2.6 The CDCL Procedure

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

Assumption:
Clauses contain neither duplicated literals nor complementary literals.

CDCL: Conflict Driven Clause Learning
Satisfiability of Clause Sets

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

\[ \mathcal{A} \models C \text{ if and only if } \mathcal{A} \models L \text{ for some literal } L \in C. \]
Partial Valuations

Since we will construct satisfying valuations incrementally, we consider partial valuations (that is, partial mappings \( A : \Sigma \rightarrow \{0, 1\} \)).

Every partial valuation \( A \) corresponds to a set \( M \) of literals that does not contain complementary literals, and vice versa:

\[
\begin{align*}
A(L) & \text{ is true, if } L \in M. \\
A(L) & \text{ is false, if } \overline{L} \in M. \\
A(L) & \text{ is undefined, if neither } L \in M \text{ nor } \overline{L} \in M.
\end{align*}
\]

We will use \( A \) and \( M \) interchangeably. Note that truth of a literal with respect to \( M \) is defined differently than for \( N_I \).
Partial Valuations

A clause is true under a partial valuation $\mathcal{A}$ (or under a set $M$ of literals) if one of its literals is true; it is false (or “conflicting”) if all its literals are false; otherwise it is undefined (or “unresolved”).
Observation: Let $A$ be a partial valuation. If the set $N$ contains a clause $C$, such that all literals but one in $C$ are false under $A$, then the following properties are equivalent:

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

$C$ is called a unit clause; $L$ is called a unit literal.
One more observation: Let $\mathcal{A}$ be a partial valuation and $P$ a variable that is undefined under $\mathcal{A}$. If $P$ occurs only positively (or only negatively) in the unresolved clauses in $N$, then the following properties are equivalent:

- there is a valuation that is a model of $N$ and extends $\mathcal{A}$.
- there is a valuation that is a model of $N$ and extends $\mathcal{A}$ and assigns 1 (0) to $P$.

$P$ is called a pure literal.
boolean DPLL(literal set $M$, clause set $N$) {
    if (all clauses in $N$ are true under $M$) return true;
    elsif (some clause in $N$ is false under $M$) return false;
    elsif ($N$ contains unit clause $P$) return DPLL($M \cup \{P\}$, $N$);
    elsif ($N$ contains unit clause $\neg P$) return DPLL($M \cup \{\neg P\}$, $N$);
    elsif ($N$ contains pure literal $P$) return DPLL($M \cup \{P\}$, $N$);
    elsif ($N$ contains pure literal $\neg P$) return DPLL($M \cup \{\neg P\}$, $N$);
    else {
        let $P$ be some undefined variable in $N$;
        if (DPLL($M \cup \{\neg P\}$, $N$)) return true;
        else return DPLL($M \cup \{P\}$, $N$);
    }
}
Initially, DPLL is called with an empty literal set and the clause set $N$. 
2.7 From DPLL to CDCL

In practice, there are several changes to the procedure:

The pure literal check is only done while preprocessing (otherwise is too expensive).

The branching variable is not chosen randomly.

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

CDCL = DPLL + Information is reused by learning + Restart + Specific Data Structures
Branching Heuristics

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

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

In general: choose variables that occur frequently, prefer variables from recent conflicts.
The Deduction Algorithm

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

Maintaining this number is expensive, however.
Better approach: “Two watched literals”:

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

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

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

Watched literal information need not be restored upon backtracking.
Conflict Analysis and Learning

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

Method: Learning:

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

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

Related technique:

non-chronological backtracking ("backjumping"):

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

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

If no solution is found within a certain time limit, it can be useful to restart from scratch with an adopted variable selection heuristics, but learned clauses are kept.

In particular, after learning a unit clause a restart is done.
Formalizing DPLL with Refinements

The DPLL procedure is modelled by a transition relation \( \Rightarrow_{\text{DPLL}} \) on a set of states.

States:
- \textit{fail}
- \((M; N)\)

where \( M \) is a \textit{list of annotated literals} and \( N \) is a set of clauses. We use \(+\) to right add a literal or a list of literals to \( M \).

Annotated literal:
- \( L \): deduced literal, due to unit propagation.
- \( L^d \): decision literal (guessed literal).
Formalizing DPLL with Refinements

Unit Propagate:

\[(M; N \cup \{C \lor L\}) \Rightarrow_{\text{DPLL}} (M + L; N \cup \{C \lor L\})\]

if \(C\) is false under \(M\) and \(L\) is undefined under \(M\).

Decide:

\[(M; N) \Rightarrow_{\text{DPLL}} (M + L^d; N)\]

if \(L\) is undefined under \(M\) and contained in \(N\).

Fail:

\[(M; N \cup \{C\}) \Rightarrow_{\text{DPLL}} \text{fail}\]

if \(C\) is false under \(M\) and \(M\) contains no decision literals.
Formalizing DPLL with Refinements

Backjump:

\[(M' + L^d + M''; N) \Rightarrow_{DPLL} (M' + L'; N)\]

if there is some “backjump clause” \(C \lor L'\) such that

\(N \models C \lor L'\),

\(C\) is false under \(M'\), and

\(L'\) is undefined under \(M'\).
Formalizing DPLL with Refinements

We will see later that the Backjump rule is always applicable, if the list of literals $M$ contains at least one decision literal and some clause in $N$ is false under $M$.

There are many possible backjump clauses. One candidate: $\overline{L_1} \lor \ldots \lor \overline{L_n}$, where the $L_i$ are all the decision literals in $M + L^d + M'$. (But usually there are better choices.)
Formalizing DPLL with Refinements

Lemma 2.16:
If we reach a state $(M; N)$ starting from (nil; $N$), then:

(1) $M$ does not contain complementary literals.

(2) Every deduced literal $L$ in $M$ follows from $N$ and decision literals occurring before $L$ in $M$. 
Lemma 2.17:
Every derivation starting from \((\text{nil}; N)\) terminates.
Lemma 2.18:
Suppose that we reach a state \((M; N)\) starting from \((\text{nil}; N)\) such that some clause \(D \in N\) is false under \(M\). Then:

(1) If \(M\) does not contain any decision literal, then “Fail” is applicable.

(2) Otherwise, “Backjump” is applicable.
Theorem 2.19:
(1) If we reach a final state \((M; N)\) starting from \((\text{nil}; N)\), then \(N\) is satisfiable and \(M\) is a model of \(N\).
(2) If we reach a final state \(\text{fail}\) starting from \((\text{nil}; N)\), then \(N\) is unsatisfiable.