南京邮电大学
Nanjing University of Posts & Telecommunications

算法设计与分析

第8章 回溯法

学习要点：

- 理解回溯法的深度优先搜索策略
- 理解剪枝函数（约束函数和限界函数）的作用
- 掌握回溯法解题的算法框架
  （1）递归回溯的算法框架
  （2）迭代回溯的算法框架
  （3）蒙特卡罗方法进行效率分析
- 通过应用范例学习回溯法的设计策略。
  （1）n-皇后问题
  （2）子集和数问题

◆章节内容：

8.1 一般方法

8.2 n-皇后

8.3 子集和数

8.4 图的着色

8.5 哈密顿环

8.6 0/1背包

8.7 批处理作业调度

8.1 回溯法的一般方法

- 采用**深度优先**产生状态空间树的结点，并使用**剪枝函数**的方法称为——**回溯法**。
- 回溯法的基本做法是：

  按深度优先策略，从根结点出发搜索问题的状态空间树，求问题的可行解或最优解。算法搜索至任一结点时，先判断以该结点为根的子树是否包含问题的解。
  - 如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；
  - 否则，进入该子树，继续按深度优先策略搜索。

回溯法适用于解一些组合数相当大的问题。

问题的解空间

回溯法要求问题的解向量具有n元组$(x_1,x_2,\ldots,x_n)$的形式，每个解分量x_i从一个给定的集合S_i取值。

●**显式约束**：规定每个x_i取值的约束条件。

（显式约束规定了问题的候选解集——解空间）

解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

●**隐式约束**：给出了判定一个候选解是否为可行解的条件。为满足问题的解而对不同分量之间施加的约束。

通常情况下,回溯法的求解目标是在状态空间树上找出满足约束条件的所有可行解.

注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小，搜索方法更简单。

例如（例8-1）：对n个元素$(a_0,a_1,\ldots\ldots,a_{n-1})$进行排序，求元素下标$(0,1,\ldots\ldots n-1)$的一种排列$X=(x_0,x_1,\ldots\ldots,x_{n-1})$。使$a_{x_i} \le a_{x_{i+1}}$ $(0\le i<n-1)$

一种方法：

- **显式约束：** $x_i \in S_i, S_i=\{0,1,\ldots\ldots,n-1\}$。此时解空间大小为$n^n$。
- **隐式约束：** $a_{x_i} \le a_{x_{i+1}}$ $(0\le i<n-1)$，且$x_i \ne x_j$ $(i\ne j)$

另一种方法：

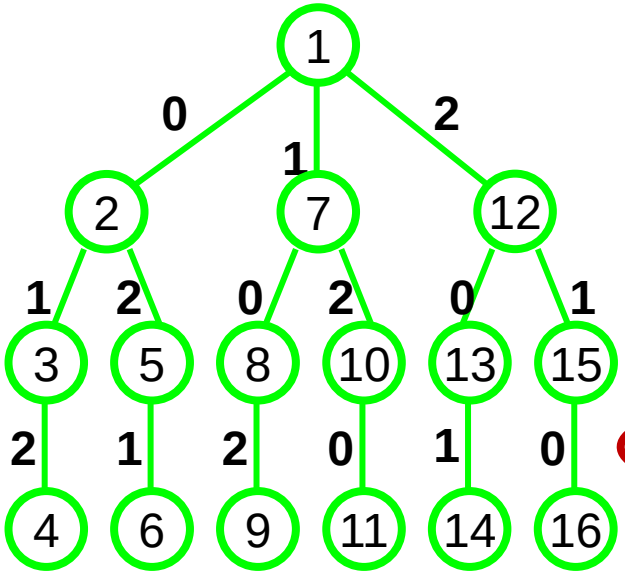
- **显式约束：** $x_i \in S_i, S_i=\{0,1,\ldots\ldots,n-1\}$且$x_i \ne x_j$ $(i\ne j)$。此时解空间大小为$n!$。
- **隐式约束：** $a_{x_i} \le a_{x_{i+1}}$ $(0\le i<n-1)$

状态空间树

状态空间树：描述问题解空间的**树形结构**。

- 树中每个结点称为一个**问题状态**；
- 若从根到树中某个状态的路径代表一个候选解元组，则该状态为**解状态**；
- 若从根到某个解状态的路径代表一个可行解元组，则该解状态为**答案状态**；
- 如果求解的是最优化问题，还要用目标函数衡量每个答案结点，找出其中目标函数取最优值的**最优答案结点**。

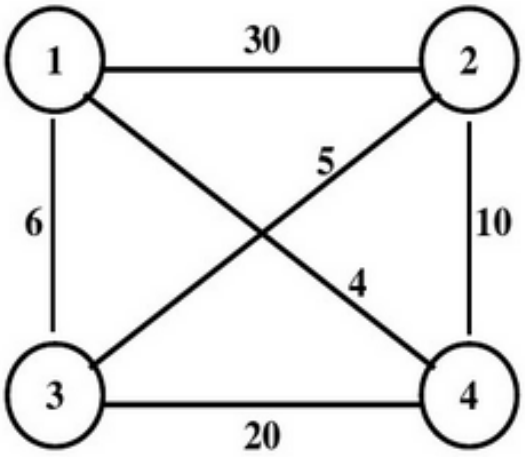
如：（例8-1）n个元素排序问题。若n=3则状态空间树为：

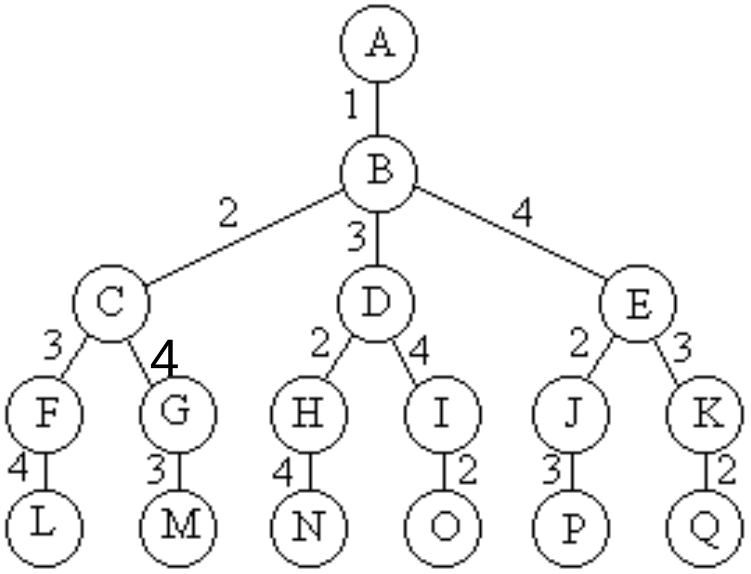


排序问题一般不用回溯法求解

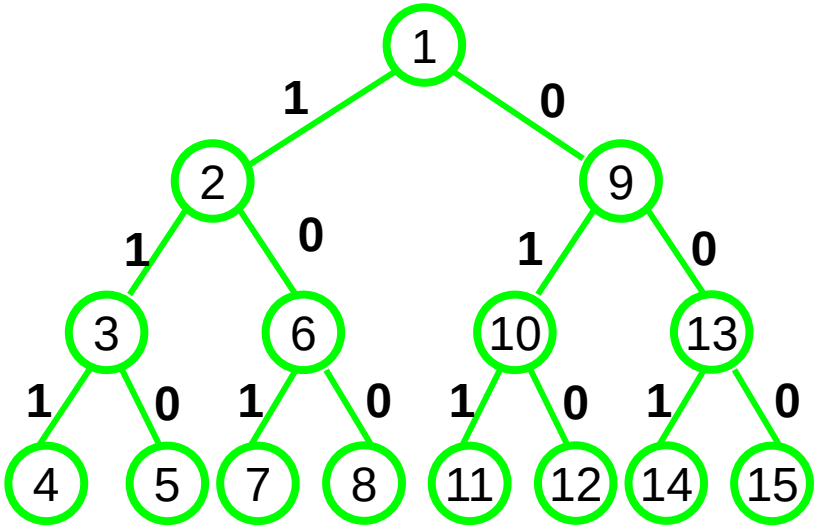
有n!个叶结点（解状态）

旅行商问题





又如：0-1背包问题。若n=3则状态空间树为一个完全二叉树：



有2^n个解状态

w=[16,15,15]，p=[45,25,25]，背包容量C=30

回溯法的基本思想

穷举搜索法：

- 使用某种
-
- 对
目标函数值，

回溯法解题的一个显著特征是：在搜索过程中**动态**产生问题的解空间。

回溯法：

- 在求解的过程中，随着搜索算法的进展，以**深度优先方式**，逐个生成状态空间树中结点；
- 为提高搜索效率，在搜索过程中用**约束函数**和**限界函数**（统称**剪枝函数**）来剪去不必要搜索的子树，减少问题求解所需实际生成的状态空间树结点数，从而避免无效搜索。

常用的剪枝函数：

用约束函数剪去已知不含答案状态（可行解）的子树；

用限界函数剪去得不到最优答案结点（最优解）的子树。

递归回溯

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

程序8-1 递归回溯算法框架

```
void RBacktrack (int k)
{   for (每个x[k],使得x[k]∈T(x[0],...,x[k-1] )&&( Bk(x[0],...,x[k]) ) {
        // T(x0,x1,...,xk-1)表示沿路径(x0,x1,...,xk-1)从根到某个问题状态时，孩子结点xk可取值的集合。
        // Bk(x0,x1,...,xk)为约束函数。若子树上不含任何答案状态，则为false；否则,为true。
        if ((x[0],x[1],...,x[k])是一个可行解)        //判断是否可行解
            输出(x[0],x[1],...,x[k]);                //输出可行解
        RBacktrack(k+1);                             //深度优先进入下一层
    }
}
```

由于叶结点的孩子集合T( )为空集合，因此对于有限状态空间树，算法必能终止。

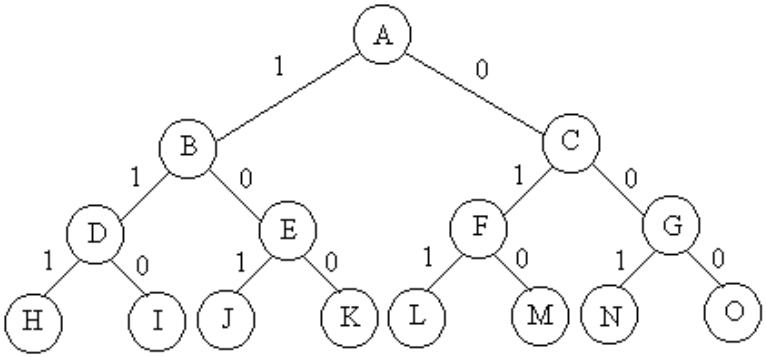
迭代回溯

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

程序8-2 迭代回溯算法框架

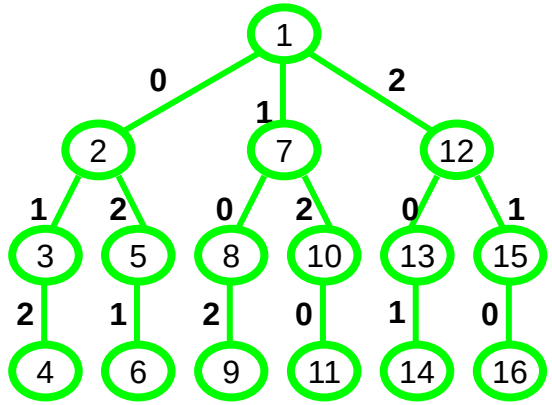
```
void IBacktrack(int n)
{   int k=0;
    while (k>=0) {
        if (还剩下尚未检测的x[k],使得x[k]∈T(x[0],...,x[k-1])&&Bk(x[0],...,x[k])
        {   if ((x[0],x[1],...,x[k])是一个可行解)
                输出(x[0],x[1],...,x[k]);          //输出可行解
            k++;                                   //深度优先进入下一层
        }
        else k--;                                  //回溯到上一层
    }
}
```

子集树与排列树



遍历子集树需$O(2^n)$计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (legal(t)) backtrack(t+1);
      }
}
```



遍历排列树需要O(n!)计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++) {
        swap(x[t], x[i]);
        if (legal(t)) backtrack(t+1);
        swap(x[t], x[i]);
      }
}
```

回溯法的效率分析

回溯法的时间通常取决于：状态空间树上实际生成的那部分问题状态的数目（即：结点总数）。

用蒙特卡罗（Monte Carlo）方法估计回溯法处理实例时，实际生成的结点数：

- 在状态空间树中,从根开始随机选择一条路径$(x_0,x_1,\ldots,x_{n-1})$;
- 假定搜索树中这条随机选出的路径上，代表部分向量$(x_0,x_1,\ldots,x_{k-1})$的结点处不受限制的孩子数目，与同层的其他路径上的结点不受限制的孩子数目一样，都是m_k;
- 则可以估计整个状态空间树上实际生成的结点数为：

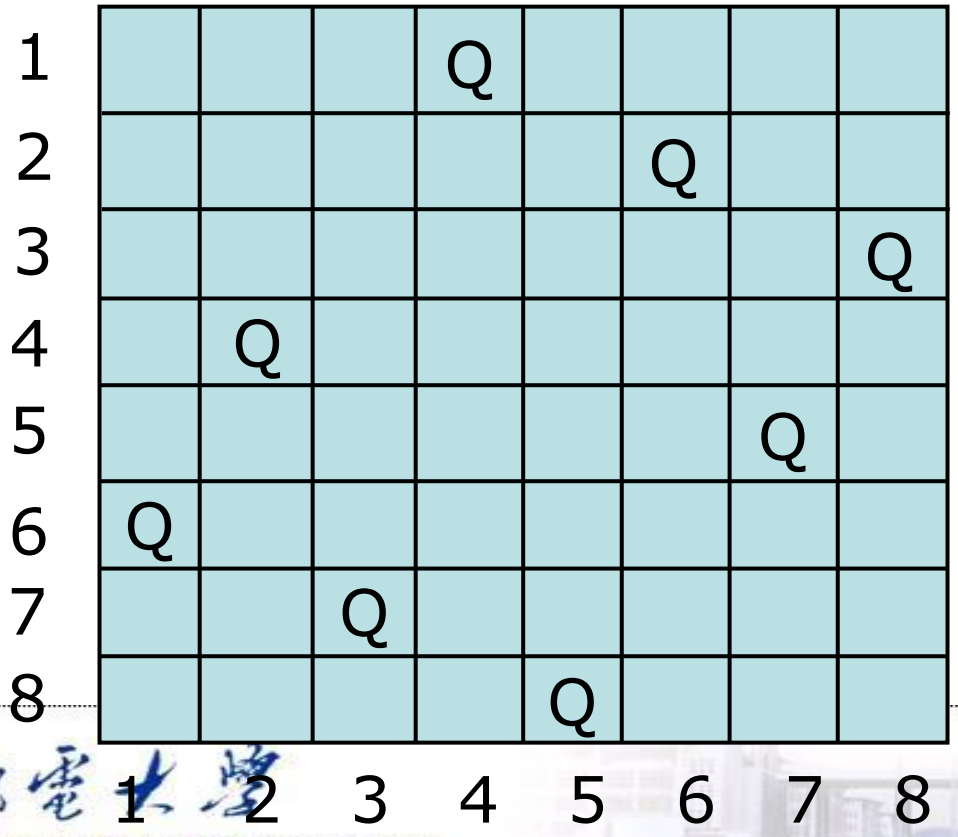
$$m=1+m_0+m_0m_1+m_0m_1m_2+\ldots\ldots$$

程序8-3 蒙特卡罗算法

```
do{
    SetType S={ x[k] | x[k]∈T(x[1],...,x[k-1])&& Bk(x[1],...,x[k])==true};
    if (!Size(S)) return m;
    r=r*Size(S);          //r为第k层中未受限结点数的估计值
    m=m+r;                //m为状态空间树中结点总数的估计值
    x[k]=Choose(S);       //从集合S中为xk随机选择一个值，向下生成随机路径
    k++;
}while (1);
```

8.2 n-皇后问题

在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。



解向量：$(x_0, x_1, \ldots, x_{n-1})$,其中$x_i$表示第i行的皇后所处的列号。

第一种显式约束观点：

- 显式约束：$x_i=0,1,2,\ldots,n-1$
- 隐式约束：当$i\neq j$时，
  1)不同列：$x_i\neq x_j$
  2)不处于同一正、反对角线：$|i-j|\neq|x_i-x_j|$

对应的解空间大小为：n^n

第二种显式约束观点：

- 显式约束：
  1)$x_i=0,1,2,\ldots,n-1$
  2)不同列：$xi\neq xj$ $(i\neq j)$
- 隐式约束：不处于同一正、反对角线：$|i-j|\neq|x_i-x_j|$ $(i\neq j)$

对应的解空间大小为：n!

（采用第二种显式约束观点的）4-皇后问题状态空间树

——见P164　图8-3

一般称这种用于确定n个元素的排列满足某些性质的状态空间树为**排列树**。

（采用第一种显式约束观点）设计约束函数：

对$0 \le i,j<k$,当$i \ne j$时，要求$x_i \ne x_j$且$|i-j| \ne |x_i-x_j|$

可得程序8-4 求n-皇后问题的全部可行解（见下页）：

程序8-4 n-皇后问题的回溯算法

```
bool Place(int k,int i,int *x)      //约束函数（隐式）
{       //判断两皇后是否在同一列或斜线上
   for (int j=0;j<k;j++)
        if ((x[j]=i)||(abs(x[j]-i)==abs(j-k))) return false;
        //i为当前第k行皇后的列数
        //x[j]为已选定的前面 0~k-1 行皇后的列数
   return true;
}
void NQueens(int k,int n,int *x)
{  for (int i=0;i<n;i++)                    //显式约束
   {    if (Place(k,i,x))                   //约束函数（隐式）
        {       x[k]=i;
                if (k==n-1)
                {  for (i=0;i<n;i++) cout<<x[i]<<" ";    //输出一个可行解
                   cout<<endl;
                }
                else NQueens(k+1,n,x);          //深度优先进入下一层
        }
   }
}
```

可求得n-皇后问题的所有可行解

此处返回for循环中当前层，搜索剩余x[k]取值。

因此可输出所有可行解。

程序8-4 n-皇后问题的回溯算法

```
bool Place(int k,int i,int *x)      //约束函数（隐式）
{       //判断两皇后是否在同一列或斜线上
   for (int j=0;j<k;j++)
        if ((x[j]=i)||(abs(x[j]-i)==abs(j-k))) return false;
        //i为当前第k行皇后的列数
        //x[j]为已选定的前面 0~k-1 行皇后的列数
   return true;
}
void NQueens(int k,int n,int *x)
{  for (int i=0;i<n;i++)                //显式约束
   {    if (Place(k,i,x))               //约束函数（隐式）
        {       x[k]=i;
                if (k==n-1)
                {  for (i=0;i<n;i++) cout<<x[i]<<" ";    //输出一个可行解
                   cout<<endl;
                }
                else NQueens(k+1,n,x);          //深度优先进入下一层
        }
   }
}
```

可求得n-皇后问题的所有可行解

若x[k]的0~n-1列号取值均已检查完毕，则自然回溯至上层调用点（也即上层函数的for循环中，继续尝试x[k-1]的其他后续取值）。

图8-5 显示了4-皇后问题得到第一个解的步骤:

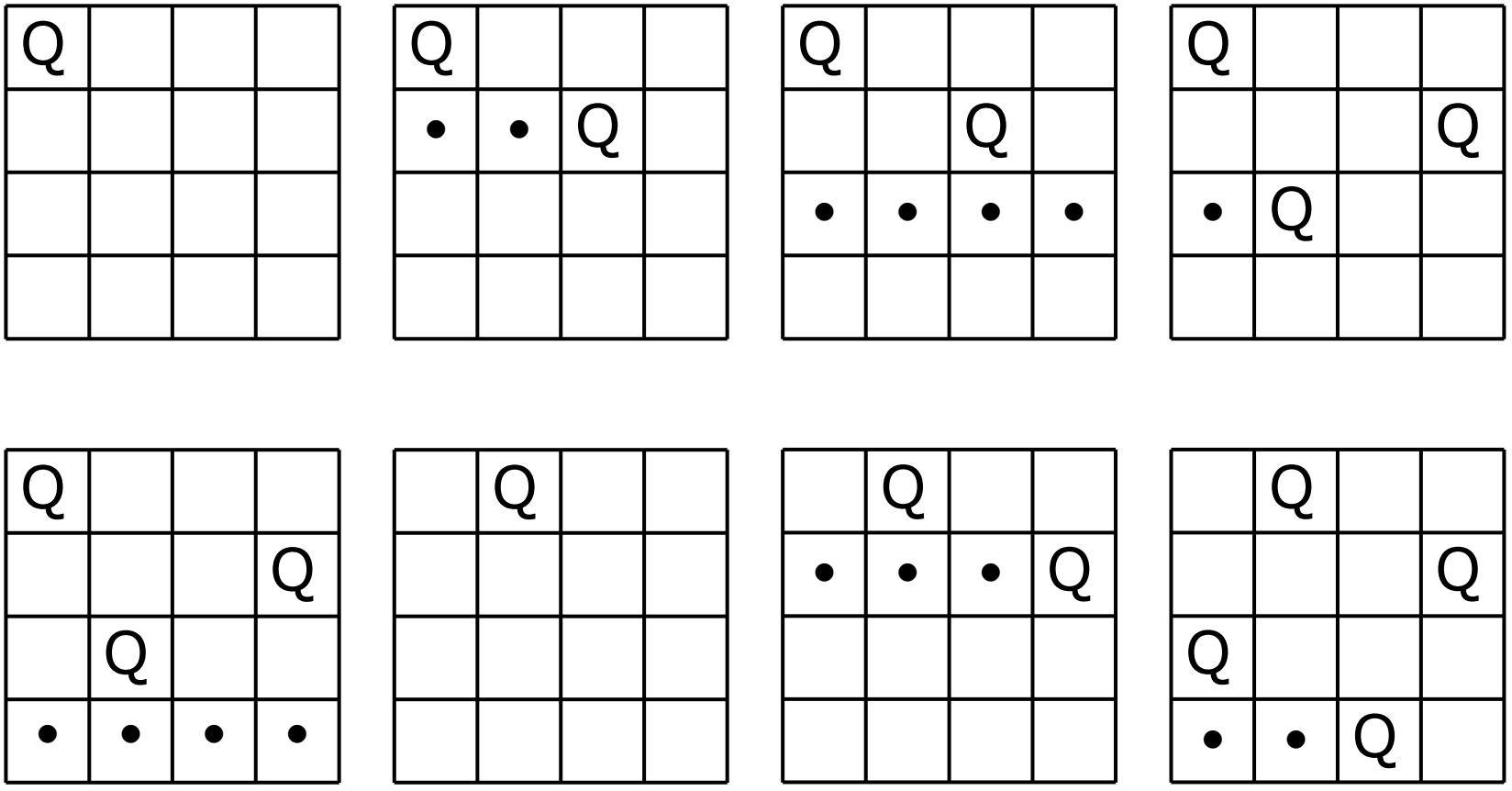
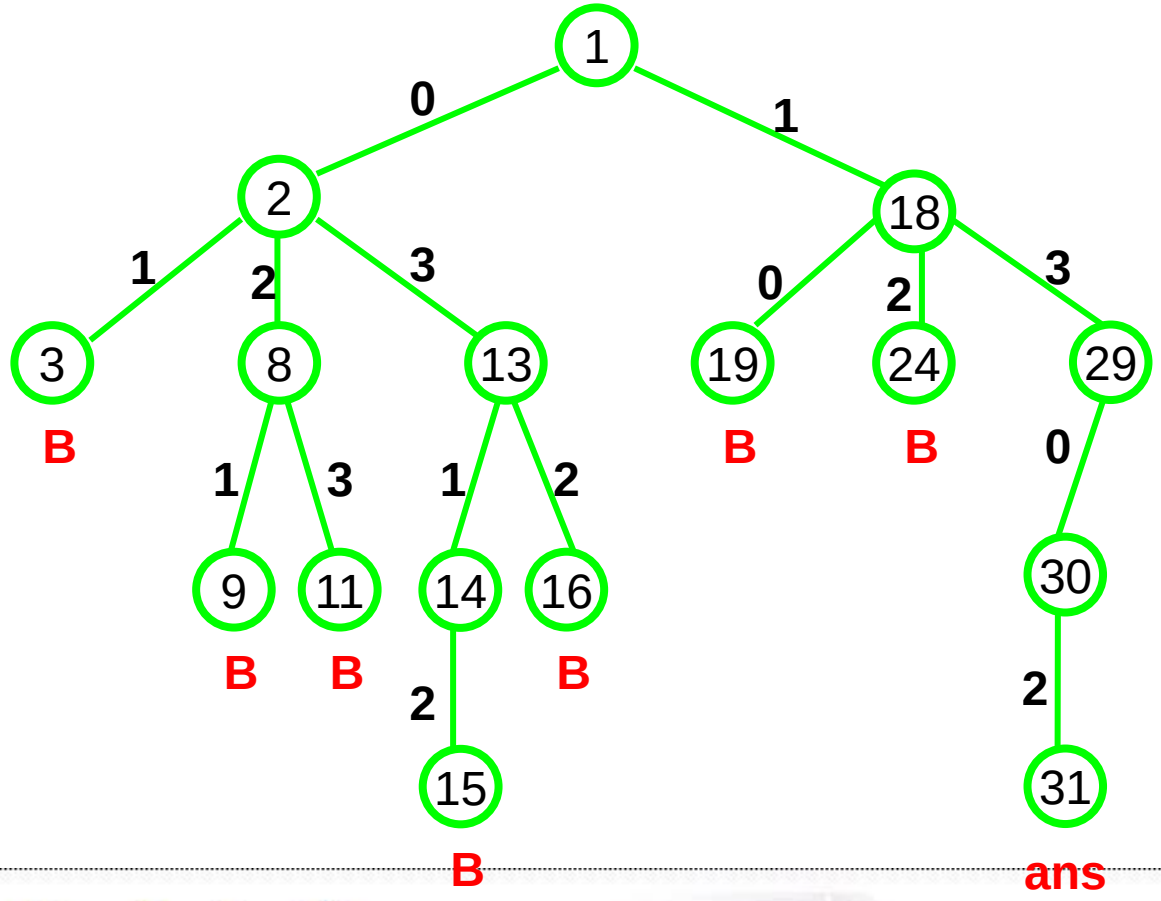


图8-6 显示了4-皇后问题在得到第一个答案状态时，实际生成的那部分状态空间树:（对比 图8-3的状态空间树）



回溯法求4-皇后问题一个可行解的演示

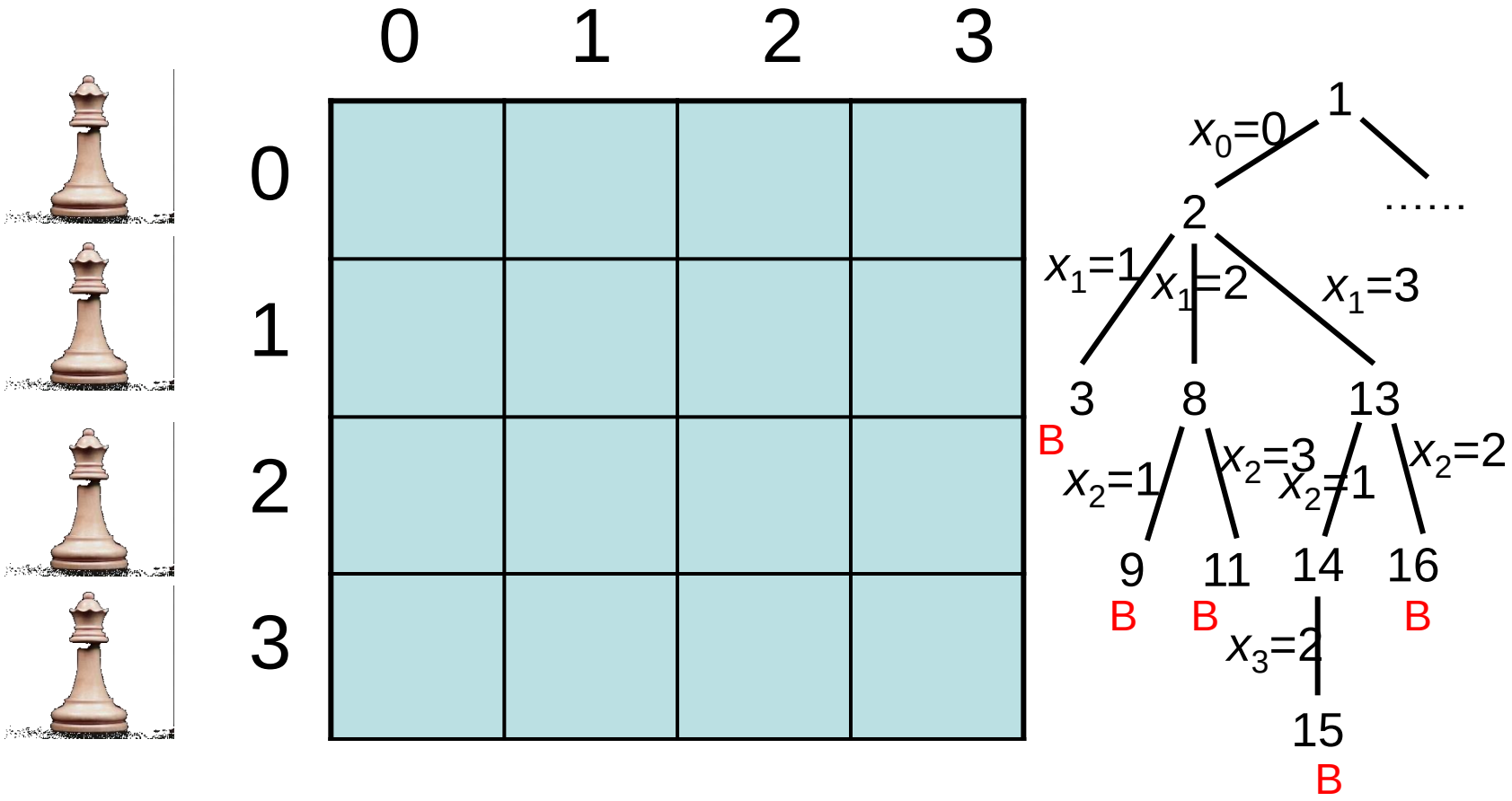


图8-5 回溯法求4-皇后问题的一个可行解

回溯法求4-皇后问题一个可行解的演示

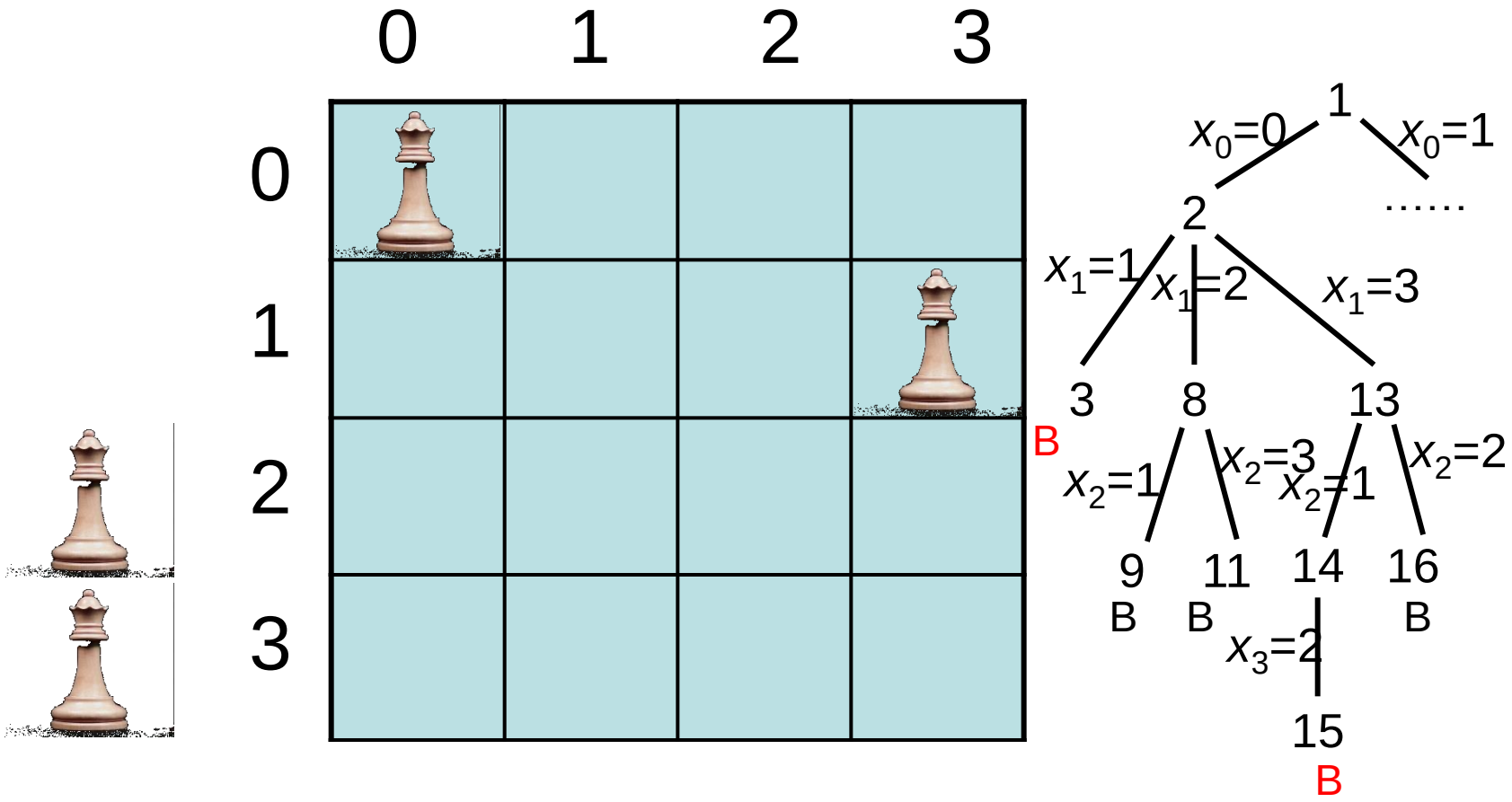
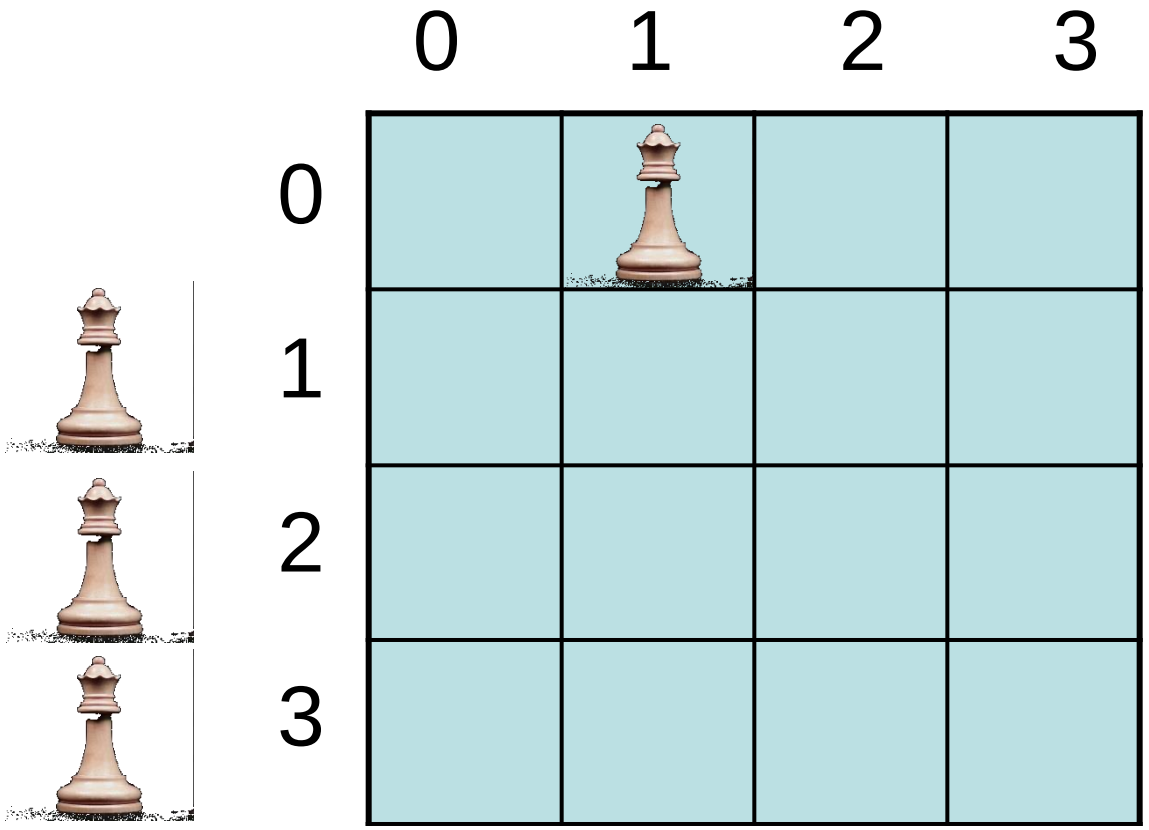
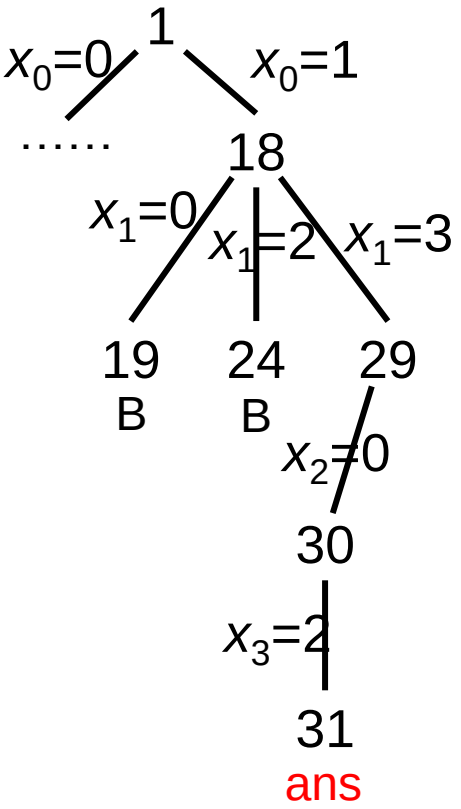


图8-5 回溯法求4-皇后问题的一个可行解

回溯法求4-皇后问题一个可行解的演示





$(x_0,x_1,x_2,x_3)=(1,3,0,2)$

图8-5 的一个可行解

时间分析

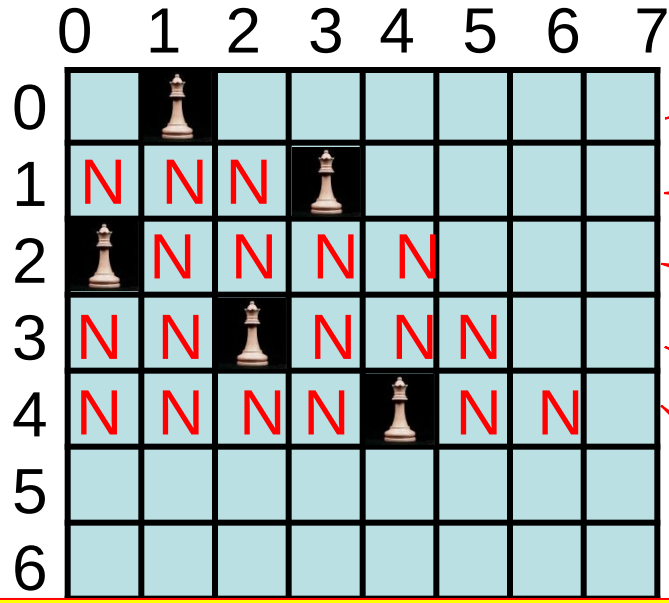
以程序8-4求解8-皇后问题为例，运用蒙特卡罗方法估计实际生成的结点数：

- 如果随机生成的5条路径分别为：(1,3,0,2,4)，(3,5,2,4,6,0)，(0,7,5,2,6,1,3)，(0,2,4,1,3)和(2,5,1,6,0,3,7,4)。
- 选择(1,3,0,2,4)路径时，生成树上实际生成的结点数目为：（见图8-7 a）

  1+8+8*5+8*5*4+8*5*4*3+8*5*4*3*2=1649

  同理，其他几条路径分别实际生成的结点数为：（见图8-7 b-e）769、1401、1977、2329。

随机路径：(1,3,0,2,4)



x_0可取值的列号有8个

x_1有5个

x_2有4个

x_3有3个

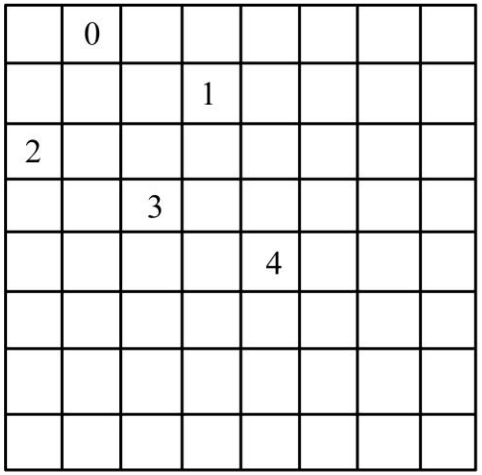
各层不受限制的结点数为(8,5,4,3,2)

根据公式：$m = 1+m_0+m_0m_1+m_0m_1m_2+\ldots+m_0m_1\ldots m_k$

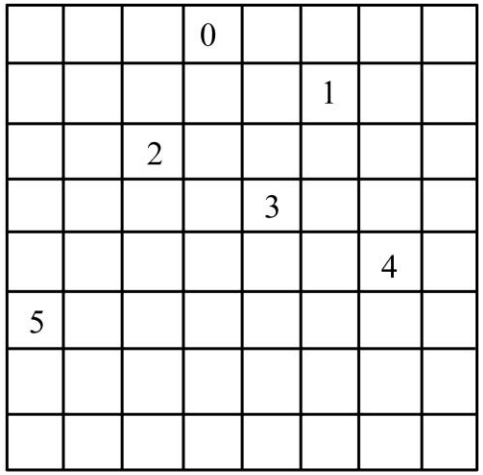
选择(1,3,0,2,4) 路经，状态空间树上实际生成的问题状态数：

(8,5,4,3,2)= 1+8+8*5+8*5*4+8*5*4*3 +8*5*4*3*2

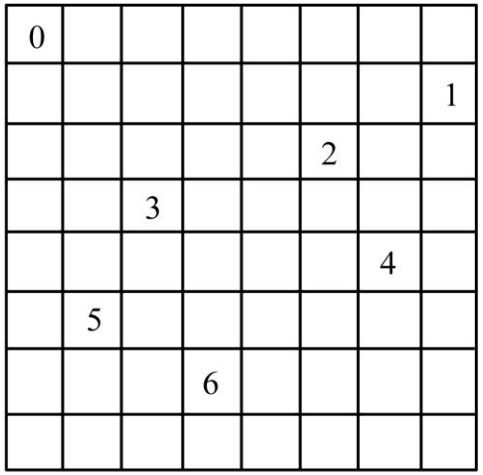
=1649



（a）(8,5,4,3,2)=1649

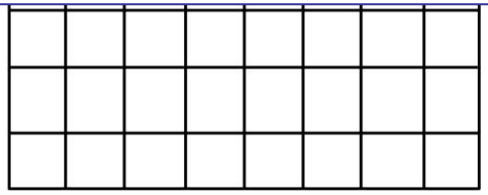


（b）(8,5,3,1,2,1)=769

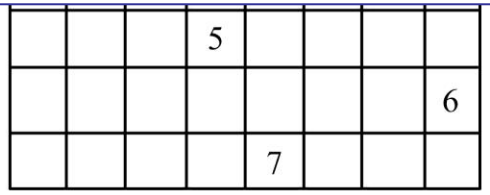


（c）(8,6,4,2,1,1,1)=1401

◆ 因此这5条随机路径实际生成的结点数平均值为1625。
而8-皇后问题状态空间树的结点总数为：
1+8+8*7+8*7*6+.....+8*7*6*5*4*3*2*1=109601
因此，需要实际生成的结点数目大约占总结点数的1.55%。



（d）(8,6,4,3,2)=1977



（e）(8,5,3,2,2,1,1,1)=2329

8.3 子集和数

- 问题描述：已知n个不同正数w_i的集合，求该集合的所有满足条件的子集，使每个子集中正数的和等于一个给定的正数M。

（例8-2）设有n=4个正数的集合$W=\{w_0,w_1,w_2,w_3\}=(11,13,24,7)$和正整数M=31，求W所有满足条件的子集，使得子集中的正数之和等于M。

- 若采用：可变长度k元组$(x_0,x_1,\ldots,x_{k-1})$,$0\le k\le n$表示解，元组的每个分量为入选正数的下标。
- 则　显式约束为：$x_i\in\{j \mid j$是整数且$0\le j<n\}$且$x_i<x_{i+1}(0\le i<n-1)$

（条件$x_i<x_{i+1}$可以避免产生重复子集）

隐式约束为：$\sum_{i=0}^{k-1} w_{x_i} = M$

◆n=4时子集和数问题的（可变长度元组解）状态空间树见下图：

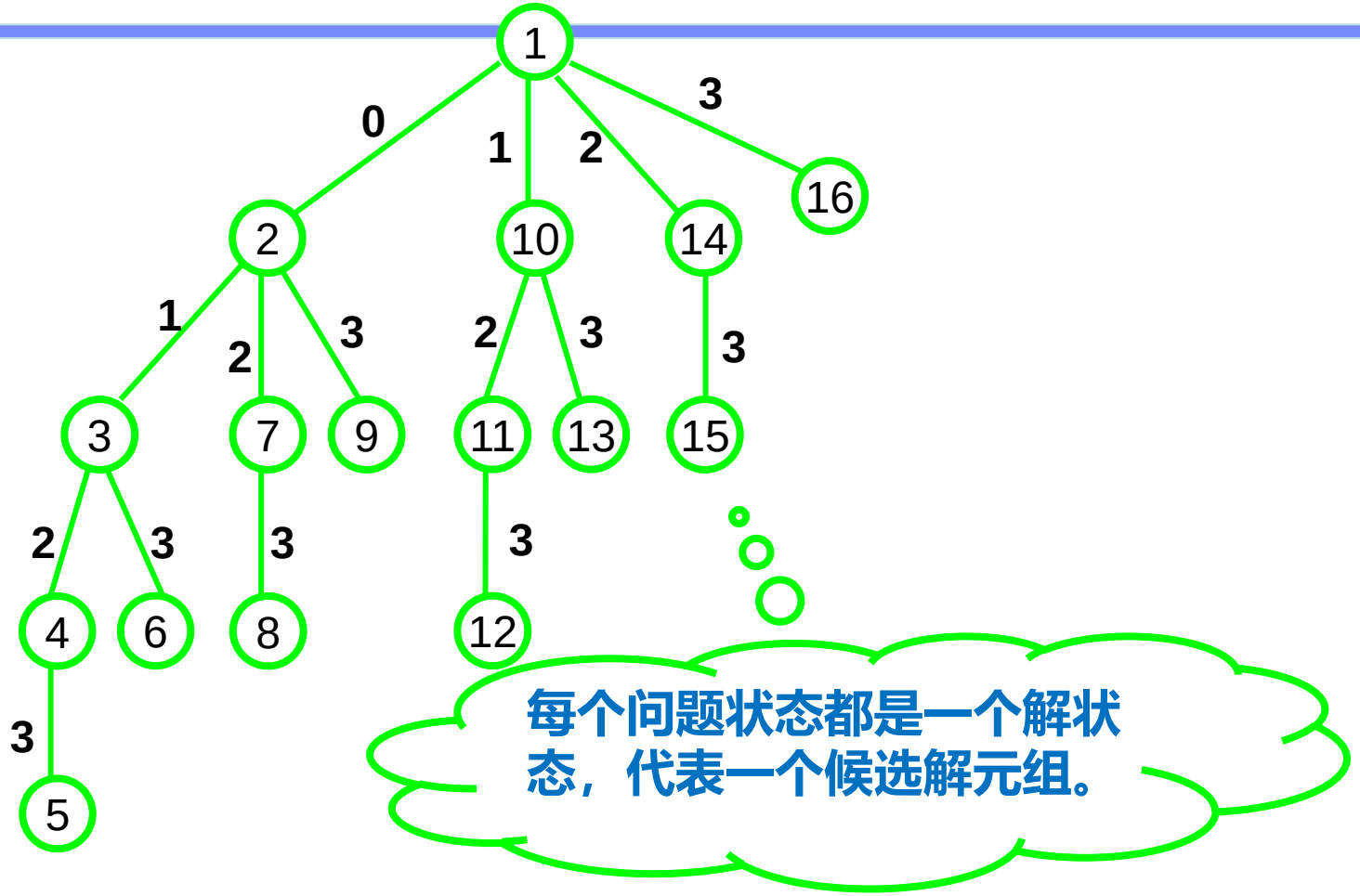


图8-8　子集和数问题（可变元组解）状态空间树

（例8-2）设有n=4个正数的集合 $W=\{w_0,w_1,w_2,w_3\}=(11,13,24,7)$和正整数M=31，求W所有满足条件的子集，使得子集中的正数之和等于M。

- 若采用：固定长度n元组$(x_0,x_1,\ldots,x_{n-1})$表示解，$x_i\in\{0,1\},0\le i<n$。

  （$x_i=0$，表示w_i未入选子集；$x_i=1$，表示w_i入选子集。）

- 则　显式约束为：$x_i\in\{0,1\}\ (0\le i<n-1)$

隐式约束为：$\sum_{i=0}^{n-1} w_i x_i = M$

n=4时子集和数问题（固定长

每个叶结点是一个解状态，
代表一个候选解元组。
非叶结点代表部分向量。

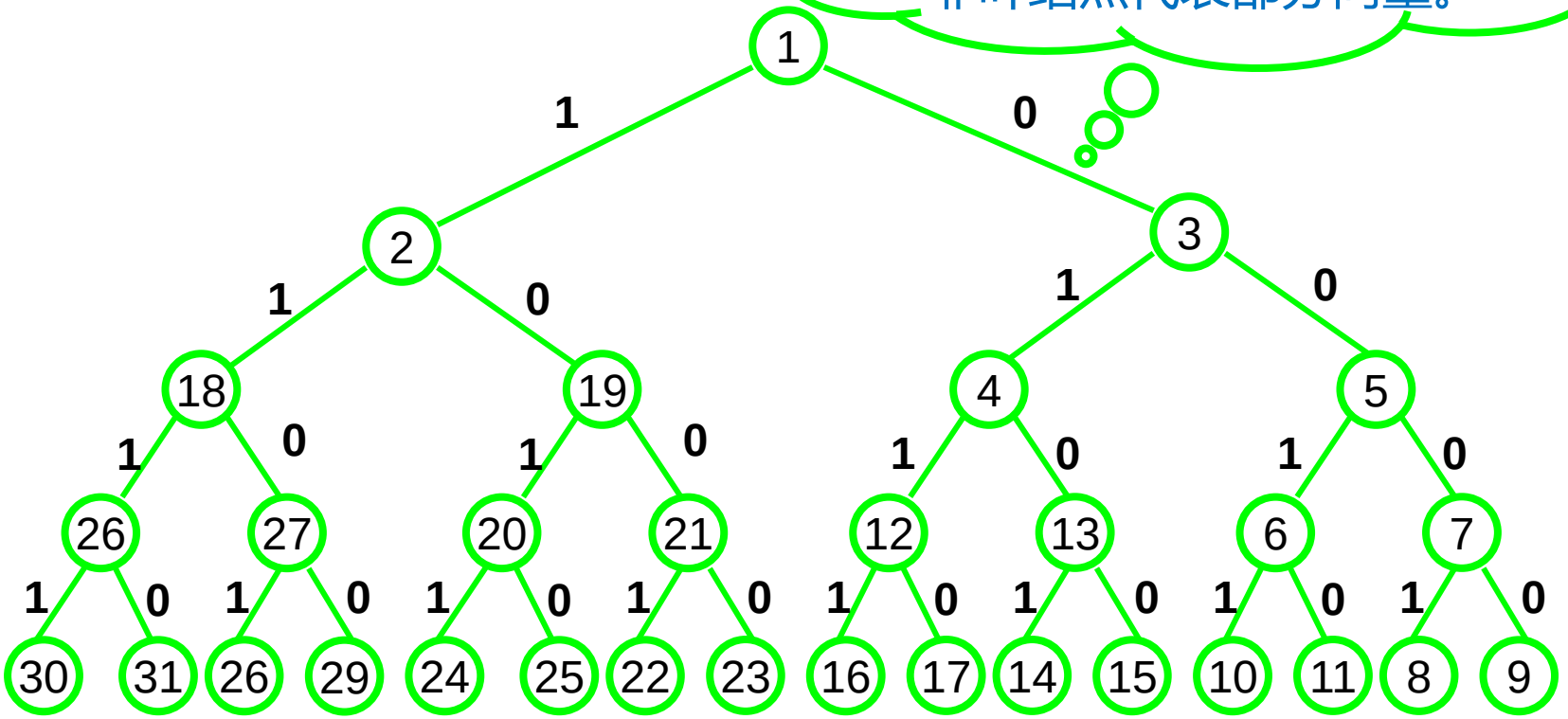


图8-9　子集和数问题（固定长度元组解）状态空间树

一般称这种从n个元素的集合中找出满足某些性质的子集的状态空间树为子集树。

（采用固定长度元组解）每次选择x_k之前应判断是否满足约束函数$B_k(x_0,x_1,\ldots,x_k)$：（若w_i已按非减次序排列）

$$\sum_{i=0}^{k-1} w_i x_i + \sum_{i=k}^{n-1} w_i \geq M \quad \text{且} \quad \sum_{i=0}^{k-1} w_i x_i + w_k \leq M$$

程序8-5（见下页）的前置条件：

$$s + r \geq \mathrm{M} \;\&\&\; s + w_k \leq \mathrm{M}$$

$$\left(s = \sum_{i=0}^{k-1} w_i x_i, \quad r = \sum_{i=k}^{n-1} w_i\right)$$

和后置条件：

在以$(x_0,x_1,\ldots,x_k)$为根的子树上搜索答案结点。

程序8-5 子集和数的回溯算法

```
void SumOfSub(float s,int k,float r,int *x,float m,float *w)
//s为已选择的元素和，r为剩余元素和，k为当前层数（即将选择的元素下标）
//m为子集数和，w为权值元组，x为解元组
{x[k]=1;
 if (s+w[k]==m) {                    //x[k]=1若得到可行解
      for (int j=0;j<=k;j++)
      }
 else if (s+w[k]+w[k+1]<=m)   //x[k]=1若没得到可行解，而约束函数B_{k+1}为真
        SumOfSub(s+w[k],k+1,r-w[k],x,m,w); //则沿左子树向第k+1层搜索
 if ((s+r-w[k]>=m)&&(s+w[k+1]<=m)) {  //x[k]=0，同时约束函数B_{k+1}为真
      x[k]=0;
```

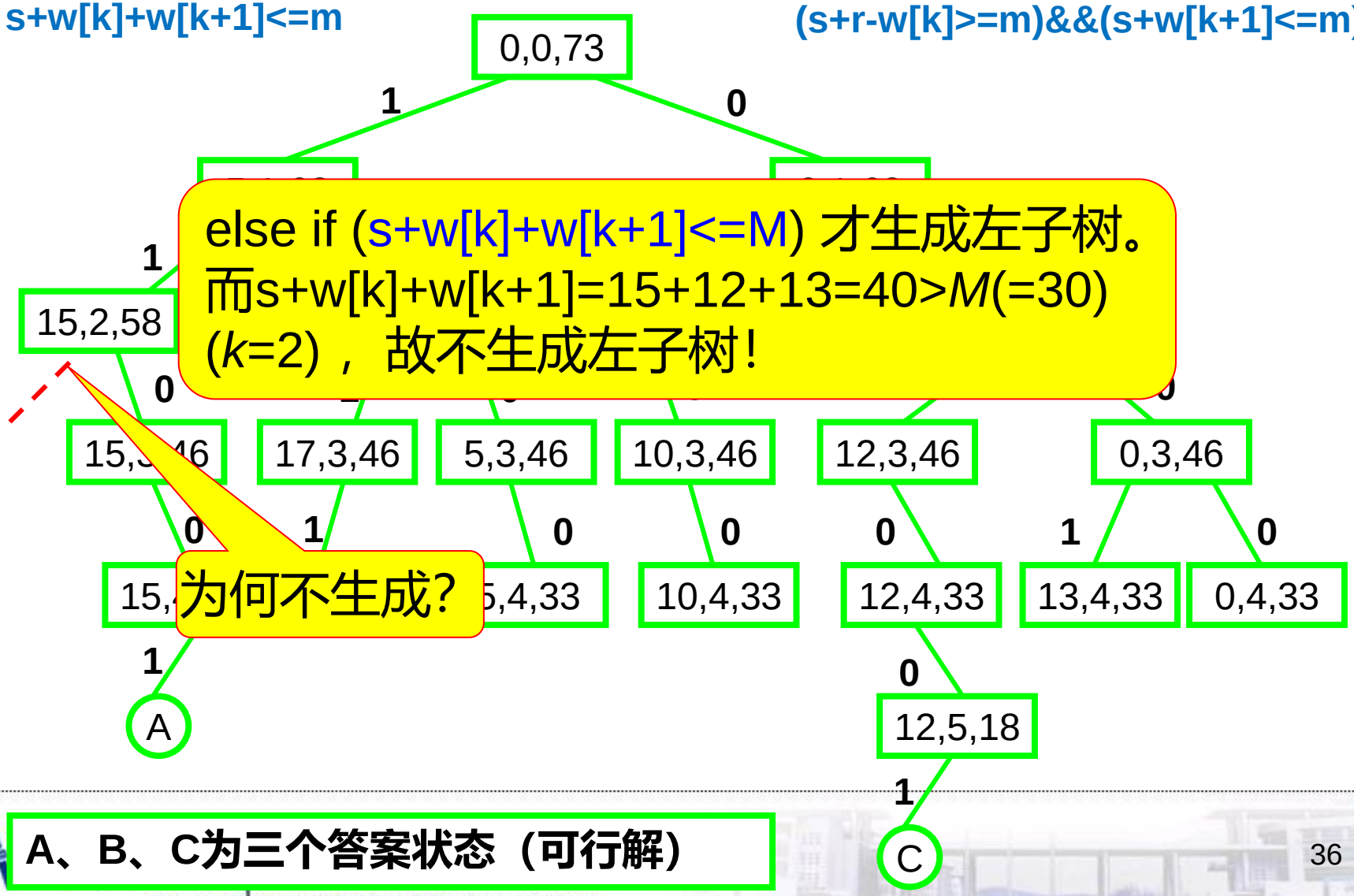
初始条件s=0，
r>=m，w_0<=m。

因(s+w[k])+(r-w[k])=s+r值
未变，因此省略判断。

由于进入第k层之前，总满足前置条件：$s+w_{k-1}\le m$且$s+r\ge m$。 而进入第n-1层（最后一层）又有$r=w_{n-1}$。因此必有$s+w_{n-1}=s+r=m$。所以，只要进入第n-1层，函数总能终止；否则不可能进入第n-1层。

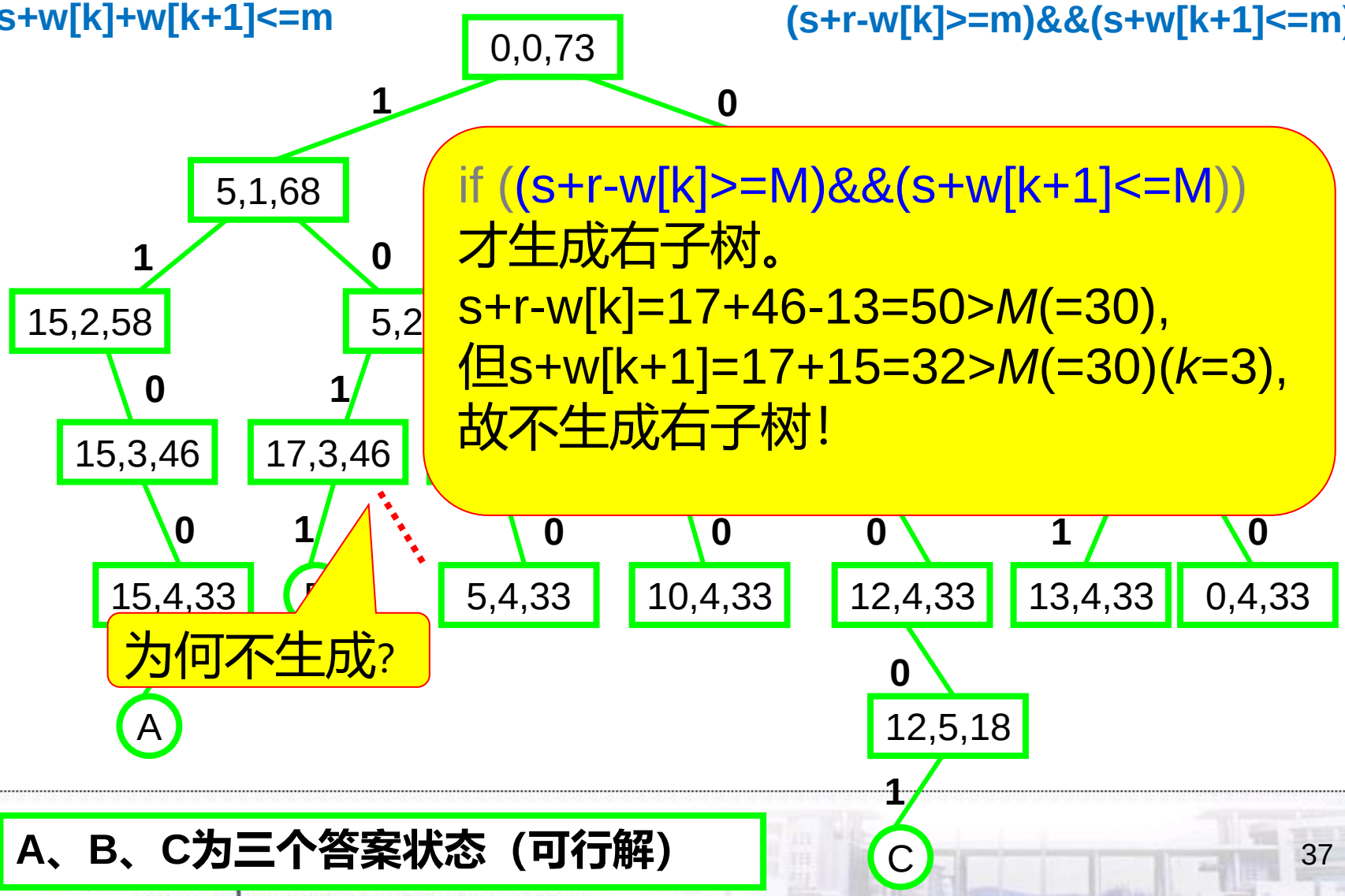
0 1 2 3 4 5

（例8-3） 设有n=6个正数的集合W={5,10,12,13,15,18}和整数M=30，求W的所有元素之和为M的子集。



A、B、C为三个答案状态（可行解）

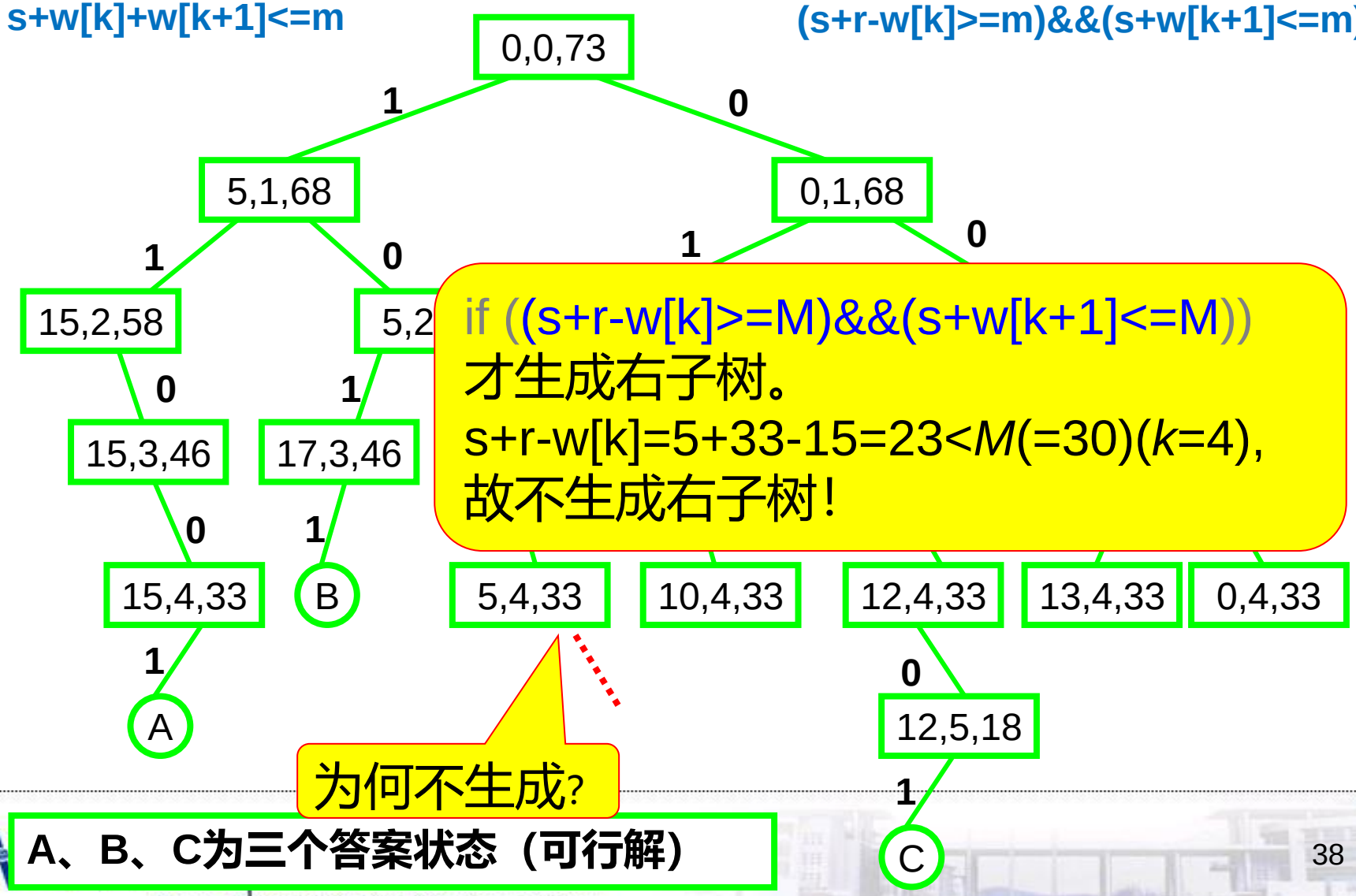
（例8-3） 设有n=6个正数的集合W={5,10,12,13,15,18}和整数M=30，求W的所有元素之和为M的子集。



A、B、C为三个答案状态（可行解）

（例8-3） 设有n=6个正数的集合W={5,10,12,13,15,18}和整数M=30，求W的所有元素之和为M的子集。

0 1 2 3 4 5



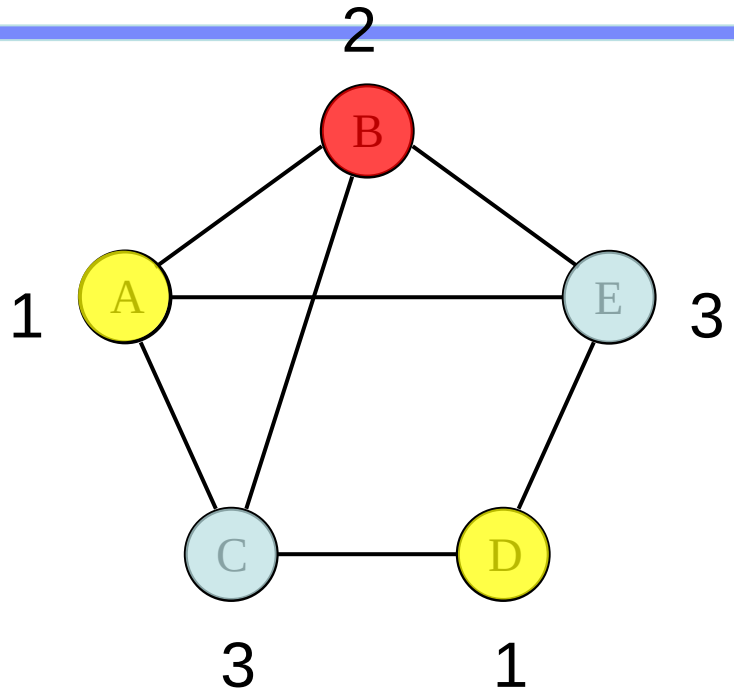
A、B、C为三个答案状态（可行解）

8.4 图的着色

8.4.1 问题描述

- 已知无向图G=(V,E)和m种不同的颜色，如果只允许使用这m种颜色对图G的结点着色，每个结点着一种颜色。问是否存在一种着色方案，使得图中任意相邻的两个结点都有不同的颜色。这就是**m-着色判定问题**（m-colorability decision problem）

例如：



1—黄
2—红
3—蓝

可用三种颜色着色，结点边上的数字代表颜色编号。

图8-11 图着色的例子

四色定理

每幅（无飞地的）地图都可以用不多于4种颜色来着色，使得有共同边界的国家着不同的颜色。

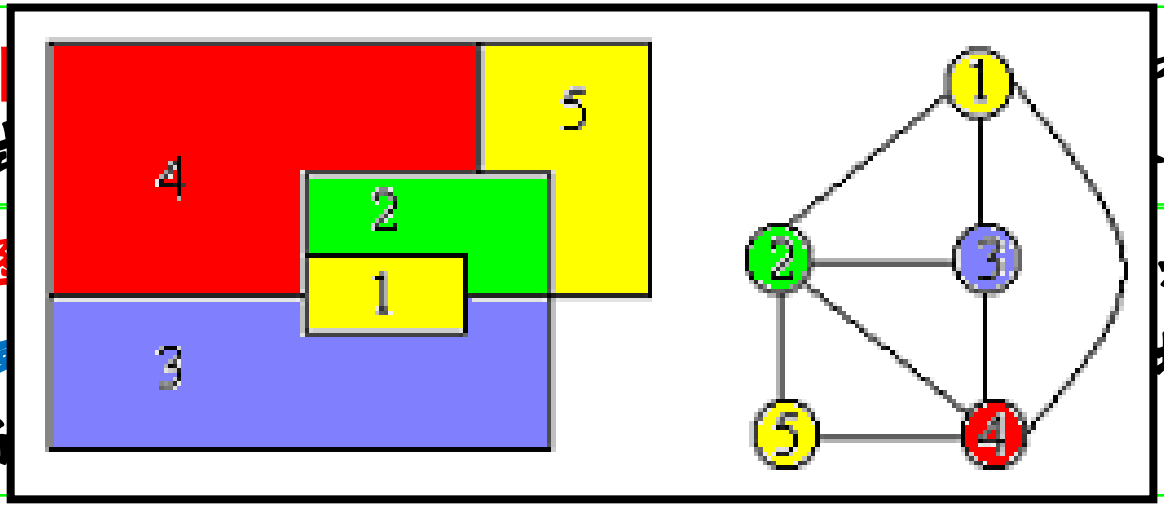
平面图(

某种方式

一幅地图

每个区域

则相应的



都能用

边相交。

地图的

域相邻，

——平面图的4-着色判定问题

8.4.2 回溯法求解

- 设无向图G=(V,E)采用如下定义的邻接矩阵a表示：

$$a[i][j]=\begin{cases}1 & \text{如果}(i,j)\in E\\ 0 & \text{其他}\end{cases} \qquad (8-8)$$

- 解的形式：n-元组$(x_0,x_1,\cdots,x_{n-1})$，$x_i\in\{1,\ldots,m\}$，$0\le i<n$，表示结点i的颜色，这就是显式约束。$x_i=0$表示没有可用的颜色。因此解空间的大小为m^n。
- 隐式约束可描述为：如果边$(i,j)\in E$，则$x_i\neq x_j$。

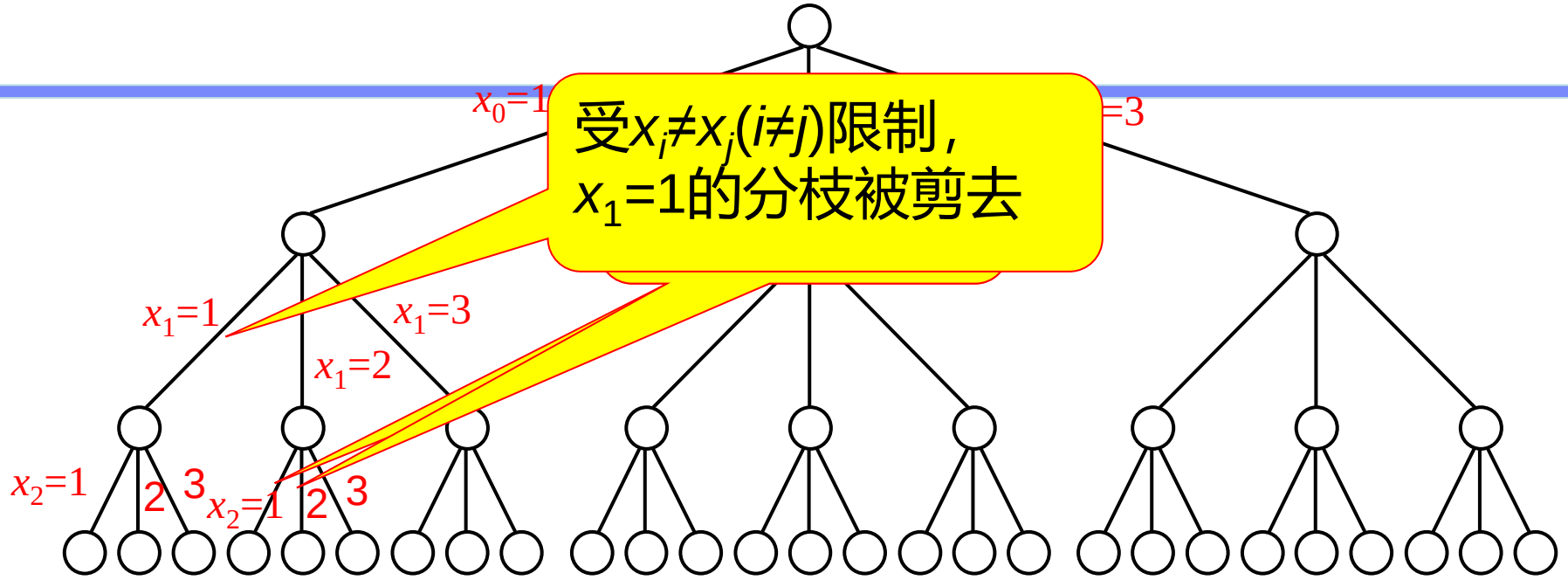


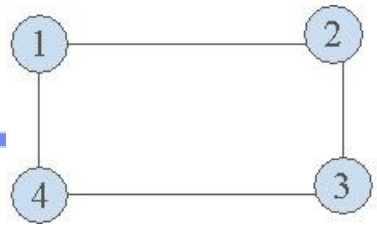
图8-13 n=3,m=3的图的m-着色判定问题的状态空间树

约束函数：

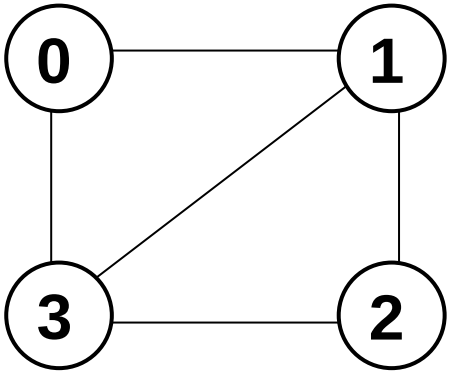
对所有顶点i和j（$0 \leq i, j < n, i \neq j$），若a[i][j]=1，则$x_i \neq x_j$。

（$1 \leq x_i, x_j \leq m$）

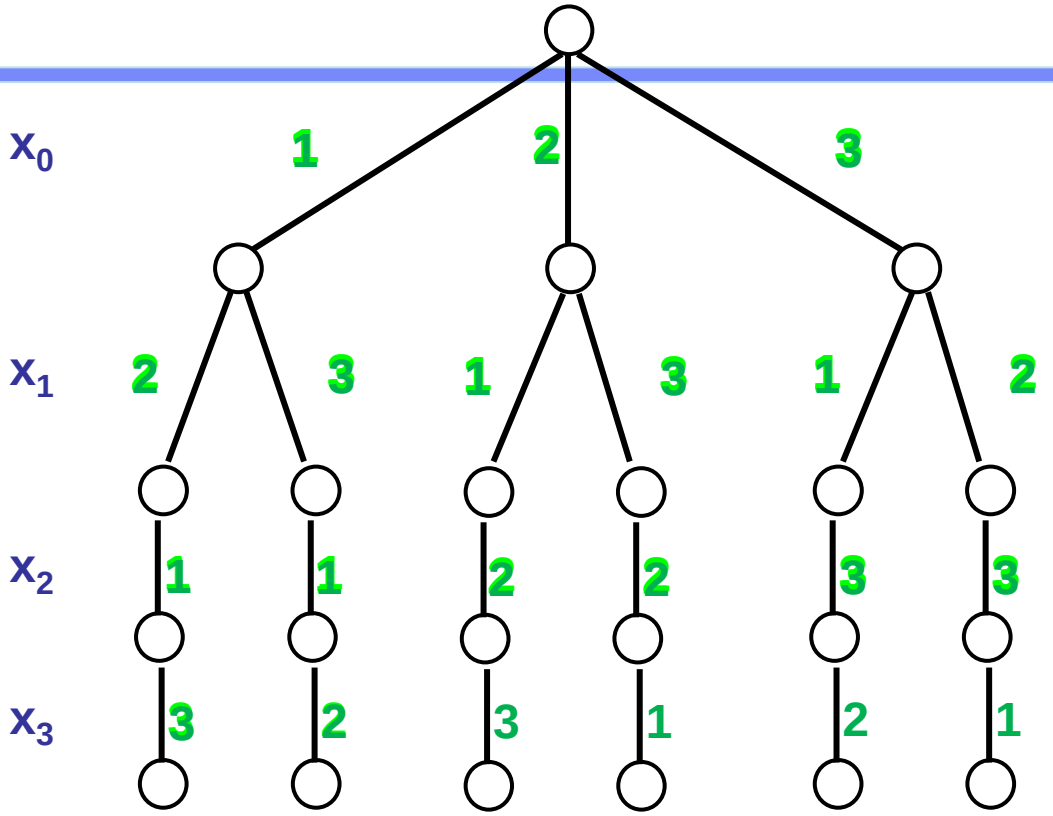
例子：n＝4，m＝3，无向图G如图



如：



4个结点的图G



图G所有可能的3-着色方案

m-着色算法实现

- 以深度优先方式生成状态空间树中的结点，寻找所有答案结点，即m-着色方案。
- 搜索中使用约束函数剪去不可能包含答案结点的分枝。
- 对给定的无向图G和m，列出图中结点所有可能的m-着色方案。

- 图类MGraph用邻接矩阵表示图；
- 函数mColoring为该类的公有成员函数；
- 递归函数mColoring和NextValue为该类的私有成员函数。（NextValue为约束函数。）

程序8-6 图的m-着色

m-着色方案中指定的最多可用颜色数

```
template <class T>
void MGraph<T> :: mColoring(int m,int *x)
{
  mColoring(0,m,x);     //从为x[0]分配
}
```

$(x_0,\ldots,x_{n-1})$的初始值均为0

程序8-6 图的m-着色算法

```
template <class T>
void MGraph<T> :: mColoring(int k,int m,int *x)          //递归函数
{
  do
  {
    NextValue(k,m,x);   //（约束函数）每次为x[k]分配一个颜色，直到
    if (!x[k]) break;   //x[k]已无适当颜色可选，则向上回溯
    if (k==n-1)         //得到图G的一种m-着色方案
    {
       for (int i=0;i<n;i++) cout<<x[i]<<'';
       cout<<endl;
    }
    else
       mColoring(k+1,m,x);
                        //深度优先搜索下一层，此时前k个结点已分配了颜色
  } while(1);
}
```

返回本层循环继续为x[k]分配颜色，搜索其它可行着色方案

向上回溯

程序8-6 图的m-着色算法

```
template <class T>
void MGraph<T> :: NextValue(int k,int m,int *x)
                                        //约束函数，每次为x[k]分配一个颜色
{ //在[1,m]中为x[k]确定一个值最小的，且不与其邻接点冲突的颜色
  //颜色从1开始编号，若x[k]=0表示没有可用颜色
  do
  {
      x[k]=(x[k]+1)%(m+1);          //为x[k]尝试下一种颜色
      if(!x[k]) return;             //直到尝试完所有m种颜色，没有可用的颜色为止
      for (int j=0;j<k;j++)         //检验当前选择的x[k]是否会造成冲突
          if (a[k][j]&&x[k]==x[j])  //若(k,j)是图的边，且结点k和j颜色相同
              break;                //产生冲突，尝试下一种颜色
      if (j==k) return;  //x[k]与已选定的结点颜色x[0]~x[k-1]无冲突，成功选择
  } while (1);
}
```

算法的计算时间上界，由状态空间树的结点数 $\sum_{i=0}^{n-1} m^i$ 确定。

每个结点的处理时间即NextValue的执行时间O(mn)。

因此：

总时间为 $$\sum_{i=1}^{n} m^i n = n\frac{(m^{n+1}-m)}{m-1} = O(nm^n)$$

8.5 哈密顿环

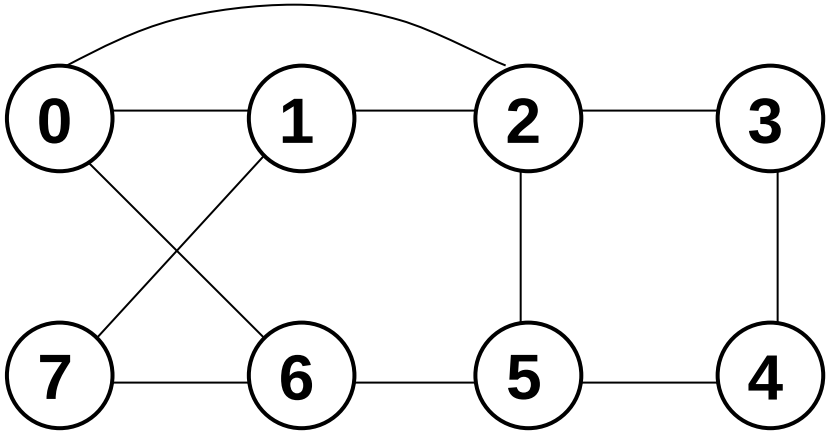
◆**哈密顿环**（Hamiltonlan cycle）：

连通图G=(V,E)中的一个回路，经过图中每个顶点，且只经过一次。

一个哈密顿环就是从某个结点v_0开始，沿着图G的n条边环行的一条路径$(v_0,v_1,\ldots,v_{n-1},v_n)$。

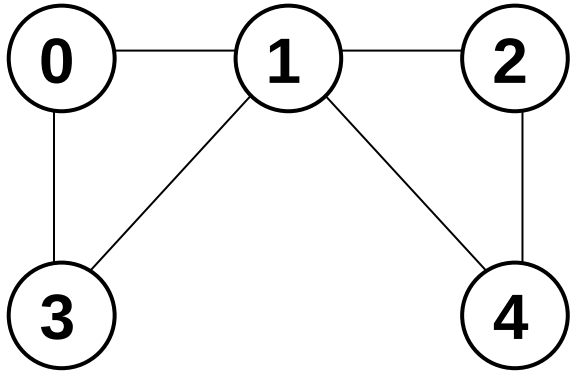
- 除$v_0=v_n$外，路径上其余结点各不相同。
- $(v_i,v_{i+1}) \in E \quad (0 \le i < n)$。
- 它访问图中每个结点且仅访问一次，最后返回开始结点。

并不是每个连通图都存在哈密顿环!



图G_1包含哈密顿环

(0,1,7,6,5,4,3,2,0)



图G_2不包含哈密顿环

要确定一个连通图是否存在哈密顿环也没有容易的办法。

采用n-元组$(x_0,x_1,\ldots,x_{n-1})$表示哈密顿环问题的解。

显示约束：$x_i \in \{0,1,\ldots,n-1\}$, $0 \le i < n$, 代表路径上一个结点的编号 。

隐式约束：$x_i \neq x_j (0 \le i, j < n, i \neq j)$，且$(x_i,x_{i+1}) \in E$ $(i=0,1,\ldots,n-2)$，又$(x_{n-1},x_0) \in E$。

解空间大小为n^n。

哈密顿环算法实现

- 求一个无向图或有向图的所有哈密顿环，并输出所有不相同的环。
- 图类MGraph用邻接矩阵表示图；
- 指定$x_0=0$，即哈密顿环始终从顶点0开始，避免同一环被重复多次输出。
- 因此公有成员函数Hamiltonian中以Hamiltonian(1,x)调用Hamiltonian私有递归函数；
- NextValue为约束函数。
- 对最后一个结点x_{n-1}，还需进一步要求$(x_{n-1},x_0)\in E$，才能形成哈密顿环。

程序8-7 哈密顿环算法

$(x_0,\ldots,x_{n-1})$的初始值均为0

```
template <class T>
void MGraph<T> :: Hamiltonian(int *x)
{
   Hamiltonian(1,x);     //x[0]=0为约定的起始结点，不需选择。
                         //因此从x[1]开始逐个选择x[1]~x[n-1]的结点编号
}
```

程序8-7 哈密顿环算法

```
template <class T>
void MGraph<T> :: Hamiltonian(int k, int *x)    //递归函数
{
  do
  {
    NextValue(k,x);  //约束函数每次产生x[k]的下一个值，直到
    if (!x[k]) return;  //x[k]已无可选值时向上回溯
    if (k==n-1)         //哈密顿环中已包含图中所有的n个顶点，则输出该环
    {
       for (int i=0;i<n;i++) cout<<x[i]<<"";
       cout<<" 0/n";  //哈密顿环的最后一个顶点，又回到起始点0
    }
    else
      Hamiltonian(k+1,x);
                        //深度优先搜索下一层，此时前k个结点编号已确定
  } while(1);
}
```

返回本层循环继续为x[k]选择结点编号，搜索其它可行着色方案

向上回溯

程序8-7 哈密顿环算法

```
template <class T>
void MGraph<T> :: NextValue(int k,int *x)  //约束函数，产生x[k]的下一个值
{ //在[1,n]中选择路径上的下一个结点编号x[k]
  //满足(x[k-1],x[k])∈E，且x[k]与前k个已经选择的结点不同。
  do
  { x[k]=(x[k]+1)%n;     //产生x[k]的下一个值（结点编号）
    if(!x[k]) return;     //直到尝试完所有n个结点，没有可选的结点编号为止
    if ( a[x[k-1]x[k]] )  //若(x[k-1],x[k])是图中的一条边
    {
      for (int j=0;j<k;j++)     //检验当前选择的x[k]与前k个结点是否相同
        if (x[j]==x[k]) break;  //若x[k]与前k个结点有重复，尝试下一个结点
      if (j==k)      //否则表示当前结点编号x[k]与前k个结点均不重复
        if ( (k<n-1)||((k==n-1)&&a[x[n-1]][x[0]]) )      //（进一步判断）
          return;     //若x[k]的确可取，则取x[k]当前值
      //否则舍弃当前编号x[k]，尝试下一个结点
    }
  } while (1);
}
```

取值成功

取值失败

算法的计算时间上界估算，请参照图的着色问题!

8.6 0/1背包

0/1背包问题是一个**困难问题**（难以设计最坏情况下可多项式时间求解的算法）。

- n-皇后问题、子集和数问题、m-图着色问题、哈密顿环问题的求解目标都是求满足约束条件的全部可行解。
- 0/1背包问题是最优化问题。

回溯法本质上是一种深度优先搜索状态空间树的算法。

- 如果不引入剪枝函数(约束函数+限界函数)，则是穷举算法。
- 引入适当的限界函数，剪去已能确信不含最优答案结点的子树，使其成为一种启发式算法。

●本节讨论采用固定长度元组解结构的 0/1背包问题的回溯算法。

●0/1背包问题的解用n-元组表示：$X=(x_0,x_1,\ldots,x_{n-1})$

x_i=0或1 (0≤i<n)

显示约束：x_i=1表示将第i件物品装入背包，x_i=0表示第i件物品不装入背包。

隐式约束：$\sum_{i=0}^{n-1} w_i x_i \le M$

解空间大小为2^n。解

对左孩子(x_k=1)起约束函数作用，可剪去不含可行解的分枝；
对右孩子(x_k=0)不需用约束函数判断，一定可行。

约束函数：$B_k(x_0,x_1,\ldots,x_k)=true$,当且仅当$\sum_{i=0}^{k-1} w_i x_i + w_k \le M$

剪去不含可行解的分枝

下界函数：变量L初值为0，遇到一个答案结点便计算该答案结点的收益值fp,且令L=max{L,fp}。则L中始终保存迄今为止已经搜索到的答案结点中收益的最大值，0/1背包的最优解值必定大于等于L，因此最优解值的下界估计值为L。

限界函数：状态空间树中任一结点，若其上界函数值bp<最优解值的下界估计值变量L，则可断定X子树上不含[illegible]答案结点，可以剪去以X为根的子树。

上界函数：当前位于状态空间树的结点X处，cw为背包当前重量，cp为当前已装入背包物品的总收益，则贪心法求解剩余载重和剩余物品构成的一般背包问题（物品编号k+1,k+2,...,n-1,载重M-cw）最大收益为rp。则以X为根的子树上所有可能答案结点的目标函数值 $cp+\sum_{k+1\le i<n} p_i x_i$ 不可能超过bp=cp+rp，因此结点X的上界函数值为bp。

例8-4 设有0/1背包n=8, M=110，

$(w_0,w_1,\ldots,w_7)=(1,11,21,23,33,43,45,55)$,

$(p_0,p_1,\ldots,p_7)=(11,21,31,33,43,53,55,65)$。

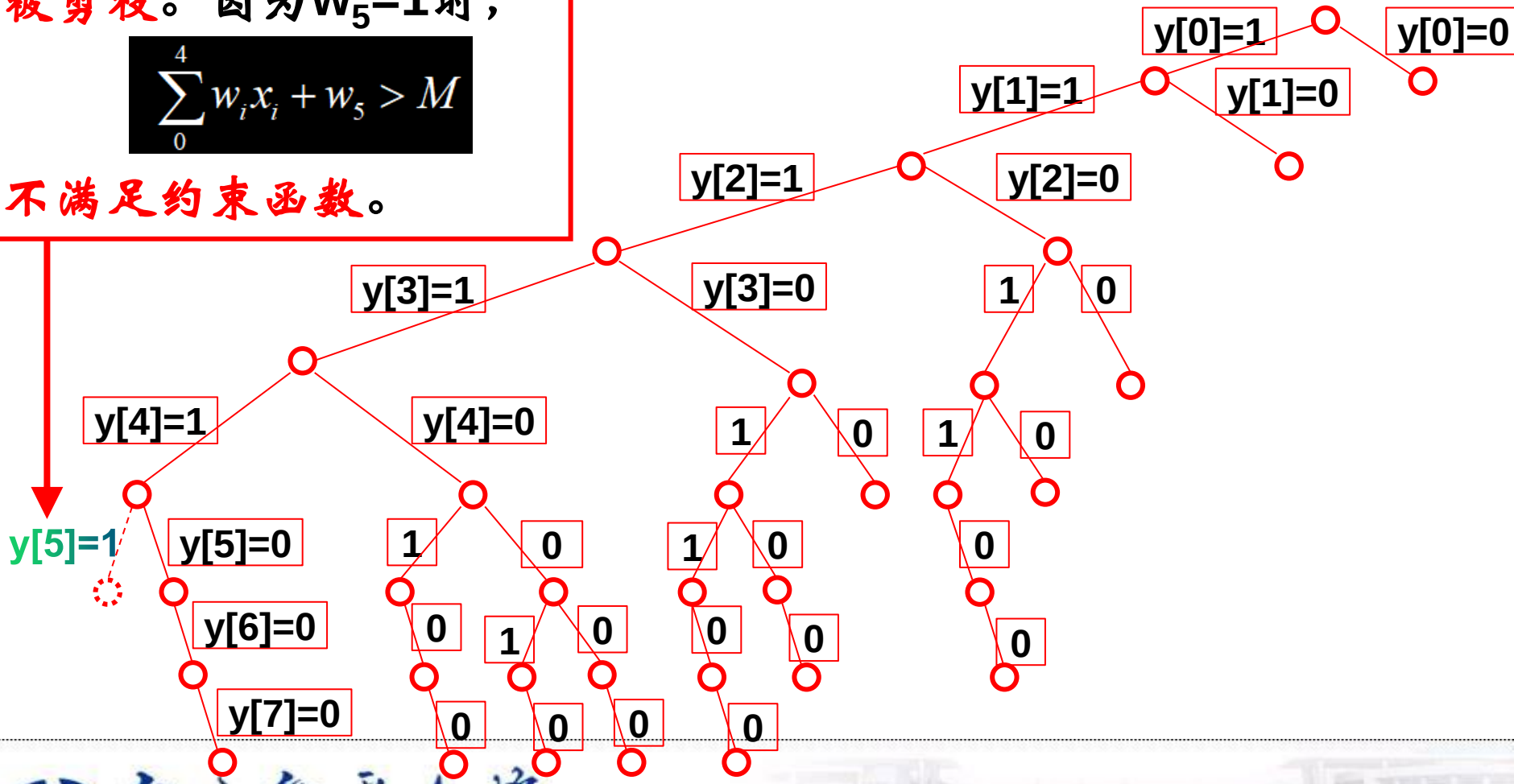
——按p_i/w_i非增排列， 即$p_i/w_i \geq p_{i+1}/w_{i+1}$

$(w_0,w_1,\ldots,w_7)=(1,11,21,23,33,43,45,55)$,
$(p_0,p_1,\ldots,p_7)=(11,21,31,33,43,53,55,65)$，M=110。

被剪枝。因为$w_5=1$时，

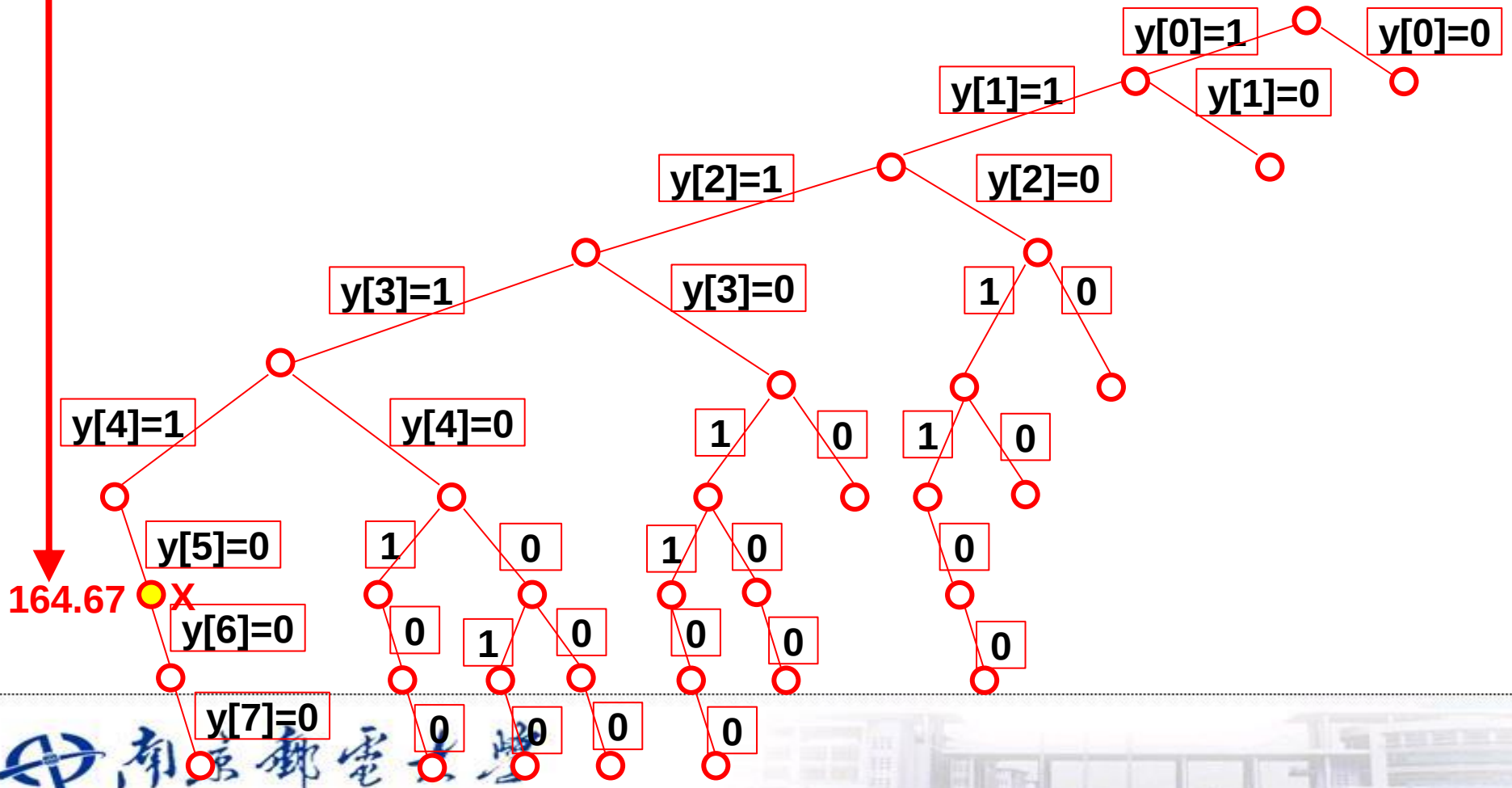
$$\sum_{0}^{4} w_i x_i + w_5 > M$$

不满足约束函数。



$bp=cp+rp=p_0*1+p_1*1+p_2*1+p_3*1+p_4*1+p_5*0+p_6/w_6*(110-89)$
$=(11+21+31+33+43)+55/45*21=139+25.67=164.67$

为当前结点X的上界函数值。若bp<当前最优解值下界估计值L，则可断定X子树上不含最优答案结点，可以剪去以X为根的子树。

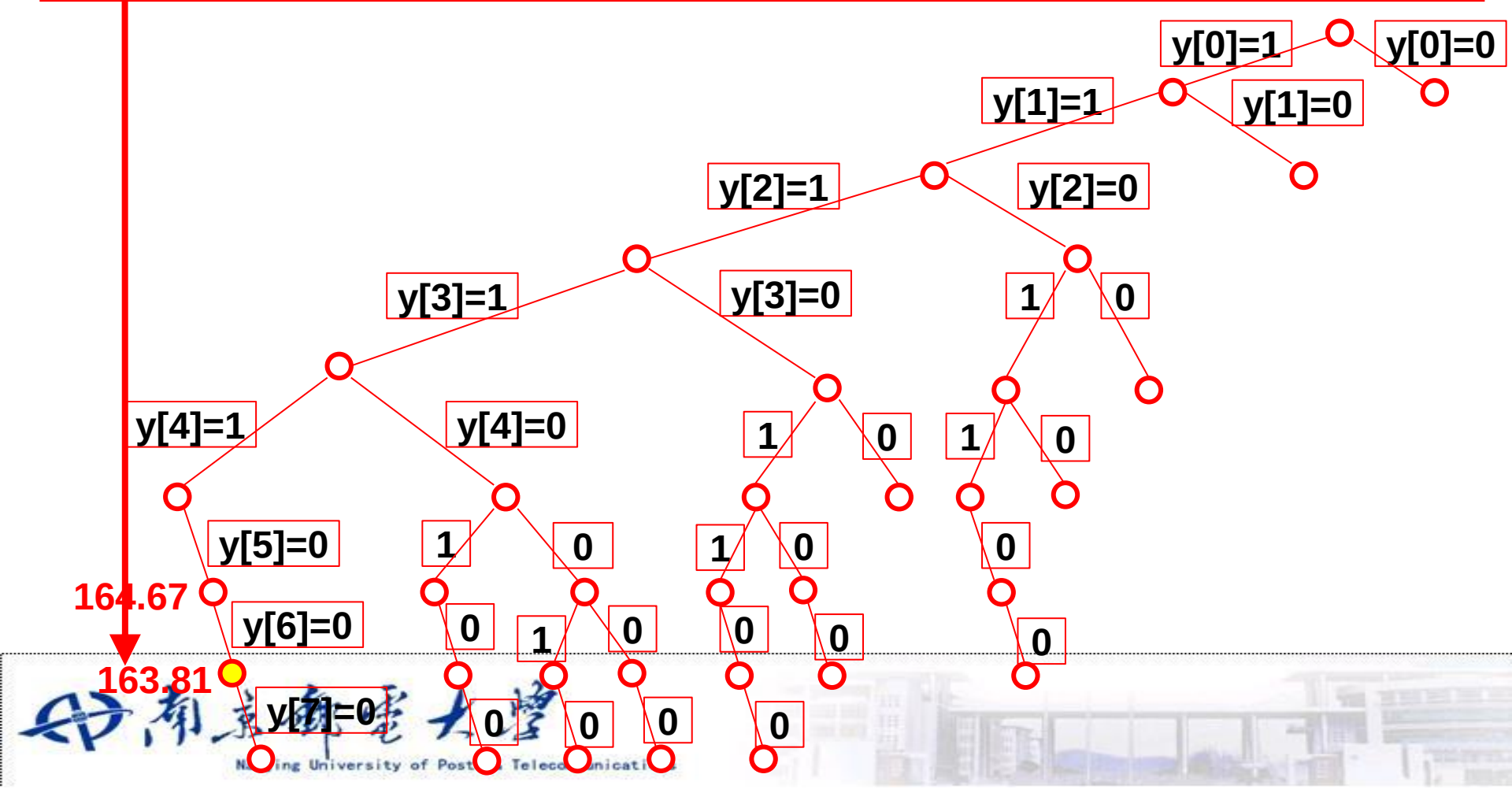


bp=cp+rp

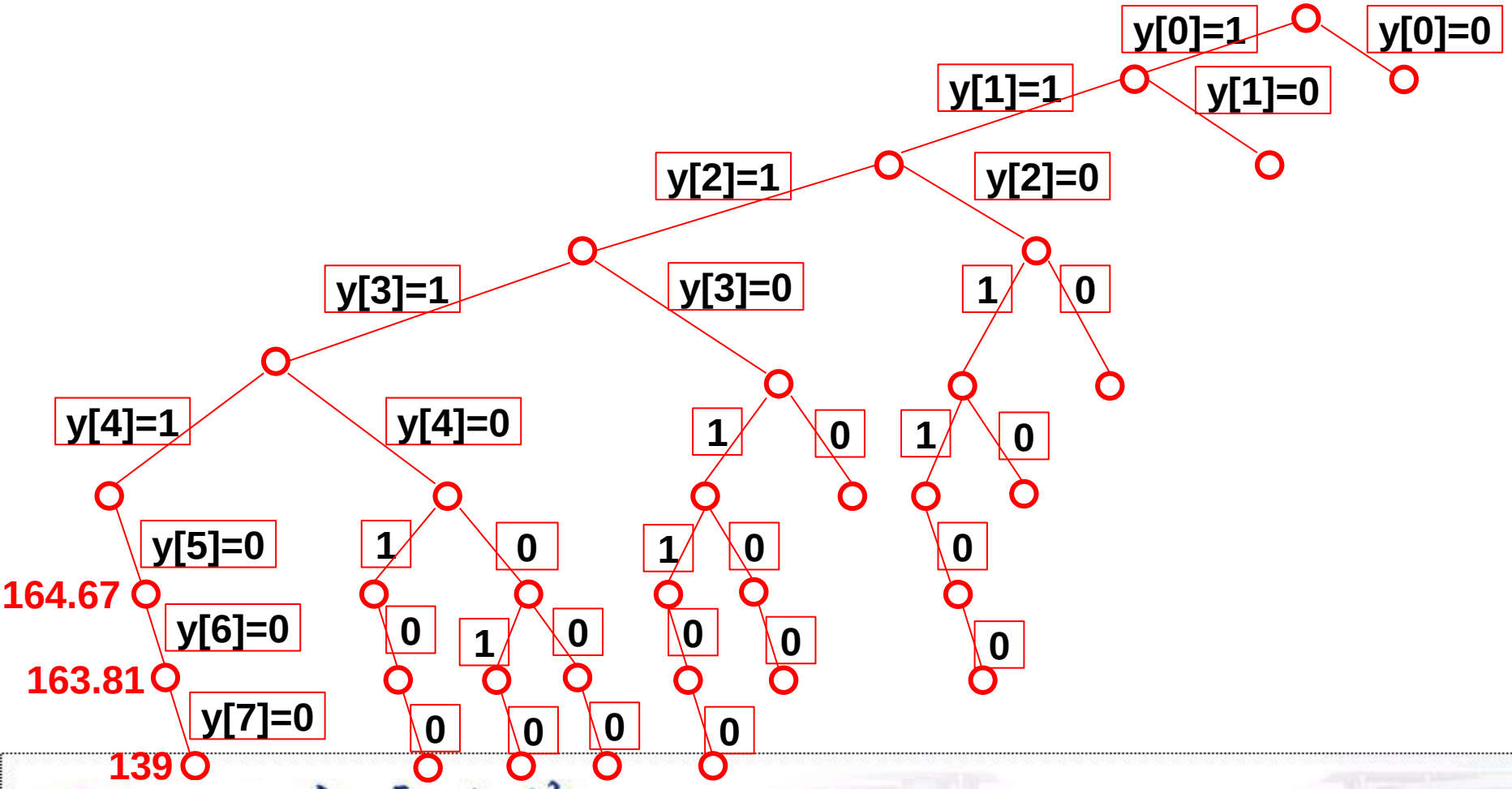
$=p_0*1+p_1*1+p_2*1+p_3*1+p_4*1+p_5*0+p_6*0+p_7/w_7*(110-89)$

$=(11+21+31+33+43)+65/55*21 \ =139+24.81 \ =163.81$

为当前结点X的上界函数值。若bp<当前最优解值下界估计值L，则可断定X子树上不含最优答案结点，可以剪去以X为根的子树。

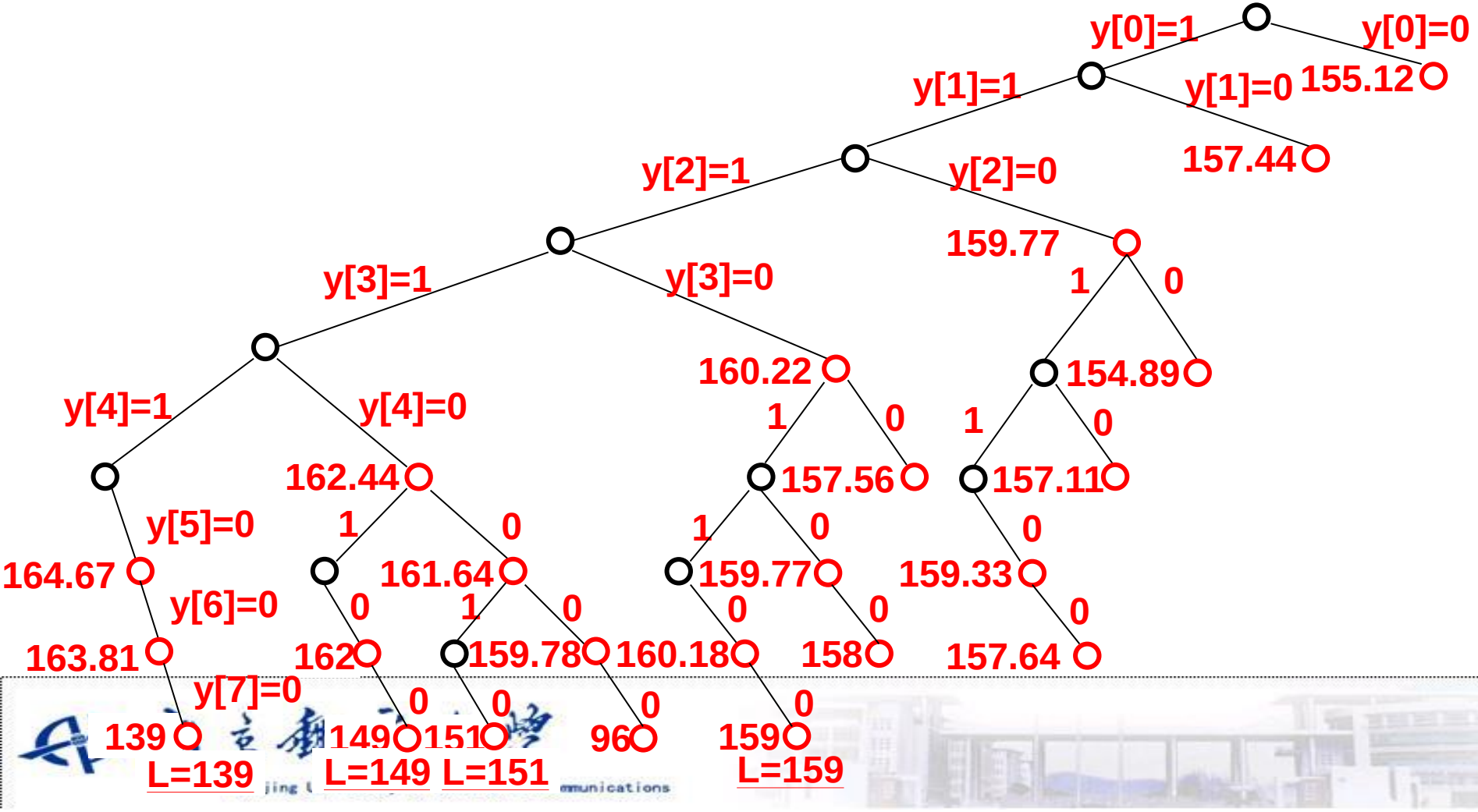


若得到最新可行解（到达状态空间树的第n+1层），则更新当前最优解下界值L。



向左走：更新 $\sum_{i=0}^{k-1} w_i x_i + w_k$ 值，bp值不变。用约束函数 $\sum_{i=0}^{k-1} w_i x_i + w_k \le M$ 剪去不含可行解的分枝；

向右走：更新bp的值，$\sum_{i=0}^{k} w_i x_i$ 值不变。用限界函数bp≤L剪去不含最优解的分枝。



8.7 批处理作业调度

8.7.1 问题描述

◆设有n个作业的集合{0, 1, ..., n-1}，每个作业有两项任务要求分别在2台设备P_1和P_2上完成。每个作业必须先在P_1上加工，然后在P_2上加工。P_1和P_2加工作业i所需的时间分别为a_i和b_i。作业i的完成时间$f_i(S)$是指在调度方案S下，该作业的所有任务得以完成的时间，则

$$MFT(S)=\frac{1}{n}\sum_{i=0}^{n-1}f_i(S)$$

是采用调度方案S的**平均完成时间**。

- **批处理作业调度**(batch job schedule)问题要求确定这n个作业的最优作业调度方案使其MFT最小。这等价于求使得所有作业的完成时间之和

$$FT(S) = \sum_{i=0}^{n-1} f_i(S)$$

最小的调度方案。

例8－5

设有三个作业和两台设备，作业任务的处理时间为（a_0,a_1,a_2）=(2,3,2)和（b_0,b_1,b_2）=(1,1,3)。这三个作业有6种可能的调度方案：(0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1),(2,1,0),它们相应的完成时间之和分别是19,18,20,21,19,19。其中，最佳调度方案S=(0,2,1)。在这一调度方案下，$f_0(S)=3$，$f_1(S)=7$，$f_2(S)=8$，FT=3+7+8=18。

8.7.2 回溯法求解

- 对于双机批处理作业调度问题，其可行解是n个作业所有可能的排列，每一种排列代表一种作业调度方案S，其目标函数是 $FT(S)=\sum_{i=0}^{n-1} f_i(S)$

使FT(S)具有最小值的调度方案或作业排列是问题的最优解。

- 对于双机作业调度，存在一个最优非抢先调度方案，使得在P_1和P_2上的作业完全以相同次序处理。 批处理作业调度问题的状态空间树解空间的大小为 n！。

- 求解这一问题没有有效的约束函数，但可以使用最优解的上界值U进行剪枝。如果对于已经调度的作业子集的某种排列，所需的完成时间和已经大于迄今为止所记录下的关于最优调度方案的完成时间和的上界值U，则该分枝可以剪去。

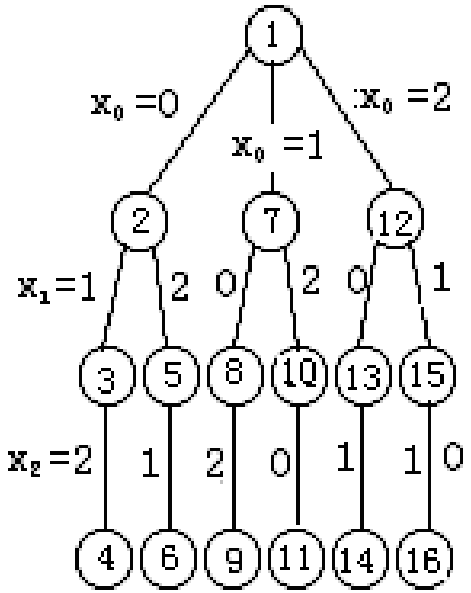


图8—18 例8—5的状态空间树

8.7.3 批处理作业调度算法

- 【程序8—9】批处理作业调度算法

```
class BatchJob{
public:
   BatchJob(int sz, int *aa,int *bb,int up)
   {
        n=sz; U=up; f=f1=0;
        a=new int[n];b=new int[n];
        f2=new int[n];y=new int[n];
        for(int i=0;i<n;i++){
              a[i]=aa[i];b[i]=bb[i];y[i]=i;
        }
   }
```

```
        int JobSch(int *x);
private:
      void JobSch(int i,int *x);
        int *a,*b,n,U,
   f,f1,*f2,*y;
};
```

```
void BatchJob::JobSch(int i,int *x)
{
    if (i==n) {
        for (int j=0; j<n;j++) x[j]=y[j];
U=f;
    }
    else
```

```
for (int j=i;j<n;j++) {
        f1+=a[y[j]];
        f2[i]=((f2[i-1]>f1)?f2[i–1]:f1)+b[y[j]];
        f+=f2[i];
        if (f<U) {
            Swap(y,i,j);
            JobSch(i+1,x);
            Swap(y,i,j);
         }
         f1-=a[y[j]];f-=f2[i];
    }
}
```

```
int  BatchJob::JobSch(int *x)
{
      JobSch(0,x);
         return U;
}
```

作业

P180　8-6

补充题：用回溯法求解4-皇后问题的所有可行解。若随机选择的路径为(0,2)和（1,3,0,2），请用蒙特卡罗方法估计实际生成的状态空间树结点数。