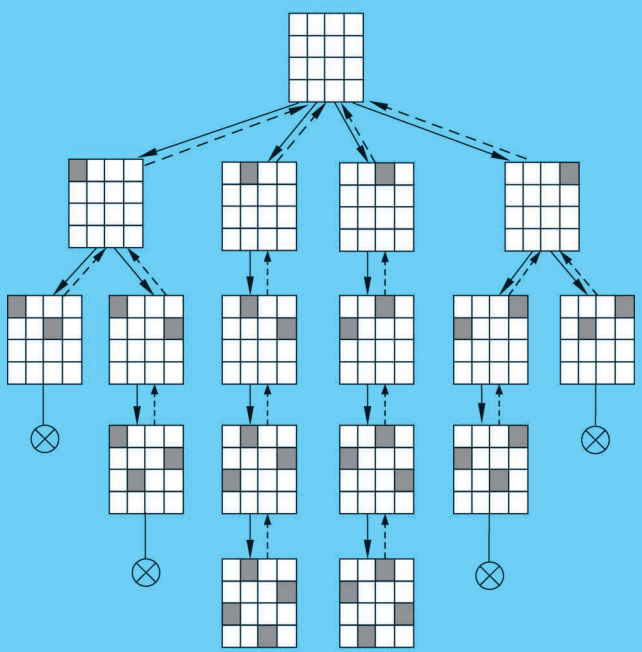
第5章 回溯法

【案例引入】

求 4 皇后问题



解 1　　解 2

本章学习目标：

(1) 掌握解空间的概念、解空间的类型和回溯法的求解过程。

(2) 掌握子集树回溯算法框架以及构造表达式、图着色、子集和、0/1 背包和完全背包问题等经典回溯算法设计方法。

(3) 掌握排列树回溯算法框架以及 n 皇后、任务分配和旅行商问题等经典回溯算法设计方法。

(4) 灵活运用回溯法算法策略解决复杂问题。

5.1 回溯法概述

5.1.1 问题的解空间

扫一扫

视频讲解

回溯法(backtracking)类似于穷举法，主要是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“通用的解题法”之称。一个复杂问题的解决方案是由若干个小的决策步骤组成的决策序列，其解可以表示成解向量 $\boldsymbol{x}=\{x_0,x_1,\cdots,x_{n-1}\}$，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 $x_i\in S_i$，S_i 为 x_i 的取值候选集。$\boldsymbol{x}$ 中各个分量 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**(solution space)，由于解空间一般是一棵树结构，所以也称为解空间树。解空间树中的每个结点确定了所求问题的一个**状态**(state)，因此解空间树也称为状态空间。求解问题的每一步决策对应于解空间树的一个分支结点，而一系列决策求解过程对应着解空间树的生长过程，在许多情况下问题的最终解都会呈现在这棵解空间树的叶子结点上，对应解的结点称为**解结点**，从根结点到解结点的路径称为**解路径**，解路径上所有 x_i 分量的取值确定了问题的一个解。

例如，对于第 2 章中表 2.1 所示的 0/1 背包问题，状态为(cw,cv,i)，表示考虑物品 i 时当前选择物品的总重量和总价值分别是 cw 和 cv，对应的解空间树如图 5.1 所示，根结点是(0,0)，i 为结点的层次(这里层次从 0 开始)，为了简便，结点中没有标注 i 值。解结点是叶子结点(6,8)，表示最大价值是 8，从根结点到解结点对应的解向量 $\boldsymbol{x}=\{0,1,1,1\}$，表示一个装入方案是选择物品 1、2 和 3。

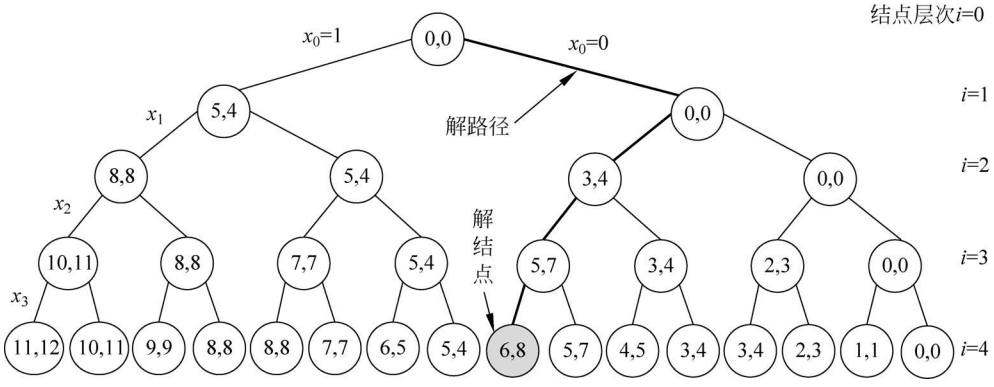


图 5.1 0/1 背包问题的解空间树

归纳起来，解空间树的一般结构如图 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_0，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，解空间树的高度为 $n+1$。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，这是理解解空间树的关键。

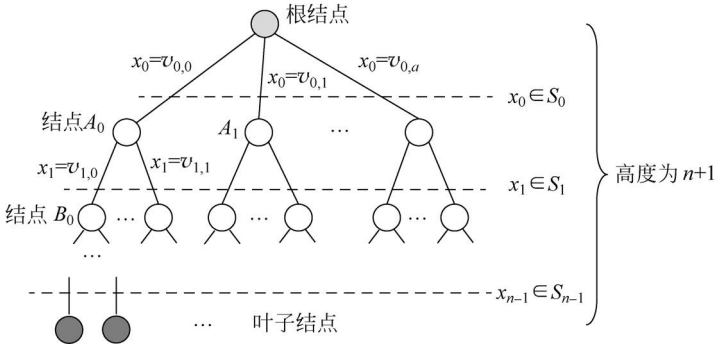


图 5.2　解空间树的一般结构

从形式化角度看，解空间树是 $S_0\times S_1\times\cdots\times S_{n-1}$ 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为过于复杂或结点过多难以画出来。

5.1.2　什么是回溯法

用回溯法求解的问题通常分为两种类型，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**，例如求幂集问题中的每一个子集都是一个解；另外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值最大或最小的最优解，称为求**最优解类型**（或优化问题），例如 0/1 背包问题属于求最优解类型。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。

从前面的讨论看出问题的解包含在对应的解空间树中，剩下的事情就是在解空间树中搜索满足约束条件的解。所谓回溯法就是在解空间树中采用深度优先搜索方法从根结点出发搜索解，与树的先根遍历类似，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）。

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还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**（E-结点）是指正在产

生孩子结点的结点。在当前的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，并成为当前的扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前的扩展结点就成为死结点，所谓**死结点**是指其所有孩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的扩展结点。

如图 5.4 所示，从结点 A 扩展出子结点 B（用实线箭头表示），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，用虚线箭头表示），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 仍是活结点。

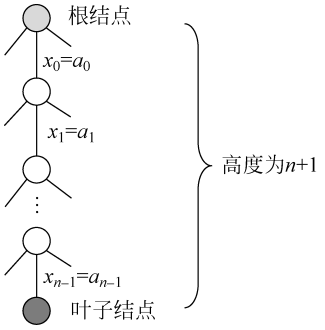


图 5.3 求解的搜索空间

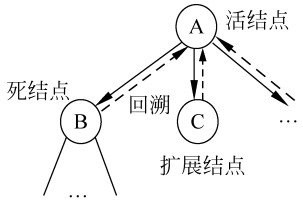


图 5.4 解空间树的搜索过程

简单地说，回溯法就是采用深度优先搜索方法搜索解空间树，并随时判定当前结点是否满足解题要求，满足要求时继续向下搜索，不满足要求时则回溯到树的上一层继续搜索另一棵子树，直到找到问题的解为止。采用回溯法的算法称为**回溯算法**。设计回溯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，不同的问题扩展方式是不同的。

（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，如果要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 0/1 背包问题中，如果选择物品 i 会导致总重量超过背包容量，则终止选择物品 i 的分支的继续搜索。

（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 0/1 背包问题中，如果沿着某个分支走下去无论如何都不可能得到比当前解 bestv 更大的价值，则终止该分支的继续搜索。

（3）改变搜索的顺序：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，这样可能会减少搜索的总结点。

严格来说改变搜索的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树，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 回溯算法分析

通常以回溯法的解空间树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高于穷举法。

5.2 基于子集树的回溯算法框架

5.2.1 解空间树的类型

解空间树通常有两种类型。当所给的问题是从 n 个元素的集合 S 中找出满足某种条件的子集时，相应的解空间树称为**子集树**(subset tree)，如图 5.1 所示的解空间树就是一棵子集树。当所给的问题是确定 n 个元素满足某种性质的排列时，相应的解空间树称为**排列树**(permutation tree)，5.10 节中介绍的求全排列的解空间树就是排列树。

5.2.2 求幂集

求幂集问题的解空间树是最经典的子集树，由此导出子集树的回溯算法框架。为了通用，将求幂集问题改为求含不同整数的集合的所有子集。

问题描述： 有一个含 n 个不同整数的数组 a，设计一个算法求其所有子集(幂集)。例如，$a=\{1,2,3\}$，所有子集是$\{\{1,2,3\},\{1,2\},\{1,3\},\{1\},\{2,3\},\{2\},\{3\},\{\}\}$(输出顺序任意)。

扫一扫

视频讲解

解法 1： 设 $a=\{a_0,\cdots,a_i,\cdots,a_{n-1}\}$，解向量为 $\boldsymbol{x}=\{x_0,\cdots,x_i,\cdots,x_{n-1}\}$，其中 $x_i=0$ 表示不选择 $a[i]$，$x_i=1$ 表示选择 $a[i]$，这里 x 的固定长度为 n。当已经生成解向量的 $\{x_0,\cdots,x_{i-1}\}$ 部分后，再由 $\{a_i,\cdots,a_{n-1}\}$ 产生解向量的 $\{x_i,\cdots,x_{n-1}\}$ 部分。这样用 (i,x) 表示状态，解空间树的根结点对应状态($i=0$，x 的元素均为 0)，目标状态是($i=n$，x 为一个解)。从状态 (i,x) 可以扩展出两个状态(即二选一)：

(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$(i+1,x[i]=0)$。

在一条搜索路径上 i 总是递增的，所以不会出现状态重复的情况。对应的回溯算法如下：

```
vector<int> x;                                  //解向量
void disp(vector<int> &a) {                     //输出一个解
    printf(" {");
    for (int i=0;i<x.size();i++) {
        if (x[i]==1) printf("%d ",a[i]);
    }
    printf("}");
}
void dfs(vector<int> &a,int i) {                //回溯算法
    if (i>=a.size())                            //到达一个叶子结点
        disp(a);                                //输出对应的解
    else {
        x[i]=1; dfs(a,i+1);                     //选择 a[i]
        x[i]=0; dfs(a,i+1);                     //不选择 a[i]
    }
}
void pset1(vector<int> &a) {                    //求幂集算法 1
    int n=a.size();
    x=vector<int>(n);
    dfs(a,0);
}
```

求解 $a=\{1,2,3\}$ 的解空间树如图 5.5 所示，图中方框旁边的“(数字)”表示递归调用次序，从中看出结点层次 i(从 0 开始)与当前考虑的元素 a_i(选择或者不选择 a_i)的下标是一致的，同时每个叶子结点对应一个解，而且每个解对应的解向量的长度相同。

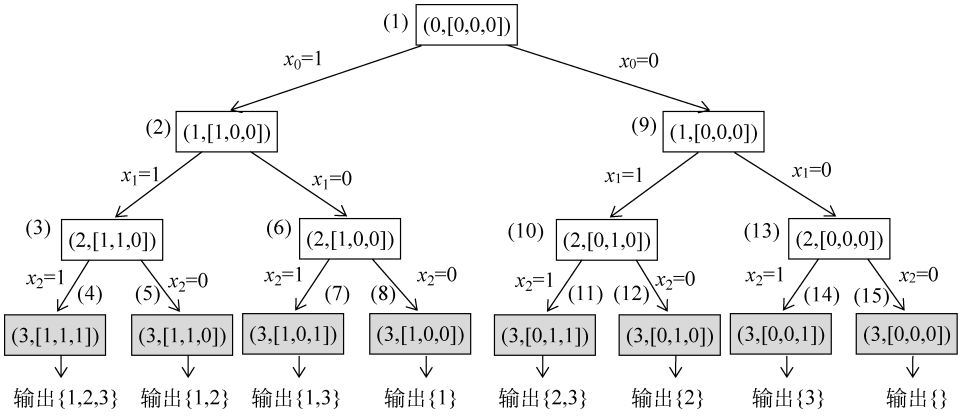


图 5.5 求解 $a=\{1,2,3\}$ 的解空间树

pset1 算法分析：在对应的解空间树中，每个层次为 i(i 从 0 开始)的分支结点对应元素 $a[i]$的选择和不选择两种情况，所以解空间树是一棵高度为 $n+1$ 的满二叉树，叶子结点共有 2^n 个，每个叶子结点对应一个解，输出一个解的时间为 $O(n)$，所以算法的最坏时间复杂度为 $O(n\times 2^n)$。

解法 2：设 $a=\{a_0,\cdots,a_i,\cdots,a_{n-1}\}$，解向量 $x=\{x_0,\cdots,x_i,\cdots,x_{m-1}\}$ 改为直接存放 a 的一个子集(m 为 x 的长度，$0\leqslant m\leqslant n$)，例如，$n=3$，$a=\{1,2,3\}$，$x=\{1\}$ 或者 $x=\{1,3\}$ 等都是该问题的解。从递归角度出发，设 $f(i)$ 表示求以 a_i 开头的子集，首先空集{}肯定是一个子集。

扫一扫

视频讲解

(1) 求以 a_0 开头的子集，$f(0)=\{a_0\}\cup\{a_0$ 合并 $f(1)$的每个元素$\}\cup\{a_0$ 合并 $f(2)$的每个元素$\}=\{\{1\},\{1,2\},\{1,2,3\},\{1,3\}\}$。

(2) 求以 a_1 开头的子集，$f(1)=\{a_1\}\cup\{a_1$ 合并 $f(2)$的每个元素$\}=\{\{2\},\{2,3\}\}$。

(3) 求以 a_2 开头的子集，$f(2)=\{a_2\}=\{\{3\}\}$。

也就是说求 $f(i)$的递推公式如下：

$$f(i)=\{a_i\}\bigcup_{i<j<n}\{a_i \text{ 合并 } f(j) \text{ 的每个元素}\}$$

则 a 的幂集为 $\bigcup_{i=0}^{n-1}f(i)$。例如 $a=\{1,2,3\}$的全部子集就是如图 5.6 所示的递归树中的所有结点。

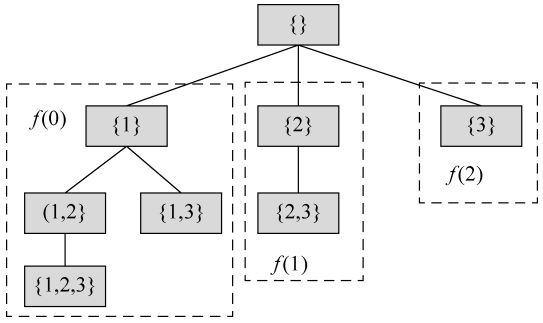


图 5.6 求 $a=\{1,2,3\}$的幂集

上述递推公式直接采用递归算法实现时性能较低(其中存在重复子问题的计算)，这里采用回溯算法，用(i,j,x)表示状态(其中 x 为解向量)，解空间中根结点的状态为$(0,0,\{\})$，用 $j1$ 遍历 $a[j..n-1]$：置 $x[i]=a[j1]$，下一层的结点状态为$(i+1,j1+1,x)$。例如，$a=\{1,2,3\}$，求其幂集的解空间如图 5.7 所示。

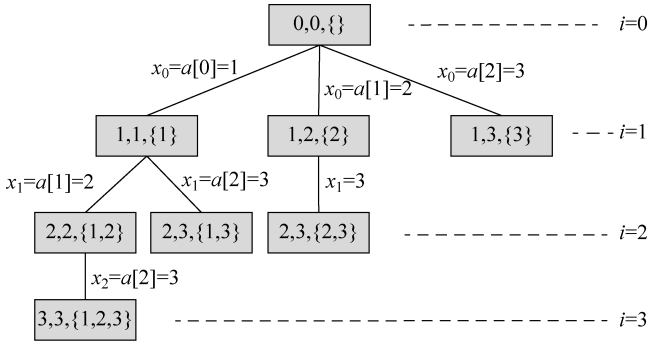


图 5.7 求 $a=\{1,2,3\}$幂集的解空间

对应的回溯算法如下：

```
int x[MAXN];                                    //解向量
void disp(int i) {                              //输出一个解
    printf(" {");
    for (int k=0;k<i;k++)
        printf("%d ",x[k]);
    printf("}");
}
void dfs(vector<int> &a,int i,int j) {          //回溯算法
    disp(i);                                    //输出一个解 x[0..i-1]
    for(int j1=j;j1<a.size();j1++) {
```

```
            x[i]=a[j1];
            dfs(a,i+1,j1+1);
            x[i]=-1;                          //回溯
        }
    }
    void pset2(vector<int> &a) {              //求幂集算法 2
        dfs(a,0,0);
    }
```

解空间中第 i 层的结点(i,j,x)就是为 $x_i(i\leqslant j)$选择$\{a_j,\cdots,a_{n-1}\}$中的一个元素，即 $n-j$ 选一，如图 5.8 所示。

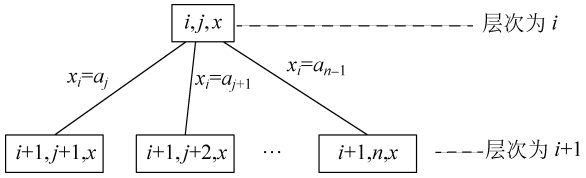


图 5.8 x_i 可以选择 $a[j..n-1]$中的任意元素

解向量 x 用 vector 向量表示，由于参数 i 与解空间中对应结点的层次相同，所以可以省略参数 i，对应的回溯算法如下：

```
vector<int> x;                                //解向量
void disp(vector<int> &x) {                   //输出一个解(子集)
    printf("{");
    for (int k=0;k<x.size();k++)
        printf("%d ",x[k]);
    printf("}");
}
void dfs(vector<int> &a,int j) {              //回溯算法
    disp(x);                                  //输出对应的解
    for(int j1=j;j1<a.size();j1++) {          //j1≥j
        x.push_back(a[j1]);
        dfs(a,j1+1);
        x.pop_back();
    }
}
void pset2(vector<int> &a) {                  //求幂集算法 2
    dfs(a,0);
}
```

pset2 算法分析：在对应的解空间树中恰好有 2^n 个结点，每个结点都是解结点，输出一个解的时间为 $O(n)$，所以算法的最坏时间复杂度为 $O(n\times 2^n)$。

说明：pset1 和 pset2 两个算法的差异如下。

(1) pset1 采用固定的二选一，结点层次 i 与当前考虑元素 a_i 的下标一致，算法简单，方便理解，而 pset2 采用多选一，算法设计相对复杂。

(2) pset1 解空间树中的结点个数几乎是 pset2 解空间树中结点个数的两倍，一般来说解空间树中的结点个数越少则算法的时空性能越好，所以 pset2 的性能好于 pset1。

(3) pset1 求出的全部子集是无序的，而 pset2 求出的全部子集是有序的，即除了第一个空集以外，首先输出以 $a[0]$为首元素的所有子集，接着是以 $a[1]$为首元素的所有子集，以此类推，如果求解问题要求有序性，应该采用以 pset2 为基础的回溯算法。

【例 5.1】 子集Ⅱ(LintCode18★★)。给定一个可能具有重复数字的数组 nums，设计

扫一扫 视频讲解

一个算法返回其所有可能的子集，子集中的每个元素都是非降序的，两个子集间的顺序是无关紧要的，解集中不能包含重复子集。例如，nums＝{1,2,2}，答案是{{2},{1},{1,2,2},{2,2},{1,2},{}}。要求设计如下成员函数：

```
vector<vector<int>> subsetsWithDup(vector<int> &nums) { }
```

扫一扫

源程序

解法1： 利用5.2.2节中pset1算法的思路求解，先对nums递增排序，由于nums中存在重复元素，结果子集也一定重复，这里采用set容器实现去重，最后将set中的子集复制到vector<vector<int>>容器ans中，返回ans即可。

扫一扫

源程序

解法2： 利用5.2.2节中pset2算法的思路求解，先对nums递增排序，在考虑求以nums[i]开头的子集时，用j遍历nums[$i..n-1$]，x_i取$a[j]$值，如果跳过nums[j]＝nums[$j-1$]($j>i$)的nums[j]便可以实现去重。

5.2.3 子集树回溯算法框架

设问题的解是一个n维向量$\{x_0,x_1,\cdots,x_{n-1}\}$，约束函数表示每个分量$x_i$应满足的约束条件，记为constraint($x_i$)；限界函数记为bound($x_i$)，一般地，解空间为子集树的递归回溯框架如下：

```
int x[n];                                          //解向量,全局变量
void dfs(int i){                                   //子集树的递归框架
    if(i>=n)                                       //搜索到叶子结点
        产生一个可能解;
    else {
        for (j=下界;j<=上界;j++){                   //用j枚举x[i]所有可能的选择
            x[i]=j;                                //产生一个可能的解分量
            ...                                    //其他操作
            if (constraint(i) && bound(i))
                dfs(i+1);                          //满足约束条件和限界函数,继续下一层
        }
    }
}
```

采用上述算法框架需要注意以下几点：

(1) i从0开始调用上述回溯算法框架，此时根结点为第0层，叶子结点为第n层。当然i也可以从1开始，这样根结点为第1层，叶子结点为第$n+1$层，需要将上述代码中的“if(i>＝n)”改为“if (i>n)”。

(2) 在上述框架中通过for循环用j枚举$x[i]$所有可能的路径，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) 这里回溯框架只有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。

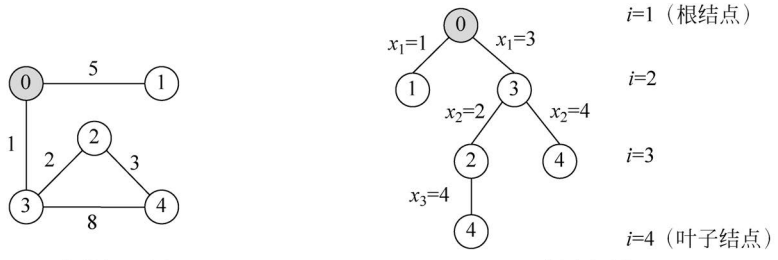
5.3 图的路径搜索

问题描述： 给定一个含n个顶点的带权无向图以及图中两个顶点s和t，设计一个算法求s到t的所有路径及其路径长度。

解 假设带权无向图采用邻接矩阵存放。例如如图5.9(a)所示的带权无向图的邻

接矩阵如下：

$$\boldsymbol{A}=\begin{bmatrix}0 & 5 & \infty & 1 & \infty\\ 5 & 0 & \infty & \infty & \infty\\ \infty & \infty & 0 & 2 & 3\\ 1 & \infty & 2 & 0 & 8\\ \infty & \infty & 3 & 8 & 0\end{bmatrix}$$



(a) 一个带权无向图　　(b) 解空间树

图 5.9　一个带权无向图及其解空间树

现在求从顶点 s 到顶点 t 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 $s\to t$ 的路径，由于路径是一个顶点序列，所以对应的解向量 $\boldsymbol{x}=\{x_0,x_1,\cdots,x_{k-1}\}$表示一条路径。下面以 $s=0,t=4$ 为例进行讨论，$x_0=0$(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，对应的解空间树如图 5.9(b)所示，图中的 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，根结点的层次 $i=1$，从中看出 $S_1=\{1,3\}$，$S_2=\{2,4\}$，$S_3=\{4\}$，其中包含该问题的两个解 $\boldsymbol{x}=\{0,3,2,4\}$和 $\boldsymbol{x}=\{0,3,4\}$。

说明：在本问题中结点的扩展就是从对应的顶点找所有相邻顶点，即多选一，对应的解空间树属于子集树，由于找到的路径长度不一定相同，所以解向量并不是固定长度的。

为了避免路径中出现重复的顶点，采用 visited 数组判重，在邻接搜索中跳过当前路径中已经出现的顶点。这里还需要求路径长度，为了简单，将路径长度存放在 x 的末尾。对应的回溯算法如下：

```
const int INF=0x3f3f3f3f;
int n=5;
vector<vector<int>> A={{0,5,INF,1,INF},{5,0,INF,INF,INF},    //邻接矩阵
                       {INF,INF,0,2,3},{1,INF,2,0,8},{INF,INF,3,8,0}};
vector<vector<int>> ans;                                      //存放答案
vector<int> x;                                                //解向量
vector<int> visited;
void dfs(int i,int curlen,int t) {                            //回溯算法
    if (i==t) {                                               //找到终点
        ans.push_back(x);                                     //将 x 添加到 ans 中
        ans.back().push_back(curlen);                         //在路径的末尾添加路径长度
    }
    else {
        for (int j=0;j<n;j++) {
            if(i==j || A[i][j]==INF) continue;                //剪支：跳过没有边的顶点
            if (visited[j]) continue;                         //剪支:跳过在路径中的顶点
            visited[j]=1;                                     //x[i]选择顶点 j
            x.push_back(j);                                   //置访问标记
            curlen+=A[i][j];                                  //增加路径长度
            dfs(j,curlen,t);                                  //继续搜索
```

```
                curlen-=A[i][j];                                    //路径长度回溯
                x.pop_back();                                       //路径回溯
                visited[j]=0;                                       //访问标记回溯
            }
        }
    };
    void allpath(int s,int t) {                                     //求解算法
        x.push_back(s);
        visited=vector<int>(n,0);
        visited[s]=1;
        dfs(s,0,t);                                                 //i从0开始
        printf("从%d到%d的所有路径:\n");                             //输出结果
        for(int i=0;i<ans.size();i++) {
            printf(" 路径%d: 长度=%d, 路径:",i+1,ans[i].back());
            for(int j=0;j<ans[i].size()-1;j++)
                printf(" %d",ans[i][j]);
            printf("\n");
        }
    }
```

调用 allpath(0,4)的求解结果如下：

```
从0到4的所有路径:
    路径1: 长度=6, 路径: 0 3 2 4
    路径2: 长度=9, 路径: 0 3 4
```

扫一扫

视频讲解

allpath 算法分析：在算法中调用 dfs(s,0,t),考虑最坏情况,s 到 t 的路径为(s,x_1,…,x_{n-2},t),x_i 可以是除了 s 和 t 以外的任意不重复的顶点,所以最坏时间复杂度为 $O((n-2)!)$。

【例 5.2】 路径搜索(LintCode1647★★)。给定一个有 n($n\leqslant10$)个顶点、m($m\leqslant50$)条边的无向图,顶点的编号是 0～$n-1$,设计一个算法输出从 s 点出发到达 t 点的所有简单路径,输出的简单路径按字典序排序。当一条路径不会经过某个顶点超过一次时,称之为简单路径。例如,$n=4$,$g=\{\{0,1\},\{0,2\},\{1,2\},\{1,3\},\{3,2\}\}$,$s=0$,$t=2$,对应的图如图 5.10 所示,从 0 到 2 有 3 条路径,按字典序输出的结果是$\{\{0,1,2\},\{0,1,3,2\},\{0,2\}\}$。要求设计如下成员函数：

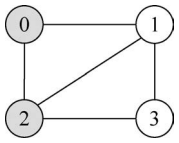


图 5.10 一个无向图

```
vector<vector<int>> getPath(int n,vector<vector<int>> &g,int s,int t) { }
```

扫一扫

源程序

扫一扫

源程序

解法 1：给定的无向图采用邻接矩阵 $\boldsymbol{A}$ 存放,采用本节前面讨论的路径搜索方法求解,在搜索顶点 i 的相邻点 j 时,j 以从 0 到 $n-1$ 的顺序试探,所以当 s 到 t 有多条路径时会按照字典序找到这些路径。

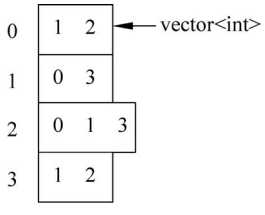


图 5.11 邻接表 G

解法 2：给定的无向图采用邻接表 G 存放,G 的类型为 vector<vector<int>>,$G[i]$表示顶点 i 的所有出边邻接点,例如图 5.10 所示的无向图对应的邻接表 G 如图 5.11 所示。

由于需要按字典序输出 s 到 t 的所有路径,所以要保证 $G[i]$中的元素按顶点编号递增排列,采用的方法是在创建 $G[i]$后调用 sort()实现排序,其他与解法 1 相同。

5.4 构造表达式

扫一扫

视频讲解

问题描述：设计一个算法在1、2、……、9(顺序不能变)数字之间插入＋或－或什么都不插入，使得计算结果总是100，并求所有的可能性。例如1＋2＋34－5＋67－8＋9＝100。

解 用数组 a 存放1～9的整数，设计解向量 $\boldsymbol{x}$，$x[i]$表示在 $a[i]$前面插入的运算符，其示意图如图5.12所示。$x[i]$只能取值'＋'、'－'或者空格(三选一)。设计回溯算法 dfs(sum,prev,i)，其中 sum 表示考虑 $x[i]$取值时前面表达式的计算结果(初始值为 $a[0]$)，prev 表示考虑 $x[i]$取值时前面的运算数(初始值为 $a[0]$)，i 从1开始($a[0]$前面没有运算符)：

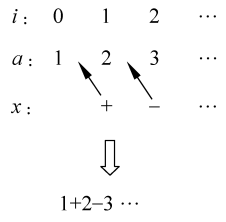


图 5.12 表达式示意图

(1) 若 $x[i]$取值'＋'，sum＋＝$a[i]$，prev＝$a[i]$，继续搜索下一层，返回时恢复 sum 和 prev。

(2) 若 $x[i]$取值'－'，sum－＝$a[i]$，prev＝－$a[i]$，继续搜索下一层，返回时恢复 sum 和 prev。

(3) 若 $x[i]$取值' '，则需要合并整数，例如 prev＝2，$a[2]$＝3，若 $x[2]$取值' '，合并的整数是23。如 prev＝－2，$a[2]$＝3，若 $x[2]$取值' '，合并的整数是－23，所以需要先执行 sum－＝prev 减去前面的元素值，再执行 tmp＝prev＊10±a[i](±为原 prev 的符号)得到合并的整数，最后执行 sum＋＝tmp 得到合并结果(tmp 为新的前面的运算数)，继续搜索下一层，返回时恢复 sum 和 prev。

当 i＝9时到达一个叶子结点，若 sum＝100 对应一个解，构造对应的表达式并添加到 ans 中，最后输出 ans。对应的算法如下：

```
#define N 9
int a[N];
vector<string> ans;                          //存放答案
char x[N];                                   //解向量
void dfs(int sum,int prev,int i) {           //回溯算法
    if (i==N) {                              //到达一个叶子结点
        if (sum==100) {                      //找到一个解
            string s=to_string(a[0]);
            for (int j=1;j<N;j++) {
                if (x[j]!=' ') s+=x[j];
                s+=to_string(a[j]);
            }
            s+="=100";
            ans.push_back(s);
        }
    }
    else {
        x[i]='+';                            //在位置i插入'+'
        sum+=a[i];                           //计算当前表达式的值
        dfs(sum,a[i],i+1);
        sum-=a[i];                           //回溯
        x[i]='-';                            //在位置i插入'-'
```

```
            sum-=a[i];                                  //计算当前表达式的值
            dfs(sum,-a[i],i+1);
            sum+=a[i];                                  //回溯
            x[i]=' ';                                   //在位置 i 插入' '
            sum-=prev;                                  //先减去前面的元素值
            int tmp;                                    //计算新合并值
            if (prev>0)
                tmp=prev*10+a[i];                       //如 prev=2,a[i]=3,结果为 23
            else
                tmp=prev*10-a[i];                       //如 prev=-2,a[i]=3,结果为-23
            sum+=tmp;                                   //计算合并结果
            dfs(sum,tmp,i+1);
            sum-=tmp;                                   //回溯 sum
            sum+=prev;
        }
    }
    void express() {
        for (int i=0;i<N;i++) a[i]=i+1;                 //为 a 赋值 1,2,…,9
        dfs(a[0],a[0],1);                               //插入位置 i 从 1 开始
        printf("求解结果\n");
        for(int i=0;i<ans.size();i++)
            cout << " (" << i+1 << ") " << ans[i] << endl;
    }
```

调用 express 算法的输出结果如下：

```
求解结果
    (1) 1+2+3-4+5+6+78+9=100
    (2) 1+2+34-5+67-8+9=100
    (3) 1+23-4+5+6+78-9=100
    ...
    (11) 123-45-67+89=100
```

express 算法分析：在整数序列 1～9 中有 8 个位置，每个位置可以插入 3 个运算符之一，所以执行时间为 3^8，如果给定整数是 1～n，则时间复杂度为 $O(3^n)$。

扫一扫

视频讲解

【例 5.3】 目标和(LintCode1208★★)。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 s($-1000 \leqslant s \leqslant 1000$)。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 s 的不同表达式的数目。要求设计如下成员函数：

```
int findTargetSumWays(vector<int>& nums, int s) { }
```

解 用 ans 表示满足要求的解个数(初始为 0),设置解向量 $x=\{x_0,x_1,\cdots,x_{n-1}\}$,$x_i$ 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=s,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 x,仅设计 expv 即可。对应的程序如下：

```
class Solution {
    int ans;                                            //存放解个数
public:
```

```
    int findTargetSumWays(vector<int>& nums,int s) {
        ans=0;
        dfs(nums,s,0,0);
        return ans;
    }
    void dfs(vector<int>& nums,int s,int i,int expv) {   //回溯算法
        if (i==nums.size()) {                            //到达一个叶子结点
            if(expv==s) ans++;                           //找到一个解
        }
        else {
            expv+=nums[i];                               //nums[i]前选择'+'
            dfs(nums,s,i+1,expv);
            expv-=nums[i];                               //回溯:恢复 expv
            expv-=nums[i];                               //nums[i]前选择'-'
            dfs(nums,s,i+1,expv);
            expv+=nums[i];                               //回溯:恢复 expv
        }
    }
};
```

上述程序提交时通过,执行用时为 163ms,内存消耗为 5.59MB。

5.5 图的 m 着色问题

扫一扫

视频讲解

问题描述:给定一个连通图 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。如果有一种着色法可以使 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 着色问题是给定图 G 和 m 种颜色,找出所有不同的着色方案数。例如,如图 5.13 所示的无向连通图,$m=3$ 时不同的着色方案数为 12。

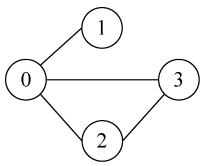


图 5.13 一个无向连通图

解 对于连通图 G,采用邻接矩阵 $\boldsymbol{A}$ 存放,这里的 $\boldsymbol{A}$ 是一个 0/1 矩阵。图中的顶点编号为 0～$n-1$,共 m 种颜色,颜色编号为 0～$m-1$。设计解向量 $x=\{x_0,x_1,\cdots,x_{n-1}\}$,其中 $x[i]$表示顶点 i 的着色,图中每个顶点可能的着色为 0～$m-1$(初始时 $x[i]$置为-1 表示未着色),所以有 $0\leqslant x[i]\leqslant m-1$,相当于 m 选一,对应的解空间是一棵 m 叉树,高度为 $n+1$。i 从 0 开始,当 $i=n$ 时对应一个叶子结点,表示找到一种着色方案,将着色方案数 ans 增 1,最后输出 ans 即可。对应的回溯算法如下:

```
int n=4;
int A[MAXN][MAXN]={{0,1,1,1},{1,0,0,0},{1,0,0,1},{1,0,1,0}};
int ans=0;                                   //全局变量,累计解个数
int x[MAXN];                                 //全局变量,x[i]表示顶点 i 的着色
bool judge(int i,int j) {                    //判断顶点 i 是否可以着色 j
    for(int k=0;k<n;k++) {
        if(A[i][k]==1 && x[k]==j)            //存在相同颜色的顶点
            return false;
    }
    return true;
}
void dfs(int m,int i) {                      //回溯算法
    if (i>=n)                                //到达一个叶子结点
        ans++;                               //着色方案数增 1
```

```
        else {
            for (int j=0;j<m;j++) {                    //试探每一种着色
                x[i]=j;
                if (judge(i,j))                        //可以着色 j,进入下一个顶点着色
                    dfs(m,i+1);
                x[i]=-1;                               //回溯
            }
        }
    }
    void color(int m) {                                //求解算法
        memset(x,0xff,sizeof(x));                      //x 初始化所有元素为-1
        dfs(m,0);
        printf("着色方案数:%d\n",ans);
    }
```

color 算法分析：算法中每个顶点试探 m 种颜色，解空间树是一棵 m 叉树(子集树)，每个结点判断当前着色是否合适的时间为 $O(n)$，所以算法的时间复杂度为 $O(n\times m^n)$。

扫一扫

视频讲解

【例 5.4】 频道分配(POJ1129，时间限制为 1000ms，空间限制为 10 000KB)。在非常大的区域广播时需要利用中继器加强信号，每个中继器使用不同的频道，以便不会相互干扰，给定一个中继器网络的描述，求所需的最小频道数目。

输入格式：输入包含多个中继器网络描述，每个描述以包含中继器个数的行开头，个数介于 1 和 26 之间，中继器由以 A 开头的连续大写字母表示。例如，10 个中继器的名称为 A、B、C、……、I 和 J，输入 0 表示结束。在中继器个数之后是相邻关系列表，每行具有形如“A：BCDH”的格式，表示中继器 B、C、D 和 H 与中继器 A 相邻，第一行描述与中继器 A 相邻的中继器，第二行描述与 B 相邻的中继器，以此类推。如果一个中继器不与任何其他中继器相邻，则形如“A：”。中继器按字母顺序列出。注意相邻关系是对称的，如果 A 与 B 相邻，则 B 必然与 A 相邻。另外，由于中继器位于一个平面内，连接相邻中继器形成的图形没有任何相交的线段。

输出格式：对于每个中继器网络描述，输出一行包含所需的最小频道数。样例输出显示了这一行的格式，当只需要一个通道时，请注意通道是单数形式。

输入样例：

```
2
A:
B:
4
A:BC
B:ACD
C:ABD
D:BC
0
```

输出样例：

```
1 channel needed.
3 channels needed.
```

扫一扫

源程序

解 本题的每个中继器网络描述可以建立这样的无向图，每个中继器看成一个顶点，相邻关系用一条无向边表示，如果将边看成着色关系，同一条边的两个顶点不能着相同的元素，这样就转换为图着色问题，用 m 表示当前颜色数目(颜色编号为 0～$m-1$)，实际上是求 m 可着色的最小 m，用 minm 表示(初始为∞，实际上 4 色定理表明最多 4 种颜色即

可，所以 minm 可以初始为 4）。

同样用 x 作为解向量（所有元素初始化为 -1），m 从 0 开始试探，与前面讨论的图 m 着色问题的求解过程相比，在为顶点 i 选择颜色 j 时，j 从 0 到 $m-1$ 循环试探，试探成功就继续走下去，否则回溯，除此之外增加另外一种选择，即增加一种颜色，置 $x[i]=m$，然后在颜色增加的情况下继续走下去，当 $i=n$ 时说明找到了一种 m 着色方案，置 minm=min(minm,m)，最后输出 minm 即可。例如，对于图 5.13 所示的连通图，求最少颜色数的过程如图 5.14 所示，图中方框为 dfs(m,i)，带阴影的结点为解结点，minm=min(3,3,4)=3，说明该图着色所需的颜色数最少为 3 种。

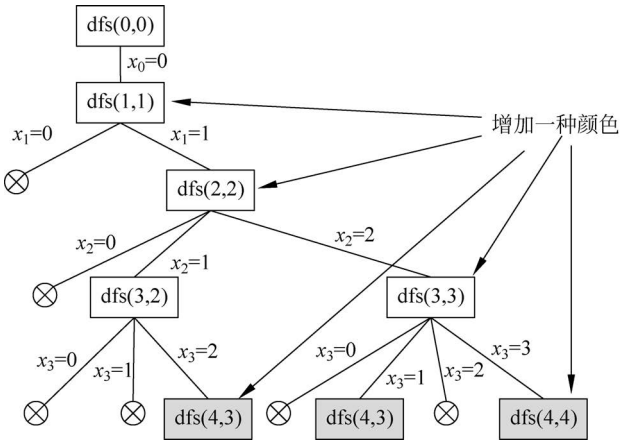


图 5.14　求图 5.13 的最少颜色数的过程

5.6 子集和问题

扫一扫

视频讲解

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 $\{1,5,2\}$。

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 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 5.15 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。

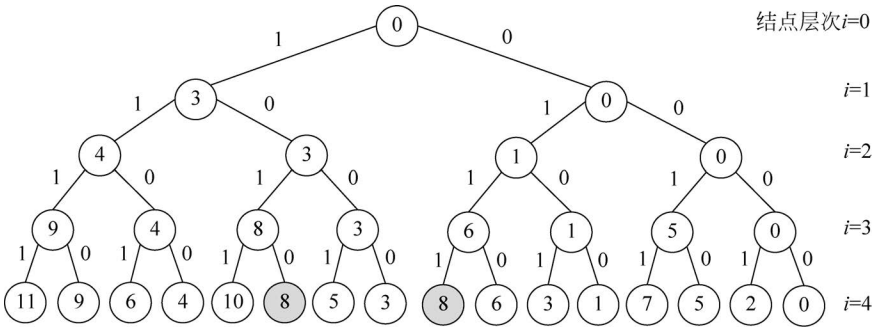


图 5.15　求 $a=\{3,1,5,2\}$，$t=8$ 时子集和的解空间

对应的回溯算法如下：

```
int n,t;
vector<int> a;                          //存放所有整数
int cnt=0;                              //累计解个数
int tot=0;                              //累计搜索的结点个数
vector<int> x;                          //解向量
void disp() {                           //输出一个解
    printf(" 第%d 个解,",++cnt);
    printf("选取的数为: ");
    for (int i=0;i<x.size();i++) {
        if (x[i]==1)
            printf("%d ",a[i]);
    }
    printf("\n");
}
void dfs(int cs,int i) {                //回溯算法
    tot++;                              //累计调用次数
    if (i>=n) {                         //到达一个叶子结点
        if (cs==t) disp();              //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]
        dfs(cs+a[i],i+1);
        x[i]=0;                         //不选取整数 a[i]
        dfs(cs,i+1);
    }
}
void subs1(vector<int> &A,int T) {      //求解子集和问题
    n=A.size();
    a=A;
    t=T;
    x=vector<int>(n);
    printf("求解结果\n");
    dfs(0,0);                           //i 从 0 开始
    printf("tot=%d\n",tot);
}
```

对于子集和问题 $A=\{3,1,5,2\}$，$T=8$，调用 subs1(A，T)时的输出结果如下：

```
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
tot=31
```

subs1 算法分析：对于含 n 个元素的数组 a 和正整数 t，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 中的所有元素都是正整数，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 5.16 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。对应的回溯算法如下：

```
void dfs(int cs,int i) {                    //回溯算法
    tot++;                                  //累计调用次数
    if (i>=n) {                             //到达一个叶子结点
        if (cs==t) disp();                  //找到一个满足条件的解,输出
    }
    else {                                  //没有到达叶子结点
        if (cs+a[i]<=t) {                   //左孩子结点剪支
            x[i]=1;                         //选取整数 a[i]
            dfs(cs+a[i],i+1);
        }
        x[i]=0;                             //不选取整数 a[i]
        dfs(cs,i+1);
    }
}
void subs2(vector<int> &A,int T) {          //求解子集和问题
    n=A.size();
    a=A;
    t=T;
    x=vector<int>(n);
    printf("求解结果\n");
    dfs(0,0);                               //i 从 0 开始
    printf("tot=%d\n",tot);
}
```

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
tot=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

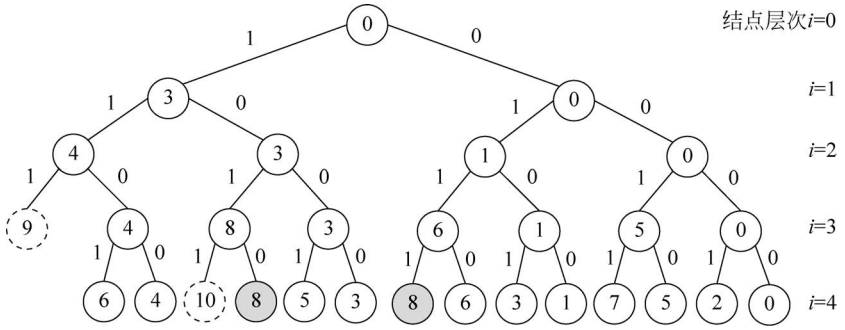


图 5.16 求 $a=\{3,1,5,2\}$，$t=8$ 时子集和的搜索空间

所有整数的和为 rs=rs−a[i](a[i+1]+…+a[n−1])，若 cs+rs<t 成立，说明即使选择所有剩余整数，其和都不可能达到 t，所以右剪支就是仅扩展满足 cs+rs≥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 5.17 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。

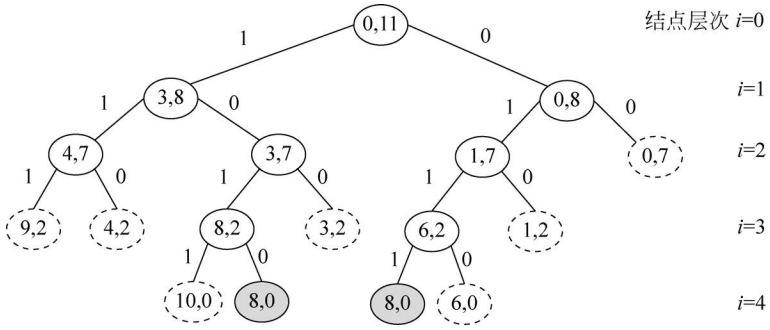


图 5.17 求 $a=\{3,1,5,2\}$，$t=8$ 时子集和的搜索过程

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搜索。

对应的回溯算法如下：

```
void dfs(int cs,int rs,int i) {                    //回溯算法
    tot++;                                         //累计调用次数
    if (i>=n) {                                    //到达一个叶子结点
        if (cs==t) disp();                         //找到一个满足条件的解,输出
    }
    else {                                         //没有到达叶子结点
        rs-=a[i];                                  //求剩余整数的和
        if (cs+a[i]<=t) {                          //左孩子结点剪支
            x[i]=1;                                //选取整数 a[i]
            dfs(cs+a[i],rs,i+1);
        }
        if (cs+rs>=t) {                            //右孩子结点剪支
            x[i]=0;                                //不选取整数 a[i]
            dfs(cs,rs,i+1);
        }
        rs+=a[i];                                  //恢复剩余整数的和(回溯)
    }
```

```
}
void subs3(vector<int>&A,int T){          //求解子集和问题
    n=A.size();
    a=A;
    t=T;
    x=vector<int>(n);
    int rs=0;                             //表示所有整数的和
    for (int j=0;j<n;j++)                 //求 rs
        rs+=a[j];
    printf("求解结果\n");
    dfs(0,rs,0);                          //i 从 0 开始
    printf("tot=%d\n",tot);
}
```

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
tot=10
```

subs2 和 subs3 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，所以这样的两个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算法中的重要性。

【例 5.5】 目标和(LintCode1208★★)。问题描述参见例 5.3。

扫一扫 视频讲解

解 不妨用 a 表示 nums 数组，先求出 a 中所有整数的和 sum，显然 sum$<s$ 时即使全部加上'+'也不可能成立，此时返回 0(无解)，否则本问题就是求以下等式成立的不同表达式的数目(±表示取'+'或者'−'之一)。

$$\pm a[0]\pm a[1]\pm\cdots\pm a[n-1]=s$$

用 sum 减去两边后转换为：

$$(a[0]+a[1]+\cdots+a[n-1])-(\pm a[0]\pm a[1]\pm\cdots\pm a[n-1])=\text{sum}-s$$

等同于：

$$(a[0]\mp a[0])+(a[1]\mp a[1])+\cdots+(+a[n-1]\mp a[n-1])=\text{sum}-s$$

扫一扫 源程序

对于$(a[i]\mp a[i])$部分，如果取'−'(对应原来的'+')则为 0，如果取'+'(对应原来的'−')则为 $2a[i]$，考虑只取'+'的部分(其他取'−')，假设对应的下标为 $i_1,i_2,\cdots,i_k$，则为：

$$2a[i_1]+2a[i_2]+\cdots+2a[i_k]=\text{sum}-s,\quad \text{即 } a[i_1]+a[i_2]+\cdots+a[i_k]=(\text{sum}-s)/2$$

其中 $i_1,i_2,\cdots,i_k$ 是 0,1,…,$n-1$ 的一个子序列，这样该问题等价于在原 a 数组中选择添加'−'的元素的和等于(sum$-s$)/2 的组合总数，属于典型的子集和问题。由于(sum$-s$)/2 一定是整数，所以(sum$-s$)为奇数时无解，返回 0。采用左、右剪支的回溯算法求解。

5.7 简单装载问题

扫一扫 视频讲解

问题描述：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},对应集装箱的重量和达到最大值 t。

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,其重量和都不可能达到 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,其选择的集装箱重量和为 cw(由于左剪支的原因,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。对应的递归算法如下:

```
vector<int> w;
int t;
int n;
vector<int> x;                          //解向量
vector<int> bestx;                      //存放最优解向量
int bestw=0;                            //存放最优解的总重量,初始化为 0
int tot=0;                              //累计搜索的结点个数
void dfs(int cw,int rw,int i) {          //回溯算法
    tot++;
    if (i>=n) {                         //到达一个叶子结点
        if (cw>bestw) {                 //找到一个满足条件的更优解
            bestw=cw;                   //保存更优解
            bestx=x;
        }
    }
    else {                              //尚未找完所有集装箱
        rw-=w[i];                       //求剩余集装箱的重量和
        if (cw+w[i]<=t) {               //左孩子结点剪支:选择满足条件的集装箱
            x[i]=1;                     //选取集装箱 i
            cw+=w[i];                   //累计当前所选集装箱的重量和
            dfs(cw,rw,i+1);
            cw-=w[i];                   //恢复当前所选集装箱的重量和(回溯)
        }
        if (cw+rw>bestw) {              //右孩子结点剪支
            x[i]=0;                     //不选择集装箱 i
            dfs(cw,rw,i+1);
        }
        rw+=w[i];                       //恢复剩余集装箱的重量和(回溯)
    }
}
```

```
void loading(vector<int> &W,int T) {                    //求解简单装载问题
    w=W;
    t=T;
    n=w.size();
    int rw=0;
    for (int i=0;i<n;i++)                               //累计全部集装箱的重量和 rw
        rw+=w[i];
    x=vector<int>(n);
    dfs(0,rw,0);                                        //i 从 0 开始
    printf("求解结果\n");
    for (int i=0;i<n;i++) {                             //输出最优解
        if (bestx[i]==1)
            printf(" 选取第%d 个集装箱\n",i);
    }
    printf(" 总重量=%d\n",bestw);
    printf("tot=%d\n",tot);
}
```

当 $w=\{5,2,6,4,3\}$，$t=10$ 时调用上述算法的求解结果如下：

```
求解结果
    选取第 0 个集装箱
    选取第 1 个集装箱
    选取第 4 个集装箱
    总重量=10
tot=16
```

实际上还有另外一个最优解，即选择第 2 个和第 3 个集装箱，它们的重量和是相等的。

说明： 在上述 dfs 算法的左结点扩展中，3 条语句 cw+=w[i],dfs(cw,rw,x,i+1)和 cw-=w[i]可以用一条语句 dfs(cw+w[i],rw,x,i+1)等价地替换。

loading 算法分析： 该算法的解空间树中有 $2^{n+1}-1$ 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 $O(n)$)，所以在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 5.18 所示。

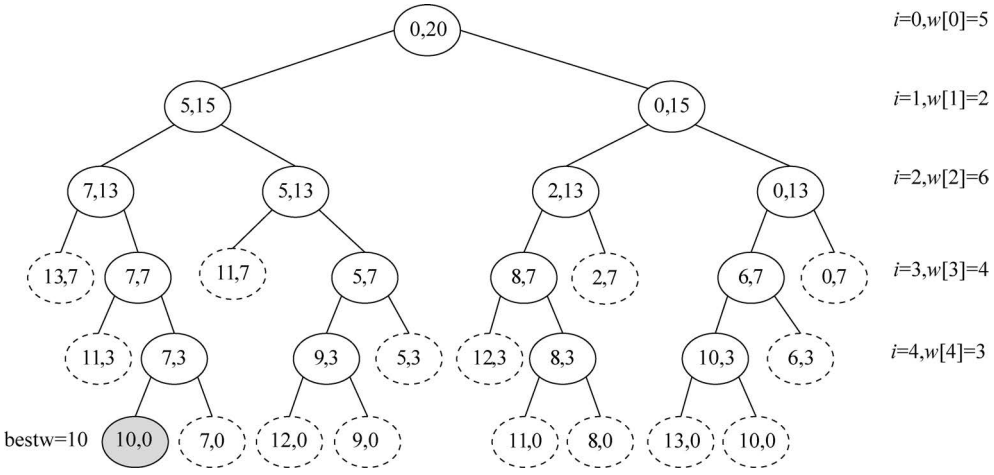


图 5.18 装载实例的搜索空间

5.8 0/1 背包问题

扫一扫

视频讲解

问题描述：问题描述见 2.4 节。

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),要求价值最大的装入方案,属于求最优解类型。

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,后面涉及按单位重量价值递减排序,存放物品的结构体类型如下:

```
struct Goods {                                  //物品类型
    int no;                                     //物品的编号
    int w;                                      //物品的重量
    int v;                                      //物品的价值
    Goods(int no,int w,int v) {                 //构造函数
        this->no=no;
        this->w=w;
        this->v=v;
    }
    bool operator<(const Goods&s) const {       //用于按 v/w 递减排序
        return (double)v/w>(double)s.v/s.w;
    }
};
```

例如,表 2.1 中的 4 个物品采用 g 容器存放:

```
vector<Goods> 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。

1) 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。

2) 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 g 中所有物品按单位重量价值递减排序,例如表 2.1 中物品排序后的结果如表 5.1 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 5.1 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重量和(其中不包

含 $w[i]$),cv 表示当前选择的物品价值和(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 $i+1$、……这样做直到物品 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 开始选择的物品价值和的最大值为 $r(i)$:

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

也就是说,从结点 A 出发的所有路径中最大价值为 bound(cw,cv,i)=cv+$r(i)$,如图 5.19 所示。

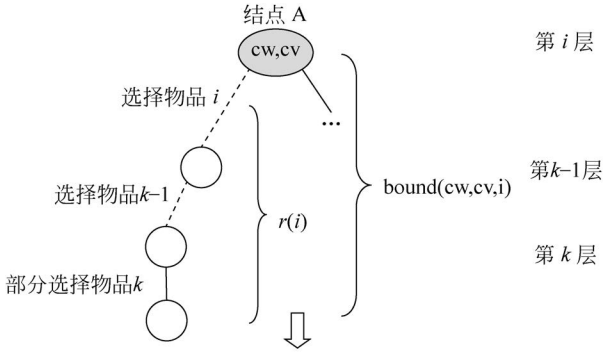


图 5.19 bound(cw,cv,i)

对应的求上界函数值的算法如下:

```
double bound(int cw,int cv,int i) {                    //计算第 i 层结点的上界函数值
    int rw=W-cw;                                       //背包的剩余容量
    double b=cv;                                       //表示物品价值的上界值
    int j=i;
    while (j<n && g[j].w<=rw) {
        rw-=g[j].w;                                    //选择物品 j
        b+=g[j].v;                                     //累计价值
        j++;
    }
    if (j<n)                                           //最后物品 k=j+1 只能部分装入
        b+=(double)g[j].v/g[j].w*rw;
    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,且从物品 $i+1$ 开始选择的物品价值和的最大值为 $r(i+1)$。如果之前已经求出一个最优解 bestv,当 cv+$r(i+1)\leqslant$bestv 时说明不选择物品 i 后面无论如何也不能够找到更优解。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。

说明: 上述 bound(cw,cv,$i+1$)算法中包含物品 k 的一部分价值,这是不是与 0/1 背包问题矛盾呢？答案是不矛盾,这里的上界值表示沿着该路径走下去可能装入背包的最大价值,是一种启发搜索信息,并不是真的取物品 k 的一部分。

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 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 层的结点的 b，b＝bound(cw,cv,i)，若 b≤bestv 则停止右分支的搜索，也就是仅扩展满足 b＞bestv 的右孩子结点。

对于表 2.1 所示的实例，n＝4，按 v/w 递减排序后如表 5.1 所示，初始时 bestv＝0，求解过程如图 5.20 所示，图中两个数字的结点为(cw,cv)，只有右结点标记为(cw,v,ub)，虚框结点表示被剪支的结点，带阴影的结点是最优解结点，其求解结果与递归法和穷举法完全相同，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。

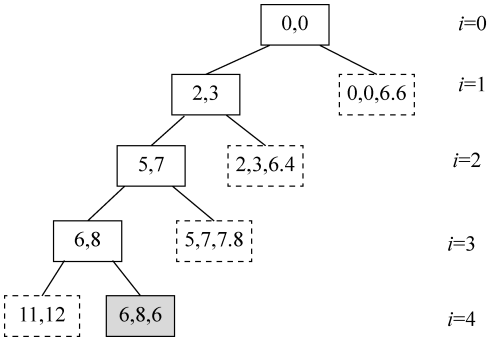


图 5.20　0/1 背包问题实例的搜索空间

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。

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个数为 31，剪支后搜索的结点个数为 5。

对应的递归算法如下：

```
vector<Goods> g;                    //存放全部物品
int W;                              //背包的容量
int n;                              //物品的个数
vector<int> x;                      //解向量
vector<int> bestx;                  //存放最优解向量
int bestv=0;                        //存放最大价值，初始为 0
int tot=0;                          //累计搜索的结点个数
```

```
double bound(int cw,int cv,int i) {…}                    //同前
void dfs(int cw,int cv,int i) {                          //回溯算法
    tot++;                                               //累计调用次数
    if (i>=n){                                           //到达一个叶子结点
        if (cw<=W && cv>bestv) {                         //找到一个满足条件的更优解,保存它
            bestv=cv;
            bestx=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);
        }
        double 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);
        }
    }
}
void knap(vector<Goods> g1,int W1) {                     //求 0/1 背包问题
    g=g1;
    W=W1;
    n=g.size();
    sort(g.begin(),g.end());                             //按 v/w 递减排序
    x=vector<int>(n,0);
    dfs(0,0,0);                                          //i 从 0 开始
    printf("最佳装填方案\n");
    for (int i=0;i<n;i++) {
        if (bestx[i]==1)
            printf(" 选取第%d 个物品\n",g[i].no);
    }
    printf(" 总重量=%d,总价值=%d\n",W,bestv);
}
```

对于表 2.1 中的 4 个物品，$W=6$ 时调用上述 knap()算法的求解结果如下：

```
最佳装填方案
    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
```

knap 算法分析：该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 $O(n\times 2^n)$。

5.9* 完全背包问题

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。

扫一扫 视频讲解

解法 1：与 0/1 背包问题不同，完全背包问题中的物品 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 的几种操作方式如下。

（1）不选择物品 i。

（2）当 cw+w[i]≤W 时，选择物品 i 一件，下一步继续选择物品 i。

（3）当 cw+w[i]≤W 时，选择物品 i 一件，下一步开始选择物品 i+1。

实际上与 5.2.2 节中求幂集的标准子集树算法 1 相比，这里仅增加了(2)操作，由于该操作后面又可以选择物品 i，从而满足每种物品可以挑选任意多件的条件。正是由于后面的物品可以挑选任意多件，所以无法采用求解 0/1 背包问题中的右剪支的操作(右剪支是针对不选择当前物品的剪支操作)。

例如，n=2，W=2，w={1,2}，v={2,5}，对应的搜索空间如图 5.21 所示，结点对应的状态是“(cw,cv,i)”，每个分支结点的 3 个分支分别进行上述 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。

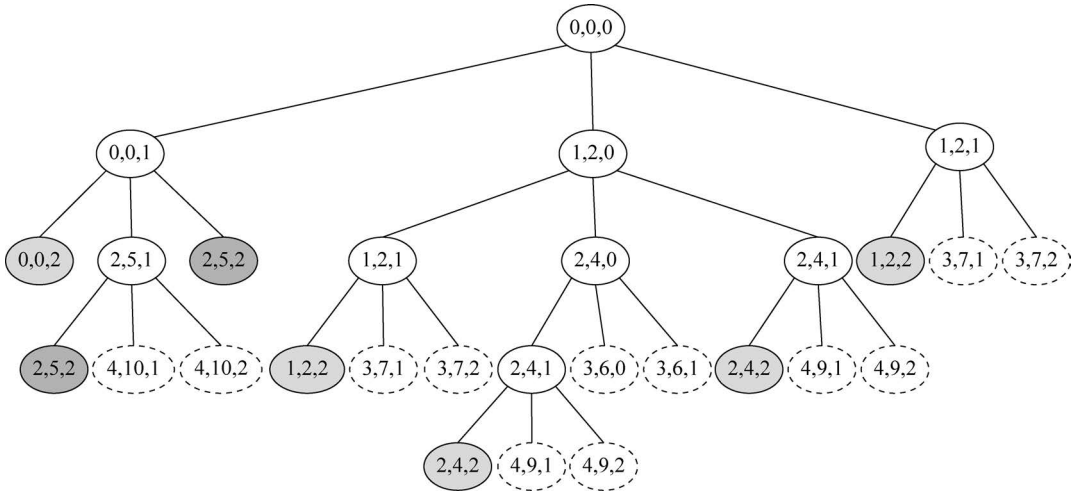


图 5.21 完全背包问题实例的搜索空间(1)

将 n、w、v 和 W 等表示求解问题的变量设置为全局变量，利用上述思路设计的回溯法算法如下：

```
int bestv=0;                                    //存放最大价值,初始为 0
void dfs(int cw,int cv,int i) {                 //回溯算法
    if(i>=n) {
        if(cw<=W && cv>bestv)                   //找到一个更优解
            bestv=cv;
    }
    else {
        dfs(cw,cv,i+1);                         //不选择物品 i
        if(cw+w[i]<=W)                          //剪支
            dfs(cw+w[i],cv+v[i],i);             //选择物品 i,然后继续选择物品 i
        if(cw+w[i]<=W)                          //剪支
            dfs(cw+w[i],cv+v[i],i+1);           //选择物品 i,然后选下一件
    }
}
void completeknap1() {                          //求完全背包问题
    dfs(0,0,0);
    printf("最大价值=%d\n",bestv);
}
```

解法 2：利用 5.2.2 节中求幂集的标准子集树算法 2 的思路，设解向量 $\boldsymbol{x}=\{x_0,x_1,\cdots,x_{m-1}\}$，解空间树中的第 i 层结点是为 x_i 在物品 $j\sim n-1$ 中选择适合的物品 j1（即置 x_i=j1），由于后面仍然可以选择物品 j1，所以置下一步（求 x_{i+1}）选择的物品仍然是 j1～$n-1$。这里只需要求最大价值，不必实际设计解向量 $\boldsymbol{x}$。对应的回溯算法如下：

```
int bestv=0;                                    //存放最大价值,初始为 0
void dfs(int cw,int cv,int j) {                 //回溯算法
    if(cw<=W && cv>bestv) {                     //找到一个更优解
        bestv=cv;
    }
    for (int j1=j;j1<n;j1++) {
        if(cw+w[j1]<=W) {                       //剪支
            dfs(cw+w[j1],cv+v[j1],j1);          //选择物品 j1,然后可以继续选择物品 j1
        }
    }
}
void completeknap2() {                          //求完全背包问题
    dfs(0,0,0);
    printf("最大价值=%d\n",bestv);
}
```

例如，$n=2$，$W=2$，$w=\{1,2\}$，$v=\{2,5\}$，对应的搜索空间如图 5.22 所示，结点对应的状态是“(cw,cv,j)”，其中深阴影结点是最优解结点，求出 bestv=5，对应的解向量 $\boldsymbol{x}=\{1\}$，表示选择物品 1 一次。

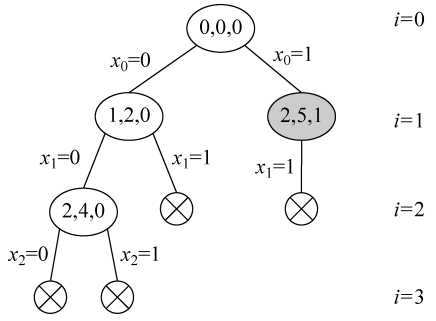


图 5.22 完全背包问题实例的搜索空间(2)

5.10 基于排列树的回溯算法框架

求全排列问题的解空间树是最典型的排列树，本节通过求全排列导出基于排列树的回溯算法框架，后面介绍相关算法设计示例。

5.10.1 求全排列

为了通用，将求全排列问题改为求含不同整数元素的数组的全排列。

问题描述：有一个含 n 个不同整数的数组 a，设计一个算法求其所有元素的全排列。例如，$a=\{1,2,3\}$，其全排列是(1,2,3)、(1,3,2)、(2,3,1)、(2,1,3)、(3,1,2)、(3,2,1)（输出顺序任意）。

扫一扫

视频讲解

解法 1：设 $a=\{a_0,\cdots,a_i,\cdots,a_{n-1}\}$，解向量为 $\boldsymbol{x}=\{x_0,\cdots,x_i,\cdots,x_{n-1}\}$，每个解

向量对应 a 的一个排列，显然 x_i 可以是 $a[0..n-1]$ 中的任意元素，即 n 选一，但所有 x_i 不能重复，为此设计一个 used 数组，used[i]表示 a_i 是否使用过。解空间树中的第 i 层结点用于确定 x_i 的值。对应的回溯算法如下：

```
vector<int> x;                                  //解向量
vector<int> used;                               //used[i]表示 a[i]是否使用过
int cnt=0;                                      //累计排列个数
void disp() {                                   //输出一个解
    printf(" %2d {",++cnt);
    for (int i=0;i<x.size()-1;i++)
        printf("%d,",x[i]);
    printf("%d}\n",x.back());
}
void dfs(vector<int> &a,int i) {                //回溯算法
    int n=a.size();
    if (i>=n) {
        disp();
    }
    else {
        for(int j=0;j<n;j++) {
            if(used[j]) continue;               //剪支:跳过已经使用过的 a[j]
            x[i]=a[j];
            used[j]=1;                          //选择 a[j]
            dfs(a,i+1);                         //转向解空间树的下一层
            used[j]=0;                          //回溯
            x[i]=0;
        }
    }
}
void perm1(vector<int> &a) {                    //求全排列算法 1
    int n=a.size();
    x=vector<int>(n);
    used=vector<int>(n,0);
    dfs(a,0);
}
```

perm1 算法分析：从表面上看调用的 dfs 算法中每个结点是 n 选一，实际上通过剪支操作使得解空间树中的根结点层（$i=0$）有一个结点，$i=1$ 层有 n 个结点，$i=2$ 层有 $n(n-1)$ 个结点，$i=3$ 层有 $n(n-1)(n-2)$ 个结点，以此类推，最后叶子结点层有 $n!$ 个结点，对应 $n!$ 个排列。设总结点个数为 $C(n)$，则：

$$\begin{aligned}C(n)&=1+n+n(n-1)+n(n-1)(n-2)+\cdots+n!\\&\approx n+n(n-1)+n(n-1)(n-2)+\cdots+n!\\&=n!\left(1+\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{(n-1)!}\right)=n!\left(\mathrm{e}-\frac{1}{n!}-\frac{1}{(n+1)!}-\cdots\right)\\&=n!\mathrm{e}-1=O(n!)\end{aligned}$$

叶子结点个数为 $n!$，也就是 $C(n)$ 和叶子结点个数同级，而每个叶子结点对应一个解，输出一个解的时间是 $O(n)$，所以算法的时间复杂度为 $O(n\times n!)$。

解法 2：改进解法 1，首先将 a 存放到 x 中，通过交换方式避免使用 used 数组，简单地说，设解向量为 $\boldsymbol{x}=\{x_0,\cdots,x_i,\cdots,x_{n-1}\}$，当已经生成解向量的 $\{x_0,\cdots,x_{i-1}\}$ 部分后，现在产生解向量的 $\{x_i,\cdots,x_{n-1}\}$ 部分，将 x_i 与其中的每一个元素交换，目的是让 x_i 取所有可能的元素值（由于 $x_0,\cdots,x_{i-1}$ 元素已经使用过，所以 x_i 不能取这些元素值），采用交换的方式也保证了 x_i 的所有元素不重复。对应的回溯算法如下：

```
vector<int> x;                          //解向量
int cnt=0;                              //累计排列个数
void disp() {                           //输出一个解
    printf(" %2d {",++cnt);
    for (int i=0;i<x.size()-1;i++)
        printf("%d,",x[i]);
    printf("%d}\n",x.back());
}
void dfs(int i) {                       //回溯算法
    int n=x.size();
    if (i>=n) {
        disp();
    }
    else {
        for(int j=i;j<n;j++) {
            swap(x[i],x[j]);            //交换 a[i]与 a[j]
            dfs(i+1);
            swap(x[i],x[j]);            //回溯:交换 a[i]与 a[j]
        }
    }
}
void perm2(vector<int> &a) {            //求全排列算法 2
    int n=a.size();
    x=vector<int>(n);
    for(int i=0;i<n;i++)x[i]=a[i];      //置 x=a
    dfs(0);
}
```

思考题： 在 dfs()中如果不执行第二个交换语句(即 swap(x[i],x[j]))会出现什么问题呢？为什么？

例如 $a=\{1,2,3\}$时，求全排列的过程如图 5.23 所示，它就是该问题的解空间树，根结点的层次 i 为 0，对于第 i 层的结点，其子树分别对应 $x[i]$位置选择 $x[i]$、$x[i+1]$、……、$x[n-1]$元素。树的高度为 $n+1$，叶子结点的层次是 n。

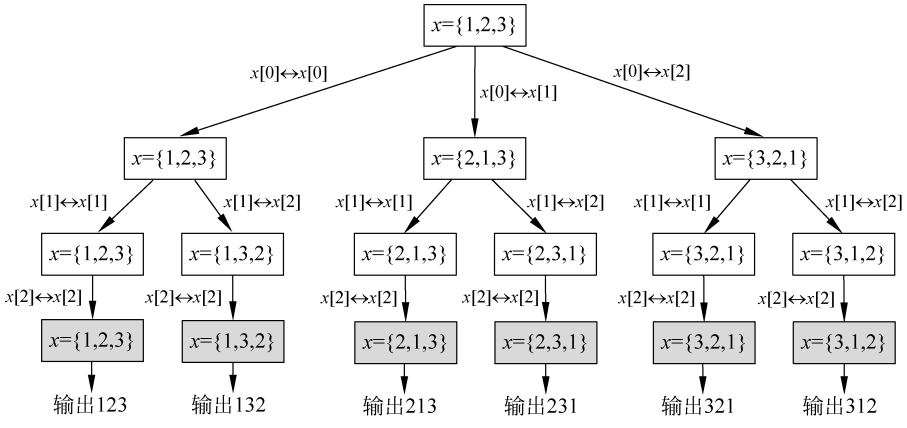


图 5.23 求 $a=\{1,2,3\}$的全排列的过程

perm2 算法分析： 分析同 perm1 算法，算法的时间复杂度为 $O(n\times n!)$。

说明： perm1 算法按字典序依次生成全排列，而 perm2 算法产生的全排列是无序的。

5.10.2 排列树回溯算法框架

设问题的解是一个 n 维向量$\{x_0,x_1,\cdots,x_{n-1}\}$，约束函数表示每个分量 x_i 应满足的

约束条件，记为 constraint(x_i)；限界函数记为 bound(x_i)。一般地，解空间为排列树的递归回溯框架如下：

```
int x[n];                                   //x 为解向量
void dfs(int i) {                           //求解排列树的递归框架
    if(i>=n)                                //搜索到叶子结点
        产生一个可能的解;
    else {
        for (j=i;j<=n;j++) {                //用 j 枚举 i 所有可能的路径
            ...                             //x[i]选择 x[j]的操作
            swap(x[i],x[j]);
            if (constraint(i) && bound(i))
                dfs(i+1);                   //满足约束条件和限界函数,进入下一层
            swap(x[i],x[j]);                //回溯: 恢复状态
            ...                             //回退到第 i 层结点的操作
        }
    }
}
```

同样的几点注意见解空间为子集树的回溯算法框架的说明。

5.11 n 皇后问题

扫一扫

视频讲解

问题描述：n 皇后问题的描述参见第 3 章中的 3.8 节。这里采用回溯法求解。

解 同样用整数数组 q 存放 n 皇后问题的求解结果，即 $q[i](0\leqslant i\leqslant n-1)$的值表示第 i 个皇后所在的列号，即该皇后放在$(i,q[i])$的位置上。

这里 q 相当于解向量，显然 $q[0..n-1]$的取值恰好是 $0\sim n-1$ 的某个排列，所以该问题转换为求全排列问题，对于每个排列判断是否为一个解，在采用回溯法时对应的解空间树是一棵排列树，根结点的层次为 0，用于确定 $q[0]$(即皇后 0 的列号)，层次为 i 的结点用于确定 $q[i]$(即皇后 i 的列号)，叶子结点的层次为 n，每个叶子结点对应一个解。

对应的基于排列树的回溯算法如下：

```
int q[MAXN];                                //存放 n 皇后问题的解(解向量)
int cnt=0;                                  //累计解个数
void disp(int n) {                          //输出一个解
    printf(" 第%d 个解:",++cnt);
    for (int i=0;i<n;i++)
        printf("(%d,%d) ",i,q[i]);
    printf("\n");
}
bool valid(int i,int j) {                   //测试(i,j)位置能否放置皇后
    if (i==0) return true;                  //皇后 0 总是可以放置的
    int k=0;
    while (k<i) {                           //k=1~i-1 是已放置了皇后的行
        if ((q[k]==j) || (abs(q[k]-j)==abs(i-k)))
            return false;
        k++;
    }
    return true;
}
void dfs(int n,int i) {                     //回溯算法
    if (i>=n)
        disp(n);                            //所有皇后放置结束,输出一个解
```

```
        else {
            for (int j=i;j<n;j++) {                    //在第 i 行上试探每一个列 j
                swap(q[i],q[j]);                       //皇后 i 放置在 q[j]列
                if(valid(i,q[i]))                      //剪支
                    dfs(n,i+1);
                swap(q[i],q[j]);                       //回溯
            }
        }
    }
    void queen(int n) {                                //求解 n 皇后问题
        for(int i=0;i<n;i++) q[i]=i;                   //初始化 q 为 0～n-1
        dfs(n,0);
    }
```

queen(n)算法分析：该算法对应的解空间树是一棵排列树，结点个数为 $O(n!)$，每个分支调用 place 算法的时间最多为 $O(n)$，所以算法的时间复杂度为 $O(n\times n!)$。

扫一扫 视频讲解

【例 5.6】 n 皇后问题Ⅱ(LintCode34★★)。给定一个整数 n($n\leqslant 10$)，设计一个算法求 n 皇后问题的解的数量。要求设计如下成员函数：

```
int totalNQueens(int n) { }
```

扫一扫 源程序

解 与前面的求解 n 皇后问题的思路完全相同，这里改为仅累计 n 皇后问题的解个数。

5.12 任务分配问题

扫一扫 视频讲解

问题描述：问题描述见第 3 章中的 3.9 节，这里采用回溯法求解。

解 n 个人和 n 个任务的编号均用 0～$n-1$ 表示，设计解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，以人为主进行搜索，即 x_i 表示人员 i 分配的任务编号($0\leqslant x_i\leqslant n-1$)，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，将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，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)。

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 $x[j]$(通过 swap($x[i],x[j]$)语句实现)，即置 $x[i]=x[j]$，cost+=$c[i][x[i]]$，此时部分解向量为 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),则一定会构成该路径的总成本下界。minsum 的计算公式如下:

$$\text{minsum}=\sum_{i1=i+1}^{n-1}\sum_{j1=0}^{n-1}\min_{x[j1]\notin P}\{c_{i1,x[j1]}\}$$

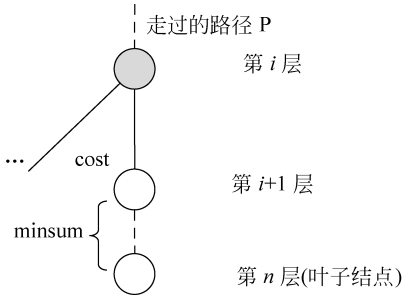


图 5.24 人员 i 安排任务 $x[j]$ 的情况

说明:minsum 的含义是在 c 中所有未分配任务的人员行($i+1\sim n-1$ 行)和未分配的任务表(used[x[j1]]=false)中每行求一个最小值,然后相加得到 minsum。

总成本下界 b=cost+minsum,如图 5.24 所示。显然如果 b≥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可能找到更优解,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。带剪支的排列树回溯算法如下:

```
int n;
vector<vector<int>> c;
vector<int> x;                          //解向量
vector<int> bestx;                      //最优解向量
int bestc;                              //最小成本
vector<bool> used;
int bound(int cost,int i) {             //求下界算法
    int minsum=0;
    for (int i1=i;i1<n;i1++) {          //求 c[i..n-1]行中的最小元素和
        int minc=INF;
        for (int j1=0;j1<n;j1++) {
            if (used[x[j1]]==false && c[i1][x[j1]]<minc)
                minc=c[i1][x[j1]];
        }
        minsum+=minc;
    }
    return cost+minsum;
}
void dfs(int cost,int i) {              //回溯算法
    if (i>=n) {                         //到达一个叶子结点
        if (cost<bestc) {               //比较求最优解
            bestc=cost;
            bestx=x;
        }
    }
    else {
        for (int j=i;j<n;j++) {         //为人员 i 试探任务 x[j]
            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
                dfs(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
            swap(x[i],x[j]);
        }
    }
}
```

```
void allocate(vector<vector<int>> &C) {          //求解任务分配问题
    c=C;
    n=c.size();
    x=vector<int>(n);
    for(int i=0;i<n;i++)                         //将 x[0..n-1]分别设置为 0 到 n-1
        x[i]=i;
    used=vector<bool>(n,false);
    bestc=INF;
    dfs(0,0);                                    //从人员 0 开始
    printf("最优分配方案\n");
    for (int k=0;k<n;k++)
        printf(" 人员%d 分配任务%d\n",k,bestx[k]);
    printf(" 总成本=%d\n",bestc);
}
```

对于表 3.2 所示的任务分配问题，调用上述算法的输出结果如下：

```
最优分配方案
    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
```

allocate 算法分析：该算法的解空间树是一棵排列树，结点个数为 $O(n!)$，剪支操作的时间为 $O(n^2)$，所以算法的时间复杂度为 $O(n^2 \times n!)$。

【例 5.7】 订单分配(LintCode1909★★)。问题描述参见例 3.8。这里采用回溯法求解。

解 与前面讨论的求解任务分配问题类似，仅将求最小成本改为求最大得分。

扫一扫 视频讲解

扫一扫 源程序

扫一扫 视频讲解

5.13 旅行商问题

问题描述：问题描述参见 3.10 节，这里采用回溯法求解。

解 同样城市道路图采用邻接矩阵 $\boldsymbol{A}$ 存储，起始点为 s，设 TSP 路径(即解向量) $\boldsymbol{x}=\{s,x_1,\cdots,x_{n-1},s\}$，如图 5.25 所示，$x_1 \sim x_{n-1}$ 只能是 $0 \sim n-1$ 中非 s 的顶点，对应的 TSP 路径长度 $=A[s][x_1]+A[x_1][x_2]+\cdots+A[x_{n-2}][x_{n-1}]+A[x_{n-1}][s]$。也就是说，$x_1 \sim x_{n-1}$ 是 $0 \sim n-1$(除了 s 顶点)的一个排列，从而该问题转换为解空间树为排列树的搜索问题。

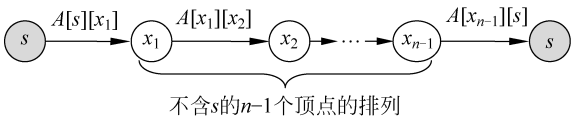


图 5.25 一条 TSP 路径

先将 s 作为 x_0，这里假设 $s=0$，再将其他非 s 的顶点的编号添加到 x 中，从 x_1 开始搜索，用 d 记录当前的路径长度，对应的解空间树如图 5.26 所示，对于层次为 i 的某个结点，j

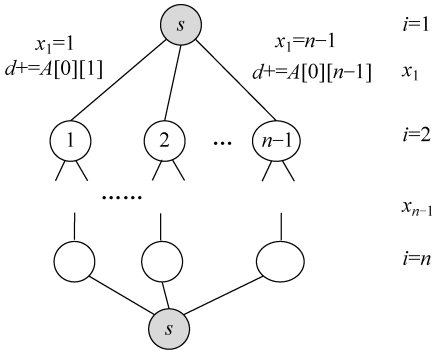


图 5.26 TSP 问题($s=0$)的解空间树

从 i 到 $n-1$ 循环，尝试 $x[i]$取值 $x[j]$。

(1) 判断是否有边：若路径上前一个顶点 $x[i-1]$到顶点 $x[j]$没有边，则跳过该 $x[j]$。

(2) 剪支：如果 $x[i]$取值 $x[j]$，则当前路径长度 $d=d+A[x[i-1]][x[j]]$，假如已经求出一个最优路径长度是 bestd，当 $d\geqslant$bestd 时跳过该 $x[j]$。

通过上述条件后则 $x[i]$取值为 $x[j]$(通过 swap($x[i]$,$x[j]$)实现)，然后继续搜索，当到达叶子结点时通过路径长度比较求最优路径 bestx 和长度 bestd，最后输出最优解。

对应的回溯算法如下：

```
vector<int> x;                                          //解向量(路径)
vector<int> bestx;                                      //保存最短路径
int d;                                                  //x 路径的长度
int bestd=INF;                                          //保存最短路径长度
void dfs(vector<vector<int>> &A,int s,int i) {          //回溯算法
    int n=A.size();
    if(i>=n) {                                          //到达一个叶子结点
        if(d+A[x[n-1]][s]<bestd) {                      //比较求最优解
            bestd=d+A[x[n-1]][s];                       //求 TSP 长度
            bestx=x;                                    //更新 bestx
            bestx.push_back(s);                         //末尾添加起始点
        }
    }
    else {
        for(int j=i;j<n;j++) {                          //试探 x[i]走到 x[j]的分支
            if (A[x[i-1]][x[j]]!=0 && A[x[i-1]][x[j]]!=INF) { //若 x[i-1]到 x[j]有边
                if(d+A[x[i-1]][x[j]]<bestd) {           //剪支
                    swap(x[i],x[j]);
                    d+=A[x[i-1]][x[i]];
                    dfs(A,s,i+1);
                    d-=A[x[i-1]][x[i]];
                    swap(x[i],x[j]);
                }
            }
        }
    }
}
void TSP(vector<vector<int>> &A,int s) {                //求解 TSP(起始点为 s)
    int n=A.size();
    x.push_back(s);                                     //添加起始点 s
    for(int i=0;i<n;i++) {                              //将非 s 的顶点添加到 x 中
        if(i!=s) x.push_back(i);
    }
    d=0;
    dfs(A,s,1);                                         //从 x[1]开始求解
    printf(" 最短路径: ");                              //输出最短路径
    for (int j=0;j<bestx.size();j++) {
        if(j==0)
            printf("%d",bestx[j]);
        else
            printf("->%d",bestx[j]);
```

```
    }
    printf("\n 路径长度: %d\n",bestd);
}
```

针对第 3 章中的图 3.14,设置起始点 $s=0$,调用上述算法的求解结果如下:

```
最短路径: 0->2->3->1->0
路径长度: 23
```

思考题: 为了提高回溯算法的时间性能,还可以采用哪些剪支方法?

TSP 算法分析: 在算法中求 $x_1 \sim x_{n-1}$ 共 $n-1$ 个元素的全排列,解空间中的结点个数为 $O((n-1)!)$,叶子结点的个数同级,到达叶子结点时进行路径长度的比较并执行 bestx=x 实现路径复制的时间为 $O(n)$,所以算法的时间复杂度为 $O(n\times(n-1)!)$,即 $O(n!)$。

【例 5.8】 旅行计划(LintCode1891★★)。问题描述参见第 3 章中的例 3.9,这里采用回溯法求解。

解 采用与前面用回溯法求解旅行商问题完全相同的思路,设置起始点 $s=0$。

扫一扫 视频讲解

扫一扫 源程序

扫一扫 自测题

5.14 练 习 题

1. 简述回溯算法中主要的剪支策略。

2. 简述回溯法中常见的两种类型的解空间树。

3. 鸡兔同笼问题是一个笼子里面有鸡和兔子若干只,数一数,共有 a 个头、b 条腿,求鸡和兔子各有多少只? 假设 $a=3$,$b=8$,画出对应的解空间树。

4. 考虑子集和问题,$n=3$,$a=\{1,3,2\}$,$t=3$,回答以下问题:

(1) 不考虑剪支,画出求解的搜索空间和解。

(2) 考虑左剪支(选择元素),画出求解的搜索空间和解。

(3) 考虑左剪支(选择元素)和右剪支(不选择元素),画出求解的搜索空间和解。

5. 考虑 n 皇后问题,其解空间树为由 1、2、……、n 构成的 $n!$ 种排列组成,现用回溯法求解,要求:

(1) 通过解搜索空间说明 $n=3$ 时是无解的。

(2) 给出剪支操作。

(3) 最坏情况下在解空间树上会生成多少个结点? 分析算法的时间复杂度。

6. 二叉树的所有路径(LintCode480★)。给定一棵二叉树,设计一个算法求出从根结点到叶子结点的所有路径。例如,对于如图 5.27 所示的二叉树,答案是{"1->2->5","1->3"}。要求设计如下成员函数:

```
vector<string> binaryTreePaths(TreeNode * root) { }
```

7. 二叉树的最大路径和Ⅱ(LintCode475★★)。给定一棵二叉树,设计一个算法找到二叉树的最大路径和,路径必须从根结点出发,路径可在任意结点结束,但至少包含一个结点(也就是根结点)。要求设计如下成员函数:

```
int maxPathSum2(TreeNode * root) {}
```

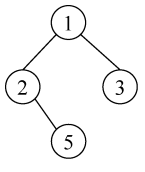


图 5.27 一棵二叉树

8. 求组合(LintCode152★★)。给定两个整数 n 和 k,设计一个算法求从 1～n 中选出 k 个数的所有可能的组合,对返回组合的顺序没有要求,但一个组合内的所有数字需要是升序排列的。例如,$n=4$,$k=2$,答案是{{1,2},{1,3},{1,4},{2,3},{2,4},{3,4}}。要求设计如下成员函数:

```
vector<vector<int>> combine(int n,int k) { }
```

9. 最小路径和Ⅱ(LintCode1582★★)。给出一个 $m\times n$ 的矩阵,每个点有一个正整数的权值,设计一个算法从($m-1$,0)位置走到(0,$n-1$)位置(可以走上、下、左、右 4 个方向),找到一条路径使得该路径所经过的权值和最小,返回最小权值和。要求设计如下成员函数:

```
int minPathSumII(vector<vector<int>> &matrix) {}
```

10. 递增子序列(LeetCode491★★)。给定一个含 n($1\leqslant n\leqslant 15$)个整数的数组 nums($100\leqslant$nums[i]$\leqslant 100$),找出并返回所有该数组中不同的递增子序列,递增子序列中至少有两个元素,可以按任意顺序返回答案。数组中可能含有重复元素,如果出现两个整数相等,也可以视作递增序列的一种特殊情况。例如,nums={4,6,7,7},答案是{{4,6},{4,6,7},{4,6,7,7},{4,7},{4,7,7},{6,7},{6,7,7},{7,7}}。要求设计如下成员函数:

```
vector<vector<int>> findSubsequences(vector<int>& nums) {}
```

11. 给表达式添加运算符(LeetCode282★★★)。给定一个长度为 n($1\leqslant n\leqslant 10$)的仅包含数字 0～9 的字符串 num 和一个目标值整数 target($-2^{31}\leqslant$target$\leqslant 2^{31}-1$),设计一个算法在 num 的数字之间添加二元运算符(不是一元)+、-或 *,返回所有能够得到 target 的表达式。注意所返回表达式中的运算数不应该包含前导零。例如,num="105",target=5,答案是{"1*0+5","10-5"}。要求设计如下成员函数:

```
vector<string> addOperators(string num,int target) { }
```

12. 求解马走棋问题。在 m 行 n 列的棋盘上有一个中国象棋中的马,马走“日”字且只能向右走。设计一个算法,求马从棋盘的左上角(1,1)走到右下角(m,n)的可行路径的条数。例如,$m=4$,$n=4$ 的答案是 2。

13. 设计一个回溯算法求迷宫中从入口 s 到出口 t 的最短路径及其长度。

14. 设计一个求解 n 皇后问题的所有解的迭代回溯算法。

15. 题目描述见第 3 章中 3.11 节的第 12 题,这里要求采用回溯法求解。

5.15 在线编程实验题

1. LintCode1353——根结点到叶子结点求和★★
2. LintCode802——数独★★★
3. LintCode135——数字组合★★
4. LintCode1915——举重★★★
5. LintCode680——分割字符串★★
6. LintCode136——分割回文串★★
7. LintCode816——旅行商问题★★★
8. LeetCode784——字母大小写全排列★★

9. LeetCode1079——活字印刷★★
10. LeetCode93——复原 IP 地址★★
11. LeetCode22——括号的生成★★
12. LeetCode89——格雷编码★★
13. LeetCode301——删除无效的括号★★★
14. POJ3050——跳房子
15. POJ1724——道路
16. POJ1699——最佳序列
17. POJ1564——求和
18. POJ2245——组合
19. POJ1321——棋盘问题
20. POJ2488——骑士之旅