ECE 595Z - Digital Systems Design Automation
Course Project
Boolean Satisfiability Solver
Project Report

Timur Ibrayev, Nikhil Sunil Chhabria

Introduction

The Boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm, introduced in 1962, still forms the basis for most efficient complete SAT solvers. In this project, we have developed a SAT solver in C++ from scratch, based on the DPLL algorithm. We have also implemented the randomization and geometric restarts (RGR) heuristic.

Motivation

SAT is often described as the "mother of all NP-complete problems". The past few years have seen an enormous progress in the performance of SAT solvers. Despite the worst-case exponential run time of all known algorithms, SAT solvers are increasingly leaving their mark as a general-purpose tool in areas as diverse as software and hardware verification, automatic test pattern generation, planning, scheduling, and even challenging problems from algebra. Given the maturity of this area, we knew from the get-go that we wouldn't get anywhere near the modern SAT solvers in terms of performance. Our main incentives for this project were to explore the complexity of the DPLL algorithm, and to implement a simple heuristic, in this case the RGR strategy.

Data Structures

I. *Primary Data Structures*

We use two primary data structures:

1. **vector_literals**: This is a two-dimensional vector (i.e. a vector of vectors) in which each variable is associated with a vector of clause numbers that contain that variable. A clause number is positive if the variable appears in uncomplemented form in that clause, and it is negative if the variable appears in complemented form in that clause. This data structure is used as lookup table for:
   i. *Finding pure literals*: A variable with only positive clause numbers would be an uncomplemented pure literal, whereas a variable with only negative clause numbers would be a complemented pure literal.
   ii. *Assigning a literal*: An uncomplemented or complemented literal is assigned using the vector of clause numbers associated with the corresponding variable.
   iii. *Backtracking chronologically on a literal*: This is essentially the reverse operation of literal assignment, where an assigned uncomplemented or complemented literal is 'undone' using the vector of clause numbers associated with the corresponding variable.
   iv. *Restarting*: When a certain number of conflicts is reached, all literals starting from the latest decision to the first free decision are 'undone'.

2. **vector_clauses**: This is a vector of clauses, wherein each clause has two properties:
   i. *satisfied_count*: The number of literals currently assigned satisfiably in the clause.
   ii. *unassigned_literals*: A vector of the currently unassigned literals in the clause.

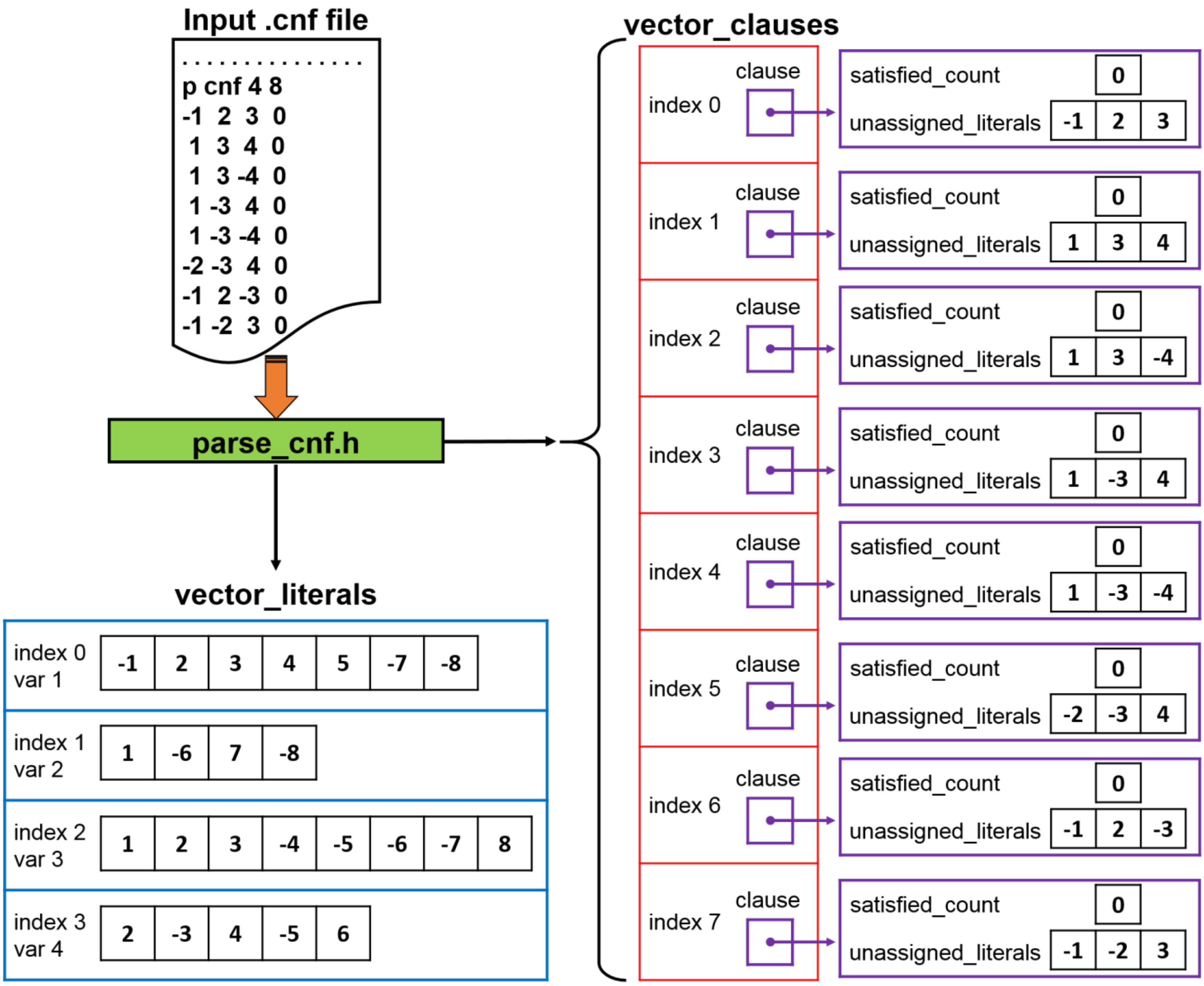

Figure 1. Primary data structures

II. Secondary Data Structures

We use six secondary data structures:

1. **pure_literals**: A vector of the pure literals found in the beginning.
2. **forced_literals**: A vector of the current forced literals.
3. **free_literals**: A vector of the current free literals.
4. **decision_literals**: A vector of the literals assigned so far.
5. **satisfied_clause_numbers**: A vector of the currently satisfied clause numbers.
6. **free_literal_positions**: A vector of the positions (indexes) of the free literals in decision_literals.

Core Algorithm

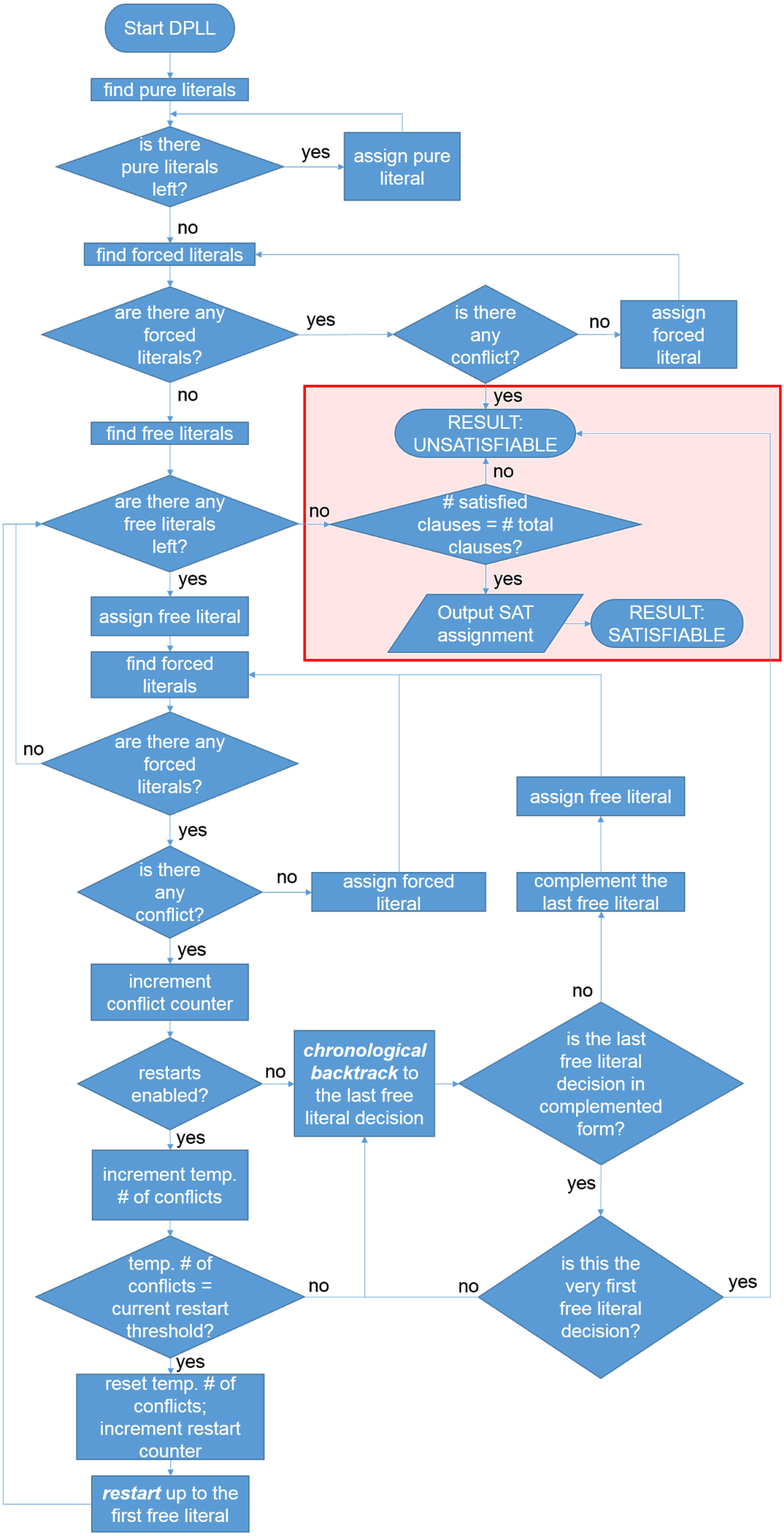

Figure 2. Flowchart of DPLL algorithm with RGR heuristic

Salient Features

I. *Conflict Detection*

To detect if there is a conflict, for each forced literal in the vector *forced_literals*, we check whether the complement of that forced literal is present in the vector. If yes, then there is a conflict.

II. *Chronological Backtracking*

Chronological backtracking is implemented as a recursive function, in which we backtrack to the last free literal in the vector *decision_literals* using the last element in the vector *free_literal_positions*. If this last free literal is in uncomplemented form, we complement it and assign it. If this last free literal is in complemented form, we backtrack further to the next last free decision.

III. *Randomization and Geometric Restarts (RGR) Heuristic*

We have implemented the RGR strategy proposed by Walsh i.e. the n^{th} restart is performed $k.a^{n-1}$ steps after the previous restart (where k = 100 and a = 1.5 by default, and steps refer to the number of conflicts). We have also allowed for user-defined values of k and a.

Experimental Methodology

To measure the performance of our implementation, the SAT solver was tested on two different types of benchmarks: Uniform Random 3-SAT and DIMACS, without and with RGR heuristic (using default values of k and a). Uniform Random 3-SAT benchmark contains 5 different pairs of sat/unsat sets with each set containing 3 .cnf files with 50, 75, 100, 125, and 150 variables. DIMACS benchmark is itself divided into two different benchmarks: AIM and PHOLE. AIM benchmark contains 2 pairs of sat/unsat sets with each set containing 3 .cnf files with 50 and 100 variables. PHOLE benchmark contains only 3 .cnf files modeling unsatisfiable functions with a varied number of variables.

Results

Table 1. Uniform Random 3-SAT benchmark performance

Set name	Number of variables | clauses	Performance parameters	01.cnf w/o | w restarts		02.cnf w/o | w restarts		03.cnf w/o | w restarts	
sat_50_218	50 | 218	Computation time (s)	0.21	0.19	0.37	0.18	0	0
		Number of conflicts	1015	931	1644	723	17	17
		Number of restarts	-	4	-	3	-	0
sat_75_325	75 | 325	Computation time (s)	0.01	0.01	4.87	0.17	9.2	4.41
		Number of conflicts	36	36	14656	501	28246	13230
		Number of restarts	-	0	-	3	-	10
sat_100_430	100 | 430	Computation time (s)	374.08	315.32	309.43	998.2	185.82	538.85
		Number of conflicts	562376	494171	450954	1489573	262848	771589
		Number of restarts	-	19	-	21	-	20
sat_125_538	125 | 538	Computation time (s)	7847	1451.6	5649.02	18292.8	1322.2	5753.18
		Number of conflicts	11464951	2313623	9148862	29454983	1999851	8792786
		Number of restarts	-	23	-	29	-	26
sat_150_645	150 | 645	Computation time (s)	***	44847.5	5171.02	11787.2	***	***
		Number of conflicts	***	51083649	5755422	13694584	***	***
		Number of restarts	-	30	-	27	-	***
unsat_50_218	50 | 218	Computation time (s)	0.19	X	0.2	X	0.27	X
		Number of conflicts	1030	X	1010	X	1257	X
		Number of restarts	-	X	-	X	-	X
unsat_75_325	75 | 325	Computation time (s)	14.78	X	16.18	X	9.67	X
		Number of conflicts	42956	X	47929	X	28152	X
		Number of restarts	-	X	-	X	-	X
unsat_100_430	100 | 430	Computation time (s)	1295.88	X	360.95	X	669.3	X
		Number of conflicts	2673316	X	744046	X	1390236	X

		Number of restarts	-	X	-	X	-	X
unsat_125_538	125 | 538	Computation time (s)	42322.3	X	32440.5	X	***	X
		Number of conflicts	67653165	X	48839468	X	***	X
		Number of restarts	-	X	-	X	-	X
unsat_150_645	150 | 645	Computation time (s)	***	X	***	X	***	X
		Number of conflicts	***	X	***	X	***	X
		Number of restarts	-	X	-	X	-	X

*** – Did not complete in time for submission

X – Not performed

Table 2. DIMACS benchmarks performance

Set name	Number of variables | clauses	Performance parameters	01.cnf w/o | w restarts		02.cnf w/o | w restarts		03.cnf w/o | w restarts	
AIM								
sat_50_300	50 | 300	Computation time (s)	0	0	0	0	0	0
		Number of conflicts	17	17	22	22	15	15
		Number of restarts	-	0	-	0	-	0
sat_100_340	100 | 340	Computation time (s)	85.16	80.34	45.73	37.88	91.87	34.93
		Number of conflicts	268766	331539	140956	156963	296317	151172
		Number of restarts	-	18	-	16	-	16
unsat_50_80	50 | 80	Computation time (s)	1215.68	X	250.03	X	212.61	X
		Number of conflicts	49637911	X	9679218	X	7851459	X
		Number of restarts	-	X	-	X	-	X
unsat_100_160	100 | 160	Computation time (s)	***	X	***	X	***	X
		Number of conflicts	***	X	***	X	***	X
		Number of restarts	-	X	-	X	-	X
PHOLE								
01.cnf 02.cnf 03.cnf	42 | 133 56 | 204 72 | 297	Computation time (s)	0.36	X	6.02	X	127.49	X
		Number of conflicts	3307	X	34204	X	469981	X
		Number of restarts	-	X	-	X	-	X

*** – Did not complete in time for submission

X – Not performed

From the above results, we observe the following:

- The computation time increases exponentially with increase in the number of variables.
- The computation time of UNSAT instances is greater than that of SAT instances.
- Due to the random nature of our implementation, there are cases when RGR heuristic (default values of *k* and *a*) reduces the computation time of a SAT instance and cases when it increases the computation time.

RGR heuristic with optimum user-defined *k* and *a* values can greatly reduce computation time of a SAT instance. For example,

- 01.cnf of sat_100_430 took 10.85 seconds with $k = 50$ and $a = 1.5$ (in contrast to 315.32 seconds with default values).
- 02.cnf of sat_100_430 took 7.73 seconds with $k = 200$ and $a = 1.5$ (in contrast to 998.2 seconds with default values).
- 03.cnf of sat_100_430 took 56.71 seconds with $k = 200$ and $a = 1.5$ (in contrast to 538.85 seconds with default values).
- 01.cnf of sat_150_645 took 270.85 seconds with $k = 50$ and $a = 1.5$ (in contrast to 44847.5 seconds with default values).

Conclusion

From this project, we understood the complexity of the DPLL algorithm, and how this complexity can at times be countered with an efficient restart heuristic. We believe that we would have seen significant improvements if we had paired the RGR heuristic with conflict-driven clause learning. But this could not be implemented due to limited time.