《数据结构》上机报告

2020 年 11 月 26 日

姓名：马家昱 学号：1950509 班级：计科 1 班 得分：500

<table>
<tr><td>实验题目</td><td colspan="2">图</td></tr>
<tr><td>问题描述</td><td colspan="2">图是一种多对多的非线性数据结构，如果给图的每条边规定一个方向，那么得到的图称为有向图，相反，边没有方向的图称为无向图。带有权值的图成为网。</td></tr>
<tr><td rowspan="2">基本要求</td><td colspan="2">1. (p1)使用邻接矩阵与邻接表实现图的存储。
2. (p2)使用邻接矩阵实现对图的深度优先搜索和广度优先搜索。
3. (p3)使用邻接表实现对图的深度优先搜索和广度优先搜索。
4. (p4)实现图的关键路径。
5. (p5)实现图的拓扑排序。</td></tr>
<tr><td>已完成基本内容（序号）：</td><td>1，2，3，4，5</td></tr>
<tr><td rowspan="2">选做要求</td><td colspan="2">1. 程序要增添适当的注释，程序的书写要采用缩进格式。
2. 程序拥有一定的健壮性，对非法输入有相应的提示。</td></tr>
<tr><td>已完成选做内容（序号）</td><td>1，2</td></tr>
<tr><td>数据结构设计</td><td colspan="2">1. 第一题图的存储结构中，使用了二维数组表示邻接矩阵，邻接表则使用头结点与节点实现。其中头结点存储每个节点的基本信息，一般使用顺序结构，每个头结点后跟一链表表示其指向的其他节点。
2. 第二与第三题仍沿用第一题的数据结构。
3. 第四与第五题中，采用邻接表的存储结构，节点中增加了 inDegree，即每个节点的入度。</td></tr>
</table>

功能（函数）说明

节点与节点的初始化函数

```
class Node {//表节点
public:
    int adjvec;
    Node* nextarc;
    int info;
public:
    Node();
};

Node::Node(){
    adjvec = -1;
    info = 0;
    nextarc = NULL;
}

class HeadNode {//头结点
public:
    char data;
    Node* firstarc;
public:
    HeadNode();
};

HeadNode::HeadNode() {
    data = '\0';
    firstarc = NULL;
}
```

根据输入向邻接表与邻接矩阵中赋值

```
bool graph(bool isWeighted, bool isDirected,int nodeNum, int arcNum) {//生成图，参数分别为 有无权值 是否有向 节点个数与边的个数
    int matrix[100][100] = { 0 };//邻接矩阵
    HeadNode* headNode = (HeadNode*)malloc(nodeNum * sizeof(HeadNode));//邻接表首个头结点
    for (int i = 0; i < nodeNum; i++) {//输入节点
        cin >> headNode[i].data;
        headNode[i].firstarc = NULL;
    }
    for (int i = 0; i < arcNum; i++) {//输入边或弧
        char start, end;
        int p, q, weight;
        if (!isWeighted) {
            cin >> start >> end;
            for (p = 0; p < nodeNum; p++)//定位边的两个节点的下标
                if (headNode[p].data == start)
                    break;
            for (q = 0; q < nodeNum; q++)
                if (headNode[q].data == end)
                    break;
            matrix[p][q] = 1;//给邻接矩阵赋值
            Node* newNode = (Node*)malloc(sizeof(Node));//头插法修改邻接表
            newNode->adjvec = q;
            newNode->nextarc = headNode[p].firstarc;
            headNode[p].firstarc = newNode;
            if (!isDirected) {
                matrix[q][p] = 1;
                Node* newNode = (Node*)malloc(sizeof(Node));
                newNode->adjvec = p;
                newNode->nextarc = headNode[q].firstarc;
                headNode[q].firstarc = newNode;
            }
        }
```

邻接矩阵的深度优先搜索

```
void DFS(int* visited, int node, int nodeNum, int depth) {//从单个节点开始进行深度优先搜索
    if (!depth)
        cout << node;
    else cout << ' ' << node;
    visited[node] = 1;
    for (int j = 0; j < nodeNum; j++)
        if (matrix[node][j] == 1 && !visited[j])
            DFS(visited, j, nodeNum, depth + 1);
}

void DFSsearch(int nodeNum) {//对整个矩阵进行深度优先搜索
    int visited[MAX_NODE_NUM] = { 0 };
    for (int i = 0; i < nodeNum; i++)
        if (!visited[i]) {
            cout << '{';
            DFS(visited, i, nodeNum, 0);
            cout << '}';
        }
    cout << endl;
}
```

邻接矩阵的广度优先搜索

```
void BFS(int* visited, int node, int nodeNum) {//从单个节点开始进行广度优先搜索
    queue<int> Q;
    cout << node;
    visited[node] = 1;
    Q.push(node);
    while (!Q.empty()) {
        node = Q.front();
        Q.pop();
        for (int j = 0; j < nodeNum; j++)
            if (matrix[node][j] == 1 && !visited[j]) {
                cout << ' ' << j;
                visited[j] = 1;
                Q.push(j);
            }
    }
}

void BFSsearch(int nodeNum) {//对整个矩阵进行广度优先搜素
    int visited[MAX_NODE_NUM] = { 0 };
    for (int i = 0; i < nodeNum; i++)
        if (!visited[i]) {
            cout << '{';
            BFS(visited, i, nodeNum);
            cout << '}';
        }
}
```

邻接表的深度优先搜索

```
void DFS(int* visited, HeadNode* G, int node,int depth) {
    Node* p;
    if (depth == 0)
        cout << G[node].data;
    else cout << ' ' << G[node].data;
    visited[node] = 1;
    p = G[node].firstarc;
    while (p) {
        if (!visited[p->adjvec])
            DFS(visited, G, p->adjvec,depth+1);
        p = p->nextarc;
    }
}

void DFSTraverse(HeadNode* G, int nodeNum) {
    int visited[100] = {0};
    for (int i = 0; i < nodeNum; i++)
        if (!visited[i]) {
            cout << '{';
            DFS(visited, G, i,0);
            cout << '}';
        }
    cout << endl;
}
```

邻接表的广度优先搜索

```
void BFS(int* visited, HeadNode* G, int node) {
    Node* p;
    queue<int> Q;
    cout << G[node].data;
    visited[node] = 1;
    Q.push(node);
    while (!Q.empty()){
        node = Q.front();
        Q.pop();
        p = G[node].firstarc;
        while (p){
            if (!visited[p->adjvec])
            {
                cout <<' '<< G[p->adjvec].data;
                visited[p->adjvec] = 1;
                Q.push(p->adjvec);
            }
            p = p->nextarc;
        }
    }
}

void BFSTraverse(HeadNode* G, int nodeNum) {
    int visited[100] = { 0 };
    for (int i = 0; i < nodeNum; i++)
        if (!visited[i]) {
            cout << '{';
            BFS(visited, G, i);
            cout << '}';
        }
}
```

拓扑排序

```
bool topologicalSort(HeadNode* G, int nodeNum) {//拓扑排序
    stack<int> S;
    int cnt = 0;
    for (int i = 0; i < nodeNum; i++)//入度为零的节点入栈
        if (G[i].inDegree == 0)
            S.push(i);
    while (!S.empty()){
        int node = S.top();
        S.pop();
        cnt++;
        for (Node* p = G[node].firstarc; p; p = p->nextarc) {
            if (!(--G[p->adjvec].inDegree))//若减一后入度为0，入栈
                S.push(p->adjvec);
        }
    }
    if (cnt == nodeNum)//输出节点数等于总结点数，说明可以拓扑排序，无环
        return true;
    else return false;
}
```

关键路径

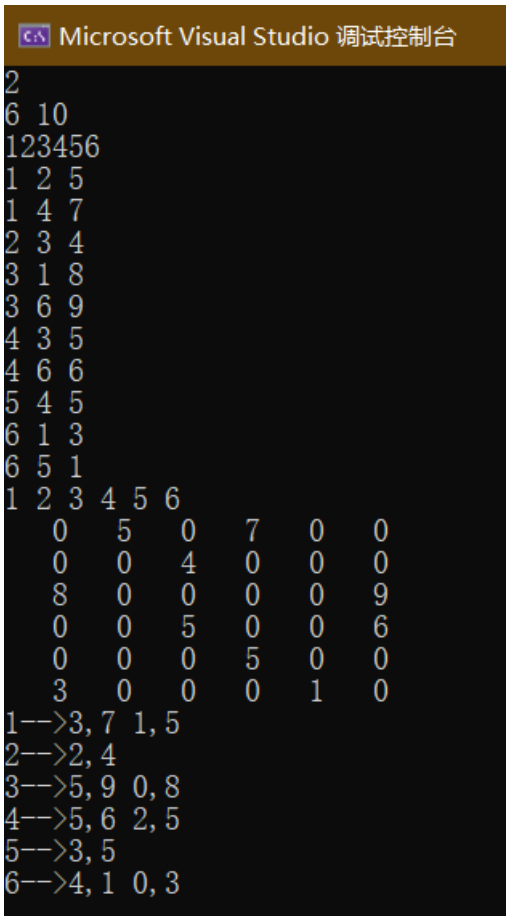
```cpp
bool criticalPath(HeadNode* G, int nodeNum, int* ve, stack<int>& T) {//生成关键路径
    if (!topologicalSort(G, nodeNum, ve, T))//若有环，则无法生成关键路径
        return false;
    cout << ve[nodeNum - 1]<<endl;
    int vl[MAX_NODE_NUM];
    for (int i = 0; i < nodeNum; i++)//初始化vl数组，为最长路径
        vl[i] = ve[nodeNum - 1];
    while (!T.empty()){//逆向拓扑序列求vl值
        int node = T.top();
        T.pop();
        for (Node* p = G[node].firstarc; p; p = p->nextarc)
            if ((vl[p->adjvec] - p->info) < vl[node])
                vl[node] = vl[p->adjvec] - p->info;
    }
    for(int i=0;i<nodeNum;i++)//若ee==el，则该活动为关键活动
        for (Node* p = G[i].firstarc; p; p = p->nextarc) {
            if (ve[i] == vl[p->adjvec] - p->info)
                cout << i + 1 << "->" << p->adjvec + 1 << endl;
        }
    return true;
}
```

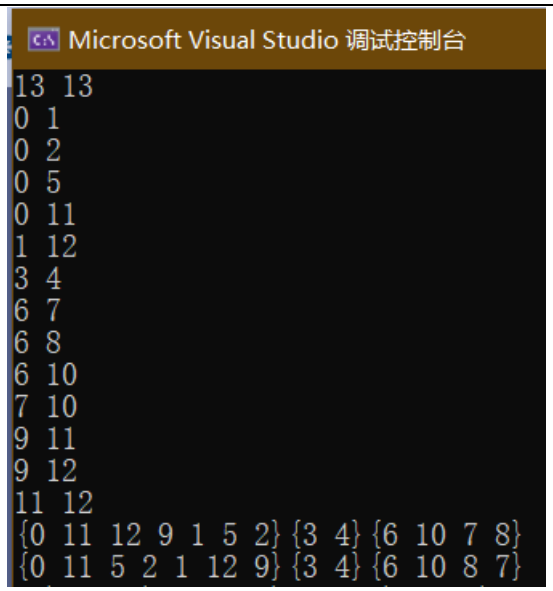
一笔画问题

```cpp
const int matrix[5][5] = {
    {0,1,1,0,1},
    {1,0,1,0,1},
    {1,1,0,1,1},
    {0,0,1,0,1},
    {1,1,1,1,0}
};//邻接矩阵

int visited[5][5] = {
    { 0,0,0,0,0 },
    { 0,0,0,0,0 },
    { 0,0,0,0,0 },
    { 0,0,0,0,0 },
    { 0,0,0,0,0 },
};//记录节点是否访问过

int order[10];//访问顺序
int cnt = 0;//可能的路径数
```

```
void printResult() {//打印单条路径
    cout << order[0]+1;
    for (int i = 1; i < 8; i++)
        cout << "->" << order[i] + 1;
    cout << endl;
}

void dfs(int nowNode, int depth) {//dfs搜索可能的路径
    for (int i = 0; i < 5; i++) {
        if (matrix[nowNode][i] && !visited[nowNode][i]) {
            order[depth] = i;
            visited[nowNode][i] = 1;
            visited[i][nowNode] = 1;
            if (depth == 8) {//层数为边数时，说明全部遍历
                printResult();
                cnt++;
            }
            else
                dfs(i, depth + 1);
            visited[nowNode][i] = 0;
            visited[i][nowNode] = 0;//还原
        }
    }
}
void oneStrokeDrawing() {//对每个节点dfs
    for (int i = 0; i < 5; i++) {
        order[0] = i;
        dfs(i, 1);
    }
    cout << "可能路径的数量是：" << cnt;
}



int main() {
    oneStrokeDrawing();
}
```

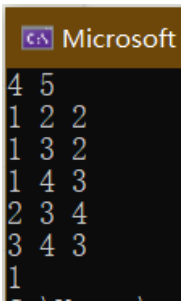
开发环境	Windows10 visual studio2019

调试分析	（运行结果截图） 1. 图的存储结构  2. 邻接矩阵遍历  3. 邻接表遍历



4. 关键路径

5. 拓扑排序



6. 圣诞老人房子一笔画

```
1->2->3->1->5->3->4->5
1->2->3->1->5->4->3->5
1->2->3->4->5->1->3->5
1->2->3->4->5->3->1->5
1->2->3->5->1->3->4->5
1->2->3->5->4->3->1->5
1->2->5->1->3->4->5->3
1->2->5->1->3->5->4->3
1->2->5->3->1->5->4->3
1->2->5->3->4->5->1->3
1->2->5->4->3->1->5->3
1->2->5->4->3->5->1->3
1->3->2->1->5->3->4->5
1->3->2->1->5->4->3->5
1->3->2->5->3->4->5->1
1->3->2->5->4->3->5->1 2->1->5->3->2->5->4->3
1->3->4->5->1->2->3->5 2->1->5->3->4->5->2->3
1->3->4->5->1->2->5->3 2->1->5->4->3->2->5->3
1->3->4->5->2->1->5->3 2->1->5->4->3->5->2->3
1->3->4->5->2->3->5->1 2->3->1->2->5->3->4->5
1->3->4->5->3->2->1->5 2->3->1->2->5->4->3->5
1->3->4->5->3->2->5->1 2->3->1->5->3->4->5->2
1->3->5->1->2->3->4->5 2->3->1->5->4->3->5->2
1->3->5->1->2->5->4->3 2->3->4->5->1->2->5->3
1->3->5->2->1->5->4->3 2->3->4->5->1->3->5->2
1->3->5->2->3->4->5->1 2->3->4->5->2->1->3->5
1->3->5->4->3->2->1->5 2->3->4->5->2->1->5->3
1->3->5->4->3->2->5->1 2->3->4->5->3->1->2->5
1->5->2->1->3->4->5->3 2->3->4->5->3->1->5->2
1->5->2->1->3->5->4->3 2->3->5->1->2->5->4->3
1->5->2->3->4->5->3->1 2->3->5->1->3->4->5->2
1->5->2->3->5->4->3->1 2->3->5->2->1->3->4->5
1->5->3->1->2->3->4->5 2->3->5->2->1->5->4->3
1->5->3->1->2->5->4->3 2->3->5->4->3->1->2->5
1->5->3->2->1->3->4->5 2->3->5->4->3->1->5->2
1->5->3->2->5->4->3->1 2->5->1->2->3->4->5->3
1->5->3->4->5->2->1->3 2->5->1->2->3->5->4->3
1->5->3->4->5->2->3->1 2->5->1->3->4->5->3->2
1->5->4->3->1->2->3->5 2->5->1->3->5->4->3->2
1->5->4->3->1->2->5->3 2->5->3->1->2->3->4->5
1->5->4->3->2->1->3->5 2->5->3->1->5->4->3->2
1->5->4->3->2->5->3->1 2->5->3->2->1->3->4->5
1->5->4->3->5->2->1->3 2->5->3->2->1->5->4->3
1->5->4->3->5->2->3->1 2->5->3->4->5->1->2->3
2->1->3->2->5->3->4->5 2->5->3->4->5->1->3->2
2->1->3->2->5->4->3->5 2->5->4->3->1->2->3->5
2->1->3->4->5->2->3->5 2->5->4->3->1->5->3->2
2->1->3->4->5->3->2->5 2->5->4->3->2->1->3->5
2->1->3->5->2->3->4->5 2->5->4->3->2->1->5->3
2->1->3->5->4->3->2->5 2->5->4->3->5->1->2->3
2->1->5->2->3->4->5->3 2->5->4->3->5->1->3->2
2->1->5->2->3->5->4->3 可能路径的数量是：88
```

心得体会	（对整个实验过程做出总结，对重要的算法做出性能分析。） 对于有 n 个节点与 e 条边或弧的图形结构来说： 1. 使用邻接表进行深度或广度优先搜索时，时间复杂度为 O（n+e）。 2. 使用邻接矩阵进行深度或广度优先搜索时，时间复杂度为 O（n^2）。 3. 拓扑排序与关键路径的时间复杂度均为 O（n+e）。 4. 一笔画路径中单个节点的时间复杂度为 O（n^2），由于要对所有节点进行 dfs，所以总时间复杂度为 O（n^3）。 心得体会：通过本次作业，我对图这一陌生的数据结构有了基本的认识，学会使用几种基本的数据结构对图进行存储，并使用不同的方法遍历。同时掌握了最小生成树、拓扑排序、关键路径等图的实际应用问题的基本算法。