SAT

ReRiSE'14 Winter School

Reasoning for Rigorous System Enineering

http://fmv.jku.at/rerise14

Johannes Kepler University

Linz, Austria

Armin Biere
Institute for Formal Models and Verification
http://fmv.jku.at

optimization of if-then-else chains

original C code

```
if(!a && !b) h();
else if(!a) g();
else f();
```

$\Downarrow$

```
if(!a) {
  if(!b) h();
  else g();
} else f();
```

$\Rightarrow$

optimized C code

```
if(a) f();
else if(b) g();
else h();
```

$\Uparrow$

```
if(a) f();
else {
  if(!b) h();
  else g(); }
```

How to check that these two versions are equivalent?

1. represent procedures as *independent* boolean variables

$$\textit{original} := \quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h \quad \textbf{else if } \neg a \textbf{ then } g \quad \textbf{else } f$$

$$\textit{optimized} := \quad \textbf{if } a \textbf{ then } f \quad \textbf{else if } b \textbf{ then } g \quad \textbf{else } h$$

2. compile if-then-else chains into boolean formulae

$$\text{compile}(\textbf{if } x \textbf{ then } y \textbf{ else } z) \quad \equiv \quad (x \wedge y) \vee (\neg x \wedge z)$$

3. check equivalence of boolean formulae

$$\text{compile}(\textit{original}) \quad \Leftrightarrow \quad \text{compile}(\textit{optimized})$$

$$
\begin{aligned}
\textcolor{magenta}{\mathit{original}} \quad &\equiv \quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv \quad (\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv \quad (\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \;\vee\; a \wedge f)
\end{aligned}
$$

$$
\begin{aligned}
\textcolor{magenta}{\mathit{optimized}} \quad &\equiv \quad \textbf{if } a \textbf{ then } f \textbf{ else } \textbf{ if } b \textbf{ then } g \textbf{ else } h \\
&\equiv \quad a \wedge f \;\vee\; \neg a \wedge \textbf{ if } b \textbf{ then } g \textbf{ else } h \\
&\equiv \quad a \wedge f \;\vee\; \neg a \wedge (b \wedge g \;\vee\; \neg b \wedge h)
\end{aligned}
$$

$$
(\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \;\vee\; a \wedge f) \quad \Leftrightarrow \quad a \wedge f \;\vee\; \neg a \wedge (b \wedge g \;\vee\; \neg b \wedge h)
$$

Reformulate it as a satisfiability (SAT) problem:

Is there an assignment to a, b, f, g, h,
which results in different evaluations of original and optimized?

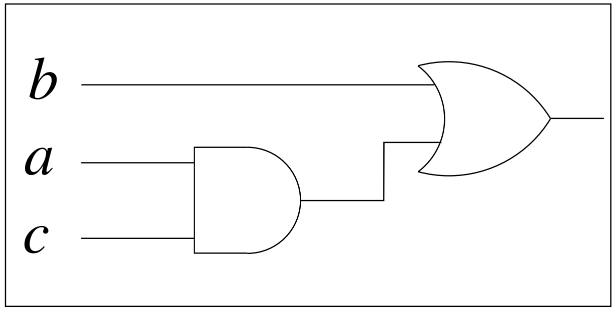
or equivalently:

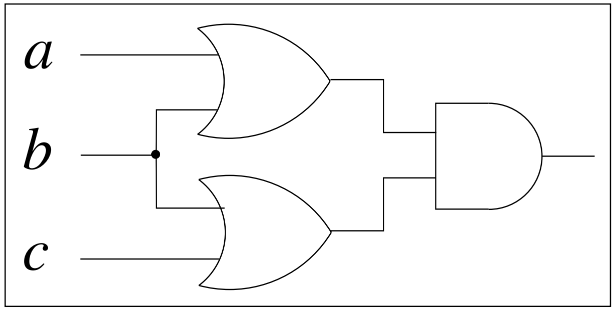
Is the boolean formula $\mathrm{compile}(\mathit{original}) \not\leftrightarrow \mathrm{compile}(\mathit{optimized})$ satisfiable?

such an assignment would provide an easy to understand counterexample

Note: by concentrating on counterexamples we moved from Co-NP to NP
(this is just a theoretical note and not really important for applications)

SAT Example: Circuit Equivalence Checking





$b \vee a \wedge c$

$(a \vee b) \wedge (b \vee c)$

equivalent?

$$b \vee a \wedge c \quad \Leftrightarrow \quad (a \vee b) \wedge (b \vee c)$$

SAT (Satisfiability) the classical NP complete Problem:

Given a propositional formula f over n propositional variables $V = \{x, y, \ldots\}$.

Is there are an assignment $\sigma : V \to \{0, 1\}$ with $\sigma(f) = 1$?

SAT belongs to NP

There is a *non-deterministic* Touring-machine deciding SAT in polynomial time:

guess the assignment σ (linear in n), calculate $\sigma(f)$ (linear in $|f|$)

Note: on a *real* (deterministic) computer this would still require 2^n time

SAT is complete for NP (see complexity / theory class)

Implications for us:
general SAT algorithms are probably exponential in time (unless NP = P)

Definition

a formula in Conjunctive Normal Form (CNF) is a conjunction of clauses

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

each clause C is a disjunction of literals

$$C = L_1 \vee \ldots \vee L_m$$

and each literal is either a plain variable x or a negated variable $\overline{x}$.

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

Note 1: two notions for negation: in $\overline{x}$ and $\neg$ as in $\neg x$ for denoting negation.

Note 2: the original SAT problem is actually formulated for CNF

Note 3: SAT solvers mostly also expect CNF as input

Assumption: we only have conjunction, disjunction and negation as operators.

a formula is in Negation Normal Form (NNF),
if negations only occur in front of variables

$\Rightarrow$ all *internal* nodes in the formula tree are either ANDs or ORs

linear algorithms for generating NNF from an arbitrary formula

often NNF generations includes elimination of other non-monotonic operators:

NNF of $f \leftrightarrow g$ is NNF of $f \wedge g \vee \overline{f} \wedge \overline{g}$

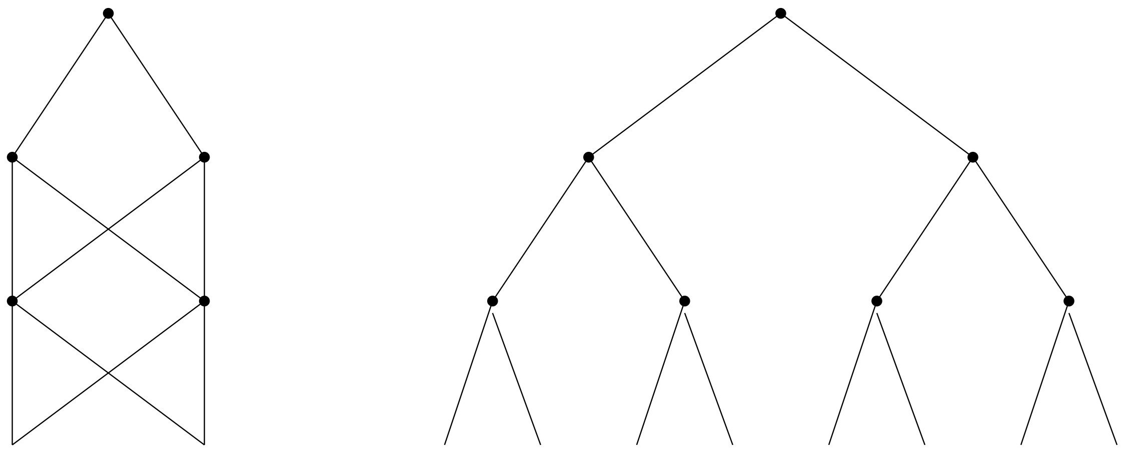
in this case the result can be exponentially larger (see parity example later).

```
Formula
formula2nnf (Formula f, Boole sign)
{
  if (is_variable (f))
    return sign ? new_not_node (f) : f;
  if (op (f) == AND || op (f) == OR)
    {
      l = formula2nnf (left_child (f), sign);
      r = formula2nnf (right_child (f), sign);
      flipped_op = (op (f) == AND) ? OR : AND;
      return new_node (sign ? flipped_op : op (f), l, r);
    }
  else
    {
      assert (op (f) == NOT);
      return formula2nnf (child (f), !sign);
    }
}
```

```
Formula
formula2cnf_aux (Formula f)
{
  if (is_cnf (f))
    return f;
  if (op (f) == AND)
    {
      l = formula2cnf_aux (left_child (f));
      r = formula2cnf_aux (right_child (f));
      return new_node (AND, l, r);
    }
  else
    {
      assert (op (f) == OR);
      l = formula2cnf_aux (left_child (f));
      r = formula2cnf_aux (right_child (f));
      return merge_cnf (l, r);
    }
}
```

```
Formula
formula2cnf (Formula f)
{
  return formula2cnf_aux (formula2nnf (f, 0));
}


Formula
merge_cnf (Formula f, Formula g)
{
  res = new_constant_node (TRUE);
  for (c = first_clause (f); c; c = next_clause (f, c))
    for (d = first_clause (g); d; d = next_clause (g, d))
      res = new_node (AND, res, new_node (OR, c, d));
  return res;
}
```

Why are Sharing / Circuits / DAGs important?



DAG may be exponentially more succinct than expanded Tree

Examples: adder circuit, parity, mutual exclusion

```
Boole
parity (Boole a, Boole b, Boole c, Boole d, Boole e,
        Boole f, Boole g, Boole h, Boole i, Boole j)
{
  tmp0 = b ? !a : a;
  tmp1 = c ? !tmp0 : tmp0;
  tmp2 = d ? !tmp1 : tmp1;
  tmp3 = e ? !tmp2 : tmp2;
  tmp4 = f ? !tmp3 : tmp3;
  tmp5 = g ? !tmp4 : tmp4;
  tmp6 = h ? !tmp5 : tmp5;
  tmp7 = i ? !tmp6 : tmp6;
  return j ? !tmp7 : tmp7;
}
```

Eliminiate the `tmp...` variables through substitution.

What is the size of the DAG vs the Tree representation?

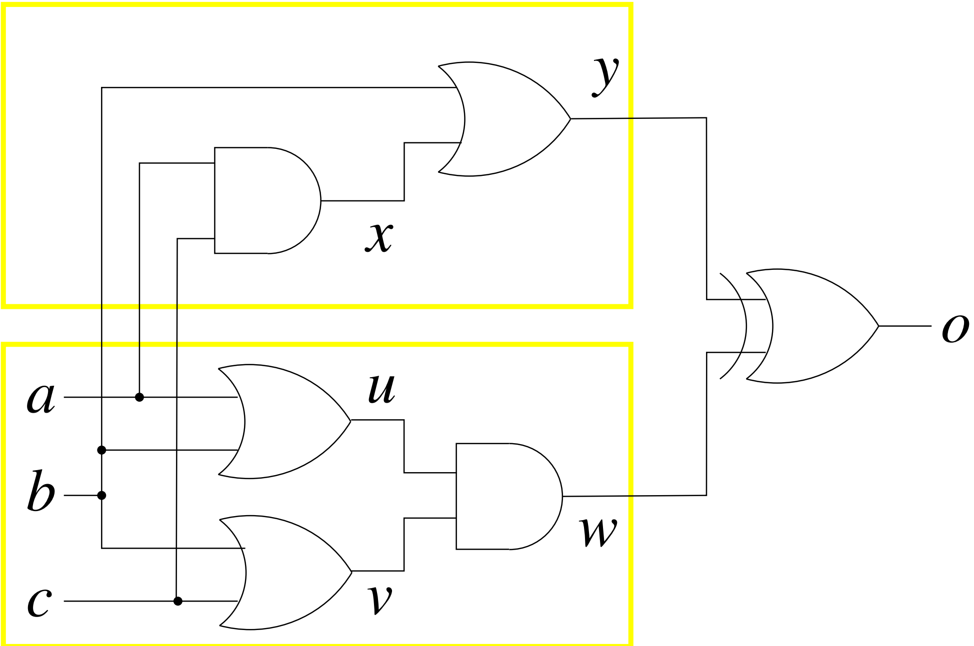
- through caching of results in algorithms operating on formulas
  (examples: substitution algorithm, generation of NNF for non-monotonic ops)

- when modeling a system: variables are introduced for subformulae
  (then these variables are used multiple times in the toplevel formula)

- structural hashing: detects structural identical subformulae
  (see Signed And Graphs later)

- equivalence extraction: e.g. BDD sweeping, Stålmarcks Method
  (we will look at both techniques in more detail later)

CNF



$$
\begin{aligned}
&o \wedge \\
(x &\leftrightarrow a \wedge c) \wedge \\
(y &\leftrightarrow b \vee x) \wedge \\
(u &\leftrightarrow a \vee b) \wedge \\
(v &\leftrightarrow b \vee c) \wedge \\
(w &\leftrightarrow u \wedge v) \wedge \\
(o &\leftrightarrow y \oplus w)
\end{aligned}
$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$

Algorithmic Description of Tseitin Transformation

1. for each non input circuit signal s generate a new variable x_s

2. for each gate produce complete input / output constraints as clauses

3. collect all constraints in a big conjunction

the transformation is *satisfiability equivalent*:
the result is satisfiable iff and only the original formula is satisfiable

not equivalent in the classical sense to original formula: it has new variables

extract satisfying assignment for original formula, from one of the result
(just project satisfying assignment onto the original variables)

Tseitin Transformation: Input / Output Constraints

$$
\begin{array}{rrcl}
\text{Negation:} & x \leftrightarrow \overline{y} & \Leftrightarrow & (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \\
& & \Leftrightarrow & (\overline{x} \vee \overline{y}) \wedge (y \vee x) \\
\\
\text{Disjunction:} & x \leftrightarrow (y \vee z) & \Leftrightarrow & (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \\
& & \Leftrightarrow & (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z) \\
\\
\text{Conjunction:} & x \leftrightarrow (y \wedge z) & \Leftrightarrow & (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\
& & \Leftrightarrow & (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\
& & \Leftrightarrow & (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x) \\
\\
\text{Equivalence:} & x \leftrightarrow (y \leftrightarrow z) & \Leftrightarrow & (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
& & \Leftrightarrow & (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y))) \wedge ((y \leftrightarrow z) \rightarrow x) \\
& & \Leftrightarrow & (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
& & \Leftrightarrow & (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\
& & \Leftrightarrow & (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x) \\
& & \Leftrightarrow & (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x) \\
& & \Leftrightarrow & (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x)
\end{array}
$$

- goal is smaller CNF (less variables, less clauses)
- extract multi argument operands (removes variables for intermediate nodes)
- half of AND, OR node constraints can be removed for *unnegated* nodes

  a node occurs negated if it has an ancestor which is a negation

  half of the constraints determine parent assignment from child assignment

  those are unnecessary if node is not used negated

  [PlaistedGreenbaum'86] and then [ChambersManoliosVroon'09]
- structural circuit optimizations like in the ABC tool from Berkeley
- however might be simulated on CNF level [JärvisaloBiereHeule-TACAS'10]
- compact technology mapping based encoding [EénMishchenkoSörensson'07]

- encoding directly into CNF is hard, so we use intermediate levels:

  1. application level
  2. bit-precise semantics world-level operations: bit-vector theory
  3. bit-level representations such as AIGs or vectors of AIGs
  4. CNF

- encoding application level formulas into word-level: as generating machine code

- word-level to bit-level: bit-blasting similar to hardware synthesis

- encoding “logical” constraints is another story

Bit-Blasting of 4-Bit Addition

addition of 4-bit numbers x, y with result s also 4-bit: $s = x + y$

$$[s_3, s_2, s_1, s_0]_4 \;=\; [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4$$

$$\begin{aligned}
[s_3, \cdot\,]_2 &= \mathsf{FullAdder}(x_3, y_3, c_2) \\
[s_2, c_2]_2 &= \mathsf{FullAdder}(x_2, y_2, c_1) \\
[s_1, c_1]_2 &= \mathsf{FullAdder}(x_1, y_1, c_0) \\
[s_0, c_0]_2 &= \mathsf{FullAdder}(x_0, y_0, \mathit{false})
\end{aligned}$$

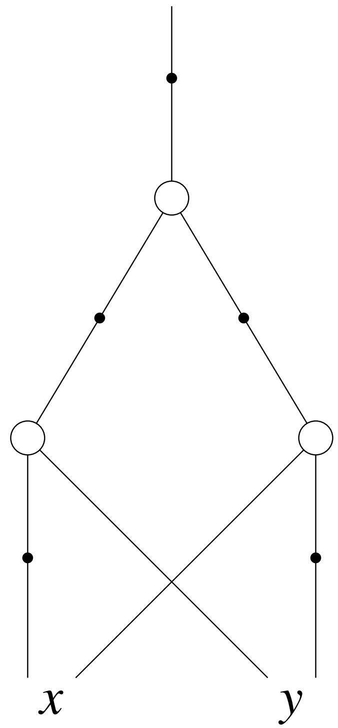
where

$$\begin{aligned}
[\,s\,,\,o\,]_2 &= \mathsf{FullAdder}(x, y, i) \quad \text{with} \\
s &= x \text{ xor } y \text{ xor } i \\
o &= (x \wedge y) \vee (x \wedge i) \vee (y \wedge i) \;=\; ((x + y + i) \geq 2)
\end{aligned}$$

And-Inverter-Graphs (AIG)

- widely adopted bit-level intermediate representation
  - see for instance our AIGER format http://fmv.jku.at/aiger
  - used in Hardware Model Checking Competition (HWMCC)
  - also used in the *structural track* in SAT competitions
  - many companies use similar techniques
- basic logical operators: *conjunction* and *negation*
- DAGs: nodes are conjunctions, negation/sign as *edge attribute*

  bit stuffing: signs are compactly stored as LSB in pointer
- automatic sharing of isomorphic graphs, constant time (peep hole) simplifications
- *or even* SAT sweeping, full reduction, etc ... see ABC system from Berkeley



negation/sign are edge attributes

not part of node

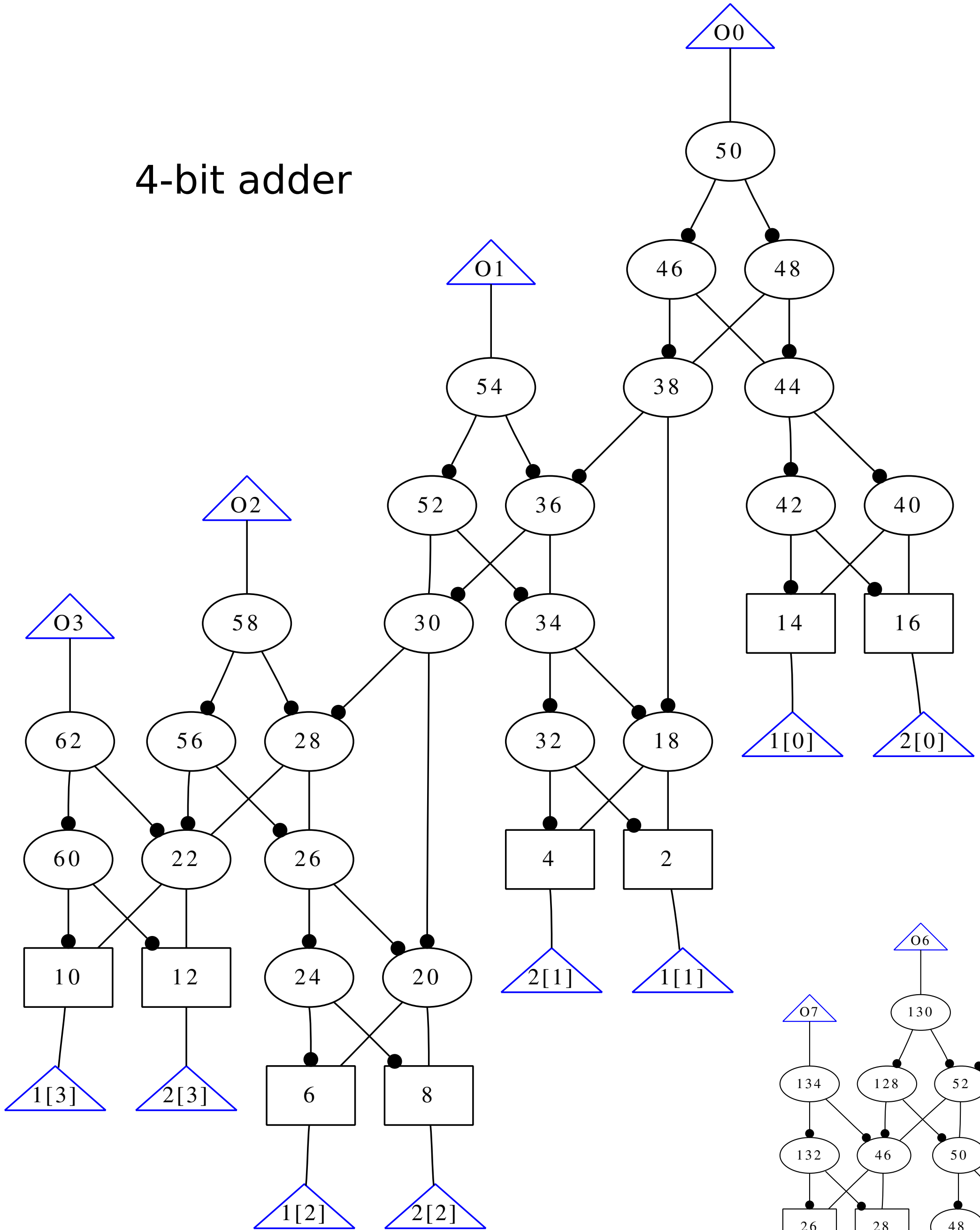
$$x \text{ xor } y \;\equiv\; (\overline{x} \wedge y) \vee (x \wedge \overline{y}) \;\equiv\; \overline{\overline{(\overline{x} \wedge y)} \wedge \overline{(x \wedge \overline{y})}}$$

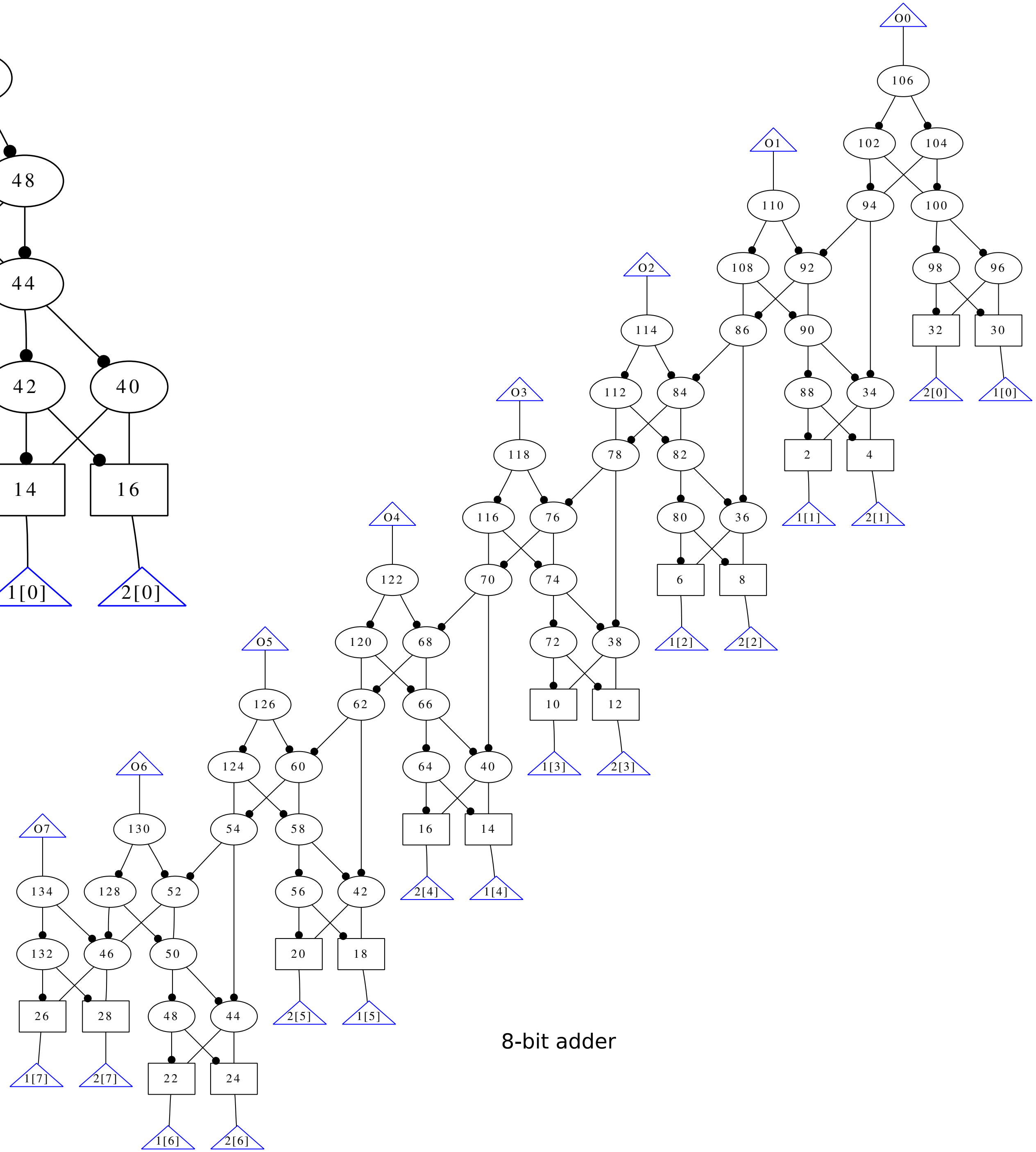
```c
typedef struct AIG AIG;

struct AIG
{
  enum Tag tag;                    /* AND, VAR */
  void *data[2];
  int mark, level;                 /* traversal */
  AIG *next;                       /* hash collision chain */
};

#define sign_aig(aig) (1 & (unsigned) aig)
#define not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
#define strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
#define false_aig ((AIG*) 0)
#define true_aig ((AIG*) 1)
```

assumption for correctness:

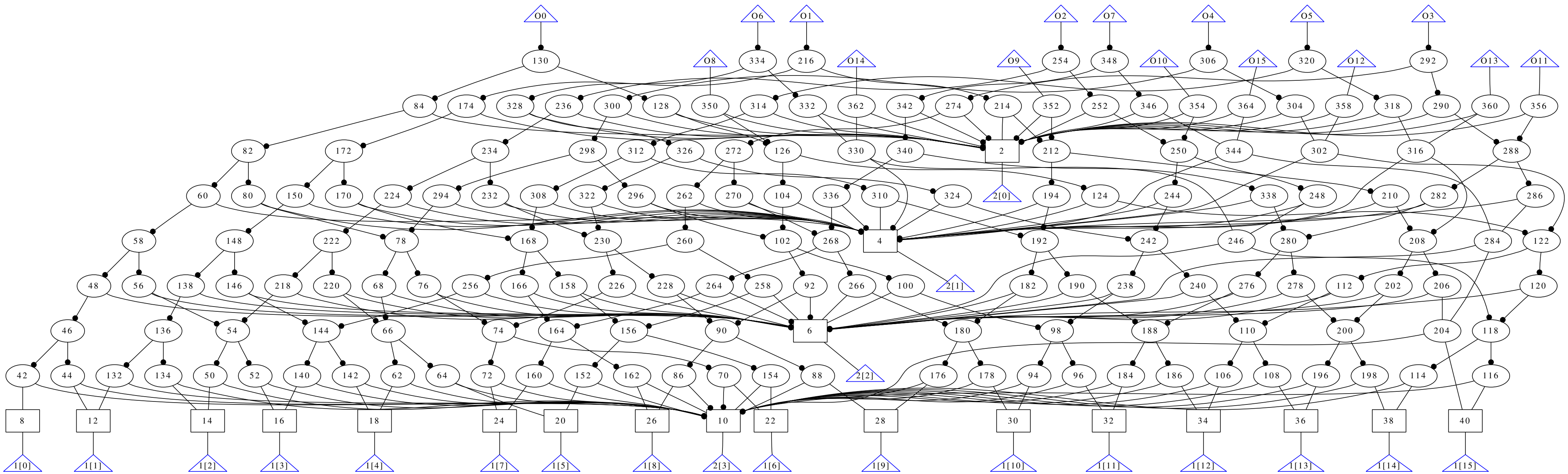
`sizeof(unsigned) == sizeof(void*)`

4-bit adder

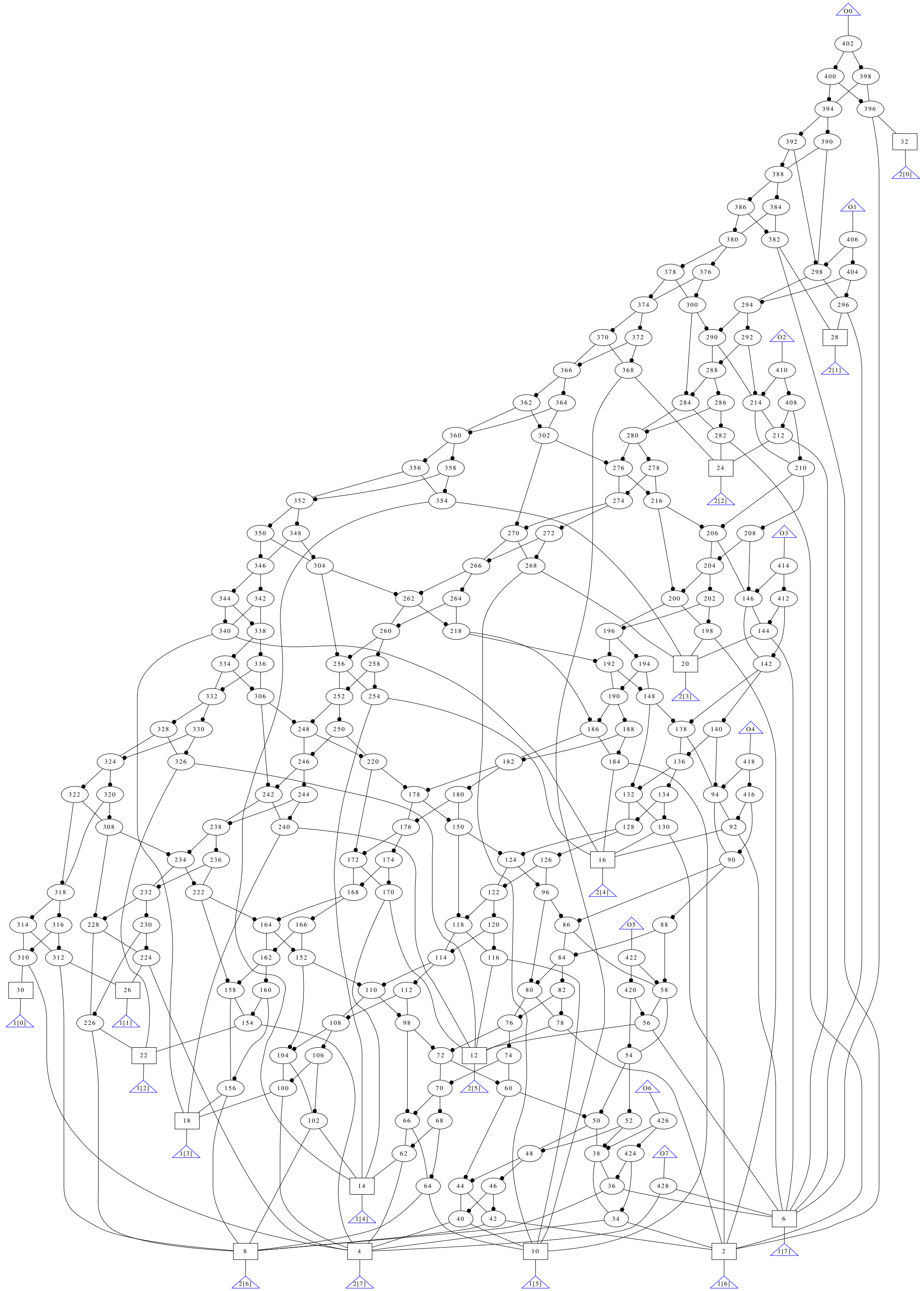




8-bit adder



bit-vector of length 16 shifted by bit-vector of length 4



- Tseitin's construction suitable for most kinds of "model constraints"
  - assuming simple operational semantics: encode an interpreter
  - small domains: *one-hot encoding* large domains: *binary encoding*
- harder to encode *properties* or additional *constraints*
  - temporal logic / fix-points
  - environment constraints
- example for fix-points / recursive equations: $x = (a \vee y), \quad y = (b \vee x)$
  - has unique *least* fix-point $x = y = (a \vee b)$
  - and unique *largest* fix-point $x = y = \mathit{true}$ but unfortunately
  - only largest fix-point can be (directly) encoded in SAT otherwise need ASP

- given a set of literals $\{l_1, \ldots l_n\}$
  - constraint the *number* of literals assigned to true
  - $|\{l_1, \ldots, l_n\}| \geq k$ or $|\{l_1, \ldots, l_n\}| \leq k$ or $|\{l_1, \ldots, l_n\}| = k$
- multiple encodings of cardinality constraints
  - naïve encoding exponential: *at-most-two* quadratic, *at-most-three* cubic, etc.
  - quadratic $O(k \cdot n)$ encoding goes back to Shannon
  - linear $O(n)$ parallel counter encoding [Sinz'05]
  - for an $O(n \cdot \log n)$ encoding see Prestwich's chapter in our Handbook of SAT
- generalization *Pseudo-Boolean* constraints (PB), e.g. $2 \cdot \overline{a} + \overline{b} + c + \overline{d} + 2 \cdot e \geq 3$
  actually used to handle MaxSAT in SAT4J for configuration in Eclipse

$$2 \leq |\{l_1, \ldots, l_9\}| \leq 3$$

$$
\begin{array}{cccccccccccccccccccc}
l_1 & --- & l_2 & --- & l_3 & --- & l_4 & --- & l_5 & --- & l_6 & --- & l_7 & --- & l_8 & --- & l_9 & --- & 0 \\
| & & | & & | & & | & & | & & | & & | & & | & & | & & \\
l_2 & --- & l_3 & --- & l_4 & --- & l_5 & --- & l_6 & --- & l_7 & --- & l_8 & --- & l_9 & --- & 0 & & \\
| & & | & & | & & | & & | & & | & & | & & | & & & & \\
l_3 & --- & l_4 & --- & l_5 & --- & l_6 & --- & l_7 & --- & l_8 & --- & l_9 & --- & 1 & & & & \\
| & & | & & | & & | & & | & & | & & | & & & & & & \\
l_4 & --- & l_5 & --- & l_6 & --- & l_7 & --- & l_8 & --- & l_9 & --- & 1 & & & & & & \\
| & & | & & | & & | & & | & & | & & & & & & & & \\
0 & & 0 & & 0 & & 0 & & 0 & & 0 & & & & & & & &
\end{array}
$$

“then” edge downward, “else” edge to the right

- dates back to the 50ies:

  original version is *resolution based* (less successful)

- **idea:** case analysis (try $x = 0, 1$ in turn and recurse)

- most successful SAT solvers

  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)

- basis for first (less successful) resolution based DP
- can be extended to first order logic
- helps to explain learning

Resolution Rule

$$\frac{C \cup \{v\} \qquad D \cup \{\neg v\}}{C \cup D} \qquad \{v, \neg v\} \cap C = \{v, \neg v\} \cap D = \emptyset$$

Read: resolving the clause $C \cup \{v\}$ with the clause $D \cup \{\neg v\}$, both above the line, on the variable v, results in the clause $D \cup C$ below the line.

Usage of such rules: if you can derive what is above the line (premise) then you are allowed to deduce what is below the line (conclusion).

Theorem. (premise satisfiable $\Rightarrow$ conclusion satisfiable)

$$\sigma(C \cup \{v\}) = \sigma(D \cup \{\neg v\}) = 1 \quad \Rightarrow \quad \sigma(C \cup D) = 1$$

Proof.

let $c \in C$, $d \in D$ with $(\sigma(c) = 1$ or $\sigma(v) = 1)$ and $(\sigma(d) = 1$ or $\sigma(\neg v) = 1)$

if $\sigma(c) = 1$ or $\sigma(d) = 1$ conclusion follows immediately

otherwise $\sigma(v) = \sigma(\neg v) = 1 \quad \Rightarrow$ contradiction

q.e.d.

Theorem. (conclusion satisfiable $\Rightarrow$ premise satisfiable)

$$\sigma(C \cup D) = 1 \quad \Rightarrow \quad \exists \sigma' \quad \text{with} \quad \sigma'(C \cup \{v\}) = \sigma'(D \cup \{\neg v\}) = 1$$

Proof.

with out loss of generality pick $c \in C$ with $\sigma(c) = 1$

define $\quad \sigma'(x) = \begin{cases} 0 & \text{if } x = v \\ \sigma(x) & \text{else} \end{cases}$

since v and $\neg v$ do not occur in C, we still have $\sigma'(C) = 1$ and thus $\sigma'(C \cup \{v\}) = 1$

by definition $\sigma'(\neg v) = 1$ and thus $\sigma'(D \cup \{\neg v\}) = 1$

q.e.d.

Idea: use resolution to *existentially* quantify out variables

1. if empty clause found then terminate with result **unsatisfiable**

2. find variables which only occur in one phase (only positive or negative)

3. remove all clauses in which these variables occur

4. if no clause left then terminate with result **satisfiable**

5. choose x as one of the remaining variables with occurrences in both phases

6. add results of all possible resolutions on this variable

7. remove all trivial clauses and all clauses in which x occurs

8. continue with **1.**

check whether XOR is weaker than OR, i.e. validity of:

$$a \vee b \;\rightarrow\; (a \oplus b)$$

which is equivalent to unsatisfiability of the negation:

$$(a \vee b) \;\wedge\; \neg(a \oplus b)$$

since negation of XOR is XNOR (equivalence):

$$(a \vee b) \;\wedge\; (a \leftrightarrow b)$$

we end up checking the following CNF for satisfiability:

$$(a \vee b) \;\wedge\; (\neg a \vee b) \wedge (a \vee \neg b)$$

Example for Resolution DP cont.

$$(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b)$$

initially we can skip **1.** - **4.** of the algorithm and choose $x = b$ in **5.**

in **6.** we resolve $(\neg a \vee b)$ with $(a \vee \neg b)$ and $(a \vee b)$ with $(a \vee \neg b)$ both on b

and add the results $(a \vee \neg a)$ and $(a \vee a)$:

$$(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (a \vee \neg a) \wedge (a \vee a)$$

the trivial clause $(a \vee \neg a)$ and clauses with ocurrences of b are removed:

$$(a \vee a)$$

in **2.** we find a to occur only positive and in **3.** the remaining clause is removed

the test in **4.** succeeds and the CNF turns out to be **satisfiable**

(thus the original formula is invalid – not a tautology)

Proof. in three steps:

(A) show that termination criteria are correct

(B) each transformation preserves satisfiability

(C) each transformation preserves unsatisfiability

Ad (A):

an empty clause is an empty disjunction, which is unsatisfiable

if literals occur only in one phase assign those to $1 \Rightarrow$ all clauses satisfied

CNF transformations preserve satisfiability:

removing a clause does not change satisfiability

thus only adding clauses could potentially not preserve satisfiability

the only clauses added are the results of resolution

correctness of resolution rule shows:

if the original CNF is satisfiable, then the added clause are satisfiable

(even with the same satisfying assignment)

CNF transformations preserve unsatisfiability:

adding a clause does not change unsatisfiability

thus only removing clauses could potentially not preserve unsatisfiability

trivial clauses $(v \vee \neg v \vee \ldots)$ are always valid and can be removed

let f be the CNF after removing all trivial clauses (in step **7.**)

let g be the CNF after removing all clauses in which x occurs (after step **7.**)

we need to show (f unsat $\Rightarrow$ g unsat), or equivalently (g sat $\Rightarrow$ f sat)

the latter can be proven as the completeness proof for the resolution rule

(see next slide)

If we interpret $\cup$ as disjunction and clauses as formulae, then

$$(C_1 \vee x) \wedge \ldots \wedge (C_k \vee x) \quad \wedge \quad (D_1 \vee \neg x) \wedge \ldots \wedge (D_l \vee \neg x)$$

is, via distributivity law, equivalent to

$$(\underbrace{(C_1 \wedge \ldots \wedge C_k)}_{C} \vee x) \quad \wedge \quad (\underbrace{(D_1 \wedge \ldots \wedge D_l)}_{D} \vee \neg x)$$

and the same proof applies as for the completeness of the resolution rule.

Note: just using the completeness of the resolution rule alone does not work, since those σ' derived for multiple resolutions are formally allowed to assign different values for the resolution variable.

- if variables have many occurences, then many resolutions are necessary
- in the worst x and $\neg x$ occur in half of the clauses ...
- ... then the number of clauses increases quadratically
- clauses become longer and longer
- unfortunately in real world examples the CNF explodes

  (we might latter see how BDDs can be used to overcome some of these problems)
- How to obtain the satisfying assignment efficiently (counter example)?

Second version of DP

- resolution based version often called DP, second version DPLL
  (DP after [DavisPutnam60] and DPLL after [DavisLogemannLoveland62])

- it eliminates variables through case analysis: time vs space

- only *unit resolution* used (also called *boolean constraint propagation*)

- case analysis is on-the-fly:

  cases are not elaborated in a predefined fixed order, but . . .

  . . . only remaining crucial cases have to be considered

- allows sophisticated optimizations

a *unit clause* is a clause with a single literal

in CNF a unit clause forces its literal to be assigned to 1

unit resolution is an application of resolution, where one clause is a unit clause

also called *boolean constraint propagation*

Unit-Resolution Rule

$$\frac{C \cup \{\neg l\} \qquad \{l\}}{C} \qquad \{l, \neg l\} \cap C = \emptyset$$

here we identify $\neg\neg v$ with v for all variables v.

Unit-Resolution Example

check whether XNOR is weaker than AND, i.e. validity of:

$$a \wedge b \;\rightarrow\; (a \leftrightarrow b)$$

which is equivalent to unsatisfiability of the CNF (exercise)

$$a \wedge b \;\wedge\; (a \vee b) \wedge (\neg a \vee \neg b)$$

adding clause obtained from unit resolution on a results in

$$a \wedge b \;\wedge\; (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg b)$$

removing clauses containing a or $\neg a$

$$b \;\wedge\; (\neg b)$$

unit resolution on b results in an empty clause and we conclude unsatisfiability.

- if unit resolution produces a unit, e.g. resolving $(a \vee \neg b)$ with b produces a, continue unit resolution with this new unit

- often this repeated application of unit resolution is also called unit resolution

- unit resolution + removal of subsumed clauses never increases size of CNF

$$C \text{ subsumes } D \quad :\Leftrightarrow \quad C \subseteq D$$

  a unit(-clause) l subsumes all clauses in which l occurs in the same phase

- *boolean constraint propagation* (BCP): given a unit l, remove all clauses in which l occurs in the same phase, and remove all literals $\neg l$ in clauses, where it occurs in the opposite phase (the latter is unit resolution)

1. apply repeated unit resolution and removal of all subsumed clauses (BCP)

2. if empty clause found then return **unsatisfiable**

3. find variables which only occur in one phase (only positive or negative)

4. remove all clauses in which these variables occur (pure literal rule)

5. if no clause left then return **satisfiable**

6. choose x as one of the remaining variables with occurrences in both phases

7. recursively call DPLL on current CNF with the unit clause $\{x\}$ added

8. recursively call DPLL on current CNF with the unit clause $\{\neg x\}$ added

9. if one of the recursive calls returns **satisfiable** return **satisfiable**

10. otherwise return **unsatisfiable**

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$$

Skip **1.** - **6.**, and choose $x = a$. First recursive call:

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b) \wedge a$$

unit resolution on a and removal of subsumed clauses gives

$$b \wedge (\neg b)$$

BCP gives empty clause, return **unsatisfiable**. Second recursive call:

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b) \wedge \neg a$$

BCP gives $\neg b$, only positive recurrence of b left, return **satisfiable**

(satisfying assignment $\{a \mapsto 0, b \mapsto 0\}$)

Expansion Theorem of Shannon

Theorem.

$$f(x) \quad \equiv \quad x \wedge f(1) \vee \overline{x} \wedge f(0)$$

Proof.

Let σ be an arbitrary assignment to variables in f including x

case $\sigma(x) = 0$:

$$\sigma(f(x)) = \sigma(f(0)) = \sigma(0 \wedge f(1) \vee 1 \wedge f(0)) = \sigma(x \wedge f(1) \vee \overline{x} \wedge f(0))$$

case $\sigma(x) = 1$:

$$\sigma(f(x)) = \sigma(f(1)) = \sigma(1 \wedge f(1) \vee 0 \wedge f(0)) = \sigma(x \wedge f(1) \vee \overline{x} \wedge f(0))$$

first observe: $x \wedge f(x)$ is satisfiable iff $x \wedge f(1)$ is satisfiable

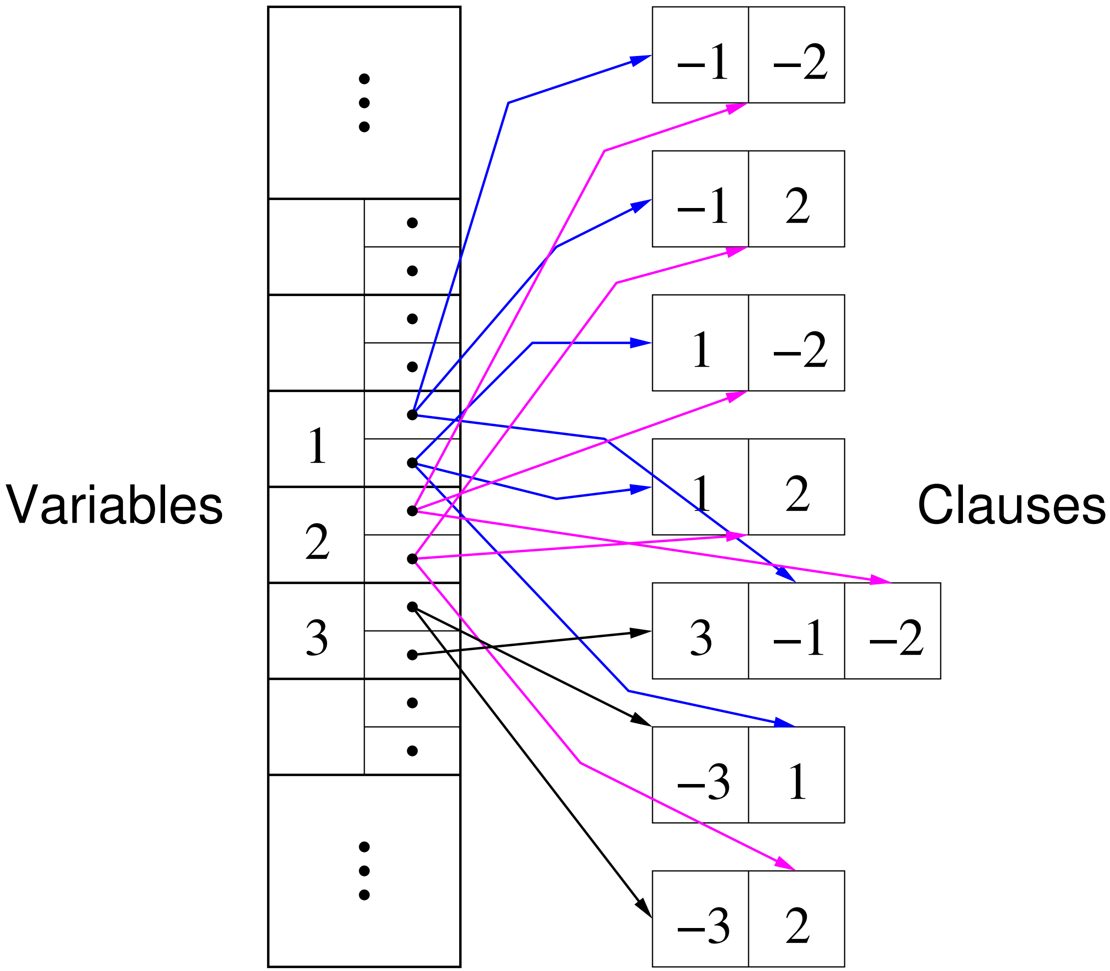
similarly, $\overline{x} \wedge f(x)$ is satisfiable iff $\overline{x} \wedge f(0)$ is satisfiable

then use expansion theorem of Shannon:

$f(x)$ satisfiable iff $\overline{x} \wedge f(0)$ or $x \wedge f(1)$ satisfiable iff $\overline{x} \wedge f(x)$ or $x \wedge f(x)$ satisfiable

rest follows along the lines of the the correctness proof for resolution based DP

Simple Data Structures in DP Implementation

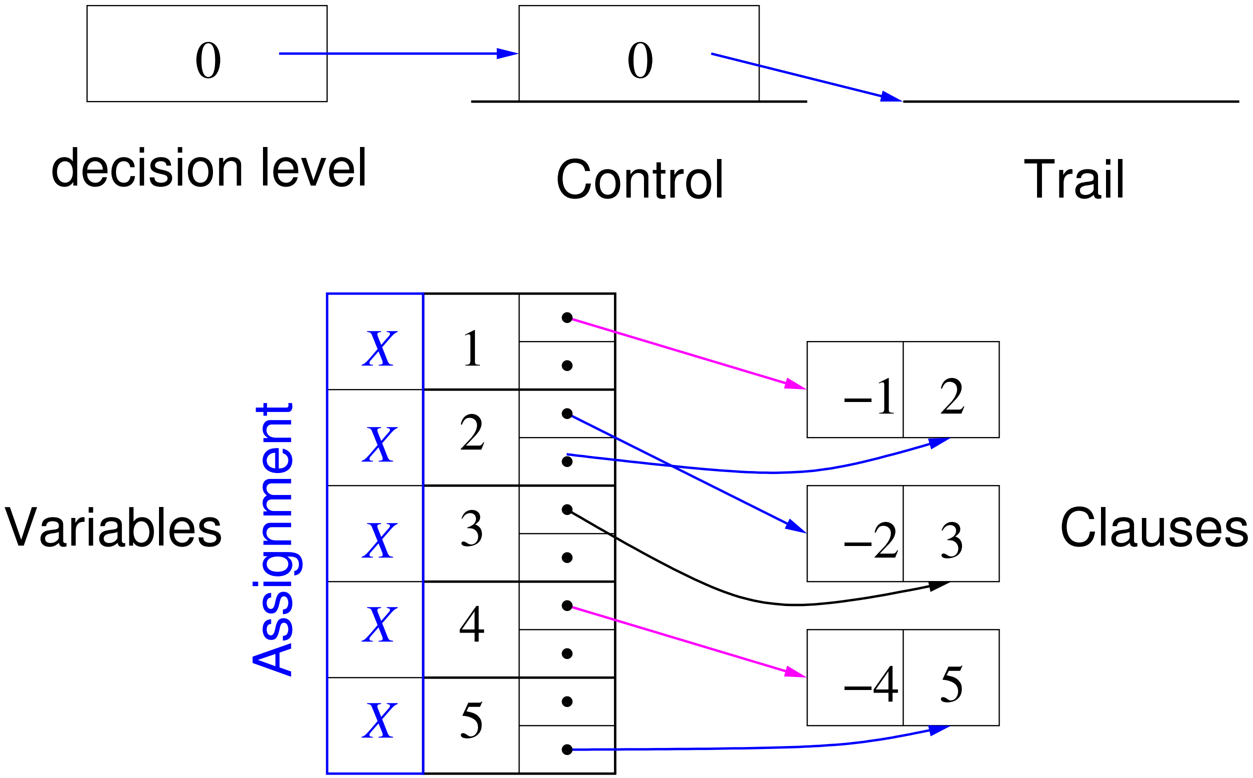


- each variable is marked as *unassigned*, *false*, or *true* ($\{X, 0, 1\}$)

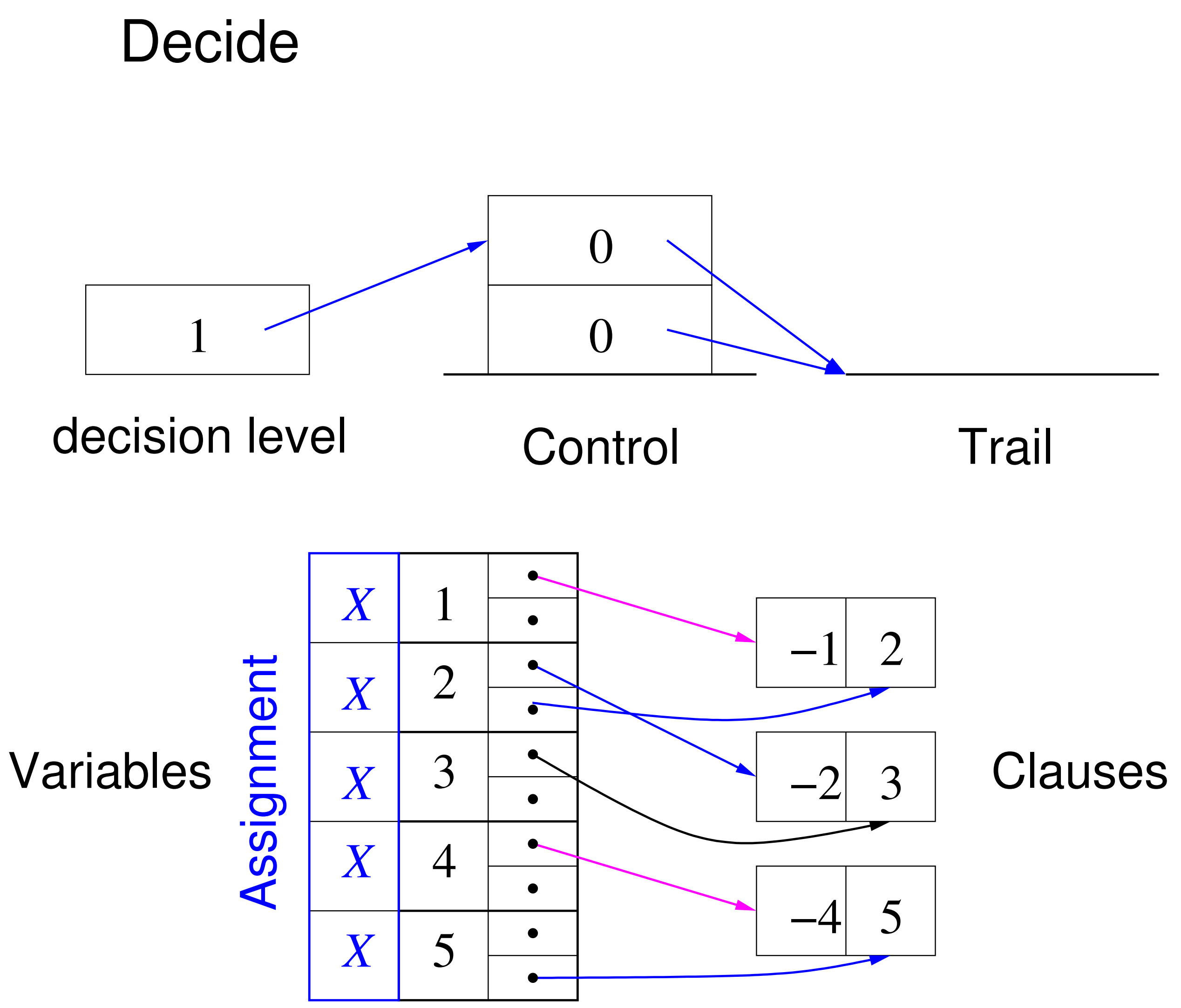
- no explicit resolution:
  - when a literal is assigned visit all clauses where its negation occurs
  - find those clauses which have all but one literal assigned to false
  - assign remaining non false literal to *true* and continue

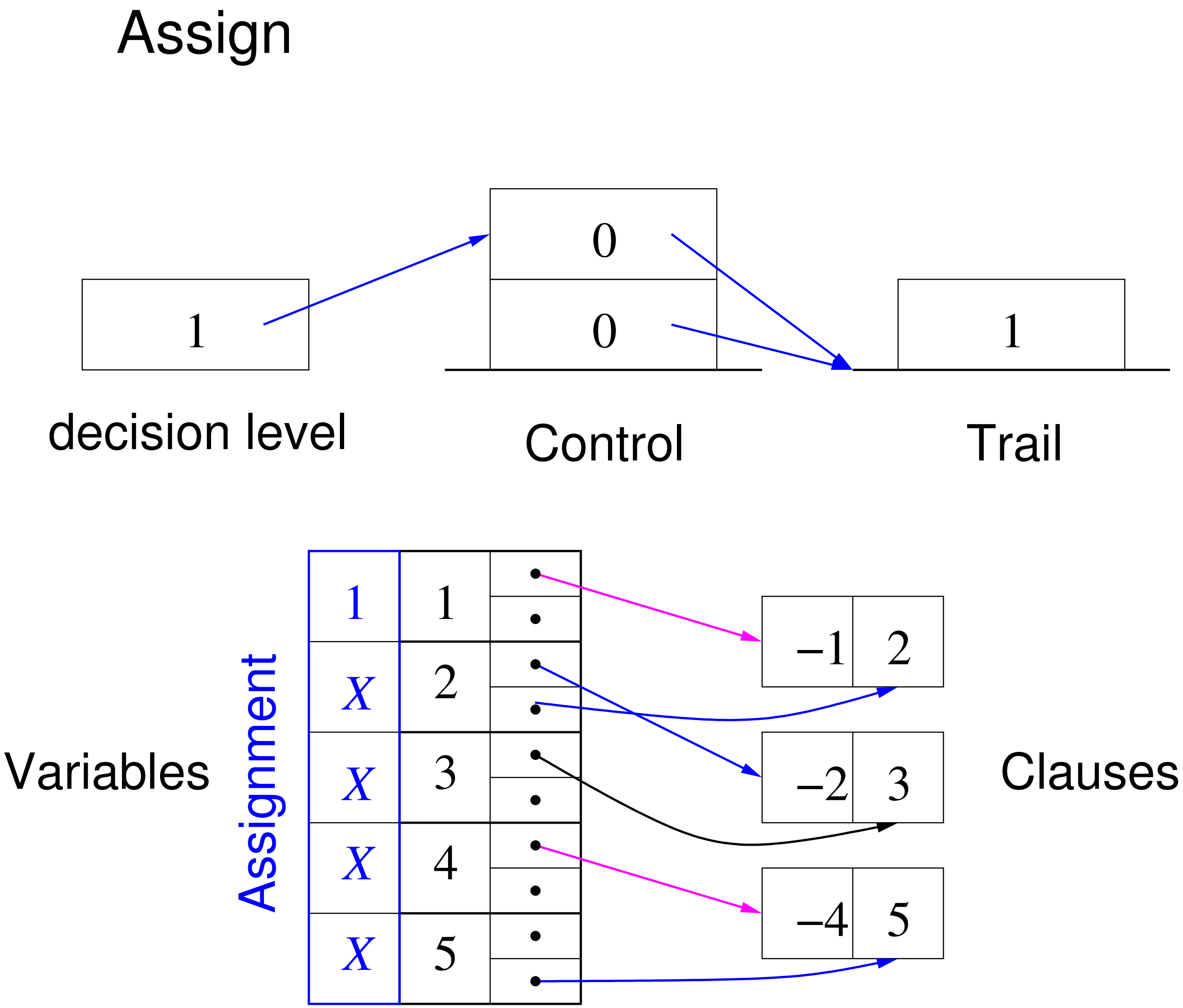
- decision:
  - heuristically find a variable that is still unassigned
  - heuristically determine phase for assignment of this variable

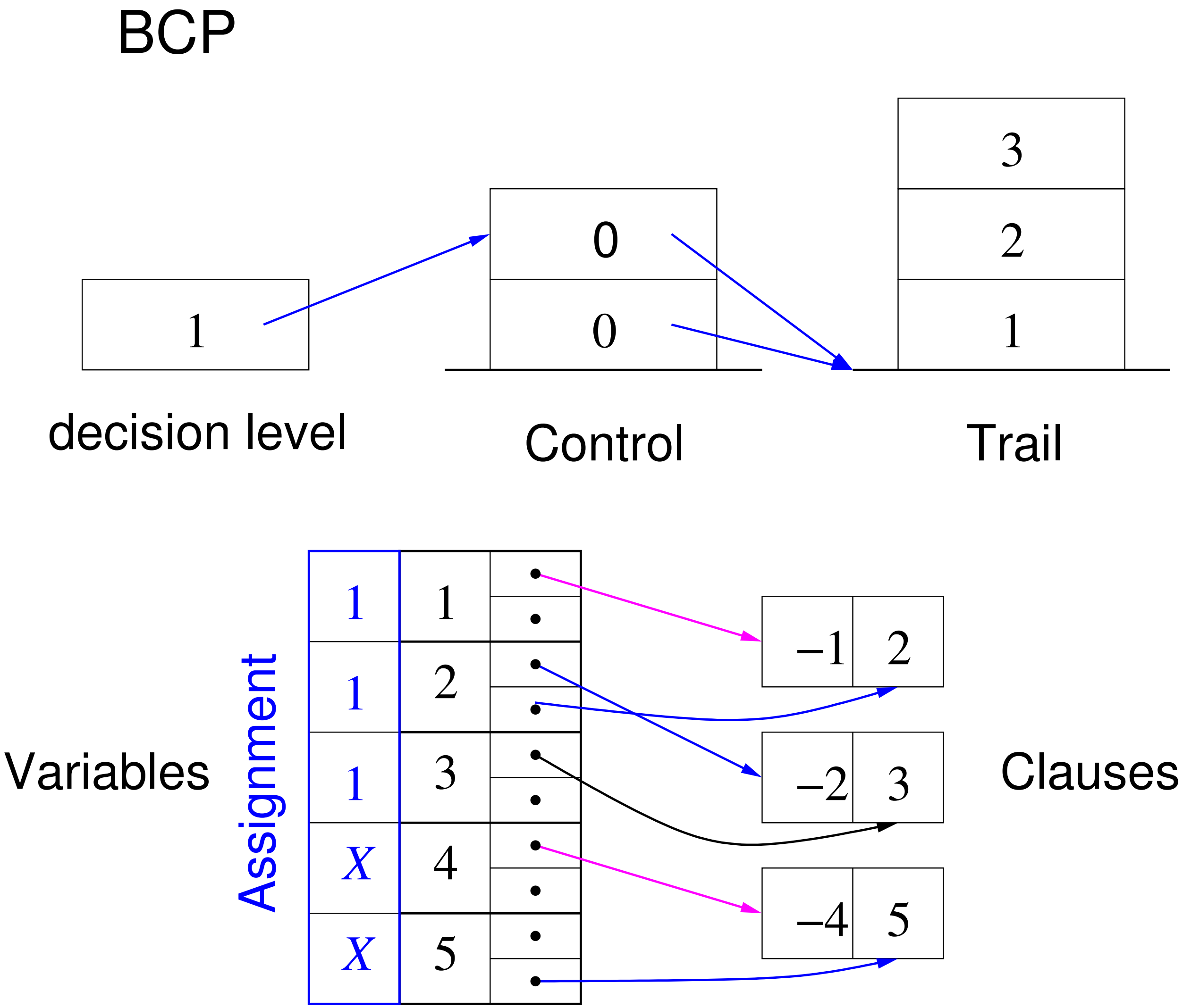
- *decision level* is the depth of recursive calls (= #nested decisions)
- the *trail* is a stack to remember order in which variables are assigned
- for each decision level the old trail height is saved on the *control stack*
- undoing assignments in backtracking:
  - get old trail height from control stack
  - unassign all variables up to the old trail height



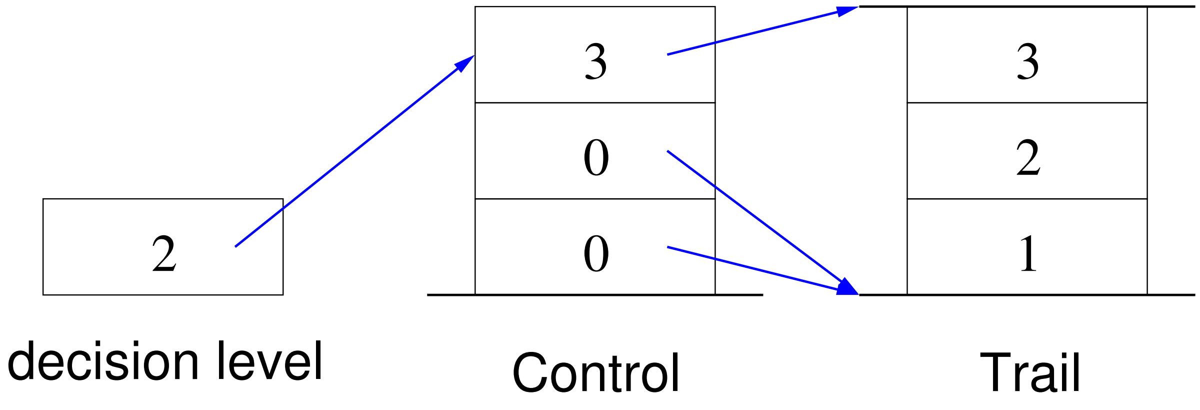
0
0
decision level
Control
Trail
Variables
Assignment
X
1
X
2
X
3
X
4
X
5
−1
2
−2
3
−4
5
Clauses

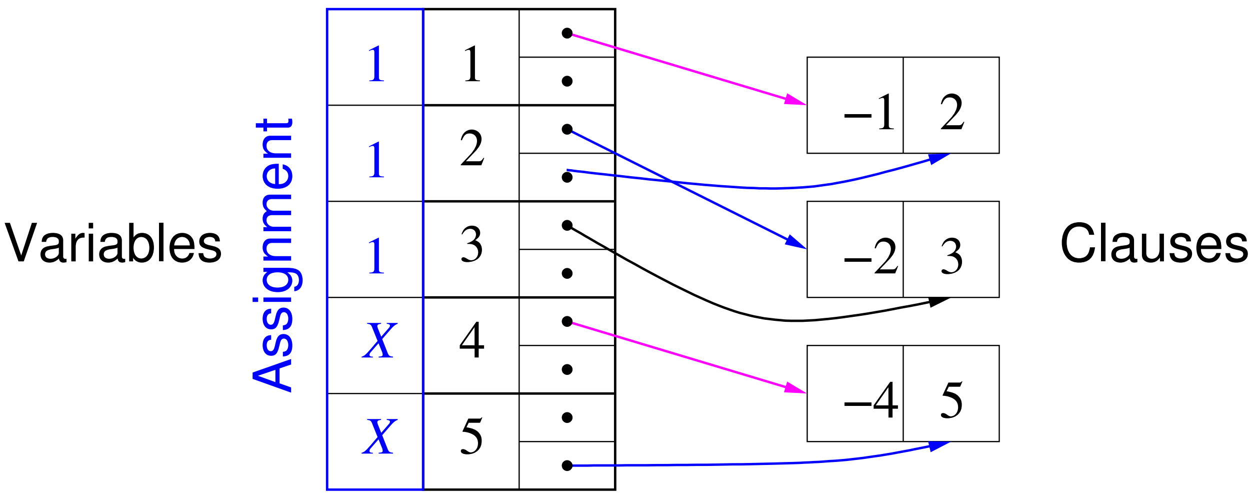

Decide
1
0
0
decision level
Control
Trail
Variables
Assignment
X 1
X 2
X 3
X 4
X 5
−1 2
−2 3
−4 5
Clauses

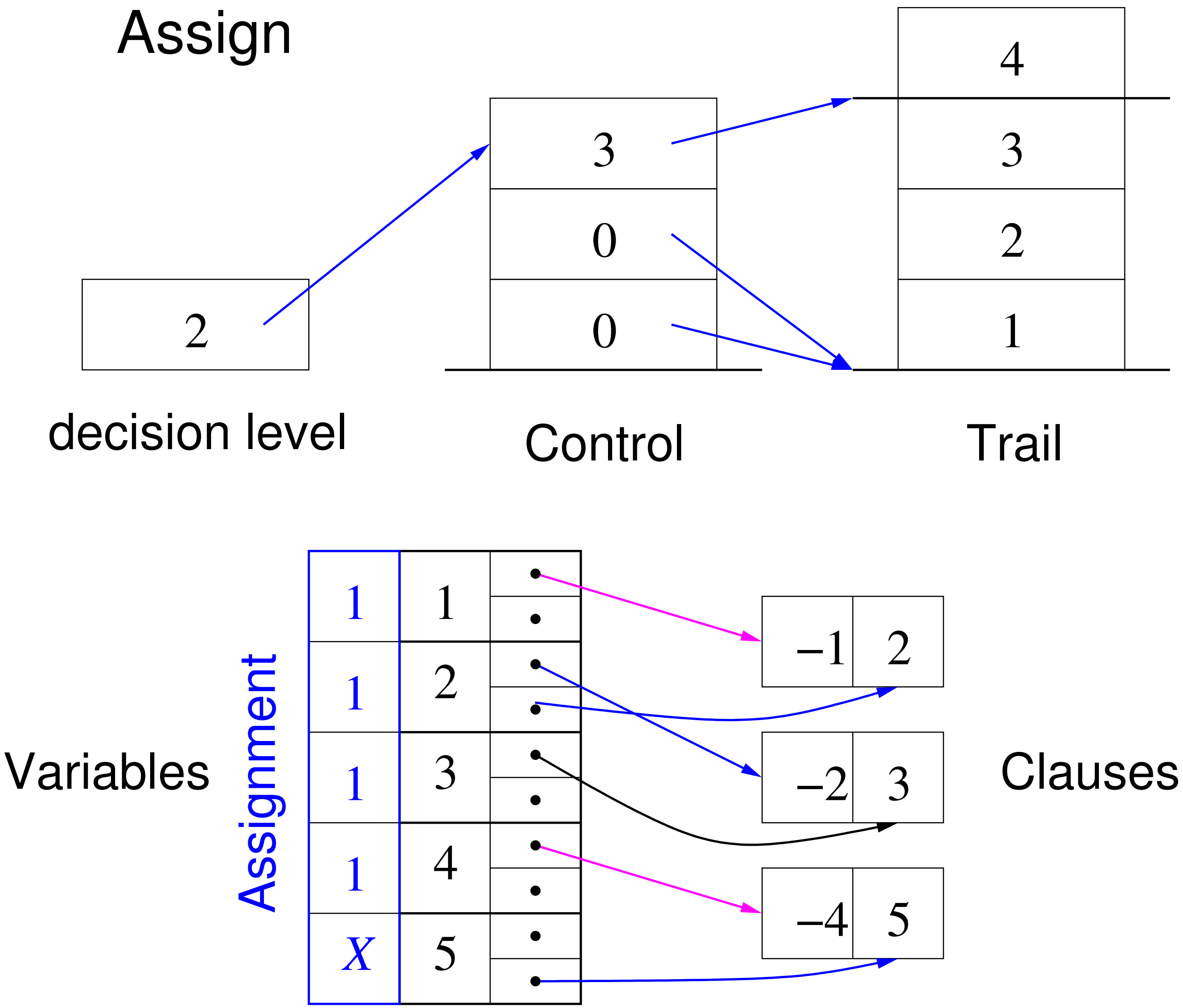

Assign
1
0
0
1
decision level
Control
Trail
Variables
Assignment
1
X
X
X
X
1
2
3
4
5
−1
2
−2
3
−4
5
Clauses

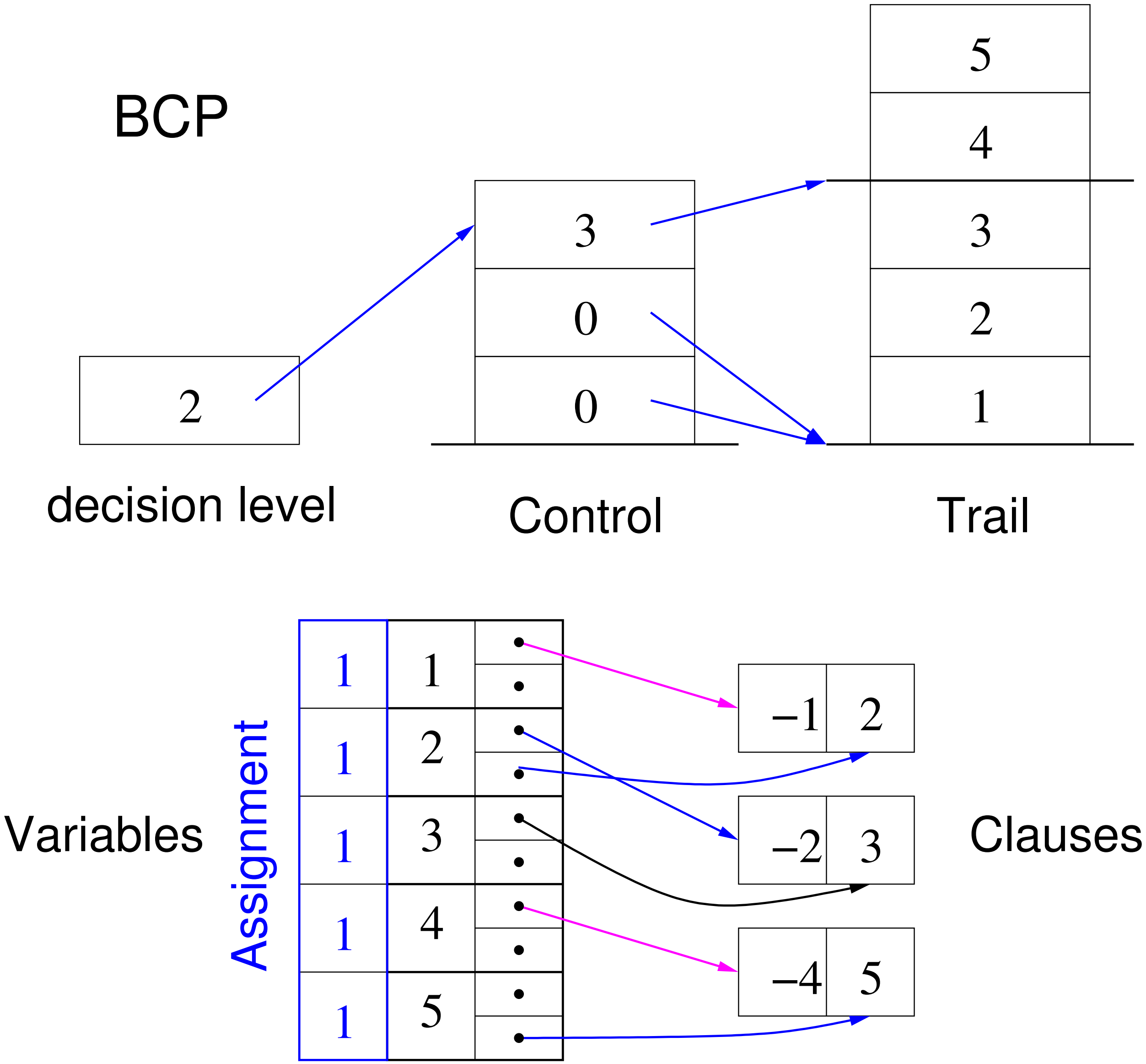

BCP
1
decision level
0
0
Control
3
2
1
Trail
Variables
Assignment
1 1
1 2
1 3
X 4
X 5
−1 2
−2 3
−4 5
Clauses

Decide


2
decision level
3
0
0
Control
3
2
1
Trail


Variables
Assignment
1 1
1 2
1 3
X 4
X 5
−1 2
−2 3
−4 5
Clauses


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



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

Decision Heuristics

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

- **dynamic heuristics**
  - typically calculated from number of occurences of literals
    (in unsatisfied clauses)
  - rather expensive, since it requires traversal of all clauses
    (or more expensive updates in BCP)
  - recently, *second order* dynamic heuristics (VSIDS in Chaff $\Rightarrow$ *see learning*)

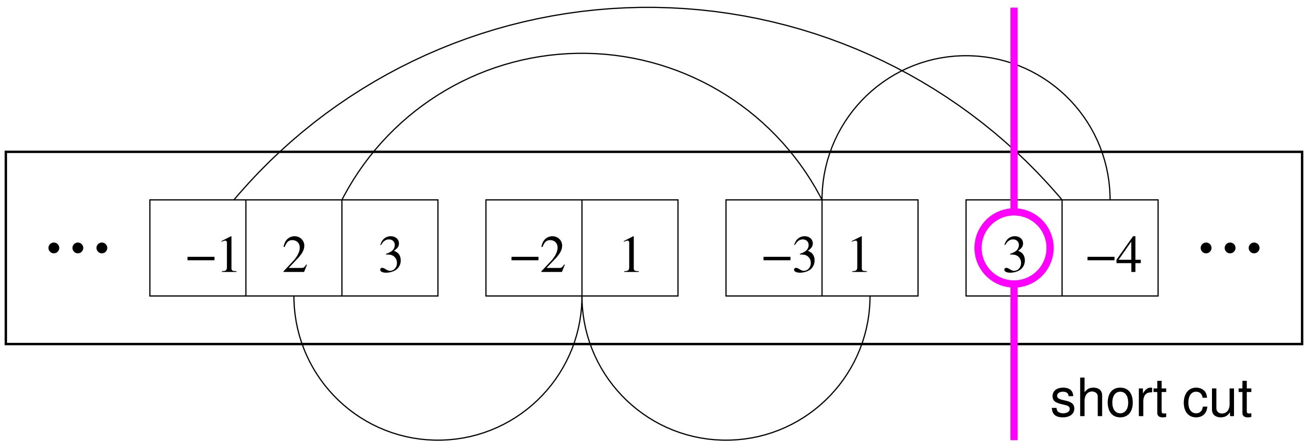
- view CNF as a graph:

  clauses as nodes, edges between clauses with same variable

- a *cut* is a set of variables that splits the graph in two parts

- recursively find short cuts that cut of parts of the graph

- static or dynamically order variables according to the cuts



assume
no occurences of
1, 2, −1, −2
on the right side

```
int
sat (CNF cnf)
{
  SetOfVariables cut = generate_good_cut (cnf);
  CNF assignment, left, right;

  left = cut_off_left_part (cut, cnf);
  right = cut_off_right_part (cut, cnf);

  forall_assignments (assignment, cut)
  {
    if (sat (apply (assignment, left)) && sat (apply (assignment, right)))
      return 1;
  }

  return 0;
}
```

Cut Width Heuristics cont.

- resembles cuts in circuits when CNF is generated with Tseitin transformation
- ideally cuts have constant or logarithmic size ...
  - for instance in tree like circuits
  - so the problem is *reconvergence*:
    the same signal / variable is used multiple times
- ... then satisfiability actually becomes polynomial (see exercise)

A clause is called *positive* if it contains a positive literal.

A clause is called *negative* if all its literals are negative.

A clause is a *Horn* clause if contains at most one positive literal.

CNF is in *Horn Form* iff all clauses are Horn clause (Prolog without negation)

Order assignments point-wise: $\sigma \leq \sigma'$ iff $\sigma(x) \leq \sigma'(x)$ for all $x \in V$

Horn Form with only positive clauses has minimal satisfying assignment.

Minimal satisfying assignment is obtained by BCP (polynomial).

A Horn Form is satisfiable iff the minimal assignments of its positive part satisfies all its negative clauses as well.

DP and Horn Form

- CNF in Horn Form: use above specialized fast algorithm
- non Horn: split on literals which occurs positive in non Horn clauses
  - actually choose variable which occurs most often in such clauses
- this gradually transforms non Horn CNF into Horn Form
- main heuristic in SAT solver SATO
- **Note:** In general, BCP in DP prunes search space by avoiding assignments incompatible to minimal satisfying assingment for the Horn part of the CNF.

non Horn part of CNF	Horn part of CNF

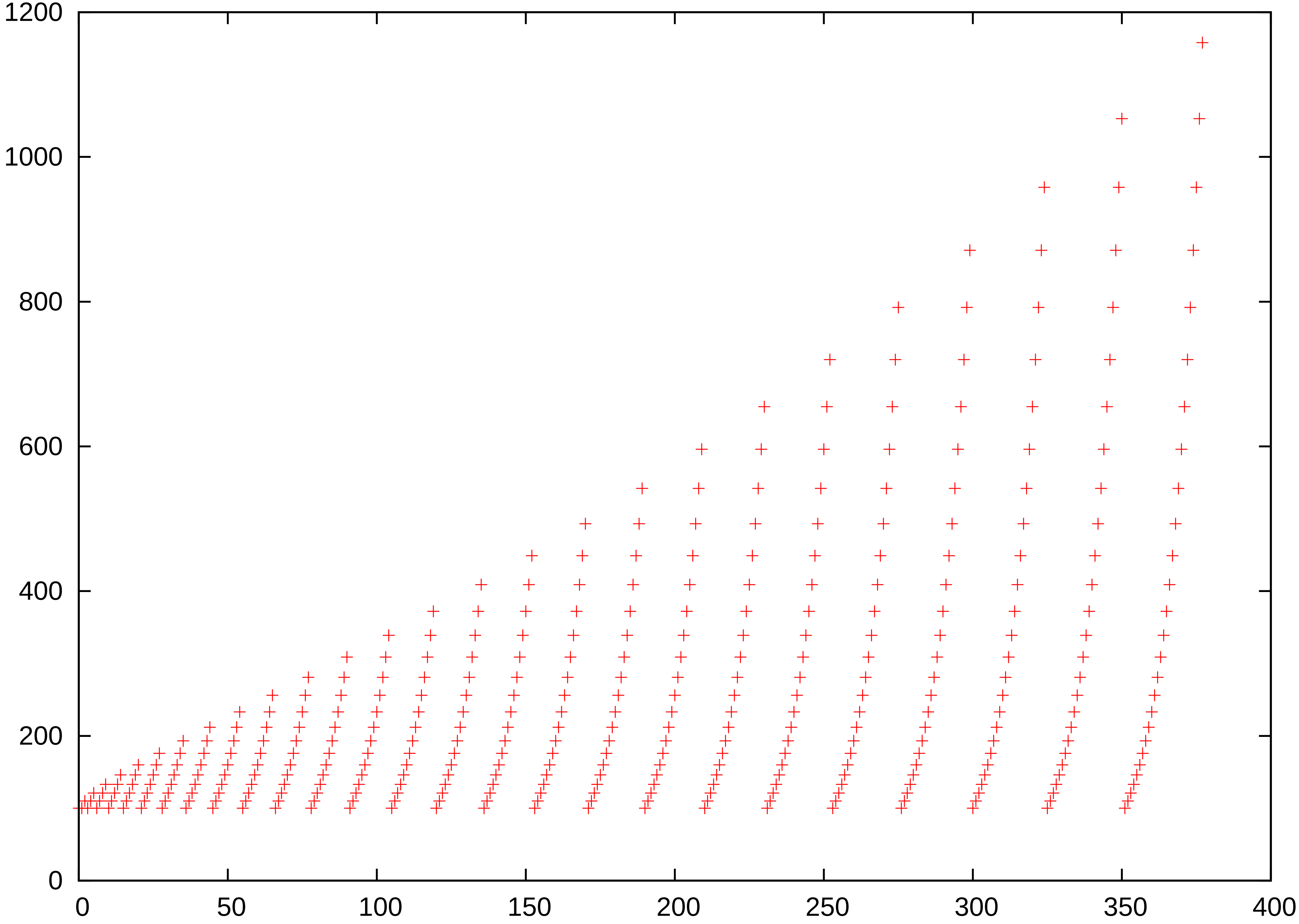
- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic first order heuristic (e.g. GRASP solver)
  - choose literal (variable + phase) which occurs most often
  - ignore satisfied clauses
  - requires explicit traversal of CNF (or more expensive BCP)

- look-forward heuristics (e.g. SATZ or MARCH solver) **failed literals, probing**
  - do trial assignments and BCP for all unassigned variables (both phases)
  - if BCP leads to conflict, force toggled assignment of current trial decision
  - skip trial assignments implied by previous trial assignments
    (removes a factor of $|V|$ from the runtime of one decision search)
  - decision variable maximizes number of propagated assignments

- distribution of SAT solver run-time shows *heavy tail behaviour*
- 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

Inner/Outer Restart Intervals

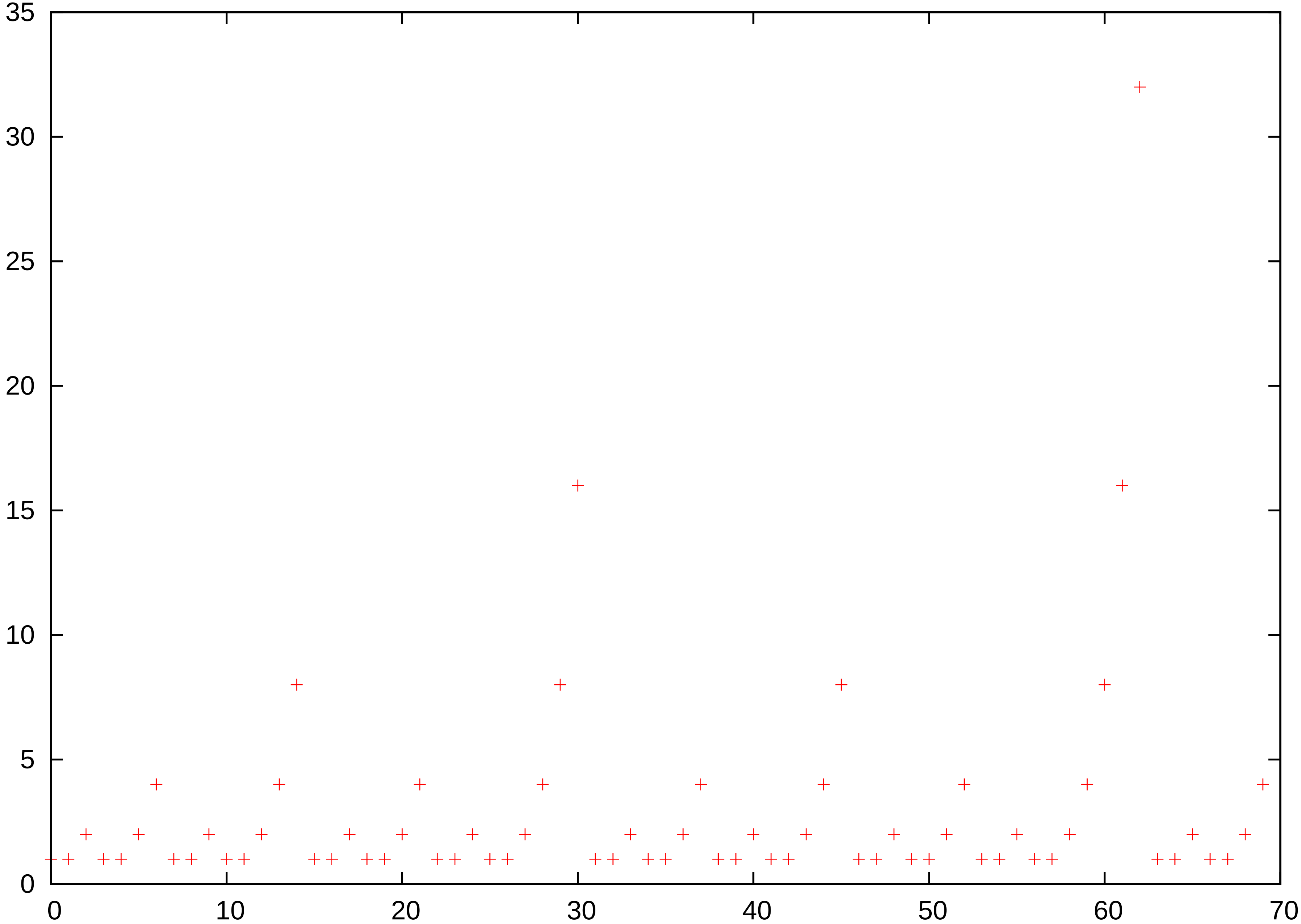
378 restarts in 104408 conflicts



```
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
```

[Knuth'12]

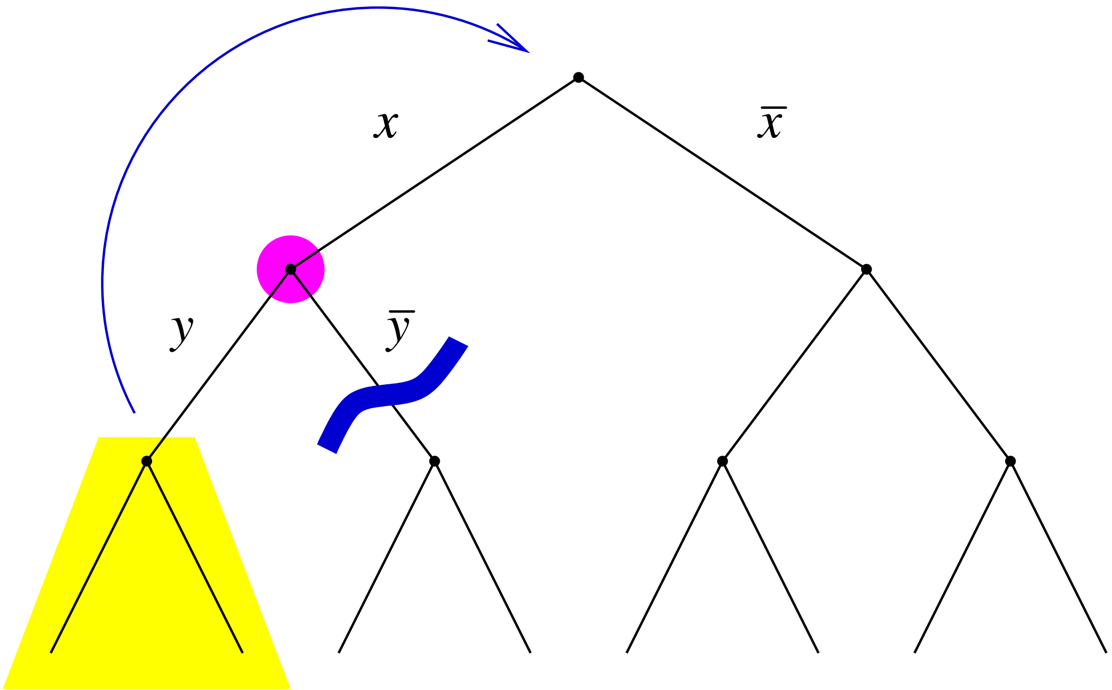
$$
\begin{aligned}
(u_1, v_1) &:= (1, 1) \\
(u_{n+1}, v_{n+1}) &:= (u_n \mathbin{\&} -u_n = v_n \; ? \; (u_n + 1, 1) : (u_n, 2v_n))
\end{aligned}
$$

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

- phase assignment:
  - assign decision variable to 0 or 1?
  - the only thing that matters in satisfiable instances

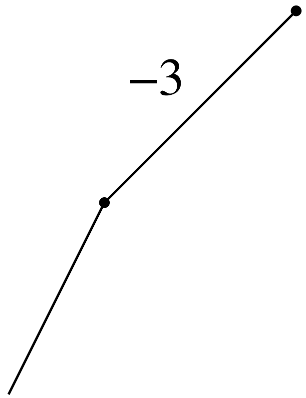
- "phase saving" as in RSat:
  - pick phase of last assignment (if not forced to, do not toggle assignment)
  - initially use statically computed phase (typically LIS)

- rapid restarts: varying restart interval with bursts of restarts
  - not ony theoretically avoids local minima
  - works nicely together with phase saving



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.



~~(−3 1)~~

~~(−3 2)~~

(−1 −2 $\cancel{3}$)

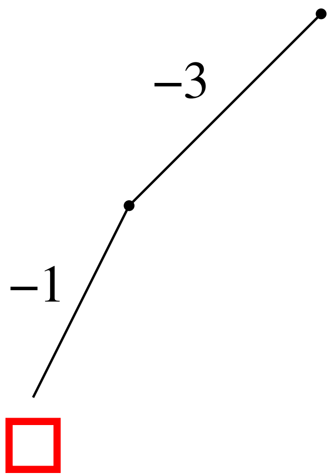
(−1 −2)

(−1 2)

(1 −2)

(1 2)

Split on -3 first (bad decision).



~~(−3 1)~~

~~(−3 2)~~

~~(−1 −2 $\cancel{3}$)~~

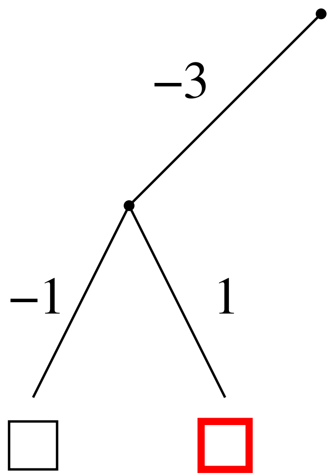
~~(−1 −2)~~

~~(−1 2)~~

($\cancel{1}$ −2)

($\cancel{1}$ 2)

Split on -1 and get first conflict.



$\cancel{(-3\ 1)}$

$\cancel{(-3\ 2)}$

$(\cancel{-1}\ {-2}\ \cancel{3})$

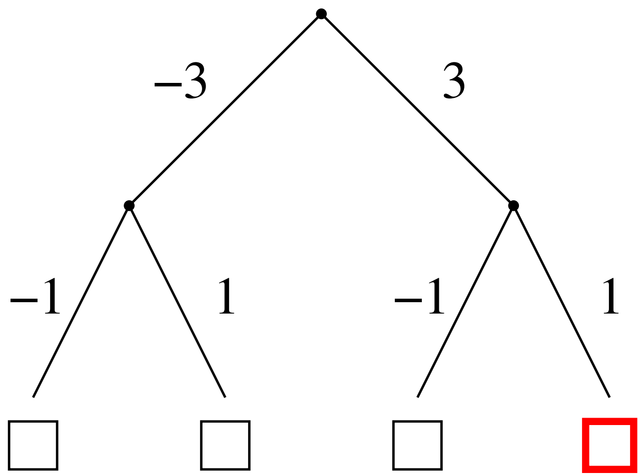
$(\cancel{-1}\ {-2})$

$(\cancel{-1}\ 2)$

$\cancel{(1\ {-2})}$

$\cancel{(1\ 2)}$

Regularly backtrack and assign 1 to get second conflict.



~~(–3 1)~~

~~(–3 2)~~

$(\cancel{-1}\ -2\ \cancel{3})$

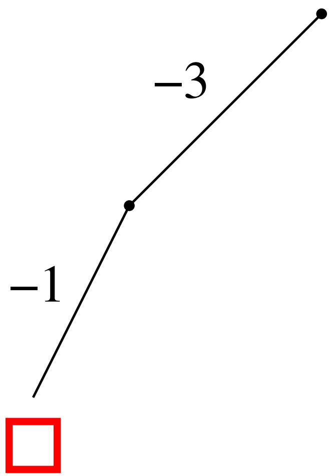
$(\cancel{-1}\ -2)$

$(\cancel{-1}\ 2)$

~~(1 –2)~~

~~(1 2)~~

Backtrack to root, assign 3 and derive same conflicts.



~~(−3 1)~~

~~(−3 2)~~

~~(−1 −2 $\cancel{3}$)~~

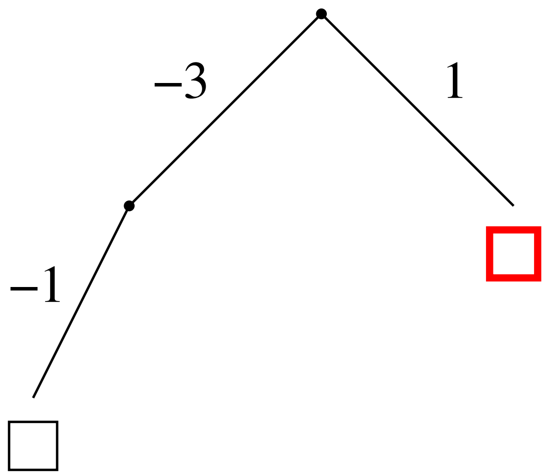
~~(−1 −2)~~

~~(−1 2)~~

($\cancel{1}$ −2)

($\cancel{1}$ 2)

Assignment -3 does not contribute to conflict.



$\cancel{(-3\ 1)}$

$(-3\ 2)$

$(\cancel{-1}\ -2\ 3)$

$(\cancel{-1}\ -2)$

$(\cancel{-1}\ 2)$

$\cancel{(1\ -2)}$

$\cancel{(1\ 2)}$

So just *backjump* to root before assigning 1.

- backjumping helps to *recover* from bad decisions
  - bad decisions are those that do not contribute to conflicts
  - without backjumping same conflicts are generated in second branch
  - with backjumping the second branch of bad decisions is just skipped
- particularly useful for unsatisfiable instances
  - in satisfiable instances good decisions will guide us to the solution
- with backjumping many bad decisions increase search space roughly quadratically instead of exponentially with the number of bad decisions

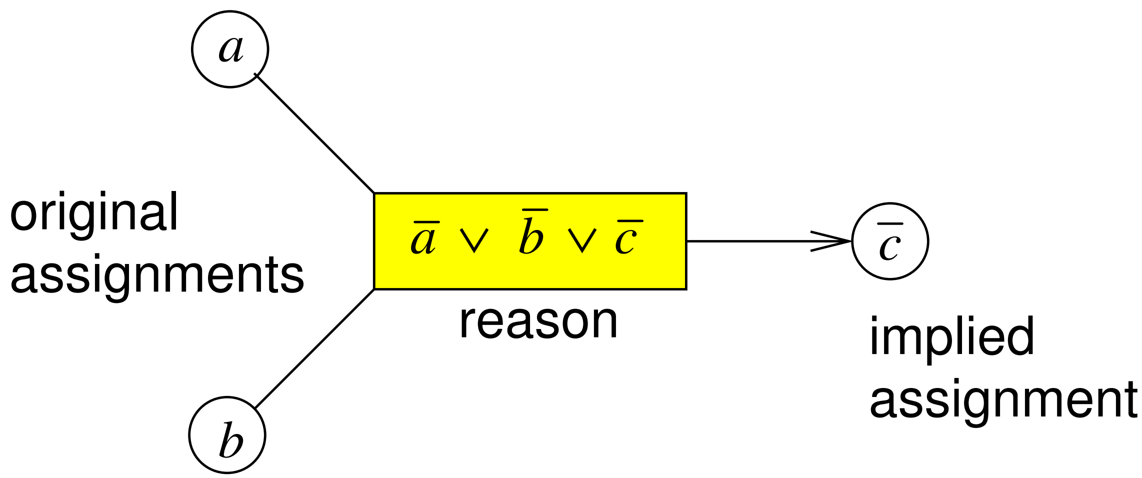
- the implication graph maps inputs to the result of resolutions

- backward from the empty clause all contributing clauses can be found

- the variables in the contributing clauses are contributing to the conflict

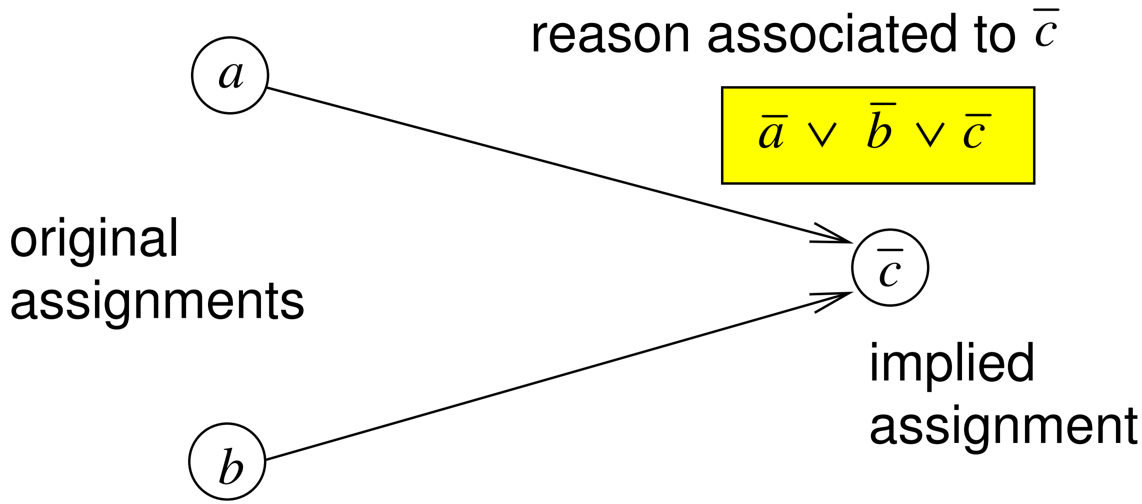
- important optimization, since we only use unit resolution
  - generate graph only for resolutions that result in unit clauses
  - the assignment of a variable is result of a decision or a unit resolution
  - therefore the graph can be represented by saving the *reasons* for assignments with each assigned variable

General Implication Graph as Hyper-Graph



(edges of directed hyper graphs may have multiple source and target nodes)

Optimized Implication Graph for Unit Resolution in DP



- graph becomes an ordinary (non hyper) directed graph
- simplifies implementation:
  - store a pointer to the reason clause with each assigned variable
  - decision variables just have a null pointer as reason
  - decisions are the roots of the graph

- can we *learn* more from a conflict?
  - backjumping does not *fully* avoid the occurrence of the same conflict
  - the same (partial) assignments may generate the same conflict
- generate *conflict clauses* and add them to CNF
  - the literals contributing to a conflict form a partial assignment
  - this partial assignment is just a conjunction of literals
  - its negation is a clause (implied by the original CNF)
  - adding this clause avoids this partial assignment to happen again

[MarquesSilvaSakallah'96: GRASP]

- observation: current decision always contributes to conflict
  - otherwise BCP would have generated conflict one decision level lower
  - conflict clause has (exactly one) literal assigned on current decision level
- instead of backtracking
  - generate and add conflict clause
  - undo assignments as long conflict clause is empty or unit clause
    (in case conflict clause is the empty clause conclude unsatisfiability)
  - resulting assignment from unit clause is called *conflict driven assignment*

CNF for following Examples

```
-3 1 2 0
3 -1 0
3 -2 0
-4 -1 0
-4 -2 0
-3 4 0
3 -4 0
-3 5 6 0
3 -5 0
3 -6 0
4 5 6 0
```

We use a version of the DIMACS format.

Variables are represented as positive integers.

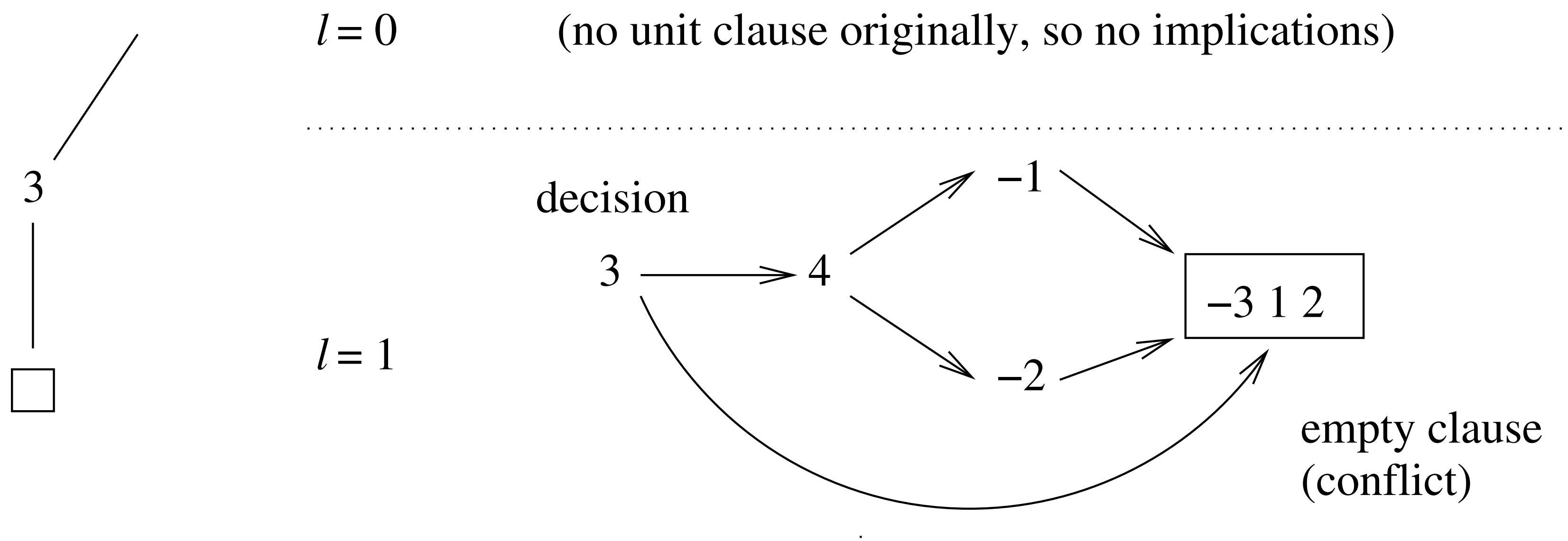
Integers represent literals.

Subtraction means negation.

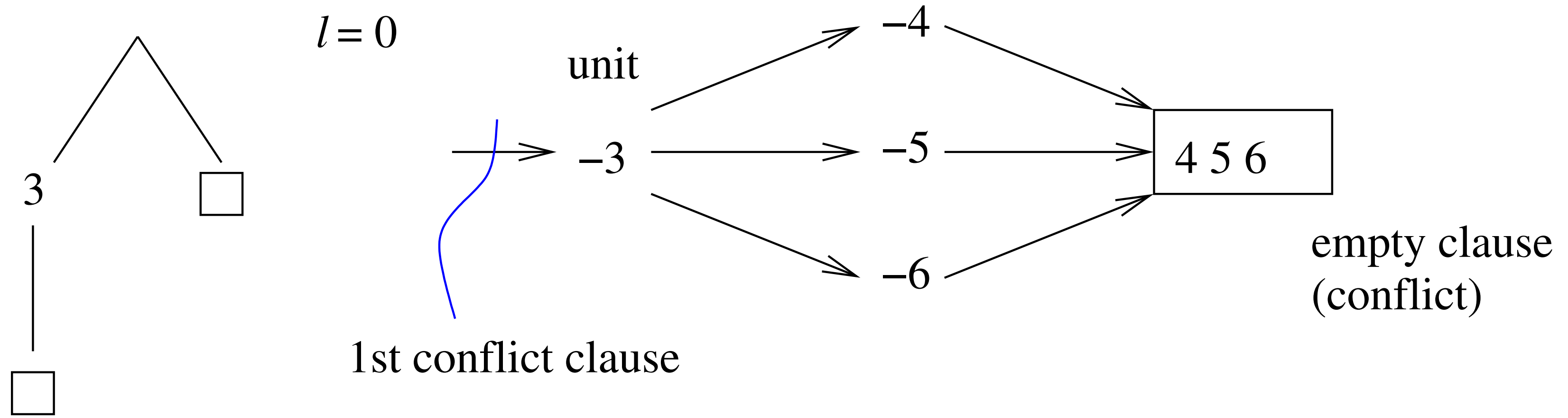
A clause is a zero terminated list of integers.

CNF has a good cut made of variables 3 and 4 (cf Exercise 4 + 5).
(but we are going to apply DP with learning to it)

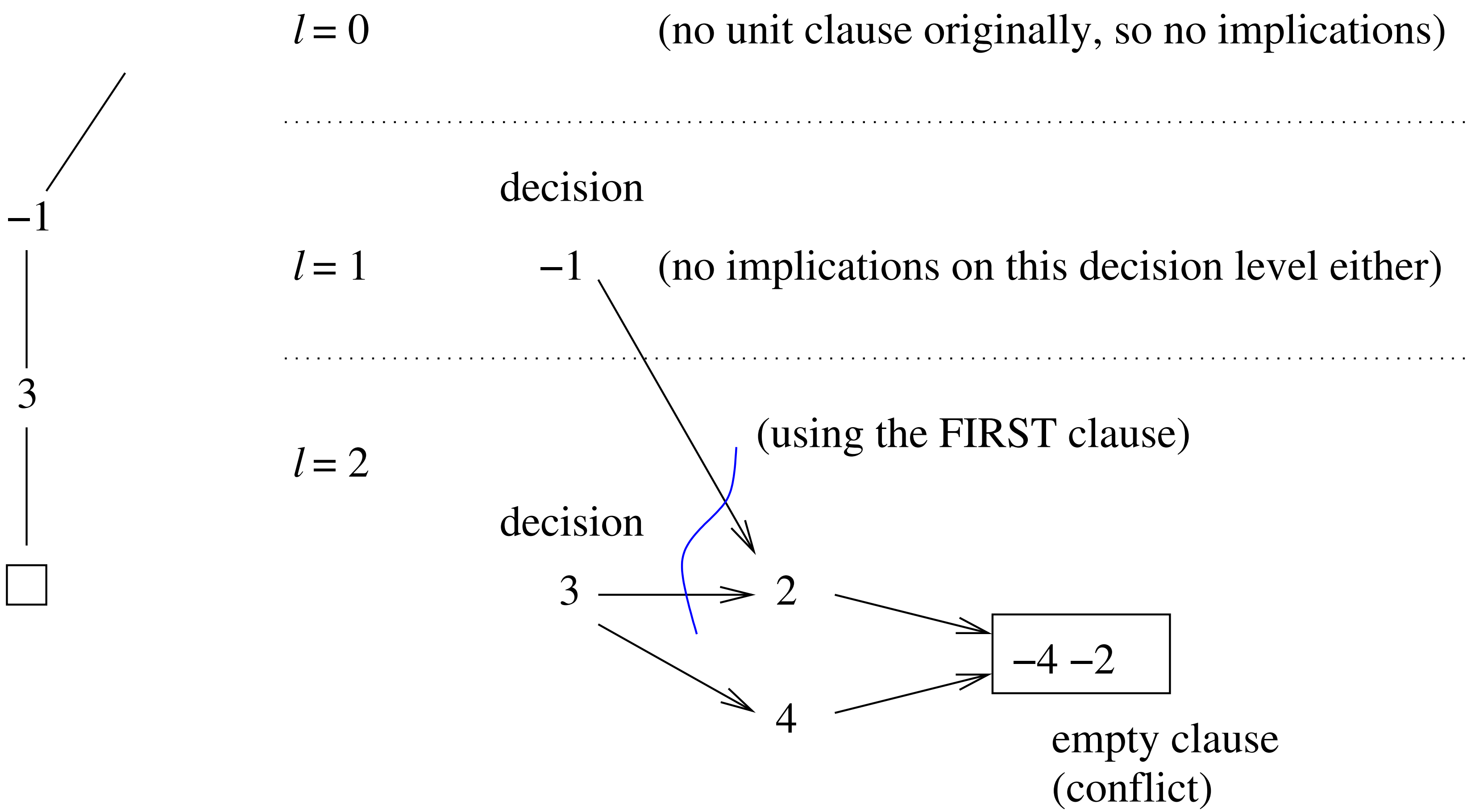
DP with Learning Run 1 (3 as 1st decision)



unit clause −3 is generated as learned clause and we backtrackt to $l = 0$



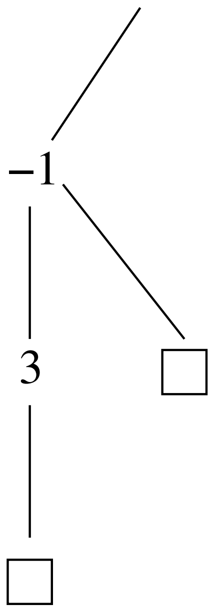
since −3 has a real unit clause as reason, an empty conflict clause is learned

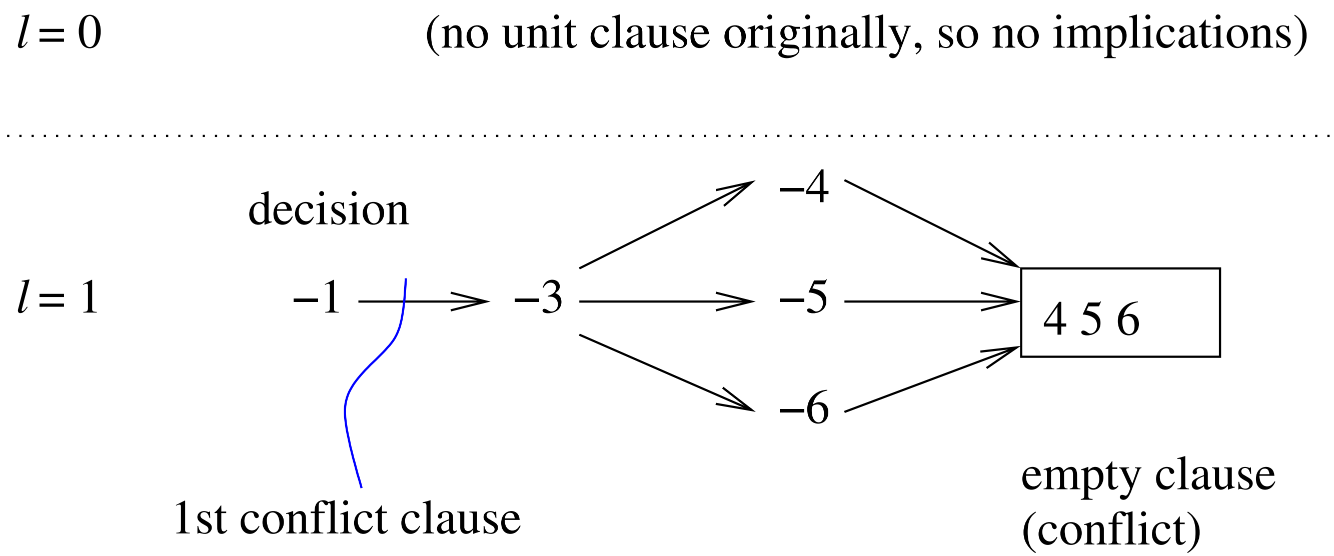


since FIRST clause was used to derive 2, conflict clause is $(1\ -3)$

backtrack to $l = 1$ (smallest level for which conflict clause is a unit clause)

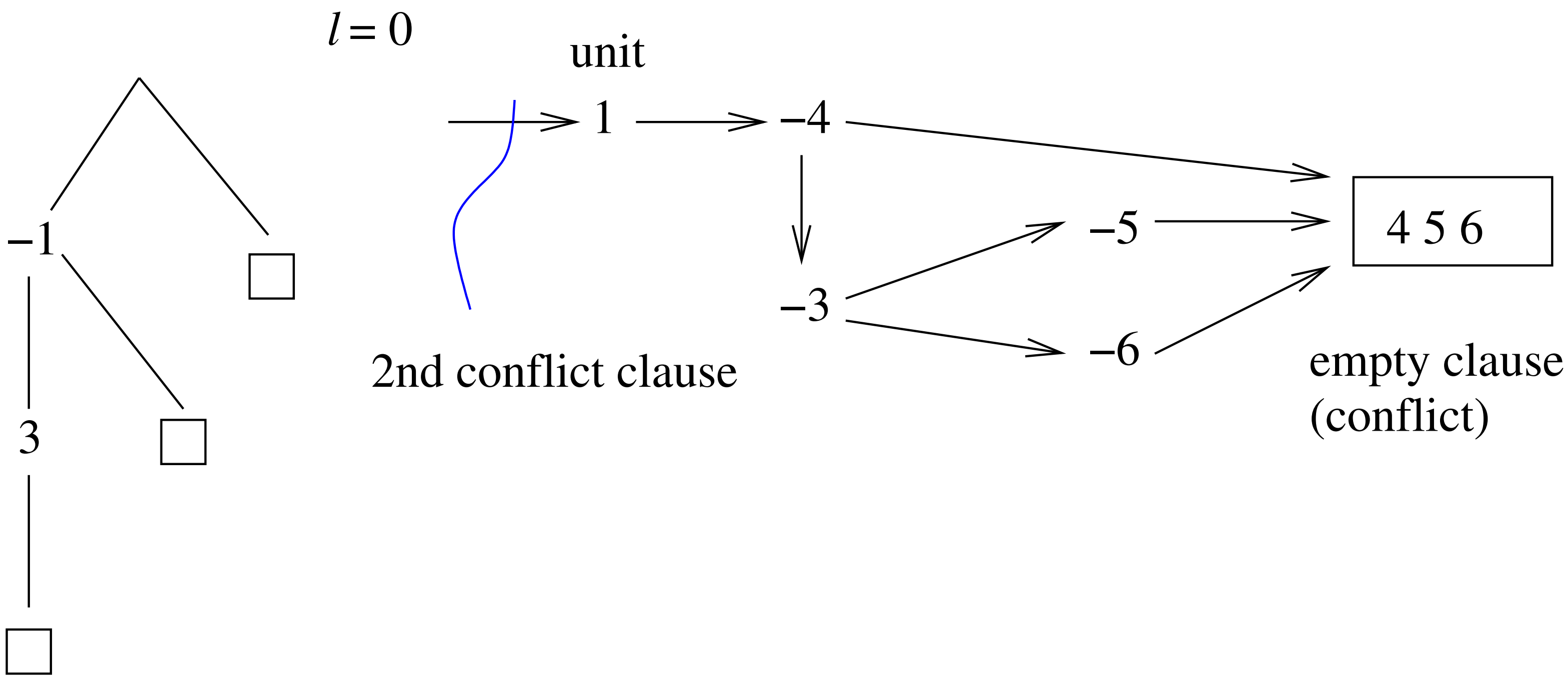
DP with Learning Run 2 Fig. 2 (-1, 3 as decision order)



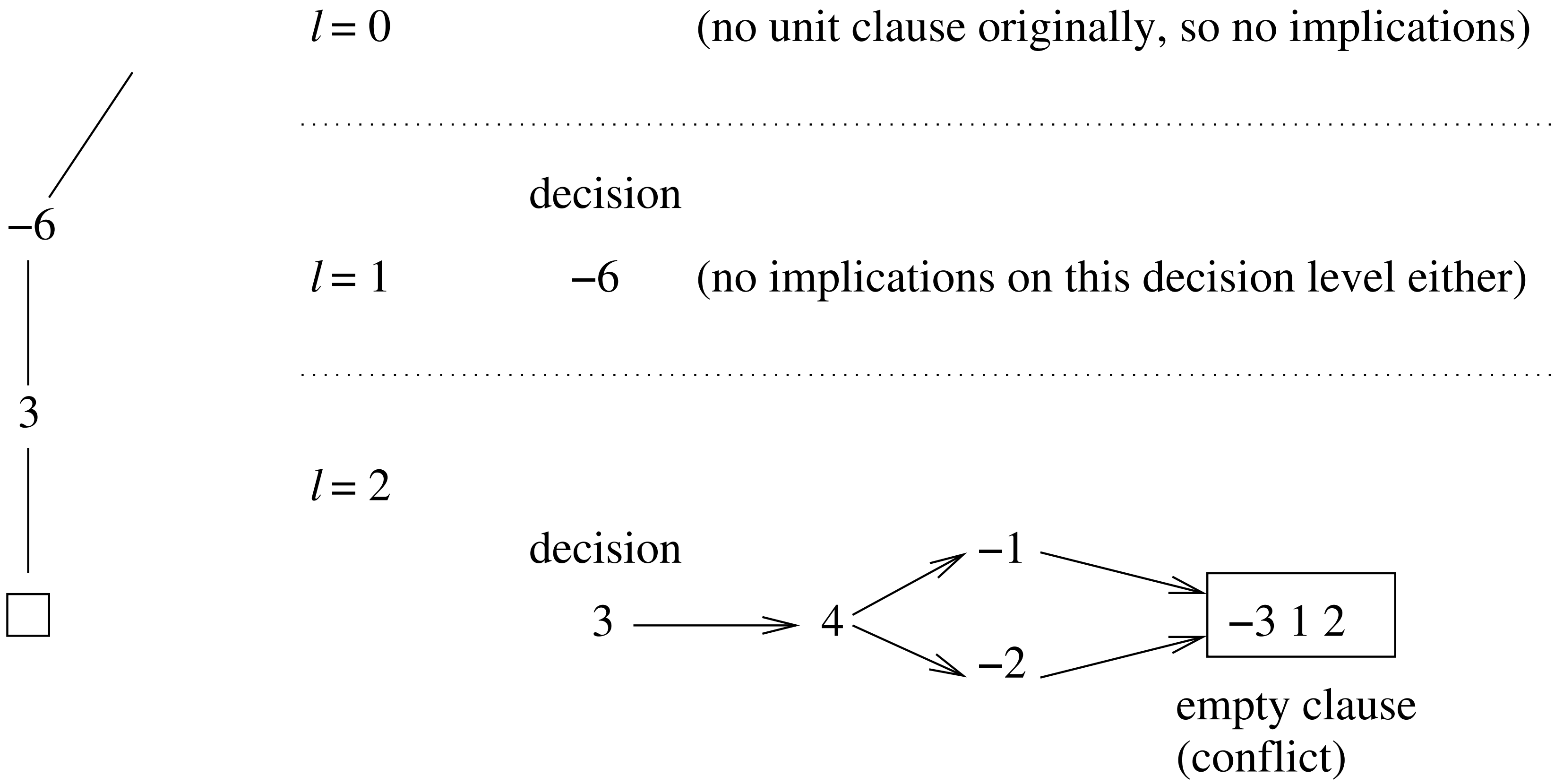


learned conflict clause is the unit clause 1

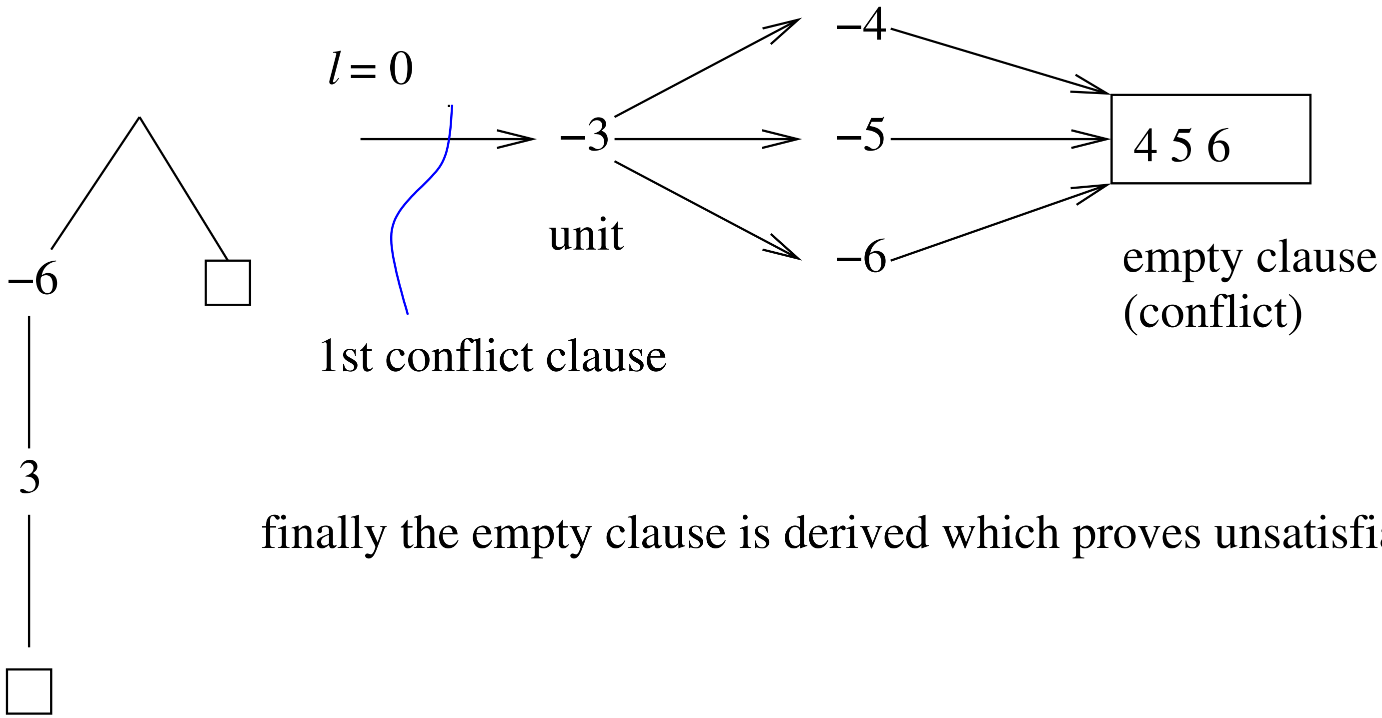
backtrack to decision level $l = 0$



since the learned clause is the empty clause, conclude unsatisfiability



learn the unit clause -3 and BACKJUMP to decision level $l = 0$



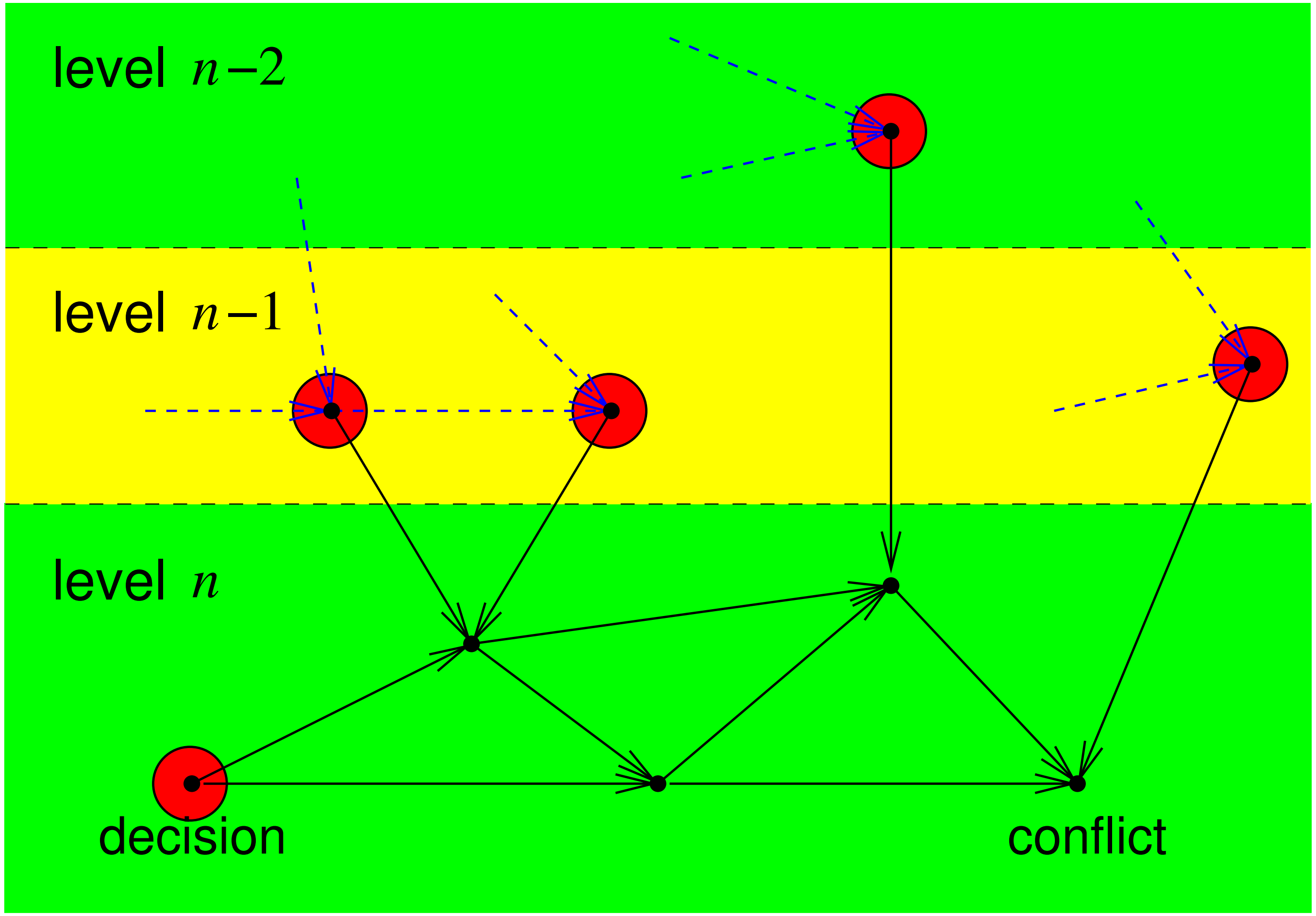
finally the empty clause is derived which proves unsatisfiability

Toplevel Loop in DP with Learning


```
int
sat (Solver solver)
{
  Clause conflict;

  for (;;)
    {
      if (bcp_queue_is_empty (solver) && !decide (solver))
        return SATISFIABLE;

      conflict = deduce (solver);

      if (conflict && !backtrack (solver, conflict))
        return UNSATISFIABLE;
    }
}
```

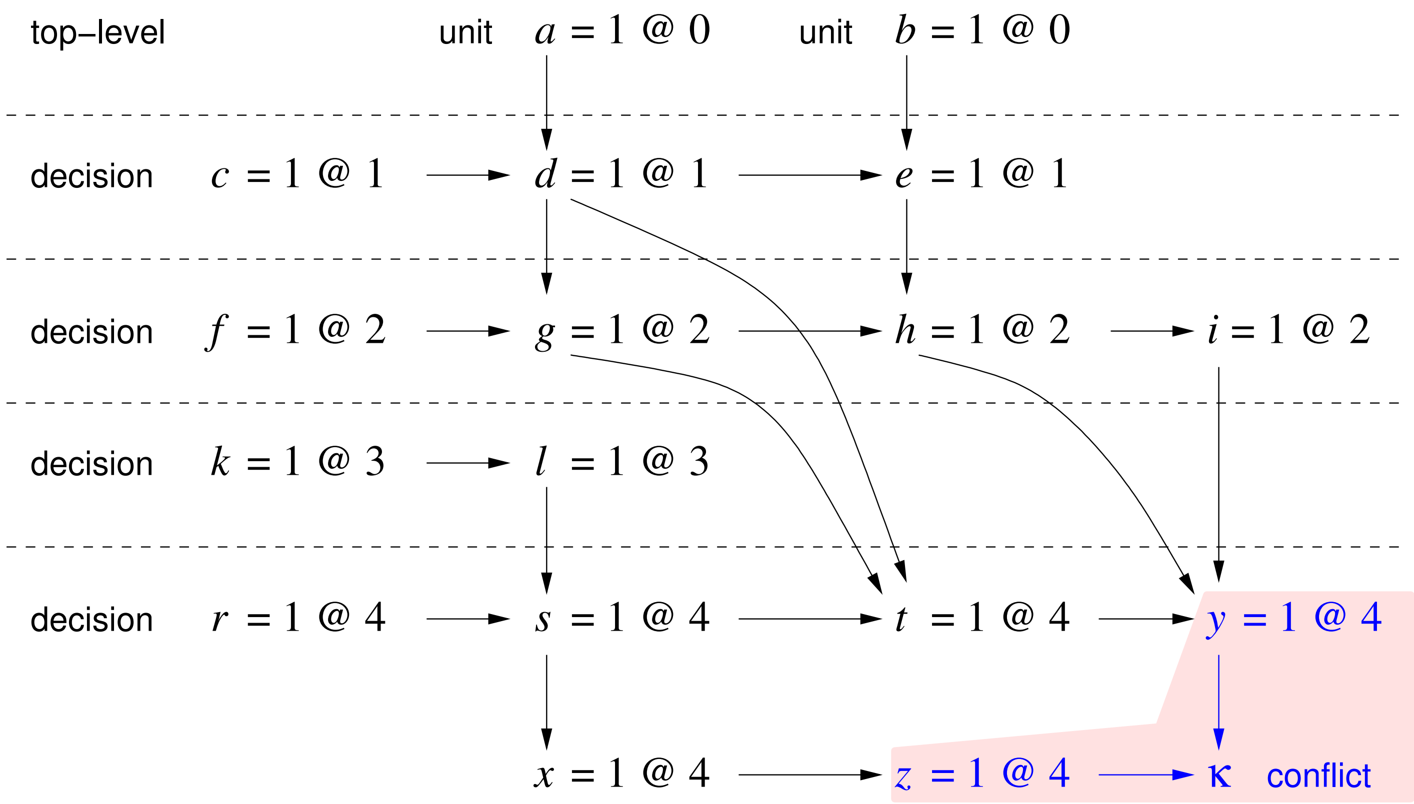
```c
int
backtrack (Solver solver, Clause conflict)
{
  Clause learned_clause; Assignment assignment; int new_level;

  if (decision_level(solver) == 0)
    return 0;

  analyze (solver, conflict);
  learned_clause = add (solver);

  assignment = drive (solver, learned_clause);
  enqueue_bcp_queue (solver, assignment);

  new_level = jump (solver, learned_clause);
  undo (solver, new_level);

  return 1;
}
```

- conflict clause: obtained by backward resolving empty clause with reasons
  - start at clause which has all its literals assigned to false
  - resolve one of the false literals with its reason
  - invariant: result still has all its literals assigned to false
  - continue until user defined size is reached
- gives a nice correspondence between resolution and learning in DP
  - allows to generate a resolution proof from a DP run
  - implemented in RELSAT solver [BayardoSchrag'97]



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

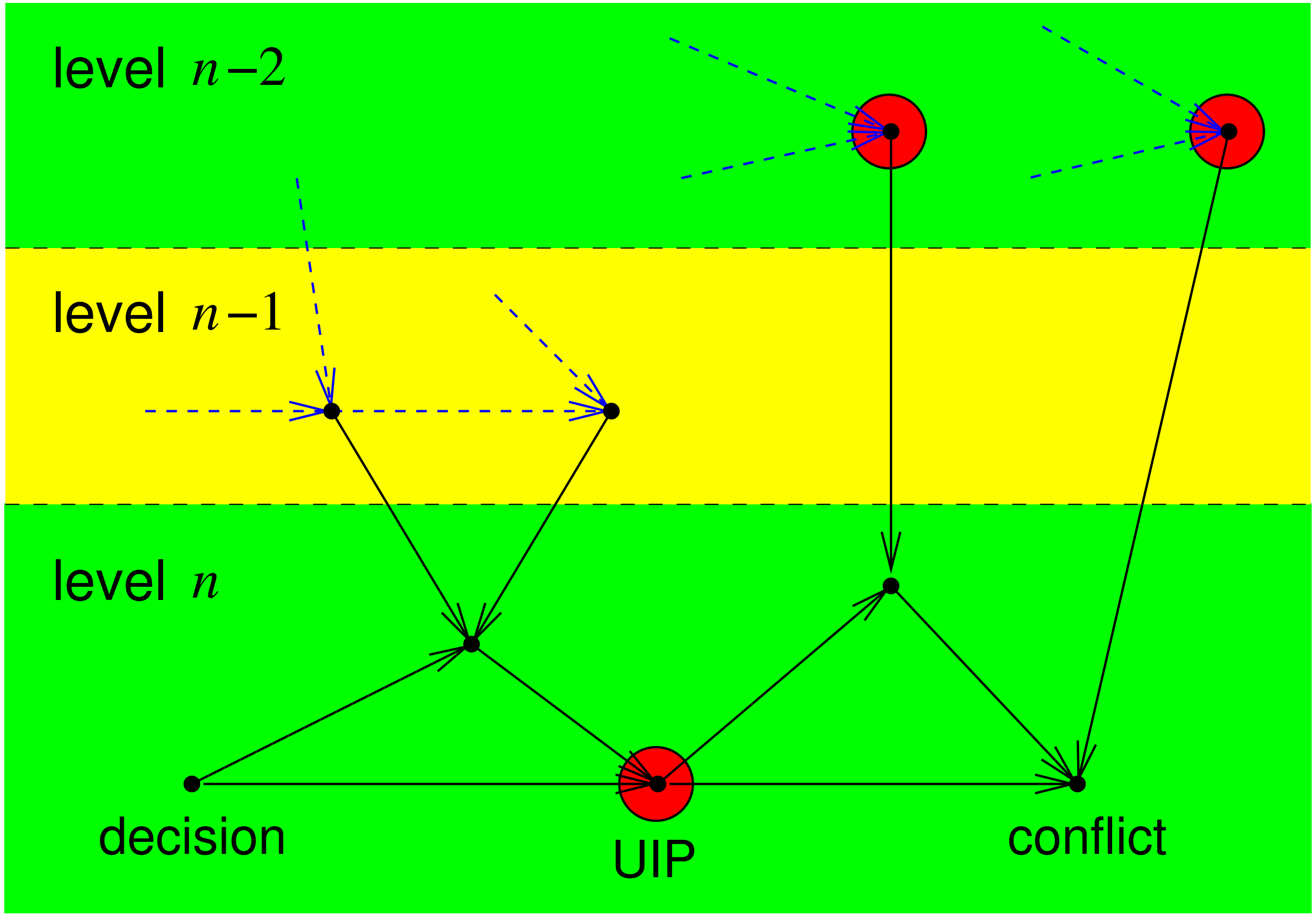
Unique Implication Points (UIP)



UIP = *articulation point* in graph decomposition into biconnected components
(simply a node which, if removed, would disconnect two parts of the graph)

- 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)
- if all paths converged at one node, then UIP is found
- decision of current decision level is a UIP and thus a *sentinel*

- assume a non decision UIP is found
- this UIP is part of a potential cut
- graph traversal may stop (everything *behind* the UIP is ignored)
- negation of the UIP literal constitutes the conflict driven assignment
- may start new clause generation (UIP replaces conflict)
  - each conflict may generate multiple learned clauses
  - however, using only the first UIP encountered seems to work best

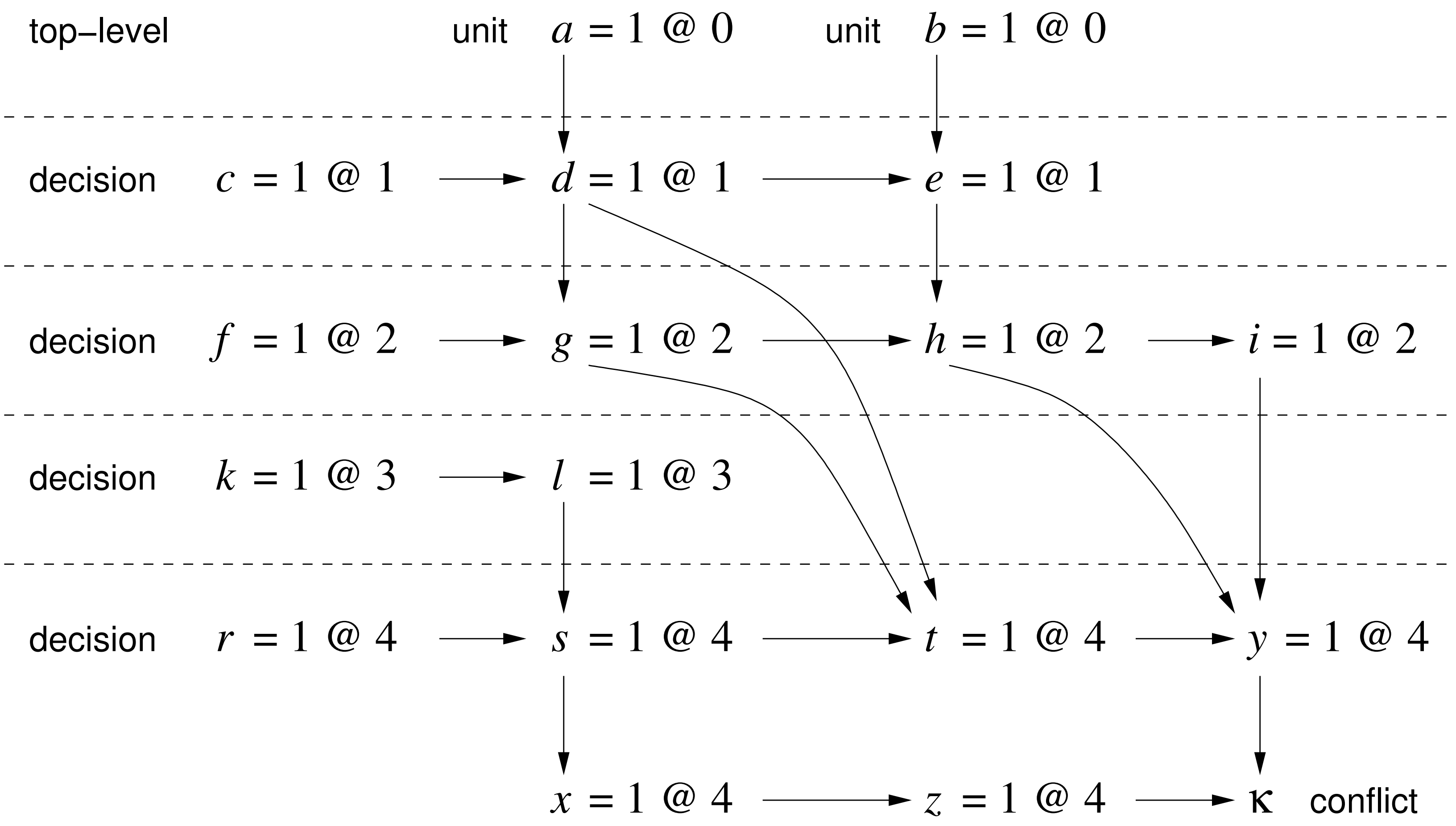


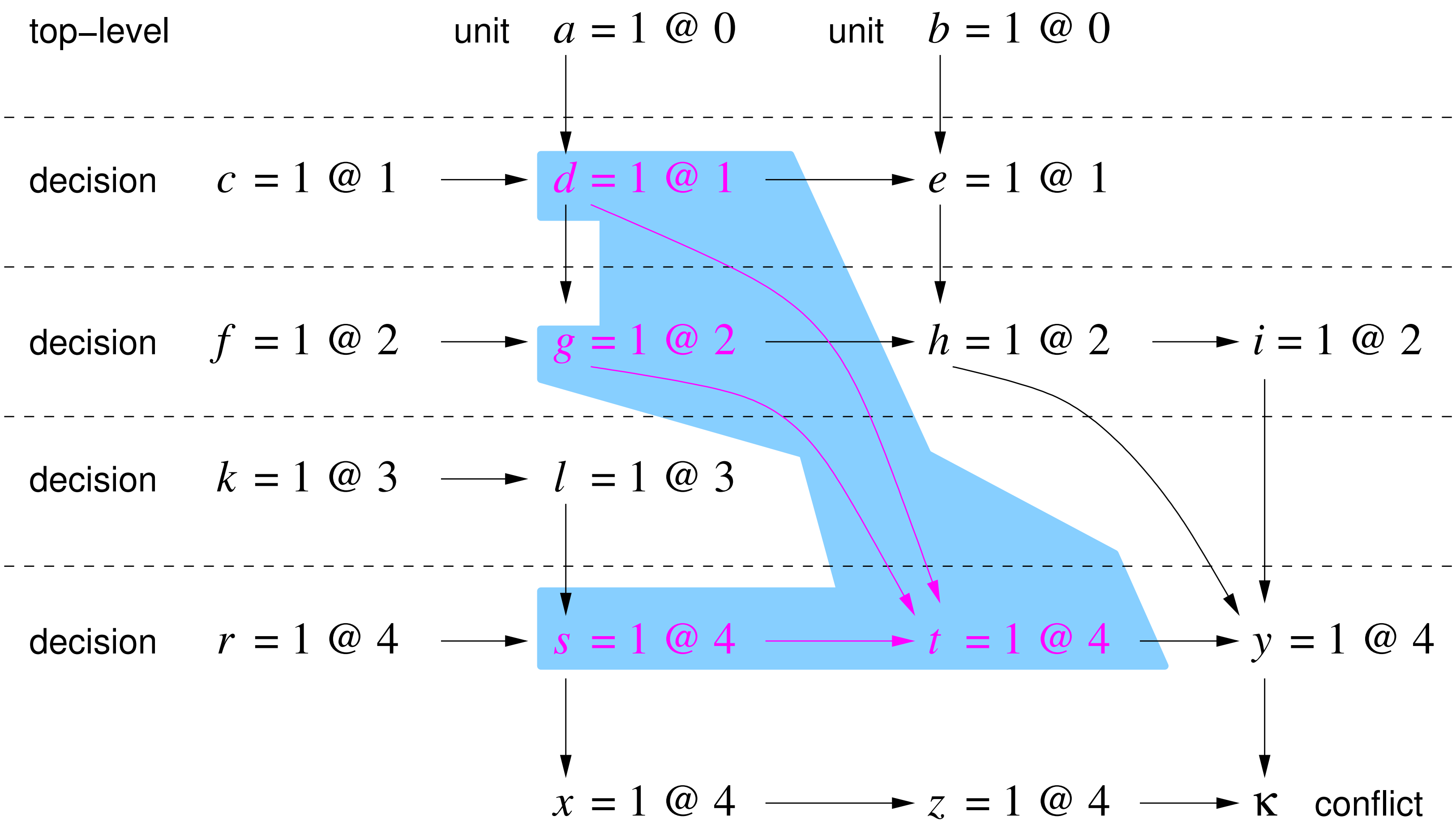
1st UIP learned clause increases chance of backjumping
("pulls in" as few decision levels as possible)

More Heuristics for Conflict Clauses Generation

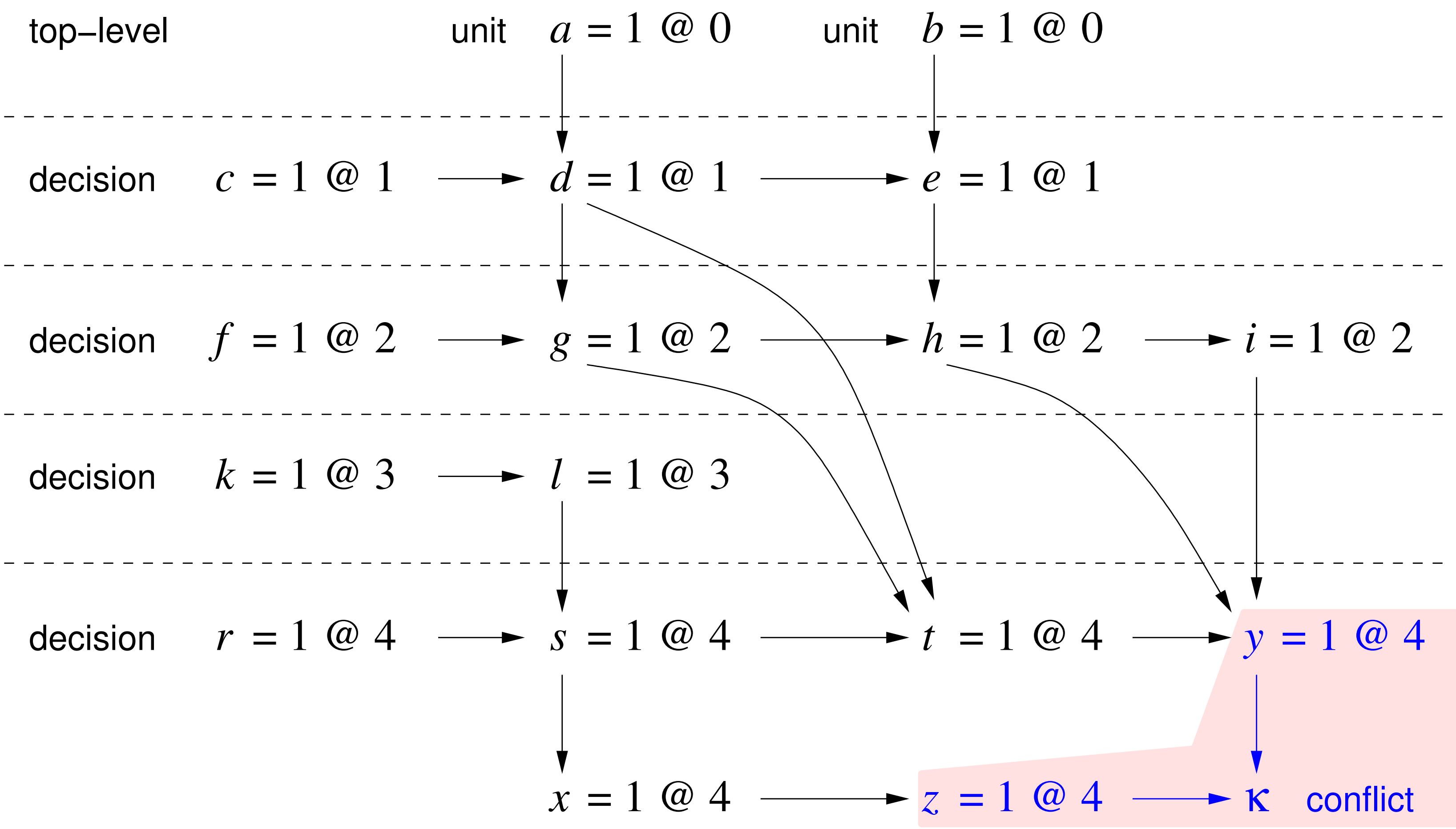
- intuitively is is important to localize the search (cf cutwidth heuristics)
- cuts for learned clauses may only include UIPs of current decision level
- on lower decision levels an arbitrary cut can be chosen
- multiple alternatives
  - include all the roots contributing to the conflict
  - find minimal cut (heuristically)
  - **cut off at first literal of lower decision level** (works best)

Implication Graph

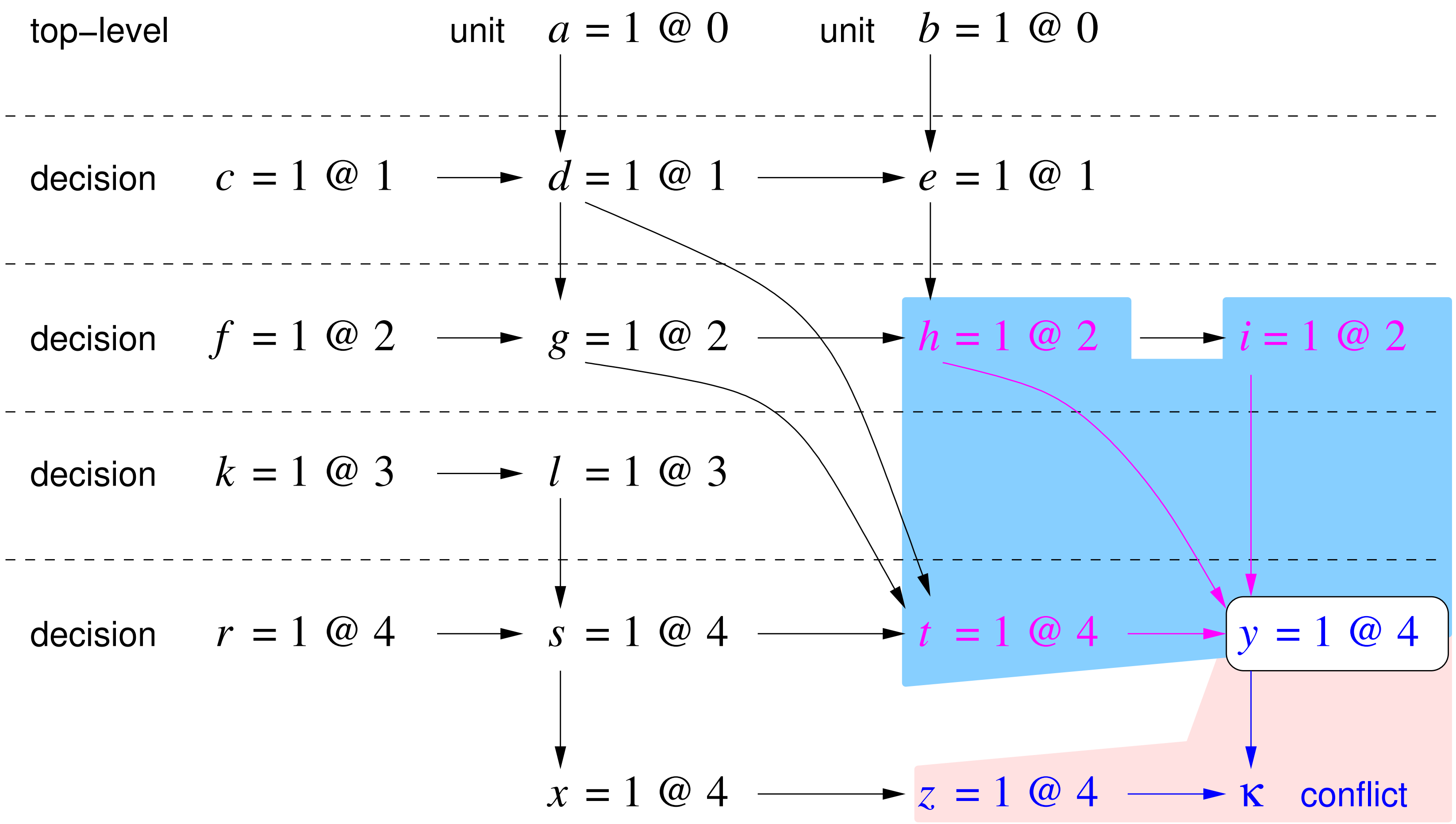




$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee t)$$

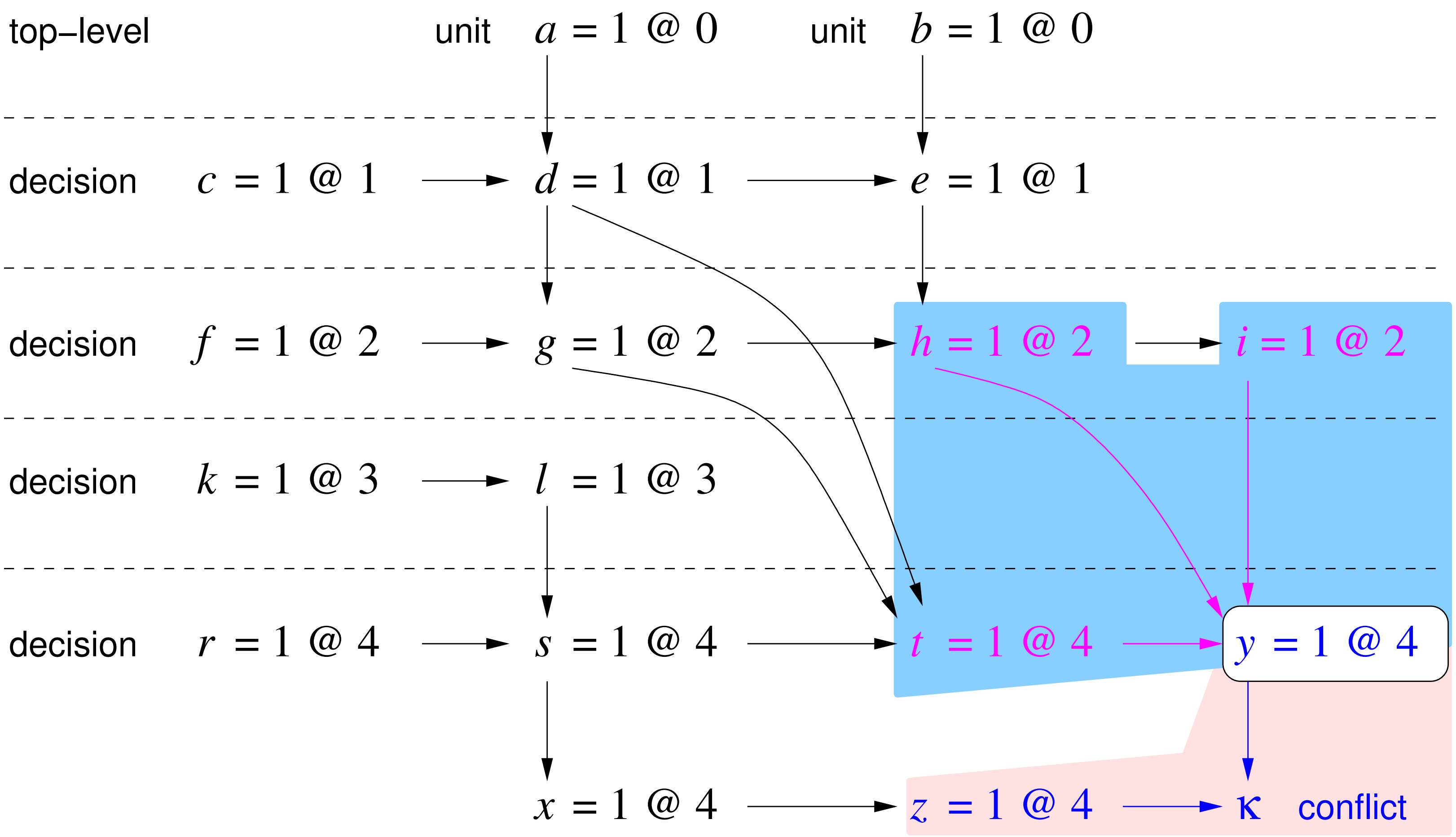


$$\neg(y \wedge z) \qquad \equiv \qquad (\overline{y} \vee \overline{z})$$



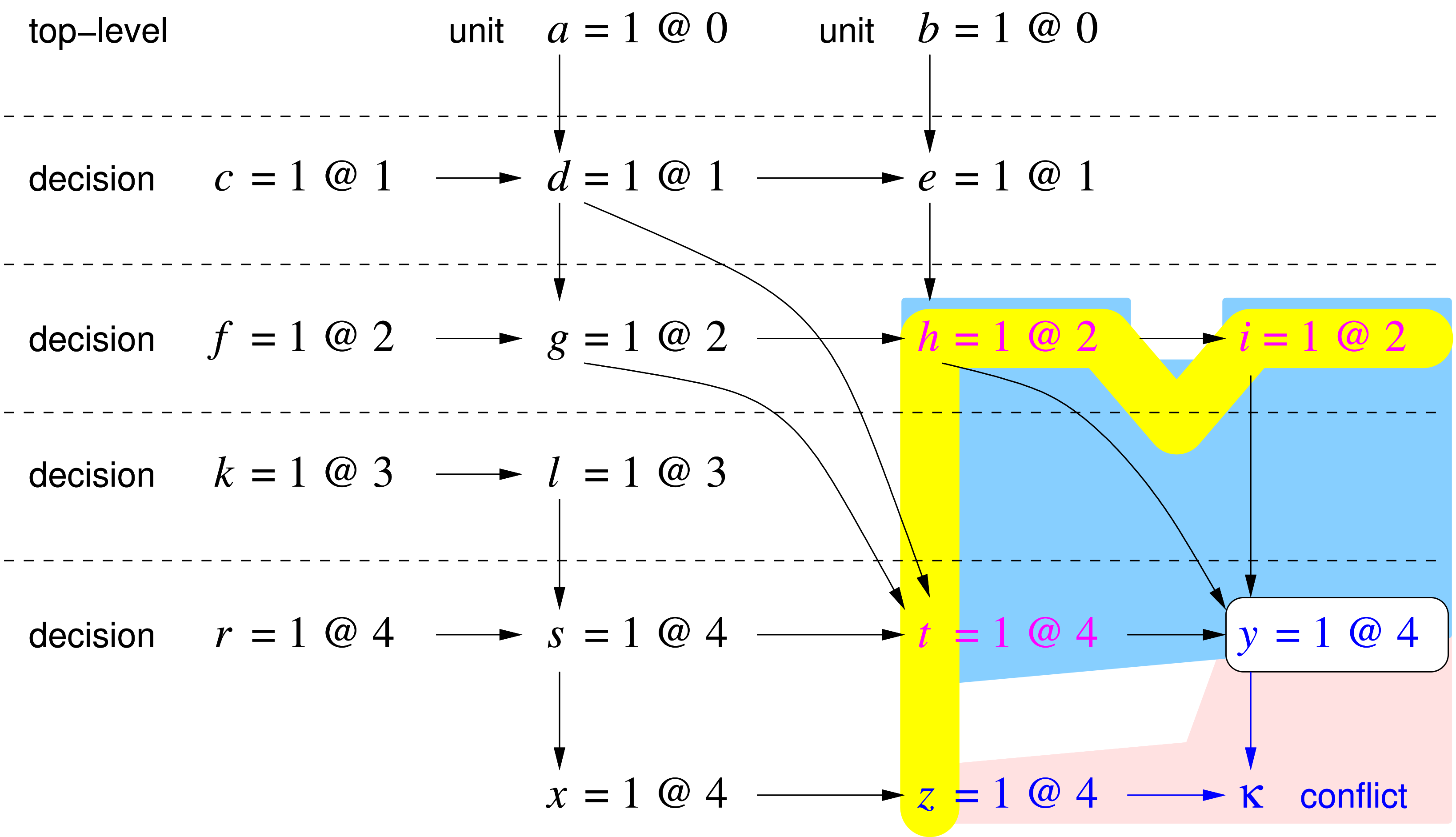
$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

Resolving Antecedents 1st Time



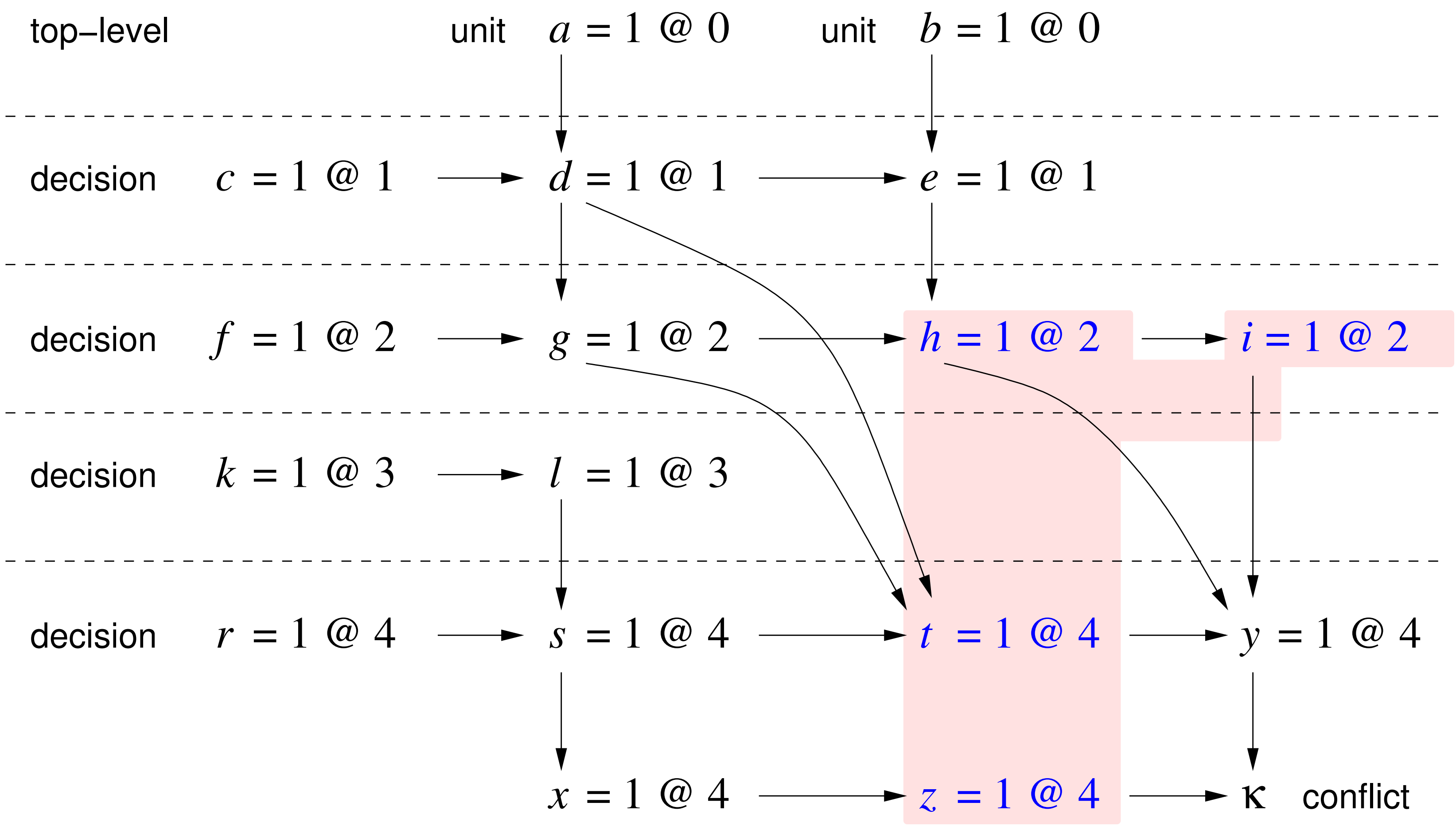
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{\boxed{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}}$$

Resolvents = Cuts = Potential Learned Clauses

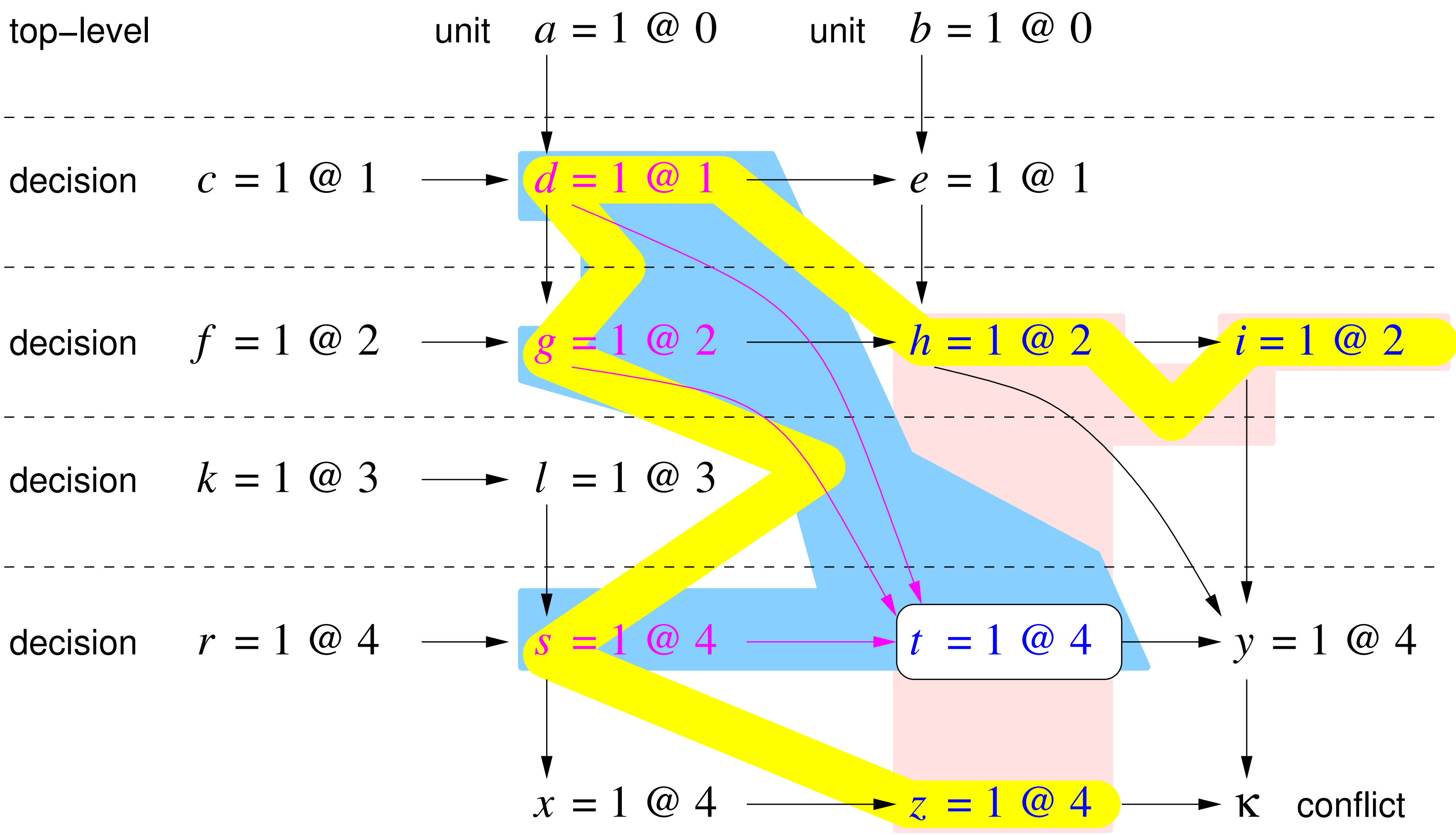


$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

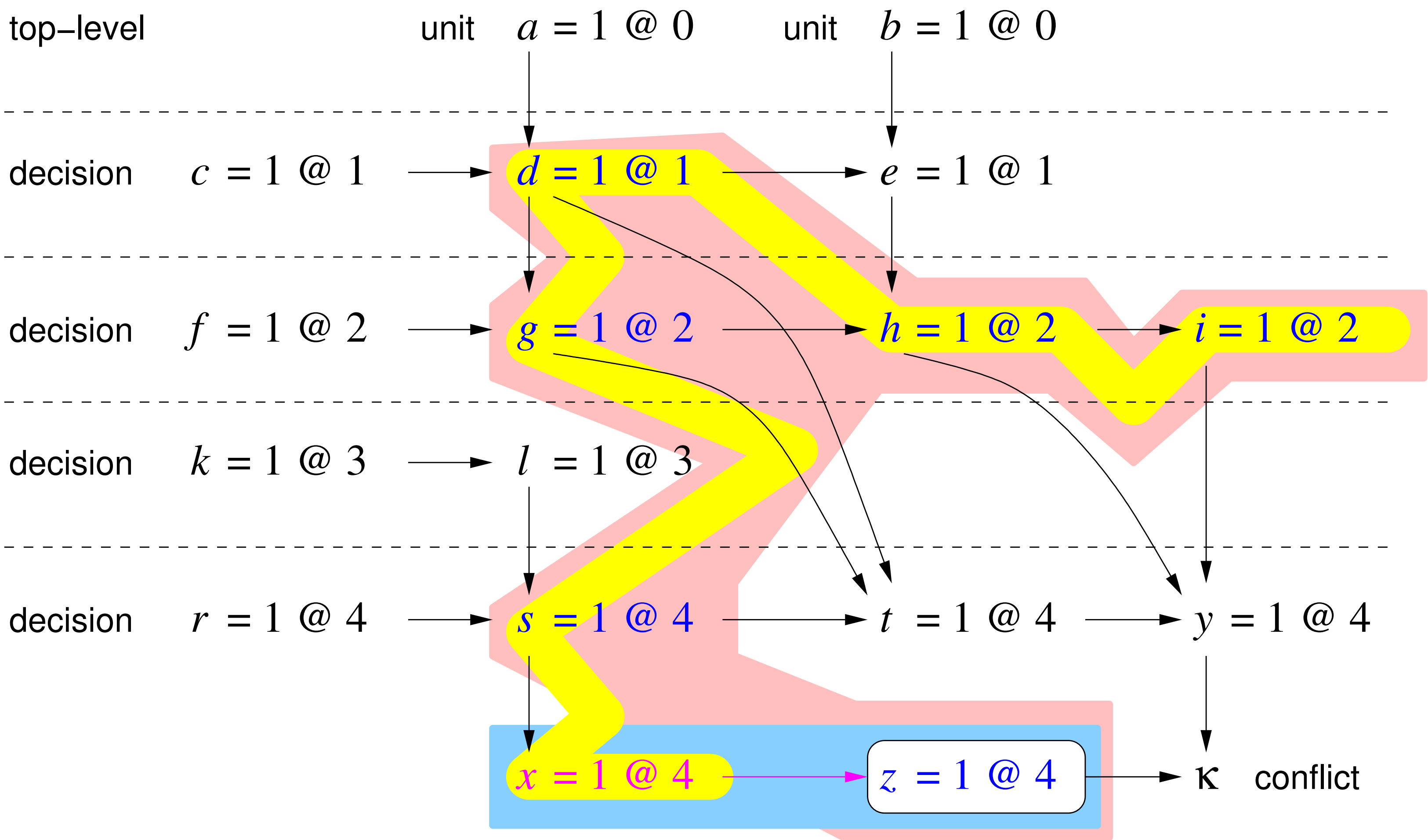
Potential Learned Clause After 1 Resolution



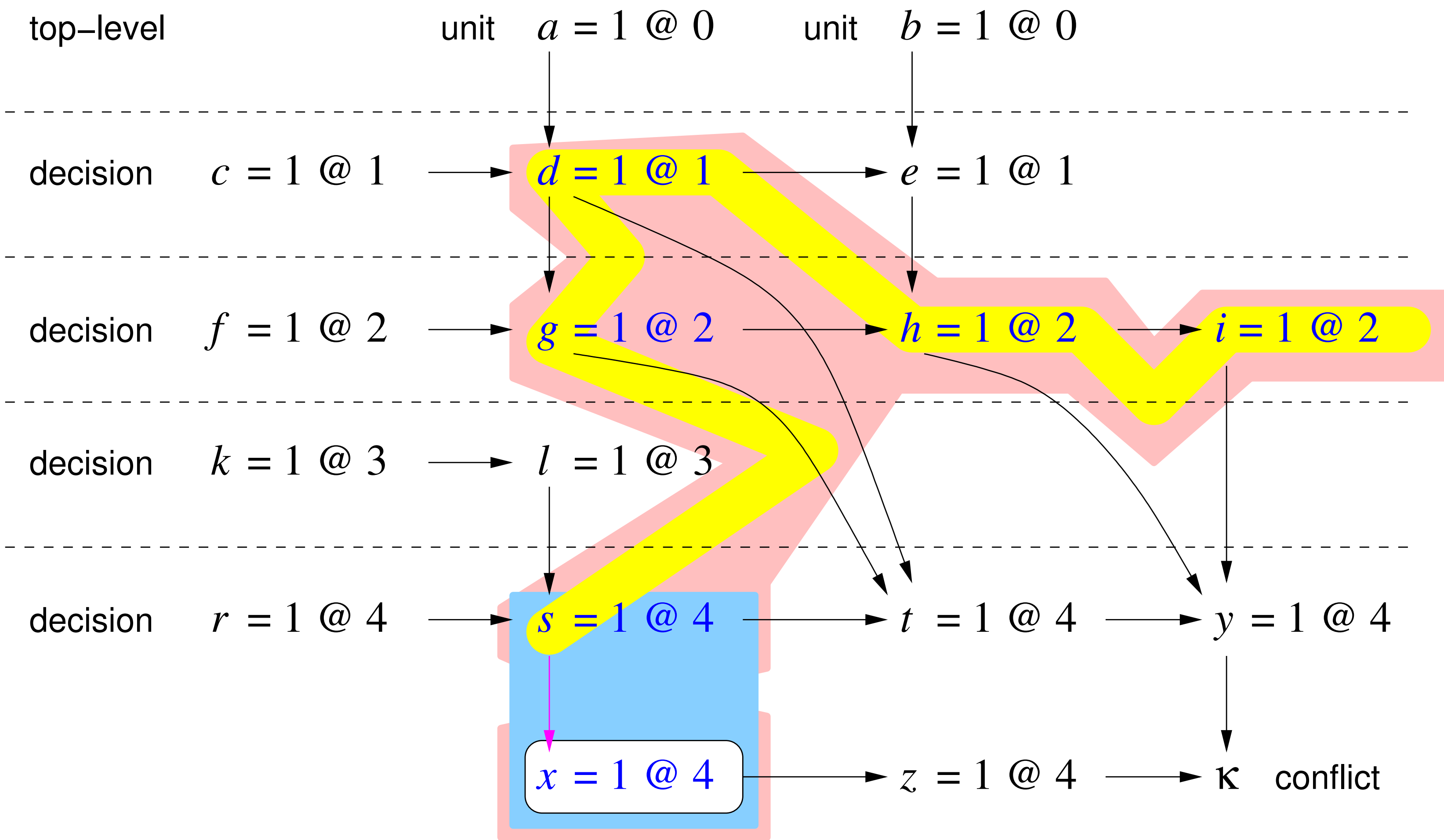
$$(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})$$



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

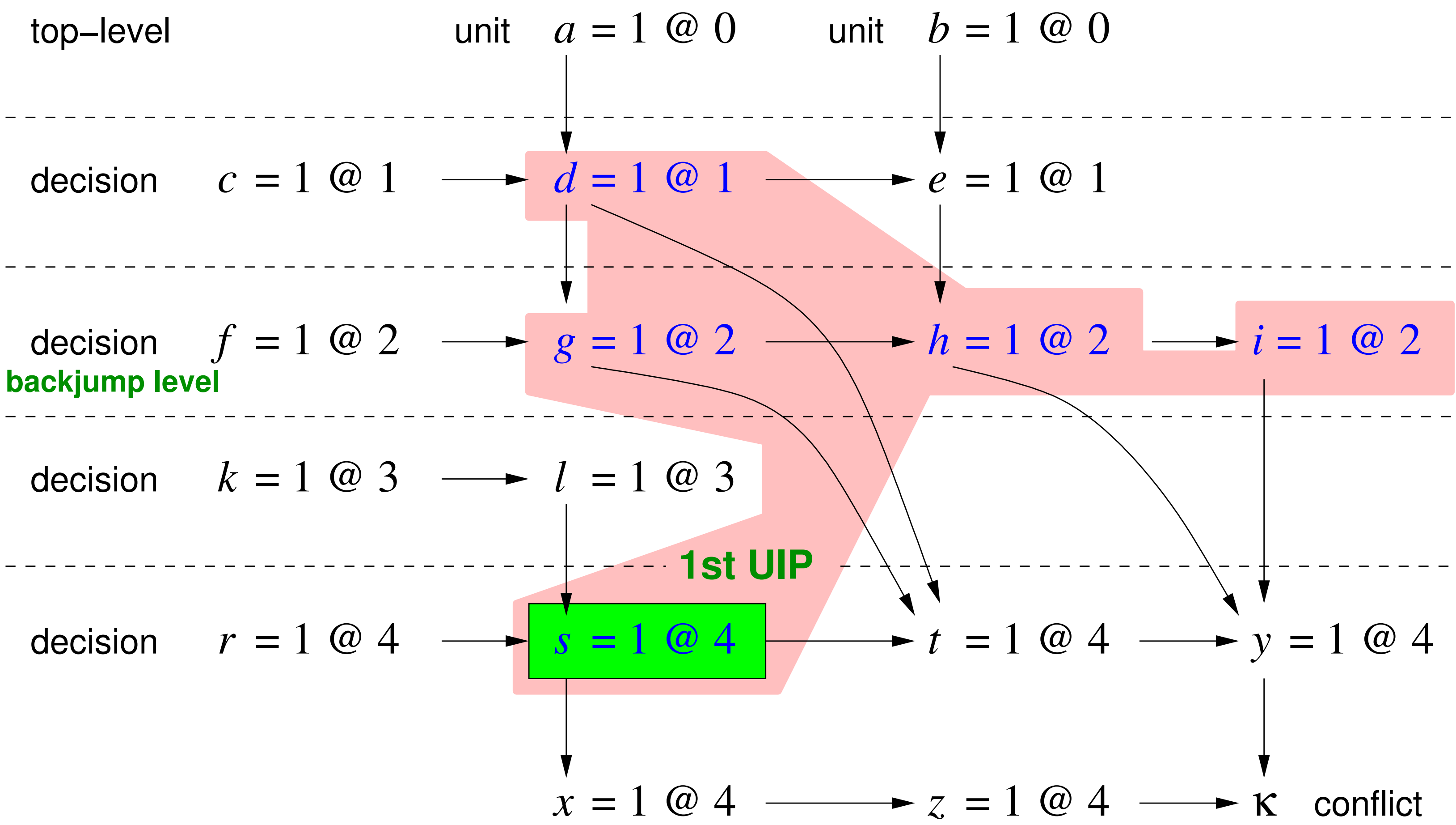


$$\frac{(\bar{x} \vee z) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}{(\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$



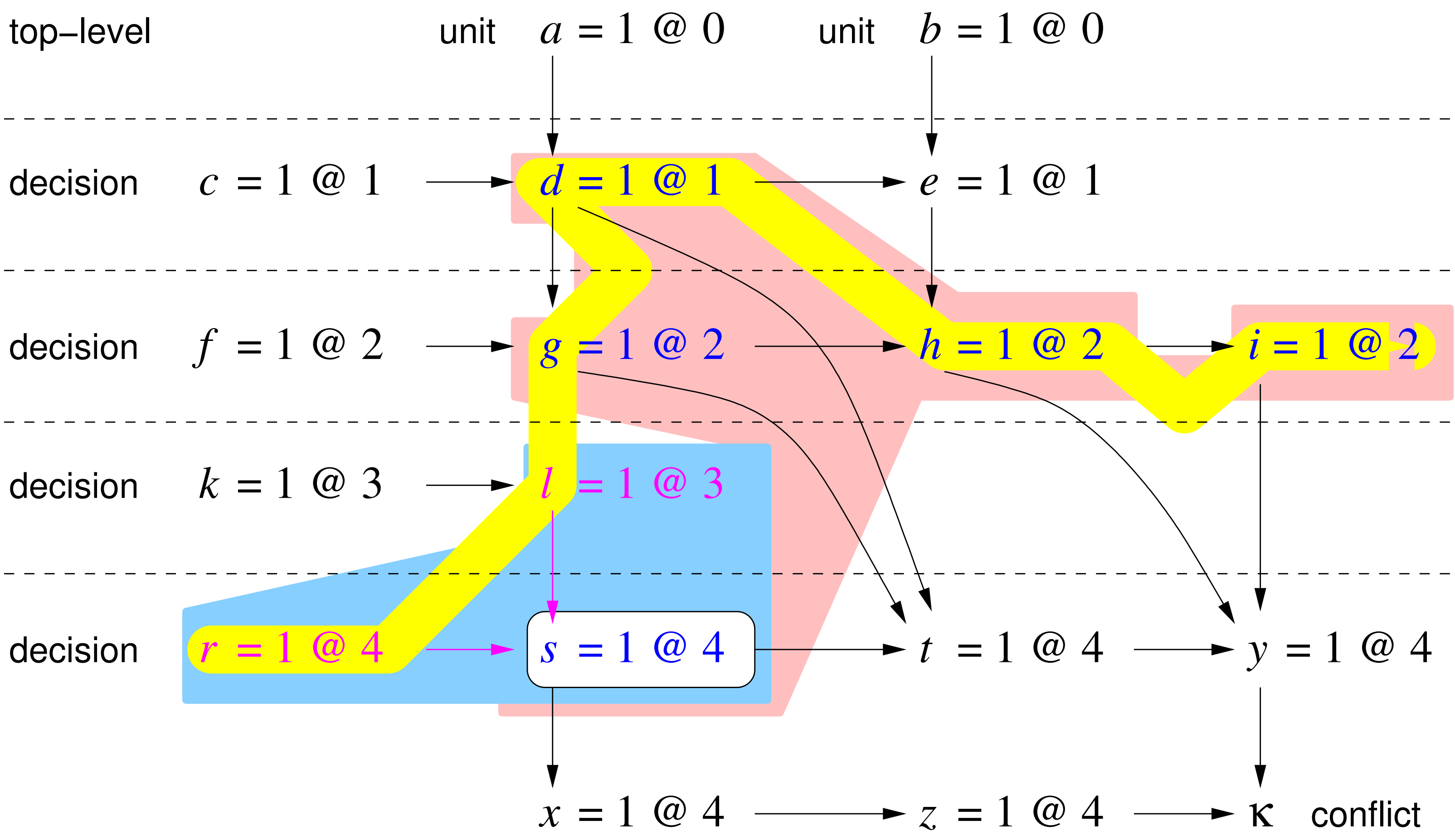
$$\frac{(\bar{s} \vee x) \qquad (\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

self subsuming resolution

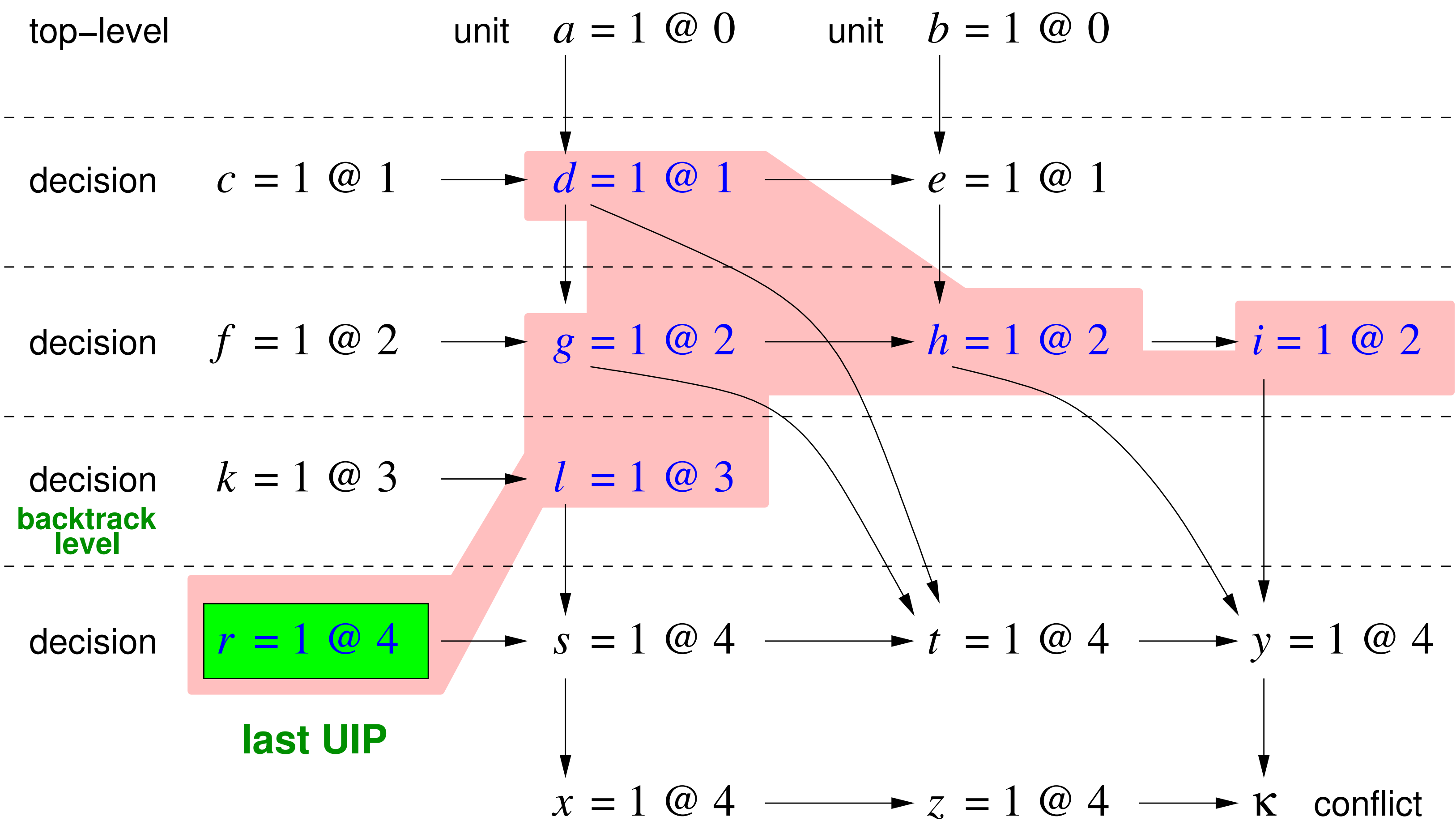


$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

Resolving Antecedents 5th Time

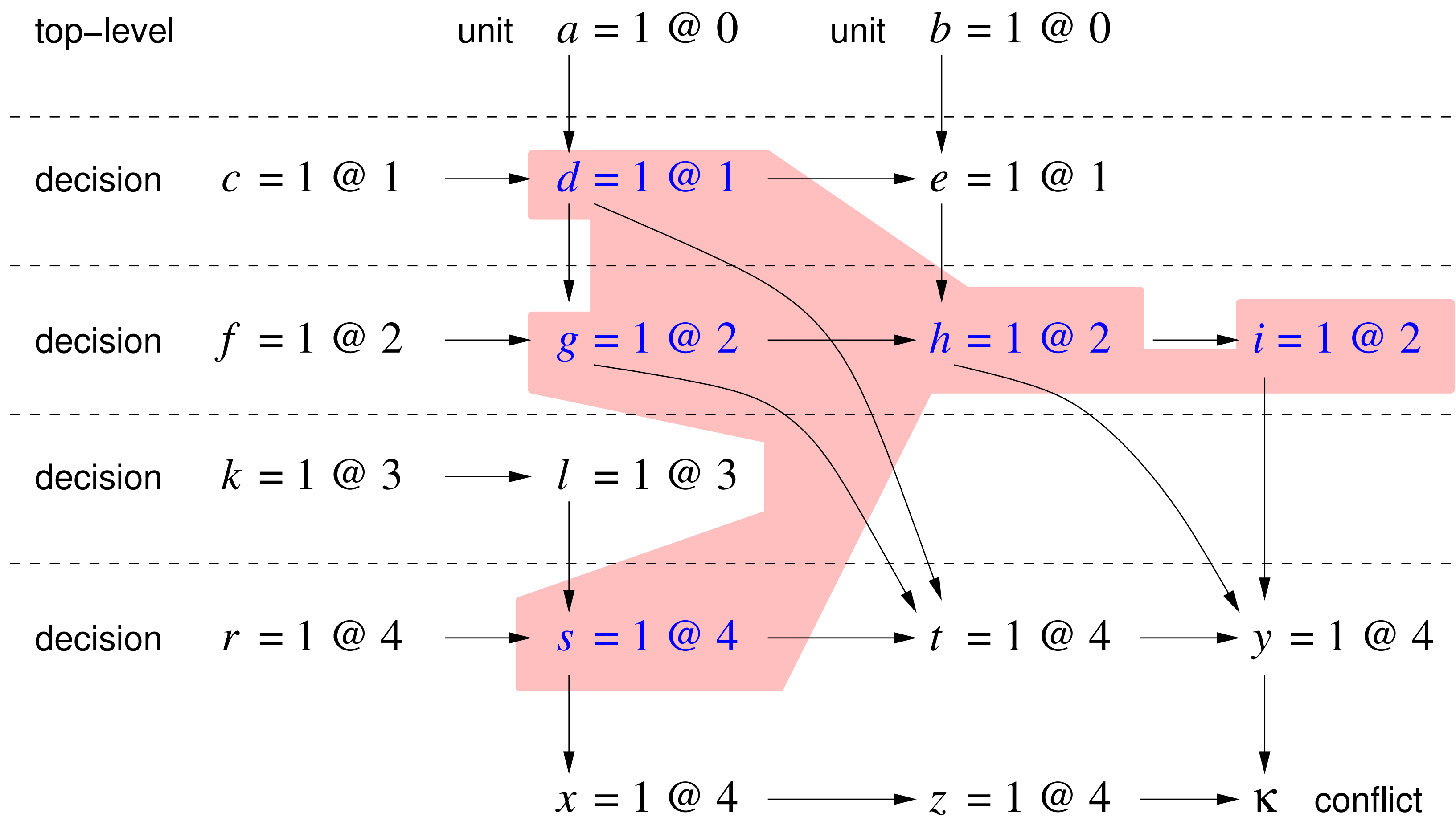


$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

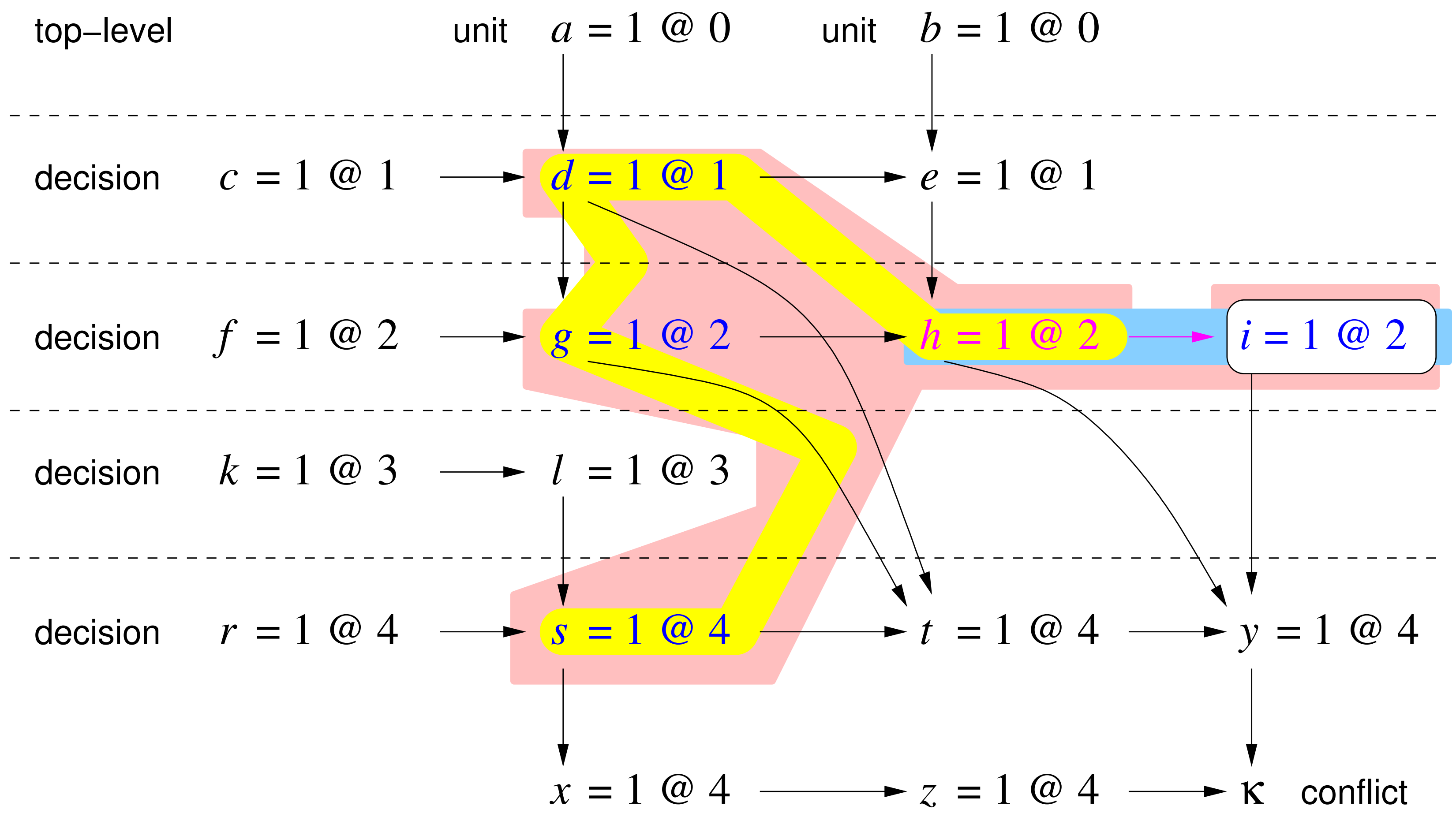


$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

1st UIP Clause after 4 Resolutions



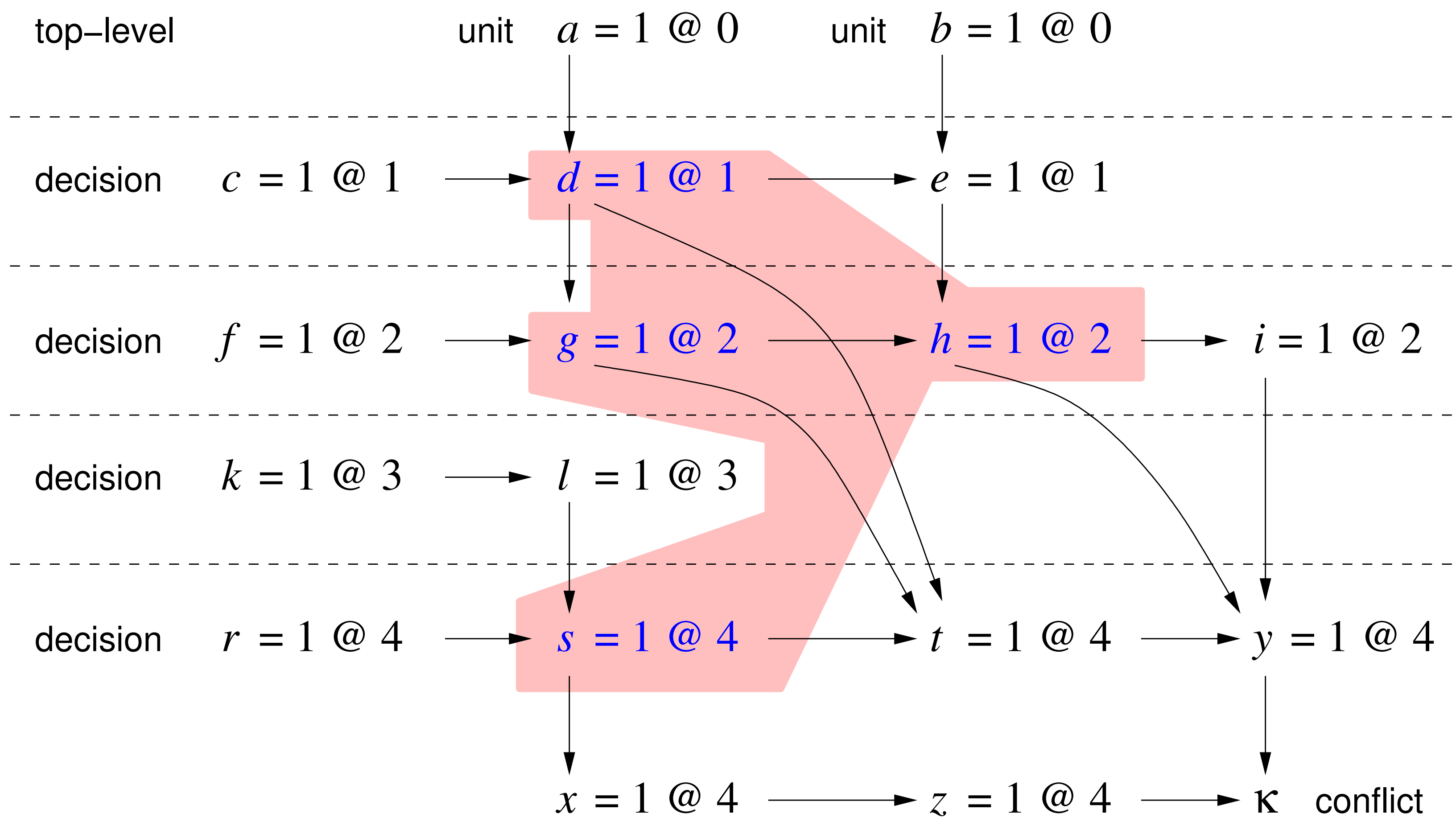
$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$



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

self subsuming resolution

Locally Minimized Learned Clause



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

Two step algorithm:

1. mark all variables in 1st UIP clause

2. remove literals with all antecedent literals also marked

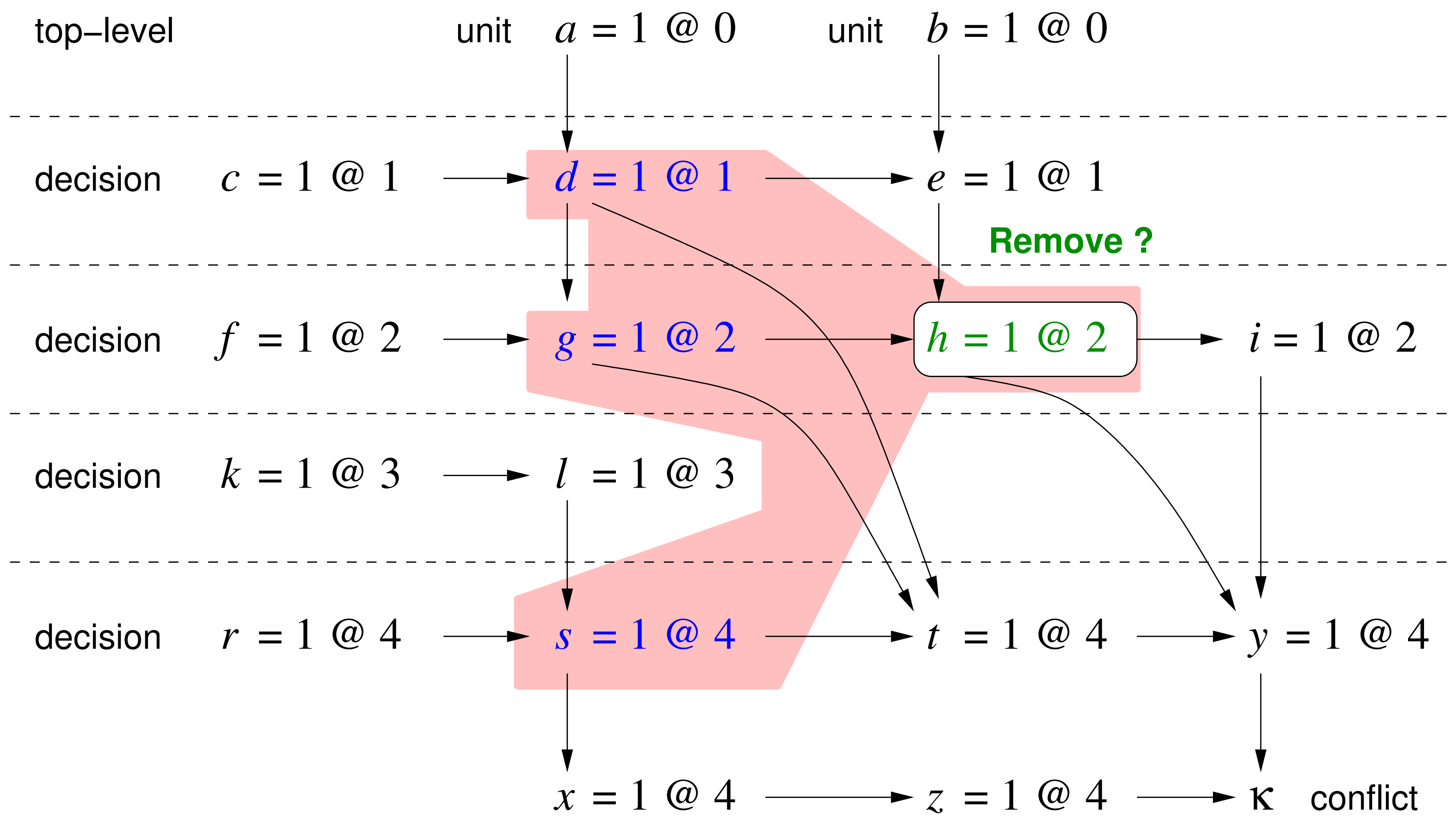
Correctness:

- removal of literals in step 2 are self subsuming resolution steps.

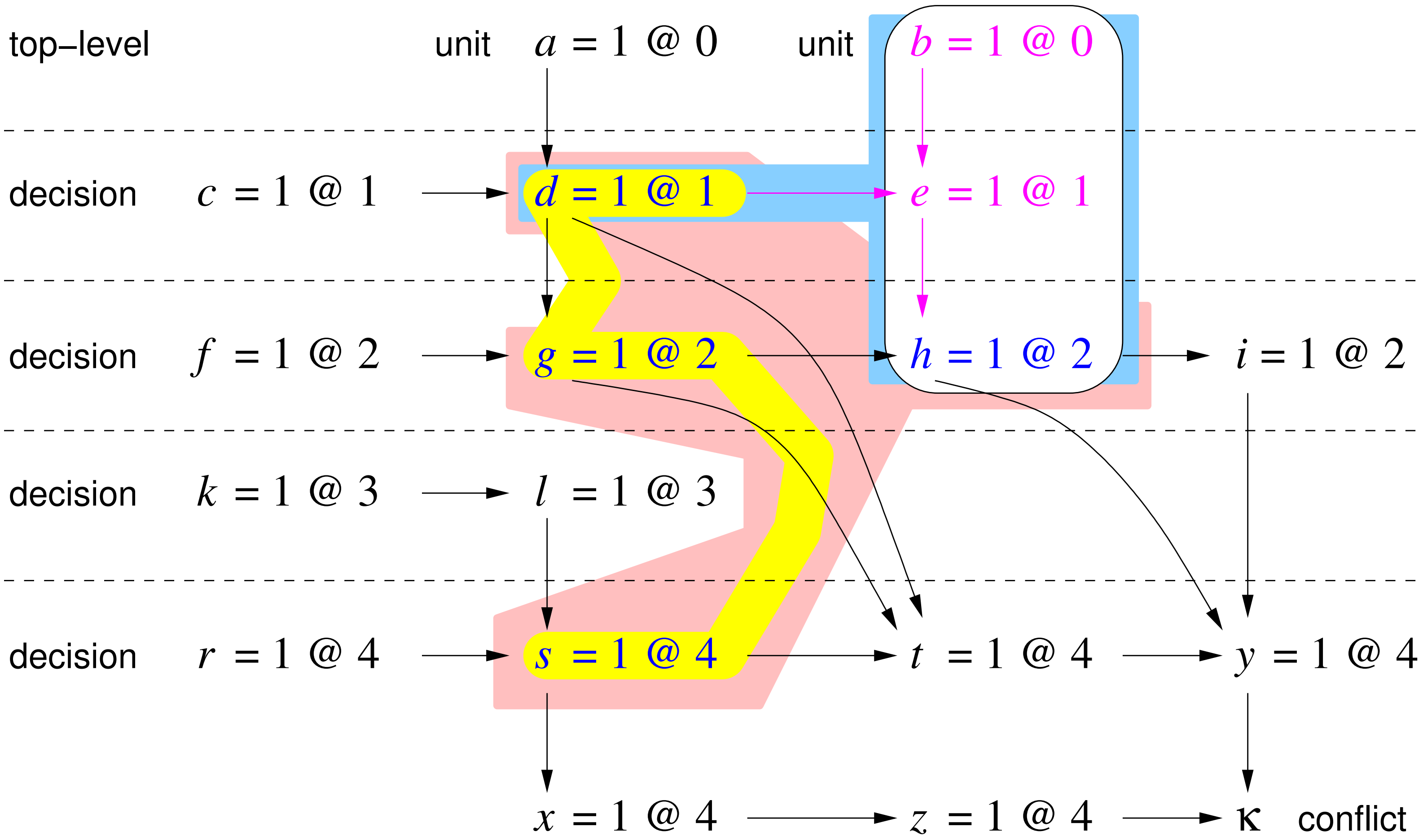
- implication graph is acyclic.

Confluence: produces a unique result.

Minimizing Locally Minimized Learned Clause Further?

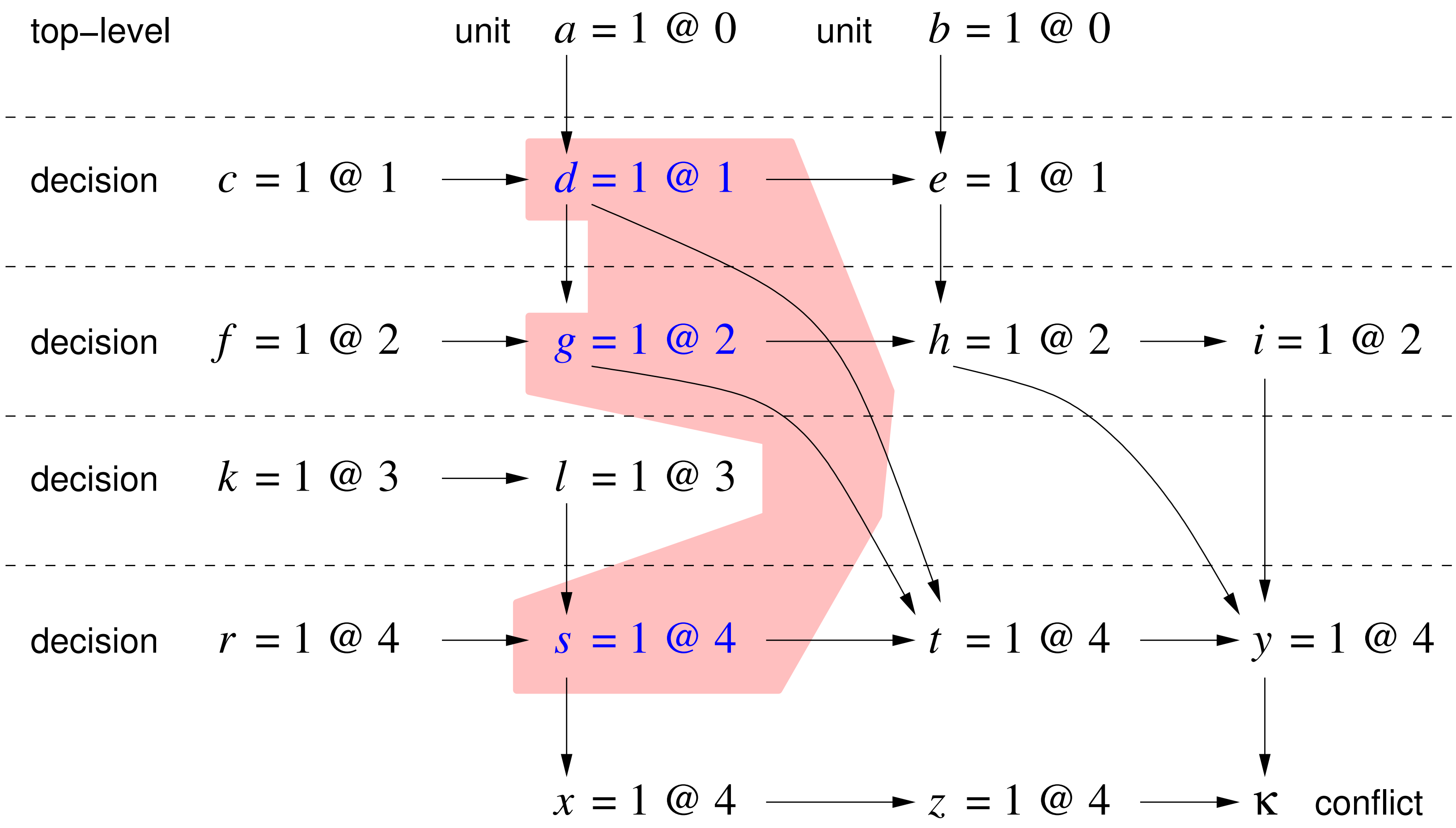


$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$



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

Recursively Minimized Learned Clause



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

Recursive Minimization Algorithm

[MiniSAT 1.13]

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”

		solved instances		time in hours		space in GB		out of memory		deleted literals
MINISAT with preprocessing	recur	788	9%	170	11%	198	49%	11	89%	33%
	local	774	7%	177	8%	298	24%	72	30%	16%
	none	726		192		392		103		
MINISAT without preprocessing	recur	705	13%	222	8%	232	59%	11	94%	37%
	local	642	3%	237	2%	429	24%	145	26%	15%
	none	623		242		565		196		
PICOSAT with preprocessing	recur	767	10%	182	13%	144	45%	10	60%	31%
	local	745	6%	190	9%	188	29%	10	60%	15%
	none	700		209		263		25		
PICOSAT without preprocessing	recur	690	6%	221	8%	105	63%	10	68%	33%
	local	679	5%	230	5%	194	31%	10	68%	14%
	none	649		241		281		31		
altogether	recur	2950	9%	795	10%	679	55%	42	88%	34%
	local	2840	5%	834	6%	1109	26%	237	33%	15%
	none	2698		884		1501		355		

10 runs for each configuration with 10 seeds for random number generator

		MiniSat with preprocessing					
		seed	solved	time	space	mo	del
1.	recur	8	82	16	19	1	33%
2.	recur	6	81	17	20	1	33%
3.	local	0	81	16	29	7	16%
4.	local	7	80	17	29	8	15%
5.	recur	4	80	17	20	1	33%
6.	recur	1	79	17	20	1	33%
7.	recur	9	79	17	20	1	34%
8.	local	5	78	18	29	7	16%
9.	local	1	78	17	29	6	16%
10.	recur	0	78	17	20	1	34%
11.	recur	5	78	17	19	1	33%
12.	local	3	77	18	31	7	16%
13.	local	8	77	18	30	8	16%
14.	recur	7	77	17	20	1	34%
15.	recur	3	77	17	20	1	34%
16.	recur	2	77	17	20	2	33%
17.	none	7	76	19	39	9	0%
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Second Order Dynamic Decision Heuristics: VSIDS

[MoskewiczMadiganZhaoZhangMalik-DAC'01: CHAFF]

- "second order" because it involves statistics about the search
- Variable State Independent Decaying Sum (VSIDS) decision heuristic
  (implemented in Chaff, Limmat, MiniSAT, PicoSAT, and many more)
- VSIDS just counts the occurrences of a literals in conflict clauses
- literal/variable with maximal count (score) is chosen
  (from a priority queue ordered by score)
- score is multiple by a factor $f < 1$ after a certain number of conflicts occurred
  (this is the "decaying" part and also called *rescoring*)
- emphasizes (negation of) literals contributing recently to conflicts (**localization**)

[Biere-SAT'08]

- VSIDS score can be normalized to the interval [0,1] as follows:
  - pick a decay factor f per conflict: typically $f = 0.95$
  - each variable is punished by this decay factor at every conflict
  - if a variable is involved in conflict, add $1 - f$ to score

    $$s, f \leq 1, \quad \text{then} \quad s' \leq \overbrace{s \cdot f}^{\text{decay in any case}} + \underbrace{1 - f}_{\text{increment if involved}} \leq f + 1 - f = 1$$

    with s old score before conflict, s' new score after conflict
- recomputing score of all variables at each conflict is costly
  - linear in the number of variables, e.g. millions
  - particularly, because number of involved variabels $<<$ number of variables

- Chaff: precision of score traded for faster decay
  - increment score of involved variables by 1
  - decay score of all variables every 256 conflicts by halfing the score
  - sort priority queue after decay and not at every conflict

- MiniSAT uses Exponential VSIDS
  - also just update score of involved variables
  - 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 again only one variable with score sequence s_n resp. S_n

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

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

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

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

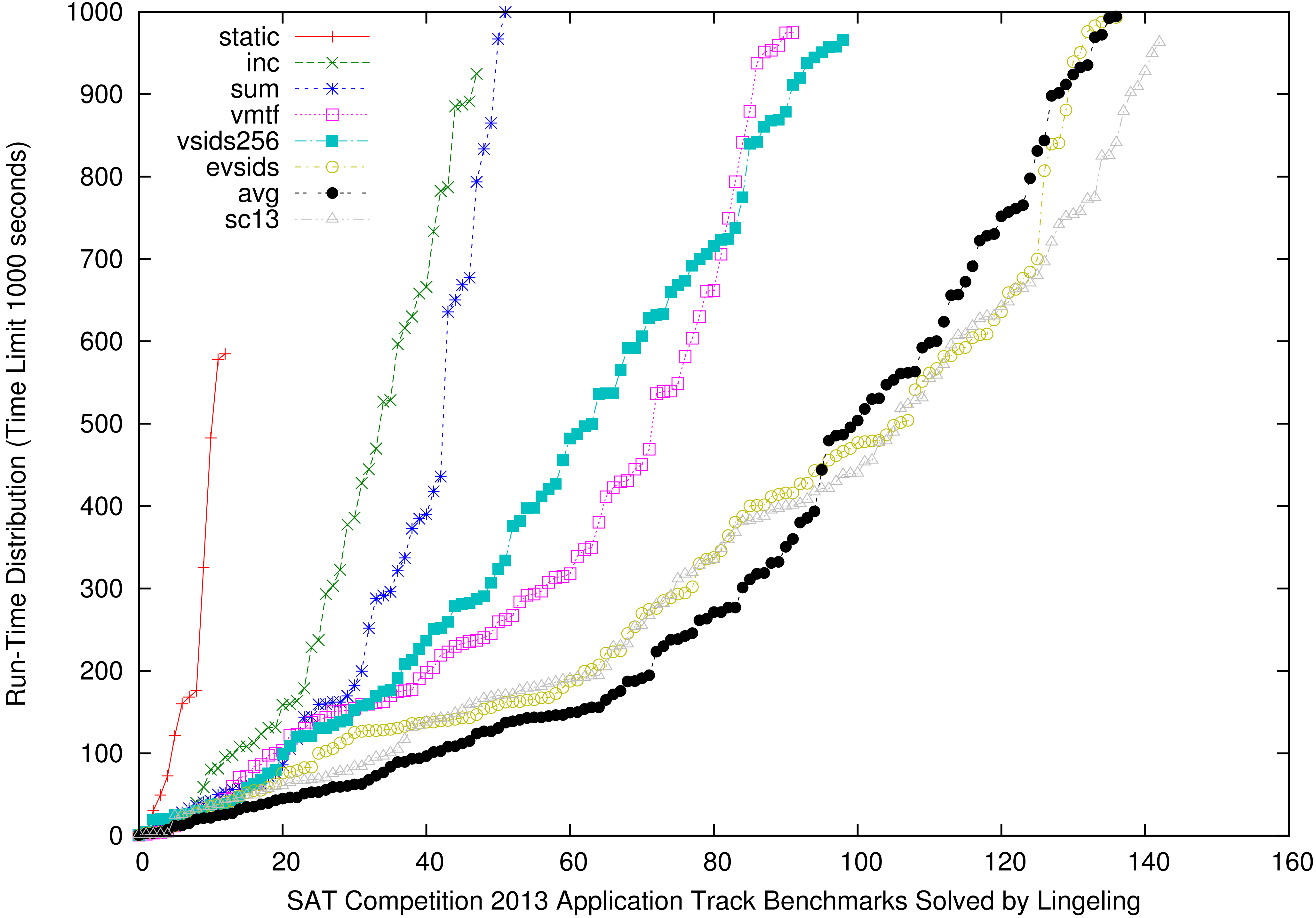
[GoldbergNovikov-DATE'02]

- 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 chose the decision variable for one clause
  - if all learned clauses are satisfied use other heuristics
  - intuitively obtains another order of localization (no proofs yet)
- results are mixed (by some authors considered to be more robust than just VSIDS)

- *variable move to front strategy* (VMTF)
  - Siege SAT Solver [Ryan'04]
  - easy and cheap to implement with doubly linked list
    - need pointer to last picked variable in queue
    - reset during back-tracking
  - rather aggressive
- *clause move to front strategy* (CMTF)
  - HaifaSAT [GershanStrichman'08] variant keeps clauses in a queue
  - queue can also be used to find less important clauses to throw away
  - refined version in PrecoSAT [Biere'09] (multiple queues per glucose level

How to Compute the Score?

- SAT solver picks unassigned variable with largest score as next decision
  - consider only change of the score s_i of one variable v during i-th conflict
  - let $\beta_i = 1$ if v is *bumped* in the i-th conflict otherwise 0
- some possible variable score update functions:
  - **static** $s_{i+1} = s_i$ initialize score statically and do not change it
  - **inc** $s_{i+1} = s_i + \beta_i$ this is in essence DLIS from Grasp
  - **vmtf** $s_{i+1} = i$
  - **sum** $s_{i+1} = s_i + i \cdot \beta_i$ emphasis on recent conflicts unpublished
  - **vsids** $s_{i+1} = d \cdot s_i + \beta_i$ decay $d \in [0,1)$ e.g. $d = 0.95$
  - **evsids** $s_{i+1} = s_i + g_i \cdot \beta_i, \quad g_{i+1} = e \cdot g_i$ factor $e \in [1,2)$ e.g. $e = 1.05$
  - **avg** $s_{i+1} = s_i + \beta_i \cdot (i - s_i)/2$ another filter function unpublished
- last four share the idea of “low-pass filtering” of the involvement of variables
  - for this interpretation see our SAT’08 paper and the video
  - important practical issue: number of bumped variables is usually small



static
inc
sum
vmtf
vsids256
evsids
avg
sc13
Run-Time Distribution (Time Limit 1000 seconds)
SAT Competition 2013 Application Track Benchmarks Solved by Lingeling
0
100
200
300
400
500
600
700
800
900
1000
0
20
40
60
80
100
120
140
160

- should not keep all learned clauses forever
  - some of them become useless
  - for instance subsumed or satisfied under learned units
  - were but are not anymore relevant to current search focus
  - memory consumption / BCP speed

- throw *unimportant* learned clauses away (reduce)
  - in regular intervals (controlled by geometric, Luby, arithmetic scheme)
  - *size* heuristics: discard long clauses
  - *least recently used* (LRU): as in HW cache (see also CMTF)
  - clause scores with bumping scheme as for VSDIS (BerkMin)
  - glucose level: number decision levels in learned clause
    called also LBD in original paper [AudemardLaurentSimon'09]

- 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

1. BCP over (in)equalities: $\dfrac{x = y \quad z = (x \oplus y)}{z = 0}$ $\quad \dfrac{x = 0 \quad z = (x \vee y)}{z = y}$ etc.

2. structural rules: $\dfrac{x = (a \vee b) \quad y = (a \vee b)}{x = y}$ etc.

3. test rule:
$$\dfrac{\begin{array}{ccc} \{x = 0\} \cup E & & \{x = 1\} \cup E \\ \Downarrow & & \Downarrow \\ E_0 \cup E & & E_1 \cup E \end{array}}{(E_0 \cap E_1) \cup E}$$

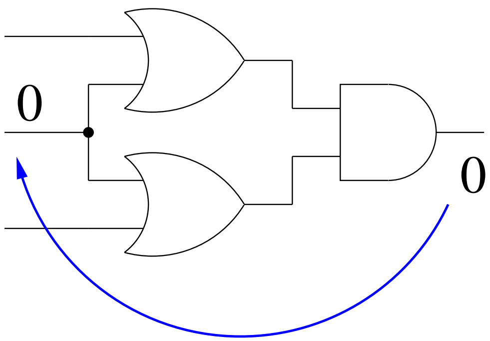
   Assume $x = 0$, BCP and derive (in)equalities E_0, then assume $x = 1$, BCP and derive (in)equalities E_1. The intersection of E_0 and E_1 contains the (in)equalities valid in *any* case.

$$
\begin{array}{cccc}
\multicolumn{2}{c}{x = 0} & \multicolumn{2}{c}{x = 1} \\
\multicolumn{2}{c}{\Downarrow} & \multicolumn{2}{c}{\Downarrow} \\
y = 0 & y = 1 & y = 0 & y = 1 \\
\Downarrow & \Downarrow & \Downarrow & \Downarrow \\
E_{00} & E_{01} & E_{10} & E_{11} \\
\hline
\multicolumn{2}{c}{E_0} & \multicolumn{2}{c}{E_1} \\
\hline
\multicolumn{4}{c}{E}
\end{array}
$$

(we do not show the (in)equalities that do not change)

- recursive application
  - depth of recursion bounded by number of variables
  - complete procedures (determines satisfiability or unsatisfiability)
  - for a fixed (constant) recursion depth k polynomial!
- k-saturation:
  - apply split rule on recursively up to depth k on all variables
  - 0-saturation: apply all rules accept test rule (just BCP: linear)
  - 1-saturation: apply test rule (not recursively) for all variables (until no new (in)equalities can be derived)

- circuits



output 0 implies middle input 0 indirectly

- CNF
  - for each clause c in the CNF
    - for each literal l in the clause c
      - assume l and propagate
      - collect set of all implied literals (direct/indirect “implications” of l)
    - intersect these sets of implied literals over all l in c
    - literals in the intersection are implied without any assumption

Variable Elimination

[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

- strenthen 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
- checking new clauses against existing clauses: backward (self) subsumption
  - new clause (self) subsumes existing clause
  - new clause smaller or equal in size
- check clause being subsumed by existing clauses forward (self) subsumption
  - can be made more efficient by one-watcher scheme [Zhang-SAT'05]

Variable Instantiation

[AnderssonBjesseCookHanna DAC'02] also in Oepir SAT solver, this is our reformulation

- for all iterals l
  - for all clauses c in which l occurs (with this particular phase)
    - assume the negation of all the other literals in c, assume l
    - if BCP does not lead to a conflict continue with next literal in outer loop
  - if all clauses produced a conflict permanently assign $\neg l$

Correctness: Let $c = l \vee d$, assume $\neg d \wedge l$.

If this leads to a conflict $d \vee \neg l$ could be learned (but is not added to the CNF).

Self subsuming resolution with c results in d and c is removed.

If all such cases lead to a conflict, $\neg l$ becomes a pure literal.

Generalization of pure literals.

Given a partial assignment σ.

A clause of a CNF is “touched” by σ if it contains a literal assigned by σ.

A clause of a CNF is “satisfied” by σ if it contains a literal assigned to true by σ.

If all touched clauses are satisfied then σ is an “autarky”.

All clauses touched by an autarky can be removed.

Example: $(-1\ 2)(-1\ 3)(1\ \ -2\ \ -3)(2\ 5)\cdots$ (more clauses without 1 and 3).

Then $\sigma = \{-1, -3\}$ is an autarky.

[Kullman'99]

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

fix a CNF F

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

$(a \vee b \vee l)$

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

since all resolvents of C on l are tautological C can be removed

Proof

assignment σ satisfying $F \backslash C$ but not C

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

[JärvisaloBiereHeule-TACAS'10]

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

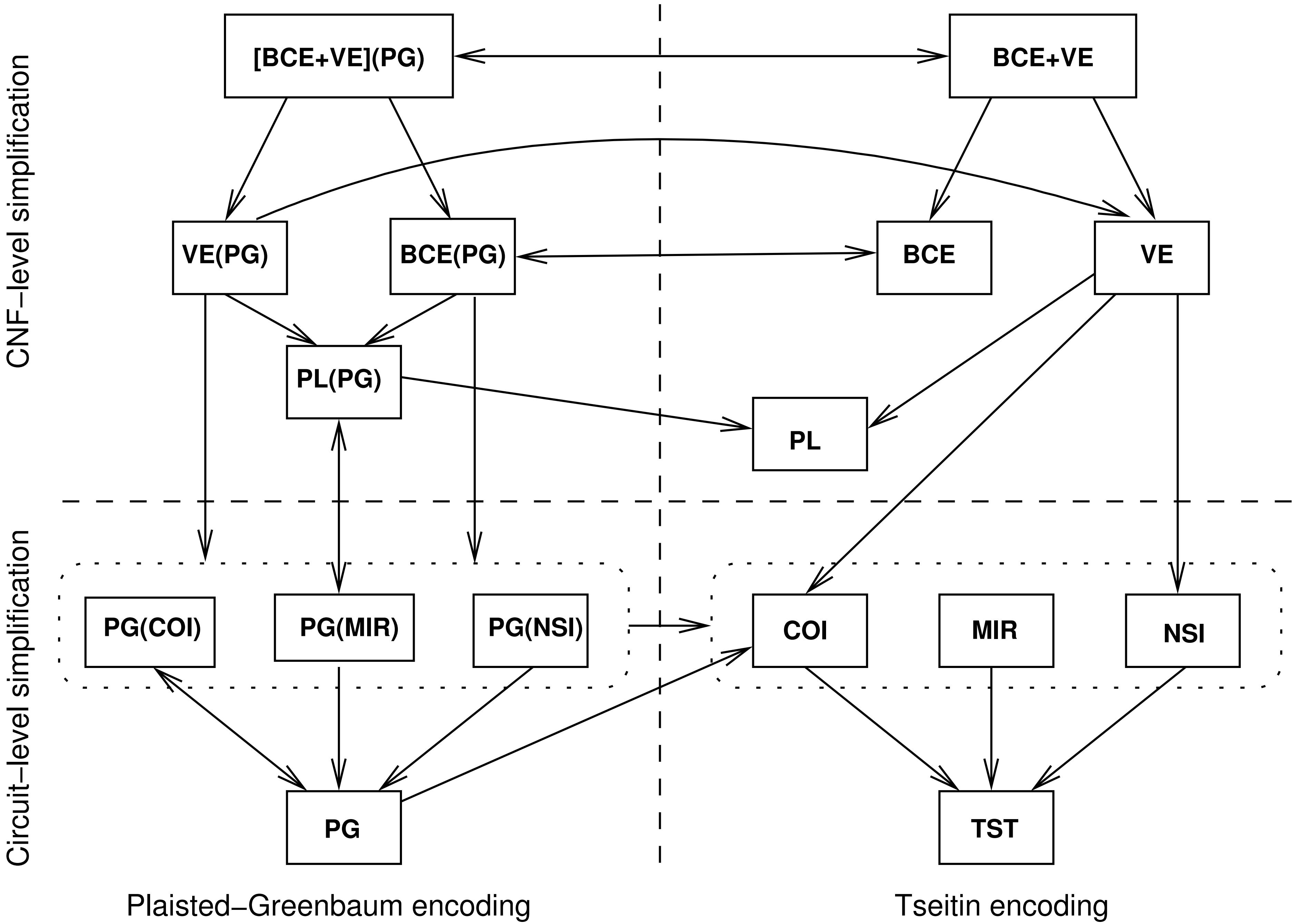
NSI Non-Shared Inputs reduction

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

VE Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination

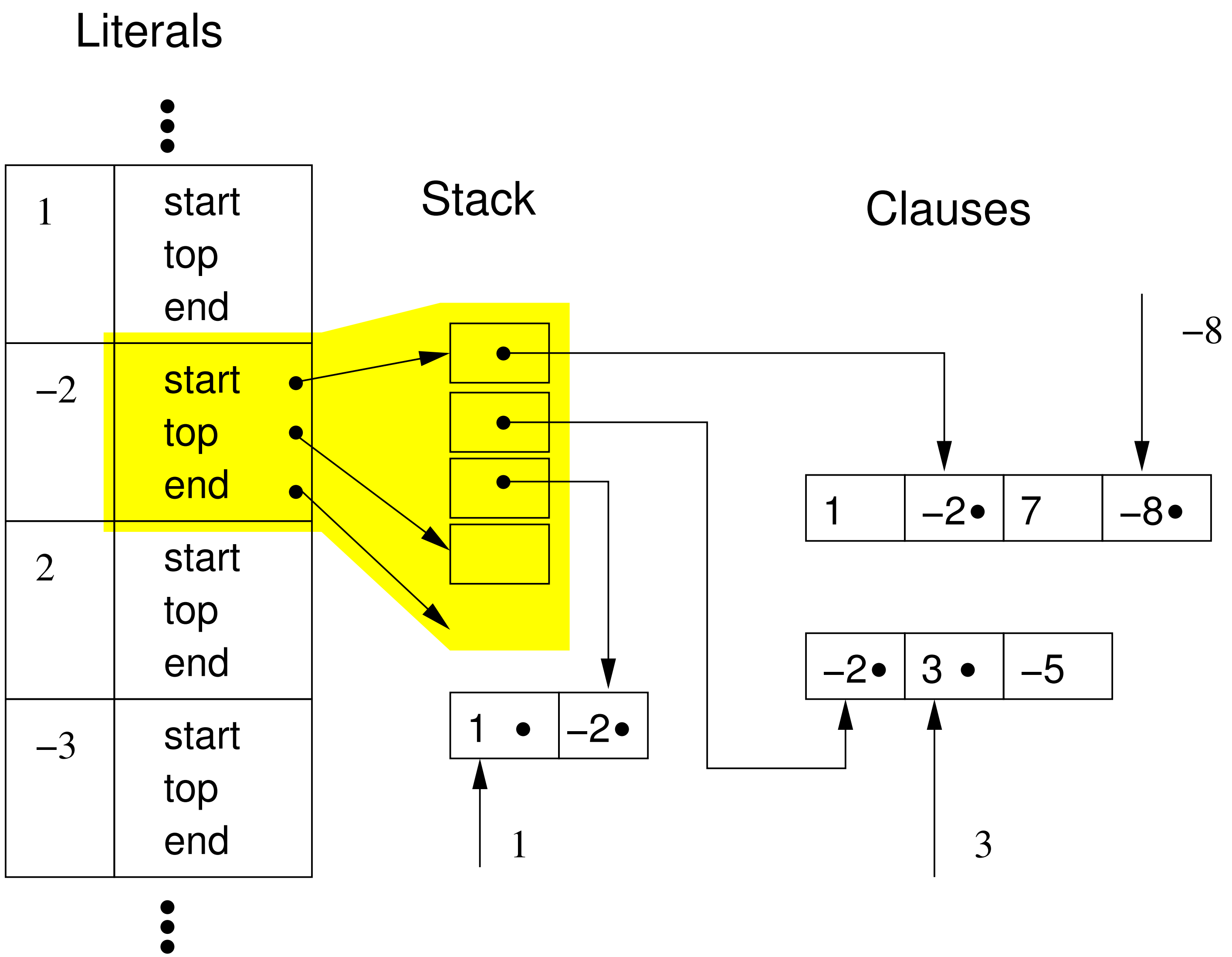


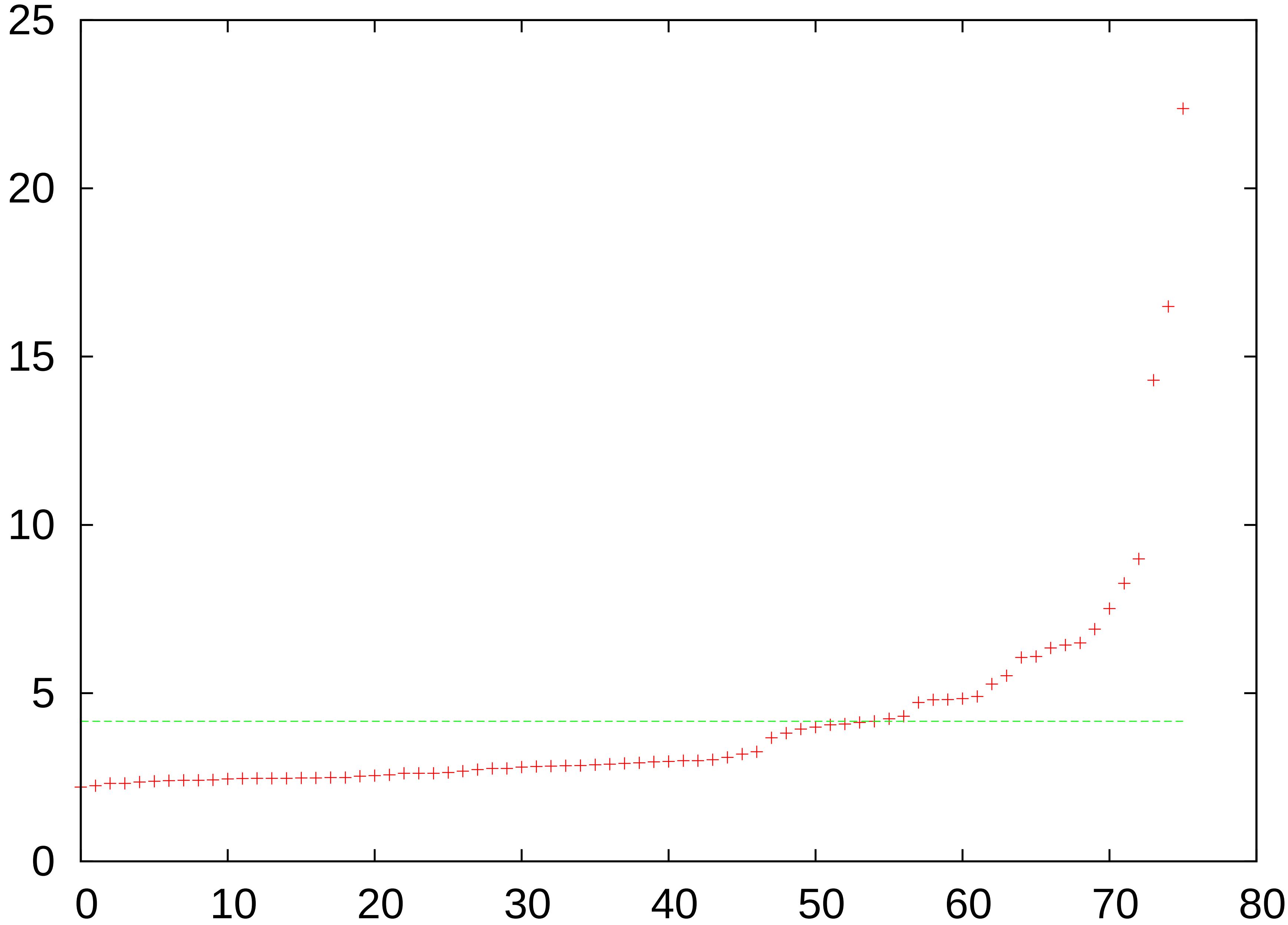
[BCE+VE](PG)
BCE+VE
VE(PG)
BCE(PG)
BCE
VE
PL(PG)
PL
PG(COI)
PG(MIR)
PG(NSI)
COI
MIR
NSI
PG
TST
CNF−level simplification
Circuit−level simplification
Plaisted−Greenbaum encoding
Tseitin encoding

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

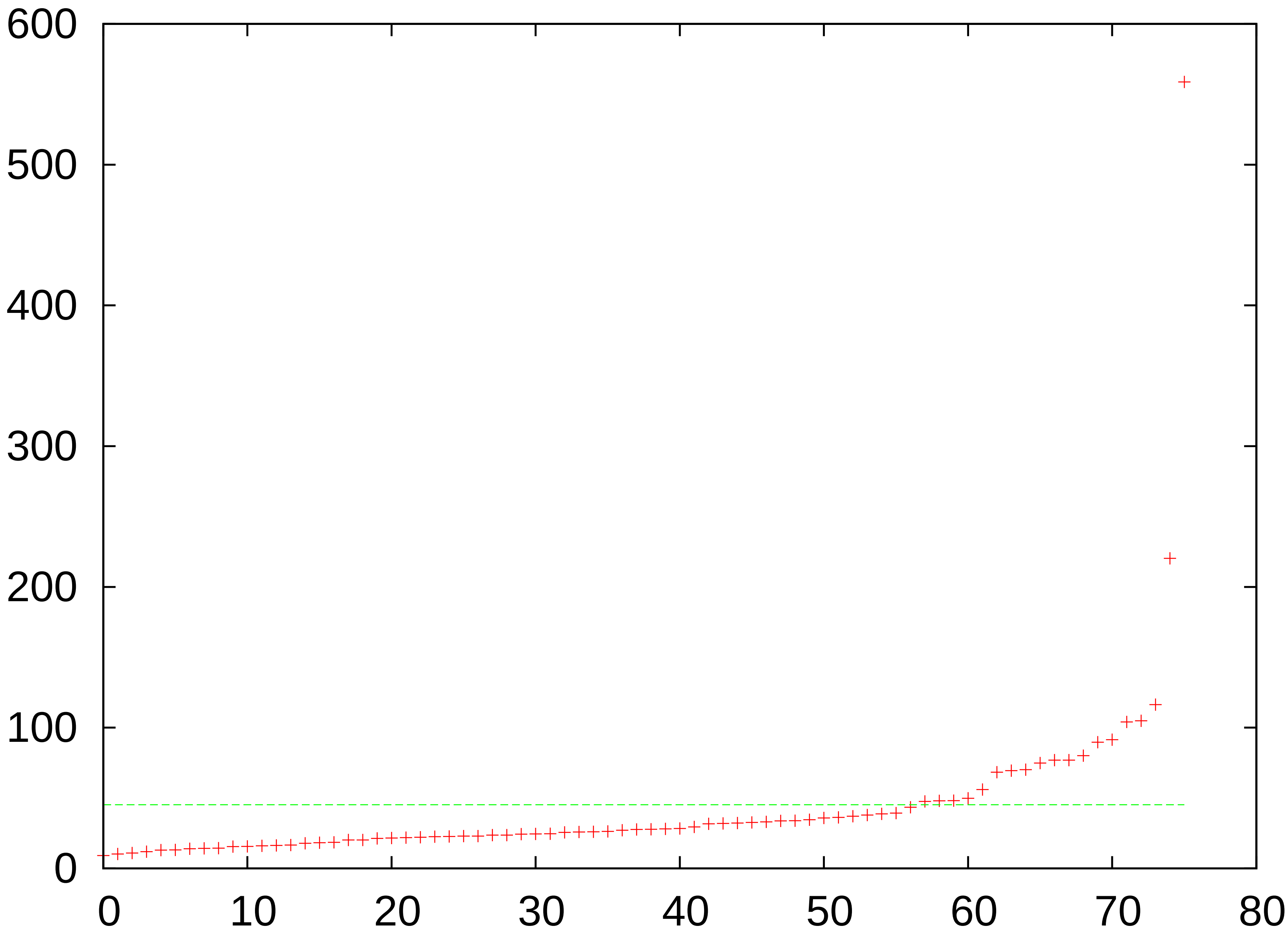
- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple 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

- special case *incremental preprocessing*:
  - preprocessing during incremental SAT solving
- 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 debug and get right — [JävisaloHeuleBiere’12]

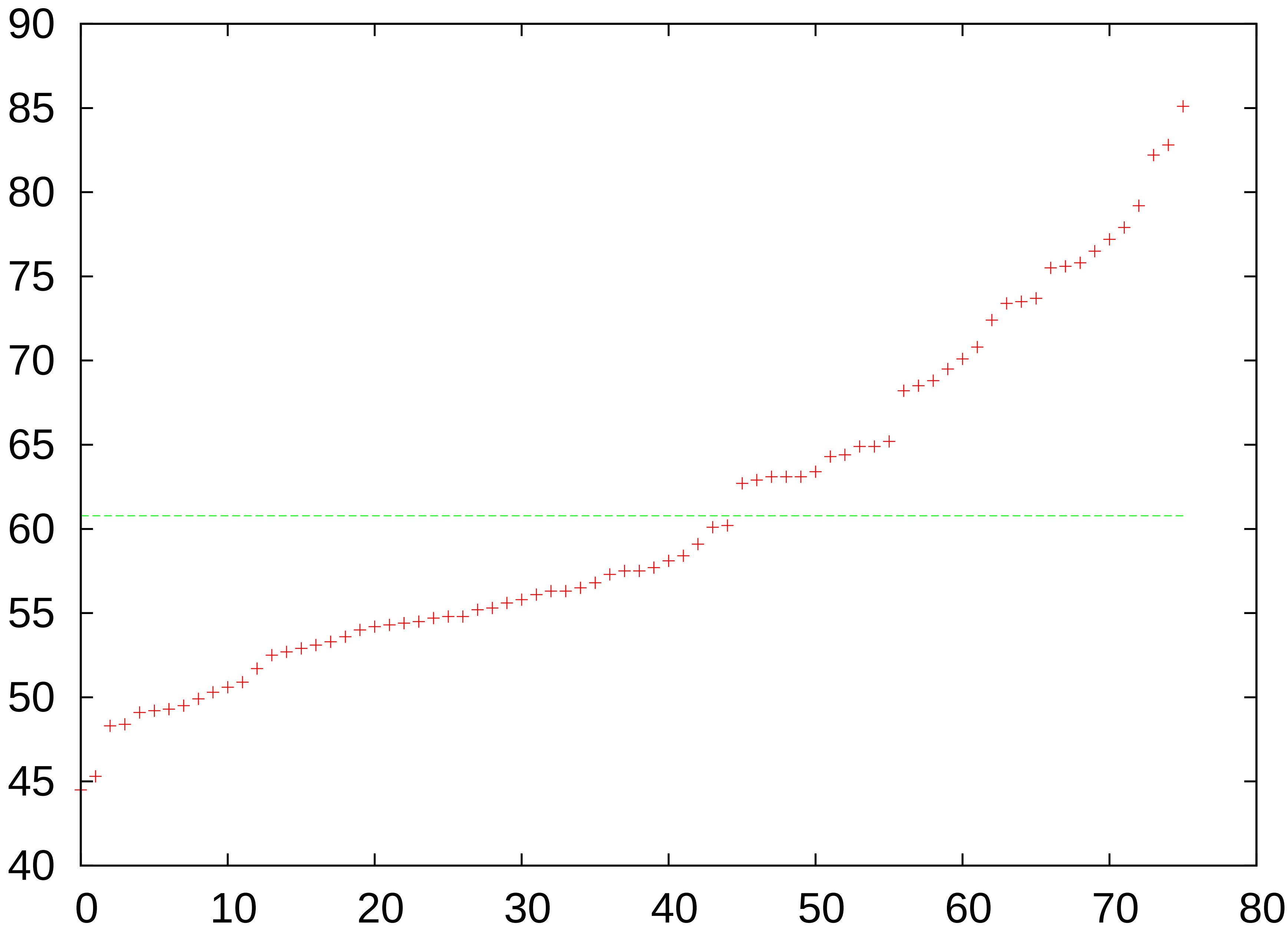

Literals
Stack
Clauses
1
start
top
end
–2
start
top
end
2
start
top
end
–3
start
top
end
–8
1
–2•
7
–8•
–2•
3 •
–5
1 •
–2•
1
3

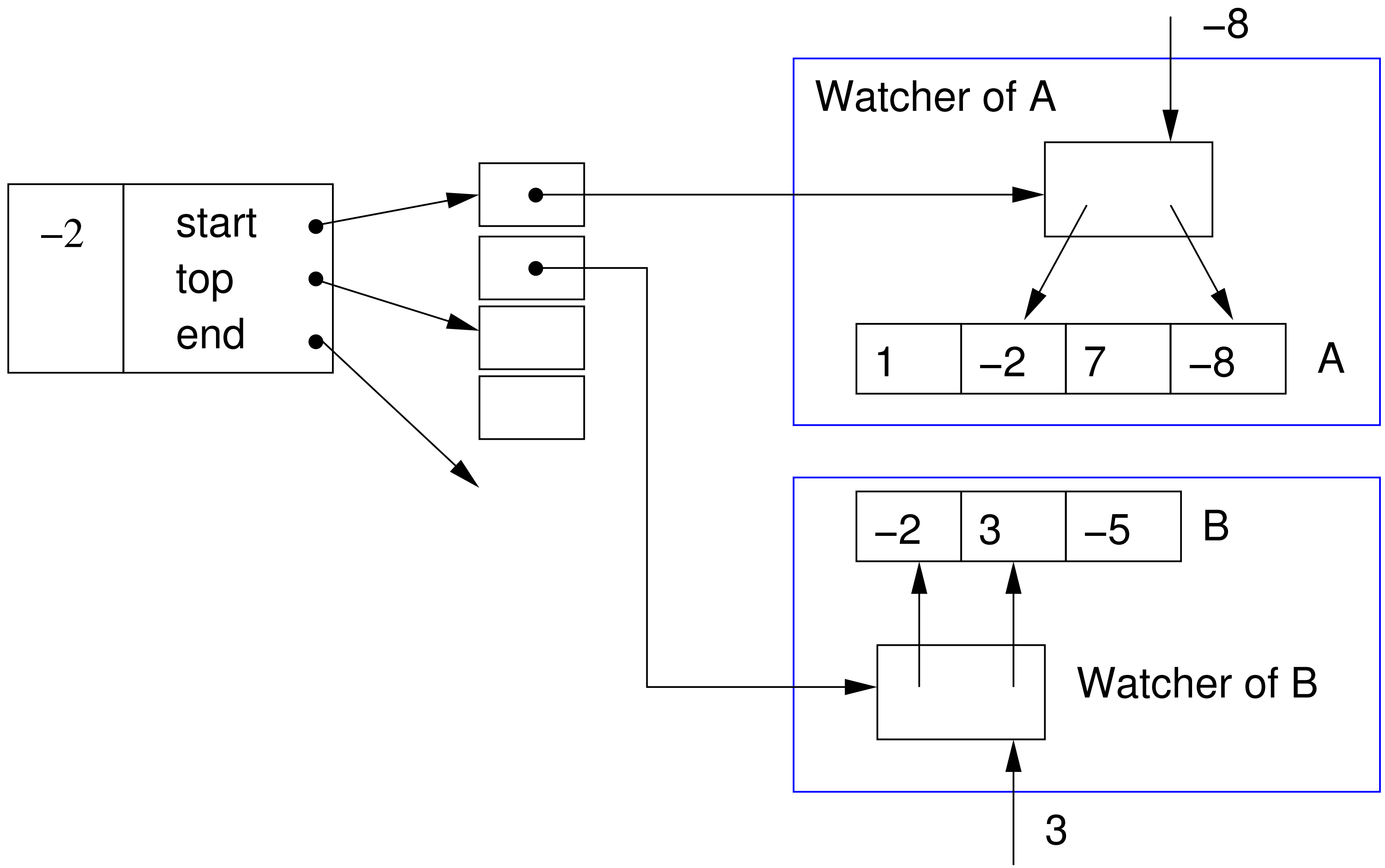

25
20
15
10
5
0
0
10
20
30
40
50
60
70
80

Average Learned Clause Length

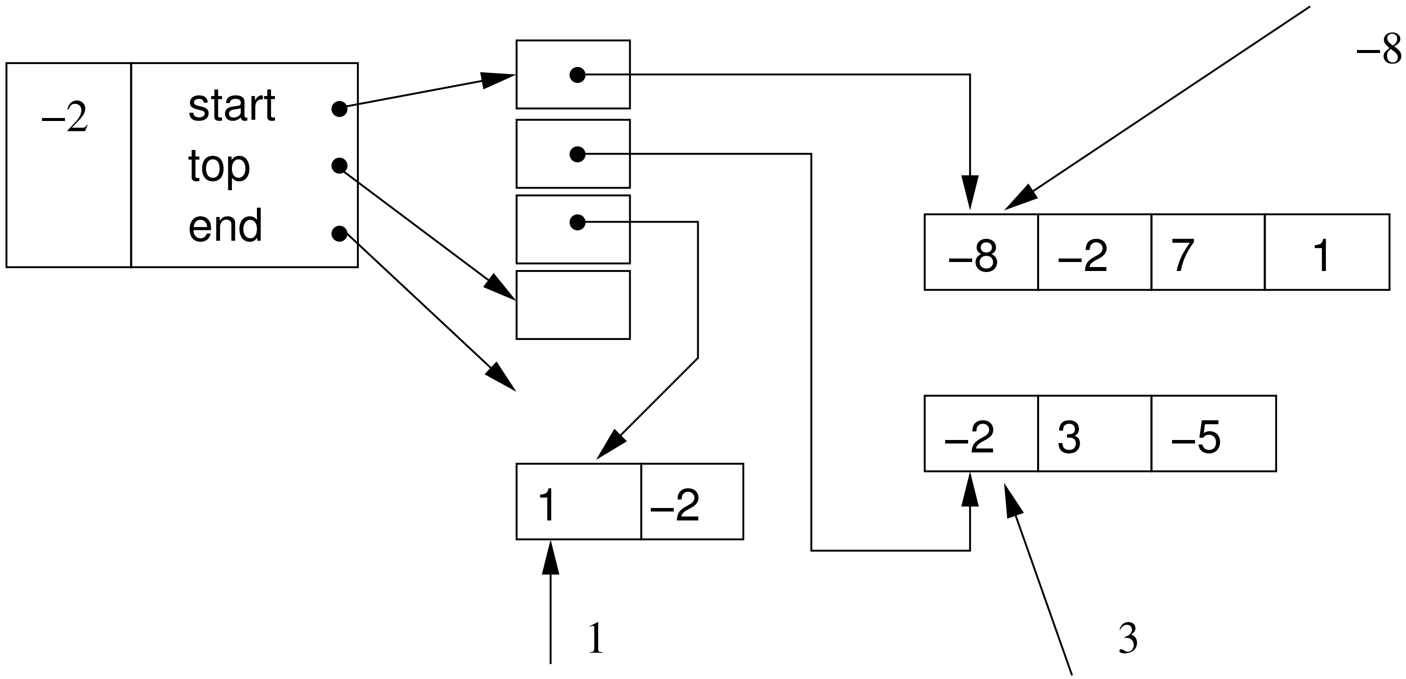


Percentage Visited Clauses With Other Watched Literal True



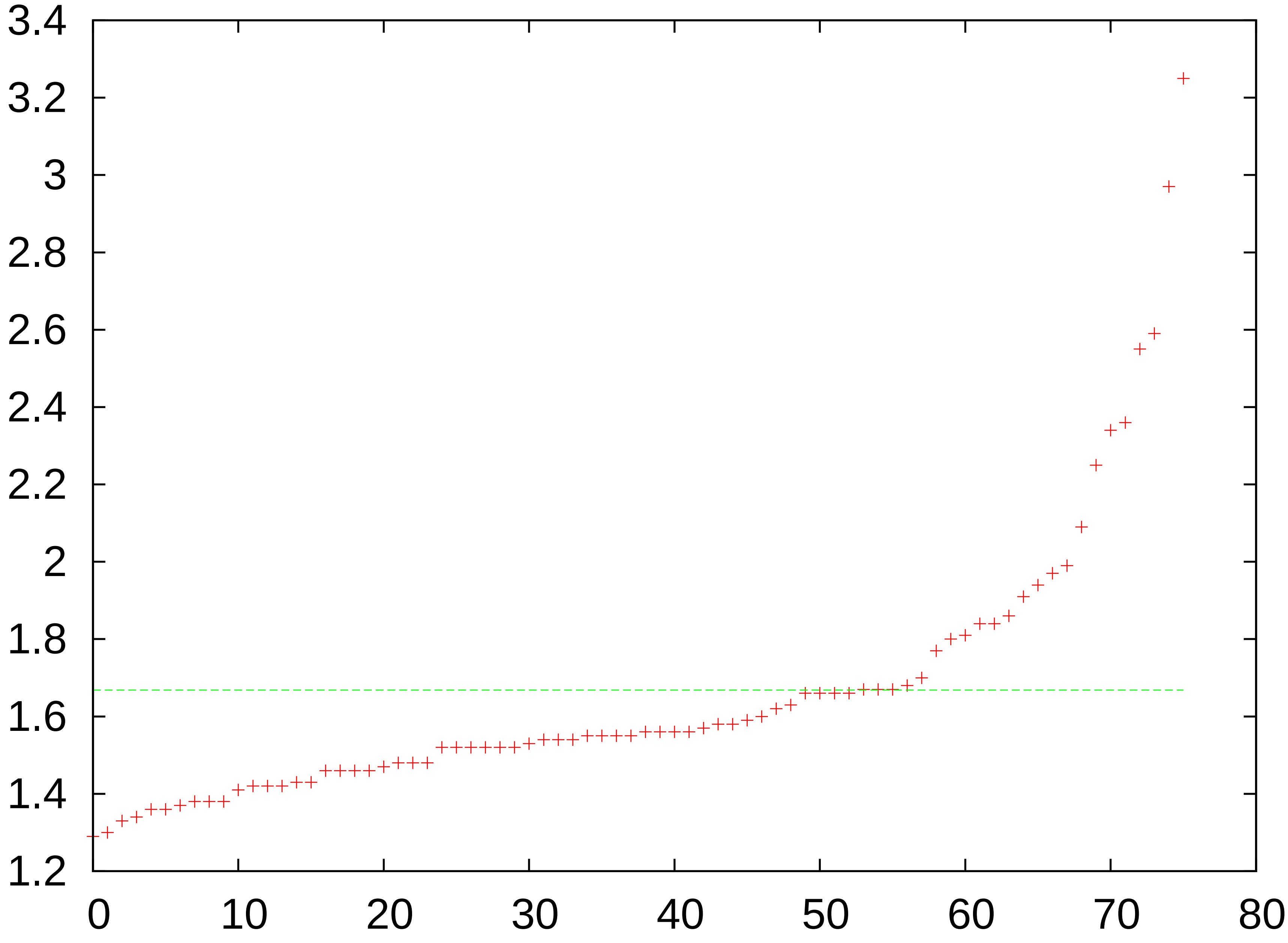


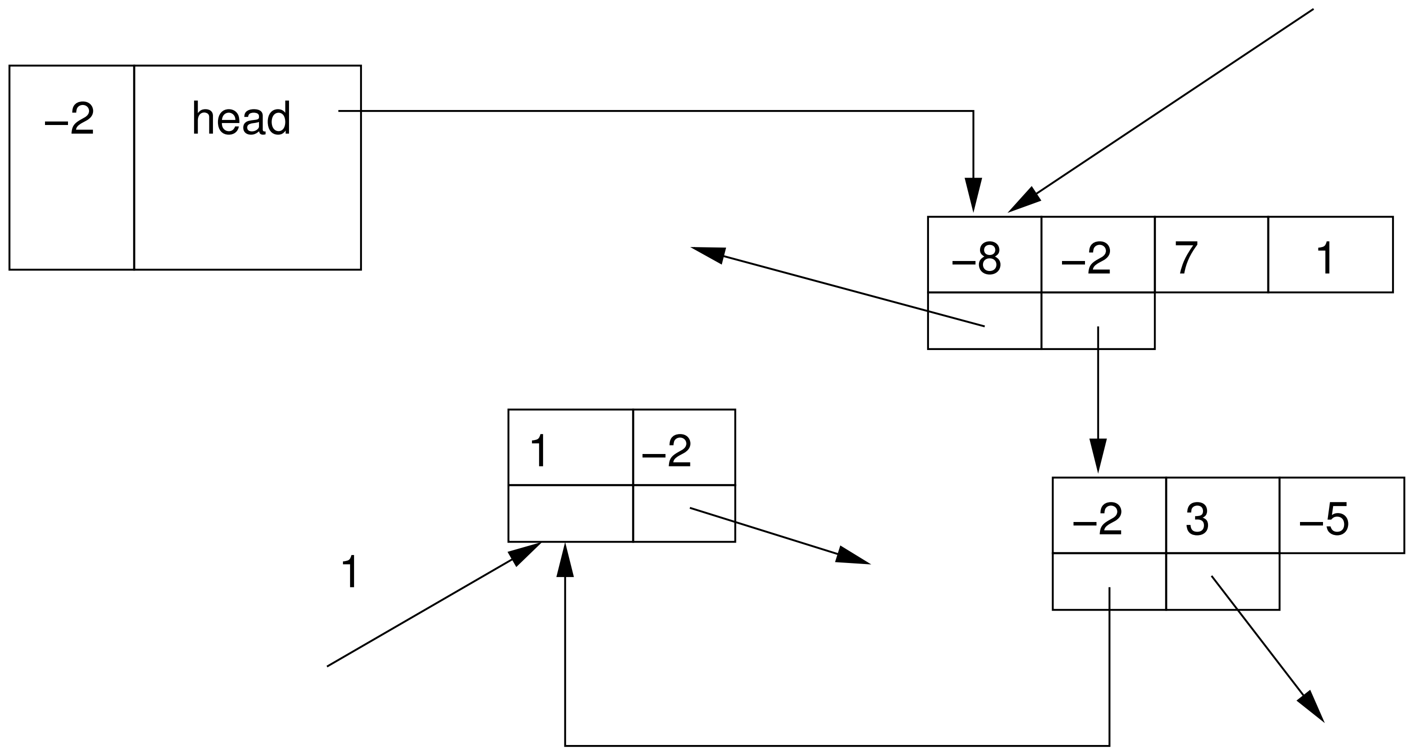
−8
Watcher of A
−2
start
top
end
1
−2
7
−8
A
−2
3
−5
B
Watcher of B
3



invariant: first two literals are watched

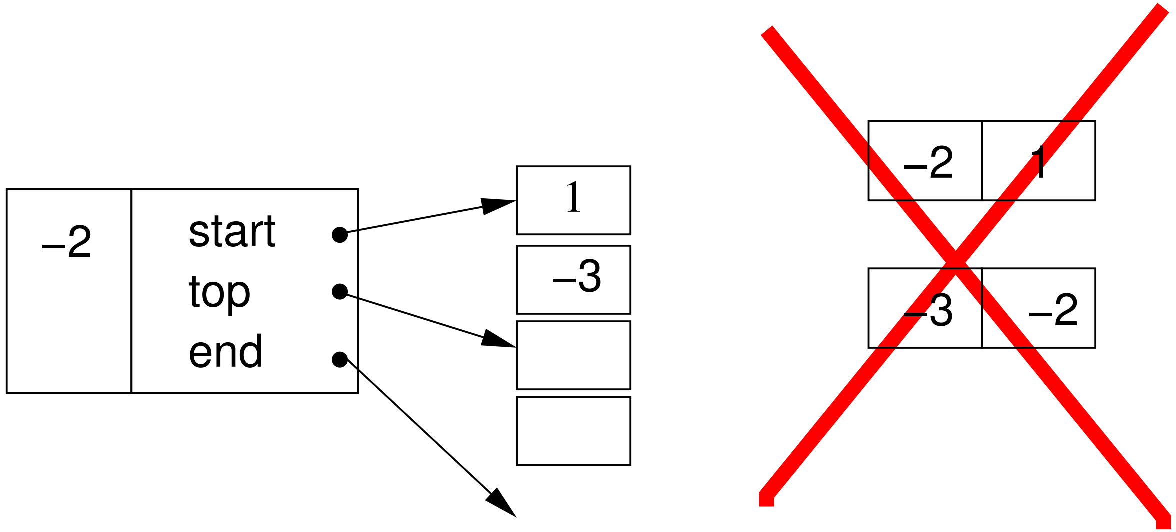
Average Number Literals Traversed Per Visited Clause



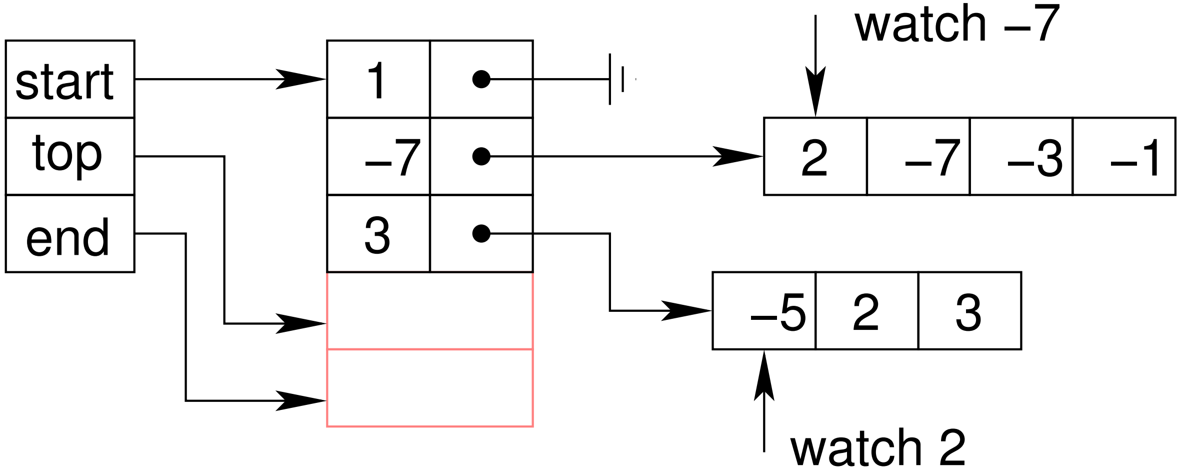


invariant: first two literals are watched

Additional Binary Clause Watcher Stack



Caching Potential Satisfied Literals (Blocking Literals)



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