最短路

dbywsc

2025/8

目录

最短路是非常常见的图论问题，分为**单源最短路径**和**多源最短路径**

本节中的一些约定:

n 为图中点的数量， m 为图中边的数量。

s 为最短路的源点。

$w(u, v)$ 为边 (u, v) 的边权。

$dis(u, v)$ 为点 u 到 v 最短路距离。常用的最短路算法有三种，本节将一一介绍。

Floyd 是用来求任意两个节点直接的最短路算法。
适用于有向图、无向图、负权图，但是前提是最短路必须存在[1]。
Floyd 的实现使用了动态规划的思想，设 $f_{k,x,y}$ 为只允许经过点 1 到 k （其中不一定包括 x 和 y)，节点 x 到 y 之间到最短路长度。
考虑如何状态转移，可以发现一共有两种情况，第一种情况是 $f_{k,x,y}$ 直接由 $f_{k-1,x,y}$ 转移过来，也就是不走 k 这个点；第二种情况是 $f_{k,x,y}$ 从 $f_{k-1,x,k}+f_{k-1,k,y}$ 转移过来，也就是先从 x 走到 k ，再从 k 走到 y ，我们每次在两者之间取一个最小值就好了。因此状态转移方程为：

$$f_{k,x,y}=\min(f_{k-1,x,y},f_{k-1,x,k}+f_{k-1,k,y})$$

接下来考虑边界，由于要求最小值，所以要初始化为最大值，但是当 $x=y$ 时，显然应该将距离初始化为 0 ；当存在一条权为 w 的边 $x\to y$ 时，要让 $f_{0,x,y}=w$。

[1]最短路存在的前提是不能有负环

```
void solve(void) {
    memset(f, 0x3f, sizeof(f));
    int n, m; std::cin >> n >> m;
    for(int i = 1; i <= m; i++) {
        int u, v, w; std::cin >> u >> v >> w;
        G[u][v] = G[v][u] = (G[u][v] ? std::min(G[u][v], w) : w);
        f[0][u][v] = f[0][v][u] = std::min(f[0][u][v], w);
    }
    for(int k = 0; k <= n; k++)
    for(int i = 1; i <= n; i++) f[k][i][i] = 0;
    for(int k = 1; k <= n; k++)
        for(int x = 1; x <= n; x++)
            for(int y = 1; y <= n; y++)
                f[k][x][y] = std::min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1]
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++)
            std::cout << f[n][i][j] << " ";
        std::cout << "\n";
    }
}
```

floyd 算法通常搭配邻接矩阵实现。上面的代码中，时空复杂度均为 $O(n^3)$

考虑用滚动数组的方式优化，可以发现 f 的第一维对结果没有影响，因此可以省去第一维：

```
for(int k = 1; k <= n; k++) {
    for(int x = 1; x <= n; x++) {
        for(int y = 1; y <= n; y++) {
            f[x][y] = std::min(f[x][y], f[x][k] + f[k][y]);
        }
    }
}
```

此时的时间复杂度为 $O(n^3)$ ，空间复杂度为 $O(n^2)$ 。

B3647 【模板】Floyd

由于 floyd 的时空复杂度都很高，所以它只能处理点数比较小的图。此外，floyd 常被用来求多源最短路、找最小环、封闭传包等问题。

Dijkstra 是由 Dijkstra[2]发明的，一种用于求解**非负权图**上单源最短路径的算法。
Dijkstra 的流程如下：
将图上所有的点分为两个点集：已经确定最短路的集合 S 和还没有确定的集合 T ，初始时，所有的点都属于 T 。
初始化 $dis(s) = 0$ ，其他为极大值。
然后重复地从 T 中选取一个最短路长度最小的节点，移到 S 中，并且对被刚刚加入 S 的节点的所有出边进行**松弛**操作，直到 T 为空。

[2] E.W.Dijkstra

松弛操作，即对于一条边 (u, v) ，执行下面的式子：
$dis(v) = \min(dis(v), dis(u) + w(u, v))$ 。
这样做的含义即尝试用 $S \to u \to v$ （保证 $S \to u$ 已经是最短路）这条路径更新 v 点的最短路，如果存在更优的方案，就进行更新。

可以发现，Dijkstra 的实现瓶颈主要在于“寻找最短路中长度最小的点”。朴素的做法是在点集 T 中通过枚举的方式暴力地找最小点，时间复杂度为 $O(n^2)$ ，这个效率我们无法接受[3]，因此在本节中，我们不会介绍朴素做法的实现。
考虑优化，可以发现，如果使用堆（即 C++ 中的优先队列）来存放点集 T ，可以保证堆顶一定最短路长度最小的点，此时我们将时间复杂度优化到了 $O(\log n)$ 。
而“对 S 的所有出边进行松弛操作”的时间复杂度为 $O(m)$ ，因此堆优化后的 $Dijkstra$ 时间复杂度为 $O(m \log n)$ 。

[3]事实上，朴素的 Dijkstra 在稠密图中表现更优，但是我们一般不考虑

P4779 【模板】单源最短路径（标准版）

```
void dijkstra(void) {
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;
    std::priority_queue<PII, std::vector<PII>, std::greater<PII> > q;
    q.push({0, s}); //由于 pair 默认按照 first 进行排序，所以我们先 dis 再 v
    while(q.size()) {
        auto u = q.top().y; q.pop();
        if(st[u]) continue;
        st[u] = 1;
        for(auto ed : G[u]) {
            int v = ed.x, w = ed.y;
            if(dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                q.push({dis[v], v});
            }
        }
    }
}
```

可以发现，dijkstra 本质上是一个特殊的 BFS。直到现在，dijkstra 仍然是最优秀的单源最短路算法，所以在处理最短路问题时，绝大部分问题都应该用 dijkstra 解决。
但是，dijkstra 无法处理负权图，此时应该使用别的算法。

SPFA 是 Bellman–Ford 算法的一种实现[4]，在 CNOI/CPC 中，选手们更热衷于称其为 “SPFA”。
SPFA 同样是依赖于松弛操作的一种最短路算法，特点是能够处理负权图和判断是否存在最短路。
SPFA 的流程如下：将 s 入队。
当队列不为空的时候，重复地对队首的点的出边进行松弛操作，同时将枚举到的新点入队。
直到队列为空。
最坏情况下，SPFA 的时间复杂度是 O(nm)。
与 Dijkstra 不同的是，SPFA 能够处理负权边，这就导致遇到负环时，SPFA 会陷入死循环中，这个时候我们可以记录一下，如果 SPFA 已经松弛了 m 条边后还在循环中，就可以确定遇到了负环，直接退出即可。

[4]严格意义上来说，SPFA 是朴素 Bellman-Ford 的队列优化，SPFA 即 Shortest Path Faster Algorithm

P3371 【模板】单源最短路径（弱化版）

```
void spfa(void) {
    memset(dis, 0x3f, sizeof(dis));
    memset(cnt, 0, sizeof(cnt));
    std::queue<int> q;
    st[s] = 1; q.push(s); dis[s] = 0;
    while(q.size()) {
        auto u = q.front(); q.pop();
        st[u] = 0;
        for(auto ed : G[u]) {
            if(dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                cnt[v] = cnt[u] + 1;
                if(cnt[v] >= n) return;
                if(!st[v]) {
                    st[v] = 1; q.push(v);
                }
            }
        }
    }
}
```

可以发现，SPFA 其实也是一个特殊的 BFS。

关于 SPFA

它死了。[5]

SPFA 算法本身有许多潜在的问题，很容易被卡掉。因此在现代算法竞赛中，大部分非负权图的最短路问题都会卡 SPFA。因此，在能够使用 Dijkstra 时务必要使用 Dijkstra。当遇到负权图、存在负环的图、或者差分约束等 Dijkstra 无法解决的问题时，再考虑用 SPFA 解决。

[5] 出自 NOI 2018 Day1

由于最短路问题有非常多的扩展，因此本节的习题除了最短路的基础问题外，还有一些拓展性的问题。

P2910 [USACO08OPEN] Clear And Present Danger S
P3905 道路重建
P1144 最短路计数
P2136 拉近距离
P2446 [SDOI2010] 大陆争霸
P6175 无向图的最小环问题
P5837 [USACO19DEC] Milk Pumping G
P3385 【模板】负环
P5960 【模板】差分约束