作业 HW04 实验报告

姓名：闫浩扬 学号：2253156 日期：2023 年 12 月 5 日

1. 涉及数据结构和相关背景

本次作业涉及图的数据结构，图(Graph)是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：G(V,E)，其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

1.1 基本术语

顶点（Vertex）：图中的数据元素。线性表中我们把数据元素叫元素，树中将数据元素叫结点。

边：顶点之间的逻辑关系用边来表示，边集可以是空的。

无向边（Edge）：若顶点 V1 到 V2 之间的边没有方向，则称这条边为无向边。

无向图(Undirected graphs)：图中任意两个顶点之间的边都是无向边。

有向边：若从顶点 V1 到 V2 的边有方向，则称这条边为有向边，也称弧(Arc)。

有向图(Directed graphs)：图中任意两个顶点之间的边都是有向边。

简单图：图中不存在顶点到其自身的边，且同一条边不重复出现。

无向完全图：无向图中，任意两个顶点之间都存在边。

有向完全图：有向图中，任意两个顶点之间都存在方向互为相反的两条弧。

稀疏图：有很少条边。稠密图：有很多条边。

权（Weight）：与图的边或弧相关的数。网（Network）：带权的图。

子图（Subgraph）：假设 G=（V,{E}）和 G'=（V',{E'}），如果 V'包含于 V 且 E'包含于 E，则称 G'为 G 的子图。

度（Degree）：无向图中，与顶点 V 相关联的边的数目。有向图中，入度表示指向自己的边的数目，出度表示指向其他边的数目，该顶点的度等于入度与出度的和。

路径的长度：一条路径上边或弧的数量。

连通图：图中任意两个顶点都是连通的。

连通分量：无向图中的极大连通子图。（子图必须是连通的且含有极大顶点数）

强连通分量：有向图中的极大强连通子图。

生成树：无向图中连通且 n 个顶点 n-1 条边叫生成树。

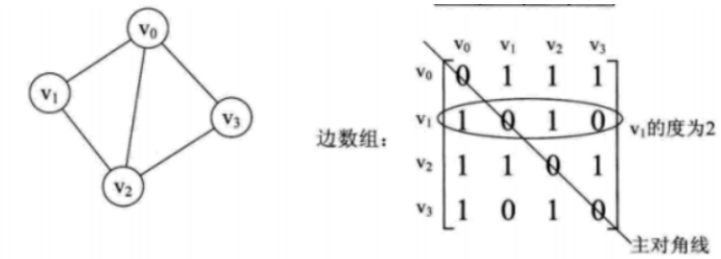
有向树：有向图中一顶点入度为 0 其余顶点入度为 1。

森林：一个有向图由若干棵有向树构成生成森林。

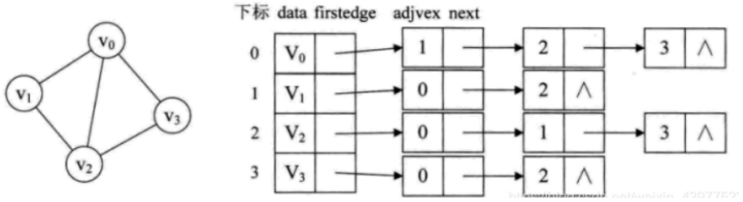
1.2 存储结构

图的存储结构可以采用邻接矩阵，邻接表，十字链表，邻接多重表的形式。

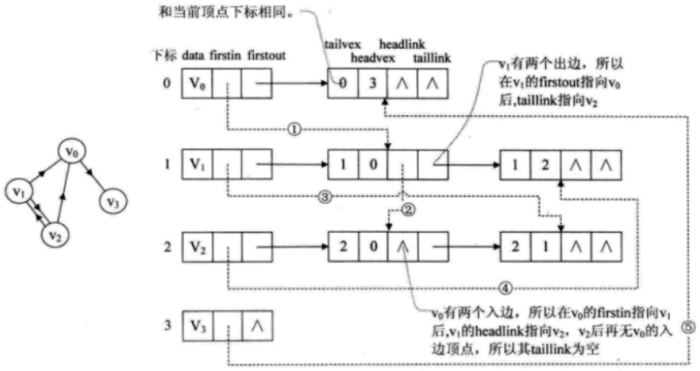
邻接矩阵：用两个数组，一个数组保存顶点集，一个数组保存边集。



邻接表：可以发现对于边数相对顶点较少的图，邻接矩阵会造成存储空间的浪费，可以采用邻接表表示稀疏图。



十字链表：对于有向图来说，邻接表是有缺陷的，只知道出度问题，想了解入度就必须要遍历整个图才能知道，反之，逆邻接表解决了入度却不了解出度的情况。为了解决这个问题，我们要引出有向图的一种存储方法：十字链表。实质上，十字链表就是将邻接表与逆邻接表结合起来。



邻接多重表：十字链表是对有向图的优化存储结构，对于无向图，如果我们关注的是顶点，那么邻接表是不错的选择，但如果我们更关注边的操作，比如对已经访问的边做标记，删除某一条边等操作。那就意味，需要找到这条边的两个边表结点进行操作，显然这是比较麻烦的。可以使用邻接多重表实验。

1.3 图的操作

广度优先遍历 BFS

深度优先遍历 DFS

最小生成树：把构造连通网的最小代价生成树称为最小生成树（Minimum Cost Spanning Tree），找连通网的最小生成树，经典的算法有两种，普里姆算法和克鲁斯卡尔算法。

拓扑排序：对一个**有向图**构造拓扑序列的过程。

最短路径/关键路径

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）。

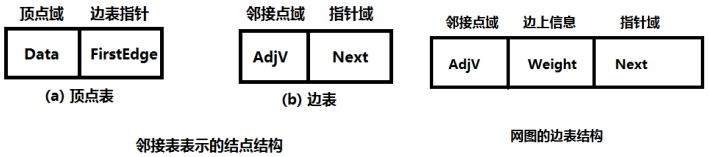
2.1.2 基本要求

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；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

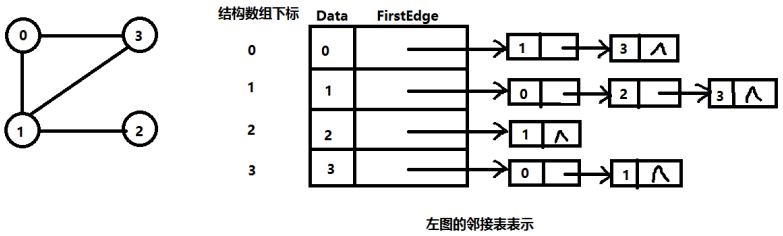
2.1.3 数据结构设计

无向图中，图中任意两个顶点之间都是无向边，由于很可能出现稀疏图，所以采用邻接表的方式实现无向图。



在本题中，边无权重，所以边表节点只包含邻接点域和指针域即可，为每一个节点创

建一个邻接表，用以表示和该节点相邻接的节点，如下图所示。



其中 Data 域用于存放节点信息，在这里为节点标号，FirstEdge 用于存放首个邻接点的地址。

图节点：图节点作为构造邻接表的基本组成，包含节点信息和指向下一节点的指针。

```
/*description：表示图的节点*/
struct Node {
    int vertex;  // 节点值
    Node* next;  // 指向下一个节点的指针
    Node(int v) : vertex(v), next(nullptr) {}  // 构造函数，初始化节点值为 v，next 为 nullptr
};
```

图：使用邻接表表示无向图，需要包含图的基本信息，并同步构造邻接表，要保证邻接表和无向图的更新是同步的。

```
// description：Graph，使用邻接表表示图
struct Graph {
    int numVertices;  // 图的顶点数
    Node** adjLists;  // 邻接表，存储每个顶点的邻接点链表头指针
    Graph(int vertices){
        numVertices = vertices;
        adjLists = new Node*[vertices];  // 创建邻接表数组
        for (int i = 0; i < vertices; i++)
            adjLists[i] = nullptr;  // 初始化每个顶点的邻接点链表为空
    }

    // 创建新节点
    Node* createNode(int v){
        Node* newNode = new Node(v);
        newNode->next = nullptr;
        return newNode;
    }

    // 添加边，无向图，双向添加
    void addEdge(int src, int dest){
        Node* newNode = createNode(dest);
        if (邻接表为空)
            adjLists[src] = newNode;  // 邻接点链表为空时，直接指向新节点
        else {
            Node* temp = adjLists[src];
            while (遍历至尾部)
                temp = temp->next;  // 尾插法
            temp->next = newNode;
        }
        newNode = createNode(src);
        if (adjLists[dest] == nullptr)

            adjLists[dest] = newNode;
        else {
            Node* temp = adjLists[dest];
            while (temp->next != nullptr)
                temp = temp->next;
            temp->next = newNode;
        }
    }
};
```

无向图的 DFS 和 BFS 遍历

所谓深度优先搜索，是从图中的一个顶点出发，每次遍历当前访问顶点的临界点，一直到访问的顶点没有未被访问过的临界点为止。然后采用依次回退的方式，查看来的路上每一个顶点是否有其它未被访问的临界点。访问完成后，判断图中的顶点是否已经全部遍历完成，如果没有，以未访问的顶点为起始点，重复上述过程。

可以见得深搜是一个不断回溯的过程，回溯我们可以使用递归的思想实现。

所谓广度优先搜索类似于树的层次遍历。从图中的某一顶点出发，遍历每一个顶点时，依次遍历其所有的邻接点，然后再从这些邻接点出发，同样依次访问它们的邻接点。按照此过程，直到图中所有被访问过的顶点的邻接点都被访问到。最后还需要做的操作就是查看图中是否存在尚未被访问的顶点，若有，则以该顶点为起始点，重复上述遍历的过程。

同样可以发现，广搜也涉及回溯操作，但是他需要回退到本层次第一个元素，更类似于队列的先排队后将队首元素出队的过程，所以决定使用队列模拟广搜。

队列

```
// description：Queue，表示队列
struct Queue{
    Node* front;  // 队头指针
    Node* rear;   // 队尾指针
    Queue() : front(nullptr), rear(nullptr) {}  // 构造函数
    // 入队操作
    void enqueue(int v){
        Node* newNode = new Node(v);  // 创建新节点
        if (rear == nullptr) {
            front = rear = newNode;  // 队列为空时，队头和队尾指向新节点
            return;
        }
        rear->next = newNode;  // 队尾指向新节点
        rear = newNode;        // 更新队尾指针
    }

    // 出队操作
    int dequeue(){
        if (队列为空) 返回 -1
        else{
            int v = front->vertex;  // 获取队头节点值
            Node* temp = front;
            front = front->next;  // 队头指向下一个节点
            if (front == nullptr)
                rear = nullptr;  // 更新队尾指针
            delete temp;  // 释放原队头节点内存
            return v;
        }
    }

    // 判断队列是否为空
    bool isEmpty()
        return front == nullptr;
};
```

2.1.4 功能说明（函数、类）

在构建好邻接表和队列后，就可以开始实现无向图的深度与广度遍历。

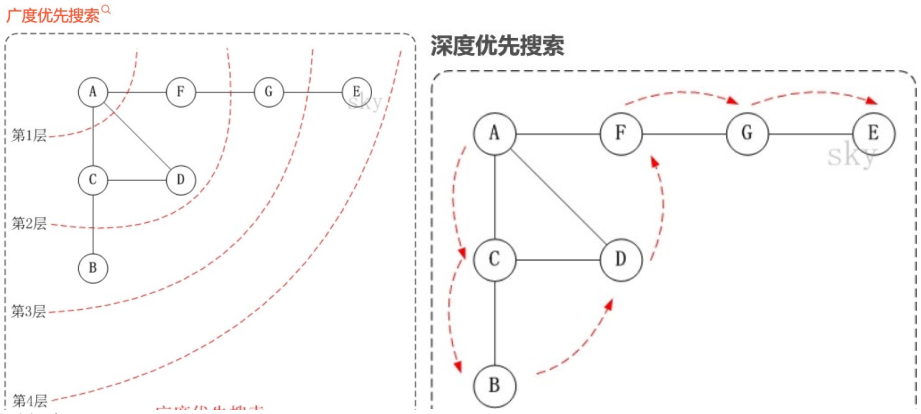
深度优先搜索 DFS

算法思想：从图中的每个未访问过的编号最小的顶点出发，通过递归深度遍历其邻接点，同时标记已访问过的顶点，以确保每个顶点都被访问且只被访问一次。在遍历的过程中，根据访问的先后顺序输出顶点值，同时通过 theFirst 变量控制输出格式，保证在第一个输出之前不输出空格。

```
/*description: 深度优先遍历
```

```
*param: vertex - 当前遍历的顶点;
*param:graph - 图的数据结构;
*param:visited - 用于标记顶点是否被访问的数组;
*param:theFirst - 控制输出格式的标志变量
*return: 无
*/
void dfs(int vertex, Graph& graph, bool* visited, bool& theFirst){
    visited[vertex] = true;  // 标记当前顶点为已访问
    if (!theFirst)  // 根据 theFirst 变量控制输出格式，确保在第一个输出之前不输出空格
        cout << " ";
    else
        theFirst = false;
    cout << vertex;  // 输出当前顶点的值
    Node* temp = graph.adjLists[vertex];  // 获取当前顶点的邻接点链表头指针
    while (temp){  // 遍历当前顶点的邻接点
        int neighbor = temp->vertex;  // 获取邻接点的值
        if (!visited[neighbor])  // 如果邻接点未被访问，则递归调用 dfs 函数
            dfs(neighbor, graph, visited, theFirst);
        temp = temp->next;  // 移动到下一个邻接点
    }
}
```

时间复杂度分析：对于每个顶点，DFS 算法访问其所有邻接点，因此总体时间复杂度取决于顶点数（V）和边数（E）。在最坏情况下，每条边都会被访问一次，因此时间复杂度为 O(V + E)。
空间复杂度分析：空间复杂度主要由递归调用和额外的数据结构（visited 数组）引起。递归调用的最大深度为图的最大连通分量的大小，因此最坏情况下为 O(V)。额外的数据结构 visited 数组和递归调用的系统栈空间也会占用一定空间，因此总体空间复杂度为 O(V)。
此外还可以使用栈的方式实现 DFS 算法，这样可以节省系统栈的调用。



广度优先搜索 BFS

BFS（广度优先搜索）算法用于从给定起始顶点开始，按照层次遍历图的顶点，并输出遍历的顶点序列。通过队列存储待访问的顶点，确保每个顶点按层次顺序访问，同时根据 theFirst 变量控制输出格式，保证在第一个输出之前不输出空格。

```
/*description: 广度优先遍历算法
*param: start - BFS 的起始顶点;
*graph - 图的数据结构;
*visited - 用于标记顶点是否被访问的数组;
*bfsQueue - 用于存储待访问顶点的队列;
*theFirst - 控制输出格式的标志变量
*return: 无
*/
void bfs(int start, Graph& graph, bool* visited, Queue& bfsQueue, bool& theFirst){
    bfsQueue.enqueue(start);  // 将起始顶点入队
    visited[start] = true;    // 标记起始顶点为已访问
```

```
    while (!bfsQueue.isEmpty()){  // 队列非空时循环
        int current = bfsQueue.dequeue();  // 出队一个顶点
        if (!theFirst) // 根据 theFirst 变量控制输出格式，确保在第一个输出之前不输出空格
            cout << " ";
        else
            theFirst = false;
        cout << current;  // 输出当前顶点的值
        Node* temp = graph.adjLists[current];  // 获取当前顶点的邻接点链表头指针
        while (temp){  // 遍历当前顶点的邻接点
            int neighbor = temp->vertex;  // 获取邻接点的值
            if (!visited[neighbor]){  // 如果邻接点未被访问，则标记为已访问并入队
                visited[neighbor] = true;
                bfsQueue.enqueue(neighbor);
            }
            temp = temp->next;  // 移动到下一个邻接点
        }
    }
}
```

时间复杂度分析：BFS 算法使用队列实现，在最坏情况下需要访问图中的所有顶点和边，每个节点都涉及一次出队和入队操作，因此时间复杂度为 O(V + E)，其中 V 为顶点数，E 为边数。
空间复杂度分析：空间复杂度主要由队列和额外的数据结构（visited 数组）引起。队列的最大长度为图的最宽层次，即最大连通分量的大小，因此空间复杂度为 O(V)。额外的数据结构 visited 数组也占用 O(V) 的空间。

2.1.5 调试分析（遇到的问题和解决方法）

1. DFS 无法回溯

在实现中，每次递归调用 dfs 都会标记当前顶点为已访问，然后在退出递归时再将其重新标记为未访问，从而实现回溯。这才能确保了每个顶点在搜索过程中只会被访问一次，同时允许算法在发现不符合条件的情况下回退到之前的选择点。

2. 邻接表创建

无向图顶点之间都是无向边，所以 addEdge 需要双向操作。在 addEdge 函数中，通过在链表尾部添加新节点的方式创建边。确保在每次添加新节点时，将新节点的 next 指针置为 nullptr，以避免形成循环链表。

3. 输出格式

添加一个标志变量用于判断是否为第一个元素，需不需要加空格。

2.1.6 总结和体会

通过这道题，我熟悉了无向图的邻接表表示方法，同时也使用了 DFS 和 BFS 两种方法遍历无向图。在这道题中，稀疏图较多，并且只涉及简单的遍历操作，所以采取邻接链表更优，可以节省空间，插入删除操作也较为简单。如果为稠密图的话，邻接矩阵可能更为方便，邻接矩阵有利于快速查询，如果要是遍历的话，时间复杂度为节点数平方，与度数无关。

2.2 小世界现象/六度空间

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。”

3.2.2 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%”。

3.2.3 数据结构设计

这道题换句话说就是要求每个节点同其他节点相隔的距离是不是超过 6。一种方法，可以遍历求出所有点两两相距的距离；另一种方法可以使用层数的思想，类似于二叉树的分层遍历，再 BFS 遍历的过程中统计层数。

算法思想如下：

1. 对每个节点，进行广度优先搜索；
2. 搜索过程中累计访问的结点数；
3. 需要记录“层”数，仅计算 6 层以内的结点数。

所以决定使用邻接表表示无向图，使用自定义队列辅助 BFS 算法。

图节点：

```
/* 图节点 */
struct VNode {
    VNode* Next;
    int V;
};
```

邻接表节点：

```
/* 邻接表节点 */
struct LNode {
    VNode* FirstEdge;
};
```

图：

```
/* 图 */
struct GNode {
    int Nv, Ne;
    LNode Head[MAXN];
};
```

队列：

```
// 队列
struct MyQueue {
    int arr[MAXN];
    int front, rear;
    MyQueue() : front(0), rear(0) {}
    void push(int value)
        arr[rear++] = value;
    int pop()
        return arr[front++];
    bool empty()
        return front == rear;
};
```

2.2.4 功能说明（函数、类）

邻接表的构建

```
/* 插入边到邻接表中，构建无向图的邻接关系
 * param: v - 顶点 v;  param: w - 顶点 w
 * return: 无
 */
void Insert(int v, int w) {
    // 创建新节点
    VNode* newNode;
    // 在 w 的邻接表头插入 v
    newNode = new VNode;
    newNode->V = v;
    newNode->Next = G->Head[w].FirstEdge;
    G->Head[w].FirstEdge = newNode;
    // 在 v 的邻接表头插入 w
    newNode = new VNode;
    newNode->V = w;
    newNode->Next = G->Head[v].FirstEdge;
    G->Head[v].FirstEdge = newNode;
}
```

这段代码使用头插法来构建邻接表，这样的好处是新的邻接点直接成为链表的头节点，不需要遍历到尾部，所以插入的时间复杂度为 O(1)。
空间复杂度：图中每个节点都要创建一个邻接表，所以时间复杂度为 O(N).

BFS 大体思想同第一题，但是新增一个 level 计数器用于记录层数

```
/*
 * 使用广度优先搜索（BFS）算法计算每个顶点到其他顶点的最短路径层数，并输出路径的占比
 * param: s - 起始顶点
 * return: 无
 */

/*
v:当前处理的节点
tail:记录每一层最后一个入队的顶点
last:记录上一次出队的顶点，用于判断是否应该进入下一层
cnt:用于统计经过的顶点数，最后也就是在 6 层之内的节点数量
level:用于记录当前层数
*/
void BFS(int s) {
    MyQueue Q;
    int v, tail, last = s, cnt = 0, level = 0;
    Q.push(s);            // 将起始顶点入队
    visited[s] = 1;        // 标记起始顶点已访问
    cnt++;
    while (!Q.empty()) {
        v = Q.pop();       // 出队一个顶点
        for (VNode* p = G->Head[v].FirstEdge; p; p = p->Next) {
            if (!visited[p->V]) {
                Q.push(p->V);     // 将未访问的邻接点入队
                visited[p->V] = 1; // 标记为已访问
                cnt++;
                tail = p->V;       // 记录最后一个入队的顶点
            }
        }
        if (v == last) {
            level++;
            last = tail;      // 更新最后一个顶点
        }
        if (level == 6) break; // 如果层数达到 6，停止 BFS，接下来计算占比
    }
    // 计算路径的占比并输出结果
    double perc = (static_cast<double>(cnt) / static_cast<double>(G->Nv)) * 100;
    cout << s << ": " << fixed << setprecision(2) << perc << "%" << endl;
}
```

功能描述：使用广度优先搜索算法从起始顶点开始遍历图，计算每个顶点到其他顶点的最短路径的层数（深度）。
时间复杂度分析：队列操作：O(E)，其中 E 为边数，每个边都会入队一次。
邻接表遍历所有顶点和边：O(V+E)
整体的时间复杂度为 O(V+E)
空间复杂度分析：
visited 数组： O(V)，存储每个顶点的访问状态。
MyQueue 队列： O(V)，队列的最大长度为顶点数。
其它变量： O(1)，常数个额外变量。
总体的空间复杂度为 O(V)。其中，V 为图的顶点数。

3.2.4 调试分析（遇到的问题和解决方法）

1. 使用 DFS 出错

这道题是一个图中的限制搜索问题，最好的方法是使用 BFS，使用 DFS 会产生问题，因为 DFS 在搜索过程中可能会遗漏某些路径或者在搜索的过程中产生不必要的剪枝，导致没有找到所有可能的长度为 6 的路径。这与 DFS 的特性有关，DFS 在搜索过程中会一直深入到底部，而在回溯时可能会丢失一些路径。相比之下，BFS 是一种逐层扩展搜索的方法，它可以更全面地探索所有可能的路径，确保找到所有长度为 6 的路径。在这道题目中，每一步 BFS 都会向外扩展一层，从而确保覆盖了所有的路径。

2. 混乱错误

visited[MaxN]申请成全局数组，所以在循环结束后不会被释放，它仍保留上一次的值，所以会影响到下一次遍历，使用 memset(visited, 0, sizeof(visited)); 将 visited 重置避免混乱。

3. 及时更新

确保在队列中的每一层结束时正确地更新 level 和 last。在这里，level 表示当前 BFS 遍历的层级，而 last 记录了上一层的最后一个节点。

2.2.6 总结和体会

这道题让我更加熟悉了 BFS 的相关操作，BFS 类似于二叉树的层次遍历，这和这道题很相似。同时，在这道题中 DFS 会出现剪枝错误，看来不同的便利方法有其不同的优势。

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

输入：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。

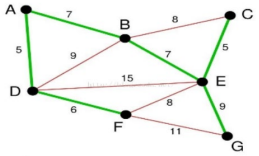
输出：输出一行，仅有一个整数，表示为使所有的村庄连通，要新建公路的长度的最小值。

2.3.3 数据结构设计

这道题实际上考察了无向图的最小生成树问题，要求找到没有连接的路的最短长度。为了实现这一目标，有两种算法。

一种是 prim 算法：可以先计算出整个连通图的最短路径长度，然后减去已经存在的路径长度。因此，可以将已经存在的路的权值设为 0，因为已经连好的路的权值为 0，这样一定会经过连接好的路。这样一来，只需记录整个图的最短路径，即为需要铺设的新路的长度。

另一种是 Kruskal 算法：首先，将已经修建好的路的权值置为 0，这样求和时就不用重复计算，确保了这些路的优先选择。然后，将所有的道路按照权值从小到大排序，逐一考察每条边。如果一条边的两个端点不属于同一集合，就将它们合并，直到所有的点都属于同一个集合为止。这里使用了并查集，用于查询元素 a 和元素 b 是否为同一组，以及将元素 a 和 b 合并为同一组。



整体算法流程如下：

1. 输入图 G。

2. 将图 G 看作一个森林，每个顶点为一棵独立的树。

3. 将所有的边加入集合 S，即一开始 S = E。

4. 从 S 中拿出一条最短的边(u, v)，如果(u, v)不在同一棵树内，则连接 u, v 并合并这两棵树，同时将(u, v)加入生成树的边集 E'。

5. 重复步骤 4，直到所有点属于同一棵树，边集 E'即为一棵最小生成树。

这样就保证了生成的树是最小生成树，即具有最小的权值和。

搜索发现：prim 算法适用于稠密图 O(N^2)，kruskal 适合稀疏图 O(eloge).这道题应该算作稀疏图，选用 Kruskal 算法效率较高些。

所以总体思路：使用 Kruskal 算法来求解最小生成树（MST）的问题。通过对边的权值进行排序，并使用并查集（Union-Find）来维护节点的连接关系，逐步选择最小权值的边，生成最小生成树。

涉及的数据结构：

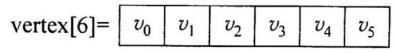
边：无向有权边，要存储边的左右端点和权值。

```
// 数据结构：边的结构体
struct edge {
    int l, r, w;  // 边的左端点、右端点、权值
};
```

图：使用边集数组来表示图，这主要是为了配合 Kruskal 算法的实现。

Kruskal 算法是一种基于边的贪心算法，它按照边的权值从小到大的顺序选取边，构建最小生成树。在这个算法中，我们需要对边进行排序，并且每次选择边时，需要判断这条边的两个端点是否在同一个连通分量中。使用边集数组的数据结构可以方便地进行排序，并在并查集中维护顶点的连通关系。如果使用邻接表表示图，虽然也能实现 Kruskal 算法，但在找到最小边的过程中需要遍历所有边，而不是直接从排序好的边集合中选择最小的边。这样会导致算法的效率降低，因为在邻接表中找到最小边的过程可能需要对图进行多次遍历。因此，为了简化 Kruskal 算法的实现，方便对边进行排序和处理，使用边集数组是一种比较合适的选择。

```
const int MAX_N = 105;
vector<edge> edges;
int parent[MAX_N];
```



下标	0	1	2	3	4	5	6	7	8
from	1	2	0	2	3	4	0	3	0
to	4	3	5	5	5	5	1	4	2
weight	12	17	19	25	25	26	34	38	46

2.3.4 功能说明（函数、类）

算法的实现涉及并查集和 kruskal

```
//作为 sort 的自定义比较函数，比较权值
bool compare(edge x, edge y) {
    return x.w < y.w;
}
```

```
// 函数名：find
// 描述：查找节点 x 所在的连通分量的根节点，并进行路径压缩
// 参数：x - 待查找的节点
// 返回值：x 所在连通分量的根节点
int find(int x) {
    // 如果 x 的父节点就是自己，说明 x 就是根节点
```

```
    if (parent[x] == x)
        return x;
    else
        // 否则，递归查找 x 的父节点，并将 x 的父节点更新为根节点
        return parent[x] = find(parent[x]);
}
```

```
// 函数名：Union
// 描述：将两个节点所在的连通分量合并
// 参数：x - 第一个节点，y - 第二个节点
// 返回值：无
void Union(int x, int y) {
    // 获取 x 和 y 所在连通分量的根节点
    int rootX = find(x);
    int rootY = find(y);

    // 如果 x 和 y 不属于同一连通分量，将其中一个的根节点指向另一个根节点
    if (rootX != rootY)
        parent[rootX] = rootY;
}
```

Union 函数实现了并查集中的合并操作，将两个节点所在的连通分量合并为一个连通分量。通过 find 函数查找两个节点的根节点，将其中一个根节点的父节点指向另一个根节点，从而实现合并。

时间复杂度分析：

find 函数的时间复杂度主要取决于树的高度，最坏情况下为 O(logN)。

Union 函数的时间复杂度同样取决于树的高度，最坏情况下为 O(logN)。

空间复杂度分析：

parent 数组占用 O(N) 空间。

edges 数组占用 O(E) 空间。

整体空间复杂度为 O(N + E)，其中 E 为边的数量，N 为节点的数量。

Kruskal 算法

```
// 函数名：kruskal
// 描述：使用 Kruskal 算法求解最小生成树的权值之和
// 参数：无
// 返回值：最小生成树的权值之和
int kruskal() {
    int ans = 0;
    // 将边按权值从小到大排序
    sort(edges.begin(), edges.end(), compare);
    // 遍历所有边
    for (const auto& e : edges) {
        int t1 = find(e.l);  // 获取起点所在连通分量的根节点
        int t2 = find(e.r);  // 获取终点所在连通分量的根节点
        // 如果起点和终点不属于同一连通分量，将它们合并，并将边的权值累加到最小生成树的权值之和
        if (t1 != t2) {
            ans += e.w;
            Union(t1, t2);
        }
    }
    return ans;
}
```

kruskal 函数中排序操作的时间复杂度为 O(E logE)，遍历所有边的时间复杂度为 O(E)，总体时间复杂度为 O(E logE)。整体时间复杂度为 O(E logE + E logN)，其中 E 为边的数量，N 为节点的数量。

整体空间复杂度为 O(N + E)，其中 E 为边的数量，N 为节点的数量。

2.3.5 调试分析（遇到的问题和解决方法）

1. Time Limit Exceeded & RunTime Error

最开始样例 9 超时，于是使用更高效的 vector 表示边集数组，队列用于排序，但是样例 9 出现了 RunTime Error 的错误。删掉 queue 就可以了，好像是因为 OJ 不能用 queue。所以应该是因为之前的数据结构不如 vector 高效。或者说有些特殊情况没有考虑到，陷入了死循环里。

2. 添加并查集和路径压缩

为了优化效率，使用并查集个路径压缩，加速后续查找，同时，还要对边进行排序，保证每次选择的边都是权值最小的。

3. 细节

在每次处理一个新的测试用例之前，清空 edges 向量，以确保不会保留上一个测试用例的边；输入的图是一个邻接矩阵，对于每一对节点 (i, j)，i < j 保证了只添加了一条边。这样可以避免重复添加边。

2.3.6 总结和体会

这道题我学习了一种新的存储结构表示无向图，边集数组，边集数组在本题中相比邻接表效率更高，搭配 Kruskal 算法和并查集，对于最小生成树问题的求解十分高效。

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

这道题矩阵元素的排序相互限制，本质是一个拓扑排序问题。

n 个元素的拓扑排序问题：

题中所给的约束关系，比如 a 在 b 前面，与图论中 a 存在一条出边连接到 b 一致，通过构建邻接表和入度，可以得到符合条件的拓扑排序。显然，若答案不存在，则会产生环，那么拓扑排序所得到的数组个数将小于 n。

在 k*k 矩阵中放入 k 个元素首先可以保证每行每列都只有一个元素，其余为 0。而行条件得到的拓扑排序与列条件得到的拓扑排序是可以独立的，可以同时满足。通过行拓扑得到的数组，每个元素在矩阵中的的行位置就是它在该数组中的下标位置，同样地，该元素的列位置就是它在列拓扑排序数组的下标位置。

所以，这道题的思路：

首先，注意到题目中的行和列是互补影响的，那么可以将行和列分开来考虑。
在这里不妨先考虑行。题目给出了一系列的上下关系(above, below)，要求 above 所在的行在 below 所在的行的的上面。对于这种有多重依赖关系，要求给出某种符合一来关系的序列的问题，通常的做法就是拓扑排序。在这里可以将每个数视为图中的一个节点，而对于一组上下关系(above, below)，则可以视为从 above 想 below 发出一条有向边。接下来在这个图中进行拓扑排序，得到的拓扑序对应的就是最终答案中每个数所在的行了。这样做可

行的原因是对于任意一组上下关系，above 的拓扑序必然是小于 below 的拓扑序的，因此 above 所在的行就必然在 below 所在行的上方，符合题目要求。

对于列也可以这样，将每一组左右关系(left, right)视为从 left 向 right 发出的有向边，然后同样是执行拓扑排序确定每个数对应的列。

需要注意的是，如果行或者列进行拓扑排序后，bfs 的队列长度不足 k，说明对应的有向图中出现了环，也就是说无法构造出符合题目要求的矩阵，此时直接返回空矩阵即可。

```
vector<vector<int>>:
          rowConditions: 二维向量，表示行的条件。每一行包含两个整数，表示该行的范围。
          colConditions: 二维向量，表示列的条件。每一行包含两个整数，表示该列的范围。
          g: 二维向量，表示图的邻接表形式，记录了图中每个节点的出边。
          matrix: 二维向量，表示最终构建的矩阵。
vector<int>:
          in_deg: 一维向量，表示每个节点的入度，用于拓扑排序。
          order: 一维向量，表示拓扑排序的结果，记录了节点的处理顺序。
          pos: 一维向量，记录了列的拓扑排序的位置，用于确定矩阵中元素的列位置。
queue<int>:
          q: 队列，用于进行拓扑排序时的广度优先搜索
```

2.4.4 功能说明（函数、类）

```cpp
// 拓扑排序函数
// 参数:
//   k: 节点数
//   edges: 图的边集，每个元素为一个边的起点和终点
// 返回值:
//   拓扑排序结果，即节点的处理顺序

vector<int> topo_sort(int k, vector<vector<int>>& edges) {
    // 邻接表表示图
    vector<vector<int>> g(k);
    // 记录每个节点的入度
    vector<int> in_deg(k);

    // 构建邻接表和入度数组
    for (auto& e : edges) {
        int x = e[0] - 1, y = e[1] - 1;
        g[x].push_back(y);
        ++in_deg[y];
    }
    // 拓扑排序结果
    vector<int> order;
    // 队列用于广度优先搜索
    queue<int> q;

    // 将入度为 0 的节点入队
    for (int i = 0; i < k; ++i) {
        if (in_deg[i] == 0)
            q.push(i);
    }

    // 广度优先搜索，生成拓扑排序
    while (!q.empty()) {
        int x = q.front();
        q.pop();
        order.push_back(x);
        // 更新与当前节点相邻节点的入度，并将入度为 0 的节点入队
        for (int y : g[x]) {
            --in_deg[y];
            if (in_deg[y] == 0)
                q.push(y);
        }
    }
    return order;
}
```

topo_sort 函数：通过广度优先搜索实现拓扑排序。参数 k 为节点数，edges 为图的边集，返回值为拓扑排序结果。

时间复杂度为 O(k + |edges|)，其中 k 为节点数，|edges| 为边数。

```
// 构建矩阵函数
// 参数：
//   k: 矩阵的维度
//   rowConditions: 行的条件，每个元素为一个行的范围
//   colConditions: 列的条件，每个元素为一个列的范围
// 返回值：
//   构建好的矩阵，如果构建失败返回空向量
vector<vector<int>> buildMatrix(int k, vector<vector<int>>& rowConditions,
vector<vector<int>>& colConditions) {
    // 获取行的拓扑排序
    auto rowOrder = topo_sort(k, rowConditions);
    // 获取列的拓扑排序
    auto colOrder = topo_sort(k, colConditions);

    // 如果排序结果不足 k 个元素，则构建失败
    if (rowOrder.size() < k || colOrder.size() < k)
        return {};

    // 记录列的拓扑排序的位置
    vector<int> pos(k);

    // 根据列的拓扑排序确定元素的列位置
    for (int i = 0; i < k; ++i) {
        pos[colOrder[i]] = i;
    }
    // 构建矩阵
    vector<vector<int>> matrix(k, vector<int>(k));

    for (int i = 0; i < k; ++i)
        matrix[i][pos[rowOrder[i]]] = rowOrder[i] + 1;
    return matrix;
}
```

buildMatrix 函数：利用 topo_sort 函数获取行和列的拓扑排序。如果排序结果不足 k 个元素，则构建失败，返回空向量。构建矩阵，根据列的拓扑排序确定元素的列位置。

返回构建好的矩阵，如果构建失败返回空向量。

buildMatrix 函数的时间复杂度与 topo_sort 函数的时间复杂度相同，为 O(k + |rowConditions| + |colConditions|)。

总体时间复杂度：设 n 为结点数，e 为边数.刚开始搜索 0 入度的结点使其入队时，复杂度为 O(n)；每次弹出队头，更新其邻接点的入度(-- g[t])时，共执行了 e 次。因此时间复杂度为 O(n + e)

2.4.5 调试分析（遇到的问题和解决方法）

刚开始没想到是拓扑排序，使用暴力破解，超时。拓扑排序还要考虑存不存在环，之前没考虑，出现错误。

2.4.6 总结和体会

这道题学习了拓扑排序的算法和它的适用问题。

2.5 小马吃草

2.5.1 问题描述

假设无向图 G 上有 N 个点和 M 条边，点编号为 1 到 N，第 i 条边长度为 w_i，其中 H 个

点上有可以食用的牧草。另外有 R 匹小马，第 j 匹小马位于点 start_j，需要先前往任意一个有牧草的点进食牧草，然后前往点 end_j，请你计算每一匹小马需要走过的最短距离。

2.5.2 基本要求

输入描述：第一行两个整数 N、M，分别表示点和边的数量，接下来 M 行，第 i 行包含三个整数 x_i,y_i,w_i，表示从点 x_i 到点 y_i 有一条长度为 w_i 的边，保证 x_i≠y_i，接下来一行有两个整数 H 和 R，分别表示有牧草的点数量和小马的数量，接下来一行包含 H 个整数，为 H 个有牧草的点编号，接下来 R 行，第 j 行包含两个整数 start_j 和 end_j，表示第 j 匹小马起始位置和终点位置，题目保证两个点之间一定是连通的，并且至少有一个点上有牧草

输出描述：输出共 R 行，表示 R 匹小马需要走过的最短距离

2.5.3 数据结构设计

对于这个图论问题，要在无向图中找到从起点到终点的最短路径，但是在到达终点之前，需要先访问至少一个特定的点（有牧草的点）。这个问题可以通过 Dijkstra 算法来解决。

首先，对于图的表示，选择使用邻接表来存储图的信息。邻接表是一种灵活的数据结构，它对稀疏图的表示非常高效。对于每个节点，创建一个列表，其中包含所有与该节点相邻的节点以及对应的边的权重。这可以通过一个哈希表实现，其中键是节点的编号，值是与该节点相邻的节点列表及对应的边权重。

在算法选择方面，采用 Dijkstra 算法来计算最短路径。Dijkstra 算法通过不断更新起始节点到其他节点的最短距离来找到最短路径。对于每匹小马，按照如下步骤进行计算：

1. 为了方便表示图，首先创建一个邻接表，其中包含所有节点及其相邻节点和对应的边权重。
2. 对于每匹小马，执行 Dijkstra 算法计算从起始位置到所有其他节点的最短距离。这可以通过优先队列（最小堆）来实现，以提高算法效率。
3. 找到距离最短的有牧草的节点。遍历有牧草的节点列表，选择距离起始位置最短的节点作为中间节点。
4. 使用 Dijkstra 算法计算从这个有牧草的节点到所有其他节点的最短距离，同样利用优先队列来提高效率。
5. 将从起始位置到有牧草的节点的距离和从有牧草的节点到终点的距离相加，得到的结果即为小马需要走过的最短距离。

```
vector<pair<int, int>> G[MAXN];: 邻接表，用于表示图。每个节点都有一个与之相连的节点列表，每个
                                 连接都有一个对应的权重（边的长度）。
vector<int> grass;             : 保存有牧草的点的编号。
vector<pair<int, int>> horse;  : 保存每匹小马的起始位置和终点位置。
vector<vector<int>> dist;      : 保存预先计算的每个有牧草的点到所有其他点的最短路径信息。
```

2.5.4 功能说明（函数、类）

```
// Dijkstra 算法实现单源最短路径
//   参数：
//   s: 源节点编号
//   返回值：
//   包含从源节点到所有其他节点的最短路径长度的数组
vector<int> dijkstra(int s) {
    // 初始化距离数组，INF 表示无穷大
    vector<int> dist(N + 1, INF);
    // 标记数组，记录节点是否已被访问
    vector<bool> vis(N + 1, false);
    // 优先队列存储距离和节点的对，使用最小堆实现
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    // 将源节点的距离设为 0，并加入优先队列
    dist[s] = 0;
    pq.push({0, s});
```

```
    // Dijkstra 算法主循环
    while (!pq.empty()) {
        // 取出当前距离最小的节点
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        // 如果节点已被访问，跳过
        if (vis[u]) continue;
        vis[u] = true;

        // 遍历当前节点的所有邻居节点
        for (auto p : G[u]) {
            int v = p.first; // 邻居节点编号
            int w = p.second; // 边权重

            // 如果通过当前节点到达邻居节点的路径更短，更新距离并加入优先队列
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    // 返回包含最短路径长度的数组
    return dist;
}
```

Dijkstra 算法用于计算单源最短路径。参数 s 为源节点编号。返回一个数组，包含从源节点到所有其他节点的最短路径长度。
dist: 存储从源节点到各节点的最短路径长度。
vis: 标记数组，记录节点是否已被访问。
pq: 优先队列，存储距离和节点的对，使用最小堆实现。
while 循环：从优先队列中取出当前距离最小的节点，更新与其邻居节点的距离，并将邻居节点加入优先队列。
return dist: 返回包含最短路径长度的数组。
时间复杂度：使用优先队列实现的 Dijkstra 算法时间复杂度为 O((V + E) * log(V))，其中 V 为节点数，E 为边数。

```
int main() {
    cin >> N >> M; // 输入点数和边数
    // 构建图的邻接表表示
    for (int i = 0; i < M; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        G[x].push_back({y, w});
        G[y].push_back({x, w});
    }
    cin >> H >> R;
    // 输入有牧草的点
    for (int i = 0; i < H; i++) {
        int g;
        cin >> g;
        grass.push_back(g);
    }
    // 输入马的起点和终点
    for (int i = 0; i < R; i++) {
        int s, e;
        cin >> s >> e;
        horse.push_back({s, e});
    }
```

```
    // 预先计算每个有牧草的点到所有其他点的最短路径
    for (int i : grass)
        dist.push_back(dijkstra(i));
    // 计算每匹马从起点到终点的最短路径
    for (int i = 0; i < R; i++) {
        int s = horse[i].first;
        int e = horse[i].second;
        int min_dist = INF;

        // 在所有有牧草的点中找到最短路径
        for (const vector<int>& d : dist) {
            min_dist = min(min_dist, d[s] + d[e]);
        }
        // 输出结果
        cout << min_dist << endl;
    }
    return 0;
}
```

功能描述：
从输入中读取图的点数 N、边数 M、每个点之间的边的权重。
读取有牧草的点数 H，以及每个有牧草的点的编号。
读取马的数量 R，以及每匹马的起点和终点。

读取图的点数、边数和边的权重，时间复杂度为 O(M)。
读取有牧草的点和马的数量，时间复杂度为 O(H + R)。
计算每个有牧草的点到其他所有点的最短路径，时间复杂度为 O(H * (V + E) * log(V))。
计算每匹马的最短路径，时间复杂度为 O(R * H)。
总体时间复杂度为 O((H + R) * (V + E) * log(V))。

整体的时间复杂度： O(H·(V + E)·log(V))，H 是有牧草的点的数量，V 是图的顶点数，D 是图的边数

2.5.5 调试分析（遇到的问题和解决方法）

1. 极少数路径计算出错

先前的代码是对每一匹小马，先用 Dijkstra 算法找到最近的有牧草的点 k，然后再用 Dijkstra 算法从 k 出发找到终点 e 的最短路径，这样就得到了从起点 s 到终点 e 的最短路径。但是这样做有一个问题，就是没有考虑到有可能存在多个有牧草的点，而且这些点之间也有边相连。这样就可能出现这样的情况：从 s 到 k 的最短路径是 d1，从 k 到 e 的最短路径是 d2，但是从 s 到 e 的最短路径不一定是 d1+d2，因为可能存在另一个有牧草的点 l，使得从 s 到 l 再到 e 的路径更短。

2. 节点未及时更新

检查优先队列的更新条件，在 Dijkstra 算法中，通过优先队列（最小堆）来选择当前距离最短的节点进行扩展。确保在将节点放入队列时，检查其距离是否更短，只有更短的距离才更新。为了解决这个问题，先预先计算每个有牧草的点到所有其他点的最短路径，然后对每一匹小马，找到所有有牧草的点中，使得从起点 s 到该点再到终点 e 的路径最短的那个点。这样就可以保证找到正确的答案。

2.5.6 总结和体会

这个题我使用了合适的数据结构，比如邻接表来表示图，优先队列来实现 Dijkstra 算法中的最小堆。同时也将一些函数模块化方便后续修改。

3. 实验总结

这次作业主要涉及图的操作，在本次作业中，我学习了无向图的不同表示方式，邻接表，邻接矩阵和编标数组，也熟悉了图的深度遍历和广度遍历，以及拓扑排序，最小生成树，最短路径的相关知识，收获很大，对数据结构和算法的选取理解更深。