

《数据结构》上机报告

2018 年 11 月 22 日

姓名：刘思源 学号：1651390 班级：电子三班 得分：_____

| | |
|--------|--|
| 实验题目 | 图 |
| 问题描述 | 图是一种描述多对多关系的数据结构，图中的数据元素称作顶点，具有关系的两个顶点形成的一个二元组称作边或弧，顶点的集合 V 和关系的集合 R 构成了图，记作 $G=(V,R)$ 。图又分成有向图，无向图，有向网，无向网。图的常用存储结构有邻接矩阵、邻接表、十字链表、邻接多重表。 |
| 基本要求 | 1. (p1) 本题练习邻接矩阵和邻接表的创建。 2. (p2) 图的遍历算法是图的连通性问题、拓扑排序和求关键路径等算法的基础。遍历图的路径有深度优先搜索 dfs 和广度优先搜索 bfs。 本题给定一个无向图，用 dfs 和 bfs 找出图的所有连通子集。所有顶点用 0 到 n-1 表示，存储结构采用邻接矩阵表示。假设搜索时总是从编号最小的顶点出发，按编号递增的顺序访问邻接点。 3. (p3) 本题给定一个无向图，用邻接表作存储结构，用 dfs 和 bfs 找出图的所有连通子集。 所有顶点用 0 到 n-1 表示，假设邻接表采用头插法建立边表，非连通子集的搜索顺序为按照编号从小到大排列。 |
| | 已完成基本内容（序号）：1, 2, 3 |
| 选做要求 | |
| | 已完成选做内容（序号） |
| 数据结构设计 | <pre> typedef int Status; typedef char ElemType; typedef struct LNode{ ElemType data; int num; int weight; LNode *next; }VextexNode, *AdjList; typedef struct MGraph { char vexs[MAXSIZE]; int arcs[MAXSIZE][MAXSIZE]; int vexnum, arcnum; </pre> |

| | |
|----------|---|
| | <pre>AdjList List[MAXSIZE]; }MGraph;</pre> |
| 功能(函数)说明 | <p>1、 初始化图</p> <p>对图进行初始化时，由于仍未规定产生怎样的图，所以对图的两种储存方式：邻接表和邻接矩阵进行初始化操作。首先将输入图的顶点数和边数，其次根据顶点数和边数，将邻接矩阵中可能遇到的元素均置为 0。同时，邻接表采用链表的形式，这里初始化一个有着顶点数目个数元素的指针数组，每一个数组内部储存着当前元素的编号，并将指针置为 NULL。</p> <pre>Status InitMGraph(MGraph &G) { int i, j; cin >> G.vexnum >> G.arcnum; for (i = 0; i < G.vexnum; i++) { cin >> G.vexs[i]; G.List[i] = new VextexNode(); G.List[i]->data = G.vexs[i]; G.List[i]->next= NULL; } for (i = 0; i < G.vexnum; i++) { for (j = 0; j < G.vexnum; j++) { G.arcs[i][j] = EMPTY; } } return 0; }</pre> <p>2、 寻找元素位置</p> <p>寻址到当前的元素位置，输入该元素的值，在初始化时生成的字符数组内寻找该值，返回它的行或列的位置。</p> <pre>int LocateVex(MGraph G, char u) { for (int i = 0; i < G.vexnum; i++) if (u==G.vexs[i]) { return i; } }</pre> |

```
    }  
    return -1;  
}
```

3、创建邻接表

创建邻接表的过程实际上是不断将链表增长的过程。首先根据输入的元素，定位其在指针数组对应的位置，接着在该指针指向的链表的尾端继续申请空间，将新的元素加入。

但这样做存在的问题是，链表内的元素很有可能无序的，最好使用头插法来创建链表。

```
void CreatAdjList(MGraph &G, char v1, char v2)  
{  
    int i, j;  
    i = LocateVex(G, v1);  
    j = LocateVex(G, v2);  
  
    AdjList p1;  
    p1 = G.List[i];  
    while (p1->next!=NULL) {  
        p1=p1->next;  
    }  
    p1->next = new VextexNode();  
    p1 = p1->next;  
    p1->data = j;  
    p1->num = j;  
    p1->weight = G.arcs[i][j];  
    p1->next = NULL;  
  
    return;  
}
```

```
void CreatAdjList(MGraph &G, ElemType i, ElemType j)  
{  
    AdjList p1;  
    p1 = G.List[i];  
    if (p1->next!=NULL) {  
        while (p1->next != NULL) {  
            if (j > p1->next->data) {  
                AdjList p2=p1->next;  
                p1->next = new VextexNode();  
                p1 = p1->next;  
                p1->data = j;  
                p1->num = j;  
                p1->next = p2;  
            }  
            p1=p1->next;  
        }  
    }  
    p1->next = new VextexNode();  
    p1 = p1->next;  
    p1->data = j;  
    p1->num = j;  
    p1->weight = G.arcs[i][j];  
    p1->next = NULL;  
  
    return;  
}
```

```

        }
        p1 = p1->next;
    }
}

p1->next = new VextexNode();
p1 = p1->next;
p1->data = j;
p1->num = j;
p1->next = NULL;

return;
}

```

4、创建无向图

无向图的特点是一旦输入两个点，在两个点之间彼此都要产生新的边。所以调用两次生成邻接表的函数，在v1下申请一个，在v2下申请一次。

```

void CreateUDG(MGraph &G)
{
    char v1, v2;
    int i, j, k;

    InitMGraph(G);

    for (k = 0; k < G.arcnum; k++)
    {
        cin >> v1 >> v2;
        i = LocateVex(G, v1);
        j = LocateVex(G, v2);
        G.arcs[i][j] = G.arcs[j][i] = 1;
        CreatAdjList(G, v1, v2);
        CreatAdjList(G, v2, v1);
    }
}

```

5、创建无向网（无向加权图）

无向网的特点是一旦输入两个点，在两个点之间彼此都要产生新的边，并且边有权值。所以调用两次生成邻接表的函数，在v1下申请一个，在v2下申请一次。

```

void CreateUDN(MGraph &G)
{
    char v1, v2;
    int i, j, k;
    int w;

    InitMGraph(G);
    for (k = 0; k < G.arcnum; k++) {

```

```

        cin >> v1 >> v2 >> w;
        i = LocateVex(G, v1);
        j = LocateVex(G, v2);
        G.arcs[i][j] = G.arcs[j][i] = w;
        CreatAdjList(G, v1, v2);
        CreatAdjList(G, v2, v1);
    }
}

```

6、创建有向图

有向图的特点是一旦输入两个点，仅在两个点之间有顺序的产生新的边，并且边无权值。所以调用一次生成邻接表的函数，在v1下申请一个。

```

void CreatedG(MGraph &G)
{
    char v1, v2;
    int i, j, k;

    InitMGraph(G);

    for (k = 0; k < G.arcnum; k++)
    {
        cin >> v1 >> v2;
        i = LocateVex(G, v1);
        j = LocateVex(G, v2);
        G.arcs[i][j] = 1;
        CreatAdjList(G, v1, v2);
    }
    return;
}

```

7、创建有向网（有向加权图）

有向网的特点是一旦输入两个点，仅在两个点之间有顺序的产生新的边，并且边有权值。所以调用一次生成邻接表的函数，在v1下申请一次。

```

void CreateDN(MGraph &G)
{
    char v1, v2;
    int i, j, k;
    int w;

    InitMGraph(G);
    for (k = 0; k < G.arcnum; k++)
    {
        cin >> v1 >> v2 >> w;
    }
}

```

```

        i = LocateVex(G, v1);
        j = LocateVex(G, v2);
        G.arcs[i][j] = w;
        CreatAdjList(G, v1, v2);

```

```
    }
```

8、打印邻接矩阵

输出从第 0 位元素开始，到第 Vexnum 位的邻接矩阵内的元素，注意要使用格式化的输出。

```

void PrintMar(MGraph G)
{
    int i, j;
    for (i = 0; i < G.vexnum; i++) {
        for (j = 0; j < G.vexnum; j++) {
            cout << setw(4) << G.arcs[i][j];
        }
        cout << endl;
    }
    return;
}

```

9、打印邻接表

由于本文第一问并未使用头插法进行链表的生成，所以链表内部的数据很有可能都是无顺序数据，首先要进行数据的排序。将链表的所有数据存入一个数组中，当数组不为空时输出数组的元素。打印无向图时需要额外打印出权值。

注意输出的格式”→”是两个-。

```

void PrintList(MGraph G)
{
    int i, k, j=0;

    for (i = 0; i < G.vexnum; i++) {
        int sort[MAXSIZE] = { -1 };
        j = 0;
        AdjList p;
        p = G.List[i];
        cout << p->data << "-->";
        while (p->next!=NULL) {
            p = p->next;
            sort[j] = p->num;
            j++;
        }
        shellSort(sort, j);
        for (k = 0; k < j; k++) {

```

```

        cout << sort[k] << " ";
    }
    cout << endl;
}
return;
}
void PrintListU(MGraph G)
{
    int i,k,j=0;

    for (i = 0; i < G.vexnum; i++) {
        int sort[MAXSIZE][2];
        j = 0;
        AdjList p;
        p = G.List[i];
        cout << p->data << "-->";
        while (p->next != NULL) {
            p = p->next;
            sort[j][0] = p->num;
            sort[j][1] = p->weight;
            j++;
        }
        shellSort2(sort, j);
        for (k = 0; k < j; k++) {
            cout << sort[k][0] << ", "<< sort[k][1] << " ";
        }
        cout << endl;
    }
    return;
}

```

10、 DFS 深度搜索

深度优先搜索（缩写 DFS）有点类似广度优先搜索，也是对一个连通图进行遍历的算法。它的思想是从一个顶点 V_0 开始，沿着一条路一直走到底，如果发现不能到达目标解，那就返回到上一个节点，然后从另一条路开始走到底，这种尽量往深处走的概念即是深度优先的概念。

我们首先定义找到每一个搜索开始点的节点的方法 StartFind，使用一个全局变量 visit。如果寻找过就设置为 false，否则就设置为 true。遍历节点从小到大，如果该结点有边连接，就返回这个节点。

GonFind 方法是在第一个节点的基础上继续寻找，如果有除了第一个节点外的点与之连接的话，就返回这个值。

接下来进行 DFS，从第一个节点开始遍历，如果 visit[i] 为假，说明还未进行过遍历，可以进行 DFS 递归，直到某一个点连接的所有点都已经被遍历过了，返回到上一层，直到所有点被遍历结束。

```

int StartFind(MGraph G, int v)
{

```

```

        for (int i = 0; i < G.vexnum; i++) {
            if (G.arcs[v][i] == 1) {
                return i;
            }
        }
        return -1;
    }

int ConFind(MGraph G, int v, int w)
{
    for (int i = w + 1; i < G.vexnum; i++) {
        if (G.arcs[v][i] == 1) {
            return i;
        }
    }
    return -1;
}

bool visited[100];
bool first = true;

void DFS(MGraph G, int v)
{
    int w;
    visited[v] = true;
    if (!first) {
        cout << " ";
    }
    first = false;
    cout << v ;
    for (w = StartFind(G, v); w >= 0; w = ConFind(G, v, w)) {
        if (!visited[w]) {
            DFS(G, w);
        }
    }
}

void DFSTraverse(MGraph G)
{
    int i;
    for (i = 0; i < G.vexnum; i++) {
        visited[i] = false;
    }
}

```



```

for (i = 0; i < G.vexnum; i++) {
    if (!visited[i]) {
        cout << "{";
        DFS(G, i);
        //printf("\b"); //oj可能不通过
        cout << "}";
        first = true;
    }
}
}
}

```

11、 BFS 广度搜索

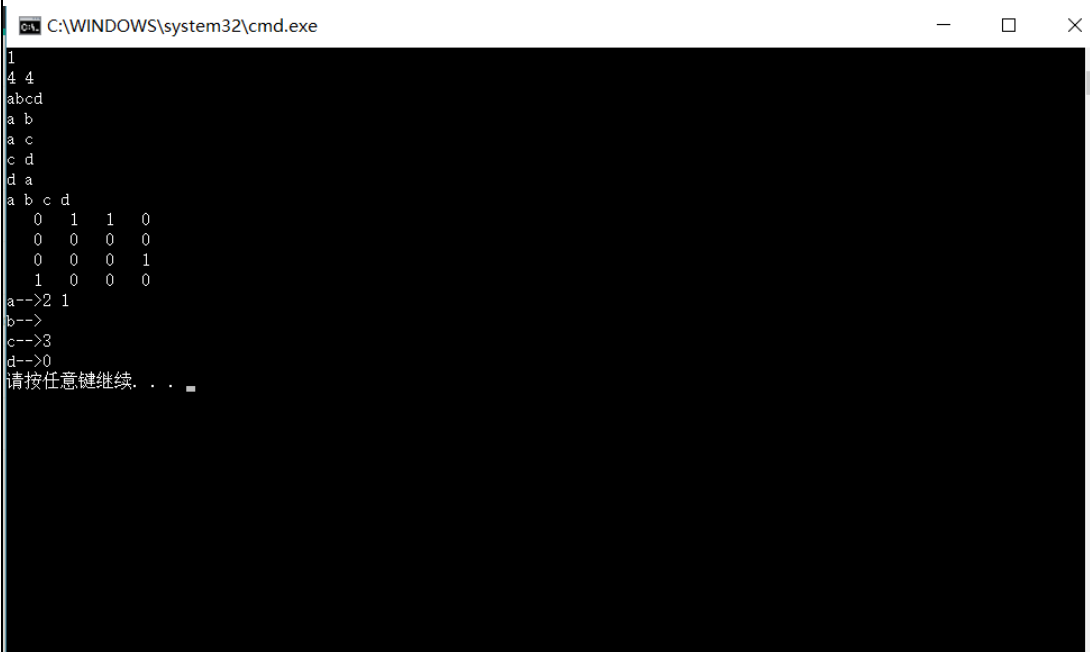
广度优先搜索类似于树的层次遍历，首先找到所有跟起始点连接的节点，再对下一层节点进行遍历。

我们使用一个队列来进行遍历结果的存储。首先是选择第一个起始点，将其入队，当队列不空时，输出队列的第一个元素，将其出队，再将所有该元素连接的节点入队。对队列中每一个元素进行重复的操作，如果该元素被遍历过，就直接出队，直到队列为空。再在剩下未遍历的元素中按照顺序选择一个元素进行上述的步骤，直到所有的元素都被遍历过。

```

void BFSTraverse(MGraph G)
{
    int i, e, w;
    first = true;
    for (i = 0; i < G.vexnum; i++) {
        visited[i] = false;
    }
    SqQueue q;
    InitQueue(q);
    for (i = 0; i < G.vexnum; i++) {
        if (!visited[i]) {
            visited[i] = true;
            EnQueue(q, i);
            cout << "{";
            while (QueueEmpty(q) == OK) {
                int v = q.base[q.front];
                DeQueue(q, e);
                if (!first) {
                    cout << " ";
                }
                first = false;
                cout << v;
                for (w = StartFind(G, v); w >= 0; w = ConFind(G, v, w)) {
                    if (!visited[w]) {
                        visited[w] = true;
                        EnQueue(q, w);
                    }
                }
            }
            cout << "}";
            first = true;
        }
    }
}

```

| | |
|------|--|
| | <pre> } } } //printf("\b"); cout << " }"; first = true; } }</pre> |
| 开发环境 | Visual studio 2017 |
| 调试分析 |  <p>The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is as follows:</p> <pre>1 4 4 abcd a b a c c d d a a b c d 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 a-->2 1 b--> c-->3 d-->0 请按任意键继续. . .</pre> |

| | |
|------|---|
| |  |
| 心得体会 | <p>1、 print("\b");不通过 OJ</p> <p>acm/oj 判断答案的原理，它会将标准输出重定向到文件，然后比较文件是否相同。比如下面的程序：</p> <pre>#include <stdio.h> int main() { printf("1+1=2 \b"); }</pre> <p>重定向到文件，会发现\b 是作为一个字符输出到文件里面的。比如，上面的代码重定向后，查看文件的 16 进制是下面的情况</p> <pre>312b 313d 3220 080d 0a 1 + 1 = 2 \s \b \r \n</pre> |

| | |
|--|--|
| | <p>20 就是空白，08 就是 \b。</p> <p>2、 DFS 与 BFS 对比</p> <p>我们假设一个节点衍生出来的相邻节点平均的个数是 N 个，那么当起点开始搜索的时候，队列有一个节点，当起点拿出来后，把它相邻的节点放进去，那么队列就有 N 个节点，当下一层的搜索中再加入元素到队列的时候，节点数达到了 N^2，你可以想想，一旦 N 是一个比较大的数的时候，这个树的层次又比较深，那这个队列就得需要很大的内存空间了。</p> <p>于是广度优先搜索的缺点出来了：在树的层次较深&子节点数较多的情况下，消耗内存十分严重。广度优先搜索适用于节点的子节点数量不多，并且树的层次不会太深的情况。</p> <p>那么深度优先就可以克服这个缺点，因为每次搜的过程，每一层只需维护一个节点。但回过头想想，广度优先能够找到最短路径，那深度优先能否找到呢？深度优先的方法是一条路走到黑，那显然无法知道这条路是不是最短的，所以你还得继续走别的路去判断是否是最短路？</p> <p>于是深度优先搜索的缺点也出来了：难以寻找最优解，仅仅只能寻找有解。其优点就是内存消耗小，克服了刚刚说的广度优先搜索的缺点。</p> <p>3、 在深度/广度搜索的过程中，其实相邻节点的加入如果是有一定策略的话，对算法的效率是有很大影响的。</p> |
|--|--|