Announcements

- Midterm 10/23
- Guest Lecture 10/02, attendance is required.
- Dafny Counter Example:
  - dafny verify --extract-counterexample file.dfy

- **Verification debugging**
  - https://dafny.org/latest/DafnyRef/DafnyRef#sec-verification-debugging

CMSC 433
Programming Language Technologies and Paradigms

DPLL (Davis-Putnam-Loveland-Logemann) Algorithm

Based on the slides from Ashutosh Gupta

DPLL Algorithm

- a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formula in conjunctive normal form (CNF).

- Davis–Putnam algorithm: Developed by Martin Davis and Hilary Putnam in 1960.

- DPLL is introduced in 1961 by Martin Davis, George Logemann and Donald W. Loveland and is a refinement of the Davis–Putnam algorithm.

Review

- Propositional satisfiability problem
  - Consider a propositional logic formula F.
  - Find a model m such that

$$m \models F .$$

- Example: Give a model of $p1 \wedge (\neg p2 \vee p3)$, find a model (satisfying assignment)
  - $m = \{p1 \rightarrow 1, p2 \rightarrow 0, p3 \rightarrow 0\}$

Review

- Propositional variables are also referred as atoms
- A literal is either an atom or its negation
- A clause is a disjunction of literals.

- Since $\vee$ is associative, commutative, and absorbs multiple occurrences, a clause may be referred as a set of literals
- Example:
  - p is an atom but $\neg p$ is not.
  - $\neg p$ and p both are literals.
  - $p \vee \neg p \vee p \vee q$ is a clause.
  - $\{p, \neg p, q\}$ is the same clause.

Conjunctive normal form(CNF)

- A formula is in CNF if it is a conjunction of clauses.
- Since $\wedge$ is associative, commutative, and absorbs multiple occurrences, a CNF formula may be referred as a set of clauses

- Example:
  - $\neg p$ and p both are in CNF.
  - $(p \vee \neg q) \wedge (r \vee \neg q) \wedge \neg r$ in CNF.
  - $\{(p \vee \neg q),(r \vee \neg q), \neg r\}$ is the same CNF formula.
  - $\{\{p,\neg q\},\{r,\neg q\},\{\neg r\}\}$ is the same CNF formula.

CNF as Input for SAT

- We assume that the input formula to a SAT solver is always in CNF.
- Tseitin encoding can convert each formula into a CNF without any blowup.
  - introduces fresh variables
- Example
  - $z = x \wedge y$ add the clause $z \leftrightarrow x \wedge y$
    - $(x \vee \neg z) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$

A Naive SAT Solver

Brute Force Case Splitting: The SAT procedure chooses an atom p from the formula F, splits it into cases p and ¬p, and recursively applies itself to the cases until the formula becomes true or false.

```
Sat(F : formula ) : bool =
   if F = ⊤ then return true
   if F = ⊥ then return false
   p = choose_atom(F)
   Ft = subst F p true
   Ff = subst F p false
   Sat Ft || tat Ff
```

Example: `Sat(p V q V ¬r)`

The Naive SAT Solver is Slow

- SAT is NP-Complete.
- The naïve algorithm will experience the worst-case runtime of 2^{n}.
- The Procedure STA may conclude the formula is satisfiable early. But for unsatisfiable formulas SAT won't terminate until it has exhausted all the possible variable assignments.

Partial Model

- Partial assignment assigns true/false values to some variables in the formula. Some variables remain unassigned.
- We will call a partial assignment of a formula F a partial model.
- Under partial model m,
  - a literal L is true if m(L) = 1 and
  - is false if m(L) = 0.
  - Otherwise, L is unassigned.

- Example:
  - Formula: $p1 \wedge (\neg p2 \vee p3)$,
  - Partial model $m = \{p1 \rightarrow 0, p2 \rightarrow 1\}$

State of a Clause

- Under partial model m
  - A clause C is true if there is $L \in C$ such that L is true and
  - C is false if for each $L \in C$, L is false.
  - Otherwise, C is unassigned.

- Example: Consider partial model $m = \{p1 \rightarrow 0, p2 \rightarrow 1\}$
  - States of the clause under m:
    - $p1 \vee p2 \vee p3$ is True

State of a Formula

- Under partial model m
  - CNF F is true if for each C∈F C is true and
  - CNF F is false if there is C∈F such that C is false.
  - Otherwise, F is unassigned.

- Example: Consider partial model m = {p1→0, p2→1}
  - States of the Formula under m:
    - $(p3 \lor \neg p1) \land \underbrace{(p1 \lor \neg p2)}_{\text{False}}$ is False

Unit Clause and Unit Literal

- C is a unit clause under m if exactly one literal L∈C is unassigned and the rest are false. L is called unit literal.

- Example
  - Consider partial model $m = \{p1 \to 0, p2 \to 1\}$
    - $p1 \lor \neg p3 \lor \neg p2$ is a Unit clause.
      - p1 and $\neg p2$ are false. p3 is unassigned.
      - p3 is the unit literal.
    - $p1 \lor \neg p3 \lor p4$ is not a Unit clause
    - $p1 \lor \neg p3 \lor p2$ is not a Unit clause

DPLL (Davis-Putnam-Loveland-Logemann) Algorithm

- DPLL
  - Maintains a partial model, initially ∅, assigns no variable.
  - Assigns an unassigned variables 0 or 1 randomly one after another
  - Sometimes forced to choose assignments due to unit literals

DPLL

DPLL(F)

// Input: CNF F   Output: sat / unsat

return DPLL(F,∅)

DPLL

```
DPLL(F,m)
  //Input: CNF F, partial assignment m     Output: sat / unsat

  if F is true under m then return sat
  if F is false under m then return unsat
```

DPLL

```
DPLL(F,m)
  //Input: CNF F, partial assignment m     Output: sat / unsat

  if F is true under m then return sat
  if F is false under m then return unsat



 …

 Choose an unassigned variable p and a random bit b ∈ {0, 1}
 if DPLL(F, m[p→b]) == sat then
   return sat
 else
   return DPLL(F, m[p→1-b])
```

DPLL

```
DPLL(F,m)
 //Input: CNF F, partial assignment m      Output: sat / unsat
  if F is true under m then return sat
  if F is false under m then return unsat

  if ∃ unit literal p under m then
    return DPLL(F,m[p→1])
  if ∃ unit literal ¬p under m then
    return DPLL(F,m[p→0])

  Choose an unassigned variable p and a random bit b ∈ {0, 1}
  if DPLL(F , m[p→b]) == sat then
    return sat
  else
    return DPLL(F, m[p→1-b])
```

DPLL

DPLL(F,m)

//Input: CNF F, partial assignment m   Output: sat / unsat

if F is true under m then return sat

if F is false under m then return unsat

Backtrack at conflict

if ∃ unit literal p under m then

return DPLL(F,m[p→1])

Unit Propagation

if ∃ unit literal ¬p under m then

return DPLL(F,m[p→0])

Choose an unassigned variable p and a random bit b ∈ {0, 1}

if DPLL(F , m[p→b]) == sat then

return sat

else

return DPLL(F, m[p→1-b])

Decision

Three actions of DPLL

- A DPLL run consists of three types of actions
  - Decision
  - Unit propagation
  - Backtracking
    - Flips its decision, continue

DPLL Example

A formula with 8 clauses and 7 variables:

$c1 = (\neg p1 \lor p2)$

$c2 = (\neg p1 \lor p3 \lor p5)$

$c3=(\neg p2 \lor p4)$

$c4=(\neg p3 \lor \neg p4)$

$c5 = (p1 \lor p5 \lor \neg p2)$

$c6 = (p2 \lor p3)$

$c7 = (p2 \lor \neg p3 \lor p7)$

$c8 = (p6 \lor \neg p5)$

DPLL Example

A formula with 8 clauses and 7 variables:

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

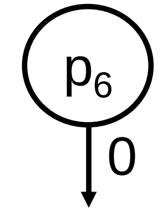
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Randomly assign p6 to be 0

Blue: Causing unit propagation

DPLL Example

A formula with 8 clauses and 7 variables:

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

c4=(¬p3 ∨ ¬p4)

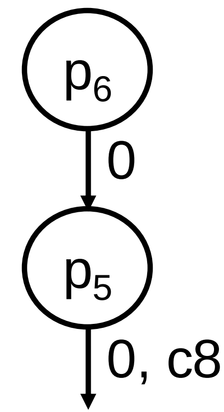
c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)

Blue: Causing unit propagation



P5 became a unit literal.

DPLL Example

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

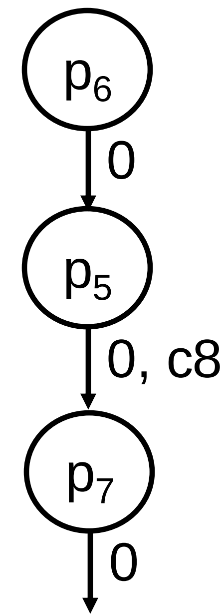
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Randomly assign p7 to be 0

Blue: Causing unit propagation

DPLL Example

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

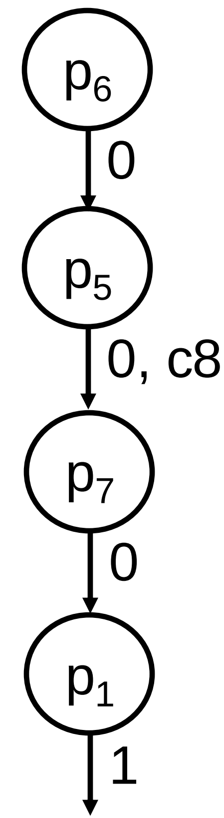
c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)

Blue: Causing unit propagation

Green: true clauses



Randomly assign p1to be 1

DPLL Example

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

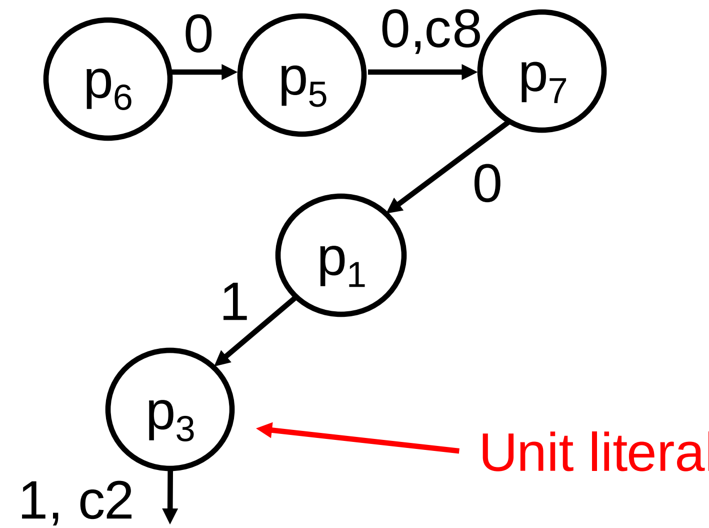
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Blue: Causing unit propagation

Green: true clauses

DPLL Example

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

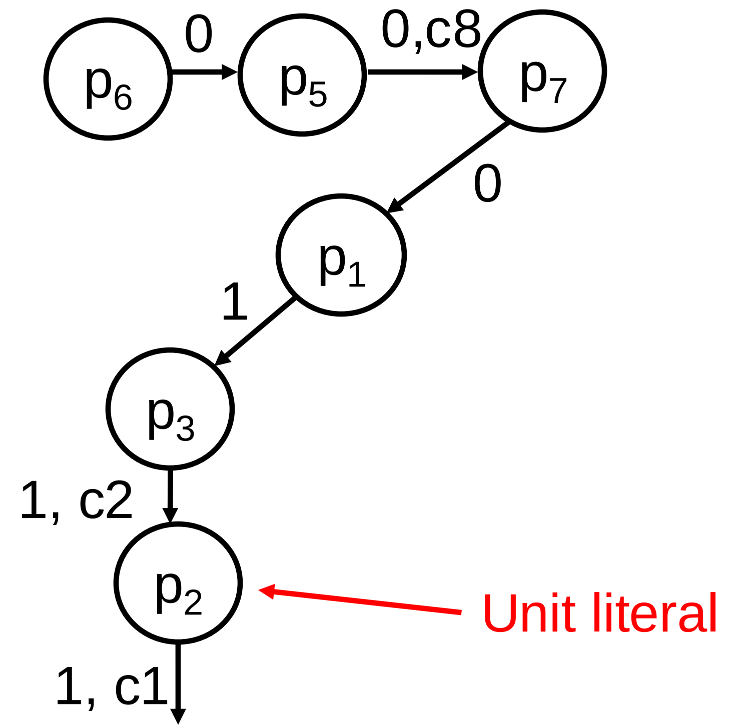
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Blue: Causing unit propagation

Green: true clauses

DPLL Example

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

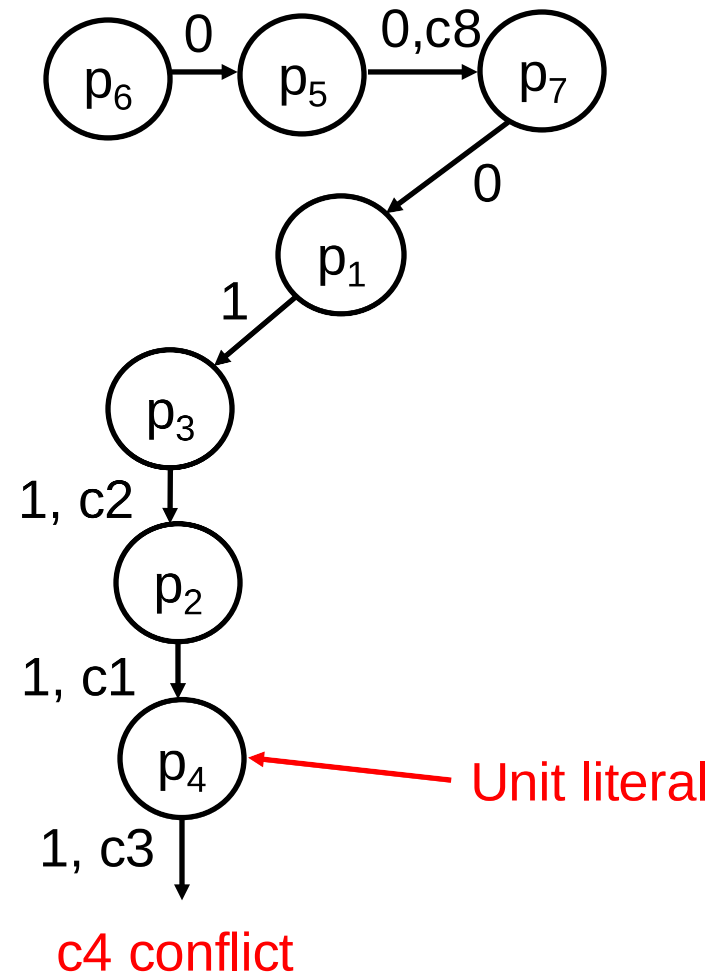
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Blue: Causing unit propagation

Green: true clauses

DPLL Example

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

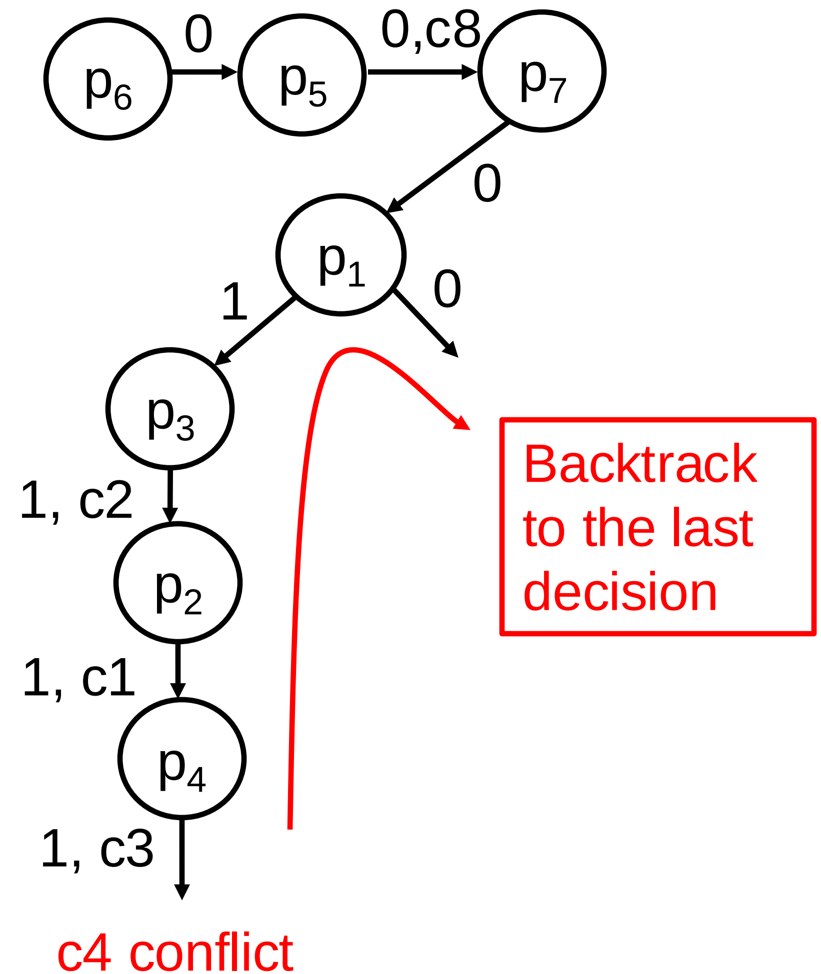
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Blue: Causing unit propagation

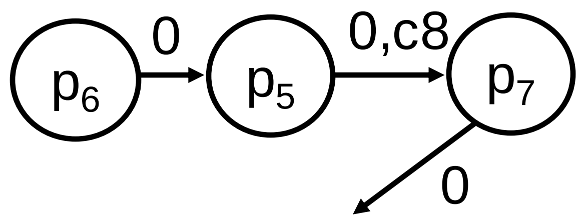
Green: true clauses

DPLL Optimizations

- DPLL allows many optimizations.
  - clause learning
  - As we decide and propagate, we construct a data structure, called implication graph, to observe the run and avoid unnecessary backtracking.

DPLL Run and Decision Level

- Run:
  - We call the current partial model a run of DPLL.
  - In the previous example, here is a run that has not reached to the conflict yet:



- Decision level
  - During a run, the decision level of a true literal is the number of decisions after which the literal was made true.
    - We write ¬p5@1 to indicate that ¬p5 was set to true after one decision.
    - Similarly, we write ¬p7@2 and ¬p6@1.

Implication Graph

- During the DPLL run, we maintain the following data structure:
  - Under a partial model m, the implication graph is a labeled DAG(N,E), where:
    - N is the set of true literals under m and a conflict node
    - E = {(L1, L2)|¬L1 ∈ *causeClause(L2)* and L2 ≠ ¬L1}
  - **causeClause(L)** :
    - clause due to which unit propagation made L true
    - ∅ for the literals of the decision variables

L1 ∨ L2
¬L1 → L2

- We also annotate each node with decision level.

Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

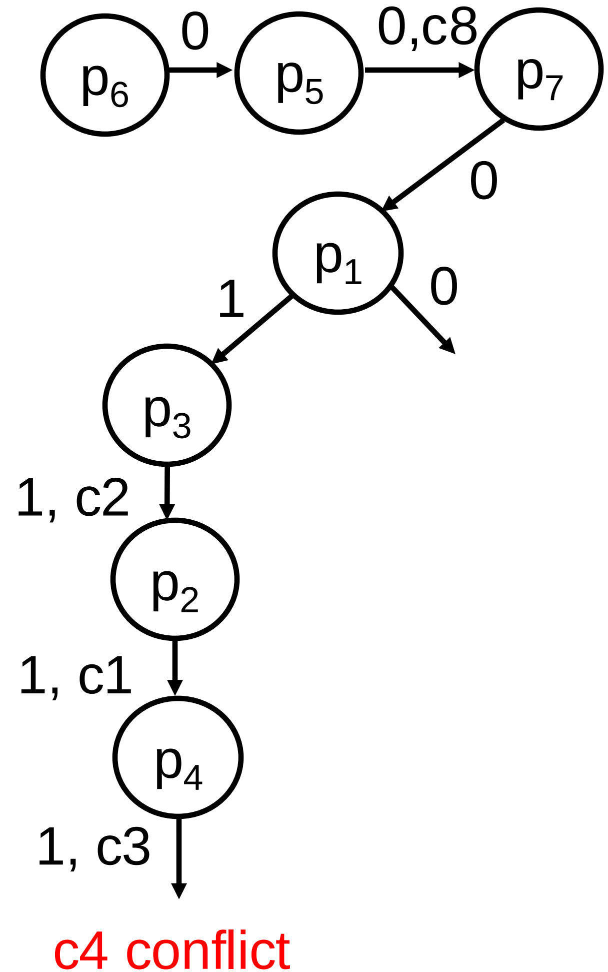
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

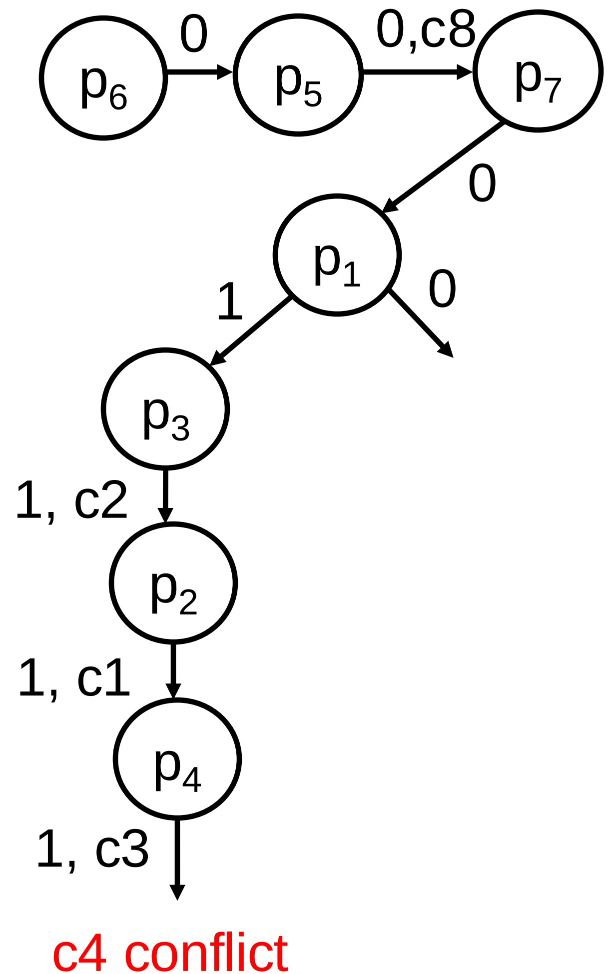
c4=(¬p3 ∨ ¬p4)

c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)



$\neg p_6 @ 1$

Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

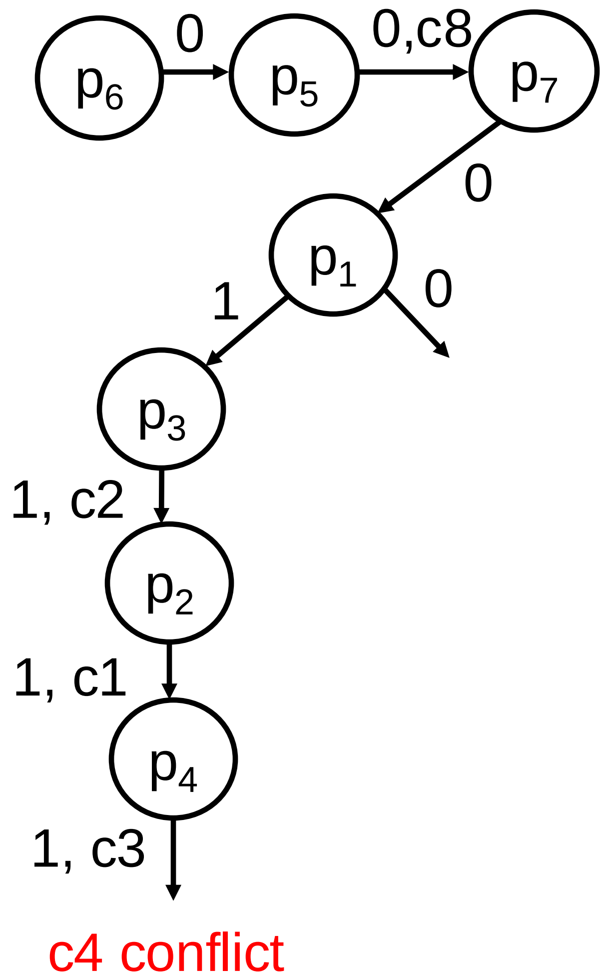
c4=(¬p3 ∨ ¬p4)

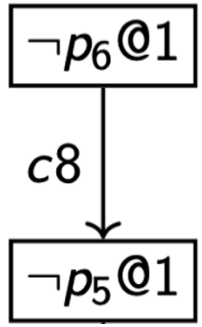
c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)





Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

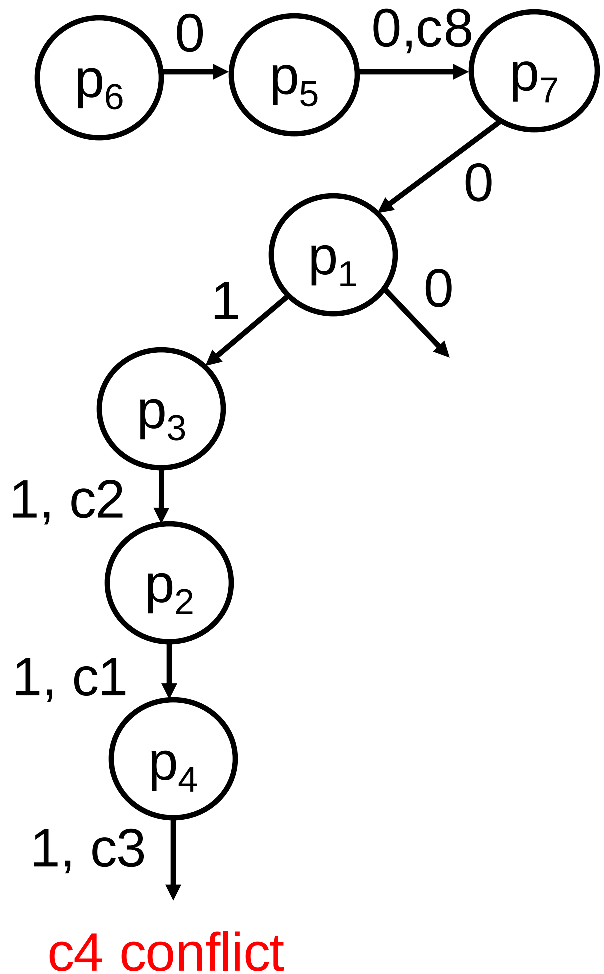
c4=(¬p3 ∨ ¬p4)

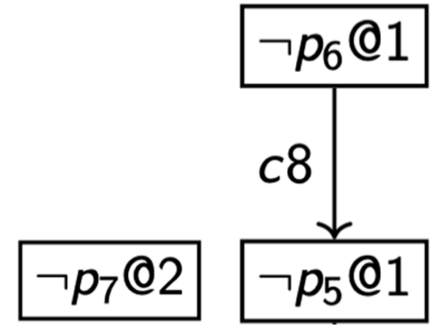
c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)





Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

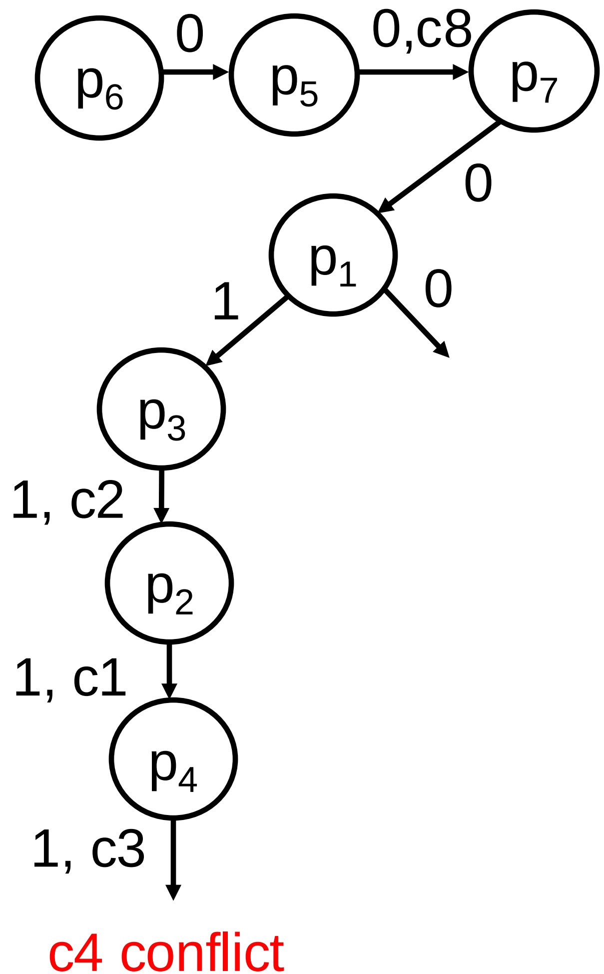
c4=(¬p3 ∨ ¬p4)

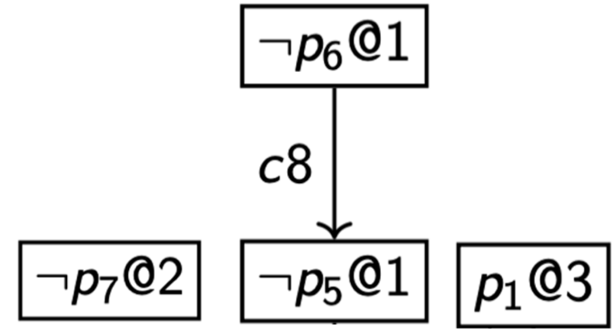
c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)





Implication Graph

$c1 = (\neg p1 \lor p2)$

$c2 = (\neg p1 \lor p3 \lor p5)$

$c3=(\neg p2 \lor p4)$

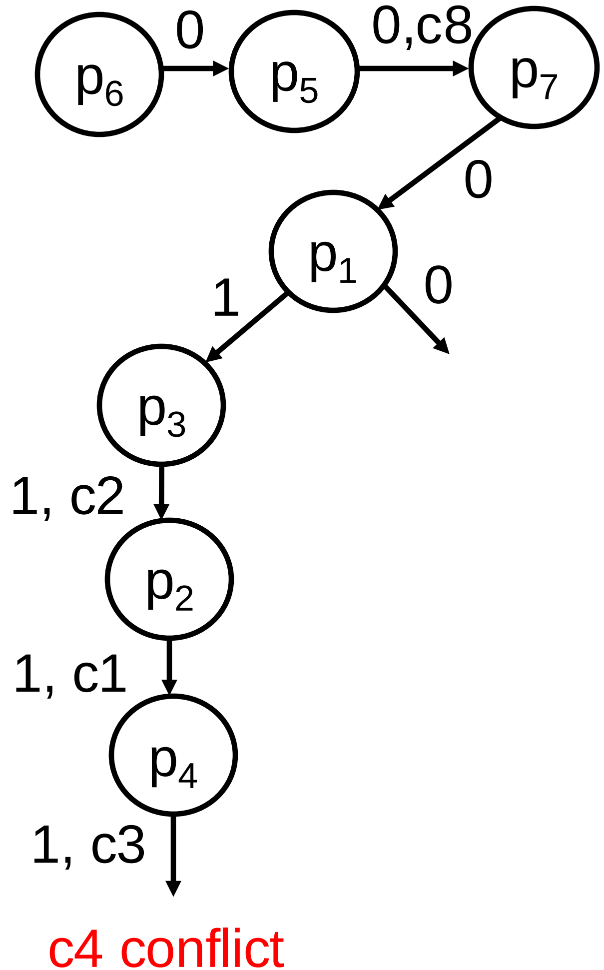
$c4=(\neg p3 \lor \neg p4)$

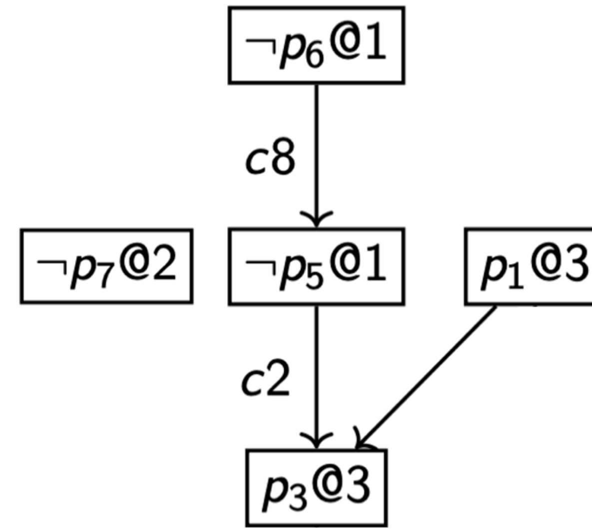
$c5 = (p1 \lor p5 \lor \neg p2)$

$c6 = (p2 \lor p3)$

$c7 = (p2 \lor \neg p3 \lor p7)$

$c8 = (p6 \lor \neg p5)$





Implication Graph

$c1 = (\neg p1 \vee p2)$

$c2 = (\neg p1 \vee p3 \vee p5)$

$c3=(\neg p2 \vee p4)$

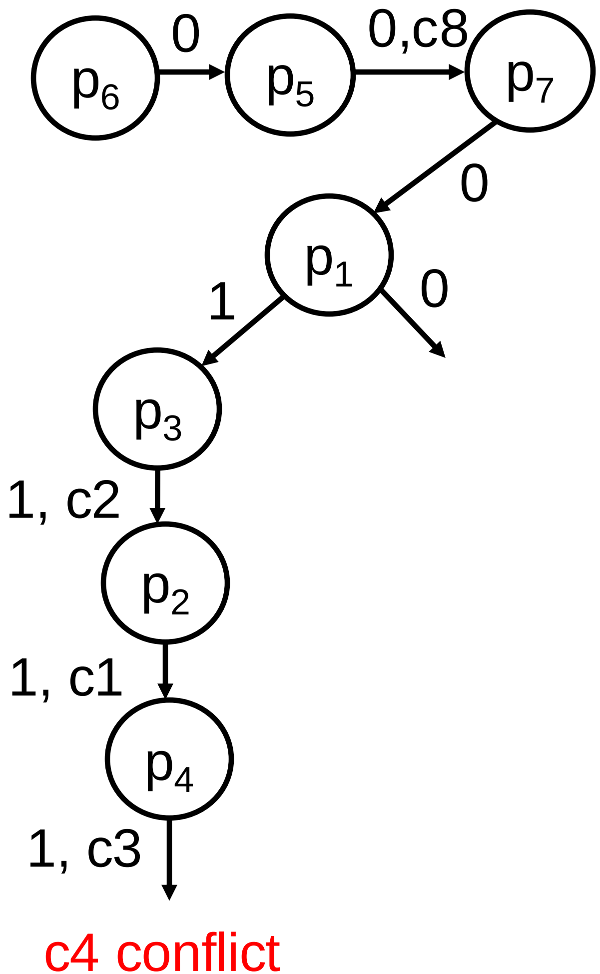
$c4=(\neg p3 \vee \neg p4)$

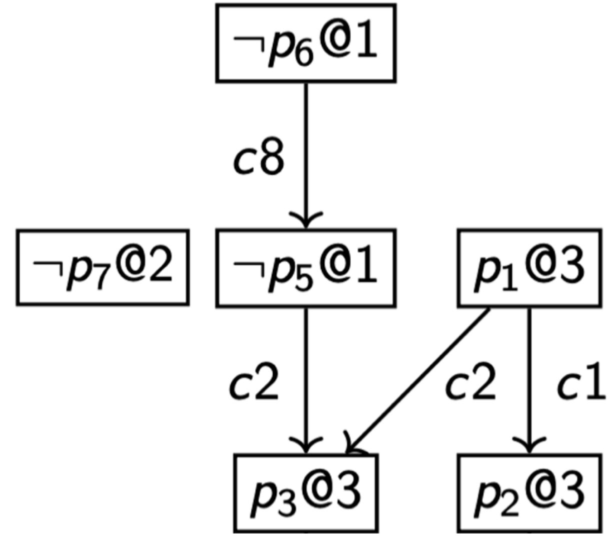
$c5 = (p1 \vee p5 \vee \neg p2)$

$c6 = (p2 \vee p3)$

$c7 = (p2 \vee \neg p3 \vee p7)$

$c8 = (p6 \vee \neg p5)$





Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

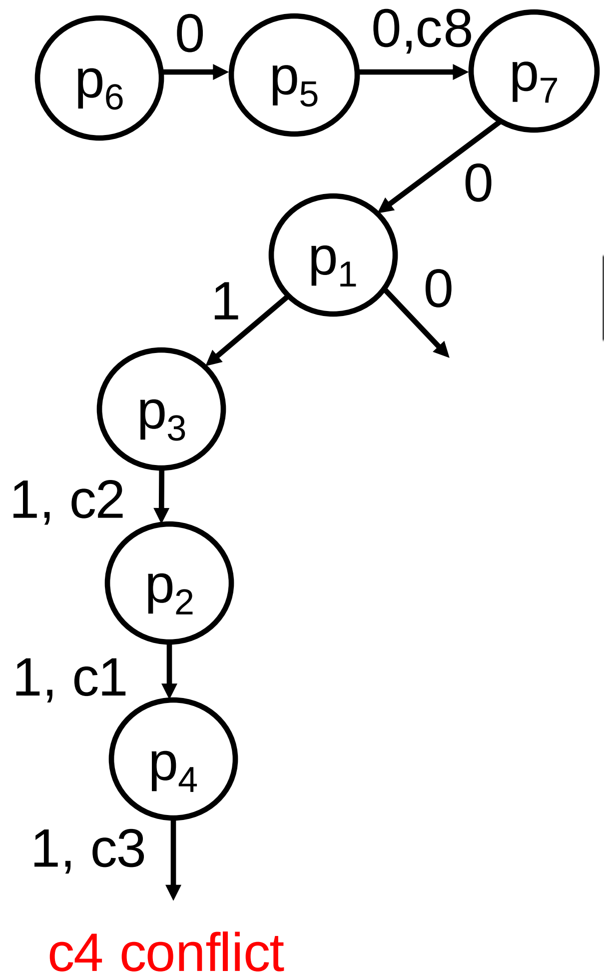
c4=(¬p3 ∨ ¬p4)

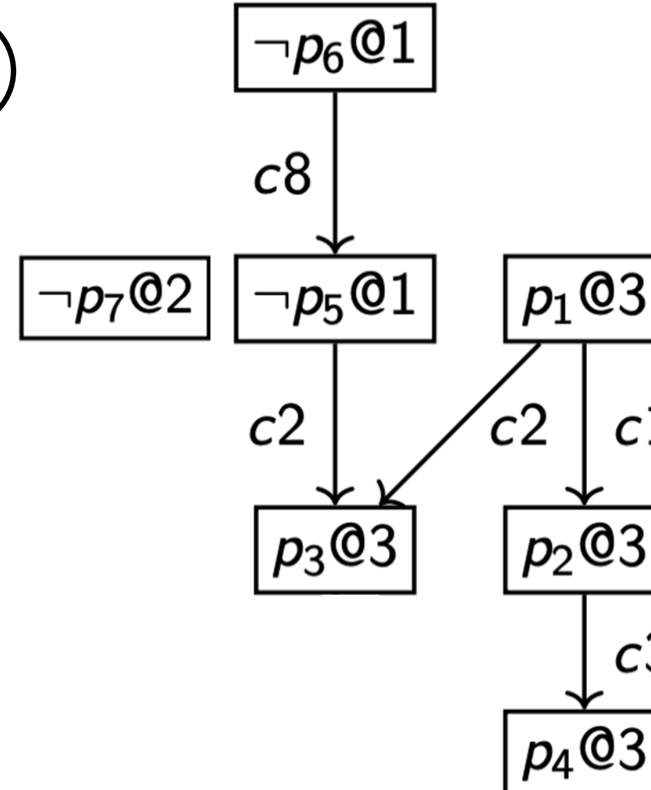
c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

c8 = (p6 ∨ ¬p5)





Implication Graph

c1 = (¬p1 ∨ p2)

c2 = (¬p1 ∨ p3 ∨ p5)

c3=(¬p2 ∨ p4)

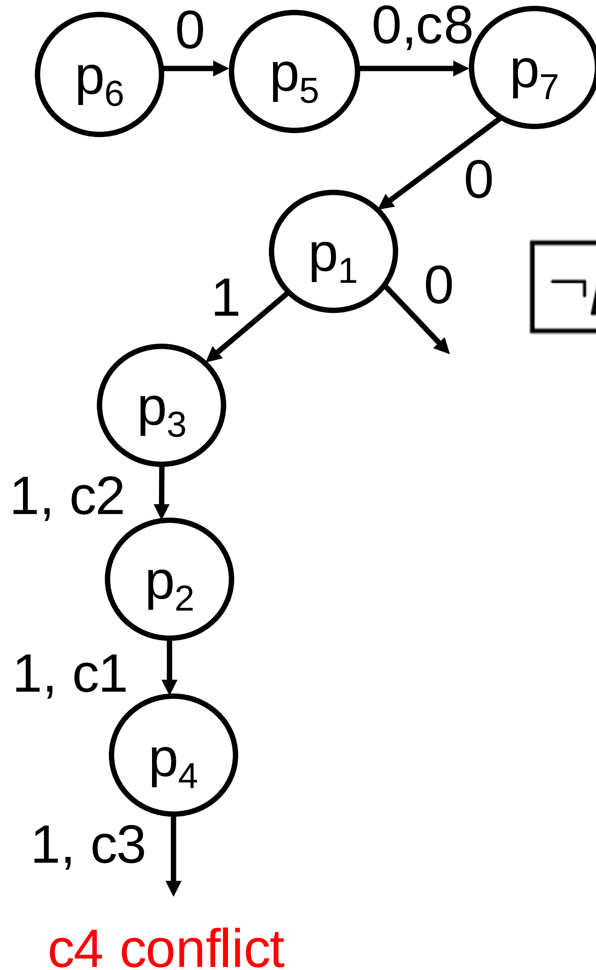
c4=(¬p3 ∨ ¬p4)

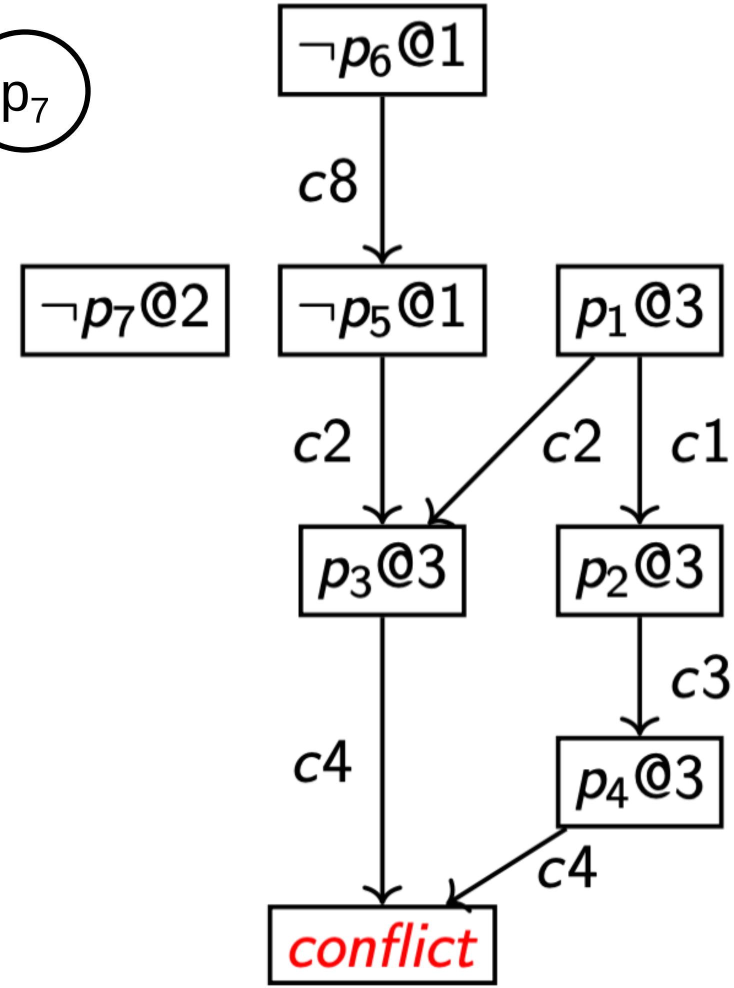
c5 = (p1 ∨ p5 ∨ ¬p2)

c6 = (p2 ∨ p3)

c7 = (p2 ∨ ¬p3 ∨ p7)

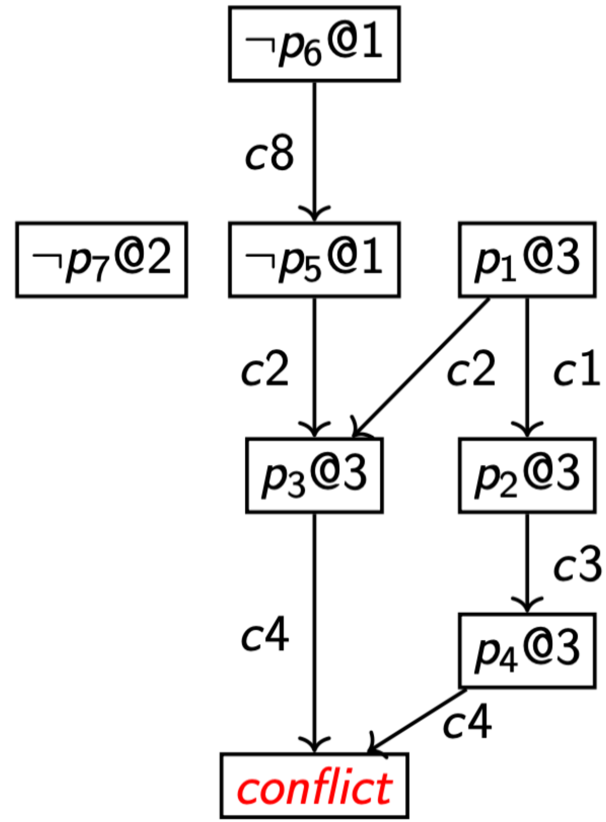
c8 = (p6 ∨ ¬p5)





Conflict Clause

- We traverse the implication graph backwards to find the set of decisions that caused the conflict.
- The clause of the negations of the causing decisions is called conflict clause.
- Example: Conflict clause: p6 ∨ ¬p1
  - p6 is set to 0 by the first decision
  - p1 is set to 1 by the third decision, literal ¬p1 is added in the conflict clause.
  - p5 decision does not contribute to the conflict, nothing is added



Clause Learning Example

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Clause learning

- Clause learning heuristics
  - add conflict clause in the input clauses and
  - backtrack to the second last conflicting decision, and proceed like DPLL

- Theorem: Adding conflict clause
  - Does not change the set of satisfying assignments
  - Implies that the conflicting partial assignment will never be tried again
- Multiple clauses can satisfy the above two conditions.
- If a clause satisfies the above two conditions, it is a conflict clause.

Clause learning:Example:

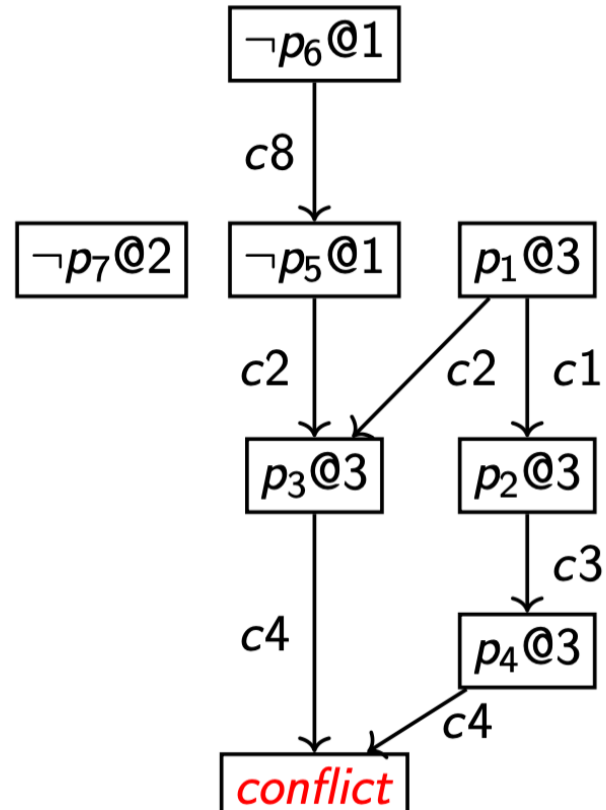
- In our running example, we added conflict clause $p6 \vee \neg p1$.

  $(\neg p6 \wedge p1) \wedge F \vDash \text{False}$

  $F \wedge \neg (\neg p6 \wedge p1)$

  $F \wedge (p6 \vee p1)$

- The second last decision in the clause is $p6 = 0$. We backtrack to it without flipping it. We run unit propagation p1 will be forced to be 0 due to the conflict clause.



Benefit of Adding Conflict Clauses

- Prunes away search space
- Records past work of the SAT solver
- Enables many other heuristics without much complications.
- Example:
  - In the previous example, we made decisions : m(p6) = 0, m(p7) = 0, and m(p1) = 1
  - We learned a conflict clause : $p6 \vee \neg p1$
  - Adding this clause to the input clauses results in
    - m(p6) = 0, m(p7) = 1, and m(p1) = 1 will never be tried
    - m(p6) = 0 and m(p1) = 1 will never occur simultaneously.

DPLL to CDCL (conflict driven clause learning)

- The optimized algorithm is called CDCL(conflict driven clause learning) instead of DPLL.

- Impact of clause learning was profound.

CDCL as an algorithm

Input: CNF F

$m := \emptyset$; $dl := 0$; $dstack := \lambda x.0$;

$m :=$ UnitPropagation(m, F);

dl stands for decision level

do

// backtracking

while F *is false under* m **do**

  if $dl = 0$ **then return** *unsat*;

  (C, dl) := AnalyzeConflict(m, F);

  $m.resize(dstack(dl))$; $F := F \cup \{C\}$;

  m := UnitPropagation(m, F);

// Boolean decision

if F *is unassigned under* m **then**

  $dstack(dl) := m.size()$;

  $dl := dl + 1$; $m :=$ Decide(m, F);

  m := UnitPropagation(m, F) ;

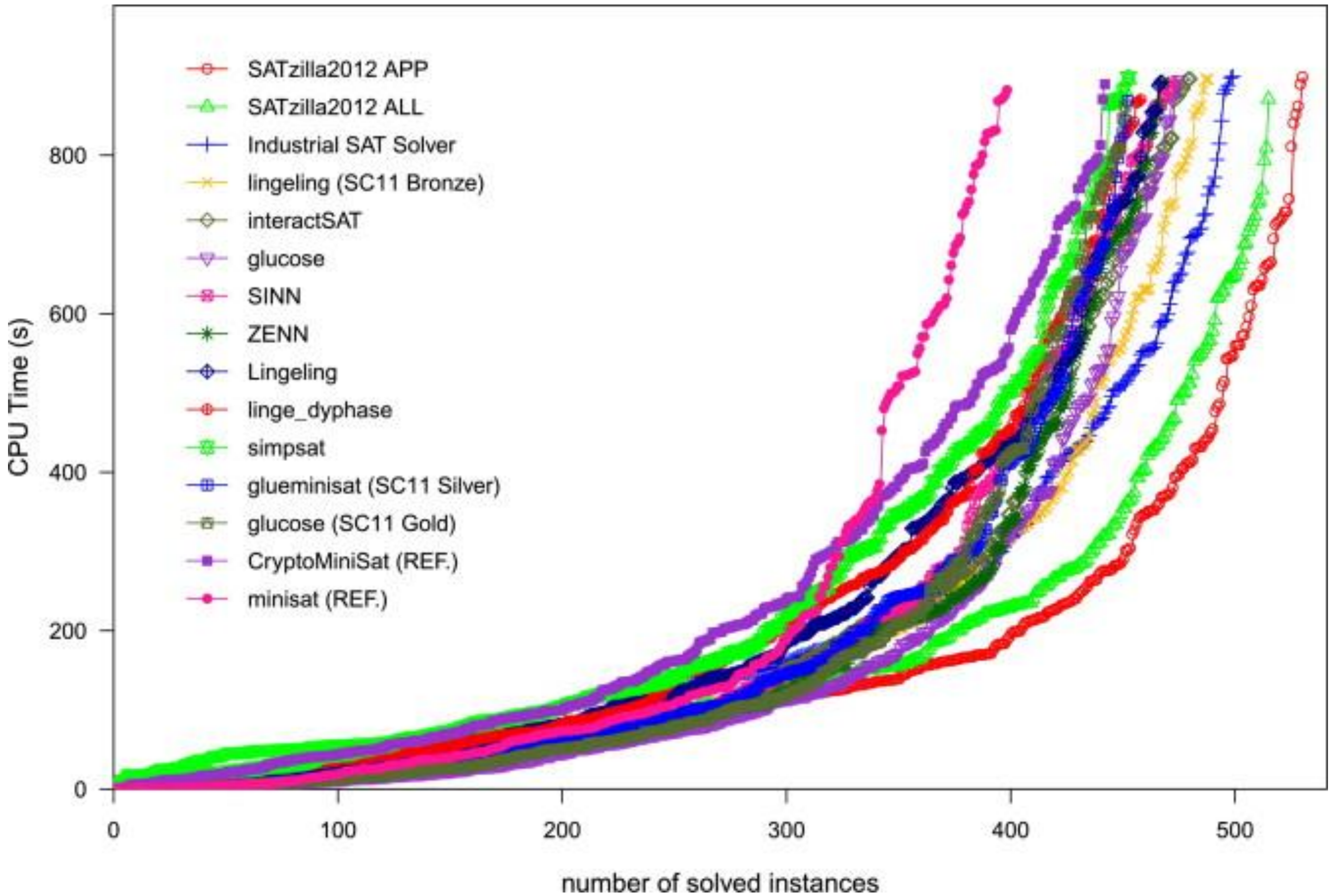
dstack records history of backtracking

while F *is unassigned or false under* m;

return *sat*

- UnitPropagation(m, F) - applies unit propagation and extends m
- AnalyzeConflict(m, F) - returns a conflict clause learned using implication graph and a decision level upto which the solver needs to backtrack
- Decide(m, F) - chooses an unassigned variable in m and assigns a Boolean value

Efficiency of SAT solvers over the years



Impact of SAT technology

- Impact is enormous.
- Probably, the greatest achievement of the first decade of this century in science after sequencing of human genome
- A few are listed here
- I Hardware verification and design assistance
  Almost all hardware/EDA companies have their own SAT solver
- I Planning: many resource allocation problems are convertible to SAT I Security: analysis of crypto algorithms
  I Solving hard problems, e. g., travelling salesman problem