



软件分析

# 布尔可满足性

熊英飞

北京大学



# 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
  - $a > 10$
  - $b < 100 \mid\mid b > 200$
  - $a+b=30$
- 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  $\cup$  {x=0}) || sat(assign  $\cup$  {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

- 单元传播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
    - ~~$\{2, 4\}$~~

# 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



# 优化3：预处理

- 归结Resolution：两个子句分别包括 $x$ 和 $\neg x$ 时，且 $x$ 在其余子句不出现，可以将两个子句合并成一个并删掉 $x$ 和 $\neg x$ 
  - $\begin{matrix} \{1, 2, -3\} \\ \{4, 3\} \end{matrix} \Rightarrow \{1, 2, 4\}$
- 多子句归结：多个子句分别包括 $x$ 和 $\neg x$ 时，且 $x$ 在其余子句不出现，可以把包括 $x$ 的子句集合和包括 $\neg x$ 的子句集合做笛卡尔积并删掉 $x$ 和 $\neg x$ 
  - $\begin{matrix} \{1, 2, 3\} & \{2, 3, 5\} \\ \{1, 4\} & \{2, 3, 6\} \\ \{-1, 5\} & \{4, 5\} \\ \{-1, 6\} & \{4, 6\} \end{matrix} \Rightarrow$
  - 多语句归结让变量变少，但并不一定会让公式集合更简单，通常配合启发式规则应用
  - 早期存在完全基于归结，不穷举赋值的算法（DP算法），但速度通常显著落后于DPLL



# 其他预处理方法

- 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

- ~~$\{-4, 6\}$~~

- ~~$\{-4, -6\}$~~

- 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4： 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
  - 注意：变量选择只是在下一次选择变量时生效，对于已经生成的搜索树仍然会遍历结束

# 优化5：冲突导向的子句学习

## CDCL Conflict-Driven Clause Learning



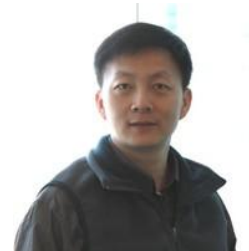
- 2000年初期大幅提升SAT效率的重要因素之一
- 基本思想：在搜索过程中学习问题的性质，加入约束集合中



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



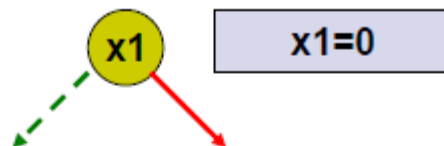
Lintao Zhang  
CDCL的通用化和效率提升



# 一个DPLL的例子

Step 1

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



●  $x_1=0$

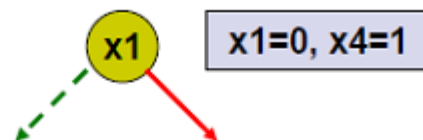
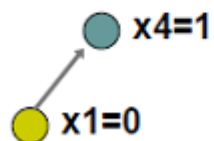




# 一个DPLL的例子

Step 2

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

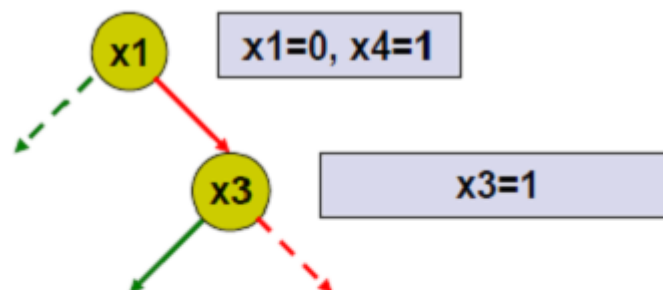
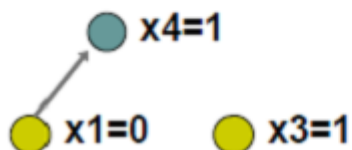




# 一个DPLL的例子

Step 3

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

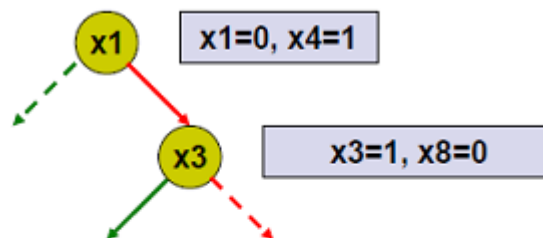
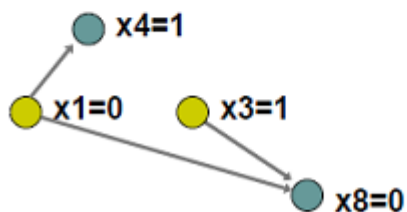




# 一个DPLL的例子

## Step 4

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

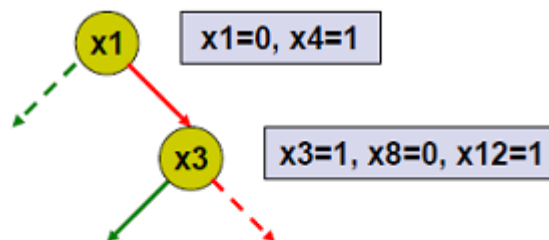
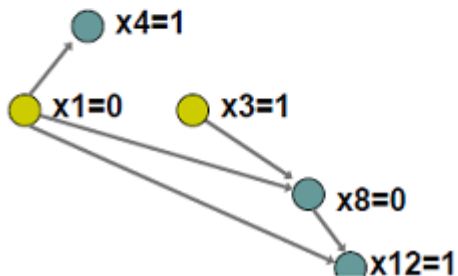




# 一个DPLL的例子

Step 5

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

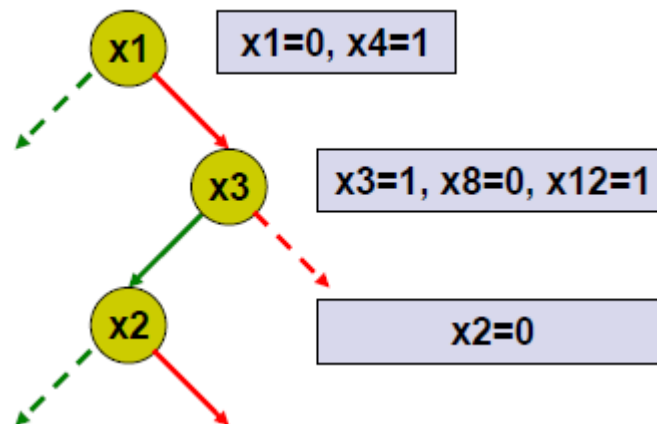
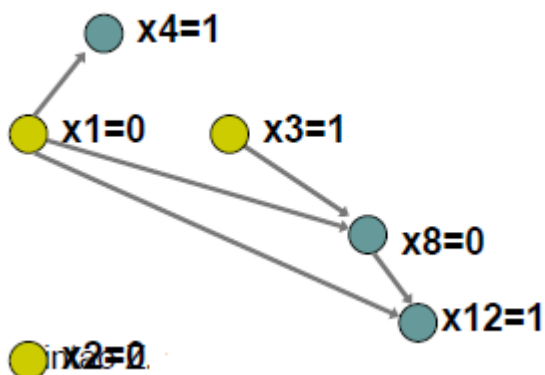




# 一个DPLL的例子

Step 6

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

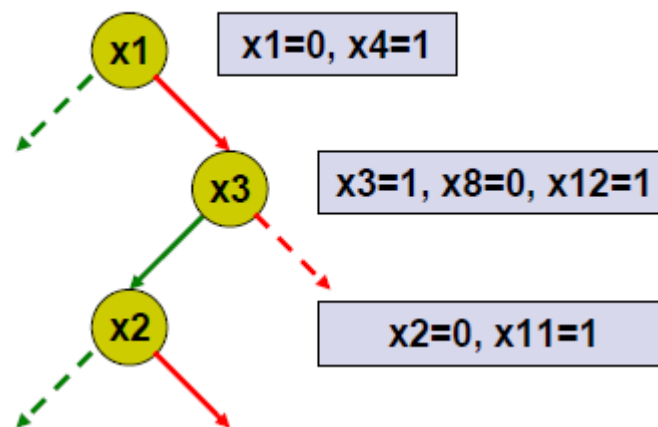
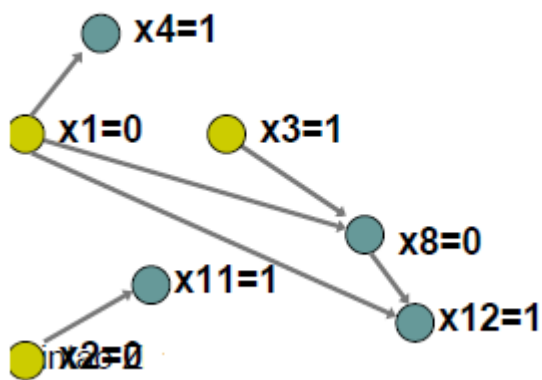




# 一个DPLL的例子

Step 7

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

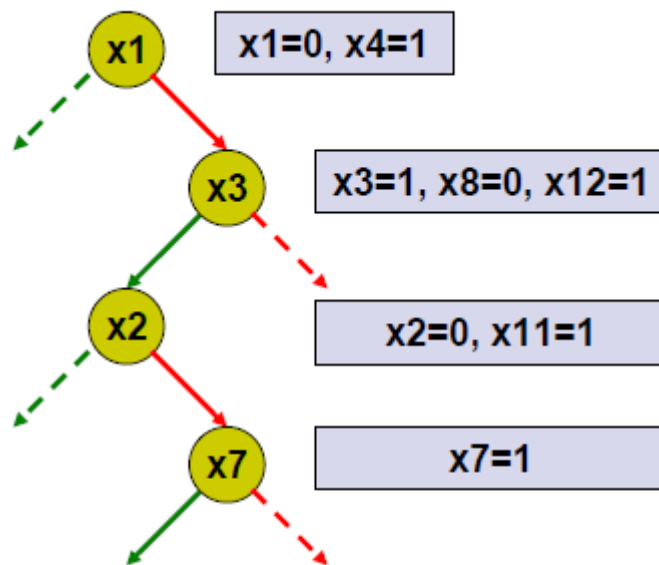
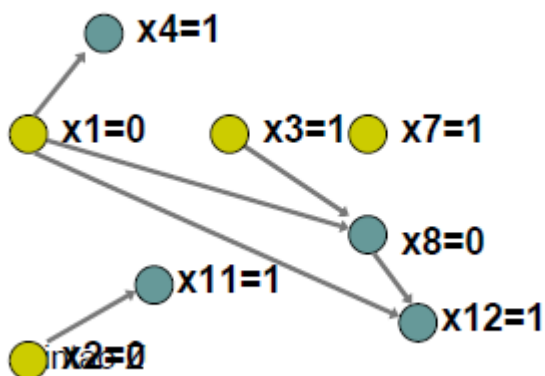




# 一个DPLL的例子

Step 9

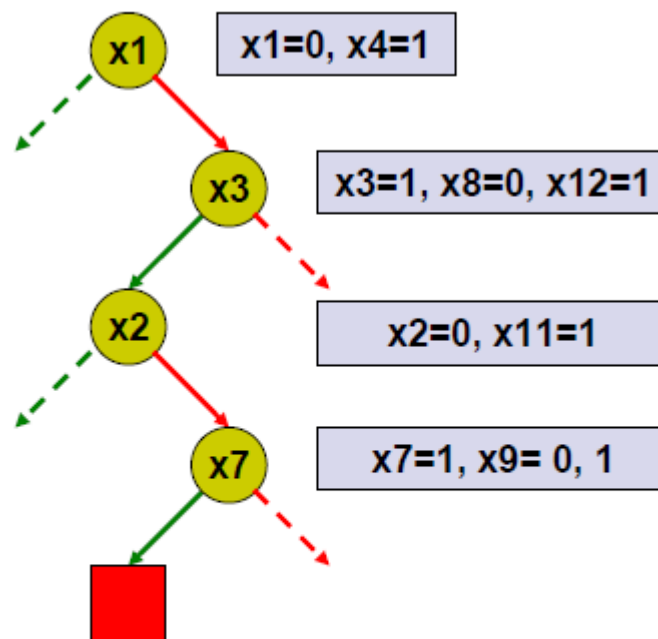
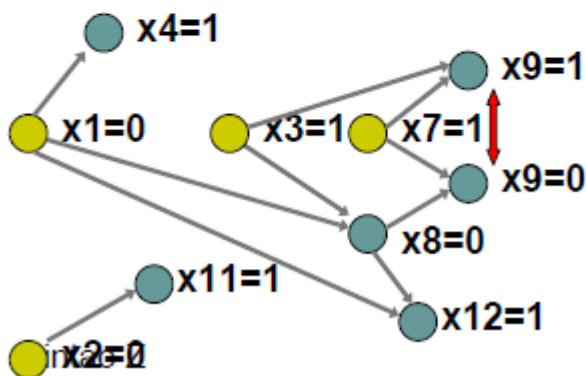
$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$



# 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

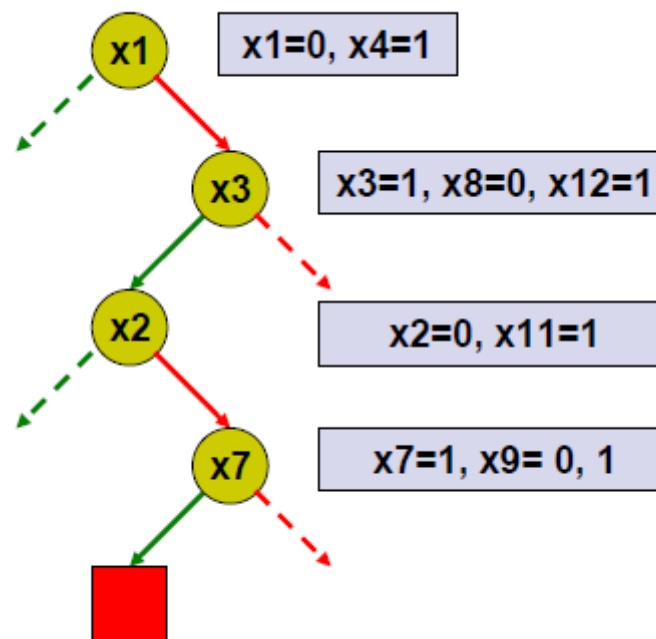






# DPLL的问题

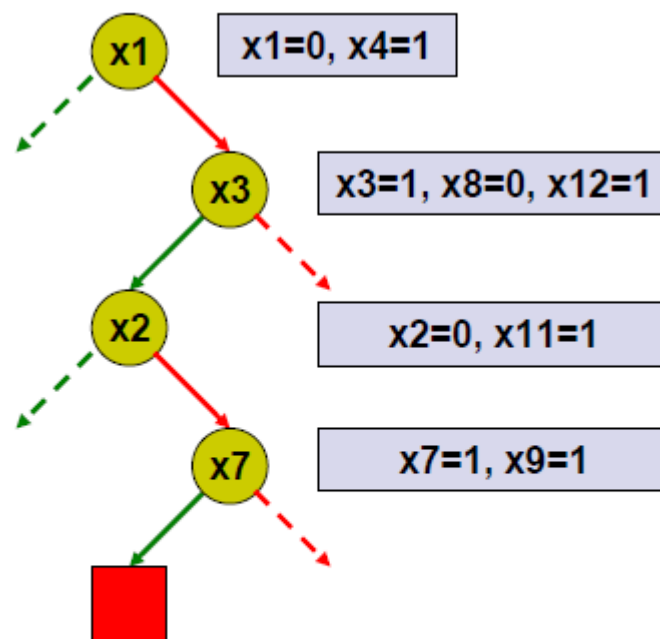
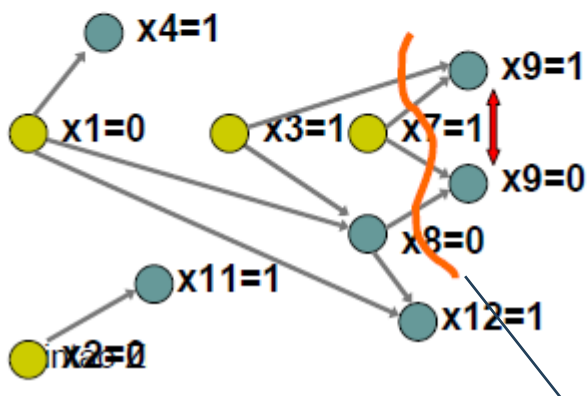
- 如果后续搜索把x7再次设置成了1，会重复出现该冲突



# CDCCL: 子句学习

$$\begin{aligned} &x_1 + x_4 \\ &x_1 + x_3' + x_8' \\ &x_1 + x_8 + x_{12} \\ &x_2 + x_{11} \\ &x_7' + x_3' + x_9 \\ &x_7' + x_8 + x_9' \\ &x_7 + x_8 + x_{10}' \\ &x_7 + x_{10} + x_{12}' \end{aligned}$$

## Step 11



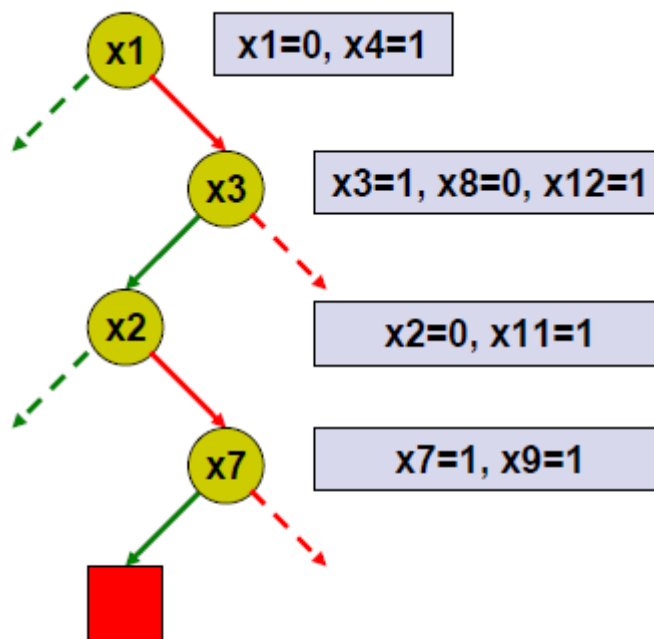
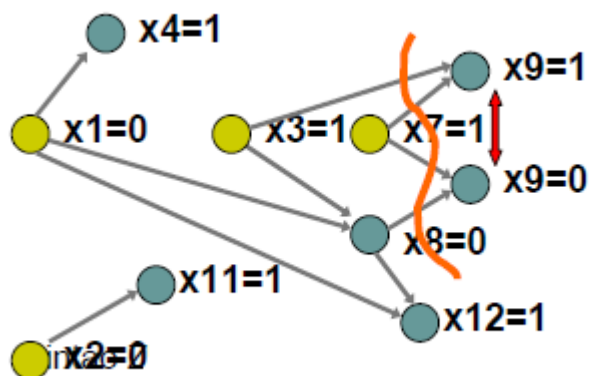
$x_3=1 \wedge x_7=1 \wedge x_8=0 \rightarrow \text{conflict}$

## 寻找一个切割，让所有决策节点和矛盾节点不连通

# 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



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

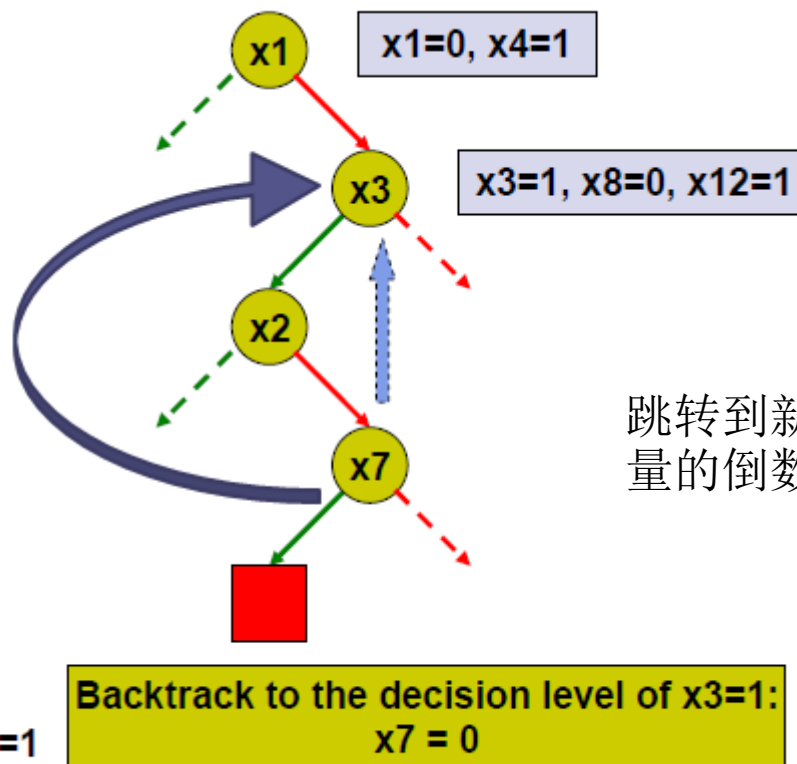
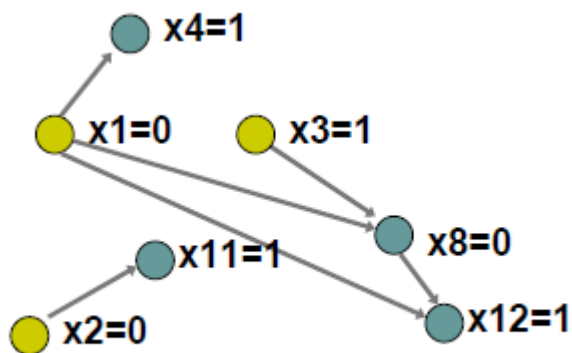
Add conflict clause:  $x3' + x7' + x8$



# 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

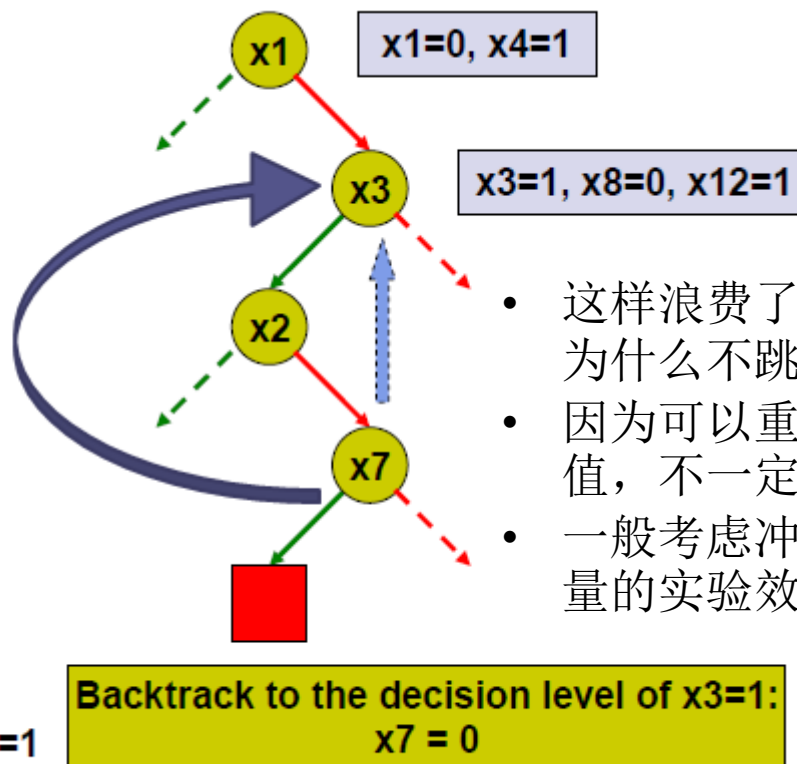
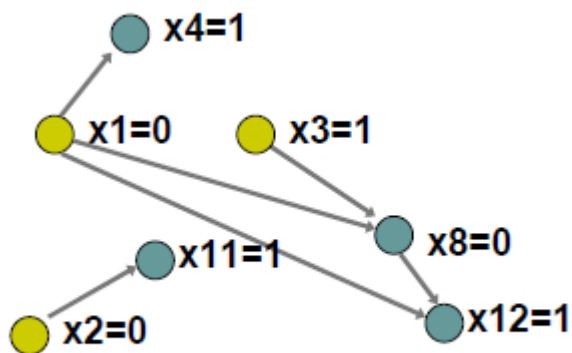


跳转到新加子句中变量的倒数第二次赋值

# CDCL: 子句学习

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

Step 14



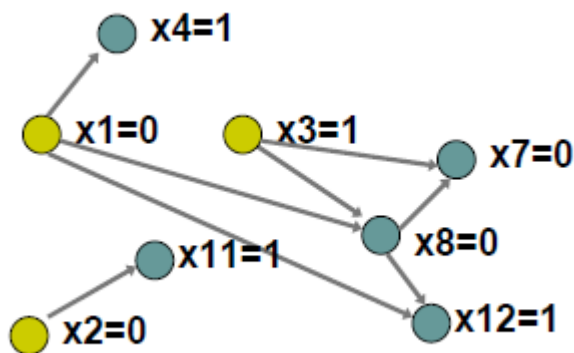
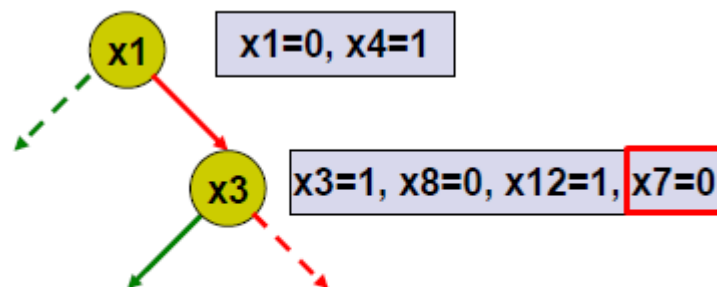
- 这样浪费了x2上的推导，为什么不跳回x2？
- 因为可以重新选变量赋值，不一定选x2。
- 一般考虑冲突重新选变量的实验效果更好。



# 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
- UNSAT情况举例
  - $\{1, 2\}$
  - $\{-1, 2\}$
  - $\{1, -2\}$
  - $\{-1, -2\}$
  - Assignment= $\{1\}$ ，推出矛盾，学会子句 $\{-1\}$
  - Assignment= $\{\}$ ，推出矛盾，UNSAT



# 如何选择切割

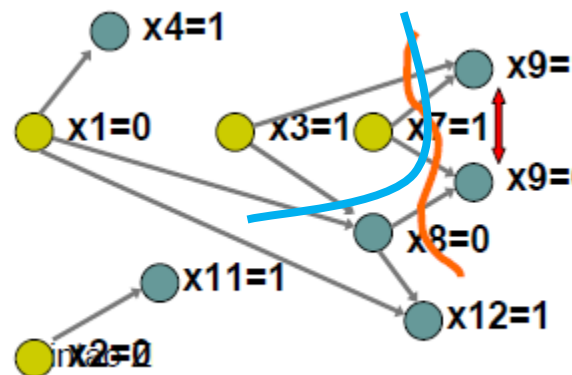
- 存在多种可能的切割，如何选择？

- 唯一蕴含点 Unique Implication Point: 最后一次决策到冲突的所有路径都要经过的节点

- 注意最后一次决策自身也是唯一蕴含点

- 切割距离冲突最近的唯一蕴含点的出边以及后继所有通往冲突路径的汇入边

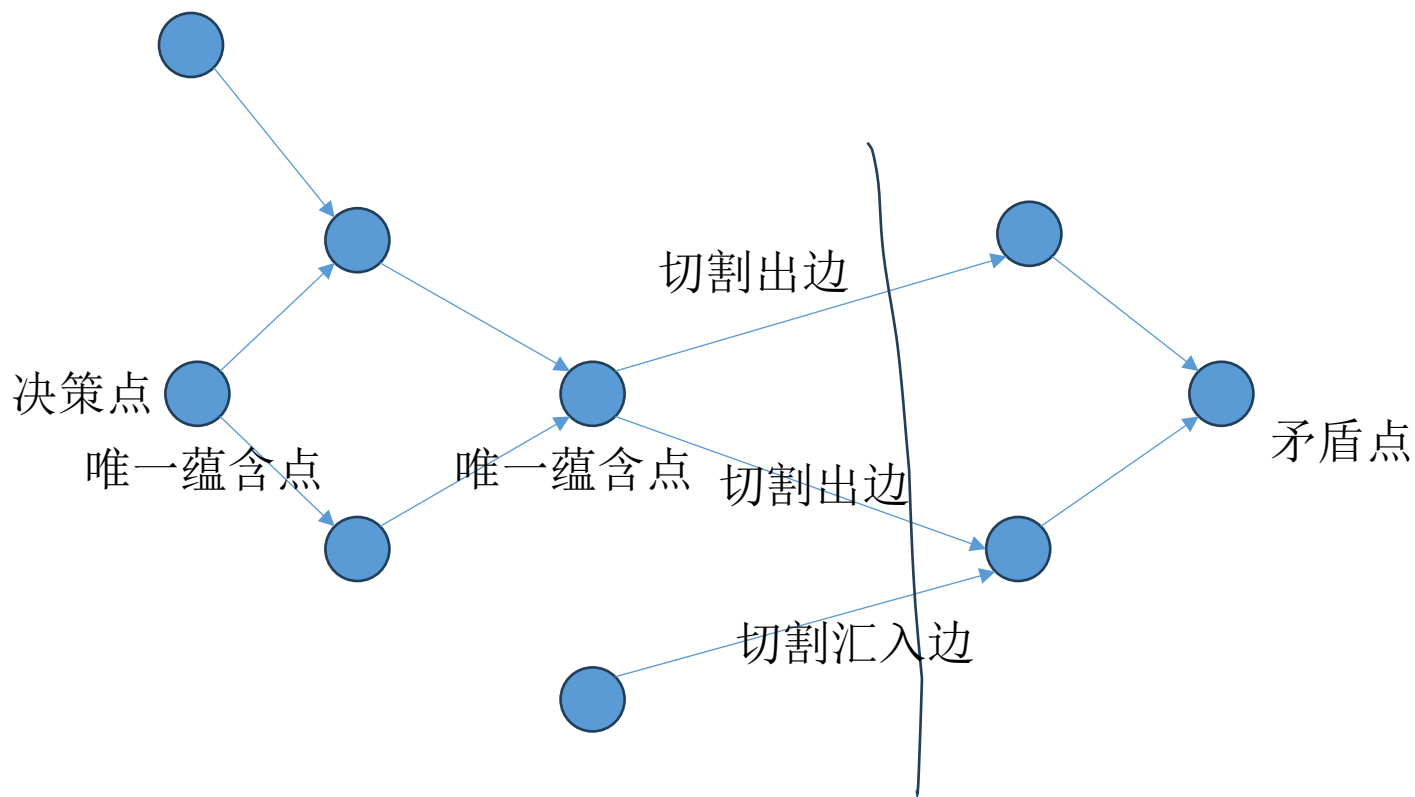
- 汇入边：目标节点在冲突路径上，但边自身不在任何冲突路径上
- 实验表明这样效果最好







# 切割示例





# 优化5：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5.1： 高效单元传播检测



- 如何知道一个子句除了一个文字之外都为假了？
  - $\{-1, 2, -3\} \Rightarrow 3$
- 基本做法： 每个子句设置变量，记录为假的文字的数量
  - 每次赋值需要更新所有相关子句的变量
  - 每次回溯也需要更新所有相关子句的变量
  - 能否减少更新的次数？

# 优化5.1： 高效单元传播检测



- 每个子句设置两个指针，指向两个未赋值的文字
  - 如果赋值让其中一个变为假了，就顺序寻找下一个未赋值的文字
  - 如果找不到，就可以单元传播
- 时间复杂度分析：
  - 每次赋值的时候，更新子句的均摊时间复杂度不变
  - 每次回溯的时候，无需再对每条子句做更新



## 优化5.2：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

# SAT求解的另一种算法——本地搜索



- 首先猜测一个解
- 如果满足，则SAT
- 如果不满足，则尝试翻转解中的某些变量，重复上述过程直到遍历所有的解
- 如何猜测解和如何选择变量翻转是本地搜索设计的关键
- 本地搜索和CDCL互补，在一些CDCL效果不好的场合能起到很好的作用



# 作业

- 用CDCL求解下面的SAT问题，画出决策树，并指出在每次回溯的时候学到了什么子句
  - 假设变量决策顺序按数字顺序，并总是首先赋值成真
  - 假设唯一蕴含点策略
  - 不考虑预处理
  - $\{1, 2\}$
  - $\{2, 3\}$
  - $\{-1, -4, 5\}$
  - $\{-1, 4, 6\}$
  - $\{-1, -5, 6\}$
  - $\{-1, 4, -6\}$
  - $\{-1, -5, -6\}$



# 参考资料

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