4402336.0	55131.0	0.4	3598460.5	44488.0	0.4	3369114.6	42127.8	0.6	6358087.1	77834.5	0.6	7008261.7	85854.2	0.4	\	\	0
	009	3670606.3	45996.5	0.9	4414599.2	54914.8	1.0	4012810.7	50326.5	1.0	3446733.4	42100.3	0.9	2886818.8	35266.3	0.8	4588804.5	57555.5	0.2
	011	2633192.1	32936.5	1.0	1501120.3	18285.3	1.0	1216875.7	15107.6	1.0	1123454.1	13590.2	1.0	978018.1	11830.0	0.9	6556025.0	82199.6	0.3

Table 1: Experiments Results. The column 'AFS' represents the average flip times over ten times running, which represents the computation complexity of the algorithm. The column 'AT' represents the average time usage over ten times running, which represents the time complexity of the algorithm. The column 'SR' represents the success rate of the algorithm. '\' means that there were no available solution for the algorithm in the certain dataset. 'OP' means 'one-point crossover and 'TP' means 'two-point crossover' in the table.