31 · El secuestro de Petunia

2/11 10:48 (81548)

TLE
~~AC~~

MARP48 | JIANUO WEN HU

Nota:

Secuestro_petunia.cpp

```
/*
 Escribe aquí un comentario general sobre la solución, explicando cómo
 se resuelve el problema y cuál es el coste de la solución, en función
 del tamaño del problema.

 Para cada pueblo con un borriquin no conectados por carreteras se hace lo siguiente:
 Se parte de ese pueblo y se va comprobando el tiempo tardado hasta llegar al otro.
 Si se llega a un borriquin se resetea el tiempo. Se cuenta el numero de pueblos a los que se
 ↪ pueden llegar sin superar el
 tiempo limite. Una vez que se ha llegado a un borriquin se indica que se ha visitado ese
 ↪ pueblo para evitar recorridos infinitos.
 A su vez, se marcan los pueblos a los que se pueden llegar en el tiempo acordado para evitar
 ↪ contar el mismo pueblo multiples veces

 Coste total: B * P * log C
 B el numero de pueblos con borriquin no conectados por carreteras
 C el numero de carreteras
 P el numero de pueblos
*/

template <typename Valor>
class Dijkstra {
public:
    Dijkstra(DigrafoValorado<Valor> const& g, int orig, int d, vector<bool>& borriquines) :
    ↪ origen(orig),
        dist(g.V(), INF), ulti(g.V()), pq(g.V()), nPueblos(0), distanciaMax(d),
        ↪ pMarcado(g.V(), false) {
        dist[origen] = 0;
        pq.push(origen, 0);
        if(borriquines[origen])
            ++nPueblos;
        // V * aristas adys de V * log(V) = A * log(V)
        while (!pq.empty()) { //V
            int v = pq.top().elem; pq.pop();
            // aristas adys de V * log(V)
            for (auto a : g.ady(v)) //aristas adyacentes de V
                relajar(a, borriquines);
            if (borriquines[v])
                borriquines[v] = false;
        }
    }
    bool hayCamino(int v) const { return dist[v] != INF; }
    Valor distancia(int v) const { return dist[v]; }
    int pueblosCercanos() { return nPueblos; }
    deque<AristaDirigida<Valor>> camino(int v) const {
        deque<AristaDirigida<Valor>> cam;
        // recuperamos el camino retrocediendo
        AristaDirigida<Valor> a;
        for (a = ulti[v]; a.desde() != origen; a = ulti[a.desde()])
            cam.push_front(a);
        cam.push_front(a);
        return cam;
```

```
    }
private:
    const Valor INF = std::numeric_limits<Valor>::max();
    int origen;
    std::vector<Valor> dist;
    std::vector<AristaDirigida<Valor>> ulti;
    IndexPQ<Valor> pq;
    int nPueblos;
    int distanciaMax;
    vector<bool> pMarcado; //pueblos donde se ha comprobado si hay borriquin
    void relajar(AristaDirigida<Valor> a, vector<bool>& borriquines) {
        int v = a.desde(), w = a.hasta();
        int valor = dist[v] + a.valor();
        if (borriquines[v]) {
            valor = a.valor();
        }
        if (dist[w] > valor) {
            dist[w] = valor; ulti[w] = a;
            pq.update(w, dist[w]); //log V
            if (!pMarcado[w] && dist[w] ≤ distanciaMax) {
                pMarcado[w] = true;
                ++nPueblos;
            }
        }
    }
};

bool resuelveCaso() {

    // leemos la entrada
    int D, P, C;
    cin >> D >> P >> C;

    if (!cin)
        return false;

    // leer el resto del caso y resolverlo
    DigrafoValorado<int> dgv(P);
    int v, w, val;
    for (int i = 0; i < C; ++i) {
        cin >> v >> w >> val;
        --v; --w;
        AristaDirigida<int> aux(v, w, val);
        AristaDirigida<int> aux2(w, v, val);
        dgv.ponArista(aux);
        dgv.ponArista(aux2);
    }

    vector<bool> borriquines(P, false);
    cin >> val;
    vector<int> pueblosBorriquin(val);
    for (int i = 0; i < val; ++i) {
        cin >> v;
        --v;
        borriquines[v] = true;
        pueblosBorriquin[i] = v;
    }
    //Dijkstra<int> dj(dgv, v, D, borriquines);
    int cont = 0;
    while (!pueblosBorriquin.empty()) {
        int ori = pueblosBorriquin.back();
```

```
            pueblosBorriquin.pop_back();
            if (borriquines[ori]) {
                Dijkstra<int> dj(dgv, ori, D, borriquines);
                cont += dj.pueblosCercanos();
            }
        }

        cout << cont << "\n";

        return true;
    }
```