作业 HW4 试验报告

姓名: 范潇　学号: 2254298

2023年 12月 1日

1　涉及数据结构和相关背景

图的主要存结构为邻接矩阵和邻接表。

对于邻接矩阵实现的图，其结构体由一个顶点表，和邻接矩阵组成，分别用于记录各顶点信息，以及表示各个顶点之间的弧。

对于邻接表实现的图，顶点数组中的每个元素代表一个顶点，其中设有一个指针域，指向以该节点为尾的第一条弧。创建这样一个邻接表的算法复杂度的的组成主要为查找结点的位置，如果结点所带的信息就是它的下标，那么由于随机存储以及头插法的复杂度都是常量，总的时间复杂度是$O(n+e)$。如果不是，那么就需要遍历结点数组来找到对应的结点，总的时间复杂度是 $O(n*e)$。

性质	邻接矩阵	邻接表
唯一性	唯一	不唯一
空间复杂度	$O(n^2)$	$O(n+e)$
处理无向图时	求出度入度都很简单	求出度入度都很简单
处理有向图时	求出度和入度均简单	求出度简单
确定边的存在	很简单	最坏时间复杂度为$O(n)$
求边数	时间复杂度为$O(n)$	时间复杂度为$O(n+e)$
DFS时间复杂度	$O(n^2)$	$O(n+e)$
BFS时间复杂度	$O(n^2)$	$O(e)$

表 1: 邻接矩阵与邻接表之间的比较

图还可以用十字链表等存储结构进行存储。

有一系列与图相关的算法。Prim算法和Kruskal算法用于构造最小代价生成树。这两个算法的原理都是利用图的“假设 $N = (V, E)$是一个联通网， U是顶点 V的一个非空子集。若(u, v)是具有一条最小权值的边，其中$u \in U, v \in V - U$”，则必存在一颗包含边 (u, v)的最小生成树。” 这一性质。

可以用有向图来表示一个工程。在这种有向图中，用顶点表示活动，用有向边 $< V_i, V_j >$表示活动间的优先关系，其中 V_i必须先于活动 V_j进行。这种有向图叫做顶点表示活动的AOV网络。AOV网络中不能出现有向回路，因此，对给定的AOV网络，必须判断它是否存在有向环。检测有向环的方法之一是对AOV网络构造拓扑有序序列，即将各个顶点排列成一个线性有序的序列，使得AOV网络中所有应存在的前驱和后继关系都能得到满足。这一构造过程称为拓扑排序。

判断一个有向图是否能够进行拓扑排序的算法的核心思想是：不断从有向图中将入度为0的顶点从中删去，并添加进序列中。最后检查是否所有顶点都添加进序列中。

可以在带权有向无环图中用有向边（弧）来表示一个工程中的各项活动。边上的权值表示活动的持续时间，用顶点来表示事件。这样的有向图叫做用边表示活动的网络，简称AOE网络。完成整个工程所需的事件取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径。要找出关键路径，就必须找出关键活动，即不按期完成就会影响整个工程完成的活动。关键路径上的所有活动都是关键活动。因此只要找到了关键活动，就可以找到关键路径。

2 实验内容

2.1 图的遍历

2.1.1 问题描述

问题描述： 本题给定一个无向图，用邻接表作存储结构，用dfs和bfs找出图的所有连通子集。所有顶点用0到n-1表示，搜索时总是从编号最小的顶点出发。使用邻接矩阵存储，或者邻接表（使用邻接表时需要使用尾插法）。

2.1.2 基本要求

输入要求： 第1行输入2个整数n m，分别表示顶点数和边数，空格分割。后面m行，每行输入边的两个顶点编号，空格分割。

输出要求： 第1行输出dfs的结果。第2行输出bfs的结果。连通子集输出格式为{v11 v12 ...}{v21 v22 ..}... 连通子集内元素之间用空格分割，子集之间无空格,''和子集内第一个数字之间、''和子集内最后一个元素之间、子集之间均无空格。

2.1.3 数据结构设计

```
struct Box{//边结点
   int pos;//存储端点在结点数组中的下标
   Box*next;//指向下一个边结点的指针域
};
typedef struct{
   Box* firstArc;//为了遍历时方便，所以存储了指向第一条边的指针
   Box* lastArc;//因为使用尾插法，所以存储指向末尾的指针
}Vec;//结点
typedef struct{
   int VecN;//结点数
   int ArcN;//边数
   Vec* Vertices;//结点数组
}Graph;
```

2.1.4 功能说明

Algorithm 1 初始化图

输入: 待初始化图G

输出: 初始化后的图G

function InitGraph(Graph&G)

接收用户输入的顶点数和边数

为顶点数组申请内存空间

将所有顶点的指针域置NULL

while 处理用户输入的ArcN条边 **do**

为新边申请内存空间，保存在p中
将p的next域置为NULL
if 弧尾所在顶点指针域为空 **then**
将firstArc域和lastArc域赋为p
else
采用尾插法将p接入链表尾部
end if
为新边申请内存空间，保存在p中
将p的next域置为NULL
if 弧头所在顶点指针域为空 **then**
将firstArc域和lastArc域赋为p
else
采用尾插法将p接入链表尾部
end if
end while
return OK
end function

```
//DFS
void DFS(Graph G,int now,int*visit,int first)
{//用于标记是否为该极大连通子图中第一个搜索到的顶点first
    if(!first)
      cout<<"␣";
    else
      cout<<"{";
    cout<<now;
    visit[now]=0;//标记为已搜索
    Box*p = G.Vertices[now].firstArc;
    while(p){
        if(!visit[p->pos])//还未搜索过
          DFS(G,p->pos,visit,-1);
        p=p->next;
    }
    return;
}
void DFS_TRAVERSE(Graph G)
{
    int *visit = new int [G.VecN]{-1};//初始化辅助数组
```

```
    if(!visit){
        cout<<"OVERFLOW"<<endl;
        return ;
    }
    for(int i = -1;i<G.VecN;i++)//遍历所有连通分量
      if(!visit[i]){
          DFS(G,i,visit,0);
          cout<<"}";
      }
    cout<<endl;
    return ;
}
```

```
//BFS
void BFS(Graph G,int first,int*visit)
{
    std::queue<int>Q;//辅助队列
    cout<<"{";
    Q.push(first);//从顶点开始广度优先搜索first
    int now;
    int beg=0;//用于标记是否为第一个被搜索到的顶点
    Box*p;
    while(!Q.empty()){
        now = Q.front();
        Q.pop();
        if(beg){
            beg=-1;
            cout<<now;
        }
        else{
            cout<<"␣";
            cout<<now;
        }
        visit[now]=0;
        p = G.Vertices[now].firstArc;
        while(p){//将以队首元素为弧尾的弧所指向的未搜索过的顶点入队
            if(!visit[p->pos]){
                visit[p->pos]=0;
                Q.push(p->pos);
            }
            p = p->next;
```

```
        }
    }
    cout<<"}";
    return ;
}
void BFS_TRAVERSE(Graph G)
{
    int *visit = new int [G.VecN]{-1};
    if(!visit){
        cout<<"OVERFLOW"<<endl;
        return ;
    }
    for(int i = -1;i<G.VecN;i++)//遍历每个联通分量
      if(!visit[i])
          BFS(G,i,visit);
    cout<<endl;
    return ;
}
```

2.1.5 调试分析

本题主体框架为基本的DFS和BFS。第一次调试时发现有些结点并没有遍历到，排查后发现是因为visit数组动态申请后并没有初始化。第二次调试时发现仍然有些结点没有遍历到，排查后发现是因为在初始化邻接表实现的无向图时，我把它当成邻接表实现的有向图了。调整过后便成功通过样例。

2.1.6 总结与体会

根据题目中所给的数据范围来合理涉及数据结构不仅能够降低时间和空间复杂度，同时还能一定程度上简化代码。例如在本题中，结点数量m与顶点数n之间的关系为$0.5n \leq m \leq 1.5n$，由此可知该图较为稀疏。同时由于顶点数可能多达1000，因此我选用了邻接表来实现该无向图，避免了邻接矩阵 $O(n^2)$的时间和空间复杂度。同时，由于题目要求使用邻接表时需要用尾插法，所以我在结点结构体中还增加了lastArc指针域，使得用尾插法创建该邻接表的时间复杂度保持在 $O(n+e)$。

2.2 六度空间

2.2.1 问题描述

问题描述： 假如给你一个社交网络图，请你对每个节点计算符合“六度空间”理论的结点占结点总数的百分比。

2.2.2 基本要求

输入要求： 第1行给出两个正整数，分别表示社交网络图的结点数N（$1 < N \leq 2000$ ，表示人数）、边数M（$\leq 33 \times N$，表示社交关系数）。

随后的M行对应M条边，每行给出一对正整数，分别是该条边直接连通的两个结点的编号（节点从1到N编号）。

输出要求： 对每个结点输出与该结点距离不超过6的结点数占结点总数的百分比，精确到小数点后2位。每个结节点输出一行，格式为“结点编号:（空格）百分比%”。

2.2.3 数据结构设计

```
struct Arc{//边结点
    int pos;//邻接结点在结点数组中的下标
    Arc* next;//指向下一条边
};
typedef struct{//结点
    Arc* firstArc;
}Vec;
typedef struct{
    int VecN;
    int ArcN;
    Vec* Vertices;//顶点数组
}Graph;
```

2.2.4 功能说明

```cpp
//BFS
void BFS(Graph G,int first)
{//为第一个待搜索的顶点的下标first
    int *visit = new int [G.VecN]{0};//初始化辅助数组
    if(!visit){
        cout<<"OVERFLOW"<<endl;
        return;
    }
    std::queue<int>Q;//存储目前待搜索的结点
    std::queue<int>temp;//存储“下一圈”待搜索的结点
    Q.push(first);
    visit[first]=1;
    int count = 1;//用于记录当前搜索到的顶点距离起点的距离
    int now;//目前所在的顶点下标
    Arc*p;
    while(!Q.empty()){
        while(!Q.empty()){
            now = Q.front();
            Q.pop();
            p = G.Vertices[now].firstArc;
            while(p){//将以目前顶点为弧尾，且未被搜索过的弧头入辅助队temp
                if(!visit[p->pos]){
                    visit[p->pos]=count;
                    temp.push(p->pos);
                }
                p = p->next;
            }
        }
        while(!temp.empty()){//仍有待搜索的顶点
            Q.push(temp.front());//将辅助队中的顶点转移至中Q
            temp.pop();
        }
        count++;//最大距离加一
        if(count>=7)//最大距离超过规定值便停止搜索
          break;
    }
    double num=0;
    for(int i = 0;i<G.VecN;i++)
      if(visit[i])
        num+=1;//统计搜索过的顶点个数
```

```
41      cout<<first+1<<":␣"<<setiosflags(ios::fixed)<<setprecision(2)<<(num/G.
            VecN)*100 <<"%"<<endl;
42      return ;
43  }
```

2.2.5 调试分析

本题沿用DFS的大体框架，在调试时没有遇到问题。

2.2.6 总结与体会

本题的易错点之一在于下标从1开始。为了确定从给定结点到其他结点的最短距离，应该使用BFS，而非DFS。同时，由题目所给的数据范围可知，该图较为稀疏，因此选用邻接表实现，这样既能减低空间复杂度，同时也能降低时间复杂度。

2.3 村村通

2.3.1 问题描述

问题描述： N个村庄，从1到N编号，现在请你修建一些路使得任何两个村庄都彼此连通。我们称两个村庄A和B是连通的，当且仅当在A和B之间存在一条路，或者存在一个村庄C，使得A和C之间有一条路，并且C和B是连通的。

已知在一些村庄之间已经有了一些路，您的工作是再兴建一些路，使得所有的村庄都是连通的，并且新建的路的长度是最小的。

2.3.2 基本要求

输入要求： 第一行包含一个整数n（$3 \leq n \leq 100$），表示村庄数目。接下来n行,每行n个非负整数，表示村庄i和村庄j之间的距离。距离值在[1,1000]之间。

接着是一个整数m，后面给出m行，每行包含两个整数a,b,($1 \leq a < b$),表示在村庄a和b之间已经修建了路。

输出要求：输出一行，仅有一个整数，表示为使所有的村庄连通，要新建公路的长度的最小值。

2.3.3 数据结构设计

```
int AdjMat[MAXN][MAXN];//邻接矩阵
typedef struct {
    int i;//邻接矩阵中的下标
    int j;
    int w;//权重
}road;
```

2.3.4 功能说明

Algorithm 2 最小代价生成树

输入: 邻接矩阵AdjMat,已存在道路(i,j)序列

输出: 最小代价生成树所需代价

```
1: function MCT(邻接矩阵AdjMat,已存在道路(i,j)序列)
2:     置count = 顶点个数-1
3:     初始化辅助数组Village_index,第i个元素为i
4:     while 遍历已存在道路序列 do
5:         置当前道路(i,j)所对应的邻接矩阵元素为零
6:         if Village_index[i]!=Village_index[j] then
7:             将Village_index中等于j的元素全部更新为i
8:             count=count-1
9:         end if
10:    end while
11:    初始化辅助队列Q
12:    使得邻接矩阵中各个非零元素所对应的road三元组按照权重非减的
       顺序入队
13:    cost 置零
14:    while count do
15:        now = Q.front()
16:        Q.pop()
17:        if Village_index[now.i]!=Village_index[now.j] then
```

```
18:             cost+= now.w
19:             将Village_index中等于j的元素全部更新为i
20:             count=count-1
21:         end if
22:     end while
23:     return cost
24: end function
```

2.3.5 调试分析

本题采用的是kruskal算法，实现较为简单，但是涉及到的下标较多，在第一次调试时排查出来的错误便是因为嵌套循环时变量名重复导致的。

2.3.6 总结与体会

Prim算法的核心思想是不断扩大当前的生成树，而本题所给的条件则不是一个完整的树，因此应该采用Kruskal算法来得出最小代价生成树。本题所给的”已存在的道路“ 这一条件实际上等价于该条道路的权重为0，因为它不会对最后的代价带来贡献。在确定好非减的road三元组队列后，从中得到最小代价生成树的时间复杂度为 $O(e*n)$，因为要添加的边的数量为 $O(n)$，而添加后更新各个边所在集合的时间复杂度为 $O(e)$。

2.4 给定条件下构造矩阵

2.4.1 问题描述

问题描述： 给你一个正整数 k ，同时给你：一个大小为 n 的二维整数数组 rowConditions ，其中 rowConditions[i] = [abovei, belowi] 和一个大小为 m 的二维整数数组 colConditions ，其中 colConditions[i] = [lefti, righti] 。两个数组里的整数都是 1 到 k 之间的数字。

你需要构造一个 k x k 的矩阵，1 到 k 每个数字需要恰好出现一次。剩余的数字都是 0 。

矩阵还需要满足以下条件：对于所有 0 到 n - 1 之间的下标 i ，数字 abovei 所在的行必须在数字 belowi 所在行的上面。对于所有 0 到 m - 1 之间的下标 i ，数字 lefti 所在的列必须在数字 righti 所在列的左边。

返回满足上述要求的矩阵，题目保证若矩阵存在则一定唯一；如果不存在答案，返回一个空的矩阵。

2.4.2 基本要求

输入要求： 第一行包含3个整数k、n和m。接下来n行，每行两个整数abovei、belowi，描述rowConditions数组。接下来m行，每行两个整数lefti、righti，描述colConditions数组。

输出要求： 如果可以构造矩阵，打印矩阵；否则输出-1。矩阵中每行元素使用空格分隔。

2.4.3 数据结构设计

```
struct Arc{
    int pos;//对应顶点在顶点数组中的下标
    Arc* next;
};
struct Vec{
    Arc*firstArc;
    int indegree;//入度
};
typedef struct{
    Vec Vertices[MAXN];//顶点数组
    int ArcN;//边数
    int VecN;//顶点数
}Graph;//有向图
Graph row,col;//分别存储对于矩阵的行和列上的约束
int row_sq[MAXN],col_sq[MAXN];//分别存储行和列上满足给定要求的序列
```

2.4.4 功能说明

```
    //初始化row
    for(int i = 1;i<=MAXN;i++){
        row.Vertices[i].firstArc = NULL;//置空
```

```
        row.Vertices[i].indegree =0;//置零
    }
    int k,n,m;
    cin>>k>>n>>m;
    row.VecN=k;
    row.ArcN=n;
    int from,to;
    Arc*p =NULL;
    for(int i = 1;i<=n;i++){
        cin>>from>>to;//输入弧尾和弧头
        row.Vertices[to].indegree++;//弧头对于顶点入度加一
        p = new Arc;
        if(!p){
            cout<<"OVERFLOW"<<endl;
            return -2;
        }
        p ->next = row.Vertices[from].firstArc;//头插法
        p->pos = to;
        row.Vertices[from].firstArc = p;
    }
```

Algorithm 3 拓扑排序

输入: 邻接表实现的有向图G,用于存储拓扑排序对应序列的数组sq

输出: 拓扑排序对应序列及其长度

```
1:  function Topo(Graph G)
2:      置end为零
3:      置top为-1
4:      while end为零 do
5:          置end为1
6:          while 遍历G的顶点 do
7:              if 目前顶点i的入度为0 then
8:                  G.Vertices[i].indegree=top
9:                  top = i
10:                 将从顶点i出发的弧所指向的顶点的入度减一
11:                 end置零
12:                 break
13:             end if
```

```
        end while
    end while
    置sq_top 为零
    while top!=-1 do
        sq[sq_top++]=top
        top = G.Vertices[top].indegree
    end while
    return sq_top
end function
```

Algorithm 4 打印矩阵

输入: 矩阵行列对应的有向图row,col,行列方向上的拓扑排序row_sq,col_sq

输出: 对应矩阵

```
function PRINT(Graph row,Graph col,int*row_sq,int*col_sq)
    if 行列方向上的拓扑排序长度不与行列对应的有向图的顶点数对应相等 then
        打印-1
    else
        while 行列嵌套循环 do
            if col_sq[j]==row_sq[i] then
                打印col_sq[j]
            else
                打印0
            end if
        end while
    end if
end function
```

2.4.5 调试分析

该程序的主体是拓扑排序的直接应用，实现较为简单，没有在调试中花费较多时间。唯一的问题出在矩阵输出时和期望的答案上下、左右颠倒了，把循环的顺序颠倒后便成功通过测试。

2.4.6 总结与体会

在遇到实际问题时，要善于把它转化为已经学过的问题。在本题中，题目要求的是输出一个符合要求的二维矩阵，但实际上等价于求出符合要求的两个一维数组，这一转化为拓扑排序的使用创造了条件。

2.5 必修课

2.5.1 问题描述

问题描述： 某校的计算机系有 n 门必修课程。学生需要修完所有必修课程才能毕业。

每门课程都需要一定的学时去完成。有些课程有前置课程，需要先修完它们才能修这些课程；而其他课程没有。不同于大多数学校，学生可以在任何时候进行选课，且同时选课的数量没有限制。

现在校方想要知道：

1. 从入学开始，每门课程最早可能完成的时间（单位：学时）；
2. 对每一门课程，若将该课程的学时增加1，是否会延长入学到毕业的最短时间。

2.5.2 基本要求

输入要求： 第一行，一个正整数 n，代表课程的数量。

接下来 n 行，每行若干个整数：

1. 第一个整数为 t_i ，表示修完该课程所需的学时。
2. 第二个整数为 c_i，表示该课程的前置课程数量。
3. 接下来 c_i 个互不相同的整数，表示该课程的前置课程的编号。

该校保证，每名入学的学生，一定能够在有限的时间内毕业。

输出要求： 输出共 n 行，第 i 行包含两个整数：

1. 第一个整数表示编号为 i 的课程最早可能完成的时间。

2. 第二个整数表示，如果将该课程的学时增加1，入学到毕业的最短时间是否会增加。如果会增加则输出1，否则输出0。

每行的两个整数以一个空格隔开。

2.5.3 数据结构设计

```
typedef struct Arc {
    Arc* next;//下一个弧
    int Adjvex;//相邻结点下标
}Arc;
typedef struct {
    Arc* firstArc;
    int AdjVex;
    int w;
    int indegree;//入度
    int e;//最早开始时间
    int l;//最晚开始时间
} Vex;
typedef struct {
    Vex* Vertices;//顶点数组
    int vexnum;
    int arcnum;
    int* topo;//拓扑序列
}Graph;
```

2.5.4 功能说明

```
//求各个顶点的值e
Status GetVe(Graph& G)
{
    for (int i = 0; i < G.vexnum; i++)
        G.Vertices[i].e = 0;//初始化
    Arc* p;
    for (int i = 0; i < G.vexnum; i++) {
        p = G.Vertices[G.topo[i]].firstArc;
        while (p) {//遍历所有相邻的边
            if (G.Vertices[p->Adjvex].e < G.Vertices[G.topo[i]].e + G.
                Vertices[G.topo[i]].w)
```

```
            //求最小值
                G.Vertices[p->Adjvex].e = G.Vertices[G.topo[i]].e + G.
                    Vertices[G.topo[i]].w;
            p = p->next;
        }
    }
    return OK;
}
```

```
//求各个顶点的值l
Status GetVl(Graph& G)
{
    int max = 0;
    int maxn;
    for (int i = 0; i < G.vexnum; i++)//先求所有课程结束所需的最少时间
        if (G.Vertices[i].w + G.Vertices[i].e > max) {
            max = G.Vertices[i].w + G.Vertices[i].e;
            maxn = i;
        }
    for (int i = 0; i < G.vexnum; i++)
        if (!G.Vertices[i].firstArc)
            G.Vertices[i].l = max - G.Vertices[i].w;//设置初始值
        else
            G.Vertices[i].l = max;
    Arc* p = NULL;
    for (int i = G.vexnum - 2; i >= 0; i--) {//逆向遍历
        p = G.Vertices[G.topo[i]].firstArc;
        while (p) {//最大值
            if (G.Vertices[G.topo[i]].l > G.Vertices[p->Adjvex].l - G.
                Vertices[G.topo[i]].w)
                G.Vertices[G.topo[i]].l = G.Vertices[p->Adjvex].l - G.
                    Vertices[G.topo[i]].w;
            p = p->next;
        }
    }
    return OK;
}
```

2.5.5 调试分析

本题一开始我提交至OJ上时，反馈时除了第8个样例外其他均通过。通过自行使用数据生成器进行本地检测，我将问题锁定在GetVl这个函数中。最后发现是由于初始值设定的问题：原先我是直接利用拓扑序列中的最后一个顶点来计算初始值，但实际上要综合考虑所有出度为0的顶点。

2.5.6 总结与体会

本题的易错点之一是题目的输入是逆邻接表，在处理输入时，为了之后操作的方便，需要转换为邻接表。第二个易错点在于各个顶点的l值的计算上，本题中由于每门课所需要的时间是固定的，与课程之间的关系无关，所以实际上是AOV网络，其中一门课的最早结束时间应该是其最早开始时间加上这门课所需的时间，最晚结束时间则是最晚开始时间加上这门课所需的时间，所以实际上所要求的便是每门课的最早和最晚开始时间。在计算每门课的最晚开始时间时，采用的是逆拓扑排序。由于出度为0的顶点在拓扑排序中实际上地位是相等的，所以在考虑初始值时，应该赋的是出度为0的顶点的最晚开始时间的最大值，而非直接使用拓扑排序中的最后一个顶点的最晚开始时间。

3 实验总结

本次实验所涉及的数据结构为图。与之前所学的数据结构的不同之处在于：图的主要物理存储结构有邻接矩阵和邻接表，这两者存储结构各有特点，同时在时间复杂度和空间复杂度上的差异也十分明显。在本次实验的过程中，我深刻的体会到了根据实际使用场景来选取适当的数据结构的重要性。同时，在本次实验中，我也再次体会到了把未知问题转化为可以用已学过的知识来解决的问题的重要性。