SAT Solving

#342.201

http://fmv.jku.at/sat

Summer Semester 2021

Johannes Kepler University

Linz, Austria

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

Motivation

- more and more complex systems

  Moore's Law $\Rightarrow$ soon we will have 10^{30} transistors / processor

  multi-million LOC / OS

  $\Rightarrow$ exploding testing costs (in general not linear in system size)

- increased dependability

  everything important depends on computers:

  stir by wire, banking, stock market, workflow, ...

  $\Rightarrow$ quality concerns

- increased functionality

  security, mobility, new business processes, ...

Test

standard definition: dynamic execution / simulation of a system

integration in development process necessary

extreme position: testing should actually “drive” the development process

Verification

standard definition: static checking, symbolic execution

hardware design: verification is the process of testing

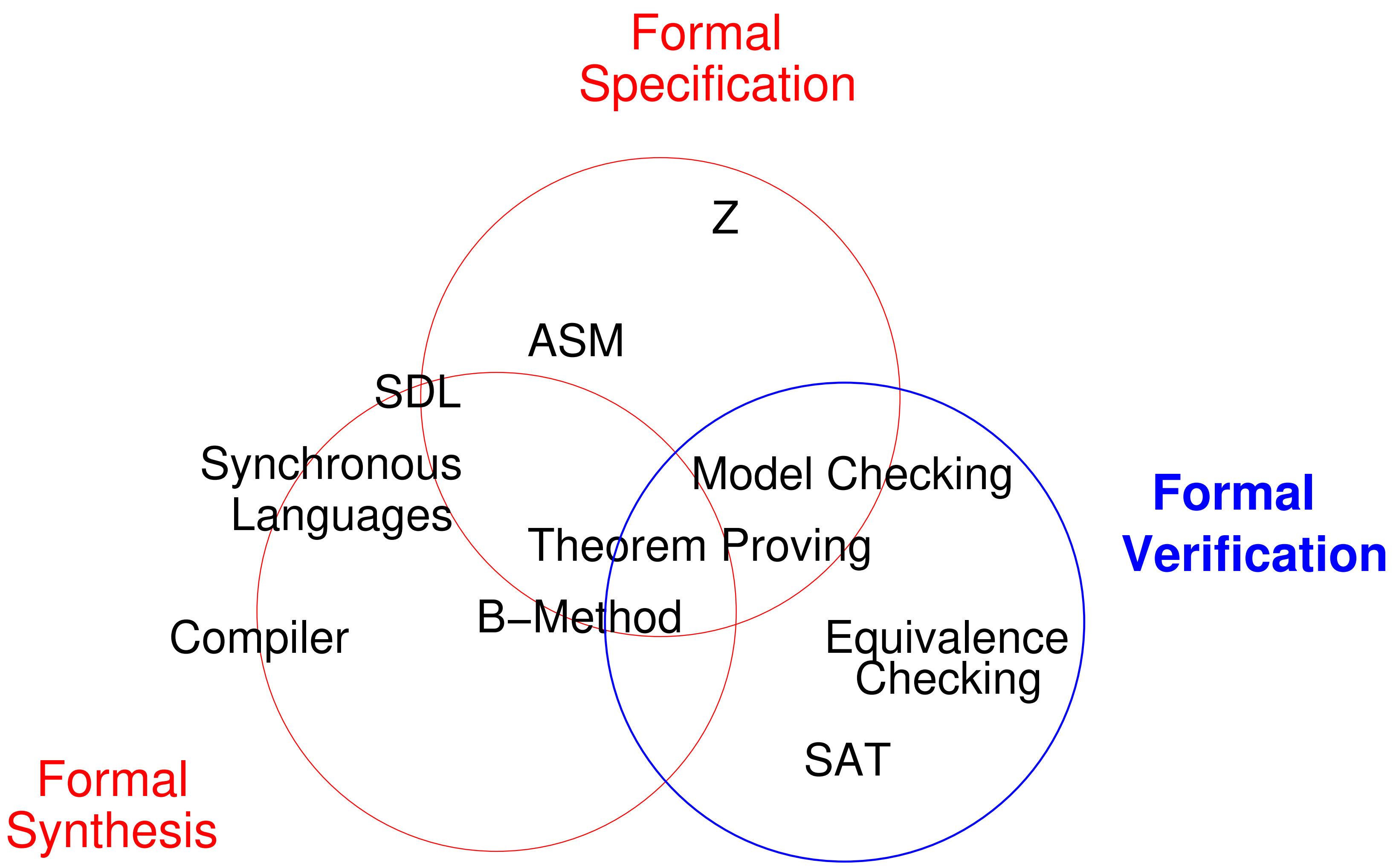
$\Rightarrow$ our view: Test = Verification

Implications

- not unusual to have more than 50% of resources allocated to testing
- testing and verification are (becoming) the bottleneck of development
- quality dilemma (drop quality for more features)
- more efficient methods for test and verification needed
  $\Rightarrow$ formal verification is the most promising approach
- experts in new testing and verification methods are lacking
- long term: more formal development process not just formal verification

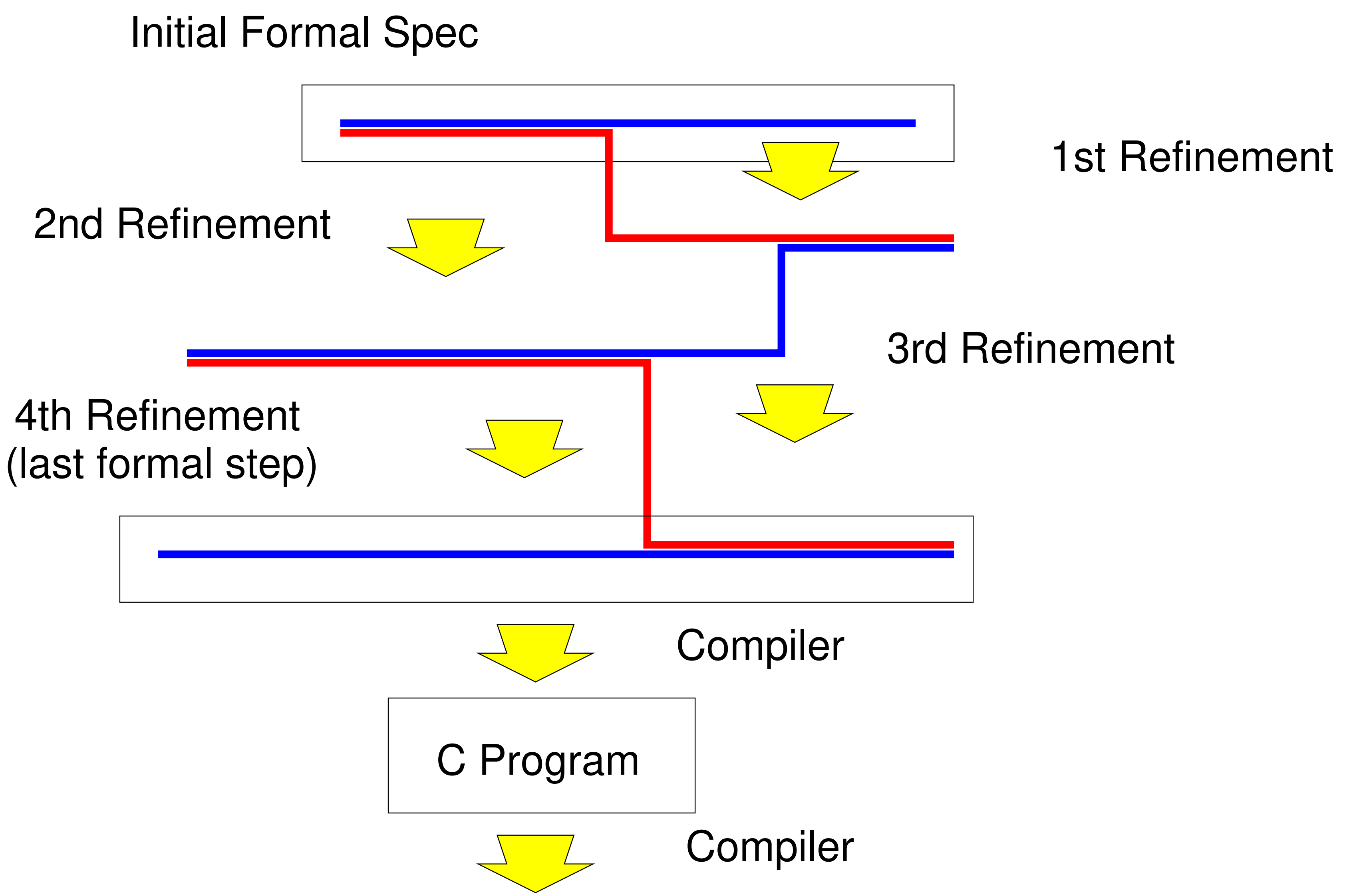
Formal Methods

- formal = mathematical
- mathematical models $\Rightarrow$ precise semantics
- emphasizes static / symbolic reasoning about programs
  (so standard definition of verification falls into this category)
- rather narrow view in digital design: equivalence and model checking
- not esoteric: compilation in a broad sense is a formal method
  (high-level description is translated into low-level description)
- our view: use tools for reasoning (i.e. programs are formal entities)

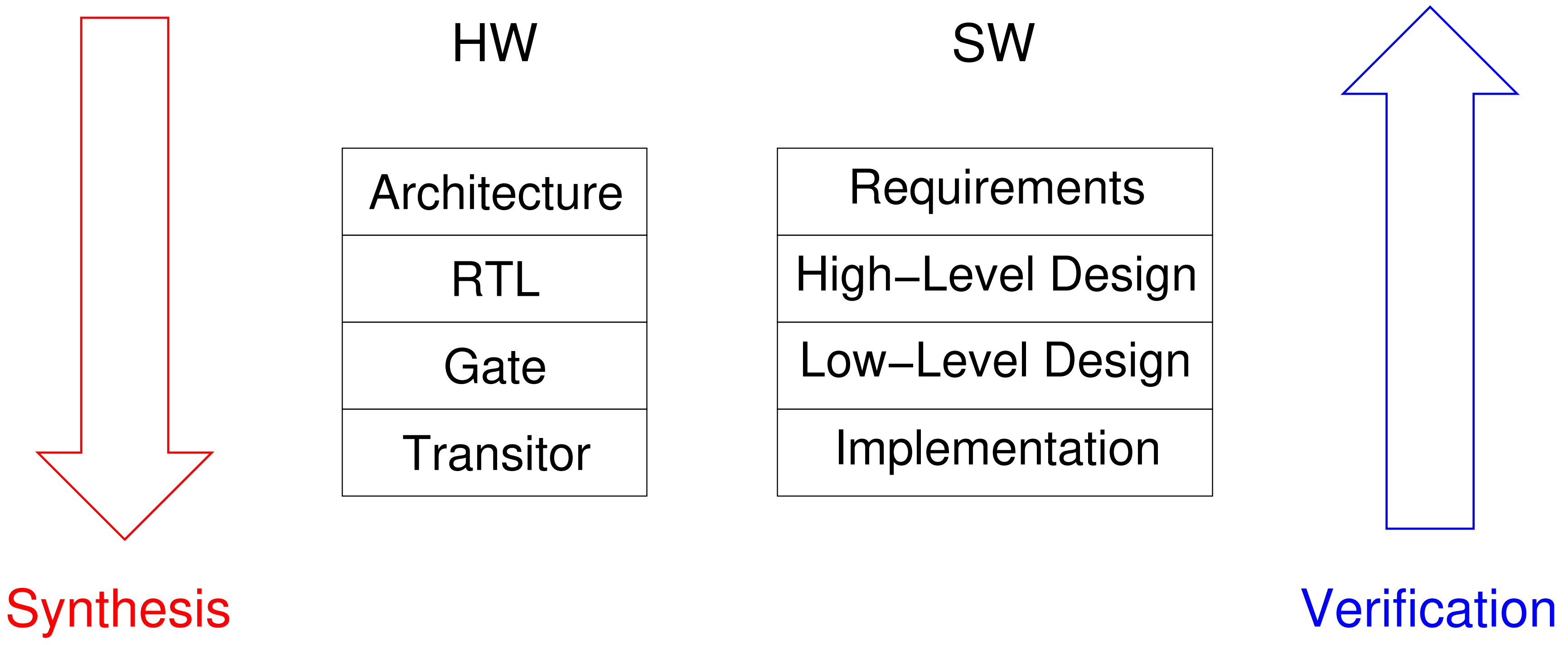

Formal
Specification
Z
ASM
SDL
Synchronous
Languages
Model Checking
Formal
Verification
Theorem Proving
B–Method
Compiler
Equivalence
Checking
SAT
Formal
Synthesis

Formal Specification

- abstracts from unnecessary implementation details
- high-level mathematical model of the system
- very useful for high-level design
- catches ambiguous or inconsistent specifications
- formal specification per se: no tools for refinement / checking
- good example: ASM


Initial Formal Spec
1st Refinement
2nd Refinement
3rd Refinement
4th Refinement
(last formal step)
Compiler
C Program
Compiler

- integrates verification in the development process
- usually pure top-down design and incremental refinement steps
- splits large verification tasks (divide et impera) ...
- ... but forces dramatic change in development process
- it works but is costly
- each refinement step uses formal verification methods
  $\Rightarrow$ more powerfull verification algorithms allow more automation
- good example: B-Method



1. no implementation without Synthesis
2. Verification is added value (Quality)
3. both processes are incremental
4. both processes can be formal

Formal Verification

- assumptions: specification and system are given
- formal verification checks formally that system fulfills specification
- least change in development process
- full blown verification is really difficult: “post mortem verification”
- simplifications: focus on simple partial specifications
  (type safety, functional equivalence of two systems, ...)
- methods (implemented in tools):

  simple algorithms for deducing properties directly

  complex algorithms for hard or even undecidable problems

- boolean methods:

  SAT, BDDs, ATPG, Combinational Equivalence Checking

- finite state methods:

  Bisimulation and Equivalence Checking of Automata, Model Checking

- term based methods:

  Term Rewriting, Resolution, Tableaux, Theorem Proving

- Abstraction (e.g. SLAM uses BDDs, Model Checking, Theorem Proving)

Focus

- how does it work?

  (algorithms and data structures)

- necessary background for use of formal verification

  (and formal methods in general)

- capacity and restrictions

- first step to become an expert in a fast expanding area

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

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

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

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

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

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

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

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

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?

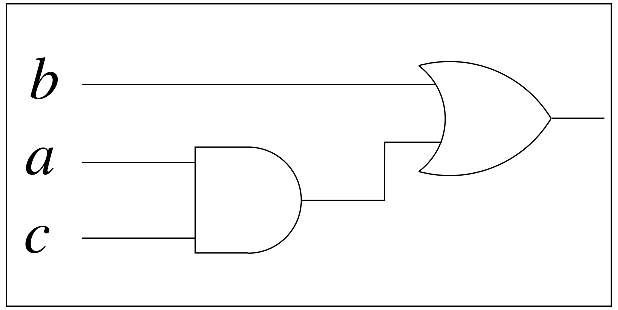
or equivalently:

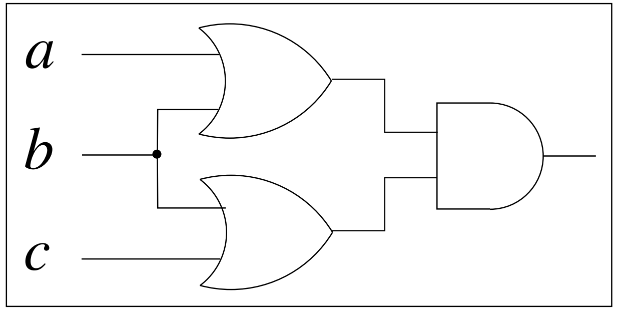
Is the boolean formula $\boxed{\text{compile}(\mathit{original}) \not\leftrightarrow \text{compile}(\mathit{optimized})}$ satisfiable?

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





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

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 $V = \{x, y, \dots\}$.

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

SAT belongs to NP

There is a *non-deterministic* Turing-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).

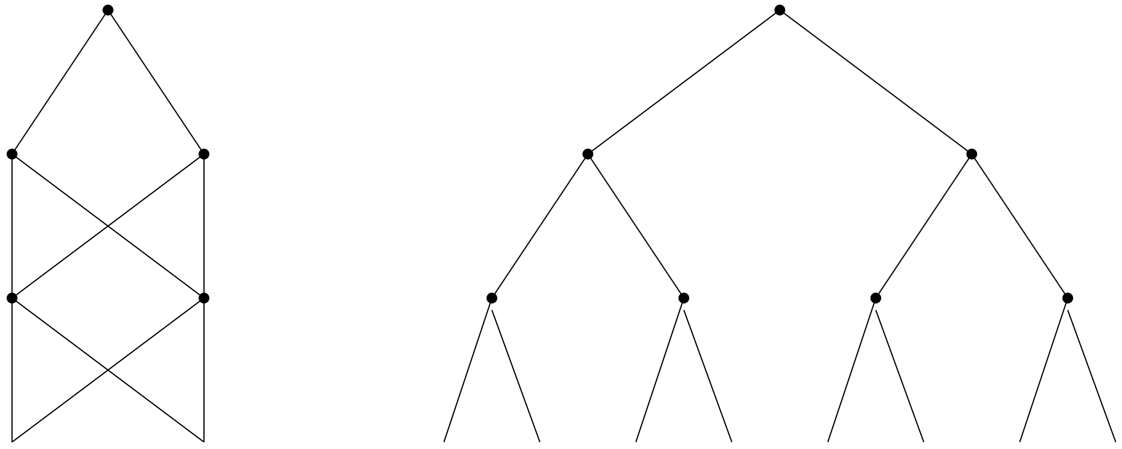
```
Formula
formula2nnf (Formula f, Boole sign)
{
  if (is_variable (f))
    return sign ? new_not_node (f) : f;
  if (op (f) == AND || op (f) == OR)
    {
      l = formula2nnf (left_child (f), sign);
      r = formula2nnf (right_child (f), sign);
      flipped_op = (op (f) == AND) ? OR : AND;
      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);
    }
}
```

Merging two CNFs

```
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;
}
```



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;
}
```

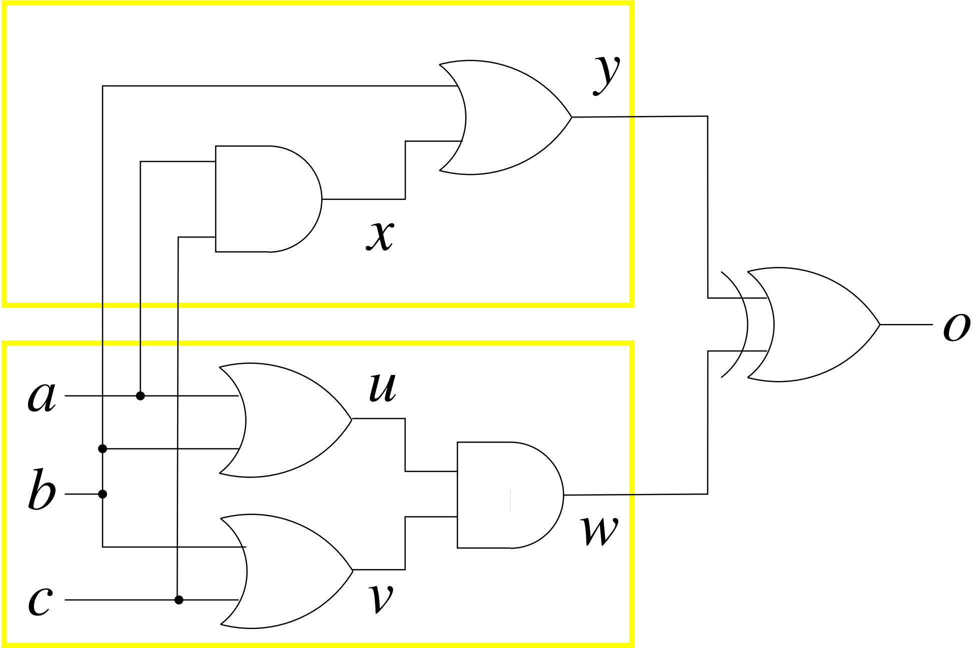
Eliminiate the `tmp...` variables through substitution.

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

How to detect Sharing

- 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



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

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

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

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 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)

Negation: $$x \leftrightarrow \overline{y} \;\Leftrightarrow\; (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \;\Leftrightarrow\; (\overline{x} \vee \overline{y}) \wedge (y \vee x)$$

Disjunction: $$x \leftrightarrow (y \vee z) \;\Leftrightarrow\; (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \;\Leftrightarrow\; (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z)$$

Conjunction: $$\begin{aligned} x \leftrightarrow (y \wedge z) &\Leftrightarrow (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\ &\Leftrightarrow (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x) \end{aligned}$$

Equivalence: $$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y))) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x) \end{aligned}$$

- 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]

Intermediate Representations

- 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: $s = x + y$

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

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

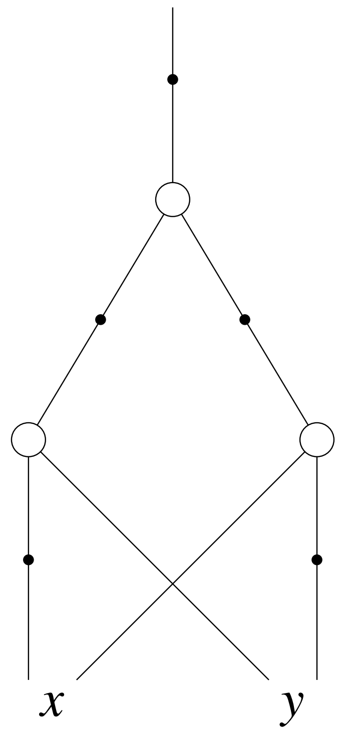
where

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

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

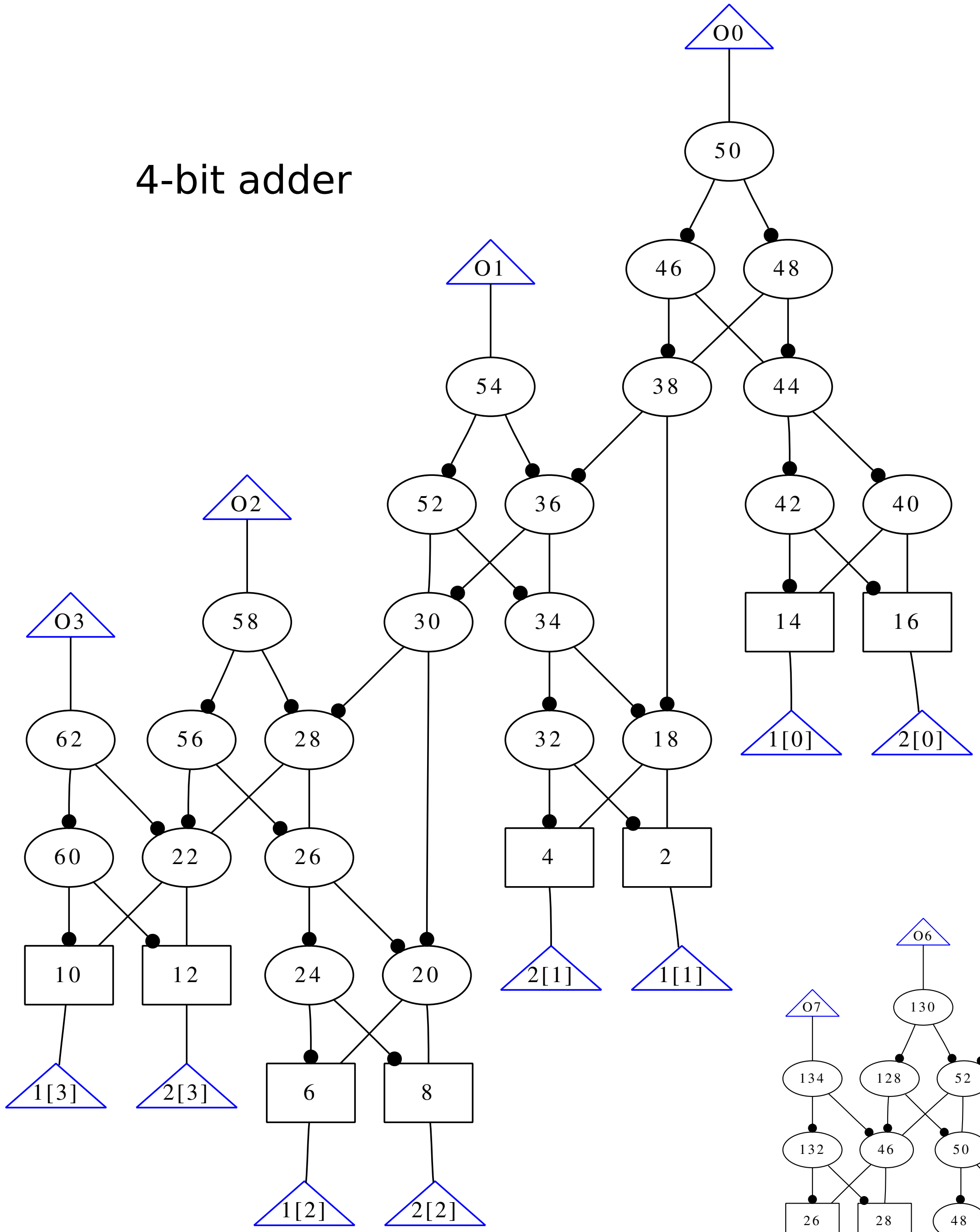
not part of node

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

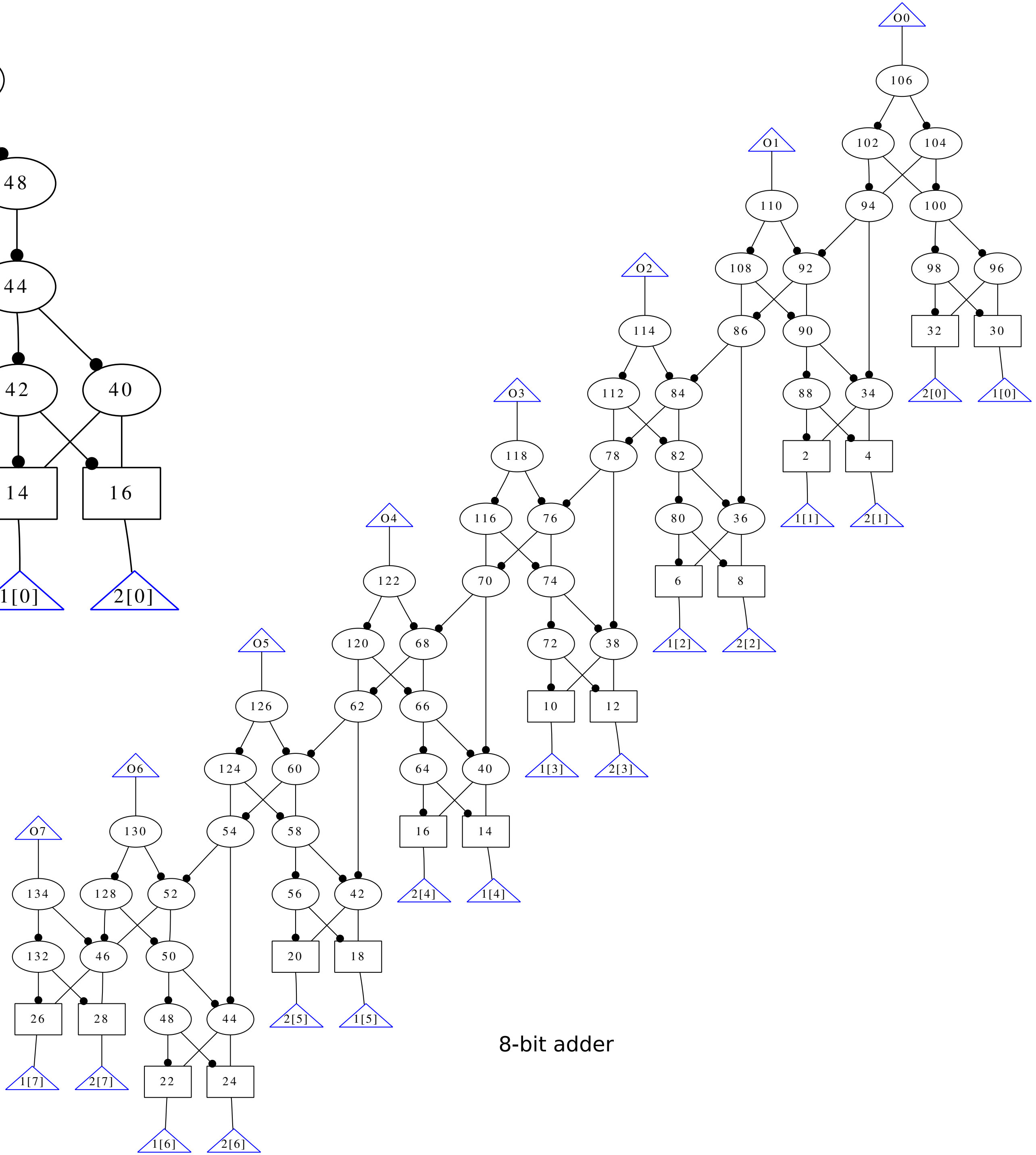
```c
typedef struct AIG AIG;

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:

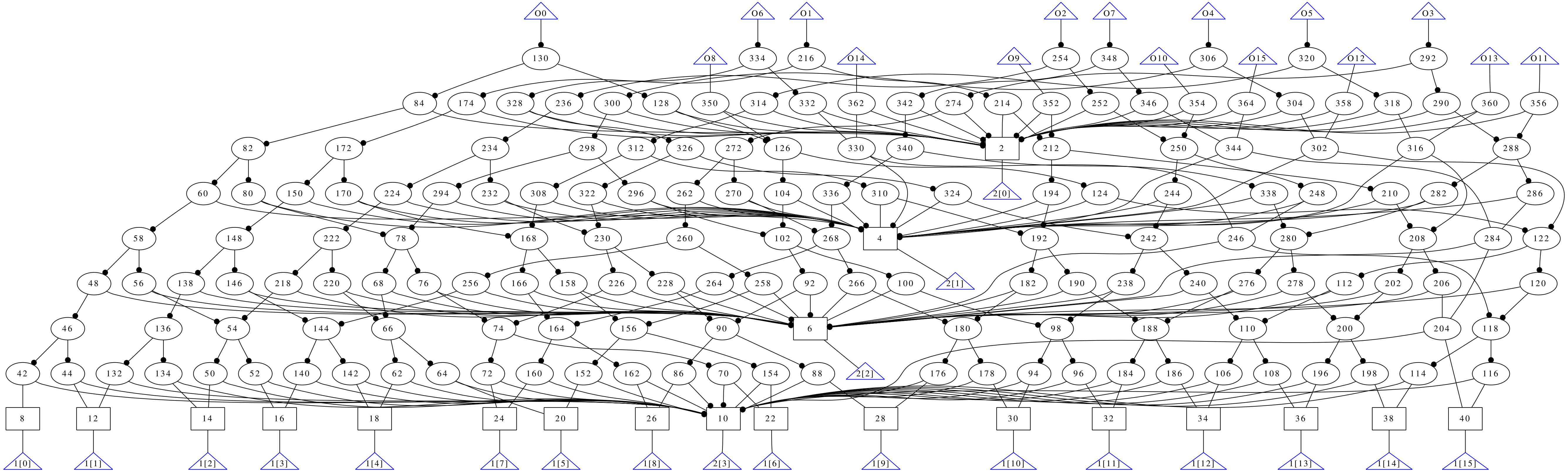
`sizeof(unsigned) == sizeof(void*)`



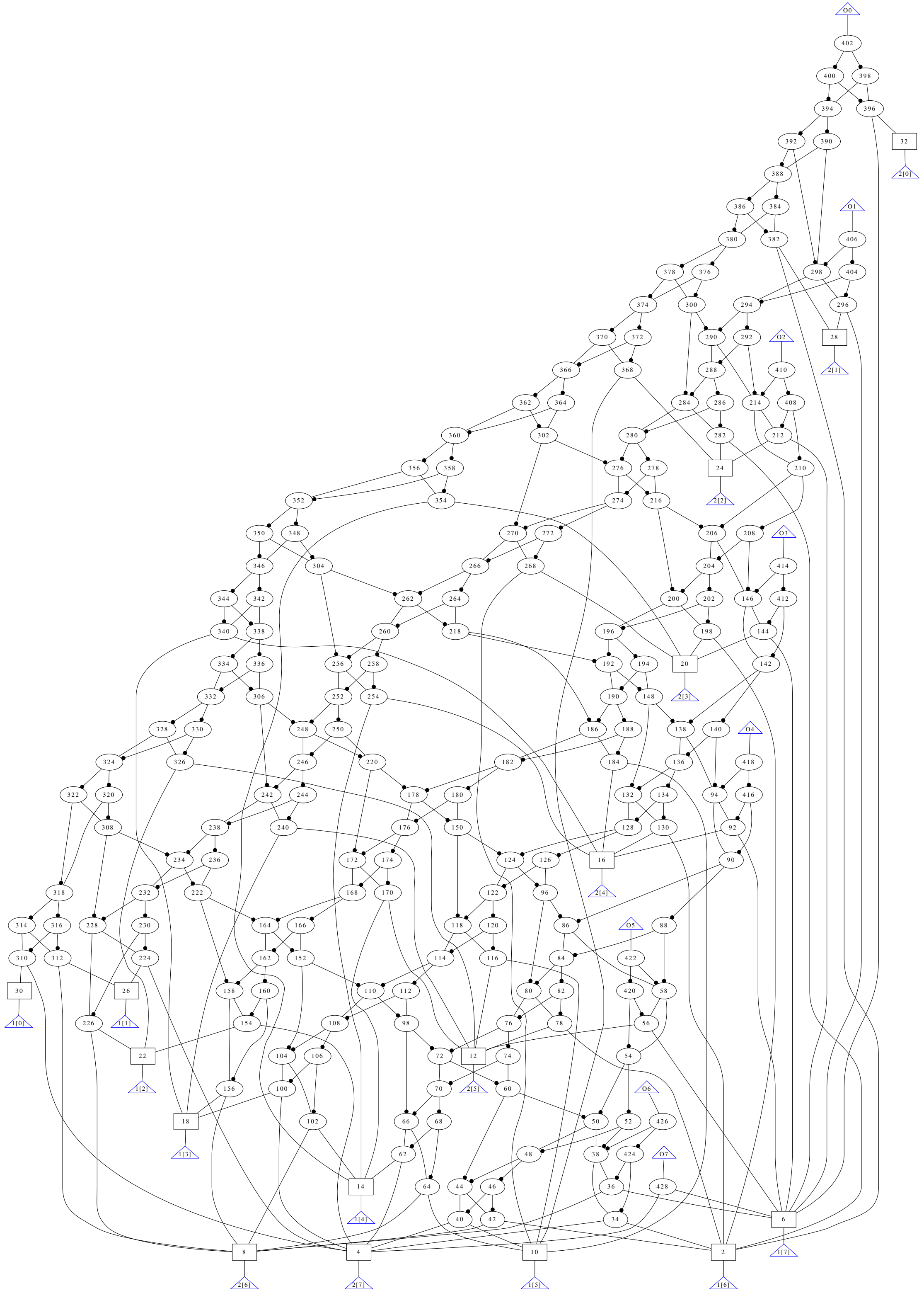
4-bit adder



8-bit adder



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


O0
O1
O2
O3
O4
O5
O6
O7
1[0]
1[1]
1[2]
1[3]
1[4]
1[5]
1[6]
1[7]
2[0]
2[1]
2[2]
2[3]
2[4]
2[5]
2[6]
2[7]

- 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 $x = y = true$ but unfortunately
  - only largest fix-point can be (directly) encoded in SAT otherwise need ASP

- given a set of literals $\{l_1, \dots l_n\}$
  - constraint the *number* of literals assigned to *true*
  - $|\{l_1, \dots, l_n\}| \geq k$ or $|\{l_1, \dots, l_n\}| \leq k$ or $|\{l_1, \dots, l_n\}| = k$
- 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

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

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

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

Davis & Putnam Procedure (DP)

- dates back to the 50ies:
  - original version is *resolution based* (successful only in preprocessors)
  - improved DPLL: case analysis (try $x = 0, 1$ in turn and recurse)
  - evolved to CDCL (conflict driven clause learning): state-of-the-art
- recent ($\leq$ 20 years) optimizations:
  - backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures
    we will have a look at each of them
- elimination procedure of original DP is similar to
  - Gaussian elimination on linear real equalities
  - Fourier-Motzikin on linear real **in**equalities
  - Buchberger's algorithm on polynomial equations

- 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 $\quad (\sigma(c) = 1$ or $\sigma(v) = 1) \quad$ and $\quad (\sigma(d) = 1$ or $\sigma(\neg v) = 1)$

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

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

Example consider incorrect resolution $\dfrac{\{v\} \cup \{v\} \quad \{\neg v\}}{v}$ violating side condition

consider the following resolution
$$\frac{a \vee b \quad \neg b \vee c}{a \vee c}$$

in logical notation, not set notation for a change

let $\sigma(x) = \begin{cases} 1 & \text{if } x = a \\ 1 & \text{if } x = b \\ 0 & \text{if } x = c \end{cases}$ be a model of resolvent $(a \vee c)$ thus $\sigma(a \vee c) = 1$

note that $\sigma(\neg b \vee c) = 0$ and thus σ is not a model of 2nd antecedent (2nd premisse)

however σ satisfies remaining literal a of 1st antecedent in resolvent

thus simply flip value of pivot b (satisfy its occurrence in 2nd antecedent)

we get $\sigma'(x) = \begin{cases} 1 & \text{if } x = a \\ 0 & \text{if } x = b \\ 0 & \text{if } x = c \end{cases}$ satisfying both antecedents $\sigma'(a \vee b) = \sigma'(\neg b \vee c) = 1.$

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.**

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 $\neg x$

$\Rightarrow$ Bounded Variable Elimination

Example for Resolution DP

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:

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

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

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)?

- 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)

Basic DPLL Algorithm

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**

$DPLL(F)$

$F := BCP(F)$ 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 $DPLL(F \wedge \{\neg l\})$

$\Rightarrow$ $\boxed{\text{CDCL}}$

DPLL Example

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

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 $\{a \mapsto 0, b \mapsto 0\}$)

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 $\sigma(x) = 1$:

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

Correctness of Basic DPLL Algorithm

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

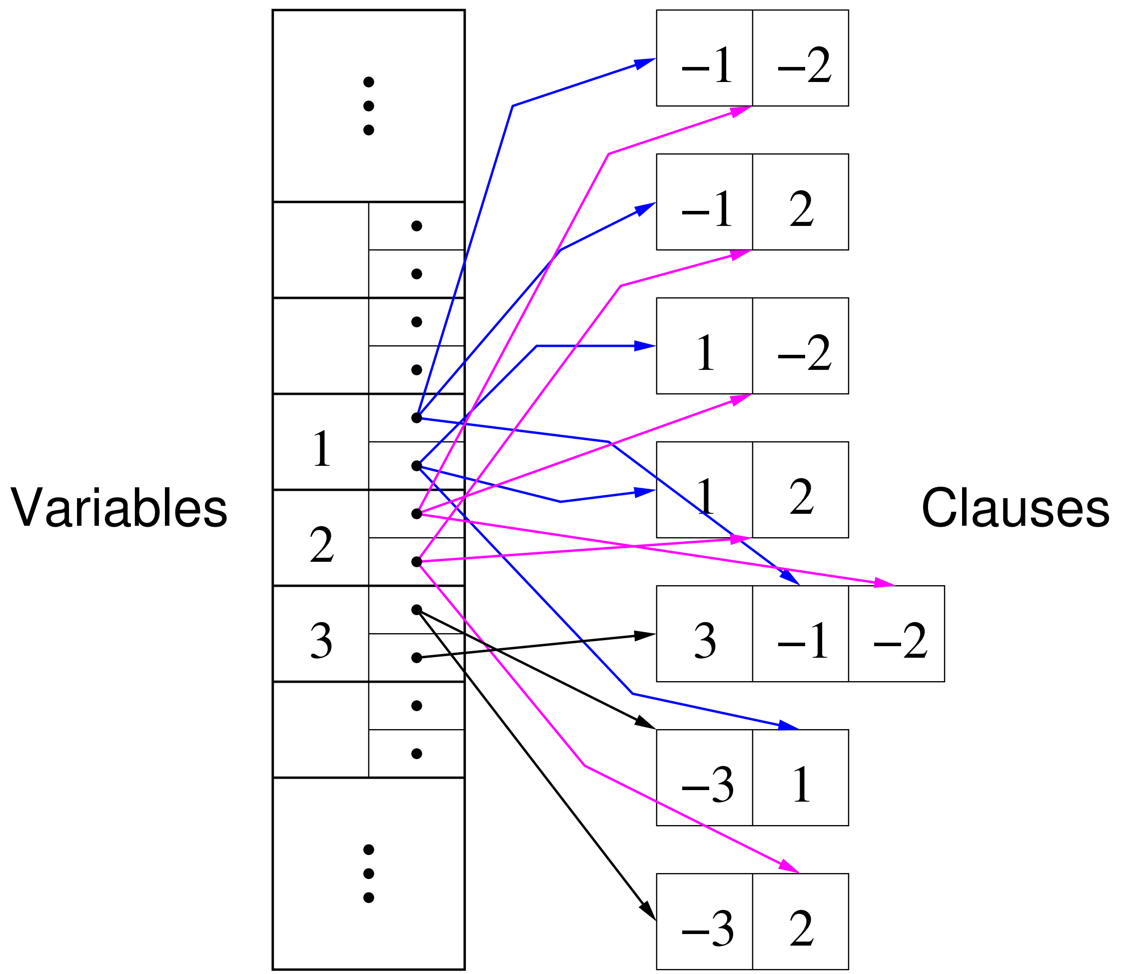
similarly, $\overline{x} \wedge f(x)$ is satisfiable iff $\overline{x} \wedge f(0)$ is satisfiable

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

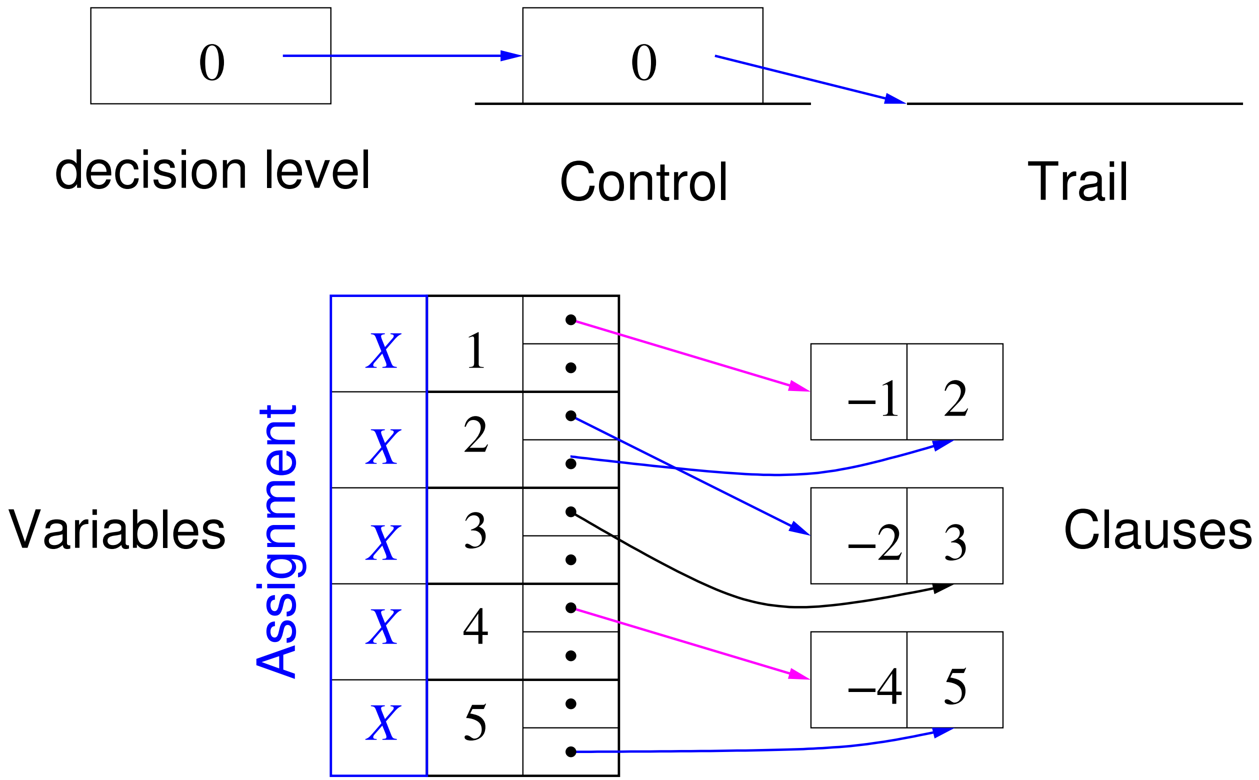
Simple Data Structures in DP Implementation

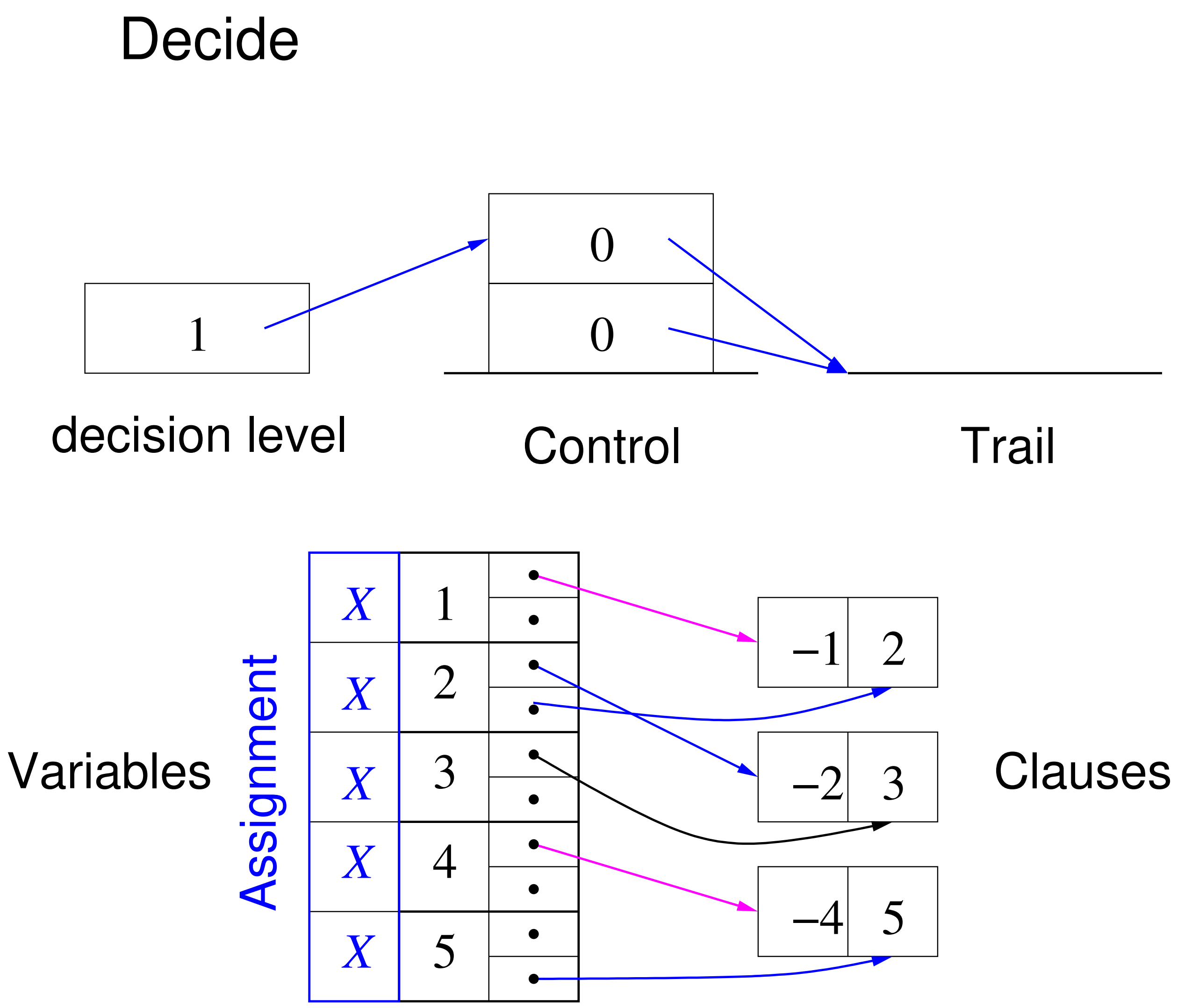


- each variable is marked as *unassigned*, *false*, or *true* ($\{X, 0, 1\}$)
- no explicit resolution:
  - when a literal is assigned visit all clauses where its negation occurs
  - find those clauses which have all but one literal assigned to false
  - assign remaining non false literal to *true* and continue
- decision:
  - heuristically find a variable that is still unassigned
  - heuristically determine phase for assignment of this variable

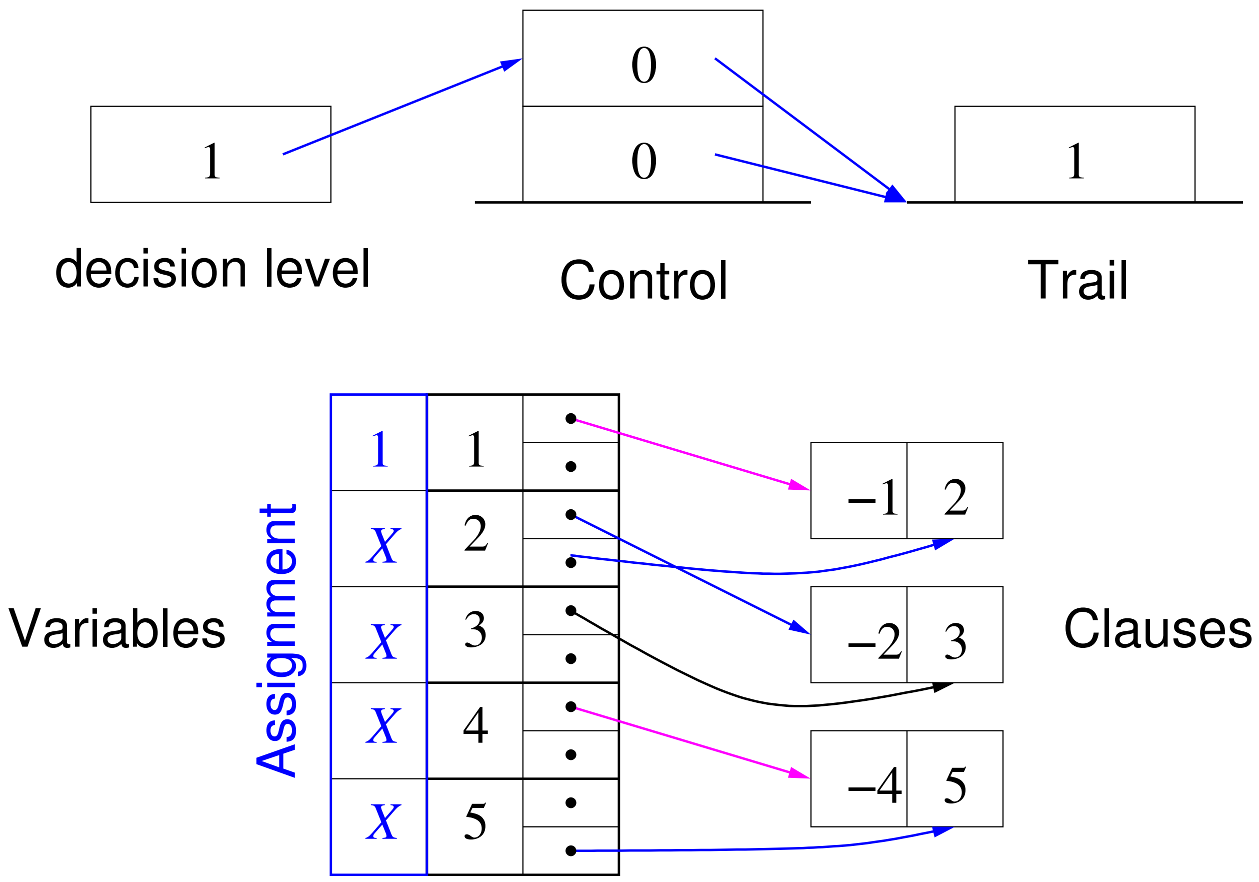
More Implementation Details

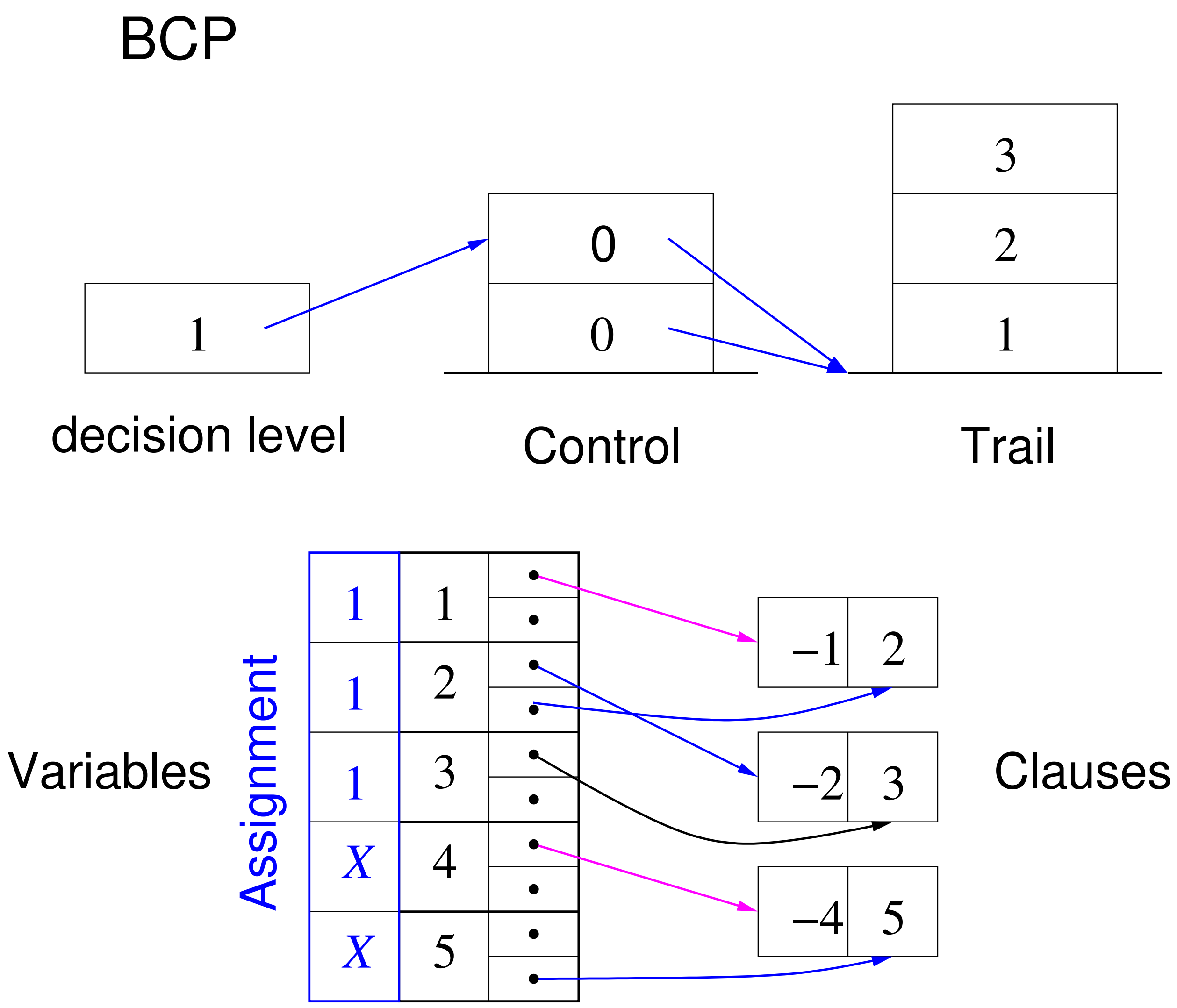
- *decision level* is the depth of recursive calls (= #nested decisions)
- the *trail* is a stack to remember order in which variables are assigned
- for each decision level the old trail height is saved on the *control stack*
- undoing assignments in backtracking:
  - get old trail height from control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

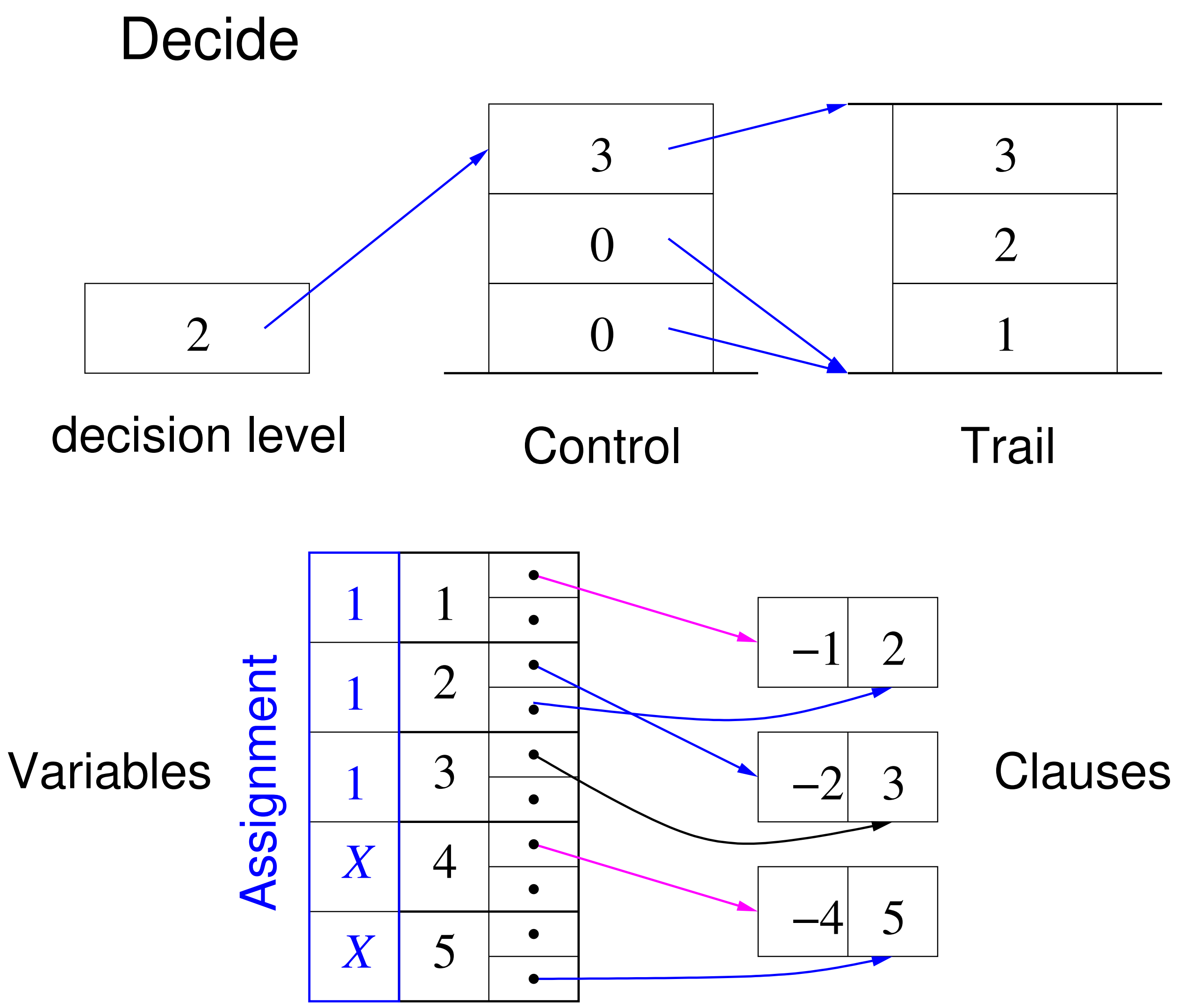

0
0
decision level
Control
Trail
Variables
Assignment
X
1
X
2
X
3
X
4
X
5
–1
2
–2
3
–4
5
Clauses

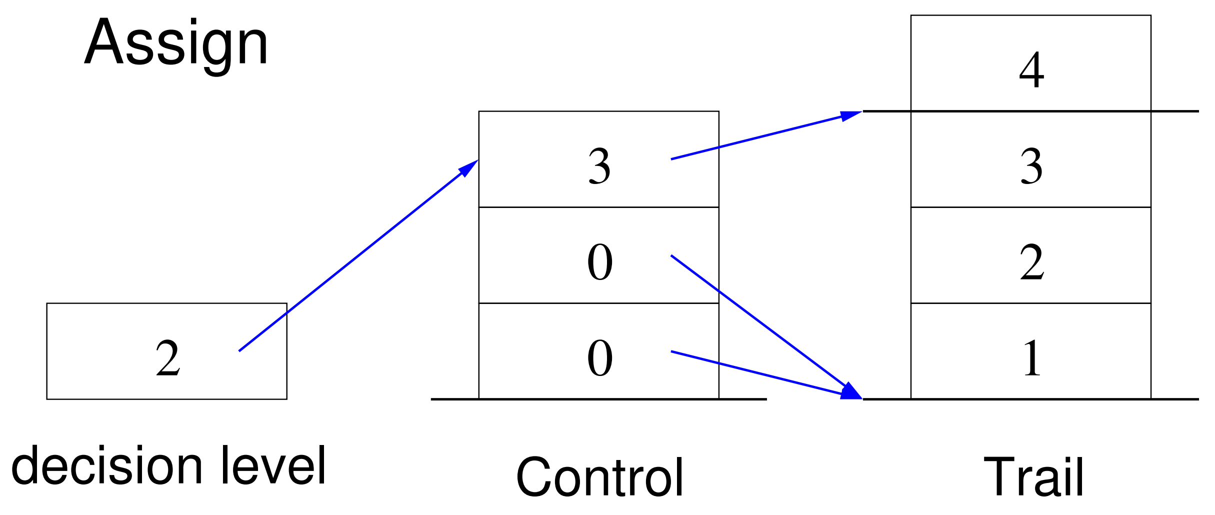

Decide
1
0
0
decision level
Control
Trail
Variables
Assignment
X
X
X
X
X
1
2
3
4
5
−1
2
−2
3
−4
5
Clauses

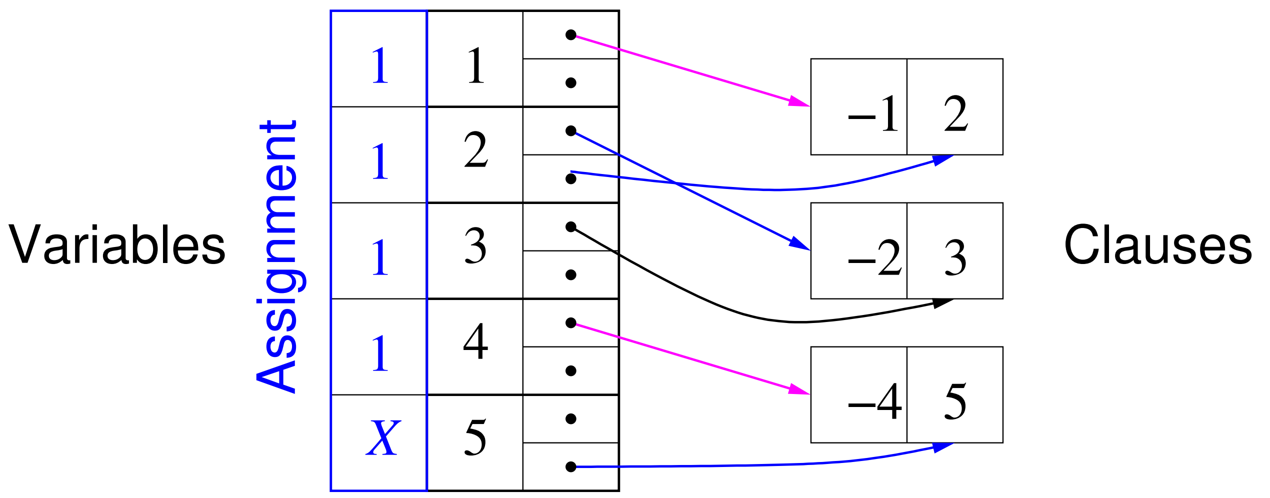
Assign


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

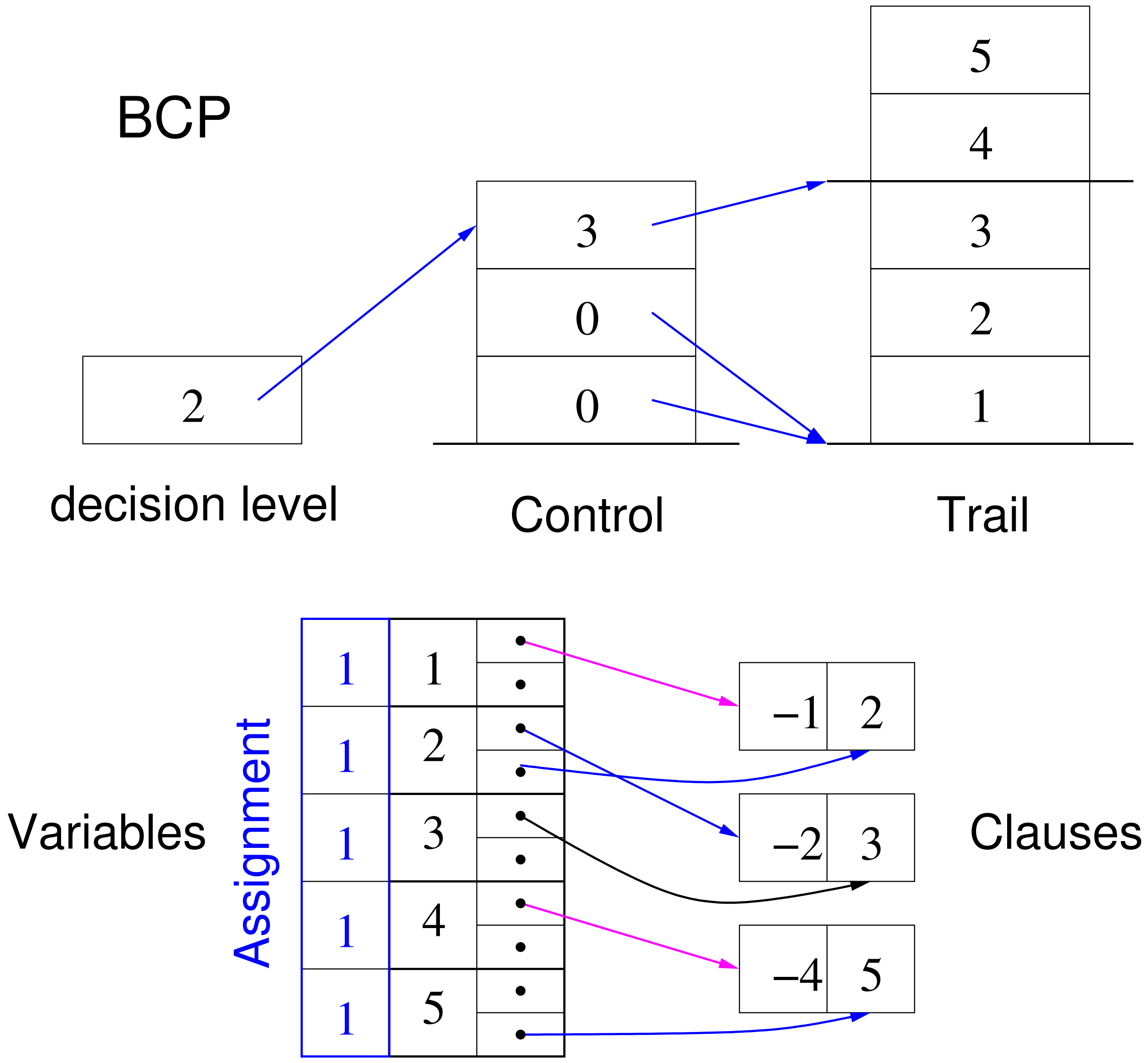

BCP
1
0
0
3
2
1
decision level
Control
Trail
Variables
Assignment
1 1
1 2
1 3
X 4
X 5
−1 2
−2 3
−4 5
Clauses


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

Example cont.



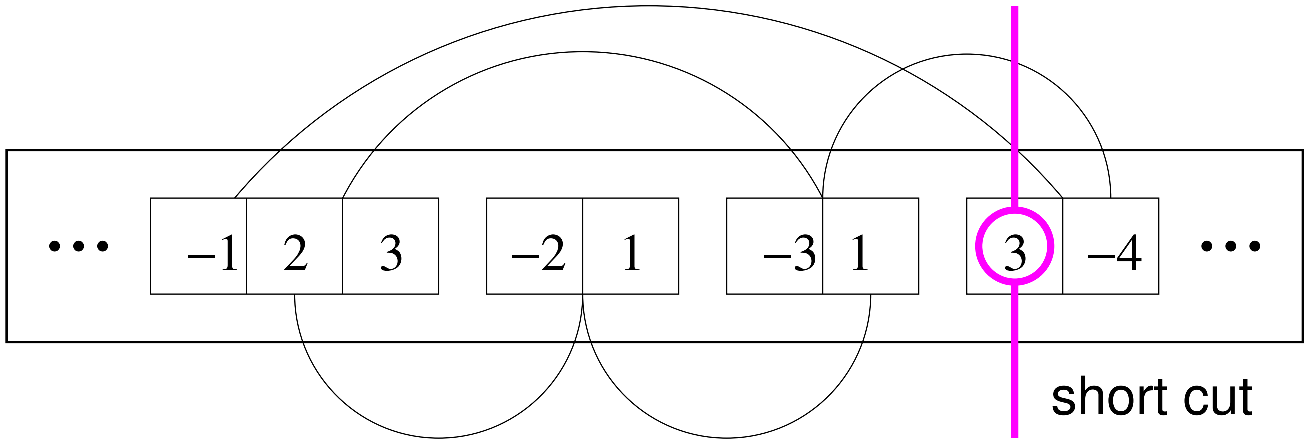
- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast, since only calculated once
  - can also use more expensive algorithms
- **dynamic heuristics**
  - typically calculated from number of occurences of literals
    (in unsatisfied clauses)
  - rather 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*)

Cut Width Heuristics

- 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;
}
```

- 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).

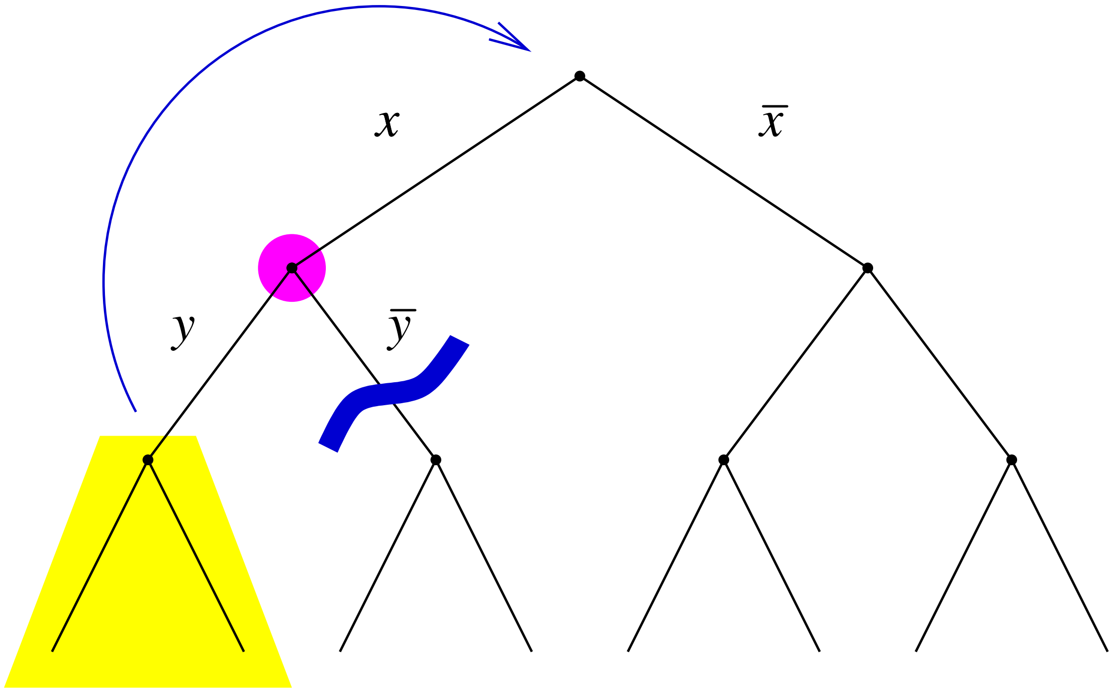
A Horn Form is satisfiable iff the minimal assignments of its positive part satisfies all its negative clauses as well.

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

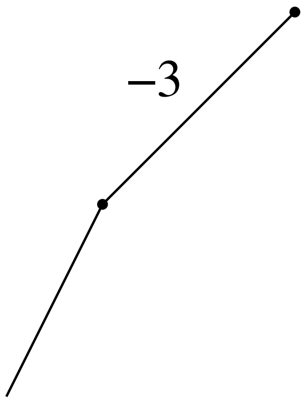
- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic first order heuristic (e.g. GRASP solver)
  - choose literal (variable + phase) which occurs most often
  - ignore satisfied clauses
  - requires explicit traversal of CNF (or more expensive BCP)
- look-forward heuristics (e.g. SATZ or MARCH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



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.

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

(−1 −2 ~~3~~)

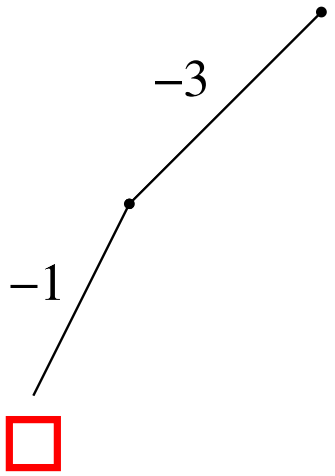
(−1 −2)

(−1 2)

(1 −2)

(1 2)

Split on -3 first (bad decision).



~~(−3 1)~~

~~(−3 2)~~

~~(−1 −2 $\cancel{3}$)~~

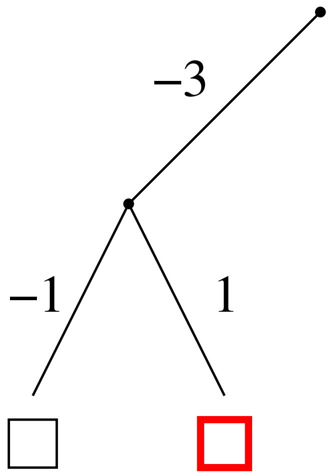
~~(−1 −2)~~

~~(−1 2)~~

($\cancel{1}$ −2)

($\cancel{1}$ 2)

Split on -1 and get first conflict.



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

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

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

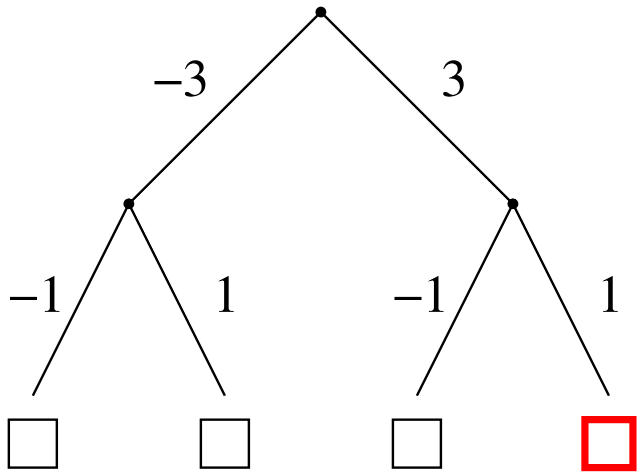
$(\cancel{-1}\ -2)$

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

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

$\cancel{(1\ 2)}$

Regularly backtrack and assign 1 to get second conflict.



~~$(-3\ 1)$~~

~~$(-3\ 2)$~~

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

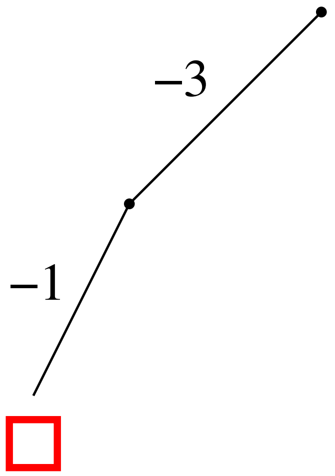
$(\cancel{-1}\ -2)$

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

~~$(1\ -2)$~~

~~$(1\ 2)$~~

Backtrack to root, assign 3 and derive same conflicts.



~~$(-3\ 1)$~~

~~$(-3\ 2)$~~

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

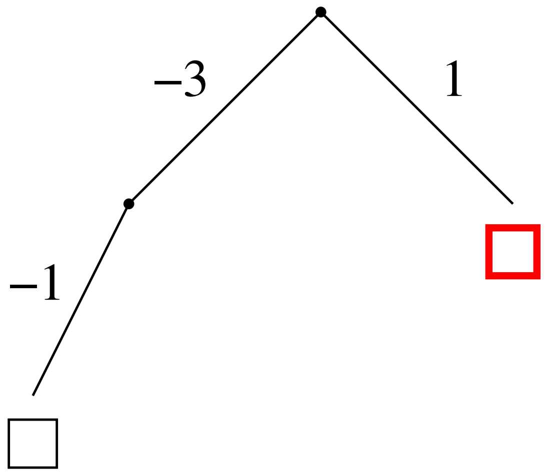
~~$(-1\ -2)$~~

~~$(-1\ 2)$~~

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

$(\cancel{1}\ 2)$

Assignment -3 does not contribute to conflict.



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

$(-3\ 2)$

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

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

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

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

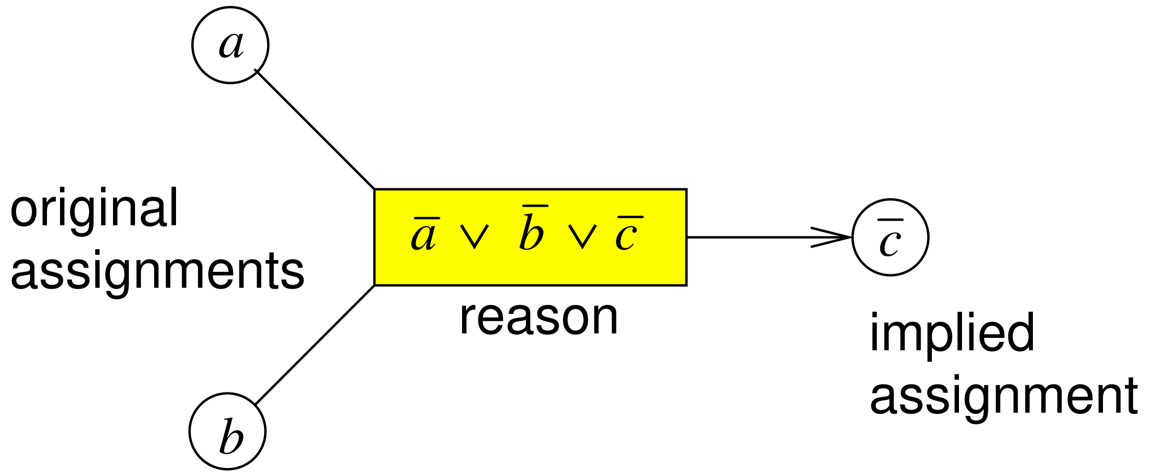
$\cancel{(1\ 2)}$

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

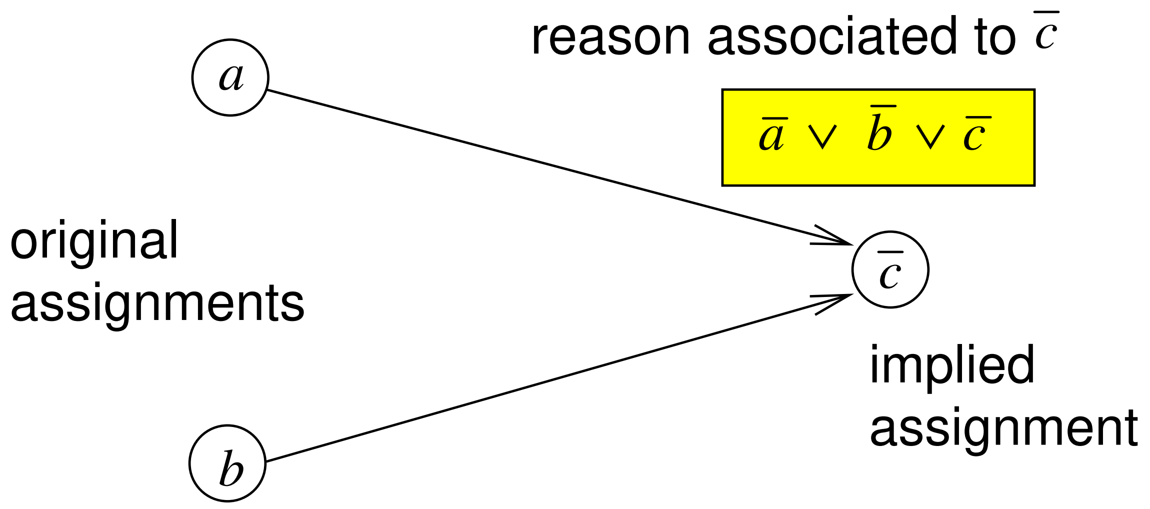
- the implication graph maps inputs to the result of resolutions
- backward from the empty clause all contributing clauses can be found
- the variables in the contributing clauses are contributing to the conflict
- important optimization, since we only use unit resolution
  - generate graph only for 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.

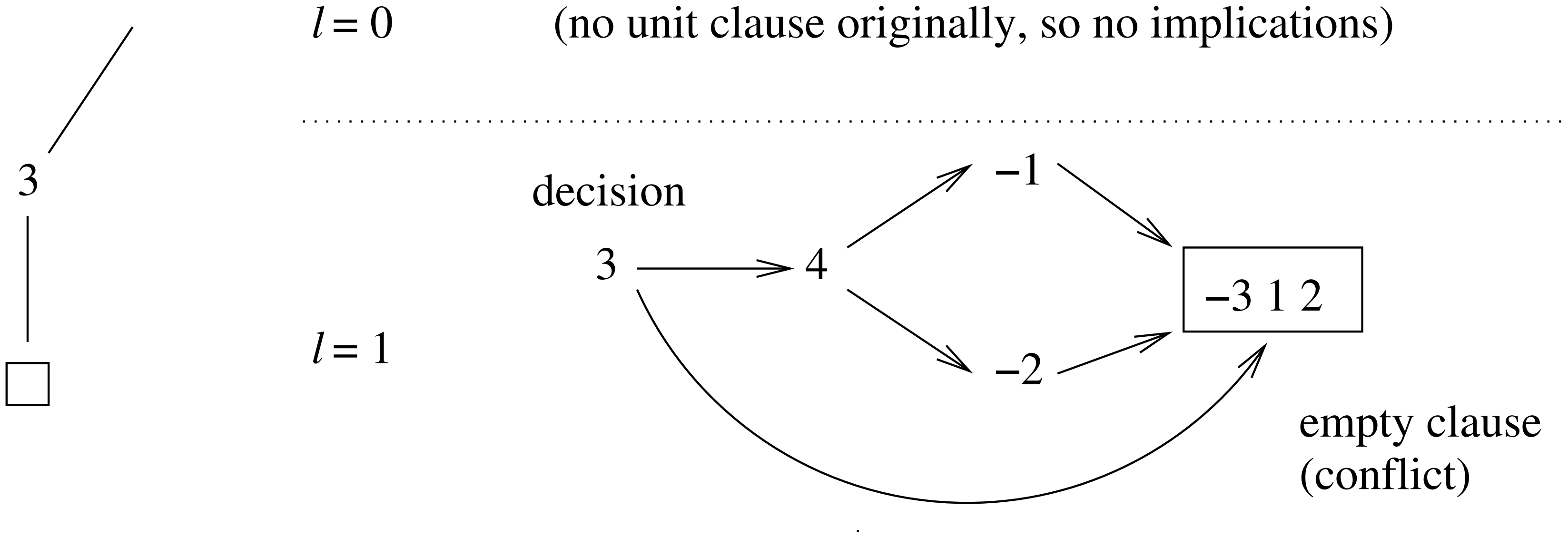
Integers represent literals.

Subtraction means negation.

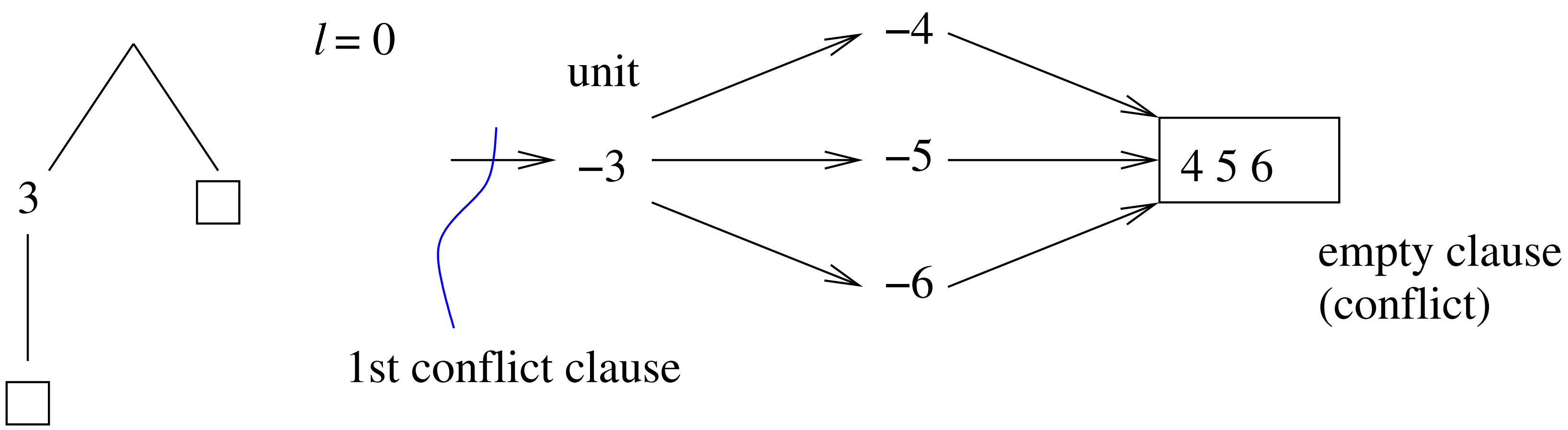
A clause is a zero terminated list of integers.

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)

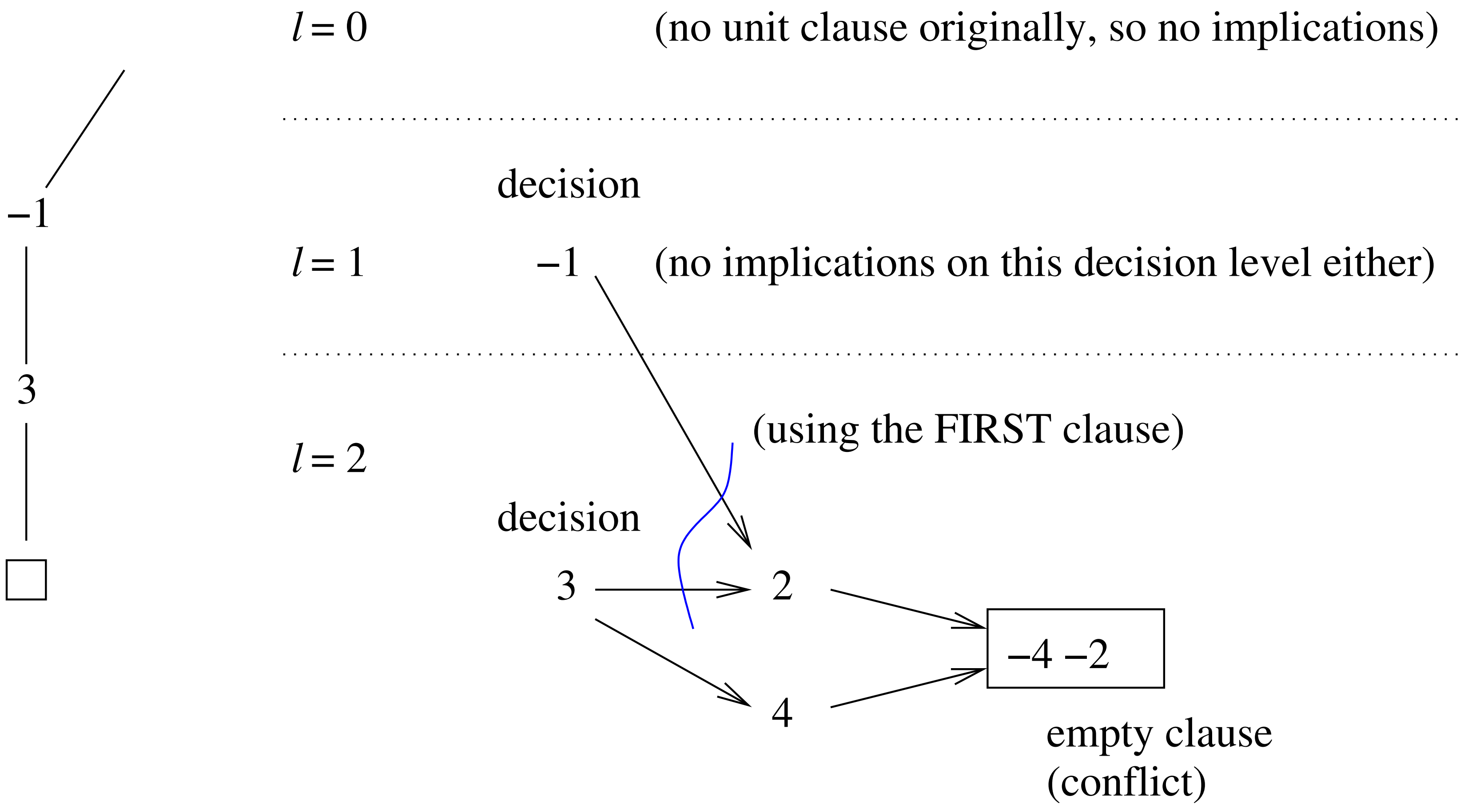


unit clause −3 is generated as learned clause and we backtrackt to $l = 0$



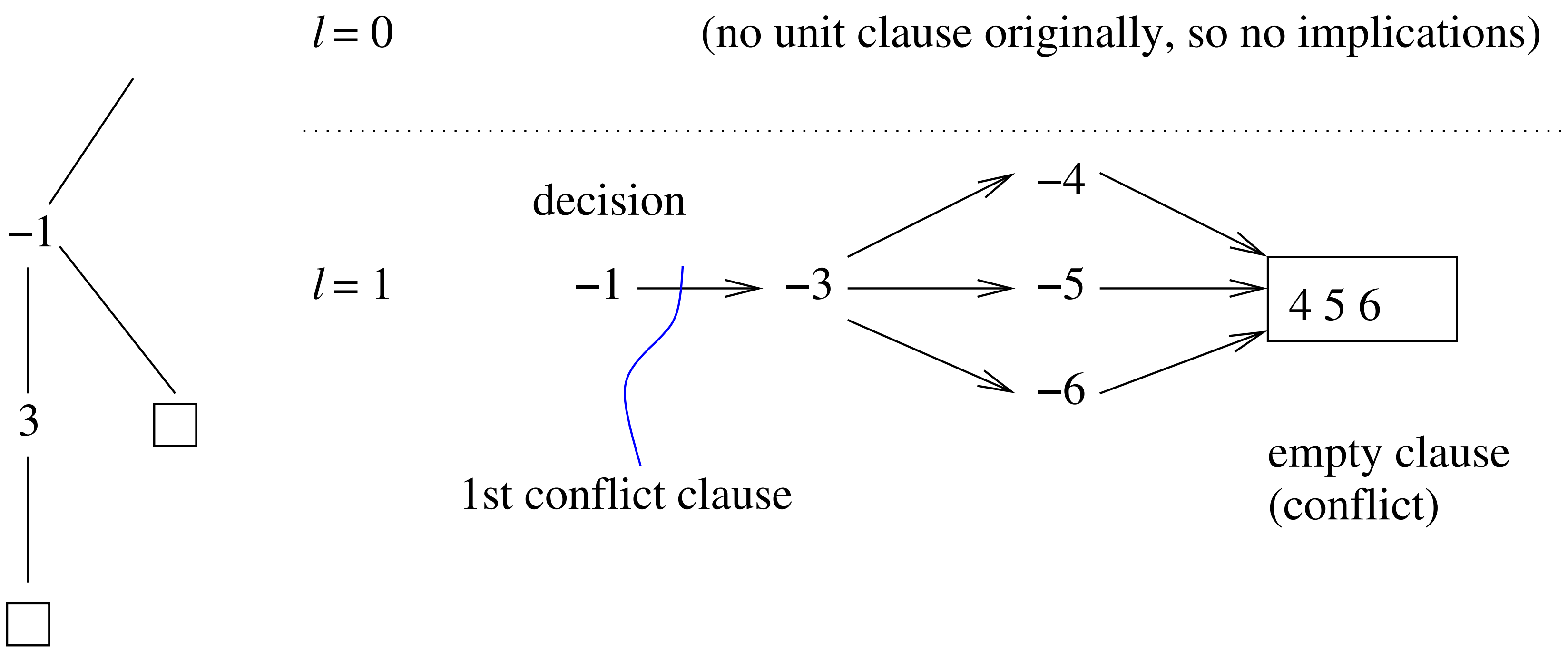
since −3 has a real unit clause as reason, an empty conflict clause is learned

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



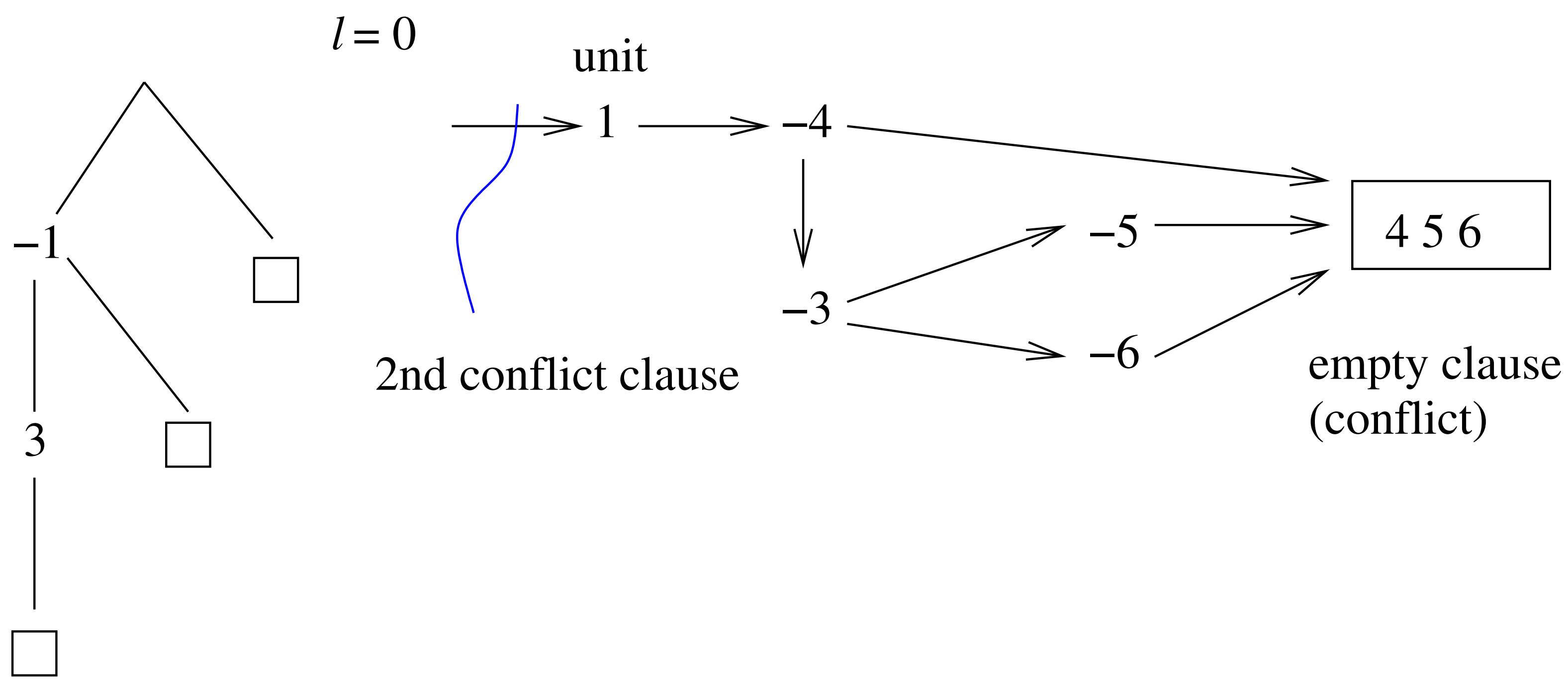
since FIRST clause was used to derive 2, conflict clause is (1 −3)

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



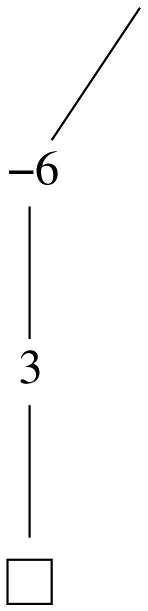
learned conflict clause is the unit clause 1

backtrack to decision level $l = 0$



since the learned clause is the empty clause, conclude unsatisfiability

DP with Learning Run 3 Fig. 1 (-6, 3 as decision order)



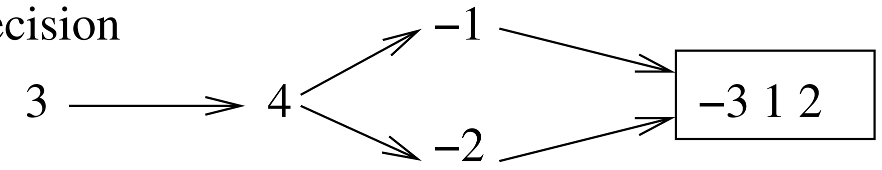
$l = 0$ (no unit clause originally, so no implications)

decision

$l = 1$ −6 (no implications on this decision level either)

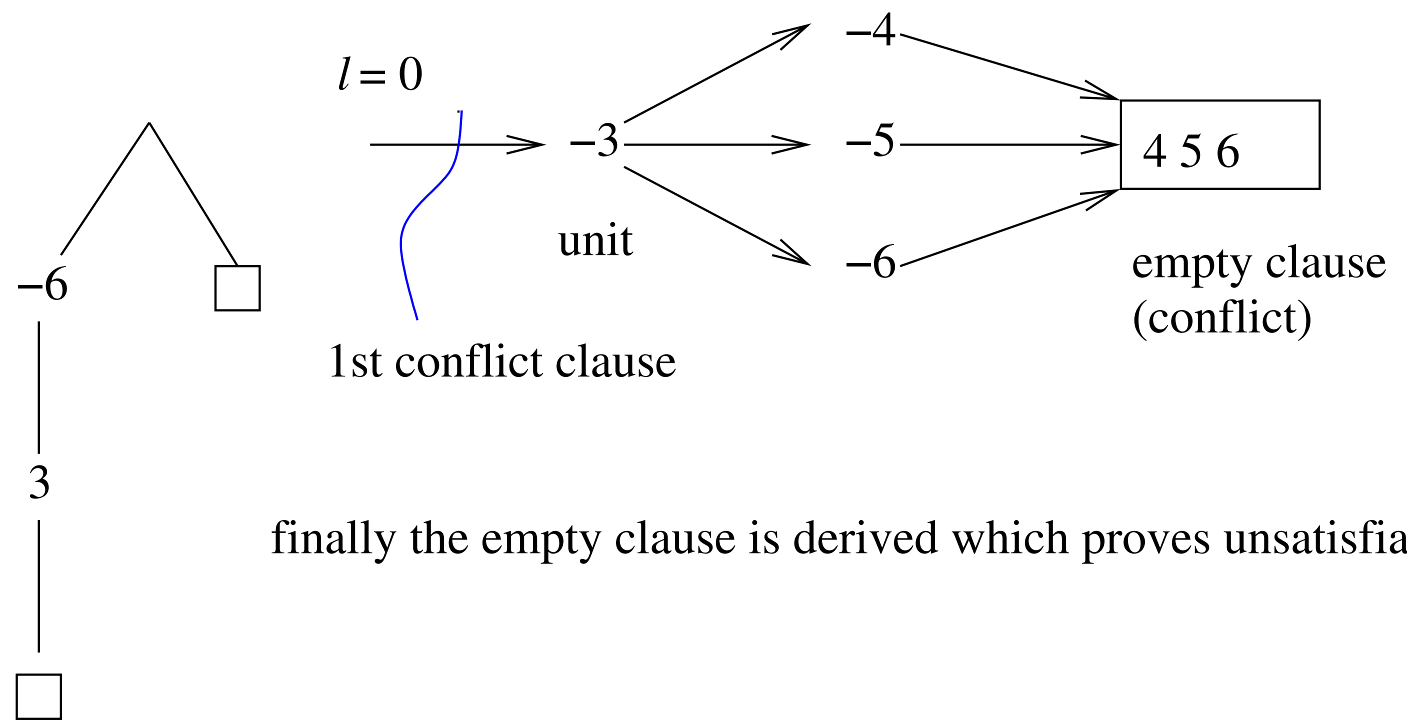
$l = 2$

decision



empty clause
(conflict)

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

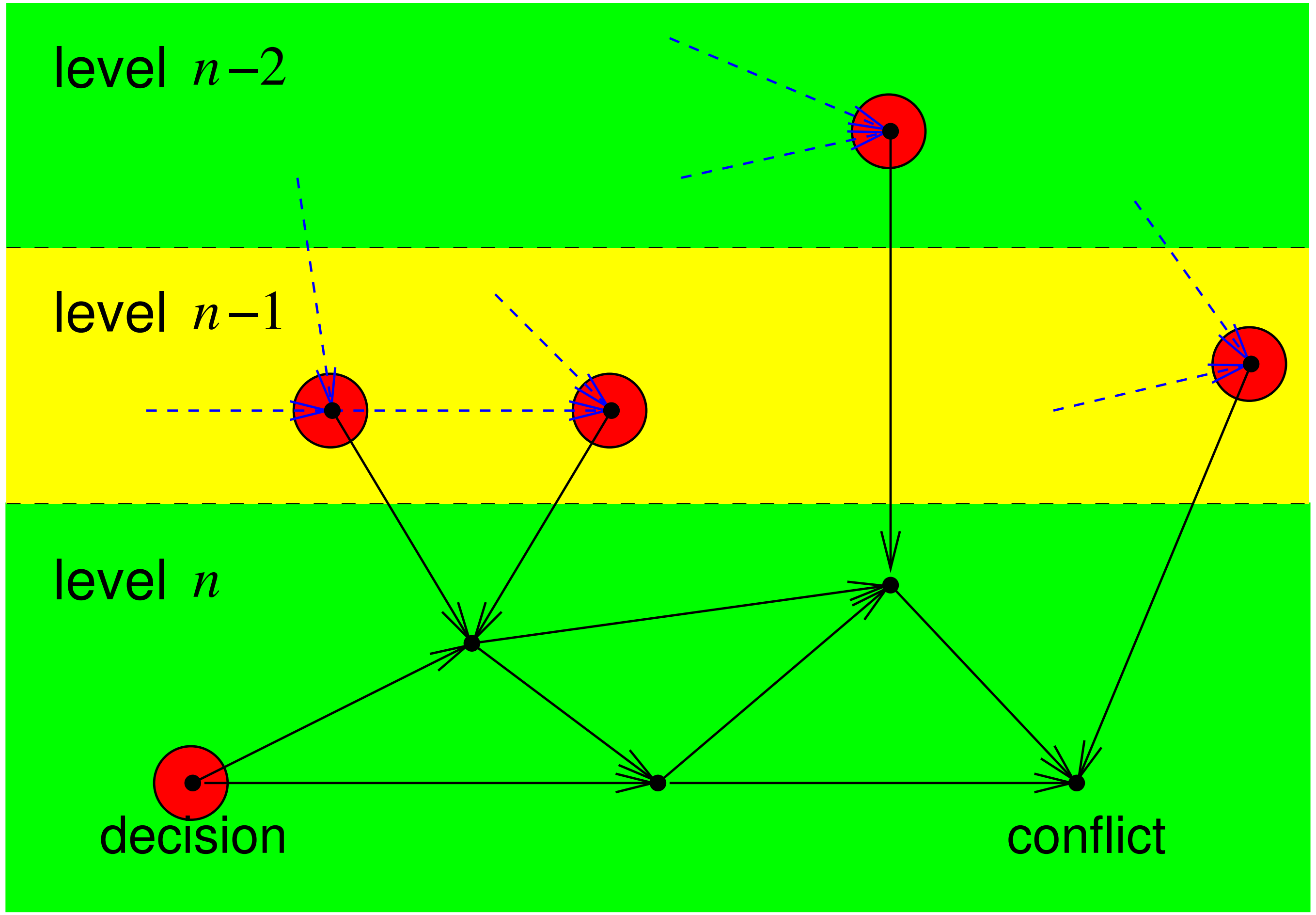


finally the empty clause is derived which proves unsatisfiability

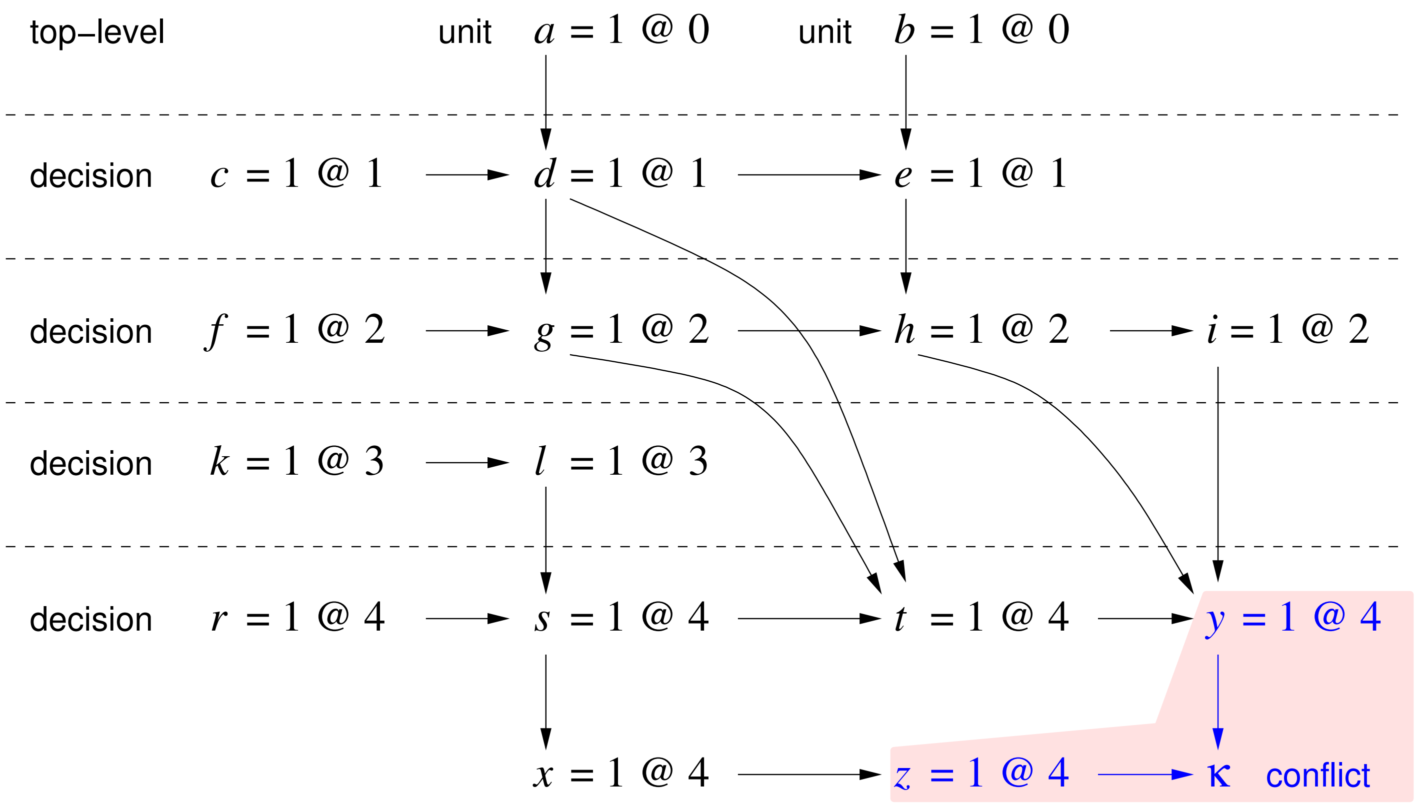
```c
int
sat (Solver solver)
{
  Clause conflict;

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

      conflict = deduce (solver);

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

```
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]



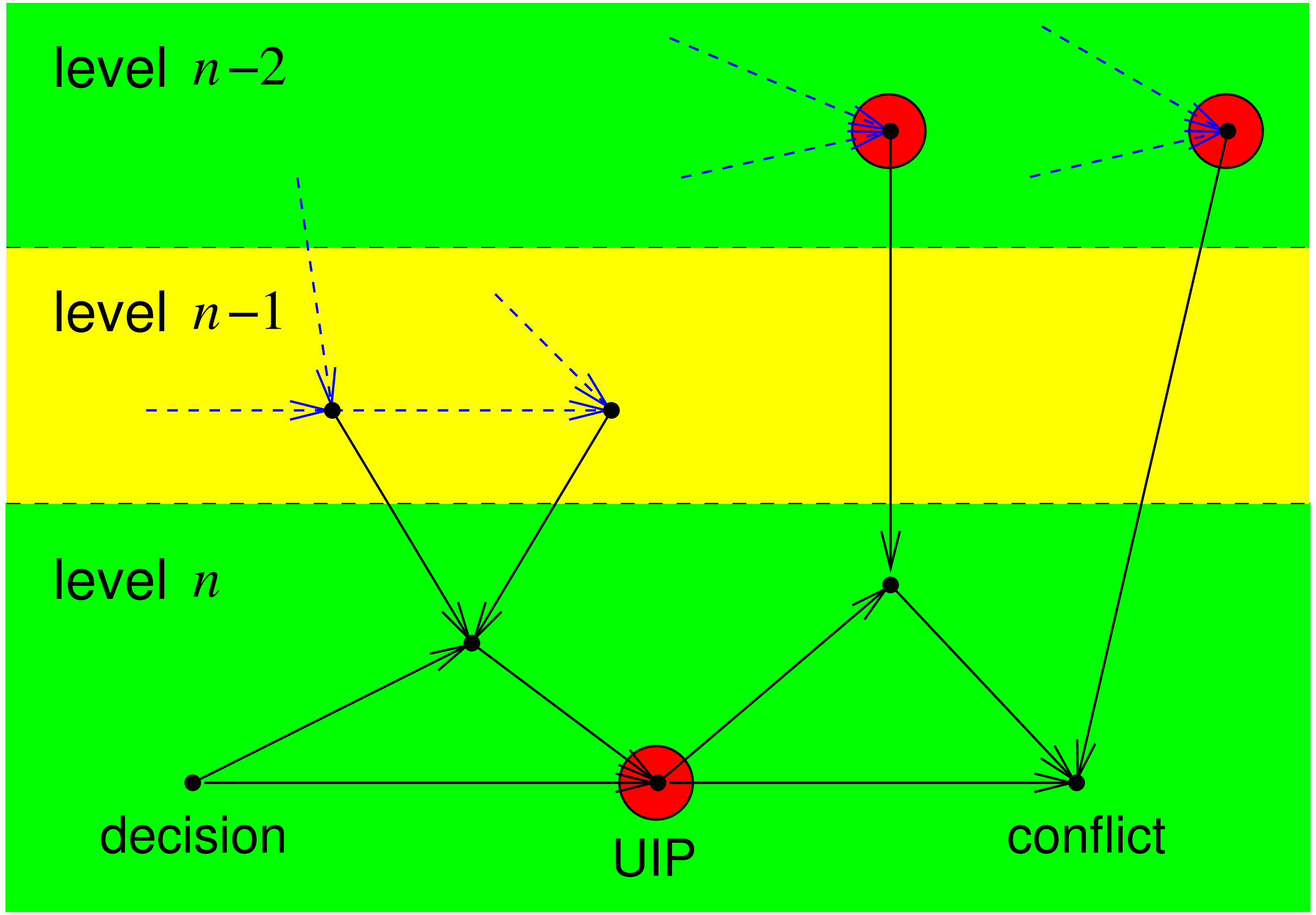
a simple cut always exists: set of roots (decisions) contributing to the conflict



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

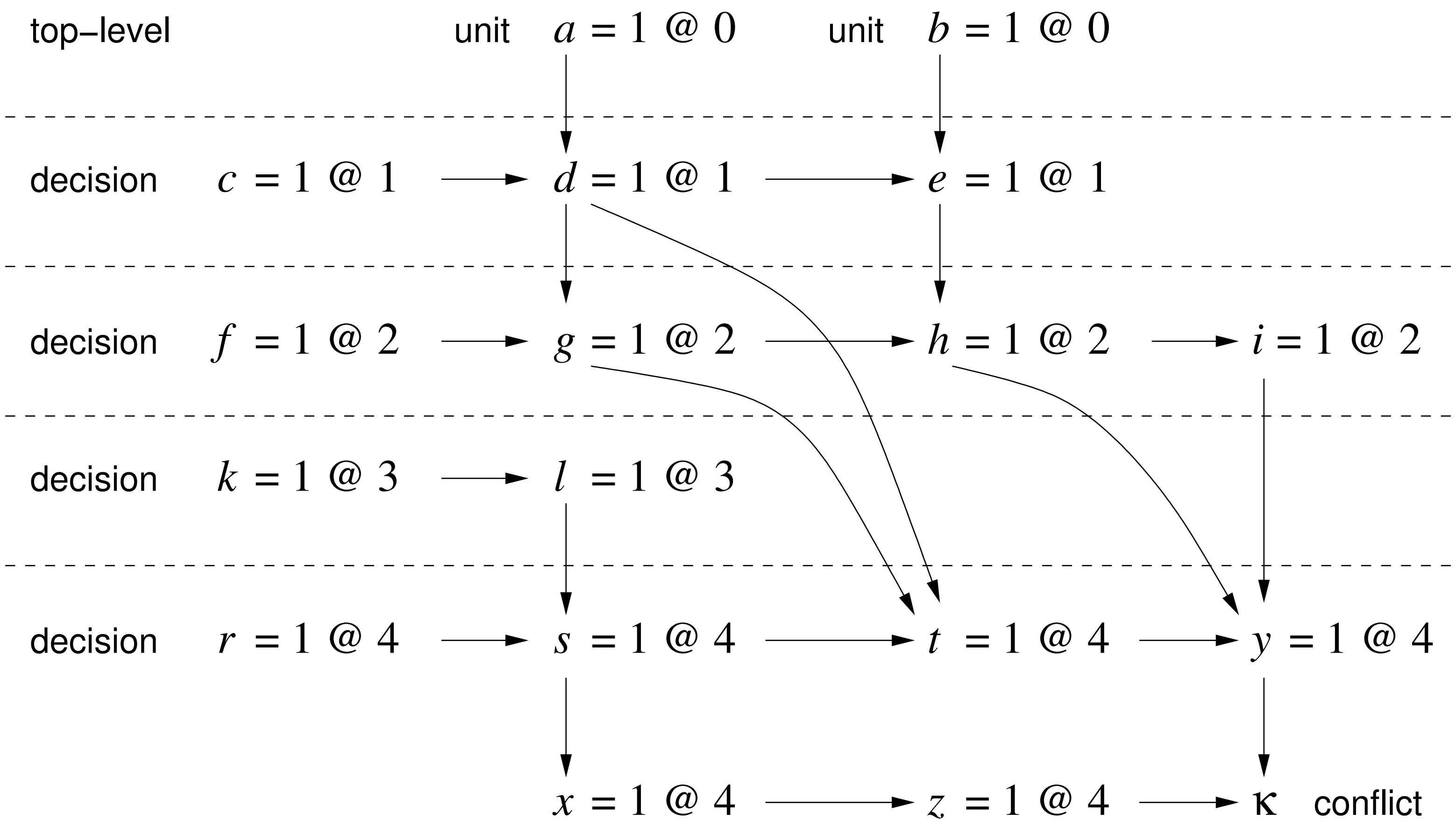
- can be found by graph traversal in the order of made assignments
- *trail* respects this order
- traverse reasons of variables on trail starting with conflict
- count “open paths”
  (initially size of clause with only false literals)
- if all paths converged at one node, then UIP is found
- decision 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

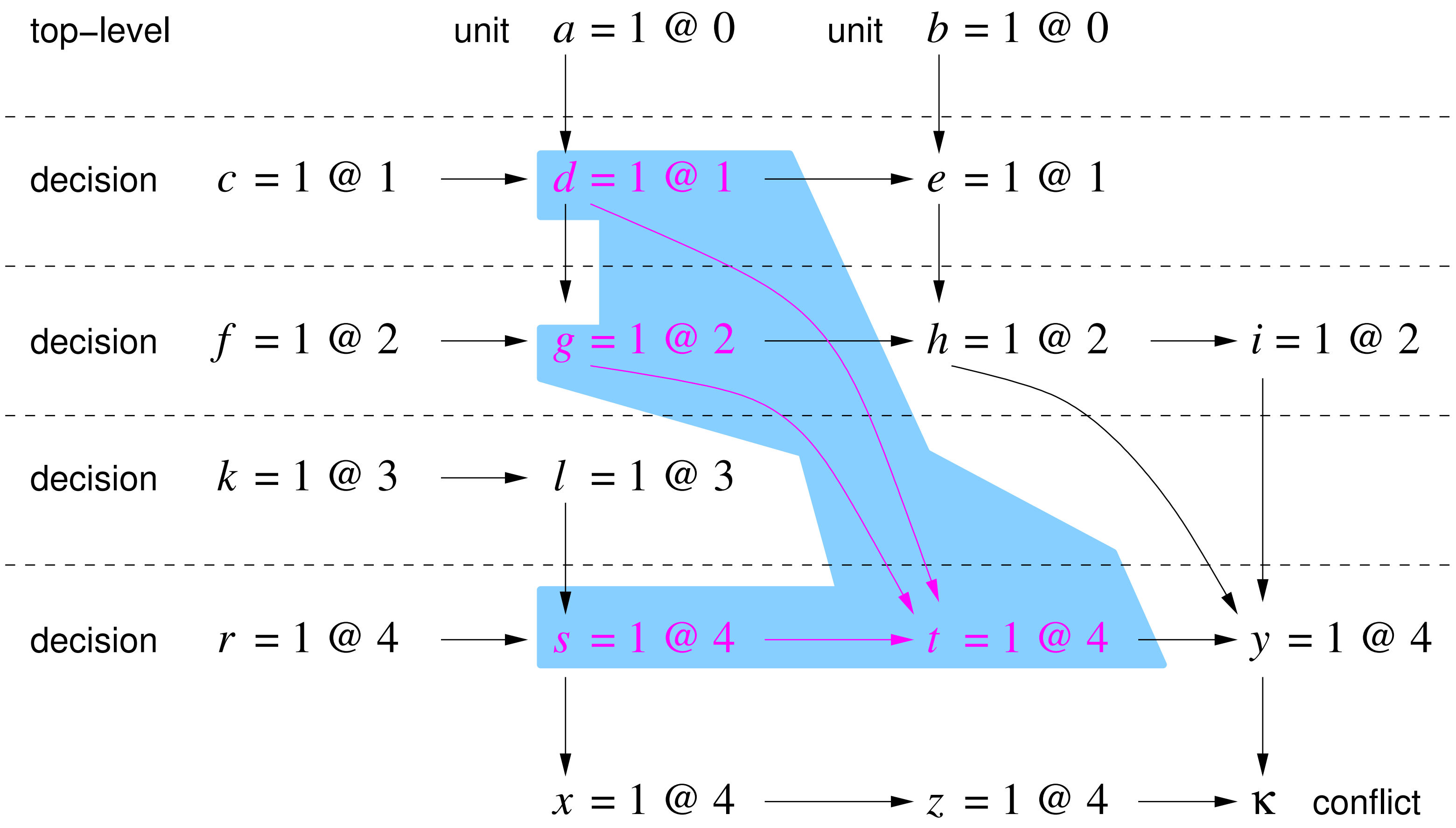


1st UIP learned clause increases chance of backjumping
(“pulls in” as few decision levels as possible)

- intuitively it 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)

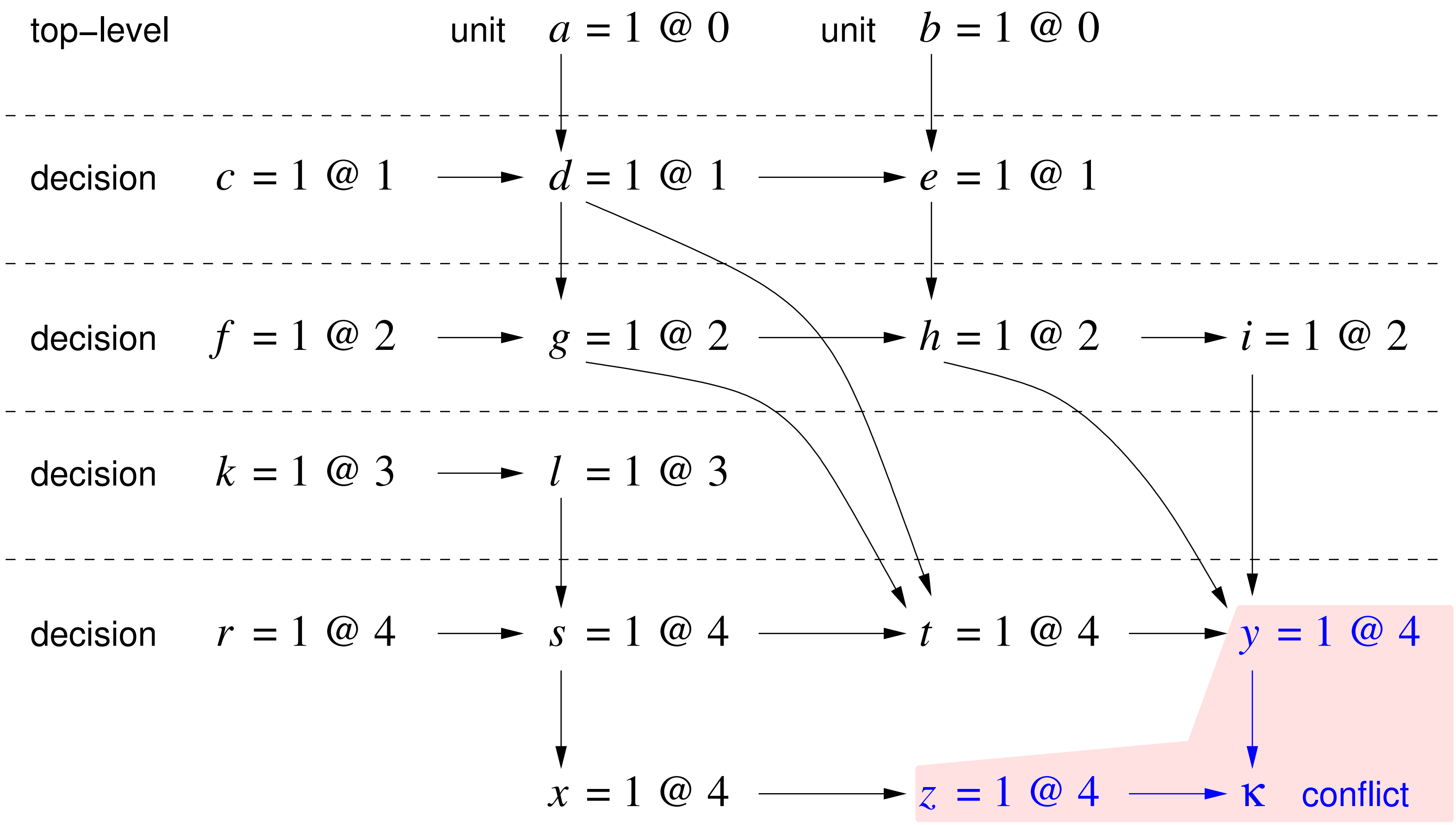

top−level
unit $a = 1 @ 0$
unit $b = 1 @ 0$
decision $c = 1 @ 1$
$d = 1 @ 1$
$e = 1 @ 1$
decision $f = 1 @ 2$
$g = 1 @ 2$
$h = 1 @ 2$
$i = 1 @ 2$
decision $k = 1 @ 3$
$l = 1 @ 3$
decision $r = 1 @ 4$
$s = 1 @ 4$
$t = 1 @ 4$
$y = 1 @ 4$
$x = 1 @ 4$
$z = 1 @ 4$
κ conflict

Antecedents / Reasons



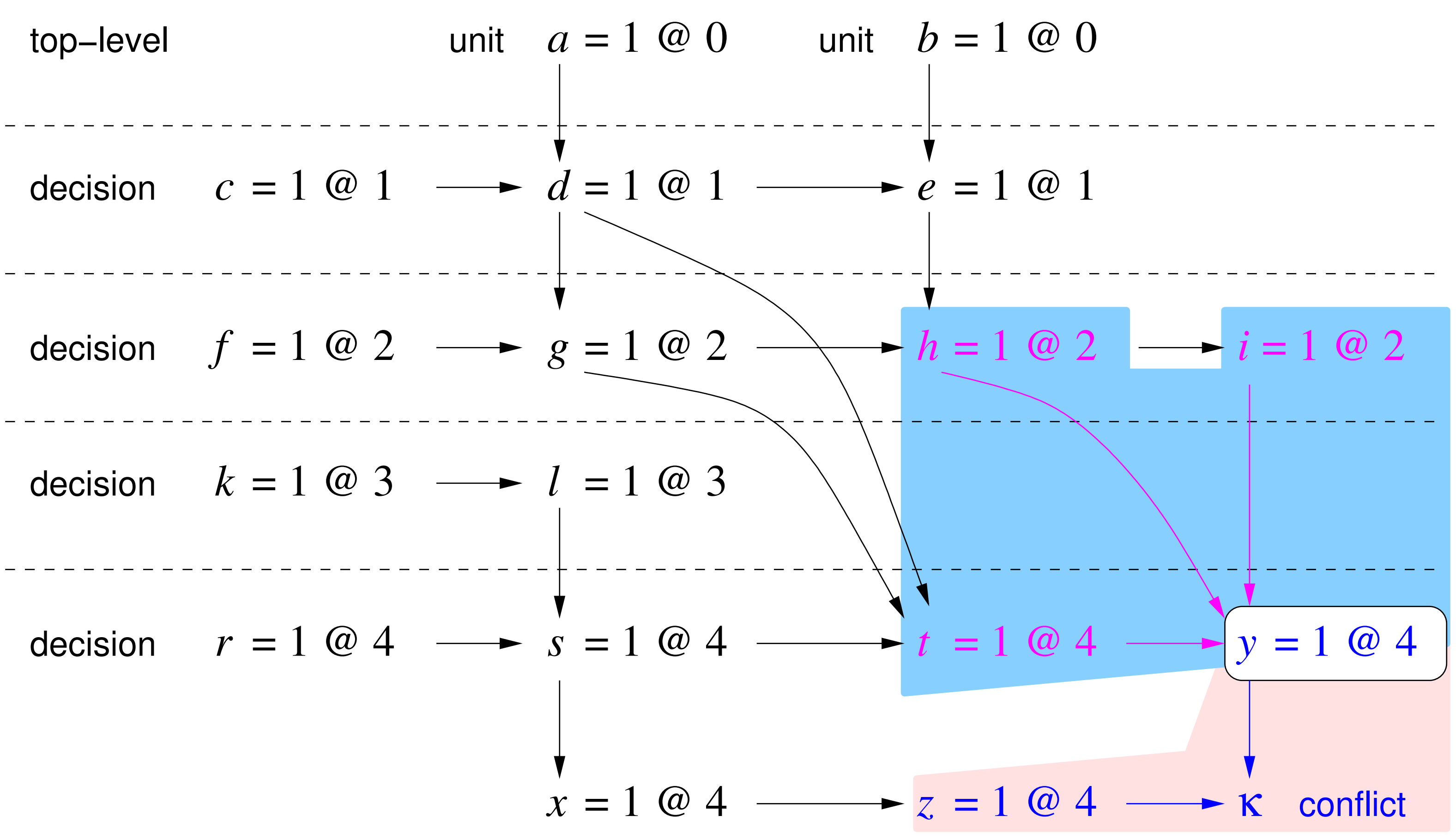
$$d \land g \land s \rightarrow t \qquad \equiv \qquad (\bar{d} \lor \bar{g} \lor \bar{s} \lor t)$$

Conflicting Clauses

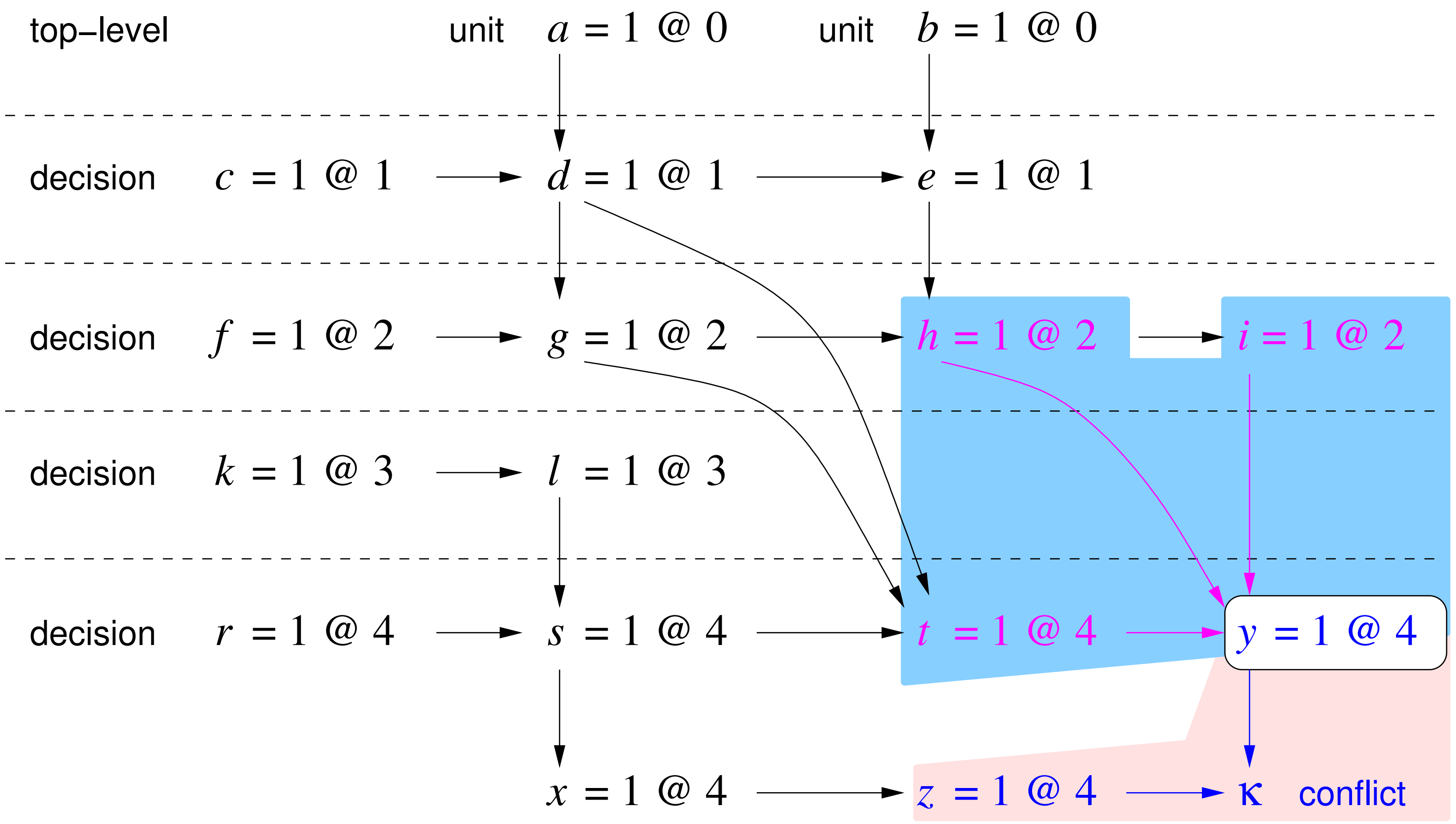


$$\neg(y \wedge z) \qquad \equiv \qquad (\bar{y} \vee \bar{z})$$

Resolving Antecedents 1st Time

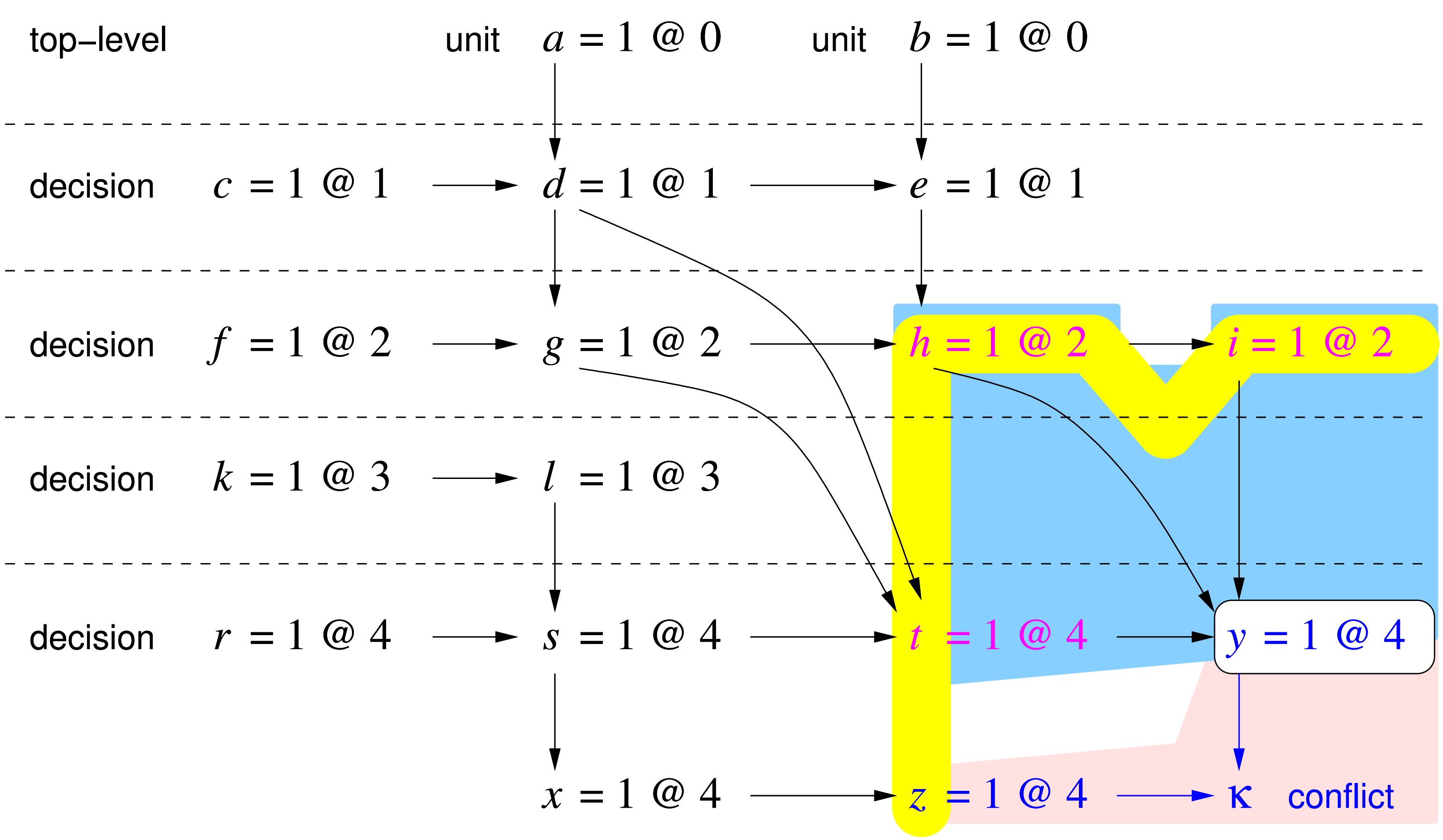


$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $\qquad$ $(\bar{y} \vee \bar{z})$



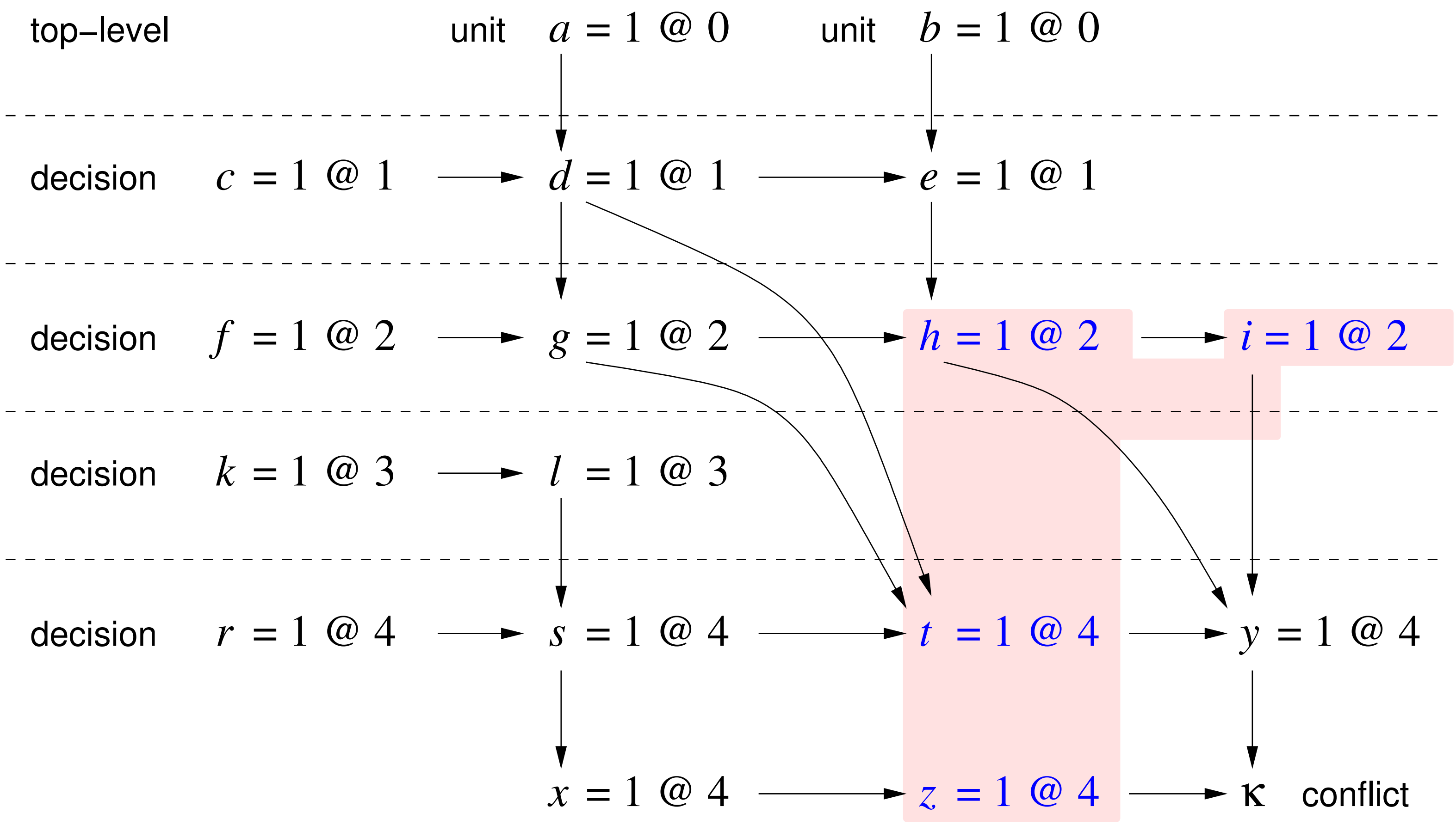
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Resolvents = Cuts = Potential Learned Clauses

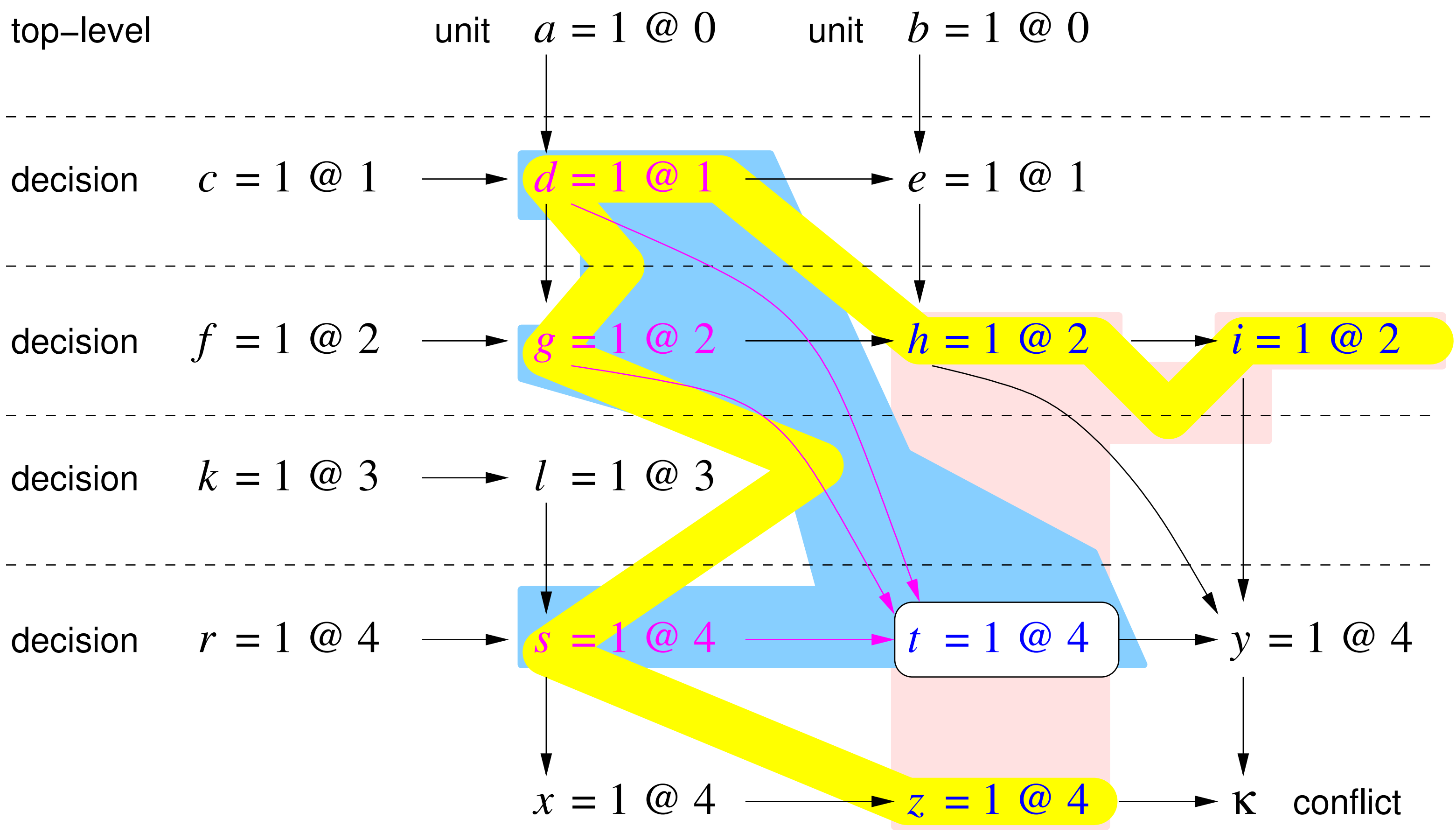


$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Potential Learned Clause After 1 Resolution

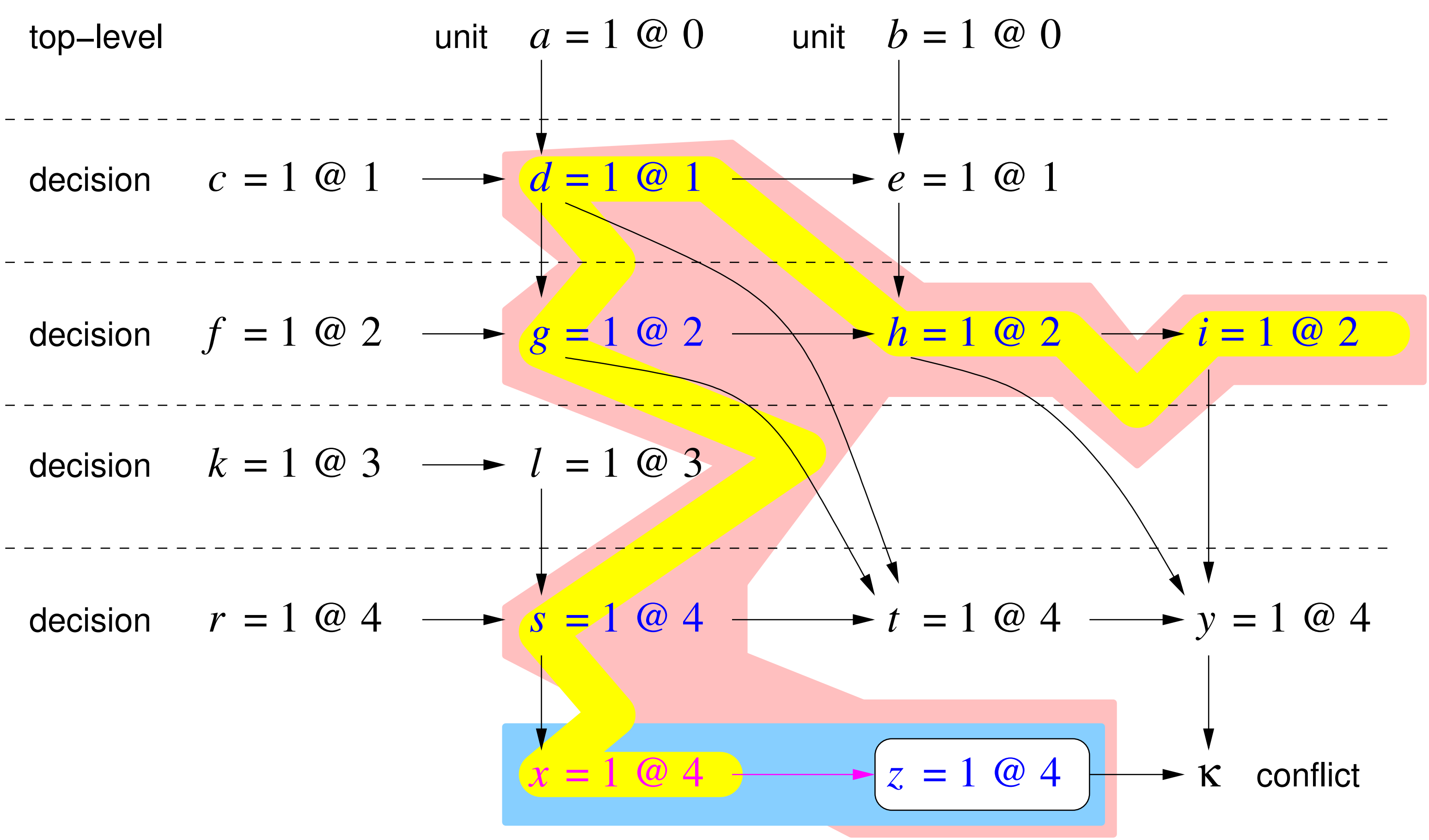


$$(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})$$



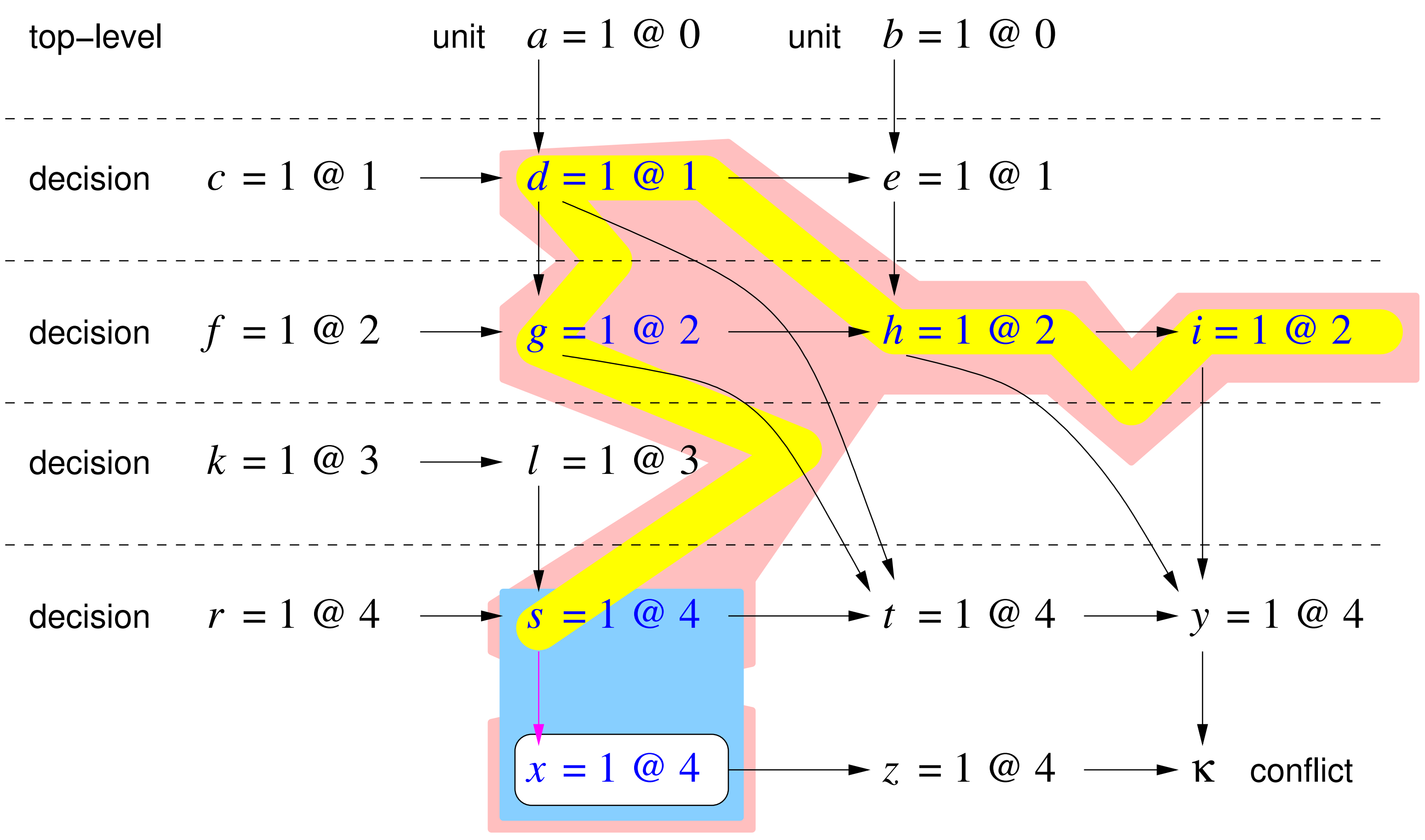
$$\frac{(\bar{d} \vee \bar{g} \vee \bar{s} \vee t) \qquad (\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}$$

Resolving Antecedents 3rd Time



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

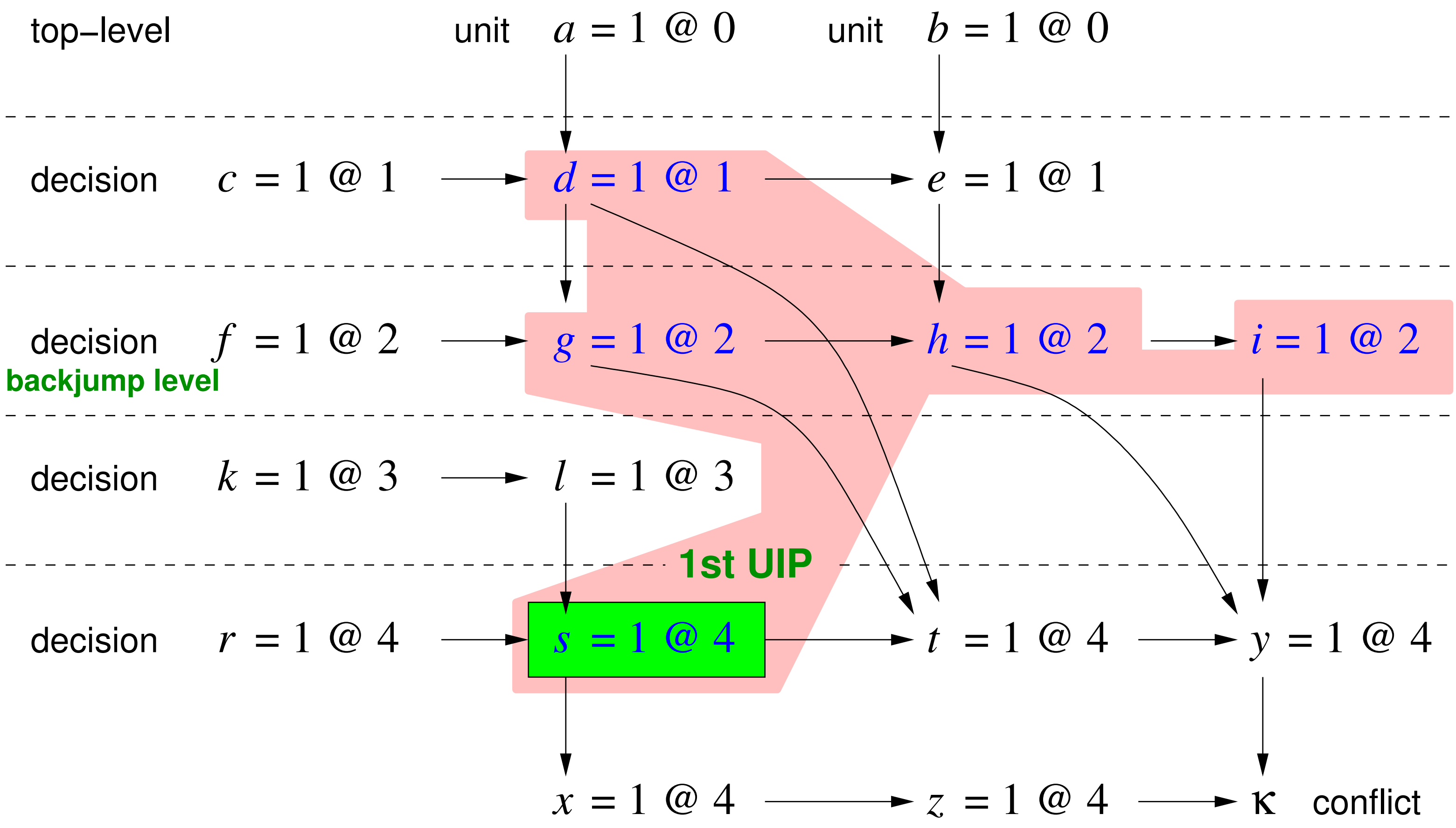
Resolving Antecedents 4th Time



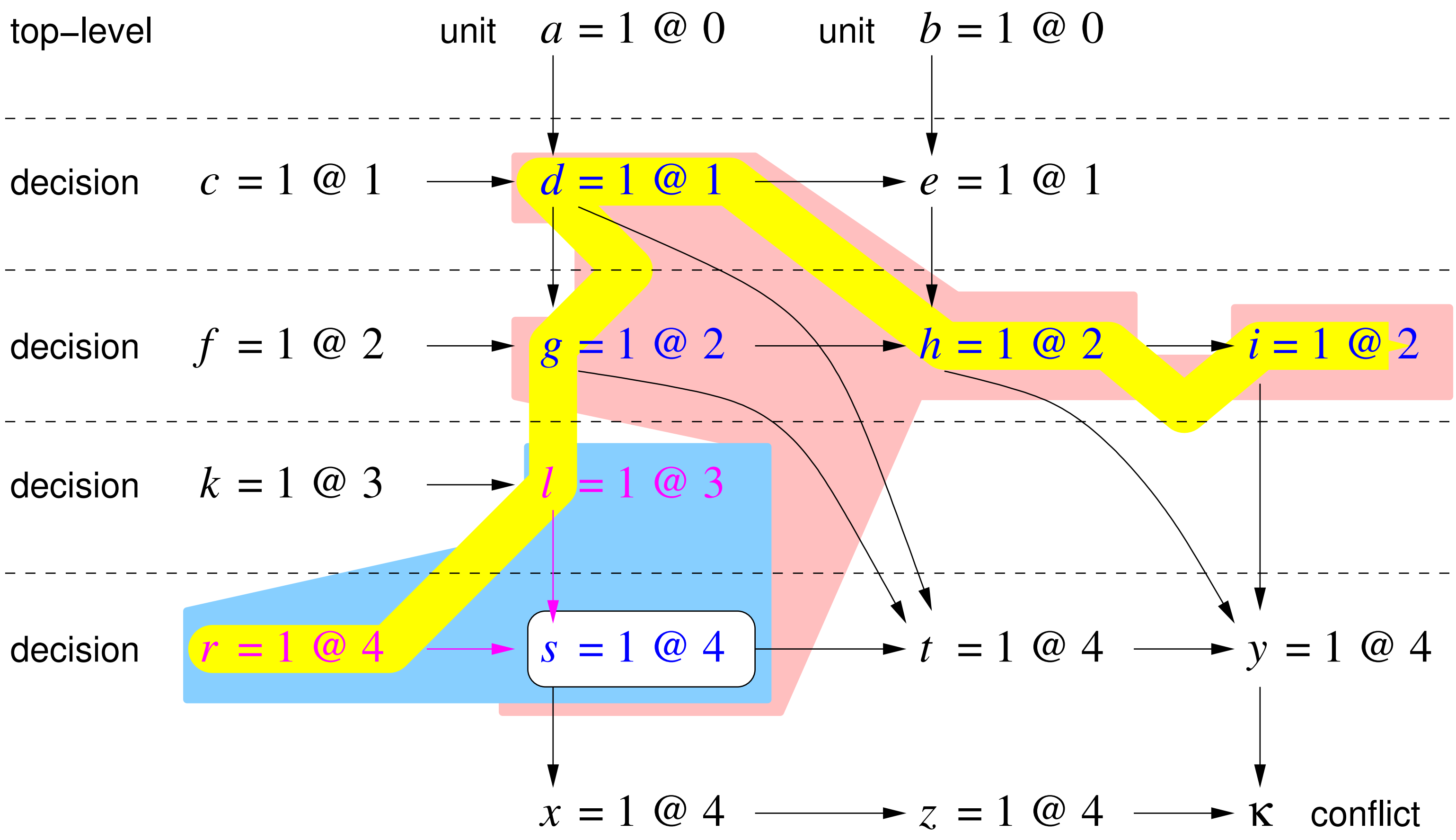
$$\frac{(\bar{s} \vee x) \qquad (\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

self subsuming resolution

1st UIP Clause after 4 Resolutions

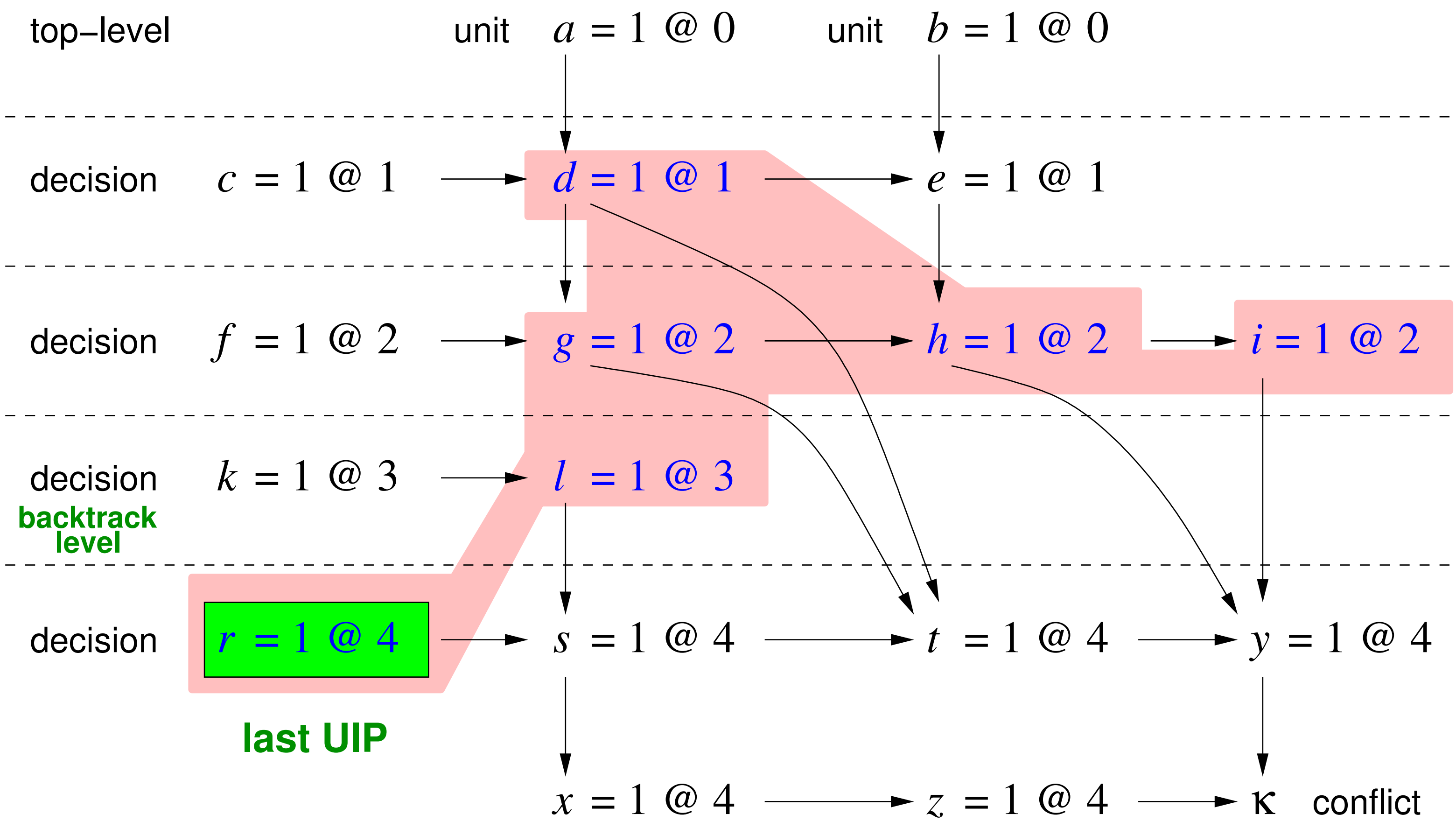


$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$



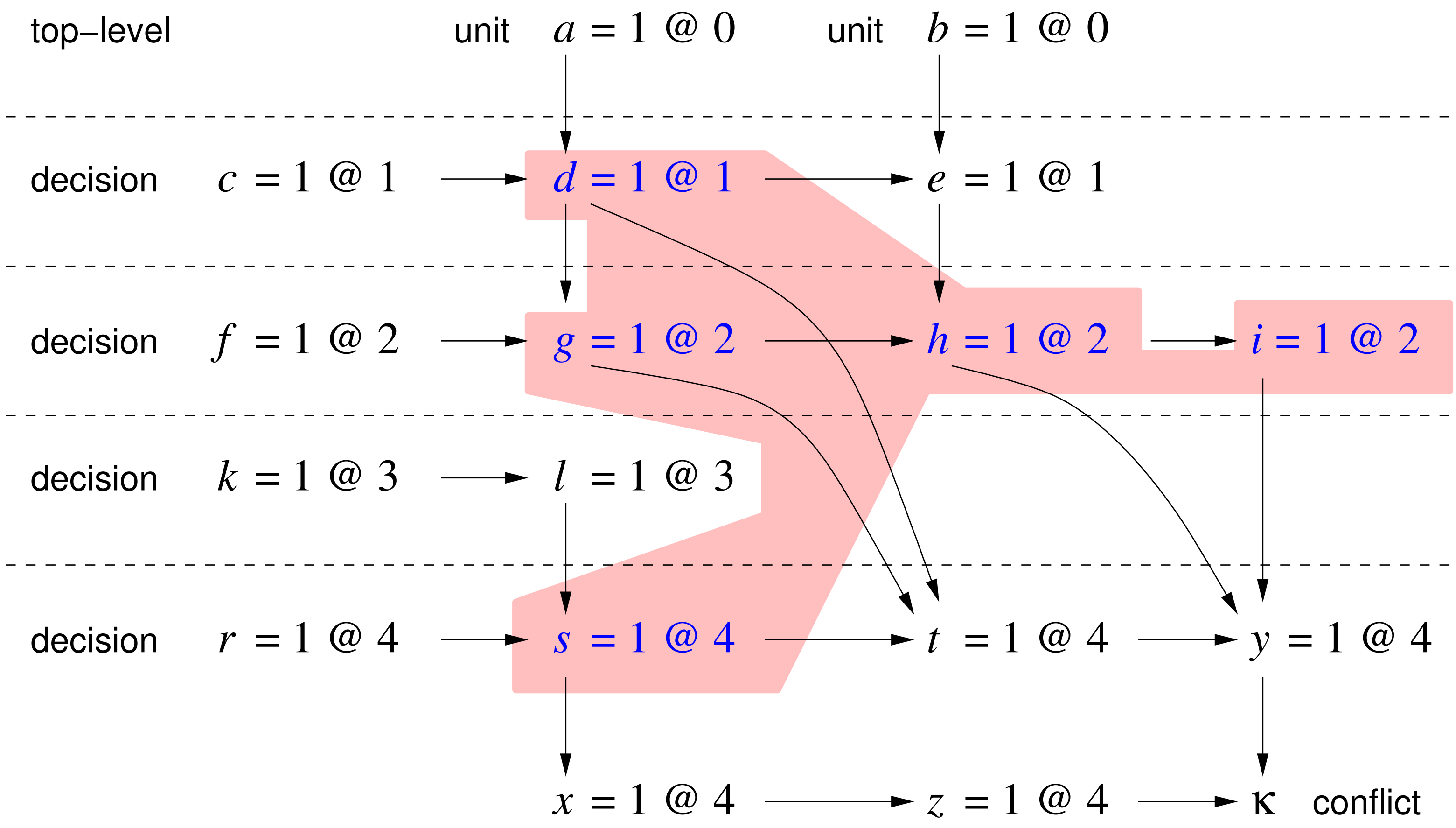
$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

Decision Learned Clause



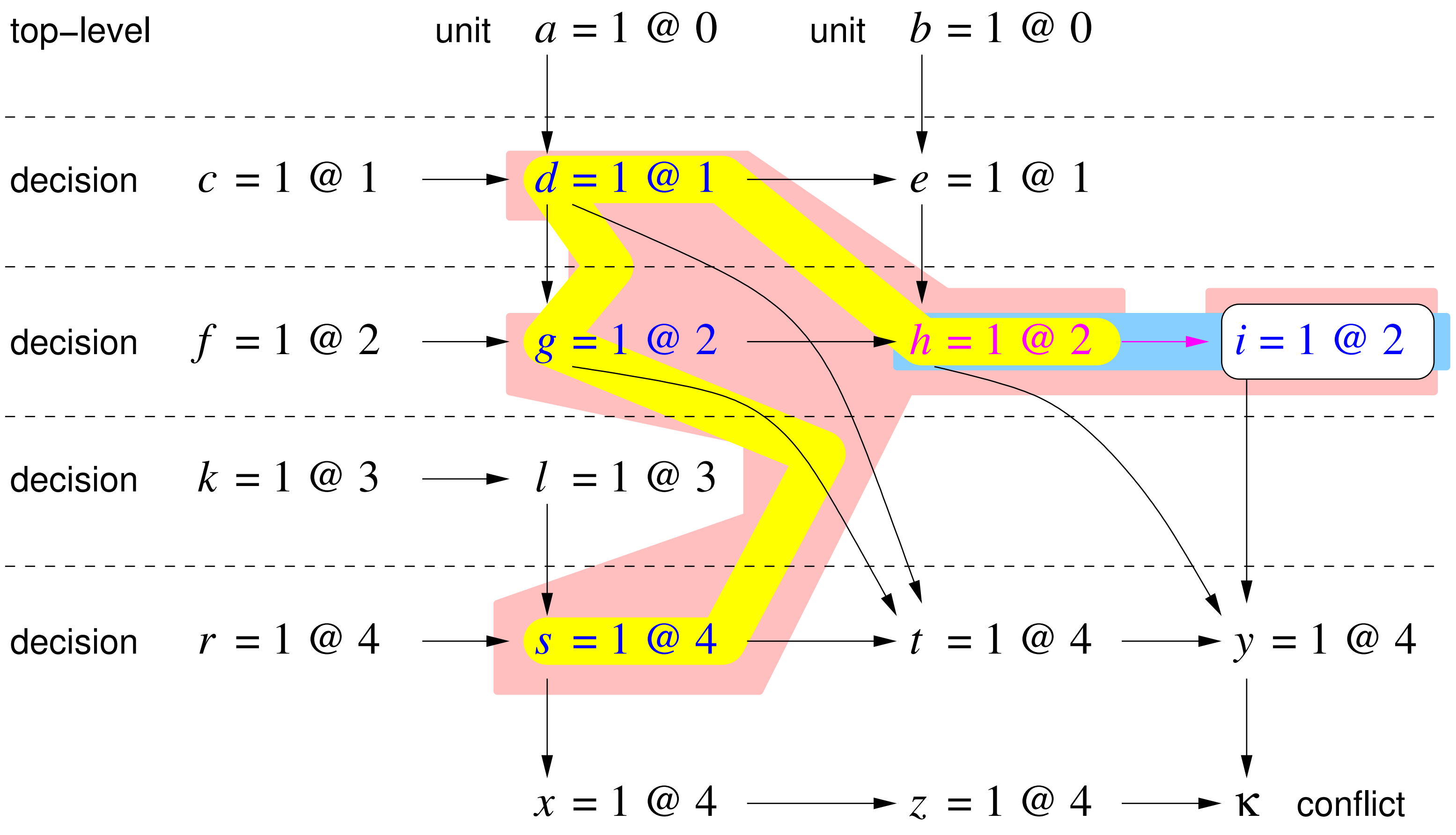
$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

1st UIP Clause after 4 Resolutions



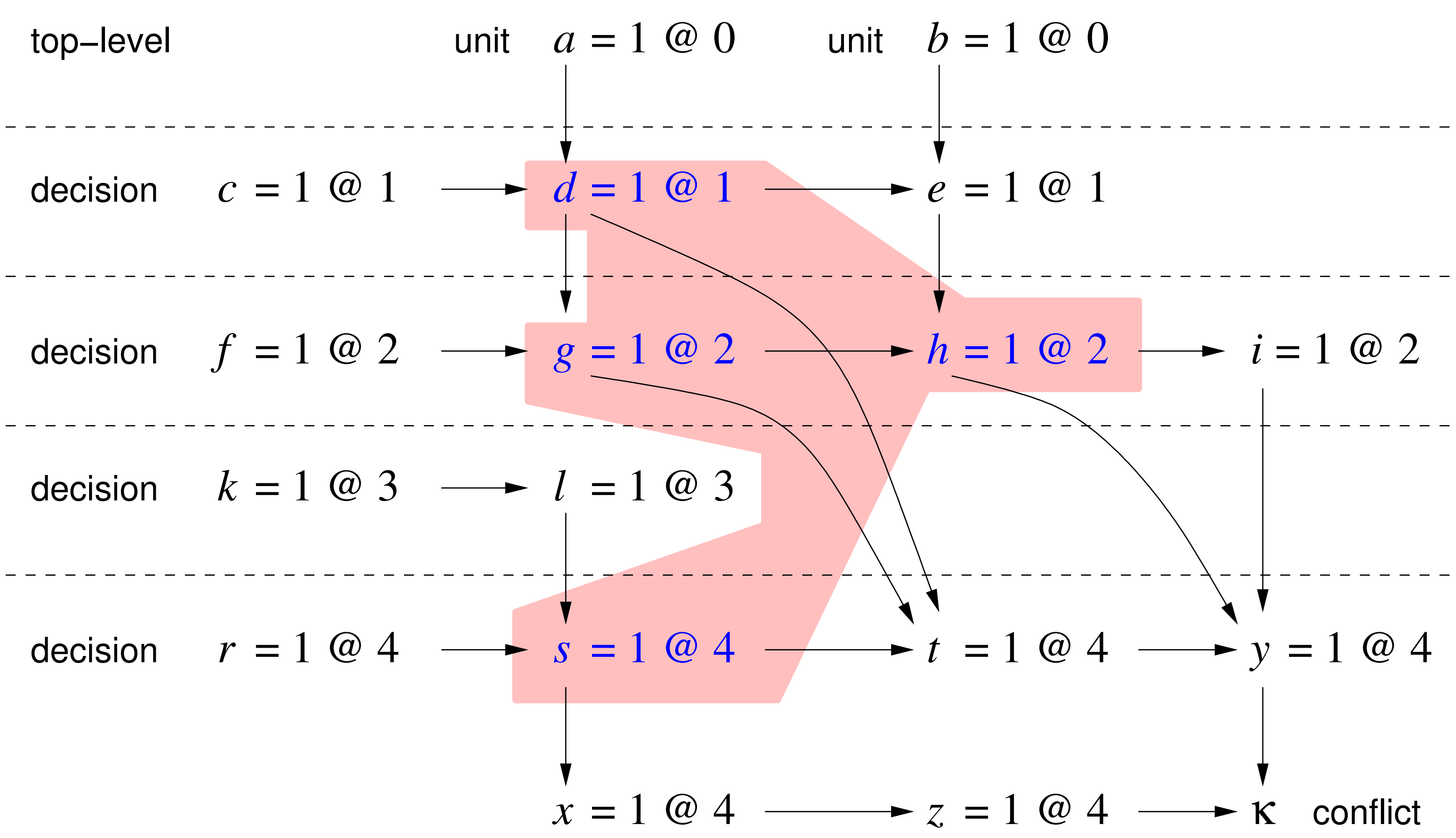
$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

Locally Minimizing 1st UIP Clause



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

self subsuming resolution



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

Two step algorithm:

1. mark all variables in 1st UIP clause

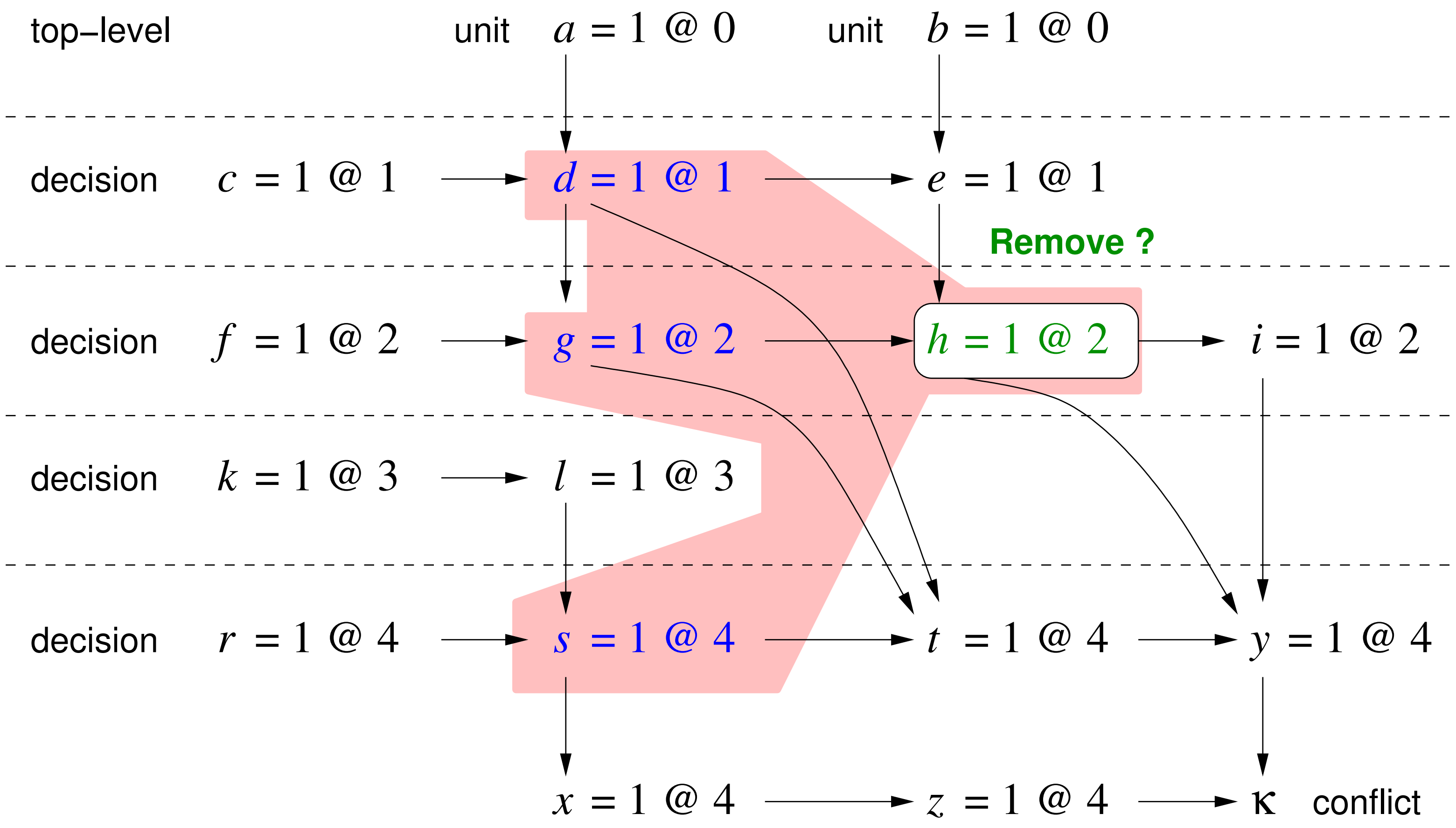
2. remove literals with all antecedent literals also marked

Correctness:

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

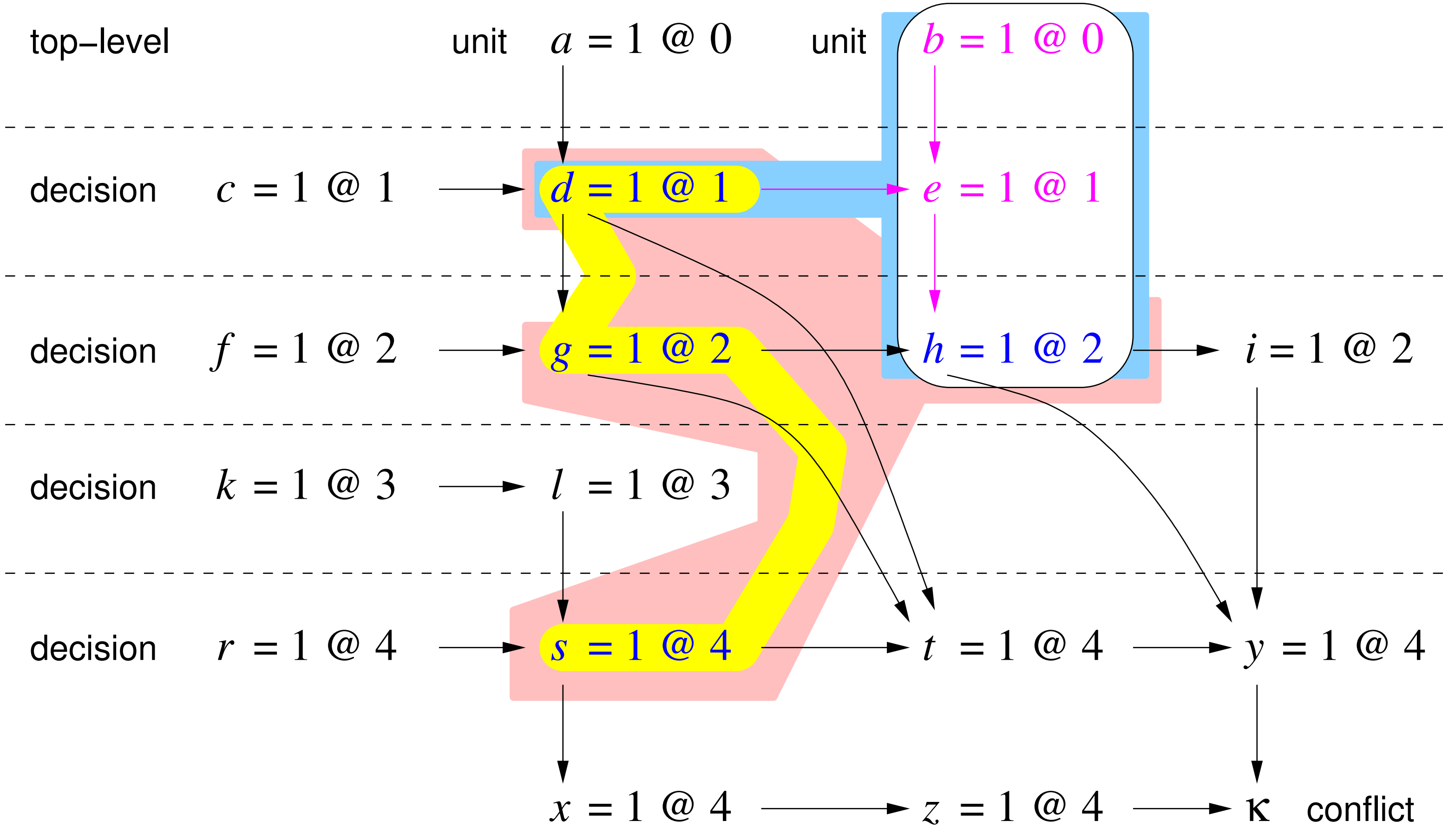
Confluence: produces a unique result.

Minimizing Locally Minimized Learned Clause Further?



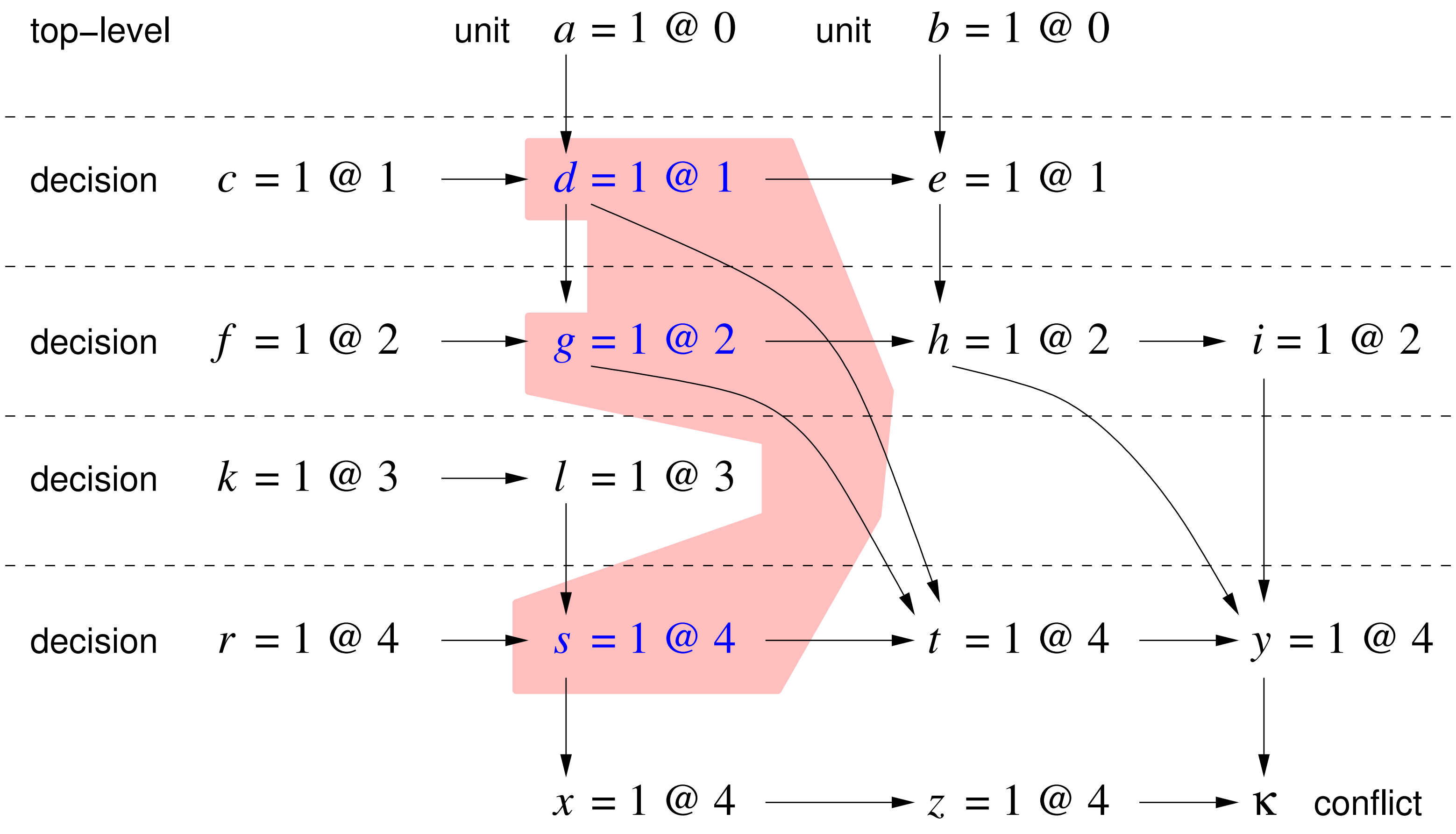
$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$

Recursively Minimizing Learned Clause



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

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

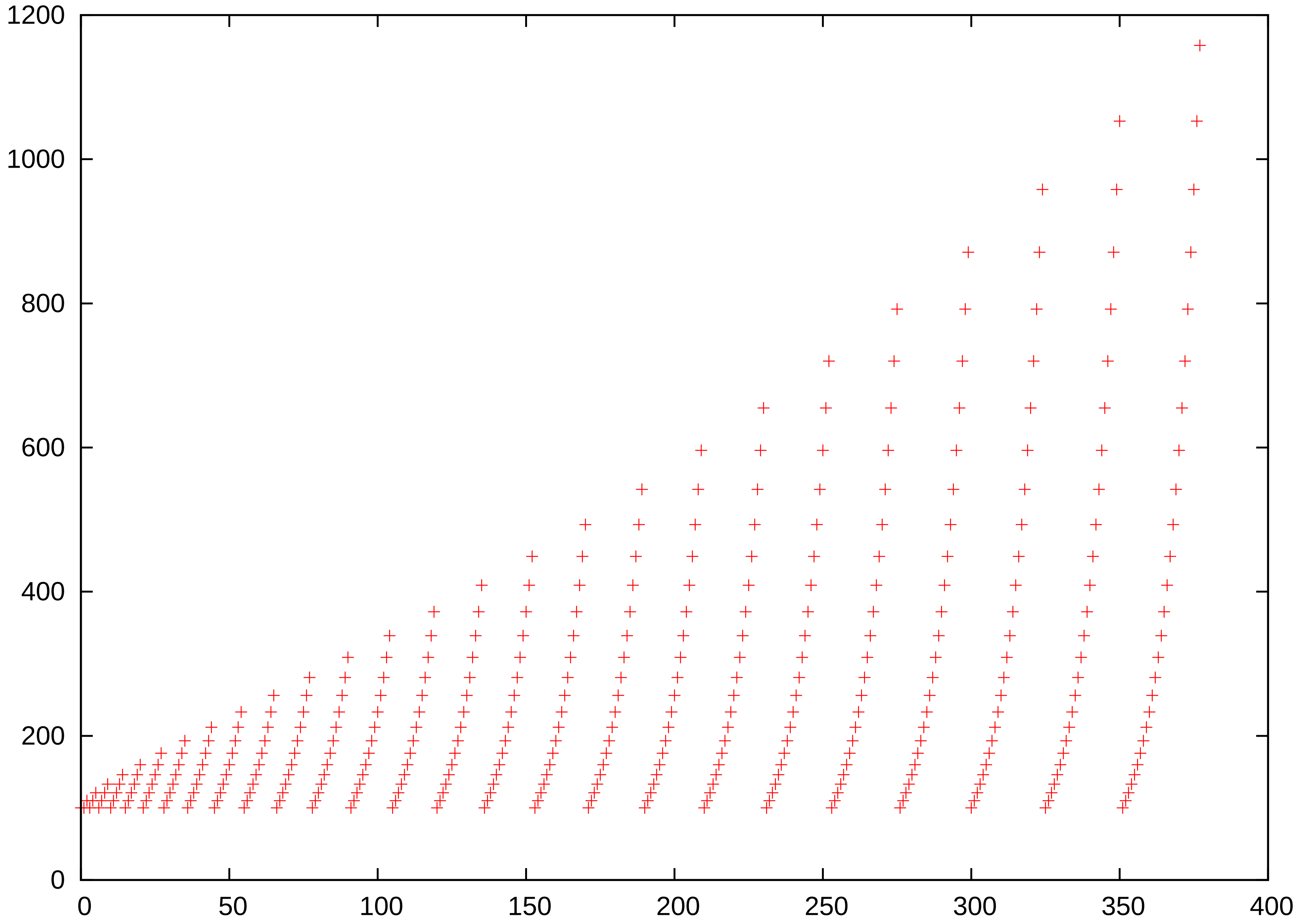
Large Variance for Different Seeds

		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%
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

- often it is a good strategy to abandon what you do and restart
  - for satisfiable instances the solver may get stuck in the unsatisfiable part
  - for unsatisfiable instances focusing on one part might miss short proofs
  - restart after the number of conflicts 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 during restart
- for completeness dynamically increase restart limit
  - arithmetically, geometrically, Luby, Inner/Outer
- Glucose restarts [AudemardSimon-CP'12]
  - short vs. large window *exponential moving average* (EMA) over LBD
  - if recent LBD values are larger than long time average then restart
- interleave "stabilizing" (no restarts) and "non-stabilizing" phases [Chanseok Oh]

Inner/Outer Restart Intervals

378 restarts in 104408 conflicts

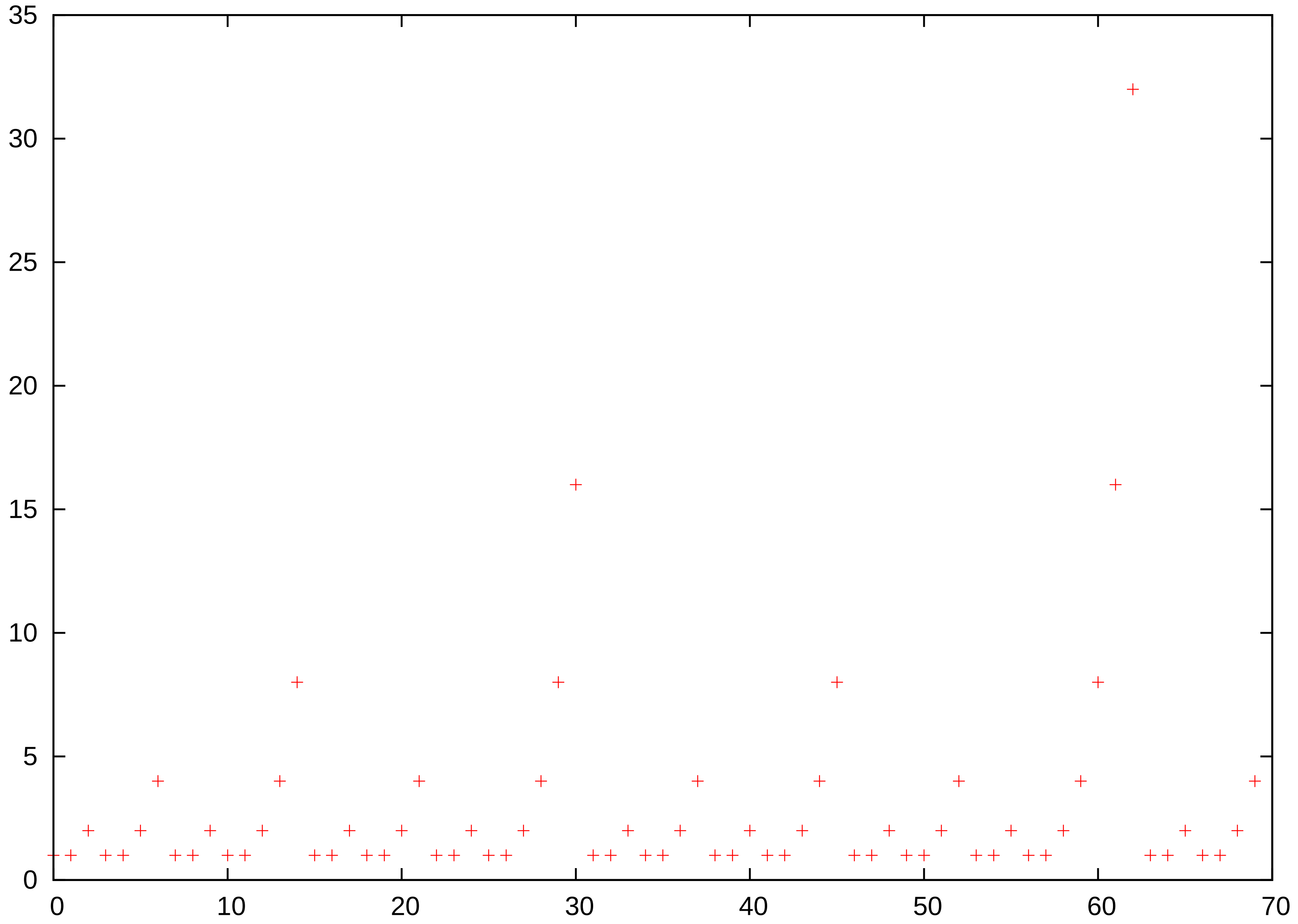


Inner/Outer Restart Scheduling


```
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



```
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

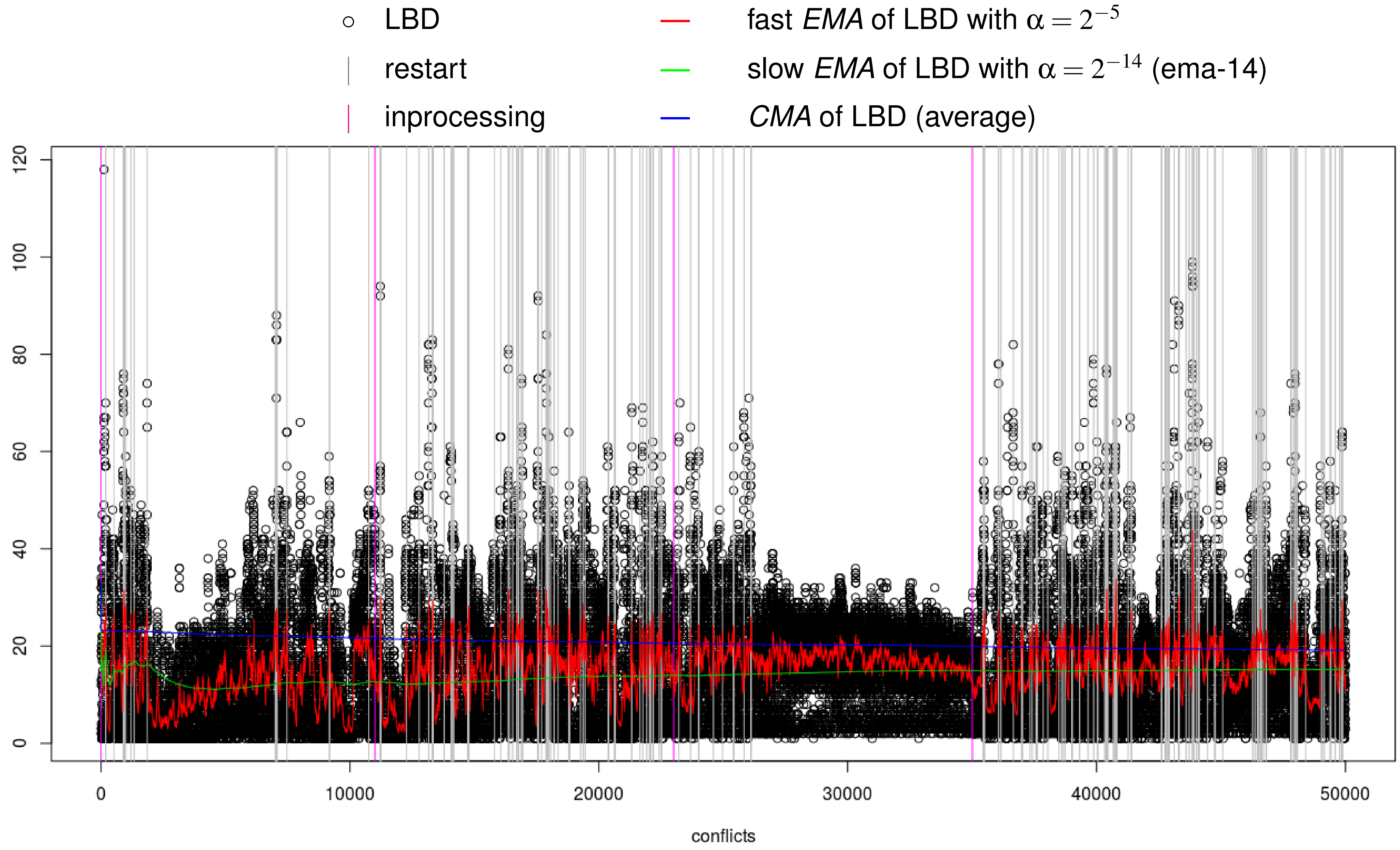
[Knuth'12]

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

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

Restart Scheduling with Exponential Moving Averages (EMA)

[BiereFröhlich-POS'15]



```
double fast, slow;
...

bool analyze () {
  int lbd;
  ...
  slow += (lbd - slow)/(double)(1<<14);
  fast += (lbd - fast)/(double)(1<<5);
  ...
}

bool restarting () {
  return conflicts > limit && fast > 1.25 * slow;
}
```

fast 64-bit fixed point implementation avoiding floating point

inspired by Donald Knuth's implementation of our agility metric

```
long fast, slow;                  // assume (sizeof (long) == 8);
...                               // initialization code skipped ...

bool analyze () {
  int lbd;                        // assume (sizeof (int) == 4);
  ...
  fast -= fast >> 5;
  fast += lbd << (32 - 5);
  slow -= slow >> 14;
  slow += lbd << (32 - 14);
  ...
}

bool restarting () {
  return conflicts > limit && fast / 125 > slow / 100;
}
```

Phase Saving and Rapid Restarts

- phase assignment:
  - assign decision variable to 0 or 1?
  - "Only thing that matters in *satisfiable* instances" Hans van Maaren
- "phase saving" as in RSat [PipatsrisawatDarwiche'07]
  - 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 only theoretically avoids local minima
  - works nicely together with phase saving
- reusing the trail can reduce the cost of restarts [RamosVanDerTakHeule-JSAT'11]
- target phases of largest conflict free trail / assignment [Biere-SAT-Race-2019]

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 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**)

Normalized VSIDS: NVSIDS

[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

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

  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
- originally the leared clause literals were "bumped"
  - better to bump all resolved variables (on conflict level) too
  - reason literals too of learned clause literals [HuiLiangGanesh...-SAT'16]

- 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

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

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

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

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

VSIDS vs. Exponential Moving Averages (EMA)

- exponential VSIDS factor f as smoothing parameter α

$$\alpha \quad = \quad 1 - f$$

- reinterpretation

$$s_{i+1} \quad = \quad s_i + (\delta_{i+1} - s_i) \cdot \alpha \quad = \quad (1-\alpha) \cdot s_i + \alpha \cdot \delta_{i+1} \quad = \quad f \cdot s_i + (1-f) \cdot \delta_{i+1}$$

- “old” prediction s_i
- “error” $\delta_{i+1} - s_i$ (predicted s_i but got δ_{i+1})
- “update” EMA into the direction of the error scaled by α

BERKMIN's Dynamic Second Order Heuristics

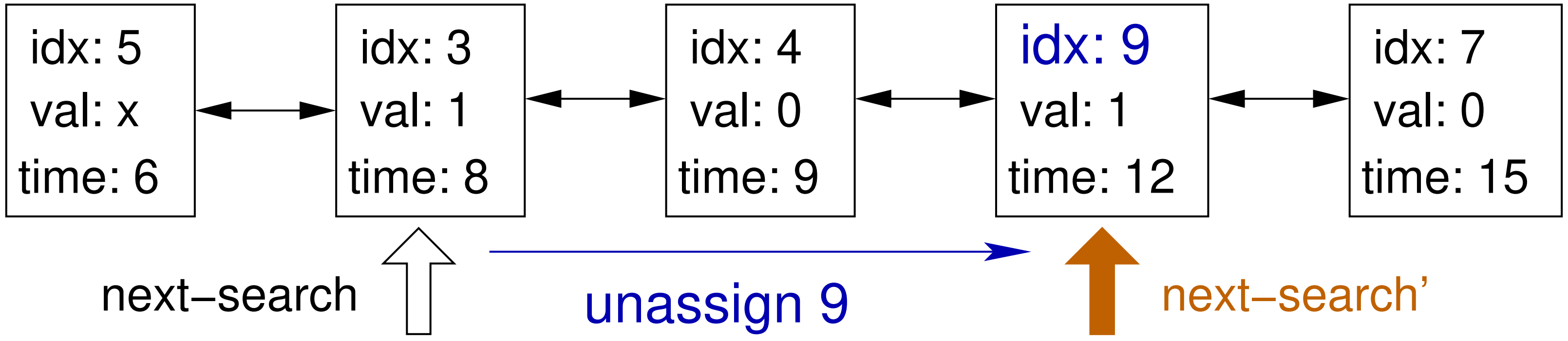
[GoldbergNovikov-DATE'02]

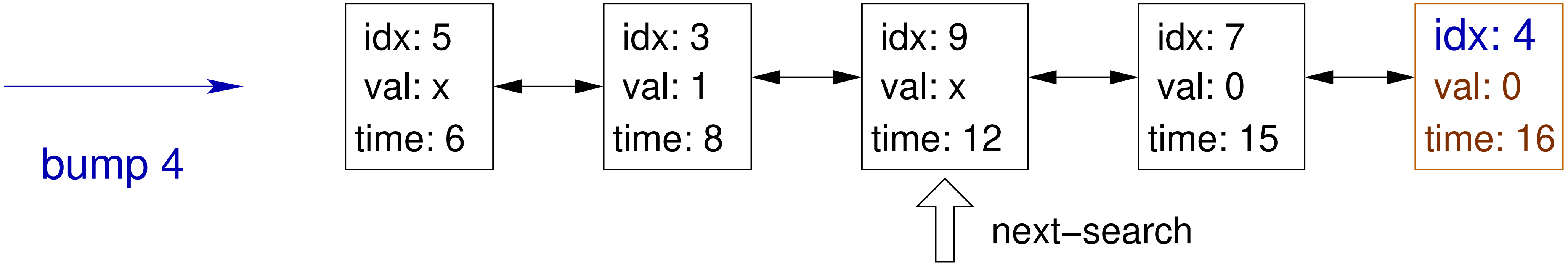
- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to chose the decision variable for one clause
  - if all learned clauses 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 [BiereFröhlich-SAT'15]
    - need pointer to last picked variable in queue "next-search"
    - reset during back-tracking
  - rather aggressive (used in "unstable" search phases in CaDiCaL)
- *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)

Fast Implementation of VMTF

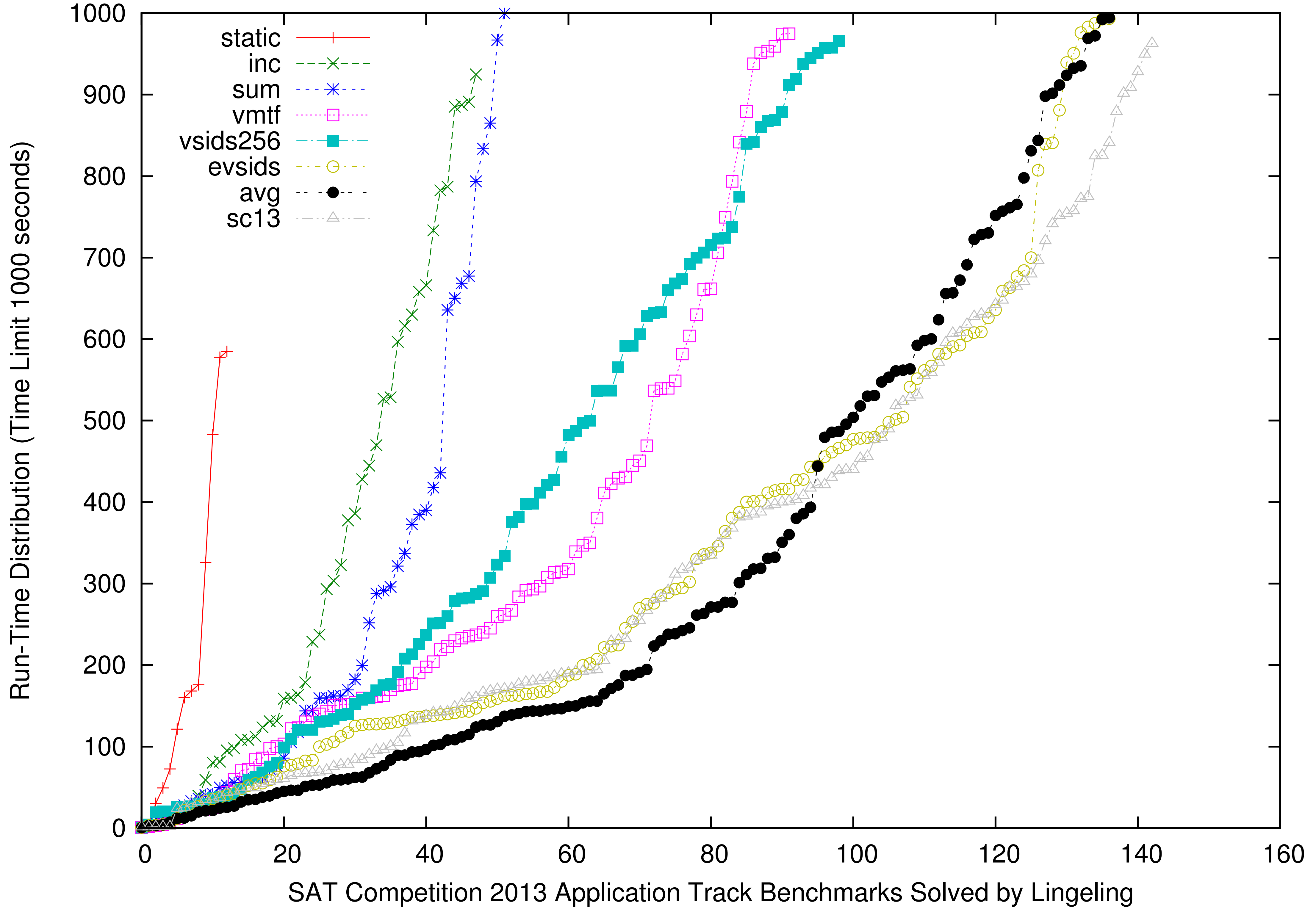
[BiereFröhlich-SAT’15]





keep queue sorted by “time stamps” (enqueue time / enqueued counter)

- 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** $s_{i+1} = i$ actually important to bump in a “stable” way
  - **sum** $s_{i+1} = s_i + i \cdot \beta_i$ emphasis on recent conflicts
  - **vsids** $s_{i+1} = d \cdot s_i + \beta_i$ decay $d \in [0, 1)$ e.g. $d = 0.95$
  - **evsids** $s_{i+1} = s_i + g_i \cdot \beta_i$, $g_{i+1} = e \cdot g_i$ factor $e = \frac{1}{f} \in [1, 2)$ e.g. $e = 1.05$
  - **avg** $s_{i+1} = s_i + \beta_i \cdot (i - s_i)/2$ another filter function
- 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

Reduction Strategies

- 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:

$$\frac{\begin{array}{ccc} \{x = 0\} \cup E & \qquad & \{x = 1\} \cup E \\ \Downarrow & & \Downarrow \\ E_0 \cup E & & E_1 \cup E \end{array}}{(E_0 \cap E_1) \cup E}$$

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.

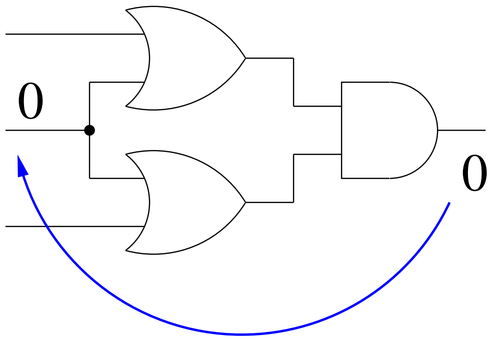
Stålmarck's Method Recursively

$$
\begin{array}{cccc}
\multicolumn{2}{c}{x = 0} & \multicolumn{2}{c}{x = 1} \\
\multicolumn{2}{c}{\Downarrow} & \multicolumn{2}{c}{\Downarrow} \\
y = 0 & y = 1 & y = 0 & y = 1 \\
\Downarrow & \Downarrow & \Downarrow & \Downarrow \\
E_{00} & E_{01} & E_{10} & E_{11} \\
\cline{1-2} \cline{3-4}
\multicolumn{2}{c}{E_0} & \multicolumn{2}{c}{E_1} \\
\hline
\multicolumn{4}{c}{E}
\end{array}
$$

(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 except 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

[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
- remove subsumed clauses, strengthen through self-subsuming resolution
  SATeLite [EénBiere SAT'05] and Quantor [Biere SAT'04]

[EénBiere-SAT'05]

Replace $(\boxed{\bar{x}} \vee a)_1$ $(\boxed{x} \vee \bar{a} \vee \bar{b})_4$ $(\boxed{\bar{x}} \vee b)_2$ $(\boxed{x} \vee d)_5$ $(\boxed{\bar{x}} \vee c)_3$ by $\cancel{(a \vee \bar{a} \vee \bar{b})}_{14}$ $(a \vee d)_{15}$ $\cancel{(b \vee \bar{a} \vee \bar{b})}_{24}$ $(b \vee d)_{25}$ $(c \vee \bar{a} \vee \bar{b})_{34}$ $(c \vee d)_{45}$

- number of clauses not increasing
- strengthen and remove subsumed clauses too
- most important and most effective preproessing we have

Bounded Variable Addition

[MantheyHeuleBiere-HVC'12]

Replace $(a \vee d)$ $(a \vee e)$ $(b \vee d)$ $(b \vee e)$ $(c \vee d)$ $(c \vee e)$ by $(\bar{x} \vee a)$ $(\bar{x} \vee b)$ $(\bar{x} \vee c)$ $(x \vee d)$ $(x \vee e)$

- number of clauses has to decrease strictly
- reencodes for instance naive at-most-one constraint encodings

Fast (Self) Subsumption

- (smaller) clause (self-)subsumes (longer) clause backward (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]
    - can use Bloom Filters (as in "bit-state hashing"), aka *signatures*
- (longer) clause (self-)subsumed by (shorter) clause forward (self-)subsumption
  - can be made more efficient by one-watcher scheme [Zhang-SAT'05]
  - inspired also pattern mining algorithms [BayardoPanda-SDM'11]
  - sort clauses for faster checking (instead of marking literals)
- probably most efficient version is in CaDiCaL [Biere'17]
  - monitor variables in added and removed clauses
  - added variables trigger <u>forward</u> (self-)-subsumption attempts
  - removed variabels trigger variable elimination attempts

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.

Vivification / Distillation

[PietteHamadiSais-ECAI'08] [HanSomenzi-DAC'07] [LuoLiXiaoManyàLü-IJCAI'17]

- generalization of failed literal probing (see earlier discussion on decision heuristics)
  - for all clauses $C = \{\ell_1, \cdots \ell_n\}$
    - for $i = 1 \ldots n-1$
      - assume and propagate $\bar{\ell}_1, \ldots, \bar{\ell}_i$ (ignoring C during propagation)
      - on conflict or if another ℓ_j is assigned $\top$ then C is implied and removed
      - if one of the latter ℓ_j with $j > i$ is assigned $\perp$ remove it from C
- can be applied to learned (redundant) clauses too [LuoLiXiaoManyàLü-IJCAI'17]
  - originaly only used for preprocessing original (irredundant) clauses
  - focus on important (low glue clauses) redundant clauses (once maybe)
- original distillation work used "tries" [PietteHamadiSais-ECAI'08]
  - factor out common prefix of clauses to save propagations
  - can be simulated by sorting literals in clauses and delay backtracking [Biere'17]
  - quite useful on many instances even if it uses 30% of running time

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-DAM'99]

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

fix a CNF F

$(a \vee b \vee l)$

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

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

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 monotone input reduction

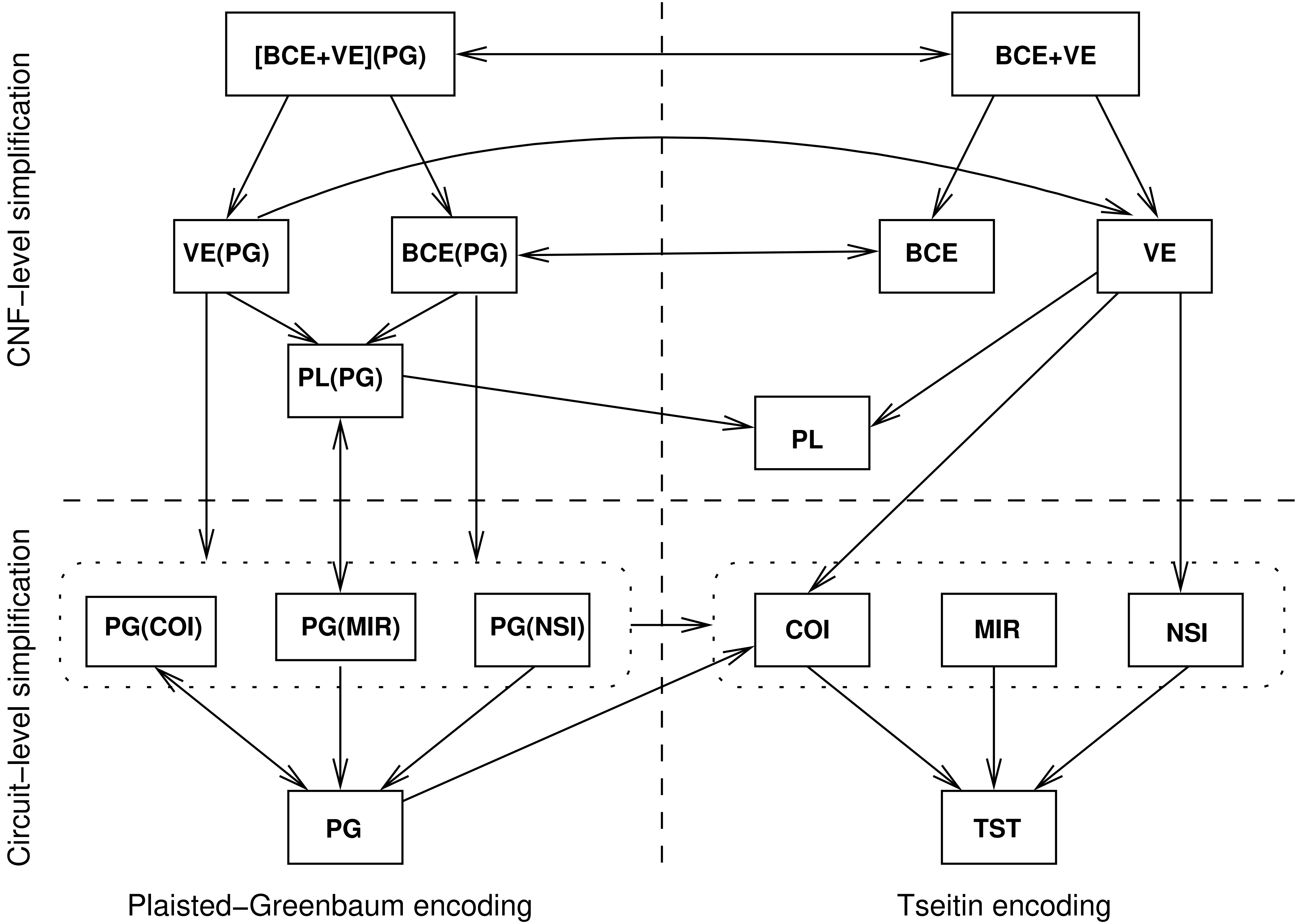
NSI non-shared inputs elimination

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

VE variable elimination as in DP / Quantor / SATeLite

BCE blocked clause elimination

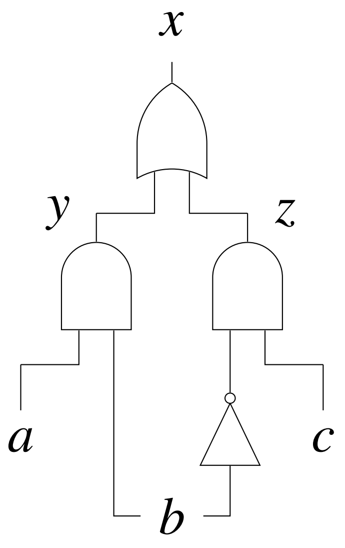

[BCE+VE](PG)
BCE+VE
CNF–level simplification
VE(PG)
BCE(PG)
BCE
VE
PL(PG)
PL
Circuit–level simplification
PG(COI)
PG(MIR)
PG(NSI)
COI
MIR
NSI
PG
TST
Plaisted–Greenbaum encoding
Tseitin encoding

- resolution proofs (RES) are simple to check but large and hard(er) to produce directly
- original idea for clausal proofs and checking them:
  - proof traces are sequences of “learned clauses” C
  - first check clause through unit propagation $\boxed{F \vdash_1 C}$ then add C to F
  - reverse unit implied clauses (RUP) [GoldbergNovikov’03] [VanGelder’12]
- deletion information:
  - “deletion” proof lines tell checker to forget clause and decreases checking time substantially
  - trace of added and deleted clauses (**D**RUP) [HeuleHuntWetzler-FMCAD’13 / STVR’14]
- RUP/RES tracks SAT Competion 2007, 2009, 2011, now DRUP/DRAT mandatory since 2013 to certify UNSAT
- big certified proofs:
  - Pythagorean Triples [HeuleKullmannMarek-SAT’16] (200TB), Schur Number Five [Heule-AAAI’18] (2PB)
  - Certification: Coq [CruzFilipeMarquesSilvaSchneiderKamp-TACAS’17 / JAR’19], ACL2, Isabelle

CNF	trace	extended trace	resolution trace	RUP	DRUP
p cnf 3 8					
-1 -2 -3 0	1 -2 -3 -1 0 0	1 -2 -3 -1 0 0	1 -1 -3 -2 0 0		
-1 -2 3 0	2 -2 3 -1 0 0	2 -2 3 -1 0 0	2 -1 3 -2 0 0		
-1 2 -3 0	3 2 -3 -1 0 0	3 2 -3 -1 0 0	3 2 -1 -3 0 0		
-1 2 3 0	4 2 3 -1 0 0	4 2 3 -1 0 0	4 2 -1 3 0 0		
1 -2 -3 0	5 1 -3 -2 0 0	5 1 -3 -2 0 0	5 -2 -3 1 0 0		
1 -2 3 0	6 1 3 -2 0 0	6 1 3 -2 0 0	6 -2 3 1 0 0		
1 2 -3 0	7 1 -3 2 0 0	7 1 -3 2 0 0	7 1 -3 2 0 0		
1 2 3 0	8 1 3 2 0 0	8 1 3 2 0 0	8 1 3 2 0 0		
	9 ⋆ 7 8 0	9 1 2 0 7 8 0	9 1 2 0 7 8 0	-2 -3 0	-2 -3 0
	10 ⋆ 9 5 6 0	10 1 0 9 5 6 0	10 -2 1 0 5 6 0	-3 0	d 1 -2 -3 0
	11 ⋆ 1 10 2 0	11 -2 0 1 10 2 0	11 1 0 10 9 0	2 0	d -1 -2 -3 0
	12 ⋆ 10 11 4 0	12 3 0 10 11 4 0	12 -1 -2 0 1 2 0	-1 0	-2 3 0
	13 ⋆ 10 11 3 12 0	13 0 10 11 3 12 0	13 -2 0 12 11 0	0	d 1 -2 3 0
			14 2 3 0 11 4 0		d -1 -2 3 0
			15 3 0 14 13 0		2 -3 0
			16 2 -3 0 11 3 0		d 1 2 -3 0
			17 -3 0 16 13 0		d -1 2 -3 0
			18 0 17 15 0		2 3 0
					d 1 2 3 0
					d -1 2 3 0
					-2 0
					0
	picosat -t	picosat -T	tracecheck -B	cadical	cadical -P1

Definition. Clause C *blocked* on literal $\boxed{\ell} \in C$ w.r.t CNF F if
for all resolution candidates $D \in F$ with $\bar{\ell} \in D$ the resolvent $(C \backslash \ell) \vee (D \backslash \bar{\ell})$ is tautological.

Assume output true, thus single unit clause constraint (x)



$$\begin{array}{ccccc}
(x) & & (x) & & (x) \\
(\boxed{x} \vee \bar{y})_1 \; (\boxed{x} \vee \bar{z})_2 \; (\bar{x} \vee y \vee z) & & (\bar{x} \vee y \vee z) & & (\bar{x} \vee y \vee z) \\
(\bar{y} \vee a) \; (\bar{y} \vee b) \; (\boxed{y} \vee \bar{a} \vee \bar{b})_3 & \Rightarrow & (\bar{y} \vee \boxed{a})_5 \; (\bar{y} \vee b) & \Rightarrow & (\bar{y} \vee b) \\
(\bar{z} \vee \bar{b}) \; (\bar{z} \vee c) \; (\boxed{z} \vee b \vee \bar{c})_4 & & (\bar{z} \vee \bar{b}) \; (\bar{z} \vee \boxed{c})_6 & & (\bar{z} \vee \bar{b})
\end{array}$$

Plaisted-Greenbaum encoding *drops* **upward** propagating clauses of only **positively** occurring gates.
Plaisted-Greenbaum encoding *drops* **downward** propagating clauses of only **negatively** occurring gates.

Unconstrained or monotone inputs can be removed too.

Resolution Asymmetric Tautologies (RAT)

"Inprocessing Rules" [JärvisaloHeuleBiere-IJCAR'12]

- justify complex preprocessing algorithms in Lingeling [Biere-TR'10]
  - examples are adding blocked clauses or variable elimination
  - interleaved with research (forgetting learned clauses = reduce)
- need more general notion of redundancy criteria
  - extension of blocked clauses
  - replace "resolvents on $\boxed{l}$ are tautological" by "resolvents on $\boxed{l}$ are RUP"

example: $(a \vee \boxed{l})$ RAT on l w.r.t. $(a \vee b) \wedge (l \vee c) \wedge \underbrace{(\bar{l} \vee b)}_{D}$

  - deletion information is again essential (DRAT) [HeuleHuntWetzler-FMCAD'13 / STVR'14]
  - now mandatory in the main track of the SAT competitions since 2013
  - pretty powerful: can for instance also cover symmetry breaking

[KieslSeidlTompitsBiere-IJCAR'16] [KieslSeidlTompitsBiere-LMCS'18]

C is *set blocked* on $L \subseteq C$ iff $(C \backslash L) \cup \bar{L} \cup D$ is a tautology for all $D \in F$ with a literal in $\bar{L}$

- easy to check if the "witness" L is given
  - NP hard to check otherwise ("exponential" in $|L|$)
- local redundancy property
  - only considering the resolution environment of a clause
  - in constrast to (R)AT / RUP
- strictly more powerful than blocked clauses ($|L|$ = 1)
- most general local redundancy property *super blocked clauses*
  - strictly more powerful than blocked clauses
  - Π_2^P complete to check

Example:

$C = \boxed{\text{a}} \vee \boxed{\text{b}}$ set blocked

in $F = (\bar{a} \vee b) \wedge (a \vee \bar{b})$

by $L = \{a, b\}$

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

Definition. A partial assignment α *blocks* a clause C if α assigns the literals in C to false (and no other literal).

Definition. A clause C is *redundant* w.r.t. a formula F if F and $F \cup \{C\}$ are satisfiability equivalent.

Definition. A formula F simplified by a partial assignment α is written as $F|_\alpha$.

Theorem.

Let F be a formula, C a clause, and α the assignment blocked by C.

Then C is redundant w.r.t. F iff exists an assignment ω such that

(i) ω satisfies C and (ii) $F|_\alpha \models F|_\omega$.

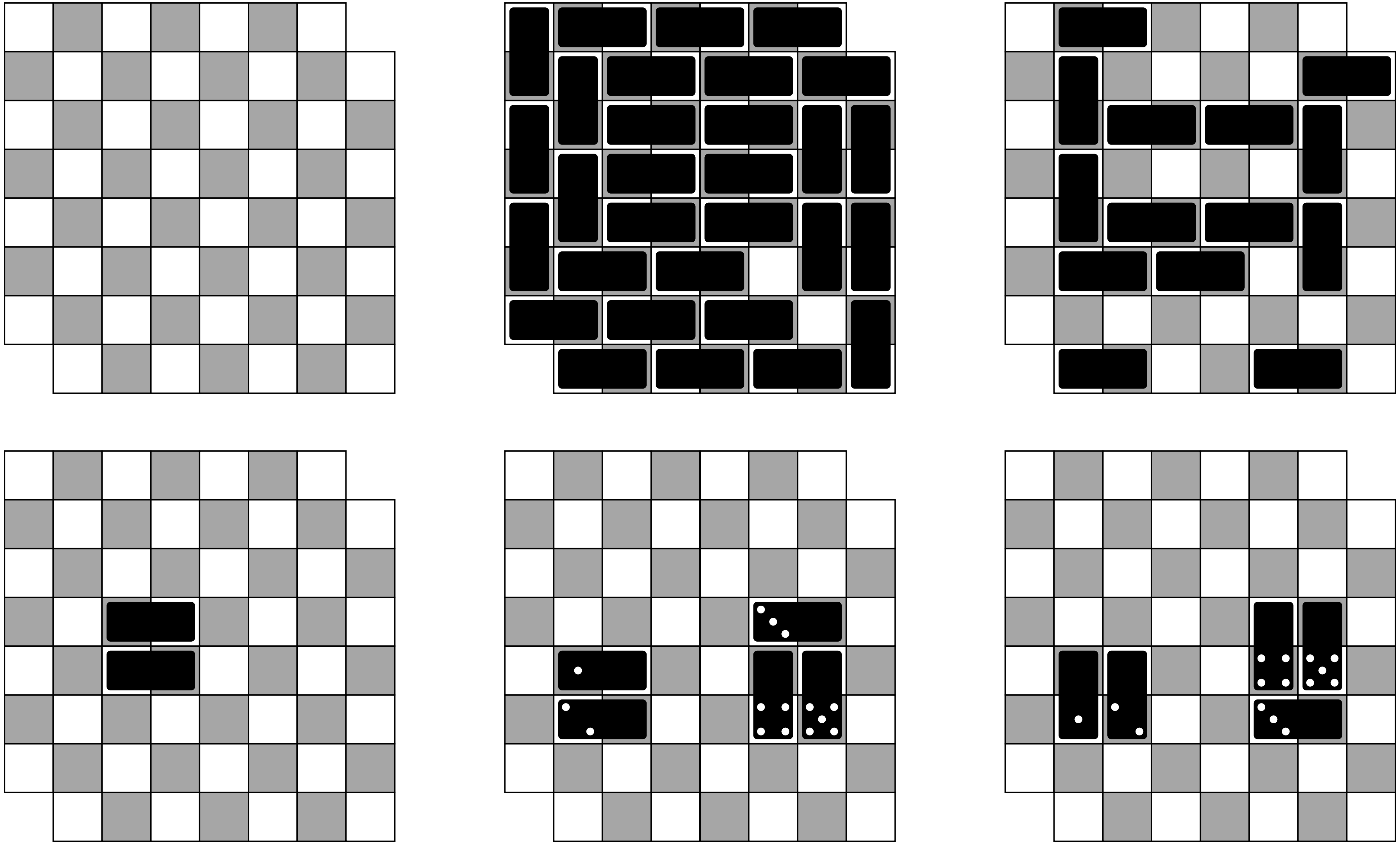
[HeuleKieslBiere-CADE'17] [HeuleKieslBiere-JAR'19]

- more general than RAT: short proofs for pigeon hole formulas *without new variables*

C propagation redundant (PR) if exists assignment ω satisfying C with $F|\alpha \vdash_1 F|\omega$

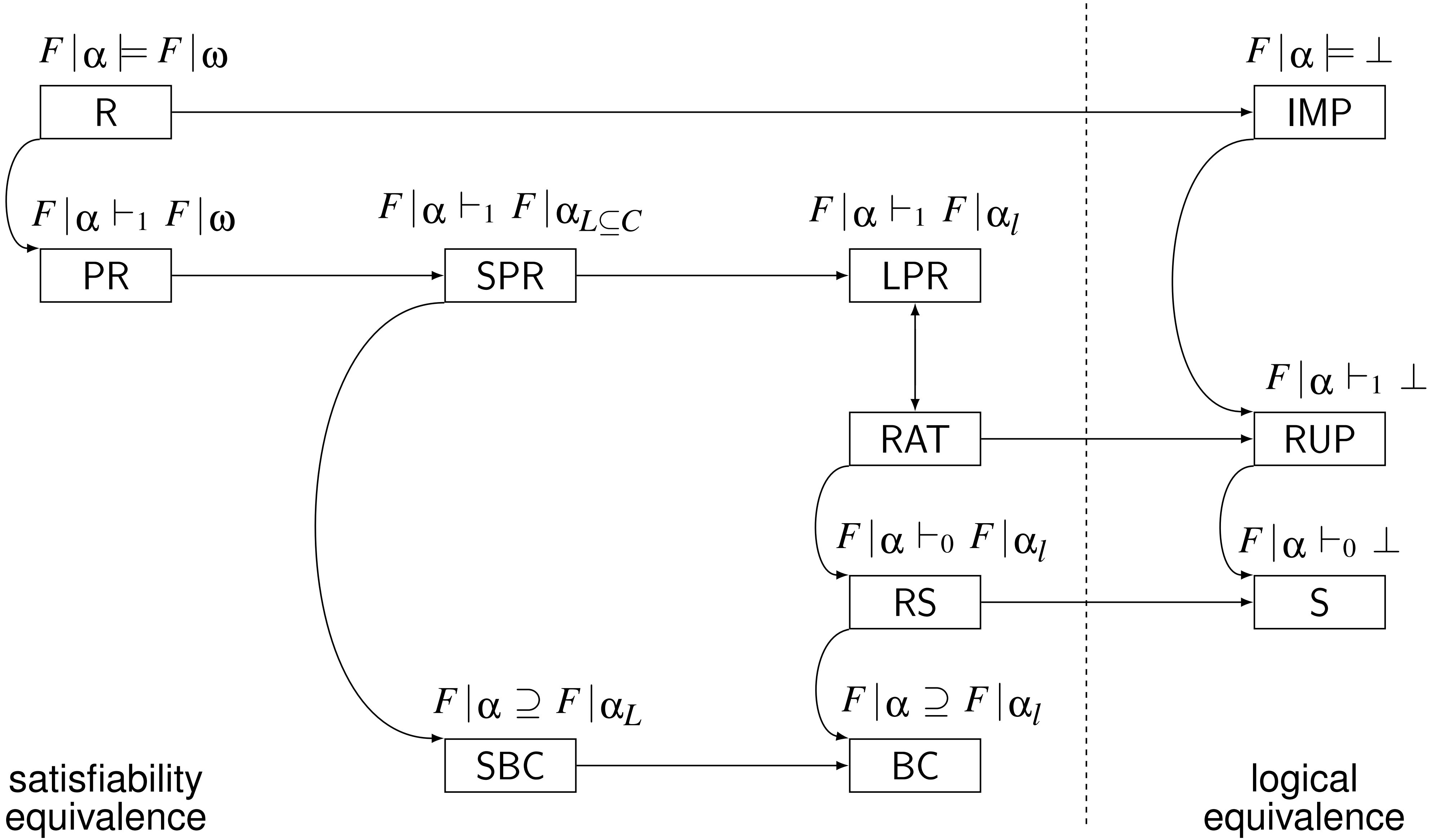
so in essence replacing "$\models$" by "$\vdash_1$" (implied by unit propagation) where again α is the clause that blocks C

- Satisfaction Driven Clause Learning (SDCL) [HeuleKieslSeidlBiere-HVC'17] best paper
  - first automatically generated PR proofs
  - prune assignments for which we have other at least as satisfiable assignments
  - (filtered) positive reduct in SaDiCaL [HeuleKieslBiere-TACAS'19] nominated best paper
- translate PR to DRAT [HeuleBiere-TACAS'18]
  - only one additional variable needed
  - shortest proofs for pigeon hole formulas
- translate DRAT to extended resolution [KieslRebolaPardoHeule-IJCAR'18] best paper
- recent seperation results in [BussThapen-SAT'19]



CDCL

SDCL


$F|\alpha \models F|\omega$
R
$F|\alpha \models \bot$
IMP
$F|\alpha \vdash_1 F|\omega$
PR
$F|\alpha \vdash_1 F|\alpha_{L\subseteq C}$
SPR
$F|\alpha \vdash_1 F|\alpha_l$
LPR
$F|\alpha \vdash_1 \bot$
RAT
RUP
$F|\alpha \vdash_0 F|\alpha_l$
$F|\alpha \vdash_0 \bot$
RS
S
$F|\alpha \supseteq F|\alpha_L$
$F|\alpha \supseteq F|\alpha_l$
SBC
BC
satisfiability equivalence
logical equivalence

CDCL(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 all variables are assigned **then return** SAT
    $l :=$ *Decide*$()$
    $\alpha := \alpha \cup \{l\}$

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*$()$
        $\alpha := \alpha \cup \{l\}$

In the *positive reduct* consider all clauses satisfied by α but remove unassigned literals and add C:

Definition. Let F be a formula and α an assignment. Then, the *positive reduct* of F and α is the formula $G \wedge C$ where C is the clause that blocks α and $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } D|_\alpha = \top\}$.

Theorem. Let F be a formula, α an assignment, and C the clause that blocks α.
Then, C is $\boxed{\text{SBC}}$ by an $L \subseteq C$ with respect to F if and only if the assignment α_L satisfies the positive reduct.

We obtain the *filtered* positive reduct by not taking *all* satisfied clauses of F but only those for which the untouched part is not implied by $F|_\alpha$ via unit propagation:

Definition. Let F be a formula and α an assignment. Then, the *filtered positive reduct* of F and α is the formula $G \wedge C$ where C is the clause that blocks α and $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \not\vdash_1 \text{untouched}_\alpha(D)\}$.

Theorem. Let F be a formula, α an assignment, and C the clause that blocks α.
Then, C is $\boxed{\text{SPR}}$ by an $L \subseteq C$ with respect to F if and only if the assignment α_L satisfies the filtered positive reduct.

where SPR extends SBC in the same way by propagation as RAT extends BC

formula	MapleChrono	[HVC'17]	plain CDCL	positive	filtered	ACL2
Urquhart-s3-b1	2.95	5.86	16.31	> 3600	**0.02**	0.09
Urquhart-s3-b2	1.36	2.4	2.82	> 3600	**0.03**	0.13
Urquhart-s3-b3	2.28	19.94	2.08	> 3600	**0.03**	0.16
Urquhart-s3-b4	10.74	32.42	7.65	> 3600	**0.03**	0.17
Urquhart-s4-b1	86.11	583.96	> 3600	> 3600	**0.32**	2.37
Urquhart-s4-b2	154.35	1824.95	183.77	> 3600	**0.11**	0.78
Urquhart-s4-b3	258.46	> 3600	129.27	> 3600	**0.16**	1.12
Urquhart-s4-b4	> 3600	> 3600	> 3600	> 3600	**0.14**	1.17
Urquhart-s5-b1	> 3600	> 3600	> 3600	> 3600	**1.27**	9.86
Urquhart-s5-b2	> 3600	> 3600	> 3600	> 3600	**0.58**	4.38
Urquhart-s5-b3	> 3600	> 3600	> 3600	> 3600	**1.67**	17.99
Urquhart-s5-b4	> 3600	> 3600	> 3600	> 3600	**2.91**	24.24
hole20	> 3600	1.13	> 3600	**0.22**	0.55	6.78
hole30	> 3600	8.81	> 3600	**1.71**	4.30	87.58
hole40	> 3600	43.10	> 3600	**7.94**	20.38	611.24
hole50	> 3600	149.67	> 3600	**25.60**	68.46	2792.39
mchess_15	51.53	1473.11	2480.67	> 3600	**13.14**	29.12
mchess_16	380.45	> 3600	2115.75	> 3600	**15.52**	36.86
mchess_17	2418.35	> 3600	> 3600	> 3600	**25.54**	57.83
mchess_18	> 3600	> 3600	> 3600	> 3600	**43.88**	100.71

CNF

$$F'(I,S,T,x,y,z) = H'(J,x,y,z,T) \wedge G'(K,S,x) \quad \text{with}$$

$$H'(J,x,y,z,T) = \underbrace{(\bar{x} \vee y) \wedge (\bar{x} \vee z) \wedge (x \vee \bar{y} \vee \bar{z})}_{\text{Tseitin encoding of top AND gate in } H} \wedge H''(J,y,z,T)$$

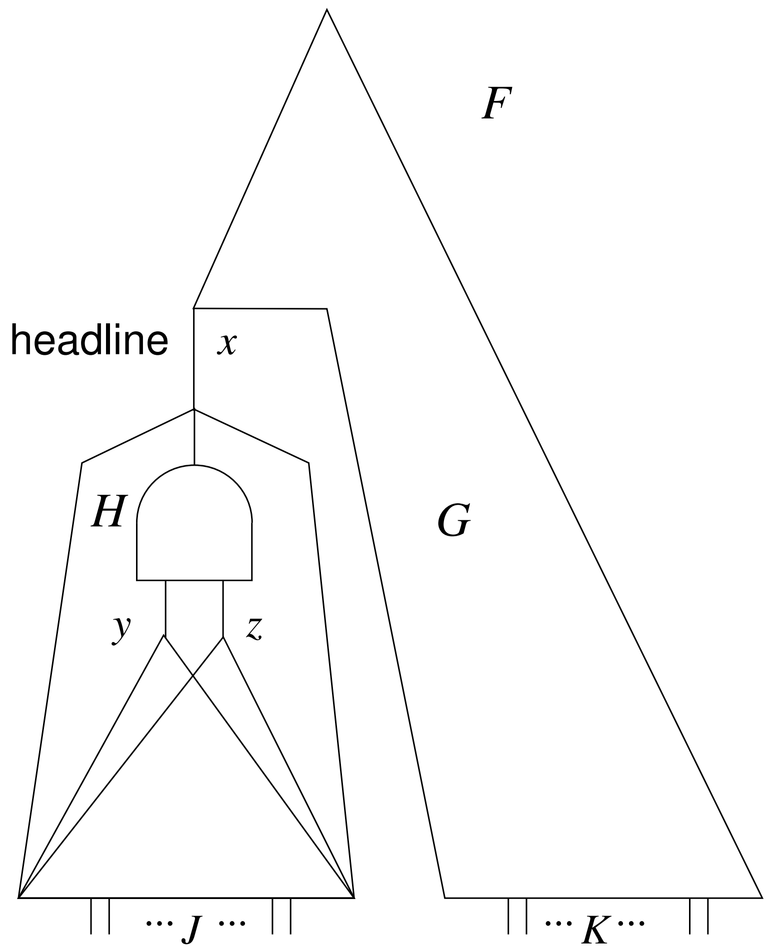
Formula

$$F(I) = G(H(J),K) = \exists x.\,(x = H(J)) \wedge G(x,K)$$

assume $\sigma_0(H(J)) = \sigma_0(x) = 0$

assume $\sigma_1(H(J)) = \sigma_1(x) = 1$

Drop H?



[MonienSpeckenmeyer-DAM'85] [HeuleKiesISeidlBiere-HVC'17]

Definition. Assignment α is an *autarky* for F if α satisfies all $C \in F$ with $var(\alpha) \cap var(C) \neq \emptyset$.

In other words, an autarky satisfies every clause it touches.

Example. Let $F = (a \vee b \vee \overline{c}) \wedge (\overline{b} \vee c \vee \overline{d}) \wedge (\overline{a} \vee d)$ and $\alpha = b\,c$.
Then, α touches only the first two clauses. Since it satisfies them, it is an *autarky* for F.

Definition. Assignment $\alpha = \gamma \cup \beta$ is a *conditional autarky* for F with conditional part γ and autarky part β if β satisfies all $C \in F|_\gamma$ with $var(\alpha) \cap var(C) \neq \emptyset$.

Thus a conditional autarky satisfies every clause its autarky part touches after applying the conditional part.

Example. Let $F = (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{d}) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge (\overline{a} \vee d)$ and $\alpha = \gamma \cup \beta = a\,b\,c$, $\gamma = a$, $\beta = b\,c$.
Then, β touches the first three clauses, α satisfies them, thus α is a *conditional autarky* for F with conditional part γ and autarky part β.

Definition. A clause C is *globally blocked* by a set $\boxed{L}$ of literals in a formula F if $L \cap C \neq \emptyset$ and for all $D \in F$ with a literal in $\overline{L}$ but no literal from L, the clause $(D \setminus \overline{L}) \vee C$ is a tautology.

Example. Let $F = (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{d}) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge (\overline{a} \vee d)$ then the clauses $(a \rightarrow b)$ and $(a \rightarrow c)$ are both globally blocked for $L = \{b, c\}$.

Theorem. Let F be a formula, let C be a clause, let L be a set of literals such that $L \cap C \neq \emptyset$, Define the assignments $\gamma = \overline{C \setminus L}$ and $\beta = L$. Then, C is globally blocked by L in F iff $\gamma \cup \beta$ is a conditional autarky.

Thus globally blocked clauses can be found by "computing" conditional autarkies!

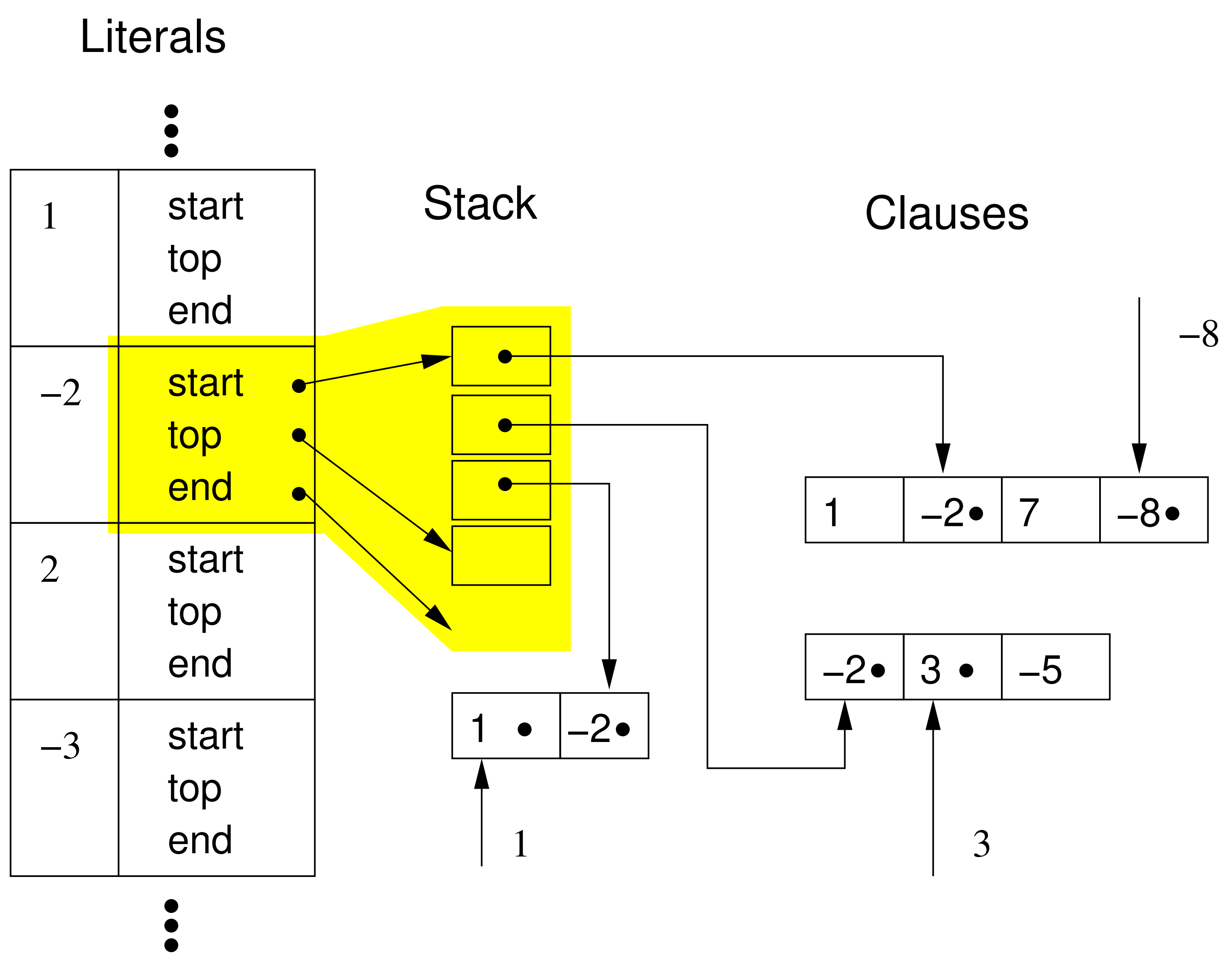
Inprocessing: Interleaving Preprocessing and Search

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

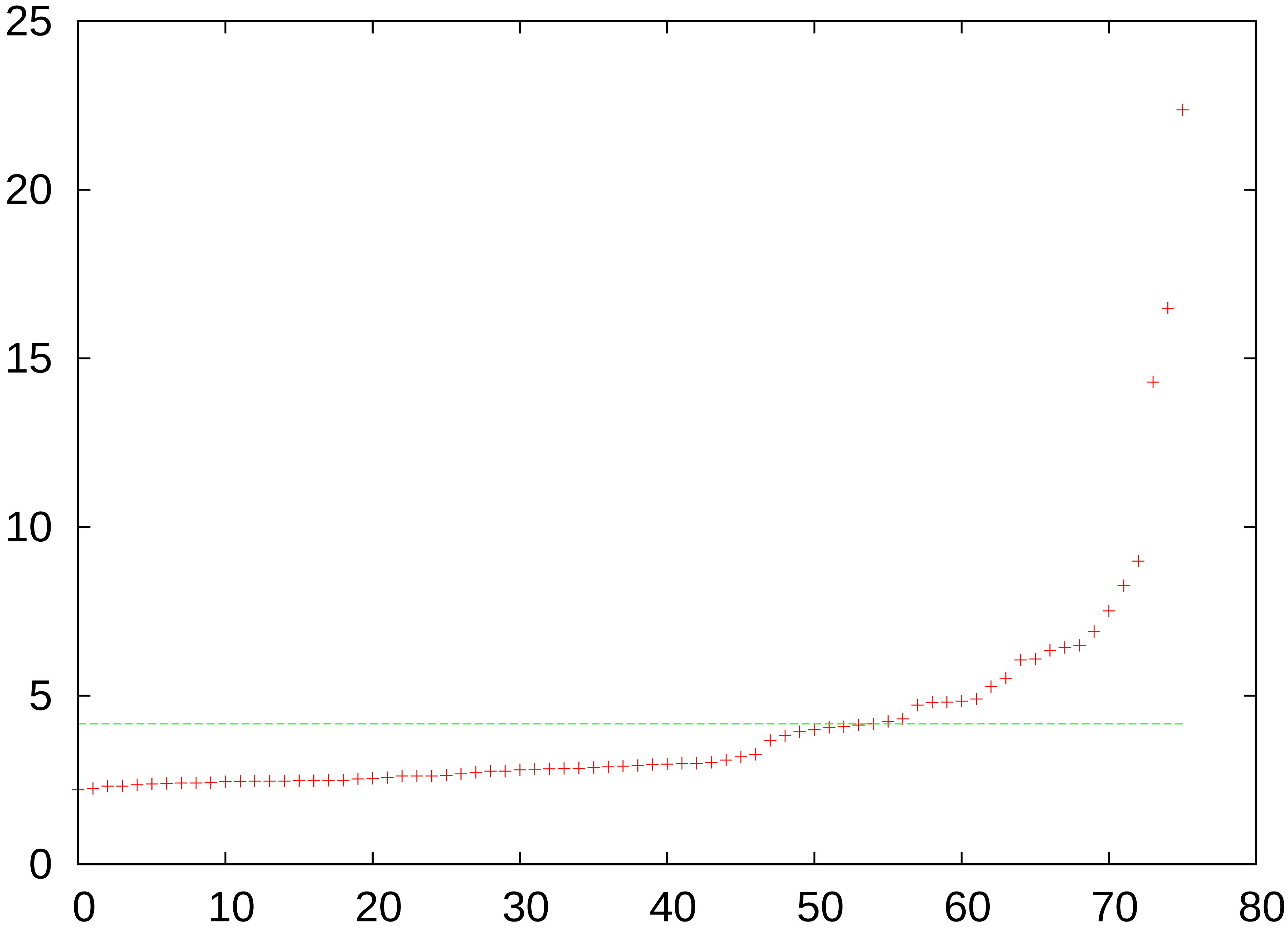
- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple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

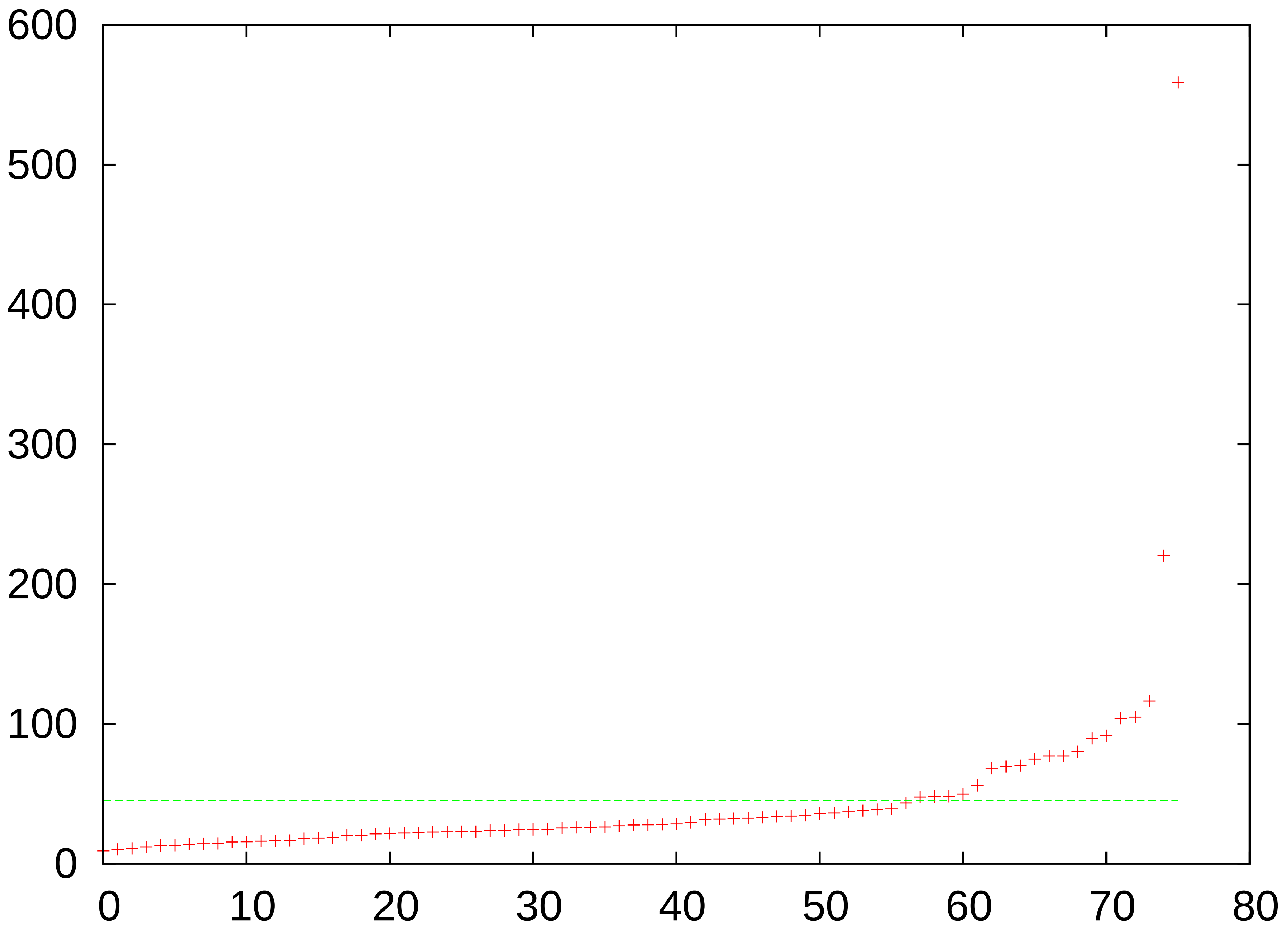
Benefits of Inprocessing

- 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]
- incremental solving [FazekasBiereScholl-SAT’19]

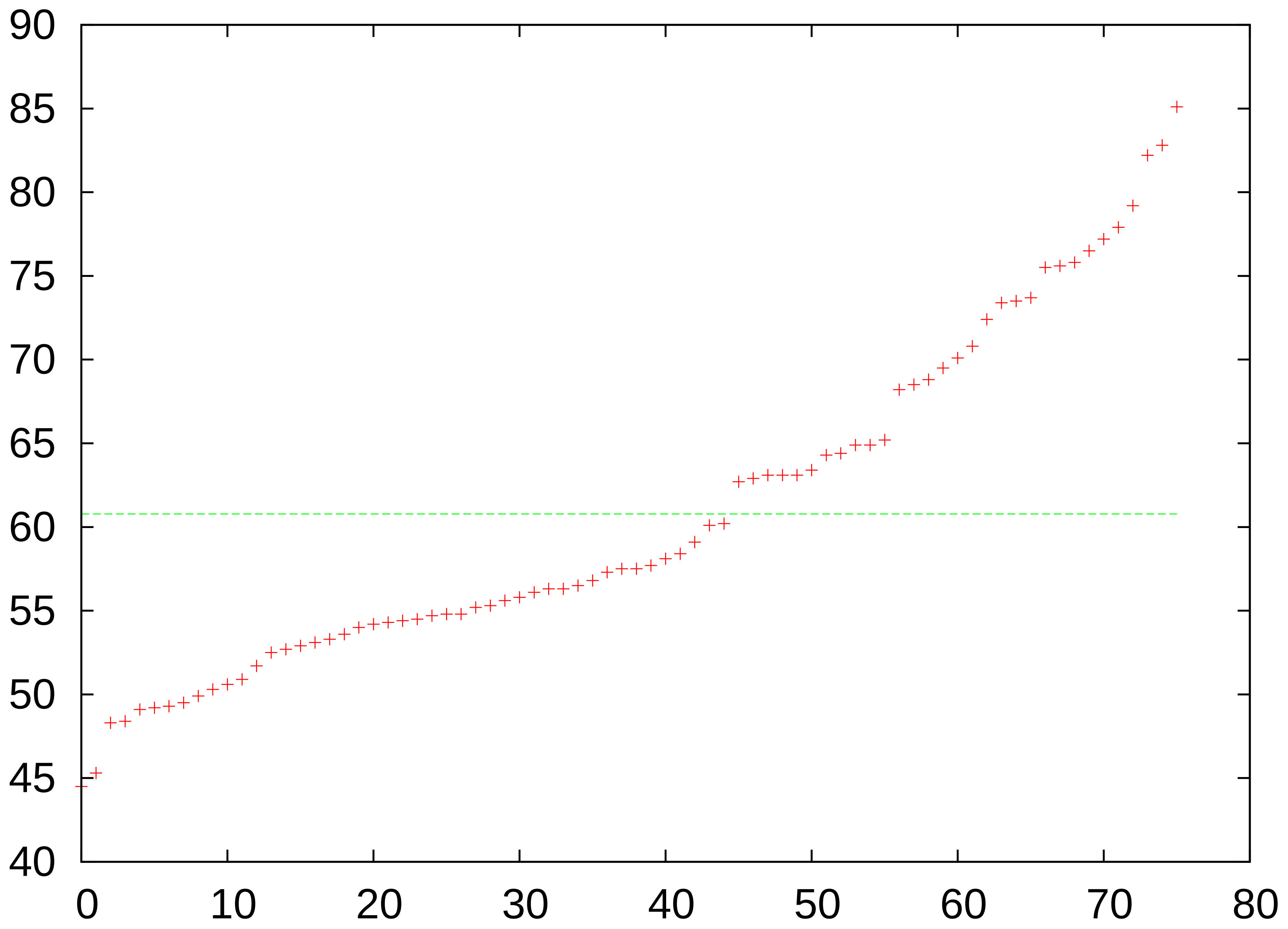

Literals
1
start
top
end
–2
start
top
end
2
start
top
end
–3
start
top
end
Stack
Clauses
–8
1
–2
7
–8
–2
3
–5
1
–2
1
3

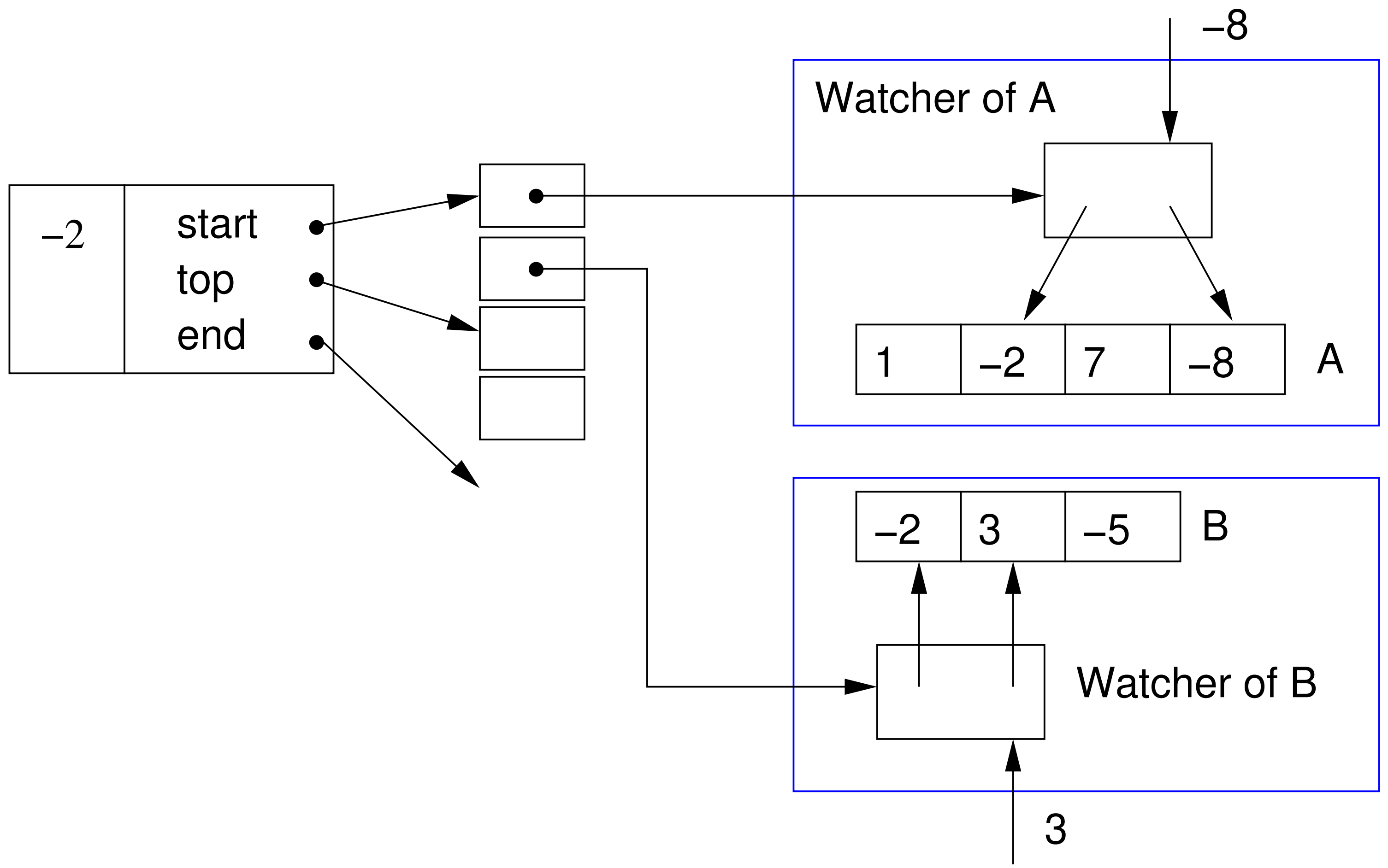
Average Number Clauses Visited Per Propagation

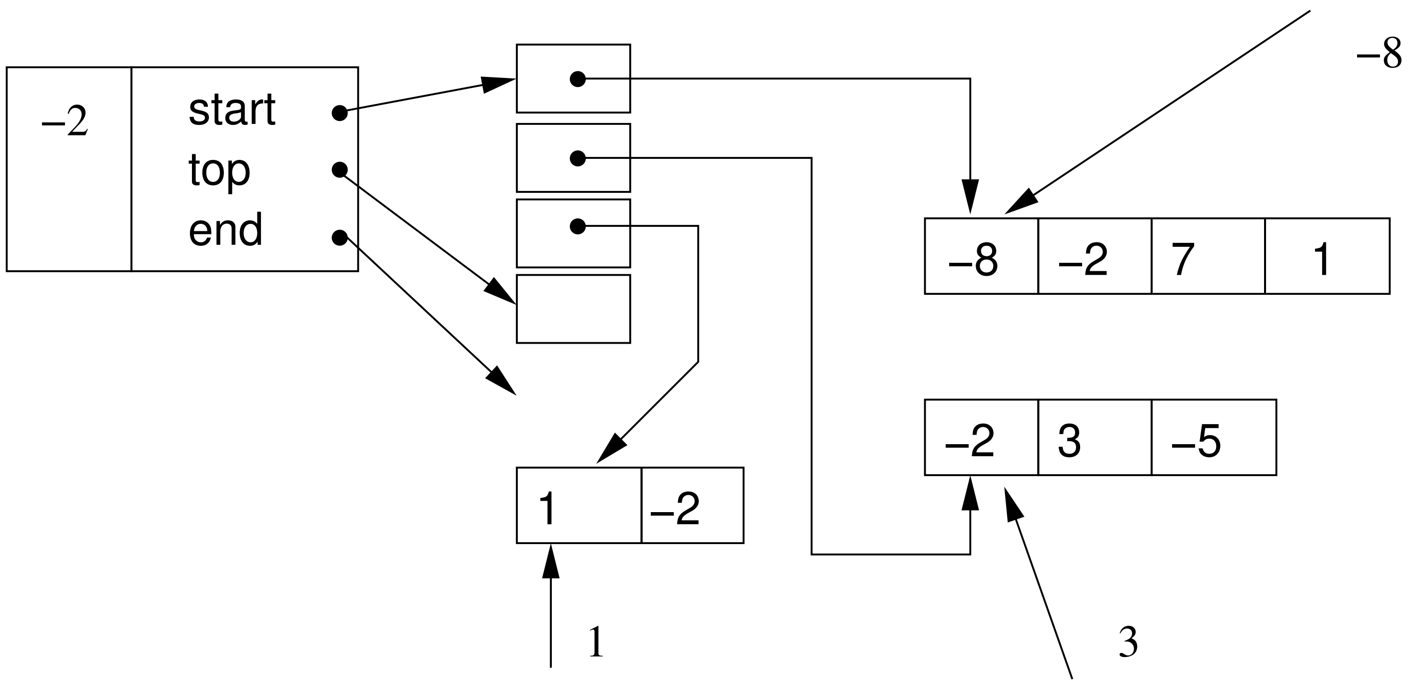



600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80

Percentage Visited Clauses With Other Watched Literal True

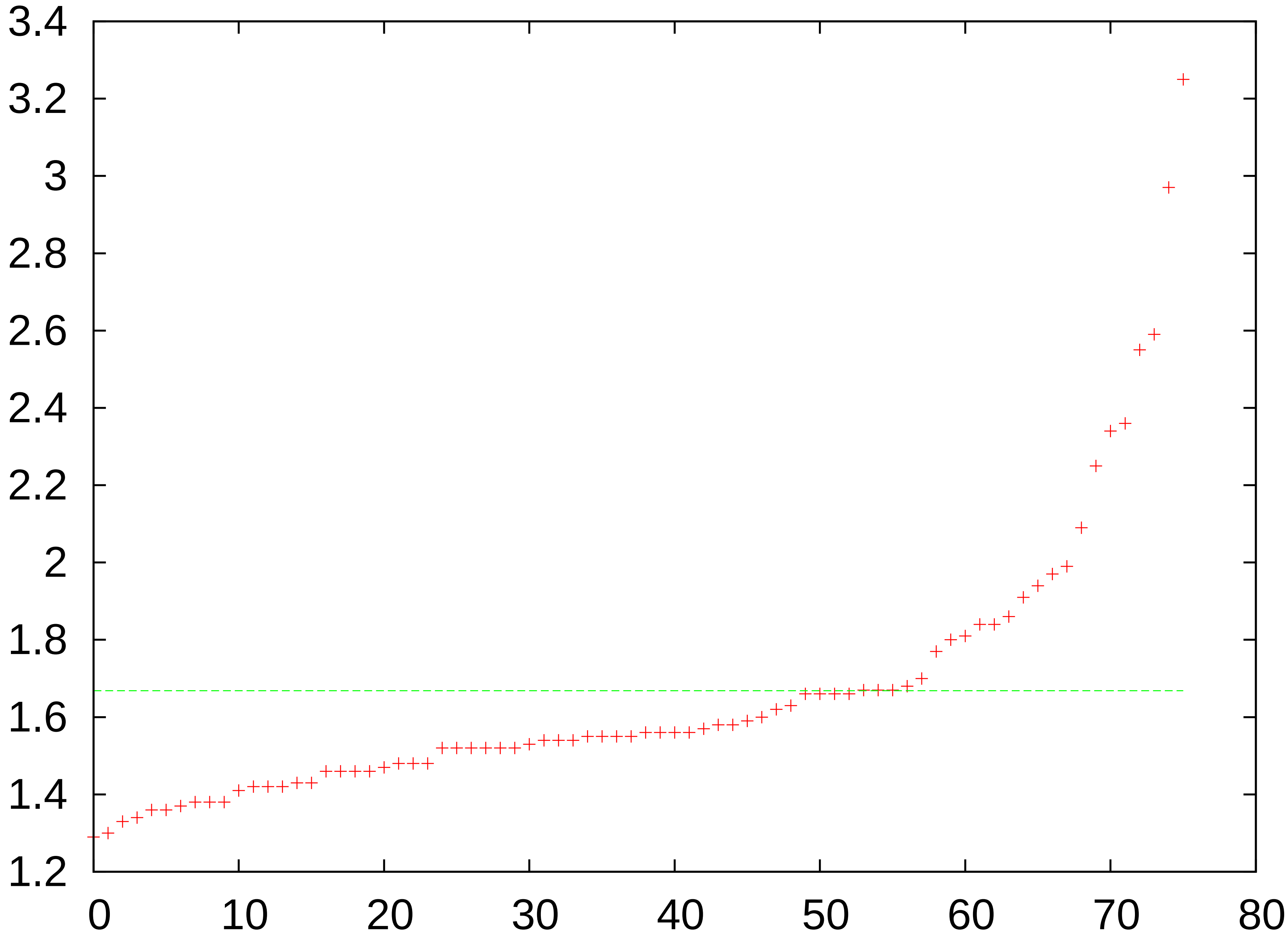


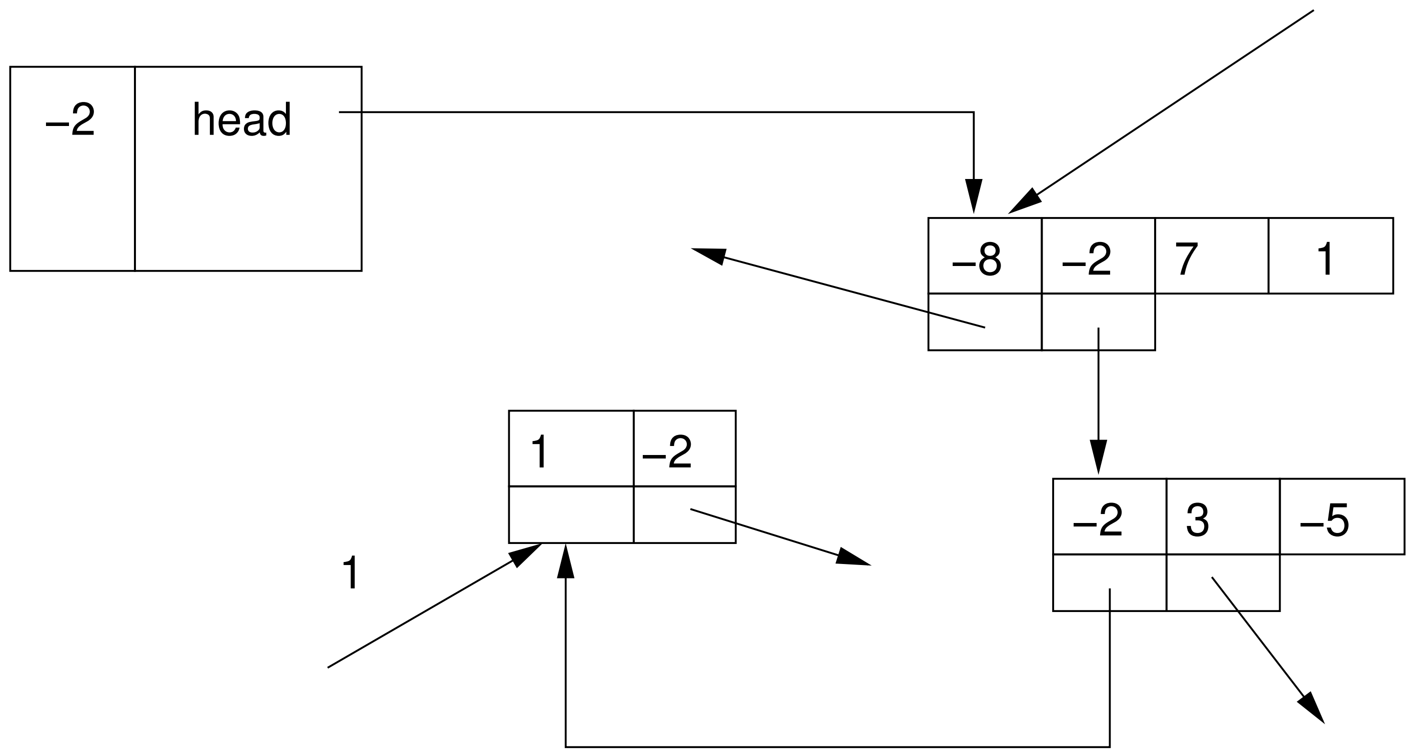

−2
start
top
end
−8
Watcher of A
1
−2
7
−8
A
−2
3
−5
B
Watcher of B
3



invariant: first two literals are watched

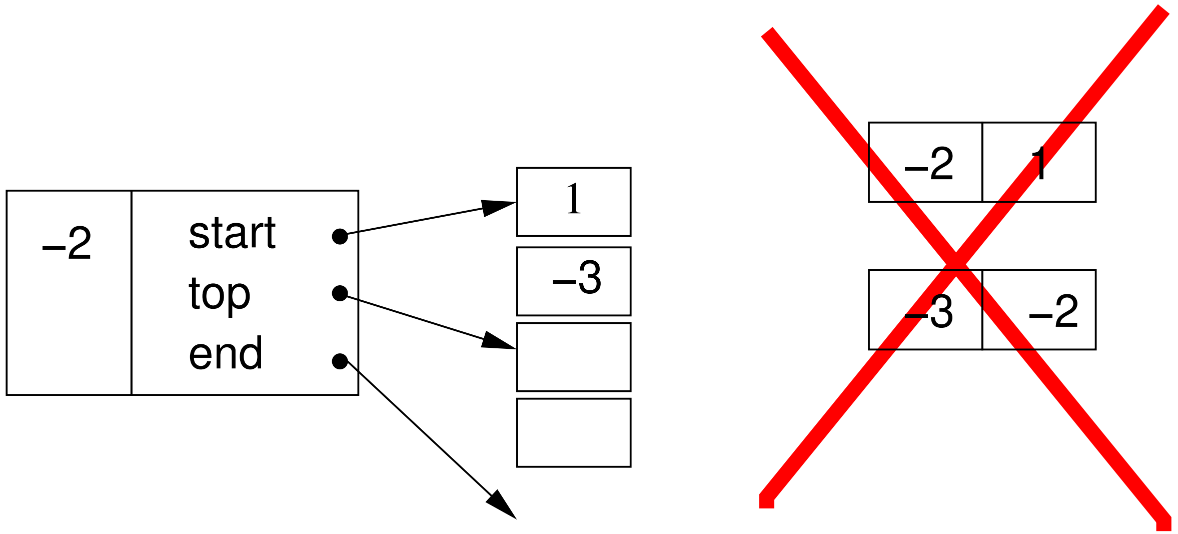
Average Number Literals Traversed Per Visited Clause



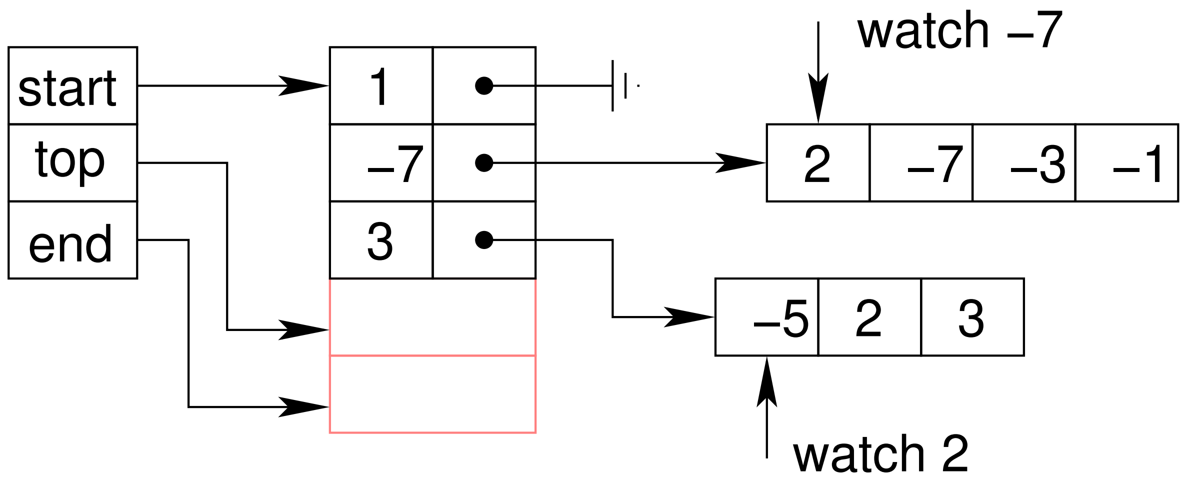


invariant: first two literals are watched

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