第 2 章　图的遍历与活动网络问题

所谓图的**遍历**(Graph Traversal)，也称为**搜索**(Search)，就是从图中某个顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次。遍历可以采取两种方法进行：深度优先搜索(Depth First Search，DFS)和广度优先搜索(Breadth First Search，BFS)。本章着重讨论这两种搜索算法。

遍历是很多图论算法的基础，可以用来解决图论中的许多问题，如活动网络问题。本章最后两节讨论了活动网络问题，包括 AOV 网络和拓扑排序、AOE 网络和关键路径的求解。

2.1　DFS 遍历

2.1.1　DFS 算法思想

深度优先搜索(DFS)是一个递归过程，有回退过程。DFS 算法的思想是：对一个无向连通图，在访问图中某一起始顶点 v 后，由 v 出发，访问它的某一邻接顶点 w_1；再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2；然后再从 w_2 出发，进行类似的访问；……；如此进行下去，直至到达所有邻接顶点都被访问过的顶点 u 为止；接着，回退一步，回退到前一次刚访问过的顶点，看是否还有其他没有被访问过的邻接顶点，如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再回退一步进行类似的访问。重复上述过程，直到该连通图中所有顶点都被访问过为止。

接下来以图 2.1(a)所示的无向连通图为例解释 DFS 搜索过程。假设在多个未访问过的邻接顶点中进行选择时，按顶点序号从小到大的顺序进行选择，比如顶点 *A* 有 3 个邻接顶点，即 *B*、*D* 和 *E*，首先选择顶点 *B* 进行深度优先搜索。

对图 2.1(a)所示的无向连通图，采用 DFS 思想搜索的过程如下(在图 1.21(a)中，实线表示前进方向，虚线表示回退方向，箭头旁的数字跟下面的序号对应)。

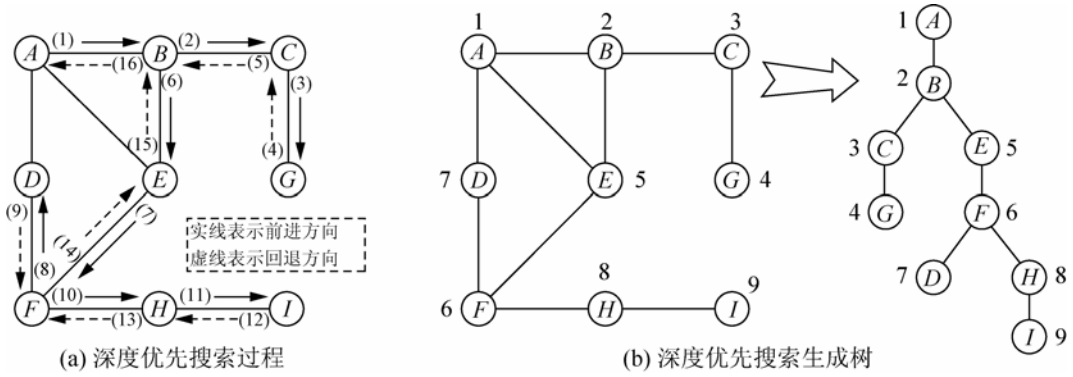


图 2.1　深度优先搜索

(1) 从顶点 *A* 出发，访问顶点序号最小的邻接顶点，即顶点 *B*。
(2) 然后访问顶点 *B* 的未访问过的、顶点序号最小的邻接顶点，即顶点 *C*。
(3) 接着访问顶点 *C* 的未访问过的、顶点序号最小的邻接顶点，即顶点 *G*。
(4) 此时顶点 *G* 已经没有未访问过的邻接顶点了，所以回退到顶点 *C*。
(5) 回退到顶点 *C* 后，顶点 *C* 也没有未访问过的邻接顶点了，所以继续回退到顶点 *B*。
(6) 顶点 *B* 还有一个未访问过的邻接顶点，即顶点 *E*，所以访问顶点 *E*。
(7) 然后访问顶点 *E* 的未访问过的、顶点序号最小的邻接顶点，即顶点 *F*。
(8) 顶点 *F* 有两个未访问过的邻接顶点，选择顶点序号最小的，即顶点 *D*，所以访问 *D*。
(9) 此时顶点 *D* 已经没有未访问过的邻接顶点了，所以回退到顶点 *F*。
(10) 顶点 *F* 还有一个未访问过的邻接顶点，即顶点 *H*，所以访问顶点 *H*。
(11) 然后访问顶点 *H* 的未访问过的、顶点序号最小的邻接顶点，即顶点 *I*。
(12) 此时顶点 *I* 已经没有未访问过的邻接顶点了，所以回退到顶点 *H*。
(13) 回退到顶点 *H* 后，顶点 *H* 也没有未访问过的邻接顶点了，所以继续回退到顶点 *F*。
(14) 回退到顶点 *F* 后，顶点 *F* 也没有未访问过的邻接顶点了，所以继续回退到顶点 *E*。
(15) 回退到顶点 *E* 后，顶点 *E* 也没有未访问过的邻接顶点了，所以继续回退到顶点 *B*。
(16) 回退到顶点 *B* 后，顶点 *B* 也没有未访问过的邻接顶点了，所以继续回退到顶点 *A*。

回退到顶点 *A* 后，顶点 *A* 也没有未访问过的邻接顶点了，而且顶点 *A* 是搜索的起始顶点。至此，整个搜索过程全部结束。由此可见，DFS 搜索最终要回退到起始顶点，并且如果起始顶点没有未访问过的邻接顶点了，则算法结束。

在图 2.1(b)中，每个顶点外侧的数字标明了进行深度优先搜索时各顶点访问的次序，称为顶点的**深度优先数**。图 2.1(b)还给出了访问 *n* 个顶点时经过的 *n*−1 条边，这 *n*−1 条边将 *n* 个顶点连接成一棵树，称此图为原图的**深度优先生成树**，该树的根结点就是深度优先搜索的起始顶点。在图 2.1(b)中，为了更加直观地描述该生成树的树形结构，将此生成树改画成图 2.1(b)中右图所示的树形形状。

2.1.2 DFS 算法的实现及复杂度分析

1. DFS 算法的实现

假设有函数 DFS(*v*)，可实现从顶点 *v* 出发访问它的所有未访问过的邻接顶点。在 DFS 算法中，必有一数组，设为 visited[*n*]，用来存储各顶点的访问状态。如果 visited[*i*] = 1，则表示顶点 *i* 已经访问过；如果 visited[*i*] = 0，则表示顶点 *i* 还未访问过。初始时，各顶点的访问状态均为 0。

用 DFS 函数实现可用如图 2.1(a)所示的搜索过程图 2.2 表示，在主函数中只要调用 DFS(*A*)就可以搜索整个图。图 2.2 中的序号(1)～(16)跟图 2.1(a)中的序号是一一对应的。

如果用邻接表存储图，则 DFS 算法实现的伪代码如下：

```
DFS( 顶点 i )    //从顶点i 进行深度优先搜索
{
    visited[ i ]=1;  //将顶点i 的访问标志置为1
    p=顶点i 的边链表表头指针;
    while( p 不为空 )
```

```
    {
        //设指针 p 所指向的边结点所表示的边中，另一个顶点为顶点 j
        if( 顶点 j 未访问过 )
        {
            //递归搜索前的准备工作需要在这里写代码，如例 2.1
            DFS( 顶点 j );
            //以下是 DFS 的回退位置，在很多应用中需要在这里写代码
            //比如例 2.1 以及求关节点的算法(8.2 节)
        }
        p=p->nextarc;   //p 移向下一个边结点
    }
}
```

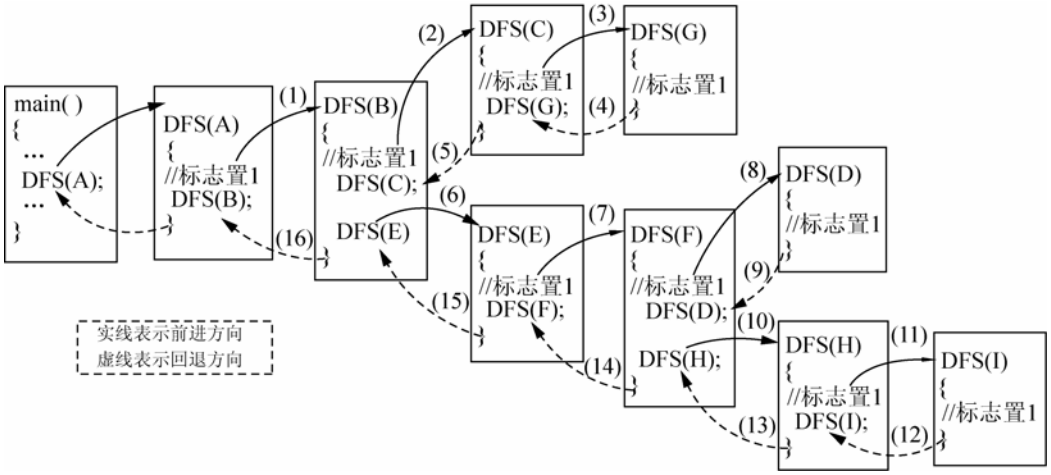


图 2.2　深度优先搜索的实现

如果用邻接矩阵存储图(设顶点个数为 n)，则 DFS 算法实现的伪代码如下：

```
DFS( 顶点 i )    //从顶点 i 进行深度优先搜索
{
    visited[ i ]=1; //将顶点 i 的访问标志置为 1
    for( j=0; j<n; j++ )    //对其他所有顶点 j
    {
        //j 是 i 的邻接顶点，且顶点 j 没有访问过
        if( Edge[i][j]==1 && !visited[j] )
        {
            //递归搜索前的准备工作需要在这里写代码，如例 2.1
            DFS( j )          //从顶点 j 出发进行 DFS 搜索
            //以下是 DFS 的回退位置，在很多应用中需要在这里写代码
            //比如例 2.1 以及求关节点的算法(8.2 节)
        }
    }
}
```

在上述伪代码中，在递归调用 DFS 函数前后的两个位置特别重要。

(1) 如果需要在递归搜索前做一些准备工作(如例 2.1 中在递归搜索前将当前方格设置为墙壁)，则需要在 DFS 递归调用前的位置写代码。

(2) 如果需要在搜索的回退后做一些还原工作(如例 2.1 中在搜索回退后将当前方格还原成空格)，或者根据搜索结果做一些统计或计算工作(如 8.2 节求关节点的算法)，则需要在 DFS 递归调用后的位置写代码。

2. 算法复杂度分析

现以图 2.1(a)所示的无向图为例分析 DFS 算法的复杂度。设图中有 n 个顶点，有 m 条边。

如果用邻接表存储图，如图 2.3 所示，从顶点 i 进行深度优先搜索，首先要取得顶点 i 的边链表表头指针，设为 p，然后要通过指针 p 访问它的第 1 个邻接顶点，如果该邻接顶点未访问过，则从这个顶点出发进行递归搜索；如果这个邻接顶点已经访问过，则指针 p 要移向下一个边结点。在这个过程中，对每个顶点递归访问 1 次，即每个顶点的边链表表头指针取出一次，而每个边结点都只访问了一次。由于总共有 $2m$ 个边结点，所以扫描边的时间为 $O(2m)$。因此，采用邻接表存储图时，进行深度优先搜索的时间复杂性为 $O(n+2m)$。

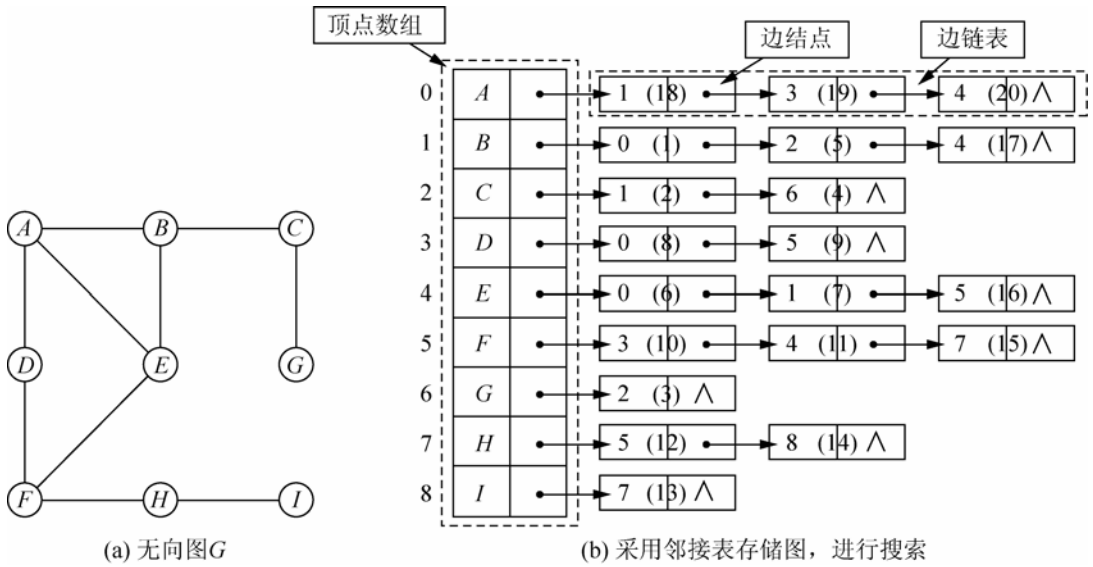


图 2.3　采用邻接表存储图，进行深度优先搜索

在图 2.3(b)中，每个边结点上用圆括号括起来的序号表示每条边的扫描顺序。这个扫描顺序不太好理解，现详细解释如下。

(1) 从顶点 A 出发进行 DFS 搜索，首先取出顶点 A 的边链表表头指针，设为 pA；pA 所指向的边结点表示边(A, B)，而顶点 B 没有访问过，所以要递归调用 DFS(B)，注意这个过程并没有去扫描边结点(A, B)。

(2) 从顶点 B 进行 DFS 搜索，首先取出顶点 B 的边链表表头指针，设为 pB；pB 所指向的边结点表示边(B, A)，而顶点 A 已经访问过了，所以将指针 pB 移向下一个边结点，即要扫描边结点(B, A)，并使得 pB 指向边(B, C)所表示的边结点，所以整个 DFS 过程最先扫描的边结点是(B, A)；现 pB 指向边结点(B, C)，而顶点 C 还未访问，所以要递归调用 DFS(C)，同样这个过程并没有去扫描边结点(B, C)。

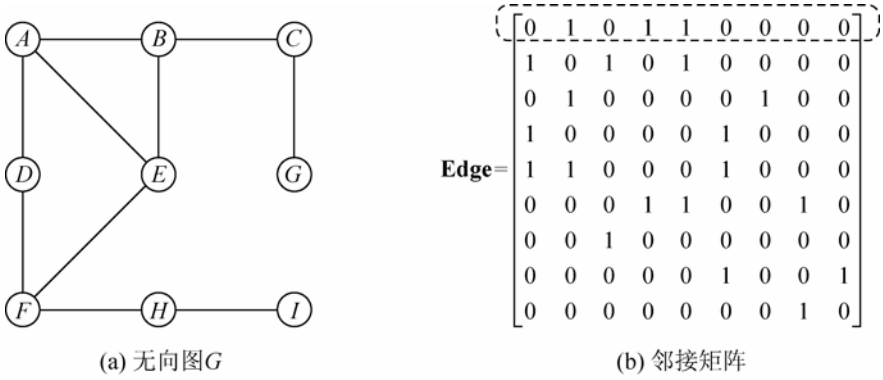
(3) 从顶点 *C* 进行 DFS 搜索，首先取出顶点 *C* 的边链表表头指针，设为 *pC*；*pC* 所指向的边结点表示边(*C*, *B*)，而顶点 *B* 已经访问过了，所以将指针 *pC* 移向下一个边结点，即要扫描边结点(*C*, *B*)(因此，边结点(*C*, *B*)的扫描顺序为 2)，并使得 *pB* 指向边(*C*, *G*)所表示的边结点；现 *pB* 指向边结点(*C*, *G*)，而顶点 *G* 还未访问，所以要递归调用 DFS(G)。

(4) 从顶点 *G* 进行 *DFS* 搜索，首先取出顶点 *G* 的边链表表头指针，设为 *pG*；p*G* 所指向的边结点表示边(*G*, *C*)，而顶点 *C* 已经访问过了，所以要扫描边结点(*G*, *C*)(因此，边结点(*G*, *C*)的扫描顺序为 3)，并将指针 *pG* 移向下一个边结点，而下一个边结点为空，所以顶点 *G* 访问完毕，回退到 DFS(C)。

(5) 回退到 DFS(C)后，继续扫描边结点(*C*, *G*)(因此，边结点(*C*, *G*)的扫描顺序为 4)后，顶点 *C* 也访问完毕，回退到 DFS(B)。

……

如果采用邻接矩阵存储图，由于在邻接矩阵中只是间接地存储了边的信息。在对某个顶点进行 DFS 搜索时，要检查其他每个顶点，包括它的邻接顶点和非邻接顶点，所需时间为 $O(n)$。例如，在图 2.4(b)中，执行 DFS(A)时，要在邻接矩阵中的第 0 行检查顶点 *A*～*I* 与顶点 *A* 是否相邻且是否已经访问过。另外，整个 DFS 过程，对每个顶点都要递归进行 DFS 搜索，因此遍历图中所有的顶点所需的时间为 $O(n^2)$。



(a) 无向图*G*　　(b) 邻接矩阵

图 2.4　采用邻接矩阵存储图，进行深度优先搜索

2.1.3　例题解析

以下通过两道例题的分析，再详细介绍深度优先搜索的算法思想及其实现方法。

例 2.1　骨头的诱惑(Tempter of the Bone)

题目来源：

Zhejiang Provincial Programming Contest 2004, ZOJ2110

题目描述：

一只小狗在一个古老的迷宫里找到一根骨头，当它叼起骨头时，迷宫开始颤抖，它感觉到地面开始下沉。它才明白骨头是一个陷阱，它拼命地试着逃出迷宫。

迷宫是一个 *N*×*M* 大小的长方形，迷宫有一个门。刚开始门是关着的，并且这个门会在第 *T* 秒钟开启，门只会开启很短的时间(少于 1 秒)，因此小狗必须恰好在第 *T* 秒达到门的位置。每秒钟，它可以向上、下、左或右移动一步到相邻的方格中。但一旦它移动到相邻的方格，这个方格开始下沉，而且会在下 1 秒消失。所以，它不能在一个方格中停留超过

一秒，也不能回到经过的方格。

小狗能成功逃离吗？请帮助它。

输入描述：

输入文件包括多个测试数据。每个测试数据的第 1 行为 3 个整数：*N*、*M*、*T*，(1<*N*、*M*<7；0<*T*<50)，分别代表迷宫的长和宽，以及迷宫的门会在第 *T* 秒时刻开启。

接下来 *N* 行信息给出了迷宫的格局，每行有 *M* 个字符，这些字符可能为如下值之一。

X: 墙壁，小狗不能进入　　*S*: 小狗所处的位置

D: 迷宫的门　　.: 空的方格

输入数据以 3 个 0 表示输入数据结束。

输出描述：

对每个测试数据，如果小狗能成功逃离，则输出"YES"，否则输出"NO"。

样例输入：	**样例输出：**
3 4 5 S... .X.X ...D 4 4 8 .X.X ..S. .... DX.X 4 4 5 S.X. ..X. ..XD 0 0 0	YES YES NO

分析：

本题要采用 DFS 搜索思想求解，本节也借助这道题目详细分析 DFS 搜索策略的实现方法及搜索时要注意的问题。

1) 搜索策略

以样例输入中的第 1 个测试数据进行分析，如图 2.5 所示。图 2.5(a)表示测试数据及所描绘的迷宫；在图 2.5(b)中，圆圈中的数字表示某个位置的行号和列号，行号和列号均从 0 开始计起，实线箭头表示搜索前进方向，虚线箭头表示回退方向。

搜索时从小狗所在初始位置 *S* 出发进行搜索。每搜索到一个方格位置，对该位置的 4 个可能方向(要排除边界和墙壁)进行下一步搜索。往前走一步，要将到达的方格设置成墙壁，表示当前搜索过程不能回到经过的方格。一旦前进不了，要回退，要恢复现场(将前面设置的墙壁还原成空的方格)，回到上一步时的情形。只要有一个搜索分支到达门的位置并且符合要求，则搜索过程结束。如果所有可能的分支都搜索完毕，还没找到满足题目要求的解，则该迷宫无解。

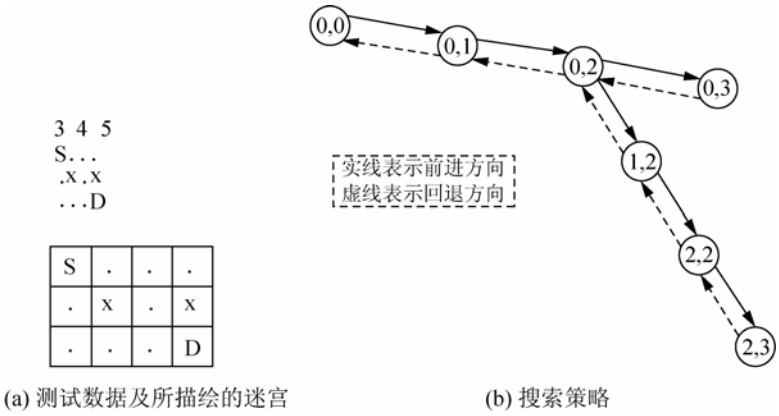


图 2.5　骨头的诱惑(搜索策略)

2) 搜索实现

假设实现搜索的函数为 dfs，它带有 3 个参数。dfs(s_i, s_j, *cnt*)：已经到达(s_i, s_j)位置，且已经花费 *cnt* 秒，如果到达门的位置且时间符合要求，则搜索终止；否则继续从其相邻位置继续进行搜索。继续搜索则要递归调用 dfs 函数，因此 dfs 是一个**递归函数**。

成功逃离条件：$s_i = d_i$，$s_j = d_j$，$cnt = t$。其中(d_i, d_j)是门的位置，在第 *t* 秒钟开启。

假设按照**上、右、下、左顺时针的顺序**进行搜索。则对样例输入中的第 1 个测试数据，其搜索过程及 dfs 函数的执行过程如图 2.6 所示。

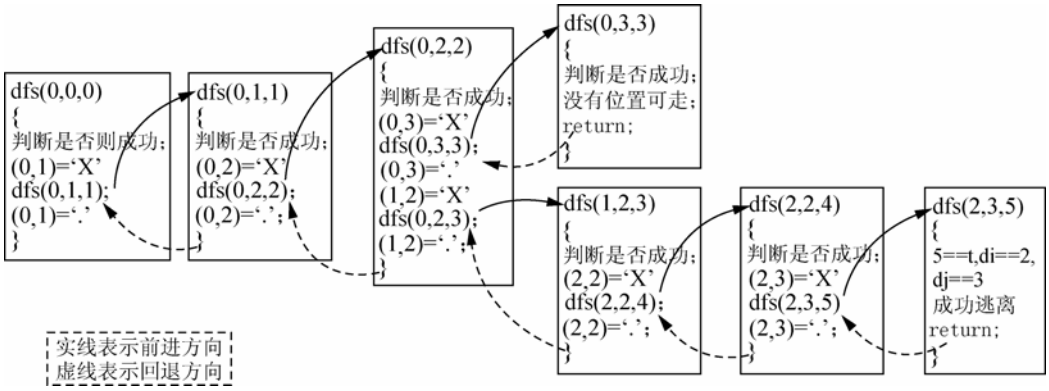


图 2.6　骨头的诱惑(搜索策略的函数实现)

在该测试数据中，小狗的起始位置在(0,0)处，门的位置在(2,3)处，门会在第 5 秒钟开启。在主函数中，调用 dfs(0,0,0)搜索迷宫。当递归执行到某一个 dfs 函数 dfs(si, sj, cnt)，满足 *sj*==2==*di*，*sj*==3==*dj*，且 *cnt*==5==*t*，则表示能成功逃离。

图 2.6 演示了 dfs(0,0,0)的递归执行过程。在图中，各符号含义如下。

(1) (0,1) = 'X'：表示往前走一步，要将当前方格设置成墙壁；

(2) (0,1) = '.'：表示回退过程，要恢复现场，即将(0,1)这个位置由原来设置的墙壁还原成空格。

在执行 dfs(0,0,0)时，按照搜索顺序，上方是边界，不能走，所以向右走一步，即要递归调用 dfs(0,1,1)。在调用 dfs(0,1,1)之前，将(0,1)位置置为墙壁。走到(0,1)位置后，下一步

要走的位置是(0,2)，要递归调用 dfs(0,2,2)。在调用 dfs(0,2,2)之前，将(0,2)位置置为墙壁。走到(0,2)位置后，下一步要走的位置是(0,3)，要递归调用 dfs(0,3,3)。在调用 dfs(0,3,3)之前，将(0,3)位置置为墙壁。在走到(0,3)位置后，其 4 个相邻位置中上边、右边是边界，下边是墙壁，左边本来是空的方格，但因为在前面的搜索前进方向上已经将它设置成墙壁了，所以没有位置可走，只能回退到上一层，即 dfs(0,3,3)函数执行完毕，要回退到主调函数处，也就是 dfs(0,2,2)函数中。

回到 dfs(0,2,2)函数处，即处在位置(0,2)，且已经走了 2 秒。(0,2)位置的 4 个相邻位置中，还有(1,2)这个位置可以走，则从(1,2)位置继续搜索。

按照上述搜索策略，一直搜索到(2,3)位置处，这个位置是门的位置，且刚好走了 5 秒。所以得出结论：能够成功逃脱。

这里要注意，搜索方向的选择是通过下面的二维数组及循环控制来实现的。该二维数组表示上、右、下、左 4 个方向相对当前位置 *x*、*y* 坐标的增量。

```
int dir[4][2] = { {-1,0}, {0,1}, {1,0}, {0,-1} };
```

3) 为什么在回退过程中要恢复现场

以样例输入中的第 2 个测试数据来解释这个问题。在这个测试数据中，如果加上回退过程的恢复现场操作，则不管按什么顺序(上、右、下、左顺序或者左、右、下、上顺序)进行搜索，都能成功脱离；但是去掉回退过程的恢复现场操作后，按某种搜索顺序能成功脱离，但按另外一种搜索顺序则不能成功脱离，这是错误的。图 2.7～图 2.10 以测试数据 2 为例分析了加上和去掉回退过程分别按两种搜索顺序进行搜索的过程和结果。

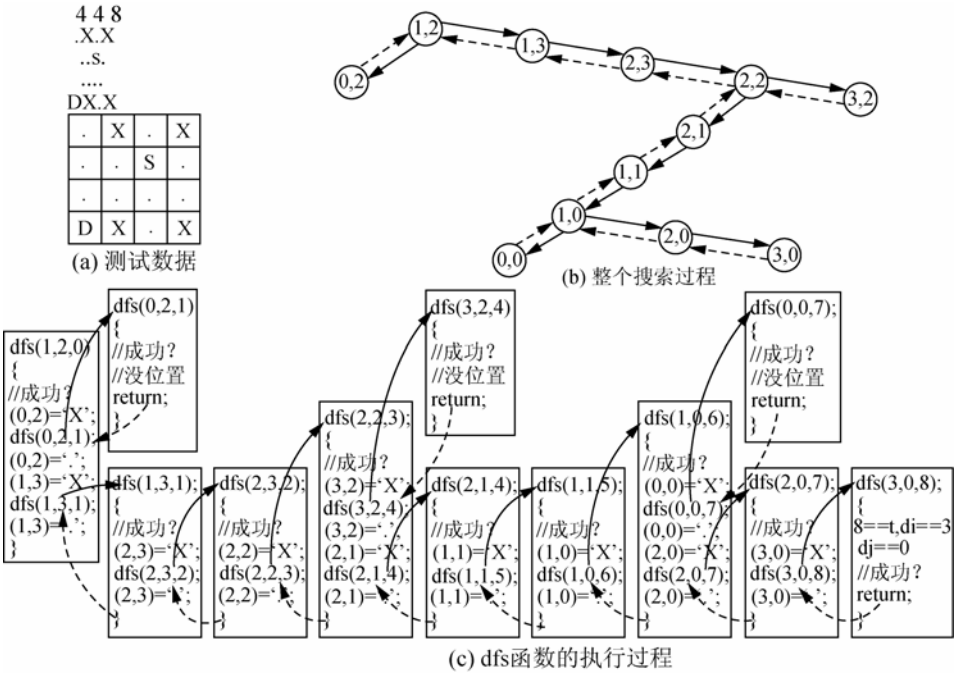


图 2.7　骨头的诱惑(测试数据 2 分析 1)

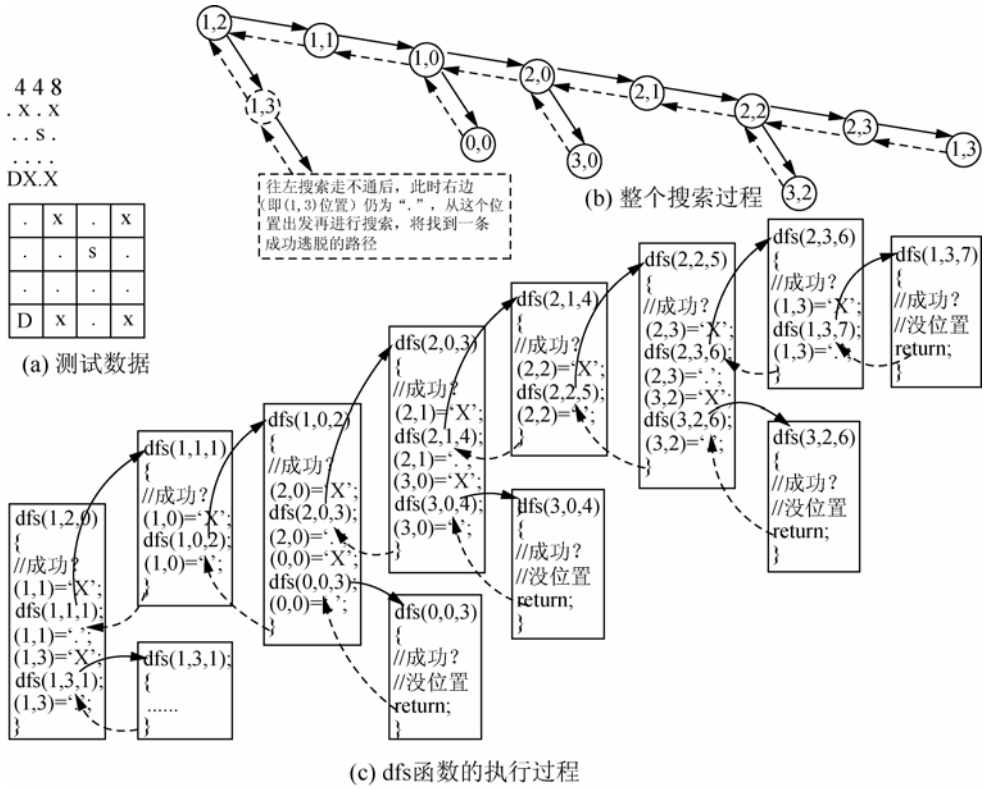


图 2.8　骨头的诱惑(测试数据 2 分析 2)

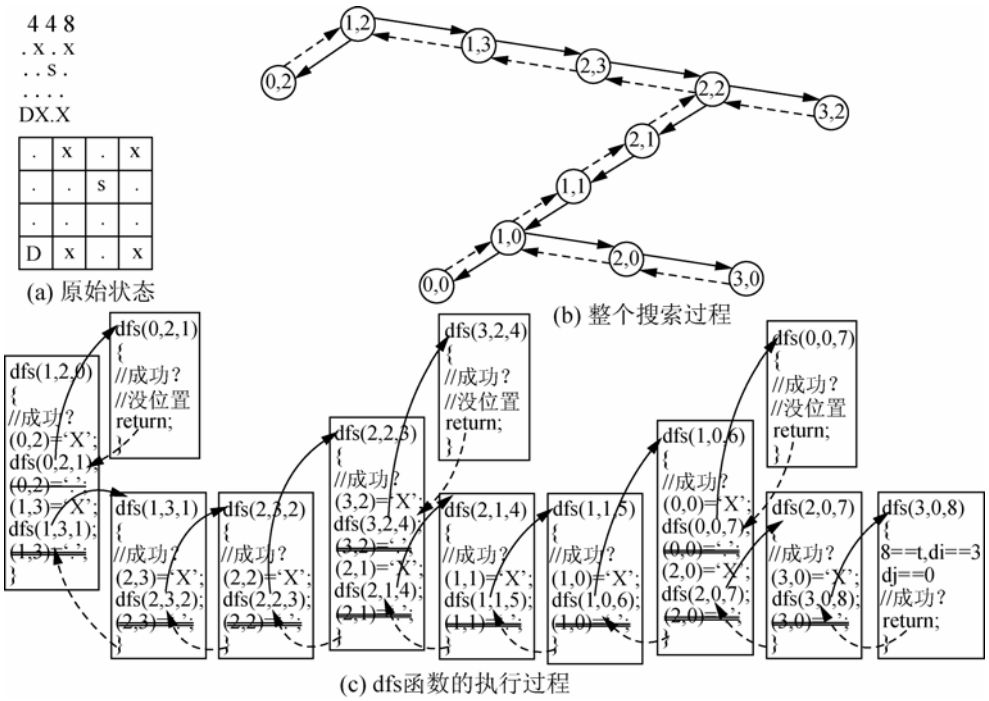


图 2.9　骨头的诱惑(测试数据 2 分析 3)

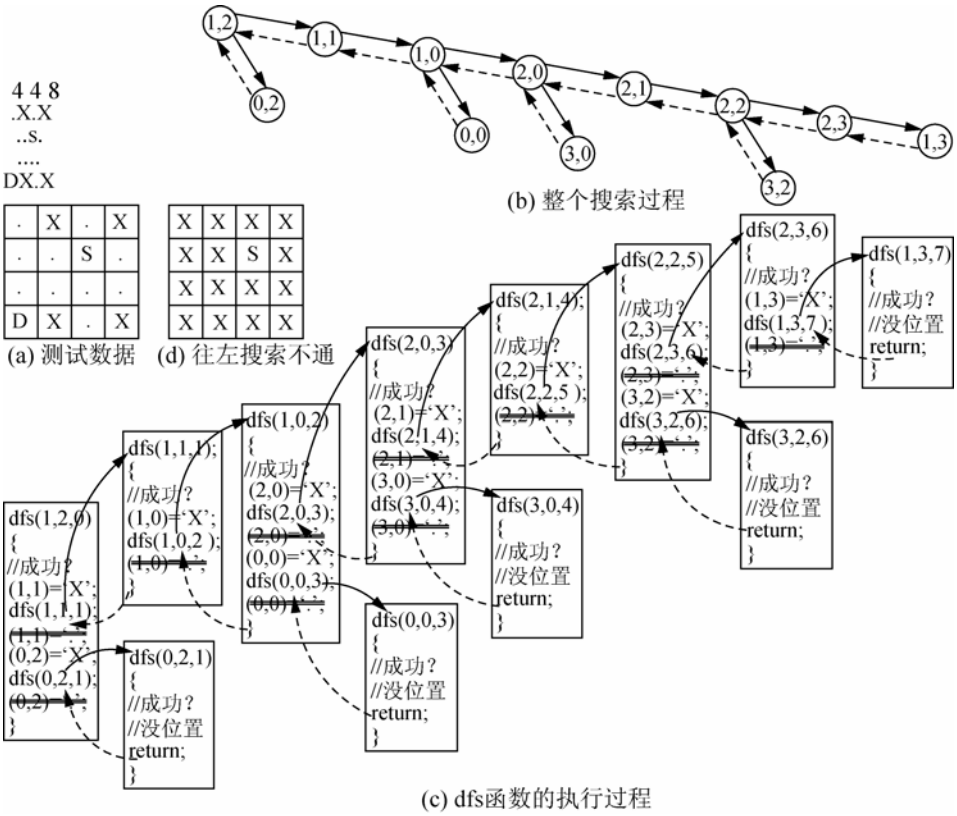


图 2.10　骨头的诱惑(测试数据 2 分析 4)

测试数据 2 分析 1：回退过程有恢复现场，dir 数组如下，即搜索方向为上、右、下、左顺序，整个搜索过程如图 2.7(b)所示，dfs 函数的执行过程如图 2.7(c)所示。dfs 函数执行的结果是能成功逃脱。

int dir[4][2] = { {-1,0}, {0,1}, {1,0}, {0,-1} };

测试数据 2 分析 2：回退过程有恢复现场，dir 数组如下，即搜索方向为左、右、下、上顺序，整个搜索过程如图 2.8(b)所示，dfs 函数的执行过程如图 2.8(c)所示。从图 2.8(b)和图 2.8(c)可知，往左搜索走不通后，此时右边(即(1,3)位置)仍为'.'(在回退的时候恢复了)，从这个位置出发再进行搜索，将找到一条能成功逃脱的路径。因此，dfs 函数执行的结果是能成功逃脱。

int dir[4][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };

测试数据 2 分析 3：去掉回退过程的恢复现场操作，在图 2.9(c)中，“(1,3) = '.' ;”等代码加上了删除线。dir 数组如下，即搜索方向为上、右、下、左顺序，整个搜索过程如图 2.9(b)所示，dfs 函数的执行过程如图 2.9(c)所示。dfs 函数执行的结果是能成功逃脱。

int dir[4][2] = { {-1,0}, {0,1}, {1,0}, {0,-1} };

测试数据 2 分析 4：去掉回退过程的恢复现场操作，在图 2.10(c)中，“(0,2) = '.';”等代码加上了删除线。dir 数组如下，即搜索方向为左、右、下、上顺序，整个搜索过程如图 2.10(b)所示，dfs 函数的执行过程如图 2.10(c)所示。由于没有恢复现场操作，在往左搜索走不通后，所有位置都被设置为墙壁，无法再进行搜索了，如图 2.10(d)所示。因此，dfs 函数执行的结

果是不能成功逃脱。

int dir[4][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };

为什么在回退过程中恢复现场？答案是：如果当前搜索方向行不通，该搜索过程要结束了，但并不代表其他搜索方向也行不通，所以在回退时必须还原到原来的状态，保证其他搜索过程不受影响。

代码如下：

```
char map[9][9];     //迷宫地图
int n, m, t;        //迷宫的大小，及迷宫的门会在第 t 秒开启
int di, dj;         //(di,dj)：门的位置
bool escape;        //是否成功逃脱的标志，escape 为 1 表示能成功逃脱
int dir[4][2]={ {0,-1},{0,1},{1,0},{-1,0} };//分别表示下、上、左、右 4 个方向
//已经到达(si,sj)位置，且已经花费 cnt 秒
void dfs( int si, int sj, int cnt )
{
    int i, temp;
    if( si>n || sj>m || si<=0 || sj<=0 )  return;  //边界
    if( si==di && sj==dj && cnt==t )               //成功逃脱
    {
        escape=1;  return;
    }
    //abs(x-ex)+abs(y-ey)表示现在所在的格子到目标格子的距离(不能走对角线)
    //t-cnt 是实际还需要的步数，将它们做差
    //如果 temp < 0 或者 temp 为奇数，那就不可能到达!
    temp=(t-cnt)-fabs(si-di)-fabs(sj-dj);  //搜索过程当中的剪枝
    if( temp<0 || temp%2 )  return;
    for( i=0; i<4; i++ )
    {
        if( map[ si+dir[i][0] ][ sj+dir[i][1] ] !='X')
        {
            //前进方向！将当前方格设置为墙壁‘X’
            map[ si+dir[i][0] ][ sj+dir[i][1] ]='X';
            dfs(si+dir[i][0], sj+dir[i][1], cnt+1); //从下一个位置继续搜索
            if(escape)  return;
            map[ si+dir[i][0] ][ sj+dir[i][1] ]='.'; //后退方向！恢复现场!
        }
    }
    return;
}
int main( )
{
    int i, j;       //循环变量
    int si, sj;     //小狗的起始位置
    while( scanf("%d%d%d", &n, &m, &t) )
    {
        if( n==0 && m==0 && t==0 )  break; //测试数据结束
```

```
            int wall=0;
            char temp;
            scanf( "%c", &temp );              //见下面的备注
            for( i=1; i<=n; i++ )
            {
                for( j=1; j<=m; j++ )
                {
                    scanf( "%c", &map[i][j] );
                    if( map[i][j]=='S' ){  si=i;  sj=j;  }
                    else if( map[i][j]=='D' ){  di=i;  dj=j;  }
                    else if( map[i][j]=='X' )  wall++;
                }
                scanf( "%c", &temp );
            }
            if( n*m-wall <= t ) //搜索前的剪枝
            {
                printf( "NO\n" );  continue;
            }
            escape=0;
            map[si][sj]='X';
            dfs( si, sj, 0 );
            if( escape )  printf( "YES\n" );    //成功逃脱
            else  printf( "NO\n" );
        }
        return 0;
    }
```

备注：

(1) 用 C 语言的 scanf 函数读入字符型数据(使用"%c"格式控制)时，会把上一行的换行符(ASCII 编码值为 10)读进来。因此在读入每一行迷宫字符前，要跳过上一行的换行符。

(2) 本程序两处地方使用了剪枝，分别是搜索前的剪枝和搜索过程中的剪枝，详见程序中的注释。而所谓**剪枝**，顾名思义，就是通过某种判断，避免一些不必要的搜索过程。形象地说，就是剪去了搜索过程中的某些“枝条”，故称剪枝。有关剪枝技术的介绍，请参考其他书籍和资料。

例 2.2　油田(Oil Deposits)

题目来源：

Mid-Central USA 1997, ZOJ1709, POJ1562

题目描述：

GeoSurvComp 地质探测公司负责探测地下油田。每次 GeoSurvComp 公司都是在一块长方形的土地上来探测油田。在探测时，他们把这块土地用网格分成若干个小方块，然后逐个分析每块土地，用探测设备探测地下是否有油田。方块土地底下有油田则称为 pocket，如果两个 pocket 相邻，则认为是同一块油田，油田可能覆盖多个 pocket。试计算长方形的土地上有多少个不同的油田。

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个网格。每个网格数据的第 1

行为两个整数：*m*、*n*，分别表示网格的行和列；如果 *m* = 0，则表示输入结束，否则 1≤*m* ≤100，1≤*n*≤100。接下来有 *m* 行数据，每行数据有 *n* 个字符(不包括行结束符)。每个字符代表一个小方块，如果为“*”，则代表没有石油，如果为“@”，则代表有石油，是一个 pocket。

输出描述：

对输入文件中的每个网格，输出网格中不同的油田数目。如果两块不同的 pocket 在水平、垂直或者对角线方向上相邻，则被认为属于同一块油田。每块油田所包含的 pocket 数目不会超过 100。

样例输入：

```
5 5
****@
*@@*@
*@**@
@@@*@
@@**@
0 0
```

样例输出：

```
2
```

分析：

从网格中某个“@”字符位置开始进行 DFS 搜索，可以搜索到跟该“@”字符位置同属一块油田的所有“@”字符位置。为避免重复搜索，可以在搜索的前进方向，将每个“@”字符位置替换成“*”字符。这样，从网格中每个“@”字符位置进行搜索，可以得到油田的数目。

在程序实现时，可以用下面的二维数组表示(*x*, *y*)位置的 8 个相邻方向，该二维数组依次表示左上、上、右上、右、右下、下、左下、左 8 个方向(顺时针顺序)相对(*x*, *y*)位置的 *x*、*y* 坐标增量。

int dir[8][2] = { {−1, −1}, {−1, 0}, {−1, 1}, {0, 1}, {1, 1}, {1, 0}, {1, −1}, {0, −1} };

另外，与例 2.1 中使用 scanf 函数(采用"%c"格式控制)读取迷宫中的字符时需要跳过上一行的换行符不同的是，在本题中，读入网格中的每行字符时，采用"%s"格式控制，这种输入方式会自动跳过每行末尾的换行符，所以不需要专门用调用 scanf 函数来读取换行符。

代码如下：

```
char grid[101][101];        //存储网格
int m, n;                   //网格的大小，即行和列
int dir[8][2]={ {-1,-1},{-1,0},{-1,1},{0,1},{1,1},{1,0},{1,-1},{0,-1} };
                            //方格的 8 个相邻方向
void DFS( int x, int y )    //从(x,y)位置进行深度优先搜索
{
    int i, xx, yy;
    grid[x][y]='*'; //将 grid[x][y]由@设置成*，保证不会再遍历这个方格了
    for( i=0; i<8; i++ )
    {
        xx=x+dir[i][0];
        yy=y+dir[i][1];
        if( xx<0 || yy<0 || xx>=m || yy>=n )  continue;
```

```
            if( grid[xx][yy] == '@' ) //如果相邻方格还是@，则继续搜索
                DFS(xx, yy);
        }
    }
    int main( )
    {
        int i, j;   //循环变量
        int count;  //对每个网格，统计得到的油田数目
        while( 1 )
        {
            scanf( "%d%d", &m, &n );
            if( m==0 )  break;
            for( i=0; i<m; i++ )  scanf( "%s", grid[i] );
            count=0;
            for( i=0; i<m; i++ )
            {
                for( j=0; j<n; j++ )
                {
                    if( grid[i][j]=='@' )
                    {
                        DFS(i,j);   //从(i,j)位置进行 DFS 搜索
                        count++;
                    }
                }
            }
            printf( "%d\n", count );
        }
        return 0;
    }
```

练　习

2.1　农田灌溉(Farm Irrigation), ZOJ2412

题目描述：

Benny 有一大片农田需要灌溉。农田是一个长方形，被分割成许多小的正方形。每个正方形中都安装了水管。不同的正方形农田中可能安装了不同的水管。一共有 11 种水管，分别用字母 A～K 标明，如图 2.11(a)所示。

Benny 农田的地图是由描述每个正方形农田中水管类型的字母组成的矩阵。例如，如果农田的地图为：

ADC

FJK

IHE

则农田中水管分布如图 2.11(b)所示。

某些正方形农田的中心有水源，因此水可以沿着水管从一个正方形农田流向另一个正方形农田。如果水可以流经某个正方形农田，则整个正方形农田可以全部被灌溉到。

Benny 想知道至少需要多少个水源，以保证整个长方形农田都能被灌溉到。例如，图 2.11(b)所示的农田至少需要 3 个水源，图中的圆点表示每个水源。

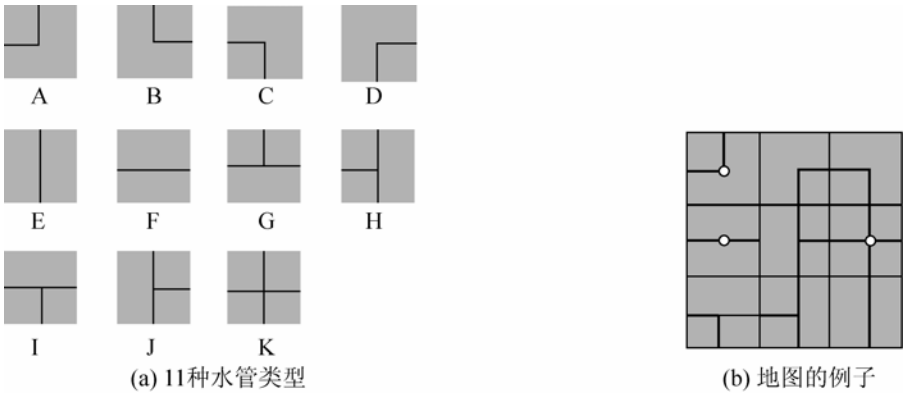


图 2.11　水管类型及农田的地图

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个整数 *M* 和 *N*，表示农田中有 *M* 行，每行有 *N* 个正方形。接下来有 *M* 行，每行有 *N* 个字符。字符的取值为'A'～'K'，表示对应正方形农田中水管的类型。当 *M* 或 *N* 取负值时，表示输入文件结束；否则 *M* 和 *N* 的值为正数，且其取值范围是 1≤*M*、*N*≤50。

输出描述：

对输入文件中的每个测试数据所描述的农田，输出占一行，为求得的所需水源数目的最小值。

样例输入：

```
3 3
ADC
FJK
IHE
-1 -1
```

样例输出：

```
3
```

2.2　Gnome Tetravex 游戏(Gnome Tetravex), ZOJ1008

题目描述：

哈特近来一直在玩有趣的 Gnome Tetravex 游戏。在游戏开始时，玩家会得到 *n*×*n*(*n*≤5)个正方形。每个正方形都被分成 4 个标有数字的三角形(数字的范围是 0～9)。这 4 个三角形分别被称为“左三角形”、“右三角形”、“上三角形”和“下三角形”。例如，图 2.12(a)所示是 2×2 的正方形的一个初始状态。

玩家需要重排正方形，到达目标状态。在目标状态中，任何两个相邻正方形的相邻三角形上的数字都相同。图 2.12(b)所示是一个目标状态的例子。

看起来这个游戏并不难。但是说实话，哈特并不擅长这种游戏，他能成功地完成最简单的游戏，但是当他面对一个更复杂的游戏时，他根本无法找到解法。

某一天，当哈特玩一个非常复杂的游戏的时候，他大喊到：“电脑在耍我！不可能解出

这个游戏。”对于这样可怜的玩家，帮助他的最好方法是告诉他游戏是否有解。如果他知道游戏是无解的，他就不需要再把如此多的时间浪费在它上面了。

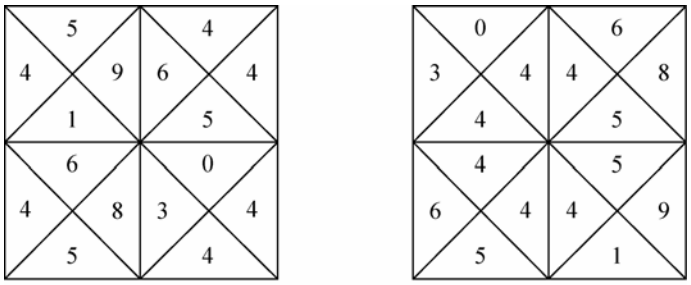


图 2.12　Gnome Tetravex 游戏

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个 Gnome Tetravex 游戏。每个游戏的第 1 行为一个整数 n，$1 \leqslant n \leqslant 5$，表示游戏的规模，该游戏中有 $n \times n$ 个正方形。

接下来有 $n \times n$ 行，描述了每个正方形中 4 个三角形中的数字。每一行为 4 个整数，依次代表上三角形、右三角形、下三角形和左三角形中的数字。

输入文件中最后一行为整数 0，代表输入结束。

输出描述：

对输入文件中的每个游戏，必须判断该游戏是否有解。对每个游戏，首先输出游戏的序号，接着是一个冒号和空格，然后是判断。如果该游戏有解，输出"Possible"，否则输出"Impossible"。

每两个游戏的输出之间有一个空行。注意不要输出多余的空格和空行。

样例输入：

```
2
5 9 1 4
4 4 5 6
6 8 5 4
0 4 4 3
2
1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
0
```

样例输出：

```
Game 1: Possible

Game 2: Impossible
```

2.3　红与黑(Red and Black), ZOJ2165, POJ1979

题目描述：

有一个长方形的房间，房间里的地面上布满了正方形的瓷砖，瓷砖要么是红色，要么是黑色。一男子站在其中一块黑色的瓷砖上。男子可以向他四周的瓷砖上移动，但不能移动到红色的瓷砖上，只能在黑色的瓷砖上移动。

本题的目的就是要编写程序，计算他在这个房间里可以到达的黑色瓷砖的数量。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个整数 *W* 和 *H*，分别表示长方形房间里 *x* 方向和 *y* 方向上瓷砖的数目。*W* 和 *H* 的值不超过 20。

接下来有 *H* 行，每行有 *W* 个字符，每个字符代表了瓷砖的颜色，这些字符的取值及含义如下。

(1) '.'——黑色的瓷砖。

(2) '#'——红色的瓷砖。

(3) '@'——表示该位置为黑色瓷砖，且一名男子站在上面，注意每个测试数据中只有一个'@'符号。

输入文件中最后一行为两个 0，代表输入文件结束。

输出描述：

对输入文件中每个测试数据，输出占一行，为该男子从初始位置出发可以到达的黑色瓷砖的数目(包括他初始时所处的黑色瓷砖)。

样例输入：

```
11 9
.#.........
.#.#######.
.#.#.....#.
.#.#.###.#.
.#.#..@#.#.
.#.#####.#.
.#.......#.
.#########.
...........
0 0
```

样例输出：

```
59
```

2.2　BFS 遍历

2.2.1　BFS 算法思想

广度优先搜索(Breadth First Search，BFS)是一个分层的搜索过程，没有回退过程，是非递归的。

BFS 算法的思想是：对一个无向连通图，在访问图中某一起始顶点 *v* 后，由 *v* 出发，依次访问 *v* 的所有未访问过的邻接顶点 $w_1, w_2, w_3, \cdots, w_t$；然后再顺序访问 $w_1, w_2, w_3, \cdots, w_t$ 的所有还未访问过的邻接顶点；再从这些访问过的顶点出发，再访问它们的所有还未访问过的邻接顶点，……，如此直到图中所有顶点都被访问到为止。

接下来以图 2.13(a)所示的无向连通图为例解释 BFS 搜索过程。假设在多个未访问过的邻接顶点中进行选择时，按顶点序号从小到大的顺序进行选择，比如顶点 *A* 有 3 个邻接顶点，即 *B*、*E* 和 *D*，则按 *B*、*D* 和 *E* 的顺序依次访问。

对图 2.13(a)所示的无向连通图，采用 BFS 思想搜索的过程如下。

(1) 访问顶点 *A*，这是第 1 层。

(2) 访问顶点 *A* 的 3 个邻接顶点 *B*、*D* 和 *E*，这是第 2 层。

(3) 访问顶点 *B* 的未访问过的邻接顶点(即顶点 *C*)，访问顶点 *D* 的未访问过的邻接顶点(即顶点 *F*)，顶点 *E* 没有未访问过的邻接顶点，这是第 3 层。

(4) 访问顶点 *C* 的未访问过的邻接顶点(即顶点 *G*)，访问顶点 *F* 的未访问过的邻接顶点(即顶点 *H*)，这是第 4 层。

(5) 顶点 *G* 没有未访问过的邻接顶点，访问顶点 *H* 的未访问过的邻接顶点(即顶点 *I*)，这是第 5 层。

至此，BFS 搜索完毕。图 2.13(a)中各顶点旁的数字表示 BFS 过程中各顶点的访问顺序，图 2.13(b)为**广度优先搜索生成树**：用访问 *n* 个顶点时经过的 *n*-1 条边，将 *n* 个顶点连接成一棵树。

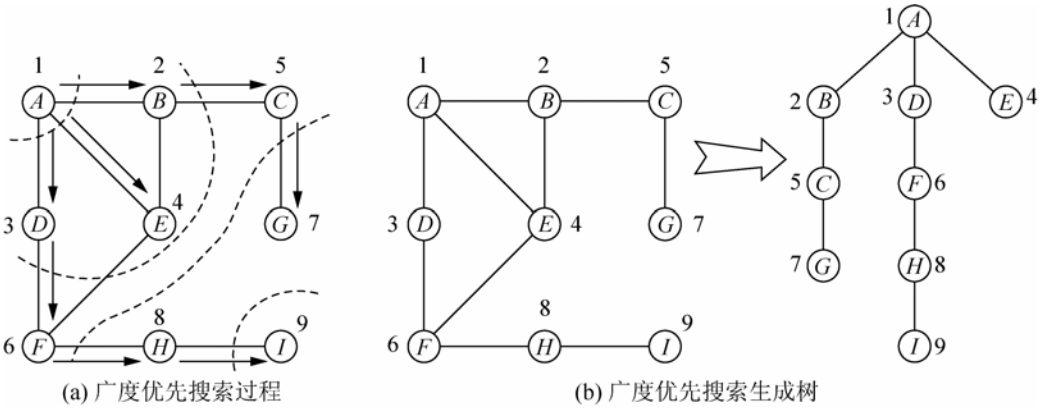


图 2.13　广度优先搜索

2.2.2　BFS 算法的实现及复杂度分析

1. BFS 算法的实现

与深度优先搜索过程一样，为避免重复访问，也需要一个状态数组 visited[*n*]，用来存储各顶点的访问状态。如果 visited[*i*] = 1，则表示顶点 i 已经访问过；如果 visited[*i*] = 0，则表示顶点 i 还未访问过。初始时，各顶点的访问状态均为 0。

为了实现逐层访问，BFS 算法在实现时需要使用一个队列，以记忆正在访问的这一层和上一层的顶点，以便于向下一层访问(约定在队列中，取出元素的一端为队列头，插入元素的一端为队列尾)。图 2.13(a)所示的搜索过程中，队列的变化如图 2.14 所示。具体如下(图 2.14 中的序号跟以下的序号是一一对应的；初始时，队列为空)。

(1) 访问顶点 *A*，然后把顶点 *A* 入队列。

(2) 取出队列头的顶点，即顶点 *A*，然后依次访问顶点 *A* 的 3 个邻接顶点 *B*、*D* 和 *E*，并把这 3 个顶点入队列。

(3) 取出此时队列头的顶点，即顶点 *B*，然后访问顶点 *B* 的未访问过的邻接顶点，即顶点 *C*，并把顶点 *C* 入队列。

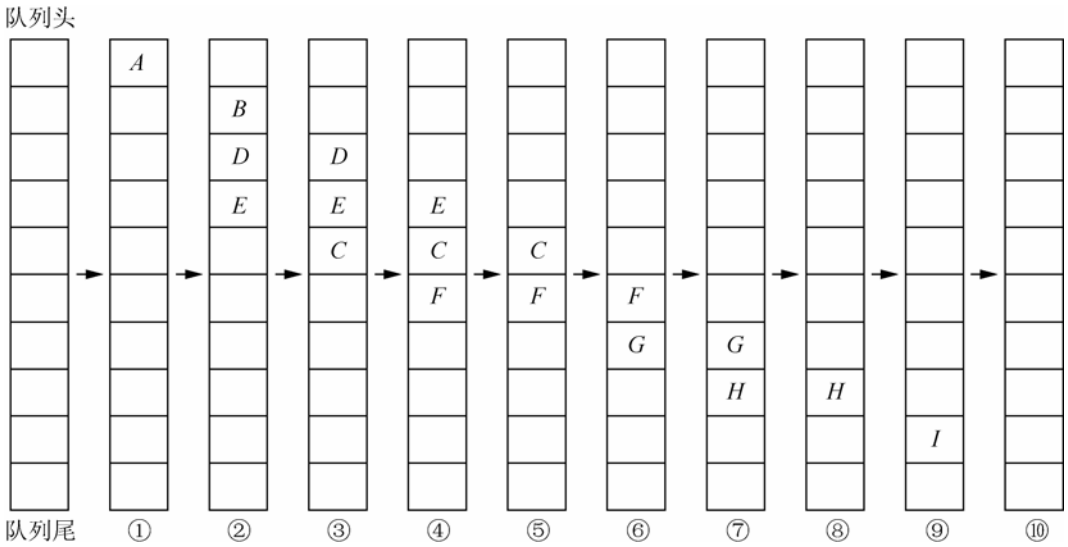


图 2.14　广度优先搜索过程的实现

(4) 取出此时队列头的顶点，即顶点 *D*，然后访问顶点 *D* 的未访问过的邻接顶点，即顶点 *F*，并把顶点 *F* 入队列。

(5) 取出此时队列头的顶点，即顶点 *E*，而顶点 *E* 已经没有未访问过的邻接顶点。

(6) 取出此时队列头的顶点，即顶点 *C*，然后访问顶点 *C* 的未访问过的邻接顶点，即顶点 *G*，并把顶点 *G* 入队列。

(7) 取出此时队列头的顶点，即顶点 *F*，然后访问顶点 *F* 的未访问过的邻接顶点，即顶点 *H*，并把顶点 *H* 入队列。

(8) 取出此时队列头的顶点，即顶点 *G*，而顶点 *G* 已经没有未访问过的邻接顶点。

(9) 取出此时队列头的顶点，即顶点 *H*，然后访问顶点 *H* 的未访问过的邻接顶点，即顶点 *I*，并把顶点 *I* 入队列。

(10) 取出此时队列头的顶点，即顶点 *I*，而顶点 *I* 已经没有未访问过的邻接顶点；至此，队列为空，BFS 搜索执行完毕。

BFS 执行过程中，各顶点的访问顺序依次为：*A*→*B*→*D*→*E*→*C*→*F*→*G*→*H*→*I*。

如果用邻接表存储图，则 BFS 算法实现的伪代码如下。

```
BFS( 顶点 i )    //从顶点i 进行广度优先搜索
{
    visited[ i ]=1;  //将顶点i 的访问标志置为1
    将顶点i 入队列;
    while( 队列不为空 )
    {
        取出队列头的顶点，设为k
        p=顶点k 的边链表表头指针
        while( p 不为空 )
        {
            //设指针p 所指向的边结点所表示的边的另一个顶点为顶点j
            if( 顶点j 未访问过 )
            {
```

```
                将顶点j 的访问标志置为1
                将顶点j 入队列
            }
            p=p->nextarc;   //p 移向下一个边结点
        }//end of while
    }//end of while
}//end of BFS
```

如果用邻接矩阵存储图(设顶点个数为 n)，则 BFS 算法实现的伪代码如下。

```
BFS( 顶点 i )    //从顶点i 进行广度优先搜索
{
    visited[ i ]=1;   //将顶点i 的访问标志置为1
    将顶点i 入队列;
    while( 队列不为空 )
    {
        取出队列头的顶点，设为k
        for( j=0; j<n; j++ )    //对其他所有顶点j
        {
            //j 是k 的邻接顶点，且顶点j 没有访问过
            if( Edge[k][j]==1 && !visited[j] )
            {
                将顶点j 的访问标志置为1
                将顶点j 入队列
            }
        }//end of for
    }//end of while
}//end of BFS
```

2. 算法复杂度分析

设无向图有 n 个顶点，有 m 条边。

如果使用邻接表存储图，对从队列头取出来的每个顶点 k，首先要取出该顶点的边链表头指针，然后沿着该顶点的边链表中的每个边结点，把未访问过的邻接顶点入队列。在这个过程每个顶点访问各一次，$2m$ 个边结点各访问一次，所以总的时间代价为 $O(n+2m)$。

如果用邻接矩阵存储图，由于在邻接矩阵中只是间接地存储了边的信息，所以对从队列头取出来的每个顶点 k，要循环检测无向图中的其他每个顶点 j(不管是否与顶点 k 相邻)，判断 j 是否跟 k 相邻且是否访问过；另外，每个顶点都要入队列，都要从队列头取出，进行判断，所以总的时间代价为 $O(n^2)$。

2.2.3 关于 DFS 算法和 BFS 算法的说明

1. DFS 算法

深度优先搜索算法的思路很简单，因此很好理解，但它求得的解不是最优的；并且一旦某个分支可以无限地搜索下去(假定结点有无穷多个)，但沿着这个分支搜索找不到解，则算法将不会停止、也找不到解，解决的方法可以采用有界深度优先搜索，本文不作进一步的讨论。

2. BFS 算法

如果某个问题有解，则采用广度优先搜索必能找到解，且找到的解的步数是最少的，解是最优的，如例 2.4；当然有些题目所要求的最优解不是简单的步数最少，而是附加了一些其他条件，如访问时间、访问代价等，则在采用广度优先搜索算法时应该做一些灵活的改动，如例 2.3。

2.2.4　例题解析

以下通过 3 道例题的分析，再详细介绍广度优先搜索的算法思想及其实现方法。

例 2.3　营救(Rescue)

题目来源：

ZOJ Monthly, October 2003, ZOJ1649

题目描述：

Angel 被 MOLIGPY 抓住了，她被关在监狱里。监狱可以用一个 $N\times M$ 的矩阵来描述，$1<N$、$M\leqslant 200$。监狱由 $N\times M$ 个方格组成，每个方格中可能为墙壁、道路、警卫、Angel 或 Angel 的朋友。

Angel 的朋友想去营救 Angel。他的任务是接近 Angel。约定“接近 Angel”的意思是到达 Angel 被关的位置。如果 Angel 的朋友想到达某个方格，但方格中有警卫，那么必须杀死警卫，才能到达这个方格。假定 Angel 的朋友向上、下、左、右移动 1 步用时用 1 个单位时间，杀死警卫用时也用 1 个单位时间。假定 Angel 的朋友很强壮，可以杀死所有的警卫。

试计算 Angel 的朋友接近 Angel 至少需要多长时间，只能向上、下、左、右移动，而且墙壁不能通过。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个整数 N 和 M，接下来有 N 行，每行有 M 个字符："."代表道路，"a"代表 Angel，"r"代表 Angel 的朋友，"#"代表墙壁，"x"代表警卫(注意，每个测试数据中字符"a"和"r"均只有一个)。输入数据一直到文件尾。

输出描述：

对输入文件中的每个测试数据，输出一个整数，表示接近 Angel 所需的最少时间。如果无法接近 Angel，则输出"Poor ANGEL has to stay in the prison all his life."。

样例输入：

```
7 8
#.######.
#a.#xr..
#.x#xx..
..xxxx.#
#.......
.#......
........
```

样例输出：

```
12
```

分析：

本题要求从 r 位置出发到达 Angel 所在位置并且所需时间最少，适合采用 BFS 求解。

但是 BFS 算法求出来的最优解通常是步数最少的解，而在本题中，步数最少的解不一定是最优解。

例如，样例输入中第 2 个测试数据所描绘的监狱如图 2.15 所示。从 *r* 到 *a* 所需的最少步数为 8 步，其中图 2.15(a)、图 2.15(b)和图 2.15(c)所表示的路线步数均为 8 步，所花费的时间分别为 13、13 和 14；而图 2.15(d)所表示的路线步数为 12 步，所花费的时间为 12。在该测试数据中，图 2.15(d)所表示的路线是最优解。因此在本题中，最优解不一定是步数最少的解。

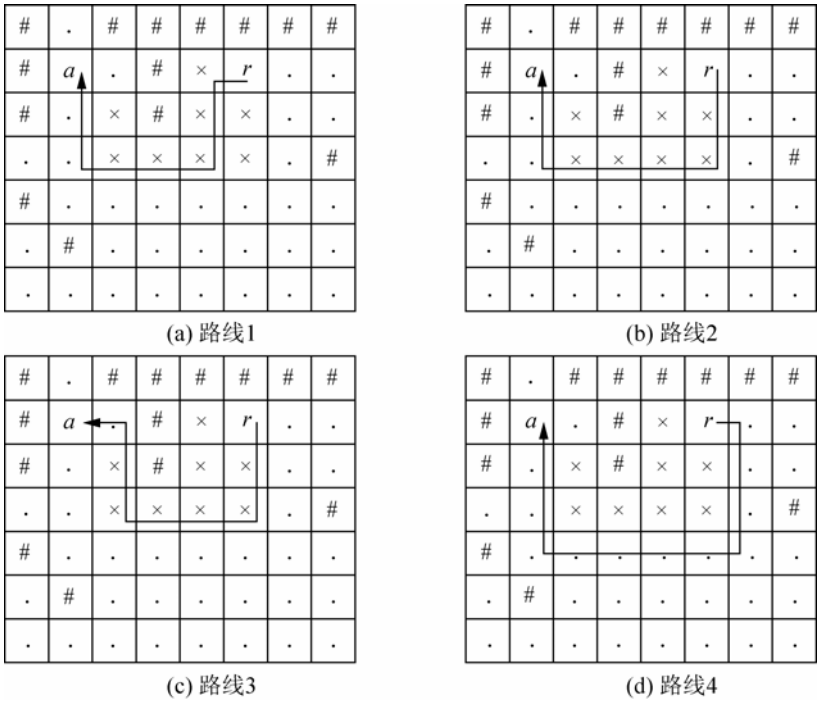


图 2.15　营救：最优解不一定是步数最少的解

为了求出最优解，本题采取如下的思路进行 BFS 搜索。

(1) 将 Angel 的朋友到达某个方格时的状态用一个结构体 point 表示，该结构体包含了 Angel 的朋友到达该方格时所走过的步数及所花费的时间；在 BFS 搜索过程中，队列中的结点是 point 型数据。

(2) 定义一个二维数组 mintime，mintime[*i*][*j*]表示 Angel 的朋友走到(*i*, *j*)位置所需最少时间；在 BFS 搜索过程中，从当前位置走到相邻位置(*x*, *y*)时，只有当该种走法比之前走到(*x*, *y*)位置所花时间更少，才会把当前走到(*x*, *y*)位置所表示的状态入队列，否则是不会入队列的。

(3) 在 BFS 搜索过程中，不能一判断出到达目标位置就退出 BFS 过程，否则求出来的最少时间仅仅是从 *r* 到达 *a* 最小步数的若干个方案中的最小时间，不一定是最优解；一定要等到队列为空、BFS 过程结束后才能求得最优解或者得出“无法到达目标位置”的结论。

另外，在本题中，并没有使用标明各位置是否访问过的状态数组 visited，也没有在 BFS 过程中将访问过的相邻位置设置成不可再访问，那么 BFS 过程会不会无限搜索下去呢？实

际上是不会的，因为从某个位置出发判断是否需要将它的相邻位置(*x*, *y*)入队列时，条件是这种走法比之前走到(*x*, *y*)位置所花时间更少；如果所花时间更少，则(*x*, *y*)位置会重复入队列，但不会无穷下去，因为到达(*x*, *y*)位置的最少时间肯定是有下界的。

代码如下：

```
#define MAXMN 200
#define INF 1000000     //走到每个位置所需时间的初始值为无穷大
using namespace std;
struct point            //表示到达某个方格时的状态
{
    int x, y;           //方格的位置
    int step;           //走到当前位置所进行的步数
    int time;           //走到当前位置所花时间
};
queue<point> Q;         //队列中的结点为当前 Angel 的朋友所处的位置
int N, M;               //监狱的大小
char map[MAXMN][MAXMN];//地图："."——道路，"a"——Angel，"r"——Angel 的朋友
                        //"#"——墙壁，"x"——警卫
int mintime[MAXMN][MAXMN]; //走到每个位置所需最少时间
int dir[4][2]={ {-1,0},{0,1},{1,0},{0,-1} }; //4 个相邻方向：上、右、下、左
int ax, ay; //Angel 所处的位置
int BFS( point s )          //从位置 s 开始进行 BFS 搜索
{
    int i;                  //循环变量
    Q.push( s );
    point hd;               //从队列头出队列的位置
    while( !Q.empty( ) )    //当队列非空
    {
        hd=Q.front( );  Q.pop( );
        for( i=0; i<4; i++ )
        {
            //判断能否移动到相邻位置(x,y)
            int x=hd.x + dir[i][0],  y=hd.y + dir[i][1];
//排除边界和墙壁
            if( x>=0 && x<=N-1 && y>=0 && y<=M-1 && map[x][y]!='#' )
            {
                point t;    //向第 i 个方向走一步后的位置
                t.x=x;  t.y=y;
                t.step=hd.step+1;
                t.time=hd.time+1;
                if( map[x][y]=='x' )  t.time++; //杀死警卫需要多花 1 个单位时间
                //如果这种走法比之前走到(x,y)位置所花时间更少，则把 t 入队列；
                //否则 t 无须入队列
                if( t.time<mintime[x][y] )
                {
                    mintime[x][y]=t.time;
                    Q.push( t );    //把 t 入队列
```

```
                }
            }//end of if
        }//end of for
    }//end of while
    return mintime[ax][ay];
}
int main( )
{
    int i, j;   //循环变量
    while( scanf( "%d%d", &N, &M )!=EOF )
    {
        memset( map, 0, sizeof(map) );
        for( i=0; i<N; i++ )  scanf( "%s", map[i] );   //读入地图
        int sx, sy; point start;                       //Angel 朋友的位置
        for( i=0; i<N; i++ )
        {
            for( j=0; j<M; j++ )
            {
                mintime[i][j]=INF;
                if( map[i][j]=='a' ) { ax=i; ay=j; }
                else if( map[i][j]=='r' ) { sx=i; sy=j; }
            }
        }
        start.x=sx; start.y=sy; start.step=0; start.time=0;
        mintime[sx][sy]=0;
        int mint=BFS( start ); //返回到达 Angel 位置的最少时间，有可能为 INF
        if( mint<INF )  printf( "%d\n", mint );
        else  printf( "Poor ANGEL has to stay in the prison all his life.\n" );
    }
    return 0;
}
```

例 2.4　公共汽车通票(Bus Pass)

题目来源：

The 2007 Benelux Algorithm Programming Contest, ZOJ2913

题目描述：

乘坐公共汽车途经一些地区，旅程费用为每张车票费用的总和。因此，想知道如果买通票是否更优惠一些。

公交系统按如下方式运转：当买一张公共汽车通票时，必须指定一个中心地区，以及一个星形阈值；当持通票在某个地区乘坐公共汽车时，只要该地区离中心地区的距离(并不是路程的距离，而是路程上地区数目)小于星形阈值，则可以免费乘坐。例如，如果选择的星形阈值为 1，则只能在中心地区免费乘坐公交汽车；如果选择的星形阈值为 2，则在中心地区及与中心地区相邻的地区里都可以免费乘坐公交汽车等。

现列出了某人经常要乘坐的公交线路，希望能确定一个最小的星形阈值，使得所有的公交线路在通票范围内都是免费的。但这并不是一件容易的工作。例如，如图 2.16 所示的地区分布图。

想乘坐公交车从 *A* 到 *B*，以及从 *B* 到 *D*，则最好的中心地区编号为 7400，这样需要选择的阈值为最小值 4。注意，在公交线路中并不需要途经选择的中心地区。

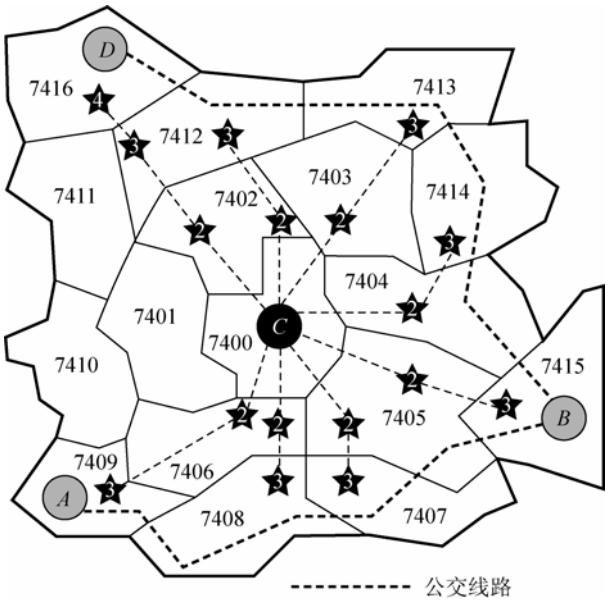


图 2.16　地区分布图及公交线路

输入描述：

输入文件的第 1 行为一个整数 *T*，1≤*T*≤100，表示输入文件中测试数据的数目。每个测试数据的格式如下。

第 1 行为两个整数：*nz*(2≤*nz*≤9 999)和 *nr*(1≤*nr*≤10)，分别表示地区的数目和公交线路的数目。

接下来有 *nz* 行，描述了每个地区。每行首先是两个整数：id_i(1≤id_i≤9 999)和 mz_i(1≤mz_i≤10)，分别表示第 *i* 个地区的编号和与该地区相邻的地区数目，然后是 mz_i 个整数，分别是与该地区相邻地区的编号。

接下来有 *nr* 行，描述了每条公交路线。每行首先是一个整数 mr_i(1≤mr_i≤20)，表示第 i 条公交线路途经的地区数目，然后是 mr_i 个整数，表示该线路通过的地区的编号，按照该线路的途经顺序排列。

所有地区之间都是连通的，直接连接或通过其他地区连接。

输出描述：

对输入文件中的每个测试数据，输出一行，为两个整数，分别表示最小的星形阈值和取得最小阈值的中心地区的编号。如果存在多个符合要求的中心地区，则输出编号最小的中心地区。

样例输入：

```
1
17 2
7400 6 7401 7402 7403 7404 7405 7406
7401 6 7412 7402 7400 7406 7410 7411
```

样例输出：

```
4 7400
```

```
7402 5 7412 7403 7400 7401 7411
7403 6 7413 7414 7404 7400 7402 7412
7404 5 7403 7414 7415 7405 7400
7405 6 7404 7415 7407 7408 7406 7400
7406 7 7400 7405 7407 7408 7409 7410 7401
7407 4 7408 7406 7405 7415
7408 4 7409 7406 7405 7407
7409 3 7410 7406 7408
7410 4 7411 7401 7406 7409
7411 5 7416 7412 7402 7401 7410
7412 6 7416 7411 7401 7402 7403 7413
7413 3 7412 7403 7414
7414 3 7413 7403 7404
7415 3 7404 7405 7407
7416 2 7411 7412
5 7409 7408 7407 7405 7415
6 7415 7404 7414 7413 7412 7416
```

分析：

这是一道典型的 BFS 题目。其基本思路是：从每条线路上的每个地区 *z* 出发进行 BFS 遍历；对每个地区 *j*，如果地区 *j* 是最终求得的中心地区，则要保证从它出发能到达每条线路上每个地区 *z*，并且选择的星形阈值要尽可能小，因此统计每条线路上每个地区 *z* 到地区 *j* 最短距离中的最大值，这个最大值记录在数组元素 res[*j*]中；最后求得的最小的星形阈值就是每个地区 *j* 的 res[*j*]的最小值，中心地区就是取得最小值的地区 *j*。

以图 2.16 所示的地区分布图及公交路线为例加以解释，为了帮助读者理解，在图 2.17 中特意将地区编号从 0 开始计起。

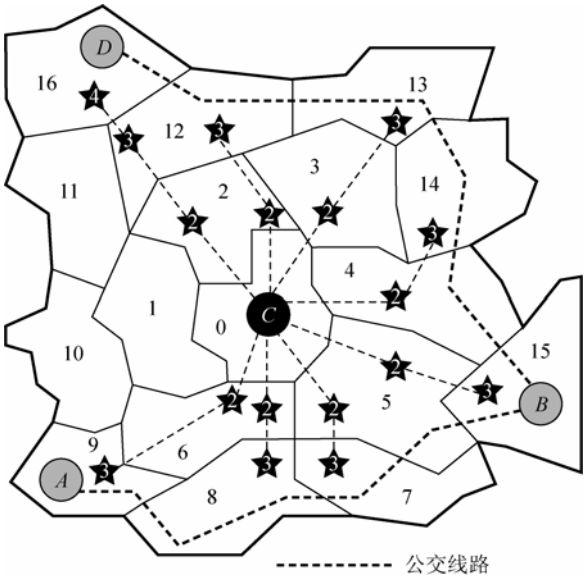


图 2.17　地区分布图及公交线路(地区编号从 0 开始计起)

针对图 2.17 所示的地区分布图，求得的 res[0]～res[16]的取值分别为：4、4、4、4、4、5、4、5、5、5、5、5、4、5、5、5、5。举例说明 res[*j*]的含义：res[0]的值为 4，含义是地区 0 距离每条线路上的每个地区其最短距离中的最大值为 4，之所以要取最大值，是因为要保证从中心地区出发能到达公交线路上的所有地区。本题最终要求的是最小的 res[*j*]，很明显，存在多个解，题目中要输出的是编号最小的 *j*。

另外，本题中并没有提及地区的编号是连续的，所以 *nz* 个地区的编号分布在 0～9 999 之间。在下面代码中，用数组 *mz* 在存储 *nz* 个地区各自的相邻地区数目时，并不是将这 *nz* 个地区连续地存储在数组 *mz* 中，而是将编号为 *j* 的地区的相邻地区数目存储在数组元素 mz[*j*]中。

代码如下：

```
#define ZMAX 10000  //地区编号的最大值
#define INF 100000  //表示一个极大值
using namespace std;
int nz, nr;         //地区的数目、公交线路的数目
int mz[ZMAX];       //mz[i]为与第 i 个地区相邻的地区数目
//"邻接矩阵", Edge[i][j]表示编号为 i 的地区的第 j 个相邻地区的编号
int Edge[ZMAX][10];
int res[ZMAX];      //res[i]表示每条线路上每个地区到地区 i 距离中的最大值
int cur;            //记录当前公交的站点次序，cur==0 表示当前是第 1 站
int reach[ZMAX];    //reach[s]==cur 表示地区 s 在第 cur+1 站已访问
int max( int x, int y )
{
    return ( x>y ) ? x:y;
}
//从地区 s 出发进行 BFS 遍历(遍历其他所有顶点)
void BFS( int s )
{
    int i, a, b;
    int val, at; //val 用于记录层数，即距离；at 用于表示 BFS 过程中的当前结点
    queue<int> q[2];        //滚动队列
    a=0, b=1, val=0;
    if( reach[s]<cur )
    {
        q[b].push(s);
        reach[s]=cur;
        res[s]=max( res[s], val+1 );
    }
    while( !q[b].empty( ) )
    {
        swap( a, b );   //滚动队列
        val++;
        while( !q[a].empty( ) ) //处理所有当前结点
        {
            at=q[a].front( );
            q[a].pop( );
            for( i=0; i<mz[at]; i++ )
```

```
            {
                if( reach[Edge[at][i]]<cur )
                {
                    q[b].push( Edge[at][i] );
                    reach[ Edge[at][i] ]=cur;
                    res[ Edge[at][i] ]=max( res[ Edge[at][i] ], val+1 );
                }
            }
        }
    }
}
int main( )
{
    int T;              //输入文件中测试数据个数
    int i, j, t;        //循环变量
    int id;             //地区编号
    int mr;             //每条公交线路途经的地区数目
    int ret, center;    //最小星形阈值和取得最小星形阈值的中心地区编号
    scanf( "%d", &T );
    for( t=0; t<T; t++ )
    {
        memset( reach,-1, sizeof(reach) );
        memset( res, 0, sizeof(res) );
        cur=0;
        scanf( "%d%d", &nz, &nr );
        for( i=0; i<nz; i++ )
        {
            scanf( "%d", &id );
            scanf( "%d", &mz[id] );
            for( j=0; j<mz[id]; j++ )  scanf( "%d",&Edge[id][j] );
        }
        for( i=0; i<nr; i++ )
        {
            scanf( "%d", &mr );
            for( j=0; j<mr; j++ )
            {
                scanf( "%d", &id );
                BFS( id ); //从每条公交线路上的每个地区 id 出发进行 BFS 编历
                cur++;
            }
        }
        ret=INT_MAX, center=-1;
        for( i=0; i<10000; i++ ) //求最小的 res[i]及对应的地区编号
        {
            if( reach[i]==cur-1 && res[i]<ret )
            {
                ret=res[i];
                center=i;
            }
```

```
            }
            printf( "%d %d\n", ret, center );
        }
        return 0;
    }
```

例 2.5　蛇和梯子游戏(Snakes & Ladders)

题目来源：

Arab and North African Region, 2002

题目描述：

蛇和梯子游戏是一个非常流行的游戏。游戏采用 $N \times N$ 的棋盘，方格编号从 1 到 N^2，如图 2.18(a)所示。棋盘中分布着蛇和梯子。玩家 A 和 B 的起始位置都在方格 1 处，每个玩家只能顺着如图 2.18(b)所示的方向走棋；每次走棋之前，先投骰子，得到一个点数，然后走该点数步。

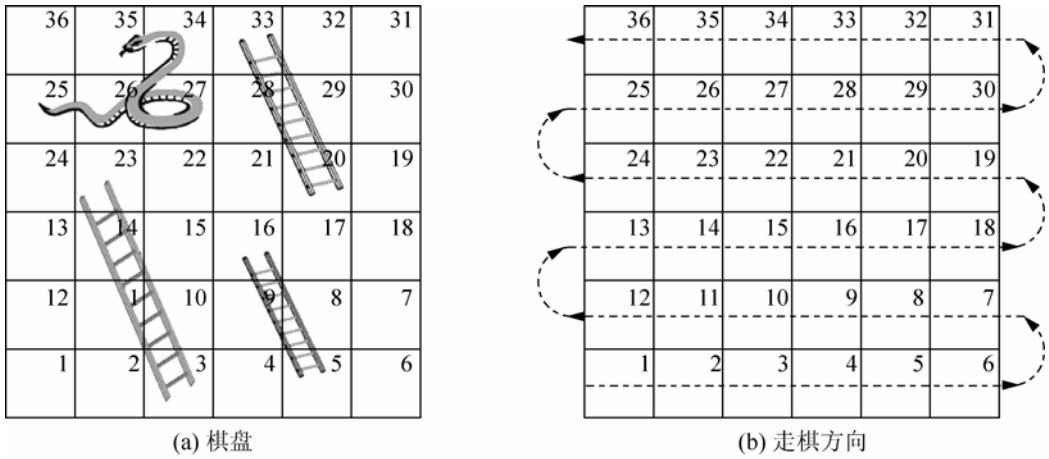


图 2.18　蛇和梯子游戏

惩罚：如果玩家的落脚地刚好是在蛇头，则降至蛇尾所在方格处。

奖励：如果玩家落脚地刚好是在梯子底部，则顺着梯子爬到顶部所在方格处。

最先到达 N^2 那一方获胜。

关于棋盘分布有两点值得注意。

(1) 第 1 个和最后一个方格处没有梯子和蛇。

(2) 并且蛇和梯子不能相邻，也就是在任何放置两者首尾的方格之间至少还有一个未放置任何东西的格子。

Fadi 希望有人编写一个程序，帮助他赢得比赛。Fadi 是一个职业骗子，他投骰子可以得到任何期望的点数(1 至 6)。Fadi 希望知道他至少需要投多少次骰子才能赢得比赛。

例如，在图 2.18(a)所示的棋盘布局图中，Fadi 投 3 次骰子就可以赢得比赛：首先投骰子得到点数 4，到达方格 5 的位置，顺着梯子爬到方格 16 的位置；然后投骰子得到点数 4，到达方格 20 的位置，然后顺着梯子爬到方格 33 的位置，接下来只要投骰子得到点数 3 就可以赢得比赛了。

输入描述：

输入文件的第 1 行是一个整数 D，代表输入文件中测试数据的个数。

每个测试数据用 3 行来描述。

(1) 第 1 行包含了 3 个整数：N、S 和 L，N 是棋盘的大小，S 是蛇的个数，L 是梯子的个数，其中 $0<N\leqslant 20$，$0<S<100$，$0<T<100$。

(2) 第 2 行包含了 S 个整数对，每对整数描述了一条蛇的起止方格位置，第 1 个整数是蛇的起始位置(蛇头)，第 2 个整数是蛇的终止位置(蛇尾)，注意方格的序号是以 1 开始计起的。

(3) 第 3 行包含了 L 对整数，描述了 L 个梯子的信息，每对整数的第 1 个整数是梯子的起始位置(底部)，第 2 个整数是梯子的终止位置(顶部)。

输出描述：

对输入文件中每个测试数据，输出占一行，为一个整数，表示为了到达 N^2 方格处，至少需要投骰子的数目。

样例输入：

```
2
6 1 3
35 25
3 23 5 16 20 33
```

样例输出：

```
3
```

分析：

首先，蛇和梯子占据哪些方格并不重要，只需要知道蛇和梯子的起止位置即可。

其次，蛇和梯子并不需要看成是两个不同的东西，因为蛇和梯子的本质都是“单向传送”的工具，只不过梯子的底部是入口而顶部是出口，蛇头是入口而蛇尾是出口。而且在输入数据里，表示蛇和梯子起止位置的数据都是入口在前，出口在后。

还有，这道题目能不能用贪心算法来求解？答案是不可以的。一方面，选用最大的点数(即点数 6)不能保证最好的效率；另一方面，在图 2.18(a)中，如果为了能利用跨度最大的梯子(3, 23)，从而第 1 次投骰子，得到的点数为 2，这种方法也不能保证所需骰子的数目最少。

本题的求解可以采用广度优先搜索算法，其思想可以用图 2.19 来表示。具体方法为：将玩家处于第 X 个方格处的状态简称为结点 X；初始时，玩家的位置在 1 号方格处，投一个骰子，点数为 1～6，可以扩展出 6 个结点，依次判断是否已经达到 N^2 方格处；如果没有达到，则对这些结点继续扩展，直至到达 N^2 方格处为止。

但是，直接应用 BFS 算法思想求解，存在的问题如下。

(1) 解空间比较大。如图 2.19 所示，第 1 层有 1 个结点，第 2 层有 6 个结点，第 3 层有 36 个结点，第 4 层有 216 个结点等。

(2) 会产生重复的结果。当棋盘中没有蛇和梯子时，玩家所处的位置最多为 $N\times N$ 个，例如 N=6，最多也就 36 个结点，很明显前面提到的结点中有很多是重复的。

为了有效地利用 BFS 算法思想求解本题，需要对蛇和梯子游戏作进一步分析。

(1) 玩家的起始位置为结点 1，第 1 步过后(第 1 次投骰子，点数从 1 到 6)，扩展的结点是 2，23，4，16，6，7，如图 2.20 所示。

(2) 考虑第 2 步，以结点 2 为例，再走 1 步可以扩展出的结点为：23、4、16、6、7、8。

……

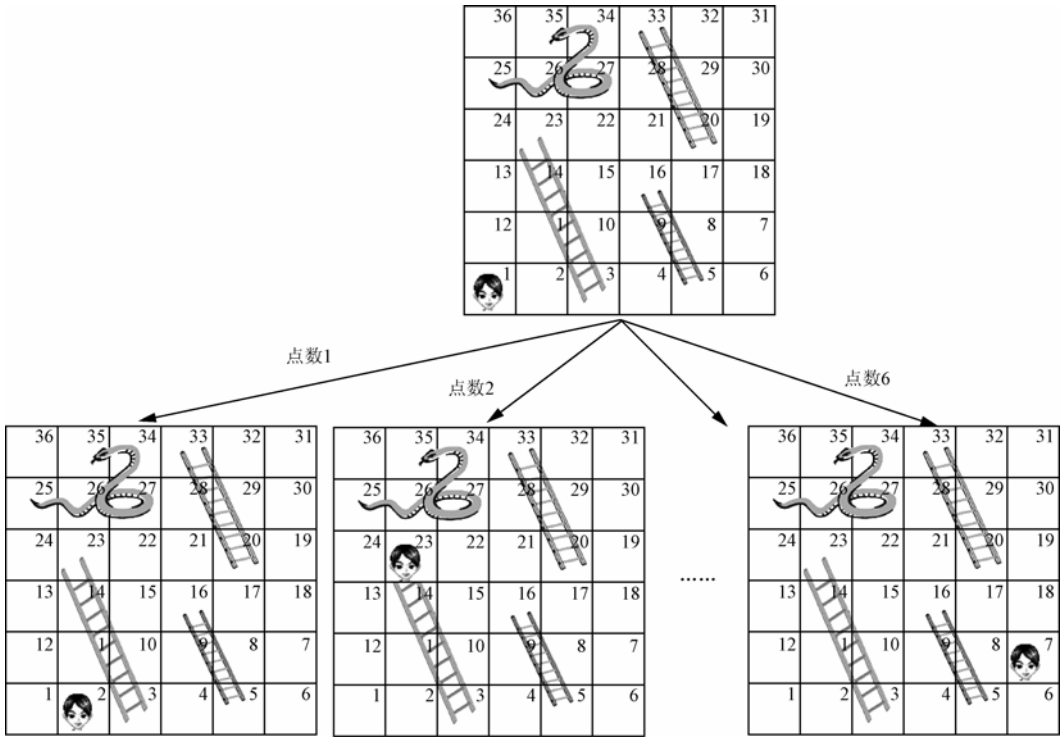


图 2.19　广度优先搜索算法求解蛇和梯子游戏

图 2.20　蛇和梯子游戏：结点扩展

在走了若干步之后，对于一个特定的格子实际上只有两种可能的状态。

(1) 在走了这些步数之后存在一种方案使得玩家的位置位于此格中；

(2) 不存在这样的一种方案。

因此，只需要记住每次投骰子后玩家可能到达的位置，直到可以到达第 N^2 个方格处时停止扩展。

如图 2.21 所示，从结点 1 出发，投 3 次骰子后就可以到达第 N^2 个方格处，因此至少需要投 3 个骰子。

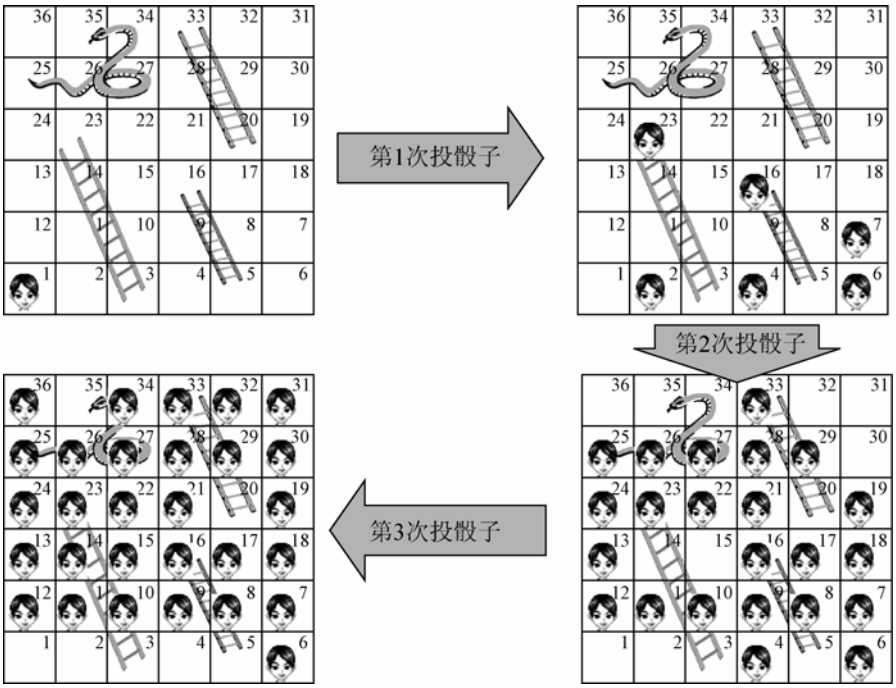


图 2.21　蛇和梯子游戏：扩展结点直至 N^2 方格可达

实现时可以用一个 $N\times N$ 大小的数组来描述若干步之后可以到达的方格的集合，其中每一个元素描述一个方格的状态，0 表示不存在一种方案到达，1 表示存在至少一种方案到达。这样，从表示第 n 步状态的数组，完全可以推出表示第 n+1 步状态的数组。而且在第 n+1 步状态的数组得到之后，表示第 n 步状态的数组也就没有利用价值了。一旦数组中表示最后一个格子的元素成为 1，就表示可以通过这个步数完成任务了。例如，图 2.18(a)所示的棋盘，每次投骰子后，数组的状态可以用下面的表 2-1 来表示。

表 2-1　每次投骰子后记录各位置的状态

	数组元素的内容
起始状态	10000000000000000000000000000000000000
第 1 步之后	01010110000000010000001000000000000000
第 2 步之后	00010111111110011110111111111110001000
第 3 步之后	00000111111111111111011111111111111101

具体实现方法如下。

(1) 用一个 $N\times N$ 大小的数组来描述若干步之后可以到达的方格的集合。

(2) 对于每个结点 X，以点数 Y(1～6)分别去扩展，对每个点数，都要考虑 M 个梯子和蛇，如果 X+Y 处刚好是蛇首或者梯子的底部，则需要按照规则上升或下降。

(3) 记住投骰子的次数 step，当数组的最后一个元素变为 1，即表明存在一种方案，使得投 step 次骰子就可以到达第 N^2 个方格处。

代码如下：

```
#define NMAX  20
#define SLMAX 200
struct SnakeAndLadder   //蛇和梯子
{
    int from, to;       //起止位置
};
int main( )
{
    int D;                  //测试数据的个数
    int N, S, L;            //每个测试数据中的N、S、L：棋盘规模、蛇和梯子数目
    int grid[NMAX*NMAX+1]; //棋盘(序号从1开始计)
    //备份上一次棋盘状态(序号从1开始计)
    int gridbak[ NMAX*NMAX+1 ];
    SnakeAndLadder obstacle[ 2*SLMAX ];    //障碍物，包括梯子和蛇
    int i, j, k, m; //循环变量
    int step;       //投骰子的数目
    int deal;       //如果落脚处是蛇(惩罚)首部或梯子(奖励)底部，则deal为1
    scanf( "%d", &D );
    for( i=0; i<D; i++ )    //处理每个测试数据
    {
        scanf( "%d%d%d", &N, &S, &L );
        for( j=0; j<S+L; j++ )
            scanf( "%d%d", &obstacle[j].from, &obstacle[j].to );
        memset( grid, 0, sizeof(grid) );   grid[1]=1;  //初始化状态数组
        step=0; //初始化骰子数目
        //只要第N^2个方格没有扩展为1，则继续按BFS进行扩展
        while( grid[N*N]==0 )
        {
            memcpy( gridbak, grid, sizeof(grid) );//备份上一步棋盘状态
            //grid[ ]数组用来保存在上一步棋盘状态(graibak[ ])下进行扩展后的状态
            memset( grid, 0, sizeof(grid) );
            //搜索所有的格子,最后一格不用搜索,因为在搜索过程结束前它一定为0
            for( j=1; j<=N*N-1; j++ )
            {
                if( gridbak[j]==0 ) //若在上一步无法到达此格则跳过
                    continue;
                //考虑点数1～6,走该点数步数后到达j+k位置
                for( k=1; k<=6; k++ )
                {
                    deal=0;
                    if( j+k>N*N )  break;
                    for( m=0; m<S+L; m++ )
                    {
                        //如果j+k位置是某个蛇(或梯子)的起始位置
                        //则沿着它到达终止位置
```

```
                        if( obstacle[m].from==j+k )
                        {
                            grid[ obstacle[m].to ]=1;  deal=1;
                            break;  //j+k 位置上最多只有
                                    //一个蛇(或梯子)的起止位置
                        }
                    }
                    //不利用蛇或梯子
                    if( deal==0 && grid[j+k]==0 )  grid[j+k]=1;
                }
            }
            step++; //骰子数加一
        }
        printf( "%d\n", step );
    }
    return 0;
}
```

练　习

2.4　倍数(Multiple), ZOJ1136, POJ1465

题目描述：

编写程序，实现：给定一个自然数 N，N 的范围为[0, 4 999]，以及 M 个不同的十进制数字 $X_1, X_2, \cdots, X_M$(至少一个，即 $M \geqslant 1$)，求 N 的最小的正整数倍数，满足：N 的每位数字均为 $X_1, X_2, \cdots, X_M$ 中的一个。

输入描述：

输入文件包含多个测试数据，测试数据之间用空行隔开。每个测试数据的格式为：第 1 行为自然数 N；第 2 行为正整数 M；接下来有 M 行，每行为一个十进制数字，分别为 $X_1, X_2, \cdots, X_M$。

输出描述：

对输入文件中的每个测试数据，输出符合条件的 N 的倍数；如果不存在这样的倍数，则输出 0。

样例输入：	**样例输出：**
22 3 7 0 1	110

2.5　蛇的爬动(Holedox Moving), ZOJ1361, POJ1324

题目描述：

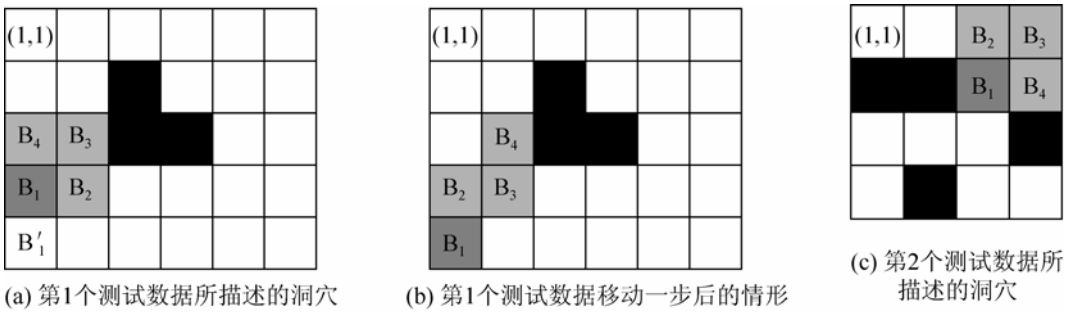
在冬天，天气最恶劣的时期，蛇待在洞穴里冬眠。当春天来临的时候，蛇苏醒了，爬到洞穴的出口，然后爬出来，开始它的新生活。

蛇的洞穴像一个迷宫，可以把它想象成一个由 $n \times m$ 个正方形区域组成的长方形。每个正方形区域要么被石头占据了，要么是一块空地，蛇只能在空地间爬动。洞穴的行和列都是有编号的，行和列的编号从 1 开始计起，且出口在(1,1)位置。

蛇的身躯，长为 L，用一块连一块的形式来表示。假设用 $B_1(r_1, c_1)$, $B_2(r_2, c_2)$, …, $B_L(r_L, c_L)$表示它的 L 块身躯，其中，B_i 与 B_i+1(上、下、左或右)相邻，$i = 1, \cdots, L-1$；B_1 为蛇头，B_L 为蛇尾。

为了在洞穴中爬动，蛇选择与蛇头(上、下、左或右)相邻的一个空的正方形区域，这个区域既没有被石头占据，也没有被它的身躯占据。当蛇头移动到这个空地，这时，它的身躯中其他每一块都移动它前一块身躯之前所占据的空地上。

例如，图 2.22(a)所示的洞穴(带有深色阴影的方格为蛇头，带有浅色阴影的方格为蛇身，黑色方格为石头)中，蛇的初始位置为：$B_1(4,1)$，$B_2(4,2)$，$B_3(3,2)$和 $B_4(3,1)$。在下一步，蛇头只能移动到 $B_1'(5, 1)$位置。蛇头移动到 $B_1'(5, 1)$位置后，则 B_2 移动到 B_1 原先所在的位置，B_3 移动到 B_2 原先所在的位置，B_4 移动到 B_3 原先所在的位置。因此移动一步后，蛇的身躯位于 $B_1(5, 1)$，$B_2(4, 1)$，$B_3(4, 2)$和 $B_4(3, 2)$，如图 2.22(b)所示。



(a) 第1个测试数据所描述的洞穴　(b) 第1个测试数据移动一步后的情形　(c) 第2个测试数据所描述的洞穴

图 2.22　蛇的爬动

给定洞穴的地图，以及蛇的每块身躯的初始位置。试编写程序，计算蛇头爬到出口(1, 1)位置所需的最少步数。

输入描述：

输入文件包含多个测试数据。每个测试数据的第 1 行为 3 个整数：n，m 和 L，$1 \leqslant n, m \leqslant 20$，$2 \leqslant L \leqslant 8$，分别代表洞穴的行、列，以及蛇的长度；接下来有 L 行，每行有一对整数，分别表示行和列，代表蛇的每一块身躯的初始位置，依顺序分别为 $B_1(r_1, c_1) \sim B_L(r_L, c_L)$；接下来一行包含一个整数 K，表示洞穴中石头的数目；接下来的 K 行，每行包含一对整数，分别表示行和列，代表每一块石头的位置。每两个测试数据之间有一个空行。输入文件的最后一行为 3 个 0，代表输入结束。

注意：B_i 总是与 B_i+1 相邻，$1 \leqslant i \leqslant L-1$，出口位置(1, 1)从不会被石头占据。

输出描述：

对输入文件中的每个测试数据，输出一行：首先是测试数据的序号；然后是蛇爬到洞穴出口所需的最少步数，如果没有解(如第 2 个测试数据如图 2.22(c)所示)，则输出-1。

样例输入：

```
5 6 4
4 1
```

样例输出：

```
Case 1: 9
Case 2: -1
```

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

4 4 4
2 3
1 3
1 4
2 4
4
2 1
2 2
3 4
4 2

0 0 0
```

注解：第 1 个测试数据中，蛇头按如下顺序移动，所需的步数是最少的：(4, 1)，(5, 1)，(5, 2)，(5, 3)，(4, 3)，(4, 2)，(4, 1)，(3, 1)，(2, 1)，(1, 1)，所需步数为 9。

2.6　跳马(Knight Moves), ZOJ1091, POJ2243

题目描述：

给定象棋棋盘上两个位置 *a* 和 *b*，编写程序，计算马从位置 *a* 跳到位置 *b* 所需步数的最小值。

输入描述：

输入文件包含多个测试数据。每个测试数据占一行，为棋盘中的两个位置，用空格隔开。棋盘位置为两个字符组成的串，第 1 个字符为字母 a～h，代表棋盘中的列；第 2 个字符为数字字符 1～8，代表棋盘中的行。

输出描述：

对输入文件中的每个测试数据，输出一行"To get from xx to yy takes n knight moves."，*xx* 和 *yy* 分别为输入数据中的两个位置，*n* 为求得的最少步数。

样例输入：	**样例输出：**
e2 e4	To get from e2 to e4 takes 2 knight moves.
a1 b2	To get from a1 to b2 takes 4 knight moves.

2.7　简单的迷宫问题(Basic Wall Maze), POJ2935

题目描述：

在本题中，需要求解一个简单的迷宫问题。

(1) 迷宫由 6 行 6 列的方格组成。

(2) 3 堵长度为 1～6 的墙壁，水平或竖直地放置在迷宫中，用于分隔方格。

(3) 一个起始位置和目标位置。

图 2.23 描述了一个迷宫。需要找一条从起始位置到目标位置的最短路径。从任一个方格出发，只能移动到上、下、左、右相邻方格，并且没有被墙壁所阻挡。

输入描述：

输入文件中包含多个测试数据。每个测试数据包含 5 行：第 1 行为两个整数，表示起始位置的列号和行号；第 2 行也是两个整数，为目标位置的列号和行号，列号和行号均从 1 开始计起；第 3～5 行均为 4 个整数，描述了 3 堵墙的位置；如果墙是水平放置的，则由左、右两个端点所在的位置指定，如果墙是竖直放置的，则由上、下两个端点所在的位置指定；端点的位置由两个整数表示，第 1 个整数表示端点距离迷宫左边界的距离，第 2 个整数表示端点距离迷宫上边界的距离。

假定这 3 堵墙互相不会交叉，但两堵墙可能会相邻于某个方格的顶点。从起始位置到目标位置一定存在路径。下面的样例输入数据描述了图 2.23 所示的迷宫。

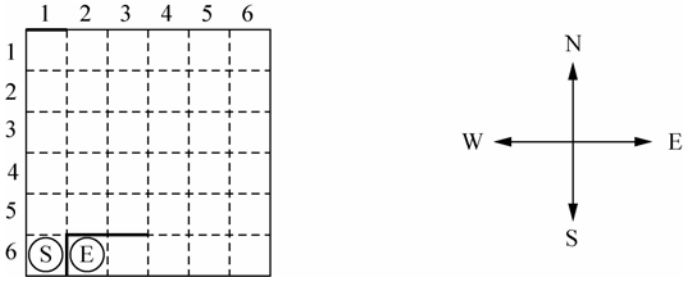


图 2.23　一个简单的迷宫问题

输入文件中最后一行为两个 0，代表输入结束。

输出描述：

对输入文件中的每个测试数据，输出从起始位置到目标位置的最短路径，最短路径由代表每一步移动的字符组成('N'表示向上移动，'E'表示向右移动，'S'表示向下移动，'W'表示向左移动)。对某个测试数据，可能存在多条最短路径，对于这种情形，只需输入任意一条最短路径即可。

样例输入：

```
1 6
2 6
0 0 1 0
1 5 1 6
1 5 3 5
0 0
```

样例输出：

```
NEEESWW
```

2.8　送情报

题目描述：

战争年代，通讯员经常要穿过敌占区去送情报。在本题中，敌占区是一个由 $M \times N$ 个方格组成的网格。通讯员要从初始方格出发，送情报到达目标方格。初始时，通讯员具有一定的体力。

网格中，每个方格可能为安全的方格、布有敌人暗哨的方格、埋有地雷的方格及被敌人封锁的方格。通讯员从某个方格出发，对上、右、下、左 4 个方向上的相邻方格：如果某相邻方格为安全的方格，通讯员能顺利到达，所需时间为 1 个单位时间、消耗的体力为 1 个单位的体力；如果某相邻方格为敌人布置的暗哨，则通讯员要消灭该暗哨才能到达该方格，所需时间为 2 个单位时间，消耗的体力为 2 个单位的体力；如果某相邻方格为埋有地雷的方格，通讯员要到达该方格，则必须清除地雷，所需时间为 3 个单位时间，消耗的体力为 1 个单位的体力。另外，从目标方格的相邻方格到达目标方格，所需时间为 1 个单位时间、消耗的体力为 1 个单位的体力。本题要求的是：通讯员能否到达指定的目的地，如果能到达，所需最少的时间是多少(只需要保证到达目标方格时，通讯员的体力>0 即可)。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个正整数：*M* 和 *N*，2<*M*、*N*<20，分别表示网格的行和列。接下来有 *M* 行，描述了网格；每行有 N 个字符，这些字符可以是'.'、'w'、'm'、'x'、'S'、'T'，分别表示安全的方格、布有敌人暗哨的方格、埋有地雷的方格、被敌人封锁的方格(通讯员无法通过)、通讯员起始方格、目标方格，输入数据保证每个测试数据中只有一个'S'和'T'。表格中各重要符号的含义及参数见表 2-2。每个测试数据的最后一行为一个整数 *P*，表示通讯员初始时的体力。*M* = *N* = 0 表示输入结束。

表 2-2　网格中各重要符号的含义及参数

符号	含义	消耗的时间	消耗的体力
'.'	安全的方格	1	1
'w'	布有敌人暗哨的方格	2	2
'm'	埋有地雷的方格	3	1

输出描述：

对输入文件中的每个测试数据，如果通讯员能在体力消耗前到达目标方格，则输出所需的最少时间；如果通讯员无法到达目标方格(即体力消耗完毕或没有从起始方格到目标方格的路径)，则输出 No。

样例输入：

```
5 6
wx.w..
Sxm.mw
xx.m..
m.w.T.
w..m.w
7
5 7
mwwxwxw
mxww...
xTx..wx
```

样例输出：

```
No
13
```

```
xm.mwww
xmmxmSw
8
0 0
```

提示：这道题跟例 2.3 有点类似，但与例 2.3 不同的是，到达某个方格不仅有时间因素，还有体力因素。本题要求的是时间最少的方案，体力因素似乎不重要。然而，如果按照例 2.3 中的方法，以“到达方格(x, y)所花费时间更少”作为“是否将这种到达(x, y)的方案入队列”的标准，所求出来的解可能是错误的。

例如，样例输入中的第 2 个测试数据所描述的地图如 2.24(a)所示，图 2.24(a)同时给出了一种时间最少的方案，按这种方案到达目标方格时所花费的时间为 13，所剩体力为 1。然而这种方案到达(3, 4)位置(图 2.24(b)中圆圈所表示的位置)所需时间为 5，另一种方案到达该位置所需时间为 4，按照例 2.3 中的方法，前一种方案可能会被舍去(而不入队列)，而按照后一种方案因为到达目标方格时体力为 0，从而得到“无法到达”的错误结论。

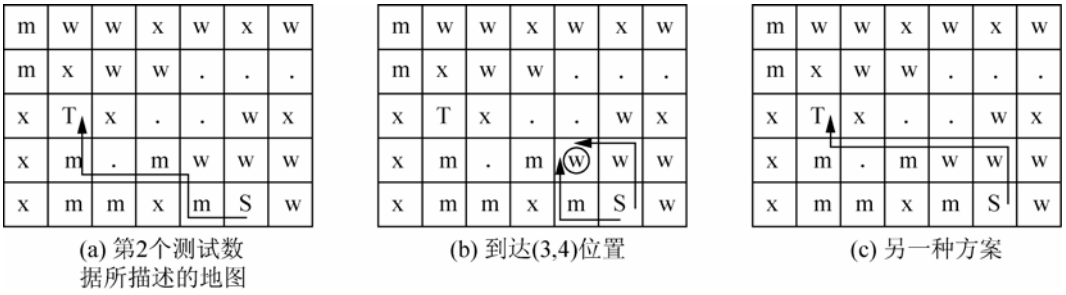


图 2.24　送情报

2.3　活动网络——AOV 网络

活动网络(Activity Network)可以用来描述生产计划、施工过程、生产流程、程序流程等工程中各子工程的安排问题。活动网络可分为两种：AOV 网络和 AOE 网络，本节介绍 AOV 网络及拓扑排序，下节将介绍 AOE 网络和关键路径。

2.3.1　AOV 网络与拓扑排序

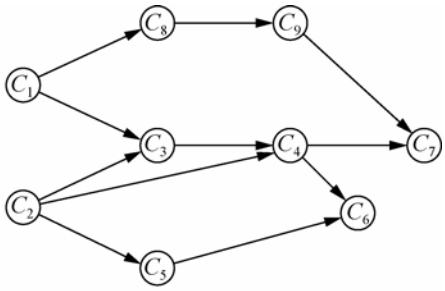
1. AOV 网络与有向无环图

一般一个工程可以分成若干个子工程，这些子工程称为**活动**(Activity)。完成了这些活动，整个工程就完成了。例如，计算机专业课程的学习就是一个工程，每门课程的学习就是整个工程中的一个活动。图 2.25(a)给出了 9 门课程，其中有 7 门课程要求先修某些课程，其他 2 门没有先修课课程。这样，这 9 门课程有些必须严格按先后顺序学习，有些课程可以并行地学习。

可以用如图 2.25(b)所示的有向图来表示这种先修关系。在这种有向图中，顶点表示课程学习活动，有向边表示课程之间的先修关系。例如，从顶点 C_1 到 C_8 有一条有向边，表示课程 C_1 必须在课程 C_8 之前先学习完。

课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计	
C_3	离散数学	C_1 C_2
C_4	数据结构	C_2 C_3
C_5	算法分析	C_2
C_6	编译技术	C_4 C_5
C_7	操作系统	C_4 C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8

(a) 课程列表



(b) 学生选课工程图

图 2.25　AOV 网络——课程安排图

实际上，可以用有向图来表示一个工程。在这种有向图中，用顶点表示活动，用有向边<*u*, *v*>表示活动 *u* 必须先于活动 *v* 进行。这种有向图叫做顶点表示活动的网络(Activity On Vertices)，记作 **AOV 网络**。

在 AOV 网络中，如果存在有向边<*u*, *v*>，则活动 *u* 必须在活动 *v* 之前进行，并称 *u* 是 *v* 的**直接前驱**(Immediate Predecessor)，*v* 是 *u* 的**直接后继**(Immediate Successor)。如果存在有向路径<*u*, u_1, u_2, …, u_n, *v*>，则称 *u* 是 *v* 的**前驱**(Predecessor)，*v* 是 *u* 的**后继**(Successor)。

这种前驱与后继的关系有**传递性**(Transitivity)。例如，如果活动 v_2 是 v_1 的后继，v_3 是 v_2 的后继，那么活动 v_3 也是 v_1 的后继。此外，任何活动不能以它自己作为自己的前驱或后继，这种特性称为**反自反性**(Irreflexivity)。

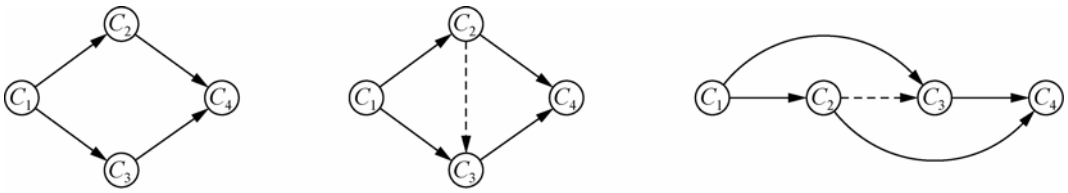
从前驱与后继的传递性和反自反性可以看出，AOV 网络中不能出现有向回路(或称为有向环)。不含有向回路的有向图称为**有向无环图**(Directed Acyclic Graph，DAG)。

在 AOV 网络中如果出现了有向回路，则意味着某项活动以自己作为先决条件，这是不对的。如果设计出这样的流程图，工程将无法进行。对于程序而言，将陷入死循环。因此，对于给定的 AOV 网络，必须先判断它是否是有向无环图。

2. 拓扑排序

判断有向无环图的方法是对 AOV 网络构造它的**拓扑有序序列**(Topological Order Sequence)，即将各个顶点排列成一个线性有序的序列，使得 AOV 网络中所有存在的前驱和后继关系都能得到满足。

例如，对图 2.26(a)所示的 AOV 网络，它的拓扑有序序列为：C_1, C_2, C_3, C_4，如图 2.26(c)所示。原来图 2.26(a)中的所有前驱和后继关系，在图 2.26(c)中都保留了；而且原来图 2.26(a)中没有前驱和后继关系的顶点(如 C_2 和 C_3)之间也人为地增加了前驱和后继关系，如图 2.26(b)中的虚线所示。



(a) AOV 网络　　(b) 人为增加的前驱/后继关系　　(c) 拓扑有序序列

图 2.26　拓扑排序与拓扑有序序列

这种构造 AOV 网络全部顶点的拓扑有序序列的运算称为**拓扑排序**(Topological Sort)。如果通过拓扑排序能将 AOV 网络的所有顶点都排入一个拓扑有序的序列中，则该 AOV 网络中必定不存在有向环；相反，如果得不到所有顶点的拓扑有序序列，则说明该 AOV 网络中存在有向环，此 AOV 网络所代表的工程是不可行的。

例如，对 2.25(b)所示的学生选课工程图进行拓扑排序，得到的拓扑有序序列为：

C_1，C_2，C_3，C_4，C_5，C_6，C_8，C_9，C_7，

或：C_1，C_8，C_9，C_2，C_5，C_3，C_4，C_7，C_6。

学生必须按照拓扑有序的顺序选修课程，才能保证学习任何一门课程时其先修课课程都已经学过。从该例子可以看出，一个 AOV 网络的拓扑有序序列可能是不唯一的。

2.3.2　拓扑排序实现方法

对一个 AOV 网络进行拓扑排序的方法如下。

(1) 从 AOV 网络中选择一个入度为 0(即没有直接前驱)的顶点并输出。

(2) 从 AOV 网络中删除该顶点及该顶点发出的所有边。

(3) 重复步骤(1)和(2)，直至找不到入度为 0 的顶点。

按照上面的方法进行拓扑排序，其结果有两种情形：第 1 种，所有的顶点都被输出，也就是整个拓扑排序完成了；第 2 种，仍有顶点没有被输出，但剩下的图中再也没有入度为 0 的顶点，这样拓扑排序不能再继续进行下去，这就说明此图是有环图。

图 2.27 给出了一个拓扑排序的例子，最后得到的拓扑有序序列为：C_5, C_1, C_4, C_3, C_2, C_6。在该图中，有阴影的顶点表示当前输出的顶点。其拓扑排序过程如下。

(1) 选择一个入度为 0 的顶点，C_5，如图 2.27(b)所示，删除 C_5 及发出的每条边。

(2) 选择一个入度为 0 的顶点，C_1，如图 2.27(c)所示，删除 C_1 及发出的每条边。

(3) 选择一个入度为 0 的顶点，C_4，如图 2.27(d)所示，删除 C_4，C_4 没有出边。

(4) 选择一个入度为 0 的顶点，C_3，如图 2.27(e)所示，删除 C_3 及发出的每条边。

(5) 选择一个入度为 0 的顶点，C_2，如图 2.27(f)所示，删除 C_2 及发出的每条边。

(6) 选择一个入度为 0 的顶点，C_6，如图 2.27(g)所示，删除 C_6，C_6 没有出边。

至此，拓扑排序执行完毕，所有顶点都排在一个线性有序的序列中。

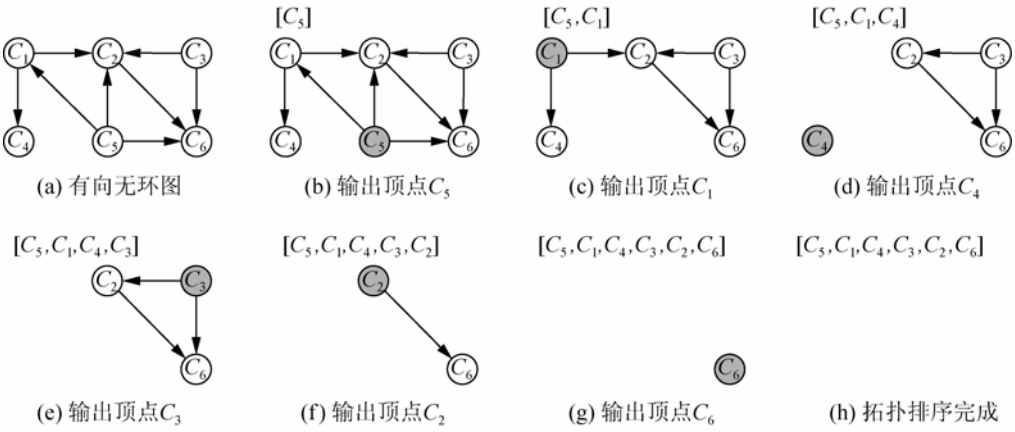


图 2.27　拓扑排序过程

拓扑排序在实现时需要建立一个 count 数组，记录各个顶点的入度。入度为 0 的顶点就是无前驱的顶点。在存储图时选择邻接表(即出边表)更为方便，因为在进行拓扑排序时要依次删除入度为 0 的顶点以及它发出的每条边。

另外，拓扑排序在实现时，还需建立一个存放入度为 0 的顶点的栈，供选择和输出无前驱的顶点。只要出现入度为 0 的顶点，就将它压入栈中。使用这种栈的拓扑排序算法可以描述如下。

(1) 建立入度为 0 的顶点栈，初始时将所有入度为 0 的顶点依次入栈。

(2) 当入度为 0 的顶点栈不为空时，重复执行。

{

从入度为 0 的顶点栈中弹出栈顶顶点，并输出该顶点；

从 AOV 网络中删除该顶点和它发出的每条边，边的终点入度减 1；

如果边的终点入度减至 0，则将该顶点推进入度为 0 的顶点栈；

}

(3) 如果输出顶点个数少于 AOV 网络中的顶点个数，则报告网络中存在有向环。

思考：是否可以采用队列来存储入度为 0 的顶点？请参考 2.3.3 节中的第 3 点。并编写程序验证。

在算法实现时，为了建立入度为 0 的顶点栈，可以不另外分配存储空间，直接利用存储顶点入度的 count[ ]数组。为此，设立一个栈顶指针 top，指示当前栈顶的位置，即最后压进栈的顶点的位置。栈初始时，top = −1。

接下来以图 2.27(a)所示的有向无环图为例详细分析拓扑排序的实现，该有向无环图的邻接表存储表示如图 2.28(a)所示，图 2.28(e)描绘了拓扑排序过程 count 数组和栈顶指针 top 的变化。

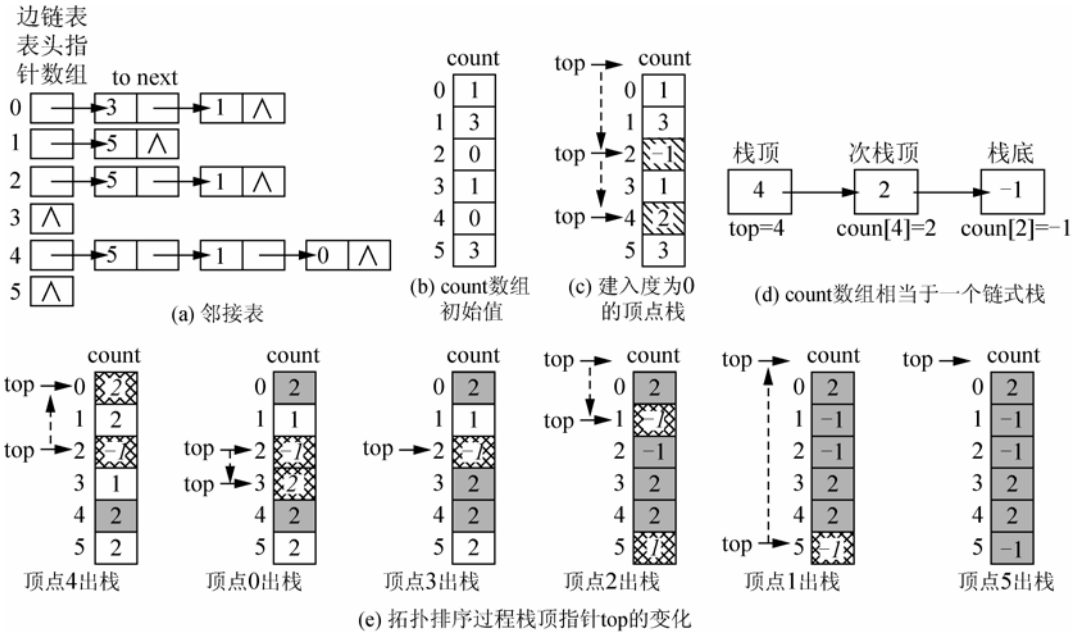


图 2.28　拓扑排序实现

顶点 i 入栈时，将执行以下栈顶指针的修改。

```
count[i]=top;
top=i;       //top 指向新栈顶，原栈顶元素存放在 count[i]中
```

例如，在拓扑排序进行前，将入度为 0 的顶点全部压入栈，建栈完毕后，count 数组存储情形如图 2.28(c)所示，这相当于建立了如图 2.28(d)所示的链式栈。

弹出栈顶顶点时，作如下修改。

```
j=top;
top=count[top];      //位于栈顶的顶点的位置记为 j，top 退至次栈顶
```

由于无前驱的顶点都压入了入度为 0 的顶点栈，每一步选取位于栈顶的顶点，所以栈顶指针 top 的变化与拓扑排序时顶点的输出次序是一致的。有一种例外情况：如果在顶点全部输出完之前，栈顶指针 top 已经变为−1，表明栈已经为空栈，也就是说，已经没有无前驱的顶点可以取了，说明 AOV 网络中存在有向环，拓扑排序应该终止。

拓扑排序算法的程序实现，详见例 2.6。

例 2.6　对输入的有向图进行拓扑排序，并输出一个拓扑有序序列；如果存在有向环，则给出提示信息。

假设输入文件中有向图的格式为：首先是顶点个数 *n* 和边数 *m*；然后是每条边，每条边的数据占一行，格式为 *u v*，表示从顶点 *u* 到顶点 *v* 的一条有向边，顶点序号从 1 开始计起。输入文件最后一行为 0 0，表示输入数据结束。

样例输入：

```
6 8
1 2
1 4
2 6
3 2
3 6
5 1
5 2
5 6
6 8
1 3
1 2
2 5
3 4
4 2
4 6
5 4
5 6
0 0
```

样例输出：

```
5 1 4 3 2 6
Network has a cycle!
```

分析：

在下面的代码中，各顶点边链表的表头指针存放在 List 数组中，如图 2.28(a)所示。在构造每个顶点的边链表时，可以统计每个顶点的入度，并存放在 count 数组中。TopSort 函数实现了拓扑排序过程：首先扫描 count 数组，将入度为 0 的顶点入栈；然后从栈中依次

弹出栈顶顶点输出，并扫描该顶点的边链表，把每个边结点的终点的入度减 1，如果减至 0，则将该终点入栈；当 n 个顶点都出栈后，拓扑有序序列也输出完毕；或者在此之前栈为空，则可以判断有向图中存在有向环。

样例输入中第 1 个测试数据所描述的有向图如图 2.27(a)所示；第 2 个测试数据所描述的有向图如图 2.29 所示，该有向图中存在有向回路。

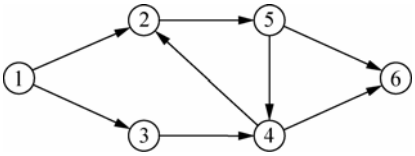


图 2.29　存在有向回路的 AOV 网络

代码如下：

```
#define MAXN 10          //顶点个数的最大值
struct ArcNode
{
    int to;
    struct ArcNode *next;
};
int n, m;                //顶点个数、边数
ArcNode* List[MAXN];     //每个顶点的边链表表头指针
int count[MAXN];         //各顶点的入度
char output[100];        //输出内容
void TopSort( )
{
    int i, top=-1;
    ArcNode* temp;
    bool bcycle=false;       //是否存在有向环的标志
    int pos=0;               //写入 output 数组的位置
    for( i=0; i<n; i++ )     //入度为 0 的顶点入栈
    {
        if( count[i]==0 )
        {
            count[i]=top;  top=i;
        }
    }
    for( i=0; i<n; i++ )
    {
        if( top==-1 )        //栈为空，存在有向回路
        {
            bcycle=true;  break;
        }
        else
        {
            int j=top; top=count[top]; //栈顶顶点 j 出栈
            pos+=sprintf( output+pos, "%d ", j+1 );
            temp=List[j];
            //遍历顶点 j 的边链表，每条出边的终点的入度减 1
            while( temp!=NULL )
            {
```

```
                int k=temp->to;
                if( --count[k]==0 )        //终点入度减至0，则入栈
                {
                    count[k]=top; top=k;
                }
                temp=temp->next;
            }//end of while
        }//end of else
    }//end of for
    if( bcycle )  printf( "Network has a cycle!\n" );
    else
    {
        int len=strlen( output );  output[len-1]=0;  //去掉最后的空格
        printf( "%s\n", output );
    }
}
int main( )
{
    int i, u, v;    //循环变量、边的起点和终点
    while( 1 )
    {
        scanf( "%d%d", &n, &m );   //读入顶点个数n，边数
        if( n==0 && m==0 )  break;
        memset( List, 0, sizeof(List) );
        memset( count, 0, sizeof(count) );
        memset( output, 0, sizeof(output) );
        ArcNode* temp;
        for( i=0; i<m; i++ )   //构造边链表
        {
            scanf( "%d%d", &u, &v );   //读入边的起点和终点
            u--;  v--;
            count[v]++;
            temp=new ArcNode;
            temp->to=v;  temp->next=NULL;   //构造邻接表
            if( List[u]==NULL ) //此时边链表中没有边结点
                List[u]=temp;
            else    //边链表中已经有边结点，插入temp
            {
                temp->next=List[u];  List[u]=temp;
            }
        }
        TopSort( );
        for( i=0; i<n; i++ )   //释放边链表上各边结点所占用的存储空间
        {
            temp=List[i];
            while( temp!=NULL )
            {
                List[i]=temp->next;  delete temp;  temp=List[i];
            }
        }
    }
    return 0;
}
```

思考：请用 STL 中的栈 stack 来改写例 2.6 的程序。

2.3.3 关于拓扑排序的进一步说明

1. 拓扑排序复杂度分析

例 2.6 的程序在实现拓扑排序过程中，搜索入度为 0 的顶点，建立链式栈所需时间为 $O(n)$。当有向图中不存在有向环时，每个顶点要进一次栈，出一次栈，每条边扫描一次且仅一次，其复杂度为 $O(m)$。所以，总的时间复杂度为 $O(n+m)$。

2. 拓扑排序与 BFS 算法的对比分析

与 2.2.2 节介绍的用邻接表实现 BFS 算法的伪代码相比，拓扑排序与其有相似之处也有不同之处。

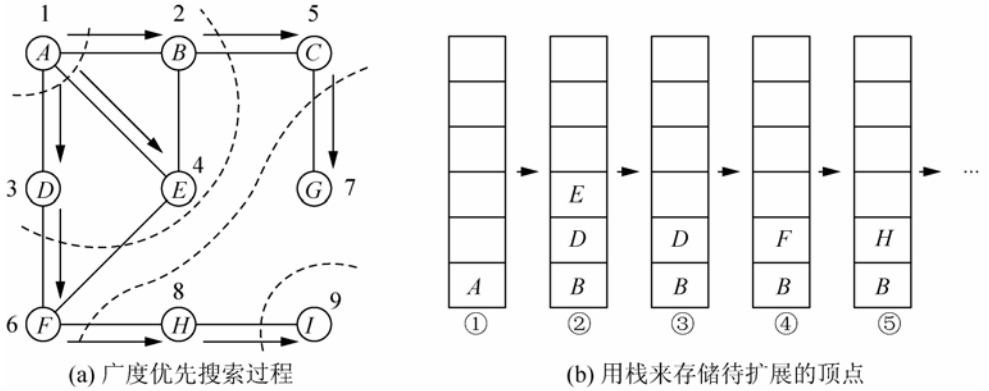
相同点：拓扑排序实质上就是一种广度优先搜索，在算法执行过程中，通过栈顶顶点访问它的每个邻接点，整个算法执行过程中，每个顶点访问一次且仅一次，每条边扫描一次且仅一次。

不同点：BFS 算法在扫描每条边时，如果边的终点没有访问过，则入队列；而拓扑排序算法在扫描每条边时，终点的入度要减 1，当减至 0 时才将该终点入栈。

3. 拓扑排序与 BFS 算法中的栈或队列的使用

请注意，在 2.2 节的 BFS 算法中是用队列来存储待扩展的顶点，在 2.3.2 节的拓扑排序算法中是用栈来存储入度为 0 的顶点。那么，在 BFS 算法中是否可以用栈来存储待扩展的顶点？在拓扑排序算法中是否可以用队列来存储入度为 0 的顶点？队列和栈的区别在于顶点出队列(或栈)的顺序，队列是先进先出，栈是后进先出。所以，能否用队列(或栈)关键要看这种顺序是否会影响算法的正确性。

BFS 算法：如果用栈存储待扩展的顶点，如图 2.30 所示，其中图 2.30(a)为正确的搜索过程，图 2.30(b)为用栈存储待扩展顶点时各顶点入栈和出栈的过程。在图 2.30(b)中，依次出栈的顶点是 $A \rightarrow E \rightarrow D \rightarrow F \rightarrow H \rightarrow I \rightarrow B \rightarrow C \rightarrow G$，很明显，这与 BFS 算法的实现过程和顶点访问顺序大相径庭。因此，在 BFS 算法中不能用栈来存储待扩展的顶点。



(a) 广度优先搜索过程　　(b) 用栈来存储待扩展的顶点

图 2.30　用栈“实现”BFS 算法

拓扑排序算法：在算法执行过程中，如果同时存在多个入度为 0 的顶点，则首先选择删除不会影响算法的正确性的顶点。所以，在拓扑排序算法中，可以使用队列或栈来存储入度为 0 的顶点。

2.3.4　例题解析

以下通过两道例题的分析，再详细介绍拓扑排序的思想及实现方法。

例 2.7　将所有元素排序(Sorting It All Out)

题目来源：

East Central North America 2001, ZOJ1060, POJ1094

题目描述：

由一些不同元素组成的升序序列是可以用若干个小于号将所有的元素按从最小到最大的顺序排列起来的序列。例如，排序后的序列为 *A, B, C, D*，这意味着 *A*<*B*、*B*<*C* 和 *C*<*D*。在本题中，给定一组形如 *A* < *B* 的关系式，试判定是否存在一个有序序列。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个正整数 *n* 和 *m*；*n* 表示要排序的元素个数，2≤*n*≤26，这 *n* 个元素为字母表的前 *n* 个大写字母；*m* 表示有 *m* 个形如 *A* < *B* 的关系式。接下来有 *m* 行，每行描述了一个关系式，包含 3 个字符：一个大写字母字符，字符"<"，另一个大写字母字符；这些字母字符均不会超过字母表前 *n* 个字母的范围。*n* = *m* = 0 表示输入结束。

输出描述：

对每个测试数据，输出一行，内容为以下 3 行之一。

Sorted sequence determined after xxx relations: yyy…y.

Sorted sequence cannot be determined.

Inconsistency found after xxx relations.

其中"*xxx*"为判定出有序序列存在或存在矛盾时已经处理的关系式数目，哪一种情形最先出现，则按哪一种情形处理，"*yyy…y*"为排序后的升序序列。

样例输入：	**样例输出：**
4 6	Sorted sequence determined after 4 relations: ABCD.
A<B	Inconsistency found after 2 relations.
A<C	Sorted sequence cannot be determined.
B<C	
C<D	
B<D	
A<B	
3 2	
A<B	
B<A	
26 1	
A<Z	
0 0	

分析：

这道题很明显需要采用拓扑排序求解。建图：假设有关系式 *A*<*B*，则在图中画一条有向边<*A*, *B*>。例如，对本题样例输入的第 1 个测试数据，建图后得到的有向图如图 2.31 所示。

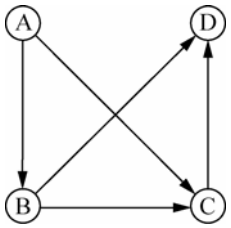


图 2.31　将所有元素排序：第 1 个测试数据的建图

这样要判断所有的关系是否能够给出最后的元素序列，就只需要对该图进行拓扑排序。在本题中，拓扑排序的结果有 3 种情形。

(1) 如果该图存在环，那么给定的关系肯定是互相矛盾的。

(2) 如果不存在环，但是拓扑排序结束后(所有的边都已经扫描完毕)，排序得到的序列中元素的个数小于给定的元素个数，那么给定的关系不足以判断出全部元素的大小关系。

(3) 如果拓扑排序出来的序列中元素的个数等于给定的元素个数，那么给出的关系可以判断出全部元素的大小关系。

代码如下：

```
using namespace std;
int count[26];  //入边表，记录顶点的入度
int temp[26];   //temp 为 count 的复制版，用于拓扑排序时进行修改
char relation[3], seq[26]; //relation 用于读入对象间关系，seq 用于存储得到的序列
bool alpha[26]; //alpha 用于记录对象是否已经检查
int n, m;
vector<vector<char>>v;  //使用 STL 的成员 vector 来存储结点
//拓扑排序，返回拓扑排序后得到的序列中元素的个数；若发现矛盾则返回-1，无法判断则返回 0
int toposort( int s )
{
    int i, j, r, cnt;   //cnt 表示入度为 0 的顶点的个数；r 表示得到序列中元素的个数
    bool flag;          //flag 为标志变量，表示拓扑排序结束后是否可以得到序列
    r=0, cnt=0;
    for( i=0; i<n; i++ )  temp[i]=count[i];
    flag=1;
    while( s-- )
    {
        cnt=0;
        for( i=0; i<n; i++ )
        {
            if( temp[i]==0 )
            {
                j=i;  cnt++;
            }
```

```
            }
            if( cnt>=1 )
            {
                //cnt=1 表示有且仅有一个入度为 0 的顶点，则该顶点必然处于序列的最前端
                if( cnt>1 )  flag=0;
                for( i=0; i<v[j].size( ); i++ )
                    temp[ v[j][i] ]--; //处理完毕后删除该顶点及其出边(即相应顶点入边)
                seq[r++]=j+'A';
                temp[j]=-1;  seq[r]=0;
            }
            else if( cnt==0 ) //cnt==0 表示没有入度为 0 的顶点，则必然存在环
                return -1;
        }
        if( flag )  return r;
        else  return 0;
    }
    int main( )
    {
        int i, j, t, k, c;  //k 用于记录已处理的关系个数；c 用于记录读入的对象个数
        int determined; //标志变量,-1 表示关系矛盾,0 表示无法得到序列,1 表示可以得到序列
        while( scanf( "%d %d", &n, &m ) !=EOF && n !=0 && m !=0 )
        {
            memset( count, 0, sizeof( count ));  //初始化
            memset( alpha, false, sizeof( alpha ));
            v.clear( );  v.resize( n );  //删除 vector 中的元素，并重新调整大小
            c=0;  determined=0;
            for( i=0; i<m; i++ )
            {
                scanf( "%s", relation );           //读入数据
                count [ relation[2]-'A' ]++;
                v[ relation[0]-'A' ].push_back( relation[2]-'A' );
                if( !alpha[ relation[0]-'A' ] )    //记录读入的对象个数
                {
                    c++;  alpha[ relation[0]-'A' ]=true;
                }
                if( !alpha[ relation[2]-'A' ] )
                {
                    c++;  alpha[ relation[2]-'A' ]=true;
                }
                if( determined==0 )
                {
                    t=toposort( c );
                    if( t==-1 )
                    {
                        determined=-1;  k=i + 1;
                    }
                    else if( t==n )
```

```
                {
                    determined=1;  k=i + 1;
                }
            }
        }
        if( determined==-1 )
            printf( "Inconsistency found after %d relations.\n",k );
        else if( determined==0 )
            printf( "Sorted sequence cannot be determined.\n" );
        else
            printf( "Sorted sequence determined after %d relations: %s.\
                    n",k,seq );
    }
    return 0;
}
```

例 2.8　窗口绘制(Window Pains)

题目来源：

South Central USA 2003, ZOJ2193, POJ2585

题目描述：

Boudreaux 喜欢多任务的系统，特别是当他用计算机时。他从不满足于每次只运行一个程序，通常他总是同时运行 9 个程序，每个程序有一个窗口。由于显示器屏幕大小有限，他把窗口重叠，并且当他想用某个窗口时，就把它调到最前面。如果他的显示器是一个 4×4 的网格，则 Boudreaux 的每一个程序窗口就应该像图 2.32 所示那样用 2×2 大小的窗口表示。

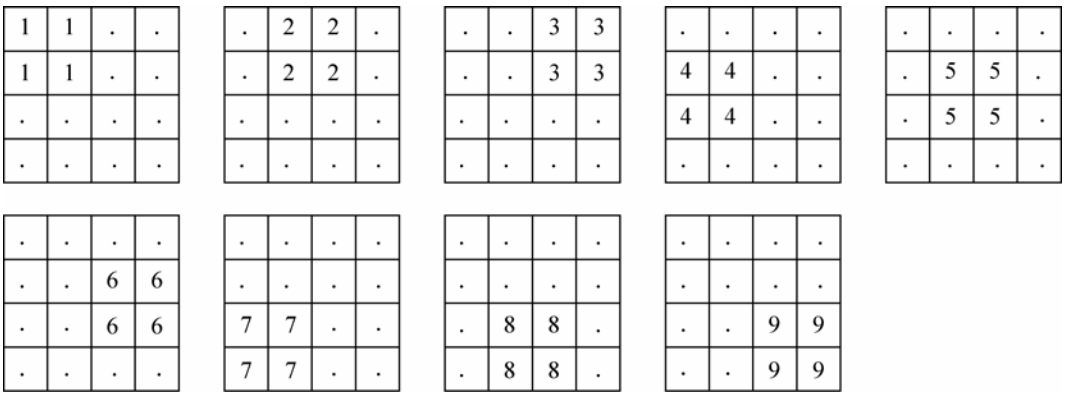


图 2.32　窗口绘制：窗口表示

当 Boudreaux 把一个窗口调到最前面时，它的所有方格都位于最前面，覆盖它与其他窗口共用的方格。例如，如果先是窗口 1 位于最前面，然后是窗口 2，那么结果如图 2.33(a)所示。如果接下来窗口 4 位于最前面，则结果如图 2.33(b)所示等。

不幸的是，Boudreaux 的计算机很不稳定，经常崩溃。他通过观察这些窗口，如果发现每个窗口都应被正确地调到最前面时，窗口显示的不应该是现在这种图形，那么就能判断出他的计算机是死机了。

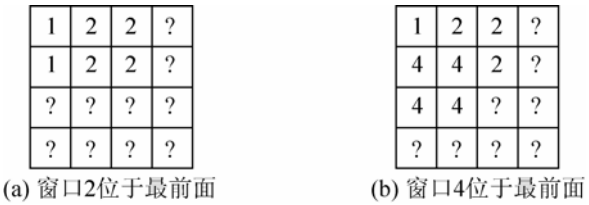


图 2.33　窗口绘制：窗口表示例子

输入描述：

输入文件包含最多 100 组数据。每组数据将按如下格式给出，各组数据间无空行。

每组数据包含以下 3 部分。

(1) 起始行——为字符串"START"。

(2) 显示器屏幕快照——用 4 行表示当前 Boudreaux 的显示器状态。这 4 行为 4×4 的矩阵，每一个元素代表显示器对应方格中所显示的一小块窗口。为使输入简单，数字间仅用一个空格分开。

(3) 结束行——为字符串"END"。

最后一组数据后，会有一行字符串，为"ENDOFINPUT"。

注意：每个小块只能出现在它可能出现的地方。例如 1 只能出现在左上方 4 个方格里。

输出描述：

对每个数据只输出一行：如果能按一定顺序依次将每个窗口调到最前面时能达到数据描述的那样(即没死机)，输出"THESE WINDOWS ARE CLEAN"，否则输出"THESE WINDOWS ARE BROKEN"。

样例输入：

```
START
1 2 3 3
4 5 6 6
7 8 9 9
7 8 9 9
END
START
1 1 3 3
4 1 3 3
7 7 9 9
7 7 9 9
END
ENDOFINPUT
```

样例输出：

```
THESE WINDOWS ARE CLEAN
THESE WINDOWS ARE BROKEN
```

分析：

本题样例输入中两个测试数据所描绘的窗口表示如图 2.34 所示。

可以想象，假如有一个正常的屏幕给用户，它可能会是如图 2.34(a)所示的这种显示。这种窗口显示 9 个窗口的顺序是：窗口 1 在最下面；然后窗口 2 覆盖上去，接下来窗口 3～9 依次覆盖上去。

1	2	3	3
4	5	6	6
7	8	9	9
7	8	9	9

(a) 测试数据1

1	1	3	3
4	1	3	3
7	7	9	9
7	7	9	9

(b) 测试数据2

图 2.34　窗口绘制：测试数据

表 2-3 描述了屏幕中每个方格能被哪些窗口覆盖住，其中一个方格最多只能被 4 个窗口覆盖住。例如，(2, 2)这个方格可以被 1、2、4、5 这 4 个窗口覆盖住。只要当前一个方格是被其中一个窗口覆盖，则这个窗口肯定盖住其他可以覆盖这个方格的窗口。例如，在图 2.34(a)中，(2, 2)位置被 5 号窗口覆盖住了，则可以知道 5 号窗口必定覆盖了 1、2、4 这 3 个窗口。

表 2-3　窗口绘制：窗口覆盖住屏幕中的方格

1	1, 2	2, 3	3
1, 4	1, 2, 4, 5	2, 3, 5, 6	3, 6
4, 7	4, 5, 7, 8	5, 6, 8, 9	6, 9
7	7, 8	8, 9	9

这样，就可以根据这个覆盖关系构成一个有向图，以 9 个窗口为顶点，如果 *X* 号窗口可以覆盖住 *Y* 窗口，则有一条有向边<*X*, *Y*>。例如，在第 1 个测试数据所构造的有向图存在有向边<5, 1>、<5, 2>和<5, 4>。根据这种思路对样例输入中的第 2 个测试数据进行构图，如图 2.35 所示。

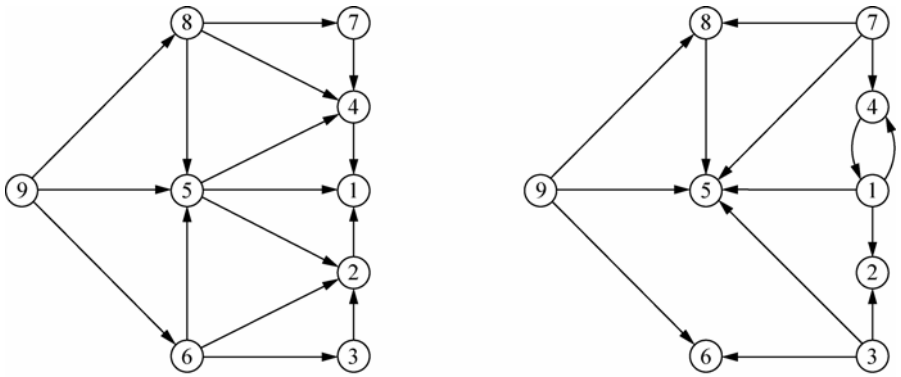


图 2.35　窗口绘制：构造有向图

有向图构造好以后，如果是一个正常的屏幕，那么通过这个屏幕构建出来的图肯定是一个有向无环图(这是因为，不可能出现 *A* 窗口覆盖了 *B* 窗口，而 *B* 窗口又反过来覆盖了 *A*)，所以第 2 个测试数据为不正常的屏幕。因此该题转换成拓扑排序问题。本题的求解过程为如下。

(1) 通过给出的屏幕构建有向网络。

(2) 通过拓扑排序算法判断该网络是否有环，如果存在环，则该屏幕为死机后屏幕，否则为正常屏幕。

代码如下：

```
using namespace std;
const int n=4;
int screen[4][4];   //屏幕快照最后显示的内容
string cover[4][4];//表示能覆盖(i, j)位置的窗口的集合
bool exist[10];     //某个窗口是否在屏幕快照上出现
int id[10];         //入度
bool g[10][10];     //邻接表
int t;          //记录屏幕上总共出现的不同的窗口种类，以这些窗口为顶点，构建有向图
string s;           //读入字符数据的临时变量
void calc( )        //统计覆盖(i, j)位置的窗口的集合
{
    int k, i, j;
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )  cover[i][j].erase( );
    }
    for( k=1; k<=9; k++ )
    {
        i=(k-1)/3;
        j=(k-1)%3;
        cover[i][j]+=char(k+'0');       //第 k 个窗口左上角位置
        cover[i][j+1]+=char(k+'0');     //第 k 个窗口右上角位置
        cover[i+1][j]+=char(k+'0');     //第 k 个窗口左下角位置
        cover[i+1][j+1]+=char(k+'0');   //第 k 个窗口右下角位置
    }
}
void init( )    //读入屏幕快照数据
{
    int i, j, k;
    memset( exist, 0, sizeof(exist) );
    memset( id, 0, sizeof(id) );  memset( g, 0, sizeof(g) );
    t=0;    //t 为屏幕快照中出现的窗口种类
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )
        {
            cin >>k;
            screen[i][j]=k;
            if( !exist[k] )  t++;
            exist[k]=true;
        }
    }
}
```

```
void build( )  //构建有向图
{
    int i, j, p;
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )
        {
            //screen[i][j]能覆盖住 cover[i][j]中除它本身外的每个窗口
            for( p=0; p<cover[i][j].length(); p++ )
            {
                if( (!g[screen[i][j]][cover[i][j][p]-'0']) && (screen[i]
                    [j]!=cover[i][j][p]-'0') )
                {
                    g[screen[i][j]][cover[i][j][p]-'0']=true;
                    id[cover[i][j][p]-'0']++;  //入度加 1
                }
            }
        }
    }
}
bool check( )  //判断有向图是否存在有向环
{
    int i, k, j;
    for( k=0; k<t; k++ )
    {
        i=1;
        //统计出现在屏幕快照中、并且入度不为 0 的窗口个数
        while( !exist[i] || (i<=9&&id[i]>0) )
            i++;
        if( i>9 )   //i>9 说明所有窗口入度均不为 0，则必然存在环
            return false;
        //处理编号为 i 的窗口，删除该窗口以及其相应出边
        exist[i]=false;
        for( j=1; j<=9; j++ )
        {   //删除相应顶点入边(入度减 1)
            if(exist[j]&&g[i][j])  id[j]--;
        }
    }
    return true;
}
int main( )
{
    calc( );
    while( cin>>s )
    {
        if( s=="ENDOFINPUT" )  break;
        init( );    //读入数据
        build( );   //构造有向图
        if( check( ) )  cout<<"THESE WINDOWS ARE CLEAN\n";
        else  cout<<"THESE WINDOWS ARE BROKEN\n";
        cin>>s;
```

```
        }
        return 0;
    }
```

练　　习

2.9　叠图片(Frame Stacking), ZOJ1083, POJ1128

题目描述：

考虑图 2.36(a)所示的 5 张放置在 9×8 大小阵列中的图片。

现在将这 5 张图片一张放一张地叠在一起，其中第 1 张放在最底下，第 5 张放在最上面，如果某张图片的某一部分覆盖了其他一张图片，则它将遮住底下这张图片。

这 5 张图片叠起来后的效果如图 2.36(b)所示，你知道它们是按照什么顺序叠起来的吗？答案是：*EDABC*。

图 2.36　叠图片

在本题中，给定叠起来的效果图，判断这些图片是按照怎样的顺序叠起来的(从最底下到最上面)，规则如下。

(1) 图片宽度均为 1 个字符，四边均不短于 3 个字符。

(2) 每张图片的 4 条边当中每条边都能看见一部分。

(3) 用每张图片中的字母代表对应的图片，任何两张图片都不会出现相同的字母。

输入描述：

输入文件中包含多个测试数据。每个测试数据的第 1 行为整数 h，$h \leqslant 30$，表示图片的高度；第 2 行为整数 w，$w \leqslant 30$，表示图片的宽度；接下来是 h 行，每行有 w 个字符，描述了叠起来后的效果。输入数据一直到文件尾。

输出描述：

对输入文件中的每个测试数据，按从底到顶的顺序输出这些图片的叠加顺序(用图片中的字母代表图片)。如果有多个解，则按字典序输出每个解，每个解占一行。对每个测试数据，至少有一个解。

样例输入：

```
9
8
.CCC....
```

样例输出：

```
EDABC
```

```
ECBCBB..
DCBCDB..
DCCC.B..
D.B.ABAA
D.BBBB.A
DDDDAD.A
E...AAAA
EEEEEE..
```

2.10　列出有序序列(Following Orders), POJ1270

题目描述：

给定变量之间形如 $x<y$ 的约束关系列表，你的任务是输出满足约束关系的所有变量序列。例如，给定约束关系 $x<y$ 和 $x<z$，则存在两个满足约束关系的变量序列：$x\ y\ z$ 和 $x\ z\ y$。

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个约束关系列表。每个约束关系列表占两行：第 1 行为变量列表，第 2 行也是变量列表，这两个变量列表中对应变量构成了一个约束，$x\ y$ 表示 $x < y$。

所有的变量均为小写字母字符，至少有 2 个变量，至多有 20 个。至少有一个约束，至多有 50 个约束。在一个约束列表中至少有一个有序序列，至多有 300 个有序序列。

测试数据一直到文件尾。

输出描述：

对每一个约束列表，按字典序输出每个有序序列，每个有序序列占一行。测试数据的输出之间用一个空行隔开。

样例输入：

```
v w x y z
v y x v z v w v
```

样例输出：

```
wxzvy
wzxvy
xwzvy
xzwvy
zwxvy
zxwvy
```

2.11　给球编号(Labeling Balls), POJ3687

题目描述：

Windy 有 N 个不同重量的球，重量分别为 1～N，现在他想给这些球按照下面的方法编号 1～N。

(1) 任何两个球的编号都不一样。

(2) 编号满足一些约束，类似于“编号为 a 的球比编号为 b 的球轻”。

你能帮助他找到一个满足条件的编号方案吗？

输入描述：

输入文件的第 1 行为一个整数 T，表示测试数据的个数。每个测试数据的第 1 行为两个整数 N(1≤N≤200)和 M(0≤M≤40 000)；接下来有 M 行，每行为两个整数 a 和 b，表示

编号为 a 的球必须轻于编号为 b 的球，$1 \leqslant a, b \leqslant N$，测试数据之间用空行隔开。

输出描述：

对每个测试数据，输出一行，为编号 $1 \sim N$ 的球的重量。如果存在多个解，则输出这样一个方案：编号为 1 的球重量最轻的方案，如果还是有多个方案，则输出编号为 2 的球重量最轻的方案，依此类推。如果没有解，则输出-1。

样例输入：

```
2

4 0

4 1
1 1
```

样例输出：

```
1 2 3 4
-1
```

2.4　活动网络——AOE 网络

2.4.1　AOE 网络与关键路径

与 AOV 网络密切相关的另一种网络是 AOE 网络。如果在有向无环图中用有向边表示一个工程中的各项活动(Activity)，用有向边上的权值表示活动的**持续时间**(Duration)，用顶点表示**事件**(Event)，则这种有向图叫做用边表示活动的网络(Activity On Edges)，简称 **AOE 网络**。

例如，图 2.37 所示是一个有 11 个活动的 AOE 网络。其中有 9 个事件 $E_0 \sim E_8$。事件 E_0 发生表示整个工程的开始，事件 E_8 发生表示整个工程的结束。其他每个事件 E_i 发生表示在它之前的活动都已完成，在它之后的活动可以开始。例如，事件 E_4 发生表示活动 a_4 和 a_5 已经完成，活动 a_7 和 a_8 可以开始。在图 2.37 中，每条边上的数字表示每个活动的持续时间，通常，这些时间只是估计值。在工程开始之后，活动 a_1、a_2 和 a_3 可以并行进行，在事件 E_4 发生之后，活动 a_7 和 a_8 也可以并行进行。

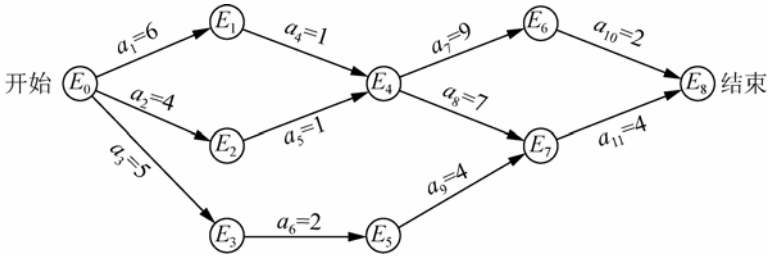


图 2.37　一个 AOE 网络

由于整个工程只有一个开始点和一个完成点，所以称**开始点**(即入度为 0 的顶点)为**源点**(Source)，称**结束点**(即出度为 0 的顶点)为**汇点**(Sink)。

AOE 网络在某些方面(如工程估算)非常有用，例如，从 AOE 网络可以了解到以下两点。

(1) 完成整个工程至少需要多长时间(假定网络中没有环)?

(2) 为缩短完成工程所需的时间，应加快哪些活动？

在 AOE 网络中，有些活动可以并行进行。从源点到各个顶点，以及从源点到汇点的有向路径可能不止一条，这些路径的长度可能也不同。完成不同路径上每个活动所需的总时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，所需时间为这条路径上所有活动的持续时间之和。这条路径长度最长的路径就称为**关键路径**(Critical Path)。

在图 2.37 所示的例子中，关键路径是 a_1, a_4, a_7, a_{10} 或 a_1, a_4, a_8, a_{11}。这条路径的所有活动的持续时间之和是 18，也就是说，完成整个工程所需时间是 18。

2.4.2 关键路径求解方法

要找出关键路径，必须找出关键活动。所谓**关键活动**(Critical Activity)，就是不按期完成就会影响整个工程完成的活动。关键路径上所有活动都是关键活动，因此，只要找到关键活动，就可以找到关键路径。

下面先定义几个与计算关键活动有关的量。

(1) **事件 E_i 的最早可能开始时间**，记为 ***Ee*[*i*]**。$Ee[i]$是从源点 E_0 到顶点 E_i 的最长路径长度。例如，在图 2.37 中，只有当 a_1, a_2, a_4, a_5 这些活动都完成了，事件 E_4 才能开始。虽然 a_2, a_5 这条路径的完成只需时间 5，但 a_1, a_4 这条路径还未完成，事件 E_4 还不能开始。只有当 a_1, a_4 也完成了，事件 E_4 才能开始。所以事件 E_4 的最早可能开始时间是 $Ee[4]= 7$。

(2) **事件 E_i 的最迟允许开始时间**，记为 ***El*[*i*]**。$El[i]$是在保证汇点 E_{n-1} 在 $Ee[n-1]$时刻完成的前提下，事件 E_i 的允许最迟开始时间，它等于 $Ee[n-1]$减去从 E_i 到 E_{n-1} 的最长路径长度。

(3) **活动 a_k 的最早可能开始时间**，记为 ***e*[*k*]**。设活动 a_k 在有向边$<E_i, E_j>$上，则 $e[k]$是从源点 E_0 到顶点 E_i 的最长路径长度。因此，$e[k] = Ee[i]$。

(4) **活动 a_k 的最迟允许开始时间**，记为 ***l*[*k*]**。设活动 a_k 在有向边$<E_i, E_j>$上，则 $l[k]$是在不会引起时间延误的前提下，该活动允许的最迟开始时间。$l[k] = El[j] - dur(<E_i, E_j>)$，其中 $dur(<E_i, E_j>)$是完成活动 a_k 所需的时间，即有向边$<E_i, E_j>$的权值。

$l[k] - e[k]$表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量，也叫做**松弛时间**(Slack Time)。$l[k] == e[k]$表示活动 a_k 没有时间余量，是关键活动。

在图 2.37 所示的例子中，活动 a_8 的最早可能开始时间是：$e[8] = Ee[4] = 7$，最迟允许开始时间是：$l[8] = El[7] - dur(<E_4, E_7>) = 14 - 7 = 7$，所以 a_8 是关键路径上的关键活动。而对活动 a_9，它的最早可能开始时间是：$e[9]= Ee[5]= 7$，最迟允许开始时间是：$l[9] = El[7] - dur(<E_5, E_7>) = 14 - 4 = 10$，它的时间余量是 $l[9] - e[9] = 3$，它推迟 3 天开始或延迟 3 天完成都不会影响整个工程的完成，所以 a_9 不是关键活动。

因此，分析关键路径的目的，是要从源点 E_0 开始估算每个活动，辨明哪些是影响整个工程进度的关键活动，以便科学地安排工作。

为了找出关键活动，就需要求得各个活动的 $e[k]$与 $l[k]$，以判别二者是否相等；而为了求得 $e[k]$与 $l[k]$，就要先求得从源点 E_0 到各个顶点 E_i 的最早可能开始时间 $Ee[i]$和最迟允许开始时间 $El[i]$。

下面分别介绍求 $Ee[i]$、$El[i]$、$e[k]$和 $l[k]$的递推公式。

(1) 求 $Ee[i]$的递推公式。从 $Ee[0] = 0$ 开始，向前递推：

$$Ee[i] = \max_j\{ Ee[j] + dur(< E_j, E_i >) \}, < E_j, E_i > \in S_2, \quad i = 1, 2, \cdots, n-1 \qquad (2\text{-}1)$$

式中：S_2 是所有指向顶点 E_i 的有向边$<E_j, E_i>$的集合。

(2) 求 $El[i]$的递推公式。从 $El[n\text{-}1] = Ee[n\text{-}1]$开始，反向递推：

$$El[i] = \min_j\{ El[j] - dur(< E_i, E_j >) \}, \ < E_i, E_j > \in S_1, \quad i = n-2, n-3, \cdots, 0 \qquad (2\text{-}2)$$

式中：S_1 是所有从顶点 E_i 发出的有向边$<E_i, E_j>$的集合。

这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。所谓逆拓扑有序，就是首先输出出度为零的顶点，以相反的次序输出拓扑排序序列，这种排序称为**逆拓扑排序**(Reverse Topological Sort)。

也就是说，在计算 $Ee[i]$时，E_i 的所有前驱顶点 E_j 的 $Ee[j]$都已求出。反之，在计算 $El[i]$时，也必须在 E_i 的所有后继顶点 E_j 的 $El[j]$都已求出的条件下才能进行计算。所以，可以以拓扑排序的算法为基础，在把各个顶点排出拓扑有序序列的同时，计算 $Ee[i]$；再以逆拓扑有序的顺序计算 $El[i]$。

(3) 求 $e[k]$和 $l[k]$的递推公式。设活动 a_k 对应的带权有向边为$<E_i, E_j>$，它的持续时间是 $dur(<E_i, E_j>)$，则有 $e[k] = Ee[i]$；$l[k] = El[j] - dur(<E_i, E_j>)$。

根据上面的分析，可以得到计算关键路径的算法如下。

(1) 输入 m 条带权的有向边，建立邻接表结构。

(2) 从源点 E_0 出发，令 $Ee[0]= 0$，按拓扑有序的顺序计算每个顶点的 $Ee[i]$，$i = 1, 2, \cdots, n-1$。若拓扑排序的循环次数小于顶点数 n，则说明网络中存在有向环，不能继续求关键路径。

(3) 从汇点 E_{n-1} 出发，令 $El[n-1] = Ee[n-1]$，按逆拓扑有序顺序求各顶点的 $El[i]$，$i = n-2, n-3, \cdots, 0$。

(4) 根据各顶点的 $Ee[i]$和 $El[i]$，求各条有向边的 $e[k]$和 $l[k]$。

(5) 对网络中的每条边，如果满足 $e[k] == l[k]$，则是关键活动；求出所有关键活动并输出。

求关键路径的算法实现详见例 2.9。

例 2.9　求图 2.37 所示的 AOE 网络的关键路径并输出。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 n 和边数 m，然后输入每条边，每条边的数据占一行。格式为：$u\ v$，表示从顶点 u 到顶点 v 的一条有向边。

分析：

在下面的代码中，表示边结点的结构体 ArcNode 增加了一个表示活动序号的成员 no。在程序中定义了两个边链表表头指针数组，List1[i]指向由顶点 i 发出的边构造的边链表，List2[i]指向由进入顶点 i 的边构造的边链表。这两个表头指针数组及边链表构造结果如图 2.38 所示。

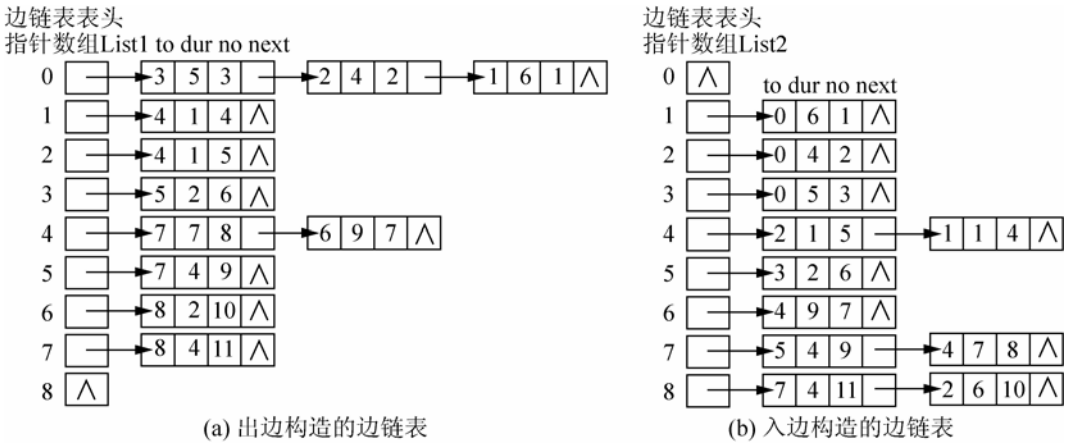


图 2.38　关键路径求解：两个邻接表

在 CriticalPath 函数中，首先根据 List1 进行拓扑排序并求各顶点的 *Ee*[*i*]；然后根据 *List*2 进行逆拓扑排序并求 *El*[*i*]；最后再根据 List1 扫描每条边一次，求每条边所表示的活动的最早可能开始时间 *e*[*k*]和最迟允许开始时间 *L*[*k*]，并判断是否为关键活动，如果是则输出。对图 2.37 所示的 AOE 网计算后各个量的结果如图 2.39 所示。

事件	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8
Ee[*i*]	0	6	4	5	7	7	16	14	18
El[*i*]	0	6	6	6	7	10	16	14	18

边 活动	<0, 1> a_1	<0, 2> a_2	<0, 3> a_3	<1, 4> a_4	<2, 4> a_5	<3, 5> a_6	<4, 6> a_7	<4, 7> a_8	<5, 7> a_9	<6, 8> a_{10}	<7, 8> a_{11}
E[*k*]	0	0	0	6	4	5	7	7	7	16	14
l[*k*]	0	2	3	6	6	8	7	7	10	16	14
l[*k*]− *e*[*k*]	0	2	3	0	2	3	0	0	3	0	0
关键活动	是			是			是	是		是	是

图 2.39　关键路径求解：相关量的计算结果

代码如下：

```
#define MAXN 100          //顶点个数的最大值
#define MAXM 200          //边数的最大值
struct ArcNode
{
    int to, dur, no;      //边的另一个顶点，持续时间，活动序号
    ArcNode *next;
};
int n, m;                 //顶点个数、边数
ArcNode* List1[MAXN];     //每个顶点的边链表表头指针(出边表)
ArcNode* List2[MAXN];     //每个顶点的边链表表头指针(入边表)
```

```
int count1[MAXN];       //各顶点的入度
int count2[MAXN];       //各顶点的出度
int Ee[MAXN];           //各事件的最早可能开始时间
int El[MAXN];           //各事件的最迟允许开始时间
int e[MAXM];            //各活动的最早可能开始时间
int L[MAXM];            //各活动的最迟允许开始时间
void CriticalPath(  )   //求关键路径
{
    //拓扑排序求 Ee
    int i, j, k;
    int top1=-1;
    ArcNode *temp1;
    memset( Ee, 0, sizeof(Ee) );
    for( i=0; i<n; i++ )
    {
        if( count1[i]==0 )
        {
            count1[i]=top1;  top1=i;
        }
    }
    for( i=0; i<n; i++ )
    {
        if( top1==-1 )
        {
            printf( "Network has a cycle!\n" );  return;
        }
        else
        {
            j=top1;  top1=count1[top1];
            temp1=List1[j];
            while( temp1!=NULL )
            {
                k=temp1->to;
                if( --count1[k]==0 )
                {
                    count1[k]=top1;  top1=k;
                }
                if( Ee[j]+temp1->dur>Ee[k] )   //有向边<j,k>
                    Ee[k]=Ee[j]+temp1->dur;
                temp1=temp1->next;
            }//end of while
        }//end of else
    }//end of for
    //逆拓扑排序求 El
    int top2=-1;
    ArcNode* temp2;
    for( i=0; i<n; i++ )
```

```
        {
            El[i]=Ee[n-1];
            if( count2[i]==0 )
            {
                count2[i]=top2;  top2=i;
            }
        }
        for( i=0; i<n; i++ )
        {
            j=top2;  top2=count2[top2];
            temp2=List2[j];
            while( temp2!=NULL )
            {
                k=temp2->to;
                if( --count2[k]==0 )
                {
                    count2[k]=top2;  top2=k;
                }
                if( El[j]-temp2->dur<El[k] )   //有向边<k,j>
                    El[k]=El[j]-temp2->dur;
                temp2=temp2->next;
            }//end of while
        }//end of for
        //求各活动的 e[k]和 L[k]
        memset( e, 0, sizeof(e) );  memset( L, 0, sizeof(L) );
        printf( "The critical activities are:\n" );
        for( i=0; i<n; i++ )   //释放边链表上各边结点所占用的存储空间
        {
            temp1=List1[i];
            while( temp1!=NULL )
            {
                j=temp1->to;  k=temp1->no;  //有向边<i,j>
                e[k]=Ee[i];  L[k]=El[j]-temp1->dur;
                if( e[k]==L[k] )  printf( "a%d : %d->%d\n", k, i, j );
                temp1=temp1->next;
            }
        }
    }
    int main( )
    {
        int i, u, v, w; //循环变量、边的起点和终点
        scanf( "%d%d", &n, &m );   //读入顶点个数 n，边数
        memset( List1, 0, sizeof(List1) );  memset( List2, 0, sizeof(List2) );
        memset( count1,0,sizeof(count1); memset( count2,0,sizeof (count2) );
        ArcNode *temp1, *temp2;
        for( i=0; i<m; i++ )
        {
```

```
            scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
            count1[v]++;
            temp1=new ArcNode; //构造邻接表
            temp1->to=v; temp1->dur=w;
            temp1->no=i+1; temp1->next=NULL;
            if( List1[u]==NULL ) List1[u]=temp1;
            else{ temp1->next=List1[u]; List1[u]=temp1; }
            count2[u]++;
            temp2=new ArcNode; //构造逆邻接表
            temp2->to=u; temp2->dur=w;
            temp2->no=i+1; temp2->next=NULL;
            if( List2[v]==NULL ) List2[v]=temp2;
            else{ temp2->next=List2[v]; List2[v]=temp2; }
        }
        CriticalPath( );
        for( i=0; i<n; i++ )   //释放边链表上各边结点所占用的存储空间
        {
            temp1=List1[i]; temp2=List2[i];
            while( temp1!=NULL )
            {
                List1[i]=temp1->next; delete temp1; temp1=List1[i];
            }
            while( temp2!=NULL )
            {
                List2[i]=temp2->next; delete temp2; temp2=List2[i];
            }
        }
        return 0;
    }
```

该程序的运行示例如下。

输入：

```
9 11
0 1 6
0 2 4
0 3 5
1 4 1
2 4 5
3 5 2
4 6 9
4 7 7
5 7 4
6 8 2
7 8 4
```

输出：

```
The critical activities are:
a1 : 0->1
a4 : 1->4
a8 : 4->7
a7 : 4->6
a10 : 6->8
a11 : 7->8
```