作业 H4 实验报告

姓名：钱子贤 学号：2054170 日期：2021 年 12 月 12 日

实验报告格式按照模板来即可，对字体大小、缩进、颜色等不做要求

实验报告要求在文字简洁的同时将内容表示清楚

1. 涉及数据结构和相关背景

图形结构的形式定义

图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构:

Graph = (V, R)

其中:

V = { x | x 某个数据对象 }

是顶点的有穷非空集合;

R——边的有限集合

R = {(x, y) | x, y V && Path (x, y)}无向图 或

R = {<x, y> | x, y V && Path <x, y>}有向图

是顶点之间关系的有穷集合，也叫做边(edge)集合。Path <x, y>表示从 x 到 y 的一条单向通路，它是有方向的。x 是弧尾，y 是弧头。

有向图与无向图

有向图中：边用<x, y>表示，且 x 与 y 是有序的

a. 有向图中的边称为“弧”

b. x——弧尾或初始点 y——弧头或终端点

无向图：边用(x, y) 表示，且顶 x 与 y 是无序的

完全图

在具有 n 个顶点的有向图中，最大弧数为 n(n-1)

在具有 n 个顶点的无向图中，最大边数为 n(n-1)/2

顶点的度

无向图：与该顶点相关的边的数目

有向图：入度 ID(v) ： 以该顶点为头的弧的数目

出度 OD(v) ： 以该顶点为尾头的弧的数目

路径 在图 G = (V, E) 中，若从顶点 vi 出发，沿一些边经过一些顶点 vp1, vp2, …, vpm，到达顶点 vj。则称顶点序列 (vi vp1 vp2 … vpm vj) 为从顶点 vi 到顶点 vj 的路径。它经过的边(vi, vp1)、(vp1, vp2)、…、(vpm, vj)应是属于 E 的边。

路径长度

非带权图的路径长度是指此路径上边/弧的条数。

带权图的路径长度是指路径上各边/弧的权之和。

简单路径 若路径上各顶点 v1,v2,...,vm 均不互相重复，则称这样的路径为简单路径。

回路 若路径上第一个顶点 v1 与最后一个顶点 vm 重合，则称这样的路径为回路或环。

连通图与连通分量 在无向图中，若从顶点 v1 到顶点 v2 有路径，则称顶点 v1 与 v2 是连通的。如果图中任意一对顶点都是连通的，则称此图是连通图。非连通图的极大连通

子图叫做连通分量。
强连通图与强连通分量　　在有向图中，若对于每一对顶点 vi 和 vj，都存在一条从 vi 到 vj 和从 vj 到 vi 的路径，则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量。
生成树　一个连通图的生成树是它的极小连通子图，在 n 个顶点的情形下，有 n-1 条边。
生成树是对连通图而言的
是连通图的极小连通子图
包含图中的所有顶点
有且仅有 n-1 条边

2. 实验内容

2.1

HW4：图状结构 > 4-1 图的存储结构

图的存储结构
描述
图是一种描述多对多关系的数据结构，图中的数据元素称作顶点，具有关系的两个顶点形成的一个二元组称作边或弧，顶点的集合 V 和关系的集合 R 构成了图，记作 G=（V,R）。图又分成有向图，无向图，有向网，无向网。图的常用存储结构有邻接矩阵、邻接表、十字链表、邻接多重表。图的基本操作包括图的创建、销毁、添加顶点、删除顶点、插入边、删除边、图的遍历。
本题练习邻接矩阵和邻接表的创建。

输入
第 1 行输入一个数字 1~4，1 为有向图,2 为有向网,3 为无向图,4 为无向网;
第 2 行输入 2 个整数 n m，分别表示顶点数和边数，空格分割
第 3 行为 n 个字符的序列，一个字符表示一个顶点
后面 m 行，若前面选择的是图，则每行输入边的两个顶点字符，空格分割，若是网，则每行输入弧的两个顶点字符和弧的权值，空格分割
输出
第 1 行输出顶点向量，顶点字符以一个空格分割
接着 n 行 n 列，输出邻接矩阵，每个数字占 4 位
接着 n 行，输出邻接表

```
#define MAX_VERTEX_NUM 200//最大顶点数
typedef int AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];//邻接矩阵类型
struct MGraph
{
    int vexnum, arcnum;//图的顶点数和边/弧数
    char vexs[MAX_VERTEX_NUM];//顶点表
    char v1[MAX_VERTEX_NUM], v2[MAX_VERTEX_NUM];
    AdjMatrix arcs;//邻接矩阵
```

```
};
int x, y;

int LocateVex(MGraph G, char u)
{
    for (int i = 0; i < G.vexnum; i++)
    {
        if (u == G.vexs[i])
            return i;
    }
    return 0;
}

void CreateMGaph(MGraph& G,int species)//邻接矩阵的建立
{
    cin >> G.vexnum >> G.arcnum;
    for (int i = 0; i < G.vexnum; i++)
        cin >> G.vexs[i];
    for (int i = 0; i < G.vexnum; i++)
    {
        for (int j = 0; j < G.vexnum; j++)
            G.arcs[i][j] = 0;
    }
    for (int i = 0; i < G.arcnum; i++)
    {
        int w = 1;
        cin >> G.v1[i] >> G.v2[i];
        if (species % 2 == 0)
            cin >> w;
        x = LocateVex(G, G.v1[i]);
        y = LocateVex(G, G.v2[i]);
        G.arcs[x][y] = w;
        if (species >= 3)
            G.arcs[y][x] = w;
    }
}

void PrintMGraph(MGraph G)
{
    for (int i = 0; i < G.vexnum; i++)
        cout << G.vexs[i] << " ";
    cout << endl;
    for (int i = 0; i < G.vexnum; i++)
```

```
    {
        for (int j = 0; j < G.vexnum; j++)
            printf("%4d", G.arcs[i][j]);
        cout << endl;
    }
}

struct ArcNode//顶点节点
{
    int adjvex;//下标（位置）
    int info;//存储的数据
    ArcNode* nextarc;
};

typedef struct VNode//弧节点
{
    char data;//头结点
    ArcNode* firstarc;//指向第一条弧的指针
} AdjList[MAX_VERTEX_NUM];


struct ALGraph
{
    AdjList vertices;//邻接表
    int vexnum, arcnum;//顶点数和弧数
};

int LocateVex_(ALGraph G, char u)
{
    for (int i = 0; i < G.vexnum; i++)
    {
        if (u == G.vertices[i].data)
            return i;
    }
    return 0;
}


void CreateALGraph_adjlist(ALGraph& G,int species,MGraph GO)
{
    ArcNode* p;
    G.vexnum = GO.vexnum;
    G.arcnum = GO.arcnum;
```

```
    for (int i = 0; i < G.vexnum; i++)
    {
        G.vertices[i].data = GO.vexs[i];
        G.vertices[i].firstarc = NULL;
    }
    for (int i = 0; i < G.arcnum; i++)
    {
        x = LocateVex_(G, GO.v1[i]);
        y = LocateVex_(G, GO.v2[i]);
        p = (ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex = y;
        p->info = GO.arcs[x][y];
        p->nextarc = G.vertices[x].firstarc;
        G.vertices[x].firstarc = p;
        if (species >= 3)
        {
            p = (ArcNode*)malloc(sizeof(ArcNode));
            p->adjvex = x;
            p->info = GO.arcs[x][y];
            p->nextarc = G.vertices[y].firstarc;
            G.vertices[y].firstarc = p;
        }
    }
    for (int i = 0; i < G.vexnum; i++)
    {
        cout << G.vertices[i].data << "-->";
        ArcNode* t = G.vertices[i].firstarc;
        while (t)
        {
            if (species % 2 == 1)
                cout << t->adjvex;
            else
                cout << t->adjvex << "," << t->info;
            t=t->nextarc;
            if (t)
                cout << " ";
        }
        cout << endl;
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

简要描述调试过程

可以使用图片辅助说明

```
6 10
123456
1 2 5
1 4 7
2 3 4
3 1 8
3 6 9
4 3 5
4 6 6
5 4 5
6 1 3
6 5 1
1 2 3 4 5 6
   0    5    0    7    0    0
   0    0    4    0    0    0
   8    0    0    0    0    9
   0    0    5    0    0    6
   0    0    0    5    0    0
   3    0    0    0    1    0
1-->3,7 1,5
2-->2,4
3-->5,9 0,8
4-->5,6 2,5
5-->3,5
6-->4,1 0,3
```

2.1.6 总结和体会

可以说说做此题的收获、分析这道题目的难点和易错点（数据边界、对算法效率的要求等）

把同一个顶点发出的边链接在同一个边链表中
链表的每一个结点代表一条边，叫做表结点（边结点）
邻接点域 adjvex 保存与该边相关联的另一顶点的顶点下标，链域 nextarc 存放指向同一链表中下一个表结点的指针，数据域 info 存放边的权
边链表的表头指针存放在头结点中。头结点以顺序结构存储，其数据域 data 存放顶点信息，链域 firstarc 指向链表中第一个顶点。
带权图的边结点中 info 保存该边上的权值
顶点 Vi 的边链表的头结点存放在下标为 i 的顶点数组中

在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定
设图中有 n 个顶点，e 条边，则用邻接表表示无向图时，需要 n 个顶点结点，2e 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 n 个顶点结点，e 个边结点
建立邻接表的时间复杂度为 O(n*e)。若顶点信息即为顶点的下标，则时间复杂度为 O(n+e)。

2.2 题目二

HW4：图状结构 > 4-2 图的遍历

图的遍历
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）。

输入
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
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

对于 20%的数据，有 0<n<=15;
对于 40%的数据，有 0<n<=100;
对于 100%的数据，有 0<n<=10000;
对于所有数据，0.5n<=m<=1.5n，保证输入数据无错。
下载 p107_data.cpp，编译运行以生成随机测试数据

```
#define MAX_VERTEX_NUM 200//最大顶点数
struct ArcNode//顶点节点
{
    int adjvex;//下标（位置）
    ArcNode* nextarc;
};

typedef struct VNode//弧节点
{
    int data;//头结点
    ArcNode* firstarc;//指向第一条弧的指针
} AdjList[MAX_VERTEX_NUM];

struct ALGraph//邻接表构造
{
```

```
    AdjList vertices;//邻接表
    int vexnum, arcnum;//顶点数和弧数
};

int LocateVex_(ALGraph G, int u)
{
    for (int i = 0; i < G.vexnum; i++)
    {
        if (u == G.vertices[i].data)
            return i;
    }
    return 0;
}

void CreateALGraph_adjlist(ALGraph& G)//尾插法建立
{
    ArcNode* p;
    int x, y;
    int v1, v2;
    cin >> G.vexnum >> G.arcnum;
    for (int i = 0; i < G.vexnum; i++)
    {
        G.vertices[i].data = i;
        G.vertices[i].firstarc = NULL;
    }
    for (int i = 0; i < G.arcnum; i++)
    {
        cin >> v1 >> v2;
        x = LocateVex_(G, v1);
        y = LocateVex_(G, v2);
        ArcNode* t;

        p = (ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex = y;
        p->nextarc = NULL;
        t=G.vertices[x].firstarc;
        if (t == NULL)//第x个顶点的边表结点为空
        {
            G.vertices[x].firstarc = p;//实现两点相连
        }
        else
        {
            while (t->nextarc != NULL)
            {
```

```
                t = t->nextarc;
            }
            t->nextarc = p;
        }
        p = (ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex = x;
        p->nextarc = NULL; G.vertices[y].firstarc;
        t=G.vertices[y].firstarc ;
        if (t == NULL)//第x个顶点的边表结点为空
        {
            G.vertices[y].firstarc = p;//实现两点相连
        }
        else
        {
            while (t->nextarc != NULL)
            {
                t = t->nextarc;
            }
            t->nextarc = p;
        }
    }
    for (int i = 0; i < G.vexnum; i++)
    {
        cout << G.vertices[i].data << "-->";
        ArcNode* t = G.vertices[i].firstarc;
        while (t)
        {
                cout << t->adjvex;
            t = t->nextarc;
            if (t)
                cout << " ";
        }
        cout << endl;
    }
}

int visited[MAX_VERTEX_NUM];//辅助数组，判断是否访问

void DFS(ALGraph G, int i)//深度优先算法
{
    ArcNode* p;
    p = G.vertices[i].firstarc;
    if (!visited[i])
    {
```

```
            cout << " ";
            cout << i;
        }
        visited[i] = 1;//已被遍历

        while (p)
        {
            if (!visited[p->adjvex])//未被遍历
                    DFS(G, p->adjvex);
            p = p->nextarc;
        }
}

void DFSTraverse(ALGraph G)
{
        for (int i = 0; i < G.vexnum; i++)//辅助数组初始化，都未被访问过
            visited[i] = 0;
        for (int i = 0; i < G.vexnum; i++)
            if (!visited[i])
            {
                cout << "{";
                visited[i] = 1;
                cout << i;
                DFS(G, i);
                cout << "}";
            }
        cout << endl;
}

#define MAXQSIZE 1000
typedef struct {
        int* base;
        int   front;
        int   rear;
}SqQueue;
int InitQueue(SqQueue& Q)
{
        Q.base = (int*)malloc(MAXQSIZE * sizeof(int));
        if (!Q.base) exit(OVERFLOW);
        Q.front = Q.rear = 0;
        return 1;
}

int c;
```

```
void BFS(ALGraph G, int v)
{
    ArcNode* p;
    SqQueue Q;
    InitQueue(Q);
    cout << G.vertices[v].data;
    visited[v] = 1;
    Q.base[Q.rear] = v;
    Q.rear = (Q.rear + 1) % MAXQSIZE;
    p = G.vertices[v].firstarc;
    while (Q.front != Q.rear)
    {
        v = Q.base[Q.front];
        Q.front = (Q.front + 1) % MAXQSIZE;
        p = G.vertices[v].firstarc;
    }
    while (p)
    {
        if (!visited[p->adjvex])
        {
            cout << " " << G.vertices[p->adjvex].data;
            visited[p->adjvex] = 1;
            Q.base[Q.rear] = p->adjvex;
            Q.rear = (Q.rear + 1) % MAXQSIZE;
        }
        p = p->nextarc;
    }
}

void BFSTraverse(ALGraph G)
{
    for (int v = 0; v < G.vexnum; ++v)
        visited[v] = 0;
    for (int v = 0; v < G.vexnum; ++v)
        if (!visited[v])
        {
            cout << "{";
            c = 0;
            BFS(G, v);
            cout << "}";
        }
}
```

```
0 5
0 11
1 12
3 4
6 7
6 8
6 10
7 10
9 11
9 12
11 12
0-->1 2 5 11
1-->0 12
2-->0
3-->4
4-->3
5-->0
6-->7 8 10
7-->6 10
8-->6
9-->11 12
10-->6 7
11-->0 9 12
12-->1 9 11
{0 1 12 9 11 2 5}{3 4}{6 7 10 8}
{0 1 2 5 11}{3 4}{6 7 8 10}{9 12}
```

DFS 的思路
（1）从图中的某个顶点 V 出发，访问之；
（2）依次从顶点 V 的未被访问过的邻接点出发，深度优先遍历图，直到图中所有和顶点 V 有路径相通的顶点都被访问到；
（3）若此时图中尚有顶点未被访问到，则另选一个未被访问过的顶点作起始点，重复上述（1）（2）的操作，直到图中所有的顶点都被访问到为止。
图中有 n 个顶点，e 条边。
如果用邻接表表示图，沿 Firstarc nextarc 链可以找到某个顶点 v 的所有邻接顶点 w。由于总共有 2e 个边结点，所以扫描边的时间为 O(e)。而且对所有顶点递归访问 1 次，所以遍历图的时间复杂性为 O(n+e)。
如果用邻接矩阵表示图，则查找每一个顶点的所有的边，所需时间为 O(n)，则遍历图中所有的顶点所需的时间为 O(n2)。

BFS 的思路
（1）从图中的某个顶点 V 出发，访问之；
（2）依次访问顶点 V 的各个未被访问过的邻接点，将 V 的全部邻接点都访问到；
（3）分别从这些邻接点出发，依次访问它们的未被访问过的邻接点，并使先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直到图中所有已被访问过的顶点的邻接点都被访问到。

（4）若此时图中尚有顶点未被访问到，则另选图中一个未曾被访问的顶点作起点，重复上述过程，直至图中所有的顶点都被访问到为止。
如果使用邻接表表示图，则循环的总时间代价为 d0 + d1 + … + dn-1 = O(e)，其中的 di 是顶点 i 的度。
如果使用邻接矩阵，则对于每一个被访问过的顶点，循环要检测矩阵中的 n 个元素，总的时间代价为O(n2)。

2.3 题目三

HW4：图状结构 > 4-3 关键路径

关键路径
描述
一个工程项目由一组子任务（或称活动）构成，子任务之间有的可以并行执行，有的必须在完成了其它一些子任务后才能执行，并且每个任务完成需要一定的时间。
对于一个工程，需要研究的问题是:
（1）由这样一组子任务描述的工程是否可行?
（2）若可行，计算完成整个工程需要的最短时间。
（3）这些任务中，哪些任务是关键活动（也就是必须按时完成的任务，否则整个项目就要延迟)。
现将这样一个工程项目用一个有向图表示，给定一组顶点，每个顶点表示任务之间的交接点（若任务 2 要在任务 1 完成后才可以开始，则这两任务之间必须有一个交接点，该点称作事件)。任务用有向边表示，边的起点是该任务可以开始执行的事件，终点是该任务已经完成的事件，边上的权值表示该任务完成需要执行的时间。
请你编写程序，回答上述三个问题。

注意：求关键路径 ve 要按照拓扑序列的顺序计算，vl 要按照拓扑序列逆序计算。本题不保证顶点已按拓扑排序从小到大编号。可参照课本增加一个栈存放拓扑序列。

输入
第一行包含两个整数 n、m，其中 n 表示顶点数，m 表示任务数。顶点按 1~n 编号。
接下来 m 行表示 m 个任务，每行包含三个正整数 ui、vi、wi，分别表示该任务开始和完成的顶点序号，及任务完成的时间。
整数之间用空格表示。
对于 20%的数据，有 0<n<=10
对于 40%的数据，有 0<n<=100
对于 100%的数据，有 0<n<=100000
对于所有数据，1.5n<=m<=2n，wi 是一个 1 到 100 的整数，保证输入的边按起点从小到大排序，起点相同的边按终点从小到大排序
下载 p55_data.cpp，编译运行以生成随机测试数据
输出
如果该有向图有环，则工程不可行，输出 0；否则
第 1 行输出完成整个工程项目需要的时间，
从第 2 行开始输出所有关键活动，每个关键活动占一行，按格式“ui->vi”输出，其中 u 和 v 为该任务开始和完成涉及的交接点编号。

注：关键活动输出的顺序规则是：按起点从小到大排序，起点相同的边按终点从小到达排序

```
class Picture {
public:
    struct Node {
        int id, earliest, latest;
        std::vector<std::pair<Node*, int>> next;
        std::vector<std::pair<Node*, int>> last;
        Node(const int id_) : id(id_) { earliest = INT_MIN; latest = INT_MAX; }
        ~Node() = default;
    };

    bool IsNotEnd(const int n) { return n >= 0 && n < nodes.size() &&
nodes[n].next.empty(); }

    void AddErrorLine(int e) {
        int step = rand() % 10 + 1;
        int i = e;
        while (step-- && nodes[i].next.size() > 0)
            i = nodes[i].next[rand() % nodes[i].next.size()].first->id;
        AddLine(i, e, rand() % 100 + 1);
    }

    void AddLine(int s, int e, int w) {
        nodes[s].next.push_back(std::make_pair(&nodes[e], w));
        nodes[e].last.push_back(std::make_pair(&nodes[s], w));
    }

    int FindKeyPath() {
        std::stack<Node*> S;
        for (auto& node : nodes) {
            if (node.last.empty()) {
                S.push(&node);
            }
        }
        int max_earliest = INT_MIN;
        while (!S.empty()) {
            Node& n = *S.top();
            S.pop();
            max_earliest = std::max(max_earliest, n.earliest);
            for (auto& next : n.next) {
                if (next.first->earliest < n.earliest + next.second) {
                    next.first->earliest = n.earliest + next.second;
                    S.push(next.first);
```

```
                }
            }
        }
        for (auto& node : nodes) {
            if (node.next.empty()) {
                node.latest = max_earliest;
                S.push(&node);
            }
        }
        while (!S.empty()) {
            Node& n = *S.top();
            S.pop();
            for (auto& last : n.last) {
                if (last.first->latest > n.latest - last.second) {
                    last.first->latest = n.latest - last.second;
                    S.push(last.first);
                }
            }
        }
        return max_earliest - INT_MIN;
    }

    void PrintKeyPath() {
        for (auto& node : nodes) {
            for (auto& next : node.next) {
                if (next.first->earliest == next.first->latest && next.first->earliest
== node.earliest + next.second)
                    cout << node.id + 1 << "->" << next.first->id + 1 << endl;
            }
        }
    }

    void FixOrder() {
        bool (*comp)(std::pair<Node*, int>p1, std::pair<Node*, int>p2) =
            [](std::pair<Node*, int>p1, std::pair<Node*, int>p2) { return p1.first->id
< p2.first->id; };
        for (auto& node : nodes) {
            std::sort(node.next.begin(), node.next.end(), comp);
            // std::sort(node.last.begin(), node.last.end(), comp);
            // std::reverse(node.next.begin(), node.next.end());
        }
    }

    void PrintLines() {
```

```
        for (auto node : nodes) {
            for (auto& next : node.next) {
                cout << node.id + 1 << ' ' << next.first->id + 1 << ' ' << next.second
<< endl;
            }
        }
    }

    void Build(const int n) { for (int i = 0; i < n; i++) nodes.push_back(Node(i)); }
    void Clear() { nodes.clear(); }
    Picture() = default;
    Picture(const int n) { Build(n); }
    ~Picture() = default;
private:
    std::vector<Node> nodes;
};
```

```
7 8
1 2 4
1 3 3
2 4 5
3 4 3
4 5 1
4 6 6
5 7 5
6 7 2
17
1->2
2->4
4->6
6->7
```

拓扑序是按照点的先后顺序排列的，也就是说入度为 0 的点一定是排在前面的，我们直接对一个图 BFS 一遍，BFS 过程中更新每个点的入度，如果一个点的入度为 0，那么就将其加入拓扑序，并且删除其与后继结点的所有边。
不是所有的有向图都是有拓扑序的，只有有向无环图才有拓扑序，所以有向无环图又被称为拓扑图。
在读入边的时候，直接计算点的入度。
若一个由图中所有点构成的序列 A 满足：对于图中的每条边(x, y)，x 在 A 中都出现在 y 之前，则称 A 是该图的一个拓扑序列。
拓扑序列都是针对有向图而言的
将图中的节点按拓扑序列排好后，所有的边都是从前指向后的，没有反向边
只要存在环，则一定没有拓扑序列
一个有向无环图一定存在一个拓扑序列（有向无环图也叫拓扑图）
所有入度为 0 的点都有可能引导一个拓扑序列
一个有向无环图一定存在一个入度为 0 的点

2.4 题目四

HW4：图状结构 > 4-4 最短路径

单源最短路径
描述
本题给出一张交通网络图，列出了各个城市之间的距离。请计算出从某一点出发到所有点的最短路径长度。

输入
第一行包含三个整数 n、m、s，分别表示 n 个顶点、m 条无向边、出发点的编号。

接下来 m 行，每行包含三个整数 ui、vi、wi，其中 1<=ui,vi<=n,1<=wi<=1000，分别表示第 i 条无向边的出发点、目标点和长度。

顶点编号从 1 开始。

输出
一行，包含 n 个用空格分隔的整数，其中第 i 个整数表示从点 s 出发到点 i 的最短路径长度

（若 s=i 则最短路径长度为 0，若从点 s 无法到达点 i，则最短路径长度为 2147483647，用 INT_MAX 表示）

```
class PICTURE {
public:
    struct NODE;
    typedef std::priority_queue<std::pair<int, NODE*>, std::vector<std::pair<int, NODE*>>, std::greater<std::pair<int, NODE*>>> PRIQUEUE;
    struct NODE {
        int id, min_distance;
        std::vector<std::pair<NODE*, int>> next;

        inline void WriteMinDistance(std::set<NODE*>& S, PRIQUEUE& que) {
            for (auto n : next) {
                if (n.first->min_distance > min_distance + n.second && S.find(n.first) == S.end()) {
                    n.first->min_distance = min_distance + n.second;
                    que.emplace(std::make_pair(min_distance + n.second, n.first));
                }
            }
        }
        NODE(int id_) : id(id_) { min_distance = INT_MAX; }
        ~NODE() = default;
    };
    PICTURE(int size) {
        for (int i = 0; i < size; i++)
            list.push_back(NODE(i));
```

```
    };

    void InsertLine(int s, int e, int w) {
        list[s].next.push_back(std::make_pair(&list[e], w));
    }

    void Solve(int start) {
        std::set<NODE*> S;
        PRIQUEUE que;
        list[start].min_distance = 0;
        que.emplace(std::make_pair(0, &list[start]));
        while (S.size() < list.size() && !que.empty()) {
            int min_dist = que.top().first;
            NODE* select = que.top().second;
            que.pop();
            if (S.find(select) != S.end())
                continue;
            S.insert(select);
            select->WriteMinDistance(S, que);
        }
    }

    void PrintMinDistance() {
        for (const auto i : list)
            cout << i.min_distance << ' ';
        cout << std::endl;
    }
private:
    std::vector<NODE> list;
};
```

```
4 6 1
1 2 2
2 3 2
2 4 1
1 3 5
3 4 3
1 4 4
0 2 4 3
```

图的最短路径

最短路径问题：如果从有向图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。
下面讨论两种最常见的最短路径问题：单源最短路径问题和所有顶点之间的最短路径。
如何求得这些路径，有一个按路径长度递增的次序产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点 v 到其

它各顶点的最短路径全部求出为止。
解决步骤描述：
1. 设置辅助数组 dist。它的每一个分量 dist[i]表示当前找到的从源点 v0 到终点 vi 的最短路径的长度；
2. 初始状态：
2.1. 若从源点 v0 到顶点 vi 有边：dist[i]为该边上的权值；
2.2. 若从源点 v0 到顶点 vi 无边：dist[i]为∞。
根据以上描述，可以得到如下描述的算法：
假设用带权的邻接矩阵 Edge[i][j]表示边(vi,vj)上的权值。若(vi,vj)不存在，则置 Edge[i][j]为∞。
S 为已找到从 v 出发的最短路径的终点的集合，它的初始状态为空集。
1.初始化： S ← {v0 };dist[j] ← Edge[0][j], j = 1, 2, ···, n-1;
2.找出最短路径所对应的点 K：dist[k] == min { dist[i] }, i ∈ V- S ;S ← S U { k };
3.对于每一个 i ∈ V- S 修改：dist[i] ← min{ dist[i],dist[k] + Edge[k][i] };
4.判断：若 S = V, 则算法结束，否则转 2。
算法的精髓：S 集内的顶点是已经找到最短路径的顶点，V0 到 w 的最短路径只能通过 S 集内的顶点。

2.5 题目五

HW4：图状结构 > 4-5 小世界现象

六度空间
描述
六度空间理论又称小世界理论。理论通俗地解释为：“你和世界上任何一个陌生人之间所间隔的人不会超过 6 个人，也就是说，最多通过五个人你就能够认识任何一个陌生人。"如图 1 所示。

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。

说明：由于浮点数精度不同导致结果有误差，请按 float 计算。

输入
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≤2000 ，表示人数）、边数 M（≤33×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
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%”。

```
class PICTURE {
public:
    struct NODE {
        int id, count;
```

```
        std::vector<int> next;
        NODE(int i) : id(i) { count = 0; }
        ~NODE() = default;
    };
    void InsertLine(int s, int e) { list[s].next.push_back(e); }
    void Solve() {
        for (auto& node : list) {
            bool chart[20000] = { 0 };
            chart[node.id] = true;
            std::queue<std::pair<int, int>> que;
            que.push(std::make_pair(0, node.id));
            while (!que.empty()) {
                node.count++;
                int count = que.front().first;
                NODE& n = list[que.front().second];
                que.pop();
                for (auto next : n.next) {
                    if (!chart[next]) {
                        chart[next] = true;
                        if (count < 6)
                            que.push(std::make_pair(count + 1, next));
                    }
                }
            }
        }
    }
    void FixOrder() {
        for (auto& i : list) {
            std::sort(i.next.begin(), i.next.end());
        }
    }
    void PrintLines() {
        for (auto& i : list) {
            for (std::vector<int>::iterator it = i.next.begin(); it != i.next.end();
it++)
            {
                if (*it >= i.id)
                    cout << i.id + 1 << ' ' << *it + 1 << std::endl;
            }
        }
    }
    void Print() {
        for (const auto& node : list) {
            cout << node.id + 1 << ':' << ' ' << std::fixed << std::setprecision(2) <<
```

```
round(node.count * 10000.0 / list.size()) / 100.0 << '%' << std::endl;
        }
    }
    PICTURE(const int n) { for (int i = 0; i < n; i++) list.push_back(NODE(i)); }
    ~PICTURE() = default;
private:
    std::vector<NODE> list;
};
```

```
10 9
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
1: 70.00%
2: 80.00%
3: 90.00%
4: 100.00%
5: 100.00%
6: 100.00%
7: 100.00%
8: 90.00%
9: 80.00%
10: 70.00%
```

（1）将输入的关系建立成一个二维数组 relation[人数][人数]，其中第 i 行第 j 列为 1 则代表 i 与 j 有关系
（2）用一个数组 visit[人数]记录第 i 个人是否已经访问
（3）用一个数组 queue[人数]记录需要遍历其广度的人的队列
（4）从编号为 1 的人开始，记录下他的一层所有关系的广度，每遍历到一个人，如果这个人对应的 visit 数组中为 0，则代表这个人还未访问过，将这个遍历到的人入 queue 数组，再将对应的 visit 中的数值改为 1，如果到了 relation 数组第 n 维的最后一个人，用 tail 记录下这个人的编号，代表着此层的最后一个人。
（5）遍历 queue 数组的人的关系，当在 queue 中遍历到此 tail 编号的人的广度时,层数加 1
（6）当层数为 6 时，停止该人的遍历，转到下个编号的人的遍历

3. 实验总结

带权图的边结点中 info 保存该边上的权值
顶点 Vi 的边链表的头结点存放在下标为 i 的顶点数组中
在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定
设图中有 n 个顶点，e 条边，则用邻接表表示无向图时，需要 n 个顶点结点，2e 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 n 个顶点结点，e 个边结点

建立邻接表的时间复杂度为 O(n*e)。若顶点信息即为顶点的下标, 则时间复杂度为 O(n+e)。
取 lowcost[i] = min{ lowcost[i], G.arcs[k][i] }，即用生成树顶点集合外各顶点 i 到刚加入该集合的新顶点 k 的距离 G.arcs[k][i] 与原来它们到生成树顶点集合中顶点的最短距离 lowcost[i] 做比较， 取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。
如果生成树顶点集合外顶点 i 到刚加入该集合的新顶点 k 的距离比原来它到生成树顶点集合中顶点的最短距离还要近，则修改 adj：adjvex = v。表示生成树外顶点 i 到生成树内顶点 k 当前距离最近。
检测是否存在有向环的一种方法是：
对 AOV 网络构造它的拓扑有序序列；
即将各个顶点 (代表各个活动)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。
这种构造 AOV 网络全部顶点的拓扑有序序列的运算就叫做拓扑排序。
如果通过拓扑排序能将 AOV 网络的所有顶点都排入一个拓扑有序的序列中，则该 AOV 网络中必定不会出现有向环；相反，如果得不到满足要求的拓扑有序序列，则说明 AOV 网络中存在有向环，此 AOV 网络所代表的工程是不可行的。