实验报告

实验题目：图及其应用

计算机学院3班，雷雨轩，PB18111791

完成日期：2019年12月15日

实验目的

熟练掌握图的存储表示特征，各类图的创建、遍历方法以及基于遍历的算法应用。

实验要求

基本要求

- 从文件读取输入数据。
- 给定无向图G，默认边权为1，完成两个搜索算法的应用
- **DFS的应用**

  参考教材P177-178,算法7.10和7.11，基于邻接矩阵的存储结构，使用非递归的深度优先搜索算法，求无向连通图中的全部关节点，并按照顶点编号升序输出。
- **BFS的应用**

  基于邻接表的存储结构，依次输出从顶点0到顶点1、2、……、n-1的最短路径和各路径中的顶点信息。

选做要求：

将输入的无向图可视化（如样例图），推荐使用 graphviz 。

输入输出样例：

- 输入的第一行是一个正整数n，表示图中的顶点数（顶点编号从0到n-1）。 之后的若干行是无序对(i, j)，表示顶点i与顶点j之间有一条边相连。

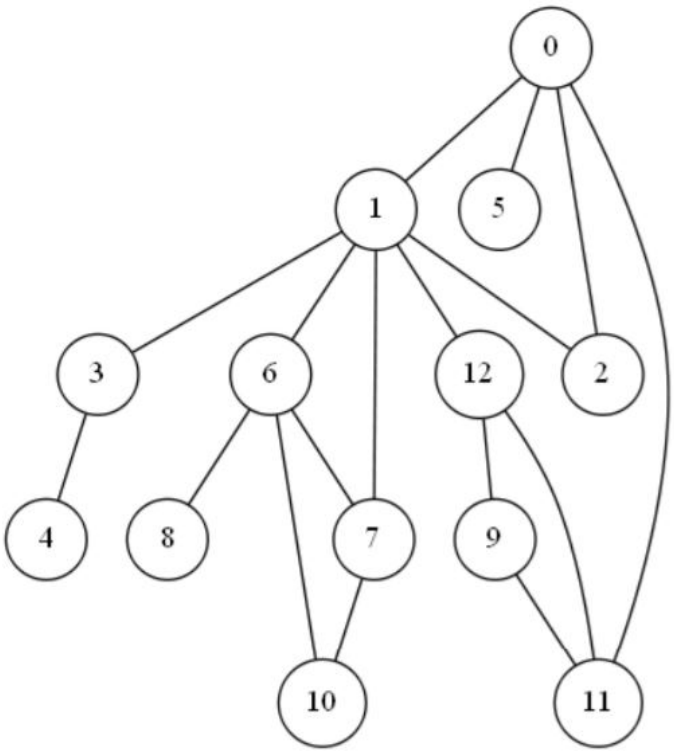
```
13
0 1
0 2
0 5
0 11
1 2
1 3
1 6
1 7
1 12
3 4
6 7
6 8
6 10
7 10
```

```
9 11
9 12
11 12
```

- 输出的第一行包括所有关节点的编号。 接下来的n-1行输出顶点0到其余每个顶点的最短路径长度以及路径信息（如果最短路径不止一条，输出任意一条即可）。

```
0 1 3 6
1 0->1
1 0->2
2 0->1->3
3 0->1->3->4
1 0->5
2 0->1->6
2 0->1->7
3 0->1->6->8
2 0->11->9
3 0->1->6->10
1 0->11
2 0->1->12
```

- 选做:



设计思路

- **DFS的应用**:
  - 使用邻接矩阵的存储结构及非递归深度优先搜索
  - 通过构造辅助栈来模拟系统栈的递归操作
  - 关节点的判断:
    - 生成树的根有两颗及以上的子树，则根顶点为关节点
    - 生成树的某个非叶子顶点v的某棵子树没有指向v的祖先(含父母节点)的回边，则v为关节点

        - 所以用数组low[]来记录每个顶点i及其子孙的回边所连接的顶点中访问次序号最小的顶点的访问次序号，并用数组min[]作为辅助方便计算
    - 用冒泡排序使得最终关节点按顶点编号升序输出，另外，若顶点i去掉后会多生成n个连通片，则算法执行过程中，会记录顶点i n次，所以需把重复点去掉

- **BFS的应用**
    - 使用邻接表的存储结构以及非递归的层次遍历方法(广度优先搜索思想)
    - 构造辅助循环队列来辅助完成广度优先搜索
        - 考虑到边权均为1，所以顶点0到顶点v2的最短路径就是顶点v1父母节点最短路径加上v1->v2弧。
        - 所以从根顶点出发，找其邻接顶点，在从其每个邻接顶点出发，找邻接顶点
    - 用PathMatrix二维数组存放路径上顶点，便于回溯，用num[]存放0到每个顶点最短路径长度

- 代码分为以下模块：

  主函数模块

  初始化栈、队列、图的两种存储结构实现，函数调用，结果输出

  栈(Sqstack)、队列(SqQueue)代码实现

  图的邻接矩阵实现

    - 数据结构定义
    - 创建图void CreateUDN(MGraph& G)
    - int FirstAdjVex(MGraph& G, int v)，给定无向图及其中一个顶点编号（从0开始），输出该顶点的第一个邻接点编号
    - int NextAdjVex(MGraph& G, int v, int w)，v是图G顶点，w是v的一个邻接顶点，返回v相对于w的下一个邻接顶点

  图的邻接表实现

    - 数据结构定义
    - 创建图void CreateUDN(ALGraph& G)
    - int FirstAdjVex(ALGraph& G, int v)，给定无向图及其中一个顶点编号（从0开始），输出该顶点的第一个邻接点编号
    - int NextAdjVex(ALGraph& G, int v, int w)，v是图G顶点，w是v的一个邻接顶点，返回v相对于w的下一个邻接顶点

  非递归深度优先搜索算法求关节点

    - void DFSArticul(MGraph& G, int v0)：从第v0个顶点出发深度优先查找关节点
    - void FindArticul(MGraph& G)：查找并输出G上全部关节点
    - void bubble_sort()：冒泡排序

  非递归广度优先搜索算法求最短路径

    - void ShorttestPath(ALGraph& G, int v0)：最短路径算法

- **选做实验**

  查询相关文档后在理解graphviz使用方法后，利用python的graphviz库来实现无向图生成

    - ```
      def separator(line, sep=[' ','\n'])
      #分词函数，将文件中每一行的两个数提取出来，返回一个列表
      ```
      ```

- 主函数:打开文件读入数据，构造图【dot = Graph('UDN')】,构造点【dot.node(name=str(k))】，并利用分词函数返回的列表构造图的边【dot.edge(list_1[0], list_1[1])】，生成视图【dot.view()】

关键代码讲解

数据结构定义

```
#define INFINITY 100             //最大值
#define MAX_VERTEX_NUM 20        //最大顶点个数
#define STACK_INIT_SIZE 100    //栈的初始空间分配量

//辅助栈
typedef struct {
    int* base;
    int* top;
    int stacksize;
}SqStack;

void InitStack(SqStack& S) {
    S.base = new int[STACK_INIT_SIZE];
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
}

void Push(SqStack& S, int e) {
    *S.top = e;
    S.top++;
}

int Pop(SqStack& S) {
    S.top--;
    int e = *S.top;
    return e;
}


typedef struct QNode {//辅助循环队列
    int* base;
    int front;
    int rear;
}SqQueue;

void InitQueue(SqQueue& Q) {
    Q.base = new int[MAX_VERTEX_NUM];
    Q.front = Q.rear = 0;
}

void EnQueue(SqQueue& Q, int e) {//入队
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear + 1) % MAX_VERTEX_NUM;
}

int DeQueue(SqQueue& Q) {
    int e;
    e = Q.base[Q.front];
```

```
    Q.front = (Q.front + 1) % MAX_VERTEX_NUM;
    return e;
}

//邻接矩阵
typedef struct ArcCell {
    int adj; //顶点关系类型，有权图代表边权
    //int* info;//该弧相关信息的指针
}ArccCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct {
    int vex[MAX_VERTEX_NUM];//顶点向量,为-1表示该顶点不存在
    AdjMatrix arcs; //邻接矩阵
    int vexnum, arcnum; //图的当前顶点数和弧数
}MGraph;

//邻接表
typedef struct ArcNode {
    int adjvex; //该弧所指向的顶点的位置
    struct ArcNode* nextarc;//指向下一条弧的指针
    int info; //该弧权值
}ArcNode;

typedef struct VNode {
    int data; //顶点信息
    ArcNode* firstarc;//指向第一条依附于该顶点的弧的指针
}VNode, AdjList[MAX_VERTEX_NUM];

typedef struct {
    AdjList vertices;
    int vexnum, arcnum; //图的当前顶点数和弧数
    //int kind; //图的种类
}ALGraph;
```

基本要求:

- 构造邻接矩阵及实现求顶点各邻接点操作

```
void CreateUDN(MGraph& G) {//采用邻接矩阵表示法，构造无向网，默认边权为1
    FILE* f;
    fopen_s(&f, "test.txt", "r");
    int i, j;
    for (i = 0; i < MAX_VERTEX_NUM; i++) {//初始化邻接矩阵
        for (j = 0; j < MAX_VERTEX_NUM; j++) {
            G.arcs[i][j].adj = INFINITY; //INFINITY表示该弧不存在
        }
    }
    fscanf_s(f, "%d", &G.vexnum);
    for (i = 0; i < G.vexnum; i++) {
        G.vex[i] = i;
    }
    for (; i < MAX_VERTEX_NUM; i++) {
        G.vex[i] = -1;  //-1表示不存在编号为i的顶点
    }
    while (fscanf_s(f, "%d %d", &i, &j) != EOF) {
        G.arcs[i][j].adj = 1;   //默认边权为1
        G.arcs[j][i].adj = 1;
```

```
            G.arcnum++;
        }
    }

    int FirstAdjVex(MGraph& G, int v) {//给定无向图及其中一个顶点编号（从0开始），输出该顶点的第一个邻接点编号，若没有邻接顶点，返回-1
        int i = 0;
        while ((i < G.vexnum) && (G.arcs[v][i].adj == INFINITY)) {
            i++;
        }
        if (i == G.vexnum) return -1;
        return i;
    }

    int NextAdjVex(MGraph& G, int v, int w) {//返回v是图G顶点，w是v的一个邻接顶点，返回v相对于w的下一个邻接顶点，若无，返回-1；
        int i = w + 1;
        while ((i < G.vexnum) && (G.arcs[v][i].adj == INFINITY)) {
            i++;
        }
        if (i == G.vexnum) return -1;
        return i;
    }
```

- 非递归的深度优先搜索算法及冒泡排序实现求关节点并升序输出

```
int count_1 = 0;
int visited[MAX_VERTEX_NUM] = { 0 };//visited[i]是深度优先搜索时访问第i个顶点的次序号
int min[MAX_VERTEX_NUM];    //作为中间量，记录当前所找到的顶点i及其子孙的回边所指向的顶点中访问次序号最小的顶点的访问次序号
int low[MAX_VERTEX_NUM];    //low[i]表示顶点i及其子孙的回边所连接的顶点中访问次序号最小的顶点的访问次序号
int result[MAX_VERTEX_NUM] = { 0 }; //记录找到的关节点的顶点编号
int num = 0;
void DFSArticul(MGraph& G, int v0) {
    //从第v0个顶点出发深度优先查找关节点
    SqStack S;
    InitStack(S);
    int v1;
    visited[v0] = min[v0] = ++count_1;  //初始化根节点
    while (1) { //用辅助栈模拟递归过程
        int flag = 0;
        for (v1 = FirstAdjVex(G, v0); v1 >= 0; v1 = NextAdjVex(G, v0, v1)) {
            if (visited[v1] == 0) {//v1未曾访问，是v0的孩子
                //mark[v0] = 1;
                Push(S, v0);
                v0 = v1;
                visited[v0] = min[v0] = ++count_1;
                flag = 1; break;
            }
            else if (visited[v1] < min[v0]) {
                min[v0] = visited[v1];  //v1已访问，是祖先节点,则更新当前找到的min[v0]
            }
        }//for
        if (flag) continue;
```

```
        low[v0] = min[v0];  //若能执行到此步骤，说明上面for循环里每一次都执行的else if
块，即当前v0已没有孩子节点，为叶子
        if (S.base == S.top) break; //栈空，表明从初始v0出发深度优先搜索执行完毕
        v1 = v0;
        v0 = Pop(S);
        if (low[v1] < min[v0]) min[v0] = low[v1];    //更新当前找到的min[v0]
        if (low[v1] >= visited[v0]) { result[num++] = v0; } //输出关节点v0
    }//while
}

void FindArticul(MGraph& G) {
    //连通图以邻接矩阵做存储结构，查找并输出G上全部关节点，按顶点编号升序输出
    for (int i = 0; i < MAX_VERTEX_NUM; i++) {
        min[i] = INFINITY;
        low[i] = INFINITY;
    }
    count_1 = 1;
    visited[0] = 1;//以0作为根节点
    int v;
    v = FirstAdjVex(G, 0);
    DFSArticul(G, v);    //从第v顶点出发深度优先查找关节点
    if (count_1 < G.vexnum) {    //生成树的根至少有两棵子树，则根为关节点
        result[num++] = 0;//输出顶点编号0；
        while ((v = NextAdjVex(G, 0, v)) >= 0) {
            if (visited[v] == 0) DFSArticul(G, v);
        }//while
    }//if
}//FindArticul

void bubble_sort() {//冒泡排序
    int i = 0;
    int j = 0;
    int temp;
    for (j = 0; j < num - 1; j++) {
        for (i = 0; i < num - 1 - j; i++) {
            if (result[i] > result[i + 1]) {
                temp = result[i];
                result[i] = result[i + 1];
                result[i + 1] = temp;
            }
        }
    }
}
```

- 构造邻接表及实现求顶点各邻接点操作

```
void CreateUDN(ALGraph& G) {
    //采用邻接表的存储结构，构造无向网G
    FILE* f;
    fopen_s(&f, "test.txt", "r");
    fscanf_s(f, "%d", &G.vexnum);
    int i, j;
    G.arcnum = 0;
    ArcNode* temp1, * temp2;
    for (i = 0; i < G.vexnum; i++) {//初始化顶点信息
        G.vertices[i].data = i;
        G.vertices[i].firstarc = NULL;
```

```
    }
    for (; i < MAX_VERTEX_NUM; i++) {
        G.vertices[i].data = -1;
        G.vertices[i].firstarc = NULL;
    }
    while (fscanf_s(f, "%d %d", &i, &j) != EOF) {//完成弧在每个单链表头结点后的插入(无向图一条弧出现两次)
        temp1 = new ArcNode;
        temp1->info = 1;
        temp1->adjvex = j;
        temp1->nextarc = G.vertices[i].firstarc;
        G.vertices[i].firstarc = temp1;
        temp2 = new ArcNode;
        temp2->info = 1;
        temp2->adjvex = i;
        temp2->nextarc = G.vertices[j].firstarc;
        G.vertices[j].firstarc = temp2;
        G.arcnum++;
    }
}

int FirstAdjVex(ALGraph& G, int v) {    //给定无向图及其中一个顶点编号（从0开始），输出该顶点的第一个邻接点编号，若没有邻接顶点，返回-1
    if (!G.vertices[v].firstarc) return -1;
    return G.vertices[v].firstarc->adjvex;
}

int NextAdjVex(ALGraph& G, int v, int w) {//返回v是图G顶点，w是v的一个邻接顶点，返回v相对于w的下一个邻接顶点，若无，返回-1；
    ArcNode* temp;
    temp = G.vertices[v].firstarc;
    while ((temp->adjvex != w)) {
        temp = temp->nextarc;
    }
    temp = temp->nextarc;
    if (temp == NULL) return -1;
    return temp->adjvex;
}
```

- 非递归广度优先搜索算法求最短路径

```
int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM] = { 0 };//P[v]存放顶点0到顶点v的最短路径上经过的顶点顺序
int final[MAX_VERTEX_NUM] = { 0 };//为1表示从顶点0到该顶点的最短路径已经找到
int num_1[MAX_VERTEX_NUM] = { 0 };//num[v]存放从顶点0到该顶点v最短路径长度

void ShorttestPath(ALGraph& G, int v0) {//最短路径算法
    int i, j;
    for (i = 0; i < G.vexnum; i++) {    //初始化最短路径
        for (j = 0; j < G.vexnum; j++) {
            PathMatrix[i][j] = -1;
        }
    }
    SqQueue Q;
    InitQueue(Q);
    final[0] = 1;   //起始点顶点0已被访问
    int v1 = 0;
```

```
    EnQueue(Q, v1);
    final[v1] = 1;
    num_1[v1]++;
    PathMatrix[0][0] = 0;
    int v2;
    while (Q.front!=Q.rear) {//考虑到边权默认为1，所以按非递归层次遍历的方式来找最短路径
        v1 = DeQueue(Q);
        for (v2 = FirstAdjVex(G, v1); v2 >= 0; v2 = NextAdjVex(G, v1, v2)) {
            if (!final[v2]) {//如果顶点v2还未找到最短路径
                for (i = 0; i < num_1[v1]; i++) {//回溯v2父母节点v1的最短路径，即v2最短路径为v1最短路径加上v1->v2边
                    PathMatrix[v2][num_1[v2]] = PathMatrix[v1][i];
                    num_1[v2]++;
                }
                PathMatrix[v2][num_1[v2]++] = v2;
                final[v2] = 1;
                EnQueue(Q,v2);
            }
        }//for
    }//while
}
```

- 主函数

```
int main() {
    //关节点
    MGraph G1;
    CreateUDN(G1);
    FindArticul(G1);
    bubble_sort();
    int temp = -1;
    for (int i = 0; i < num; i++) {
        if (temp == result[i]) continue;//考虑到若顶点i去掉后会多生成n个连通片，则算法执行过程中，会记录顶点i n次，所以需把重复点去掉
        cout << result[i] << " ";
        temp = result[i];
    }
    cout << endl;
    //最短路径
    ALGraph G2;
    CreateUDN(G2);
    ShorttestPath(G2, 0);
    int i = 0;
    int j = 0;
    for (i = 1; i < G2.vexnum; i++) {
        cout << num_1[i] - 1 << " " << PathMatrix[i][0];
        for (j = 1; j < num_1[i]; j++) {
            cout << "->" << PathMatrix[i][j];
        }
        cout << endl;
    }
    return 0;
}
```

选做要求:

```
from graphviz import Graph
def separator(line, sep=[' ','\n']):#分词函数，将文件中每一行的两个数提取出来，返回一个列表
    result = []
    i = 0
    while line[i] not in sep:        #找第一个数的切片
        i = i+1
    result.append(line[0:i])
    j = i+1
    while j < len(line) and line[j] not in sep:        #找第二个数的切片
        j = j + 1
    result.append(line[i+1:j])
    return result

dot = Graph('UDN')                   #构造无向图

with open('D:/Microsoft Visual Studio/MyProjects/wf_ep4/wf_ep4/test.txt', 'r')
as f:
    num = int(f.readline())
    for k in range(num):              #创建点
        dot.node(name=str(k))
    for line in f:                   #创建边
        list_1 = separator(line)
        dot.edge(list_1[0], list_1[1])

dot.view()
```

调试分析

时间复杂度:

- DFS:
  - 邻接矩阵构造: O(n^2)
  - int FirstAdjVex(MGraph& G, int v)： O(1)
  - int NextAdjVex(MGraph& G, int v, int w)：O(e)（e为图的边数）
  - void DFSArticul(MGraph& G, int v0): O(n * e^2):此处因为一个顶点每有一个邻接顶点，就可能会多一次while循环。这个算法也可改进，在改进NextAdjVex函数后，可改进为O(n*e)
  - void FindArticul(MGraph& G)： O(n)
  - void bubble_sort(): O(n^2)
- BFS:
  - 邻接表构造: O(n)
  - int FirstAdjVex(ALGraph& G, int v): O(1)
  - int NextAdjVex(ALGraph& G, int v, int w)：O(e)
  - void ShorttestPath(ALGraph& G, int v0)：O(n^2)

空间复杂度:

DFS及BFS：

输入数据所占空间取决于问题本身，以及栈、队列深度，最坏情况下n个顶点都要入栈或队列，所以复杂度为MAX（O(n)，O(e^2)）

遇到的问题

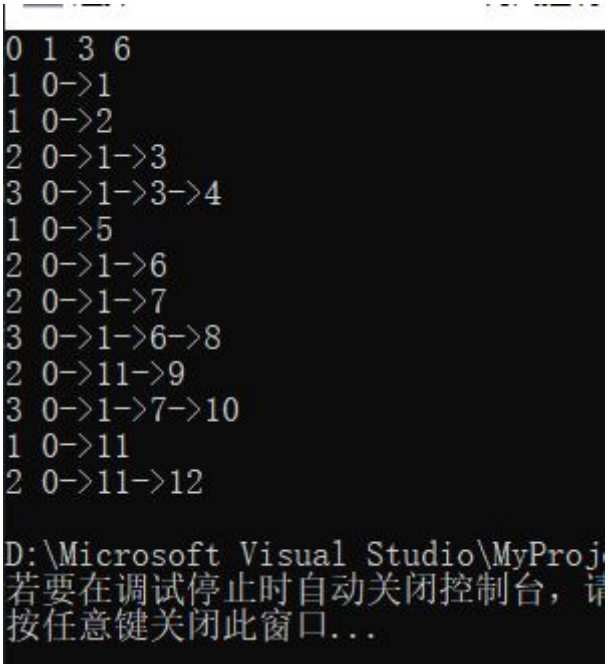
- 实验中面临的主要问题还是在于如何具体实现递归函数,在DFS实现上花的时间较多，总结发现还是自己对于递归问题转换到非递归问题的实现不够熟悉，一些判断条件，出栈入栈，循环的掌握不够到位，所以仍需在反复训练中理解熟悉此类问题。
- 关节点的判断条件也需要一定的理解，在书的帮助下自己才完成了具体算法的实现
- BFS应用相对简单，思路清晰

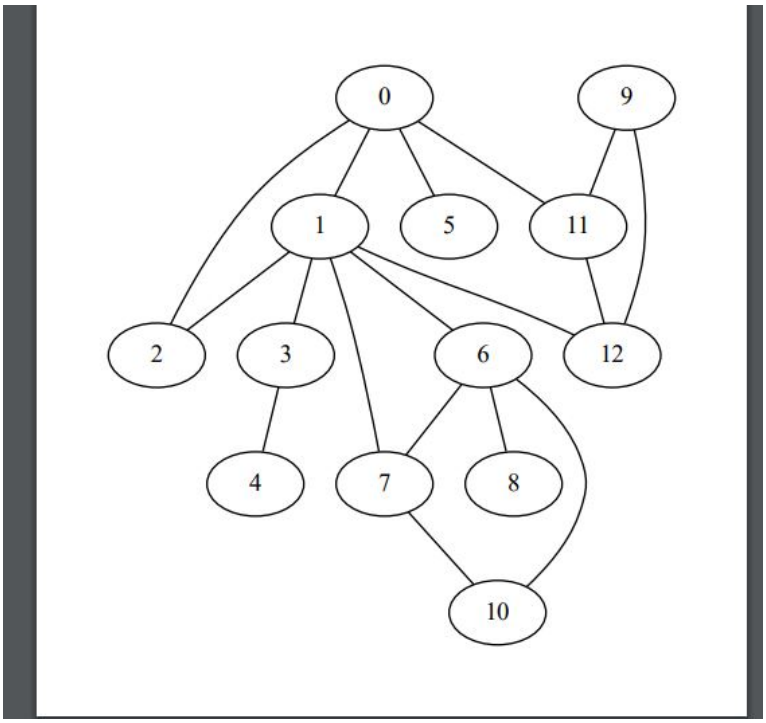
代码测试

读入相应格式的txt文件即可，注意因为在BFS问题中构建图时是按照头插法生成的边，所以最短路径不止一条时会优先输出序号之和较大的那条路径

基本及选做要求

- 样例实现

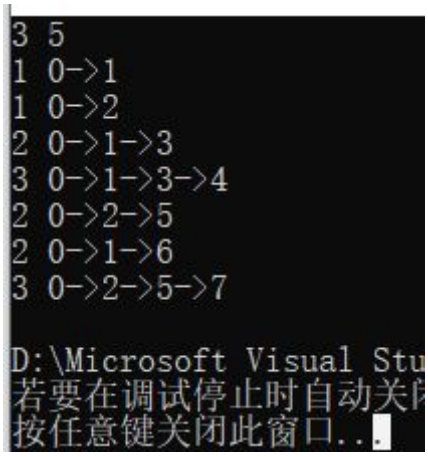


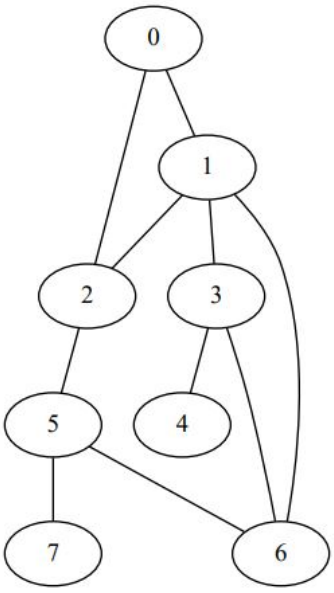


- 输入

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

输出

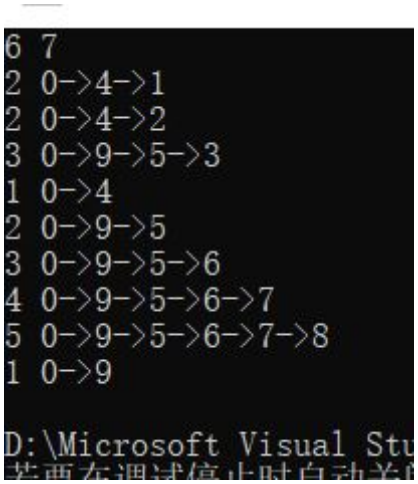


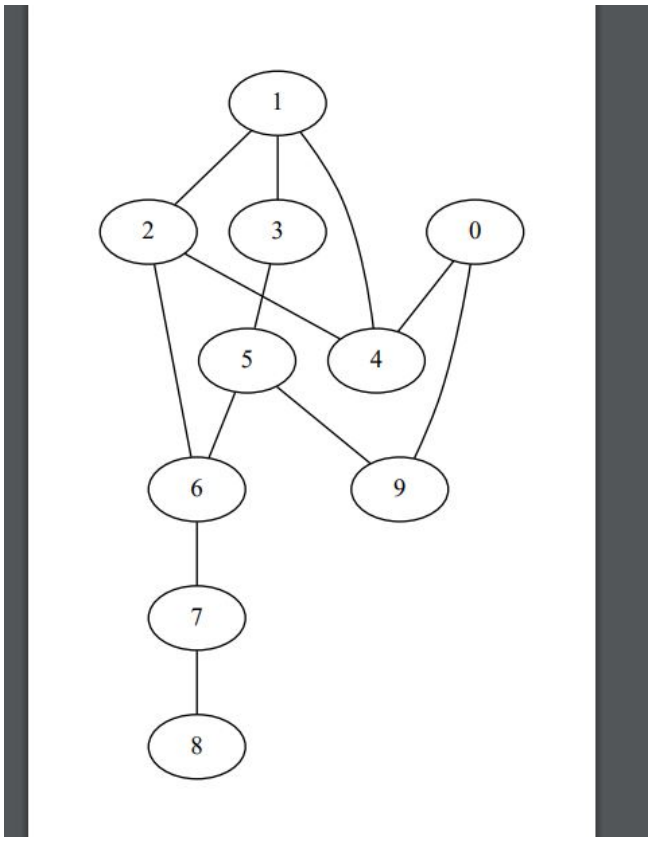


- 输入

```
10
0 4
0 9
1 2
1 3
1 4
2 4
2 6
3 5
5 6
5 9
6 7
7 8
```

输出





实验总结

- DFS应用里发现自己对于递归到非递归的转换尚存在一些问题，过于生疏，对用辅助栈手动模拟时一些关键条件的判断理得不够清楚，关键还是要理清楚何时入栈，出栈，何时循环等一些关键点。
- BFS应用里因为本次实验边权均默认为1，所以可以利用简单的广度优先遍历即可求得结果，更一般的情况则还是应该考虑Dijkstra算法的实现
- graphviz的使用:个人感觉做出的图不够美观，不过这次实验也让我明白要学会多尝试一些新工具、模型。只有了解的知识足够广，才能更好地思考出属于自己的想法。