

第5章 回溯法



5.1

本章知识结构



本章主要讨论回溯法的概念、回溯算法搜索解的过程、利用剪支函数提高搜索效率、基于子集树和排列树的算法框架及其经典应用示例,其知识结构如图 5.1 所示。

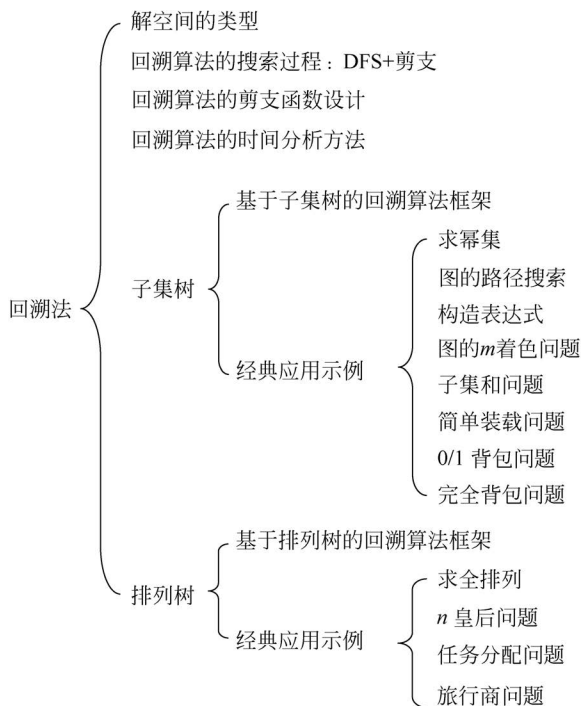


图 5.1 本章知识结构图

5.2

《教程》中的练习题及其参考答案

1. 简述回溯算法中主要的剪支策略。

答: 回溯算法中主要的剪支策略如下。

(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。

2. 简述回溯法中常见的两种类型的解空间树。

答: 回溯法中常见的两种类型的解空间树是子集树和排列树。

当给定的问题是从 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,这类子集树通常有 2^n 个叶子结点,遍历子集树需要 $O(2^n)$ 的时间。

当给定的问题是确定 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。这类排列树通常有 $n!$ 个叶子结点,遍历排列树需要 $O(n!)$ 的时间。

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。

答: 用 x 和 y 分别表示鸡和兔子的数目,显然鸡的数目最多为 $\min(a, b/2)$, 兔的数目最多为 $\min(a, b/4)$, 也就是说 x 的取值范围是 $0 \sim 3, y$ 的取值范围是 $0 \sim 2$ 。对应的解空间树如图 5.2 所示,共 17 个结点。

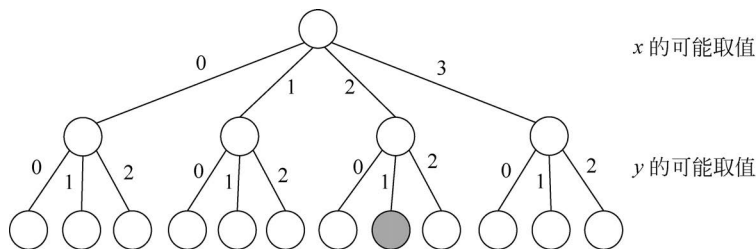


图 5.2 $a=3, b=8$ 的解空间树

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。

答: (1) 对应的搜索空间如图 5.3 所示, 图中结点为“ (cs, i) ”, 其中 cs 表示当前选择的元素的和, i 为结点的层次。最后找到的两个解是 $\{1, 2\}$ 和 $\{3\}$ 。

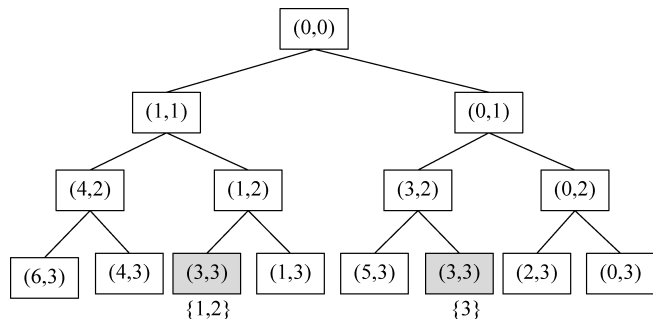


图 5.3 子集和问题的搜索空间(1)

(2) 对应的搜索空间如图 5.4 所示, 图中结点所包含数字的含义同(1), 最后找到的两个解是 $\{1, 2\}$ 和 $\{3\}$ 。

(3) 对应的搜索空间如图 5.5 所示, 图中结点为“ (cs, rs, i) ”, 其中 cs 表示当前选择的元素的和, rs 为剩余元素的和(含当前元素), i 为结点的层次。最后找到的两个解是 $\{1, 2\}$ 和 $\{3\}$ 。

5. 考虑 n 皇后问题, 其解空间树由 $1, 2, \dots, n$ 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。

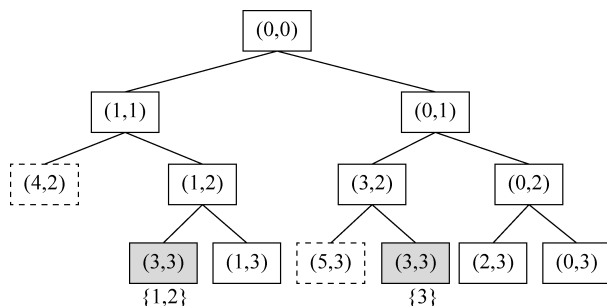


图 5.4 子集和问题的搜索空间(2)

(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。

答: (1) $n=3$ 时的解搜索空间如图 5.6 所示, 不能得到任何叶子结点, 所以无解。

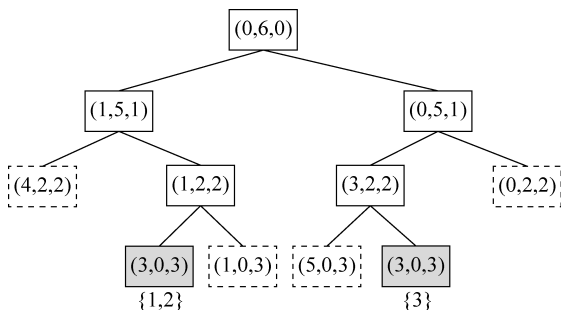


图 5.5 子集和问题的搜索空间(3)

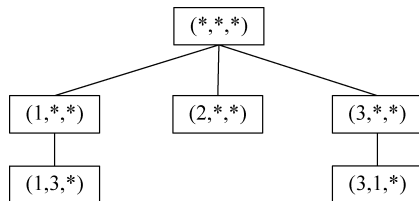


图 5.6 3 皇后问题的解搜索空间

(2) 剪支操作是任何两个皇后不能同行、同列和同对角线。

(3) 最坏情况下解空间树中根结点层($i=0$)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,设结点总数为 $C(n)$:

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

每个结点进行冲突判断的时间为 $O(n)$, 所以算法的时间复杂度为 $O(n \times n!)$ 。

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 5.7 所示的二叉树, 答案是 $\{ "1 \rightarrow 2 \rightarrow 5", "1 \rightarrow 3" \}$ 。要求设计如下成员函数:

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

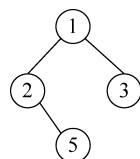


图 5.7 一棵二叉树

解: 将二叉树看成一棵解空间树, 从根结点开始搜索所有的结点, 用解向量 x 表示从根结点到当前结点的路径串, 当每次到达一个叶子结点时将 x 添加到 ans 中, 最后返回 ans 即可。对应的回溯算法如下:

```

class Solution {
    vector< string> ans;
    string x;
public:
    vector< string> binaryTreePaths(TreeNode * root) {
        if(root == NULL) return {};
        x = to_string(root->val);
        dfs(root);
        return ans;
    }
    void dfs(TreeNode * root) {          //回溯算法
        if(root->left == NULL && root->right == NULL)
            ans.push_back(x);
        else {
            if(root->left != NULL) {
                string tmp = x;
                x += "-" + to_string(root->left->val);
                dfs(root->left);
                x = tmp;                //回溯
            }
            if(root->right != NULL) {
                string tmp = x;
                x += "-" + to_string(root->right->val);
                dfs(root->right);
                x = tmp;                //回溯
            }
        }
    }
};

```

上述程序提交后通过,执行用时为 42ms,内存消耗为 5.36MB。

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) {}
```

解: 将二叉树看成一棵解空间树,用 ans 表示答案,从根结点开始搜索,用 cursum 记录路径上的结点值之和,当到达任意一个结点时,如果 $\text{cursum} > \text{ans}$,则置 $\text{ans} = \text{cursum}$,最后返回 ans。对应的回溯算法如下:

```

class Solution {
    int ans;                                //存放答案
public:
    int maxPathSum2(TreeNode * root) {
        if(root == NULL) return 0;
        ans = root->val;
        int cursum = root->val;
        dfs(cursum, root);
        return ans;
    }
    void dfs(int cursum, TreeNode* root) {    //回溯算法
        if(root == NULL) return;
        if(cursum > ans) ans = cursum;
        if(root->left != NULL) {

```

```

        cursum += root->left->val;
        dfs(cursum, root->left);
        cursum -= root->left->val;
    }
    if(root->right!= NULL) {
        cursum += root->right->val;
        dfs(cursum, root->right);
        cursum -= root->right->val;
    }
}
};

```

8. 求组合(LintCode152★★)。给定两个整数 n 和 k , 设计一个算法求从 $1 \sim n$ 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) { }
```

解: 采用《教程》5.2 节中求幂集的解法 2 的思路, 设置解向量 $x, x[i]$ 表示一个组合中位置为 i 的整数, 按从 i 到 n 的顺序试探, 所以得到的解中数字是按升序排列的。用 cnt 表示选择的整数的个数, 当 $cnt=k$ 时对应一个解, 将所有解添加到 ans 中, 最后返回 ans 即可。对应的回溯程序如下:

```

class Solution {
    vector<vector<int>> ans;
    vector<int> x;
public:
    vector<vector<int>> combine(int n, int k) {
        dfs(0, n, k, 1);
        return ans;
    }
    void dfs(int cnt, int n, int k, int i) {           //回溯算法
        if(cnt == k) {
            ans.push_back(x);
        }
        else {
            for(int j = i; j <= n; j++) {
                x.push_back(j);
                dfs(cnt + 1, n, k, j + 1);
                x.pop_back();
            }
        }
    }
};

```

上述程序提交后通过, 执行用时为 61ms, 内存消耗为 5.46MB。

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) { }
```

解法 1: 采用《教程》5.3 节中图路径搜索的思路, 用 $cursum$ 累计路径的长度 (路径上所

经过点的权值和),为了避免路径上的点重复,设置 visited 数组(初始元素均为 0),一旦走到一个顶点便将对应位置的 visited 置为 1,每一步只能走 visited 为 0 的点。从起始点($m-1,0$)出发进行搜索,当搜索到终点($0,n-1$)时将较小的 cursum 存放到 ans 中。最后返回 ans 即可。对应的回溯算法如下:

```
class Solution {
    const int INF = 0x3f3f3f3f;
    int dx[4] = {0,0,1,-1};           //水平方向的偏移量
    int dy[4] = {1,-1,0,0};           //垂直方向的偏移量
    int ans;                           //存放答案
    vector<vector<int>> visited;
    vector<vector<int>> A;
public:
    int minPathSumII(vector<vector<int>> &matrix) {
        A = matrix;
        int m = A.size();
        int n = A[0].size();
        visited = vector<vector<int>>(m,vector<int>(n,0));
        int x = m-1,y = 0;
        visited[x][y] = 1;
        int cursum = A[x][y];
        ans = INF;
        dfs(m,n,cursum,x,y);
        return ans;
    }
    void dfs(int m,int n,int cursum,int x,int y) { //回溯算法
        if (x == 0 && y == n-1) { //找到终点
            ans = min(ans,cursum);
        }
        else {
            for(int di = 0;di < 4;di++) {
                int nx = x + dx[di];
                int ny = y + dy[di];
                if(nx < 0 || nx >= m || ny < 0 || ny >= n)
                    continue;
                if(visited[nx][ny] == 1)
                    continue;
                visited[nx][ny] = 1;
                cursum += A[nx][ny];
                if(cursum < ans) //剪支
                    dfs(m,n,cursum,nx,ny);
                cursum -= A[nx][ny];
                visited[nx][ny] = 0;
            }
        }
    }
};
```

上述程序提交后通过,执行用时为 183ms,内存消耗为 5.4MB。

解法 2: 上述算法中的剪支操作是仅扩展 $\text{cursum} < \text{ans}$ 的路径(用一条部分路径长度与一条完整路径长度比较),性能较低,可以将 $\text{visited}[x][y]$ 改为存放从起始点到 (x,y) 位置的最小路径长度(初始时所有元素置为 ∞),当试探到 (nx,ny) 时,只有当前路径长度 $\text{cursum} + A[nx][ny]$ 小于或等于 $\text{visited}[nx][ny]$ 才将当前路径继续走下去,否则终止该路径。由于

A 中的元素均为正整数,这样比较一定会避免路径上出现重复点。对应的回溯算法如下:

```
class Solution {
    const int INF = 0x3f3f3f3f;
    int dx[4] = {0,0,1,-1};           //水平方向的偏移量
    int dy[4] = {1,-1,0,0};           //垂直方向的偏移量
    vector<vector<int>> visited;
    vector<vector<int>> A;
public:
    int minPathSumII(vector<vector<int>> &matrix) {
        A = matrix;
        int m = A.size();
        int n = A[0].size();
        visited = vector<vector<int>>(m,vector<int>(n,INF));
        int x = m-1,y = 0;
        visited[x][y] = A[x][y];
        int cursum = A[x][y];
        dfs(m,n,cursum,x,y);
        return visited[0][n-1];
    }
    void dfs(int m,int n,int cursum,int x,int y) { //回溯算法
        visited[x][y] = cursum;
        for(int di = 0;di < 4;di++) {
            int nx = x + dx[di];
            int ny = y + dy[di];
            if(nx < 0 || nx >= m || ny < 0 || ny >= n)
                continue;
            if(cursum + A[nx][ny] > visited[nx][ny])
                continue;
            cursum += A[nx][ny];
            dfs(m,n,cursum,nx,ny);
            cursum -= A[nx][ny];
        }
    }
};
```

上述程序提交后通过,执行用时为 41ms,内存消耗为 5.45MB。

10. 递增子序列(LeetCode491★★)。给定一个含 n ($1 \leq n \leq 15$) 个整数的数组 `nums` ($100 \leq \text{nums}[i] \leq 100$),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) {}
```

解: 用 x 表示解向量(一个满足题目要求的递增子序列),`ans` 存放最后答案。用 i 遍历 `nums`,分为以下两类情况:

(1) 当 x 为空时,有将 `nums[i]` 添加到 x 中和不将 `nums[i]` 添加到 x 中两种选择。

(2) 当 x 非空时,若 `nums[i] ≥ x.back()`,将 `nums[i]` 添加到 x 中,在 `nums[i] ≠ x.back()` 时不将 `nums[i]` 添加到 x 中,因为 `nums[i] > x.back()` 时只能将 `nums[i]` 添加到 x 中,否则会得不到一些递增子序列。

对应的代码如下:

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> x;
    vector<vector<int>> findSubsequences(vector<int> & nums) {
        dfs(nums, 0);
        return ans;
    }
    void dfs(vector<int> & nums, int i) {           //回溯算法
        if (i == nums.size()) {
            if (x.size() >= 2)
                ans.push_back(x);
        }
        else {
            if (x.size() == 0) {
                x.push_back(nums[i]);
                dfs(nums, i + 1);
                x.pop_back();
                dfs(nums, i + 1);
            }
            else {
                if (nums[i] >= x.back()) {
                    x.push_back(nums[i]);
                    dfs(nums, i + 1);
                    x.pop_back();
                }
                if (nums[i] != x.back()) {
                    dfs(nums, i + 1);
                }
            }
        }
    }
};
```

上述程序提交后通过,执行用时为 20ms,内存消耗为 19.3MB。如果改为当 x 非空时,若 $\text{nums}[i] \geq x.\text{back}()$,做将 $\text{nums}[i]$ 添加到 x 中和不将 $\text{nums}[i]$ 添加到 x 中两种选择,也就是将上述 dfs 中最下面的两个 if 语句改为如下:

```
if (nums[i] >= x.back()) {
    x.push_back(nums[i]);           //将 nums[i] 添加到 x 中
    dfs(nums, i + 1);
    x.pop_back();
    dfs(nums, i + 1);              //不将 nums[i] 添加到 x 中
}
```

则会出现重复的递增子序列,例如 $\text{nums} = \{4, 6, 7, 7\}$ 时,输出结果是 $\{\{4, 6, 7, 7\}, \{4, 6, 7\}, \{4, 6, 7\}, \{4, 6\}, \{4, 7, 7\}, \{4, 7\}, \{4, 7\}, \{6, 7, 7\}, \{6, 7\}, \{6, 7\}, \{7, 7\}\}$,从中看出 $\{4, 6\}$ 等重复项。

另外也可增加一个 last 参数表示子序列 x 中最后的元素,当 x 为空时 last 为 INT_MIN。对应的代码如下:

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> x;
```

```

vector<vector<int>>> findSubsequences(vector<int> & nums) {
    int last = INT_MIN;
    dfs(nums, last, 0);
    return ans;
}

void dfs(vector<int> & nums, int last, int i) {    //回溯算法
    if (i == nums.size()) {
        if (x.size() >= 2)
            ans.push_back(x);
    }
    else {
        if (nums[i] >= last) {
            x.push_back(nums[i]);
            dfs(nums, nums[i], i + 1);
            x.pop_back();
        }
        if (nums[i] != last) {
            dfs(nums, last, i + 1);
        }
    }
}
};

```

上述程序提交后通过,执行用时为 32ms,内存消耗为 19.4MB。

11. 给表达式添加运算符(LeetCode282★★★)。给定一个长度为 n ($1 \leq n \leq 10$) 的仅包含数字 0~9 的字符串 num 和一个目标值整数 target ($-2^{31} \leq \text{target} \leq 2^{31} - 1$), 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) { }
```

解: 采用回溯法求解, 用 x 表示值为 target 的表达式, 用 ans 存放这样的全部表达式。在算法设计中需要注意以下两个方面:

(1) 产生的表达式中运算数可以是连续的一个或者多个数字, 当分隔点为 num[i] 时, 通过 num.substr(i, j-i+1) 取出后面的数字子串 curs, 对应的值为 curd, 由于运算数不应该包含前导零, 所以当 $j \neq i$ && num[i] == '0' 成立时返回。

(2) 当分隔点 num[i] 后面的运算数产生后, 分隔点处可以取 '+', '-', 或者 '*', 即三选一。若当前求出的表达式的值为 cursum, 分隔点前面的一个运算数是 pred:

① 若取 '+', 执行 $x += '+' + \text{curs}$, $\text{cursum} += \text{curd}$, $\text{prev} = \text{curd}$, 递归处理对应的子问题, 回溯时恢复修改的参数。

② 若取 '-', 执行 $x += '-' + \text{curs}$, $\text{cursum} -= \text{curd}$, $\text{prev} = \text{curd}$, 递归处理对应的子问题, 回溯时恢复修改的参数。

③ 若取 '*', 执行 $x += '*' + \text{curs}$, $\text{cursum} = \text{cursum} - \text{pred} + \text{pred} * \text{curd}$, $\text{prev} = \text{pred} * \text{curd}$, 例如 $1+2 \times 3 \times 4$, 假设当前在 3 和 4 之间取 '*', 则 cursum 应减去 $2 \times 3 = 6$, 然后加上 $2 \times 3 \times 4 = 24$ 。再递归处理对应的子问题, 回溯时恢复修改的参数。

例如, num="105", target=5, 采用回溯法的过程如图 5.8 所示, 图中用 '.' 表示分隔点, 虚框表示为前导零的结点, 带阴影结点对应一个解。

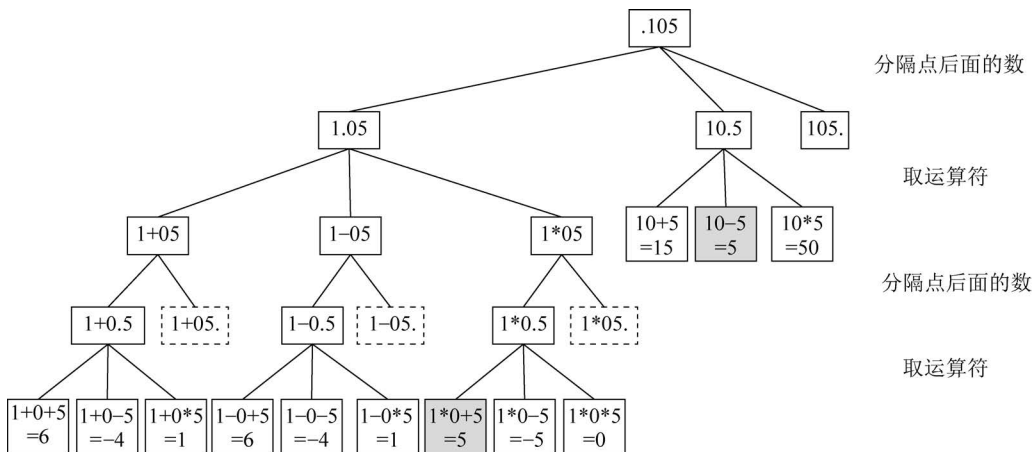


图 5.8 num="105",target=5 的求解过程

对应的程序如下：

```
class Solution
{
public:
    string num;
    int target;
    vector<string> ans;
    string x;
    vector<string> addOperators(string num, int target) {
        this->num = num;
        this->target = target;
        dfs(0,0,0);
        return ans;
    }
    void dfs(int i,long long cursum,long long pred) { //回溯算法
        if (i==num.size()) {
            if (cursum==target)
                ans.push_back(x);
        }
        else {
            int oldlen = x.size();
            for (int j = i; j < num.size(); j++) {
                if (j!=i && num[i] == '0') //为前导零时返回
                    return;
                string curs = num.substr(i, j - i + 1);
                long long curd = stoll(curs);
                if (i==0) {
                    x += curs;
                    dfs(j + 1, cursum + curd, curd);
                    x.resize(oldlen); //回溯(恢复 x)
                }
                else {
                    x += '+' + curs;
                    dfs(j + 1, cursum + curd, curd);
                    x.resize(oldlen); //回溯(恢复 x)
                    x += '-' + curs;
                    dfs(j + 1, cursum - curd, -curd);
                    x.resize(oldlen); //回溯(恢复 x)
                }
            }
        }
    }
};
```

```

        x += ' ' + curs;
        dfs(j + 1, cursum - pred + pred * curd, pred * curd);
        x.resize(oldlen);           //回溯(恢复 x)
    }
}
};

```

上述程序提交后通过,执行用时为 96ms,内存消耗为 14.5MB。

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。

解: 在 m 行 n 列的棋盘上,若马的位置为 (x,y) ,其满足要求的 4 种走法如图 5.9 所示。

采用以下增量数组表示:

```

int dx[4] = {1, 2, 2, 1};
int dy[4] = {-2, -1, 1, 2};

```

用二维数组 visited 表示棋盘上的对应位置是否已经访问,从 $(1,1)$ 位置出发进行搜索,到达 (m,n) 位置时解个数 ans 增 1,搜索完毕返回 ans 即可。对应的回溯算法如下:

	$x+1, y-2$	
		$x+2, y-1$
x, y		
		$x+2, y-1$
	$x+1, y+2$	

图 5.9 马的 4 种走法

```

int dx[4] = {1, 2, 2, 1};
int dy[4] = {-2, -1, 1, 2};
int ans;                                     //路径的条数
vector<vector<int>>> visited;
void dfs(int m, int n, int x, int y) {      //回溯算法
    if (x == m && y == n) {                //找到目标位置
        ans++;                             //路径的条数增加 1
    }
    else {
        for (int di = 0; di < 4; di++) {    //试探所有可走路径
            int nx = x + dx[di];            //求出从(x,y)走到的位置(x1,y1)
            int ny = y + dy[di];
            if (nx < 1 || nx > m || ny < 1 || ny > n) //跳过越界位置
                continue;
            if (visited[nx][ny] == 1)         //仅考虑没有访问的位置
                continue;
            visited[nx][ny] = 1;
            dfs(m, n, nx, ny);
            visited[nx][ny] = 0;             //回溯
        }
    }
}

void solve(int m, int n) {                  //求解算法
    ans = 0;
    visited = vector<vector<int>>>(m + 1, vector<int>(n + 1, 0));
    visited[1][1] = 1;
    dfs(m, n, 1, 1);
    printf("%d * %d 象棋中马的路径条数 = %d\n", m, n, ans);
}

```

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。

解: 用 $bestx$ 和 $bestlen$ 保存最短路径及其长度。从 $curp$ 方格(初始为 s)出发进行搜索,当前路径及其长度分别保存在 x 和 len 中:

(1) 若 $curp=t$,说明找到一条迷宫路径,通过比较路径长度将最优解保存在 $bestx$ 和 $bestlen$ 中。

(2) 否则试探每个方位 di (取值为 $0\sim 3$),求出相邻方格 (nx,ny) ,跳过超界、已经访问或者为障碍物的位置,走到 (nx,ny) 位置并继续搜索,再从 (nx,ny) 位置回溯。最后输出 $bestx$ 和 $bestlen$ 。

对应的回溯算法如下:

```
int dx[4] = { -1, 1, 0, 0 };
int dy[4] = { 0, 0, -1, 1 };
struct Box {                                     //方格类型
    int x,y;
    Box() {}
    Box(int x, int y):x(x),y(y) {}
};
int visited[MAXM][MAXN];
vector<vector<int>>> A;
int m,n;
Box t;
int bestlen;                                     //最短路径长度
vector<Box> bestx;                               //最短路径
void dfs(Box&curp, vector<Box> &x, int len) {     //回溯算法
    if(curp.x == t.x && curp.y == t.y) {
        if(len < bestlen) {
            bestlen = len;
            bestx = x;
        }
    }
    else {
        for(int di = 0; di < 4; di++) {
            int nx = curp.x + dx[di];
            int ny = curp.y + dy[di];
            if(nx < 0 || nx >= m || ny < 0 || ny >= n || visited[nx][ny] == 1)
                continue;
            if(A[nx][ny] == 1) continue;
            if(len < bestlen) {                   //剪支
                visited[nx][ny] = 1;
                Box nextp = Box(nx, ny);
                x.push_back(nextp);
                len++;
                dfs(nextp, x, len);
                len--;
                x.pop_back();
                visited[nx][ny] = 0;
            }
        }
    }
}

void maze(vector<vector<int>>> mg, Box&start, Box&goal) { //求解算法
    A = mg;
    m = A.size();
```

```

n = A[0].size();
Box curp = start;
visited[start.x][start.y] = 1;
t = goal;
memset(visited, 0, sizeof(visited));
vector<Box> x;
x.push_back(curp);
bestlen = INF;
dfs(curp, x, 0);
printf("最短路径: ");
for(int i = 0; i < bestx.size(); i++)
    printf("[ %d, %d] ", bestx[i].x, bestx[i].y);
printf("\n 最短路径长度: %d\n", bestlen);
}

```

14. 设计一个求解 n 皇后问题的所有解的迭代回溯算法。

解: n 皇后问题的迭代回溯算法设计如下。

(1) 用数组 $q[]$ 存放皇后的列位置, $(i, q[i])$ 表示第 i 个皇后的放置位置, n 皇后问题的一个解是 $(1, q[1]), (2, q[2]), \dots, (n, q[n])$, 数组 q 的 0 下标不用。

(2) 先放置第 1 个皇后, 然后依 2、3、 $\dots$ 、 n 的次序放置其他皇后, 当第 n 个皇后放置好后产生一个解。为了找出所有解, 此时算法还不能结束, 继续试探第 n 个皇后的下一个位置。

(3) 放置第 $i (i < n)$ 个皇后后, 接着放置第 $i+1$ 个皇后, 在试探第 $i+1$ 个皇后的位置时都是从第 1 列开始的。

(4) 当第 i 个皇后试探了所有列都不能放置时, 回溯到第 $i-1$ 个皇后, 此时与第 $i-1$ 个皇后的位置 $(i-1, q[i-1])$ 有关, 如果第 $i-1$ 个皇后的列号小于 n , 即 $q[i-1] < n$, 则将其移到下一列, 继续试探; 否则再回溯到第 $i-2$ 个皇后, 以此类推。

(5) 若第 1 个皇后的所有位置回溯完毕, 则算法结束。

(6) 放置的第 i 个皇后应与前面已经放置的 $i-1$ 个皇后不发生冲突。

对应的迭代回溯算法如下:

```

int q[MAXN]; //存放各皇后所在的列号, 为全局变量
int cnt = 0; //累计解的个数
void dispasolution(int n) { //输出一个解
    printf(" 第 %d 个解:", ++cnt);
    for (int i = 1; i <= n; i++)
        printf("( %d, %d) ", i, q[i]);
    printf("\n");
}
bool place(int i) { //测试第 i 行的 q[i] 列上能否摆放皇后
    if (i == 1) return true;
    int k = 1;
    while (k < i) { //k = 1 ~ i-1 是已放置了皇后的行
        if ((q[k] == q[i]) || (abs(q[k] - q[i]) == abs(k - i)))
            return false;
        k++;
    }
    return true;
}
void Queens(int n) { //求解 n 皇后问题
    int i = 1; //i 表示当前行, 也表示放置第 i 个皇后

```

```

q[i] = 0; //q[i]是当前列,从0列(即开头)开始试探
while (i >= 1) { //重复试探
    q[i]++;
    while (q[i] <= n && !place(i)) //试探一个位置(i, q[i])
        q[i]++;
    if (q[i] <= n) { //为第i个皇后找到了一个合适的位置(i, q[i])
        if (i == n) //若放置了所有皇后,输出一个解
            dispasolution(n);
        else { //皇后没有放置完
            i++; //转向下一行,即开始下一个皇后的放置
            q[i] = 0; //每次放一个新皇后都从该行的首列进行试探
        }
    }
    else i--; //若第i个皇后找不到合适的位置,则回溯到上一个皇后
}
}

```

15. 题目描述见《教程》第3.11节的第12题,这里要求采用回溯法求解。

解: 本问题与旅行商问题类似,不同之处是不必回到起点。先将边数组 tuple 转换为邻接矩阵 A ,为了简单,将城市编号由 $1 \sim n$ 改为 $0 \sim n-1$,这样起点变成了顶点0,并且不必考虑路径中最后一个顶点到顶点0的道路。然后采用《教程》第5章中5.13节求TSP问题的回溯法求出最短路径长度 bestd 并返回。对应的回溯法算法如下:

```

const int INF = 0x3f3f3f3f; //表示∞
vector<int> x; //解向量(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
        bestd = min(bestd, d); //通过比较求最优解
    }
    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]);
                }
            }
        }
    }
}

int mincost(int n, vector<vector<int>> &tuple) { //求解算法
    if (n == 1) return 0;
    vector<vector<int>> A(n, vector<int>(n, INF)); //邻接矩阵
    for (int i = 0; i < tuple.size(); i++) {
        int a = tuple[i][0] - 1;
        int b = tuple[i][1] - 1;
        int w = tuple[i][2];
        A[a][b] = A[b][a] = w;
    }
}

```

```
int s = 0;
x.push_back(s);
for(int i = 0; i < n; i++){
    if(i != s) x.push_back(i);
}
d = 0;
dfs(A, s, 1);
return bestd;
}
```

//添加起始顶点 s
//将非 s 的顶点添加到 x 中

5.3

补充练习题及其参考答案



扫一扫



在线资源

5.3.1 单项选择题及其参考答案

1. 以下问题中最适合用回溯法求解的是_____。
A. 迷宫问题 B. 二分查找 C. 求水仙花数 D. 求最大公约数
2. 以下问题中最适合用回溯法求解的是_____。
A. 汉诺塔问题 B. 8 皇后问题 C. 求素数 D. 破解密码
3. 回溯法是在问题的解空间中按_____策略从根结点出发搜索的。
A. 广度优先 B. 活结点优先 C. 扩展结点优先 D. 深度优先
4. 回溯法解题的步骤中通常不包含_____。
A. 确定结点的扩展规则
B. 针对给定的问题定义其解空间
C. 枚举所有可能的解,并通过搜索到的解优化解空间结构
D. 采用深度优先搜索方法搜索解空间树
5. 关于回溯法,以下叙述中不正确的是_____。
A. 回溯法有通用解题法之称,可以系统地搜索一个问题的所有解或任意解
B. 回溯法是一种既带有系统性又带有跳跃性的搜索算法
C. 回溯法需要借助队列来保存从根结点到当前扩展结点的路径
D. 回溯法在生成解空间中的任一结点时,先判断该结点是否可能包含问题的解,如果肯定不包含,则跳过对以该结点为根的子树的搜索,逐层向祖先结点回溯
6. 回溯法的效率不依赖于下列_____。
A. 确定解空间的时间 B. 满足显约束的值的个数
C. 计算约束函数的时间 D. 计算限界函数的时间
7. 以下关于回溯法的叙述中正确的是_____。
A. 即使问题的解存在,回溯法也不一定能找到问题的解
B. 回溯法找到的问题的解不一定是最优解
C. 回溯法不能找到问题的全部解
D. 回溯法无法避免求出的问题的解重复
8. 以下关于回溯法的叙述中错误的是_____。

- A. 以深度优先方式搜索解空间树
B. 可用约束函数剪去得不到可行解的子树
C. 可用限界函数剪去得不到最优解的子树
D. 即使在最坏情况下回溯法的性能也好于穷举法
9. 关于使用回溯法求解 0/1 背包问题,以下说法中正确的是_____。
A. 使用限界函数剪去得不到更优解的左子树(装入该物品)。
B. 使用限界函数剪去得不到更优解的右子树(不装入该物品)。
C. 使用约束函数剪去不合理的右子树(不装入该物品)。
D. 使用限界函数剪去不合理的左子树(装入该物品)。
10. 通常排列树的解空间树中有_____个叶子结点。
A. $n!$ B. n^2 C. $n+1$ D. $n(n+1)/2$
11. 对于含有 n 个元素的子集树问题(每个元素二选一),最坏情况下解空间树的叶子结点个数是_____。
A. $n!$ B. 2^n C. $2^{n+1}-1$ D. 2^{n-1}
12. 一般来说,回溯算法的解空间树不会是_____。
A. 有序树 B. 子集树 C. 排列树 D. 无序树
13. 用回溯法求解 0/1 背包问题时最坏时间复杂度是_____。
A. $O(n)$ B. $O(n \log_2 n)$ C. $O(2^n)$ D. $O(n^2)$
14. 用回溯法求解旅行商问题时的解空间是_____。
A. 子集树 B. 排列树
C. 深度优先生成树 D. 广度优先生成树
15. 求中国象棋中马从一个位置到另外一个位置的所有走法,采用回溯法求解时对应的解空间是_____。
A. 子集树 B. 排列树
C. 深度优先生成树 D. 广度优先生成树
16. n 个人排队在一台机器上做某个任务,每个人等待的时间不同,完成他的任务的时间是不同的,求完成这 n 个任务的最少时间,采用回溯法求解时对应的解空间是_____。
A. 子集树 B. 排列树
C. 深度优先生成树 D. 广度优先生成树



扫一扫

在线资源

5.3.2 问答题及其参考答案

- 简述回溯法和递归的区别。
- 在采用回溯法求解问题时,通常设计解向量为 $x = (x_0, x_1, \dots, x_{n-1})$,是不是说同一个问题的所有解向量的长度是固定的?
- 在用回溯法求解 0/1 背包问题时,该问题的解空间树是何种类型? 在用回溯法求解旅行商问题时,该问题的解空间树是何种类型?
- 对于递增序列 $a = \{1, 2, 3, 4, 5\}$,采用《教程》中 5.10.1 节的回溯法求全排列时,以 1、2 开头的排列一定最先出现吗? 为什么?
- 对于如图 5.10 所示的连通图,使用回溯算法来求解 3-着色问题,回答以下问题:

(1) 给出解向量的形式,指出解空间树的类型。

(2) 描述剪支操作。

(3) 画出找到一个解的搜索空间,并给出这个解。

6. 以下是求解 n 皇后问题的回溯算法, q 数组存放皇后的列号, $\text{valid}(i,j)$ 的功能是判断是否可以在 (i,j) 位置放置皇后 i ,指出算法中的错误并改正。

```
void dfs(int n,int i) {
    if (i >= n)
        disp(n);
    else {
        for (int j = i; j < n; j++) {
            if(valid(i,q[i])) {
                swap(q[i],q[j]);
                dfs(n, i + 1);
            }
            swap(q[i],q[j]);
        }
    }
}
```

//回溯算法

//所有皇后放置结束,输出一个解

//在第 i 行上试探每一个 j 列

//剪支

//皇后 i 放置在 $q[j]$ 列

//回溯

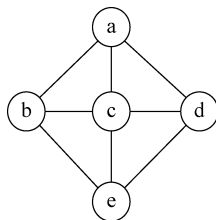


图 5.10 一个连通图

5.3.3 算法设计题及其参考答案

1. 二叉树的路径和(LintCode376★)。给定一棵二叉树,找出所有路径中各结点相加总和等于给定目标值的路径。一个有效的路径指的是从根结点到叶子结点的路径。要求设计如下成员函数:

```
vector<vector<int>> binaryTreePathSum(TreeNode * root, int target) {}
```

解: 将二叉树看成一棵解空间树,从根结点开始搜索,解向量 x 记录路径上的结点, cursum 记录路径上的结点值之和,当到达叶子结点时,如果 $\text{cursum} = \text{target}$,则将 x 添加到答案 ans 中,最后返回 ans 。对应的回溯算法如下:

```
class Solution {
    vector<vector<int>> ans;           //存放答案
    vector<int> x;                     //解向量
public:
    vector<vector<int>> binaryTreePathSum(TreeNode * root, int target) {
        if(root == NULL) return ans;
        x.push_back(root->val);
        int cursum = root->val;
        dfs(cursum, target, root);
        return ans;
    }
    void dfs(int cursum, int target, TreeNode * root) { //回溯算法
        if (root->left == NULL && root->right == NULL) {
            if(cursum == target)
                ans.push_back(x);
        }
        else {
            if(root->left != NULL) {
                x.push_back(root->left->val);
                cursum += root->left->val;
            }
            if(root->right != NULL) {
                x.push_back(root->right->val);
                cursum += root->right->val;
            }
            dfs(cursum, target, root);
            x.pop_back();
            cursum -= x.back();
        }
    }
}
```

```

        dfs(cursum, target, root->left);
        cursum -= root->left->val;
        x.pop_back();
    }
    if(root->right!= NULL) {
        x.push_back(root->right->val);
        cursum += root->right->val;
        dfs(cursum, target, root->right);
        cursum -= root->right->val;
        x.pop_back();
    }
}
};

```

2. 第 k 个排列(LintCode88★★)。给定 n 和 k , 设计一个算法求 $1 \sim n$ 的全排列中字典序的第 k 个排列。要求设计如下成员函数:

```
string getPermutation(int n, int k) {}
```

解: 利用《教程》5.10.1 节的 perm1 算法的思路, 增加 cnt 计数, x 作为解向量存放一个排列, 每次找到一个排列时执行 cnt++, 当 cnt= k 时将 x 存放到 ans 中, 并置 flag 为 true, 终止对其他路径的搜索。最后返回 ans。对应的回溯算法如下:

```

class Solution {
    string ans;                //存放第 k 个排列
    int cnt;                   //计数
    bool flag;                 //是否找到
    string x;                  //解向量
    vector<bool> used;          //used[i]表示 a[i]是否使用过
public:
    string getPermutation(int n, int k) {
        used = vector<bool>(n+1, false);
        x = string(n, '0');
        flag = false;
        cnt = 1;
        dfs(n, k, 0);
        return ans;
    }
    void dfs(int n, int k, int i) {                //回溯算法
        if (i >= n) {
            if(cnt == k) {
                flag = true;
                ans = x;
            }
            cnt++;
        }
        else if(!flag) {
            for(int j = 1; j <= n; j++) {
                if(used[j]) continue;
                x[i] = '0' + j;
                used[j] = true;
                dfs(n, k, i+1);
            }
        }
    }
}

```

```

        used[j] = false;
        x[i] = '0';
    }
}
};

```

3. 设计一个算法求解这样的子集和问题：给定含 n 个正整数的数组 a ，从中选出若干整数，使它们的和恰好为 t ，如果找到任意一种解，返回 true，否则返回 false。

解法 1：利用《教程》5.6 节的求子集和问题的思路，用 flag 表示子集和问题是否有解，初始设置为 false，一旦找到一个解置 flag 为 true。含左、右剪支的回溯算法如下：

```

bool flag; //子集和问题是否有解
void dfs(vector<int> &a, int t, int cs, int rs, int i) { //回溯算法
    if (i >= a.size()) { //到达一个叶子结点
        if (cs == t) flag = true; //找到一个满足条件的解
    }
    else if (!flag) { //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
            dfs(a, t, cs + a[i], rs, i + 1);
        }
        if (cs + rs >= t) { //右孩子结点剪支
            dfs(a, t, cs, rs, i + 1);
        }
        rs += a[i]; //恢复剩余整数和(回溯)
    }
}

bool solve(vector<int> &a, int t) { //判断子集和问题是否有解
    int rw = 0;
    for (int j = 0; j < a.size(); j++) //求 a 中元素的和 rw
        rw += a[j];
    flag = false;
    dfs(a, t, 0, rw, 0); //i 从 0 开始
    return flag;
}

```

解法 2：利用《教程》5.2.2 节中求幂集的解法 2 的思路求解，同样用 flag 表示子集和问题是否有解，初始设置为 false，一旦找到一个解置 flag 为 true。对应的回溯算法如下：

```

bool flag; //子集和问题是否有解
void dfs(vector<int> &a, int t, int cs, int i) { //回溯算法
    if (cs == t) {
        flag = true;
    }
    if (!flag) {
        for (int j = i; j < a.size(); j++) {
            cs += a[j];
            dfs(a, t, cs, j + 1);
            cs -= a[j];
        }
    }
}

bool solve(vector<int> &a, int t) { //判断子集和问题是否有解
    flag = false;
}

```

```

        dfs(a, t, 0, 0);                                //i 从 0 开始
        return flag;
    }

```

4. 设计一个算法求解这样的子集和问题：给定含 n 个正整数的数组 a ，从中选出若干整数，使它们的和恰好为 t ，假设该问题至少有一个解，当有多个解时求选出的整数个数最少的一个解。

解：利用《教程》5.6 节的求子集和问题的思路，用 x 存放当前解， $bestx$ 存放最优解（初始时置其长度为 n ），当找到一个解 x 时，将较小长度的 x 存放到 $bestx$ 中，最后返回 $bestx$ 。含左、右剪支的回溯算法如下：

```

vector<int> x;
vector<int> bestx;
void dfs(vector<int> &a, int t, int cs, int rs, int i) { //回溯算法
    if (i >= a.size()) {                                //到达一个叶子结点
        if (cs == t && x.size() < bestx.size())
            bestx = x;
    }
    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.push_back(a[i]);
            dfs(a, t, cs + a[i], rs, i + 1);
            x.pop_back();
        }
        if (cs + rs >= t) {                             //右孩子结点剪支
            dfs(a, t, cs, rs, i + 1);
        }
        rs += a[i];                                     //恢复剩余整数和(回溯)
    }
}

vector<int> solve(vector<int> &a, int t) {             //判断子集和问题是否有解
    int rw = 0;
    for (int j = 0; j < a.size(); j++)                //求 a 中元素的和 rw
        rw += a[j];
    bestx = vector<int>(a.size());                    //初始化 bestx 的长度为 n
    dfs(a, t, 0, rw, 0);                              //i 从 0 开始
    return bestx;
}

```

5. 求有重复元素的排列问题(LintCode16★★)。设有一个含 n 个整数的数组 a ，其中可能含有重复的元素，求这些元素的所有不同排列。例如 $a = \{1, 1, 2\}$ ，输出结果是 $\{1, 1, 2\}, \{1, 2, 1\}, \{2, 1, 1\}$ 。

解：在用回溯法求全排列的基础上增加对元素的重复性判断。例如，对于 $a = \{1, 1, 2\}$ ，不判断重复性时输出结果是 $\{1, 1, 2\}, \{1, 2, 1\}, \{1, 1, 2\}, \{1, 2, 1\}, \{2, 1, 1\}, \{2, 1, 1\}$ ，共 6 个排列，其中有 3 个是重复的。重复性判断是这样的：当扩展 $a[i]$ 时仅选取在 $a[i..j-1]$ 中没有出现的元素 $a[j]$ ，将其与 $a[i]$ 交换，如果 $a[j]$ 出现在 $a[i..j-1]$ 中，则对应的排列已经在前面求出了，如果再这样做会产生重复的排列。基于排列树的回溯算法如下：

```

class Solution {
    vector<vector<int>> ans;                            //存放答案
public:

```

```

vector<vector<int>>> permuteUnique(vector<int> &nums) {
    dfs(nums,0);
    return ans;
}

bool ok(vector<int> &a,int i,int j) {           //ok()用于判断重复元素
    if (j>i) {
        for(int k=i;k<j;k++) {
            if (a[k]==a[j])
                return false;
        }
    }
    return true;
}

void dfs(vector<int> &a,int i) {                //求有重复元素的排列问题
    int n=a.size();
    if (i==n) {
        ans.push_back(a);
    }
    else {
        for (int j=i;j<n;j++) {
            if (ok(a,i,j)) {                    //选取 a[i..j-1]中没有出现的元素 a[j]
                swap(a[i],a[j]);
                dfs(a,i+1);
                swap(a[i],a[j]);
            }
        }
    }
}
};

```

6. 设计一个回溯算法求从 $1 \sim n$ 的 n 个整数中取出 m 个元素的排列,要求每个元素最多只能取一次。例如, $n=3,m=2$ 时的输出结果是 $\{1,2\},\{1,3\},\{2,1\},\{2,3\},\{3,1\},\{3,2\}$ 。

解: 设解向量 $x=\{x_1,x_2,\dots,x_m\}$, x_i 取 $1 \sim n$ 中任意非重复的整数, i 从 1 开始,当 $i>m$ 时到达一个叶子结点,输出一个排列。采用 used 数组避免重复,used[j]=false 表示没有选择整数 j ,used[j]=true 表示已经选择整数 j 。对应的回溯算法如下:

```

vector<int> x;                                //x[1..m]中存放一个排列
vector<bool> used;
void dfs(int n,int m,int i){                  //回溯算法
    if (i>m) {
        for (int j=1;j<=m;j++)
            printf(" %d",x[j]);              //输出一个排列
        printf("\n");
    }
    else {
        for (int j=1;j<=n;j++) {
            if (!used[j]) {
                used[j]=true;                 //修改 used[i]
                x[i]=j;                       //x[i]选择 j
                dfs(n,m,i+1);
                x[j]=0;
                used[j]=false;                //回溯:恢复 used[i]
            }
        }
    }
}

```

```

}
void solve(int n,int m) {                                //求 1~n 中 m 个元素的全排列
    x = vector<int>(m+1);
    used = vector<bool>(n,false);
    dfs(n,m,1);                                          //i 从 1 开始
}

```

7. 对于 n 皇后问题,有人认为当 n 为偶数时,其解具有对称性,即 n 皇后问题的解个数恰好为 $n/2$ 皇后问题的解个数的两倍,这个结论正确吗? 请编写回溯法程序对 $n=4,6,8,10$ 的情况进行验证。

解: 这个结论不正确。验证程序如下:

```

int q[MAXN];                                            //存放 n 皇后问题的解(解向量)
int cnt;                                              //累计解个数
bool valid(int i,int j) {                             //测试(i,j)位置能否放置皇后
    if (i == 0) return true;                         //第一个皇后总是可以放置
    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)                                       //所有皇后放置结束
        cnt++;
    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]);                          //第 i 个皇后放置在 q[j] 列
            if(valid(i,q[i]))                          //剪支
                dfs(n,i+1);
            swap(q[i],q[j]);                          //回溯
        }
    }
}
int queens(int n) {                                   //求 n 皇后问题的解个数
    for(int i = 0; i < n; i++)                       //初始化 q 为 0~n-1
        q[i] = i;
    cnt = 0;
    dfs(n,0);
    return cnt;
}
bool solve() {                                        //验证算法
    bool flag = true;
    for (int n = 4; n <= 10; n += 2) {
        if (queens(n) != 2 * queens(n/2)) {
            flag = false;
            break;
        }
    }
    return flag;
}

```

8. 给定一个无向图,由指定的起点前往指定的终点,途中经过所有其他顶点且只经过一次,这称为哈密顿路径,闭合的哈密顿路径称为哈密顿回路(Hamiltonian cycle)。设计一

个回溯算法求无向图的所有哈密顿回路。

解法 1: 假设无向图有 n 个顶点(顶点编号为 $0 \sim n-1$), 采用邻接矩阵数组 $A(0/1$ 矩阵)存放, 求从顶点 s 出发回到顶点 s 的哈密顿回路。如同求解旅行商问题, 采用《教程》5.10.1 节中解法 1 的思路, 对应的回溯算法如下:

```
vector<int> x; //解向量
vector<int> used; //顶点访问标记
int cnt; //累计哈密顿回路的数目
void disp(int n,int s) { //输出一个解路径
    printf(" ( %d) ",cnt++);
    for (int i = 0;i < n;i++)
        printf(" %d->",x[i]);
    printf(" %d\n",s);
}
void dfs(vector<vector<int>> &A,int s,int curlen,int i) { //回溯算法
    int n = A.size();
    if(curlen == n-1) {
        if(A[x.back()][s] == 1)
            disp(n,s);
    }
    else {
        for(int j = 0;j < n;j++) {
            if(A[i][j] == 0) continue;
            if(used[j] == 1) continue;
            x.push_back(j);
            used[j] = 1;
            dfs(A,s,curlen+1,j);
            used[j] = 0;
            x.pop_back();
        }
    }
}
void Hamiltonian(vector<vector<int>> &A,int s){ //求从顶点 s 出发的哈密顿回路
    int n = A.size();
    x.clear();
    x.push_back(s);
    used = vector<int>(n,0);
    used[s] = 1;
    printf("从顶点 %d 出发的哈密顿回路:\n",s);
    dfs(A,s,0,s);
}
```

解法 2: 采用《教程》5.10.1 节中解法 2 的思路。对应的回溯算法如下:

```
vector<int> x; //解向量
int cnt; //累计哈密顿回路的数目
void disp(int n,int s) { //输出一个解路径
    printf(" ( %d) ",cnt++);
    for (int i = 0;i < n;i++)
        printf(" %d->",x[i]);
    printf(" %d\n",s);
}
void dfs(vector<vector<int>> &A,int s,int i) { //回溯算法
    int n = A.size();
    if (i >= n) {
        if(A[x[n-1]][s] == 1)
```

```

        disp(n, s);
    }
    else {
        for (int j = i; j < n; j++) {
            swap(x[i], x[j]);
            if (A[x[i-1]][x[i]] == 1)           //剪支
                dfs(A, s, i+1);
            swap(x[i], x[j]);                   //回溯
        }
    }
}

void Hamiltonian(vector<vector<int>> &A, int s){ //求从顶点 s 出发的哈密顿回路
    int n = A.size();
    x.clear();
    x.push_back(s);
    for(int i = 0; i < n; i++) {
        if(i != s) x.push_back(i);
    }
    printf("从顶点 %d 出发的哈密顿回路:\n", s);
    dfs(A, s, 1);
}

```

9. 求解满足方程的解。设计一个算法求出所有满足 $a * b - c * d + e = 1$ 方程的 a 、 b 、 c 、 d 、 e ，其中所有变量的取值范围为 $1 \sim 5$ ，并且均不相同。

解：本题相当于求出 $1 \sim 5$ 的满足方程要求的所有排列。采用解空间为排列树的框架，对应的回溯算法如下：

```

int x[5];                                     //解向量
int n = 5;
void disp(int x[]){                           //输出一个解
    printf(" %d * %d - %d * %d + %d = 1\n", x[0], x[1], x[2], x[3], x[4]);
}
void dfs(int i) {                             //回溯算法
    if (i == n) {                             //到达叶子结点
        if (x[0] * x[1] - x[2] * x[3] - x[4] == 1)
            disp(x);
    }
    else {
        for (int j = i; j < n; j++) {
            swap(x[i], x[j]);
            dfs(i+1);
            swap(x[i], x[j]);
        }
    }
}

void solve() {                                //求解算法
    for (int j = 0; j < n; j++)
        x[j] = j+1;
    dfs(0);
}

```

10. 求解最小重量机器设计问题I。设某一机器由 n 个部件组成，部件的编号为 $1 \sim n$ ，每一种部件都可以从 m 个供应商处购得，供应商的编号为 $1 \sim m$ 。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量， c_{ij} 是相应的价格。对于给定的机器部件重量和机器部件价格，设计一个算法计算总价格不超过 cost 的最小重量机器设计，可以在同一个供应商处购得多个部件。

例如, $n=3, m=3, \text{cost}=7, w=\{\{1,2,3\}, \{3,2,1\}, \{2,3,2\}\}, c=\{\{1,2,3\}, \{5,4,2\}, \{2,1,2\}\}$, 答案是部件 1、2、3 分别选择供应商 1、3 和 1, 最小重量为 4。

解: 采用基于子集树的回溯法求解, 由于可以在同一个供应商处购得多个部件, 所以每个部件有 m 个选择方案, 对应的解空间是一个 m 叉树的子集数。将 n, m, cost, w 和 c 设置为全局变量, 对应的回溯算法如下:

```
vector<int> x;
vector<int> bestx;
int bestw;
void dfs(int cw, int cc, int i) {
    if(i >= n) {
        if(cw < bestw) {
            bestw = cw;
            bestx = x;
        }
    }
    else {
        for(int j = 0; j < m; j++) {
            if(cc + c[i][j] > cost) continue;
            if(cw + w[i][j] > bestw) continue;
            x[i] = j;
            cc += c[i][j];
            cw += w[i][j];
            dfs(cw, cc, i + 1);
            cc -= c[i][j];
            cw -= w[i][j];
            x[i] = -1;
        }
    }
}

void solve() {
    int cw = 0, cc = 0;
    x = vector<int>(n, -1);
    printf("求解结果\n");
    bestw = INF;
    dfs(cw, cc, 0);
    for(int i = 0; i < n; i++)
        printf("    部件 %d 选择供应商 %d\n", i + 1, bestx[i] + 1);
    printf("    最小重量 = %d\n", bestw);
}
```

//回溯算法
//搜索到叶子结点
//通过比较产生最优解
//当前最小重量

//试探每一个供应商
//剪支
//剪支
//部件 i 选择供应商 j

//cc 回溯
//cw 回溯

//求解算法

11. 求解最小重量机器设计问题II。设某一机器由 n 个部件组成, 部件的编号为 $1 \sim n$, 共有 m 个供应商, 供应商的编号为 $1 \sim m (m \geq n)$ 。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量, c_{ij} 是相应的价格。对于给定的机器部件重量和机器部件价格, 设计一个算法计算总价格不超过 cost 的最小重量机器设计, 要求在同一个供应商处最多只能购得一个部件。例如, $n=3, m=3, \text{cost}=7, w=\{\{1,2,3\}, \{3,2,1\}, \{2,3,2\}\}, c=\{\{1,2,3\}, \{5,4,2\}, \{2,1,2\}\}$, 答案是部件 1、2、3 分别选择供应商 1、2 和 3, 最小重量为 5。

解: 采用回溯法求解, 解法类似算法设计题 10, 但这里要求在同一个供应商处最多只能购得一个部件, 所以设置一个 used 数组 (初始时所有元素为 false), 一旦为部件分配了供应商 j , 则置 $\text{used}[j] = \text{true}$ 。对应的回溯算法如下:

```
vector<int> x;
```

```

vector<int> bestx;
int bestw;
vector<bool> used;
void dfs(int cw,int cc,int i) {
    if(i>=n){
        if (cw<bestw) {
            bestw = cw;
            bestx = x;
        }
    }
    else {
        for(int j = 0;j<m;j++) {
            if(used[j]) continue;
            if(cc + c[i][j]> cost)
                continue;
            if(cw + w[i][j]> bestw)
                continue;
            x[i] = j;
            used[j] = true;
            cc += c[i][j];
            cw += w[i][j];
            dfs(cw,cc,i + 1);
            cc -= c[i][j];
            cw -= w[i][j];
            used[j] = false;
            x[i] = -1;
        }
    }
}

void solve() {
    int cw = 0,cc = 0;
    x = vector<int>(n, -1);
    used = vector<bool>(m,false);
    printf("求解结果\n");
    bestw = INF;
    dfs(cw,cc,0);
    for(int i = 0;i<n;i++)
        printf("    部件 %d 选择供应商 %d\n",i + 1,bestx[i] + 1);
    printf("    最小重量 = %d\n",bestw);
}

```

//供应商是否已经使用
//回溯算法
//搜索到叶子结点
//通过比较产生最优解
//当前最小重量

//试探每一个供应商
//供应商 j 已经使用过
//剪支

//剪支

//部件 i 选择供应商 j

//cc 回溯
//cw 回溯

//求解算法

12. 求解最大团问题。一个无向图 G 中含顶点个数最多的完全子图称为最大团。给定一个采用邻接矩阵 A (0/1 矩阵) 存储的无向图, 设计一个算法求其中最大团的顶点数。例如, $A = \{\{0,1,0,0,0\}, \{1,0,1,1,0\}, \{0,1,0,1,1\}, \{0,1,1,0,1\}, \{0,0,1,1,1,0\}\}$, 其中最大团由 3 个顶点(即顶点 1、2、3 或者 2、3、4) 构成, 答案为 3。

解: 用解向量 x 表示当前最大团, $x[i]=1$ 表示当前团包含顶点 i , cn 表示当前团的顶点数, 用 $bestn$ 表示最大团的顶点数。从顶点 0 出发进行搜索, 若当前顶点 i 与 x 中的所有顶点相连, 则选择将顶点 i 加入当前团中(对应左子树); 否则不选择顶点 i 进入右子树, 采用的剪支方式是当前团的顶点数 cn + 剩余的顶点数 $n - i + 1 \geq bestn$ 。当到达叶子结点时通过比较求 $bestn$ (初始值为 0)。对应的回溯算法如下:

```

vector<vector<int>> A;
int n;
vector<int> x;

```

//邻接矩阵
//解向量

```

int cn;
int bestn;
void dfs(int cn,int i){
    if (i >= n) {
        bestn = max(bestn,cn);
    }
    else {
        bool complete = true;
        for (int j = 0;j < i;j++) {
            if (x[j] && A[i][j] == 0) {
                complete = false;
                break;
            }
        }
        if (complete) {
            x[i] = 1;
            dfs(cn + 1,i + 1);
            x[i] = 0;
        }
        if (cn + n - i + 1 >= bestn) {
            x[i] = 0;
            dfs(cn,i + 1);
        }
    }
}

void solve(vector<vector<int>>> &a){
    A = a;
    n = A.size();
    x = vector<int>(n,0);
    cn = 0;
    dfs(cn,0);
    printf("最大团中的顶点数 = %d\n",bestn);
}

```

//当前解的顶点数
 //最大团的顶点数
 //回溯算法
 //到达叶子结点

 //检查顶点 i 与当前团的相连关系

 //顶点 i 与顶点 j 不相连

 //全相连,进入左子树
 //选中顶点 i

 //回溯

 //剪支(右子树)
 //不选中顶点 i

 //求最大团问题

 //当前团中的顶点数