Constraint Programming

Lecture 4. Solving CSPs using Lazy Clause Generation.

Ruslan Sadykov

INRIA Bordeaux—Sud-Ouest

27 January 2022

Contents

What is the SAT problem ?

Given a propositional formula (Boolean variables with AND, OR, NOT), is there an assignment to the variables such that the formula evaluates to true ?

- NP-complete problem with applications in AI, formal methods
- Input usually given as Conjunctive Normal Form (CNF) formulas
- It is possible to do the linear reduction from general propositional formulas

Conjunctive Normal Form

SAT solvers usually take input in CNF : an AND of ORs of literals :

- Atom — a propositional variable : a, b, c
- Literal — an atom or its negation : a, $\bar{a}$, b, $\bar{b}$
- Clause — A disjunction of some literals : $a \vee \bar{b} \vee c$
- CNF formula — A conjunction of some clauses : $(a \vee \bar{b} \vee c) \wedge (\bar{c} \vee \bar{a})$

A formula is *satisfied* by a variable assignment if every clause has at least one literal which is true under that assignment.

A formula is *unsatisfied* by a variable assignment if some clause's literals are all false under that assignment.

DPLL algorithm for the SAT problem (1)

Unit propagation

If a clause is a *unit clause*, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary value to make this literal true.

Pure literal elimination

If a propositional variable occurs with only one polarity in the formula, it is called *pure*. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted.

DPLL algorithm for the SAT problem (2)

Algorithm 1: `DPLL`(Φ)

if Φ *is a consistent set of literals* **then**
  return true;
if Φ *contains an empty clause* **then**
  return false;
foreach *unit clause* $\{l\}$ **in** Φ **do**
  $\Phi \leftarrow$ *unit-propagate*(l, Φ);
foreach *literal* l *that occurs pure* **in** Φ **do**
  $\Phi \leftarrow$ *pure-literal-assign*(l, Φ);
$l \leftarrow$ *choose-literal*(Φ);
return `DPLL` $(\Phi \wedge \{l\})$ **or** `DPLL` $(\Phi \wedge \{\bar{l}\})$;

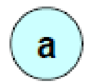
DPLL algorithm : illustration

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

All clauses making a CNF formula

DPLL algorithm : illustration

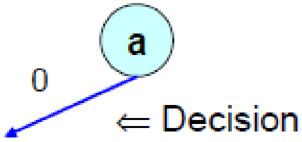
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Pick a variable

DPLL algorithm : illustration

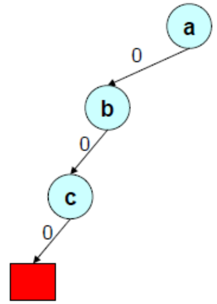
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



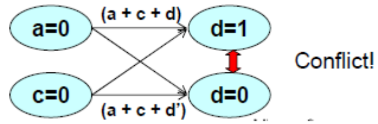
Make a decision, variable $a = False$ ($a = 0$)

DPLL algorithm : illustration

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



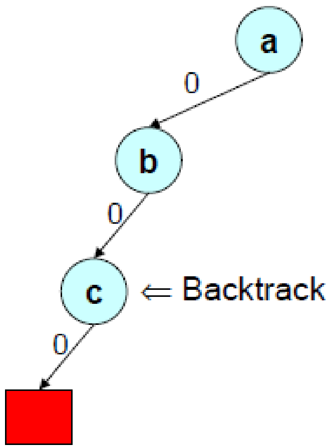
Implication Graph



After making several decisions, we find an implication graph that leads to a conflict

DPLL algorithm : illustration

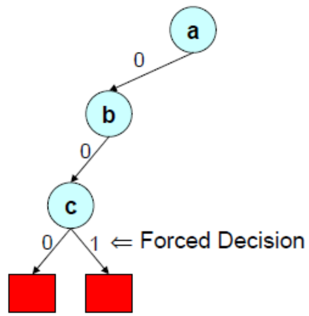
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

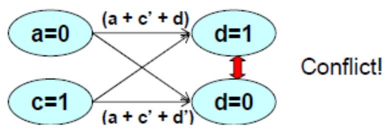


Now backtrack to immediate level and by force assign opposite value to that variable

DPLL algorithm : illustration

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

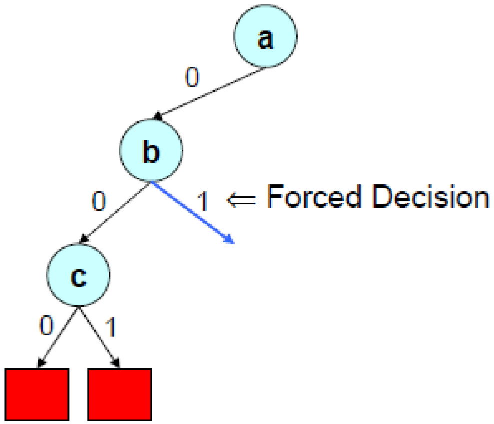




But a forced decision still leads to another conflict

DPLL algorithm : illustration

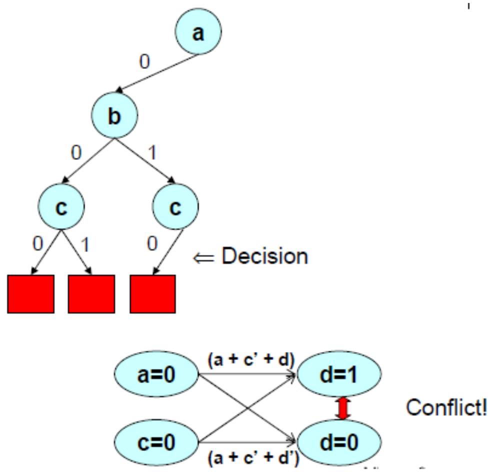
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Backtrack to previous level and make a forced decision

DPLL algorithm : illustration

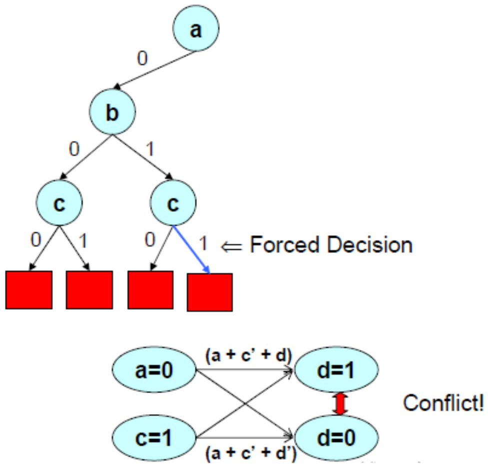
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Make a new decision, but it leads to a conflict

DPLL algorithm : illustration

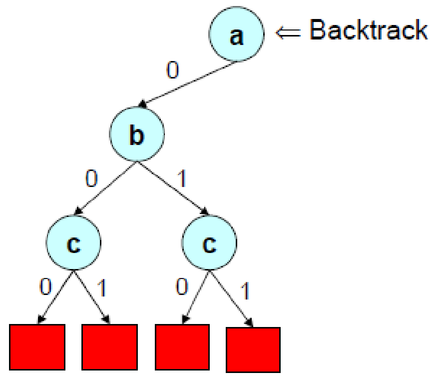
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Make a forced decision, but again it leads to a conflict

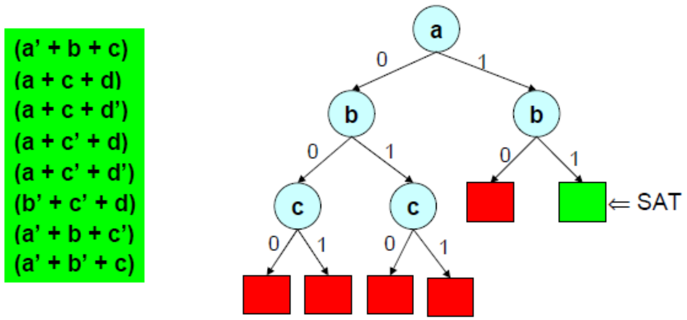
DPLL algorithm : illustration

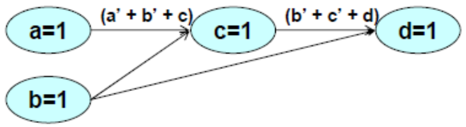
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Backtrack to previous level

DPLL algorithm : illustration





Continue in this way and the final implication graph

Conflict-Driven Clause Learning (CDCL)

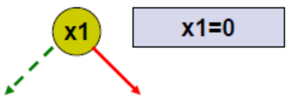
Works as follows

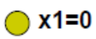
1. Select a variable and assign True or False. This is called decision state. Remember the assignment.
2. Apply Boolean Constraint Propagation (unit propagation).
3. Build the implication graph.
4. If there is any conflict
   - Find the cut in the implication graph that led to the conflict
   - Derive a new clause which is the negation of the assignments that led to the conflict
   - Non-chronologically backtrack (*back jump*) to the appropriate decision level, where the first-assigned variable involved in the conflict was assigned
5. Otherwise continue from step 1 until all variable values are assigned

CDCL algorithm : illustration

Step 1

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'



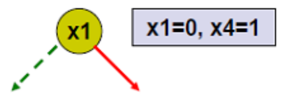


At first pick a branching variable, namely x_1. A yellow circle means an arbitrary decision

CDCL algorithm : illustration

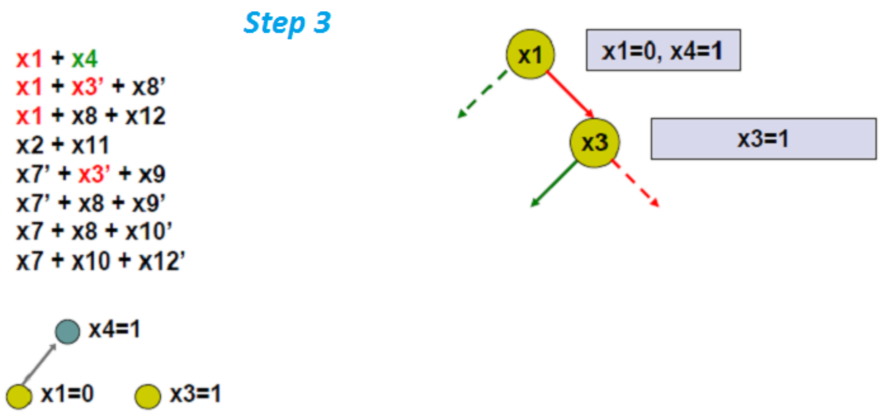
Step 2

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'



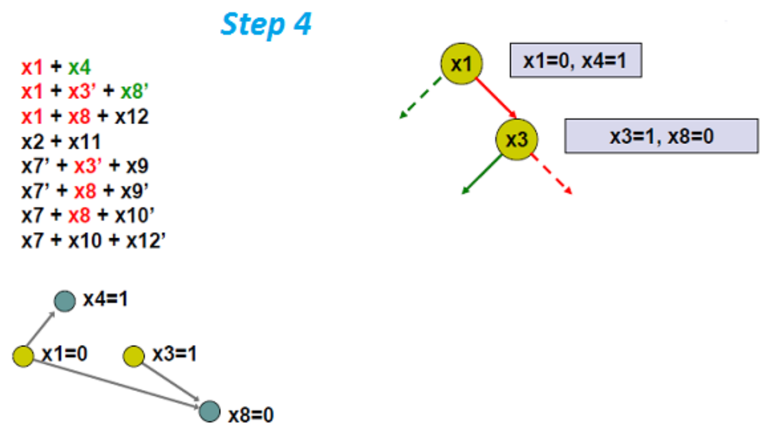
Now apply unit propagation, which yields that x_4 must be 1 (i.e. True). A gray circle means a forced variable assignment during unit propagation. The resulting graph is called an *implication graph*

CDCL algorithm : illustration



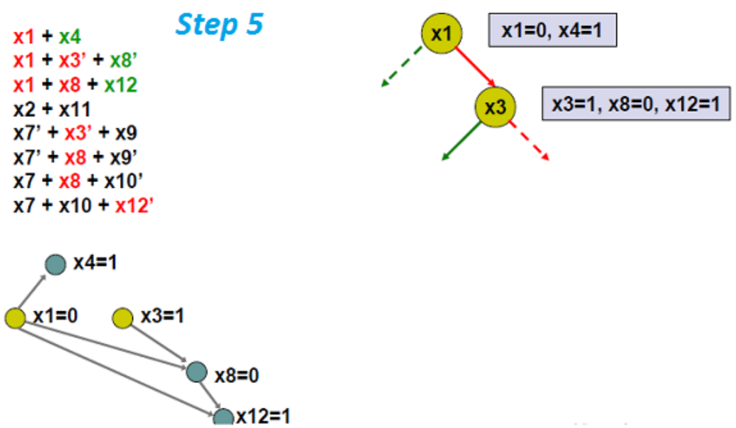
Arbitrarily pick another branching variable, x_3

CDCL algorithm : illustration



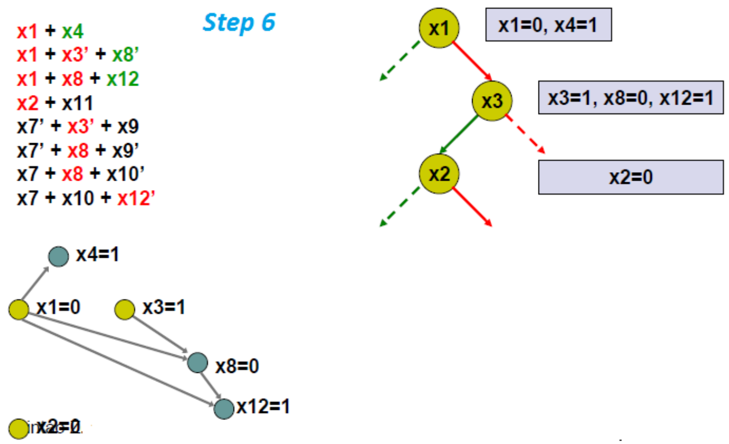
Apply unit propagation and find the new implication graph

CDCL algorithm : illustration



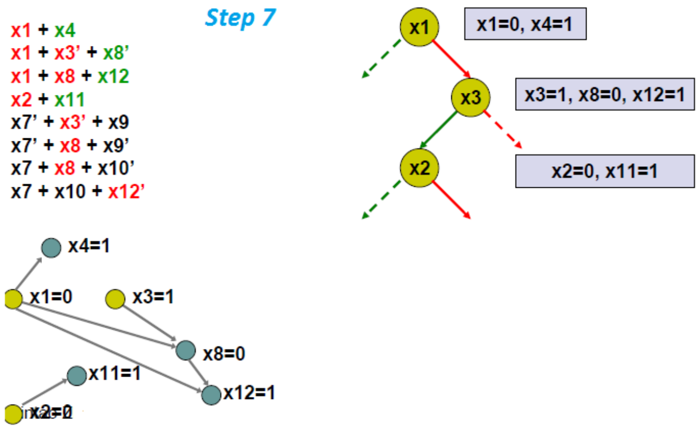
Here the variable x_8 and x_{12} are forced to be 0 and 1, respectively

CDCL algorithm : illustration



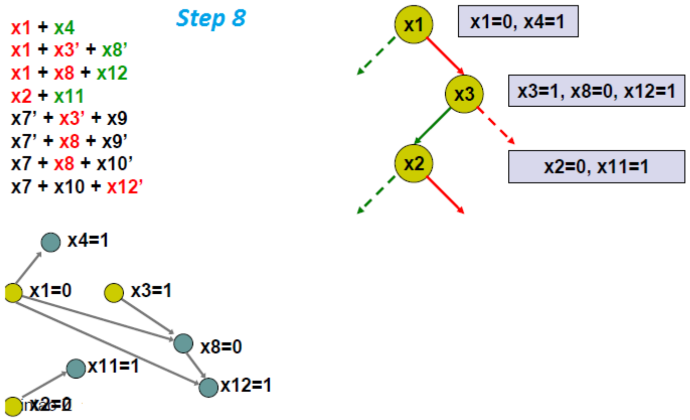
Pick another branching variable, x_2

CDCL algorithm : illustration



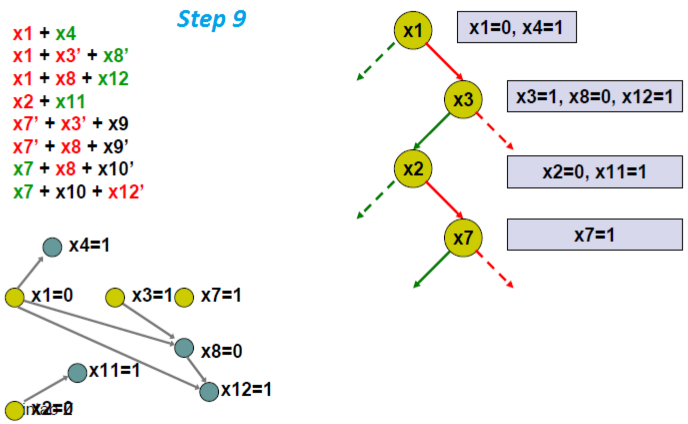
Find implication graph

CDCL algorithm : illustration



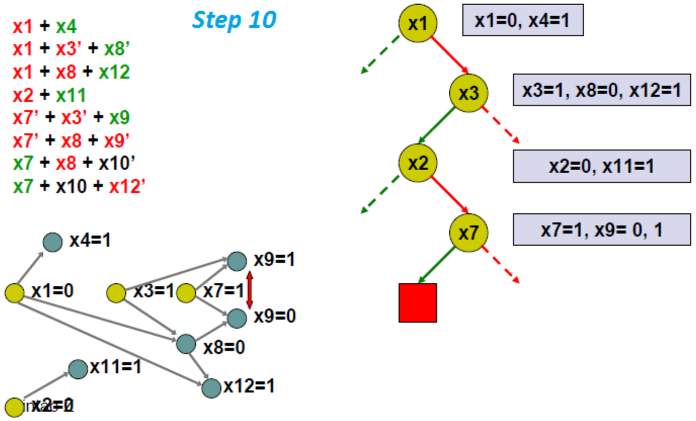
Pick another branching variable, x_7

CDCL algorithm : illustration



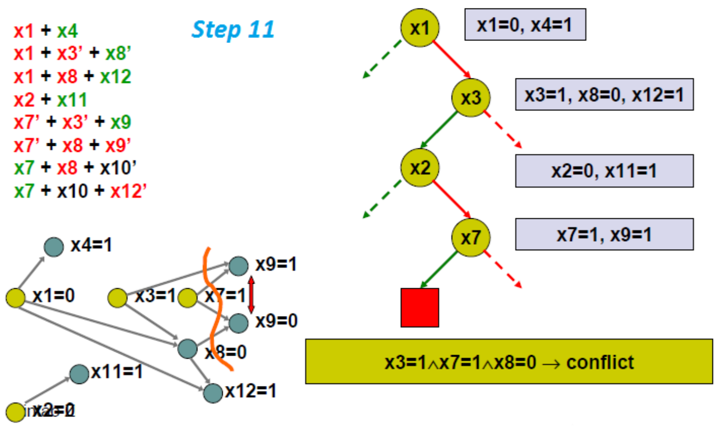
Find implication graph

CDCL algorithm : illustration



Found a conflict !

CDCL algorithm : illustration



Find the cut that led to this conflict. From the cut, find a conflicting condition

CDCL algorithm : illustration

If a implies b, then b' implies a'

Step 12

x3=1∧x7=1∧x8=0 → conflict

Not conflict → (x3=1∧x7=1∧x8=0)'

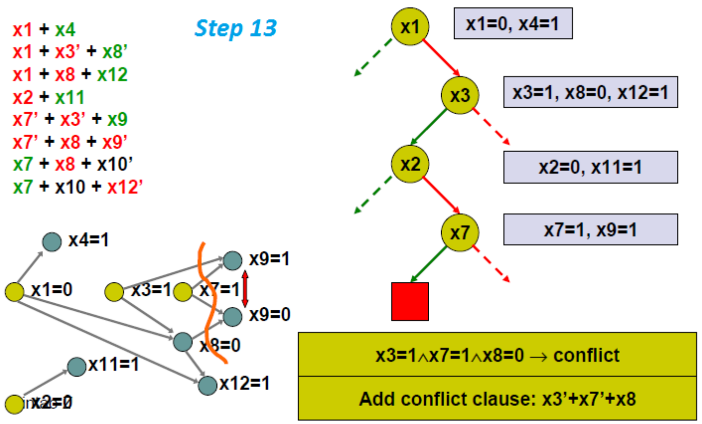
true → (x3=1∧x7=1∧x8=0)'

(x3=1∧x7=1∧x8=0)'

(x3' + x7' + x8)

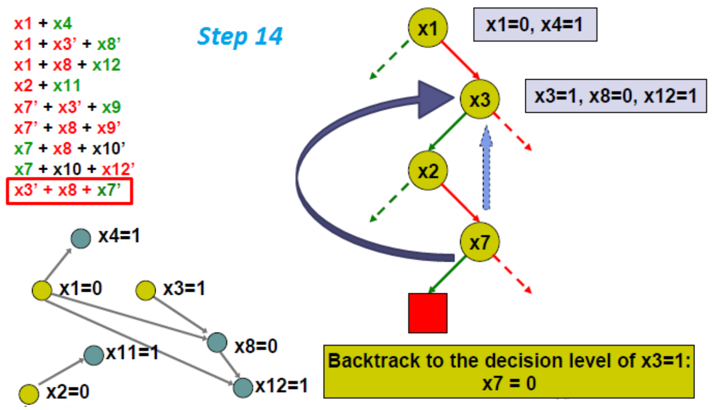
Take the negation of this condition and make it a clause

CDCL algorithm : illustration



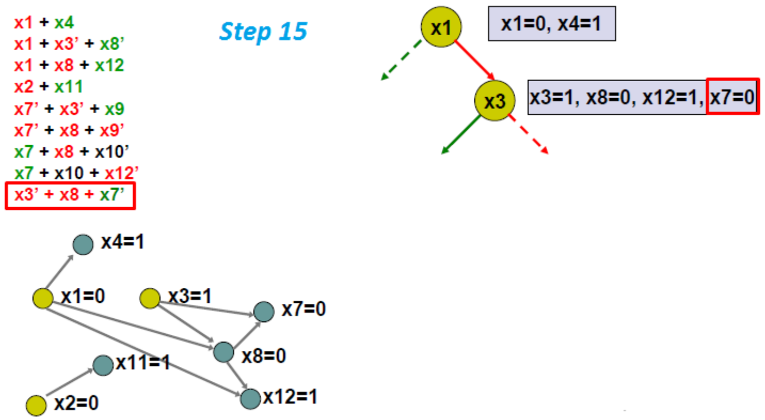
Add the conflict clause to the problem

CDCL algorithm : illustration



Non-chronological back jump to appropriate decision level, which in this case is the second highest decision level of the literals in the learned clause

CDCL algorithm : illustration



Back jump and set variable values accordingly

Contents

SAT problem and solution algorithms

Lazy Clause Generation

Representing integers with propositional variables (booleans)

- Integer x with initial domain $\{l, \ldots, u\}$
  - Bounds booleans : $[[x \leq d]]$, $l \leq d < u$
  - Equation booleans : $[[x = d]]$, $l \leq d \leq u$
- An efficient form of unary representation
- We need constraints to represent relationship among variables
  - $[[x \leq d]] \Rightarrow [[x \leq d+1]]$, $l \leq d < n-1$
  - $[[x = d]] \Leftrightarrow [[x \leq d]] \wedge \neg[[x \leq d-1]]$
- Ensures one to one correspondence between domains and assignments

Atomic constraints

- Atomic constraints define changes in domain
  - Fixing variable : $x = d$
  - Changing bound : $x \leq d$, $x \geq d$
  - Removing value : $x \neq d$
- Atomic constrains are just boolean literals
  - $x = d \Leftrightarrow [[x = d]]$
  - $x \leq d \Leftrightarrow [[x \leq d]]$
  - $x \geq d \Leftrightarrow \neg[[x \leq d]]$
  - $x \neq d \Leftrightarrow \neg[[x = d]]$

Explaining propagation

- A propagation must explain the domain changes it makes
- If $f(D) \neq D$ then propagator f returns an explanation for the atomic constraint changes

Example

- $D(x_1) = D(x_2) = D(x_3) = D(x_4) = D(x_5) = \{1, \dots, 4\}$
- $\texttt{all-different}(x_1, x_2, x_3, x_4)$
- $D(x_1) = \{1\}$ makes $D(x_2) = \{2, \dots, 4\}$
- Explanation : $x_1 = 1 \Rightarrow x_2 \neq 1$
- Implications of atomic constraints are clauses on the boolean literals :
  - $x_1 = 1 \Rightarrow x_2 \neq 1$
  - $[[x_1 = 1]] \Rightarrow \neg[[x_2 = 1]]$
  - $[[x_1 = 1]] \vee \neg[[x_2 = 1]]$
- Unit propagation on the clause will cause the change in domain

Explaining propagation : continued example

- $x_2 \leq x_5$
  - $D(x_2) = \{2, \dots, 4\}$ enforces $D(x_5) = \{2, \dots, 4\}$
  - Explanation : $x_2 \geq 2 \Rightarrow x_5 \geq 2$
- $x_1 + x_2 + x_3 + x_4 \leq 9$
  - $D(x_1) = \{1, \dots, 4\}$, $D(x_2) = \{2, \dots, 4\}$, $D(x_3) = \{3, 4\}$, $D(x_4) = \{1, \dots, 4\}$ enforces $D(x_4) = \{1, \dots, 3\}$
  - Explanation : $x_2 \geq 2 \wedge x_3 \geq 3 \Rightarrow x_4 \leq 3$
  - $x_1 \geq 1$ is not included in the explanation since this is universally true (initial domain)

Explaining failure

- When $f(D)(x) = \{\}$, failure detected
- The propagator must also *explain* failure
- $\texttt{all-different}(x_1, x_2, x_3, x_4)$
  - $D(x_3) = \{3\}$, $D(x_4) = \{3\}$ gives failure
  - Explanation : $x_3 = 3 \wedge x_4 = 3 \Rightarrow \textit{false}$
- And
  - $D(x_1) = \{1, 3\}$, $D(x_2) = \{1, 2, 3\}$, $D(x_3) = \{1, 3\}$, $D(x_4) = \{1, 3\}$
  - Explanation : $x_1 \leq 3 \wedge x_1 \neq 2 \wedge x_3 \leq 3 \wedge x_3 \neq 2 \wedge x_4 \leq 3 \wedge x_2 \neq 2 \Rightarrow \textit{false}$

Minimal explanations

- An explanation should be as general as possible. Why ?
- Sometimes there are multiple possible explanations, none better than others

Example

$D(x_1) = \{4, 6, \dots, 9\}$, $D(x_2) = \{1, 2\}$, $x_1 + 1 \leq x_2$

- $x_1 \geq 4 \wedge x_1 \neq 5 \wedge x_2 \leq 2 \Rightarrow \textit{false}$
- $x_1 \geq 4 \wedge x_2 \leq 2 \Rightarrow \textit{false}$
- $x_1 \geq 4 \wedge x_2 \leq 4 \Rightarrow \textit{false}$
- $x_1 \geq 2 \wedge x_2 \leq 2 \Rightarrow \textit{false}$

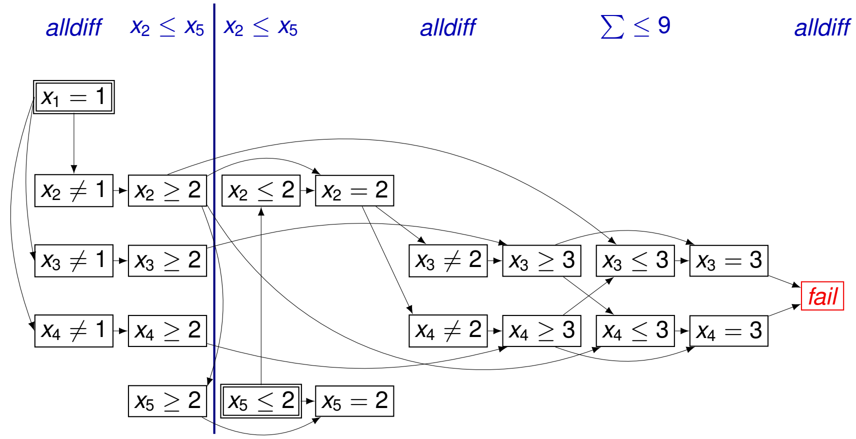
Finite Domain Propagation Example

$D(x_1) = D(x_2) = D(x_3) = D(x_4) = D(x_5) = \{1, \ldots, 4\}$

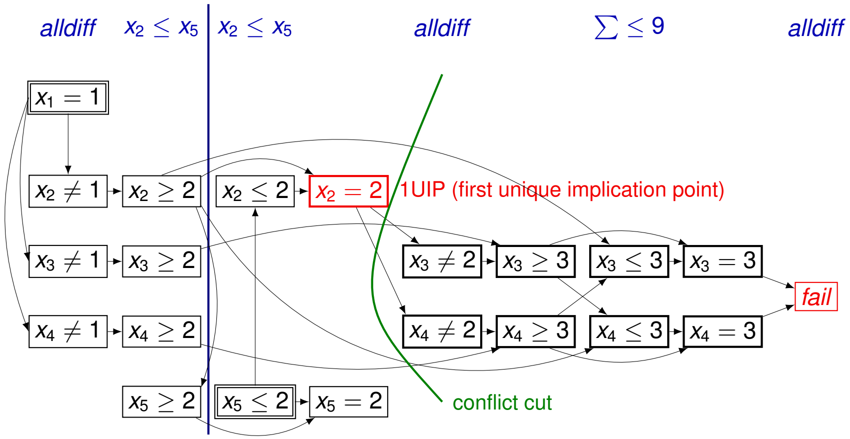
- $x_2 \leq x_5$
- `all-different`(x_1, x_2, x_3, x_4)
- $x_1 + x_2 + x_3 + x_4 \leq 9$

On the table

Finite Domain Propagation Example



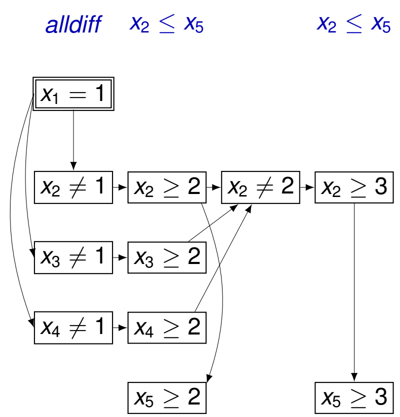
Lazy Clause Generation



Explanation : $x_2 \geq 2 \wedge x_3 \geq 2 \wedge x_4 \geq 2 \wedge x_2 = 2 \Rightarrow$ *false*

1UIP No-good (learned clause) :
$[[x_2 \leq 1]] \vee [[x_3 \leq 1]] \vee [[x_4 \leq 1]] \vee \neg[[x_2 = 2]]$

Non-chronological backtrack (backjumping)



- Backtrack to second last level in the no-good (learned clause)
- Learned clause will propagate
- We obtain smaller domains than after usual backtracking.
  - Here : $D(x_2) = \{3, 4\}$