数

理

逻

辑

复

习

# 知识点（重点）

1. BDD—>OBDD 谓词公式的化简
2. SAT问题求解中的相关问题：
   1. Unit clause rule
   2. BCP（reduce方法）
   3. 冲突分析和学习
   4. Zchaff（chaff）中用数组表示子句（相关的数据结构）
3. 将程序转换为谓词公式（FDS）
4. Alloy模型语言

# BDD二叉决策图

## BDD

引入：数字电路的表现形式：逻辑表达式、真值表、卡诺图 ｛缺点：存在冗余 ，所以使用BDD｝

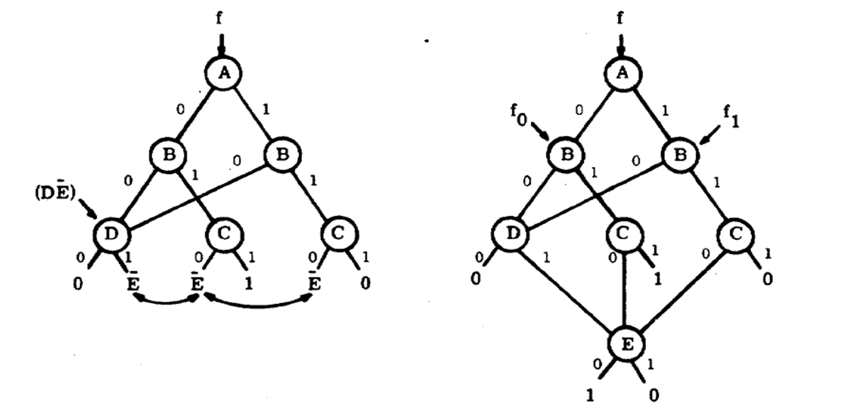
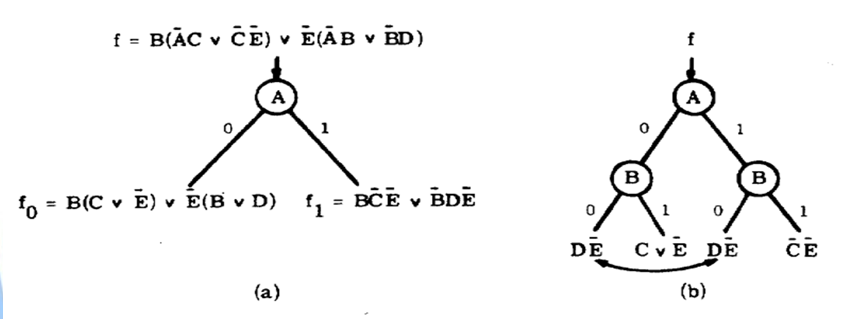
定义：BDD描述了一个过程，这个过程按照给定的值（0/1）进行向下探索，直到终点。

BDD的生成（使用shannon公式 ）：

步骤：

1. 固定一个变量，画出此变量的节点及0、1分支（分支的结果表示的是一个逻辑表达式）
2. 观察产生的分支是否可以合并（有的分支需要化简）：如果可以合并就合并；否则执行步骤1
3. 直到分支节点的值为0/1为止。

Example：



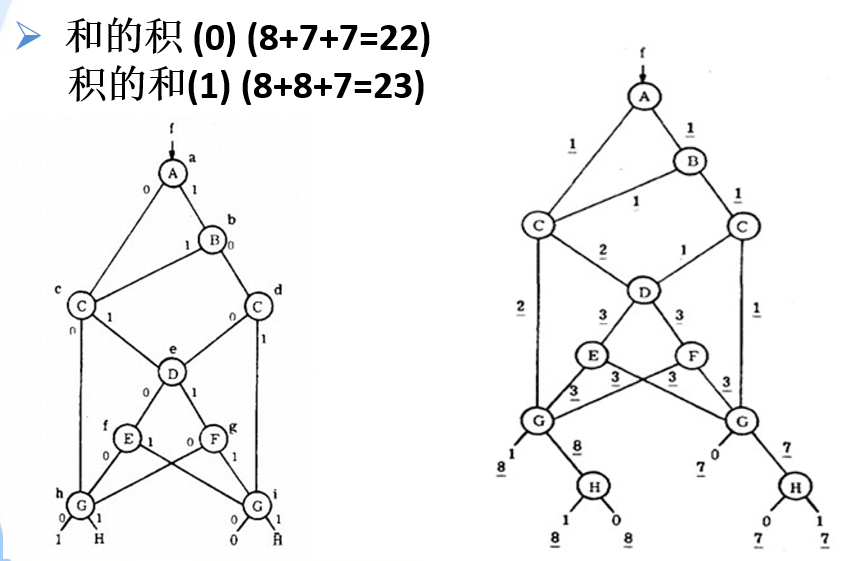
BDD的分析：

一、计算BDD的输出、｛给定输入---------确定输出｝

二、计算“和的积”和“积的和”的个数 ｛“和的积”：真值为0的个数；“积的和”：真值为1的个数｝

步骤：a.最上层的分支赋值为1；b.其余节点的分支的赋值为入度权值之和。

Example：



## OBDD

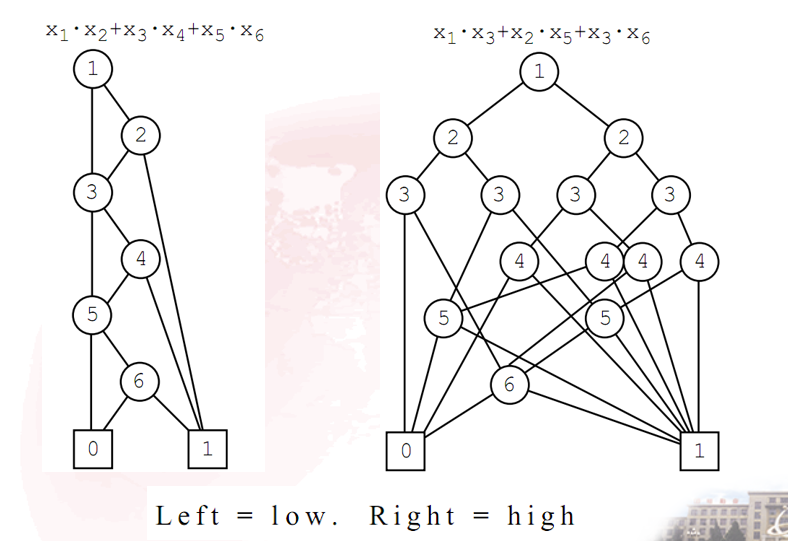
引入： BDD是无序的；一个布尔表达式对应多个BDD｛解决方案：使用OBDD｝

和BDD比较：与BDD没有什么区别，就是有序，

相关定义，看老师的PPT

OBDD的作用：简化function graph 从而 减少时间复杂度

Example：



使用OBDD对function graph进行化简reduced：

理解程序、跟着 PPT上的例子走一遍。

【

冗余的情况：

叶子节点：全部加入key，value中，对其进行消除冗余

非叶子节点：

一个结点的左右子树相同：直接消除冗余

两个结点的子树同构：加入key,value中，对其进行消除冗余

】

# SAT 之 DPLL算法

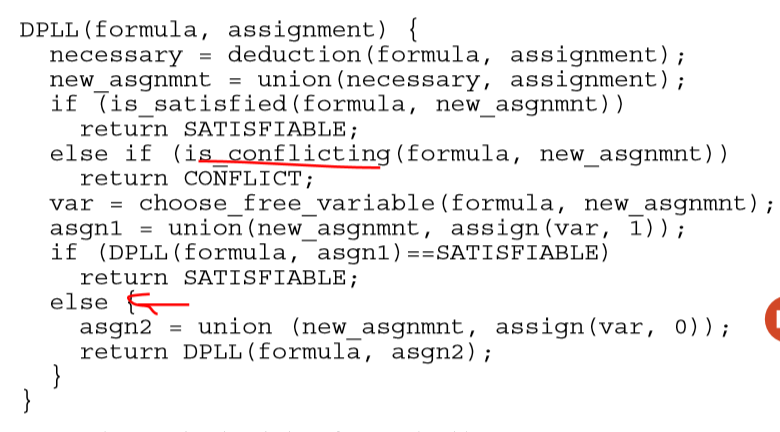
SAT问题：对于一个合取范式（CNF），是否存在一组真值指派，使得CNF的所有子句的取值都为真，如果存在，就是SAT；如果不存在，就是UNSAT。

DPLL算法的引入： 解决SAT问题

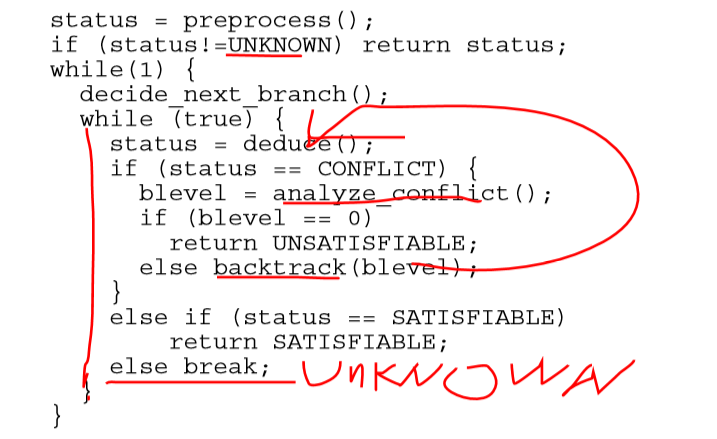
DPLL算法的核心思想：依次对 CNF 实例的每个变量进行赋值，其搜索空间可以用一个二叉树来表示，树中的每个节点对应一个变量，取值只能为0或 1，左右子树分别表示变量取0或 1的情况，从二叉树中根节点到叶子节点的一条路径就表示 CNF 实例中的一组变量赋值序列，DPLL 算法就是对这棵二叉树从根节点开始进行 DFS（深度优先搜索）遍历所有的通路，以找到使问题可满足的解。

DPLL的实现： 递归 迭代

**递归（原始）**



**迭代（目前主流）**



迭代实现比递归实现的好处、优点：

* 递归的速度慢，且容易发生溢出；同时递归相对于迭代还有很多的自身缺陷。
* 当发生冲突的时候，就需要使用回溯来解决。

递归方法：回到上一层，将变量的取值翻转，继续向下搜索，采用的是时序回溯。这样做的结果很可能使冲突依旧存在，因为上一层的赋值也许不是产生冲突的直接原因，从而浪费一次计算时间，效率低下。

迭代方法：对冲突进行分析和学习，并回跳多个决策层，通过分析，找到冲突产生的根本原因；通过学习，将此次产生冲突的情况加入到子句集中，进而帮助缩小搜索范围；采用的非时序回溯（智能回溯）。

迭代方法中的deduce()： 当一个变量赋值后，通过推理可以减少不必要的搜索。

依赖：单元子句规则

单元子句：在一个子句中，除了一个文字外，其余的所有文字均赋值且该子句表现为假。【剩余的一个文字叫做单元文字】

BCP（布尔约束传播）： 目标是识别出所有的单元子句并对单元文字进行赋值，能够 减少搜索空间 或者 提前逼出冲突。

实现BCP的方法：

1counter方法：

|  |  |
| --- | --- |
| 每个变量v有两个list | 含有(V)的子句 |
| 含有(非V)的子句 |

|  |  |
| --- | --- |
| 每个子句有两个计数器 | 体现为真的字母数 |
| 体现为假的字母数 |

2head/tail方法：

|  |  |
| --- | --- |
| 每个子句有两个指针，head和tail | Head指向子句的第一变量 |
| Tail指向子句的最后一个变量 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Head为(V)的子句 | Head为(非V)的子句 | Tail为(V)的子句 | Tail为(非V)的子句 |
| V |  |  |  |  |

3 2-literal watching

|  |  |
| --- | --- |
| 每个子句有两个指针，指针没有限制 |  |
|  |

|  |  |  |
| --- | --- | --- |
|  | Pos\_watched(v) | Neg\_wathed(V) |
| V |  |  |

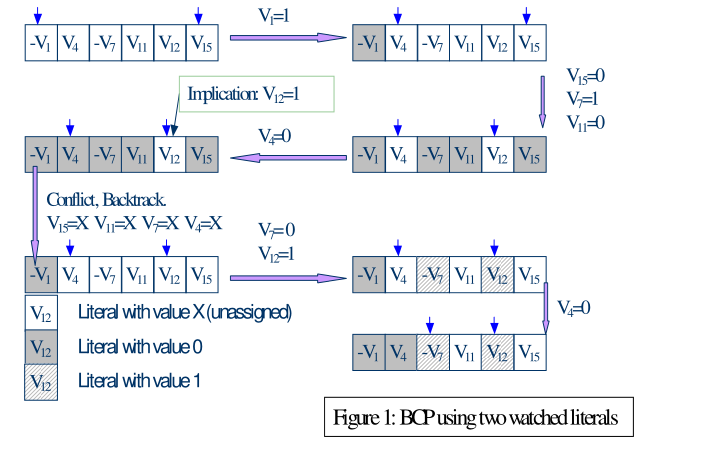
指向任意阶段中与之对应的所有 受监视文字的指针

冲突的分析和学习：analyze\_conflict()

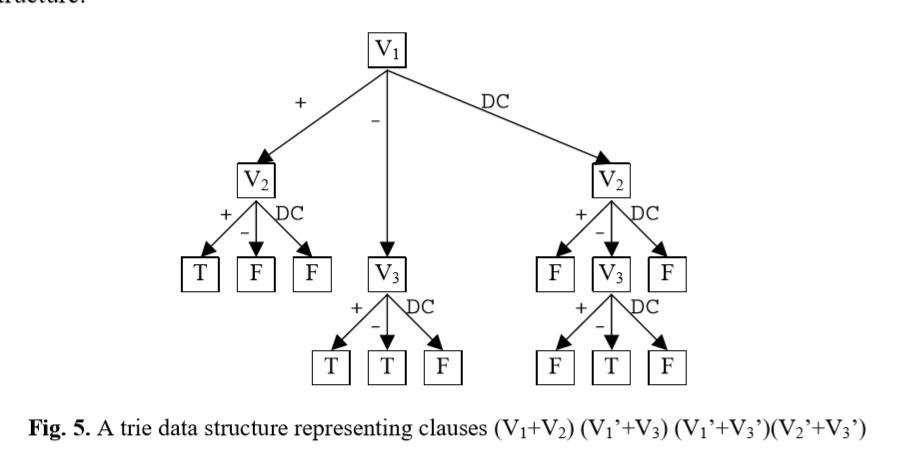
数据的存储方式：

**链表和指针数组的方式**

**Chaff使用数据的方式**

****

**SATO采用的是Tire方式**



指针和数组存储方式的比较：

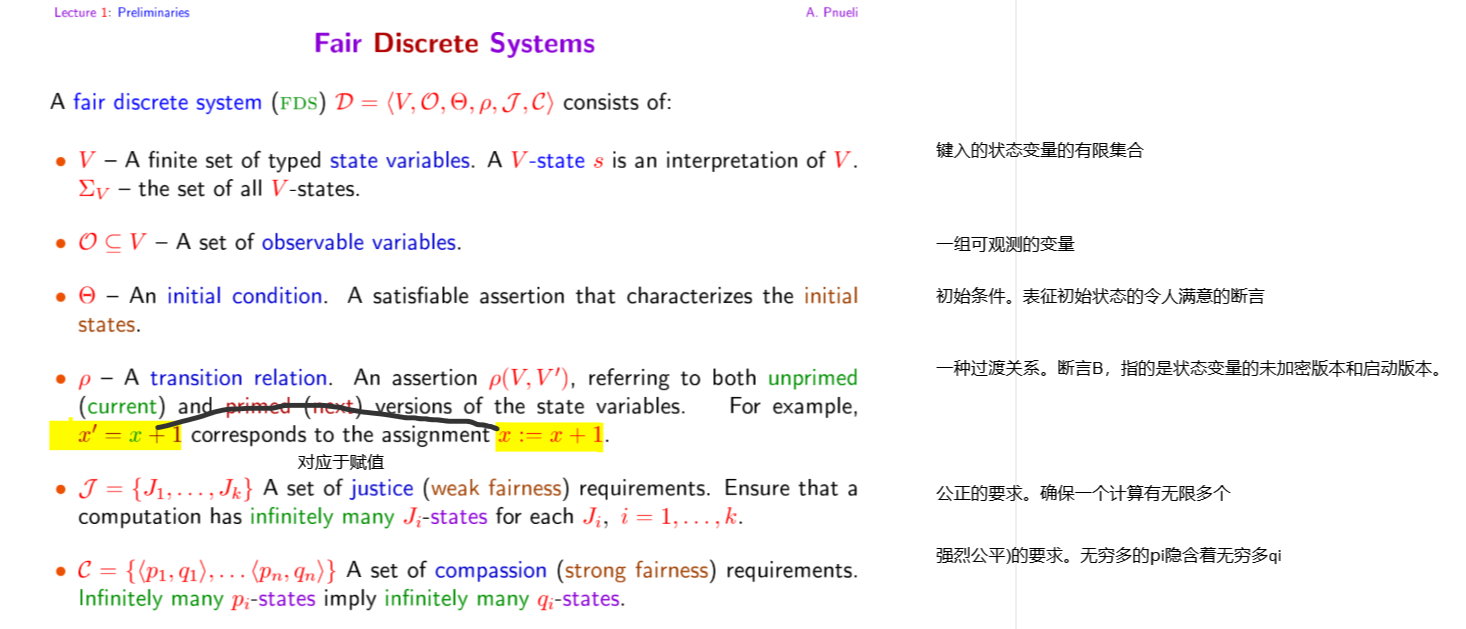
1数组在存储器上利用率高

2数组占用连续的内存空间，访问局部速度加快

3数组与链表相比较，就告诉缓存丢失而言，有一个很大的优势，它在求解过程中大大加快求解速度。

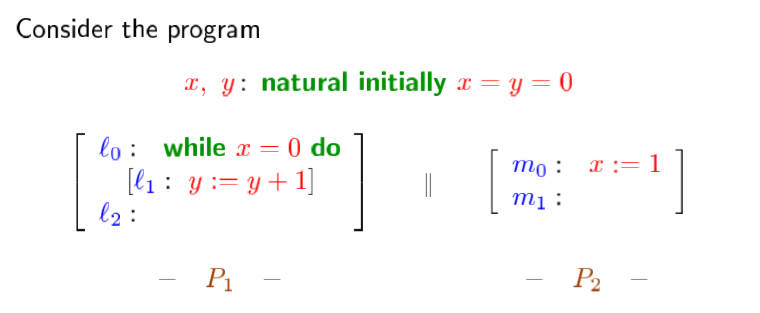
4链表的存取利用率低，同时在处理过程中，因为缺少局部访问的能力，往往会造成很多的缓存丢失。

# 将程序翻译为逻辑FDS

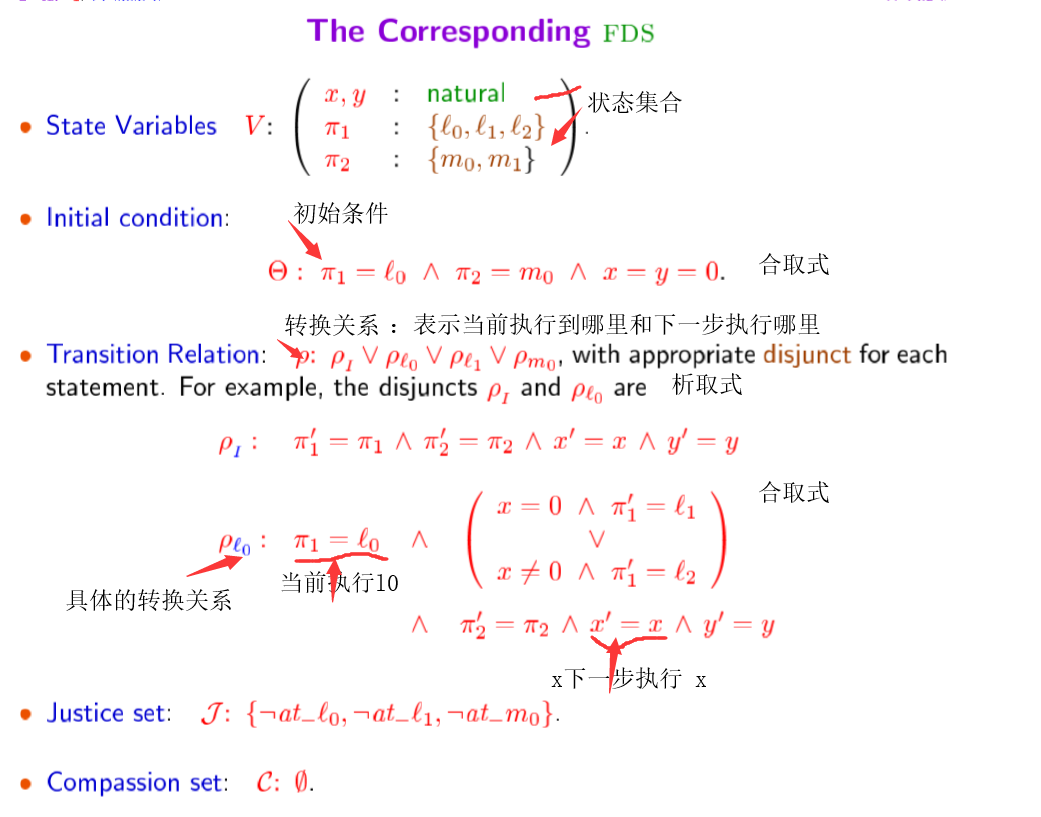


Example：

**程序：**



**对应的逻辑公式**



# Alloy