A Complete SAT Solver for Satisfiability problem

Ashis Kumar Dash

Abstract- SAT solver plays an important role in cryptography, computer design, VLSI design. SAT is a NP-complete problem. In this paper a greedy algorithm is designed to find a complete SAT solver. There are strong incomplete SAT solvers; still complete SAT solvers have their own importance. Sometimes we need all the satisfiable instances for SAT problems. This algorithm describes how to get all the satisfiable instances of a SAT problem. This algorithm starts with all satisfiable instances of a clause present in a CNF Boolean function. Then these instances are improved to satisfy all the clauses present in the Boolean function.

Key Words: SAT, Boolean function, Complete SAT solver, Literals Clauses, CNF, NP-complete, Satisfiability, Satisfiable instance, Boolean function.

1. INTRODUCTION

Complexity of SAT solvers inform about the strength of security systems. Harder the SAT solver, stronger the security system. So SAT solvers are tools to design the strong security systems. It is also helpful for VLSI design. Defectives in VLSI design can be detected by SAT solvers. In this paper a greedy algorithm is discussed for complete Sat solver. SAT is NP-complete problem. Complete SAT solvers provide all the satisfiable instances for a SAT problem. In case of Incomplete SAT problem, either one satisfiable instance is obtained or nothing can be said about existence of an instance. Comparison to incomplete SAT solver, complete SAT solver has many alternatives for design issue.

2. BOOLEAN FUNCTION IN CNF

A Boolean Formula(BF) F is a logical expression defined over variables that takes the value in the set {True, False} which we will identify with {0, 1}. A truth assignment to set V of Boolean variables is a map σ: V → {0, 1}. A satisfying assignment for F is a truth assignment σ such that F evaluates to 1 under σ. A Boolean formula has two special form, conjunctive normal form (CNF) and disjunctive normal form (DNF). In this area we consider only CNF of BF. BF is in CNF if it is conjunction (∧) of clauses, where each clause is disjunction (∨) of literals .Each literal is either a variable or its negation. For example F= (x ∨ y) ∧ (x ∨¬y ∨¬z) ∧ z is in CNF with three variables and three clauses. x, y, z,¬y,¬z are literals and x ∨ y, x ∨¬y ∨¬z and ¬z are clauses.

3. PROBLEM DEFINITION

The SAT problem is shorthand for Boolean satisfiability problem. SAT problem refers to the question that given a Boolean expression, determine if there exists an assignment of TRUE (1) or FALSE(0) to all Boolean variables that make the entire Boolean expression to be TRUE. There is another equally important question that there exists no such assignment. Both of them are NP-complete problem [3]. In the first case one assignment is sufficient if such assignment exists. Then we call the Boolean expression is satisfiable, otherwise we call it unsatisfiable which is proven in the second case which needs exhaustive search of all possible assignments. According to the rules of logical equivalence, each Boolean expression can be transformed into CNF form which sometimes simplifies the problem to some extent for exposing the underlying structure of the SAT problem, so that a couple of optimization strategies can be applied to reduce the size of the original problem. In addition, Boolean expressions in CNF can be easily treated as input for SAT solvers. In this paper, SAT problem inputs to the solver are assumed to be in general CNF form.

4. EARLIER WORKS

The *satisfiability* problem can be solved deterministically in time $poly(n).2^n$ time, where n is the number of literals and *poly(n)* is a polynomial in n. This worst-case upper bound can be decreased to $poly(n).c^n$, $c \leq 2$, if we restrict to k-SAT problems, where each clause in the Boolean CNF expression contains at most k literals. In [3], it has been shown that the 2-SAT problem can be solved in polynomial time using a randomized local search procedure. Local search is a well-known heuristic that is applied widely to solve the SAT problem. The best known bound for randomized 3- SAT problem is $poly(n).(4/3)^n$ [4]. Random k-SAT problems exhibit a so-called "Phase Transition Phenomenon" [6], when there are exactly k literals

in each clause, which randomly choose the number of clauses c_k and the number of variables v_k, the probability of the satisfiability of the problem falls sharply from near 1 to near 0 as the ratio $r_k=c_k/v_k$ passes some critical point called threshold. For example, when k=3, the threshold value is about 4.25 (Figure 4.1). However, it is much more complicated to find the threshold value once k is larger than 3. In Figure 4.1, when r_k is close to Y axis, the problem can be easily proved to be satisfiable. Conversely, the problem can be easily proved to be unsatisfiable when it is far from Y axis. The hardest instances appear at the region near the peak (when $r_k \approx 4.25$). In this region, enormous search space needs to be traversed until the solution is found.

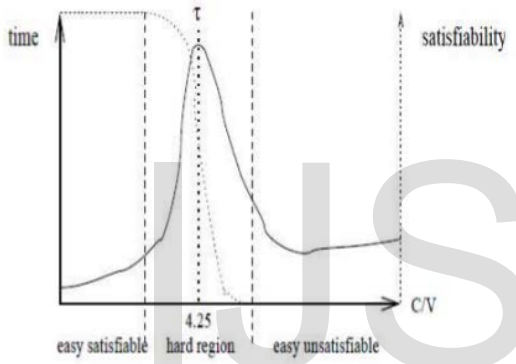

Figure. 4.1: SAT Problem Phase Transition Phenomenon [5]

Random walk strategy [5] for SAT problem is one of the most efficient methods to search the solution for SAT problems by making use of the heuristic variable selection. It evolved from a pure (unbiased) random walk selection strategy. In this strategy, performance of sequential execution is significantly improved. But it is not suitable for parallel environment.

5. SAT SOLVER

Among all of the SAT solvers, two main categories of SAT solvers are widely studied by researchers. One is Complete SAT solver and another one is incomplete SAT solver.

5.1 Complete SAT Solver

Complete SAT solver is the algorithm that checks the satisfiability of the SAT problems. It guarantees to give the result of whether a SAT problem is satisfiable or unsatisfiable. Most of modern complete SAT solvers are based on the classical DPLL algorithm [6]. DPLL itself is still a highly-efficient procedure for SAT problems even under contemporary performance standards. The fundamental principles of DPLL algorithm are backtracking and divide-and-conquer. It firstly simplifies the problem by assigning some values to some variables, so if the rest smaller problem is satisfiable, then the entire formula is satisfiable, otherwise it goes back to assign the opposite values to the appropriate previously assigned variables, and keep doing this recursively until a solution is found or the entire search space is traversed. DPLL actively calls two subroutines *Unit Propagation* and *Pure Literal Elimination* to enhance the efficiency of the algorithm, and recent researchers are also eagerly looking for efficient approaches to improve these two functions.

5.2 Unit Propagation

Some clauses only contain one literal, there is only one choice for the value of the corresponding variable, and then these variables can be safely eliminated from the problem without affecting the search of the values of other variables. In addition, these eliminations may lead to the deterministic cascades of unit clause which is able to dramatically reduce the size of the original problem to avoid naive search or early detection of assignment conflict which is able to prove the unsatisfiability of the problem.

5.3 Pure Literal Elimination

If one variable occurs in the problem with only one form (positive or negative), then all of the clauses that contain this variable can be eliminated from the problem since the Boolean value that makes the corresponding literal true can make all of those clauses be true, and there is only one choice for this value. While *Pure Literal Elimination* is not used in DPLL as intensively as *Unit Propagation*, because finding all of the clauses containing single form of variable is a computation intensive process, and sometimes, it is not worthwhile.

6. ALGORITHM OF COMPLETE SAT SOLVER

In this discussion a greedy strategy is used in order to verify the satisfiability of SAT problem. At the initial stage a clause generates a set of possible instances that satisfy the 1st clause. In the

successive stages each instance considers the next clause in order to generate instances that satisfy present clause as well as all the previous clauses. After the last clause verification all possible satisfiable instances can be reported or problem is unsatisfiable. Details of the strategy are given below. Suppose the Boolean expression in CNF consists of m number of clauses in n variables. There will be 2n literals. Suppose number of literals in the clauses is not equal. Let n_1, n_2, ..., n_m are number of literals present in clauses C_1, C_2, ..., C_m respectively. Denote a negative variable by 0, a positive variable by 1. From C_1, n_1 distinct instances can be formed where each instance contains exactly one literal (0/1) and rest n-1 places may have any value 0/1 (*don't cares*) such that these instances satisfy the clause C_1. For generating instances leave those n-1 places blank. Now each instance generated in the first stage will generate n_2 instances by assigning some literals in *don't care* positions that satisfy clause C_2. Conflict instances are to be eliminated. So, in the second stage there will be at most $n_1 * n_2$ number of instances. Proceeding in this way, at the end the method will generate at most $n_1*n_2*...*n_m$ number of instances. But actual number is less than this due to conflict and repetition. The following example with two clauses in four variables illustrates the method. Let the clauses be $C_1 : x_1 \vee x_3 \vee \neg x_4$ and $C_2: x_2 \vee \neg x_3 \vee \neg x_4$ respectively. The first clause generates following three satisfiable instances where blank space may take any value 0 or 1.

1 _ _ _, _ _ 1 _, _ _ _ 0.

Now each instance generated will verify second clause to generate new instances those satisfy both the clauses. First instance generates three, second two as there is one conflict (x_3, $\neg x_3$) and third one three (two new & one remains same) instances.

1 1 _ _, 1 _ 0 _, 1 _ _ 0.
_ 1 1 _ , _ _ 1 0.
_ 1 _ 0 , _ _ 0 0 , _ _ _ 0.

The blank spaces (hyphenated) may take any value 0 or 1. The eight possibilities so generated, in which each of first seven instances generates four and last one generates eight instances, will satisfy both the clauses. We cannot have more than these possibilities that satisfy both the clauses.

7. LIMITATION

This algorithm is a sequential algorithm. So complexity is $O(m^n)$. If this algorithm is parallelized using CUDA architecture or some multi-processor architecture it will reduce the time complexity to O(m). Here m is a small number.

8. CONCLUSION

If each stage generates the instances in parallel, then we need `m' stages for a CNF Boolean expression consisting of m clauses. In stage-m we get either all possible satisfiable instances (after assigning all possible values to blank spaces left) or conclude that the Boolean expression is unsatisfiable. At each stage the computation can be done in parallel but the stages are to be executed sequentially. It is a complete solver and produces all assignments of variables which make the function true. However, the scope of parallelism is limited.

REFERENCES

[1] Cook, Stephen "The complexity of theorem proving procedures" Proceedings of the Third Annual ACM Symposium on Theory of Computing. Pages: 151-158, Year 1971.

[2] Papadimitriou, C.H, Computational Complexity. 1994. Addison–Wesley.

[3] H. Papadimitrious. "On selecting a satisfying truth assignment". In the proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS'91, pages 163-169, 1991.

[4] Christos Papadimitrious, Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch,Ravi Kannan, Jon Kleinberg "A deterministic Algorithm for k-SAT based on local search".

[5] Wei and Bart Selman "Accelerating Random Walks," In Principles and Practice of Constraint Programming, pages: 61-67, Year 2002 .

[6] D.Singer."Parallel resolution of the satisfiability problem: a survey." In E.Talbi,editor. Parallel Combinatorial Optimization. John Wiley and Sons, pages: 123-147, Year 2006.