Overview

The general DPLL algorithm

```
DPLL(F, α)
  simplify(F, α)
  if F = 0 then return UNSAT
  if F = 1 then return α
  pick x ∈ V(F) and ε ∈ {0,1}
  β := DPLL(F[x := ε], α ∪ [x := ε])
  if β ≠ UNSAT
     then return β
     else return DPLL(F[x := ε̄], α ∪ [x := ε̄])
```

Main program

```
read formula
unit propagation
repeat
    choose literal b
    set value b
    unit propagation
    if conflict detected
        backtrack
    if all clauses satisfied
        output assignment
```

Global variables

- n number of variables
- m number of clauses
- V list of n variables
- F list of m clauses
- q unit queue
- α assignment stack
- d branching depth

Storing clauses and literals

The data structure for variable x contains

- value $\in \{0, 1, \texttt{free}\}$
- list pos_occ of clauses where x occurs
- list neg_occ of clauses where $\bar{x}$ occurs
- branching level dp
- clause reason

The data structure for clause C contains

- Flag sat by literal s
- List lit of literals in C
- Number act of active literals in C

Assigning a value to a variable

To set x to 1:

- update `value` $= 1$
- for every clause C in `pos_occ`

  mark C as `sat` by x, if not `sat`
- for every unsatisfied clause C in `neg_occ`

  decrement `act`

  if `act` $= 1$ then

    find unique free literal a in C

    enqueue a in unit queue q

  if `act` $= 0$ then

    report conflict

Unit propagation

To find unit clauses:

- maintain number of unset literals in clauses.
- decremented when literal set to 0

To propagate units:

- keep literals to be set in a queue q
- while q not empty

    $b :=$ last literal in q

    set value b

Backtracking

Undo the last assignment.

- Assignments performed stored on a stack α

Undo all assignments forced by unit propagation until last branching:

- Assignments on stack marked as `forced` or `branching`
- while $b = \mathtt{pop}(\alpha)$ is forced
  unset value b

  if α empty
  output "Unsatisfiable."

  $b = \mathtt{pop}(\alpha)$

  unset value b

  set value $-b$ as `forced`

  empty q

Unassigning a variable

To undo setting of x to 1:

- update `value` = `free`
- for every clause C in `pos_occ`
  if C satisfied by x
  mark C as `not sat`
- for every unsatisfied clause C in `neg_occ`
  increment `act`

Branching heuristics

$h_k(a) := \quad \#\{\, C\,;\, w(C) = k \text{ and } a \in C \,\}$

$h(a) := \quad \sum_k h_k(a)$

DLIS: Pick literal a with $h(a)$ maximal.

DLCS: Pick variable x with $h(x) + h(\bar{x})$ maximal,
set $[x := 1]$ if $h(x) \geq h(\bar{x})$, and $[x := 0]$ otherwise.

Branching heuristics

The MOM heuristic:

Let $\ell \geq 2$ be the current minimal clause width.

Pick a variable x with $\big(h_\ell(x) + h_\ell(\bar{x})\big)2^\alpha + h_\ell(x)h_\ell(\bar{x})$ maximal

Bohm's heuristic:

Let $H(x) := (H_2(x), \ldots, H_n(x))$,
where $H_k(x) := \alpha \max(h_k(x), h_k(\bar{x}) + \beta \min(h_k(x), h_k(\bar{x}))$.

Pick x with $H(x)$ lexicographically maximal.

The Jeroslaw-Wang heuristic:

Let $J(a) := \sum_{k=1}^{n} h_k(a)2^{-k} \; = \; \sum_{a \in C} 2^{-w(C)}$

Pick literal a with $J(a)$ maximal

Head-Tail lists

Head literal: first unassigned literal in a clause.

Tail literal: last unassigned literal in a clause.

For clauses, keep two pointers

- `head` to the head literal
- `tail` to the tail literal

For variables, keep lists:

- `pos_head_occ` clauses where x occurs as head literal
- `pos_tail_occ` clauses where x occurs as tail literal

Invariant: if `head` in C points to x,

- all literals before x in C are set
- C occurs in `pos_head_occ` in x.

Head-Tail lists

To x set to 1:

- for every clause C in `neg_head_occ`

  find next unassigned literal b

  if literal set to 1 encountered

      abort

  if no unassigned literal found

      report conflict

  if b is tail literal

      enqueue b in unit queue

  add C to head list of b, mark b as head

Similarly for `neg_tail_occ`, and for setting x to 0.

If setting x to 1 is undone:

- if $\bar{x}$ occurs in C before `head`, update lists.

Watched Literals

In every clause C, mark two arbitrary literals as watched,
e.g. by two pointers `watched1` and `watched2`

Instead of head and tail lists:

- list of clause `pos_watched_occ` where x occurs as watched literal.

Invariant: While C is not satisfied,

- both watched literals in C are unset.
- if x is watched in C, then C occurs in `pos_watched_occ` in x

Watched Literals

To x set to 1:

- for every clause C where $\bar{x}$ occurs as watched literal

  find some other unassigned literal b

  if literal set to 1 encountered

  abort

  if no unassigned literal found

  report conflict

  if only b found is watched

  enqueue b in unit queue q

  add C to watch list of b, mark b as watched

When setting x to 1 is undone, watched literals can be kept.

Branching depth

Branching depth of an assignment $[a := 1]$:

- number of branching asignments on stack below $[a := 1]$

Implemented by a global counter `bd`

- incremented at each branching assignment
- decremented on backtracking

Implication graph

Directed acyclic graph representing implications between assignments.

For every assignment $[x := \epsilon]$ of branching depth d

- create vertex $v(x)$ labelled (x, ϵ, d)

Branching assignment: source vertex

Assignment $[x := \epsilon]$ forced by unit propagation:

- Clause $x^{\epsilon} \vee y_1^{\delta_1} \vee \ldots \vee y_k^{\delta_k}$ in F
- variables y_i assigned values $(1 - \delta_i)$ at depth $d_i \leq d$
- vertices $v(y_i)$ labelled $(y_i, 1 - \delta_i, d_i)$ already present
- insert edges from $v(y_i)$ to $v(x)$

Conflict in the implication graph

At a conflict create conflict vertex $v(\square)$ labelled $(\square, d)$

- clause $y_1^{\delta_1} \vee \ldots \vee y_k^{\delta_k}$ empty
- variables y_i assigned values $(1 - \delta_i)$ at depth $d_i \leq d$
- vertices $v(y_i)$ labelled $(y_i, 1 - \delta_i, d_i)$ already present
- insert edges from $v(y_i)$ to $v(\square)$.

From now on:

- consider only the part of the implication graph from which $v(\square)$ is reachable.

Implementing the implication graph

Assignment vertices:

- with variables set store branching depth

Edges:

- with variables set store reason for the setting:
  the clause triggering the unit propagation.
- `NULL` for branching assignments

Conflict vertex and edges to it:

- at a conflict, store the clause that became empty.

Cuts and conflict clauses

A cut in the implication graph:

- partition into two disjoint sets B and C
- the branching side B is downward closed and contains all branching literals.
- the conflict side is upward closed and contains the conflict node.

A cut defines a conflict clause $\quad y_1^{\delta_1} \vee \ldots \vee y_k^{\delta_k}$
where $v(y_i) = (y_i, 1 - \delta_i, d_i)$ are the vertices in B with an edge into C.

The resolution rule

The resolution rule:

from $C \vee a$ and $D \vee \bar{a}$ derive $C \vee D$.

Theorem

If C is derived from F by resolution,
then F is satisfiable iff $F \wedge C$ is satisfiable.

Fact: conflict clauses are derived by resolution.

Corollary

Adding conflict clauses does not change satisfiablity.

Asserting clauses and backtracking

A conflict clause C is asserting, if it contains exactly one literal of maximal branching depth.

The assertion level of C is the second largest branching depth of literals in C.

Backtracking procedure:

- at a conflict, find a cut in the implication graph giving an asserting conflict clause C
- add C to the formula (learn C), let d be its assertion level
- undo all assignments of branching level $> d$ set branching depth to d
- now C is a unit clause a
- enqueue a, goto unit propagation

Differences to DPLL

- Assertion level can be smaller than the maximal level -1

  $\rightsquigarrow$ non-chronological backtracking

- The literal that is flipped can be an implied literal

  $\rightsquigarrow$ not modelled by DPLL recursion

- Added conflict clause avoids finding the same conflict again.

TODO: methods to find asserting conflict clauses

$\rightsquigarrow$ learning scheme

The *RelSAT* and *decision* schemes

The *RelSAT* scheme:

Let d be the current branching depth.

- C: all vertices of depth d, except the branching vertex.
- B: the branching vertex of depth d, all vertices of depth $< d$.

The *decision* scheme:

- B: all branching vertices (from which $v(\Box)$ can be reached).
- C: all other vertices, i.e., implied vertices and $v(\Box)$.

Unique implication points

A unique implication point (UIP) is a vertex v
of maximal branching depth with

- every path from the last branching vertex to the conflict vertex goes through v.

The branching vertex is a UIP, so there exists at least one.

The cut corresponding to a UIP v

- C: all vertices on paths between v and $v(\Box)$

defines an asserting conflict clause.

The *1UIP* scheme

The *1UIP* scheme:

always learn the asserting conflict clause
obtained from the cut at the first UIP (from $v(\Box)$).

Computing the 1UIP conflict clause:

let C be the conflict clause

while C is not asserting

    let D be the reason clause of the next assignment on the stack

    let C be the resolvent of C with D

The VSIDS heuristic

The variable state independent decaying sum (VSIDS) heuristic:

- Every literal a has a priority $s(a)$, initially $h(a)$, and a counter $r(a)$, initially 0.
- Heuristic picks a literal of highest priority, with ties broken randomly.
- Literals stored in a priority queue for fast finding of maximum.
- When clause C is learned, counters $r(a)$ of literals a in C incremented.
- Periodically (every 255 branchings) all priorities updated:
  $s(a) := s(a)/2 * r(a)$
  $r(a) := 0$

Thus: VSIDS picks literals that ocurred in many recent conflict clauses.

The BerkMin heuristic

The following heuristic is implemented in BerkMin:

- Clauses are ordered in the order of being added.
- Literals have a priority $n(a)$.
- Heuristic picks a literal of highest priority from the unassigned literals in the most recent clause.
- In conflict analysis, $n(a)$ is incremented for all literals in clauses in the derivation of the conflict clause.
- Periodically, all priorities updated: $n(a) := n(a)/4$.

The VMTF heuristic

The variable move to front (VMTF) heuristic:

- Literals have a counter $n(a)$, initialized as $h(a)$
- Literals are stored in an ordered list L,
  initially sorted by decreasing $n(a)$.
- Heuristic picks earliest unassigned literal from L.
- When clause C is learned,

  $n(a)$ is incremented for all a in C

  the $min(C, |8|)$ literals in C with $n()$ largest
  are moved to the front of L

Simple clause deletion strategies

Learned clauses need to be deleted (forgotten), otherwise:

- solver runs out of memory
- unit propagation costs too much time

k-bounded learning

- Clauses C of width $w(C) \leq k$ are kept indefinitely.
- Larger clauses C are deleted as soon as 2 literals in C are unassigned.

m-size relevance based learning

- Clauses C are deleted as soon as more than m literals in C are unassigned.

Both strategies can be combined.

BerkMin's clause deletion strategies

The following strategy is implemented in BerkMin:

- Clauses are ordered in the order of being added.
- Clauses have an activity counter $n(C)$.
- $n(C)$ is increased when C contributes to a conflict.
- A clause is old if it is among the first $1/16$ of the learned clauses, otherwise young.
- A young clause C is deleted if $w(C) > 42$ and $n(C) \leq 7$.
- An old clause is deleted if $w(C) > 8$ and $n(C) \leq t$.
- The threshold value t is initially 60, then gradually increased.

Restarts

Periodically, CDCL solvers do a restart after a conflict:

- empty the assignment stack
- undo all assignments
- keep learned clauses and scores for branching heuristics

Many solvers restart after a fixed number of conflicts.

Problem: completeness!

Completeness can be preserved by:

- increasing intervals between restarts.
- or guaranteeing to keep some learned clauses between any two restarts.

Restart policies

Fixed interval policy:

- restart after a fixed number c of conflicts
- siege: $c = 16.000$, Chaff 2004 $c = 700$, BerkMin $c = 550$

Geometric policy:

- restart after c conflicts, then multiply by a factor $c := c \cdot f$
- MiniSat: $t = 100$, $f = 1,5$

Luby policy:

- Define the Luby sequence $t_1, t_2, \ldots$ by
  $t_i = 2^{k-1}$ if $i = 2^k - 1$, $\qquad t_i = t_{i-2^{k-1}+1}$ if $2^{k-1} \leq i < 2^k - 1$
- The first values are $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$
- Fix $c = 32$. The ith restart is performed $c \cdot t_i$ conflicts after the previous restart.

Phase Saving

Counterintuitive heuristics:

- for each variable, remember the value it was assigned at the time of restart.
- when a variable is selected as branching variable, assign the stored value again.

Preprocessing

Expensive reductions are still worthwile in a preprocessing phase:

- Pure literal elimination
- Deleting subsumed clauses

Modern solvers use more reduction in preprocessing.

Equivalence substitution:

If F contains clauses $(a \vee \bar{b})$ and $(\bar{a} \vee b)$

- replace b by a everywhere
- delete these clauses

Efficient Subsumption Testing

Let h be a hash function from literals to $\{0, \ldots, 63\}$.

With every clause C, store a fingerprint

$sig(C) := \bigvee_{a \in C} 2^{h(a)}$.

Subsumption test:

$subsumes(C_1, C_2)$
  if $sig(C_1) \wedge \neg sig(C_2) \neq 0$ then
    return false
  else
    return $C_1 \subseteq C_2$

Variable Elimination Resolution (VER)

Resolution operator:

Let $C = C' \vee x$ and $D = D' \vee \bar{x}$, then $Res_x(C, D) := C' \vee D'$.

Elimination of variable x:

Decompose $F = F^- \cup F_x \cup F_{\bar{x}}$, where $F_a := \{C \in F\,;\, a \in C\}$.

Let $F_x \otimes F_{\bar{x}} := \{Res_x(C, D)\,; C \in F_x \text{ and } D \in F_{\bar{x}}\}$

$VER(x)$ replaces $F_x \cup F_{\bar{x}}$ with $F_x \otimes F_{\bar{x}}$, with tautologies omitted.

Basic building block of classical Davis-Putnam-algorithm.

NiVER: Non-increasing VER

Variable x is only eliminated if formula not enlarged.

NiVER(x)

$S := \emptyset$
for $C \in F_x$ and $D \in F_{\bar{x}}$ do
    $R := Res_x(C, D)$
    if R not tautology
        $S := S \cup \{R\}$

if $size(S) \leq size(F_x \cup F_{\bar{x}})$
    $F := F^{-} \cup S$
    change := true

- Measure $size(F)$ can be

- number of clauses
- number of literal occurrences

NiVER: Non-increasing VER

NiVER is applied for all variables,

and iterated until no more variables can be eliminated.

```
while(change) do
    change := false
    for x ∈ V(F)
        NiVER(x)
```

Self-Subsuming Resolution

Theorem

Let $C = C' \vee a$ be a clause in $F = F' \wedge C$.

If there is a clause $D \in F'$ with $D \setminus \bar{a} \subseteq C'$, then F is satisfiable iff $F' \wedge C'$ is.

Proof: $Res_a(C, D) = C'$ subsumes $C = C' \vee a$.

Terminology: C is strengthened by self-subsumption using D.

Conflict Clause Minimization

Self-subsuming Resolution is also used for minimizing conflict clauses, e.g. in MiniSAT.

```
minimizeCC(C)
  for a ∈ C do
    if reason(ā) \ ā ⊆ C
      mark a
  remove marked literals from C
```

Variable Elimination by Substitution

Idea: make use of definition of variables in Tseitin transformation.

E.g. transforming $x = a \wedge b$ gives clauses

$$x \vee \bar{a} \vee \bar{b}, \quad \bar{x} \vee a, \quad \bar{x} \vee b$$

Elimination of x then generates many redundant clauses.

The same holds for many other clauses originating from transforming logic gates.

Variable Elimination by Substitution

$G :=$ clauses from the definition of x

$R :=$ other clauses containing x

Now $(G_x \cup R_x) \otimes (G_{\bar{x}} \cup R_{\bar{x}})$ can be decomposed into

- $S' := \; G_x \otimes R_{\bar{x}} \; \cup \; R_x \otimes G_{\bar{x}}$
- $G' := G_x \otimes G_{\bar{x}}$
- $R' := R_x \otimes R_{\bar{x}}$

Now we have:

- G' contains only tautologies
- all clauses in R' are derived from clauses in S'

Thus: only need to consider S' in elimination of x.

SatELite Preprocessor

SatELite iterates the following sequence of operations,
until no more changes happen:

In every round, do:

repeat
    strengthen clauses by self-subsumption
    unit propagation
until no more clauses are strengthened

remove subsumed clauses

for all variables x do
    NiVER(x)

NiVER(x) here uses the optimization for variables having definitions.

SatELite Preprocessor

Further optimizations to speed up SatELite:

- Clauses are only tested for subsumption
  if they were added in the previous round.
- Clauses are only tested for self-subsumption
  if they were added or strengthened recently.
- NiVER is only applied to variables occurring in clauses
  that were added, strengthened or removed recently.
- Recently: in the previous round, or earlier in the current round.
- NiVER is not applied to variables x with $\min(h(x), h(\bar{x})) > 10$.
  Heuristically shown to be not worthwile.

Failed Literal Probing

Test for settings that immediately imply conflicts:

FLP(a)

set $[a \leftarrow 1]$

unitProp()

if conflict found

    add unit clause $\bar{a}$

FLP is iterated for all literals, until no more change.

Blocked Clause Elimination

Definition: Literal $a \in C$ blocks C,
if $Res_a(C, D)$ is a tautology for all clauses $D \ni \bar{a}$.

Clause C is blocked in F, if some literal $a \in C$ blocks it.

Theorem

If C is blocked in F, then F is satisfiable iff $F \setminus C$ is satisfiable.

Theorem

If C and C' are both blocked in F, then C' is blocked in $F \setminus C$.

$\rightsquigarrow$ BCE is confluent.

What about pure literals?

Fact: NiVER performs pure literal elimination.

If a is pure in F, then $F_a \otimes F_{\bar{a}} = \emptyset$,
so $|F_a \otimes F_{\bar{a}}| = 0 \leq |F_a \cup F_{\bar{a}}|$.

Fact: BCE performs pure literal elimination.

If a is pure and $a \in C$, then a blocks C.

Inprocessing

Some preprocessing techniques are uesful, but still too expensive.

- Preempt preprocessing after some time
- Resume preprocessing between restarts
- Limit preprocessing time vs. search time ($\sim$ 20% : 80%)

Additional benefit:

- allows to use learned clauses for preprocessing.