Specification and Proof

There are 3 categories of proof methods:

- Proofs not computationaly assisted:proofs made by hand and can be informally described.

- Tools that allow the formal definition of the proofs.

- Computer assisted proofs.

Logics:

- propositional logic, first-order logic, high-order logic

- classic logic versus intuicionistic logic

- modal and temporal logics

Proof Tools

- Automatic proof tools: use a decidable logic fragment

  - ELAN: first-order rewrite
  - ACS2: first-order logic
  - SMT Solvers (Satisfability Modulo Theory): Yices, CVC3, **Z3**, Alt-Ergo, Simplify: combined decision algorithms for integers, reals, pointers, bit-vectors, "arrays", etc.
  - Allow reason about infinite sets

- Interactive proof tools:: allow more expressive logics, potentially nondecidable (Coq, Mizar, Isabelle, Agda, Lean, etc)

  - Combine two capacities: proof check and assisted proof construction
  - Proofs are build interactively using tactics: case, elim, change, rewrite, simpl, discriminate, injection, induction.

- The approaches are not strict: interactive systems use tactics that are automatic solvers and automatic systems allow some interaction

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 (Cook's theorem)

  – There are no known polynomial time algorithms for SAT.

- SETH (*Strong Exponential Time Hipothesis*) any algorithm to solve CNF SAT in the worst-case runs in time 2^n, being n the number of variables.

CNF and Clauses

Usually SAT solvers deal with formulas in conjunctive normal form (CNF)

- literal: propositional variable or its negation: x, $\neg x$, y , $\neg y$

- clause: disjuntion of literals. $(x_1 \vee \neg x_2 \vee x_3)$

- conjunctive normal form (CNF): conjuction of clauses.

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$$

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.

- There are small instances for which is difficult to find a solution

- A huge number of practical applications.

- 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 to deciding satisfiability of formulas in logic.

Efficiency of SAT solvers since 2000

The x-axis represents CPU time, and the y-axis represents the number of solved instances.
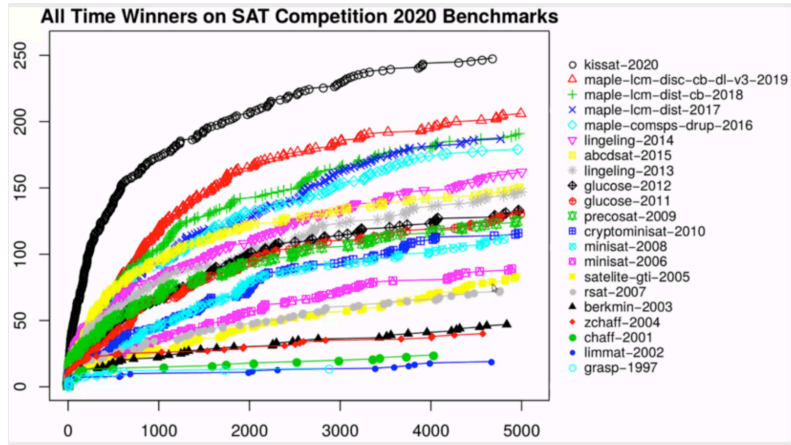
SAT solving algorithms

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

  - SAT solvers based on the *Davis-Putman-Loveland-Logemann* (DPLL) framework:
    - ∗ tools traverse and backtrack in a binary tree which nodes represent partial assigments and leaves total assigments (corresponds to backtrack search through the space of possible variable assignments).
    - ∗ Optimizations to DPLL
    - ∗ Most SAT solvers use the strategy CDCL (*conflit-driven clause learning*)
  - Stochastic based 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.

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

- DPLL-based solvers are complete.

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 are 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).

Syntaxe of Propositional logic

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

- Propositional variables: $\mathcal{V}_{Prop} = \{p, q, r, \ldots, p_1, \ldots\}$

- Logical connectives: true, false, $\wedge$, $\vee$, $\neg$, $\implies$, $\iff$

- Auxiliary symbols: (and)

The set of formulas Form of propositional logic is given by the abstract syntax $(\phi, \psi, \theta, \ldots)$

$$\varphi, \psi ::= p \in \mathcal{V}_{Prop} \mid \text{true} \mid \text{false} \mid (\neg\varphi) \mid (\varphi \wedge \psi) \mid (\varphi \vee \psi) \mid (\varphi \implies \psi) \mid (\varphi \iff \psi)$$

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 $v : \mathcal{V}_{Prop} \to \{0, 1\}$, that assigns to every propositional variable a truth value.

- An assignment v naturally extends to all formulas, $v : Form \to \{0, 1\}$.

- The truth value of a formula is computed using truth tables:

φ	ψ	$\neg\varphi$	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \implies \psi$	$\varphi \iff \psi$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

4

Satisfability

Let v be an assignment, the relation $\models_v$ is inductively defined in the structure of φ by:

1. $\models_v p$ if $v(p) = 1$;

2. $\models_v \mathsf{true}$

3. $\not\models_v \mathsf{false}$

4. $\models_v \neg\varphi$ if $\not\models_v \varphi$;

5. $\models_v \varphi \wedge \psi$ if $\models_v \varphi$ and $\models_v \psi$;

6. $\models_v \varphi \vee \psi$ if $\models_v \varphi$ or $\models_v \psi$;

7. $\models_v \varphi \implies \psi$ if $\not\models_v \varphi$ or $\models_v \psi$;

8. $\models_v \varphi \iff \psi$ if $\models_v \varphi \implies \psi$ and $\models_v \psi \implies \varphi$

We say that v *satisfies* φ if $\models_v \varphi$.

Negative normal form (NNF)

Transforming a formula φ to equivalent formula ψ 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

$$\varphi \implies \psi \equiv \neg\varphi \vee \psi \qquad\qquad \neg\neg\varphi \equiv \varphi$$
$$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi \qquad\qquad \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$$

This algorithm is linear on the size of the formula.

CNF

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

$$\varphi \vee (\psi \wedge \gamma) \equiv (\varphi \vee \psi) \wedge (\varphi \vee \gamma) \qquad (\varphi \wedge \psi) \vee \gamma \equiv (\varphi \vee \gamma) \wedge (\psi \vee \gamma)$$

and the identities.

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

Worst-case example

Compute the CNF of $(\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2) \vee \cdots (\varphi_n \wedge \psi_n)$

$$(\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2) \vee \cdots (\varphi_n \wedge \psi_n)$$
$$\equiv (\varphi_1 \vee (\varphi_2 \wedge \psi_2) \vee \cdots (\varphi_n \wedge \psi_n)) \wedge (\psi_1 \vee (\varphi_2 \wedge \psi_2) \vee \cdots (\varphi_n \wedge \psi_n))$$
$$\equiv \cdots$$
$$\equiv ((\varphi_1 \vee \cdots \vee \varphi_n) \wedge$$
$$(\varphi_1 \vee \cdots \vee \varphi_{n-1} \vee \psi_n) \wedge$$
$$(\varphi_1 \vee \cdots \vee \varphi_{n-2} \vee \psi_{n-1} \vee \varphi_n) \wedge$$
$$(\varphi_1 \vee \cdots \vee \varphi_{n-2} \vee \psi_{n-1} \vee \psi_n) \wedge$$
$$\cdots \wedge$$
$$(\psi_1 \vee \cdots \vee \psi_n)$$

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 φ and ψ are *equisatisfiable* when φ is satisfiable iff ψ 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 Boolean variables, where n is the number of logical connectives in the formula.

- This transformation can be done via Tseitin's encoding

- This transformation 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 transformation

1. Introduce a new fresh variable for each compound subformula.

2. Assign new variable to each subformula (logic conective)

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.

Example 1. Encode $p \implies q \wedge r$

1. We consider a_1 and a_2 new variables.

2. We need to satisfy a_1 together with the following equivalences $a_1 \Leftrightarrow (p \implies a_2)$ and $a_2 \Leftrightarrow (q \wedge r)$

3. These equivalences can be rewritten in CNF

$$(a_1 \vee p) \wedge (a_1 \vee \neg a_2) \wedge (\neg a_1 \vee \neg p \vee a_2)$$
$$(\neg a_2 \vee q) \wedge (\neg a_2 \vee r) \wedge (a_2 \vee \neg q \vee \neg r)$$

4. The CNF which is equisatisfiable with $p \implies q \wedge r$ is

$$a_1 \wedge (a_1 \vee p) \wedge (a_1 \vee \neg a_2) \wedge (\neg a_1 \vee \neg p \vee a_2)$$
$$\wedge (\neg a_2 \vee q) \wedge (\neg a_2 \vee r) \wedge (a_2 \vee \neg q \vee \neg r)$$

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 φ is a function mapping φ's variables to 1 or 0.
  - We say it is
    * full if all of φ '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.

- Initial version from 1960 but now solvers are much more efficient. See

  – https://www.satlive.org
  – Competition https://www.satcompetition.org/.

State of a clause under an assignment

Given a partial assignment, a clause is

- *satisfied* if one or more of its literals are satisfied,

- *confliting* 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.

Example 2. Let $v(p) = 1$, $v(r) = 0$, and $v(q) = 1$

- $(p \lor x \lor \neg q)$ is satisfied

- $(\neg p \lor r)$ is confliting

- $(\neg p \lor \neg q \lor x)$ 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 $v(p) = 0$, and $v(q) = 1$

- 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. $v(r) = 0$.
  – now $(\neg q \lor x \lor r)$ becomes a unit clause, and x must be assigned the value 1, i.e., $v(x) = 1$.

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.

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

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

- 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 false, and an empty outer set with true. Therefore,

  - if $\{\} \in S$, then S is equivalent to false;
  - if $S = \{\}$, then S is true.

Simplification of a clause under an assignment

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

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

When we set a literal to be true,

- any clause that has the literal l is now garantee to be satisfied, so we throw it away for next part of the search

- any clause that had the literal $-l$ must rely on the other literals in the clause, hence we throw the literal $-l$ before going forward

Simplification of S assuming l holds

$$S|_l = \{c \setminus \{-l\} \mid c \in S \land l \notin c\}$$

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.

> **procedure** UNIT-PROPAGATION(S, v)
>   **while** $\{\} \notin S$ and S has a unit clause l **do**
> $S \to S|_l$
> $v \to v \cup \{l\}$

DPLL algorithm (basic)

> **function** DPLL(S, v)
>   UNIT-PROPAGATION(S, v)
>   **if** $S = \{\}$ **then**
>   **return** SAT;
>   **else if** $\{\} \in S$ **then**
>   **return** UNSAT;
>   **else**
> $l \to$ a literal of S;
>   **if** DPLL$(S|_l, v \cup \{l\})$ =SAT **then**
>   **return** SAT;
>   **else**
>   DPLL$(S|_{-l}, v \cup \{-l\})$;

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

Example 3. Apply the algorithm to this set of clauses:

$$
\begin{aligned}
c_1 &= x_1 \\
c_2 &= \neg x_1 \lor x_2 \\
c_3 &= \neg x_1 \lor x_3 \\
c_4 &= \neg x_2 \lor \neg x_3 \lor x_4
\end{aligned}
$$

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 (CDCL)

- Other techniques:

  - preprocessing (detection of subsumed clauses, simplification, ...)
  - (random) restart (restarting the solver when it seams to be in a hopeless branch of the search tree)

DPLL-CDCL Solvers

- The process can be seen as a search in a binary (decision) tree in such way that each decision of a variable value is associated to a *level* that corresponds to the tree depth.

- The assignments that are implied by that decision has the same level of the decision.

- The assignments of *unary clauses* (literals) have level 0.

- At each level:

  - Decide the value of a variable (assignment)
  - propagation of that decision (implications)
  - backtracking in case of conflict

State of a Clause under an assignment (again)

Given a partial assignment, the state of a clause can be:

- *satisfied if at least one literal is satisfied.*

- *conflict* if all literals have an assignment but are not satisfied.

- *unit* if it is not satisfied but all except one literal are assigned . If C is an unit and l the unassigned literal then C is the antecedent clause of l:

$$C = \text{ANTECEDENT}(l)$$

- *unresolved*, otherwise

Example 4. The clause $C := \neg x_1 \vee \neg x_4 \vee x_3$ with assignment $\{x_1 \leftarrow 1, x_4 \leftarrow 1\}$ imply the assignment of x_3 and ANTECEDENT$(x_3) = C$.

DPLL-CDCLAlgorithm

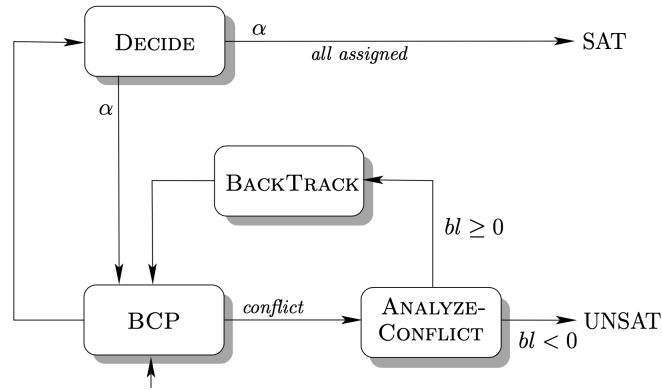**Input:** proposicional formula $\mathcal{B}$ in CNF
Output: SAT or UNSAT
 function DPLL-CDCL$(\mathcal{B})$
   while TRUE **do**
     while BCP() = "conflict" **do**
       backtrack-level$\leftarrow$ ANALIZE-CONFLICT();
       if backtrack-level< 0 **then**
         return UNSAT;
       else
         BACKTRACK(backtrack-level)
     if not DECIDE() **then**
       return SAT

DPLL-CDCL based iterative algorithm

where bl is the backtracking level and α an assignment.

Algoritmo DPLL-CDCL - Components

Name:	DECIDE()
Output:	false iff there are no more variables to assign
Description:	Chooses an unassigned variable and a truth value for it (there are many heuristics)

Name:	BCP()
Output:	"conflict" iff a conflict is encountered
Description:	Repeated application of the unit clause rule until either a conflict is encountered or there are no more implications. (*Boolean Constraint Propagation*)

DPLL-CDCL - Components

Name:	ANALIZE-CONFLICT()
Output:	-1 if a conflict at decision level 0 is detected (which implies that the formula is unsatisfiable). Otherwise, a decision level which the solver should backtrack to.

Name:	BACKTRACK(bl)
Description:	Sets the current decision level to bl and erases assignments at desicion levels larger than bl

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.
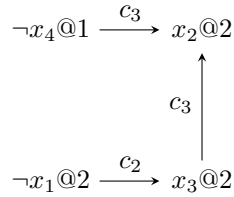
Implication graph

- labeled acyclic digraph $G = (V, E)$ where

- V represents the current partial assignment,

  - and each node represents a literal and is labeled with $l_i@bl$ where l_i is the literal

- and bl the decision level at which it entered the partial assignment.
  - If it is $x_i@bl$ this means that the value of x_i is 1; otherwise it is $\neg x_i@bl$ and its value is 0.
  - and if it is $\neg x_i@bl$ then the value is 0.

- $E = \{(v_i, v_j) \mid v_i, v_j \in V \wedge \neg v_i \in \textsc{antecedent}(v_j)\}$ and $(v_i, v_j) \in E$ is labeled by $\textsc{antecedent}(v_j) = C$, i.e.. $v_i \xrightarrow{C} v_j$.

- G can have a single *conflict node* labeled by κ and incoming edges $\{(v, \kappa) \mid \neg v \in C\}$, labeled with C for some conflicting clause C. In this case, G is a *conflict graph*.

Example 5. Consider

$$c_1 \wedge c_2 \wedge c_3 = (x_1 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_3 \vee x_2 \vee x_4)$$

There are no unitary clauses so one needs to decide an assignment. For instance $\neg x_4@1$ and also $\neg x_1@2$. Then $\text{BCP}()$ determines by implication $x_3@2$ and also $x_2@2$. Compute the implication graph. What can be conclude?

$$\neg x_4@1 \xrightarrow{c_3} x_2@2$$

$$c_3 \uparrow$$

$$\neg x_1@2 \xrightarrow{c_2} x_3@2$$

The formula is SAT.

Implication graph

- Root nodes are decisions and internal nodes correspond to implications of BCP ($(v_i, v_j) \in E$ then $v_i \implies v_j$).

- A conflict node where incoming edges are labeled by C mean that all literals in C are not satisfied.

- $\textsc{Backtrack}()$ diminish the size of G and $\text{BCP}()$ makes G grow.

- The digraph depends on the implications order, thus is not unique

Consider the following set of clauses ψ

$$
\begin{array}{llll}
c_1 & = & (\neg x_1 \vee x_2) & c_5 & = & (x_1 \vee x_5 \vee \neg x_2) \\
c_2 & = & (\neg x_1 \vee x_3 \vee x_5) & c_6 & = & (x_2 \vee x_3) \\
c_3 & = & (\neg x_2 \vee x_4) & c_7 & = & (x_2 \vee \neg x_3) \\
c_4 & = & (\neg x_3 \vee \neg x_4) & c_8 & = & (x_6 \vee \neg x_5)
\end{array}
$$

we have the following (fragment of) implication digraph if $\neg x_5@3$ and if on level 6 we decide $x_1 = 1$, denoted by $x_1@6$.
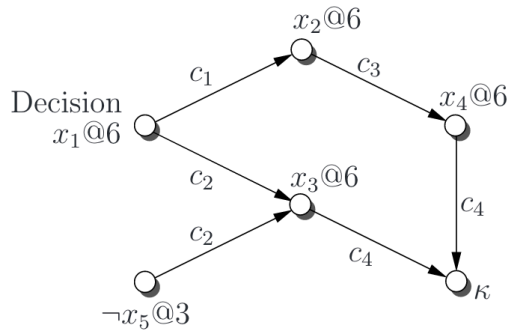
$$
\begin{aligned}
c_1 &= (\neg x_1 \vee x_2) \\
c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\
c_3 &= (\neg x_2 \vee x_4) \\
c_4 &= (\neg x_3 \vee \neg x_4) \\
c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\
c_6 &= (x_2 \vee x_3) \\
c_7 &= (x_2 \vee \neg x_3) \\
c_8 &= (x_6 \vee \neg x_5)
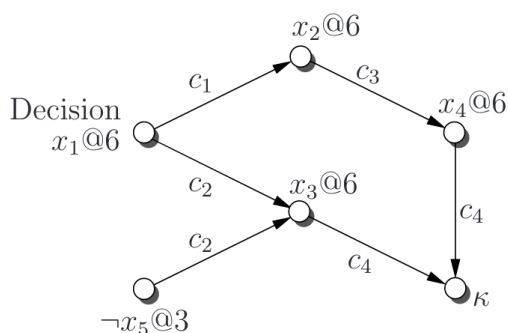\end{aligned}
$$
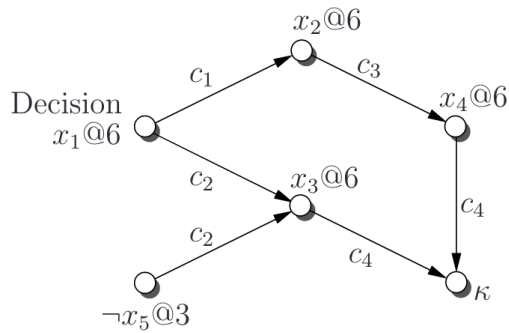
Then: c_5 satisfied, c_1, c_2 unit:

At level 6 by BCP(): $x_2 = 1$ ($x_2@6$)

$$
\begin{aligned}
c_1 &= (\neg x_1 \vee x_2) \\
c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\
c_3 &= (\neg x_2 \vee x_4) \\
c_4 &= (\neg x_3 \vee \neg x_4) \\
c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\
c_6 &= (x_2 \vee x_3) \\
c_7 &= (x_2 \vee \neg x_3) \\
c_8 &= (x_6 \vee \neg x_5)
\end{aligned}
$$

and also $x_3 = 1$ (x_3@6)

$$
\begin{aligned}
c_1 &= (\neg x_1 \vee x_2) \\
c_2 &= (\neg x_1 \vee x_3 \vee x_5) \\
c_3 &= (\neg x_2 \vee x_4) \\
c_4 &= (\neg x_3 \vee \neg x_4) \\
c_5 &= (x_1 \vee x_5 \vee \neg x_2) \\
c_6 &= (x_2 \vee x_3) \\
c_7 &= (x_2 \vee \neg x_3) \\
c_8 &= (x_6 \vee \neg x_5)
\end{aligned}
$$

Now if $x_4 = 1$ (x_4@6) a conflict by BCP() occurs:

the roots $\neg x_5$@3 and x_1@6 are suficient to generate a conflict and the clause learned is

$$c_9 = (x_5 \vee \neg x_1)$$

We have $\neg x_5 \wedge x_1 \implies \neg \psi$, thus $\psi \implies x_5 \vee \neg x_1$.

The learned clause c_9 is added to the set to prune the search space (but does not change the result).

In general, the learned clause corresponds to the disjunction of the negation of the root literals in the conflict graph (or optimizations).
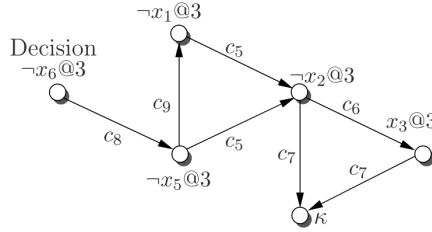
Conflict-driven backtracking

After detecting the conflict and adding the learned clause 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 leaned clause, while erasing all decisions and implications made after that level.

- In the case of $(x_5 \vee \neg x_1)$, 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.

- In the case of learning a unary clause, the solver backtracks to the ground level.

Now at level 3 the clause $c_9 = x_5 \vee \neg x_1$ is unit and implies $\neg x_1@3$

- The clause $c_9 = x_5 \vee \neg x_1$ is *an asserting clause*, that is, it forces an immediate implication after backtracking.

- ANALIZE-CONFLICT can be designed to generate asserting clauses only.

- In the example, another conflict is reached with clause c_7. And in this case the conflict clause learned is just x_2, as explained later.

Example 6. Consider

$$c_1 \wedge c_2 \wedge c_3 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$$

Suppose the decision $x_2@2$ and $x_1@4$. Compute the implication graph.
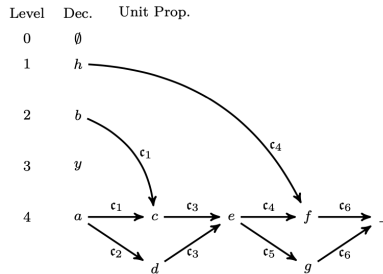
Conflict-driven backtracking terminates

Theorem 7. *It is never the case that the solver enters a decision level again with the same partial assignment.*

Proof. Consider a partial assignment up to the decision level $dl - 1$ that does not end with a conflict, and assume falsely that this state is repeated later, after the solver backtracks to some lower decision level dl^- ($0 \leq dl^- < dl$). Any backtracking from a decision level dl^+ ($dl^+ \geq dl$) to decision level dl^- adds an implication at level dl^- of a variable that was assigned at decision level dl^+. Since this variable has not so far been part of the partial assignment up to decision level dl, once the solver reaches dl again, it is with a different partial assignment, which contradicts our assumption. $\qquad\square$

Example 8. Let

$$
\begin{aligned}
& c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \\
= \quad & (\neg a \vee \neg b \vee c) \wedge (\neg a \vee d) \wedge (\neg c \vee \neg d \vee e) \wedge (\neg h \vee \neg e \vee f) \\
& \wedge (\neg e \vee g) \wedge (\neg f \vee \neg g)
\end{aligned}
$$

Suppose $h@1$, $b@2$ e $a@4$. We have the following conflict graph

Level   Dec.   Unit Prop.

0   ∅
1   h

2   b
                    c_1 c_4
3   y

4   a $\xrightarrow{c_1}$ c $\xrightarrow{c_3}$ e $\xrightarrow{c_4}$ f $\xrightarrow{c_6}$ $\perp$
                c_2 $\searrow$ d $\nearrow$ c_3 c_5 $\searrow$ g $\nearrow$ c_6

Analyze-conflict

Computes a conflict clause and a backtracking level, by transversing the digraph from the conflict node.

Input:
Output: backtracking level and a new asserting clause
function ANALIZE-CONFLICT
    if current-decision-level= 0 **then**
        return −1;
    $cl \leftarrow$ current-conflicting-clause
    while ¬STOPCRITERIONMET(cl) **do**
        $lit \leftarrow$ LASTASSIGNEDLITERAL(cl)
        $var \leftarrow$ VARIABLEOFLITERAL(lit)
        $ante \leftarrow$ ANTECENDENT(lit)
        $cl \leftarrow$ RESOLVE($cl, ante, var$)
    ADDCLAUSE(cl)
    return CLAUSEASSERTINGLEVEL(cl)

To ensure an asserting clause is found we need the following notions

- *unique implication point (UIP)* : Given a partial conflict graph corresponding to the decision level of the conflict, a unique implication point (UIP) is any node other than the conflict node (κ) that is on all paths from the decision node to the conflict node.

- The decision node is a UIP as well as any internal node corresponding to implications at the decision level of the conflict.

- *The first UIP* is a UIP that is closest to the conflict node.

- STOPCRITERIONMET(cl) returns true if and only if cl contains the negation of the first UIP as its single literal at the current decision level. This negated literal becomes asserted immediately after backtracking.

- This strategy backtracks to the lowest decision level

- The function RESOLVE(c_1, c_2, x) corresponds to the resolution rule applied

to clauses c_1 and c_2 with variable x and returns the resolvent clause:

$$\frac{(a_1 \vee \cdots \vee a_n \vee x) \qquad (b_1 \vee \cdots \vee b_m \vee \neg x)}{a_1 \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_m}$$

- A CNF formula is unsatisfiable iff there exists a finite series of resolution steps ending with the empty clause.

- ANALYZE-CONFLICT progresses from right to left on the conflict graph, starting from the conflicting clause, while constructing the new conflict clause through a series of resolution steps. It begins with the conflicting clause cl, in which all literals are set to 0. The literal lit is the literal in cl assigned last, and var denotes its associated variable. The antecedent clause of var, denoted by $ante$, contains : lit as the only satisfied literal, and other literals, all of which are currently unsatisfied. The clauses cl and $ante$ thus contain lit and $-lit$, respectively, and can therefore be resolved with the resolution variable var. The resolvent clause is again a conflicting clause, which is the basis for the next resolution step.

Example 9. Verify that in Example 8 the asserting clause is $\neg h \vee \neg b \vee \neg a$ if one goes to the decision node and $\neg g$ the literal assigned last. The backtracking level would be 2.
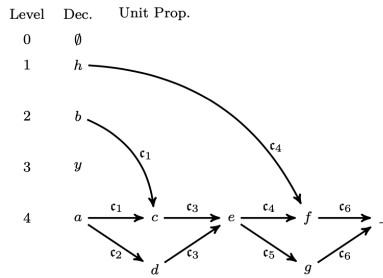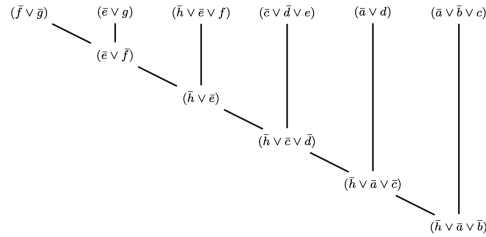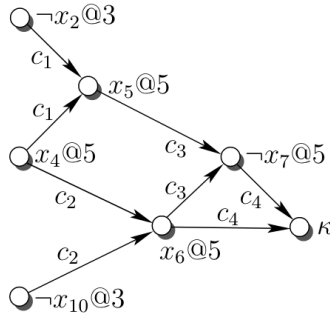

Considering the first UIP e we have the clause $\neg h \vee \neg e$. The backtracking level would be 1.

Example 10. Consider the partial implication graph and set of clauses c_i, and

assume that the implication order in the BCP was x_4, x_5, x_6, x_7.

$$
\begin{aligned}
c_1 &= (\neg x_4 \vee x_2 \vee x_5) \\
c_2 &= (\neg x_4 \vee x_{10} \vee x_6) \\
c_3 &= (\neg x_5 \vee \neg x_6 \vee \neg x_7) \\
c_4 &= (\neg x_6 \vee x_7) \\
&\vdots
\end{aligned}
$$

The conflict clause $c_5 = (x_{10} \vee x_2 \vee \neg x_4)$ is computed through a series of binary resolutions.

name	cl	lit	var	ante
c_4	$(\neg x_6 \vee x_7)$	x_7	x_7	c_3
	$(\neg x_5 \vee \neg x_6)$	$\neg x_6$	x_6	c_2
	$(\neg x_4 \vee \neg x_{10} \vee \neg x_5)$	$\neg x_5$	x_5	c_1
c_5	$(x \vee x_2 \vee \neg x_4)$			

The clause c_5 is an asserting clause in which the negation of the first UIP (x_4) is the only literal from the current decision level.

Decision heuristics

The strategy by which the variables and the value given to them are chosen corresponds to several heuristics.

- Higher priority to literals that appear frequently in short clauses (Jeroslow-Wang): choose literal l that maximises

$$
J(l) = \sum_{C \in \mathcal{B}, l \in C} 2^{-|C|}
$$

- DLIS (*Dynamic Largest Individual Sum*): choose the unassigned literal that satisfies the largest number of currently unsatisfied clauses.

- VSIDS (*Variable State Independent Decaying Sum*): when counting the number of clauses in which every literal appears, disregard the question of whether that clause is already satisfied or not and periodically divide all scores by 2.

- Berkmin: as VSDIS but consider variables and not literals. Gives higher priority to variables that appered in recent conflicts.

Conflict-Driven Clause Learning (CDCL) solvers- Summary

- 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!* is an online resource that proposes, as a standard, a unified notation and a collection of benchmarks for performance evaluation and comparison of tools.

- SAT Live!, `https://www.satlive.org`

- SAT Association, `http://satassociation.org/sat-smt-school.html`

- SAT Examples: `https://sat-smt.codes`

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).

Example

$$a_1 \wedge (a_1 \vee p) \wedge (a_1 \vee \neg a_2) \wedge (\neg a_1 \vee \neg p \vee a_2)$$
$$\wedge (\neg a_2 \vee q) \wedge (\neg a_2 \vee r) \wedge (a_2 \vee \neg q \vee \neg r)$$

- we have 5 variables and 7 clauses

- ```
  p cnf 5 7
  1 0
  1 3 0
  -1 -3 2 0
  1 -2 0
  -2 5 0
  -2 4 0
  -2 -5 -4 0
  ```

Modeling with PL

Scheduling 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:

$$\neg Fri \wedge (Mon \vee Wed \vee Thu) \wedge (\neg Tue \wedge \neg Thu)$$

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

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 of if-then-else chains

Original C code   Optimized C code

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

Are these two programs equivalent?

- Model the variables a and b and the procedures that are called using the Boolean variables a, b, f, g, and h.

- Compile if-then-else chains into Boolean formulae $compile(\textbf{if } x \textbf{ then } y \textbf{ else } z) = (x \wedge y) \vee (\neg x \wedge z)$

- Check the validity of the following formula $compile(original) \Leftrightarrow compile(optimized)$ by reformulating it as a SAT problem.

In this case we have:

$(\neg a \wedge \neg b) \wedge h) \vee ((a \vee a) \wedge (\neg a \wedge g) \vee (a \wedge f))\ iff\ (a \wedge f) \vee (\neg a \wedge (((b \wedge g) \vee \neg (b \wedge h))$

Graph coloring - 2 colors

Graph coloring refers to assigning colors to the nodes such that no two nodes connected by an edge have the same color.
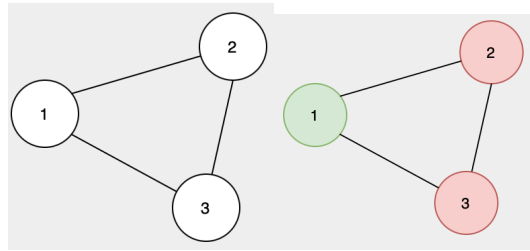
Considering only 2 colors, we will have for nodes 1 and 2

- They both should not be true: $\neg(x_1 \wedge x_2)$

- They both should not be false: $\neg(\neg x_1 \wedge \neg x_2)$

- That is $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$

And the same for x_1 and x_3 and x_2 and x_3.

Graph coloring - 3 colors

All variables are two-digit integers. The first digit denotes the node, and the second digit the color. So literal $x_2 1$ means that node two has color 1. Literal $\neg x_{21}$ means that node 2 does not have color 1.

- No two nodes can have the same color

$$(\neg x_{11} \vee \neg x_{21}) \wedge (\neg x_{12} \vee \neg x_{22}) \wedge (\neg x_{13} \vee \neg x_{23})$$
$$(\neg x_{11} \vee \neg x_{31}) \wedge (\neg x_{12} \vee \neg x_{32}) \wedge (\neg x_{33} \vee \neg x_{33})$$
$$(\neg x_{21} \vee \neg x_{31}) \wedge (\neg x_{22} \vee \neg x_{32}) \wedge (\neg x_{23} \vee \neg x_{23})$$

- Every node has a color

$$(x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{23}) \wedge (x_{31} \vee x_{32} \vee x_{33})$$

- Every node cannot have two colors at the same time

$$(\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{11} \vee \neg x_{13}) \wedge (\neg x_{12} \vee \neg x_{13})$$
$$(\neg x_{21} \vee \neg x_{22}) \wedge (\neg x_{21} \vee \neg x_{23}) \wedge (\neg x_{22} \vee \neg x_{23})$$
$$(\neg x_{31} \vee \neg x_{32}) \wedge (\neg x_{31} \vee \neg x_{33}) \wedge (\neg x_{32} \vee \neg x_{33})$$

Graph coloring: general case

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:

$$\forall \, 1 \le j \le K(x_{uj} \implies \neg x_{vj})$$

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

$$\forall \, 1 \le i \le |V| \left(\bigvee_{j=1}^{K} x_{ij} \right)$$

    - At most one color to each vertex:

$$\forall \, 1 \le i \le |V| \left(\bigwedge_{a=1}^{K} (x_{ia} \implies \bigwedge_{b=1, b \ne a}^{K} \neg x_{ib}) \right)$$

    or equivalently

$$\forall \, 1 \le i \le |V| \left(\bigwedge_{a=1}^{K-1} (x_{ia} \implies \bigwedge_{b=a+1}^{K} \neg x_{ib}) \right)$$

    equivalent to

$$\forall \, 1 \le i \le |V| \left(\bigwedge_{a=1}^{K-1} \bigwedge_{b=a+1}^{K} (\neg x_{ia} \vee \neg x_{ib}) \right)$$

Make a Python program to solve the problem.

References

[BHvMW21] Armin Biere, Marjin Heulen, Hans van Maaren, and Tobis Walsh. *Handbook of Satisfiability*. IOS Press, second edition, 2021.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Verlag, 2007.

[Knu22] Donald E. Knuth. *The Art of Computer Programming.Combinatorial ALgorithms Part II.* Addison-Wesley, 2022.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures:An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.