Formale Methoden der Informatik

Block 2: Satisfiability Problems

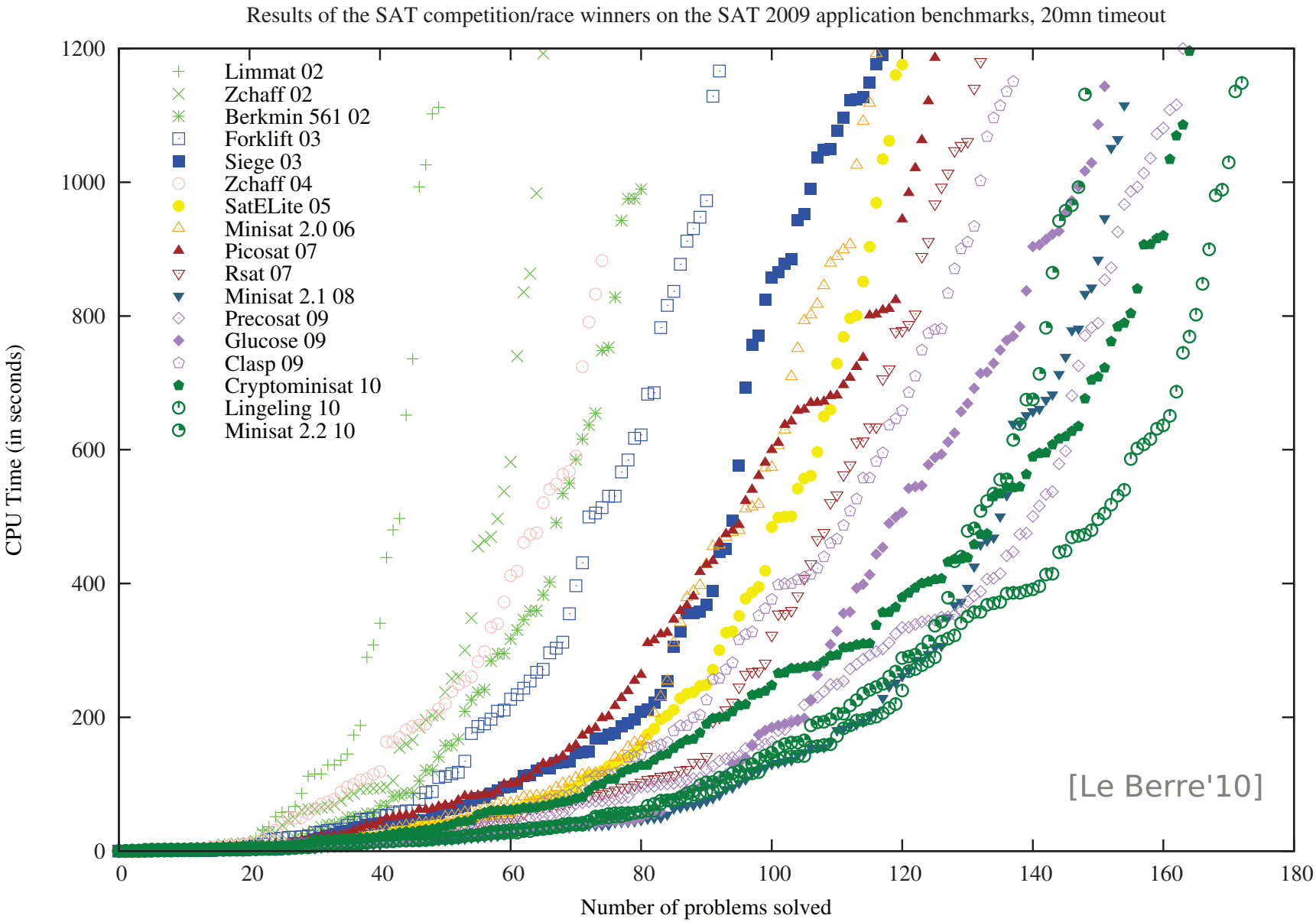
2. Techniques for Modern SAT Solvers

Uwe Egly

Knowledge-Based Systems Group
Institute of Information Systems
Vienna University of Technology

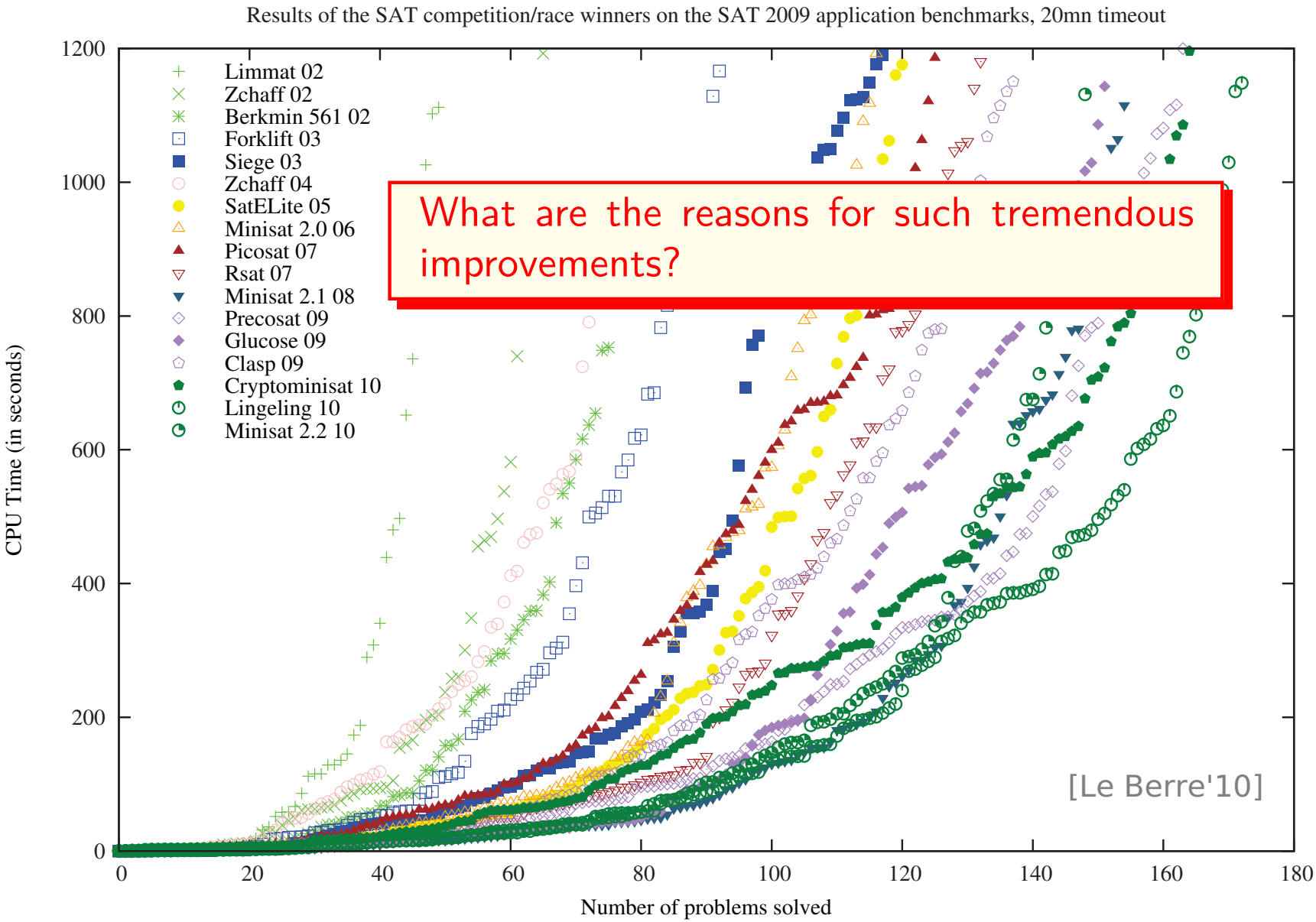
Results of the SAT 2009 application benchmarks

for leading solvers from 2002 to 2010



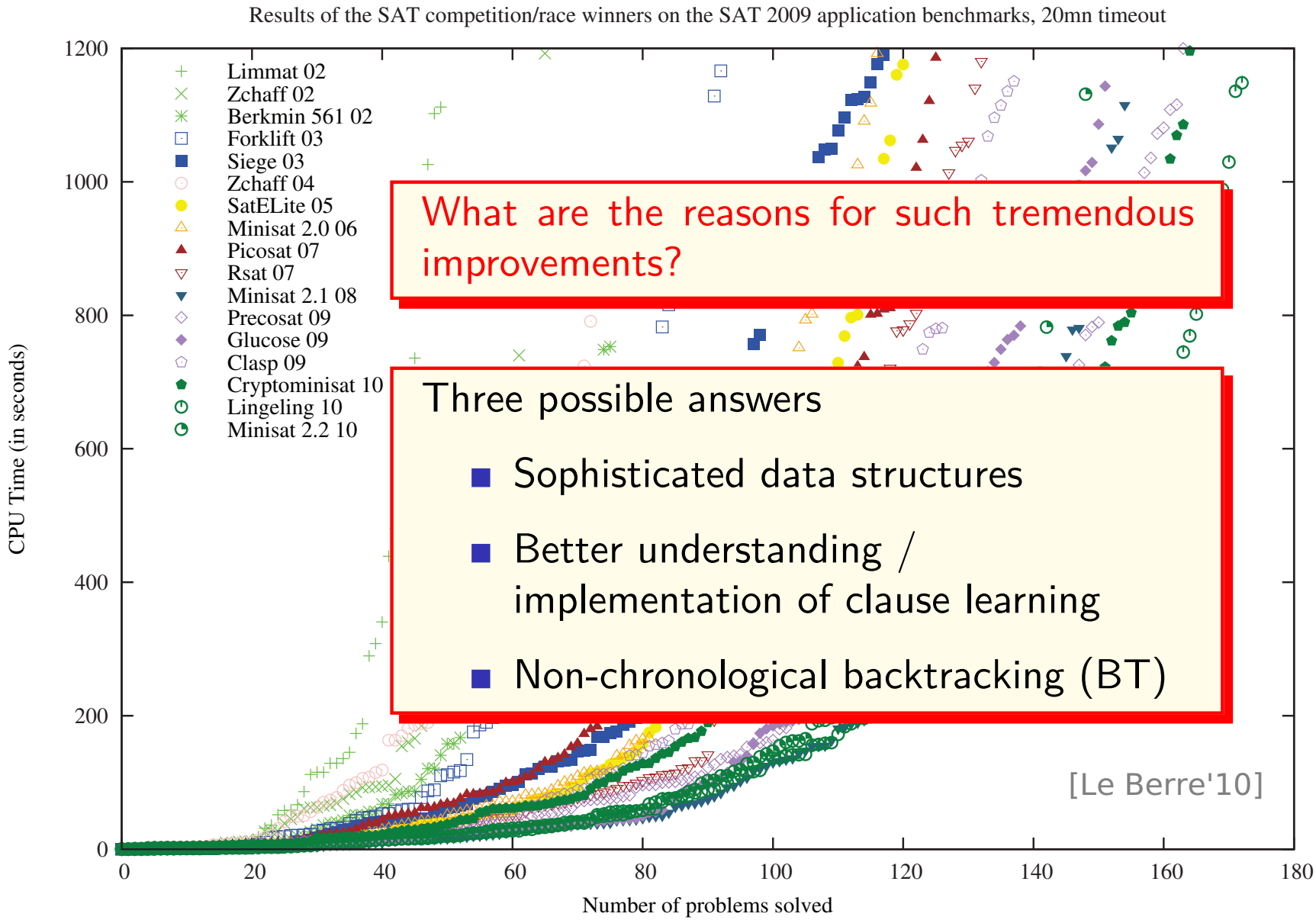
Results of the SAT 2009 application benchmarks

for leading solvers from 2002 to 2010



Results of the SAT 2009 application benchmarks

for leading solvers from 2002 to 2010



The goal and program for this lecture

Goal

The goal of this lecture is to understand how a modern SAT solver based on DLL works and why the performance for many practically relevant application-oriented examples is so high.

Program

Start with a basic "traditional" method and refine it stepwisely towards its modern version. We will focus on a data structure called implication graph which supports clause learning. We discuss how learned clauses can prune the search space resulting in an improved performance and how learned clauses influence backtracking.

Outline

The method of Davis, Logeman and Loveland

- M. Davis, G. Logeman, D. Loveland *A machine program for theorem proving*, CACM, Vol. 5, No. 7, pp. 394-397, 1962 (A copy is available in TUWEL)
- Basic framework for all modern SAT solvers (require additional features like clause learning and non-chronological backtracking to be competitive)
- The procedure, nowadays called DLL, is search-based
- Basic Idea: Given a clause set $\mathcal{C}$
  - Try all (partial) assignments and test whether $\mathcal{C}$ is sat
  - Stop enlarging the current partial assignment I when there is a conflict (i.e., $\mathcal{C}$ cannot be satisfied with extensions of I)
  - Stop if I satisfying $\mathcal{C}$ is found or all possible I have been tried

The basic DLL procedure: An example

$a' \vee b \vee c$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d$
$a' \vee b \vee c'$
$a' \vee b' \vee c$

p' means $\neg p$

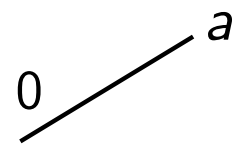
The basic DLL procedure: An example

a

$a' \vee b \vee c$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d$
$a' \vee b \vee c'$
$a' \vee b' \vee c$

The basic DLL procedure: An example

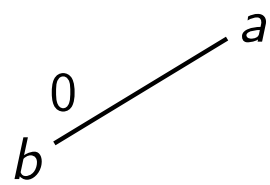
$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$



A decision (at level 1): choose var and value

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

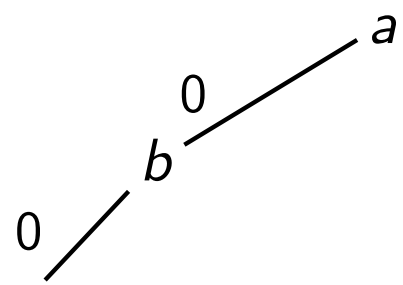


$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d \quad \leftarrow$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$

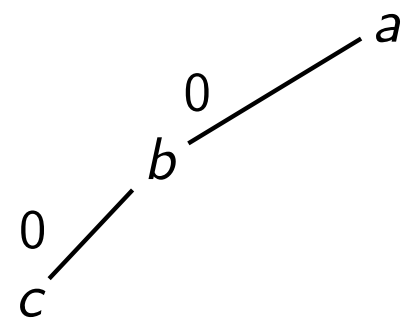


Another decision now at level 2

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

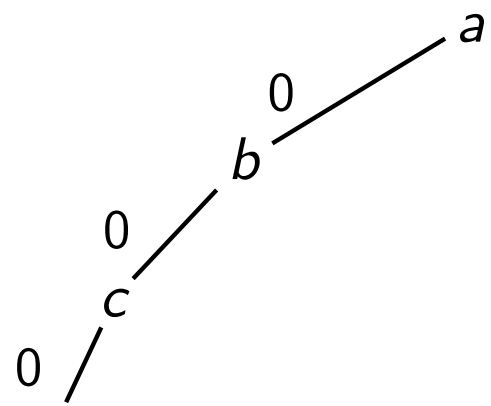
$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d \quad \leftarrow$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$



$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d \quad \leftarrow$
$a \vee c \vee d' \quad \leftarrow$
$a \vee c' \vee d \quad \leftarrow$
$a \vee c' \vee d' \quad \leftarrow$
$b' \vee c' \vee d \quad \leftarrow$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$

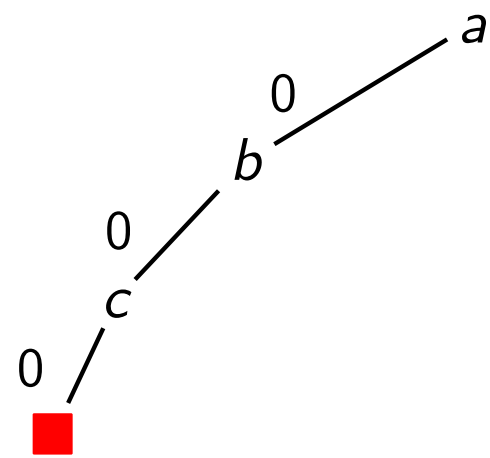


Conflict between $\leftarrow$ clauses wrt d and d'
(d and d' become unit under $\sigma : a, b, c \mapsto 0$, i.e., applying σ to the clauses and simplifying them result in unit clauses d and d')

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

$a' \vee b \vee c$ $\leftarrow$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$ $\leftarrow$
$a \vee c' \vee d'$ $\leftarrow$
$b' \vee c' \vee d$ $\leftarrow$
$a' \vee b \vee c'$ $\leftarrow$
$a' \vee b' \vee c$ $\leftarrow$



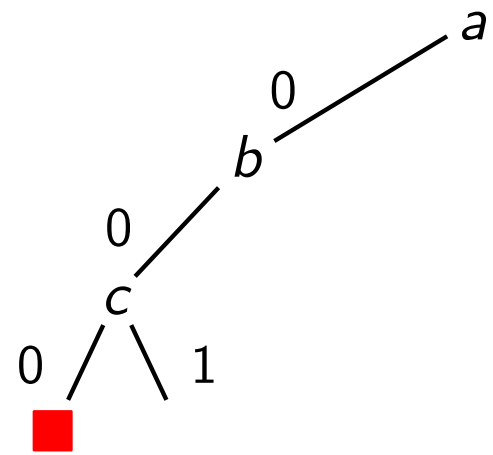
Resolve conflict: choose other value for c

Backtrack chronologically

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d$
$a \vee c \vee d' \quad$
$a \vee c' \vee d \quad \leftarrow$
$a \vee c' \vee d' \quad \leftarrow$
$b' \vee c' \vee d \quad \leftarrow$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$

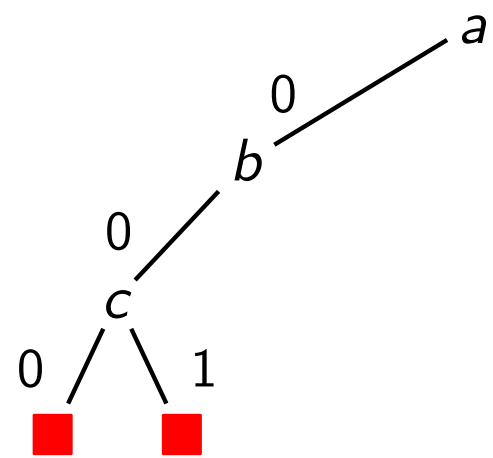


Conflict between $\leftarrow$ clauses wrt d and d'
(d and d' become unit under $a, b \mapsto 0, c \mapsto 1$)

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

$a' \vee b \vee c \quad \leftarrow$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d \quad \leftarrow$
$a' \vee b \vee c' \quad \leftarrow$
$a' \vee b' \vee c \quad \leftarrow$

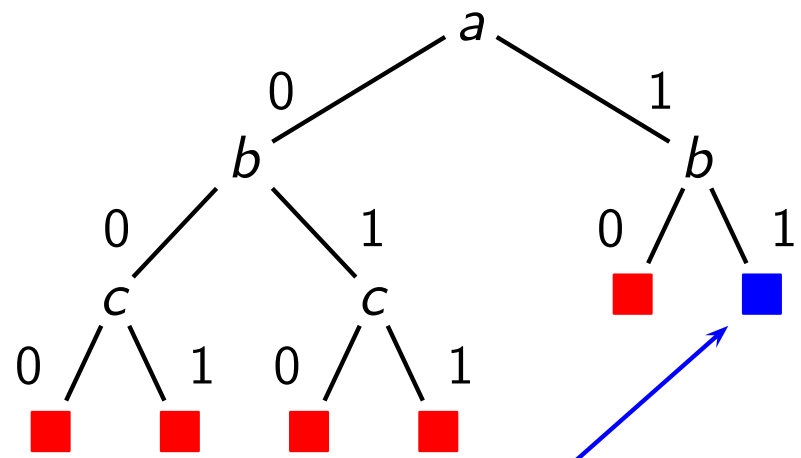


Resolve conflict: choose other value for b

$\leftarrow$ indicates clauses satisfied by current partial assignment

The basic DLL procedure: An example

$a' \vee b \vee c$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d$
$a' \vee b \vee c'$
$a' \vee b' \vee c$



$I(c) = I(d) = 1$ forced by $I(a) = I(b) = 1$ by UNIT

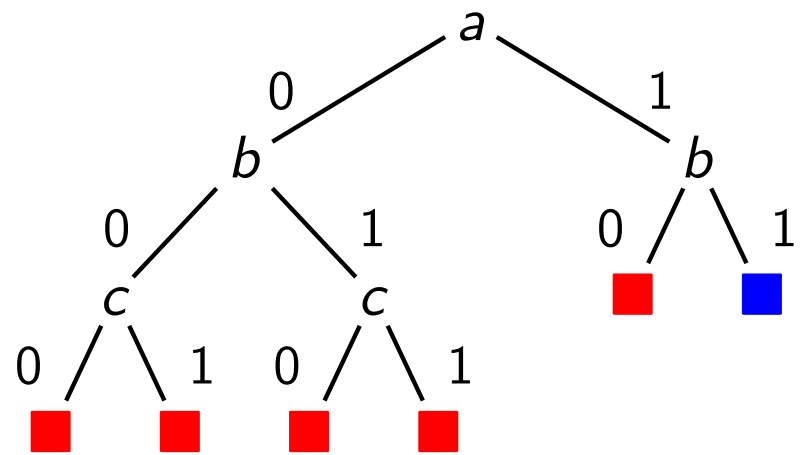
UNIT: Set the only remaining literal in the clause to 1

Boolean Constraint Propagation (BCP):

Iterate the application of UNIT until no longer possible

The basic DLL procedure: An example

$a' \vee b \vee c$
$a \vee c \vee d$
$a \vee c \vee d'$
$a \vee c' \vee d$
$a \vee c' \vee d'$
$b' \vee c' \vee d$
$a' \vee b \vee c'$
$a' \vee b' \vee c$



Easy to check: $I(a) = I(b) = I(c) = I(d) = 1$ is a model

Status of a clause

In the example we saw that the status of a clause can change.
Under a (partial) assignment, a clause can be

- satisfied: at least one of its literals is assigned to true,
- unsatisfied: all its literals are assigned to false
- unit: all but one of its literals are assigned to false
- unresolved: otherwise.

Example ($C : x_1 \vee x_2 \vee x_3$)

x_1	x_2	x_3	C is
1	0		satisfied
0	0	0	unsatisfied
0	0		unit
	0		unresolved

Partial assignments, unit clauses and antecedents

Definition

For a unit clause D with the unassigned literal ℓ under the current partial assignment, D is called the antecedent of ℓ.

Example

Let $C : \neg x_1 \vee x_4 \vee x_3$ be a clause and let $\sigma = \{x_1 \mapsto 1, x_4 \mapsto 0\}$ be a partial assignment. Then C is unit under σ (because applying σ to C and simplifying the clause result x_3). The partial assignment σ then implies $x_3 \mapsto 1$ by UNIT and $\mathit{antecedent}(x_3) = C$.

The features of the basic DLL procedure

- Polynomial memory requirement (exponential in DP!)
- Exponential time requirement remains
  (except for restricted inputs like 2-CNF or Horn clauses)
- Chronological backtracking (BT)
  Ok for random instances, bad for practical problems w. structure
- Need extensions to be of practical value
  - Heuristics to select variables to branch next
    (including the assignment to consider first)
  - "Intelligent" (e.g., non-chronological) backtracking
  - Learning from successful and unsuccessful decisions
  - Integration of all these extensions in one procedure

A first look at heuristics

We have seen the selection of a variable and a truth value!

- Heuristics are used to perform "good" selections
- We discuss two old ones first
  - DLIS: Dynamic Largest Individual Sum
  - The heuristic of Jeroslov and Wang (JW heuristic)
- Later, a more elaborated heuristic will be discussed which is "tailored" for the clause learning procedure!

Decision heuristics

DLIS: Dynamic Largest Individual Sum

Objective: Maximize the number of satisfied clauses

- Choose assignment with highest increase of satisfied clauses
- For each variable p, let
  - C_p^+, the no of unresolved clauses in which p occurs positively (i.e., p occurs unnegated in the clause)
  - C_p^-, the no of unresolved clauses in which p occurs negatively (i.e., $\neg p$ occurs in the clause)
- Let p (q) be the atom for which C_p^+ (C_q^-) is maximal
- If $C_p^+ > C_q^-$, choose p and assign it to true
- Otherwise, choose q and assign it to false
- Expensive! Requires O(nbr of literals) queries for each decision

Decision heuristics: Jeroslov-Wang (JW)

Objective: Try to get unit clauses soon

- Given a CNF $\mathcal{C}$, compute for every literal ℓ

$$J(\ell) = \sum_{\ell \in c, c \in \mathcal{C}} 2^{-|c|} \qquad (|c|\text{: nbr of literals in } c)$$

- Choose a (yet unasserted) literal ℓ that maximizes $J(\ell)$
- Literals occurring frequently in short clauses have exponentially higher weight (Try to get unit clauses soon!)
- What truth value is tried first?

The basic SAT algorithm

```
while (true)
{
   if (!Decide()) return (SAT);

   while (!BCP())

      if (!Resolve_Conflict()) return (UNSAT);
}
```

Choose next variable and value. Return `false` if all variables are assigned.

Apply repeatedly the unit rule. Return `false` if a conflict is reached.

Backtrack until no conflict occurs any more. Return `false`, if this is impossible.

The components of a modern SAT algorithm

- Decision: Choose a variable and a truth value
- Boolean Constraint Propagation (BCP): Propagate consequences (implications) of a decision through the formula, thereby changing the status of clauses. The implication graph is used to keep track of the changes.
  NB: Since 80–90% of the run time is spent in BCP, an efficient implementation is vital for a good overall performance of a solver.
- Resolve conflicts and organize backtracking: Depending on conflict resolution, backtrack non-chronologically

Outline

What is next?

Describe extensions which make SAT solvers practically useful

- Implication graphs, clause learning and non-chronological BT
- Different ways for finding clauses which can be learned
- Use of learned clauses
- An example of a conflict-driven branching heuristic

What is not topic of this lecture?

- Sophisticated data structures (for efficient implementations)
- Restarts and randomization of search
- Deletion schemes for learned clauses

The organization of the search

The search is organized in form of an implication graph (IG)

- Each node corresponds to a variable assignment (either from a decision or implied by BCP)
- Each decision is made at some decision level (dl)
- dl ranges from -1 (unassigned variables), 0 (unit clauses in the input clause set $\mathcal{C}$), up to the number of variables in $\mathcal{C}$
- Notation: $x = v@d$ means that x is assigned to v at dl d
- We often write $x@d$ for $x = 1@d$ and $\neg x@d$ for $x = 0@d$
- We often identify a node and its label of the form $\ell@d$
- Notation: The dual of a literal ℓ, ℓ^d, is defined as follows:
  ℓ^d is 1 if ℓ is 0 and ℓ^d is 0 if ℓ is 1

The implication graph (IG)

Definition (Implication graph, conflict graph)

An implication graph is a labeled directed acyclic graph (dag) $G = (V, E)$, where the following holds:

- Each node has a label of the form $\ell @ d$ (for a literal ℓ).
- $E = \{(v_i, v_j) \mid v_i, v_j \in V, \neg v_i \in \text{Antecedent}(v_j)\}$ denotes the set of directed edges where each edge (v_i, v_j) is labeled with $\text{Antecedent}(v_j)$.
- In case G is a conflict graph, it also contains a single conflict node labeled with κ and incoming edges $\{(v, \kappa) \mid \neg v \in c\}$ labeled with clause c.

Warning: IGs are hard to understand, even for winners of a SAT Race (cf the example IG at the home page of M. Soos at Link)

How to construct an implication graph?

Proposal for the construction of IG G

S1: Create a node for each decision literal ℓ and label it with $\ell@dl$.

S2: While there is a clause (either from the input formula or a learned one) of the form $C: \ell_1 \vee \cdots \vee \ell_k \vee \ell$ such that $\ell_1^d@dl_1, \ldots, \ell_k^d@dl_k$ label nodes in G and $dl = \max\{dl_1, \ldots, dl_k\}$,

1. add a node labeled $\ell@dl$ if not already present, and
2. add edges (ℓ_i^d, ℓ) $(1 \leq i \leq k)$ labeled C if not already present.

S3: If there exists a clause of the form $C: \ell_1 \vee \cdots \vee \ell_k$ such that $\ell_1^d@dl_1, \ldots, \ell_k^d@dl_k$ label nodes in G, then add a conflict node κ and perform 2. with κ instead of ℓ.

IGs are dynamic. A new decision or backtracking force changes. Often only a part of the IG (a partial IG) with the conflict node and its "responsible" decisions/implications is of interest.

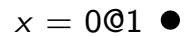
Example: Construct an implication graph

$c_1: \quad x \vee \quad y$
$c_2: \quad x \vee \quad z$
$c_3: \neg y \vee \neg z$

Example: Construct an implication graph

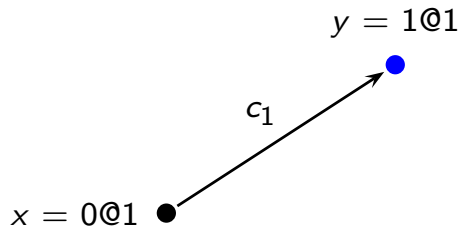
c_1 : $x \vee y$
c_2 : $x \vee z$
c_3 : $\neg y \vee \neg z$

- Set $x = 0@1$ (decision!)



Example: Construct an implication graph

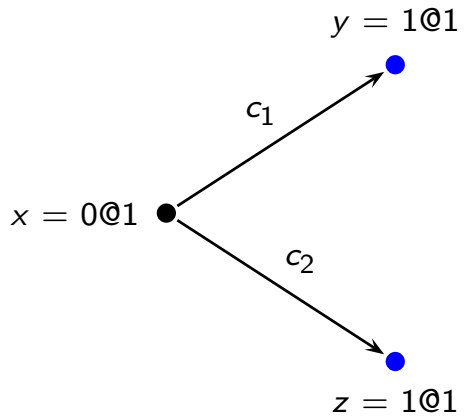
c_1 : $x \vee y$
c_2 : $x \vee z$
c_3 : $\neg y \vee \neg z$



- Set $x = 0@1$ (decision!)
- Applying BCP results in
  - $y = 1@1$ (antecedent$(y) = c_1$)

Example: Construct an implication graph

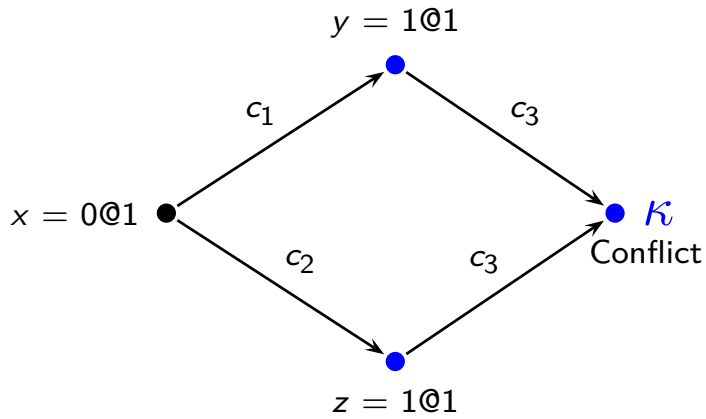
c_1 : $x \vee y$
c_2 : $x \vee z$
c_3 : $\neg y \vee \neg z$



- Set $x = 0@1$ (decision!)
- Applying BCP results in
  - $y = 1@1$ (antecedent$(y) = c_1$)
  - $z = 1@1$ (antecedent$(z) = c_2$)

Example: Construct an implication graph

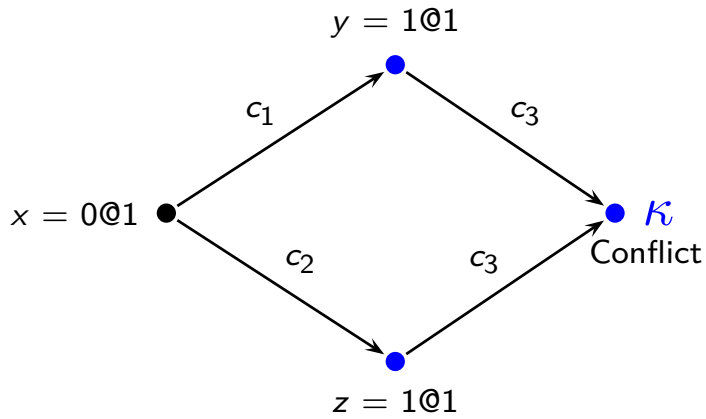
c_1: $x \vee y$
c_2: $x \vee z$
c_3: $\neg y \vee \neg z$



- Set $x = 0@1$ (decision!)
- Applying BCP results in
  - $y = 1@1$ (antecedent$(y) = c_1$)
  - $z = 1@1$ (antecedent$(z) = c_2$)
- With c_3, we obtain a conflict

Example: Construct an implication graph

c_1: $x \vee y$
c_2: $x \vee z$
c_3: $\neg y \vee \neg z$

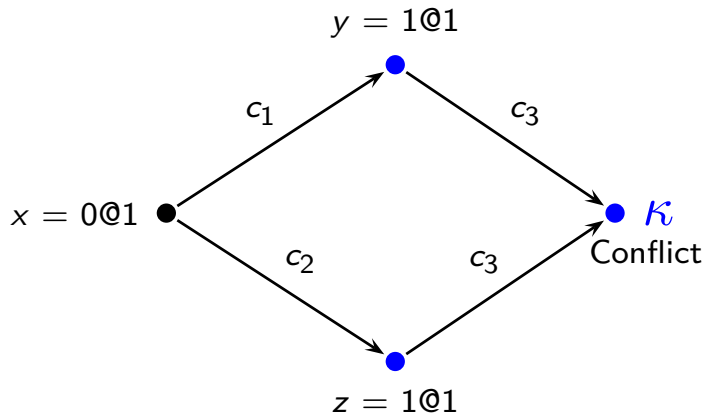


- Set $x = 0@1$ (decision!)
- Applying BCP results in
  - $y = 1@1$ (antecedent$(y) = c_1$)
  - $z = 1@1$ (antecedent$(z) = c_2$)
- With c_3, we obtain a conflict

How can we avoid this conflict in other parts of the search space?

Example: Construct an implication graph

c_1: $x \vee y$
c_2: $x \vee z$
c_3: $\neg y \vee \neg z$

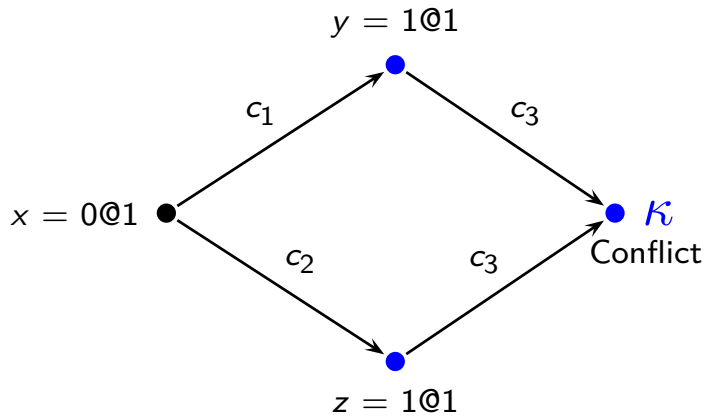


- Set $x = 0@1$ (decision!)
- Applying BCP results in
  - $y = 1@1$ (antecedent$(y) = c_1$)
  - $z = 1@1$ (antecedent$(z) = c_2$)
- With c_3, we obtain a conflict

How can we avoid this conflict in other parts of the search space?
Add a clause which becomes false $\Longrightarrow$ conflict-driven clause learning

Example: Construct an implication graph

c_1: $x \vee y$
c_2: $x \vee z$
c_3: $\neg y \vee \neg z$



- Set $x = 0@1$ (decision!)
- Applying BCP results in
  - $y = 1@1$ (antecedent$(y) = c_1$)
  - $z = 1@1$ (antecedent$(z) = c_2$)
- With c_3, we obtain a conflict

How can we avoid this conflict in other parts of the search space?

Add a clause which becomes false $\Longrightarrow$ conflict-driven clause learning

Which clause do we add in this case?

The first approach

Clause learning and backtracking in GRASP

We discuss clause learning in combination with dependency-directed backtracking. The discussion is based on the paper:

> J. P. Marques Silva and K. A. Sakallah.
> GRASP: A Search Algorithm for Propositional Satisfiability.
> IEEE Trans. Computers, 48:5 1999, pp. 506–521.

We continue afterwards with more modern features like clause learning according to the first UIP scheme, conflict-driven backtracking and conflict-driven heuristics.

Example: Learning a clause from an implication graph

Current truth assignment: $\{\neg x_9@1, \neg x_{10}@3, \neg x_{11}@3, x_{12}@2, x_{13}@2\}$

Current decision assignment: $\{x_1@6\}$

$c_1 : \neg x_1 \vee x_2$
$c_2 : \neg x_1 \vee x_3 \vee x_9$
$c_3 : \neg x_2 \vee \neg x_3 \vee x_4$
$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$
$c_7 : x_1 \vee x_7 \vee \neg x_{12}$
$c_8 : x_1 \vee x_8$
$c_9 : \neg x_7 \vee \neg x_8 \vee \neg x_{13}$

Construct the implication graph!

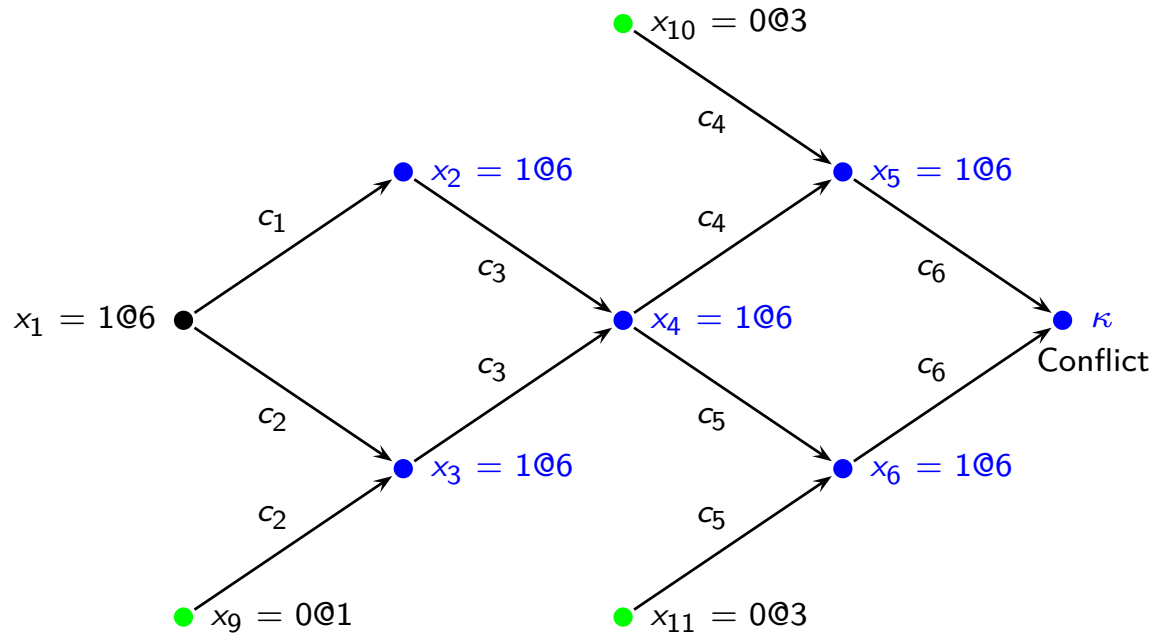
- indicates an assignment by a decision
- indicates an assignment by BCP
- indicates an assignment by a decision or BCP

Example: Learning a clause from an implication graph

Current truth assignment: $\{\neg x_9@1, \neg x_{10}@3, \neg x_{11}@3, x_{12}@2, x_{13}@2\}$

Current decision assignment: $\{x_1@6\}$

$c_1 : \neg x_1 \vee x_2$
$c_2 : \neg x_1 \vee x_3 \vee x_9$
$c_3 : \neg x_2 \vee \neg x_3 \vee x_4$
$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$
$c_7 : x_1 \vee x_7 \vee \neg x_{12}$
$c_8 : x_1 \vee x_8$
$c_9 : \neg x_7 \vee \neg x_8 \vee \neg x_{13}$



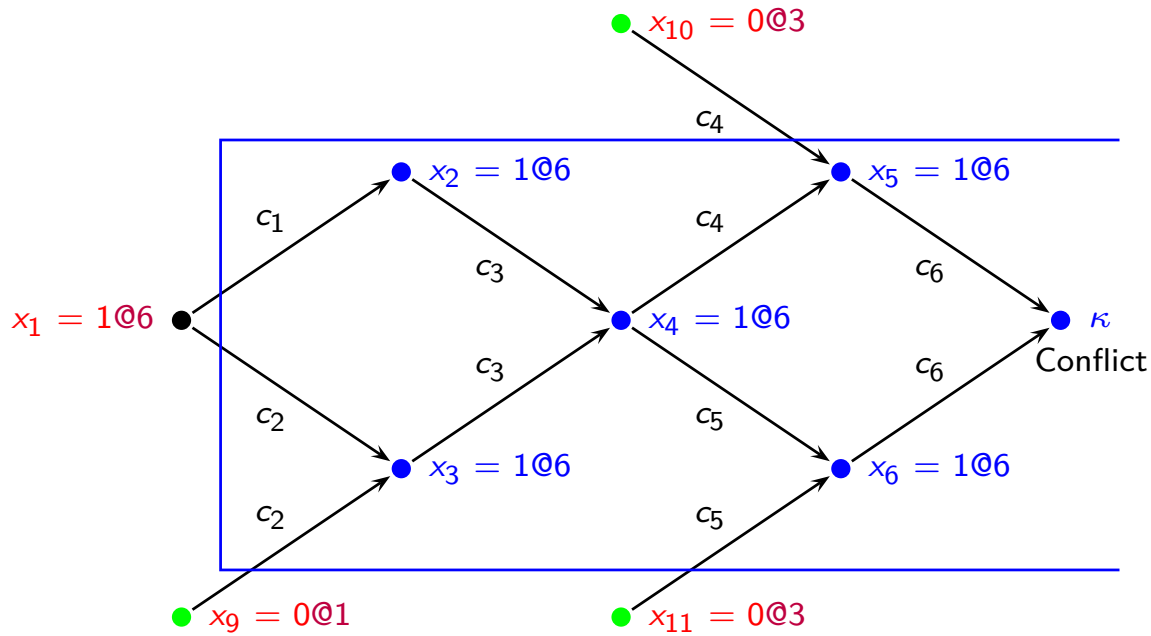
Next: Separate the "roots" from the "conflict"!
(We will see later how to do this systematically)

Example: Learning a clause from an implication graph

Current truth assignment: $\{\neg x_9@1, \neg x_{10}@3, \neg x_{11}@3, x_{12}@2, x_{13}@2\}$

Current decision assignment: $\{x_1@6\}$

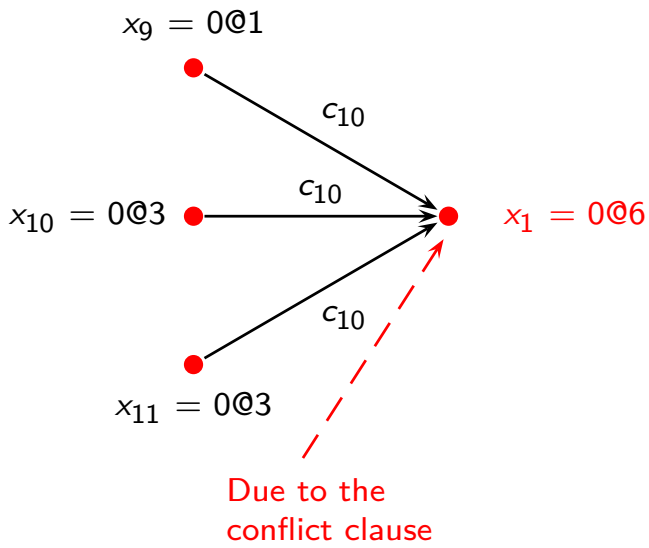
$c_1 : \neg x_1 \vee x_2$
$c_2 : \neg x_1 \vee x_3 \vee x_9$
$c_3 : \neg x_2 \vee \neg x_3 \vee x_4$
$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$
$c_7 : x_1 \vee x_7 \vee \neg x_{12}$
$c_8 : x_1 \vee x_8$
$c_9 : \neg x_7 \vee \neg x_8 \vee \neg x_{13}$



We learn the conflict clause $c_{10} : \neg x_1 \vee x_9 \vee x_{11} \vee x_{10}$
The partial assignment (with the indicated literals) makes c_{10} false
We backtrack to dl 6, undo the decision and flip x_1! Why?

The impact of learned clauses

$c_1 : \neg x_1 \vee x_2$
$c_2 : \neg x_1 \vee x_3 \vee x_9$
$c_3 : \neg x_2 \vee \neg x_3 \vee x_4$
$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$
$c_7 : x_1 \vee x_7 \vee \neg x_{12}$
$c_8 : x_1 \vee x_8$
$c_9 : \neg x_7 \vee \neg x_8 \vee \neg x_{13}$
$c_{10} : \neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$



At dl=6, $\neg x_1$ becomes unit and is asserted to 0. We construct the IG.

The impact of learned clauses

$c_1 : \neg x_1 \vee x_2$

$c_2 : \neg x_1 \vee x_3 \vee x_9$

$c_3 : \neg x_2 \vee \neg x_3 \vee x_4$

$c_4 : \neg x_4 \vee x_5 \vee x_{10}$

$c_5 : \neg x_4 \vee x_6 \vee x_{11}$

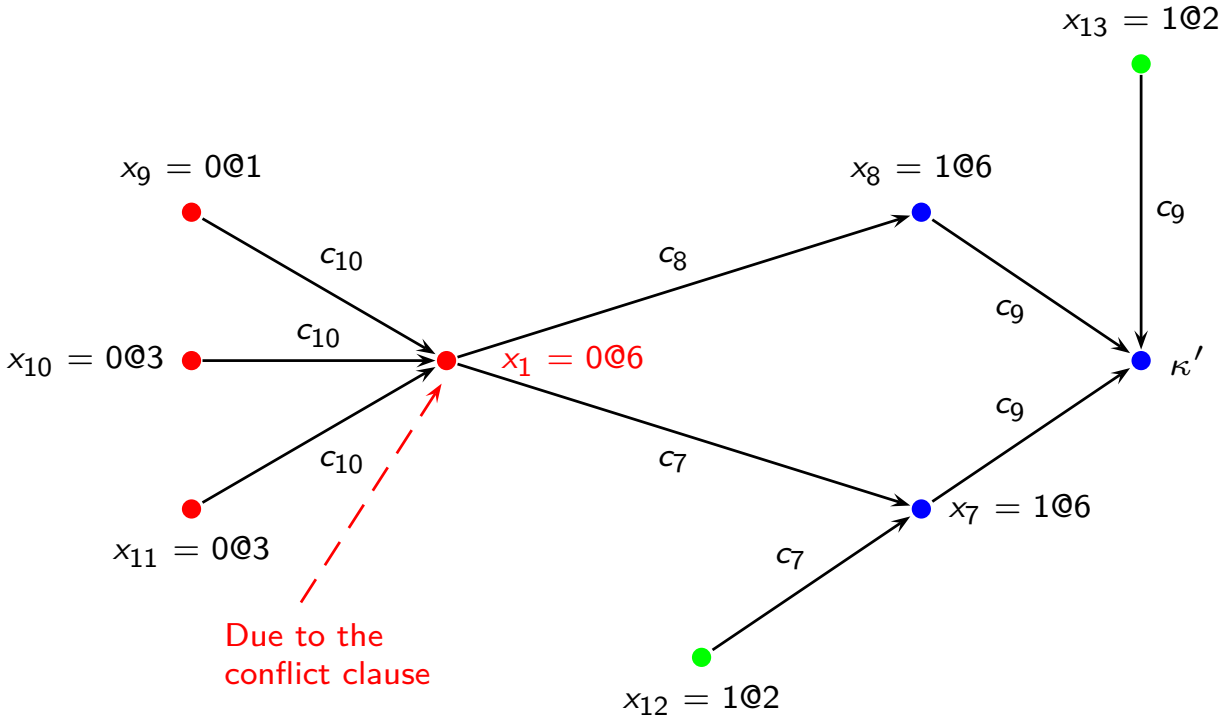
$c_6 : \neg x_5 \vee \neg x_6$

$c_7 : x_1 \vee x_7 \vee \neg x_{12}$

$c_8 : x_1 \vee x_8$

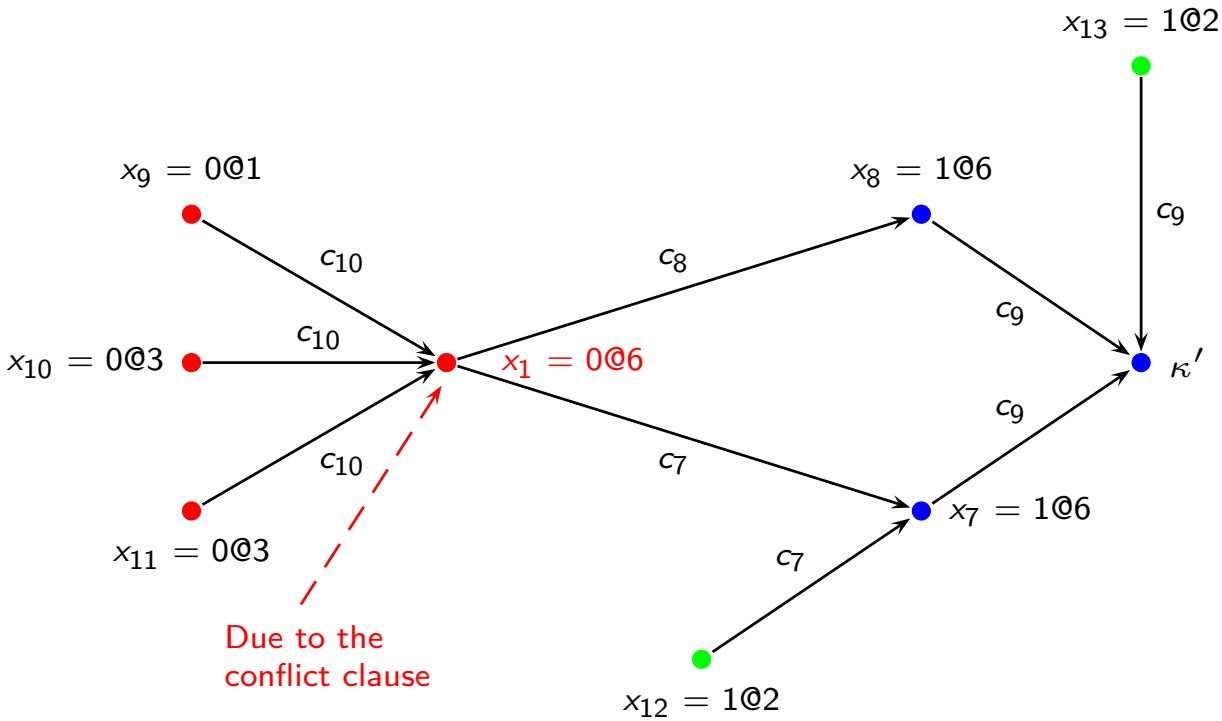
$c_9 : \neg x_7 \vee \neg x_8 \vee \neg x_{13}$

$c_{10} : \neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$



The impact of learned clauses

$c_1 : \neg x_1 \vee x_2$
$c_2 : \neg x_1 \vee x_3 \vee x_9$
$c_3 : \neg x_2 \vee \neg x_3 \vee x_4$
$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$
$c_7 : x_1 \vee x_7 \vee \neg x_{12}$
$c_8 : x_1 \vee x_8$
$c_9 : \neg x_7 \vee \neg x_8 \vee \neg x_{13}$
$c_{10} : \neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$



Then we learn the new conflict clause $c_{11} : \neg x_{13} \vee \neg x_{12} \vee x_{11} \vee x_{10} \vee x_9$

No decision is involved! Where do we backtrack to?

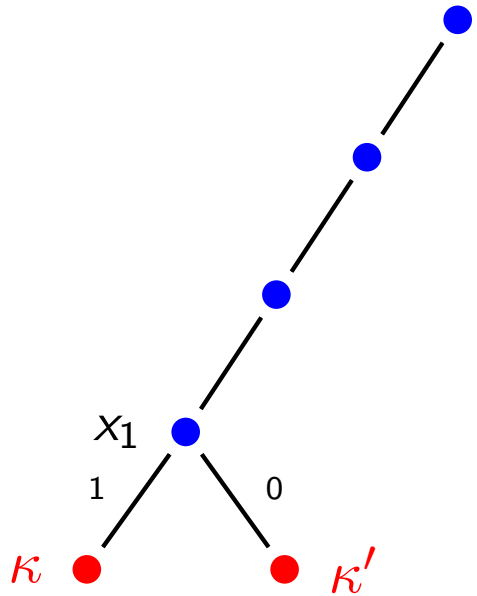
Dependency-directed backtracking

Which assignments caused the conflicts?

$x_9 = 0@1$
$x_{10} = 0@3$
$x_{11} = 0@3$
$x_{12} = 1@2$
$x_{13} = 1@2$

Variables in learned clause

These ones are sufficient!



$\mathrm{dl} = 3$

$\mathrm{dl} = 4$

$\mathrm{dl} = 5$

$\mathrm{dl} = 6$

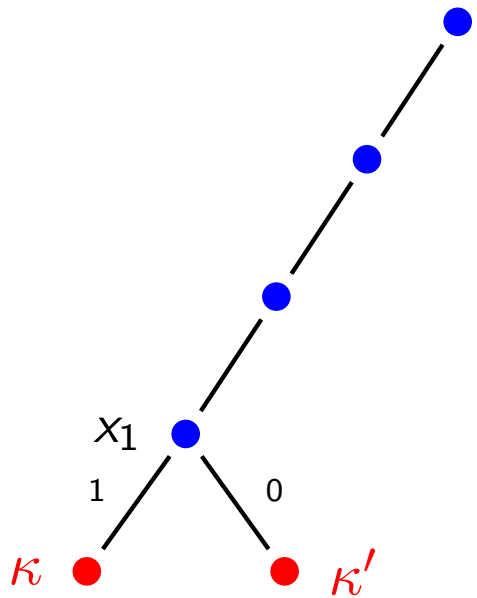
Dependency-directed backtracking

Which assignments caused the conflicts?

$x_9 = 0@1$
$x_{10} = 0@3$
$x_{11} = 0@3$
$x_{12} = 1@2$
$x_{13} = 1@2$

Variables in learned clause

These ones are sufficient!



$\mathrm{dl} = 3$

$\mathrm{dl} = 4$

$\mathrm{dl} = 5$

$\mathrm{dl} = 6$

All possibilities for x_1 exhausted! To which level do we backtrack?

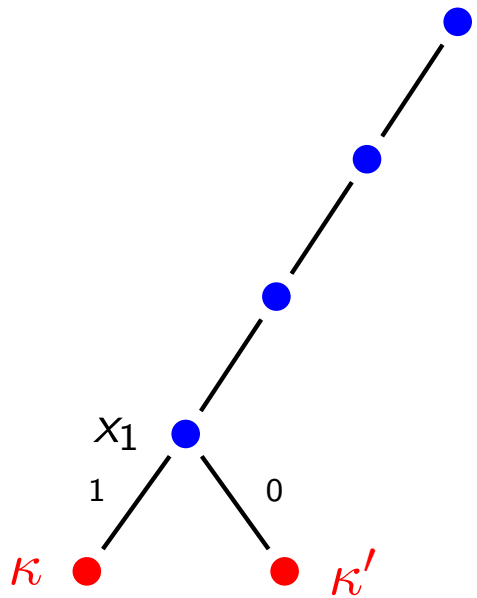
Dependency-directed backtracking

Which assignments caused the conflicts?

$x_9 = 0@1$
$x_{10} = 0@3$
$x_{11} = 0@3$
$x_{12} = 1@2$
$x_{13} = 1@2$

Variables in learned clause

These ones are sufficient!



$\mathrm{dl} = 3$

$\mathrm{dl} = 4$

$\mathrm{dl} = 5$

$\mathrm{dl} = 6$

All possibilities for x_1 exhausted! To which level do we backtrack? Could go to dl=5. Is this clever?

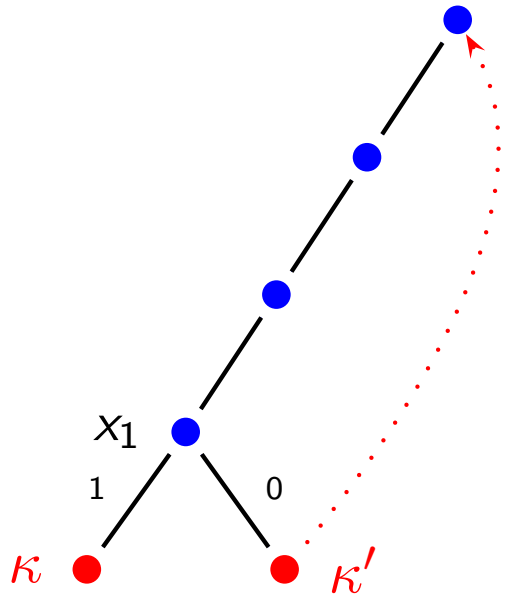
Dependency-directed backtracking

Which assignments caused the conflicts?

$x_9 = 0@1$
$x_{10} = 0@3$
$x_{11} = 0@3$
$x_{12} = 1@2$
$x_{13} = 1@2$

Variables in learned clause

These ones are sufficient!



$\mathrm{dl} = 3$

$\mathrm{dl} = 4$

$\mathrm{dl} = 5$

$\mathrm{dl} = 6$

Backtrack to $\mathrm{dl} = 3$, which is the biggest dl occurring in the clause.
For a bigger dl, the conflicts occur again!

Dependency-directed backtracking

Rule

- Backtrack to the largest dl in the conflict clause
- Delete the decision (but keep the dl)
- Works for both, initial conflicts and conflict after adding learned conflict clauses (causing a flip of a truth value)

What if the flipped assignment for a variable x works?

- Then we don't get a conflict immediately simply by BCP
- We continue with the next decision level, letting the current one without a decision variable
- Later, if we backtrack to the current dl, then the variable x is flipped again. But now, the conflict clause from before results in a conflict which initiates further backtracking.

Extending the approach

We extend and change the first approach as follows:

- We discuss a systematic approach to derive conflict clauses by the first UIP scheme used in all modern SAT solvers.
- It has been demonstrated empirically that this scheme works well and outperforms other schemes.
- Resolution can be used to actually compute the conflict clause from the implication graph. Consequently, one can get a resolution refutation as a witness for unsatisfiability.
- A modified backtracking scheme called conflict-driven backtracking together with an especially tailored decision heuristics like VSIDS or Berkmin is used.

Cuts in graphs

Separating the conflict from the roots

We have already seen one possibility to separate the "roots" from the "conflict". In the following, possible separation methods are described which influence the clause learned from a conflict.

Definition (cut, cut set)

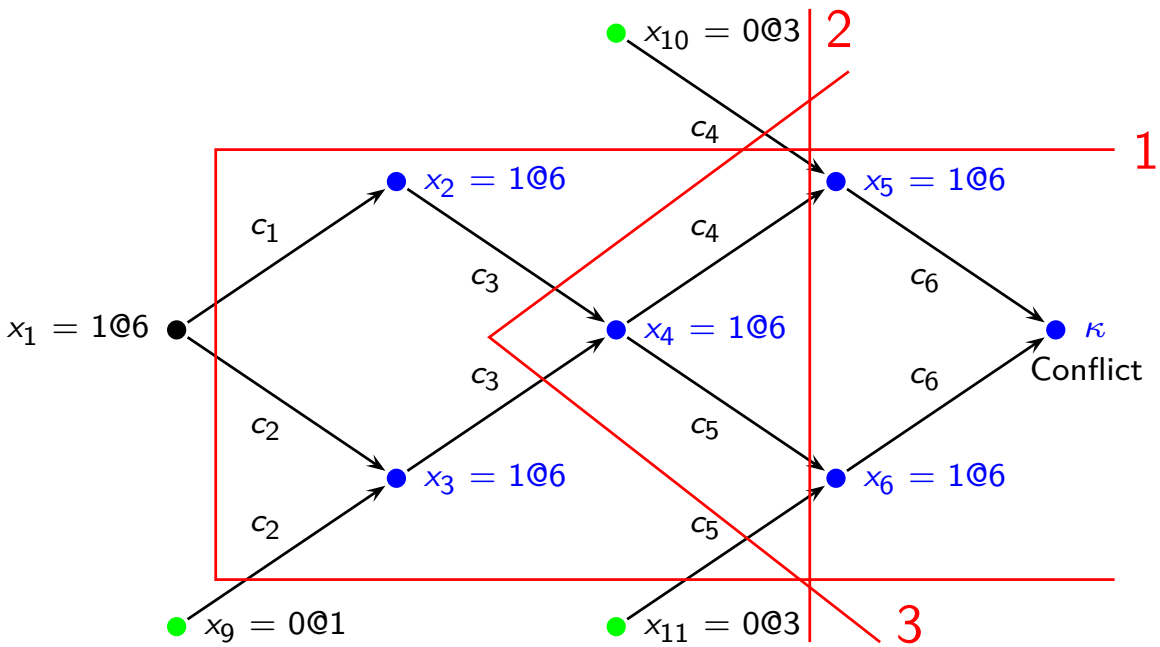
Let $G = (V, E)$ be a graph with vertices in V and edges in E. A cut $C = (S, T)$ is a partition of V. A cut set of C is a set of edges whose end parents occur in different sets of the partition (i.e., one in S and the other one in T).

Definition

A conflict clause (for a clause set $\mathcal{C}$) is any clause implied by $\mathcal{C}$.

More conflict clauses

Let C be a cut in the IG, separating the conflict node from all roots. Let E_c be the cut edges of the form (m_i, n_i). Let m_i be labeled with literal ℓ_i in IG. Then $\bigvee_i \ell_i^d$ is a conflict clause where ℓ^d is the dual of ℓ.



Cuts and conflict clauses

1: $x_{10} \vee \neg x_1 \vee x_9 \vee x_{11}$

2: $x_{10} \vee \neg x_4 \vee x_{11}$

3: $x_{10} \vee \neg x_2 \vee \neg x_3 \vee x_{11}$

$\vdots$

Which conflict clause(s) shall we add?

- How many of the conflict clauses should we add?
- If not all of them, then which ones?
  - Shorter ones?
  - The ones with a "good" influence on the backtrack level
    (But how to measure/estimate this influence?)
  - The ones directing search to yet unexplored regions?
  - The most influential/beneficial?

Which conflict clause(s) shall we add?

- How many of the conflict clauses should we add?
- If not all of them, then which ones?
  - Shorter ones?
  - The ones with a "good" influence on the backtrack level
    (But how to measure/estimate this influence?)
  - The ones directing search to yet unexplored regions?
  - The most influential/beneficial?

We will see in the following a further possibility to implement clause learning and conflict-driven backtracking

Asserting clauses

Definition

An asserting clause (AC) is a conflict clause with a single literal from the current decision level.

- ↪ Backtracking to the right level makes it a unit clause

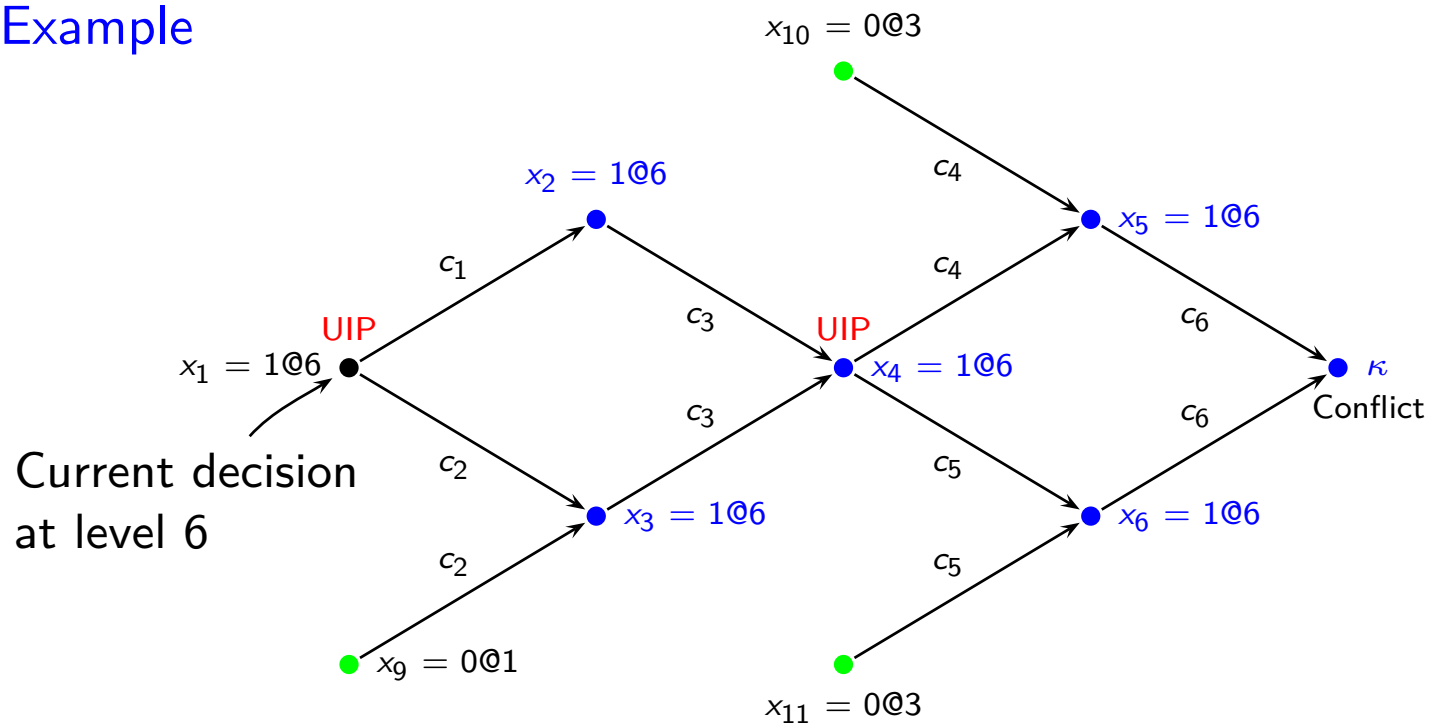
Modern SAT solvers only consider asserting clauses!

Unique implication points (UIPs)

Definition

A unique implication point (UIP) is an internal node in the IG that all paths from the decision node (at the current dl) to the conflict node go through it. The first UIP is the UIP closest to the conflict.

Example

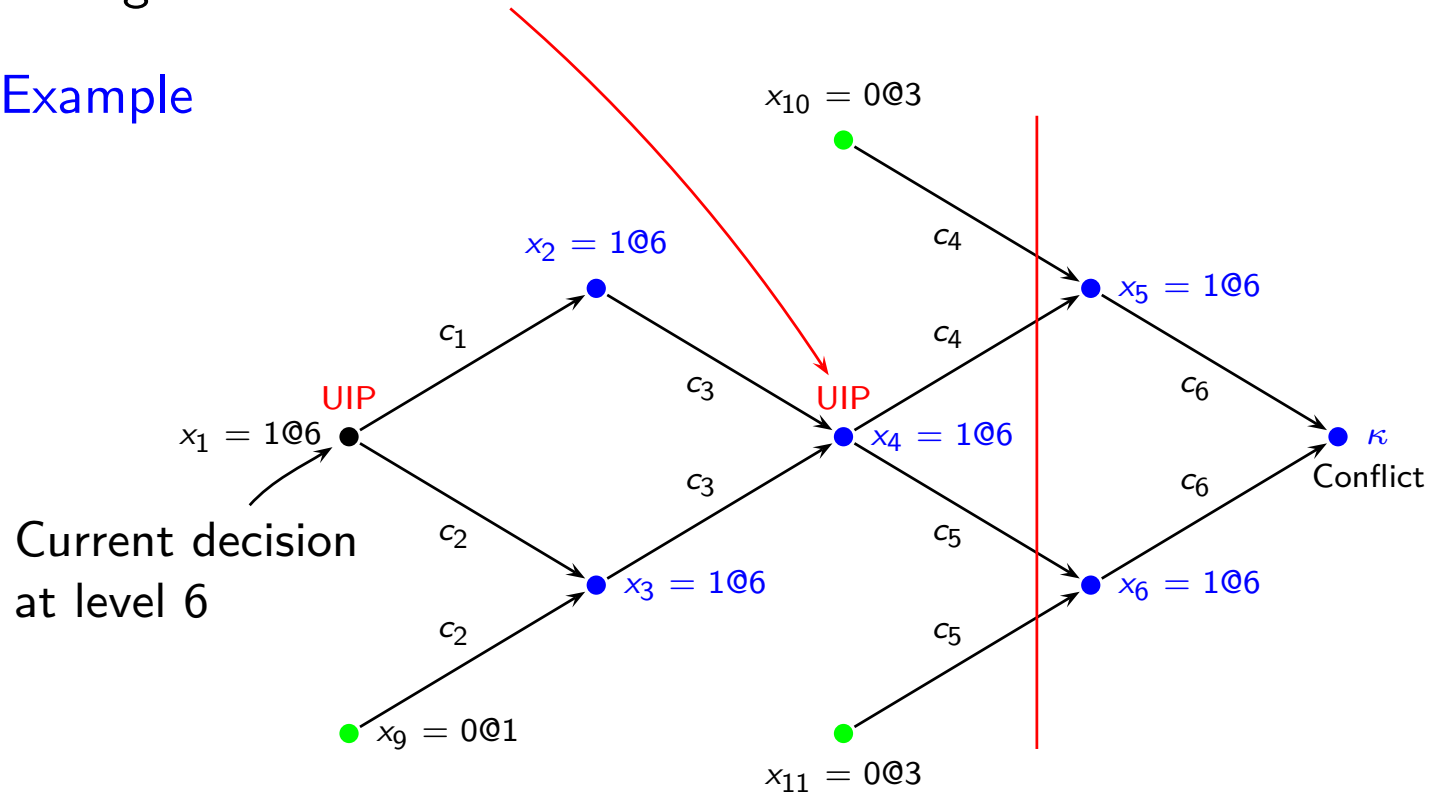


Unique implication points (UIPs)

Definition

A unique implication point (UIP) is an internal node in the IG that all paths from the decision node (at the current dl) to the conflict node go through it. The first UIP is the UIP closest to the conflict.

Example



Conflict clause wrt first UIP: $x_{10} \vee \neg x_4 \vee x_{11}$

Conflict-driven backtracking

- The conflict clause is $x_{10} \vee \neg x_4 \vee x_{11}$
- Recall: $x_{10} = 0@3$, $x_4 = 1@6$, and $x_{11} = 0@3$
- With standard non-chronological backtracking: $\mathrm{dl} = 6$
- With conflict-driven backtracking: backtrack to the second highest dl, dl, in the clause (without erasing it)
- Here, we backtrack to $\mathrm{dl} = 3$ and erase all decisions at $\mathrm{dl} > 3$
- Then the literal with the currently highest dl (here 6) is implied at $\mathrm{dl} = 3$, resulting in $x_4 = 0@3$

The use of resolution to derive conflict clauses

- Conflict clauses can be computed by resolution
- Recall: The order of literals in clauses is irrelevant!
- The propositional binary resolution rule is

$$\frac{k_1 \vee \cdots \vee k_n \vee z \qquad \ell_1 \vee \cdots \vee \ell_m \vee \neg z}{k_1 \vee \cdots \vee k_n \vee \ell_1 \vee \cdots \vee \ell_m}\ \mathit{res}$$

- z is called the atom resolved upon
- The propositional factoring rule is

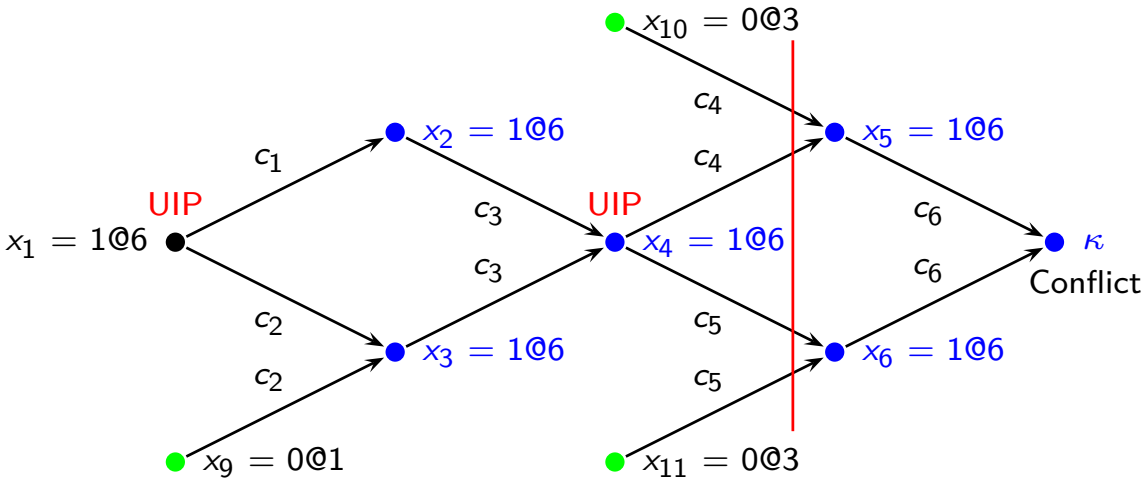
$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee \ell \vee \ell_{k+1} \vee \cdots \vee \ell_j \vee \ell \vee \ell_{j+1} \vee \cdots \vee \ell_n}{\ell_1 \vee \cdots \vee \ell_k \vee \ell \vee \ell_{k+1} \vee \cdots \vee \ell_j \vee \ell_{j+1} \vee \cdots \vee \ell_n}\ \mathit{fac}$$

Example: Compute a conflict clause by resolution

$c_4 : \neg x_4 \vee x_5 \vee x_{10}$

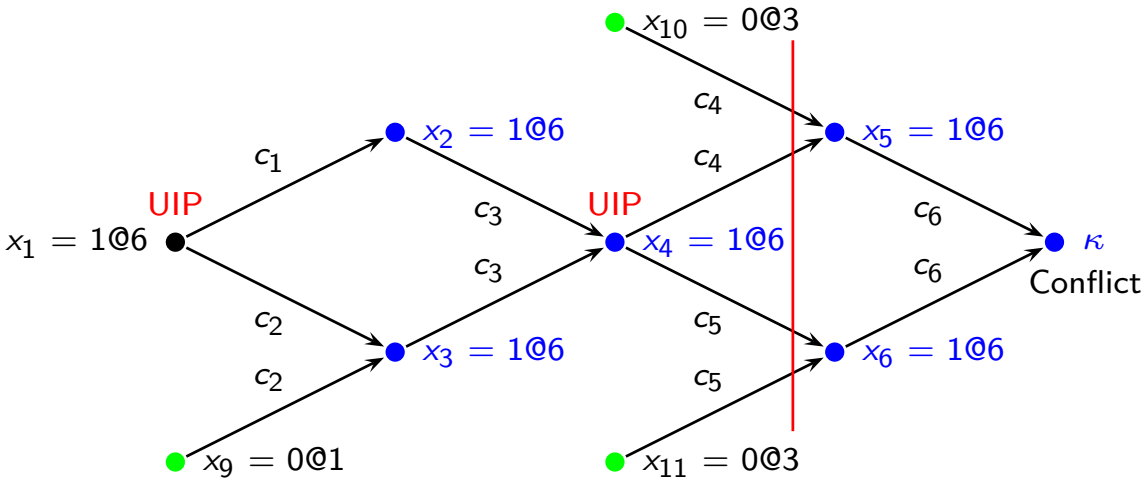
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$

$c_6 : \neg x_5 \vee \neg x_6$



Example: Compute a conflict clause by resolution

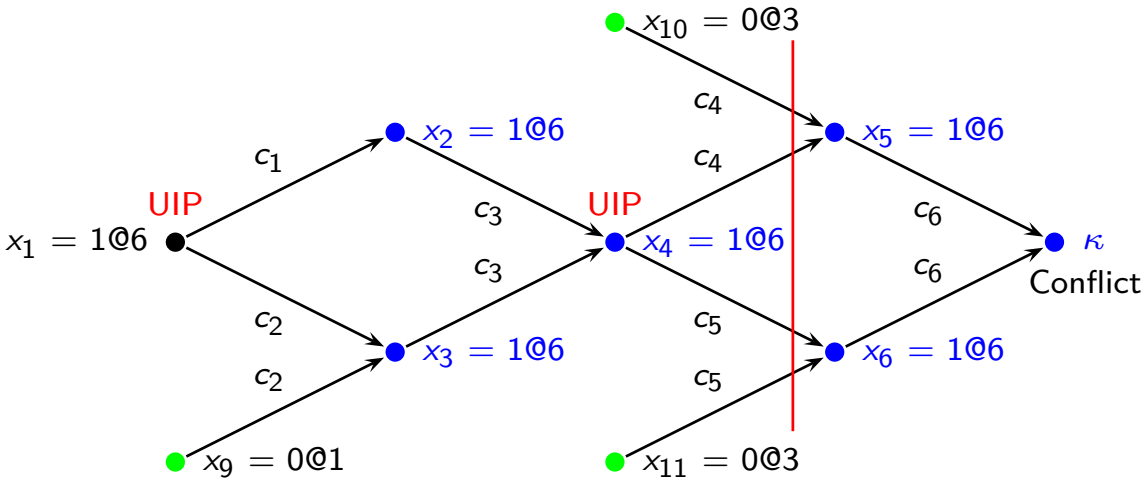
$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$



- Start with the unsatisfied clause c_6 (for κ)

Example: Compute a conflict clause by resolution

$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$

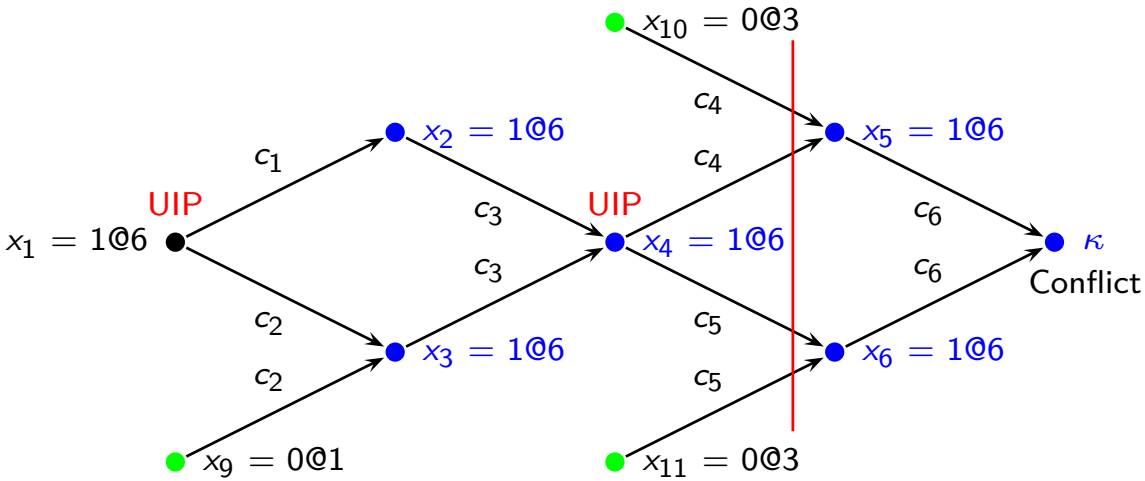


- Start with the unsatisfied clause c_6 (for κ)
- Take predecessor x_5 (from current dl) and the corresponding (antecedent) clause c_4 and build

$$r_1 = res(c_6, c_4, x_5) = \neg x_4 \vee x_{10} \vee \neg x_6$$

Example: Compute a conflict clause by resolution

$c_4 : \neg x_4 \vee x_5 \vee x_{10}$
$c_5 : \neg x_4 \vee x_6 \vee x_{11}$
$c_6 : \neg x_5 \vee \neg x_6$



- Start with the unsatisfied clause c_6 (for κ)
- Take predecessor x_5 (from current dl) and the corresponding (antecedent) clause c_4 and build

$$r_1 = res(c_6, c_4, x_5) = \neg x_4 \vee x_{10} \vee \neg x_6$$

- Take predecessor x_6 (from current dl) and the corresponding (antecedent) clause c_5 and build

$$r_2 = fac(res(r_1, c_5, x_6)) = \neg x_4 \vee x_{10} \vee x_{11} \quad (= \text{conflict clause})$$

Finding the conflict clause

Algorithm 1: Analyze-conflict

Input:
Output: BT level and new conflict clause
begin
    if *current-decision-level* = *0* **then**
        return −1
    ;
    cl := current-conflicting-clause;
    while ¬ *Stop-criterium-met(cl)* **do**
        lit := Last-assigned-literal(cl);
        var := Variable-of-literal(lit);
        ante := Antecedent(var);
        cl := Resolve(cl,ante,var);
    add-clause-to-database(cl);
    return clause-asserting-level(cl);
    `/* 2nd highest dl in cl */`
end

Stop when *cl* is asserting!

Applied to our example:

	cl	lit	var	ante
c_6	$\neg x_5 \vee \neg x_6$	$\neg x_5$	x_5	c_4
r_1	$\neg x_4 \vee x_{10} \vee \neg x_6$	$\neg x_6$	x_6	c_5
r_2	$\neg x_4 \vee x_{10} \vee x_{11}$			

Decision heuristic: VSIDS

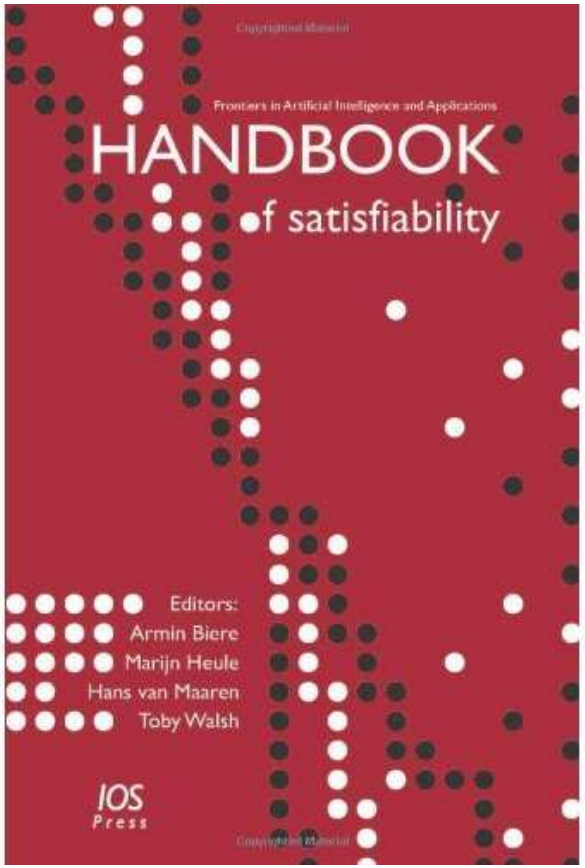
- Basic idea: Make the heuristic conflict-driven, i.e., give higher scores to variables involved in conflicts
- Similar to DLIS with the following differences:
  - When counting the no of clauses in which a literal appears, disregard whether clause is already satisfied or not (decision quality $\downarrow$, performance $\uparrow$ with "good" data struct.)
  - Periodically, divide the scores by 2

Background reading

Handbook of Satisfiability, IOS Press, 2009

Editors

- Armin Biere
- Marijn Heule
- Hans van Maaren
- Toby Walsh



Learning objectives

You should be able to

- explain and apply the basic DLL procedure,
- explain different methods to construct conflict clauses,
- construct implication graphs and conflict clauses (including resolution),
- explain the use of cuts, UIPs, etc.,
- distinguish different BT schemes,
- calculate the BT level for different versions,
- explain different heuristics,
- prove basic properties about the procedures and implication graphs (ex).