E Email Loss

> 21351023 李永琦

题目描述

现在有 n 个通信节点，想要从源点向其余 $n-1$ 个点发送电子邮件。同时给出 m 条光缆路线 (x_i, y_i, t_i) 表示在点 x_i 与 y_i 之间有一条光缆连接，两点之间相互发送电子邮件所需的时间为 t_i。

现给出发送电子邮件的源点 s，请你求出从该点发送的电子邮件至少需要多少时间到达其余 $n-1$ 个点，我们认为所需时间超过 t_0 即是不可接受的，请你输出所有所需时间不可接受的通信节点接收到电子邮件所需的时间，若永远接收不到电子邮件，则输出时间为 -1。

形式化地说：你需要求出**无向图**中一个源点到其余点的最短距离，并判断这个最短距离是否可接受。

输入

第一行四个正整数 $n, m, s, t_0 (1 \leq n, m \leq 10^5, 1 \leq s \leq n, 1 \leq t0 \leq 10^{18})$，分别表示通信节点的数量、通信节点之间的光缆连接数、发送电子邮件的源点和最大的可接受时间。

接下来 m 行：每行三个整数 $x_i, y_i, t_i (1 \leq x_i, y_i \leq n, 0 \leq t_i \leq 10^9)$ 表示在通信节点 x_i 与 y_i 之间有一条光缆，两者相互发送电子邮件所需的时间为 t_i。

输出

对于每组数据，输出一行一个正整数 k 表示有 k 个节点（k 可能为 0）接收到电子邮件所需的时间是不可接受的。

接下来 k 行，每行两个整数 x_i 和 t_i，分别表示所需时间不可接受的通信节点序号和所需的时间，若 x_i 完全无法收到电子邮件，输出的 t_i 为 -1。输出要求**按照通信节点的编号排序**。

题目分析

本题中，只需读图并对源点计算各点最短路，并遍历找出距离大于 t_0 的点即可。

单源最短路问题，主要有 Dijkstra 和 Bellman-Ford 算法，区别如下：

单源最短路算法	**Dijkstra**	**Bellman-Ford**
作用于	非负权图	任意图
能否检测负环	不能	能
时间复杂度	$O(M \log M)$	$O(NM)$

本题中的图没有负权，可以使用 Dijkstra 算法。算法的过程如下：

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。初始化 $dis(s) = 0$，其他点的 dis 均为 $+\infty$。然后重复这些操作：

1. 从 T 集合中，选取一个最短路长度最小的结点 u，移到 S 集合中。
2. 对 u 的所有出边执行松弛操作：对每一出边 $\langle u, v \rangle$，权重为 w，更新 $dis(v)$，$dis(v) = \min(dis(v), dis(u) + w)$。

直到 T 集合为空，算法结束。

选取未确定最短路长度的点集中 dis 最小的点时，主要有暴力和优先队列两种方法，推荐使用优先队列来维护。

图的存储

一般情形下，图的存储采用邻接表非常方便。使用 C++ 的 `vector` 动态数组，可以方便地进行存储和遍历。常见的存储和遍历方法如下，`g` 是一个嵌套的 `vector`，`g[u]` 存储了以 `u` 为起点的所有边及其权重构成的 `pair`：

```
vector<vector<pair<int, long long>>> g;
void add(int u, int v, long long w)
{
    g[u].emplace_back(v, w);
}
// 遍历 u 的出边
for (auto [v, w] : g[u]) { /* */ }
```

还有一种名为“链式前向星”的存储方式。上一种方式不使用 `vector` 难以实现，而这种方式适合用 C 语言实现。下面的代码声明了这些变量：

- `head`：`head[u]` 表示以 `u` 为起点的第一条边；
- `nxt`：`nxt[i]` 表示遍历至编号为 `i` 的边之后，下一条边的编号；
- `to`：`to[i]` 表示第 `i` 号边的终点；
- `weights`：`weights[i]` 表示第 `i` 号边的权重；
- `cnt`：当前已分配的最大编号。

每次添加边时，首先为该边获取一个编号（`++cnt`），然后使用链表头插法的思想来更新起点的出边：获取当前起点的第一个出边编号，让当前边指向该边，然后把起点的第一个出边改为自己。最后存储其终点和权重，完成加边操作。

```
#define N 200005
int head[N], nxt[N], to[N], weights[N], cnt = -1;
// head[u] 的初始值为 -1
void add(int u, int v, int w)
{
    nxt[++cnt] = head[u]; // 当前边的后继
    head[u] = cnt;        // 起点 u 的第一条边
    to[cnt] = v;          // 当前边的终点
    weights[cnt] = w;     // 当前边的权重
}
// 遍历 u 的出边
for (int i = head[u]; ~i; i = nxt[i])
{ // ~i 表示 i != -1
    int v = to[i];
    int w = weights[i];
}
```

使用 C++ 的 `vector` 也可实现该存储方式，但不需维护 `cnt` 变量，代码略去。

Dijkstra 算法关键代码

代码如下，用 `vis` 标记是否已确定最短距离。将源点的 dis 设为 0 后加入优先队列，将松弛过的边的终点加入优先队列中。优先队列的队头对应 dis 最小的点。

```
void solve(int s)
{
    vector<bool> vis(n);
    priority_queue<pair<long long, int>> q;
    dis[s] = 0;
    q.push({0, s});
    while (!q.empty())
    {
        int u = q.top().second;
        q.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (auto [v, w] : g[u])
        {
            if (dis[v] > dis[u] + w)
            {
                dis[v] = dis[u] + w;
                q.push({-dis[v], v});
            }
        }
    }
}
```

语法说明

程序中有一些语法细节需注意。

`std::priority_queue`

`<queue>` 头文件中提供了优先队列的实现，有 3 个模板参数：

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

其中，第一个参数必须提供，后两个具有默认值。第二个参数指定所用容器，`vector`、`deque` 均可；第三个参数默认为二元谓词 `less`，函数 `operator()` 返回第一个参数是否小于第二个参数，因此小的元素的优先级高。但标准规定 “come before” 的元素 “output last”，所以这时实现的是一个**大根堆**。

```
constexpr bool operator()(const T& lhs, const T& rhs) const
{
    return lhs < rhs; // assumes that the implementation uses a flat address
space
}
```

`pair<T1, T2>` 中重载了 `<` 运算符，标准规定先比较 `first`，相同时再比较 `second`，因此本题中需要把 `dis` 放在 `first` 的位置。

```
  template<typename _T1, typename _T2>
    inline _GLIBCXX_CONSTEXPR bool
    operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    { return __x.first < __y.first
      || (!(__y.first < __x.first) && __x.second < __y.second); }
```

本题需要的是小根堆，标程的实现方式是使用默认的比较函数，但加入队列的元素中将距离取了相反数，这样也达到了小根堆的效果。

```
priority_queue<pair<long long, int>> q;
// ...
q.push({-dis[v], v});
```

也可以更改 `priority_queue` 的第三个参数。`<functional>` 头文件提供了二元谓词 `greater`，与 `less` 相反，使用它就可以实现小根堆：

```
priority_queue<pair<long long, long long>, vector<pair<long long, long long>>,
greater<pair<long long, long long>>> q;
// ...
q.push({dis[v], v});
```

如果需要自定义排序规则，则需编写一个谓词，这里略去。

C++ 中的 class

标程中使用了 OOP 的范式，定义了一个 `Dijkstra` 类，成员变量和方法为 `public`。C++ 中类的权限修饰符有 `private`、`protected`、`public` 三种，默认为 `private`，此时类外不可访问，外部需要访问的成员变量和方法须显式声明 `public`：

```
class Dijstra
{
  private:
    int n;
    vector<vector<pair<int, long long>>> g;

  public:
    vector<long long> dis;
    Dijstra(int _n) : n(_n) {
        g.resize(n);
        dis.resize(n, inf);
    }
    void add(int u, int v, long long w) { /* */ }
    void solve(int s) { /* */ }
};
```

构造函数中使用了成员初始化列表（member initializer lists）的语法，这里 `: n(_n)` 相当于 `this->n = _n`。

与 C 不同，C++ 的 struct 可以定义方法，且默认权限为 `public`；结构变量的类型定义不再需要写 `struct`，只需使用结构名即可。因此做算法题目时把 `class` 改成 `struct` 也是可以的。

结构化绑定

标程中取出出边和权重的代码如下：

```
for (auto e : g[u])
{
 int v = e.first;
 long long w = e.second;
}
```

C++17 中引入了结构化绑定（structured binding）语法，因此可以写出如下代码：

```
for (auto [v, w] : g[u]) { /* */ }
```

虽然评测命令指定语言版本为 C++11，但亲测高版本 g++ 指定 `-std=c++11` 时可正常编译，不过会产生警告。

代码

C++

```
#include <functional>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
const long long inf = 2e18;
class Dijstra
{
  public:
    int n;
    vector<vector<pair<int, long long>>> g;
    vector<long long> dis;
    Dijstra(int _n) : n(_n)
    {
        g.resize(n);
        dis.resize(n, inf);
    }
    void add(int u, int v, long long w)
    {
        g[u].emplace_back(v, w);
    }
    void solve(int s)
    {
        vector<bool> vis(n);
        priority_queue<pair<long long, int>, vector<pair<long long, int>>,
greater<pair<long long, int>>> q;
        dis[s] = 0;
        q.push({0, s});
        while (!q.empty())
```

```
        {
            int u = q.top().second;
            q.pop();
            if (vis[u])
                continue;
            vis[u] = true;
            for (auto [v, w] : g[u])
            {
                if (dis[v] > dis[u] + w)
                {
                    dis[v] = dis[u] + w;
                    q.push({dis[v], v});
                }
            }
        }
    }
};
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m, s;
    long long t;
    cin >> n >> m >> s >> t;
    --s;
    Dijstra d(n);
    for (int i = 0; i < m; i++)
    {
        int x, y, w;
        cin >> x >> y >> w;
        --x;
        --y;
        d.add(x, y, w);
        d.add(y, x, w);
    }
    d.solve(s);
    vector<int> ans;
    for (int i = 0; i < n; i++)
    {
        if (d.dis[i] > t)
        {
            ans.push_back(i);
        }
    }
    cout << ans.size() << '\n';
    for (auto i : ans)
    {
        cout << i + 1 << ' ' << (d.dis[i] == inf ? -1 : d.dis[i]) << '\n';
    }
    return 0;
}
```