

# 实验

## 一、实验要求

### <DFS 的应用>

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,求无向图中的全部关节点,并按照定点编号升序输出。

### <BFS 的应用>

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

### <无向图的可视化>

使用 graphviz 的 Python 接口即可。

## 二、设计思路

### <DFS 的应用>

根据书中算法,易写出求图中关节点的递归算法。由于题目要求使用非递归算法,故使用栈模拟递归过程,可以将递归算法“翻译”为非递归算法。

### <BFS 的应用>

维护一个队列和一个 dist 数组,先将顶点 0 入队,dist[0]设为 0,dist 中其他元素设为 inf。之后每次从队首取出一个顶点 u,检查每一个与之相邻的顶点 v,如果  $\text{dist}[u]+1 < \text{dist}[v]$ ,则将 dist[v]置为 dist[u]+1,并将 v 入队。反复执行上述过程,直到队列为空。此时 dist[u]即为顶点 0 到顶点 u 的最短路径长度。

输出 0 到 u 的最短路径时,维护一个栈,初始时将 u 入栈。之后从 u 开始,检查每个与之相邻的顶点 v,如果  $\text{dist}[v] == \text{dist}[u]-1$ ,则将 v 入栈,之后再从 v 开始执行相同的过程,直到到达顶点 0。最后将栈中的元素从顶到底依次输出,即为 0 到 u 的最短路径。

## 三、关键代码讲解

```
#include<stdio.h>
#include<algorithm>
#include<vector>
#include<queue>
#include<stack>
#define MAXN 100
#define INF 0x3f3f3f3f
using namespace std;
int n;
vector<int> graph[MAXN];           //图的邻接表表示, graph[i]表示 i 的邻居列表
bool mat[MAXN][MAXN],iscut[MAXN]; //图的邻接矩阵表示
//iscut 数组记录某个节点是否为关节点
int dist[MAXN],dfn[MAXN],low[MAXN]; //dfn[i]==j 表示顶点 i 在 DFS 过程中被第 j 个访问到
struct State{
    int lowu,child,i,u,fa;
    State (int &dfs_clock,int _u,int _fa):u(_u),fa(_fa){
        //u 表示当前访问的节点, fa 表示当前节点在 DFS 树中的父节点
```

```

//在依次访问 u 的相邻节点的过程中，已经访问到了节点 i-1，之后将从 i 开始轮询
//child 记录了当前已经给节点 u 在 DFS 树中已有的孩子节点数目
lowu = dfn[u] = ++dfs_clock;    //因为这里的 dfs_clock 在同一个 DFS 中是共享的
child = 0;                    //所以需要传引用
i = 0;

}
};

void dfs(){
    stack<State> s;
    int dfs_clock = 0, i, lowv;    //lowv 仅仅只是为了方便计算的中间变量
    for(i = 0; i < n; ++i){        //i 被作为循环变量在两处使用
        iscut[i] = false;
        dfn[i] = 0;
    }
    State cur(dfs_clock, 0, -1);    //cur 存储的是当前一级的状态
    while(dfs_clock < n || !s.empty()){    //dfs_clock==n 表明所有点都已遍历完
        for(i = cur.i; i < n; ++i){        //在邻接矩阵中寻找与 cur.u 相邻的顶点
            if(mat[cur.u][i]){
                cur.i = i + 1;
                break;
            }
        }
        if(i == n){    //如果没有找到，则退栈，返回上一级
            if(cur.fa < 0 && cur.child > 1) iscut[cur.u] = true;
            lowv = low[cur.u] = cur.lowu;
            cur = s.top();
            s.pop();
            cur.lowu = min(cur.lowu, lowv);
            if(cur.fa >= 0 && lowv >= dfn[cur.u]){
                iscut[cur.u] = true;
            }
        }
        else{    //如果找到了，则拓展 DFS 树，并进入下一级
            if(!dfn[i]){
                ++cur.child;
                s.push(cur);
                cur = State(dfs_clock, i, cur.u);
            }
            else if(dfn[i] < dfn[cur.u] && i != cur.fa){
                cur.lowu = min(cur.lowu, dfn[i]);
            }
        }
    }
}
}

```

```

void print_trace(int x){
    stack<int> s;                                //使用栈来存储回溯路径上的节点
    s.push(x);
    while(s.top() != 0){
        for(auto i : graph[s.top()]){
            if(dist[i]+1 == dist[s.top()]){      //i 可以作为路径中的上一个节点
                s.push(i);
                break;
            }
        }
    }
    bool isfirst = true;                         //第一个节点输出时不用在前面加箭头
    while(!s.empty()){
        if(!isfirst)printf("->%d",s.top());
        else{
            printf("%d",s.top());
            isfirst = false;
        }
        s.pop();
    }
}

void bfs(){
    //先初始化 dist 数组
    for(int i = 0; i < n; ++i){
        dist[i] = INF;
    }
    dist[0] = 0;
    queue<int> q;
    q.push(0);                                  //从 0 开始 BFS
    int head;
    while(!q.empty()){
        head = q.front();                       //取出头结点，更新与之相邻的节点
        for(auto i : graph[head]){
            if(dist[head]+1 < dist[i]){
                dist[i] = dist[head]+1;
                q.push(i);
            }
        }
        q.pop();
    }
}

int main(){
    FILE *f;

```

```

f=fopen("test.txt","r");           //打开名为 test.txt 的测试文件
fscanf(f,"%d",&n);
int i,j;
while(fscanf(f,"%d %d",&i,&j)!=EOF){ //读取文件并构建图
    graph[i].push_back(j);
    graph[j].push_back(i);
    mat[i][j]=mat[j][i]=true;
}
dfs();
bfs();
for(i=0;i<n;++i){
    if(iscut[i])printf("%d ",i);
}
printf("\n");
for(i=1;i<n;++i){
    printf("%d ",dist[i]);
    print_trace(i);
    printf("\n");
}
fclose(f);
return 0;
}

```

---

```

from graphviz import Graph
g = Graph("graph")
def sp(s, separator = ['\t',' ','\n']):
    #将字符串按空白字符拆分，返回一个列表
    #与主算法无关
    ans=[]
    i=j=0
    while j<len(s) and s[j] in separator:
        j+=1
    i=j
    while i<len(s):
        while j<len(s) and s[j] not in separator:
            j+=1
        ans.append(s[i:j])
        while j<len(s) and s[j] in separator:
            j+=1
        i=j
    return ans
with open('test.txt','r') as f:
    #调库，绘图
    n = int(f.readline())
    for i in range(n): #创建顶点 0~n-1

```

```

        g.node(name = str(i))
    for line in f:
        numbers = sp(line)
        g.edge(numbers[0], numbers[1])
g.view()

```

## 四、调试分析

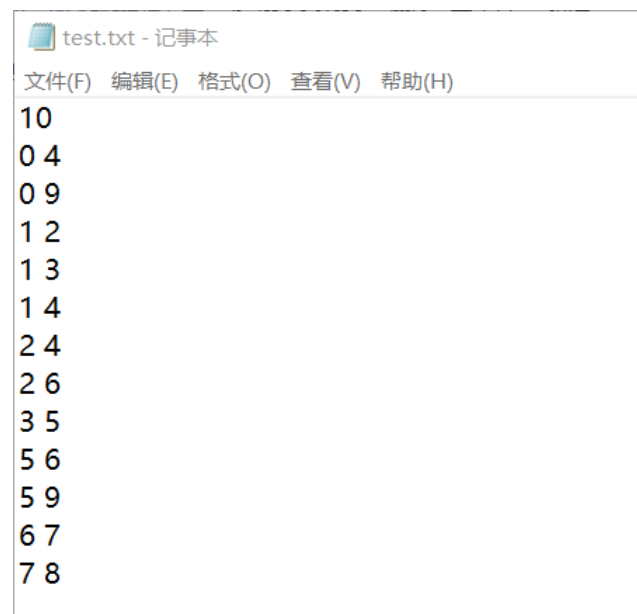
由于使用邻接矩阵，DFS 的时间复杂度为  $O(n^2)$ ；最坏情况下  $n$  个顶点均需入栈，空间复杂度为  $O(n)$ 。

由于使用邻接表，BFS 的时间复杂度为  $O(n+e)$ ；最坏情况下， $n$  个顶点均需入列，空间复杂度为  $O(n)$ 。

因此，总的时间复杂度为  $O(n^2 + e)$ ，总的空间复杂度为  $O(n)$ 。

## 五、代码测试

测试数据：

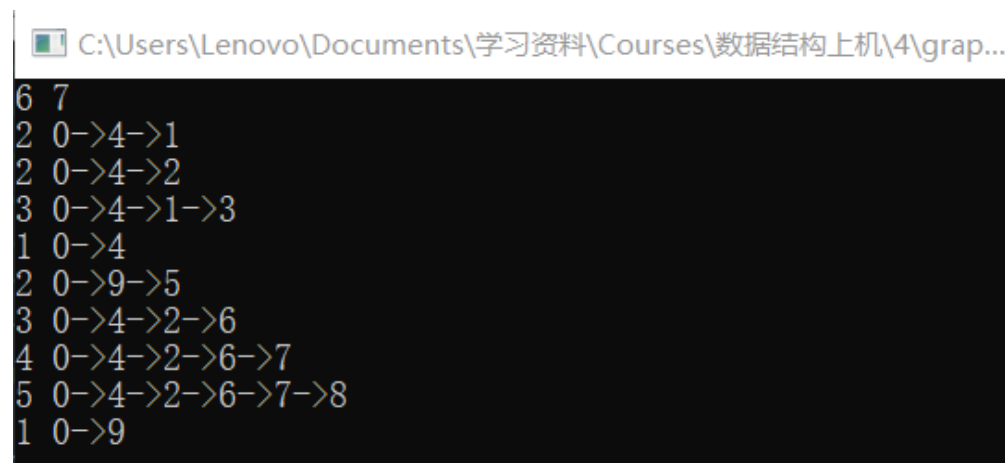


```

test.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
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

```

运行结果：



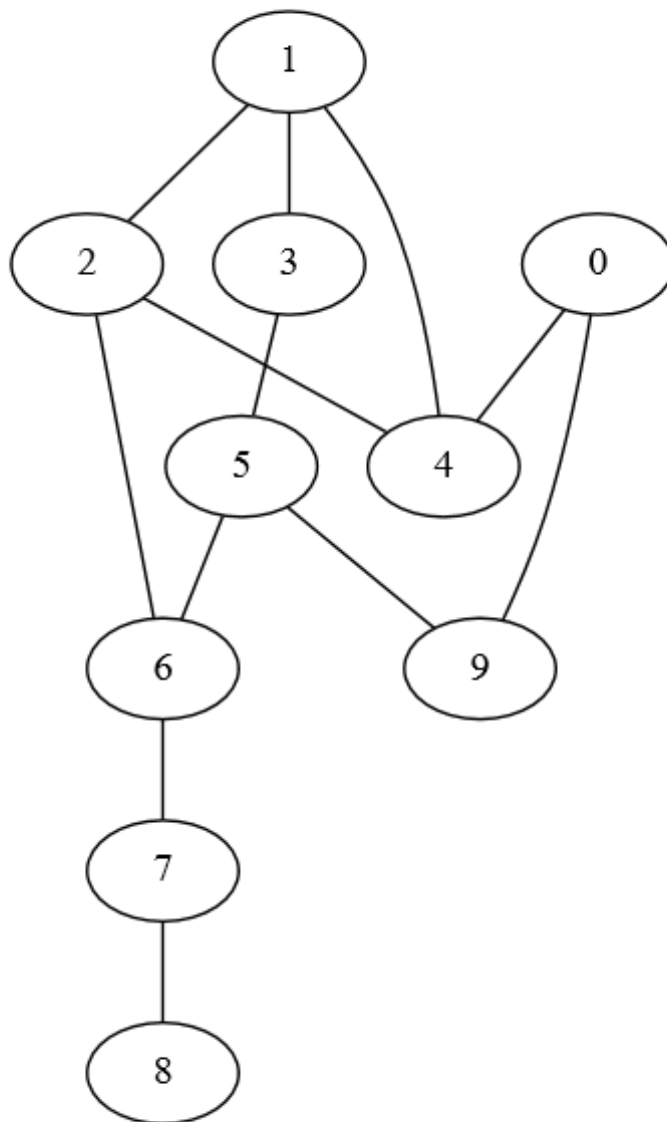
```

C:\Users\Lenovo\Documents\学习资料\Courses\数据结构上机\4\grap...
6 7
2 0->4->1
2 0->4->2
3 0->4->1->3
1 0->4
2 0->9->5
3 0->4->2->6
4 0->4->2->6->7
5 0->4->2->6->7->8
1 0->9

```

结果正确。

绘图结果：



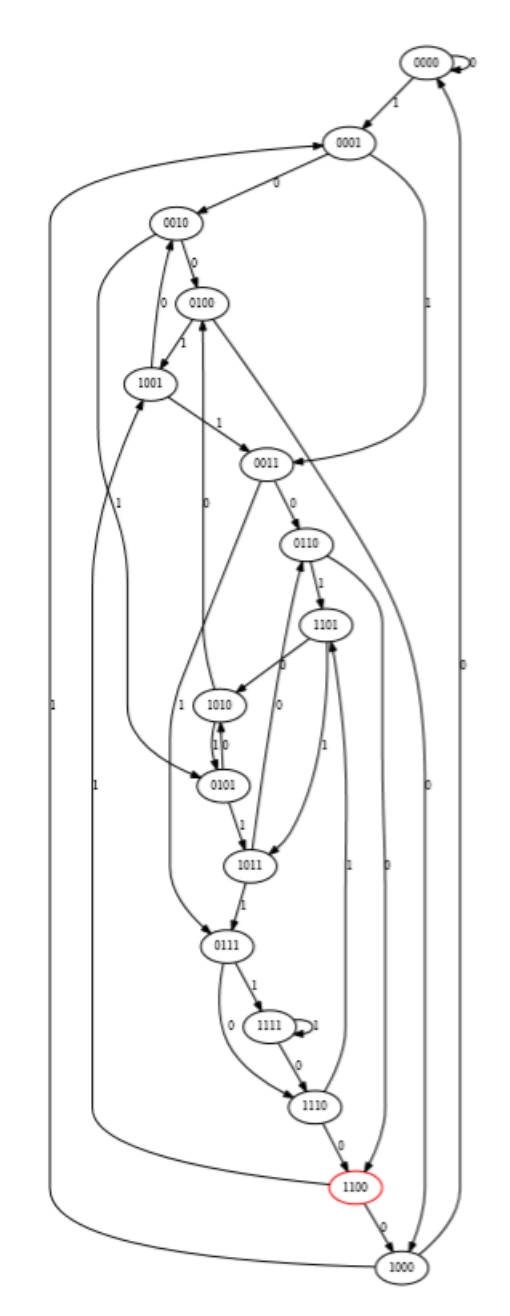
## 六、实验总结

本题只要写出了递归的 **DFS** 算法，就能通过手动模拟递归栈的方式来构造迭代算法。在递归版本的程序中只会有一个入栈口，就以该入栈口为分割，每个循环的过程都是从一个入栈口到另一个入栈口。另外为了简便和美观，部分初始化操作被写在了 **State** 的构造函数中。

**BFS** 部分也可以另外添加一个数组来记录路径每一个点在最短路径中的上一个节点，这样就可以无需在输出时根据 **dist** 数组来判断，可以用空间换一些时间。本程序没有采用这样的做法。

图的可视化部分非常简单，这还得归功于 **graphviz** 的接口设计简洁易用。但是在本人进行了一些尝试之后发现，**graphviz** 对于顶点和边在图中的排布不太擅长，导致绘制的图缺乏美感。比如上文中所生成的图，明明是一个平面图，却发生了边的交叉。当顶点更多时绘制

的结果会呈长条状(如下图):



可见 graphviz 在图的美观方面表现较差。

## 七、附录

源程序文件名清单:

PB18111789\_夏寒\_4.cpp

PB18111789\_夏寒\_4.py

//DFS、BFS 的程序

//图的可视化