

- Boolean Algebra Basics
 - Cofactor
 - Cofactor properties
 - 香农展开
- Boolean Difference
 - Boolean difference properties
- Quantification Operators
 - Universal quantification
 - Existential quantification
 - Additional properties
- Application To Logic Network Repair
- Recursive tautology
 - Tautology
 - Positional Cube Notation(PCN)
 - Cofactor与tautology
 - 判断tautology的方法
 - Unate recursive paradigm
 - 求取Cofactor
 - Positive cofactor:
 - Negative cofactor:
 - 终止规则
 - 变量选取规则
 - 伪代码
- BDD
 - BDD简化(ROBDD)
 - BDD共享
 - BDD构建
 - BDD的用途举例
 - 判断布尔函数是否等价
 - 寻找使得两个布尔函数取不同值的输入
 - tautology checking
 - SAT
 - BDD变量排序(经验方法)
- SAT
 - CNF
 - Clause的状态
 - Recursively方法DPLL
 - Boolean constraint propagation
 - BCP终止规则
 - 电路等价性校验转化为SAT

- 组合逻辑电路转化为CNF
 - Gate consistency function
 - 各种gate的gate consistency function
 - 转化方法
- Two level logic
 - Two level minimization
 - Reduce-Expand-Irredundant optimization
 - Expand step detail
 - Build off set
 - Build the blocking matrix
 - 矩阵覆盖
- Multi-level logic
 - Boolean logic network model
 - Optimize on boolean logic network
 - Operations on boolean logic Network
 - Algebraic Division
 - 伪代码
 - Factoring
 - Kernels and co-kernels
 - Find Kernels
 - Summary
 - Divisor extraction: single cube case
 - Cube-literal matrix
 - rectangle
 - Prime rectangle
 - Common single-cube divisor
 - Divisor extraction: multiple cube case
 - Co-kernel-cube matrix
 - Find prime rectangle
 - Summary
 - Implicit don't cares
 - Multi-level don't cares
 - Satisfiability don't cares
 - Controllability don't cares
 - Observability don't cares
 - Summary

Boolean Algebra Basics

Cofactor

令 $F(x_1, \dots, x_n)$ 为一个 n 元布尔函数

- Positive cofactor related to x_i

$$F_{x_i} = F(x_1, \dots, x_i = 1, \dots, x_n)$$

- Negative Cofactor related to x_i

$$F_{x'_i} = F(x_1, \dots, x_i = 0, \dots, x_n)$$

Cofactor properties

- 取反

$$(F')_x = (F_x)'$$

- 二元运算

$$(F \cdot G)_x = F_x \cdot G_x$$

$$(F + G)_x = F_x + G_x$$

$$(F \oplus G)_x = F_x \oplus G_x$$

香农展开

$$F(x_1, \dots, x_n) = x_i F_{x_i} + x'_i F_{x'_i}$$

也可以对多个变量进行展开

$$F(x_1, \dots, x_n) = x_i x_j F_{x_i x_j} + x_i x'_j F(x_i x'_j) + x'_i x_j F_{x'_i x_j} + x'_i x'_j F_{x'_i x'_j}$$

其中 $F_{x_i x'_j} = F(x_1, \dots, x_i = 1, \dots, x_j = 0, \dots, x_n)$, 其他项依此类推。

Boolean Difference

- 布尔函数 F 关于 x 的导数定义为

$$\frac{\partial F}{\partial x} = F_x \oplus F_{x'}$$

意义 $\frac{\partial F}{\partial x} = 1$ 时, x 的取值变化时 F 也会发生变化

Boolean difference properties

- 导数与变量顺序无关

$$\frac{\partial F}{\partial x \partial y} = \frac{\partial F}{\partial y \partial x}$$

- 导数的异或等于异或的导数

$$\frac{F \oplus G}{\partial x} = \frac{\partial F}{\partial x} \oplus \frac{\partial G}{\partial x}$$

- 常量函数的导数为0

- 与和或的导数

$$\frac{\partial}{\partial x} (F \cdot G) = (F \cdot \frac{\partial G}{\partial x}) \oplus (G \cdot \frac{\partial F}{\partial x}) \oplus (\frac{\partial F}{\partial x} \frac{\partial G}{\partial x})$$

$$\frac{\partial}{\partial x} (F + G) = (F' \cdot \frac{\partial G}{\partial x}) \oplus (G' \cdot \frac{\partial F}{\partial x}) \oplus (\frac{\partial F}{\partial x} \frac{\partial G}{\partial x})$$

Quantification Operators

Universal quantification

$$\forall_x F = F_x \cdot F_{x'}$$

意义: $\forall_x F = 1$ 时其他变量的值能够让 F 对于任意的 x 都是 1

$$e.g : \text{令 } F = xy + x'z$$

$$F_x = y, F_{x'} = z$$

$$\forall_x F = F_x \cdot F_{x'} = yz$$

$$\forall_x F = 1 \Rightarrow yz = 1 \Rightarrow y = 1, z = 1$$

$$F_{y=1,z=1} = x + x' = 1 \Rightarrow \forall x \text{都有 } F_{y=1,z=1} = 1$$

Existential quantification

$$\exists_x F = F_x + F_{x'}$$

意义: $\exists_x F = 1$ 时其他变量的值能使得存在一个 x 让 F 为 1

$$e.g : \text{令 } F = xy + x'z$$

$$F_x = y, F_{x'} = z$$

$$\exists_x F = F_x + F_{x'} = y + z$$

$\exists_x F = 1 \Rightarrow y = 1 \text{ or } z = 1$

$F_{y=1} = x + x'z \Rightarrow \exists x = 1 \text{ 使得 } F_{y=1} = 1$

$F_{z=1} = xy + x' \Rightarrow \exists x = 0 \text{ 使得 } F_{y=1} = 1$

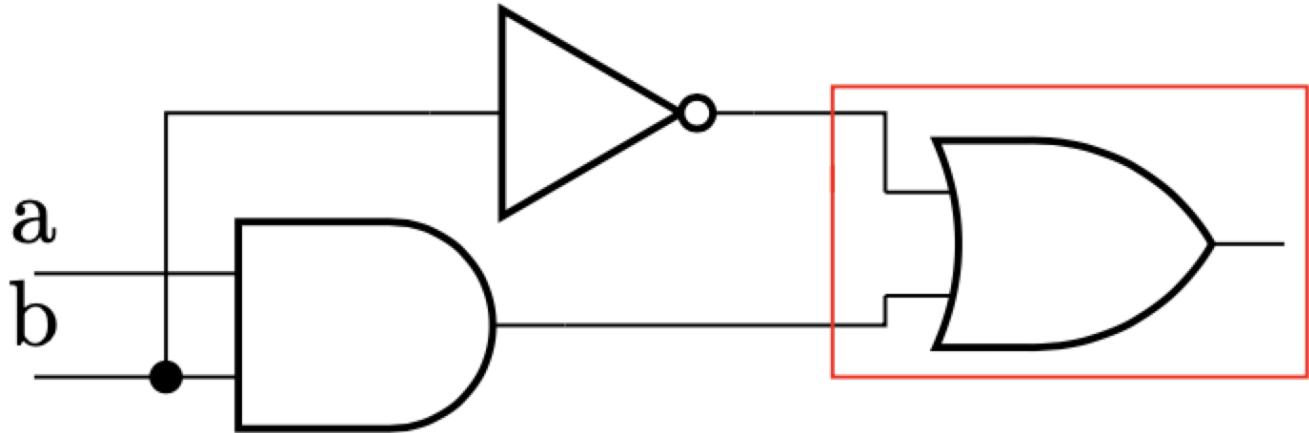
Additional properties

$$\forall_{xy} F = \forall_x (\forall_y F) = F_{xy} \cdot F_{xy'} \cdot F_{x'y} \cdot F_{x'y'}$$

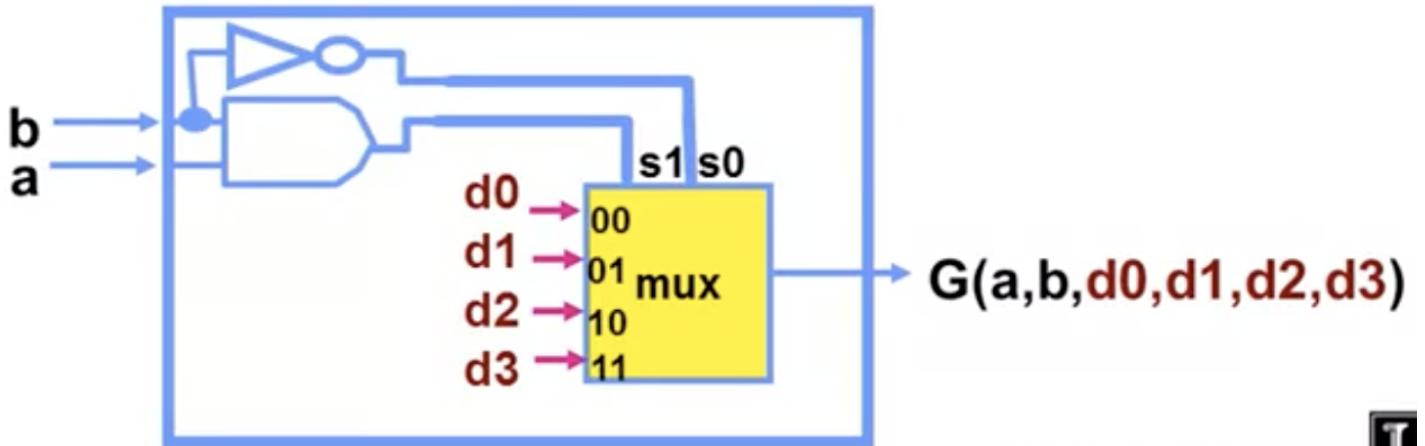
$$\exists_{xy} F = \exists_x (\exists_y F) = F_{xy} + F_{xy'} + F_{x'y} + F_{x'y'}$$

Application To Logic Network Repair

需要一个电路来实现 $f(a, b) = ab + b'$ 的逻辑功能，假设实现错了一个gate，如图中的红色框部分，需要修复它并得到它正确实现时的逻辑门。

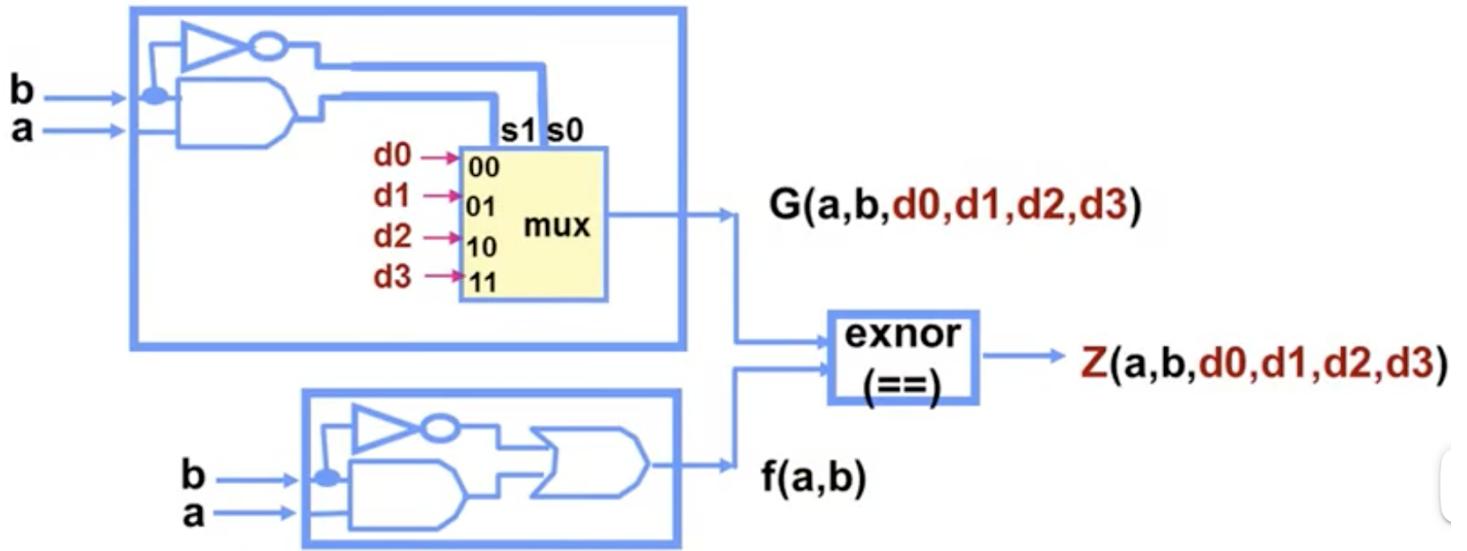


如下图，首先将错误的gate用一个 4:1 来代替，引入了四个新的变量，我们可以将这四个新的变量取不同的值来模拟任意的逻辑门。



接下来构建一个新的逻辑函数 $Z(ab, d1, d2, d3, d4)$ 使得它仅在 $f == G$ 时恒为1，如下图所示，只需要将

G 与 f 的正确实现进行同或即可得到相应的函数 Z



现在想得到的是一组mux的输入 d_0, d_1, d_2, d_3 使得对于所有的 a, b 都有 $Z = 1$, 通过universal quantification能够得到该问题的答案, 即通过

$$\forall_{ab} Z = 1$$

求出对应的 d_0, d_1, d_2, d_3 的取值。

上图中 mux 的输出

$$G(a, b, d_0, d_1, d_2, d_3) = d_0 a' b + d_1 b' + d_2 a b$$

$$Z = G \overline{\oplus} f$$

$$Z_{ab} = G_{ab} \overline{\oplus} f_{ab}$$

$$Z_{ab'} = G_{ab'} \overline{\oplus} f_{ab'}$$

$$Z_{a'b} = G_{a'b} \overline{\oplus} f_{a'b}$$

$$Z_{a'b'} = G_{a'b'} \overline{\oplus} f_{a'b'}$$

$$\forall_{ab} Z = Z_{ab} Z_{a'b} Z_{ab'} Z_{a'b'} = d'_0 d_1 d_2 = 1 \Rightarrow$$

$$d_0 = 0, d_1 = 1, d_2 = 1, d_3 = x$$

如果 d_3 取 1, 该 mux 实现的是一个 or gate, 如果 d_3 取 0, 该 mux 实现的是一个 exor gate, 因此使用 or 或者 exor 均可实现对该网络的修复。

Recursive tautology

Tautology

一个布尔函数 f 如果恒为真则可以将其称为tautology

Positional Cube Notation(PCN)

用PCN来表示一个SOP形式的布尔函数，一个cube表示SOP中的单个乘积项，cube用2bit的slot来表示一个单变量

- 01表示乘积项中包含该变量
- 10表示乘积项中包含该变量的反变量
- 11表示乘积项中不包含该变量的原变量或者反变量

如 abc, abc', bc 分别可以表示为：

$abc : [01 01 01]$

$abc' : [01 01 10]$

$bc : [11 01 01]$

整个SOP用一个cube list表示，如 $f(a, b, c) = a + bc + ab$ 可以表示为：

$[01 11 11], [11 01 01], [01 0111]$

Cofactor与tautology

f 是 tautology 当且仅当 f_x 和 $f_{x'}$ 都是 tautology

证明：

如果 $f() == 1$, 显然 f_x 和 $f_{x'}$ 均为1

如果 $f_x = 1, f_{x'} = 1$, 根据香农展开式有 $f() = xf_x + x'f_{x'} = x + x' = 1$

判断tautology的方法

根据以上描述的tautology与cofactor的关系可以采用递归的方式来判断f是否是tautology, 如果能判断出f是tautology则直接能得到结果, 否则分别求出f关于某个变量的positive 和 negative cofactor, 再分别判断两个cofactor是否是tautology, 这个方法被称作Unate recursive paradigm(URP), 进行URP需要确定如下三点：

- 求Cofactor的方法
- 终止规则：什么时候可以断定 $f == 1$ 或者 $f! == 1$ 并终止递归
- 变量选取规则：选取哪个变量来求cofactor

Unate recursive paradigm

求取Cofactor

布尔函数用cube list表示，求取布尔函数关于 x 的cofactor需要对cube list中的每个cube根据 x 在该cube中对应slot的取值进行如下操作

Positive cofactor:

- x 对应的slot是 10: 将该cube从cube list中移除，因为它包含 x'
- x 对应的slot是 01: 将该cube中的该slot改为11，因为 x 取1之后该乘积项中就不包含 x 了
- x 对应的slot是 11: 不做任何操作，因为此乘积项中不包含 x

Negative cofactor:

- x 对应的slot是 01: 将该cube从cube list中移除，因为它包含 x
- x 对应的slot是 10: 将该cube中的该slot改为11，因为 x 取0之后该乘积项中就不包含 x 了
- x 对应的slot是 11: 不做任何操作，因为此乘积项中不包含 x

终止规则

- Unate function: SOP形式的布尔函数中所有变量都以同一个极性出现

$ab + ac'd + c'de'$: 是unate

$xy + x'y + xyz' + z$: 不是unate但是它关于变量 y 是unate

重要结论：一个cube list如果是unate，当且仅当它包含一个所有slot都是11的cube。

当得到一个 unate cube list 时可以采用如下终止规则

- 终止规则1: 该cube list包含所有slot都是11的cube，此时该cube list对应的布尔函数是tautology
- 终止规则2: 该cube list不包含所有slot都是11的cube，此时该cube list对应的布尔函数不是tautology

也可以增加其它的可能的终止规则，比如cube list中存在不同极性的单变量cube

变量选取规则

选取most not-unate 变量

- 选取规则1: 选取被最多乘积项依赖的binate变量
- 选取规则2: 如果被最多乘积项依赖的binate变量有多个时，选取正变量和反变量的差值绝对值小的变量

如从如下的cube list中选取一个变量

x	y	z	w
01	01	01	01
10	11	01	01
10	11	11	10
01	01	11	01

根据规则一可以选出 x 和 w 他们都是unate且被四个乘积项依赖

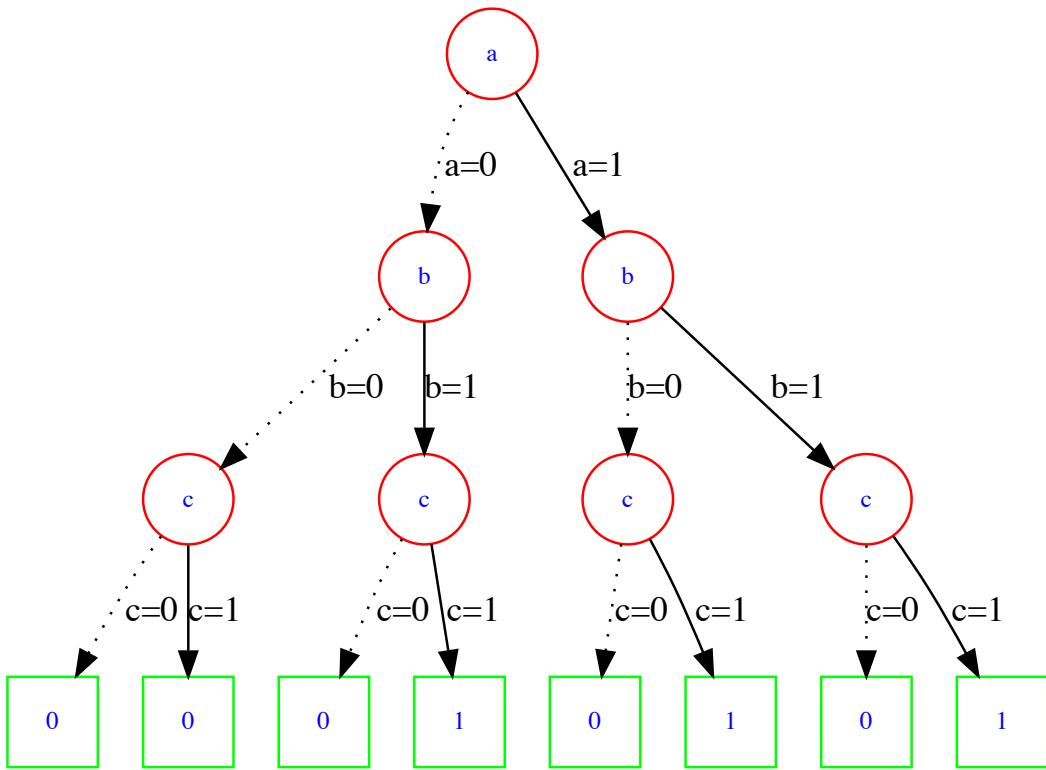
然后根据规则二, x 中正变量和反变量数目的差值绝对值是0, w 中正变量和反变量数目的差值绝对值是2, 因此 x 是最终的候选。

伪代码

```
IsTautology(f represent by cube list)
    if (f is unate) {
        apply unate tautology terminationn rules
        if (f is Constant 1) {
            return 1
        } else {
            return 0
        }
    } else {
        appply other termination rules
        if (f is Constant 1) {
            return 1
        } else {
            return 0
        }
        apply variable selection rules get variable x
        return IsTautology(fx) && IsTautology(fx')
    }
}
```

BDD

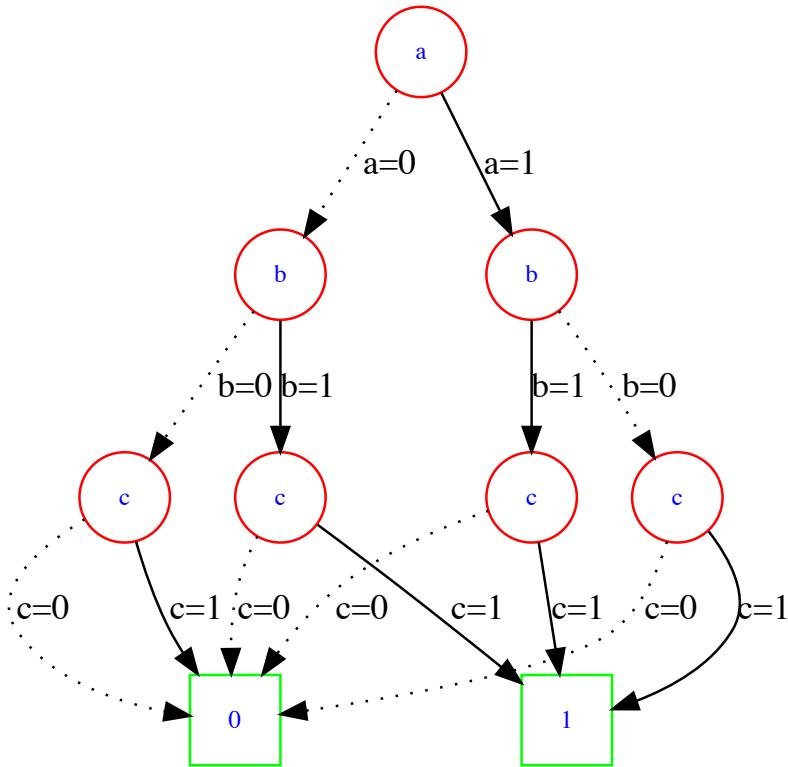
用顶点表示变量, 出边表示对该变量的一个决策(取0或者取1), 叶子节点表示布尔函数的取值, 如下图表示一个三变量布尔函数的BDD



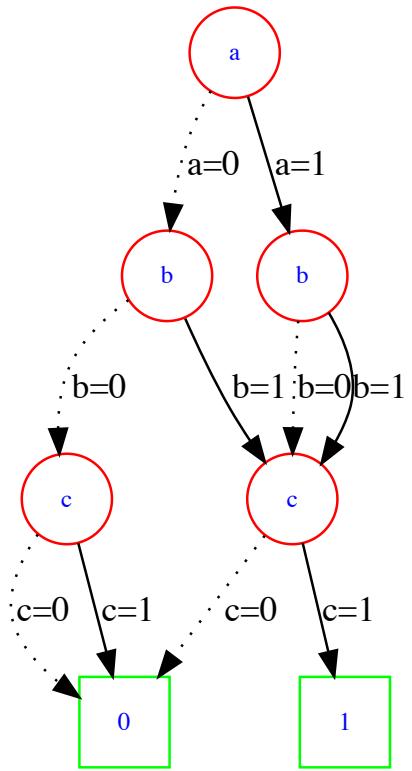
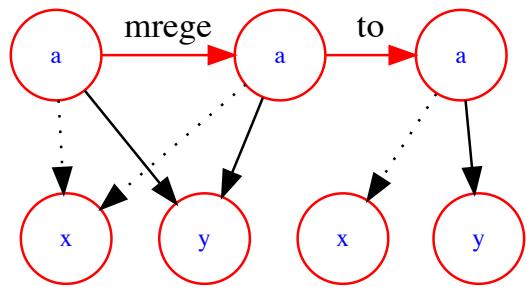
从上图中可以看出，BDD的规模与真值表的规模是一致的，随着变量的增加，其规模呈现出爆炸式增长，可以采取一些策略对BDD进行简化

BDD简化(ROBDD)

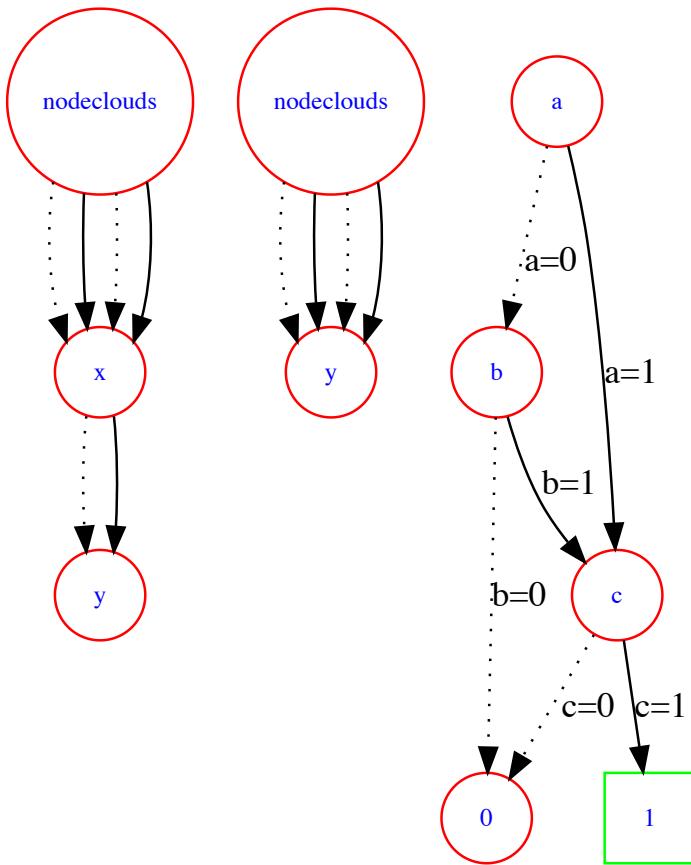
- 合并等价的叶子节点



- 合并同构的节点



- 消除冗余节点



BDD共享

BDD里的任意一个节点都可以表示一个布尔函数，因此如果两个布尔函数中有相同的部分，无需为该部分创建多次，只需要创建一次该结构并在多个BDD中共享。

BDD构建

从PI遍历到PO，每一个gate都被当作一个BDD operation，保证每个operation产生的新的BDD都会reduced, ordered, shared。

BDD的用途举例

判断布尔函数是否等价

只需要判断两个BDD的指针地址是否相同

寻找使得两个布尔函数取不同值的输入

构建出两个布尔函数相异或之后的BDD，并找出该BDD从根节点到为1的叶子节点的路径

tautology checking

一个布尔函数如果是tautology，它的BDD是单个1节点

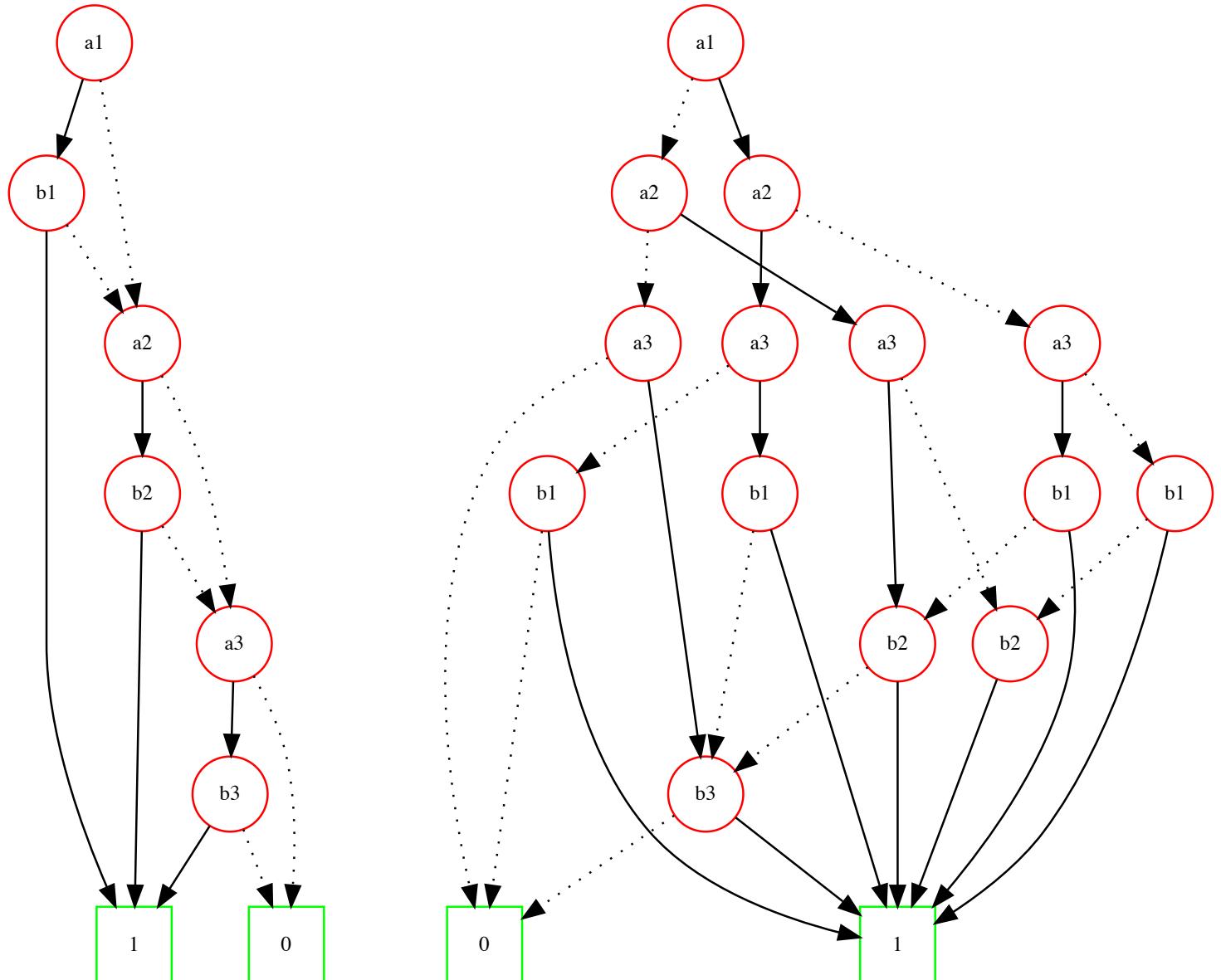
SAT

判断一个布尔函数是否可满足，从该布尔函数的根节点到叶子节点的路径就是一个解，若找不到说明不可满足。

BDD变量排序(经验方法)

- 相关的输入应该排在一起
- 对函数取值可以起决定性作用的变量应该排在一起且靠近BDD的顶部

如 $a_1b_1 + a_2b_2 + a_3b_3$ 在取两种不同的变量顺序时的BDD分别为



SAT

- 给出一个布尔函数 $F(x_1, \dots, x_n)$ 的适当表示
- 寻找一组输入使得 $F = 1$
- 如果没有可以满足的输入则给出证明并返回信息

CNF

标准的POS形式

$$F = (a + c)(b + c)(a' + b' + c')$$

上式中每一个求和项如 $a + c$ 被称作一个 clause, a, c 等被称作 positive literal, a', b' 被称作 negative literal。

Clause的状态

当对一些输入变量赋一定的值的时, CNF中的每个 clause 可以处于以下三种状态

- Satisfied (该 clause 在该赋值下为 1)
- Conflicting (该 clause 在该赋值下为 0)
- unresolved (该 clause 在该赋值下取值不确定)

例：给定CNF

$$F = (a + b')(a' + b + c')(a + c + d)(a' + b' + c')$$

令 $a = 0, b = 1$, CNF 中的各个 clause 的状态分别为 $a + b'$ conflicting, $a' + b + c'$ satisfied, $a + c + d$ unresolved, $a' + b' + c'$ satisfied。

Recursively方法DPLL

- Decision:
 - 选择一个变量并赋值; 简化CNF
 - 如果可以确定是否SAT, 则停止
- Deduction
 - 基于已有的赋值与 clause 的结构对CNF进行迭代的简化
 - 当不能简化时, 如果可以确定是否SAT则停止, 否则回溯到Decide

Boolean constraint propagation

BCP 主要用来进行上述的 Deduction 过程, 其中最著名的策略是使用 Unit clause rule

- 一个clause如果只有一个literal没有被赋值则称为**unit clause**

unit clause只有一种方式能SAT，即对该clause中未赋值赋值的literal赋1或者0（取决于该literal的极性），这一赋值操作被称作**implication**，这一implication有可能使得新的clause变为unit clause，因此又能对新的变量进行implication，如此循环的进行implication的过程被称作BCP。

例：

$$F = (a + c)(b + c)(a' + b' + c')$$

假设已经进行了部分赋值 $a = 1, b = 1$, clause $(a' + b' + c')$ 为unit clause，可以进行implication得到 $c = 0$

BCP终止规则

- SAT: 寻找到一组赋值使得所有的clause都为1
 - 返回结果
- Unresolved: 有一个或者更多的clause 的取值不确定
 - 选择一个新的未赋值的变量对其进行新的赋值并基于新的赋值进行BCP
- Conflict: 有一个或者更多的clause的取值为0
 - 回溯到最近的一次赋值操作(注意不是implication)
 - 若该变量的所有取值均已尝试过，则直接选取一个新的变量进行赋值
 - 若该变量还有其他取值未尝试过，则选择新的取值对该变量进行赋值并基于新的赋值进行BCP

电路等价性校验转化为SAT

假设有两个组合逻辑电路，他们的输入相同，输出数目相同，需要取check这两个电路的等价性，只需要将这两个电路对应的输出异或并将所有异或的结果相或得到一个新的多输入单输出的电路，只需要check这个电路是否SAT就能判断这两个电路是否等价。

组合逻辑电路转化为CNF

Gate consistency function

假设给定一个与非门其输入与输出的关系为 $d = (ab)'$, 它的gate consistency function为

$$\phi_d = [d == (ab)'] = d\overline{\oplus}(ab)' = (a + d)(b + d)(a' + b' + d')$$

gate consistency function为1表征的意义是gate的输出与其由输入表达的逻辑应该一致。

各种gate的gate consistency function

$$\begin{aligned}
z = x &\rightarrow (x' + z)(x + z') \\
z = x' &\rightarrow (x + z)(x' + z') \\
z = NOR(x_1, \dots, x_n) &\rightarrow (\sum_{i=1}^n x_i + z) \prod_{i=1}^n (x'_i + z') \\
z = OR(x_1, \dots, x_n) &\rightarrow (\sum_{i=1}^n x_i + z) \prod_{i=1}^n (x'_i + z) \\
z = NAND(x_1, \dots, x_n) &\rightarrow (\sum_{i=1}^n x'_i + z') \prod_{i=1}^n (x_i + z) \\
z = AND(x_1, \dots, x_n) &\rightarrow (\sum_{i=1}^n x'_i + z) \prod_{i=1}^n (x_i + z')
\end{aligned}$$

转化方法

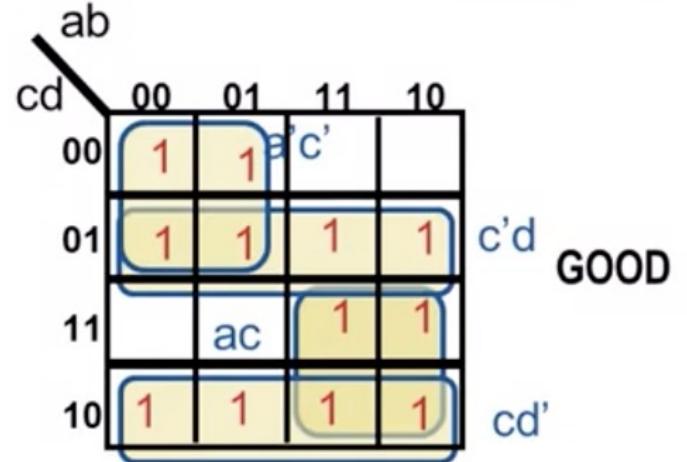
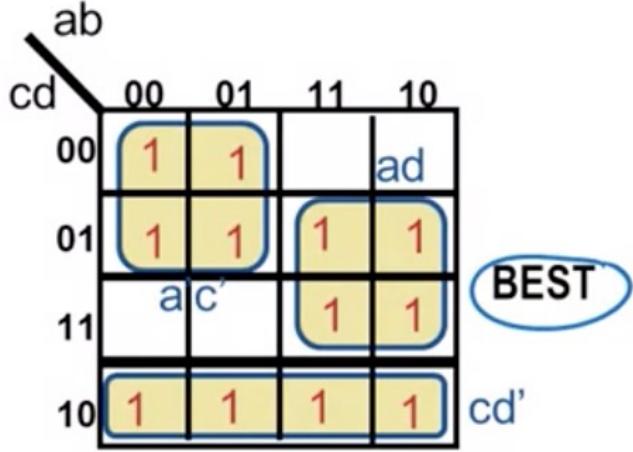
将该组合逻辑电路(单输出)的输出变量与该电路中所有gate的gate consistency function相与即可得到该组合逻辑电路的CNF表示。

Two level logic

Two level logic指的是使用SOP形式的电路来表示一个布尔逻辑函数，它只需要两级电路即可以表达任意的布尔逻辑，第一级用若干个与门，第二级使用一个或门将所有第一级所有与门的输出相或。

Two level minimization

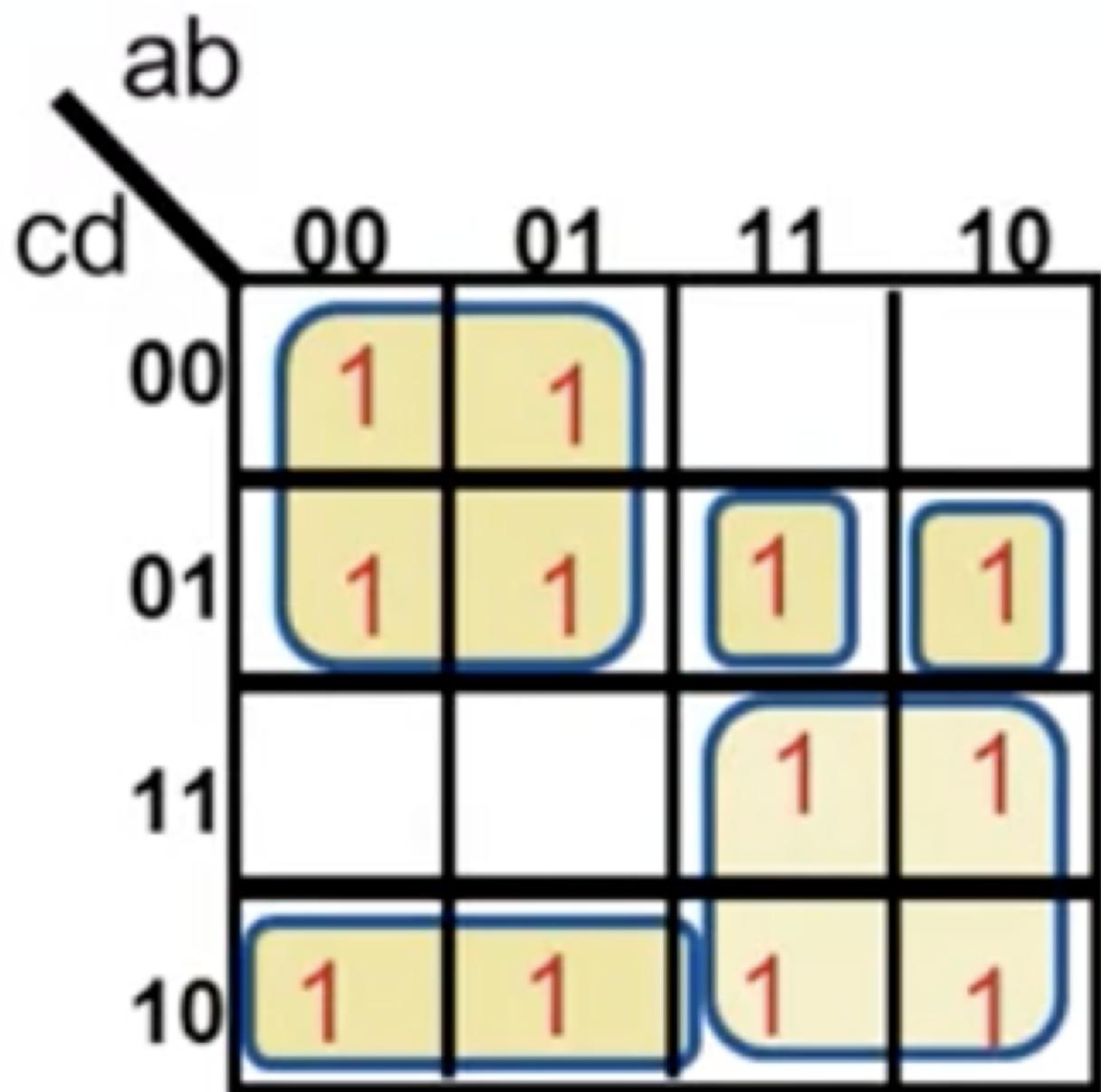
寻找一个two level logic的最优表示形式，使得所使用的与门的数目最少，但是要寻找一个最优解太难实现，因此退而求其次尽可能的求到一个比较优的解。



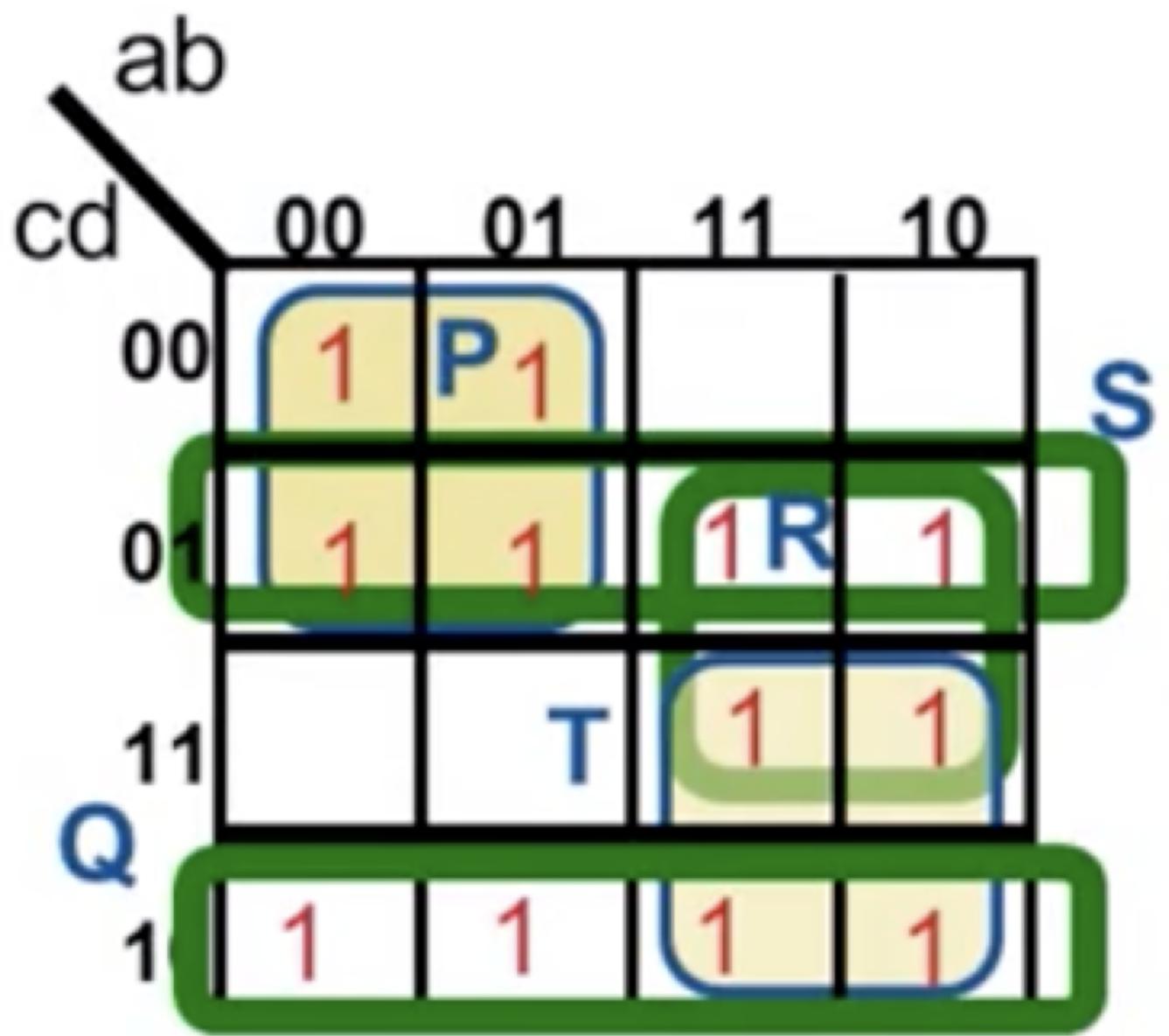
如上图是一个最优解和一个相对较优的解的卡诺图表示，他们都是使每一个product项“尽可能的大”，这些“尽可能的大”的product项被称作**Primes**，最优解一定是Primes的覆盖 (result from 1950s)，可以看出这两个解都是irredundant，即不能通过移除一个prime而获得更优的解。

Reduce-Expand-Irredundant optimization

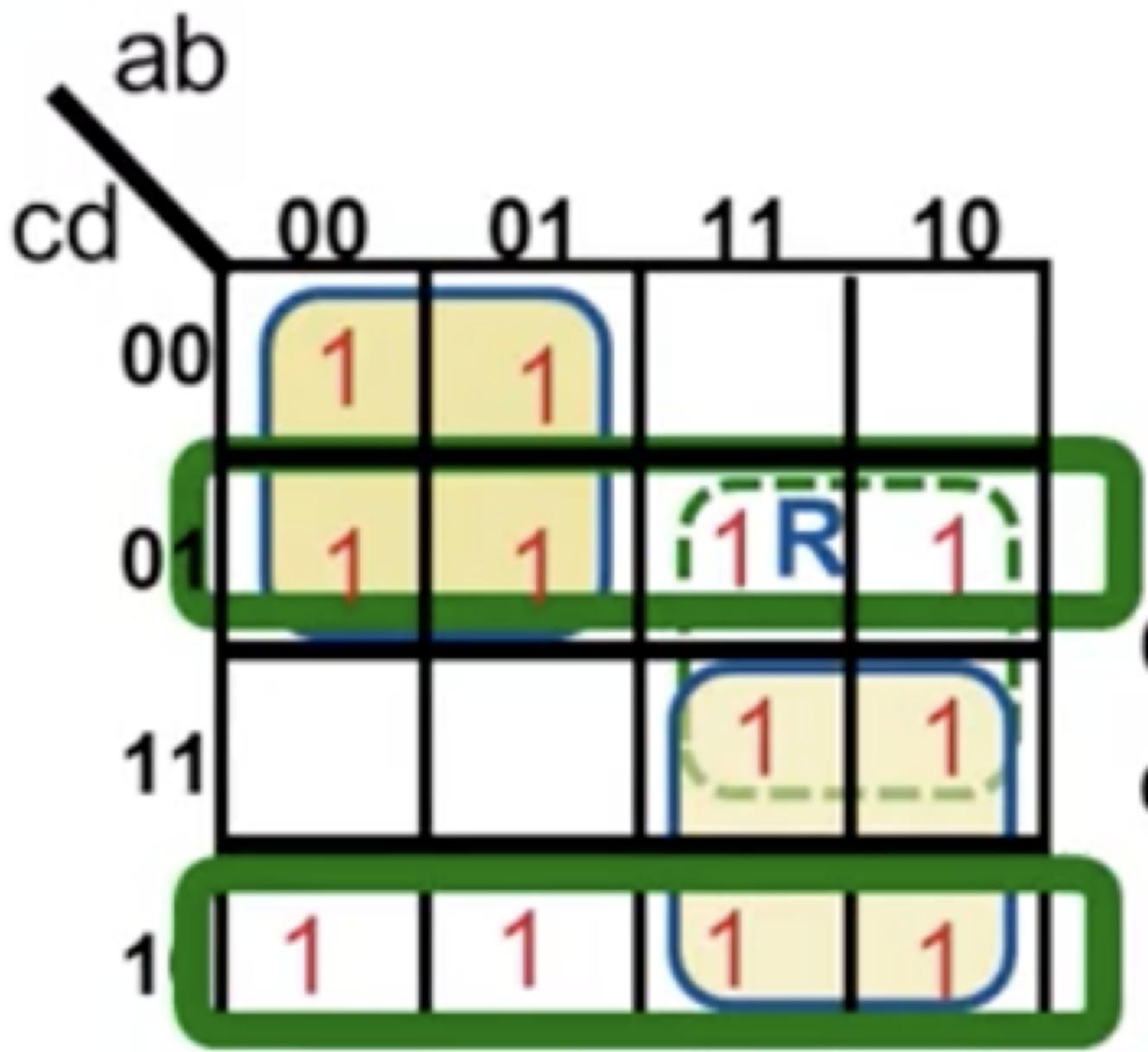
假设以下面的卡诺图作为开始



- Expand: 扩充每个cube使其尽可能的大,即让它变为prime



- Irredundant: 移除冗余cube，该cube中的所有的1均被其他的cube所覆盖则称该cube是冗余的，经过irredundant之后所有的cube仍然是prime

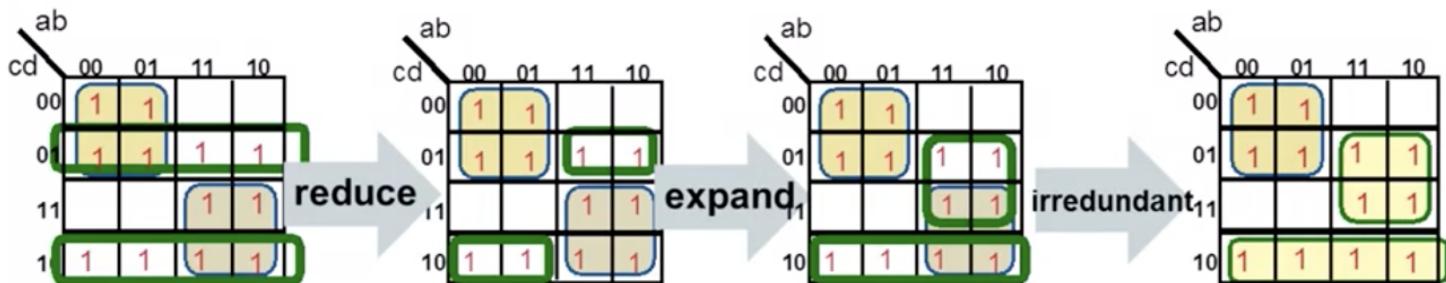
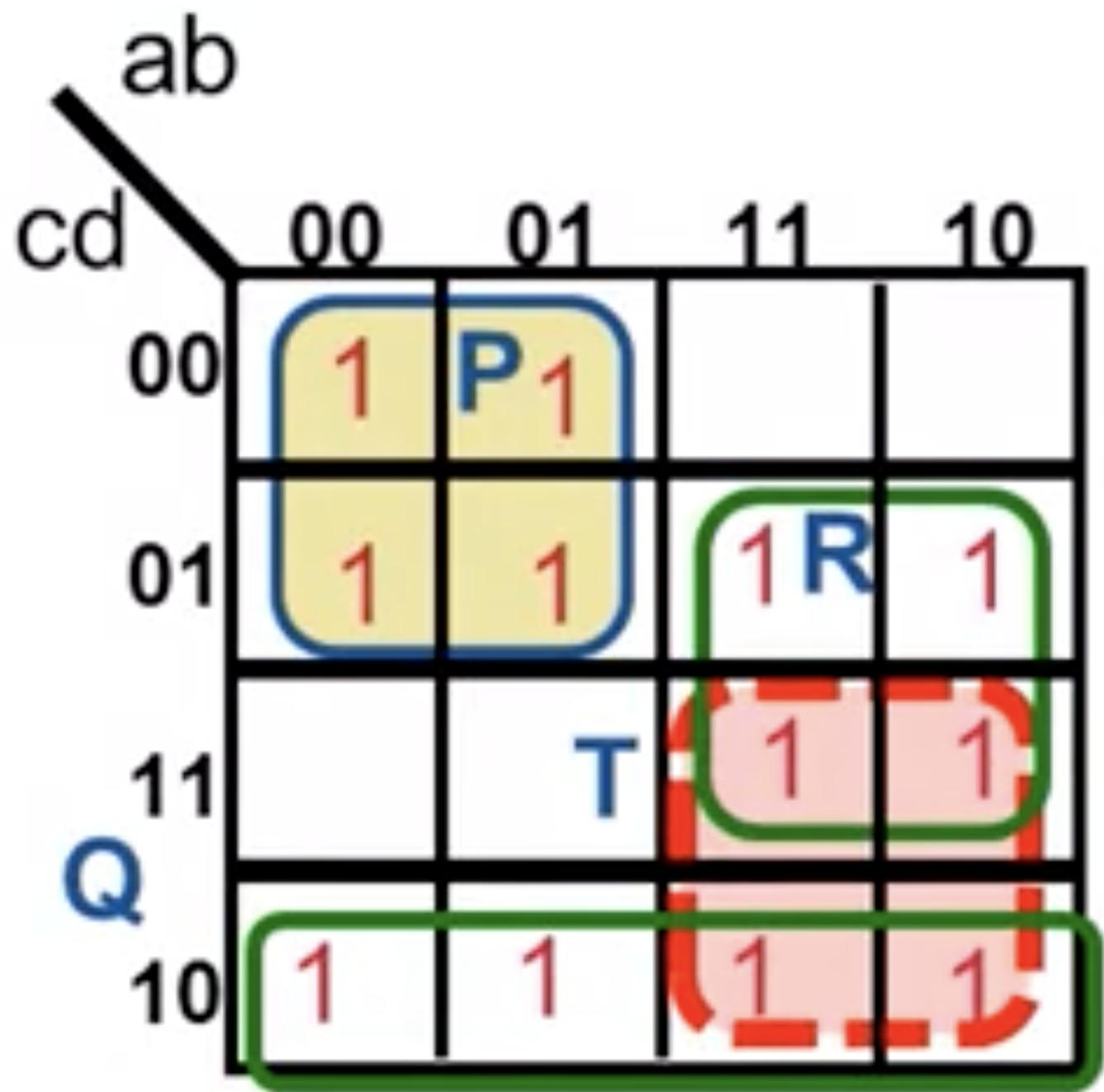


- Reduce: reduce步骤尽可能的缩小每个cube但是仍然保证所有的1都被覆盖， 经过reduce之后cube可能不是prime

Diagram illustrating a state transition table for a state-space search algorithm. The columns represent states s (00, 01, 11, 10) and the rows represent actions Q (00, 01, 11, 10). The table shows the next state s' and a value P or T associated with each transition.

	ab	cd	00	01	11	10	s
00	1	P 1					
01	1	1		1	1		
11					T 1		
10	1	1		1	1		

执行完这一个循环之后如果再执行expand irredundant, 可能能够得到一个更优的解。



Expand step detail

expand的意思是从cube的移除一些变量，即将PCN中一些变量对应的slots变为11

Build off set

给定一个函数 F , 构建该函数中0的cube cover, 这些cubes被称作off set, 在expand过程中不能触碰到这些cube, 可以使用URP complement来获取一个函数的Off set

Build the blocking matrix

- 每一行表示cube中待expand的变量
- 每一列表示off set中的一个cube
- 如果行中的变量的极性与列中的极性不一致则该位置为1, 否则为0

	$x'z$	wxz'	$wy'z'$
w'		1	1
x	1		
y			1
z'	1		

blocking matrix 中的1表示该1所在的行所对应的变量只要存在就能够不触碰该1所在的列对应的cube, 寻找该blocking matrix的一个覆盖并将在该覆盖下所选择的行所对应的变量保留下来即可得到该cube的一个expansion。

矩阵覆盖

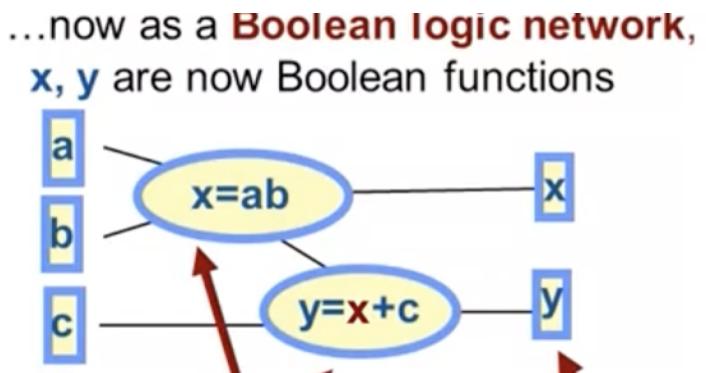
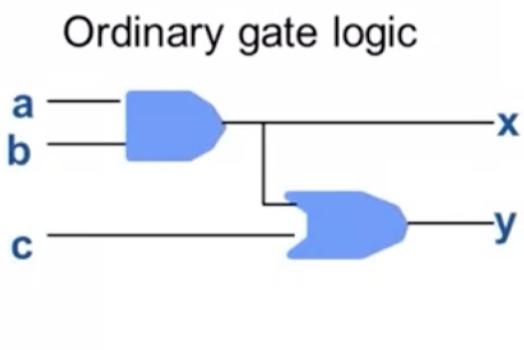
矩阵中只有0和1, 从中选取最少的行使得每一列至少有一个1被某一行所覆盖。

Multi-level logic

Two level logic的delay通常很低，但是他的面积相对较大，multi-lelve logic可以在面积和delay上进行trade-off。

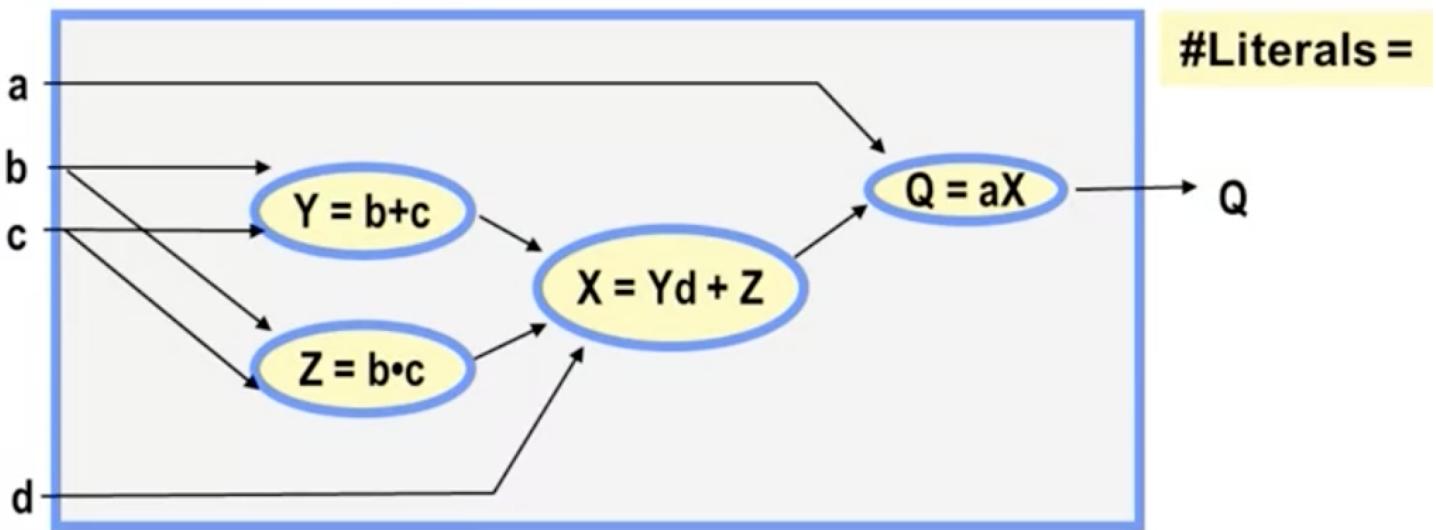
Boolean logic network model

boolean logic network类似gate logic电路图，不过这个图中的每个节点不再局限于与或非等逻辑门而是一个SOP形式的2-level的布尔函数。



Optimize on boolean logic network

在一个boolean logic network上最简单的优化目标是Total literal count, total literal count指的是网络中所有节点所表示的布尔函数等号右边变量的总数目，如下图的，它的total literal count是9。



Operations on boolean logic Network

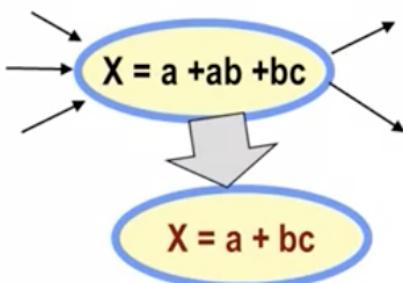
- Simplify: 不改变节点的数目，只是对节点的内部的2-level logic进行简化
- Remove: 把一些相对来说太小的节点推进该节点的fanout节点

- Add: 也叫factoring, 把一些大的节点给分成更小的节点

Simplify操作其实就是2-level synthesis也就是ESPRESSO， Remove也比较简单只涉及graph edit操作， 重点是Add操作。

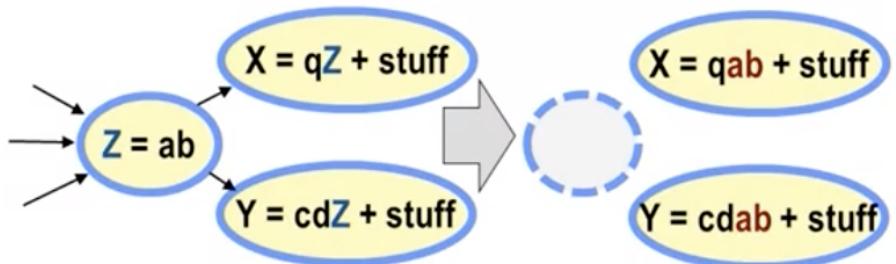
- Simplifying a node

- Just run ESPRESSO on 2-level form **inside** the node, to reduce # literals
 - As structural changes happen across network, “insides” of nodes may present opportunity to simplify



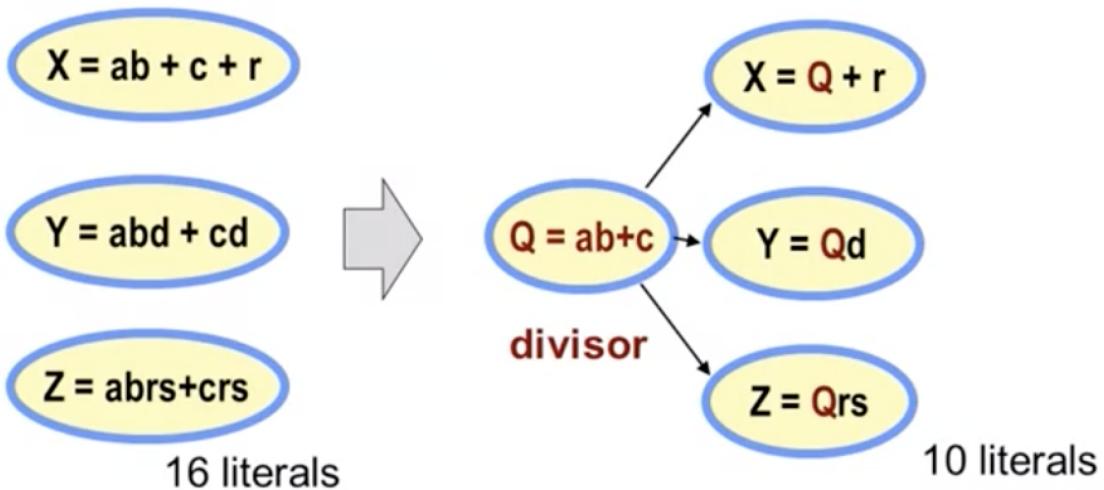
- Removing a node

- Typical case is you have a “small” factor which doesn’t seem to be worth making it a separate node
 - “Push” it back into its fanouts, make those nodes bigger, and hope you can use 2-level simplification on them



Adding new node(s): this is Factoring, this is new, and hard

- Look at existing nodes, identify **common divisors**, extract them, connect as fanins
 - Tradeoff: **more** delay (another level of logic), but **fewer literals** (less gate area)



Algebraic Division

给定一个布尔函数 F 的SOP形式，如果它可以表示成

$$F = D \cdot Q + R$$

那么可以将 D 称作 divisor(除数), Q 称作 Quotient(商), R 称作 remainder(余数), 如果余数为0, 则可以将 divisor 称作 F 的一个 factor, 注意在这里面原变量和反变量需要被当成完全无关的变量。

$$\begin{aligned} F &= ac + ad + bc + bd + e \\ &= (a+b) \cdot (c+d) + e \end{aligned}$$

$(a+b) = D = \text{divisor}$
 $(c+d) = Q = \text{quotient}$
 $e = R = \text{remainder}$

伪代码

```
AlgebraicDivision( F, D ) { // divide D into F
```

```
    for ( each cube d in divisor D ) {
        let C = { cubes in F that contain this product term "d" };
        if ( C is empty ) {
            return ( quotient = 0, remainder = F );
        }
        let C = cross out literals of cube "d" in each cube of C;
        if ( d is the first cube we have looked at in divisor D )
            let Q = C;
        else
            Q = Q ∩ C;
    }
    R = F - ( Q • D );
    return ( quotient = Q, remainder = R )
}
```

Example:
Cube $xyzw$ contains product term "yz"

Example:

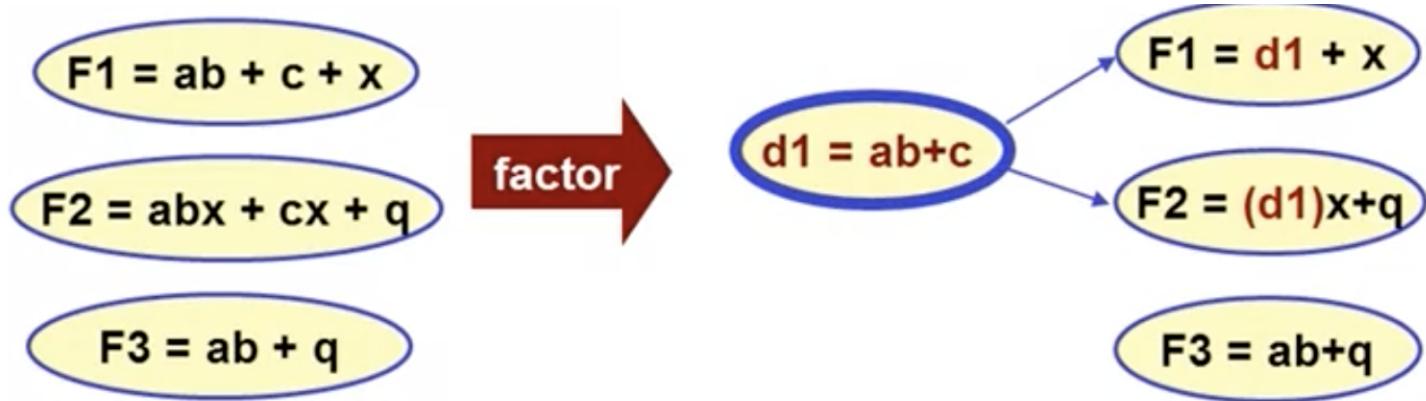
Suppose $Q = xyz + yzw + pqyz$
and $C = "xyz"$. Then $Q \cap C$
is just the cubes that are in both Q and C
in this case: xzv

Example:
Suppose $C = xyz + yzw + pqyz$
and $d = "yz"$. Then crossing out all the "yz" parts yields
 $x + w + pq$

需要注意 F 中的必须没有 redundant cube, 即没有一个 cube 被其它的 cube 完全覆盖, 如 $F = a + ab + bc$ 中的 ab 就是 redundant cube。

Factoring

有了上面所述的Algebraic division操作，factoring变成如下问题：
给定若干个布尔函数，寻找出若干common divisor。



divisor可以分为两种

- 仅包含一个cube的divisor(e.g $d = ab$)
- 包含多个cube的divisor(e.g $d = ab + cd + e$)

Kernels and co-kernels

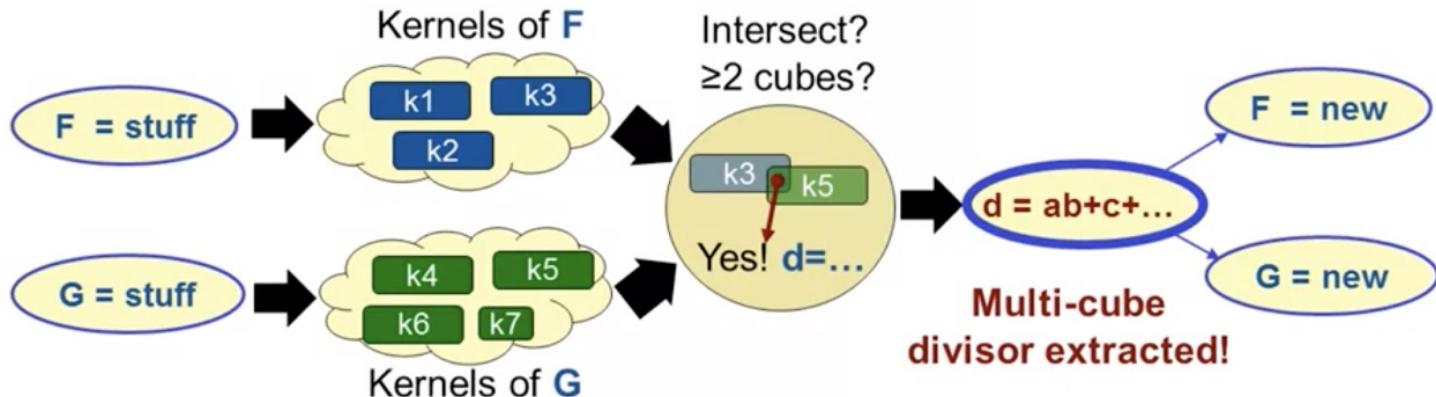
一个布尔表达式 F 的Kernel指的是用 F 除以一个single cube c 得到的商 k , 该商必须是cube-free的, 这个single cube c 被称作是kernel k 的co-kernel.

$$F = c \cdot k + R$$

- single cube指的是单个乘积项, 如 abc 或 xy .
- cube free指的是不能再从中分解出一个single cube而没有余数, 如 $abc + abd$ 不是cube free的, 它可以分解出single cube ab , 而 $ab + c$ 是cube free的。

重要结论 F 和 G 存在common multiple-cube divisor的充分必要条件是：

存在 $k1 \in K(F), k2 \in K(G)$ 使得 $d = k1 \cap k2$, 并且 d 中至少有2个cube, 式子中的 $k1 \cap k2$ 指的是 $k1$ 和 $k2$ 中的comon cube。



Find Kernels

假设 k_1 是 F 的一个kernel, 那么我们有

$$F = \text{cube1} \cdot k_1 + R_1$$

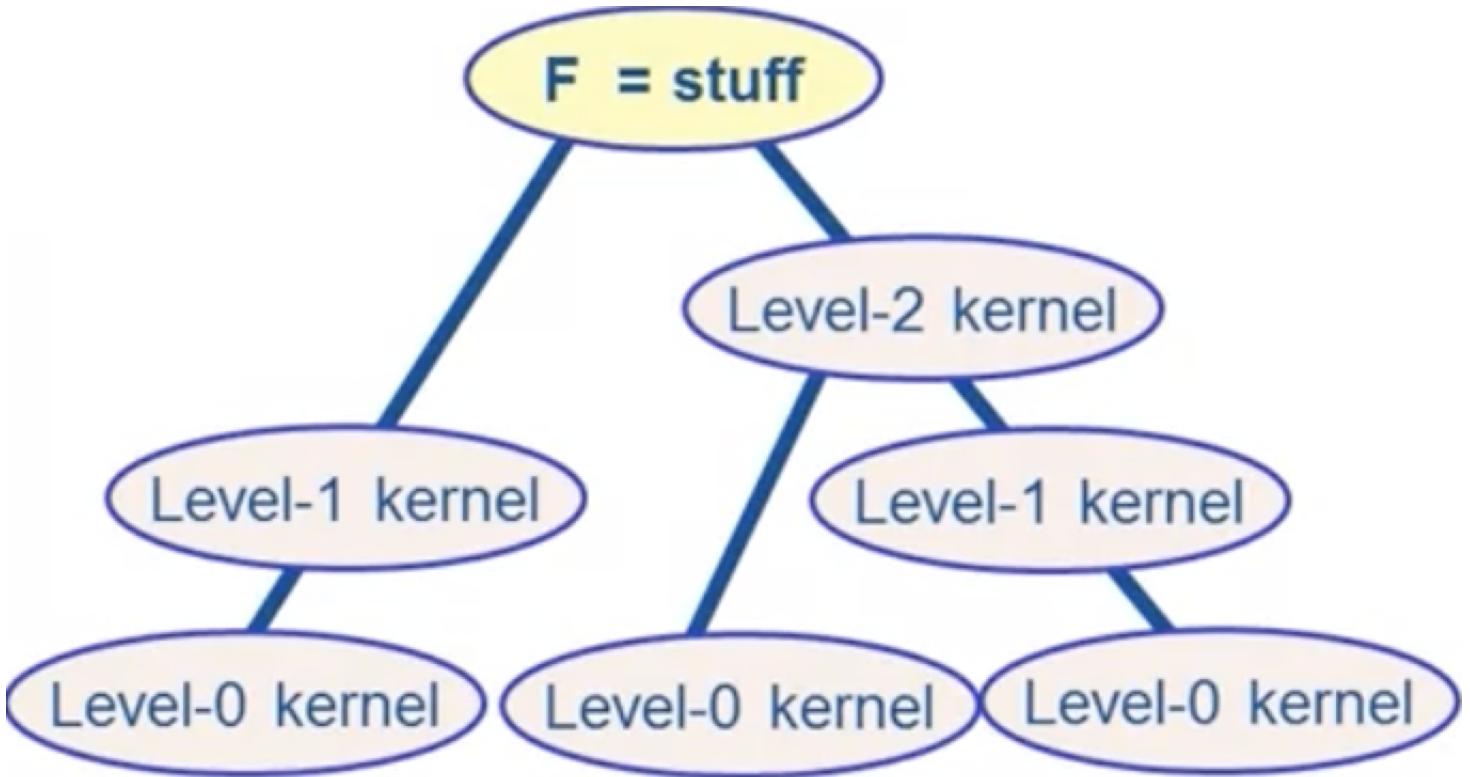
假设 k_2 是 k_1 的一个kernel, 则

$$k_1 = \text{cube2} \cdot k_2 + R_2$$

结合上面两个式子有

$$\begin{aligned} F &= \text{cube1} \cdot (\text{cube2} \cdot k_2 + R_2) + R_1 \\ &= \text{cube1} \cdot \text{cube2} \cdot k_2 + R_1 + \text{cube1} \cdot R_2 \end{aligned}$$

因此可以得到 k_2 也是 F 的一个kernel, 对应的co-kernel是 $\text{cube1} \cdot \text{cube2}$, 从上面的结果可以知道, kernel 实际上是一种hierarchical的结构



- 一个level-0的kernel除了自身之外没有其他的kernel
- 一个level-n的kernel至少包含一个level-(n-1)的kernel而除了自身之外没有其他的level-n的kernel

另外一个重要结论是一个布尔表达式的co-kernel跟其PCN中两个以上cube的交集相关, 注意这里的交集指的是两个cube中的common literal, 如 $ace + bce + de + g$ 的潜在co-kernel有 $ace \cap bce = ce$, $ace \cap bce \cap de = e$, 因此咱们可以用 F 除以潜在co-kernel来启动算法然后递归的获取kernel, 需要注意的是, F 必须是cube free的, 如果不是则需要将 F 除以最大的 common cube将 F 变为cube free的。

```

FindKernels( cube-free SOP expression F {
    K = empty;
    for ( each variable x in F ){
        if ( there are at least 2 cubes in F that have variable x ) {
            let S = { cubes in F that have variable x in them };
            let co = cube that results from intersection of all cubes in S,
                this will be the product of just those literals
                that appear in each of these cubes in S;
            K = K ∪ FindKernels( F / co );
        }
    }
    K = K ∪ F ;
    return( K )
}

```

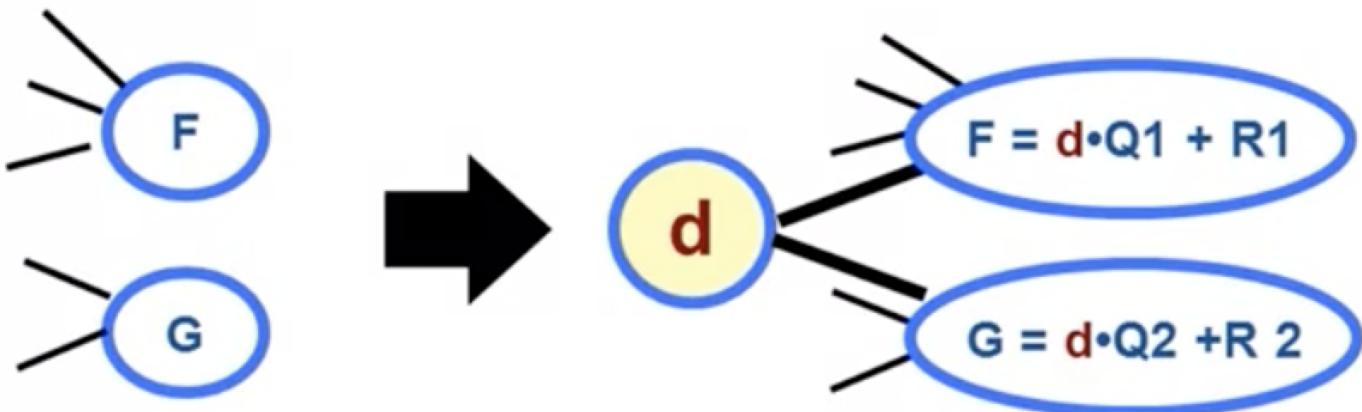
Cube-free **F** is always its own kernel, with trivial co-kernel = 1

F/co = a kernel, returned at end.
 This is algebraic division, but simpler since it always just divides by **exactly 1 cube, a simple product term**

© 2013, R.A. Rutenbar

通过这个算法，可以得到布尔函数所有的kernel和co-kernel，因此

- 如果想从多个表达式中寻找single-cube common divisor，可以考虑它们的co-kernel
- 如果想从多个表达式中寻找multiple-cube common divisor，可以考虑它们的kernel



Summary

- **Boolean network model**
 - Like a gate network, but each node in network is an SOP form
 - Supports many operations to add, reduce, simplify nodes in network
- **Algebraic model & algebraic division**
 - Simplifies Boolean functions to behave like polynomials of real numbers
 - Divides one Boolean function by another: $F = (\text{divisor } D) \cdot (\text{quotient } Q) + \text{remainder } R$
- **Kernels / Co-kernels of a function F**
 - **Kernel** = cube-free quotient obtained by dividing by a single cube (**co-kernel**)
 - Intersections of F, G kernels \rightarrow all multiple-cube common divisors (Brayton-McMullen)

接下来需要解决如何选择一个最好common divisor来做factoring操作

Divisor extraction: single cube case

- 构建一个仅包含0与1的矩阵，称作cube-literal matrix
- 在矩阵中启发式的寻找prime rectangles
- 每一个prime是一个好的common single cube divisor

Cube-literal matrix

cube-literal matrix 的每一行表示一个single cube，每一列表示一个literal，矩阵中某个位置所在列对应的literal如果在行对应的cube中出现了，那么该位置的值就为1，否则为一个“.”。

假设有三个布尔表达式

- $P = abc + abd + eg$
- $Q = abfg$
- $R = bd + ef$

他们对应的cube-literal matrix为

$P = abc + abd + eg$
 $Q = abfg$
 $R = bd + ef$

Example from: Richard Rudell, Logic Synthesis for VLSI Design, PhD Thesis, Dept of EECS, University of California at Berkeley, 1989.

	a	b	c	d	e	f	g
1	1	2	3	4	5	6	7
abc	1						
abd	2	1	1	.	1	.	.
eg	3	1	.
abfg	4	1	1	.	.	1	1
bd	5	.	1	.	1	.	.
ef	6	1	1

图中第一行的值未填写，可以自己补全。

rectangle

矩阵中的一个rectangle指的是：一个行和列的集合，集合中每一行和每一列的在矩阵中的交叉点必须为1，注意此处没有要求行或者列连续，任意的行和列的集合均可。

Prime rectangle

prime rectangle指的是该rectangle不能再添加其他的行或者列，下图是一个prime rectangle的例子

Prime rectangle example

	a	b	c	d	e	f	g
1	1	1	1	1	.	.	.
2	1	1	.	.	1	.	.
3	1	.
4	1	1	.	.	.	1	1
5	.	1	.	.	1	.	.
6	1	1	.

Common single-cube divisor

Prime rectangle中的列给出的是该single-cube divisor的literal，行给出的是表达式中哪些cube中包含该divisor。

$$\begin{aligned} P &= abc + abd + eg \\ Q &= abfg \\ R &= bd + ef \end{aligned}$$

	a	b	c	d	e	f	g
1	1	1	1
2	1	1	.	.	1	.	.
3	1	.	1
4	1	1	.	.	.	1	1
5	.	1	.	1	.	.	.
6	1	1	.

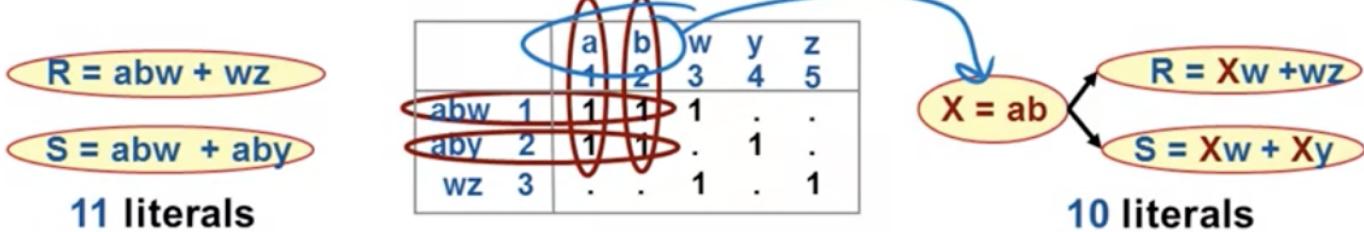
1 cube divisor
 $X = ab$

$$\begin{aligned} P &= Xc + Xd + eg \\ Q &= Xfg \\ R &= bd + ef \\ X &= ab \end{aligned}$$

经过这种common-divisor extraction之后可以减少literal的数目为

$$L = (C - 1) \cdot \sum_r w(r) - C$$

其中 C 为该prime rectangle中包含的列的数目， r 为其中某一行对应的single cube， $w(r)$ 为cube r 在该网络中出现的次数。



- Literals saved formula: Compute $L = (C-1) \times [\sum_{\text{rows } r} \text{Weight}(r)] - C$
 - $C = 2 \text{ columns}$
 - $\text{Weight}(abw) = 2$ (appears twice in network). $\text{Weight}(aby) = 1$ (appears once)
 - $L = (2-1)(2+1) - 2 = 1$ it works! We did, in fact, **save just 1 literal.**
 - Counts all places ab appeared (6 literals), the savings to replace each with X (3 literals), and new overhead of adding $X=ab$ term (+2 literals). $-6+3+2 = 1 \text{ saved.}$

Divisor extraction: multiple cube case

multiple-cube 的整个过程与single-cube case的类似，也是在一个矩阵中寻找prime rectangle。

假设有三个布尔表达式

- $P = af + bf + ag + cg + ade + bde + cde$
- $Q = qf + bf + ace + bce$
- $R = ade + cde$

第一步需要寻找这些布尔函数的co-kernel和kernel，因为multiple-cube factors是这些函数kernel中乘机项的交集，他们的kernel和co-kernel分别是

$$P = af + bf + ag + cg + ade + bde + cde$$

- co-kernel: a kernel: $de + f + g$
- co-kernel: b kernel: $de + f$
- co-kernel: de kernel: $a + b + c$
- co-kernel: f kernel: $a + b$
- co-kernel: c kernel: $de + g$
- co-kernel: g kernel: $a + c$
- co-kernel: 1 kernel: P trivial, ignore

$$Q = qf + bf + ace + bce$$

- co-kernel: a or b kernel $ce + f$
- co-kernel: f or ce kernel $a + b$
- co-kernel: 1 kernel: Q trivial, ignore

$$R = ade + cde$$

- co-kernel: de kernel: $a + c$
- 注意此处 R 本身不是自己的kernel, 因为 R 不是cube-free的

Co-kernel-cube matrix

Co-kernel-cube matrix 的每一行用一个pair<function, co-kernel>来表示, pair的第一个元素为一个布尔函数, 第二个元素为该布尔函数的一个co-kernel, 每一列表示所有这些布尔函数kernel中的一个cube, 我们的例子中的co-kernel-cube matrix 的行和列如下图

One row for each unique (Function, Co-kernel) pair in problem

One column for each unique cube among all kernels in problem

	a	b	c	ce	de	f	g
1	2	3	4	5	6	7	
P a 1	1	1	1
P b 2	1	1	.
P de 3	1	1	1
P f 4	1	1
P c 5	1	.	1
Q a 7	.	.	.	1	.	1	.
Q b 8	.	.	.	1	.	1	.
Q ce 9	1	1
Q f 10	1	1
R de 11	1	.	1

What goes in the individual (row, column) entries here in the matrix?

矩阵中元素所在列对应的cube如果在该元素所在行的布尔函数相对于该行中co-kernel的kernel中出现, 则该元素值为1, 否则为“.”

P = af+bf+ag+cg+ade+bde+cde

P co-kernel (a) kernel (de + f + g)

P co-kernel (g) kernel (a+c)

In effect, each row says "here is a function, if I divide it by this co-kernel cube, that is the result (kernel) I get..."

	a	b	c	ce	de	f	g
1	2	3	4	5	6	7	
P a 1	1	1	1
P b 2	1	1	.
P de 3	1	1	1
P f 4	1	1
P c 5	1	.	1
P g 6	1	.	1
Q a 7	.	.	.	1	.	1	.
Q b 8	.	.	.	1	.	1	.
Q ce 9	1	1
Q f 10	1	1
R de 11	1	.	1

构建好co-kernel-cube matrix之后，在其上定义的rectangle和prime rectangle与cube-literal matrix上的一致，将prime rectangle中的所有列对应的cube加在一起即可得到一个好的multiple-cube common divisor。

	a	b	c	ce	de	f	g
P a 1	1	2	3	4	5	6	7
P b 2	1	1	1
P de 3	1	1	1
P f 4	1	1	.	.	1	.	.
P c 5	1	.	1
P g 6	1	.	1
Q a 7	.	.	.	1	.	1	.
Q b 8	.	.	.	1	1	.	.
Q ce 9	1	1
Q f 10	1	1
R de 11	1	1

Interpretation

$\mathbf{P} = (\text{co-kernel } \mathbf{de}) \cdot (a + b)$
 $\mathbf{P} = (\text{co-kernel } \mathbf{f}) \cdot (a + b)$

$+ \text{stuff1} + \text{rem1}$
 $+ \text{stuff2} + \text{rem2}$

$\mathbf{Q} = (\text{co-kernel } \mathbf{ce}) \cdot (a + b)$
 $\mathbf{Q} = (\text{co-kernel } \mathbf{f}) \cdot (a + b)$

$+ \text{stuff3} + \text{rem3}$
 $+ \text{stuff4} + \text{rem4}$

OR together cubes of the **columns** of the prime rectangle – **this** is the divisor **(a+b)**!

通过这种common divisor提取之后能节省的literal的数目为

$$L = \sum_{r,c} V(r, c) - \sum_r w_R(r) - \sum_c w_C(c)$$

其中 r 为某一行, c 为某一列, $V(r, c)$ 为行 r 中的co-kernel与列 c 中cube相与之后得到的新的乘积项中的literal的数目, $w_R(r)$ 为1 + co-kernel中包含的literal数目, $w_C(c)$ 为列 c 中的cube所包含的literal的数目。

Find prime rectangle

寻找这些prime rectangle被称作rectangle covering问题，使用启发式的贪心算法能很好的解决这个问题

- 以能节省较多literal的单独一行开始
- 交替的增加更多的行和列
如 Rudell 的pingpong heuristic

1. 选择能节省最多literal的一行作为开始
2. 寻找另外在相同位置为1(或者更多)的行, 依次将其加入直到不能加入更多的行
3. 寻找另外在相同位置为1(或者更多)的列, 依次加入直到不能加入更多的列
4. 重复2-3直到不能添加任何行和列

- **Greedy heuristics work well for this rectangle covering problem**
 - Start with a single row rectangle with “good #literal savings”
 - Grow the rectangle alternatively by adding more rows, more columnss
- **Example: Rudell's Ping Pong heuristic (from his '89 Berkeley PhD)**
 - To grow a big prime rectangle, from a good 1-row rectangle
 1. Pick the **best single row** (the 1-row rectangle with best #literals saved)
 2. Look at **other rows with 1s** in same places (and more!) that you can add, one at a time, that each maximize #literals saved. Add them until can't find any more.
 3. Look at **other columns with 1s** in same places (and more!) that you can add, one at a time, that maximize # literals saved. Add them until you can't find any more.
 4. Goto 2.
 5. Quit when cannot grow rectangle any more in any direction.

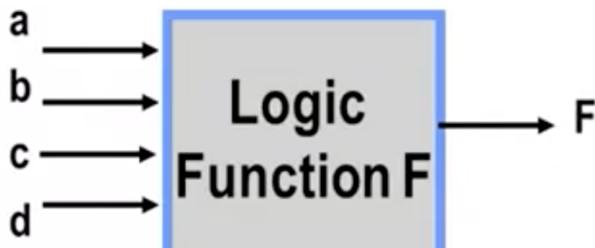
Summary

- Single cube extraction
 - 构建cube-literal matrix; prime rectangle是一个好的single cube divisor
 - simple bookkeeping可以估计出在布尔网络中能节省多少literal
- Multiple cube extraction
 - 构建co-kernal-cube FontMatrix;
 - 每个prime rectangle是一个好的multiple cube divisor
 - simple bookkeeping可以估计出在布尔网络中能节省多少literal
- 在原理上他们都是rectangle covering问题
 - 可以用启发式算法获取一个好的prime rectangle
 - 有办法从网络中提取超过一个divisor

Implicit don't cares

使用布尔逻辑网络来优化multi-level logic时会对每个node进行ESPRESSO风格的优化，这丢失了一部分外部的信息，可以从node周围的logic中提取出don't cares并辅助当前节点的优化，这些don't care是implicit的，需要方法将其提取出来，下面给出一个使用don't cares进行优化的例子(**don't care**: 输入不可能取到的pattern)。

- **Don't Care (DC)** = an input pattern that can **never** happen



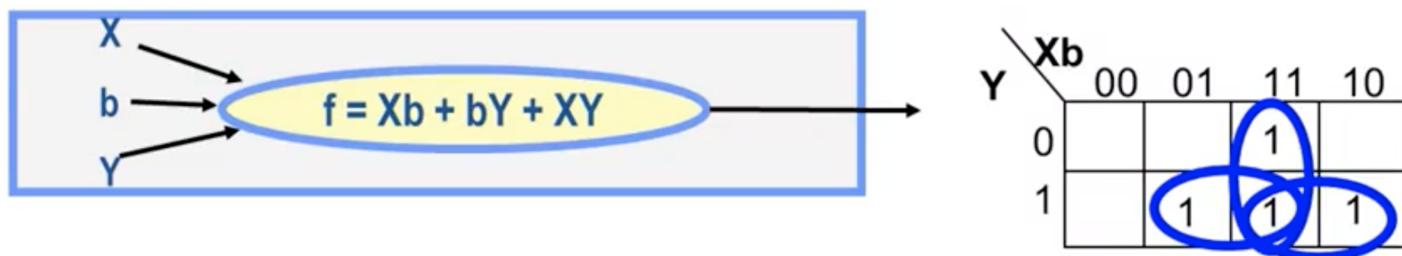
		ab	00	01	11	10
		cd	00	01	11	10
a	b	00			d	1
		01			d	1
a	b	11	1		d	d
		10			d	d

Patterns **a b c d = 1010, 1011, 1100, 1101, 1110, 1111** cannot happen

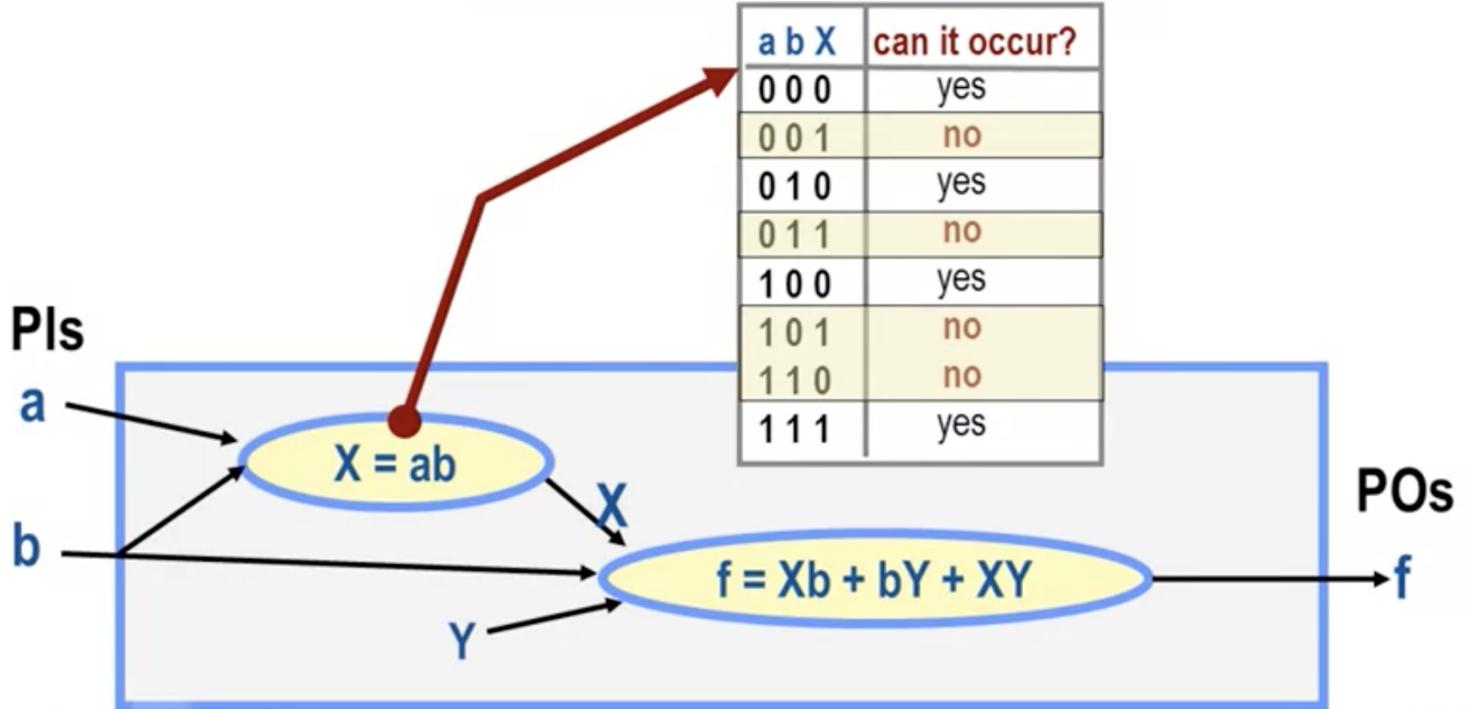
如上图, a, b, c, d 有六个不可能出现的pattern, 在这些pattern下, 布尔函数可以自由的选择取1还是取0, 在卡诺图中用d标记出don't cares, 可以将右侧的八个全部圈出来以获取更进一步的优化。

Multi-level don't cares

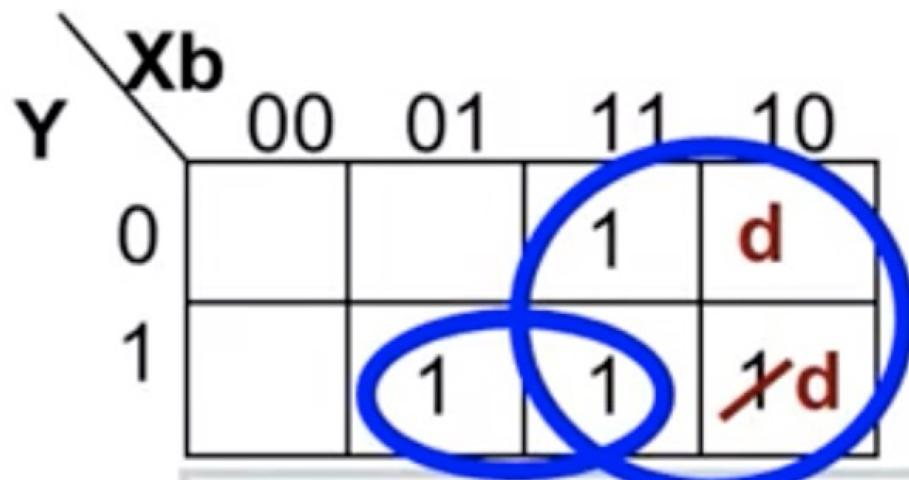
假设有一个布尔逻辑网络中的一个node f ,



只考虑这单个的node并不能得到任何关于don't care的信息, X, b, Y 可以取到任意值, 如果**知道 f 的输入 node**, 从中可以得到 X, a, b 不可能取到的pattern, 也可以得到 X, b, Y 不可能取到的pattern

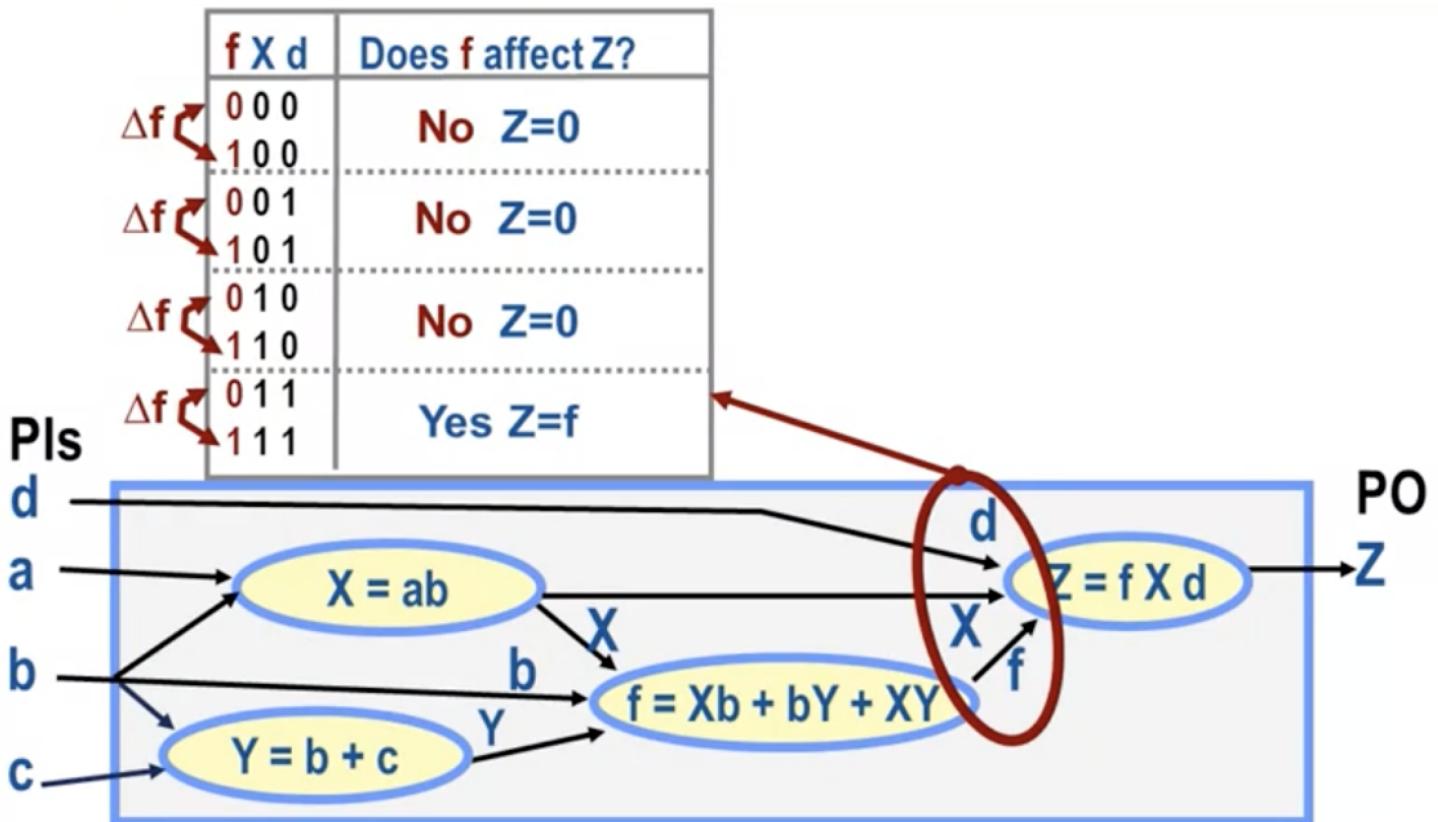


此时可以将 f 简化为 $f = X + bY$



Conclusion

另一方面，如果知道 f 的输出node在某些情况下对 f 的取值不敏感，那么也可以用其作为don't care来优化 f node

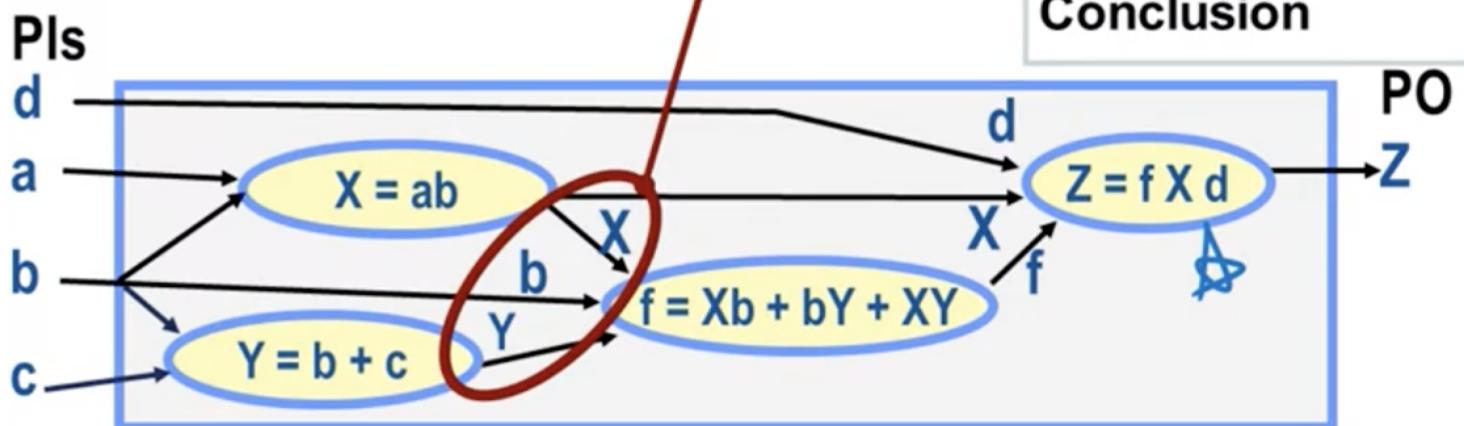


如上图, 用 Z 对 f 不敏感的pattern进一步的得出: 只要 X 为0 Z 就对 f 不敏感, 因此 $XbY = 0--$ 又是一个新的dont care, 进一步将 f 优化成常量1。

Pattern $XbY = 0--$ at input to f
will make network's Z output
insensitive to changes in f

Conclusion

	Xb	00	01	11	10
Y	0	d	d	x d	d
	1	d	x d	1	x d



Satisfiability don't cares

Satisfiability don't cares(SDC)表示的是每个node的输出与其输入不可能取到的pattern, 如一个node的function为 F 它的输入为 a, b, c 那么它的satisfiability don't cares写做 $SDC_F(F, a, b, c)$, $SDC_F(F, a, b, c) = 1$ 对应的pattern表征的是 F, a, b, c 不可能取到的pattern, 将 F 与 F 的布尔表达式进

行异或可以得到 SDC_F

$$SDC_F(F, a, b, c) = F \oplus F(a, b, c)$$

如：

$$F = ab + c$$

$$SDC_F(F, a, b, c) = F \oplus (ab + c) = F'ab + F'c + Fa'c' + Fb'c'$$

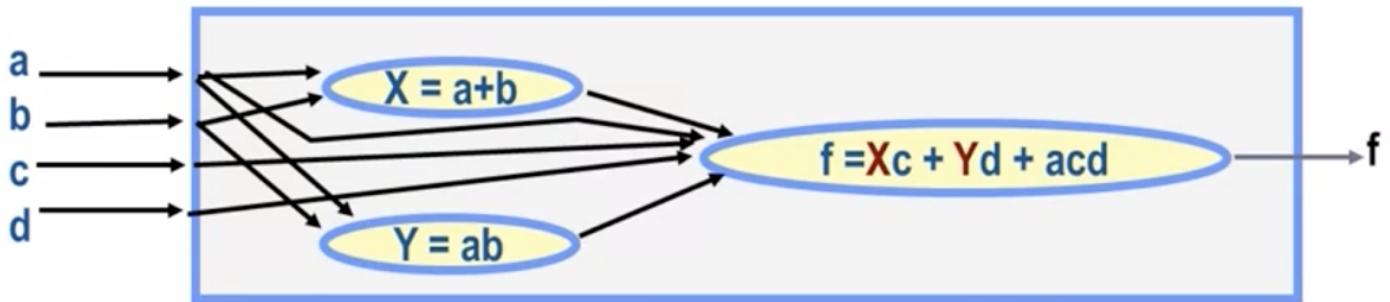
Controllability don't cares

SDC并不能实际用于对某个node进行简化，需要通过SDC得到Controllability don't cares(CDC)才能对node进行简化，CDC指的是某个node的输入不可能取到的pattern，如一个node的function为 F 他的输入为 a, b, c 那么它的Controllability don't cares写做 $CDC_F(a, b, c)$

计算方法：

- 计算出当前node所有输入的SDC
- 将所有这些SDC相或
- Universally Quantify掉所有当前node没有用到的变量

$$CDC_F = \forall_{\text{variable not used in } F} \sum_{X \in \text{input of } F} SDC_X$$



如上图的例子中

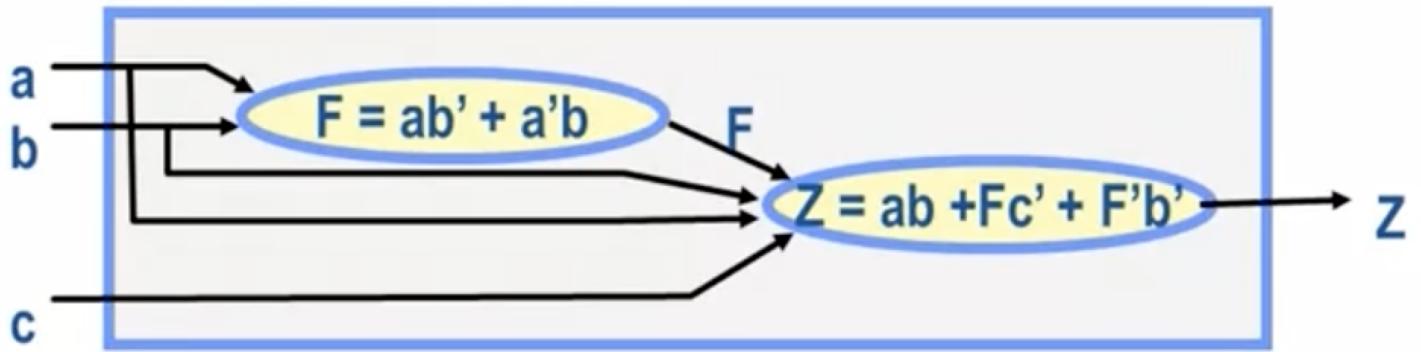
$$\begin{aligned} CDC_f &= \forall_b (X \oplus (a + b) + Y \oplus (ab)) \\ &= (X \oplus (a + b) + Y \oplus (ab))|_{b=1} \cdot (X \oplus (a + b) + Y \oplus (ab))|_{b=0} \\ &= X'a + X'Y + Ya' \end{aligned}$$

如果外部给出了一些可能出现的pattern(external don't cares)，他们也可以用来计算CDC，如上面的例子中如果给定 $b = 1, c = 1, d = 1$ 是外部不可能出现的pattern，将这些或进CDC计算式子中的 $\sum_{X \in \text{input of } F} SDC_X$ 部分即可

$$\begin{aligned} CDC_f &= \forall_b (X \oplus (a + b) + Y \oplus (ab) + bcd) \\ &= X'a + X'Y + Ya' + (a'cdX + acdY) \end{aligned}$$

Observability don't cares

Observability don't cares指的是输入的一些pattern使得当前node的输出在布尔逻辑网络的输出端不可被观测到，也就是说网络的输出对于当前node的取值不敏感，如下图所示， ODC_F 表示的是能使 Z 对 F 不敏感的 a, b 的pattern。



计算方法(如上面的例子):

- 计算出 $\overline{\partial Z / \partial F}$
- Universally quantify 掉所有F中没有用到的变量

$$ODC_F = \forall_{variable \ not \ used \ in F} (\overline{\partial Z / \partial F})$$

解释:

根据香农分解

$$Z = F \cdot Z_F + F' \cdot Z_{F'}$$

在 $Z_F = Z_{F'}$ 时， Z 对 F 不敏感，也就是 $Z_F \overline{\oplus} Z_{F'} = 1 \Rightarrow \overline{Z_F \oplus Z_{F'}} \Rightarrow \overline{\partial Z / \partial F} = 1$
如上图的例子中

$$\begin{aligned} ODC_F &= \forall_c \overline{(ab + Fc' + F'b')|_{F=1} \oplus (ab + Fc' + F'b')|_{F=0}} \\ &= \forall_c (ab + ac' + a'b'c' + a'bc) \\ &= ab \end{aligned}$$

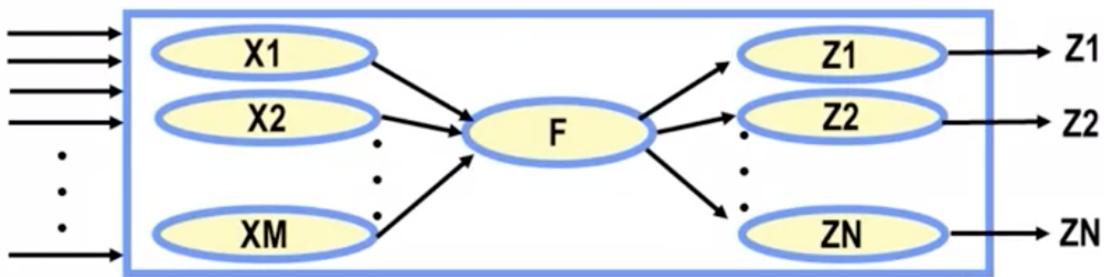
如果网络有很多输出都依赖于某一个node，那么只有所有的这些输出都不能观测到该node输出时才是ODC，因此更通用的计算方式是

$$ODC_F = \forall_{variable \ not \ used \ in F} \left(\prod_{output \ i} \overline{\partial Z_i / \partial F} \right)$$

Summary

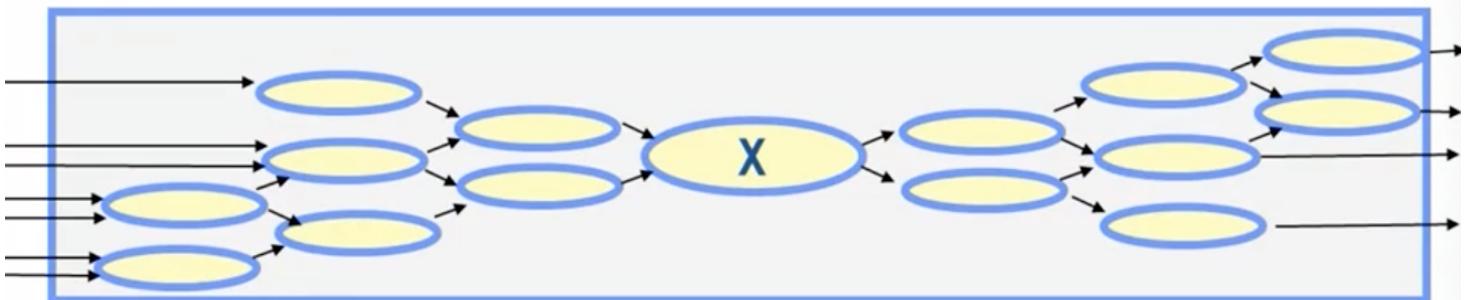
ODC和CDC给出了所有能用来优化某个node的的don't care的集合

Multi-Level Don't Cares: Are We Done?



- Yes – if your networks look **just like this**
 - More precisely, if you only want to get **CDCs** from nodes immediately “before” you
 - And if you only want to get **ODCs** for one layer of nodes between you and output

Don't Cares, In General



- But, this is what **real** multi-level logic can look like (!)
 - **CDCs** are a function of **all nodes** “in front” of **X**
 - **ODCs** a function of **all nodes** between **X** and any output
 - In general, we can **never get all** the DCs for node **X** in a big network
 - Representing all this stuff can be explosively large, **even** with BDDs

Summary: Getting Network DCs

- **How we really do it:** generally **do not get all the DCs**
 - Lots of tricks that trade off effort (time, memory) vs. quality (how many **DCs**)
 - **Example:** Can just extract “local **CDCs**”, which requires looking at outputs of immediate antecedent vertices and computing from the **SDC** patterns, which is easy
 - There are also incremental, node-by-node algorithms that walk the network to compute more of the **CDC** and **ODC** set for **X**, but these are more **complex**
- **For us, knowing these “limited” DC recipes is sufficient**
 - You now understand why **DCs** are implicit, why you must find them, and the big differences between **SDCs**, **CDCs**, and **ODCs**. This is good!