CS 357 Lecture 2: Practical SAT Solving (part 1)

David L. Dill
Department of Computer Science
Stanford University

Motivation

A "SAT solver" is a program that automatically decides whether a propositional logic formula is satisfiable.

If it is satisfiable, a SAT solver will produce an example of a truth assignment that satisfies the formula.

SAT solvers have proved (haha) to be an indispensable component of many formal verification and (more recently) program analysis applications.

Basic idea: Since all NP-complete problems are mutually reducible:

1. Write one really good solver for NP-complete problems (in fact, get lots of people to do it. Hold competitions.)
2. Translate your NP-complete problems to that problem.

This doesn't always work, but it has worked pretty well for many formal verification and program analysis applications.

Readings

"GRASP: A Search Algorithm for Propositional Satisfiability" by Marques-Silva and Sakallah. This has a good explanation of conflict clause learning, including implication graphs.

"Chaff: Engineering an Efficient SAT Solver" Moskowicz, Madigan, Zhao, Zhang, and Malik When Chaff first came out, it was so much better than other SAT solvers that it opened up many new applications. The basic ideas were already invented, but they were integrated to perform much better than any existing solver. The best current general SAT solvers are based on the ideas of Chaff.

For enthusiasts, I recommend Don Knuth's draft of the SAT chapter of Vol 4B of "The Art of Programming."
`http://www-cs-faculty.stanford.edu/~uno/fasc6a.ps.gz`

Propositional logic

Propositional formulas:

1. Constants: **T** and **F** (which I may call 0 and 1 sometimes because I can't help it).
2. Propositional variables: p, q, x_{381274}, etc.
3. Propositional connectives: $\alpha \wedge \beta$, $\alpha \vee \beta$, $\neg\alpha$, $\alpha \rightarrow \beta$, $\alpha \leftrightarrow \beta$, etc., where α, β are propositional formulas.
4. Sometimes if-then-else ($\mathrm{ite}(x, y, z)$).

Of course, you only need to define a few connectives (possibly just one, **nand** or **nor**, and the rest can be defined

"EE" notation: 0, 1, p, q, $\alpha + \beta$, $\alpha \cdot \beta$, $\overline{\alpha}$, $\alpha \oplus \beta$, etc.

Word-level operations

One reason propositional logic is so useful is that it can be used to implement "first-order logic" over finite types.

Idea: If a type has n distinct values, use $\lceil \log_2(n) \rceil$ bits to encode it.

(or use n bits, which, surprisingly, is sometimes more efficient).

"Bit-blast" all variables into collections of $\lceil \log_2(n) \rceil$ propositional variables, where n is the number of values in the variable's types.

This is especially useful when the finite types are machine bytes, words, etc.

Language operations (e.g., "$+$") are not hard to implement if you know the simple hardware implementations in logic gates.

Advantage: Gives a bit-precise representation of k-bit signed and unsigned arithmetic, including wrap-around.

(More in future lectures.)

Conjunctive Normal Form (CNF)

A key property of current fast SAT solvers is tight inner loops.

The simple form of CNF is conducive to this (very few special cases in the code, etc.)

All current fast SAT solvers work on CNF (or slightly generalized CNF).

Terminology:

- A *literal* is a propositional variable or its negation (e.g., p or $\neg q$).
- A *clause* is a disjunction of literals (e.g., $(p \vee \neg q \vee r)$). Since $\vee$ is associative, we can represent clauses as lists of literals.
- A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses e.g., $(p \vee q \vee \neg r) \wedge (\neg p \vee s \vee t \vee \neg u)$ We can represent CNF formulas as vectors of vectors of literals, which are often integers. If "5" means x, "-5" means $\neg x$.

Reducing arbitrary propositional formulas to CNF

An arbitrary propositional logic can be reduced to an *equisatisfiable* (not equivalent) CNF formula in low-order polynomial time and space.

Preliminary step: Eliminate constants: $p \wedge T = p$, $p \wedge F = F$, etc. The resulting formula will be a constant (no need to convert to CNF), or will not contain constants.

You can also do other simplifications: $p \wedge p = p$, $p \wedge \neg p = F$, etc.

The obvious method is to apply simple identities, including distribution of $\vee$ over $\wedge$ in the right order until you have an equivalent CNF formula – but this can blow up exponentially.

Tseytin's transformation

Trick (Tseytin's transformation): Add new propositional variables, which act as names for subformulas of the original formula.

Definition: Propositional formulas α and β are equisatisfiable if one satisfiable iff the other is satisfiable.

If the CNF solver produces a satisfying assignment for the CNF formula, convert that to a satisfying assignment for the original formula by deleting the label variables.

Tseytin's transformation

Tseytin's transformation:

$$(p \land q) \lor \neg(q \lor r)$$

is equisatisfiable to

$$[x_1 \leftrightarrow (p \land q)] \land [x_2 \leftrightarrow (q \lor r)] \land [x_3 \leftrightarrow \neg x_2] \land (x_1 \lor x_3)$$

The conjuncts can be rewritten as clauses, using simple logical identities. E.g., the first conjunct above:

$[x_1 \to (p \land q)] \land [(p \land q) \to x_1]$ is equivalent to $(\neg x_1 \lor p) \land (\neg x_1 \lor q) \land (\neg p \lor \neg q \lor x_1)$.

Other issues in conversion to CNF

Subsequent processing is simplified if the clauses are not trivial or redundant, so we should simplify them.

If a literal occurs twice in a clause, delete one occurrence.

If a literal and its complement occur in a clause, delete the clause.

Also, the previous construction can be optimized in various ways to make the CNF smaller (e.g., don't have two new variables for α and $\neg\alpha$). (But, it's not clear in practice how important such optimizations are.)

Naive solver

Maximally naive solver (uses backtracking recursion):

ϕ is the formula. V is the propositional variables in ϕ, $A : V \to \{T, F\}$ is a (partial) truth assignment.

```
satisfy($\phi$) {
  if every clause of $\phi$ has a true literal, return T;
  if any clause of $\phi$ has all false literals, return F;
  choose an $x \in V$ that is unassigned in $A$,
    and choose $v \in \{T, F\}$.
  $A(x) = v$;
  if satisfy($\phi$) return T;
  $A(x) = \neg v$;
  if satisfy($\phi$) return T;
  unassign $A(x)$; // undo assignment for backtracking.
  return F; }
```

Naive satisfy, cont.

This can terminate early if:

- The formula is satisfied before all truth assignments are tested (less than full tree width).
- All clauses are false before all variables have been assigned (less than full tree depth).

. . . but it is not very fast.

Pure literal rule

If a variable is *always positive* or *always negative* in a CNF formula, you only need to set it to one value T for positive variables, F for negative variables.

Suppose x occurs only as positive literals in ϕ.

If ϕ is satisfied by A and $A(x) = F$, then ϕ is also satisfied by A' which is identical to A except that $A'(x) = T$.

. . . so don't bother trying $A(x) = F$.

Note that literals may "become pure" as variables are assigned, becaus all clauses in which a variable has one value may become true because of other variables.

The pure literal rule is not always used, and it is not clear how important it is.

Unit propagation

Unit propagation (a.k.a "Boolean constraint propagation" or BCP) is arguably the key component to fast SAT solving.

Whenever all the literals in a clause are false except one, the remaining literal must be true in any satisfying assignment (such a clause is called a "unit clause").

Therefore, the algorithm can assign it to true immediately.

After choosing a variable there are often *many* unit clauses.

Setting a literal in a unit clause often creates other unit clauses, leading to a cascade.

A good SAT solve often spends 80-90% of its time in unit propagation.

Unit propagation

```
BCP():
  Repeatedly search for unit clauses, and
    set unassigned literal to required value.
  If a literal is assigned conflicting values, return F
    else return T;


satisfy($\phi$) {
  if every clause of $\phi$ has a true literal, return T;
  if BCP() == F, return F;
  assign appropriate values to all pure literals;
  choose an $x \in V$ that is unassigned in $A$,
    and choose $v \in \{T, F\}$.
  $A(x) = v$;
  if satisfy($\phi$) return T;
  $A(x) = \neg v$;
  if satisfy($\phi$) return T;
  unassign $A(x)$; // undo assignment for backtracking.
  return F; }
```

DPLL

The naive algorithm with the pure literal rule and unit propagation is the classical Davis-Putnam-Logemann-Loveland (DPLL) method for solving CNF formulas from 1962.

Next: With additional optimizations, it is the preferred logic engine of for many formal verification and program analysis programs (and other applications, such as VLSI logic optimization, AI planning, optimization, etc.)

Watch pointers

SAT solvers spend most of their time in unit clause propagation.

The obvious implementation would do things like:

- Mark clauses as “satisfied”.
- Maintain a count of non-false literals in a clause, use this to detect unit clauses, unsatisfiable clauses.
- For each literal, maintain a list of clauses that maintain it.

But this is a lot of bookkeeping. It has to look at every clause that has the current decision literal to maintain the count.

“Watch pointers” eliminate the need to scan through clauses in this way.

Watch pointers allow the algorithm to focus on those clauses that are most likely to become unit clauses.

Watch pointers

Every clause has two watch pointers, which point to literals in the clause.

Each variable has two lists of “watched clauses”: Those with a watch pointer that points to its positive literal, and another for watch pointers that point to its negative literal.

Watch pointers, cont.

Invariant: If the clause is not satisfiable, the watch pointers point to two distinct literals, neither of which is false in A.

When a variable is newly assigned a truth value, only the clauses on one of these lists are searched.

E.g., when a variable is set to T, the clauses on its negative watch list will be violated (they now point to a false literal).

To restore the invariant, each unsatisfiable clause is processed

- If there is another non-false literal in the clause that is not watched, the watch pointer is changed to point to that literal. In this case, the clause is not unsatisfiable, not a unit clause.
- If the clause has no unwatched, non-false literals, it is a unit clause. Add it to a queue for unit propagation.
- If you propagate a literal and its negation, you have an inconsistency. Backtrack.

Conflict clauses

A conflict clause is a clause that is added to capture the causes of an inconsistency discovered during search.

Conflict clauses are very important. In a sense, they can “learn” from failed searches to improve future search.

They can also be thought of as “caching” previous search results.

More about conflict clauses in the next SAT lecture