**深度优先搜索：回溯法**

回溯法指从初始状态出发，按照深度优先搜索的方式，根据产生子节点的条件约束，搜索问题的解，当发现当前节点不满足求解条件时，就回溯，尝试其他路径。

**回溯法是一种“能进则进，进不了则换，换不了则退”的搜索方法。**

**1. 算法要素**

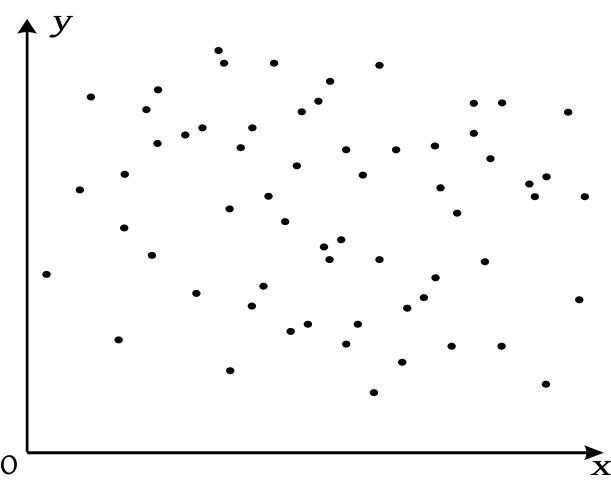
用回溯法解决实际问题时，**首先要确定解的组织形式，定义问题的解空间**。

**1）解空间解的组织形式：**回溯法的解的组织形式可以被规范为一个n元组{x1,x2,…,xn}，例如对3个物品的0-1背包问题，解的组织形式为{x1,x2,x3}。

**显约束：对解分量的取值范围的限定**。

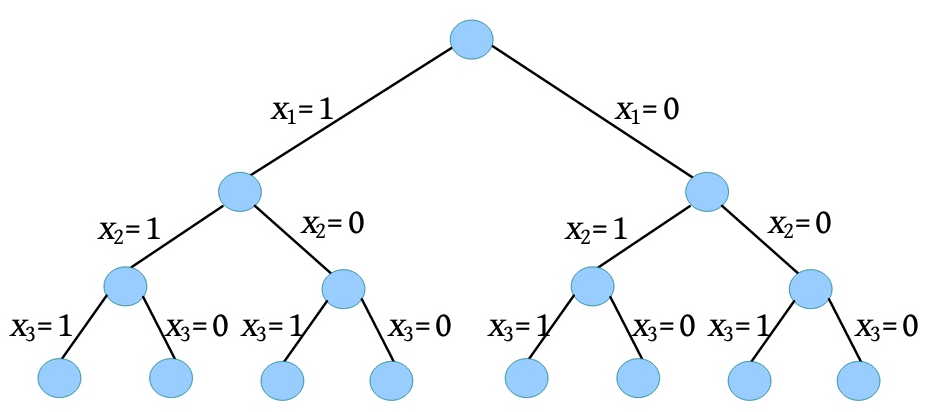
比如有3个物品的0-1背包问题，解的组织形式为{x1,x2,x3}。它的解分量xi的取值范围很简单，xi=0或者xi=1。xi=0表示将第i个物品不放入背包，xi=1表示将第i个物品放入背包，因此xi∈{0,1}。3个物品的0-1背包问题，其所有可能解是{0,0,0}、{0,0,1}、{0,1,0}、{0,1,1}、{1,0,0}、{1,0,1}、{1,1,0}、{1,1,1}。

**解空间：**顾名思义，就是由所有可能解组成的空间。二维解空间如下图所示。假设图中的每一个点都有可能是我们要的解，这些可能解就组成了解空间，而我们需要根据问题的约束条件，在解空间中寻找最优解。**解空间越小，搜索效率越高。解空间越大，搜索效率越低**。这犹如大海捞针，在大海里捞针相当困难，如果把解空间缩小到一平方米的海底就容易得多了。



**2）解空间的组织结构**

一个问题的解空间通常由很多可能解组成，不能像无头苍蝇一样乱飞乱撞去寻找最优解，盲目搜索的效率太低了，需要按照一定的套路即一定的组织结构搜索最优解。如果把这种组织结构用树形象地表达出来，就是**解空间树**。例如对3个物品的0-1背包问题，解空间树如下图所示。



解空间树只是解空间的形象表示，有利于解题时对搜索过程有直观理解，并不是真的要生成一棵树。有了解空间树，不管是写代码还是手工搜索求解，都能看得非常清楚，更能直接看到整个搜索空间的大小。

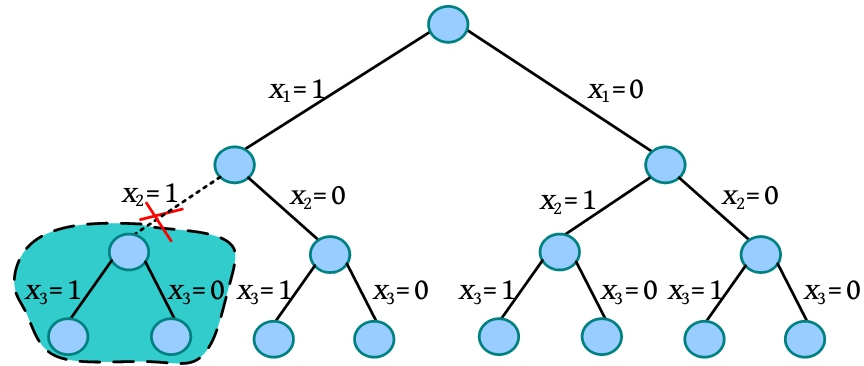
**3）搜索解空间**

**隐约束：指对能否得到问题的可行解或最优解做出的约束。**

如果**不满足隐约束**，就说明得不到问题的可行解或最优解，**就没必要再沿着该节点的分支进行搜索**了，**相当于把这个分支剪掉**了。

因此**隐约束也被称为剪枝函数**，实质上不是剪掉该分支，而是不再搜索该分支。

例如对3个物品的0-1背包问题，如果将前两个物品放入(x1=1,x2=1)后，背包超重了，就没必要再考虑是否将第3个物品放入背包的问题，如下图所示。即对圈中的分支不再搜索了，相当于剪枝了。



**隐约束（剪枝函数）包括约束函数和限界函数**。

判断能否得到**可行解**的函数被称为**约束函数**，判断能否得到**最优解**的函数被称为**限界函数**。有了剪枝函数，就可以剪掉得不到可行解或最优解的分支，避免无效搜索，提高搜索效率。**剪枝函数设计得好，搜索效率就高**。

**解空间的大小**和**剪枝函数**的好坏都直接影响搜索效率。因此这两项是**搜索算法的关键**。

在搜索解空间时，有以下几个术语需要说明。

• **扩展节点：**一个正在生成孩子的节点。

• **活节点：**一个自身已生成，但孩子还没有全部生成的节点。

• **死节点：**一个所有孩子都已经生成的节点。

• **子孙：**节点e的子树上所有节点都是e的子孙。

• **祖宗：**从节点e到树根路径上的所有节点都是e的祖宗。

**2. 回溯法解题过程**

**（1）定义解空间。**因为解空间的大小对搜索效率有很大的影响，因此使用回溯法时首先要定义合适的解空间，包括解的组织形式和显约束。

• **解的组织形式：**将解的组织形式都规范为一个n元组{x1,x2,…,xn}，只是对具体问题表达的含义不同而已。

**• 显约束：**显约束是对解分量的取值范围的限定，可以控制解空间的大小。

**（2）确定解空间的组织结构。**解空间的组织结构通常以**解空间树**形象地表达，根据解空间树的不同，解空间分为子集树、排列树、m叉树等。

**（3）搜索解空间。**按照**深度优先搜索策略**，**根据隐约束（约束函数和限界函数）**，在解空间中搜索问题的**可行解或最优解**。当发现当前节点不满足求解条件时，就回溯，尝试其他路径。如果问题只是**求可行解**，则只需**设定约束函数**即可，如果要**求最优解**，则需要**设定约束函数和限界函数**。**解的组织形式**都是通用的**n元组形式**，是解空间的形象表达。解空间和隐约束是控制搜索效率的关键。显约束可以控制解空间的大小，约束函数决定剪枝的效率，限界函数决定是否得到最优解。所以**回溯法解题的关键**是设计有效的**显约束和隐约束**。

**子集树**

假设有**n个物品和1个背包**，每个物品i对应的价值都为vi，重量都为wi，背包的容量为W。每个物品只有一件，**要么装入，要么不装入，不可拆分。**如何选取物品装入背包，使背包所装入的物品的总价值最大？要求输出最优值（装入物品的最大价值）和最优解（装入了哪些物品）。

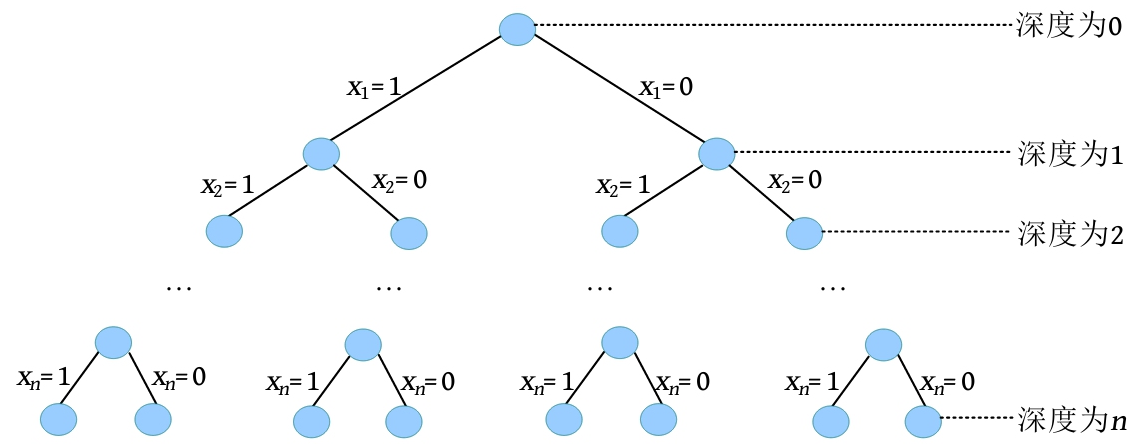
**1. 问题分析**

从n个物品中选择一些物品，**相当于从n个物品组成的集合S中找到一个子集**，这个子集内所有物品的总重量都不超过背包容量，并且这些物品的总价值最大。S的所有子集都是问题的可能解，这些可能解组成解空间。我们在解空间中找总重量不超过背包容量且价值最大的物品集作为最优解。**这些由问题的子集组成的解空间，其解空间树被称为子集树**。

**2. 算法设计**

**（1）定义问题的解空间。**该问题属于典型的0-1背包问题，问题的解是从n个物品中选择一些物品，使其在不超过容量的情况下价值最大。每个物品都有且只有两种状态：要么装入背包，要么不装入背包。那么第i个物品被装入背包能够达到目标，还是不被装入背包能够达到目标呢？可以用变量xi表示第i种物品是否被装入背包，“0”表示不被装入背包，“1”表示被装入背包，则xi的取值为0或1。第i（i=1,2,…,n）个物品被装入背包，xi=1；不被装入背包，xi=0。**该问题解的形式是一个n元组，且每个分量的取值都为0或1**。由此可得，问题的**解空间为{x1,x2,…,xi,…,xn}，**其中**显约束xi =0或1（i=1,2,…,n）**。

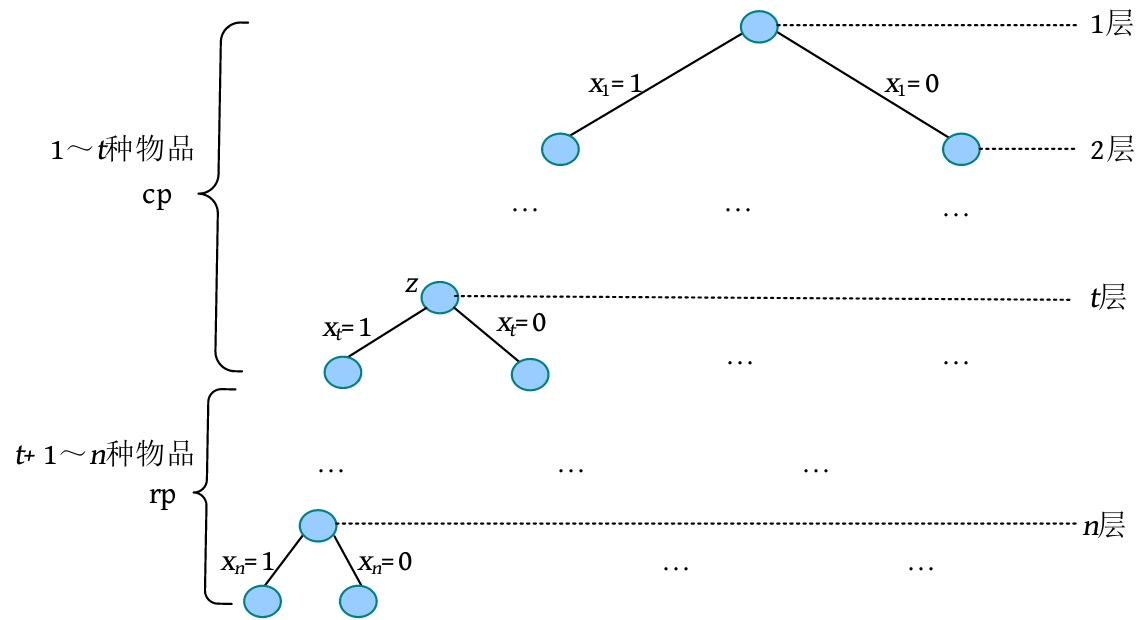
**（2）确定解空间的组织结构。**问题的解空间描述了2n种可能解，也可以说是由n个元素组成的集合的所有子集的个数。比如3个物品的背包问题，解空间是{0,0,0}、{0,0,1}、{0,1,0}、{0,1,1}、{1,0,0}、{1,0,1}、{1,1,0}、{1,1,1}。该问题有23个可能解。可见，问题的**解空间树为子集树**，**解空间树的深度为问题的规模n**，如下图所示。



**（3）搜索解空间。**根据解空间的组织结构，对于任何一个中间节点z（中间状态），从根节点到z节点的分支所代表的状态（是否装入背包）已经确定，从z到其子孙节点的分支的状态待确定。也就是说，如果z在解空间树中所处的层次是t，则说明第1种物品到第t-1种物品的状态已经确定了，只需沿着z的分支扩展来确定第t种物品的状态，那么前t种物品的状态就确定了。在前t种物品的状态确定后，对当前已被装入背包的物品的总重量用cw表示，对总价值用cp表示。

**• 约束条件。**判断第i个物品被装入背包后总重量是否超出背包容量，如果超出，则为不可行解；否则为可行解。约束条件为**cw+w[i]≤W**。其中，w[i]为第i个物品的重量，W为背包容量。

**• 限界条件。**已被装入物品的价值高不一定就是最优的，因为还有剩余物品未确定。目前还不确定第t+1种物品到第n种物品的实际状态，因此**只能用估计值**。假设第t+1种物品到第n种物品都被装入背包，对第t+1种物品到第n种物品的总价值用rp来表示。因此cp+rp是所有从根出发经过中间节点z的可行解的价值上界，如下图所示。



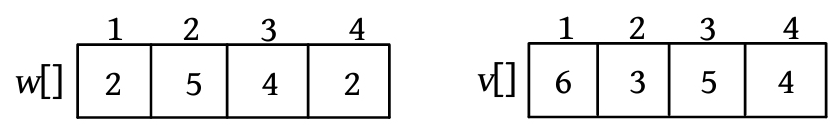
如果价值上界小于或等于当前搜索到的最优值（最优值用bestp表示，初始值为0），则说明从中间节点z继续向子孙节点搜索不可能得到一个比当前更优的可行解，没有继续搜索的必要；反之，继续向z的子孙节点搜索。

**限界条件为：cp+rp>bestp。**

**搜索过程：**从根节点开始，**以深度优先的方式进行搜索**。根节点首先成为活节点，也是当前的扩展节点。由于在子集树中约定左分支上的值为“1”，因此沿着扩展节点的左分支扩展，则代表装入物品。此时需要判断能否装入该物品，即判断约束条件成立与否，如果成立，即生成左孩子，左孩子成为活节点，并且成为当前的扩展节点，继续向纵深节点扩展；如果不成立，则剪掉扩展节点的左分支，沿着其右分支扩展，右分支代表物品不被装入背包，肯定有可能导致可行解。但是沿着右分支扩展有没有可能得到最优解呢？这一点需要由限界条件来判断。如果限界条件满足，则说明有可能得到最优解，即生成右孩子，右孩子成为活节点，并成为当前的扩展节点，继续向纵深节点扩展；如果不满足限界条件，则剪掉扩展节点的右分支，向最近的祖宗活节点回溯，继续搜索。直到所有活节点都变成死节点，搜索结束。

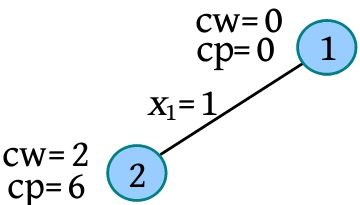
**3.图解**

假设现在有4个物品和1个背包，每个物品的重量w都为(2,5,4,2)，价值v都为(6,3,5,4)，背包的容量为10（W=10）。求在不超过背包容量的前提下把哪些物品放入背包，才能获得最大价值。



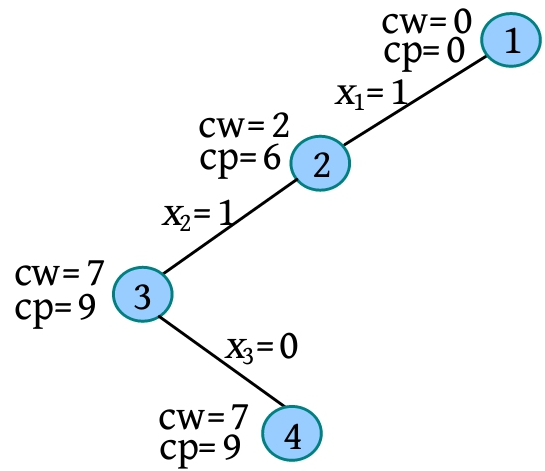
**（1）初始化。**sumw和sumv分别用来统计所有物品的总重量和总价值。sumw=13，sumv=18，如果sumw≤W，则说明可以全部装入，最优值为sumv。如果sumw>W，则不能全部装入，需要通过搜索求解。初始化当前放入背包的物品重量cw=0；当前放入背包的物品价值cp=0；当前最优值bestp=0。

**（2）搜索第1层（t=1）。**扩展节点1，判断cw+w[1]=2<W，满足约束条件，扩展左分支，令x[1]=1，cw=cw+w[1]=2，cp=cp+v[1]=6，生成节点2，如下图所示。



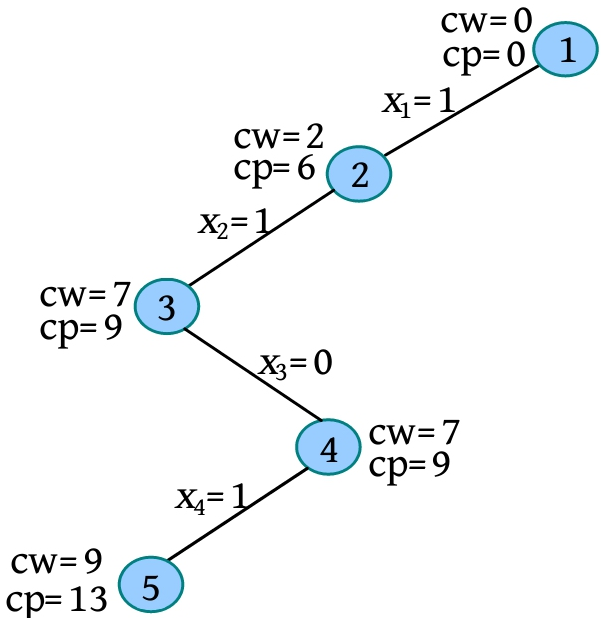
**（3）扩展节点2（t=2）。**判断cw+w[2]=7<W，满足约束条件，扩展左分支，令x[2]=1，cw=cw+w[2]=7，cp=cp+v[2]=9，生成节点3。

**（4）扩展节点3（t=3）。**判断cw+w[3]=11>W，超过了背包容量，第3个物品不能被放入。那么判断bound(t+1)是否大于bestp。bound(4)中剩余的物品只有第4个，rp=4，cp+rp=13，bestp=0，满足限界条件，扩展右子树。令x[3]=0，生成节点4，如下图所示。

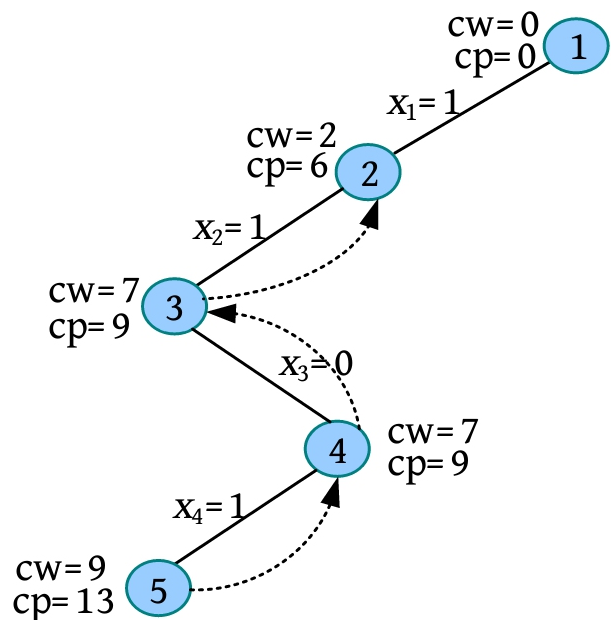


（5）扩展节点4（t=4）。判断cw+w[4]=9<W，满足约束条件，扩展左分支，令x[4]=1，cw=cw+w[4]=9，cp=cp+v[4]=13，生成节点5。

（6）扩展节点5（t=5）。t>n，找到一个当前最优解，用bestx[]保存当前最优解{1,1,0,1}，保存当前最优值bestp=cp=13，节点5成为死节点，如下图所示。



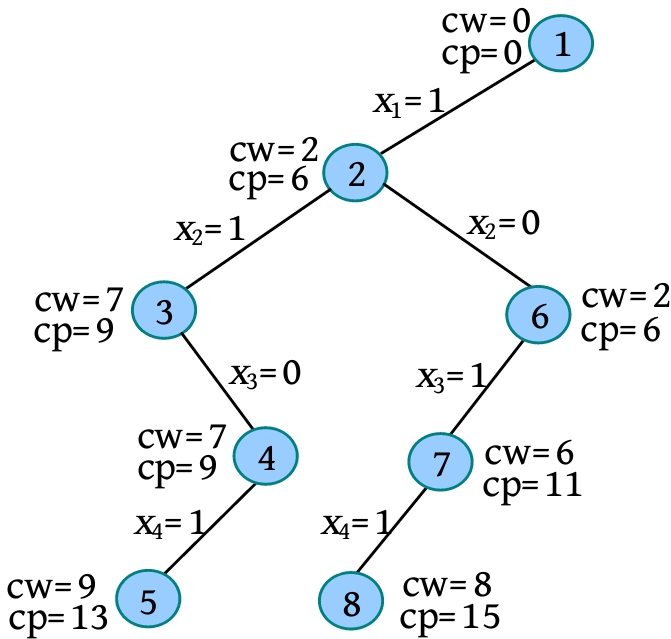
（7）回溯到节点4（t=4），一直向上回溯到节点2。向上回溯到节点4，回溯时cw=cw−w[4]=7，cp=cp−v[4]=9。怎么加上的，怎么减回去。节点4的右子树还未生成，考查bound(t+1)是否大于bestp，在bound(5)中没有剩余的物品，rp=0，cp+rp=9，bestp=13，因此不满足限界条件，不再扩展节点4的右子树。节点4成为死节点。向上回溯，回溯到节点3，节点3的左右孩子均已被考查过，是死节点，继续向上回溯到节点2。回溯时，cw=cw−w[2]=2，cp=cp−v[2]=6。怎么加上的，怎么减回去，如下图所示。



（8）扩展节点2（t=2）。节点2的右子树还未生成，考查bound(t+1)是否大于bestp，bound(3)中剩余的物品为第3、4个，rp=9，cp+rp=15，bestp=13，因此满足限界条件，扩展右子树。令x[2]=0，生成6号节点。

（9）扩展节点6（t=3）。判断cw+w[3]=6<W，满足约束条件，扩展左分支，令x[3]=1，cw=cw+w[3]=6，cp=cp+v[3]=11，生成节点7。

（10）扩展节点7（t=4）。判断cw+w[4]=8<W，满足约束条件，扩展左分支，令x[4]=1，cw=cw+w[4]=8，cp=cp+v[4]=15，生成节点8，如下图所示。

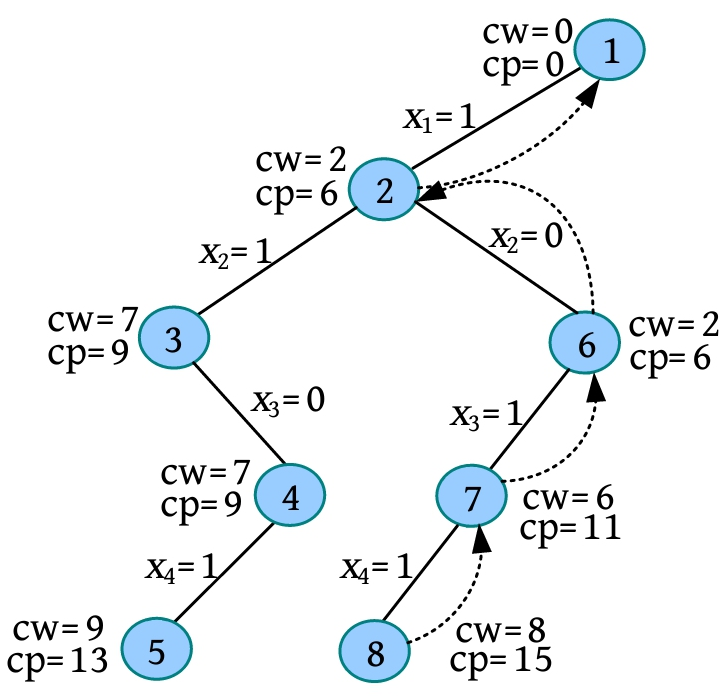


（11）扩展节点8（t=5）。t>n，找到一个当前最优解，用bestx[]保存当前最优解{1,0,1,1}，保存当前最优值bestp=cp=15，节点8成为死节点。向上回溯到节点7，回溯时cw=cw−w[4]=6，cp=cp−v[4]=11。怎么加上的，怎么减回去。

（12）扩展节点7（t=4）。节点7的右子树还未生成，考查bound(t+1)是否大于bestp，在bound(5)中没有剩余的物品，rp=0，cp+rp=11，bestp=15，因此不满足限界条件，不再扩展节点7的右子树。节点7成为死节点。向上回溯，回溯到节点6，回溯时，cw=cw−w[3]=2，cp=cp−v[3]=6，怎么加上的，怎么减回去。

（13）扩展节点6（t=3）。节点6的右子树还未生成，考查bound(t+1)是否大于bestp，bound(4)中剩余的物品是第4个，rp=4，cp+rp=10，bestp=15，因此不满足限界条件，不再扩展节点6的右子树。节点6成为死节点。向上回溯，回溯到节点2，节点2的左右孩子均已被考查过，是死节点，继续向上回溯到节点1。回溯时，cw=cw−w[1]=0，cp=cp−v[1]=0。怎么加上的，怎么减回去。

（14）扩展节点1（t=1）。节点1的右子树还未生成，考查bound(t+1)是否大于bestp，bound(2)中剩余的物品是第2、3、4个，rp=12，cp+rp=12，bestp=15，因此不满足限界条件，不再扩展节点1的右子树，节点1成为死节点。所有节点都是死节点，算法结束。

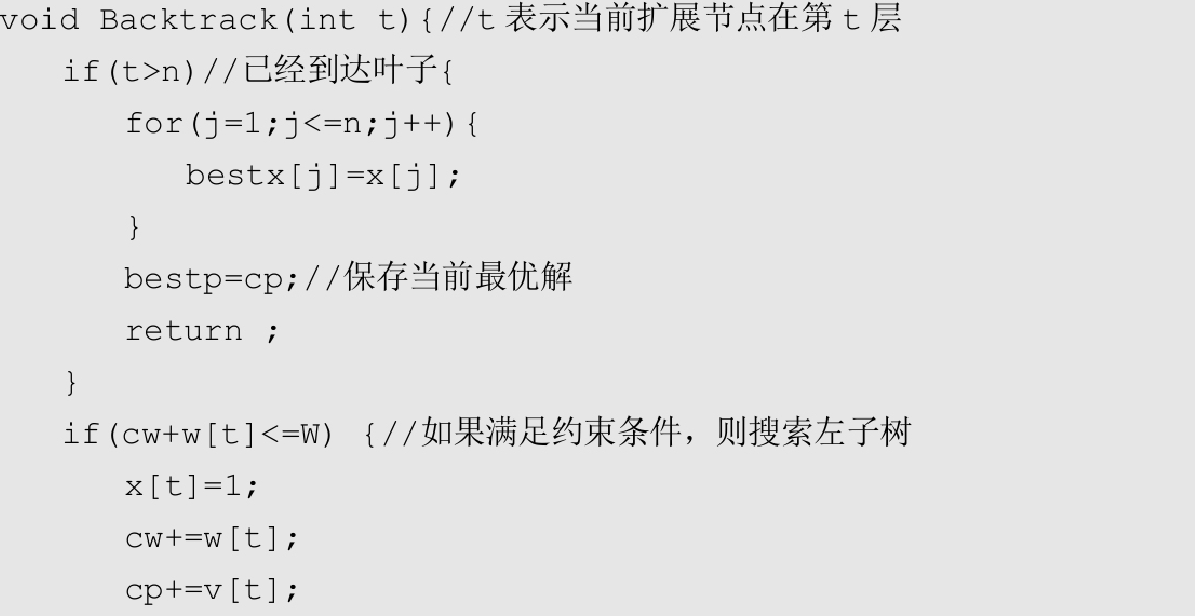


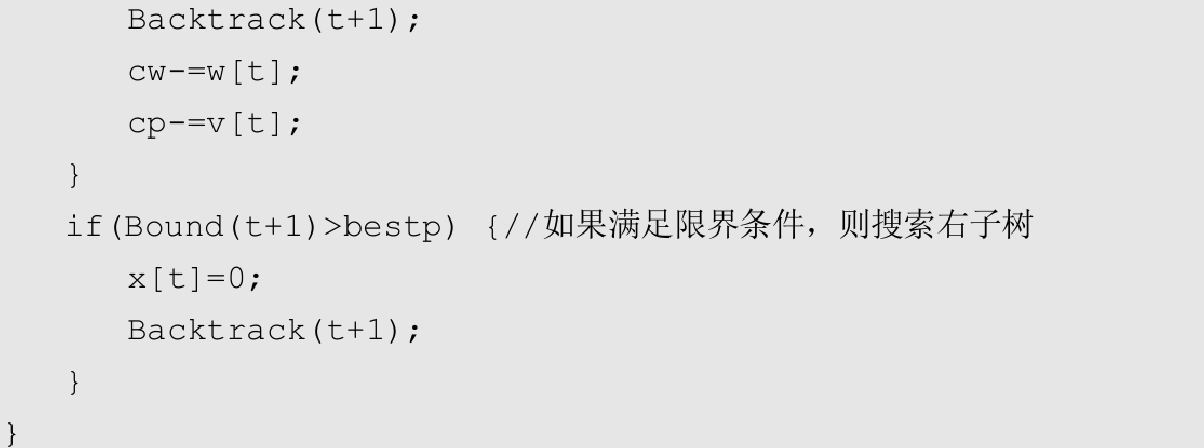
**4. 算法实现**

**（1）计算上界。**计算上界指计算已装入物品的价值cp及剩余的物品价值的总价值rp。我们已经知道已被装入背包的物品价值cp，对剩余的物品不确定要装入哪些，按照假设都被装入的情况计算，即按最大值计算（剩余的物品的总价值），因此得到的值是可装入物品价值的上界。



**（2）按约束条件和限界条件搜索求解。**t表示当前扩展节点在第t层，cw表示当前已被放入物品的重量，cp表示当前已被放入物品的价值。如果t>n，则表示已经到达叶子，记录最优值的最优解，返回；否则，判断是否满足约束条件，满足则搜索左子树。因为左子树表示放入该物品，所以令x[t]=1，表示放入第t个该物品。cw+=w[t]，表示当前已被放入物品的重量增加w[t]。cp+=v[t]，表示当前已被放入物品的价值增加v[t]。Backtrack(t+1)表示递推，深度优先搜索第t+1层。回归时即向上回溯时，把增加的值减去，cw−=w[t]，cp−=v[t]。判断是否满足限界条件，满足则搜索右子树。因为右子树表示不放入该物品，所以令x[t]=0，当前已被放入物品的重量、价值均不改变。Backtrack(t+1)表示深度优先搜索第t+1层。

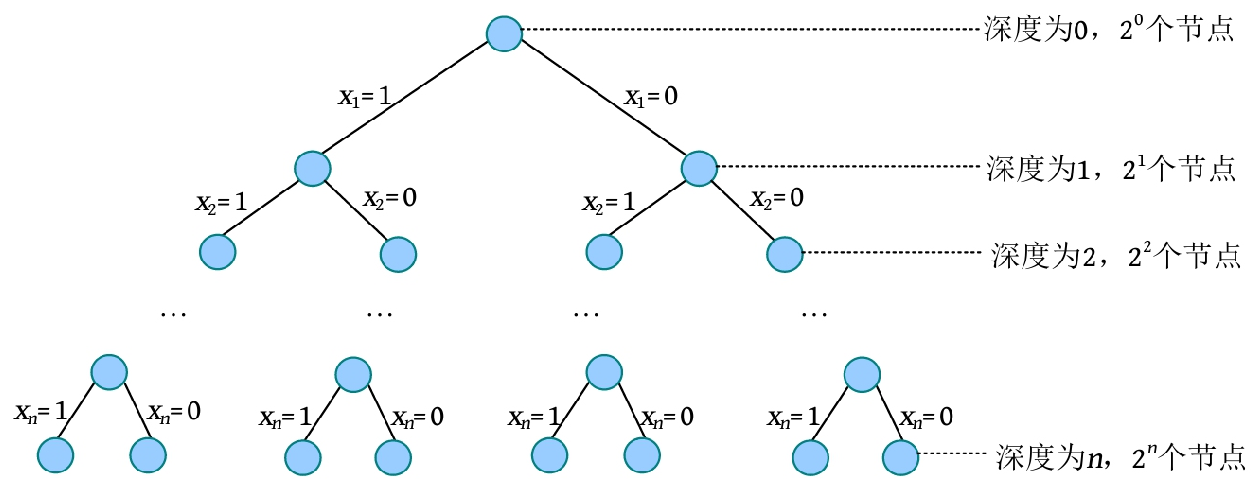




**5. 算法分析**

**时间复杂度：**回溯法的运行时间取决于它在搜索过程中生成的节点数。而限界函数可以大大减少所生成的节点个数，避免无效搜索，加快搜索速度。

左孩子需要判断约束函数，右孩子需要判断限界函数，那么在最坏情况下有多少个左孩子和右孩子呢？规模为n的子集树在最坏情况下的状态如下图所示。



总的节点个数为20+21+…+2n=2n+1-1，减去树根再除以2，就得到了左右孩子的个数，左右孩子的个数为(2n+1-1-1)/2=2n-1。

**约束函数的时间复杂度为O(1)，限界函数的时间复杂度为O(n)**。在最坏情况下有O(2n)个左孩子调用约束函数，有O(2n)个右孩子调用限界函数，所以采用回溯法解决背包问题的时间复杂度为O(1×2n+n×2n)=O(n×2n)。

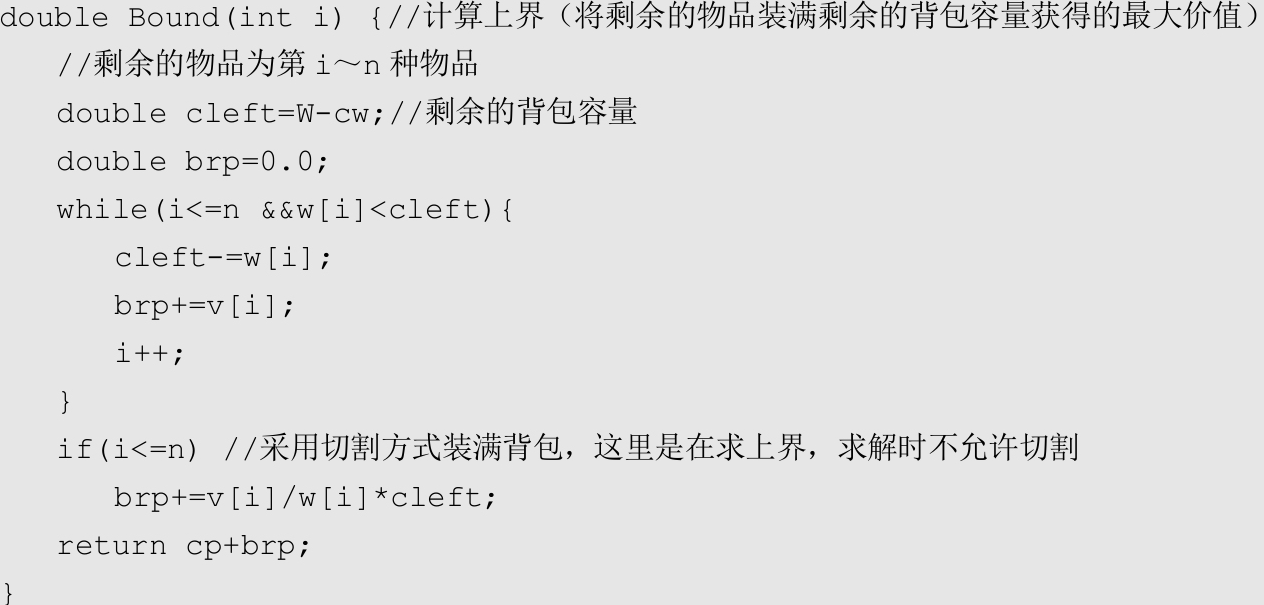
**空间复杂度：** 在搜索过程中的任何时刻，仅保留从开始节点到当前扩展节点的路径，从开始节点到叶子最长的路径长度为n。在程序中使用bestx[]数组记录该最长路径作为最优解，所以该算法的空间复杂度为O(n)。

**6. 算法优化拓展**

在上面的程序中，上界函数是当前价值cp加剩余物品的总价值rp，这个估值过高，因为剩余物品的重量很有可能是超过背包容量的。可以缩小上界，加快剪枝速度，提高搜索效率。

上界函数Bound()：当前价值cp+剩余容量可容纳的剩余物品的最大价值brp。

为了更好地计算和运用上界函数剪枝，先将物品按照其单位重量价值（价值/重量）从大到小排序，然后按照排序后的顺序考察各个物品。



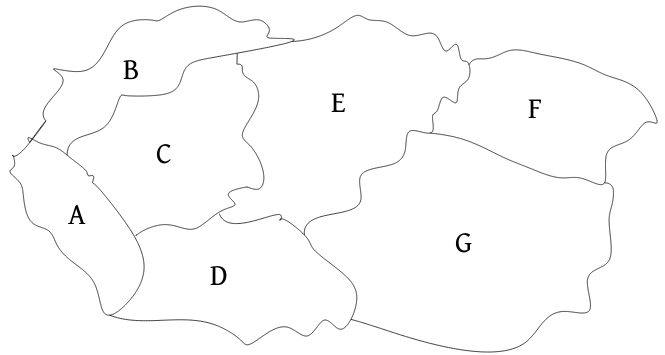
**时间复杂度：**约束函数的时间复杂度为O(1)，限界函数的时间复杂度为O(n)。在最坏情况下有O(2n)个左孩子需要调用约束函数，有O(2n)个右孩子需要调用限界函数，回溯算法Backtrack的时间复杂度为O(n2n)。排序函数的时间复杂度为O(nlogn)。这考虑了最坏情况，实际上，经过上界函数优化后，剪枝速度很快，根本不需要生成所有节点。

**空间复杂度：**这里除了记录了最优解数组，还使用了一个结构体数组用于排序，两个辅助数组传递排序后的结果，这些数组的规模都是n，因此空间复杂度仍是O(n)。

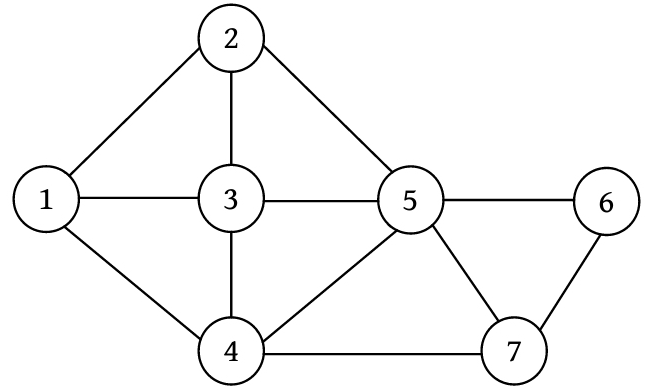
**m叉树**

**给定无向连通图G和m种颜色，找出所有不同的着色方案，使相邻的区域有不同的颜色**。如果把地图上的每一个区域都退化成一个点，将相邻的区域用线连接起来，地图就变成了一个无向连通图，给地图着色相当于给该无向连通图的每个点都着色，要求有连线的点不能有相同的颜色，这就是**图的m着色问题**。

以下图为例：



该地图共有7个区域，分别是A、B、C、D、E、F、G，按上面的顺序编号1～7，对每个区域都用一个节点表示，相邻的区域有连线，地图就转化成了一个无向连通图，如下图所示。



如果用3种颜色给该地图着色，那么该问题中每个节点所着的颜色均有3种选择，7个节点所着的颜色号组合是一个可能解，例如{1,2,3,2,1,2,3}。

**每个节点都有m种选择，即在解空间树中每个节点都有m个分支，称之为m叉树**。

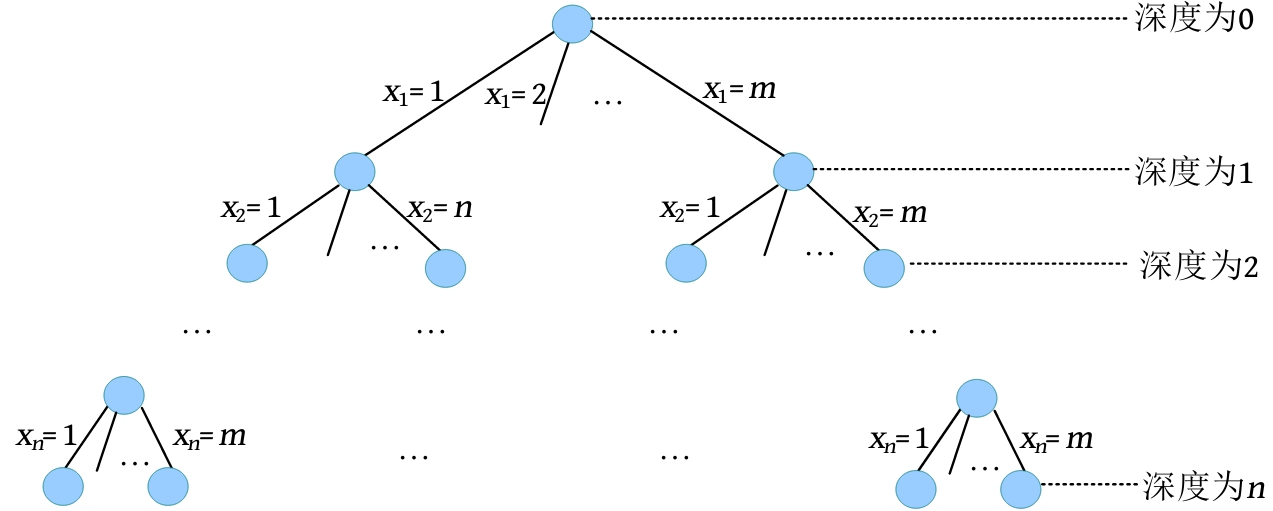
**1. 算法设计**

**1）定义问题的解空间**

定义问题的解空间及其组织结构是很容易的。图的m着色问题的解空间形式为**n元组{x1,x2,…,xi,…,xn}**，每一个分量的取值都为1,2,…,m，即问题的解是一个**n元向量**。由此可得，问题的解空间为{x1,x2,…，xi,…,xn}，其中**显约束xi=1,2,…,m（i=1,2,3,…,n）**。xi=2表示在图G中将第i个节点着色为2号色。

**2）确定解空间的组织结构**

问题的解空间组织结构是一棵**满m叉树**，树的**深度为n**，如下图所示。

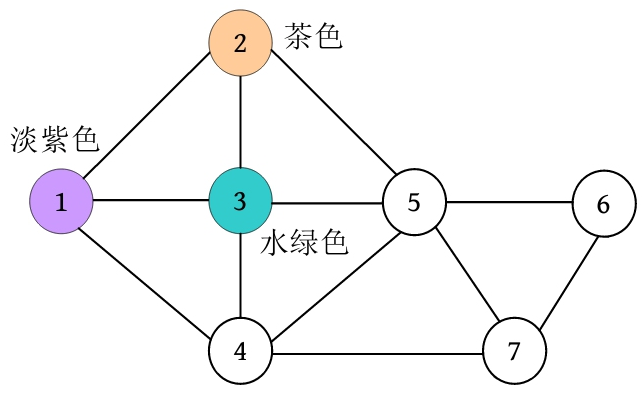


**3）搜索解空间**

**（1）约束条件。**假设当前扩展节点处于解空间树的第t层，那么从第1个节点到第t-1个节点的状态（着色的色号）已经确定。接下来沿着扩展节点的第1个分支进行扩展，此时**需要判断第t个节点的着色情况**。第t个节点的颜色**要与前t-1个节点中与其有边相连的节点颜色不同**，如果有颜色相同的，则第t个节点不能用这个色号，换下一个色号尝试，如下图所示。



例如，假设当前扩展节点z在第4层，则说明前3个节点的色号已经确定，如下图所示。



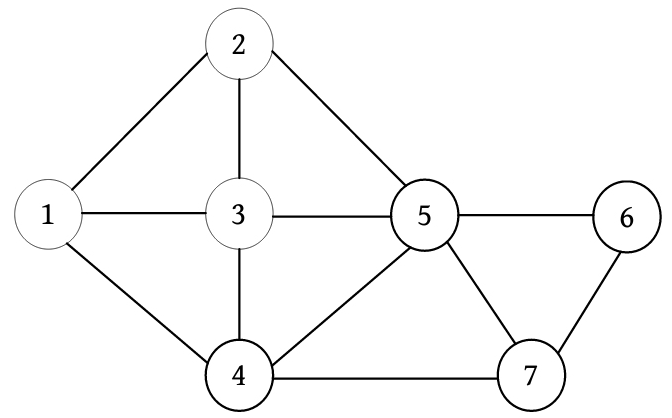
在前3个已着色的节点中，节点4与节点1、3有边相连，那么节点4的色号不可以与节点1、3的色号相同。

**（2）限界条件。**因为只找可行解就可以了，**不是求最优解**，因此**不需要限界条件**。

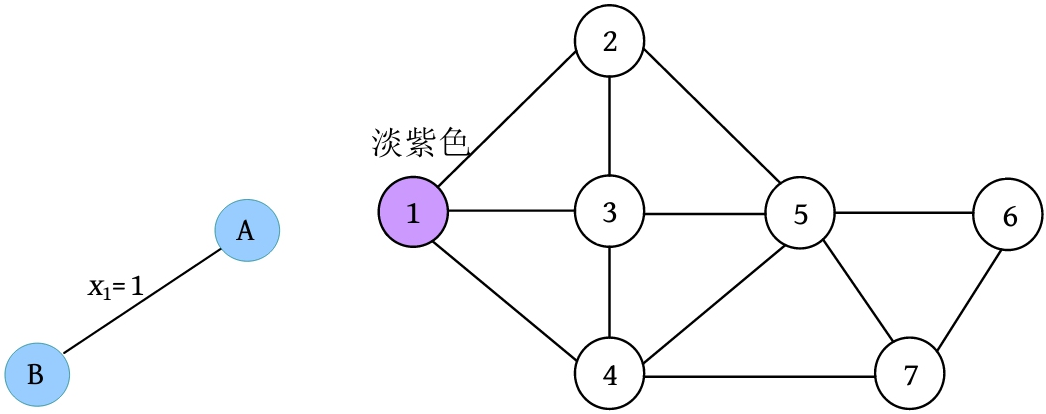
**（3）搜索过程。**扩展节点沿着第1个分支扩展，判断约束条件，如果满足，则进入深一层继续搜索；如果不满足，则扩展生成的节点被剪掉，换下一个色号尝试。如果所有的色号都尝试完毕，则该节点变成死节点，向上回溯到离其最近的活节点，继续搜索。搜索到叶子节点时，找到一种着色方案。搜索到全部活节点都变成死节点时为止。

**2. 图解**

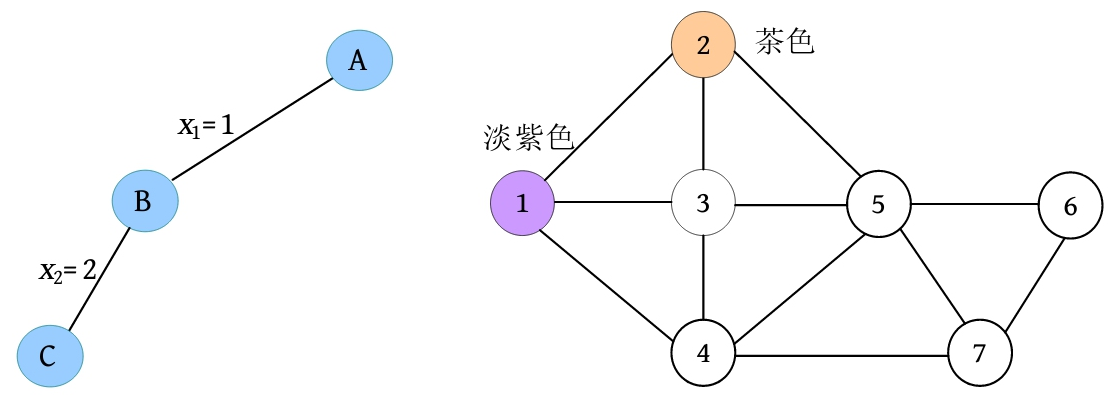
地图的７个区域转化成的无向连通图如下图所示。如果现在用3种颜色（淡紫、茶色、水绿色）给该地图着色，那么该问题中每个节点所着的颜色均有3种选择（m=3），7个节点所着的颜色组合是一个可能解。



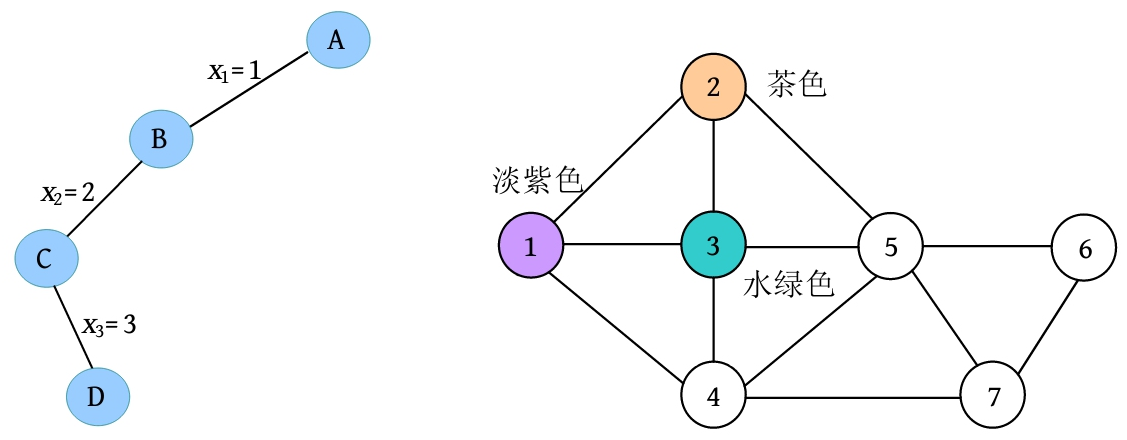
（1）开始搜索第1层（t=1）。扩展节点A的第1个分支，首先判断是否满足约束条件，因为之前还未着色任何节点，所以满足约束条件。然后扩展该分支，令节点1着1号色（淡紫），即x[1]=1，生成节点B。搜索过程和着色方案如下图所示。



（2）扩展节点B（t=2）。扩展第1个分支x[2]=1，首先判断节点2是否与前面已确定色号的节点（1号）有边相连且色号相同，不满足约束条件，剪掉该分支。然后沿着x[2]=2扩展，节点2与前面已确定色号的节点（1号）有边相连但色号不相同，满足约束条件，扩展该分支，令节点2着2号色（茶色），即x[2]=2，生成节点C。搜索过程和着色方案如下图所示。



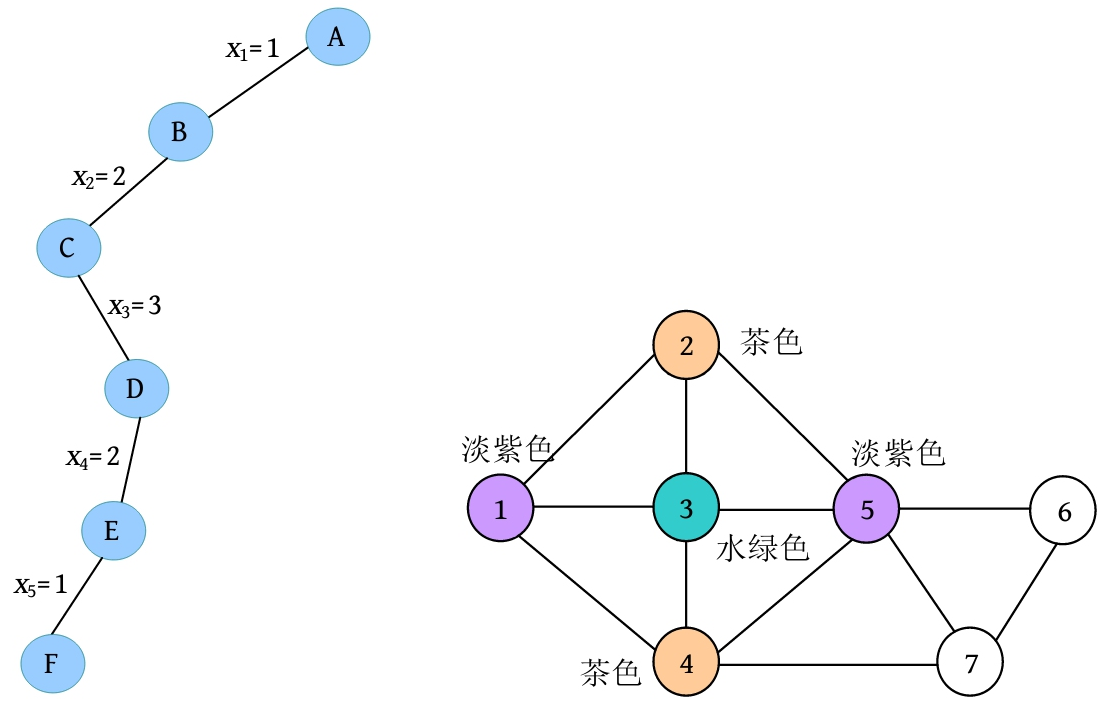
（3）扩展节点C（t=3）。扩展第1个分支x[3]=1，首先判断节点3是否与前面已确定的节点（1、2号）有边相连且色号相同，节点3与节点1有边相连且色号相同，不满足约束条件，剪掉该分支。然后沿着x[3]=2扩展，节点3与前面已确定色号的节点（节点2）有边相连且色号相同，不满足约束条件，剪掉该分支。然后沿着x[3]=3扩展，节点3与前面已确定色号的节点（节点1、2）有边相连且色号均不相同，满足约束条件，扩展该分支，令节点3着3号色（水绿色），即令x[3]=3。生成节点D。搜索过程和着色方案如下图所示。

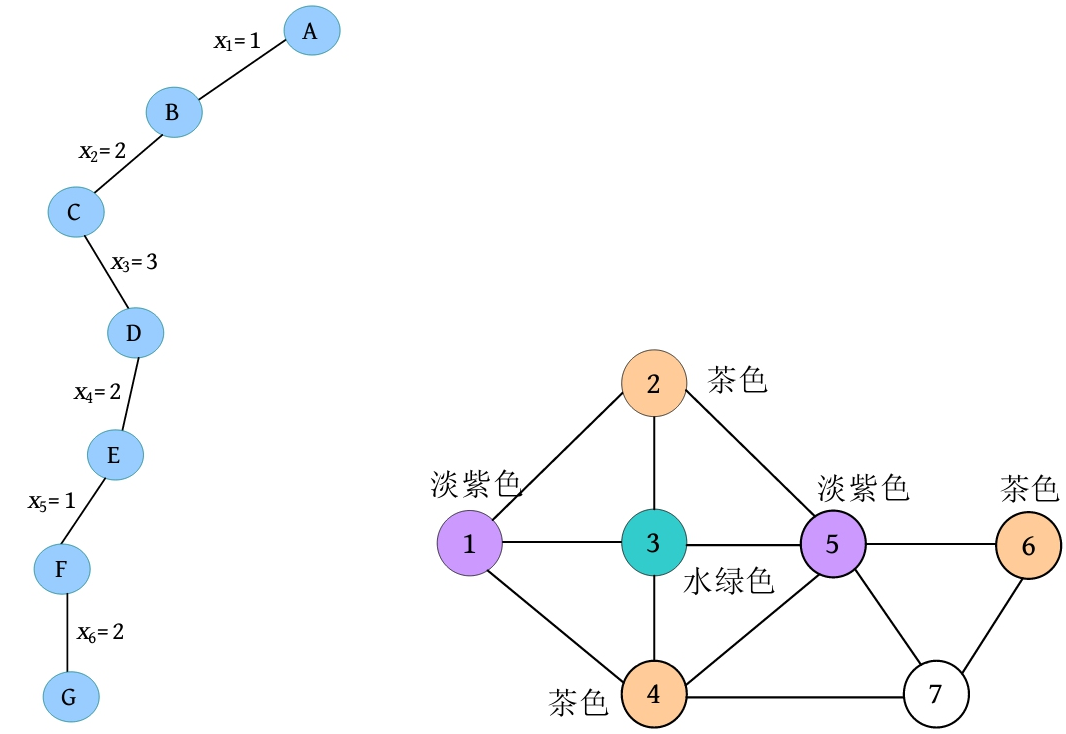


（4）扩展节点D（t=4）。扩展第1个分支x[4]=1，首先判断节点4是否与前面已确定的节点（节点1、2、3）有边相连且色号相同，节点4和节点1有边相连且色号相同，不满足约束条件，剪掉该分支。然后沿着x[4]=2扩展，节点4与前面已确定色号的节点（节点1、3）有边相连但色号均不同，满足约束条件，扩展该分支，令节点4着2号色（茶色），令x[4]=2。生成节点E。搜索过程和着色方案如下图所示。

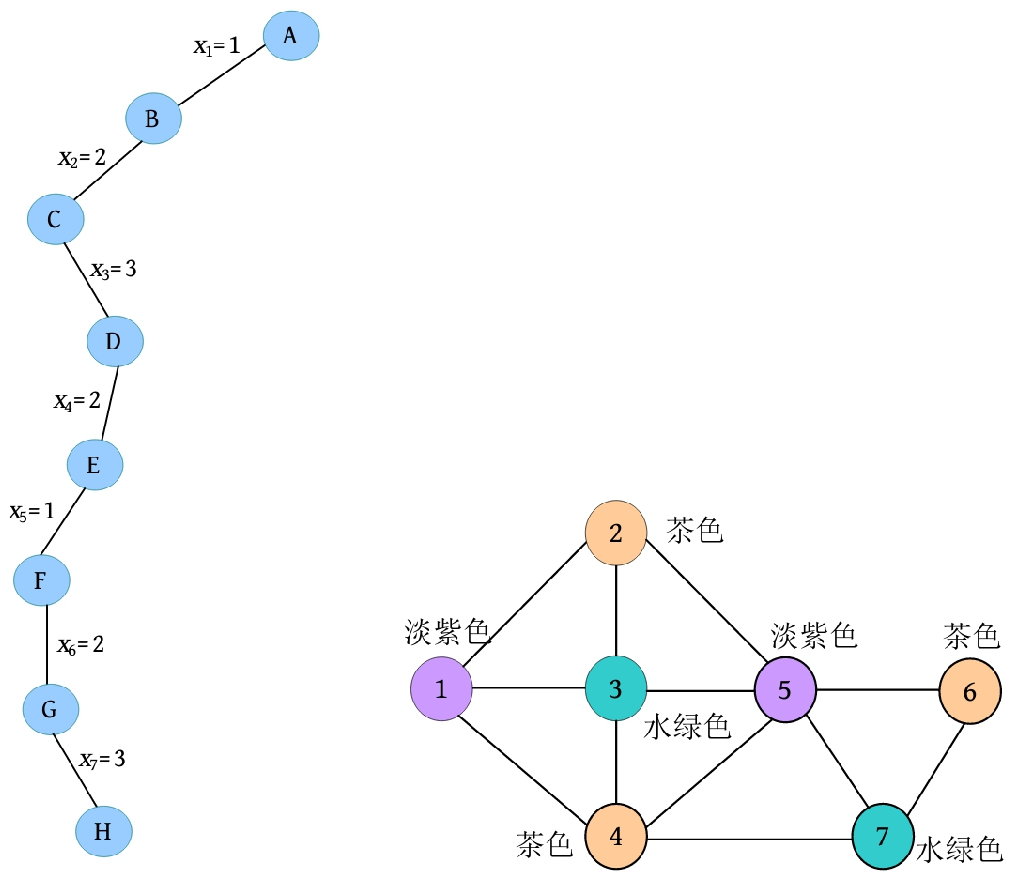


（5）扩展节点E（t=5）。扩展第1个分支x[5]=1，首先判断节点5是否与前面已确定的节点（节点1、2、3、4）有边相连且色号相同，节点5与前面已确定色号的节点（节点2、3、4）有边相连但色号均不同，满足约束条件，扩展该分支，令节点5着1号色（淡紫色），令x[5]=1，生成节点F。搜索过程和着色方案如下图所示。



（6）扩展节点F（t=6）。扩展第1个分支x[6]=1，首先判断节点6是否与前面已确定的节点（节点1、2、3、4、5）有边相连且色号相同，节点6与前面已确定色号的节点（节点5）有边相连且色号相同，不满足约束条件，剪掉该分支。然后沿着x[6]=2扩展，节点6和前面已确定色号的节点（节点5）有边相连但色号不同，满足约束条件，扩展该分支，令节点6着2号色（茶色），令x[6]=2。生成节点G。搜索过程和着色方案如下图所示。

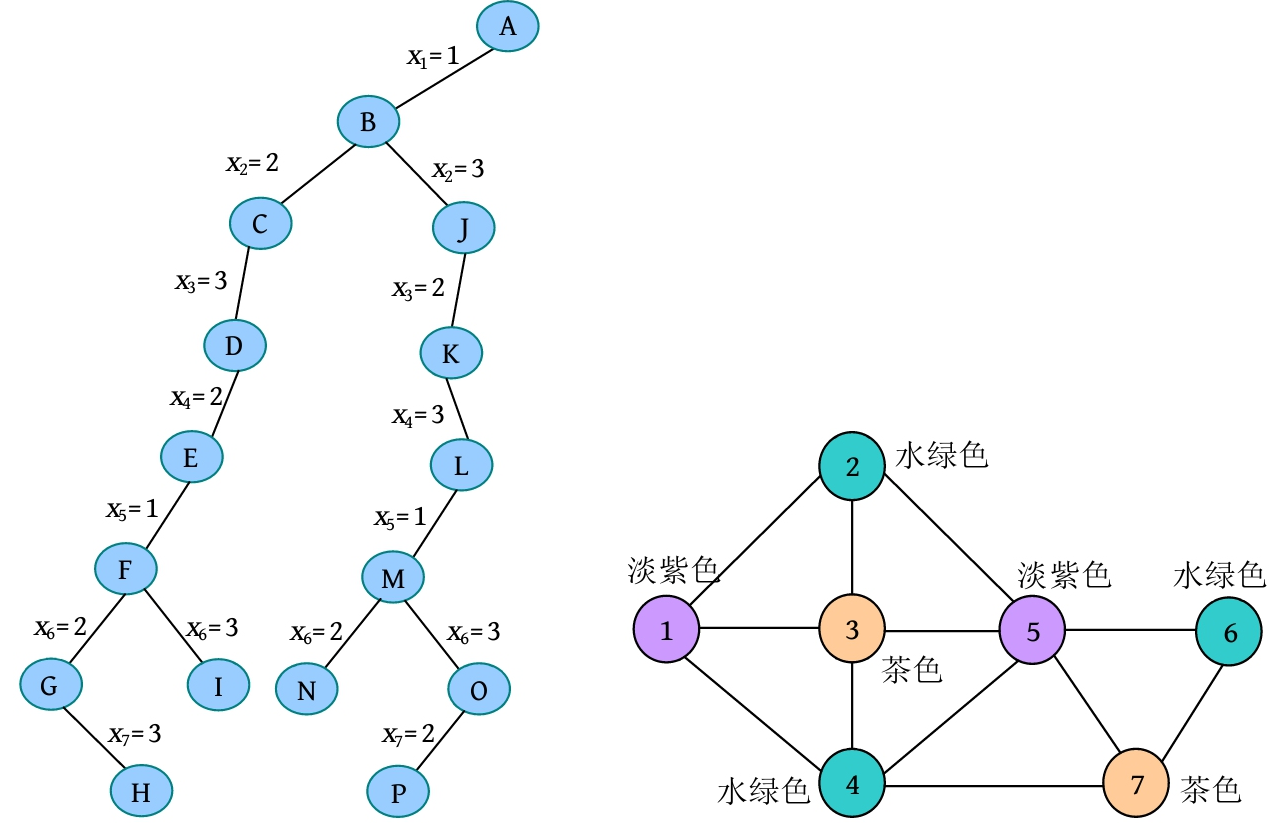
（7）扩展节点G（t=7）。扩展第1个分支x[7]=1，首先判断节点7是否与前面已确定的节点（节点1、2、3、4、5、6）有边相连且色号相同，节点7与前面已确定色号的节点（节点5）有边相连且色号相同，不满足约束条件，剪掉该分支。然后沿着x[7]=2扩展，节点7与前面已确定色号的节点（节点4、6）有边相连且色号相同，不满足约束条件，剪掉该分支。接着沿着x[7]=3扩展，节点7与前面已确定色号的节点（节点4、5、6）有边相连但色号不同，满足约束条件，扩展该分支，令节点7着3号色（水绿色），令x[7]=3。生成节点H。搜索过程和着色方案如下图所示。



（8）扩展节点H（t=8）。t>n，找到一个可行解，输出该可行解{1,2,3,2,1,2,3}。回溯到最近的活节点G。

（9）重新扩展节点G（t=7）。节点G的m（m=3）个孩子均已考察完毕，成为死节点。回溯到最近的活节点F。

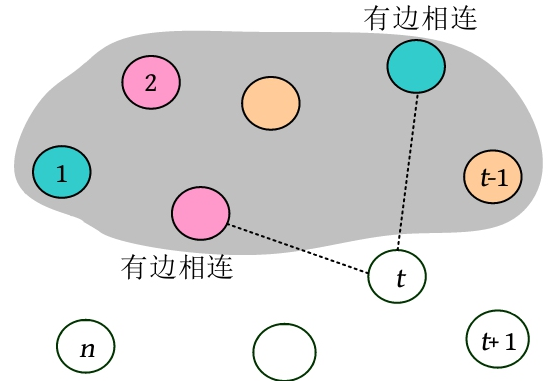
（10）继续搜索，又找到第2种着色方案，输出该可行解{1,3,2,3,1,3,2}。搜索过程和着色方案如下图所示。



（11）继续搜索，又找到4个可行解，分别是{2,1,3,1,2,1,3}、{2,3,1,3,2,3,1}、{3,1,2,1,3,1,2}、{3,2,1,2,3,2,1}。

**1. 算法实现**

**（1）约束函数。**假设当前扩展节点处于解空间树的第t层，那么从第1个节点到第t-1个节点的状态（着色的色号）已经确定。接下来沿着扩展节点的第1个分支进行扩展，此时需要判断第t个节点的着色情况。第t个节点的颜色号要与前t-1个节点中与其有边相连的节点颜色不同，如果有一个颜色相同的，则第t个节点不能用这个色号，换下一个色号尝试，如下图所示。



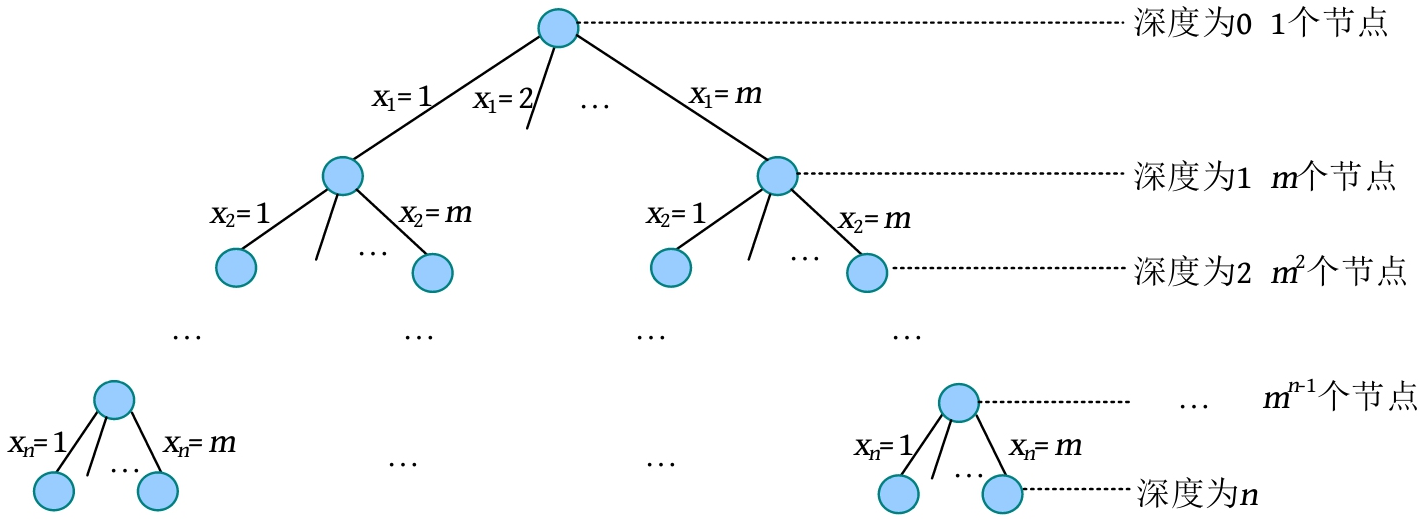


**（2）按约束条件搜索求解。**t表示当前扩展节点在第t层。如果t>n，则表示已经到达叶子，sum累计第几个着色方案，输出可行解。否则，扩展节点沿着第1个分支扩展，判断是否满足约束条件，如果满足，则进入深一层继续搜索；如果不满足，则扩展生成的节点被剪掉，换下一个色号尝试。如果所有色号都尝试完毕，则该节点变成死节点，向上回溯到离其最近的活节点，继续搜索。搜索到叶子时，找到一种着色方案。搜索到全部活节点都变成死节点为止。



**4. 算法分析**

**时间复杂度：**在最坏情况下，除了最后一层，有1+m+m2 +…+mn-1= (mn-1)/(m-1)≈mn-1个节点需要扩展，而这些节点每个都要扩展m个分支，总的分支个数为mn，每个分支都判断约束函数，判断约束条件需要O(n)时间，因此耗时O(nmn)。在叶子节点处输出可行解需要耗时O(n)，在最坏情况下会搜索到所有叶子，叶子个数为mn，故耗时O(nmn)。因此，时间复杂度为**O(nmn)**，如下图所示。

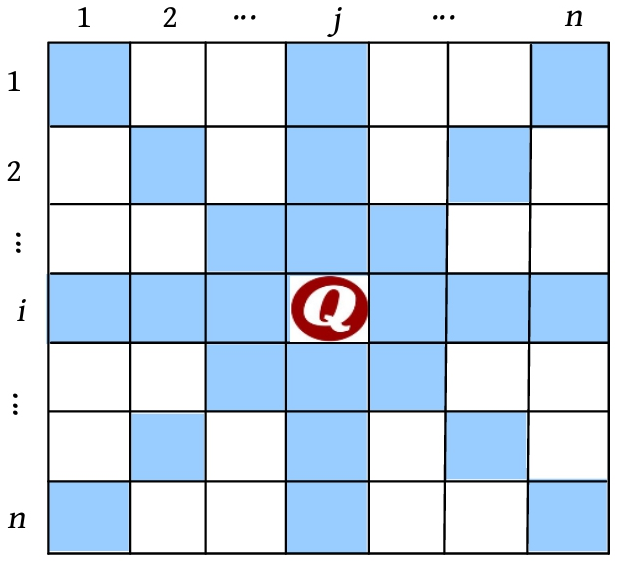


**空间复杂度：**回溯法的另一个重要特性就是在搜索执行的同时产生解空间。在搜索过程中的任何时刻，仅保留从开始节点到当前扩展节点的路径，从开始节点起最长的路径为n。在程序中使用x[]数组记录该最长路径作为可行解，所以该算法的空间复杂度为**O(n)**。

**排列树**

在n×n的棋盘上放置了彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之在同一行、同一列、同一斜线上的棋子。请在n×n的棋盘上放置n个皇后，使其彼此不受攻击。

如果棋盘如下图所示，我们在第i行第j列放置一个皇后，那么第i行的其他位置（同行）、第j列的其他位置（同列）、同一斜线上的其他位置，都不能再放置皇后。



不可能杂乱无章地尝试每个位置，需要有策略地求解，可以以行为主导。

（1）在第1行第1列放置第1个皇后。

（2）在第2行放置第2个皇后。第2个皇后的位置不能与第1个皇后同列、同斜线，不用再判断是否同行，因为每行只放置一个，本来就已经不同行。

（3）在第3行放置第3个皇后，第3个皇后的位置不能与前2个皇后同列、同斜线。

（4）……

（5）在第t行放置第t个皇后，第t个皇后的位置不能与前t-1个皇后同列、同斜线。

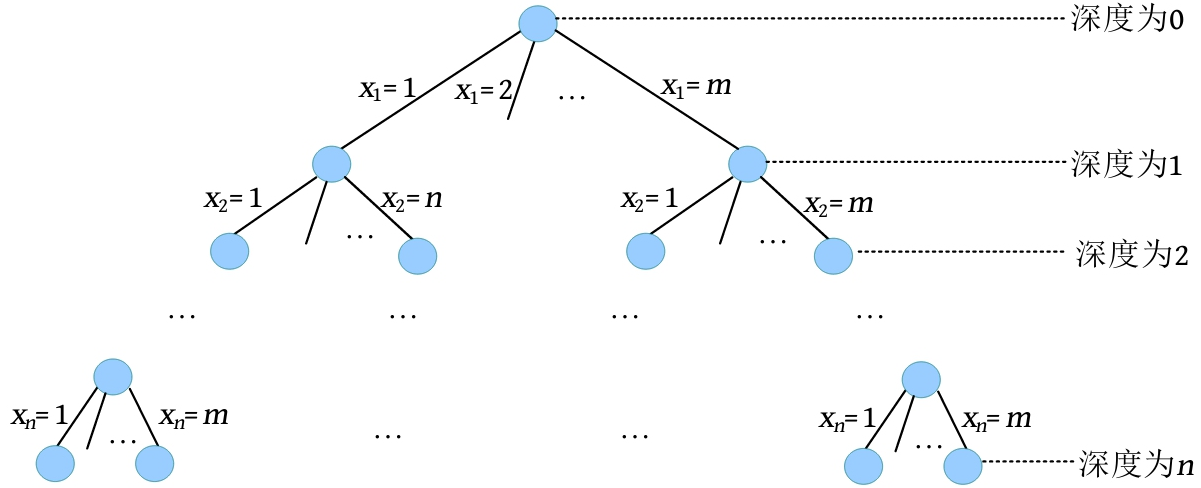
（6）……

（7）在第n行放置第n个皇后，第n个皇后的位置不能与前n-1个皇后同列、同斜线。

**1. 算法设计**

**（1）定义问题的解空间。**n皇后问题的解的形式为**n元组：{x1,x2,…,xi,…,xn}**，分量xi表示第i个皇后被放置在第i行第xi列，xi的取值为1,2,…,n。例如x2=5，表示第2个皇后被放置在第2行第5列。显约束为不同行。

**（2）解空间的组织结构。**n皇后问题的解空间是一棵**m（m=n）叉树**，树的**深度为n**，如下图所示。



**（3）搜索解空间。**

**• 约束条件。**在第t行放置第t个皇后时，第t个皇后的位置不能与前t-1个皇后同列、同斜线。第i个皇后与第j个皇后不同列，即**xi!=xj**，**并且不同斜线|i-j| != |xi-xj|。**

**• 限界条件。**该问题**不存在放置方案是否好坏**的情况，所以**不需要设置限界条件**。

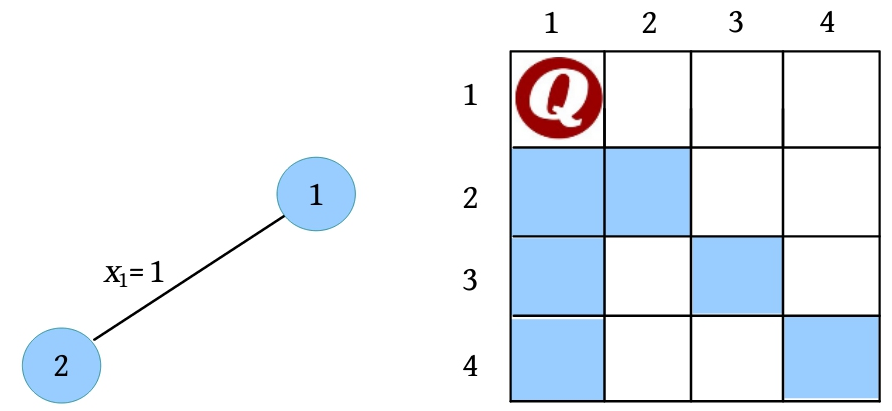
**• 搜索过程。**从根开始，以**深度优先搜索**的方式进行搜索。根节点是活节点并且是当前扩展节点。在搜索过程中，当前扩展节点沿纵深方向移向一个新节点，判断该新节点是否满足隐约束，如果满足，则新节点成为活节点，并且成为当前扩展节点，继续深一层的搜索，如果不满足，则换到该新节点的兄弟节点继续搜索；如果新节点没有兄弟节点，或其兄弟节点已全部搜索完毕，则扩展节点成为死节点，搜索回溯到其父节点处继续进行。搜索到问题的根节点变成死节点时为止。

**2. 图解**

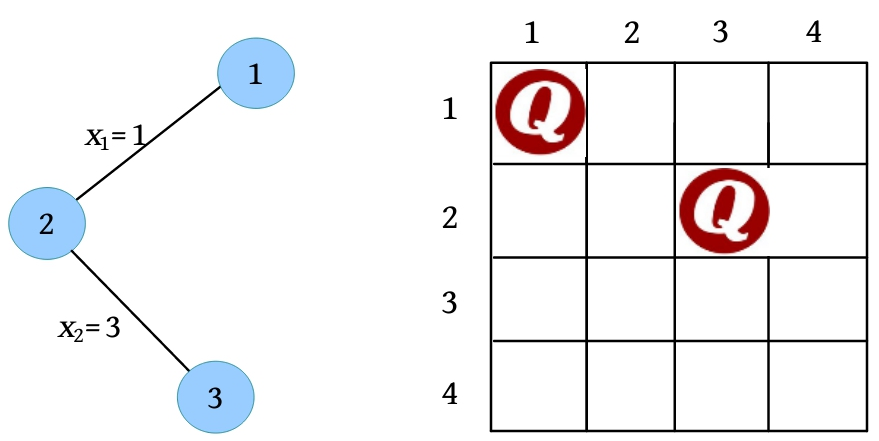
在4×4的棋盘上放置4个皇后，使其彼此不受攻击。



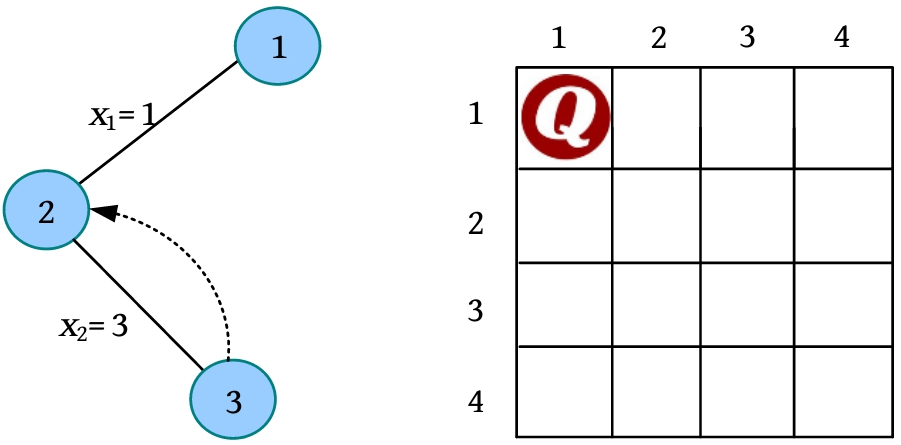
（1）开始搜索第1层（t=1）。扩展节点1，判断x1=1是否满足约束条件，因为之前还未选中任何节点，满足约束条件。令x[1]=1，生成节点2。



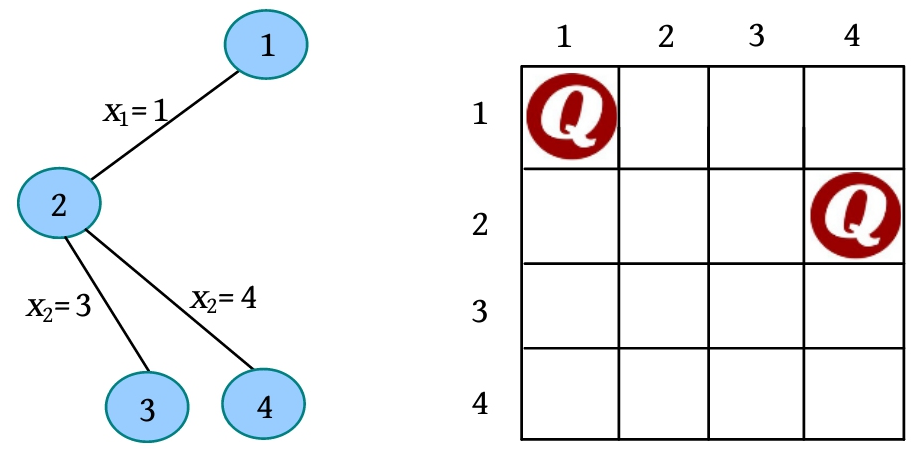
（2）扩展节点2（t=2）。判断x2=1不满足约束条件，因为与之前放置的第1个皇后同列；考察x2=2也不满足约束条件，因为与之前放置的第1个皇后同斜线；考察x2=3满足约束条件，因为与之前放置的皇后不同列、不同斜线，令x[2]=3，生成节点3。



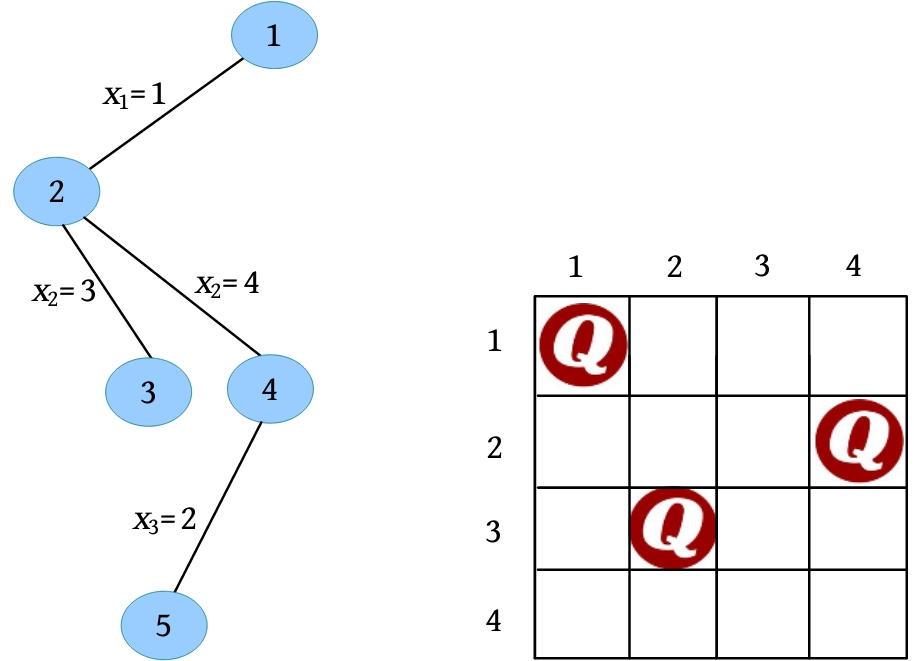
（3）扩展节点3（t=3）。判断x3=1不满足约束条件，因为与之前放置的第1个皇后同列；考察x3=2也不满足约束条件，因为与之前放置的第2个皇后同斜线；考察x2=3不满足约束条件，因为与之前放置的第2个皇后同列；考察x3=4也不满足约束条件，因为与之前放置的第2个皇后同斜线；对节点3的所有孩子均已考察完毕，节点3成为死节点。向上回溯到节点2。



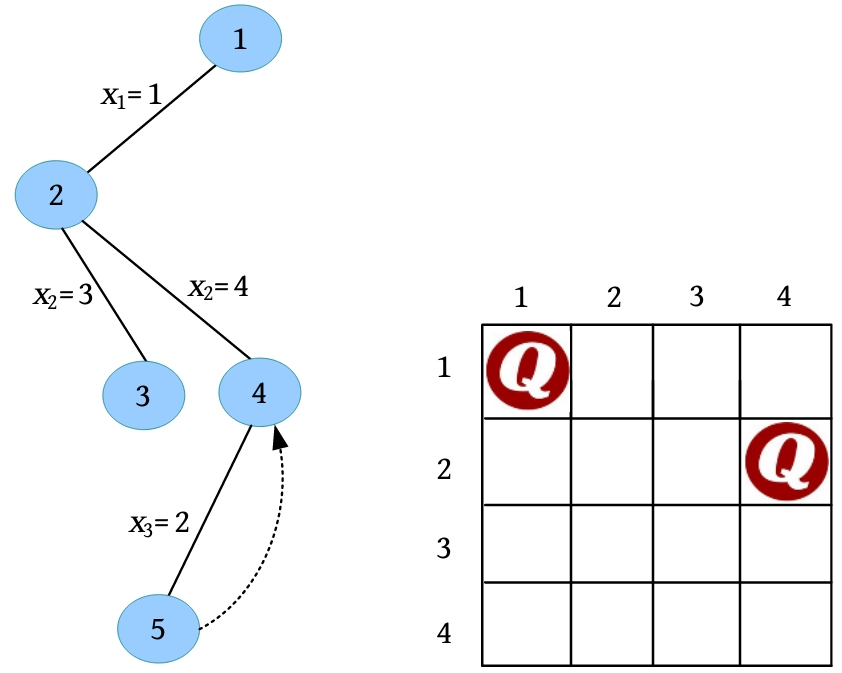
（4）重新扩展节点2（t=2）。判断x2=4满足约束条件，因为与之前放置的第1个皇后不同列、不同斜线，令x[2]=4，生成节点4。



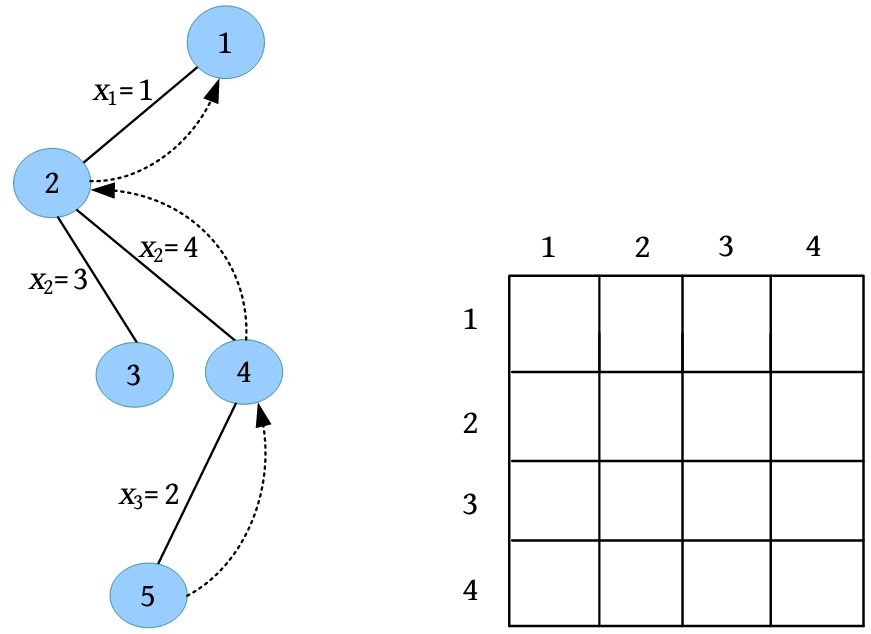
（5）扩展节点4（t=3）。判断x3=1不满足约束条件，因为与之前放置的第1个皇后同列；考察x3=2满足约束条件，因为与之前放置的第1、2个皇后不同列、不同斜线，令x[3]=2，生成节点5。



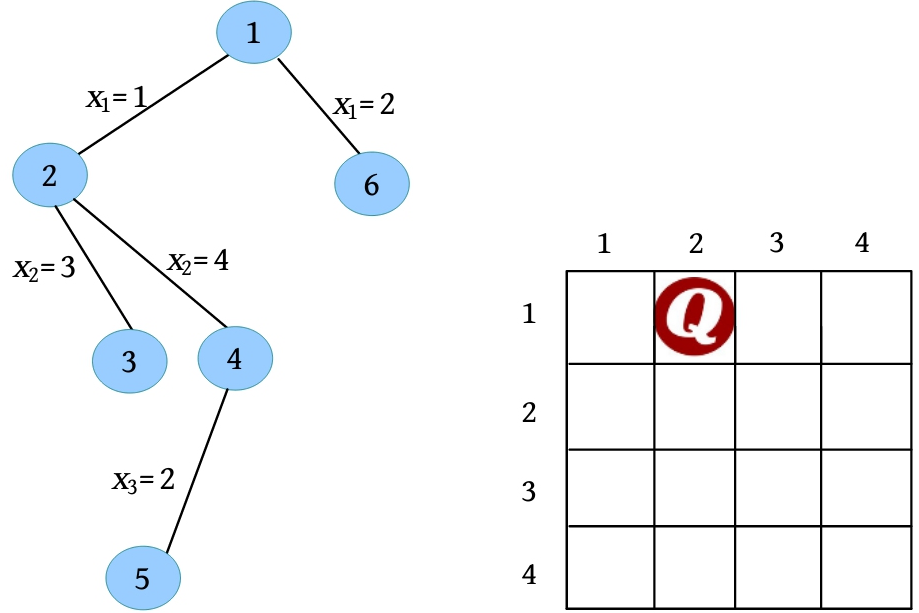
（6）扩展节点5（t=4）。判断x4=1不满足约束条件，因为与之前放置的第1个皇后同列；考察x4=2也不满足约束条件，因为与之前放置的第3个皇后同列；考察x4=3不满足约束条件，因为与之前放置的第3个皇后同斜线；考察x4=4也不满足约束条件，因为与之前放置的第2个皇后同列；对节点5的所有孩子均已考察完毕，节点5成为死节点。向上回溯到节点4。



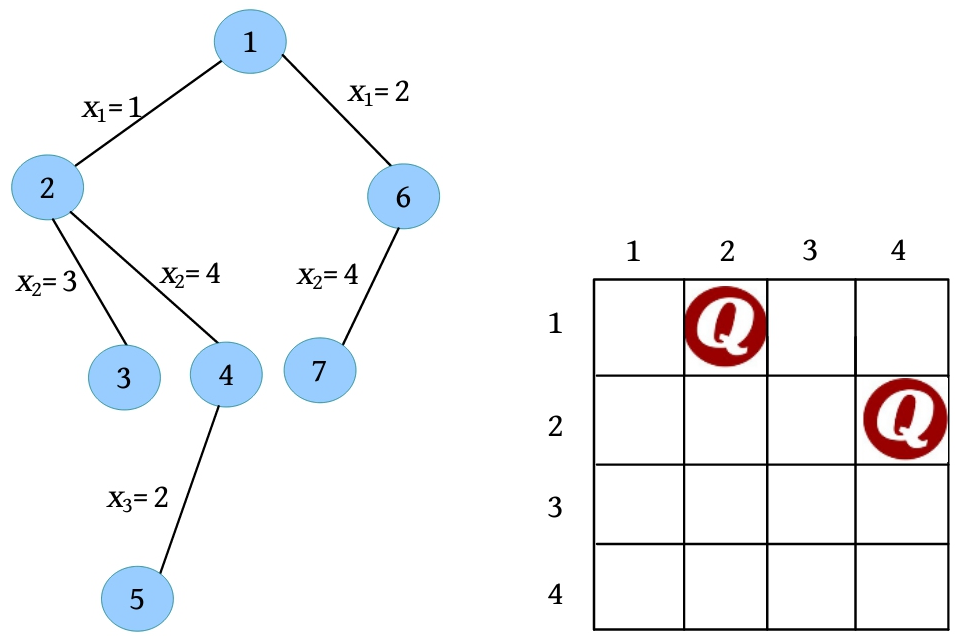
（7）继续扩展节点4（t=3）。判断x3=3不满足约束条件，因为与之前放置的第2个皇后同斜线；考察x3=4也不满足约束条件，因为与之前放置的第2个皇后同列；节点4的所有孩子均已考察完毕，节点4成为死节点。向上回溯到节点2。对节点2的所有孩子均已考察完毕，节点2成为死节点。向上回溯到节点1。



（8）继续扩展节点1（t=1）。判断x1=2是否满足约束条件，因为之前还未选中任何节点，满足约束条件。令x[1]=2，生成节点6。



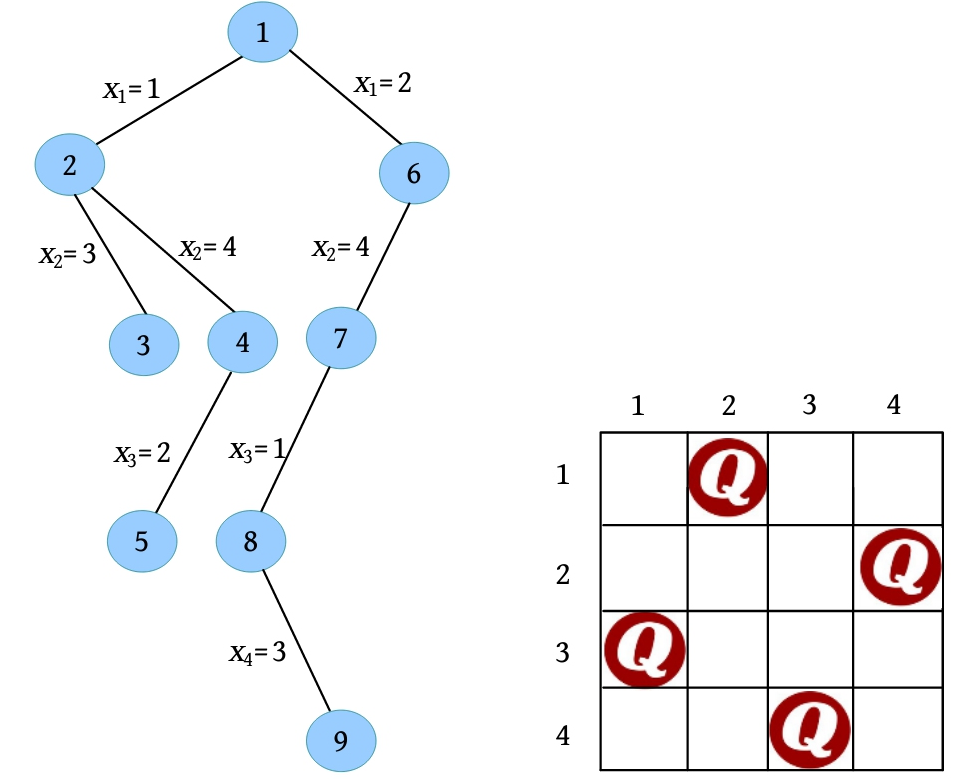
（9）扩展节点6（t=2）。判断x2=1不满足约束条件，因为与之前放置的第1个皇后同斜线；考察x2=2也不满足约束条件，因为与之前放置的第1个皇后同列；考察x2=3不满足约束条件，因为与之前放置的第1个皇后同斜线；考察x2=4满足约束条件，因为与之前放置的第1个皇后不同列、不同斜线，令x[2]=4，生成节点7。



（10）扩展节点7（t=3）。判断x3=1满足约束条件，因为与之前放置的第1、2个皇后不同列、不同斜线，令x[3]=1，生成节点8。



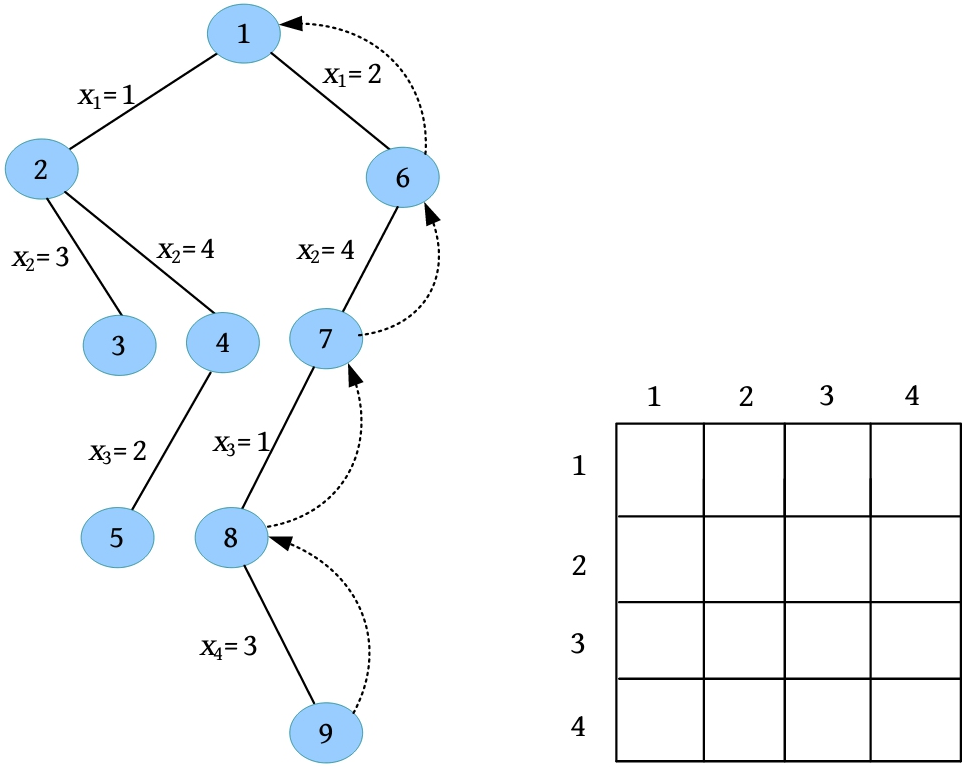
（11）扩展节点8（t=4）。判断x4=1不满足约束条件，因为与之前放置的第3个皇后同列；考察x4=2也不满足约束条件，因为与之前放置的第1个皇后同列；考察x4=3满足约束条件，因为与之前放置的第1、2、3个皇后不同列、不同斜线，令x[4]=3，生成节点9。



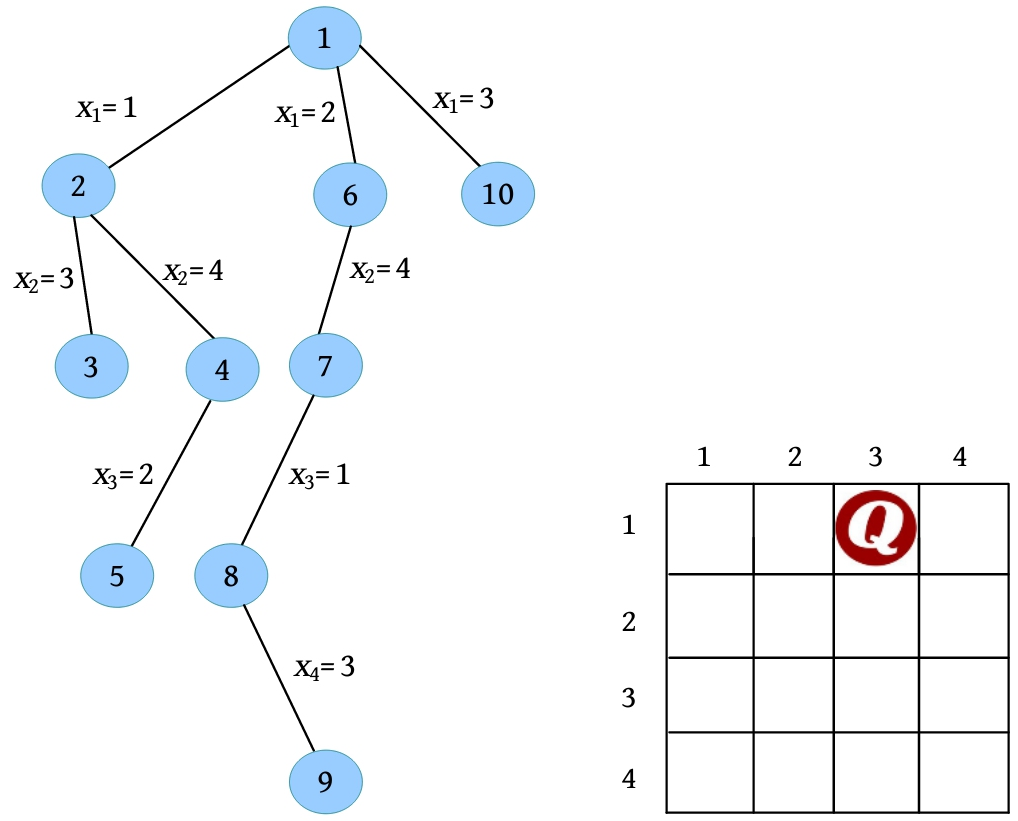
（12）扩展节点9（t=5）。t>n，找到一个可行解，用bestx[]保存当前可行解{2,4,1,3}。节点9成为死节点。向上回溯到节点8。

（13）继续扩展节点8（t=4）。判断x4=4不满足约束条件，因为与之前放置的第2个皇后同列；对节点8的所有孩子均已考察完毕且成为死节点。向上回溯到节点7。

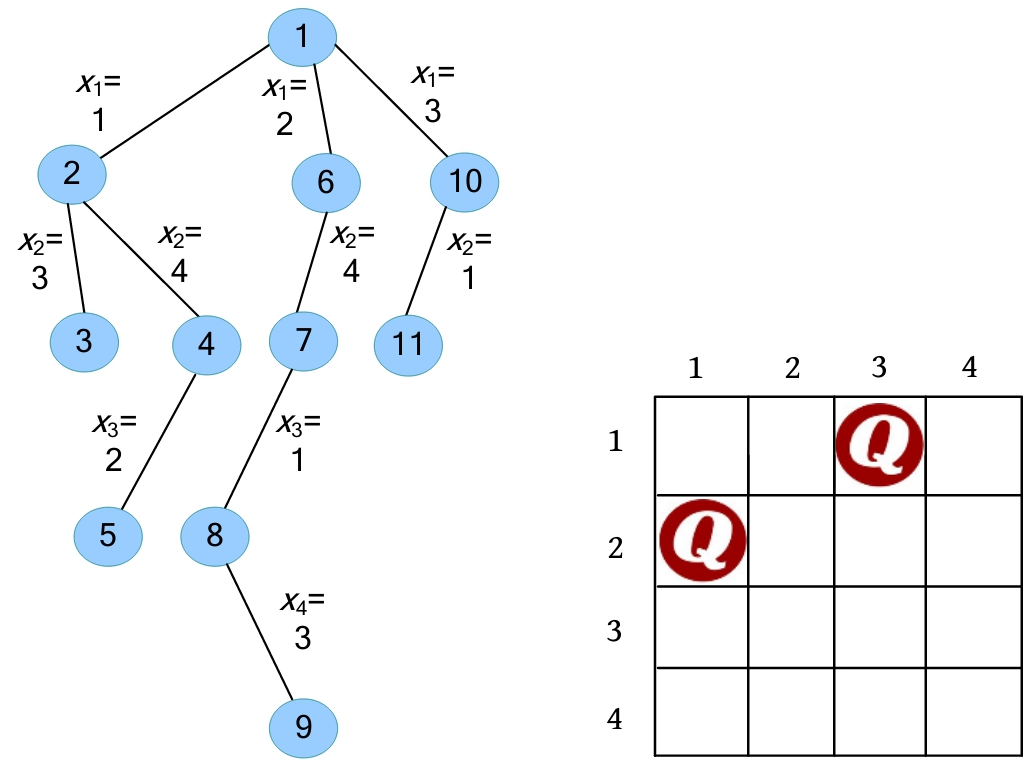
（14）继续扩展节点7（t=3）。判断x3=2不满足约束条件，因为与之前放置的第1个皇后同列；判断x3=3不满足约束条件，因为与之前放置的第2个皇后同斜线；判断x3=4不满足约束条件，因为与之前放置的第2个皇后同列；对节点7的所有孩子均已考察完毕成为死节点。向上回溯到节点6。节点6的所有孩子均已考察完毕并成为死节点。向上回溯到节点1。



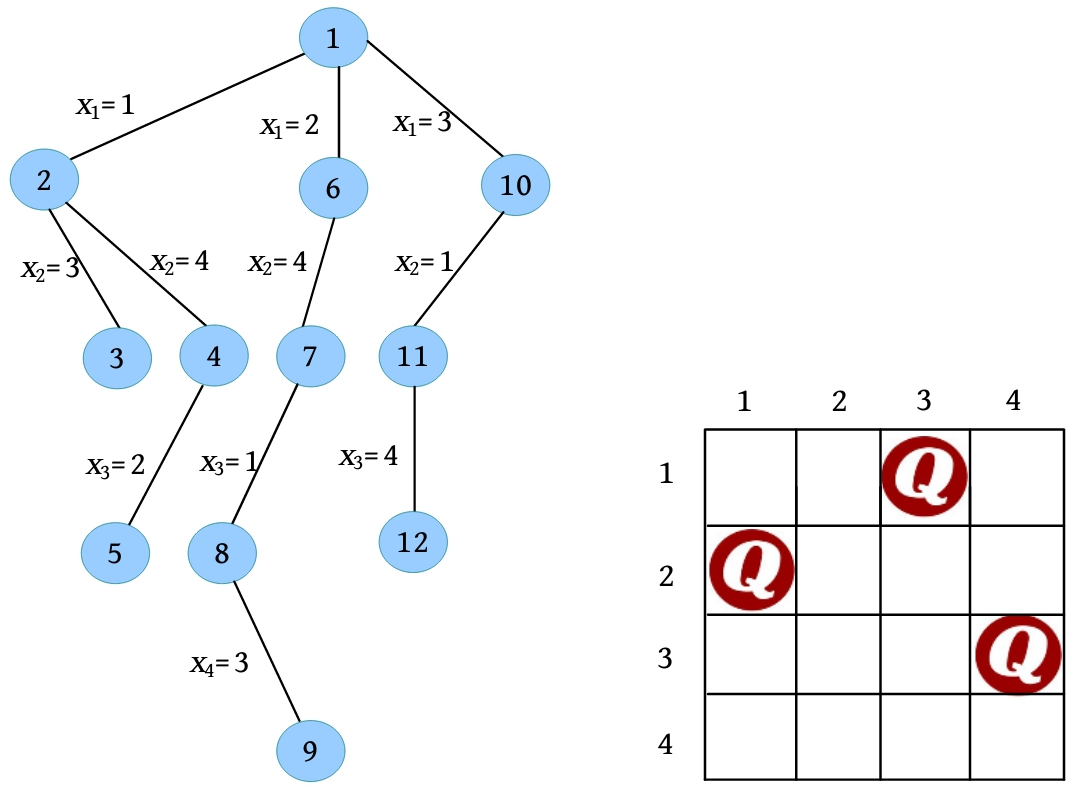
（15）继续扩展节点1（t=1）。判断x1=3是否满足约束条件，因为之前还未选中任何节点，满足约束条件。令x[1]=3，生成节点10。



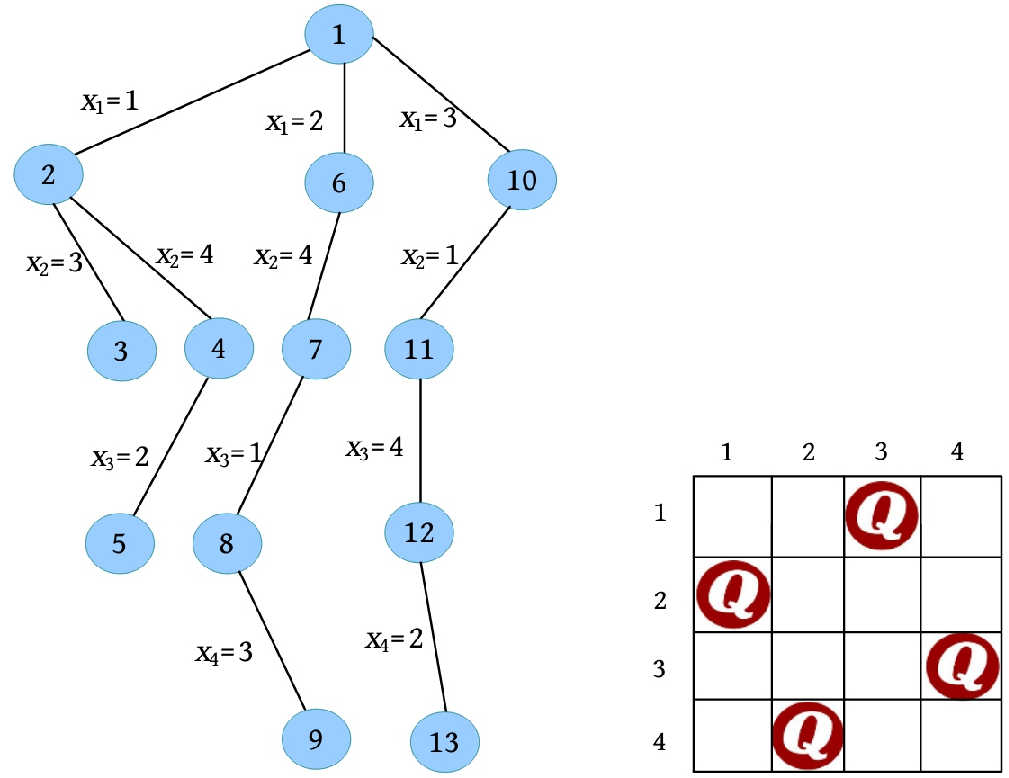
（16）扩展节点10（t=2）。判断x2=1满足约束条件，因为与之前放置的第1个皇后不同列、不同斜线，令x[2]=1，生成节点11。



（17）扩展节点11（t=3）。判断x3=1不满足约束条件，因为与之前放置的第2个皇后同列；考察x3=2也不满足约束条件，因为与之前放置的第2个皇后同斜线；考察x3=3不满足约束条件，因为与之前放置的第1个皇后同列；考察x3=4满足约束条件，因为与之前放置的第1、2个皇后不同列、不同斜线，令x[3]=4，生成节点12。



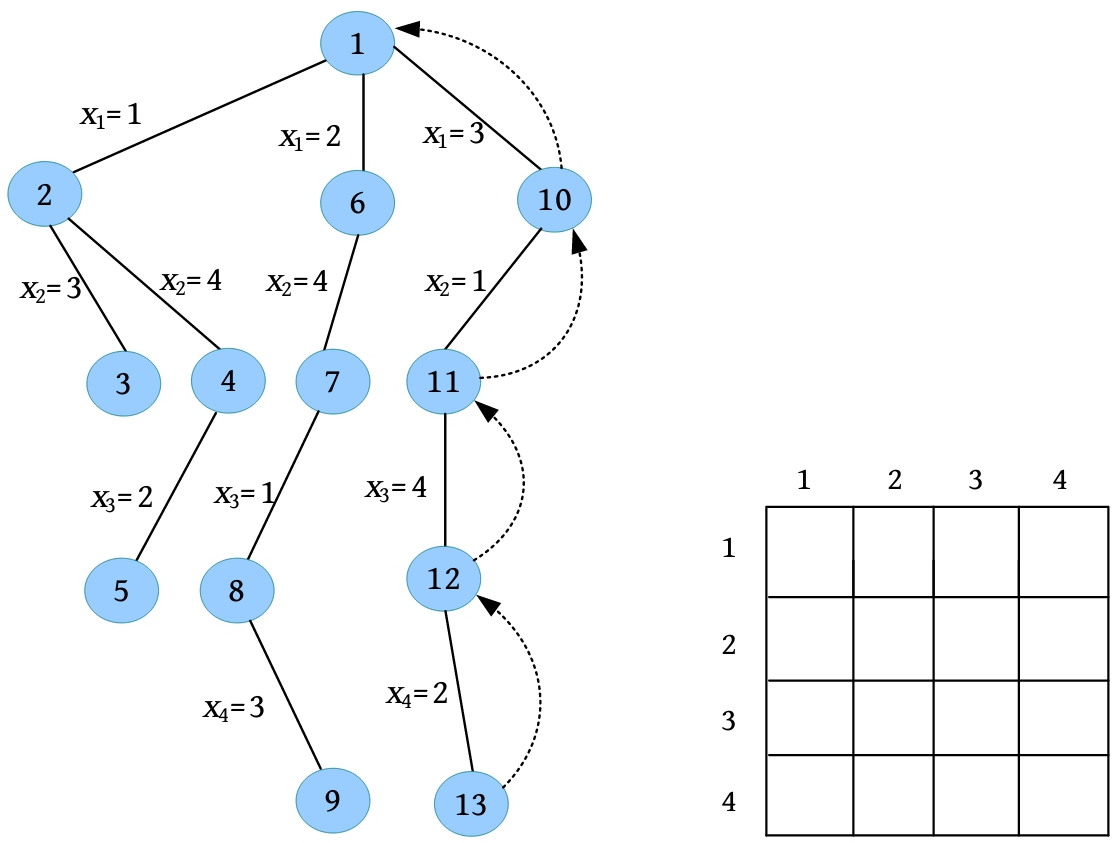
（18）扩展节点12（t=4）。判断x4=1不满足约束条件，因为与之前放置的第2个皇后同列；考察x4=2满足约束条件，因为与之前放置的第1、2、3个皇后不同列、不同斜线，令x[4]=2，生成节点13。



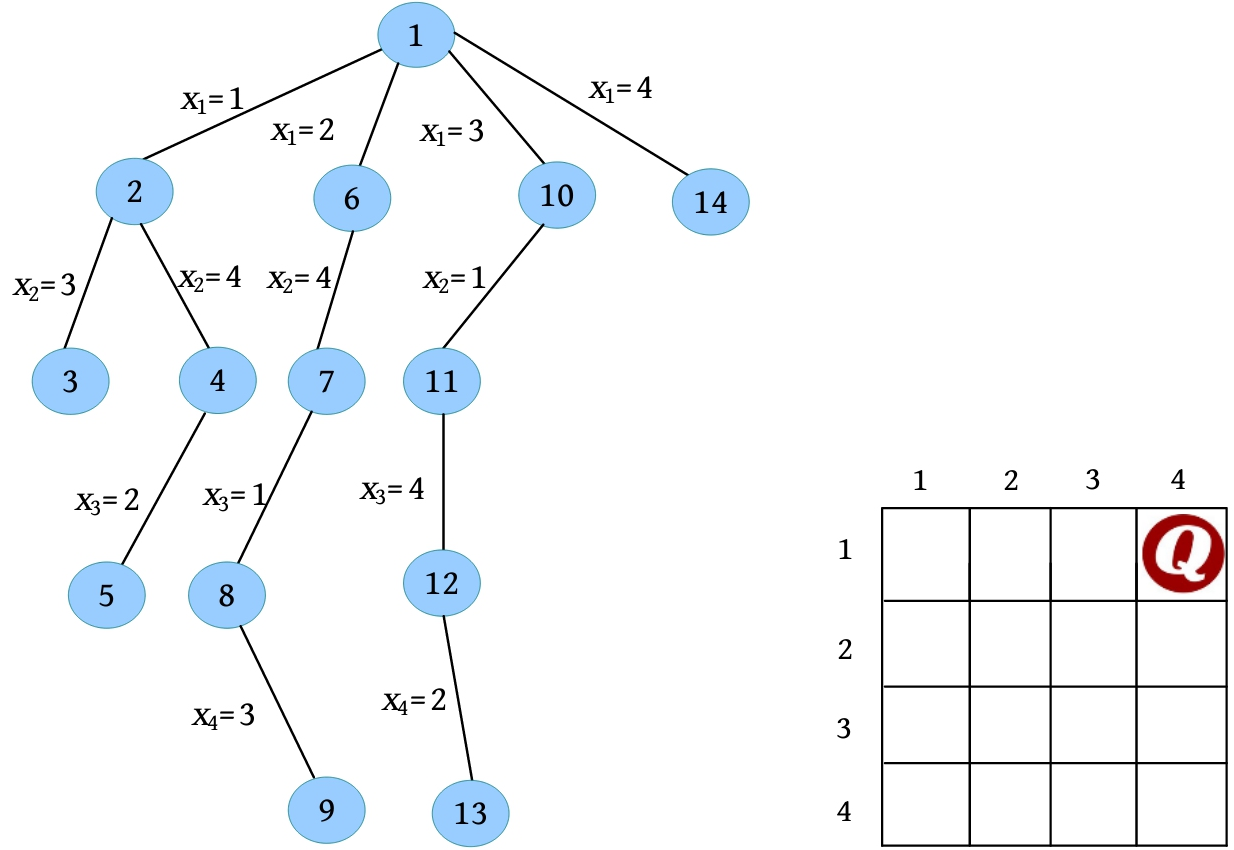
（19）扩展节点13（t=5）。t>n，找到一个可行解，用bestx[]保存当前可行解{3,1,4,2}。节点13成为死节点。向上回溯到节点12。

（20）继续扩展节点12（t=4）。判断x4=3不满足约束条件，因为与之前放置的第1个皇后同列；判断x4=4不满足约束条件，因为与之前放置的第3个皇后同列；对节点12的所有孩子均已考察完毕并成为死节点。向上回溯到节点11。对节点11的所有孩子均已考察完毕并成为死节点。向上回溯到节点10。

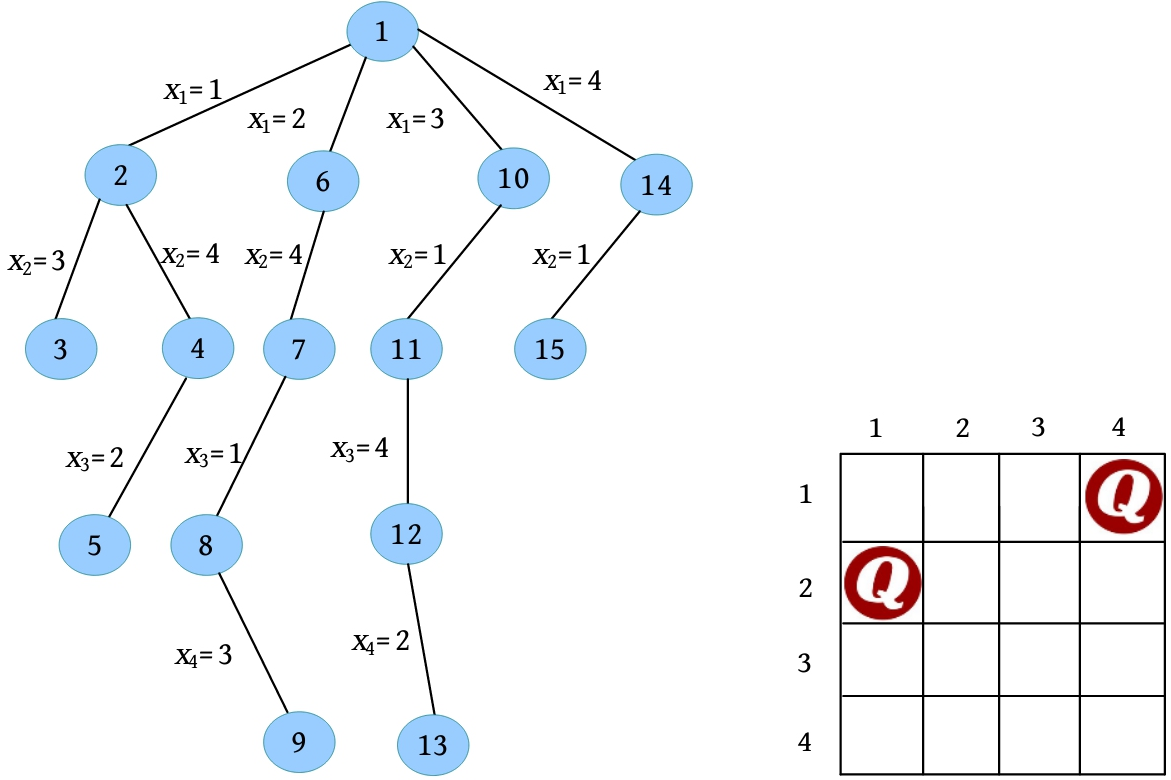
（21）继续扩展节点10（t=2）。判断x2=2不满足约束条件，因为与之前放置的第1个皇后同斜线；判断x2=3不满足约束条件，因为与之前放置的第1个皇后同列；判断x2=4不满足约束条件，因为与之前放置的第1个皇后同斜线；对节点10的所有孩子均已考察完毕并成为死节点，向上回溯到节点1。



（22）继续扩展节点1（t=1）。判断x1=4是否满足约束条件，因为之前还未选中任何节点，满足约束条件。令x[1]=4，生成节点14。



（23）扩展节点14（t=2）。判断x2=1满足约束条件，因为与之前放置的第1个皇后不同列、不同斜线，令x[2]=1，生成节点15。



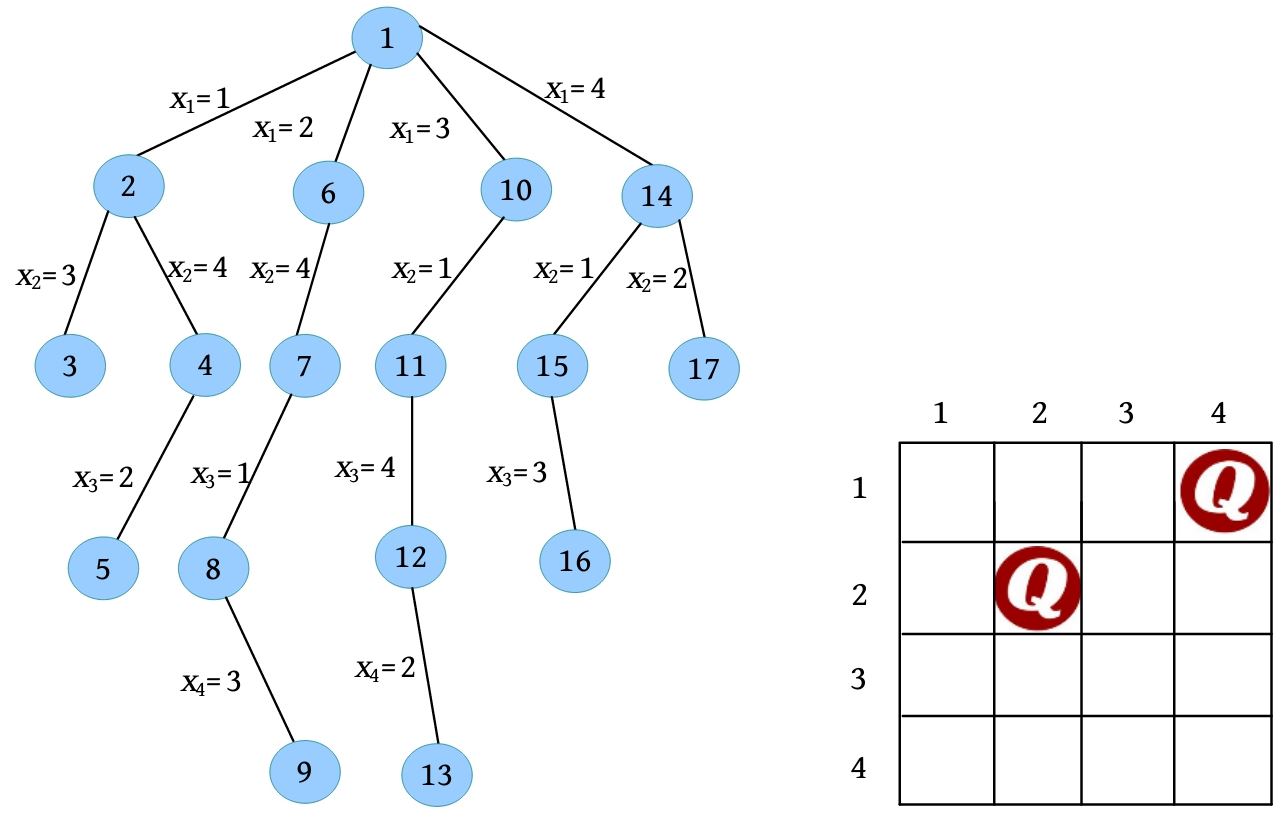
（24）扩展节点15（t=3）。判断x3=1不满足约束条件，因为与之前放置的第2个皇后同列；考察x3=2也不满足约束条件，因为与之前放置的第2个皇后同斜线；考察x3=3满足约束条件，因为与之前放置的第1、2个皇后不同列、不同斜线，令x[3]=3，生成节点16。



（25）扩展节点16（t=4）。判断x4=1不满足约束条件，因为与之前放置的第2个皇后同列；考察x4=2也不满足约束条件，因为与之前放置的第3个皇后同斜线；考察x4=3不满足约束条件，因为与之前放置的第3个皇后同列；考察x4=4也不满足约束条件，因为与之前放置的第1个皇后同列；对节点16的所有孩子均已考察完毕并成为死节点。向上回溯到节点15。

（26）继续扩展节点15（t=3）。判断x3=4不满足约束条件，因为与之前放置的第1个皇后同列；对节点15的所有孩子均已考察完毕并成为死节点。向上回溯到节点14。

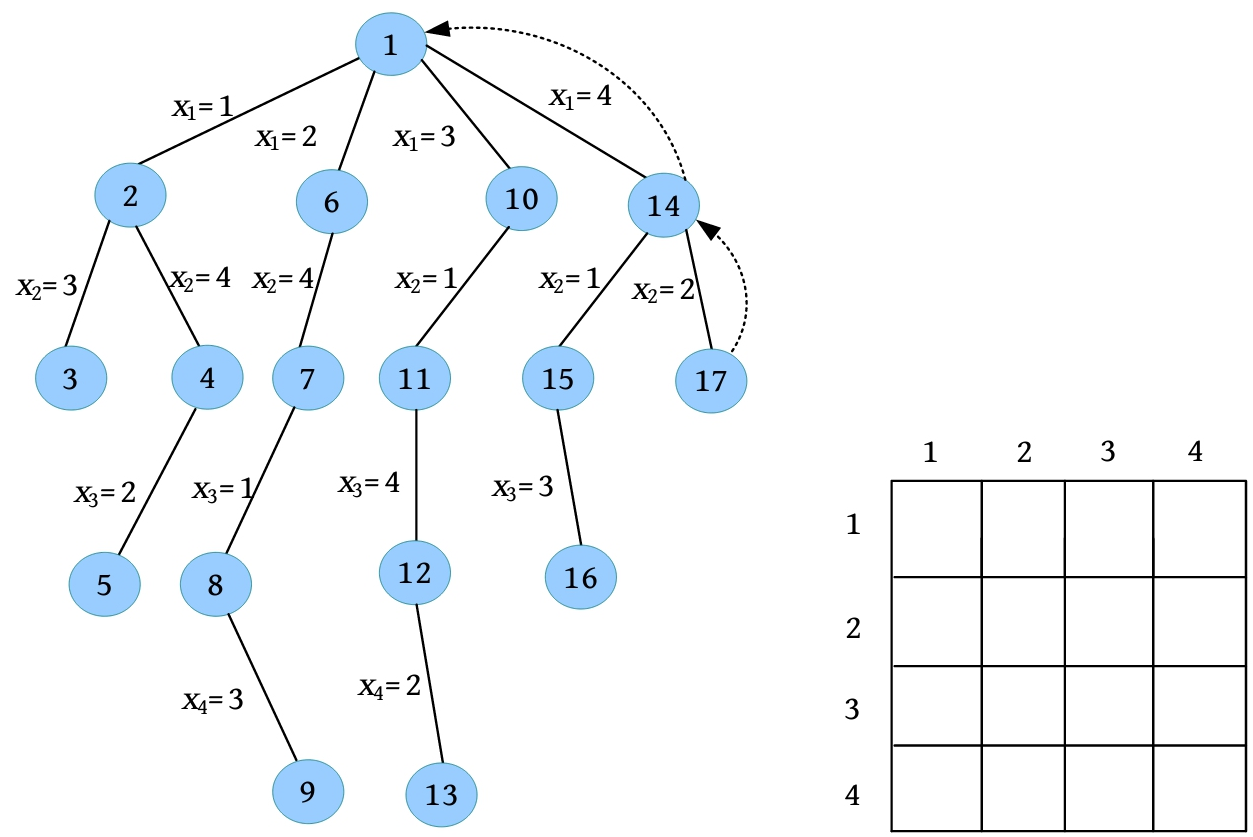
（27）继续扩展节点14（t=2）。判断x2=2满足约束条件，因为与之前放置的第1个皇后不同列、不同斜线，令x[2]=2，生成节点17。



（28）扩展节点17（t=3）。判断x3=1不满足约束条件，因为与之前放置的第2个皇后同斜线；考察x3=2也不满足约束条件，因为与之前放置的第2个皇后同列；考察x3=3不满足约束条件，因为与之前放置的第2个皇后同斜线；考察x3=4也不满足约束条件，因为与之前放置的第1个皇后同列；对节点17的所有孩子均已考察完毕并成为死节点。向上回溯到节点14。

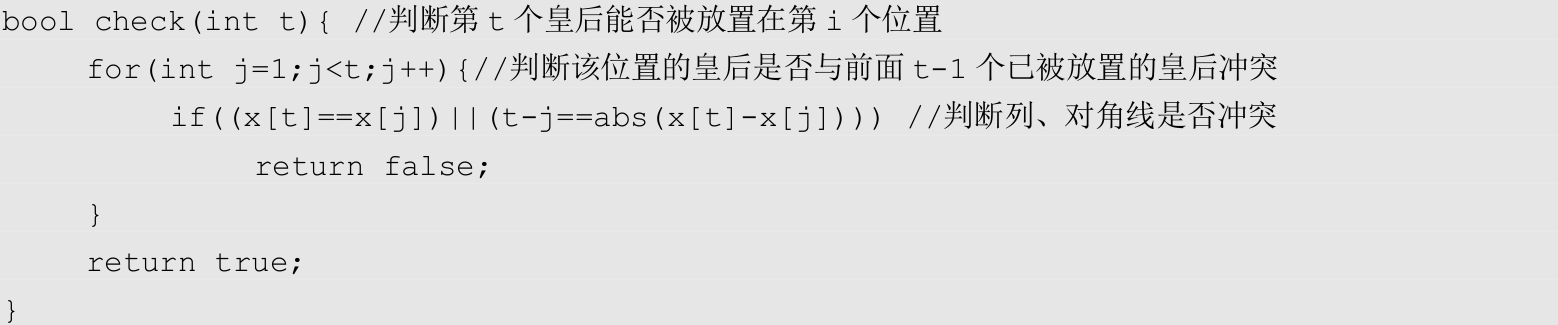
（29）继续扩展节点14（t=2）。判断x3=3不满足约束条件，因为与之前放置的第2个皇后同斜线；判断x3=4不满足约束条件，因为与之前放置的第1个皇后同列；对节点14的所有孩子均已考察完毕并成为死节点。向上回溯到节点1。

（30）对节点1的所有孩子均已考察完毕并成为死节点，算法结束。

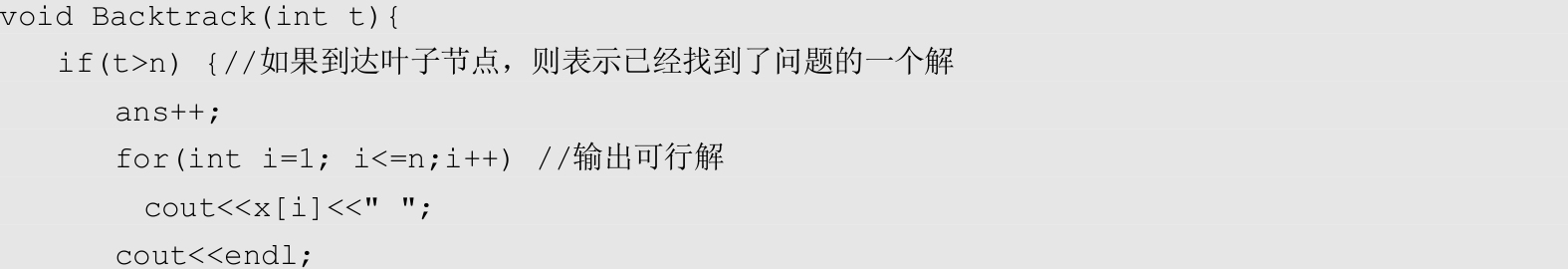


**3. 算法实现**

**（1）约束函数。**在第t行放置第t个皇后时，第t个皇后与前t-1个已放置好的皇后不能同列或同斜线。如果有一个成立，则第t个皇后不可以被放置在该位置。**x[t]==x[j]**表示第t个皇后与第j个皇后同列，**t-j==abs(x[t]-x[j])**表示第t个皇后与第j个皇后同斜线。abs是求绝对值的函数，使用该函数时要引入头文件#include<cmath>。



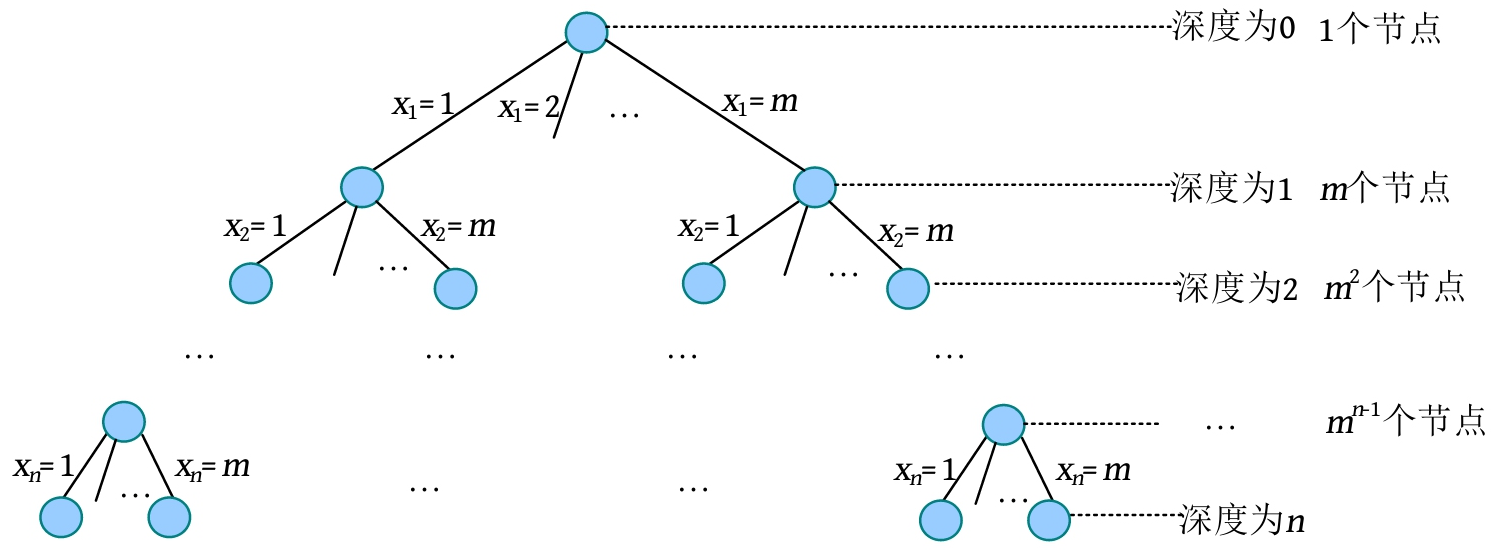
**（2）按约束条件搜索求解。**t表示当前扩展节点在第t层。如果t>n，则表示已经到达叶子节点，记录最优值和最优解，返回。否则，分别判断n(i=1…n)个分支，x[t]=i；判断每个分支是否满足约束条件，如果满足，则进入下一层Backtrack(t+1)，否则考察下一个分支（兄弟节点）。



****

**4. 算法分析**

**时间复杂度：**在最坏情况下，解空间树如下图所示。除了最后一层，有1+n+n2+…+nn-1= (nn-1)/(n-1)≈nn-1个节点需要扩展，而这些节点的每一个都要扩展n个分支，总的分支个数为nn，每个分支都判断约束函数，判断约束条件需要O(n)时间，因此耗时O(nn+1)。在叶子节点处输出当前最优解需要耗时O(n)，在最坏情况下会搜索到每一个叶子节点，叶子个数为nn，故耗时O(nn+1)。因此，时间复杂度为**O(nn+1)**。



**空间复杂度：**回溯法的另一个重要特性就是在搜索执行的同时产生解空间。在搜索过程中的任何时刻，仅保留从开始节点到当前扩展节点的路径，从开始节点起最长的路径为n。在程序中使用x[]数组记录该最长路径作为可行解，所以该算法的空间复杂度为**O(n)**。

**5. 算法优化**

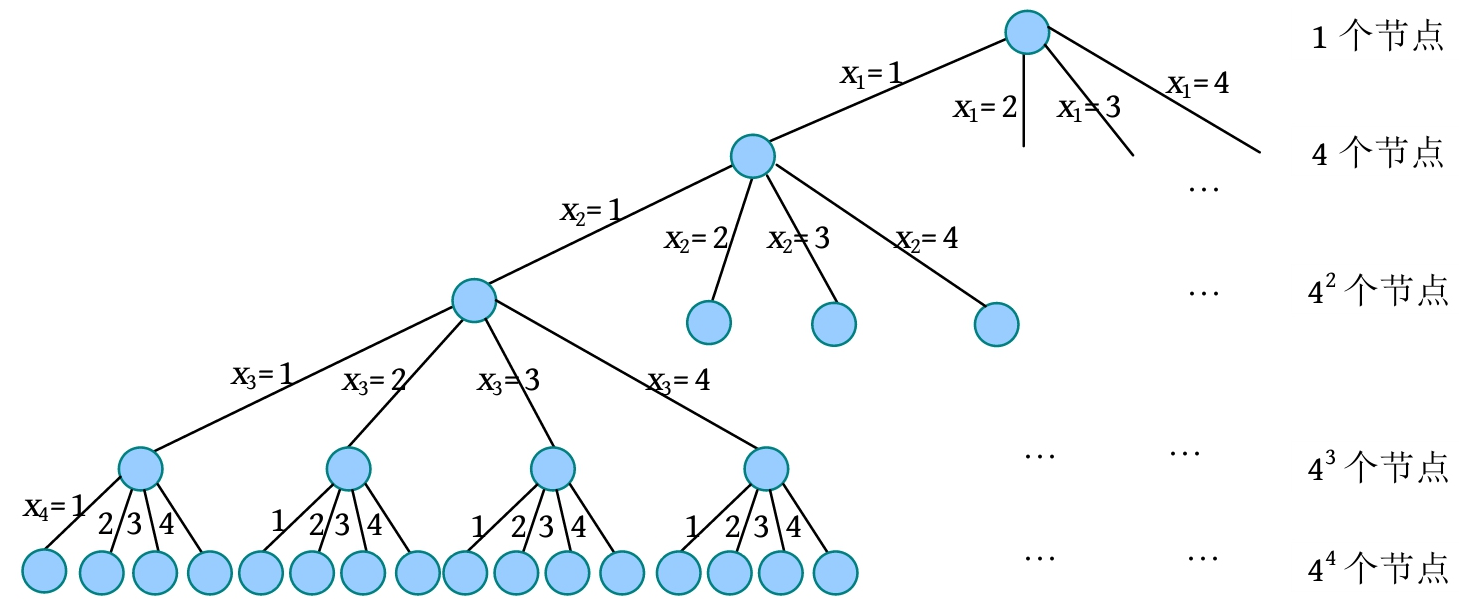
在上面的求解过程中，我们的解空间过于庞大，所以时间复杂度很高，算法效率当然会降低。**解空间越小，算法效率越高**。

那么能不能把解空间缩小呢？n皇后问题要求每一个皇后都不同行、不同列、不同斜线。

上图所示的解空间**使用了不同的行作为显约束**。

**隐约束**为**不同列、不同斜线**。

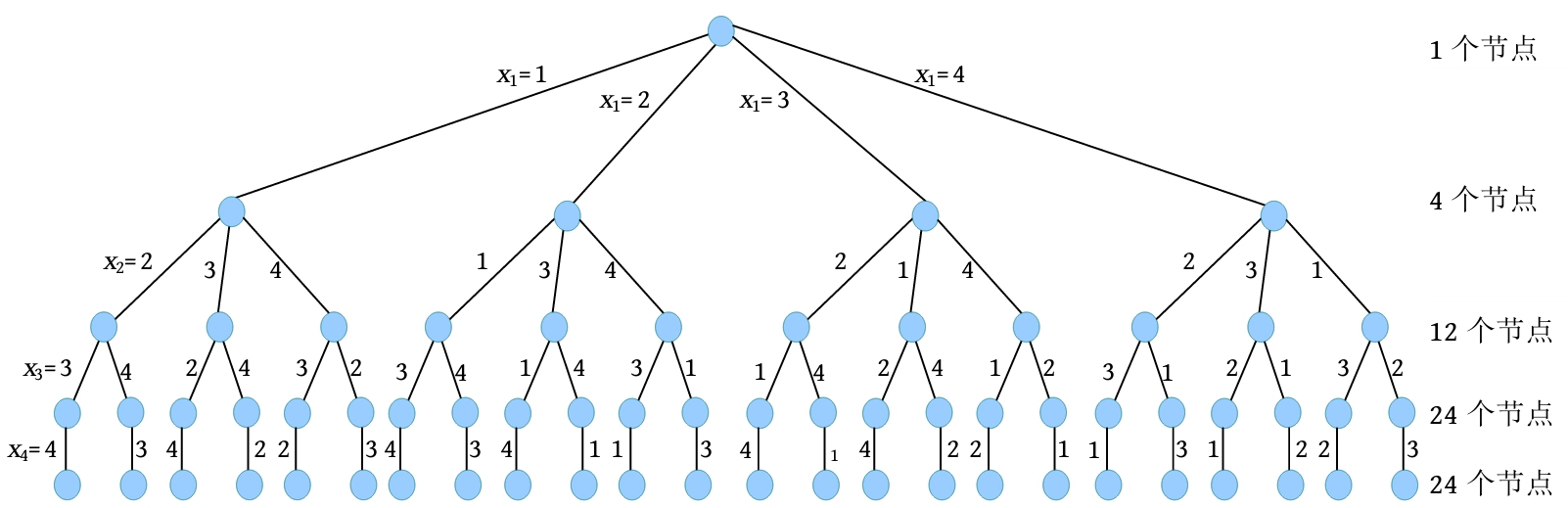
对4皇后问题，显约束为不同行的解空间树如下图所示。



**显约束可以控制解空间大小**，**隐约束是在搜索解空间过程中判定可行解或最优解的**。

**如果我们把显约束定为不同行、不同列，把隐约束定义为不同斜线，那么解空间是怎样的呢？**

例如x1=1的分支，x2就不能再等于1，因为这样就同列了。如果x1=1，x2=2，x3就不能再等于1、2。也就是说，**xt的值不能与前t-1个解的取值相同**。**每层节点产生的孩子数都比上一层少1**。4皇后问题，**显约束为不同行、不同列**的解空间树如下图所示。



我们可以清楚地看到解空间变小许多，仔细观察就会发现，上图中，**从根到叶子的每一个可能解其实都是一个排列**，**该**解空间树**是一棵排列树**。使用排列树求解n皇后问题的代码如下。

