1. SAT

1. 文字：一个变量或者它的补。
2. 子句：合取范式中两个合取符号间的部分叫子句。
3. 单元子句、单元文字：子句中其他变量的逻辑值都为 0，只剩一个文字未被赋值，则该文字是单元文字，对应的子句是单元子句。
4. 矛盾子句：子句中所有文字的逻辑值都为 0。
5. 可满足问题：一个合取范式，存在一组赋值使所有子句的逻辑值都为 1。
6. SAT 求解算法：1. 局部搜索算法（不完备算法）；2. DPLL 算法（完备算法）
7. DPLL：

递归：

DPLL 递归形式，入参：一个合取范式和一组变量赋值。

```
DPLL(formula，assignment){
    necessary==deduction(formula,assignment)；//找出必要的变量赋值
    new_asgnmnt=union(necessary,assignment)；//得到新的一组赋值
    if(is_satisfied(formula,new_asgnmnt))//判断是否可满足
        return SATISFIABLE；
    else if(is_conflicting(formula,new_asgnmnt))//判断是否出现冲突
        return CONFLICT；
    var=choose_free_variable(formula,new_asgnmnt)；//挑选一个未赋值变量
    asgnl=union(new_asgnmnt，assign(var,1))；//给未赋值变量赋 1，得到新的赋值
    if(DPLL(formula，asgn 1)==SATISFIABLE)//递归判断
        return SATISFIABLE；
    else{
        asgn2=union(new_asgnmnt，assign(var,0))；//给未赋值变量赋 0
        return DPLL(formula，asgn2)；//递归判断
    }
}
```

迭代：

DPLL 迭代形式

```
status = preprocess()；//预处理
if (status!=UNKNOWN) return status；
while(1) {
    decide_next_branch()；//变量决策
    while (true){
        status = deduce()；//推理（BCP）
        if (status == CONFLICT){
            blevel = analyze_conflict()；//冲突分析
            if (blevel == 0)//如果冲突层次为顶层，则逻辑式直接不可满足
                return UNSATISFIABLE；
            else backtrack(blevel)；//智能回溯
        }
        else if (status == SATISFIABLE)
            return SATISFIABLE；
        else break；
    }
}
```

迭代比递归的优势:

1) 递归速度慢且容易发生溢出，相对于迭代就有很多自身的劣势。
2) 迭代具有非时间顺序回溯（智能回溯）的优势。

- 预处理：所谓预处理就是根据逻辑蕴含推理以及冗余子句删除和添加等值子句等相关技术。
- 变量决策：如何快速有效地决策出下一个将要赋值的变量。
  - MOM 方法：

    思想：如果一个子句长度很小，那它就很不容易被满足，所以我们要优先考虑它们，给予它们更高的权重。

    方法要点：子句长度短，出现频率高。

- VSIDS 方法：

  1) 为每一个变量设定一个 score，这个 score 的初始值就是该变量在所有子句集中出现的次数。
  2) 每当一个包含该文字的冲突子句被添加进子句库，该文字的 score 就会加 1。
  3) 哪个变量的 socre 值最大，就从这个变量开始赋值。
  4) 为了防止一些变量长时间得不到赋值，经过一定时间的决策后，每一个变量的 score 值都会被 decay，具体方式是把 score 值除以一个常数（通常为 2-4 左右）。——“周期衰减”

  优点：统计数据与变量赋值状态无关，因此系统资源开销非常低。可以显著提高求解器性能。

- 推理：通过推理来减少一些不必要的搜索，加快效率。主要依赖单元子句规则。BCP 算法。

  - BCP

    1) Counters 方法：每个子句有两个计数器，一个记录逻辑值为 true 的文字数，另一个记录逻辑值为 false 的文字数。如果一个 CNF 实例有 m 个子句，n 个变量，每个子句平均有 L 个文字，则有一个变量被赋值的时候，会有平均 mL/n 个计数器需要更新，在回溯的时候，每取消一个变量赋值，也会平均有 mL/n 个计数器的更新。如果一个子句的 false 计数器的值等于该子句的文字数，则说明出现了冲突；如果一个子句的 false 计数器的值等于该子句的文字数减 1，则该子句是单元子句。
    2) head/tail 方法：每个子句存在一个数组中，并且有两个指针，一个头指针，一个尾指针，每个变量 v 有四个附加链表，分别装有句头是 v 的子句，句头是非 v 的子句，句尾是 v 的子句，句尾是非 v 的子句。所以，如果有 m 个子句，四个链表中就存放着 2m 个子句。移动指针有四种情况：

       1. 第一个遇到的文字为真，说明子句已经满足，直接忽略该子句。
       2. 第一个遇到的文字未赋值且不是句尾，那就将该子句删除，放到

对应文字的链表中。

3. 头尾指针之间只有一个变量未赋值且其他文字都为假，则出现单元子句。
4. 头尾指针之间所有文字都为假，产生冲突，回溯。

当一个变量被赋值的时候，平均有 m/n 个子句需要更新（m 为子句数，n 为变量数）。

3) 2-literal watching 方法（应用于 zChaff 求解器）：与 head/tail 方法相似，为每个子句关联两个指针，但这两个指针没有头和尾之分，位置任意。它与 head/tail 方法的关键区别是回溯时指针的位置无需移动，取消一个变量的赋值时间的开销是常量，但坏处是只有遍历完所有文字才能找到单元文字。两个指针随时监视子句中任意两个未被赋值为 0 的文字。每个变量 v 有两个附加链表，对应变量的正负形态，存放的是对应的子句。设初始时变量 v 赋值为 1，则搜索将在一个含有文字-v 的子句中进行（因为 v 已满足），并且此时一个监视指针指向文字-v，继续搜索，该过程中有四种情况：

1. 如果存在文字 L 不是该子句的另外一个被监视文字，则删除指向文字-v 的指针，并添加指向文字 L 的指针，相当于指针移动。
2. 如果唯一符合条件的文字 L 是另外一个被监视文字，且它没有被赋值，则该被监视文字是单元文字，该子句是单元子句。
3. 如果唯一符合条件的文字 L 是另外一个被监视文字，且它已经被赋值为 1，说明该子句已经满足，不需要进行任何处理。
4. 如果所有文字都已经被赋值为 0，那么出现冲突，回溯。

- 冲突分析：寻找冲突的原因并学习。
- 智能回溯：回跳多个决策层，并把会导致冲突的子句加入到子句集中。
  - 时序回溯：直接返回上一层，将变量取值翻转。
  - 非时序回溯：结合冲突学习，智能分析，跳回多个决策层，并且会将导致冲突的子句加入到子句集中。

8. 子句数据结构：
   - 数组方式：数组采用连续空间存储，内存利用率和存储效率更高，局部访

问能力更强。连续存储能提高 cache 命中率，从而增加计算速度。但不灵活。

- 链表方式：便于对子句进行增加和删除操作，存储效率低。因为缺少局部访问能力，往往会造成缓存丢失。
- 注：head/tail 和 2-literal watching 都被称为“膨胀数据结构”，它们都采用数组存储子句，最具竞争力。

例题：设计一个基于数组的数据结构存储子句。

方案 1：head/tail

用数组存储子句的文字，对于每个数组设置两个指针，分别为头指针与尾指针，然后对于每个变量 v 设置 4 个链表，分别装有句头是 v 的子句，句头是非 v 的子句，句尾是 v 的子句，句尾是非 v 的子句，这样存储的子句，无论子句顺序及其中变量顺序的改变都不会影响所有变量的四个链表中的子句数量。取所有变量 v 是句头的子句与非 v 是句头的子句或者 v 是句尾的子句与非 v 是句尾的子句合取可以得到公式。

方案 2：2-literal watching

为每个子句关联两个指针，这两个指针位置任意，随时监视子句中任意两个未被赋值为 0 的文字。每个变量 v 有两个附加链表，对应变量的正负形态，存放的是含有对应文字的子句。该种结构的好处是回溯时指针的位置无需移动，取消一个变量的赋值时间的开销是常量。此外，重新分配最近分配和未分配的变量将比第一次分配的变量更快，并且可以减少内存访问总数，提高缓存命中率。

2. FDS

1. 一个程序 D 由五部分组成$D = <V, O, \theta, \rho, J, C>$
   - V：有限状态集合。包括程序中的变量和语句。
   - O：可观测状态集合。定义O属于V。一般没有说明的话，默认所有状态都可观测，因此O等于V。这个不重要，应该不考。
   - θ：初始条件。

- ρ：转换关系。考察的重点，先不管什么意思，下面举例说明。
- J：justice 集合，弱公平的。
- C：compassion 集合，强公平的。

例子：

$$x,\ y\colon \textbf{natural initially } x = y = 0$$

$$\left[\begin{array}{l} \ell_0: \ \textbf{while } x = 0 \textbf{ do} \\ \quad [\ell_1: \ y := y + 1] \\ \ell_2: \end{array}\right] \quad \| \quad \left[\begin{array}{ll} m_0: & x := 1 \\ m_1: & \end{array}\right]$$

➢ 首先表示V：

$$V\colon \left(\begin{array}{lcl} x, y & : & \text{natural} \\ \pi_1 & : & \{\ell_0, \ell_1, \ell_2\} \\ \pi_2 & : & \{m_0, m_1\} \end{array}\right)$$

a) 首先第一行就是程序最上面的初始化，左边两个变量一写，右边写个 natural。

b) 接下来定义π，程序有几个部分(用||连接)就定义几个π，每个π对应的元素就是每一行语句l_0~l_k

➢ 表示θ：

$$\Theta:\ \pi_1 = \ell_0 \ \wedge\ \pi_2 = m_0 \ \wedge\ x = y = 0.$$

θ好理解，因为它表示初始条件，而初始时，每个π都处于第一行语句l_0，再加上变量的初始化，把它们合取即可。

➢ 表示ρ：表示ρ之前得先说明几个定义：

1. π'这里不知道带个上标是什么意思，但是貌似表示的是下一个状态，记住就行，不需要知道是啥意思。
2. $pres(V)$这个是重点。对于V里面的每个元素都这样表示：$e = e'$，然后把他们用合取符号连接起来，上面例子的$pres(V)$就应该表示为：

$$pres(V):\ \pi_1 = \pi_1{}' \wedge \pi_2 = \pi_2{}' \wedge x = x' \wedge y = y'$$

3. at_{l_j}是一个缩写形式，表示$\pi_i = l_j$，　at'_{l_j}表示$\pi'_i = l_j$

4. $\rho_I = pres(V)$，ρ_{l_0}表示语句l_0转换成逻辑公式之后的一个符号。考试重点就是表示ρ_{l_k}

5. ρ: $\rho_I \vee \rho_{\ell_0} \vee \rho_{\ell_1} \vee \rho_{m_0}$，注意这里面不包括空的语句。

对于语句的表示，考试应该出的就是赋值语句、if 语句和 while 语句。其他的先不管。

a) 赋值语句：$y := e$

开始背：$at\_\ell_j \wedge at'\_\ell_k \wedge y' = e \wedge pres(V - \{\pi_i, y\})$。比如上面例子中的$m_0$就表示成：（$l_k$，我的理解就是下一步要执行的语句）

$$\rho_{m_0}: \pi_2 = m_0 \wedge \pi_2' = m_1 \wedge \pi_1' = \pi_1 \wedge x' = 1 \wedge y' = y$$

b) if 语句：**if** b **then** $\ell_1 : S_1$ **else** $\ell_2 : S_2$

开始背：（如果这里没有 else 语句，那l_2就是跳出 if 下一步要执行的语句）

$$at\_\ell_j \wedge \begin{pmatrix} b & \wedge & at'\_\ell_1 \\ & \vee & \\ \neg b & \wedge & at'\_\ell_2 \end{pmatrix} \wedge pres(V - \{\pi_i\})$$

c) while 语句：**while** b **do** $[\ell_1 : S_1]$

开始背：（l_k，我的理解就是跳出循环后下一步要执行的语句）

$$at\_\ell_j \wedge \begin{pmatrix} b & \wedge & at'\_\ell_1 \\ & \vee & \\ \neg b & \wedge & at'\_\ell_k \end{pmatrix} \wedge pres(V - \{\pi_i\})$$

上面的例子l_0就可表示为：

$$\rho_{\ell_0}: \quad \pi_1 = \ell_0 \quad \wedge \quad \begin{pmatrix} x = 0 \wedge \pi_1' = \ell_1 \\ \vee \\ x \neq 0 \wedge \pi_1' = \ell_2 \end{pmatrix}$$
$$\wedge \quad \pi_2' = \pi_2 \wedge x' = x \wedge y' = y$$

- 表示 J：这个集合一般都是固定的，程序中出现了几条语句，语句翻译后都会把$\neg at_\ell_j$这个符号加入 J 集合。

上面例子的 J 集合可表示为：

$$\mathcal{J}: \{\neg at\_\ell_0, \neg at\_\ell_1, \neg at\_m_0\}$$

- 表示 C：这个集合一般都为空，除非程序中有 request 语句，应该不会考。

3. OBDD

1. 题型 1：要么根据真值表画出 BDD，要么根据公式画出 BDD。这块比较容易，先列一下香农展开式：

$$f(A, B, C, \cdots) = Af(1, B, C, \cdots) \vee \bar{A}f(0, B, C, \cdots)$$

给个例子：

$$f = B(\bar{A}C \vee \bar{C}\bar{E}) \vee \bar{E}(\bar{A}B \vee \bar{B}D)$$

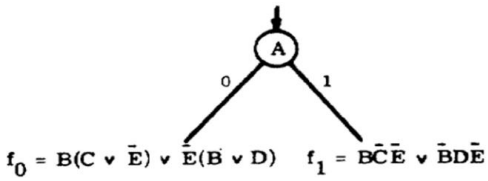
三步走（不断利用香农展开的过程）：

a) 固定一个变量，画出此变量的节点及 0 1 分支。

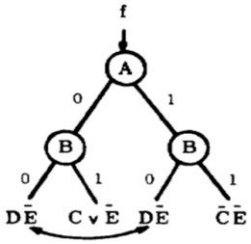
b) 看是否有分支可以合并，如果可以则合并，否则再选取另一个变量转到步骤 a）。

c) 直到分支节点处变为 0 或 1，则结束。

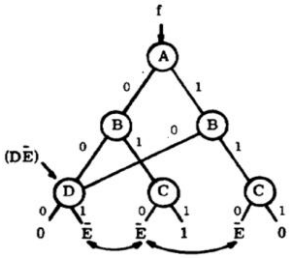
第一步：固定变量 A，画出两个分支，下面写下展开后的公式 f0,f1。



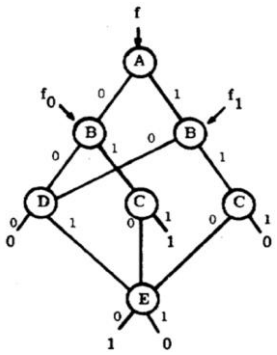
第二步：固定变量 B，画出分支，写出剩下的公式。发现有相同的，用双向箭头连接起来。



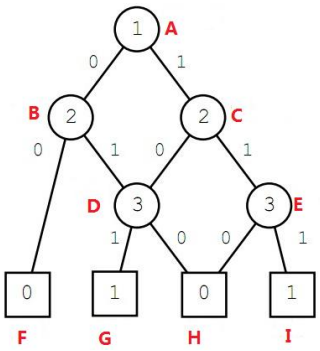
第三步：先把上一步的相同结点合并。固定变量 C，画出分支，固定变量 D，画出分支。继续。



第四步：固定变量 E，画出分支，结束。



2. 题型 2：化简图。这里就使用 PPT 里面给的那个 Reduction 操作。这里把流程走一遍。



Step1：先给图按照从上到下，从左到右的顺序标上字母。然后写出 vlist。vlist[1]就是索引号为 1 的结点。

vlist[1]: A
vlist[2]: B,C
vlist[3]: D,E
vlist[4]: F,G,H,I

Step2：从下到上、从终止结点到根节点对每个 vlist 操作。初始时 nextid=0，nextid 就是每生成一个新结点它的 id 号。首先是 vlist[4]，对于每一个元素构造一个<key,u>，其中 key 是该结点在图中对应子结点的 id 号，u 是该节点的字母。由于 vlist[4]是最后一层叶子结点，因此元素的 key 就用它们对应的值表示。然后对它们进行排序，key 相等的就排在一块，代表它们是一个结点。

vlist[4]
Q:<key,u>

<0,F>	**Sort by keys**	**<0,F>**
<1,G>		**<0,H>**
<0,H>		**<1,G>**
<1,I >		**<1,I >**

然后对于每一个 u，按照下面的格式把它们的属性值写出来。oldkey 里面是其子结点的 id 号，由于 vlist[4]的结点没有子结点，所以它们的 oldkey 就是对应的值。Oldkey 初始值是(-1，-1)。如果 key 相同，那就直接写一个 id。

```
F:  oldkey=(-1,-1);
    nextid=1;  F.id = 1;  subgraph[1]=F;
    oldkey=(0);

H:  H.id=1;   (已存在key相同的节点)

G:   nextid=2;  G.id=2;  subgraph[2]=G;
     oldkey=(1);

 I:   I.id=2
```

接下来是 vlist[3]:(这里面注意：结点有子结点的时候，要写 low，high 两个属性)

```
Q:<key,u>
<1,2,D>   Sort by keys   <1,2,D>
<1,2,E>                  <1,2,E>

D:  oldkey=(-1,-1);
    nextid=3;  D.id = 3;  subgraph[3]=D;
    D.low=F;  D.high=G;
    oldkey=(1,2);

E:  E.id = 3;
```

接下来是 vlist[2]：（这里注意如果一个结点的两个子结点 id 一样，那就忽略它）

```
Q:<key,u>
<1,3,B>

注：因为C.low.id = C.high.id，所以C.id=C.low.id=3;并且C不加入Q集合中

B:  oldkey=(-1,-1);
    nextid=4;  B.id = 4;  subgraph[4]=B;
    B.low=F;  B.high=D;
    oldkey=(1,3);
```

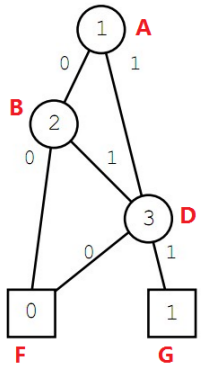
最后是 vlist[1]：

```
vlist[1]

Q:<key,u>
<4,3,A>

 A:  oldkey=(-1,-1);
     nextid=5;  A.id = 5;  subgraph[5]=A;
     A.low=B;  A.high=D;
     oldkey=(4,3);
```

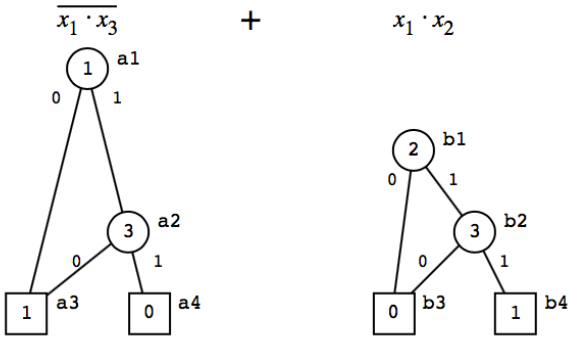
Step3：完成，画图：

subgraph[1]=F
subgraph[2]=G
subgraph[3]=D
subgraph[4]=B
subgraph[5]=A

D.low=F; D.high=G;
B.low=F; B.high=D;
A.low=B; A.high=D;



3. 题型 3：还有一种题型是给出两个 f，对它们进行<op>操作。这题需要两步，第一步是 Apply 操作，将两个图合并成一个图，第二步是 Reduction 操作，就是上面的化简。这里主要说 Apply 操作。例题：



a) f1 的根结点 v1，f2 的跟结点 v2，操作为<op>，整个过程就是在 f1 中取一个结点，在 f2 中取一个结点，然后比较。

b) 如果 v1 和 v2 均为终止结点，那么目标图上画一个终止结点，值为 v1<op>v2。

c) 如果 v1 和 v2 都不是终止结点，那就比较他们的索引值。

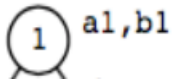
   1. 如果 index(v1)＝index(v2)＝i，目标图画一个非终止结点，对应 index＝i；然后在 low(v1)和 low(v2)递归地应用该算法，深度优先，完成之后，在 high(v1)和 high(v2)递归地应用该算法。
   2. 如果 index(v1)＝i，index(v2)＞i，目标图画一个非终止结点，对应 index＝i；然后，在子树中递归调用该算法。

d) 如果 v1 和 v2 一个是非终止结点，另一个是终止结点，终止结点的值是 x。

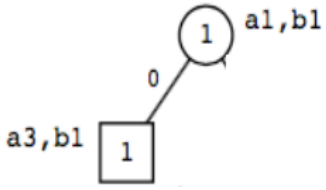
   1. 如果 x<op>v2 的值由 x 决定，目标图上画一个终止结点，值为 x。

2. 如果 x<op>v2 的值不由 x 决定，目标图上画一个非终止结点，index = index(v2)。

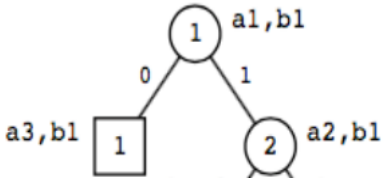
Step1：拿出 a1 和 b1，index(b1)>index(a1)，目标图画一个结点，index=index(a1)



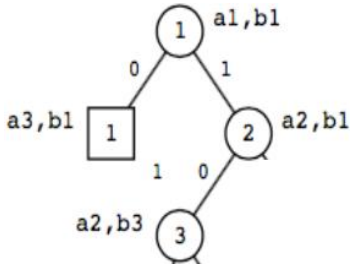
Step2：进入 a1 的 low 子树，拿出 a3 和 b1 比较，因为 1+b1=1，所以目标图 low(a1,b1)画一个终止结点(a3,b1)，值为 1。



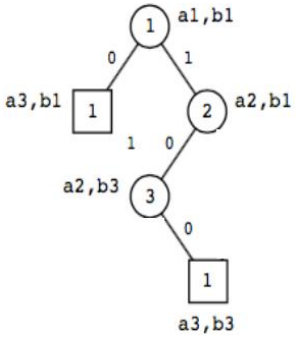
Step3：进入 a1 的 high 子树，拿出 a2 和 b1 比较，因为 index(a2)>index(b1)，所以目标图 high(a1,b1)画一个结点(a2,b1)，index=index(b1)。



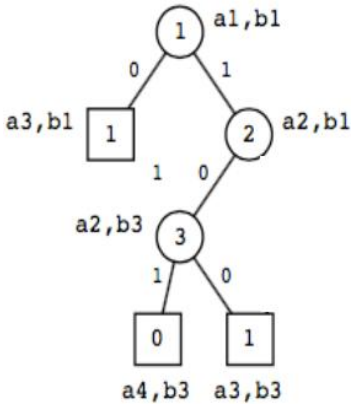
Step4：进入 b1 的 low 子树，拿出 a2 和 b3 比较，因为 a2+0=a2，所以目标图 low(a2,b1)画一个结点(a2,b3)，index=index(a2)。



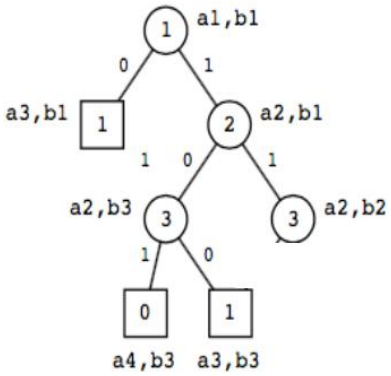
Step5：进入 a2 的 low 子树，拿出 a3 和 b3 比较，因为 1+b3=1，所以目标图 low(a2,b3)画一个终止结点(a3,b3)，值为 1。



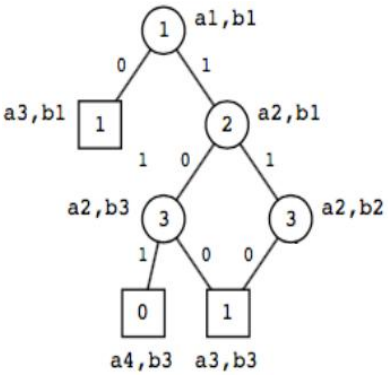
Step6：进入 a2 的 high 子树，拿出 a4 和 b3 比较，因为 0+0=0，所以目标图 low(a2,b3)画一个终止结点(a4,b3)，值为 0。



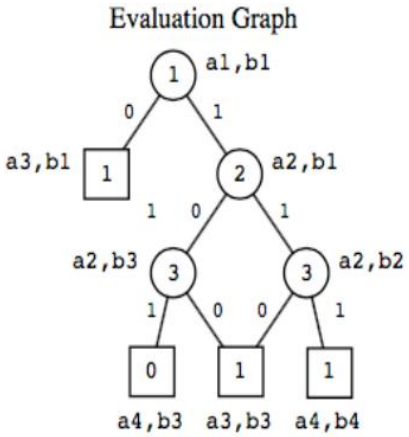
Step7：注意，这时候要回溯，之前是 a2 和 b3 比较，现在应该是拿出 a2 和 b2 比较，因为 inde(a2)=index(b2)=3，所以目标图 high(a2,b1)画一个结点(a2,b2)，index=3。



Step8：因为上一步 index 相等了，所以同时进入 a2 和 b2 的 low 子树，拿出 a3 和 b3，因为已经存在结点(a3,b3)，直接连接。



Step9：同时进入 a2 和 b2 的 high 子树，拿出 a4 和 b4，因为 0+1=1，所以目标图 high(a2,b2)画一个终止结点(a4,b4)，值为 1。比较完成，最终目标图完成。



4. Alloy

1. Alloy 就说两个例题，目的就是找出不符合命题条件的例子。

例子 1：

```
sig Platform {}
```
there are "Platform" things

```
sig Man {ceiling, floor: Platform}
```
each Man has a ceiling and a floor Platform

```
pred Above(m, n: Man) {m.floor = n.ceiling}
```
Man m is "above" Man n if m's floor is n's ceiling

首先我们来解读这个例子的含义，第一个，sig，主要作用就和 java 中的 class 是一样的，意思是要定义一个类，名称是 Platform，它里边不需要任何属性，或者说他有属性，但我们不关注，与我们的关注点无关。

第二个一样的意思是定义了一个 Man 的类，表示人，他的属性有两个分别是 ceiling 以及 floor，冒号表示这两个属性的类型是 Platform，所以用中文来理解，那就是说，每个人都有两个属性，一个是 ceiling 天花板，一个是地板 floor。第三个就是 pred，pred 用来定义谓词，上例说的是有一个 above（高于）的动词他有两个参数 m 和 n，并且这两个参数都是 man 类型，同时方法体中说明，m 的地板=n 的天花板，所以我们就能定位了，也就是 m 在 n 的上边，也就是说，m 高于 n。

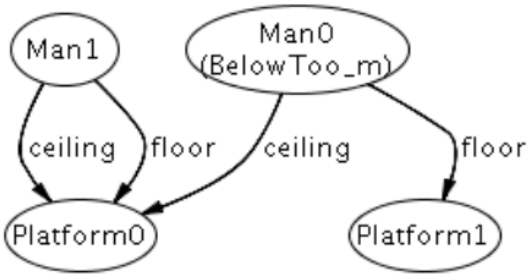
```
fact {all m: Man | some n: Man | Above (n,m)}
```
"One Man's Ceiling Is Another Man's Floor"

Fact 表示的是这个内容是给出的条件，也就是我们的已知条件。All 表示的是所有，也就是每一个，而 some 表示存在，所以我们从已知条件就可以得到一些信息：对于每一个 m，都存在 n 且高于 m。

```
assert BelowToo {
  all m: Man | some n: Man | Above (m,n)
}
   "One Man's Floor Is Another Man's Ceiling"?

check BelowToo for 2
   check "One Man's Floor Is Another Man's Ceiling"
   counterexample with 2 or less platforms and men?
```

接下来是 assert，表示假设，也就是说假设条件是存在一个 belowToo，内容是：对于每个 m 都存在 n，且 m 高于 n。

最后是 check，也就是检验，检验对象是假设条件，范围就是 for 后边的数字，2 表示有两个 platform 和两个人。也就是 plat0，plat1，m0 和 m1，我们要找不满足 belowToo 的，证明命题是错误的。因为有四个对象，所以有 2^4 种可能，也就是 16 种可能。可以找到一个反例：



对于 m1，有 m0 在他下边，可是对于 m0 来说，没有人在他下边，所以对于 m0，当 m1 的天花板和地面都在他上边这种情况，他就不满足了。

例子 2：

```
module examples/tutorial/grandpa

abstract sig Person {
  father: lone Man,
  mother: lone Woman
}

sig Man extends Person {
  wife: lone Woman
}

sig Woman extends Person {
  husband: lone Man
}

fact {
  no p: Person |
    p  in p.^(mother + father)
  wife = ~husband
}
```

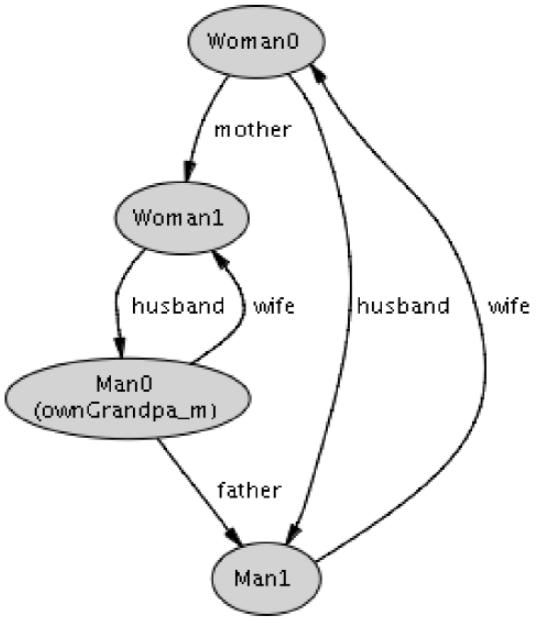
```
assert noSelfFather {
 no m: Man | m = m.father
}

check noSelfFather

fun grandpas(p: Person) : set Person {
 p.(mother + father).father
}

pred ownGrandpa(p: Person) {
 p in grandpas(p)
}

run ownGrandpa for 4 Person
```

首先有一个抽象类 person，它里边有两个参数，father 和 mother，并且这两个参数分别是 Man 类型和 Woman 类型，lone 表示最多 1 个，也就是说一个人最多有一个父亲和一个母亲。然后是分别定义了 Man 和 Woman，这两个对象都是 person 的子类，属性就不解释了，和上边一样。

然后已知条件 fact：第一个意思是说没有人自己是自己的父母。第二个意思是妻子和丈夫是转置关系，也就是妻子的另一半是丈夫，丈夫的另一半是妻子。

下一个是假设不存在一个男人，并且这个男人自己是自己的父亲。然后 check。定义了一个辅助函数 grandpas(p: Person)：set Person 参数是一个人，set 表明返回一个人的集合。p.(mother + father).father 表明为返回的集合内容是这个人 p 的（父亲和母亲）的父亲，也就是说返回的是他的 grandpas 的集合，他的 grandpas 是他父母的父亲。所以集合中的元素最多两个，最少 0 个。

然后定义了一个谓词 ownGrandpa(p: Person) ，意思是参数为一个人：p in grandpas(p)表明 p 是 grandpas(p)这个集合里的一个子集（这里觉得应该是说自己是自己的祖父）最后一个 run，run 和 check 的区别是，run 找的是正确的，check 找的是错误的，所以也就是找一个例子，看存不存在自己是自己的祖父这种情况。



以上所有内容仅供参考，纯属面向考试的个人理解，如有问题，欢迎指出。