Satisfiability Checking

05 SAT solving

Prof. Dr. Erika Ábrahám

RWTH Aachen University
Informatik 2
LuFG Theory of Hybrid Systems

WS 22/23

Satisfiability problem

CNF: c1 ∧ c2 ∧ ⋯ cm (c1, c2, ...为 clause)

Given:

- Propositional logic formula φ in CNF.

Question:

- Is φ satisfiable?

  (Is there a model for φ?)

  model: 可以使得φ 为真的assignment

SAT 问题的求解算法大致可以分为两类：完备性算法和非完备性算法

完备性算法，由于其完备性的搜索技术，能判定一个 SAT 实例是可满足的还是不可满足的, 但有可能不能在合理的时间对 SAT 实例进行判定。非完备性算法，通常指局部搜索算法，仅能判定一个 SAT 实例是可满足的，但其能非常高效地对可满足的 SAT 实例进行.

1. Exploration (also called enumeration) all possibility of assignment
   完备性算法
2. Boolean constraint propagation (BCP)
3. Conflict resolution and backtracking
4. Exploration revisited

Exploration algorithm

```
bool explore(CNF_Formula φ){
    trail.clear(); //stack of entries (x, v, b) assigning value v to proposition x
                   //and a flag b stating whether ¬v has already been processed for x
    while (true) {
        if (!decide()) {
            if all clauses of φ are satisfied by the assignment in trail then return SAT;
            else if (!backtrack()) then return UNSAT
        }
    }
}
bool decide() {
    if (all variables are assigned) then return false;
    choose unassigned variable x not yet in trail;
    choose value v ∈ {0, 1};
    trail.push(x, v, false);
    return true
}
bool backtrack() {
    while (true){
        if (trail.empty()) then return false;
        (x, v, b):=trail.pop()
        if (!b) then { trail.push((x, ¬v, true)); return true }
    }
}
```

trail为栈,存储树的路径

flip or not 是否翻转过值

回溯到上一节点

如果栈空了, 则说明已无法回溯, 返回false

弹栈

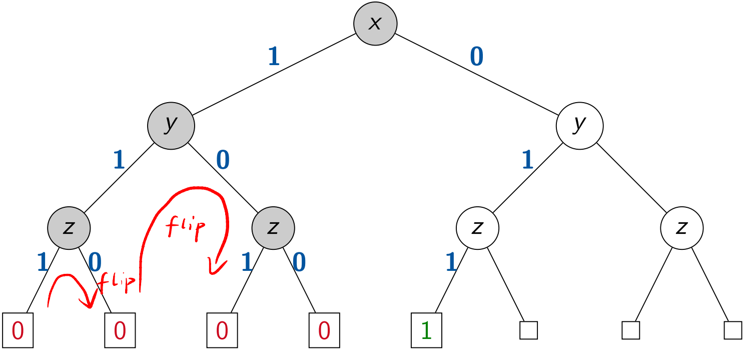
检查是否flip

若b=false,说明未flip, 则重新将¬v(v的相反值, 0-->1, 1-->0)压栈

Static decision heuristics example

$$\underbrace{(\neg x \vee y \vee z)}_{c_1} \wedge \underbrace{(y \vee \neg z)}_{c_2} \wedge \underbrace{(\neg x \vee \neg y)}_{c_3}$$

Static variable order $x < y < z$, sign: try positive first



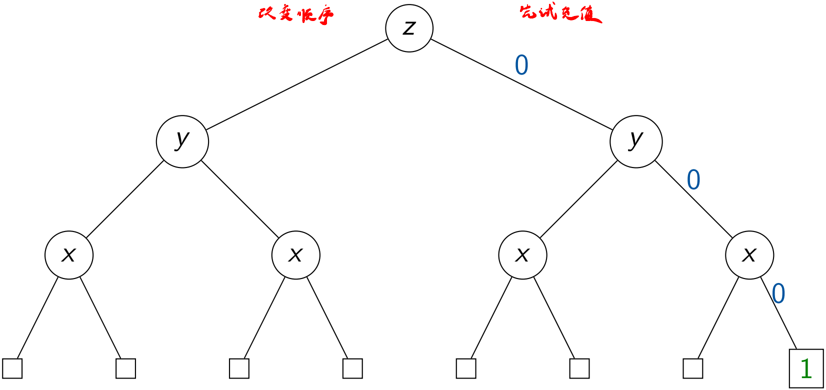
For unsatisfiable problems, all assignments need to be checked.
For satisfiable problems, variable and sign ordering might strongly influence the running time.

Static decision heuristics example

$$\underbrace{(\neg x \vee y \vee z)}_{c_1} \wedge \underbrace{(y \vee \neg z)}_{c_2} \wedge \underbrace{(\neg x \vee \neg y)}_{c_3}$$

Static variable order $z < y < x$, sign: try negative first

改变顺序 先试负值



Dynamic decision heuristics example: DLIS

Dynamic Largest Individual Sum (DLIS): Choose an assignment that increases the most the number of satisfied clauses.

- For each literal ℓ, let C_ℓ be the number of unresolved clauses in which ℓ appears.
- Let ℓ be a literal for which C_ℓ is maximal ($C_{\ell'} \leq C_\ell$ for all literals ℓ').
- If ℓ is a variable x then assign `true` to x.
- Otherwise if ℓ is a negated variable $\neg x$ then assign `false` to x.

- Requires $\mathcal{O}(\#\mathit{literals})$ queries for each decision.

number of literals

∵ 每一个 literal 都要计算

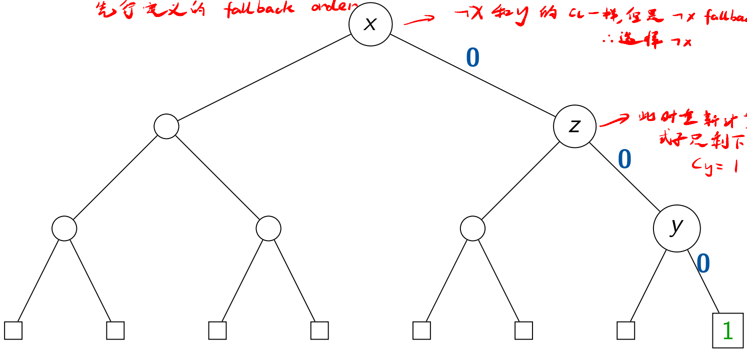
Dynamic decision heuristics example: DLIS

$$\underbrace{(\neg x \vee y \vee z)}_{c_1} \wedge \underbrace{(y \vee \neg z)}_{c_2} \wedge \underbrace{(\neg x \vee \neg y)}_{c_3}$$

[*Animation skipped for handout.*]

Fallback literal order (in case of equal values): $\neg x < x < \neg z < z < \neg y < y$

先行定义的 fallback order



→ ¬X 和 y 的 cL 一样，但是 ¬x fallback order 更靠前
∴ 选择 ¬x

→ 此时重新计算 cL
式子只剩下：$(y \wedge \neg z)$
$c_y = 1$ $\quad c_{\neg z} = 1$

Jeroslow-Wang method

short clause first，优先解决短的clause

Compute for every literal ℓ the following static value:

clause的长度，literals in clause

clause 中 literal 的数量

$$J(\ell) : \sum_{\textit{clause } c \textit{ in the CNF containing } \ell} 2^{-|c|}$$

- Choose a literal ℓ that maximizes $J(\ell)$.
- This gives an exponentially higher weight to literals in shorter clauses.

$$\underbrace{(\neg x \vee y \vee z)}_{c_1} \wedge \underbrace{(y \vee \neg z)}_{c_2} \wedge \underbrace{(\neg x \vee \neg y)}_{c_3}$$

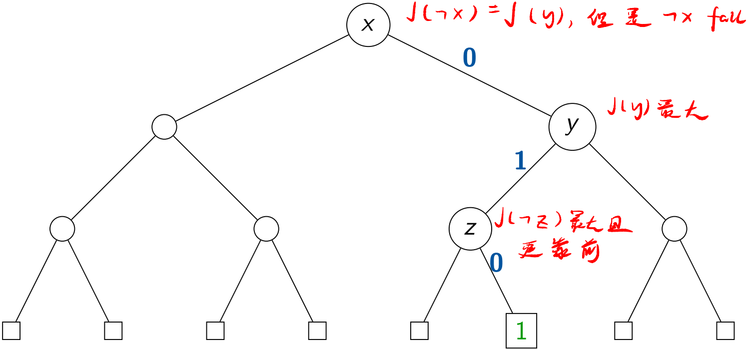
Static Jeroslow-Wang method

$J(x) = 0,\ J(\neg x) = \frac{1}{8} + \frac{1}{4},\ J(y) = \frac{1}{8} + \frac{1}{4},\ J(\neg y) = \frac{1}{4},\ J(z) = \frac{1}{8},\ J(\neg z) = \frac{1}{4}$

3个literals, $2^{-3} = \frac{1}{8}$

Fallback literal order (in case of equal values): $\neg x < x < \neg z < z < \neg y < y$

$J(\neg x) = J(y)$, 但是 $\neg x$ fallback 更靠前



$J(y)$ 最大

$J(\neg z)$ 最大且更靠前

Decision heuristics

- We will see other (more advanced) decision heuristics later.

Status of a clause

- Given a (partial) assignment, a clause can be

satisfied:	at least one literal is satisfied
unsatisfied:	all literals are assigned but none are statisfied
unit:	all but one literals are assigned but none are satisfied
unresolved:	all other cases 有大于一个clause未赋值, 但赋值都为0

- Example: $c = (x_1 \vee x_2 \vee x_3)$

x_1	x_2	x_3	c
1	0		satisfied
0	0	0	unsatisfied
0	0		unit
	0		unresolved

unsatisfied + unassigned

BCP: Unit clauses are used to imply consequences of decisions.

Some notations:

- Decision Level (DL) is a counter for decisions → how many decisions are made (flip 不是 decision)
- Antecedent(ℓ): unit clause implying the value of the literal ℓ (nil if decision)

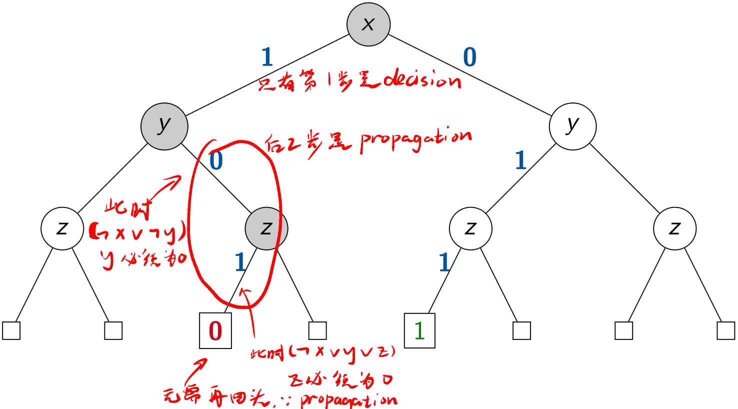
The DPLL algorithm: Exploration + propagation

```
bool DPLL(CNF_Formula φ){
  trail.clear(); //trail is a global stack of assignments
  if (!BCP()) then return UNSAT;
  while (true) {
    if (!decide()) then return SAT;
    while (!BCP())
      if (!backtrack()) then return UNSAT;
  }
}

bool decide() { as for exploration }

bool backtrack() { as for exploration }

bool BCP() { //more advanced implementation: return false as soon as an unsatisfied clause is detected
  while (there is a unit clause implying that a variable x must be set to a value v)
    trail.push(x, v, true); → unit clause, 00U…—, 所以不需要flip
  if (there is an unsatisfied clause) then return false;
  return true;
}             → all clause are satisfied
```

Boolean constraint propagation: Example

$$\underbrace{(\neg x \lor y \lor z)}_{c_1} \land \underbrace{(y \neg z)}_{c_2} \land \underbrace{(\neg x \lor \neg y)}_{c_3}$$

Static variable order $x < y < z$, sign: try positive first



BCP using watched literals

不可出现的情况是clause中仍有unsigned时仍旧选择flase作为wathed literals

- For BCP, it would be a large effort to check for each propagation the value of each literal in each clause.
- Idea: in each clause watch two different literals such that either one of them is `true` or both are `unassigned`
  $\rightarrow$ clause is neither unit nor unsatisfied.

  If a literal ℓ gets `false`, we propagate it by checking each clause c in which it is watched. Let ℓ' be the other watched literal in c.

  - If ℓ' is `true`, the clause is satisfied.
  - Else, if we find a non-false literal different from ℓ and ℓ' to be watched instead of ℓ, we are done.
  - Else, if ℓ' is unassigned, the clause is unit; we assigne true to ℓ'.
  - Else, if ℓ' is `false`, the clause is conflicting.

  We do this iteratively until either a conflicting clause is detected or all assigned (decided or implied) values are propagated.

BCP using watched literals

- For BCP, it would be a large effort to check for each propagation the value of each literal in each clause.

$c : (\underline{l_1} \lor \underline{l_2} \lor l_3 \lor l_4)$

$w(l_1) = c \ldots$

$w(l_2) = c \ldots$

Grade **0.00** out of 0.33 (**0**%)

Question 1
Incorrect
Mark 0.00 out of 0.33
Flag question

In the clause $(a \vee b \vee \neg c \vee \neg d)$, which literal pairs are suited to be watched under the assigment $a = 1$, $c = 0$, $d = 0$, and all other propositions unassigned?

Select one or more:

- [ ] (a, b)
- [x] $(a, \neg c)$ ✔
- [x] $(a, \neg d)$ ✔
- [x] $(b, \neg c)$ ✔
- [x] $(b, \neg d)$ ✔
- [x] $(\neg c, \neg d)$ ✔
- [ ] None of the above

The correct answers are: (a, b), $(a, \neg c)$, $(a, \neg d)$, $(b, \neg c)$, $(b, \neg d)$, $(\neg c, \neg d)$

Finish review

◄ Bonus test 05
Jump to...
E-test 1 ►

Quiz navigation
1
Finish review

1 Exploration (also called enumeration)

2 Boolean constraint propagation (BCP)

3 Conflict resolution and backtracking

4 Exploration revisited

Implication graph

We represent (partial) variable assignments in the form of an implication graph.

Definition

An implication graph is a labeled directed acyclic graph $G = (V, E, L)$, where

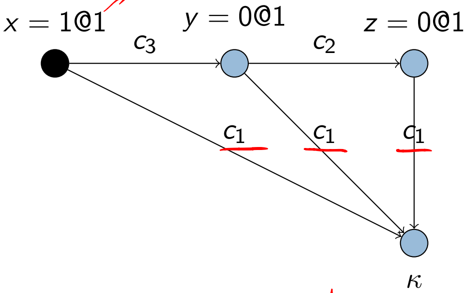
- V is a set of nodes, one for each currently assigned variable and an additional conflict node κ if there is a currently conflicting clause c_{confl}.
- L is a labeling function assigning a lable to each node. The conflict node (if any) is labelled by $L(\kappa) = \kappa$. Each other node n, representing that x is assigned $v \in \{0, 1\}$ at decision level d, is labeled with $L(n) = (x = v@d)$; we define $literal(n) = x$ if $v = 1$ and $literal(n) = \neg x$ if $v = 0$.
- $E = \{(n_i, n_j) | n_i, n_j \in V, n_i \neq n_j, \neg literal(n_i) \in \text{Antecedent}(literal(n_j))\} \cup \{(n, \kappa) | n, \kappa \in V,\ \neg literal(n) \in c_{confl}\}$ is the set of directed edges where each edge (n_i, n_j) is labeled with Antecedent($literal(n_j)$) if $n_j \neq \kappa$ and with c_{confl} otherwise.

Implication graph: Example

$$\underbrace{(\neg x \vee y \vee z)}_{c_1} \wedge \underbrace{(y \vee \neg z)}_{c_2} \wedge \underbrace{(\neg x \vee \neg y)}_{c_3}$$

Static variable order $x < y < z$, sign: try positive first

decision level 1

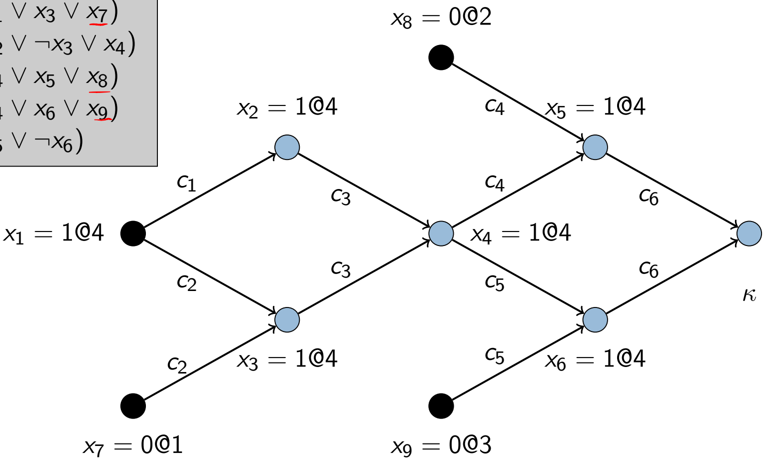


c1中 x, y, z 的赋值，产生冲突

Implication graph: Example

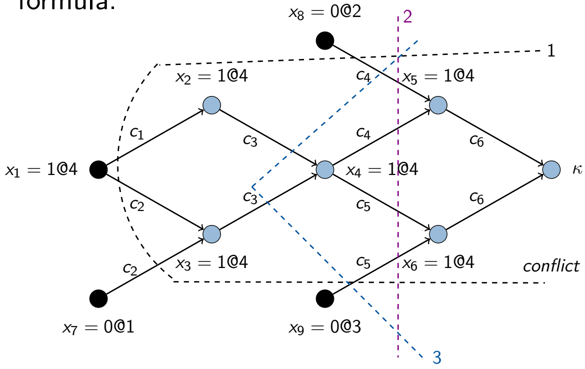
Decisions: $\{x_7 = 0@1,\ x_8 = 0@2,\ x_9 = 0@3,\ x_1 = 1@4\}$

$c_1 = (\neg x_1 \vee x_2)$
$c_2 = (\neg x_1 \vee x_3 \vee x_7)$
$c_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 = (\neg x_4 \vee x_5 \vee x_8)$
$c_5 = (\neg x_4 \vee x_6 \vee x_9)$
$c_6 = (\neg x_5 \vee \neg x_6)$



Conflict resolution

- Assume that the current (partial) assignment doesn't satisfy our formula.
- Let L be a set of literals labeling nodes that form a cut in the implication graph, seperating a conflict node from the roots.
- $\bigvee_{l \in L} \neg l$ is called a conflict clause: it is false under the current assignment but its satisfaction is necessary for the satisfaction of the formula.



1.$(x_8 \vee \neg x_1 \vee x_7 \vee x_9)$

2.$(x_8 \vee \neg x_4 \vee x_9)$

3.$(x_8 \vee \neg x_2 \vee \neg x_3 \vee x_9)$

⋮

⋮

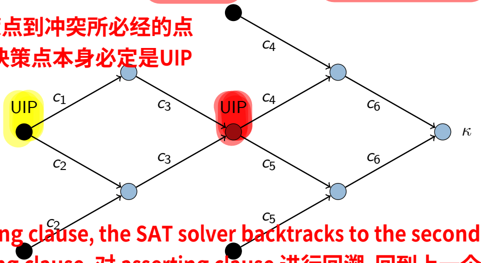
Conflict resolution

The clause is asserting if exactly one literal in that clause was assigned at the current decision level. The assignment can be a decision or a propagation,

- Which conflict clauses should we consider?
- An asserting clause is a conflict clause with a single literal from the current decision level.
  Backtracking (to the right level) makes it a unit clause.
- Modern solvers consider only asserting clauses.
- Assume an implication graph G with a conflict node κ. A unique implication point (UIP) for κ in G is a node $n \neq \kappa$ in G such that all paths from the last decision to κ go through n.
  最后一次赋值点
- The first UIP is the UIP closest to the conflict node.

UIP即是从最后一次决策点到冲突所必经的点

UIP>=1, 因为最后一次决策点本身必定是UIP



After finding the asserting clause, the SAT solver backtracks to the second highest decision level of any literal in the asserting clause. 对 asserting clause 进行回溯, 回到上一个决策层(第二深的决策层)

Conflict-driven backtracking

- Usually, the asserting conflict clause is learnt by adding it to the clause set. However, this is not necessary for completeness.
- Backtrack to the second highest decision level *dl* in the asserting conflict clause (but do not erase it).
- This way the literal with the currently highest decision level will be implied at decision level *dl*.
- Propagate all new assignments.

Q: What happens if the asserting conflict clause has a single literal?
For example, from $(x \vee \neg y) \wedge (x \vee y)$ and decision $x = 0$, we get (x).
A: Backtrack to DL0.

Q: What happens if the conflict appears at decision level 0?
A: The formula is unsatisfiable.

Bonus exercise 7 (8 minutes)

Assume the following propositional logic formula in CNF:

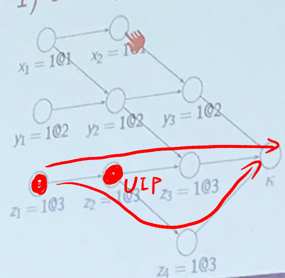
$c_0:(\neg x_1 \vee x_2)\wedge$ $c_1:(\neg x_1 \vee \neg y_1 \vee y_2)\wedge$ $c_2:(\neg x_2 \vee \neg y_2 \vee y_3)\wedge$
$c_3:(\neg z_1 \vee z_2)\wedge$ $c_4:(\neg y_2 \vee \neg z_2 \vee z_3)\wedge$ $c_5:(\neg z_2 \vee z_4)$
$c_6:(\neg y_3 \vee \neg z_3 \vee \neg z_4)$

Assume furthermore the following trail:

null point，可以自由赋值，不被 propagate 的点.

DL0: -
DL1: x_1:nil x_2:c_0
DL2: y_1:nil y_2:c_1 y_3:c_2
DL3: z_1:nil z_2:c_3 z_3:c_4 z_4:c_5

We detect a conflicting clause c_6. How many unique implication points are in the implication graph? 1) 0 2) 1 3) 2 4) 3 5) 4 6) 5



The DPLL+CDCL algorithm

如果一开始BCP()就为false(只有一个Literal的clause为false), 则返回unsat

```
if (!BCP()) return UNSAT;
while (true)
{
      if (!decide()) return SAT;
      while (!BCP())
            if (!resolve_conflict()) return UNSAT;
}
```

如果没有变量未被赋值且下面的while循环中BCP为true, 跳出循环 ==> 返回sat

若BCP为false

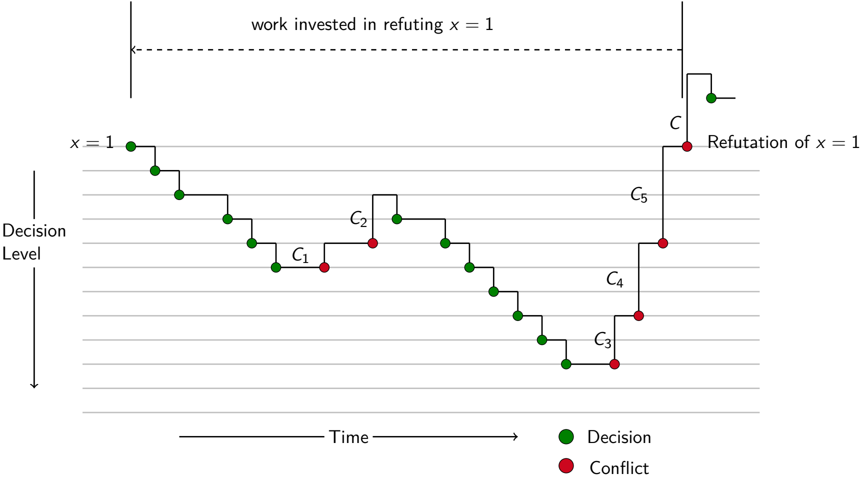
如果resolve confilict为impossible
==> 返回unsat

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

Boolean constraint propagation.
Return false if reached a conflict.

Conflict resolution and backtracking. Return false if impossible.

Progress of a DPLL+CDCL-based SAT solver



Conflict clauses and (binary) resolution

- The (binary) resolution is a sound (and complete) inference rule:

$$\frac{(\beta \vee a_1 \vee \ldots \vee a_n) \qquad (\neg\beta \vee b_1 \vee \ldots \vee b_m)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)} \text{(Binary Resolution)}$$

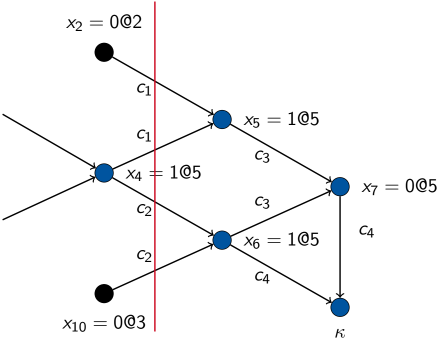
- Example:

$$\frac{(x_1 \vee x_2) \qquad (\neg x_1 \vee x_3 \vee x_4)}{(x_2 \vee x_3 \vee x_4)}$$

What is the relation of binary resolution and conflict clauses?

- Consider the following example:

$c_1 = (\neg x_4 \vee x_2 \vee x_5)$
$c_2 = (\neg x_4 \vee x_{10} \vee x_6)$
$c_3 = (\neg x_5 \vee \neg x_6 \vee \neg x_7)$
$c_4 = (\neg x_6 \vee x_7)$
$\vdots \quad\quad \vdots$

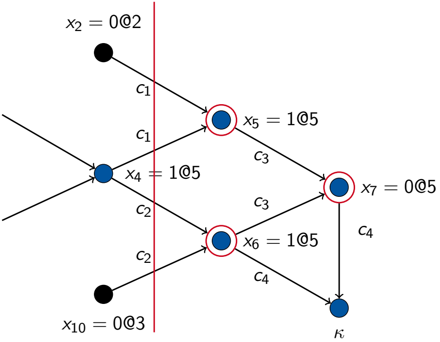


- Asserting conflict clause: $c_5 : (x_2 \vee \neg x_4 \vee x_{10})$

Conflict clauses and (binary) resolution

- Assigment order: x_4, x_5, x_6, x_7 Conflict clause: $c_5 : (x_2 \vee \neg x_4 \vee x_{10})$

$c_1 = (\neg x_4 \vee x_2 \vee x_5)$
$c_2 = (\neg x_4 \vee x_{10} \vee x_6)$
$c_3 = (\neg x_5 \vee \neg x_6 \vee \neg x_7)$
$c_4 = (\neg x_6 \vee x_7)$
$\vdots \qquad \vdots$



- Starting with the conflicting clause, apply resolution with the antecedent of the last assigned literal, until we get an asserting clause:
  - $T1 = Res(c_4, c_3, x_7) = (\neg x_5 \vee \neg x_6)$
  - $T2 = Res(T1, c_2, x_6) = (\neg x_4 \vee \neg x_5 \vee x_{10})$
  - $T3 = Res(T2, c_1, x_5) = (x_2 \vee \neg x_4 \vee x_{10})$

此时T3为asserting clause, 即找到learned clause!!!

此时只有x4在最高决策层, 为asserting clause

Finding the asserting conflict clause

```
bool analyze_conflict() {
    if (current_decision_level = 0) then return false;
    cl := current_conflicting_clause;
    while (cl is not asserting) do {
literal lit := last_assigned_literal(cl);
        var := variable_of_literal(lit);
        ante := antecedent(var);
        cl := resolve(cl, ante, var);
    }
    add_clause_to_database(cl);
    return true;
}
```

Applied to our example:

name	*cl*	*lit*	*var*	*ante*
c_4	$(\neg x_6 \vee x_7)$	x_7	x_7	c_3
	$(\neg x_5 \vee \neg x_6)$	$\neg x_6$	x_6	c_2
	$(\neg x_4 \vee x_{10} \vee \neg x_5)$	$\neg x_5$	x_5	c_1
c_5	$(\neg x_4 \vee x_2 \vee x_{10})$			

Unsatisfiable core

对CNF perform resolution得到empty clause ==> CNF is unsatisfiable
但是原来的CNF中不是所有clause都导致了empty clause
所以定义unsatisfiable core为: 导致CNF resolution为empty clause的 subset of original clauses

Definition

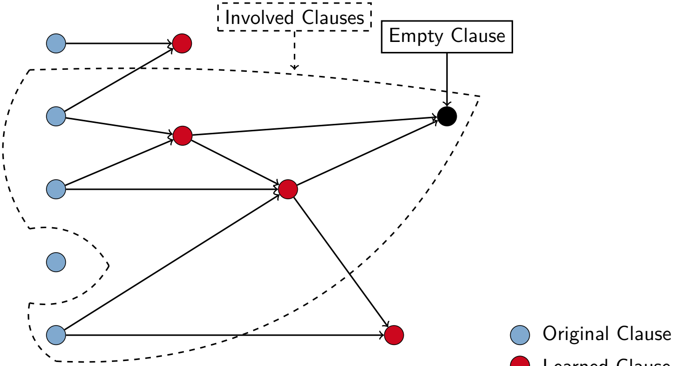
An unsatisfiable core of an unsatisfiable CNF formula is an unsatisfiable subset of the original set of clauses.

首先, 全体clauses肯定会导致empty clause ==> 全体clauses为unsatisfiable core

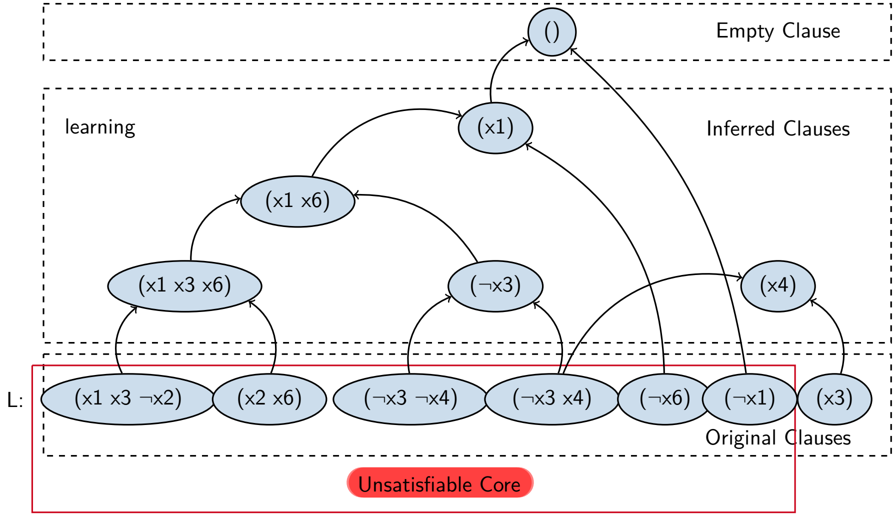
- The set of all original clauses is an unsatisfiable core.
- The set of those original clauses that were used for resolution in conflict analysis during SAT-solving (inclusively the last conflict at decision level 0) gives us an unsatisfiable core which is in general much smaller.
- However, this unsatifiable core is still not always minimal (i.e., we can remove clauses from it still having an unsatisfiable core).

The resolution graph

A resolution graph gives us more information to get a minimal unsatisfiable core.



Resolution graph: Example



Termination

Theorem

It is never the case that the solver enters decision level dl again with the same partial assignment. sat solver再次进入某一决策层时, assignment必定不相同

Proof.

Define a partial order on partial assignments: $\alpha < \beta$ iff either α is an extension of β or α has more assignments at the smallest decision level at that α and β do not agree.
BCP decreases the order, conflict-driven backtracking also. Since the order always decreases during the search, the theorem holds. $\square$

Decision heuristics: VSIDS

- VSIDS (variable state independent decaying sum)
- Gives priority to variables involved in recent conflicts.
- "Involved" can have different definitions. We take those variables that occur in clauses used for conflict resolution.

1. Each variable has a counter initialized to 0.
2. We define an increment value (e.g., 1).
3. When a conflict occurs, we increase the counter of each variable, that occurs in at least one clause used for conflict resolution, by the increment value.
   Afterwards we increase the increment value (e.g., by 1).
4. For decisions, the unassigned variable with the highest counter is chosen.
5. Periodically, all the counters and the increment value are divided by a constant.

Decision heuristics: VSIDS

- Chaff holds a list of unassigned variables sorted by the counter value.
- Updates are needed only when adding new conflict causes.
- Thus - decision is made in constant time.

Decision heuristics: VSIDS

VSIDS is a 'quasi-static' strategy:

- static because it doesn't depend on current assignment
- dynamic because it gradually changes. Variables that appear in recent conflicts have higher priority.

  This strategy is a conflict-driven decision strategy.

"...employing this strategy dramatically (i.e., an order of magnitude) improved performance..."

Learning target

- Exploration:
  What kind of (static and dynamic) variable ordering heuristics can be used?
- DPLL SAT solving:
  How does propagation work with exploration?
  What are watched literals?
- DPLL+CDCL SAT solving:
  How can resolution be used for conflict resolution?
  How to formalize and execute the resulting DPLL+CDCL SAT solving algorithm?
  How to construct unsatisfiable cores?