数据结构

实验报告（五）

图的连通分支

学号：3020205015

姓名：石云天

班级：智能机器平台 2 班

日期：2022.12.10

目　录

一、实验内容描述

本次实验主题是求图的连通分支，对于给定的无向图，需要对该图进行遍历，并打印连通分支。有以下三点具体要求：

（1） 需要保存所有连通分支，并打印。

（2） 需要使用深度优先和广度优先两种遍历方式。

（3） 不允许使用递归。

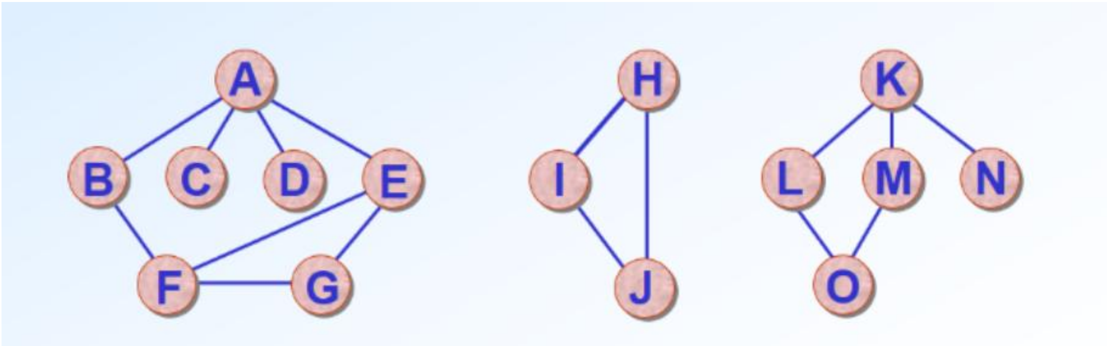

图 1 实验要求的图

二、实验步骤

（1）根据上课所讲，回顾图和连通分支的基本概念，考虑图的存储结构。回顾图的邻接表和邻接矩阵两类存储结构，并根据实验要求选取合适的存储结构。回顾图的遍历算法，思考深度优先（DFS）算法和广度优先（BFS）算法的区别。

（2）仔细阅读实验要求，考虑图的连通分支的求解方法，并进一步考虑 DFS 和 BFS 算法要素。尤其需要注意的是本次实验不允许使用递归，所以需要对于 DFS 算法中的递归部分进行修改，改为栈来实现。列写伪代码，体会算法思想。

（3）利用 CodeBlocks 编译器，配置环境，基于 C++语言将算法用程序实现。

（4）编译运行程序，使用样例进行程序测试，观察所编程序是否实现要求的功能。

（5）考察算法的时间复杂度和空间复杂度，评价算法的优劣，进一步优化程序。

（6）撰写实验报告，进行实验总结与反思。

三、程序设计

（一）抽象数据类型 ADT

首先由于本次实验中不允许使用递归，因此 DFS 算法要依靠栈来实现，BFS 算法要依靠队列来实现，故需要提前准备这两种数据结构的抽象数据类型。

不带头节点的链式栈的抽象数据类型：

ADT Stack{

数据对象：D={$a_i|a_i \in char, i = 1,2,3 \ldots$}

数据关系：R={<a_{i-1}, a_i>|$a_i, a_{i-1} \in D$}

基本操作：

`void InitStack(Stack &S)`

操作结果：链栈的初始化函数

`void DeleteStack(Stack &S)`

操作结果：链栈的销毁

`int StackEmpty(Stack S)`

操作结果：判断栈空

`void Push(Stack &S,char e)`

操作结果：入栈

`void Pop(Stack &S,char &e)`

操作结果：出栈

`char GetTop_Stack(Stack S)`

操作结果：返回栈顶元素

`void PrintStack(Stack S)`

操作结果：输出栈内元素(LIFO)

}ADT Stack;

顺序循环队列的抽象数据类型：

ADT Queue{

数据对象：D={$a_i|a_i \in int, i = 1,2,3 \ldots$}

数据关系：R={<a_{i-1}, a_i>|$a_i, a_{i-1} \in D$}

基本操作：

`int InitQueue(Queue &Q)`

操作结果：队列的初始化

`void DeleteQueue(Queue &Q)`

操作结果：队列的销毁

int EnQueue(Queue &Q,int e)

操作结果：入队

int DeQueue(Queue &Q,int &e)

操作结果：出队

int GetTop_Queue(Queue &Q,int &e)

操作结果：返回队头元素

int PrintQueue(Queue Q)

操作结果：输出队列内元素(FIFO)

}**ADT SqQueue;**

图的抽象数据类型：

ADT Graph{

数据对象：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系：一条弧连接了两个顶点，这些弧组成的集合称为边集。

基本操作：

CreateGraph(&G)

操作结果：创建图

InitGraph(&G)

操作结果：图的初始化

DeleteGraph(&G)

操作结果：图的销毁

InsertVex (&G,v)

操作结果：在图 G 中增添新的顶点

DeleteVex(&G,v)

操作结果：在图 G 中删除顶点

InsertArc (&G,v,w)

操作结果：在图 G 中<v,w>增添新的边<v,w>

DeleteArc(&G,v,w)

操作结果：在图 G 中删除边<v,w>

GetVex (G,v)

操作结果：获取图中的某一节点

GetNextVex (G,v,w)

操作结果：获取图的邻接表中 v 的 w 之后的节点

DFS (G，visited[])

操作结果：对图进行深度优先遍历

BFS（G，visited[]）

操作结果：对图进行广度优先遍历

}ADT Graph;

（二）存储结构

图的存储结构主要有邻接矩阵和邻接表两种方式。邻接矩阵利用静态二维数组存储结果，邻接表是一种链式存储结构。大多数图都利用邻接表进行存储，本实验中同样采用邻接表的方式。为了建立邻接表，我们还需要建立一些辅助用的数据结构：边、结点、图，具体如下所示：

```
typedef struct Node{
    int to;
    int from;
    Node *link;
}edge;

typedef struct vex{
    char data;
    edge *adj;
}vex;

typedef struct Gnode{
    int n;//节点个数
    int e;//边的个数
    vex vlist[10000];
}Graph;
```

此外由于本实验不允许使用递归完成，因此还需要利用栈与队列来实现 DFS 与 BFS 算法，所用数据结构如下：

```
typedef struct Qnode{
    int data;
    Qnode *next;
}Qnode,*Queue;
```

```
typedef struct Snode{
    int data;
    Snode *next;
}Snode,*Stack;
```

（三）算法简述

图的遍历主要有 DFS(深度优先)与 BFS(广度优先)两种算法，在遍历前，首先需要建立一个 visited[]数组以记录节点是否被访问，此外由于本实验需要记录各连通分支的结点，故还需额外添加一个变量 k 用于计数。下面分别对 DFS 与 BFS 两种算法思想进行介绍。

一、DFS 算法

DFS 是指从一个节点开始，若其有邻居节点就会不断探索，直至访问的节点无邻居节点为止，随后采用回退的办法再去遍历其他节点，因此 DFS 需要使用递归或调用栈进行实现，具体流程可以分为五步：

（1）在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一未访问过的邻接顶点w_1；

（2）再从w_1出发，访问与w_1邻接但还没有访问过的顶点w_2；

（3）然后再从w_2出发，进行类似的访问。如此进行下去, 直至到达所有的邻接顶点都被访问过的顶点 u 为止；

（4）接着退回一步，退到前一次刚访问过的顶点，观察其是否还有其它没有被访问的邻接顶点，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；如果没有，就再退回一步进行搜索；

（5）重复上述过程，直至连通图中所有顶点都被访问过为止。

二、BFS 算法

BFS 是一种分层的搜索过程，每次前进可能访问多个节点，它不会像 DFS 一样出现回退的过程。为了达到逐层访问的目的，该算法实现时使用了队列，以存储正在访问的一层与下一层的顶点，从而便于访问下一层，具体流程可分为三步：

（1）访问起始节点 v，由 v 出发，依次访问 v 的各个未被访问过的邻接节点 $w_1, w_2, \ldots$；

（2）再顺序访问$w_1, w_2, \ldots$的所有还未被访问过的邻接顶点；

（3）再从这些邻接顶点出发，访问它们的所有还未被访问过的邻接顶点。如此进行下去，直至图中所有顶点都被访问到为止。

三、统计连通分支

在遍历算法的主函数中，采用 for 循环的方式，for 循环的次数就代表了连通分支的个数，因此需要增加一个计数变量 k，在打印每个节点时附加打印它所在的连通分支序号，这样就能将所有连通分支都记录下来。

四、栈与队列的基本操作

由于本实验不允许使用递归的办法，故需要借助栈来实现 DFS 算法，借助队列来实现 BFS 算法，因此需要实现一些栈与队列的基本操作，如：出入栈、栈判空、出入队、队判空等，这些操作在实验二中已经实现并做出具体介绍，此处不再赘述。

（四）程序代码

为保证实验报告的清晰和可读性，将源代码以**附录形式**附于文末。

四、调试分析

（一）调试过程和主要错误

本次实验编写的两种算法在整体思路上都较为清晰，在调试过程中也没有出现很多问题。在利用栈实现 DFS 算法的非递归版本时，刚开始时出入栈的顺序存在一些问题，导致程序无法正常运行，在对顺序进行改进调整后便可顺利完成。

同时在编写栈和队列这两种数据结构时，出现了 Segmentation Fault 的错误。这个类型的错误以前经常出现，根据经验判断，应该是出现了野指针的问题。于是利用断点调试工具进行测试，发现是一个判断条件导致数组越界产生错误，修改后程序顺利运行。

（二）时间复杂度

BFS 算法与 DFS 算法的不同之处仅在于对节点访问顺序不同，其时间复杂度相同。其时间复杂度主要取决于所用的存储结构，假设顶点数为 n，边的数量为 e，则对于本实验中利用邻接表存储的情况而言，时间复杂度为 $O(n+e)$，若利用邻接矩阵进行存储，时间复杂度为 $O(n^2)$。

五、程序测试

在完成全部程序编写后，输入测试样例进行测试，由于使用邻接表的存储结构，需要将所求图的各条边的信息输入进去。用于本次测试的是一个无向图，其有 15 个节点与 16 条边，由于无向图边的双向连通性，故应输入 32 组，输入方法为<起始节点序号+结束节点的序号>。具体输入如下：

```
1 0
0 1
0 2
2 0
0 3
3 0
0 4
4 0
1 5
5 1
5 4
4 5
5 6
6 5
6 4
4 6
8 7
7 8
8 9
9 8
7 9
9 7
10 11
11 10
10 12
12 10
10 13
13 10
11 14
14 11
14 12
12 14
```

BFS 与 DFS 算法测试结果见下图 2：

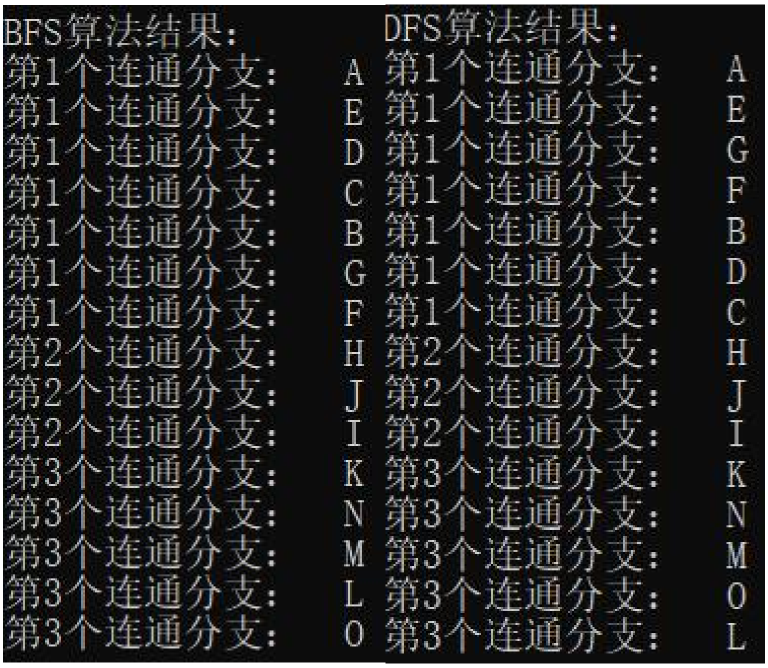

图 2 BFS 与 DFS 算法测试结果

通过测试结果可以看出，程序运行符合预期。

六、实验总结

通过这次试验，我发现我对图的遍历算法的理解不够深入全面，需要不断巩固学习，加深理解。图为非线性结构，在存储过程中，有邻接表与邻接矩阵两种选择，由于使用邻接表进行存储，遍历时更加容易也更为灵活，故本次实验选择邻接表这一存储结构。在实验过程中，我对 BFS 与 DFS 算法的区别与联系掌握得更加清楚，两种算法的时间复杂度完全相同，唯一不同的是节点的访问顺序，其中 DFS 需要回退，需要借助栈进行实现，而 BFS 算法是分层次的访问，需要借助队列进行实现。除此之外，在本次实验过程中，编写、调试程序花费了很长时间：首先是将递归程序改写为非递归的形式，由于本次实验限制了递归算法的使用，因此需要将老师上课讲的递归版本的 DFS 算法程序改写为非递归版本，其主要方法就是借助额外建立的一个栈，不断地出入栈，以达到与递归相同的效果，在此过程中还需注意出栈的先后顺序，不然会出现逻辑上的错误。其次是对于空指针的理解更加深刻，我最初以为 Segmentation Fault 类型报错是因为数组越界，经过不断查阅资料发现是因为调用了空指针，空指针不会指向任何实体，因此在程序编写过程中需要格外注意各指针变量指向的变化，在 delete 操作完成后，最好在后面加一行将指针置为 NULL 的代码，这可以有效避免调用空指针的错误。在本次实验后，我还需要精益求精，不断改进程序，优化函数性能，实现预期目标与功能所需。

附录：程序源代码：BFS 算法与 DFS 算法

```
#include<iostream>
using namespace std;

//邻接表数据结构

typedef struct Node{
    int to;
    int from;
    Node * link;
}edge;

typedef struct vex{
    char data;
    edge* adj;
}vex;

typedef struct Gnode{

    int n;//节点个数

    int e;//边的个数

    vex vlist[10000];
}Graph;

//邻接表的建立

void InitGraph(Graph &G){
    G.n = 15;
    G.e = 32;
    for(int i = 0;i < G.n;i++){
        G.vlist[i].data = 65 + i;
        G.vlist[i].adj = NULL;
    }
    for(int i = 0;i < G.e;i++){
        int tail,head;
        cin >> tail >> head;
        edge* p = new edge;
        p->to = head;
        p->link = G.vlist[tail].adj;
        G.vlist[tail].adj = p;
```

```
    }
}
```

//获取节点的第一个相邻节点

```
int getfirst(Graph G,int v){
    if(G.vlist[v].adj == NULL) return -1;
    else return G.vlist[v].adj->to;
}
```

//获取邻接表中 w 后面的节点

```
int GetNextVex(Graph G,int v,int w){
    edge *p = G.vlist[v].adj;
    while(p->to != w){
        p = p->link;
    }
    if(p->link == NULL) return -1;
    else return p->link->to;

}
```

//建立队列用于 BFS 算法

```
typedef struct Qnode{
    int data;
    Qnode* next;
}Qnode,*Queue;

void InitQueue(Queue &Q){
    Q->next = NULL;
}

int QueueEmpty(Queue Q){
    if(Q->next == NULL) return 1;
    else return 0;
}

void EnQueue(Queue &Q,int v){
    Qnode* p = new Qnode;
    p->data = v;
    p->next = NULL;
    Qnode *q = Q;
    while(q->next != NULL){
```

```
        q = q->next;
    }
    q->next = p;
}

void DeQueue(Queue &Q,int &v){
    Qnode* p = Q->next;
    v = p->data;
    Q->next = p->next;
    delete p;
}
```

```
//建立栈用于 DFS 算法
typedef struct Snode{
    int data;
    Snode *next;
}Snode,*Stack;

void InitStack(Stack &S){
    S = new Snode;
    S->next = NULL;
}

int StackEmpty(Stack S){
    if(S->next == NULL) return 0;
    else return 1;
}

void Push(Stack &S,int e){
    Snode *p = new Snode;
    S->data = e;
    p->next = S;
     S = p;
}

void Pop(Stack &S,int &e){
    if(!StackEmpty(S)) cout << "The Stack is EMPTY!" << endl;
    else{
        Snode* p = S;
        e = p->next->data;
        S = S->next;
        delete(p);
    }
```

```
}

//DFS 算法的递归实现
void DFS(Graph G,int k,int v,int visited[]){
    //执行遍历并记录连通分支
    cout << "第" << k << "个连通分支:   " << G.vlist[v].data <<
endl;
    visited[v] = 1;
    int w = getfirst(G,v);
    while(w != -1){
        if(visited[w] == 0){
            DFS(G,k,w,visited);
        }
        w = GetNextVex(G,v,w);
    }
}

//DFS 算法的非递归实现（调用栈）
void DFS2 (Graph G,int k,int v,int visited[]) {
   Stack s;
   InitStack(s);
   while (StackEmpty(s)){
            Pop(s, v);
            if (visited[v] == 0)  {
            cout  <<  " 第 "  <<  k  <<  " 个 连 通 分 支 :     "  <<
G.vlist[v].data << endl;
            visited[v] = 1;
            }
        int w = getfirst(G, v);
        while( w != -1){
            if(!visited[w]) Push (s, w);
            w = GetNextVex(G, v, w);
        }
    }
}

//BFS 算法的非递归实现（调用队列）
void BFS(Graph G,int k,int v,int visited[]){
```

```
    //执行遍历 并记录连通分支
    cout << "第" << k << "个连通分支:   " << G.vlist[v].data << 
endl;
    visited[v] = 1;
    Queue q;
    InitQueue(q);
    EnQueue(q,v);
    while(QueueEmpty(q) == 0){
        DeQueue(q,v);
        int w = getfirst(G,v);
        while(w != -1){
            if(visited[w] == 0){
                cout << "第" << k << "个连通分支:   " << 
G.vlist[w].data << endl;
                visited[w] = 1;
                EnQueue(q,w);
            }
            w = GetNextVex(G,v,w);
        }
    }
}

//BFS 主函数
void GraphBFS(Graph &G){
    int visited[G.n];
    for(int i = 0;i < G.n;i++) visited[i] = 0;
    int k = 1;//用来表示第几个连通分支
    for(int i = 0;i < G.n;i++){
        if(visited[i] == 0){
            //深度优先遍历
            BFS(G,k,i,visited);
            k++;
        }
    }
}

//DFS 主函数
```

```
void GraphDFS(Graph &G){
    int visited[G.n];
    for(int i = 0;i < G.n;i++) visited[i] = 0;
    int k = 1;//用来表示第几个连通分支
    for(int i = 0;i < G.n;i++){
        if(visited[i] == 0){
            //深度优先遍历
            DFS(G,k,i,visited);
            k++;
        }
    }
}

int main(){
    Graph G;
    InitGraph(G);
    cout << "BFS 算法结果: " << endl;
    GraphBFS(G);
    cout << endl;
    cout << "DFS 算法结果: " << endl;
    GraphDFS(G);
}
```