Introduction to SAT

Christoph Weidenbach

Winter Term 2007/2008

Part 1: Propositional Logic

Propositional logic

- logic of truth values
- decidable (but NP-complete)
- can be used to describe functions over a finite domain
- important for hardware applications (e. g., model checking)

1.1 Syntax

- propositional variables
- logical symbols
  $\Rightarrow$ Boolean combinations

Propositional Variables

Let Π be a set of propositional variables.

We use letters P, Q, R, S, to denote propositional variables.

Propositional Formulas

F_Π is the set of propositional formulas over Π defined as follows:

$$
\begin{array}{rcll}
F, G, H & ::= & \bot & \text{(falsum)} \\
& | & \top & \text{(verum)} \\
& | & P, \quad P \in \Pi & \text{(atomic formula)} \\
& | & \neg F & \text{(negation)} \\
& | & (F \wedge G) & \text{(conjunction)} \\
& | & (F \vee G) & \text{(disjunction)} \\
& | & (F \rightarrow G) & \text{(implication)} \\
& | & (F \leftrightarrow G) & \text{(equivalence)}
\end{array}
$$

1.2 Semantics

In classical logic (dating back to Aristoteles) there are “only” two truth values “true” and “false” which we shall denote, respectively, by 1 and 0.

There are multi-valued logics having more than two truth values.

Valuations

A propositional variable has no intrinsic meaning. The meaning of a propositional variable has to be defined by a valuation.

A Π-valuation is a map

$$\mathcal{A} : \Pi \rightarrow \{0, 1\}.$$

where $\{0, 1\}$ is the set of truth values.

Truth Value of a Formula in $\mathcal{A}$

Given a Π-valuation $\mathcal{A}$, the function $\mathcal{A}^*$: Σ-formulas $\to \{0, 1\}$ is defined inductively over the structure of F as follows:

$$\begin{aligned} \mathcal{A}^*(\bot) &= 0 \\ \mathcal{A}^*(\top) &= 1 \\ \mathcal{A}^*(P) &= \mathcal{A}(P) \\ \mathcal{A}^*(\neg F) &= \mathsf{B}_\neg(\mathcal{A}^*(F)) \\ \mathcal{A}^*(F \rho G) &= \mathsf{B}_\rho(\mathcal{A}^*(F), \mathcal{A}^*(G)) \end{aligned}$$

where B_ρ is the Boolean function associated with ρ defined by the usual truth table.

Truth Value of a Formula in $\mathcal{A}$

For simplicity, we write $\mathcal{A}$ instead of $\mathcal{A}^*$.

We also write ρ instead of B_ρ, i. e., we use the same notation for a logical symbol and for its meaning (but remember that formally these are different things.)

1.3 Models, Validity, and Satisfiability

F is valid in $\mathcal{A}$ ($\mathcal{A}$ is a model of F; F holds under $\mathcal{A}$):

$$\mathcal{A} \models F \;\;:\Leftrightarrow\;\; \mathcal{A}(F) = 1$$

F is valid (or is a tautology):

$$\models F \;\;:\Leftrightarrow\;\; \mathcal{A} \models F \text{ for all } \Pi\text{-valuations } \mathcal{A}$$

F is called satisfiable if there exists an $\mathcal{A}$ such that $\mathcal{A} \models F$.
Otherwise F is called unsatisfiable (or contradictory).

Entailment and Equivalence

F entails (implies) G (or G is a consequence of F), written $F \models G$, if for all Π-valuations $\mathcal{A}$, whenever $\mathcal{A} \models F$ then $\mathcal{A} \models G$.

F and G are called equivalent, written $F \models\mid G$, if for all Π-valuations $\mathcal{A}$ we have $\mathcal{A} \models F \Leftrightarrow \mathcal{A} \models G$.

Proposition 1.1:
$F \models G$ if and only if $\models (F \rightarrow G)$.

Proposition 1.2:
$F \models\mid G$ if and only if $\models (F \leftrightarrow G)$.

Entailment and Equivalence

Extension to sets of formulas N in the "natural way":

$N \models F$ if for all Π-valuations $\mathcal{A}$:

if $\mathcal{A} \models G$ for all $G \in N$, then $\mathcal{A} \models F$.

Validity vs. Unsatisfiability

Validity and unsatisfiability are just two sides of the same medal as explained by the following proposition.

Proposition 1.3:
F is valid if and only if $\neg F$ is unsatisfiable.

Hence in order to design a theorem prover (validity checker) it is sufficient to design a checker for unsatisfiability.

Validity vs. Unsatisfiability

In a similar way, entailment $N \models F$ can be reduced to unsatisfiability:

Proposition 1.4:
$N \models F$ if and only if $N \cup \{\neg F\}$ is unsatisfiable.

Checking Unsatisfiability

Every formula F contains only finitely many propositional variables. Obviously, $\mathcal{A}(F)$ depends only on the values of those finitely many variables in F under $\mathcal{A}$.

If F contains n distinct propositional variables, then it is sufficient to check 2^n valuations to see whether F is satisfiable or not.
$\Rightarrow$ truth table.

So the satisfiability problem is clearly deciadable
(but, by Cook's Theorem, NP-complete).

Nevertheless, in practice, there are (much) better methods than truth tables to check the satisfiability of a formula. (later more)

1.4 Normal Forms

We define conjunctions of formulas as follows:

$$\bigwedge_{i=1}^{0} F_i = \top.$$

$$\bigwedge_{i=1}^{1} F_i = F_1.$$

$$\bigwedge_{i=1}^{n+1} F_i = \bigwedge_{i=1}^{n} F_i \wedge F_{n+1}.$$

and analogously disjunctions:

$$\bigvee_{i=1}^{0} F_i = \bot.$$

$$\bigvee_{i=1}^{1} F_i = F_1.$$

$$\bigvee_{i=1}^{n+1} F_i = \bigvee_{i=1}^{n} F_i \vee F_{n+1}.$$

Literals and Clauses

A literal is either a propositional variable P or a negated propositional variable $\neg P$.

A clause is a (possibly empty) disjunction of literals.

CNF and DNF

A formula is in conjunctive normal form (CNF, clause normal form), if it is a conjunction of disjunctions of literals (or in other words, a conjunction of clauses).

A formula is in disjunctive normal form (DNF), if it is a disjunction of conjunctions of literals.

Warning: definitions in the literature differ:

- are complementary literals permitted?
- are duplicated literals permitted?
- are empty disjunctions/conjunctions permitted?

CNF and DNF

Checking the validity of CNF formulas or the unsatisfiability of DNF formulas is easy:

A formula in CNF is valid, if and only if each of its disjunctions contains a pair of complementary literals P and $\neg P$.

Conversely, a formula in DNF is unsatisfiable, if and only if each of its conjunctions contains a pair of complementary literals P and $\neg P$.

On the other hand, checking the unsatisfiability of CNF formulas or the validity of DNF formulas is known to be coNP-complete.

1.5 The DPLL Procedure

Goal:
Given a propositional formula in CNF (or alternatively, a finite set N of clauses), check whether it is satisfiable (and optionally: output *one* solution, if it is satisfiable).

Assumption:
Clauses contain neither duplicated literals nor complementary literals.

Notation:
$\overline{L}$ is the complementary literal of L,
i. e., $\overline{P} = \neg P$ and $\overline{\neg P} = P$.

Satisfiability of Clause Sets

$\mathcal{A} \models N$ if and only if $\mathcal{A} \models C$ for all clauses C in N.

$\mathcal{A} \models C$ if and only if $\mathcal{A} \models L$ for some literal $L \in C$.

Partial Valuations

Since we will construct satisfying valuations incrementally,
we consider partial valuations
(that is, partial mappings $\mathcal{A} : \Pi \to \{0, 1\}$).

Every partial valuation $\mathcal{A}$ corresponds to a set M of literals that does not contain complementary literals, and vice versa:

$\mathcal{A}(L)$ is true, if $L \in M$.

$\mathcal{A}(L)$ is false, if $\overline{L} \in M$.

$\mathcal{A}(L)$ is undefined, if neither $L \in M$ nor $\overline{L} \in M$.

We will use $\mathcal{A}$ and M interchangeably.

Partial Valuations

A clause is true under a partial valuation $\mathcal{A}$
(or under a set M of literals) if one of its literals is true;
it is false (or "conflicting") if all its literals are false;
otherwise it is undefined (or "unresolved").

Unit Clauses

Observation:

Let $\mathcal{A}$ be a partial valuation. If the set N contains a clause C, such that all literals but one in C are false under $\mathcal{A}$, then the following properties are equivalent:

- there is a valuation that is a model of N and extends $\mathcal{A}$.
- there is a valuation that is a model of N and extends $\mathcal{A}$ and makes the remaining literal L of C true.

C is called a unit clause; L is called a unit literal.

Pure Literals

One more observation:

Let $\mathcal{A}$ be a partial valuation and P a variable that is undefined under $\mathcal{A}$. If P occurs only positively (or only negatively) in the unresolved clauses in N, then the following properties are equivalent:

- there is a valuation that is a model of N and extends $\mathcal{A}$.
- there is a valuation that is a model of N and extends $\mathcal{A}$ and assigns true (false) to P.

P is called a pure literal.

The Davis-Putnam-Logemann-Loveland Proc.

boolean DPLL(literal set M, clause set N) {
    if (all clauses in N are true under M) return true;
    elsif (some clause in N is false under M) return false;
    elsif (N contains unit clause P) return DPLL($M \cup \{P\}$, N);
    elsif (N contains unit clause $\neg P$) return DPLL($M \cup \{\neg P\}$, N);
    elsif (N contains pure literal P) return DPLL($M \cup \{P\}$, N);
    elsif (N contains pure literal $\neg P$) return DPLL($M \cup \{\neg P\}$, N);
    else {
        let P be some undefined variable in N;
        if (DPLL($M \cup \{\neg P\}$, N)) return true;
        else return DPLL($M \cup \{P\}$, N);
    }
}

The Davis-Putnam-Logemann-Loveland Proc.

Initially, DPLL is called with an empty literal set and the clause set N.

DPLL Iteratively

In practice, there are several changes to the procedure:

The pure literal check is often omitted (it is too expensive).

The branching variable is not chosen randomly.

The algorithm is implemented iteratively;
the backtrack stack is managed explicitly
(it may be possible and useful to backtrack more than one level).

Information is reused by learning.

Branching Heuristics

Choosing the right undefined variable to branch is important for efficiency, but the branching heuristics may be expensive itself.

State of the art: use branching heuristics that need not be recomputed too frequently.

In general: choose variables that occur frequently.

The Deduction Algorithm

For applying the unit rule, we need to know the number of literals in a clause that are not false.

Maintaining this number is expensive, however.

The Deduction Algorithm

Better approach: "Two watched literals":

In each clause, select two (currently undefined) "watched" literals.

For each variable P, keep a list of all clauses in which P is watched and a list of all clauses in which $\neg P$ is watched.

If an undefined variable is set to 0 (or to 1), check all clauses in which P (or $\neg P$) is watched and watch another literal (that is true or undefined) in this clause if possible.

Watched literal information need not be restored upon backtracking.

Conflict Analysis and Learning

Goal: Reuse information that is obtained in one branch in further branches.

Method: Learning:

> If a conflicting clause is found, derive a new clause from the conflict and add it to the current set of clauses.

Problem: This may produce a large number of new clauses; therefore it may become necessary to delete some of them afterwards to save space.

Backjumping

Related technique:

non-chronological backtracking ("backjumping"):

If a conflict is independent of some earlier branch, try to skip over that backtrack level.

Restart

Runtimes of DPLL-style procedures depend extremely on the choice of branching variables.

If no solution is found within a certain time limit, it can be useful to restart from scratch with another choice of branchings (but learned clauses may be kept).

1.6 Splitting into Horn Clauses

- A *Horn clause* is a clause with at most one positive literal.
- They are typically denoted as implications: $P_1, \ldots, P_n \rightarrow Q$. (In general we can write $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$ for $\neg P_1 \vee \ldots \vee \neg P_n \vee Q_1 \vee \ldots \vee Q_m$.)
- Compared to arbitrary clause sets, Horn clause sets enjoy further properties:
  - Horn clause sets have unique minimal models.
  - Checking satisfiability is often of lower complexity.

Propositional Horn Clause SAT is in P

boolean HornSAT(literal set M, Horn clause set N) {
    if (all clauses in N are supported by M) return true;
    elsif (a negative clause in N is not supported by M) return false;
    elsif (N contains clause $P_1, \ldots, P_n \rightarrow Q$ where
            $\{P_1, \ldots, P_n\} \subseteq M$ and $Q \notin M$)
        return HornSAT($M \cup \{Q\}$, N);
}

A clause $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$ is *supported* by M if $\{P_1, \ldots, P_n\} \not\subseteq M$ or some $Q_i \in M$. A *negative* clause consists of negative literals only.

Initially, HornSAT is called with an empty literal set M.

Propositional Horn Clause SAT is in **P**

Lemma 1.5:

Let N be a set of propositional Horn clauses.Then:

(1) HornSAT($\emptyset$, N)=true iff N is satisfiable

(2) HornSAT is in **P**

SplitHornSAT

boolean SplitHornSAT(clause set N) {

if (N is Horn)

g return HornSAT($\emptyset$,N);

else {

select non Horn clause $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$ from N;

$N' = N \setminus \{P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m\}$;

if (SplitHornSAT($N' \cup \{P_1, \ldots, P_n \rightarrow Q_1\}$)) return true;

else return

SplitHornSAT($N' \cup \{\rightarrow Q_2, \ldots, Q_m\} \cup \bigcup_i \{\rightarrow P_i\} \cup \{Q_1 \rightarrow\}$);

}

}

SplitHornSAT

Lemma 1.6:

Let N be a set of propositional clauses. Then:

(1) SplitHornSAT(N)=true iff N is satisfiable

(2) SplitHornSAT(N) terminates

1.7 Other Calculi

OBDDs (Ordered Binary Decision Diagrams):

Minimized graph representation of decision trees, based on a fixed ordering on propositional variables,

see script of the Computational Logic course,

see Chapter 6.1/6.2 of Michael Huth and Mark Ryan: *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge Univ. Press, 2000.

FRAIGs (Fully Reduced And-Inverter Graphs)

Minimized graph representation of boolean circuits.