组号: 18

图片包含 游戏机, 画

描述已自动生成

上海大学计算机工程与科学学院

**实 验 报 告**

（数据结构2）

学 期：2022-2023年春季

组 长： 郑力铖

学 号： 21122873

指导教师： 朱能军

成绩评定： （教师填写）

二〇二三年四月二日

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **小组信息** | | | | |
| 登记序号 | 姓名 | 学号 | 贡献比 | 签名 |
| 1 | 郑力铖 | 21122873 | 33 |  |
| 2 | 闫城锦 | 21122908 | 34 |  |
| 3 | 张思祺 | 21122909 | 33 |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **实验列表** | | |
| 实验一 | （熟悉上机环境、进度安排、评分制度；分组） |  |
| 实验二 | (*有向网的邻接矩阵验证及拓展*) | P |
| 实验三 | (*实验题目*) |  |
| 实验四 | (*实验题目*) |  |
| 实验五 | (*实验题目*) |  |

实验二

一、**实验题目**

有向网的邻接矩阵验证及拓展

二、**实验内容**

模仿无向图的邻接矩阵类模板，完成（带权：非负）有向网的邻接矩阵类模板的设计与实现。要求实现图的基本运算（如增加删除顶点和弧等），并增加如下成员函数：

1. CountOutDegree(v)，统计顶点 v的出度；

2. CountInDegree(v)，统计顶点 v的入度；

3. ShortestPath(v1,v2)，求两个顶点之间最短路径；

三、**解决方案**

**1、算法设计（*主要描述数据结构、算法思想、主要操作、用例分析、改进方法等*）**

**1.1 统计入度与出度**

遍历邻接矩阵中的节点，找到需要找的顶点，然后对边进行统计，计数，返回出度或入度。

**1.2 求两个顶点间的最短路径**

Dijkstra算法适用于有向图或者无向图中没有负边权的情况，它通过维护一个已访问的顶点集合和一个未访问的顶点集合，不断更新起点到每个未访问顶点的最短路径，最终得到起点到终点的最短路径。

Floyd算法则适用于有向图或者无向图中有负边权的情况，它对每对顶点之间的距离进行递推求解，最终得到任意两点之间的最短路径。

Bellman-Ford算法是一种单源最短路径算法，可以用于处理带有负权边的图。从源点开始，对图中的所有边进行|V|-1轮松弛操作，其中|V|是图中节点的数量。每轮松弛操作都会更新当前所有节点到源点的最短距离，因此最后得到的就是源点到所有节点的最短距离。

**2、源程序代码（*要求有必要注释、格式整齐、命名规范，利于阅读*）**

**2.1 CountOutDegree(v)，统计顶点 v的出度**

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::CountOutDegree(ElemType v) {

int v1 = 0;

**// 遍历图，找到顶点。**

for (; v1 < vexNum; v1++) {

if (vertexes[v1] == v) break;

}

if (v1 == vexNum) {

throw Error("查询节点不存在!");

}

**// 统计出度**

int s = 0;

for (int i = 0; i < vexNum; ++i) {

if (arcs[v1][i] != -1) s++;

}

return s;

}

**2.2 CountInDegree(v)，统计顶点 v的入度**

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::CountOutDegree(ElemType v) {

int v1 = 0;

**// 遍历图，找到节点。**

for (; v1 < vexNum; v1++) {

if (vertexes[v1] == v) break;

}

if (v1 == vexNum) {

throw Error("查询节点不存在!");

}

**// 统计出度**

int s = 0;

for (int i = 0; i < vexNum; ++i) {

if (arcs[i][v1] != -1) s++;

}

return s;

}

**2.3 ShortestPath\_DJ(e1, e2)，Dijkstra算法统计e1-e2的最短路**

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::ShortestPath\_DJ(ElemType &e1, ElemType &e2) {

int v1 = -1, v2 = -1;

for (int i = 0; i < vexNum; ++i) {

if (vertexes[i] == e1) v1 = i;

if (vertexes[i] == e2) v2 = i;

}

if (v1 == -1 || v2 == -1) {

throw Error("输入的顶点不全存在！");

}

int dis[vexNum];

int visited[vexNum];

int to\_visit;

for (int i = 0; i < vexNum; ++i) {

if (i == v1) dis[i] = 0;

else dis[i] = DEFAULT\_INFINITY;

visited[i] = 0;

}

**// 循环，直到访问所有顶点**

for (int visited\_num = 0; visited\_num < vexNum; visited\_num++) {

int mindis = DEFAULT\_INFINITY;

**// 找到未访问的距离最小的顶点**

for (int i = 0; i < vexNum; ++i) {

if (visited[i] == 0 && mindis >= dis[i]) {

to\_visit = i;

mindis = dis[i];

}

}

**// 如果找到了更短的路径，则更新相邻顶点的距离**

for (int i = 0; i < vexNum; ++i) {

if (arcs[to\_visit][i] != -1 && visited[i] == 0 && (dis[to\_visit] + arcs[to\_visit][i]) < dis[i]) {

dis[i] = dis[to\_visit] + arcs[to\_visit][i];

}

}

visited[to\_visit] = 1;

}

return dis[v2];

}

**2.4 ShortestPath\_Floyd(e1, e2)，Floyd算法统计e1-e2的最短路**

**// 使用Floyd算法计算最短路径**

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::ShortestPath\_Floyd(ElemType &e1, ElemType &e2) {

**// 查找目标顶点在顶点数组中的下标**

int v1 = -1, v2 = -1;

for (int i = 0; i < vexNum; ++i) {

if (vertexes[i] == e1) v1 = i;

if (vertexes[i] == e2) v2 = i;

}

if (v1 == -1 || v2 == -1) {

throw Error("输入的顶点不全存在！");

}

**// 初始化路径矩阵**

int sp[vexNum][vexNum];

for (int i = 0; i < vexNum; ++i) {

for (int j = 0; j < vexNum; ++j) {

sp[i][j] = (arcs[i][j] == -1 ? DEFAULT\_INFINITY : arcs[i][j]);

}

}

**// 计算最短路径**

for (int mid = 0; mid < vexNum; ++mid) {

for (int start = 0; start < vexNum; ++start) {

for (int end = 0; end < vexNum; end++) {

if (sp[start][end] > sp[start][mid] + sp[mid][end]) {

sp[start][end] = sp[start][mid] + sp[mid][end];

}

}

}

}

return sp[v1][v2];

}

**2.4 limitedPath\_Ford(e1, e2)，Bellman-Ford算法统计e1-e2在限制条件下的最短路**

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::limitedPath\_ford(ElemType &e1, ElemType &e2, int limits) {

int v1 = -1, v2 = -1;

for (int i = 0; i < vexNum; ++i) {

if (vertexes[i] == e1) v1 = i;

if (vertexes[i] == e2) v2 = i;

}

if (v1 == -1 || v2 == -1) {

throw Error("输入的顶点不全存在！");

}

struct Edge {

int a, b, c;

} edges[arcNum + 5];

int num = 0;

for (int i = 0; i < GetVexNum(); i++)

for (int j = 0; j < GetVexNum(); j++) {

if (arcs[i][j] != -1) {

edges[num++] = {i, j, arcs[i][j]};

}

}

**// 初始化最短路径数组和上一次迭代的最短路径数组**

int dist[GetVexNum() + 5];

memset(dist, DEFAULT\_INFINITY, sizeof(dist));

int last[GetVexNum() + 5];

memset(last, DEFAULT\_INFINITY, sizeof(last));

dist[v1] = 0;

**// 使用Bellman-Ford算法求解有限制的最短路径**

for (int i = 0; i < limits; i++) {

memcpy(last, dist, sizeof dist);

for (int j = 0; j < arcNum; j++) {

auto e = edges[j];

dist[e.b] = min(dist[e.b], last[e.a] + e.c);

}

}

if(dist[v2] == DEFAULT\_INFINITY){

cout << "未找到符合条件的路径" << endl;

return -1;

}

return dist[v2];

}

**2.4 SECOND\_ShortestPath\_dfs\_1 (e1, e2)，深度优先搜索e1-e2的次短路**

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::SECOND\_ShortestPath\_dfs\_1(ElemType &e1, ElemType &e2) {

int v1 = -1, v2 = -1;

for (int i = 0; i < vexNum; ++i) {

if (vertexes[i] == e1) v1 = i;

if (vertexes[i] == e2) v2 = i;

}

if (v1 == -1 || v2 == -1) {

throw Error("输入的顶点不全存在！");

}

for (int i = 0; i < GetVexNum(); i++) {

if (i != v1) SetTag(v1, UNVISITED);

else SetTag(v1, VISITED);

}

return dfs\_1(v1, v2, 0);

}

template<class ElemType>

int AdjMatrixUndirGraph<ElemType>::dfs\_1(int v1, int v2, int flag) {

int pathlen = DEFAULT\_INFINITY;

int secondpathlen = DEFAULT\_INFINITY;

if (v1 == v2) return 0;

for (int i = 0; i < GetVexNum(); i++) {// 遍历所有节点

int k = DEFAULT\_INFINITY;

if (GetTag(i) == UNVISITED && arcs[v1][i] != -1) {// 当前节点未被访问过且与v1相邻

SetTag(i, VISITED);

k = arcs[v1][i] + dfs\_1(i, v2, 1);// 递归计算从当前节点i到终点v2的路径长度

SetTag(i, UNVISITED);

}

pathlen = min(pathlen, k);// 更新当前路径长度为min(当前路径长度，临时路径长度)

if (k > pathlen) secondpathlen = min(secondpathlen, k);// 如果临时路径长度比当前路径长度大，则更新次短路径长度为min(次短路径长度，临时路径长度)

}

for (int i = 0; i < GetVexNum(); i++) {

SetTag(v1, UNVISITED);

}

if (flag == 0 && secondpathlen == DEFAULT\_INFINITY)

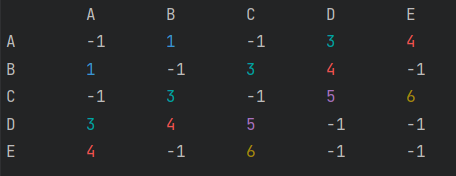
cout << "无次短路!" << endl;

else if (flag == 0 && secondpathlen != DEFAULT\_INFINITY)

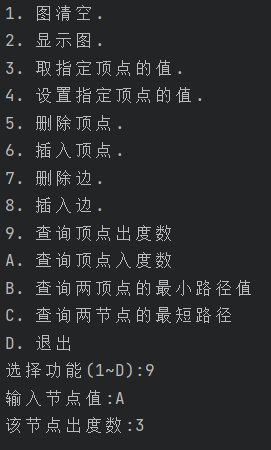
cout << "次短路:" << secondpathlen << endl;

return pathlen;

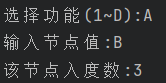
}

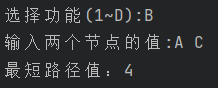
**3、实验结果（*展示实验结果、测试情况、结果分析等*）**

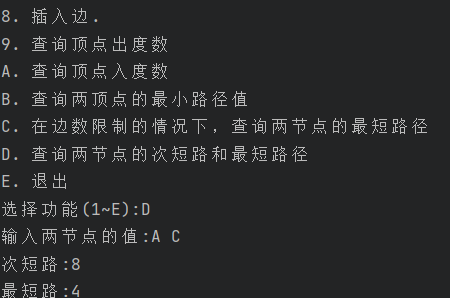
初始化的领接矩阵如图。



入度输出正确

****

出度输出正确

最小路径值输出正确

次短路输出正确

**4、算法分析（*对算法空间、时间效率进行必要分析，可能的改进建议等*）**

**4.1 计算入度与出度**

先遍历找到节点，耗时O(N)，后计数，总时间复杂度O(N)。

**4.2 两个顶点间的最短路径**

Dijkstra算法的时间复杂度为O(ElogV)，其中V表示顶点数，E表示边数。

Floyd算法的时间复杂度为O(V^3)，其中V表示顶点数。因为需要遍历每个顶点作为中间点的情况，所以时间复杂度为O(V^3)。

Bellman-Ford算法的时间复杂度为O(V\*E)。因为需要进行V-1轮操作，每轮操作需要遍历所有的边，因此时间复杂度是V\*E。

**5、总结与心得（*主要描述实验过程中存在的问题、原因、解决方法、收获、对实验内容的其他应用思考等*）**

有组员在编写成员函数中使用了如下语句：

memset(dist, DEFAULT\_INFINITY, sizeof(dist));

原先DEFAULT\_INFINITY在定义时，初始化为10000。虽然在自然语言中能理解其用意，但在memset函数的实现中，原数组并不能真正化为十进制的10000，进而导致后续程序错误。

因此修改原宏定义为：

#define DEFAULT\_INFINITY 0x3f3f3f3f

因DEFAULT\_INFINITY仅用于初始化dist数组，其数据类型为int，因此可以设置为该值，该值足够大，且在后续能够正确进行比较，较为保险。

四、**分工说明**（*小组成员具体分工和完成情况*）

郑力铖：统计入度、出度、报告撰写

张思祺：Floyd算法，代码汇总

闫城锦：Dijkstra算法，Bellman-Ford算法，代码调优