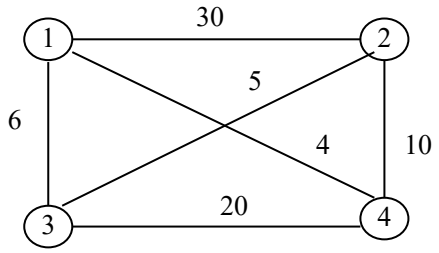
第六章　回 溯 法

§1. 回溯法的基本思想

回溯法有“通用的解题法”之称。应用回溯法求解问题时，首先应该明确问题的解空间。一个复杂问题的解决往往由多部分构成，即，一个大的解决方案可以看作是由若干个小的决策组成。很多时候它们构成一个决策序列。解决一个问题的所有可能的决策序列构成该问题的解空间。解空间中满足约束条件的决策序列称为***可行解***。对于优化问题，在约束条件下使目标值达到最大（或最小）的可行解称为该问题的***最优解***。在解空间中，前 k 项决策已经取定的所有决策序列之集称为 k 定子解空间。0 定子解空间即是该问题的解空间。

例6.1.1　旅行商问题：某售货员要到若干个城市去推销商品。已知各个城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一遍，最后回到原驻地的路线，使得总的路程（或总旅费）最小。

用一个赋权图 G(V, E)来表示，顶点代表城市，边表示城市之间的道路。图 6-1-1 中各边所带的权即是城市间的路程（或城市间的交通费）。



求赋权图 G 的具有最小权的 Hamilton 圈

图 6-1-1 一个赋权图

则旅行商问题归结为：

在赋权图 G 中找到一条路程最短的周游路线，即权值之和最小的 Hamilton 圈。

注：连通图 G 的一条 Hamilton 圈(或叫 Hamilton 回路)是指通过 G 的每个顶点恰好一次的圈。

如果假定城市 A 是驻地。则推销员从 A 地出发，第一站有 3 种选择：城市 B、C 或城市 D；第一站选定后，第二站有两种选择：如第一站选定 B，则第二站只能选 C、D 两者之一。当第一、第二两站都选定时，第三站只有一种选择：比如，当第一、第二两站先后选择了 B 和 C 时，第三站只能选择 D。最后推销员由城市 D 返回驻地 A。推销员所有可能的周游路线共有 3!=6 条，它们构成该旅行商问题实例的解空间，可用一棵有根树表示，称为状态空间树，见图 6-1-2。

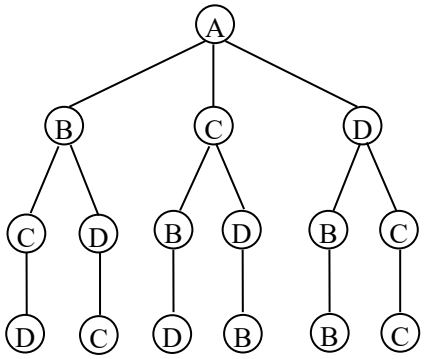


图 6-1-2 旅行商问题 6 种可能解

例 6.1.2 定和子集问题： 已知一个正实数的集合 $A=\{w_1,w_2,\cdots,w_n\}$ 和另一个正实数 M. 试求 A 的所有子集 S，使得 S 中的数之和等于 M。这个问题的解可以表示成 0/1 数组$(x_1,x_2,\cdots,x_n)$，依据 w_i 是否属于 S， x_i 分别取值 1 或 0。故解空间中共有 2^n 个元素。它的状态空间树是一棵完整二叉树。

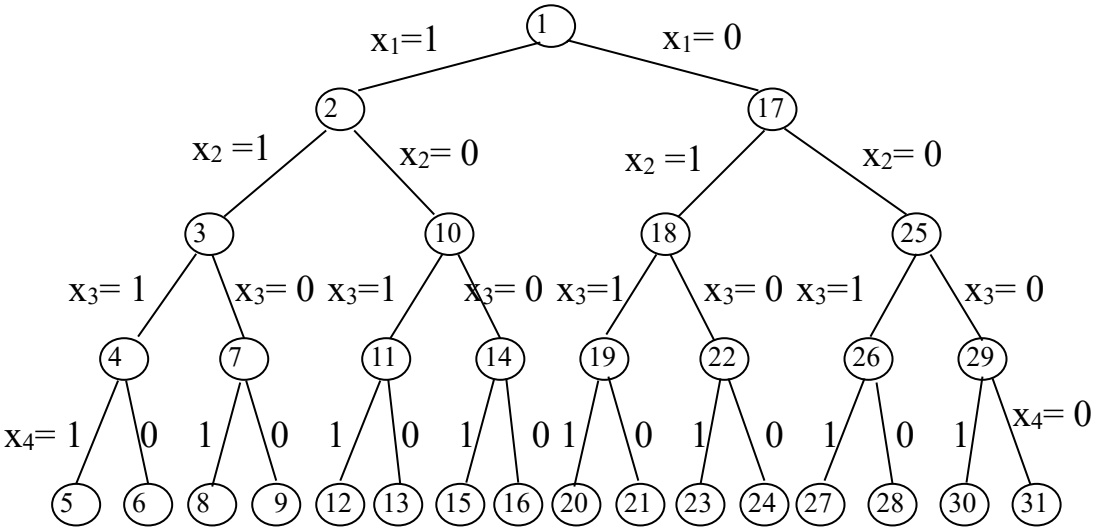


图 6-1-3 定和子集问题的状态空间树

例6.1.3 4 皇后问题: 在 4×4 棋盘上放置 4 个皇后，要使得每两个皇后之间都不能互相攻击，即任意两个皇后都不能放在同一行、同一列及同一对角线上。

将 4 个皇后分别给以 1 到 4 的编号，这个问题的解决就是要安排 4 个皇后的位置。因而，每个决策序列由 4 个决策组成：P_1，P_2，P_3，P_4，这里 P_k 代表安排第 k 个皇后的位置，每个皇后都有 16 种可能。该问题的解空间有 16^4 个决策序

列。这时的约束条件是:

任意两个皇后均不能位于同一行、同一列及同一条平行于对角线的直线上。

注意到这个解空间比较大，从中搜索可行解较为困难。现在把约束条件中的“任意两个皇后均不在同一行” 也放在问题中考虑，即：将 4 个皇后放在 4×4 棋盘的不同行上（比如，第 i 个皇后确定放在第 i 行），约束条件是：

任意两个皇后均不能位于同一列及同一条平行于对角线的直线上

这时的解空间中共有 4^4 个决策序列，因为此时每个皇后只有 4 个可能的位置可选。事实上，我们还可以用另一种方法描述，使得解空间进一步缩小。将问题陈述为：将 4 个皇后放在 4×4 棋盘的不同行、不同列上，使得

任意两个皇后均不能处在同一条平行于对角线的直线上

这时的解空间应当由 4! 个决策序列构成。因为此时每个决策序列实际上对应于 {1, 2, 3, 4}的一个排列。我们可以用一棵排列树来描述解空间。

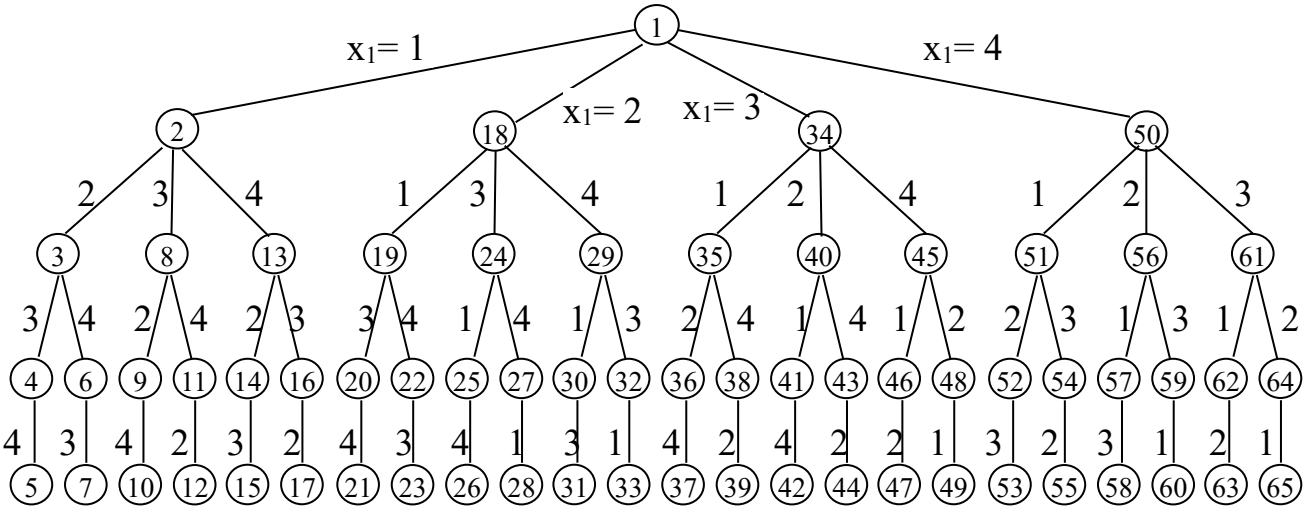


图 6-1-4 四皇后问题的状态空间树

从例 6.1.3 来看，解空间的确定与我们对问题的描述有关。如何组织解空间的结构会直接影响对问题的求解效率。这是因为回溯方法的基本思想是通过搜索解空间来找到问题所要求的解。一般地，可以用一棵有根树来描述解空间，称为状态空间树，树上的每个节点反映当前决策状态。根节点对应初始状态（还未作任何决策），称为 0-级节点；第一步决策有多少种选择，根节点就有多少个子节点，每个节点对应第一步决策后所处的一种状态，称为 1-级节点。如此类推，就得到状态空间树，它最后一级节点是叶节点。当所给的问题是从 n 个元素的集合中找出满足某种性质的子集时，相应的状态空间树称为子集合树。此时，解空间有 2^n 个元素，遍历子集树的任何算法均需 $\Omega(2^n)$ 的计算时间，如例 2。当所给

的问题是确定 n 个元素的满足某种性质的排列时，相应的状态空间树称为排列树，此时，解空间有$n!$个元素。遍历排列树的任何算法均需$\Omega(n!)$的计算时间，如例 1 和例 3。本章只讨论能用这两类状态空间树来描述解空间的问题。

状态空间树上有两类特殊的状态节点，解状态和答案状态。解状态是这样一些问题状态 S，对于这些问题状态，由根到 S 的那条路径可能延伸到一个可行解。答案状态是这样的一些解状态 S，由根到 S 的这条路径确定了这个问题的一个解（即可行解）。

确定了解空间的组织结构后，回溯法就从初始节点（状态空间树的根节点）出发，以深度优先的方式搜索整个状态空间树。这个开始节点就成为一个活节点，同时也作为当前的扩展节点。在当前扩展节点处，搜索向纵深方向移至一个新节点（即子节点）。这个新节点就成为一个新的活节点，并且作为当前的扩展节点。如果在当前的扩展节点处不能再向纵深方向搜索，则当前的扩展节点就成为死节点。此时应往回移动（回溯）至最近一个活节点处，并使这个活节点成为当前扩展节点。如此继续。回溯法就是以这种工作方式递归地在状态空间树上搜索，直至找到要求的解或状态空间树上已无活节点时为止。

事实上，当我们将问题的有关数据以一定的数据结构存储好以后（例如，旅行商问题存储赋权图的邻接矩阵、定和子集问题是存储已知的 n+1 个数、4 皇后问题用整数对(i, j)表示棋盘上各个位置等，不必先建立完整的状态空间树），就搜索并生成状态空间树的一部分或全部，并寻找所需要的解。也就是说，对于实际问题不必生成整个状态空间树，然后在状态空间上搜索，我们只需有选择地搜索。为了使搜索更加有效，常常在搜索过程中加一些判断以决定搜索是否该终止或改变路线。通常采用两种策略来避免无效的搜索，以提高回溯法的搜索效率。其一是使用***约束函数***，在扩展节点处剪去不满足约束的子树；其二是用***限界函数***（后面将阐述）“剪去”不能达到最优解的子树。这两种函数统称为剪枝函数。总结起来，运用回溯法解题通常包括以下三个步骤：

1).针对所给问题，确定问题的解空间；

2).确定易于搜索的解空间结构—状态空间树；

3).以深度优先的方式搜索状态空间树，并且在搜索过程中用剪枝函数避免无效的搜索。

§2.定和子集问题和 0/1 背包问题

定和子集问题可以写成如下数学规划模型：确定所有向量 $x=(x_1,x_2,\cdots,x_n)$ 满足：

$$\begin{aligned} & x_i \in \{0,1\}, i = 1,2,\cdots,n \\ & s.t. \quad \sum_{1\le i\le n} w_i x_i = M \end{aligned} \tag{6.2.1}$$

如果让w_i，p_i分别表示第 i 件物品的重量和价值，M 代表背包的容量，则 0/1 背包问题可以建立如下的数学模型：确定一个向量$x=(x_1,x_2,\cdots,x_n)$，使得

$$\begin{aligned} & \max \sum_{1\le i\le n} p_i x_i \\ & s.t. \quad \sum_{1\le i\le n} w_i x_i \le M \\ & x_i \in \{0,1\}, i = 1,2,\cdots,n \end{aligned} \tag{6.2.2}$$

这是一个优化问题，与定和子集问题比较，它多出目标函数，但只要求确定一个最优解；定和子集问题则往往要求确定所有可行解。

1．定和子集问题

用回溯法求解定和子集问题的过程也即是生成状态空间树的一棵子树的过程，因为,在搜索期间将剪掉不能产生可行解的子树（即不再对这样的子树进行搜索）。按照回溯算法的基本思想，搜索是采用深度优先的路线，算法只记录当前路径。假设当前扩展节点是当前路径的深度为k的节点，也就是说当前路径上，$x_i,\ 1\le i\le k-1$ 已经确定，算法要决定第k步决策，即w_k是否选进解集合中。此时，有三种情况：

$$w_k+\sum_{1\le i\le k-1} w_i x_i = M\ ,\quad w_k+\sum_{1\le i\le k-1} w_i x_i < M\ ,\quad w_k+\sum_{1\le i\le k-1} w_i x_i > M$$

前一种情况说明已经建立一个可行解，当前扩展节点就是一个答案节点。此时，应该记录已经获得的解（后面的变量取 0），停止对该路径的继续搜索，返回到前面最近活节点。第二种情况出现，说明当前扩展节点的左儿子节点是活节点，按照回溯法原则作为下一个扩展节点；第三种情况出现，说明当前扩展节点的左儿子节点不是活节点，需要考虑右儿子节点是否是活节点。如果集合$A=\{w_1,w_2,\cdots,w_n\}$中的元素是按不降的次序排列的，则只有$\sum_{1\le i\le k-1} w_i x_i + w_{k+1} \le M$时，右儿子节点才是活节点，按照回溯法原则作为下一个扩展节点。否则，当前扩展节点的右儿子节点也不是活节点，算法至此只能回溯了。

这样，深度为的节点可作为当前扩展节点的条件为

$$\sum_{1\le i\le k} w_i x_i + \sum_{k+1\le i\le n} w_i \ge M \quad ,\quad \sum_{1\le i\le k} w_i x_i + w_{k+1} \le M \tag{6.2.3}$$

(6.2.3)中两个不等式要求的条件记做 B_k，即当这两个不等式满足时，B_k=true；否则，B_k=false.

程序 6-2-1 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]
   // 的值已经选定，而且 Bk=true。记 s = ∑W[j]X[j] (1≤j≤k-1)，r = ∑W[j] (k≤j≤n)，
   //并假定 W[1] ≤ M, ∑W[j] ≥ M (1≤j≤n)
1   global integer M, n;
2   global real W[1..n];
3   global bool X[1..n];
4   real r, s;
5   integer k, j;
    //由于 Bk =true，因此 s + W[k]≤ M
6   X[k]:=1; // 生成左儿子。
7   if s + W[k]= M then
8     print (X[j],j from 1 to k);
9   else
10    if s + W[k] + W[k+1] ≤ M then
11      SumSubset(s+W[k], k+1, r-W[k]);
12    end{if}
13  end{if}
14  if s + r - W[k]≥ M and s + W[k+1] ≤ M then
15    X[k]:=0; //生成右儿子
16    SumSubset(s,k+1,r-W[k]);
17  end{if}
18  end{SumSubset}
```

因为假定 W[1]≤M，$\sum_{1\le j\le n} W[j]\ge M$，所以程序开始执行时，$B_1$=true。同样，程序在递归执行过程中，总是以前提条件 B_k=true 开始处理深度为 k-1 的节点的儿子们的生成过程。左儿子生成由语句 6 完成，右儿子生成由语句 14～15 完成。在此过程中决定是否转到生成深度为 k 的节点的儿子节点的生成过程，由语句 10～12 和 14～16 完成。所以该算法所生成的子树的叶节点或是某个左儿子，此时找到一个可行解；或者是某个右儿子，此时由根到该节点的路径不能延伸为可行解。

例 6.2.1 n=6,M=30; W[1..6]=(5,10,12,13,15,18)。图 6-1-5 给出了由算

法 SumSubset 所生成的状态空间树的子树。

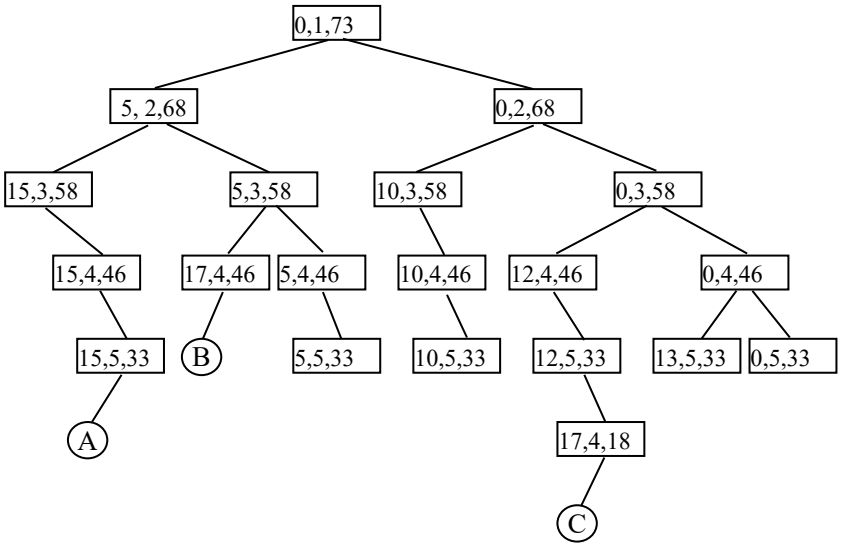


图 6-1-5 算法 SumSubset 所生成的解空间树的子树

2．0/1 背包问题

0/1 背包问题是一个优化问题,因此不仅可以使用约束函数，而且可以使用限界函数做剪枝处理。如果当前背包中的物品总重量是 cw,前面 k-1 件物品都已经决定好是否放入包中,那么第 k 件物品是否放入包中取决于不等式 $cw + w_k \le M$ 是否满足.这即是搜索的约束函数.

我们可以如下确定限界函数。回忆背包问题的贪心算法，当物品按照 $p_i / w_i \ge p_{i+1} / w_{i+1}$ 的规则排序时，最优解具有形式：

$$(x_1,\cdots,x_{l-1},x_l,x_{l+1},\cdots,x_n)=(1,\cdots,1,t,0,\cdots,0)$$

其中, $0 \le t < 1$.假设当前背包中物品的总效益值和总重量分别为 cp、cw,如下方式确定限界函数:

程序 6-2-2 构造限界函数

```
proc BoundF(cp,cw,k,M) //在前 k-1 件物品装包决策已经作出的前提下,
  //考虑可能达到的最大效益值，返回一个尽量小的上界。
  // cp、cw 分别代表背包中当前物品的价值和重量
 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+(M-c)/w[i])*p[i]);
    end{if}
  end{for}
  return(b);
end{BoundF}
```

这样确定的上界函数表明,从当前确定的 k-1 定子空间中寻求到的可行解所能达到的效益值以该函数的当前值为上界。所以,如果搜索最优解的算法在此之前已经知道大于这个界值的效益,则没有必要在当前确定的 k 定子空间中搜索.据此我们给出 0/1 背包问题的回溯算法.

程序 6-2-3 0/1 背包问题的回溯算法

```
proc BackKnap(M,n,W,P,fp,X)//M 是背包容量。n 件物品，数组 W[1..n]
  //和 P[1..n]分别表示物品的重量和价值，并按单位价值的不增顺序排
  //列下标。fp 是当前所知背包中物品所能达到的最大价值。X[1..n]中
  //每个元素值取 0 或 1。若物品 k 未放入背包,则 X[k]=0;否则,X[k]=1。
1   integer n,k,Y[1..n],X[1..n];
2   real M,W[1..n],P[1..n],fp,cw,cp;
    //cw、cp 分别代表当前背包中物品的重量和价值;
3   cw:=0; cp:=0; k:=1; fp:=-1;
4   loop
5     while k≤n and cw+W[k]≤M do //使用约束函数
6       cw:=cw+W[k]; cp:=cp+P[k]; Y[k]:=1; k:=k+1;
7     end{while}
8     if k>n then
9       fp:=cp; k:=n; X:=Y;  //记录当前最好的解
10    else Y[k]:=0;  //准备修改解
11    end{if}
12    while BoundF(cp,cw,k,M)≤fp do
```

```
13      while k≠0 and Y[k]≠1 do
14        k:=k-1;
15      end{while}
16      if k:=0 then return; end{if}
17      Y[k]:=0; cw:=cw-W[k]; cp:=cp-P[k];
18    end{while}
19    k:=k+1;
20  end{loop}
21 end{BackKnap}
```

算法采用一个大的循环搜索各条可能的路径。当一条路径的搜索到某一步不能继续往下搜索时,此时或是因为约束条件不满足,或是因为限界函数不满足。它们分别在语句 5~7 的子循环和语句 12~18 的子循环中体现。这是程序的两个主要部分，其余部分主要处理边界条件，如 k > n 的验证等。第一种情况出现时, 语句 8~11 给出部分处理，剩下的事情交给语句 12~19 来处理，给出了搜索退回的方案。

例 6.2.2　0/1 背包问题：n=8，M=110，W=[1，11，21，23，33，43，45，55]，P=[11，21，31，33，43，53，55，65]。解空间树的搜索情况参看本章附页：“0/1 背包问题解空间树搜索情况”。

§3. n-皇后问题和旅行商问题

在第一节已经给出了这两个问题的状态空间树. 如果用 $(x_1, x_2, \cdots, x_n)$ 表示解，则它是自然数 $\{1,2,\cdots,n\}$ 的一个排列。对于旅行商问题，我们可以假定 $x_1 = 1$，表示售货员的驻地是城市 1。对于旅行商问题，采用回溯法主要是给出约束函数和限界函数；而对于 n 皇后问题，主要任务是给出约束函数。对于旅行商问题可以假定赋权图是完全图，这只要将实际不相邻的两个顶点间用带有权 $+\infty$ 的边连接即可。

1. n-皇后问题

这里的约束条件是: ***任何两个皇后都不能位于同一对角线上***。如果用 (i, j) 表示棋盘上第 i 行与第 j 列交叉处的位置，则在同一条平行于主对角线的直线上的两点 (i, j) 和 (k, l) 满足关系式 $j - l = -(i - k)$；而处于同一条平行于副对角线的

直线上的两点(i,j)和(k,l)则满足关系式$j-l=i-k$.将这两个关系式略作变形即得两种情况的统一表示：

$$|j-l|=|i-k| \tag{6.3.1}$$

反之，如果棋盘上的两点(i,j)和(k,l)满足关系式(6.3.1)，则它们一定位于同一条平行于对角线的直线上。所以，如果$(x_1,x_2,\cdots,x_n)$是 n 皇后问题的一个可行解，则对任何$i\neq k$，等式

$$|x_i-x_k|=|i-k| \tag{6.3.2}$$

均不得成立，这即是约束条件。如果假定前面的 k-1 个皇后位置已经排好，现在要安排第 k 个皇后的位置，必须使得$x_i\neq x_k$，而且等式(6.3.2)对于$i=1,2,\cdots,k-1$都不得成立。

算法可以这样设计：对于x_k所有可能的取值，验证上述条件。对于每个取值x_k，这要用一个 bool 函数 Place 来完成。

程序 6-3-1 皇后问题放置函数

```
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，进入此过程时已经设置了 k-1 个值，
  //检验第 k 个皇后的位置 X[k]是否与前 k-1 个皇后的位置有冲突。
    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}
```

使用函数 Place 能使求 n-皇后问题的回溯算法具有简单的形式.

程序 6-3-2 求 n-皇后问题可行解的回溯算法

```
proc 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}
```

程序 nQueens 处理 4-皇后问题的执行过程参看本章附页“4 皇后解空间树搜索情况”.

2.旅行商问题

用 0,1,…,n-1 代表图的 n 个顶点,一个周游$i_1i_2\cdots i_ni_1$用数组$(i_1,i_2,\cdots,i_n)$表示，它是旅行商问题的一个可行解。如果可行解的前 k-1 个分量$x_1,x_2,\cdots,x_{k-1}$已经确定，则判定$x_1x_2\cdots x_{k-1}x_k$能否形成一条路径，只需做 k-1 次比较:

$$x_k \neq x_1, x_k \neq x_2, \cdots, x_k \neq x_{k-1}, \tag{6.3.3}$$

此即构成旅行商问题的约束条件。用 w(i,j)记边(i,j)的权值。cl 记当前路径$x_1x_2\cdots x_{k-1}$的长度,即$cl=\sum_{1\le i\le k-2} w(x_i,x_{i+1})$。如果当前知道的最短周游的线路长度为$fl$，则当

$$cl + w(x_{k-1}, x_k) > fl \tag{6.3.4}$$

时，$x_1x_2\cdots x_{k-1}x_k$不会是最短周游路线的一部分，在状态空间树中，相应的一枝

被剪掉，(6.3.4)即是旅行商问题的限界条件。此外，当$k=n$时，若

$$cl+w(x_{k-1},x_k)+w(x_k,x_1)<fl \tag{6.3.5}$$

则算法需要更新 fl，令

$$fl=cl+w(k-1,k)+w(k,1) \tag{6.3.6}$$

程序 6-3-3 旅行商问题的约束条件函数

```
NextValue(k) //从节点 1 出发的路径,如果 X[k]是前面某个节点 X[i](i<k)
   //则返回 false，否则返回 true。X 是一个全程数组，进入此过程时前
   //k-1 个值已经确定，而且第 k 个值 X[k]已经选出备检。
   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}
```

使用函数 NextValue 能使求旅行商问题的回溯算法具有简单的形式

程序 6-3-4 求旅行商问题的回溯算法

```
proc BackTSP(n,W) // 求图 G 的从顶点 0 出发的最短周游（Hamilton 圈）
   //路径，W 是 G 的邻接矩阵。X[k]是当前路径的第 k 个节点，cl 是当前
   //路径的长度，fl 是当前所知道的最短周游长度.
    integer  k,n,X[1..n];
    real W[1..n,1..n],cl,fl;
    for i to n do
      X[i]:=0;
    end{for}
    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]分配一个值。
        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}
```

旅行商问题的例子

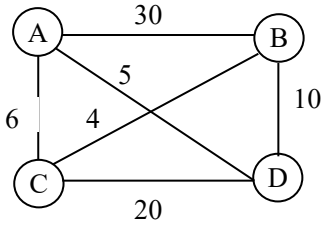
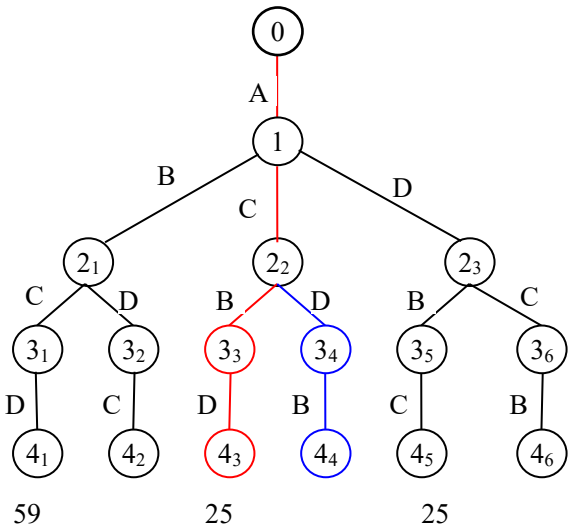


图　G



旅行商解空间树上的搜索情况

§4 图的着色问题

已知一个无向图 G 和 m 种颜色，在只准使用这 m 种颜色对图 G 的顶点进行着色的情况下，是否有一种着色方法，使图中任何两个相邻的顶点都具有不同的颜色？如果存在这样的着色方法，则说图 G 是 m-可着色的，这样的着色称为图 G

的一种 m-着色。使得 G 是 m-可着色的最小数 m 称为图 G 的色数。这一节所讨论的图的着色问题是：

给定无向图 G 和 m 种颜色，求出 G 的所有 m-着色

用 G 的邻接矩阵 W 表示图 G，W[i,j]=1 表示顶点 i 与 j 相邻（有边相连），否则 W[i,j]=0，(i,j = 1,2,…, n)。 m 种颜色分别用 1，2，…,m 表示。图 G 的每一种 m-着色都可以用一个 n-维数组 X[1..n]表示。X[i]=k 表示顶点 i 被着上颜色 k。因为每一个顶点至多有 m 种着色可选，简单分析可知，该问题的解空间是一个完全的 m-叉树，树的高度是 n。X 是可行解，当且仅当

$$W[i,j]=1 \Rightarrow X[i]\neq X[j] \qquad (6.4.1)$$

此即是约束条件。假如前 j-1 个顶点已经着好颜色，即 X[1],…,X[j-1]已经确定，则确定第 j 个顶点是否有可着的颜色 X[j]，根据约束条件，应该验证关系

$$W[i,j]=1 \Rightarrow X[i]\neq X[j],\ 1\leq i<j \qquad (6.4.2)$$

是否成立。这可由下面程序完成。

程序 6.4.1 下一步选色算法

```
NextColor ( j ) //进入此过程前，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（满足约束条件），则令 X[j]＝k；否则，令 X[j]＝0。
  global integer m, n, X[1..n], 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}
```

在程序 NextColor 开始执行之前，诸 X[k]均已经有值。这可在着色问题的回溯算法初次调用该函数时赋予初值：X[i]=0,i=1,2,…,n。

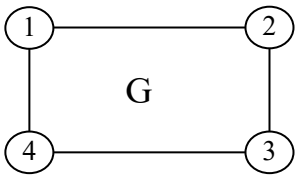
程序 6-4-2 图的着色问题回溯算法

```
proc GraphColor（k）// 采用递归。W[1..n, 1..n]是图 G 的邻接矩阵，
```

```
// 1, 2,… m 代表 m 种颜色。k 是下一个要着色的顶点。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
   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}
```

例 6.4.1 n＝4，m＝3，
无向图 G 如右图



正好用 3 种颜色
的解只有 12 个

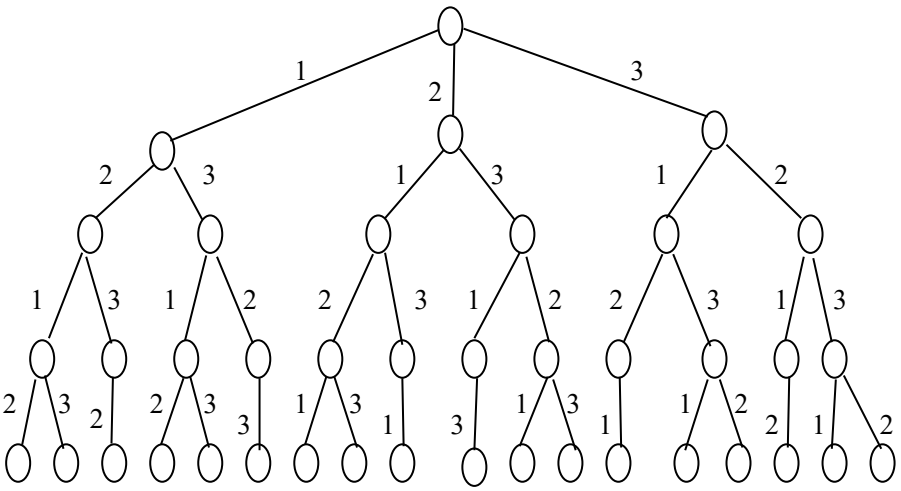


图 6-4-1 一个 4 顶点的图和所有可能的 3 着色

§5. 回溯法的效率分析

一. 回溯法的一般型式

每个节点确定所求解问题的一个问题状态，由根节点到其它节点的所有路径确定了这个问题状态空间。解状态 S 是指由根到节点 S 那条路径能够确定解空间中的一个决策序列。在图 6-5-2 的状态空间树中，所有的节点都是解状态，而在图 6-5-1 的状态空间树中，只有叶节点才是解状态。答案状态是指这样的解状态 S，由根到节点 S 的路径确定了这个问题的一个可行解。以下是定和子集问题状态空间树的两个不同表示形式：图 6-5-1 是静态的，而图 6-5-2 是动态的，因为它每个解状态的元组大小是变化的。

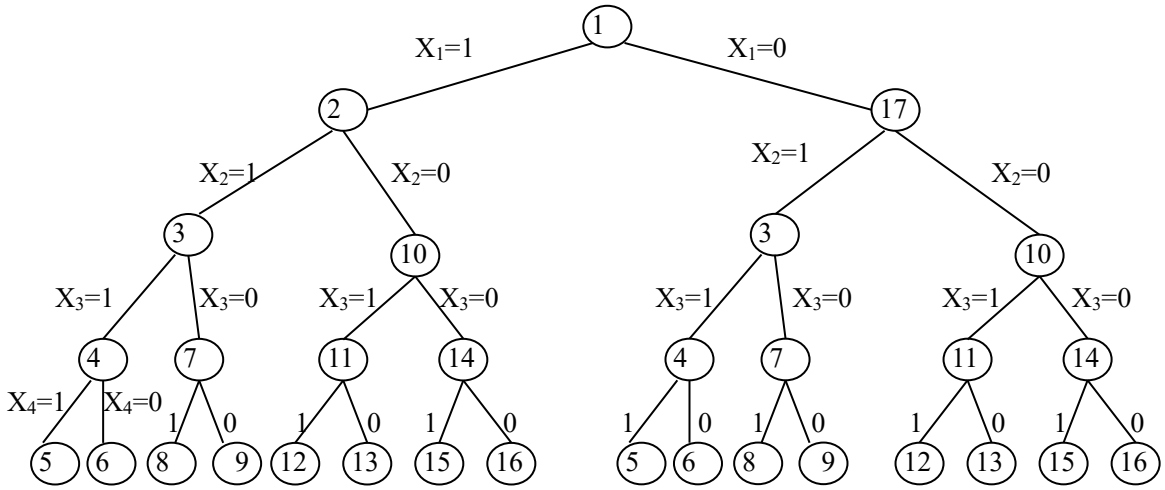
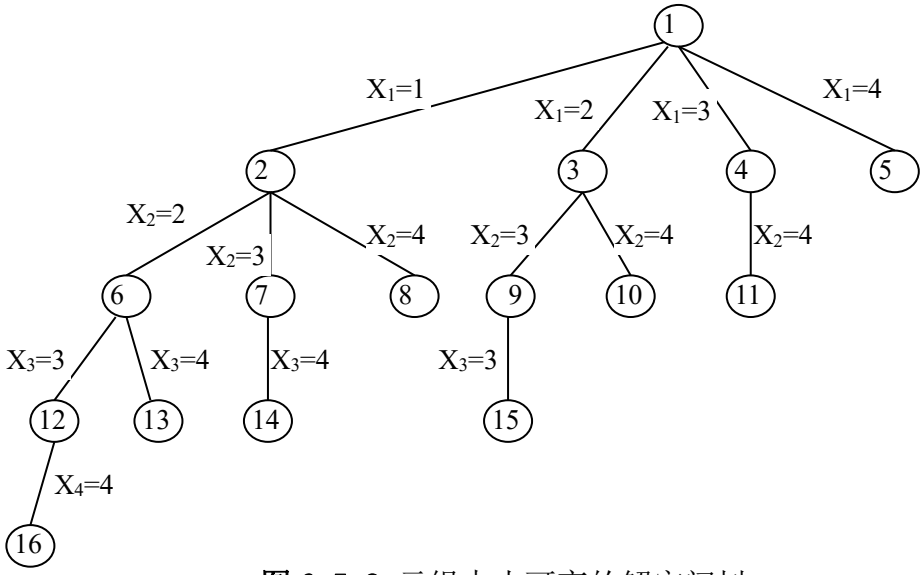


图 6-5-1 元组大小固定的解空间树表示



定和子集问题：
n=4, M=31
(w_1,w_2,w_3,w_4)
=(11,13,24,7)
满足要求的解是：
（11，13，7）
和（24，7）
即 {1，2，4}和
{3，4}或表示为：
（x_1, x_2, x_3, x_4）
= (1,1,0,1)
和(0,0,1,1)

图 6-5-2 元组大小可变的解空间树

对于任何一个问题，一旦设想出一种状态空间树，那么就可以先系统地生成问题状态，接着确定这些问题状态中的哪些状态是解状态，最后确定那些解状态是答案状态，从而将问题解出。回溯法的形式描述如下：

假定要找出所有问题的答案节点。设（x_1，x_2，…，x_{k-1}）是状态空间树中由根到一个节点（即，问题状态）的路径，而 T（x_1，x_2，…，x_{k-1}）是下述所有 x_k 的集合，它使得（x_1，x_2，…，x_{k-1}，x_k）是一条由根到问题状态（节点 X）的路径。还假定存在一些约束条件函数 B_k，如果路径（x_1，x_2，…，x_{k-1}，x_k）不可能延伸到一个答案节点，则 B_k（x_1，x_2，…，x_{k-1}，x_k）取假值，否则，取真值。于是，解向量 X=（x_1，x_2，…，x_{n-1}，x_n）中的第 k 个分量就是选自集合 T（x_1，x_2，…，x_{k-1}）、且使得 B_k 为真的 x_k。

程序 6-5-1 回溯算法可以抽象地描述

```
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。约束条件函数 Bk（x1, x2, …, xk-1,xk）
  //给出哪些元素 X(k)满足隐式约束条件。
  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
       Bk（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)）;
       else k:=k+1; //考虑下一个集合
       end{if};
     else k:=k-1;//回溯先前的集合
     end{if};
   end{while}
end{BACKTRACK}
```

二．回溯法的效率分析

回溯法是以深度优先的方式系统地搜索问题的状态空间树。状态空间树是解空间的树表示，解空间由所有可能的决策序列$(x_1,x_2,\cdots,x_n)$构成。在搜索过程中，采用剪枝函数（约束函数和限界函数）尽量避免无意义的搜索。通过前面的具体实例容易看出，一个回溯算法的效率在很大程度上依赖于以下几个因素：

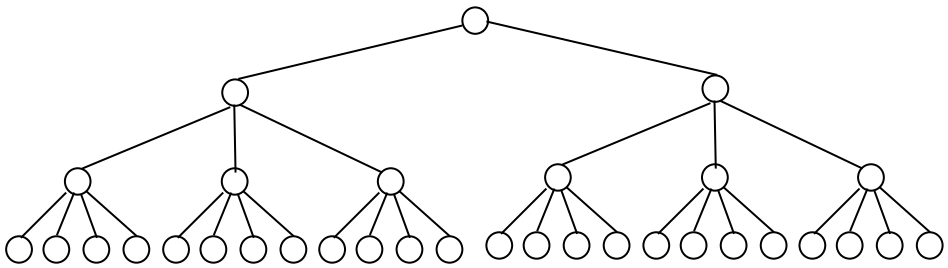
1)．产生x_k的时间；

2)． x_k的取值范围；

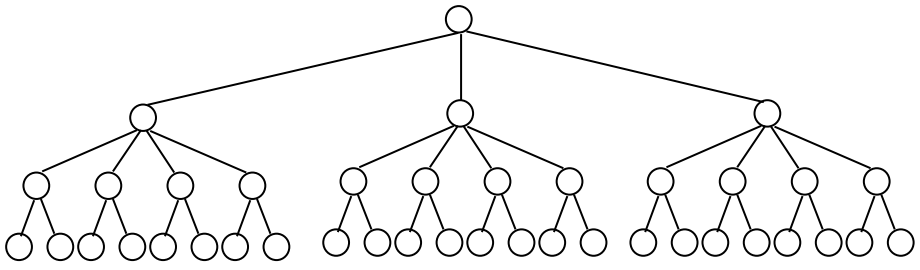
3)．计算约束函数的时间和限界函数的时间；

4)．满足约条件及限界函数条件的x_k的个数。

一般地，一个好的约束函数能够显著地减少所生成的节点的数目。但是，这样的约束函数往往计算量较大。因此在选择约束函数时通常存在着节点数与约束函数计算量之间的折中。我们希望总的计算时间较少。为了提高效率，通常采用所谓的“重排原理”。对于许多问题而言，在进行搜索试探时，选取x_k值的顺序是任意的。这提示我们，在其它条件相同的前提下，如果让***可能取值个数最少的x_k优先考虑***，则将会使算法更有效。图 6-5-3 给出了同一个问题的不同的状态空间树。在图(a)中，若从第 1 层剪去一棵子树，则从所有应当考虑的三元组中一次消去 3 个 4 元组；在图(b)中，虽然同样从第 1 层剪去一棵子树，却只从应当考虑的三元组中一次消去 4 个 2 元组。前者的效果显然比后者好，从中可以体会到这种策略的效力（但这不是绝对的，只是从信息论的角度看，平均来说更为有效，n 皇后问题就是一个反例）。



（a）



（b）

图 6-5-3 解空间树的比较

解空间结构(状态空间树)一经选定，影响回溯法效率的前三个因素就可以确定，只剩下生成节点的数目是可变的，它将随问题的具体内容以及拟定的不同生成方式而变化。即使是对于同一问题的不同实例，回溯法所产生的节点数目也会有很大变化。对于一个实例，回溯法可能只产生$O(n)$个节点,而对于另一个相近的实例，回溯法可能产生状态空间树中的所有节点。如果状态空间树的节点数是2^n或$n!$，则在最坏情况下，回溯法的时间耗费一般为$O(p(n)2^n)$或$O(q(n)n!)$,其中，$p(n)$和$q(n)$均为n的多项式。对于一个具体问题来说，回溯法的有效性往往就体现在当问题实例的规模n较大时，它能够用很少的时间求出问题的解。而对于一个问题的具体实例，我们又很难预测回溯法的行为。特别地，很难估计在解这一具体实例时所产生的节点数。这是在分析回溯法效率时遇到的主要困难。下面所介绍的概率方法也许对解决这个问题有所帮助。

当用回溯法解某一个具体问题实例之前，可用蒙特卡罗方法估算一下即将采用的回溯算法可能要产生的节点数目。该方法的基本思想是在状态空间树上产生一条随机路径，然后沿此路径来估算状态空间上满足约束条件的节点数 m。设 x 是所产生的随机路径上的一个节点,且位于状态空间树的第 i 级上(即深度为 i)。对于 x 的所有儿子节点，用约束函数检测出满足约束条件的节点数 m_i。路径上的下一个节点是从 x 的 m_i 个满足约束条件的儿子节点中随机选取的。这条路径一直延伸到一个叶节点或一个所有儿子节点都不满足约束条件的节点为止。通过这些 m_i 的值，就可估计出状态空间树中满足约束条件的节点的总数 m。在用回溯法求问题的所有可行解时，这个估计值特别有用。因为在这种情况下，状态空间树所有满足约束条件的节点都必须生成，设这样节点的总数是 m。若只要求用回溯法找出问题的一个解,则所生成的节点一般只是 m 个满足约束条件的节点中的一部分。此时用 m 来估计回溯法生成的节点数就过于保守了。为了从 m_i 的值求出 m 的值，还需要对约束函数做一些假定。在估计 m 时，假定所有约束函数是静态的，即在回溯法执行过程中，约束函数并不随随算法所获得信息的多少而动态地改变。进一步还假定对状态空间树中***同一级的节点所用的约束函数是相同的。***对于大多数回溯法，这些假定都太强了。实际上，在大多数回溯法中，约束函数是随着搜索过程的深入而逐渐加强的。在这种情形下，按照前面所做假定来估计 m 就显得保守。如果将约束函数变化的因素也加进来考虑，所得出的满足约束条件的节点总数将会少于 m，而且会更精确些。

在静态约束函数假设下，第 0 级共有 m_0 个满足约束条件的节点。若状态空间树的同一级的节点具有相同的出度，则在第 0 级上的每个节点将会有 m_1 个儿子节点满足约束条件。因此，第 1 级将会有 m_0m_1 个满足约束条件的节点。同理，第 2 级上满足约束条件的节点个数是 $m_0m_1m_2$。依此类推，第 i 级上满足约束条件

的节点的数目是 $m_0m_1m_2\cdots m_i$. 因此，对于给定的实例，如果产生状态空间树上的一条途径，并求出 m_0，m_1，m_2，…，m_i，…，则可以估计出回溯法要生成的满足约束条件的节点数目为 $m = m_0 + m_0m_1 + m_0m_1m_2 + \cdots$。假定回溯法要找出所有的可行解，设$(x_1,x_2,\cdots,x_{i-1})$是状态空间树中由根到一个节点的路径，而$T(x_1,x_2,\cdots,x_{i-1})$表示所有这样的$x_i$，使得$(x_1,x_2,\cdots,x_{i-1},x_i)$能够成为状态空间树中的一条路径。$B_k$表示第 k 级上约束函数：$B_k$(X[1], …, X[k])=true 表示这条路径到目前为止未违背约束条件。否则 B_k(X[1], …, X[k])=false。

程序 6-5-2 回溯法的递归流程

```
RecurBackTrack（k）//进入算法时，解向量 X[1..n]的前 k-1 个分
  //量 X[1],…, X[k-1]已经赋值
  global n, X[1..n];
  for 每个满足
    “X[k]∈ T(X[1], …, X[k-1]) and Bk(X[1], …, X[k])=true”
    的 X[k]  do
    if (X[1], …, X[k]) 是一条抵达某一答案节点路径 then
       print (X[1], …, X[k]);
    end{if}
    RecurBackTrack（k+1）
  end{for}
end{RecurBackTrack}
```

根据前面的分析，我们不能给出回溯法递归流程生成的状态空间树中节点数目 m。但是，可以采用下面程序给出状态空间树中节点个数的估计。

程序 6-5-3 回溯法效率估计

```
Estimate //程序沿着解空间树中一条随机路径估计回溯法生成的顶点
  //总数 m
  m:=1; r:=1; k:=1;
  loop
    T:={X[k]| X[k]∈ T(X[1], …, X[k-1]) and
    Bk(X[1], …, X[k])=true};
    if T ={} then exit; end{if}
    r:=r*size(Tk); m:=m+r;
```

```
    X[k]:=Choose(T);  k:=k+1;
  end{loop}
  return(m);
end{Estimate}
```

其中 Choose(T)是从 T 中随机地挑选一个元素。

当用回溯法求解某个具体问题时，可用算法 Estimate 估算回溯法生成的节点数。若要估计得精确些，可选取若干条不同的随机路径（通常不超过 20 条），分别对各随机路径估计节点数，然后以平均值作为 m。例如 8 皇后问题，回溯算法 nQueens 可以用 Estimate 来估计。参本章附页的“效率分析例图”给出了算法 Estimate 产生的 5 条随机路径所相应的 8×8 棋盘状态。当需要在棋盘上某行放入一个皇后时，所放的列是随机选取的。它与已在棋盘上的其它皇后互不攻击。在图中棋盘下面列出了每层可能生成的满足约束条件的节点数，即

$$m_0,\ m_1,\ m_2,\ \cdots,\ m_i,\ \cdots,$$

以及由此随机路径算出的节点总数 m 的值。这 5 个总点数的均值为 1812.2。

注意到 8 皇后状态空间树的节点总数是：

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

1812.2/109601≈1.65%，可见，回溯法效率远远高于穷举法。

习题六

1. 网络设计问题。石油传输网络通常可表示为一个非循环带权有向图 G。G 中有一个称为源的顶点 s。石油从该顶点输送到 G 的其它顶点。图 G 中每条边的权表示该边连接的两个顶点间的距离。网络中的油压随距离增大而减小。为了保证整个输油网络正常工作，需要维持网络中的最低油压 Pmin。为此需要在网络某些或全部顶点处设置增压器。在设置增压器的顶点处油压可升至最大值 Pmax。油压从 Pmax 减至 Pmin 可使石油传输的距离至少为 d。试设计一个算法，计算出网络中增压器的最优放置方案，使得用最少的增压器保证石油运输畅通。

2. 工作分配问题。设有 n 件工作需要分配给 n 个人去完成。将工作 i 分配给第 j 个人完成所需要的费用为 c_{ij}。试设计一个算法，为每一个人分配一件不同的工作，并使总费用达到最小。

3. 最佳调度问题。假设有 n 个任务要由 k 个可并行工作的机器来完成。完成任务 i 需要的时间为 t_i。试设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的结束时间最早。

4. 世界名画陈列馆问题。世界名画陈列馆由 m×n 个陈列室组成。为了防止名画被盗，需要在陈列室中设置机器人哨位。每个警卫机器人除了监视它所在的陈列室外，还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使得名画陈列馆中的每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人最少。

5. 最小重量机器设计问题。设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

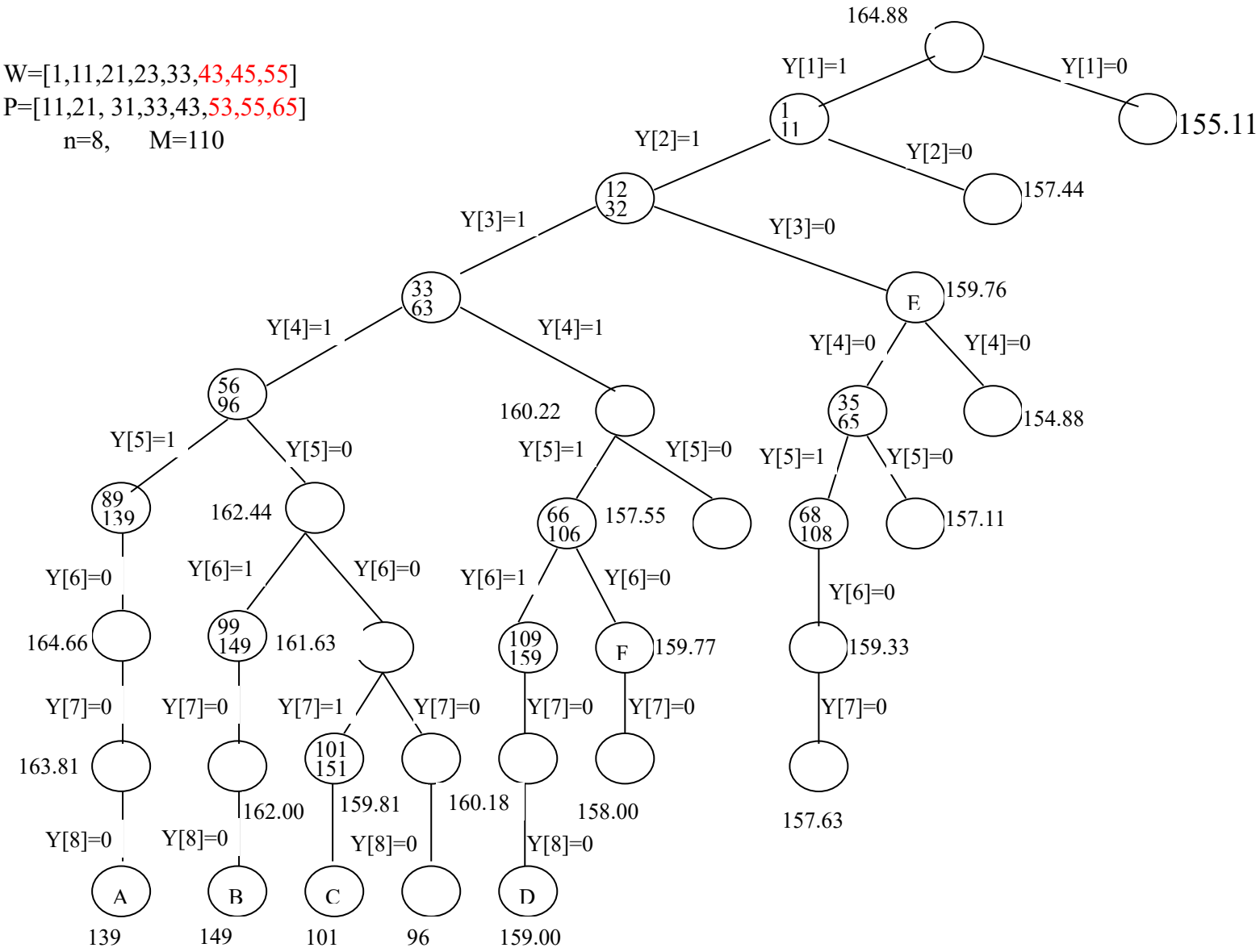
0/1 背包

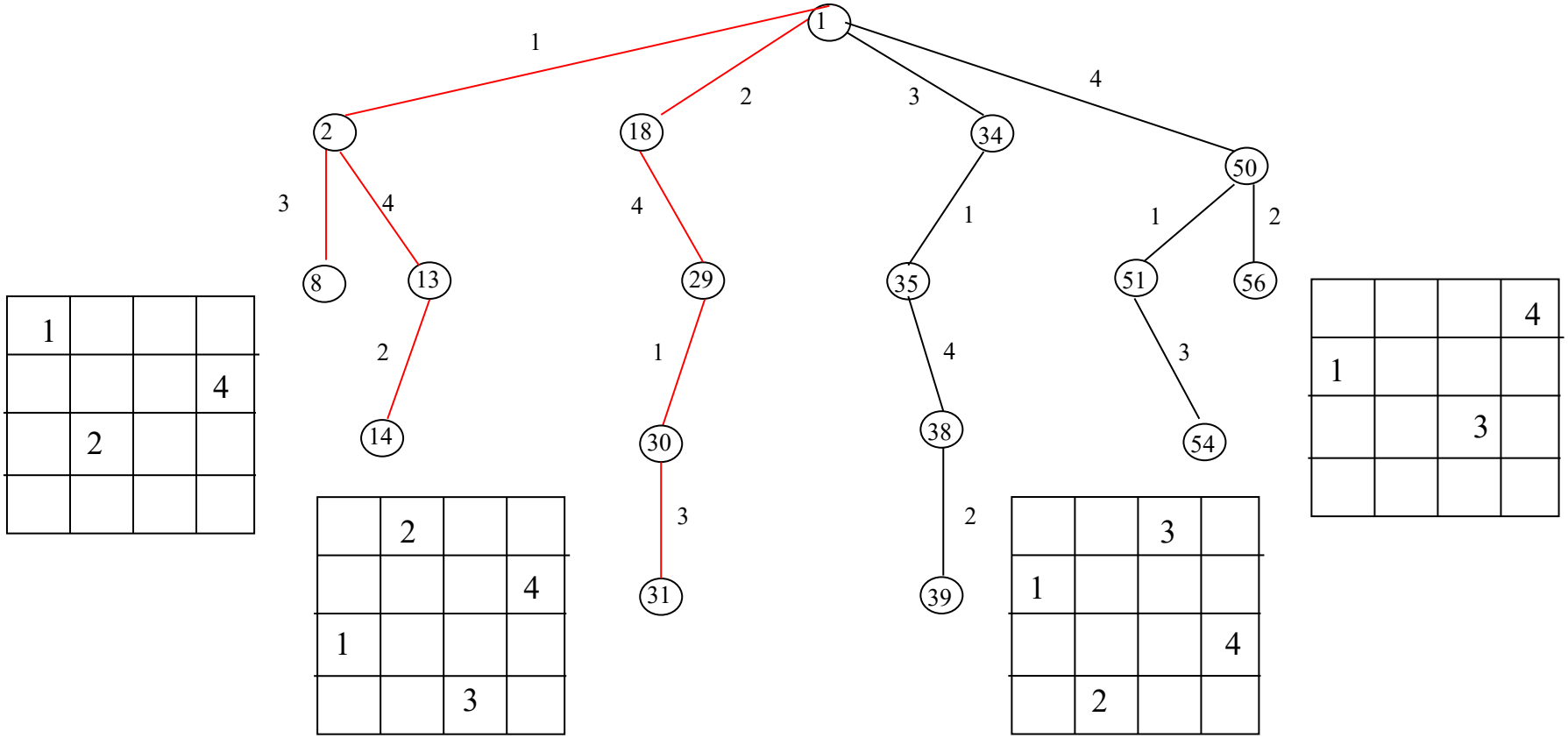
问题

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

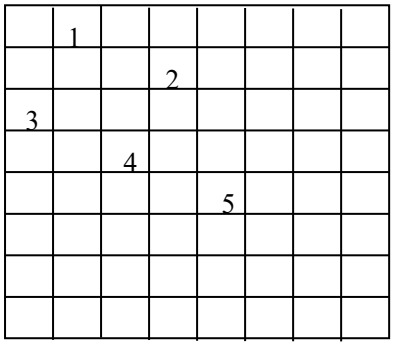
解空间

搜索情况

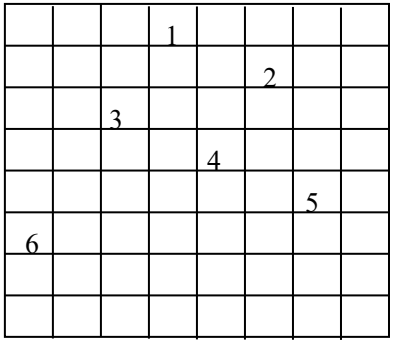




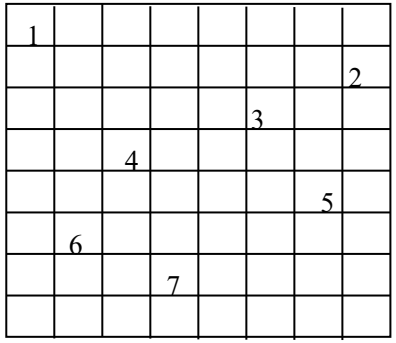
4 皇后解空间树搜索情况



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



(8,5,3,1,2,1)=769



(8,6,4,2,1,1,1)=1785

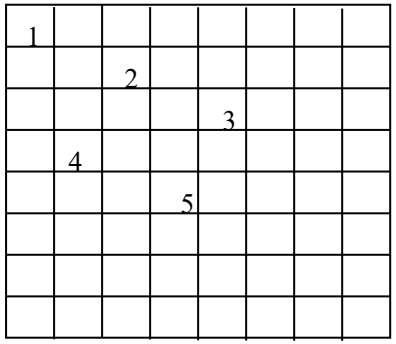
8 皇后解空间树的顶点

总数是：

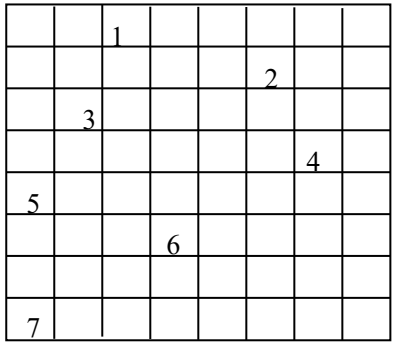
$$1+\sum_{j=0}^{7}\prod_{i=0}^{j}(8-i)=\mathbf{109601}$$

相当于

1702/109601≈1.55%



(8,6,3,2)=1977



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