2.5 The DPLL Procedure

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

Assumption:
Clauses contain neither duplicated literals nor complementary literals.

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

Satisfiability of Clause Sets

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

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

Partial Valuations

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

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

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

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

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

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

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

Unit Clauses

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

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

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

Pure Literals

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

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

P is called a *pure literal.*

The Davis-Putnam-Logemann-Loveland Proc.

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

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

2.6 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 $\Rightarrow$ the backtrack stack is managed explicitly (it may be possible and useful to backtrack more than one level).

Information is reused by learning.

Under certain circumstances, the procedure is restarted.

$\Rightarrow$ CDCL: Conflict Driven Clause Learning.

Branching Heuristics

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

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

In general: Choose variables that occur frequently; after a restart prefer variables from recent conflicts.

Implementing Unit Propagation Efficiently

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 addition, it is useful to restart after a unit clause has been learned.

Formalizing DPLL with Refinements

The DPLL procedure is modeled by a transition relation $\Rightarrow_{\mathrm{DPLL}}$ on a set of states.

States:

- *fail*
- $M \parallel N$,

where M is a *list of annotated literals* and N is a set of clauses.

Annotated literal:

- L: deduced literal, due to unit propagation.
- L^{d}: decision literal (guessed literal).

Unit Propagate:

$M \parallel N \cup \{C \vee L\} \;\Rightarrow_{\mathrm{DPLL}}\; M\, L \parallel N \cup \{C \vee L\}$

if C is false under M and L is undefined under M.

Decide:

$M \parallel N \;\Rightarrow_{\mathrm{DPLL}}\; M\, L^{\mathrm{d}} \parallel N$

if L is undefined under M and contained in N.

Fail:

$M \parallel N \cup \{C\} \;\Rightarrow_{\mathrm{DPLL}}\; \mathit{fail}$

if C is false under M and M contains no decision literals.

Backjump:

$M'\, L^{\mathrm{d}}\, M'' \parallel N \;\Rightarrow_{\mathrm{DPLL}}\; M'\, L' \parallel N$

if there is some "backjump clause" $C \vee L'$ such that
$N \models C \vee L'$,
C is false under M', and
L' is undefined under M'.

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: $\overline{L_1} \vee \ldots \vee \overline{L_n}$, where the L_i are all the decision literals in $M'\, L^{\mathrm{d}}\, M''$. (But usually there are better choices.)

Lemma 2.11 *If we reach a state $M \parallel N$ starting from $\varepsilon \parallel N$, 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.

Proof. By induction on the length of the derivation. □

Lemma 2.12 *Every derivation starting from $\varepsilon \parallel N$ terminates.*

Proof. Let $M \parallel N$ and $M' \parallel N'$ be two DPLL states, such that $M = M_0 L_1^{\mathrm{d}} M_1 \ldots L_k^{\mathrm{d}} M_k$ and $M' = M_0' L_1'^{\mathrm{d}} M_1' \ldots L_{k'}'^{\mathrm{d}} M_{k'}'$. We define a relation $\succ$ on lists of annotated literals by $M \succ M'$ if and only if

(i) there is some j such that $0 \le j \le \min(k, k')$, $|M_i| = |M_i'|$ for all $0 \le i < j$, and $|M_j| < |M_j'|$, or

(ii) $|M_i| = |M_i'|$ for all $0 < i \le k < k'$ and $|M| < |M'|$.

It is routine to check that $\succ$ is irreflexive and transitive, hence a strict partial ordering, and that for every DPLL step $M \parallel N \Rightarrow_{\mathrm{DPLL}} M' \parallel N'$ we have $M \succ M'$. Moreover, the set of propositional variables in N is finite, and each of these variables can occur at most once in a literal list (positively or negatively, with or without a d-superscript). So there are only finitely many literal lists that can occur in a DPLL derivation. Consequently, if there were an infinite DPLL derivation, there would be some cycle $M \parallel N \Rightarrow^{+}_{\mathrm{DPLL}} M \parallel N'$, so by transitivity $M \succ M$, but that would contradict the irreflexivity of $\succ$.

Lemma 2.13 *Suppose that we reach a state $M \parallel N$ starting from $\varepsilon \parallel N$ 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.

Proof. (1) Obvious.

(2) Let $L_1, \ldots, L_n$ be the decision literals occurring in M (in this order). Since $M \models \neg D$, we obtain, by Lemma 2.11, $N \cup \{L_1, \ldots, L_n\} \models \neg D$. Since $D \in N$, this is a contradiction, so $N \cup \{L_1, \ldots, L_n\}$ is unsatisfiable. Consequently, $N \models \overline{L_1} \vee \cdots \vee \overline{L_n}$. Now let $C = \overline{L_1} \vee \cdots \vee \overline{L_{n-1}}$, $L' = \overline{L_n}$, $L = L_n$, and let M' be the list of all literals of M occurring before L_n, then the condition of "Backjump" is satisfied. □

Theorem 2.14 *Suppose that we reach a final state starting from $\varepsilon \parallel N$.*

(1) If the final state is $M \parallel N$, then N is satisfiable and M is a model of N.

(2) If the final state is fail, then N is unsatisfiable.

Proof. (1) Observe that the "Decide" rule is applicable as long as literals are undefined under M. Hence, in a final state, all literals must be defined. Furthermore, in a final state, no clause in N can be false under M, otherwise "Fail" or "Backjump" would be applicable. Hence M is a model of every clause in N.

(2) If we reach *fail*, then in the previous step we must have reached a state $M \parallel N$ such that some $C \in N$ is false under M and M contains no decision literals. By part (2) of Lemma 2.11, every literal in M follows from N. On the other hand, $C \in N$, so N must be unsatisfiable. □

Getting Better Backjump Clauses

Suppose that we have reached a state $M \parallel N$ such that some clause $C \in N$ (or following from N) is false under M.

Consequently, every literal of C is the complement of some literal in M.

(1) If every literal in C is the complement of a decision literal of M, then C is a backjump clause.

(2) Otherwise, $C = C' \vee \overline{L}$, such that L is a deduced literal.

For every deduced literal L, there is a clause $D \vee L$, such that $N \models D \vee L$ and D is false under M.

Then $N \models D \vee C'$ and $D \vee C'$ is also false under M. ($D \vee C'$ is a *resolvent* of $C' \vee \overline{L}$ and $D \vee L$.)

By repeating this process, we will eventually obtain a clause that consists only of complements of decision literals and can be used in the "Backjump" rule.

Moreover, such a clause is a good candidate for learning.

Learning Clauses

The DPLL system can be extended by two rules to learn and to forget clauses:

Learn:

$$M \parallel N \;\Rightarrow_{\text{DPLL}}\; M \parallel N \cup \{C\}$$

if $N \models C$.

Forget:

$$M \parallel N \cup \{C\} \;\Rightarrow_{\text{DPLL}}\; M \parallel N$$

if $N \models C$.

If we ensure that no clause is learned infinitely often, then termination is guaranteed.

The other properties of the basic DPLL system hold also for the extended system.

Restart

The restart rule is typically applied after a certain number of clauses have been learned or a unit is derived:

Restart:

$M \parallel N \;\Rightarrow_{\mathrm{DPLL}}\; \varepsilon \parallel N$

If Restart is only applied finitely often, termination is guaranteed.

The restart rule is closely coupled with the variable order heuristic.

Variable Order Heuristic

We associate a positive *score* to every propositional variable P_i. At the start, k_i may for example be the number of occurrences of P_i in N.

The variable order is then the descending ordering of the P_i according to the k_i.

The scores k_i are adjusted during a CDCL run.

- Every time a learned clause is computed after a conflict, the involved propositional variables obtain a bonus b, i.e., $k_i := k_i + b$.
- After each restart, the variable order is recomputed, using the new scores.
- After each j^{th} restart, the scores are leveled: $k_i := k_i/l$ for some l.

The purpose of these mechanisms is to keep the search focused. The parameter b directs the search around the conflict, the parameter j decides how many learned clauses are "sufficient" to move in "speed" of parameter l away from this conflict.

Preprocessing

Before DPLL search, and before computation of the variable order heuristics, a number of preprocessing steps are performed:

(i) Subsumption

$N \cup \{C\} \cup \{D\} \;\Rightarrow\; N \cup \{C\}$

if $C \subseteq D$ considering C, D as multisets of literals.

(ii) Purity deletion

Delete all clauses containing a literal L where $\overline{L}$ does not occur in the clause set.

(iii) Merging replacement resolution

$N \cup \{C \vee L\} \cup \{D \vee \overline{L}\} \ \Rightarrow\ N \cup \{C \vee L\} \cup \{D\}$

if $C \subseteq D$ considering C, D as multisets of literals.

(iv) Tautology deletion

(v) Literal Elimination

Compute all possible resolution steps

$$\frac{C \vee L \qquad D \vee \overline{L}}{C \vee D}$$

on a literal L with premises in N; add the conclusions to N and then throw away all clauses containing L or $\overline{L}$; repeat this as long as $|N|$ does not grow.

Further Information

The ideas described so far have been implemented in all modern SAT solvers: *zChaff*, *miniSAT*, *picoSAT*.

Because of the importance of clause learning the algorithm is now called CDCL: Conflict Driven Clause Learning.

Literature

Lintao Zhang and Sharad Malik: The Quest for Efficient Boolean Satisfiability Solvers; Proc. CADE-18, LNAI 2392, pp. 295–312, Springer, 2002.

Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli: Solving SAT and SAT Modulo Theories; From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T), pp. 937–977, Journal of the ACM, 53(6), 2006.

Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh (eds.): Handbook of Satisfiability; IOS Press, 2009

Daniel Le Berre's slides at VTSA'09: `http://www.mpi-inf.mpg.de/vtsa09/`.

2.7 Other Calculi

OBDDs (Ordered Binary Decision Diagrams):

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

⇒ canonical representation of formulas.

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

FRAIGs (Fully Reduced And-Inverter Graphs)

Minimized graph representation of boolean circuits.

⇒ semi-canonical representation of formulas.

Implementation needs DPLL (and OBDDs) as subroutines.

Ordered resolution
Tableau calculus
Hilbert calculus
Sequent calculus
Natural deduction

see next chapter