

PROPOSITIONAL LOGIC: MODERN SAT SOLVING

Course “Computational Logic”



Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

Wolfgang.Schreiner@risc.jku.at



SAT: The Satisfiability Problem of Propositional Logic

We now consider another deduction calculus for propositional logic.

- **Judgement:** sequent $F \vdash$.
 - Clause set $F = \{C_1, \dots, C_n\}$ with interpretation “ F is **unsatisfiable**”.

- **Inference rules:**

$$\frac{\{\} \in F}{F \vdash} \text{ (AX)} \qquad \frac{F[p \leftarrow \text{true}] \vdash \quad F[p \leftarrow \text{false}] \vdash}{F \vdash} \text{ (SPLIT)}$$

- $F[p \leftarrow t]$: F without any occurrence of p or $\neg p$ by assigning truth value t to p .
 - If $t = \text{true}$, we remove every occurrence of $\neg p$ and every clause that contains p .
 - If $t = \text{false}$, we remove every occurrence of p and every clause that contains $\neg p$.
 - Intuitively justified by the following logical equivalences:

$$(C \vee \perp) \equiv C$$

$$(C \vee \top) \wedge D \equiv D$$

The basis for modern decision procedures (“SAT solvers”) for the SAT problem.

Deduction Tree

We show the validity of $(p \Rightarrow (q \Rightarrow r)) \wedge (p \Rightarrow q) \wedge p \Rightarrow r$.

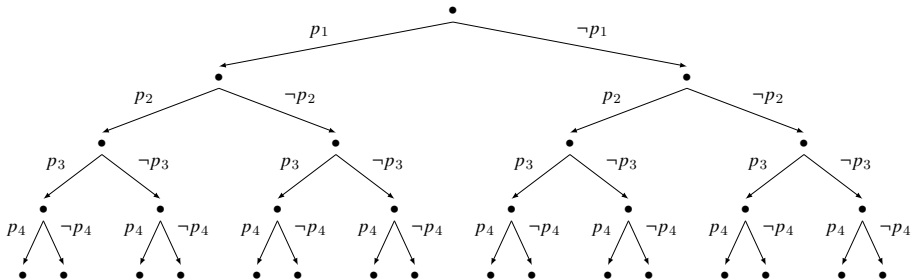
- We show the unsatisfiability of clause set $\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}$.

$$\frac{\frac{\frac{}{\{\} \vdash} \text{AX} \quad \frac{}{\{\} \vdash} \text{AX}}{\{r\}, \{\neg r\} \vdash} \text{SPLIT}(r) \quad \frac{}{\{\}, \{\neg r\} \vdash} \text{AX}}{\frac{\{ \neg q, r \}, \{q\}, \{\neg r\} \vdash}{\{ \neg p, \neg q, r \}, \{ \neg p, q \}, \{p\}, \{\neg r\} \vdash} \text{SPLIT}(q) \quad \frac{}{\{\}, \{\neg r\} \vdash} \text{AX}} \text{SPLIT}(p)$$

The calculus gives rise to binary deduction trees.

Soundness and Completeness

- **Soundness:** Assume valuation v satisfies F . Then v also satisfies $F[p \leftarrow v(p)]$. Thus, if both $F[p \leftarrow \text{true}]$ and $F[p \leftarrow \text{false}]$ are unsatisfiable, F is unsatisfiable.
- **Completeness:** For an unsatisfiable F with atoms $p_1, \dots, p_n$, we have a deduction tree of (at most) height n with 2^n branches $F \xrightarrow{p_1 \leftarrow v_1} F_1 \xrightarrow{p_2 \leftarrow v_2} \dots \xrightarrow{p_n \leftarrow v_n} F_n = \{\{\}\}$.
 - We typically write p for $p \leftarrow \text{true}$ and $\neg p$ for $p \leftarrow \text{false}$.



Every path in the tree denotes a potential satisfying valuation.

The DPLL Algorithm

An implementation of (the dual form of) the inference rules (Davis, Putnam, Logemann, Loveland, 1961).

```
function DPLL( $F$ )                                ▶ returns true, if clause set  $F$  is satisfiable  
  if  $F = \{ \}$  then return true  
  if  $\{ \} \in F$  then return false  
  choose  $p \in \bigcup F$   
  return DPLL( $F[p \leftarrow \text{true}]$ ) or DPLL( $F[p \leftarrow \text{false}]$ )  
end function
```

- **Worst-case time complexity** $O(2^n)$ for n propositional variables.
 - Probably there is no *generally* better algorithm: since the SAT problem is NP -complete (Cook, 1971), there exists (unless $P = NP$) no deterministic way to solve the SAT problem in worst-case polynomial time.

Modern SAT solvers are based on the DPLL algorithm.

The DPLL Algorithm

The algorithm is typically augmented to produce a satisfying valuation.

```
function DPLL( $F$ )  
    return DPLL( $F$ , EMPTY)  
end function
```

```
function DPLL( $F$ ,  $stack$ )  
    if  $F = \{ \}$  then  
        print  $stack$   
        return true  
    end if  
    if  $\{ \} \in F$  then return false  
    choose  $p \in \bigcup F$   
    return DPLL( $F[p \leftarrow \text{true}]$ , PUSH( $p$ ,  $stack$ ))  
        or DPLL( $F[p \leftarrow \text{false}]$ , PUSH(NEGATE( $p$ ),  $stack$ ))  
end function
```

The search for a satisfying valuation of a propositional formula.

The DPLL Algorithm

Furthermore, the algorithm actually contains the optimizations of the DP algorithm.

```
function DPLL( $F$ )  
  if  $F = \{ \}$  then return true  
  if  $\{ \} \in F$  then return false  
  if there is some literal  $L$  and  $C \in F$  with  $C = \{L\}$  then                                 $\triangleright$  unit propagation  
    remove from  $F$  every clause that contains  $L$  and from every clause in  $F$  the negation of  $L$   
    return DPLL( $F$ )  
  else if there is a literal  $L$  such that no clause in  $F$  contains its negation then           $\triangleright$  pure literal elimination  
    remove from  $F$  every clause that contains  $L$   
    return DPLL( $F$ )  
  else                                                                                               $\triangleright$  split  
    choose  $p \in \bigcup F$   
    return DPLL( $F[p \leftarrow \text{true}]$ ) or DPLL( $F[p \leftarrow \text{false}]$ )  
  end if  
end function
```

This is the logical core of modern SAT solvers.

The DPLL Algorithm in OCaml

```
let rec dpll clauses =  
  if clauses = [] then true else if mem [] clauses then false else  
  try dpll(one_literal_rule clauses) with Failure _ ->  
  try dpll(affirmative_negative_rule clauses) with Failure _ ->  
  let pvs = filter positive (unions clauses) in  
  let p = maximize (posneg_count clauses) pvs in  
  dpll (insert [p] clauses) or dpll (insert [negate p] clauses);;  
  
let dpll_sat fm = dpll(defcnfs fm);;  
let dpll_taut fm = not(dpll_sat(Not fm));;  
  
dpll_taut << (p ==> (q ==> r)) /\ (p ==> q) /\ p ==> r >> ;;  
# - : bool = true
```

While DPLL is faster than DP, some crucial optimizations are still missing.

The DPLL Algorithm: Iterative Version

Actually, the algorithm is implemented *iteratively* by using a *stack* (“trail”).

```
function DPLL( $F$ )  
   $stack \leftarrow \text{EMPTY}$   
  BCP( $F, stack, conflict$ )  
  if  $conflict$  return false  
  while  $\exists p. \text{UNASSIGNED}(F, stack, p)$  do  
    choose  $p$  with  $\text{UNASSIGNED}(F, stack, p)$   
    PUSH( $\langle p, \text{guessed} \rangle, stack$ )  
    BCP( $F, stack, conflict$ )  
    if  $conflict$  then  
       $dlevel \leftarrow \text{ANALYZECONFLICT}(F, stack)$   
      if  $dlevel < 0$  return false  
      BACKTRACK( $F, stack, dlevel$ )  
    end if  
  end while  
  return true  
end function
```

```
function ANALYZECONFLICT( $F, stack$ )  
   $dlevel \leftarrow \text{SIZE}(stack) - 1$   
  loop  
    if  $dlevel < 0$  return  $dlevel$   
     $\langle p, t \rangle \leftarrow \text{ELEMAT}(stack, dlevel)$   
    if  $t = \text{guessed}$  return  $dlevel$   
     $dlevel \leftarrow dlevel - 1$   
  end loop  
end function  
  
procedure BACKTRACK( $F, \uparrow stack, dlevel$ )  
  repeat  
     $\langle p, t \rangle \leftarrow \text{POP}(stack)$   
    until  $\text{SIZE}(stack) = dlevel - 1$   
    PUSH( $\langle \text{NEGATE}(p), \text{deduced} \rangle, stack$ )  
end procedure
```

Stack of pairs $\langle p, t \rangle$ with literal p and tag $t \in \{\text{guessed}, \text{deduced}\}$.

The DPLL Algorithm: Auxiliary Functions

procedure BCP($F, \uparrow stack, \uparrow conflict$)

...

end procedure

function UNASSIGNED($F, stack, p$)

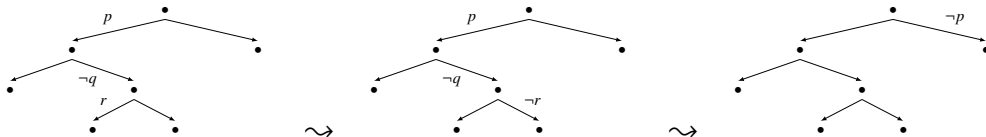
...

end function

- BCP($F, stack, conflict$): **binary constraint propagation**.
 - Repeatedly applies unit propagation deducing the truth values of literals.
 - Pushes pairs $\langle p, \text{deduced} \rangle$ on stack.
 - Sets *conflict* to true if a conflict is detected.
 - The last literal pushed on the stack conflicts another literal on the stack.
- UNASSIGNED($F, stack, p$)
 - Returns true if p is a literal of F that does not appear (neither positively nor negatively) on *stack*.

The explicit use of a stack allows various optimization techniques.

The DPLL Algorithm: Iterative Version



$\langle p, \text{guessed} \rangle \rightarrow \langle \neg q, \text{deduced} \rangle \rightarrow \langle r, \text{guessed} \rangle$
 $\leadsto \langle p, \text{guessed} \rangle \rightarrow \langle \neg q, \text{deduced} \rangle \rightarrow \langle \neg r, \text{deduced} \rangle$
 $\leadsto \langle \neg p, \text{deduced} \rangle$

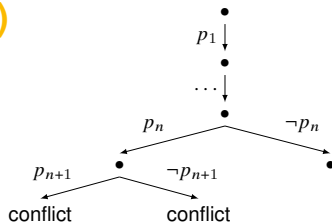
Traversal of tree where backtracking skips the deduced literals.

The Iterative Version of DPLL in OCaml

```
type trailmix = Guessed | Deduced;;
let rec backtrack trail =
  match trail with (p,Deduced)::tt -> backtrack tt | _ -> trail;;
let rec dpli cls trail =
  let cls',trail' = unit_propagate (cls,trail) in
  if mem [] cls' then
    match backtrack trail with
      (p,Guessed)::tt -> dpli cls ((negate p,Deduced)::tt)
    | _ -> false
  else
    match unassigned cls trail' with
      [] -> true
    | ps -> let p = maximize (posneg_count cls') ps in
              dpli cls ((p,Guessed)::trail');;
let dplisat fm = dpli (defcnfs fm) [];;
let dplitaut fm = not(dplisat(Not fm));;

# dplitaut << (p ==> (q ==> r)) /\ (p ==> q) /\ p ==> r >> ;;
- : bool = true
```

Conflict-Driven Clause Learning (CDCL)



An optimization of DPLL that combines “learning” with “backjumping”.

- **Clause Learning:** DPLL backtracks to $p_1 \rightarrow \dots \rightarrow p_n$ to continue with $\neg p_n$.
 - Thus trail $p_1 \rightarrow \dots \rightarrow p_n$ determines an unsatisfying valuation of F .
 - We have **learned clause** $C = \{\neg p_1, \dots, \neg p_n\}$ with property $F \equiv F \cup \{C\}$.
 - Before backtracking, we may add C to F (only using the guessed literals of C).
- **Non-Chronological Backjumping:** backtrack not only to $p_1 \rightarrow \dots \rightarrow p_n$.
 - Determine subset $S \subseteq \{p_1, \dots, p_{n-1}\}$ of guessed literals such that $S \cup \{p_n\}$ is unsatisfying.
 - Backjump to shortest path $p_1 \rightarrow \dots \rightarrow p_{i < n}$ that contains S and extend it by $\neg p_n$.
 - Learned clause $\{\neg p \mid p \in S\} \cup \{\neg p_n\}$.

Backjumping may prune the search tree substantially.

Clause Learning: Example

$\{\{\neg p, \neg q, \neg r\}, \{\neg p, \neg q, r\}, \{\neg p, q, \neg r\}, \{\neg p, q, r\}, \{p, \neg q, \neg r\}, \{p, \neg q, r\}, \{p, q, \neg r\}, \{p, q, r\}\}$

$stack = \langle p, \text{guessed} \rangle \rightarrow \langle q, \text{guessed} \rangle \rightarrow \langle \neg r, \text{deduced} \rangle : \text{conflict}$

$\{\{\neg p, \neg q, \neg r\}, \{\neg p, \neg q, r\}, \{\neg p, q, \neg r\}, \{\neg p, q, r\}, \{p, \neg q, \neg r\}, \{p, \neg q, r\}, \{p, q, \neg r\}, \{p, q, r\}, \{\neg p, \neg q\}\}$

$stack = \langle p, \text{guessed} \rangle \rightarrow \langle \neg q, \text{deduced} \rangle \rightarrow \langle \neg r, \text{deduced} \rangle : \text{conflict}$

$\{\{\neg p, \neg q, \neg r\}, \{\neg p, \neg q, r\}, \{\neg p, q, \neg r\}, \{\neg p, q, r\}, \{p, \neg q, \neg r\}, \{p, \neg q, r\}, \{p, q, \neg r\}, \{p, q, r\}, \{\neg p, \neg q\}, \{\neg p\}\}$

$stack = \langle \neg p, \text{deduced} \rangle \rightarrow \langle q, \text{guessed} \rangle \rightarrow \langle \neg r, \text{deduced} \rangle : \text{conflict}$

$\{\{\neg p, \neg q, \neg r\}, \{\neg p, \neg q, r\}, \{\neg p, q, \neg r\}, \{\neg p, q, r\}, \{p, \neg q, \neg r\}, \{p, \neg q, r\}, \{p, q, \neg r\}, \{p, q, r\}, \{\neg p, \neg q\}, \{\neg p\}, \{\neg q\}\}$

$stack = \langle \neg p, \text{deduced} \rangle \rightarrow \langle \neg q, \text{deduced} \rangle \rightarrow \langle \neg r, \text{deduced} \rangle : \text{conflict}$

$stack = [] : \text{unsat}$

Non-Chronological Backjumping: Example

$$F[x_1, \dots, x_9] \cup \{\{\neg x_2, \neg x_9, x_{10}\}, \underline{\{\neg x_2, \neg x_9, \neg x_{10}\}}\}$$

$stack = \langle x_1, \text{guessed} \rangle \rightarrow \langle x_2, \text{guessed} \rangle \rightarrow \dots \rightarrow \langle x_9, \text{guessed} \rangle \rightarrow \langle x_{10}, \text{guessed} \rangle : \underline{\text{conflict}}$

$$F[x_1, \dots, x_9] \cup \{\{\neg x_2, \neg x_9, x_{10}\}, \underline{\{\neg x_2, \neg x_9, \neg x_{10}\}}\}$$

$stack = \langle x_1, \text{guessed} \rangle \rightarrow \langle x_2, \text{guessed} \rangle \rightarrow \dots \rightarrow \langle x_9, \text{guessed} \rangle \rightarrow \langle \neg x_{10}, \text{deduced} \rangle : \underline{\text{conflict}}$

$$F[x_1, \dots, x_9] \cup \dots \cup \{\{\neg x_2, \neg x_9\}\} \quad (\text{learn } \underline{\text{minimal}} \text{ conflict clause})$$

$stack = \langle x_1, \text{guessed} \rangle \rightarrow \langle x_2, \text{guessed} \rangle \rightarrow \langle \neg x_9, \text{deduced} \rangle \quad (\text{backjump to level of } x_2)$

...

The DPLL Algorithm with CDCL

procedure BACKTRACK($\uparrow F, \uparrow stack, dlevel$)

repeat

$\langle p, t \rangle \leftarrow \text{POP}(stack)$

until $\text{SIZE}(stack) = dlevel - 1$

$S \leftarrow \text{LITERALS}(F, stack, p)$

$C \leftarrow \{\text{NEGATE}(p) \mid p \in S\} \cup \{\text{NEGATE}(p)\}$

$F \leftarrow F \cup \{C\}$

loop

$\langle p, t \rangle \leftarrow \text{TOP}(stack)$

if $p \in S$ **break**

$\text{POP}(stack)$

end loop

$\text{PUSH}(\langle \text{NEGATE}(p), \text{deduced} \rangle, stack)$

end procedure

▸ $stack$ and p determine conflict

▸ Compute minimal literal set S that also implies conflict

▸ Construct clause C from S

▸ Extend F by learned clause C

▸ Backjump to highest level that contains some literal from S

$\text{LITERALS}(F, stack, p)$ actually computes S from an “implication graph” that records the variable dependencies previously established by BCP (we omit the details).

CDCL in OCaml

```
let rec dplb cls trail =  
  let cls',trail' = unit_propagate (cls,trail) in  
  if mem [] cls' then  
    match backtrack trail with  
      (p,Guessed)::tt ->  
        let trail' = backjump cls p tt in  
        let declits = filter (fun (_,d) -> d = Guessed) trail' in  
        let conflict = insert (negate p) (image (negate ** fst) declits) in  
        dplb (conflict::cls) ((negate p,Deduced)::trail')  
      | _ -> false  
  else  
    match unassigned cls trail' with  
      [] -> true  
    | ps -> let p = maximize (posneg_count cls') ps in  
             dplb cls ((p,Guessed)::trail');;
```

CDCL in OCaml

```
let rec backjump cls p trail =  
  match backtrack trail with  
    (q,Guessed)::tt ->  
      let cls',trail' = unit_propagate (cls,(p,Guessed)::tt) in  
      if mem [] cls' then backjump cls p tt else trail  
  | _ -> trail;;  
  
let dplbsat fm = dplb (defcnfs fm) [];;  
let dplbtaut fm = not(dplbsat(Not fm));;  
  
# dplbtaut << (p ==> (q ==> r)) /\ (p ==> q) /\ p ==> r >> ;;  
- : bool = true
```

Only a simple prototype; modern SAT solvers are heavily optimized with respect to coding techniques, data structures, and many more heuristic improvements.

The SAT Solver MiniSat

We now consider an efficient implementation of DPLL with CDCL.

- **MiniSat:** An open source SAT solver.

<http://minisat.se>

Debian/Ubuntu: `apt-get install minisat`

- Minimalistic but efficient.
 - Winner of the industrial categories of the SAT 2005 competition
 - For true state-of-the art solvers, see <http://www.satcompetition.org>.
 - Lingeling, Plingeling and Treengeling: <http://fmv.jku.at/lingeling>.

Most SAT solvers typically support the same input format.

The DIMACS Format

- **DIMACS**: a standard textual input format for MiniSat and other SAT solvers.

```
c comment
p cnf nv nc
v v ... v 0
...
```

- *comment*: a comment line.
- *nv*: number of variables, *nc*: number of clauses.
 - *nc* lines *v v ... v 0*
- *v*: an integer in the ranges $1, \dots, nv$ respectively $-1, \dots, -nv$.
 - Denotes literal $x_1, \dots, x_{nv}$ or $\neg x_1, \dots, \neg x_{nv}$, respectively.

- **Example**: $x_1 \wedge (\neg x_2 \vee x_3)$

```
c file "example.cnf"
p cnf 3 2
1 0
-2 3 0
```

MiniSat Example

```
debian10!1> minisat example.cnf example.out
```

```
WARNING: for repeatability, setting FPU to use double precision
```

```
===== [ 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) (476 /sec)
propagations       : 1          (476 /sec)
conflict literals  : 0          (-nan % deleted)
Memory used        : 14.00 MB
CPU time           : 0.002101 s
```

```
SATISFIABLE
```

```
debian10!1> cat example.out
```

```
SAT
```

```
1 -2 3 0
```

The SAT Solver Limboole

Another SAT solver that is more suitable for interactive use.

<http://fmv.jku.at/limboole/>

This is a simple boolean calculator. It reads a boolean formula and checks whether it is valid. In case '-s' is specified satisfiability is checked instead of validity (tautology). The input format has the following syntax in BNF: ...

```
expr ::= iff
iff ::= implies { '<-' implies }
implies ::= or [ '->' or | '<-' or ]
or ::= and { '|' and }
and ::= not { '&' not }
not ::= basic | '!' not
basic ::= var | '(' expr ')'
```

and 'var' is a string over letters, digits and the following characters:

- _ . [] \$ @

The last character of 'var' should be different from '-'.

Limboole: Command Line Version

```
debian10!1> limboole -s
x1 & (~x2 | x3)
% SATISFIABLE formula (satisfying assignment follows)
x1 = 1
x2 = 0
x3 = 0
debian10!2> limboole
x1 & (~x2 | x3)
% INVALID formula (falsifying assignment follows)
x1 = 1
x2 = 1
x3 = 0
debian10!4> cat > example.bool
x1 & (-x2 | x3)
alan!355> limboole example.bool
% INVALID formula (falsifying assignment follows)
x1 = 1
x2 = 1
x3 = 0
```

Limboole: Web Version

<https://maximaximal.github.io/limboole>

Limboole on the Go!

Uses [Limboole](#) (MIT licensed), [PicoSAT](#) (MIT licensed), and [DepQBF](#) (GPLv3 licensed) to parse an easy SAT and QBF DSL (instead of relying on DIMACS). Compiled using [Emscripten](#), [Source Code and Modifications are available on GitHub](#). Created by [Max Heisinger](#). I also wrote a short [blog entry](#) about this. Support on GitHub and on [#limboole](#) on [Libera.Chat](#).

Open How-To

Validity Check

Run (or Shift+Enter in input area)

Input Drag&Drop ✓
x1 & (~x2 | x3)

Output
% INVALID formula (falsifying assignment follows)
x1 = 1
x2 = 1
x3 = 0

Errors