第六章 回溯方法

算法的基本思想

定和子集 与 0/1背包问题

n皇后和旅行商问题

图的着色问题

回溯法效率分析

回溯算法的基本思想

- 解空间

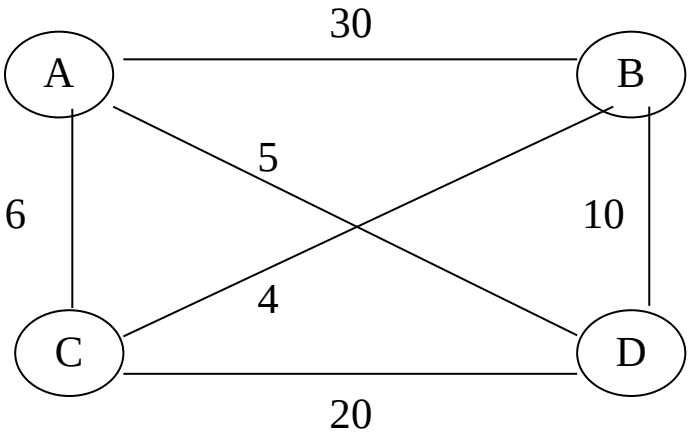
多阶段解决问题的方案，决策序列，所有可能的决策序列构成解空间。k-定解空间：前面k个决策确定后所有可能的决策序列。从求解问题的角度看，每确定一组前k个决策就相当于问题求解到一个状态。未解状态对应0-定子空间。

- 状态空间树

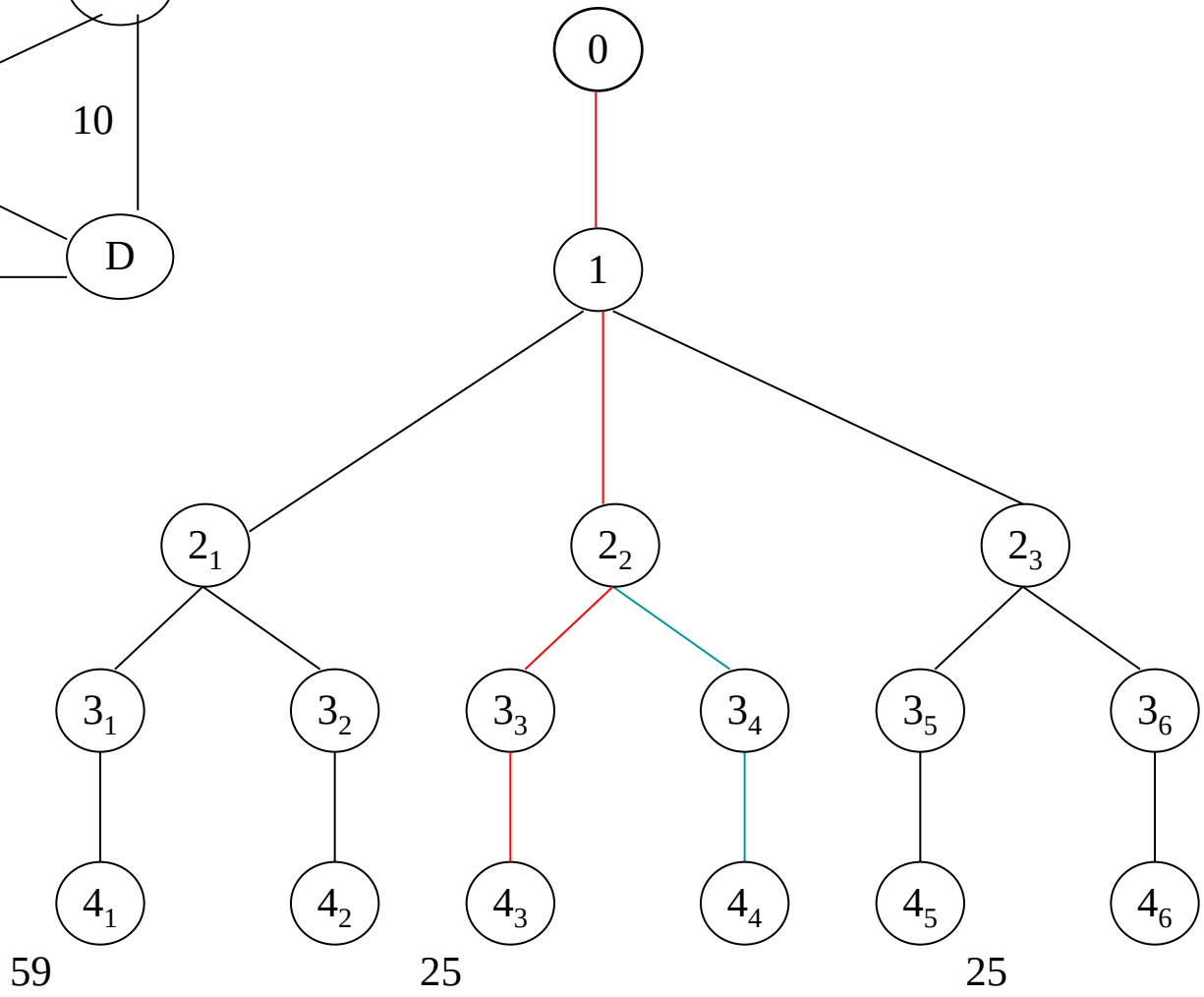
以未解状态为根节点，确定各步决策的过程可以用一棵树描述出来。根节点的子节点就是第一步决策确定后的所有问题状态，对应于各个1-定子空间。当一个完整的决策序列做出之后，实际上已经达到了一个解状态。

- 解向量: $x_1, x_2, \ldots, x_n$
- 显式约束: 只约束变量的取值范围。
- 隐式约束：强制变量之间的关联和制约。
- 目标函数：极大或极小倾向或目的，有些问题不设目标函数。
- 可行解：满足约束条件的解，在状态空间树中的答案节点。

旅行商问题状态空间树

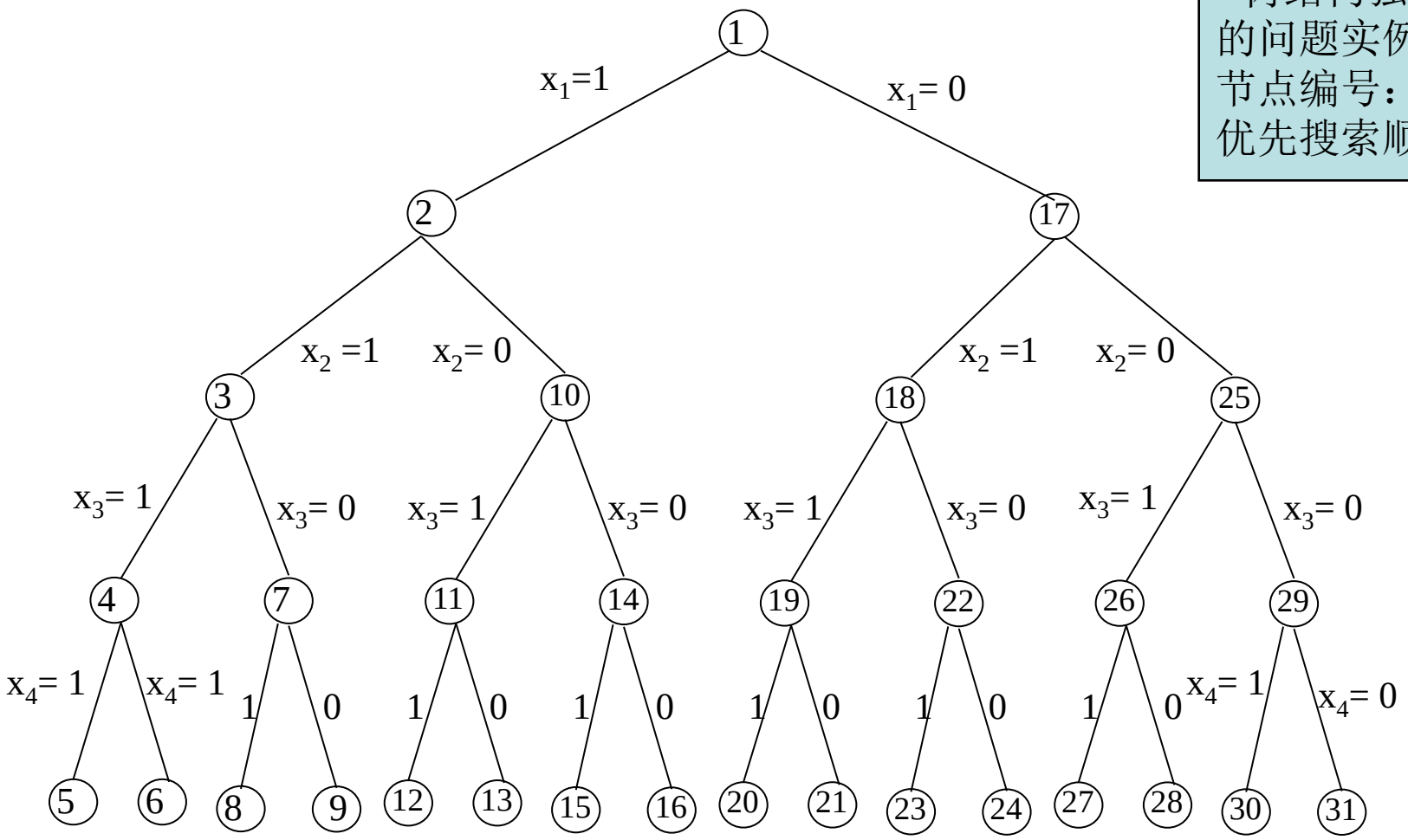


1. A；
2. AB, AC, AD;
3. ABC,ABD,ACB, ACD,ADB,ADC;
4. ABCA,ABDA, ACBA,ACDA, ADBA, ADCA

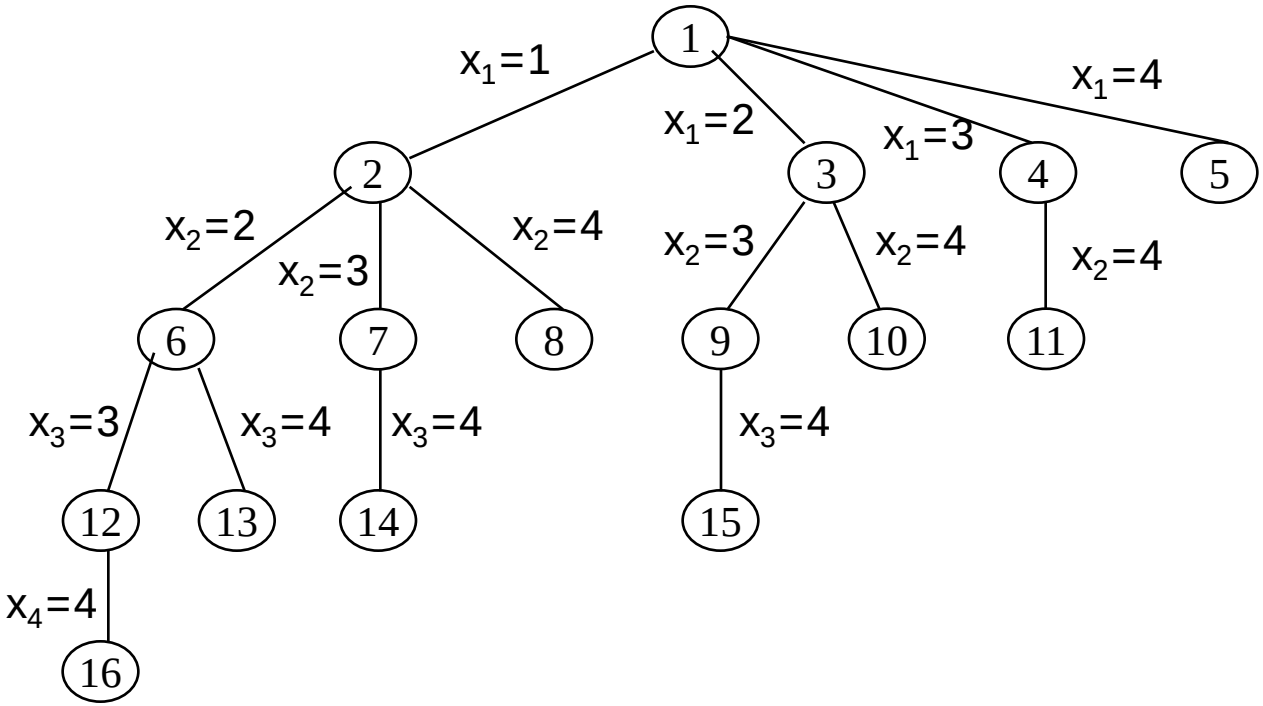


定和子集问题的状态空间树(1)

静态树：
树结构独立于解决的问题实例本身。
节点编号：以深度优先搜索顺序编号



定和子集问题状态空间树(2)

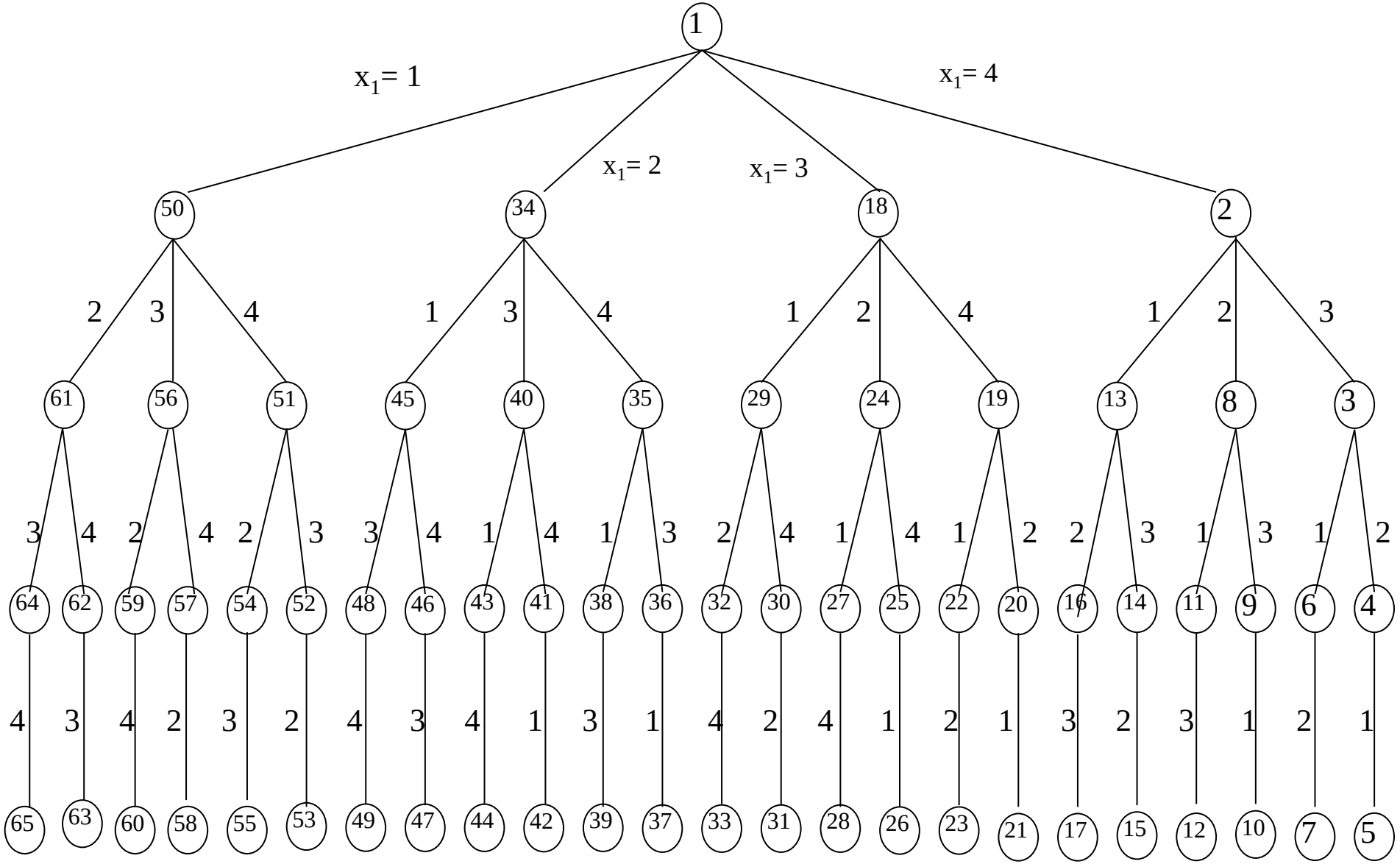


动态树：
树结构依赖于问题实例的树结构。
节点编号：以宽度优先搜索顺序编号

n=4, m=31

$(w_1,w_2,w_3,w_4)=(11,13,24,7)$,

n皇后问题状态空间树



N=4, 节点按D-搜索次序编号

回溯算法基本思想

枚举： $m=m_1m_2\ldots m_n$ 个可能的解——硬性处理。

剪枝函数：约束条件(约束函数)、最优值的界（限界函数）

回溯法: 一次针对一个部分生成一个解向量，使用限界函数测试正在形成的解向量是否有成功的可能性。如果得知通过部分向量 $(x_1,x_2,\ldots,x_k)$ 无法得到最优解，那么完全可以忽略之后的 $m_{k+1}\ldots m_n$ 个测试向量。

解的搜索：使用状态空间树，以深度优先顺序系统生成问题状态，决定其中那些状态是解状态，最终确定那些状态是答案状态。

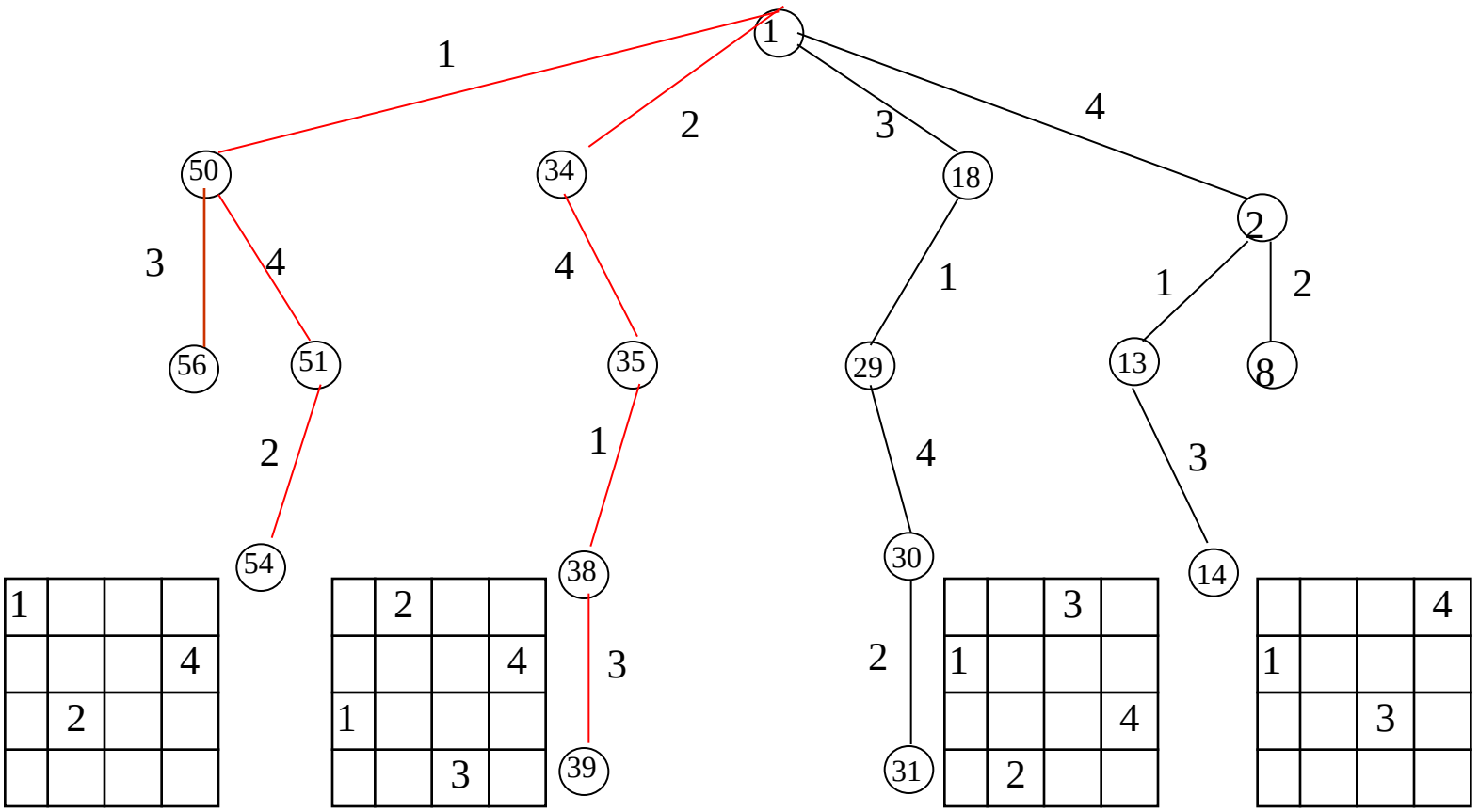
从根节点开始，逐步生成各节点的子节点。节点已经生成并且其子孙节点还有未生成的，这样的节点称为活节点。对于一个活节点，当前它的子节点正在生成，称为扩展节点。一个已经生成、不再扩展（没有子节点需要继续生成）的节点称为死节点。当前扩展节点R的新的子节点C一旦生成，这个节点就做为新的扩展节点。当子树C被扩展到最大极限时，R再次成为扩展节点，这就是回溯。生成问题状态总是在扩展节点处生成其子节点。

n皇后问题

解向量：$(x_1,x_2,\ldots,x_n)$

显式约束：$x_i \in \{1,2,\ldots,n\}$；　$x_i \neq x_j$　当$i \neq j$时。

隐式约束：　$i \neq j \Rightarrow |x_i - x_j| \neq |i-j|$。



n皇后问题的回溯算法

```
Place(k)//如果第k个皇后能放
  //在第X[k]列,则返回true,否则
  //返回false. X是一个全程数
  //组,进入此过程时已经确定了前
  //k-1个皇后的位置。
  global X[1..k]; integer i,k;
  i:=1;
  while i<k do
    if X[i]=X[k] or
      |X[i]-X[k]|=|i-k|
    then
      return (false);
    end{if}
    i:=i+1;
  end{while}
  return(true);
end{Place}
```

```
nQueens(n)
  integer k,n,X[1..n];
  X[1]:=0; k:=1; //k是当前行,X[k]是当前列
  while k>0 do
     X[k]:=X[k]+1;//转到下一列
     while X[k]<n and Place(k)=false do
       X[k]:=X[k]+1;
     end{while}
     if X[k]≤ n then
       if k=n then
         Print(X);
       else k:=k+1; X[k]:=0; //转到下一行
       end{if}
     else k:=k-1;//回溯
     end{if}
   end{while}
end{nQueens}
```

旅行商问题

- 代表n个城市的顶点0,1,2,…,n-1; 城市 i, j 之间的路程 w(i, j); 数组 $(i_1,i_2,\ldots,i_n)$ 代表一个周游 $i_1\ i_2\ \ldots\ i_n\ i_1$。
- 可行解的前 k-1个分量$x_1,\ldots,x_{k-1}$确定后，判定$x_1\ldots x_{k-1}x_k$能否构成一条路径，需要检查

$$x_k \neq x_1,\ \ldots,\ x_k \neq x_{k-1}$$ ——约束条件

- 当前路长：$cl = w(x_1, x_2)+w(x_2, x_3)+ \ldots +w(x_{k-2}, x_{k-1})$,
- 已知最佳游路长：fl。如果

$$cl + w(x_{k-1}, x_k) > fl$$ ——限界函数

则 $x_1\ldots x_{k-1}x_k$不是最短周游路径的一部分，因而不必继续向前搜索，即状态空间树中此状态后面的子树被剪掉（剪枝）。

- 更新最佳目标值：当 k = n时，如果

$$cl + w(x_{k-1}, x_k) +w(x_k, x_1) < fl$$

则令 $fl = cl + w(x_{k-1}, x_k) +w(x_k, x_1)$ 。

旅行商问题的回溯算法

```
BackTSP(n,W) //W是G的邻接矩阵, cl是当前路
  //径的长度，fl是当前所知道的最短周游长度.
  in teger  k,n,X[1..n];
  real W[1..n,1..n],cl,fl;
 for i to n do
   X[i]:=0;
 end{for}
 X[1]:=1; k:=2; cl:=0; fl:=+∞;
 while k>1 do
   X[k]:=X[k]+1 mod n;//给X[k]预分配一个值
   for j from 1 to n do
     if NextValue(k)=true then
       cl:=cl+W(X[k-1],X[k]); break;
     end{if}
     X[k]:= X[k]+1 mod n; //重新给X[k]分配值
   end{for}
   if fl ≤ cl or k=n and fl < cl+W[X[k],1] then
     cl:=cl-W(X[k-1],X[k]); k:=k-1; //回溯
   elif k=n and fl ≥ cl+W(X[k],1) then
     fl:=cl+W(X[k],1); cl:=cl-W(X[k-1],X[k]);
     k:=k-1; //回溯
   else k:=k+1;//继续纵深搜索
   end{if}
  end{while}
end{BackTSP}
```

```
NextValue(k) //进入此过程时已经
//取定了k-1个值,其中X[1]=1, X[k]
//是当前备选值。若X[k]与前面某
//个X[i](i<k)相等,则返回false,否则
//返回true. X是一个全程数组。
•   global X[1..k]; integer i,k;
•   i:=1;
•   while i<k do
•     if X[k] = X[i] then
•       return (false);
•     end{if}
•     i:=i+1;
•   end{while}
•   return(true);
end{NextValue}
```

定和子集问题

- 数的集合$S=\{w_1,w_2,\ldots,w_n\}$（按不减顺序排列）；定数：M
- 解向量：$x=(x_1,x_2,\ldots,x_n)$
- 数学模型：$x_i\in\{0,1\}$，s.t. $\sum_{1\le i\le n}w_ix_i = M$
- 假定前k-1项选择已经确定，并且第k项已选择，使得

$$\sum_{1\le i\le k}w_ix_i \le M$$

- 确定是否需要继续向前搜索：当$\sum_{1\le i\le k}w_ix_i =M$时，已达到答案节点，回溯，搜寻其它的解；当$\sum_{1\le i\le k}w_ix_i <M$时，继续向前搜索的条件是$B_{k+1}$：

$$\sum_{1\le i\le k}w_ix_i + w_{k+1}\le M \text{ 且 } \sum_{1\le i\le k}w_ix_i + \sum_{k+1\le i\le n}w_ix_i \ge M$$

- 实际问题一般满足

$$w_1 \le M, \quad \sum_{1\le i\le n}w_i \ge M$$

定和子集问题的回溯算法

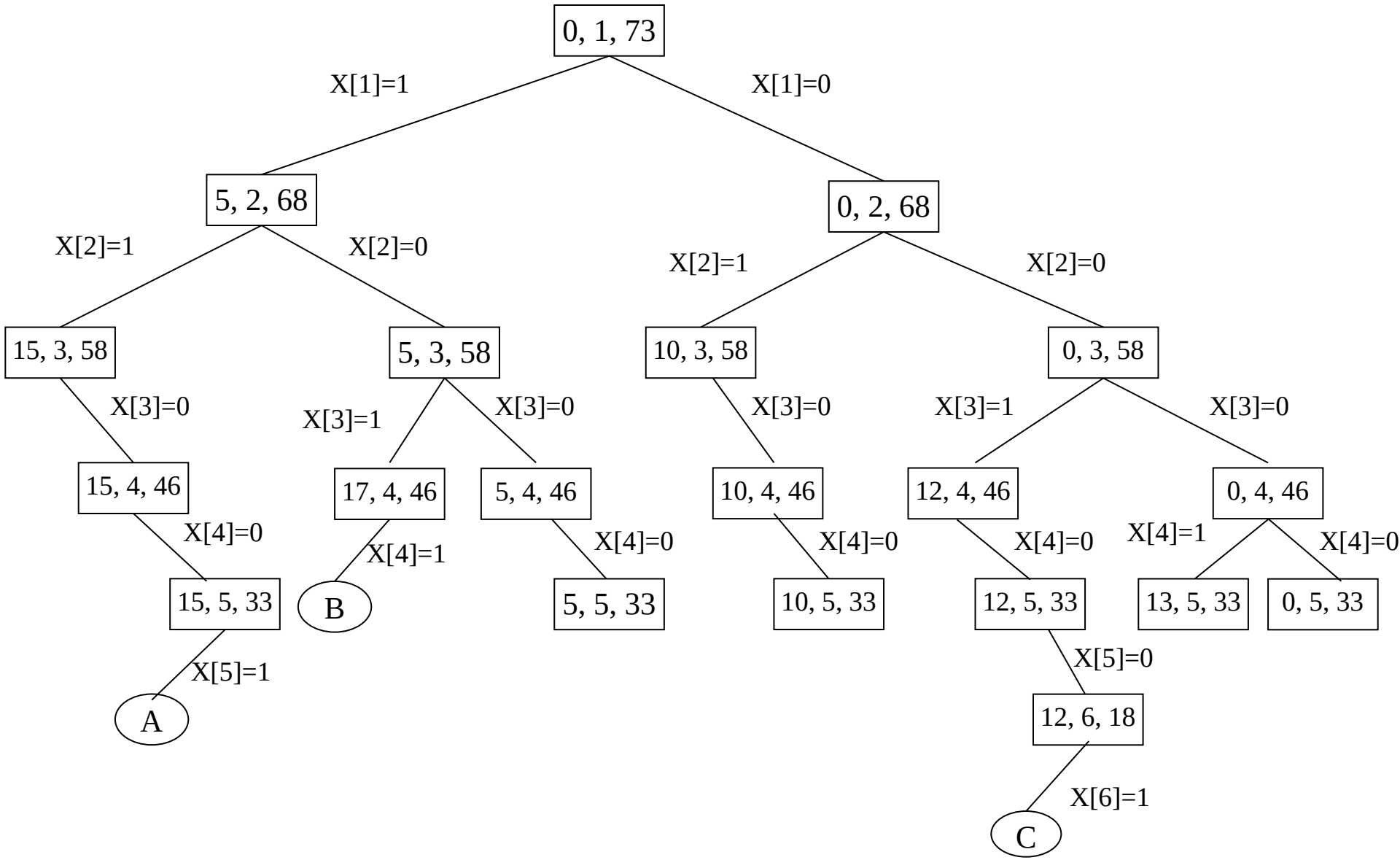
```
SumSubset(s,k,r) // 寻找W[1..n]
  //中元素和为M的所有子集。
  // W[1..n]中元素按不降次序
  //排列，进入此过程时，
  //X[1], . . . , X[k-1]的值已经确
  //定。记 s = ∑1≤i≤k-1 W[i]X[i] ,
 // r = ∑k≤i≤n X[i], 并假定
 //W[1] ≤ M， ∑1≤i≤n W[i]≥M。
 global integer M, n;
 global real W[1..n];
 global bool X[1..n];
 real r, s; integer k, j;
   //由于Bk = true，因此
   //s + W[k] ≤ M
 X[k]:=1; // 生成左儿子。
```

- **if** s + W[k]= M **then**
- print (X[j],j from 1 to k);
- **else**
- **if** s + W[k] + W[k+1] ≤ M **then**
- SumSubset(s+W[k], k+1, r-W[k]);
- **end{if}**
- **end{if}**
- **if** s + r–W[k]≥ M and s + W[k+1] ≤ M **then**
- X[k]:=0; //生成右儿子
- SumSubset(s,k+1,r-W[k]);
- **end{if}**

end{SumSubset}

定和子集问题例

n=6,M=30; W[1:6]=(5,10,12,13,15,18)



0/1背包问题

- 物品重量：$w=(w_1,w_2,\ldots,w_n)$；价值：$p=(p_1,p_2,\ldots,p_n)$。
- 数学模型： 解向量 $x=(x_1,x_2,\ldots,x_n)$

$$\max \sum_{1\le i\le n} p_i x_i$$

$$\text{s.t. } \sum_{1\le i\le n} w_i x_i \le M$$

$$x_i \in \{0,1\},\ i=1,2,\ldots,n$$

- 在前k-1件物品是否装包的选择确定之后，确定第k件物品是否装进包内。装包条件是

$$\sum_{1\le i\le k-1} w_i x_i + w_k \le M \quad \text{———约束条件}$$

- 根据背包问题贪心算法可以估计前k-1件物品是否装包的选择确定之后的状态下可能达到的目标值的一个上界B_k，只有当目前所知最佳目标值低于这个上界时，即

$$fp < B_k \quad \text{———限界函数}$$

才有必要继续向前搜索。

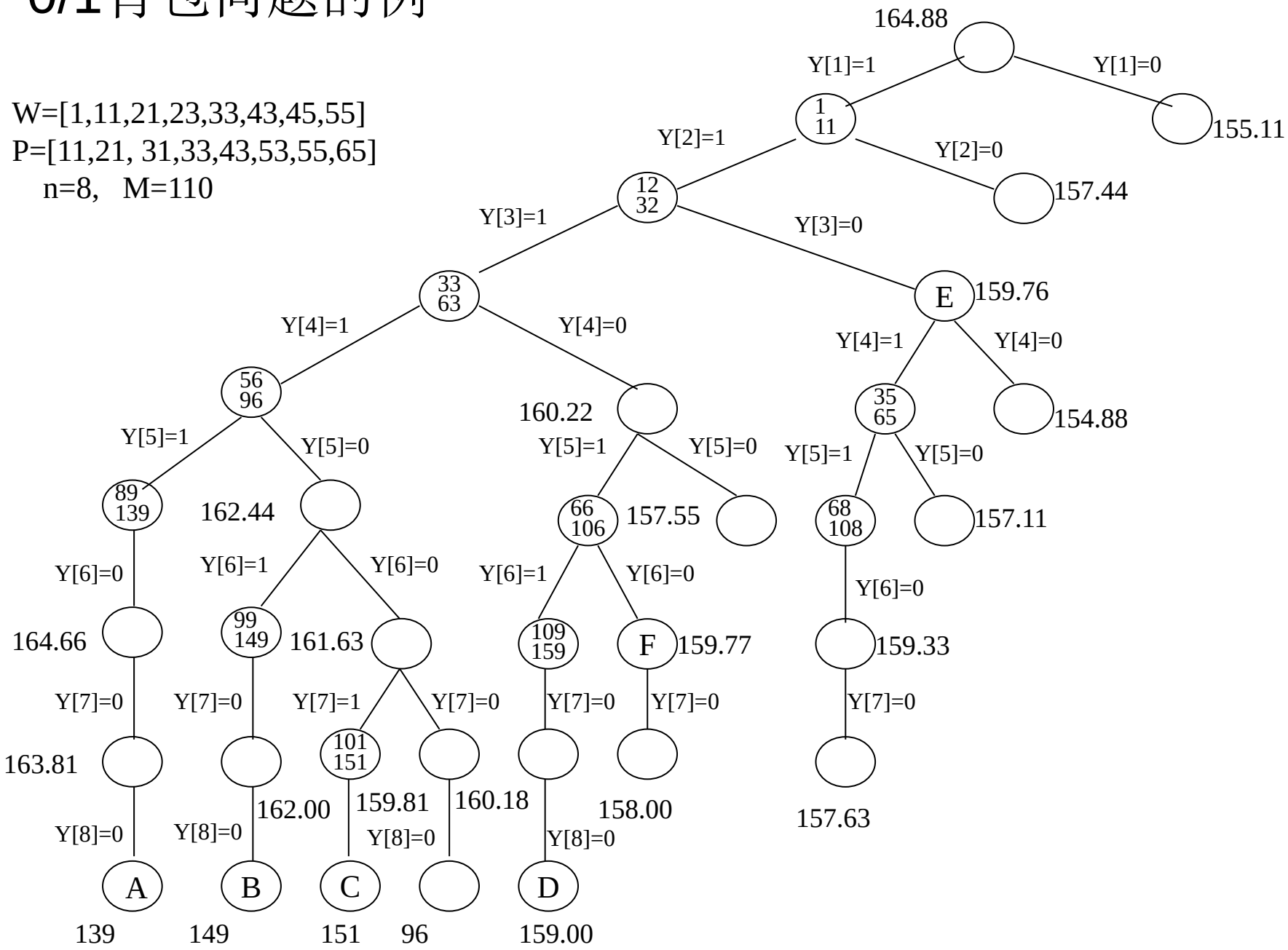
- 将物品按单位价值不增顺序排列：$p_1/w_1 \ge p_2/w_2 \ge \ldots \ge p_n/w_n$

0/1背包问题的回溯算法

```
BackKnap(M,n,W,P,fp,X)//背包容量M.n件
  //物品,重量W[1..n]，价值 P[1..n]，按单位价
  //值的不增顺序排列下标。当前所知最佳目标
  //值fp，解向量X[1..n]. cw, cp当前重量和价值
  integer n,k,Y[1..n],X[1..n];
  real M,W[1..n],P[1..n],fp,cw,cp;
  cw:=0; cp:=0; k:=1; fp:=-1;
  loop
    while k≤n and cw+W[k] ≤ M do
      cw:=cw+W[k]; cp:=cp+P[k]; Y[k]:=1; k:=k+1;
    end{while}
    if k>n then
      fp:=cp; k:=n; X:=Y;//修改解
    else Y[k]:=0;
    end{if}
    while BoundF(cp,cw,k,M)≤fp do
       while k≠0 and Y[k]≠1 do
          k:=k-1;
       end{while}
    if k:=0 then return; end{if}
     Y[k]:=0; cw:=cw-W[k];
      cp:=cp-p[k];
    end{while}
    k:=k+1;
  end{loop}
 end{BackKnap}
BoundF(cp,cw,k,M) //前k-1
  //件物品装包决策已定,
  //可能达到的最大目标值.
  global n, p[1..n],w[1..n];
  integer k,i;
  real b,c,cp,cw,M;
  b:=cp; c:=cw;
  for i from k to n do
    c:=c+w[i];
    if c < M then b:=b+p[i];
    else
      return(b+(1-(c-M)/w[i])p[i]);
    end{if}
  end{for}    return(b);
end{BoundF}
```

0/1背包问题的例

W=[1,11,21,23,33,43,45,55]
P=[11,21, 31,33,43,53,55,65]
n=8, M=110



图的着色问题

- **问题：给定无向图G和m种颜色，求出G的所有m-着色。**
- 图G有n个顶点，邻接矩阵W=(w_{ij})；m种颜色：1, 2, … , m。
- 解向量X=(x_1,x_2,…,x_n)，X[i]=k表示顶点i被着以颜色k。
- 当顶点1,2,…, j-1已经着好颜色，那么顶点j要着的颜色应该满足约束条件

$$W[i,j]=1 \Rightarrow X[i]\neq X[j],\ 1\leq i<j$$

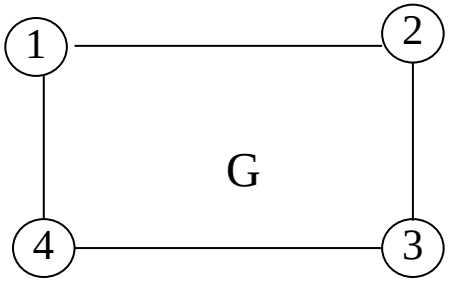
- X[i] = 0表示顶点 i 未被着色。
- 选择顶点j的颜色时采用巡回选色的方法；出现选不到合适的颜色时就回溯，修改前一个顶点已着的颜色，没有可换颜色时再向前回溯，如此等等。
- 每完成最后顶点的着色就打印该着色方案，并回溯。

图着色问题的回溯算法

```
GraphColor(k) // 采用递归。图G的邻接矩
  //阵 W[1..n, 1..n], m种颜色 1, 2,…, m。下一个
  //要着色的顶点 k。在调用GraphColor(1)之前
  //数组X的每个分量已经赋值0。
 global integer m, n, X[1..n], W[1..n, 1..n] ;
 if k=0 then exit; end{if}
 NextColor(k); // 确定X[k]的取值
 if X[k]=0 then
   k:=k-1; //说明没有颜色可以分配给第k个点
   X[k]:=0;
 else
    if k=n then   //已经找到一种着色方法
      print(X);
    else
      k:=k+1;
      GraphColor( k );  //递归
    end{if}
  end{if}
end{GraphColor}
```

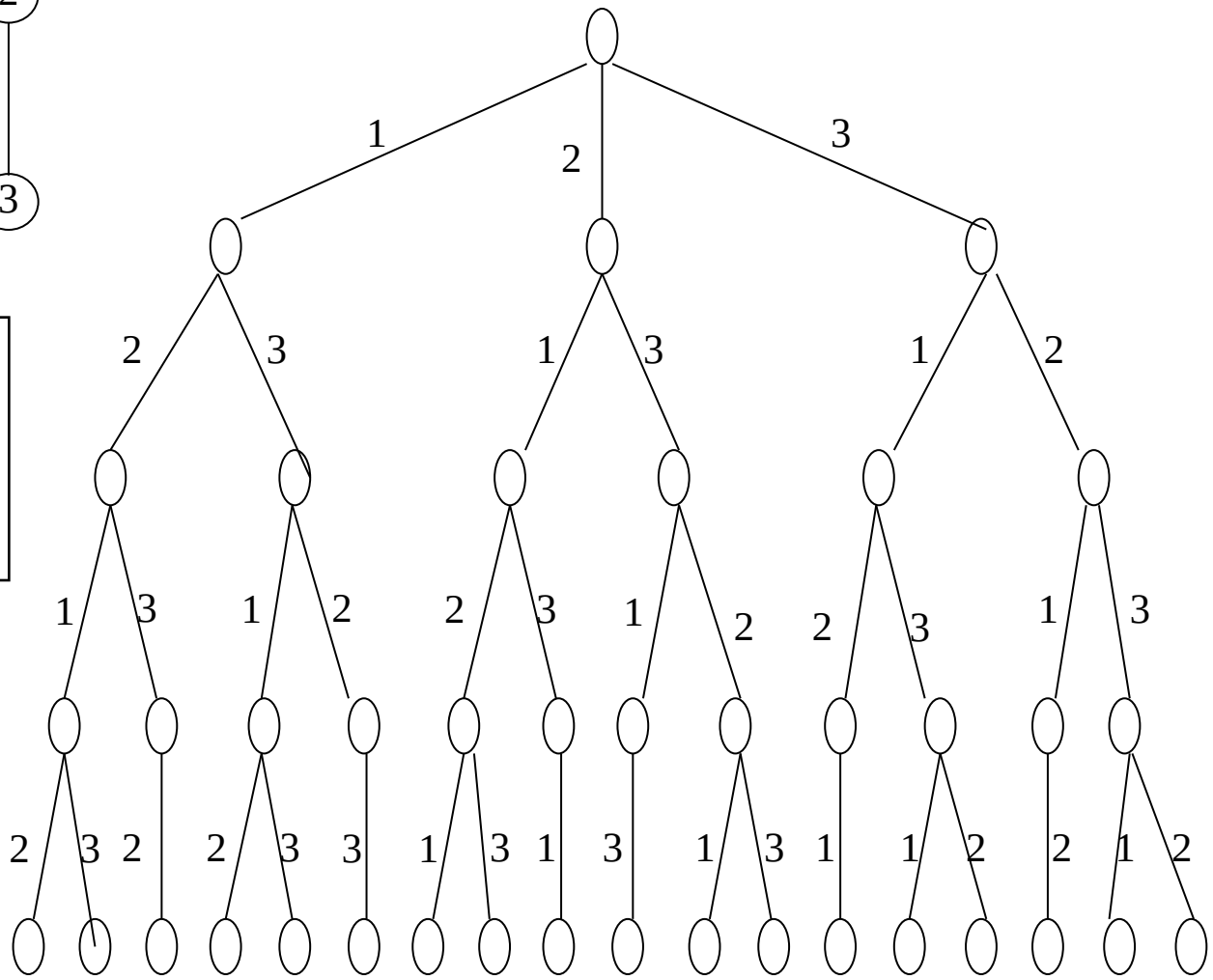
```
NextColor ( j ) //X[1],…,X[j-1]
  //已经确定且满足约束条件。
  //本过程给X[j]确定一个整数
  //k，0≤k≤m：如果还有一种
  //颜色k可以分配给顶点j,则令
  //X[j]＝k ;否则,令X[j]＝0。
•   global integer m, n, X[1..n],
•   global integer W[1..n, 1..n];
•   integer j, k;
•   loop
•     X[j]:=X[j]+1 mod (m+1);
•     if X[j]=0 then return; end{if}
•     for i to j-1 do
•       //验证约束条件
•       if W[i,j]=1 and X[i]=X[j]
•       then break; end{if}
•     end{for}
•     if i=j then return; end{if}
•       //找到一种颜色
•     end{loop}
end{NextColor}
```

图着色问题的例



n＝4，m＝3

图G的所有可能的3着色方法



回溯算法的抽象描述

- **BACKTRACK(n)**
- **//每个解都在X(1..n)中生成，一个解一经确定就立即打印。在X(1), … ,**
- **//X(k-1)已经被选定的情况下，T(X(1), … , X(k-1)) 给出X(k)的所有可能的**
- **//取值。约束函数$B_k(x_1, x_2, …, x_{k-1},x_k)$ 给出哪些元素X(k)满足约束条件。**
- **integer k, n;**
- **local X(1..n);**
- **k:=1;**
- **while k > 0 do**
- **if 还剩有没有被检验过的X(k)使得**
- **X(k)∈ T(X(1), … , X(k-1)) and**
- **B_k(X(1), … , X(k-1),X(k)) = true**
- **then**
- **if (X(1), … , X(k-1),X(k)) 是一条已到达答案节点的路径**
- **then print(X(1), … , X(k-1),X(k));**
- **end{if};**
- **k:=k+1; //考虑下一个集合**
- **else k:=k-1; //回溯先前的集合**
- **end{if};**
- **end{while}**
- **end{BACKTRACK}**

重排原理

- 影响算法效率的因素:

  1) 诸x_k的取值范围;

  2) 产生诸x_k所用的时间

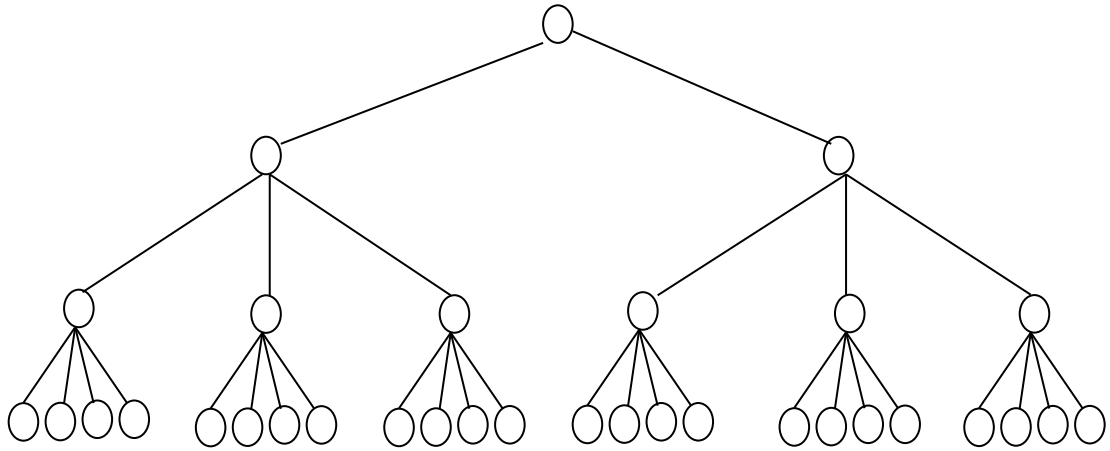
  3) 计算剪枝函数的时间;

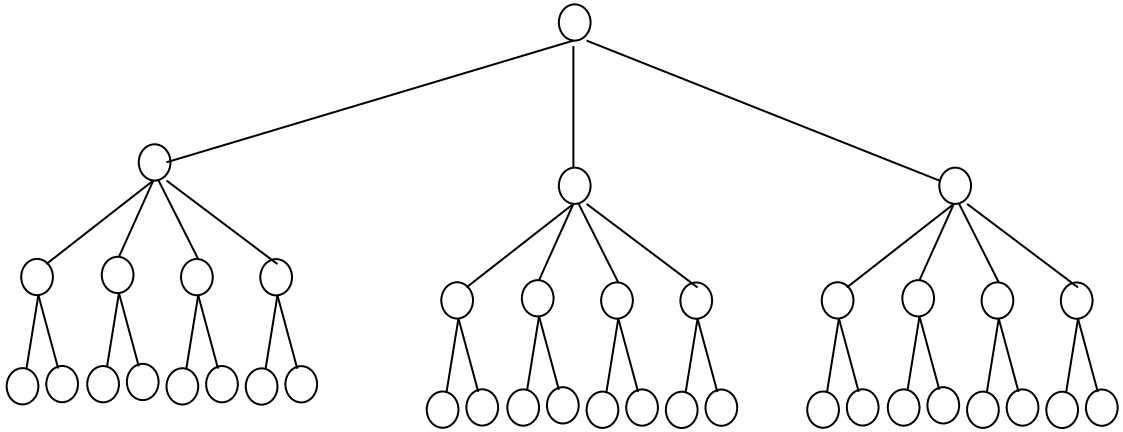
  4) 约束函数与限界函数的强度。

- 重排原理

  解向量中，可能取值少的分量尽量放在前头。

  : 状态空间树结构与解向量分量的排序有关, 在右面第一个状态空间树中搜索比第二个中搜索剪枝力度大。





第二个状态空间树只是交换了解向量分量的位置

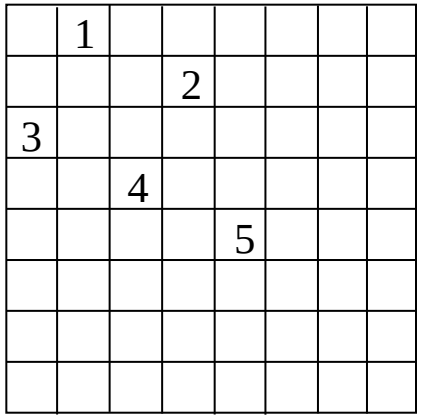
回溯算法的效率分析

- 状态空间树上结点的个数：假定x_k有m_k种可能的取值，则解空间树中结点总数为

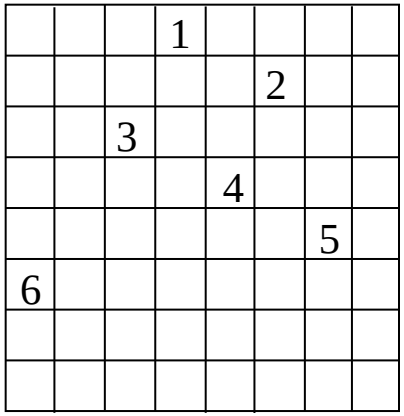
  $m=1+m_1+\ldots+m_1\ldots m_n$
- 回溯算法在搜索所有可行解过程中，由于采用剪枝函数，并没有生成所有结点。
- 当$x_1,\ldots,x_{k-1}$的值取定之后，x_k必须在$T(x_1,\ldots,x_{k-1})$中取值，且满足约束条件B_k。
- 可以采用“随机生成路径”的办法估计回溯算法对于具体实例所产生的状态空间树中结点的数目，以此来估计算法的效率。

- **Estimate** //程序沿着状态空
- //间树中一条随机路径估计
- //回溯法生成的节点总数 m
- m:=1; r:=1; k:=1;
- **loop**
- T:={X[k]： X[k]∈
- T(X[1], …， X[k-1]) and
- B_k(X[1], …， X[k])=true};
- **if** T ={} **then** exit; **end{if}**
- r:=r*size(T); m:=m+r;
- X[k]:=Choose(T); k:=k+1;
- **end{loop}**
- return(m);
- **end{Estimate}**

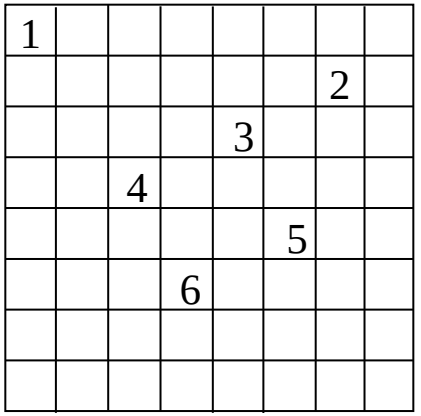
回溯法效率估计实例



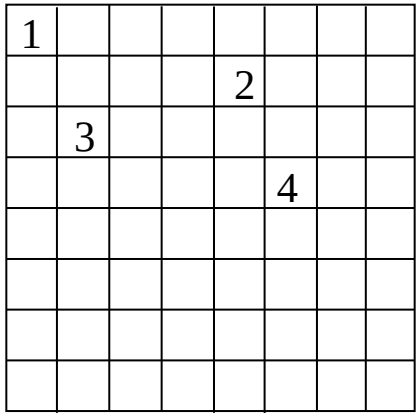
(8,5,4,3,2)=1649



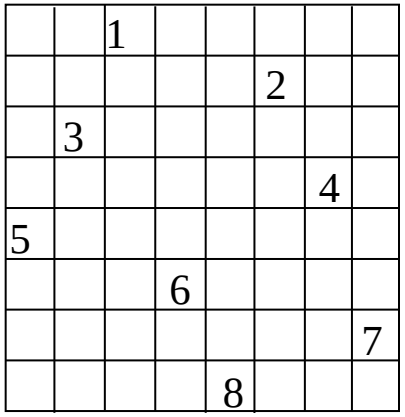
(8,5,3,2,2,1)=1369



(8,6,3,3,2,2)=3225



(8,6,3,2)=489



(8,5,3,2,2,1,1,1)=2329

8皇后状态空间树的节点数的估计值是

1812.2

而解空间树的节点总数为

$$1+\sum_{j=0}^{7}\prod_{i=0}^{j}(8-i)=109601$$

1812.2/109601≈1.65%