CS 357 Lecture 2: Practical SAT Solving

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.

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 ($\text{ite}(x, y, z)$).

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.

Tseiten's transformation

Trick (Tseiten'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.

Relational Composition

General idea: Function composition vs. relation composition.

Function composition: $(f \circ g)(x) = f(g(x))$

See how to do the equivalent things with relations (predicates):

$R_g(x, y) : y = f(x)$

$R_f(y, z) : z = g(y)$

$(R_f \circ R_g)(x, z) : \exists y \; [R_g(x, y) \wedge R_f(y, z)]$

This is the same as the relation for $f \circ g$: $R_{f \circ g} : z = f(g(x))$

Relational Composition

Note: An arbitrary composition of multivariate functions, written in relational form, will be of the form $\exists x, y, \ldots \; R_1 \wedge R_2 \wedge \ldots \wedge R_n$

Surrounding a formula with existential quantifiers preserves satisfiability (satisfiability asks: "Does there *exist* a model?").

So this is satisfiable iff $R_1 \wedge R_2 \wedge \ldots \wedge R_n$ is satisfiable.

Tseiten's transformation

Tseiten's transformation is the previous "relational composition" idea applied to propositional formulas, using $\leftrightarrow$ for $=$.

$$(p \wedge q) \vee \neg(q \vee r)$$

is equisatisfiable to

$$[x_1 \leftrightarrow (p \wedge q)] \wedge [x_2 \leftrightarrow (q \vee r)] \wedge [x_3 \leftrightarrow \neg x_2] \wedge (x_1 \vee x_3)$$

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

$[x_1 \rightarrow (p \wedge q)] \wedge [(p \wedge q) \rightarrow x_1]$ is equivalent to
$(\neg x_1 \vee p) \wedge (\neg x_1 \vee q) \wedge (\neg p \vee \neg q \vee x_1)$.

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, $A \to V$ 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” were first used in the Chaff SAT solver, which was a major breakthrough in formal verification.

Watch pointers allow the algorithm to ignore all clauses that are not going to become unit clauses.

(Chaff was invented by undergraduates at Princeton.)

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.

Conflict clauses

Suppose, after a series of assignments, the solver discovers an inconsistency.

E.g., $x_1 = 1$, $x_2 = 0$, $x_3 = 1$

Then we know that: $\phi \rightarrow \wedge[x_1 \wedge \neg x_2 \wedge x_3) \rightarrow F]$

which is logically equivalent to: $\phi \rightarrow (\neg x_1 \vee x_2 \vee \neg x_3)$

Terminology: If $\phi \rightarrow \psi$, we call ψ an *implicate* of ϕ.

(Actually, I try not to call it that, because it is so easily confused with "implicant," a more common but essentially oppposite concept. But the GRASP paper uses the term a lot.)

And the cool thing is that we can add this clause to ϕ *without changing it:* $(\neg x_1 \vee x_2 \vee \neg x_3) \wedge \phi$ is logically equivalent to ϕ.

Implication graphs

To do useful clause learning, many SAT solvers maintain an *implication graph*, which captures the *relevant variables that caused an implication in BCP.*

Whenever we have to choose a variable assignment, the recursion depth is called the *decision level* of the variable.

The implication graph is a DAG. Vertices are of the form $x_i = v\texttt{@}d$, where x_i is a variable, v is a truth value, and d is a decision level.

BCP adds to the graph when it does unit propagation.

Implication Graph Example

Conflicts are also included as nodes in the implication graph.

Edges are called “implications.”

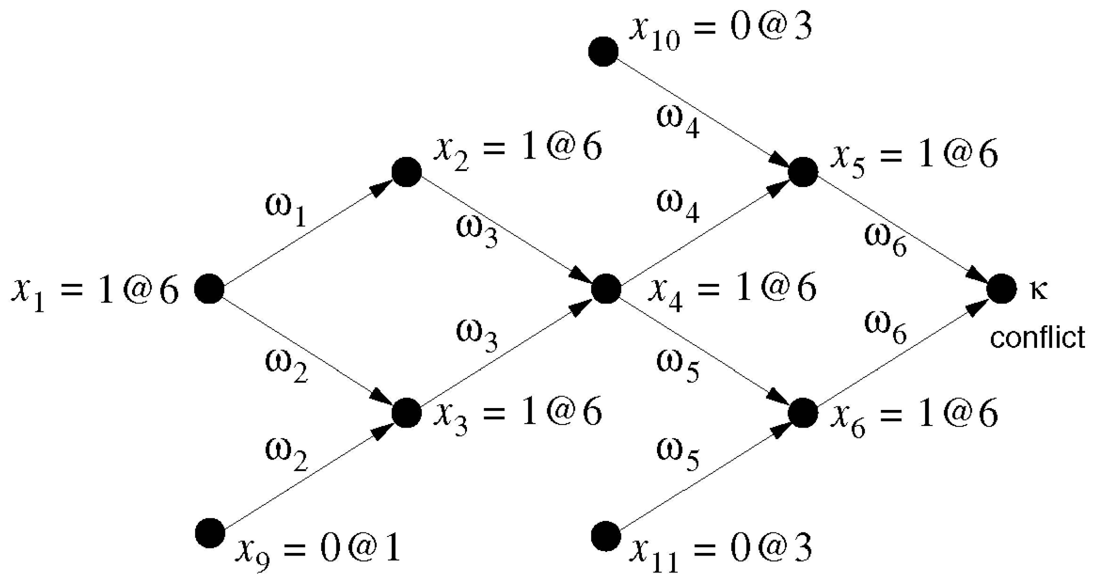
This is the example from the GRASP paper:

Current Truth Assignment: $\{x_9 = 0@1,\ x_{10} = 0@3,\ x_{11} = 0@3,\ x_{12} = 1@2,\ x_{13} = 1@2,\ \ldots\}$

Current Decision Assignment: $\{x_1 = 1@6\}$

$$
\begin{aligned}
\omega_1 &= (\neg x_1 + x_2)\\
\omega_2 &= (\neg x_1 + x_3 + x_9)\\
\omega_3 &= (\neg x_2 + \neg x_3 + x_4)\\
\omega_4 &= (\neg x_4 + x_5 + x_{10})\\
\omega_5 &= (\neg x_4 + x_6 + x_{11})\\
\omega_6 &= (\neg x_5 + \neg x_6)\\
\omega_7 &= (x_1 + x_7 + \neg x_{12})\\
\omega_8 &= (x_1 + x_8)\\
\omega_9 &= (\neg x_7 + \neg x_8 + \neg x_{13})\\
&\ldots
\end{aligned}
$$

Clause Database



Implication Graph for Current Decision Assignment

Implication graph

One way to find a conflict clause is to trace back from the conflict until find the source nodes.

The $\vee$ of the negations of these nodes is a conflict clause.

In this case, the conflict clause $(\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$ can be added.

Failure-Driven Assertions

Things happen automatically that we might otherwise have had to invent and work hard to implement.

Failure-driven assertions: When a decision to set $x_i = v$ fails, the only other possibility is $x_i = \neg v$ – if you don't undo previous decisions.

The conflict clause that was added automatically forces the solver to set $x_i = \neg v$.

If you add the conflict clause and then undo $x_i = v$, the new clause is unit clause and BCP immediately propagates $x_i = \neg v$, with no more implementation effort.

When this happens, x_i is not a decision variable because it was set by BCP.

Conflict-directed backtracking

(I think this was called “backjumping” in older SAT algorithms.)

The naive algorithm backtracks one level after trying both assignments for a variable.

Sometimes it is possible to jump back *many* levels, which can cut off a massive portion of the search tree.

Intuition: If we get conflicts for both values of a variable at decision level 20, and the previous relevant variable was at level 10, changing variables at levels 11..19 is not going to make any progress (none of those variables were relevant).

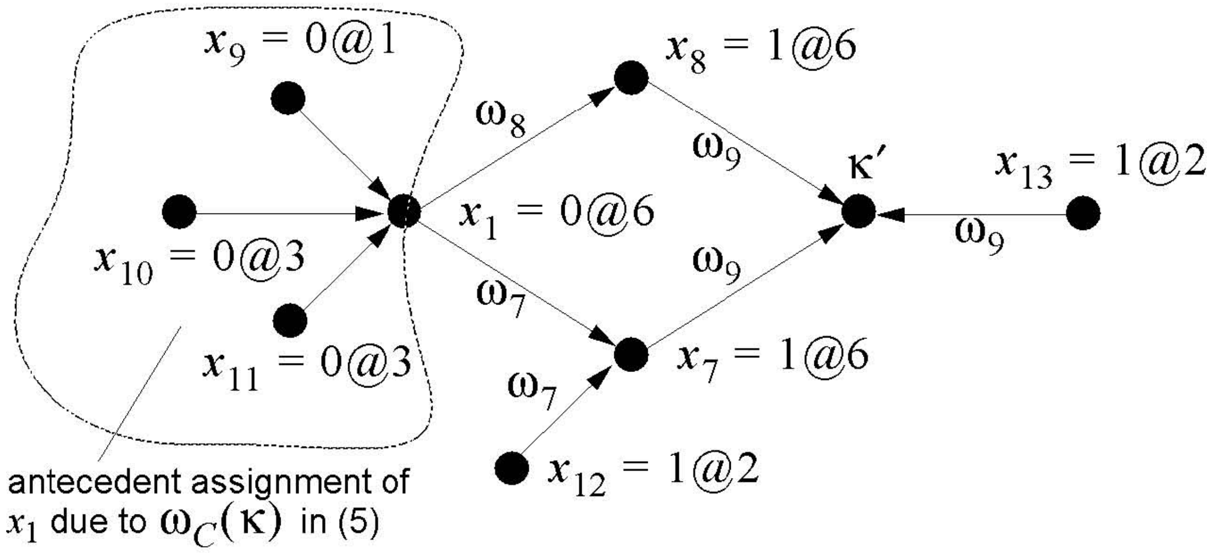
Conflict clauses exploit this automatically.

Conflict-directed backtracking

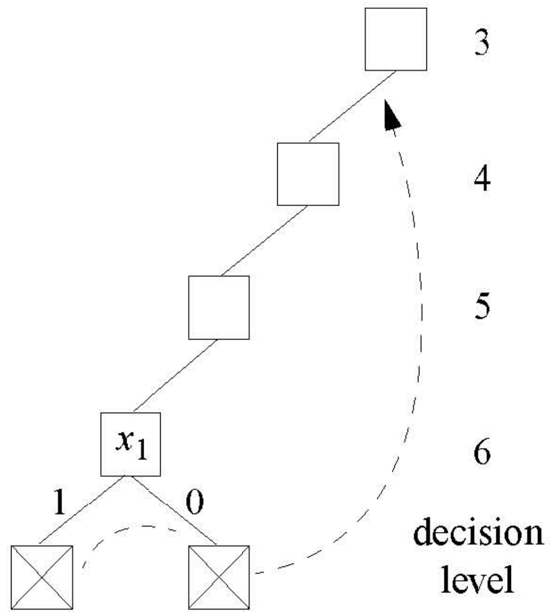
In the example, *after we add the first conflict clause*, the graph looks like this:

Even though we are at decision level 6, the highest-level decision variables in the conflict graph are at level 3.

Adding the resulting clause will cause all pending decisions at levels 4 and 5 to fail immediately (via BCP), so it immediately backtracks to level 3, skipping BCP.



(a)



(b)

Better conflict clauses

Sometimes you can find better conflict clauses if there is one node that separates the highest-level decision variable from the conflict.

Such a node is called a “unique implication point” (UIP).

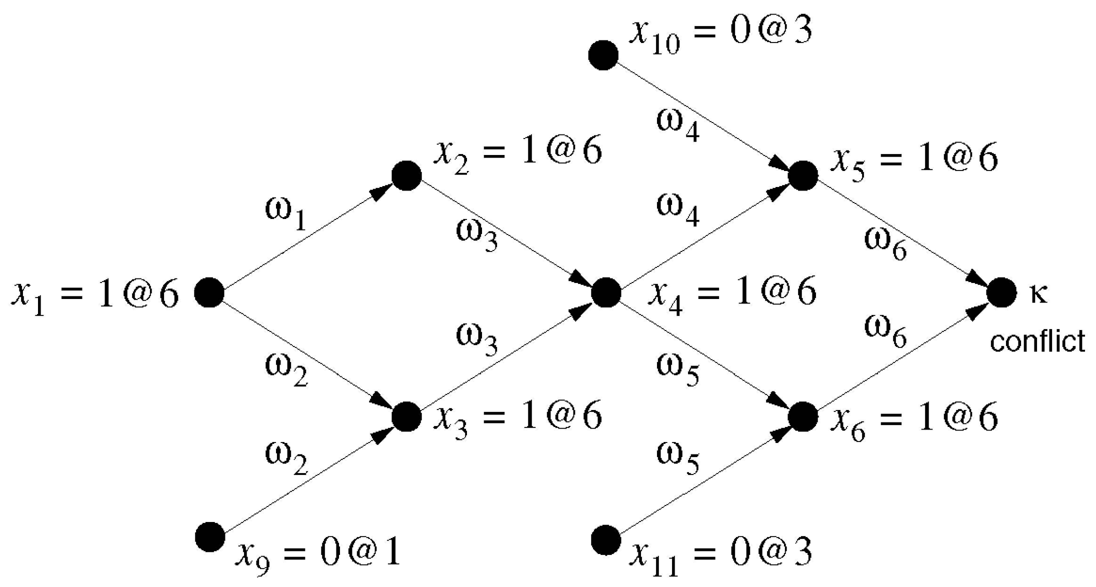
That node breaks a large conflict clause into two conflict clauses.

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2, \ldots\}$

Current Decision Assignment: $\{x_1 = 1@6\}$

$$\begin{aligned}
\omega_1 &= (\neg x_1 + x_2) \\
\omega_2 &= (\neg x_1 + x_3 + x_9) \\
\omega_3 &= (\neg x_2 + \neg x_3 + x_4) \\
\omega_4 &= (\neg x_4 + x_5 + x_{10}) \\
\omega_5 &= (\neg x_4 + x_6 + x_{11}) \\
\omega_6 &= (\neg x_5 + \neg x_6) \\
\omega_7 &= (x_1 + x_7 + \neg x_{12}) \\
\omega_8 &= (x_1 + x_8) \\
\omega_9 &= (\neg x_7 + \neg x_8 + \neg x_{13}) \\
&\ldots
\end{aligned}$$

Clause Database



Implication Graph for Current Decision Assignment

UIPs

Node x_4 is a UIP, so we can add clauses $(\neg x_4 \vee x_{10} \vee x_{11})$ and $(\neg x_1 \vee x_9 \vee x_4)$.

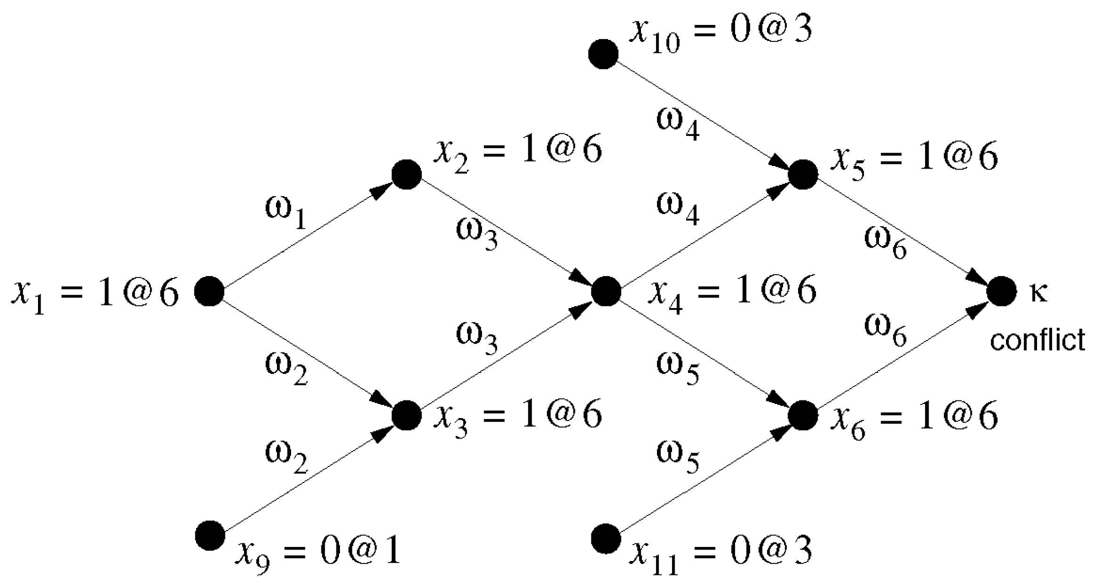
Perhaps more obviously: $(\neg x_9 \wedge x_1) \rightarrow x_4$ and $(x_4 \wedge \neg x_{10} \wedge \neg x_{11}) \rightarrow \kappa)$.

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2, \ldots\}$

Current Decision Assignment: $\{x_1 = 1@6\}$

$$
\begin{aligned}
\omega_1 &= (\neg x_1 + x_2) \\
\omega_2 &= (\neg x_1 + x_3 + x_9) \\
\omega_3 &= (\neg x_2 + \neg x_3 + x_4) \\
\omega_4 &= (\neg x_4 + x_5 + x_{10}) \\
\omega_5 &= (\neg x_4 + x_6 + x_{11}) \\
\omega_6 &= (\neg x_5 + \neg x_6) \\
\omega_7 &= (x_1 + x_7 + \neg x_{12}) \\
\omega_8 &= (x_1 + x_8) \\
\omega_9 &= (\neg x_7 + \neg x_8 + \neg x_{13}) \\
&\ldots
\end{aligned}
$$

Clause Database



Implication Graph for Current Decision Assignment

Variable selection

There are many heuristics for choosing which variable to assign next (and whether to assign T or F first).

While the variable selection heuristic can have a profound impact on efficiency, it is not clear that there is robust "best" heuristic.

Example hueristics:

RAND: Randomly choose the next variable and assignment. This has turned out to be better than you might have expected in some tests.

DLIS ("Dynamic largest individual sum"): Use literal that appears most frequently in unsatisfied clauses.

Variable selection, cont.

VSIDS: Increment a counter for each literal when a clause is added with that variable. Choose literal with highest count. Periodically divide counts by a constant. This focusses effort on recently added conflict clauses. (Used in CHAFF and MiniSAT).

MOM ("Maximum Occurrences on clauses of Minimum Size"). Let $f^*(\ell)$ be the number of occurrences of literal ℓ in the smallest unsatisfied clauses. Select literals that maximize: $[f^*(x) + f^*(\neg x)] * 2^k + f^*(x) * f^*(\neg x)$. Intuition: Focuses on making small clauses smaller, and prefers literals that appear in many clauses, and variables where both polarities appear in many clauses.

MOM is an old heuristic, and part of a family that combine several metrics, such as clause size, number of occurrences of literals in various types of clauses, etc.

Other ideas

Exploit polynomial-time special cases:

- Horn clauses
- 2-sat
- Linear algebra in GF(2) (CNF with XOR instead of OR in clauses).

For now, it seems that only the last has really been helpful (cryptominisat), but maybe that will change.

Closing remark

Simple and fast beats complex and smart.

In SAT. For the moment . . .