第5章 走不下去就回退——回溯法

回溯法采用类似穷举法的搜索尝试过程，在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时就“回溯”（即回退），尝试其他路径，所以回溯法有“通用解题法”之称。本章介绍回溯法求解问题的一般方法，并给出一些用回溯法求解的经典示例。本章的学习要点和学习目标如下：

（1）掌握问题的解空间的结构和深度优先搜索过程。

（2）掌握回溯法的原理和算法的框架。

（3）掌握剪支函数（约束函数和限界函数）的一般设计方法。

（4）掌握各种回溯法经典算法的设计过程和算法分析方法。

（5）综合运用回溯法解决一些复杂的实际问题。

5.1 回溯法概述

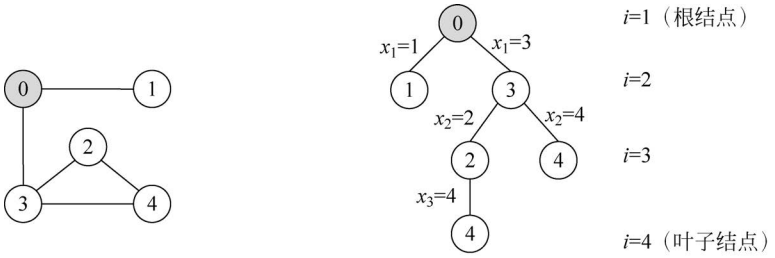
5.1.1 问题的解空间

先看求解问题的类型，通常求解问题分为两种类型，一种类型是给定一个约束函数，需要求所有满足约束条件的解，称为求**所有解类型**。例如在鸡兔同笼问题中，所有鸡兔头数为 a，所有腿数为 b，求所有头数为 a、腿数为 b 的鸡兔数，设鸡兔数分别为 x 和 y，则约束函数是 $x+y=a$，$2x+4y=b$。另一种类型是除了约束条件以外还包含目标函数，最后是求使目标函数最大或者最小的最优解，称为求**最优解类型**。例如在鸡兔同笼问题中，求所有鸡兔头数为 a、所有腿数为 b 并且鸡最少的解，这就是一个求最优解问题，除了前面的约束函数以外还包含目标函数 min(x)。这两类问题本质上是相同的，因为只有求出所有解再按目标函数进行比较才能求出最优解。

求问题的所有解涉及解空间的概念，在 3.1 节中讨论穷举法时简要介绍了解空间，这里作进一步讨论。实际上问题的一个解是由若干决策（即选择）步骤组成的决策序列，可以表示成解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，其中分量 x_i 对应第 i 步的选择，通常可以有两个或者多个取值，表示为 $x_i\in S_i(0\leqslant i\leqslant n-1)$，$S_i$ 为 x_i 的取值候选集，即 $S_i=(v_{i,0},v_{i,1},\cdots,v_{i,|S_i|-1})$。$x$ 中各分量 x_i 的所有取值的组合构成问题的解向量空间，简称为**解空间**，解空间一般用树形式来组织，树中的每个结点对应问题的某个状态，所以解空间也称为解空间树或者状态空间树。

例如，对于如图 5.1(a)所示的连通图，现在要求从顶点 0 到顶点 4 的所有路径（默认为简单路径），这是一个求所有解的问题，约束条件就是 0→4 的路径，由于路径是一个顶点序列，所以对应的解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$ 表示一条路径，这里 $x_0=0$（没有其他选择），需要求出满足约束条件的其他 $x_i(i\geqslant 1)$，这里的路径有多条，每条路径对应一个解向量，对应的解空间如图 5.1(b)所示，图中 i 表示结点的层次，由于 x_0 是固定的，只需要从 x_1 开始求，所有根结点层次 $i=1$，从中看出 $S_1=\{1,3\}$，$S_2=\{2,4\}$，$S_3=\{4\}$。在确定解空间后该问题转换为在其中求从根结点出发搜索到叶子结点并且叶子结点为顶点 4 的所有解，对应的两个解为 $x=\{0,3,2,4\}$ 和 $x=\{0,3,4\}$。如果问题是求从顶点 0 到顶点 4 的一条最短路径，属于求最优解问题，同样需要求从顶点 0 到顶点 4 的所有路径，通过比较路径长度得到一条最短路径是 $x=\{0,3,4\}$。

归纳起来，解空间的一般结构如图 5.2 所示，根结点（为第 0 层）的每个分支对应分量 x_0 的一个取值（或者说 x_0 的一个决策），若 x_0 的候选集为 $S_0=\{v_{0,1},\cdots,v_{0,a}\}$，即根结点的子树的个数为 $|S_0|$，例如 $x_0=v_{0,0}$ 时对应第 1 层的结点 A_0，$x_0=v_{0,1}$ 时对应第 1 层的结点 A_1，…。对于第 1 层的每个结点 A_i，A_i 的每个分支对应分量 x_1 的一个取值，若 x_1 的取值候选集为 $S_1=\{v_{1,0},\cdots,v_{1,b}\}$，$A_i$ 的分支数为 $|S_1|$，例如对于结点 A_1，当 $x_1=v_{1,0}$ 时对应第 2 层的结点 B_0，…。以此类推，最底层是叶子结点层，叶子结点的层次为 n，解空间的高度为 $n+1$。从中看出，第 i 层的结点对应 x_i 的各种选择，从根结点到每个叶子结点有一条路径，路径上的每个分支对应一个分量的取值，这是理解解空间的关键。



(a) 一个连通图　　(b) 解空间

图 5.1　一个连通图及其问题的解空间

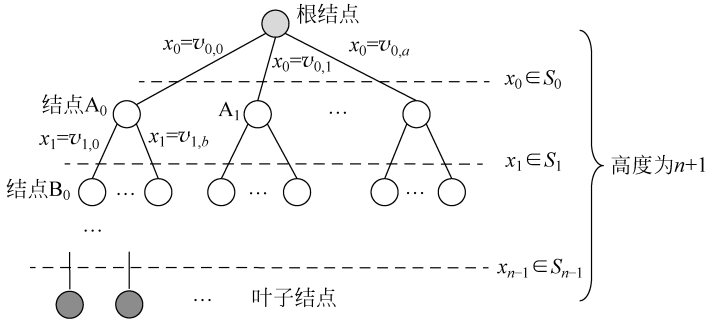


图 5.2　解空间的一般结构

从形式化角度看，解空间是 $S_0\times S_1\times\cdots\times S_{n-1}$ 的笛卡儿积，例如当 $|S_0|=|S_1|=\cdots|S_{n-1}|=2$ 时，解空间是一棵高度为 $n+1$ 的满二叉树。需要注意的是，问题的解空间是虚拟的，并不需要在算法运行中真正地构造出整棵树结构，然后在该解空间中搜索问题的解。实际上，有些问题的解空间因过于复杂或结点过多难以画出来。

5.1.2　什么是回溯法

从前面的讨论看出问题的解包含在解空间中，剩下的问题就是在解空间中搜索满足约束条件的解。所谓回溯法，就是在解空间中采用深度优先搜索方法从根结点出发搜索解，与树的遍历类似，当搜索到某个叶子结点时对应一个可能解，如果同时又满足约束条件，则该可能解是一个**可行解**。所以一个可行解就是从根结点到对应叶子结点的路径上所有分支的取值，例如一个可行解为$(a_0,a_1,\cdots,a_{n-1})$，其图示如图 5.3 所示，在解空间中搜索到可行解的部分称为搜索空间。简单地说，回溯法采用深度优先搜索方法寻找从根结点到每个叶子结点的路径，判断对应的叶子结点是否满足约束条件，如果满足，该路径就构成一个解（可行解）。

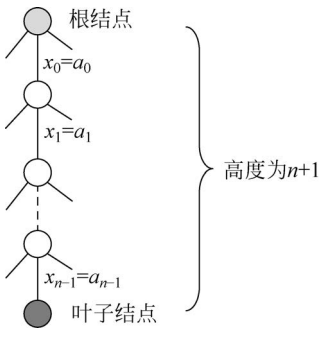


图 5.3　求解的搜索空间

回溯法在搜索解时首先让根结点成为活结点，所谓**活结点**，是指自身已生成但其孩子结点没有全部生成的结点，同时也成为当前的扩展结点，所谓**扩展结点**，是指正在产生孩子结点的结点。在当前扩展结点处沿着纵深方向移至一个新结点，这个新结点又成为新的活结点，并成为当前扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当

前扩展结点就成为死结点，所谓**死结点**，是指其所有子结点均已产生的结点，此时应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前扩展结点。

如图 5.4 所示，从结点 A 扩展出子结点 B，从结点 B 继续扩展，当结点 B 的所有子结点扩展完毕，结点 B 变为死结点，从结点 B 回退到结点 A（即回溯），通过回溯使结点 A 恢复为扩展结点 B 之前的状态，再扩展出子结点 C，此时开始做结点 C 的扩展，结点 C 就是扩展结点，由于结点 A 可能还有尚未扩展的其他子结点，结点 A 称为活结点。

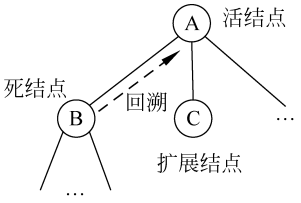


图 5.4　回溯过程

从上看出，求问题的解的过程就是在解空间中搜索满足约束条件和目标函数的解。所以设计搜索算法的关键点有以下 3 个。

① 根据问题的特性确定结点是如何扩展的，不同问题的扩展方式是不同的。例如，在求图中从顶点 s 到顶点 t 的路径时，其扩展十分简单，就是从一个顶点找所有相邻顶点。

② 在解空间中按什么方式搜索解？实际上树的遍历主要有先根遍历和层次遍历，前者就是深度优先搜索（DFS），后者就是广度优先搜索（BFS）。回溯法就是采用深度优先搜索解，第 6 章介绍的分支限界法则是采用广度优先搜索解。

③ 解空间通常十分庞大，如何高效地找到问题的解？通常采用一些剪支的方法实现。

所谓剪支，就是在解空间中搜索时提早终止某些分支的无效搜索，减少搜索的结点个数但不影响最终结果，从而提高了算法的时间性能。常用的剪支策略如下。

① 可行性剪支：在扩展结点处剪去不满足约束条件的分支。例如，在鸡兔同笼问题中，若 $a=3,b=8$，兔数的取值范围只能是 0～2，因为有 3 只或者更多只兔时腿数就超过 8 了，不再满足约束条件。

② 最优性剪支：用限界函数剪去得不到最优解的分支。例如，在求鸡最少的鸡兔同笼问题中，若已经求出一个可行解的鸡数为 3，后面就不必再搜索鸡数大于 3 的结点。

③ 交换搜索顺序：在搜索中改变搜索的顺序，比如原先是递减顺序，可以改为递增顺序，或者原先是无序，可以改为有序，这样可能减少搜索的结点总数。

严格来说，交换搜索顺序并不是一种剪支策略，而是一种对搜索方式的优化。前两种剪支策略采用的约束函数和限界函数统称为剪支函数。归纳起来，回溯法可以简单地理解为深度优先搜索加上剪支。因此用回溯法求解的一般步骤如下。

① 针对给定的问题确定其解空间，其中一定包含所求问题的解。

② 确定结点的扩展规则。

③ 采用深度优先搜索方法搜索解空间，并在搜索过程中尽可能采用剪支函数避免无效搜索。

5.1.3　回溯法算法的时间分析

通常以回溯法的解空间中的结点个数作为算法的时间分析依据。假设解空间树共有 $n+1$ 层（根结点为第 0 层，叶子结点为第 n 层），第 1 层有 m_0 个结点，每个结点有 m_1 个子结点，则第 2 层有 m_0m_1 个结点，同理，第 3 层有 $m_0m_1m_2$ 个结点，以此类推，第 n 层有 $m_0m_1\cdots m_{n-1}$ 个结点，则采用回溯法求所有解的算法的执行时间为 $T(n)=m_0+m_0m_1+m_0m_1m_2+\cdots+m_0m_1m_2\cdots m_{n-1}$。这是一种最坏情况下的时间分析方法，在实际中可以通

过剪支提高性能。为了使估算更精确,可以选取若干条不同的随机路径,分别对各随机路径估算结点总数,然后取这些结点总数的平均值。在通常情况下,回溯法的效率通常会高于穷举法。

5.2 深度优先搜索

深度优先搜索是在访问一个顶点 v 之后尽可能地先对纵深方向进行搜索,在解空间中搜索时类似树的先根遍历方式。

5.2.1 图的深度优先遍历

图的遍历是从图中某个起始点出发访问图中所有顶点并且每个顶点仅访问一次的过程,其顶点访问序列称为图的遍历序列。采用深度优先搜索方法遍历图称为图的深度优先遍历,得到的遍历序列称为深度优先遍历序列,其过程是从起始点 v 出发,以纵向方式一步一步沿着边访问各个顶点。例如,对于如图 5.1(a)所示的连通图,采用如下邻接表存储:

```
adj=[[1,3],[0],[3,4],[0,2,4],[2,3]]
```

从顶点 v 出发求深度优先遍历序列 ans 的算法如下:

```
def DFS1(adj,v):                    #深度优先遍历
    global visited,ans
    ans.append(v)                   #访问顶点 v
    visited[v]=1
    for u in adj[v]:                #找到 v 的相邻点 u
        if visited[u]==0:           #若顶点 u 尚未访问
            DFS1(adj,u)             #从 u 出发继续搜索

def DFS(adj,v):
    global visited,ans
    ans=[]                          #存放一个 DFS 序列
    visited=[0]*len(adj)            #初始化所有元素为 0
    DFS1(adj,v)
    return ans
```

上述算法求得一个深度优先遍历序列为{0,1,3,2,4}。需要注意的是,深度优先遍历特指图的一种遍历方式,而深度优先搜索是一种通用的搜索方式,前者是后者的一种应用,目前人们往往将两者等同为一个概念。

5.2.2 深度优先遍历和回溯法的差别

深度优先遍历和回溯法都是基于深度优先搜索,但两者在处理方式上存在差异,下面通过一个示例进行说明。

【例 5-1】 对于如图 5.1(a)所示的连通图,求从顶点 0 到顶点 4 的所有路径。

解 在采用深度优先遍历求 u 到 v 的所有路径时是从顶点 u 出发以纵向方式进行顶点

的搜索，用 x 存放一条路径，用 ans 存放所有的路径，如果当前访问的顶点 $u=v$，将找到的一条路径 x 添加到 ans 中，同时从顶点 u 回退，以便找其他路径，否则找到 u 的所有相邻点 w，若顶点 w 尚未访问，则从 w 出发继续搜索路径，当从 u 出发的所有路径搜索完毕，再从 u 回退。对应的算法如下：

```
import copy
def dfs11(adj,u,v,x):                       #深度优先搜索
    global visited,ans
    x.append(u)                             #访问顶点 u
    visited[u]=1
    if u==v:                                #找到一条路径
        ans.append(copy.deepcopy(x))        #将路径 x 添加到 ans 中
        visited[u]=0                        #置 u 可以重新访问
        x.pop()                             #路径回退
        return
    for w in adj[u]:                        #找到 u 的相邻点 w
        if visited[w]==0:                   #若顶点 w 尚未访问
            dfs11(adj,w,v,x)                #从 w 出发继续搜索
    visited[u]=0                            #从 u 出发的所有路径找完后回退
    x.pop()                                 #路径回退

def dfs1(adj,u,v):                          #求 u 到 v 的所有路径
    global visited,ans
    ans=[]
    visited=[0]*len(adj)                    #初始化所有元素为 0
    x=[]
    dfs11(adj,u,v,x)
    return ans
```

在调用上述 dfs1(adj,0,4)时求出的两条路径是 0→3→2→4 和 0→3→4。现在采用回溯法，对应的解空间如图 5.1(b)所示，解向量 $\boldsymbol{x}$ 表示一条路径，首先将起始点 u(初始 $u=0$)添加到 $\boldsymbol{x}$ 中，再求 $x_i(i\geqslant 1)$，当 $u=v(v=4)$时对应解空间的一个叶子结点，此时 $\boldsymbol{x}$ 中就是一条满足约束条件的路径，将其添加到 ans 中，否则从顶点 u 进行扩展，若相邻点 w 尚未访问，将 w 添加到 $\boldsymbol{x}$ 中，然后从 w 出发进行搜索，当从 w 出发的路径搜索完后再回退到顶点 u，简单地说从 u 出发搜索再回到 u，这就是回溯法的核心。对应的算法如下：

```
def dfs21(adj,u,v,x):                       #回溯法
    global visited,ans
    if u==v:                                #找到一条路径
        ans.append(copy.deepcopy(x))        #将路径 x 添加到 ans 中
    else:
        for w in adj[u]:                    #找到 u 的相邻点 w
            if visited[w]==0:               #若顶点 w 尚未访问
                x.append(w)                 #访问 v,将 v 添加到 ans 中
                visited[w]=1
                dfs21(adj,w,v,x)            #从 w 出发继续搜索
                visited[w]=0                #从 w 回退到 u
                x.pop()

def dfs2(adj,u,v):                          #求 u 到 v 的所有路径
```

```
    global visited,ans
    ans=[]                          #存放所有路径
    visited=[0]*len(adj)            #初始化所有元素为0
    x=[]
    x.append(u)                     #将起始点u添加到x中
    visited[u]=1
    dfs21(adj,u,v,x)
    return ans
```

在调用上述 dfs2(adj,0,4)时求出的两条路径同样是 0→3→2→4 和 0→3→4。从中看出,深度优先遍历主要考虑顶点 u 的前进和回退,不需要专门表示回退到哪个顶点,而回溯法主要考虑顶点 u 扩展的子结点以及从子结点的回退,需要专门处理出发点 u 和子结点 w 之间的扩展和回退关系。尽管都是采用深度优先搜索,但后者解决问题的思路更清晰,特别是对于复杂的问题求解要方便得多。

扫一扫

视频讲解

5.2.3 实战——二叉树的所有路径(LeetCode257★)

1. 问题描述

给定一棵含 $n(1\leqslant n\leqslant 100)$ 个结点的二叉树的根结点 root,结点值在[-100,100]内,设计一个算法按任意顺序返回从根结点到叶子结点的所有路径。例如,对于如图 5.5 所示的二叉树,返回结果是{"1-> 2-> 5","1-> 3"}。

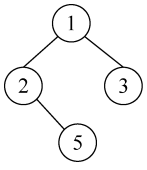


图 5.5 一棵二叉树

2. 问题求解——深度优先遍历

采用深度优先遍历。从根结点 root 出发搜索到每个叶子结点时构成一条路径 x,将其转换为字符串 tmp 后添加到 ans 中,由于是树结构,不会重复访问顶点,所以不必设置访问标记数组。对应的算法如下:

```
class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        if root==None:return []
        self.ans=[]
        x=[]                                        #存放一条路径
        self.dfs(root,x)                            #求ans
        return self.ans

    def dfs(self,root,x):                           #深度优先遍历
        x.append(root.val)
        if root.left==None and root.right==None:    #找到一条路径
            tmp=str(x[0])                           #将路径转换为字符串
            for i in range(1,len(x)):
                tmp+="->"+str(x[i])
            self.ans.append(tmp)
        else:
            if root.left!=None:
                self.dfs(root.left,x)
            if root.right!=None:
                self.dfs(root.right,x)
        x.pop()                                     #从结点root回退
```

上述程序的提交结果为通过，运行时间为 40ms，消耗的空间为 14.9MB。

3. 问题求解——回溯法

在采用回溯法时将给定的一棵树看成解空间，存放一条路径的 x 就是一个解向量。从根结点 root 出发搜索，当到达一个叶子结点时构成一条路径，将解向量 x 转换为字符串 tmp 后添加到 ans 中，否则从 root 扩展出左、右孩子结点，并从孩子结点回退到 root。对应的算法如下：

```
class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        if root==None:return []
        self.ans=[]
        x=[]                                          #存放一条路径
        x.append(root.val)
        self.dfs(root,x)                              #求 ans
        return self.ans

    def dfs(self,root,x):                             #回溯算法
        if root.left==None and root.right==None:      #找到一条路径
            tmp=str(x[0])                             #将路径转换为字符串
            for i in range(1,len(x)):
                tmp+="->"+str(x[i])
            self.ans.append(tmp)
        else:
            if root.left!=None:
                x.append(root.left.val)
                self.dfs(root.left,x)
                x.pop()                               #回溯
            if root.right!=None:
                x.append(root.right.val)
                self.dfs(root.right,x)
                x.pop()                               #回溯
```

上述程序的提交结果为通过，运行时间为 44ms，消耗的空间为 14.9MB。

5.3 基于子集树框架的问题求解

5.3.1 子集树算法框架概述

通常求解问题的解空间分为子集树和排列树两种类型。当求解问题是从 n 个元素的集合 S 中找出满足某种性质的子集时，相应的解空间树称为**子集树**，在子集树中每个结点的扩展方式是相同的，也就是说每个结点的子结点的个数相同。例如在整数数组 a 中求和为目标值 target 的所有解，每个元素 $a[i]$ 只有选择和不选择两种方式，对应的解空间就是子集树。假设子集树的解空间的高度为 $n+1$，每个非叶子结点有 c 个子结点，对应算法的时间复杂度为 $O(c^n)$。

设问题的解是一个 n 维向量 $(x_1,x_2,\cdots,x_n)$，约束函数为 constraint(i,j)，限界函数为

bound(i,j),解空间为子集树的递归回溯框架如下:

```
x=[0] * MAXN                                  #x存放解向量,这里作为全局变量
def dfs(i):                                   #求解子集树的递归框架
    if i>n:                                   #搜索到叶子结点,输出一个可行解
        输出一个解
    else:
        forj in range(下界,上界+1):           #用j表示x[i]的所有可能候选值
            x[i]=j                            #产生一个可能的解分量
            ...                               #其他操作
            if constraint(i,j) and bound(i,j):
                dfs(i+1)                      #满足约束条件和限界函数,继续下一层
            回溯 x[i]
            ...
```

在采用上述算法框架时需要注意以下几点。

① 如果 i 从 1 开始调用上述递归框架,此时根结点为第 1 层,叶子结点为第 $n+1$ 层。当然 i 也可以从 0 开始,这样根结点为第 0 层,叶子结点为第 n 层,所以需要将上述代码中的"if i>n"改为"if i>=n"。

② 在上述递归框架中通过 for 循环用 j 枚举 x_i 的所有可能候选值,如果扩展路径只有两条,可以改为两次递归调用(例如求解 0/1 背包问题、子集和问题等都是如此)。

③ 这里递归框架只有 i 一个参数,在实际应用中可以根据具体情况设置多个参数。

5.3.2 实战——子集(LeetCode78★★)

1. 问题描述

见 3.3.5 节,这里采用回溯法求解。

扫一扫

视频讲解

2. 问题求解 1

本问题的解空间是典型的子集树,假设求集合 a 的所有子集(即幂集),集合 a 中的每个元素只有两种选择,要么选取,要么不选取。设解向量为 x,$x[i]=1$ 表示选取 $a[i]$,$x[i]=0$ 表示不选取 $a[i]$。用 i 遍历数组 a,i 从 0 开始(与解空间中根结点的层次为 0 相对应),根结点为初始状态($i=0$,x 的元素均为 0),叶子结点为目标状态($i=n$,x 为一个可行解,即一个子集)。从状态(i,x)可以扩展出两个状态。

① 选择 $a[i]$元素 ⇨ 下一个状态为($i+1$,$x[i]=1$)。

② 不选择 $a[i]$元素 ⇨ 下一个状态为($i+1$,$x[i]=0$)。

这里 i 总是递增的,所以不会出现状态重复的情况。如图 5.6 所示为求{1,2,3}幂集的解空间,每个叶子结点对应一个子集,所有子集构成幂集。

对应的递归回溯算法如下:

```
1  def subset1(a):                    #解法1:求a的幂集
2      x=[0] * len(a)                 #解向量
3      dfs1(a,x,0)
4
5  def dfs1(a,x,i):                   #回溯算法
6      if i>=len(a):                  #到达一个叶子结点
```

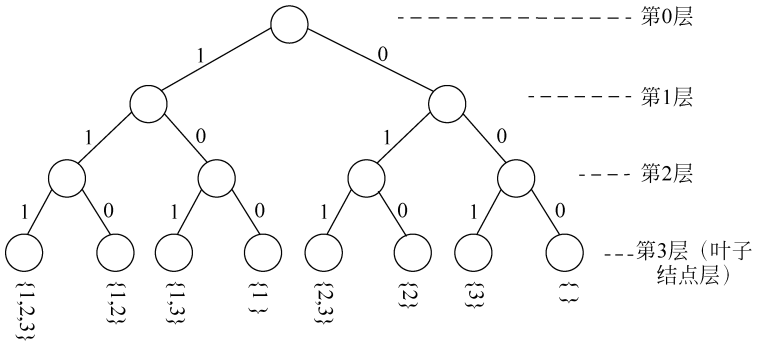


图 5.6　求 $a=\{1,2,3\}$幂集的解空间(1)

```
            print("[",end='')                    #输出一个子集
            for j in range(0,len(x)):
                if x[j]==1:print(a[j],end=' ')
            print("]",end=' ')
        else:
            x[i]=1
            dfs1(a,x,i+1)                        #选择 a[i]
            x[i]=0
            dfs1(a,x,i+1)                        #不选择 a[i]
```

回到本问题，用双层列表类型的 ans 变量存放幂集，将解向量 $\boldsymbol{x}$ 改为直接存放一个子集，在解空间中搜索时，每次到达一个叶子结点得到一个子集 x，将其深复制到 ans 中，否则对于第 i 层的某个结点 A，左分支是选择 nums[i]，将 nums[i]添加到 x 中，当返回到 A 时将 nums[i]从 x 中删除；右分支是不选择 nums[i]。对应的算法如下：

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        self.ans,self.x=[],[]
        self.dfs(nums,0)
        return self.ans

    def dfs(self,nums,i):                        #回溯算法
        if i==len(nums):                         #到达一个叶子结点
            self.ans.append(copy.deepcopy(self.x))
        else:
            self.x.append(nums[i])               #选择 nums[i]，x 中添加 nums[i]
            self.dfs(nums,i+1)
            self.x.pop()                         #回溯，即删除前面添加的 nums[i]
            self.dfs(nums,i+1)                   #不选择 nums[i]，x 中不添加 nums[i]
```

上述程序提交时通过，执行用时为 44ms，内存消耗为 15MB。

3. 问题求解 2

扫一扫

视频讲解

在前面求 a 的幂集的回溯算法中用解向量 $\boldsymbol{x}$ 间接存放一个子集(每个子集对应的 x 的长度相同，分量 x_i 中的下标 i 既表示结点的层次，同时又用于访问 a_i)，现在将 x 改为直接存放 a 的一个子集。设解向量 $\boldsymbol{x}=\{x_0,\cdots,x_i,\cdots,x_{m-1}\}$($m$ 为 x 的长度，$0\leqslant m\leqslant n$)，仍以 $a=\{1,2,3\}$为例，对应的解空间如图 5.7 所示，每个结点的状态为“i，j，x”，其中 i 为结点

的层次，j 表示 x_i 的取值范围为 $a[j..n-1]$，x 为解向量。

根结点的状态为“0,0,{}”，即表示 x_0 的取值范围是 $a[0..2]$，每个取值对应一个分支，共 3 个分支。

① 当 $x_0=a_0=1$ 时($i=0,j=0$)，对应的子结点状态为“$i+1,j+1,x\cup\{1\}$”，即“1,1,{1}”，这里 $i<3$ 同时 $j=1$ 没有超界($j=3$ 时超界)，继续向下扩展。

② 当 $x_0=a_1=2$ 时($i=0,j=1$)，对应的子结点状态为“$i+1,j+1,x\cup\{2\}$”，即“1,2,{2}”，这里 $i<3$ 同时 $j=2$ 没有超界，继续向下扩展。

③ 当 $x_0=a_2=3$ 时($i=0,j=2$)，对应的子结点状态为“$i+1,j+1,x\cup\{3\}$”，即“1,3,{3}”，这里 $i<3$，但 $j=3$ 超界，不再向下扩展。

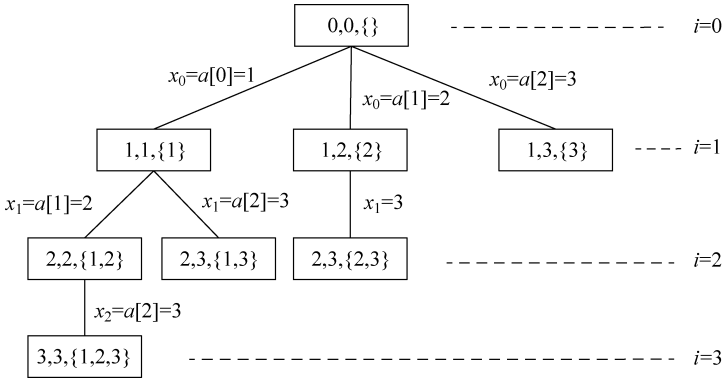


图 5.7　求 $a=\{1,2,3\}$ 幂集的解空间(2)

不同于解法 1，这里解空间中每个结点的 x 都是一个子集，因此解空间中结点的个数相对较少，性能更高。对应的递归回溯算法如下：

```
def subsets2(a):                    #解法 2:求 a 的幂集
    x=[]                            #解向量
    dfs2(a,x,0)

def dfs2(a,x,i):                    #回溯算法
    print(x,end=' ')                #输出一个子集
    for j in range(i,len(a)):
        x.append(a[j])              #向 x 中添加 a[j]
        dfs2(a,x,j+1)
        x.pop()                     #回溯,即删除前面添加的 a[j]
```

回到本问题，同样用双层列表类型的 ans 变量存放幂集，将每次找到的解 x 深复制到 ans 中。对应的算法如下：

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        self.ans,x=[],[]
        self.dfs(nums,x,0)
        return self.ans

    def dfs(self,nums,x,i):                     #回溯算法
        self.ans.append(copy.deepcopy(x))       #找到一个解
```

```
            for j in range(i,len(nums)):
                x.append(nums[j])
                self.dfs(nums,x,j+1)
                x.pop()                    #回溯,即删除前面添加的 nums[j]
```

上述程序提交时通过,执行用时为 40ms,内存消耗为 14.8MB。

5.3.3 实战——子集Ⅱ(LeetCode90★★)

1. 问题描述

给定一个含 n 个整数的数组 nums($1\leqslant n\leqslant 10$,$-10\leqslant$nums[i]$\leqslant 10$),其中可能包含重复元素,请设计一个算法返回该数组的所有可能的子集(幂集)。解集不能包含重复的子集,在返回的解集中子集可以按任意顺序排列。例如,nums={1,2,2},结果为{{},{1},{1,2},{1,2,2},{2},{2,2}}。

2. 问题求解 1

扫一扫

视频讲解

由于这里 nums 中包含重复元素,如果直接采用 5.3.2 节中的解法 1 会得到重复的子集,例如,nums[]={1,2,1}时,求解结果为{{1,2,1},{1,2},{1,1},{1},{2,1},{2},{1},{}},其中{1,2}和{2,1}重复,并且{1}出现两次。两个相同的{1}容易消除,但{1,2}和{2,1}如何消除呢?可以采用先将 nums 排序的方法,例如 nums[]={1,1,2},其结果为{{1,1,2},{1,1},{1,2},{1},{1,2},{1},{2},{}},这样重复的子集的顺序均相同,剩下的问题是消除顺序相同的重复子集。

采用 5.3.2 节中求 a 的幂集的解法 1 的思路,利用集合 set 实现去重(其元素为由 x 转换的元组)。对应的算法如下:

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()                        #nums 递增排序
        self.ans,self.x=set(),[]
        self.dfs(nums,0)
        return list(self.ans)

    def dfs(self,nums,i):                  #回溯算法
        if i==len(nums):                   #到达一个叶子结点
            self.ans.add(tuple(self.x))
        else:
            self.x.append(nums[i])         #选择 nums[i], x 中添加 nums[i]
            self.dfs(nums,i+1)
            self.x.pop()                   #回溯
            self.dfs(nums,i+1)             #不选择 nums[i],x 中不添加 nums[i]
```

上述程序提交时通过,执行用时为 44ms,内存消耗为 16.3MB。

3. 问题求解 2

扫一扫

视频讲解

采用 5.3.2 节中求 a 的幂集的解法 2 的思路,对于解空间中状态为“i,j,x”的结点,表示 x_i 可能的取值范围是 nums[j..$n-1$],如果 $j>i$ 时 nums[j]和 nums[$j-1$]相同,则 x_i 同时取值为 nums[j]和 nums[$j-1$]时会导致出现重复的子集,因此只要跳过这样的重复

情况即可。对应的算法如下：

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()                          #nums 递增排序
        self.ans,x=[],[]
        self.dfs(nums,x,0)
        return self.ans

    def dfs(self,nums,x,i):                  #回溯算法
        self.ans.append(copy.deepcopy(x))
        for j in range(i,len(nums)):
            if j>i and nums[j]==nums[j-1]: continue
            x.append(nums[j])
            self.dfs(nums,x,j+1)
            x.pop()
```

扫一扫

视频讲解

上述程序提交时通过，执行用时为 36ms，内存消耗为 15.2MB。

5.3.4 实战——目标和(LeetCode494★★)

1. 问题描述

给定一个含 n 个整数的数组 nums($1\leqslant n\leqslant 20$，$0\leqslant$nums[i]$\leqslant 1000$)和一个整数 target($-1000\leqslant$target$\leqslant 1000$)，在数组中的每个整数前添加 '+' 或 '-'，然后串联起来所有整数，可以构造一个表达式。例如，nums={2,1}，可以在 2 之前添加 '+'，在 1 之前添加'-'，然后串联起来得到表达式 "+2-1"。设计一个算法求可以通过上述方法构造的运算结果等于 target 的不同表达式的数目。

2. 问题求解

用 ans 表示满足要求的解的个数(初始为 0)，设置解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i$ 表示 nums[i]($0\leqslant i\leqslant n-1$)前面添加的符号，$x_i$ 只能在'+'和'-'符号中二选一，所以该问题的解空间为子集树。用 expv 表示当前运算结果(初始为 0)。对于解空间中第 i 层的结点 A，若 x_i 选择'+'，则 expv+=nums[i]，若 x_i 选择'-'，则 expv-=nums[i]，在回退到 A 时要恢复 expv。当到达一个叶子结点时，如果 expv=target，说明找到一个解，置 ans++。

由于该问题只需要求最后的解个数，所以不必真正设计解向量 $\boldsymbol{x}$，仅设计 expv 即可。对应的算法如下：

```
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        self.ans=0
        self.dfs(nums,target,0,0)
        return self.ans

def dfs(self,nums,target,i,expv):                #回溯算法
    if i==len(nums):                             #到达一个叶子结点
        if expv==target:self.ans+=1              #找到一个解
    else:
        expv+=nums[i]                            #nums[i]前选择'+'
        self.dfs(nums,target,i+1,expv)
```

```
13            expv－＝nums[i]                    ＃回溯:恢复 expv
14            expv－＝nums[i]                    ＃nums[i]前选择'－'
15            self.dfs(nums,target,i＋1,expv)
16            expv＋＝nums[i]                    ＃回溯:恢复 expv
```

说明：上述程序提交时出现超时现象，但同样的思路采用 C/C++或者 Java 编程提交时通过。

扫一扫

视频讲解

5.3.5 子集和问题

1. 问题描述

给定 n 个不同的正整数集合 $a=(a_0,a_1,\cdots,a_{n-1})$和一个正整数 t，要求找出 a 的子集 s，使该子集中所有元素的和为 t。例如，当 $n=4$ 时，$a=(3,1,5,2)$，$t=8$，则满足要求的子集 s 为(3,5)和(1,5,2)。

2. 问题求解

与求幂集问题一样，该问题的解空间是一棵子集树(因为每个整数要么选择，要么不选择)，并且是求满足约束函数的所有解。

1）无剪支

设解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选择 a_i 元素，$x_i=1$ 表示不选择 a_i 元素。在解空间中按深度优先方式搜索所有结点，并用 cs 累计当前结点之前已经选择的所有整数的和，一旦到达叶子结点(即 $i\geqslant n$)，表示 a 的所有元素处理完毕，如果相应的子集和为 t(即约束函数 cs$=t$ 成立)，则根据解向量 $\boldsymbol{x}$ 输出一个解。当解空间搜索完后便得到所有解。

例如 $a=(3,1,5,2)$，$t=8$，其解空间如图 5.8 所示，图中结点上的数字表示 cs，利用深度优先搜索得到两个解，解向量分别是(1,0,1,0)和(0,1,1,1)，对应图中两个带阴影的叶子结点，图中共 31 个结点，每个结点都要搜索。实际上，解空间是一棵高度为 5 的满二叉树，从根结点到每个叶子结点都有一条路径，每条路径是一个决策向量，满足约束函数的决策向量就是一个解向量。

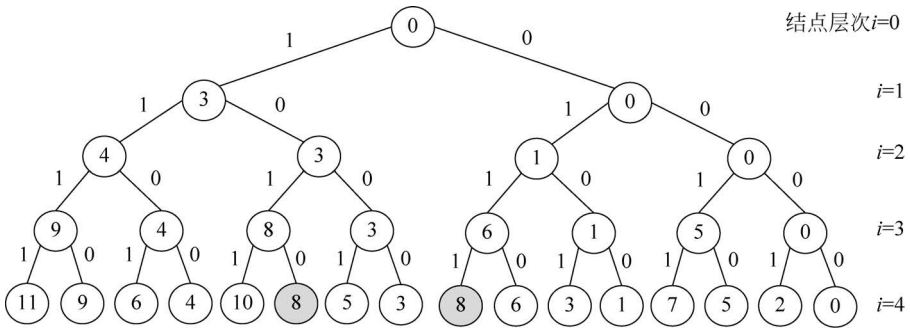


图 5.8 求 $a=(3,1,5,2)$、$t=8$ 时子集和的解空间

对应的递归回溯算法如下：

```
1   cnt＝0                                  ＃累计解的个数
2   sum＝0                                  ＃累计搜索的结点的个数
```

```
def disp(a):                          #输出一个解
    global cnt,x
    cnt+=1;print(" 第%d 个解,"%(cnt),end=")
    print("选取的数为: ",end=")
    for i in range(0,len(x)):
        if x[i]==1:print(a[i],end=")
    print()

def dfs1(a,t,cs,i):                   #回溯算法
    global sum,x
    sum+=1
    if i>=len(a):                     #到达一个叶子结点
        if cs==t:disp(a)              #找到一个满足条件的解,输出
    else:                             #没有到达叶子结点
        x[i]=1                        #选取整数 a[i]
        dfs1(a,t,cs+a[i],i+1)
        x[i]=0                        #不选取整数 a[i]
        dfs1(a,t,cs,i+1)

def subs1(a,t):                       #求解子集和问题
    global x
    x=[0]*len(a)                      #解向量
    print("求解结果")
    dfs1(a,t,0,0)                     #i 从 0 开始
    print("sum=",sum)
```

当 $a=[3,1,5,2]$、$t=8$ 时调用 subs1(a,t)算法的求解结果如下:

```
求解结果
    第 1 个解,选取的数为: 3 5
    第 2 个解,选取的数为: 1 5 2
sum=31
```

【算法分析】 上述算法的解空间是一棵高度为 $n+1$ 的满二叉树,共有 $2^{n+1}-1$ 个结点,递归调用 $2^{n+1}-1$ 次,每找到一个满足条件的解就调用 disp()输出,而执行 disp()的时间为 $O(n)$,所以 subs()算法的最坏时间复杂度为 $O(n\times 2^n)$。

2) 左剪支

由于 a 中的所有元素是正整数,每次选择一个元素时 cs 都会变大,当 cs$>t$ 时沿着该路径继续找下去一定不可能得到解。利用这个特点减少搜索的结点的个数。当搜索到第 $i(0\leqslant i<n)$层的某个结点时,cs 表示当前已经选取的整数的和(其中不包含 $a[i]$),判断选择 $a[i]$是否合适:

① 若 cs+$a[i]>t$,表示选择 $a[i]$后子集和超过 t,不必继续沿着该路径求解,终止该路径的搜索,也就是左剪支。

② 若 cs+$a[i]\leqslant t$,沿着该路径继续下去可能会找到解,不能终止。

简单地说,仅扩展满足 cs+$a[i]\leqslant t$ 的左孩子结点。

例如 $a=(3,1,5,2)$,$t=8$,其搜索空间如图 5.9 所示,图中共 29 个结点,除去两个被剪支的结点(用虚框结点表示),剩下 27 个结点,也就是说递归调用 27 次,性能得到了提高。

对应的递归回溯算法如下:

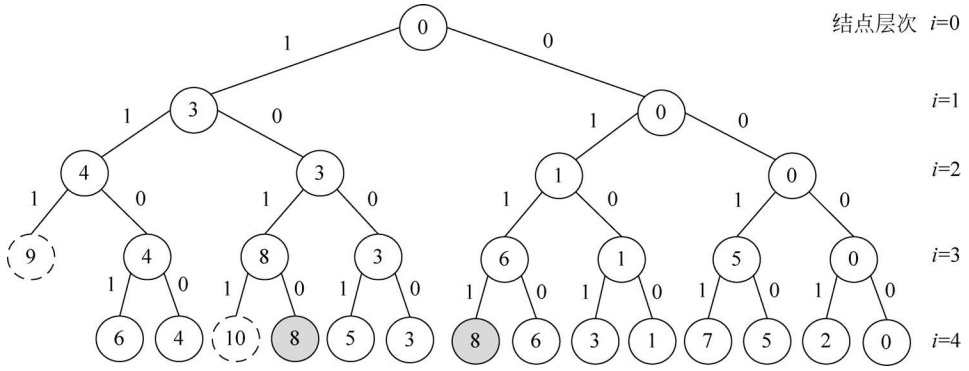


图 5.9 求 $a=(3,1,5,2)$、$t=8$ 时子集和的搜索空间(1)

```
#前面部分与无剪支算法的 1～10 行相同
 1  def dfs2(a,t,cs,i):                    #回溯算法
 2      global sum,x
 3      sum+=1
 4      if i>=len(a):                      #到达一个叶子结点
 5          if cs==t:disp(a)               #找到一个满足条件的解,输出
 6      else:                              #没有到达叶子结点
 7          if cs+a[i]<=t:                 #左孩子结点剪支
 8              x[i]=1                     #选取整数 a[i]
 9              dfs2(a,t,cs+a[i],i+1)
10          x[i]=0                         #不选取整数 a[i]
11          dfs2(a,t,cs,i+1)
12
13  def subs2(a,t):                        #求解子集和问题
14      global x
15      x=[0]*len(a)                       #解向量
16      print("求解结果")
17      dfs2(a,t,0,0)                      #i 从 0 开始
18      rint("sum=",sum)
```

当 $a=[3,1,5,2]$、$t=8$ 时调用 subs2(a,t)算法的求解结果如下：

```
求解结果
    第 1 个解,选取的数为: 3 5
    第 2 个解,选取的数为: 1 5 2
sum= 27
```

3) 右剪支

左剪支仅考虑是否扩展左孩子结点,可以进一步考虑是否扩展右孩子结点。当搜索到第 $i(0\leqslant i<n)$层的某个结点时,用 rs 表示余下的整数的和,即 rs=$a[i]+\cdots+a[n-1]$(其中包含 $a[i]$),因为右孩子结点对应不选择整数 $a[i]$的情况,如果不选择 $a[i]$,此时剩余的所有整数的和为 rs=rs$-a[i]$($a[i+1]+\cdots+a[n-1]$),若 cs+rs$<t$ 成立,说明即使选择所有剩余整数,其和也不可能达到 t,所以右剪支就是仅扩展满足 cs+rs$\geqslant t$ 的右孩子结点,注意在左、右分支处理完后需要恢复 rs,即执行 rs=$+a[i]$。

例如 $a=(3,1,5,2)$,$t=8$,其搜索过程如图 5.10 所示,图中共 17 个结点,除去 7 个被

剪支的结点(用虚框结点表示),剩下 10 个结点,也就是说递归调用 10 次,性能得到更有效的提高。

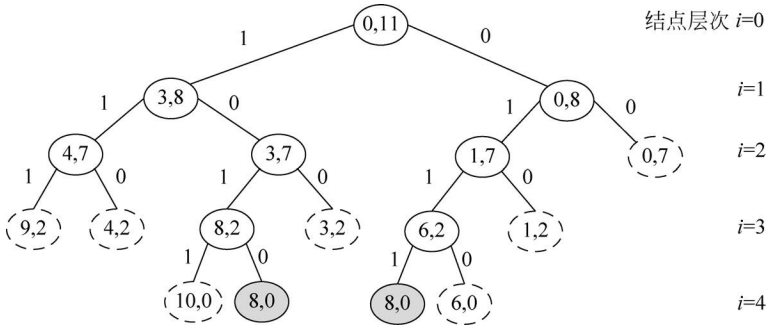


图 5.10 求 $a=(3,1,5,2)$、$t=8$ 时子集和的搜索空间(2)

说明:本例给定 a 中的所有整数为正整数,如果 a 中有负整数,这样的左、右剪支是不成立的,因此无法剪支,算法退化为基本深度优先遍历。

对应的递归回溯算法如下:

```
#前面部分与无剪支算法的 1~10 行相同
 1  def dfs3(a,t,cs,rs,i):                    #回溯算法
 2      global sum,x
 3      sum+=1
 4      if i>=len(a):                         #到达一个叶子结点
 5          if cs==t:disp(a)                  #找到一个满足条件的解,输出
 6      else:                                 #没有到达叶子结点
 7          rs-=a[i]                          #求剩余的整数的和
 8          if cs+a[i]<=t:                    #左孩子结点剪支
 9              x[i]=1                        #选取整数 a[i]
10              dfs3(a,t,cs+a[i],rs,i+1)
11          if cs+rs>=t:                      #右孩子结点剪支
12              x[i]=0                        #不选取整数 a[i]
13              dfs3(a,t,cs,rs,i+1)
14          rs+=a[i]                          #恢复剩余整数和(回溯)
15
16  def subs3(a,t):                           #求解子集和问题
17      global x
18      x=[0]*len(a)                          #解向量
19      rs=0
20      for e in a:rs+=e
21      print("求解结果")
22      dfs3(a,t,0,rs,0)                      #i 从 0 开始
23      print("sum=",sum)
```

当 $a=[3,1,5,2]$、$t=8$ 时调用 subs3(a,t)算法的求解结果如下:

```
求解结果
    第 1 个解,选取的数为: 3 5
    第 2 个解,选取的数为: 1 5 2
sum=10
```

【算法分析】 尽管通过剪支提高了算法的性能,但究竟剪去多少结点与具体的实例数

据相关，所以上述算法在最坏情况下的时间复杂度仍然为 $O(n\times 2^n)$。从上述实例可以看出剪支在回溯法算法中的重要性。

5.3.6 简单装载问题

1. 问题描述

扫一扫

视频讲解

有 n 个集装箱要装上一艘载重量为 t 的轮船，其中集装箱 $i(0\leqslant i\leqslant n-1)$ 的重量为 w_i。不考虑集装箱的体积限制，现要选出重量和小于或等于 t 并且尽可能重的若干集装箱装上轮船。例如，$n=5$，$t=10$，$w=\{5,2,6,4,3\}$ 时，其最佳装载方案有两种，即(1,1,0,0,1)和(0,0,1,1,0)，对应的集装箱重量和达到最大值 t。

2. 问题求解

与求幂集问题一样，该问题的解空间树是一棵子集树(因为每个集装箱要么选择，要么不选择)，但要求求最佳装载方案，属于求最优解类型。设当前解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选择集装箱 i，$x_i=1$ 表示不选择集装箱 i，最优解向量用 bestx 表示，最优重量和用 bestw 表示(初始为 0)，为了简洁，将 bestx 和 bestw 设计为全局变量。

当搜索到第 $i(0\leqslant i<n)$ 层的某个结点时，cw 表示当前选择的集装箱重量和(其中不包含 $w[i]$)，rw 表示余下集装箱的重量和，即 rw=$w[i]+\cdots+w[n-1]$(其中包含 $w[i]$)，此时处理集装箱 i，先从 rw 中减去 $w[i]$，即置 rw−=$w[i]$，采用的剪支函数如下。

① 左剪支：判断选择集装箱 i 是否合适。检查当前集装箱被选中后总重量是否超过 t，若是则剪支，即仅扩展满足 cw+$w[i]\leqslant t$ 的左孩子结点。

② 右剪支：判断不选择集装箱 i 是否合适。如果不选择集装箱 i，此时剩余的所有整数的和为 rw，若 cw+rw≤bestw 成立(bestw 是当前找到的最优解的重量和)，说明即使选择所有剩余集装箱，其重量和也不可能达到 bestw，所以仅扩展满足 cw+rw>bestw 的右孩子结点。

说明：由于深度优先搜索是纵向搜索的，可以较快地找到一个解，以此作为 bestw，再对某个搜索结点(cw,rw)做 cw+rw>bestw 的右剪支通常比广度优先搜索的性能更好。

当第 i 层的这个结点扩展完成后需要恢复 rs，即置 rs+=$a[i]$(回溯)。如果搜索到某个叶子结点(即 $i\geqslant n$)，得到一个可行解，其选择的集装箱的重量和为 cw(由于是左剪支，cw 一定小于或等于 t)，若 cw>bestw，说明找到一个满足条件的更优解，置 bestw=cw，bestx=x。全部搜索完毕，bestx 就是最优解向量。

扫一扫

程序代码

说明：看完整的源程序请扫描右侧二维码。

【算法分析】 在该算法中解空间树有 $2^{n+1}-1$ 个结点，每找到一个更优解时需要将 x 复制到 bestx(执行时间为 $O(n)$)，所以最坏情况下算法的时间复杂度为 $O(n\times 2^n)$。在前面的实例中，$n=5$，解空间树中结点的个数应为 63，采用剪支后结点的个数为 16(不计虚框中被剪支的结点)，如图 5.11 所示。

5.3.7 0/1 背包问题

扫一扫

视频讲解

1. 问题描述

有 n 个编号为 0～$n-1$ 的物品，重量为 $w=\{w_0,w_1,\cdots,w_{n-1}\}$，价值为 $v=\{v_0,v_1,\cdots,$

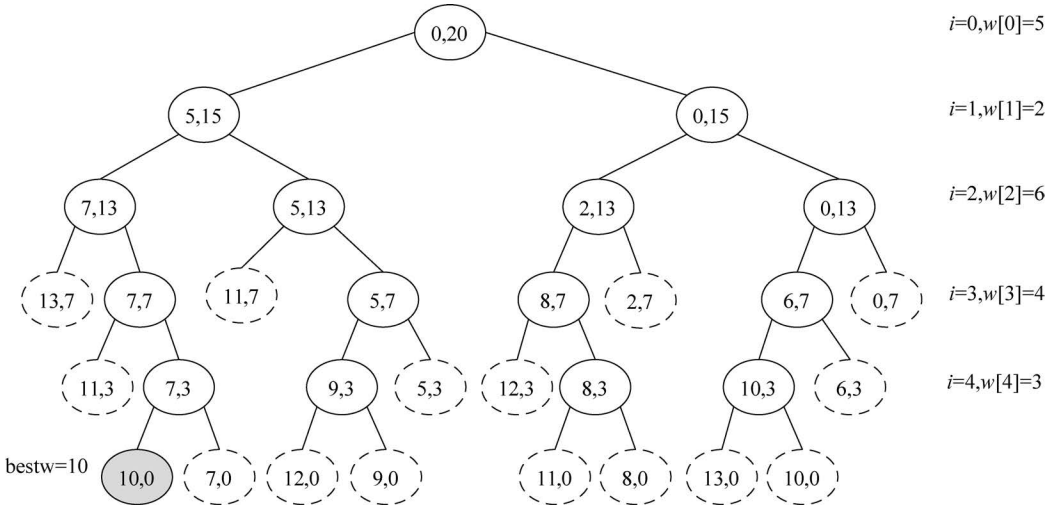


图 5.11　装载实例的搜索空间

$v_{n-1}\}$,给定一个容量为 W 的背包。从这些物品中选取全部或者部分物品装入该背包中,每个物品要么选中,要么不选中,即物品不能被分割,找到选中物品不仅能够放到背包中而且价值最大的方案,并对表 5.1 所示的 4 个物品求出 $W=6$ 时的一个最优解。

表 5.1　4 个物品的信息

物品编号	重　量	价　值
0	5	4
1	3	4
2	2	3
3	1	1

2. 问题求解

该问题的解空间树是一棵子集树(因为每个物品要么选择,要么不选择),要求求价值最大的装入方案,属于求最优解类型。

1) 存储结构设计

每个物品包含编号、重量和价值,为此采用结构体数组存放所有物品,因为后面涉及按单位重量价值递减排序,所以设计物品结构体类型如下:

```
class Goods:                          #物品类
    def __init__(self,x,y,z):
        self.no=x                     #物品的编号
        self.w=y                      #物品的重量
        self.v=z                      #物品的价值
    def __lt__(self,other):           #用于按 v/w 递减排序
        return 1.0*self.v/self.w>=1.0*other.v/other.w
```

例如,表 5.1 中的 4 个物品用向量 g 存放:

```
g=[Goods(0,5,4),Goods(1,3,4),Goods(2,2,3),Goods(3,1,1)]
```

设当前解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选择物品 i，$x_i=1$ 表示不选择物品 i，最优解向量用 bestx 表示，最大价值用 bestv 表示(初始为 0)，为了简洁，将 n、W、bestx 和 bestv 均设计为全局变量。

2) 左剪支

由于所有物品的重量为正数，采用左剪支与子集和问题类似。当搜索到第 $i(0\leqslant i<n)$ 层的某个结点时，cw 表示当前选择的物品的重量和(其中不包含 $w[i]$)。检查当前物品被选中后总重量是否超过 W，若超过则剪支，即仅扩展满足 cw+$w[i]\leqslant W$ 的左孩子结点。

3) 右剪支

这里右剪支相对复杂一些，题目求的是价值最大的装入方案，显然优先选择单位重量价值大的物品，为此将 $\boldsymbol{g}$ 中的所有物品按单位重量价值递减排序，例如表 5.1 中的物品排序后的结果如表 5.2 所示，序号 i 发生了改变，后面改为按 i 而不是按物品编号 no 的顺序依次搜索。

表 5.2　4 个物品按 v/w 递减排序后的结果

序号 i	物品编号 no	重量 w	价值 v	v/w
0	2	2	3	1.5
1	1	3	4	1.3
2	3	1	1	1
3	0	5	4	0.8

先看这样的问题，对于第 i 层的某个结点 A，cw 表示当前选择的物品的重量和(其中不包含 $w[i]$)，cv 表示当前选择的物品的价值和(其中不包含 $v[i]$)，那么继续搜索下去能够得到的最大价值是多少？由于所有物品已按单位重量价值递减排序，显然在背包容量允许的前提下应该依次连续地选择物品 i、物品 $i+1$、……，直到物品 k 装不进背包，假设再将物品 k 的一部分装进背包，直到背包装满，此时一定会得到最大价值。从中看出，从物品 i 开始选择的物品的价值和的最大值为 $r(i)$：

$$r(i)=\sum_{j=i}^{k-1}v_j+(\mathrm{rw}-\sum_{j=i}^{k-1}w_j)(v_k/w_k)$$

也就是说，从结点 A 出发的所有路径中最大价值 bound(cw,cv,i)=cv+$r(i)$，如图 5.12 所示。对应的求上界函数值的算法如下：

```
1  def bound(cw,cv,i):                    # 计算第 i 层结点的上界函数值
2      global g,W,n
3      rw=W-cw                            # 背包的剩余容量
4      b=cv                               # 表示物品价值的上界值
5      j=i
6      while j<n and g[j].w<=rw:
7          rw-=g[j].w                     # 选择物品 j
8          b+=g[j].v                      # 累计价值
9          j+=1
10     if j<n:                            # 最后物品 k=j+1,只能部分装入
11         b+=1.0*g[j].v/g[j].w*rw
12     return b
```

再回过来讨论右剪支，右剪支是判断不选择物品 i 时是否能够找到更优解。如果不选

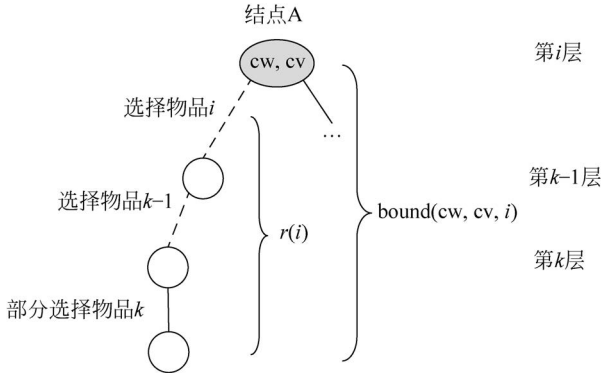


图 5.12　bound(cw,cv,i)

择物品 i,按上述讨论可知在背包容量允许的前提下依次选择物品 $i+1$、物品 $i+2$、……,可以得到最大价值,并且从物品 $i+1$ 开始选择的物品的价值和的最大值为 $r(i+1)$。如果之前已经求出一个最优解 bestv,当 cv+$r(i+1)$≤bestv 时说明若不选择物品 i,后面无论如何也找不到更优解。所以当搜索到第 i 层的某个结点时,对应的右剪支就是仅扩展满足 bound(cw,cv,$i+1$)>bestv 的右孩子结点。

例如,对于根结点,cw=0,cv=0,若不选择物品 0(对应根结点的右孩子结点),剩余背包容量为 rw=W=6,b=cv=0,考虑物品 1,g[1].w<rw,可以装入,$b=b+g$[1].v=4,rw=rw−g[1].w=3;考虑物品 2,g[2].w<rw,可以装入,$b=b+g$[2].v=5,rw=rw−g[2].w=2;考虑物品 3,g[3].w>rw,只能部分装入,$b=b+$rw×(g[3].v/g[3].w)=6.6。

右剪支是求出第 i 层的结点,b=bound(cw,cv,i),若 b≤bestv,则停止右分支的搜索,也就是说仅扩展满足 b>bestv 的右孩子结点。

对于表 5.1 所示的实例,n=4,按 v/w 递减排序后为表 5.2,初始时 bestv=0,求解过程如图 5.13 所示,图中两个数字的结点为(cw,cv),只有右结点标记为(cw,cv,ub),其中虚结点为被剪支的结点,带阴影的结点是最优解结点,其求解结果与回溯法的完全相同,图中结点的数字为(cw,cv),求解步骤如下。

① i=0,根结点为(0,0),cw=0,cv=0,cw+w[0]≤W 成立,扩展左孩子结点,cw=cw+w[0]=2,cv=cv+v[0]=3,对应结点(2,3)。

② i=1,当前结点为(2,3),cw+w[1](5)≤W 成立,扩展左孩子结点,cw=cw+w[1]=5,cv=cv+v[1]=7,对应结点(5,7)。

③ i=2,当前结点为(5,7),cw+w[2](6)≤W 成立,扩展左孩子结点,cw=cw+w[2]=6,cv=cv+v[1]=7,对应结点(6,8)。

④ i=3,当前结点为(6,8),cw+w[2](6)≤W 不成立,不扩展左孩子结点。

⑤ i=3,当前结点为(6,8),不选择物品 3 时计算出 b=cv+0=8,而 b>bestv(0)成立,扩展右孩子结点。

⑥ i=4,当前结点为(6,8),由于 i≥n 成立,它是一个叶子结点,对应一个解 bestv=8。

⑦ 回溯到 i=2 层次,当前结点为(5,7),不选择物品 2 时计算出 b=7.8,b>bestv 不成立,不扩展右孩子结点。

⑧ 回溯到 $i=1$ 层次，当前结点为(2,3)，不选择物品 1 时计算出 $b=6.4$，$b>$bestv 不成立，不扩展右孩子结点。

⑨ 回溯到 $i=0$ 层次，当前结点为(0,0)，不选择物品 0 时计算出 $b=6.6$，$b>$bestv 不成立，不扩展右孩子结点。

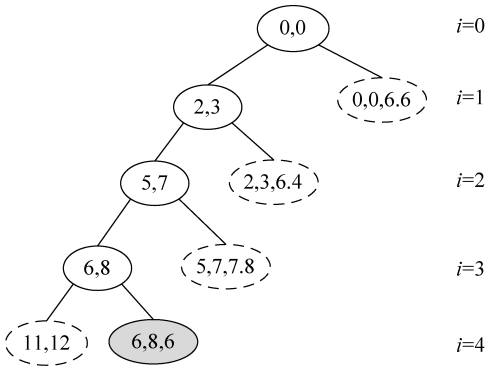


图 5.13　0/1 背包问题实例的搜索空间

解空间搜索完，最优解为 bestv=8，装入方案是选择编号为 2、1、3 的 3 个物品。从中看出，如果不剪支搜索的结点的个数为 31，剪支后搜索的结点的个数为 5。

对应的递归回溯算法如下：

```
def dfs(cw,cv,i):                          #回溯算法
    global g,W,n,x,bestx,bestv,sum
    sum+=1
    if i>=n:                               #到达一个叶子结点
        if cw<=W and cv>bestv:             #找到一个满足条件的更优解,保存它
            bestv=cv
            bestx=copy.deepcopy(x)
    else:                                  #没有到达叶子结点
        if cw+g[i].w<=W:                   #左剪支
            x[i]=1                         #选取物品 i
            dfs(cw+g[i].w,cv+g[i].v,i+1)
        b=bound(cw,cv,i+1)                 #计算限界函数值
        if b>bestv:                        #右剪支
            x[i]=0                         #不选取物品 i
            dfs(cw,cv,i+1)

def knap(g,W):                             #求 0/1 背包问题
    global n,x,bestx,bestv,sum
    n=len(g)                               #物品的个数
    x=[0]*n                                #解向量
    bestx=[0]*n                            #存放最优解向量
    bestv=0                                #存放最大价值,初始为 0
    sum=0                                  #累计搜索的结点的个数
    print("求解结果")
    g.sort()
    dfs(0,0,0)                             #i 从 0 开始
    for i in range(0,n):
```

```
28              if bestx[i]==1:print(" 选取第%d 个物品"%(g[i].no))
29          print(" 总重量=%d,总价值=%d"%(W,bestv))
30          print("sum=",sum)
```

对于表 5.1 中的 4 个物品,$W=6$ 时调用上述 knap()算法的求解结果如下:

```
求解结果
    选取第 2 个物品
    选取第 1 个物品
    选取第 3 个物品
    总重量=6,总价值=8
sum=5
```

【算法分析】 上述算法在不考虑剪支时解空间树中有 $2^{n+1}-1$ 个结点,求上界函数值和保存最优解的时间为 $O(n)$,所以最坏情况下算法的时间复杂度为 $O(n\times 2^n)$。

扫一扫

视频讲解

5.3.8 完全背包问题

1. 问题描述

有 n 种重量和价值分别为 w_i、v_i($0\leqslant i<n$)的物品,从这些物品中挑选总重量不超过 W 的物品,每种物品可以挑选任意多件,求挑选物品的最大价值。该问题称为完全背包问题。

2. 问题求解

与 0/1 背包问题不同,在完全背包问题中物品 i 指的是第 i 种物品,每种物品可以取任意多件。对于解空间中第 i 层的结点,用 cw、cv 表示选择物品的总重量和总价值,这样处理物品 i 的方式如下。

① 不选择物品 i。

② 当 cw+w[i]$\leqslant W$ 时,选择一件物品 i,下一步继续选择物品 i。

③ 当 cw+w[i]$\leqslant W$ 时,选择一件物品 i,下一步开始选择物品 $i+1$。

例如,$n=2$,$W=2$,$w=(1,2)$,$v=(2,5)$,对应的搜索空间如图 5.14 所示,结点对应的状态是"(cw,cv,i)",每个分支结点的 3 个分支分别对应上述 3 种处理方式。所有阴影结点是叶子结点,其中深阴影结点是最优解结点,虚框结点为被剪支的结点。求出该问题的最大价值为 5。

仅求最大价值的回溯算法如下:

```
1   def dfs(cw,cv,i):                          #回溯算法
2       global w,v,n,W,bestv
3       if i>=n:
4           if cw<=W and cv>bestv:             #找到一个更优解
5               bestv=cv
6       else:
7           dfs(cw,cv,i+1)                     #不选择物品 i
8           if cw+w[i]<=W:
9               dfs(cw+w[i],cv+v[i],i)         #剪支:选择物品 i,然后继续选择物品 i
```

```
10            if cw+w[i]<=W:
11                dfs(cw+w[i],cv+v[i],i+1)          #剪支:选择物品 i,然后选择下一件
12
13    def compknap(w,v,n,W):                        #求解完全背包问题
14        global bestv
15        bestv=0                                   #存放最大价值,初始为 0
16        dfs(0,0,0)
17        print("最大价值=",bestv)
```

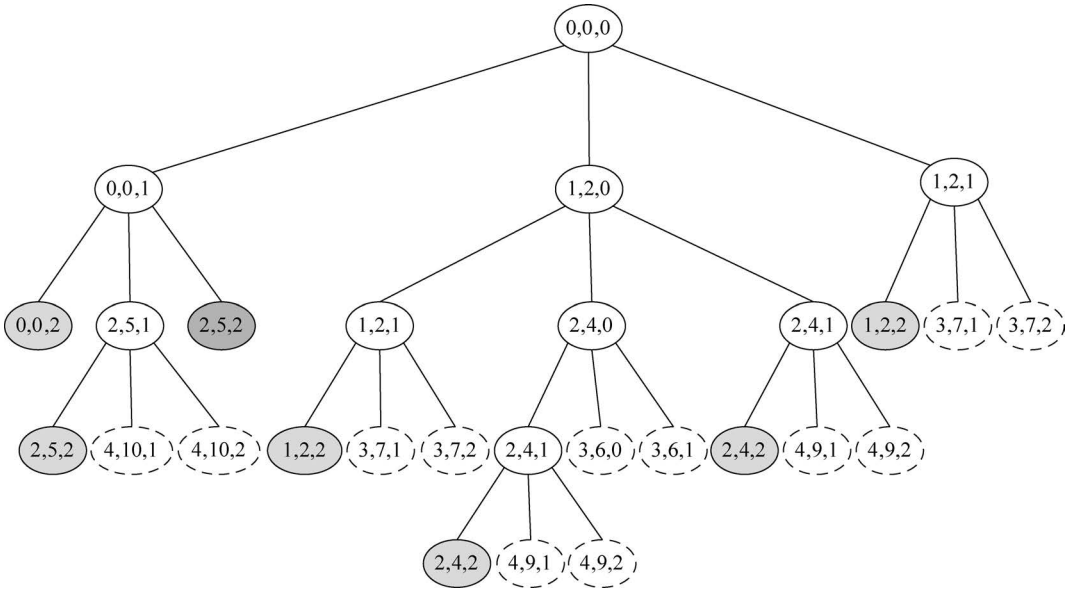


图 5.14　完全背包问题实例的搜索空间

扫一扫

视频讲解

5.3.9　实战——皇后Ⅱ(LeetCode52★★★)

1. 问题描述

在 $n\times n(1\leqslant n\leqslant 9)$的方格棋盘上放置 n 个皇后，每个皇后不同行、不同列、不同对角线(否则称为有冲突)。如图 5.15 所示为 6 皇后问题的一个解。设计一个算法求 n 个皇后的解的个数，例如 $n=6$ 时 6 皇后问题有 4 个解，因此返回结果为 4。

2. 问题求解

本问题的解空间是一棵子集树(每个皇后在 1～n 列中找到一个适合的列号，即 n 选一)，并且要求求所有解。采用整数数组 $q[N]$存放 n 皇后问题的求解结果，因为每行只能放一个皇后，$q[i]$ $(1\leqslant i\leqslant n)$的值表示第 i 个皇后所在的列号，即第 i 个皇后放在$(i,q[i])$的位置上。对于图 5.15 所示的解，$q[1..6]=\{2,4,6,1,3,5\}$(为了简便，不使用 $q[0]$元素)。

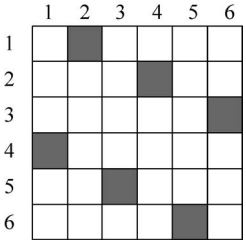


图 5.15　6 皇后问题的一个解

若在(i,j)位置上放第 i 个皇后，是否与已放好的 $i-1$ 个皇后

$(k,q[k])(1\leqslant k\leqslant i-1)$有冲突？显然它们是不同行的(因为皇后的行号 i 总是递增的)，所以不必考虑行冲突，对是否存在列冲突和对角线冲突的判断如下。

① 如果(i,j)位置与前面的某个皇后 $k(1\leqslant k\leqslant i-1)$同列，则有 $q[k]==j$ 成立。

② 如果(i,j)位置与前面的某个皇后同对角线，如图 5.16 所示，则恰好构成一个等腰直角三角形，即有$|q[k]-j|==|i-k|$成立。

归纳起来，只要(i,j)位置满足以下条件，则存在冲突(有冲突时说明第 i 个皇后不能放在第 i 行的第 j 列)，否则不存在冲突：

$$(q[k]==j)\ ||\ (\text{abs}(q[k]-j)==\text{abs}(i-k)) \qquad 1\leqslant k\leqslant i-1$$

现在采用递归回溯框架求解。设 $f(i,n)$是在 $1\sim i-1$ 行上已经放好了 $i-1$ 个皇后，用于在 $i\sim n$ 行放置剩下的 $n-i+1$ 个皇后，为大问题，$f(i+1,n)$表示在 $1\sim i$ 行上已经放好了 i 个皇后，用于在 $i+1\sim n$ 行放置 $n-i$ 个皇后，为小问题，则求解皇后问题的所有解的递归模型如下：

$f(i,n)$≡n 个皇后放置完毕，输出一个解　　若 $i>n$

$f(i,n)$≡在第 i 行找到一个合适的位置(i,j)放置一个皇后；$f(i+1,n)$　　其他

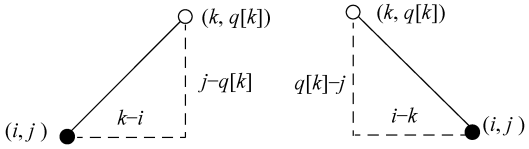


图 5.16　两个皇后构成对角线的情况

对应的算法如下：

```
1   MAXN=20                                  #最多皇后个数
2   q=[0] * MAXN                             #q[i]存放第 i 个皇后的列号
3   class Solution:
4       def totalNQueens(self, n: int) -> int:
5           self.cnt=0                       #累计解的个数
6           self.dfs(1,n)
7           return self.cnt
8
9       def place(self,i,j):                 #测试(i,j)位置能否放置皇后
10          if i==1: return True             #第一个皇后总是可以放置
11          k=1
12          while k<i:                       #k=1~i-1 是已放置了皇后的行
13              if q[k]==j or (abs(q[k]-j)==abs(i-k)):
14                  return False
15              k+=1
16          return True
17
18      def dfs(self,i,n):                   #回溯算法
19          if i>n:                          #所有皇后放置结束
20              self.cnt+=1
21          else:
22              for j in range(1,n+1):       #在第 i 行上试探每个列 j
```

```
23                  if self.place(i,j):          #在第 i 行上找到一个合适位置(i,j)
24                      q[i]=j
25                      self.dfs(i+1,n)
```

上述程序提交时通过，执行用时为 72ms，内存消耗为 15MB。

【算法分析】 在该算法中每个皇后都要试探 n 列，共 n 个皇后，其解空间是一棵子集树，每个结点可能有 n 棵子树，考虑每个皇后试探一个合适位置的时间为 $O(n)$，所以算法的最坏时间复杂度为 $O(n\times n^n)$。

扫一扫

视频讲解

5.3.10 任务分配问题

1. 问题描述

有 $n(n\geqslant 1)$个任务需要分配给 n 个人执行，每个任务只能分配给一个人，每个人只能执行一个任务，第 i 个人执行第 j 个任务的成本是 $c[i][j](0\leqslant i,j\leqslant n-1)$，求出总成本最小的一种分配方案。如表 5.3 所示为 4 个人员、4 个任务的信息。

表 5.3 4 个人员、4 个任务的信息

人员	任务 0	任务 1	任务 2	任务 3
0	9	2	7	8
1	6	4	3	7
2	5	8	1	8
3	7	6	9	4

2. 问题求解

n 个人和 n 个任务的编号均用 0～$n-1$ 表示。所谓一种分配方案，就是由第 i 个人执行第 j 个任务，也就是说每个人从 n 个任务中选择一个任务，即 n 选一，所以本问题的解空间树可以看成一棵子集树，并且要求求总成本最小的解(最优解是最小值)，属于求最优解类型。

设计解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，这里以人为主，即人找任务(也可以以任务为主，即任务找人)，也就是第 i 个人执行第 x_i 个任务$(0\leqslant x_i\leqslant n-1)$。bestx 表示最优解向量，bestc 表示最优解的成本(初始值为∞)，$\boldsymbol{x}$ 表示当前解向量，cost 表示当前解的总成本(初始为 0)，另外设计一个 used 数组，其中 used[j]表示任务 j 是否已经分配(初始时所有元素均为 False)，为了简单，将这些变量均设计为全局变量。

在解空间中根结点的层次 i 为 0，当搜索到第 i 层的每个结点时，表示为第 i 个人分配一个没有分配的任务，即选择满足 used[j]=0$(0\leqslant j\leqslant n-1)$的任务 j。对应的递归回溯算法如下：

```
1  import copy
2  INF=0x3f3f3f3f                          #表示∞
3  def dfs1(cost,i):                       #回溯算法
4      global c,n,x,bestx,bestc,used,sum
5      sum+=1
6      if i>=n:                            #到达一个叶子结点
7          if cost<bestc:                  #通过比较求最优解
```

```
                bestc=cost; bestx=copy.deepcopy(x)
        else:
            for j in range(0,n):                    #为人员 i 试探任务 j:0 到 n-1
                if used[j]:continue                 #跳过已经分配的任务 j
                used[j]=True
                x[i]=j                              #任务 j 分配给人员 i
                cost+=c[i][j]
                dfs1(cost,i+1)                      #为人员 i+1 分配任务
                used[j]=False                       #回溯
                x[i]=-1
                cost-=c[i][j]

def allocate1(c,n):                                 #求解任务分配问题
    global x,bestx,bestc,used,sum
    x=[0]*n
    bestx=[0]*n
    bestc=INF                                       #初始化为∞
    used=[False]*n
    sum=0
    dfs1(0,0)                                       #从人员 0 开始
    print("求解结果")
    for k in range(0,n):
        print("    人员%d 分配任务%d"%(k,bestx[k]))
    print("    总成本=",bestc)
    print("sum=",sum)
```

对于表 5.3,调用上述 allocate1 算法的执行结果如下:

```
求解结果
    人员 0 分配任务 1
    人员 1 分配任务 0
    人员 2 分配任务 2
    人员 3 分配任务 3
    总成本=13
sum=65
```

【算法分析】 算法的解空间是一棵 n 叉树(子集树),所以最坏的时间复杂度为 $O(n\times n^n)$。例如,在上述实例中 $n=4$,经测试搜索的结点的个数为 65。

现在考虑采用剪支提高性能,该问题是求最小值,所以设计下界函数。当搜索到第 i 层的某个结点时,前面人员 0～人员 $i-1$ 已经分配好了各自的任务,已经累计的成本为 cost,现在要为人员 i 分配任务,如果为其分配任务 j,即置 $x[i]=j$,cost+=$c[i][j]$,此时部分解向量为 $\boldsymbol{P}=(x_0,x_1,\cdots,x_i)$。那么沿着该路径走下去的最小成本是多少呢?后面人员 $i+1$～$n-1$ 尚未分配任务,如果为每个人员分配一个尚未分配的最小成本的任务(其总成本为 minsum),则一定能构成该路径的总成本下界。minsum 的计算公式如下:

$$\text{minsum}=\sum_{i1=i+1}^{n-1}\min_{j1\notin P}\{c_{i1,j1}\}$$

总成本下界 $b=$cost+minsum,如图 5.17 所示。显

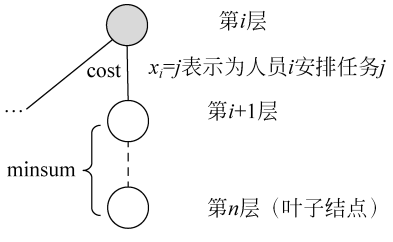


图 5.17 为人员 i 安排任务 j 的情况

然如果 $b \geqslant$bestc(bestc 是当前已经求出的一个最优成本),说明 $x[i]=j$ 这条路径走下去一定不能找到更优解,所以停止该分支的搜索。这里的剪支就是仅扩展 $b<$bestc 的孩子结点。

求一个结点的总成本下界的算法如下:

```
def bound(cost,i):                          #求下界算法
    global c,n,used
    minsum=0
    for i1 in range(i,n):                   #求 c[i..n-1]行中未分配的最小成本和
        minc=INF                            #置为∞
        for j1 in range(0,n):
            if not used[j1] and c[i1][j1]<minc:minc=c[i1][j1]
        minsum+=minc
    return cost+minsum
```

带剪支的递归回溯算法如下:

```
def dfs2(cost,i):                           #回溯算法
    global c,n,x,bestx,bestc,used,sum
    sum+=1
    if i>=n:                                #到达一个叶子结点
        if cost<bestc:                      #通过比较求最优解
            bestc=cost;bestx=copy.deepcopy(x)
        else:
            for j in range(0,n):            #为人员 i 试探任务 j:0 到 n-1
                if used[j]:continue         #跳过已经分配的任务 j
                used[j]=True
                x[i]=j                      #将任务 j 分配给人员 i
                cost+=c[i][j]
                if bound(cost,i+1)<bestc:   #剪支(考虑 c[i+1..n-1]行中的最小成本)
                    dfs2(cost,i+1)          #为人员 i+1 分配任务
                used[j]=False               #回退
                x[i]=-1
                cost-=c[i][j]

def allocate2(c,n):                         #求解任务分配问题
    global x,bestx,bestc,used,sum
    x=[0]*n
    bestx=[0]*n
    bestc=INF                               #初始化为∞
    used=[False]*n
    sum=0
    dfs2(0,0)                               #从人员 0 开始
    print("求解结果")
    for k in range(0,n):
        print("    人员%d 分配任务%d"%(k,bestx[k]))
    print("    总成本=",bestc)
    print("sum=",sum)
```

对于表 5.3,调用上述 allocate2 算法的执行结果如下:

```
求解结果
    人员 0 分配任务 1
```

```
        人员 1 分配任务 0
        人员 2 分配任务 2
        人员 3 分配任务 3
        总成本=13
sum=9
```

从中看出,采用剪支后搜索的结点的个数为 9,算法的时间性能得到明显提高。

【算法分析】 算法的解空间是一棵 n 叉树(子集树),剪支的时间为 $O(n^2)$,所以最坏的时间复杂度为 $O(n^2\times n^n)$。

扫一扫

视频讲解

5.3.11* 实战——完成所有工作的最短时间(LeetCode1723★★★)

1. 问题描述

给定一个整数数组 jobs,其中 jobs[i]是完成第 i 项工作要花费的时间($1\leqslant$jobs.length$\leqslant 12$,$1\leqslant$jobs[i]$\leqslant 10^7$)。将这些工作分配给 k($1\leqslant k\leqslant$jobs.length)位工人。所有工作都应该分配给工人,且每项工作只能分配给一位工人,每个工人至少完成一项工作。工人的工作时间是完成分配给他们的所有工作所花费时间的总和。设计一套最佳的工作分配方案,使工人的最大工作时间得以最小化,返回分配方案中尽可能最小的最大工作时间。例如,jobs=$\{1,2,4,7,8\}$,$k=2$,结果为 11,对应的一种分配方案是 1 号工人分配时间为 1、2、8 的任务(工作时间为 1+2+8=11),2 号工人分配时间为 4、7 的任务(工作时间为 4+7=11),最大工作时间是 11。

2. 问题求解 1

用 times[0..$k-1$]表示所有工人分配工作的总时间(初始时所有元素均为 0),其中 times[j]表示工人 j 的总时间,用 ans 存放最优解(初始为∞),按工作序号 i 从 0 到 $n-1$ 遍历,解空间中根结点对应的 $i=0$,ct 表示当前的总时间,采用基于 k 选一的子集树框架求解。显然到达一个叶子结点后,ct=$\max_{0\leqslant j\leqslant k-1}\{\text{times}[j]\}$,ans=$\min\{\text{ct}\}$。第 i 层的结点用于为工作 i 寻找工人 j,ct 即为完成 0～$i-1$ 共 i 个任务的时间。例如,jobs=$\{1,2,4\}$,$k=2$ 的搜索空间如图 5.18 所示,图中结点为(times[0],times[1]),对应的最优解 ans=4。

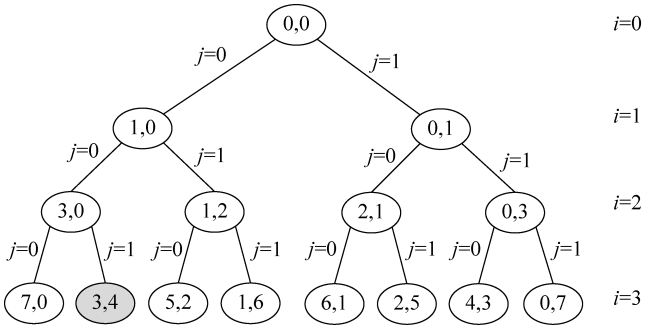


图 5.18 搜索空间

从中看出,解空间是一棵高度为 $n+1$ 的满 k 叉树,这样搜索会超时,可以采用如下剪支方法提高性能。

剪支 1：从图 5.18 看出，$k=2$ 时左、右子树是对称的，当 $k>2$ 时存在更多的重复子树，同时题目中规定每个工人至少分配一个工作，所以当给某个工人 j 分配任务 i 时，若他是初次分配(times[j]=0)，并且前面已有工人没有分配任务，则不必继续搜索下去，这样就剪去了(0,1)的分支。

剪支 2：采用常规的限界函数剪支，若已经求出一个解 ans，如果将任务 i 分配给工人 j，完成 0～i 任务的时间和 curtime=max(ct,times[j])，若 curtime>ans，则不必继续搜索下去。

由于采用了剪支 2，ct 会越来越小，那么满足 ct≤ans 的最后一个 ct 就是 ans。前面的示例采用剪支后的搜索空间如图 5.19 所示，从中看出几乎剪去了一半的结点。

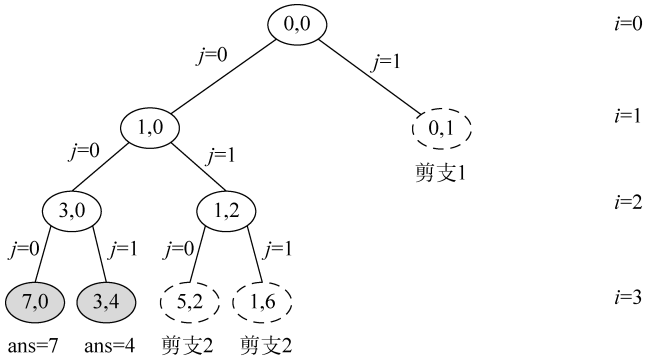


图 5.19　剪支后的搜索空间(1)

对应的算法如下：

```
1  class Solution:
2      def minimumTimeRequired(self, _jobs:List[int], _k:int)->int:
3          self.ans=0x3f3f3f3f                          #存放最优解,初始为∞
4          self.times=[0]*_k
5          self.jobs=_jobs
6          self.k=_k
7          self.dfs(0,0)
8          return self.ans
9
10     def dfs(self,ct,i):                              #回溯算法
11         if i==len(self.jobs):                        #到达一个叶子结点
12             self.ans=ct                              #求得一个解
13         else:
14             flag=True
15             for j in range(0,self.k):
16                 if self.times[j]==0:
17                     if not flag:return               #剪支 1
18                     flag=False
19                 self.times[j]+=self.jobs[i]          #将工作 i 分配给工人 j
20                 curtime=max(ct,self.times[j])
21                 if curtime<=self.ans:                #剪支 2
22                     self.dfs(curtime,i+1)
23                 self.times[j]-=self.jobs[i]          #回溯
```

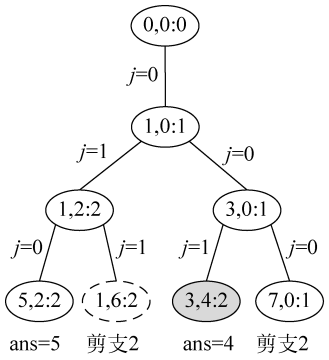


图 5.20 剪支后的搜索空间(2)

上述程序提交时通过,执行用时为 6536ms,内存消耗为 14.9MB。

3. 问题求解 2

当然也可以采用这样的分配方式,即优先分配给空闲的工人,之后再给已经分配工作的工人分配工作(保证每个工人至少分配一个工作)。用 cnt 累计已经分配工作的人数。对于第 i 层的结点,当 cnt$<k$ 时,说明工人 cnt 一定是空闲工人,优先给他分配工作 i 并回溯,然后试探为 0 到 cnt-1 的工人分配工作 i 再回溯。前面的示例采用该方法的搜索空间如图 5.20 所示,图中结点为(times[0],times[1]: cnt),从中看出工作 0 只会分配给工人 0,从而剪去一半的结点。

对应的算法如下:

```
class Solution:
    def minimumTimeRequired(self,_jobs:List[int],_k:int)->int:
        self.ans=0x3f3f3f3f                          #存放最优解,初始为∞
        self.times=[0]*_k
        self.jobs=_jobs
        self.k=_k
        self.dfs(0,0,0)
        return self.ans

    def dfs(self,cnt,ct,i):                          #回溯算法
        if i==len(self.jobs):                        #到达一个叶子结点
            self.ans=ct                              #求得一个解
        else:
            if cnt<self.k:                           #剪支 1:优先分配给空闲工人
                self.times[cnt]=self.jobs[i]
                self.dfs(cnt+1,max(self.times[cnt],ct),i+1)
                self.times[cnt]=0                    #回溯
            for j in range(0,cnt):                   #给已有工作的工人分配工作
                self.times[j]+=self.jobs[i]
                curtime=max(ct,self.times[j])
                if curtime<=self.ans:                #剪支 2
                    self.dfs(cnt,curtime,i+1)        #cnt 不变
                self.times[j]-=self.jobs[i]          #回溯
```

上述程序提交时通过,执行用时为 1ms,内存消耗为 35.8MB。

说明:本题等同于将 n 个正整数分为 k 份,使得每份的整数和最接近,求其中最大的整数和。

扫一扫

视频讲解

5.3.12 图的 m 着色

1. 问题描述

给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色,每个顶点着一种颜色。如果有一种着色法使 G 中每条边的两个顶点着不同颜色,则称这个图是 m 可

着色的。图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色法。

2. 问题求解

对于含 n 个顶点的无向连通图 G，顶点的编号是 0～$n-1$，采用邻接表 A 存储，其中 $A[i]$向量为顶点 i 的所有相邻顶点。例如，如图 5.21 所示的无向连通图，对应的邻接表如下：

$A=[[1,2,3],[0],[0,3],[0,2]]$

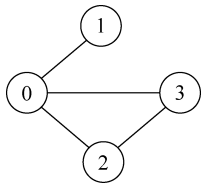


图 5.21　一个无向连通图

m 种颜色的编号为 0～$m-1$，这里实际上就是为每个顶点 i 选择 m 种颜色中的一种（m 选一），使得任意两个相邻顶点的着色不同，所以将解空间树看成一棵子集树，并且求解个数，属于求所有解类型。

设计解向量为 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，其中 x_i 表示顶点 i 的着色（$0\leqslant x_i\leqslant m-1$），初始时置 x 的所有元素为 -1，表示所有顶点均没有着色，用 ans 累计解个数（初始为 0）。采用递归回溯方法从顶点 0 开始试探（$i=0$ 对应根结点），当 $i\geqslant n$ 时表示找到一种着色方案（对应解空间中的一个叶子结点）。

对于顶点 i，所有可能的着色 j 为 0 到 $m-1$ 中的一种，如果顶点 i 的所有相邻顶点的颜色均不等于 j，说明顶点 i 着色 j 是合适的，只要有一个相邻顶点的颜色等于 j，则顶点 i 着色 j 就是不合适的，需要回溯。对应的算法如下：

```
def judge(i,j):                          #判断顶点 i 是否可以着色 j
    global A,n,m,x,ans
    for k in range(0,len(A[i])):
        if x[A[i][k]]==j:return False    #存在相同颜色的顶点
    return True

def dfs(i):                              #回溯算法
    global m,x,ans
    if i>=n:ans+=1                       #到达一个叶子结点
    else:
        for j in range(0,m):
        x[i]=j                           #置顶点 i 为颜色 j
        if judge(i,j):dfs(i+1)           #若顶点 i 可以着色 j
        x[i]=-1                          #回溯

def colors(A,n,m):                       #求图的 m 着色问题
    global x,ans
    x=[-1]*n                             #解向量元素初始化为-1
    ans=0                                #着色方案数
    dfs(0)                               #从顶点 0 开始搜索
    return ans
```

对于图 5.21，$m=3$ 时调用上述 colors()算法求出有 12 种不同的着色方案。

【算法分析】 解空间中最多生成 $O(m^n)$个结点，每个结点花费 $O(n)$的时间判断当前颜色是否合适，所以算法的最坏时间复杂度为 $O(n\times m^n)$。

5.4 基于排列树框架的问题求解

5.4.1 排列树算法框架概述

当求解问题是确定 n 个元素满足某种条件的排列时,相应的解空间树称为**排列树**。解空间为排列树的递归框架是以求全排列为基础的,下面通过示例讨论一种不同于第 3 章中求全排列的递归算法。

扫一扫

视频讲解

【例 5-2】 (LeetCode46★★)有一个含 n 个整数的数组 a,所有元素均不相同,求其所有元素的全排列。例如,$a=\{1,2,3\}$,得到的结果是(1,2,3),(1,3,2),(2,3,1),(2,1,3),(3,1,2),(3,2,1)。

解 用数组 a 存放初始数组 a 的一个排列,采用递归法求解。设 $f(a,n,i)$表示求 $a[i..n-1]$(共 $n-i$ 个元素)的全排列,为大问题,$f(a,n,i+1)$表示求 $a[i+1..n-1]$(共 $n-i-1$ 个元素)的全排列,为小问题,如图 5.22 所示。

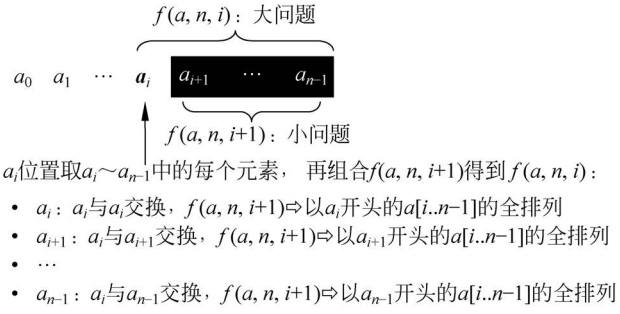


图 5.22 求 $f(a,n,i)$的过程

显然 i 越小求全排列的元素个数越多,当 $i=0$ 时求 $a[0..n-1]$的全排列。当 $i=n-1$ 时求 $a[n-1..n-1]$的全排列,此时序列中只有一个元素(单个元素的全排列就是该元素),再合并 $a[0..n-2]$($n-1$ 个元素的排列)就得到 n 个元素的一个排列。当 $i=n$ 时求 $a[n..n-1]$的全排列,此时序列为空,说明 $a[0..n-1]$是一个排列,后面两种情况均可以作为递归出口。所以求 a 中全排列的过程是 $f(a,n,0)\rightarrow f(a,n,1)\rightarrow f(a,n,2)\rightarrow\cdots\rightarrow f(a,n,n-1)$。

那么如何由小问题 $f(a,n,i+1)$求大问题 $f(a,n,i)$呢? 假设由 $f(a,n,i+1)$求出了 $a[i+1..n-1]$的全排列,考虑 a_i 的位置,该位置可以取 $a[i..n-1]$中的任何一个元素,但是排列中元素不能重复,为此采用交换方式,即 $j=i$ 到 $n-1$ 循环,每次循环将 $a[i]$与 $a[j]$交换,合并子问题的解得到一个大问题的排列,再恢复成循环之前的顺序,即将 $a[i]$与 $a[j]$再次交换,然后进入下一轮求其他大问题的排列。注意,如果不做再次交换(即恢复)会出现重复的排列情况,例如 $a=\{1,2,3\}$,在不做恢复时的输出结果为(1,2,3),(1,3,2),(3,1,2),(3,2,1),(1,2,3),(1,3,2),显然是错误的。

归纳起来,求 a 的全排列的递归模型 $f(a,n,i)$如下:

$f(a,n,i)$≡输出产生的解　　当 $i=n-1$ 时

$f(a,n,i)$≡对于 $j=i\sim n-1$：　　其他
　　$a[i]$与 $a[j]$交换位置；
　　$f(a,n,i+1)$；
　　将 $a[i]$与 $a[j]$交换位置(回溯)

例如 $a=\{1,2,3\}$时，求全排列的解空间如图 5.23 所示，数组 a 的下标从 0 开始，所以根结点“$a=\{1,2,3\}$”的层次为 0，同时 a 数组也作为解向量，由根结点扩展出 3 个子结点，分别对应 $a[0]$位置选择 $a[0]$、$a[1]$和 $a[2]$元素，采用交换方式实现，当从子结点返回时需要恢复，采用再次交换的方式实现。实际上，对于第 i 层的结点，其子树分别对应 $a[i]$位置选择 $a[i]$、$a[i+1]$、……、$a[n-1]$元素。树的高度为 $n+1$，叶子结点的层次是 n。

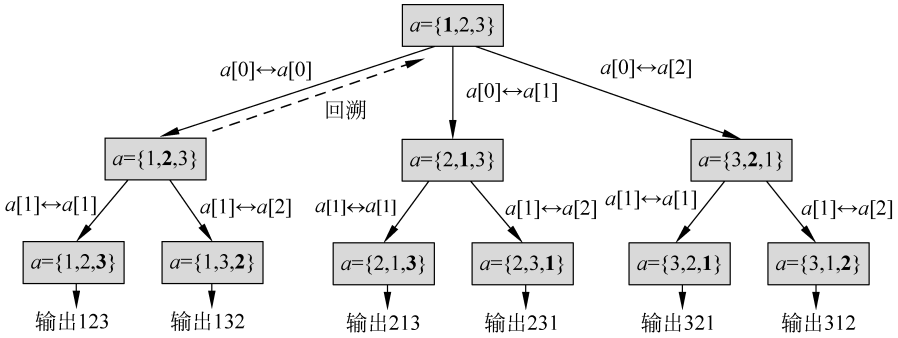


图 5.23　求 $a=\{1,2,3\}$的全排列的解空间

对应的递归算法如下：

```
def dfs(x,i):                          #回溯算法
    if i==len(x):
        print(x)
    else:
        for j in range(i,len(x)):
            x[i],x[j]=x[j],x[i]        #x[i]与 x[j]交换
            dfs(x,i+1)
            x[i],x[j]=x[j],x[i]        #回溯

def perm(a):                           #求 a 的全排列
    x=a                                #解向量
    dfs(x,0)
```

LeetCode46 用于求数组 nums 的全排列，并且返回该全排列，按照上述过程设计求解程序如下：

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        self.ans=[]                            #存放 nums 的全排列
        x=nums
        self.dfs(x,0)
        return self.ans

    def dfs(self,x,i):                         #回溯算法
        if i==len(x):
```

```
10                  self.ans.append(copy.deepcopy(x))
11              else:
12                  for j in range(i,len(x)):
13                      x[i],x[j]=x[j],x[i]
14                      self.dfs(x,i+1)
15                      x[i],x[j]=x[j],x[i]
```

上述程序提交时通过,执行用时为 32ms,内存消耗为 15.2MB。

现在证明上述算法的正确性。实际上在递归算法中求值顺序与递推顺序相反,求 a 的全排列是从 $f(a,n,0)$开始的,求值顺序是 $f(a,n,n-1)\rightarrow f(a,n,n-2)\rightarrow\cdots\rightarrow f(a,n,1)\rightarrow f(a,n,0)$。循环不变量是 $f(a,n,i)$,用于求 $a[i..n-1]$的全排列,证明如下。

初始化:在循环的第一轮迭代开始之前,$i=n-1$,表示求 $a[n-1..n-1]$的全排列,而一个元素就是该元素,显然是正确的。

保持:若前面 $f(a,n,i+1)$正确,表示求出了 $a[i+1..n-1]$的全排列,将 $a[i]$与 $a[i..n-1]$中的每个元素交换,合并 $a[i+1..n-1]$的一个排列得到 $f(a,n,i)$的一个排列,在恢复后继续做完,从而得到 $f(a,n,i)$的全排列。

终止:当求值结束时 $i=0$,得到 $f(a,n,0)$即 a 的全排列。

从上述求 a 的全排列的示例可以归纳出解空间为排列树的递归回溯框架如下:

```
x=[]                                    #x存放解向量,需要初始化
def dfs(i):                             #求解排列树的递归框架
    if i>=n:                            #到达一个叶子结点,输出一个可行解
        输出结果
    else:
        for j in range(i,n):            #用j枚举x[i]的所有可能候选值
            ...                         #第i层的结点选择x[j]的操作
            swap(x[i],x[j])             #为保证排列中的每个元素不同,通过交换来实现
            if constraint(i,j) and bound(i,j):
                dfs(i+1)                #满足约束条件和限界函数,进入下一层
            swap(x[i],x[j])             #恢复状态:回溯
            ...                         #第i层的结点选择x[j]的恢复操作
        }
    }
}
```

如何进一步理解上述算法呢?假设解向量为$(x_0,x_1,\cdots,x_i,\cdots,x_j,\cdots,x_{n-1})$,当从解空间的根结点出发搜索到达第 i 层的某个结点时,对应的部分解向量为$(x_0,x_1,\cdots,x_{i-1})$,其中每个分量已经取好值了,现在为该结点的分支选择一个 x_i 值(每个不同的取值对应一个分支,x_i 有 $n-i$ 个分支),前一个 swap($x[i]$,$x[j]$)表示为 x_i 取 x_j 值,后一个 swap($x[i]$,$x[j]$)用于状态恢复,这一点是利用排列树的递归回溯框架求解实际问题的关键。另外几点需要注意的说明事项与解空间为子集树的递归回溯框架相同。

在排列树中有 $m_0=n$,$m_1=n-1$,$\cdots$,$m_{n-1}=1$,假设输出一个排列的时间为 $O(n)$,对应算法的时间复杂度为 $O(n\times n!)$。

扫一扫

视频讲解

5.4.2 实战——含重复元素的全排列Ⅱ(LeetCode47★★)

1. 问题描述

给定一个可包含重复数字的序列 nums,设计一个算法按任意顺序返回所有不重复的

全排列。例如，nums={1,1,2}，输出结果是{{1,1,2},{1,2,1},{2,1,1}}。

2. 问题求解

该问题与求非重复元素的全排列的问题类似，解空间是排列树，并且属于求所有解类型。先按求非重复元素的全排列的一般过程来求含重复元素的全排列，假设 $a=\{1,\boxed{1},2\}$，其中包含两个 1，为了区分，后面一个 1 加上一个框，求其全排列的过程如图 5.24 所示，从中看出，1↔1 的分支和 1↔$\boxed{1}$ 的分支产生的所有排列是相同的，属于重复的排列，应该剪去后者，再看第 1 层的“{2,$\boxed{1}$,1}”结点，同样它扩展的两个分支分别是$\boxed{1}$↔$\boxed{1}$和$\boxed{1}$↔1，也是相同的，也应该剪去后者。这样剪去后得到的结果是{1,1,2},{1,2,1},{2,1,1}，也就是不重复的全排列。

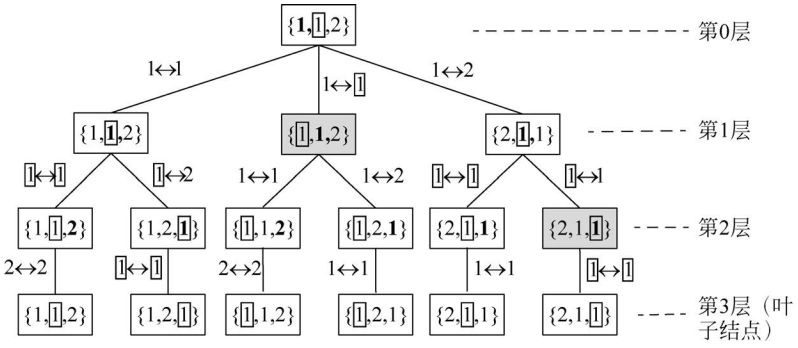


图 5.24　求 $a=\{1,\boxed{1},2\}$的全排列的过程

同样设解向量为 $\boldsymbol{x}=(x_0,x_1,\cdots,x_n)$，每个 x 表示一个排列，x_i 表示该排列中 i 位置所取的元素，初始时 x=nums。在解空间中搜索到第 i 层的某个结点 C 时，如图 5.25 所示，C 结点的每个分支对应 x_i 的一个取值，从理论上讲 x_i 可以取 $x_i\sim x_{n-1}$ 中的每一个值，也就是说从根结点经过结点 C 到达第 $i+1$ 层的结点有 $n-1-i+1=n-i$ 条路径，这些路径中从根结点到 C 结点都是相同的。当 x_i 取值 x_j 时（对应图中粗分支）走到 B 结点，如果 x_j 与前面 $x_i\sim x_{j-1}$ 中的某个值 x_k 相同，当 x_i 取值 x_k 时走到 A 结点，显然根结点到 A 和 B 结点的路径完全相同，而且它们的层次相同，后面的操作也相同，则所有到达叶子结点产生的解必然相同，属于重复的排列，需要剪去。

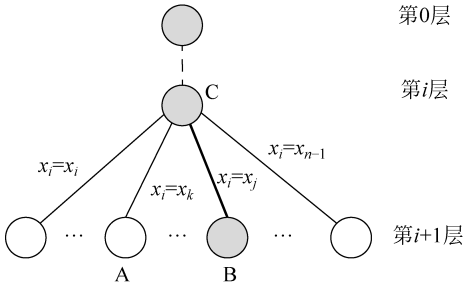


图 5.25　x_i 的各种取值

剪去重复的解的方法是，当 j 从 i 到 $n-1$ 循环时，每次循环执行 swap($x[i]$,$x[j]$)为 i 位置选取元素 $x[j]$，如果 $x[j]$与 $x[i..j-1]$中的某个元素相同会出现重复的排列，则跳过（称为同层去重），也就是说在执行 swap($x[i]$,$x[j]$)之前先判断 $x[j]$是否在前面的元素 $x[i..j-1]$中出现过，如果没有出现过，则继续做下去，否则跳过 $x[j]$的操作。对应的算法如下：

```
1  class Solution:
2      def permuteUnique(self, nums: List[int]) -> List[List[int]]:
```

```
        self.ans=[]                                  #存放 nums 的全排列
        x=nums
        self.dfs(x,0)
        return self.ans

    def dfs(self,x,i):                               #回溯算法
        if i==len(x):
            self.ans.append(copy.deepcopy(x))
        else:
            for j in range(i,len(x)):
                if self.judge(x,i,j):continue        #检测 x[j]
                x[i],x[j]=x[j],x[i]
                self.dfs(x,i+1)
                x[i],x[j]=x[j],x[i]

    def judge(self,x,i,j):                           #判断 x[j]是否在 x[i..j-1]中出现过
        if j>i:
            for k in range(i,j):                     #判断 x[j]是否与 x[i..j-1]中的元素相同
                if x[k]==x[j]:return True            #若相同则返回真
        return False                                 #若全部不相同返回假
```

扫一扫

视频讲解

上述程序提交时通过,执行用时为 52ms,内存消耗为 15.2MB。

5.4.3 任务分配问题

1. 问题描述

见 5.3.10 节,这里采用基于排列树框架求解。

2. 问题求解

n 个人和 n 个任务的编号均为 0～$n-1$,设计解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$,同样以人为主,也就是第 i 个人执行第 x_i 个任务($0\leqslant x_i\leqslant n-1$),显然每个合适的分配方案 x 一定是 0～$n-1$ 的一个排列,可以求出 0～$n-1$ 的全排列,每个排列作为一个分配方案,求出其成本,通过比较找到一个最小成本 bestc 即可。

用 bestx 表示最优解向量,bestc 表示最优解的成本,$\boldsymbol{x}$ 表示当前解向量,cost 表示当前解的总成本(初始为 0),另外设计一个 used 数组,其中 used[j]表示任务 j 是否已经分配(初始时所有元素均为 False),为了简单,将这些变量均设计为全局变量。

根据排列树的递归算法框架,当搜索到第 i 层的某个结点时,第一个 swap(x[i],x[j])表示为人员 i 分配任务 x[j](注意不是任务 j),成本是 c[i][x[i]](因为 x[i]就是交换前的 x[j]),所以执行 used[x[i]]=True,cost+=c[i][x[i]],调用 dfs(x,cost,i+1)继续为人员 i+1 分配任务,回溯操作是 cost-=c[i][x[i]],used[x[i]]=False 和 swap(x[i],x[j])(正好与调用 dfs(x,cost,i+1)之前的语句顺序相反)。

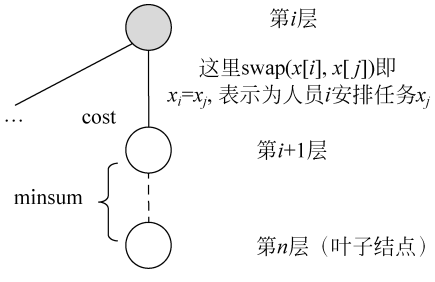


图 5.26 为人员 i 安排任务 x[j]的情况

考虑采用剪支提高性能,设计下界函数,与 5.3.10 节中的 bound 算法相同,仅需要将 j(指任务编号)改为 x[j]即可,如图 5.26 所示,这样仅扩展 bound(x,cost,i+1)<bestc 的孩子结点。

带剪支的排列树的递归回溯算法如下：

```
import copy
INF=0x3f3f3f3f                          #表示∞
def dfs3(cost,i):                       #回溯算法
    global c,n,x,bestx,bestc,used,sum
    sum+=1
    if i>=n:                            #到达一个叶子结点
        if cost<bestc:                  #通过比较求最优解
            bestc=cost
            bestx=copy.deepcopy(x)
    else:
        for j in range(i,n):            #为人员 i 试探任务 x[j]
            if used[x[j]]:continue      #跳过已经分配的任务 j
            x[i],x[j]=x[j],x[i]         #swap(x[i],x[j]):为人员 i 分配任务 x[j]
            used[x[i]]=True
            cost+=c[i][x[i]]
            if bound(cost,i+1)<bestc:   #剪支
                dfs3(cost,i+1)          #继续为人员 i+1 分配任务
            cost-=c[i][x[i]]            #cost 回溯
            used[x[i]]=False            #used 回溯
            x[i],x[j]=x[j],x[i]         #x 回溯

def bound(cost,i):                      #求下界算法
    global c,n,x,used
    minsum=0
    for i1 in range(i,n):               #求 c[i..n-1]行中的最小元素和
        minc=INF
        for j1 in range(0,n):
            if not used[x[j1]] and c[i1][x[j1]]<minc:minc=c[i1][x[j1]]
        minsum+=minc
    return cost+minsum

def allocate3(c,n):                     #求解任务分配问题
    global x,bestx,bestc,used,sum
    x=[]
    for i in range(0,n):x.append(i)     #初始化解向量 x
    bestx=[0]*n                         #最优解向量
    bestc=INF                           #将最优成本初始化为∞
    used=[False]*n
    sum=0
    dfs3(0,0)                           #从人员 0 开始
    print("求解结果")
    for k in range(0,n):
        print("    人员%d 分配任务%d"%(k,bestx[k]))
    print("    总成本=",bestc)
    print("sum=",sum)
```

对于表 5.3，调用上述 allocate3 算法的执行结果如下：

```
求解结果
    人员 0 分配任务 1
    人员 1 分配任务 0
```

```
        人员 2 分配任务 2
        人员 3 分配任务 3
        总成本=13
sum=9
```

【算法分析】 算法的解空间是一棵排列树,同时复制更优解和求下界的时间为$O(n^2)$,所以最坏的时间复杂度为$O(n^2\times n!)$。例如,在上述实例中$n=4$,经测试不剪支(除去 dfs3 中的 if(bound(cost,i)<bestc))时搜索的结点个数为 65,而剪支后搜索的结点个数为 9。

说明:任务分配问题在 5.3.10 节采用基于子集树框架时最坏时间复杂度为$O(n^n)$,这里采用基于排列树框架的最坏时间复杂度为$O(n!)$,显然$n>2$时$O(n!)$优于$O(n^n)$,实际上由于前者通过 used 判重剪去了重复的分支,其解空间本质上也是一棵排列树,两种算法的最坏时间复杂度都是$O(n!)$。

扫一扫

视频讲解

5.4.4 货郎担问题

1. 问题描述

货郎担问题又称为旅行推销员问题(TSP),是数学领域中著名的问题之一。假设有一个货郎担要拜访n个城市,他必须选择所要走的路径,路径的限制是每个城市只能拜访一次,而且最后要回到原来出发的城市,要求求出路径长度最短的路径。

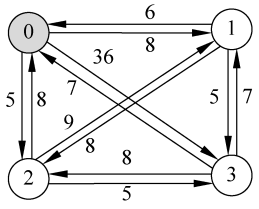


图 5.27 一个 4 城市的道路图

以图 5.27 所示的一个 4 城市的道路图为例,假设起点s为 0,所有从顶点 0 回到顶点 0 并通过所有顶点的路径如下:

路径 1:0→1→2→3→0:28

路径 2:0→1→3→2→0:29

路径 3:0→2→1→3→0:26

路径 4:0→2→3→1→0:23

路径 5:0→3→2→1→0:59

路径 6:0→3→1→2→0:59

最后求得的最短路径长度为 23,最短路径为 0→2→3→1→0。

2. 问题求解

本问题是求路径长度最短的路径,属于求最优解类型。假设图中有n个顶点,顶点的编号为 0~$n-1$,采用邻接矩阵$\boldsymbol{A}$存储。显然 TSP 路径是简单回路(除了起始点和终点相同,其他顶点不重复),可以采用穷举法,以全排列的方式求出所有路径及其长度,再加上回边,在其中找出长度最短的回路即为 TSP 路径,但这样做难以剪支,时间性能较低。

现在采用基于排列树的递归回溯算法,设计当前解向量$\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$,每个$x_i$表示图中一个顶点,实际上每个$x$表示一条路径,初始时将$x_0$置为起点$s$,$x_1$~$x_{n-1}$为其他$n-1$个顶点的编号,$d$表示当前路径的长度,用 bestx 保存最短路径,bestd 表示最短路径长度,将其初始值置为∞。算法 dfs(x,d,s,i)设计的几个重点如下。

① x_0固定作为起点s,不能取其他值,所以不能从$i=0$开始调用 dfs,应该改为从$i=1$(此时$d=0$)开始调用 dfs。为了简单,假设$s=0$,x初始时为(0,1,…,$n-1$),$i=1$时x_1会

取 $x[1..n-1]$中的每一个值(共 $n-1$ 种取值),如图 5.28 所示,当 $x_1=x_1$(1)时,对应路径长度为 $d+A[0][1]$,当 $x_1=x_2$(2)时,对应路径长度为 $d+A[0][2]$,以此类推。归纳起来,当搜索到解空间的第 i 层的某个结点时,x_i 取 $x[i..n-1]$中的某个值后当前路径长度为 $d+A[x[i-1]][x[i]]$。

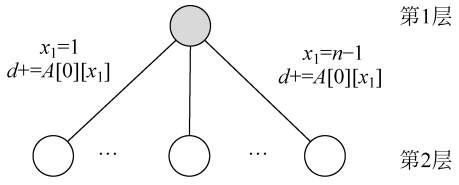


图 5.28 x_1 的各种取值情况

② 当搜索到达某个叶子结点时($i \geqslant n$),对应的 TSP 路径长度应该是 $d+A[x[n-1]][s]$(因为 TSP 路径是闭合的回路),对应的路径是 $x \cup \{s\}$。通过比较所有回路的长度求最优解。

③ 如何剪支呢?若当前已经求出最短路径长度 bestd,如果 x_i 取 x_j 值,对应的路径长度为 $d+A[x[i-1]][x[j]]$,若 $d+A[x[i-1]][x[j]] \geqslant$bestd,说明该路径走下去不可能找到更短路径,终止该路径的搜索,也就是说仅扩展满足 $d+A[x[i-1]][x[j]]<$bestd 的路径。

对应的回溯法求 TSP 问题的算法如下:

```
1  import copy
2  INF=0x3f3f3f3f
3  def dfs(d,s,i):                                    #回溯算法
4      global A,n,x,bestx,bestd
5      if i>=n:                                       #到达一个叶子结点
6          if d+A[x[n-1]][s]<bestd:                   #通过比较求最优解
7              bestd=d+A[x[n-1]][s]                   #求 TSP 路径长度
8              bestx=copy.deepcopy(x)                 #更新 bestx
9      else:
10         for j in range(i,n):                       #试探 x[i]走到 x[j]的分支
11             if A[x[i-1]][x[j]]!=0 and A[x[i-1]][x[j]]!=INF:#若 x[i-1]到 x[j]有边
12                 if d+A[x[i-1]][x[j]]<bestd:        #剪支
13                     x[i],x[j]=x[j],x[i]            #swap(x[i],x[j])
14                     dfs(d+A[x[i-1]][x[i]],s,i+1)
15                     x[i],x[j]=x[j],x[i]            #swap(x[i],x[j])
16
17 def TSP1(A,n,s):                                   #求解 TSP(起始点为 s)
18     global x,bestx,bestd
19     x=[s]                                          #x[0]=s,解向量初始化
20     for i in range(0,n):                           #将非 s 的顶点添加到 x 中
21         if i==s:continue
22         x.append(i)
23     bestx=[0]*n
24     bestd=INF
25     dfs(0,s,1)
26     bestx.append(s)                                #bestx 的末尾添加起始点
27     print("求解结果")
28     print(" 最短路径:",end='')                      #输出最短路径
29     for j in range(0,len(bestx)):
30         if j==0:print(bestx[j],end='')
31         else:print("->%d"%(bestx[j]),end='')
32     print("\n 路径长度:",bestd)
```

当 $A=[[0,8,5,36],[6,0,8,5],[8,9,0,5],[7,7,8,0]]$、$n=4$、$s=1$ 时调用 TSP1(A,n,s)的输出结果如下：

```
求解结果
    最短路径：1-> 0-> 2-> 3-> 1
    路径长度：23
```

【算法分析】 上述算法的解空间是一棵排列树，由于是从第一层开始搜索的，排列树的高度为 n(含叶子结点层)，同时复制更优解的时间为 $O(n)$，所以最坏的时间复杂度为 $O(n\times(n-1)!)$，即 $O(n!)$。

思考题：TSP 问题是在一个图中查找从起点 s 经过其他所有顶点又回到顶点 s 的最短路径，在上述算法中为什么不考虑路径中出现重复顶点的情况？

习题 5

扫一扫

练习题

扫一扫

自测题