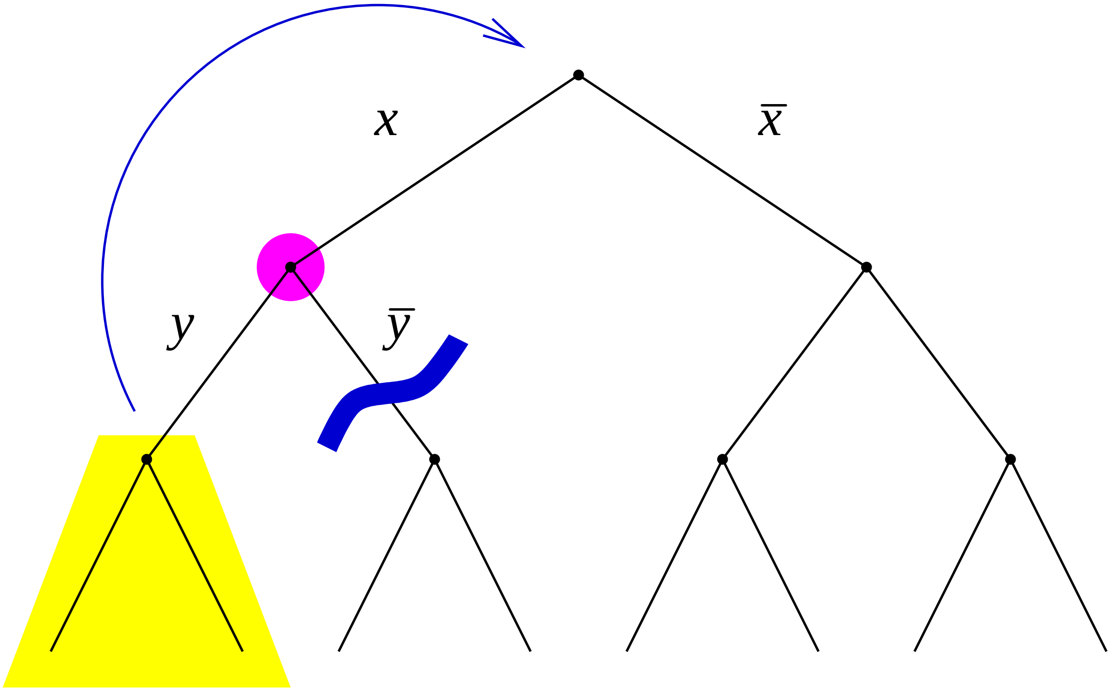
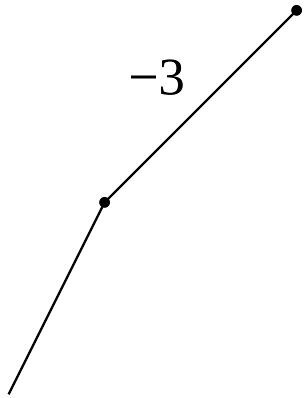
Backjumping



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

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

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

(−1 −2 $\cancel{3}$)

(−1 −2)

(−1 2)

(1 −2)

(1 2)

Split on -3 first (bad decision).

Backjumping Example

~~(−3 1)~~

~~(−3 2)~~

~~(−1 −2 $\cancel{3}$)~~

~~(−1 −2)~~

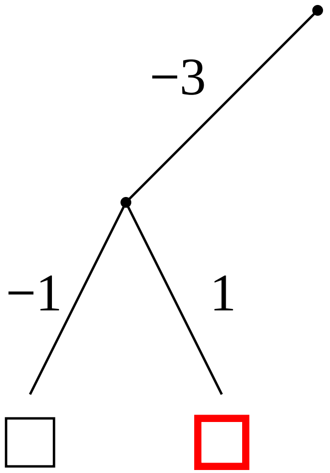
~~(−1 2)~~

($\cancel{1}$ −2)

($\cancel{1}$ 2)

Split on -1 and get first conflict.

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

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

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

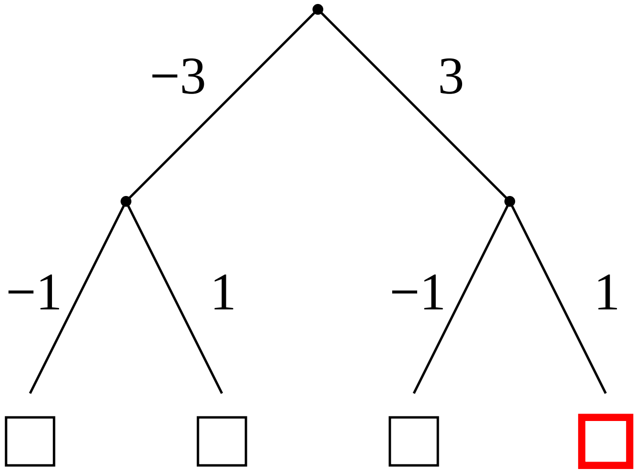
$(\cancel{-1}\ 2)$

~~(1 −2)~~

~~(1 2)~~

Regularly backtrack and assign 1 to get second conflict.

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

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

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

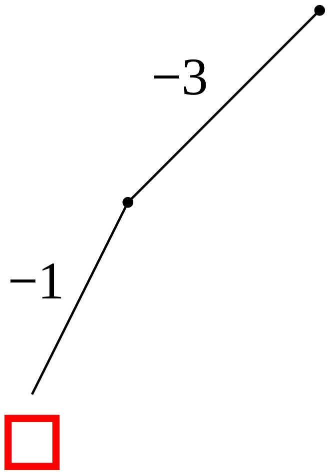
$(\cancel{-1}\ 2)$

~~(1 −2)~~

~~(1 2)~~

Backtrack to root, assign 3 and derive same conflicts.

Backjumping Example



~~(−3 1)~~

~~(−3 2)~~

~~(−1 −2 $\cancel{3}$)~~

~~(−1 −2)~~

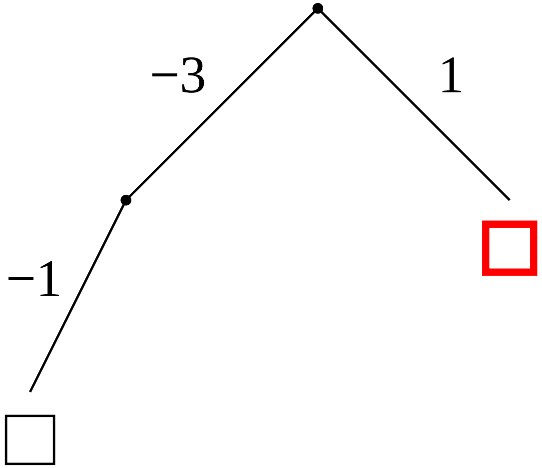
~~(−1 2)~~

($\cancel{1}$ −2)

($\cancel{1}$ 2)

Assignment -3 does not contribute to conflict.

Backjumping Example



~~(−3 1)~~

(−3 2)

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

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

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

~~(1 −2)~~

~~(1 2)~~

So just *backjump* to root before assigning 1.

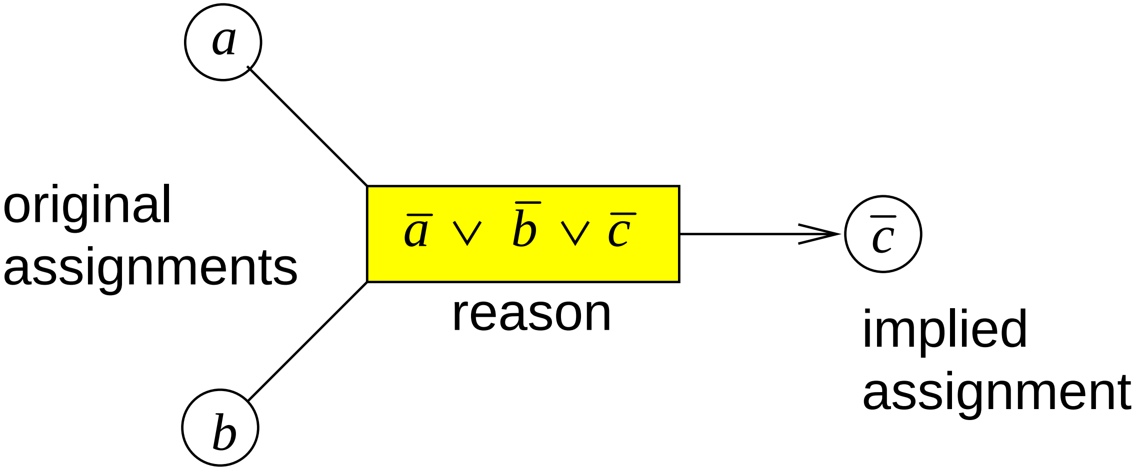
Backjumping

- 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

Implication Graph

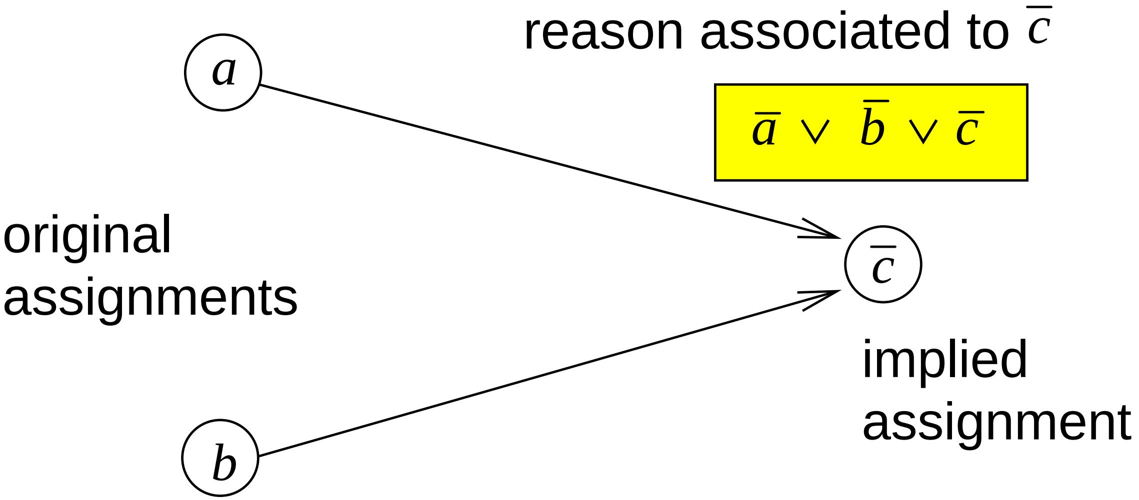
- the implication graph maps inputs to the result of resolutions
- backward from the empty clause all contributing clauses can be found
- the variables in the contributing clauses are contributing to the conflict
- important optimization, since we only use unit resolution
  - generate graph only for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

Learning

- 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

Conflict Driven Backtracking/Backjumping

- 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.

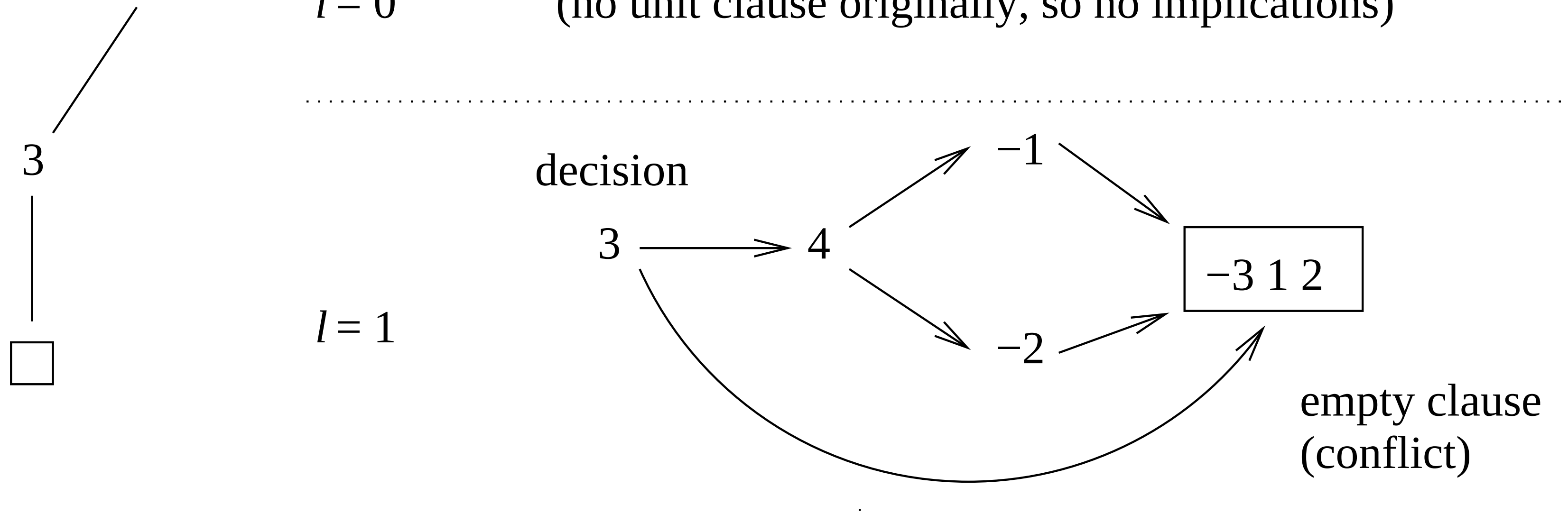
Integers represent literals.

Subtraction means negation.

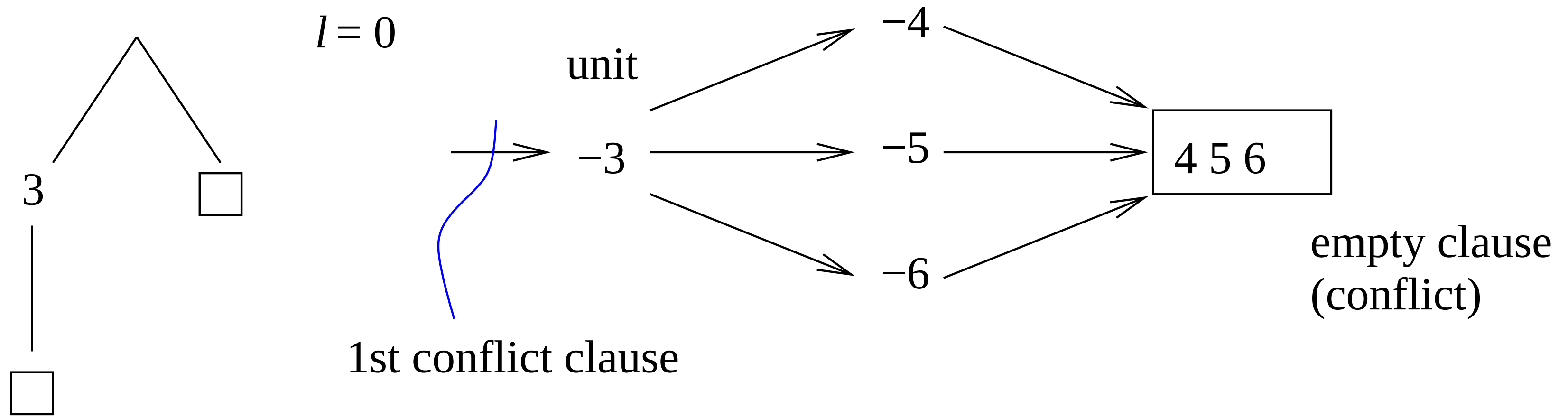
A clause is a zero terminated list of integers.

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)

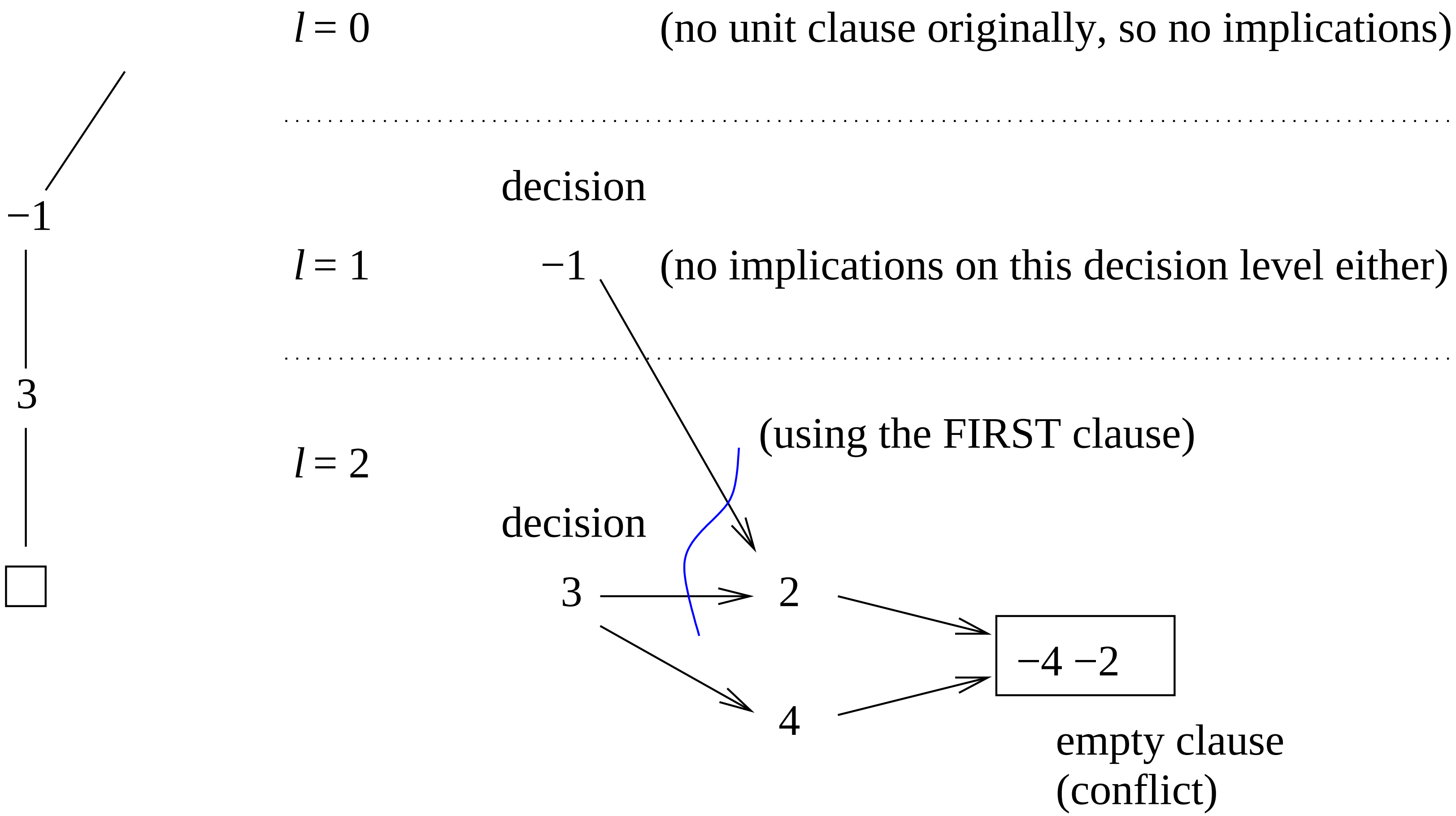


unit clause −3 is generated as learned clause and we backtrackt to $l = 0$



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)



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

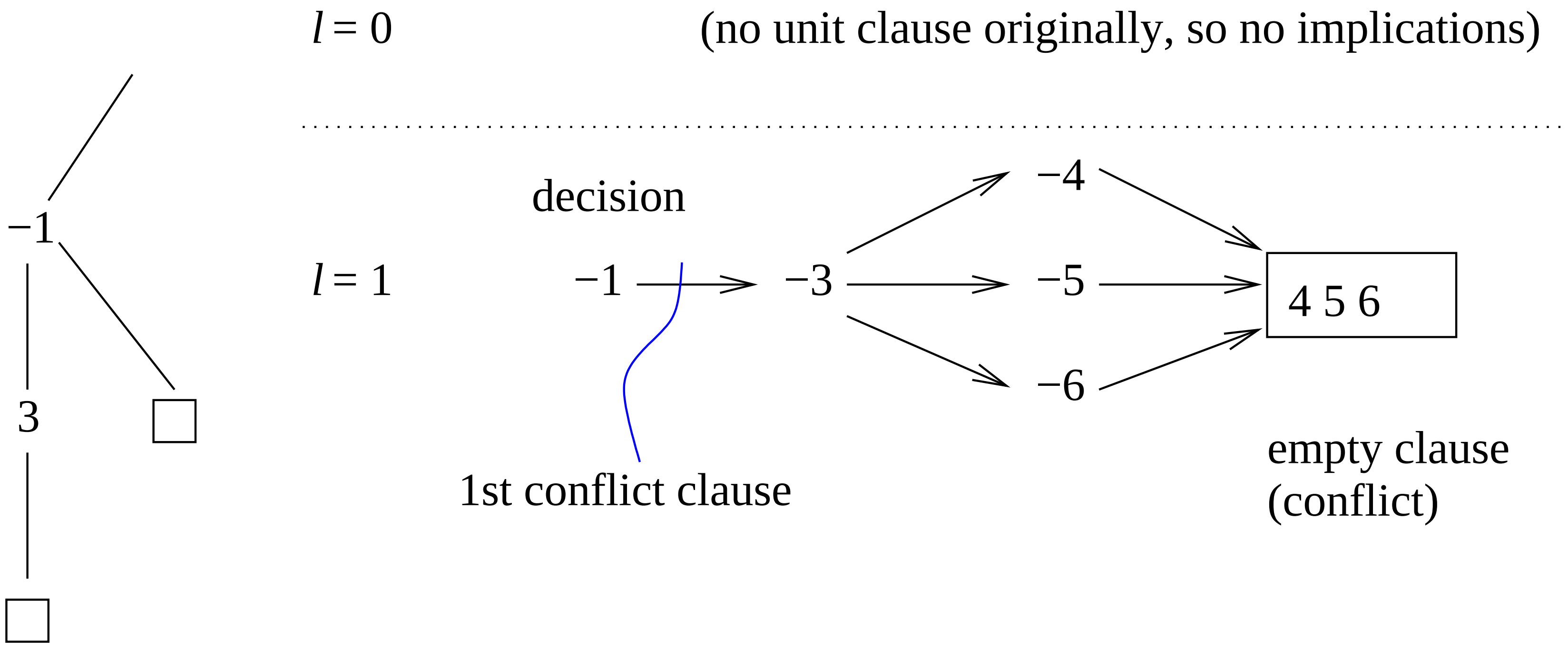
$l = 1$ decision -1 (no implications on this decision level either)

$l = 2$ decision 3 (using the FIRST clause)

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

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

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



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

-1

3

decision

$l = 1$

$-1 \rightarrow -3$

-4

-5

-6

$4\ 5\ 6$

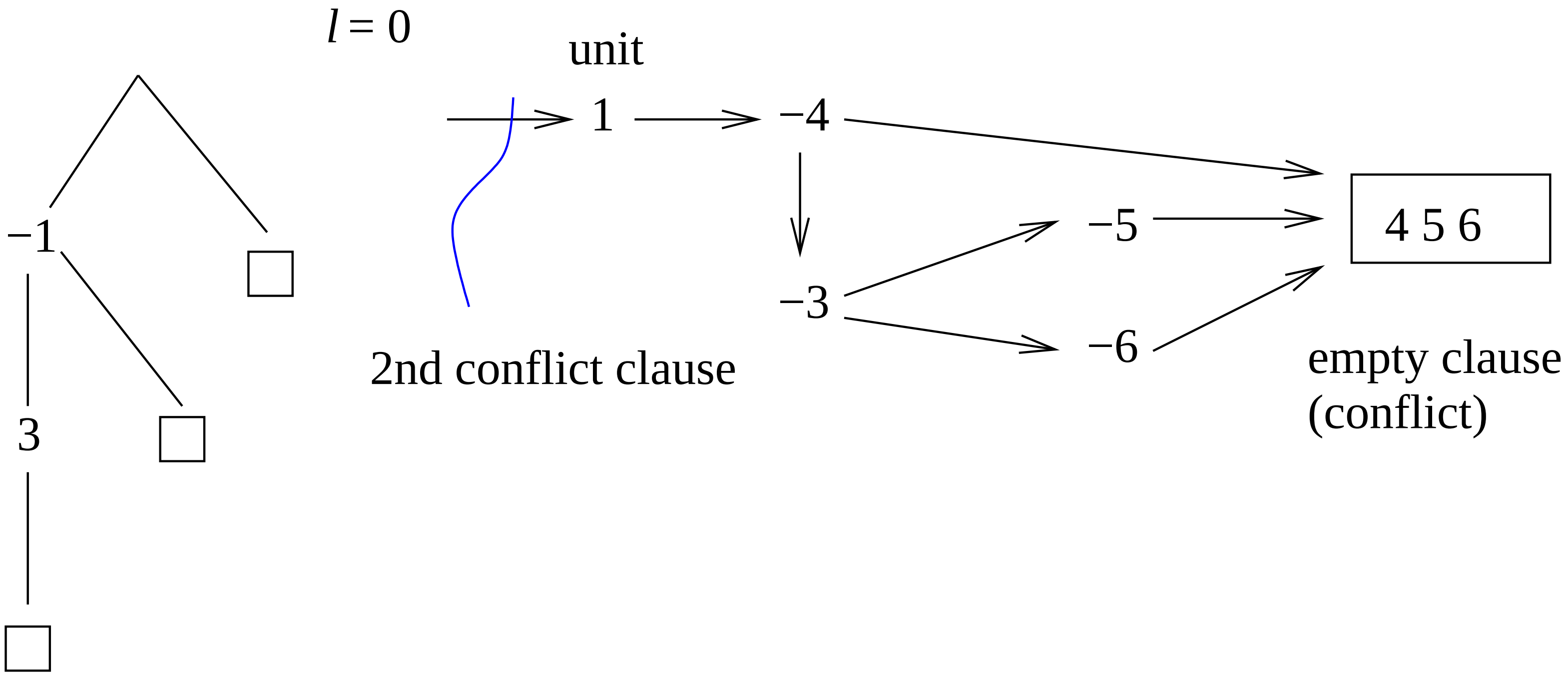
1st conflict clause

empty clause
(conflict)

learned conflict clause is the unit clause 1

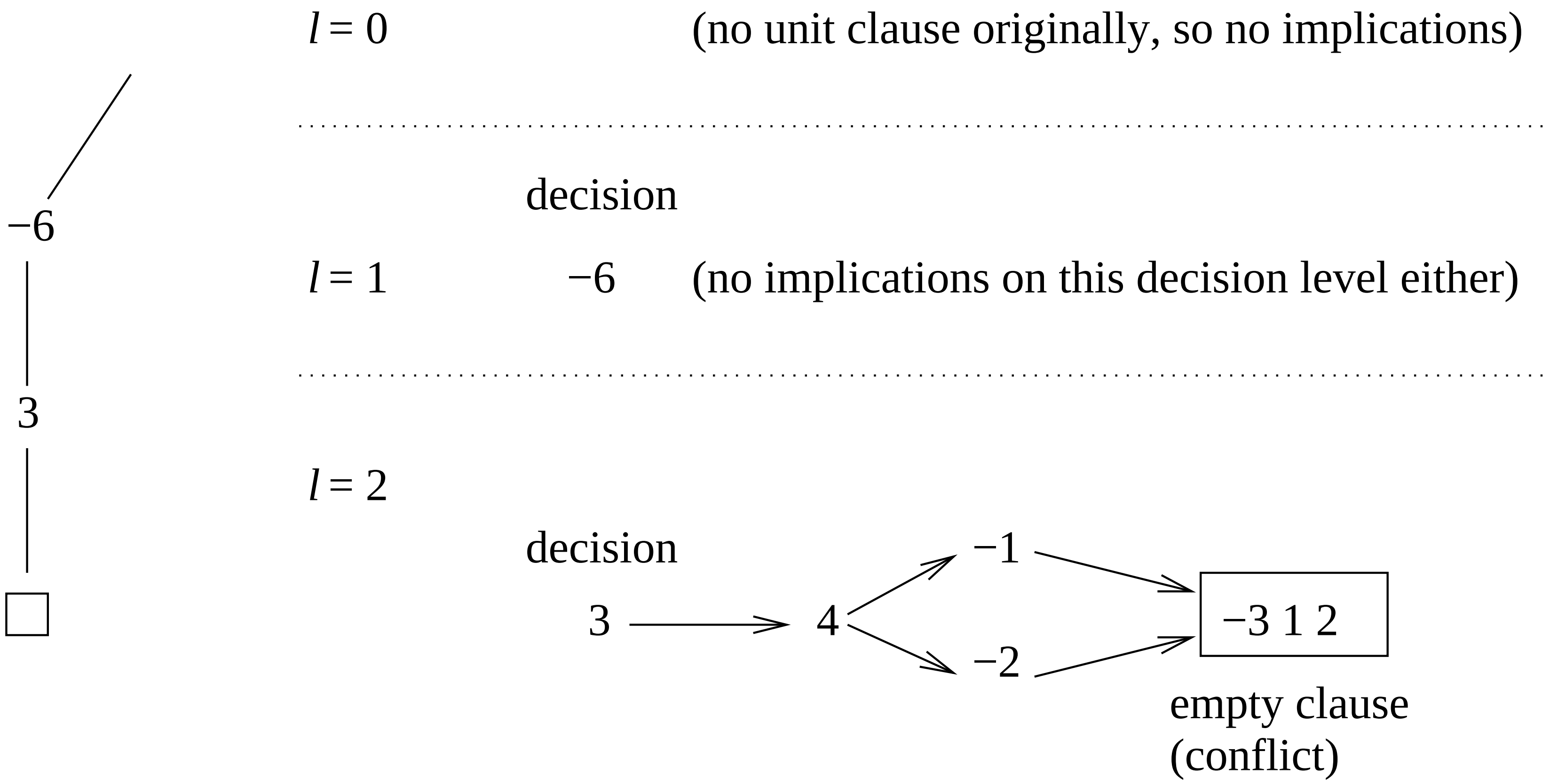
backtrack to decision level $l = 0$

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



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)

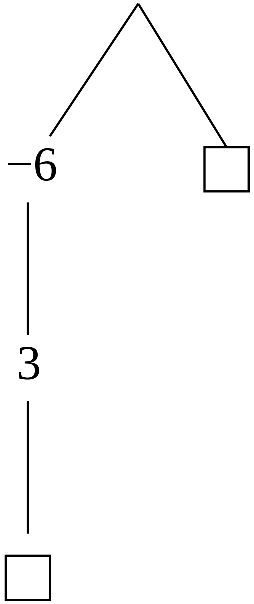
$l = 2$

decision

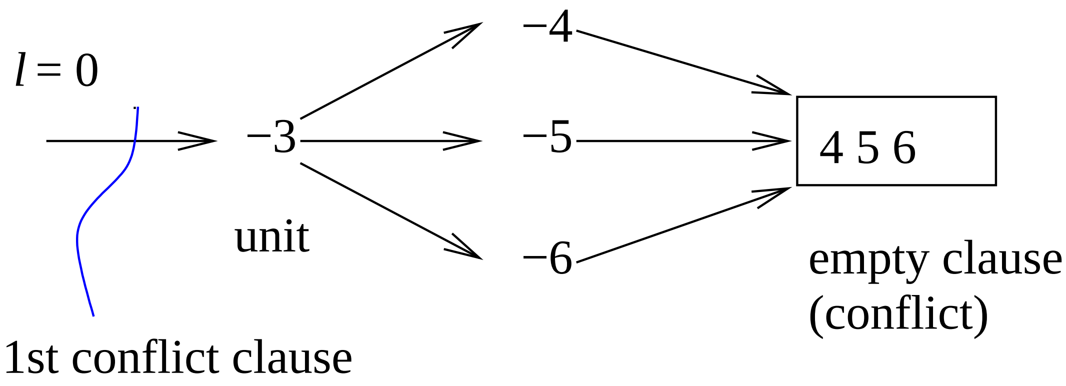
empty clause
(conflict)

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

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


−6

3


$l = 0$

−3

unit

−4

−5

−6

4 5 6

empty clause
(conflict)

1st conflict clause

finally the empty clause is derived which proves unsatisfiability

Toplevel Loop in DP with Learning

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

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

      conflict = deduce (solver);

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

Backtracking in DP with Learning

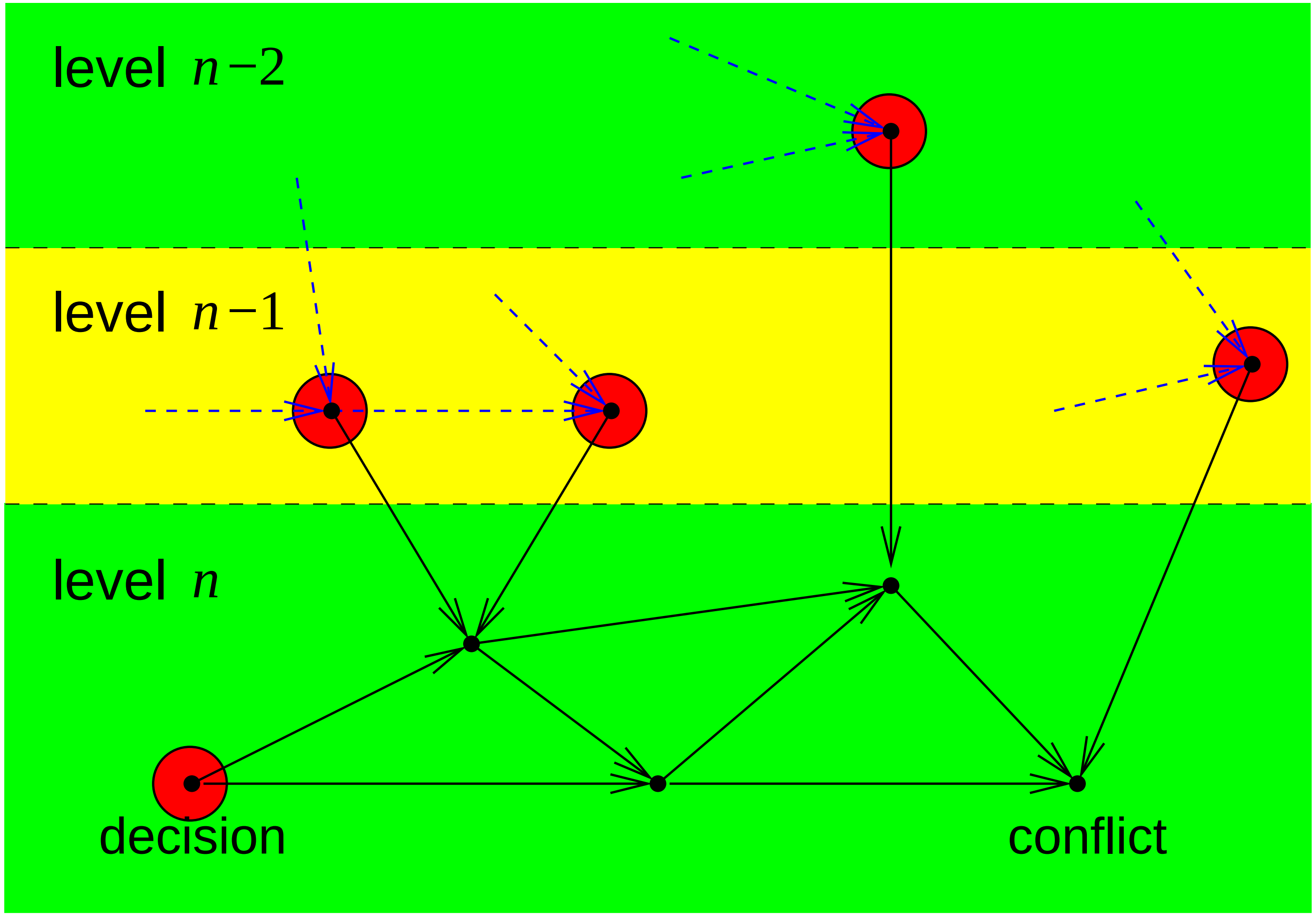
```
int
backtrack (Solver solver, Clause conflict)
{
  Clause learned_clause; Assignment assignment; int new_level;

  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;
}
```

Learning as Resolution

- 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

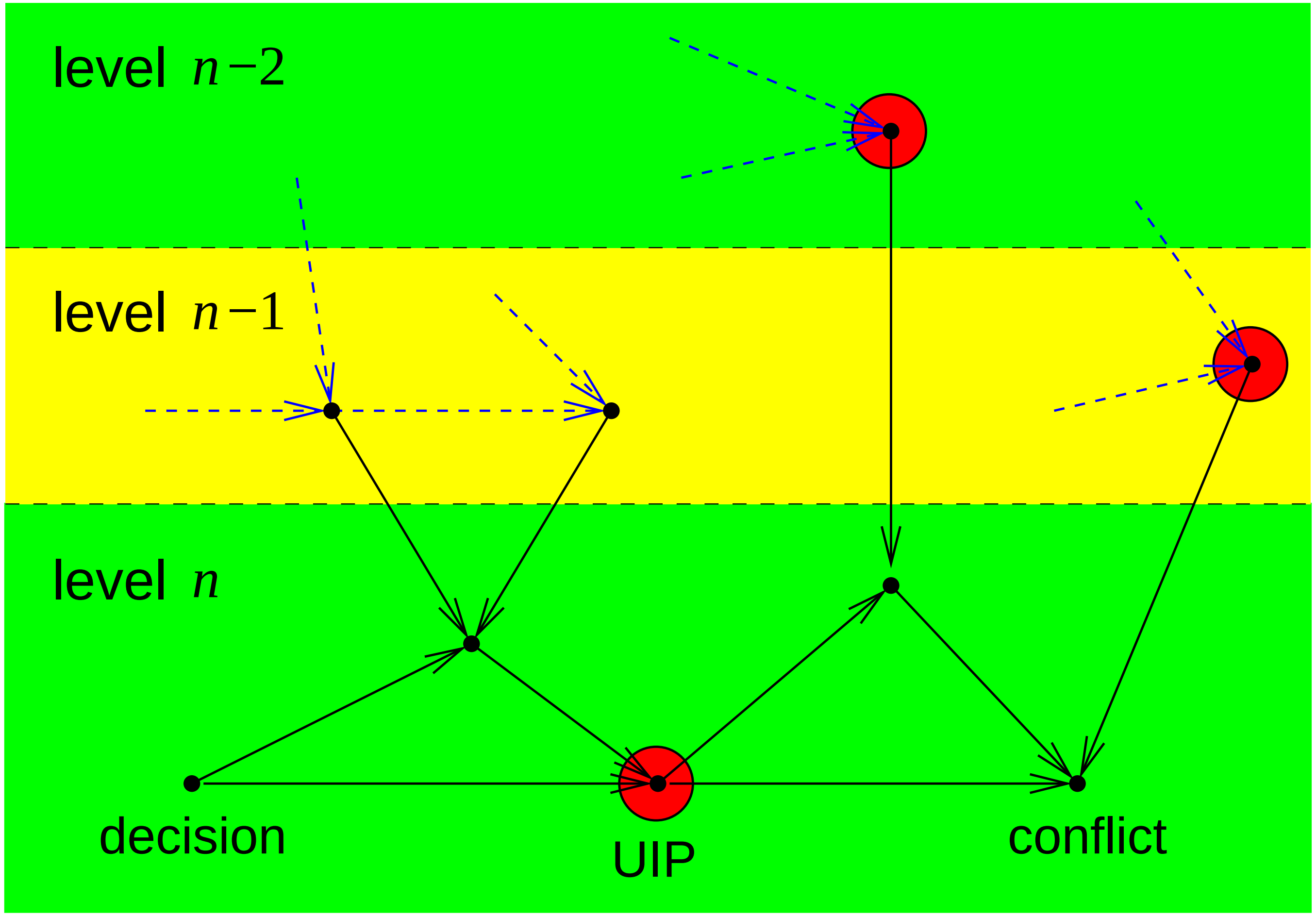
- gives a nice correspondence between resolution and learning in DP
    - allows to generate a resolution proof from a DP run
    - implemented in RELSAT solver

Conflict Clauses as Cuts in the Implication Graph



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

Unique Implication Points (UIP)



UIP = *articulation point* in graph decomposition into biconnected components

(simply a node which, if removed, would disconnect two parts of the graph)

Detection of UIPs

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

Further Options in Using UIPs

- 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

More Heuristics for Conflict Clauses Generation

- intuitively is is important to localize the search (cf cutwidth heuristics)

- cuts for learned clauses may only include UIPs of current decision level

- on lower decision levels an arbitrary cut can be chosen

- multiple alternatives
  - include all the roots contributing to the conflict
  - find minimal cut (heuristically)
  - **cut off at first literal of lower decision level** (works best)

Second Order Dynamic Decision Heuristics: VSIDS

- “second order” because it involves statistics about the search
- Variable State Independent Decaying Sum (VSIDS) decision heuristic
  (implemented in CHAFF and LIMMAT solver)
- VSIDS just counts the occurrences of a literals in conflict clauses
- literal with maximal count (score) is chosen
- 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**)

BERKMIN’s Dynamic Second Order Heuristics

- 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, but in general sligthly more robust than just VSIDS

Restarts

- for satisfiable instances the solver may get stuck in the unsatisfiable part
    - even if the search space contains a large satisfiable part
- often it is a good strategy to abandon the current search and restart
    - restart after the number of decisions reached a *restart limit*
- avoid to run into the same dead end
    - by randomization (either on the decision variable or its phase)
    - and/or just keep all the learned clauses
- for completeness dynamically increase restart limit

Other Types of Learning

- similar to look-ahead heuristics: polynomially bounded search
  - may be recursively applied (however, is often too expensive)

- Stålmarck's Method
  - works on triplets (intermediate form of the Tseitin transformation):

    $x = (a \wedge b)$, $y = (c \vee d)$, $z = (e \oplus f)$ etc.
  - generalization of BCP to (in)equalities between variables
  - **test rule** splits on the two values of a variable

- Recursive Learning (Kunz & Pradhan)
  - works on circuit structure (derives implications)
  - splits on different ways to *justify* a certain variable value

Stålmarck's Method

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

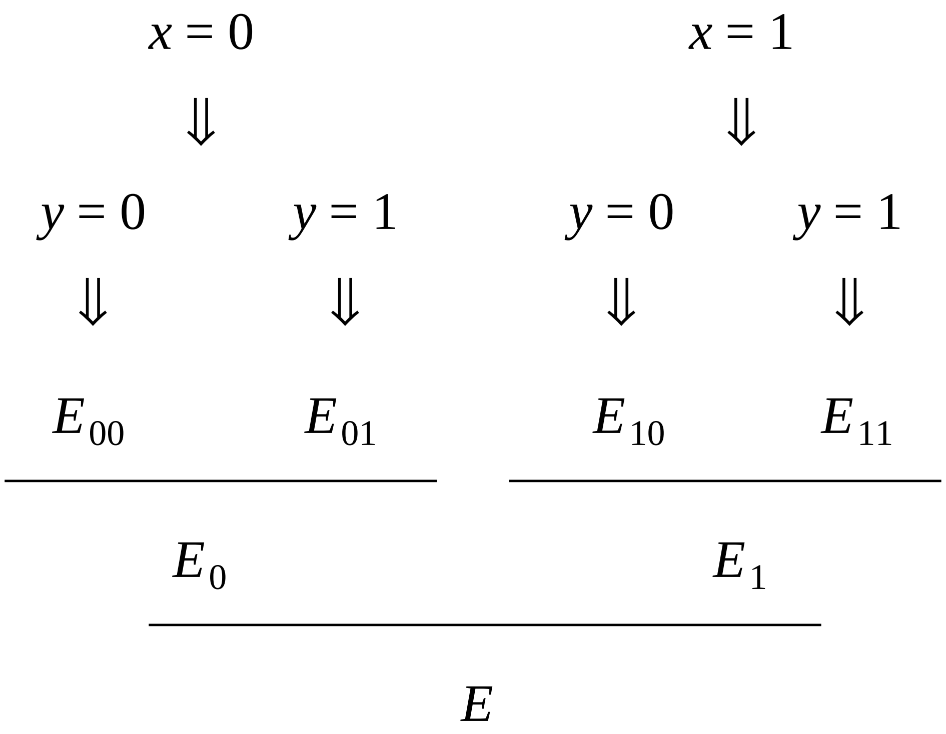
2. structural rules: $\dfrac{x = (a \vee b) \quad y = (a \vee b)}{x = y}$ etc.

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



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

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

Stålmarck’s Method Summary

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