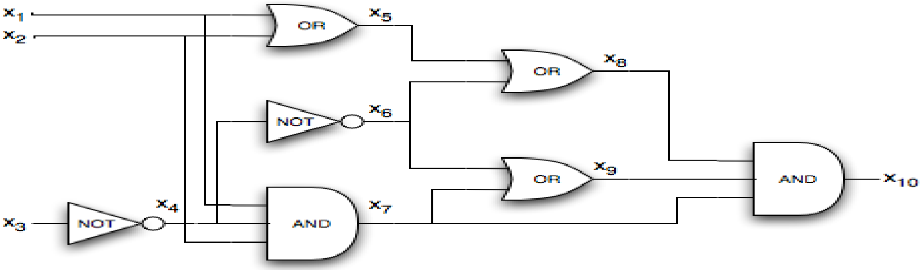
Karlsruhe Institute of Technology

Practical SAT Solving

Lecture 5

Carsten Sinz, Tomáš Balyo | May 21, 2019

INSTITUTE FOR THEORETICAL COMPUTER SCIENCE



Stochastic Local Search (SLS)

SAT as an optimization problem: minimize the number of unsatisfied clauses

Start with a complete random assignment α:

0	0	1	0	1	1	0	0	1	1	0	1	0	1	0

Repeatedly flip (randomly/heuristically chosen) variables to decrease the number of unsatisfied clauses:

0	0	1	0	1	0	0	0	1	1	0	1	0	1	0

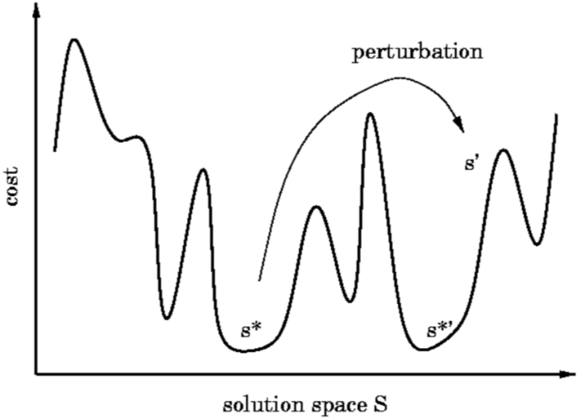
SLS Algorithms

- Local search algorithms are **incomplete**: they cannot show unsatisfiability!
- Many variants of local search algorithms
- Main question: Which variable should be flipped next?
  - select variable from an unsatisfied clause
  - select variable that increases the number of satisfied clauses most
- How to avoid local minima?

GSAT Algorithm [1]

```
Maybe[Assignment] GSAT(ClauseSet S)
{
  for i = 1 to MAX_TRIES do {
    α = random-assignment to variables in S
    for j = 1 to MAX_FLIPS do {
      if ( α satisfies all clauses in S ) return α
      x = variable that produces least number of
        unsatisfied clauses when flipped
      flip x
    }
  }
  return Nothing   // no solution found
}
```

SLS: Illustration



[Source: Alan Mackworth, UBC, Canada]

Walksat [2]

- Variant of GSAT
- Try to avoid local minima by introducing “random noise”
  - Select unsatisfied clause C at random
  - If by flipping a variable $x \in C$ no new unsatisfied clauses emerge, flip x
  - Otherwise:
    - With probability p select a variable $x \in C$ at random
    - With probability $1 - p$ select a variable that changes as few as possible clauses from satisfied to unsatisfied when flipped

SLS: Important Notions

- Consider a flip taking α to α'
- **breakcount:** number of clauses satisfied in α, but not in α'
- **makecount:** number of clauses unsatisfied in α, but satisfied in α'
- **diffscore:** number of unsatisfied clauses in α minus number of clauses unsatisfied in α'
- Typically, **breakcount**, **makecount** and **diffscore** are updated after each flip
- Question: How can we do this efficiently?

GSAT and Walksat Flip Heuristics

- GSAT: select variable with highest **diffscore**
- Walksat:
  - First randomly select unsatisfied clause C
  - If there is a variable with **breakcount** 0 in C, select it
  - otherwise with probability p select a random variable from C, and with probability $1 - p$ a variable with minimal **breakcount** from C

Runtime Comparison Walksat vs. GSAT

formula			DP	GSAT+w	WSAT
id	vars	clauses	time	time	time
2bitadd_12	708	1702	*	0.081	0.013
2bitadd_11	649	1562	*	0.058	0.014
3bitadd_32	8704	32316	*	94.1	1.0
3bitadd_31	8432	31310	*	456.6	0.7
2bitcomp_12	300	730	23096	0.009	0.002
2bitcomp_5	125	310	1.4	0.009	0.001

Table 4: Comparing an efficient complete method (DP) with local search strategies on circuit synthesis problems. (Timings in seconds.)

formula			DP	GSAT+w	WSAT
id	vars	clauses	time	time	time
ssa7552-038	1501	3575	7	129	2.3
ssa7552-158	1363	3034	*	90	2
ssa7552-159	1363	3032	*	14	0.8
ssa7552-160	1391	3126	*	18	1.5

Table 5: Comparing DP with local search strategies on circuit diagnosis problems by Larrabee (1989). (Timings in seconds.)

[Source: Selman, Kautz, Cohen Local Search Strategies for Satisfiability Testing, 1993]

Repetition: Resolution/Saturation

Saturation Algorithm

- INPUT: CNF formula F
- OUTPUT: $\{SAT, UNSAT\}$

while (true) **do**
  $R = \text{resolveAll}(F)$
  if $(R \cap F \neq R)$ **then** $F = F \cup R$
  else break
if $(\bot \in F)$ **then return** *UNSAT* **else return** *SAT*

Properties of the saturation algorithm:

- it is sound and complete – always terminates and answers correctly
- has exponential time and space complexity

Can we do better?

- Question: Can we do better than saturation-based resolution?
  - Avoid exponential space complexity
  - Improve average-case complexity (for important problem classes)

Davis-Putnam Algorithm [3]

- Presented in 1960 as a procedure for first-order (predicate) logic
- Procedure to check satisfiability of a formula F in CNF
- Three (deduction) rules:
  1. **Unit propagation:** if there is a unit clause $C = \{l\}$ in F, simplify all other clauses containing l
  2. **Pure literal elimination:** If a literal l never occurs negated in F, add the clause $\{l\}$ to F
  3. **Case splitting:** Assume that F is put in the form $(A \vee l) \wedge (B \vee \bar{l}) \wedge R$, where A, B, and R are free of l. Replace F by the clausification of $(A \vee B) \wedge R$
- Apply deduction rules (giving priority to rules 1 and 2) until no further rule is applicable

From Davis' and Putnam's Paper

Karlsruhe Institute of Technology

> The superiority of the present procedure over those previously available is indicated in part by the fact that a formula on which Gilmore's routine for the IBM 704 causes the machine to compute for 21 minutes without obtaining a result was worked successfully by hand computation using the present method in 30 minutes.

DPLL Algorithm: Outline

- DPLL: Davis-Putnam-Logemann-Loveland [4]
- Algorithmic improvements over DP algorithm
- Basic idea: case splitting and simplification
- Simplification: unit propagation and pure literal deletion
- Unit propagation: 1-clauses (unit clauses) fix variable values: if $\{x\} \in S$, in order to satisfy S, variable x must be set to 1.
- Pure literal deletion: If variable x occurs only positively (or only negatively) in S, it may be fixed, i.e. set to 1 (or 0).

Pure Literal Deletion: Example

Karlsruhe Institute of Technology

- Let $F_0 = \{\{x, y\}, \{\neg x, y, \neg z\}, \{\neg x, z, u\}, \{x, \neg u\}\}$.
- All clauses containing y may be deleted, as y occurs only positively in F. This yields:

$$F_1 = \{\{\neg x, z, u\}, \{x, \neg u\}\}$$

- Each solution α_1 of F_1 can be extended to a solution α_0 of F_0 by setting $\alpha_0(y) = 1$.
- Moreover, if F_1 does not possess a solution, then so does F_0.
- Repeating yields $F_2 = \{\{x, \neg u\}\}$ and $F_3 = \emptyset$, thus F_0 is satisfiable.

DPLL Algorithm

```
boolean DPLL(ClauseSet S)
{
  while ( S contains a unit clause {L} ) {
    delete from S clauses containing L;  // unit-subsumption
    delete ¬L from all clauses in S;     // unit-resolution
  }
  if ( ⊥ ∈ S ) return false;            // empty clause?
  while ( S contains a pure literal L )
    delete from S all clauses containing L;
  if ( S = ∅ ) return true;             // no clauses?
  choose a literal L occurring in S;     // case-splitting
  if ( DPLL(S ∪ {{L}} ) return true;     // first branch
  else if ( DPLL(S ∪ {{¬L}} ) return true;  // second branch
  else return false;
}
```

DPLL: Implementation Issues

- How can we implement unit propagation efficiently?
- Which literal L to use for case splitting?
- How can we efficiently implement the case splitting step?

“Modern” DPLL Algorithm with “Trail”

boolean `mDPLL(`**ClauseSet** S, **PartialAssignment** α)
{
  while ((S, α) contains a unit clause $\{L\}$) **{**
    add $\{L = 1\}$ to α
  }
  if (a literal is assigned both 0 and 1 in α) **return** false;
  if (all literals assigned) **return** true;
  choose a literal L not assigned in α occurring in S;
  if (mDPLL(S, $\alpha \cup \{L = 1\}$) **return** true;
  else if (mDPLL(S, $\alpha \cup \{L = 0\}$) **return** true;
  else return false;
}

(S, α): clause set S as “seen” under partial assignment α

DPLL: Implementation Issues

- How can we implement unit propagation efficiently?
- (How can we implement pure literal elimination efficiently?)
- Which literal L to use for case splitting?
- How can we efficiently implement the case splitting step?

Properties of a good decision heuristic

Properties of a good decision heuristic

- Fast to compute
- Yields efficient sub-problems
  - More short clauses?
  - Less variables?
  - Partitioned problem?

Bohm's Heuristic

- Best heuristic in 1992 for random SAT (in the SAT competition)
- Select the variable x with the maximal vector $(H_1(x), H_2(x), \dots)$

$$H_i(x) = \alpha \max(h_i(x), h_i(\overline{x})) + \beta \min(h_i(x), h_i(\overline{x}))$$

- where $h_i(x)$ is the number of not yet satisfied clauses with i literals that contain the literal x.
- α and β are chosen heuristically ($\alpha = 1$ and $\beta = 2$).
- Goal: satisfy or reduce size of many preferably short clauses

MOMS Heuristic

- Maximum Occurrences in clauses of Minimum Size
- Popular in the mid 90s
- Choose the variable x with a maximum $S(x)$.

$$S(x) = (f^*(x) + f^*(\overline{x})) \times 2^k + (f^*(x) \times f^*(\overline{x}))$$

- where $f^*(x)$ is the number of occurrences of x in the smallest not yet satisfied clauses, k is a parameter
- Goal: assign variables with high occurrence in short clauses

Jeroslow-Wang Heuristic

- Considers all the clauses, shorter clauses are more important
- Choose the literal x with a maximum $J(x)$.

$$J(x) = \sum_{x \in c, c \in F} 2^{-|c|}$$

- Two-sided variant: choose variable x with maximum $J(x) + J(\overline{x})$
- Goal: assign variables with high occurrence in short clauses
- Much better experimental results than Bohm and MOMS
- One-sided version works better

(R)DLCS and (R)DLIS Heuristics

- (Randomized) Dynamic Largest (Combined | Individual) Sum
- Dynamic = Takes the current partial assignment in account
- Let C_P (C_N) be the number of positive (negative) occurrences
- DLCS selects the variable with maximal $C_P + C_N$
- DLIS selects the variable with maximal $\max(C_P, C_N)$
- RDLCS and RDLIS does a random selection among the best
  - Decrease greediness by randomization
- Used in the famous SAT solver GRASP in 2000

LEFV Heuristic

- Last Encountered Free Variable
- During unit propagation save the last unassigned variable you see, if the variable is still unassigned at decision time use it otherwise choose a random
- Very fast computation: constant memory and time overhead
  - Requires 1 int variable (to store the last seen unassigned variable)
- Maintains search locality
- Works well for pigeon hole and similar formulas

How to Implement Unit Propagation

The Task

Given a partial truth assignment ϕ and a set of clauses F identify all the unit clauses, extend the partial truth assignment, repeat until fix-point.

Simple Solution

- Check all the clauses
- Too slow
- Unit propagation cannot be efficiently parallelized (is P-complete)

How to Implement Unit Propagation

The Task

Given a partial truth assignment ϕ and a set of clauses F identify all the unit clauses, extend the partial truth assignment, repeat until fix-point.

Simple Solution

- Check all the clauses
- Too slow
- Unit propagation cannot be efficiently parallelized (is P-complete)

In the context of DPLL the task is actually a bit different

- The partial truth assignment is created incrementally by adding (decision) and removing (backtracking) variable value pairs
- Using this information we will avoid looking at all the clauses

How to Implement Unit Propagation

The Real Task

We need a data structure for storing the clauses and a partial assignment ϕ that can efficiently support the following operations

- detect new unit clauses when ϕ is extended by $x_i = v$
- update itself by adding $x_i = v$ to ϕ
- update itself by removing $x_i = v$ from ϕ
- support restarts, i.e., un-assign all variables at once

Observation

- We only need to check clauses containing x_i

Occurrences List and Literals Counting

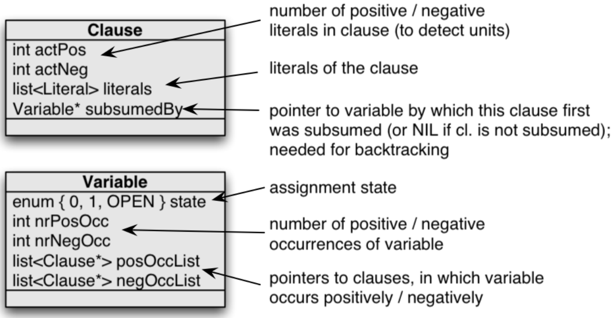
The Data Structure

- For each clause remember the number unassigned literals in it
- For each literal remember all the clauses that contain it

Operations

- If $x_i = T$ is the new assignment look at all the clauses in the occurrence of of $\overline{x_i}$. We found a unit if the clause is not SAT and counter=2
- When $x_i = v$ is added or removed from ϕ update the counters

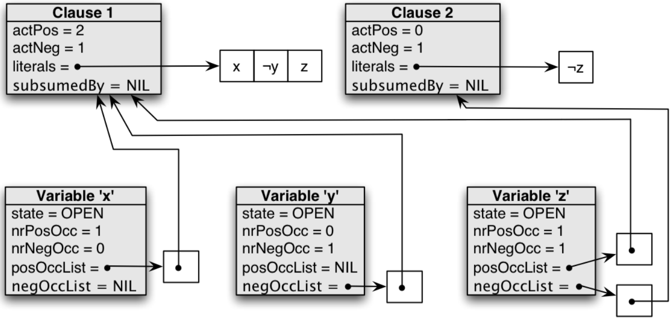
"Traditional" Approach



Crawford, Auton (1993)

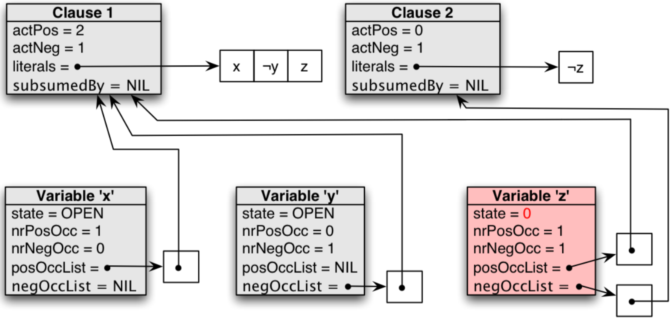
Traditional Approach: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$



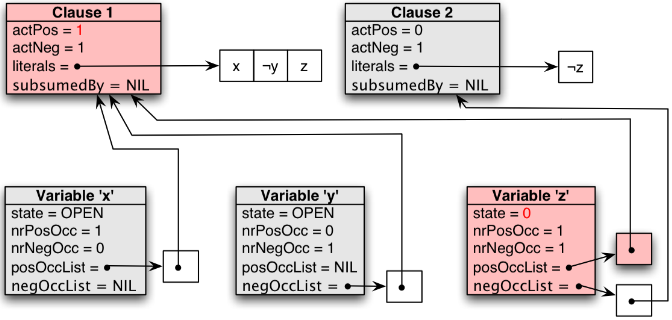
Traditional Approach: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ unit propagation: set $z = 0$



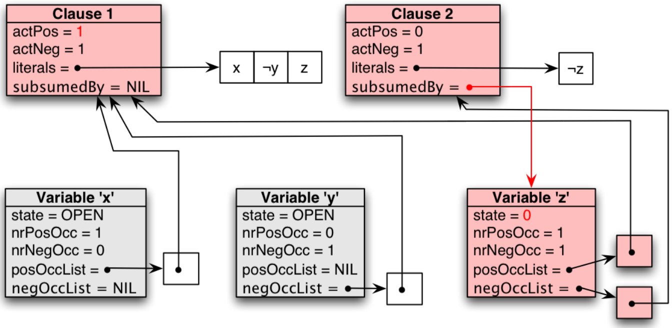
Traditional Approach: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ unit propagation: set $z = 0$

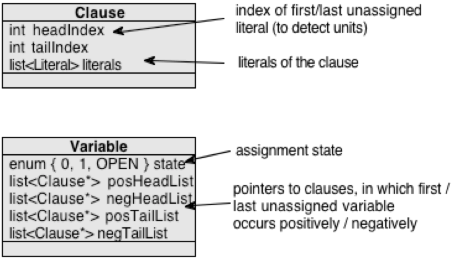


Traditional Approach: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ unit propagation: set $z = 0$



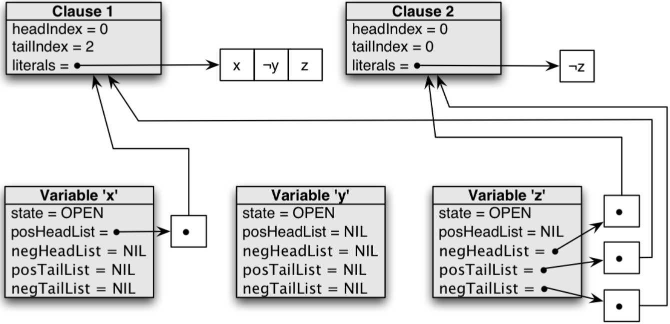
Head/Tail Lists



Zhang, Stickel (1996)

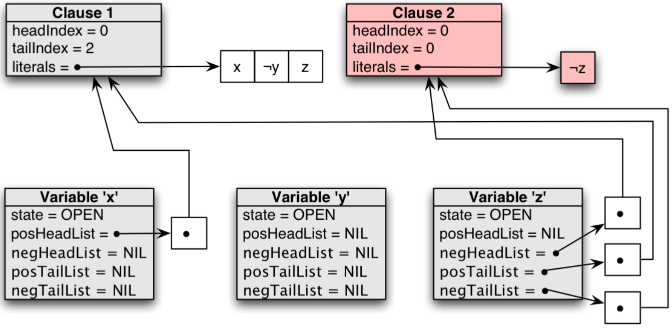
Head/Tail Lists: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$



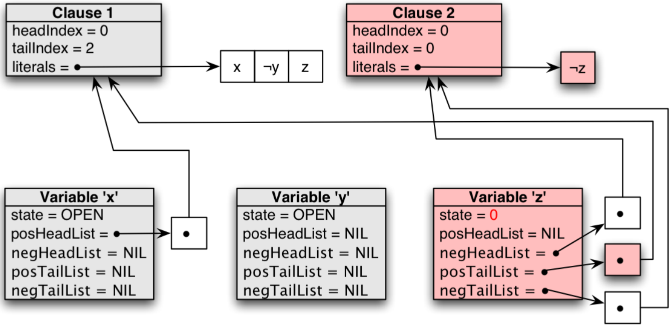
Head/Tail Lists: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ detected unit clause: $\{\neg z\}$



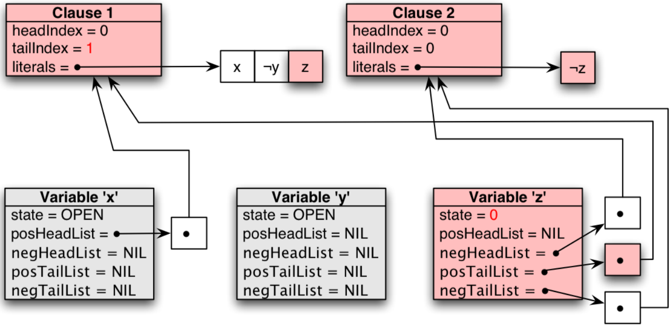
Head/Tail Lists: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ unit propagation: set $z = 0$



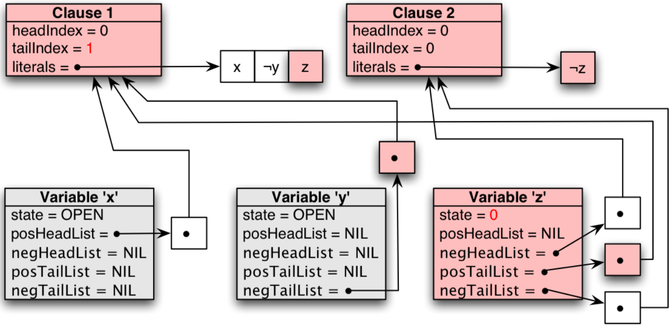
Head/Tail Lists: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ unit propagation: set $z = 0$



Head/Tail Lists: Example

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ unit propagation: set $z = 0$



2 watched literals

Karlsruhe Institute of Technology

The Data Structure

- In each non-satisfied clause "watch" two non-false literals
- For each literal remember all the clauses where it is watched

Maintain the invariant: two watched non-false literals in non-sat clauses

- If a literal becomes false find another one to watch
- If that is not possible the clause is unit

Advantages

2 watched literals

The Data Structure

- In each non-satisfied clause "watch" two non-false literals
- For each literal remember all the clauses where it is watched

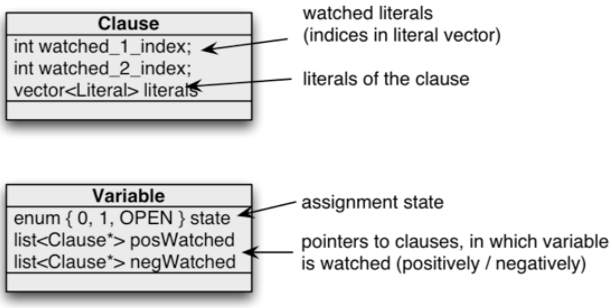
Maintain the invariant: two watched non-false literals in non-sat clauses

- If a literal becomes false find another one to watch
- If that is not possible the clause is unit

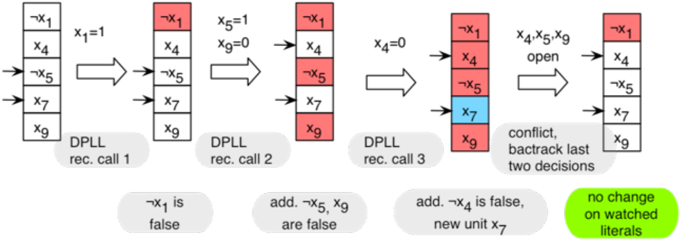
Advantages

- visit fewer clauses: when $x_i = T$ is added only visit clauses where $\overline{x_i}$ is watched
- no need to do anything at backtracking and restarts
  - watched literals cannot become false

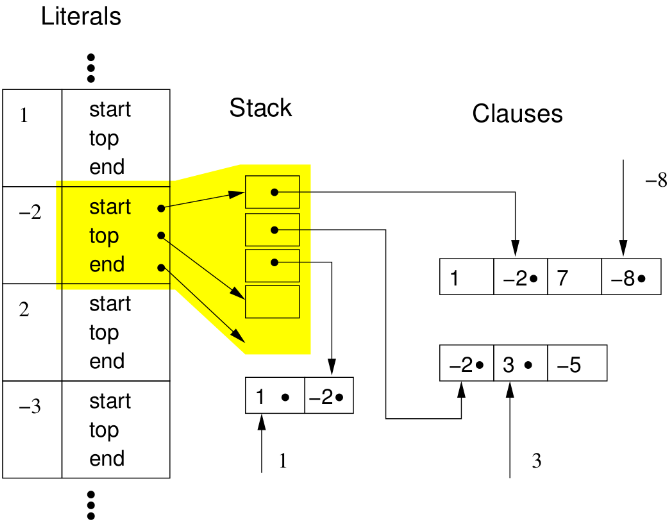
2 Watched Literals: Data Structures



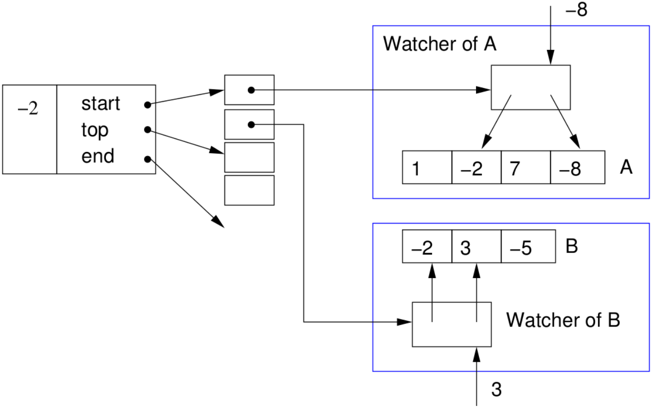
2 Watched Literals: Example



zChaff

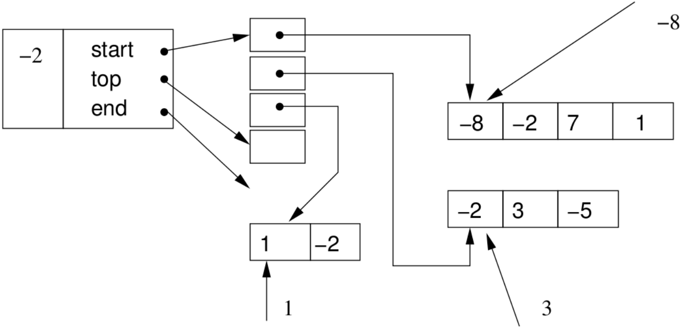

Literals
1
start
top
end
−2
start
top
end
2
start
top
end
−3
start
top
end
Stack
Clauses
−8
1
−2•
7
−8•
−2•
3 •
−5
1 •
−2•
1
3

Limmat



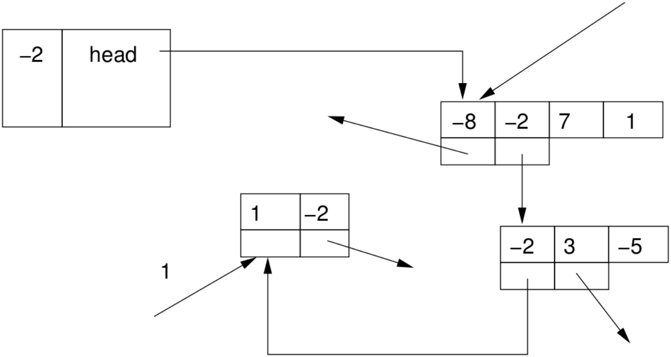
- Good for parallel SAT solvers with shared clause database

MiniSat



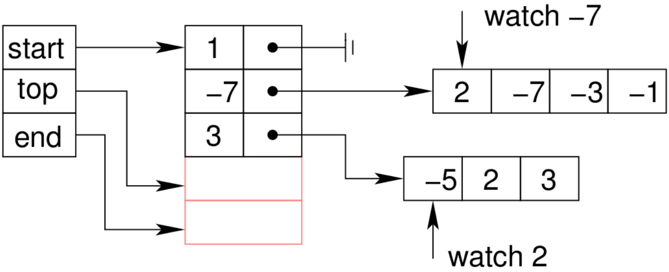
invariant: first two literals are watched

PicoSat



invariant: first two literals are watched

Lingeling



- often the other watched literal satisfies the clause
- for binary clauses no need to store the clause

MiniSAT `propagate()`-Function

```
CRef Solver::propagate()
{
  CRef   confl    = CRef_Undef;
  int    num_props = 0;

  while (qhead < trail.size()){
  Lit p   = trail[qhead++]; // propagate 'p'.
  vec<Watcher>&  ws   = watches.lookup(p);
  Watcher   *i, *j, *end;
  num_props++;

  for (i = j = (Watcher*)ws, end = i + ws.size();
    i != end;){
    // Try to avoid inspecting the clause:
    Lit blocker = i->blocker;
    if (value(blocker) == l_True){
    *j++ = *i++; continue; }

    // Make sure the false literal is data[1]:
    CRef    cr    = i->cref;
    Clause&   c    = ca[cr];
    Lit   false_lit = ~p;
    if (c[0] == false_lit)
    c[0] = c[1], c[1] = false_lit;
    assert(c[1] == false_lit);
    i++;

    // If 0th watch is true, clause is satisfied.
    Lit    first = c[0];
    Watcher w    = Watcher(cr, first);
    if (first != blocker && value(first) == l_True){
    *j++ = w; continue; }

    // Look for new watch:
    for (int k = 2; k < c.size(); k++)
      if (value(c[k]) != l_False){
        c[1] = c[k]; c[k] = false_lit;
        watches[~c[1]].push(w);
        goto NextClause; }

    // Did not find watch -- clause is unit
    *j++ = w;
    if (value(first) == l_False){
      confl = cr;
      qhead = trail.size();
      // Copy the remaining watches:
      while (i < end)
        *j++ = *i++;
    }else
      uncheckedEnqueue(first, cr);

  NextClause:;
  }
  ws.shrink(i - j);
 }
 propagations += num_props;
 simpDB_props -= num_props;

 return confl;
}
```

References I

- B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI'92, AAAI Press, 1992, pp. 440–446.
  URL http://dl.acm.org/citation.cfm?id=1867135.1867203
- D. J. Johnson, M. A. Trick (Eds.), Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993, American Mathematical Society, Boston, MA, USA, 1996.
- M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (3) (1960) 201–215.
  doi:10.1145/321033.321034.
  URL http://doi.acm.org/10.1145/321033.321034

References II

M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Commun. ACM 5 (7) (1962) 394–397. doi:10.1145/368273.368557.
URL http://doi.acm.org/10.1145/368273.368557