作业 HW4* 实验报告

姓名：张三 学号：2351114 日期：2024 年 12 月 2 日

实验报告格式要求按照模板（使用 Markdown 等也请保证报告内包含模板中的要素）
对字体大小、缩进、颜色等不做强制要求（但尽量代码部分和文字内容有一定区分，可参考 vscode 配色）
报告内不要大段贴代码，尽量控制在 20 页以内

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

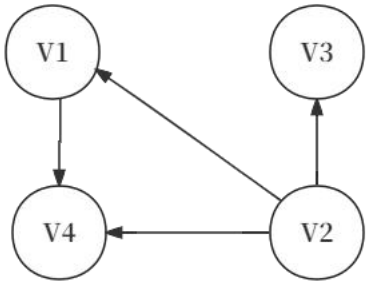


图 1.1 邻接矩阵

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：

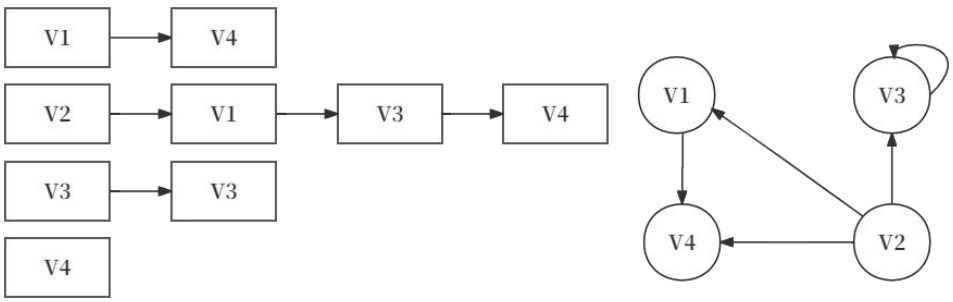


图 1.2 邻接链表

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：

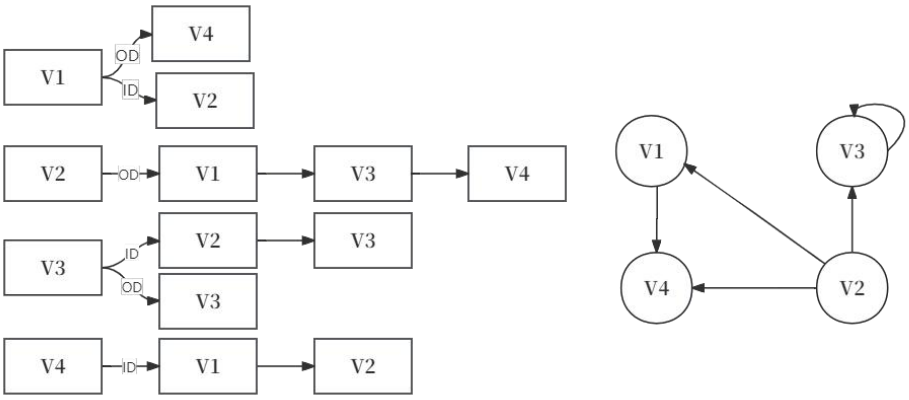


图 1.3 邻接链表-改进

2. 实验内容

2.1 题目一 图的遍历

2.1.1 问题描述

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

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<=1000;
对于所有数据，0.5n<=m<=1.5n，保证输入数据无错。但可能会有重边，不需特别处理。
下载 p107_data.cpp，编译运行以生成随机测试数据

2.1.3 数据结构设计

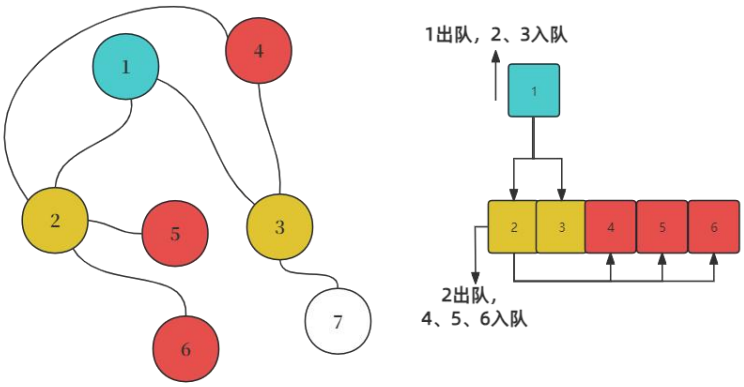
本题是一个无向图连接问题，选择使用邻接矩阵表示无向图需要对存储方式进行规定。由于无向图可以看做对称双向的有向图，因此无向图的邻接矩阵是一个对称矩阵，因此只需要存储半边三角即可。这里选择存储下半三角，即所有边都默认起点是大序号结点，终点是小序号结点，同时这样的方法还可以消除重边的读入。

```
// 邻接矩阵表示图
static vector<vector<int>> AdjMatrix(MAX_VERTEX_NUM, vector<int>(MAX_VERTEX_NUM, 0));
static vector<int> visited(MAX_VERTEX_NUM, 0);
static int vexnum = 0, arcnum = 0;
```

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：

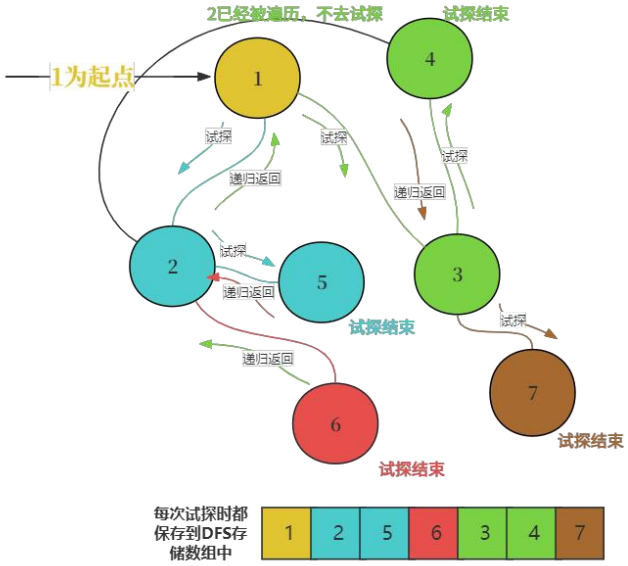


```cpp
// BFS 遍历
void BFS(int src_node, vector<int>& result) {
    queue<int> q;
    q.push(src_node);
    visited[src_node] = 1;

    while (!q.empty()) {
        int tmp_node = q.front();
        q.pop();
        result.push_back(tmp_node);

        for (int i = 0; i < vexnum; i++) {
            if (AdjMatrix[tmp_node][i] && !visited[i]) {
                q.push(i);
                visited[i] = 1;
            }
        }
    }
}
```

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。



```cpp
// DFS 遍历
void DFS(int src_node, vector<int>& result) {
    visited[src_node] = 1;
    result.push_back(src_node);

    for (int i = 0; i < vexnum; i++) {
        if (AdjMatrix[src_node][i] && !visited[i]) {
            DFS(i, result);
        }
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

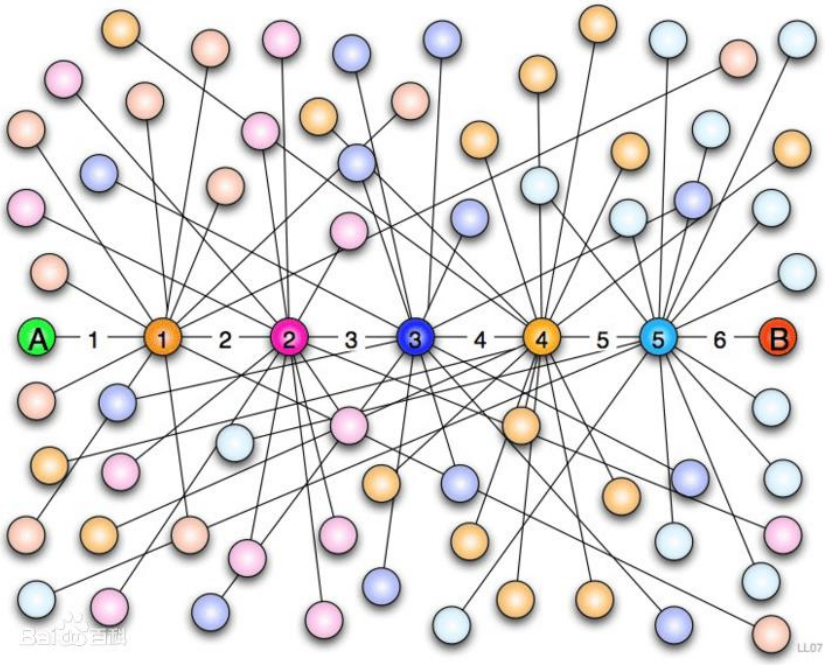
2.1.6 总结和体会

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。

2.2 题目二 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。"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。

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）。

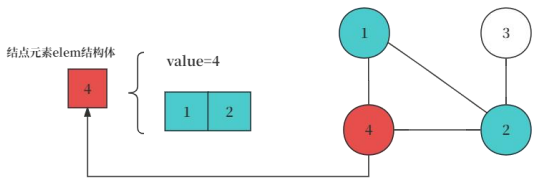
随后的 M 行对应 M 条边，每行给出一对正整数，分别是该条边直接连通的两个结点的编号（节点从 1 到 N 编号）。

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

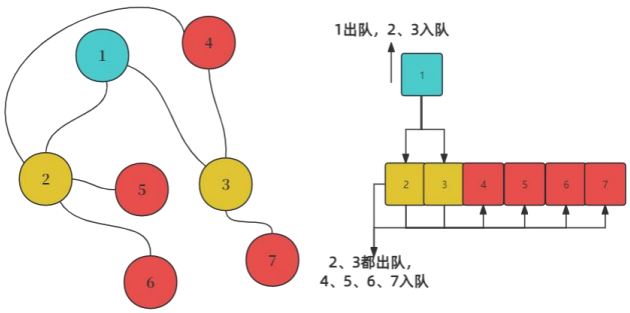
本题最大数据量共有 2000 个结点，如果使用邻接矩阵，则需要开一个 2000*2000 的矩阵，但是社交关系却最多只有 33*2000 个，空间利用率不足 2%，带来巨大的空间浪费。此外，每次对结点遍历都需要遍历为连接的结点，带来巨大的时间成本开销。因此这里不建议使用邻接矩阵完成。于是考虑使用邻接链表。
邻接链表结构需要单独开链表，构造起来较为复杂，这里选择直接使用 STL 容器中的 vector 来替代链表的手写，对于每个元素结点，构造如下元素结构体：




```
// 结点结构
struct Node {
    int value;
    vector<int> neighbors;
};

// 图的表示
vector<Node> graph;
vector<int> visited;
```

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：



```cpp
// BFS 遍历，返回 6 层内访问到的节点数量
int BFS(const Node& startNode, int vexnum) {
    queue<Node> q;
    visited.assign(vexnum + 1, 0); // 初始化访问状态
    q.push(startNode);
    visited[startNode.value] = 1;

    int count = 1; // 包括起始节点
    int layerCount = 1, currentLayer = 0;

    while (!q.empty() && currentLayer < 6) {
        int nodesInLayer = q.size();
        for (int i = 0; i < nodesInLayer; i++) {
            Node current = q.front();
            q.pop();

            for (int neighbor : current.neighbors) {
                if (!visited[neighbor]) {
                    visited[neighbor] = 1;
                    q.push(graph[neighbor]);
                    count++;
                }
            }
        }
        currentLayer++;
    }
    return count;
}
```

2.2.5 调试分析（遇到的问题和解决方法）

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。

2.3 题目三 村村通

2.3.1 问题描述

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。

下载编译并运行 p134.cpp 生成随机测试数据

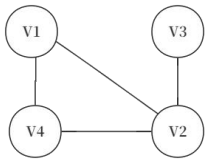
输出

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。

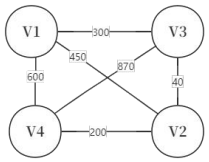
2.3.3 数据结构设计

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。

	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



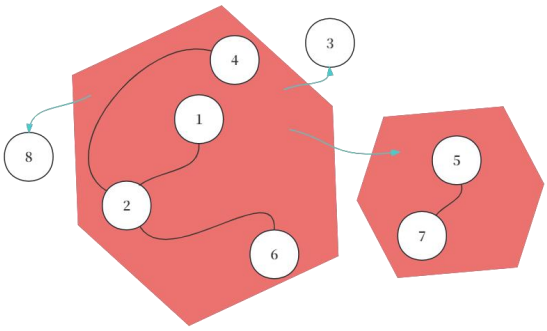

```
// 边的结构体
struct Edge {
    int src, dst, weight;
};
```

此外关于本问题中成环的问题我加入了并查集的 root 来控制，他表示某一点的父亲节点

```
class UnionFind {
    vector<int> parent, rank;
```

2.3.4 功能说明（函数、类）

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。
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:



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。

先执行排序按照最合适的边去挑选，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。在环问题上有一个并查集问题。

使用 unite 操作可以把并查集收紧，让 rank 最低的成为父亲。

```
void unite(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) {
        if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}

bool connected(int x, int y) {
    return find(x) == find(y);
}
```

```
for (int i = 0; i < m; ++i) {
    int src, dst;
    cin >> src >> dst;
    uf.unite(src, dst); // 标记为已连接，但不计权值
}
```

最后进行 kruskal 的最小生成树判断

```
int kruskal(int n, vector<Edge>& edges, UnionFind& uf) {
    // 按边权值排序
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.weight < b.weight;
    });

    int totalWeight = 0;
    int edgeCount = 0;

    for (const auto& edge : edges) {
        if (!uf.connected(edge.src, edge.dst)) {
            uf.unite(edge.src, edge.dst);
            totalWeight += edge.weight;
            edgeCount++;

            // 当边数达到（顶点数 - 1）时生成树完成
            if (edgeCount == n - 1) {
                break;
            }
        }
    }

    return totalWeight;
}
```

2.3.5 调试分析（遇到的问题和解决方法）

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。

2.4 题目四 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

邻接矩阵 AdjMatrix_row 和 AdjMatrix_col:分别表示行和列的依赖关系。用二维数组存储，适合稠密图场景。
DI_row 和 DI_col:记录行和列的入度。
Order_row 和 Order_col:存储拓扑排序后的节点顺序。
Matrix:最终生成的矩阵，结果存储在这里。
邻接表 row_links 和 col_links:记录每个节点的所有连接关系（行和列依赖）。

```
int k, n, m;
vector<vector<int>> AdjMatrix_row, AdjMatrix_col, Matrix;
vector<int> DI_row, DI_col, Order_row, Order_col;
vector<vector<int>> row_links, col_links;

Graph() : k(0), n(0), m(0) {}
```

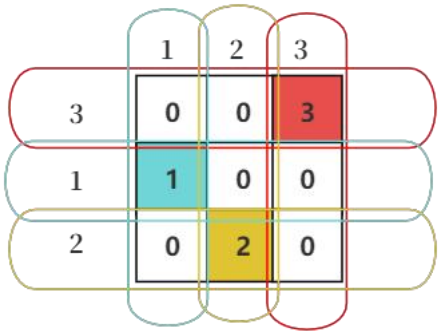
2.4.4 功能说明（函数、类）

createOrder():根据入度排序节点，生成拓扑排序的顺序。用于保证矩阵生成时的依赖关系正确。行和列分别调用此函数。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。

```
void createOrder(vector<int>& DI, vector<int>& Order, const vector<vector<int>>& links) {
    int count = 1;
    while (count <= k) {
        for (int i = 1; i <= k; ++i) {
            if (DI[i] == 0) {
                Order[count++] = i;
                DI[i] = -1;
                for (int v : links[i]) DI[v]--;
                break;
            }
        }
    }
}
```

createMatrix():利用拓扑排序生成最终的矩阵：对每个节点，判断其是否满足行和列的依赖一致性。若满足，则将节点填入矩阵对应位置。

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



```
void createMatrix() {
    for (int i = 1; i <= k; ++i)
        for (int j = 1; j <= k; ++j)
            Matrix[i][j] = (Order_row[i] == Order_col[j]) ? Order_row[i] : 0;
}
```

2.4.5 调试分析（遇到的问题和解决方法）

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 题目五 必修课

2.5.1 问题描述

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

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。

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

输出共 n 行，第 i 行包含两个整数：

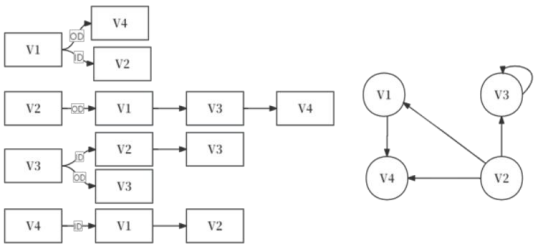
- 第一个整数表示编号为 i 的课程最早可能完成的时间。

- 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 11，否则输出 00。

每行的两个整数以一个空格隔开。

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：




```
struct Node {
    int value;                  // 节点编号
    int length;                 // 当前节点任务耗时
    vector<int> predecessors;   // 入度的节点
    vector<int> successors;     // 出度的节点
};
```

然后使用临接链表存贮图

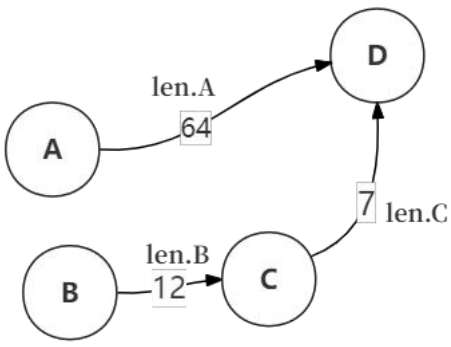
```
vector<Node> graph;              // 图的存储结构
vector<int> ve, vl, slack;       // 最早开始时间，最晚完成时间，每个任务的余量
vector<int> inDegree, outDegree; // 入度和出度
int nodeCount;                   // 节点数量
```

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

   如果要求 D 的最早开始时间：



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

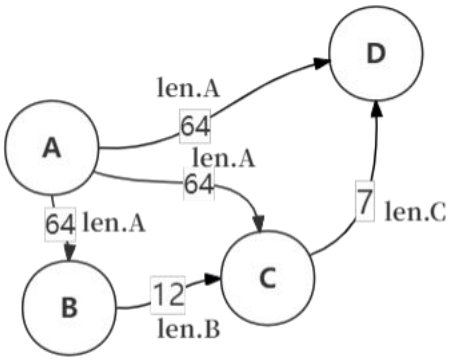
ve.this=max{ve.ID[i]+len.ID[i]}(i=0,1,2,3,……)。通过如上分析，得到了递推关系式。

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：

```
// 计算节点的最早开始时间
void calculateVe() {
    queue<int> q;

    for (int i = 0; i < nodeCount; i++) {
        if (inDegree[i] == 0) q.push(i);
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int successor : graph[u].successors) {
            ve[successor] = max(ve[successor], ve[u] + graph[u].length);
            if (--inDegree[successor] == 0) q.push(successor);
        }
    }
}
```

2.最晚开始时间 *vl*

如果要求 A 的最晚开始时间



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

$vl.this=min\{vl.OD[i]-len.this\}(i=0,1,2,3,\ldots\ldots)$。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：

```
// 计算节点的最晚完成时间
void calculateVl() {
    int maxGraduationTime = *max_element(ve.begin(), ve.end());

    for (int i = 0; i < nodeCount; i++) {
        vl[i] = maxGraduationTime;
    }

    queue<int> q;
    for (int i = 0; i < nodeCount; i++) {
        if (outDegree[i] == 0) q.push(i);
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int predecessor : graph[u].predecessors) {
            vl[predecessor] = min(vl[predecessor], vl[u] - graph[predecessor].length);
            if (--outDegree[predecessor] == 0) q.push(predecessor);
        }
    }
}
```

2.5.5 调试分析（遇到的问题和解决方法）

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。

2.6 题目六 小马吃草

2.6.1 问题描述

题目描述
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。

2.6.2 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

对于 20%的数据，1<=N、R<=10，1 <=w_i<=10；

对于 40%的数据，1<=N、R<=100，1<=w_i<=100；

对于 100%的数据，1<=N、R<=1000，1<=w_i<=100000；

对于所有数据，N-1<=M<=2N，1<=H<=N。

下载编译并运行 p132_data.cpp 生成随机测试数据

输出描述

输出共 R 行，表示 R 匹小马需要走过的最短距离

2.6.3 数据结构设计

牧草点：存储在 vector<int> grassPoints 中。
小马的起点和终点：存储在 vector<pair<int, int>> horses 中，每对元素表示一匹小马的起点和终点。
全图最短距离：vector<int> grassDist 存储每个点到所有牧草点的最短距离。
全图的距离用邻接链表存贮图的边以邻接表形式存储，采用 `vector<vector<Edge>> graph`：每个节点是一个 `vector<Edge>`，存储其所有的邻接边。每条边由结构体 `Edge` 表示，包含目标节点编号 `to` 和边权 `weight`。邻接表的存储方式适合稀疏图，空间效率较高。

```
vector<vector<Edge>> graph(n + 1);
```

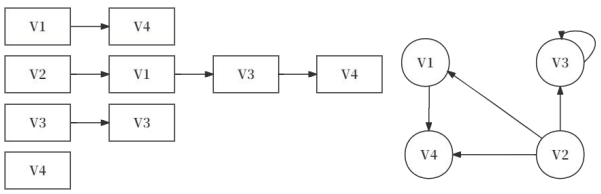


图 1.2 邻接链表

```
// 定义边
struct Edge {
    int to;
    int weight;
};
```

2.6.4 功能说明（函数、类）

先用一个朴素的 dijkstra 算法计算从单个起点到所有节点的最短距离

```cpp
// Dijkstra 算法计算单源最短路径
vector<int> dijkstra(int start, int n, const vector<vector<Edge>>& graph) {
    vector<int> dist(n + 1, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    dist[start] = 0;
    pq.push(make_pair(0, start));

    while (!pq.empty()) {
        pair<int, int> top = pq.top();
        int currentDist = top.first;
        int u = top.second;
        pq.pop();

        if (currentDist > dist[u]) {
            continue;
        }

        for (size_t i = 0; i < graph[u].size(); i++) {
            int v = graph[u][i].to;
            int weight = graph[u][i].weight;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    return dist;
}
```

对每匹小马：

1.分别运行两次 Dijkstra，计算从起点到所有节点（`distFromStart`）和从终点到所有节点（`distFromEnd`）的最短距离。

2.遍历所有牧草点，计算到达牧草点再到终点的总距离，取最小值作为答案。

```cpp
// 输出每匹小马的最短路径
for (size_t i = 0; i < horses.size(); i++) {
    int start = horses[i].first;
    int end = horses[i].second;

    vector<int> distFromStart = dijkstra(start, n, graph);
    vector<int> distFromEnd = dijkstra(end, n, graph);

    int minDist = INT_MAX;
    for (size_t j = 0; j < grassPoints.size(); j++) {
        int grass = grassPoints[j];
        int totalDist = distFromStart[grass] + distFromEnd[grass];
        if (totalDist < minDist) {
            minDist = totalDist;
        }
    }

    cout << minDist << endl;
}

return;
```

2.6.5 调试分析（遇到的问题和解决方法）

2.6.6 总结和体会

本题针对稀疏图的 node 存贮方式采用了邻接链表。重点考察了 **Dijkstra 算法**：是解决单源最短路径问题的经典算法，适用于正权图。使用优先队列实现堆优化，复杂度降为O((n+m)log⁡n)O((n + m) \log n)O((n+m)logn)。本题中，Dijkstra 算法被多次调用，用于：计算从牧草点到所有节点的最短距离。计算小马的起点或终点到所有节点的最短距离。 本题对路径的组合进行了优化：通过两次 Dijkstra 计算起点和终点分别到牧草点的最短距离，然后枚举所有牧草点找到最短路径。理解了如何组合路径的子问题，以及通过优化中间点选择提升效率的思想。

3. 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。还有一些 Dijkstra 算法的优化问题。