SAT Solving

Ringvorlesung

DK LogiCS

25. February 2020

TU Wien

Armin Biere

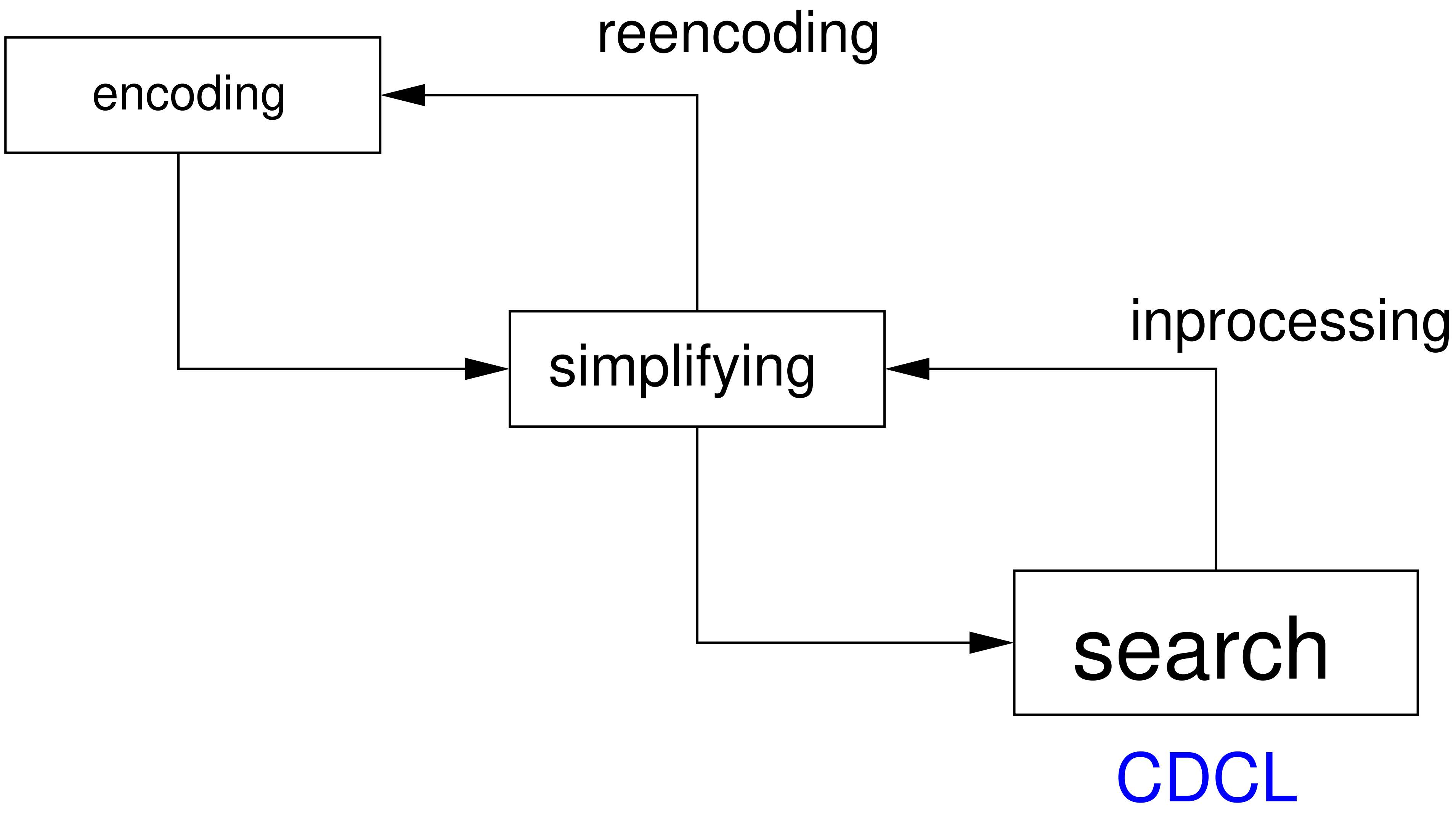
JOHANNES KEPLER
UNIVERSITY LINZ

http://fmv.jku.at/ringvorlesung

http://fmv.jku.at/sat

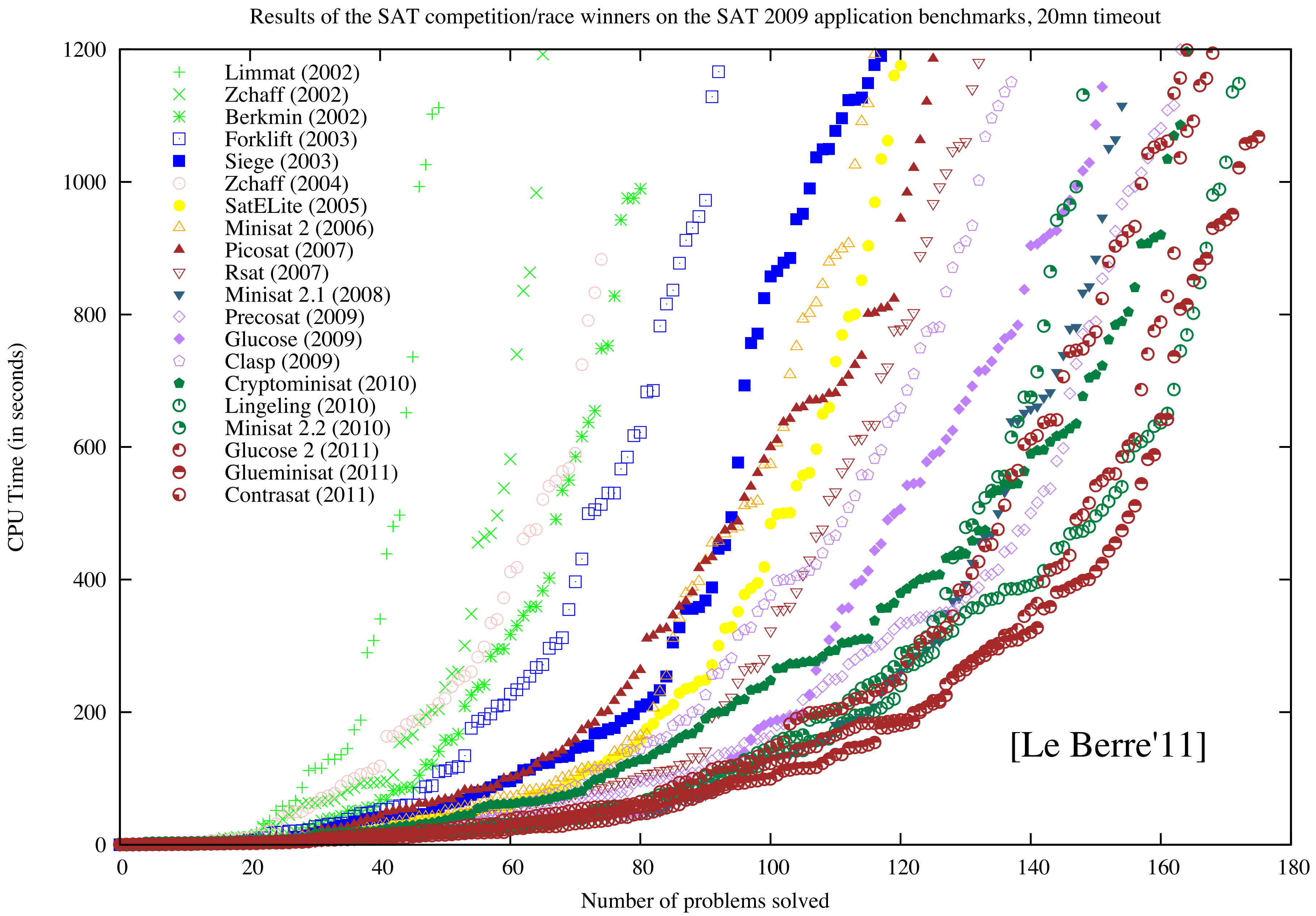
http://fmv.jku.at/biere/talks/{Trento,Banff}

http://fmv.jku.at/{picosat,lingeling,cleaneling,cadical}

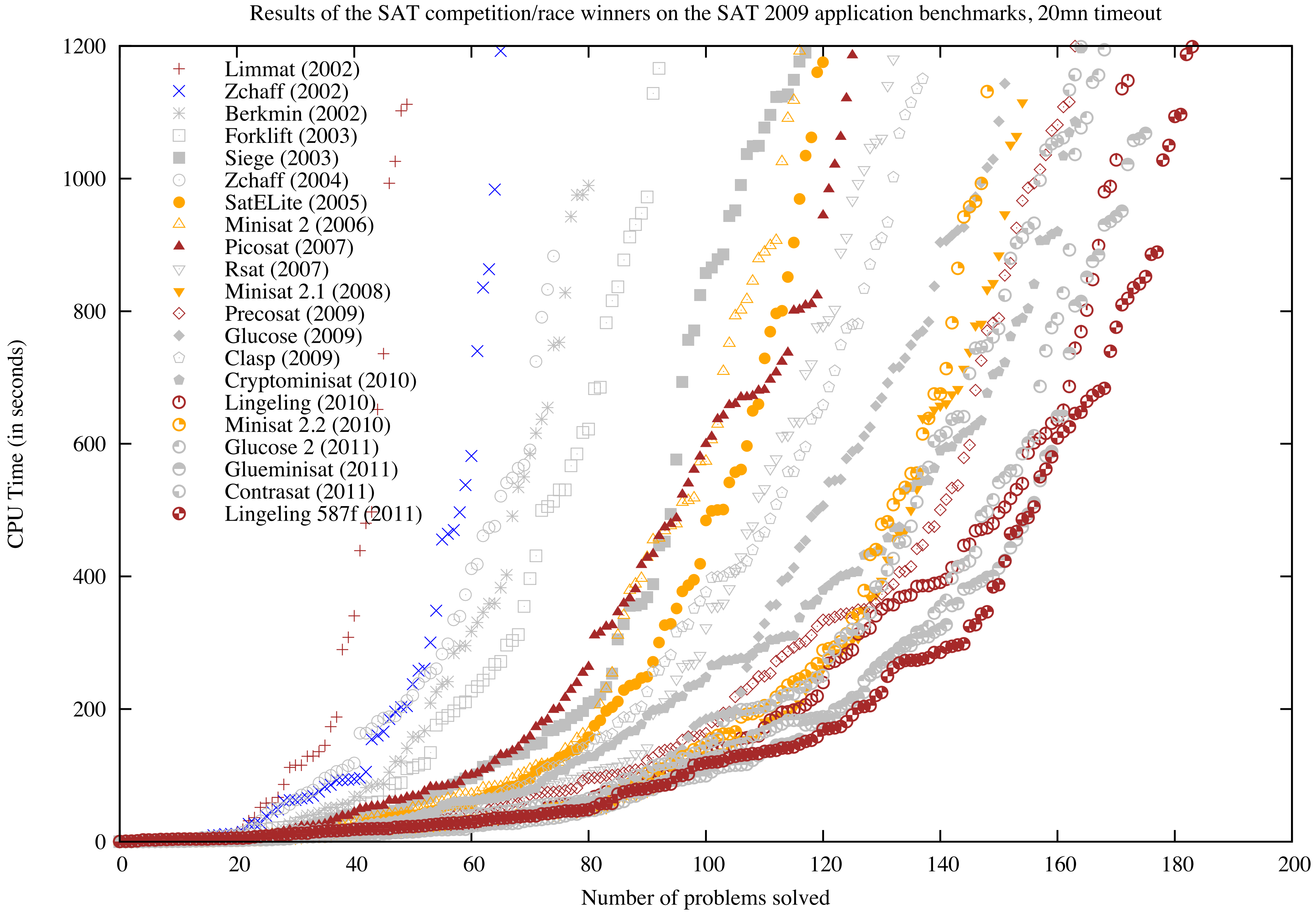

reencoding
encoding
simplifying
inprocessing
search
CDCL

89 medals
52 gold

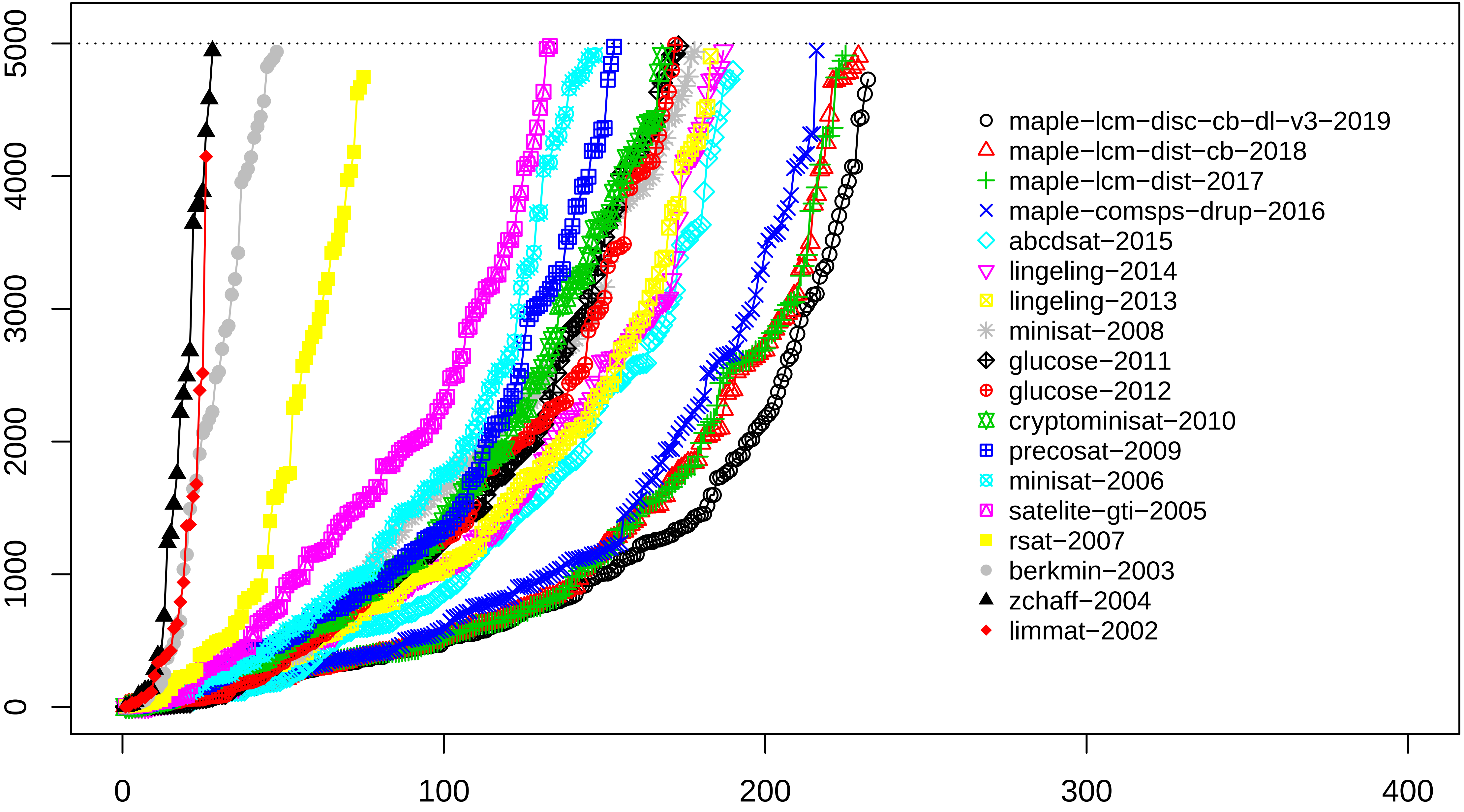
SAT Competition / Race Winners on SC 2009 Application Benchmarks



ZChaff, MiniSAT, My Solvers



All Time SAT Competition Winners on SAT 2019 Benchmarks



DP / DPLL

- dates back to the 50'ies:

  1^{st} version DP is *resolution based* $\Rightarrow$ SatELite preprocessor [EénBiere05]

  2^{st} version D(P)LL splits space for time $\Rightarrow$ CDCL

- **ideas:**
  - 1st version: eliminate the two cases of assigning a variable in space or
  - 2nd version: case analysis in time, e.g. try $x = 0, 1$ in turn and recurse

- most successful SAT solvers are based on variant (CDCL) of the second version

  works for very large instances

- recent ($\leq$ 20 years) optimizations:

  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures

  (we will have a look at each of them)

DP Procedure

[DavisPutnam'61]

forever

- if $F = \top$ **return** *satisfiable*
- if $\bot \in F$ **return** *unsatisfiable*
- pick remaining variable x
- add all resolvents on x
- remove all clauses with x and $\neg x$

$\Rightarrow$ SatELite preprocessor [EénBiere05]

D(P)LL Procedure

[DavisLogemannLoveland'62]

$DPLL(F)$

$F := BCP(F)$ boolean constraint propagation

if $F = \top$ **return** *satisfiable*

if $\bot \in F$ **return** *unsatisfiable*

pick remaining variable x and literal $l \in \{x, \neg x\}$

if $DPLL(F \wedge \{l\})$ returns *satisfiable* **return** *satisfiable*

return $DPLL(F \wedge \{\neg l\})$

$\Rightarrow$ CDCL

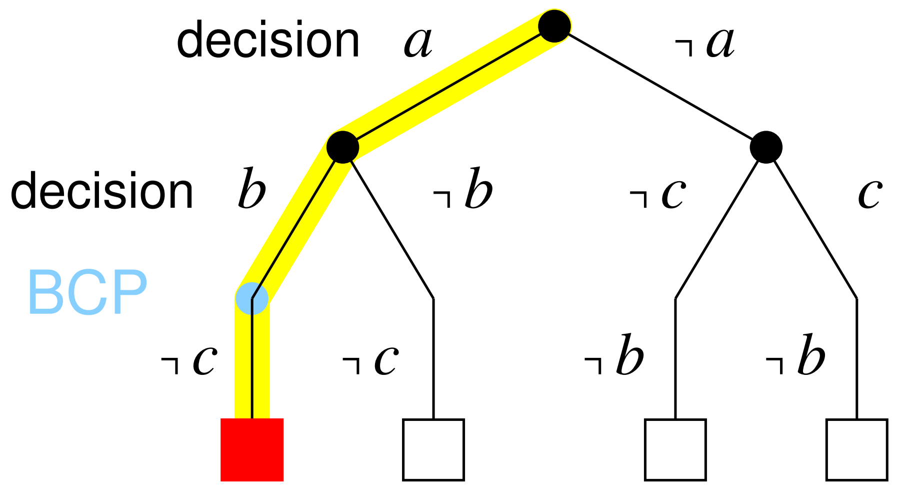
DPLL Example

[DavisLogemannLoveland'62]

$a = 1$

$b = 1$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

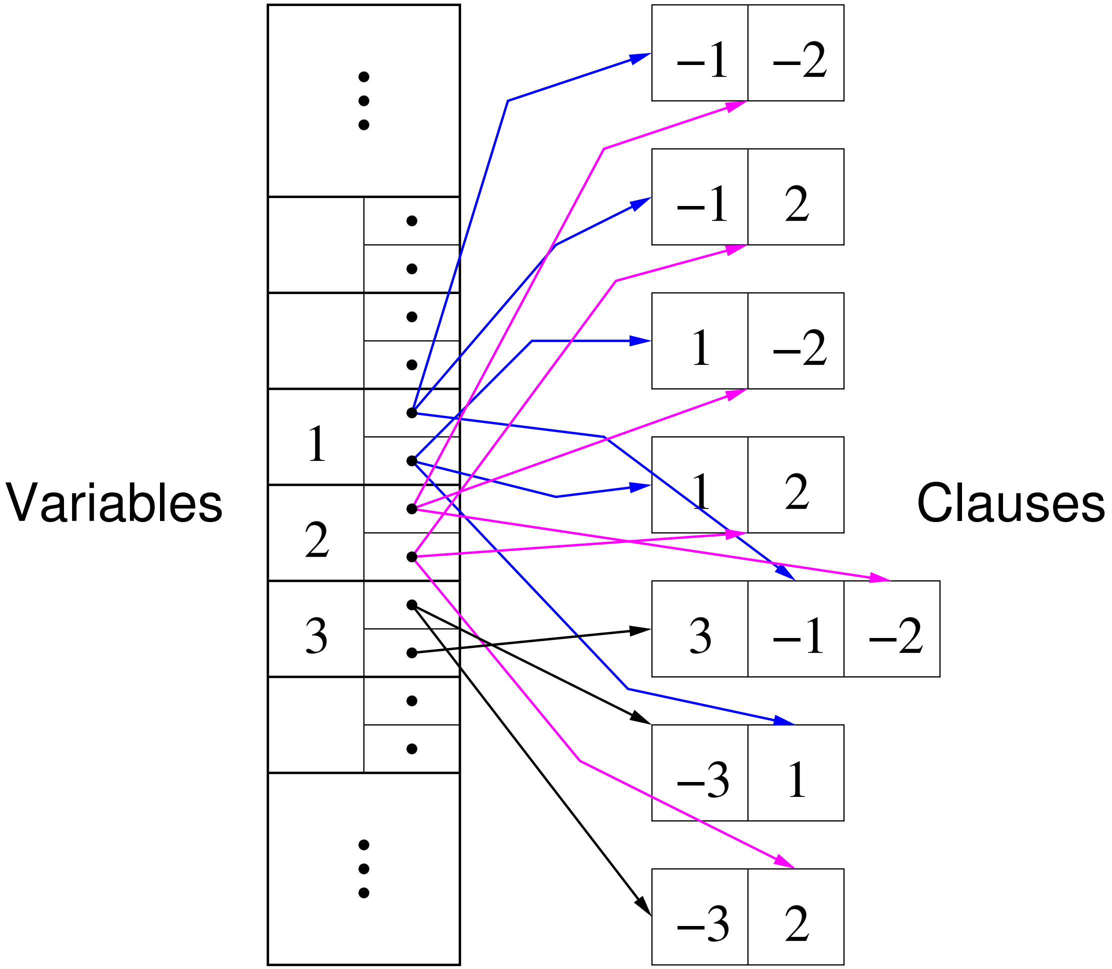
$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

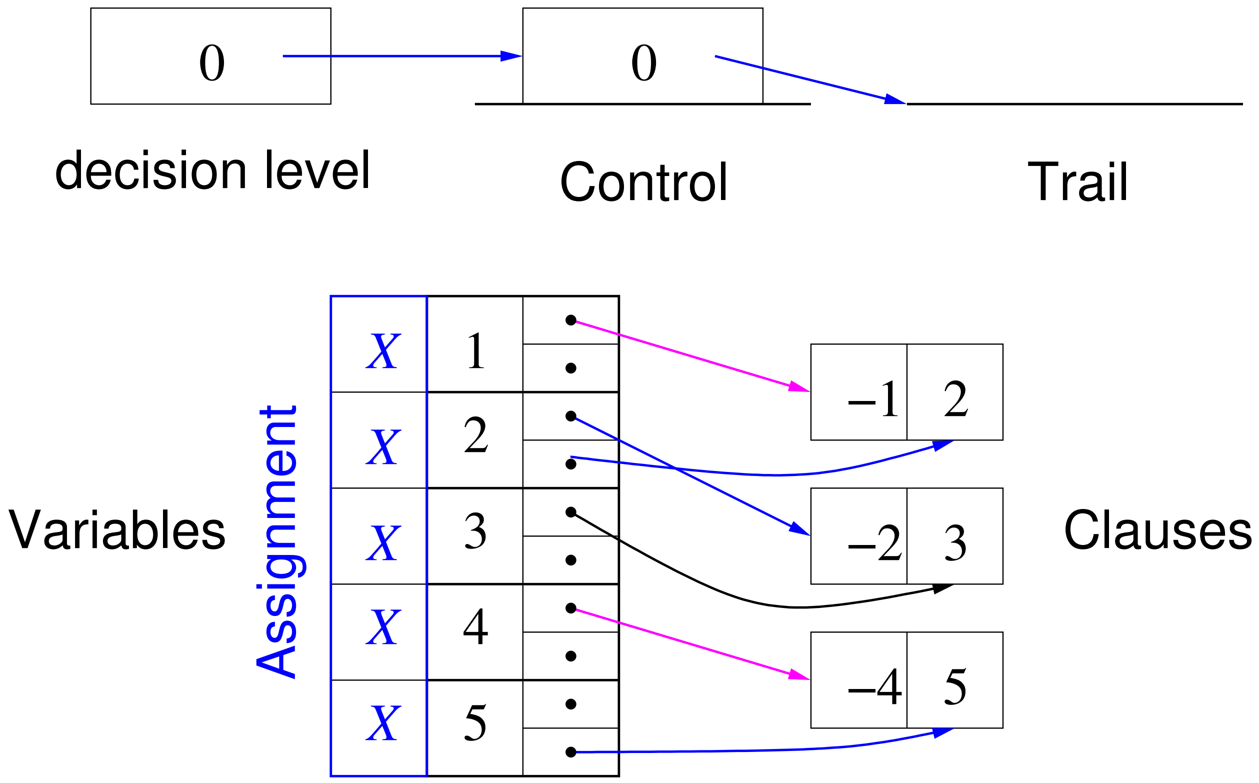
Simple Data Structures in DPLL Implementation

[DavisLogemannLoveland'62]



BCP Example

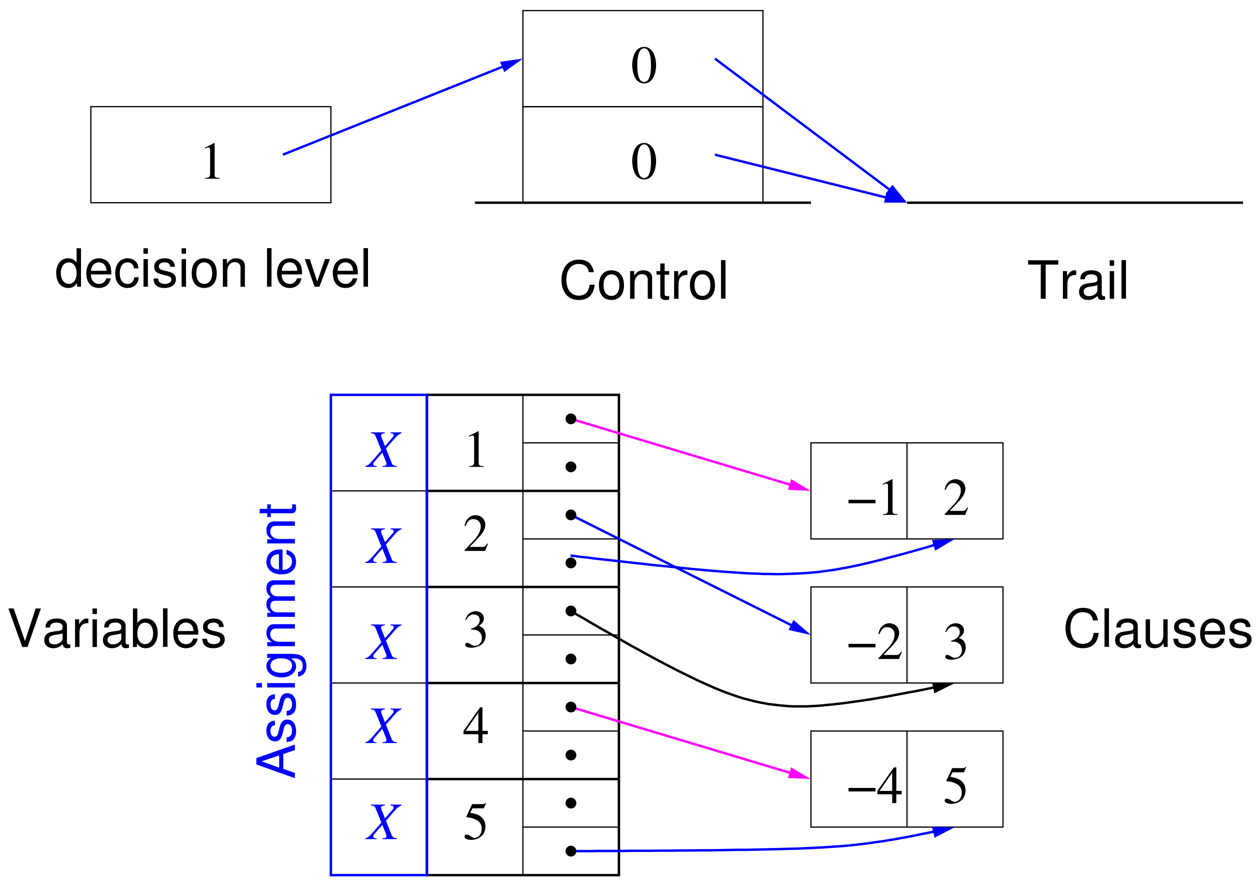
[DavisLogemannLoveland'62]



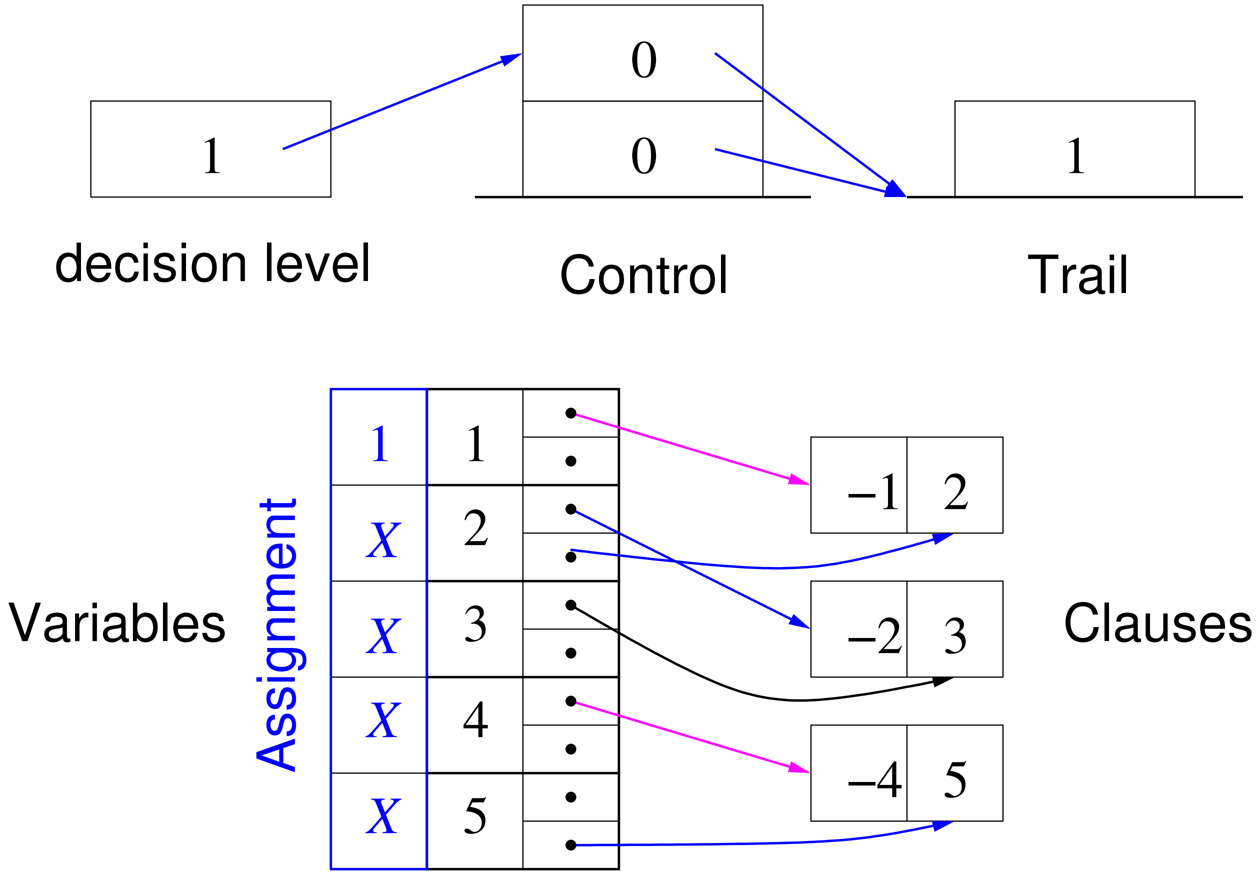
Example cont.

[DavisLogemannLoveland'62]

Decide

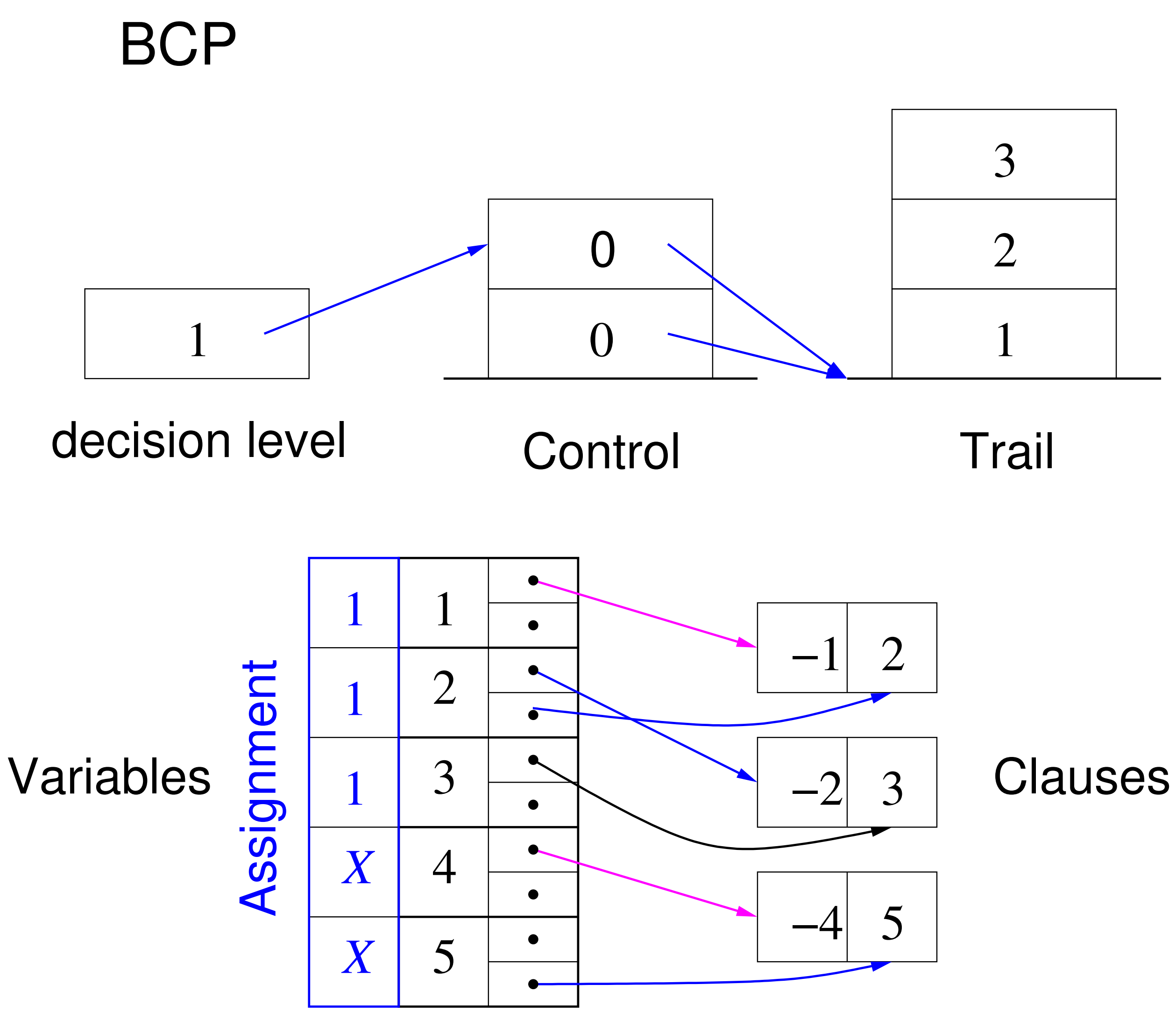


Assign


1
decision level
0
0
Control
1
Trail
Variables
Assignment
1
1
X
2
X
3
X
4
X
5
−1
2
−2
3
−4
5
Clauses

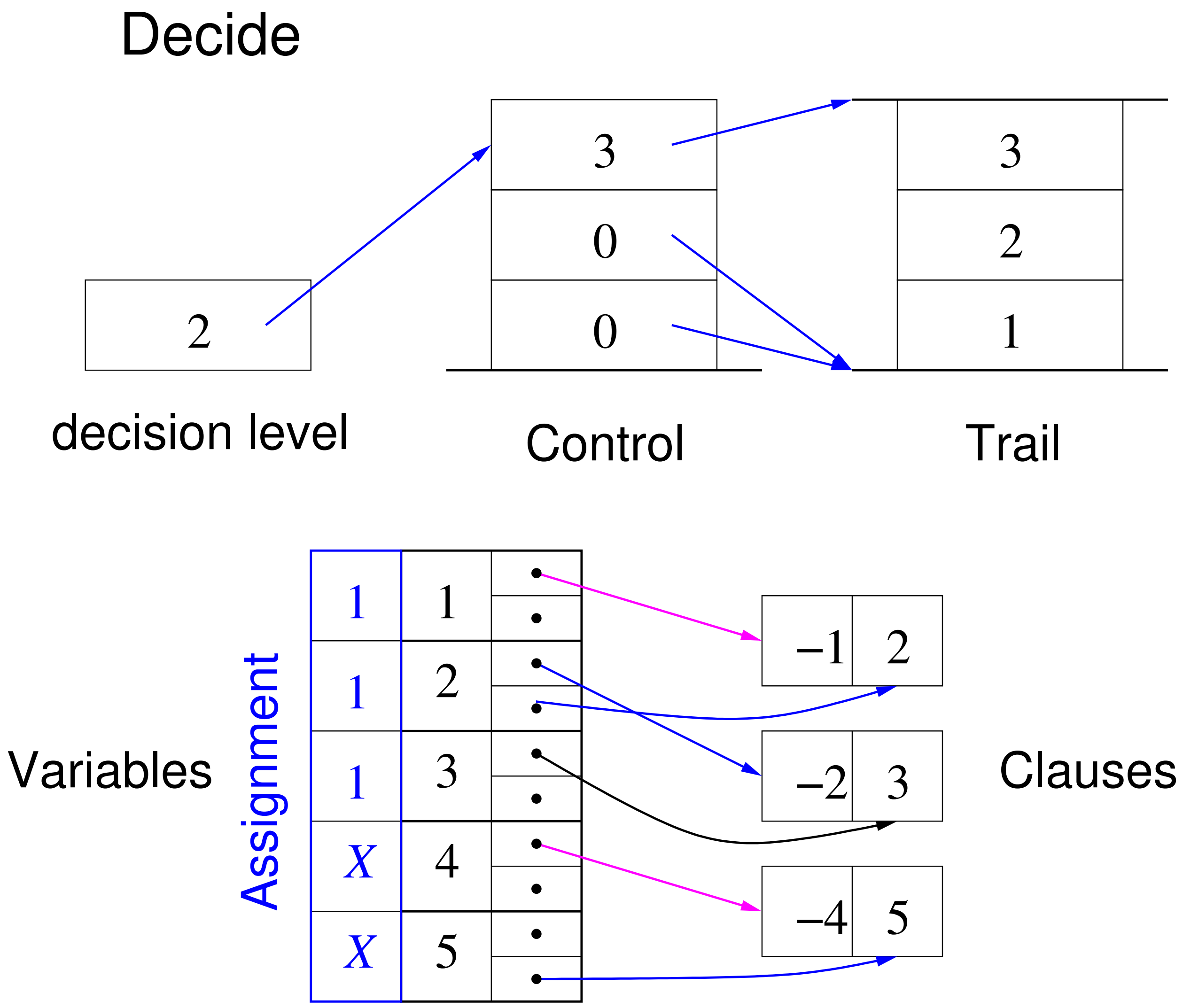
Example cont.

[DavisLogemannLoveland'62]



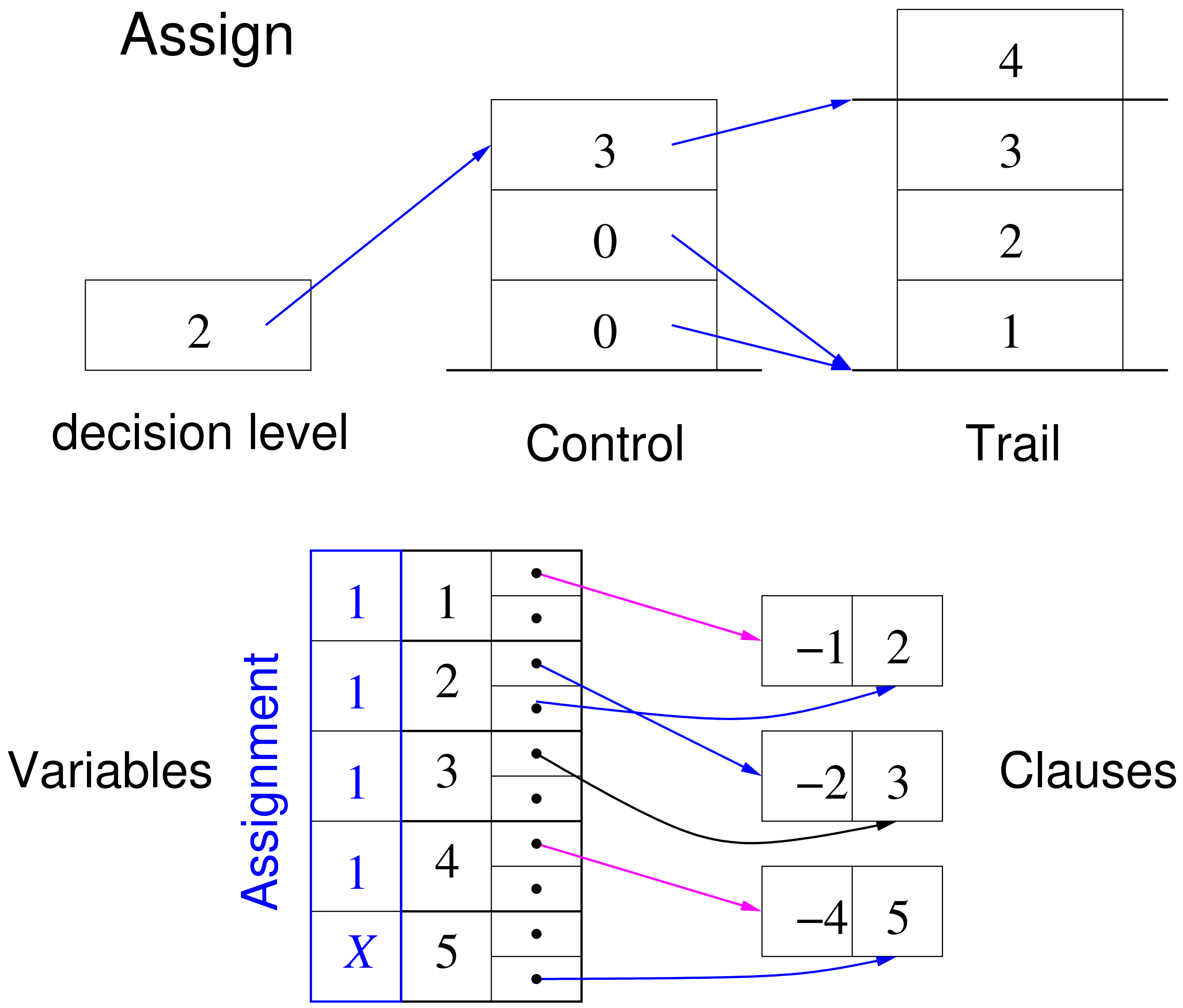
Example cont.

[DavisLogemannLoveland'62]



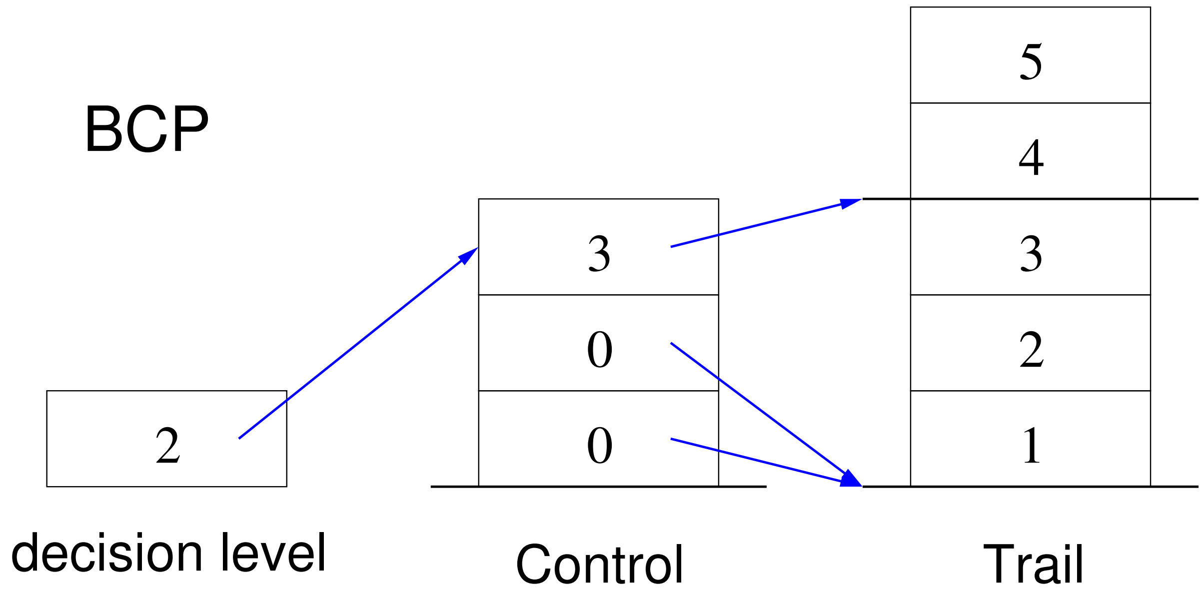
Example cont.

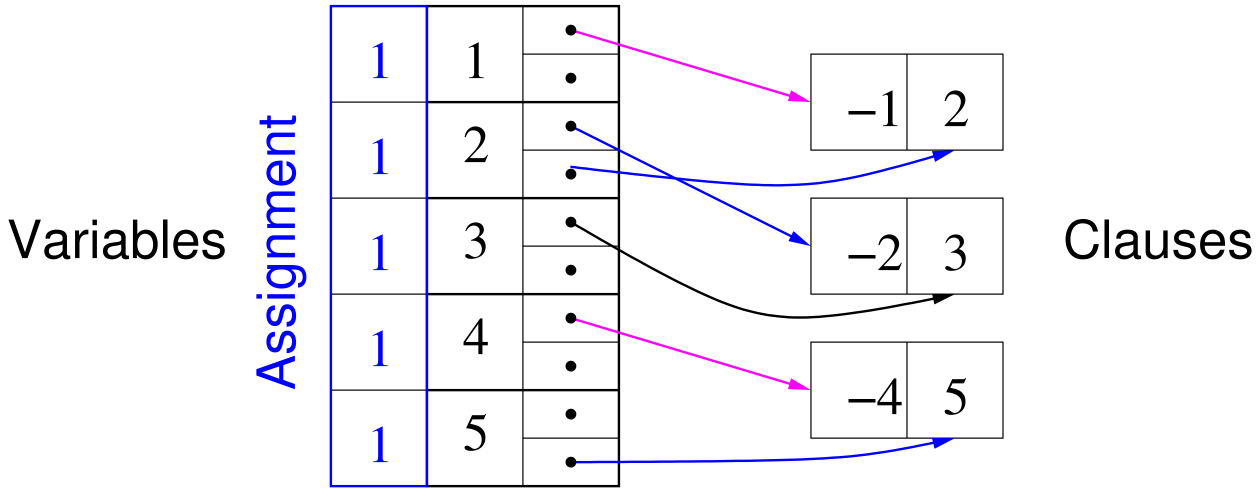
[DavisLogemannLoveland'62]



Example cont.

[DavisLogemannLoveland'62]





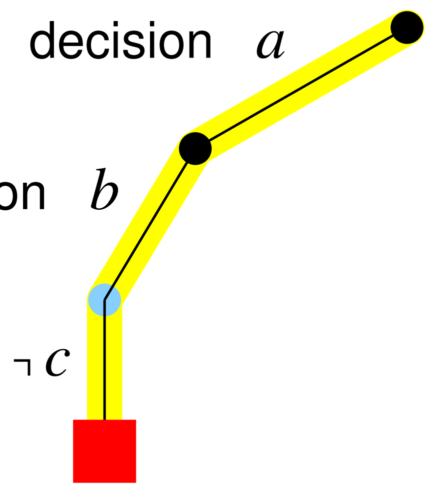
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 1$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

learn $\neg a \vee \neg b$

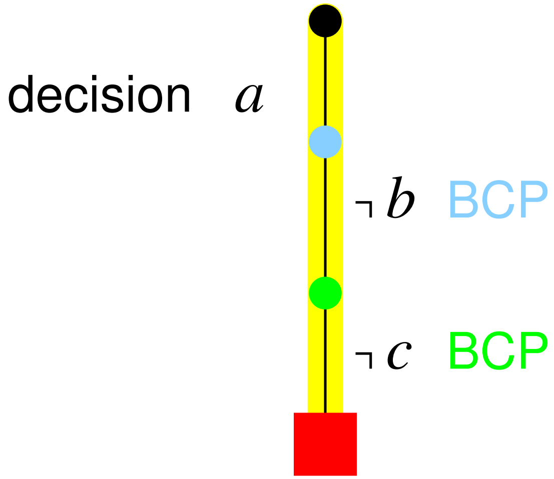
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

learn $\neg a$

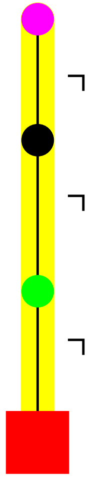
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$



$\neg a$ BCP

$\neg c$ decision

$\neg b$ BCP

clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

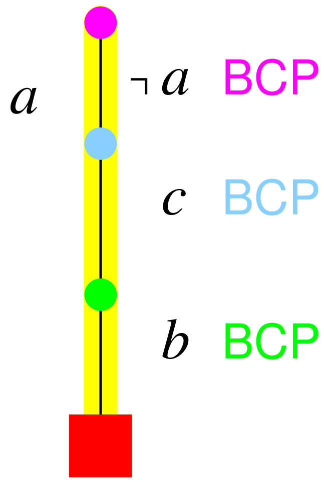
learn c

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

c

learn $\perp$

empty clause

Decision Heuristics

- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast to compute, since only calculated once
  - and thus can also use more expensive algorithms

- **dynamic heuristics**
  - typically calculated from number of occurences of literals
    (in unsatisfied clauses)
  - could be rather expensive, since it requires traversal of all clauses
    (or more expensive updates in BCP)
  - effective *second order* dynamic heuristics (e.g. VSIDS in Chaff)

Other popular Decision Heuristics

- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic *first order* heuristic (e.g. GRASP solver)
  - choose literal (variable + phase) which occurs most often (ignore satisfied clauses)
  - requires explicit traversal of CNF (or more expensive BCP)
- look-ahead heuristics (e.g. SATZ or MARCH solver) **failed literals, probing**
  - trial assignments and BCP for all/some unassigned variables (both phases)
  - if BCP leads to conflict, enforce toggled assignment of current trial decision
  - optionally learn binary clauses and perform equivalent literal substitution
  - decision: most balanced w.r.t. prop. assignments / sat. clauses / reduced clauses
  - related to our recent Cube & Conquer paper [HeuleKullmanWieringaBiere-HVC'11]

    see also "Everything's Bigger in Texas: The Largest Math Proof Ever"

    `https://www.cs.utexas.edu/~marijn/ptn`

Exponential VSIDS (EVSIDS)

Chaff [MoskewiczMadiganZhaoZhangMalik'01]

- increment score of involved variables by 1
- decay score of all variables every 256'th conflict by halfing the score
- sort priority queue after decay and not at every conflict

MiniSAT uses EVSIDS [EénSörensson'03/'06]

- update score of involved variables — as actually LIS would also do
- dynamically adjust increment: $\delta' = \delta \cdot \frac{1}{f}$ — typically increment δ by 5%
- use floating point representation of score
- "rescore" to avoid overflow in regular intervals
- EVSIDS linearly related to NVSIDS

Relating EVSIDS and NVSIDS

(consider only one variable)

$$\delta_k \quad = \quad \begin{cases} 1 & \text{if involved in } k\text{-th conflict} \\ 0 & \text{otherwise} \end{cases}$$

$$i_k \quad = \quad (1-f)\cdot\delta_k$$

$$\boxed{s_n} = (\ldots(i_1\cdot f + i_2)\cdot f + i_3)\cdot f \cdots)\cdot f + i_n = \sum_{k=1}^{n} i_k \cdot f^{n-k} = (1-f)\cdot\sum_{k=1}^{n}\delta_k\cdot f^{n-k} \qquad \text{(NVSIDS)}$$

$$\boxed{S_n} = \frac{f^{-n}}{(1-f)}\cdot s_n = \frac{f^{-n}}{(1-f)}\cdot(1-f)\cdot\sum_{k=1}^{n}\delta_k\cdot f^{n-k} = \sum_{k=1}^{n}\delta_k\cdot f^{-k} \qquad \text{(EVSIDS)}$$

BerkMin's Dynamic Second Order Heuristics

[GoldbergNovikov-DATE'02]

- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to choose the decision variable for one clause
  - if all learned clauses are satisfied use other heuristics
  - intuitively obtains another order of localization (no proofs yet)
- mixed results as other variants VMTF, CMTF (var/clause move to front)
  - see our Banff talk / video for actual experimental results
  - recently showed that VMTF can be competitive [BiereFröhlich-SAT'15]
    (if implemented carefully, probably needs Glucose restart scheme)

Reducing Learned Clauses

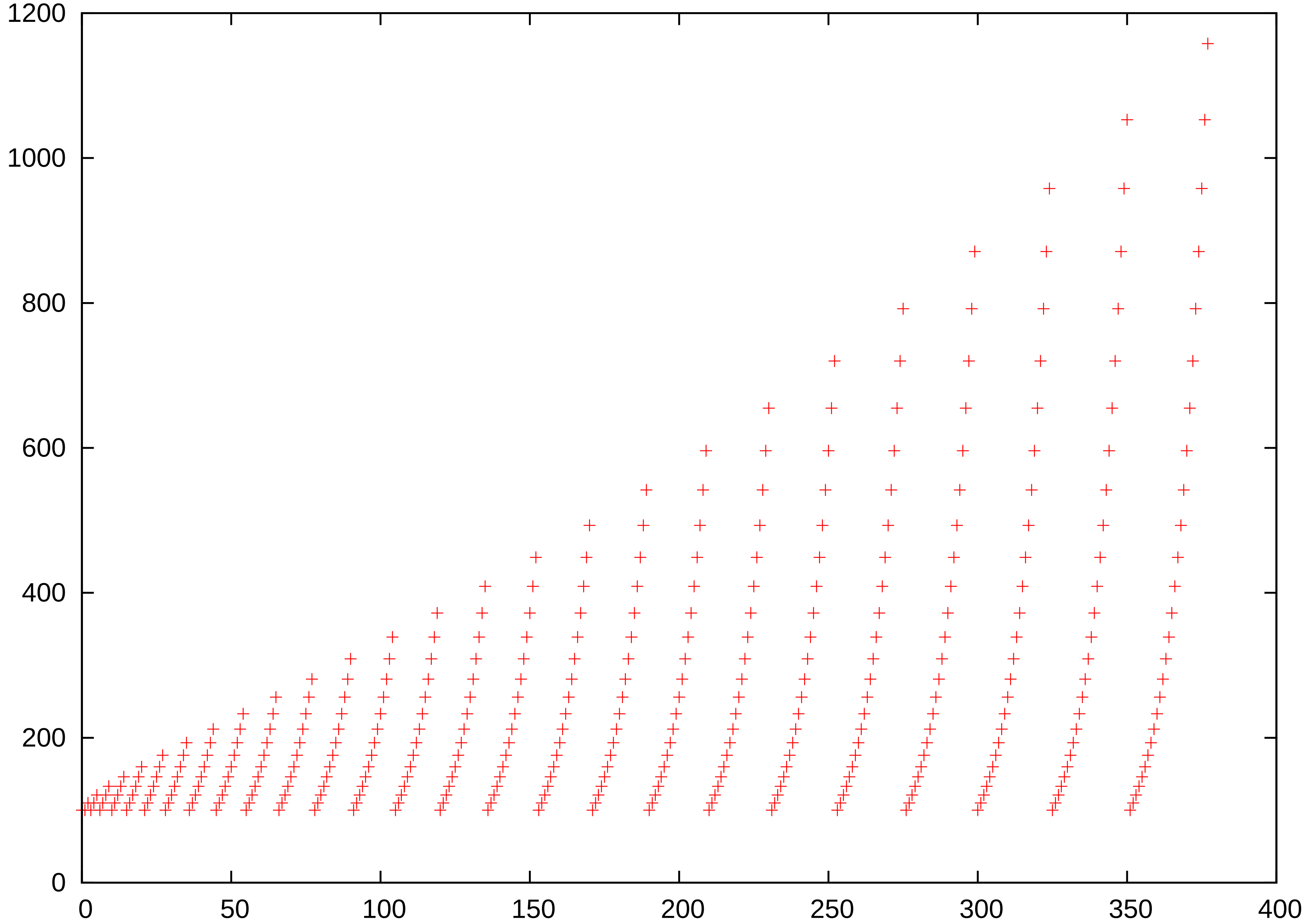
- keeping all learned clauses slows down BCP — kind of quadratically
  - so SATO and RelSAT just kept only “short” clauses
- better periodically delete “useless” learned clauses
  - keep a certain number of learned clauses — “search cache”
  - if this number is reached MiniSAT reduces (deletes) half of the clauses
  - keep *most active*, then *shortest*, then *youngest* (LILO) clauses
  - after reduction maximum number kept learned clauses is increased geometrically
- LBD (Glue) based (apriori!) prediction for usefulness — [AudemardLaurent’09]
  - LBD (Glue) = number of decision-levels in the learned clause
  - allows arithmetic increase of number of kept learned clauses
  - keep clauses with small LBD forever ($\leq 2 \ldots 5$)

Restarts

- for satisfiable instances the solver may get stuck in the unsatisfiable part
  - even if the search space contains a large satisfiable part
- often it is a good strategy to abandon the current search and restart
  - restart after the number of decisions reached a *restart limit*
- avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses
- for completeness dynamically increase restart limit
  - arithmetically, geometrically, Luby, Inner/Outer
  - recent technique from Glucose:
    - short vs. large window running average LBD
    - if recent LBD values are larger than long time average then restart
- interleave fast / slow restart phases [Chanseok Oh]

Inner/Outer Restart Intervals

378 restarts in 104408 conflicts

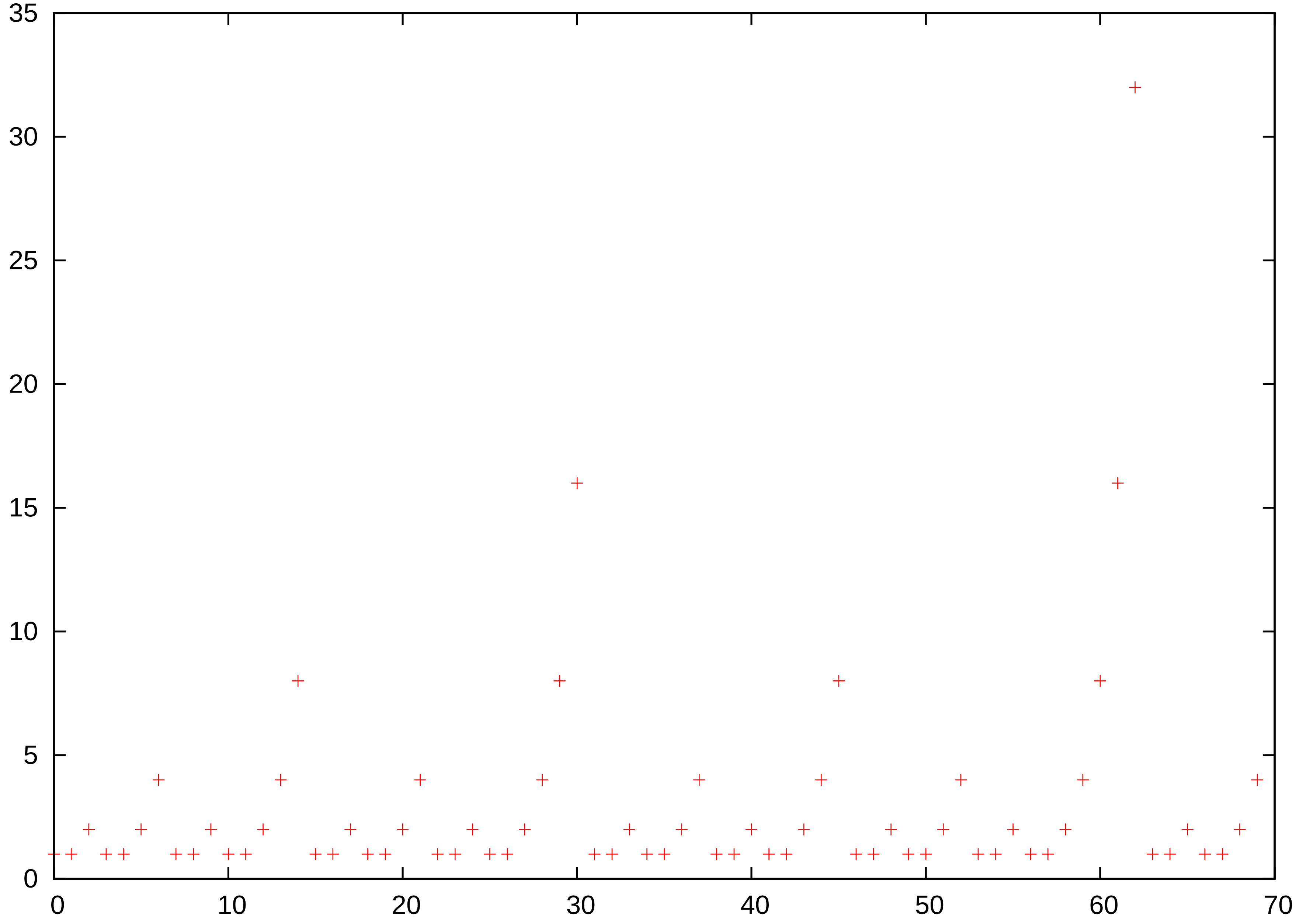


Inner/Outer Restart Scheduling

```
int inner = 100, outer = 100;
int restarts = 0, conflicts = 0;

for (;;)
  {
    ... // run SAT core loop for 'inner' conflicts

    restarts++;
    conflicts += inner;

    if (inner >= outer)
      {
        outer *= 1.1;
        inner = 100;
      }
    else
      inner *= 1.1;
  }
```

Luby's Restart Intervals

70 restarts in 104448 conflicts



Luby Restart Scheduling

```
unsigned
luby (unsigned i)
{
  unsigned k;

  for (k = 1; k < 32; k++)
    if (i == (1 << k) - 1)
      return 1 << (k - 1);

  for (k = 1;; k++)
    if ((1 << (k - 1)) <= i && i < (1 << k) - 1)
      return luby (i - (1 << (k-1)) + 1);
}

limit = 512 * luby (++restarts);
...  // run SAT core loop for 'limit' conflicts
```

Reluctant Doubling Sequence

[Knuth'12]

$$
\begin{aligned}
(u_1, v_1) &= (1, 1) \\
(u_{n+1}, v_{n+1}) &= ((u_n \mathbin{\&} -u_n == v_n) \mathbin{?} (u_n + 1, 1) : (u_n, 2v_n))
\end{aligned}
$$

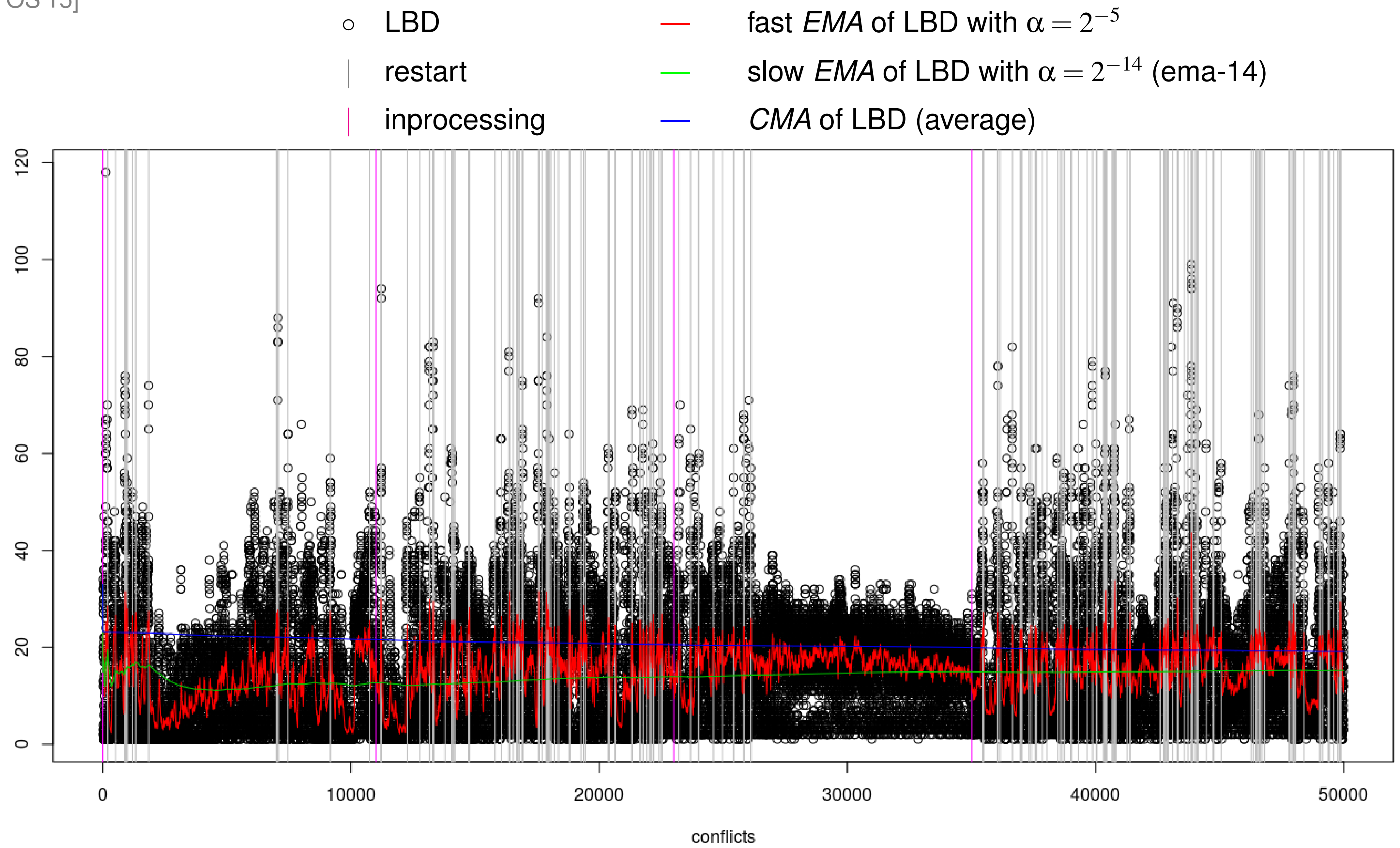
$$(1,1),\ (2,1),\ (2,2),\ (3,1),\ (4,1),\ (4,2),\ (4,4),\ (5,1),\ \ldots$$

Phase Saving and Rapid Restarts

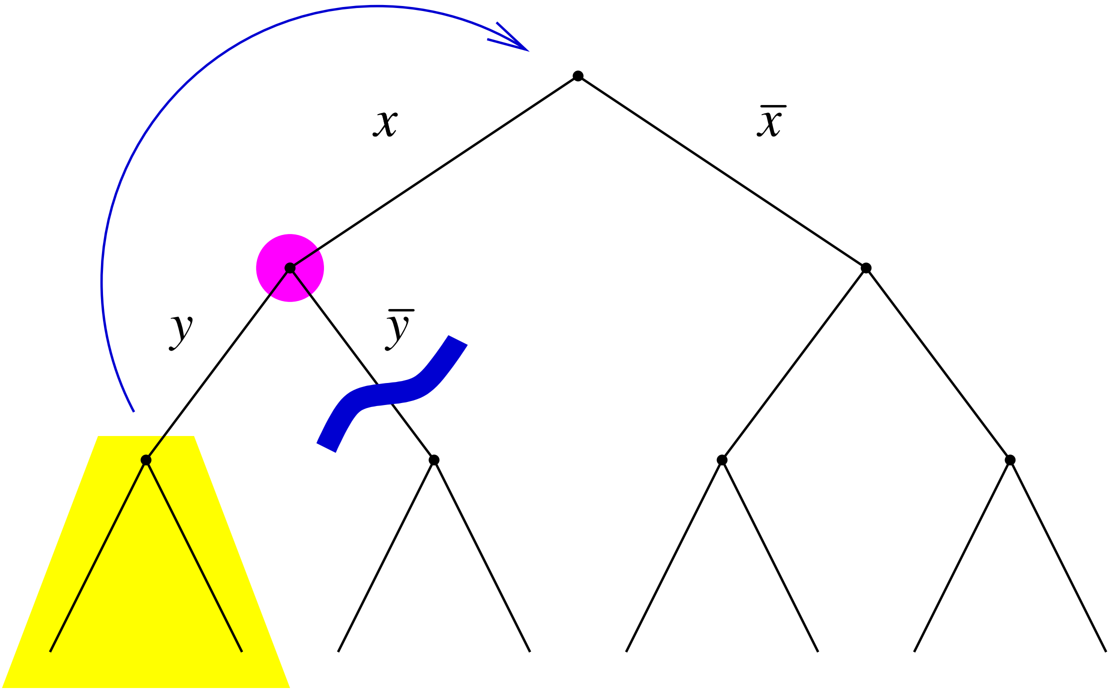
- phase assignment:
  - assign decision variable to 0 or 1?
  - only thing that matters in *satisfiable* instances
- "phase saving" as in RSat:
  - pick phase of last assignment (if not forced to, do not toggle assignment)
  - initially use statically computed phase (typically LIS)
  - so can be seen to maintain a *global full assignment*
- rapid restarts: varying restart interval with bursts of restarts
  - not ony theoretically avoids local minima
  - works nicely together with phase saving

Restart Scheduling with Exponential Moving Averages

[BiereFröhlich-POS'15]



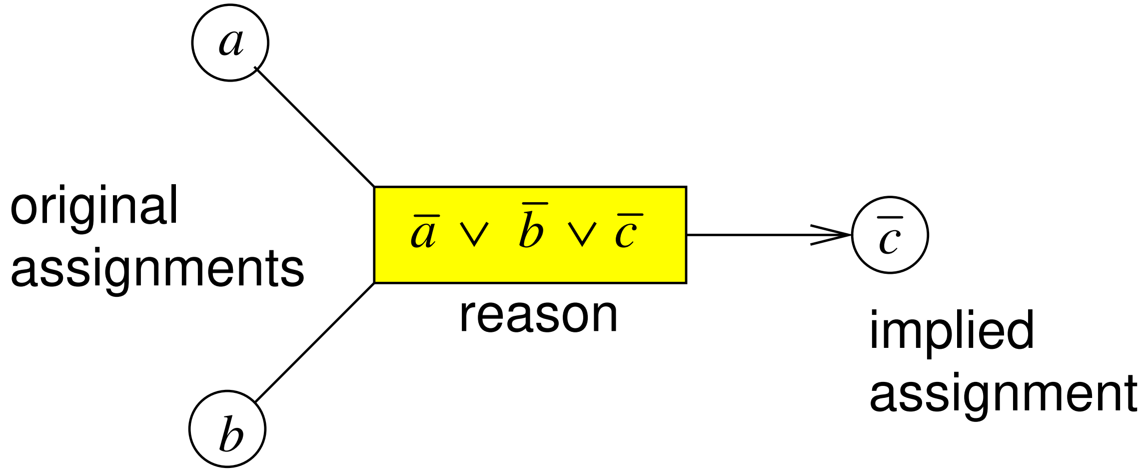
Backjumping



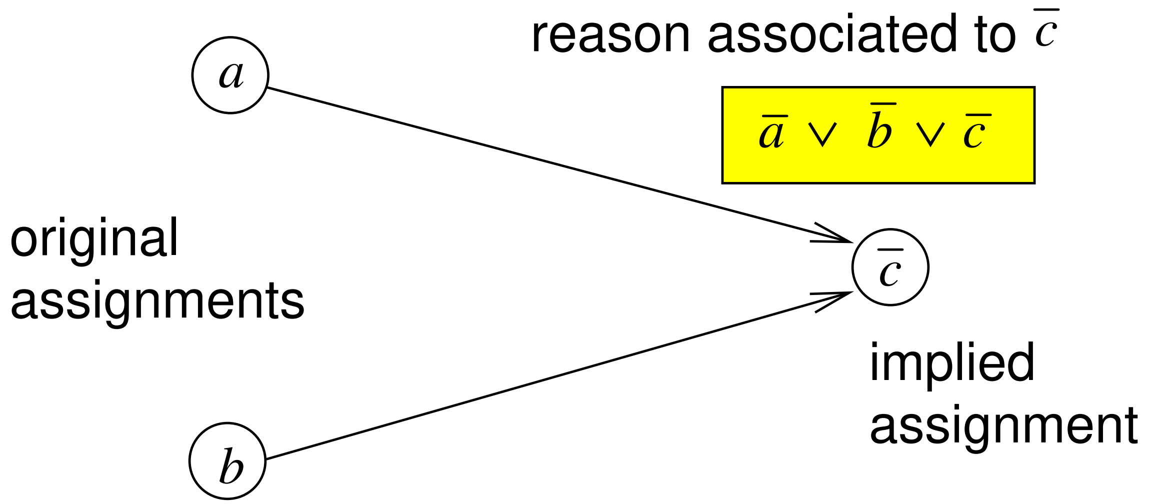
If y has never been used to derive a conflict, then skip $\overline{y}$ case.

Immediately *jump back* to the $\overline{x}$ case – assuming x was used.

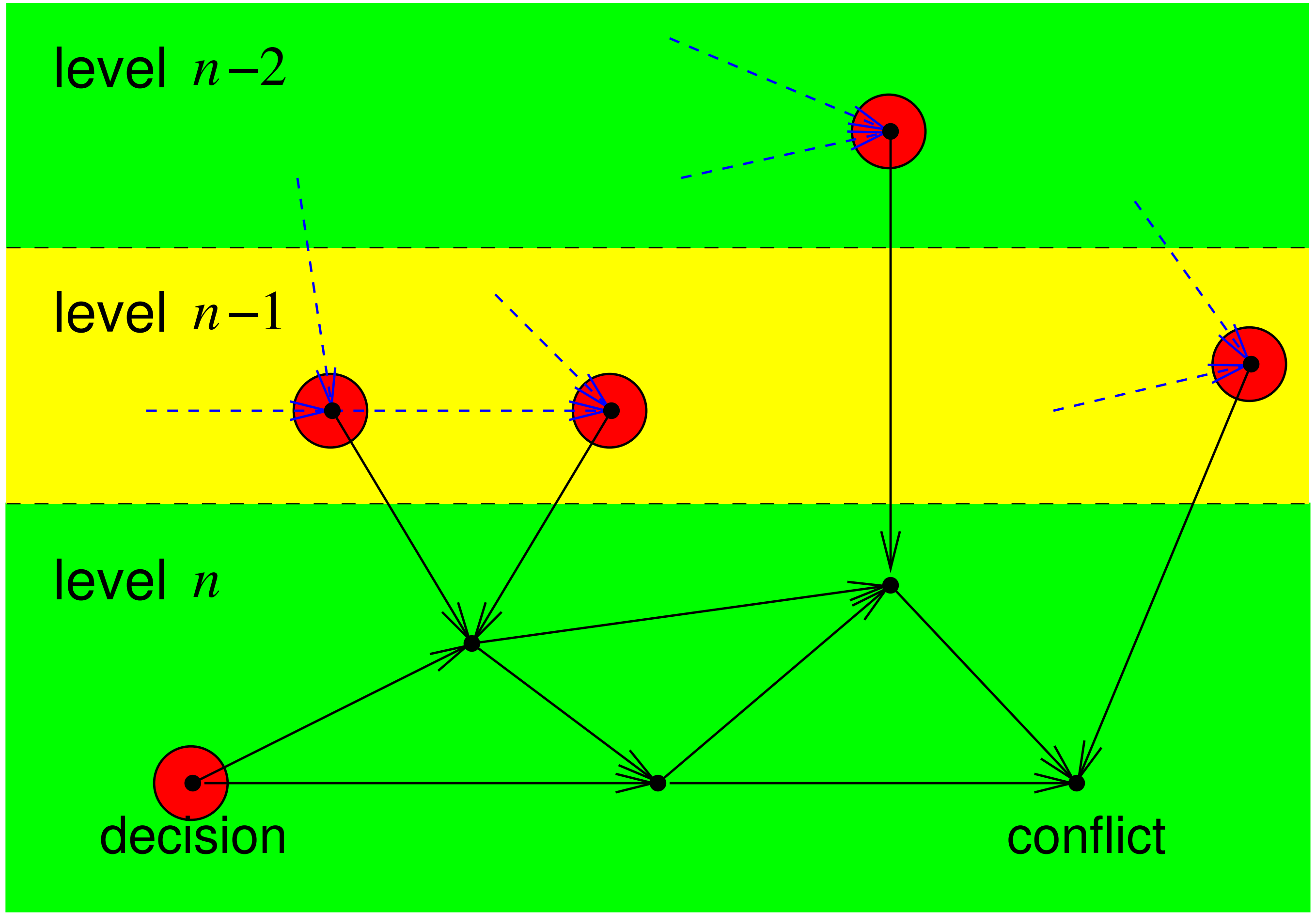
General Implication Graph as Hyper-Graph



Optimized Implication Graph for Unit Resolution in DP



Conflict Clauses as Cuts in the Implication Graph



a simple cut always exists: set of roots (decisions) contributing to the conflict

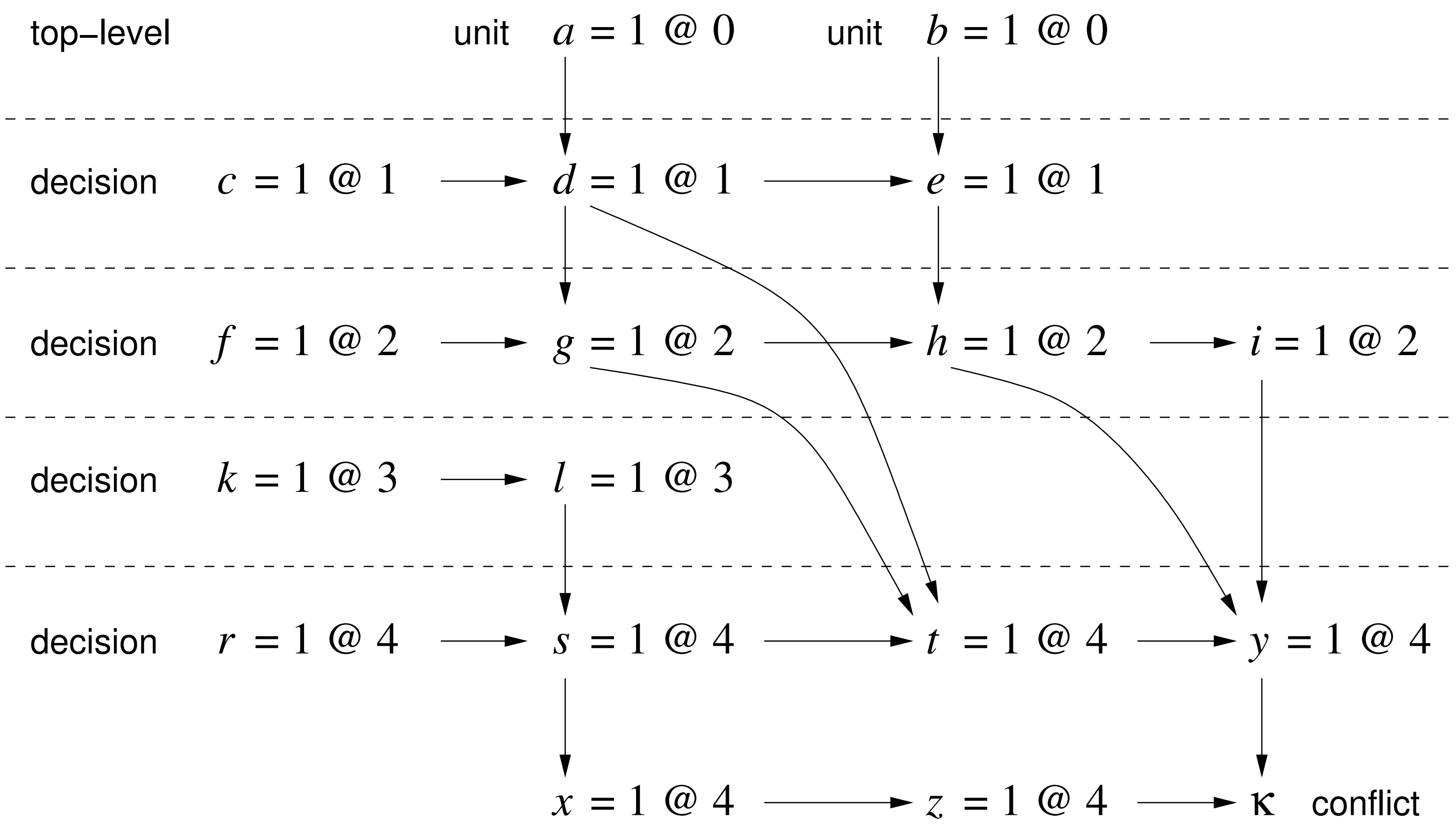
Modern CDCL Loop


```
Status Solver::search (long limit) {
  long conflicts = 0; Clause * conflict; Status res = UNKNOWN;
  while (!res)
    if (empty) res = UNSATISFIABLE;
    else if ((conflict = bcp ())) analyze (conflict), conflicts++;
    else if (conflicts >= limit) break;
    else if (reducing ()) reduce ();
    else if (restarting ()) restart ();
    else if (!decide ()) res = SATISFIABLE;
  return res;
}

Status Solver::solve () {
  long conflicts = 0, steps = 1e6;
  Status res;
  for (;;)
    if ((res = search (conflicts))) break;
    else if ((res = simplify (steps))) break;
    else conflicts += 1e4, steps += 1e6;
  return res;
}
```

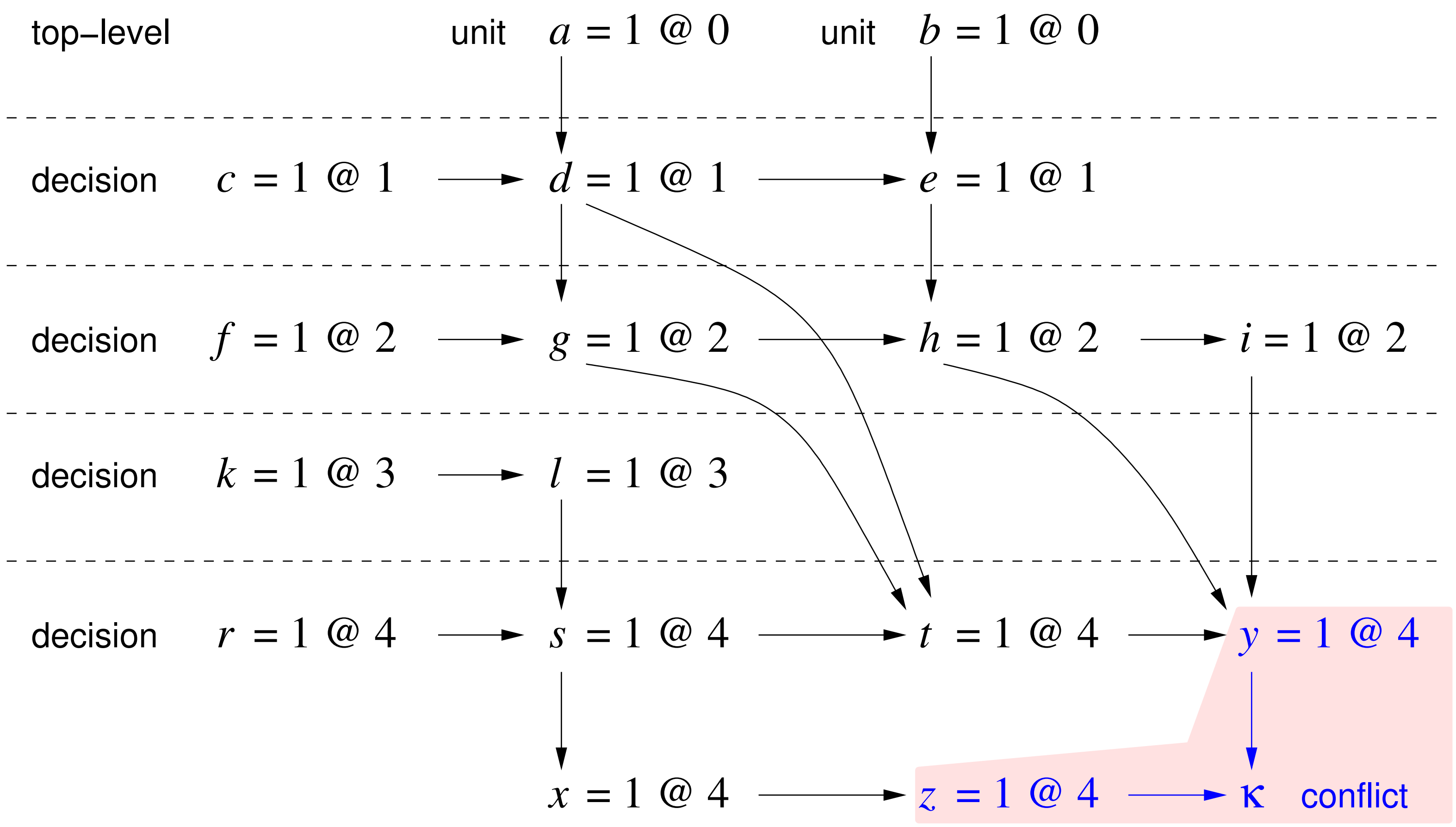
CaDiCaL CDCL Loop


```
int Internal::cdcl_loop_with_inprocessing () {
  int res = 0;
  START (search);
  while (!res)
           if (unsat) res = 20;
     else if (!propagate ()) analyze ();      // analyze propagated conflict
     else if (iterating) iterate ();          // report learned unit
     else if (satisfied ()) res = 10;         // all variables assigned
     else if (terminating ()) break;          // limit hit or asynchronous abort
     else if (restarting ()) restart ();      // restart by backtracking
     else if (rephasing ()) rephase ();       // reset variable phases
     else if (reducing ()) reduce ();         // collect useless learned clauses
     else if (probing ()) probe ();           // failed literal probing
     else if (subsuming ()) subsume ();       // subsumption algorithm
     else if (eliminating ()) elim ();        // bounded variable elimination
     else if (compactifying ()) compact ();   // collect internal variables
     else if (conditioning ()) condition (); // globally blocked clauses
     else decide ();                          // otherwise pick next decision
  STOP (search);
  return res;
}
```

Implication Graph

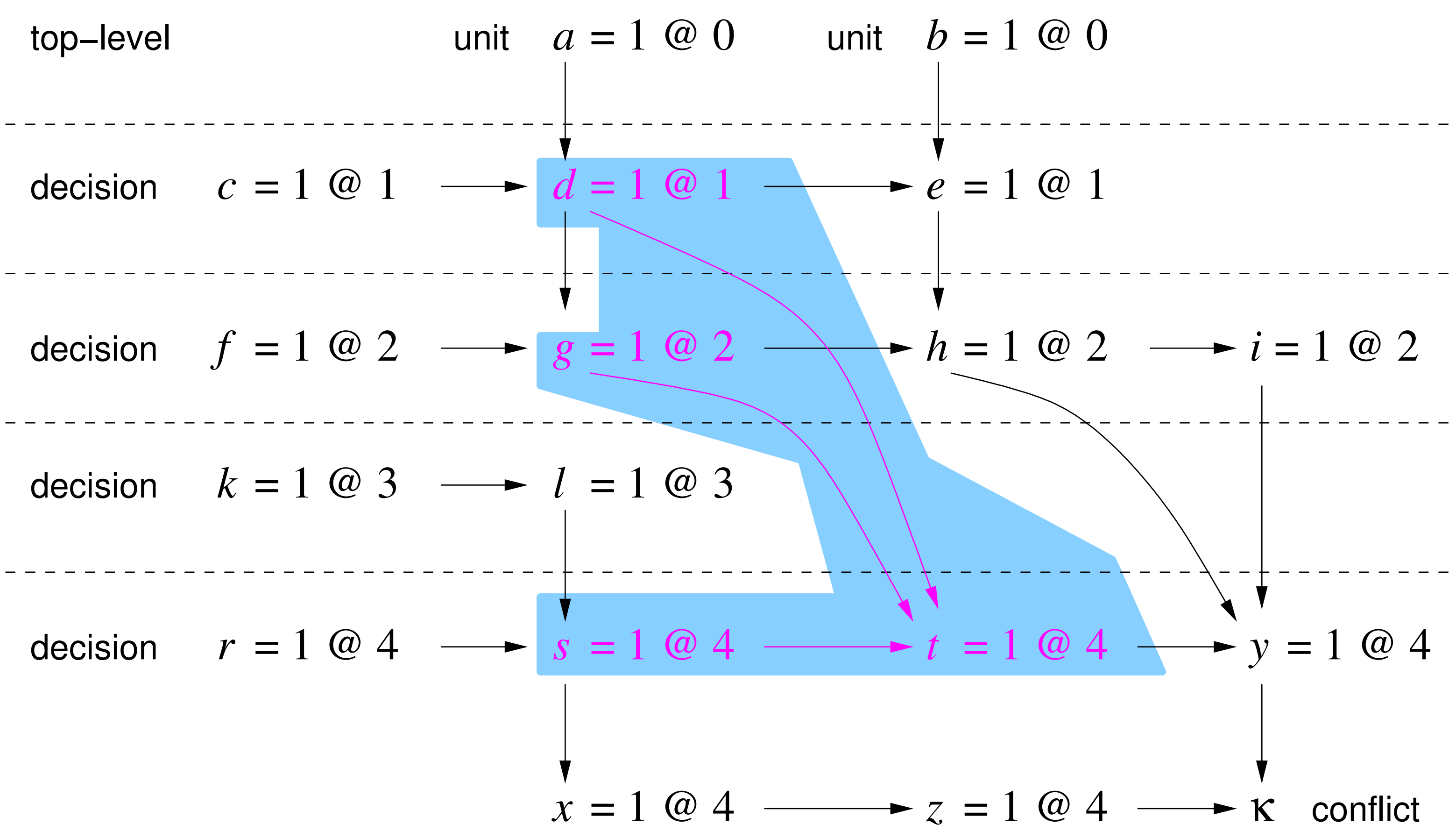


Conflict



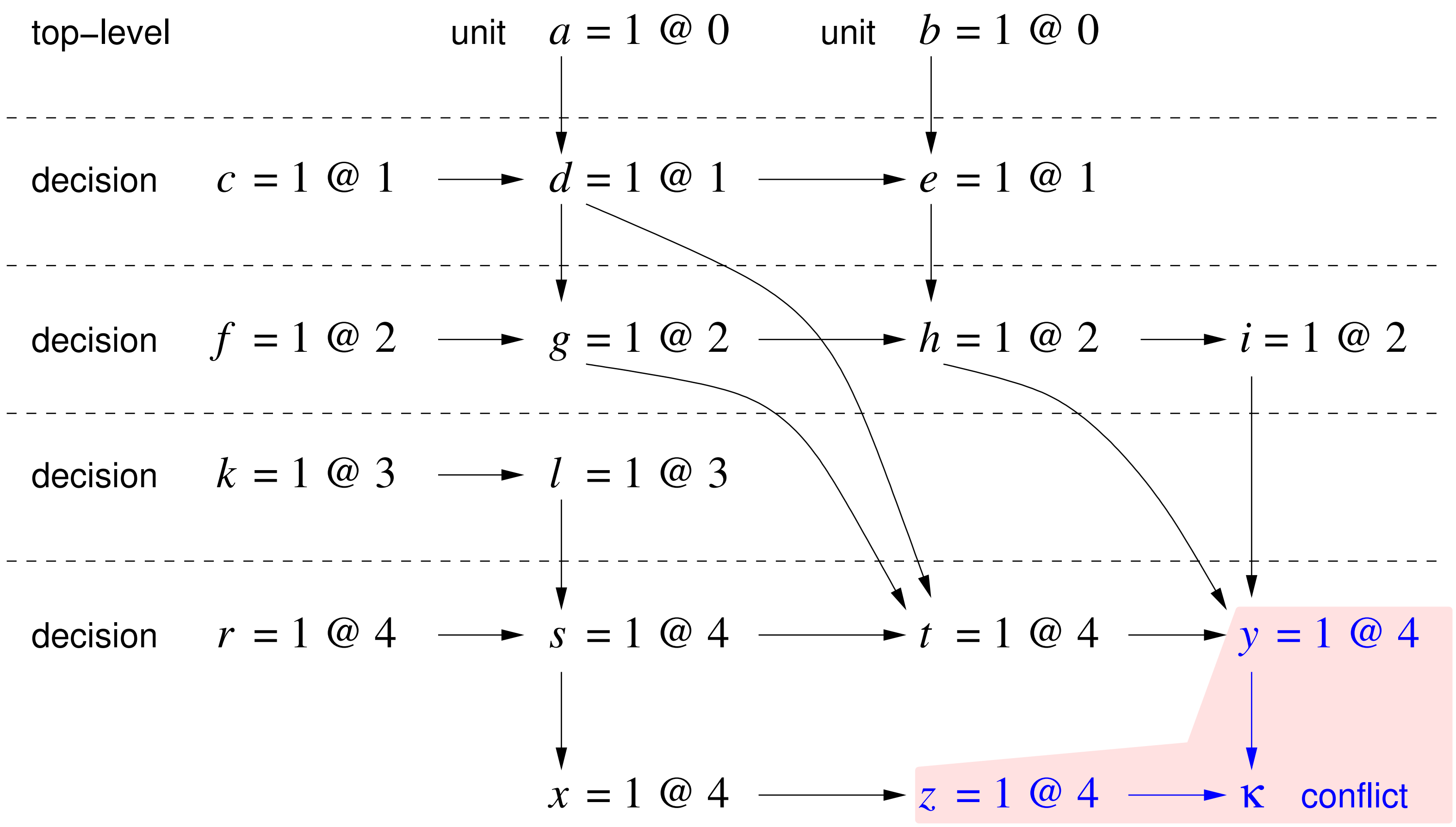
top–level
unit $a = 1 @ 0$
unit $b = 1 @ 0$
decision $c = 1 @ 1$
$d = 1 @ 1$
$e = 1 @ 1$
decision $f = 1 @ 2$
$g = 1 @ 2$
$h = 1 @ 2$
$i = 1 @ 2$
decision $k = 1 @ 3$
$l = 1 @ 3$
decision $r = 1 @ 4$
$s = 1 @ 4$
$t = 1 @ 4$
$y = 1 @ 4$
$x = 1 @ 4$
$z = 1 @ 4$
κ conflict

Antecedents / Reasons



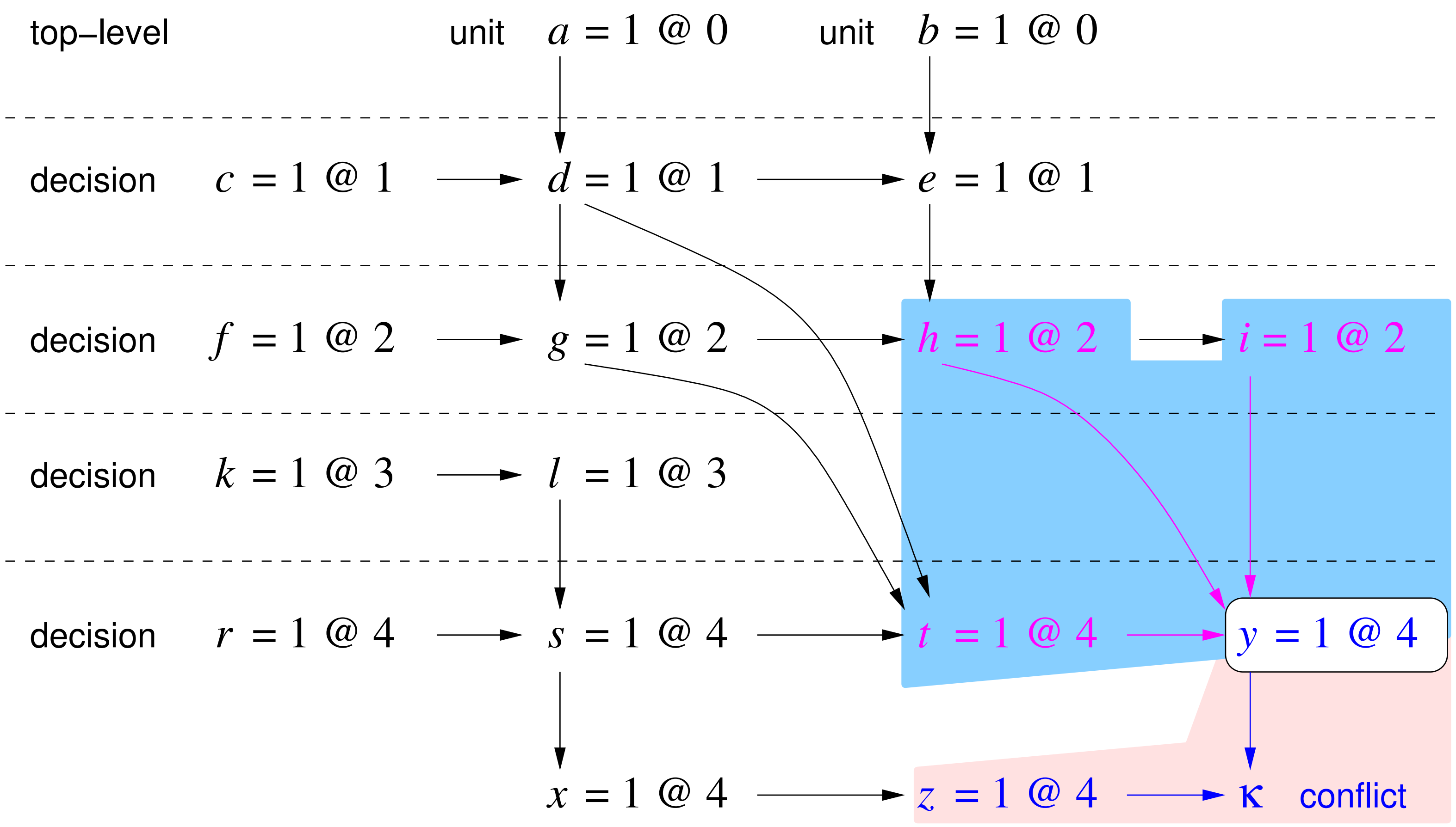
$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee t)$$

Conflicting Clauses



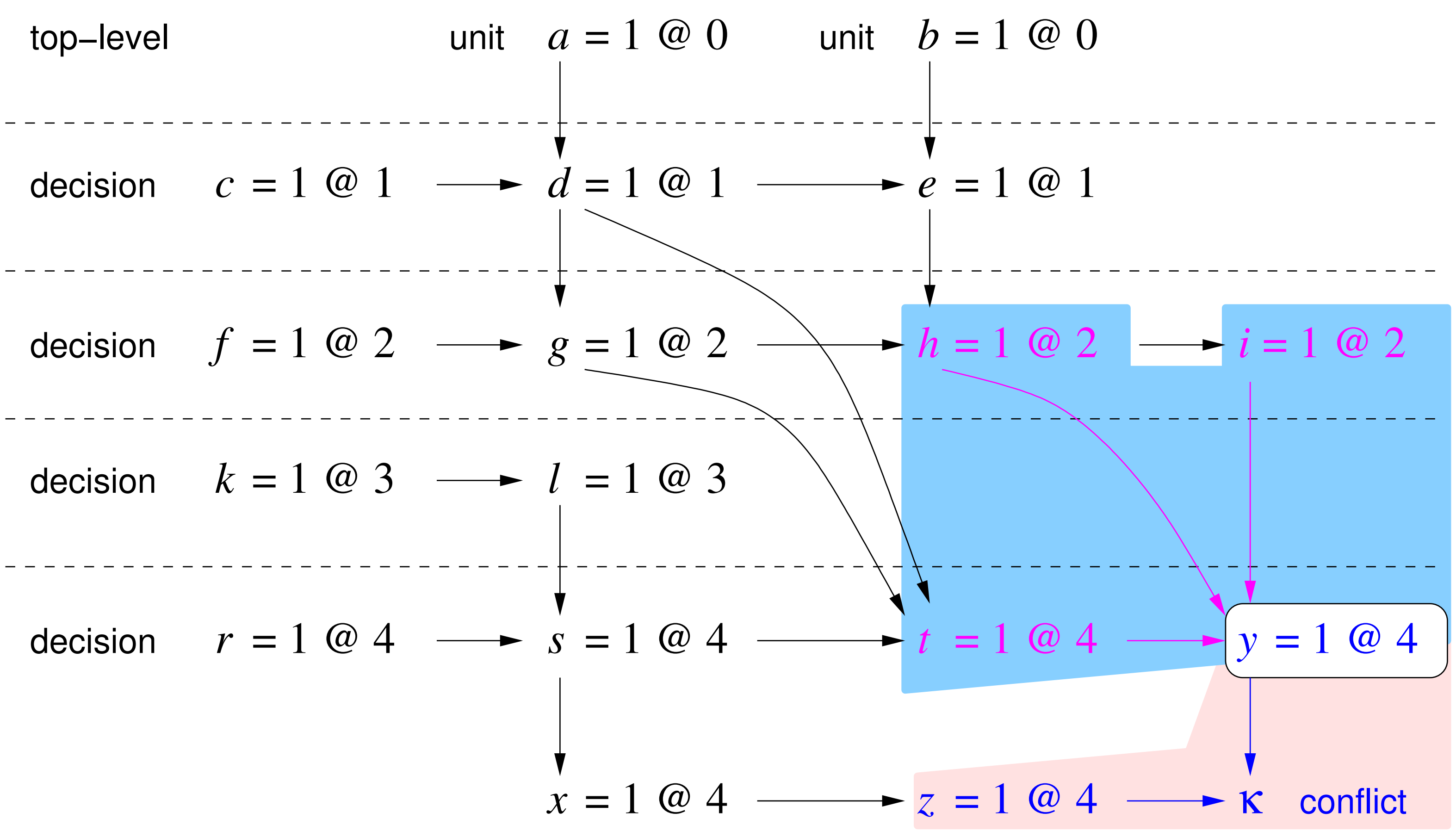
$$\neg(y \land z) \qquad \equiv \qquad (\bar{y} \lor \bar{z})$$

Resolving Antecedents 1st Time



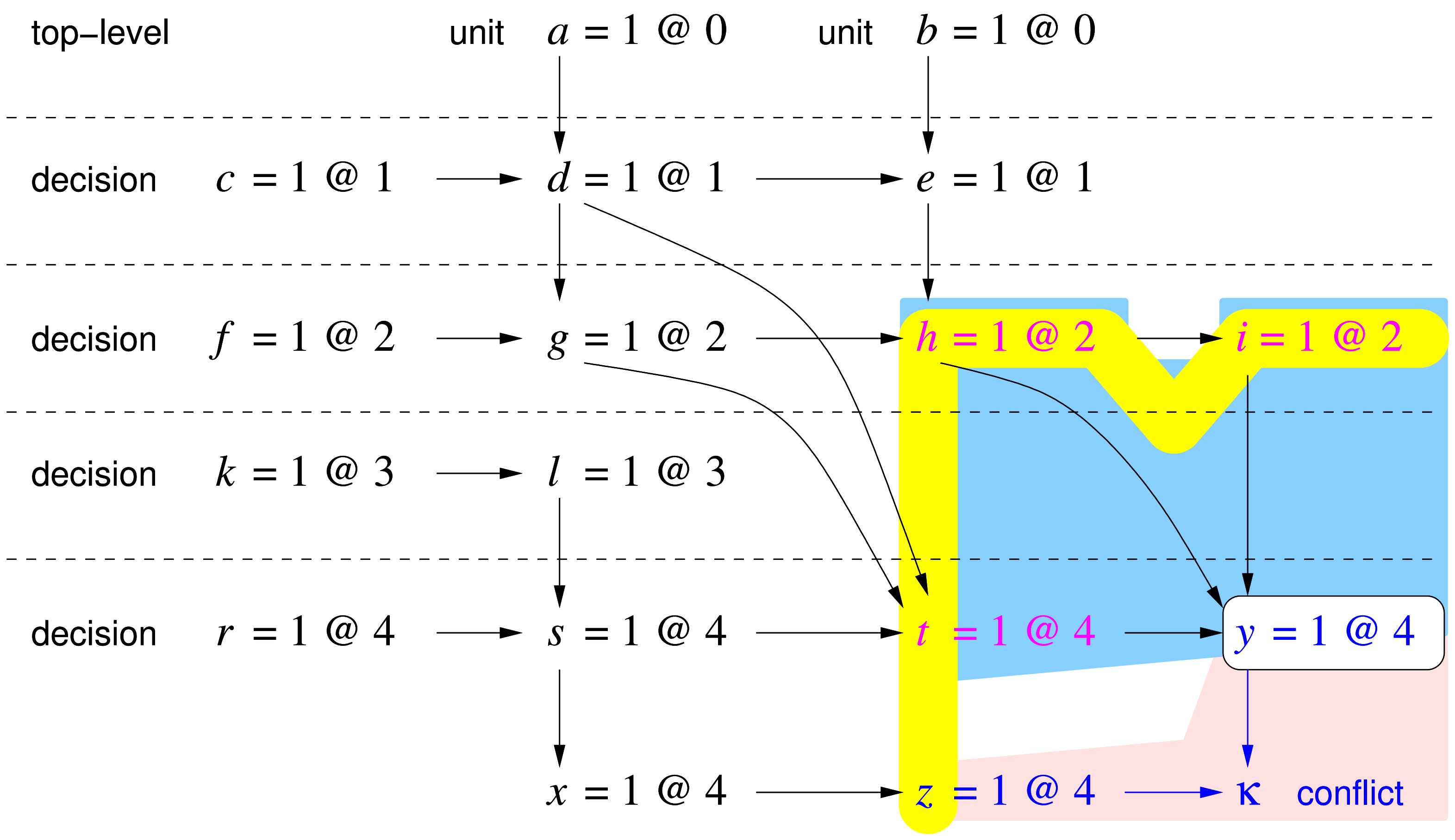
$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

Resolving Antecedents 1st Time



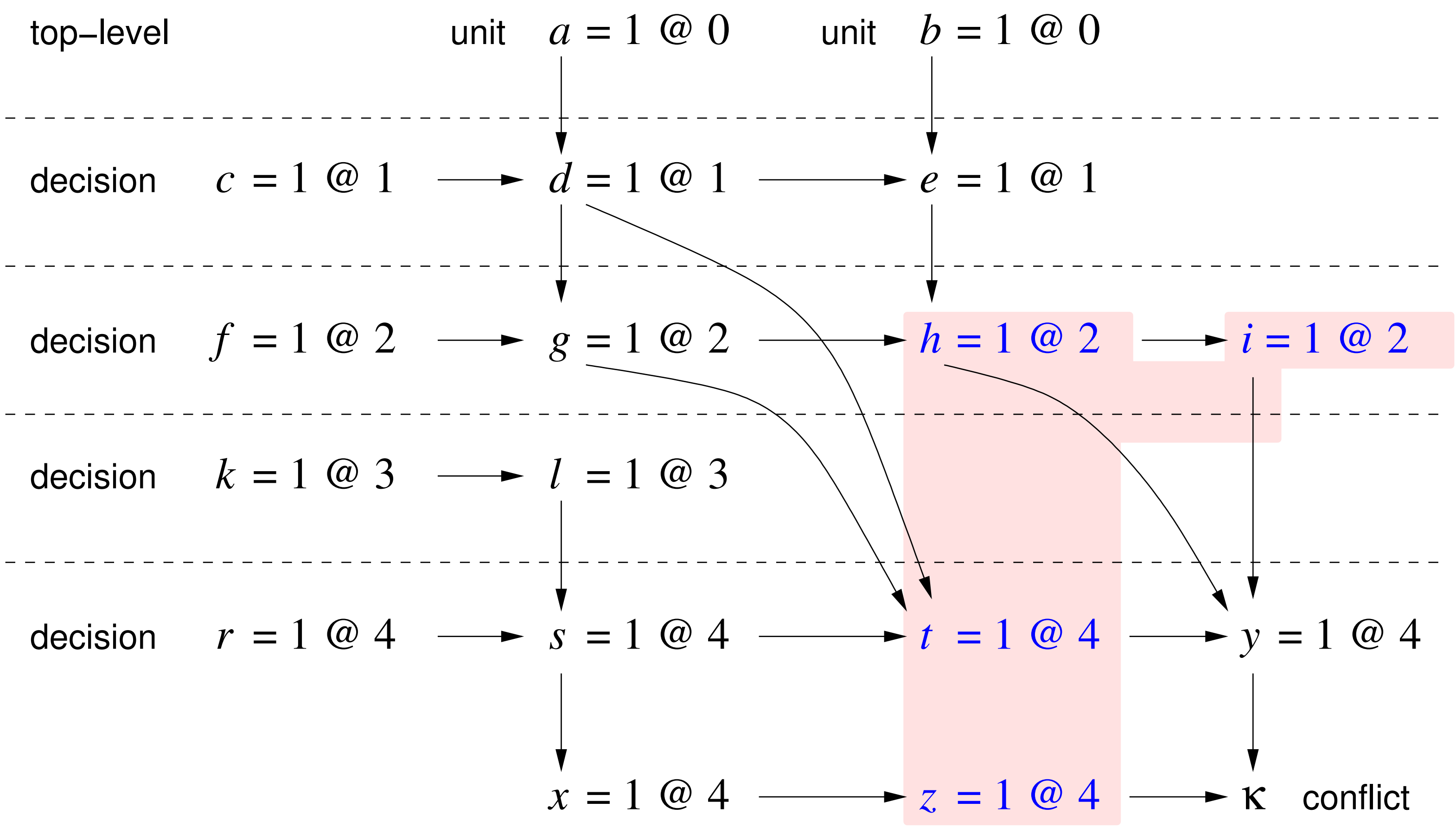
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Resolvents = Cuts = Potential Learned Clauses



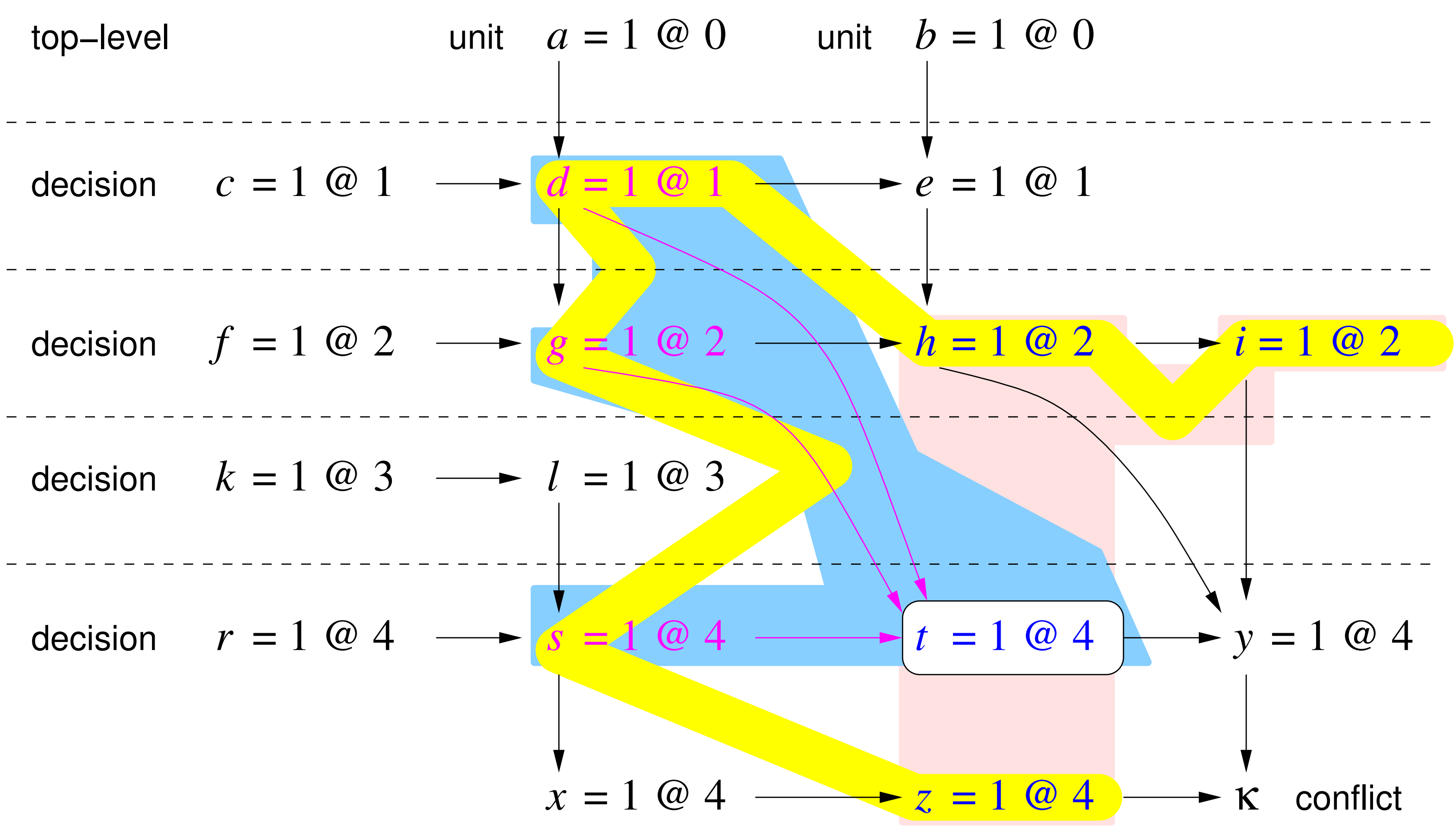
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Potential Learned Clause After 1 Resolution



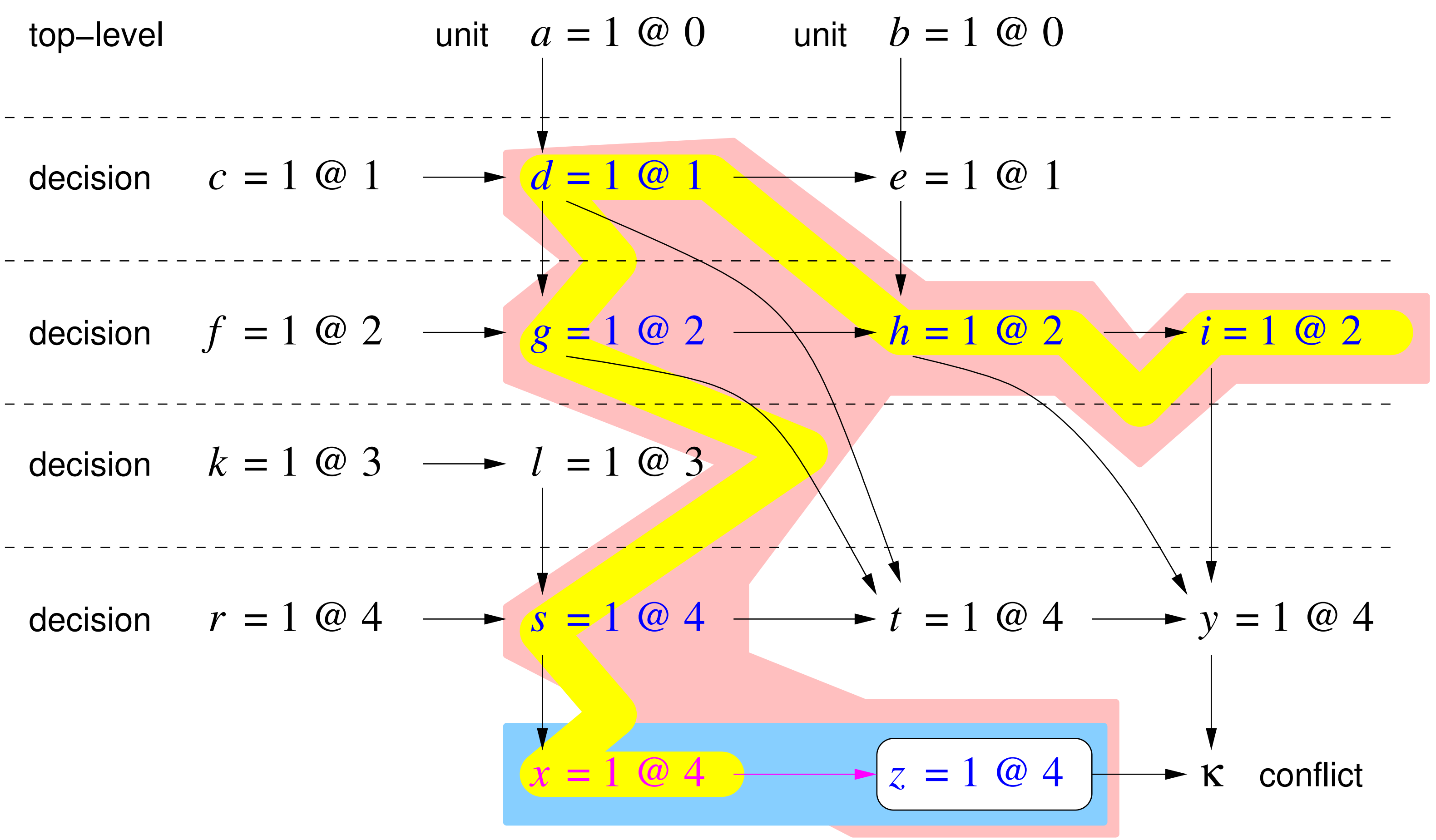
$$(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})$$

Resolving Antecedents 2nd Time



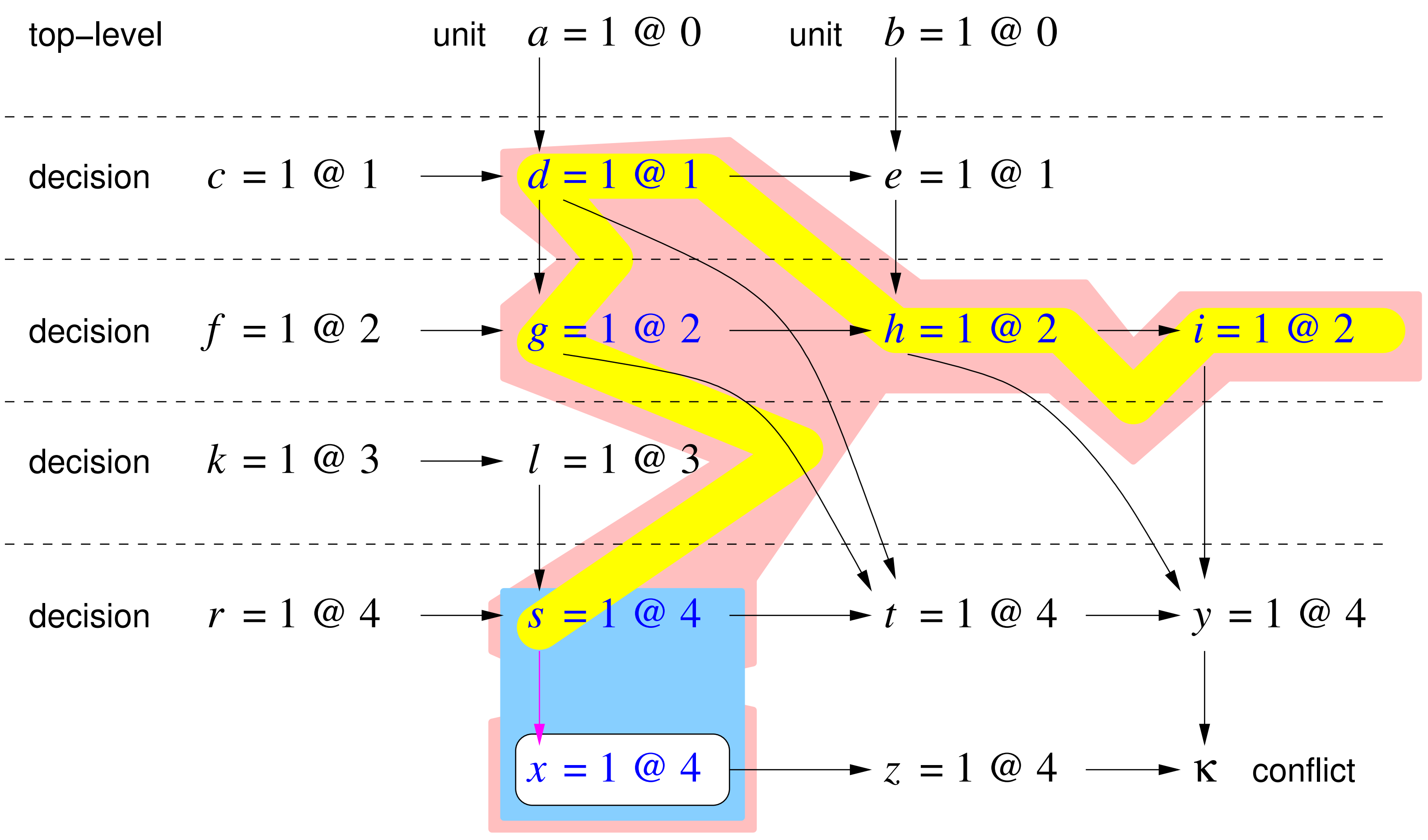
$$\frac{(\bar{d} \vee \bar{g} \vee \bar{s} \vee t) \qquad (\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}{\boxed{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}}$$

Resolving Antecedents 3rd Time



$$\frac{(\bar{x} \vee z) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}{(\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

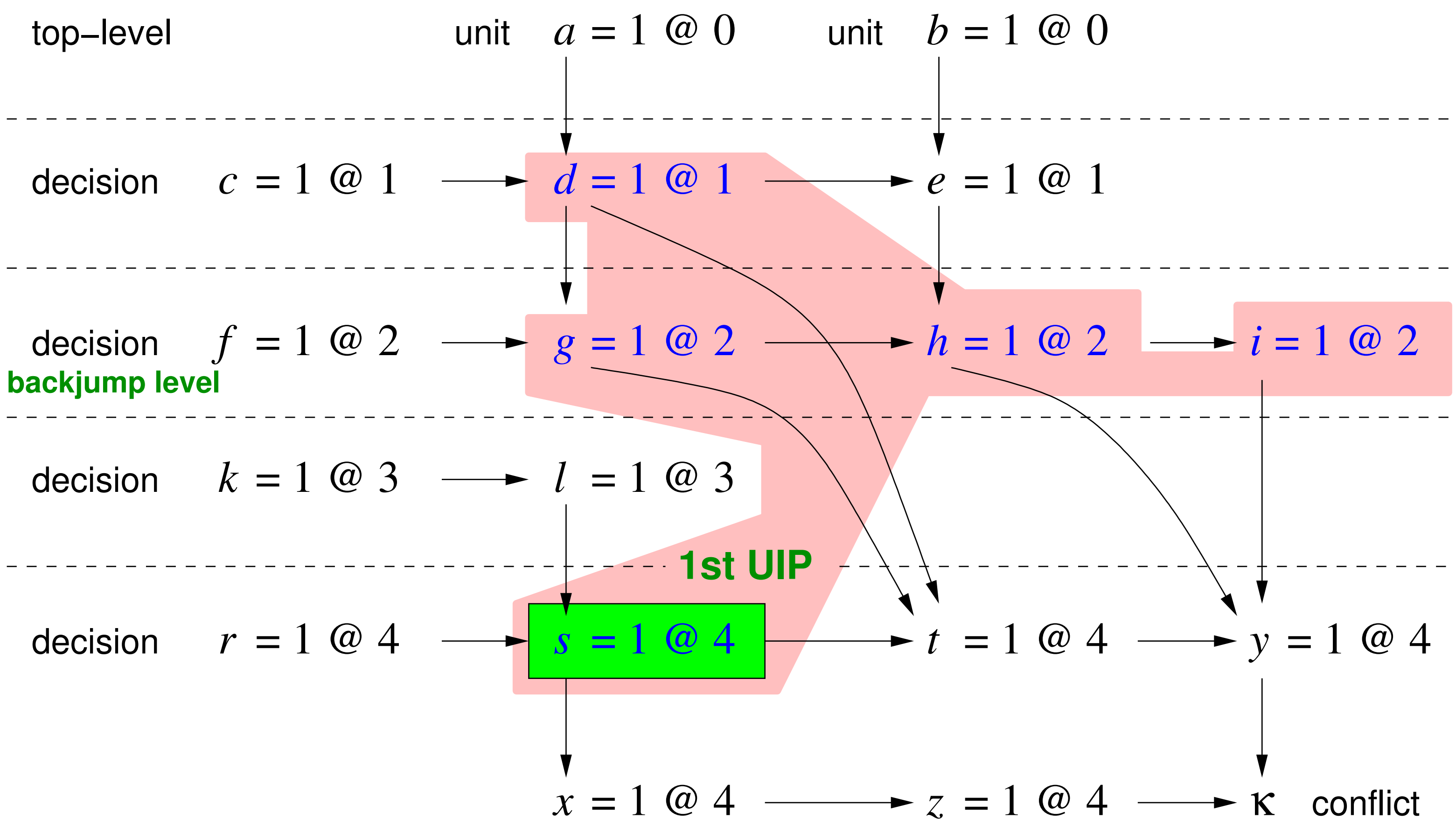
Resolving Antecedents 4th Time



$$\frac{(\bar{s} \vee x) \qquad (\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

self subsuming resolution

1st UIP Clause after 4 Resolutions



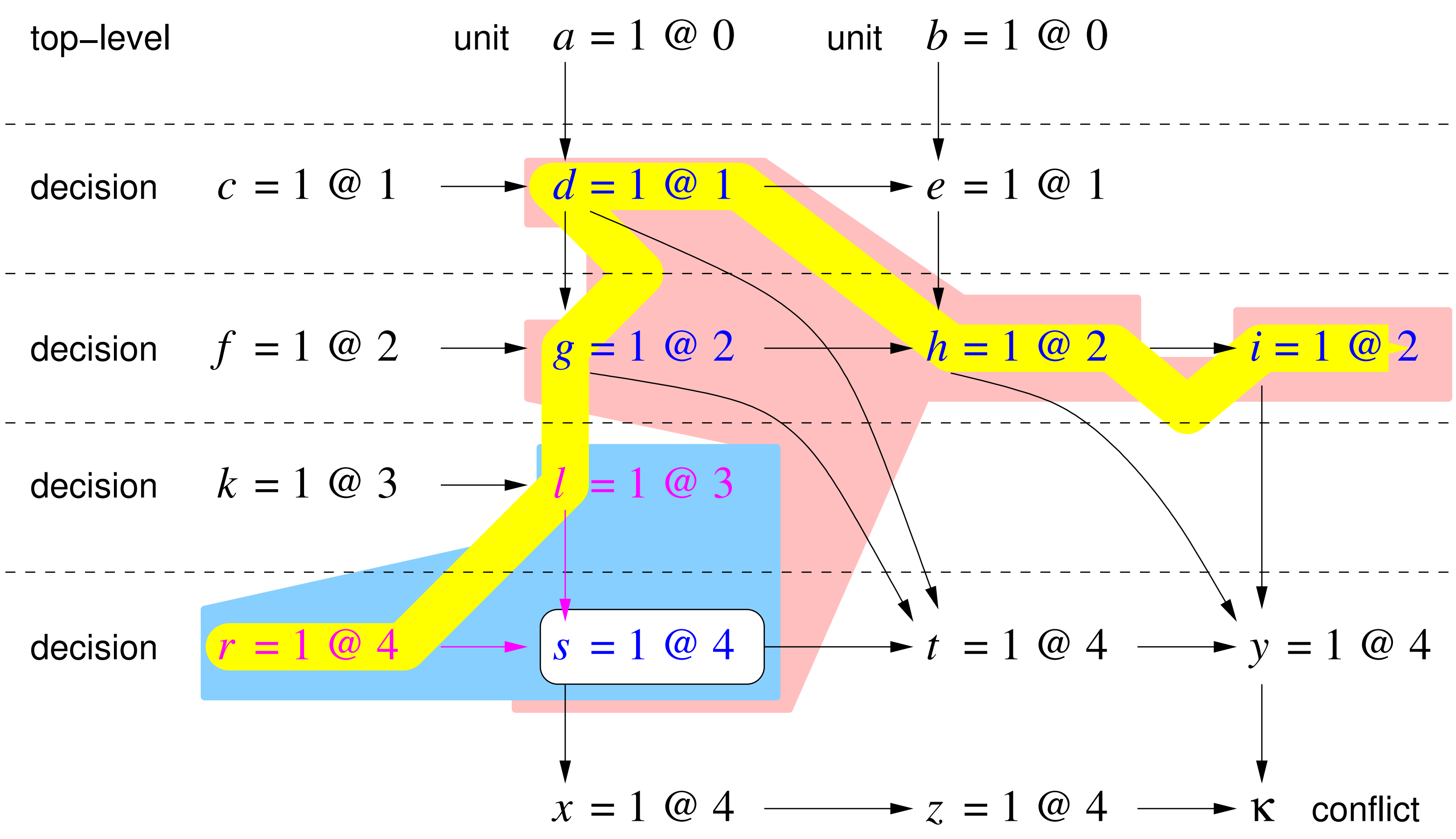
$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

UIP = *unique implication point*   dominates conflict on the last level

Detection of UIPs

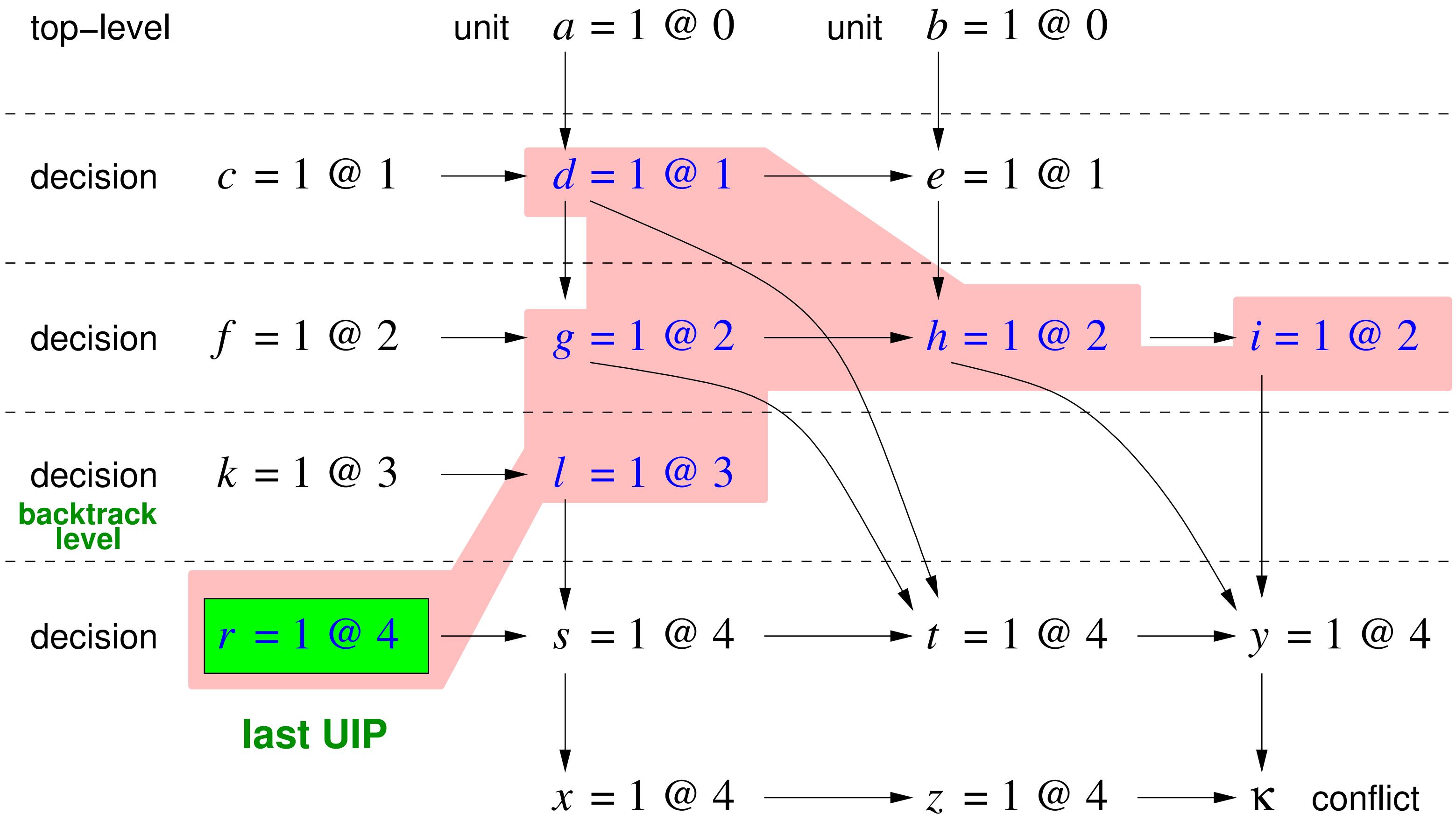
- can be found by graph traversal in the order of made assignments
  - *trail* respects this order
  - traverse reasons of variables on trail starting with conflict
- count “open paths”
  - initially size of clause with only false literals
  - decrease counter if new reason / antecedent clause resolved
  - if all paths converged, i.e. counter = 1, then this node is a UIP
  - decision of current decision level is a UIP and thus a *sentinel*

Resolving Antecedents 5th Time



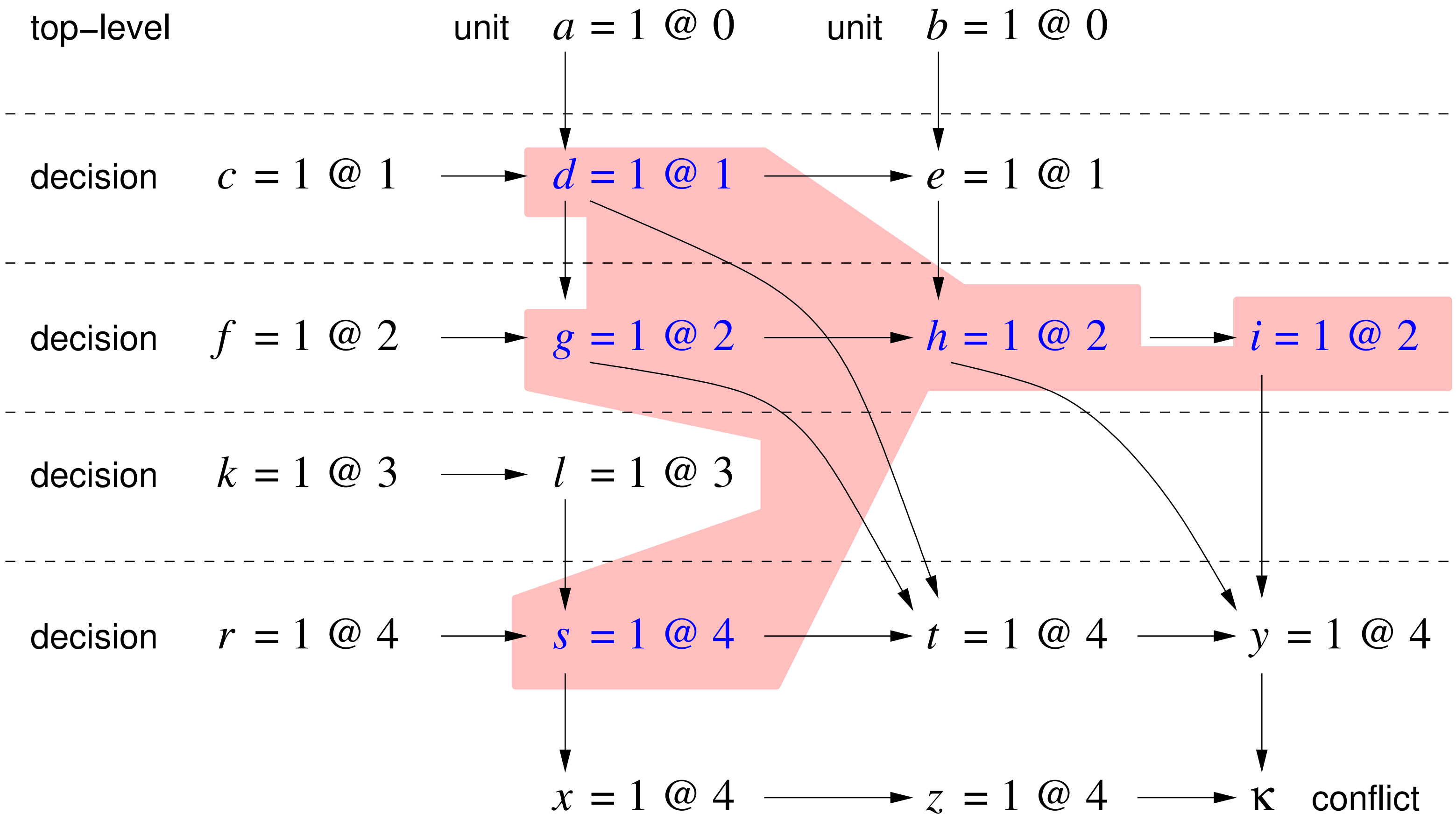
$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

Decision Learned Clause



$$(\bar{d} \vee \bar{g} \vee \bar{l} \vee \bar{r} \vee \bar{h} \vee \bar{i})$$

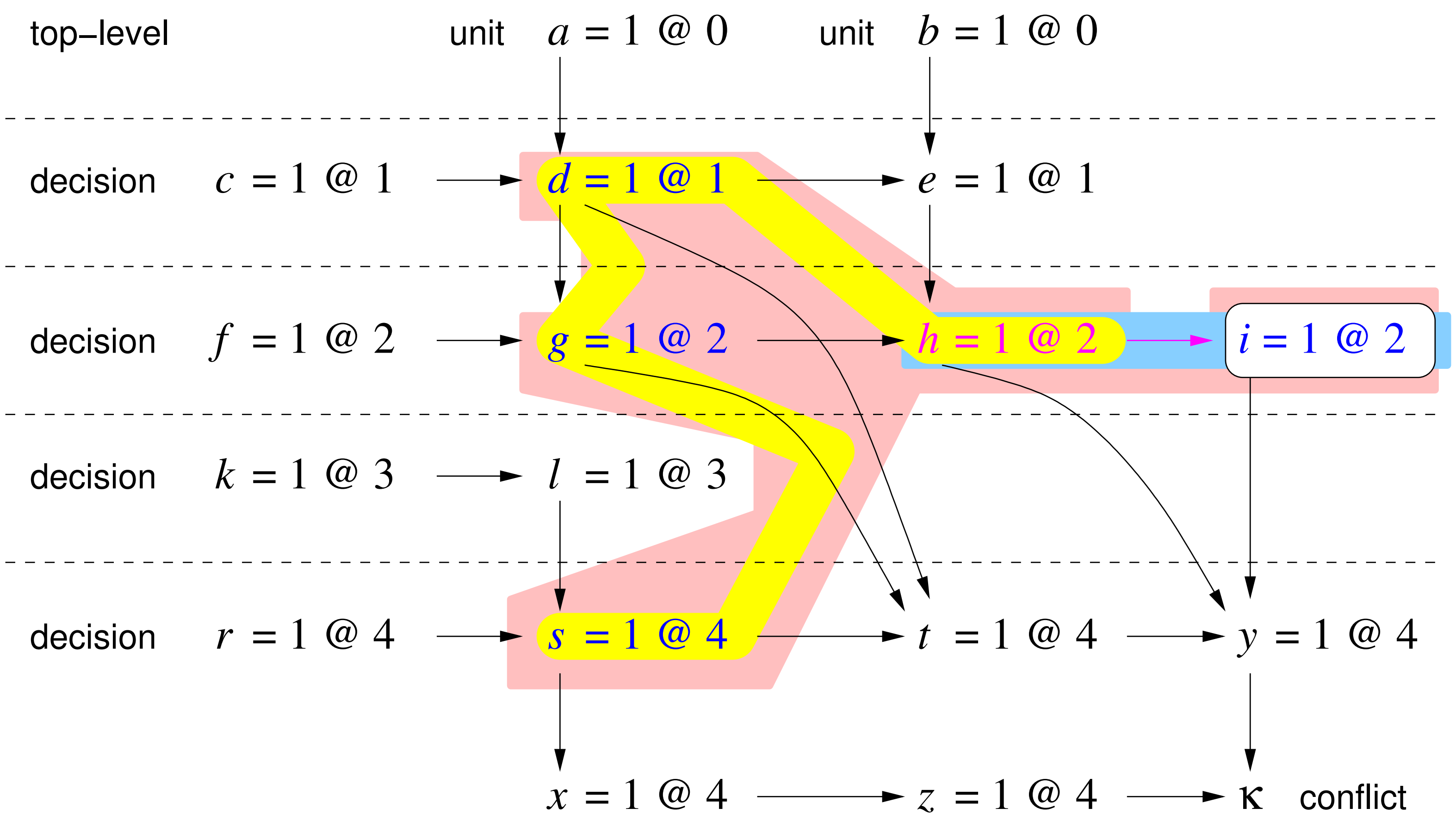
1st UIP Clause after 4 Resolutions



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

Locally Minimizing 1st UIP Clause

Sörensson'06, BiereSörensson'09

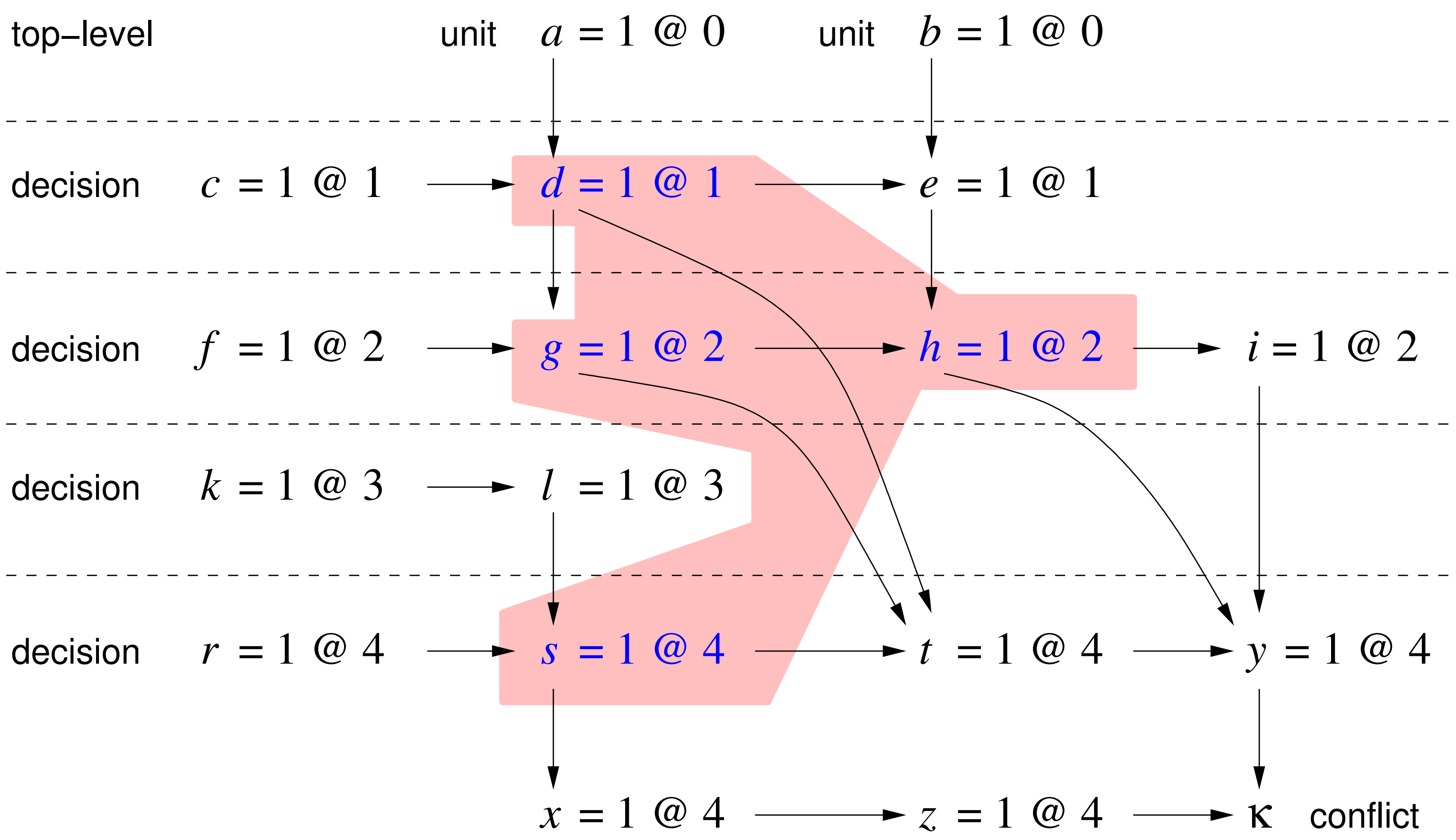


$$\frac{(\bar{h} \vee i) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h})}$$

self subsuming resolution

Locally Minimized Learned Clause

Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})$$

Local Minimization Algorithm

Sörensson'06, BiereSörensson'09

Two step algorithm:

1. mark all variables in 1^{st} UIP clause

2. remove literals with all antecedent literals also marked

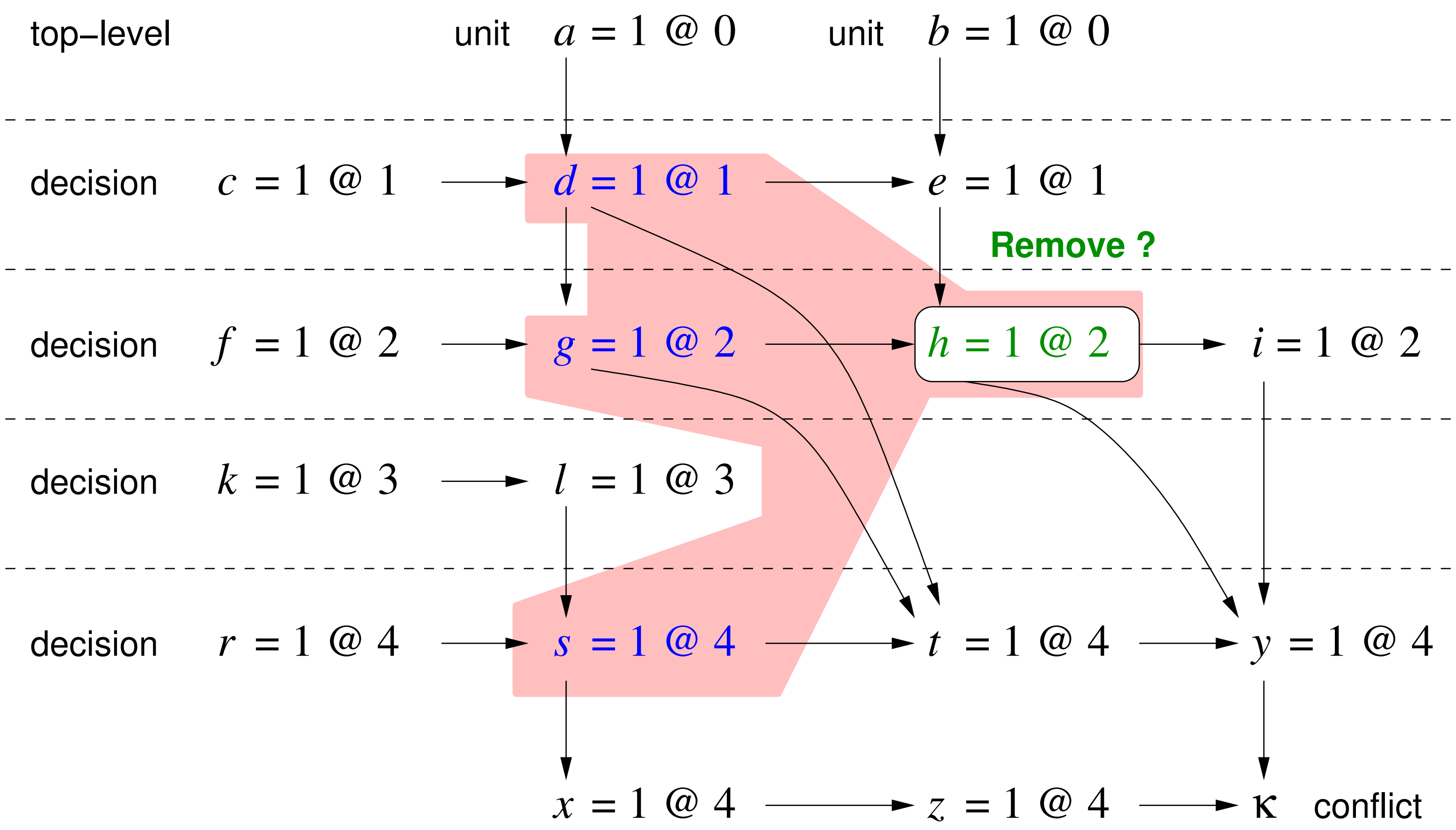
Correctness:

- removal of literals in step 2 are self subsuming resolution steps.
- implication graph is acyclic.

Confluence: produces a unique result.

Minimizing Locally Minimized Learned Clause Further?

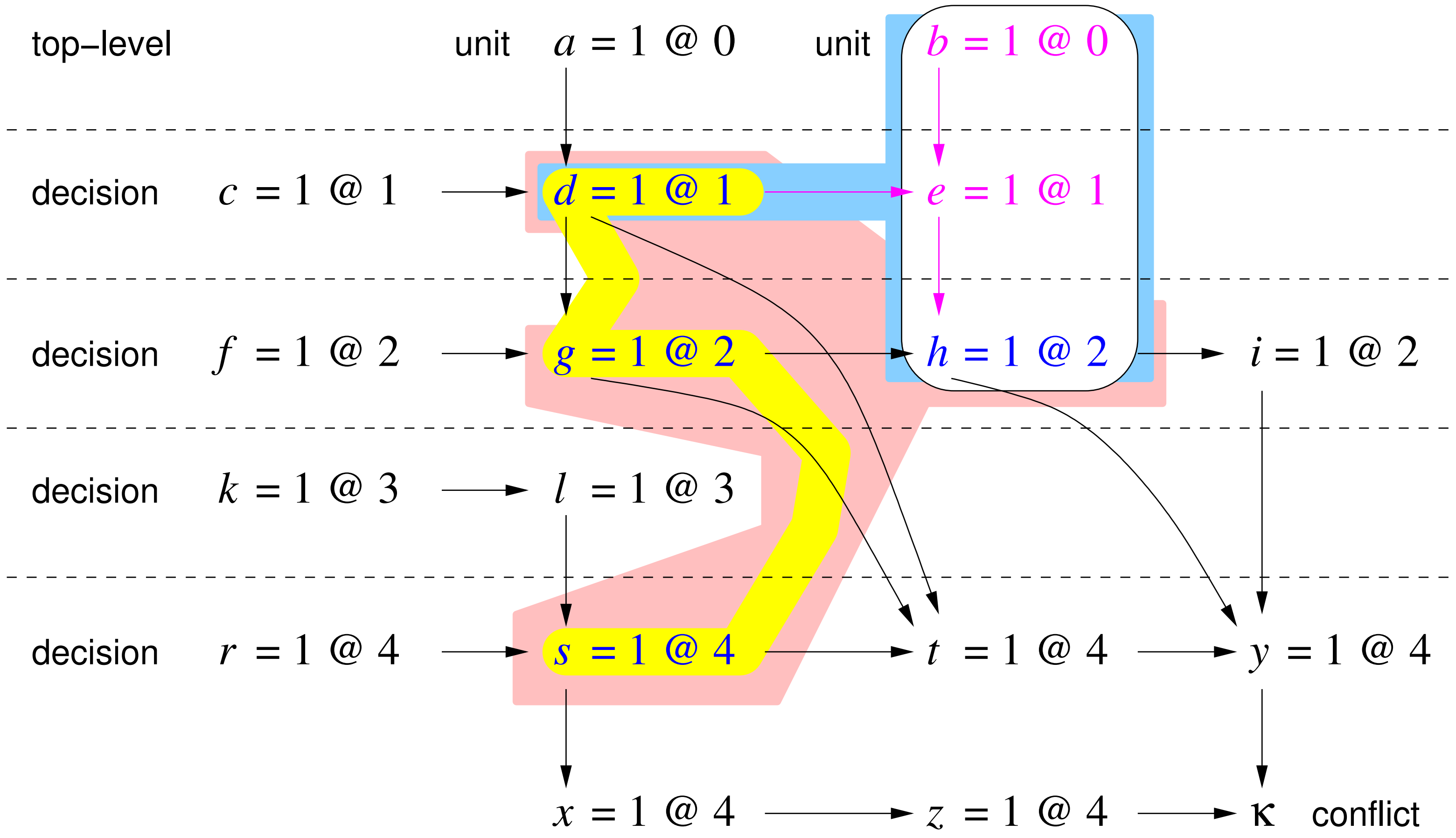
Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$

Recursively Minimizing Learned Clause

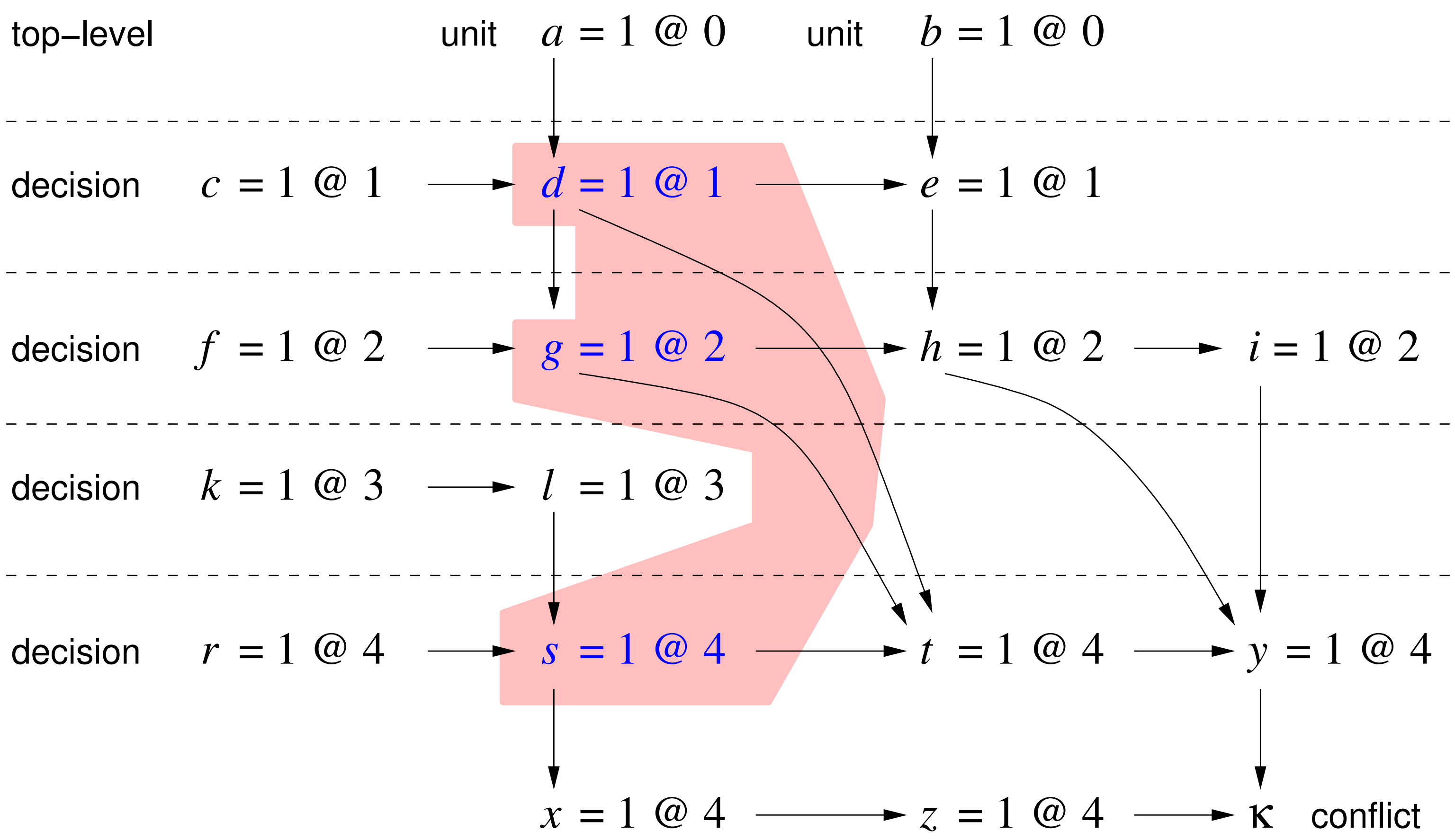
Sörensson’06, BiereSörensson’09



$$\dfrac{(b) \qquad \dfrac{(\bar d \vee \bar b \vee e) \qquad \dfrac{(\bar e \vee \bar g \vee h) \qquad (\bar d \vee \bar g \vee \bar s \vee \bar h)}{(\bar e \vee \bar d \vee \bar g \vee \bar s)}}{(\bar b \vee \bar d \vee \bar g \vee \bar s)}}{(\bar d \vee \bar g \vee \bar s)}$$

Recursively Minimized Learned Clause

Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s})$$

Recursive Minimization Algorithm

[MiniSAT 1.13]

Sörensson'06, BiereSörensson'09

Four step algorithm:

1. mark all variables in 1st UIP clause
2. for each candidate literal: search implication graph
3. start at antecedents of candidate literals
4. if search always terminates at marked literals remove candidate

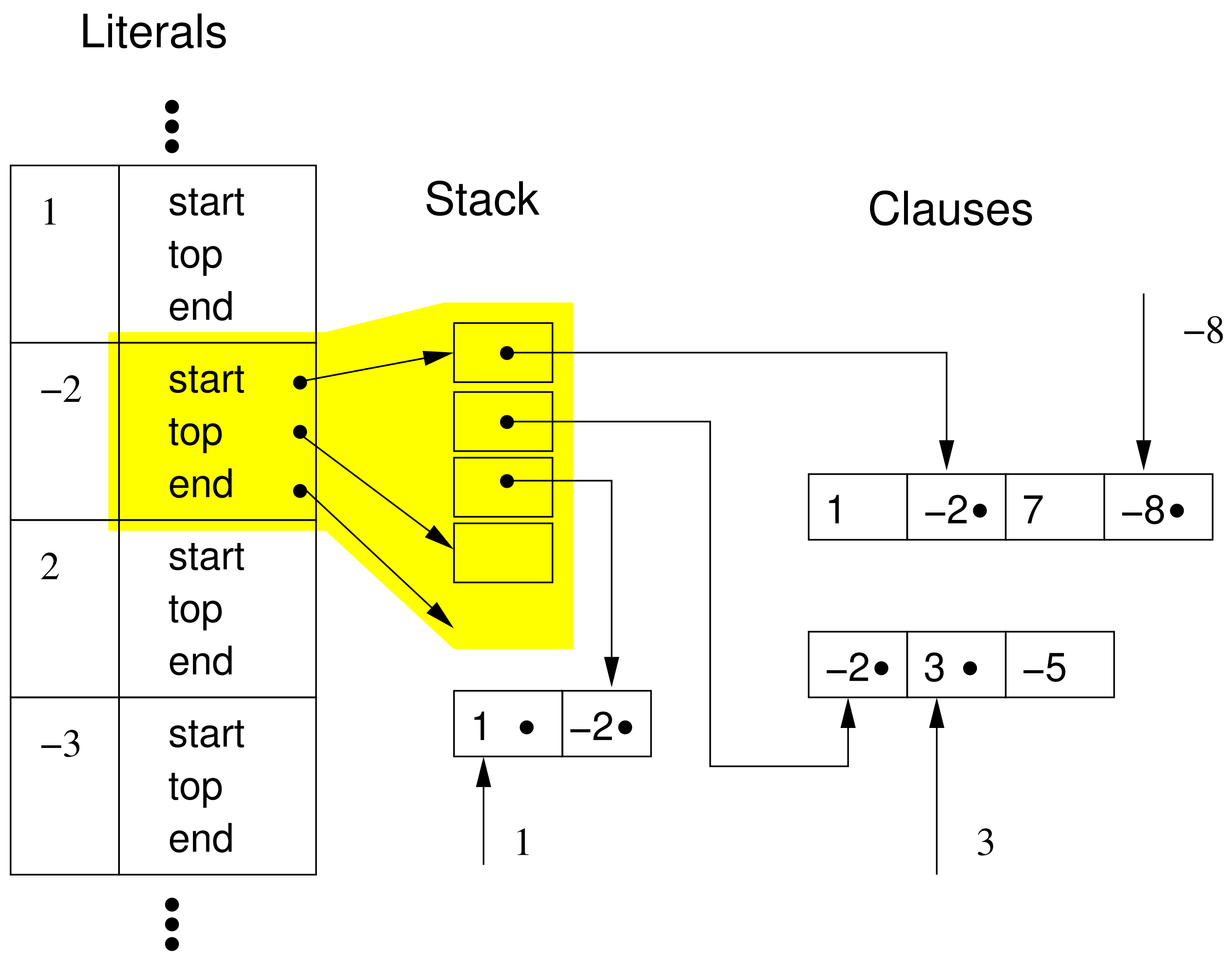
Correctness and Confluence as in local version!!!

Optimization: terminate early with failure if new decision level is "pulled in"

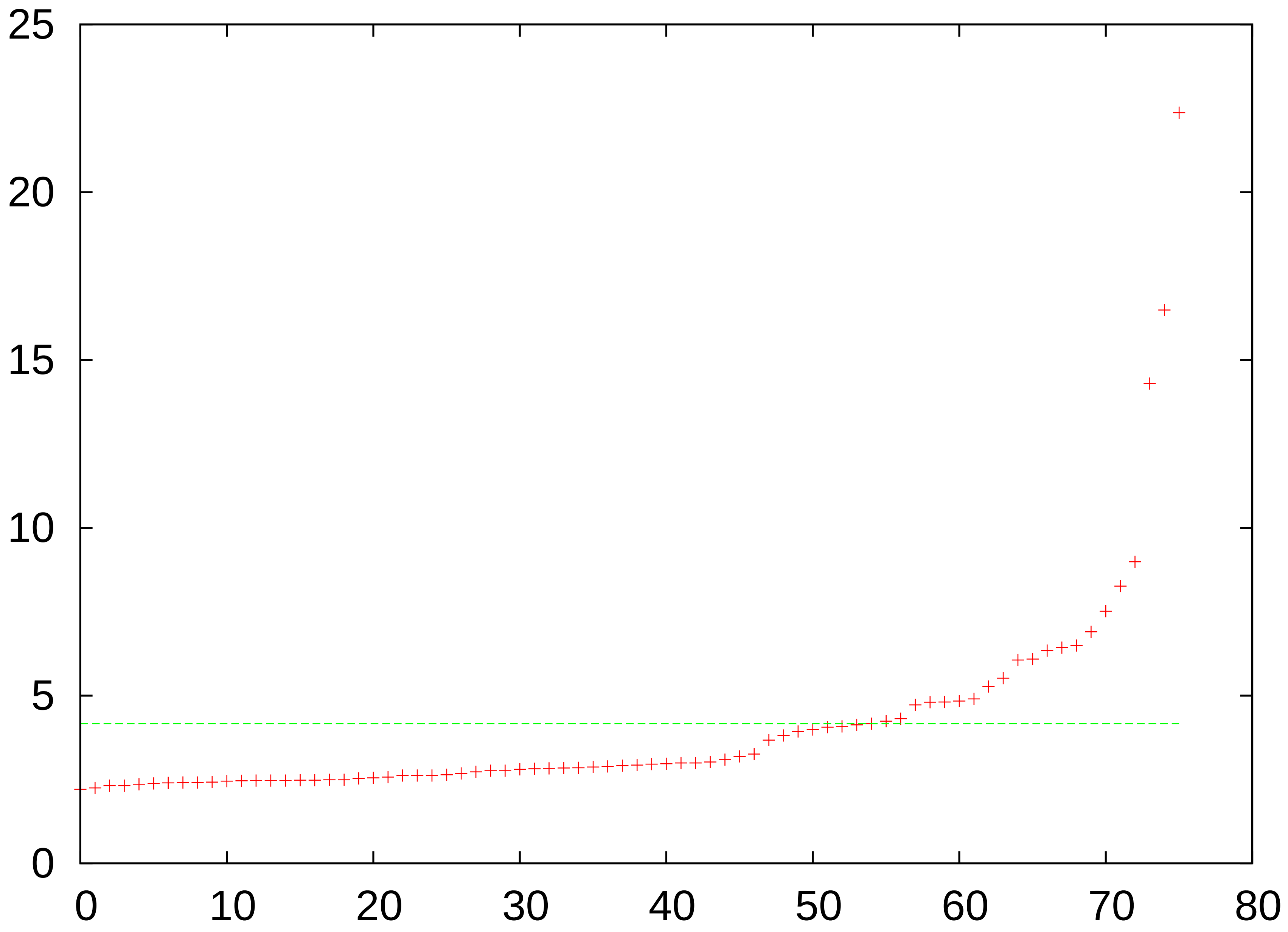
Two-Watched Literal Schemes

- original idea from SATO [ZhangStickel'00]
  - maintain the invariant: always watch two non-false literals
  - if a watched literal becomes *false* replace it
  - if no replacement can be found clause is either unit or empty
  - original version used *head* and *tail* pointers on Tries
- improved variant from Chaff [MoskewiczMadiganZhaoZhangMalik'01]
  - watch pointers can move arbitrarily — SATO: *head* forward, *trail* backward
  - no update needed during backtracking
- *one* watch is enough to ensure correctness — but looses *arc consistency*
- reduces *visiting* clauses by 10x, particularly useful for large and many learned clauses

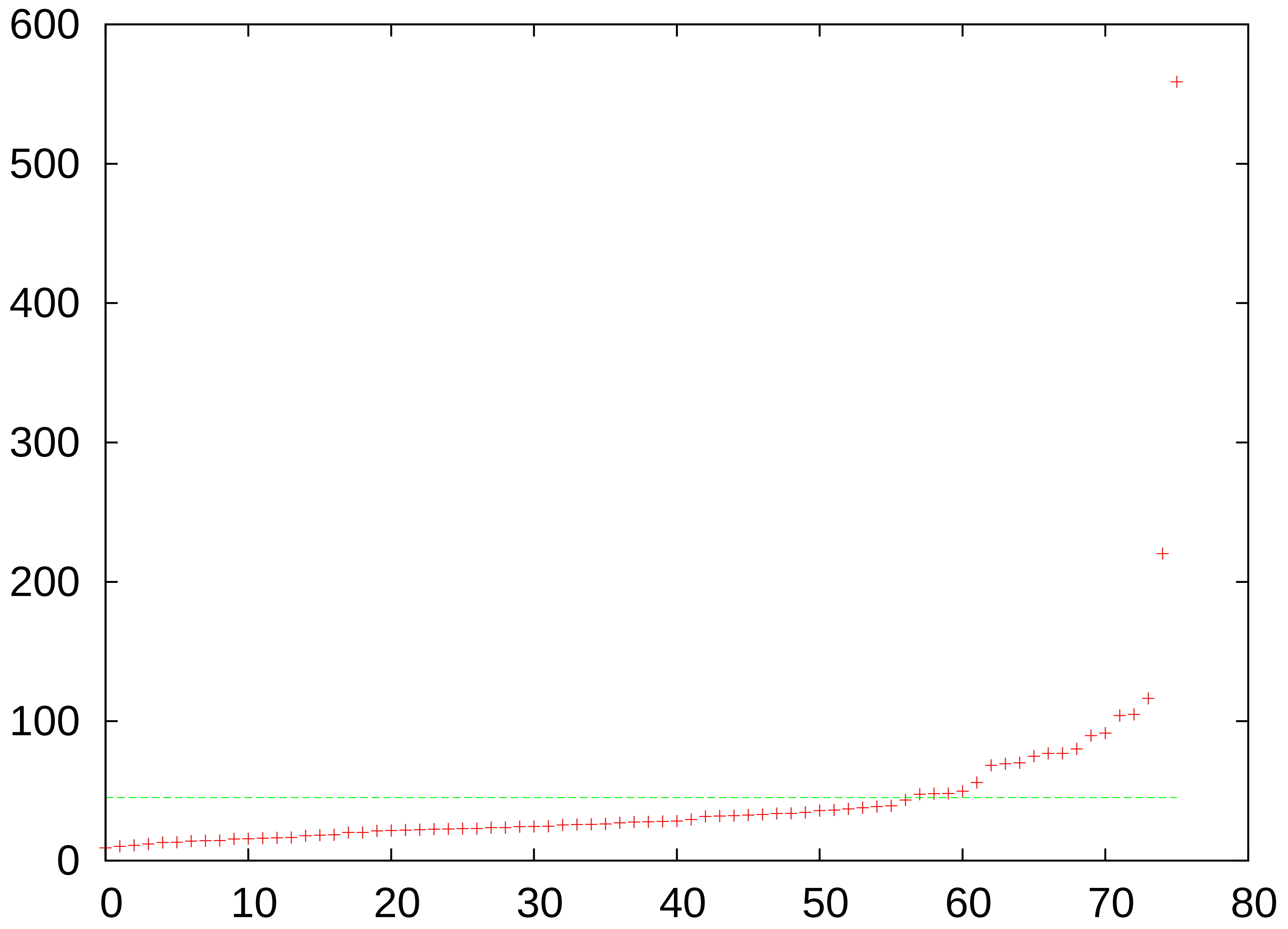
ZChaff Occurrence Stacks



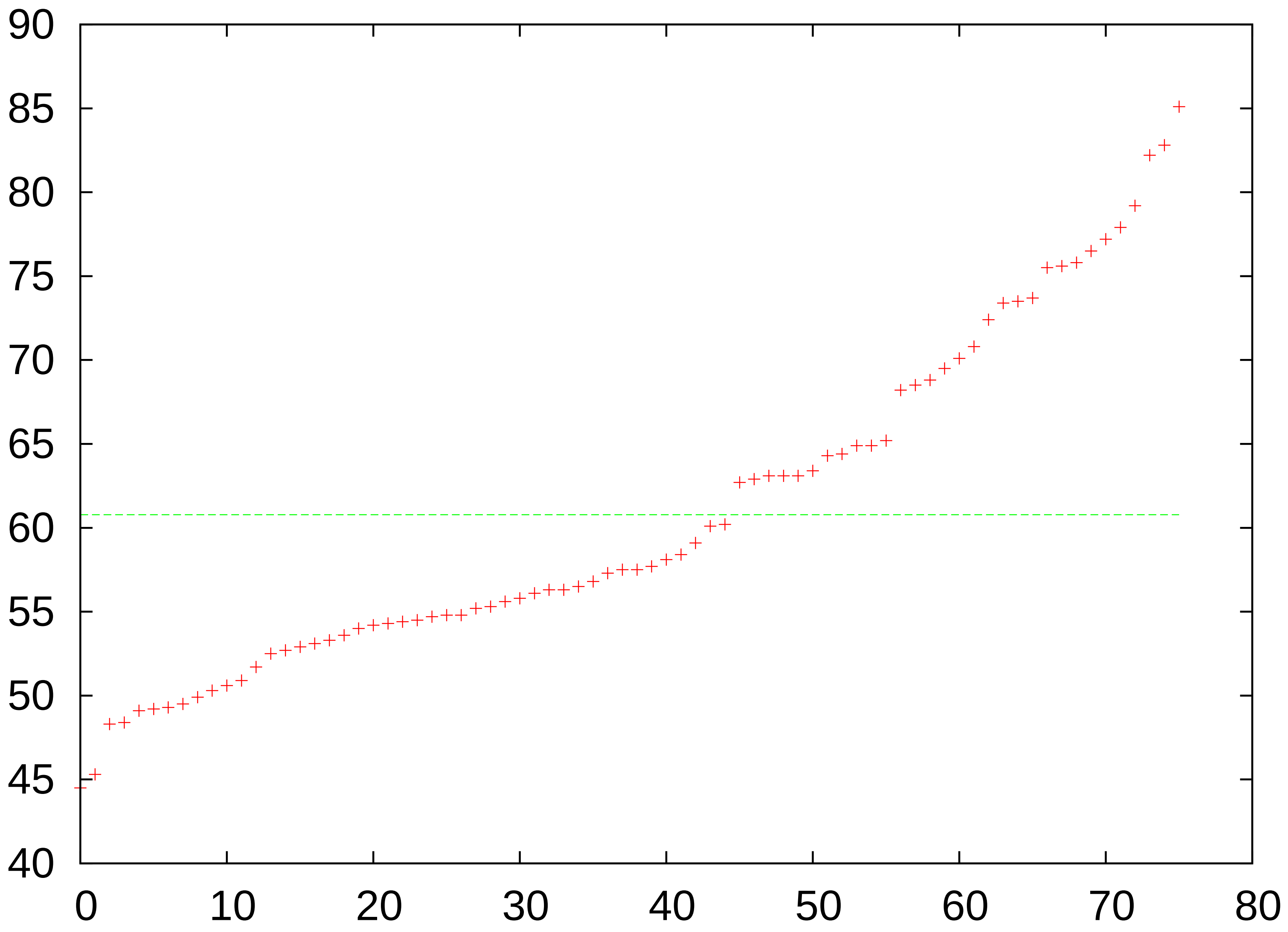
Average Number Clauses Visited Per Propagation



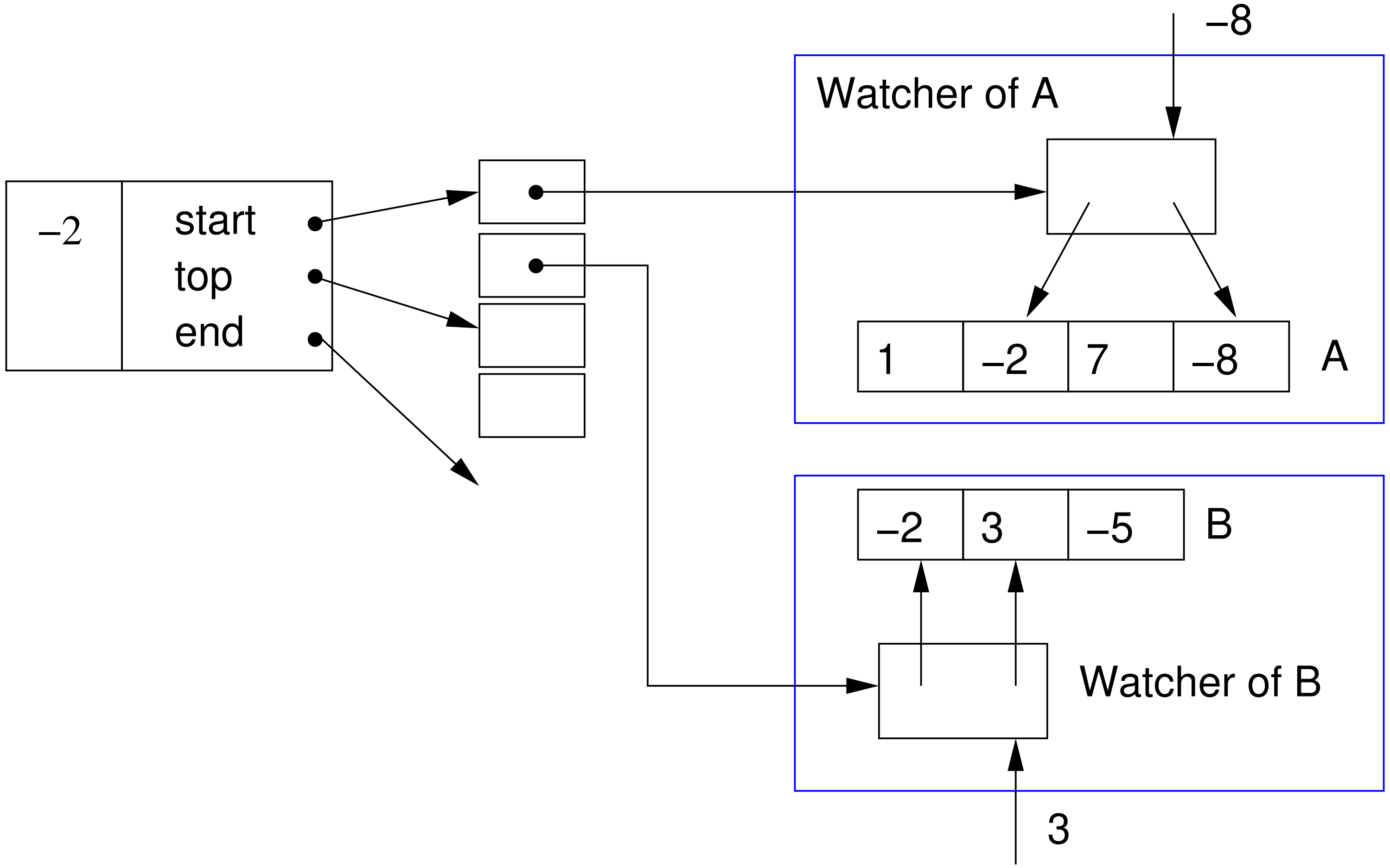
Average Learned Clause Length



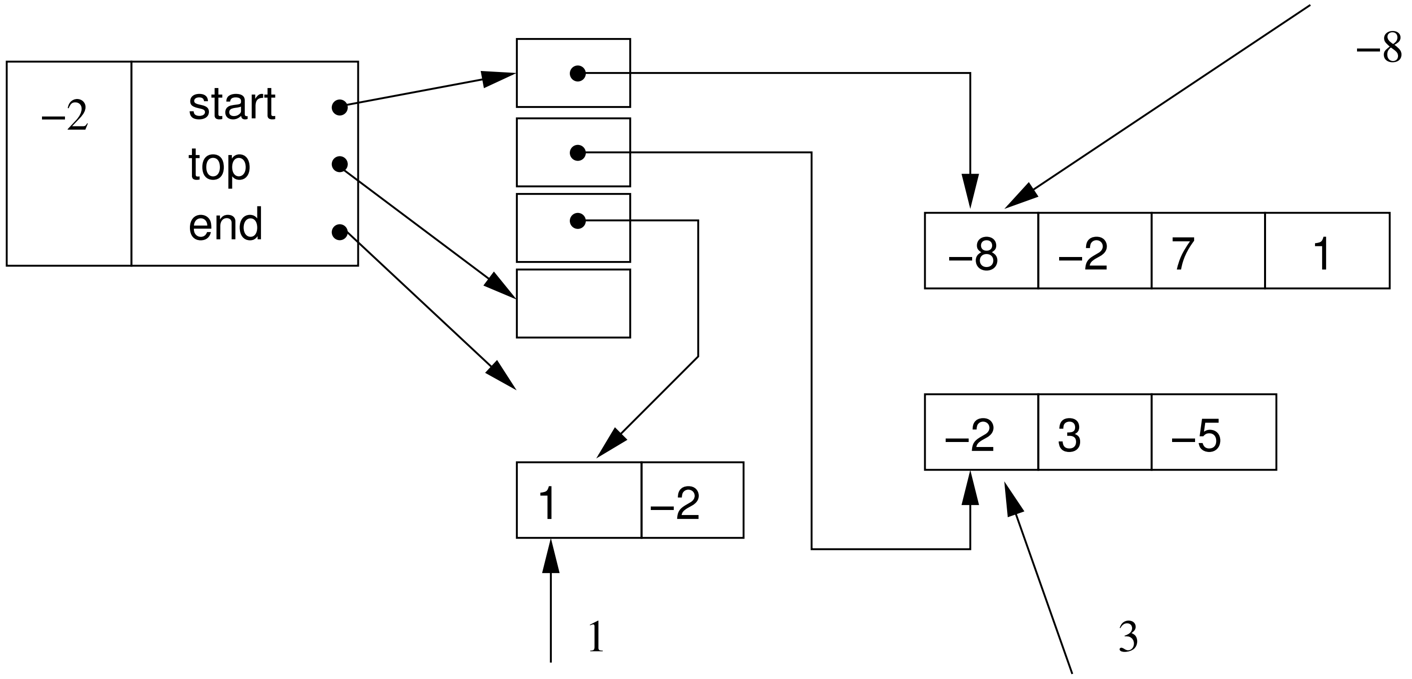
Percentage Visited Clauses With Other Watched Literal True



Limmat / FunEx Occurrence Stacks

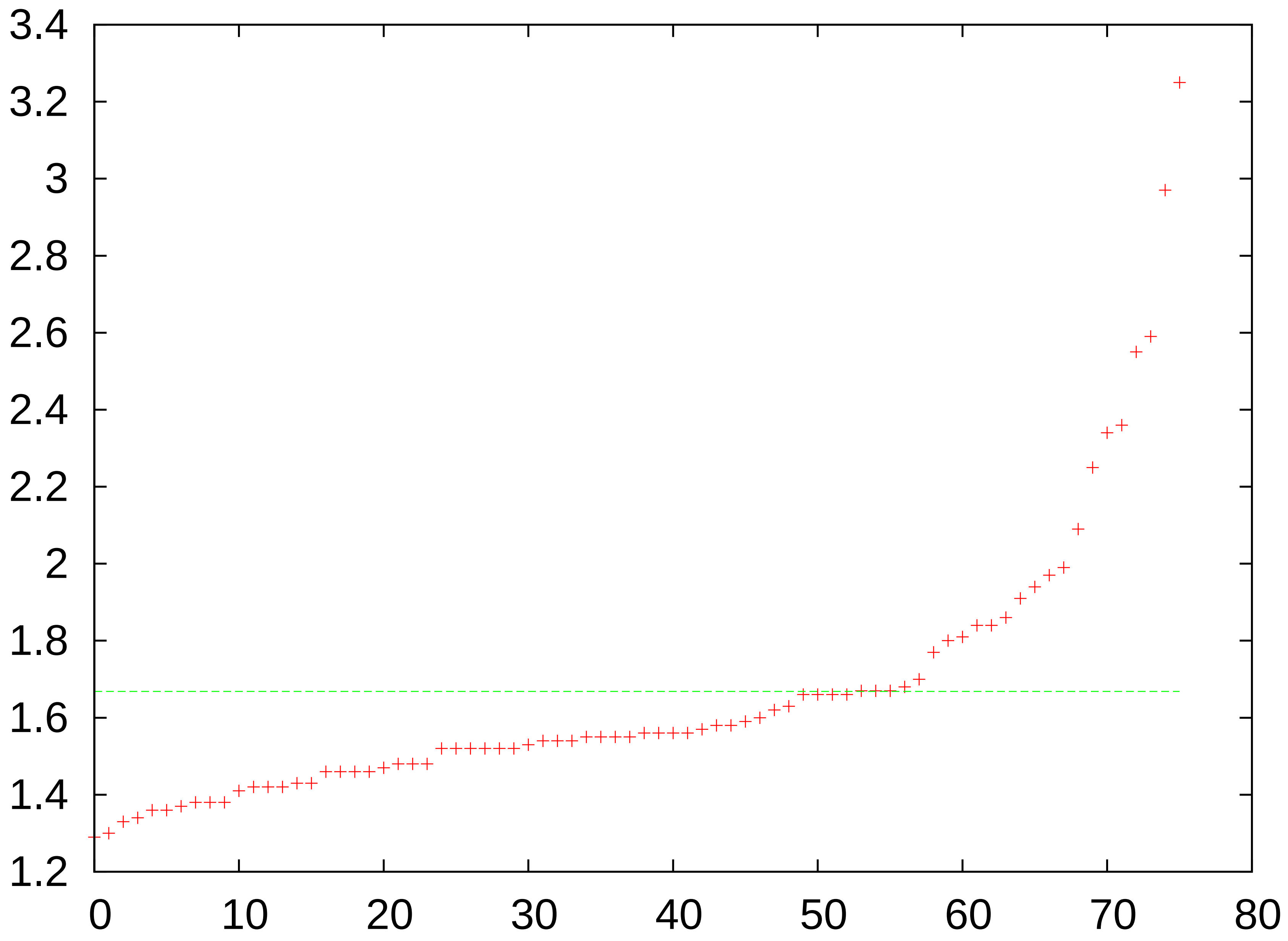


still seems to be best way for *real* sharing of clauses in multi-threaded solvers

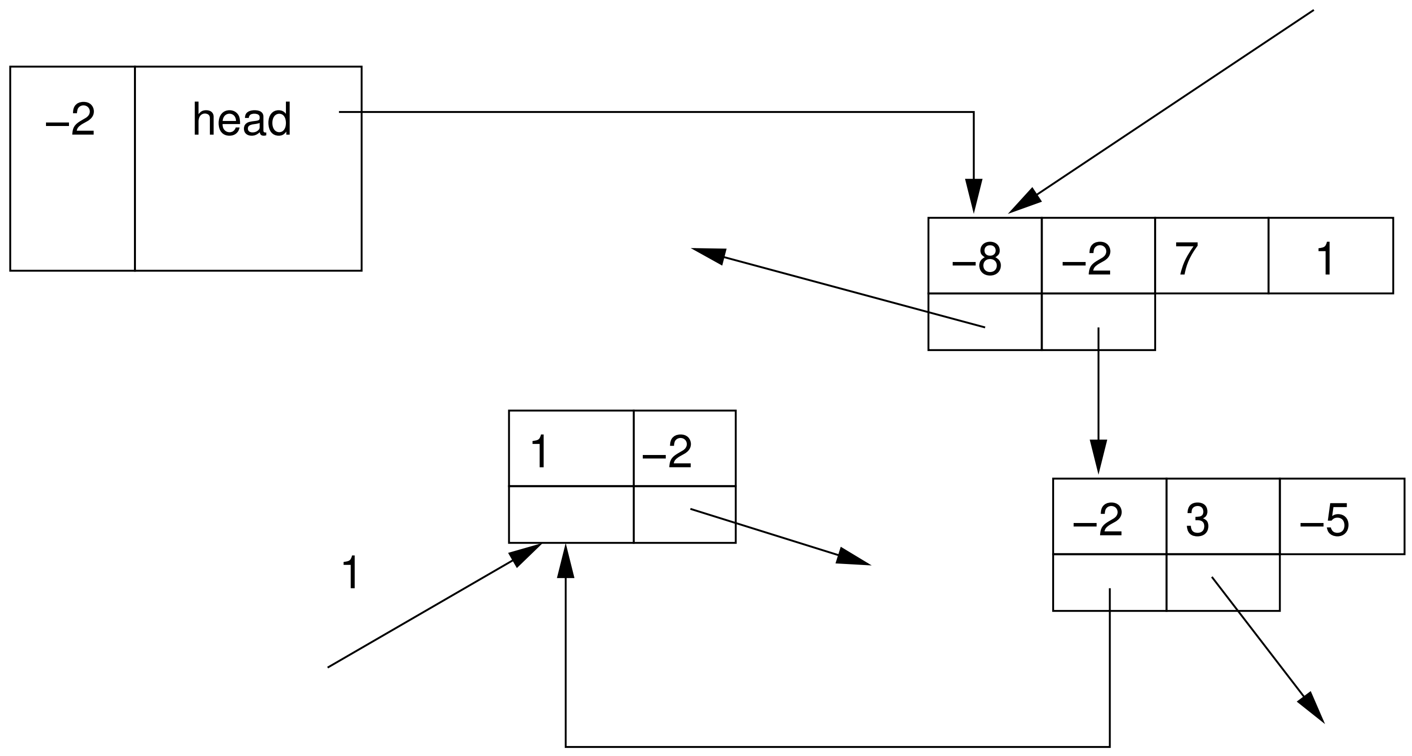


invariant: first two literals are watched

Average Number Literals Traversed Per Visited Clause

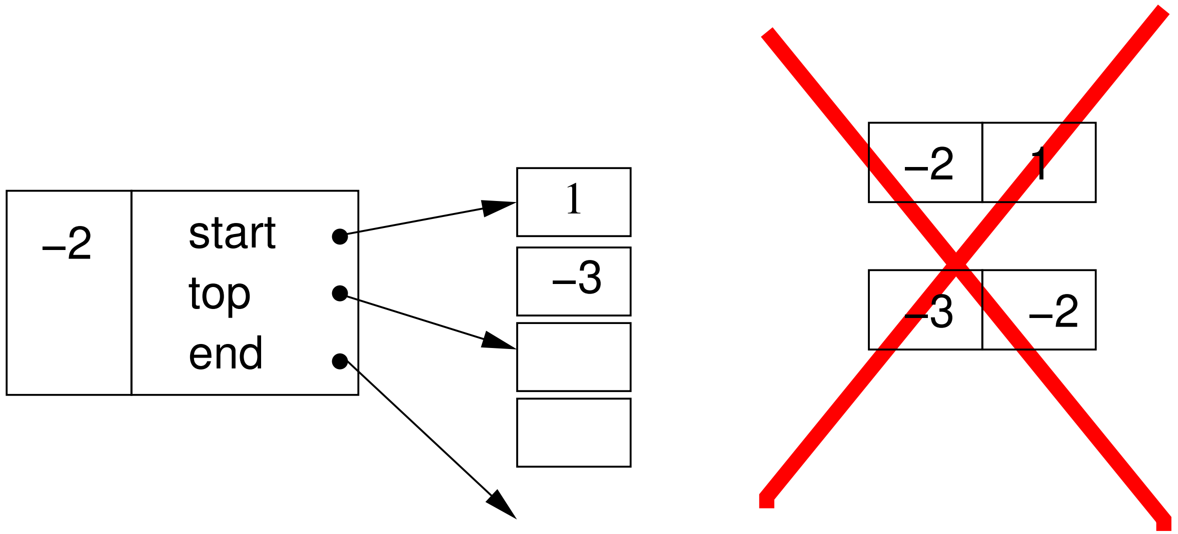


MChaff / PicoSAT Occurrence Lists



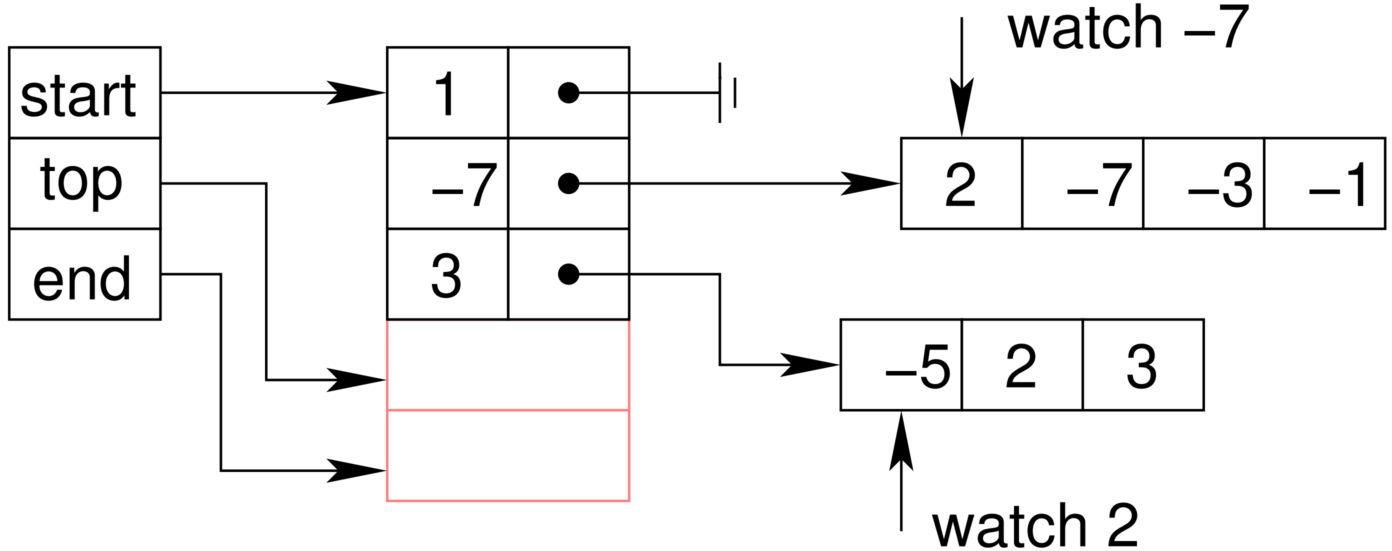
invariant: first two literals are watched

Additional Binary Clause Watcher Stack



Caching Potential Satisfied Literals (Blocking Literals)

ChuHarwoodStuckey'09



observation: often the *other* watched literal satisfies the clause

so cache this literals in watch list to avoid pointer dereference

for binary clause no need to store clause at all

can easily be adjusted for ternary clauses (with full occurrence lists)

LINGELING uses more compact pointer-less variant

Failed Literal Probing

[Freeman'95] [LeBerre'01]

- key technique in look-ahead solvers such as Satz, OKSolver, March
  - failed literal probing at all search nodes
  - used to find the best decision variable and phase
- simple algorithm
  - assume literal l, propagate (BCP), if this results in conflict, add unit clause $\neg l$
  - continue with all literals l until *saturation* (nothing changes)
- quadratic to cubic complexity
  - BCP linear in the size of the formula — 1st linear factor
  - each variable needs to be tried — 2nd linear factor
  - and tried again if some unit has been derived — 3rd linear factor
- optimizations
  - only probe roots (at most quadratic, does not derive all hyper binary resolvents)
  - practically almost linear through tree-based look-ahead — [HeuleJärvisaloBiere'13]

Extensions

- lifting
  - complete case split: literals implied in all cases become units
  - similar to Stålmark's method and Recursive Learning [PradhamKunz'94]
- asymmetric branching
  - assume all but one literal of a clause to be false
  - if BCP leads to conflict remove originally remaining unassigned literal
  - implemented for a long time in MiniSAT but switched off by default
- generalizations:
  - vivification [PietteHamadiSais ECAI'08]
  - distillation [JinSomenzi'05][HanSomenzi DAC'07] probably most general (+ tries)

Other Types of Learning

- similar to look-ahead heuristics: polynomially bounded search
  - may be recursively applied (however, is often too expensive)
- Stålmarck's Method
  - works on triplets (intermediate form of the Tseitin transformation):

    $x = (a \wedge b)$, $y = (c \vee d)$, $z = (e \oplus f)$ etc.
  - generalization of BCP to (in)equalities between variables
  - **test rule** splits on the two values of a variable
- Recursive Learning (Kunz & Pradhan)
  - (originally) works on circuit structure (derives implications)
  - splits on different ways to *justify* a certain variable value
- sweeping (for instance [HeuleBiere'13])

Bounded Variable Elimination (VE)

[DavisPutnam60][Biere SAT’04] [SubbarayanPradhan SAT’04] [EénBiere SAT’05]

- use DP to existentially quantify out variables as in [DavisPutnam60]
- only remove a variable if this does not add (too many) clauses
  - do not count tautological resolvents
  - detect units on-the-fly
- schedule removal attempts with a priority queue [Biere SAT’04] [EénBiere SAT’05]
  - variables ordered by the number of occurrences
- strengthen and remove subsumed clauses (on-the-fly)
  (SATeLite [EénBiere SAT’05] and Quantor [Biere SAT’04])

Fast (Self) Subsumption

- for each (new or strengthened) clause
  - traverse list of clauses of the least occuring literal in the clause
  - check whether traversed clauses are subsumed or
  - strengthen traversed clauses by self-subsumption [EénBiere SAT'05]
  - use Bloom Filters (as in "bit-state hashing"), aka signatures
- check old clauses being subsumed by new clause: backward (self) subsumption
  - new clause (self) subsumes existing clause
  - new clause smaller or equal in size
- check new clause to be subsumed by existing clauses forward (self) subsumption
  - can be made more efficient by one-watcher scheme [Zhang-SAT'05]

Blocked Clause Elimination (BCE)

one clause $C \in F$ with l | *all* clauses in F with $\bar{l}$

fix a CNF F

$a \vee b \vee l$

$\bar{l} \vee \bar{a} \vee c$

$\bar{l} \vee \bar{b} \vee d$

all resolvents of C on l are tautological $\Rightarrow$ C can be removed

Proof assume assignment σ satisfies $F \backslash C$ but not C

can be extended to a satisfying assignment of F by flipping value of l

Blocked Clauses

Kullmann'99

Definition A literal l in a clause C of a CNF F blocks C w.r.t. F if for every clause $C' \in F$ with $\bar{l} \in C'$, the resolvent $(C \setminus \{l\}) \cup (C' \setminus \{\bar{l}\})$ obtained from resolving C and C' on l is a tautology.

Definition [Blocked Clause] A clause is blocked if has a literal that blocks it.

Definition [Blocked Literal] A literal is blocked if it blocks a clause.

Example $(a \vee b) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee c)$

only first clause is not blocked.

second clause contains two blocked literals: a and $\bar{c}$.

literal c in the last clause is blocked.

after removing either $(a \vee \bar{b} \vee \bar{c})$ or $(\bar{a} \vee c)$, the clause $(a \vee b)$ becomes blocked

actually all clauses can be removed

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

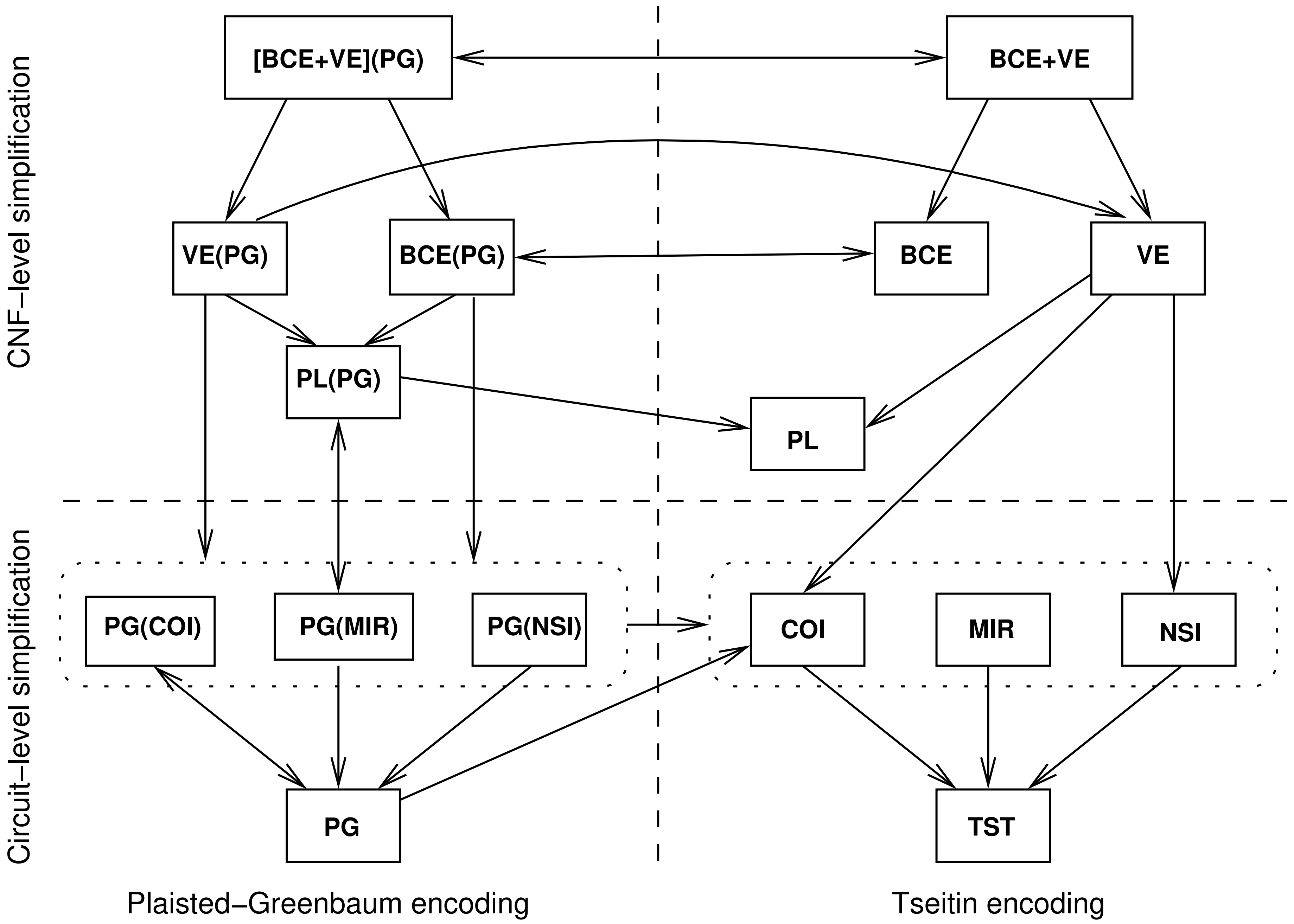
NSI Non-Shared Inputs reduction

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

VE Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination


CNF–level simplification
Circuit–level simplification
[BCE+VE](PG)
BCE+VE
VE(PG)
BCE(PG)
BCE
VE
PL(PG)
PL
PG(COI)
PG(MIR)
PG(NSI)
COI
MIR
NSI
PG
TST
Plaisted–Greenbaum encoding
Tseitin encoding

PrecoSAT [Biere'09], Lingeling [Biere'10], also in CryptoMiniSAT (Mate Soos)

- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple idea to benefit from full preprocessing without penalty
  - "preempt" preprocessors after some time
  - resume preprocessing between restarts
  - limit preprocessing time in relation to search time

Other Inprocessing / Preprocessing Techniques

equivalent literal substitution find strongly connected components in binary implication graph, replace equivalent literals by representatives

boolean ring reasoning extract XORs, then Gaussian elimination etc.

hyper-binary resolution focus on producing binary resolvents

hidden/asymmetric tautology elimination discover redundant clauses through probing

covered clause elimination use covered literals in probing for redundant clauses

unhiding randomized algorithm (one phase linear) for clause removal and strengthening

Benefits of Inprocessing

- allows to use *costly* preprocessors
  - without increasing run-time "much" in the worst-case
  - still useful for benchmarks where these costly techniques help
  - good examples: probing and distillation — even VE can be costly
- additional benefit:
  - makes units / equivalences learned in search available to preprocessing
  - particularly interesting if preprocessing simulates encoding optimizations
- danger of hiding "bad" implementation though ...
- ... and hard(er) to get right!

"Inprocessing Rules" JärvisaloHeuleBiere'12 at IJCAR

Resolution Asymmetric Tautologies (RAT)

"Inprocessing Rules" [JärvisaloHeuleBiere-IJCAR'12]

- justify complex preprocessing algorithms in Lingeling
  - examples are adding blocked clauses or variable elimination
  - interleaved with research (forgetting learned clauses = reduce)
- need more general notion of redundancy criteria
  - simply replace "resolvents are tautological" by "resolvents on l are RUP"

$$(a \vee l) \qquad \text{RAT on } l \quad \text{w.r.t.} \qquad (a \vee c) \wedge (b \vee \bar{c}) \wedge \underbrace{(\bar{l} \vee b)}_{D}$$

  - deletion information is again essential (DRAT)
  - now mandatory in the main track of the last two SAT competitions
  - pretty powerful: can for instance also cover symmetry breaking

Propagation Redundant (PR)

"Short Proofs Without New Variables" [HeuleKieslBiere-CADE'17] best paper

- more general than RAT: short proofs for pigeon hole formulas without new variables
- most general: clause C *redundant* iff exists (partial) witness assignment ω satisfying C with $\boxed{F|\overline{C} \models F|\omega}$
- clause C *propagation redundant* iff exists (partial) witness assignment ω satisfying C with $\boxed{F|\overline{C} \vdash_1 F|\omega}$
- Satisfaction Driven Clause Learning (SDCL) [HeuleKieslSeidlBiere-HVC'17] best paper
  - if **positive reduct** satisfiable learn clause
  - *first* set of automatically generated PR proofs
  - prune paths for which we have other at least as satisfiable paths
- translate PR to DRAT [HeuleBiere-TACAS'18]
  - only one additional variable needed (but in general quadratic)
  - shortest proofs for pigeon hole formulas
- SDCL with **filtered positive reduct** [HeuleKieslBiere-TACAS'19] to appear
  - short clausal proofs for Tseitin and Mutilated Chessboard formulas boards

$F \vdash_1 G$ iff

$F \vdash_1 C$ for all $C \in G$ iff

$F \wedge \neg C \vdash_1 \bot$ for all $C \in G$ iff

every clause in G is unit implied by F

Positive Reduct $\mathrm{p}_\alpha(F)$

Filtered Positive Reduct $\mathrm{f}_\alpha(F)$

Given partial assignment α.

Collect form F all clauses satisfied by α.

Remove unassigned literals but keep assigned literals as is.

Add negation of α as a clause C.

These clauses make up the positive reduct.

Filter the positive reduct by removing clauses D
with unit implied unassigned part, i.e., $F|\alpha \vdash_1 \mathit{unassigned}(\alpha, D)$

If (filtered) positive reduct *satisfied* by ω then ω is a “conditional autarky” under α and C can be added to F.

Satisfaction Driven Clause Learning (SDCL)

SDCL(formula F)

$\alpha := \emptyset$

forever do

 $\alpha :=$ *UnitPropagate*(F, α)

 if α falsifies a clause in F **then**

  $C :=$ *AnalyzeConflict*$()$

  $F := F \wedge C$

  if C is the empty clause $\perp$ **then return** UNSAT

  $\alpha :=$ *BackJump*(C, α)

 else if the pruning predicate $P_\alpha(F)$ is satisfiable **then**

  $C :=$ *AnalyzeWitness*$()$

  $F := F \wedge C$

  $\alpha :=$ *BackJump*(C, α)

 else

  if all variables are assigned **then return** SAT

  $l :=$ *Decide*$()$

  $\alpha := \alpha \cup \{l\}$

Exercise Failed Literal Probing in MiniSAT

- use same version as in MiniSAT hack track of SAT competition 2014:
  http://www.satcompetition.org/2014/description.shtml
  http://fmv.jku.at/ringvorlesung
- basic qubic version: probe all literals until *saturation*
- simple optimizations
  - roots only: probe only roots of the *binary implication graph*
  - quadratic version:
    - time stamp assigned/propagated literals
    - skip probing literals assigned after last probed literal failed
- advanced optimizations
  - inprocessing version
  - decompose binary implication graph into *strongly connected components* and make sure to probe exactly one literal in each root component
  - implement tree-based look-ahead