SAT Solving and CDCL(T)

Mate Soos

SAT Winter School'2019

IIT Bombay, India

December 7, 2019

Based on slides by **Armin Biere**

About Me

- PhD at INRIA Grenoble 2009
- Maintainer of CryptoMiniSat, STP, ApproxMC
- Working as a Senior Research Fellow at National University of Singapore (3mo a year)
- Working as a Senior IT Security Architect at Zalando (9mo a year)
- Interests: Higher level abstractions, Counting, Inprocessing, ML, Visualisation

Dress Code Tutorial Speaker as SAT Problem

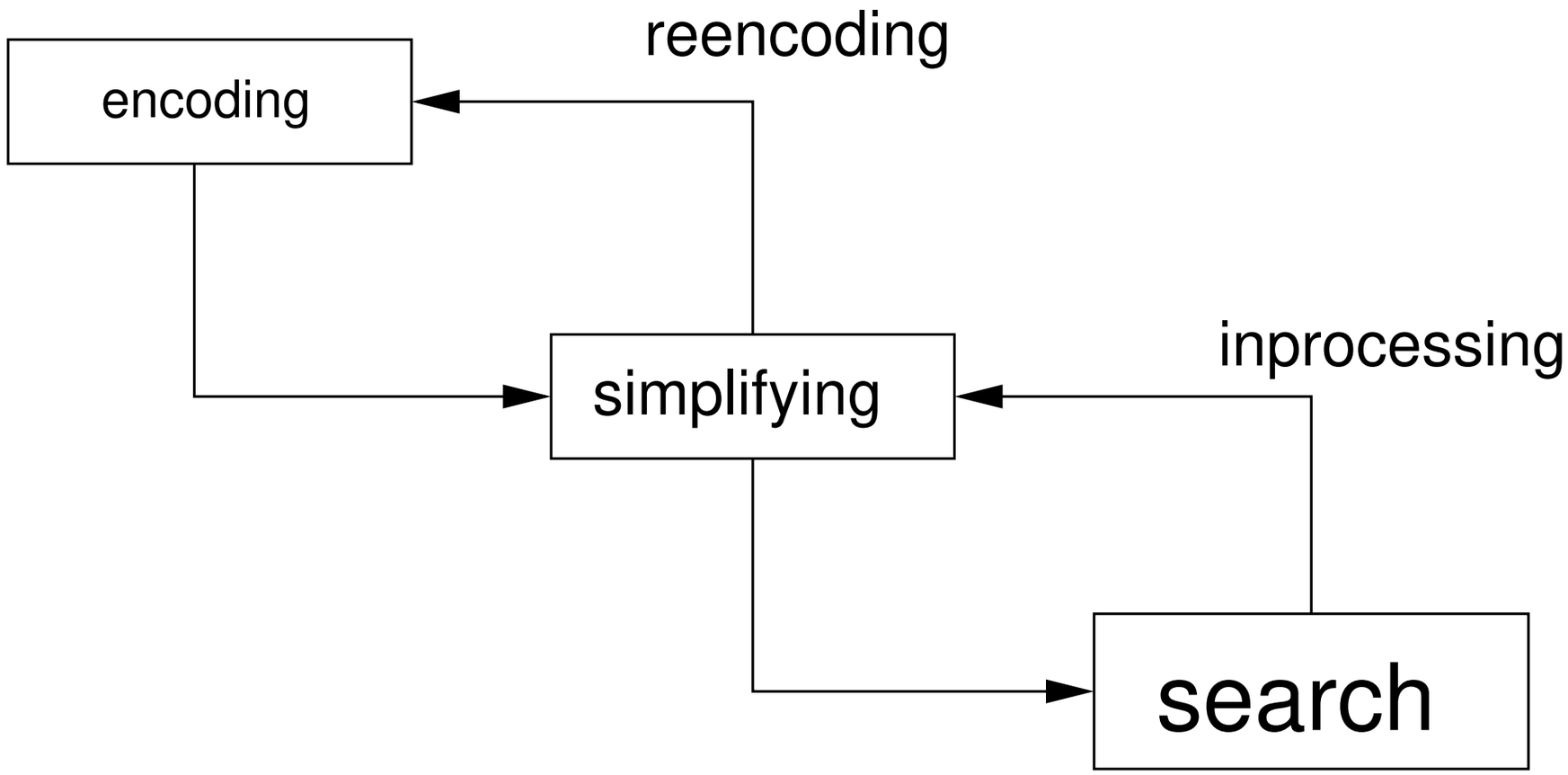
- propositional logic:
  - variables **jewellery** **shirt**
  - negation $\neg$ (not)
  - disjunction $\vee$ (or)
  - conjunction $\wedge$ (and)
- clauses (conditions / constraints)

1. clearly one should not wear a **jewellery** without a **shirt** $\neg\textbf{jewellery} \vee \textbf{shirt}$
2. not wearing a **jewellery** nor a **shirt** is impolite $\textbf{jewellery} \vee \textbf{shirt}$
3. wearing a **jewellery** and a **shirt** is overkill $\neg(\textbf{jewellery} \wedge \textbf{shirt}) \equiv \neg\textbf{jewellery} \vee \neg\textbf{shirt}$

- Is this formula in conjunctive normal form (CNF) **satisfiable?**

$$(\neg\textbf{jewellery} \vee \textbf{shirt}) \wedge (\textbf{jewellery} \vee \textbf{shirt}) \wedge (\neg\textbf{jewellery} \vee \neg\textbf{shirt})$$

What is Practical SAT Solving?



Equivalence Checking 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

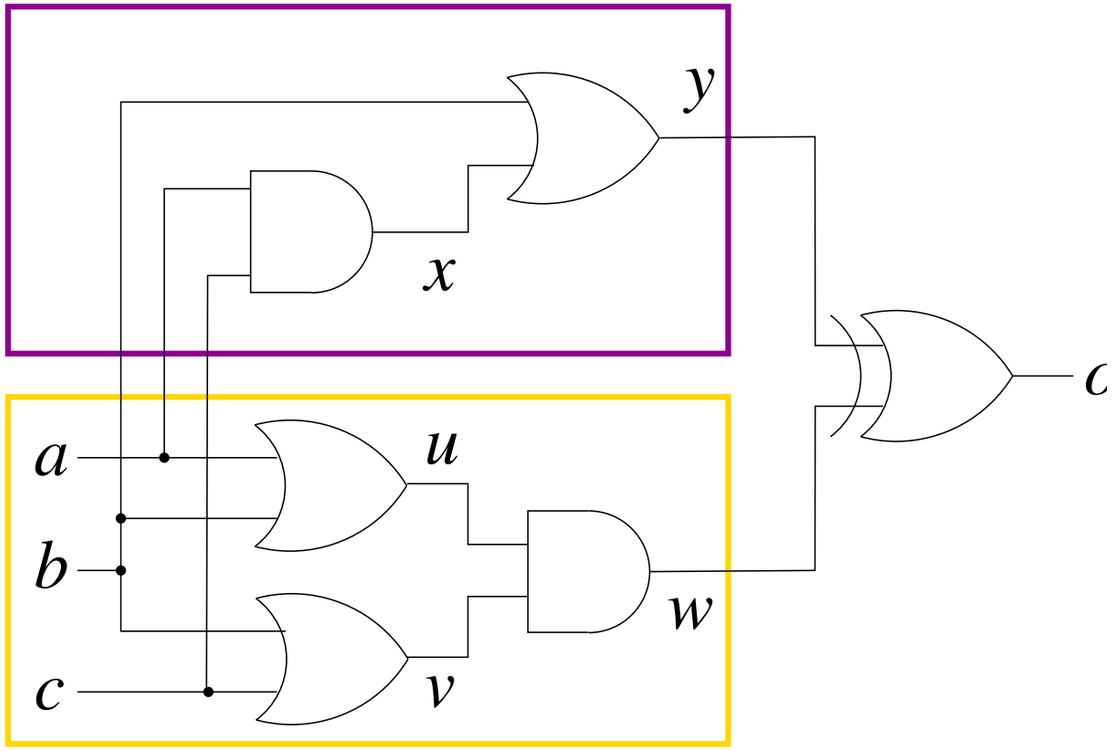
```
if(a) f();
else if(b) g();
else h();
```

$\Uparrow$

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

How to check that these two versions are equivalent?

Tseitin Transformation: Circuit to CNF



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

$$
\begin{aligned}
& o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge \\
& (x \leftarrow a \wedge c) \wedge \ \ldots
\end{aligned}
$$

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

Tseitin Transformation: Gate Constraints

Negation:

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

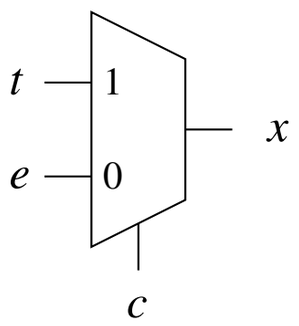
Disjunction:

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

Conjunction:

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

Tseitin Encoding of If-Then-Else Gate



$$
\begin{aligned}
x \leftrightarrow (c\ ?\ t : e) \quad &\Leftrightarrow \quad (x \rightarrow (c \rightarrow t)) \wedge (x \rightarrow (\bar{c} \rightarrow e)) \wedge (\bar{x} \rightarrow (c \rightarrow \bar{t})) \wedge (\bar{x} \rightarrow (\bar{c} \rightarrow \bar{e})) \\
&\Leftrightarrow \quad (\bar{x} \vee \bar{c} \vee t) \wedge (\bar{x} \vee c \vee e) \wedge (x \vee \bar{c} \vee \bar{t}) \wedge (x \vee c \vee \bar{e})
\end{aligned}
$$

minimal but <u>not</u> arc consistent:

- if t and e have the same value then x needs to have that too
- possible additional clauses

$$(\bar{t} \wedge \bar{e} \rightarrow \bar{x}) \equiv (t \vee e \vee \bar{x}) \qquad\qquad (t \wedge e \rightarrow x) \equiv (\bar{t} \vee \bar{e} \vee x)$$

- but can be learned or derived through preprocessing (ternary resolution)
  keeping those clauses redundant is better in practice

Example of Logical Constraints: XOR Constraints

2-long XOR:

$$l_1 \oplus l_2 = 1 \quad \Leftrightarrow \quad \begin{array}{l} \bar{l}_1 \vee l_2 \wedge \\ l_1 \vee \bar{l}_2 \wedge \end{array}$$

3-long XOR:

$$l_1 \oplus l_2 \oplus l_3 = 1 \quad \Leftrightarrow \quad \begin{array}{l} l_1 \vee l_2 \vee l_3 \wedge \\ \bar{l}_1 \vee \bar{l}_2 \vee l_3 \wedge \\ \bar{l}_1 \vee l_2 \vee \bar{l}_3 \wedge \\ l_1 \vee \bar{l}_2 \vee \bar{l}_3 \wedge \end{array}$$

4-long XOR:

$$l_1 \oplus l_2 \oplus l_3 \oplus l_4 = 1 \quad \Leftrightarrow \quad \begin{array}{l} l_1 \vee l_2 \vee l_3 \vee l_4 \wedge \\ \bar{l}_1 \vee \bar{l}_2 \vee l_3 \vee l_4 \wedge \\ \bar{l}_1 \vee l_2 \vee \bar{l}_3 \vee l_4 \wedge \\ l_1 \vee \bar{l}_2 \vee \bar{l}_3 \vee l_4 \wedge \\ \bar{l}_1 \vee l_2 \vee l_3 \vee \bar{l}_4 \wedge \\ l_1 \vee \bar{l}_2 \vee l_3 \vee \bar{l}_4 \wedge \\ l_1 \vee l_2 \vee \bar{l}_3 \vee \bar{l}_4 \wedge \\ \bar{l}_1 \vee \bar{l}_2 \vee \bar{l}_3 \vee \bar{l}_4 \wedge \end{array}$$

In general, a k-long XOR constraint translates to 2^{k-1} clauses without helper variables

Example of Logical Constraints: XOR Constraints Cont.

We use helper variables to bring down the 2^{k-1} clauses needed:

$$l_1 \oplus l_2 \oplus l_3 \oplus l_4 \oplus l_5 \oplus l_6 \oplus l_7 = 1 \quad \Leftrightarrow \quad \begin{array}{l} l_1 \oplus l_2 \oplus l_3 \oplus h_1 \wedge \\ h_1 \oplus l_4 \oplus l_5 \oplus h_2 \wedge \\ h_3 \oplus l_6 \oplus l_7 \end{array}$$

Now we have:

- $\lfloor k-1/2 \rfloor$ helper variables
- $\lfloor (k-1)/2 \rfloor + \lceil k/2 \rceil$ XORs, each at most 4 long
- $\rightarrow$ the number of clauses needed is linear in k

Different trade-offs are possible, this is called the “cutting number”.

Example of Logical Constraints: Cardinality Constraints

- given a set of literals $\{l_1, \ldots l_n\}$
  - constraint the <u>number</u> of literals assigned to $true$
  - $l_1 + \cdots + l_n \geq k$ or $l_1 + \cdots + l_n \leq k$ or $l_1 + \cdots + l_n = k$
  - combined make up exactly all fully symmetric boolean functions
- multiple encodings of cardinality constraints
  - naive encoding exponential: <u>at-most-one</u> quadratic, <u>at-most-two</u> cubic, etc.
  - quadratic $O(k \cdot n)$ encoding goes back to Shannon
  - linear $O(n)$ parallel counter encoding [Sinz'05]
- many variants even for <u>at-most-one</u> constraints
  - for an $O(n \cdot \log n)$ encoding see Prestwich's chapter in Handbook of SAT
- typically arc consistency is expensive in terms of encoding

DIMACS Format

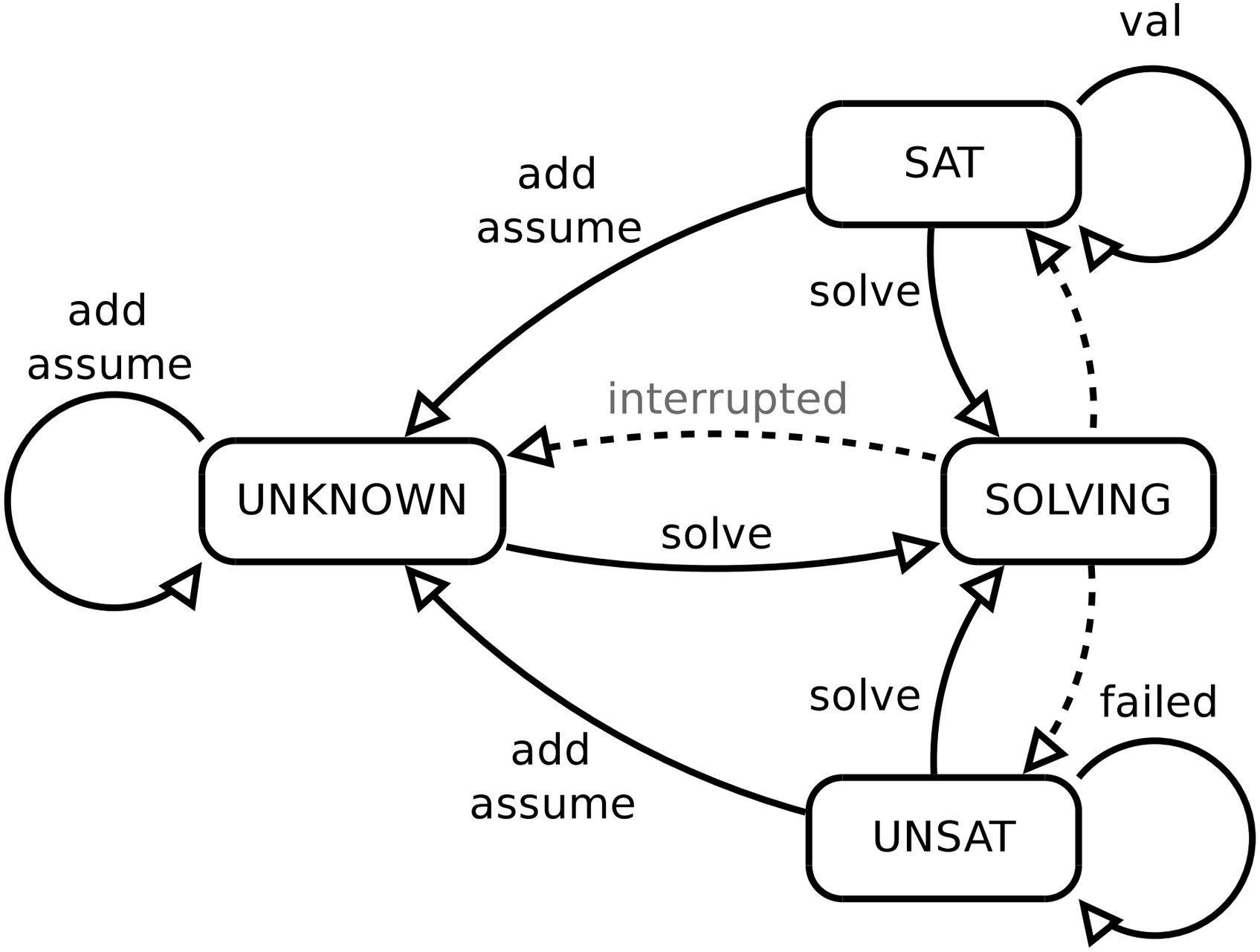
```
$ cat example.cnf
c comments start with 'c' and extend until the end of the line
c
c variables are encoded as integers:
c
c   'jewellery'   becomes '1'
c   'shirt' becomes '2'
c
c header 'p cnf <variables> <clauses>'
c
p cnf 2 3
-1  2 0           c  !jewellery  or  shirt
 1  2 0           c   jewellery  or  shirt
-1 -2 0           c  !jewellery  or !shirt
```

```
$ picosat example.cnf
s SATISFIABLE
v -1 2 0
```

SAT Application Programmatic Interface (API)

- incremental usage of SAT solvers
  - add facts such as clauses incrementally
  - call SAT solver and get satisfying assignments
  - optionally retract facts
- retracting facts
  - remove clauses explicitly: complex to implement
  - push / pop: stack like activation, no sharing of learned facts
  - MiniSAT assumptions [EénSörensson'03]
- assumptions
  - unit assumptions: assumed for the next SAT call
  - easy to implement: force SAT solver to decide on assumptions first
  - shares learned clauses across SAT calls
- IPASIR: Reentrant Incremental SAT API
  - used in the SAT competition / race since 2015 [BalyoBiereIserSinz'16]

IPASIR Model



IPASIR Functions

```
const char * ipasir_signature ();

void * ipasir_init ();

void ipasir_release (void * solver);

void ipasir_add (void * solver, int lit_or_zero);

void ipasir_assume (void * solver, int lit);

int ipasir_solve (void * solver);

int ipasir_val (void * solver, int lit);

int ipasir_failed (void * solver, int lit);

void ipasir_set_terminate (void * solver, void * state,
                           int (*terminate)(void * state));
```

```c
#include "ipasir.h"
#include <assert.h>
#include <stdio.h>
#define ADD(LIT) ipasir_add (solver, LIT)
#define PRINT(LIT) \
  printf (ipasir_val (solver, LIT) < 0 ?  " -" #LIT : " " #LIT)
int main () {
  void * solver = ipasir_init ();
  enum { tie = 1, shirt = 2 };
  ADD (-tie); ADD ( shirt); ADD (0);
  ADD ( tie); ADD ( shirt); ADD (0);
  ADD (-tie); ADD (-shirt); ADD (0);
  int res = ipasir_solve (solver);
  assert (res == 10);
  printf ("satisfiable:"); PRINT (shirt); PRINT (tie); printf ("\n");
  printf ("assuming now: tie shirt\n");
  ipasir_assume (solver, tie); ipasir_assume (solver, shirt);
  res = ipasir_solve (solver);
  assert (res == 20);
  printf ("unsatisfiable, failed:");
  if (ipasir_failed (solver, tie)) printf (" tie");
  if (ipasir_failed (solver, shirt)) printf (" shirt");
  printf ("\n");
  ipasir_release (solver);
  return res;
}
```

DP / DPLL

- dates back to the 50'ies:

  1st version DP is resolution based

  2nd version D(P)LL splits space for time

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

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

- recent ($\leq$ 25 years) optimizations:

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

  (we will have a look at some of these)

DP Procedure

forever

if $F = \top$ **return** <u>satisfiable</u>

if $\bot \in F$ **return** <u>unsatisfiable</u>

pick remaining variable x

add all resolvents on x

remove all clauses with x and $\neg x$

Bounded Variable Elimination

[EénBiere-SAT'05]

Replace $(\bar{x} \vee a)_1$ $(\bar{x} \vee c)_4$ $(\bar{x} \vee b)_2$ $(x \vee d)_5$ $(x \vee \bar{a} \vee \bar{b})_3$ by $\cancel{(a \vee \bar{a} \vee \bar{b})}_{13}$ $(a \vee d)_{15}$ $(c \vee d)_{45}$ $\cancel{(b \vee \bar{a} \vee \bar{b})}_{23}$ $(b \vee d)_{25}$ $(c \vee \bar{a} \vee \bar{b})_{34}$

- number of clauses not increasing
- strengthen and remove subsumbed clauses too
- most important and most effective preprocessing 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

D(P)LL 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

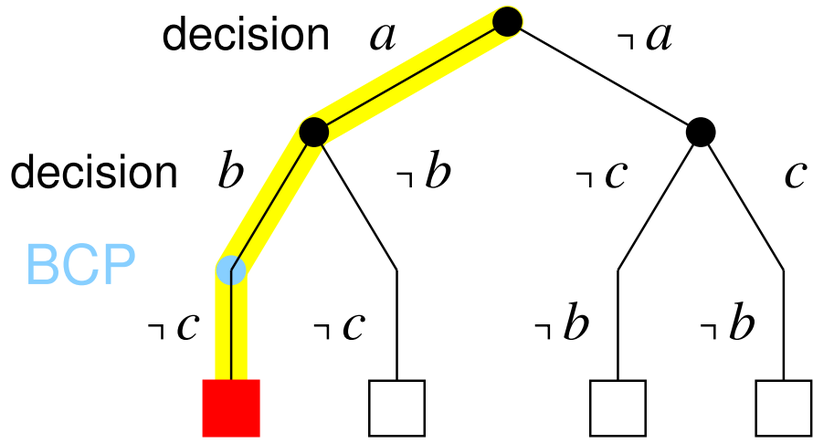
return $DPLL(F \wedge \{\neg l\})$

DPLL Example

$a = 1$

$b = 1$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

Lookahead solvers are based on this with:

- smart heuristics to pick variable to branch on
- processing of instance after every branch

Conflict Driven Clause Learning (CDCL)

[MarqueSilvaSakallah'96]

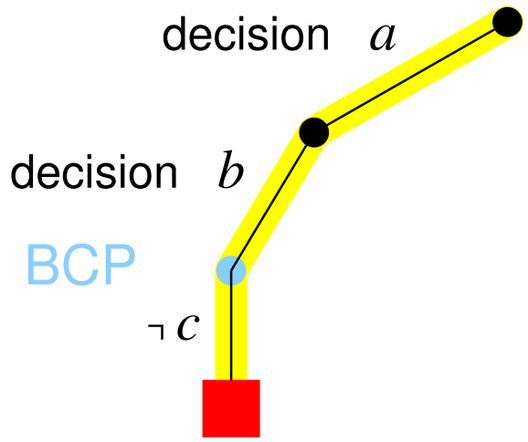
- first implemented in the context of GRASP SAT solver
  - name given later to distinguish it from DPLL
  - not recursive anymore
- essential for SMT
- learning clauses as no-goods
- notion of implication graph
- (first) unique implication points

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 1$

$c = 0$



clauses

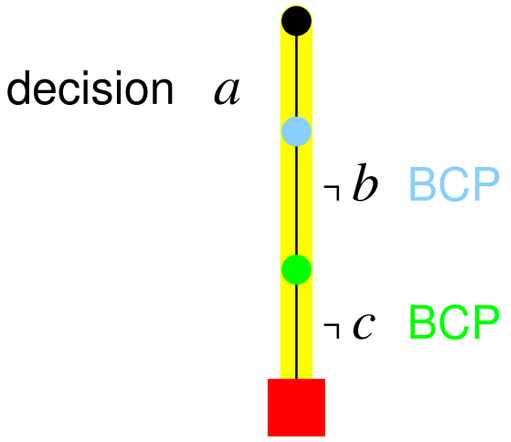
	$\neg a \vee \neg b \vee \neg c$
	$\neg a \vee \neg b \vee c$
	$\neg a \vee b \vee \neg c$
	$\neg a \vee b \vee c$
	$a \vee \neg b \vee \neg c$
	$a \vee \neg b \vee c$
	$a \vee b \vee \neg c$
	$a \vee b \vee c$
learn	$\neg a \vee \neg b$

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

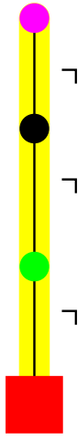
learn $\neg a$

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



$\neg a$ BCP

$\neg c$ decision

$\neg b$ BCP

clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

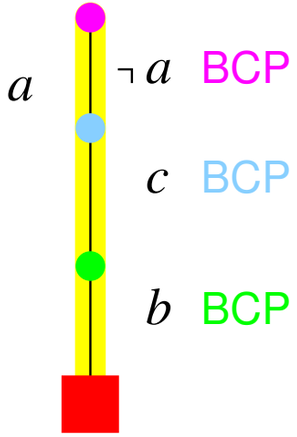
learn c

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

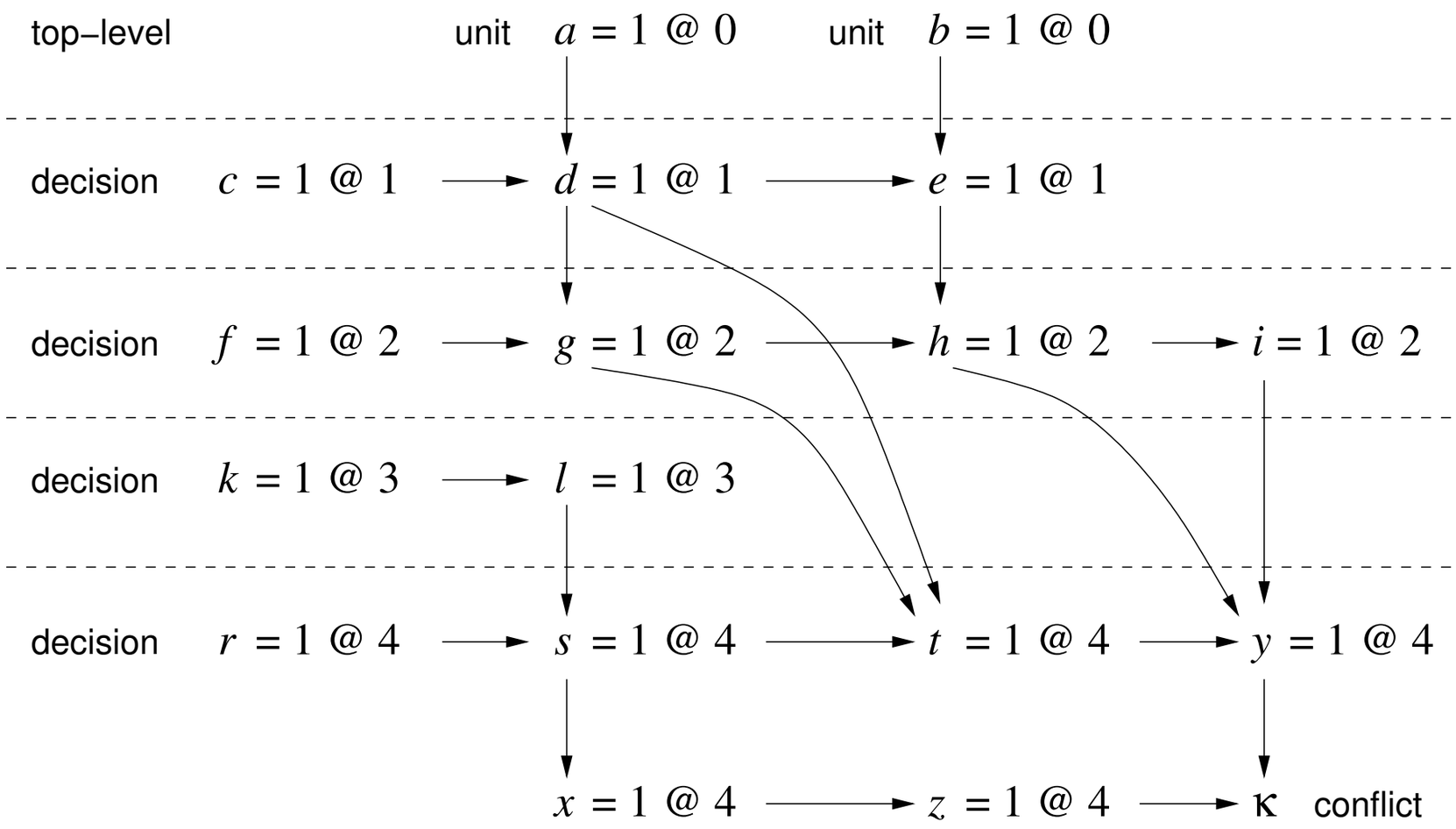
$\neg a$

c

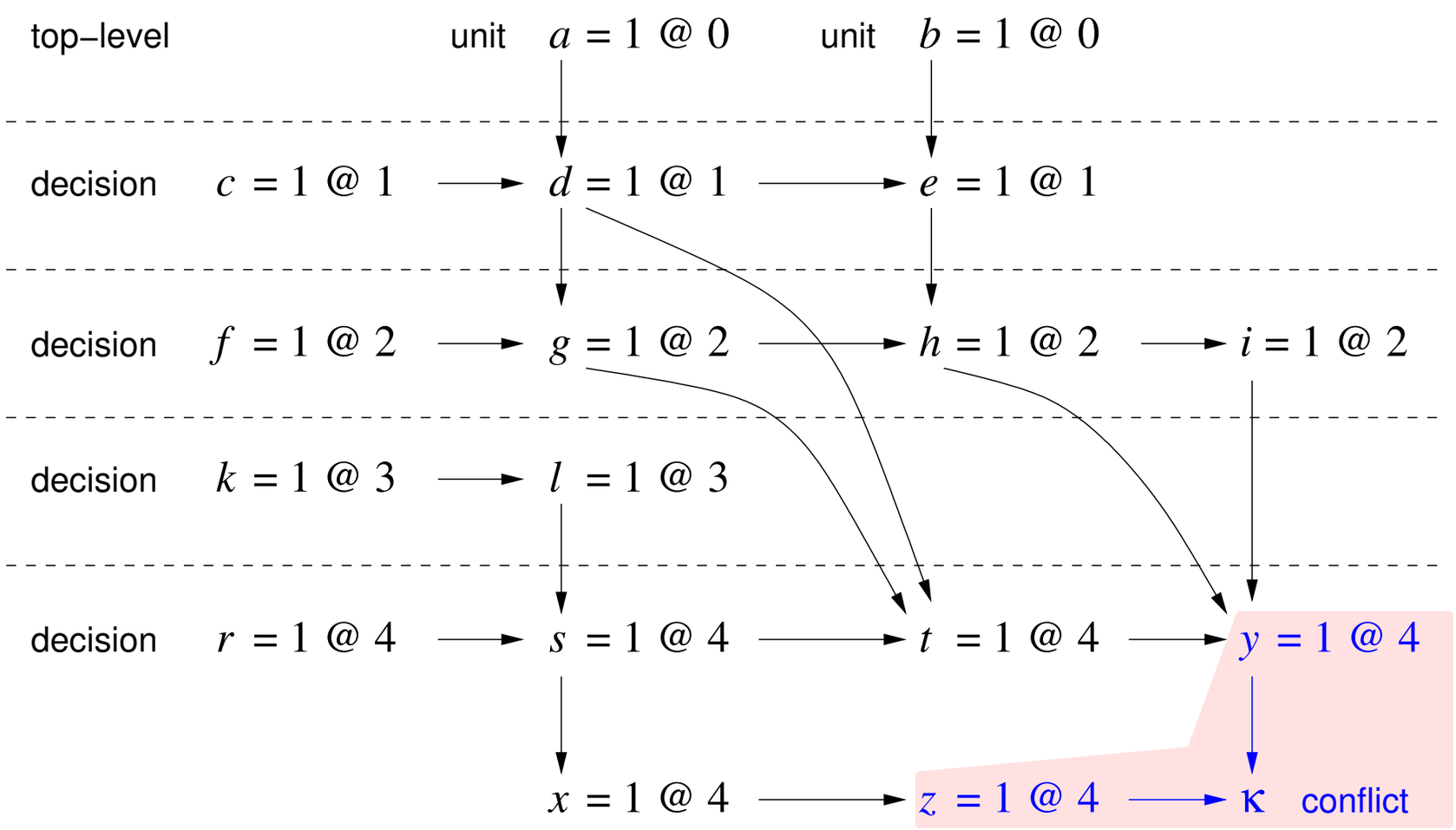
learn $\bot$

empty clause

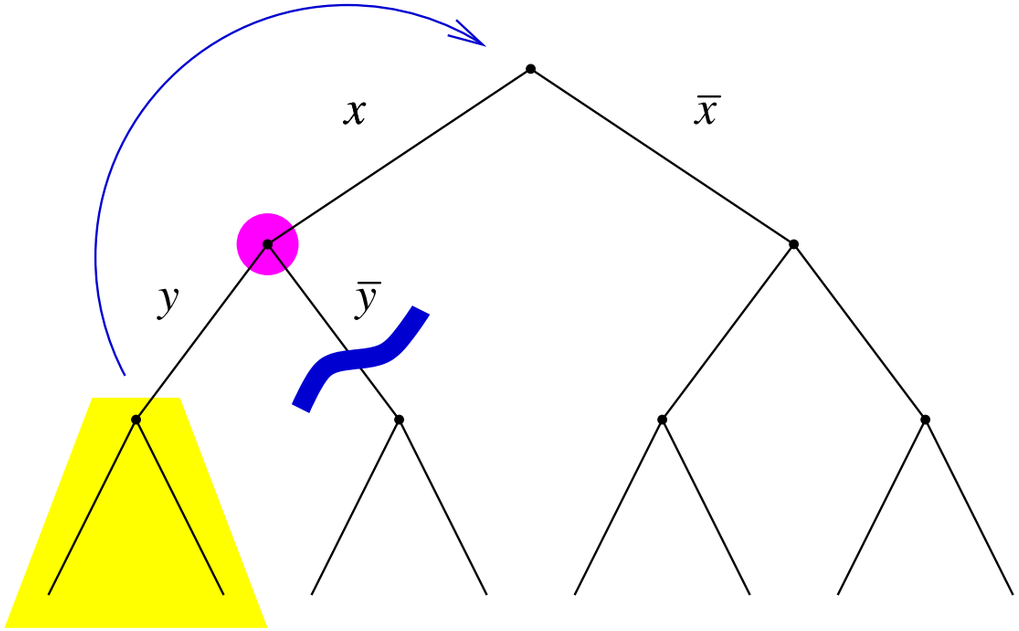
Implication Graph



Conflict


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

Backjumping



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

Immediately <u>jump back</u> to the $\overline{x}$ case – assuming x was used.

Decision Heuristics

- number of variable occurrences in (remaining unsatisfied) clauses (LIS)
  - eagerly satisfy many clauses with many variations studied in the 90ies
  - actually expensive to compute
- dynamic heuristics
  - **focus on variables which were useful recently in deriving learned clauses**
  - can be interpreted as reinforcement learning
  - started with the VSIDS heuristic [MoskewiczMadiganZhaoZhangMalik'01]
  - most solvers rely on the exponential variant in MiniSAT (EVSIDS)
  - recently showed VMTF as effective as VSIDS [BiereFröhlich-SAT'15] survey
- look-ahead
  - spent more time in selecting good variables (and simplification)
  - related to our Cube & Conquer paper [HeuleKullmanWieringaBiere-HVC'11]
  - "The Science of Brute Force" [Heule & Kullman CACM August 2017]
- EVSIDS during stabilization VMTF otherwise [Biere-SAT-Race-2019]

Exponential VSIDS (EVSIDS)

Chaff [MoskewiczMadiganZhaoZhangMalik'01]

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

MiniSAT uses EVSIDS [EénSörensson'03]

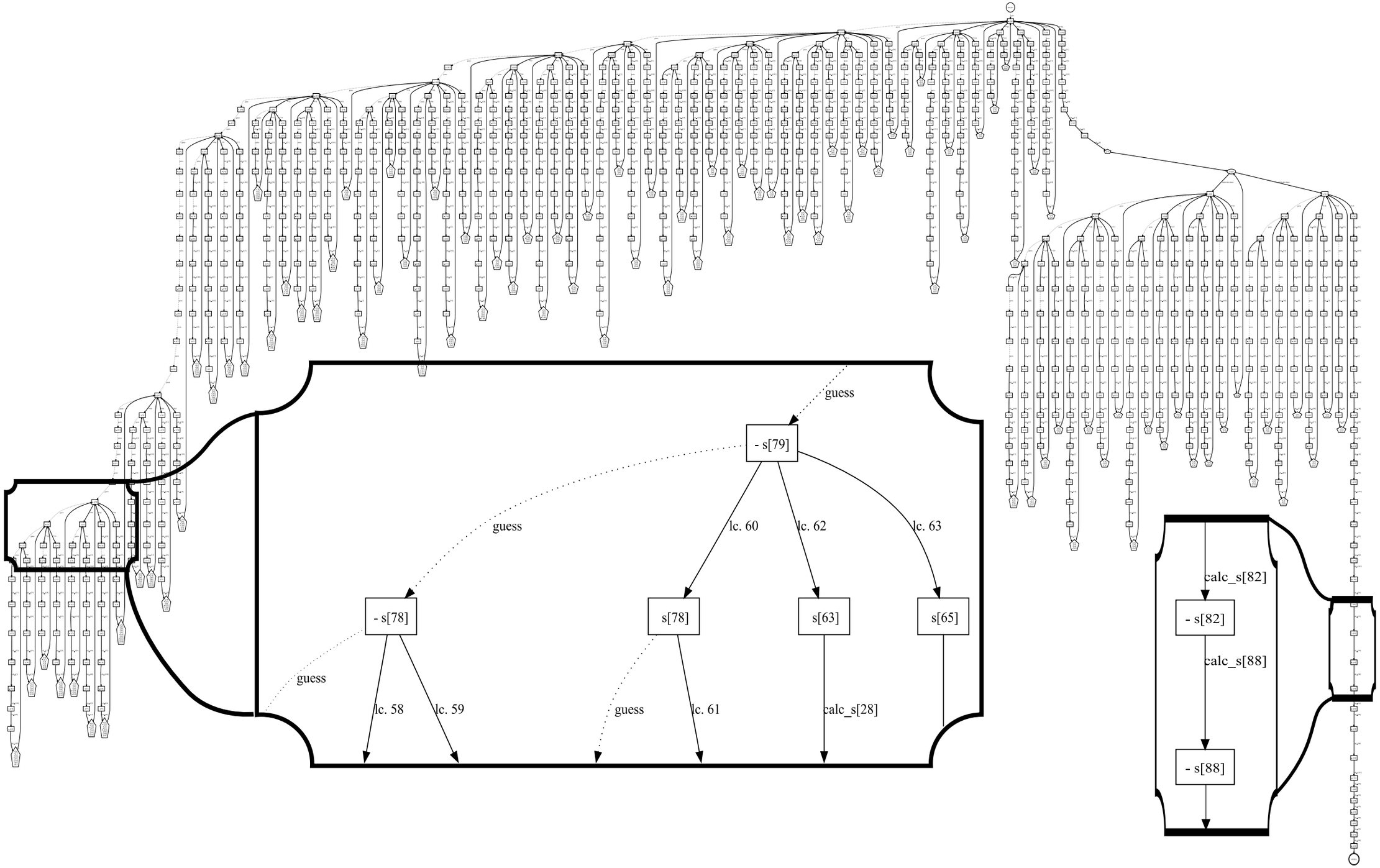
- update score of involved variables — as actually LIS would also do
- dynamically adjust increment: $\delta' = \delta \cdot \frac{1}{f}$ — typically increment δ by 5%
- use floating point representation of score
- "rescore" to avoid overflow in regular intervals
- EVSIDS linearly related to NVSIDS

Basic CDCL Loop

```
int basic_cdcl_loop () {
  int res = 0;

  while (!res)
        if (unsat) res = 20;
    else if (!propagate ()) analyze ();     // analyze propagated conflict
    else if (satisfied ()) res = 10;        // all variables satisfied
    else decide ();                         // otherwise pick next decision

  return res;
}
```


guess
- s[79]
guess
lc. 60
lc. 62
lc. 63
- s[78]
s[78]
s[63]
s[65]
guess
lc. 58
lc. 59
guess
lc. 61
calc_s[28]
calc_s[82]
- s[82]
calc_s[88]
- s[88]

Reducing Learned Clauses

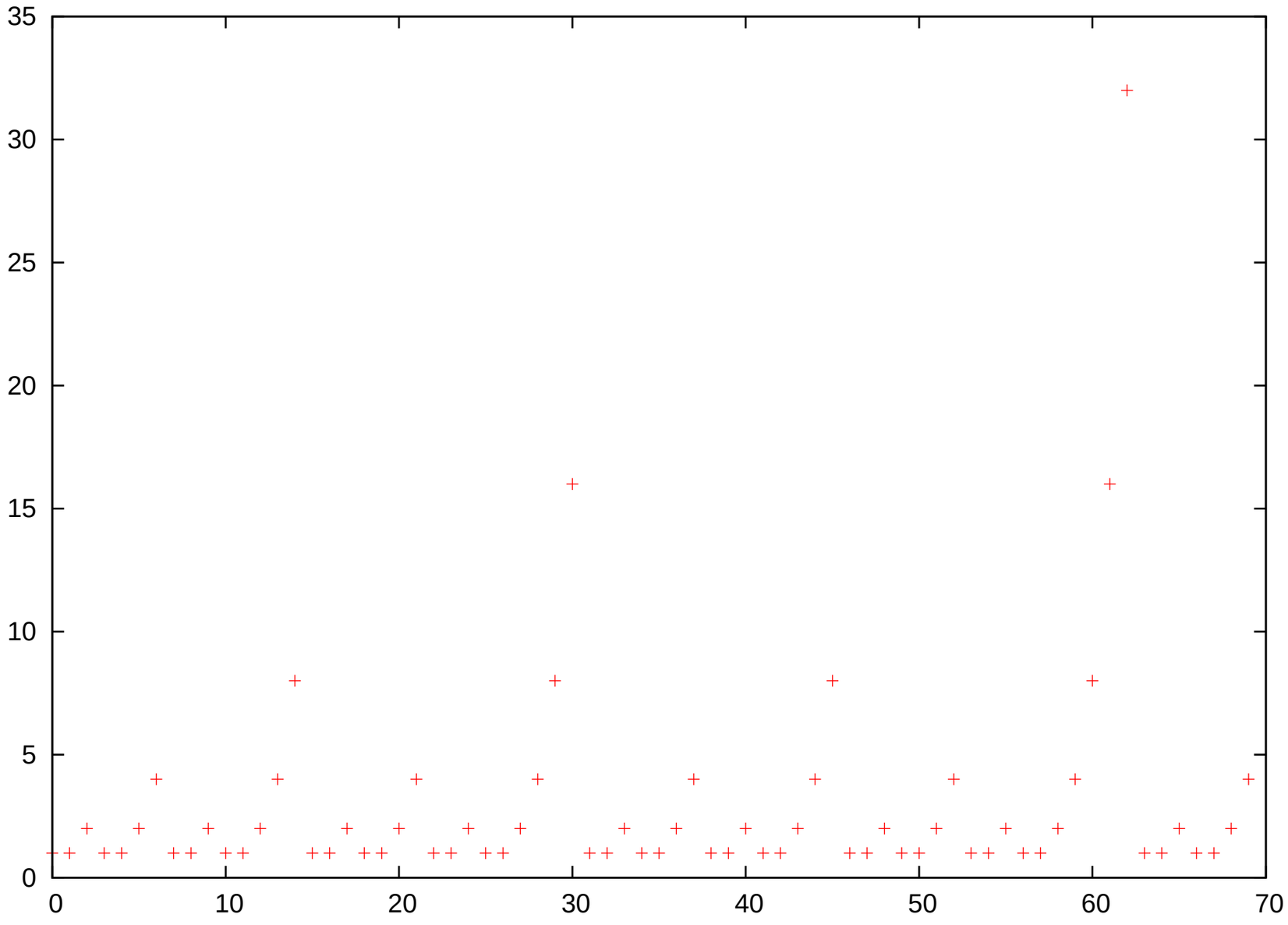
- keeping all learned clauses slows down BCP — kind of quadratically
  - so SATO and RelSAT just kept only “short” clauses
- better periodically delete “useless” learned clauses
  - keep a certain number of learned clauses — “search cache”
  - if this number is reached MiniSAT reduces (deletes) half of the clauses
  - then maximum number kept learned clauses is increased $\boxed{\text{geometrically}}$
- LBD (glucose level / glue) prediction for usefulness — [AudemardSimon-IJCAI’09]
  - LBD = number of decision-levels in the learned clause
  - allows $\boxed{\text{arithmetic}}$ increase of number of kept learned clauses
  - keep clauses with small LBD forever ($\leq 2 \ldots 5$)
  - three Tier system by [Chanseok Oh]
- recent work on machine-learning heuristic based on labelled proof data — [SoosKulkarniMeel2019]

Restarts

- 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]

Luby's Restart Intervals

70 restarts in 104448 conflicts



Phase Saving and Rapid Restarts

- phase assignment:
  - assign decision variable to 0 or 1?
  - lucky guess can lead to immediate solution to a satisfiable instance
- “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]

CDCL Loop with Reduce and Restart

```
int basic_cdcl_loop_with_reduce_and_restart () {

  int res = 0;

  while (!res)
        if (unsat) res = 20;
    else if (!propagate ()) analyze ();     // analyze propagated conflict
    else if (satisfied ()) res = 10;        // all variables satisfied
    else if (restarting ()) restart ();     // restart by backtracking
    else if (reducing ()) reduce ();        // collect useless learned clauses
    else decide ();                         // otherwise pick next decision

  return res;
}
```

Code from the SAT Solver CaDiCaL by Armin Biere

```
int Internal::cdcl_loop_with_inprocessing () {

  int res = 0;

  while (!res) {
        if (unsat) res = 20;
    else if (!propagate () analyze ();      // propagate and analyze
    else if (iterating) iterate ();         // report learned unit
    else if (satisfied () res = 10;         // found model
    else if (terminating () break;          // limit hit or async abort
    else if (restarting () restart ();      // restart by backtracking
    else if (rephasing () rephase ();       // reset variable phases
    else if (reducing () reduce ();         // collect useless clauses
    else if (probing () probe ();           // failed literal probing
    else if (subsuming () subsume ();       // subsumption algorithm
    else if (eliminating () elim ();        // variable elimination
    else if (compacting () compact ();      // collect variables
    else if (conditioning () condition ();  // globally blocked clauses
    else res = decide ();                   // next decision
  }

  return res;
}
```

https://fmv.jku.at/cadical

Two-Watched Literal Schemes

- original idea from SATO [ZhangStickel'00]
  - invariant: always watch two non-false literals
  - if a watched literal becomes false replace it
  - if no replacement can be found clause is either unit or empty
  - original version used head and tail pointers on Tries
- improved variant from Chaff [MoskewiczMadiganZhaoZhangMalik'01]
  - watch pointers can move arbitrarily SATO: head forward, tail backward
  - no update needed during backtracking
- one watch is enough to ensure correctness but looses arc consistency
- reduces visiting clauses by 10x
  - particularly useful for large and many learned clauses
- blocking literals [ChuHarwoodStuckey'09]
- special treatment of short clauses (binary [PilarskiHu'02] or ternary [Ryan'04])
- cache start of search for replacement [Gent-JAIR'13]

Parallel SAT

- Application level parallelism
- Guiding path principle
- Portfolio (with or without sharing)
- Concurrent cube & conquer

Proofs

SAT solvers are search-directed proof systems.
They only incidentally find satisfying assignments.

When and why are they important?

- If solution is UNSAT then proofs are super-important
  - Determines minimum number of resolutions
  - SAT solver cannot finish in less than that many steps
  - If it's exponential in input size, we are in a mess ☹
- If solution is SAT then maybe not so important?
  - Observe: pruning solution space is done through resolvents
  - We are building a proof that certain parts of the search space are devoid of solutions
  - Experimentally easy to validate: give XOR matrix with a solution to a SAT solver ¯_(ツ)_/¯

Hence, the proof we are generating is very important.

Proofs: Example proof

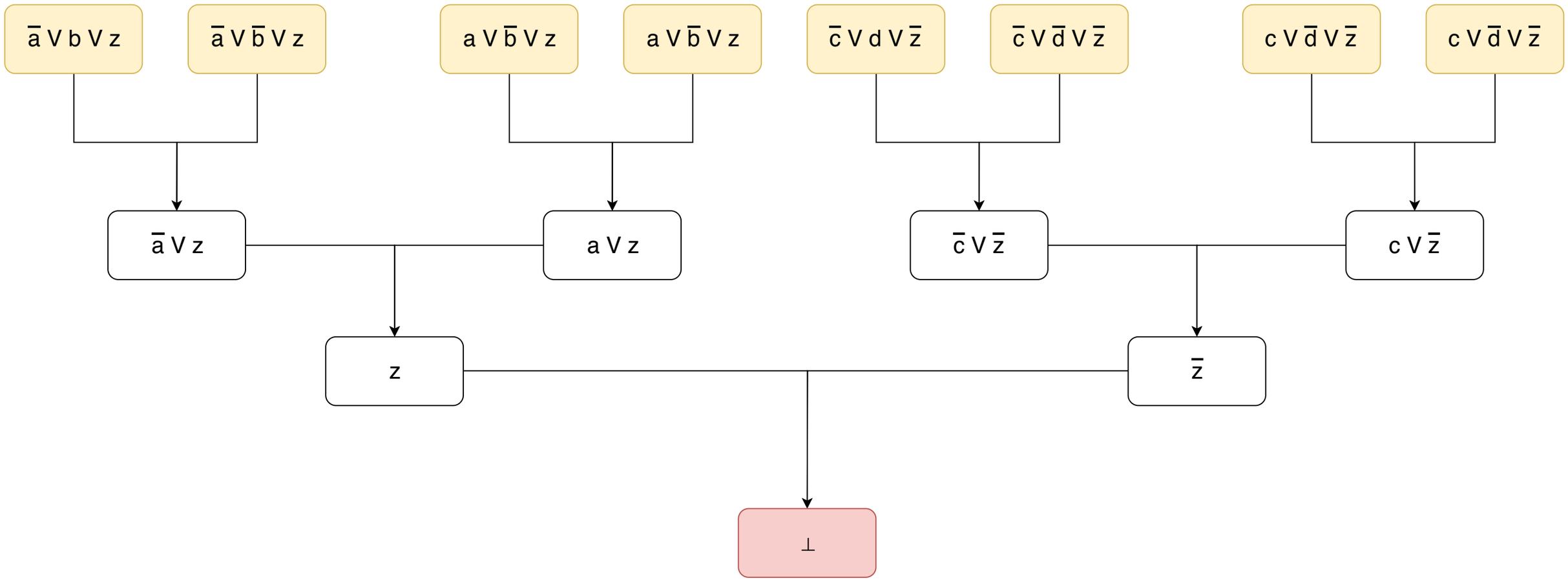
Say we want to prove that the following set of clauses is UNSAT:

$$\begin{array}{lcl} \overline{a} \vee \overline{b} \vee z & \wedge & \overline{c} \vee \overline{d} \vee \overline{z} \quad \wedge \\ a \vee b \vee z & \wedge & c \vee d \vee \overline{z} \quad \wedge \\ \overline{a} \vee b \vee z & \wedge & \overline{c} \vee d \vee \overline{z} \quad \wedge \\ a \vee \overline{b} \vee z & \wedge & c \vee \overline{d} \vee \overline{z} \end{array}$$

$$\begin{array}{c} \overline{a} \vee \overline{b} \vee z \\ \odot \\ \overline{a} \vee b \vee z \\ \\ a \vee \overline{b} \vee z \\ \odot \\ a \vee b \vee z \end{array} \quad \leftrightarrow \quad \begin{array}{c} \overline{a} \vee z \\ \\ \odot \\ \\ a \vee z \end{array} \quad \leftrightarrow \quad z$$

Observe: we could have used $b \vee z$ and $\overline{b} \vee z$, too!

Proofs: Example proof cont.



Homework: how many different resolution trees are there for deriving $\perp$ here?
(How many ways to derive z? And $\bar{z}$?)

Proofs: Some observations

- In general there are many different proofs
- Proof forms a DAG
- Proof is acyclic but not necessarily tree-like
- Different proofs can be very different in size
- Input set of clauses to the proof called the “core” of the CNF
- Often many different cores, too (like above)
- Cores are <u>useful</u>: For example, can tell us why we cannot schedule a tournament
  - we must relax some of the constraints indicated by the core clauses
  - but there might be more than one core, so may need to relax more than one!
- Pigeonhole principle [Hak85] formulas’ proofs are lower bound exponential in size ☹
  - We can (and should) explore stronger reasoning methods
  - One way is to do CDCL(T), where T are the new theories

RUP / DRUP

- original idea for proofs: proof traces / sequence consisting of “learned clauses”
- can be checked clause by clause through unit propagation
- reverse unit implied clauses (RUP) [GoldbergNovikov’03] [VanGelder’12]
- deletion information (DRUP): trace of added and deleted clauses [HeuleHuntWetzler-FMCAD’13/STVR’14]
- RUP in SAT competition 2007, 2009, 2011, DRUP since 2013 to certify UNSAT

Blocked Clauses

[Kullman-DAM’99] [JärvisaloHeuleBiere-JAR’12]

- clause $\overbrace{(a \vee l)}^{C}$ “blocked” on l w.r.t. CNF $\overbrace{(\bar{a} \vee b) \wedge (l \vee c) \wedge \underbrace{(\bar{l} \vee \bar{a})}_{D}}^{F}$
  - all resolvents of C on l with clauses D in F are tautological
- blocked clauses are “redundant” too
  - adding or removing blocked clauses does not change satisfiability status
  - however it might change the set of models

Resolution Asymmetric Tautologies (RAT)

"Inprocessing Rules" [JärvisaloHeuleBiere-IJCAR'12]

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

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

  - 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

Gauss-Jordan Elimination

Gaussian part, getting upper-triangular matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Jordan part, getting row-echelon form:

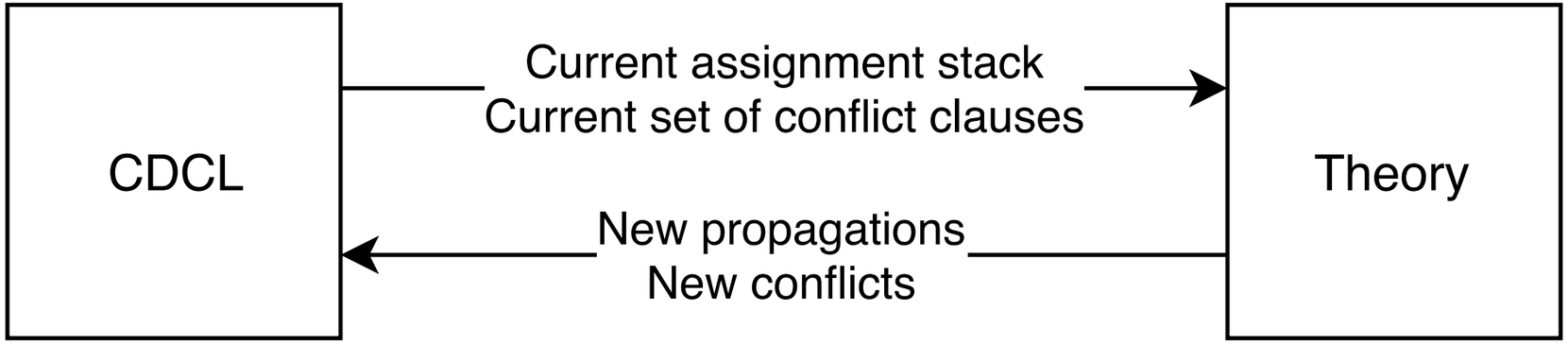
$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

- The naive implementation above is $O(n^3)$ steps
- More sophisticated versions take around $O(n^{2.8})$ steps
- If resolution operator is all we have, shortest proof is exponential in n

CDCL(T)

For theories that are not efficiently simulated by CDCL

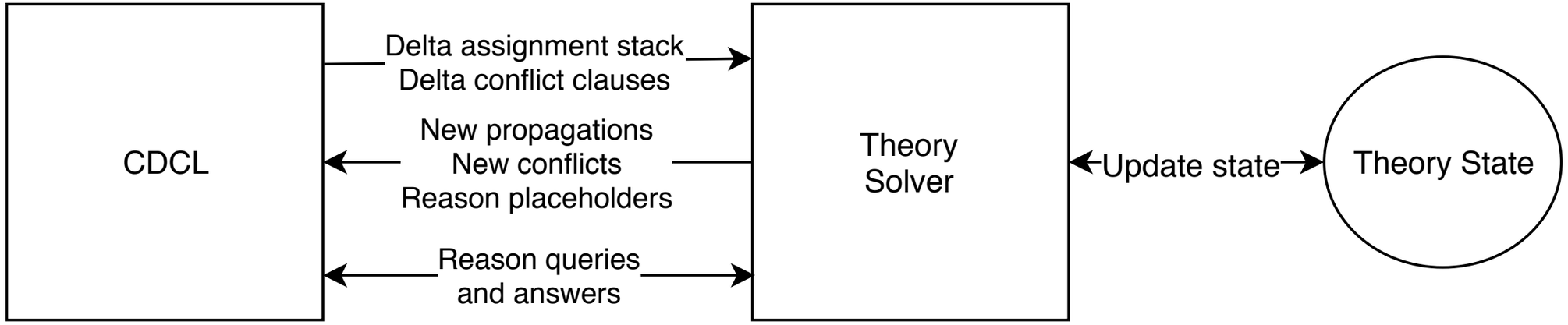
- T is the theory, e.g.:
  - Gauss-Jordan Elimination [SoosNohlCastelluccia'2010]
  - Pseudo-Boolean Reasoning [ChaiKuehlmann'2006]
  - Symmetric Explanation Learning [DevriendtBogaertsBruynooghe'2017]
- Theory is run side-by-side to the CDCL algorithm
- **Propagate** values implied by Theory given current assignment stack of CDCL
- **Conflict** if Theory implies 1=0 given current assignment stack of CDCL
- Theory must give reason for propagations&conflicts



CDCL(T) Cont.

Optimizations:

- Should only send delta of assignment stack + conflict clauses
  - Variables assigned (decisions + propagations)
  - Variables unassigned (backtracking, restarting)
  - New conflict clauses
- Theory only needs to compute delta relative to old state
- Theory can give placeholders for reasons
  - If reason is needed during conflict generation, Theory is queried
  - Called “lazy” (vs “greedy”) interpolant generation



CDCL(T) Gauss-Jordan Elimination: Ingredients

What components do we need?

- **Extractor for XOR constraints**: XORs may be encoded as CNF
- **Disjoint matrix detection**: disjoint matrices should be handled separately
- **Delta update mechanism** for row-echelon form matrix:
  - how to handle when variable is set
  - how to handle when variable is unset
- **Efficient data structures** to allow for quick updates
- **Reason generation**

CDCL(T) Gauss-Jordan Elimination: Extraction

$$
\begin{aligned}
l_1 \oplus l_2 \oplus l_3 = 1 \quad \Leftrightarrow \quad & l_1 \vee l_2 \vee l_3 \wedge \\
& \bar{l}_1 \vee \bar{l}_2 \vee l_3 \wedge \\
& \bar{l}_1 \vee l_2 \vee \bar{l}_3 \wedge \\
& l_1 \vee \bar{l}_2 \vee \bar{l}_3 \wedge
\end{aligned}
$$

$$
\begin{aligned}
l_1 \oplus l_2 \oplus l_3 = 1 \quad \leftarrow \quad & l_1 \vee l_2 \vee \wedge \\
& \bar{l}_1 \vee \bar{l}_2 \vee l_3 \wedge \\
& \bar{l}_1 \vee l_2 \vee \bar{l}_3 \wedge \\
& l_1 \vee \bar{l}_2 \vee \bar{l}_3 \wedge
\end{aligned}
$$

- Missing literals only mean something stronger than XOR
- XOR is still implied and should be detected

CDCL(T) Gauss-Jordan Elimination: Extraction

Algorithm 1 ComputeBloom

```
abst ← 0
for var in clause do
    abst ← abst | (1 << (var % 32))
return abst
```

Algorithm 2 Barbet(clauses, M)

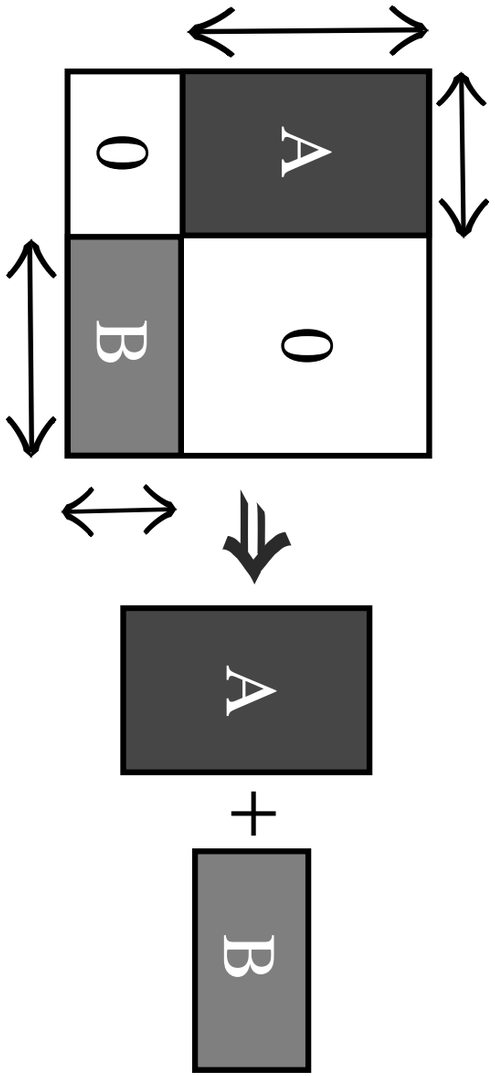
```
xorclauses ← ∅
for base_cl ∈ clauses do
    if base_cl.size > M then continue
    if base_cl.used == 1 then continue
    FIND_ONE_XOR(base_cl)
return xorclauses
```

CDCL(T) Gauss-Jordan Elimination: Extraction

```
function FindOneXOR(base_cl)
    quickcheck ← array of zeroes
    found_comb ← array of zeroes
    comb ← 0
    base_rhs ← 1                                        ▷ right-hand-side of the XOR
    for i ← 0 . . . base_cl size-1 do
        base_rhs ← base_rhs ⊕ base_cl[i].sign
        comb ← comb | (base_cl[i].sign << i)
        quickcheck[base_cl[i].var] ← 1
    base_abst ← CALC_ABST(base_cl)
    found_comb[comb] ← 1
    for v ∈ Vars(base_cl) do
        for abst, cl ∈ occurrence[v] do
            if CheckClause(abst, cl, base_cl, base_abst) then return
```

CDCL(T) Gauss-Jordan Elimination: Matrix Separation

```
function FINDMATRIXES(xors)
    matrixnum ← 0, var-to-matrix ← -1, matrix-to-vars ← empty
    for xor ∈ xors do
        xor-belongs ← -1
        for var ∈ xor do
            if var-to-matrix[var] != -1 then
                if xor-belongs == -1 then xor-belongs = var-to-matrix[var]
                else if xor-belongs != var-to-matrix[var] then
                    Move all variables from var-to-matrix[var] to xor-belongs
        if xor-belongs == -1 then
            xor-belongs ← matrixnum++
        for var ∈ xor do
            var-to-matrix[var] = xor-belongs
```



CDCL(T) Gauss-Jordan Elimination: None of that row swapping please!

Observations:

- We are using binary matrixes (1/0), so bit-packed format is best
- Packed format: row-swapping becomes expensive – it's a copy
- Row-echelon form is nice for the eyes [HanJiang2012]:
  - But we only need a row to be responsible for a column's "1"
  - What we loose: have to check all rows, not only ones below
- So, any row can be responsible for being a column's "1"

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

CDCL(T) Gauss-Jordan Elimination: 2-variable watchlist scheme

Let's use a 2-variable watch scheme [HanJiang2012]:

- If 2 or more variables are unset in XOR constraint, it cannot propagate or conflict
- If 1 variable is unset, it must propagate
- If 0 variable is unset, it is either satisfied or is in conflict

We'll use the Simplex Method's terminology:

- Let's call the column that the row is responsible for "basic"
- Let's call the column that the row is NOT responsible for "nonbasic"

What data structures do we need for this? Let's see:

- Watchlist for variables (not literals!)
- column-has-responsible-row[column] = 1/0
- row-to-nonbasic-column[row] = column

CDCL(T) Gauss-Jordan Elimination: Propagation

A rough outline:

- Observe that the matrix is usually underdetermined: more columns than rows
- Many unset columns will have no responsible rows
- If we set a variable, its column doesn't need a responsible row
- The more variables we decide on, the more the matrix will be determined

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Let's set the first column to "1" $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

we get a propagation! $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Notice: we were were watching both of this row's variables where it has a "1". It's a 2-variable watch scheme!

CDCL(T) Gauss-Jordan Elimination: Propagation

We got a propagation from last slide:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable is now set by Gauss-Jordan $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable is decided on $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Need new responsible variable $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Must adjust matrix $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

New propagation $\rightarrow$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

And the story goes on...

CDCL(T) Gauss-Jordan Elimination: Reason Clauses

What combination of XOR constraints gave us the propagation?

- The above set of matrixes cannot give us the reason clause
- Easy solution: the “green” columns are actually not zeroed out
- When looking for propagations/conflicts, we check if columns’ variable is set. If yes, we pretend it’s a 0
- When looking for reasons, we use the actual values
- All the row-XOR operations happen as before

Hence:

- Each row is a combination of input XOR constraints
- It is guaranteed to propagate/conflict under current variable assignment

When a variable is set, we are just wearing “green glasses”

CDCL(T) Gauss-Jordan Elimination: Backtracking

If we don't zero out the columns, we get a free bonus! If we need to unset an assignment due to backtracking, we pretend we never set it (remove "green glasses"):

- All previous invariants still hold
- If the column had a responsible row, it still has it
- Both watches of the row are still good and in the watchlists
- Matrix looks differently than when we last had this assignment... is that a problem?
- No! Observe: new matrix could have been reached from the starting position, pivoting differently(!)

CDCL(T) Gauss-Jordan Elimination: Recap

Let's recap! What was hard:

- Extracting XOR constraints
- Keeping CDCL and GJ in sync:
  - Fast update for variable setting (propagation)
  - Fast update for backtracking (conflict)
- Reason clause generation

Symmetries: teaser

- Let's put 10 birds into 10 holes, 1 bird per hole: pigeonhole principle
- Let's schedule 10 teams to 5 stadiums over 200 days
- Symmetries are often non-trivially encoded into the CNF
- Sometimes, encoding them differently can get rid of them, but sometimes it's hard

Symmetries: preliminaries

- For a given formula φ, an assignment of the variables of φ is a function $\alpha : \mathcal{V} \to \{1,0\}$
- **Permutation** is a bijection from a set to itself
- Cycle notation of a permutation: $(abc)(de)$ maps a to b, b to c, c to a, swaps d with e, and maps all other elements to themselves
- Permutations form algebraic groups under the composition relation ($\odot$)
- Group of permutations of $\mathcal{V}$ (i.e. bijections from $\mathcal{V}$ to $\mathcal{V}$) is noted $\mathfrak{G}(V)$
- Group $\mathfrak{G}(V)$ acts on the set of literals. For $g \in \mathfrak{G}(V)$ and a literal $l \in \mathcal{L}$
  - $g.l = g(l)$ if l is a positive literal
  - $g.l = \overline{g(\bar{l})}$ if l is a negative literal
- Group $\mathfrak{G}(V)$ also acts on (partial) assignments of $\mathcal{V}$: for $g \in \mathfrak{G}(V), \alpha \in Ass(\mathcal{V}), g.\alpha = \{g.l | l \in \alpha\}$
- Let φ be a formula, and $g \in \mathfrak{G}(V)$. We say that $g \in \mathfrak{G}(V)$ is a **symmetry** of φ if for every complete assignment $\alpha, \alpha \models \varphi$ if and only if $g.\alpha \models \varphi$

Symmetries: Example permutation

All of this did not click until I found the work of Devriend, Bogaerts, Bruynooghe and Denecker, **BreakID**:

```
$ cat mycnf.cnf
p cnf 4 4
1 2 3 0
1 -2 3 0
-1 4 0
-3 4 0
```

```
$ ./breakid mycnf.cnf
*** Detecting symmetry group...
-- Permutations:
( 2 -2 )
( 1 3 ) ( -1 -3 )          [<-- "cycle notation"]
```

Makes sense:

- If we substitute 1 with 3 everywhere and vica versa, it's the same!
- If we substitute 2 with -2 everywhere and vica versa, it's the same!

Symmetries: examples

```
$ cat c.cnf
p cnf 6 7
1 -2 3 0
1 2 3 0
-1 4 0
-3 4 0
c ---------------
5 -2 6 0
5 4 0
6 4 0
```

```
$ ./breakid mycnf.cnf
*** Detecting symmetry group...
-- Permutations:
( 1 3 ) ( -1 -3 )
( 5 6 ) ( -5 -6 )
```

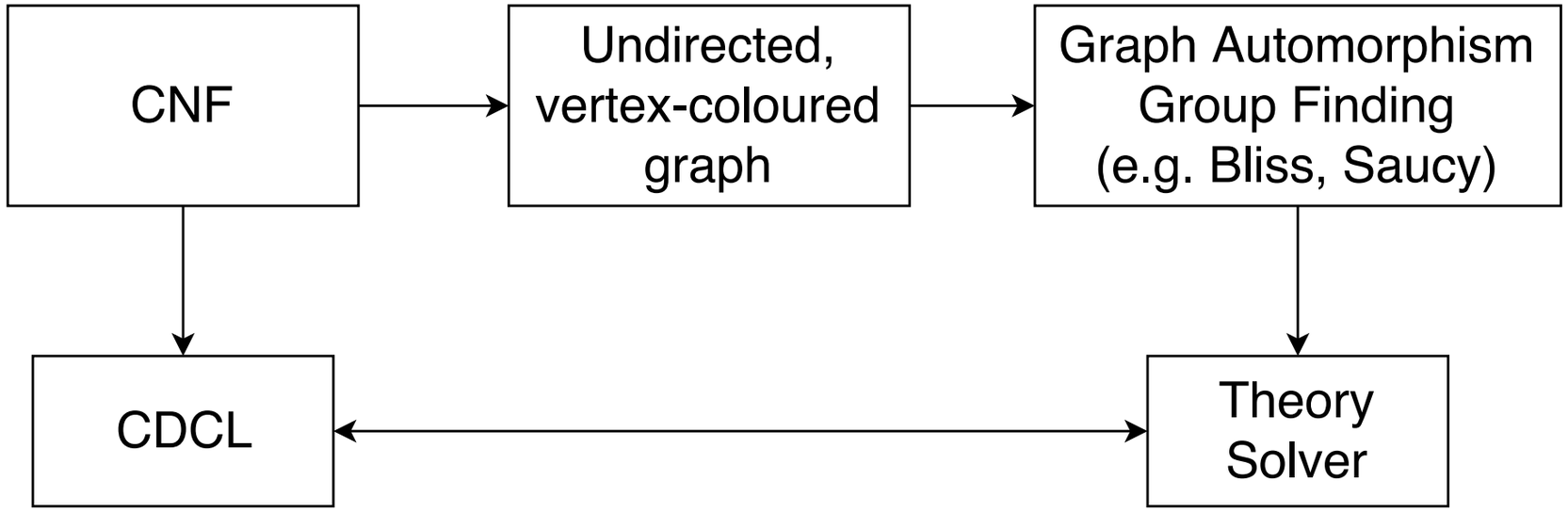
Makes sense:

- I can no longer substitute 2 for -2 and vica-versa, it won't be the same CNF
- **Any** combination of $1 \leftrightarrow 3$ and $5 \leftrightarrow 6$ works. Hence these permutations can be combined.

Symmetries: obtaining them

Let's create an undirected, vertex-coloured graph:

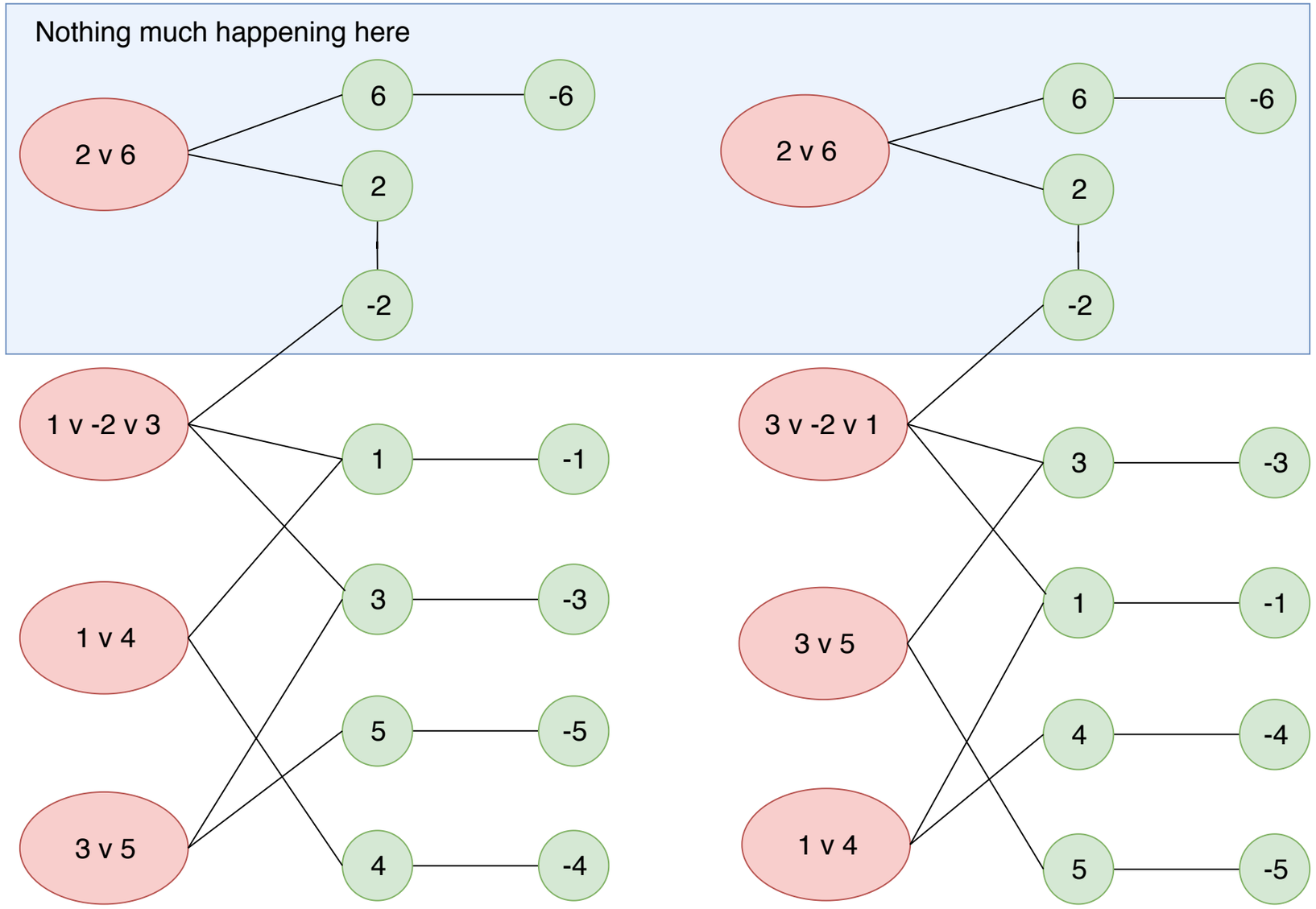
- Each literal is a vertex, colour green
- Each clause is a vertex, colour red
- Each literal is connected to its inverse
- Each clause's vertex is connected to the literals' vertices inside it
- The automorphism groups of this graph are the symmetry groups of the CNF



Symmetries: the graph

```
$ cat c.cnf
p cnf 6 4
2 6 0
1 -2 3 0
1 4 0
3 5 0
```

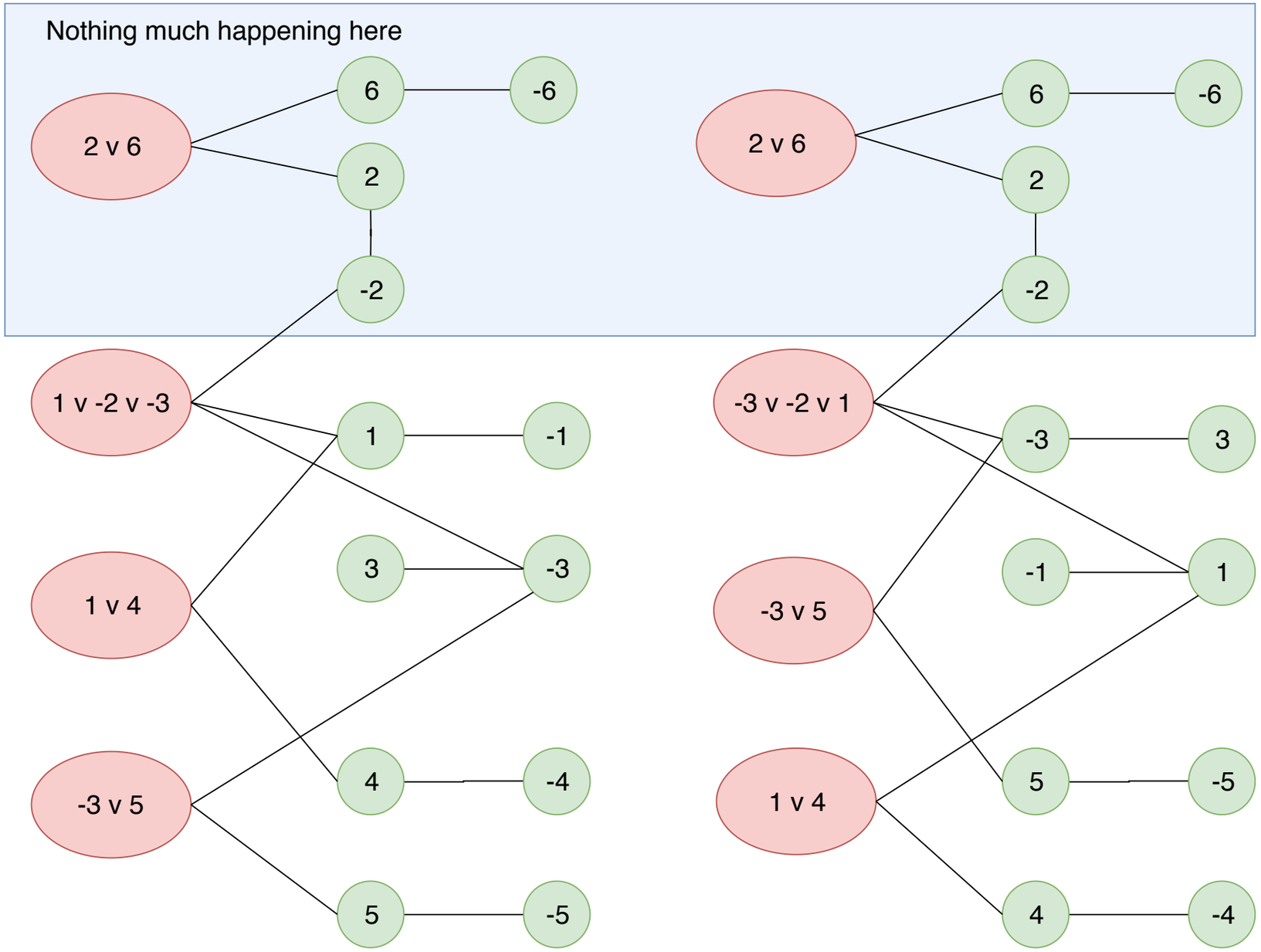
```
$ ./breakid mycnf.cnf
-- Permutations:
(1 3) (-1 -3) (4 5) (-4 -5)
```



Symmetries: the graph, example 2

```
$ cat d.cnf
p cnf 6 4
2 6 0
1 -2 -3 0
1 4 0
-3 5 0
```

```
$ ./breakid mycnf.cnf
-- Permutations:
(1 -3) (-1 3) (4 5) (-4 -5)
```



Symmetries: solutions

```
$ ./breakid mycnf.cnf
*** Detecting symmetry group...
-- Permutations:
( 1 3 ) ( -1 -3 )
( 5 6 ) ( -5 -6 )
```

OK, so how about the solutions?

- If a solution has $v_1 = 1, v_3 = 0$ we obviously have another solution: $v_1 = 0, v_3 = 1$
- If a solution has $v_5 = 1, v_6 = 0$ we obviously have another solution: $v_5 = 0, v_6 = 1$
- But do we always have 4x more solutions?
- NO! How about when the only solution has $v_1 = 0, v_3 = 0$?

Symmetries: solutions, example 2

```
$ ./breakid mycnf.cnf
-- Permutations:
(1 3) (-1 -3) (4 5) (-4 -5)
```

OK, so how about the solutions?

- If a solution has $v_1 = 1, v_3 = 0$ we obviously have another solution: $v_1 = 0, v_3 = 1$
- If a solution has $v_1 = 0, v_3 = 0, v_4 = 1, v_5 = 0$ we still have another solution: $v_1 = 0, v_3 = 0, v_4 = 0, v_5 = 1$
- But if a solution has $v_1 = 0, v_3 = 0, v_4 = 0, v_5 = 0 \rightarrow$ we can't do anything
- Similarly if a solution has $v_1 = 1, v_3 = 1, v_4 = 1, v_5 = 1 \rightarrow$ we can't do anything

Symmetries: breaking them

```
$ cat c.cnf
p cnf 6 4
2 6 0
1 -2 3 0
1 4 0
3 5 0
```

```
$ ./breakid mycnf.cnf
-- Permutations:
(1 3) (-1 -3) (4 5) (-4 -5)
```

Let's observe the following:

- If we make sure that $v_4 \geq v_5$ then we eliminate <u>some</u> of the symmetry
- But that doesn't eliminate the symmetry where $v_4 = v_5$
- For that, we need another constraint: $v_4 = v_5 \rightarrow v_1 \geq v_3$
- The above two eliminate solutions where:
  - $v_4 = 0, v_5 = 1$
  - $v_4 = 0, v_5 = 0, v_1 = 1, v_3 = 0$
  - $v_4 = 1, v_5 = 1, v_1 = 1, v_3 = 0$
- These correspond to clauses:
  - $v_4 \vee \overline{v}_5$
  - $v_7 \leftrightarrow \overline{v}_4 \vee v_5$
  - $v_7 \rightarrow \overline{v}_1 \vee v_3$
- Note that v_7 is an indicator variable. It is true when:
  - $v_4 = 0, v_5 = 0$
  - $v_4 = 1, v_5 = 1$
  - $v_4 = 0, v_5 = 1$ But this never occurs! (remember: $v_4 \geq v_5$)
  - Hence, it's only true when $v_4 = v_5$
- Is this symmetry breaking complete?

Symmetries: breaking them

```
$ cat c.cnf
p cnf 6 4
2 6 0
1 -2 3 0
1 4 0
3 5 0
```

```
$ ./breakid c.cnf -b --only-b
-- Permutations:
(1 3) (-1 -3) (4 5) (-4 -5)
c breaking clauses: 4
c aux vars: 1
-5 4 0
-7 -1 3 0
-7 -4 5 0
 7  4 0
 7 -5 0
```

Let's observe the following:

- If we make sure that $v_4 \geq v_5$ then we eliminate <u>some</u> of the symmetry
- But that doesn't eliminate the symmetry where $v_4 = v_5$
- For that, we need another constraint: $v_4 = v_5 \rightarrow v_1 \geq v_3$
- The above two eliminate solutions where:
  - $v_4 = 0, v_5 = 1$
  - $v_4 = 0, v_5 = 0, v_1 = 1, v_3 = 0$
  - $v_4 = 1, v_5 = 1, v_1 = 1, v_3 = 0$
- These correspond to clauses:
  - $v_4 \vee \overline{v}_5$
  - $v_7 \leftrightarrow \overline{v}_4 \vee v_5$
  - $v_7 \rightarrow \overline{v}_1 \vee v_3$
- Note that v_7 is an indicator variable. It is true when:
  - $v_4 = 0, v_5 = 0$
  - $v_4 = 1, v_5 = 1$
  - $v_4 = 0, v_5 = 1$ But this never occurs! (remember: $v_4 \geq v_5$)
  - Hence, it's only true when $v_4 = v_5$
- Is this symmetry breaking complete?

Symmetries: CDCL(T)

CDCL(T) systems for symmetries:

- “Static” handling through symmetry breaking clauses
  - Shatter [AloulRamanMiarkovSakallah2003]
  - BreakID [DevriendtBogaertsBruynoogheDenecker2016]
- “Dynamic” handling through dynamic symmetry breaking clauses, propagations, and conflicts:
  - Symmetric explanation learning [DevriendtBogaertsBruynooghe2017]
  - Symmetry status tracking [MetinBaarirColangeKordon2018]

Symmetries: CDCL(T) static breaking

If $\mathfrak{G}(\mathcal{V}$ is a symmetry group, then a symmetry breaking formula ψ is sound if for each assignment α there exists at least one symmetry $g \in \mathfrak{G}(\mathcal{V})$ such that $g.\alpha$ satisfies ψ. ψ is complete if for each assignment α there exists at most one symmetry $g \in \mathfrak{G}(\mathcal{V})$ such that $g.\alpha$ satisfies ψ [Walsh2012].

- It's easy to make a sound symmetry breaking formula
- It's hard to make it compact and complete

Biggest issue is size:

- Adding lots of clauses makes the SAT solver slow
- Adding lots of variables can make the SAT solver loose track of the real problem (VSIDS may go off the rails)

Solutions:

- Only add clauses up to a certain size
- Only add a maximum N number of clauses or literals
- Detect symmetries that are cheap to break and can be broken completely

Symmetries: CDCL(T) dynamic breaking

Different ways:

- Add symmetric learnt clauses (“Symmetric Learning”) [BenhamouNabhaniOstrowskiSaidi2010]
- Keep only active symmetry blocking clauses (“Symmetric Explanation Learning”) [DevriendtBogaertsBruynooghe2017]
- Don’t branch into search space that are symmetric (“SymChaff”) [Sabharwal2009]
- Any ideas in the audience?