



软件分析

约束求解

熊英飞
北京大学
2016



路径敏感性

- 路径非敏感分析：不考虑程序中的路径可行性，忽略分支循环语句中的条件
- 路径敏感分析：考虑程序中的路径可行性，只分析可能的路径



路径敏感vs路径非敏感

```
int f(int x){  
    if (x > 5)  
        x = 10;  
    return x;  
}
```

假设输入的区间为(5, 10]

- 使用路径非敏感的区间分析，得到函数的返回值为
 - (5, 10]
- 使用路径敏感的区间分析，得到函数的返回值为
 - [10, 10]

给数据流分析添加基本路径敏感性



```
int f(int x){  
    if (x > 5) {  
        assert(x > 5);  
        x = 10; }  
    else {  
        assert(x <= 5);  
    }  
    return x;  
}
```

假设输入的区间为(5, 10]

- 给每条分支添加assert语句
- assert(x>5)转换函数
 - $f(x) = x \cap (5, +\infty)$
- assert(x>5)转换函数
 - $f(x) = x \cap (-\infty, 5]$



路径敏感的数据流分析

- 优点:
 - 完全兼容已有数据流分析框架
 - 利用已有技术直接支持循环、过程间等复杂分析
- 缺点:
 - 当前数据流分析内容必须与条件判断兼容
 - 给定条件 $x > 5$ ，如何做reaching definition分析？
 - 给定条件 $x > y$ ，如何做路径敏感的区间分析？
 - 对条件判断的精度不高
 - `if (y > 0) x = 10; else x=1; if (x<6&& x>2) x=20;`
 - x 不会赋值成20，但是判断不出来
 - 无法分路径输出结果



路径敏感分析

- 符号执行、模型检查等
- 关键问题：如何知道哪些路径是可行的？
 - 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
 - 如果可满足，给出一组赋值
 - 如果不可满足，给出最小矛盾集minimal unsatisfiable core
- 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



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

- 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
 - 有限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



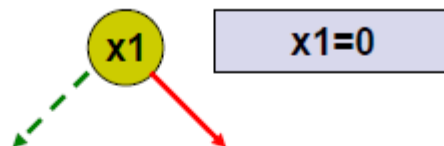
- 由微软亚洲研究院的Lintao Zhang在普林斯顿读书时提出
- 2000年初期大幅提升SAT效率的重要因素之一



一个DPLL的例子

Step 1

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



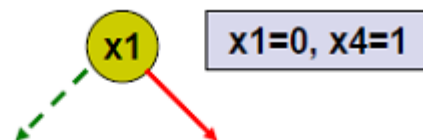
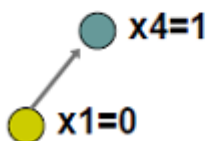
● $x_1=0$



一个DPLL的例子

Step 2

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

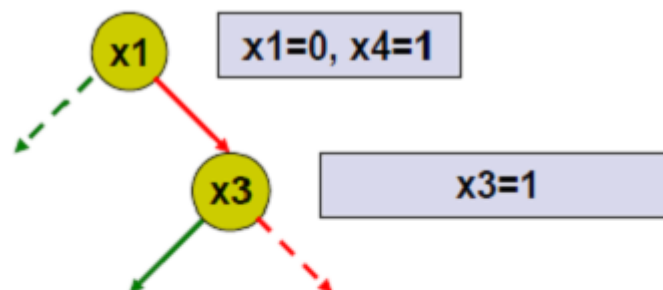
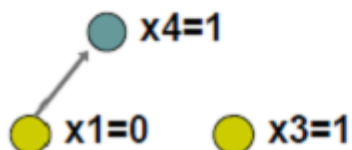




一个DPLL的例子

Step 3

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

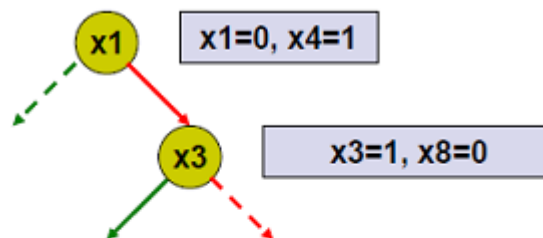
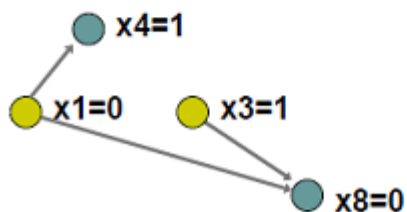




一个DPLL的例子

Step 4

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

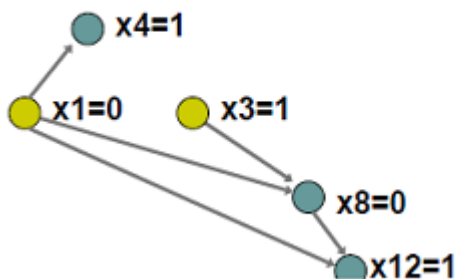
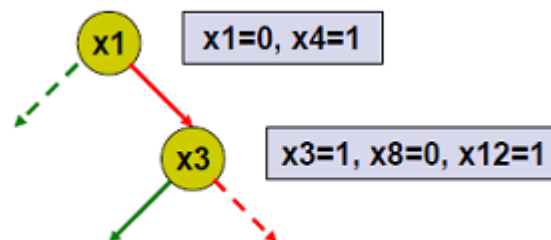




一个DPLL的例子

Step 5

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

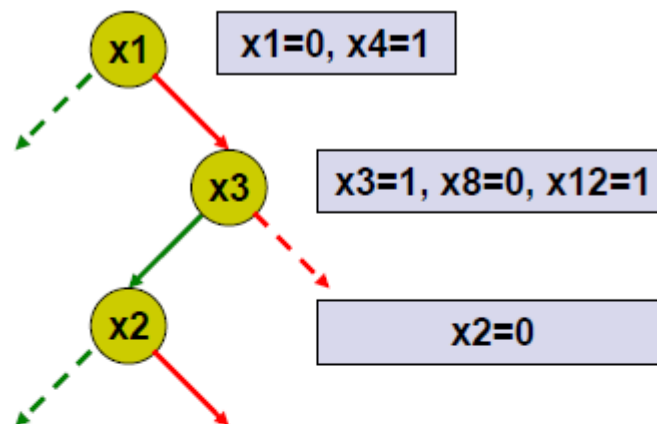
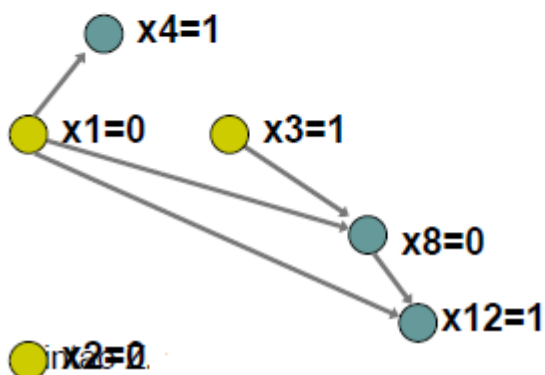




一个DPLL的例子

Step 6

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

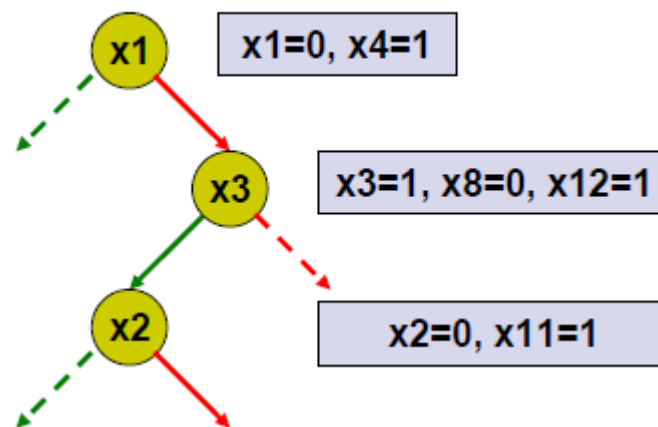
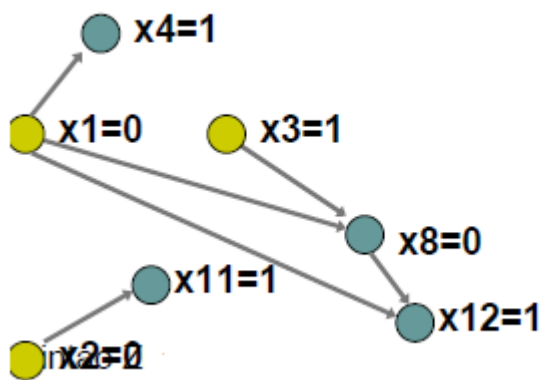




一个DPLL的例子

Step 7

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

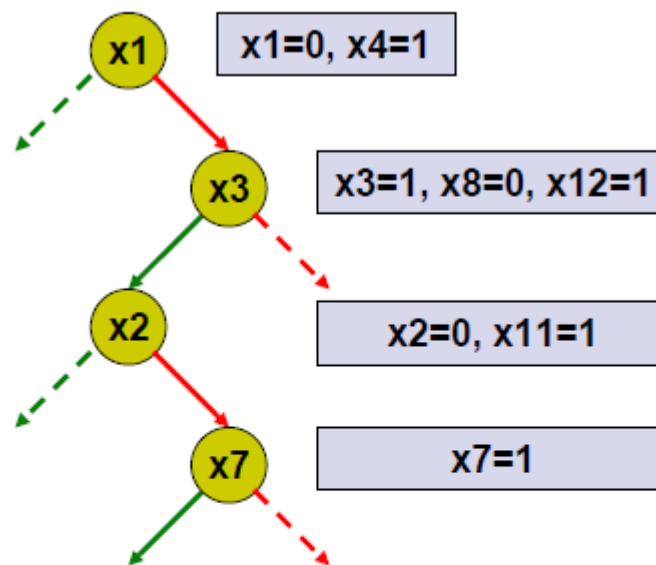
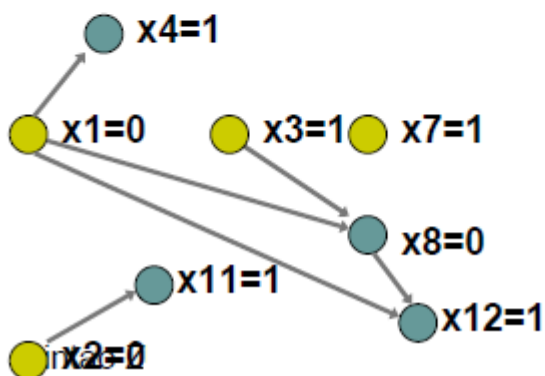




一个DPLL的例子

Step 9

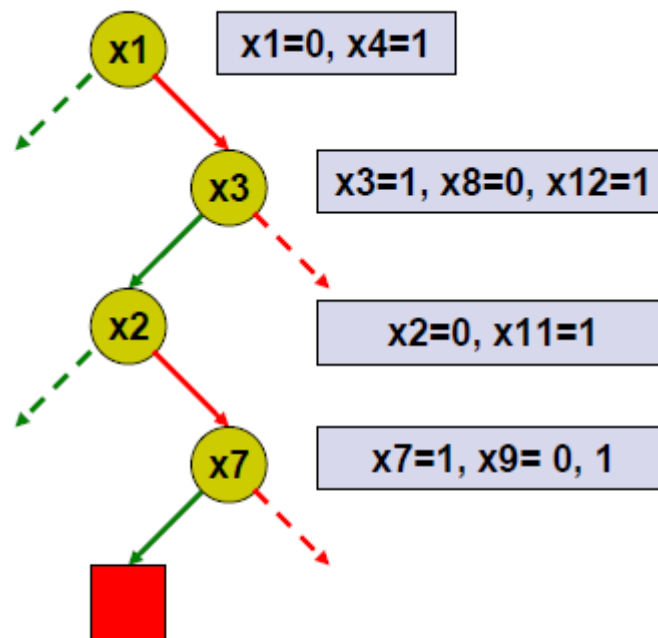
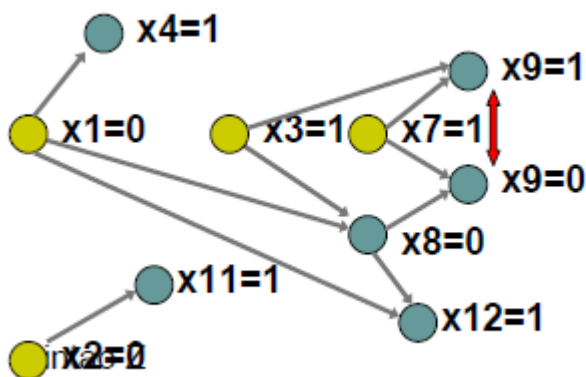
$x_1 + x_4$
 $x_1 + x_3' + x_8'$
 $x_1 + x_8 + x_{12}$
 $x_2 + x_{11}$
 $x_7' + x_3' + x_9$
 $x_7' + x_8 + x_9'$
 $x_7 + x_8 + x_{10}'$
 $x_7 + x_{10} + x_{12}'$



一个DPLL的例子

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

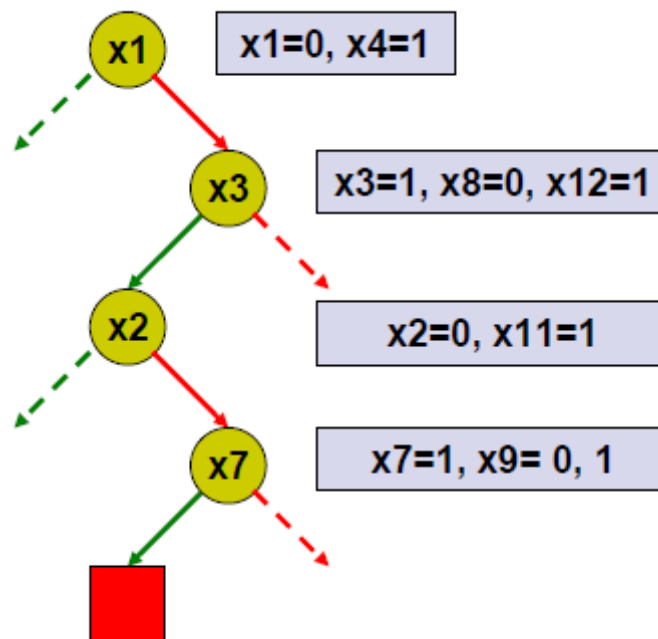
Step 10





DPLL的问题

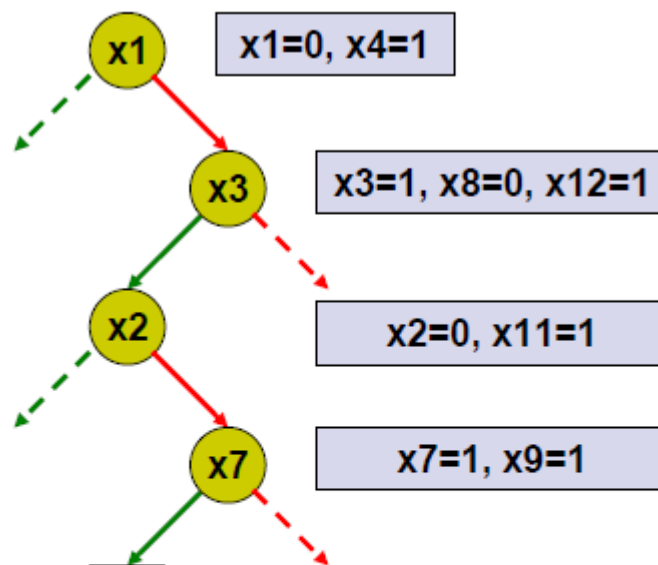
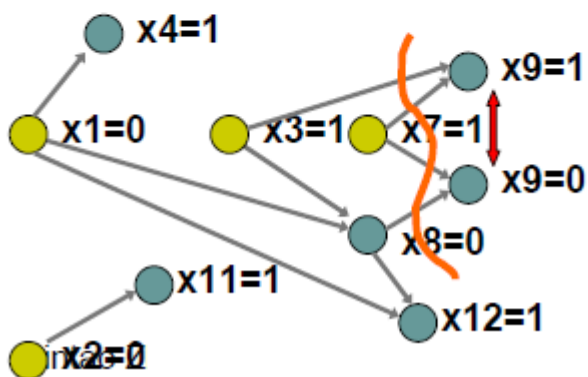
- 如果后续搜索把x2设置成1，x7再次设置成了1，会重复出现该冲突



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



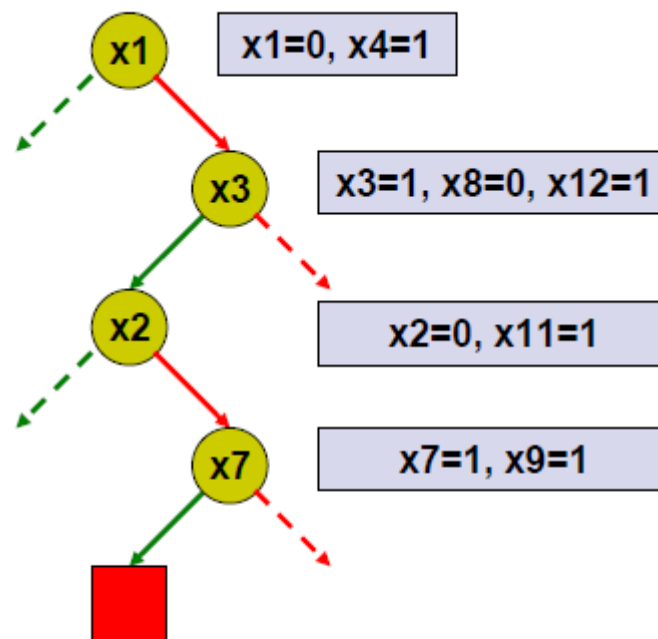
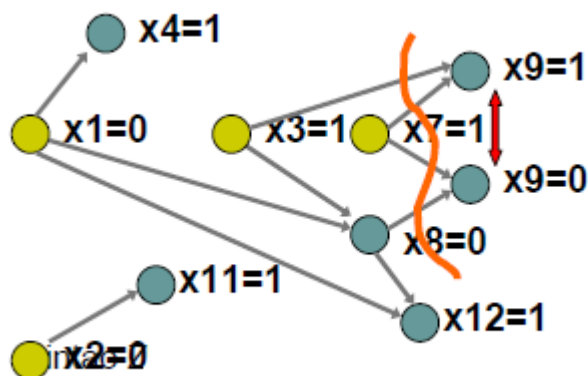
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow \text{conflict}$



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



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

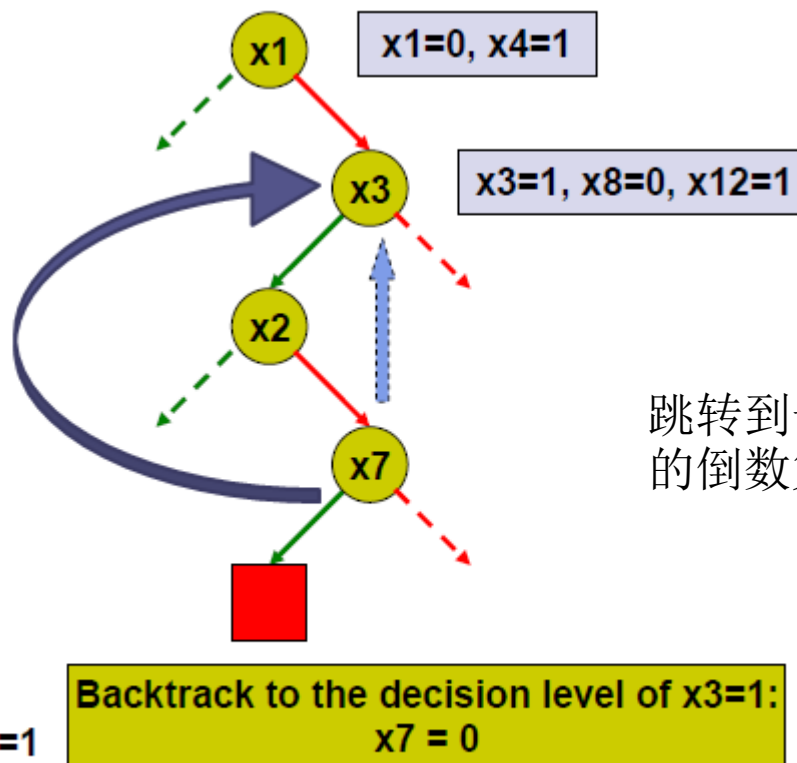
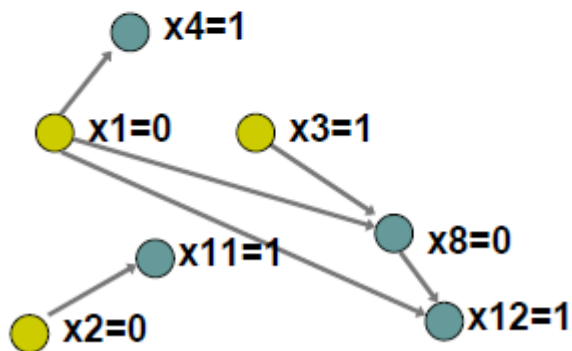
Add conflict clause: $x3' + x7' + x8$



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



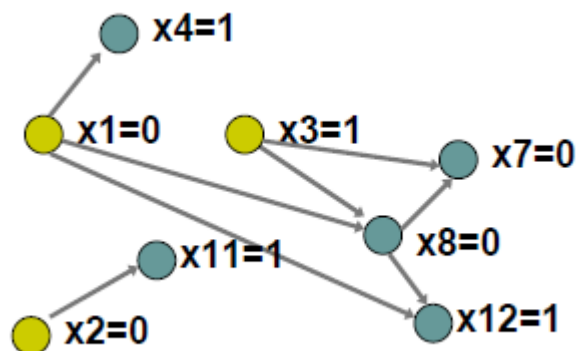
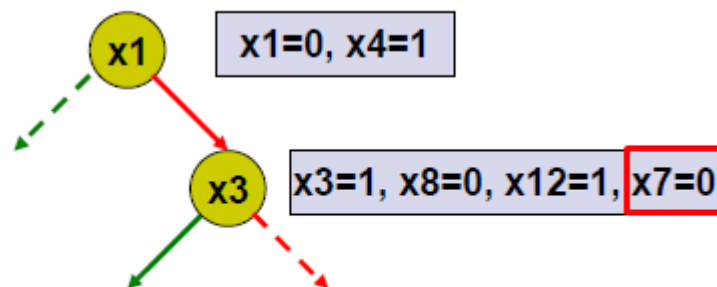
跳转到子句中变量的
倒数第二次赋值



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





优化4：冲突导向的子句学习

CDCL Conflict-Driven Clause Learning

```
cdcl() {  
  assign={};  
  while (true) {  
    assign'=赋值推导(assign);  
    if (assign'有冲突) {  
      if (assign为空) return false;  
      添加新约束;  
      撤销赋值;  
    } else {  
      if (assign'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



SMT Solver

using the slides from Albert Oliveras



SMT Solver的使用

- SMT-LIB
 - 标准的SMT输入格式
 - 被几乎所有的SMT Solver支持
 - 用于每年的SMT比赛中



SMT-LIB by Example

- `> (declare-fun x () Int)`
- `> (declare-fun y () Int)`
- `> (assert (= (+ x (* 2 y)) 20))`
- `> (assert (= (- x y) 2))`
- `> (check-sat)`
- `sat`
- `> (get-value (x y))`
- `((x 8)(y 6))`
- `> (exit)`



Scope

- > (declare-fun x () Int)
- > (declare-fun y () Int)
- > (assert (= (+ x (* 2 y)) 20))
- > (push 1)
- > (assert (= (- x y) 2))
- > (check-sat)
- sat
- > (pop 1)
- > (push 1)
- > (assert (= (- x y) 3))
- > (check-sat)
- unsat
- > (pop 1)
- > (exit)



Defining a new type

- > (declare-sort A 0)
- > (declare-fun a () A)
- > (declare-fun b () A)
- > (declare-fun c () A)
- > (declare-fun d () A)
- > (declare-fun e () A)
- > (assert (or (= c a)(= c b)))
- > (assert (or (= d a)(= d b)))
- > (assert (or (= e a)(= e b)))
- > (push 1)
- > (distinct c d)
- > (check-sat)
- sat
- > (pop 1)
- > (push 1)
- > (distinct c d e)
- > (check-sat)
- unsat
- > (pop 1)
- > (exit)



常见的SMT Solver

- Z3
 - 微软开发
 - 目前使用最广稳定性最好
 - 仅支持Windows，不开源
- Yices 2
 - Z3之前使用最广稳定性最好的Solver
 - 由Z3的作者在加入微软之前撰写
 - 支持所有平台，开源



课后作业

- 下载安装任意SMT Solver
- 发邮件给助教，回答如下问题：
 - 该SMT Solver的名字
 - 该SMT Solver支持的Theory
 - 构造该SMT Solver无法求解的约束，将运行结果截屏附在邮件中
 - 解释该SMT Solver为什么不能求解这个约束