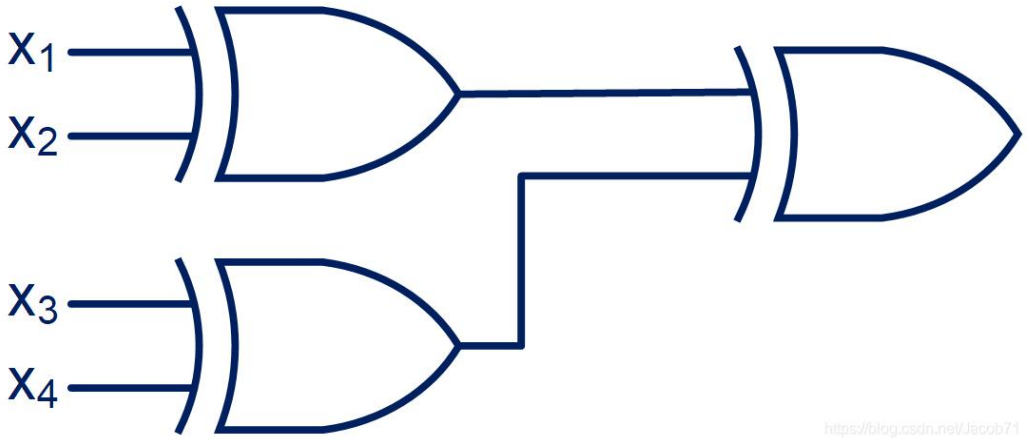
在形式化验证、数字系统的设计和验证中，许多任务都涉及大型命题逻辑公式的运算。二元决策图（BDD）已经成为许多应用的首选表示方法。1986 年，Bryant 发表论文指出归约有序的二元决策图是布尔函数的规范表示。

几个基本概念：
布尔函数（Boolean function）描述如何基于对布尔输入的某种逻辑计算确定布尔值输出，它们在复杂性理论的问题和数字计算机的芯片设计中扮演基础角色。比如下面的逻辑电路：



可以使用布尔函数：$(x_1 \oplus x_2) \oplus (x_3 \oplus x_4)$来表示。

有 n 个变量的布尔函数 F 为：

$$F : \{0,1\}^n \to \{0,1\}$$

函数有2^n个不同的可能输入，输出为一个 True 或 False 的布尔值，我们使用 1 或 0 来表示。

我们定义新的具有 n-1 个变量的布尔函数：

$$F_{x_1}(x_2, ..., x_n) = F(1, x_2, x_3, ..., x_n)$$

$$F_{x_1'}(x_2, ..., x_n) = F(0, x_2, x_3, ..., x_n)$$

F_{x_1}和$F_{x_1'}$称为 F 的余因子(cofactor)。F_{x_1}为正因子，$F_{x_1'}$为否定因子。

香农展开（Shannon's expansion），或称香农分解（Shannon decomposition）是对布尔函数的一种变换方式。它可以将任意布尔函数表达为其中任何一个变量乘以一个子函数，加上这个变量的反变量乘以另一个子函数，即：

$$F(x_1, ..., x_n) = x_i F_{x_i} + x_i' F_{x_i'}$$

INF范式（If-then-else Normal Form – INF）

If-then-else运算符表示为$x \to y_0, y_1$，则INF运算符定义为：

$$x \to y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

即，如果*x*值为1，则结果为*y0*，否则结果为*y1*。

所有逻辑运算符都可以仅使用 if-then-else 运算符和常量0和1表达。此外，if-then-else运算可以这样实现，即所有的测试都只在（未取反的）变量上进行，变量不会出现在其他地方。比如，¬*x*就是*x*→*0,1*。这样，if-then-else算子产生了一种新的范式 – INF范式（If-then-else Normal Form – INF）。

INF范式是完全由 if-then-else 运算符和常量构建的布尔表达式，因此仅需要对变量执行测试。

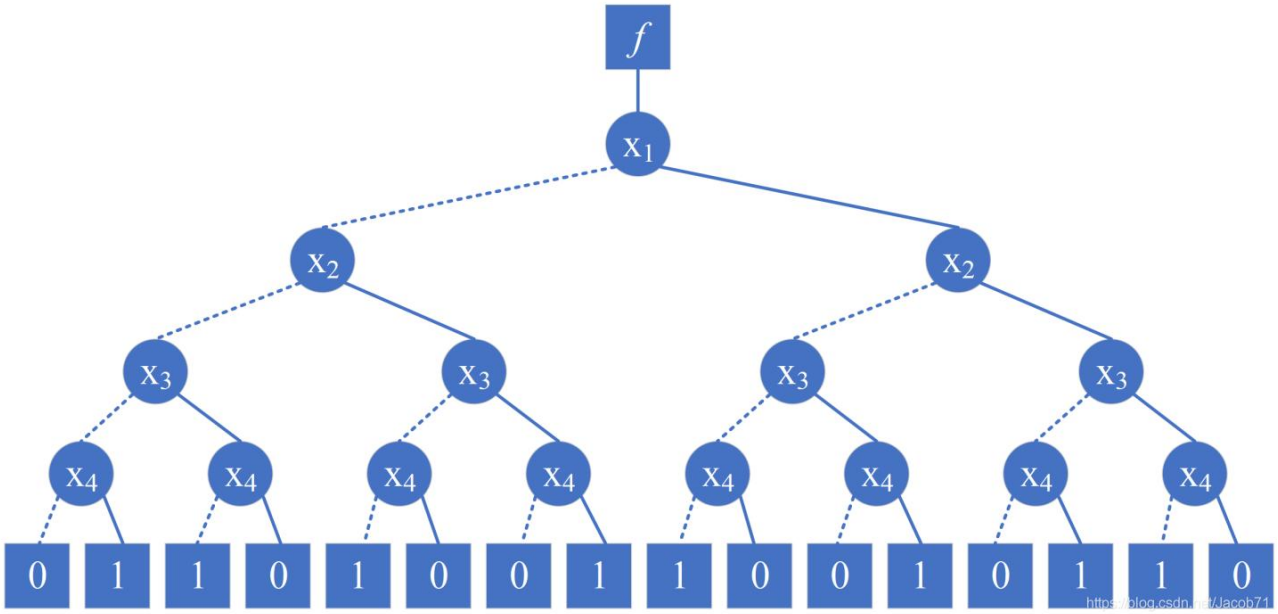
用*t[0/x]*表示将 *t* 中的变量 *x* 赋值为 0 得到的布尔表达式，得到以下等式：

$$t = x \rightarrow t[1/x], t[0/x]$$

这个就是表达式 *t* 对变量 *x* 的香农展开。这个简单的方程有很多有用的应用。第一个是可以从任何表达式 *t* 生成一个 INF。 如果 *t* 不包含任何变量，则它等于 0 或 1，即一个INF。否则，我们就生成表达式*t*中的一个变量 *x* 的香农展开式。其中 *t[1/x]* 和 *t[0/x]* 比表达式 *t* 少一个变量，这样可以递归做香农展开。

二元决策树（Binary Decision Tree）

二元决策树与二叉树类似，上面的逻辑电路/函数对应的有序二元决策树如下图：



标记为 f 的顶部节点为函数节点，标记有变量名字的圆形节点为内部节点，底部的正方形节点为终端节点（标记为1-True或0-False）。给定一组变量值，计算函数 f 的值，只需要从函数节点沿着路径找到终端节点，终端节点的标记即为函数的值。

一个内部节点，如果所标记变量的值为1，则沿着实线弧前进（也称为then弧/边，或High弧/边）；否则沿着虚线弧前进（也称为else弧/边，或low弧/边）。

在一个有序二元决策树中，变量在所有的路径上出现的顺序是一样的，如上图中：$x_1 < x_2 < x_3 < x_4$。

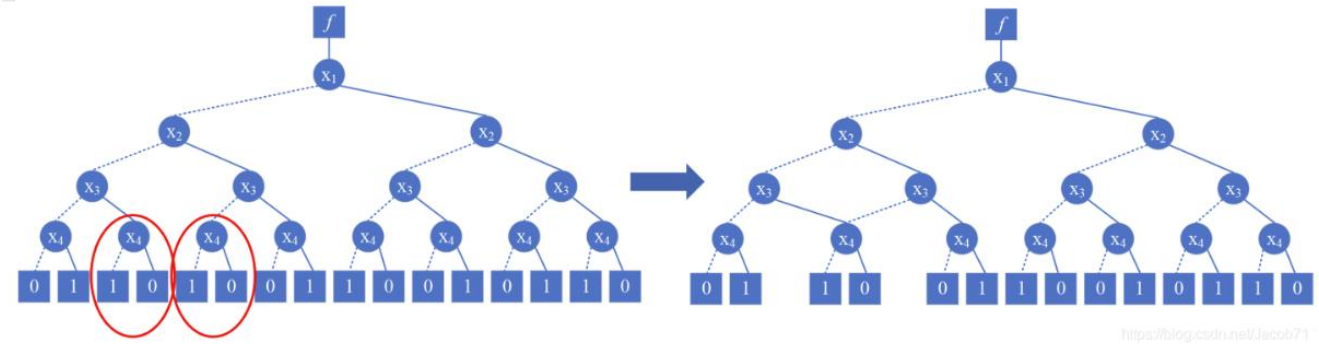
二元决策树最大的问题是它们的规模。一个有 n 个变量的二元决策树有$2^n - 1$个节点，加上2^n个最低层次终端节点的链接，指向返回值0和1。

二元决策图(Binary Decision Diagrams - BDD)

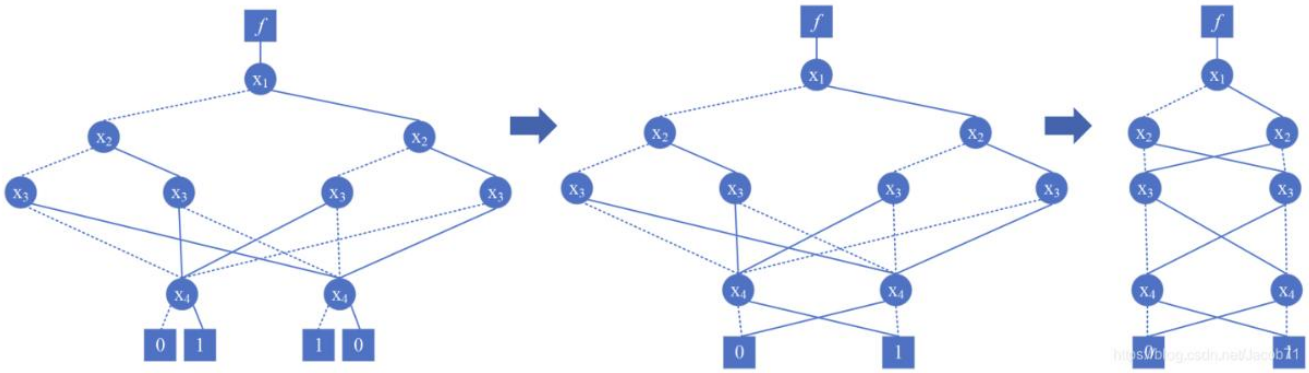
归约（Reduction）

通常，BDD通过对BDT归约而得到。归约由以下两条规则的应用组成，从决策树开始，一直到两条规则都不能应用为止。

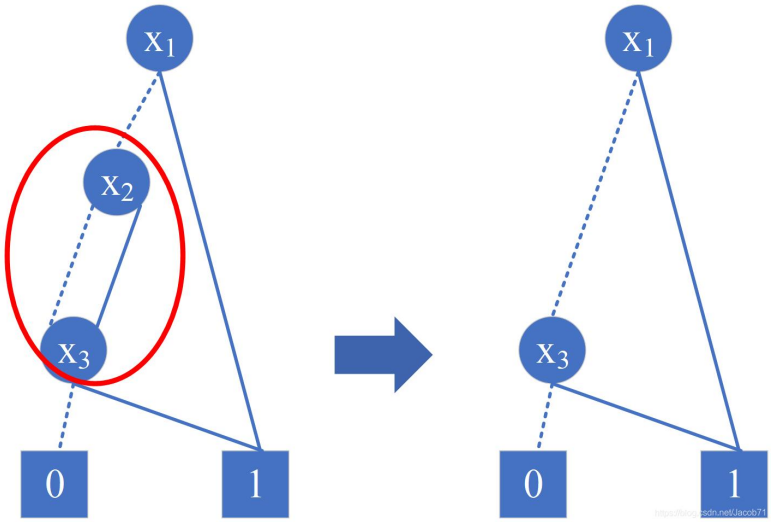
1. 如果两个节点是终端节点且具有相同的标签，或者是内部节点且具有相同的子节点，则将它们合并。
2. 如果内部节点的 high 边和 low 边指向相同的子节点，则将该节点从图中删除，并将其父节点重定向到子节点。



上图左图中，红色椭圆中的两个节点相同，应用规则1得到右图。重复应用规则1：



应用规则2，简化下图中的左侧椭圆中的节点，得到右侧图



通常，我们说BDD是指ROBDD，即归约有序的二元决策图

现在我们给出定义：

BDD是一个有根的，有向无环图：

1. 一个或两个出度为0的终端节点，标记为0或1，并且
2. 出度为2的变量节点 *u* 的集合。节点的2个出边由 *low*(*u*) 和 *high*(*u*) 定义（在图中显示为虚线和实线），节点关联变量 *var*(*u*)。

如果在图中的所有路径上，变量都遵循给定的线性顺序$x_1 < x_2 < \cdots < x_n$，则BDD为有序的(OBDD)。

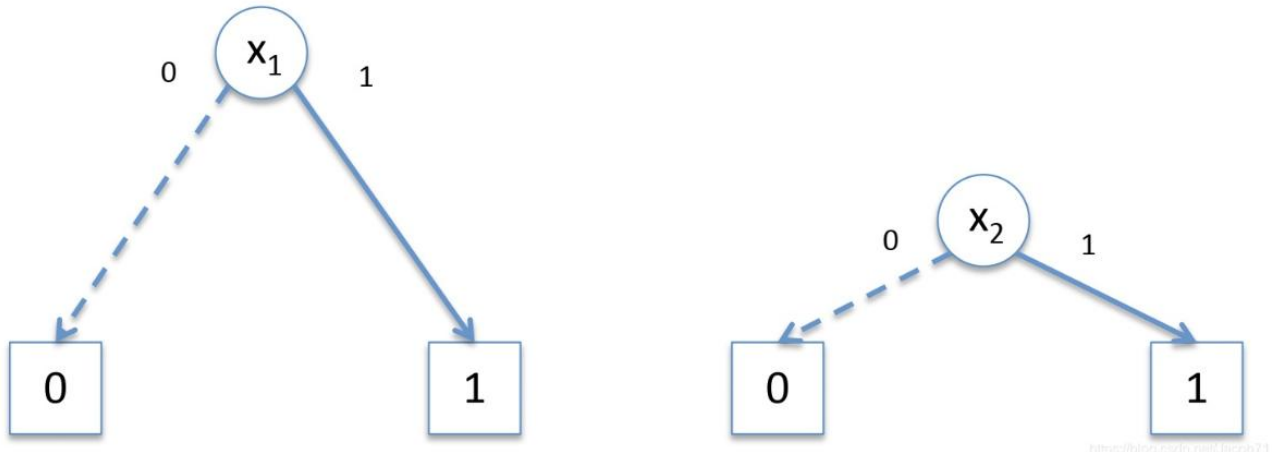
一个(O)BDD是归约的（R(O)BDD），如果：

1. (唯一性)没有两个不同的节点 *u* 和 *v* 具有相同的变量名和 low- 和 high- 子节点，即: *var*(*u*)=*var*(*v*), *low*(*u*)=*low*(*v*), *high*(*u*)=*high*(*v*)，这意味着 *u*=*v*，并且
2. (非冗余测试)没有变量节点 *u* 具有相同的 low- 和 high- 子节点，即: *low*(*u*)≠*high*(*u*)

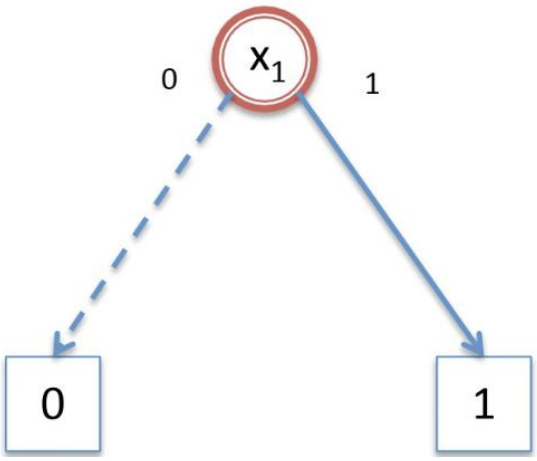
构建ROBDD（简写为BDD）

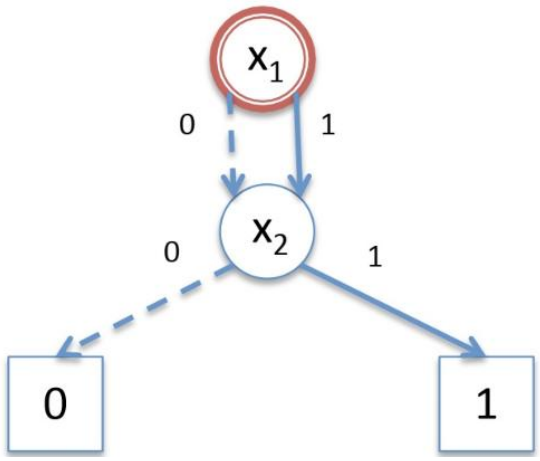
构造BDD的简单方法是构造一个二元决策树，然后逐步消除冗余并标识相同的子树。但是，因为需要构造原始决策树，所以需要指数级时间。

另一种方法是在构建BDD的过程中同时进行BDD的归约过程。将每个布尔函数作为表达式，使用BDD进行运算、归约，得到最终的BDD。以$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$为例，看布尔运算如何使用BDD实现。首先，我们构建$x_1 \vee x_2$的BDD。变量x_1和x_2的BDD为（变量顺序：$x_1 < x_2$）：

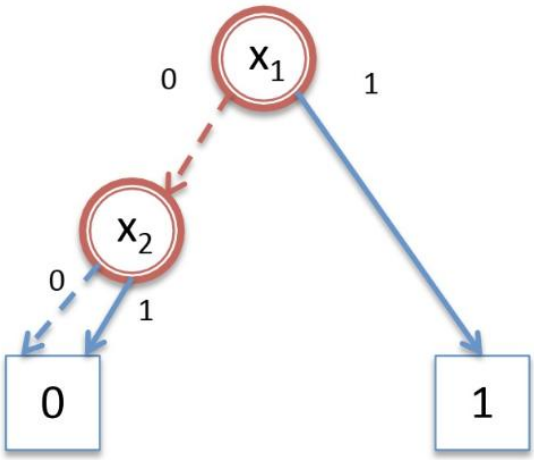


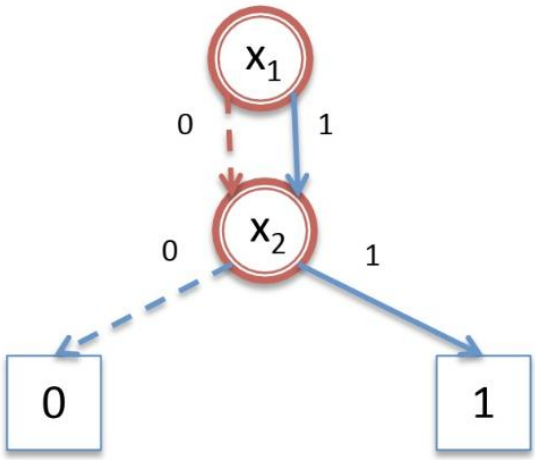
对两个具有相同变量顺序的BDD，在应用任何布尔操作时，需要从根节点开始，沿着平行路径到终端节点。一旦到达终端节点，将指定的布尔运算应用于布尔常数0和1，以形成指定路径的结果。对于没有出现的变量，需要进行(隐式地)扩展。在这个例子中，左边是x_1，右边是假设的x_1，隐式扩展如下：

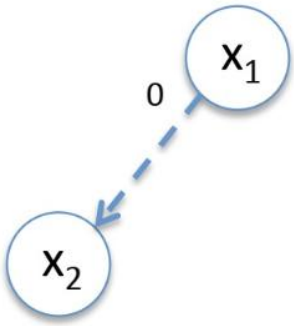




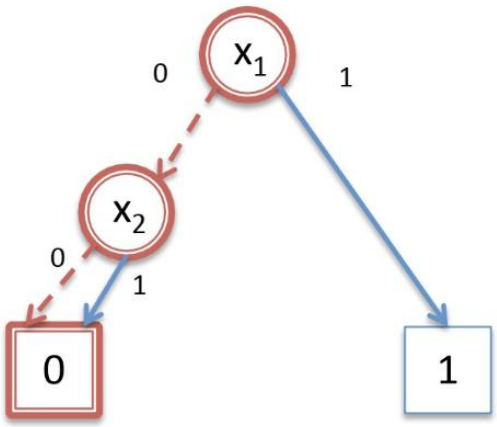
沿着x_1节点的左边和右边往下到达子节点，同样这里需要进行(隐式地)扩展，得到一个假设的x_2，如下图，底部是这一步构造的BDD。

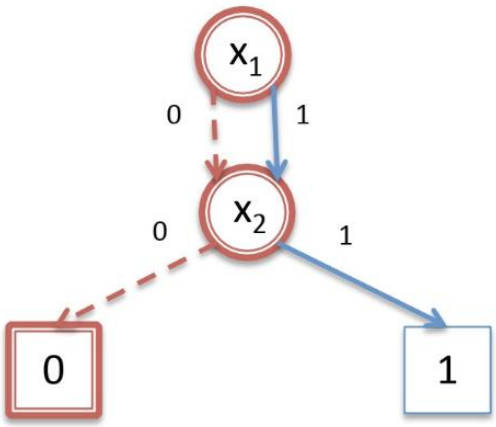


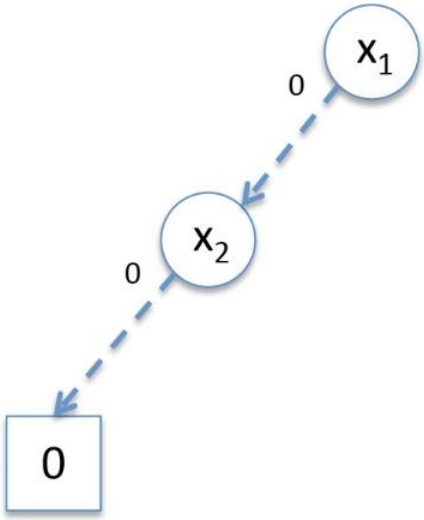




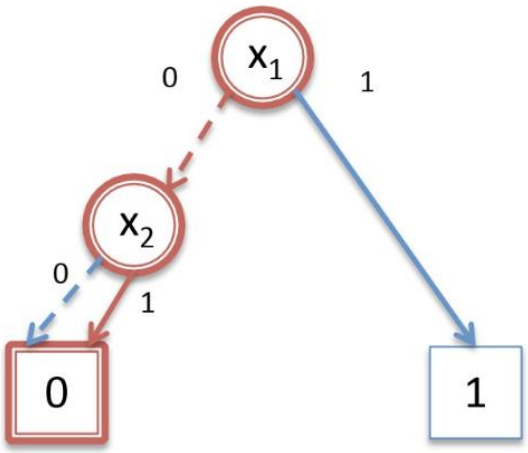
在两个给定的BDD中都沿着0-边往子节点走。两者都指向0，我们计算$0 \vee 0 = 0$，并将0作为结果，得到下面底部的BDD。

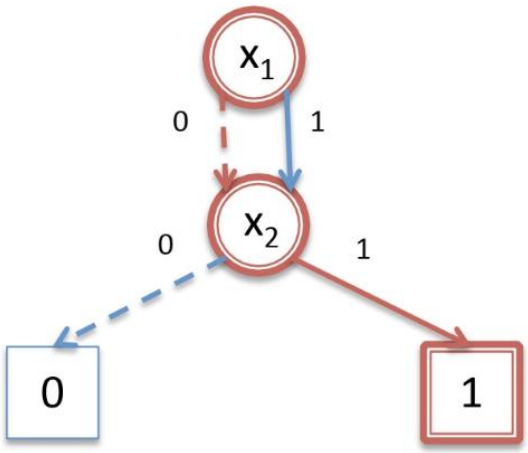


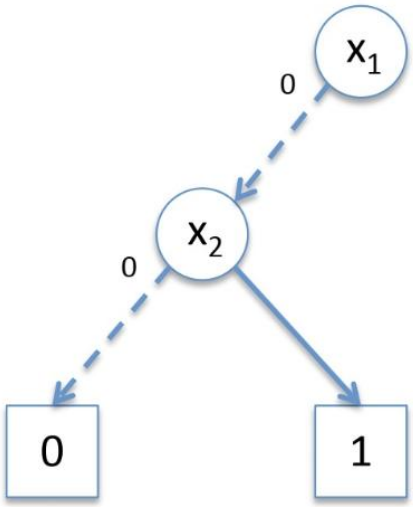




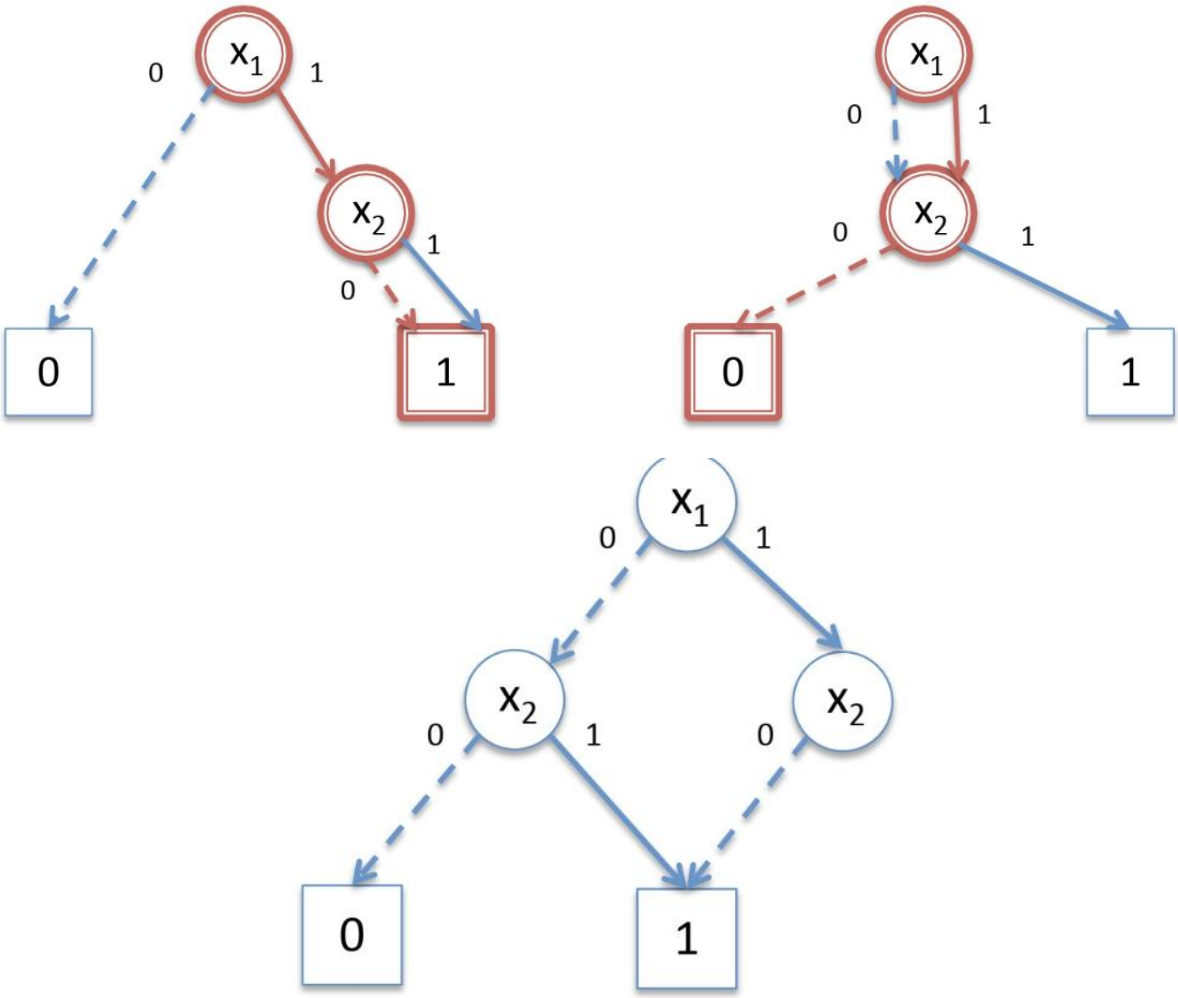
完成终端节点处理后，返回到上一层到x_2节点，并沿着1-边往子节点走，左边的图上是标记为0的终端节点，右边的图上是标记为1的终端节点，计算$0 \mathrm{V} 1 = 1$，得到下面底部的BDD：



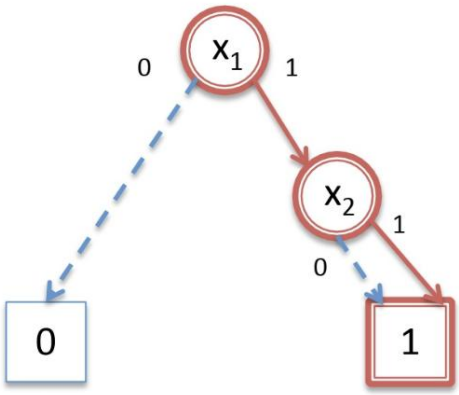


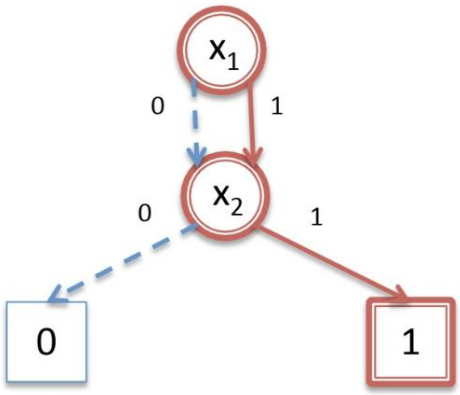


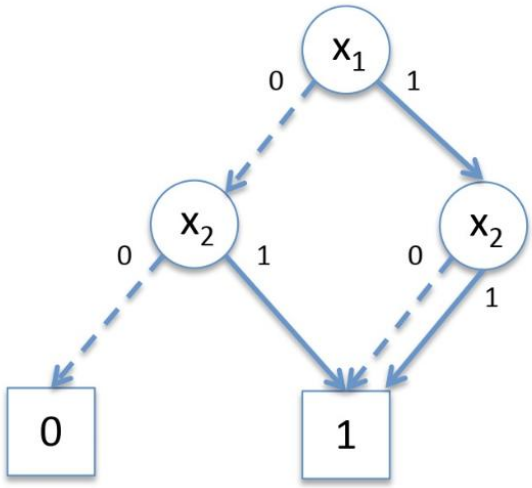
继续从x_2节点返回x_1节点，沿着x_1节点的1边往下。左图需要进行(隐式地)扩展，得到一个假设的x_2节点。然后，沿着x_2节点的0边往下，得到$1 \vee 0 = 1$，得到下面底部的BDD：



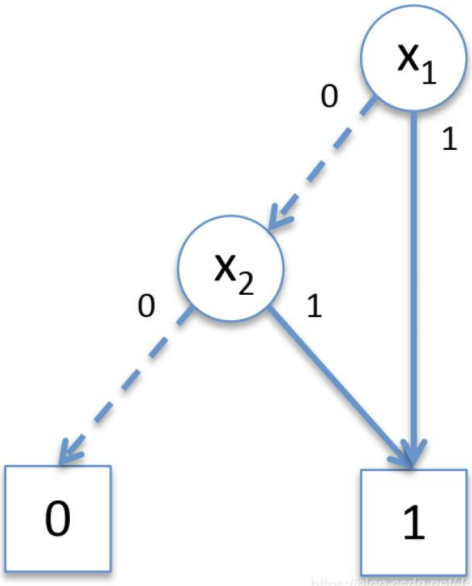
最后沿着x_2节点的1边往下，再次到达标记为1的终端节点，如下图。







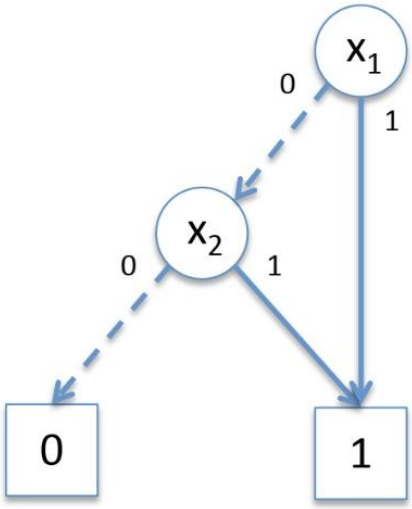
从图中可以看到，右侧x_2节点的low和high子节点是相同的，按照前面提到的规则2，需要消除这个节点，只返回一个对常量1终端节点的引用。至此，我们得到了$x_1 \vee x_2$的已经熟悉的BDD，如下图所示。

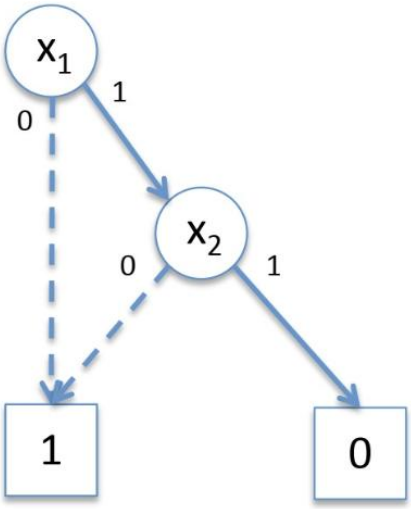


由此，我们可以总结两个二元决策图应用布尔运算的算法如下：

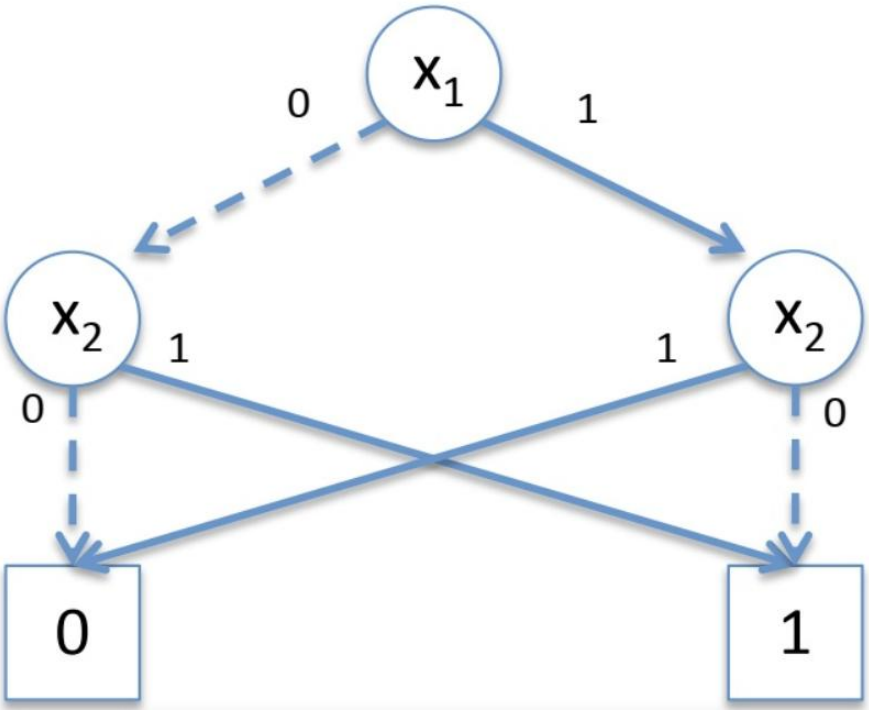
- 并行地遍历两个图，在两个给定的图中始终同时沿着0-边或1-边移动
- 当一个变量在一个图中存在，而不在另一个图中时，我们就像它存在并且有相同的high和low子节点一样继续进行（前面提到隐式地扩展，即实际上并不会创建节点，而是直接递归调用下去，后面的代码分析中可以看到，这里采用扩展结点是为了方便说明）
- 当到达两个图的终端结点0或1时，对这些常量应用布尔运算并返回相应的常量
- 如果对low和high子节点的运算返回相同的节点，不构造一个新节点，而只是返回在树中已经获得的单个节点。避免冗余
- 如果将要构造一个已经在结果图中某处的节点(也就是说，具有相同的变量标签和相同的后续节点)，不要创建一个新的副本，而只是返回已经存在的节点

对于$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$，按照上述的步骤构建$(\neg x_1 \vee \neg x_2)$，如下图的右侧：





用上述步骤完成2个BDD的运算，得到下面最终的BDD：



BDD构建算法

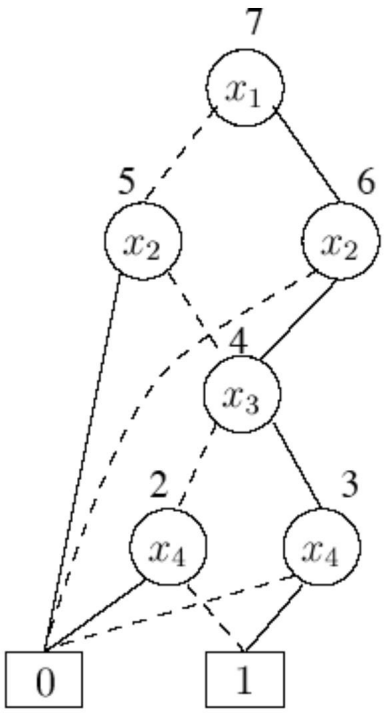
本节介绍构建BDD的算法以及实现，代码实现以BuDDy开源代码为例。

数据结构

算法实现需要2个核心的数据结构，即节点表和节点hash表。

节点表

这里以数组形式描述节点表，如下图所示，数组下标使用 *u* 表示，即节点ID，其中数组中第0和1元素预留给终端节点。BDD的变量以索引形式表示，如按照一定顺序排列的变量$x_1 < x_2 < \ldots < x_n$，使用*1,2,⋯,n* 索引来表示。节点表中的一个节点元素由变量、左子节点和右子节点组成，即*var*(*u*) = *i*, *low*(*u*)=*l*, *high*(*u*)=*h*。节点表*T:u⟼(i,l,h)*，使用节点索引查询节点的信息，即*var*(*u*) = *i*, *low*(*u*)=*l*, *high*(*u*)=*h*。



$T : u \mapsto (i, l, h)$

u	var	low	$high$
0	5		
1	5		
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

SAT 问题是判断给定命题公式是否存在可满足指派的过程，而#SAT 问题计算给定布尔公式的可满足指派的个数，也就是模型计数.目前，#SAT 求解器主要分为直接求解器和基于编译的求解器.基于编译的求解器在求解能力方面表现更突出，但是求解效率不如直接求解器[1].

直接求解主要包括精确求解和近似求解.目前，精确求解算法主要包括:1)Birnbaum 和 Lozinskii 提出的基于 DP 的#SAT 求解算法 CDP(counting models using Davis-Putnam procedure)[2]，该算法基于归结原理实现，归结原理的主要思想是通过推出空子句来判定给定子句集的不可满足性；2)Sang 等结合组件缓存和子句学习策略开发的 Cachet 求解器[3-4]，该系统调用了 SAT 求解器 zChaff，通过记录出现冲突和子公式模型个数来简化计算，从而加快求解效率；3)Thurley 提出的 sharpSAT 求解器[5]，它与 Cachet 求解器相比加入了 BCP 策略，并且修改了缓冲管理模式，较 Cachet 在求解效率上有了较大的提高.此外，近似求解算法主要包括 SampleCount，ApproxCount 等；基于编译的求

解器包括 C2D, Dsharp, SDD, cnf2obdd 等.