Computer programming

Application: SAT checking

Marius Minea

marius@cs.upt.ro

7 January 2014

Problem: Satisfiability of a propositional formula

Given a formula in *propositional logic*, is there a truth assignment (for its propositions) that makes the formula true ?

i.e., is the formula *satisfiable* ?

$$
\begin{aligned}
& (a \vee \neg b \vee \neg d) \\
\wedge\ & (\neg a \vee \neg b) \\
\wedge\ & (\neg a \vee c \vee \neg d) \\
\wedge\ & (\neg a \vee b \vee c)
\end{aligned}
$$

Usually, formulas are given/converted to *conjunctive normal form*:

a *conjunction* (AND) of *clauses*

each clause is a *disjunction* (OR) of *literals*

a literal is a positive or negated proposition

Why is SAT-checking important ?

Computers are built from *logic circuits*
which implement the same functions as in Boolean logic
⇒ to check equivalence of two function implementations f_1 and f_2
check if $f_1(v_1, ..., v_n) \oplus f_2(v_1, ..., v_n)$ is `UNSAT` (f_1, f_2 never differ)

Numbers are represented in base 2 (Boolean values 0 or 1, F or T)

Arithmetic is implemented in logic circuits

```
unsigned add(unsigned a, unsigned b) {
  // a ^ b: sum, a & b: carry (must shift left)
  return b ? add(a ^ b, (a & b) << 1) : a; // base: a + 0 = a
}
```

Sets can be represented as bitstrings of Boolean values
for each potential element: is it in the set or not ?

Anything in a computer ultimately has a bit representation

⇒ can use SAT-checking for decision problems, constraint solving, search, planning, software checking and testing, genetics, etc.

Why is SAT-checking important ?

It's the first problem proved to be *NP-complete*.
(believed not to have a solution in polynomial time)

P = class of problems solvable in polynomial time (in problem size)
NP = class of problems where an answer can be *checked* in polynomial time (checking a solution is easier than finding it)

NP-complete: the hardest problems in NP
if a polynomial solution to any of them were found,
then any problem in NP could be solved in polynomial time

$P = NP$? is one of the most fundamental questions in CS

Classic NP-complete problems: maximal clique, graph coloring, knapsack, subset sum, vertex cover, ...

How do we check satisfiability?

Simplification rules:

R1 A clause with a single literal $\Rightarrow$ has only one feasible value

in $\quad a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$ $\qquad$ a must be 1

in $\quad (a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$ $\qquad$ b must be 0

R2 If a literal is 1, *delete clauses* where it appears (they are true)
If a literal is 0, *delete literal* in all clauses (makes no difference)

Examples above simplify to:

$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \quad \stackrel{a=1}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$

$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c) \quad \stackrel{b=0}{\rightarrow} \quad a$
(thus $a = 1$, formula is $\texttt{SAT}$)

How do we check satisfiability?

R3) If *no more clauses*, done (we have a satisfying assignment)
If we get an *empty clause*, formula is *unsatisfiable* (can't be true)

$$a \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

$$\stackrel{a=1}{\rightarrow} b \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$$

$$\stackrel{b=1}{\rightarrow} c \wedge \neg c \quad \stackrel{c=1}{\rightarrow} \quad \emptyset \ (\neg c \text{ becomes empty clause} \Rightarrow \texttt{UNSAT})$$

What if *no more simplifications* can be done ?

$$a \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \quad \stackrel{a=1}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c) \quad ??$$

R4) Choose a variable and try both values (*case splitting*)

- try value 1 (true)
- try value 0 (false)

A solution in any case is good.
If no case has a solution, formula is `UNSAT`

Towards an algorithm

Need to manipulate

- list of clauses (the formula)
- set of already assigned variables (initially empty)

Rules 1 and 2 *reduce the problem* to a simpler one
(fewer unknowns or fewer clauses or simpler clauses)

Rule 3 gives the *stopping condition*

Rule 4 reduces problem to *two simpler problems* (one variable less)
$\Rightarrow$ naturally *recursive* solution

DPLL Algorithm (Davis-Putnam-Logemann-Loveland)

```
function solve(env: lit set, clauses: lit list list)
(newenv, clauses) = simplify(env, clauses) (* R1, R2 *)
if clauses = empty list then
   return env; (* variable assignment *)
if clauses has empty clause then
   return false; (* unsatisfiable *)
if clauses contains single literal a then
   solve (env with a=true, clauses)
else
   return solve (env with a=false, clauses)
        or solve (env with a=true, clauses);
```

Current optimized SAT-checkers can handle $\sim 10^6$ variables

Implementation: need lists and sets

Data structures:

- *list* of clauses (list of list of literals)
- *set* of true literals

Processing

- *membership* check: is a literal in set of assigned literals ?
- *add* a literal to set of assigned literals
- *traverse* literals in a clause
- *delete* literal in a clause
- *delete* clause from a list (formula)