第5章 图的搜索算法

在“数据结构”课程中，对每种数据结构的基本操作都包括“遍历”操作。这是很容易理解的，如果不能用一定的方法访问问题中的所有对象，其他操作就无从谈起。在学习了第4章的基本算法策略后，还有很多复杂的问题不能解决，本章介绍的是一类适用性较强，但效率有限的又一个蛮力策略实例——搜索算法。

搜索算法(search algorithms)是利用计算机的高性能来有目的地枚举一个问题的部分或所有可能情况，从而找出问题的解。搜索过程实际上是根据初始条件和扩展规则，构造一棵解答树并在其上寻找符合目标状态结点的过程。

5.1 图搜索概述


图的搜索算法

图是一种限制最少的数据结构，因此更接近现实，实际问题中很多数据关系都可以抽象成图，相关问题则可利用图的基本算法进行求解，很早就有专门研究图的一门数学学科“图论”，其中的计算问题包括图的搜索、路径问题、连通性问题、可平面性检验、着色问题、网络优化等。图论中的著名算法有：求最小生成树的Kruskal算法、求最短路径的Dijkstra算法和Floyd算法、求二部图最大匹配(指派问题)的匈牙利算法、求一般图最大匹配的Edmonds“花”算法、求网络最大流和最小流的算法等。其中的一些算法在“数据结构”课程中已经学习过了。

5.1.1 图及其术语

1. 显式图与隐式图

在路径问题、连通性问题、可平面性检验、着色问题和网络优化问题中，图的结构是显式给出的，包括图中的顶点、边及权重，这类图称为显式图，也就是一般意义上的图(graph)。

还有一些求解、最优化或证明类问题，采用的方法是类似枚举的试探搜索方法。问题一般仅给出初始结点和想要达到的目标；解决过程是通过在某个可能的解空间(solution space)内寻找所要的解或最优解。问题的解空间多是树形或图形结构，但并没有显式给出，称为隐式图(implicit graph)。隐式图是由问题的初始结点，为了求解或求证问题，根据题目的规则(一般是由题目的意思隐含给出的)，也就是生成子结点的约束条件，逐步扩展结点，直到得到目标结点为止的一个隐式的图。

很多在二维表格上提出的问题，如 8 皇后问题、老鼠走迷宫，甚至于五子棋、象棋等博弈类问题，问题的描述和解决其实都是隐式图的应用，它们在二维表格上根据不同的规则进行搜索求解或求证问题。

【提示】 “图”一词在“离散数学”“数据结构”课程中都学习过，你思考过它与现实中所说“图”的区别与联系吗？

2. 显式图的常用术语

如图 5-1(a)，(b)，(c)所示的均为显式图 (graph)。它有若干个不同的点 v_1，v_2，…，v_n，在其中一些点之间用直线或曲线连接。图中的这些点称为顶点 (vertex)或结点，连接顶点的曲线或直线称为边 (edge)。通常将这种由若干个顶点以及连接某些顶点的边所组成的图形称为图，顶点通常称作图中的数据元素。

带权图：给图 5-1 中各图的边上附加一个代表性的数据 (例如表示长度、流量或其他)，则称其为带权图，如图 5-2 所示。

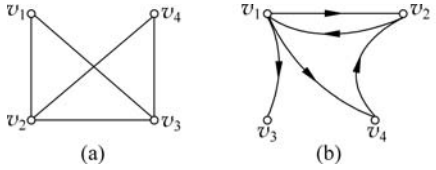

图 5-1　显式图

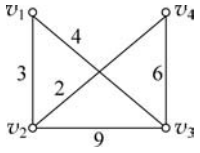

图 5-2　带权图

环(cycle)：图 5-1(c)所示图中的 v_1 点本身也有边相连，这种边称为环。

有限图：顶点与边数均为有限的图，如图 5-1 中的 3 个图均属于有限图。

简单图：没有环且每两个顶点间最多只有一条边相连的图，如图 5-1(a)所示。

邻接与关联：当(v_1,v_2)(无向边)$\in E$ 或$<v_1,v_2>$(有向边)$\in E$，即 v_1，v_2 间有边相连时，则称 v_1 和 v_2 是相邻的，它们互为邻接点(adjacent)，同时称(v_1,v_2)或$<v_1,v_2>$是与顶点 v_1，v_2 相关联的边。

顶点的度数(degree)：从该顶点引出的边的条数，即与该顶点相关联的边的数目，简称度。

入度(indegree)：有向图中把以顶点 v 为终点的边的条数称为顶点 v 的入度。

出度(outdegree)：有向图中把以顶点 v 为起点的边的条数称为顶点 v 的出度。

终端顶点：有向图中把出度为 0 的顶点称为终端顶点，如图 5-1(b)中的 v_3。

路径与路长：在图 $G=(V,E)$中，如果存在由不同的边(v_{i0},v_{i1})，(v_{i1},v_{i2})，…，(v_{in-1},v_{in})或是$(<v_{i0},v_{i1}>,<v_{i1},v_{i2}>,\cdots,<v_{in-1},v_{in}>)$组成的序列，则称顶点 v_{i0}，v_{in} 是连通的，顶点序列$(v_{i0},v_{i1},v_{i2},\cdots,v_{in})$是从顶点 v_{i0} 到顶点 v_{in} 的一条道路。路长是道路上边的数目，v_{i0} 到 v_{in} 的这条道路上的路长为 n。

连通图：对于图中任意两个顶点 v_i，$v_j\in V$，v_i，v_j 之间有道路相连，则称该图为连通图，如图 5-1(a)所示。

网络：带权的连通图，如图 5-2 所示。

3. 隐式图术语

隐式图没有明确的点和边,一般仅有起点,其他点和边由隐含的规则得出,如博弈树、迷宫问题等。树是图的一个特例,是很多问题的搜索空间。下面就是两种典型的隐式图。

1) 子集树

当要求解的问题需要在 n 个元素的子集中进行搜索,其搜索空间树称作子集树(subset tree)。这 n 个元素在子集中或被选取记为 1,不在子集中或被舍去记为 0,这样搜索空间为:

$(0,0,\cdots,0,0),(0,0,\cdots,0,1),(0,0,\cdots,1,0),$

$(0,0,\cdots,1,1),\cdots,(1,1,\cdots,1,1)$

共 2^n 个状态。若表示为树形结构就是一棵有 2^n 个叶结点的二叉树,$n=3$ 时的子集树如图 5-3 所示。

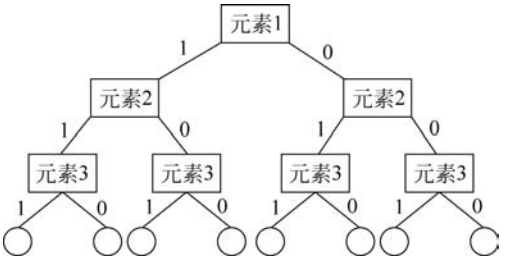

图 5-3 $n=3$ 的子集树

因此,对树中所有分支进行遍历的算法都必须耗时 $O(2^n)$。

2) 排列树

当要求解的问题需要在 n 元素的排列中搜索问题的解时,解空间树称作排列树(permutation tree)。搜索空间为:

$(1,2,3,\cdots,n-1,n),(2,1,3,\cdots,n-1,n),(2,3,1,\cdots,n-1,n),$

$(2,3,4,1,\cdots,n-1,n),\cdots,(n,n-1,\cdots,3,2,1)$

第 1 个元素有 n 种选择,第 2 个元素有 $n-1$ 种选择,第 3 个元素有 $n-2$ 种选择,……,第 n 个元素有 1 种选择,共计 $n!$ 个状态。若表示为树形就是一个 n 度树,这样的树有 $n!$ 个叶结点,所以遍历树中所有结点的算法都耗时 $O(n!)$。当 $n=4$ 时的排列树如图 5-4 所示。

【提示】 根据以上两个隐式图的特例,思考其他隐式图的例子。

4. 图的存储

从隐式图的定义和以上介绍的两个特殊隐式图知道,隐式图是在一定规律下构造的,一般不需要专门的存储,而是搜索中根据隐式图的规律特点确定搜索过程。

而显式图的顶点及顶点之间的关系没有规律可以遵循,只有通过一定的存储方式将其存储后,才能对其进行操作。“数据结构”课程中已经介绍,显式图的最常用的方法有两种:邻接矩阵法和邻接表法。

【提示】 请思考现实中的图应该存储哪些信息。

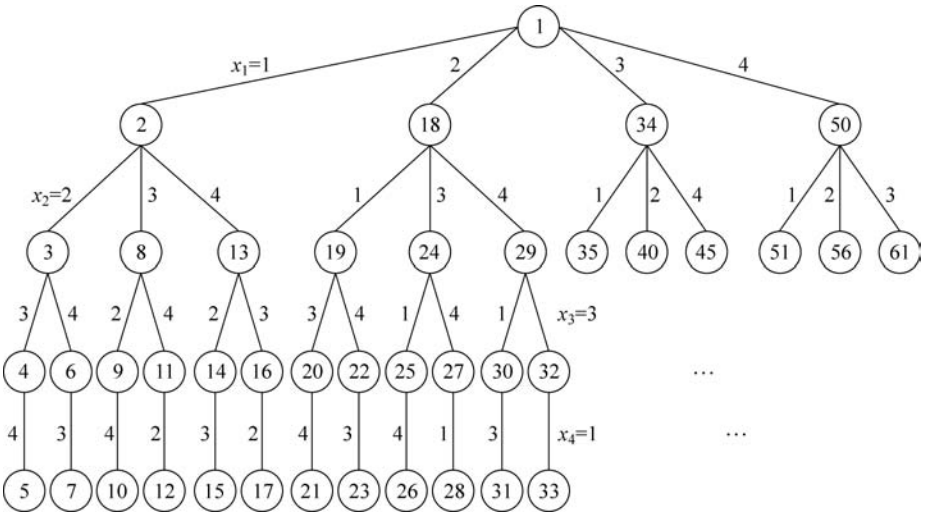

图 5-4　$n=4$ 的排列树

1）邻接矩阵

邻接矩阵是表示顶点之间相邻关系的矩阵，设 $G=(V,E)$ 是具有 n 个顶点的图，则 G 的邻接矩阵是具有如下性质的 n 阶方阵：

$\boldsymbol{A}[i,j]=1$，若(v_i,v_j)或$<v_i,v_j>$是$E(G)$中的边。

$\boldsymbol{A}[i,j]=0$，若(v_i,v_j)或$<v_i,v_j>$不是$E(G)$中的边。

若 G 是网络，则邻接矩阵可定义为：

$\boldsymbol{A}[i,j]=W_{ij}$，若$(v_i,v_j)$或$<v_i,v_j>\in E(G)$。

$\boldsymbol{A}[i,j]=0$ 或∞，若(v_i,v_j)或$<v_i,v_j>\notin E(G)$。

其中，W_{ij} 表示边上的权值，∞表示一个计算机允许的，大于所有边上权值的数。

例如，图 5-1(a)的邻接矩阵为：

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

2）邻接表

对于图 G 中的每个结点 v_i，该方法把所有邻接于 v_i 的顶点 v_j 链成一个单链表，这个单链表就称为顶点 v_i 的邻接表。邻接表由顶点表和边表两部分组成。

边表为一个单链表，每个表结点均有两个域：

① 邻接点域 adjvex，存放与 v_i 相邻接的顶点 v_j 的序号 j。

② 链域 next，将邻接表的所有表结点链在一起。

顶点表为一数组，每个元素均有两个域：

① 顶点域 vertex，存放顶点 v_i 的信息。

② 指针域 firstedge，存放 v_i 的边表的头指针。

对于无向图来说，v_i 的邻接表中每个表结点都对应于与 v_i 相关联的一条边，对于有向图来说，v_i 的邻接表中每个表结点对应于 v_i 为始点射出的一条边。

例如，图 5-1(a)的邻接表如图 5-5 所示。

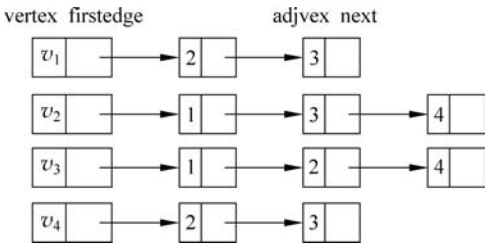

图 5-5 图 5-1(a)的邻接表

5.1.2 图搜索及其术语

1. 穷举搜索与启发式搜索

假设问题的初始状态、算符和目标状态的定义都是完全确定的，就可以决定问题的一个解空间。然后求解问题就在于如何有效地搜索这个给定的解空间，从中找出问题的真正解。

对图的最基本搜索算法是穷举搜索，是蛮力策略的一种表现形式。因为算法要解决的问题只有有限种可能解，在没有更好的算法时，总可以用穷举搜索的办法解决，即逐个地检查所有可能的情况，从中找到解。

可以想象，当问题的解空间状态较多时，穷举搜索方法极为费时。有时并不需要机械地检查每一种状态，常常可以利用一些启发信息，提前判断出先搜索哪些状态有可能尽快找到问题的解或某些情况不可能取到最优解，从而可以提前舍弃对这些状态的尝试。这样也就隐含地检查了所有可能的情况，既减少了搜索量，又保证了不漏掉最优解，这就是启发式搜索。

总之，搜索分为两种：不考虑给定问题的特有性质，按图形特点事先定好的顺序，依次运用规则，即盲目搜索(穷举搜索)；另一种则考虑问题给定的特有性质，选用合适的规则，提高搜索的效率，即启发式的搜索。

一般对显式图的穷举搜索分为深度优先搜索算法和广度优先搜索算法；对隐式图的穷举搜索一般有回溯算法和分支算法。在这些算法中都可以加入提高搜索效率的启发信息，优化成对应的启发式搜索，如分支限界算法等。

2. 相关概念和术语

为便于讨论，引进一些关于搜索解空间树结构的术语。树中的每一个结点确定所求解问题的一个问题状态(problem states)。由根结点到其他结点的所有路径(分支)，就确定了这个问题的状态空间(state space)。解状态(solution states)是这样一些问题状态 S，对于这些问题状态，由根到 S 的那条路径确定了该解空间中的一个元组。答案状态(leaves states，搜索到叶子就是答案状态)是这样的一些解状态 S，对于这些解状态而言，由根到 S 的这条路径确定了该问题的一个解(即它满足隐式约束条件)。解空间的树结构称为状态空间树(隐式图)(state space tree)。

对于任何一个问题，一旦设想出一种状态空间树，那么就可以先系统地生成问题状态，接着确定这些问题状态中的哪些状态是解状态，最后确定哪些解状态是答案状态，从而将问题解出。搜索状态空间树一般都是从根结点开始然后生成其他结点，如果已生成一个结点而它的所有儿子结点还没有全部生成，则这个结点叫作活结点(active node)。当前正在生成其儿子结点的活结点叫 E-结点(expansion node)(正在扩展的结点)。不再进一步扩展或

者其儿子结点已全部生成的结点就是死结点(dead node)。

5.2 广度优先搜索

设图 G 的初始状态是所有顶点均未访问过。以 G 中任选一顶点 v 为起点,则广度优先搜索定义为:首先访问出发点 v,接着依次访问 v 的所有邻接点 $w_1,w_2,\cdots,w_t$,然后再依次访问与 $w_1,w_2,\cdots,w_t$ 邻接的所有未曾访问过的顶点。以此类推,直至图中所有和起点 v 有路径相通的顶点都已访问到为止。此时从 v 开始的搜索过程结束。

若 G 是连通图,则一次就能搜索完所有结点;否则,在图 G 中另选一个尚未访问的顶点作为新源点继续上述的搜索过程,直至 G 中所有顶点均已被访问为止。

由于在数据结构中已经学习过广度优先搜索(遍历),这一节的重点并不是学习简单的搜索方法,而是要学习广度优先搜索的应用。

5.2.1 广度优先算法框架

1. 算法的基本思路

此算法主要用于解决在显式图中寻找某一方案的问题,解决问题的方法就是通过搜索图的过程中进行相应的操作,从而解决问题。由于在搜索过程中一般不能确定问题的解,只有在搜索结束(或到达目标)后,才能得出问题的解。这样在搜索过程中,有一个重要操作就是记录当前找到的解决问题的方案。

算法设计的基本步骤如下:

(1) 确定图的存储方式;

(2) 设计图搜索过程中的操作,其中包括为输出问题解而进行的存储操作;

(3) 输出问题的结论。

2. 算法框架

从广度优先搜索定义可以看出活结点的扩展是按先来先处理的原则进行的,所以在算法中要用“队”来存储每个 E-结点扩展出的活结点。为了算法的简洁,抽象地定义:

queue 为队列类型,InitQueue( ) 为队列初始化函数,EnQueue(Q,k) 为入队函数,QueueEmpty(Q) 为判断队空函数,DeQueue(Q) 为出队函数。

在实际应用中根据操作的方便性,用数组或链表实现队列。

在广度优先扩展结点时,一个结点可能多次作为扩展对象,这是需要避免的。一般开辟数组 visited 记录图中结点被搜索的情况。

在算法框架中以输出结点值表示“访问”,具体应用中可根据实际问题进行相应的操作。在下一节的算法应用中,大家会有所体会。

1) 邻接表表示图的广度优先搜索算法

```
int visited[n];                                    // n 为结点个数,数组元素的初值均置为 0
bfs(int k,graph head[])
  {int i;
```

```
queue Q;                                          // 定义队列
edgenode * p;
InitQueue(Q);                                     // 队列初始化
print("visit vertex",k);                          // 访问源点 vk
visited[k] = 1;
EnQueue(Q,k);                                     // vk 已访问,将其入队
while(not QueueEmpty(Q))                          // 队非空则执行
   { i = DeQueue(Q);                              // vi 出队为 E-结点
     p = head[i].firstedge;                       // 取 vi 的边表头指针
     while(p <> null)                             // 扩展 E-结点
       {if(visited[p -> adjvex] = 0)              // 若 vj 未访问过
         { print ("visitvertex",p -> adjvex);     // 访问 vj
           visited[p -> adjvex] = 1;
           EnQueue(Q,p -> adjvex); }              // 访问过的 vj 入队
        p = p -> next;                            // 找 vi 的下一邻接点
       }
    }
}
```

2) 邻接矩阵表示的图的广度优先搜索算法

```
int visited[n];                                   // n 为结点个数,数组元素的初值均置为 0
bfsm(int k,graph g[][100],int n)
  {int i,j;
   queue Q;
   InitQueue(Q);
   print ("visit vertex",k);                      // 访问源点 vk
   visited[k] = 1;
   EnQueue(Q,k);
   while(not QueueEmpty(Q))
     {i = DeQueue(Q);                             // vi 出队
      for(j = 0; j < n; j = j + 1)                // 扩展结点
        if(g[i][j] = 1 and visited[j] = 0)
           {print("visit vertex",j);
            visited[j] = 1;
            EnQueue(Q,j); }                       // 访问过的 vj 入队
     }
  }
```

5.2.2 广度优先搜索的应用

下面通过例题学习广度优先搜索的应用,不同问题的广度优先搜索算法框架虽然是一样的,但在具体处理方法上,编程技巧上不尽相同。

【例 1】 已知若干个城市的地图,求从一个城市到另一个城市的路径,要求该路径经过的城市最少。

算法设计:图的广度优先搜索类似于树的层次遍历,逐层搜索正好可以尽快找到一个结点与另一个结点相对而言最直接的路径。所以此问题适用于广度优先算法。下面通过一

个具体例子来进行算法设计。

图 5-6 表示的是从城市 A 到城市 H 的交通图。从图中可以看出，从城市 A 到城市 H 要经过若干个城市。现要找出经过城市最少的一条路线。

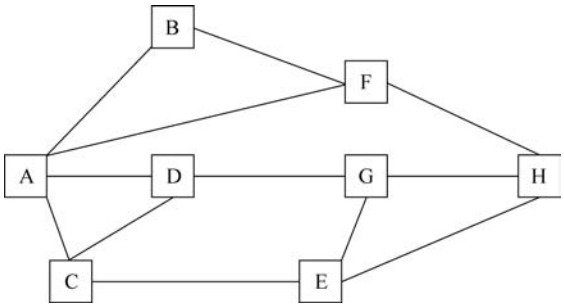

图 5-6　交通图

该图的邻接矩阵表示如表 5-1 所示，0 表示能走，1 表示不能走。

表 5-1　图 5-6 所示交通图的邻接矩阵

城　市	A	B	C	D	E	F	G	H
A	0	1	1	1	0	1	0	0
B	1	0	0	0	0	1	0	0
C	1	0	0	1	1	0	0	0
D	1	0	1	0	0	0	1	0
E	0	0	1	0	0	0	1	1
F	1	1	0	0	0	0	0	1
G	0	0	0	1	1	0	0	1
H	0	0	0	0	1	1	1	0

具体过程如下：

(1) 将城市 A(编号 1)入队，队首指针 qh 置为 0，队尾指针 qe 置为 1。

(2) 将队首指针所指城市的所有可直通的城市入队，当然如果这个城市在队中出现过就不入队，然后将队首指针加 1，得到新的队首城市。重复以上步骤，直到城市 H(编号为 8)入队为止。当搜索到城市 H 时，搜索结束。

(3) 输出经过最少城市的线路。

数据结构设计：考虑到算法的可读性，用线性数组 a 作为活结点队的存储空间。为了方便输出路径，队列的每个结点有两个成员，$a[i]$.city 记录入队的城市，$a[i]$.pre 记录该城市的前趋城市在队列中的下标，这样通过 $a[i]$.pre 就可以倒推出最短线路。也就是说活结点队同时又是记录所求路径的空间。因此，数组队并不能做成循环队列，所谓“出队”只是队首指针向后移动，其空间中存储的内容并不能被覆盖。

和广度优先算法框架一样，设置数组 visited[]记录已搜索过的城市。算法如下：

```
int jz[8][8] = {{0,1,1,1,0,1,0,0},{1,0,0,0,0,1,0,0},{1,0,0,1,1,0,0,0},{1,0,1,0,0,0,1,0},
{0,0,1,0,0,0,1,1},{1,1,0,0,0,0,0,1},{0,0,0,1,1,0,0,1},{0,0,0,0,1,1,1,0}};
struct {int city,pre; }sq[100];
```

```
int qh,qe,i,visited[100];
main( )
{int i,n = 8;
  for(i = 1; i <= n,i = i + 1)
    visited[i] = 0;
  search( );
}
search( )
  { qh = 0;
    qe = 1;
    sq[1].city = 1;
    sq[1].pre = 0;
    visited[1] = 1;
    while(qh <> qe)                                  // 当队不空
        {qh = qh + 1;                                // 结点出队
        for(i = 1; i <= n,i = i + 1)                 // 扩展结点
          if (jz[sq[qh].city][i] = 1 and visited[i] = 0)
            { qe = qe + 1;                           // 结点入队
              sq[qe].city = i;
              sq[qe].pre = qh;
              visited[i] = 1;
              if (sq[qe].city = 8)
                 {out( );
                 return; }
             }
          }
       print("Non solution.");
      }
out( )                                               // 输出路径
    {print(sq[qe].city);
        while(sq[qe].pre <> 0)
          { qe = sq[qe].pre;
          print('- -',sq[qe].city); }
    }
```

算法分析：算法的时间复杂度是 $O(n^2)$。算法的空间复杂度为 $O(n^2)$，包括图本身的存储空间和搜索时辅助空间“队”的存储空间。

【提示】 题目是求城市 A 到 H 的路径，而算法输出的结果是城市 H 到 A 的路径，虽然也算解决了问题，还是请读者改写算法给出更合理的输出。

还可以充分利用图的存储空间来记录结点的访问情况，如下面的例子。

【例 2】 走迷宫问题。

迷宫是许多小方格构成的矩形，如图 5-7 所示，在每个小方格中有的是墙(图中的“1”)有的是路(图中的“0”)。走迷宫就是从一个小方格沿上、下、左、右四个方向到邻近的方格，当然不能穿墙。设迷宫的入口是在左上角(1,1)，出口是右下角(8,8)。根据给定的迷宫，找出一条从入口到出口的路径。

1,1

0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0
0	0	0	0	1	0	1	0
0	1	0	0	0	0	1	0
0	1	0	1	1	0	1	0
0	1	0	0	0	0	1	1
0	1	0	0	1	0	0	0
0	1	1	1	1	1	1	0

8,8

图 5-7　矩形图

算法设计：此题的设计思路与例 1 完全相同，从入口开始广度优先搜索所有可到达的方格入队，再扩展队首的方格，直到搜索到出口时算法结束。

根据迷宫问题的描述，若把迷宫作为图，则每个方格为顶点，其上、下、左、右的方格为其邻接点。迷宫是 8×8＝64 个结点的图，那样邻接矩阵将是一个 64×64 的矩阵，且需要编写专门的算法去完成迷宫的存储工作。这显然是不必要的，因为搜索方格的过程是有规律的。对于迷宫中的任意一点 $A(Y,X)$，有 4 个搜索方向：向上 $A(Y-1,X)$；向下 $A(Y+1,X)$；向左 $A(Y,X-1)$；向右 $A(Y,X+1)$。当对应方格可行(值为 0)，就扩展为活结点，同时注意防止搜索不要出边界就可以了。

数据结构设计：这里同样用数组做队的存储空间，队中结点有 3 个成员：行号、列号、前一个方格在队列中的下标。与例 1 不同的是，搜索过的方格不另外开辟空间记录其访问的情况，而是用迷宫原有的存储空间，元素值置为“－1”时，标识已经访问过该方格。

为了构造循环体，用数组 fx[]＝{1，－1，0，0}，fy[]＝{ 0，0，－1，1 }模拟上下左右搜索时的下标的变化过程。算法如下：

```
int maze[8][8] = {{0,0,0,0,0,0,0,0},{0,1,1,1,1,0,1,0},{0,0,0,0,1,0,1,0},{0,1,0,0,0,0,1,0},
{0,1,0,1,1,0,1,0},{0,1,0,0,0,0,1,1},{0,1,0,0,1,0,0,0},{0,1,1,1,1,1,1,0}};
int fx[4] = {1, - 1,0,0},fy[4] = { 0,0, - 1,1 };      // 下标起点为 1
struct {int x,y,pre; }sq[100];
int qh,qe,i,j,k;
main( )
{search( );
}
search( )
  {   qh = 0;
      qe = 1;
      maze[1][1] = - 1;
      sq[1].pre = 0; sq[1].x = 1; sq[1].y = 1;
      while(qh <> qe)                                  // 当队不空
         {qh = qh + 1;                                 // 出队
         for(k = 1; k < = 4; k = k + 1)                // 搜索可达的方格
           {i = sq[qh].x + fx[k];
            j = sq[qh].y + fy[k];
            if  (check(i,j) = 1)
              { qe = qe + 1;                           // 入队
```

```
                sq[qe].x = i; sq[qe].y = j;
                sq[qe].pre = qh;
                maze[i][j] = -1;
                if (sq[qe].x = 8 and sq[qe].y = 8)
                  {out( );
                    return; }
                }
            }
            print("Non solution.");
          }
check(int i,int j)
   {int flag = 1;
    if(i<1 or i>8 or j<1 or j>8)                    // 是否在迷宫内
        flag = 0;
    if(maze[i][j] = 1 or maze[i][j] = -1)           // 是否可行
        flag = 0;
    return(flag);
    }
out( )                                              // 输出过程
   {print("(",sq[qe].x,"," sq[qe].y,")");
      while(sq[qe].pre<>0)
        { qe = sq[qe].pre;
        print('- -',"(",sq[qe].x,"," sq[qe].y,")"); }
    }
```

算法分析：这个题目的搜索空间为 8^2 时间复杂度是 $O(n^2)$。算法的空间复杂度为 $O(n^2)$，包括图本身的存储空间和搜索时辅助空间“队”的存储空间。

5.3 深度优先搜索

给定图 G 的初始状态是所有顶点均未曾访问过，在 G 中任选一顶点 v 为初始出发点(源点或根结点)，则深度优先遍历可定义如下：首先访问出发点 v，并将其标记为已访问过；然后依次从 v 出发搜索 v 的每个邻接点(子结点)w。若 w 未曾访问过，则以 w 为新的出发点继续进行深度优先遍历，直至图中所有和源点 v 有路径相通的顶点(亦称为从源点可达的顶点)均已被访问为止。若此时图中仍有未访问的顶点，则另选一个尚未访问的顶点作为新的源点重复上述过程，直至图中所有顶点均已被访问为止。

深度搜索与广度搜索相近，最终都要扩展一个结点的所有子结点，区别在于扩展结点的过程不同，深度搜索扩展的是 E-结点的邻接结点(子结点)中的一个，并将其作为新的 E-结点继续扩展，当前 E-结点仍为活结点，待搜索完其子结点后，回溯到该结点扩展它的其他未搜索的邻接结点。而广度搜索，则是连续扩展 E-结点的所有邻接结点(子结点)后，E-结点就成为一个死结点。

5.3.1 深度优先算法框架

1. 算法的基本思路

深度优先搜索和广度优先搜索的基本思路相同。由于深度优先搜索的 E-结点是分多

次进行扩展的，所以它可以搜索到问题所有可能的解方案。但对于搜索路径的问题，不像广度优先搜索容易得到最短路径。

和广度优先搜索一样，搜索过程中也需要记录解决问题的方案。

深度优先搜索算法设计的基本步骤为：

(1) 确定图的存储方式。

(2) 设计搜索过程中的操作，其中包括为输出问题解而进行的存储操作。

(3) 搜索到问题的解，则输出；否则回溯。

(4) 一般在回溯前应该将结点状态恢复为原始状态，特别是在有多解需求的问题中。

2. 算法框架

从深度优先搜索定义可以看出算法是递归定义的，用递归算法实现时，将结点作为参数，这样参数栈就能存储现有的活结点。当然若是用非递归算法，则需要自己建立并管理栈空间。

同样用“输出结点值”抽象地表示实际问题中的相应操作。

(1) 用邻接表存储图的搜索算法如下：

```
int visited[n];                          // n 为结点个数,数组元素的初值均置为 0
graph head[100];                         // graph 为邻接表存储类型
dfs(int k)                               // head 图的顶点数组
{ edgenode * ptr;                        // ptr 图的边表指针
  visited[k] = 1;
  print("访问 ",k);
  ptr = head[k].firstedge;               // 顶点的第一个邻接点
  while (ptr <> NULL)                    // 遍历至链表尾
       {if (visited[ptr -> vertex] = 0)
             dfs(ptr -> vertex);         // 递归遍历
  ptr = ptr -> nextnode;                 // 下一个顶点
  }
}
```

算法分析：图中有 n 个顶点，e 条边。如果用邻接表表示图，由于总共有 $2e$ 个边结点，所以扫描边的时间为 $O(e)$。而且对所有顶点递归访问 1 次，所以遍历图的时间复杂度为 $O(n+e)$。

(2) 用邻接矩阵存储图的搜索算法如下：

```
int visited[n];                          // n 为结点个数,数组元素的初值均置为 0
int g[n][n];
dfsm(int k)
  {int j;
  print("访问 ",k);
  visited[k] = 1;
  for(j = 1; j <= n; j = j + 1)          // 依次搜索 vk 的邻接点
    if(g[k][j] = 1 and visited[j] = 0)
      dfsm(j);                           // (vk,vj)∈E,且 vj 未访问过,故 vj 为新出发点
  }
```

如果用邻接矩阵表示图,则查找每一个顶点的所有的边,所需时间复杂度为 $O(n)$,则遍历图中所有的顶点所需的时间复杂度为 $O(n^2)$。

5.3.2 深度优先搜索的应用

下面学习深度优先搜索的应用,并通过例子认识广度优先搜索和深度优先搜索应用的不同特点。

【例 3】 走迷宫问题。

问题同 5.2.2 节中例 2,这里用深度优先搜索算法求解。

算法设计:深度优先搜索,就是一直向着可通行的下一个方格行进,直到搜索到出口就找到一个解。若行不通时,则返回上一个方格,继续搜索其他方向。

数据结构设计:广度优先搜索算法的路径是依赖"队列"中存储的信息,在深度优先搜索过程中虽然也有辅助存储空间栈,但并不能方便地记录搜索到的路径。因为并不是走过的方格都是可行路径,也就是通常说的可能走入了"死胡同"。所以,还是利用迷宫本身的存储空间,除了记录方格走过的信息,还要标识是否可行:

```
maze[i][j] = 3                          // 标识走过的方格
maze[i][j] = 2                          // 标识走入死胡同的方格
```

这样,最后存储为"3"的方格为可行的方格。而当一个方格 4 个方向都搜索完还没有走到出口,说明该方格或无路可走或只能走入了"死胡同"。

算法如下:

```
int maze[8][8] = {{0,0,0,0,0,0,0,0},{0,1,1,1,1,0,1,0},{0,0,0,0,1,0,1,0},{0,1,0,0,0,0,1,
0},{0,1,0,1,1,0,1,0},{0,1,0,0,0,0,1,1},{0,1,0,0,1,0,0,0},{0,1,1,1,1,1,1,0}},fx[4] =
{1,-1,0,0},fy[4] = { 0,0,-1,1 };        // 下标从 1 开始
int i,j,k,total = 0;
main( )
{
  maze[1][1] = 3;                       // 入口坐标设置已走标志
  search(1,1);
}
search(int i,int j)
{int k,newi,newj;
    for(k = 1; k <= 4; k = k + 1)       // 搜索可达的方格
      if(check(i,j,k) = 1)
        {newi = i + fx[k];
         newj = j + fy[k];
         maze[newi][newj] = 3;          // 来到新位置后,设置已走过标志
         if (newi = 8 and newj = 8)     // 如到出口则输出,否则下一步递归
            Out( );
         else
            search(newi,newj);
        }
    maze[i][j] = 2;                     // 某一方格只能走入死胡同
    }
```

```
Out( )
{int i,j;
 for(i = 1; i <= 8; i = i + 1)
   { print("换行符");
     for(j = 1; j <= 8; j = j + 1)
         if (maze[i][j] = 3)
           {print("V");
           total = total + 1; }          // 统计总步数
       else
           print(" * ");
     }
  print("Total is",total);
}
check(int i,int j,int k)
  {int flag = 1;
   i = i + fx[k];
   j = j + fy[k];
   if(i < 1 or i > 8 or j < 1 or j > 8)      // 是否在迷宫内
       flag = 0;
   else if(maze[i][j]<> 0)                   // 是否可行
       flag = 0;
   return(flag);
   }
```

算法说明：

(1) 本算法和广度优先算法一样每个方格有 4 个方向可以进行搜索，这样一个结点(方格)有可能多次成为活结点，而在广度优先算法中一个结点(方格)就只有一次成为活结点，出队后就变成了死结点，不再进行操作。

(2) 与广度优先算法相比较，在空间效率上二者相近，都需要辅助空间。

【提示】 用广度优先算法，最先搜索到的就是一条最短路径，而用深度优先搜索则能方便地找出一条可行路径。但若要保证找到最短路径，则需要找出所有路径，再从中筛选出最短路径。请改进算法求问题的最短路径。

从迷宫问题看，似乎广度优先算法优于深度优先算法，但其实它们各有所长，在不知道目标的情况下，要解决以下染色问题，广度优先算法就没有优势了。

【例 4】 有如图 5-8 所示的七巧板，试设计算法，使用至多 4 种不同颜色对七巧板进行涂色(每块涂一种颜色)，要求相邻区域的颜色互不相同，打印输出所有可能的涂色方案。

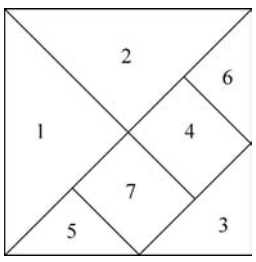

图 5-8　七巧板

问题分析： 本题实际上是一个简化的“4 色地图”问题，无论地图多么复杂，只需要用 4 种颜色就可以将相邻区域区分开。

为了让算法能识别不同区域间的相邻关系，把七巧板上每一个区域看成一个顶点，若两个区域相邻，则相应的顶点间用一条边相连，这样就将七巧板转化为图，于是该问题就是一个图的搜索问题了。数据采用邻接矩阵存储如下(顶点编号如图 5-8 所示)：

0 1 0 0 1 0 1
1 0 0 1 0 1 0
0 0 0 0 1 0 1
0 1 1 0 0 1 1
1 0 0 0 0 0 1
0 1 0 1 0 0 0
1 0 1 1 1 0 0

算法设计：前面已说过，搜索算法是蛮力算法的一个范例，在这个例子中体现得更加突出。在深度优先搜索顶点(即不同区域)时，并不加入任何涂色策略，只是对每一个顶点逐个尝试4种颜色，检查当前顶点的颜色是否与前面已确定的相邻顶点的颜色发生冲突(就像枚举算法检查是否满足约束条件)，若没有发生冲突，则继续以同样的方法处理下一个顶点；若4个颜色都尝试完毕，仍然与前面顶点的颜色发生冲突，则返回到上一个还没有尝试完4种颜色的顶点，再去尝试别的颜色。已经有研究证明，对任意平面图至少存在一种4色涂色法，所以问题肯定是有解的。

按顺序分别对1号，2号，…，7号区域进行试探性涂色，用1，2，3，4号代表4种颜色。则涂色过程如下：

(1) 对某一区域涂上与其相邻区域不同的颜色。

(2) 若使用4种颜色进行涂色均不能满足要求，则回溯一步，更改前一区域的颜色。

(3) 转步骤(1)继续涂色，直到全部区域均已涂色为止，输出结果。

算法如下：

```
  int data[7][7],n,color[7],total = 0; // 下标从 1 开始
main( )
  { int i,j;
    for(i = 1; i < = 7; i = i + 1)
      for(j = 1; j < = 7; j = j + 1)
        input(data[i][j]);
    for(j = 1; j < = 7; j = j + 1)
        color[j] = 0;
    total = 0;
    try(1);
    print("换行符,Total = ",total);
  }
try(int s)
  {int i;
   if (s > 7)
     output( );
   else
     for(i = 1; i < = 4; i = i + 1)
       {color[s] = i;
        if (colorsame(s) = 0)
           try(s + 1);
       }
```

```
    }
colorsame(int s)                          // 判断相邻点是否同色
    {int i,flag;
     flag = 0;
     for(i = 1; i <= s - 1; i = i + 1)
        if (data[i][s] = 1 and color[i] = color[s])
            flag = 1;
     return(flag);
     }
output( )
{int i;
 print("换行符,serial number: ",total);
 for(i = 1; i <= 7; i = i + 1)
         {print(color[i]); color[i] = 0;}
 total = total + 1;
 }
```

【提示】 思考算法能搜索到问题的全部解吗？或者说函数 try( )在调用了 output( )函数后，还会继续递归调用吗？若不是全部解，如何改进算法实现输出全部解。

【例 5】 割点的判断及消除。

1. 网络安全相关的概念

假设有两个通信网，如图 5-9 和图 5-10 所示。图中结点代表通信站，边代表通信线路。这两个图虽然都是无向连通图，但它们所代表的通信网的安全程度却截然不同。在图 5-9 所代表的通信网中，如果结点 2 代表的通信站发生故障，除它本身不能与任何通信站联系外，还会导致 1,3,4,9,10 号通信站与 5,6,7,8 号通信站之间的通信中断。结点 3 和结点 5 代表的通信站，若出故障也会发生类似的情况。而图 5-10 所示的通信网则不然，不管哪一个站点(仅一个)发生故障，其余站点之间仍可以正常通信。

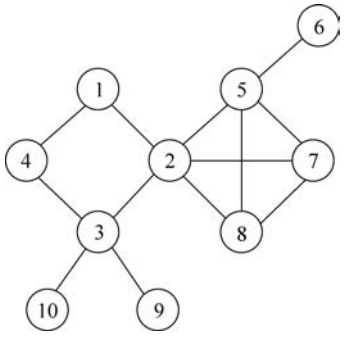

图 5-9　一个连通图

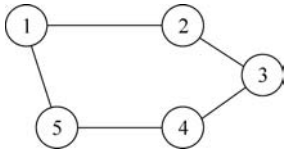

图 5-10　不含割点的连通图

出现以上差异的原因在于，这两个图的连通程度不同。图 5-9 是一个含有称之为割点 2,3,5 的连通图，而图 5-10 是不含割点的连通图。在一个无向连通图 G 中，当且仅当删去 G 中的顶点 v 及所有依附于 v 的边后，可将图分割成两个以上的连通分量，则称 v 为图 G

的割点(vertex connectivity)。将图 5-9 的结点 2 和与之相连的所有边删去后留下两个彼此不连通的非空分图,如图 5-11 所示,则结点 2 就是割点。

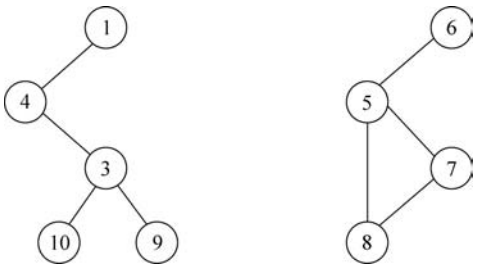

图 5-11 图 5-9 删去割点 2 的结果

没有割点的连通图称为重连通图(biconnected graph)。一个通信网络图的连通度越高,则系统越可靠,无论是哪一个站点单独出现故障或遭到外界破坏,都不影响系统的正常工作;又如,一个航空网若是重连通的,则当某条航线因天气等原因关闭时,旅客仍可从别的航线绕道而行;再如,若将大规模集成电路中的关键线路设计成重连通的,则在某些元件失效的情况下,整个电路的功能不受影响;反之,在战争中,若要摧毁敌方的运输线,仅需破坏其运输网中的割点即可。

网络安全比较低级的要求就是重连通图。在重连通图上,任何一对顶点之间至少存在有两条路径,在删除某个顶点及与该顶点相关联的边时,也不会破坏图的连通性。不是所有的无向连通图都是重连通图,根据重连通图的定义可以看出,其等价定义是"如果无向连通图 G 根本不包含割点,则称 G 为重连通图"。

以重连通图作为通信网络安全的判别及改进,要讨论的问题是:判别一个通信网络是否安全,若不安全则找出改进方法。直接一点说,就是能找出无向图中的割点,并能用简单的方法消灭找到的割点。

2. 连通图 G 的割点的判别

算法设计:从有关图论的资料中不难查到,连通图中割点的判别方法如下。

(1) 从图的某一个顶点开始深度优先遍历图,得到深度优先生成树。

(2) 用开始遍历的顶点作为生成树的根,则:

① 根顶点是图的割点的充要条件是,根至少有两个相邻结点。

② 其余顶点 u 是图的割点的充要条件是,该顶点 u 至少有一个相邻结点 w,从该相邻结点出发不可能通过该相邻结点顶点 w 和该相邻结点 w 的子孙顶点,以及一条回边所组成的路径到达 u 的祖先(图中非生成树的边称为回边)。

③ 特别地,叶结点不是割点。

以上判别方法不能说是一个算法,或者只能说是一个顶层的算法,因为(2)中的②不具有可行性。下面通过例子说明算法设计的过程。

图 5-12(a)和(b)显示了图 5-9 所示的深度优先生成树。图中,每个结点的外面都有一个数,它表示按深度优先检索算法访问这个结点的次序,这个数叫作该结点的深度优先数

(DFN)。例如,DFN(1)=1,DFN(4)=2,DFN(8)=10 等。图 5-12 (b)中的实线边构成这个深度优先生成树,这些边是递归遍历顶点的路径,叫作树边,虚线边为回边。

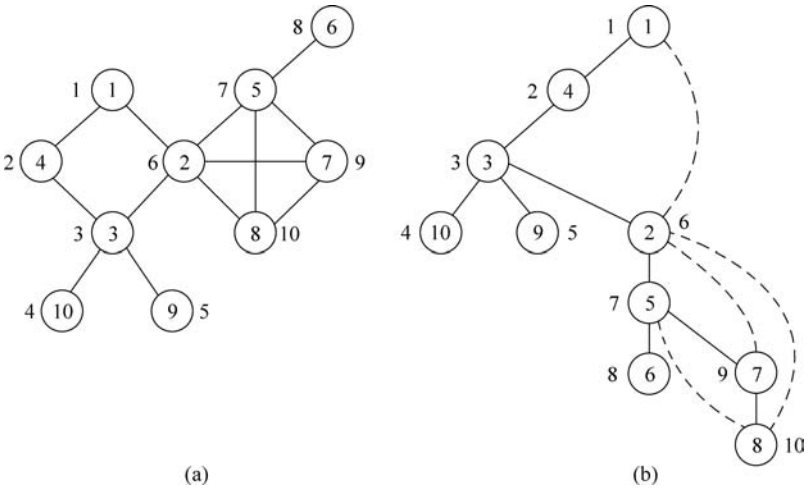

图 5-12　树图

若对图 Graph=(V,{Edge}) 重新定义遍历时的访问数组 Visited 为 DFN,并引入一个新的数组 L,则由一次深度优先遍历便可求得连通图中存在的所有割点。

定义:

$$L[u]=\mathrm{Min}\left\{\mathrm{DFN}[u],L[w],\mathrm{DFN}[k]\ \middle|\ \begin{array}{l}w\text{ 是 }u\text{ 在 DFS 生成树上的孩子结点;}\\k\text{ 是 }u\text{ 在 DFS 生成树上由回边联结的祖先结点;}\\(u,w)\in\text{ 实边;}\\(u,k)\in\text{ 虚边。}\end{array}\right\}$$

显然,$L[u]$是结点 u 通过一条子孙路径且至多关联一条回边,所可能到达的最低深度优先数。如果 u 不是根,那么当且仅当 u 有一个使得 $L[w]\geqslant\mathrm{DFN}[u]$的相邻结点 w 时,u 是一个割点。因为 w 通过一条子孙路径且至多关联一条回边所可能到达的最低深度比 u 的深度优先树还要深,这就意味着去掉 u 后,w 不能有一条路径回到 u 的某个祖先,于是就会产生两个分图,所以 u 就是割点。对于图 5.9(b)所示的生成树,各结点的最低深度优先数是:

$L[1:10]=\{1,1,1,1,6,8,6,6,5,4\}$。

由此,结点 3 是割点,因为它的儿子结点 10 有 $L[10]=4>\mathrm{DFN}[3]=3$。同理,结点 2,5 也是割点。

按后根次序访问深度优先生成树的结点,可以很容易地算出 $L[u]$。于是,为了确定图 G 的割点,必须既完成对 G 的深度优先检索,产生 G 的深度优先生成树 T,又要按后根次序访问树 T 的结点。设计计算 DFN 和 L 的算法 TRY,在图搜索的过程中将两件工作同时完成。

由于计算 $L[u]$与 u 结点的父或子结点有关,所以不同于一般深度优先图搜索的函数,这里 TRY 函数有两个参数,一个是深度优先搜索起点结点 u,另一个是它的父亲 v。设置数组 DFN 为全局量,并将其初始化为 0,表示结点还未曾搜索过。用变量 num 记录当前结

点的深度优先数,也设置为全局变量,被初始化为 1。变量 n 是 G 的结点数。

算法如下:

```
int  DFN[100] = {0},L[100],num = 1,n;
TRY(int u,int v)
{DFN[u] = num;
 L[u] = num;
 num = num + 1;
 while (每个邻接于 u 的结点 w)
       if (DFN[w] = 0)
       {  TRY(w,u);
          if(L[u]> L[w])
             L[u] = L[w];
       }
       else if (w <> v)
            if (L[u]> DFN[w])
                L[u] = DFN[w];
}
```

算法说明:算法 TRY 实现了对图 G 的深度优先检索;在检索期间,对每个新访问的结点赋予深度优先数;同时对这棵树中每个结点 u 计算了 $L[u]$的值。

由算法可以看出,在结点 u 检测完毕返回时,$L[u]$已正确地算出。要指出的是,如果 $w \neq v$,则(u,w)是一条回边或者 DFN$[w]$>DFN$[u]$≥$L[u]$。在这两种情况下 $L[u]$都能得到正确修正。

一旦算出 $L[1:n]$,G 的割点就能在$O(n)$时间识别出来,请读者自行设计完成。最终,识别割点的总时间不超过 $O(n+e)$。

算法分析:如果连通图 G 有 n 个结点 e 条边,且 G 由邻接表表示,算法的计算时间为 $O(n+e)$。

3. 进一步讨论

为了确定使非重连通图转化为重连通图,必然需要增加边集,去除图中割点。一般方法是找出图 G 的最大重连通子图。$G'=(V',E')$是 G 的最大重连通子图,指的是 G 中再没有这样的重连通子图$G''=(V'',E'')$存在,使得 $V' \subset V''$且 $E' \subset E''$。最大重连通子图称为重连通分图。图 5-10 所示的只有一个重连通分图,即这个图的本身。图 5-9 所示的重连通分图在图 5-13 中列出。

两个重连通分图至多有一个公共结点,且这个结点就是割点。因而可以推出任何一条边不可能同时出现在两个不同的重连通分图中(因为这需要两个公共结点)。选取两个重连通分图中不同的结点连接为边,则生成的新图为重连通的。多个重连通分图的情况以此类推。

使用这个方法将图 5-9 变成重连通图,需要对应于割点 3 增加边(4,10)和(10,9);对应割点 2 增加边(1,5);对应割点 5 增加(6,7),结果如图 5-14 所示。

重连通分图的生成和存储比较复杂这里就不深入讨论了。

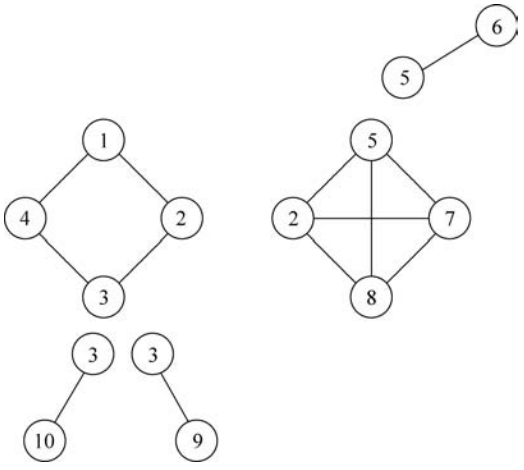

图 5-13　图 5-9 所示的重连通分图

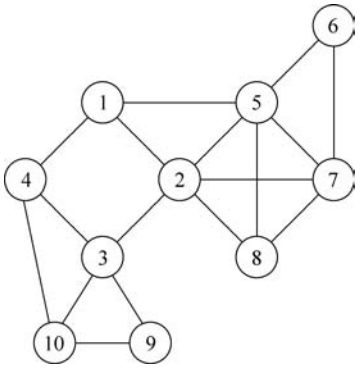

图 5-14　图 5-9 改进为重连通图

5.4　回溯法

在讲解递归算法时已提到回溯(backtracking)这个词,其意义是在递归直到可解的最小问题后,逐步返回原问题的过程。而这里所说的回溯算法实际是一个类似枚举的搜索尝试方法,它的主题思想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”返回,尝试别的路径。

第 4 章中介绍的基础算法中的贪婪算法、动态规划等都具有“无后效性”,也就是在分段处理问题时,某阶段状态一旦确定,将不再改变。而多数问题很难找到“无后效性”的阶段划分和相应决策,是通过深入搜索尝试和回溯操作完成的。

回溯算法是尝试搜索算法中最为基本的一种算法,其采用了一种“走不通就掉头”的思想,作为其控制结构。在使用回溯算法解决问题中每向前走一步都有很多路径需要选择,但当没有决策信息或决策信息不充分时,只好尝试某一路径向下走,直至走到一定程度后得知此路不通时,再回溯到上一步尝试其他路径;在尝试成功时,则问题得解而算法结束。

5.4.1　认识回溯法

下面看一个回溯算法的经典例题。

【例 6】 8 皇后问题:要在 8×8 的国际象棋棋盘中放 8 个皇后,使任意两个皇后都不能互相吃掉。规则是皇后能吃掉同一行、同一列、同一对角线的任意棋子。图 5-15 为一种方案,求所有的解。

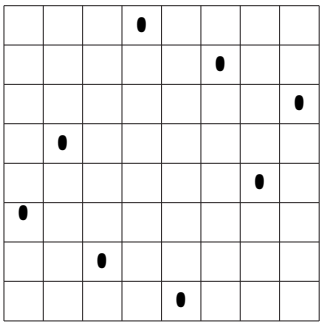

图 5-15　8 皇后问题

模型建立:不妨设 8 个皇后为 x_i,她们分别在第 i 行($i=1,2,3,\cdots,8$),这样问题的解空间,就是一个 8 个皇后所在列的序号,为 n 元一维向量($x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8$),搜索空间是 $1\leqslant x_i\leqslant 8(i=1,2,3,\cdots,8)$,共 8^8 个状态。

约束条件是8个点$(1,x_1),(2,x_2),(3,x_3),(4,x_4),(5,x_5),(6,x_6),(7,x_7),(8,x_8)$不在同一列和同一对角线上。

虽然问题共有8^8个状态,但算法不会真正地搜索这么多的状态,因为前面已经说明,回溯法采用的是"走不通就掉头"的策略,而形如$(1,1,x_3,x_4,x_5,x_6,x_7,x_8)$的状态共有$8^6$个,由于1,2号皇后在同一列不满足约束条件,回溯后这8^6个状态是不会搜索的。

算法设计1:加约束条件的枚举算法。

最简单的算法就是通过八重循环模拟搜索空间中的8^8个状态,按深度优先思想,从第一个皇后开始搜索,确定一个位置后,再搜索第二个皇后的位置……;每前进一步检查是否满足约束条件,不满足时,用continue语句回溯到上一个皇后,继续尝试下一位置;满足约束条件时,开始搜索下一个皇后的位置,直到找出问题的解。

约束条件有3个。

(1) 不在同一列的表达式为:$x_i \neq x_j$;

(2) 不在同一主对角线上的表达式为:$x_i - i \neq x_j - j$;

(3) 不在同一负对角线上的表达式为:$x_i + i \neq x_j + j$。

条件(2),(3)可以合并为一个"不在同一对角线上"的约束条件,表示为:

abs(xi－xj) ≠abs(i－j)(abs( )取绝对值)。

算法1如下:

```
main( )
{int a[9];
  for (a[1] = 1; a[1]<= 8; a[1] = a[1] + 1)
    for (a[2] = 1; a[2]<= 8; a[2] = a[2] + 1)
      {if (check(a,2) = 0) continue;
        for (a[3] = 1; a[3]<= 8; a[3] = a[3] + 1)
             {if (check(a,3) = 0) continue;
              for (a[4] = 1; a[4]<= 8; a[4] = a[4] + 1)
                 {if (check(a,4) = 0) continue;
                  for (a[5] = 1; a[5]<= 8; a[5] = a[5] + 1)
                   {if (check(a,5) = 0) continue;
                     for (a[6] = 1; a[6]<= 8; a[6] = a[6] + 1)
                       {if (check(a,6) = 0) continue;
                        for (a[7] = 1; a[7]<= 8; a[7] = a[7] + 1)
                            {if (check(a,7) = 0) continue;
                              for (a[8] = 1; a[8]<= 8; a[8] = a[8] + 1)
                                   {if (check(a,8) = 0)
                                       continue;
                                   else
                                     for (i = 1; i <= 8; i = i + 1)
                                         print(a[i]);
                                   } } } } } } } }
check(int a[ ],int n)
{   int i;
   for (i = 1; i <= n - 1; i = i + 1)
     if (abs(a[i] - a[n]) = abs(i - n)) or (a[i] = a[n])
        return(0);
   return(1);
}
```

算法分析:前面已经分析过,表面看算法的时间复杂度为8^8,其实算法在执行中不断运

行到 continue 语句回溯，所以算法的复杂度远远低于 8^8。若将算法中，循环嵌套间检查是否满足约束条件的语句：

```
"if(check(a[],i)=0)continue; "    i=2,3,4,5,6,7;
```

语句都去掉，只保留最后一个检查语句：

```
"if(check(a[],8)=0)continue; "
```

相应地将 check( )函数修改成：

```
_check(int a[],int n)
{int i,j;
 for (i=2; i<=n; i=i+1)
    for (j=1; j<=i-1; j=j+1)
      if (abs(a[i]-a[j])=abs(i-j)) or(a[i]=a[j])
         return(0);
 return(1);
}
```

则算法退化成完全的盲目搜索，算法复杂度就是 8^8 了。

算法设计 2：非递归回溯算法。

以上枚举算法有很好的可读性，读者可以从中体会到回溯的思想。不过它只能解决皇后“个数为常量”的问题，却不能解决任意的 n 皇后问题，因此也不是典型的回溯算法模型。下面的非递归算法可以说是典型的回溯算法模型。算法思想同上，用深度优先搜索，并在不满足约束条件时及时回溯。

算法 2 如下：

```
int a[20],n;
main( )
   { input(n);
     backdate(n);
  }
backdate (int n)
  {int k;
   a[1]=0;
   k=1;
   while(k>0)
      {a[k]=a[k]+1;
       while ((a[k]<=n) and (check(k)=0))     // 为第 k 个皇后搜索位置
          a[k]=a[k]+1;
       if(a[k]<=n)
            if (k=n)                  // 找到一组解
              output();
            else
              {k=k+1;                 // 前 k 个皇后找到位置，继续为第 k+1 个皇后找到位置
               a[k]=0; }              // 注意下一个皇后一定要从头开始搜索
       else
            k=k-1;                    // 回溯
       }
  }
```

```
check(int k)
  {int i;
   for (i = 1; i <= k - 1; i = i + 1)
     if (abs(a[i] - a[k]) = abs(i - k)) or(a[i] = a[k])
        return(0);
   return(1);
   }
output( )
  {int i;
   for (i = 1; i <= n; i = i + 1)
      print(a[i]);
}
```

【提示】 与枚举算法比较,此算法也是对 n 个数组元素分别从 1 到 n 进行尝试。可以认为:"数组+循环控制+回溯"实现了任意 n 层循环嵌套的功能。

算法设计 3:递归算法。

对于回溯算法,更方便地是用递归控制方式实现,用这种方式也可以解决任意的 n 皇后问题。算法思想同样用深度优先搜索,并在不满足约束条件时及时回溯。

与上面的两个算法不同,都是用 check( )函数来检查当前状态是否满足约束条件,由于递归调用、回溯的整个过程是非线性的,用 check( )函数来检查当前状态是否满足约束条件是不充分的,而用_check( )函数(在算法分析 1 中说明)来检查当前状态是否满足约束条件又有太多的冗余。

这里用 3.2.3 节"数组记录状态信息"的技巧,用 3 个数组 c,b,d 分别记录棋盘上的 n 个列、$2n-1$ 个主对角线和 $2n-1$ 个负对角线的占用情况。

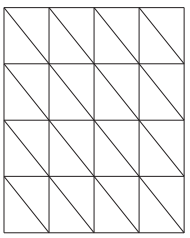

图 5-16 4 皇后棋盘示意图

以 4 阶棋盘为例,如图 5-16 所示,共有 $2n-1=7$ 个主对角线,对应地也有 7 个负对角线。

用 i,j 分别表示皇后所在的行列(或者说 i 号皇后在 j 列),同一主对角线上的行列下标的差一样,若用表达式 $i-j$ 编号,则范围为 $-n+1\sim n-1$,所以用表达式 $i-j+n$ 对主对角线编号,范围就是 $1\sim 2n-1$。同样地,负对角线上行列下标的和一样,用表达式 $i+j$ 编号,则范围为 $2\sim 2n$。

算法 3 如下:

```
int a[20],b[20],c[40],d[40],n,t,i,j,k;                    // t 记录解的个数
main( )
  {
    input(n);
    for (i = 1; i <= n; i = i + 1)
     { b[i] = 0;
       c[i] = 0; c[n + i] = 0;
       d[i] = 0; d[n + i] = 0; }
    try(1);
    }
  output( )
  { t = t + 1;
    print(t,'');
```

```
    for(k = 1; k <= n; k = k + 1)
      print(a[k],'');
    print(" 换行符");
  }
try(int i);
{int j;
 for (j = 1; j <= n; j = j + 1)                          // 第 i 个皇后有 n 种可能位置
     if (b[j] = 0) and (c[i + j] = 0) and (d[i - j + n] = 0)// 判断位置是否冲突
            {a[i] = j;                                   // 摆放皇后
             b[j] = 1;                                   // 占领第 j 列
             c[i + j] = 1;                               // 占领两个对角线
             d[i - j + n] = 1;
             if (i < n)
                try(i + 1);                              // n 个皇后没有摆完,递归摆放下一个皇后
             else
                output( );                               // 完成任务,打印结果
             b[j] = 0;                                   // 回溯
             c[i + j] = 0;
             d[i - j + n] = 0;
             }
}
```

算法说明：递归算法的回溯是由函数调用结束自动完成的，不需要指出回溯点(类似算法 2 中的 $k=k-1$)，但需要“清理现场”——将当前点占用的位置释放，也就是算法 try( ) 中的后 3 个赋值语句。

5.4.2　回溯算法框架

1. 回溯法基本思想

回溯法是在包含问题的所有解的解空间树(或森林)中，图 5-17 是 4 皇后问题的解空间

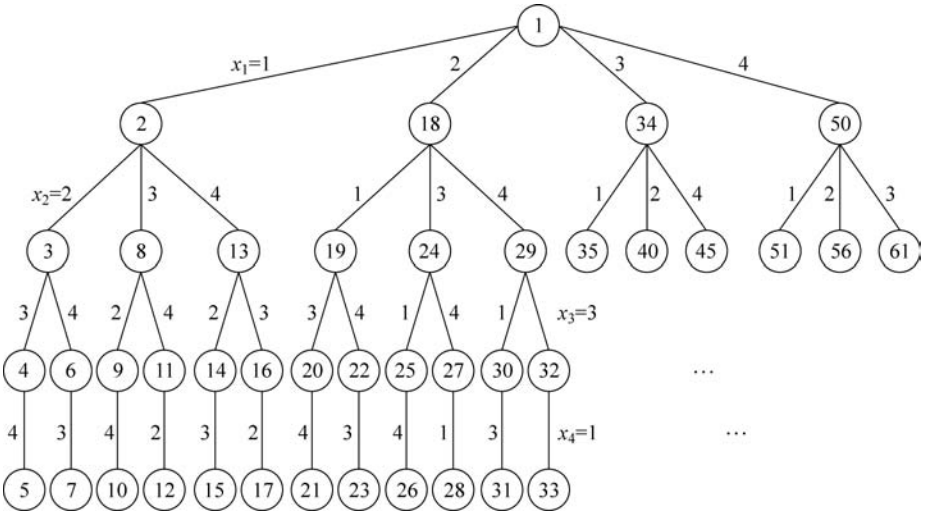

图 5-17　4 皇后问题的解空间树

树。按照深度优先的策略,从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时,总是先判断该结点是否满足问题的约束条件。如果满足进入该子树,继续按深度优先的策略进行搜索。否则,不去搜索以该结点为根的子树,而是逐层向其祖先结点回溯。其实回溯法就是对隐式图的深度优先搜索算法加约束条件。图 5-18 是 4 皇后问题的搜索过程。

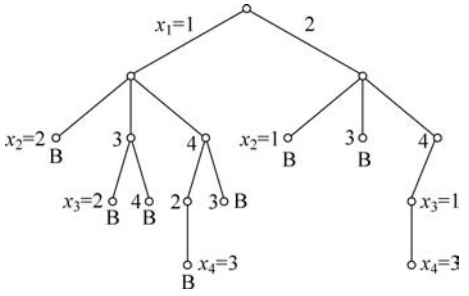

图 5-18 4 皇后问题的部分搜索过程(B 表示失败回溯)

回溯法在用来求问题的所有解时,要回溯到根,且根结点的所有可行的子树都已被搜索遍才结束。而回溯法在用来求问题的任一解时,只要搜索到问题的一个解就可以结束。这就是以深度优先的方式系统地搜索问题解的回溯算法,它适用于解决一些类似 n 皇后问题等求解方案问题,也可以解决一些最优化问题。

2. 算法设计过程

(1) 确定问题的解空间。

应用回溯法解问题时,首先应明确定义问题的解空间。问题的解空间应至少包含问题的一个(最优)解。

(2) 确定结点的扩展规则,如每个皇后在一行中的不同位置移动,而象棋中的马只能走“日”字等。

(3) 搜索解空间。

回溯算法从开始结点(根结点)出发,以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点,同时也成为当前的扩展结点。在当前的扩展结点处,搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点,并成为当前扩展结点。如果在当前的扩展结点处不能再向纵深方向移动,则当前扩展结点就成为死结点。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。回溯法即以这种工作方式递归地在解空间中搜索,直至找到所要求的解或解空间中已没有活结点时为止。

3. 算法框架

通过 n 皇后问题的学习,对回溯算法,以及它们所能解决的问题类型都已经理解,为了更好地认识回溯算法,并利用它解决问题,下面抽象地给出算法框架。

1) 问题框架

设问题的解是一个 n 维向量$(a_1,a_2,\cdots,a_n)$,约束条件是 $a_i(i=1,2,3,\cdots,n)$满足某种条件,记为 $f(a_i)$。

2）非递归回溯框架

```
int a[n],i;
初始化数组 a[ ];
i = 1;
While (i > 0(有路可走)) and ([未达到目标])          // 还未回溯到头
  {if (i > n)                                       // 搜索到叶结点
    搜索到一个解,输出;
  else                                              // 正在处理第 i 个元素
    {a[i]第一个可能的值;
      while (a[i]在不满足约束条件且在搜索空间内)
            a[i]下一个可能的值;
      if (a[i]在搜索空间内)
            {标识占用的资源;
            i = i + 1; }                            // 扩展下一个结点
      else
            {清理所占的状态空间;                    // 回溯
            i = i - 1; }
      }
}
```

3）递归算法框架

回溯法是对解空间的深度优先搜索，在一般情况下用递归函数来实现回溯法比较简单，其中 i 为搜索深度。框架如下：

```
int a[n];
try(int i)
{if (i > n)
    输出结果;
else
    for(j = 下界; j <= 上界; j = j + 1)              // 枚举 i 所有可能的路径
        if(f(j))                                    // 满足限界函数和约束条件
          { a[i] = j;
            …                                       // 其他操作
            try(i + 1);
            回溯前的清理工作(如 a[i]置空值等);
          }
}
```

5.4.3 应用 1——基本的回溯搜索

【例 7】 象棋中马遍历棋盘的问题。

问题描述：在 $n \times m$ 的棋盘中，马只能走日字。马从位置(x,y)处出发，把棋盘的每一点都走一次，且只走一次，找出所有路径。

问题分析：此问题为博弈算法的基础。马是在棋盘的点上行走的，所以这里的棋盘是指行有 n 条边、列有 m 条边。而一个马在不出边界的情况下有 8 个方向可以行走（走日字），如当前坐标为(x,y)则行走后的坐标可以为：

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

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

算法设计：搜索空间是整个 $n\times m$ 个棋盘上的点。约束条件是不出边界且每个点只经过一次。结点的扩展规则如问题分析中所述。

搜索过程是从任一点(x,y)出发，按深度优先的原则，从 8 个方向中尝试一个可以走的点，直到走过棋盘上所有 $n\times m$ 个点。用递归算法易实现此过程。

注意：问题要求找出全部可能的解，就要注意回溯过程的清理现场工作，也就是置当前位置为未经过。

数据结构设计：

(1) 用一个变量 dep 记录递归深度，也就是走过的点数，当 dep$=n\times m$ 时，找到一组解。

(2) 用 $n\times m$ 的二维数组记录马行走的过程，初始值为 0 表示未经过。搜索完毕后，起点存储的是 1，终点存储的是 $n\times m$。

算法如下：

```
int n=5,m=4;
int fx[8]={1,2,2,1,-1,-2,-2,-1},fy[8]={2,1,-1,-2,-2,-1,1,2},a[5][4];   // 下标从1开始
int dep,x,y,count;
main( )
  { int i,j;
    count=0;
    dep=1;
    print("input x,y");
    input(x,y);
    if (x>n or y>m or x<1 or y<1)
       { print("x,y error!");
         return; }
    for(i=1; i<=n; i=i+1)
      for(j=1; j<=m; j=j+1)
        a[i][j]=0;
    a[x][y]=1;
    find(x,y,2);
    if (count=0)
      print("Non solution!");
    else
      print("count=",count);
    }
  find(int x,int y,int dep)
    {int i,xx,yy;
     for(i=1; i<=8; i=i+1)                    // 加上方向增量,形成新的坐标
      {xx=x+fx[i];
       yy=y+fy[i];
       if (check(xx,yy)=1)                    // 判断新坐标是否出界,是否已走过
        {a[xx][yy]=dep;                       // 走向新的坐标
         if (dep=n*m)
           output( );
         else
           find(xx,yy,dep+1);                 // 从新坐标出发,递归下一层
```

```
            a[xx][yy] = 0;                          // 回溯,恢复未走标志
            }
        }
    }
output( )
{count = count + 1;
print("换行符");
print("count = ",count);
for(y = 1; y < = n; y = y + 1)
    {print("换行符");
    for(x = 1; x < = m; x = x + 1)
        print(a[y][x]);
    }
}
```

【提示】 请读者仿照前面的例题自己编写判断新坐标是否出界,是否已走过的函数check(x,y)。

【例 8】 素数环问题。

把从 1～20 这 20 个数摆成一个环,要求相邻的两个数的和是一个素数。

算法设计:非常明显,这是一道需要进行尝试的题目。而且是在必要时必须进行回溯。尝试搜索从 1 开始,每个空位有 2～20 共 19 种可能,约束条件就是填进去的数满足以下两条。

(1) 与前面的数不相同(填入的数据不重复);

(2) 与前面相邻数据的和是一个素数。

特别注意,第 20 个数还要判断和第 1 个数的和是否素数。

根据模块化的思想,重复判断数据,判断相邻数据的和是否是素数,分别编写成独立的函数。

算法如下:

```
main( )
{int a[20],k;                                       // 下标从 1 开始
 for (k = 1; k < = 20; k = k + 1)
     a[k] = 0;
 a[1] = 1;
 try(2);
 }
 try(int i)
 {int k
 for (k = 2; k < = 20; k = k + 1)
      if (check1(k,i) = 1 and check3(k,i) = 1)
            {a[i] = k;
             if (i = 20)
                output( );
             else
                {try(i + 1);
                 a[i] = 0; }
             }
```

```
 }
check1(int j,int i)
  {int k;
   for (k = 1; k <= i - 1; k = k + 1)
      if (a[k] = j)
        return(0);
    return(1);
    }
check2(int x)
{int k,n;
 n = sqrt(x);                                    // sqrt()开平方
 for (k = 2; k <= n; k = k + 1)
    if (x mod k = 0)
       return(0);
 return(1);
 }
check3(int j,int i)
{ if (i<20)
      return(check2(j + a[i - 1]));
  else
      return(check2(j + a[i - 1]) and check2(j + a[1]));
  }
  output( )
  {int k;
   for (k = 1; k <= 20; k = k + 1)
       print(a[k]);
   print("换行符");
   }
```

算法说明：这个算法中也要注意在回溯前要“清理现场”，也就是置 $a[i]$为 0。

回溯算法不仅能搜索问题的解，也可能构造问题的解，看以下例子。

【例 9】 找 n 个数中 r 个数的组合。

算法设计：3.1.3 节例 9 中曾利用循环嵌套和递归控制结构解决过这个问题，这里再用回溯法来构造自然数的各种组合。

问题的解空间为一组 r 元一维向量，$(a_1,a_2,\cdots,a_r)$，$1\leqslant a_i\leqslant n$，$1\leqslant i\leqslant r$。用一维数组 a 存储正在搜索的向量。

怎样进行搜索？怎样表示搜索过程中的约束条件？还是通过实例来归纳这些算法的要点。仍以 $n=5$，$r=3$ 为例，组合结果如下：

```
5    4    3
5    4    2
5    4    1
5    3    2
5    3    1
5    2    1
4    3    2
4    3    1
```

4　　2　　1

3　　2　　1

分析数据的特点，搜索时依次对数组(一维向量)元素 $a[1]$，$a[2]$，$a[3]$进行尝试：

$a[ri]i1\sim i2$，

$a[1]$尝试范围 $5\sim3$，

$a[2]$尝试范围 $4\sim2$，

$a[3]$尝试范围 $3\sim1$，

且有这样的规律："后一个元素至少比前一个数小 1"；$ri+i2$ 均为 $4=r+1$，$ri+i1$ 均为 $6=n+1$。

归纳为一般情况：

$a[1]$尝试范围 $n\sim r$，$a[2]$尝试范围 $n-1\sim r-1$，…，$a[r]$尝试范围 $n-r+1\sim1$。

由此，搜索过程中的约束条件为 $ri+a[ri]\geqslant r+1$，若 $ri+a[ri]<r+1$ 就要回溯到元素 $a[ri-1]$搜索，特别地 $a[r]=1$ 时，回溯到元素 $a[r-1]$搜索。

算法如下：

```
main( );
  {int n,r,a[20];
   print("n,r = ");
   input(n,r);
   if (r > n)
     print("Input n,r error!");
   else
     comb(n,r,a);
   }
comb(int n,int r,int a[])
    {int i,ri;
     ri = 1; a[1] = n;
     while (a[1]> r - 1)
       if (ri < r)                          // 没有搜索到底
        {if (ri + a[ri]> = r + 1)           // 是否回溯
           {a[ri + 1] = a[ri] - 1;
            ri = ri + 1; }
         else
             {ri = ri - 1;
              a[ri] = a[ri] - 1; }          // 回溯
       else
        { for(j = 1; j < = r; j = j + 1)    // 输出组合数
              print(a[j]);
          print("换行符");
          if (a[r] = 1)                     // 是否回溯
            {ri = ri - 1;
              a[ri] = a[ri] - 1; }          // 回溯
          else
              a[ri] = a[ri] - 1;            // 搜索到下一个数
          }
  }
```

5.4.4 应用2——排列及排列树的回溯搜索

有一类问题是应该在排列树中进行搜索的,而 n 个数据的排列是不易枚举的。如8皇后问题中,要求各行8皇后不同列,则8皇后的列号就是 n 个数据的一个排列,问题的解可以在 $n!$ 个排列中搜索。而在5.4.1节中是通过枚举了每个皇后可能的8个位置,在 8^8 的空间中,并将不同列作为约束条件进行搜索的,这样的效率当然要低一些。5.4.3节的例8也是如此。

本小节先学习用回溯法构造 n 个数据的不同排列,再介绍如何对排列树进行搜索。

下面的例子类似5.4.1节中8皇后问题的算法3,也在 n^n 的空间中进行搜索,并用数组记录前面数据的排列情况。

【例10】 输出自然数1～n 所有不重复的排列,即 n 的全排列。

算法设计:n 的全排列是一组 n 元一维向量,$(x_1,x_2,x_3,\cdots,x_n)$,搜索空间是:

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

约束条件很简单,x_i 互不相同。

怎样在搜索过程中检查是否满足约束条件呢?一般的方法是与本节例7一样设计check( )函数进行判断,check( )函数中当前元素与前面的元素进行逐个比较。这里采用第3章3.2.3"数组记录状态信息"的技巧,设置 n 个元素的一维数组 d,其中的 n 个元素用来记录数据1～n 的使用情况,已使用置1,未使用置0。

算法如下:

```
int p = 0,n,a[100],d[100];
main( )
{int j;
 print('Input n = ');
 input(n);
 for(j = 1; j <= n; j = j + 1)
   d[j] = 0;
 try(1);
}
try(int k)
    {int j;
     for(j = 1; j <= n; j = j + 1)
       {if (d[j] = 0)
         {a[k] = j;
           d[j] = 1; }
        else
          continue;
        if (k < n)
            try(k + 1);
        else
            {p = p + 1;
             output(); }
        d[a[k]] = 0;
      }
    }
```

```
output( )
  {int j;
   print(p,": ")
   for(j = 1; j <= n; j = j + 1)
      print(a[j]);
   print("换行符");
}
```

算法说明：变量 p 记录排列的组数，k 为当前处理的第 k 个元素。

算法分析：用以上回溯搜索算法完成的全排列问题的复杂度为 $O(n^n)$，不是一个好的算法。因此不可能用它的结果去搜索排列树。下面的全排列算法的复杂度为 $O(n!)$，其结果可以为搜索排列树所用。

【例 11】 全排列算法另一解法——也是搜索排列树的算法框架。

算法设计：根据全排列的概念，定义数组初始值为(1,2,…,n)，这是全排列中的一种结果，然后通过数据间的交换，则可产生所有的不同排列。

算法如下：

```
int a[100],n,s = 0;
main( )
{int i;
 input(n);
 for(i = 1; i <= n; i = i + 1)
    a[i] = i;
 try(1);
 print("换行符","s = ",s);
}
try(int t)
{int j;
 if (t > n)
   {output( ); }
 else
    for(j = t; j <= n; j = j + 1)
    {swap(t,j);
     try(t + 1);
     swap(t,j);                                  // 回溯时,恢复原来的排列
    }
 }
output( )
{int j;
 print("换行符");
 for(j = 1; j <= n; j = j + 1)
   print(a[j]);
s = s + 1;
}
swap(int t1,int t2)
{int t;
 t = a[t1];
 a[t1] = a[t2];
 a[t2] = t; }
```

算法说明：

(1) 有的读者可能会想 try( )函数中，不应该出现自身之间的交换，for 循环是否应该改为“for(j=t+1; j<=n; j=j+1)”？回答是否定的。以实例说明，当 $n=3$ 时，算法的输出是：123,132,213,231,321,312。排列 123 的输出说明第一次到达叶结点是不经过数据交换的，排列 132 中的 1 也是不进行交换的结果。

(2) for 循环体中的第二个 swap( )调用，是用来恢复原顺序的，这就是前面提到的回溯还原操作。为什么要有如此操作呢？还是通过实例进行说明，排列“132”是由“123”进行 2，3 交换得到的，同样排列“213”是由“123”进行 1,2 交换得到的，所以在每次回溯时，都要恢复本次操作前的原始顺序。

【提示】 对此算法不仅仅要理解，而且要记忆，因为它是解决所有与排列有关问题的算法框架。

【例 12】 按排列树搜索解决 8 皇后问题。

算法设计：利用例 11“枚举”出的所有 1～n 排列，从中选出满足约束条件的解来。这时约束条件就只有“皇后不在同一对角线上”，而不需要“皇后在不同列”的约束条件了。

和例 6 的算法 3 一样，用数组 c，d 记录每个对角线的占用情况。

算法如下：

```
int a[100],n,s = 0,c[20],d[20];
main( )
{int i;
 input(n);
 for(i = 1; i <= n; i = i + 1)
  a[i] = i;
 for (i = 1; i <= n; i = i + 1)
 {  c[i] = 0;
    c[n + i] = 0;
    d[i] = 0;
    d[n + i] = 0; }
 try(1);
 print("s = ",s);
 }
 try(int t)
 {int j;
 if (t > n)
   {output( ); }
 else
  for(j = t; j <= n; j = j + 1)
  {swap(t,j);
   if (c[t + a[t]] = 0 and d[t - a[t] + n] = 0)    // 是否满足不在同一对角线
     {c[t + a[t]] = 1;
      d[t - a[t] + n] = 1;
      try(t + 1);
      c[t + a[t]] = 0;
      d[t - a[t] + n] = 0;
     }
   swap(t,j);
```

```
  }
}
output( )
{int j;
print("换行符");
for(j = 1; j <= n; j = j + 1)
    print(a[j]);
s = s + 1;
}
swap(int t1,int t2)
{int t;
 t = a[t1];
 a[t1] = a[t2];
 a[t2] = t; }
```

【提示】 类似地，按排列树搜索方法也可以解决例 8 素数环等问题，请大家尝试完成。对于解空间为排列树的最优化问题，5.4.5 节介绍，基本算法框架是一样的。

5.4.5　应用 3——最优化问题的回溯搜索

前面的例题都是在回溯搜索过程中找解决问题的方案，这一节学习用回溯算法解决最优化问题。直观的方法就是在搜索全部解的过程中经过比较，从而确定和保留最优解。如下面的例子。

【例 13】 一个有趣的高精度数据：构造一个尽可能大的数，使其从高到低满足前一位能被 1 整除，前 2 位能被 2 整除，……，前 n 位能被 n 整除。

数学模型：记高精度数据为 $a_1,a_2,\cdots,a_n$，题目很明确有两个要求：

(1) a_1 能被 1 整除且$(a_1\times10+a_2)$ 能被 2 整除且……$(a_1\times10^{n-1}+a_2\times10^{n-2}+\cdots+a_n)$能被 n 整除；

(2) 求最大的这样的数。

算法设计：此数只能用从高位到低位逐位尝试，失败回溯的算法策略求解，生成的高精度数据用数组从高位到低位存储，1 号元素开始存储最高位。此数的大小无法估计不妨为数组开辟 100 个空间。

算法中数组 A 为当前求解的高精度数据的暂存处，数组 B 为当前最大的满足条件的数。

算法的首位 $A[1]$(最高位)从 1 开始枚举。以后各位从 0 开始枚举。所以求解出的满足条件的数据之间只需要比较位数就能确定大小。n 为当前满足条件的最大数据的位数，i 为当前满足条件数据的位数，当 $i\geqslant n$ 就认为找到了更大的解。当 $i>n$ 不必解释，位数多数据一定大；$i=n$ 时，由于尝试是由小到大进行的，虽然位数相等，但后来满足条件的数据一定比前面的大。

算法细节在算法说明中解释。

算法如下：

```
main( )
{int A[101],B[101];
```

```
    int i,j,k,n,r;
    A[1] = 1;
    for(i = 2; i <= 100; i = i + 1)                    // 置初值: 首位为 1,其余为 0
        A[i] = 0;
    n = 1;
    i = 1;
    while(A[1]<= 9)
      {if (i >= n)                                     // 发现有更大的满足条件的高精度数据
        {n = i;                                        // 转存到数组 B 中
         for (k = 1; k <= n; k = k + 1)
           B[k] = A[k];
         }
      i = i + 1;
      r = 0;
      for (j = 1; j <= i; j = j + 1)                   // 检查第 i 位是否满足条件
        {r = r * 10 + A[j];
         r = r mod i; }
      if (r <> 0)                                      // 若不满足条件
        {A[i] = A[i] + i - r;                          // 第 i 位可能的解
         while (A[i]> 9 and i > 1)                     // 搜索完第 i 位的解,回溯到前一位
             {A[i] = 0;
              i = i - 1;
              A[i] = A[i] + i; }
        }
      }
}
```

算法说明:

(1) 从 $A[1]=1$ 开始,每增加一位 $A[i]$(初值为 0)先计算 $r=(A[1]\times 10^{i-1}+A[2]\times 10^{i-2}+\cdots+A[i])$,再测试 $r=r \bmod i$ 是否为 0。

(2) $r=0$ 表示增加第 i 位后,满足条件,与原有满足条件的数(存在数组 B 中)比较,若前者大,则更新后者(数组 B),继续增加下一位。

(3) $r\neq 0$ 表示增加 i 位后不满足整除条件,接下来算法中并不是继续尝试 $A[i]=A[i]+1$,而是继续尝试 $A[i]=A[i]+i-r$,因为若 $A[i]=A[i]+i-r\leqslant 9$ 时,$(A[1]\times 10^{i-1}+A[2]\times 10^{i-2}+\cdots+A[i]-r+i)\bmod i$ 肯定为 0。这样可减少尝试次数。如:17 除 5 余 2,17−2+5 肯定能被 5 整除。

有读者肯定会想$(A[1]\times 10^{i-1}+A[2]\times 10^{i-2}+\cdots+A[i]-r)\bmod i$ 也肯定为 0,为什么不尝试 $A[i]=A[i]-r$,因 $A[i]$初值为 0,这样必然要退(借)位,就会破坏前 $i-1$ 位已满足条件的前提。

(4) 同理,当 $A[i]-r+i>9$ 时,要进位也不能算满足条件。这时,只能将此位恢复初值 0 且回退到前一位($i=i-1$),尝试 $A[i]=A[i]+i$,以此类推……。这正是算法中最后一个 while 循环所做的工作。

(5) 当回溯到 $i=1$ 时,$A[1]$加 1 开始尝试首位为 2 的情况,最后直到将 $A[1]=9$ 的情况尝试完毕,算法结束。

【提示】 最优化问题一般都可以通过找出全部解方法,从而确定其中的最优解,但这样

做算法的效率肯定得不到保证。在搜索解的过程中，可以根据前期解的信息，确定某些分支一定达不到最优解而不去搜索这些分支，以提高算法效率。

就本题而言可以考虑每一位都从 9 开始尝试，这样只有搜索到更多位数的解时，才算找到了更大的解，可以减少记录当前最大解的操作。

【例 14】 流水作业车间调度。

n 个作业{1,2,…,n}要在由两台机器 M_1 和 M_2 组成的流水线上完成加工。每个作业加工的顺序都是先在 M_1 上加工，然后在 M_2 上加工。M_1 和 M_2 加工作业 i 所需的时间分别为 a_i 和 b_i。流水作业调度问题要求确定这 n 个作业的最优加工顺序，使得从第一个作业在机器 M_1 上开始加工，到最后一个作业在机器 M_2 上加工完成所需的时间最少。作业在机器 M_1 和 M_2 的加工顺序相同。

算法设计：

(1) 问题的解空间是一棵排列树，简单的解决方法就是在搜索排列树的同时，不断更新最优解，最后找到问题的解。算法框架和例 11 完全相同，用数组 x(初值为 1,2,3,…,n)模拟不同的排列，在不同排列下计算加工耗时情况。

(2) 机器 M_1 进行顺序加工，其加工 f_1 时间是固定的，$f_1[i]=f_1[i-1]+a[x[i]]$。机器 M_2 则有可能空闲(如图 5-19(a)所示)或积压(如图 5-19(b)所示)的情况，总加工时间 f_2，当机器 M_2 空闲时，$f_2[i]=f_1[i]+b[x[i]]$；当机器 M_2 有积压情况出现时，$f_2[i]=f_2[i-1]+b[x[i]]$。总加工时间就是 $f_2[n]$。

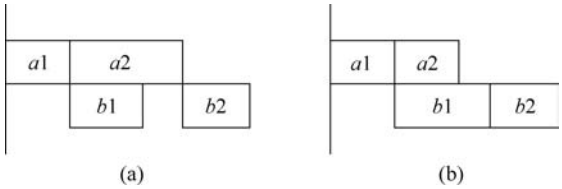

图 5-19 加工顺序

(3) 一个最优调度应使机器 M_1 没有空闲时间，且机器 M_2 的空闲时间最少。在一般情况下，当作业按在机器 M_1 上由小到大排列后，机器 M_2 的空闲时间较少，当然，最少情况一定还与 M_2 上的加工时间有关。所以，还需要对解空间进行搜索，但排序后可以尽快找到接近最优的解。

(4) 经过排序，就会尽快出现一个接近最优的解，在以后的搜索过程中，当某一排列前几步的加工时间，已经大于当前总加工时间的最小值时，就不进行进一步的搜索计算了，这个操作就称为“限界操作”，它减小了搜索范围，提高了搜索效率。

数据结构设计：

(1) 用二维数组 job[100][2]存储作业在 M_1，M_2 上的加工时间。

(2) 由于 f_1 在计算中，只需要当前值，所以用变量存储即可；而 f_2 在计算中，还依赖前一个作业的数据，所以有必要用数组存储。

算法如下：

```
int job[100][2],x[100],bestx[100],n,f1 = 0,bestf,f2[100] = {0};
main( )
```

```
{int i,j;
input(n);
for(i = 1; i <= 2; i = i + 1)
  for(j = 1; j <= n; j = j + 1)
    input(job[j][i]);
bestf = 32767;
for(i = 1; i <= n; i = i + 1)
    x[i] = i;
try(1);
for(i = 1; i <= n; i = i + 1)
    print(bestx[i]);
print(bestf);
}
try(int i)
{int j;
if (i = n + 1)
    {for(j = 1; j <= n; j = j + 1)
      bestx[j] = x[j];
     bestf = f2[i];
     }
else
  for(j = i; j <= n; j = j + 1)
    {f1 = f1 + job[x[j]][1];
     if(f2[i - 1]> f1)
       f2[i] = f2[i - 1] + job[x[j]][2];
    else
       f2[i] = f1 + job[x[j]][2];
    if (f2[i]< bestf)
        {swap(x[i],x[j]);
         try(i + 1);
         swap(x[i],x[j]); }
    f1 = f1 - job[x[j]][1];
   }
}
```

对解空间为排列树的最优化问题,都可以依此算法解决,且可以仿照本例加入相应的限界策略,提高搜索效率。

而对于解空间为子集树的最优化问题,可以仿照本节例 6、例 7、例 8,枚举问题中各元素的选取和不选取两种情况进行搜索,从中找出问题的最优解。

5.5 分支限界法

在现实生活中,有这样一类问题:问题有 n 个输入,而问题的解就由 n 个输入的某种排列或某个子集构成,只是这个排列或子集必须满足某些事先给定的条件。把那些必须满足的条件称为约束条件;而把满足约定条件的排列或子集称为该问题的可行解。满足约束条件的子集可能不止一个,也就是说可行解一般来说是不唯一的。为了衡量可行解的优劣,事先也可给出一定的标准,这些标准一般以函数形式给出,这些函数称为目标函数。那些使目

标函数取极值的可行解，称为最优解。如工作安排问题，任意顺序都是问题的可行解，人们真正需要的是最省时间的最优解。

用回溯算法解决问题时，是按深度优先的策略在问题的状态空间中，尝试搜索可能的路径，不便于在搜索过程中对不同的解进行比较，只能在搜索到所有解的情况下，才能通过比较确定哪个是最优解。这类问题更适合用广度优先策略搜索，因为在扩展结点时，可以在E-结点的各个子结点之间进行必要的比较，有选择地进行下一步扩展。这里要介绍的分支限界法就是一种较好的解决最优化问题的算法。分支限界法是由“分支”策略和“限界”策略两部分组成。“分支”策略体现在对问题空间是按广度优先的策略进行搜索；“限界”策略是为了加速搜索速度而采用启发信息剪枝的策略。

5.5.1 分支搜索算法

分支搜索法是一种在问题解空间上进行搜索尝试的算法。所谓“分支”是采用广度优先的策略，依次搜索E-结点的所有分支，也就是所有的相邻结点。和回溯法一样，在生成的结点中，抛弃那些不满足约束条件(或者说不可能导出最优可行解)的结点，其余结点加入活结点表。然后从表中选择一个结点作为下一个E-结点，继续搜索。选择下一个E-结点的方式不同，会出现几种不同的分支搜索方式。

1. FIFO搜索

先进先出(first in first out，FIFO)搜索算法要依赖“队”做基本的数据结构。一开始，根结点是唯一的活结点，根结点入队。从活结点队中取出根结点后，作为当前扩展结点。对当前扩展结点，先从左到右地产生它的所有儿子，用约束条件检查，把所有满足约束函数的儿子加入活结点队列中。再从活结点表中取出队首结点(队中最先进来的结点)为当前扩展结点，……，直到找到一个解或活结点队列为空为止。

2. LIFO搜索

后进先出(last in first out，LIFO)搜索算法要依赖“栈”做基本的数据结构。一开始，根结点入栈。从栈中弹出一个结点为当前扩展结点。对当前扩展结点，先从左到右地产生它的所有儿子，用约束条件检查，把所有满足约束函数的儿子入栈，再从栈中弹出一个结点(栈中最后进来的结点)为当前扩展结点，……，直到找到一个解或栈为空为止。

3. 优先队列式搜索

为了加速搜索的进程，应采用有效地方式选择E-结点进行扩展。优先队列(priority queue)式搜索，对每一活结点计算一个优先级(某些信息的函数值)，并根据这些优先级，从当前活结点表中优先选择一个优先级最高(最有利)的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。这种扩展方式要到下一节才会用到。

下面用FIFO搜索方式为例，学习分支搜索算法。

【例15】 布线问题：如图5-20所示，印刷电路板将布线区域划分成 $n \times m$ 个方格。精确的电路布线问题要求确定连接方格a的中点到方格b的中点的最短布线方案。在布线

时,电路只能沿直线或直角布线。为了避免线路相交,已经布线的方格做了封锁标记(图中阴影部分),其他线路不允许穿过被封锁的方格。

算法选择:题目的要求是要找到最短的布线方案,从图 5-20 的情况看,可以用贪婪算法解决问题,也就是从 a 开始朝着 b 的方向垂直布线即可。实际上,再看一下图 5-21,就知道贪婪算法策略是行不通的。因为已布线的方格是没有规律的,所以直观上说只能用搜索方法去找问题的解。

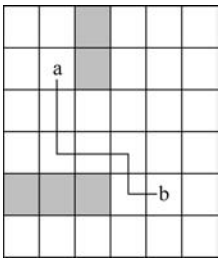

图 5-20 一个布线区域和布线方案

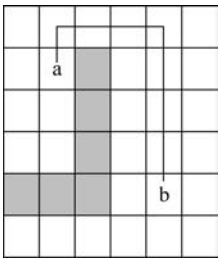

图 5-21 另一个布线区域和布线方案

2	1	2	3	4	5
1	a		4	5	6
2	1		5	6	7
3	2		6	7	8
			7	b	

图 5-22 a 到 b 的扩展活结点的过程

问题分析:根据布线方法的要求,除边界处或已布线处,每个 E-结点分支扩充的方向有 4 个,即上、下、左、右,也就是说一个 E-结点扩充后最多产生 4 个活结点。以图 5-21 的情况为例,图的搜索过程如图 5-22 所示。

搜索以 a 为第一个 E-结点,以后不断扩充新的活结点,直到 b 结束(当然反之也可以)。反过来从 b 到 a,按序号 8-7-6-5-4-3-2-1 就可以找到最短的布线方案。从图中也可以发现最短的布线方案是不唯一的。且由此可以看出,此问题适合用分支搜索方法。

算法设计:

(1) 初始化部分:开辟 $m \times n$ 的二维数组模拟布线区域,初始值均置为 -1,表示没有被使用。已使用的位置,通过键盘输入其下标,将对应值置为 0。输入方格 a,b 的下标,存储在变量中。

(2) 用 FIFO 分支搜索的过程。

① 一开始,唯一的活结点是 a。

② 从活结点表中取出后为当前扩展结点。

③ 对当前扩展结点,按"上、下、左、右"的顺序,找可布线的位置,加入活结点队列中。

④ 再从活结点队列中取出一个结点为当前扩展结点……

⑤ 直到搜索到达 b 结点,然后按序号 8-7-6-5-4-3-2-1 输出最短的布线方案,算法结束。或活结点队列已为空,表示无解,算法结束。

(3) 可布线位置的识别。

可布线位置不能简单地认为是结点为 -1 的点,因为还要考虑数组越界的情况。反过来说不可布线的位置有两种情况。

① 已占用结点，标识为 0；

② 布线区域外的结点，标识比较复杂，是一个含四个关系表达式的逻辑表达式，结点下标(i,j)满足：

```
not (i>0 and i<=m and j>0 and j<=n)
```

第二种情况逻辑表达式较复杂，可以用设置“边界空间”的技巧，把以上两种不可布线的情况，都归纳为第一种情况，如图 5-23 所示，把布线区域扩充为$(m+2)\times(n+2)$数组，边界置占用状态，就无须做越界的检查了。

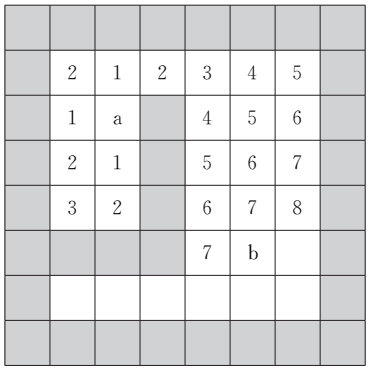

图 5-23　有边界的布线区域图

【提示】 这是一个用空间效率换取时间效率的技巧。

(4) 为了突出算法思想，关于队列的结构类型和操作，只用抽象的符号代替：

teamtype 为队列的类型说明符，具体可以是数组或链表；

inq(temp)结点 temp 入队 q；

outq(q)结点从队列 q 中出队，并返回队首结点；

empty(q)判断队列是否为空，空返回“真”，非空返回“假”。

队列操作的详细实现请参照数据结构中的算法。

算法如下：

```
struct node
 {int x,y;
 } stTRY,end;
int a[100][100];
teamtype q;
main( )
{int i,j,k;
struct node temp;
print("How many row?");
input(m);
print("How many col?");
input(n);
print("input stTRY and end?");
input(stTRY.x,stTRY.y);
input(end.x,end.y);
if (stTRY.x=end.x or stTRY.y=end.y) error( );
if (stTRY.x<1 or stTRY.x>m or stTRY.y<1 or stTRY.y>n) error( );
if (end.x<1 or end.x>m or end.y<1 or end.y>n) error( );
for(i=1; i<=m; i=i+1)
  for(j=1; j<=n; j=j+1)
    a[i][j]=-1;
for(i=1; i<=m; i=i+1)                          // 设置边界
    a[i][0]=a[i][n+1]=0;
for(i=0; i<=n+1; i=i+1)
    a[0][j]=a[m+1][j]=0;
print("input the node of tie-up");
```

```
input(temp.x,temp.y);                                  // 设置占用区域
while(x <> 0)
    {a[x][y] = 0;
     input(temp.x,temp.y); }
k = search( );
if (k = -1)
  print("Non solution");
else
  output(k);
}
search( )
{ struct node temp,temp1;
  inq(stTRY);
  k = 0;
  while(not empty(q))
       {temp = outq(q);
        if (a[temp.x-1][temp.y] = -1)                  // 上
          { temp1.x = temp.x-1;
            temp1.y = temp.y;
            k = k + 1;
            a[temp1.x][temp1.y] = k;
                if (temp1.x = end.x and temp1.y = end.y) return(k);
            inq(temp1);
          }
        if (a[temp.x+1][temp.y] = -1)                  // 下
         { temp1.x = temp.x + 1;
           temp1.y = temp.y;
           k = k + 1;
           a[temp1.x][temp1.y] = k;
               if (temp1.x = end.x and temp1.y = end.y) return(k);
           inq(temp1);
      if (a[temp.x][temp.y-1] = -1)                    // 左
      { temp1.x = temp.x;
        temp1.y = temp.y-1;
        k = k + 1;
        a[temp1.x][temp1.y] = k;
        if (temp1.x = end.x and temp1.y = end.y) return(k);
         inq(temp1);
      }
      if (a[temp.x][temp.y+1] = -1)                    // 右
      { temp1.x = temp.x;
        temp1.y = temp.y+1;
        k = k + 1;
        a[temp1.x][temp1.y] = k;
        if (temp1.x = end.x and temp1.y = end.y) return(k);
        inq(temp1);
      }
    }
return(-1);
}
output(int k)
```

```
{int x,y;
 print("(",end.x,",",end.y,")");
 x = end.x;
 y = end.y;
 while(k > 2)
     {k = k - 1;
      if (a[x - 1][y] = k)
        {x = x - 1;
         print("(",x,",",y,")");
         continue; }
     if (a[x + 1][y] = k)
       {x = x + 1;
        print("(",x,",",y,")");
        continue; }
     if (a[x][y - 1] = k)
       {y = y - 1;
        print("(",x,",",y,")");
        continue; }
     if (a[x][y + 1] = k)
       {y = y + 1;
        print("(",x,",",y,")");
        continue; }
     }
   print("(",stTRY.x,",",stTRY.y,")");
}
```

在回溯算法中，当前的 E-结点记为 F 每次搜索一个儿子 C，这个儿子结点就变成一个新的 E-结点，原来的 E-结点仍为活结点。当完全检测了子树 C 之后 F 结点就再次成为 E-结点，搜索其他儿子。而在 FIFO 分支搜索方法中，在搜索当前 E-结点全部儿子后，其儿子成为活结点，E-结点变为死结点；活结点存储在队列中，队首的活结点出队后变为 E-结点，其再生成其他活结点的儿子……直到找到问题的解或活结点队列为空搜索才完毕。

下面再看一个用 FIFO 分支搜索方法求解最优解的问题。

【例 16】 有两艘船和需要装运的 n 个货箱，第一艘船的载重量是 c_1，第二艘船的载重量是 c_2，w_i 是货箱 i 的重量，且 $w_1+w_2+\cdots+w_n \leqslant c_1+c_2$。希望确定是否有一种可将所有 n 个货箱全部装船的方法。若有，则找出该方法。

问题分析：先看一个实例，当 $n=3$，$c_1=c_2=50$，$w=\{10,40,40\}$时，可将货箱 1，2 装到第一艘船上，货箱 3 装到第二艘船上。但如果 $w=\{20,40,40\}$，则无法将货箱全部装船。由此可知问题可能有解、可能无解，也可能有多解。下面以找出问题的一个解为目标设计算法。

虽然是关于两艘船的问题，其实只讨论一艘船的最大装载问题即可。因为当第一艘船的最大装载为 bestw 时，若 $w_1+w_2+\cdots+w_n-\text{bestw} \leqslant c_2$ 则可以确定一种解，否则问题就无解。这样问题转化为第一艘船的最大装载问题。

算法设计 1：转化为一艘船的最优化问题后，问题的解空间为一个子集树。也就是算法要考虑所有物品取、舍情况的组合，n 个物品的取舍组合共 2^n 次个分支，搜索子集树是 NP-

复杂问题。图 5-24 是 $n=3$ 的子集树,它是用 FIFO 分支搜索算法解决该问题的扩展结点的过程编号的。

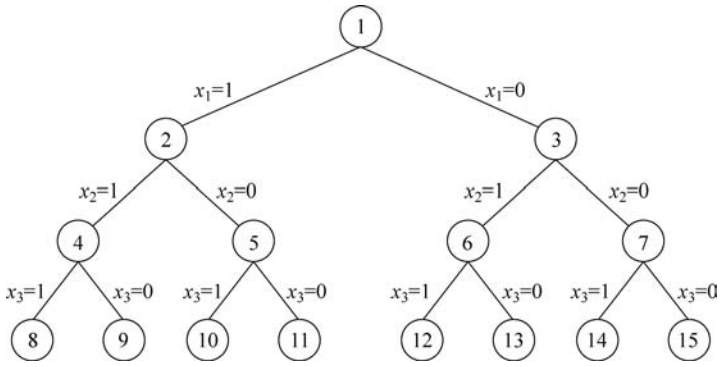

图 5-24 $n=3$ 的子集树

(1) 用 FIFO 分支搜索所有的分支,并记录已搜索分支的最优解,搜索完子集树也就找出了问题的解。图 5-24 中结点 1 为第零层,是初始 E-结点;扩展后结点 2,3 为第一层……最后结点 8,9,10,…,15 为第三层。

(2) 要想求出最优解,必须搜索到叶结点。所以要记录树的层次,当层次为 $n+1$ 时,搜索完全部叶结点,算法结束。不同于回溯算法,分支搜索过程中活结点的"层"是需要标识的,否则在入队后无法识别结点所在的层。下面算法,每处理完一层让"-1"入队,以此来标识"层",并用记数变量 i 来记录当前层。

(3) 每个活结点要记录当前船的装载量。

(4) 为了突出算法思想,对数据结构队及其操作只进行抽象描述。用 Queue 代表队列类型,则 Queue Q;定义了一个队 Q,相关操作有:

add(Q,…)表示入队;

Empty(Q)测试队列是否为空,为空返回真值;

Delete(Q,…);表示出队。

算法 1 如下:

```
float bestw,w[100];
int n;
Queue Q;
main( )
  { float c1,c2,s = 0;
    int i;
    input(c1,c2,n);
    for(i = 1; i < = n; i = i + 1)
       {input(w[i]);
           s = s + w[i]; }
    if (s < = c1 or s < = c2)
       {print("need only one ship");
          return; }
    if (s > c1 + c2)
       {print("Non solution");
```

```
            return; }
    MaxLoading(c1);
    if (s-bestw < = c2);
        {print("The first ship loading",bestw);
         print("The second ship loading",s-bestw); }
    else
        print("Non solution");
}
MaxLoading(float c)                          // 返回最优装载值
{ Add(Q, -1);                                // 初始化活结点队列,标记分层
  int i = 1;                                 // E-结点的层
  ew = 0;                                    // 当前船的装载量
  bestw = 0;                                 // 目前的最优值
  while (not Empty(Q))                       // 搜索子集空间树
      {if (ew + w[i]< = c)                   // 检查 E-结点的左孩子,物品 i 是否可以装载
          AddLiveNode(ew + w[i],i);          // 物品 i 可以装载
      AddLiveNode(ew,i);                     // 右孩子总是可行的,不装载物品 i
      Delete(Q,ew);                          // 取下一个 E-结点
      if (ew = -1)                           // 到达层的尾部
        { if (Empty(Q)) return bestw;
          Add(Q, -1);                        // 添加分层标记
          Delete(Q,ew);                      // 取下一个 E-结点
          i = i + 1; }                       // Ew 的层
      }
}
AddLiveNode(float wt,int i)
{if (i = n)
    { if (wt > bestw)                        // 是叶子
      bestw = wt; }
else
    Add(Q,wt);                               // 不是叶子
}
```

算法分析：子集树搜索的时间与空间复杂度均为 $O(2^n)$。

5.5.2 分支限界搜索算法

这里先通过实际的例子认识“分支限界(branch and bound)搜索算法”,接上小节继续讨论例 16。

算法设计 2：用 FIFO 分支限界搜索算法解决例 16 的问题。

上一小节例 16 的算法 1 存在的问题与改进方法。

(1) 在可解的情况下,没有给出每艘装载物品的方案。

回溯法是用一个数组在搜索中记录解方案的,数组的大小与解向量的大小相同。而分支搜索算法以广度优先的思想搜索,同时存在的活结点很多,用固定大小的一维数组不易保存它们,且活结点之间的扩展关系也不能准确存储。要想记录第一艘船最大装载的方案,可以像 5.2.2 节中的例子用一个结点较大的数组队列记录解方案。

这里采用显式地构造解空间二叉树的方法,问题的解就是二叉树中的某一个分支。这个解是要搜索到二叉树的叶结点才能确定的,且只需要记录最优解的叶结点,就能找到问题

的解。比较方便的存储方式是二叉树要有指向父结点的指针,以便从叶结点回溯寻找解的方案。又为了方便知道当前结点对物品的取舍情况,还要记录当前结点是父结点的哪一个儿子。

数据结构:由上面的分析,树中结点的信息包括物品重量 Weight、父指针 Parent、是否左儿子 Lchild(值为 1 为左儿子,表示取该物品;值为 0 为右儿子,表示不取该物品)。同时这些结点的地址又是搜索队列的元素,队列操作与算法 1 相同。

(2) 算法 1 是在对子集树进行盲目搜索,是 NP 类问题。虽然不能将搜索算法改进为多项式级的复杂度,但若在算法中加入了"限界"技巧,还是能降低算法的复杂度的。

要想进行算法改进,当前最大装载 bestw 就不能只是在搜索到叶结点才确定,而是每次搜索到要装载的情况(搜索左儿子)时,都要重新确定 bestw 的值。一个简单的现象,若当前分支的"装载上界",比当前的最大装载 bestw 小,则该分支就无须继续搜索。而一个分支的"装载上界",也是容易求解的,就是假设装载当前物品以后的所有物品。

举一个简单的例子,当有 3 件物品重量为 $w=\{50,10,10\}$,装载量为 $c_1=60$。问题所构成的子集树如图 5-25 所示,x_i 为 1 表示选取第 i 件物品;x_i 为 0 表示不选取第 i 件物品。

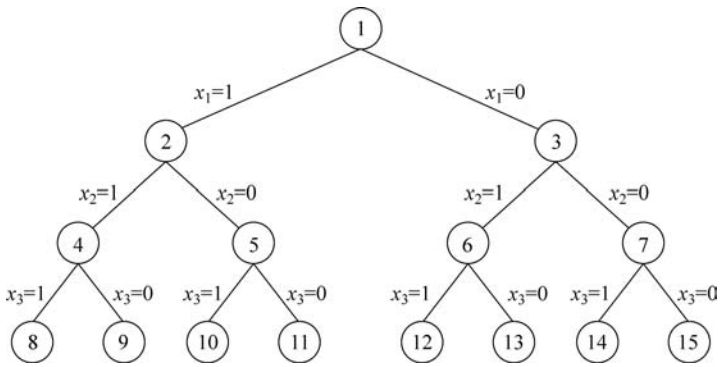

图 5-25 子集树

搜索结点 3 时可以确定它不必被扩充为活结点;因为扩展结点 1 后,就知道最大装载量不会小于 50;而扩展结点 3 时,发现此分支的"装载上界"为 $w_2+w_3=20<50$,无须搜索此分支,结点 3 不必入队。

数据结构:为了方便计算一个分支的"装载上界",用变量 r 记录当前层以下分支的最大重量。

算法 2 如下:

```
float bestw,w[100],bestx[100];
int n;
Queue Q;
struct QNode
  { float weight;
    QNode * parent;
    QNode LChild;
  };
main( )
```

```
{int c1,c2,n,s = 0,i;
  input(c1,c2,n);
  for(i = 1; i < = n; i = i + 1)
    {input(w[i]);
       s = s + w[i]; }
  if (s < = c1 or s < = c2)
     {print("need only one ship");
      return; }
  if (s > c1 + c2)
     {print("Non solution");
      return; }
  MaxLoading(c1);
  if (s-bestw < = c2);
    {print("The first ship loading",bestw,"choose: ");
     for(i = 1; i < = n; i = i + 1)
      if(bestx[i] = 1)
        print(i,",");
    print("换行符 The second ship loading",s-bestw,"choose");
     for(i = 1; i < = n; i = i + 1)
       if(bestx[i] = 0)
        print(i,",");
   }
 else
  print("Non solution");
}
AddLiveNode(folat wt,int i,QNode * E,int ch)
{ Qnode * b;
  if (i = n)                                    // 若是叶子
  { if (wt > bestw)                             // 目前的最优解
    { bestE = E;
      bestx[n] = ch; }                          // bestx[n] 取值为 ch
  return;
    }
  b = new QNode;                                // 不是叶子,添加到队列中
  b -> weight = wt;
  b -> parent = E;
  b -> LChild = ch;
  add (Q,b);
  }
MaxLoading(int c)
{Qnode * E;                                     // 活结点队列
  int i = 1;                                    // E-结点的层
  add (Q,0);                                    // 0 代表分层标记
  E = new QNode;
  E -> weight = 0;                              // E-结点的重量
  E -> parent = null;
  E -> Lchild = 0;
  add (Q,E);
  Ew = 0;                                       // E-结点的重量
```

```
bestw = 0;                                  // 迄今得到的最优值
r = 0;                                      // E-结点中余下的重量
for (int j = 2; j <= n; j = j + 1)
   r = r + w[j];
while (true)                                // 搜索子集空间树
{ wt = Ew + w[i];                           // 检查 E-结点的左孩子
  if (wt <= c)                              // 可行的左孩子
    { if (wt > bestw)
        bestw = wt;
      AddLiveNode(wt,i,E,1);
    }
  if (Ew + r > bestw)                       // 检查右孩子
       AddLiveNode(Ew,i,E,0);
  Delete (Q,E);                             // 下一个 E-结点
   if (E = 0)                               // 层的尾部
    {if (Empty(Q))
       break;
      add (Q 0);                            // 分层标记
      Delete(Q,E);                          // 下一个 E-结点
      i = i + 1;                            // E-结点的层次
      r = r - w[i]; }                       // E-结点中余下的重量
      Ew = E -> weight;                     // 新的 E-结点的重量
      }
 for (j = n - 1; j > 0; j = j - 1)          // 沿着从 bestE 到根的路径构造 bestx
      {bestx[j] = bestE -> LChild;
       bestE = bestE -> parent; }
return bestw;
}
```

算法设计 3：用优先队列式分支限界搜索算法解决例 16 的问题。

5.5.1 节介绍了 3 种不同的分支搜索方式：FIFO、LIFO 和优先队列，前面介绍的算法都是用 FIFO 搜索方式。当然用 LIFO 搜索方式也同样可以设计出对应的算法，请读者尝试。

对于优先队列式扩展方式，不加入限界策略其实是无意义的，因为要说明解的最优性，必需搜索完问题全部解空间，才能下结论。那么先搜索哪一个结点就不重要了，也就是说考虑扩充结点的优先级就没有任何意义了。

优先队列式搜索通过结点的优先级，可以使搜索尽快朝着解空间树上有最优解的分支推进，这样当前最优解一定较接近真正的最优解。其后将当前最优解作为一个“标准”，对上界(或下界)不可能达到(或大于)这个“标准”的分支，则不去进行搜索，这样剪枝的效率更高，能较好地缩小搜索范围，从而提高搜索效率。这种搜索策略称为优先队列式分支限界法，即“LC-搜索”。

优先队列式分支限界搜索算法进行算法设计的要点如下。

(1) 结点扩展方式：无论哪种分支限界搜索算法，都需要有一张活结点表。优先队列的分支限界搜索算法将活结点组织成一个优先队列，并按优先队列中规定的结点优先级选

取优先级最高的下一个结点成为当前扩展结点。

(2) 结点优先级确定：优先队列中结点优先级常规定为一个与该结点相关的数值 w，w 一般表示以该结点为根的子树中的分支(其中最优的分支)接近最优解的程度。

在本例中，以当前结点所在分支的装载上界为优先值。

(3) 优先队列组织：结点优先级确定后，按结点优先级进行排序，就生成了优先队列。

排序算法的时间复杂度较高，考虑到搜索算法每次只扩展一个结点，使用数据结构中介绍的堆排序比较合适，这样每次扩展结点时，交换的次数最少。

在本例中，采用最大堆来实现优先队列。为了突出算法本身的思想，对堆操作也只进行抽象的描述。

用 HeapNode 代表堆类型，则“HeapNode H;”表示定义了一个堆 H，相关操作有：

“Insert(H,…);”表示入堆；

“DeleteMax(H,…);”表示取出堆中的最大值。

数据结构设计：

(1) 要输出解的方案，在搜索过程中仍需要生成解结构树，其结点信息包括指向父结点的指针和标识物品取舍(或是父结点的左、右孩子)。

(2) 堆结点首先应该包括结点优先级信息：结点所在分支的装载上界 uweight；堆中无法体现结点的层次信息(Level)，只能存储在结点中。

AddLiveNode 用于把 bbnode 类型的活结点加到子树中，并把 HeapNode 类型的活结点插入最大堆。

选取 E-结点是在堆上进行的，堆结点增加指针类型成员 ptr，指向解结构树对应结点。

(3) 不同于算法 2，由于扩展结点不是按层进行的，在计算结点的所在分支的装载上界时，要用数组变量 r 记录各层以下的最大重量，这样可以随时方便使用各层结点的装载上界。

算法 3 如下：

```
HeapNode H[1000];
struct bbnode
{bbnode * parent;                          // 父结点指针
  int LChild; };                           // 当且仅当是父结点的左孩子时,取值为 1
struct HeapNode
{bbnode * ptr;                             // 活结点指针
  float uweight;                           // 活结点的重量上限
  int level; };                            // 活结点所在层
AddLiveNode(float wt,int lev,bbnode * E,int ch)
{bbnode * b = new bbnode;
 b -> parent = E;
 b -> LChild = ch;
 HeapNode N;
 N.uweight = wt;
 N.level = lev;
 N.ptr = b;
 Insert(H,N);
}
MaxLoading(float c,int n,int bestx[])
```

```
{froat r[100],Ew,bestw = 0;                    // r[j]为 w [ j+1: n ]的重量之和
 r[n] = 0;
 for (int j = n - 1; j > 0; j = j - 1)
     r[j] = r[j + 1] + w[j + 1];
 int i = 1;                                    // 初始化 E-结点的层
 bbnode * E = 0;                               // 当前 E-结点
 Ew = 0;                                       // E-结点的重量
 // 搜索子集空间树
 while (i <> n + 1)                            // 不在叶子上
   { if (Ew + w[i]< = c)                       // 可行的左孩子
        {AddLiveNode(Ew + w[i] + r[i],i + 1,E,1); }
     if (bestw < Ew + w[i])
        bestw = Ew + w[i];
     if(bestw < Ew + r[i])                     // 可行的右孩子
        AddLiveNode(Ew + r[i],i + 1,E,0);
     DeleteMax(H,E);                           // 取下一个 E-结点
     i = N.level;
     E = N.ptr;
     Ew = N.uweight - r[i - 1];
   }
 for (int j = n; j > 0; j = j - 1)             // 沿着从 E-结点 E 到根的路径构造 bestx[]
     {bestx[j] = E -> LChild;
       E = E -> parent;
     }
 return Ew;
}
```

算法说明：算法的复杂度仍为 $O(2^n)$，但通过限界策略，并没有搜索子集树中的所有结点，且由于每次都是选取的最接近最优解的结点扩展，所以当搜索到叶结点作 E-结点时，算法就可以结束了。算法结束时堆并不一定为空。

算法 2 中在 FIFO 搜索算法中加入了“限界”策略，但由于 FIFO 搜索是盲目地扩展结点，当前最优解与真正的最优解距离较大，作为剪枝“标准”的界 bestw 所起到的作用很有限，不能有效提高搜索速度。

为了进一步理解算法，并与 FIFO 分支限界搜索算法进行比较，看下面一个简单的例子。

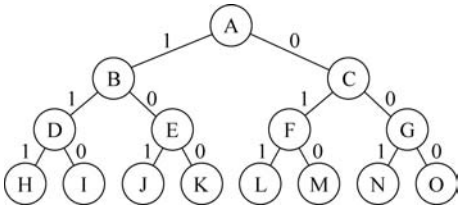

图 5-26 问题所构成的子集树

例如，当有 3 件物品重量为 $w=\{10,30,50\}$，装载量为 $c_1=60$。问题所构成的子集树如图 5-26 所示，1 表示取物品，0 表示不取物品。

FIFO 分支限界搜索过程为：

(1) 初始队列中只有结点 A。

(2) 结点 A 变为 E-结点扩充 B 入队，bestw=10；结点 C 的装载上界为 30+50=80>bestw，也入队。

(3) 结点 B 变为 E-结点扩充 D 入队，bestw=40；结点 E 的装载上界为 60>bestw，也入队。

(4) 结点 C 变为 E-结点扩充 F 入队，bestw 仍为 40；结点 G 的装载上界为 50＞bestw，也入队。

(5) 结点 D 变为 E-结点，叶结点 H 超过容量，叶结点 I 的装载为 40，bestw 仍为 40。

(6) 结点 E 变为 E-结点，叶结点 J 装载量为 60，bestw 为 60；叶结点 K 被剪掉。

(7) 结点 F 变为 E-结点，叶结点 L 超过容量，bestw 为 60；叶结点 M 被剪掉。

(8) 结点 G 变为 E-结点，叶结点 N，O 都被剪掉。

(9) 此时队列空算法结束。

而 LC-搜索的过程如下：

(1) 初始队列中只有结点 A。

(2) 结点 A 变为 E-结点扩充 B 入堆，bestw＝10；结点 C 的装载上界为 30＋50＝80＞bestw，也入堆。且在堆中结点 B 的上界为 90，在优先队列首。

(3) 结点 B 变为 E-结点扩充 D 入堆，bestw＝40；结点 E 的装载上界为 60＞bestw，也入堆；此时堆中结点 D 的上界为 90，在优先队列首。

(4) 结点 D 变为 E-结点，叶结点 H 超过容量，叶结点 I 的装载为 40 入堆，bestw 仍为 40；此时堆中结点 C 的上界为 80，在优先队列首。

(5) 结点 C 变为 E-结点扩充 F 入堆，bestw 仍为 40；结点 G 的装载上界为 50＞bestw，也入堆；此时堆中结点 E 的上界为 60，在优先队列首。

(6) 结点 E 变为 E-结点，叶结点 J 装载量为 60 入堆，bestw 变为 60；叶结点 K 上界为 10＜bestw 被剪掉；此时堆中 J 上界为 60，在优先队列首。

(7) 结点 J 变为 E-结点，扩展的层次为 4，算法结束。虽然此时堆并不空，但可以确定已找到了最优解。

FIFO 分支限界搜索算法搜索解空间的过程是按图 5-26 子集树中字母序进行的，而优先队列分支限界搜索算法搜索解空间的过程是：A-B-D-C-E-J。

看了上面的例子大家会发现，优先队列法扩展结点的过程，一开始实际是在进行类似“深度优先”的搜索。

5.5.3 算法框架

5.5.2 节的例子是求最大值的最优化问题，下面以求找最小成本的最优化问题，给出 FIFO 分支限界搜索算法框架。

假定问题解空间树为 T，T 至少包含一个解结点(即答案结点)。u 为当前的最优解，初值为一个较大的数；E 表示当前扩展的活结点，x 为 E 的儿子，$s(x)$为结点 x 的下界函数，当其值比 u 大时，不可能为最优解，不继续搜索此分支，该结点不入队；当其值比 u 小时，可能达到最优解，继续搜索此分支，该结点入队；cost(X)为当前叶结点所在分支的解。

算法框架如下：

```
search(T)                                   // 为找出最小成本答案结点检索 T
  { leaf = 0;
    初始化队;
    ADDQ(T);                                // 根结点入队
    parent(E) = 0;                          // 记录扩展路径,当前结点的父结点 parent
```

```
    while (队不空)
      {DELETEQ(E)                          // 队首结点出队为新的 E 结点;
      for (E 的每个儿子 X)
      if (s(X)< u)                         // 当是可能的最优解时入队
         { ADD Q(X);
           parent(X) = E;
         if (X 是解结点)                    // x 为叶结点
             {U = min(cost(X),u);
             leaf = x; }                   // 方案的叶结点存储在 leaf 中
          }
       }
print("least cost = ",u);
  while (leaf <> 0)                        // 输出最优解方案
        {print(leaf);
        leaf = parent(leaf); }
}
```

找最小成本的 LC 分支限界搜索算法框架与 FIFO 分支限界搜索算法框架结构大致相同,只是扩展结点的顺序不同,因而存储活结点的数据结构不同。FIFO 分支限界搜索算法用队存储活结点,LC 分支限界搜索算法用堆存储活结点,以保证比较优良的结点先被扩展。且对于 LC 分支限界搜索算法,当扩展到叶结点就已经找到最优解,可以停止搜索。

5.6 图的搜索算法小结

1. 深度优先搜索与广度优先搜索算法

通常深度优先搜索法不全部保留结点,扩展完的结点从数据存储结构栈中弹出删去,这样,一般在数据栈中存储的结点数就是解空间树的深度,因此它占用空间较少。所以,当搜索树的结点较多,用其他方法易产生内存溢出时,深度优先搜索不失为一种有效的求解方法。

广度优先搜索算法,一般须存储产生的所有结点,占用的存储空间要比深度优先搜索大得多,因此,程序设计中,必须考虑溢出和节省内存空间的问题。但广度优先搜索法一般无回溯操作,即入栈和出栈的操作,所以运行速度比深度优先搜索要快些。

2. 回溯法与分支限界法

回溯法以深度优先的方式搜索解空间树 T,而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树 T。由于它们在问题的解空间树 T 上搜索的方法不同,适合解决的问题也就不同。一般情况下,回溯法的求解目标是找出 T 中满足约束条件的所有解的方案,而分支限界法的求解目标则是找出满足约束条件的一个解,或是在满足约束条件的解中找出使用某一目标函数值达到极大或极小的解,即在某种意义下的最优解。相对而言,分支限界法的解空间比回溯法大得多,因此当内存容量有限时,回溯法成功的可能性更大。

表 5-2 列出了回溯法和分支限界法的一些区别。

表 5-2　回溯法和分支限界法的比较

方　法	对解空间树的搜索方式	存储结点的常用数据结构	结点存储特性	主要应用
回溯法	深度优先搜索	堆栈	活结点的所有可行子结点被遍历后才被从栈中弹出	找出满足约束条件的所有解
分支限界法	广度优先或最小消耗优先搜索	队列、优先队列、堆	每个结点只有一次成为活结点的机会	找出满足约束条件的一个解或特定意义下的最优解

在处理最优问题时，采用穷举法、回溯法或分支限界法都可以通过利用当前最优解和上界函数加速。仅就限界剪枝的效率而言，优先队列的分支限界法显然要更充分一些。在穷举法中通过上界函数与当前情况下函数值的比较可以直接略过不合要求的情况而省去了更进一步的枚举和判断；回溯法则因为层次的划分，可以在上界函数值小于当前最优解时，剪去以该结点为根的子树，也就是节省了搜索范围；分支限界法在这方面除了可以做到回溯法能做到的之外，若采用优先队列的分支限界法，用上界函数作为活结点的优先级，一旦有叶结点成为当前扩展结点，就意味着该叶结点所对应的解即为最优解，可以立即终止其余的过程。在前面的例题中曾说明，优先队列的分支限界法更像是有选择、有目的地进行深度优先搜索，时间效率、空间效率都是比较高的。

3. 动态规划与搜索算法

撇开时空效率的因素不谈，在解决最优化问题的算法中，搜索可以说是“万能”的。所以动态规划可以解决的问题，搜索也一定可以解决。动态规划要求阶段决策具有无后向性，而搜索算法没有此限制。

动态规划是自底向上的递推求解，而无论深度优先搜索或广度优先搜索都是自顶向下求解。利用动态规划法进行算法设计时，设计者在进行算法设计前已经用大脑自己构造好了问题的解空间，因此可以自底向上递推求解；而搜索算法是在搜索过程中根据一定规则自动构造，并搜索解空间树的。由于在很多情况下，问题的解空间太复杂用大脑构造有一定困难，仍然需要采用搜索算法。

另外动态规划在递推求解过程中，需要用数组存储有关信息，而数组的下标只能是整数，所以要求问题中相关的数据必须为整数（如 4.5.3 节例 25“资源分配问题”中的资金就必须为整数），对于这类信息非整数或不便于转换为整数的问题，同样需要采用搜索算法。

一般来说，动态规划算法在时间效率上的优势是搜索无法比拟的，但动态规划总要遍历所有的状态，而搜索可以排除一些无效状态。更重要的是搜索还可以剪枝，可以剪去大量不必要的状态，因此在空间开销上往往比动态规划要低很多。如何协调好动态规划的高效率与高消费之间的矛盾呢？有一种折中的办法就是记忆化限界搜索算法。

记忆化限界搜索以搜索算法为核心，只不过使用“记录求过的状态”的办法，来避免重复搜索，这样，记忆化限界搜索的每一步，也可以对应到动态规划算法中去。记忆化限界搜索有优化方便、调试容易、思维直观的优点，但是效率上比循环的动态规划差一个常数，但是时间和空间复杂度是同一数量级的（尽管空间上也差一个常数，那就是堆栈空间）。当 n 比较小的时候，可以忽略这个常数，从而记忆化限界搜索可以和动态规划达到完全相同的效果。

记忆化限界搜索算法在求解时,还是按着自顶向下的顺序,但是每求解一个状态,就将它的解保存下来,以后再次遇到这个状态的时候,就不必重新求解了。这种方法综合了搜索和动态规划两方面的优点,因而还是很有实用价值的。

习题

(1) 括号检验:输入一个代数表达式,表达式只能含有+,-,*,/,(,),1,2,3,4,5,6,7,8,9,0 字符且每个数字均小于 10,设表达式除括号匹配有误外,再无其他错误。编写算法对输入的表达式进行检验,判断括号匹配是否正确。

正确的:

```
1+2+4
(1+2)+4
(1+2)
```

错误的:

```
(1+)2
(1+2(4+3))
(1+2+3*(4+5( )))
1+2+3*(4+5))
```

(2) 有一个由数字 1,2,3,…,9 组成的数字串(长度不超过 200),问如何将 M($M \leqslant 20$)个加号("+")插入到这个数字串中,使所形成的算术表达式的值最小。请编写算法解决这个问题。

注意:加号不能加在数字串的最前面或最末尾,也不应有两个或两个以上的加号相邻。M 要小于数字串的长度。

例如:数字串 79846,若需要加入两个加号,则最佳方案为 79+8+46,算术表达式的值为 133。

(3) 有分数 1/2,1/3,1/4,1/5,1/6,1/8,1/10,1/12,1/15,求将其中若干个分数相加和恰好为 1 的组成方案,并打印成等式。例如:

① 1/2+1/3+1/6=1

② …

(4) 是否存在一个由 1~9 组成的 9 位数,每个数字只能出现一次,且这个 9 位数由高位到低位前 i 位能被 i 整除。

(5) 一个整数 n($n \leqslant 100$)可以有多种分划,分划整数之和为 n。例如:

```
输入: n=6
6
5  1
4  2
4  1  1
3  3
3  2  1
3  1  1  1
2  2  2
2  2  1  1
2  1  1  1  1
1  1  1  1  1  1
total=11 {表示分划数有 11 种}
```

求 n 的分划数。

(6) 旅行售货员问题：某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程(或总旅费)最小。

(7) 将 $M\times N$ 个 0 和 1 填入一个 $M\times N$ 的矩阵中，形成一个数表 A，数表 A 中第 i 行数的和记为 $r_i(i=1,2,\cdots,m)$，它们叫作 A 的行和向量，数表 A 第 j 列的数的和记为 $q_j(j=1,2,\cdots,m)$，它们叫作 A 的列和向量。已知数表 A 的行数和列数，以及行和向量与列和向量，编写算法求出满足条件的所有数表 A。

(8) 翻币问题：有 N 个硬币($N\geqslant 10$)，正面向上排成一排，每次必须翻 5 个硬币，直到全部反面向上。

(9) 等分液体，在一个瓶子中装有 8(N 偶数)L 汽油，要平均分成两份，但只有一个装 3($N/2-1$)L 的量杯和装 5($N/2+1$)L 的量杯(都没有刻度)。打印出所有把汽油分成两等分的操作过程。若无解打印“NO”，否则打印操作过程。

(10) 有一个自然数的集合，其中最小的数是 1，最大的数是 100。这个集合中的数除 1 外，每个数都可由集合中的某两个数(这两个数可以相同)求和得到。编写一个程序，求符合上述条件的自然数的个数为 10 的所有集合。

(11) 设有 A,B,C,D,E 5 人从事 J1,J2,J3,J4,J5 5 项工作，每人只能从事一项，他们的效益如图 5-27 所示，求最佳安排使效益最高。

	J1	J2	J3	J4	J5
A	10	11	10	4	7
B	13	10	10	8	5
C	5	9	7	7	4
D	15	12	10	11	5
E	10	11	8	8	4

图 5-27　效益图

(12) 一个正整数有可能可以被表示为 $n(n\geqslant 2)$个连续正整数之和，如 $n=15$ 时：

$$15=1+2+3+4+5$$
$$15=4+5+6$$
$$15=7+8$$

请编写算法，根据输入的任何一个正整数，找出符合这种要求的所有连续正整数序列。