# Lab 04

——图及其应用

## 一．实验要求

给定无向图 G，默认边权为 1，完成以下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。

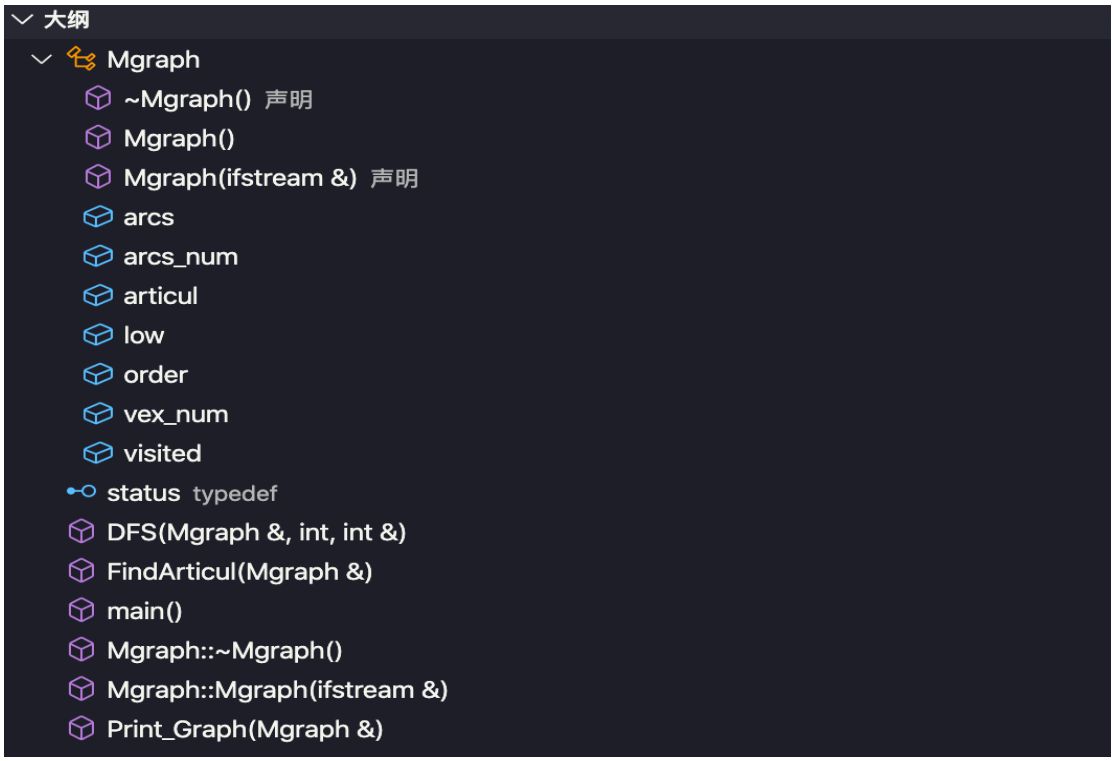
**BFS 的应用**

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。

## 二．设计思路

（1） 非递归的深度优先搜索寻找关节点：

大纲如下：



基本思路仍为在生成树上，通过各结点的 low 值进行判断。参考教材上算法 7.10 与 7.11 的实现，自己设立一个工作栈，存储上一步的信息，将 DFS 函数中的递归部分改为非递归实现。

（2） 寻找最短路径：

大纲如下：


由于给定的图 G 是无权图（默认边权为 1），故只需进行广度优先搜索，即可求得从 V0 到其它顶点的最短路径；我们采用“队列”这一数据结构，来辅助存储搜索时各个层次的信息。

## 三．关键代码讲解

（1） 非递归实现的 DFS 函数：

```
void DFS(Mgraph &G,int v1,int &count)
{
    int e;
```

```
    stack<int> S;
    int v2 = 0;
    int _v1 = v1;
    vector<int> min(G.vex_num);
    do{
        if(G.order[v1] == 0 )
            G.order[v1] = min[v1] = ++count;
        for(;v2<G.vex_num;v2++)
        {
            if(G.arcs[v1][v2] == 1 )    //v1 与 v2 之间存在边
            {
                if(G.visited[v2] == false) //v2 未曾访问，是 v1 的孩子
                {
                    G.visited[v2] = true;
                    //DFS(G,w,count);
                    S.push(v1);
                    v1 = v2;
                    v2 = 0;
                    break;
                }
                else if(G.order[v2] < min[v1]) //v2 已访问，是 v1 在生成树上的祖先
                    min[v1] = G.order[v2];
            }
        }//for
        /*叶子节点直接被排除*/
        if(v2 == G.vex_num) //此时自 v1 已经无法向下进行,已访问完 v1 的所有子树,计算
low[v1]
        {
            G.low[v1] = min[v1];
            v2 = v1;
            if(v1 != _v1)
            {
                v1 = S.top();
```

```
                S.pop();
            }
            else
                break;
            if(G.low[v2] < min[v1])
                min[v1] = G.low[v2];
            if(G.low[v2] >= G.order[v1])    //v1 的任意孩子没有指向 v1 祖先的回边
                G.articul.push_back(v1);
        }
    }while(!S.empty() || v1 == _v1);
    return;
}
```

用栈来进行非递归实现的关键在于，要保证顶点各个值（low、order、min）在逻辑上的运算顺序合理。

（2） 用队列辅助进行广度优先搜索：

```
void BFS_FindPath(ALGraph &G)
{
    /*设置局部变量*/
    Queue<int> Q(G.vex_num);
    vector<vector<int> > path(G.vex_num);
    bool final[G.vex_num];
    int distance[G.vex_num];
    for(int i = 0;i<G.vex_num;i++)
    {
        final[i] = false;
        distance[i] = G.vex_num;
    }
    int _distance = 1;
    int rear,rear_temp;
    //进行第一次搜索
    for(ArcNode* p = G.vertices[0]->firstedge; p ; p=p->nextarc)
    {
```

```
        Q.EnQueue(p->adjvex);    //与 v0 相邻的边进入队列，作为下一次搜索的出发点之一
        final[p->adjvex] = true;     //标记为已访问
        distance[p->adjvex] = _distance;     //记录距离
        path[p->adjvex].push_back(p->adjvex);    //记录路径
        if(p->nextarc == NULL)
            rear = p->adjvex;    //用于控制距离增长
    }
    _distance++;
    //进行之后的搜索
    while(!Q.if_empty())
    {
        int v = Q.DeQueue();     //从 v 出发广度优先搜索
        for(ArcNode* p = G.vertices[v]->firstedge; p ; p = p->nextarc)
        {
            if(final[p->adjvex] == false)    //与 v 相邻且未访问过的节点
            {
                rear_temp = p->adjvex;
                Q.EnQueue(p->adjvex);
                final[p->adjvex] = true;
                distance[p->adjvex] = _distance;
                /*复制重复路径*/
                path[p->adjvex] = path[v];
                path[p->adjvex].push_back(p->adjvex);
            }
            if( p->nextarc == NULL && rear == v )
            {
                rear = rear_temp;
                _distance++;
            }
        }
    }
    for(int j = 1;j < G.vex_num;j++)
    {
```

```
        cout<<distance[j]<<" 0";
        for(int k = 0;k<path[j].size();k++)
                cout<<"->"<<path[j][k];
        cout<<endl;
    }
    return;
}
```

路径信息：

使用了一个二维的 vector 容器来记录 v0 到其它各个顶点的路径信息：每当从顶点 v1 出发搜索找到一个未访问过的顶点 v2，则将 v0 到 v1 的路径复制给 v2，再将 v1 添加到该路径中。

计算距离的方法：

由于广度优先搜索具有层次性，所以我们设置了一个变量来记录当前层的最后一个顶点；每当从最后的顶点出发并搜索完其相邻的所有顶点后，距离 distance++

（3） 调用 graphviz 软件作图：

```
void Print_Graph(Mgraph &G)
{
    ofstream fout;
    fout.open("graph.dot",ios_base::out);
    fout<<"digraph G{"<<endl;
    for(int i = 0;i<G.vex_num;i++)
    {
        for(int j = i;j<G.vex_num;j++)
        {
            if(G.arcs[i][j] == true )
                fout<<i<<" -> "<<j <<" [arrowhead= none]"<<endl;
        }
    }
    fout<<"}";
    fout.close();
```

```
    system("dot -Tpng graph.dot -o sample.png");
    return;
}
```

将顶点、边信息按照相应的语法输入至文件，再用 system 调用命令行控制 graphviz 软件作图即可。
本函数使用邻接矩阵的存储方式进行绘图。为了保证无重边，仅根据上三角矩阵存储的信息进行输入；无向图用[arrowhead = none]删去边上的箭头。

## 四．调试分析

***时间复杂度：***

（1） DFS 非递归遍历：

图采用邻接矩阵存储时，每次查找当前顶点的邻接点耗时最多为 O(N),因此整个遍历最坏情况下的时间复杂度为 $O(N^2)$

（2） BFS 搜索最短路径：

图采用邻接表的方式存储时查询邻接点速度较快，为 O(1)，进而查找所有的邻接点的复杂度为 O(E),程序总共的时间复杂度为 O(N+E)

***空间复杂度：***

（1） 邻接矩阵存储方式：

所用空间为 $O(N^2)$

（2） 邻接表存储方式：

所用空间为 O(N+E)

（3） DFS 非递归实现：

除局部变量外仅用了一个栈来存储之前的顶点，最大容量不会超过所有顶点的总数，所以空间复杂度为 O(N)

（4） BFS 搜索最短路径：

设置了一个队列来存储上一层遍历中访问的结点,其占用的空间最大仅与 N 成线性关系；而存储路径上的结点信息时，空间上的开销较

大，为 $O(N^2)$,故总的空间复杂度为 $O(N^2)$

## 五．代码测试

必做：

运行结果截图如下：

（1）讲义中的例子：

```
yuqideMacBook-Pro:第四次 yuqilee$ cd "/Users/yuqilee/data_structure/上机实验/
验/第四次/"Articul
0 1 3 6
yuqideMacBook-Pro:第四次 yuqilee$ cd "/Users/yuqilee/data_structure/上机实验/
cture/上机实验/第四次/"Shortest_Path
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->7->10
1 0->11
2 0->11->12
yuqideMacBook-Pro:第四次 yuqilee$
```

（2）test1:

```
yuqideMacBook-Pro:第四次 yuqilee$ cd "/Users/yuqilee/data_structure/上机实验/
验/第四次/"Articul
3 5
yuqideMacBook-Pro:第四次 yuqilee$ cd "/Users/yuqilee/data_structure/上机实验/
cture/上机实验/第四次/"Shortest_Path
1 0->1
1 0->2
2 0->1->3
3 0->1->3->4
2 0->2->5
2 0->1->6
3 0->2->5->7
yuqideMacBook-Pro:第四次 yuqilee$
```

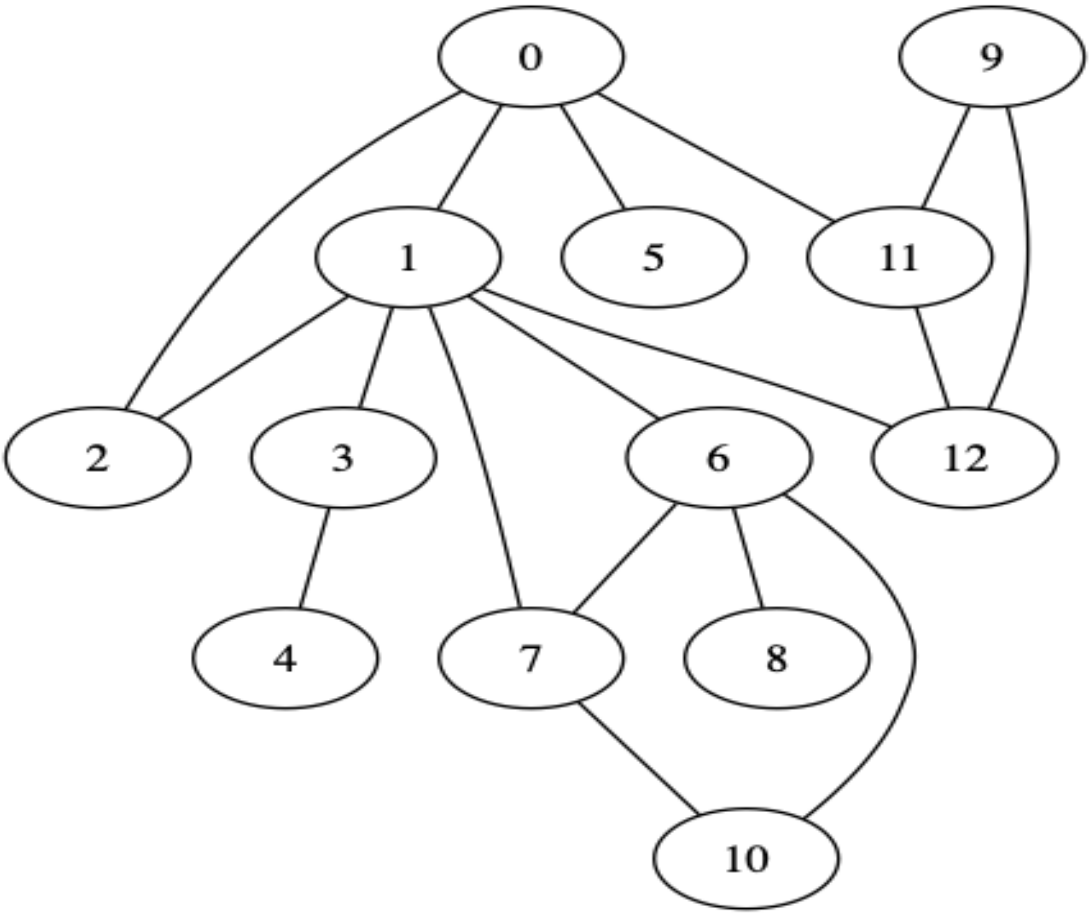
（3）test2:

```
yuqideMacBook-Pro:第四次 yuqilee$ cd "/Users/yuqilee/data_structure/上机实验/
验/第四次/"Articul
6 7
yuqideMacBook-Pro:第四次 yuqilee$ cd "/Users/yuqilee/data_structure/上机实验/
cture/上机实验/第四次/"Shortest_Path
2 0->4->1
2 0->4->2
3 0->9->5->3
1 0->4
2 0->9->5
3 0->9->5->6
4 0->9->5->6->7
5 0->9->5->6->7->8
1 0->9
yuqideMacBook-Pro:第四次 yuqilee$
```

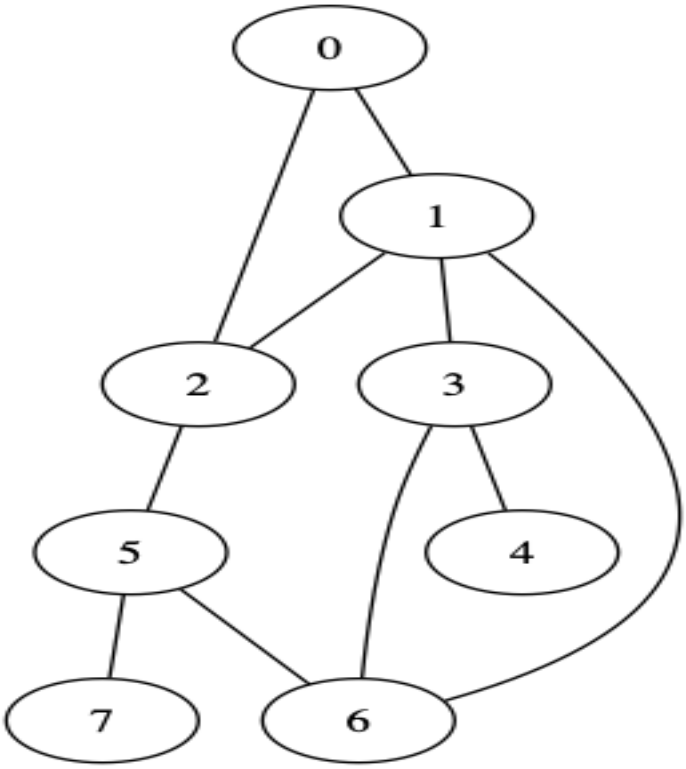
选做：

graphviz 画图如下：

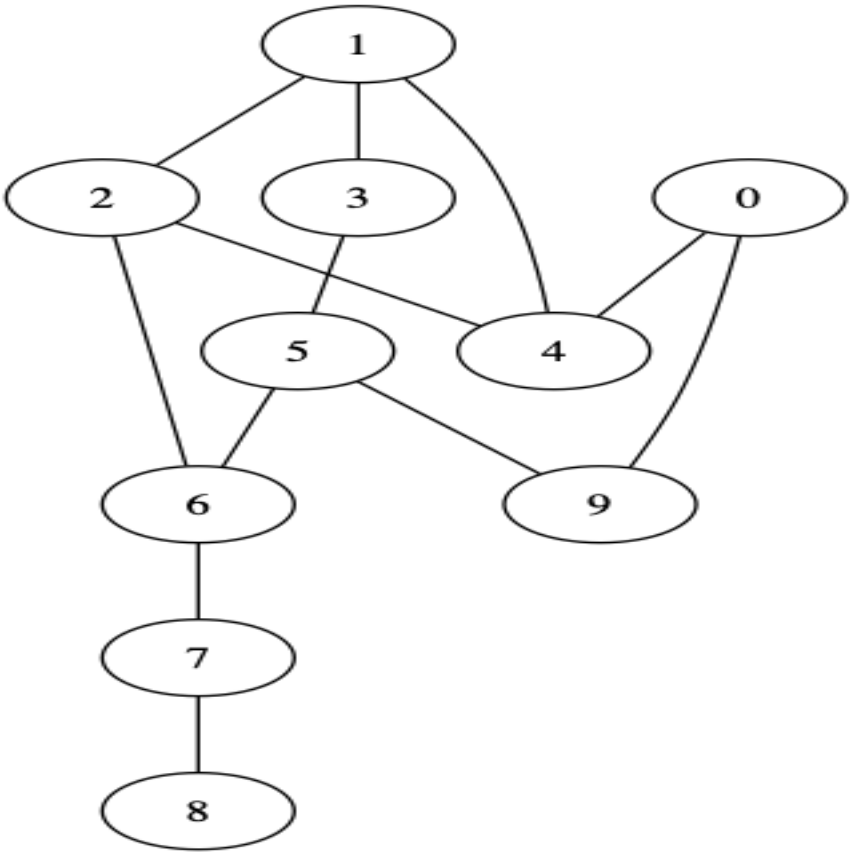
（1）讲义中的例子：




（2）test1:




test2:



## 六．实验总结

(1) 回顾了“图”这一数据结构常见的存储方式，对不同存储方式下图的创建进行了练习

(2) 用栈、队列等数据结构，将 DFS、BFS 两种遍历方法用非递归方式实现，锻炼了逻辑思维能力，增强了对线性数据结构的使用熟练度

(3) 使用 Graphviz 软件，在图创建后动态生成了由图的顶点及边构成的图片，将这一数据结构可视化，使之更为形象

## 七．附录

“PB18081496_李昱祁_4.pdf”

“PB18081496_李昱祁_4_1.cpp”

(通过文件中内容读取数据，建立用邻接矩阵存储的图，输出其关节点，并调用 graphviz 软件动态生成可视图片)

“PB18081496_李昱祁_4_2.cpp”

(通过文件中内容读取数据，建立用邻接表存储的图，输出 v0 到其它各个顶点的路径信息)

“case0.txt”　（讲义中的数据）

“case1.txt”　（test1）

“case2.txt ”　（test2）