南京信息工程大学__数据结构 I__实验(实习)报告

实验（实习）名称__图的建立和遍历__日期__2024.11.____得分_____指导教师_____

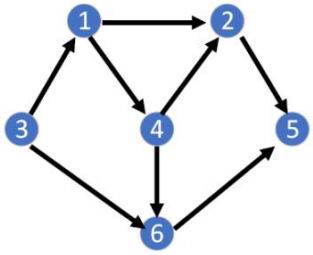
学院__计院__专业__计科_____

一、实验目的

1、掌握图的相关概念；
2、掌握图的逻辑结构和物理结构；
3、掌握图的深度优先遍历和广度优先遍历算法及实现。

二、实验内容与步骤

1、以邻接表为存储结构（数据结构的定义见教材 P163），编写程序根据输入的顶点数、边数、顶点信息、边信息等构造下图 G。



2、编写程序以键盘输入的顶点作为起始点，深度优先遍历图 G，输出深度优先遍历的顶点序列（即实现教材算法 7.4 和 7.5）。

（1）写出深度优先搜索算法的基本思路

基本思想：从起始顶点出发，沿着每一条路径向下搜索，直到到达图的最深处，然后回溯到上一个顶点，继续沿其他未访问过的路径深入，直到所有顶点都被访问。

基本步骤：选择一个起始顶点，并标记为已访问；访问当前顶点，并对与其相邻的尚未访问的顶点递归进行深度优先搜索；对每个未访问的邻接顶点，重复上述步骤，直到图中所有的顶点都被访问过为止。当没有未访问的邻接顶点时，回溯到上一个顶点，继续查找其他未访问的邻接顶点。递归结束条件：当所有顶点都被访问过或者图的所有路径都被搜索过时，DFS 结束。

3、编写程序以键盘输入的顶点作为起始点，广度优先遍历图 G，输出广度优先遍历的顶点序列（即实现教材算法 7.6）。

（1）写出广度优先搜索算法的基本思路

基本思想：从一个源节点出发，逐层访问图中的所有顶点。它通过探索每一层的所有顶点，先访问离源节点较近的顶点，再访问离源节点较远的顶点，直到图中所有顶点都被访问。

基本步骤：选择一个起始顶点，并将其标记为已访问；将起始顶点放入一个队列 Q 中（队列用于按层次顺序存储待访问的顶点）；从队列中取出一个顶点并访问它，对于当前访问的顶点，检查它所有的邻接顶点，如果邻接顶点没有被访问过，将其标记为已访问并加入队列。重复步骤 3 和 4，直到队列为空，即所有可以从起始顶点访问到的顶点都被遍历过。

4、编写程序利用拓扑排序，判段有向图 G 是否存在环。（拓扑排序算法见教材算法 7.12，其中 FindInDgree 函数功能需要自己编程实现）。

（1）写出解决本题的算法思路

拓扑排序的基本原理：如果一个有向图存在环，那么该图无法进行拓扑排序。拓扑排序

的实现通常使用入度的概念。如果一个图中所有顶点的入度都大于零，那么就存在环；否则可以进行拓扑排序。

基本步骤：计算入度：遍历图的邻接表，计算每个顶点的入度。拓扑排序：使用一个队列，首先将入度为零的顶点加入队列，然后逐个访问队列中的顶点，并减少其邻接点的入度。如果某个邻接点的入度变为零，则将其加入队列。继续这个过程，直到队列为空。检查是否存在环：如果拓扑排序的过程中访问的顶点数量少于图中顶点的数量，则说明图中存在环。

（2）实现程序源代码（文本）和多批测试数据的运行结果（截图）（本次实验的 4 个题目可放在一个程序中完成）。

实验代码：

```
#include<iostream>
#include<climits>
#include<cstdlib>
#include<queue>
#include<vector>
#include<algorithm>   // 用于排序
using namespace std;
#define INFINITY INT_MAX // 最大值无穷大
#define MAX_VERTEX_NUM 20 // 最大顶点个数
typedef enum{DG,DN,UDN,UDG}GraphKind;
typedef char VertexType; // 顶点为字符类型
typedef int InfoType; // 如果需要存储边的信息
// 边表节点类型
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
    InfoType *info;// 该弧相关信息的指针
}ArcNode;
// 顶点表节点类型
typedef struct VNode{
    int inDegree; // 顶点的入度
    VertexType data;// 顶点信息
    ArcNode *firstarc;// 指向第一条依附该顶点指针的弧的指针
}VNode,AdjList[MAX_VERTEX_NUM];
// 图的邻接表结构
typedef struct{
    AdjList vertices;// 邻接表
    int vexnum,arcnum;// 顶点数，边数
    int kind;// 图的种类标志
}ALGraph;
typedef enum{OK,ERROR}Status;
bool visited[MAX_VERTEX_NUM];
Status(*VisitFunc)(VertexType v);
//构造有向图的邻接表
Status CreateDG(ALGraph &G)
```

```
{
    int i,j;
    ArcNode *p;
    cout<<"请输入顶点和边数：";
    cin>>G.vexnum>>G.arcnum;
    cout<<"请输入顶点信息：";
    for(int i=0;i<G.vexnum;i++)
    {
        cin>>G.vertices[i].data;
        G.vertices[i].firstarc=NULL;
        G.vertices[i].inDegree=0;
    }
    cout<<"请输入每条边的信息（起始序号 终点序号）：";
    for(int k=0;k<G.arcnum;k++)
    {
        cin>>i>>j;
        if(i<1||i>G.vexnum||j<1||j>G.vexnum)
        {
            cout<<"输入顶点序号超过范围"<<endl;
            return ERROR;
         }
        p=(ArcNode *)malloc(sizeof(ArcNode));
        p->adjvex=j-1;
        p->info=NULL;
        p->nextarc=G.vertices[i-1].firstarc;
        G.vertices[i-1].firstarc=p;
    }
    // 排序每个顶点的邻接点
    for (int i = 0; i < G.vexnum; i++) {
        // 使用 vector 临时存储邻接点
        vector<int> adjacents;
        ArcNode* p = G.vertices[i].firstarc;
        while (p) {
            adjacents.push_back(p->adjvex);
            p = p->nextarc;
        }
        // 排序邻接点
        sort(adjacents.begin(), adjacents.end());
        // 重新链接排序后的邻接点
        G.vertices[i].firstarc = NULL;
        for (int j = adjacents.size() - 1; j >= 0; j--) {
            p = (ArcNode*)malloc(sizeof(ArcNode));
            p->adjvex = adjacents[j];
            p->nextarc = G.vertices[i].firstarc;
```

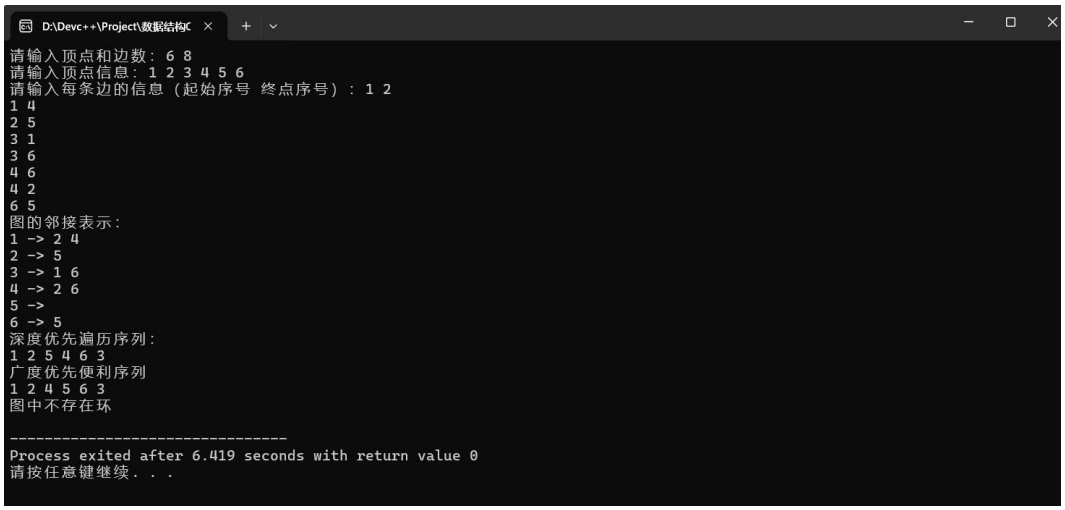
```
                G.vertices[i].firstarc = p;
            }
        }
        return OK;
}
//打印图的邻接表示
void PrintGraph(ALGraph &G){
        cout<<"图的邻接表示："<<endl;
        for (int i = 0; i < G.vexnum; i++) {
            cout << G.vertices[i].data << " -> ";
            ArcNode *p = G.vertices[i].firstarc;
            while (p) {
                cout << G.vertices[p->adjvex].data << " ";
                p = p->nextarc;
            }
            cout << endl;
        }
}
// 深度优先搜索递归函数
void DFS(ALGraph &G, int v) {
        visited[v] = true; // 标记顶点已访问
        cout << G.vertices[v].data << " "; // 访问顶点

        ArcNode *p = G.vertices[v].firstarc;
        while (p) {
            if (!visited[p->adjvex]) { // 如果该邻接点未被访问
                DFS(G, p->adjvex);       // 递归访问
            }
            p = p->nextarc;
        }
}
// 深度优先遍历图
void DFSTraverse(ALGraph &G) {
        // 初始化访问标记数组
        for (int i = 0; i < G.vexnum; i++) {
            visited[i] = false;
        }
        cout << "深度优先遍历序列：" << endl;
        for (int v = 0; v < G.vexnum; v++) {
            if (!visited[v]) { // 如果顶点未被访问
                DFS(G, v);
            }
        }
        cout << endl;
```

```
}
// 广度优先搜索递归函数
void BFS(ALGraph &G,int v){
    queue<int> q; //  使用队列来进行广度优先搜索
    visited[v]=true;
    cout<<G.vertices[v].data<<" ";
    q.push(v);
    while(!q.empty()){
        int u=q.front();
        q.pop();
        ArcNode *p=G.vertices[u].firstarc;
        while(p){
            int adjv=p->adjvex;
            if(!visited[adjv]){
                visited[adjv]=true;
                cout<<G.vertices[adjv].data<<" ";
                q.push(adjv);
            }
            p=p->nextarc;
        }
    }
}
// 广度优先遍历图
void BFSTraverse(ALGraph &G){
    for(int i=0;i<G.vexnum;i++){
        visited[i]=false;
    }
    cout<<"广度优先便利序列"<<endl;
    for(int v=0;v<G.vexnum;v++){
        if(!visited[v]){
            BFS(G,v);
        }
    }
    cout<<endl;
}
//拓扑排序判断图是否存在环
bool TopologicalSort(ALGraph &G){
    vector<int> inDegree(G.vexnum);
    queue<int> q;
    //将所有入度为 0 的顶点入队
    for(int i=0;i<G.vexnum;i++){
        inDegree[i]=G.vertices[i].inDegree;
        if(inDegree[i]==0){
            q.push(i);
```

```
            }
        }
        int count=0; //用于统计已访问的顶点数
        while(!q.empty()){
            int u=q.front();
            q.pop();
            count++;
            ArcNode* p=G.vertices[u].firstarc;
            while(p){
                int v=p->adjvex;
                inDegree[v]--;
                if(inDegree[v]==0){
                    q.push(v);
                }
                p=p->nextarc;
            }
        }
        if(count==G.vexnum){
            cout<<"图中不存在环"<<endl;
            return true;
        }
        else{
            cout<<"图中存在环"<<endl;
            return false;
        }
}
int main()
{
        ALGraph G;
        G.kind = DG; //  设置图的类型为有向图
        if (CreateDG(G) == OK) {
            PrintGraph(G);    //  打印图的邻接表
            DFSTraverse(G); //  深度优先遍历图
            BFSTraverse(G); //  执行广度优先遍历
            TopologicalSort(G); //  判断图是否有环
        } else {
            cout << "图创建失败！" << endl;
        }
        return 0;
}
/*6 8
1 2 3 4 5 6
1 2 1 4 2 5 3 1 3 6 4 6 4 2 6 5*/
```

运行结果：



三、实验心得（必写）

本次实验我围绕图的基本操作与算法重点学习了如何使用邻接表表示图，以及如何实现深度优先遍历（DFS）、广度优先遍历（BFS）和拓扑排序等基本算法。通过实验，我加深了对图的结构和遍历算法的理解，并掌握了如何在程序中实现这些算法。

图的表示：实验开始时，我首先学习了图的邻接表表示方法。邻接表通过为每个顶点存储与之相邻的顶点列表来表示图，节省了存储空间。构造邻接表时，除了存储顶点信息外，我还需要为每个顶点记录入度信息，这有助于拓扑排序算法的优化。

深度优先遍历（DFS）：在实现深度优先遍历时，我学习了如何通过递归来遍历图的每个顶点。DFS 的基本思路是沿着一条路径一直深入，直到无法继续深入时回溯到上一个节点并继续搜索其他未访问的节点。通过递归调用，我能够很好地实现这一过程，并成功遍历图中的所有顶点。这让我更清晰地理解了递归在图遍历中的应用。

广度优先遍历（BFS）：广度优先遍历与深度优先遍历有所不同，它是逐层遍历图中的顶点。BFS 使用队列来维护当前待访问的顶点，从起始顶点出发，先访问所有与其相邻的顶点，再访问与这些相邻顶点相连的顶点，直到遍历完整个图。在实现 BFS 时，队列的使用是一个关键点，它确保了顶点是按层次顺序访问的。

拓扑排序与有环判断：拓扑排序是针对有向图的一种特殊排序，它要求图中没有环，并且每个顶点的前驱都在它之前。通过计算图中每个顶点的入度，结合拓扑排序的过程，我能够判断出图中是否存在环。如果在拓扑排序过程中发现有节点无法被访问（即入度始终不为零），就说明图中存在环。在本次实验中，我成功实现了拓扑排序，并利用它判断了图中是否存在环。

当然，在实现这些算法时，我遇到了一些挑战，尤其是在实现邻接表的构造和排序时。由于每个顶点的邻接点可能没有按序号排序，因此我需要额外处理这些邻接点的排序问题。此外，拓扑排序的判断环的过程也需要仔细处理，特别是在入度为零的顶点出现多次时，如何保证正确的拓扑顺序。通过这些挑战，我提高了编程的思维能力和调试技巧，也更加熟悉了图的操作和算法实现。