作业 HW4 实验报告

姓名：刘彦 学号：2352018 日期：2024 年 11 月 26 日

1. 涉及数据结构和相关背景

1.1 图的概念

图是一种表示对象及其相互关系的数据结构，由顶点（节点）和边（连接）组成。

- 无向图：边没有方向，表示双向关系。
- 有向图：边有方向，表示单向关系。
- 带权图：边上有权重，用于表示代价、距离等。

1.2 图的存储方式

图的存储方式直接影响算法的效率和资源消耗，常见方法包括：

1.2.1 邻接矩阵

用二维数组表示顶点之间的关系。

优点：访问任意两点关系简单高效 O(1)。

缺点：对稀疏图浪费大量空间。

空间复杂度：$O(n^2)$

1.2.2. 邻接表

每个顶点用链表存储与其相连的顶点。

优点：对稀疏图节省空间。

缺点：查找特定边的效率较低。

空间复杂度：O(n+m)（n 为顶点数，m 为边数）

1.2.3 逆邻接表

与邻接表类似，但存储的是每个顶点的入边（指向该顶点的边）。

常用于反向图操作，例如反向拓扑排序。

1.3 图的遍历

1.3.1 广度优先搜索（BFS）

按层次逐层访问图的节点，通常用队列实现。

优点：找到最短路径，适合连通性检测。

时间复杂度：O(n+m)

深度限制：通过限制搜索深度，避免无意义遍历。

1.3.2 深度优先搜索（DFS）

采用递归或栈，沿一条路径尽可能深地探索。

优点：适合路径、环检测等问题。

时间复杂度：O(n+m)

深度限制：限制递归深度以控制性能或防止爆栈。

1.4 最小生成树（MST）

MST 是一种连接所有顶点且权重总和最小的树，常用于网络优化问题。

- Kruskal 算法

基于边排序的贪心算法，按权值从小到大加入边，避免成环。
通常用并查集判断连通性。
时间复杂度：O(mlogm)
适合稀疏图。

- Prim 算法
  基于顶点扩展，始于某点，每次加入权值最小的相邻边。
  可用优先队列优化选择边的过程。
  时间复杂度：O(mlogn)（堆优化）。
  适合稠密图

1.5 拓扑排序

拓扑排序是对有向无环图（DAG）节点的线性排序，保证任意一条边的起点在终点前。

- AOV 网（Activity on Vertex）：用于描述事件之间的先后关系。
- AOE 网（Activity on Edge）：用于表示事件的时间计划问题，可用于关键路径分析。

1.6 最短路径问题

1.6.1 Dijkstra 算法

解决单源最短路径问题，适合非负权图。
时间复杂度：$O(n^2)$（未优化），O(mlogn)（堆优化）

1.6.2 Floyd 算法

解决多源最短路径问题，基于动态规划思想。
时间复杂度：$O(n^3)$
优点：适合处理稠密图且需要多源路径时使用。

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。

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。

2.1.3 数据结构设计

```cpp
struct Graph { // 图的结构体
    int adj[MAXN][MAXN]; // 邻接矩阵  int n;  // 顶点数
    bool visited[MAXN];  // 访问标记数组
    void init(int vertices) { // 初始化图
        n = vertices;
        memset(adj, 0, sizeof(adj));
        memset(visited, 0, sizeof(visited));
    }
    void addEdge(int u, int v) { // 添加一条边
        adj[u][v] = 1;  adj[v][u] = 1;
    }
};
struct Queue { // 队列结构体
    int data[MAXN];  int front, rear;
    void init() { // 初始化队列
        front = rear = 0;
    }
    void enqueue(int value) { // 入队
        data[rear++] = value;
    }
    int dequeue() { // 出队
        return data[front++];
    }
    bool isEmpty();  // 判断队列是否为空
};
struct Component { // 存储连通分量的结构体
    int nodes[MAXN]; // 节点列表  int size;        // 节点数量
    void init() { // 初始化
        size = 0;
    }
    void addNode(int node) { // 添加节点
        for (int i = 0; i < size; ++i)
            if (nodes[i] == node) return;
        nodes[size++] = node;
    }
    void print(); // 输出连通分量
};
```

2.1.4 功能说明（函数、类）

深度优先搜索的函数

```
/
 * @brief          深度优先搜索（DFS）算法
 * @param graph     图的引用
 * @param v         当前节点
```

```
 * @param component 连通分量的引用
 */
void dfs(Graph& graph, int v, Component& component) {
    graph.visited[v] = true;
    component.addNode(v);
    for (i 从 0 到 graph.n)
        if (graph.adj[v][i] && !graph.visited[i])
            dfs(graph, i, component);
}
```

广度优先搜索的函数

```
/
 * @brief          广度优先搜索（BFS）算法
 * @param graph     图的引用
 * @param start     起始节点
 * @param component 连通分量的引用
 */
void bfs(Graph& graph, int start, Component& component) {
    Queue queue;  queue.初始化和入队
    graph.visited[start] = true;
    while (!queue.isEmpty()) {
        int v = queue.dequeue();  component.addNode(v);
        for (i 从 0 到 graph.n)
            if (graph.adj[v][i] && !graph.visited[i]) {
                queue.enqueue(i);  graph.visited[i] = true;
            }
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

2.1.5.1 重复添加节点到连通分量

DFS 或 BFS 可能会重复将节点加入连通分量。最后的解决方法是在 addNode 方法中增加检查，确保一个节点只添加一次。

2.1.5.2 扩展优化

动态调整图规模，当前 MAXN 是固定的，可考虑动态分配图的存储空间。

```
int adj;
adj = new int*[n];
for (int i = 0; i < n; ++i) {
    adj[i] = new int[n]();
}
```

2.1.6 总结和体会

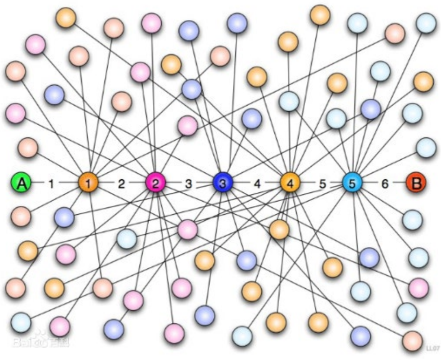
DFS 常用于搜索一个可行解，比如找某个节点是否可以到达终点，或者找到一条满足条件的路径。空间复杂度与递归深度成正比，在图中为 O(V)（V 为节点数）。如果图的深度较深，DFS 的栈消耗会显著增加。

BFS 会优先找到路径长度较短的解，适合用于寻找所有解中具有特殊性质的解（例如最短路径）。由于按层次遍历，BFS 能够更早发现距离起点较近的目标节点。时间复杂度为 O(V+E)，其中 E 是边数。空间复杂度较高，需要存储当前层和下一层的节点（最坏情况下为 O(V)）。

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。”如图所示。



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

```
struct Graph {
    int head[MAXN + 1]; // 每个节点的邻接链表起始位置
    int edges[MAXM * 2]; // 边数组
    int next[MAXM * 2];  // 下一条边的索引
    int edgeCount;       // 当前边数  int n; // 节点数
    void init(int nodes) { // 初始化图
        n = nodes;  edgeCount = 0;
        memset(head, -1, sizeof(head));
    }
    void addEdge(int u, int v) { // 添加一条无向边
        edges[edgeCount] = v;
        next[edgeCount] = head[u];
```

```
        head[u] = edgeCount++;
        edges[edgeCount] = u;
        next[edgeCount] = head[v];
        head[v] = edgeCount++;
    }
};
struct Queue { // 队列结构体，与 2.1 的非常类似，故省略了一些代码
    int data[MAXN + 1];  int front, rear;
    void init() // 初始化队列  void enqueue(int value); // 入队
    int dequeue() // 出队  bool isEmpty() // 判断队列是否为空
};
```

2.2.4 功能说明（函数、类）

计算六度空间范围的函数

```
/
 * @brief            计算六度空间范围内的节点数
 * @param graph      图的引用
 * @param start      起始节点
 * @return           返回覆盖的节点数
 */
int sixDegrees(Graph& graph, int start) {
   bool visited[MAXN + 1]; // 访问标记数组
    memset(visited, 0, sizeof(visited));
    Queue queue;  queue.初始化和入队
    visited[start] = true;
    int level = 0;  // 当前层数  int count = 1;  // 覆盖的节点数，包括自身
    while (!queue.isEmpty() && level < 6) { // 开始 BFS 遍历
        int size = queue.rear queue.front; // 当前层节点数
        for (i 从 0 到 size) {
            int node = queue.dequeue(); // 从队列中取出一个节点
            for (int j = graph.head[node]; j != -1; j = graph.next[j]) {
                int neighbor = graph.edges[j]; // 遍历该节点的所有邻居节点
                if (未被访问) {
                    queue.enqueue(neighbor);
                    标记访问并计数
                }
            }
        }
        level++;
    }
    return count;
}
```

2.2.5 调试分析（遇到的问题和解决方法）

2.2.5.1 邻接矩阵导致的遍历效率低下

一开始使用邻接矩阵，在六度空间的遍历中，循环检查 graph.adj[node][j]对于所有节点 j 是否相邻，复杂度为 $O(n^2)$。当 n 较大且每个节点的邻接点很少时（稀疏图），大部分检查是无意义的。

```
struct Graph {
    bool adj[MAXN + 1][MAXN + 1]; // 邻接矩阵
    void addEdge(int u, int v) { // 添加边
        adj[u][v] = true;
        adj[v][u] = true;
    }
};
```

现在改为邻接链表，只检查实际的邻接点，避免不必要的遍历，复杂度降低为 O(m)。BFS 遍历的总复杂度减少为 O(n+m)，更适合稀疏图。

2.2.5.2 访问标记初始化问题

使用 bool visited[MAXN + 1] = {0}初始化时，可能由于数组大小超限或初始化效率较低，导致程序性能不佳。每次调用 sixDegrees 函数都会重新初始化 visited，如果节点数量较大，效率会进一步降低。最终版本在在主程序中统一管理 visited 数组，避免重复初始化。并使用 memset 初始化，提高初始化速度：memset(visited, 0, sizeof(visited))。

2.2.6 总结和体会

选择合适的图存储方式是关键，邻接矩阵在处理大规模稀疏图时空间浪费严重，邻接链表虽然实现复杂度更高，但更高效。

时间复杂度：邻接矩阵：每次 BFS 的复杂度为 $O(n^2)$。邻接链表：每次 BFS 的复杂度为 O(n+m)（更优）。

空间复杂度：邻接矩阵：$O(n^2)$，适合稠密图。邻接链表：O(n+m)，适合稀疏图。

易错点是数组初始化多次调用 BFS 时，忘记重新初始化 visited 数组，导致结果错误。使用局部变量 visited 或手动清零时，可能超出数组范围或忘记初始化部分元素。

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。

3<=n<=100;

边权都是[1,1000]之间的整数。

2.3.3 数据结构设计

```
class Edge { // 边的类
public:
    int u, v, weight;
    Edge() {
        u = 0;  v = 0;  weight = 0;
    }
    Edge(int u, int v, int weight) { // 带参数构造函数
        this->u = u;  this->v = v;
        this->weight = weight;
    }
};
class Graph {  // 图的类
private:
    Edge edges[MAXE]; // 存储所有边
    int edgeCount;    // 边的总数
    int parent[MAXN]; // 并查集父节点
    int rank[MAXN];   // 并查集秩
public:
    Graph() {
        edgeCount = 0;
    }
    void addEdge(int u, int v, int weight);
    void initUnionFind(int n);
    int find(int x);
    bool unionSets(int x, int y);
    void sortEdges();
    int kruskal(int n);
};
```

2.3.4 功能说明（函数、类）

向图中添加边的函数

```
/
 * @brief          向图中添加边
 * @param u        边的起点
 * @param v        边的终点
 * @param weight    边的权重
 */
void Graph::addEdge(int u, int v, int weight) {
    edges[edgeCount++] = Edge(u, v, weight); // 将边添加到边数组，并增加边计数
}
```

初始化的函数

```
/
 * @brief          初始化并查集
 * @param n        图中节点的数量
 */
void Graph::initUnionFind(int n) {
    for (i 从 1 到 n)
        每个节点的父节点初始化为自身，每个节点的排序初始化为 0
}
```
查找根节点的函数
```
/
 * @brief          查找节点 x 的根节点，并进行路径压缩
 * @param x        要查找的节点
 * @return         返回根节点
 */
int Graph::find(int x) {
    if (parent[x] != x)
        parent[x] = find(parent[x]); // 路径压缩
    return parent[x];
}
```
合并两个集合的函数
```
/
 * @brief          合并两个集合
 * @param x        第一个节点
 * @param y        第二个节点
 * @return         如果合并成功返回 true，否则返回 false
 */
bool Graph::unionSets(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]++;
        }
        return true;
    }
    return false;
}
```
对图中的边进行排序的函数
```
/
 * @brief          对图中的边进行排序
 */
void Graph::sortEdges() {
    for (int i = 0; i < edgeCount 1; ++i)
```

```
        for (int j = 0; j < edgeCount i 1; ++j)
            if (edges[j].weight > edges[j + 1].weight)
                交换 edges[j]和 edges[j + 1];
}
```

Kruskal 算法函数

```
/
 * @brief           使用 kruskal 算法找到最小生成树的总权重
 * @param n         图中节点的数量
 * @return          返回最小生成树的总权重
 */
int Graph::kruskal(int n) {
    sortEdges();          // 对边按权重排序
    int totalCost = 0;
    for (int i = 0; i < edgeCount; ++i) {
        int u = edges[i].u, v = edges[i].v;
        if (find(u) != find(v)) { // 如果两点属于不同连通分量
            unionSets(u, v);  totalCost += edges[i].weight;
        }
    }
    return totalCost;
}
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.5.1 已有连通性未处理

已经连通的边重复计入生成树。经检查，原因是主函数中 unionSets 调用不正确。调试 unionSets，打印初始父节点数组，检查合并逻辑。在处理边时，通过下标调整（如(a 1 和 b 1），保证了输入和内部逻辑的一致性。并使用 addEdge，显式标记已有边权重为零，在处理已有的边时，可以将这些边的权重直接标记为零，以避免在排序时被重新考虑

```
    for (int i = 0; i < m; ++i) { // 处理已有的边
        int a, b;
        cin >> a >> b;
        graph.addEdge(a 1, b 1, 0);
        graph.unionSets(a 1, b 1); // 直接合并已有的路
    }
```

2.3.5.2 避免冗余边检查

一开始的逻辑在已经通过 unionSets 避免了重复选择边的情况后，但为了进一步优化，可显示跳过冗余边。原来代码如下：

```
for (i 从 0 到 edgeCount)
        if (unionSets(edges[i].u, edges[i].v)) // 如果合并成功
            totalCost += edges[i].weight; // 增加权重
```

修改后：

```
for (int i = 0; i < edgeCount; ++i) {
        int u = edges[i].u, v = edges[i].v;
        if (find(u) != find(v)) { // 如果两点属于不同连通分量
```

```
            unionSets(u, v);  totalCost += edges[i].weight;
        }
    }
```
这种优化能够减少无效的 unionSets 调用，提升运行效率。

2.3.6 总结和体会

我学会了如何利用 Kruskal 算法通过边的排序与并查集高效地解决最小生成树问题。并明白了并查集的路径压缩和秩合并优化，在算法中起到了关键作用，有效降低了连通性判断的复杂度。

从冗余边的处理到循环终止条件的添加，逐步优化算法效率的过程中，我理解了排序对性能的影响，并学习如何在实现中避免不必要的操作。

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×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

```
class Graph {
private:
    struct ArcNode {
        int adj_vex;            // 该弧的终点
        ArcNode* next_arc;      // 后继弧
    };
    struct VNode {
        int in_degree = 0;      // 顶点的入度
        ArcNode* first_arc = nullptr;  // 顶点的弧链表
    };
    VNode* vertices;          // 存储图的顶点
    int vex_num, arc_num;       // 顶点数和边数
public:
```

```
    Graph(int vex_num, int arc_num);
    ~Graph();
    void add_arc(int src, int dst); // 添加弧到图中
    bool topo_sort(int* arr); // 拓扑排序
    int get_vex_num(); // 获取顶点数
};
```

2.4.4 功能说明（函数、类）

向图中添加弧的函数

```
/
 * @brief          向图中添加弧
 * @param src      弧的起始顶点
 * @param dst      弧的终点顶点
 */
void Graph::add_arc(int src, int dst) {
    ArcNode* q = new ArcNode;
    q->adj_vex = dst;
    q->next_arc = vertices[src].first_arc;
    vertices[src].first_arc = q;
    vertices[dst].in_degree++;  // 目标顶点入度加 1
}
```

拓扑排序函数

```
/
 * @brief          图的拓扑排序
 * @param arr      拓扑排序结果数组
 * @return         如果拓扑排序成功返回 true，否则返回 false
 */
bool Graph::topo_sort(int* arr) {
    int count = 0;
    while (true) {
        int cur = -1;
        for (cur 从 1 到 vex_num)
            if (vertices[cur].in_degree == 0)
                break;  // 找到入度为 0 的点
        if (cur > vex_num)
            break;  // 没有入度为 0 的点了
        count++;  arr[cur] = count;  // 记录拓扑排序顺序
        vertices[cur].in_degree = -1;  // 置为已访问
        for (ArcNode* p = vertices[cur].first_arc; p; p = p->next_arc)
            vertices[p->adj_vex].in_degree--;  // 更新相邻节点的入度
    }
    return count == vex_num;  // 如果排序数量等于顶点数，说明拓扑排序成功
}
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.5.1 拓补排序不同记录方式的区别

arr[cur] = count 的值是顶点的拓扑排序顺序，适用于查询顶点编号的排序顺序。arr[count] = cur 的值是对应位置的顶点编号，适用于输出或遍历整个拓扑排序序列。一开始使用错误导致输出结果错误。

2.4.5.2 环检测逻辑

一开始没有环检测逻辑，topo_sort 直接 return true，程序未正确识别存在环的情况，导致死循环或错误输出。后来改为 return count == vex_num 当存在环时，直接输出错误信息，并返回 false，停止进一步操作。

2.4.6 总结和体会

在实现拓扑排序时，容易出错的点：找入度为 0 的节点时如何高效地遍历。多个入度为 0 的节点可能导致不同的拓扑排序结果，这在输出顺序上需谨慎处理。

环的处理需要在拓扑排序中添加逻辑检测。确保当图无法完成排序时，能够正确返回提示或终止程序。

虽然题目中顶点数和边数范围不大，但在处理较大的 DAG 时，邻接表的实现比邻接矩阵更高效。

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 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 1，否则输出 0。每行的两个整数以一个空格隔开。

对于所有数据，满足：

$$1 \le n \le 100$$

$$1 \le ti \le 100$$

$$0 \le ci < n$$

时间限制： 1 sec

内存限制： 256 MB。

2.5.3 数据结构设计

```
class Graph_course {
public:
```

```
    struct ArcNode {
        int adj_vex;             // 该弧的终点
        ArcNode* next_arc;       // 后继的弧
        ArcNode();
    };
    struct VNode {
        int in_degree;          // 入度
        int val;                // 学时
        int max_preval;         // 前驱结点的最大学时
        int* pre;               // 前驱结点（用数组代替）
        int pre_size;           // 前驱结点数量
        ArcNode* first_arc;      // 邻接链表头指针
        VNode() {
            //……
            pre = new int[100];  // 假设最多 100 个前驱结点
        }
        ~VNode();
    };
    int vex_num;             // 图中课程数量
    int arc_num;             // 图中边的数量
    VNode* vertices;        // 图的邻接表
    Graph_course(int n);
    ~Graph_course();
    void add(int src, int dst); // 添加边
    int max(int a, int b); // 返回最大值
    bool topo_sort();// 拓扑排序计算每门课程的最早完成时间
    bool dfs(int cur, int tgt); // 深度优先搜索，检查课程是否影响总毕业时间
};
```

2.5.4 功能说明（函数、类）

向图中添加边的函数

```
/
 * @brief           向图中添加边
 * @param src        边的起点
 * @param dst        边的终点
 */
void Graph_course::add(int src, int dst) {
    ArcNode* p = vertices[src].first_arc;
    ArcNode* q = new ArcNode();
    q->adj_vex = dst;
    if (!q) exit(-1);
    if (p) {
        while (p->next_arc) p = p->next_arc;
        p->next_arc = q;
```

```
    } else  vertices[src].first_arc = q;
    arc_num++;
}
```

拓扑排序计算每门课程的最早完成时间的函数

```
/
 * @brief           拓扑排序计算每门课程的最早完成时间
 * @return          如果拓扑排序成功返回 true，否则返回 false
 */
bool Graph_course::topo_sort() {
    int cnt = 0;
    while (true) {
        int cur = -1;
        for (int i = 1; i <= vex_num; i++)
            if (vertices[i].in_degree == 0) {
                cur = i;  break;
            }
        if (cur == -1) break;  // 没有入度为 0 的点，退出
        cnt++;
        vertices[cur].in_degree--;  // 处理该课程
        vertices[cur].val += vertices[cur].max_preval;  // 更新课程的完成时间
        for (ArcNode* p = vertices[cur].first_arc; p; p = p->next_arc) {
            vertices[p->adj_vex].in_degree--;
            如果当前节点的值大于后继节点已知的最大前驱值：
               后继节点将更新其最大前驱值为当前节点的值，
               同时将前驱节点集合重置，仅包含当前节点。
            如果当前节点的值等于后继节点的最大前驱值：
               当前节点会被追加到后继节点的前驱节点集合中，
               表示该节点也是一个等值的前驱节点。
    }
    return cnt == vex_num;
}
```

深度优先搜索函数

```
/
 * @brief           深度优先搜索，检查课程是否影响总毕业时间
 * @param cur       当前课程
 * @param tgt       目标课程
 * @return          如果当前课程影响目标课程的毕业时间返回 true，否则返回 false
 */
bool Graph_course::dfs(int cur, int tgt) {
    if (cur == tgt) return true;
    if (vertices[cur].pre_size == 0) return false;
    for (int i = 0; i < vertices[cur].pre_size; i++) {
        if (dfs(vertices[cur].pre[i], tgt)) return true;
    }
```

```
    return false;
}
```

2.5.5 调试分析（遇到的问题和解决方法）

2.5.5.1 大数据集处理

对于大规模数据集，考虑使用 `std::queue` 代替遍历查找入度为零的节点，以实现更高效的查找。

2.5.5.2 数组越界

考虑使用使用 `std::vector<int>`代替固定大小的 `pre` 数组，以便动态调整大小，避免数组越界问题。

现在在访问 `vertices[cur].pre[i]`时，确保 i 小于 `vertices[cur].pre_size`。

2.5.6 总结和体会

我深入了解了如何在节点中存储前驱节点的信息，并在后续处理中遍历它们。在拓扑排序中，更新相邻节点的入度时需要小心，确保每次处理后都能正确地减少入度。
在更新前驱节点信息时，如果不小心处理重复的前驱或数组越界，可能导致程序出错。

2.6 小马吃草

2.6.1 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。

2.6.2 基本要求

输入：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。

输出：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

```
struct ArcNode {
    int adjvex;     // 该弧的终点
    int val;        // 边权
    ArcNode* next_arc;
    ArcNode(int v, int w);
};
```

```
struct VNode {
    ArcNode* first_arc;
    VNode();
};
struct Node {
    int dis, vex;
    bool operator<(const Node& a) const {
        return dis > a.dis;  // 小根堆
    }
};
class Graph {
public:
    VNode vertices[MAX_NODE_NUM];
    int vex_num, arc_num;
    Graph(int n, int m) {
        vex_num = n;  // 设置图的顶点数量
        arc_num = m;  // 设置图的边的数量
    }
    void addArc(int src, int dst, int val) {
        ArcNode* arc = new ArcNode(dst, val);
        arc->next_arc = vertices[src].first_arc;
        vertices[src].first_arc = arc;
    }
};
```

2.6.4 功能说明（函数、类）

向图中添加弧的函数

```
/
 * @brief          使用 dijkstra 算法计算从起点到所有其他点的最短路径
 * @param graph     图的引用
 * @param dis       距离数组，存储从起点到各点的距离
 * @param start     起点
 */
void dijkstra(Graph& graph, int* dis, int start) {
    priority_queue<Node> pq;
    bool vis[MAX_NODE_NUM] = {false};
    for (i 从 1 到 graph.vex_num)
        dis[i] = INT_MAX;
    dis[start] = 0;
    pq.push({0, start});
    while (pq 不空) {
        Node curr = pq.top();
        pq.pop();
        int u = curr.vex;
```

```
        if (vis[u]) continue;
        vis[u] = true;
        for (ArcNode* p = graph.vertices[u].first_arc; p; p = p->next_arc){
            int v = p->adjvex, w = p->val;
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;  pq.push({dis[v], v});
            }
        }
    }
}
```

main 函数中关键部分

```
    // 创建并初始化一个二维数组，存储每个牧草点到所有点的距离
    int dis[MAX_NODE_NUM][MAX_NODE_NUM];  // 牧草点到所有点的距离
    for (i 从 0 到 grass_num)
        dijkstra(graph, dis[grass[i]], grass[i]); // 对每个牧草点运行
    // 处理每一对马匹起点和终点，找到其最短路径
    for (i 从 0 到 horse_num) {
        int src, dst;
        cin >> src >> dst; // 输入马匹的起点和终点
        int min_path = INT_MAX; // 初始化最小路径长度为无穷大
        // 遍历所有的牧草点，找到从源点到终点的最短路径
        for (j 从 0 到 grass_num) {
            int g = grass[j];
            // 确保从牧草点到源点和从牧草点到终点的路径存在
            if (dis[g][src] != INT_MAX && dis[g][dst] != INT_MAX)
                min_path = min(min_path, dis[g][src] + dis[g][dst]); // 更新
        }
        cout << min_path << endl; // 输出从源点到终点经过牧草点的最短路径长度
    }
```

2.6.5 调试分析（遇到的问题和解决方法）

队列的使用：使用一个数组来存储每个顶点的当前距离，并在每次迭代中选择未访问顶点中距离最小的点进行更新。这种方法的时间复杂度是 $O(V^2)$，导致超时，最后我通过通过优化 Dijkstra 的实现，将数组替换为堆结构（优先队列）`priority_queue<Node>`以降低复杂度到 $O((V+E)\log V)$。

2.4.6 总结和体会

我学习并实现了 Dijkstra 算法，基础实现中，未优化的 Dijkstra 算法的时间复杂度是 $O(V^2)$，适合处理稠密图。优化后使用堆，时间复杂度降低为 $O((V+E)\log V)$，对稀疏图更为高效。

针对多个牧草点作为起点的场景，我通过多次调用 Dijkstra，分别计算出从每个牧草点到其他所有节点的最短距离。在查询时，逐一比较起点和终点经过每个牧草点的最短路径，最终找到全局最优解。

3. 实验总结

3.1 图的存储方式对比

邻接矩阵适用于稠密图，存储简单，但在边较少时浪费空间。邻接表和链式前向星更加灵活，特别是对于稀疏图，显著降低了空间消耗。逆邻接表在需要频繁访问入边信息时（如反向图操作）效率更高。我通过实现和对比发现，选择合适的存储方式对算法效率和内存消耗有显著影响。

3.2 图的遍历

BFS 适合解决层次关系、最短路径（无权图）等问题。DFS 在路径查找、连通分量检测和环检测等问题中表现良好。深度限制有效控制了搜索范围，避免了不必要的计算和内存开销。

我通过实验观察到：对于大规模图数据，遍历过程中适当限制深度或优化存储结构可以显著提升性能。

3.3 堆优化的基本思想

堆优化通过最小堆（优先队列），动态维护待处理顶点的最短距离或最小权值。特别适用于需要频繁处理优先级队列或选择最小/最大值的场景。

堆操作（插入、删除、更新）的复杂度为 O(logn)，显著减少了从线性扫描寻找最小值的时间开销。

例如：堆优化 Dijkstra 算法，用于求解单源最短路径问题。

- 核心思想：将所有顶点的当前最短距离存入最小堆。每次从堆中弹出当前距离最小的顶点，对其邻接边进行松弛操作。松弛操作更新邻接顶点的最短距离，若距离变小，将其重新插入堆中。
- 时间复杂度
  建堆操作：每个顶点最多入堆一次，时间复杂度为 O(VlogV)。
  边松弛操作：每条边最多更新一次，复杂度为 O(ElogV)。
  总复杂度：O((V+E)logV)

3.4 实验收获与思考

通过实验，我掌握了图的存储方式及不同算法的实现细节和优化方法。理解了算法设计的核心思想，如贪心、动态规划等在图问题中的应用。对算法的时间、空间复杂度有了更深入的认识，学会了根据问题需求选择合适的算法，也学习了一些新的算法。