第5章 图

5.1 基础实践

5.1.1 基于邻接矩阵的广度优先搜索遍历

1. 实践目的

(1) 能够正确描述图的邻接矩阵存储在计算机中的表示。

(2) 能够正确编写在邻接矩阵存储表示的图上进行广度优先搜索遍历的实现算法。

(3) 能够编写程序验证广度优先搜索遍历算法的正确性。

2. 实践内容

(1) 用邻接矩阵作为图的存储表示创建一个图。

(2) 在邻接矩阵存储表示的图上完成广度优先搜索遍历操作。

3. 实践要求

1) 数据结构

由于实践内容指定是在邻接矩阵存储表示的图上实现操作,而对于一个具有 n 个顶点的图来说,邻接矩阵是一个 n 阶方阵,定义如下:

$$A[i][j]=\begin{cases}1, & (v_i,v_j)\in E \text{ 或} \{v_i,v_j\}\in E\\ 0, & (v_i,v_j)\notin E \text{ 或} \{v_i,v_j\}\notin E\end{cases}$$

其中,$0\leqslant i,j\leqslant n-1$。而对于网来说,假设 w_{ij} 代表边(v_i,v_j)或$\{v_i,v_j\}$上的权值,则网的邻接矩阵也是一个 n 阶方阵,定义如下:

$$A[i][j]=\begin{cases}w_{ij}, & (v_i,v_j)\in E \text{ 或} \{v_i,v_j\}\in E\\ \infty, & (v_i,v_j)\notin E \text{ 或} \{v_i,v_j\}\notin E\end{cases}$$

其中,$0\leqslant i,j\leqslant n-1$。由此定义,可以将图的邻接矩阵存储在一个二维数组中,而对于图或网来说又有“无向”与“有向”之分,为此,它的存储结构具体描述如下:

```
#define INFINITY INT_MAX                    //最大值∞
#define MAX_VERTEX_NUM 20                   //约定的最大顶点数
typedef enum
{    UDG,                                   //无向图(UnDirected Graph)
     DG,                                    //有向图(Directed Graph)
     UDN,                                   //无向网(UnDirected Network)
     DN                                     //有向网(Directed Network)
} GraphKind;
```

```
typedef char * VertexType;     //VertexType 是顶点类型,一般表示顶点名等信息,这里采用字符串
typedef int VRType;                                          //VRType 是顶点关系类型
                                                             //对图,用 1 或 0 表示是否相邻
                                                             //对网,则为权值类型

typedef struct                                               //图的类型定义
  {     VertexType vexs[MAX_VERTEX_NUM];                     //顶点信息
        VRType arcs [MAX_VERTEX_NUM][MAX_VERTEX_NUM];        // 邻接矩阵
        int vexnum, arcnum;                                  // 顶点数和边数
        GraphKind kind;                                      // 图的种类标志
  } MGraph;
```

2）函数接口说明

```
VertexType GetVex(MGraph G, int v)
//获取指定顶点值的操作: v 是已知图 G 的某个顶点号,返回顶点 v 的值
int LocateVex(MGraph G, VertexType u)
//顶点定位操作: 在已知图 G 中查找指定的顶点 u,若 G 中存在顶点 u,则返回该顶点在图中位置;
//否则返回"空"
int FirstAdjVex(MGraph G, int v)
//求首个邻接点的操作: v 是已知图 G 的某个顶点,返回 v 的首个邻接点,若顶点在 G 中没有邻接点,
//则返回"空"
int NextAdjVex(MGraph G, int v, int w)
//求下一个邻接点的操作: v 是已知图 G 的某个顶点,w 是 v 的邻接点,返回 v 的(相对于 w 的)下一个
//邻接点;若 w 是 v 的最后一个邻接点,则返回"空"
Status CreateUDG(MGraph &G)
// 采用数组(邻接矩阵)表示法,构造无向图 G
Status CreateDG(MGraph &G)
//采用数组(邻接矩阵)表示法,构造有向图 G
Status CreateUDN(MGraph &G)
// 采用数组(邻接矩阵)表示法,构造无向网 G
Status CreateDN(MGraph &G)
// 采用数组(邻接矩阵)表示法,构造有向网 G
Status CreateGraph(MGraph& G)
// 采用数组(邻接矩阵)表示法,构造图 G
void BFS(MGraph G, int v)
//从顶点 v 开始按广度优先搜索遍历图 G 中 v 所在的连通分量,使用辅助队列 Q 和访问标志数
//组 visited
void BFSTraverse(MGraph G)
//按广度优先搜索遍历图 G
```

3）输入输出说明

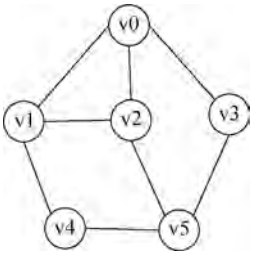
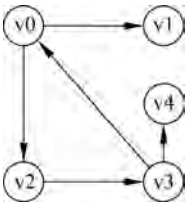
输入说明：输入信息的第 1 行为图的类型，其中 0 表示无向图 UDG，1 表示有向图 DG，2 表示无向网 UDN，3 表示有向网 DN；第 2 行为构造无向图的顶点数 m；第 3 行为边的数目 n。其次输入的 m 行是每个顶点信息，最后输入的 n 行是每条边的顶点对信息。

输出说明：最后一行输出所构造图的广度优先搜索遍历序列。为了能使输入与输出信息时更加人性化，可以在指定的输入或输出信息之前适当添加相关提示信息。

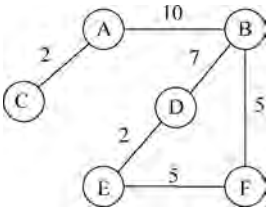
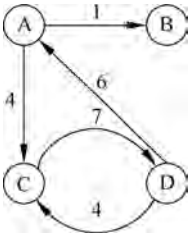
4）测试用例

测试用例信息如表 5-1 所示。

表 5-1　测试用例

序号	输入	输　出	说　明
1	0 6 8 v0 v1 v2 v3 v4 v5 v0 v1 v0 v2 v0 v3 v1 v2 v1 v4 v2 v5 v3 v5 v4 v5	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入顶点 G.vexs[4]： 输入顶点 G.vexs[5]： 输入第 1 条边(vi vj)： 输入第 2 条边(vi vj)： 输入第 3 条边(vi vj)： 输入第 4 条边(vi vj)： 输入第 5 条边(vi vj)： 输入第 6 条边(vi vj)： 输入第 7 条边(vi vj)： 输入第 8 条边(vi vj)： 广度优先搜索遍历序列： v0 v1 v2 v3 v4 v5	构造如下图所示无向图，如果正常输入，测试结果正确 
2	1 5 5 v0 v1 v2 v3 v4 v0 v1 v0 v2 v2 v3 v3 v0 v3 v4	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入顶点 G.vexs[4]： 输入第 1 条边<vi vj>： 输入第 2 条边<vi vj>： 输入第 3 条边<vi vj>： 输入第 4 条边<vi vj>： 输入第 5 条边<vi vj>： 广度优先搜索遍历序列： v0 v1 v2 v3 v4	构造如下图所示有向图，如果正常输入，测试结果正确 

续表

序号	输入	输　出	说　明
3	2 6 6 A B C D E F A B 10 A C 2 B D 7 B F 5 D E 2 E F 5	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入顶点 G.vexs[4]： 输入顶点 G.vexs[5]： 输入第 1 条边 vi、vj 和权值 w： 输入第 2 条边 vi、vj 和权值 w： 输入第 3 条边 vi、vj 和权值 w： 输入第 4 条边 vi、vj 和权值 w： 输入第 5 条边 vi、vj 和权值 w： 输入第 6 条边 vi、vj 和权值 w： 广度优先搜索遍历序列： A B C D F E	构造如下图所示无向网，如果正常输入，测试结果正确 
4	3 4 5 A B C D A B 1 A C 4 C D 7 D A 6 D C 4	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入第 1 条边 vi、vj 和权值 w： 输入第 2 条边 vi、vj 和权值 w： 输入第 3 条边 vi、vj 和权值 w： 输入第 4 条边 vi、vj 和权值 w： 输入第 5 条边 vi、vj 和权值 w： 广度优先搜索遍历序列： A B C D	构造如下图所示有向网，如果正常输入，测试结果正确 

4. 解决方案

上述接口的实现方法分析如下。

(1) 获取指定顶点值的操作 GetVex(G, v)：输入参数为图 G 和顶点编号 v，由于图 G 中的顶点信息保存在 vexs 数组中，因此，只需返回 vexs 数组中下标为 v 的元素值即可；若编号 v 越界，则退出。该操作只需随机存取顶点数组，则对一个具有 n 个顶点的图 G，其时间复杂度是 O(1)。

(2) 顶点定位操作 LocateVex(G,u)：输入参数为图 G 和要找的顶点 u，由于图 G 中的顶点信息保存在 vexs 数组中，因此，只需要在 vexs 数组中通过依次比对其中的顶点信息来

查找到顶点 u,若查找成功,则返回该顶点在数组中的序号,否则返回-1。该操作需遍历顶点数组。对一个具有 n 个顶点的图 G,其时间复杂度是 O(n)。

(3) 求首个邻接点的操作 FirstAdjVex(G,v):已知 v 是图 G 的某个顶点(0≤v<G.vexnum),要求返回 v 的首个邻接点,则需遍历邻接矩阵 arcs 的第 v 行,找到首个非 0,或非无穷大的值的元素,并返回其列标值;若找不到,意味着顶点 v 在 G 中没有邻接点,则返回-1。该操作对一个具有 n 个顶点的图 G,其时间复杂度是 O(n)。

(4) 求下一个邻接点的操作 NextAdjVex(G, v, w):已知 v 是已知图 G 的某个顶点,w 是 v 的邻接点(0≤v, w<G.vexNum),要返回 v 的(相对于 w 的)下一个邻接点,则需从 w+1 位置开始遍历邻接矩阵 arcs 的第 v 行,找到第一个非 0,或非无穷大的值的元素,并返回其列下标值;若找不到,意味着顶点 v(相对于 w 的)没有下一个邻接点,则返回-1。该操作对一个具有 n 个顶点的图 G,其时间复杂度是 O(n)。

(5) 采用数组(邻接矩阵)表示法,构造无向图 G 操作 CreateUDG(&G):首先输入顶点数和边数确定邻接矩阵的大小,再根据顶点数输入各个顶点的具体信息,根据边数输入各条边依附的两个顶点信息,根据顶点定位操作获取顶点的位置,最后将邻接矩阵中对应位置的元素值置为 1。

特别注意:由于无向图的邻接矩阵是对称的,所以在输入一条边的信息后,则需对邻接矩阵的相应位置的两个对称元素值都置为 1。

(6) 采用数组(邻接矩阵)表示法,构造有向图 G 操作 CreateDG(&G):本操作与操作(5)的区别是在构造有向图时,如果输入一条孤的信息后,只需对邻接矩阵中的相应位置上的一个元素值置为 1。

(7) 采用数组(邻接矩阵)表示法,构造无向网 G 操作 CreateUDN(&G):本操作与操作(5)的区别是在构造无向网时,如果输入一条边的信息后,需对邻接矩阵的相应位置的两个对称元素值置为权值。

(8) 采用数组(邻接矩阵)表示法,构造有向网 G 操作 CreateDN(&G):本操作与操作(6)的区别是在构造有向网时,如果输入一条孤的信息后,需对邻接矩阵中的相应位置上的一个元素值置为权值。

(9) 采用数组(邻接矩阵)表示法,构造图 G 操作 CreateGraph(&G):这是一个主控函数,根据输入的图的种类,调用操作(5)、(6)、(7)、(8)之一即可。

(10) 广度优先搜索遍历连通分量的操作 BFS(G, v):首先需要说明的是,图的遍历要比树的遍历复杂,因为图中一般存在回路,也就是说,在访问了某个顶点后,可能会沿着某条路径再次回到该顶点。为了避免顶点的重复访问,在遍历图的过程中,必须记下已访问过的顶点。为此,需要增设一个辅助数组 visited[MAX_VERTEX_NUM],其初始值为“假”,一旦访问了顶点 v_i,则置 visited[i]为“真”。

本操作是从图中的某个顶点 v 开始,先访问该顶点,再依次访问该顶点的每一个未被访问过的邻接点,假设为 w_1,w_2,…;然后按此顺序访问顶点 w_1,w_2,…的各个还未被访问过的邻接点。重复上述过程,直到图中的所有顶点都被访问过为止。也就是说,广度优先搜索遍历的过程是一个以顶点 v 为起始点,由近及远,依次访问和顶点 v 有路径相通且路径长度为 1,2,3…的顶点,并且遵循“先被访问的顶点,其邻接点就先被访问”。广度优先搜索是一种分层的搜索过程,每向前走一步就可能访问一批顶点。因此,在广度优先搜索遍历中,需

要使用队列，依次记住被访问过的顶点。因此，本操作开始时，访问起始点 v，并将其插入队列中，以后每次从队列中删除一个数据元素，就依次访问它的每一个未被访问过的邻接点，并将其插入队列中。这样，当队列为空时，表明所有与起始点相通的顶点都已被访问完毕，遍历结束。

例如：在如图 5-1 所示的无向图 G 中，从顶点 v_0 出发进行广度优先搜索遍历的过程如图 5-2 所示。在访问过程中，队列的状态及操作过程如表 5-2 所示。

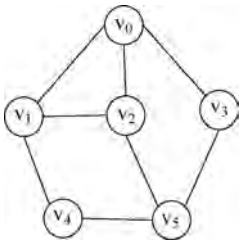


图 5-1　无向图 G

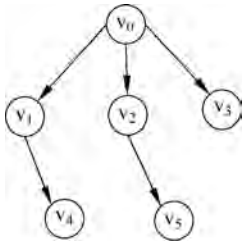


图 5-2　无向图 G 的广度优先搜索的过程

表 5-2　广度优先搜索遍历过程中队列的状态及操作过程

步骤	队列的状态	队列的操作过程
1	v_0	从顶点 v_0 开始执行广度优先搜索，将顶点 v_0 入队
2	v_1，v_2，v_3	将顶点 v_0 从队列中取出，将顶点 v_0 的邻接点 v_1，v_2 和 v_3 依次入队
3	v_2，v_3，v_4	将顶点 v_1 从队列中取出，然后，按照先访问顶点 v_1 未被访问过的邻接点，再访问顶点 v_2 未被访问过的邻接点，最后访问顶点 v_3 未被访问过的邻接点的次序，将 v_1 的邻接点 v_4 入队，由于顶点 v_1 的邻接点 v_2 已被访问过，故不必入队
4	v_3，v_4，v_5	将顶点 v_2 从队列中取出，再将顶点 v_2 未被访问过的邻接点 v_5 入队
5	v_4，v_5	将顶点 v_3 从队列中取出，访问顶点 v_3 的邻接点，由于顶点 v_3 的邻接点 v_5 已被访问过，故不必入队
6		按照先访问顶点 v_4 未被访问过的邻接点，再访问顶点 v_5 未被访问过的邻接点的次序，由于顶点 v_4 和 v_5 的邻接点都被访问过。将顶点 v_4 和 v_5 依次从队列中取出。此时队列为空，广度优先搜索遍历结束，顶点出队的顺序，就是广度优先搜索遍历的序列，即{v_0，v_1，v_2，v_3，v_4，v_5}

(11) 广度优先搜索遍历图的操作 BFSTraverse(G)：操作(10)仅能遍历顶点 v 所在的连通分量，如果图为非连通图，则需要另选图中一个未曾被访问的顶点作为起始点，重复上述过程，直到图中所有顶点都被访问到为止。

5. 程序代码参考

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "LinkQueue.h"
#define ERROR 0
#define OK 1
```

```
#define OVERFLOW -2
typedef int Status;

#define INFINITY INT_MAX                                //最大值∞
#define MAX_VERTEX_NUM 20                               //最大顶点个数
typedef enum {
    UDG,                                                //无向图(UnDirected Graph)
    DG,                                                 //有向图(Directed Graph)
    UDN,                                                //无向网(UnDirected Network)
    DN                                                  //有向网(Directed Network)
} GraphKind;
typedef char * VertexType;    //VertexType 是顶点类型,为了表示地名等信息,这里采用字符串
typedef int VRType;                                     //VRType 是顶点关系类型
                                                        //对图,用 1 或 0 表示是否相邻
                                                        //对网,则为权值类型

typedef struct {                                        // 图的定义
     VertexType vexs[MAX_VERTEX_NUM];                   //顶点信息
     VRType arcs[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 邻接矩阵
     int vexnum, arcnum;                                // 顶点数和边数
     GraphKind kind;                                    // 图的种类标志
} MGraph;

int visited[MAX_VERTEX_NUM];                            // 访问标志数组

VertexType GetVex(MGraph G, int v)
//获取指定顶点值的操作,v 是已知图 G 的某个顶点号,返回顶点 v 的值
{   if (v < 0 || v >= G.vexnum)
        exit(0);
    return G.vexs[v];
}

int LocateVex(MGraph G, VertexType u)
//顶点定位操作,在已知图 G 中查找指定的顶点 u,若 G 中存在顶点 u,则返回该顶点在图中位置;
//否则返回"空"
{   int i;
    for (i = 0; i < G.vexnum; i++)
        if (strcmp(G.vexs[i], u) == 0)
            return i;
    return -1;
}

int FirstAdjVex(MGraph G, int v)
//求首个邻接点的操作,v 是已知图 G 的某个顶点,返回 v 的首个邻接点,若顶点在 G 中没有邻接点,
//则返回"空"
{   if (v < 0 || v >= G.vexnum)
        return -1;
    for (int j = 0; j < G.vexnum; j++)            //遍历邻接矩阵第 v 行
        if (G.arcs[v][j] != 0 && G.arcs[v][j] < INFINITY)
```

```
                return j;
    return -1;
}

int NextAdjVex(MGraph G, int v, int w)
//求下一个邻接点的操作,v是已知图G的某个顶点,w是v的邻接点,返回v的(相对于w的)下一
//个邻接点;若w是v的最后一个邻接点,则返回"空"
{   if (v<0 || v>=G.vexnum)
         return -1;
    for (int j = w + 1; j<G.vexnum; j++)  //遍历邻接矩阵第v行
         if (G.arcs[v][j] != 0 && G.arcs[v][j]<INFINITY)
              return j;
    return -1;
}

Status CreateUDG(MGraph& G)
{                                          // 采用数组(邻接矩阵)表示法,构造无向图G
    int i, j, k;
    VertexType v1 = (char*)malloc(20), v2 = (char*)malloc(20);
    printf("输入顶点数G.vexnum: ");
    scanf("%d", &G.vexnum);
    printf("输入边数G.arcnum: ");
    scanf("%d", &G.arcnum);
    for (i = 0; i<G.vexnum; i++) {
         printf("输入顶点G.vexs[%d]: ", i);
         G.vexs[i] = (char*)malloc(20);
         scanf("%s", G.vexs[i]);
    }                                      // 构造顶点向量
    for (i = 0; i<G.vexnum; i++)           // 初始化邻接矩阵
         for (j = 0; j<G.vexnum; j++)
              G.arcs[i][j] = 0;
    for (k = 0; k<G.arcnum; k++) {         // 构造邻接矩阵
         printf("输入第%d条边(vi vj)(用空格隔开): ", k + 1);
         scanf("%s %s", v1, v2);           // 输入一条边依附的顶点
         i = LocateVex(G, v1); j = LocateVex(G, v2);       // 确定v1和v2在G中位置
         G.arcs[i][j] = 1;                 //边(v1,v2)
         G.arcs[j][i] = G.arcs[i][j];      // 置(v1,v2)的对称边(v2,v1)
    }
    return OK;
}

Status CreateDG(MGraph& G)
{                                          // 采用数组(邻接矩阵)表示法,构造有向图G
    int i, j, k;
    VertexType v1 = (char*)malloc(20), v2 = (char*)malloc(20);
    printf("输入顶点数G.vexnum: ");
    scanf("%d", &G.vexnum);
    printf("输入边数G.arcnum: ");
    scanf("%d", &G.arcnum);
```

```
    for (i = 0; i < G.vexnum; i++) {
        printf("输入顶点 G.vexs[ %d]: ", i);
        G.vexs[i] = (char *)malloc(20);
        scanf(" %s", G.vexs[i]);
    }                                          // 构造顶点向量
    for (i = 0; i < G.vexnum; i++)             // 初始化邻接矩阵
        for (j = 0; j < G.vexnum; j++)
            G.arcs[i][j] = 0;
    for (k = 0; k < G.arcnum; k++) {           // 构造邻接矩阵
        printf("输入第 %d 条边< vi vj >(用空格隔开): ", k + 1);
        scanf(" %s %s", v1, v2);               // 输入一条边依附的顶点
        i = LocateVex(G, v1);                  // 确定 v1 和 v2 在 G 中位置
        j = LocateVex(G, v2);
        G.arcs[i][j] = 1;                      // 弧< vi,vj >
    }
    return OK;
}                                              // CreateDG

Status CreateUDN(MGraph& G)
{                                              // 采用数组(邻接矩阵)表示法,构造无向网 G
    int i, j, k, w;
    VertexType v1 = (char *)malloc(20), v2 = (char *)malloc(20);
    printf("输入顶点数 G.vexnum: ");
    scanf(" %d", &G.vexnum);
    printf("输入边数 G.arcnum: ");
    scanf(" %d", &G.arcnum);
    for (i = 0; i < G.vexnum; i++) {
        printf("输入顶点 G.vexs[ %d]: ", i);
        G.vexs[i] = (char *)malloc(20);
        scanf(" %s", G.vexs[i]);
    }                                          // 构造顶点向量
    for (i = 0; i < G.vexnum; i++)             // 初始化邻接矩阵
        for (j = 0; j < G.vexnum; j++)
            G.arcs[i][j] = INFINITY;
    for (k = 0; k < G.arcnum; k++) {           // 构造邻接矩阵
        printf("输入第 %d 条边 vi、vj 和权值 w (用空格隔开): ", k + 1);
        scanf(" %s %s %d", v1, v2, &w);        // 输入一条边依附的顶点及权值
        i = LocateVex(G, v1);                  // 确定 v1 和 v2 在 G 中位置
        j = LocateVex(G, v2);
        G.arcs[i][j] = w;                      // 边(v1,v2)的权值
        G.arcs[j][i] = G.arcs[i][j];           // 置(vi,vj)的对称边(vj,vi)
    }
    return OK;
}                                              // CreateUDN

Status CreateDN(MGraph& G)
{                                              // 采用数组(邻接矩阵)表示法,构造有向网 G
    int i, j, k, w;
    VertexType v1 = (char *)malloc(20), v2 = (char *)malloc(20);
```

```
    printf("输入顶点数 G.vexnum: ");
    scanf("%d", &G.vexnum);
    printf("输入边数 G.arcnum: ");
    scanf("%d", &G.arcnum);
    for (i = 0; i < G.vexnum; i++) {
        printf("输入顶点 G.vexs[%d]: ", i);
        G.vexs[i] = (char*)malloc(20);
        scanf("%s", G.vexs[i]);
    }                                          // 构造顶点向量
    for (i = 0; i < G.vexnum; i++)             // 初始化邻接矩阵
        for (j = 0; j < G.vexnum; j++)
            G.arcs[i][j] = INFINITY;
    for (k = 0; k < G.arcnum; k++) {           // 构造邻接矩阵
        printf("输入第%d条边 vi、vj 和权值 w(用空格隔开): ", k + 1);
        scanf("%s %s %d", v1, v2, &w);         // 输入一条边依附的顶点及权值
        i = LocateVex(G, v1);                  // 确定 v1 和 v2 在 G 中位置
        j = LocateVex(G, v2);
        G.arcs[i][j] = w;                      // 弧<vi,vj>的权值
    }
    return OK;
}                                              // CreateDN

Status CreateGraph(MGraph& G)
// 采用数组(邻接矩阵)表示法,构造图 G
{   printf("请输入图的种类: 0 表示无向图 UDG,1 表示有向图 DG,2 表示无向网 UDN,3 表示有向
网 DN\n");
    scanf("%d", &G.kind);                      // 自定义输入函数,读入一个随机值
    switch (G.kind) {
    case UDG: return CreateUDG(G);             // 构造无向图 G
    case DG: return CreateDG(G);               // 构造有向图 G
    case UDN: return CreateUDN(G);             // 构造无向网 G
    case DN: return CreateDN(G);               // 构造有向网 G
    default: return ERROR;
    }
}                                              // CreateGraph


void BFS(MGraph G, int v)
//按广度优先搜索遍历连通分量 G.使用辅助队列 Q 和访问标志数组 visited
{   int u, w;
    VertexType V;
    LinkQueue Q;
    InitQueue(&Q);                             // 置空的辅助队列 Q
    visited[v] = 1;
    V = GetVex(G, v);
    printf("%s ", V);
    EnQueue(&Q, v);                            // v 入队
    while (!QueueEmpty(Q))                     // 队列不空
    {
```

```
            DeQueue(&Q, &u);                          //元素出队并置为 u
            for (w = FirstAdjVex(G, u); w >= 0; w = NextAdjVex(G, u, w))
                 if (!visited[w])                     // w 为 u 的尚未访问的邻接顶点
                 {
                       visited[w] = 1;
                       V = GetVex(G, w);
                       printf("%s ", V);
                       EnQueue(&Q, w);                // w 入队
                 }
      }
}

void BFSTraverse(MGraph G)
//按广度优先搜索遍历图 G
{    int v;
     for (v = 0; v < G.vexnum; v++)
          visited[v] = 0;                             // 置初值
     for (v = 0; v < G.vexnum; v++)                   // 如果是连通图,只 v = 0 就遍历全图
          if (!visited[v])                            // v 尚未访问
               BFS(G, v);
}

int main()
{    MGraph G;
     CreateGraph(G);
     printf("广度优先搜索遍历序列: ");
     BFSTraverse(G);
}
```

6. 运行结果参考

程序测试用例的运行结果如图 5-3 所示。

7. 延伸思考

(1) 给定一个图,对于固定的存储结构,按给定的广度优先搜索遍历算法得到的搜索结果是否唯一?

(2) 如果在图的邻接表存储结构上实现图的广度优先搜索遍历,其算法该如何编写?同时比较并分析在两种不同的存储结构上实现同一种遍历,其时间性能的优劣。

5.1.2 基于邻接表的深度优先搜索遍历

1. 实践目的

(1) 能够正确描述图的邻接表存储在计算机中的表示。

(2) 能够正确编写邻接表存储表示的图上进行深度优先搜索遍历的实现算法。

(3) 能够编写程序验证深度优先搜索遍历算法的正确性。

2. 实践内容

(1) 用邻接表作为图的存储结构创建一个图。

(2) 在邻接表存储表示的图上完成深度优先搜索遍历操作。

(a) 测试用例1

(b) 测试用例2

(c) 测试用例3

(d) 测试用例4

图 5-3　程序部分测试用例的运行结果

3. 实践要求

1）数据结构

由于实践内容中指定是在邻接表存储表示的图上实现指定操作，而邻接表是由一个顺序存储的顶点表和 n 个链式存储的边表组成的。其中：顶点表由顶点结点组成；边表是由边(或弧)结点组成的一个单链表，表示所有依附于顶点 v_i 的边(对于有向图就是所有以 v_i

为始点的弧)。例如,图 5-1 的无向图 G 所对应的邻接表存储表示如图 5-4 所示。

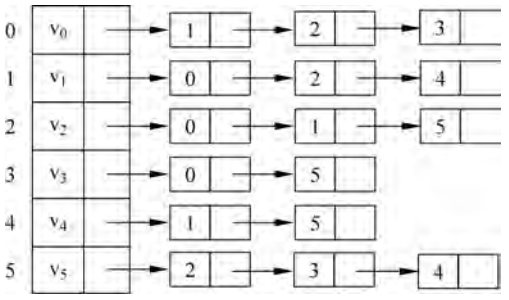


图 5-4　无向图 G_1 的邻接表

图的邻接表存储结构描述如下:

```
//边(或弧)结点类型定义
typedef struct ArcNode
{     int  adjVex;            //该边(弧)所指向的顶点的位置
      int  value;             //该边的权值
      ArcNode  * nextArc;     //指向下条弧的指针
}ArcNode;
typedef char * VertexType;    //VertexType 是顶点类型,一般表示顶点名等信息,这里采用字符串
//顶点结点类型定义
typedef struct VNode
{     VertexType  data;       //存放一个顶点的信息
       ArcNode  * firstArc;   //第一个表结点的地址,指向第一条依附该顶点的边(弧)的指针
}VNode, AdjList[MAX_VERTEX_NUM];
//图的邻接表存储结构类型定义
typedef struct
{     AdjList vertices;       //存放所有顶点信息的顶点数组
      int  vexNum, arcNum;    //图的顶点数和弧数
      int      kind;          //图的种类标志
}ALGraph;
```

2) 函数接口说明

```
VertexType GetVex(ALGraph G, int v)
//获取指定顶点值的操作: v 是已知图 G 的某个顶点号,返回顶点 v 的值
int LocateVex(ALGraph G, VertexType u)
//顶点定位操作: 在已知图 G 中查找指定的顶点 u,若 G 中存在顶点 u,则返回该顶点在图中位置;
//否则返回"空"
int FirstAdjVex(ALGraph G, int v)
//求首个邻接点的操作: v 是已知图 G 的某个顶点,返回 v 的首个邻接点,若顶点在 G 中没有邻接点,
//则返回"空"
int NextAdjVex(ALGraph G, int v, int w)
//求下一个邻接点的操作: v 是已知图 G 的某个顶点,w 是 v 的邻接点,返回 v 的(相对于 w 的)下一个
//邻接点。若 w 是 v 的最后一个邻接点,则返回"空"
Status CreateUDG(ALGraph &G)
// 采用邻接表表示法,构造无向图 G
```

```
Status CreateDG(ALGraph &G)
//采用邻接表表示法,构造有向图 G
Status CreateUDN(ALGraph &G)
// 采用邻接表表示法,构造无向网 G
Status CreateDN(ALGraph &G)
// 采用邻接表表示法,构造有向网 G
Status CreateGraph(ALGraph& G)
// 采用邻接表表示法,构造图 G
void DFS(ALGraph G, int v)
//从顶点 v 开始,按深度优先搜索遍历图 G 中 v 所在的连通分量。使用访问标志数组 visited
void DFSTraverse(ALGraph G)
//按深度优先搜索遍历图 G
```

3）输入输出说明

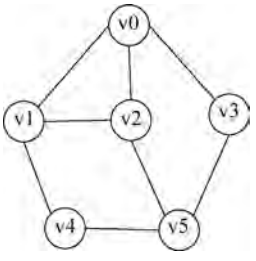
输入说明：输入信息的第 1 行为图的类型，其中 0 表示无向图 UDG，1 表示有向图 DG，2 表示无向网 UDN，3 表示有向网 DN；第 2 行为构造无向图的顶点数 m；第 3 行为边的数目 n。其次输入的 m 行是每个顶点的信息，最后输入的 n 行是每条边的信息。

输出说明：最后一行输出所构造图的深度优先搜索得到的遍历序列。

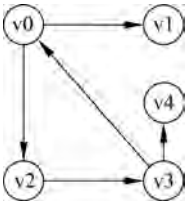
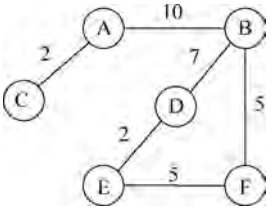
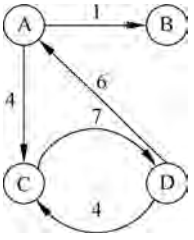
4）测试用例

测试用例信息如表 5-3 所示。

表 5-3　测试用例

序号	输入	输　出	说　明
1	0	请输入图的种类：	构造如下图所示无向图，如果正常输入，测试结果正确 
	6	输入顶点数 G.vexnum：	
	8	输入边数 G.arcnum：	
	v0	输入顶点 G.vexs[0]：	
	v1	输入顶点 G.vexs[1]：	
	v2	输入顶点 G.vexs[2]：	
	v3	输入顶点 G.vexs[3]：	
	v4	输入顶点 G.vexs[4]：	
	v5	输入顶点 G.vexs[5]：	
	v0 v1	输入第 1 条边(vi vj)：	
	v0 v2	输入第 2 条边(vi vj)：	
	v0 v3	输入第 3 条边(vi vj)：	
	v1 v2	输入第 4 条边(vi vj)：	
	v1 v4	输入第 5 条边(vi vj)：	
	v2 v5	输入第 6 条边(vi vj)：	
	v3 v5	输入第 7 条边(vi vj)：	
	v4 v5	输入第 8 条边(vi vj)：	
		深度优先搜索遍历序列： v0 v3 v5 v4 v1 v2	

续表

序号	输入	输　出	说　明
2	1 5 5 v0 v1 v2 v3 v4 v0 v1 v0 v2 v2 v3 v3 v0 v3 v4	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入顶点 G.vexs[4]： 输入第 1 条边< vi vj >： 输入第 2 条边< vi vj >： 输入第 3 条边< vi vj >： 输入第 4 条边< vi vj >： 输入第 5 条边< vi vj >： 深度优先搜索遍历序列： v0 v2 v3 v4 v1	构造如下图所示有向图，如果正常输入，测试结果正确 
3	2 6 6 A B C D E F A B 10 A C 2 B D 7 B F 5 D E 2 E F 5	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入顶点 G.vexs[4]： 输入顶点 G.vexs[5]： 输入第 1 条边 vi、vj 和权值 w： 输入第 2 条边 vi、vj 和权值 w： 输入第 3 条边 vi、vj 和权值 w： 输入第 4 条边 vi、vj 和权值 w： 输入第 5 条边 vi、vj 和权值 w： 输入第 6 条边 vi、vj 和权值 w： 深度优先搜索遍历序列： A C B F E D	构造如下图所示无向网，如果正常输入，测试结果正确 
4	3 4 5 A B C D A B 1 A C 4 C D 7 D A 6 D C 4	请输入图的种类： 输入顶点数 G.vexnum： 输入边数 G.arcnum： 输入顶点 G.vexs[0]： 输入顶点 G.vexs[1]： 输入顶点 G.vexs[2]： 输入顶点 G.vexs[3]： 输入第 1 条边 vi、vj 和权值 w： 输入第 2 条边 vi、vj 和权值 w： 输入第 3 条边 vi、vj 和权值 w： 输入第 4 条边 vi、vj 和权值 w： 输入第 5 条边 vi、vj 和权值 w： 深度优先搜索遍历序列： A C D B	构造如下图所示有向网，如果正常输入，测试结果正确 

4. 解决方案

上述接口的实现方法分析如下。

(1) 获取指定顶点值的操作 GetVex(G,v)：输入参数为图 G 和顶点编号，由于图 G 中的顶点信息保存在 vertices 数组的 data 数据域中，因此，只需返回 vertices 数组 v 号元素的 data 值即可；若编号 v 越界，则退出。该操作只需随机存取顶点数组，则对一个具有 n 个顶点的图 G，其时间复杂度是 $O(1)$。

(2) 顶点定位操作 LocateVex(G,u)：输入参数为图 G 和要找的顶点 u，由于图 G 中的顶点信息保存在 vertices 数组的 data 数据域中，因此，只需要在 vertices 数组中通过依次比对其中的顶点信息来查找到顶点 u，若查找成功，则返回该顶点在数组中的序号，否则返回 −1。该操作需遍历顶点数组。对一个具有 n 个顶点的图 G，其时间复杂度是 $O(n)$。

(3) 求首个邻接顶点的操作 FirstAdjVex(G,v)：已知 v 是图 G 的某个顶点($0 \leqslant v < G.vexnum$)，要求返回 v 的首个邻接顶点。首先，要定位顶点 v 对应的序号；其次，根据序号在顶点数组中直接找到对应顶点，再通过顶点结点中的 firstArc 域的值找到该顶点指向的首个边或弧结点的指针，如果指针值为空，说明没有邻接顶点，否则返回该指针指向的边或弧结点中 adjVex 域的顶点位置值。该操作需要通过访问顶点顺序表去确定 v 顶点的位置，再直接去找这个位置上的边或弧表上的首个结点，其时间复杂度是 $O(1)$。

(4) 求下一个邻接顶点的操作 NextAdjVex(G, v, w)：已知 v 是已知图 G 的某个顶点，w 是 v 的邻接点($0 \leqslant v, w < G.vexNum$)，要返回 v 的(相对于 w 的)下一个邻接顶点，则只需在依附于顶点 v 的边或弧表中沿着边结点的后继指针依次去查找 v 的邻接顶点 w，若找到，则该边弧结点的后继结点即为顶点 v 相对于顶点 w 的下一个邻接点。若此邻接顶点存在，则返回该邻接顶点的序号值，否则返回 −1。该操作需要通过访问顶点顺序表去确定 v 顶点的位置，然后再在这个位置上的边或弧表上去寻找边或弧结点 w 的下一个边弧结点，其时间复杂度为 $O(e/n)$。

(5) 采用邻接表表示法，构造无向图 G 操作 CreateUDG(&G)：本操作在建立边结点时，输入的都是边依附的顶点值，而不是顶点的位置，因此在建立每个边结点时，先要通过操作(2)确定顶点的位置，其时间复杂度是 $O(n)$，最后在邻接表对应的两个边表上采用头插法分别插入一个对称的边结点。因此，构造一个有 n 个顶点和 e 条边的图时，该操作的时间复杂度是 $O(n \times e)$。

(6) 采用邻接表表示法，构造有向图 G 操作 CreateDG(&G)：本操作与操作(5)的区别是在构造有向图时，只需在邻接表对应的一个弧表中插入一个弧结点即可。

(7) 采用邻接表表示法，构造无向网 G 操作 CreateUDN(&G)：本操作与操作(5)的区别是在构造无向网时，对边结点数据域赋值时需填入权值。

(8) 采用邻接表表示法，构造有向网 G 操作 CreateDN(&G)：本操作与操作(6)的区别是在构造有向网时，对弧结点数据域赋值时需填入权值。

(9) 采用邻接表表示法，构造图 G 操作 CreateGraph(&G)：这是一个主控函数，根据输入的图的种类，调用操作(5)、(6)、(7)、(8)之一即可。

(10) 深度优先搜索遍历连通分量的操作 DFS(G, v)：从图的某个顶点 v 开始访问，然后访问它的任意一个邻接点，假定为 w_1；再从 w_1 出发，访问与 w_1 邻接但未被访问的顶点，假定为 w_2；然后再从 w_2 出发，进行类似访问，如此进行下去，直至所有的邻接点都被访问

过为止。接着,回退一步,退到前一次刚访问过的顶点,看是否还有其他未被访问的邻接点。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。重复上述过程,直到连通图中的所有顶点都被访问过为止。该遍历的过程是一个递归的过程。

例如:在图 5-5 所示的无向图 G 中,从顶点 v0 出发进行深度优先搜索遍历的过程如图 5-6 所示[①]。

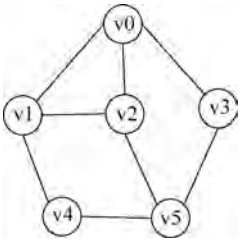


图 5-5　无向图 G

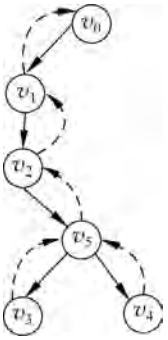


图 5-6　无向图 G 的深度优先搜索的过程

(11) 深度优先搜索遍历图的操作 DFSTraverse(G):操作(10)仅能遍历顶点 v 所在的连通分量,如果图为非连通图,则需要另选图中一个未曾被访问的顶点作为起始点,重复上述过程,直到图中所有顶点都被访问到为止。

5. 程序代码参考

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "LinkQueue.h"
#define ERROR 0
#define OK 1
#define OVERFLOW -2
typedef int Status;

#define INFINITY INT_MAX                    //最大值∞
#define MAX_VERTEX_NUM 20                   //最大顶点个数
typedef enum {
    UDG,                                    //无向图(UnDirected Graph)
    DG,                                     //有向图(Directed Graph)
    UDN,                                    //无向网(UnDirected Network)
    DN                                      //有向网(Directed Network)
} GraphKind;
typedef char * VertexType;     //VertexType 是顶点类型,为了表示地名等信息,这里采用字符串
//----- 图的邻接表存储表示 -----

//边(或弧)结点
```

① 图中以带箭头的实线表示遍历时的访问路径,以带箭头的虚线表示回溯的路径。

```
typedef struct ArcNode {
    int         adjVex;                 //该边(弧)所指向的顶点的位置
    int value;                          //该边的权值
    ArcNode * nextArc;                  //指向下条弧的指针
}ArcNode;

//头结点
typedef struct VNode {
     VertexType data;                   //顶点信息
     ArcNode * firstArc;                //边表的头指针,指向首条依附该顶点的边(弧)的指针
}VNode, AdjList[MAX_VERTEX_NUM];

typedef struct {
     AdjList vertices;
     int  vexnum, arcnum;               //图的顶点数和弧数
     int      kind;                     //图的种类标志
}ALGraph;

int visited[MAX_VERTEX_NUM];            //访问标志数组,全局变量

VertexType GetVex(ALGraph G, int v)
//获取指定顶点值的操作,v是已知图G的某个顶点号,返回顶点v的值
{   if (v < 0 || v >= G.vexnum)
          exit(0);
    return G.vertices[v].data;
}

int LocateVex(ALGraph G, VertexType u)
//顶点定位操作,在已知图G中查找指定的顶点u,若G中存在顶点u,则返回该顶点在图中位置;
//否则返回"空"
{   int i;
    for (i = 0; i < G.vexnum; ++i)
         if (strcmp(G.vertices[i].data, u) == 0)
               return i;
    return -1;
}

int FirstAdjVex(ALGraph G, int v)
//求首个邻接点的操作,v是已知图G的某个顶点,返回v的首个邻接点,若顶点在G中没有邻接点,
//则返回"空"
{   ArcNode * p;
    p = G.vertices[v].firstArc;
    if (p)
         return p->adjVex;
    else
         return -1;
}

int NextAdjVex(ALGraph G, int v, int w)
```

```
//求下一个邻接点的操作,v是已知图G的某个顶点,w是v的邻接点,返回v的(相对于w的)下一
//个邻接点;若w是v的最后一个邻接点,则返回"空"
{   ArcNode * p;
    p = G.vertices[v].firstArc;
    while (p && p->adjVex != w)                    //指针p不空且所指边结点不是w
        p = p->nextArc;
    if (!p || !p->nextArc)                         //没找到w或w是最后一个邻接点
        return -1;
    else
        return p->nextArc->adjVex;                 //返回v的(相对于w的)下个邻接点的序号

}

Status CreateUDG(ALGraph& G)
//采用邻接表存储表示,构造无向图G
{   int i, j, k;
    ArcNode * pi, * pj;
    VertexType v1 = (char *)malloc(20), v2 = (char *)malloc(20);
    printf("输入顶点数G.vexnum: ");
    scanf("%d", &G.vexnum);
    printf("输入边数G.arcnum: ");
    scanf("%d", &G.arcnum);
    for (i = 0; i < G.vexnum; i++)
    {                                              //构造顶点表
        printf("输入顶点G.vertices[%d].data: ", i);
        G.vertices[i].data = (char *)malloc(20);
        scanf("%s", G.vertices[i].data);           //输入顶点值
        G.vertices[i].firstArc = NULL;             //初始化链表头指针为"空"
    }
    for (k = 0; k < G.arcnum; k++)
    {                                               //输入各边并构造邻接表
        printf("输入第%d条边的两个顶点: ", k + 1);
        scanf("%s %s", v1, v2);                     //输入一条边的始点和终点
        i = LocateVex(G, v1);                       //确定v1和v2在G中位置,即顶点的序号
        j = LocateVex(G, v2);
        if (!(pi = (ArcNode *)malloc(sizeof(ArcNode))))       //创建新的结点pi
            exit(OVERFLOW);
        pi->adjVex = j;                             //对弧结点pi赋邻接点"位置"信息
        pi->nextArc = G.vertices[i].firstArc; //将结点pi插入链表G.vertices[i]的头部
        G.vertices[i].firstArc = pi;

        if (!(pj = (ArcNode *)malloc(sizeof(ArcNode))))        //创建新的结点pj
            exit(OVERFLOW);
        pj->adjVex = i;                             //对弧结点pj赋邻接点"位置"信息
        pj->nextArc = G.vertices[j].firstArc; //将结点pj插入链表G.vertices[j]的头部
        G.vertices[j].firstArc = pj;
    }
    return OK;
}
```

```
Status CreateDG(ALGraph& G)
//采用邻接表存储表示,构造有向网 G
{   int i, j, k;
    ArcNode * pi;
    VertexType v1 = (char * )malloc(20), v2 = (char * )malloc(20);
    printf("输入顶点数 G.vexnum: ");
    scanf(" % d", &G.vexnum);
    printf("输入边数 G.arcnum: ");
    scanf(" % d", &G.arcnum);
    for (i = 0; i < G.vexnum; ++i)
    {                                                    //构造顶点表
        printf("输入顶点 G.vertices[ % d].data: ", i);
        G.vertices[i].data = (char * )malloc(20);
        scanf(" % s", G.vertices[i].data);               //输入顶点值
        G.vertices[i].firstArc = NULL;                   //初始化链表头指针为"空"
    }                                                    //endfor
    for (k = 0; k < G.arcnum; k++)
    {                                                    //输入各边并构造邻接表
        printf("输入第 % d 条边的两个顶点: ", k + 1);
        scanf(" % s % s", v1, v2);                       //输入一条边的始点和终点
        i = LocateVex(G, v1); j = LocateVex(G, v2); //确定 v1 和 v2 在 G 中位置,即顶点的序号
        if (!(pi = (ArcNode * )malloc(sizeof(ArcNode))))     //创建新的结点 pi
            exit(OVERFLOW);
        pi -> adjVex = j;                                //对弧结点 pi 赋邻接点"位置"信息
        pi -> nextArc = G.vertices[i].firstArc;  //将结点 pi 插入链表 G.vertices[i]的头部
        G.vertices[i].firstArc = pi;
    }
    return OK;
}

Status CreateUDN(ALGraph& G)
//采用邻接表存储表示,构造无向网 G
{   int i, j, k, w;
    ArcNode * pi, * pj;
    VertexType v1 = (char * )malloc(20), v2 = (char * )malloc(20);
    printf("输入顶点数 G.vexnum: ");
    scanf(" % d", &G.vexnum);
    printf("输入边数 G.arcnum: ");
    scanf(" % d", &G.arcnum);
    for (i = 0; i < G.vexnum; i++)
    {                                                //构造顶点表
        printf("输入顶点 G.vertices[ % d].data: ", i);
        G.vertices[i].data = (char * )malloc(20);
        scanf(" % s", G.vertices[i].data);           //输入顶点值
        G.vertices[i].firstArc = NULL;               //初始化链表头指针为"空"
    }
    for (k = 0; k < G.arcnum; k++)
    {                                                //输入各边并构造邻接表
        printf("输入第 % d 条边 vi、vj 和权值 w : ", k + 1);
```

```
        scanf("%s %s %d", v1, v2, &w);
        i = LocateVex(G, v1);                    //确定 v1 和 v2 在 G 中位置,即顶点的序号
        j = LocateVex(G, v2);
        if (!(pi = (ArcNode*)malloc(sizeof(ArcNode))))        //创建新的结点 pi
            exit(OVERFLOW);
        pi->adjVex = j;                          //对弧结点 pi 赋邻接点"位置"信息
        pi->value = w;
        pi->nextArc = G.vertices[i].firstArc; //将 pi 结点插入链表 G.vertices[i]的头部
        G.vertices[i].firstArc = pi;
        if (!(pj = (ArcNode*)malloc(sizeof(ArcNode))))        //创建新的结点 pj
            exit(OVERFLOW);
        pj->adjVex = i;                          //对弧结点 pj 赋邻接点"位置"信息
        pj->value = w;
        pj->nextArc = G.vertices[j].firstArc; //将 pj 结点插入链表 G.vertices[j]的头部
        G.vertices[j].firstArc = pj;
    }
    return OK;
}

Status CreateDN(ALGraph& G)
//采用邻接表存储表示,构造有向网 G
{   int i, j, k, w;
    ArcNode* pi;
    VertexType v1 = (char*)malloc(20), v2 = (char*)malloc(20);
    printf("输入顶点数 G.vexnum: ");
    scanf("%d", &G.vexnum);
    printf("输入边数 G.arcnum: ");
    scanf("%d", &G.arcnum);
    for (i = 0; i < G.vexnum; ++i)
    {                                            //构造顶点表
        printf("输入顶点 G.vertices[%d].data: ", i);
        G.vertices[i].data = (char*)malloc(20);
        scanf("%s", G.vertices[i].data);         //输入顶点值
        G.vertices[i].firstArc = NULL;           //初始化链表头指针为"空"
    }                                            //endfor
    for (k = 0; k < G.arcnum; k++)
    {                                            //输入各边并构造邻接表
        printf("输入第%d条边 vi、vj 和权值 w: ", k + 1);
        scanf("%s %s %d", v1, v2, &w);
        i = LocateVex(G, v1);                    //确定 v1 和 v2 在 G 中位置,即顶点的序号
        j = LocateVex(G, v2);
        if (!(pi = (ArcNode*)malloc(sizeof(ArcNode))))          //创建新的结点 pi
            exit(OVERFLOW);
        pi->adjVex = j;                          //对弧结点 pi 赋邻接点"位置"信息
        pi->value = w;
        pi->nextArc = G.vertices[i].firstArc; //将 pi 结点/插入链表 G.vertices[i]的头部
        G.vertices[i].firstArc = pi;
    }
    return OK;
```

```
}

Status CreateGraph(ALGraph& G)
//采用邻接表,构造图
{   printf("请输入图的种类: 0 表示无向图 UDG,1 表示有向图 DG,2 表示无向网 UDN,3 表示有向网 DN\n");
    scanf("%d", &G.kind);
    switch (G.kind) {
    case UDG: return CreateUDG(G);                  //构造无向图 G
    case DG: return CreateDG(G);                    //构造有向图 G
    case UDN: return CreateUDN(G);                  //构造无向网 G
    case DN: return CreateDN(G);                    //构造有向网 G
    default: return ERROR;
    }
}

void DFS(ALGraph G, int v)
//从顶点 v 开始,按深度优先搜索遍历图 G 中 v 所在的连通分量。使用访问标志数组 visited
{   int w;
    VertexType v1;

    visited[v] = 1;                                 //设置访问标志为 1(已访问)
    v1 = GetVex(G, v);
    printf("%s ", v1);                              //访问第 v 个顶点
    for (w = FirstAdjVex(G, v); w >= 0; w = NextAdjVex(G, v, w))
        if (!visited[w])
            DFS(G, w);                              //对 v 的尚未访问的邻接点 w 递归调用 DFS
}

void DFSTraverse(ALGraph G)
//按深度优先搜索遍历图 G
{ int v;
    for (v = 0; v < G.vexnum; v++)
        visited[v] = 0;                             //访问标志数组初始化
    for (v = 0; v < G.vexnum; v++)
        if (!visited[v])
            DFS(G, v);                              //对尚未访问的顶点调用 DFS
}

int main()
{   ALGraph G;
    CreateGraph(G);
    printf("深度优先搜索遍历序列: ");
    DFSTraverse(G);
}
```

6. 运行结果参考

程序测试用例的运行结果如图 5-7 所示。

```
请输入图的种类：0表示无向图UDG，1表示有向图DG，2表示无向网UDN，3表示有向网DN
0
输入顶点数G.vexnum: 6
输入边数G.arcnum: 8
输入顶点G.vertices[0].data: v0
输入顶点G.vertices[1].data: v1
输入顶点G.vertices[2].data: v2
输入顶点G.vertices[3].data: v3
输入顶点G.vertices[4].data: v4
输入顶点G.vertices[5].data: v5
输入第1条边的两个顶点：v0 v1
输入第2条边的两个顶点：v0 v2
输入第3条边的两个顶点：v0 v3
输入第4条边的两个顶点：v1 v2
输入第5条边的两个顶点：v1 v4
输入第6条边的两个顶点：v2 v5
输入第7条边的两个顶点：v3 v5
输入第8条边的两个顶点：v4 v5
深度优先搜索遍历序列:  v0 v3 v5 v4 v1 v2
```

(a) 测试用例1

```
请输入图的种类：0表示无向图UDG，1表示有向图DG，2表示无向网UDN，3表示有向网DN
1
输入顶点数G.vexnum: 5
输入边数G.arcnum: 5
输入顶点G.vertices[0].data: v0
输入顶点G.vertices[1].data: v1
输入顶点G.vertices[2].data: v2
输入顶点G.vertices[3].data: v3
输入顶点G.vertices[4].data: v4
输入第1条边的两个顶点：v0 v1
输入第2条边的两个顶点：v0 v2
输入第3条边的两个顶点：v2 v3
输入第4条边的两个顶点：v3 v0
输入第5条边的两个顶点：v3 v4
深度优先搜索遍历序列:  v0 v2 v3 v4 v1
```

(b) 测试用例2

```
请输入图的种类：0表示无向图UDG，1表示有向图DG，2表示无向网UDN，3表示有向网DN
2
输入顶点数G.vexnum: 6
输入边数G.arcnum: 6
输入顶点G.vertices[0].data: A
输入顶点G.vertices[1].data: B
输入顶点G.vertices[2].data: C
输入顶点G.vertices[3].data: D
输入顶点G.vertices[4].data: E
输入顶点G.vertices[5].data: F
输入第1条边vi、vj和权值w : A B 10
输入第2条边vi、vj和权值w : A C 2
输入第3条边vi、vj和权值w : B D 7
输入第4条边vi、vj和权值w : B F 5
输入第5条边vi、vj和权值w : D E 2
输入第6条边vi、vj和权值w : E F 5
深度优先搜索遍历序列:  A C B F E D
```

(c) 测试用例3

```
请输入图的种类：0表示无向图UDG，1表示有向图DG，2表示无向网UDN，3表示有向网DN
3
输入顶点数G.vexnum: 4
输入边数G.arcnum: 5
输入顶点G.vertices[0].data: A
输入顶点G.vertices[1].data: B
输入顶点G.vertices[2].data: C
输入顶点G.vertices[3].data: D
输入第1条边vi、vj和权值w : A B 1
输入第2条边vi、vj和权值w : A C 4
输入第3条边vi、vj和权值w : C D 7
输入第4条边vi、vj和权值w : D A 6
输入第5条边vi、vj和权值w : D C 4
深度优先搜索遍历序列:  A C D B
```

(d) 测试用例4

图 5-7　程序部分测试用例的运行结果

7. 延伸思考

(1) 给定一个图,对于固定的存储结构,按给定的深度优先搜索遍历算法得到搜索结果是否唯一?

(2) 如果在图的邻接矩阵存储结构上实现图的深度优先搜索遍历,其算法该如何编写?同时比较分析在两种不同的存储结构上实现这同一种遍历,其时间性能的优劣。

(3) 如何将上述深度优先搜索遍历算法 DFS(G, v)改为非递归算法?

5.2 进阶实践

5.2.1 红色警报问题

1. 实践目的

(1) 能够正确分析红色警报问题要解决的关键问题及其解决思路。

(2) 能够根据红色警报问题的操作特点选择适当的存储结构。

(3) 能够运用图的相关基本操作的实现方法设计红色警报问题的关键操作算法。

(4) 能够编写程序模拟红色警报问题的实现,并验证其正确性。

(5) 能够对实践结果的性能进行辩证分析或优化。

2. 实践内容

战争中保持一个国家各个城市间的连通性是非常重要的。本实践要求编写一个报警程序,当失去一个城市导致国家被分裂为多个无法连通的区域时,就发出红色警报。若该国家本来就不完全连通,是分裂的 k 个区域,失去一个城市并不改变其他城市之间的连通性,则不发出警报。

如图 5-8(a)所示,一个国家初始有两个连通区域。当失去 1 号城市时,该国家仍然有两个连通区域,如图 5-8(b)所示,并未影响连通性,无须发出警报;若此时再失去 2 号城市,使得该国家剩下一个区域,如图 5-8(c)所示,也无须发出警报;但若再失去 0 号城市,使得该国家分裂成两个区域,如图 5-8(d)所示,其中 3 号城市与 4 号城市之间又不连通,此时需要发出警报。

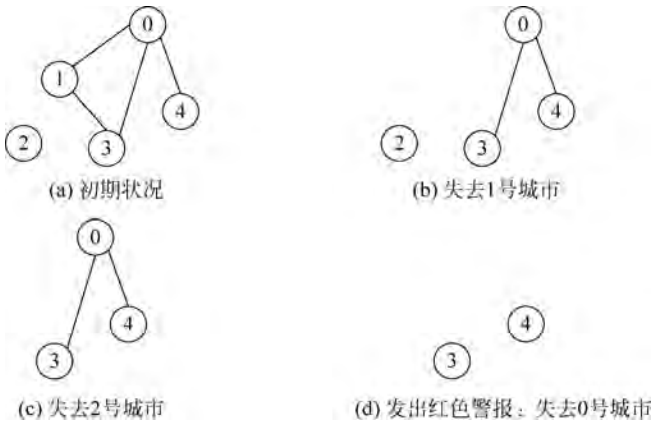


图 5-8 红色警报示意图

3. 实践要求

(1) 为使算法具有普适性,各城市之间的地图信息可由用户自主设定。

(2) 根据问题的处理对象及其操作特性,选择恰当的存储结构。

(3) 抽象出红色警报问题中的关键性操作模块,并给出其接口描述及其实现算法。

(4) 输入输出说明:本实践要求输入的内容分为两个部分:第一个部分构建一个国家城市之间的地图信息对应的无向图;第二个部分先是以一行输入失去城市的数量,再以一

行输入失去城市编号序列,之间用空格隔开。本实践的输出内容由输入的失去城市数量和城市序号值决定,按失去城市的序号依次判断是否需要报警,每个城市一行,如果无须报警则仅显示"City ×× is lost.",如果需要报警,则显示"Red Alert: City ×× is lost!"。最后,如果失去了所有的城市,则显示"Game Over."。

4. 解决方案

1) 数据结构

本实践关键是要对国家各城市之间的连通性做出判断,用图的顶点表示城市,图的边表示城市间的道路,那么国家各城市之间的交通状况可以用无向图进行表示,即用无向图表示国家城市间的交通地图信息。由于现代城市之间的交通往往比较发达,从而决定城市间的交通地图是一个较为稠密的无向图,为此,本实践拟采用邻接矩阵存储表示,其存储结构描述如下:

```
typedef char * VertexType;                                  //VertexType 表示城市名
typedef int VRType;                  //VRType 表示城市间是否有道路相通,1 表示"有",0 表示"无"
typedef struct                                              //城市间的交通地图结构信息
{    VertexType  vexs[MAX_VERTEX_NUM];                      //各城市名信息
     VRType  arcs[MAX_VERTEX_NUM][MAX_VERTEX_NUM];          //两个城市间是否有路相通的信息
     int  vexnum, arcnum;                                   //城市数和道路数
  } MGraph;
```

2) 关键操作实现要点

由前面的分析可知,本实践中失去一个城市意味着在无向图中删除一个顶点以及与该顶点所有相关联的边,如果这一操作使得图的连通分量变多了,则说明需要报警。因此本实践最核心的操作是在给定的地图中,确定连通分量的数量。

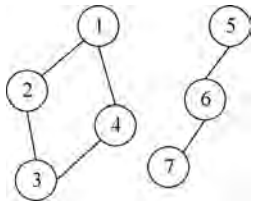


图 5-9 由两个连通分量组成的非连通图

当无向图是非连通图时,从图中的一个顶点出发遍历图,不能访问该图的所有顶点,而只能访问包含该顶点的连通分量中的所有顶点。因此,从无向图的每个连通分量中的一个顶点出发遍历图,则可求得无向图的所有连通分量。例如,图 5-9 是由两个连通分量组成的非连通图。

因此,我们可以通过将无向图的遍历算法进行改造,如果无向图是个连通图,只从一个顶点出发就能遍历全图;如果无向图是非连通的,从一个顶点完成遍历后,图中还存在未访问到的点,需要再次从未访问到的点出发进行遍历。

根据上述分析可知:本实践中可抽象出 3 个关键操作模块,其实现要点说明如下。

(1) 广度或深度优先搜索遍历连通分量。

只是为了说明问题,在本实践的实现中选择了广度优先搜索遍历。该遍历的实现方法可参见 5.1.1 节中解决方案中的操作(10)BFS(G, v),此处不再赘述。

(2) 求图的连通分量数。

上面的操作仅能遍历起始顶点所在的连通分量,如果图为非连通图,则需要另选图中一个未曾被访问的顶点作为起始点,重复上述过程,直到图中所有顶点都被访问到为止。这与图的遍历操作方法相同。但为了求图的连通分量数,可以在此操作中引进一计数器,初值为 0,每次选择图中一个未曾被访问的顶点作为起始点时,计数器值增 1。最后计数器的终值

就为该无向图连通分量的数量。

最后，还有一个问题需要加以说明，本实践中失去一个城市意味着在无向图中删除该顶点以及与该顶点所有相关联的边，而在计算机中实现时，并不会真正删去该城市对应的顶点，只是删除了与该顶点相关联的所有边，这样，该顶点成为一个孤立点，自身就成为一个连通分量，因此在失去城市判断是否需要报警时，判断标准为失去城市后连通分量数量大于原来连通分量数量加 1。

3）关键操作接口描述

```
void BFS(MGraph G,int v)
//按广度优先搜索遍历连通分量。使用辅助队列 Q 和访问标志数组 visited
int CC_BFSTraverse (MGraph G)
//按广度优先搜索遍历图 G,计数连通分量的个数,并返回其值
```

4）关键操作算法参考

（1）广度优先搜索遍历连通分量的算法。

```
void BFS(MGraph G, int v)
//按广度优先遍历连通图 G。使用辅助队列 Q 和访问标志数组 visited
{   int u, w;
    LinkQueue Q;
    InitQueue(&Q);                          //置空的辅助队列 Q
    visited[v] = 1;
    EnQueue(&Q, v);                         //v 入队
    while (!QueueEmpty(Q))                  //队列不空
    {
        DeQueue(&Q, &u);                    // 队头元素出队并置为 u
        for (w = FirstAdjVex(G, u); w >= 0; w = NextAdjVex(G, u, w))
            if (!visited[w])                // w 为 u 的尚未访问的邻接顶点
            {
                visited[w] = 1;
                EnQueue(&Q, w);             // w 入队
            }
    }
}
```

（2）求图的连通分量数（广度优先搜索遍历图）的算法。

```
int CC_BFSTraverse(MGraph G)
//按广度优先遍历图 G,记数连通分量的个数
{   int v, count = 0;                       //count 用于记数连通分量的个数
    LinkQueue Q;
    for (v = 0; v < G.vexnum; ++v)
        visited[v] = 0;                     // 置初值
    InitQueue(&Q);                          // 置空的辅助队列 Q
    for (v = 0; v < G.vexnum; v++)          // 如果是连通图,只 v = 0 就遍历全图
        if (!visited[v])                    // v 尚未访问
        {
            count++;
```

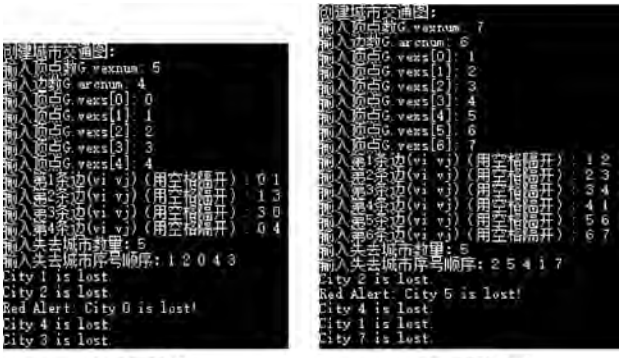
```
                BFS(G, v);
            }
        return count;
    }
```


5. 程序代码参考

扫码查看：5-2-1.cpp

6. 运行结果参考

程序的运行结果如图 5-10 所示。



(a) 运行结果1　　(b) 运行结果2

图 5-10　程序运行结果参考

7. 延伸思考

(1) 本实践如果采用在深度优先搜索遍历的操作过程中求得连通分量该如何实现？同时比较并分析在两种不同的优先搜索遍历的操作基础上实现同一问题求解，其时间性能的优劣如何？

(2) 如果问题改为判断攻占哪些城市会触发红色警报，那么其中关键操作的算法又该如何设计和实现呢？

5.2.2 “村村通”公路工程问题

1. 实践目的

(1) 能够深刻理解我国“村村通”公路工程的目标和任务，激发学生对新农村的向往和对祖国建设的热爱之情，增强学生投入到乡村振兴战略的自觉性。

(2) 能够根据“村村通”公路工程问题的操作特点，选择恰当的存储结构。

(3) 能够运用图的相关基本操作的实现方法设计“村村通”公路工程问题的关键操作算法。

(4) 能够编写程序模拟“村村通”公路工程问题的实现，并验证其正确性。

(5) 能够对实践结果的性能进行辩证分析和优化提升。

2. 实践背景

“晴天一身灰,雨天一身泥。”很多曾经在农村生活过的人,对于乡村公路的泥泞不堪有着刻骨铭心的记忆。行路难给广大农村地区的生产生活带来严重影响,修路一度成了农民们的最大渴盼。

“要致富,先修路”“经济发展,交通先行”,农村公路建设已经成为推进社会主义新农村建设的重要内容。便利的交通能够促进农村地区经济发展,从而增加农民的收入。基于对农村地区交通问题的深刻认识,为支持新农村建设,国家在 1998 年提出“村村通”工程,2006 年进入正式实施阶段。这项系统工程包括电力、饮用水、电话网、有线电视网、互联网等,而公路则是其中至关重要的一环。“村村通”公路工程又称“五年千亿元”工程,是指中国力争在 5 年时间实现所有村庄通沥青路或水泥路,以打破农村经济发展的交通瓶颈,解决 9 亿农民的出行难①。

2003 年以来,中央加大了对农村公路的投资力度,农村公路建设突飞猛进,截至 2019 年底,全国农村公路里程已达 420 万千米,实现具备条件的乡镇和建制村 100%通硬化路,为决战决胜脱贫攻坚,为全面建成小康社会提供了坚实的交通保障。

3. 实践内容

岭头乡位于浙江省永嘉县东北部,东邻乐清市,北界台州市黄岩区,南接西源乡、鹤盛乡,西连鲤溪乡、张溪乡,一鸡鸣三县是真实写照,有高山上的平原之称。

岭头乡从 2004 年至 2006 年,用三年时间累计投资 400 多万元,通车里程 14.1 千米,完成黄宙坑、塘堀、后村、徐洋、龙洋、钵下、富楼源等“村村通”公路建设工程。

根据岭头乡的村庄间道路的统计数据,用图 5-11 所示描绘出了各村庄间有可能建设成标准公路中的部分公路规划及其建设的预算成本。其中圆圈表示村庄名,边表示两个村庄可以建成的标准公路,边上的数据值表示公路建设的预算成本(单位:10 万元)。

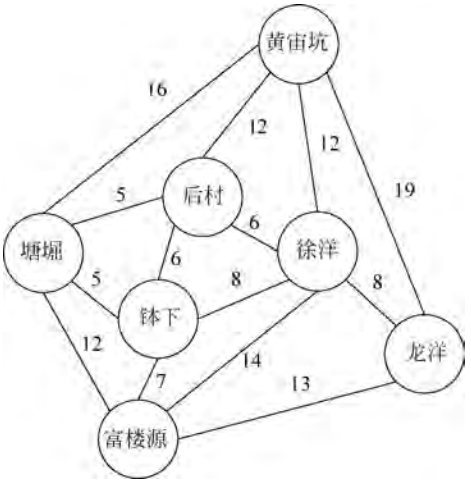


图 5-11 岭头乡各村庄部分公路的建设规划及预算成本图

① “十一五”期间农村公路逐步实现村村通[J].轮胎工业,2006,26(4):197.

编程模拟求出岭头乡的公路“村村通”的设计方案和投资成本最低的工程设计方案。

4. 实践要求

(1) 根据“村村通”公路工程问题中的处理对象及操作特性,请自主设计恰当的存储结构。

(2) 抽象出本实践的关键性操作模块,并给出其接口描述及其实现算法。

(3) 输入输出说明:先输入村庄数目 n 和可建公路数目 e(即图 5-11 中的顶点数和边数);随后输入 n 个村庄的名称;最后输入每条公路连接的两个村名及其建设的预算成本信息,并且分行输入每条公路的这 3 个值。输出的信息就是“村村通”公路工程应建的公路以及所需要的最低成本。

5. 解决方案

1) 数据结构

本实践可以用无向网来表示如图 5-11 所示的信息。由于理论上任意两个村庄之间都有可能建设道路,因此本实践涉及的“村村通”公路建设图就是一个稠密图。为此,本实践宜采用邻接矩阵存储表示,其存储结构的描述如下:

```
typedef char * VertexType;                                  //VertexType 表示村庄名
typedef int VRType;                                         //VRType 表示可能建设的公路成本
typedef struct                                              //村庄之间有可能建设的公路信息
{     VertexType  vexs[MAX_VERTEX_NUM];                     //村庄名信息
      VRType  arcs [MAX_VERTEX_NUM][MAX_VERTEX_NUM];        //可能建设的公路成本信息
      int  vexnum, arcnum;                                  //村庄数和公路数
  } MGraph;
```

我们知道,对于 n 个顶点的连通网可以建立许多不同的生成树,每一棵生成树都是一个最小的连通子网。从实践内容及要求可知,问题的求解结果就是要由如图 5-11 所示的连通网来确定一棵生成树,并使其总的代价最低。显然,本实践要解决的关键问题转化为构造已经连通网的最小代价生成树(minimum cost spanning tree)(简称为最小生成树)的问题。一棵生成树的代价就是树上各边的投资成本之和。

常见的构造最小生成树的算法有普里姆(Prim)算法和克鲁斯卡尔(Kruskal)算法。其中普里姆算法更适用于稠密图,而克鲁斯卡尔算法更适用于稀疏图。根据前面的分析,公路建设图可能是一个接近于完全图的稠密图,为此本实践宜采用普里姆算法来求最小生成树。

而在应用普里姆算法对当前生成树进行扩展时,每次要对最小边权(投资成本)的那个点继续进行扩展,因此需要一个辅助数组记录从生成树顶点集 U 到图中剩余顶点集 V-U 的代价最小的边。数组的存储结构描述如下:

```
struct Record{
  VertexType adjvex;
  VRType      lowcost;
}closedge[MAX_VERTEX_NUM];     //其中 MAX_VERTEX_NUM 为约定的最大顶点(城市)数
```

2) 关键操作实现要点

普里姆算法的基本思想是从任意一顶点 v_0 开始选择其最近的邻点 v_1 构成树 T_1,再连接与 T_1 最近的邻点 v_2 构成树 T_2,如此重复直到所有顶点均在所构成树中为止。因此,这里涉及两个重要操作:

(1) 求出与当前生成树最近的邻点。

从记录顶点集 U 到 V-U 的代价最小的边的辅助数组中查找最小值,对应的顶点即为所求与生成树最近的邻点。

(2) 用普里姆算法构造网 G 的最小生成树 T。

从无向连通网 G 的某一顶点 v_0 出发,选择与它关联的具有最小权值的边(v_0,v_1),将其顶点加入到生成树顶点集 U 中。以后每一步从一个顶点在 U 中而另一个顶点不在 U 中的各条边中选择权值最小的边(u,v),把它的顶点加入到集合 U 中。如此继续下去,直到网络中的所有顶点都加入到生成树顶点集 U 中为止。

3) 关键操作接口描述

```
int Minimum(Record closedge[])
//求出与当前生成树最近的邻点,其中 closedge 是一个辅助数组,记录 U 到 V-U 具有最小代价的边
int MiniSpanTree_PRIM(MGraph G, VertexType u)
//用普里姆算法从第 u 个顶点出发构造网 G 的最小生成树 T,输出 T 的各条边
```

4) 关键操作算法参考

(1) 求出与生成树最近的邻点的算法。

```
int Minimum(Record closedge[])
//求出与当前生成树最近的邻点
{   int i = 0, min, adj;
    while (!closedge[i].lowcost)
        i++;
    min = closedge[i].lowcost;              //首个不为 0 的值
    adj = i;
    i++;
    for (; i < MAX_VERTEX_NUM; i++)
        if (closedge[i].lowcost > 0 && closedge[i].lowcost < min)
        {
            min = closedge[i].lowcost;
            adj = i;
        }
    return adj;
}
```

(2) 普里姆算法。

```
int MiniSpanTree_PRIM(MGraph G, VertexType u)
//用普里姆算法从第 u 个顶点出发构造网 G 的最小生成树 T,输出 T 的各条边
{   int i, j, k, weight = 0;
    k = LocateVex(G, u);
    closedge[k].lowcost = 0;                    // 初始,U={u}
    for (i = 0; i < G.vexnum; i++)              // 辅助数组初始化
        if (i != k)
        {
            closedge[i].adjvex = u;
```

```
                closedge[i].lowcost = G.arcs[k][i];
            }
        for (i = 1; i < G.vexnum; i++)                    //选择其余 N-1 个顶点
        {
            k = minimum(closedge);                        //求出加入生成树的下一个顶点 k
            weight += closedge[k].lowcost;
            //此时 closedge[k].lowcost = MIN{closedge[vi ].lowcost|closedge[vi].lowcost > 0,
Vi∈V-U}
            printf("(%s,%s) ", closedge[k].adjvex, G.vexs[k]);
                                                          //输出生成树上一条边的两个顶点
            closedge[k].lowcost = 0;                      // 第 k 顶点并入 U 集
            for (j = 0; j < G.vexnum; j++)                // 修改其他顶点的最小边
                if (G.arcs[k][j] < closedge[j].lowcost)   //新顶点并入 U 后重新选择最小边
                {
                    closedge[j].adjvex = G.vexs[k];
                    closedge[j].lowcost = G.arcs[k][j];   //{adjvex, lowcost}
                }
        }
        return weight;
}
```

6. 程序代码参考

扫码查看：5-2-2.cpp

7. 运行结果参考

程序运行的结果如图 5-12 所示。

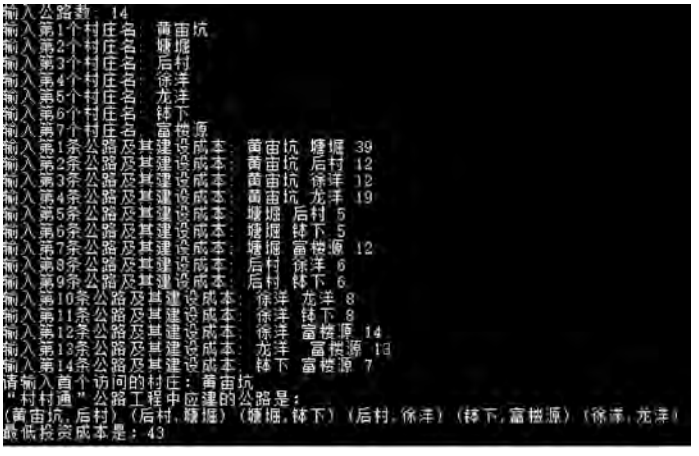


图 5-12　程序运行结果参考

8. 延伸思考

(1) 上面实现的算法要保证图是连通图，如果输入的数据不足以保证各村庄之间的连通性，则输出－1，表示需要建设更多条公路，那么相关核心算法应该如何修改？

(2) 普里姆算法比较适合于稠密图，如果在稀疏图上构造最小生成树该采用什么算法？其实现算法又该如何编写？

9. 结束语

“村村通”公路工程给我国农村地区的发展带来了契机和希望。主要有以下两方面的作用：其一，在经济发展方面，村道公路拉通了村与村、居民与居民之间的距离。条件具备的村可实行规模化大生产，集合优势资源、集中农村劳动力，更充分地利用人力、财力和物力。投入成本相应降低、产量增加，同时公路的便捷使得销售渠道大开，大大节省了从资源转化为利润的时间；其二，在人民生活方面，交通的便捷、收入的增加使得整个乡村地区面目一新，也带动了当地就学、就医、娱乐休闲及健身配套设施的兴起和完善，使得整个乡村地区的幸福指数显著提高。

5.3 拓展实践

5.3.1 最大食物链计数问题

1. 实践目的

(1) 能够正确分析最大食物链计数问题要解决的关键问题及其解决思路。

(2) 能够根据最大食物链计数问题的特点选择适当的存储结构。

(3) 能够运用图的相关基本操作的实现方法设计最大食物链计数问题中关键操作的算法。

(4) 能够编写程序验证最大食物链计数问题的正确性。

(5) 能够对实践结果的性能进行辩证分析或优化提升。

2. 实践内容

这里的“最大食物链”指的是生物学意义上的食物链，如图 5-13 所示，正方形顶点表示的是不会捕食其他生物的生产者，五边形顶点表示的是不会被其他生物捕食的消费者。如果把这个食物网转换为一个图，这个食物网中的捕食关系一定是单向的，因此这个食物网可以抽象成一个有向图；同时，这个食物网中的捕食关系一定要是无环的，因此考虑将这个食物网转换为一个有向无环图(Directed Acyclic Graph)，简称为 DAG。而“最大食物链”定义就是一条从入度为 0 的顶点开始到出度为 0 的顶点结束的路径。

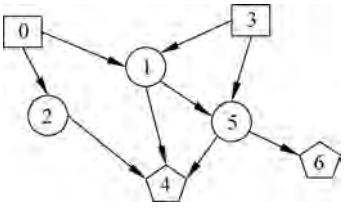


图 5-13 食物网

编程实现求解出如图 5-13 所示的食物网中“最大食物链”的数目。

3. 实践要求

(1) 根据“最大食物链”计数问题中数据的逻辑结构特征，自主设计恰当的数据存储结构。

(2) 抽象出本实践问题的关键操作功能模块,并给出接口描述及其实现算法。

(3) 输入要求:第 1 行输入顶点的数目;第 2 行输入弧的数目;再分行输入各顶点的值;最后分行输入各条弧的信息。

(4) 输出要求:输出求得的最大食物链的数目。

4. 解决思路

1) 数据存储结构的设计要点提示

本实践的处理对象是食物网,而食物网可以用有向无环图表示,其中图的顶点表示生物,图的弧表示生物之间的捕食关系,弧的始点为被捕食者,终点为捕食者,由于生物的食物具有针对性,因此本实践涉及的食物网一般是一个稀疏图。那么,基于以上分析,你觉得本实践应该采用何种存储结构更为合适呢?

2) 关键操作的实现要点提示

由于"最大食物链"就是一条从入度为 0 的顶点开始到出度为 0 的顶点结束的路径。对于如图 5-13 所示食物网,正方形所示的顶点入度为 0,五边形所示的顶点出度为 0,其中 0→1→5→6 就是一条"最大食物链"。那么这个食物网中到底有多少条"最大食物链"呢?从图中可以看出,我们需要将所有从 0 和 3 开始,最终到达 4 和 6 的路径数统计出来即可。比如,能到达 4 号顶点的边有 3 条,分别是从 1→4、2→4、5→4,而能到达 1 的边有 2 条,分别是 0→1 和 3→1,到达 2 的边有 1 条,是 0→2,能到达 5 的边有 2 条,是 1→5 和 3→5,因此,最终能到达 4 的路径条数有 2+1+(2+1)=6 条。

可用一个函数 $f(x)$的值表示从任意一个入度为 0 的顶点到顶点 x 的食物链计数值。注意考虑以下情况。

(1) 初始时,对于任意一个入度为 0 的点,$f(x)=1$。

(2) 对于任意一个入度非 0 的顶点,$f(x)=\sum_{\text{顶点}u\text{能到达顶点}x} f(u)$,即 $f(x)$的值等于能到达 x 的所有点的函数值之和。

(3) "最大食物链"计数为 $\sum_{\text{出度为0的顶点}x} f(x)$,即所有出度为 0 的顶点的函数值之和。

所以,求"最大食物链"的过程是一个递推过程,可以由入度为 0 的顶点开始递推,依次从图中去掉入度为 0 的顶点及其关联的边,即将关联的边终点的入度减 1,函数值累加。

图 5-14 模拟了食物链的求解过程,最后如图 5-14(f)所示的最大食物链计数为 $f(4)+f(6)=9$。这一递推顺序与图的拓扑排序是一致的。整个拓扑排序过程的关键操作包括 3 项,分别是求各个顶点的出度、求各个顶点的入度和求拓扑排序序列。以上操作的实现要点提示如下。

(1) 求各顶点出度。

要求各顶点的出度,需要遍历各顶点对应的边表,边表长度就是该顶点的出度,这就需要遍历整个邻接表。对于包含 n 个顶点和 e 条弧的有向图而言,求各个顶点出度算法的时间复杂度为 $O(n+e)$。

(2) 求各顶点入度。

要求各顶点的入度,也需要遍历整个邻接表,边表边结点的 adjVex 域的值出现的次数指示了其对应顶点的入度。这也需要遍历整个邻接表。对于包含 n 个顶点和 e 条弧的有向图而言,求各个顶点入度算法的时间复杂度为 $O(n+e)$。

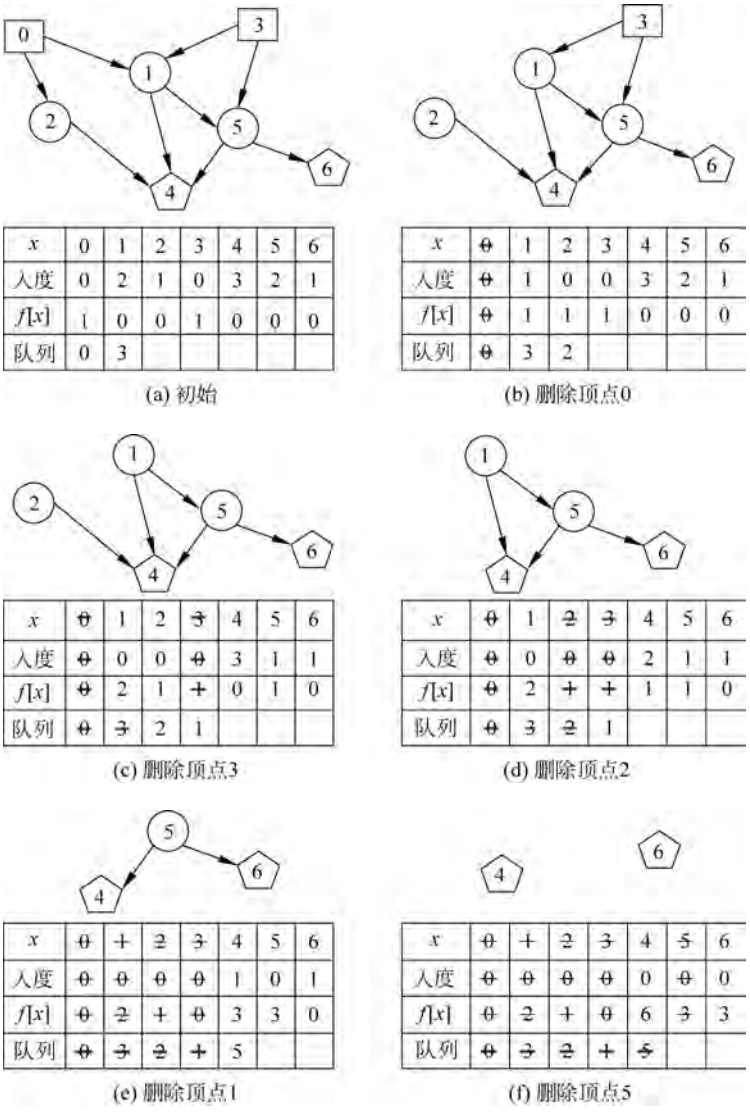


图 5-14　模拟食物链的求解过程

(3) 求拓扑排序序列。

求解拓扑排序序列的具体步骤如下。

步骤 1：在有向图中选择一个没有前驱的顶点并输出。

步骤 2：从有向图中删除该顶点以及从它出发的弧。

步骤 3：重复步骤 1 和步骤 2 直至有向图为空(即已输出所有的顶点)，或者剩余子图中不存在没有前驱的顶点。后一种情况则说明该有向图中存在有向环。

在计算机中实现该算法时，需要以“入度为零”作为“没有前驱”的量度，而“删除顶点及以它为尾的弧”的这类操作不必真正对图的存储结构执行，可用“弧头顶点的入度减 1”的办法来替代。并且为了方便查询入度为零的顶点，该算法中附设了“队列”，用于保存当前出现的入度为零的顶点。本实践最终要求解的“最大食物链”计数为所有出度为 0 的顶点的函数值之和。

5. 程序代码参考

扫码查看：5-3-1.cpp

6. 延伸思考

(1) 本实践实际是拓扑排序的一个应用问题,那么能否利用深度优先搜索遍历实现本实践？如果可以,应该怎么进行,思考一下利用了深度优先搜索遍历的什么特性。

(2) 大学中某专业的课程学习,有些课程是基础课,它们可独立于其他课程,即无先修课；有些课程则必须在某些课程学习完成以后才能开始。如果把课程作为顶点,则课程的学习安排构成一个有向图,现在要找出一个合理的课程学习流程图,以便顺利进行课程学习,那么该如何解决这个问题？

5.3.2 北斗卫星导航系统

1. 实践目的

(1) 能够加深理解我国研发“北斗卫星导航系统”的重大意义,激发学生的学习热情,引导学生把“卡脖子”清单变成“学习任务”清单,培养学生创新精神和责任担当。

(2) 能够正确分析北斗卫星导航系统中要解决的关键问题及其解决思路。

(3) 能够根据北斗卫星导航系统需实现的功能要求选择恰当的存储结构。

(4) 能够运用图的相关基本操作的实现方法设计北斗卫星导航系统的关键操作算法。

(5) 能够编写程序测试北斗卫星导航系统设计的正确性。

(6) 能够对实践结果的性能进行辩证分析和优化提升。

2. 实践背景

“司南之杓,投之于地,其柢指南。”苍茫夜空,7 颗排列成勺形的星星,被古人赋予一个诗意的名字——北斗。中国自己的卫星导航系统,就是以这个寓意光明和方向的星座——“北斗”命名,叫北斗卫星导航系统(BeiDou Navigation Satellite System,BDS)。它是中国自行研制的全球卫星导航系统,也是继 GPS、GLONASS 之后的第三个成熟的卫星导航系统[①]。

1994 年,中国在财政十分拮据的情况下,为什么要建立自己的卫星导航系统？

其直接原因,可以用两个事件说明：第一,“海湾战争”引发新军事革命。1991 年,“海湾战争”开创了以空中打击力量决胜的先例；最亮眼的是精确制导武器,美国 GPS 为精确制导提供了关键技术支持。“海湾战争”引发了一场世界性的新军事革命,GPS 定位系统成为各国关注的焦点；第二,20 世纪 90 年代末的“印巴战争”中,美国切断了两国的 GPS 信号,让双方遭遇了巨大损失,同时也给其他国家敲响了警钟。

各国政府主导的卫星定位系统,除了常见的民用领域外,核心功能依然是为军事和国防安全服务,很大程度上是出于打破完全被美国政府控制的 GPS 垄断的考虑。

北斗卫星导航系统,经历的艰难险阻实在太多,我们只能了解其“冰山一角”。最艰难险

① 凌云. 中国向全球卫星导航系统迈进第 4 颗北斗卫星升空. 国际展望,2007.

阻的是原子钟的研制。原子钟的精度，直接决定着卫星导航系统的精度。按北斗总设计师杨长风制定的目标，原子钟误差要达到 10^{-12}s，即每 10 万年只出现 1s 误差[①]。原子钟对整个工程的重要性如同人的心脏，这种核心技术别人绝不会给我们，中国只能靠自己。中国组建了中科院、航天科技、航天科工三支队伍同时攻关。经过两年拼搏，国产星载原子钟被研制出来，性能比欧洲原子钟还要好。目前，北斗卫星导航系统的原子钟 1000 万年才误差 1s。

3. 实践内容

在智能驾驶中，高精度定位技术已成为核心要素。2020 年 7 月 31 日，千寻位置网络有限公司宣布，已于近日启动业内首个高精度定位大规模路测，将在全国所有高速公路和主要城市高速路展开算法验证。这标志着北斗高精度定位量产技术将在智能驾驶领域大规模落地，为智能驾驶提供时空智能基础设施的安全保障。

根据北斗高精度定位技术绘制出了全国部分高速公路地图，全国部分主体骨干网络如图 5-15 所示。其中圆圈中是城市名，连线及连线上的数分别代表两个城市之间的公路及路程模拟数值。

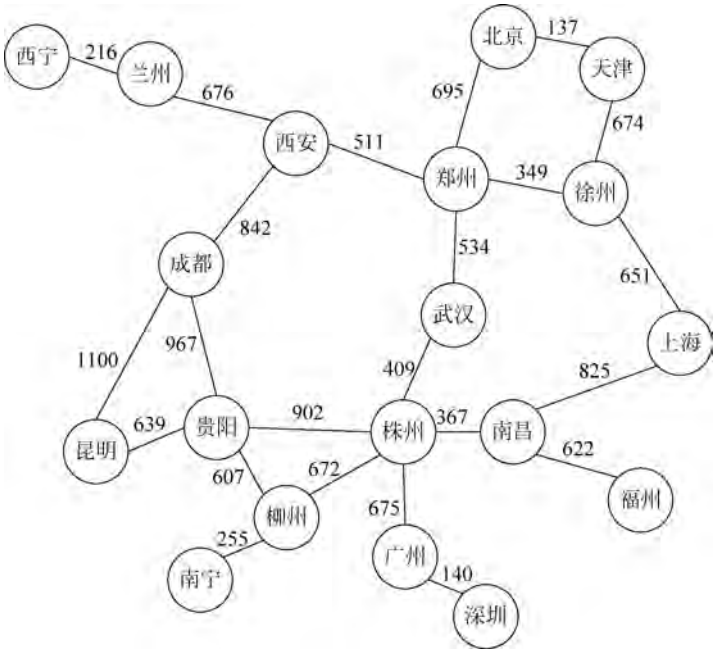


图 5-15　全国部分高速公路主体骨干网络

智能驾驶需要计算从源点到终点的最短路径长度，并显示出具体路径。根据如图 5-15 所示的信息，输入源点和终点，编程实现模拟智能驾驶。

4. 实践要求

(1) 为使算法具有普适性，要求输入的源点和终点由用户自主设定。

(2) 输入要求：输入的内容只有 1 行，即路径的源点和终点，用空格分开。

(3) 输出要求：输出的内容有 2 行，第 1 行是从源点到终点的最短路径，以箭头指向各

① 北斗简史：一文读懂国产导航的 26 年成长路. https://www.mscbsc.com/viewnews-2290211.html.

途经城市,第 2 行显示最短路径长度。

(4) 测试用例信息如表 5-4 所示。

表 5-4　测试用例

序号	输　入	输　出
1	北京 上海	最短路径是:北京→天津→徐州→上海 最短路径长度是:1462km
2	郑州 福州	最短路径是:郑州→武汉→株州→南昌→福州 最短路径长度是:1932km
3	上海 成都	最短路径是:上海→徐州→郑州→西安→成都 最短路径长度是:2353km
4	兰州 南昌	最短路径是:兰州→西安→郑州→武汉→株州→南昌 最短路径长度是:2497km

5. 解决思路

1) 数据存储结构的设计要点提示

本系统的处理对象是如图 5-16 所示的中国高速公路地图,用图的顶点表示城市,图的边表示城市之间的高速公路,边上的权值表示城市之间的距离,一般来说高速公路网应该是一个稀疏图。那么,你在本系统实现中会选择何种存储结构来表示此图?

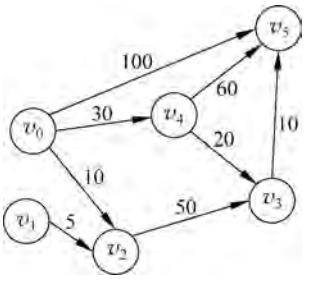


图 5-16　有向网 G

2) 关键操作的实现要点提示

本实践要解决的核心问题就是在给定的地图中,找到从源点到终点的最短路径。求图的最短路径可根据戴克斯特拉(Dijkstra)提出的一个“按最短路径长度递增的次序”产生最短路径的算法来实现。其实现要点说明如下。

给定一有向网,若从源点到某个终点存在路径,则必定存在一条最短路径。这些从某个源点到其余各顶点的最短路径彼此之间的长度不一定相等,下面分析这些最短路径的特点。

首先,在这些最短路径中,长度最短的这条路径上必定只有一条弧,且它的权值是从源点出发的所有弧上权值的最小值。例如:在如图 5-16 有向网 G 中,从源点 v_0 出发有 3 条弧,其中以弧$\{v_0,v_2\}$的权值为最小。因此,(v_0,v_2)不仅是 v_0 到 v_2 的一条最短路径,并且它是从源点到其他各个终点的最短路径中长度最短的路径。

其次,第二条长度次短的最短路径只可能有两种情况:它或者只含一条从源点出发的弧且弧上的权值大于已求得最短路径的那条弧的权值,但小于其他从源点出发的弧上的权值;或者是一条只经过已求得最短路径的顶点的路径。

以此类推,按照戴克斯特拉算法先后求得的每一条最短路径必定只有两种情况,或者是由源点直接到达终点,或者是只经过已经求得最短路径的顶点到达终点。

例如:有向网 G 中从源点 v_0 到其他终点的最短路径的过程。从这个过程中可见,类似于普里姆算法,在该算法中应保存当前已得到的从源点到各个终点的最短路径,初值为:若从源点到该顶点有弧,则存在一条路径,路径长度即为该弧上的权值。每求得一条到达某个终点 w 的最短路径,就需要检查是否存在经过这个顶点 w 的其他路径(即是否存在从顶点

w 出发到尚未求得最短路径顶点的弧），若存在，判断其长度是否比当前求得的路径长度短，若是，则修改当前路径。

该算法中需要引入一个辅助向量 $\boldsymbol{D}$，它的每个分量 $\boldsymbol{D}[i]$存放当前所找到的从源点到各个终点 v_i 的最短路径的长度。戴克斯特拉算法求最短路径的过程为：

① 令 $S=\{v\}$，其中 v 为源点，并设定 $\boldsymbol{D}[i]$的初始值为 $\boldsymbol{D}[i]=|v,v_i|$。

② 选择顶点 v_j 使得

$$\boldsymbol{D}[j]=\min_{v_i\in V-S}\{\boldsymbol{D}[i]\}$$

并将顶点 v_j 并入到集合 S 中。

③ 对集合 $V-S$ 中所有顶点 v_k，若 $\boldsymbol{D}[j]+|v_j,v_k|<\boldsymbol{D}[k]$，则修改 $\boldsymbol{D}[k]$的值为

$$\boldsymbol{D}[k]=\boldsymbol{D}[j]+|v_j,v_k|$$

④ 重复操作②、③共 $n-1$ 次，由此求得从源点到所有其他顶点的最短路径是依路径长度递增的序列。

6. 程序代码参考

扫码查看： 5-3-2.cpp

7. 延伸思考

（1）迪杰斯特拉算法能否适用于带负权的图，为什么？

（2）迪杰斯特拉算法会找出从源点到所有其他顶点的最短路径，但在实际应用中，一般只需找出从源点到某一终点的最短路径，则此情况下，迪杰斯特拉算法的时间复杂度又是如何？

8. 结束语

“河汉纵且横，北斗横复直。”自古以来，北斗如天河中的一座灯塔，指引着人类前行的方向。从京张高铁应用北斗系统实现自动驾驶，到 C919 大飞机通过北斗短报文平台实现全程位置追踪，这一切应用的起点，都始于北斗三号全球定位系统的全面建成。该系统由 30 颗北斗卫星组成，其中 20 颗卫星的导航分系统、天线分系统、星间链路子系统等全部有效载荷，均由中国航天科技集团五院西安分院北斗三号研制团队提供。

这支研制团队共 120 人，其中 35 岁以下青年占比 89.2%。2021 年 4 月 11 日，这支团队被授予“中国青年五四奖章集体”称号。在与北斗卫星相伴的日日夜夜里，这群年轻人把青春芳华融入祖国的航天事业，用青春热血点亮星空[①]。正在加速融入世界的“北斗”，未来将以崭新的姿态、更强的能力和更好的服务，服务全球，造福人类。

追求卓越、注重学习，把好核心竞争“定盘星”。不驰于空想，不骛于虚声。当今社会，最大的挑战是能力的挑战，作为一名大学生，要以空杯心态，终身学习，涵养科技报国的情怀，增强国家的道路自信，把“卡脖子”清单变成科研任务清单，早日摆脱祖国关键领域核心技术受制于人的局面。

① 这支北斗国家队 35 岁青年占 89.2%. 中国青年报. http://qnsx.cyol.com/html/2021-07/17/uw.D110000qusx_20210617_1-04.htm.