Propositional Logic & SAT Solvers

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2022/2023

Roadmap

- Introduction
- Review of Propositional Logic
- SAT solving algorithms
- Modeling with PL

Introduction

What is a (formal) logic?

A formal logic consists of

- A *logical language* in which (well-formed) sentences are expressed. It consists of
  - *logical symbols* whose interpretations are fixed
  - *non-logical symbols* whose interpretations vary
- A *semantics* that defines the intended interpretation of the symbols and expressions of the logical language.
- A *proof system* that is a framework of rules for deriving valid judgments.

Logic and computer science

- Logic and computer science share a symbiotic relationship.
  - Logic provides language and methods for the study of theoretical computer science.
  - Computers provide a concrete setting for the implementation of logic.
- Logic is a fundamental part of computer science.
  - Program analysis: static analysis, software verification, test case generation, program understanding, ...
  - Artificial intelligence: constraint satisfaction, automated game playing, planning, ...
  - Hardware verification: correctness of circuits, ATPG, ...
  - Programming Languages: logic programming, type systems, programming language theory, ...

What is SAT?

- The Boolean satisfiability (SAT) problem:
  - *Find an assignment to the propositional variables of the formula such that the formula evaluates to TRUE, or prove that no such assignment exists.*
- SAT is an NP-complete decision problem.
  - SAT was the first problem to be shown NP-complete.
  - There are no known polynomial time algorithms for SAT.

What is SAT?

- Usually SAT solvers deal with formulas in conjunctive normal form (CNF)
  - literal: propositional variable or its negation. $A, \neg A, B, \neg B, C, \neg C$
  - clause: disjuntion of literals. $(A \vee \neg B \vee C)$
  - conjunctive normal form: conjuction of clauses. $(A \vee \neg B \vee C) \wedge (B \vee \neg A) \wedge \neg C$
- SAT is a success story of computer science
  - Modern SAT solvers can check formulas with hundreds of thousands variables and millions of clauses in a reasonable amount of time.
  - A huge number of practical applications.

Why should we care?

- No matter what your research area or interest is, SAT solving is likely to be relevant.
- Very good toolkit because many difficult problems can be reduced deciding satisfiabilty of formulas in logic.

(Classical) Propositional Logic

Propositional logic

- The language of propositional logic is based on propositions, or declarative sentences which one can, in principle, argue as being "true" or "false".
- Propositional symbols are the atomic formulas of the language. More complex sentences are constructed using logical connectives.
- In classical propositional logic (PL) each sentence is either true or false.
- In fact, the content of the propositions is not relevant to PL. PL is not the study of truth, but of the relationship between the truth of one statement and that of another.

Syntax

The alphabet of the propositional language is organised into the following categories.

- *Propositional variables*: $P, Q, R, \ldots \in \mathcal{V}_{\text{Prop}}$ (a countably infinite set)
- *Logical connectives*: $\bot$ (*false*) ,$\top$ (*true*), $\neg$ (*not*), $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*), $\leftrightarrow$ (*equivalent*)
- *Auxiliary symbols*: "(" and ")".

The set **Form** of *formulas* of propositional logic is given by the abstract syntax

$$\textbf{Form} \ni A, B \ ::= \ P \mid \bot \mid \top \mid (\neg A) \mid (A \wedge B) \mid (A \vee B) \mid (A \rightarrow B) \mid (A \leftrightarrow B)$$

We let $A, B, C, F, G, H, \ldots$ range over **Form**.

Outermost parenthesis are usually dropped. In absence of parentheses, we adopt the following convention about precedence. Ranging from the highest precedence to the lowest, we have respectively: $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. All binary connectives are right-associative.

Semantics

The meaning of PL is given by the truth values true and false, where true $\neq$ false. We will represent true by 1 and false by 0.

An *assignment* is a function $\mathcal{A} : \mathcal{V}_{\text{Prop}} \rightarrow \{0, 1\}$, that assigns to every propositional variable a truth value.
An assignment $\mathcal{A}$ naturally extends to all formulas, $\mathcal{A} : \textbf{Form} \rightarrow \{0, 1\}$.
The truth value of a formula is computed using *truth tables*:

F	A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$	$\bot$	$\top$
$\mathcal{A}_1(F)$	0	1	1	0	1	1	0	0	1
$\mathcal{A}_2(F)$	0	0	1	0	0	1	1	0	1
$\mathcal{A}_3(F)$	1	1	0	1	1	1	1	0	1
$\mathcal{A}_4(F)$	1	0	0	0	1	0	0	0	1

Semantics

Let $\mathcal{A}$ be an assignment and let F be a formula.
If $\mathcal{A}(F) = 1$, then we say F *holds* under assignment $\mathcal{A}$, or $\mathcal{A}$ *models* F.
We write $\mathcal{A} \models F$ iff $\mathcal{A}(F) = 1$, and $\mathcal{A} \not\models F$ iff $\mathcal{A}(F) = 0$.

An alternative (inductive) definition of $\mathcal{A} \models F$ is

$$
\begin{array}{lll}
\mathcal{A} \models \top & & \\
\mathcal{A} \not\models \bot & & \\
\mathcal{A} \models P & \text{iff} & \mathcal{A}(P) = 1 \\
\mathcal{A} \models \neg A & \text{iff} & \mathcal{A} \not\models A \\
\mathcal{A} \models A \wedge B & \text{iff} & \mathcal{A} \models A \text{ and } \mathcal{A} \models B \\
\mathcal{A} \models A \vee B & \text{iff} & \mathcal{A} \models A \text{ or } \mathcal{A} \models B \\
\mathcal{A} \models A \to B & \text{iff} & \mathcal{A} \not\models A \text{ or } \mathcal{A} \models B \\
\mathcal{A} \models A \leftrightarrow B & \text{iff} & \mathcal{A} \models A \text{ iff } \mathcal{A} \models B
\end{array}
$$

Validity, satisfiability, and contradiction

A formula F is

valid	iff	it holds under every assignment. We write $\models F$. A valid formula is called a *tautology*.
satisfiable	iff	it holds under some assignment.
unsatisfiable	iff	it holds under no assignment. An unsatisfiable formula is called a *contradiction*.
refutable	iff	it is not valid.

Proposition

F is **valid** iff $\neg F$ is **unsatisfiable**

$(A \wedge (A \to B)) \to B$ is valid. $A \to B$ is satisfiable and refutable.
$A \wedge \neg A$ is a contradiction.

Consequence and equivalence

- $F \models G$ iff for every assignment $\mathcal{A}$, if $\mathcal{A} \models F$ then $\mathcal{A} \models G$. We say G is a *consequence* of F.
- $F \equiv G$ iff $F \models G$ and $G \models F$. We say F and G are *equivalent*.
- Let $\Gamma = \{F_1, F_2, F_3, \dots\}$ be a set of formulas.
  $\mathcal{A} \models \Gamma$ iff $\mathcal{A} \models F_i$ for each formula F_i in Γ. We say $\mathcal{A}$ *models* Γ.
  $\Gamma \models G$ iff $\mathcal{A} \models \Gamma$ implies $\mathcal{A} \models G$ for every assignment $\mathcal{A}$. We say G is a *consequence* of Γ.

Proposition

- $F \models G$ iff $\models F \to G$
- $\Gamma \models G$ and Γ finite iff $\models \bigwedge \Gamma \to G$

Some basic equivalences

$$
\begin{array}{lcl}
A \vee A & \equiv & A \\
A \wedge A & \equiv & A \\
\\
A \vee B & \equiv & B \vee A \\
A \wedge B & \equiv & B \wedge A \\
\\
A \wedge (A \vee B) & \equiv & A \\
\\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C) \\
A \vee (B \wedge C) & \equiv & (A \vee B) \wedge (A \vee C) \\
\\
\neg(A \vee B) & \equiv & \neg A \wedge \neg B \\
\neg(A \wedge B) & \equiv & \neg A \vee \neg B
\end{array}
$$

$$
\begin{array}{lcl}
A \wedge \neg A & \equiv & \bot \\
A \vee \neg A & \equiv & \top \\
\\
A \wedge \top & \equiv & A \\
A \vee \top & \equiv & \top \\
\\
A \wedge \bot & \equiv & \bot \\
A \vee \bot & \equiv & A \\
\\
\neg\neg A & \equiv & A \\
\\
A \to B & \equiv & \neg A \vee B
\end{array}
$$

Consistency

Let $\Gamma = \{F_1, F_2, F_3, \dots\}$ be a set of formulas.

- Γ is *consistent* or *satisfiable* iff there is an assignment that models Γ.
- We say that Γ is *inconsistent* or *unsatisfiable* iff it is not consistent and denote this by $\Gamma \models \bot$.

Proposition

- $\{F, \neg F\} \models \bot$
- If $\Gamma \models \bot$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \models \bot$.
- $\Gamma \models F \quad \text{iff} \quad \Gamma, \neg F \models \bot$

Substitution

- Formula G is a *subformula* of formula F if it occurs syntactically within F.
- Formula G is a *strict subformula* of F if G is a subformula of F and $G \neq F$

Substitution theorem

Suppose $F \equiv G$. Let H be a formula that contains F as a subformula. Let H' be the formula obtained by replacing some occurrence of F in H with G. Then $H \equiv H'$.

Decidability

Given formulas F and G as input, we may ask:

Decision problems

Validity problem:	"Is F valid ?"
Satisfiability problem:	"Is F satisfiable ?"
Consequence problem:	"Is G a consequence of F ?"
Equivalence problem:	"Are F and G equivalent ?"

All these problems are **decidable!**

Decidability

Any algorithm that works for one of these problems also works for all of these problems!

F is satisfiable	iff	$\neg F$ is not valid
$F \models G$	iff	$\neg(F \to G)$ is not satisfiable
$F \equiv G$	iff	$F \models G$ and $G \models F$
F is valid	iff	$F \equiv \top$

Truth-table method

For the satisfiability problem, we first compute a truth table for F and then check to see if its truth value is ever one.

This algorithm certainly works, but is very inefficient.
It's exponential-time! $\quad \mathcal{O}(2^n)$

If F has n atomic formulas, then the truth table for F has 2^n rows.

Complexity

- Computing a truth table for a formula is exponential-time in order to the number of propositional variables.
- There are several techniques and algorithms for SAT solving that perform better in average.
- There are no known polynomial time algorithms for SAT.
  - If it exists, then **P** = **NP**, because the SAT problem for PL is NP-complete (it was the first one to be shown NP-complete).

Cook's theorem (1971)

SAT is NP-complete.

- Conjecture: Any algorithm that solves SAT is exponential in the number of variables, in the worst-case.

An example

The unicorn puzzle

- If the unicorn is mythical, then it is immortal.
- If the unicorn is not mythical, then it is a mortal mammal.
- If the unicorn is either immortal or a mammal, then it is horned.
- The unicorn is magical if it is horned.

- Questions:
  - Is the unicorn magical?
  - Is it horned?
  - Is it mythical?

An example

- Consider the following propositional variables:
  - M - the unicorn is mythical
  - I - the unicorn is immortal
  - A - the unicorn is mammal
  - H - the unicorn is horned
  - G - the unicorn is magical

- If the unicorn is mythical, then it is immortal.
  $M \to I$
- If the unicorn is not mythical, then it is a mortal mammal.
  $\neg M \to (\neg I \wedge A)$
- If the unicorn is either immortal or a mammal, then it is horned.
  $(I \vee A) \to H$
- The unicorn is magical if it is horned.
  $H \to G$

An example

Let Γ be $\{\, M \to I, \neg M \to (\neg I \wedge A), (I \vee A) \to H, H \to G \,\}$

Questions:

- Is the unicorn magical? $\Gamma \models G$?
- Is it horned? $\Gamma \models H$?
- Is it mythical? $\Gamma \models M$?

Recall that

$$\Gamma \models F \qquad \text{iff} \qquad \Gamma, \neg F \ \ \text{UNSAT}$$

SAT solving algorithms

SAT solving algorithms

- There are several techniques and algorithms for SAT solving.
- Usually SAT solvers receive as input a formula in a specific syntactical format.
- So, one has first to transform the input formula to this specific format preserving satisfiability.

Normal forms

SAT solvers usually take input in *conjunctive normal form*.

- A *literal* is a propositional variable or its negation.
    - A literal is *negative* if it is a negated atom, and *positive* otherwise.
- A formula A is in *negation normal form (NNF)*, if the only connectives used in A are $\neg$, $\wedge$ and $\vee$, and negation only appear in literals.
- A *clause* is a disjunction of literals.
- A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses, i.e., it has the form

$$\bigwedge_i \big(\bigvee_j l_{ij}\big)$$

where l_{ij} is the j-th literal in the i-th clause.

Normalization

Transforming a formula F to equivalent formula F' in NNF can be computed by repeatedly replace any subformula that is an instance of the left-hand-side of one of the following equivalences by the corresponding right-hand-side

$$A \to B \equiv \neg A \vee B \qquad \neg\neg A \equiv A$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B \qquad \neg(A \vee B) \equiv \neg A \wedge \neg B$$

This algoritm is linear on the size of the formula.

Normalization

To transform a formula already in NNF into an equivalent CNF, apply recursively the following equivalences (left-to-right):

$$A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \qquad (A \land B) \lor C \equiv (A \lor C) \land (B \lor C)$$

$$A \land \bot \equiv \bot \qquad \bot \land A \equiv \bot \qquad A \land \top \equiv A \qquad \top \land A \equiv A$$

$$A \lor \bot \equiv A \qquad \bot \lor A \equiv A \qquad A \lor \top \equiv \top \qquad \top \lor A \equiv \top$$

This algorithm converts a NNF formula into an equivalent CNF, but its worst case is exponential on the size of the formula.

Example

Compute the CNF of $((P \to Q) \to P) \to P$

The first step is to compute its NNF by transforming implications into disjunctions and pushing negations to proposition symbols:

$$\begin{aligned}((P \to Q) \to P) \to P &\equiv \neg((P \to Q) \to P) \lor P \\ &\equiv \neg(\neg(P \to Q) \lor P) \lor P \\ &\equiv \neg(\neg(\neg P \lor Q) \lor P) \lor P \\ &\equiv \neg((P \land \neg Q) \lor P) \lor P \\ &\equiv (\neg(P \land \neg Q) \land \neg P) \lor P \\ &\equiv ((\neg P \lor Q) \land \neg P) \lor P\end{aligned}$$

To reach a CNF, distributivity is then applied to pull the conjunction outside:

$$((\neg P \lor Q) \land \neg P) \lor P \equiv (\neg P \lor Q \lor P) \land (\neg P \lor P)$$

Worst-case example

Compute the CNF of $(P_1 \land Q_1) \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n)$

$$\begin{aligned} & (P_1 \land Q_1) \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n) \\ \equiv \quad & (P_1 \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n)) \land (Q_1 \lor (P_2 \land Q_2) \lor \ldots \lor (P_n \land Q_n)) \\ \equiv \quad & \ldots \\ \equiv \quad & (P_1 \lor \ldots \lor P_n) \land \\ & (P_1 \lor \ldots \lor P_{n-1} \lor Q_n) \land \\ & (P_1 \lor \ldots \lor P_{n-2} \lor Q_{n-1} \lor P_n) \land \\ & (P_1 \lor \ldots \lor P_{n-2} \lor Q_{n-1} \lor Q_n) \land \\ & \ldots \land \\ & (Q_1 \lor \ldots \lor Q_n)\end{aligned}$$

The original formula has $2n$ literals, while the equivalent CNF has 2^n clauses, each with n literals.
The size of the formula increases exponentially.

Definitional CNF

Equisatisfiability

Two formulas F and F' are *equisatisfiable* when F is satisfiable iff F' is satisfiable.

Any propositional formula can be transformed into a equisatisfiable CNF formula with only linear increase in the size of the formula.
The price to be paid is n new Boolean variables, where n is the number of logical conectives in the formula.
This transformation can be done via *Tseitin's encoding* [Tseitin, 1968].

This tranformation compute what is called the *definitional CNF* of a formula, because they rely on the introduction of new proposition symbols that act as names for subformulas of the original formula.

Tseitin's encoding

Tseitin transformation

1. Introduce a new fresh variable for each compound subformula.
2. Assign new variable to each subformula.
3. Encode local constraints as CNF.
4. Make conjunction of local constraints and the root variable.

- This transformation produces a formula that is equisatisfiable: the result is satisfiable if and only the original formula is satisfiable.
- One can get a satisfying assignment for original formula by projecting the satisfying assignment onto the original variables.

There are various optimizations that can be performed in order to reduce the size of the resulting formula and the number of additional variables.

Tseitin's encoding: an example

Encode $P \to Q \wedge R$

1. $$\overbrace{P \to \underbrace{Q \wedge R}_{A_2}}^{A_1}$$
2. We need to satisfy A_1 together with the following equivalences
   $$A_1 \leftrightarrow (P \to A_2) \qquad\qquad A_2 \leftrightarrow (Q \wedge R)$$
3. These equivalences can be rewritten in CNF as $(A_1 \vee P) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee \neg P \vee A_2)$ and $(\neg A_2 \vee Q) \wedge (\neg A_2 \vee R) \wedge (A_2 \vee \neg Q \vee \neg R)$, respectively.
4. The CNF which is equisatisfiable with $P \to (Q \wedge R)$ is
   $$\begin{array}{ccl} A_1 & \wedge & (A_1 \vee P) \wedge (A_1 \vee \neg A_2) \wedge (\neg A_1 \vee \neg P \vee A_2) \\ & \wedge & (\neg A_2 \vee Q) \wedge (\neg A_2 \vee R) \wedge (A_2 \vee \neg Q \vee \neg R) \end{array}$$

CNFs validity

- The strict shape of CNFs make them particularly suited for checking validity problems.
  - A CNF is a tautology iff all of its clauses are closed (there exists a proposition symbol P, such that both P and $\neg P$ are in the clause).
- However, the applicability of this simple criterion for validity is compromised by the potential exponential growth in the CNF transformation.
- This limitation is overcomed considering instead SAT, with satisfiability preserving CNFs (definitional CNF). Recall that

F is **valid** iff $\neg F$ is **unsatisfiable**

SAT solving algorithms

The majority of modern SAT solvers can be classified into two main categories:

- SAT solvers based on a **stochastic local search**
  - the solver guesses a full assignment, and then, if the formula is evaluated to false under this assignment, starts to flip values of variables according to some heuristic.
- SAT solvers based on the **DPLL framework**
  - optimizations to the Davis-Putnam-Logemann-Loveland algorithm (DPLL) which corresponds to backtrack search through the space of possible variable assignments.

DPLL-based SAT solvers, however, are considered better in most cases.

Stochastic local search

- Local search is incomplete; usually it cannot prove unsatisfiability.
- However, it can be very effective in specific contexts.
- The algorithm:
  - Start with a (random) assignment and repeat a number of times:
    - If not all clauses satisfied, change the value of a variable.
    - If all clauses satisfied, it is done.
  - Repeat (random) selection of assignment a number of times.
- The algorithm terminates when a satisfying assignment is found or when a time bound is elapsed (inconclusive answer).

DPLL framework

- *A CNF is satisfied* by an assignment if all its clauses are satisfied. And *a clause is satisfied* if at least one of its literals is satisfied.
- The ideia is to incrementally construct an assignment compatible with a CNF.
  - An assignment of a formula F is a function mapping F's variables to 1 or 0. We say it is
    - *full* if all of F's variables are assigned,
    - and *partial* otherwise.
- Most current state-of-the-art SAT solvers are based on the *Davis-Putnam-Logemann-Loveland (DPLL)* framework: the tool can be thought of as traversing and backtracking on a binary tree, in which
  - internal nodes represent partial assignments;
  - and each branch represents an assignment to a variable.

State of a clause under an assignment

Given a partial assigment, a clause is

- satisfied if one or more of its literals are satisfied,
- conflicting if all of its literals are assigned but not satisfied.
- unit if it is not satisfied and all but one of its literals are assigned,
- unresolved otherwise.

Let $\mathcal{A}(P) = 1$, $\mathcal{A}(R) = 0$, $\mathcal{A}(Q) = 1$

- $(P \lor X \lor \neg Q)$ is satisfied
- $(\neg P \lor R)$ is conflicting
- $(\neg P \lor \neg Q \lor X)$ is unit
- $(\neg P \lor X \lor A)$ is unsolved

Unit propagation (a.k.a. Boolean Constraint Propagation)

- *Unit clause rule*
  Given a unit clause, its only unassigned literal must be assigned value 1 for the clause to be satisfied.
- *Unit propagation* is the iterated application of the unit clause rule.
- This technique is extensively used.

Consider the partial assignment $\mathcal{A}(P) = 0, \mathcal{A}(Q) = 1$

- Under this assignment
  - $(P \lor \neg R \lor \neg Q)$ is a unit clause.
  - $(\neg Q \lor X \lor R)$ is not a unit clause.
- Performing unit propagation
  - from $(P \lor \neg R \lor \neg Q)$ we have that R must be assigned the value 0, i.e. $\mathcal{A}(R) = 0$.
  - now $(\neg Q \lor X \lor R)$ becames a unit clause, and X must be assigned the value 1, i.e., $\mathcal{A}(X) = 1$.

DPLL algorithm

- Traditionally the DPLL algorithm is presented as a recursive procedure.
- The procedure DPLL is called with the CNF and a partial assignment.
- We will represent a CNF by a set of sets of literals.
- We will represent the partial assignment by a set of literals (P denote that P is set to 1, and $\neg P$ that P is set to 0).
- The algorithm:
  - Progresses by making a decision about a variable and its value.
  - Propagates implications of this decision that are easy to detect, simplifying the clauses.
  - Backtracks in case a conflict is detected in the form of a falsified clause.

CNFs as sets of sets of literals

- Recall that CNFs are formulas with the following shape (each l_{ij} denotes a literal):

$$(l_{11} \vee l_{12} \vee \ldots \vee l_{1k}) \wedge \ldots \wedge (l_{n1} \vee l_{n2} \vee \ldots \vee l_{nj})$$

- Associativity, commutativity and idempotence of both disjunction and conjunction allow us to treat each CNF as a set of sets of literals S

$$S = \{\{l_{11}, l_{12}, \ldots, l_{1k}\}, \ldots, \{l_{n1}, l_{n2}, \ldots, l_{nj}\}\}$$

- An empty inner set will be identified with $\bot$, and an empty outer set with $\top$. Therefore,
  - if $\{\} \in S$, then S is equivalent to $\bot$;
  - if $S = \{\}$, then S is $\top$.

Simplification of a clause under an assignment

The *opposite* of a literal l, written $-l$, is defined by

$$-l = \begin{cases} \neg P & \text{, if } l = P \\ P & \text{, if } l = \neg P \end{cases}$$

When we set a literal l to be true,

- any clause that has the literal l is now guaranteed to be satisfied, so we throw it away for the next part of the search;
- any clause that had the literal $-l$, on the other hand, must rely on one of the other literals in the clause, hence we throw out the literal $-l$ before going forward.

Simplification of S assuming l holds

$$S|_l = \{c \backslash \{-l\} \mid c \in S \text{ and } l \notin c\}$$

Simplification of a clause under an assignment

If a CNF S contains a clause that consists of a single literal (a unit clause), we know for certain that the literal must be set to true and S can be simplified.

One should apply this rule while it is possible and worthwhile.

```
UNIT_PROPAGATE $(S, \mathcal{A})$ {
    while $\{\} \notin S$ and $S$ has a unit clause $l$ do {
        $S \leftarrow S|_l$ ;
        $\mathcal{A} \leftarrow \mathcal{A} \cup \{l\}$
    }
}
```

DPLL algorithm

DPLL is called with a CNF S and a partial assignment $\mathcal{A}$ (initially $\emptyset$).

```
DPLL($S, \mathcal{A}$) {
    UNIT_PROPAGATE($S, \mathcal{A}$);
    if $S = \{\}$ then return SAT;
    else if $\{\} \in S$ then return UNSAT;
    else { $l \leftarrow$ a literal of $S$ ;
        if DPLL ($S|_l, \mathcal{A} \cup \{l\}$) = SAT then return SAT;
        else return DPLL ($S|_{-l}, \mathcal{A} \cup \{-l\}$)
    }
}
```

- DPLL complete algorithm for SAT.
- Unsatisfiability of the complete formula can only be detected after exhaustive search.

DPLL algorithm

Is $(\neg P \vee Q) \wedge (\neg P \vee R) \wedge (Q \vee R) \wedge (\neg Q \vee \neg R) \wedge (P \vee \neg R \vee Q)$ satisfiable?

		S	$\mathcal{A}$
DPLL		$\{\{\neg P, Q\}, \{\neg P, R\}, \{Q, R\}, \{\neg Q, \neg R\}, \{P, \neg R, Q\}\}$	$\{\}$
UNIT_PROPAGATE			
		$\{\{\neg P, Q\}, \{\neg P, R\}, \{Q, R\}, \{\neg Q, \neg R\}, \{P, \neg R, Q\}\}$	$\{\}$
choose $l = P$			
DPLL	$S\|_l$	$\{\{Q\}, \{R\}, \{Q, R\}, \{\neg Q, \neg R\}\}$	$\{P\}$
UNIT_PROPAGATE			
		$\{\{\}\}$	$\{P, Q, R\}$
$-l = \neg P$			
DPLL	$S\|_{-l}$	$\{\{Q, R\}, \{\neg Q, \neg R\}, \{\neg R, Q\}\}$	$\{\neg P\}$
UNIT_PROPAGATE			
		$\{\{Q, R\}, \{\neg Q, \neg R\}, \{\neg R, Q\}\}$	$\{\neg P\}$
choose $l = Q$			
DPLL	$S\|_l$	$\{\{\neg R\}\}$	$\{\neg P, Q\}$
UNIT_PROPAGATE			
		$\{\}$	$\{\neg P, Q, \neg R\}$
		SAT	

DPLL framework: heuristics & optimizations

Many different techniques are applied to achieve efficiency in DPLL-based SAT solvers.

- Decision heuristic: a very important feature in SAT solving is the strategy by which the literals are chosen.
- Look-ahead: exploit information about the remaining search space.
    - unit propagation
    - pure literal rule
- Look-back: exploit information about search which has already taken place.
    - non-chronological backtracking (a.k.a. backjumping)
    - clause learning
- Other techniques:
    - preprocessing (detection of subsumed clauses, simplification, ...)
    - (random) restart (restarting the solver when it seams to be is a hopeless branch of the search tree)

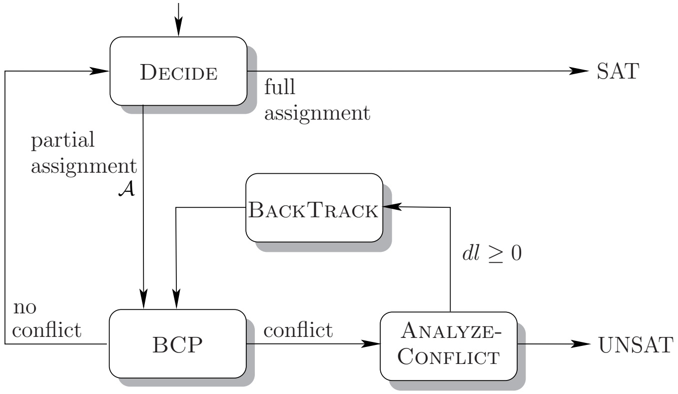
DPLL-based iterative algorithm [Marques-Silva&Sakallah,1996]

At each step:

- **Decide** on the assignment of a variable (which is called the *decision variable*, and it will have a *decision level* associated with it).
- **Deduce** the consequences of the decision made. (Variables assigned will have the same decision level as the decision variable.)
    - If all the clauses are satisfied, then the instance is satisfiable.
    - If there exists a conflicting clause, then **analyze** the conflit and determine the decision level to backtrack. (The solver may perform some analysis and record some information from the current conflict in order to prune the search space for the future.)
        - Decision level < 0 indicates that the formula is unsatisfiable.
    - Otherwise, proceed with another decision.

Different DPLL-based modern solvers differ mainly in the detailed implementation of each of these functions.

DPLL-based iterative algorithm



Conflict analysis and learning

- *Non-chronological backtracking* does not necessarily flip the last assignment and can backtrack to an earlier decision level.
- The process of adding conflict clauses is generally referred to as *learning*.
- The conflict clauses record the reasons deduced from the conflict to avoid making the same mistake in the future search. For that *implication graphs* are used.
- *Conflict-driven backtracking* uses the conflict clauses learned to determine the actual reasons for the conflict and the decision level to backtrack in order to prevent the repetition of the same conflict.

Conflict analysis and learning

Consider, for example, a formula ψ that contains the following set of clauses, among others:

$$\begin{array}{lcl}
c_1 & = & (\neg x_1 \vee x_2) \\
c_2 & = & (\neg x_1 \vee x_3 \vee x_5) \\
c_3 & = & (\neg x_2 \vee x_4) \\
c_4 & = & (\neg x_3 \vee \neg x_4) \\
c_5 & = & (x_1 \vee x_5 \vee \neg x_2) \\
c_6 & = & (x_2 \vee x_3) \\
c_7 & = & (x_2 \vee \neg x_3) \\
\ldots & &
\end{array}$$

and assume that at decidsion level 3 the decision was $\mathcal{A}(x_5) = 0$.

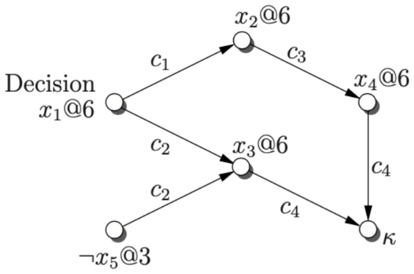
Conflict analysis and learning

At level 3, decide $x_5 = 0$, denoted $\neg x_5@3$.

Clauses of ψ

$$\begin{array}{lcl}
c_1 & = & (\neg x_1 \vee x_2) \\
c_2 & = & (\neg x_1 \vee x_3 \vee x_5) \\
c_3 & = & (\neg x_2 \vee x_4) \\
c_4 & = & (\neg x_3 \vee \neg x_4) \\
c_5 & = & (x_1 \vee x_5 \vee \neg x_2) \\
c_6 & = & (x_2 \vee x_3) \\
c_7 & = & (x_2 \vee \neg x_3) \\
\ldots & &
\end{array}$$

Implication graph



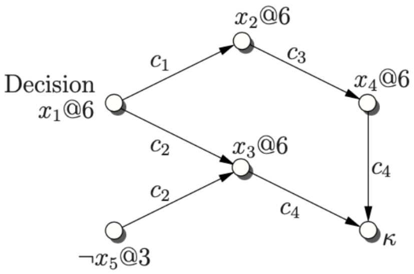
Conflict analysis and learning

At level 6, decide $x_1 = 1$, denoted $x_1@6$.

Clauses of ψ

$$\begin{aligned} c_1 &= (\neg x_1 \vee x_2) \\ c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\ c_3 &= (\neg x_2 \vee x_4) \\ c_4 &= (\neg x_3 \vee \neg x_4) \\ c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\ c_6 &= (x_2 \vee x_3) \\ c_7 &= (x_2 \vee \neg x_3) \\ &\dots \end{aligned}$$

Implication graph



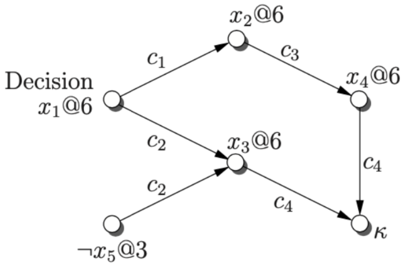
Conflict analysis and learning

At level 6, BCP: $x_2 = 1$, denoted $x_2@6$.

Clauses of ψ

$$\begin{aligned} c_1 &= (\neg x_1 \vee x_2) \\ c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\ c_3 &= (\neg x_2 \vee x_4) \\ c_4 &= (\neg x_3 \vee \neg x_4) \\ c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\ c_6 &= (x_2 \vee x_3) \\ c_7 &= (x_2 \vee \neg x_3) \\ &\dots \end{aligned}$$

Implication graph



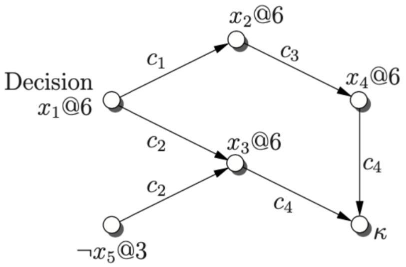
Conflict analysis and learning

At level 6, BCP: $x_3 = 1$ denoted $x_3@6$.

Clauses of ψ

$$\begin{aligned} c_1 &= (\neg x_1 \vee x_2) \\ c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\ c_3 &= (\neg x_2 \vee x_4) \\ c_4 &= (\neg x_3 \vee \neg x_4) \\ c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\ c_6 &= (x_2 \vee x_3) \\ c_7 &= (x_2 \vee \neg x_3) \\ &\dots \end{aligned}$$

Implication graph



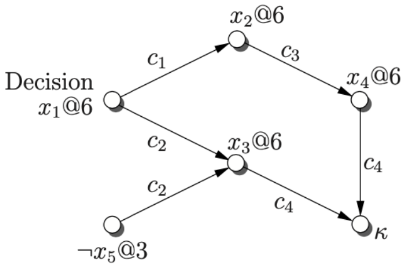
Conflict analysis and learning

At level 6, BCP: $x_4 = 1$, denoted $x_4@6$.

Clauses of ψ

$$\begin{aligned} c_1 &= (\neg x_1 \vee x_2) \\ c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\ c_3 &= (\neg x_2 \vee x_4) \\ c_4 &= (\neg x_3 \vee \neg x_4) \\ c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\ c_6 &= (x_2 \vee x_3) \\ c_7 &= (x_2 \vee \neg x_3) \\ &\dots \end{aligned}$$

Implication graph



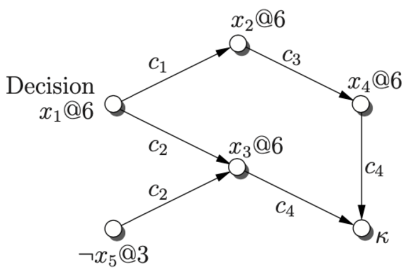
Conflict analysis and learning

At level 6, BCP: $x_4 = 1$, denoted $x_4@6$.

Clauses of ψ

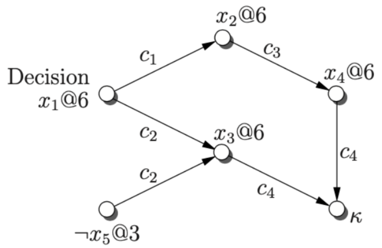
$$
\begin{aligned}
c_1 &= (\neg x_1 \lor x_2)\\
c_2 &= (\neg x_1 \lor x_3 \lor x_5)\\
c_3 &= (\neg x_2 \lor x_4)\\
c_4 &= (\neg x_3 \lor \neg x_4)\ \textbf{conflit}\ (k)\\
c_5 &= (x_1 \lor x_5 \lor \neg x_2)\\
c_6 &= (x_2 \lor x_3)\\
c_7 &= (x_2 \lor \neg x_3)\\
&\dots
\end{aligned}
$$

Implication graph



Clause learned: $(x_5 \lor \neg x_1)$

Conflict analysis and learning



We have $\neg x_5 \land x_1 \to \neg\psi$, so

$$\psi \to x_5 \lor \neg x_1$$

Therefore, we can safely add to our formula the clause $(x_5 \lor \neg x_1)$.

The clause learned, $(x_5 \lor \neg x_1)$, does not change the result, but it prunes the search space.

Conflict-driven backtracking

After detecting the conflict and adding the clause learned the solver determines which decision level to backtrack to according to the conflict-driven backtracking strategy.

For instance:

- The backtracking level is set to the second most recent decision level in the clause learned, while erasing all decisions and implications made after that level.
- In the case of $(x_5 \lor \neg x_1)$, the solver backtracks to decision level 3, and erases all assignments from decision level 4 onwards, including the assignments to x_1, x_2, x_3 and x_4.

Conflict-Driven Clause Learning (CDCL) solvers

- DPLL framework.
- New clauses are learnt from conflicts.
- Structure (implication graphs) of conflicts exploited.
- Backtracking can be non-chronological.
- Efficient data structures (compact and reduced maintenance overhead).
- Backtrack search is periodically restarted.

- Can deal with hundreds of thousand variables and tens of million clauses!

Modern SAT solvers

- In the last two decades, satisfiability procedures have undergone dramatic improvements in efficiency and expressiveness. Breakthrough systems like GRASP (1996), SATO (1997), Chaff (2001) and MiniSAT (2003) have introduced several enhancements to the efficiency of DPLL-based SAT solving.
- New SAT solvers are introduced every year.
- The satisfiability library SAT Live![1] is an online resource that proposes, as a standard, a unified notation and a collection of benchmarks for performance evaluation and comparison of tools.

---

[1] http://www.satlive.org

DIMACS CNF format

- DIMACS CNF format is a standard format for CNF used by most SAT solvers.
- Plain text file with following structure:

```
c <comments>
...
p cnf <num.of variables> <num.of clauses>
<clause> 0
<clause> 0
...
```

  - Every number 1, 2, . . . corresponds to a variable (variable names have to be mapped to numbers).
  - A negative number denote the negation of the corresponding variable.
  - Every clause is a list of numbers, separated by spaces. (One or more lines per clause).

DIMACS CNF format

Example

$$A_1 \wedge (A_1 \vee P) \wedge (\neg A_1 \vee \neg P \vee A_2) \wedge (A_1 \vee \neg A_2)$$

- We have 3 variables and 4 clauses.
- CNF file:

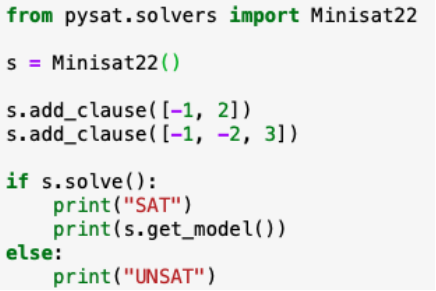
```
p cnf 3 4
1 0
1 3 0
-1 -3 2 0
1 -2 0
```

Minisat demo

```
$ cat example.cnf
c A1 = 1
c A2 = 2
c P = 3

p cnf 3 4
1 0
1 3 0
-1 -3 2 0
1 -2 0

$ minisat example.cnf OUT
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:             3                                         |
|  Number of clauses:               1                                         |
|  Parse time:                   0.00 s                                       |
|  Eliminated clauses:           0.00 Mb                                      |
|  Simplification time:          0.00 s                                       |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
===============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 1              (0.00 % random) (490 /sec)
propagations          : 1              (490 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 0.16 MB
CPU time              : 0.002041 s

SATISFIABLE

$ cat OUT
SAT
1 -2 -3 0
```

SAT solver API

- Several SAT solvers have API's for different programming languages that allow an incremental use of the solver.
- For instance, PySAT[2] is a Python toolkit which provides a simple and unified interface to a number of state-of-art SAT solvers, enabling to prototype with SAT oracles in Python while exploiting incrementally the power of the original low-level implementations of modern SAT solvers.



```
from pysat.solvers import Minisat22

s = Minisat22()

s.add_clause([-1, 2])
s.add_clause([-1, -2, 3])

if s.solve():
    print("SAT")
    print(s.get_model())
else:
    print("UNSAT")
```

---

[2] https://pysathq.github.io

Variations on the Boolean Satisfiability Problem

- So far, we considered the basic Boolean satisfiability problem:

  Given a propositional formula F, is F satisfiable?

- Some common variants of Boolean SAT:
  - MaxSAT problem: Given formula F in CNF, find assignment maximizing the number of satisfied clauses of F.
  - Partial MaxSAT problem: Given CNF formula F where each clause is marked as hard or soft, find an assignment that satisfies all hard clauses and maximizes the number satisfied soft clauses.
  - Partial Weighted MaxSAT problem: Find assignment maximizing the sum of weights of satisfied soft clauses

---

Modeling with PL

SAT example: Schedule a meeting

When can the meeting take place?

– Anne cannot meet on Friday.
– Peter can only meet either on Monday, Wednesday or Thursday.
– Mike cannot meet neither on Tuesday nor on Thursday.

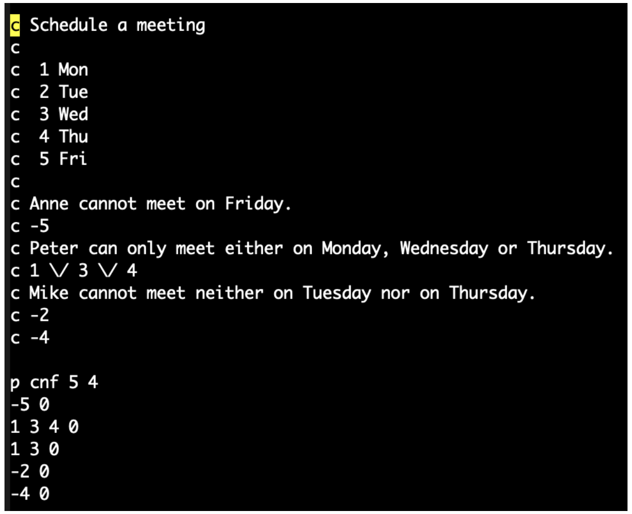
- Create 5 variables to represent the days of week.
- The constraints can be encoded into the following proposition:

$$\neg \text{Fri} \land (\text{Mon} \lor \text{Wed} \lor \text{Thu}) \land (\neg \text{Tue} \land \neg \text{Thu})$$

- How can we use a SAT solver to explore the possible solutions to this problem?

SAT example: Schedule a meeting

First, encode de problem in DIMACS CNF format.



```
c Schedule a meeting
c
c  1 Mon
c  2 Tue
c  3 Wed
c  4 Thu
c  5 Fri
c
c Anne cannot meet on Friday.
c -5
c Peter can only meet either on Monday, Wednesday or Thursday.
c 1 \/ 3 \/ 4
c Mike cannot meet neither on Tuesday nor on Thursday.
c -2
c -4

p cnf 5 4
-5 0
1 3 4 0
1 3 0
-2 0
-4 0
```

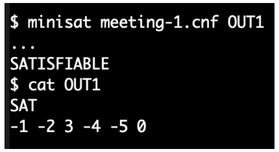
SAT example: Schedule a meeting

- Check SAT and see the model produced.

```
$ minisat meeting.cnf OUT
...
SATISFIABLE
$ cat OUT
SAT
1 -2 -3 -4 -5 0
```

The meeting can take place on Monday.

- Add a clausule to exclude Monday (-1) and check SAT again.



```
$ minisat meeting-1.cnf OUT1
...
SATISFIABLE
$ cat OUT1
SAT
-1 -2 3 -4 -5 0
```

The meeting can take place on Wednesday.

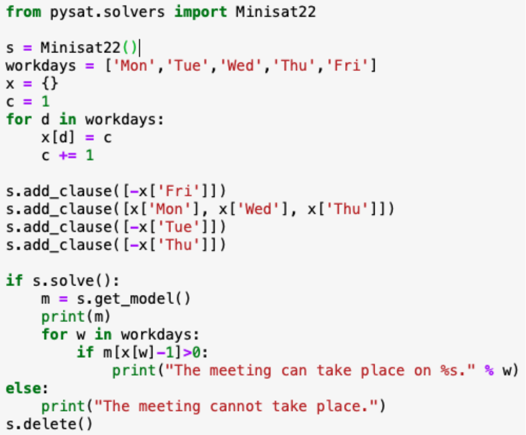
- Add a clausule to exclude Wednesday (-3) and check SAT again.

```
$ minisat meeting-2.cnf OUT
...
UNSATISFIABLE
```

No more solutions.

SAT example: Schedule a meeting

Using the PySAT toolkit.



```python
from pysat.solvers import Minisat22

s = Minisat22()
workdays = ['Mon','Tue','Wed','Thu','Fri']
x = {}
c = 1
for d in workdays:
    x[d] = c
    c += 1

s.add_clause([-x['Fri']])
s.add_clause([x['Mon'], x['Wed'], x['Thu']])
s.add_clause([-x['Tue']])
s.add_clause([-x['Thu']])

if s.solve():
    m = s.get_model()
    print(m)
    for w in workdays:
        if m[x[w]-1]>0:
            print("The meeting can take place on %s." % w)
else:
    print("The meeting cannot take place.")
s.delete()
```

Change the code to print all possible solutions to the problem.

Modeling with PL

Equivalence checking of if-then-else chains

Original C code

```
if(!a && !b) h();
else if(!a) g();
else f();
```

Optimized C code

```
if(a) f();
else if(b) g();
else h();
```

Are these two programs equivalent?

1. Model the variables `a` and `b` and the procedures that are called using the Boolean variables a, b, f, g, and h.
2. Compile if-then-else chains into Boolean formulae

$$\text{compile}(\textbf{if } x \textbf{ then } y \textbf{ else } z) \equiv (x \land y) \lor (\neg x \land z)$$

3. Check the validity of the following formula

$$\text{compile}(\textit{original}) \leftrightarrow \text{compile}(\textit{optimized})$$

by reformulating it as a SAT problem.

SAT example: Graph coloring

Graph coloring

Can one assign one of K colors to each of the vertices of graph $G = (V, E)$ such that adjacent vertices are assigned different colors?

- Create $|V| \times K$ variables:
  - $x_{ij} = 1$ iff vertex i is assigned color j;
  - $x_{ij} = 0$ otherwise.
- For each edge (u, v), require different assigned colors to u and v:

$$\text{for each } \ 1 \leq j \leq K, \ \ (x_{uj} \rightarrow \neg x_{vj})$$

- ...

SAT example: Graph coloring

- Each vertex is assigned exactly one color.
  - *At least* one color to each vertex:

$$\text{for each } 1 \leq i \leq |V|, \quad \bigvee_{j=1}^{K} x_{ij}$$

  - *At most* one color to each vertex:

$$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K} (x_{ia} \rightarrow \bigwedge_{b=1, b \neq a}^{K} \neg x_{ib})$$

since $\vee$ and $\wedge$ are commutative and idempotent, a better encoding is

$$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K-1} (x_{ia} \rightarrow \bigwedge_{b=a+1}^{K} \neg x_{ib})$$

or equivalently,

$$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K-1} \bigwedge_{b=a+1}^{K} (\neg x_{ia} \vee \neg x_{ib})$$

SAT example: Graph coloring

Let's make a Python program to solve this problem!