An Introduction to SAT Solvers

Knowles Atchison, Jr.
Fall 2012
Johns Hopkins University
Computational Complexity
Research Paper
December 11, 2012

Abstract

As the first known example of an NP Complete problem, the Boolean Satisfiability Problem (SAT) has no known algorithms that efficiently solve all of its instances. Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation and Artificial Intelligence [Moskewicz et al. 2001]. There are SAT solvers that can solve certain subsets of the problem domain efficiently enough for application in the real world. This paper will serve as a technical introduction to the world of SAT solving and the various algorithms that comprise the base of modern research.

1 Introduction

In complexity theory, SAT is a decision problem whose instance is a Boolean expression written in conjunctive normal form:

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$$

Is the equation given satisfiable? Given a series of truth values for all variables, does it equate to true?

x_1	x_2	$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$
0	0	1
0	1	1
1	0	1
1	1	1

In this trivial example, the equation has multiple satisfying truth assignments. For the decision problem, only one need exist for it to be satisfiable.

In his 1971 paper, “The Complexity of Theorem Proving Procedures”, Stephen Cook proved SAT to be NP-Complete. Every problem in NP can be reduced to SAT in polynomial time by a deterministic Turing machine [Cook 1971]. Thus, if there exists a deterministic polynomial time algorithm to solve SAT, then there is one to solve all problems in NP. Richard Karp followed up on Cook’s work with a list of 21 problems that are NP-Complete due to reduction from SAT.

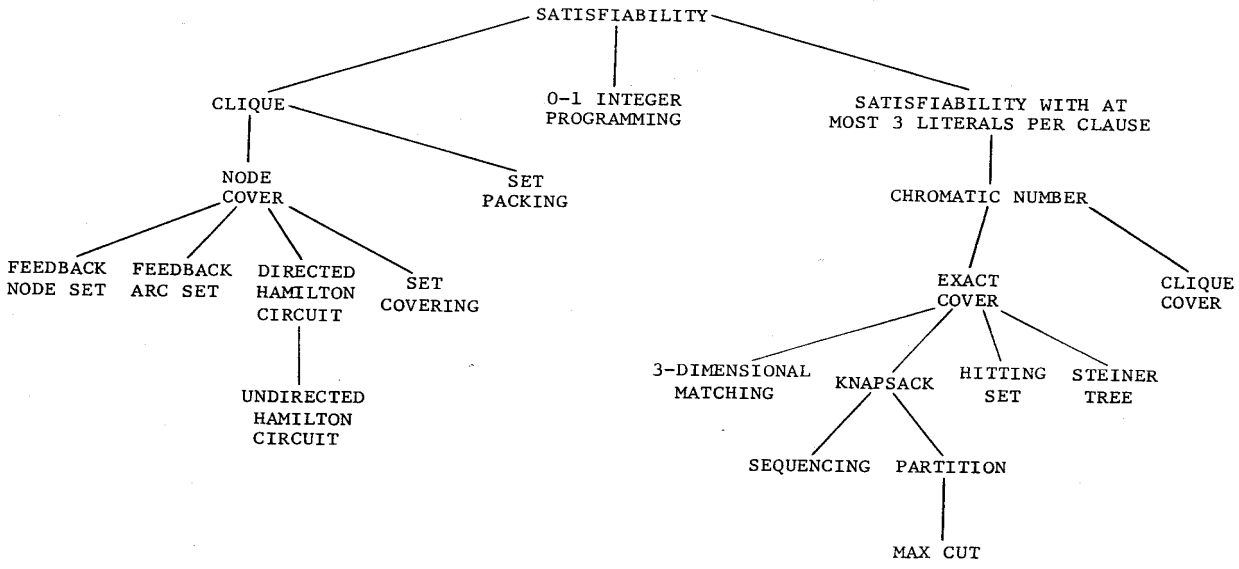

Figure 1: Complete Problems [Karp 1972]

The significance of the previous sentence cannot be understated. If the SAT domino falls, then the repercussions for complexity theory, not to mention areas like cryptography, will be wide spread and impacting.

1.1 Contributions

The goal of this paper is to be a technical introduction to the various implementations of SAT solvers in practice today. The underlying algorithms will also be discussed. Lastly, some future areas of research will be touched upon.

2 SAT Solver Categories

The current state of the art in SAT Solvers divides roughly into two categories: complete and incomplete.

Complete: Always finds an assignment if the equation is satisfiable
Incomplete: It may not find an assignment if the equation is satisfiable [rjlipton 2009]

2.1 Complete

Complete solvers use the Davis–Putnam–Logemann–Loveland (DPLL) algorithm as a base and then throw additional functionality on top such as advanced forward reasoning, efficient low level data structures, clause learning, watched literals, optimized conflict analysis, non-chronological backtracking, and decision heuristics [Oe et al. 2012].

Algorithm 1 DPLL

Input : *CNF formula* ϕ
Output : *Satisfiable* ?(*yes* / *no*)
– –
DPLL(ϕ)
  if $\phi ==$ set of consistent literals(satisfiable):
    return true
  if ϕ contains an empty clause/is inconsistent:
    return false
  $\phi =$ unit-propagate(ϕ)
  $\phi =$ pure-literal(ϕ)
  $l =$ choose-literal(ϕ)
  return $DPLL(\phi \wedge l)$ *or* $DPLL(\phi \wedge \neg l)$
– –
unit-propagate(ϕ)
  if ϕ contains a clause with a single literal l :
    remove clauses with l
    delete $\neg l$ from clauses that contain it

pure-literal(ϕ)
  if some literal l has only one polarity($l \mid \neg l$)
    assign l to true
    assign $\neg l$ to false

The basic backtracking algorithm runs by choosing a literal, giving it a truth assignment, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If the new formula is satisfiable, the original was as well. If not, the opposite truth value is attempted.

If the formula can be simplified by means of unit propagation or pure literals, the formula is modified and the cycle begins again. Note that some implementations will return true on a partial satisfying result. In this case, not every literal's value is found as long as the formula is still satisfiable given a subset assignment, whereas unsatisfiability of the complete formula can only be found after an exhaustive search.

A few examples:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

Initial formula has no single variable literals, unit propagation cannot be done. x_1 is a pure literal however and thus every instance of it can be replaced with true. The formula is then (true and true) and the partial satisfying assignment returns true.

$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ (x1 V x2 V ~x3) ^ (~x1 V ~x2 V x3)

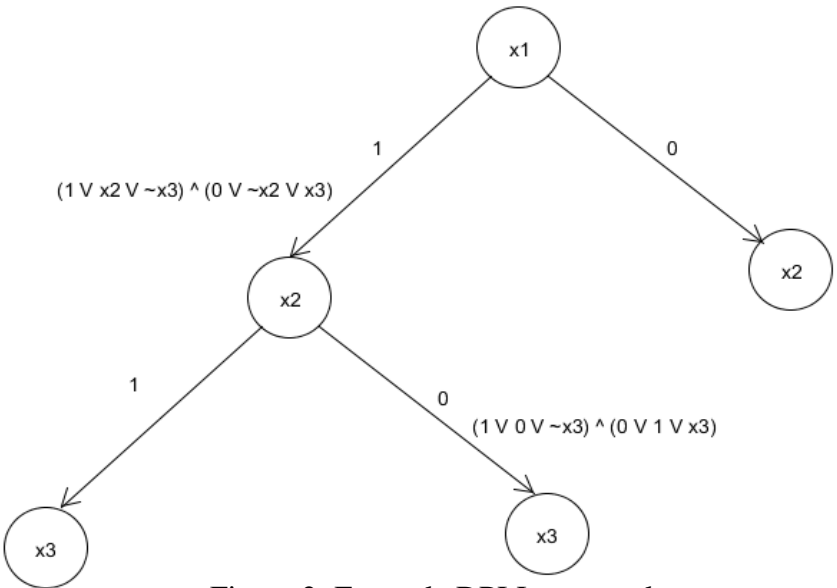

Figure 2: Example DPLL traversal

Starting with a choice for x_1 (it is not a pure literal this time), we then satisfy one clause, which is removed. Then the choice for x_2 is made knowing it would satisfy the remaining clause and thus the entire formula. The choice for x_3 at this point is irrelevant; the partial assignment satisfies the original formula.

2.1.1 Chaff Algorithm

In practice, most SAT Solvers spend the vast majority of their time doing Boolean Constraint Propagation (BCP). BCP identifies variable assignments required by the current state to satisfy the formula. For example, if there was a clause $(x_1 \vee 0)$, x_1 (an unassigned literal) would have to be true in order for the formula to be satisfied.

Chaff builds upon the backtracking capabilities of DPLL above (specifically DP, but the distinction and evolution of DP to DPLL is out of the scope of this paper). Chaff's success/speedup is the result of specifically engineering key steps of a SAT solver for efficiency:

- a highly optimized BCP algorithm
- a decision strategy highly optimized for speed, as well as focused on recently added clauses

Algorithm 2 Chaff

```
while (true) {
        if (!decide()) // if no unassigned vars
                return(satisifiable);
        while (!bcp()) {
                if (!resolveConflict())
                return(not satisfiable);
        }
}
bool resolveConflict() {
        d = most recent decision not 'tried both ways';
        if (d == NULL) // no such d was found
                return false;
        flip the value of d;
        mark d as tried both ways;
```

```
        undo any invalidated implications;
        return true;
}
```

The latest Chaff [zChaff] source code can be found at http://www.princeton.edu/~chaff/zchaff.html.

In order to optimize BCP, Chaff watches literals. Until a clause has two literals left (n-2 decisions, where n is the number of literals in a clause), there is no need to "check in" with it. When one of the watched literals in assigned zero, the clause is visited. One of two conditions must hold:

> (1) The clause is not implied, and thus at least 2 literals are not assigned to zero, including the other currently watched literal. This means at least one non-watched literal is not assigned to zero. We choose this literal to replace the one just assigned to zero. Thus, we maintain the property that the two watched literals are not assigned to 0.
>
> (2) The clause is implied. Follow the procedure for visiting an implied clause One should take note that the implied variable must always be the other watched literal, since, by definition, the clause only has one literal not assigned to zero, and one of the two watched literals is now assigned to zero [Moskewicz et al. 2001].

The naïve implementation of which variable to assign next is randomly selected. Other techniques exist such as seeing which variable appears most often in the remaining clauses. In order to improve on existing processes, Chaff employs a technique called Variable State Independent Decaying Sum(VSIDS):

> (1) Each variable in each polarity has a counter, initialized to 0.
> (2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.
> (3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.
> (4) Ties are broken randomly by default, although this is configurable
> (5) Periodically, all the counters are divided by a constant [Moskewicz et al. 2001].

Finally, Chaff uses “restarts”. In a backtracking algorithm, one might restart when stuck or when external factors call for it. Here, Chaff restarts voluntarily and uses some of the knowledge it gained in the previous pass. State is cleared, but clauses remained stored in the database for reference. This is to allow the chance for a change in early decisions.

2.2 Incomplete

Incomplete solvers are generally a permutation of the WalkSAT algorithm [rjlipton 2009].

Algorithm 3 WalkSAT

```
WalkSAT(φ)
  arbitrary truth assignment r
  if r satisfies φ
    return true
  while φ contains an unsatisfied clause and #steps < MAXSTEPS
    C = first unsatisfied clause
    flip random literal in C, update r
    if r(φ) == 1
      return true
return false
```

The greedy variant of the above is similar; it flips a variable in r that results in the max satisfied clauses rather then localizing the choice to an unsatisfied clause.

Algorithm 4 GSAT

```
GreedySAT(φ)
  arbitrary truth assignment r
  if r satisfies φ
    return true
  while φ contains an unsatisfied clause and #steps < MAXSTEPS
    flip random literal in r which results in max satisfied clauses
    if r(φ) == 1
      return true
return false
```

GSAT has been shown to be substantially faster than the DP/DPLL method on certain data sets including randomly generated formulas and SAT encodings of graph coloring problems [Selman et al. 1993]. WalkSAT and GSAT also outperform one another depending on problem class.

The issue with the greedy function is local minima. The algorithms reach a plateau where all choices result in similar (or same) number of unsatisfied clauses. Various methods have been added to attempt to resolve this issue including random walk and simulated annealing.

Algorithm 5 Random Walk

With probability p, pick a variable occurring in some unsatisfied clause
and flip its truth assignment.
With probability p-1, follow GSAT,
i.e. make the best possible local move [Selman et al. 1993].

Algorithm 5 combines the walk of algorithm 3 with the local greedy heuristic of GSAT. The probabilistic choice of which literal to flip allows the algorithm to make an "uphill" move out of the local minima.

Algorithm 6 Simulated Annealing

repeatedly pick a random variable
calculate δ: the change in the # of unsatisfied clauses when above
variable is flipped
if $\delta \leq 0$ //downhill/sideways move
make the flip
else
flip with probability $e^{\delta/T}$, *where* T is temperature

In the above algorithm, temperature can be constant or some decreasing factor. Given a finite cooling schedule, a global optimum is not guaranteed, random starts and possible restarts may be required.

In general, a “mixed” walk strategy has the best results dependent on the problem set. Local search with the appropriate random strategy is a powerful method for solving computationally challenging SAT problems [Selman et al. 1993].

3 Future Research and Conclusions

SAT solvers seem to be following the general trend on computer hardware: smaller and faster. Illustrated in this paper, both complete and incomplete categories have a base algorithm that comprises the core with various accruements placed on top to make the results faster. This does not appear to be changing any time soon. The engine(s) of SAT solvers is forecasted to remain relatively stable, given the latest and greatest developments are approaching their 20th anniversary.

As seen in this paper, the successfulness of a given approach to a SAT instance varies greatly on the data being processed. WalkSAT is well suited for automation planning/artificial intelligence problems, whereas Chaff is faster on Electronic Design Automation items. It speaks to the diversity of the problem domain where all the data can be represented in such a manner that a SAT solver can operate on it, but with such varying results. Knowing thy data is as much a solution to the problem as the algorithm itself.

4 References

Published Books and Papers

COOK, S.A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing,* Shaker Heights, Ohio, United States, Anonymous ACM, New York, NY, USA, 151-158.

KARP, R.M. 1972. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum. 85–103.

MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., ZHANG, L., MALIK, S. 2001. Chaff: Engineering an efficient sat solver. *Proceedings - Design Automation Conference*, *ISSU*(PAGE), 530-535.

OE, D., STUMP, A., OLIVER, C. AND CLANCY, K. 2012. versat: A Verified Modern SAT Solver. In *Verification, Model Checking, and Abstract Interpretation,* V. KUNCAK AND A. RYBALCHENKO, Eds. Springer Berlin Heidelberg, 363-378.

SELMAN, B., KAUTZ, H.A., COHEN, B. 1993. Local Search Strategies for Satisfiability Testing. *Second DIMACS Challenge on Cliques, Coloring, and Satisfiability*, David S. Johnson and Michael A. Trick, (editors). DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, AMS, 1996.

WWW References

SAT Solvers: Is SAT Hard or Easy? http://rjlipton.wordpress.com/2009/07/13/sat-solvers-is-sat-hard-or-easy/