Modern SAT Solvers

Part A

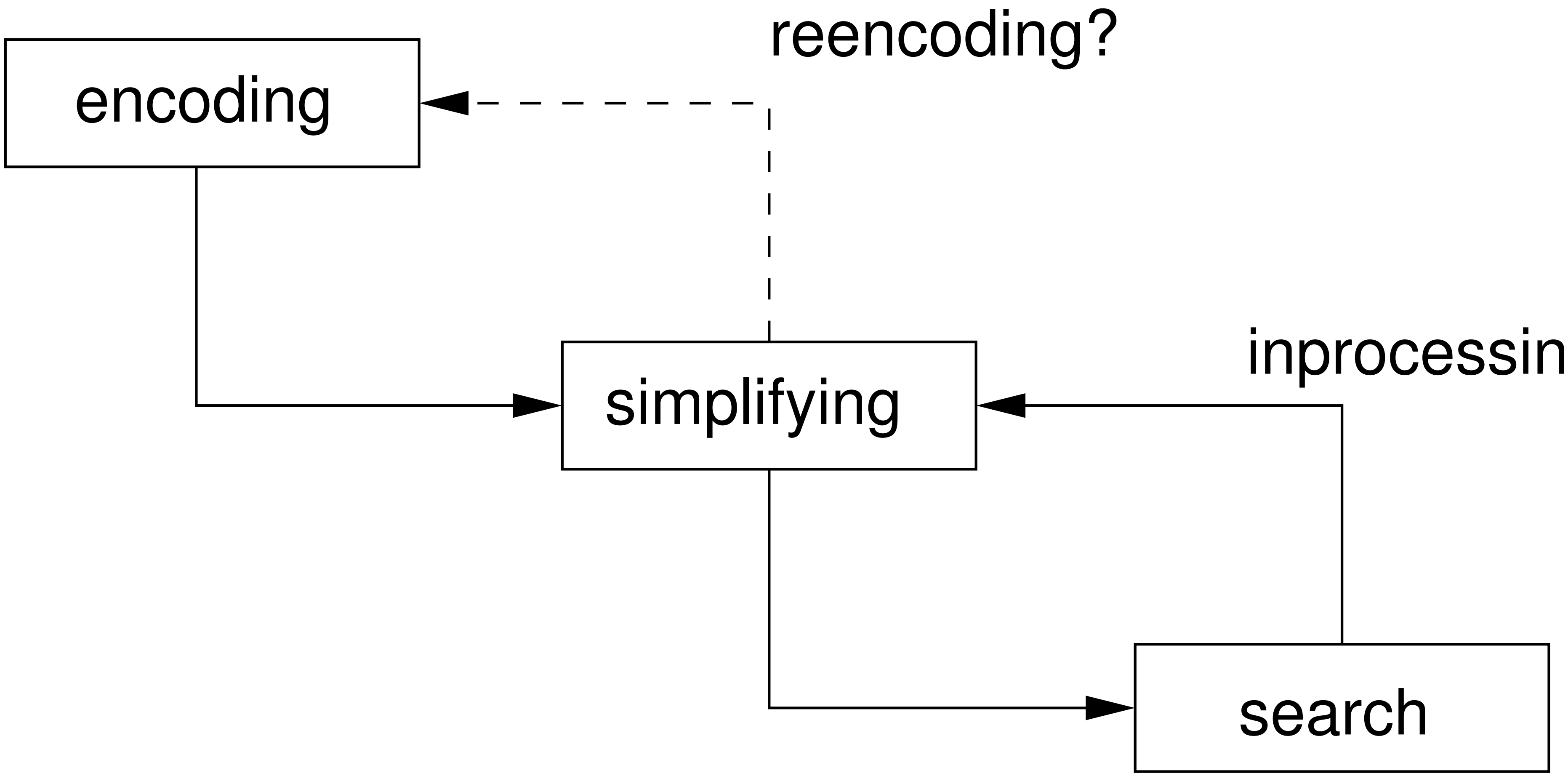
Vienna Winter School on Verification

6. February 2012

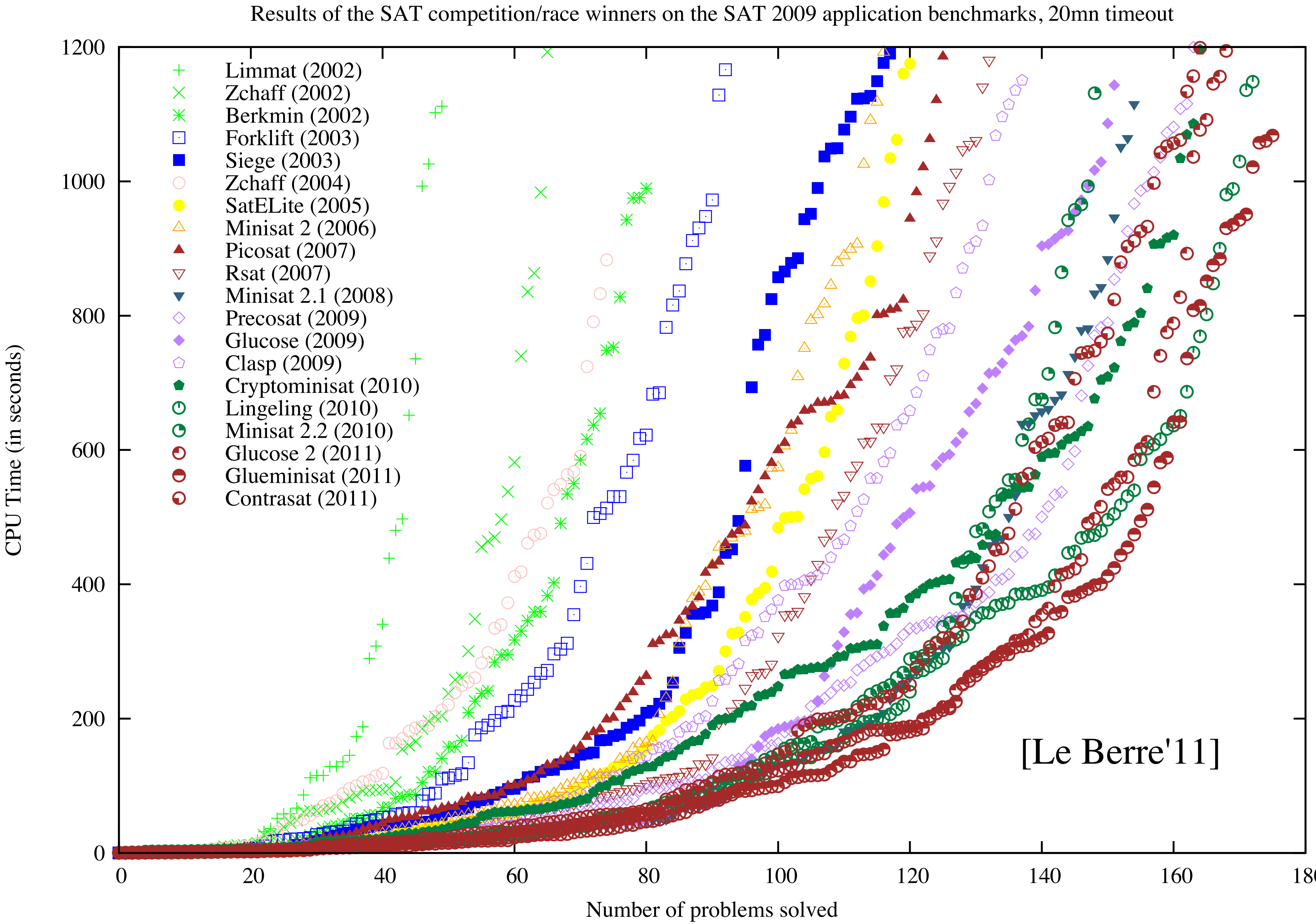
TU Vienna, Austria

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

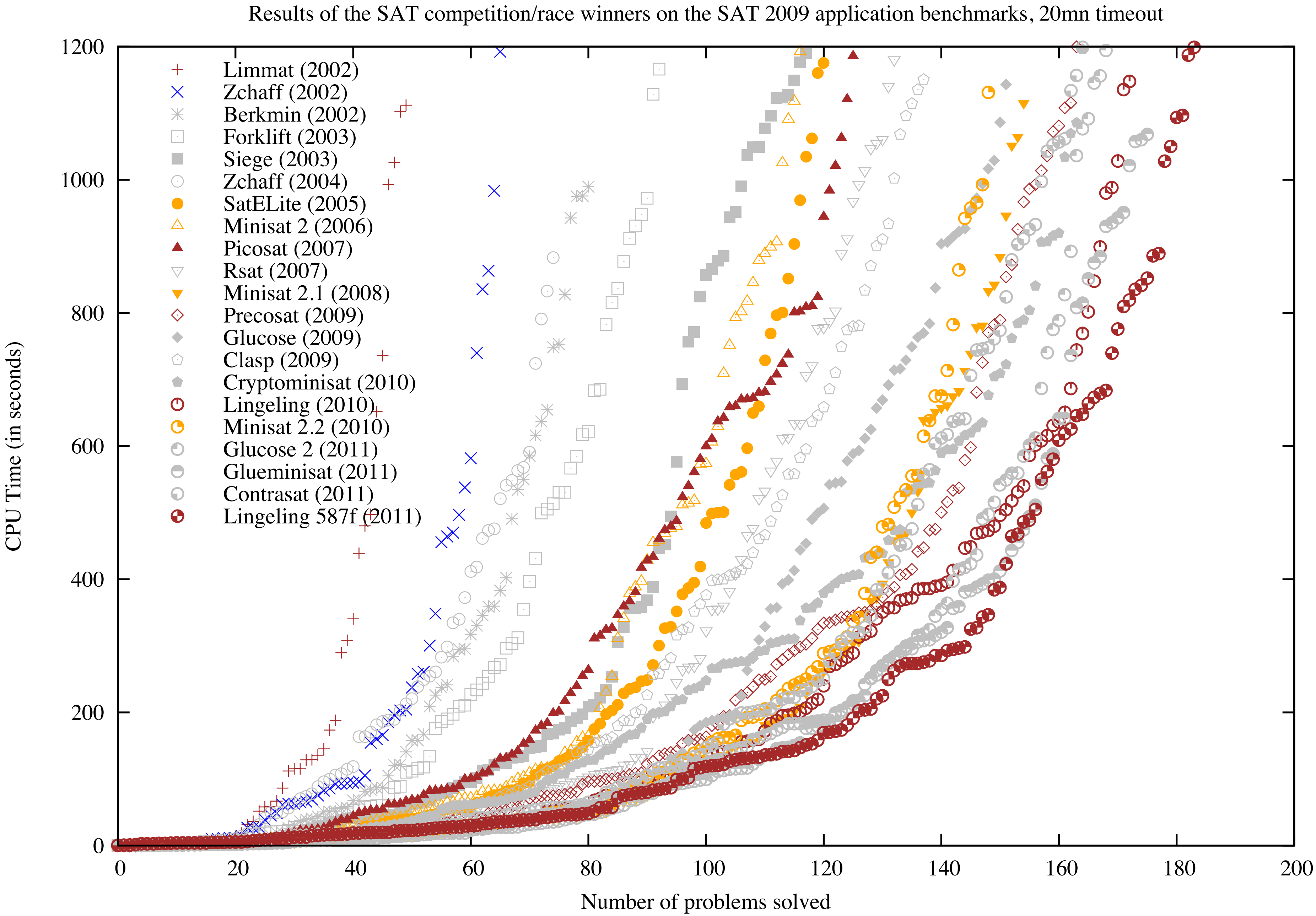
http://fmv.jku.at


reencoding?
encoding
simplifying
inprocessing
search

SAT Competition / Race Winners on SC 2009 Application Benchmarks



RiSE


Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout
CPU Time (in seconds)
Number of problems solved
0
200
400
600
800
1000
1200
0
20
40
60
80
100
120
140
160
180
200
Limmat (2002)
Zchaff (2002)
Berkmin (2002)
Forklift (2003)
Siege (2003)
Zchaff (2004)
SatELite (2005)
Minisat 2 (2006)
Picosat (2007)
Rsat (2007)
Minisat 2.1 (2008)
Precosat (2009)
Glucose (2009)
Clasp (2009)
Cryptominisat (2010)
Lingeling (2010)
Minisat 2.2 (2010)
Glucose 2 (2011)
Glueminisat (2011)
Contrasat (2011)
Lingeling 587f (2011)

original C code		**optimized C code**

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

$\Downarrow$		$\Uparrow$

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

How to check that these two versions are equivalent?

SAT Example cont.

1. represent procedures as *independent* boolean variables

$$
\begin{array}{ll}
\mathit{original} := & \mathit{optimized} := \\
\quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h & \quad \textbf{if } a \textbf{ then } f \\
\quad \textbf{else if } \neg a \textbf{ then } g & \quad \textbf{else if } b \textbf{ then } g \\
\quad \textbf{else } f & \quad \textbf{else } h
\end{array}
$$

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 the following boolean formulae

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

4. same problem as checking the following formula to be **unsatisfiable**

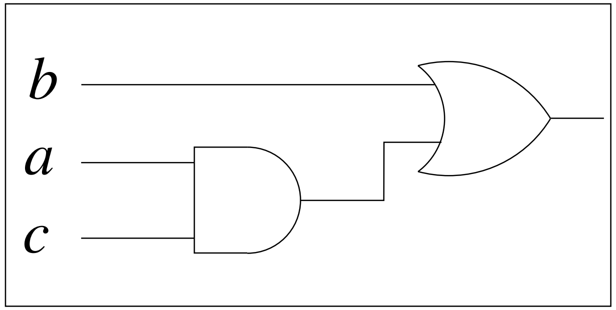
$$\text{compile}(\mathit{original}) \not\leftrightarrow \text{compile}(\mathit{optimized})$$

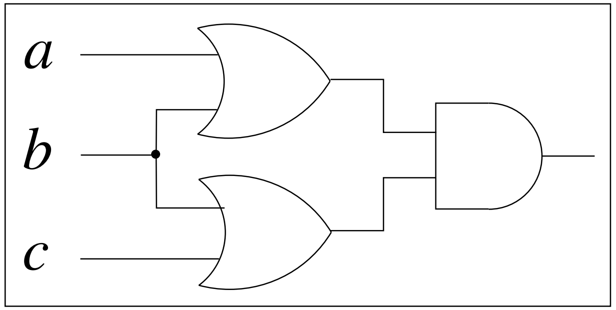
$$
\begin{aligned}
\textit{original} \quad &\equiv \quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv \quad (\neg a \wedge \neg b) \wedge h \ \vee \ \neg(\neg a \wedge \neg b) \wedge \textbf{ if } \neg a \textbf{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

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

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

SAT Example: Circuit Equivalence Checking





$b \vee a \wedge c$

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

equivalent?

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

SAT (Satisfiability) the classical NP complete Problem:

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

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

SAT belongs to NP

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

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

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

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

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

Definition

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

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

each clause C is a disjunction of literals

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

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

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

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

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

Note 3: SAT solvers mostly also expect CNF as input

Translation into CNF via NNF

Negation Normal Form (NNF)   AND/OR form + negations only occur in front of variables

use De'Morgan (push negations inward) to translate into NNF

$$
\begin{aligned}
a \leftrightarrow (b \wedge a) \quad &\equiv \quad (a \rightarrow (b \wedge a)) \wedge (a \leftarrow (b \wedge a)) \\
&\equiv \quad (\bar{a} \vee (b \wedge a)) \wedge (a \vee \overline{(b \wedge a)}) \\
&\equiv \quad \boxed{(\bar{a} \vee (b \wedge a)) \wedge (a \vee (\bar{b} \vee \bar{a}))} \qquad \text{in NNF}
\end{aligned}
$$

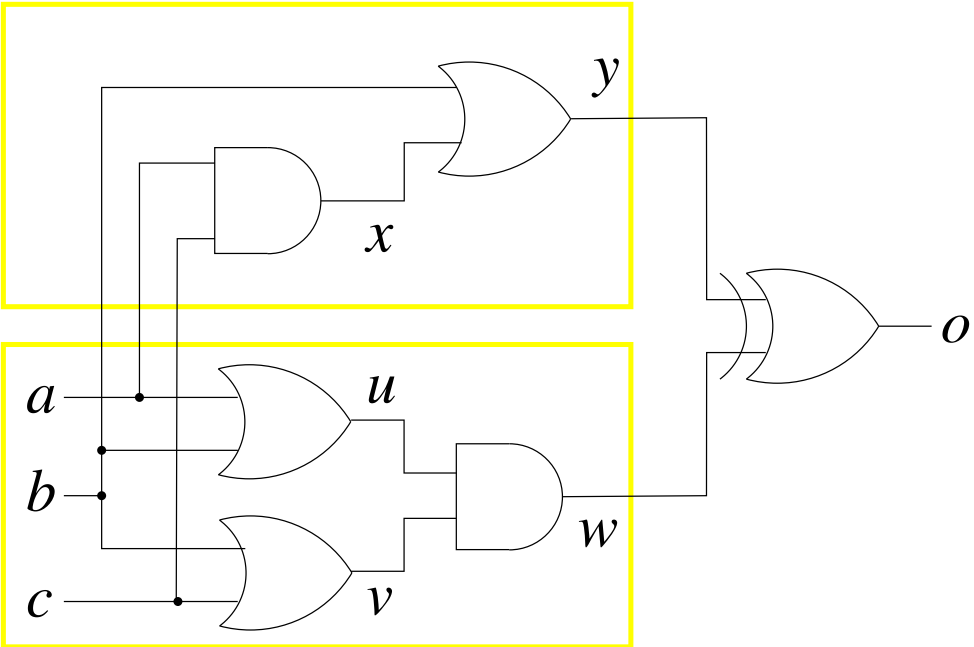
use distributivity of OR over AND   ("multiply out *outer* $\vee$")

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

and simplify to finally obtain $(\bar{a} \vee b)$

unfortunaly really expensive: $(\bigwedge C_i) \vee (\bigwedge D_j) \equiv \bigwedge (C_i \vee D_j)$ $O(n^2)$

CNF



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

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

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

Tseitin Transformation: Input / Output Constraints

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

Davis & Putnam Procedure (DP)

- dates back to the 50'ies:

  1st version is *resolution based*

  second version splits space for time

- **ideas:**

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

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

  works for very large instances

- recent ($\leq$ 15 years) optimizations:

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

  (we will have a look at each of them)

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 two antecedent clauses $C \cup \{v\}$ and $D \cup \{\neg v\}$,

both above the line, on the variable v, results in the

resolvent (clause) $D \cup C$ below the line.

Eliminating Variables with Resolution

1. pick variable x

2. add all resolvents on x

3. remove all clauses with x and $\bar{x}$

For instance given: $(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee \bar{c})$

Resolvents on a: $\dfrac{(a \vee b) \quad (\bar{a} \vee b)}{b} \quad \dfrac{(a \vee b) \quad (\bar{a} \vee \bar{b} \vee c)}{\bar{b} \vee c} \quad \dfrac{(a \vee b) \quad (\bar{a} \vee \bar{b} \vee \bar{c})}{\bar{b} \vee \bar{c}}$

Remaining clauses after removing all clauses containing a or $\bar{a}$: $b \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$

Resolving on b gives the remaining clauses $c \wedge \bar{c}$

Which finally (resolving on c) gives the inconsistent **empty clause**

---

corresponds to eliminate a Tseitin variable for OR by distributivity

Problems with Resolution Based DP

- if variables have *many* occurences, then *many* resolvents are added
  - 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
- currently practically only useful
  - in the context of bounded variable elimination (preprocessing)
  - as in SatELite preprocessor [EénBiere05]

DPLL Procedure

$DPLL(F)$

$F := BCP(F)$ 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*

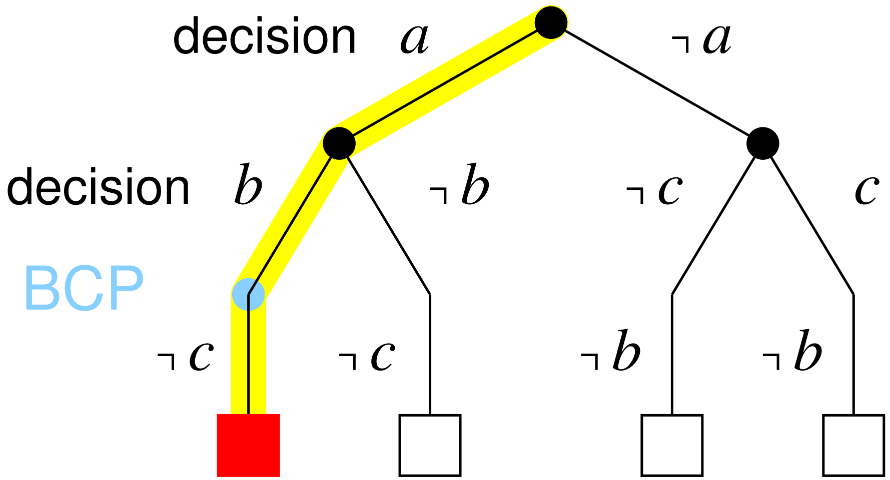
return $DPLL(F \wedge \{\neg l\}$

DPLL Example

$a = 1$

$b = 1$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

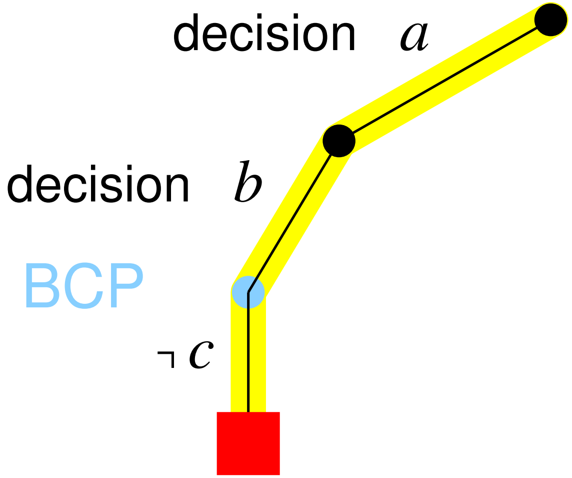
$a \vee b \vee c$

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 1$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

learn $\neg a \vee \neg b$

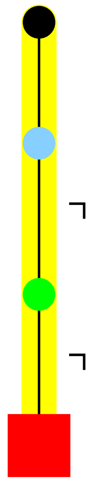
Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$

decision a



$\neg b$ BCP

$\neg c$ BCP

clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

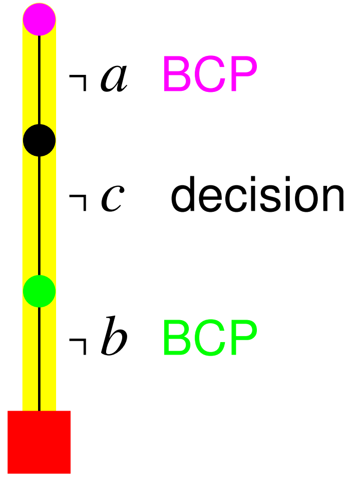
$\neg a \vee \neg b$

learn $\neg a$

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

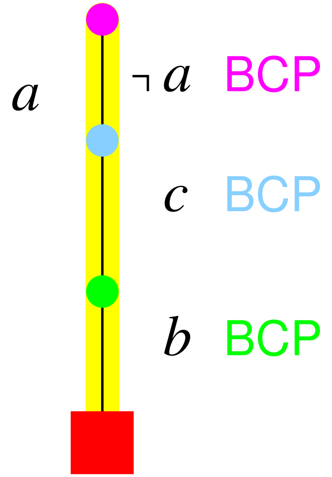
learn c

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

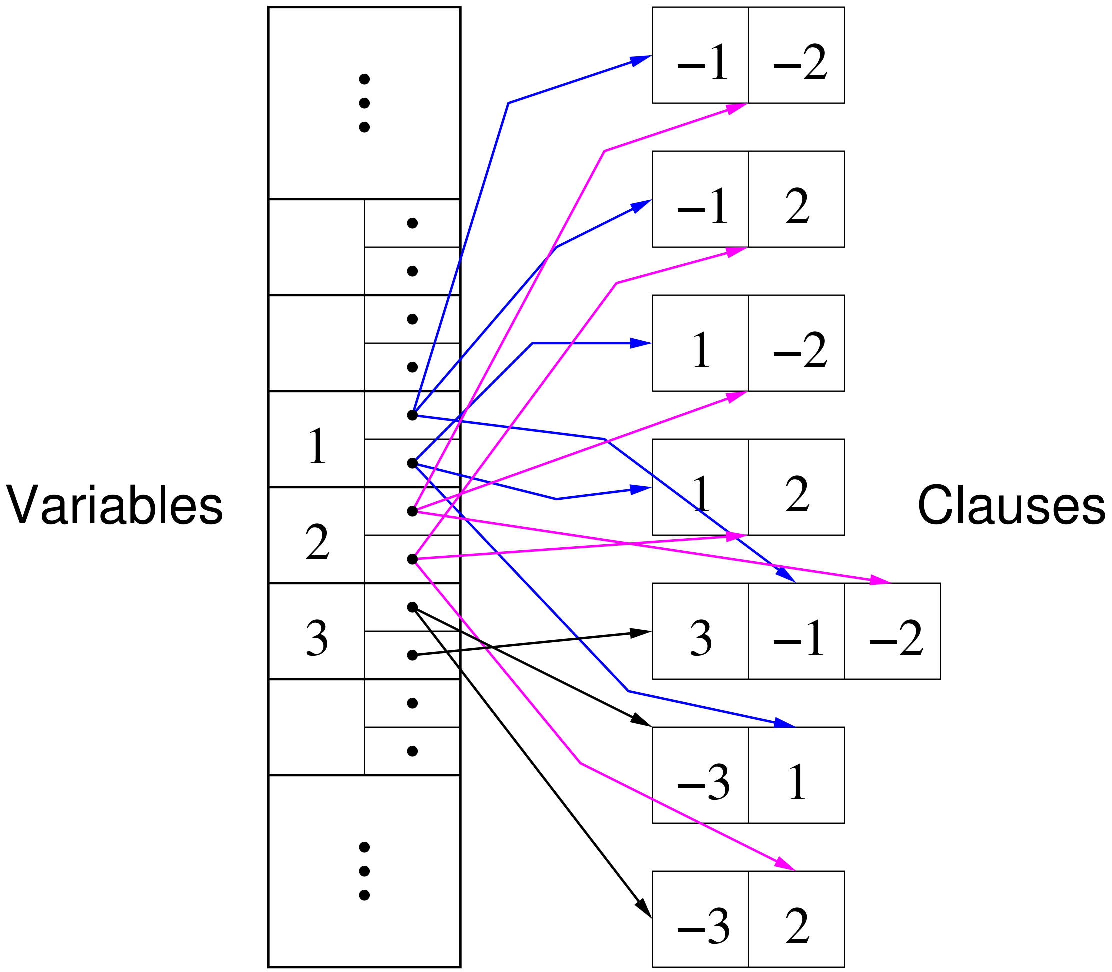
$\neg a$

c

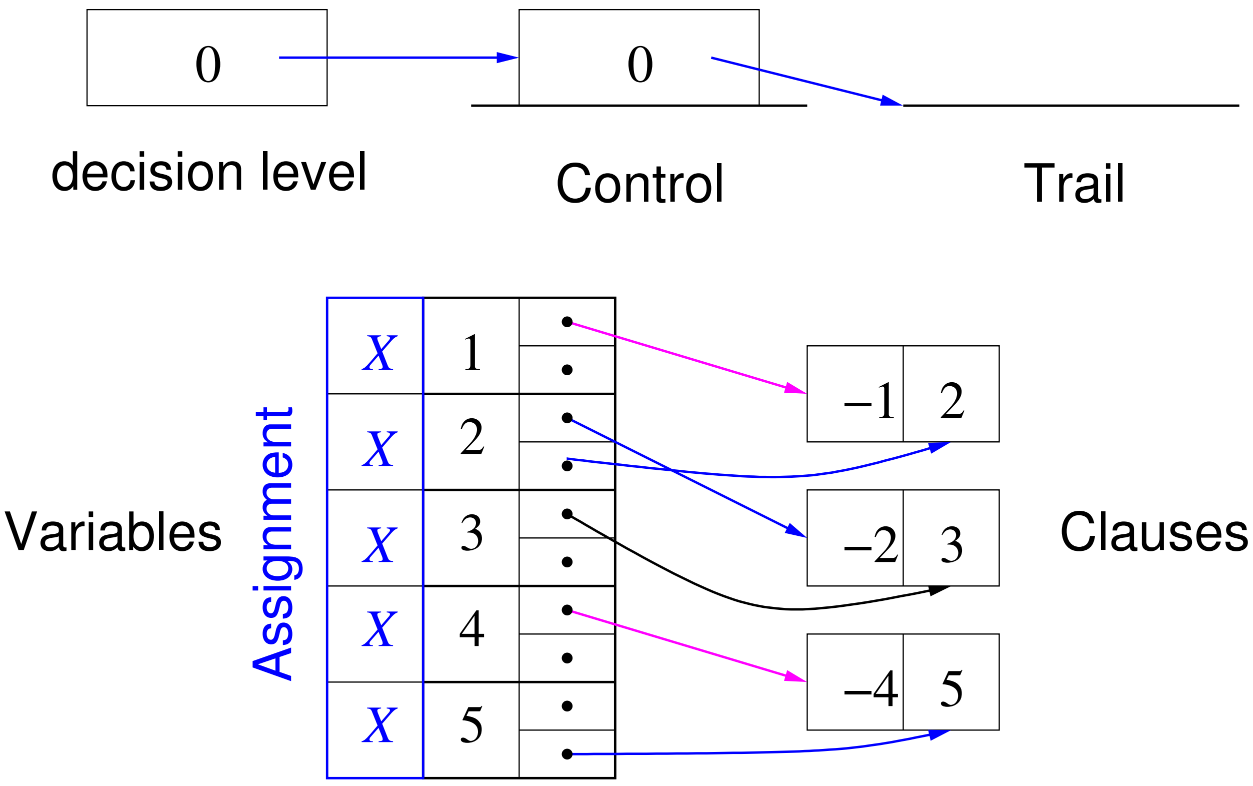
learn $\perp$

empty clause

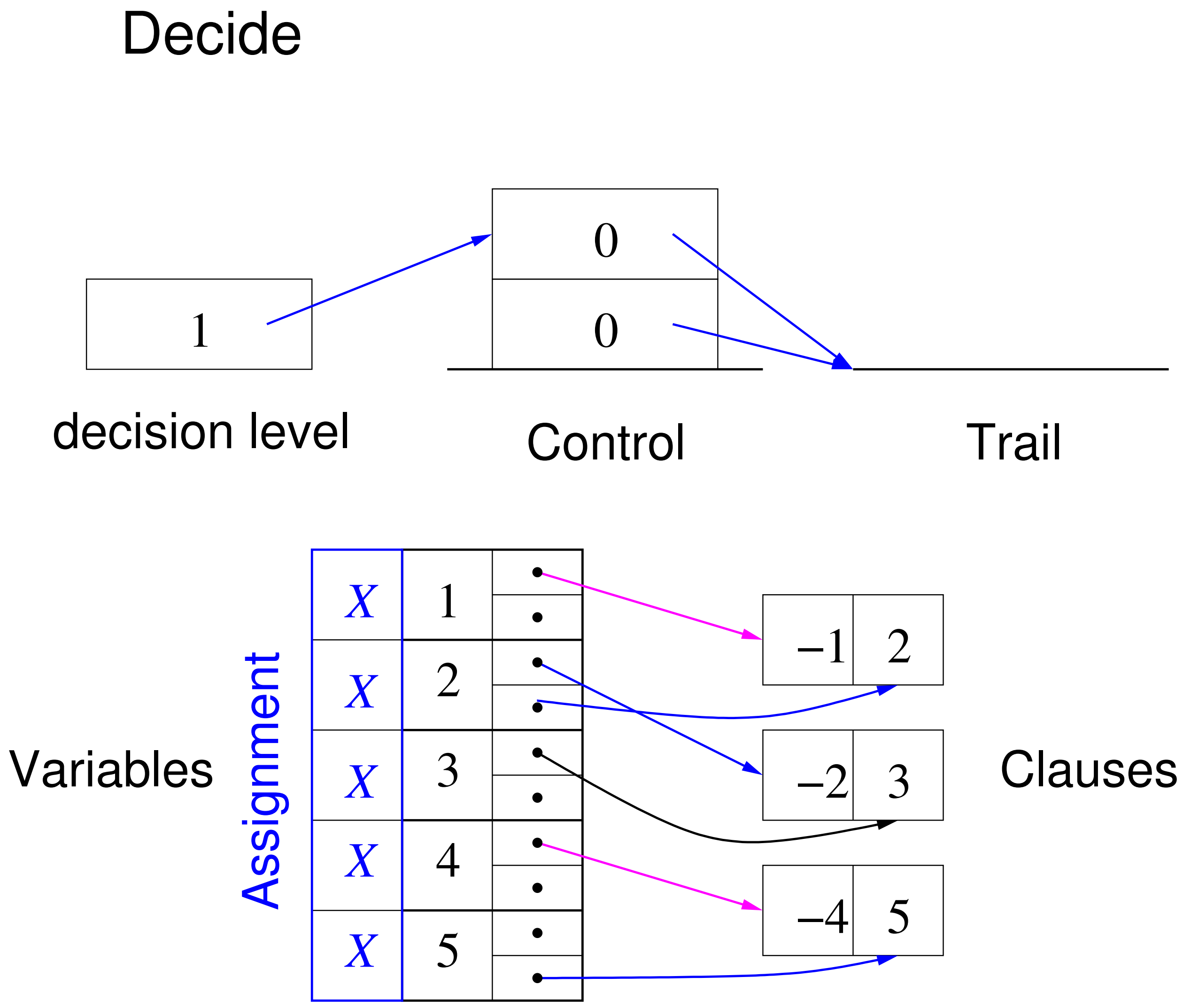
Simple Data Structures in DP Implementation

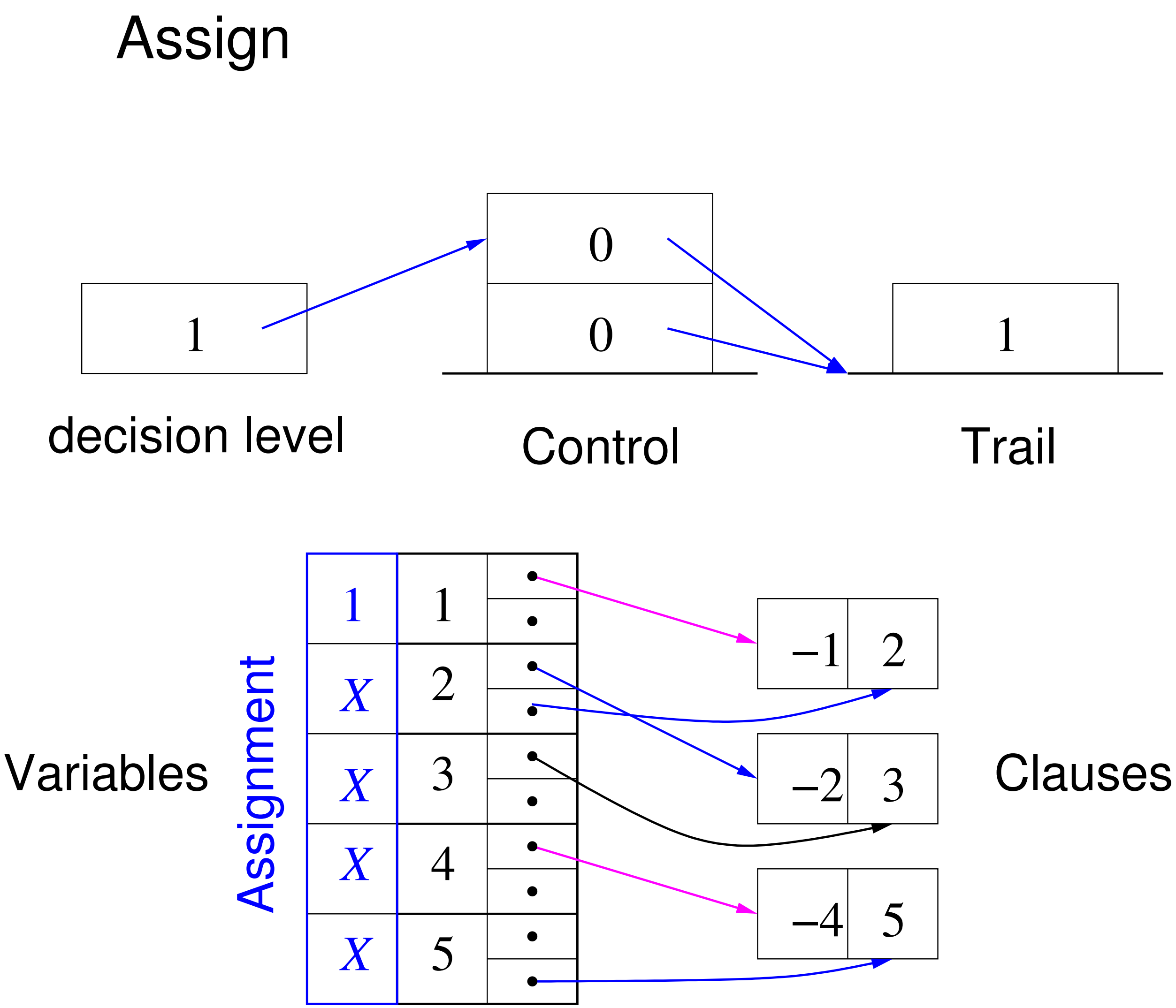


BCP Example

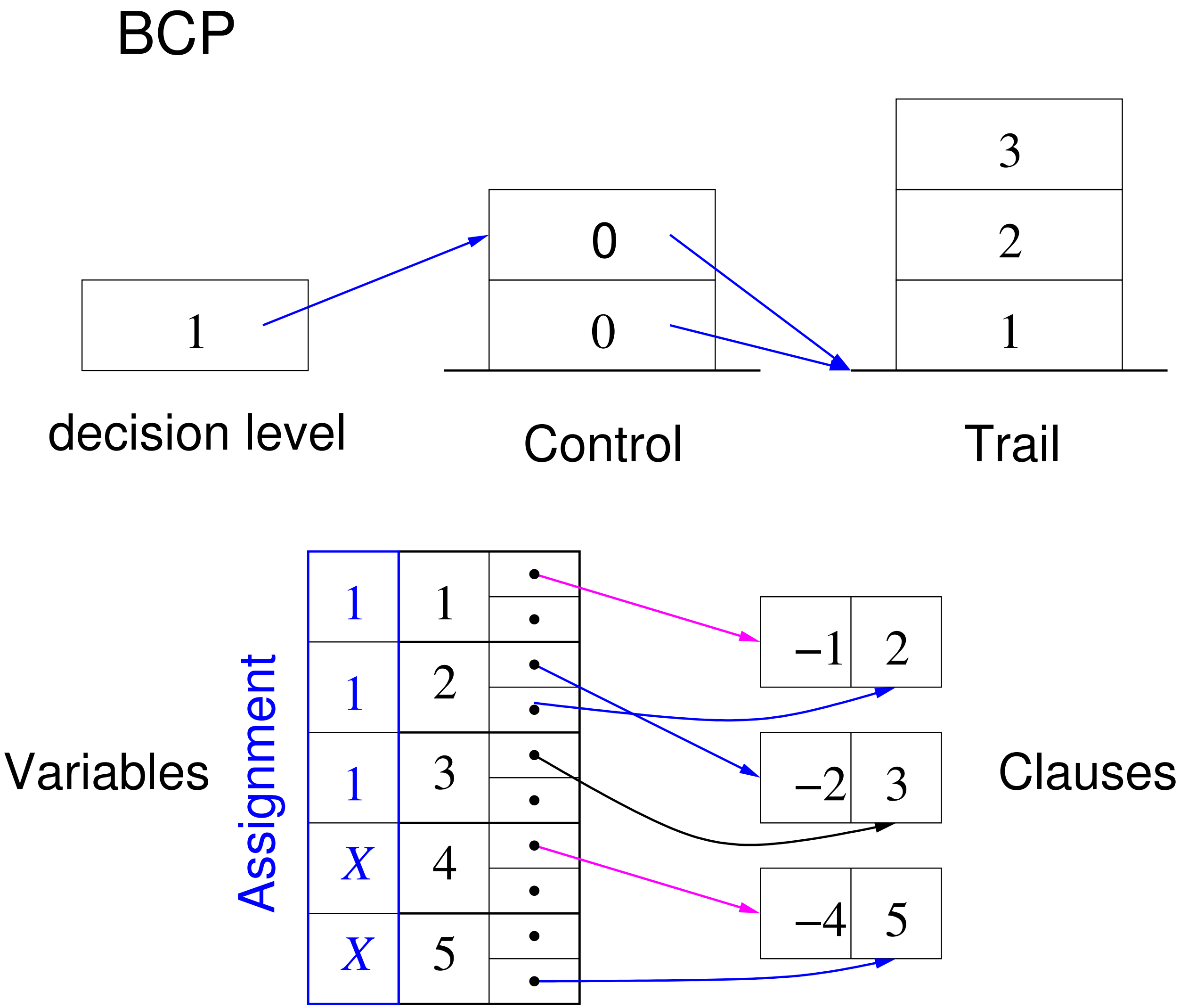

0
0
decision level
Control
Trail
Variables
Assignment
X
X
X
X
X
1
2
3
4
5
−1
2
−2
3
−4
5
Clauses

Example cont.

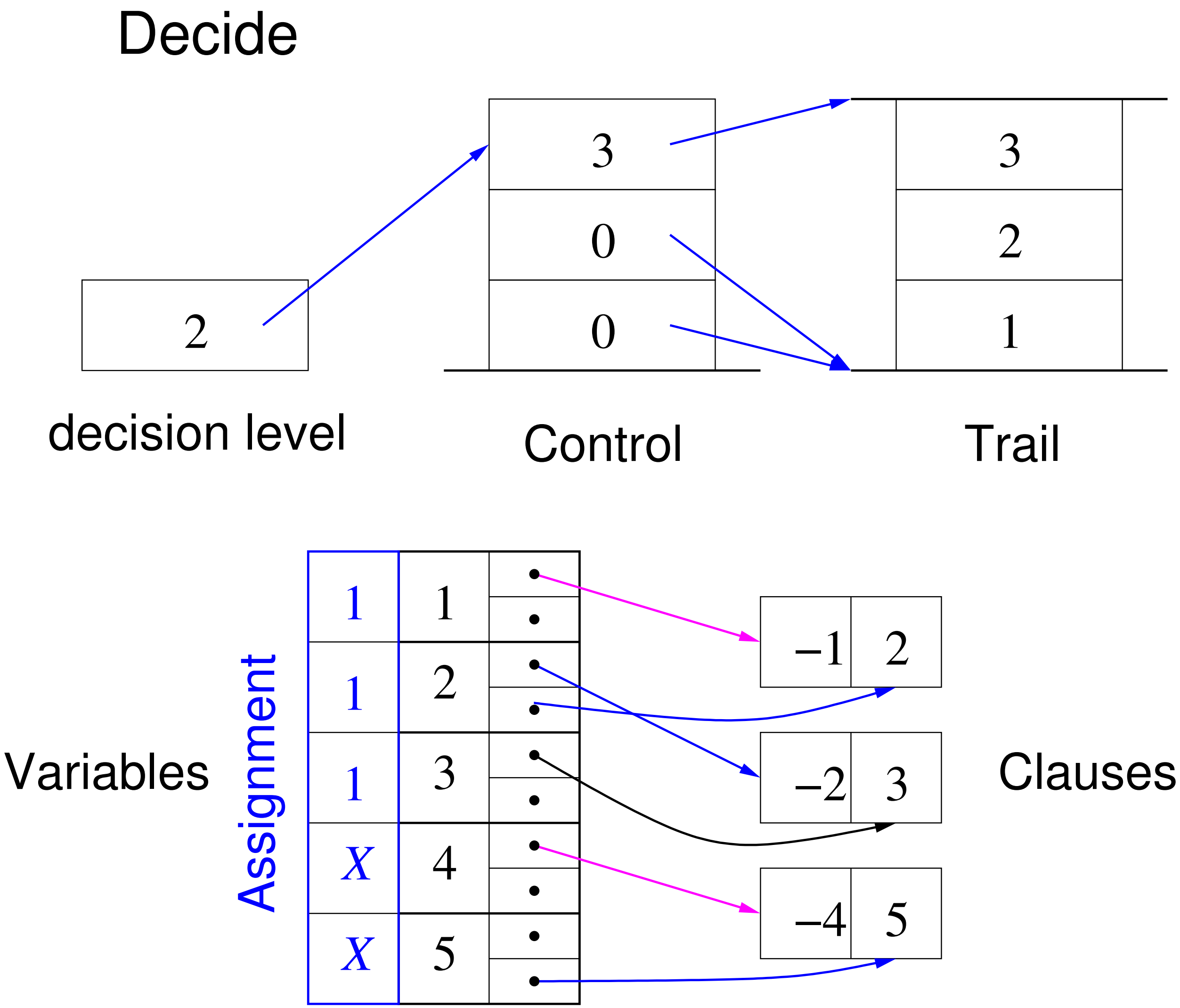



Assign
1
0
0
1
decision level
Control
Trail
1
1
X
2
X
3
X
4
X
5
Assignment
Variables
−1
2
−2
3
−4
5
Clauses

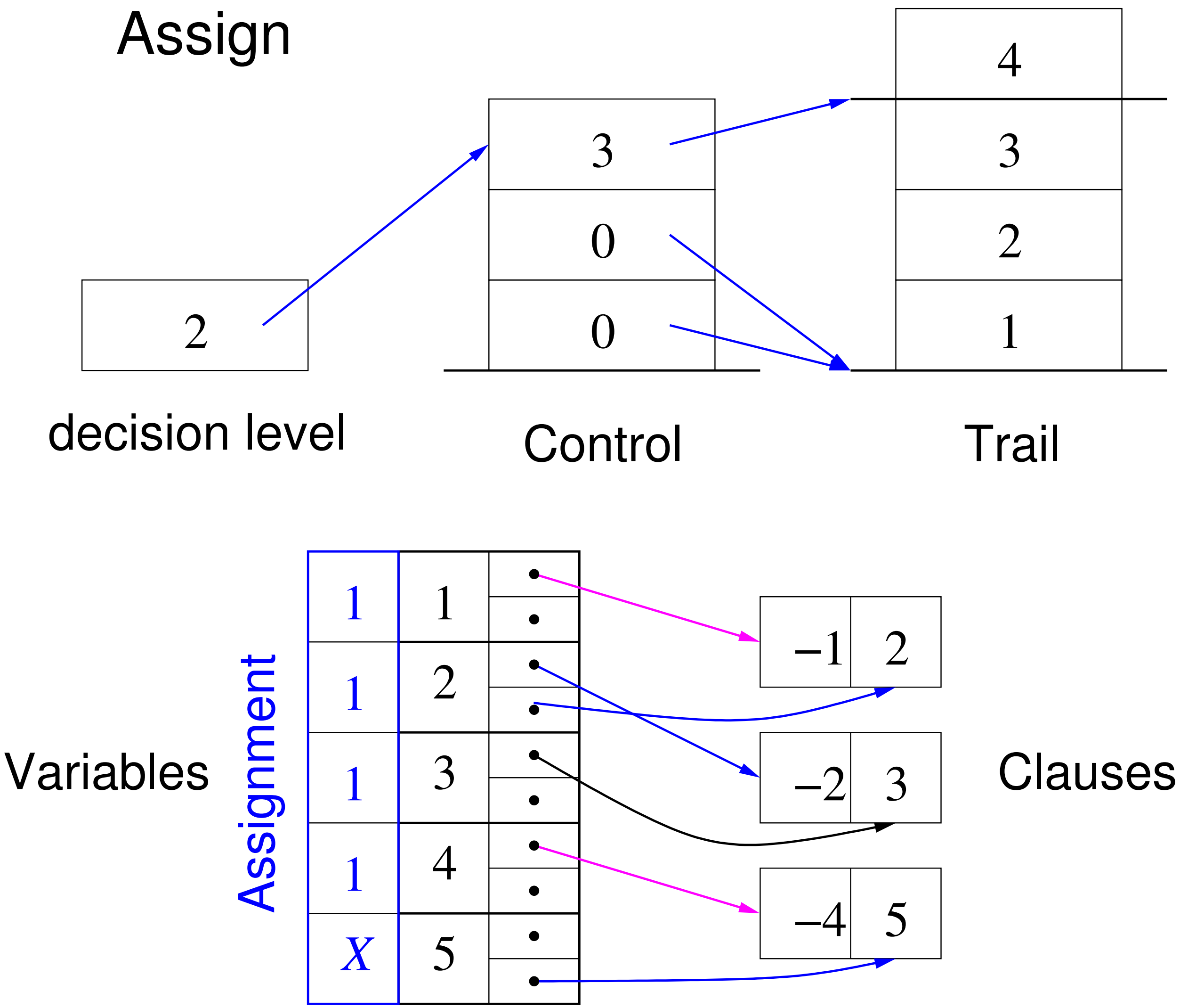
Example cont.



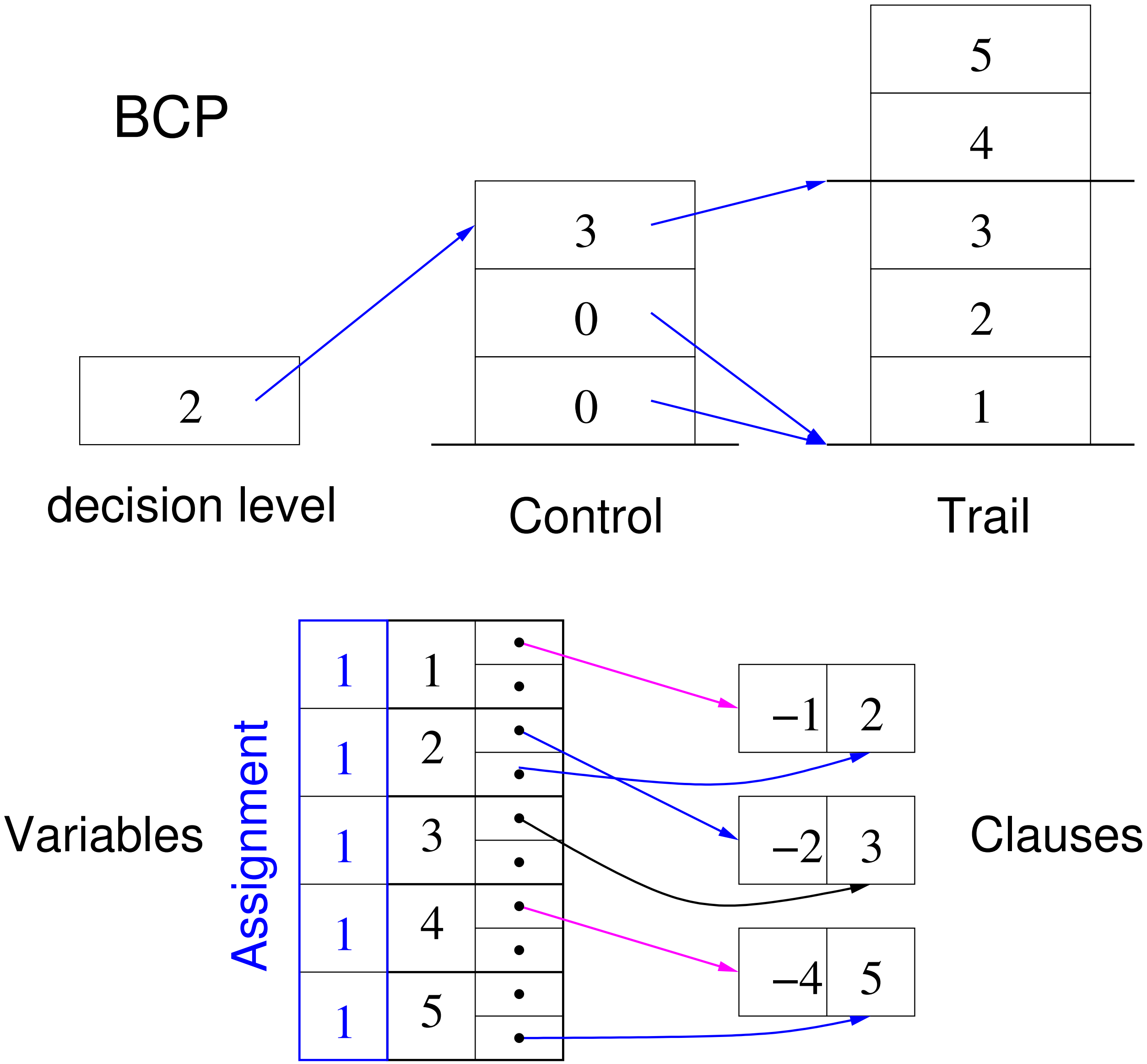
Example cont.



Example cont.



Example cont.



Decision Heuristics

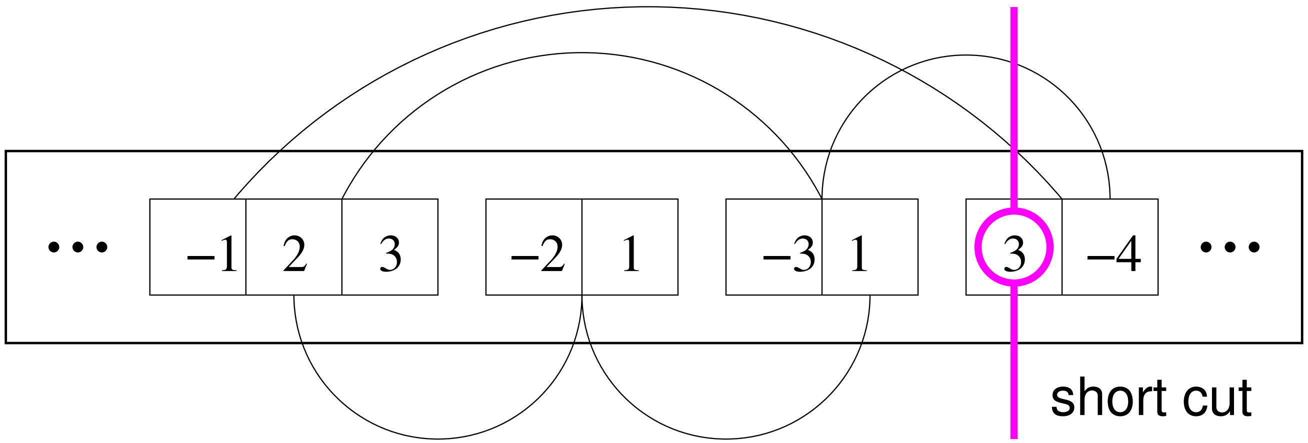
- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast to compute, since only calculated once
  - and thus can also use more expensive algorithms

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

Cut Width Heuristics

- not really used in practice, but instructive to understand why SAT solvers might work
- 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

Cut Width Algorithm


```
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;
}
```

Other popular Decision Heuristics

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

- look-ahead heuristics (e.g. SATZ or MARCH solver) **failed literals, probing**
  - do trial assignments and BCP for all unassigned variables (both phases)
  - if BCP leads to conflict, force toggled assignment of current trial decision
  - optionally learn binary clauses and perform equivalent literal substitution
  - decision: most balanced w.r.t. prop. assignments / sat. clauses / reduced clauses
  - see also our recent Cube & Conquer paper [HeuleKullmanWieringaBiere-HVC'11]

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 EVSIDS

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

Relating EVSIDS and NVSIDS

(consider only one variable)

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

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

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

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

BerkMin's Dynamic Second Order Heuristics

[GoldbergNovikov-DATE'02]

- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to 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)
- mixed results as other variants VMTF, CMTF (var/clause move to front)