第5章 走不下去就回退——回溯法

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

（1）掌握问题解空间的结构和深度优先搜索过程。

（2）掌握回溯法的原理和算法框架。

（3）掌握剪支函数（约束函数和限界函数）设计的一般方法。

（4）掌握各种回溯法经典算法的设计过程和算法分析方法。

（5）综合运用回溯法解决一些复杂的实际问题。

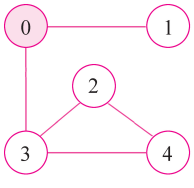
5.1 回溯法概述

5.1.1 问题的解空间

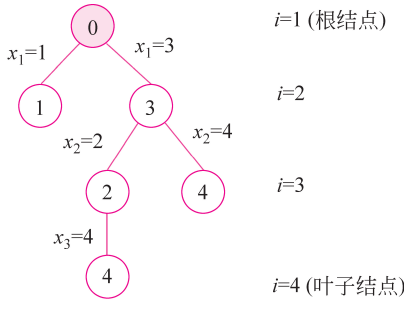
先看看求解问题的类型，通常求解问题分为两种类型，一种类型是给定一个约束函数，需要求所有满足约束条件的解，称为求**所有解类型**。例如鸡兔同笼问题中，所有鸡兔头数为 a，所有腿数为 b，求所有头数为 a、腿数为 b 的鸡兔数，设鸡兔数分别为 x 和 y，则约束函数是 $x+y=a$，$2x+4y=b$。另外一种类型是除了约束条件外还包含目标函数，最后是求使目标函数最大或者最小的最优解，称为求**最优解类型**。例如鸡兔同笼问题中，求所有鸡兔头数为 a、所有腿数为 b 并且鸡最少的解，这就是一个求最优解问题，除了前面的约束函数外还包含目标函数 $\min(x)$。这两类问题本质上是相同的，因为只有求出所有解再按目标函数进行比较才能求出最优解。

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

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



(a) 一个连通图



(b) 解空间

图 5.1 一个连通图及其问题的解空间

径，属于求最优解问题，同样需要求从顶点 0 到顶点 4 的所有路径，通过对路径长度的比较得到一条最短路径是 $x=\{0,3,4\}$。

归纳起来，解空间的一般结构如图 5.2 所示，根结点（为第 0 层）的每个分支对应分量 x_0 的一个取值（或者说 x_0 的一个决策），若 x_0 的候选集为 $S_0=\{v_{0,1},\cdots,v_{0,a}\}$，即根结点的子树个数为 $|S_0|$，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，解空间的高度为 $n+1$。从中看出第 i 层的结点对应 x_i 的各种选择，从根结点到每个叶子结点有一条路径，路径上的每个分支对应一个分量的取值，这是理解解空间的关键。

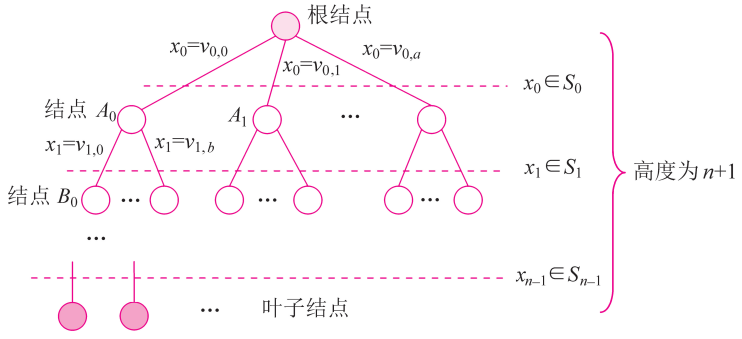


图 5.2 解空间的一般结构

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

5.1.2 什么是回溯法

从前面的讨论看出问题的解包含在解空间中，剩下的问题就是在解空间中搜索满足约束条件的解。所谓回溯法就是在解空间中采用深度优先搜索方法从根结点出发搜索解，与树的遍历类似，当搜索到某个叶子结点时对应一个可能解，如果同时又满足约束条件则该可能解是一个**可行解**。所以一个可行解就是从根结点到对应叶子结点的路径上所有分支的取值，例如一个可行解为 $(a_0,a_1,\cdots,a_{n-1})$，其图示如图 5.3 所示，在解空间中搜索到可行解的部分称为搜索空间。简单地说，回溯法采用深度优先搜索方法寻找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没有全部生成的结点，同时也成为当前的扩展结点，所谓**扩展结点**是指正在产生孩子结点的结点。在当前扩展结点处沿着纵深方向移至一个新结点，这个新结点又成为新的活结点，并成为当前扩展结点。如果在当前扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点，所谓**死结点**是指其所有子结点均已产生的结点，此时应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。

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

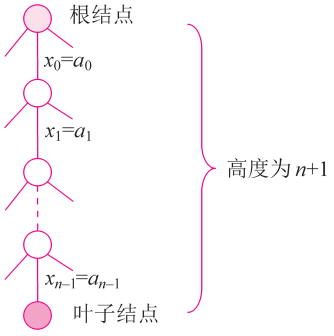


图 5.3　求解的搜索空间

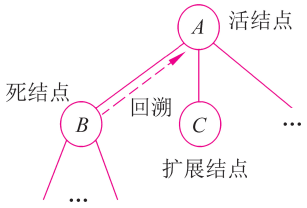


图 5.4　回溯过程

从上看出，求问题解的过程就是在解空间中搜索满足约束条件和目标函数的解。所以搜索算法设计的关键点有 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。例如，求图中从顶点 s 到顶点 t 的路径时，其扩展十分简单，就是从一个顶点找所有相邻顶点。

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

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

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

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

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

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

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

① 针对给定的问题确定其解空间，其中一定包含所求问题的解。

② 确定结点的扩展规则。

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

5.1.3 回溯法算法的时间分析

通常以回溯法的解空间中的结点个数作为算法的时间分析依据。假设解空间树共有 $n+1$ 层(根结点为第 0 层,叶子结点为第 n 层),第 1 层有 m_0 个结点,每个结点有 m_1 个子结点,则第 2 层有 m_0m_1 个结点,同理,第 3 层有 $m_0m_1m_2$ 个结点,以此类推,第 n 层有 $m_0m_1\cdots m_{n-1}$ 个结点,则采用回溯法求所有解的算法的执行时间为 $T(n)=m_0+m_0m_1+m_0m_1m_2+\cdots+m_0m_1m_2\cdots m_{n-1}$。这是一种最坏情况下的时间分析方法,在实际中可以通过剪支提高性能。为了估算更精确,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,然后再取这些结点总数的平均值。通常情况下,回溯法的效率高于穷举法。

5.2 深度优先搜索

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

5.2.1 图的深度优先遍历

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

```
int adj[][]={{1,3},{0},{3,4},{0,2,4},{2,3}};
```

从顶点 v 出发求深度优先遍历序列 ans 的算法如下:

```
int visited[];                              //访问标记数组
List<Integer>ans;                           //存放一个 DFS 序列
void dfs1(int adj[][],int v) {              //深度优先遍历
    ans.add(v);                             //访问顶点 v
    visited[v]=1;
    for(int u:adj[v]) {                     //找到 v 的相邻点 u
        if(visited[u]==0)                   //若顶点 u 尚未访问
            dfs1(adj,u);                    //从 u 出发继续搜索
    }
}
List<Integer>dfs(int adj[][],int v) {       //返回 DFS 序列
    ans=new ArrayList<>();                  //存放一个 DFS 序列
    visited=new int[adj.length];
    Arrays.fill(visited,0);                 //初始化所有元素为 0
    dfs1(adj,v);
    return ans;
}
```

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

5.2.2 深度优先遍历和回溯法的差别

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

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

解:采用深度优先遍历求 u 到 v 的所有路径时是从顶点 u 出发以纵向方式进行顶点搜索,用 x 存放一条路径,用 ans 存放所有的路径,如果当前访问的顶点 $u=v$,将找到的一条路径 x 添加到 ans 中,同时从顶点 u 回退以便找其他路径,否则找到 u 的所有相邻点 w,若顶点 w 尚未访问,则从 w 出发继续搜索路径,当从 u 出发的所有路径搜索完毕,再从 u 回退。对应的算法如下:

```
int visited[];
List<List<Integer>>ans;                    //存放所有路径
void dfs11(int adj[][],int u,int v,ArrayList<Integer>x) {      //深度优先遍历
    x.add(u);                              //访问顶点 u
    visited[u]=1;
    if(u==v) {                             //找到一条路径
        ans.add(new ArrayList<>(x));       //将路径 x 添加到 ans 中
        visited[u]=0;                      //置 u 可以重新访问
        x.remove(x.size()-1);              //路径回退
        return;
    }
    for(int w:adj[u]) {                    //找到 u 的相邻点 w
        if(visited[w]==0)                  //若顶点 w 尚未访问
            dfs11(adj,w,v,x);              //从 w 出发继续搜索
    }
    visited[u]=0;                          /从/u 出发的所有路径找完后回退
    x.remove(x.size()-1);                  //路径回退
}
List<List<Integer>>dfs1(int adj[][],int u,int v) {     //求 u 到 v 的所有路径
    ans=new ArrayList<>();                 //存放所有路径
    ArrayList<Integer>x=new ArrayList<>();  //存放一条路径
    visited=new int[adj.length];
    Arrays.fill(visited,0);                //初始化所有元素为 0
    dfs11(adj,u,v,x);
    return ans;
}
```

调用上述 dfs1(adj,0,4)时求出的两条路径是 0→3→2→4 和 0→3→4。现在采用回溯

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

```
void dfs21(int adj[][],int u,int v,ArrayList<Integer>x) {      //回溯法
    if(u==v)                                                   //找到一条路径
        ans.add(new ArrayList<>(x));                           //将路径 x 添加到 ans 中
    else {
        for(int w:adj[u]) {                                    //找到 u 的相邻点 w
            if(visited[w]==0) {                                //若顶点 w 尚未访问
                x.add(w);                                      //访问 v,将 v 添加到 ans 中
                visited[w]=1;
                dfs21(adj,w,v,x);                              //从 w 出发继续搜索
                visited[w]=0;                                  //从 w 回退到 u
                x.remove(x.size()-1);
            }
        }
    }
}
List<List<Integer>>dfs2(int adj[][],int u,int v) {     //求 u 到 v 的所有路径
    ans=new ArrayList<>();                                     //存放所有路径
    ArrayList<Integer>x=new ArrayList<>();                     //存放一条路径
    visited=new int[adj.length];
    Arrays.fill(visited,0);                                    //初始化所有元素为 0
    x.add(u);                                                  //将起始点 u 添加到 x 中
    visited[u]=1;
    dfs21(adj,u,v,x);
    return ans;
}
```

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

扫一扫

视频讲解

5.2.3 实战——二叉树的所有路径(LeetCode257★)

1. 问题描述

给定一棵含 $n(1\leqslant n\leqslant 100)$个结点的二叉树的根结点 root，结点值在[−100,100]内，设计一个算法按任意顺序返回所有从根结点到叶子结点的路径。例如，对于如图 5.5 所示的

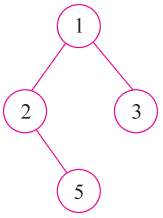


图 5.5　一棵二叉树

二叉树,返回结果是{"1->2->5","1->3"}。要求设计如下方法:

```
public List<String>binaryTreePaths(TreeNode root) { }
```

2. 问题求解——深度优先遍历

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

```
class Solution {
    List<String>ans=new ArrayList<>();                  //存放所有路径
    public List<String>binaryTreePaths(TreeNode root) {
        if(root==null)
            return ans;
        ArrayList<Integer>apath=new ArrayList<>(); //存放一条路径
        dfs(root,apath);                                //DFS 求 ans
        return ans;
    }
    void dfs(TreeNode root,ArrayList apath) {           //回溯算法
        apath.add(root.val);
        if(root.left==null && root.right==null) {   //找到一条路径
            String tmp="";                              //路径转换为字符串
            tmp+=apath.get(0);
            for(int i=1;i<apath.size();i++) {
                tmp+="->";
                tmp+=apath.get(i);
            }
            ans.add(tmp);
        }
        else {
            if(root.left!=null)
                dfs(root.left,apath);
            if(root.right!=null)
                dfs(root.right,apath);
        }
        apath.remove(apath.size()-1);                   //从结点 root 回退
    }
}
```

上述程序的提交结果为通过，运行时间为 2ms，消耗空间为 37.9MB。

3. 问题求解——回溯法

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

```
class Solution {
    List<String>ans=new ArrayList<>();                  //存放所有路径
    public List<String>binaryTreePaths(TreeNode root) {
        if(root==null)
            return ans;
        ArrayList<Integer>apath=new ArrayList<>();      //存放一条路径
        apath.add(root.val);                            //将根结点添加到路径中
        dfs(root,apath);                                //DFS 求 ans
        return ans;
    }
    void dfs(TreeNode root,ArrayList apath) {           //回溯算法
        if(root.left==null && root.right==null) {       //找到一条路径
            String tmp="";                              //路径转换为字符串
            tmp+=apath.get(0);
            for(int i=1;i<apath.size();i++) {
                tmp+="->";
                tmp+=apath.get(i);
            }
            ans.add(tmp);
        }
        else {
            if(root.left!=null) {
                apath.add(root.left.val);               //扩展左结点
                dfs(root.left,apath);
                apath.remove(apath.size()-1);           //从 root.left 回退
            }
            if(root.right!=null) {
                apath.add(root.right.val);              //扩展右结点
                dfs(root.right,apath);
                apath.remove(apath.size()-1);           //从 root.right 回退
            }
        }
    }
}
```

上述程序的提交结果为通过，运行时间为 2ms，消耗空间为 38.5MB。

5.3 基于子集树框架的问题求解

5.3.1 子集树算法框架概述

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

设问题解是一个 n 维向量 $(x_1,x_2,\cdots,x_n)$，约束函数为 constraint(i,j)，限界函数为 bound(i,j)，解空间为子集树的递归回溯框架如下：

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

对采用上述算法框架的几点注意事项说明如下：

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

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

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

扫一扫

视频讲解

5.3.2 实战——子集(LeetCode78★★)

1. 问题描述

见 3.3.5 节，这里采用回溯法求解。