第 5 章 回溯与分支限界

本章介绍算法设计的第四种方法：回溯算法. 先用例子说明回溯算法设计的基本思想和适用条件，然后给出主要设计步骤和效率分析方法，最后给出求解优化问题的一种改进回溯方法：分支限界.

5.1 回溯算法的基本思想和适用条件

有些问题，如搜索问题和优化问题，它们的解分布在一个解空间里，求解这些搜索问题的算法就是一种遍历搜索解空间的系统方法，所以解空间又称为搜索空间. 求解搜索问题就是在搜索空间中找到一个或全部解，求解组合优化问题就是找到该问题的一个最优解或所有的最优解.

回溯算法将搜索空间看作一定的结构，通常为树形结构，一个解对应于树中的一片树叶. 算法从树根(即初始状态)出发，尝试所有可达的结点. 当不能前行时，就后退一步或若干步，再从另一个结点继续搜索，直到所有的结点都试探过. 回溯算法遍历一棵树可以用深度优先、宽度优先或者宽度－深度结合等多种方法. 为加快搜索，人们又给出了分支限界等各种在树中剪枝的方法，以改善算法的运行时间. 简单来说，回溯(backtracking)是一种遵照某种规则(避免遗漏)、跳跃式(带裁剪)地搜索解空间的技术.

5.1.1 几个典型的例子

例 5.1 8 皇后问题. 在有 8×8 个方格的棋盘中放置 8 个皇后，使得任何两个皇后之间不能互相攻击，即在同一行、同一列不能有两个及以上的皇后，在与主对角线、副对角线的平行线上也不能有两个及以上的皇后，试给出所有的放置方法.

解 首先找出所有可行解，即所有可能的放置方法. 由于每行不能有两个及以上皇后，而棋盘共有 8 行，要放置的皇后个数也恰有 8 个，所以在可行解中每行正好有一个皇后. 这样，每个可行解可以表示成一个 8 维向量$<x_1,x_2,\cdots,x_8>$，其中 x_i 表示第 i 行放置皇后的位置(列号). 例如$<4,2,7,1,3,5,8,6>$，表示第一行中皇后放在第 4 列，…，第 8 行中皇后放在第 6 列. 所以所有可行解为如下 8 维向量构成的集合$\{<x_1,x_2,\cdots,x_8>\mid 1\leqslant x_i\leqslant 8,\ 1\leqslant i\leqslant 8\}$. 将这些可行解按一定的结构进行排列. 在本例中，我们将之排成完全 8 叉树，即搜索空间. 搜索树有 8 层，最下层有 8^8 个叶结点. 如图 5.1 所示.

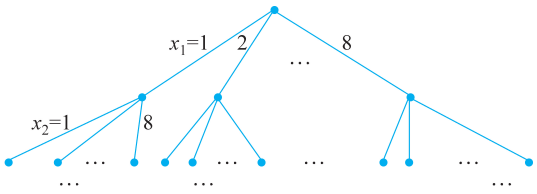

图 5.1　8 皇后问题的搜索空间

回溯算法用深度优先策略遍历整棵树，找出所有解. 算法从树根开始，经过结点<1>，<1,1>，<1,1,1>，…，最后到结点<8,8,…,8>为止，然后回溯到根，算法停止，恰好按深度优先顺序跳跃式搜索了所有的可行解向量. 在实际搜索过程中不是真正遍历所有的结点，如果发现向下搜索不可能达到解结点，那么就回头. 搜索过程是解向量不断生成的过程. 根结点为空向量，算法依次对 $x_1,x_2,\cdots,x_n$ 进行赋值. 每进行一次赋值需要检查“互不攻击”的条件. 当不满足条件时，算法不再继续向下搜索，而是从这个结点回到父结点. 一旦对 $x_1,x_2,\cdots,x_k$ 进行了赋值，算法就到达某个结点，将该结点标记为向量$<x_1,x_2,\cdots,x_k>$，称为部分向量. 从根结点到解结点的路径上的所有标记正是从空向量到可行解的生成过程. 例如，算法从根结点沿着边 $x_1=1$ 走到第一层最左边的那个结点，表示第 1 行的皇后放在第 1 列，该结点的标记是<1>. 按深度优先策略算法沿 $x_2=1$ 走到第二层最左边的那个结点，表示第 2 行的皇后放在第 1 列，这违反了同一列不能有两个及以上的皇后的放置规则，不满足约束条件，算法回溯到父结点<1>. 下一个选择是 $x_2=2$，走到第二层从左边数的第二个结点，表示第 2 行的皇后放在第 2 列，这违反了与主对角线平行的平行线上不能有两个及以上的皇后的放置原则，所以也不可能找到解. 于是，算法又回溯到父结点<1>. 这相当于把这两个分支对应的子树从整棵树中裁剪掉了. 接着算法沿着 $x_2=3$ 走到第二层左边数的第三个结点，表示第 2 行的皇后放在第 3 列，这符合放置规则，该结点标记为<1,3>. 再按照深度优先策略往下探查，显然沿 $x_3=1,x_3=2,x_3=3,x_3=4$ 的方向向下的搜索都违背了放置规则，只能回溯到<1,3>. 下面可以选择 $x_3=5$，对应于该结点的部分向量记为<1,3,5>. 照这样不断向下搜索，如果得到一个满足约束条件的 8 维向量，它就是 8 皇后问题的一个解. 如果从某个结点向下分支的所有方向都破坏了约束条件，部分向量不能继续扩张，则意味着以这个结点为根的子树中没有可行解存在. 算法从这个结点继续向上回溯到其父结点，接着探查父结点其他可能的向下分支. 如此下去，可以得到第一个解<1,5,8,6,3,7,2,4>，按此策略再遍历其他结点，可得到其余解，共有 92 个解.

一般地，对于 n 皇后问题，搜索树有 $1+n+n^2+\cdots+n^n$ 个结点，而

$$1+n+n^2+\cdots+n^n=\frac{n^{n+1}-1}{n-1}\leqslant\frac{n^{n+1}}{\frac{n}{2}}=2n^n \quad n\geqslant 2$$

在每个结点处，要判断此位置的皇后与已经放置的皇后是否相互攻击，最多要看 $3n$ 个位置(沿列方向、主与副对角线方向)是否已有皇后，故 n 皇后问题的该算法在最坏情形下的时间复杂度为 $O(3n\times 2n^n)=O(n^{n+1})$. 这是一个粗略的上界估计. 实际运行中由于搜索树的剪枝，时间要少得多.

例 5.2　0-1 背包问题. 一个旅行者准备随身携带一个背包. 可以放入背包的物品有

n 种,每种物品只有一个,重量和价值分别为 w_j 和 v_j,$1 \leqslant j \leqslant n$. 如果背包的最大重量限制是 B,问怎样选择放入背包的物品,使得背包的价值最大?

解 一个可行解就是对每个物品的选择,选择其是放入背包还是不放,放入背包就标记为1,不放入背包就标记为0. 这样,所有可行解是满足 $\sum_{i=1}^{n} w_i x_i \leqslant B$ 的 n 维布尔向量 $<x_1, x_2, \cdots, x_n>$,其中 $x_i = 1$ 或 $x_i = 0$,$1 \leqslant i \leqslant n$.

搜索空间的每片树叶对应了物品的一个子集,所有的树叶构成集合$\{<x_1, x_2, \cdots, x_n> | x_i = 1$ 或 $0, 1 \leqslant i \leqslant n\}$,整个搜索空间可以排成一棵完全二叉树. 对于每个内结点来说,到达左子结点的边代表1,到达右子结点的边代表0. 这样的完全二叉树称为子集树,该树有 2^n 个树叶,代表了选入物品子集的特征序列.

搜索策略还是深度优先方法. 下面以一个实例说明0-1背包问题的回溯算法的运行过程:设 $n=4$,价值向量为 $\boldsymbol{V}=\{12,11,9,8\}$,重量向量为 $\boldsymbol{W}=\{8,6,4,3\}$,背包承载量为 $B=13$.

按深度优先策略,从根结点开始,沿着边 $x_1=1$ 走到第一层最左边的那个结点,表示第一个物品被选中放入背包,此时背包中只有第一个物品,其重量是8,不超过背包承载量13,可以继续选择. 于是,搜索按深度优先策略向下层进行. 算法沿着 $x_2=1$ 走到第二层最左边的那个结点,表示第二个物品也被选中放入背包,此时背包中物品的重量达到14,大于背包的承载量,破坏了约束条件,回溯到父结点. 接着按深度优先策略沿边 $x_2=0$ 向下搜索,表示第二个物品不放入背包,此时背包中物品重量是8,还可以继续选择物品放入. 继续沿着左边的 $x_3=1$ 走到其左儿子,表示第三个物品被选中放入背包,此时背包中物品的重量是12,小于承载量13. 继续向下,沿着 $x_4=1$ 走到其左儿子,表示第四个物品被选中放入背包,此时背包中物品的重量将达到15,大于承载量13,破坏约束条件,回溯到父结点. 于是算法只能沿着右边的 $x_4=0$ 走到其右儿子,表示第四个物品不放入背包,此时背包中物品的重量是12,小于承载量13,从而得到第一个可行解$<1,0,1,0>$,即将第1个和第3个物品放入背包,第2个和第4个物品不放,此时放入背包物品的价值为21,现在还不知道这个可行解是不是最优解. 如此下去,算法搜索到$<0,0,0,0>$结点以后,将沿着树中最右边的路径逐步回溯,直到树根,算法结束,如图5.2所示. 此时已经找到了所有的可行解,所有这些价值中的最大值即为最优值,对应的解即为最优解. 在这个实例中,$<0,1,1,1>$是最优解,即将第2个、第3个和第4个物品放入背包,背包中物品总价值最大达到28.

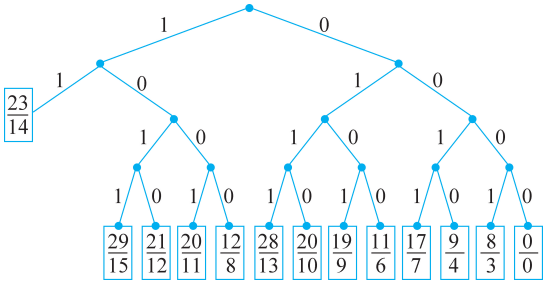

图 5.2 0-1 背包问题的一个实例

该子集树有 $1+2+2^2+\cdots+2^n=2^{n+1}-1 \leqslant 2 \times 2^n = O(2^n)$ 个结点. 在每个结点处要计算放入背包物品的重量是否超过背包的承载量,父结点已经记录了在此结点之前已经放入

背包物品的重量. 再加上当前新放入背包物品的重量,就可得到该结点处放入背包物品的重量. 在最坏的情况下,该算法只进行了 1 次加法和一次大小比较,即进行了 $O(1)$次运算,从而该算法在最坏的情况下的时间复杂性为 $O(2^n)$.

对于一般的背包问题,其可行解和 0-1 背包问题类似,也是 n 维向量$<x_1,x_2,\cdots,x_n>$,只是其分量 x_i 是整数,不一定只是 0 或 1. 由于背包承载量是一个定值 B,所以第 i 种物品放入背包的最大个数 x_i 满足 $0\leqslant x_i\leqslant\lfloor B/w_i\rfloor,1\leqslant i\leqslant n$. 在构造解空间结构时,根结点的分支个数为$\lfloor B/w_1\rfloor$,第一层结点的分支个数为$\lfloor B/w_2\rfloor$. 以此类推,可以构造完整的搜索树. 搜索策略仍然是深度优先方法,此处略.

例 5.3 货郎问题(TSP). 某售货员要到若干城市去推销商品,各城市之间的距离为已知. 他要选定一条从驻地出发经过所有城市最后回到驻地的周游路线,使得总的路程最短. 其数学模型是:已知一个带权完全图(结点代表城市,边代表城市之间的道路,权代表城市之间的距离,如两个城市之间无直接连接的道路,设其权为∞,所以权为正数或无穷),求权和最短的一条哈密顿回路.

货郎问题中,带权图又可分为有向图和无向图。下面的算法对有向图和无向图都适用.

解 一个可行解是 n 个城市的一个排列,排列的第一个元素是售货员的驻地。最后一个元素是其最后周游的城市,由此城市又回到驻地. 如果将城市编号为 $1,2,\cdots,n$,且驻地为城市 1,则一个可行解即为首元素是 1 的一个 n 元排列.

这些排列可以安排成如下的搜索树:从根到叶结点的一条路径对应了$\{1,2,\cdots,n\}$的排列,根结点只有一个子结点,其余各结点分支数不同. 第一层结点有 $n-1$ 个分支,第二层结点有 $n-2$ 个分支,…,第 $n-1$ 层有 1 个分支,这样的树称为排列树.

搜索策略依然是深度优先.

图 5.3 的右边给出货郎问题的一个实例,其对应的搜索树为左边的排列树. 按深度优先策略,可求出最优解为$<1,2,4,3>$,对应于巡回路线:1→2→4→3→1,巡回路径的长度为 $5+2+7+9=23$.

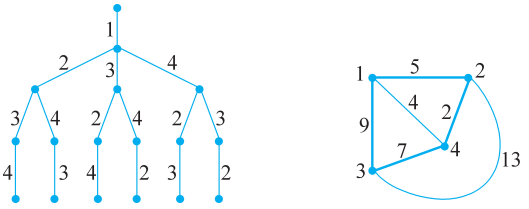

图 5.3 货郎问题的一个实例

该排列树有 $K_n=1+1+(n-1)+(n-1)(n-2)+\cdots+((n-1)(n-2)\cdots2)+(n-1)!$ 个结点. 而

$$K_n=1+(n-1)!\times\left(\frac{1}{(n-1)!}+\frac{1}{(n-2)!}+\frac{1}{(n-3)!}+\cdots\frac{1}{1!}+1\right)$$

$$\leqslant 1+(n-1)!\times\left(\frac{1}{2^{n-2}}+\frac{1}{2^{n-3}}+\cdots+\frac{1}{2}+1+1\right)$$

$$\leqslant 1+(n-1)!\times\left(\frac{1}{1-\frac{1}{2}}+1\right)$$

$$=1+3(n-1)!=O((n-1)!)$$

而在每个结点处要计算已得到的路径长度,这只要将父结点得到的路径长度加上父结点到本结点的距离即可;在叶结点处还要计算得到的回路长度,并判断得到的回路是否为当前的最短回路,所以算法在每个结点处最多进行两次加法和一次大小比较,故该算法最坏情形的时间复杂性为 $O((n-1)!)\times O(1)=O((n-1)!)$.

由这些例子可以看出回溯算法的共同特征:

(1) 可求解搜索问题和优化问题,搜索问题可定义如下:

一个搜索问题 π 有实例集 $D\pi$,对于 π 中的任何实例 I,有一个有穷的解集合 $S_\pi[I]$.

如果存在算法 A,对于任何实例 $I\in D\pi$,A 都停止,并且如果 $S\pi[I]=\varnothing$,则回答无解,否则给出 $S\pi[I]$中的一个解,那么称 A 解搜索问题 π.

(2) 搜索空间是一棵树,每个结点对应了部分向量,满足约束条件的树叶对应了可行解,在优化问题中不一定是最优解.

(3) 搜索过程一般采用深度优先、宽度优先、函数优先或宽深结合等策略隐含遍历搜索树. 所谓隐含遍历是指:不是真正访问到每个结点,需要从搜索树中进行裁剪.

(4) 判定条件(分支与回溯条件):满足约束条件则分支扩张解向量;不满足约束条件,回溯到该结点的父结点.

5.1.2 回溯算法的适用条件

要使回溯算法得到正确应用,必须满足如下的多米诺性质:

假设 $P(x_1,x_2,\cdots,x_i)$是关于向量$<x_1,x_2,\cdots,x_i>$的某个性质(如例 5.1 中的前 i 个皇后放置在彼此不能攻击的位置),那么 $P(x_1,x_2,\cdots,x_{i+1})$是真蕴涵 $P(x_1,x_2,\cdots,x_i)$为真,即

$$P(x_1,x_2,\cdots,x_{k+1})\rightarrow P(x_1,x_2,\cdots,x_k),\quad 0<k<n$$

其中 n 代表解向量的维数.

下面是一个不满足多米诺性质因而使用回溯算法不能得到正确解的反例.

例 5.4 求满足下列不等式的所有整数解:

$$5x_1+4x_2-x_3\leqslant 10$$

$$1\leqslant x_i\leqslant 3,\quad i=1,2,3$$

$P(x_1,x_2,\cdots,x_k)$表示将 $x_1,x_2,\cdots,x_k$ 代入原不等式的相应部分使得左边小于或等于 10,如 $P(x_1,x_2,x_3)$表示 $5x_1+4x_2-x_3\leqslant 10$. 存在$<x_1,x_2,x_3>=<1,2,3>$,使得$P(x_1,x_2,x_3)$为真,但是 $P(x_1,x_2)$表示 $5x_1+4x_2\leqslant 10$,显然为假,因而 P 不满足多米诺性质,不能使用回溯算法. 如果使用回溯算法,其执行过程是:

搜索空间是$\{<x_1,x_2,x_3>\mid 1\leqslant x_i\leqslant 3,i=1,2,3\}$,是一棵完全 3 叉树,如图 5.4 所示. 搜索策略还是深度优先.

回溯算法很容易求出$<1,1,1>$,$<1,1,2>$,$<1,1,3>$是其解,但当算法搜索到结点 A 时,$x_1=1$,$x_2=2$,此时 $5x_1+4x_2=13>10$,不满足条件,算法将回溯到其父结点,进而搜

索结点 B，从而丢掉<1,2,3>这个解. 如果令 $x_3'=4-x_3$，那么不等式将改为

$$5x_1+4x_2+x_3'\leqslant 14$$
$$1\leqslant x_i,\quad x_3'\leqslant 3,\quad i=1,2$$

则该不等式满足多米诺性质，可以使用回溯算法. 对所得到的解 x_1，x_2，x_3'很容易地转换成原来不等式的解 x_1,x_2,x_3.

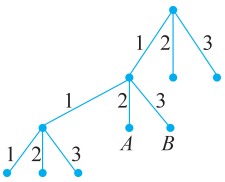

图 5.4　不等式求解的搜索树

回溯算法中，结点状态随算法的进行会改变，结点一般有三种状态：白结点(尚未访问)、灰结点(正在访问以该结点为根的树中结点)、黑结点(该结点为根的子树遍历完成).

5.2　回溯算法的设计步骤

回溯算法设计的主要步骤如下：

(1) 定义搜索问题的解向量和每个分量的取值范围.

(2) 确定子结点的排列规则.

(3) 判断是否满足多米诺性质.

(4) 确定搜索策略：深度优先、宽度优先、宽深结合等.

(5) 确定每个结点能够分支的约束条件.

(6) 确定存储搜索路径的数据结构.

回溯算法一般可以如下描述：设解向量为$<x_1,x_2,\cdots,x_n>$，x_i 的可能取值的集合为 X_i，$i=1,2,\cdots,n$. 设当 $x_1,x_2,\cdots,x_{k-1}$确定以后 x_k 的取值集合为 S_k，显然 $S_k\subseteq X_k$. 回溯算法的实现可以有两种方法：递归回溯和迭代递归. 下面给出算法的伪码.

5.2.1　回溯算法的递归实现和迭代实现

算法 5.1　ReBack(k)

1. if $k>n$ then $<x_1,x_2,\cdots,x_n>$是解
2. else while $S_k\neq\varnothing$ do
3. 　　$x_k\leftarrow S_k$ 中最小值
4. 　　$S_k\leftarrow S_k-\{x_k\}$
5. 　　计算 S_{k+1}
6. 　　ReBack($k+1$)

算法 5.2　递归回溯 ReBacktrack(n)

输入：n

输出：所有的解

1. for $i\leftarrow 1$ to n do 计算 X_k
2. $S_1\leftarrow X_1$
3. ReBack(1)

算法 5.3　迭代回溯 Backtrack(n)

输入：n

输出：所有的解

```
1. 对于 i=1,2,…,n,确定 X_i
2. k←1
3. 计算 S_k
4. while S_k≠∅ do
5.       x_k←S_k 中最小值; S_k←S_k-{x_k}
6.       if k<n then
7.            k←k+1; 计算 S_k
8.       else <x_1,x_2,…,x_n>是解
9.  if k>1 then k←k-1;goto 4
```

以 4 皇后问题为例,上述算法执行的部分过程如下:

初始:$X_1=X_2=X_3=X_4=\{1,2,3,4\}$.
$k=1,S_1=\{1,2,3,4\}$,取 $x_1=1$,修改 $S_1=\{2,3,4\}$.
$k=2,S_2=\{3,4\}$,取 $x_2=3$,修改 $S_2=\{4\}$.
$k=3,S_3=\varnothing$,回溯.
$k=2,S_2=\{4\}$,取 $x_2=4$,修改 $S_2=\varnothing$.
$k=3,S_3=\{2\}$,取 $x_3=2$,修改 $S_3=\varnothing$.
$k=4,S_4=\varnothing$,回溯.
$k=3,S_3=\varnothing$,回溯.
$k=2,S_2=\varnothing$,回溯.
$k=1,S_1=\{2,3,4\}$,取 $x_1=2$,修改 $S_1=\{3,4\}$.
$k=2,S_2=\{4\}$,取 $x_2=4$,修改 $S_2=\varnothing$.
$k=3,S_3=\{1\}$,取 $x_3=1$,修改 $S_3=\varnothing$.
$k=4,S_4=\{3\}$,取 $x_4=3$,修改 $S_4=\varnothing$,得到解$<2,4,1,3>$.
$k=3,S_3=\varnothing$,回溯.
$k=2,S_2=\varnothing$,回溯.
$k=1,S_1=\{3,4\}$,取 $x_1=3$,修改 $S_1=\{4\}$.
⋮

上面描述了算法在左子树中的执行情况,后面右子树的过程与前面类似,不再重复. 下面给出几个典型的回溯算法例子.

5.2.2 几个典型的例子

例 5.5 装载问题. n 个集装箱装上两艘载重分别为 c_1 和 c_2 的轮船,w_i 为集装箱 i 的重量,且

$$\sum_{i=1}^{n} w_i \leqslant c_1 + c_2$$

问是否存在一种合理的装载方案,将 n 个集装箱装上轮船? 如果有,给出一种方案.

解 可以证明:如果装载问题有解,则存在一个使得第一条船装载量与 c_1 的差达到最小的解. 从而有如下解题思路:

(1) 用回溯算法确定使第一条船装载量 W_1 与 c_1 的差达到最小的装载方案$<x_1,x_2,\cdots,x_n>$,即使得第一条船装载量达到最大值 W_1 的装载方案;

(2) 如果 $\sum_{i=1}^{n} w_i - W_1 \leqslant c_2$,则回答 Yes,否则回答 No.

算法设计步骤如下：

(1) 解向量是$<x_1,x_2,\cdots,x_n>$，其中$x_i\in\{1,0\}$，$1\leqslant i\leqslant n$. 搜索空间是子集树.

(2) 在结点$<x_1,x_2,\cdots,x_k>$的约束条件：

$$\sum_{i=1}^{k} w_i x_i \leqslant c_1$$

(3) 满足多米诺条件：令$P(x_1,x_2,\cdots,x_k)$为$\sum_{i=1}^{k} w_i x_i > c_1$，从而

$$\sum_{i=1}^{k} w_i x_i > c_1 \Rightarrow \sum_{i=1}^{k+1} w_i x_i > c_1$$

(4) 搜索策略：深度优先.

上述回溯算法的伪码如下. 其中B为目前空隙，best为目前为止最优解的空隙.

算法 5.4 Loading(W,c_1)

输入：集装箱重量$W=<w_1,w_2,\cdots,w_n>$，c_1是第一条船的载重

输出：使得第一条船装载量最大的装载方案$<x_1,x_2,\cdots,x_n>$，其中$x_i=0,1$

```
1.  B←c1; best←c1; i←1
2.  while i≤n do
3.      if 装入 i 后重量不超过 c1
4.      then B←B−wi; x[i]←1; i←i+1
5.      else x[i]←0; i←i+1
6.  if B<best then 记录解;best←B; i←i−1
7.  Backtrack(i)
8.  if i=1 then return 最优解
9.  else goto 3
```

算法 5.5 Backtrack(i)

```
1. while i>1 and x[i]=0 do
2.    i←i−1                                  //回溯到父结点
3. if x[i]=1 then x[i]←0; B←B+wi; i←i+1      //搜索右分支
```

下面以实例说明算法执行过程. 实例：$W=<90,80,40,30,20,12,10>$，$c_1=152$，$c_2=130$. 回溯算法使用深度优先搜索策略，搜索过程如图 5.5 所示. 虚线表示回溯. 算法首先给出第一个可行解$<1,0,1,0,1,0,0>$，其装载量为150，但其后给出第二个可行解$<1,0,1,0,0,1,1>$，其装载量为 152，恰好为第一条船的承载量，因而也是最大的装载量. 第一条船装载后，货物剩下的重量为 80＋30＋20＝130，正好可装入第二条船. 所以本实例的解为：第 1、第 3、第 6 和第 7 集装箱装入第一条船，第 2、第 4 和第 5 集装箱装入第二条船.

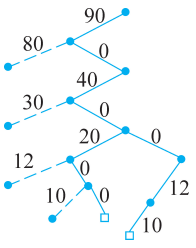

图 5.5 装载问题的实例

在最坏的情况下，算法要遍历图中几乎所有的点，叶结点有2^n个，结点总数为$O(2^n)$个. 每个结点要计算装载量以判定是否回溯，达到叶结点的计算时间为$O(1)$，所以算法的计算时间复杂度为$O(2^n)$.

例 5.6 图的m着色问题. 给定无向连通图G和m种颜色，用这些颜色给图的顶点着色，每个顶点一种颜色. 如果要求G的每条边的两个顶点着不同颜色. 给出所有可能的着色

方案;如果不存在着这样的方案,则回答 No.

解 设 G 有 n 个顶点,将顶点编号为 1,2,…,n,则搜索空间为深度 n 的 m 叉完全树.将颜色编号为 1,2,…,m,树的结点$<x_1,x_2,\cdots,x_k>$($x_1,x_2,\cdots,x_k\in\{1,2,\cdots,m\}$,$1\leqslant k\leqslant n$)表示顶点 1 着颜色 x_1,顶点 2 着颜色 x_2,…,顶点 k 着颜色 x_k.

约束条件:该顶点邻接表中已着色的顶点与该顶点没有同色的.

搜索策略:深度优先.

图 5.6 给出了一个图着色的实例,其中顶点数 $n=7$,颜色数 $m=3$. 按照 1,2,3,4,5,6,7 顺序构造搜索树. 按深度优先策略可以得到第一个着 3 色方案(图中粗线路径所示)是:顶点 1、顶点 3 和顶点 5 着色 1;顶点 2 和顶点 6 着色 2;顶点 4 和顶点 7 着色 3. 在$<1,2>$的路径上只有这一个可行解. 在$<1,3>$的路径上也只有一个可行解:顶点 1、顶点 3 和顶点 5 着色 1;顶点 4 和顶点 7 着色 2;顶点 2 和顶点 6 着色 3. 其实这个解只是将第一个解中的颜色 2 和颜色 3 交换而已. 根据对称性,在这棵树中只需搜索 1/3 的空间即可,搜索算法共可得到 6 个解.

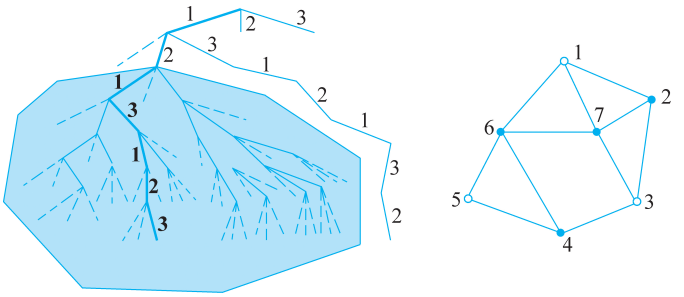

图 5.6 一个图着色问题的实例

该搜索树中有

$$1+m+m^2+\cdots+m^n=\frac{m^{n+1}-1}{m-1}\leqslant\frac{m^{n+1}}{\frac{m}{2}}=2m^n\quad(m\geqslant 2)$$

个结点,在每个结点处要判断当前顶点的着色与已着色的顶点是否颜色冲突,最坏情况下与其他所有顶点的颜色都要进行比较,即进行 $n-1$ 次比较,故该算法在最坏情况下的时间复杂性为 $O(nm^n)$.

5.3 回溯算法的效率估计和改进途径

回溯算法的时间复杂度一般取决于在搜索空间中真正遍历的结点个数以及在每个结点的工作量,而结点个数通常是指数量级. 在最坏的情况下,算法的裁剪策略几乎没有用处,往往需要遍历整个搜索空间,而平均情况下算法的复杂度比起蛮力算法会好一些. 为了估计算法运行的效率,可以用在搜索树中真正遍历的结点数作为度量标准,通常采用概率方法中的蒙特卡洛(Monte Carlo)方法来做出估计. 从根开始,算法在每个结点从所有可行的分支中随机选择一个分支. 当算法最终到达某片树叶,就得到一条随机选择的路径. 把其他的路径按照这条路径的形式进行复制,从而生成一棵对称的树,以这棵树的结点数作为本次遍

历的结点数的估计. 图 5.7 给出了一个 4 皇后问题的估计. 假若算法某次抽样的路径是<1,4,2>. 在根结点,$S_1=\{1,2,3,4\}$,因此根有 4 个儿子,这棵树第一层的 4 条路径与<1,4,2>具有相同的结构. 当第二次随机选择时,$S_2=\{3,4\}$,有两种选择,因此第二层的每个结点都有2 个分支. 当算法选择了 4 之后,下一步进行第三次随机选择时,$S_3=\{2\}$,这时算法只有一种选择,这意味着该结点只有 1 个分支,只能到达<1,4,2>结点. 因此,这棵树的第三层的每个结点都是 1 个分支. 到此为止,最终形成的树就是在图 5.7 中标记为<1,4,2>的树. 类似地,如果随机选择的路径是<1,3>和<2,4,1,3>,可以得到图中的另外两棵树. 而算法真正遍历的树就是搜索空间所表示的树. 在多次抽样中,每次形成的树可能是<1,4,2>,<1,3>和<2,4,1,3>三种树之一. 对每次抽样得到的树的结点数取平均值,就得到一个平均情况下算法真正遍历结点数的估计值.

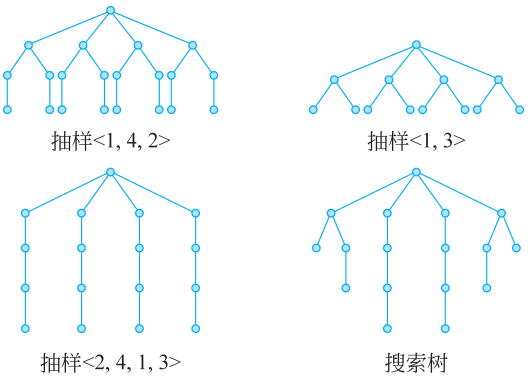

图 5.7　4 皇后问题的 Monte Carlo 估计

设 t 为取样次数,sum 为 t 次取样遍历结点的平均数,m 为本次取样树中结点总数,k 为目前访问结点的层数,r_1 为在路径中的本层分支数,r_2 为路径中的上层分支数,n 为树的层数. 下面的伪码给出上述估计方法的算法.

算法 5.6　Monte Carlo

输入：n 和 t 为正整数,n 为皇后数,t 为抽样次数

输出：sum,即 t 次抽样平均访问的结点数

```
1. sum←0
2. for i←1 to t do
3.       m←Estimate(n)                    //Estimate(n)是第 i 次抽样得到的树中的结点数
4.       sum←sum+m
5. sum←sum/t
```

算法 5.7　Estimate(n)

```
1. m←1; r2←1; k←1
2. While k≤n do
3.    if Sk=∅ then return m
4.    r1←|Sk| * r2                        //计算第 k 层的结点数
5.    m←m+r1                              //第 k 层以上各层的结点总数
6.    xk←随机选择 Sk 的元素
7.    r2←r1
8.    k←k+1
```