Constraint Reasoning
Part 2: SAT, PB, WCSP

Olivier ROUSSEL
roussel@cril.univ-artois.fr

CRIL-CNRS UMR 8188
Université d'Artois
Lens, France

Tutorial ECAI 2012
Montpellier – August, 22 2012

Outline

- SAT
  - Unit propagation: principle, implementation, incompleteness
  - Inferences: Resolution
  - Learning in CDCL solvers
  - CSP encodings and knowledge compilation to maintain GAC
- Pseudo-Boolean constraints
  - Definitions and expressivity
  - Boolean propagation
  - Inferences: cutting plane system
  - Learning in PB solvers
  - SAT encodings and GAC
- WCSP
  - Definitions and expressivity
  - Equivalence Preserving Transformations
  - Soft consistencies

SAT

The SAT problem

Definition (SAT Problem)

Problem of deciding if a propositional formula in Conjunctive Normal Form (CNF) is satisfiable

Definition (Variable)

A variable only takes Boolean values: $x_i \in \{\text{false}, \text{true}\}$

Definition (Literal)

A literal l_i is a variable or its negation: x_i or $\neg x_i$

Definition (Clause)

A clause C_i is a disjunction of literals: $l_1 \vee l_2 \vee \ldots \vee l_n$
Clauses are often considered as sets of literals

Definition (Formula in CNF)

A formula is a conjunction of clauses: $C_1 \wedge C_2 \wedge \ldots \wedge C_m$

Definitions

Definition (Unit clause)

A unit clause is a clause that contains only one literal.

Definition (Empty clause)

An empty clause is a clause that contains no literal. It is denoted by $\square$.

Definition (Satisfaction of a clause)

A clause is satisfied if at least one of its literal is assigned true.
If every literal is false, the clause is falsified.
The empty clause is always falsified.

Definition

A formula is satisfied if each clause is satisfied. A formula is falsified if at least one of its clauses is falsified.

SAT Problem

- The SAT problem consists in finding an assignment which satisfies the formula (decision problem).
- The SAT problem is NP-complete.

Notations

- Boolean values will be denoted as
  - {false,true},
  - {T,F}
  - or sometimes {0,1}
- In the figures, green will be used to indicate a literal that is assigned true, and red will be used for a falsified literal.

Introduction to SAT: the SAT game

A funny illustration of the SAT problem:
`http://www.cril.univ-artois.fr/~roussel/satgame/satgame.php`

Lessons learnt

- Two main steps in the resolution process:
  - search:
    guess an assignment (heuristic) repeatedly until every constraint is satisfied or until a constraint is violated (in this case, backtrack and try another new assignment in a consistent way)
  - inference:
    infer which assignments must be made in order to prevent dead ends
- Search is much easier when we do strong inferences. Search even becomes linear if we are able to detect inconsistencies in *any* case!
- No silver bullet: complete inference is NP-complete!
- Balance between search and inference is especially critical in SAT (millions of variables/clauses)

Unit Propagation (UP)

- When all literals in a clause are false but one, this literal must be assigned true (otherwise the clause is falsified).
- Unit Propagation is the iterated application of this rule.
- Linear complexity
- Main inference rule in SAT solvers!

UP Implementation: Counters

- Occurrence lists allow to know which clauses contain a given literal.
- For each clause, a counter indicates the number of unassigned literals present in the clause.
- When a literal l is assigned true:
  - Each clause containing l is marked as satisfied and later ignored.
  - For each clause containing $\neg l$, the counter of unassigned literals is decremented. When it reaches 1, the only unassigned literal remaining in the clause must be assigned true.
- On backtrack, counters must be restored and previously satisfied clauses must be considered again.

Comments On the Counters Implementation

- Advantages:
  - A precise count of the clauses still unsatisfied can be kept, as well as statistics on these clauses (number of clauses of length n, number of occurrences of each literals,...).
  - This information is especially useful in heuristics.
- Disadvantages:
  - After the assignment of variable, every clause containing that variable must be updated.
  - The backtrack requires the same amount of work.

UP Implementation: watched literals

- To detect that all literals but one are false, it is sufficient to make sure that at least two literals are not falsified in the clause (invariant).
- These two literals are called watched literals.
- The watched literals can be either unassigned or true.
- Whenever a watched literal becomes false, it must be replaced by another literal of the clause with is not watched, and either unassigned or true.
- If the falsified watched literal cannot be replaced, propagation occurs, deriving either an implied literal, or identifying a falsified clause (conflict).

Watched Literals Illustration (1/7)

Assignment stack:

invariant: watched literals must not be false

WL1 WL2

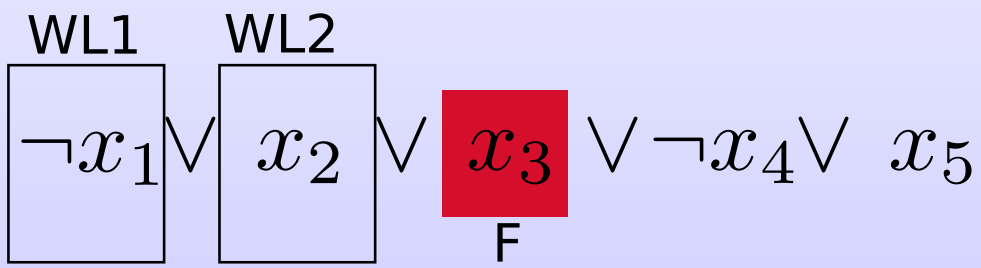
$$\boxed{\neg x_1} \vee \boxed{x_2} \vee x_3 \vee \neg x_4 \vee x_5$$

Watched Literals Illustration (2/7)

Assignment stack:

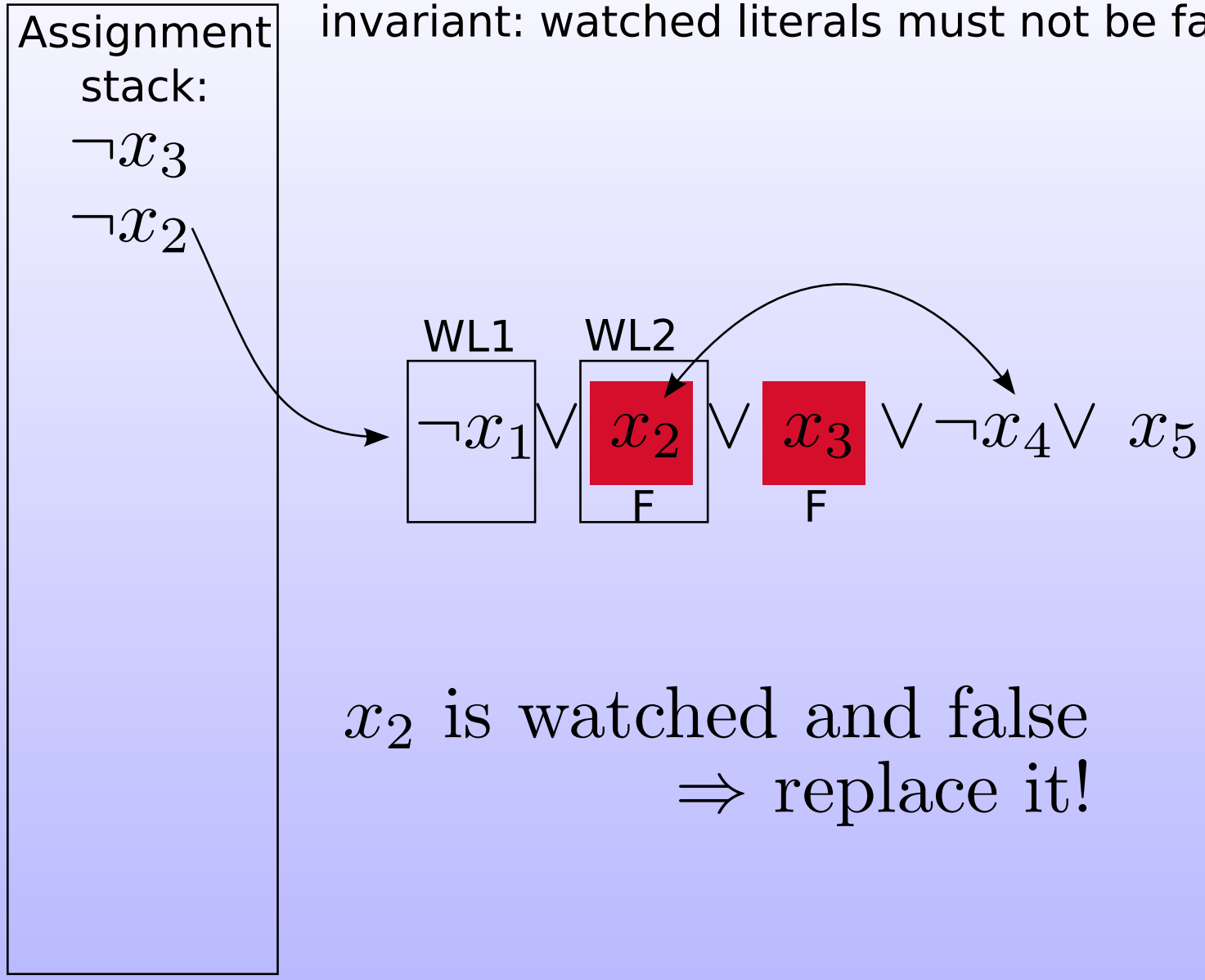
$\neg x_3$

invariant: watched literals must not be false

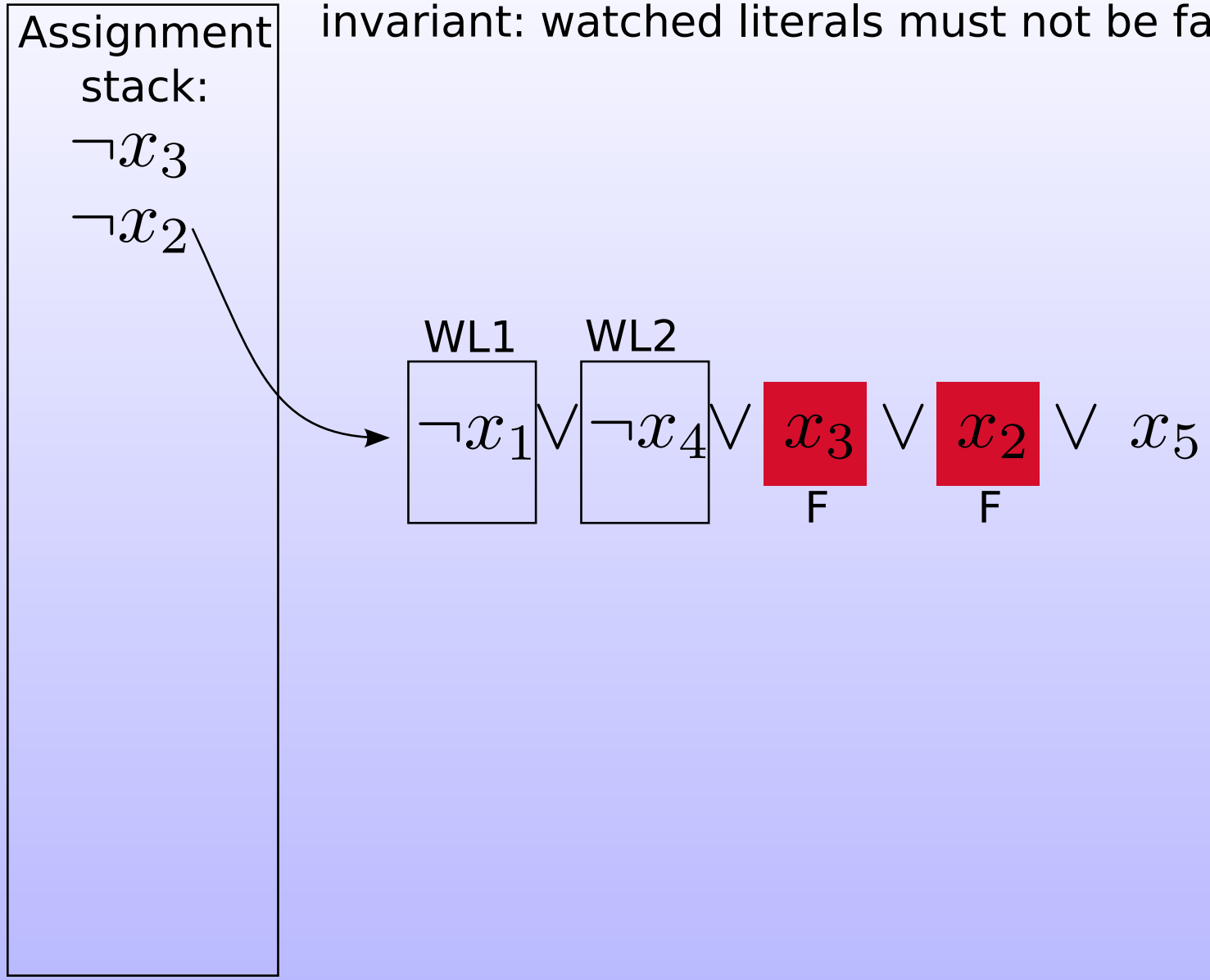


x_3 is not watched, do nothing!

Watched Literals Illustration (3/7)



Watched Literals Illustration (4/7)



Watched Literals Illustration (5/7)

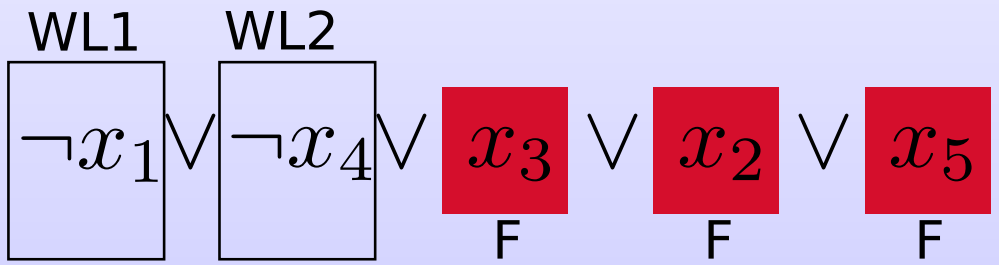
Assignment stack:

$\neg x_3$

$\neg x_2$

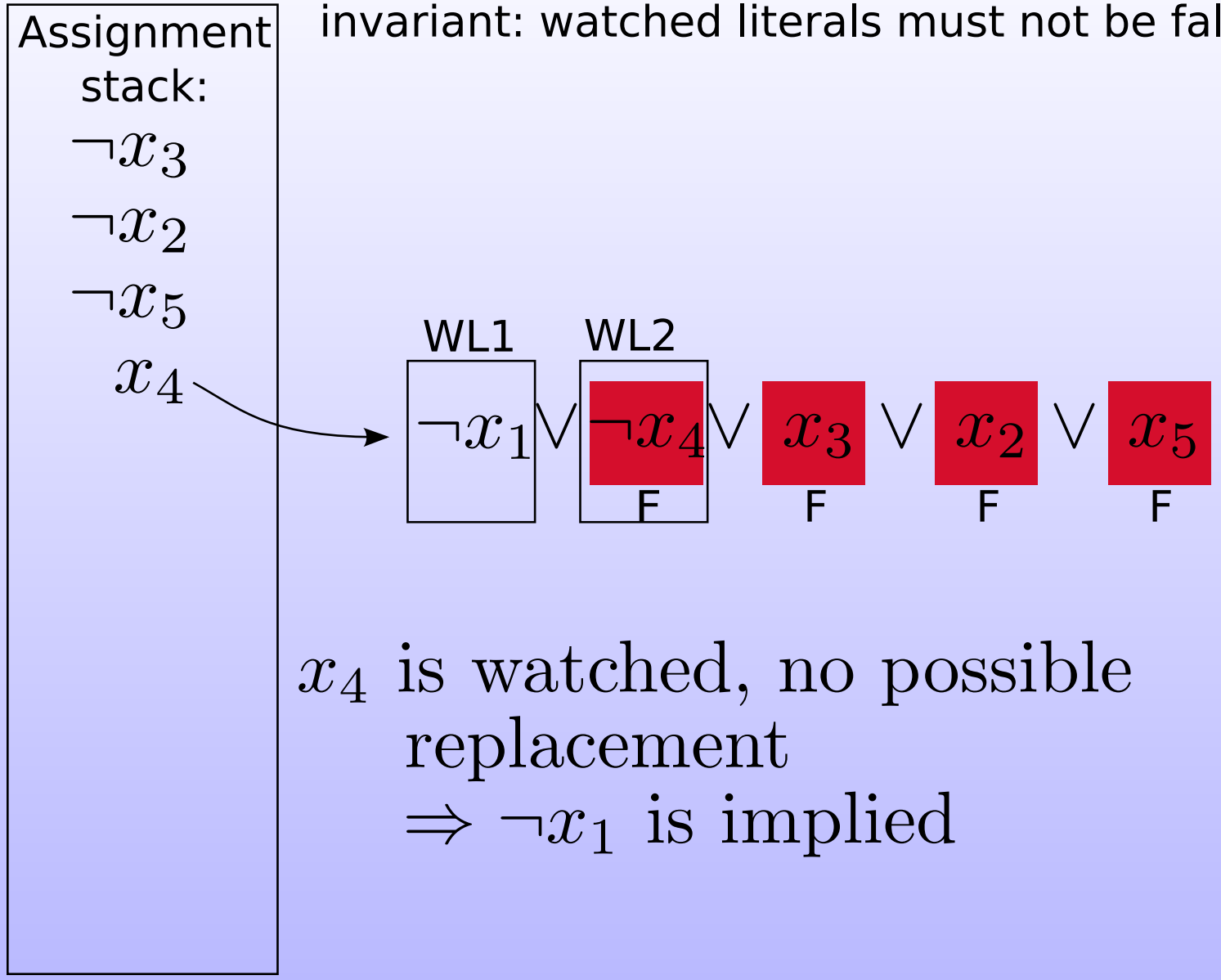
$\neg x_5$

invariant: watched literals must not be false



x_5 is not watched, do nothing!

Watched Literals Illustration (6/7)



Watched Literals Illustration (7/7)

Assignment stack:

$\neg x_3$

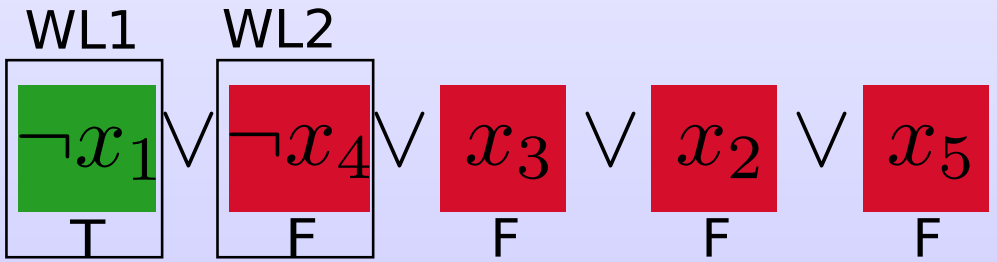
$\neg x_2$

$\neg x_5$

x_4

$\neg x_1$

invariant: watched literals must not be false



Benefits of Watched literals

- Advantages:
  - When a literal is assigned, UP will only consider clauses where this literal is watched. This is generally a small percentage of the number of clauses containing this literal.
    $\Rightarrow$ less work to do, possibility to increase the number of clauses.
  - Very efficient on long clauses.
  - No work on backtrack!
- Disadvantages:
  - No gain for clauses of length 2.
  - No way to know exactly which clauses are satisfied and the number of unassigned literals in a clause. Heuristics can't use this information anymore.
    $\Rightarrow$ new family of heuristics based on clauses and literals activities (VSIDS)

Incompleteness of UP

- Unit Propagation is not complete, which means that it does not derive every literal which is semantically implied by the formula.
- Example:

$$\begin{cases} a \vee b \\ a \vee c \\ \neg b \vee \neg c \end{cases}$$

- a is semantically implied by this formula
  - Either b or c must be false (third clause).
  - If b is false, then a must be true (first clause).
  - If c is false, then a must be true (second clause).
  - In both cases, a must be true.
- Each clause contains 2 unassigned literals, UP propagation does not infer anything.
- Determining if a literal is implied is NP-complete. UP has linear complexity. Therefore, incompleteness is not a surprise.

Failed literals

- At any time, one can try to assert a literal l, perform unit propagation and then unassign l.
- If unit propagation ended with a falsified clause, then l is a failed literal. Setting l to true generates an inconsistency. Therefore, it is necessarily false.
- If unit propagation doesn't generate a conflict, relevant statistics on the number of propagated literals can be collected and used in the heuristics (look-ahead).
- Example:

$$\begin{cases} a \vee b \\ a \vee c \\ \neg b \vee \neg c \end{cases}$$

Asserting $\neg a$ will allow unit propagation to derive b and c which falsifies clause $\neg b \vee \neg c$. Therefore, we can infer that a must be true and use it in the following propagations.

Failed literals (2)

- The failed literal rule derives more literals than UP, but it uses additional information (the literal that is tried)
- Only a limited number of literals can be tested at each node of the search tree, otherwise the process is too time consuming (balance between search and inference).
- Singleton Arc Consistency (SAC) in CSP uses the same principle as Failed Literals in SAT.

Resolution Proof System

- Two inference rules:

$$\frac{\begin{array}{c} l \vee A \\ \neg l \vee B \\ \neg \exists l' \text{ s.t. } l' \in A \wedge \neg l' \in B \end{array}}{A \vee B} \quad \text{(resolution)}$$

$$\frac{l \vee l \vee C}{l \vee C} \quad \text{(merging)}$$

- The merging rule is often implicit in propositional logic (because clauses are often considered as sets). But it is explicit (and compulsory) in first order logic, and allows to understand the incompleteness of unit propagation.

Comments on Resolution

- The resolution proof system is complete, which means that when a formula is unsatisfiable, there always exists a resolution proof that derives the empty clause $\square$ from its clauses.
- Unit propagation can be seen as the iteration of a weak form of resolution where one of the two clauses must be a unit clause.

Characterization of UP Incompleteness

- A literal which is implied by a formula, but cannot be derived by resolution without using the merging rule, cannot be derived by UP.
- Example:

$$\begin{cases} a \vee b \\ a \vee c \\ \neg b \vee \neg c \end{cases}$$

a can be obtained by resolution but the merging rule is required to derive it. Therefore, UP cannot derive it.

$$\dfrac{\dfrac{a \vee b \quad \neg b \vee \neg c}{a \vee \neg c} \qquad a \vee c}{\dfrac{a \vee a}{a}}$$

Learning

- When UP falsifies a clause (conflict), it is possible to generate an explanation of this conflict and learn a clause that will prevent other occurrences of this conflict.
- The explanation is generated by tracing back the UP steps and performing resolution steps on the involved clauses.
- To trace back UP, a reason must be kept for each implication of a literal. The reason is the clause which implied the literal. The reasons form an implication graph.
- Learned clauses are also used to drive the search (more on this later) and perform non chronological backtracking.
- Learned clauses must be regularly forgotten, otherwise UP will take more and more time.

Decision level

Definition (Decision)

A literal that a solver asserts and which is not derived by an inference process is a decision.

Definition (Decision level)

The decision level is the number of decision that were taken.

Definition (Decision level 0)

The literals that are asserted before any decision is taken are said to be assigned at level 0.

Decision level

- Literals assigned at level 0 are either unit clauses from the initial formula or unit clauses that were derived during search.
- The formula is unsatisfiable iff a conflict occurs at level 0 in a CDCL solver.
- A literal l asserted at decision level d will be denoted $l@d$

Example of Learning

Assignment stack:

$$C_1 : \neg x_1 \vee x_2$$

$$C_2 : \neg x_3 \vee x_4$$

$$C_3 : \neg x_2 \vee \neg x_6 \vee x_7$$

$$C_4 : \neg x_5 \vee x_6$$

$$C_5 : \neg x_6 \vee x_8$$

$$C_6 : \neg x_7 \vee \neg x_8$$

Example: New Decision $x_1 = \mathit{true}$

Assignment stack:

$x_1@1$

$C_1 : \underset{\text{F}}{\neg x_1} \vee x_2$

$C_2 : \neg x_3 \vee x_4$

$C_3 : \neg x_2 \vee \neg x_6 \vee x_7$

$C_4 : \neg x_5 \vee x_6$

$C_5 : \neg x_6 \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

Example: Unit Propagation

Assignment stack:

$x_1@1$

$x_2@1(C_1)$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \neg x_3 \vee x_4$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \neg x_6 \vee x_7$

$C_4 : \neg x_5 \vee x_6$

$C_5 : \neg x_6 \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

Example: New Decision $x_3 = \mathit{true}$

Assignment stack:

$x_1@1$

$x_2@1(C_1)$

$x_3@2$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \underset{\text{F}}{\neg x_3} \vee x_4$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \neg x_6 \vee x_7$

$C_4 : \neg x_5 \vee x_6$

$C_5 : \neg x_6 \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

Example: Unit Propagation

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \underset{\text{F}}{\neg x_3} \vee \underset{\text{T}}{x_4}$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \neg x_6 \vee x_7$

$C_4 : \neg x_5 \vee x_6$

$C_5 : \neg x_6 \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

Example: New Decision $x_5 = \mathit{true}$

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \underset{\text{F}}{\neg x_3} \vee \underset{\text{T}}{x_4}$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \underset{\text{F}}{\neg x_6} \vee x_7$

$C_4 : \underset{\text{F}}{\neg x_5} \vee x_6$

$C_5 : \neg x_6 \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

Example: Unit Propagation

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$
$x_6@3(C_4)$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \underset{\text{F}}{\neg x_3} \vee \underset{\text{T}}{x_4}$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \underset{\text{F}}{\neg x_6} \vee x_7$

$C_4 : \underset{\text{F}}{\neg x_5} \vee \underset{\text{T}}{x_6}$

$C_5 : \underset{\text{F}}{\neg x_6} \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

Example: Unit Propagation

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$
$x_6@3(C_4)$
$x_7@3(C_3)$

$C_1 : \neg x_1 \vee x_2$
F T

$C_2 : \neg x_3 \vee x_4$
F T

$C_3 : \neg x_2 \vee \neg x_6 \vee x_7$
F F T

$C_4 : \neg x_5 \vee x_6$
F T

$C_5 : \neg x_6 \vee x_8$
F

$C_6 : \neg x_7 \vee \neg x_8$
F

Example: Unit Propagation

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$
$x_6@3(C_4)$
$x_7@3(C_3)$
$x_8@3(C_5)$

$$C_1 : \underset{F}{\neg x_1} \vee \underset{T}{x_2}$$

$$C_2 : \underset{F}{\neg x_3} \vee \underset{T}{x_4}$$

$$C_3 : \underset{F}{\neg x_2} \vee \underset{F}{\neg x_6} \vee \underset{T}{x_7}$$

$$C_4 : \underset{F}{\neg x_5} \vee \underset{T}{x_6}$$

$$C_5 : \underset{F}{\neg x_6} \vee \underset{T}{x_8}$$

$$C_6 : \underset{F}{\neg x_7} \vee \underset{F}{\neg x_8}$$

Example: Conflict Discovery

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$
$x_6@3(C_4)$
$x_7@3(C_3)$
$x_8@3(C_5)$
$\square@3(C_6)$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \underset{\text{F}}{\neg x_3} \vee \underset{\text{T}}{x_4}$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \underset{\text{F}}{\neg x_6} \vee \underset{\text{T}}{x_7}$

$C_4 : \underset{\text{F}}{\neg x_5} \vee \underset{\text{T}}{x_6}$

$C_5 : \underset{\text{F}}{\neg x_6} \vee \underset{\text{T}}{x_8}$

$C_6 : \underset{\text{F}}{\neg x_7} \vee \underset{\text{F}}{\neg x_8}$

Example: Conflict Analysis

Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$
$x_6@3(C_4)$
$x_7@3(C_3)$
$x_8@3(C_5)$
$\square@3(C_6)$

$C_6 : \neg x_7@3 \vee \neg x_8@3$

Example: Conflict Analysis

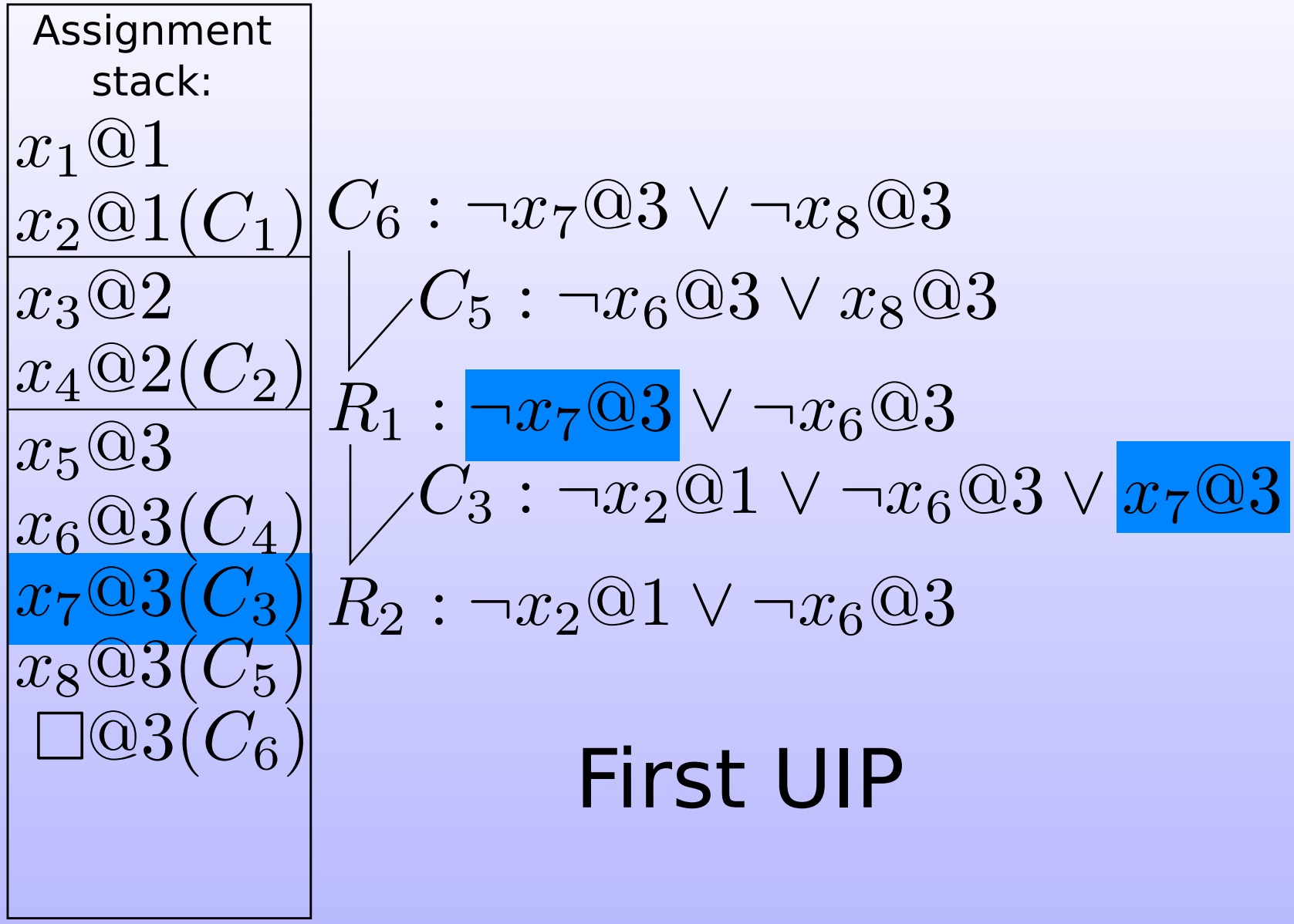
Assignment stack:
$x_1@1$
$x_2@1(C_1)$
$x_3@2$
$x_4@2(C_2)$
$x_5@3$
$x_6@3(C_4)$
$x_7@3(C_3)$
$x_8@3(C_5)$
$\square@3(C_6)$

$C_6 : \neg x_7@3 \vee \neg x_8@3$

$C_5 : \neg x_6@3 \vee x_8@3$

$R_1 : \neg x_7@3 \vee \neg x_6@3$

Example: Conflict Analysis



Learned Clause

- Resolvent R_2 contains only one literal at the current decision level ($\neg x_6@3$). This was not the case for R_1 or C_6.
- $\neg x_6@3$ is called a Unique Implication Point (UIP). The assignment of this literal at the current decision level is the cause of the conflict.
- In the first UIP scheme (most frequently used), we stop conflict analysis at the first UIP, that is, as soon as the resolvent contains only one literal at the conflict level. The decision literal is always a UIP.
- The clause we learn is $R_2 : \neg x_2@1 \vee \neg x_6@3$. Learning this clause now allows UP to infer $\neg x_6$ from x_2, which was impossible before since x_6 is a merge literal.
- Learning can also be seen as an on demand knowledge compilation that adds clauses to improve the completeness of UP.

Backtrack

- We backtrack to the greatest decision level of literals in the learned clause except the UIP.
- In our example, the learned clause is $R_2 : \neg x_2@1 \vee \neg x_6@3$ and we backtrack to level 1.
- In this example, level 2 is skipped because it is not relevant for this conflict.
- At level 1, the learned clause is a unit clause which propagates $\neg x_6$. It is an *asserting clause*.
- The learned clause prevents the conflict from occurring again.

Example: Backtrack

Assignment stack:

$x_1@1$

$x_2@1(C_1)$

$\neg x_6@1(R_2)$

$C_1 : \underbrace{\neg x_1}_{\text{F}} \vee \underbrace{x_2}_{\text{T}}$

$C_2 : \neg x_3 \vee x_4$

$C_3 : \underbrace{\neg x_2}_{\text{F}} \vee \underbrace{\neg x_6}_{\text{T}} \vee x_7$

$C_4 : \neg x_5 \vee \underbrace{x_6}_{\text{F}}$

$C_5 : \underbrace{\neg x_6}_{\text{T}} \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

$R_2 : \underbrace{\neg x_2}_{\text{F}} \vee \underbrace{\neg x_6}_{\text{T}}$

Example: Backtrack

Assignment stack:

$x_1@1$

$x_2@1(C_1)$

$\neg x_6@1(R_2)$

$\neg x_5@1(C_4)$

$C_1 : \underset{\text{F}}{\neg x_1} \vee \underset{\text{T}}{x_2}$

$C_2 : \neg x_3 \vee x_4$

$C_3 : \underset{\text{F}}{\neg x_2} \vee \underset{\text{T}}{\neg x_6} \vee x_7$

$C_4 : \underset{\text{T}}{\neg x_5} \vee \underset{\text{F}}{x_6}$

$C_5 : \underset{\text{T}}{\neg x_6} \vee x_8$

$C_6 : \neg x_7 \vee \neg x_8$

$R_2 : \underset{\text{F}}{\neg x_2} \vee \underset{\text{T}}{\neg x_6}$

Conflict Analysis Algorithm

- Start with the conflict clause
- As long as there is more than one literal at the current decision level in the clause,
  - Identify the literal l in the current clause that was assigned last (closest to the top of the assignment stack) and resolve the current clause with the reason of l upon literal l. The resolvent becomes the new current clause.
- Simplify the learned clause (*optional*)
- Backtrack to the greatest decision level of a literal in the learned clause (except the conflict level)

Learned Clause Simplification

- The learned clause can be simplified when a subset of the literals in the clause imply the negation of another literal in the clause. In this case, this literal can be erased by resolution (the resolvent subsumes the learned clause).
- This can be easily checked by using the results of UP and the identified reasons.
- Example:

Learned :	$l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6$
UP	$\neg l_1 \vee l_2 \vee l_3 \vee x_1 \vee x_2$
UP	$\neg x_1 \vee l_4$
UP	$\neg x_2 \vee l_2$
Simplification :	$l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6$

- Advantages: backtrack to a higher level and better pruning of the search tree by the shortened clause.

Conflict Driven, Clause Learning (CDCL) Solvers

```
1:  function BASIC_CDCL(F: formula)
2:      while true do
3:          while UP(F)=CONFLICT do
4:              if ConflictAnalysis(F)=conflict at level 0 then
5:                  return UNSATISFIABLE
6:              else
7:                  Add the learned clause and backtrack
8:              end if
9:          end while
10:         if all variables assigned in F then
11:             return SATISFIABLE
12:         else
13:             Decide(F)                    ▷ select and assert a literal
14:         end if
15:     end while
16: end function
```

Other components of a CDCL solver

- Heuristics:
  Increase the weight of variables appearing in a learned clause and regularly decay the old weights to focus on recent conflicts (VSIDS: Variable State Independent Decaying Sum)
- Restarts:
  Periodically, backtrack to level 0 and restart the search to avoid being stuck in a wrong initial assignment.
- Phase Saving:
  When a variable is chosen by the heuristics (decision), assign it the same value as in the last assignment.

Other components of a CDCL solver

- Clause Deletion Strategy:
  Periodically remove the learned clauses that are considered to be less useful. Very important to keep good performances for UP.
- LBD (Literals Block Distance):
  - LBD = number of different decisions levels in a learned clause.
  - Clauses with small LBD appear to be more important (tight coupling between propagation sequences) and should be preserved.
  - Allows a more agressive clause deletion strategy.

Some Encodings of CSP Problems into SAT

Several encodings have been proposed to encode CSP instances with constraints in extension into a SAT formula:

- Direct encoding
- Support encoding
- Log encoding
- Order encoding
- ...

Direct encoding: Encoding Variables

- Let X be a CSP variable and $dom(X) = \{v_1, v_2, \ldots, v_n\}$
- For each $v_i \in dom(X)$, a Boolean variable b_{X,v_i} is created with the intended semantics $b_{X,v_i} = true \Leftrightarrow X = v_i$.
- "Variable X must be assigned a value" is translated as

$$\bigvee_{v_i \in dom(X)} b_{X,v_i} \text{ (at-least-one clause)}$$

- In CSP, a variable can only take one value. This can be enforced by the following exclusion clauses (at-most-one clauses)

$$\forall v_i \in dom(X), v_j \in dom(X), i \neq j, \neg b_{X,v_i} \vee \neg b_{X,v_j}$$

Exclusion clauses

- The number of exclusion clauses required to enforce that a variable take at most one value is quadratic in the size of the clause.
- Two solutions:
  - Just forget these exclusion clauses. In this case the SAT formula is no more model-equivalent to the CSP problem, but is is satisfiability-equivalent (i.e. satisfiable iff the CSP is satisfiable).
  - use a straightforward encoding of the at-most-one constraint which is linear, only uses binary clauses, but introduces extra variables.

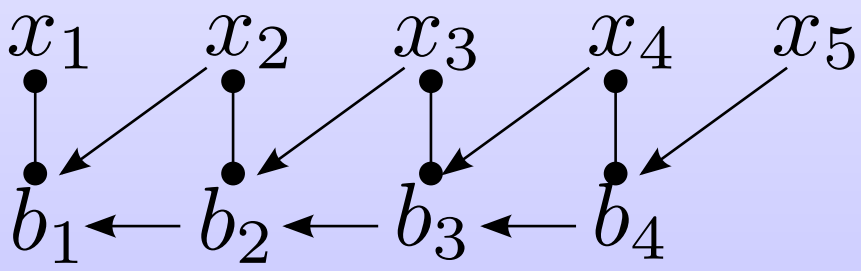
Linear Encoding of at-most-one

$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5$$

Linear Encoding of at-most-one: extra clauses

$$a \longrightarrow b \equiv a \text{ implies } b \equiv \neg a \vee b$$

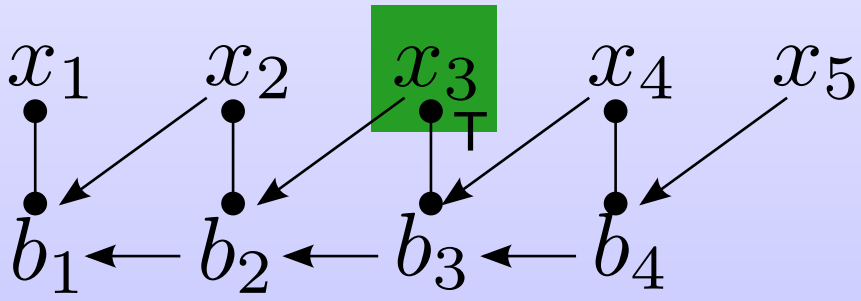
$$a \bullet\!\!-\!\!\bullet\, b \equiv \neg a \vee \neg b \equiv atmost(1, \{a, b\})$$



Linear Encoding of at-most-one: example

$$a \longrightarrow b \equiv a \text{ implies } b \equiv \neg a \vee b$$

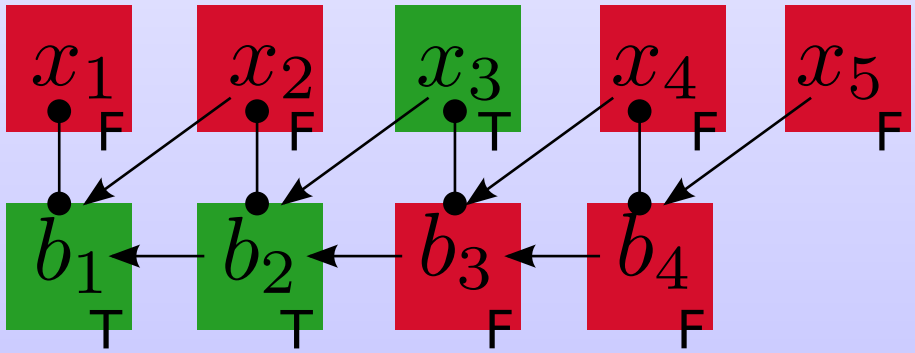
$$a \bullet\!\!-\!\!\bullet\, b \equiv \neg a \vee \neg b \equiv atmost(1, \{a, b\})$$



Linear Encoding of at-most-one: propagation

$$a \longrightarrow b \equiv a \text{ implies } b \equiv \neg a \vee b$$

$$a \bullet\!\!-\!\!\bullet\, b \equiv \neg a \vee \neg b \equiv atmost(1, \{a, b\})$$



Direct encoding: Encoding Constraints

- Let C be a CSP constraint in extension represented by a list of forbidden forbidden tuples $\{t_1, t_2, \ldots, t_n\}$
- For each forbidden tuple $t_i = ((X_1, v_1), (X_2, v_2), \ldots, (X_n, v_n))$, we add a clause that forbids this tuple

$$\neg b_{X_1,v_1} \vee \neg b_{X_2,v_2} \vee \ldots \vee \neg b_{X_n,v_n}$$

- When the contraint is represented by allowed tuples, we can either complement this set to obtain forbidden tuples (possible combinatorial explosion), or associate a Boolean variable to each allowed tuple which is true iff the variables in the constraint scope are assigned the values of the tuple, and add binary constraints to enforce assignment of variables consistent with this tuple when it is selected.
- UP propagation does not maintain AC on this encoding.

Support Encoding

- Same encoding of variables as in the direct encoding
- For a binary constraint C with scope X, Y, we encode the sets of support of each value by clauses which encode the following reasoning: whenever X is assigned v_i, Y must be assigned a value which is compatible with $X = v_i$.

$$\neg b_{X,v_i} \vee b_{Y,w_1} \vee \ldots \vee b_{Y,w_n}$$

  such that $\forall j$, $Y = w_j$ is a support of $X = v_i$.
- The same kind of clauses must be added for the sets of support of values of Y
- UP propagation maintains AC on this encoding.

Knowledge Compilation of Direct Encoding

- A knowledge compilation can be performed on the direct encoding to obtain the support encoding. Let s_i be the values of Y that are compatible with $X = v_i$ (supports) and c_i be the values of Y which are incompatible with $X = v_i$ (conflicts). The direct encoding generates the following clauses:

$$
\begin{aligned}
& b_{Y,c_1} \vee \ldots \vee b_{Y,c_n} \vee b_{Y,s_1} \vee \ldots b_{Y,s_m} \\
& \neg b_{Y,c_1} \vee \neg b_{X,v_i} \\
& \vdots \\
& \neg b_{Y,c_n} \vee \neg b_{X,v_i}
\end{aligned}
$$

  which by hyper resolution generates

$$\neg b_{X,v_i} \vee b_{Y,s_1} \vee \ldots b_{Y,s_m}$$

Knowledge Compilation of Direct Encoding

- This kind of resolvents can be added either by preprocessing (knowledge compilation) or even learned by a CDCL solver.
- This clause is only needed when $\neg b_{X,v_i}$ is a merge literal, that is, when there are at least 2 values conflicting with $X = v_i$
- These clauses are the ones generated by the support encoding.
- Hence, support encoding can be seen as a knowledge compilation of the direct encoding.

Log Encoding

- A CSP variable X is encoded with $\lceil \log_2 |dom(X)| \rceil$ Boolean variables. These Boolean variables encode the index of the value assigned to the variable.
- Forbidden tuples are encoded by a single clause which states that the indices of the conflicting values cannot be used simultaneously.
- Example: $X = 2$ and $Y = 1$ are incompatible is encoded as

$$(x_1, x_0) \neq (1, 0) \vee (y_1, y_0) \neq (0, 1))$$

In clausal form:

$$\neg x_1 \vee x_0 \vee y_1 \vee \neg y_0$$

Order Encoding

- A CSP variable X which takes integer values is encoded by Boolean variables $p_{x,i}$ which are true iff $X \leq i$.
- Example: $x \in \{2,3,4,5,6\}$ is encoded by

clause	meaning
$\neg p_{x,1}$	$X > 1$
$\neg p_{x,6}$	$x \leq 6$
$\neg p_{x,i-1} \vee p_{x,i}$ with $i \in \{2..6\}$	$X \leq i-1 \Rightarrow X \leq i$

- Constraints are encoded by considering conflict regions (a region is a cartesian product of subsets of the variables domains).
- Example: If every pair $(X, Y) \in \{2,3,4\} \times \{5,6,7,8\}$ is forbidden, we can write

$$X \notin \{2,3,4\} \vee Y \notin \{5,6,7,8\}$$

which is translated as

$$p_{x,1} \vee \neg px,4 \vee p_{x,4} \vee \neg p_{x,8}$$

Pseudo-Booleans (PB)

Linear Pseudo-Boolean (PB) Constraints

- A **linear** pseudo-Boolean constraint is defined over Boolean variables by

$$\sum_i a_i . l_i \triangleright d$$

with

  - $a_i, d \in \mathbb{Z}$
  - $l_i \in \{x_i, \bar{x}_i\}, x_i \in \mathbb{B}$
  - $\triangleright \in \{<, >, \leq, \geq, =\}$

- Examples:

$$3x_1 - 3x_2 + 2\bar{x}_3 + \bar{x}_4 + x_5 \geq 5$$

$$x_1 + x_2 + \bar{x}_4 < 3$$

- A pseudo-Boolean instance is a conjunction of PB constraints

Normalization

- Any constraint using operators $\{<, >, \leq, \geq, =\}$ can be rewritten as a conjunction of constraints using only $\geq$ where each coefficient is positive.
- Steps of the normalization:
  - Replace $=$ by a conjunction of $\leq$ and $\geq$
  - Multiply by -1 constraints with $<, \leq$
  - Replace $> d$ by $\geq d + 1$
  - Since $\overline{x} = 1 - x$, replace $-a.x$ (with $a > 0$) by $a.\overline{x} - a$

Cardinalities

Definition (atleast)

atleast$(k, \{x_1, x_2, \ldots, x_n\})$ is true if and only if at least k literals among $x_1, x_2, \ldots, x_n$ are true.

Definition (atmost)

atmost$(k, \{x_1, x_2, \ldots, x_n\})$ is true if and only if at most k literals among $x_1, x_2, \ldots, x_n$ are true.

$$atmost(k, \{x_1, x_2, \ldots, x_n\}) \equiv atleast(n - k, \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\})$$

Expressivity of linear constraints

- PB constraints generalize both clauses and cardinality constraints
  - clauses: $\geq$, all $a_i = 1$ and $d = 1$

    $a \vee b \vee c$ is represented by $a + b + c \geq 1$
  - cardinalities: $\geq$, all $a_i = 1$ and $d \geq 1$

    $atleast(2, \{a, b, c\})$ is represented by $a + b + c \geq 2$

    $atmost(2, \{a, b, c\})$ is represented by $\bar{a} + \bar{b} + \bar{c} \geq 1$
- PB constraints facilitates some encodings. As an example, $A + B = C$ where A, B are two integers of n bits can be encoded as one single constraint

$$\sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i = \sum_{i=0}^{n} 2^i c_i$$

- PB constraints are more expressive than clauses (one PB constraint may replace an exponential number of clauses)

Translation of $atleast(k, \{x_1, x_2, \ldots, x_n\})$ to SAT

Encoding without adding extra variables

- Express that no more than $n - k$ literals can be false, which means that as soon as $n - k + 1$ literals are selected, at least one of them must be true (a clause).
- So, $atleast(k, \{x_1, x_2, \ldots, x_n\})$ is equivalent to the conjunction of all possible clauses obtained by choosing $n - k + 1$ literals among $\{x_1, x_2, \ldots, x_n\}$.
- There are $\binom{n}{n-k+1}$ such clauses. The worst case is obtained when $k = n/2 - 1$ and since $\binom{n}{n/2} \geq 2^{n/2}$, this encoding is exponential.

Polynomial encodings exist but they require the introduction of extra variables.

Non-Linear Pseudo-Boolean Constraints

- A **non-linear** pseudo-Boolean constraint is defined over Boolean variables by

$$\sum_i a_i(\prod_j l_{i,j}) \triangleright d$$

with

  - $a_i, d \in \mathbb{Z}$
  - $l_i \in \{x_i, \bar{x}_i\}, x_i \in \mathbb{B}$
  - $\triangleright \in \{<, >, \leq, \geq, =\}$

- Example:

$$3x_1\bar{x}_2 - 3x_2x_4 + 2\bar{x}_3 + \bar{x}_4 + x_5x_6x_7 \geq 5$$

- A product is a AND

Expressivity of non-linear constraints

- More compact and natural encoding for several problems
- Example: factorization problem.
  Given an integer N, find P and Q such that $N = P.Q$ and $P > 1, Q > 1$.
  $N = P.Q$ is encoded as one single non-linear PB constraint: $\sum_i \sum_j 2^{i+j} p_i q_i = N$.
- Example: CNF
  Any CNF can be encoded as one single non-linear PB constraint.
  - a clause $C = a \vee b \vee c$ can be expressed as $P(C) = 1$ with $P(C) = a + b + c - a.b - a.c - b.c + a.b.c$
  - rewrite the whole CNF as $\sum_i P(C_i) = n$, with n the number of clauses in the CNF.

Some complexities

Complexity of the satisfiability problem in specific cases:

	Clauses	Linear PB constraints ($\geq$)	Non-linear PB constraints ($\geq$)
1 constraint	$O(1)$	$O(n)$	NP-complete
2 constraints	$O(n)$	NP-complete	NP-complete

Linearization

- A non-linear contraint can be easily translated into a linear pseudo-Boolean by introducing new variables and constraints s.t.

$$v \Leftrightarrow \prod_{k=1}^{n} l_k$$

This can be done

- with 2 PB constraints

$$v \vee \neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n$$
$$\sum_{k=1}^{n} l_k - nv \geq 0$$

- or n+1 clauses

$$v \vee \neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n$$
$$l_1 \vee \neg v$$
$$l_2 \vee \neg v$$
$$\vdots$$
$$l_n \vee \neg v$$

Decision and Optimization problems

- As in SAT, the decision problem consists in finding an assignment which satisfies all constraints.
- But most PB problems contain a cost function to optimize. For example,

$$\text{minimize } f = \sum_i c_i.x_i \text{ with } c_i \in \mathbb{Z}, x_i \in \mathbb{B}$$

- The optimization problem consists in finding an assignment which satisfies all constraints and has the least cost.

Example of an optimization instance

- Problem:

$$\begin{cases} \text{minimize} & 5x_1 + x_2 + 8x_3 + 2x_4 + 3x_5 \\ \text{subject to} & x_1 + \bar{x}_2 + x_3 \geq 1 \\ & \bar{x}_1 + x_2 + \bar{x}_3 + x_4 \geq 3 \\ & 2\bar{x}_1 + 4x_2 + 2x_3 + x_4 + 5x_5 \geq 5 \\ & 5x_1 + 4x_2 + 6x_3 + x_4 + 3x_5 \geq 10 \end{cases}$$

- Solution:

Optimum: 8

$x_1 = x_2 = x_4 = 1$

$x_3 = x_5 = 0$

Obvious rules

$$\frac{}{\overline{x} = 1 - x} \quad \text{(negation)}$$

$$\frac{}{x.x = x} \quad \text{(idempotence)}$$

$$\frac{}{\begin{array}{c} x \geq 0 \\ -x \geq -1 \end{array}} \quad \text{(bounds)}$$

Cutting Plane Proof System

$$\frac{\sum_j a_j L_j \geq b \quad \sum_j c_j L_j \geq d}{\sum_j (a_j + c_j) L_j \geq b + d} \qquad \text{(addition)}$$

$$\frac{\sum_j a_j L_j \geq b \quad \alpha > 0 \quad \alpha \in \mathbb{N}}{\sum_j \alpha a_j L_j \geq \alpha b} \qquad \text{(multiplication)}$$

$$\frac{\sum_j a_j L_j \geq b \quad \alpha > 0}{\sum_j \lceil \frac{a_j}{\alpha} \rceil L_j \geq \lceil \frac{b}{\alpha} \rceil} \qquad \text{(division)}$$

These three rules form a complete proof system,which means that whenever the constraints are unsatisfiable, these rules allow to derive the contradiction $0 \geq 1$.

Another system

Generalization of resolution:

$$\frac{\begin{array}{c}\sum_j a_j L_j \geq b \\ \sum_j c_j L_j \geq d \\ \alpha \in \mathbb{N}, \beta \in \mathbb{N} \\ \alpha > 0, \beta > 0\end{array}}{\sum_j (\alpha a_j + \beta c_j) L_j \geq \alpha b + \beta d}$$

(generalized resolution)

Truncate coefficients large than the right side term

$$\frac{\begin{array}{c}\sum_j a_j L_j \geq b \\ \forall j, a_j \geq 0 \\ a_k > b\end{array}}{b L_k + \sum_{j \neq k} a_j L_j \geq b}$$

(saturation)

Cutting Plane Stronger than Resolution

- Saturation corresponds to the merging rule in SAT. Hence, the cutting plane proof system can polynomially simulate the resolution proof system. This means that the cutting plane proof system is at least as powerful as the resolution proof system.
- With the Cutting Plane proof system, the Pigeon-Hole problem can be solved in polynomial time, in a way which is natural for a CDCL solver.
- Proofs of the Pigeon-Hole problem in the resolution system are exponential.
- Hence, the Cutting Plane system is stronger than the resolution proof system.

The Pigeon-Hole Example

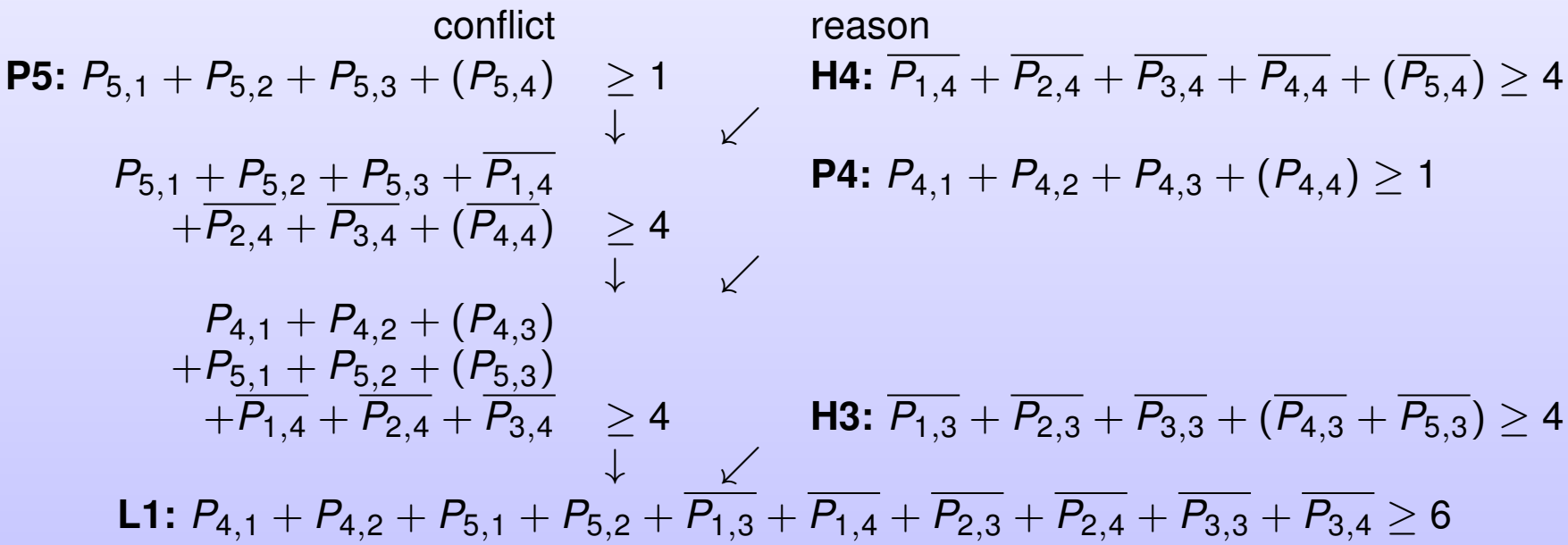
Let $P_{p,h}$ be a Boolean variable which is true if and only if pigeon number p is placed in hole number h. The problem of placing 5 pigeons in only 4 holes, with at most one pigeon per hole is encoded as:

$$P_{1,1} + P_{1,2} + P_{1,3} + P_{1,4} \geq 1 \quad \text{(P1)}$$
$$P_{2,1} + P_{2,2} + P_{2,3} + P_{2,4} \geq 1 \quad \text{(P2)}$$
$$P_{3,1} + P_{3,2} + P_{3,3} + P_{3,4} \geq 1 \quad \text{(P3)}$$
$$P_{4,1} + P_{4,2} + P_{4,3} + P_{4,4} \geq 1 \quad \text{(P4)}$$
$$P_{5,1} + P_{5,2} + P_{5,3} + P_{5,4} \geq 1 \quad \text{(P5)}$$
$$\overline{P_{1,1}} + \overline{P_{2,1}} + \overline{P_{3,1}} + \overline{P_{4,1}} + \overline{P_{5,1}} \geq 4 \quad \text{(H1)}$$
$$\overline{P_{1,2}} + \overline{P_{2,2}} + \overline{P_{3,2}} + \overline{P_{4,2}} + \overline{P_{5,2}} \geq 4 \quad \text{(H2)}$$
$$\overline{P_{1,3}} + \overline{P_{2,3}} + \overline{P_{3,3}} + \overline{P_{4,3}} + \overline{P_{5,3}} \geq 4 \quad \text{(H3)}$$
$$\overline{P_{1,4}} + \overline{P_{2,4}} + \overline{P_{3,4}} + \overline{P_{4,4}} + \overline{P_{5,4}} \geq 4 \quad \text{(H4)}$$

First conflict

Literal	val@lvl	reason
$P_{1,1}$	1@1	–
$P_{2,1}$	0@1	$H1$
$P_{3,1}$	0@1	$H1$
$P_{4,1}$	0@1	$H1$
$P_{5,1}$	0@1	$H1$
$P_{2,2}$	1@2	–
$P_{1,2}$	0@2	$H2$
$P_{3,2}$	0@2	$H2$
$P_{4,2}$	0@2	$H2$
$P_{5,2}$	0@2	$H2$
$P_{3,3}$	1@3	–
$P_{1,3}$	0@3	$H3$
$P_{2,3}$	0@3	$H3$
$P_{4,3}$	0@3	$H3$
$P_{5,3}$	0@3	$H3$
$P_{4,4}$	1@3	$P4$
$P_{1,4}$	0@3	$H4$
$P_{2,4}$	0@3	$H4$
$P_{3,4}$	0@3	$H4$
$P_{5,4}$	0@3	$H4$
$\square$		$P5$

First conflict analysis



Second conflict

Literal	val@lvl	reason
$P_{1,1}$	1@1	–
$P_{2,1}$	0@1	$H1$
$P_{3,1}$	0@1	$H1$
$P_{4,1}$	0@1	$H1$
$P_{5,1}$	0@1	$H1$
$P_{2,2}$	1@2	–
$P_{1,2}$	0@2	$H2$
$P_{3,2}$	0@2	$H2$
$P_{4,2}$	0@2	$H2$
$P_{5,2}$	0@2	$H2$
$P_{1,3}$	0@2	$L1$
$P_{1,4}$	0@2	$L1$
$P_{2,3}$	0@2	$L1$
$P_{2,4}$	0@2	$L1$
$P_{3,3}$	0@2	$L1$
$P_{3,4}$	0@2	$L1$
$\square$		$P3$

Second conflict analysis

conflict

P3: $P_{3,1} + P_{3,2} + (P_{3,3} + P_{3,4}) \geq 1$

reason

L1: $P_{4,1} + P_{4,2} + P_{5,1} + P_{5,2} + \overline{P_{1,3}} + \overline{P_{1,4}} + \overline{P_{2,3}} + \overline{P_{2,4}} + (\overline{P_{3,3}} + \overline{P_{3,4}}) \geq 6$

$\downarrow$ $\swarrow$

$P_{3,1} + (P_{3,2}) + P_{4,1} + (P_{4,2}) + P_{5,1} + (P_{5,2}) + \overline{P_{1,3}} + \overline{P_{1,4}} + \overline{P_{2,3}} + \overline{P_{2,4}} \geq 5$

H2: $\overline{P_{1,2}} + \overline{P_{2,2}} + (\overline{P_{3,2}} + \overline{P_{4,2}} + \overline{P_{5,2}}) \geq 4$

$\downarrow$ $\swarrow$

L2: $P_{3,1} + P_{4,1} + P_{5,1} + \overline{P_{1,2}} + \overline{P_{1,3}} + \overline{P_{1,4}} + \overline{P_{2,2}} + \overline{P_{2,3}} + \overline{P_{2,4}} \geq 6$

Third conflict

Literal	val@lvl	reason
$P_{1,1}$	1@1	–
$P_{2,1}$	0@1	$H1$
$P_{3,1}$	0@1	$H1$
$P_{4,1}$	0@1	$H1$
$P_{5,1}$	0@1	$H1$
$P_{1,2}$	0@1	$L2$
$P_{1,3}$	0@1	$L2$
$P_{1,4}$	0@1	$L2$
$P_{2,2}$	0@1	$L2$
$P_{2,3}$	0@1	$L2$
$P_{2,4}$	0@1	$L2$
$\square$		$P2$

Third conflict analysis

conflict

P2: $P_{2,1} + (P_{2,2} + P_{2,3} + P_{2,4}) \geq 1$

$\downarrow \quad \swarrow$

$(P_{2,1} + P_{3,1} + P_{4,1} + P_{5,1}) + \overline{P_{1,2}} + \overline{P_{1,3}} + \overline{P_{1,4}} \geq 4$

$\downarrow \quad \swarrow$

L3: $\overline{P_{1,1}} + \overline{P_{1,2}} + \overline{P_{1,3}} + \overline{P_{1,4}} \geq 4$

reason

L2: $P_{3,1} + P_{4,1} + P_{5,1} + \overline{P_{1,2}} + \overline{P_{1,3}} + \overline{P_{1,4}} + (\overline{P_{2,2}} + \overline{P_{2,3}} + \overline{P_{2,4}}) \geq 6$

H1: $\overline{P_{1,1}} + (\overline{P_{2,1}} + \overline{P_{3,1}} + \overline{P_{4,1}} + \overline{P_{5,1}}) \geq 4$

Final step

Literal	val@lvl	reason
$P_{1,1}$	0@0	$L3$
$P_{1,2}$	0@0	$L3$
$P_{1,3}$	0@0	$L3$
$P_{1,4}$	0@0	$L3$
$\square$		$P1$

PB propagation: slack based

- The slack of a constraint $\sum_i a_i.l_i \geq b$ is defined as

$$s = \sum_{l_i \text{not false}} a_i - b$$

- It represents the maximal amount by which the left side may exceed the right side of the constraint when all yet unassigned variables are set to true.
- If $s < 0$, the constraint is falsified.
- If $s - a_i < 0$ then l_i must be set to true (propagation).
- Several literals can be propagated by a constraint! (example: $4a + b + c + d + e \geq 4$ when a becomes false)
- Propagation can be performed by incrementally maintaining the slack. This corresponds to the counter scheme in SAT.

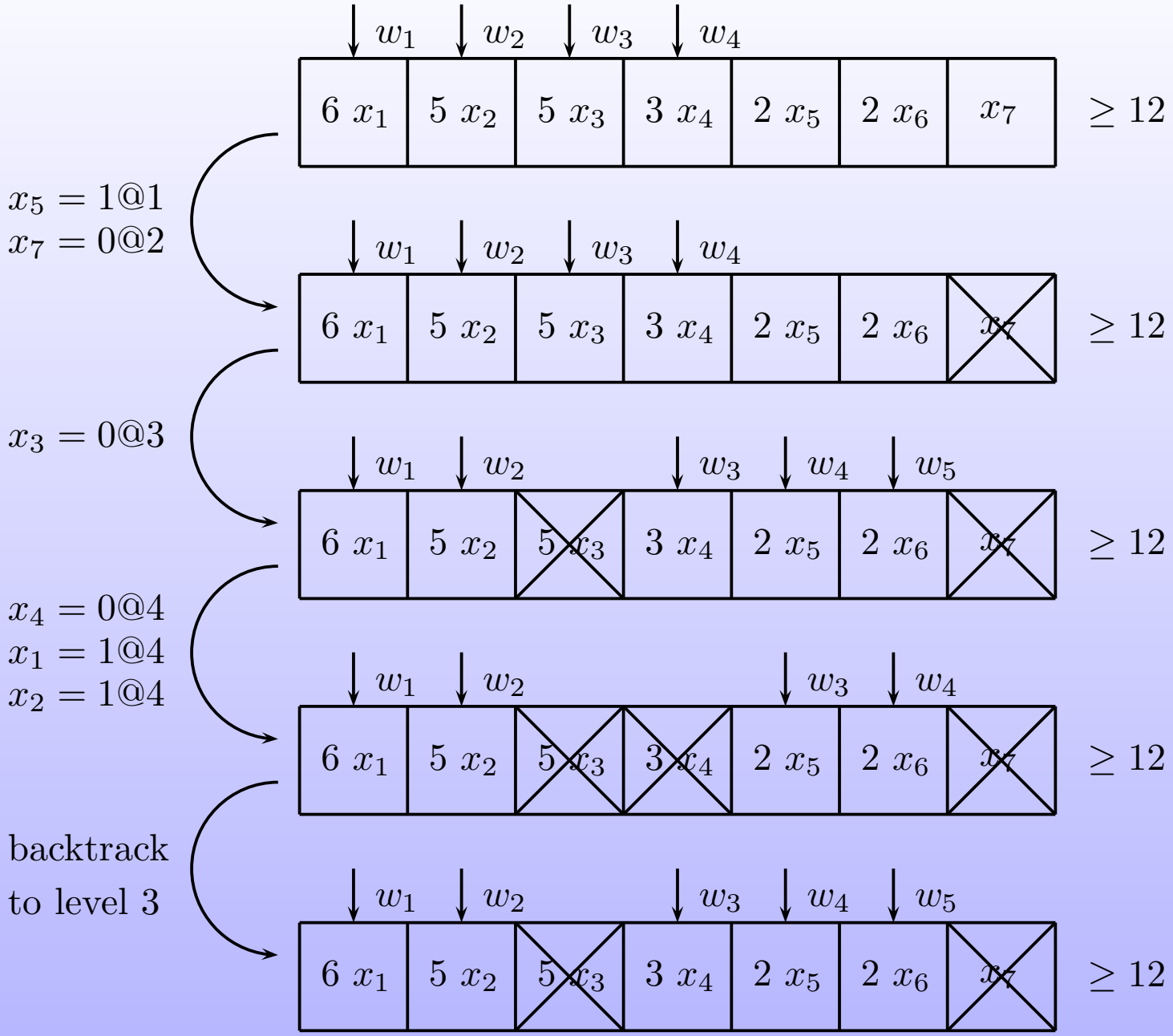
PB propagation: watched literals

- Let W be the set of watched literals (which may not be false) in the constraint and

$$S(W) = \sum_{l_i \in W} a_i$$

- Let a_{max} be the largest coefficient of an unassigned literal in the constraint.
- As long as $S(W) \geq b + a_{max}$, no propagation can occur because the watched literals are sufficient to satisfy the constraint even when l_{max} is set to false.
- When this condition is not met, new watched literals must be found in order to satisfy it, otherwise propagation occurs.
- The number of watched literals can greatly vary during search!

PB propagation: watched literals illustration



CDCL on PB constraints

The general approach is the same as in SAT. However:

- A constraint may propagate more than one literal and hence appear as reason several times.
- "Resolution" may generate a constraint which is not falsified and hence cannot become an asserting constraint. This occurs because reasons may be oversatisfied. To avoid this, careful reductions must be performed (removing literals which are not relevant).

Example of problem in conflict analysis

- Consider the two constraints below:

$$C_1 : 3x_1 + 2x_2 + x_3 + 2\overline{x_4} \geq 3$$
$$C_2 : 3\overline{x_1} + x_5 + x_6 + x_7 \geq 3$$

- With $\{x_2 = 0, x_4 = 1, x_5 = 0\}$, C_1 propagates $x_1 = 1$ and then C_2 is falsified.
- "Resolving" upon x_1 gives

$$2x_2 + x_3 + 2\overline{x_4} + x_5 + x_6 + x_7 \geq 3$$

which is satisfiable!

Comments

- Propagation and conflict analysis is harder in PB and coefficients in the learned constraints keep growing (arbitrary precision numbers are needed!)
- Therefore, it is sometimes more efficient to learn clauses rather than PB constraints. But of course, clauses are less powerful than PB constraints (allow less inferences).

PB Constraints stronger than clauses?

Several facts suggest that PB constraints are more expressive than clauses

- PB constraints use basic arithmetic
- many problems are more easily encoded as PB (e.g. adder)
- NP-complete problems can be encoded as one single PB constraint (e.g. variants of Knapsack)
- any CNF can be encoded as one single non-linear PB constraint
- without additional variables, encoding a PB constraint into CNF is exponential
- the Pigeon-Hole problem can be solved polynomially when encoded as PB constraints (with learning).

but is it so sure?

Encoding PB constraints into CNF

1. more or less specialized “direct encoding”
   - doesn’t introduce additional variables
   - exponential
2. BDD (Binary Decision Diagrams)
   - requires additional variables
   - exponential
3. Adder+Comparator (Warner’s encoding)
   - requires additional variables
   - polynomial
4. and a few other encodings

Generalized Arc Consistency (GAC)

- Let C be a PB constraint, l and l_i be literals
- Whenever $C \wedge \{l_1, l_2, \ldots l_n\} \models l$, we expect that l will be generated by the inference process
- When this is the case for any set of literals, the inference process is said to maintain Generalized Arc Consistency (GAC).
- Basic PB inference rules maintain GAC.
- Depending on the encoding into CNF, Unit Propagation (UP) may or may not maintain GAC

Example

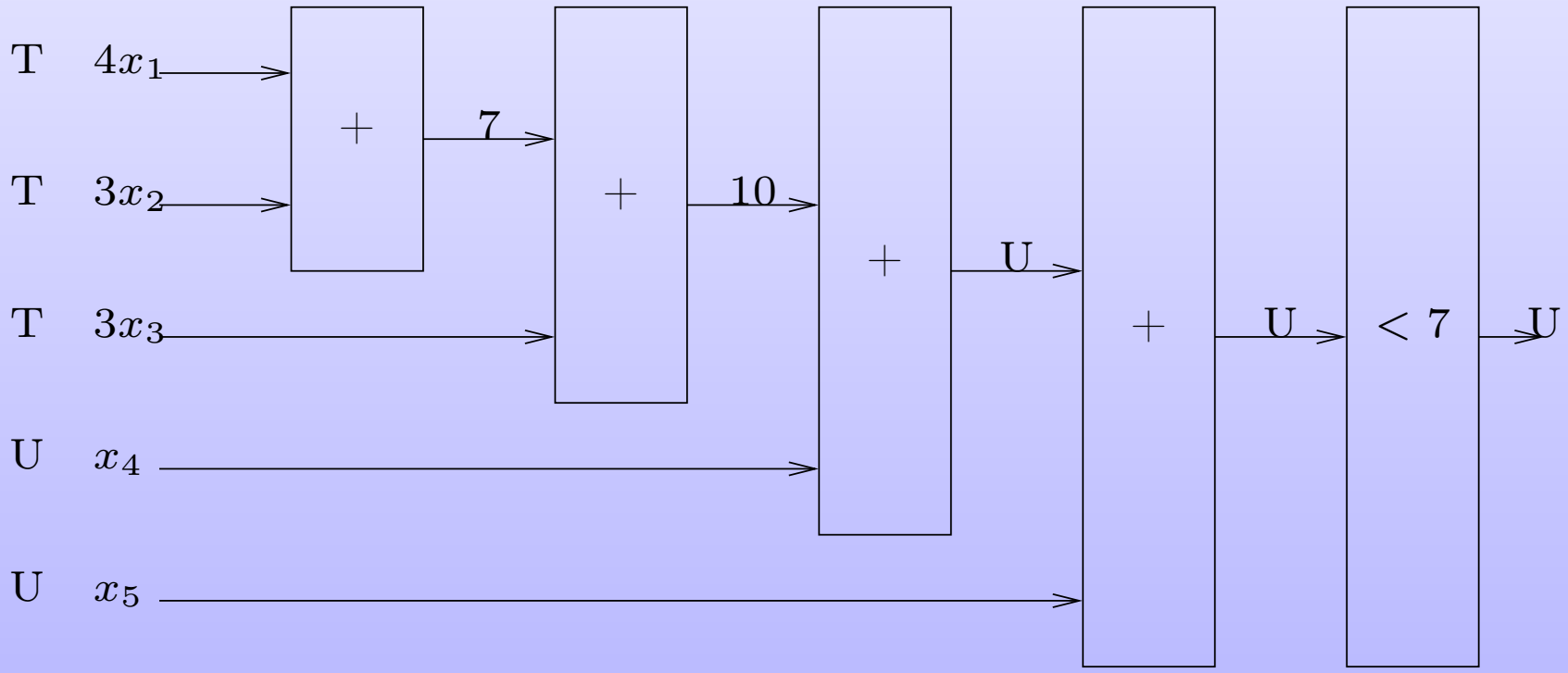
- $4x_1 + 3x_2 + 3x_3 + x_4 + x_5 < 7$
- As soon as x_1 becomes true, x_2 and x_3 must be set to false (otherwise the constraint will be falsified)

Encoding PB constraints into CNF

1. more or less specialized “direct encoding”
   - doesn’t introduce additional variables
   - **exponential**
   - **UP generally maintains GAC**
2. BDD (Binary Decision Diagrams)
   - requires additional variables
   - **exponential**
   - **UP maintains GAC**
3. Adder+Comparator (Warner’s encoding)
   - requires additional variables
   - **polynomial**
   - **UP does not maintain GAC**

The trouble with UP on the adder encoding

- $4x_1 + 3x_2 + 3x_3 + x_4 + x_5 < 7$
- $x_1 = T$, all other variables are unknown (U), UP doesn't infer anything
- $x_1 = x_2 = x_3 = T$, all other variables are unknown (U), UP doesn't even detect inconsistency



The big question:

Does there exist an encoding which is both

- polynomial
- and such that UP maintains GAC ?

Mainly a theoretical question (such an encoding may not be efficient in practice).

But the existence of this encoding would narrow the gap between PB constraints and clauses.

And the answer is...

YES!

It does exist, and it is rather easy. Sketch:

1. normalize constraints to use only $<$
2. for each literal l_j, transform the constraint $\sum_i a_i.l_i < d$ into "watchdogs" $\sum_{i \neq j} a_i.l_i \geq d - a_j \implies \neg l_j$

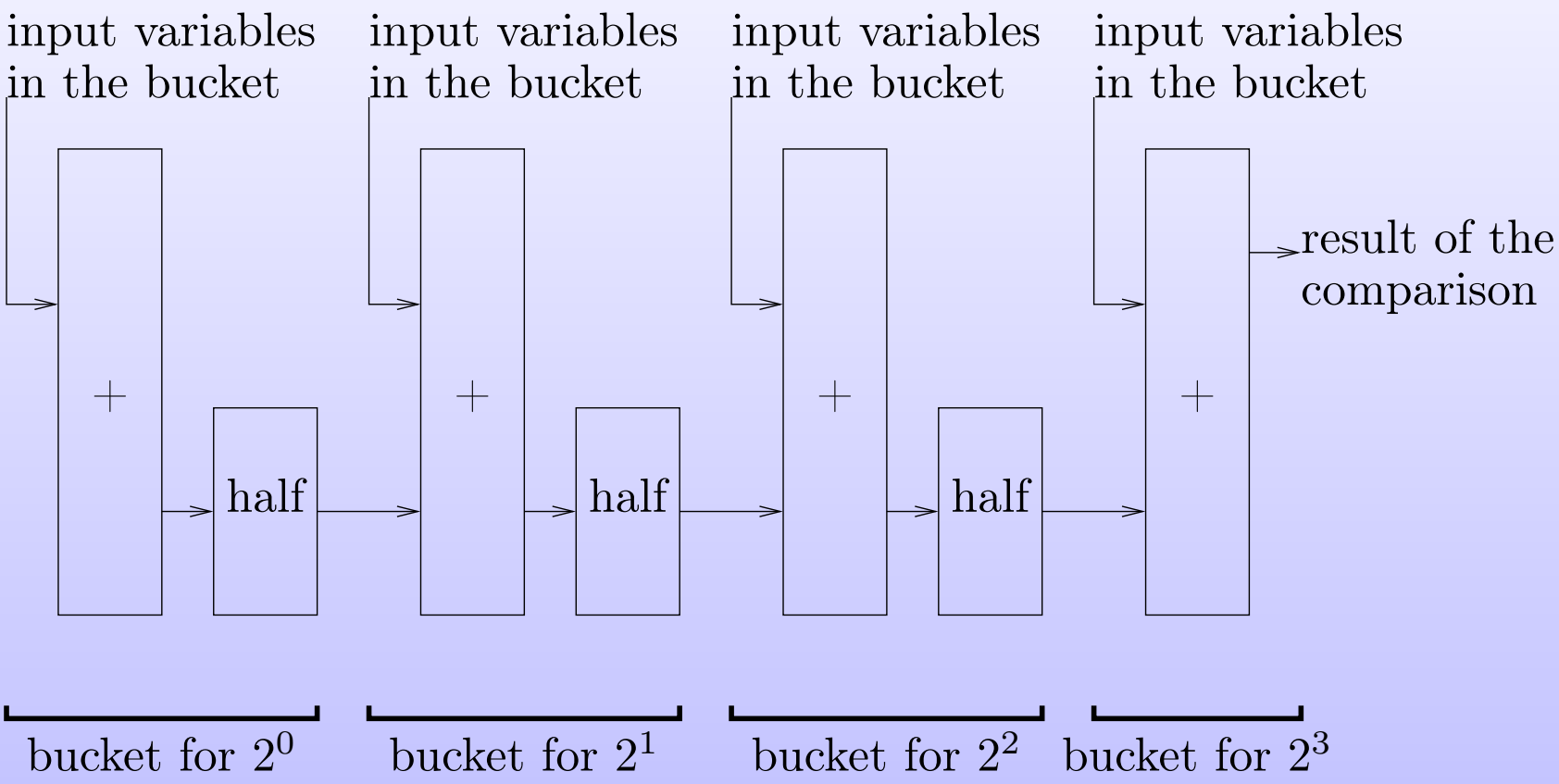
4. decompose each coefficient a_i into binary
5. for each power of two occurring in the binary decomposition, use a unary encoding to sum the variables having a coefficient with this bit set to 1
6. compute the half of each sum and use it as a carry for the next stage
7. compare the result of the final stage with $d - a_j$ (**UP doesn't work well on this)**

YES!

It does exist, and it is rather easy. Sketch:

1. normalize constraints to use only $<$
2. for each literal l_j, transform the constraint $\sum_i a_i.l_i < d$ into "watchdogs" $\sum_{i \neq j} a_i.l_i \geq d - a_j \implies \neg l_j$
3. **add an offset so that the new right term d' is a multiple of the power of 2 corresponding to the last stage**
4. decompose each coefficient a_i into binary
5. for each power of two occurring in the binary decomposition, use a unary encoding to sum the variables having a coefficient with this bit set to 1
6. compute the half of each sum and use it as a carry for the next stage
7. **compare the result of the final stage with d' (one single bit to compare, this is the trick!)**

The big picture



Rewrite coefficient in binary

$10x_1 + 7x_2 + 3x_3$ is rewritten as

- $(8 + 2)x_1 + (4 + 2 + 1)x_2 + (2 + 1)x_3$

Rewrite coefficient in binary

$10x_1 + 7x_2 + 3x_3$ is rewritten as

- $(8 + 2)x_1 + (4 + 2 + 1)x_2 + (2 + 1)x_3$
- and then as
  $(x_1).2^3 + (x_2).2^2 + (x_1 + x_2 + x_3).2^1 + (x_2 + x_3).2^0$

Rewrite coefficient in binary

$10x_1 + 7x_2 + 3x_3$ is rewritten as

- $(8 + 2)x_1 + (4 + 2 + 1)x_2 + (2 + 1)x_3$
- and then as
  $(x_1).2^3 + (x_2).2^2 + (x_1 + x_2 + x_3).2^1 + (x_2 + x_3).2^0$
- input variables multiplied by a given power of 2 form a bucket
- variables in a bucket are added and represented in unary
- the half operator reports a carry from one bucket (for 2^i) to the next one (for 2^{i+1})
- the last bucket represents the sum (almost)

Unary representation

n bits encode an integer x between 0 and n

- $X = 0$

0	0	0	0	0
x_1	x_2	x_3	x_4	x_5

- $X = 1$

1	0	0	0	0
x_1	x_2	x_3	x_4	x_5

- $X = 2$

1	1	0	0	0
x_1	x_2	x_3	x_4	x_5

- $X = 5$

1	1	1	1	1
x_1	x_2	x_3	x_4	x_5

Convention

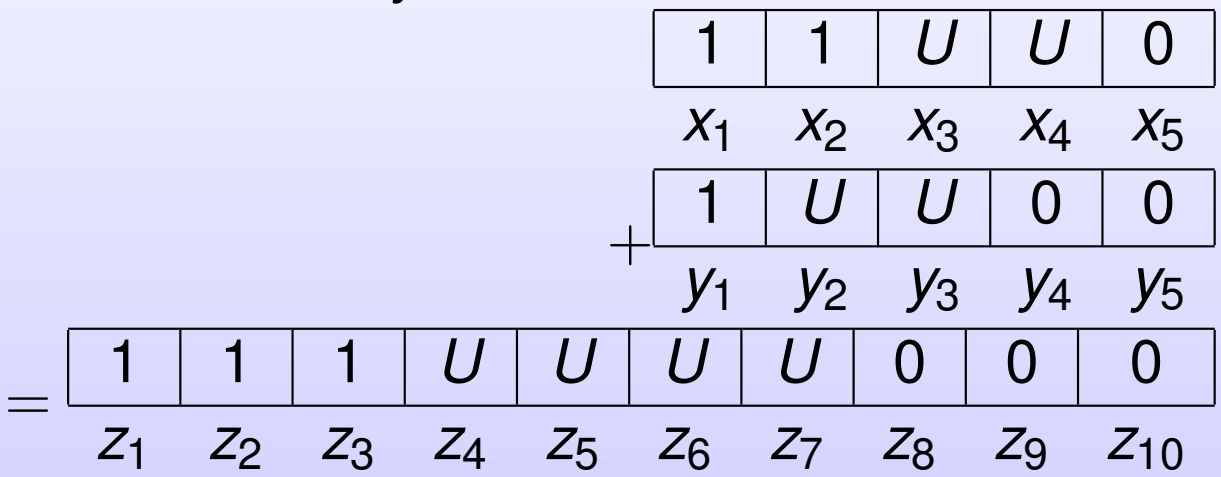
- 1s must be at the beginning, 0s must be at the end
- so, the unary representation

1	1	U	U	0
x_1	x_2	x_3	x_4	x_5

  encodes a number which is ≥ 2 and ≤ 4
- any input vector of Booleans can be converted to this representation by a cascade of unary adders/sorters. UP actually does the conversion (sorts the bits).

Unary adder (Totalizer)

- sum of two numbers in unary notation $X + Y = Z$



- can be encoded with simple clauses $x_i \wedge y_j \implies z_{i+j}$ (meaning if $X \geq i$ and $Y \geq j$ then $Z \geq i + j$) to deal with 1s. Same principle with 0s but we don't need them in our context.
- or by using sorting networks (asymptotically more efficient)

Half operator

- Retaining bits of even indices in the unary representation of X gives the unary representation of $\lfloor X/2 \rfloor$
- Example

1	1	1	1	0	0
x_1	x_2	x_3	x_4	x_5	x_6

1	1	0
x_2	x_4	x_6

Comparator and offset

- Unit Propagation doesn't work well on a usual comparator
- Solution: add the same constant to both side of the constraint so that the right term becomes a multiple of 2^{max} (the weight of the last bucket)
- this only adds a constant term to the buckets (easy)
- but most importantly it makes the comparator trivial (the result of the comparator is just one output bit of the last adder)

Properties of this encoding

- Unit Propagation maintains GAC
- $O(n^2 \log(n) \log(a_{max}))$ variables
- $O(n^3 \log(n) \log(a_{max}))$ clauses

Weighted CSP (WCSP)

WCSP

- A WCSP instance is defined by a 4-tuple $< X, D, C, k >$ such that
  - X is the set of variables
  - D is the set of domains of variables
  - C is the set of constraints
  - k is the forbidden cost (a positive integer or ∞). Any instantiation with a cost greater or equal to k is unacceptable.

Notations

- $t[X]$ is the value of X in tuple t
- $l(S)$ (labeling) is the set of all tuples than can be built on scope S
- n is the number of variables
- e is the number of constraints
- d is the maximum domain size
- r is the greatest arity of a constraint
- k is the forbidden cost

Bounded addition

- k plays the role of ∞. To add or subtract costs, we use the following operators:
- $a \oplus b = max(a + b, k)$
- $a \ominus b = \begin{cases} a - b & \text{if } a \neq k \wedge b \neq k \\ k & \text{if } a = k \wedge b \neq k \\ -k & \text{if } a \neq k \wedge b = k \\ undef & \text{if } a = k \wedge b = k \end{cases}$

Constraints

- A constraint C_i defined on a scope $(X_1, \ldots, X_n)$ maps each tuple of values to a cost in $[0..k]$. A cost of k means that the tuple if strictly forbidden, a cost of 0 means that the tuple is completely satisfying.
- The cost of an instantiation I is the sum of the constraints costs

$$cost(I) = \bigoplus_i C_i(I)$$

- The WCSP problem consists in finding an instantiation with a minimal cost which must be smaller than k (optimization problem)

Special constraints

- Costs of values
  - Values of variables can be assigned individual costs.
  - For simplicity, this is modelled by constraints of arity 1.
- Constant cost
  - The cost of an instantiation may contain a constant term that does not depend on the instantiation.
  - For simplicity, this cost is associated to a single constaint with an empty scope named $C_\varnothing$.
  - Every instantiation has a cost at least equal to $cost(C_\varnothing)$.
  - This constant cost plays an important role in Soft Arc Consistencies.

Branch and Bound

- UB: upper bound, cost of the best solution found so far
- LB: lower bound, minimum cost of any instantiation that extends the current partial instantiation considered in the search.
  LB is obtained by inference.
  As soon as LB$\geq$UB, the branch can be pruned (any complete instantiation will have a cost at least equal to UB and hence cannot be preferred to the current best solution)
- Whenever a solution is found with a cost c, k can be set to c in order to prune instantiation with a cost greater or equal to c.

Equivalence Preserving Transformation (EPT)

- An Equivalence Preserving Transformation (EPT) transforms a WCSP instance into an equivalent instance.
- Two instances are equivalent if they maps any instantiation to the same cost.
- The basic EPT are:
  - UnaryProject
  - Project

Unary Project

Transfer a cost α from a unary constraint C_X on X to $C_\varnothing$. Each value of X must have a cost $\geq \alpha$

Algorithm 1 Unary Project

Require: $0 \leq \alpha \leq min(C_X(v_i)|v_i \in dom(X))$

```
1: function UNARYPROJECT(X: variable, α: cost)
2:     C_∅ ← C_∅ + α
3:     for all v_i ∈ dom(X) do
4:         C_x(v_i) ← C_x(v_i) − α
5:     end for
6: end function
```

Unary Project Example

- Each value of X has at least a cost of 3

$C_\varnothing$
0

C_X	
X	*cost*
a	4
b	3
c	5
d	7

- Hence a cost of 3 can be factored to $C_\varnothing$

$C_\varnothing$
$0 + 3 = 3$

C_X	
X	*cost*
a	$4 - 3 = 1$
b	$3 - 3 = 0$
c	$5 - 3 = 2$
d	$7 - 3 = 4$

Project

Transfer a cost α between a value v of X and the tuples of a constraint C_i where X equals v. If $\alpha > 0$, cost is moved from the constraint to the value. If $\alpha < 0$, cost is moved from the value to the constraint.

Algorithm 2 Project

Require: $|scope(C_i)| \geq 2$
Require: $X \in scope(C_i)$
Require: $-C_X(v) \leq \alpha \leq min(\{C_i(t)|t[X] = v\})$

```
1: function PROJECT(C_i, X, v, α)
2:     C_X(v) ← C_X(v) + α
3:     for all tuple t built on scope(C_i) s.t. t[X] = v do
4:         C_i(t) ← C_i(t) − α
5:     end for
6: end function
```

Project Example

- Initial constraints

C_X	
X	*cost*
a	0
b	1

C_{XY}		
X	Y	*cost*
a	c	3
a	d	0
b	c	0
b	d	0

C_Y	
Y	*cost*
c	0
d	1

$C_\varnothing$
0

- Project a cost of 1 from $X = b$ to the tuples of C_{XY} where $X = b$

C_X	
X	*cost*
a	0
b	$1 - 1 = 0$

C_{XY}		
X	Y	*cost*
a	c	3
a	d	0
b	c	$0 + 1 = 1$
b	d	$0 + 1 = 1$

C_Y	
Y	*cost*
c	0
d	1

$C_\varnothing$
0

Project Example (2)

- Project a cost of 1 from tuples of C_{XY} where $Y = c$ to $Y = c$

C_X	
X	*cost*
a	0
b	0

C_{XY}		
X	Y	*cost*
a	c	$3 - 1 = 2$
a	d	0
b	c	$1 - 1 = 0$
b	d	1

C_Y	
Y	*cost*
c	$0 + 1 = 1$
d	1

$C_\varnothing$
0

- Unary project from Y

C_X	
X	*cost*
a	0
b	0

C_{XY}		
X	Y	*cost*
a	c	2
a	d	0
b	c	0
b	d	1

C_Y	
Y	*cost*
c	$1 - 1 = 0$
d	$1 - 1 = 0$

$C_\varnothing$
$0 + 1 = 1$

EPTs are not confluent

C_X	
X	*cost*
a	0
b	0

C_{XY}		
X	Y	*cost*
a	c	1
a	d	1
b	c	1
b	d	0

C_Y	
Y	*cost*
c	0
d	0

can give

C_X	
X	c
a	1
b	0

C_{XY}		
X	Y	c
a	c	0
a	d	0
b	c	1
b	d	0

C_Y	
Y	c
c	0
d	0

or

C_X	
X	c
a	0
b	0

C_{XY}		
X	Y	c
a	c	0
a	d	1
b	c	0
b	d	0

C_Y	
Y	c
c	1
d	0

Node Consistency

Definition (NC)

A WCSP is Node Consistent (NC) if for any variable X

- $\forall v \in dom(X) C_\varnothing \oplus C_X(v) < k$
- $\exists v \in dom(X) C_X(v) = 0$

(Soft) Arc Consistency

Definition (AC)

A WCSP is (Soft) Arc Consistent if for any constaint C_S such that $S = scope(S)$ and $|S| \geq 2$

- $\forall t \in I(S), C_s(t) = k$ if $C_\varnothing \oplus C_S(t) \oplus \bigoplus_{X \in S} C_X(t[X]) = k$
- $\forall X \in S, \forall v \in dom(X), \exists t \in I(S)$ such that $t[X] = v$ and $C_S(t) = 0$

$I(S)$ (labeling) represents the set of all tuples that can be built on scope S

Directional Arc Consistency (DAC)

Definition (DAC)

A binary WCSP is Directional Arc Consistent (DAC) for a given ordering of variables if $\forall C_{XY}$ such that $X < Y$,
$\forall v \in dom(X), \exists w \in dom(Y)$ such that
$C_{XY}(X = v, Y = w) = C_Y(w) = 0$

Directional arc consistency can be established in $O(ed^2)$ time.

Full Directional Arc Consistency (FDAC)

Definition (FDAC)

A binary WCSP is full directional arc consistent (FDAC) with respect to an order $<$ on the variables if it is arc consistent and directional arc consistent with respect to $<$.

Full directional arc consistency can be established in $O(end^3)$.

Existential Arc Consistent (EAC)

Definition (EAC)

A binary WCSP is existential arc consistent (EAC) if it is node consistent and if $\forall X, \exists v \in dom(X)$, such that $C_X(v) = 0$ and for all constraint C_{XY}, $\exists w \in dom(Y)$ such that $C_{XY}(X = v, Y = w) = C_Y(w) = 0$.
Value v is called the EAC support value of variable X.

Existential Directional Arc Consistent (EDAC)

Definition (EDAC)

A binary WCSP is existential directional arc consistent (EDAC) with respect to an order $<$ on the variables if it is existential arc consistent (EAC) and full directional arc consistent (FDAC) with respect to $<$.

EDAC can be established in $O(ed^2 max(nd, k))$

Virtual Arc Consistency (VAC)

Definition (*Bool*(P))

Let P be a WCSP instance. *Bool*(P) is a CSP instance built by copying P (same variables, domains and constraints scopes) and only allowing tuples which a cost of 0 in P. Any tuple of P with a cost > 0 is forbidden in *Bool*(P).

If *Bool*(P) has a solution, then P has a solution of cost $C_\varnothing$

Definition (VAC)

A WCSP instance P is Virtual Arc Consistent (VAC) if *Bool*(P) is AC.

VAC

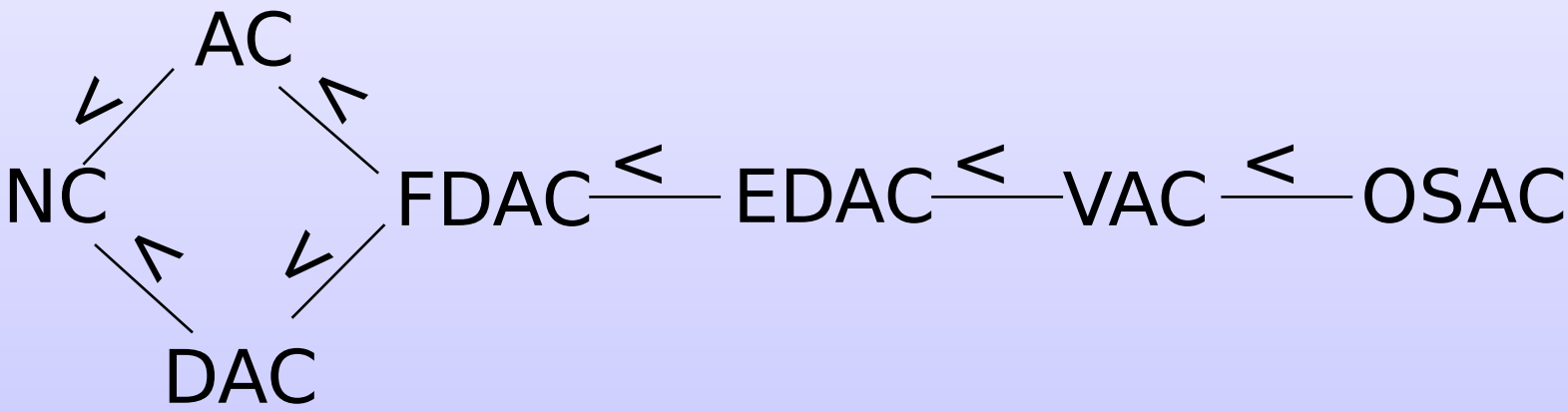
- When a domain wipe-out occurs in enforcing AC on $Bool(P)$, it can be traced to identify cost transfers in P than will increase $C_\varnothing$. This process is similar to conflict analysis.
- However, costs may be used in more than one direction and hence have to be split. Therefore, cost transfers become fractional.
- Cost transfers may become smaller and smaller at each iteration. Therefore, the number of iteration is not bounded.
- One can decide to abort the algorithm when cost transfers become smaller than ϵ for a given number of iterations (algorithm VACϵ). The instance produced by VACϵ is not necessarily VAC.
- VACϵ can be enforced in $O(ed^2k/\epsilon)$

Optimal Soft Arc Consistency (OSAC)

- The idea is to translate every possible EPT (UnaryProject and Project performed simultaneously) into inequations and choose the values of cost transfers (α) in order to maximize the increase of $C_\varnothing$.
- We obtain a linear program which can be solved in polynomial time if we allow costs to be real (instead of integers).
- Short description of the constraints in the linear program:
  - maximize sum of transfers from variable values to $C_\varnothing$
  - for each variable value:
    initial cost-transfer to $C_\varnothing$+sum of transfers from constraints≥ 0
  - for each tuple t in a constraint:
    initial cost - sum of transfers to $t[X]$ for any X in the scope≥ 0

- The resulting costs are real.
- The linear program has $O(edr + n)$ variables and $O(ed^r + nd)$ constraints
- In practice, only useful for pre-processing.

Comparison of WCSP consistencies



References

Main references to start with:

- “Handbook of Satisfiability” edited by Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh, IOS Press
- “Handbook of Constraint Programming” edited by F. Rossi, P. van Beek and T. Walsh, Elsevier

Many other references:

- to be detailed in the final version...