宫水三叶的刷题日记

最短路

Author ：宫水三叶

Date ：2021/10/07

QQ Group：703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「图论：最短路」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「图论：最短路」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「图论：最短路」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **743. 网络延迟时间** ，难度为 **中等**。

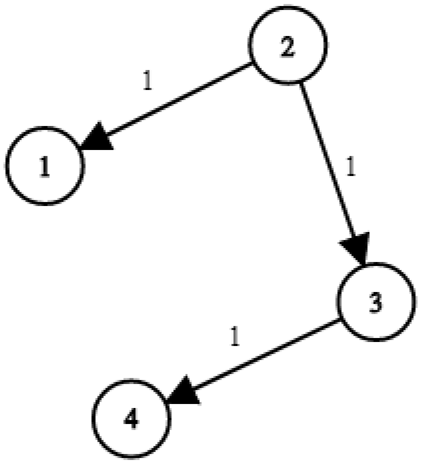
Tag : 「最短路」、「图」、「优先队列（堆）」

有 n 个网络节点，标记为 1 到 n。

给你一个列表 times，表示信号经过 有向 边的传递时间。 times[i] = (ui, vi, wi)，其中 ui 是源节点，vi 是目标节点， wi 是一个信号从源节点传递到目标节点的时间。

现在，从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点都收到信号，返回 -1 。

示例 1：




```
输入：times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

输出：2
```

示例 2：

```
输入：times = [[1,2,1]], n = 2, k = 1

输出：1
```

示例 3：

```
输入：times = [[1,2,1]], n = 2, k = 2

输出：-1
```

提示：

- 1 <= k <= n <= 100
- 1 <= times.length <= 6000
- times[i].length == 3
- 1 <= ui, vi <= n
- ui != vi
- 0 <= wi <= 100
- 所有 (ui, vi) 对都 互不相同（即，不含重复边）

基本分析

为了方便，我们约定 n 为点数，m 为边数。

根据题意，首先 n 的数据范围只有 100，m 的数据范围为 6000，使用「邻接表」或「邻接矩阵」来存图都可以。

同时求的是「**从 k 点出发，所有点都被访问到的最短时间**」，将问题转换一下其实就是求「**从 k 点出发，到其他点 x 的最短距离的最大值**」。

存图方式

在开始讲解最短路之前，我们先来学习三种「存图」方式。

邻接矩阵

这是一种使用二维矩阵来进行存图的方式。

适用于边数较多的**稠密图**使用，当边数量接近点的数量的平方，即 $m \approx n^2$ 时，可定义为**稠密图**。

```
// 邻接矩阵数组：w[a][b] = c 代表从 a 到 b 有权重为 c 的边
int[][] w = new int[N][N];

// 加边操作
void add(int a, int b, int c) {
    w[a][b] = c;
}
```

邻接表

这也是一种在图论中十分常见的存图方式，与数组存储单链表的实现一致（头插法）。

这种存图方式又叫**链式前向星存图**。

适用于边数较少的**稀疏图**使用，当边数量接近点的数量，即 $m \approx n$ 时，可定义为**稀疏图**。

```java
int[] he = new int[N], e = new int[M], ne = new int[M], w = new int[M];
int idx;

void add(int a, int b, int c) {
    e[idx] = b;
    ne[idx] = he[a];
    he[a] = idx;
    w[idx] = c;
    idx++;
}
```

首先 `idx` 是用来对边进行编号的，然后对存图用到的几个数组作简单解释：

- `he` 数组：存储是某个节点所对应的边的集合（链表）的头结点；
- `e` 数组：由于访问某一条边指向的节点；
- `ne` 数组：由于是以链表的形式进行存边，该数组就是用于找到下一条边；
- `w` 数组：用于记录某条边的权重为多少。

因此当我们想要遍历所有由 `a` 点发出的边时，可以使用如下方式：

```java
for (int i = he[a]; i != -1; i = ne[i]) {
    int b = e[i], c = w[i]; // 存在由 a 指向 b 的边，权重为 c
}
```

类

这是一种最简单，但是相比上述两种存图方式，使用得较少的存图方式。

只有当我们需要确保某个操作复杂度严格为 $O(m)$ 时，才会考虑使用。

具体的，我们建立一个类来记录有向边信息：

```java
class Edge {
    // 代表从 a 到 b 有一条权重为 c 的边
    int a, b, c;
    Edge(int _a, int _b, int _c) {
        a = _a; b = _b; c = _c;
    }
}
```

通常我们会使用 List 存起所有的边对象，并在需要遍历所有边的时候，进行遍历：

```java
List<Edge> es = new ArrayList<>();

...

for (Edge e : es) {
    ...
}
```

Floyd（邻接矩阵）

根据「基本分析」，我们可以使用复杂度为 $O(n^3)$ 的「多源汇最短路」算法 Floyd 算法进行求解，同时使用「邻接矩阵」来进行存图。

此时计算量约为 10^6，可以过。

跑一遍 Floyd，可以得到「**从任意起点出发，到达任意起点的最短距离**」。然后从所有 $w[k][x]$ 中取 max 即是「**从 k 点出发，到其他点 x 的最短距离的最大值**」。

代码：

```java
class Solution {
    int N = 110, M = 6010;
    // 邻接矩阵数组：w[a][b] = c 代表从 a 到 b 有权重为 c 的边
    int[][] w = new int[N][N];
    int INF = 0x3f3f3f3f;
    int n, k;
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        // 初始化邻接矩阵
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                w[i][j] = w[j][i] = i == j ? 0 : INF;
            }
        }
        // 存图
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            w[u][v] = c;
        }
        // 最短路
        floyd();
        // 遍历答案
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, w[k][i]);
        }
        return ans >= INF / 2 ? -1 : ans;
    }
    void floyd() {
        // floyd 基本流程为三层循环：
        // 枚举中转点 - 枚举起点 - 枚举终点 - 松弛操作
        for (int p = 1; p <= n; p++) {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                    w[i][j] = Math.min(w[i][j], w[i][p] + w[p][j]);
                }
            }
        }
    }
}
```

- 时间复杂度：$O(n^3)$
- 空间复杂度：$O(n^2)$

朴素 Dijkstra（邻接矩阵）

同理，我们可以使用复杂度为 $O(n^2)$ 的「单源最短路」算法朴素 Dijkstra 算法进行求解，同时使用「邻接矩阵」来进行存图。

根据题意，k 点作为源点，跑一遍 Dijkstra 我们可以得到从源点 k 到其他点 x 的最短距离，再从所有最短路中取 max 即是「**从 k 点出发，到其他点 x 的最短距离的最大值**」。

朴素 Dijkstra 复杂度为 $O(n^2)$，可以过。

代码：

```java
class Solution {
    int N = 110, M = 6010;
    // 邻接矩阵数组：w[a][b] = c 代表从 a 到 b 有权重为 c 的边
    int[][] w = new int[N][N];
    // dist[x] = y 代表从「源点/起点」到 x 的最短距离为 y
    int[] dist = new int[N];
    // 记录哪些点已经被更新过
    boolean[] vis = new boolean[N];
    int INF = 0x3f3f3f3f;
    int n, k;
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        // 初始化邻接矩阵
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                w[i][j] = w[j][i] = i == j ? 0 : INF;
            }
        }
        // 存图
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            w[u][v] = c;
        }
        // 最短路
        dijkstra();
        // 遍历答案
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, dist[i]);
        }
        return ans > INF / 2 ? -1 : ans;
    }
    void dijkstra() {
        // 起始先将所有的点标记为「未更新」和「距离为正无穷」
        Arrays.fill(vis, false);
        Arrays.fill(dist, INF);
        // 只有起点最短距离为 0
        dist[k] = 0;
        // 迭代 n 次
        for (int p = 1; p <= n; p++) {
            // 每次找到「最短距离最小」且「未被更新」的点 t
            int t = -1;
            for (int i = 1; i <= n; i++) {
                if (!vis[i] && (t == -1 || dist[i] < dist[t])) t = i;
            }
            // 标记点 t 为已更新
```

```
            vis[t] = true;
            // 用点 t 的「最小距离」更新其他点
            for (int i = 1; i <= n; i++) {
                dist[i] = Math.min(dist[i], dist[t] + w[t][i]);
            }
        }
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

堆优化 Dijkstra（邻接表）

由于边数据范围不算大，我们还可以使用复杂度为 $O(m \log n)$ 的堆优化 Dijkstra 算法进行求解。

堆优化 Dijkstra 算法与朴素 Dijkstra 都是「单源最短路」算法。

跑一遍堆优化 Dijkstra 算法求最短路，再从所有最短路中取 max 即是「**从 k 点出发，到其他点 x 的最短距离的最大值**」。

此时算法复杂度为 $O(m \log n)$，可以过。


执行结果： 通过 显示详情 >
堆优化 Dijkstra
执行用时： 7 ms ，在所有 Java 提交中击败了 84.57% 的用户
内存消耗： 41.5 MB ，在所有 Java 提交中击败了 76.92% 的用户
炫耀一下:
写题解，分享我的解题思路

代码：

```java
class Solution {
    int N = 110, M = 6010;
    // 邻接表
    int[] he = new int[N], e = new int[M], ne = new int[M], w = new int[M];
    // dist[x] = y 代表从「源点/起点」到 x 的最短距离为 y
    int[] dist = new int[N];
    // 记录哪些点已经被更新过
    boolean[] vis = new boolean[N];
    int n, k, idx;
    int INF = 0x3f3f3f3f;
    void add(int a, int b, int c) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx;
        w[idx] = c;
        idx++;
    }
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        // 初始化链表头
        Arrays.fill(he, -1);
        // 存图
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            add(u, v, c);
        }
        // 最短路
        dijkstra();
        // 遍历答案
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, dist[i]);
        }
        return ans > INF / 2 ? -1 : ans;
    }
    void dijkstra() {
        // 起始先将所有的点标记为「未更新」和「距离为正无穷」
        Arrays.fill(vis, false);
        Arrays.fill(dist, INF);
        // 只有起点最短距离为 0
        dist[k] = 0;
        // 使用「优先队列」存储所有可用于更新的点
        // 以 (点编号, 到起点的距离) 进行存储，优先弹出「最短距离」较小的点
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->a[1]-b[1]);
        q.add(new int[]{k, 0});
        while (!q.isEmpty()) {
```

```
            // 每次从「优先队列」中弹出
            int[] poll = q.poll();
            int id = poll[0], step = poll[1];
            // 如果弹出的点被标记「已更新」，则跳过
            if (vis[id]) continue;
            // 标记该点「已更新」，并使用该点更新其他点的「最短距离」
            vis[id] = true;
            for (int i = he[id]; i != -1; i = ne[i]) {
                int j = e[i];
                if (dist[j] > dist[id] + w[i]) {
                    dist[j] = dist[id] + w[i];
                    q.add(new int[]{j, dist[j]});
                }
            }
        }
    }
}
```

- 时间复杂度：$O(m \log n + n)$
- 空间复杂度：$O(m)$

Bellman Ford（类 & 邻接表）

虽然题目规定了不存在「负权边」，但我们仍然可以使用可以在「负权图中求最短路」的 Bellman Ford 进行求解，该算法也是「单源最短路」算法，复杂度为 $O(n * m)$。

通常为了确保 $O(n * m)$，可以单独建一个类代表边，将所有边存入集合中，在 n 次松弛操作中直接对边集合进行遍历（代码见 $P1$）。

由于本题边的数量级大于点的数量级，因此也能够继续使用「邻接表」的方式进行边的遍历，遍历所有边的复杂度的下界为 $O(n)$，上界可以确保不超过 $O(m)$（代码见 $P2$）。


执行结果： 通过 显示详情 ›
Bellman Ford
执行用时： 24 ms ，在所有 Java 提交中击败了 36.54% 的用户
内存消耗： 43.7 MB ，在所有 Java 提交中击败了 5.02% 的用户
炫耀一下:
写题解，分享我的解题思路

代码：

```java
class Solution {
    class Edge {
        int a, b, c;
        Edge(int _a, int _b, int _c) {
            a = _a; b = _b; c = _c;
        }
    }
    int N = 110, M = 6010;
    // dist[x] = y 代表从「源点/起点」到 x 的最短距离为 y
    int[] dist = new int[N];
    int INF = 0x3f3f3f3f;
    int n, m, k;
    // 使用类进行存边
    List<Edge> es = new ArrayList<>();
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        m = ts.length;
        // 存图
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            es.add(new Edge(u, v, c));
        }
        // 最短路
        bf();
        // 遍历答案
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, dist[i]);
        }
        return ans > INF / 2 ? -1 : ans;
    }
    void bf() {
        // 起始先将所有的点标记为「距离为正无穷」
        Arrays.fill(dist, INF);
        // 只有起点最短距离为 0
        dist[k] = 0;
        // 迭代 n 次
        for (int p = 1; p <= n; p++) {
            int[] prev = dist.clone();
            // 每次都使用上一次迭代的结果，执行松弛操作
            for (Edge e : es) {
                int a = e.a, b = e.b, c = e.c;
                dist[b] = Math.min(dist[b], prev[a] + c);
            }
        }
    }
```

```
}
```

```java
class Solution {
    int N = 110, M = 6010;
    // 邻接表
    int[] he = new int[N], e = new int[M], ne = new int[M], w = new int[M];
    // dist[x] = y 代表从「源点/起点」到 x 的最短距离为 y
    int[] dist = new int[N];
    int INF = 0x3f3f3f3f;
    int n, m, k, idx;
    void add(int a, int b, int c) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx;
        w[idx] = c;
        idx++;
    }
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        m = ts.length;
        // 初始化链表头
        Arrays.fill(he, -1);
        // 存图
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            add(u, v, c);
        }
        // 最短路
        bf();
        // 遍历答案
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, dist[i]);
        }
        return ans > INF / 2 ? -1 : ans;
    }
    void bf() {
        // 起始先将所有的点标记为「距离为正无穷」
        Arrays.fill(dist, INF);
        // 只有起点最短距离为 0
        dist[k] = 0;
        // 迭代 n 次
        for (int p = 1; p <= n; p++) {
            int[] prev = dist.clone();
            // 每次都使用上一次迭代的结果，执行松弛操作
            for (int a = 1; a <= n; a++) {
                for (int i = he[a]; i != -1; i = ne[i]) {
                    int b = e[i];
```

```
                        dist[b] = Math.min(dist[b], prev[a] + w[i]);
                    }
                }
            }
        }
}
```

- 时间复杂度：$O(n * m)$
- 空间复杂度：$O(m)$

SPFA（邻接表）

SPFA 是对 Bellman Ford 的优化实现，可以使用队列进行优化，也可以使用栈进行优化。

通常情况下复杂度为 $O(k * m)$，k 一般为 4 到 5，最坏情况下仍为 $O(n * m)$，当数据为网格图时，复杂度会从 $O(k * m)$ 退化为 $O(n * m)$。

代码：

```java
class Solution {
    int N = 110, M = 6010;
    // 邻接表
    int[] he = new int[N], e = new int[M], ne = new int[M], w = new int[M];
    // dist[x] = y 代表从「源点/起点」到 x 的最短距离为 y
    int[] dist = new int[N];
    // 记录哪一个点「已在队列」中
    boolean[] vis = new boolean[N];
    int INF = 0x3f3f3f3f;
    int n, k, idx;
    void add(int a, int b, int c) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx;
        w[idx] = c;
        idx++;
    }
    public int networkDelayTime(int[][] ts, int _n, int _k) {
        n = _n; k = _k;
        // 初始化链表头
        Arrays.fill(he, -1);
        // 存图
        for (int[] t : ts) {
            int u = t[0], v = t[1], c = t[2];
            add(u, v, c);
        }
        // 最短路
        spfa();
        // 遍历答案
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = Math.max(ans, dist[i]);
        }
        return ans > INF / 2 ? -1 : ans;
    }
    void spfa() {
        // 起始先将所有的点标记为「未入队」和「距离为正无穷」
        Arrays.fill(vis, false);
        Arrays.fill(dist, INF);
        // 只有起点最短距离为 0
        dist[k] = 0;
        // 使用「双端队列」存储，存储的是点编号
        Deque<Integer> d = new ArrayDeque<>();
        // 将「源点/起点」进行入队，并标记「已入队」
        d.addLast(k);
        vis[k] = true;
```

```
        while (!d.isEmpty()) {
            // 每次从「双端队列」中取出，并标记「未入队」
            int poll = d.pollFirst();
            vis[poll] = false;
            // 尝试使用该点，更新其他点的最短距离
            // 如果更新的点，本身「未入队」则加入队列中，并标记「已入队」
            for (int i = he[poll]; i != -1; i = ne[i]) {
                int j = e[i];
                if (dist[j] > dist[poll] + w[i]) {
                    dist[j] = dist[poll] + w[i];
                    if (vis[j]) continue;
                    d.addLast(j);
                    vis[j] = true;
                }
            }
        }
    }
}
```

- 时间复杂度：$O(n * m)$
- 空间复杂度：$O(m)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **787. K 站中转内最便宜的航班** ，难度为 **中等**。

Tag : 「最短路」、「Bellman Ford」

有 `n` 个城市通过一些航班连接。给你一个数组 `flights`，其中 $flights[i] = [from_i, to_i, price_i]$ ，表示该航班都从城市 $from_i$ 开始，以价格 $price_i$ 抵达 to_i。

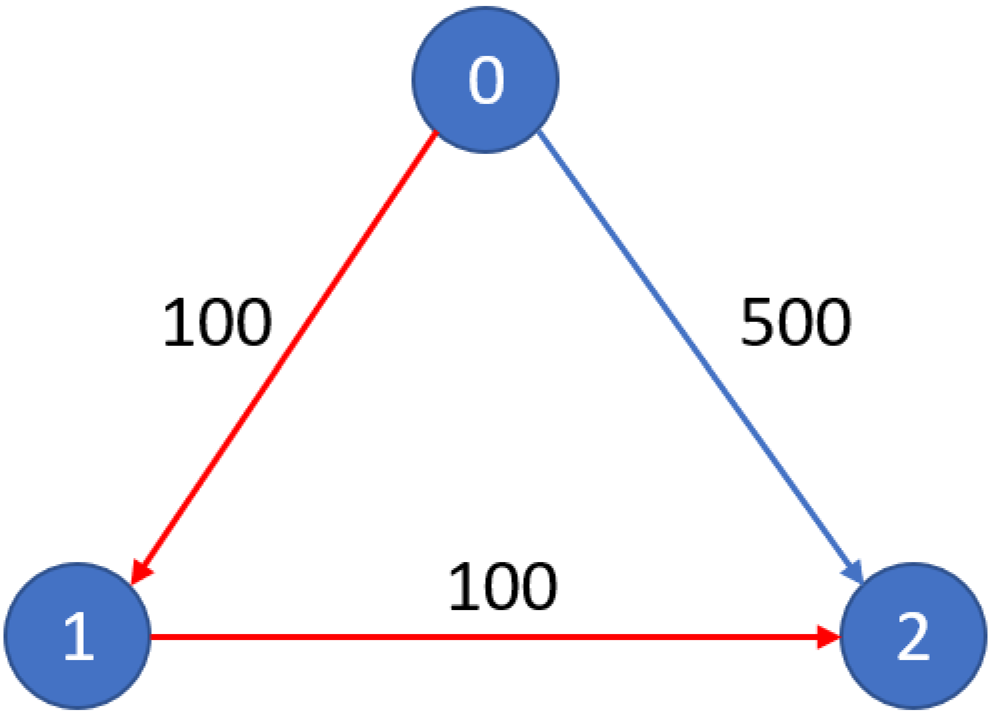
现在给定所有的城市和航班，以及出发城市 `src` 和目的地 `dst`，你的任务是找到出一条最多经过 `k` 站中转的路线，使得从 `src` 到 `dst` 的 价格最便宜 ，并返回该价格。 如果不存在这样的路线，则输出 `-1`。

示例 1：

```
输入:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 1

输出: 200

解释:
城市航班图如下
从城市 0 到城市 2 在 1 站中转以内的最便宜价格是 200，如图中红色所示。
```

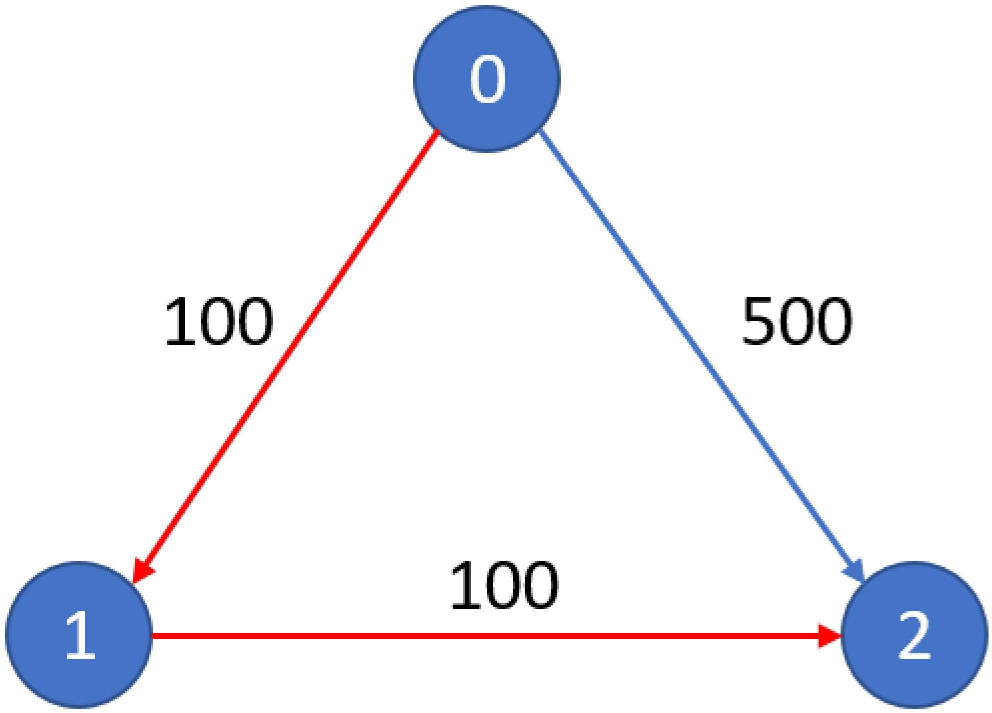


示例 2：

```
输入:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 0

输出: 500

解释:
城市航班图如下
从城市 0 到城市 2 在 0 站中转以内的最便宜价格是 500，如图中蓝色所示。
```



提示：

- 1 <= n <= 100
- 0 <= flights.length <= (n * (n - 1) / 2)
- flights[i].length == 3
- 0 <= fromi, toi < n
- fromi != toi
- 1 <= pricei <= 10^4
- 航班没有重复，且不存在自环
- 0 <= src, dst, k < n
- src != dst

基本分析

从题面看就能知道，这是一类「有限制」的最短路问题。

「限制最多经过不超过 k 个点」等价于「限制最多不超过 $k+1$ 条边」，而解决「有边数限制的最短路问题」是 SPFA 所不能取代 Bellman Ford 算法的经典应用之一（SPFA 能做，但不能直接做）。

Bellman Ford/SPFA 都是基于动态规划，其原始的状态定义为 $f[i][k]$ 代表从起点到 i 点，且经过最多 k 条边的最短路径。这样的状态定义引导我们能够使用 Bellman Ford 来解决有边数限制的最短路问题。

同样多源汇最短路算法 Floyd 也是基于动态规划，其原始的三维状态定义为 $f[i][j][k]$ 代表从点 i 到点 j，且经过的所有点编号不会超过 k（即可使用点编号范围为 $[1, k]$）的最短路径。这样的状态定义引导我们能够使用 Floyd 求最小环或者求"重心点"（即删除该点后，最短路值会变大）。

如果你对几类最短算法不熟悉，可以看 这里，里面涵盖所有的「最短路算法」和「存图方式」。

Bellman Ford + 邻接矩阵

回到本题，「限制最多经过不超过 k 个点」等价于「限制最多不超过 $k+1$ 条边」，因此可以使用 Bellman Ford 来求解。

点的数量只有 100，可以直接使用「邻接矩阵」的方式进行存图。

> 需要注意的是，在遍历所有的"点对/边"进行松弛操作前，需要先对 $dist$ 进行备份，否则会出现「本次松弛操作所使用到的边，也是在同一次迭代所更新的」，从而不满足边数限制的要求。
> 举个🌰，例如本次松弛操作使用了从 a 到 b 的当前最短距离来更新 $dist[b]$，直接使用 $dist[a]$ 的话，不能确保 $dist[a]$ 不是在同一次迭代中所更新，如果 $dist[a]$ 是同一次迭代所更新的话，那么使用的边数将会大于 k 条。
> 因此在每次迭代开始前，我们都应该对 $dist$ 进行备份，在迭代时使用备份来进行松弛操作。


执行结果： 通过 显示详情 >
添加备注
执行用时： 8 ms ，在所有 Java 提交中击败了 38.11% 的用户
内存消耗： 39.5 MB ，在所有 Java 提交中击败了 63.70% 的用户
炫耀一下：
写题解，分享我的解题思路

代码：

```java
class Solution {
    int N = 110, INF = 0x3f3f3f3f;
    int[][] g = new int[N][N];
    int[] dist = new int[N];
    int n, m, s, t, k;
    public int findCheapestPrice(int _n, int[][] flights, int _src, int _dst, int _k) {
        n = _n; s = _src; t = _dst; k = _k + 1;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                g[i][j] = i == j ? 0 : INF;
            }
        }
        for (int[] f : flights) {
            g[f[0]][f[1]] = f[2];
        }
        int ans = bf();
        return ans > INF / 2 ? -1 : ans;
    }
    int bf() {
        Arrays.fill(dist, INF);
        dist[s] = 0;
        for (int limit = 0; limit < k; limit++) {
            int[] clone = dist.clone();
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    dist[j] = Math.min(dist[j], clone[i] + g[i][j]);
                }
            }
        }
        return dist[t];
    }
}
```

- 时间复杂度：$O(k * n^2)$
- 空间复杂度：$O(n^2)$

Bellman Ford + 类

我们知道 Bellman Ford 需要遍历所有的边，而使用「邻接矩阵」的存图方式让我们不得不遍历所有的点对，复杂度为 $O(n^2)$。

而边的数量 m 的数据范围为 $0 <= flights.length <= (n * (n - 1)/2)$，因此我们可以使

用「类」的方式进行存图，从而确保在遍历所有边的时候，复杂度严格为 $O(m)$，而不是 $O(n^2)$。

代码：

```java
class Solution {
    class Edge {
        int x, y, w;
        Edge(int _x, int _y, int _w) {
            x = _x; y = _y; w = _w;
        }
    }
    int N = 110, INF = 0x3f3f3f3f;
    int[] dist = new int[N];
    List<Edge> list = new ArrayList<>();
    int n, m, s, t, k;
    public int findCheapestPrice(int _n, int[][] flights, int _src, int _dst, int _k) {
        n = _n; s = _src; t = _dst; k = _k + 1;
        for (int[] f : flights) {
            list.add(new Edge(f[0], f[1], f[2]));
        }
        m = list.size();
        int ans = bf();
        return ans > INF / 2 ? -1 : ans;
    }
    int bf() {
        Arrays.fill(dist, INF);
        dist[s] = 0;
        for (int i = 0; i < k; i++) {
            int[] clone = dist.clone();
            for (Edge e : list) {
                int x = e.x, y = e.y, w = e.w;
                dist[y] = Math.min(dist[y], clone[x] + w);
            }
        }
        return dist[t];
    }
}
```

- 时间复杂度：共进行 $k+1$ 次迭代，每次迭代备份数组复杂度为 $O(n)$，然后遍历所有的边进行松弛操作，复杂度为 $O(m)$。整体复杂度为 $O(k*(n+m))$
- 空间复杂度：$O(n+m)$

Bellman Ford

更进一步，由于 Bellman Ford 核心操作需要遍历所有的边，因此也可以直接使用 $flights$ 数组作为存图信息，而无须额外存图。

代码：

```
class Solution {
    int N = 110, INF = 0x3f3f3f3f;
    int[] dist = new int[N];
    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
        Arrays.fill(dist, INF);
        dist[src] = 0;
        for (int limit = 0; limit < k + 1; limit++) {
            int[] clone = dist.clone();
            for (int[] f : flights) {
                int x = f[0], y = f[1], w = f[2];
                dist[y] = Math.min(dist[y], clone[x] + w);
            }
        }
        return dist[dst] > INF / 2 ? -1 : dist[dst];
    }
}
```

- 时间复杂度：共进行 $k+1$ 次迭代，每次迭代备份数组复杂度为 $O(n)$，然后遍历所有的边进行松弛操作，复杂度为 $O(m)$。整体复杂度为 $O(k*(n+m))$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1631. 最小体力消耗路径** ，难度为 **中等**。

Tag : 「最小生成树」、「并查集」、「Kruskal」

你准备参加一场远足活动。

给你一个二维 `rows x columns` 的地图 heights ，其中 `heights[row][col]` 表示格子 `(row, col)` 的高度。

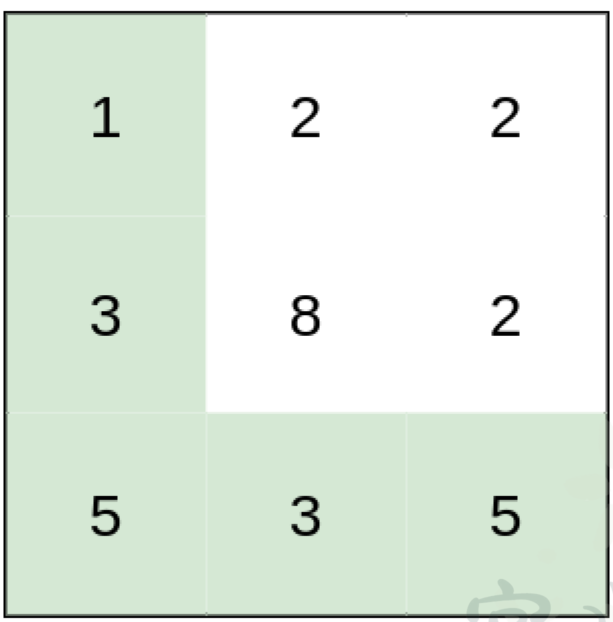
一开始你在最左上角的格子 `(0, 0)` ，且你希望去最右下角的格子 `(rows-1, columns-1)` （注意下标从 0 开始编号）。

你每次可以往 上，下，左，右 四个方向之一移动，你想要找到耗费 体力 最小的一条路径。

一条路径耗费的「体力值」是路径上相邻格子之间「高度差绝对值」的「最大值」决定的。

请你返回从左上角走到右下角的最小 体力消耗值 。

示例 1：



```
输入：heights = [[1,2,2],[3,8,2],[5,3,5]]

输出：2

解释：路径 [1,3,5,3,5] 连续格子的差值绝对值最大为 2 。
这条路径比路径 [1,2,2,2,5] 更优，因为另一条路径差值最大值为 3 。
```

示例 2：

```
输入：heights = [[1,2,3],[3,8,4],[5,3,5]]

输出：1

解释：路径 [1,2,3,4,5] 的相邻格子差值绝对值最大为 1 ，比路径 [1,3,5,3,5] 更优。
```

示例 3：

```
输入：heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]

输出：0

解释：上图所示路径不需要消耗任何体力。
```

提示：

- rows == heights.length
- columns == heights[i].length
- 1 <= rows, columns <= 100
- 1 <= heights[i][j] <= 10^6

基本分析

对于这道题，可能会有同学想这是不是应该用 DP 呀？

特别是接触过「路径问题」但又还没系统学完的同学。

事实上，当题目允许往任意方向移动时，考察的往往就不是 DP 了，而是图论。

从本质上说，DP 问题是一类特殊的图论问题。

那为什么有一些 DP 题目简单修改条件后，就只能彻底转化为图论问题来解决了呢？

这是因为修改条件后，导致我们 DP 状态展开不再是一个拓扑序列，也就是我们的图不再是一个拓扑图。

换句话说，DP 题虽然都属于图论范畴。

但对于不是拓扑图的图论问题，我们无法使用 DP 求解。

而此类看似 DP，实则图论的问题，通常是最小生成树或者最短路问题。

Kruskal

当一道题我们决定往「图论」方向思考时，我们的重点应该放在「如何建图」上。

因为解决某个特定的图论问题（最短路/最小生成树/二分图匹配），我们都是使用特定的算法。

由于使用到的算法都有固定模板，因此编码难度很低，而「如何建图」的思维难度则很高。

对于本题，我们可以按照如下分析进行建图：

因为在任意格子可以往「任意方向」移动，所以相邻的格子之间存在一条无向边。

题目要我们求的就是**从起点到终点的最短路径中，边权最大的值**。

我们可以先遍历所有的格子，将所有的边加入集合。

存储的格式为数组 $[a, b, w]$ ，代表编号为 a 的点和编号为 b 的点之间的权重为 w。

按照题意，w 为两者的高度差的绝对值。

对集合进行排序，按照 w 进行从小到大排序（Kruskal 部分）。

当我们有了所有排好序的候选边集合之后，我们可以对边进行从前往后处理，每次加入一条边之后，使用并查集来查询「起点」和「终点」是否连通（并查集部分）。

当第一次判断「起点」和「终点」联通时，说明我们「最短路径」的所有边都已经应用到并查集上了，而且由于我们的边是按照「从小到大」进行排序，因此最后一条添加的边就是「最短路径」上权重最大的边。

代码：

```java
class Solution {
    int N = 10009;
    int[] p = new int[N];
    int row, col;
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    boolean query(int a, int b) {
        return p[find(a)] == p[find(b)];
    }
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }
    public int minimumEffortPath(int[][] heights) {
        row = heights.length;
        col = heights[0].length;

        // 初始化并查集
        for (int i = 0; i < row * col; i++) p[i] = i;

        // 预处理出所有的边
        // edge 存的是 [a, b, w]：代表从 a 到 b 的体力值为 w
        // 虽然我们可以往四个方向移动，但是只要对于每个点都添加「向右」和「向下」两条边的话，其实就已经覆
        List<int[]> edges = new ArrayList<>();
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                int idx = getIndex(i, j);
                if (i + 1 < row) {
                    int a = idx, b = getIndex(i + 1, j);
                    int w = Math.abs(heights[i][j] - heights[i + 1][j]);
                    edges.add(new int[]{a, b, w});
                }
                if (j + 1 < col) {
                    int a = idx, b = getIndex(i, j + 1);
                    int w = Math.abs(heights[i][j] - heights[i][j + 1]);
                    edges.add(new int[]{a, b, w});
                }
            }
        }

        // 根据权值 w 降序
        Collections.sort(edges, (a,b)->a[2]-b[2]);

        // 从「小边」开始添加，当某一条边别应用之后，恰好使用得「起点」和「结点」联通
        // 那么代表找到了「最短路径」中的「权重最大的边」
```

```java
        int start = getIndex(0, 0), end = getIndex(row - 1, col - 1);
        for (int[] edge : edges) {
            int a = edge[0], b = edge[1], w = edge[2];
            union(a, b);
            if (query(start, end)) {
                return w;
            }
        }
        return 0;
    }
    int getIndex(int x, int y) {
        return x * col  + y;
    }
}
```

令行数为 r，列数为 c，那么节点的数量为 $r * c$，无向边的数量严格为 $r * (c - 1) + c * (r - 1)$，数量级上为 $r * c$。

- 时间复杂度：获取所有的边复杂度为 $O(r * c)$，排序复杂度为 $O((r * c)\log{(r * c)})$，遍历得到最终解复杂度为 $O(r * c)$。整体复杂度为 $O((r * c)\log{(r * c)})$。
- 空间复杂度：使用了并查集数组。复杂度为 $O(r * c)$。

证明

我们之所以能够这么做，是因为「跳出循环前所遍历的最后一条边必然是最优路径上的边，而且是 w 最大的边」。

我们可以用「反证法」来证明这个结论为什么是正确的。

我们先假设「跳出循环前所遍历的最后一条边必然是最优路径上的边，而且是 w 最大的边」不成立：

我们令循环终止前的最后一条边为 a

1. *假设 a 不在最优路径内*：如果 a 并不在最优路径内，即最优路径是由 a 边之前的边构成，那么 a 边不会对左上角和右下角节点的连通性产生影响。也就是在遍历到该边之前，左上角和右下角应该是联通的，逻辑上循环会在遍历到该边前终止。与我们循环的决策逻辑冲突。

2. *a 在最优路径内，但不是 w 最大的边*：我们在遍历之前就已经排好序。与排序逻辑冲突。

因此，我们的结论是正确的。`a` 边必然属于「最短路径」并且是权重最大的边。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1786. 从第一个节点出发到最后一个节点的受限路径数** ，难度为 **中等**。

Tag : 「图论最短路」、「线性 DP」

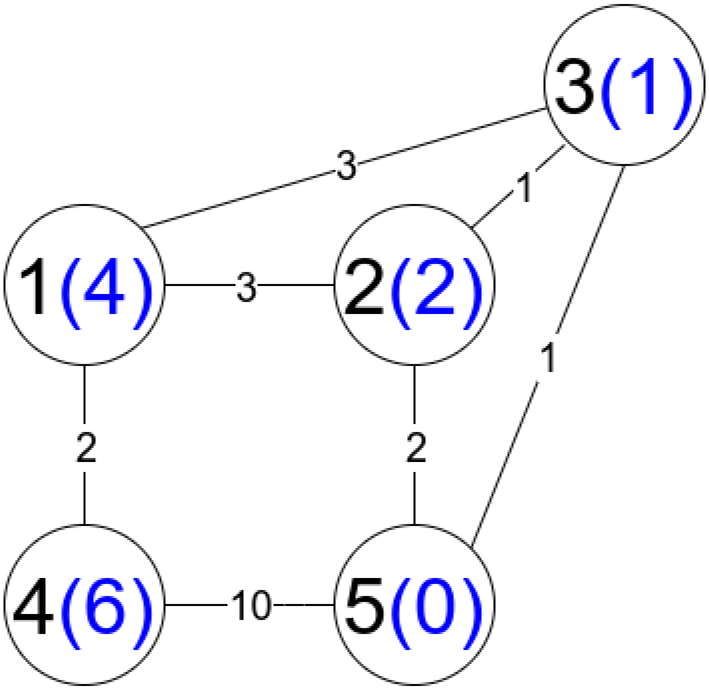
现有一个加权无向连通图。给你一个正整数 n ，表示图中有 n 个节点，并按从 1 到 n 给节点编号；另给你一个数组 edges ，其中每个 edges[i] = [ui, vi, weighti] 表示存在一条位于节点 ui 和 vi 之间的边，这条边的权重为 weighti 。

从节点 start 出发到节点 end 的路径是一个形如 [z0, z1, z2, ..., zk] 的节点序列，满足 z0 = start 、zk = end 且在所有符合 0 <= i <= k-1 的节点 zi 和 zi+1 之间存在一条边。

路径的距离定义为这条路径上所有边的权重总和。用 distanceToLastNode(x) 表示节点 n 和 x 之间路径的最短距离。受限路径 为满足 distanceToLastNode(zi) > distanceToLastNode(zi+1) 的一条路径，其中 0 <= i <= k-1 。

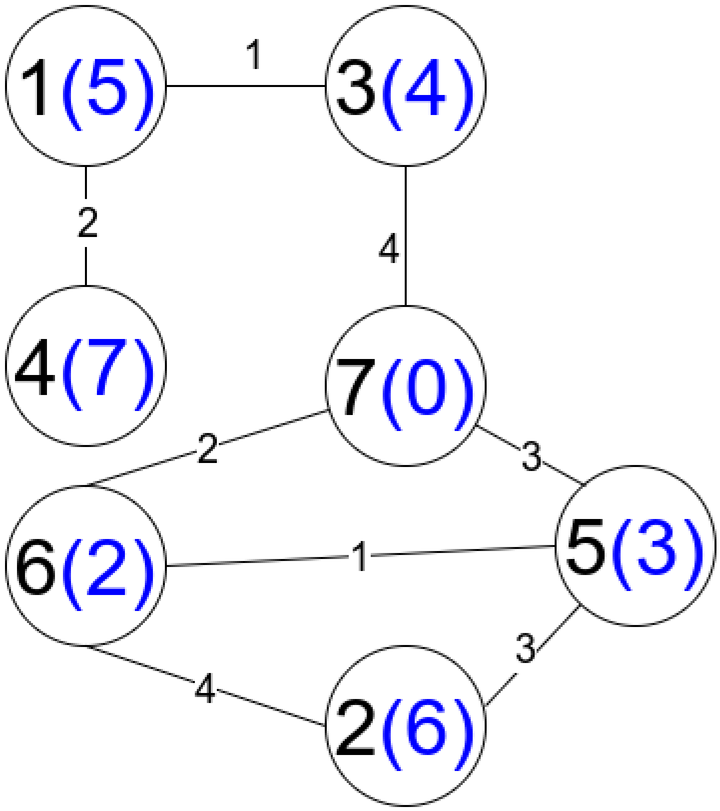
返回从节点 1 出发到节点 n 的 受限路径数 。由于数字可能很大，请返回对 109 + 7 取余 的结果。

示例 1：




```
输入：n = 5, edges = [[1,2,3],[1,3,3],[2,3,1],[1,4,2],[5,2,2],[3,5,1],[5,4,10]]
输出：3
解释：每个圆包含黑色的节点编号和蓝色的 distanceToLastNode 值。三条受限路径分别是：
1) 1 --> 2 --> 5
2) 1 --> 2 --> 3 --> 5
3) 1 --> 3 --> 5
```

示例 2：




```
输入：n = 7, edges = [[1,3,1],[4,1,2],[7,3,4],[2,5,3],[5,6,1],[6,7,2],[7,5,3],[2,6,4]]
输出：1
解释：每个圆包含黑色的节点编号和蓝色的 distanceToLastNode 值。唯一一条受限路径是：1 --> 3 --> 7 。
```

提示：

- 1 <= n <= 2 * 10^4
- n - 1 <= edges.length <= 4 * 10^4
- edges[i].length == 3
- 1 <= ui, vi <= n
- ui != vi
- 1 <= weighti <= 10^5
- 任意两个节点之间至多存在一条边
- 任意两个节点之间至少存在一条路径

堆优化 Dijkstra + 动态规划

n 为点的数量，m 为边的数量。

为了方便理解，我们将第 n 个点称为「起点」，第 1 个点称为「结尾」。

按照题意，我们需要先求每个点到结尾的「最短路」，求最短路的算法有很多，通常根据「有无负权边」&「稠密图还是稀疏图」进行选择。

该题只有正权变，而且"边"和"点"的数量在一个数量级上，属于稀疏图。

因此我们可以采用「最短路」算法：堆优化的 Dijkstra，复杂度为 $O(m\log n)$。

PS. 通常会优先选择 SPFA，SPFA 通常情况下复杂度为 $O(m)$，但最坏情况下复杂度为 $O(n*m)$。从数据上来说 SPFA 也会超，而且本题还结合了 DP，因此可能会卡掉图论部分的 SPFA。出于这些考虑，我直接使用堆优化 Dijkstra。

当我们求得了每个点到结尾的「最短路」之后，接下来我们需要求得从「起点」到「结尾」的**受限路径数量**。

这显然可以用 DP 来做。

我们定义 f(i) 为从第 i 个点到结尾的受限路径数量，f(1) 就是我们的答案，而 f(n) = 1 是一个显而易见的起始条件。

因为题目的**受限路径数**的定义，我们需要找的路径所包含的点，必须是其距离结尾的最短路越来越近的。

举个🌰，对于示例 1，其中一条符合要求的路径为 1 --> 2 --> 3 --> 5。

这条路径的搜索过程可以看做，从结尾（第 5 个点）出发，逆着走，每次选择一个点（例如 a）之后，再选择下一个点（例如 b）时就必须**满足最短路距离比上一个点（点 a）要远**，如果最终能选到起点（第一个点），说明统计出一条有效路径。

我们的搜索方式决定了需要先按照最短路距离进行从小到大排序。

不失一般性，当我们要求 f(i) 的时候，其实找的是 i 点可以到达的点 j，并且 j 点到结尾的最短路要严格小于 i 点到结尾的最短路。

符合条件的点 j 有很多个，将所有的 f(j) 累加即是 f(i)。

代码：

```java
class Solution {
    int mod = 1000000007;
    public int countRestrictedPaths(int n, int[][] es) {
        // 预处理所有的边权。 a b w -> a : { b : w } + b : { a : w }
        Map<Integer, Map<Integer, Integer>> map = new HashMap<>();
        for (int[] e : es) {
            int a = e[0], b = e[1], w = e[2];
            Map<Integer, Integer> am = map.getOrDefault(a, new HashMap<Integer, Integer>()
            am.put(b, w);
            map.put(a, am);
            Map<Integer, Integer> bm = map.getOrDefault(b, new HashMap<Integer, Integer>()
            bm.put(a, w);
            map.put(b, bm);
        }

        // 堆优化 Dijkstra：求 每个点 到 第n个点 的最短路
        int[] dist = new int[n + 1];
        boolean[] st = new boolean[n + 1];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[n] = 0;
        Queue<int[]> q = new PriorityQueue<int[]>((a, b)->a[1]-b[1]); // 点编号，点距离。根据
        q.add(new int[]{n, 0});
        while (!q.isEmpty()) {
            int[] e = q.poll();
            int idx = e[0], cur = e[1];
            if (st[idx]) continue;
            st[idx] = true;
            Map<Integer, Integer> mm = map.get(idx);
            if (mm == null) continue;
            for (int i : mm.keySet()) {
                dist[i] = Math.min(dist[i], dist[idx] + mm.get(i));
                q.add(new int[]{i, dist[i]});
            }
        }

        // dp 过程
        int[][] arr = new int[n][2];
        for (int i = 0; i < n; i++) arr[i] = new int[]{i + 1, dist[i + 1]}; // 点编号，点距
        Arrays.sort(arr, (a, b)->a[1]-b[1]); // 根据点距离从小到大排序

        // 定义 f(i) 为从第 i 个点到结尾的受限路径数量
        // 从 f[n] 递推到 f[1]
        int[] f = new int[n + 1];
        f[n] = 1;
        for (int i = 0; i < n; i++) {
            int idx = arr[i][0], cur = arr[i][1];
```

```java
            Map<Integer, Integer> mm = map.get(idx);
            if (mm == null) continue;
            for (int next : mm.keySet()) {
                if (cur > dist[next]) {
                    f[idx] += f[next];
                    f[idx] %= mod;
                }
            }
            // 第 1 个节点不一定是距离第 n 个节点最远的点，但我们只需要 f[1]，可以直接跳出循环
            if (idx == 1) break;
        }
        return f[1];
    }
}
```

- 时间复杂度：求最短路的复杂度为 $O(m\log n)$，DP 过程坏情况下要扫完所有的边，复杂度为 $O(m)$。整体复杂度为 $O(m\log n)$
- 空间复杂度：$O(n+m)$


“给作者手机充个电”
YOLO 的赞赏码