作业 HW4 实验报告

姓名：段威呈 学号：2252109 日期：2023 年 12 月 9 日

1. 涉及数据结构和相关背景

图是一种非线性的数据存储结构，可以用来存储具象的任意拓扑图结构，也可以展现多个元素结点的链接关系、先后次序等逻辑关系。图由边和结点构成，边用于连接每个结点的多个前驱和后继，根据边是否具有方向性可以把图分为有向图或无向图。

图常见存储方式有如下几种：

邻接矩阵：二维数组存储图结构，行标表示代表结点起点，列标代表结点终点，两者对应的矩阵的点存储的是边的连接情况（一般用 0 代表边未连接，1 代表连接）或边权重（连接则标边权，未连接则标注∞）。

	V1	V2	V3	V4
V1	0	0	0	1
V2	1	0	1	1
V3	0	0	0	0
V4	0	0	0	0

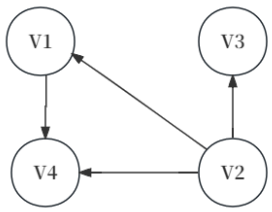


图 1.1 邻接矩阵

对于邻接矩阵，可以通过访问行直接获得每个结点的出度；通过访问列，获得每个结点的出度；邻接矩阵实现方式简单，但是邻接矩阵的缺点是占用空间较大，同时在遍历访问的时还需再重复访问未连接的结点。

邻接链表：邻接链表对于有向图借助通过直接连接对应相连结点编号实现节点存储。如下图：

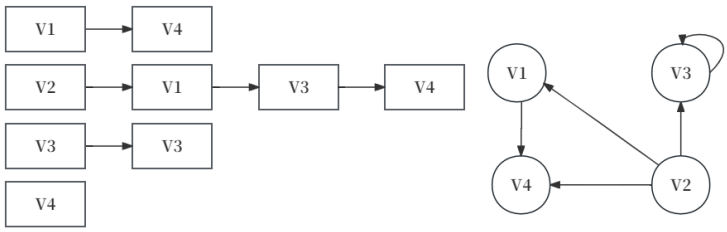


图 1.2 邻接链表

邻接链表的只存储了每个参与连接的结点，即需要添加的总结点空间数等于图中边的个数。但以上结构对于有向图，不便于访问每个结点的入度，且手写链表实现方式困难。在本次实验中，多选用结点元素结构体嵌套 vector 的方式实现邻接链表，每个元素结构体分别嵌套入度结点链表与出度结点链表，可以实现有向图前驱与后继的直接访问。结构如下：

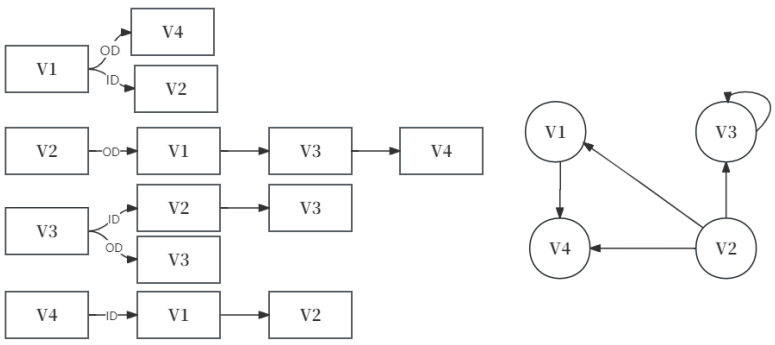


图 1.3 邻接链表-改进

2. 实验内容

2.1 图的遍历

2.1.1 问题描述

本题给定一个无向图，用 dfs 和 bfs 找出图的所有连通分量。
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：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。
对于 100%的数据，有 0<n<=1000；

2.1.3 数据结构设计

本题是一个无向图连接问题，选择使用邻接矩阵表示无向图需要对存储方式进行规定。由于无向图可以看做对称双向的有向图，因此无向图的邻接矩阵是一个对称矩阵，因此只需要存储半边三角即可。这里选择存储下半三角，即所有边都默认起点是大序号结点，终点是小序号结点，同时这样的方法还可以消除重边的读入。结构如下：

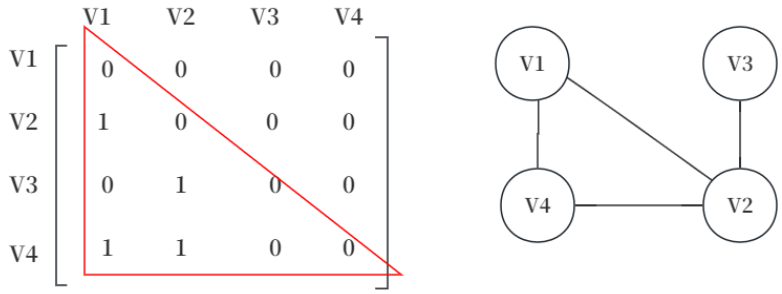


图 2.1.1 无向图邻接矩阵

因此需要在读入的时候进行序号交换判断：

```
if (src_node > dst_node) {
    tmp = src_node;
    src_node = dst_node;
    dst_node = tmp;
}
```

本题由于需要用到 BFS 算法，要借助队列辅助。队列结构定义如下：

```
typedef struct queue {//定义队列的结构
    int* base;
    int front;
    int rear;
};
```

队列功能函数：

① 初始化队列：InitQueue()：

```
void InitQueue(queue& Q) {
    Q.base = (int*)malloc(MAX_VERTEX_NUM * sizeof(int));
    if (!Q.base) exit(-1);
    Q.front = Q.rear = 0;
}
```

② 元素入队：enqueue():

```
void enqueue(queue& Q, int e) {    //在队列的尾端添加元素e
    Q.base[Q.rear] = e;
    ++Q.rear;
}
```

③ 元素出队，并返回出队元素：‘’

```
int dequeue(queue& Q) {        //在队列首端删除元素并返回删除的值
    int e = Q.base[Q.front];
    ++Q.front;
    return e;
}
```

④ 判断队列是否为空

```
bool isEmpty(queue& Q) {
    if (Q.front == Q.rear)
        return 1; //返回1意味着为空
    else
        return 0;  //返回0不为空
}
```

2.1.4 功能说明（函数、类）

本题直接考察了 BFS 与 DFS 算法。

①BFS 算法要求按从小到大的顺序依次遍历每层的所有的序列，因此只需要按顺序将每次遍历的元素入队后出队，再遍历出队元素所连接的元素再从小到大依次入队即可。过程如下：

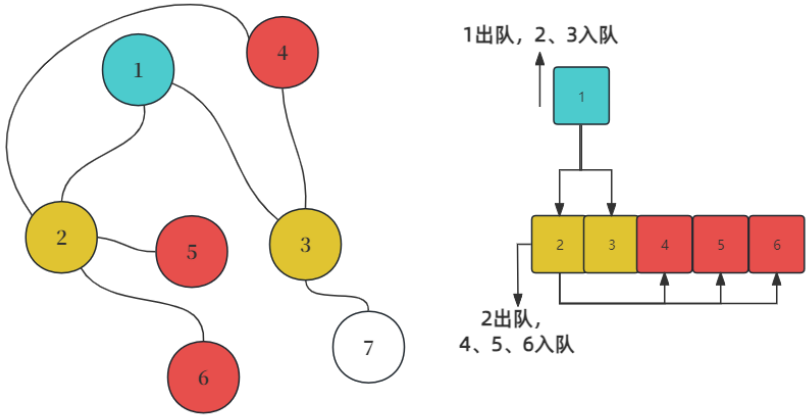


图2.1.2 图的广度优先搜索过程演示

BFS()代码如下：

```
static int vex[MAX_VERTEX_NUM];   //记录遍历情况，没有被遍历则是0，被遍历则是1
void BFS(int src_node, int BFS_Result[MAX_VERTEX_NUM]) {
    queue Q;
    q.InitQueue(Q);
    q.enqueue(Q, src_node);
    vex[src_node] = 1;//代表元素已经遍历过
    int tmp_node;
    int length = 0;
    while (q.isEmpty(Q)!=1) {
        tmp_node = q.dequeue(Q);
        for (int i = 0; i < vexnum; i++) {
            if ((AdjMatrix[tmp_node][i] == 1 || AdjMatrix[i][tmp_node] == 1) && vex[i] != 1) {   //有相连的结点（包括tmp_node连接的与连接tmp_node的）
                q.enqueue(Q, i);
                vex[i] = 1;
            }
        }
        BFS_Result[length++] = tmp_node;//记录出队结点的结果
    }
}
```

其中需要借助一个 vex[i]来记录第 i 个结点的遍历情况，初始时 vex 中所有值都设置为 0，每次元素 i 入队后，对应的 vex[i]值都变为 1，代表已经被遍历过，则之后元素 i 不再入队参与遍历。

②DFS 算法是依次对图的每一支一支遍历到底，再回溯到起点遍历另一支。

由于中间存在回溯操作，通常意义上讲可以借助栈结构进行回溯，即每次遍历都讲元素入栈，若栈顶元素还未遍历的相连结点，则将相连结点从小到大依次入栈，否则将栈顶元素出栈，再检查下一个栈顶元素。

除了使用以上用栈方法存储回溯结点，还可以借助递归的方法实现。即一个从一个头结点开始对其所有相连接的结点进行序号从小到大的试探检索，即让试探结点去依次从小到大试探自己的其他相连结点，体现在代码中就是将试探结点作为顶结点传入 DFS 函数进行试探；每次试探的过程都保存试探的结点，直到作为头结点的试探结点没有相邻结点，则某一支从起点到结束的试探检索结束，这时候递归函数会自动返回上一级“父亲”结点再重新试探其另外的“兄弟”结点。

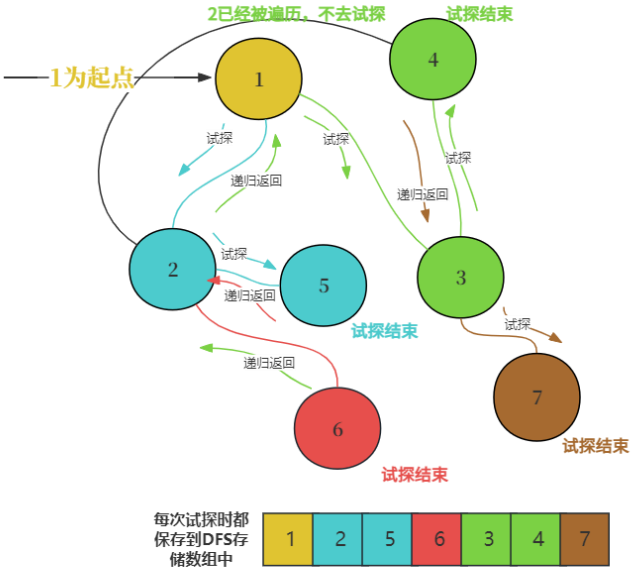


图2.1.3 图的深度优先搜索过程演示

DFS()代码如下：

```
static int Count = 0;
void DFS(int src_node,int DFS_Result[MAX_VERTEX_NUM]) {
    DFS_Result[Count++] = src_node;
    vex[src_node] = 1;
    for (int i = 0; i < vexnum; i++) {    //遍历src_node连接的所有可能的结点(序号从1~vexnum全遍历一遍)
        if (vex[i] != 1) {      //没有被遍历过
            if (AdjMatrix[src_node][i] == 1 || AdjMatrix[i][src_node] == 1) {  //该结点连接有结点i
                DFS(i, DFS_Result);
            }
        }
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

本题在数据读入方面需要有特别注意的地方。由于并不是所有结点都建立了连接，因此在读入的时候连接行数并不是固定的，因此读入结束判断要根据后续输入为空作为终止条件。经过多种方法尝试，这里选择使用 cin.peek()函数实现，cin.peek()只会读取后续一位字符，但是不会移位。另外注意，在每一行结尾还会有'\n'，需要先用 getchar()读取掉，否则会 cin.peek()读取到的永远都只是'\n'。

读取图连接关系的部分代码如下：

```
getchar();//读入空格
if(cin.peek()==EOF)
    break;
scanf("%d %d", &src_node, &dst_node);
```

2.1.6 总结和体会

本题直接考察了图最常用的两种遍历方法，BFS（广度优先搜索）与 DFS（深度优先搜索）。在实际问题中，两者各自有不同的用途。如需要寻找路径的时候需要使用 DFS；在依次获取层序信息的时候，需要借助 BFS。另外，BFS 和 DFS 不仅仅是在图结构中的方法，早在树结构中就已使用过。事实上深度与广度遍历方式并不局限于树、图等结构，对于任意具有邻接关系的元素都可以使用这两种方式进行遍历。

2.2 小世界现象

2.2.1 问题描述

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。" 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。

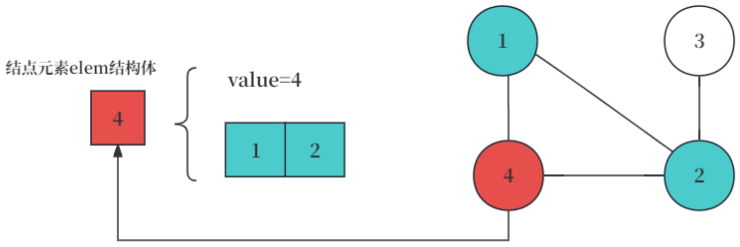
2.2.2 基本要求

输入：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%"。

2.2.3 数据结构设计

本题最大数据量共有 2000 个结点，如果使用邻接矩阵，则需要开一个 2000*2000 的矩阵，但是社交关系却最多只有 33*2000 个，空间利用率不足 2%，带来巨大的空间浪费。此外，每次对结点遍历都需要遍历为连接的结点，带来巨大的时间成本开销。因此这里不建议使用邻接矩阵完成。于是考虑使用邻接链表。

邻接链表结构需要单独开链表，构造起来较为复杂，这里选择直接使用 STL 容器中的 vector 来替代链表的手写，对于每个元素结点，构造如下元素结构体：




```
typedef struct elem {
    int value;
    vector<int>Link;
};
```

图 2.2.1 结点元素结构体

2.2.4 功能说明（函数、类）

本题直接使用 BFS 进行遍历即可。但是注意只需遍历 6 层，并记录是否可以遍历所有节点。这里需要注意，这与在题目一种的 BFS 不同，这里要求同一层的节点元素同时出队，即上一次入队的结点这一次要同时出队。过程如下：

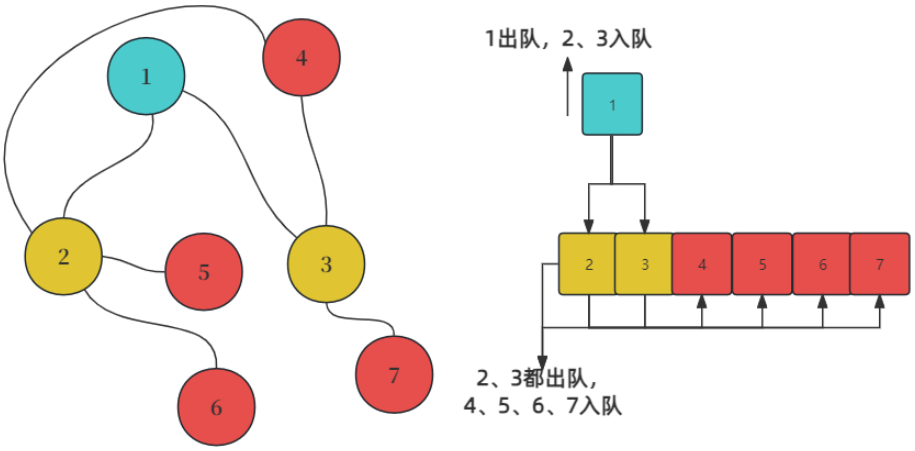


图2.2.2 图的层序广度优先搜索过程演示

层序广度优先搜索的 BFS()函数如下：

```
static int vex[MAX_VERTEX_NUM];   //记录遍历情况，没有被遍历则是0，被遍历则是1
int BFS(elem src_node) {
    int Count = 0;
    int tmp_count=0;
    int tmp_c=0;
    queue<elem> Q;
    Q.push(src_node);
    vex[src_node.value] = 1;
    ++tmp_count;
    ++Count;
    elem tmp_node;
    int length = 0;
    for(int j=1;j<=6;j++){ //总共搜索6层
        tmp_c = 0;
        for (int i = 0; i < tmp_count; i++) {
            tmp_node = Q.front();
            Q.pop();
            //将tmp_node相连的结点入队
            for (int k = 0; k < tmp_node.Link.size(); k++) {
                if (vex[UDG[tmp_node.Link[k]].value] == 0) {
                    Q.push(UDG[tmp_node.Link[k]]);
                    vex[UDG[tmp_node.Link[k]].value] = 1;
                    ++tmp_c; //统计本层入队的个数
                    ++Count;
                }
            }
        }
        tmp_count = tmp_c;
    }
    return Count;
}
```

2.2.5 调试分析（遇到的问题和解决方法）

本题一开始选择使用了邻接矩阵实现，但是提交到 OJ 平台上发现有测试点超时。分析原因才发现是在遍历矩阵的过程中遍历的绝大部分边都是为连接的 0，造成了大量的时间开销，因此

改用邻接链表的结构存储图，这里为了简化操作，选择用 STL 容器中的 vector 直接代替手写链表。最终提交到 OJ 平台上所有结点都通过。

2.2.6 总结和体会

通过本题发现，常规情况下邻接链表都会表现出比邻接矩阵更好的访问效率以及更小的内存空间占用。

但是对于有向图，如果直接采取邻接链表存储，由于每个链表头结点后连接的都是其出度结点，因此对于一个结点的入度结点访问比较麻烦。本题使用的一种邻接链表变形的存储结构则可以有效解决此问题，通过对元素构造结构体，让每个元素同时包含入度和出度结点两个 vector，从而可以出、入的直接访问。虽然本题无向图并未用到该方法，但是可以为后续有向图题目提供一个可行的解决方案。

2.3 村村通

2.3.1 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

第一行包含一个整数 n（3<=n<=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<=a<b),表示在村庄 a 和 b 之间已经修建了路。

2.3.3 数据结构设计

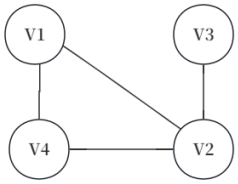
本题由于给出了每一个村子之间的距离权值，在读入之初即便未连接也需要保存，因此必须使用邻接矩阵来存储图结构，保存权重的邻接矩阵为 AdjMatrix_weight；

除此之外，还需要一个 AdjMatrix_link 矩阵存储村与村之间的连接状态，有连接则为 1,没有连接则为 0。

另外注意，无向图均保存下半三角矩阵即可，这样的目的是避免遍历时候会遍历重边造成结果错误。

两矩阵结构如下图：

	V1	V2	V3	V4
V1	0	0	0	0
V2	1	0	0	0
V3	0	1	0	0
V4	1	1	0	0



	V1	V2	V3	V4
V1	0	0	0	0
V2	450	0	0	0
V3	300	40	0	0
V4	600	200	870	0

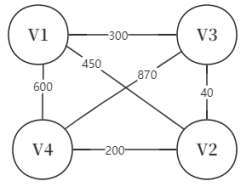


图 2.3.1 权重矩阵与连接矩阵

2.3.4 功能说明（函数、类）

1、算法选择分析

本题用最小生成树解决。最小生成树常见有 Prim 和 Kruskal 算法。这里使用 Prim 算法会较难处理。常规的 Prim 算法解决最小生成树问题是将已连接好整体大部分作为一个集合，，选择该大集合与其他散点的最短连接即可。

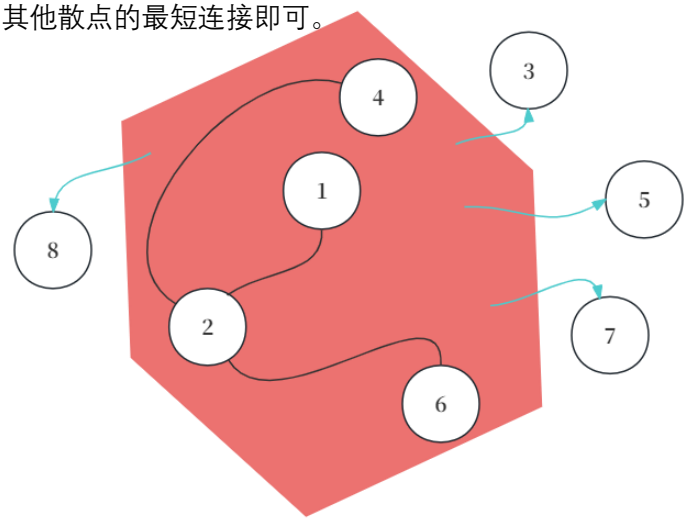


图2.3.2 常规最小生成树问题

但是因为本题在一开始给出了一些连接好的结点。如果存在如下结构:

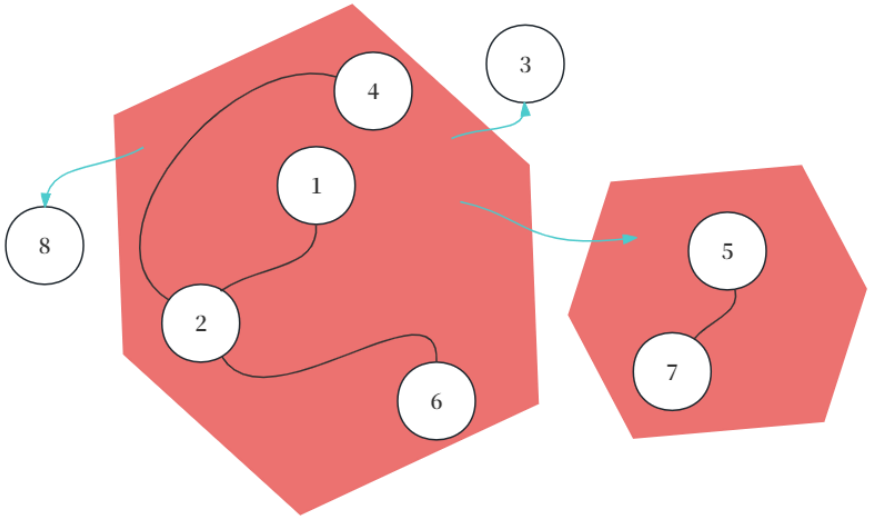


图2.3.3 本题可能出现的情况

即有可能会预先出现两组连接好的大集合，这时需要将其中一个集合看做散点来处理。这需要先对整个图进行连通性的遍历，找出各个集合并将其映射为一个新的点，无形中给代码编写增加了巨大的工作量。因此这里选择使用 Kruskal 算法完成最小生成树的构建。

Kruskal 算法的核心思想是依次连接最短且不会使得图成环的边，直到所有点都被连接。

2、快速排序

在上一步构建权重和邻接矩阵的过程中已经保存了还未连接的各个边的权值，因此首先需要将所有边从大到小依次进行排序。这里选择效率较高的快速排序方法进行排列，其用到了分治思想的方法，QuickSort 具体操作流程如下:

① 选择序列的第一个元素为枢轴元素，并用 middle 指针指向该枢轴元素的位置。

以下步骤的目的是将序列调整顺序，直到枢轴元素左侧的元素的值都小于枢轴元素，枢轴元素右侧的所有元素的值都大于枢轴元素。

② 头尾 i，j 指针开始移动。

i)先移动尾指针 j，从尾向头移动，直到指向一个小于枢轴元素的值；

ii)之后开始移动头指针 i，从头向尾移动，直到指向一个大于枢轴元素的值；

iii)执行完成 i)、ii)后，交换 i，j 指针所指向位置的元素；

重复以上 i)、ii)、iii)步骤，直到 i，j 指针重合，交换 i（或 j）指向位置与 middle 指针指向的枢轴元素。在该过程中 iii)步骤可能一直不会执行；

③ 执行完毕①、②后，枢轴元素已经调整到序列的中间部分，其左侧的元素都比枢轴元素更小，其右侧的元素都不枢轴元素更大。这时候把两端的序列也当成新序列，重新传入 QuickSort（）函数进行排序（分治思想），每次重复执行①、②步骤；

④ 直到最终传入数列的长度为 1，结束排序。

QuickSort()代码如下：

```
void QuickSort(int low, int high) { //快排
    if (low >= high) {  //若待排序序列只有一个元素，返回空
        return;
    }
    int tmp;
    int i = low;    //i作为指针从左向右扫描
    int j = high;   //j作为指针从右向左扫描
    OrderElem key = (vex_order[low]);//第一个数作为基准数
    while (i < j) {
        while ((vex_order[j]).d >= key.d && i < j) {    //从右边找小于基准数的元素 （此处由于j值可能会变，所以仍需判断i是否小于j）
            j--;    //找不到则j减一
        }

        while ((vex_order[i]).d <= key.d && i < j) {    //从左边找大于基准数的元素
            i++;    //找不到则i加一
        }
        swap(i, j);
    }
    swap(low, i);//相遇时则交换
    QuickSort(low, i - 1);  //i左边的序列继续递归调用快排
    QuickSort(i + 1, high); //i右边的序列继续递归调用快排
}
```

通过快速排序，已经将未连接的所有边按照权重次序从小到大排列。下一步需要将这些边按照从小到大的顺序依次填入图中，并保证每次填入不成环。

3、判断成环

对于一个图是否成环，需要进行拓扑结构分析。

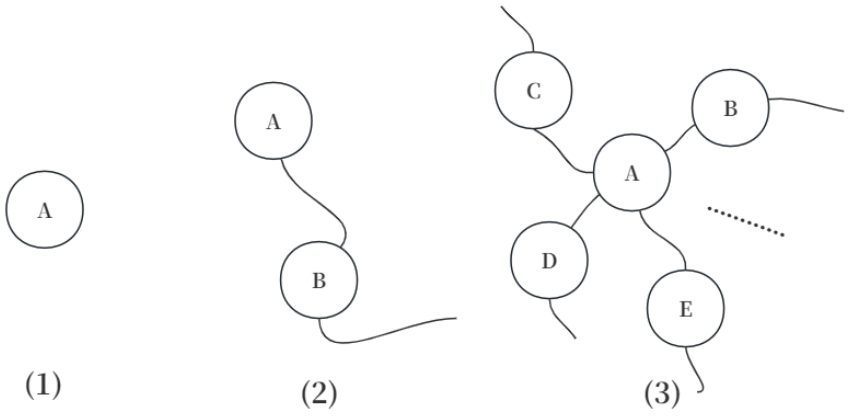


图2.3.4 结点可能的拓扑结构

如上，可以将结点分为两类，第一类是（1）中所示单点，度 TD=0；第二类是（2）中所示的单连接，度 TD=1，这可以被视作尾结点；第三类是（3）中所示的多连接的结点，度 TD>1，显然只有此类结点才可能会参与产生闭环。如果（3）中结构经过深度搜索遍历可以得到一个尾结点，说明则该图中不存在环。

为了判断是否存在环，可以考虑一种逆向的从外到内的检索。即从尾结点开始，不断删去尾巴结点与单节点，

这种操作会使得原来的第（3）类多连接的结点中的部分退化成为尾结点（如图 2.3.5）

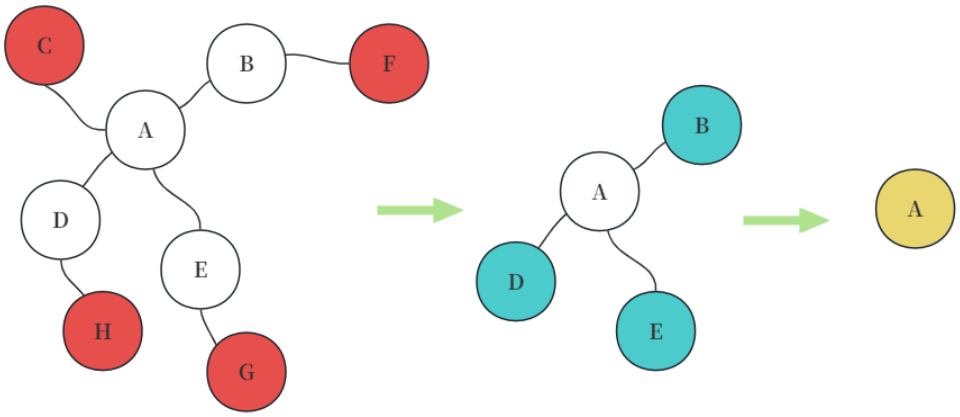


图2.3.5 无环

维护一个度 TD≤1 的集合（在代码编写中可以选择使用队列实现），每次将其中的元素删除，并加入新的图中 TD≤1 的元素，再重复执行“删除->增添操作->删除->……“操作。

若该图没有环，最终必然可以获得一个单点（图 2.3.5 情况），再次执行操作，则图中结点全部被删除。至此，度 TD≤1 的集合为空，总执行删除操作次数等于总结点数 n。

但若图有环，则图不会被完全删除（如图 2.3.6）

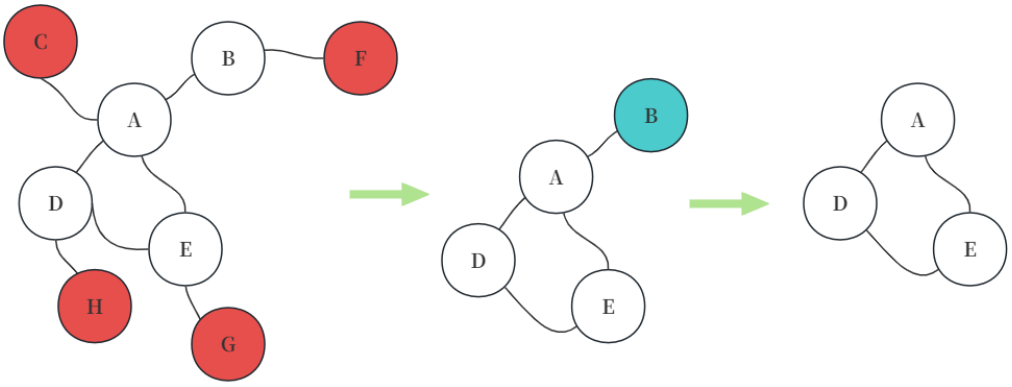


图2.3.6 有环

在这种情况下，度 TD≤1 的集合为空，但总执行删除操作次数小于总结点数 n。

因此判断是否有环的过程可以抽象为如下算法：

① 将所有 TD≤1 的结点入队，并从图中删除；

②将队列所有元素依次出队，出队的时候，如果该出队元素原先所相连的元素（若有相连元素）在①删除后得到的新图中的度 TD=1，则该元素也入队；

③重复进行②操作，直到队列为空，如果总出队次数等于总结点个数 n，则说明图没有环。反之则说明有环。

该过程被写为判断成环的函数 is_Circle()，有环返回 1，无环返回 0。代码如下：

```
    //计算i的度
    for (int j = 1; j <= vexnum; j++) {
        if (AdjMatrix_link[i][j] == 1 || AdjMatrix_link[j][i] == 1) {
            sum += 1;
            tmp.link_value = j;
        }
    }
    //判断度是否<=1
    if (sum <= 1) {
        q.enqueue(Q, tmp);

    }
}
//②依次出队，每次出队都+1，如果其相连的结点的度也是1
while (q.isEmpty(Q) != 1) { //队列不为空时
    ++Count;//记录遍历过的结点的个数
    tmp = q.dequeue(Q);
    vex[tmp.value] = -1;//断开连接
    if (tmp.link_value != -1) {  //有相连
        sum = 0;
        tmp2.value = tmp.link_value;  //更新相连结点
        tmp2.link_value = -1;
        //计算连接结点的度
        for (int k = 1; k <= vexnum; k++) {
            //要除去该出队结点
            if ((AdjMatrix_link[tmp.link_value][k] == 1 || AdjMatrix_link[k][tmp.link_value] == 1)&& vex[k]!=-1) {
                sum += 1;
                tmp2.link_value = k;

            }
        }
        //度等于1，入队
        if (sum == 1) {
            q.enqueue(Q, tmp2);
        }
    }
}
//③判断出队次数
if (Count == vexnum)
    return 0;  //无环
else
    return 1;  //有环
```

2.3.5 调试分析（遇到的问题和解决方法）

①题完成至此仍有未考虑完全的地方，且该情况在 OJ 平台上的测试数据中并没有体现。即如果一开始的连接即包含成环的情况，将要如何解决？由于一开始被判定成环，则后续的所有操作都不能根据 Kruskal 算法中判断不能成环的情况进行判定。换句话讲，题目一开始给出的连接结构不是树结构，将如何应用 Kruskal 算法？

如果在一开始就把这些连接好的数据点排除在外，只对未连接的点构成的集合应用 Kruskal 算法，则就相当于忽略了数据点构成的集合会失去与剩下点的连接，必然会导致结果错误；

这里想到了一种连接距离等效拓扑结构的方式：即断开环的一边。这样做的意义在于，首先破坏了环；其次没有破坏点与点之间的连接，因此不会改变一开始的题给情形，结果也必然与原来的情况是等价的。

为了破坏环，第一步就是要寻找环。在编写的 is_Circle()函数中，恰好用到一个记录了变量结点信息的 vex 数组，其中为被遍历的元素（仍为 0）的元素结点是环的结点，因此直接对这部分结点进行断开操作即可。

②本题在 oj 平台的测试样例中有关于重边的测试数据，即既有 A->B 连接的输入，也有 B->A 连接的输入，这种情况下，如果不把无向图邻接矩阵存为下（上）半三角，则对整个邻接矩阵进行遍历来查询连接节点时候，必然会把 A 和 B 连接重复遍历了两次，可能会带来结果的错误。因此为了避免重边导致的结果误差，一定要把无向图的邻接矩阵存为上（下）半三角。

2.3.6 总结和体会

本题是最小生成树相关的问题，但是原有的 Prim 和 Kruskal 都不能直接套到此题目上，需要做一些调整。要了解算法的本质和拓扑图结构的特点才能对算法进行合理的修改。

此外，在排序方面考虑到巨大的测试数据量，本题选择使用了较为复杂但是效率更高的快速排序。

特别注意，尽管本题对村与村距离权值的范围取值是 1~1000，数字比较小，但是本题也不要使用数组键值索引的方法进行排序（即以距离权值作为键，把边的信息直接存储到以距离权值数组下标的位置处），因为很有可能存在两个距离权值一样边，导致存储发生冲突（情况类似哈希冲撞）。但是该问题仍有其他的解决方法，如设计一层映射转换函数，避免冲撞的发生（一定保证所有 1~1000 的数不会得到同一个映射结果）；或设计一个存储链表结构，每个位置不止保存一个元素，而是保存一个链表，将所有距离权值相等的边元素保存到该位置的链表后（即构造哈希链表）。通过以上解决方式理论上也可以实现键值索引方法排序，且排序与检索的效率会更高。

2.4 给定条件下构造矩阵

2.4.1 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：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：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

本题应当考虑用邻接链表的方式实现。方式与题目二中实现方式类似：

```
struct elem {
    int value;//自己的序号
    vector<int> link_value;//相连结点的序号，等于-1代表没有相连
};
```

2.4.4 功能说明（函数、类）

1.判断是否可以构成矩阵

本题可以转化为一个拓扑排序的问题。将 row 与 col 实际上是独立的，构造符合要求的矩阵其实是构造两个符合要求的序列，而两个序列都可以用拓扑排序的方法完成构造。即把在前的数字可以当做在后的数字的前驱，从而构造出有向图。但是拓扑排序的前提条件是不可以构成强连通图（如图 2.4.1）

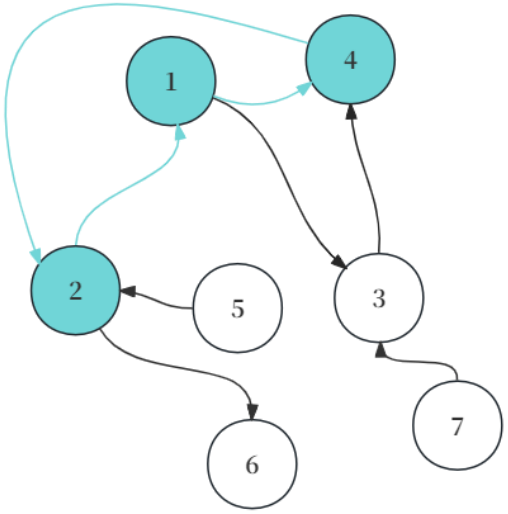


图 2.4.1 强连通图

其中 1->4->2 构成的环是强连通环，放在此题的背景中，1->4->2 环表示的意思是，2 要放在 1 前，1 要放在 4 前，根据传递性，则有 2 放在 4 前；但是根据图，又有 4 要放在 2 前的要求。因此出现了内部逻辑的矛盾。故拓扑排序存在的前提条件是图不能是强连通图。

不允许是强连通图不意味着不能存在环（如图 2.4.2）

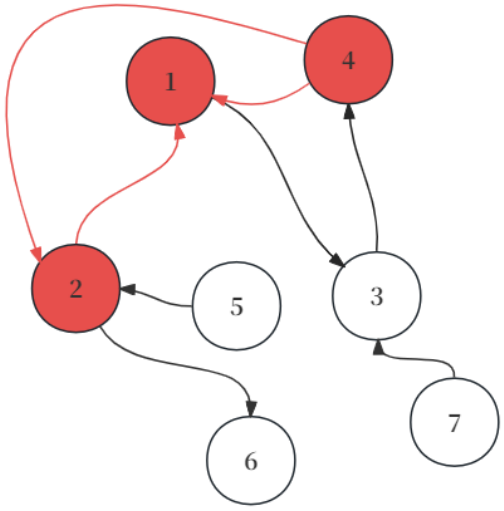


图 2.4.2 成环的有向图

显然对于此图，序列 4 1 2 满足要求。事实上本题的关键在于判断是否存在一个“绕一圈能回到自己”的环结构。

为了研究寻找这种环结构的方法，这里参考第三题中判断环的方法进行拓扑结构分析。但是

这里除了关注结点的度 TD，还要额外关注其中入度 ID 和出度 OD 的数目。

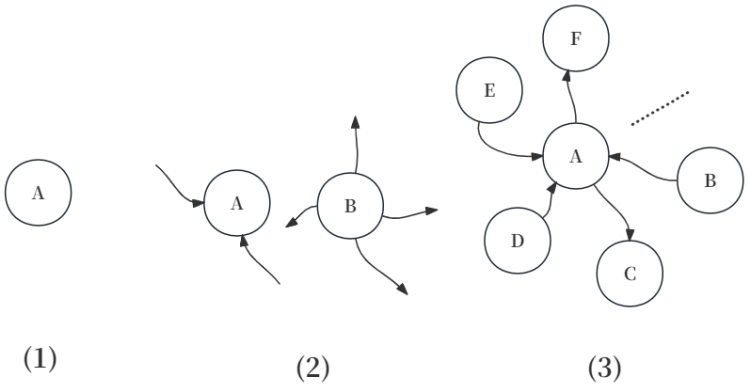


图 2.4.3 结点拓扑结构

以上拓扑结构中，（1）为单点，入度 ID=0，出度 OD=0；（2）的结构中，结点仅有入边或出边，在该情况下，必不可能通过深搜绕一圈再回到自己，说明（2）结构的结点不能作为强连通环的结点；（3）表示结点既有入边又有出边，因此可能作为强连通环的结点。

为了判断是否存在环，同样可以考虑一种逆向的从外到内的检索。即维护一个满足 ID*OD=0 的结点所构成的集合，每次将其中的元素删除，并加入新的图中 ID*OD=0 的元素，再重复执行“删除->增添操作->删除->……“操作。

这种操作会使得原来的第（3）类多连接的结点不断退化：

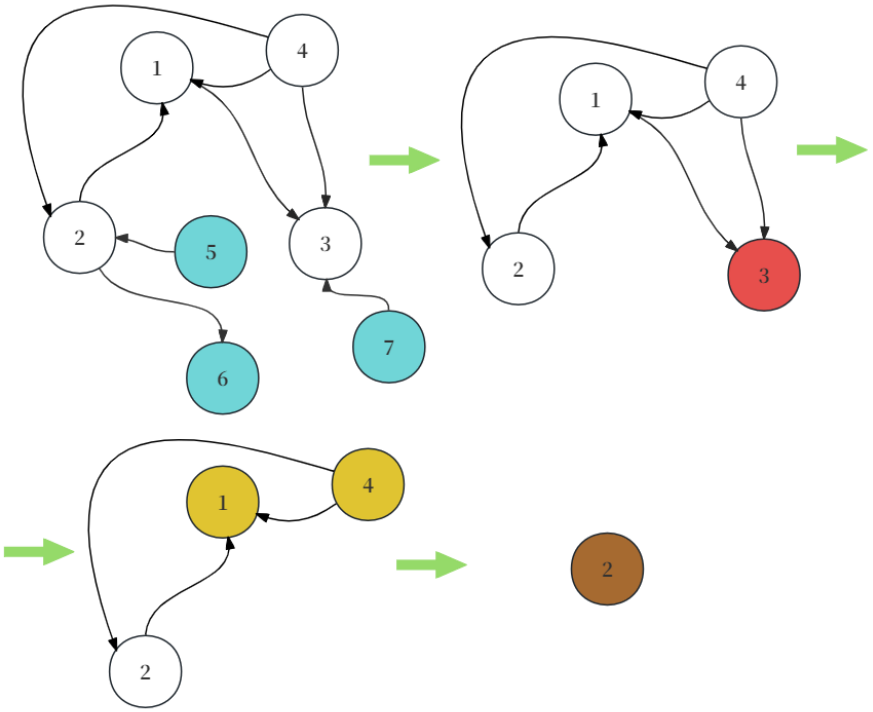


图 2.4.4 删除ID*OD=0的结点

在这种情况下，度 ID*OD=0 的集合为空，但总执行删除操作次数等于总结点数 n。

因此判断是否有环的过程可以抽象为如下算法：

② 将所有 ID*OD=0 的结点入队，并从图中删除；

③ 将队列所有元素依次出队，出队的时候，如果该出队元素原先所相连的元素（若有相连

元素）在①删除后得到的新图中的度 ID*OD=0 并且 ID 和 OD 都不为 0 的结点，则该元素也入队；

③重复进行②操作，直到队列为空，如果总出队次数等于总结点个数 n-1，则说明图没有环。反之则说明有环。

该过程与判断无向图是否有环的 is_Circle () 类似，仅仅只是入队条件不同。具体代码省略。

2. 拓扑排序

拓扑排序的方法是根据拓扑图来构成排序序列。算法的流程如下：

①维护入度 ID=0 的结点所构成的集合，每次将该集合所有的元素从图中断开连接，代表加入排序序列，并将这些点从集合中删除。

注意到，在本题中的数据保证有且仅有一个序列，因此每次仅可能出现一个入度为 0 的结点，直接将其保存到排序序列中即可；

②经过断开连接，必然会产生新的入度为 0 的结点，将其再次加入集合。

④ 重复①、②过程，直到图中所有结点都被断开删除，排序完成。

函数 Creat_Order()如下：

```
void Creat_Order(int DI[MAX_VERTEX_NUM],int Order[MAX_VERTEX_NUM],const int state) {
    int Count = 1;
    while (Count <= k) {  //每个点都要排进去，即按照题目要求每个点都可以成为一次入度为0的点
        //检索入度为0的点
        for (int i = 1; i <= k; i++) {
            if (DI[i] == 0) {   //i结点的入度为
                Order[Count++] = i;  //加入到排序序列中
                DI[i] = -1;
                if (state == 0) {  //行排列
                    //让i的连接值的vex入度都减1
                    for (int q = 0; q < vex_Link_row[i].link_value.size(); q++) {
                        --DI[vex_Link_row[i].link_value[q]];
                    }
                }
                if (state == 1) {  //列排列
                    //让i的连接值的vex入度都减1
                    for (int q = 0; q < vex_Link_col[i].link_value.size(); q++) {
                        --DI[vex_Link_col[i].link_value[q]];
                    }
                }
                break;
            }
        }

    }
}
```

3. 构造矩阵

该步骤只需要将两个序列组成一个矩阵即可。运用的方法类似织网格点相交（如图 2.4.5）

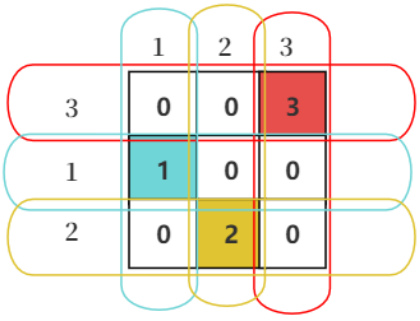


图2.4.5 两序列构成矩阵

2.4.5 调试分析（遇到的问题和解决方法）

本题一开始对于构成强连通图的算法不太清晰，导致编写修改多次都没有实现预期功能。事实上在整个遍历过程中，最终必然会留下一个全部连接都会被断开的结点，其计数统计不会被计算。因此判断不是强连通图的最终判据是遍历结点数目是 n-1（这时不能为了兼顾这个点而把入队的条件改为包含度为 0 的单点，因为这样会把原先已经断开连接的点再次入队，从而出现死循环）。

在这种问题出现的时候，最好的方法是手工进行一遍算法模拟看看最终结果。同时当代码出现死循环的时候，应考虑借助 cout 进行结果输出，从而发现出现死循环的原因。

2.4.6 总结和体会

本题是经典的拓扑排序题目，难点在于强连通图的判断。在本题中选择了一种根据拓扑图结构的“删除点、保留环”的方法。事实上本题也可以直接编写 DFS，判断遍历的时候是否会存在遍历回到自己的情况即可，其思路会相比这种方式更加直接。

2.5 必修课

2.5.1 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：
从入学开始，每门课程最早可能完成的时间（单位：学时）；
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：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：
第一个整数为 ti ，表示修完该课程所需的学时。
第二个整数为 ci，表示该课程的前置课程数量。
接下来 ci 个互不相同的整数，表示该课程的前置课程的编号
输出：输出共 n 行，第 i 行包含两个整数：
第一个整数表示编号为 i 的课程最早可能完成的时间。
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。
对于所有数据，满足：
$1 \leqslant n \leqslant 100$
$1 \leqslant ti \leqslant 100$
$0 \leqslant ci < n$
时间限制： 1 sec
内存限制： 256 MB

2.5.3 数据结构设计

本题由于有较强的程序执行时间效率和空间占用约束，因此这里应当选择采用更高效且节约内存的邻接链表的方式存储图结构。选择的是题目二中改进的邻接链表形式，这种形式可以便于直接访问结点的所有前驱和后继。结构如下图所示：

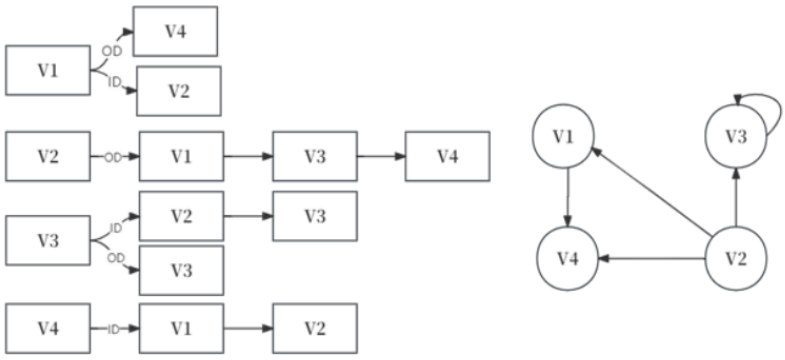


图 2.5.1 改进的邻接链表结构

元素结点结构体的定义如下：

```
struct v_elem {   //存储在连接
    int value;
    int length;
    vector<int>ID_value;//存储入度结点
    vector<int>OD_value;//存储出度结点
};
```

其中每个结点代表的是每个课程的信息。value 代表课程的课号，length 代表课程的学时，ID_value 代表课程的先修课程的课号，OD_value 代表后继课程的课号。

2.5.4 功能说明（函数、类）

本题是关键路径算法的经典案例。关键路径算法需要分别求出每个活动开始的最早以及最晚的时间，作差求出其中余量，余量为 0 的活动是决定整个项目时长的关键活动。

求解活动最早开始时间 ve 和最晚开始时间 vl 是借助前后活动的 ve 以及 vl 的递推关系实现的。

1.最早开始时间 ve

如图 2.5.2，要求 D 的最早开始时间 ve.D。

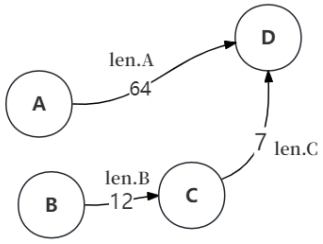


图 2.5.2 求D的ve

现在对该问题进行分析：D 课程可以开始进行的前提条件课程 A 和课程 C 都要修读完成，其中有任意一者未完成则 D 就不可以开始。因此 D 的最早开始时间取决于 A 和 C 两者修读结束的最晚时间。即 **ve.D=max{ve.A+len.A，ve.C+len.C}**。

根据以上过程分析可知，任意一个结点的最早开始时间都取决于其前驱结点，即其所有前驱结点中活动结束的最晚时间，也就是 **ve.this=max{ve.ID[i]+len.ID[i]}(i=0,1,2,3,……)**。

通过如上分析，得到了递推关系式。接下来只需要按照拓扑排序的方式，维护一个入度为 0 的结点所构成的集合，依次从前向后按照递推关系式求解最早开始时间 ve 即可。
实现代码如下：

```
//求结点最早完成时间 维护一个入度为0的集合
void Get_ve() {
    int MAX_ve = 0;
    int Count = 0;
    while(Count<vexnum) {
        for (int i = 1; i <= vexnum; i++) {
            if (ID[i] == 0) {  //入度为0，计算器其开始最早时间
                if (V[i - 1].ID_value.size() == 0) { //i没有前置连接
                    ve[i] = 0;
                }
                else {  //i有前置连接
                    MAX_ve = 0;
                    //检索前置连接的ve
                    for (int q = 0; q < V[i - 1].ID_value.size(); q++) {
                        ve[i] = ve[V[i - 1].ID_value[q]] + V[V[i - 1].ID_value[q] - 1].length;
                        MAX_ve = ve[i] > MAX_ve ? ve[i] : MAX_ve;
                    }
                    ve[i] = MAX_ve;
                }
                //该节点断开连接
                ID[i] = -1;
                //该结点的出度连接结点的入度减1
                for (int j = 0; j < V[i - 1].OD_value.size(); j++) {
                    ID[V[i - 1].OD_value[j]]--;
                }
                Count++;
                break;
            }
        }
    }
```

2.最晚开始时间 vl

如图 2.5.3，要求 A 的最晚开始时间 vl.A。

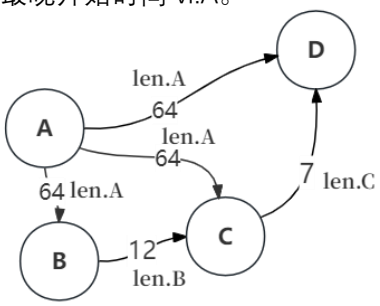


图 2.5.3 求A的vl

现在对该问题进行分析：C、B、D 三个课程想要开始必须先要完成 A，因此如果对于其中任意一个课程，假设 B 最晚开始时间为 vl.B，则能保证 B 按时完成的情况下，A 课程的最晚开始时间为 vl.B-len.A。在此逻辑下，A 必须保证 B、C、D 都可以按时完成，因此 A 课程的最晚开始时间，要保证 C、B、D 三者中最急迫完成的活动（即最晚开始时间最早的活动）也可以完成。故 **vl.A=min{vl.B-len.A，vl.C-len.A，vl.D-len.A }**。

根据以上过程分析可知，任意一个结点的最晚开始时间都取决于其后继结点，即其所有后继结点中活动开始的最早时间，也就是 **vl.this=min{vl.OD[i]-len.this}(i=0,1,2,3,……)**。

通过如上分析，得到了递推关系式。同理接下来只需要按照拓扑排序的方式，维护一个出度度为 0 的结点所构成的集合（实现过程正好与求解最早开始实践性实践性相反），依次从前向后按照递推关系式求解最早开始时间 vl 即可。
实现代码如下：

```
//求结点最晚完成时间 维护一个出度为0的集合
void Get_vl() {
    int Graduate = 0;//毕业的时间，是ve中的最大值
    int Max = 0;
    for (int p = 1; p <= vexnum; p++) {
        Max = Max > ve[p] ? Max : ve[p];
    }
    Graduate = Max;
    int MIN_vl = 2147483647;
    int Count = 0;
    while (Count < vexnum) {
        for (int i = 1; i <= vexnum; i++) {
            if (OD[i] == 0) {  //入度为0，计算器其开始最早时间
                if (V[i - 1].OD_value.size() == 0) { //i没有前置连接
                    vl[i] = Graduate;
                }
                else {  //i有前置连接
                    MIN_vl = 2147483647;
                    //检索后继连接的vl
                    for (int q = 0; q < V[i - 1].OD_value.size(); q++) {
                        vl[i] = vl[V[i - 1].OD_value[q]] - V[i-1].length;
                        MIN_vl = vl[i] < MIN_vl ? vl[i] : MIN_vl;
                    }
                    vl[i] = MIN_vl;
                }
                //该节点断开连接
                OD[i] = -1;
                //该结点的入度连接结点的入度减1
                for (int j = 0; j < V[i - 1].ID_value.size(); j++) {
                    OD[V[i - 1].ID_value[j]]--;
                }
                Count++;
                break;
            }
        }
    }
}
```

2.5.5 调试分析（遇到的问题和解决方法）

本题完成后提交到 OJ 平台，发现有部分测试点样例未通过，发现是因为对于“最晚毕业时间”这一条件的理解有错误。在一开始误把每一个尾结点的最晚开始时间当做了各自的最早结束时间。但是其实真正影响毕业时间的，应当是所有结束时间中的最大值。因此对于所有初始时候出度为 0 的结点，他们的最晚结束时间应当是他们之中的最大值。如下图：

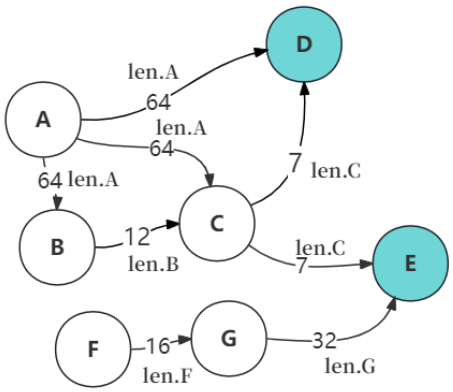


图 2.5.4 毕业时间是D、E中结束时间最晚的

图中的 D 和 E 都是结束课程，但毕业时间应当是两者中结束最晚的一者。因此需要先统一最晚结束时间，进而获得各尾结点的 vl。

2.5.6 总结和体会

本题是关键路径求解题目的经典例题，尤其是要关注这种借助递推表达式求解问题的方法。最早开始时间取决于前驱结点，最晚开始时间取决于后继结点。此外，大部分的关键路径题目时常只包含一个结束结点，但是本题背景下会有多个结束结点，要注意这一点的不同对总结束时间带来的影响。

4. 实验总结

本次实验主要涉及的是有向图、无向图的存储方式以及图的常见问题的算法。

在图存储结构方面，本次实验使用的是邻接矩阵和邻接链表两种方法，前者优点是存储结构简单、便于操作，但是占用空间大，遍历的时间效率低；后者的优点是占用空间小，可以直接访问无向图中与某一结点直接相连的所有结点及有向图中某一结点的所有后继结点信息，但是仍然不方便访问其前驱结点，同时构造起来也比较麻烦，需要建立多张链表。在第五题中采用了一种将前驱、后继链表分开存储的邻接链表结构（为了方便构建，直接采用了 STL 容器中的 vector），很好的解决了常规的邻接链表不方便访问前驱结点的问题。

此外，除了在本次实验中用到的邻接矩阵、邻接链表两种存储图的方式，在解决问题中还经常使用一种直接存储边的链式向前星的存储结构。这种结构不仅存储和读取效率高，同时也可以直接访问所有结点的前驱和后继，在代码实现中也比较方便。

在算法方面，主要进行了 BFS、DFS、图连通性判断、拓扑排序、关键路径、最小生成树(Kruskal、Prim)以及最短路径(Dijkstra)算法的实现。这些算法作为图相关问题中常用的朴素算法，在此基础上针对各类问题的变式还有对应的延伸算法。在本次实验中，很多题目是在经典模型上进行了微小的改动，从而使得算法也需要发生调整。如在题目三村村通中，需要额外考虑一开始就出现环的情况，但是其本质思想没有发生改变。同时，也需要关注各算法的外延条件，例如 Dijkstra 算法要求所有边的权重都要为正数。

纵观以上各类“最优化”类算法，其本质思想都是贪心算法，并根据对应的算法步骤再对应设计了一些存储中间量的表与数组。要关注动态规划的递推转移思想以及分治/递归思想的在算法中的应用。