Study and Implementation of MINISAT: "An Extensible SAT Solver"

Debashri Roy
University of Central Florida, USA
debashri@cs.ucf.edu

***Abstract*—Efficient implementation of Satisfiability solver has gained more and more relevance with huge advancement of Electronic Design Automation and Artificial Intelligence fields. This project report is targeted to achieve a compact and clear overview of satisfiability problem, its applicability and studying of various implementations. Satisfiability problem in computer science is known to be the first NP complete problem recognized by Cook. Various kind of heuristic based implementation has been proposed and encouraged throughout the years. MINISAT is an established SAT solver used in number of fields EDA and AI. Purpose of this project was to study and implement MINISAT solver and compare it with original version. The reported implementation is in Java and producing results SAT 2006 Competition benchmarks(upto 224K variables).**

I. Introduction

Satisfiability problem is the problem for determining an interpretation, if there exists one, that satisfies a given Boolean formula. It is one of the most challenging problem in computer science. Brute-force approach to solve this is of exponential time complexity. It is O(2^n) for n number of variables for generating all combinations. As it is an NP-Complete problem, researchers can only solve satisfiability problem with intelligent heuristics. In Table I, it is shown that boolean expression $(a+c')(a'+b+c)$ is satisfiable for four combinations(marked green) of variable assignment and unsatisfiable for other four combinations(marked red) of variable assignment, making the expression actually SATISFIABLE. So finding such an one and only one combination of variable assignment is sufficient to prove satisfiability of any boolean expression, where the reverse is not so easy. Here comes the complexity of solving this problem.

a	b	c	$(a+c')$	$(a'+b+c)$	SAT/UNSAT
F	F	F	(F+T)	(T+F+F)	SAT
F	F	T	(F+F)	(T+F+T)	UNSAT
F	T	F	(F+T)	(T+T+F)	SAT
F	T	T	(F+F)	(T+T+T)	SAT
T	F	F	(T+T)	(F+F+F)	UNSAT
T	F	T	(T+F)	(F+F+T)	UNSAT
T	T	F	(T+T)	(F+T+F)	UNSAT
T	T	T	(T+F)	(F+T+T)	SAT

TABLE I
SATISFIABILITY OF $(a+c')(a'+b+c)$

II. Background and Motivation

The project of building a new satisfiability solver is actually intended to build a robust and user friendly efficient SAT solver in style of conflict-driven learning, as demonstrated in [1], and [2]. Though there are certain state-of-art tool [3] for solving SAT problem in Java, but still there exists technological challenges to get better running time and quality output(like getting result for more number of variables).

A. Java based State-of-the-art:

Some existing Java based implementation of SAT solvers are OpenSAT [4], SAT4J [3] [3], JQUEST2 [5], JSAT [6] etc. The key-point of any object-oriented design is that everything needs to be an object(like here literals, clauses, formula). Object oriented approach exhibits the user-friendliness and cleanliness of code for the designers. OpenSAT was implemented based on the experience of JQUEST2 [5] and JSAT [6] of developers. JQUEST2 [5](object-oriented approach) was less than a factor of 2 slower than Chaff [2](C implementation). So it can be said that Java SAT solver is competitive with state-of-the-art C/C++ SAT solvers. Java based SAT solver like OpenSAT [4] is easy to use or embed in SAT-powered Java application(where original or modified version of SAT solver is needed) for both the users and developers.

III. MINISAT: An Extensible SAT Solver

Three main components for any SAT solver is (a) Representation; (b) Inference and (c) Search. MINISAT is a small, complete, and efficient conflict-driven SAT solver inspired from CHAFF [2] and [7]. Properties of those three components for MINISAT are described bellow:

- Representation: MINISAT must be able to store different constraints.
- Inference: In this step unit informations are derived from those constraints.
- Search: It should be capable of analyzing and generating conflict clauses from the constraints.

A quick overview of total mechanism of MINISAT presented in Figure. 1. Total implementation of MINISAT has been deployed in three distinct part. Those are:

- **Propagation:** This portion is inspired by CHAFF [2]. MINISAT had implemented Boolean Constraint Propagation(BCP) for propagation with unit clauses.
- **Learning:** The procedure for learning the clauses was inspired by ideas of Marques-Silva and Sakallah in [7].
- **Search:** Searching is implicit here. Decision is based on back-jumping or non-chronological backtracking of [7].

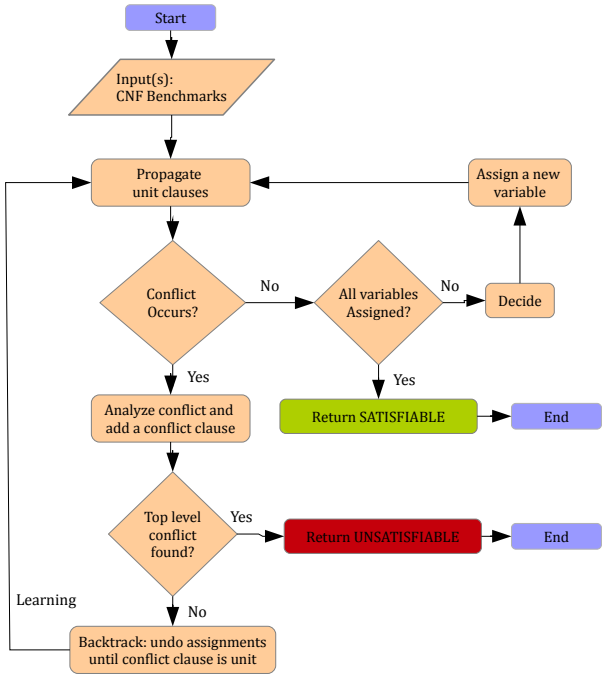

Fig. 1. MINISAT: In a Glipmse

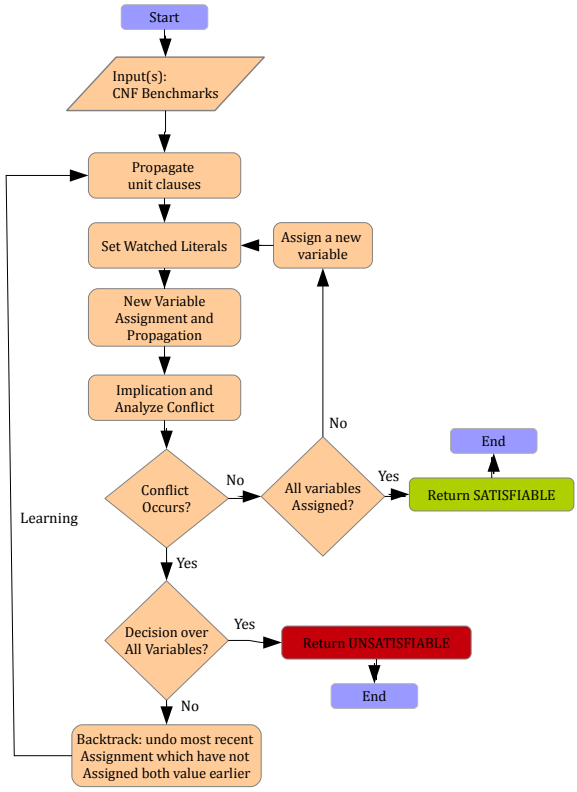

Fig. 2. Flowchart of Adapted Approach: Based on Boolean Constraint Propagation

IV. Adapted Approach

The overall strategy to implement SAT solver is presented in Algorithm 1. In the algorithm, unit propagation is first applied over only those benchmarks having unit clauses. Then it goes on applying BCP from CHAFF [2] staring with watched literal setting. Implemented algorithm ends with double checking the variable assignment for satisfiable benches. A glimpse of overall implemented approach is presented in Figure 2 for better understandability.

SAT_Solver(CNF benchmark)

```
begin
    unit_propagate(); /* Propagate Unit Clauses */
    set_watched_lit(); /* Set Watched Literals for each Clauses */
    decide(); /* Pick a New Variable and Assign it */
    analyze(); /*Derive Implications and Analyze Conflict */
    if not conflict then
        if all variable assigned then
            if check-satisfiablity() then
                return SATISFIABLE; /* If variable assignments are really
                satisfying the whole circuit */
            end
        else
            decide(); /* Pick a New Variable and Assign it and reset
            watched literals */
        end
    else
        if all-variable-checked then
            return UNSATISFIABLE;
        else
            backtrack(); /*Undo Most Recent Assignment which have not
            Assigned Both Value Earlier */
        end
    end
end
```

Algorithm 1: Algorithm for Implementing SAT Solver

V. Results and Discussions

A. Experimental Framework

The SAT solver is implemented in Java and Swing. The experiments were performed on a standard desktop environment of 4 GB memory with an Intel chip running at 2.30 GHz. The implementation is tested on two sets of benchmarks, i.e. sample benchmarks [8] and SAT 2006 competition benchmarks [9].

B. Comparison for Test Benches

Results with respect to CPU time vs number of variables and number of clauses, for Sample benchmarks [8], is presented in Table V-B. The implemented project is named as MySAT here. It is clear from result that MySAT is taking much longer time than MINISAT one, though both are capable of identifying the satisfiability. It is also evident from Figure V-B and V-B that CPU time increases in similar manner with increasing number of variables and clauses. Ultimately, MySAT is able to produce correct result and satisfiable variable assignment(for satisfiable circuits only) within a second upto roughly 1040 variable, which is quite encouraging for a Java based application of SAT Solver.

C. Comparison for SAT 2006 Benchmarks

Next MySAT is compared with state-of-the-art IBM benchmarks [9] of SAT 2006 Completion. The comparison study, presented in Table V-C, has been done with respect to CPU

Instances	# Variables	SAT/UNSAT	MINISAT2.0 [10](Sec)	MySAT(Sec)
ibm-2002-07r-k100.cnf	24767	UNSAT	6.08	166.06
ibm-2002-11r1-k45.cnf	156626	SAT	73.48	16174.11
ibm-2002-26r-k45.cnf	224477	UNSAT	42.64	74502.63
ibm-2004-03-k70.cnf	69839	SAT	22.33	4880.65
ibm-2004-04-k100.cnf	120445	SAT	124.65	18126.84
ibm-2004-06-k90.cnf	112376	SAT	29.07	11380.90
ibm-2004-1_11-k25.cnf	78503	UNSAT	4.95	3901.62
ibm-2004-1_31_2-k25.cnf	31125	UNSAT	79.20	858.12
ibm-2004-26-k25.cnf	125791	UNSAT	55.20	22015.90
ibm-2004-2_02_1-k100.cnf	68848	UNSAT	10.57	2576.61
ibm-2004-2_14-k45.cnf	65032	UNSAT	16.86	2988.83
ibm-2004-3_02_1-k95.cnf	61780	UNSAT	7.85	1949.31
ibm-2004-3_02_3-k95.cnf	73525	SAT	13.55	3111.56
ibm-2004-6_02_3-k100.cnf	80370	UNSAT	5.53	4047.47

TABLE III
COMPARISON OF CPU TIME FOR 3-1040 VARIABLES

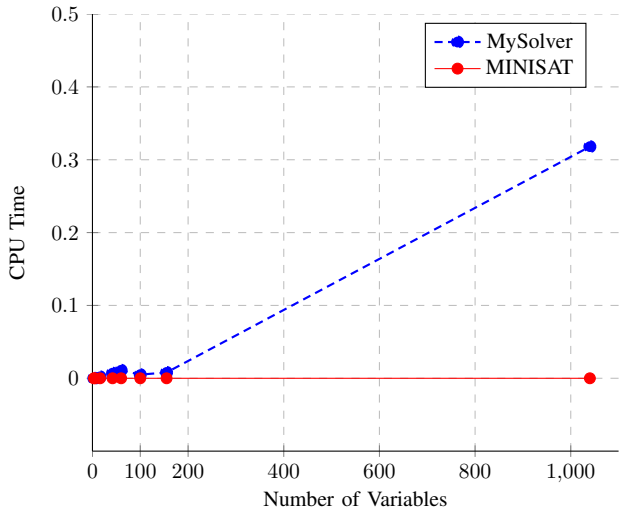

Fig. 3. Comparison with Test Benchmarks: No of Variable vs CPU Time

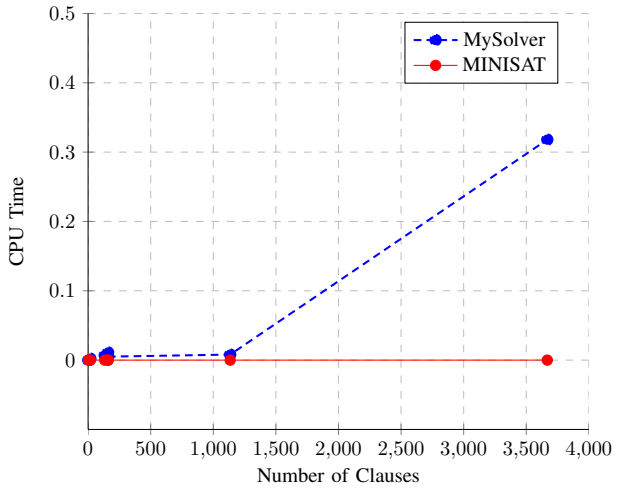

Fig. 4. Comparison with Test Benchmarks: No of Clauses vs CPU Time

Instances	# Variables	# Clauses	SAT/UNSAT	MINISAT2.0(Sec)	MySAT(Sec)
simple_v3_c2.cnf	3	2	SAT	0.0	0.0
sample_v5_c5.cnf	5	5	SAT	0.0	0.0
hole6.cnf	42	133	UNSAT	0.0	0.001
dubois20.cnf	60	160	UNSAT	0.0	0.004
dubois21.cnf	60	160	UNSAT	0.0	0.004
dubois22.cnf	60	160	UNSAT	0.0	0.004
aim-100-1_6-no-1.cnf	100	160	UNSAT	0.0	0.005
bf0432-007.cnf	1040	3668	UNSAT	0.03	0.318

TABLE II
COMPARISON OF CPU TIME FOR 3-1040 VARIABLES

time on total number of variables in any benchmark circuits. Two important points to consider are: first it is capable of producing results roughly upto 0.22 million of variables, and second it's CPU time differs more and more with increasing number of variables. The results of Table V-C is plotted in Figure V-C.

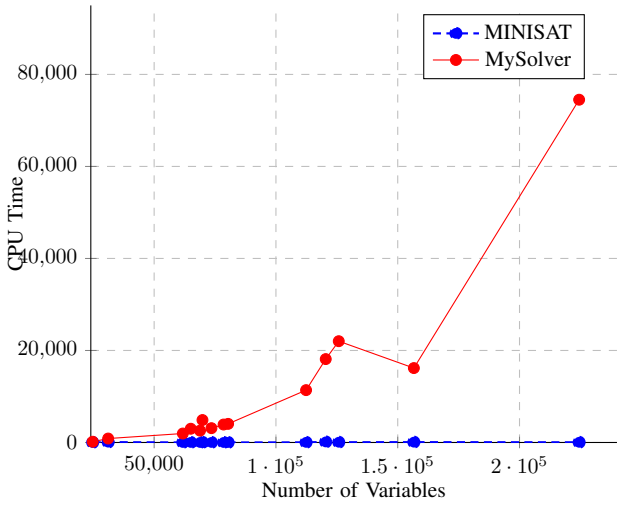

Fig. 5. Comparison with State-of-the-art: No of Variable vs CPU Time

VI. Novel Contributions

This implementation is based on the concepts of already established SAT solvers [1] and [2]. There is no explicit novelty in it. But the implementation is dared to deploy in JAVA, remembering huge overhead of JVM. Presented running CPU time includes that timing overhead in anyway. Algorithm mainly follows the method of boolean constraint propagation(BCP) from CHAFF [2]. Though in course of implementation the unit propagation method is called distinctly for those having unit clauses. Others follow BCP as usual. This implementation always terminates, it either assigns TRUE or FALSE values to all variables and return SATISFIABLE as a result, or it cannot assign few variables and return UNSATISFIABLE. As last one satisfiability checking has been done, to get assured for SATISFIABLE outcomes.This increases reliability of overall implementation for SATISFIABLE circuits.

VII. Conclusion

In a nutshell this project was intended to create an insightful overview of SAT problem, its relevance and its different kind of solution approach. In my report and implementation, I have been able to deploy a JAVA based robust tool which is able to give result within a second upto 1000 variables and are producing results for state-of-the-art benchmarks. Implementing the total MINISAT mechanism or exploiting Java threads for faster implementation of it, can be considered as future scope.

References

[1] N. En and N. Srensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518, Springer Berlin Heidelberg, 2004.

[2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, (New York, NY, USA), pp. 530–535, ACM, 2001.

[3] "The boolean satisfaction and optimization library in java." http://www.sat4j.org/index.php.

[4] G. Audemard, D. L. Berre, O. Roussel, I. Lynce, and J. Marques-Silva, "Opensat: An open source sat software project," 2003.

[5] I. Lynce and J. Marques-Silva, "On implementing more efficient data structures," in *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[6] "Java based sat solver." http://cafe.newcastle.edu.au/daniel/JSAT.

[7] J. a. P. M. Silva and K. A. Sakallah, "Grasp: a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, (Washington, DC, USA), pp. 220–227, IEEE Computer Society, 1996.

[8] "Cnf files." http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html.

[9] "Sat-race 2006 benchmarks." http://fmv.jku.at/sat-race-2006/downloads.html.

[10] "Sat-race 2006 result." http://fmv.jku.at/sat-race-2006/results.html.