UNIVERSITÀ DEGLI STUDI DI MILANO

PROPOSITIONAL SATISFIABILITY AND CDCL

Mathematical Logic – Corti Filippo (77044A)

Code available at:
https://github.com/Filippo-Corti/SAT-Solvers

Table of Contents

UNIVERSITÀ DEGLI STUDI DI MILANO

1. Overview of the $\Gamma \vDash A$ Problem

The $\Gamma \vDash A$ Problem

Definition: Let $\Gamma \subseteq F_{\mathcal{L}}$, $A \in F_{\mathcal{L}}$, then A is a Logical Consequence of Γ ($\Gamma \vDash A$) IFF for every truth-evaluation $v: \mathcal{L} \to \{0,1\}$ it holds that:

$$v \vDash \Gamma \implies v \vDash A$$

Lemma: Let $\Gamma \subseteq F_{\mathcal{L}}$, $A \in F_{\mathcal{L}}$, then:

$$\Gamma \vDash \mathrm{A} \quad \mathrm{IFF} \quad \Gamma \cup \{\neg A\} \text{ is UNSAT}$$

This allows us to **reduce** the problem of Logical Consequence (LOGCONS) to a problem of Unsatisfiability (UNSAT).

LOGCONS, UNSAT and CNFUNSAT

Normal Forms are standardized ways of writing formulas, which can significally simplify their analysis. Deterministic polynomial (Linear) time procedures are known to transform formulas:

- To an equivalent Implication Free Normal Form (IFNF);
- From IFNF to an equivalent Negation Normal Form (NNF);
- From NNF to an equisatisfiable Conjunctive Normal Form (CNF).

This allows us to **polynomially reduce** the problem of Unsatisfiability to the problem of CNF-Unsatisfiability:

$$\text{LOGCONS} \preccurlyeq_p \text{UNSAT} \preccurlyeq_p \text{CNFUNSAT}$$

SAT Solving Algorithms

SAT Solving Algorithms are the means with which we solve Logical Consequence Problems.

These are:

- **Davis-Putnam Procedure** (DPP);
- **Davis-Putnam-Logemann-Loveland** (DPLL);
- **Conflict-Driven Clause Learning** (CDCL).

Note that the problem of **Propositional Unsatisfiability** is:

- **Decidable**, for Finite Theories.
- Only **Semi-Decidable**, for Infinite Theories (via the Compactness Theorem).

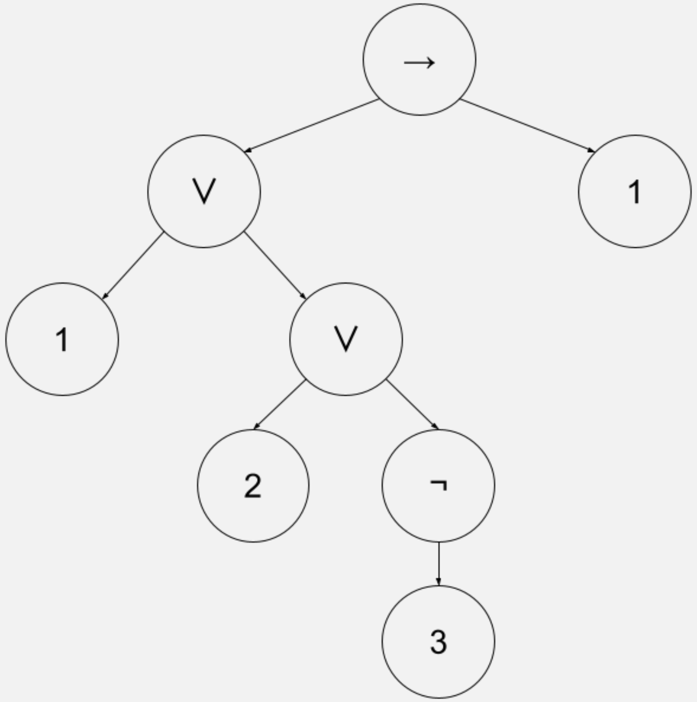
That is, for infinite theories we are only able to give "**UNSAT**" as an answer.

2. Parsing

Representations of Formulas

Abstract Syntax Tree (AST):

AST of $(A \vee B \vee \neg C) \rightarrow A$:



DIMACS-CNFs:

DIMACS format for

$(x \vee y \vee \neg z) \wedge (\neg y \vee z)$:

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

3. Preprocessing

Preprocessing Pipeline

Given the Logical Consequence Problem $\Gamma \vDash \mathrm{A}$, “Preprocessing” consists in:

1. Rewriting $\Gamma \vDash \mathrm{A}$ as:

$$\mathrm{F} = \Gamma_1 \wedge \Gamma_2 \wedge \cdots \wedge \Gamma_n \wedge \neg A$$

2. Converting $F \in \mathcal{F}_{\mathcal{L}}$ into $F^I \in IFNF$, such that $F \equiv F^I$;
3. Converting $F^I \in IFNF$ into $F^N \in NNF$, such that $F^I \equiv F^N$;
4. Converting $F^N \in NNF$ into $F^C \in CNF$, such that $F^N\ EQUISAT\ F^C$.

Notice:

$$\Gamma \vDash \mathrm{A}$$

$$\text{IFF} \quad \Gamma \cup \{\neg A\} \text{ is UNSAT}$$

$$\text{IFF } \mathrm{F} = \Gamma_1 \wedge \Gamma_2 \wedge \cdots \wedge \Gamma_n \wedge \neg A \text{ is UNSAT}$$

$$\text{IFF } F^C \text{is UNSAT}$$

IFNF and NNF

Lemma: For any $F \in \mathcal{F}_L^{(n)}$, we can build $G \in \mathcal{F}_L^{(n)}$ in a finite number of steps such that:

$$F \equiv G \text{ and } G \in IFNF$$

Intuitively:

- Each instance $A \rightarrow B$ is replaced with $\neg A \vee B$.

Lemma: For any $F \in IFNF$, we can build $G \in \mathcal{F}_L^{(n)}$ in a finite number of steps such that:

$$F \equiv G \text{ and } G \in NNF$$

Intuitively:

- Each instance $\neg\neg C$ is replaced with C;
- Each instance $\neg(C \vee D)$ is replaced with $\neg C \wedge \neg D$;
- Each instance $\neg(C \wedge D)$ is replaced with $\neg C \vee \neg D$.

Equisatisfiable CNF

Linear Time Algorithms that convert to CNF only preserve **Equisatisfiability**.

Assuming w.l.o.g. that, given $A \in NNF$, all violations of the Conjunctive Normal Form are of the shape $C \vee (D_1 \wedge D_2)$:

- For each violation $\mathrm{B} = C \vee (D_1 \wedge D_2)$, $B \preccurlyeq A$:
  a) Introduce a new propositional letter $a_B \in \mathcal{L}$.
  b) Build $B' = B\left[{}^{a_B}/_{D_1 \wedge D_2}\right] \wedge (\neg a_B \vee D_1) \wedge (\neg a_B \vee D_2)$
  c) Replace B with B'.

Notice that the "**tail**" is just $a_B \rightarrow (D_1 \vee D_2)$.

4. DPLL

DPLL

DPLL is a **Backtracking**-based Search Algorithm. It tries to build a Satisfying Truth Assignment $v: Var(S) \to \{0,1\}$ for the CNF S, by applying the following rules:

1. **Unit Subsumption**: “removes” verified clauses;
2. **Unit Resolution**: “removes” unsatisfied literals;
3. **Assertion**: determines forced assignments under the current v;
4. **Split**: chooses a propositional letter currently unassigned in v and tries both values;
5. **Backtracking**: jumps back to the nearest ancestor that generated from a split.

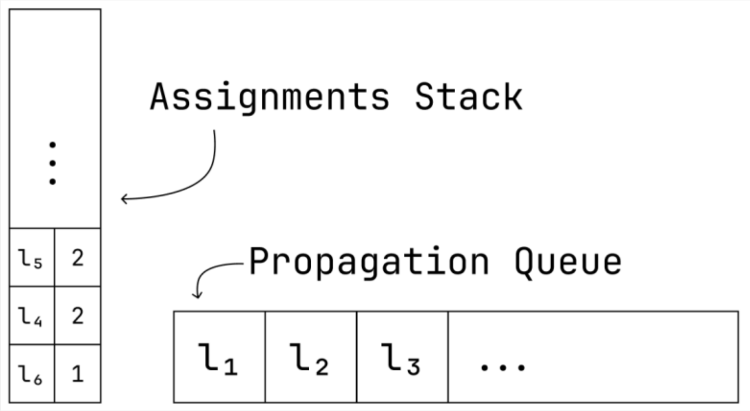
The procedure ends when:

- Backtracking is required but no further possible $\Longrightarrow$ S is **UNSAT**; or
- All clauses and literals have been removed $\Longrightarrow$ S is **SAT** (by any completion of v).

DPLL (in practice)

At its core, a DPLL implementation uses:

- The DIMACS-CNF to solve;
- The Partial Truth Assignment v to complete;
- An **Assignments Stack**, tracking which assignments have been made;
- A **Propagation Queue**, tracking which assignments should be (made and) propagated.




```
function DPLL(cnf, v):
  <cnf: the CNF to solve for;
   v: the partial truth assignment>

  conflict = propagate()
  if conflict:
    return UNSAT
  if not conflict and v.is_total():
    return SAT

  decision = choose_splitting_letter()
  propagation_queue.add(decision)
  ok = DPLL(cnf, v)
  if ok:
    return SAT
  backtrack()

  propagation_queue.add(-decision)
  ok = DPLL(cnf, v)
  if ok:
    return SAT
  backtrack()

  return UNSAT
```

DPLL Optimizations

The core of DPLL is enriched with multiple optimizations:

- **Watchlists**: dictionaries for quick access to which clauses contain which literals.
- **Branching Heuristics**: smarter ways to pick literals to split on make DPLL reach a conclusion faster:
  - A common choice is Dynamic Largest Individual Sum (**DLIS**) – Literals that appear more often in unsatisfied clauses;
- **Two-Watched Literals Mechanism**: as an improvement of watchlists, we can actually limit ourselves to observing two literals for each clause:
  - If one of the two literals becomes False, we look for a substitute to watch;
  - If one of the two literals is True, the clause is verified;
  - If the literals are [False, None], we force the assignment of the unassigned one.

Testing DPLL

Even with multiple optimizations, DPLL struggles with large CNFs:

- It took minutes to run 50- instances;
- It ended in reasonable time only in some lucky 100- instances.

Before implementing the Two-Watched Literals Mechanism, even the smallest benchmark problems were too large.

5. CDCL

CDCL

CDCL is a **Backjumping**-based Search Algorithm. The main differences w.r.t DPLL are:

1. Once a conflict is reached in the current Partial Truth Assignment, CDCL determines a clause that better represents the reason for the conflict. The clause is then "**learned**", in the sense that it is added to the CNF we are trying to solve.
2. After learning a clause, CDCL does **not** backtrack **chronologically**. Instead, it finds the last state in which the learnt clause is not conflicting anymore. Then, it jumps to that state.

The procedure ends when:

- A conflict happens at level 0 $\Longrightarrow$ S is **UNSAT**; or
- There are no conflicts and v is a Total Assignment $\Longrightarrow$ S is **SAT** (by v).

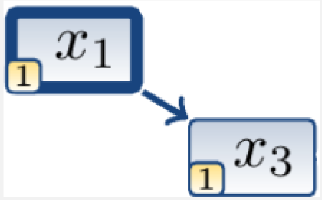
CDCL – The "Computer Science" approach (1)

We can formalize the process of Clause Learning using Graph Theory:

- The Assignments Stack (Trail) can be represented as an Implication Graph (V, E), in which:
  - Decisions are nodes with in-degree = 0.
  - Each connection $(l_1 \rightarrow l_2) \in E$ represents the fact that the literal l_2 was set True due to a forced assignment that involved a clause containing $\neg l_1$.

For example, consider the case of a CNF with a clause $\neg x_1 \vee x_3$:

1. A Decision may set x_1 = True.
2. For the clause to be verified, we need to force x_3 = True.
3. The Implication Graph will be:

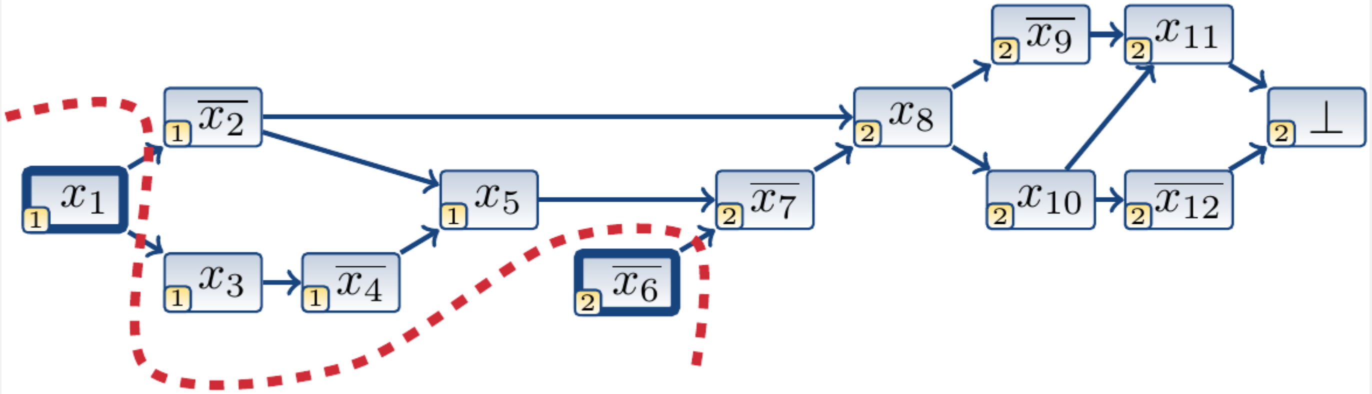


CDCL – The “Computer Science” approach (2)

- A Conflict Cut is a partition W = (A, B) of the set of vertices such that:
  - All decision literals are in A;
  - The conflict vertex $\perp$ is in B.
- Each Conflict Cut W is associated with the Reason Clause c_W :

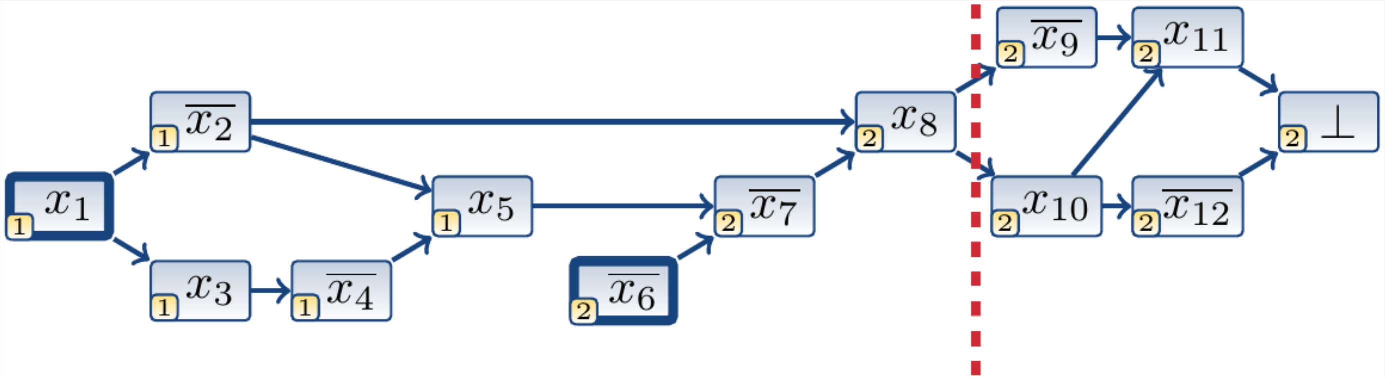
$$c_W = \bigvee_{l \in R} \neg l$$

$$R = \{ l \in A \mid \exists l' \in B : (l, l') \in E \}$$



CDCL – The "Computer Science" approach (3)

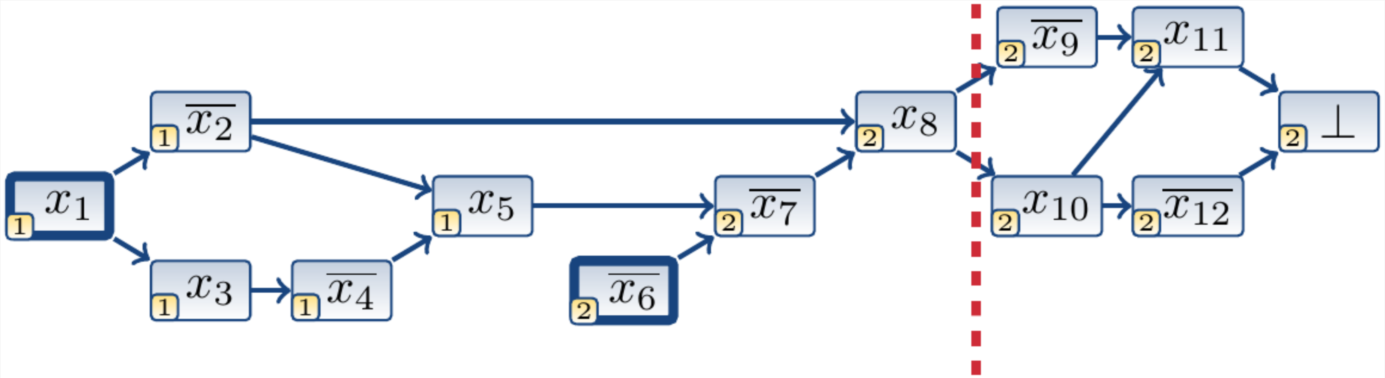
- A Vertex l in the Implication Graph is called a Unique Implication Point (UIP) if it is traversed by every path from the latest decision literal to the conflict vertex $\perp$.
  - To each UIP corresponds a UIP Cut (A, B), where B contains all successors of l up to $\perp$, and A contains the rest of vertices.
- The First-UIP cut is the first UIP we encounter when traversing the Graph backwards from the conflict vertex $\perp$: it corresponds to the UIP Cut with the largest set A.



CDCL – The "Computer Science" approach (4)

- CDCL learns the Reason Clause C of the First-UIP Cut.

Intuitively, C is the clause that summarizes the minimal cause of the conflict: if we want to avoid the conflict ⊥, at the very least we need to jump back to this point.



- CDCL then backjumps to the second largest decision level of the literals in C, then propagates over C.

CDCL – The "Mathematical Logic" approach

In pratice, the «Computer Science» approach is not very straightforward to implement. A better choice is to try with the «Mathematical Logic» approach to the problem of determining the clause to learn.

From a Logical standpoint, CDCL is simply applying the Resolution Calculus:

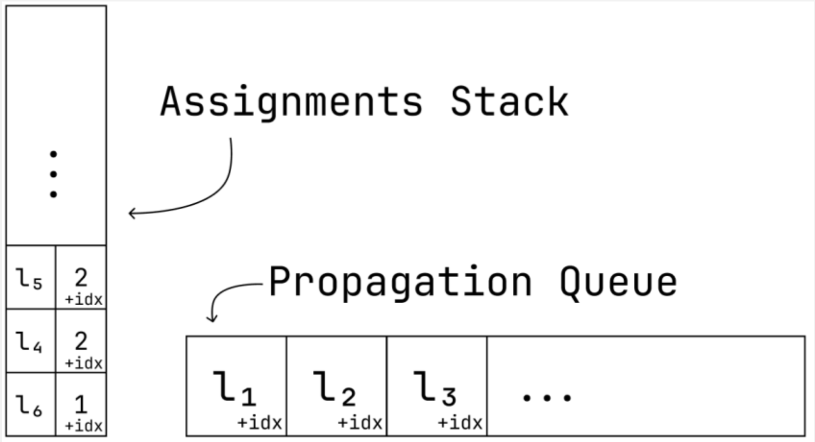
- Starting from the conflict clause, moving backwards along the trail;
- Up until only one literal from the decision level of the conflict remains. That literal is then the one on which propagation happens.

Resolution Rule: $$\frac{\{a\} \vee B \qquad \{\neg a\} \vee C}{B \vee C}$$

CDCL (in practice)

CDCL implementation refines the structures of DPLL:

- The **Assignments Stack** also tracks the index of the conflicting clause;
- The **Propagation Queue** also tracks the index of the conflicting clause;
- A **Level Map** is used to conveniently know at which level each literal was assigned.




```
function CDCL(cnf, v):
  <cnf: the CNF to solve for;
   v: the partial truth assignment>

  while True:
    conflict = propagate()

    switch (conflict):
      case NO_CONFLICT:
        if v.is_total():
          return SAT
        decision = choose_branching_letter()
        propagation_queue.add(decision)

      case UNIT_CLAUSE_CONFLICT:
        return UNSAT

      default:
        if current_level == 0:
          return UNSAT

        clause, literal = resolve(conflict)
        learn(clause)
        backjump(clause.level)
        propagation_queue.add(literal)
```

CDCL Optimizations

On top of the optimizations inherited by DPLL, CDCL was further improved:

- **Branching Heuristics**: better heuristics have been developed for CDCL:
  - A common choice is Variable State Independent Decaying Sum (**VSIDS**) – propositional letters that appear more often in conflicts are more likely to be chosen;
- **Phase Saving**: working together with the heuristic, it consists in saving the last truth-value to which each letter was assigned to. When the letter is chosen by the heuristic, the truth-value is set to the previous one;
- **Restarts**: CDCL can get stuck in areas of the search space that are dense with conflicts. A good strategy is to periodically force a backjump to level 0.
- **Forgetting Learned Clauses**: in order to make propagation faster and reduce memory space, we can consider deleting some of the learnt clauses that are deemed less useful.

Testing CDCL

CDCL was orders of magnitude faster than DPLL:

- From the first implementation, it solved the entire **AIM Dataset** (72 instances of 50-, 100- and 200-variables CNFs, both SAT and UNSAT) in around **5.5** seconds;
- Implementing the **Heuristic VSIDS** made the algorithm around 10x faster, solving the AIM Dataset in around **0.45** seconds;

- **Restarts** were a significant improvement only for very large instances (200+ variables CNFs) or specifically hard ones.
- **Forgets** gave mixed results, with some CNFs being solved much faster and some much slower.

UNIVERSITÀ DEGLI STUDI DI MILANO

THANK YOU

Mathematical Logic – Corti Filippo (77044A)

Code available at:
https://github.com/Filippo-Corti/SAT-Solvers

Other resources:
DPLL and CDCL – https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html
DIMACS CNF Format – https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html
Benchmark Problems for SAT – https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

Generative AI was used as a complementary tool to investigate state-of-the-art techniques on SAT Solving. Code was written entirely by the author.
The author assumes full responsibility for the final accuracy and errors in the material.