Mathematical Logic 2016

Lecture 10: SAT Solvers

Instructor: Ashutosh Gupta

TIFR, India

Compile date: 2016-09-04

Where are we and where are we going?

We have seen

- definition of propositional logic and proof systems for the logic
- various properties of the proof systems
- subclasses of propositional logic
- BDD, a practical solving method for SAT

We will look into the modern SAT solvers

Topic 10.1

DPLL

DPLL(Davis-Putnam-Loveland-Logemann) framework

Before presenting the framework, let us define a few terminology

Under partial model m,

A literal ℓ is true if $m(\ell) = 1$
ℓ is false if $m(\ell) = 0$
Otherwise, ℓ is undefined.

A clause C is true if there is $\ell \in C$ s.t. ℓ is true
C is false if for each $\ell \in C$, ℓ is false
Otherwise, C is undefined.

CNF F is true if for each $C \in F$, C is true
F is false if there is $C \in F$ s.t. C is false
Otherwise, F is undefined.

Unit clause

Definition 10.1

C is a unit clause under m if exactly one literal in C is undefined and all others are false. The undefined literal is called unit literal.

DPLL (Davis-Putnam-Loveland-Logemann)

Algorithm 10.1: DPLL(F,m)

Input: CNF F, partial model m

if *F is true under m* **then**
  return *sat*

if *F is false under m* **then**
  return *unsat*

Backtracking at conflict

if *$\exists$ unit literal x under m* **then**
  return $DPLL(F, m[x \mapsto 1])$

if *$\exists$ unit literal $\neg x$ under m* **then**
  return $DPLL(F, m[x \mapsto 0])$

Choose an undefined x;

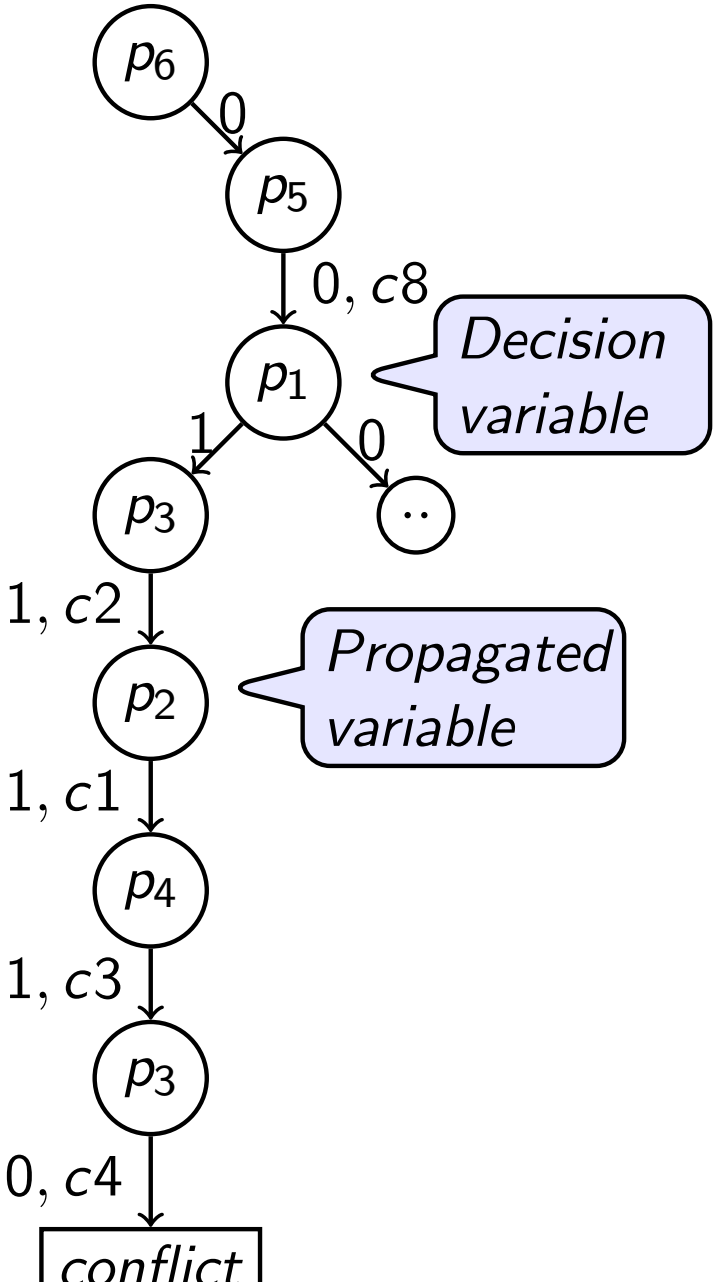
if $DPLL(F, m[x \mapsto 0]) == sat$ **then**
  return *sat*

else
  return $DPLL(F, m[x \mapsto 1])$

Example: Brancing and bracktracking in DPLL

Example 10.1

$$c_1 = (\neg p_1 \vee p_2)$$
$$c_2 = (\neg p_1 \vee p_3 \vee p_5)$$
$$c_3 = (\neg p_2 \vee p_4)$$
$$c_4 = (\neg p_3 \vee \neg p_4)$$
$$c_5 = (p_1 \vee p_5 \vee \neg p_2)$$
$$c_6 = (p_2 \vee p_3)$$
$$c_7 = (p_2 \vee \neg p_3)$$
$$c_8 = (p_6 \vee \neg p_5)$$



Exercise 10.1

Complete the DPLL run

Optimizations

There are various optimizations in implementing DPLL

We will discuss only four optimizations.

- **clause learning**
- 2-watch literals
- variable ordering
- restarts

Topic 10.2

Clause learning

Clause learning

As we decide and propagate, we may construct a data structure that allows us to do efficient back tracking.

Definition 10.2 (implication graph)

An implication graph is a labeled directed graph (N, E)*, where*

- N *contains true literals and a conflict node to denote contradiction*
- $E = \{(\ell_1, \ell_2) | \neg\ell_1 \in \mathit{clause}(\ell_2)\}$

$\mathit{clause}(\ell) \triangleq$ *clause due to which unit propagation made* ℓ *true*

Note: For decision literals $\mathit{clause}(\ell)$ *is undefined*

Note: Not same definition as defined for 2-SAT!

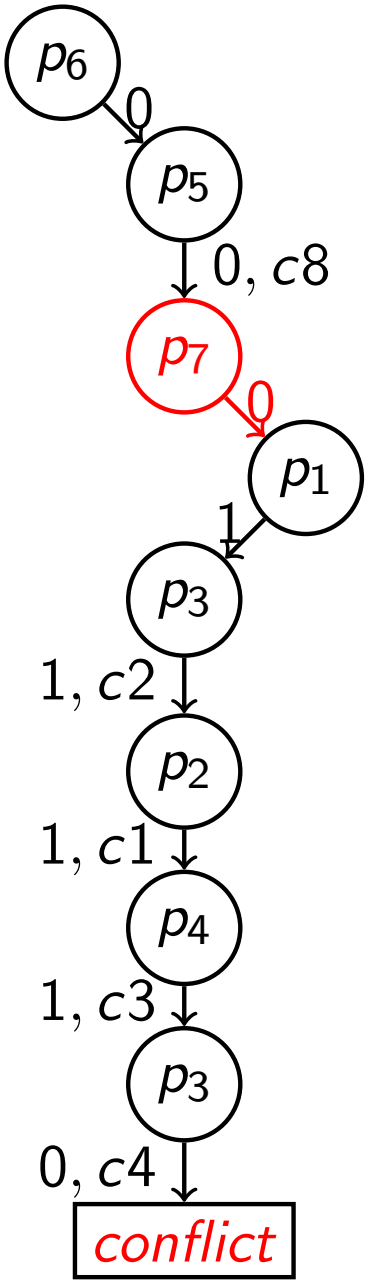
We also annotate each node with decision level (*e. g.*, $\neg p@3$), i.e., the number of decisions after which the variable was assigned

Example: implication graph

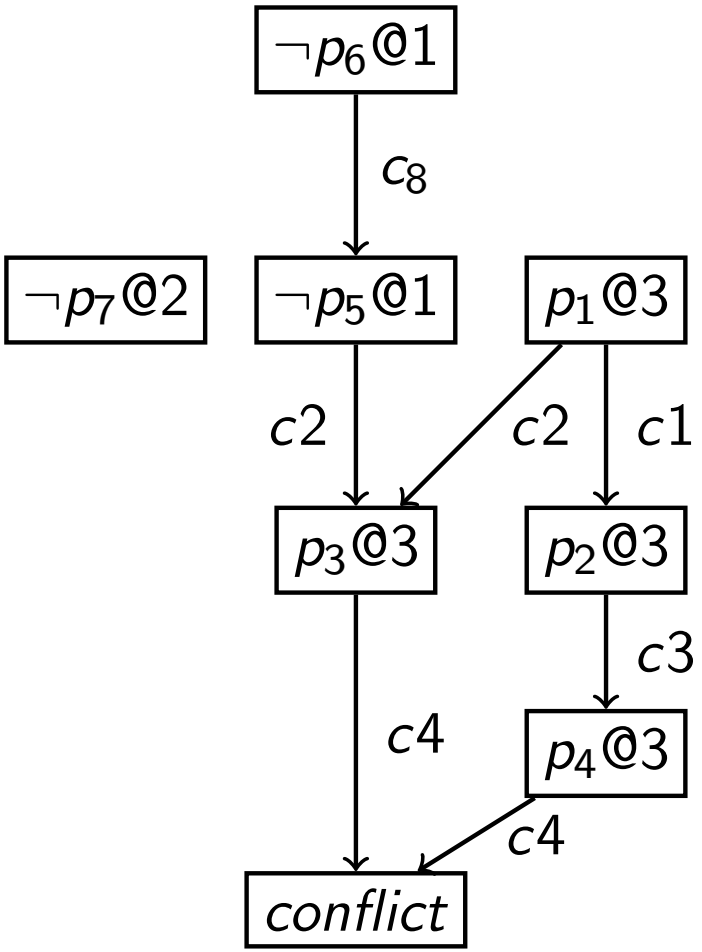
Example 10.2

$$c_1 = (\neg p_1 \vee p_2)$$
$$c_2 = (\neg p_1 \vee p_3 \vee p_5)$$
$$c_3 = (\neg p_2 \vee p_4)$$
$$c_4 = (\neg p_3 \vee \neg p_4)$$
$$c_5 = (p_1 \vee p_5 \vee \neg p_2)$$
$$c_6 = (p_2 \vee p_3)$$
$$c_7 = (p_2 \vee \neg p_3 \vee p_7)$$
$$c_8 = (p_6 \vee \neg p_5)$$

Note: Modified example



Implication graph

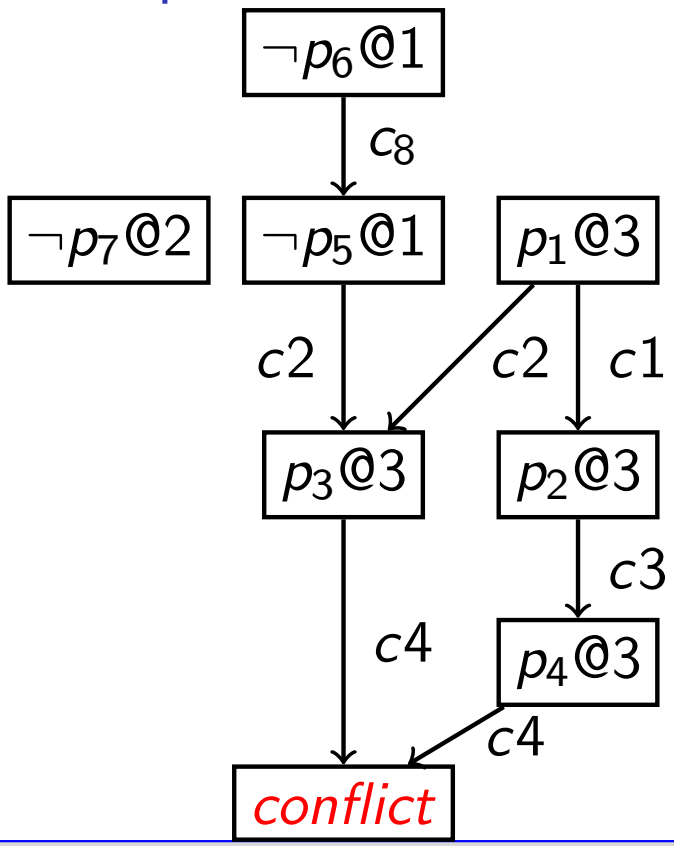


Conflict clause

In the case of conflict, we traverse the implication graph backwards to find the set of decisions that caused the conflict.

We construct a clause of the negations of the decisions, which we call conflict clause.

Example 10.3



Conflict clause : $p_6 \vee \neg p_1$

Clause learning

We add conflict clause in the original constraints and back track to the second last conflicting decision and then proceed like DPLL

Theorem 10.1

Adding conflict clause does not change the set of satisfying assignments

Theorem 10.2

Adding conflict clause implies that the conflicting partial assignment will be rejected, if tried again.

We will see that many clauses can satisfy the above two conditions.

Definition 10.3

In the following we will say if a clause satisfies the above two conditions then it is a conflict clause.

Benefit of adding conflict clauses

1. Prunes away search space
2. Records past work of the SAT solver
3. Enables very many other heuristics without much complications. We will see them shortly.

Example 10.4

In the previous example, we made decisions :
$m(p_6) = 0$, *and* $m(p_7) = 0$, *and* $m(p_1) = 1$

We learned a conflict clause : $p_6 \vee \neg p_1$

One does not have to choose the decision literals to construct a conflict clause.

Adding this clause to the input clauses results in

1. $m(p_6) = 0$, $m(p_7) = 1$, *and* $m(p_1) = 1$ *will never be tried*
2. $m(p_6) = 0$ *and* $m(p_1) = 1$ *will never occur simultaneously.*

Impact of clause learning was so profound that some people call the optimized algorithm CDCL(conflict driven clause learning) instead of DPLL

CDCL as an algorithm

Algorithm 10.2: CDCL

Input: CNF F

AddClauses(F); m := UnitPropagation(); $dl := 0$; $dstack := \lambda x.0$;

do

  // backtracking

  while $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$ **do**

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

    (C, dl) := AnalyzeConflict(m) ; // clause learning via implication graph

    $m.resize(dstack(dl))$; AddClauses($\{C\}$); m := UnitPropagation();

  // Boolean decision

  if m *is partial* **then**

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

    $dl := dl + 1$; m := Decide(); m := UnitPropagation() ;

while m *is partial or* $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$;

return *sat*

stands for decision level

dstack records history for backtracking

Choices of conflict clauses

Some choices of clauses are found to be better than others

- Smaller conflict clauses prune more search space
- Decision variables may not be the critical variables that are the center of action for producing conflict.

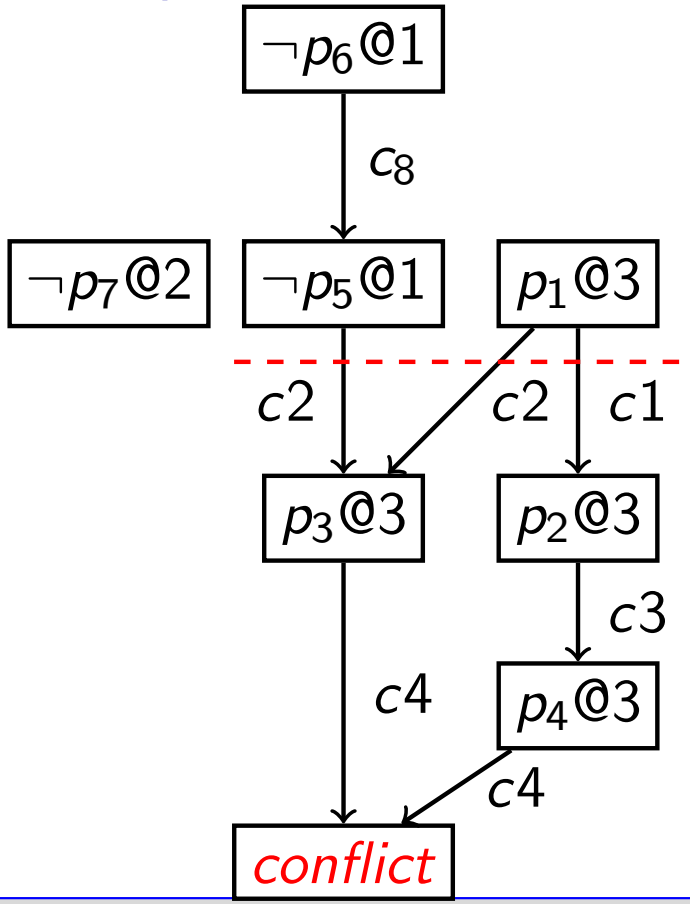
Cut clauses

Definition 10.4

Consider a cut of an implication graph that separates the decision nodes from the conflict node. Let $\ell_1, \ldots, \ell_k$ be the literals that occur at the cut boundary. The cut clause for the cut is $\neg\ell_1 \vee \cdots \vee \neg\ell_k$.

Literals that are sources of cut edges

Example 10.5



Cut clause : $\neg p_1 \vee p_5$

observation

Cut clauses may act as conflict clauses.

Exercise 10.2

Other choices for the cut clauses?

Cut clauses preserve models

Theorem 10.3

Cut clauses satisfy all the models of the original formula.

Proof.

Choose a cut.

Consider the nodes at the boundary as the decision literals.

The graph from the boundary to conflict node is a valid implication graph.(why?)

Therefore, the cut clause will satisfy all the assignments of the original formula.

□

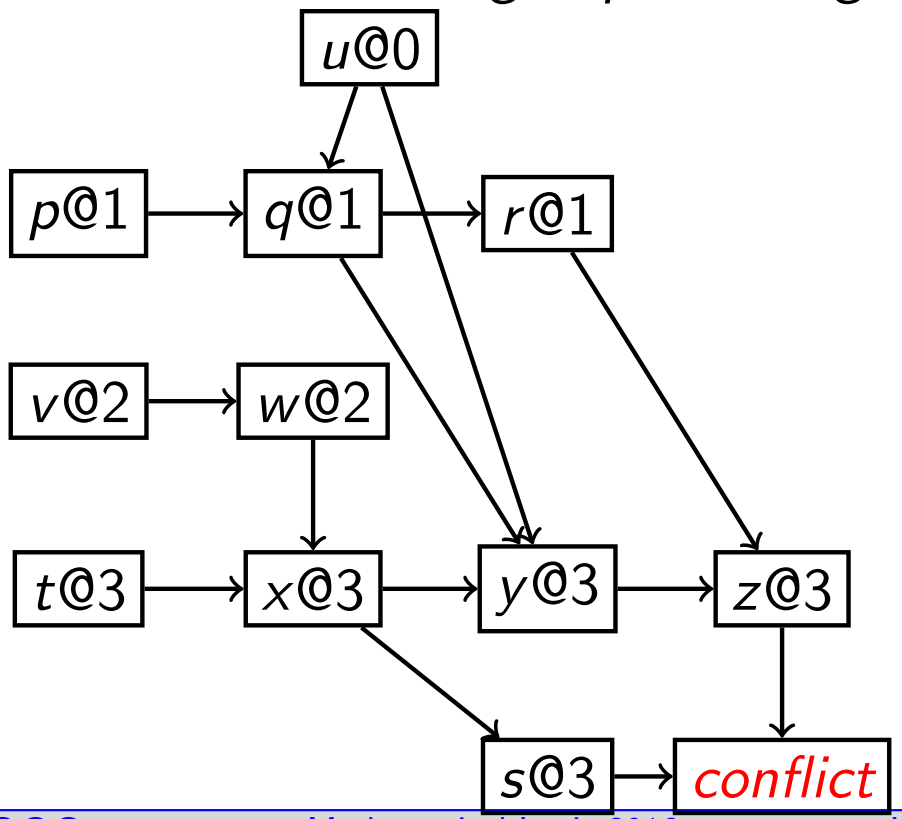
Unique implication point (UIP)

Definition 10.5

In an implication graph, a node $\ell@d'$ is a unique implication point (UIP) at decision level d if every path from dth decision literal to the conflict passes through $\ell@d'$.

Example 10.6

Consider the following implication graph *(Example source: SörenssonBiere-SAT09)*



Note: *Edges need not labeled with clauses.*

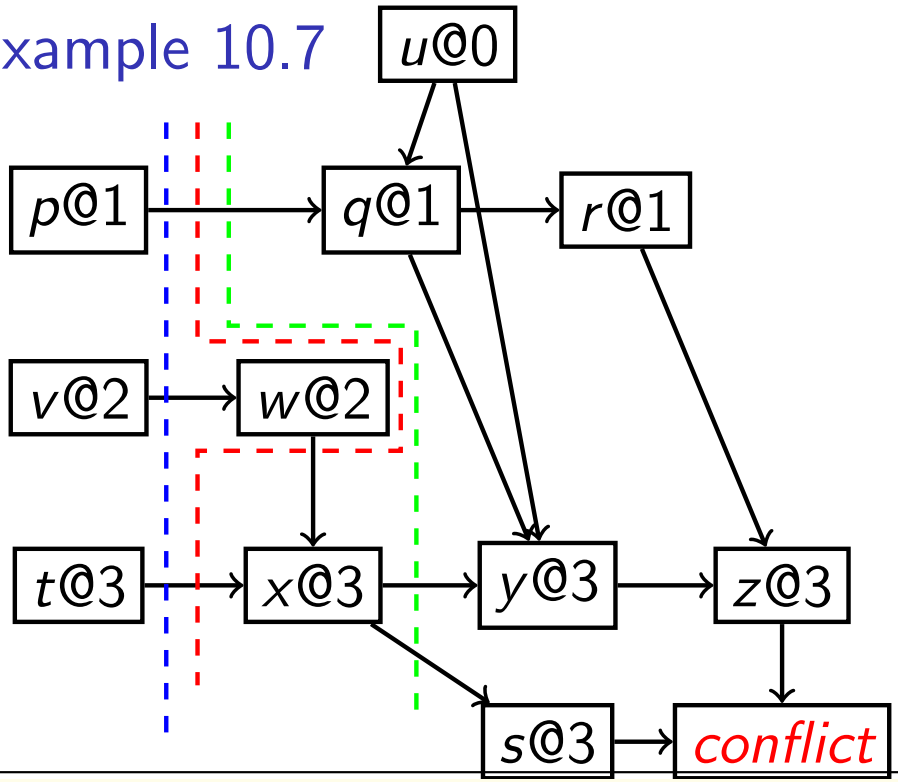
UIPs @ level 1 : $p@1, q@1$
UIPs @ level 2 : $v@2, w@2, x@3$
UIPs @ level 3 : $t@3, x@3$

First UIP strategy

Algorithm:
Iteratively replace a decision literal by one of its UIP in the conflict clause.

Preferably choose UIP that is closest to the conflict, which may result in introduction of a single UIP that replaces multiple decision literals.

Example 10.7



Conflict clause using decision literals:
$\neg p \vee \neg v \vee \neg t$

We can replace v with w
$\neg p \vee \neg w \vee \neg t$

After replacing t with x
$\neg p \vee \neg x$

Commentary: We may not always able to choose all UIPs. Since some of them are descendents of one another. A cut cannot have two nodes that have a path from one to another.

Topic 10.3

Other heuristics

2-watched literals

This data structure optimizes unit clause propagation

Observation:
To decide if a clause is ready for unit propagation, we need to look at only two literals that are not false

For each clause we choose two literals and we call them watched literals.

In a clause,

- if watched literals are non-false then the clause is not a unit clause
- if any of the two becomes false then we look for another two non-false literals
- If we can not find another two then the clause is a unit clause

Exercise 10.3

Reason why this scheme may optimize DPLL?

Example: 2-watched literals

Example 10.8

Consider clause $p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4$ *in a formula among other variables and clauses. Let us suppose initially we watch* p_1 *and* p_2 *in the clause.*

* $\triangleq$ *watched literals.*

☺ $\triangleq$ *no work to be done!*

Initially: $p_1^* \vee p_2^* \vee \neg p_3 \vee \neg p_4 \quad m = \{\}$

$\vdots$

Assign $p_1 = 0$*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots, p_1 \mapsto 0\}$

Assign $p_2 = 1$*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots, p_1 \mapsto 0, p_2 \mapsto 1\}$ ☺

Backtrack to p_1*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots\}$ ☺

Assign $p_4 = 1$*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots, p_4 \mapsto 1\}$ ☺

The benefit: often no work to be done!

Detecting pure literals

Definition 10.6

A literal ℓ is called pure in F if $\bar{\ell}$ does not occur in F.

Benefit: assign ℓ immediately 1.

As DPLL proceeds, more and more literals may become pure literals.(why?)
We may remove them similar to unit clause propagation.

However, this optimization is at odds with 2-watched literal optimization.

- In each step, 2-watched literal optimization only visits those clauses that have literals that are just assigned
- No data structure to track disabled clauses due to true literals
- Adding such data structure will defeat the benefit of 2-watched literal.

Often not implemented

Decision ordering

There are many proposed strategies for the decision order.

Important properties:
allows different order after backtracking and less overhead

A couple of famed strategies are.

- select a literal with maximum occurrences in undefined clauses
- Variable state independent decaying sum
  - each literal has a score
  - initial score based on the number of occurrences of the literals in the formula
  - score of a literal incremented whenever a new clause containing the literal is learned
  - pick the unassigned literal with the highest score, tie broken randomly
  - regularly divided the scores by a constant

Restart

SAT solvers are likely to get stuck in a local search space.

The solution is to time to time restart DPLL with a different variable ordering

- Keep learned clauses across restarts
- Slowly increase the interval of restarts such that tool becomes a complete solver (various strategies in the literature.)

Exercise 10.4

Suggest a design of a parallel sat solver.

Learned clause deletion

CDCL may learn a lot of clauses.

The solvers delete learned clauses time to time with some strategy.
For example, clauses are deleted randomly and longer ones with higher probability.

Cache aware CDCL

SAT solvers are memory intensive.

The implementation should try to make localized accesses.

Topic 10.4

Resolution proof generation from SAT solver

Evidence of unsat

If a formula is sat then SAT solver produces a model as evidence of satisfiability.

Otherwise, it produces only unsat.

Solvers should also produce a proof for unsat.

We will see how learned clause find their another use here.

Resolution Proofs

A resolution proof rule is

$$\frac{p \vee C \qquad \neg q \vee D}{C \vee D}$$

Example 10.9

Suppose $F = (p \vee q) \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg r$

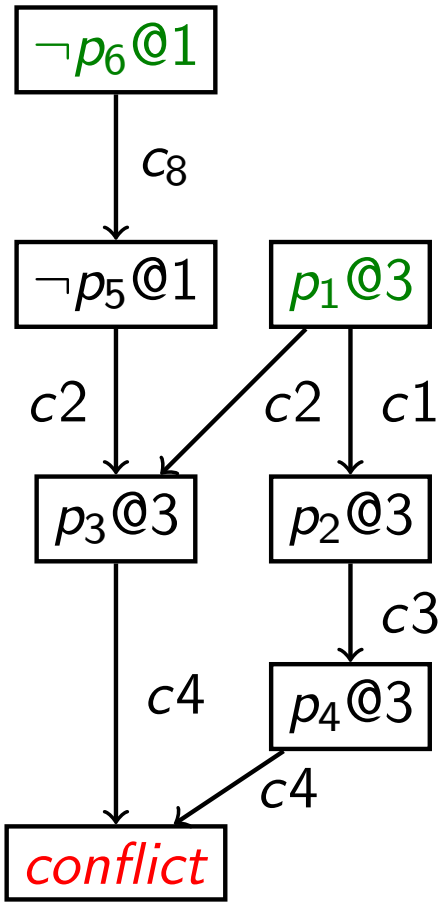
$$\dfrac{\dfrac{\dfrac{p \vee q \qquad \neg p \vee q}{q} \qquad \neg q \vee r}{r} \qquad \neg r}{\bot}$$

Reading proofs from implication graphs

- For each learned clause we assign a resolution proof that proves that the learned clause is implied by the clauses in the solver so far.

We demonstrate the process using an example.

Example 10.10



Input clauses:

$c_8 = (p_6 \vee \neg p_5)$ $\quad c_2 = (\neg p_1 \vee p_3 \vee p_5)$

$c_1 = (\neg p_1 \vee p_2)$ $\quad c_3 = (\neg p_2 \vee p_4)$ $\quad c_4 = (\neg p_3 \vee \neg p_4)$

Conflict clause : $p_6 \vee \neg p_1$

Conflict as a resolution proof:

$$\dfrac{\dfrac{\dfrac{p_6 \vee \neg p_5 \quad \neg p_6}{\neg p_5} \quad \neg p_1 \vee p_3 \vee p_5}{\neg p_1 \vee p_3} \quad p_1}{p_3} \qquad \dfrac{\dfrac{\dfrac{\neg p_1 \vee p_2 \quad p_1}{p_2} \quad \neg p_2 \vee p_4}{p_4} \quad \neg p_3 \vee \neg p_4}{\neg p_3}$$

$$\dfrac{p_3 \qquad \neg p_3}{\bot}$$

Resolution proofs for conflict clauses

Example 10.11 (contd.)

$$\dfrac{\dfrac{\dfrac{\dfrac{p_6 \vee \neg p_5 \quad \neg p_6}{\neg p_1 \vee p_3 \vee p_5 \qquad p_6 \vee \neg p_5}}{p_6 \vee \neg p_1 \vee p_3} \qquad p_1}{p_6 \vee \neg p_1 \vee p_3} \qquad \dfrac{\dfrac{\dfrac{\neg p_1 \vee p_2 \quad p_1}{\neg p_2 \vee p_4 \qquad \neg p_1 \vee p_2}}{\neg p_3 \vee \neg p_4 \qquad \neg p_1 \vee p_4}}{\neg p_1 \vee \neg p_3}}{p_6 \vee \neg p_1 \vee \bot}$$

The above is a resolution proof of the conflict clause.

One more issue:

There may be a leaf of the above proof that is a conflict clause in itself.

- In the case, there must be a resolution proof for the conflict clause.
- We "stitch" that proof on top of the above proof .

CDCL with proof generation

Algorithm 10.3: CDCL

Input: CNF F

AddClauses(F); m := UnitPropagation(); $dl := 0$; $dstack := \lambda x.0$; $proofs = \lambda C.C$;

do

  // backtracking

  while $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$ **do**

    $(C, dl, proof)$:= AnalyzeConflict(m,$proofs$); $proofs(C) := proof$;

    if $C = \emptyset$ **then return** *unsat(proof)*;

    $m.resize(dstack(dl))$; AddClauses($\{C\}$); m := UnitPropagation();

  // Boolean decision

  if *m is partial* **then**

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

    $dl := dl + 1$; m := Decide(); m := UnitPropagation() ;

while *m is partial or* $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$;

return *sat(m)*

Issues in generation proofs in SAT solvers or any solver

Proof format vs. checking

- Detailed proofs require non-trivial work from solvers, causing overhead.
- Missing details in proofs imply expensive proof checkers.

Proof minimization

- Problems of moderate size may have very large proofs
- Proofs often have redundancies
- It is wise to minimize proofs before dumping it out

Proof formats in SAT solvers

SAT solvers typically return two kinds of proofs

- List of learned clauses (low overhead)
- Resolution proofs (detailed)

Example 10.12

Input CNF

```
p cnf 3 6
-2   3  0
 1   3  0
-1   2  0
-1  -2  0
 1  -2  0
 2  -3  0
```

Learned clauses

```
-2  0
 3  0
 0
```

Resolution proof

```
1 -2  3 0 0
2  1  3 0 0
3 -1  2 0 0
4 -1 -2 0 0
5  1 -2 0 0
6  2 -3 0 0
7 -2  0 4 5 0
8  3  0 1 2 3 0
9  0  6 7 8 0
```

$$\frac{\ell_1 \vee C_1 \quad .. \quad \ell_k \vee C_k \quad \neg\ell_1 \vee .. \vee \neg\ell_k \vee D}{C_1 \vee .. \vee C_k \vee D}$$

Proof checking

A proof is a proof only if an independent checker can check it efficiently.

To check a learned clauses proof, we need to check the following things

1. Unit propagation on the input+learned clauses leads to conflict
2. Each learned clause is implied by the preceding+input clauses

Exercise 10.5

a. Give procedure for the 2nd step in the above proof checking
b. Give procedure for checking resolution proofs as presented in the the previous slide

Proof minimization

- There are several kinds of redundancies that may occur in proofs.
- We may apply several passes to minimize for each kind
- A minimization pass should preferably be a linear-time algorithm

Here we present one such case.

Redundent resolutions

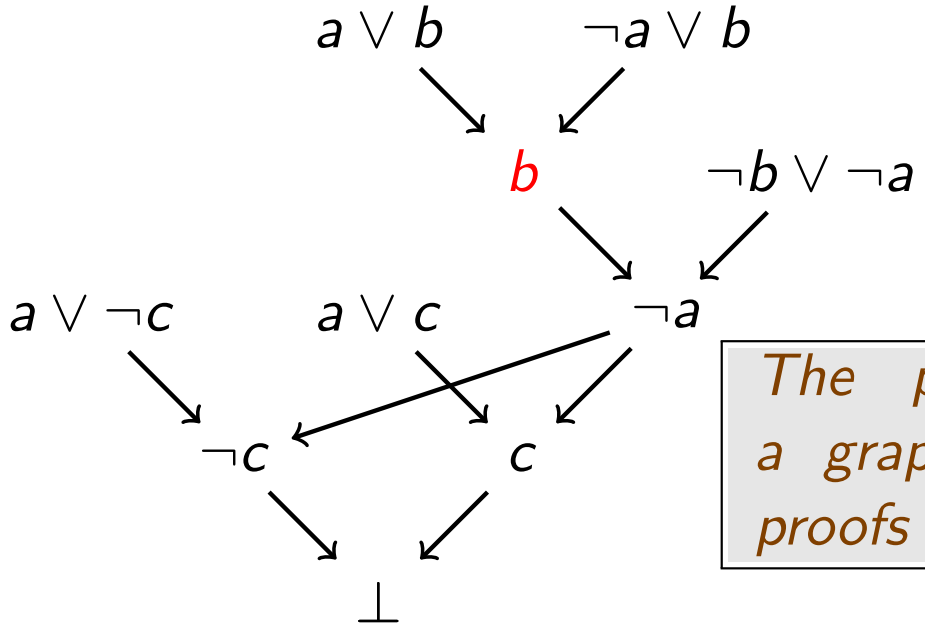
The process of resolution removes a literal in each step until none is left.
In a step, the pivot literal is removed and others may be introduced.

Definition 10.7

if a pivot is repeated in a derivation path to $\bot$, then the earlier resolution is redundant in the path.

Example 10.13

Consider the following resolution proof:



The proof is shown as a graph to illustrate that proofs are DAGs not trees.

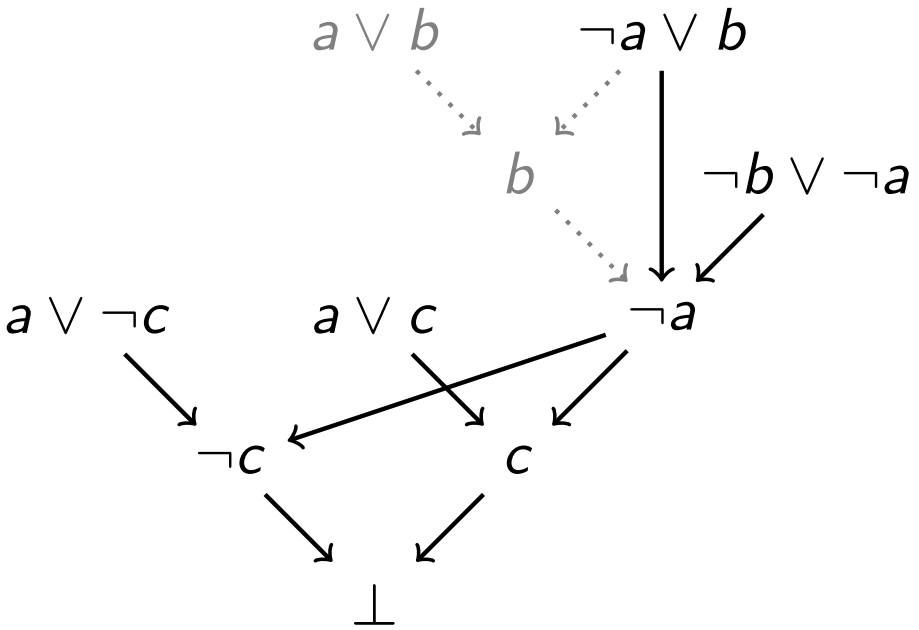
The resolution at b is redundant in both the paths to $\bot$.

Removing redundant resolution

By rewiring the proof, we may remove the redundant node v.

One of the parent of v will be wired to the children of v.

Example 10.14



After rewiring we may need to update clauses in some proof nodes.

Exercise 10.6

Which parent to choose?

Detecting redundant resolution - expansion set

Definition 10.8

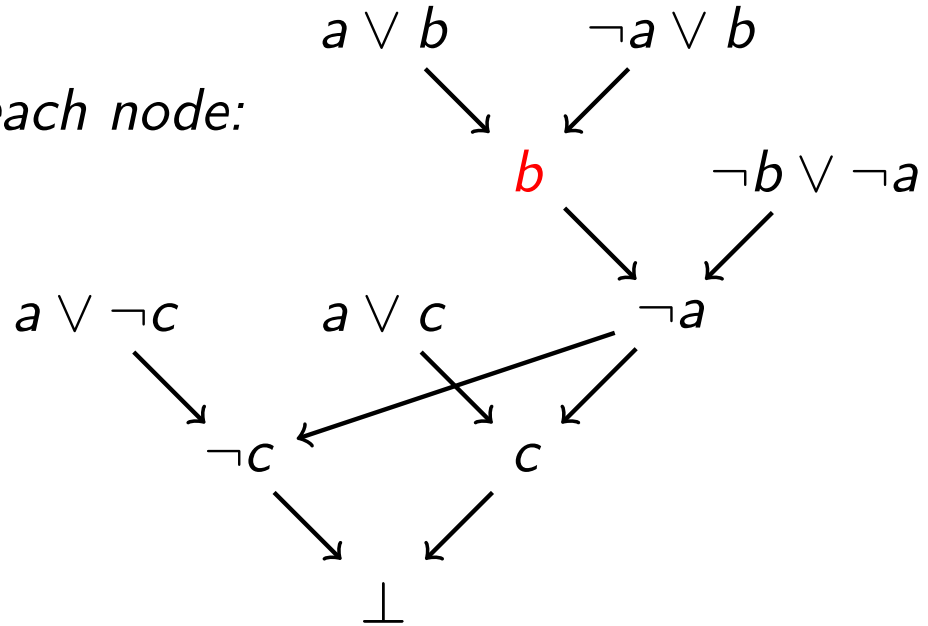
For a proof node v, expansion set $\rho(v)$ is the set of literals such that $\ell \in \rho(v)$ iff ℓ will be removed in all paths to $\bot$. ρ is defined as follows.

$$\rho(v) = \begin{cases} \emptyset & v = \bot \\ \bigcap_{v' \in children(v)} \rho(v') \cup \{rlit(v, v')\} - \{\neg rlit(v, v')\} & otherwise \end{cases}$$

where $rlit(v, v')$ is the literal involved on the edge (v, v').

Exercise 10.7

Calculate $\rho(v)$ for each node:



Detecting redundant resolution (contd.)

Theorem 10.4

If $pivot(v)$ or $\neg pivot(v) \in \rho(v)$ then v is redundant.

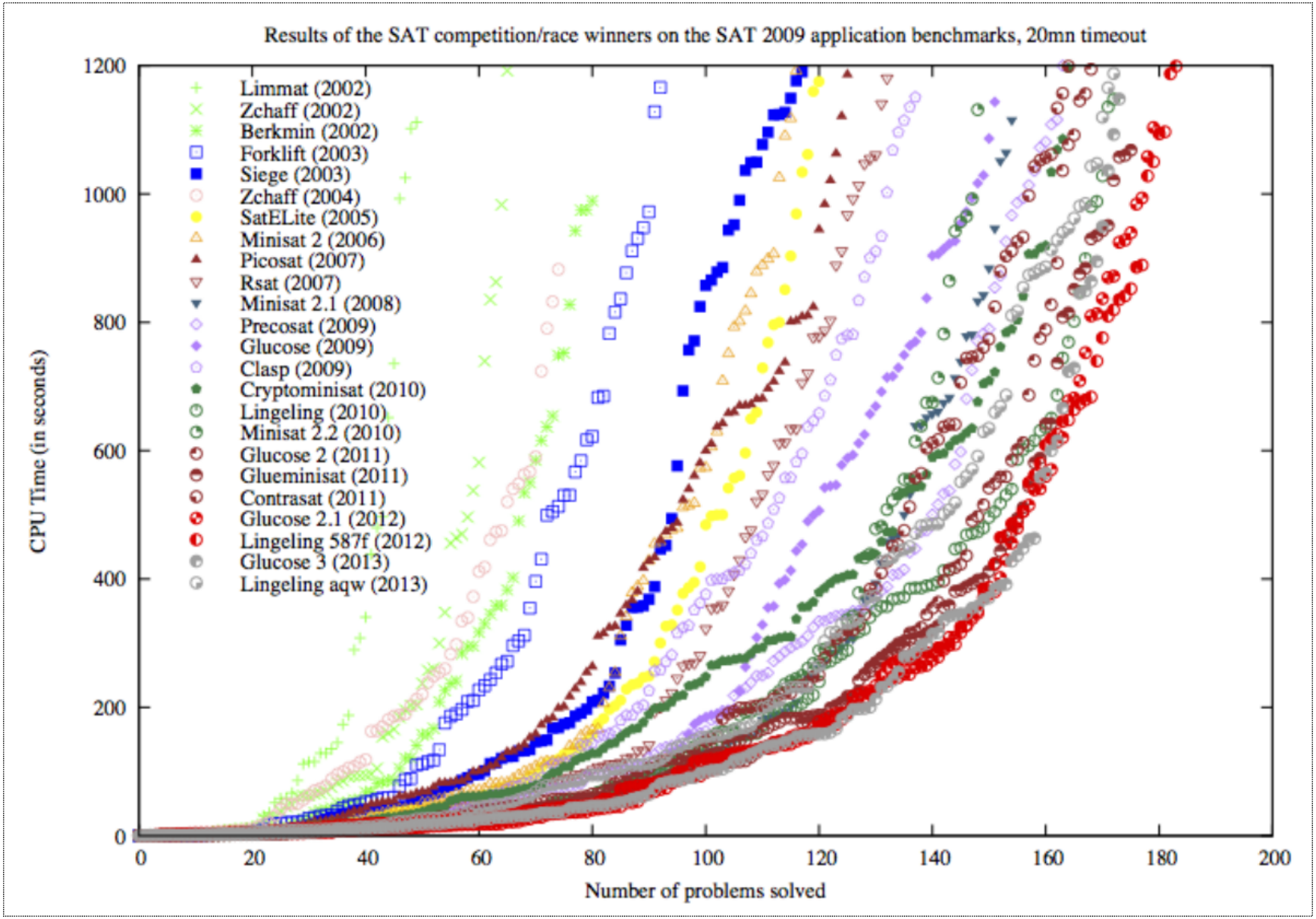
Exercise 10.8

a. *What is the complexity of computing ρ?*
b. *Prove $\rho(v) \supseteq$ literals in v*
c. *Given the above observation suggest an heuristic optimization.*

Topic 10.5

SAT technology and its impact

Efficiency of SAT solvers over the years



Source: `http://satsmt2014.forsyte.at/files/2014/07/SAT-introduction.pdf`

Impact of SAT technology

Impact is enormous.

Probably, the greatest achievement of the last decade in science after sequencing of human genome

A few are listed here

- Hardware verification and design assistance
  Almost all hardware/EDA companies have their own SAT solver
- Planning: many resource allocation problem is convertible to SAT problem
- Security: analysis of crypto algorithms
- Solving hard problems, *e. g.*, travelling salesman problem

Latest trends in SAT solving

- Portfolio solvers
- Learned clause management
- Optimizations for applications, e.g., maxsat, unsatcore, etc.

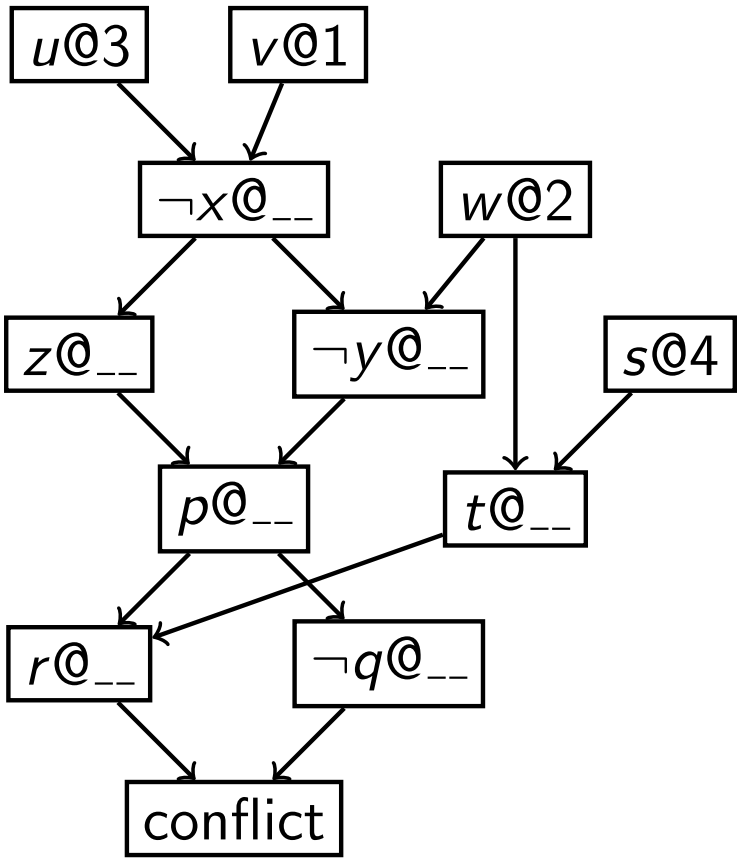
Topic 10.6

Problems

UIP

Exercise 10.9

Consider the following implication graph generated in a CDCL solver.



a. *Assign decision level to every node (write within the node)*
b. *Write unique implication points(UIPs) for each level*
c. *Give the conflict clause that is learned by first UIP strategy.*

Lovàsz local lemma vs. SAT solvers

Theorem 10.5 (Lovàsz local lemma)

Let ϕ be a k-CNF formula with all clauses of size k. If each variable in ϕ occurs less than $2^{k-2}/k$ times then ϕ is sat.

Definition 10.9

A Lovàsz k-CNF formula is a k-CNF formula that has all variables occur $\frac{2^{k-2}}{k} - 1$ times, and for each variable p, p and $\neg p$ occur nearly equal number of times.

Exercise 10.10

- *Write a program that generates uniformly random Lovàsz k-CNF formula*
- *Generate 10 instances for $k = 3, 4, 5, ...$*
- *Solve the instances using some sat solver*
- *Report a plot k vs. average run times*

Commentary: There are many sat solvers available online. Look into the following webpage of sat competition to find a usable and downloadable tool. `http://www.satcompetition.org`. Please discuss with the instructor if there is any confusion.

End of Lecture 10