组号: 1

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：2021-2022 年春季

组　　长：李昀哲

学　　号：20123101

指导教师：朱能军

成绩评定：

二〇二二年 3 月 25 日

小组信息				
登记序号	姓名	学号	贡献比	签名
72	李昀哲	20123101	25%	
21	唐铭锋	20121489	25%	
20	刘沛根	20121483	25%	
22	李正宇	20121517	25%	

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*	✓
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. SecShortestPath(v1,v2)，求两个顶点之间的次短路径；
4. hasCycle()，判断有向图是否存在环。

三、解决方案

1、算法设计

在用邻接矩阵作为存储表示的有向网的类模板声明中，数组 vertexes 用以存放图中的顶点信息；还有一个作为邻接矩阵使用的二维数组 arcs[][]用来表示图中顶点之间的关系，其中：

$$arcs[i][j]=\begin{cases}W(i,j):i\neq j,且(i、j)\in E\\ \infty:i\neq j,且(i、j)\notin E\\ 0:i=j\end{cases}$$

W(i,j)表示两个顶点之间弧的权重，类的成员 infinity 用来表示无穷大的值。矩阵元素的个数取决于顶点个数，与边数无关。vexNum,vexMaxNum 和 arcNum 三个数据成员，分别记录图中当前顶点数目，允许的顶点最大数目，和边数；标志数组 tag[]，用来记录顶点的访问状况。

(1) 统计顶点 v 的出度:通过统计 v 所在行，不为 0 和 infinity 元素的个数即可得到顶点 v 的入度。

(2) 统计顶点 v 的入度:通过统计 v 所在列，不为 0 和 infinity 元素的个数即可得到顶点 v 的入度。

(3) 求两个顶点之间的次短路径：

通过对求最短路径长度的 dijkstra 算法进行修改，实现求次短路径的算法。

在 dijkstra 算法中，通过已经确定的最短路径向前推进，更新 dist[]来求得更多的最短路径。而在求次短路径的算法中，将 dist[]数组中每次被替换的最短路径的值保留下来，保存在另一个次短路径长度数组 dist2[]中。

当最短路径更新完毕后，所得到的次短路径长度的数组中的最小值一定是所对应的顶点的次短路径的长度，记为 L。因为次短路径的长度可以拆分为起点到某个顶点的最短路径加上该点到终点的最短路径，或起点到某个顶点的次短路径加上该点到终点的最短路径。而当用 dijkstra 算法求完最短路径时,可以理解为已经算过了所有的起点到某个顶点的最短路径加上该点到终点的最短路径，因此若目前还有未求到的次短路径长度，那么其次短路径长度只能由其他的次短路径求得，而 L 又是目前最短的次短路径，因此 L 被确定。然后通过已经确定次短路径再通过 dijkstra 算法向前推进，就可求得所有的次短路径。

具体实现过程中创建了以下几个数组：min 是最短路径长度，sec 是次短路径长度，flag 是标记最短路径是否找到，flag_sec:标记次短路径是否找到，pre_point：最短路径的前驱节点，sec_pre_point：次短路径的前驱节点，sec_pre_type：存储当前节点的次短路径的前驱节点走的是次短还是最短路径。

一开始在求最短路径长度的过程中，更新 min 和 flag，并保留第二短的路径存放在 sec 中。然后，用 sec 再进行一次 dijkstra 算法，更新 sec 和 flag_sec，即可实现求次短路径长度。在过程中要将次短路径的前驱节点记录在 sec_pre_point 中，将次短路径的前驱节点走的是次短还是最短路径记录在 sec_pre_type 中。用于在算出次短路径长度后回推次短路径。

(4) 判断有向图是否存在环：

方法 1：用 DFS 遍历图

从一顶点开始，进行 DFS 遍历，对访问到的顶点对其 tag 进行标记；当再次即将访问到已被访问到的顶点时，说明有环。

因为可能有多个连通分支因此需要对每个顶点进行遍历。

若 DFS 整个图后，若未发现，则无环。

方法 2：拓扑排序思想

如果一个顶点是某个环的一部分，那么该顶点的入度一定不为 0，若要判断是否有环，可以将那些一定不能构成环顶点删去。在删去后可能会再次出现入度为 0 的顶点，重复该过程直至无法再删去顶点。若仍有顶点存在说明该图有环，若所有顶点都被删去，说明该图无环。

在代码实现过程中，若删去图中的顶点则需要对图先进行备份，为了节省空间与时间。用一个队列记录入度为 0 的顶点，每次有顶点入队计数器 cnt++，用一个 vector 存储与那些入度为 0 的顶点相邻的顶点，另一个 vector 存储与那些入度为 0 的顶点相邻的顶点的入度。每当一个入度为 0 的顶点出队时，与其相邻的顶点的入度减一。若该顶点的入度减为 0，则入队，cnt++，与其相邻的顶点及其入度进入 vector。重复该过程，直至队列为空。若 cnt 与顶点个数相等，说明所有顶点都被删去，图中无环。反之有环。

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*）

（1）统计顶点 v 的出度

```
template<class ElemType>
int AdjMatrixdirNetwork<ElemType>::CountOutDegree(int v) const {
    int out_degree = 0;                 // 记录出度
    if (v < 0 || v >= vexNum)
        throw Error( mes: "v1不合法!");    // 抛出异常
    for (int u = 0; u < vexNum; u++)
        if(arcs[v][u] != infinity && arcs[v][u] != 0 )  out_degree++;
    return out_degree;
}
```

（2）统计顶点 v 的入度

```
template<class ElemType>
int AdjMatrixdirNetwork<ElemType>::CountInDegree(int v) const {
    int in_degree = 0;                  // 记录入度
    if (v < 0 || v >= vexNum)
        throw Error( mes: "v1不合法!");    // 抛出异常
    for (int u = 0; u < vexNum; u++)
        if(arcs[u][v] != infinity && arcs[u][v] != 0)  in_degree++;
    return in_degree;
}
```

（3）求两个顶点之间的次短路径

```cpp
template<class ElemType>
void AdjMatrixdirNetwork<ElemType>::SecShortestPath(int start, int end) {
    vector<int> min_distance,pre_point,sec_min_distance,pre_point_sec;
    bool *flag_min = new bool [vexNum],*flag_sec = new bool[vexNum];
    bool *pre_point_sec_type = new bool[vexNum]; // 次短路径前驱顶点是次短或最短(false为最短，true为次短)
    for(int i=0;i<vexNum;++i){
        flag_min[i]=false;
        flag_sec[i]=false;
        pre_point_sec_type[i]=false;
        min_distance.emplace_back(arcs[start][i]);    // 初始化距离
        pre_point.emplace_back(start);  // 初始化前驱顶点为0
        sec_min_distance.emplace_back(infinity);
        pre_point_sec.emplace_back(start);
    }
    pre_point[start] = -1;
    flag_min[start]=true, min_distance[start]=0; // 初始化起点
    for(int i=1;i<vexNum;++i){     // 循环n-1次（除了自己）
        int min_distance_from_start(infinity),vex(0);   // 到“此点”的最短路径，“此点”
        for(int j=0;j<vexNum;++j)   // 找到距离最近的点（j）
            if(!flag_min[j] && min_distance[j] < min_distance_from_start){
                min_distance_from_start=min_distance[j];
                vex=j;
            }
        flag_min[vex] = true;   // 标记已得到从start到point的最短路径
        for(int j=0;j<vexNum;++j){
            if(min_distance_from_start == infinity) continue;   // 防止不可达
            int start2next_distance = arcs[vex][j] == infinity ? infinity : min_distance_from_start + arcs[vex][j]; // 防止溢出
            if(!flag_min[j] && start2next_distance < min_distance[j]){

                pre_point_sec_type[j] = false;  // 次短路径前驱为最短路径
                sec_min_distance[j] = min_distance[j];
                pre_point_sec[j] = pre_point[j];
                min_distance[j] = start2next_distance;    // 若次路径更短则更新最短距离和前驱点
                pre_point[j] = vex;
            }else if(start2next_distance < sec_min_distance[j] && start2next_distance > min_distance[j]){   // 筛选第二长的
                pre_point_sec_type[j] = false;  // 次短路径前驱为最短路径
                sec_min_distance[j] = start2next_distance;
                pre_point_sec[j] = vex;
            }
        }
    }
    for(int i=0;i<vexNum;++i){  // 循环n次，处理次短路径
        int min_distance_from_start(infinity),vex(0);   // 到“此点”的最短的次短路径，“此点”
        for(int j=0;j<vexNum;++j)   // 找到距离最近的点（j）
            if(!flag_sec[j] && sec_min_distance[j] < min_distance_from_start){
                min_distance_from_start=sec_min_distance[j];
                vex=j;
            }
        flag_sec[vex] = true;   // 标记已得到从start到point的最短的次短路径
        for(int j=0;j<vexNum;++j){
            if(min_distance_from_start == infinity) continue;
            int start2next_distance = arcs[vex][j] == infinity ? infinity : min_distance_from_start + arcs[vex][j]; // 防止溢出
            if(!flag_sec[j] && start2next_distance < sec_min_distance[j]){
                pre_point_sec_type[j] = true;  // 次短路径前驱为次短路径
                sec_min_distance[j] = start2next_distance;
                pre_point_sec[j] = vex;
            }
```

```
        }
    }
    if(sec_min_distance[end]==infinity) // 输出
        cout<<"not found second shortest path"<<endl;
    else{
        cout<<"长度为 "<<sec_min_distance[end]<<endl<<vertexes[end];
        bool now_is_shortest_path=false;  // 当前顶点走的是否为最短路径
        for(int pre=pre_point_sec[end],tail=end; pre != -1;){
            cout<<" <- "<<vertexes[pre];
            if(!now_is_shortest_path && pre_point_sec_type[tail]){
                tail=pre;
                pre = pre_point_sec[pre];
            }
            else {
                tail=pre;
                pre = pre_point[pre];
                now_is_shortest_path=true;
            }

        }
        // cout<<" <- "<<vertexes[start]<<endl;
    }
    delete[] flag_min;
    delete[] flag_sec;
    delete[] pre_point_sec_type;
}
```

（4）判断有向图是否存在环

1. 深度优先遍历

```
template<class ElemType>
bool AdjMatrixdirNetwork<ElemType>::CycleDFS(int v, std::vector<int> &output)
{
    /// 深度优先遍历，求回路
    /// 从v顶点开始，依次访问并DFS与之相连的顶点，访问到的顶点对其tag进行标记，当再次即将访问到已被访问到的顶点时，即：有环
    /// 若DFS整个图后，未发现，则无环。
    /// 对于删除顶点后的图，只能处理删除最后一个顶点的情况
    /// 时间复杂度：O(n*n)，空间复杂度：O(n)
    bool flag = false;              // 用于标记是否有环
    int pre_vex = v;                // 用于记录上一个结点
    SetTagVisited(v);               // 将访问到的结点标记为Visited
    for(int w = FirstOutAdjVex(v); w != -1; w = NextOutAdjVex(v, w)){    // 遍历与v点相连的顶点
        if (GetTag(w) == VISITED && w != pre_vex){                 // 若与v相连的顶点已经被访问过，且不是父顶点，则是回路的头（尾）
            output.emplace_back(w);                               // 将头（尾）push_back
            return true;
        }
        else if (GetTag(w) == UNVISITED){                         // 若未访问过

            flag = CycleDFS(w, &output);                          // 对与v相连的第一个点进行DFS
            if(flag && output[0] == 0){                           // 用于回溯时记录回路尾顶点是否被访问过，若标记为1，则跳过

                output.emplace_back(w);                           // 回溯时push_back经过的顶点
                if(w == output[1])                                // 若再次回溯到相同的结点，则标记output的第一个元素，标记为1
                    output[0] = 1;
            }
            return flag;
        }
    }
    return false;
}
```

2. 拓扑排序

```
template<class ElemType>
bool AdjMatrixdirNetwork<ElemType>::Cycle(){
    /// 思路：参考拓扑排序
    /// 将所有入度为0的点入队；每次从队列中pop一个顶点，直至为空；
    /// 遍历所有与pop出来这个顶点相连的顶点，并将相连顶点入度减一，若减一后入度为0，入队。
    /// 若最终cnt == 顶点个数，说明所有顶点都被访问到，说明没cycle，否则说明有顶点入度不为1.
    int cnt = 0;
    queue<int> queue;
    std::vector<int> in_degree,linked_vex;
    for(int vex = 0; vex < vexNum; vex++){          // 算各顶点入度
        in_degree.push_back(this->CountInDegree(vex));
        if(in_degree[vex] == 0)
            queue.push(vex);
    }
    while(!queue.empty()){
        linked_vex.clear();
        int front_elem = queue.front();
        for(int i = 0; i < vexNum; i++){          // 找出与pop顶点相连的顶点，push入linked_vex
            if(arcs[front_elem][i] != infinity && front_elem != i)
                linked_vex.push_back(i);
        }
        queue.pop();
        cnt ++;
        for(int i : linked_vex)                     // 遍历相连结点，删边
            if (--in_degree[i] == 0)              // 若删边后，结点入度为0，push入队
                queue.push(i);
    }
    if(cnt == vexNum)   return false;
    else    return true;
}
```

3、实验结果

```
1. 图清空.
2. 显示图.
3. 删除顶点.
4. 插入顶点.
5. 删除边.
6. 插入边.
7. 求顶点的出度.
8. 求顶点的入度.
9. 是否有环.
0. 求次短路径
a. 退出
选择功能(1~7):2
  A       B       C       D       E
A       0       5       5       inf     10
B       inf     0       100     inf     8
C       inf     inf     0       4       5
D       inf     inf     inf     0       2
E       inf     1       inf     inf     0
```

```
选择功能(1~7):7

输入求出度的顶点的值:A
3

选择功能(1~7):8

输入求入度的顶点的值:A
0

选择功能(1~7):9
DFS求回路: 有环 环为:  B -> C -> D -> E -> B
拓扑排序求回路: 有环
```

```
选择功能(1~7):0
请输入起点和终点:AE
长度为 11
E <- D <- C <- A

选择功能(1~7):0
请输入起点和终点:BC
长度为 109
C <- B <- E <- B
```

```
选择功能(1~7):0
请输入起点和终点:AC
长度为 105
C <- B <- A

选择功能(1~7):0
请输入起点和终点:EA
not found second shortest path
```

实验现象与预想的实验现象一致。

4、算法分析

1. CountOutDegree 函数的时间复杂度为 O(n),空间复杂度为 O(1);
2. CountInDegree 函数的时间复杂度为 O(n),空间复杂度为 O(1);
3. SecShortestPath 函数的时间复杂度为 $O(n^2)$,空间复杂度为 O(n);
4. CycleDFS 函数的时间复杂度为 $O(n^2)$, 空间复杂度为 O(n);

CycleDFS 函数在功能实现时因为可能存在连通分支,因此遍历了所有的顶点求环，但是其实这是在大部分情况下连通分支数远小于顶点数，若要改进可以在类中先记录图中有哪些连通分支再进行计算。

Cycle 函数的时间复杂度为 $O(n^2)$, 空间复杂度为 O(n);

5、总结与心得

在本次实验的过程中我们小组通过讨论，实践，查阅资料等方法完成了实验任务。通过实验，我们对于邻接矩阵的基本概念以及其基本功能的实现有了更深入的理解。此外，我们还学习拓扑排序，dijkstra 算法等算法并运用到了解决问题中。而在本次实验中我们遇到的最大的难题，就是对次短路径的求解，通过查阅大量的资料，并且经过了我们的思考讨论后，我们最终选择了通过运用 dijkstra 算法的思想来解决次短路径的问题。

四、分工说明

李昀哲：算法设计，代码编写，ppt 制作

唐铭锋：算法设计，代码编写，撰写报告

刘沛根：算法设计，代码编写，ppt 制作

李正宇：算法设计，代码编写，撰写报告