FDS Project 3 (Hard): Transportation Hub

Anonymous Author

Date: 2026-05-10

1. Introduction

> 题目：P3 Transportation Hub
> 目标：对于给定的源点和终点，找出所有位于不少于 k 条最短路径上的中转城市（不包含源点和终点）。

本项目实现了一个“**双向 Dijkstra + 最短路条数统计**”程序。对于每个查询 $(source, destination)$，程序分别从源点和终点出发做一次 Dijkstra，然后判断某个城市是否满足：

1. 它位于一条源点到终点的最短路径上；
2. 从源点到该城市的最短路径条数，与从该城市到终点的最短路径条数的乘积不小于 k。

满足条件的城市按编号升序输出；若不存在，则输出 `None`。

1.1 该程序的特性

与题目要求相比，本程序采用了较直接但稳定的实现方式：

1. **邻接矩阵存图**：$n \le 500$，矩阵写法简单，便于 Dijkstra 扫描。
2. **每个查询执行两次 Dijkstra**：分别从 source 和 destination 计算最短距离与路径条数。
3. **支持多最短路计数**：不是只求最短距离，而是同时统计最短路条数。
4. **输出时自然有序**：由于按城市编号从小到大扫描，输出本身就是升序。

1.2 补充说明

1.2.1 为什么选择邻接矩阵

本题点数上限只有 500，邻接矩阵的实现非常直接：

1. 查找是否存在边是 $O(1)$；
2. 枚举邻接点时逻辑简单；
3. 不需要额外维护复杂的结构。

代价是每轮 Dijkstra 都要扫描整个点集，空间复杂度是$O(n^2)$，但在本题规模下问题不大。

1.2.2 最短路条数的处理

最开始我直接把 `Count` 写成了 `int`。这个实现能过普通数据，但在较大样本下(多个“菱形”结构，即有两个不同分支回到一个节点)，最短路径条数会指数级增长，很快溢出。

此时程序本来应该输出大量 hub，但实际却输出了 `None`。原因不是最短路算错，而是路径条数在溢出后变成了错误的小值，导致阈值判断失败。

最后的修复方式是：

1. 把 `Count[]` 改成 `long long`；
2. 在松弛时对 `Count[W]` 做截断，超过 `k` 就直接压到 `k`。

这样既能避免溢出，也不会影响题目判断，因为我们只关心是否不少于 k。

1.2.3 为什么不使用更复杂的计数方式

题目只要求判断“是否不少于 k”，而 $k \le 5$。所以最合理的做法其实不是精确保存所有路径数，而是保留到 k 就足够了。

这也是最终采用“`long long` + 截断”的原因：实现简单、速度较快。

2. Algorithm Specification

2.1 主要数据结构

2.1.1 图

1. 邻接矩阵 `G[MAXN][MAXN]`。
2. `G[i][j] = INF` 表示无边，`G[i][i] = 0`。

2.1.2 Dijkstra 相关数组

1. `Dist[]`：源点到各点的最短距离。
2. `Known[]`：是否已经确定最短距离。
3. `Path[]`：记录前驱点，主要用于调试和辅助理解。
4. `Count[]`：记录最短路径条数。

2.2 算法 A：单源最短路 + 条数统计

> 输入：起点 `S`。
> 输出：`Dist[]` 与 `Count[]`。

```
初始化 Dist 为 INF，Known 为 0，Count 为 0
Dist[S] = 0, Count[S] = 1
重复：
  选出当前未确定且 Dist 最小的点 V
  若不存在则结束
  标记 V 已确定
  枚举 V 的每个邻接点 W：
    若 Dist[V] + G[V][W] < Dist[W]：
      更新最短距离，Count[W] = Count[V]
    若 Dist[V] + G[V][W] == Dist[W]：
      累加路径条数，Count[W] += Count[V]
```

这里的关键点是：Dijkstra 不仅维护最短距离，还维护最短路径条数。由于题目只关心阈值 k，因此条数没有必要无限增大，后面会做截断处理。

2.3 算法 B：交通 hub 判定

> 输入：`source, destination`。
> 输出：所有满足条件的 hub。

```
对 source 做一次 Dijkstra，得到 Dist1 和 Count1
对 destination 做一次 Dijkstra，得到 Dist2 和 Count2

若 destination 不可达，直接输出 None

枚举每个城市 j：
  若 j 不是 source / destination
  且 Dist1[j] + Dist2[j] == Dist1[destination] // j在最短路径上
  且 Count1[j] * Count2[j] >= k // j是交通枢纽
    则 j 是 hub
```

这个判定的含义是：如果一个点位于某条最短路径上，那么从 source 到 destination 的最短路径可以分解为两段；两段路径数的乘积就是“经过该点的最短路径条数”。

2.4 复杂度分析

设城市数为 n，边数为 m，查询数为 T。

1. 单次 Dijkstra 使用邻接矩阵，复杂度约为 $O(n^2)$。
2. 每个查询要做两次 Dijkstra，因此单次查询约为 $O(n^2)$。
3. 总复杂度约为 $O(Tn^2)$。

由于题目给定 $n \le 500$，并且 $T \le 500$，这种实现虽然不是最优的堆优化版本，但在本题约束下是可接受的。

3. Testing Results

3.1 测试环境

- 编译：`make` 或直接 `gcc project3.c -o project3`
- 运行：输入一组图和若干查询

3.2 测试用例

> 测试样例在项目文件夹中以文件的形式给出，为了节省篇幅此处省略具体数据。

3.2.1 T1：基础连通图

特点：只有少量最短路径，验证基本输出格式和 hub 判定。

3.2.2 T2：多最短路径图

特点：构造多个等长路径，验证路径计数逻辑。

3.2.3 T3：不可达与自环查询

特点：验证 `None` 输出、`source == destination` 的特殊情况。

3.2.4 T4：大规模例子

特点：`n = 500`，多层等长路径图，最短路条数多，用来卡 `Count` 溢出。

这个测试最开始把程序打出了问题：`int` 版本在大例子下把正确答案算成了 `None`。改成 `long long` 并做**阈值截断**后，输出恢复正常。

3.3 测试小结

1. 基础样例、大规模样例可以正确输出。
2. 多最短路径情况下，hub 判定正确。
3. 自身到自身、不可达情况都会输出 `None`。

4. Analysis and Comments

4.1 时间复杂度

1. 单次 Dijkstra 的复杂度是 $O(n^2)$。
2. 每个查询需要两次 Dijkstra，因此是 $O(n^2)$。
3. 总体为 $O(Tn^2)$。

4.2 空间复杂度

1. 邻接矩阵占用 $O(n^2)$ 空间。
2. 其余辅助数组是 $O(n)$。
3. 总空间复杂度由邻接矩阵主导，为 $O(n^2)$。

4.3 不足与改进

1. 不足之处：
   (a) 当前实现使用邻接矩阵，空间上比较固定，对稀疏图不友好。
   (b) 每个查询都重新跑两次 Dijkstra，重复计算较多。
2. 改进方向：
   (a) 如果想进一步提速，可以改成 `邻接表 + 优先队列`。
   (b) 如果查询很多，可以考虑缓存部分结果。

5. Appendix: Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define INF 0x7FFFFFFF
#define MAXN 505

int G[MAXN][MAXN];     // 邻接矩阵存图
int n, m, k;
// Dijkstra算法
void Dijkstra(int S, int *Dist, int *Known, int *Path, long long
*Count){
```

```
    int V, W;
    for (V = 0; V < n; V++){// 初始化
        Dist[V] = INF;
        Known[V] = 0;
        Path[V] = 0;
        Count[V] = 0;
    }
    Dist[S] = 0;// 初始化源点，距离为0，路径数为1
    Count[S] = 1;
    while(1){
        V = -1;
        int minDist = INF;
        for (int i = 0; i < n; i++){
            // 可以考虑用优先队列优化，但这里n较小，用数组简单实现
            if (!Known[i] && Dist[i] < minDist){
                // 找到未确定的距离最小的点
                minDist = Dist[i];
                V = i;
            }
        }
        // 没有可访问的点了，所有的点均已确定，结束
        if (V == -1) break;
        // 标记V为已访问
        Known[V] = 1;
        // 对V的每个邻接点W进行松弛操作
        for (W = 0; W < n; W++){
            // W是V的邻接点并且W未被固定
            if (G[V][W] < INF && !Known[W]){
                // 松弛成功，更新Dist[W]和Path[W]
                if (Dist[V] + G[V][W] < Dist[W]){
                    Dist[W] = Dist[V] + G[V][W];
                    Path[W] = V;
                    Count[W] = Count[V];
                    // 只需要知道路径数是否达到k，超过k就截断
                    if (Count[W] > k) Count[W] = k;
                }
                // 松弛成功，更新路径数
                else if (Dist[V] + G[V][W] == Dist[W]){
                    Path[W] = V;
                    Count[W] += Count[V];
                    if (Count[W] > k) Count[W] = k;
                }
            }
        }
    }
}
int main() {
    scanf("%d%d%d", &n, &m, &k);
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            if(i == j) G[i][j] = 0;
```

```
            else G[i][j] = INF;
        }
    }
    for (int i = 0; i < m; i++){
        int a, b, l;
        scanf("%d%d%d", &a, &b, &l);
        G[a][b] = G[b][a] = l;
    }
    int T;
    scanf("%d", &T);
    for(int i = 0; i < T; i++){
        int source, destination;
        scanf("%d%d", &source, &destination);
        // Dist: 从source出发到各点的最短距离
        // Known: 该点的最短距离是否已确定
        // Path: 最短路径上该点的前一个点
        // Count: 从source出发到该点的最短路径数
        int Dist1[MAXN], Known1[MAXN], Path1[MAXN];
        int Dist2[MAXN], Known2[MAXN], Path2[MAXN];
        long long Count1[MAXN], Count2[MAXN];
        // 运行两次Dijkstra, 一次从source出发, 一次从destination出发
        Dijkstra(source, Dist1, Known1, Path1, Count1);
        Dijkstra(destination, Dist2, Known2, Path2, Count2);

        if (Dist1[destination] == INF || source == destination) {
            printf("None\n");
            continue;
        }
        int flag = 0, found = 0;
        for(int j = 0; j < n; j++){
            // 该点在最短路径上, 且路径数不小于k
            if (Dist1[j] + Dist2[j] == Dist1[destination]
                && Count1[j] * Count2[j] >= (long long)k
                && j != source && j != destination) {
                if(flag == 0){
                    printf("%d", j);
                    flag = 1;
                }
                else printf(" %d", j);
                found = 1;
            }
        }
        if (!found) printf("None\n");
        else printf("\n");
    }
    return 0;
}
```

Declaration

> I hereby declare that all the work done in this project titled **Transportation Hub** is of my independent effort.