



软件分析

# 布尔可满足性

熊英飞  
北京大学



# 课程项目的一些细节

- 代码提交（12月1日）
  - Readme.pdf: A4两页以内，描述算法的主要设计思想，小组成员姓名、学号和分工
  - Code目录：项目源代码
  - analyzer.jar: 编译好的jar文件
- 现场报告（12月3日-4日）
  - 各组交流所采用的算法，每组报告12分钟左右，提问3分钟



# 课程项目组队情况

1. 张文杰 赵哲昊 贺宁宇
2. 徐瑞帆 梁清源 郝嘉琨
3. 韩佳良 杨程旭 张浩然 ( p2019092408 )
4. 杨晨阳 袁野 关智超
5. 阮思凯
6. 沈若冰 张石然
7. 邱睿 薛骁勇
8. 张云帆 谷晟 王家恒
9. 黄璨 蔡佳晋 魏安江
10. 吴宜谦 孙奕灿

报告顺序12月3日上课前抽签决定



# 抽象vs搜索

- 哥德尔不完备定理/停机问题/莱斯定理
  - 有限时间内无法得到精确的分析结果
- 抽象——在可快速计算的抽象域上得到上近似或下近似结果
- 搜索——尝试得到精确结果，如果超时就放弃
- 关键技术：约束求解技术



# 约束求解

- 给定一组约束，求
  - 这组约束是否可满足
  - (可选) 如果可满足，给出一组赋值
  - (可选) 如果不可满足，给出一个较小的矛盾集  
unsatisfiable core
- 总的来说是不可判定的问题，但存在很多可判定的子问题
- 如
  - $a > 10$
  - $b < 100 \ || \ b > 200$
  - $a+b=30$
- 可满足:  $a=15, b=15$



# 约束求解

- SAT solver: 解著名的NP完全问题
- Linear solvers: 求线性方程组
- Array solvers: 求解包含数组的约束
- String solver: 求解字符串约束
  
- SMT: 综合以上各类约束求解工具



# 历史

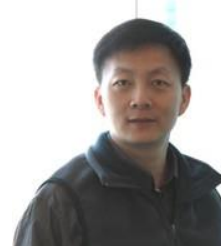
- 约束求解历史上一直有两个特点
  - 速度慢
  - 约束求解算法分散发展，各自只能解小部分约束
- 进入2000年以来
  - SAT的求解速度得到了突飞猛进的进步
    - 理论上还无法完全解释SAT的高速求解
  - 以SAT为核心，各种单独的约束求解算法被整合起来，形成了SMT



Sharad Malik  
领导开发Chaff求解器  
Lintao Zhang导师



Joao Marques-Silva  
关键算法CDCL提出人



Lintao Zhang  
CDCL的通用化和效率提升



# 复习：SAT问题

- 最早被证明的NP完全问题之一（1971）
- 文字literal: 变量 $x$ 或者是 $x$ 取反
  - 如 $\neg x$
- 子句clause: 文字的析取（disjunction）
  - 如 $x \vee \neg y$
- 布尔赋值: 从变量到布尔值上的映射
- SAT问题: 子句集上的约束求解问题
  - 给定一组子句，寻找一个布尔赋值，使得所有子句为真





# 复习：合取范式

## Conjunctive Normal Form

- 合取范式：子句的合取
  - 如  $(x \vee \neg y) \wedge \neg x$
- SAT问题通常是通过合取范式定义的
- 任何命题逻辑公式可以表达为合取范式
- 即：SAT问题可以求解任何命题逻辑公式



# SAT举例

- 每行为一个子句
- $\{1, -4\}$
- $\{-2, 3\}$
- $\{2, 4\}$
- $\{-2, -3, 4\}$
- $\{-1, -4\}$
- 该SAT问题是否可满足？
  - 不可满足
- 可满足条件：存在一组解，使得每个子句中至少有一个文字为真



# SAT基本求解算法-穷举

```
Sat(assign) {  
  if (assign是完整的)  
    if(每个子句中都有至少一个文字为真)  
      return true;  
    else return false;  
  else  
    选择一个未赋值的变量x;  
    return sat(assign  $\cup$  {x=0}) || sat(assign  $\cup$  {x=1})  
}
```



# 优化1：冲突检测

- $\text{assign}=\{1, 4\}$  红色为假，绿色为真
- $\{1, -4\}$
- $\{-2, 3\}$
- $\{2, 4\}$
- $\{-2, -3, 4\}$
- $\{-1, -4\}$  //出现冲突
- 不需要完整赋值就能知道结果

# 优化1: 基于冲突检测的SAT 求解算法



```
Sat(assign) {  
    if (assign有冲突) return false;  
    if (assign是完整的) return true;  
    选择一个未赋值的变量x;  
    return sat(assign U {x=0}) || sat(assign U {x=1})  
}
```



# 优化2：赋值推导

- $\text{assign}=\{2\}$  红色为假，绿色为真
- $\{1, -4\}$
- $\{-2, 3\}$
- $\{2, 4\}$
- $\{-2, -3, 4\}$
- $\{-1, -4\}$



# 优化2：赋值推导

- $\text{assign}=\{2\}$  红色为假，绿色为真
- $\{1, -4\}$
- $\{-2, 3\}$  //推导
- $\{2, 4\}$
- $\{-2, -3, 4\}$
- $\{-1, -4\}$



# 优化2：赋值推导

- $\text{assign}=\{2\}$  红色为假，绿色为真
- $\{1, -4\}$
- $\{-2, 3\}$
- $\{2, 4\}$
- $\{-2, -3, 4\}$  //推导
- $\{-1, -4\}$





# 优化2：赋值推导

- $\text{assign}=\{2\}$  红色为假，绿色为真
- $\{1, -4\}$  //推导
- $\{-2, 3\}$
- $\{2, 4\}$
- $\{-2, -3, 4\}$
- $\{-1, -4\}$



## 优化2：赋值推导

- $\text{assign}=\{2\}$  红色为假，绿色为真
- $\{1, -4\}$
- $\{-2, 3\}$
- $\{2, 4\}$
- $\{-2, -3, 4\}$
- $\{-1, -4\}$  //矛盾
  
- 无需继续遍历赋值就能得出结果



# 标准推导方法

- Unit Propagation

- 其他文字都为假，剩下的一个文字必定为真
- $\{-1, 2, -3\} \Rightarrow 3$

- Unate Propagation

- 当一个子句存在为真的文字时，可以从子句集合中删除
- ~~$\{2, 4\}$~~

- Pure literal elimination

- 当一个变量只有为真或者为假的形式的时候，可以把包含该变量的子句删除
- ~~$\{4, 6\}$~~
- ~~$\{4, -6\}$~~

# 优化2：基于赋值推导和冲突检测的SAT求解——DPLL



```
dppll(assign) {  
    assign'=赋值推导(assign);  
    if (assign'有冲突) return false;  
    if (assign'是完整的) return true;  
    选择一个未赋值的变量x;  
    return dppll(assign'  $\cup$  {x=0}) || dppll(assign'  $\cup$  {x=1});  
}
```

- 该算法被称为DPLL，由Davis, Putnam, Logemann Loveland在1962年代提出



# 高级推导方法

- Probing
  - 如果令 $x=0$ 或者 $x=1$ 都能推导出 $y=0$ ，则推导出 $y=0$
- Equivalence classes
  - 预先检查出等价的子句集合，然后删除其中一个
    - $\{1, 2, -3\}$
    - ~~$\{2, 1, -3\}$~~



# 优化3： 变量选择

- 先选择哪个变量赋值可能对求解造成很大影响
  - {1, -2}
  - {1, 2}
  - {-1, -2}
  - {-1, 2}
  - {3, 4, 5, 6, 7, 8}
- 优先选择1或者2可以快速发现不可满足
- 优先选择3-8需要反复回溯多次

# 变量选择方法 Branching Heuristics



- 基于子句集的
  - 优先选择最短子句里的变量
  - 优先选择最常出现的变量
  - 例：上页例子中可以直接选到1或者2
- 基于历史的
  - 优先选择之前导致过冲突的变量
  - 例：上页例子一次完整赋值后，会优先选择1或者2

# 优化4: 冲突导向的子句学习

## CDCL Conflict-Driven Clause Learning



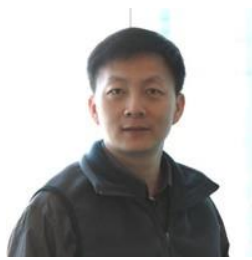
- 2000年初期大幅提升SAT效率的重要因素之一
- 基本思想: 在搜索过程中学习问题的性质, 加入约束集合中



Sharad Malik  
领导开发Chaff求解器  
Lintao Zhang导师



Joao Marques-Silva  
关键算法CDCL提出人



Lintao Zhang  
CDCL的通用化和效率提升

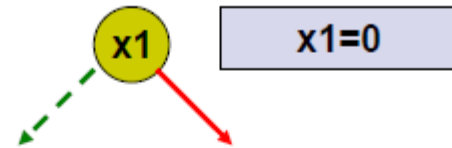




# 一个DPLL的例子

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

Step 1



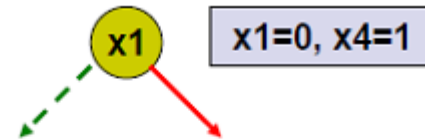
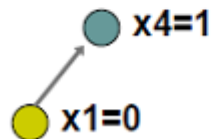
●  $x_1=0$



# 一个DPLL的例子

## Step 2

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10}'$
- $x_7 + x_{10} + x_{12}'$

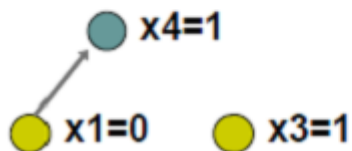
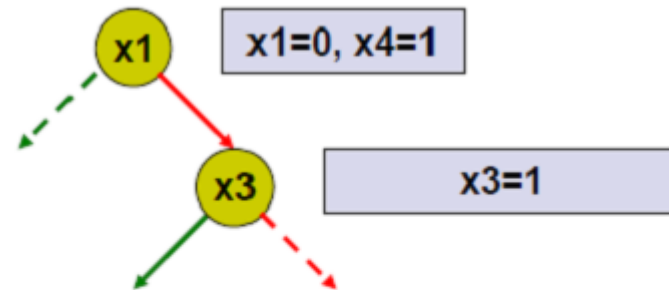




# 一个DPLL的例子

## Step 3

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

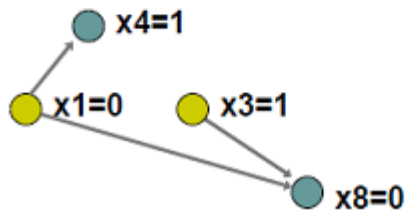
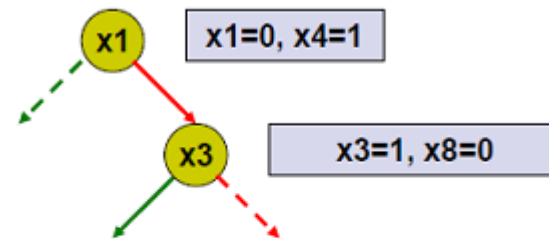




# 一个DPLL的例子

## Step 4

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

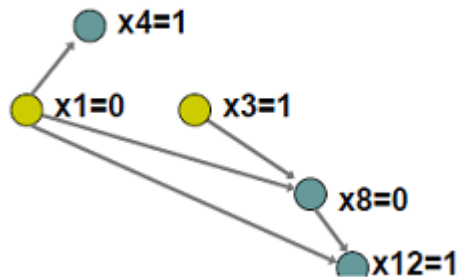
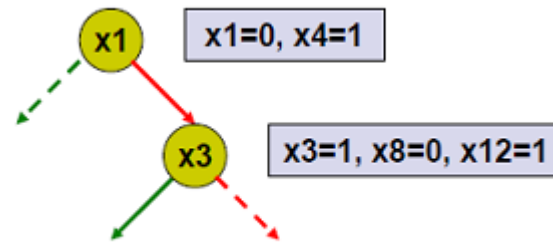




# 一个DPLL的例子

*Step 5*

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

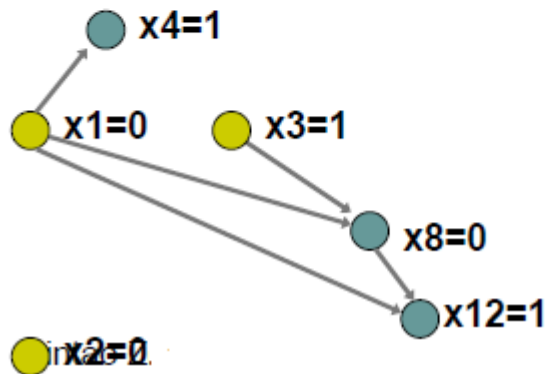
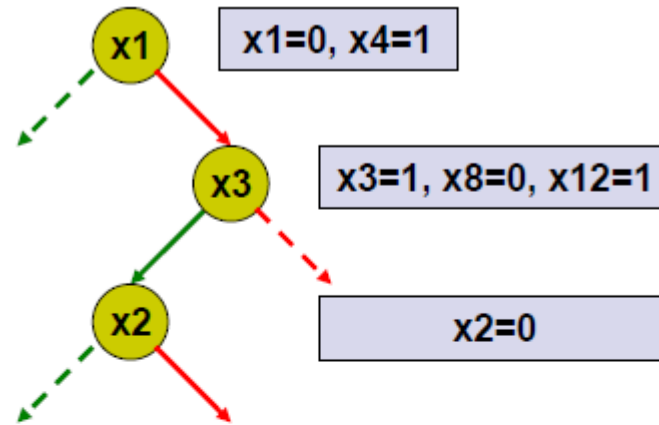




# 一个DPLL的例子

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10}'$
- $x_7 + x_{10} + x_{12}'$

Step 6

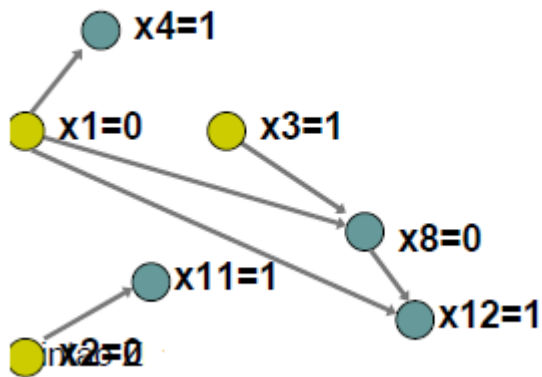
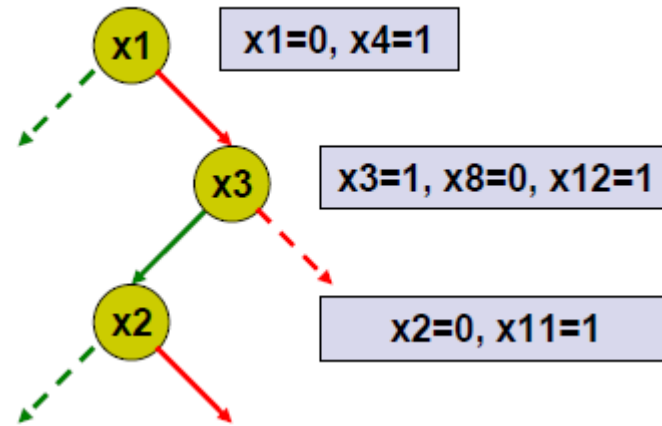




# 一个DPLL的例子

Step 7

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10}'$
- $x_7 + x_{10} + x_{12}'$

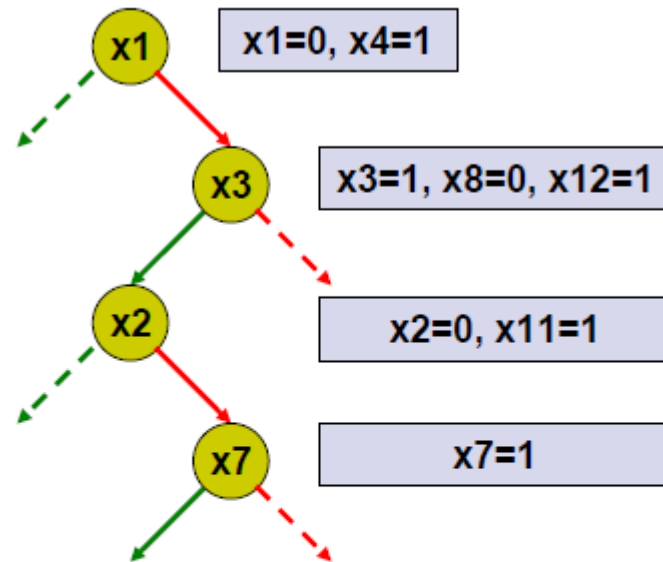
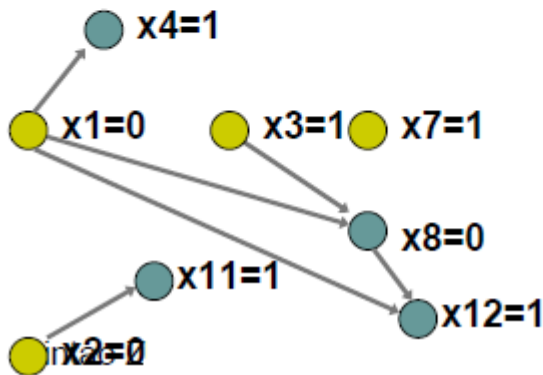




# 一个DPLL的例子

Step 9

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10}'$
- $x_7 + x_{10} + x_{12}'$



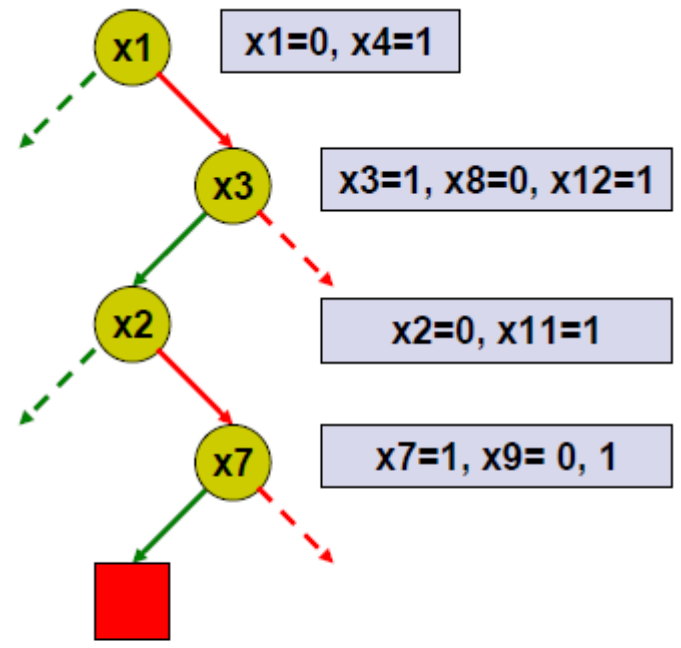
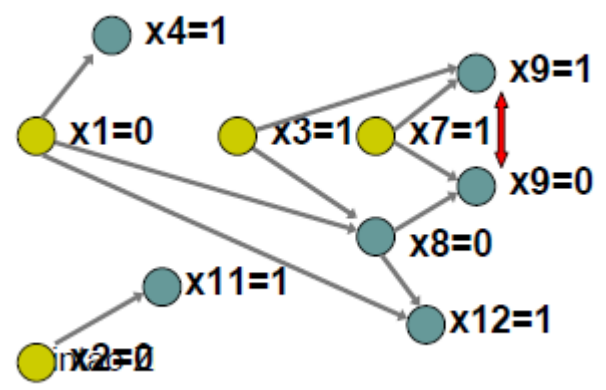




# 一个DPLL的例子

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10}'$
- $x_7 + x_{10} + x_{12}'$

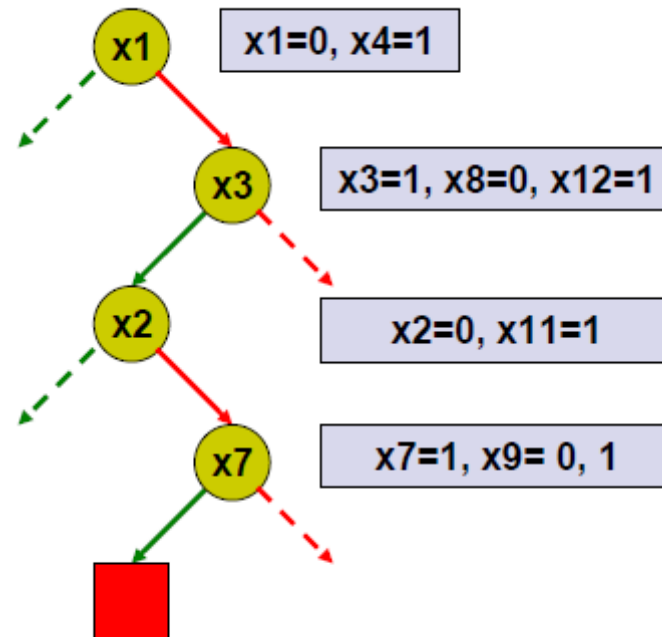
Step 10





# DPLL的问题

- 如果后续搜索把x7再次设置成了1，会重复出现该冲突

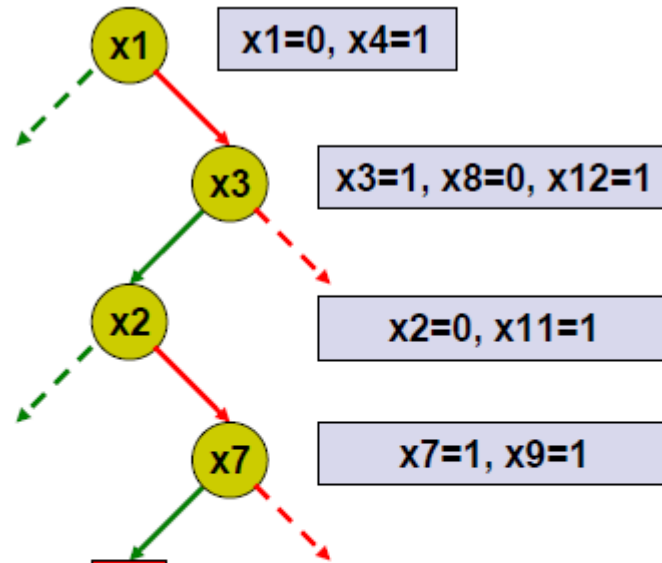
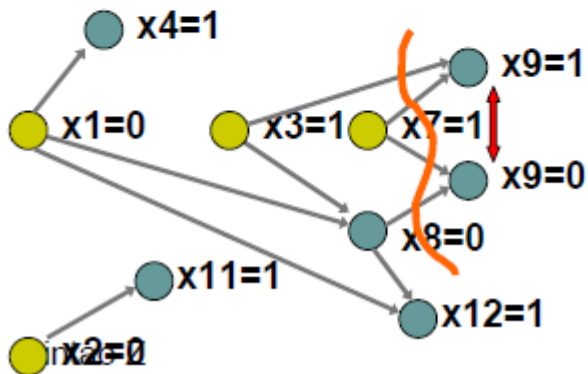




# CDCL: 子句学习

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

Step 11



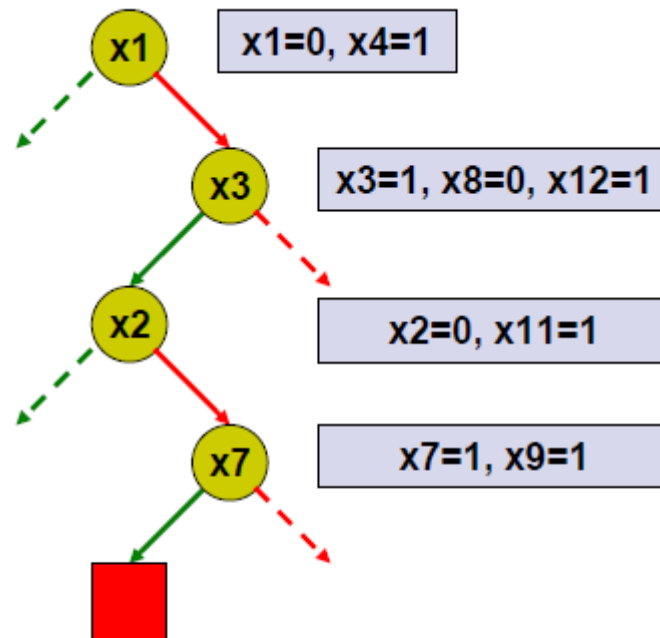
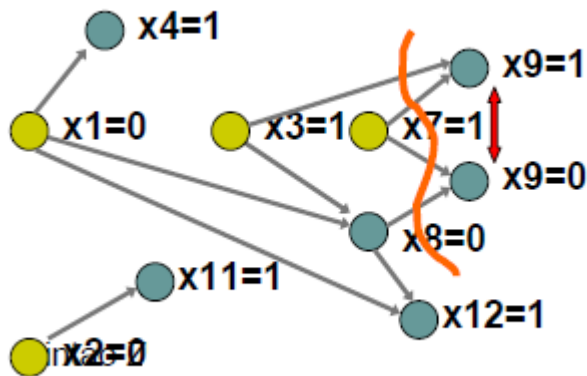
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow \text{conflict}$



# CDCL: 子句学习

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

Step 13



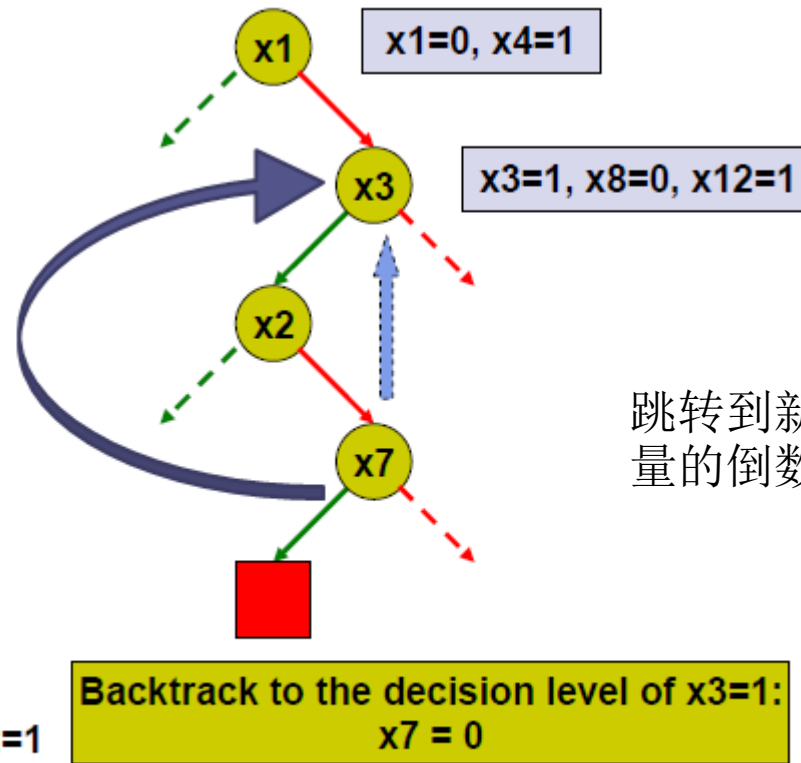
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow \text{conflict}$   
 Add conflict clause:  $x3' + x7' + x8$



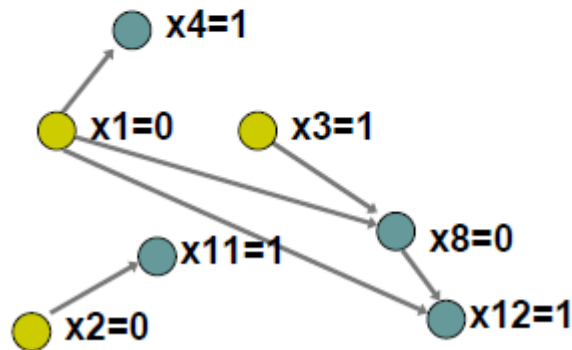
# CDCL: 子句学习

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

Step 14



跳转到新加子句中变量的倒数第二次赋值

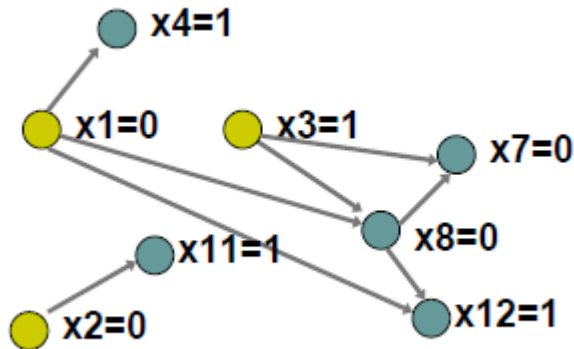
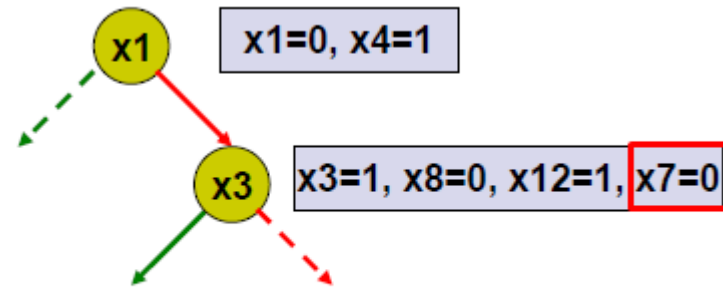




# CDCL: 子句学习

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10}'$
- $x_7 + x_{10} + x_{12}'$
- $x_3' + x_8 + x_7'$

Step 15





# CDCL: 子句学习

- 注意从新添加约束出发的推导实际保证了之前探过的冲突赋值不会出现
- 所以不再需要记录之前遍历过的赋值，每次任意选择剩下的变量和赋值即可
  - 空赋值推出冲突意味着UNSAT



# 优化4：冲突导向的子句学习

## CDCL Conflict-Driven Clause Learning

```
cdcl() {  
  assign=空赋值;  
  while (true) {  
    赋值推导(assign);  
    if (推导结果有冲突) {  
      if (assign为空) return false;  
      添加新约束;  
      撤销赋值;  
    } else {  
      if (推导结果是完整的) return true;  
      选择一个未尝试的赋值x=1或者x=0;  
      添加该赋值到assign;  
    }  
  }  
}
```





## 优化4.1：新的变量选择方法VSIDS

- VSIDS=Variable State Independent Decaying Sum
- 首先按变量出现次数给所有变量打分
- 添加新子句的时候给子句中的变量加分
- 每隔一段时间把所有变量的分数除以一个常量



# 参考资料

- Decision Procedures: An Algorithmic Point of View
  - Daniel Kroening and Ofer Strichman
  - Springer, 2008