Soluciones OIFem II Nivel 2

Entrenamiento 7

Tree Matching

Teoría

- DFS
- Programación dinámica

Solución

Hacemos una DP donde cogemos el nodo y una donde no lo cogemos y la rellenamos con DFS.

Código

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>> adjList;
vector<int> DPcogido, DPnoCogido;

void dfs(int S, int par) {
        for (int v: adjList[S]) {
                if (v == par)
            continue;
                dfs(v, S);
                DPnoCogido[S] += max(DPcogido[v], DPnoCogido[v]);
        }
        for (int v: adjList[S])
                if (v != par)
            DPcogido[S] = max(DPcogido[S], 1 + DPnoCogido[S] + min(0,
            ↪  DPnoCogido[v] - DPcogido[v]));
}

int main() {
        int N, a, b;
        cin >> N;
        adjList = vector<vector<int>>(N);
        DPcogido.assign(N, 0);
        DPnoCogido.assign(N, 0);
```

```
        for (int i = 0; i < N-1; i++) {
                cin >> a >> b;
                a--;
                b--;
                adjList[a].push_back(b);
                adjList[b].push_back(a);
        }
        dfs(0, -1);
        cout << max(DPcogido[0], DPnoCogido[0]) << "\n";
        return 0;
}
```

Tree Distances I

Teoría

- Diámetro de un árbol

Solución

Modificamos el algoritmo para encontrar el diámetro de un árbol de forma que nos quedemos las distancias de todos los nodos a u y v.

Código

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<vector<pair<int, int>>> adjList;
int inf = 1e9;

pair<vector<int>, int> nodoMasLejano(int raiz) {
        int N = (int) adjList.size();
        vector<int> dist(N, inf);
        dist[raiz] = 0;
        queue<int> Q;
        Q.push(raiz);
        int ultNodo = raiz;
        while(!Q.empty()) {
                int nodoActual = Q.front();
                Q.pop();
                ultNodo = nodoActual;
                for (pair<int, int> conexion: adjList[nodoActual]) {
                        int indiceConexion = conexion.first;
                        int costeArista = conexion.second;
                        if (dist[indiceConexion] == inf) {
                                dist[indiceConexion] = dist[nodoActual] +
                                ↪ costeArista;
                                Q.push(indiceConexion);
                        }
```

```
                }
        }
        return make_pair(dist, ultNodo);
}

void maxDist() {
        pair<vector<int>, int> masLejanoDe0 = nodoMasLejano(0);
        pair<vector<int>, int> distanciaDe_u = nodoMasLejano(masLejanoDe0.second);
        pair<vector<int>, int> distanciaDe_v =
        ↪  nodoMasLejano(distanciaDe_u.second);
        for (size_t i = 0; i < distanciaDe_u.first.size(); i++)
        cout << max(distanciaDe_u.first[i], distanciaDe_v.first[i]) << ' ';
}

int main() {
        int N, a, b;
        cin >> N;
        adjList = vector<vector<pair<int, int>>>(N);
        for (int i = 0; i < N-1; i++) {
        cin >> a >> b;
        a--;
        b--;
        adjList[a].push_back(make_pair(b, 1));
        adjList[b].push_back(make_pair(a, 1));
        }
        maxDist();
        return 0;
}
```

Longest Consecutive Sequence

Teoría

- UFDS

Solución

Usamos UFDS y conectamos las posiciones de la lista que contengan números consecutivos. De esta forma, la solución es el conjunto más grande. Hay que tener cuidado con los casos en los que un número aparece en varias posiciones de forma que no lo contemos cada vez que aparece.

Código

```
typedef vector<int> vi;

class Solution {
public:
    vi par, sz;
    int root(int a) {
        if (a == par[a]) return a;
        return par[a] = root(par[a]);
```

```
    }

    void merge(int a, int b) {
        int rA = root(a), rB = root(b);
        if (rA == rB) return;
        if (sz[rA] <= sz[rB]) {
            par[rA] = rB;
            sz[rB] += sz[rA];
        } else {
            par[rB] = rA;
            sz[rA] += sz[rB];
        }
    }

    int longestConsecutive(vector<int>& nums) {
        unordered_map<int, int> pos;
        pos.reserve(4096);
        int n = (int) nums.size();
        if (!n) return 0;
        par = vi(n);
        sz.assign(n, 1);
        for (int i = 0; i < n; i++) par[i] = i;
        for (int i = 0; i < n; i++) pos[nums[i]] = i;
        for (int i = 0; i < n; i++) {
            if (i != pos[nums[i]] || pos.find(nums[i]+1) == pos.end()) continue;
            merge(i, pos[nums[i]+1]);
        }
        int rec = 0;
        for (int i = 0; i < n; i++) rec = max(rec, sz[i]);
        return rec;
    }
};
```

Accounts Merge

Teoría

- UFDS
- Mapas

Solución

Simulamos el proceso usando UFDS y mapas. Lo complicado de este problema era evitar errores en la implementación.

Código

```
class Solution {
public:
    vector<int> par, rnk;
```

```
    int root(int a) {
        if (a == par[a]) return a;
        return par[a] = root(par[a]);
    }

    void merge(int a, int b) {
        int rA = root(a), rB = root(b);
        if (rA == rB) return;
        if (rnk[rA] != rnk[rB]) {
            if (rnk[rA] < rnk[rB]) par[rA] = rB;
            else par[rB] = rA;
        } else {
            rnk[rB]++;
            par[rA] = rB;
        }
    }

    vector<vector<string>> accountsMerge(vector<vector<string>> & accounts) {
        int num = 0;
        map<string, int> dict;
        map<int, string> dict_inv;
        for (int i = 0; i < (int) accounts.size(); i++) {
            for (int j = 1; j < (int) accounts[i].size(); j++) {
                if (dict.find(accounts[i][j]) == dict.end()) {
                    dict[accounts[i][j]] = num;
                    dict_inv[num] = accounts[i][j];
                    num++;
                }
            }
        }
        par = rnk = vector<int>(num, 0);
        for (int i = 1; i < num; i++) par[i] = i;
        for (int i = 0; i < (int) accounts.size(); i++) {
            for (int j = 2; j < (int) accounts[i].size(); j++) {
                merge(dict[accounts[i][1]], dict[accounts[i][j]]);
            }
        }
        set<int> visited;
        vector<vector<string>> ret = {};
        vector<string> cur;
        map<int, vector<int>> groups;
        for (int i = 0; i < num; i++) {
            groups[root(i)].push_back(i);
        }
        for (int i = 0; i < (int) accounts.size(); i++) {
            if (visited.find(root(dict[accounts[i][1]])) != visited.end())
            ↪  continue;
            visited.insert(root(dict[accounts[i][1]]));
            cur = {};
            for (auto p: groups[root(dict[accounts[i][1]])])
            ↪  cur.push_back(dict_inv[p]);
            sort(cur.begin(), cur.end());
            cur.insert(cur.begin(), accounts[i][0]);
            ret.push_back(cur);
```

```
        }
        return ret;
    }
};
```

Traffic

Teoría

- Programación dinámica sobre árboles

Solución

Usamos un DFS para calcular una DP, donde `DP[i]` contiene la suma de las poblaciones de las ciudades en el subárbol de i, con la ciudad 0 como raíz.

Una vez procesados estos subárboles, es sencillo ver cuál sería la máxima concentración eligiendo a cada nodo i como sede del partido. Basta con coger el máximo de las concentraciones de los subárboles de los hijos de i y de la población que va desde el padre de i a i.

La única que hay que calcular de forma distinta es la máxima concentración que va a la raíz, que es la máxima suma de poblaciones en el subárbol de un hijo suyo.

Código

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
typedef long long int ll;

ll dfs(int S, int padre, vector<vector<int>> & listaAdyacencia, vector<ll> & DP) {
        for (int v: listaAdyacencia[S]) {
                if (v == padre)
                        continue;
                DP[S] += dfs(v, S, listaAdyacencia, DP);
        }
        // DP[S] contiene la suma de las poblaciones del subárbol de S
        return DP[S];
}

int LocateCentre(int N, int P[], int S[], int D[]) {
        vector<ll>DP(N);
        vector<vector<int>> listaAdyacencia(N, vector<int>());
        for (int i = 0; i < N-1; i++) {
                // pasamos el formato de entrada a una lista de adyacencia, más
                ↪  cómodo para operar
                // al ser aristas bidireccionales, las añadimos en ambas
                ↪  direcciones
                listaAdyacencia[S[i]].emplace_back(D[i]);
                listaAdyacencia[D[i]].emplace_back(S[i]);
```

```
        }
        for (int i = 0; i < N; i++)
                DP[i] = P[i]; // inicializamos la DP para que contenga la
                ↪  población de cada ciudad

        // rellenamos la DP
        dfs(0, -1, listaAdyacencia, DP);

        // por defecto, elegimos 0 como ciudad arbitrariamente y encontramos la
        ↪  máxima congestión
        int ciudadElegida = 0;
        ll maximaConcentracionCiudadElegida = 0;
        for (int vecinoRaiz: listaAdyacencia[0]) {
                maximaConcentracionCiudadElegida =
                ↪  max(maximaConcentracionCiudadElegida, DP[vecinoRaiz]);
        }

        ll maximaConcentracionCiudadActual;
        for (int ciudadActual = 1; ciudadActual < N; ciudadActual++) {
                // inicialmente, diremos que la máxima congestión es la densidad
                ↪  de coches que viaja desde el "padre" de la ciudad actual (con
                ↪  0 como raíz) a la ciudad actual
                maximaConcentracionCiudadActual = DP[0] - DP[ciudadActual];
                for (int v: listaAdyacencia[ciudadActual]) {
                        // comprobamos para todos los hijos de la ciudad actual si
                        ↪  la densidad de la ciudad al hijo es superior que la
                        ↪  del padre a la ciudad actual
                        if (DP[v] < DP[ciudadActual])
                                maximaConcentracionCiudadActual =
                                ↪  max(maximaConcentracionCiudadActual, DP[v]);
                }

                if (maximaConcentracionCiudadActual <
                ↪  maximaConcentracionCiudadElegida) {
                        // la ciudad es un nuevo récord -> actualizamos
                        maximaConcentracionCiudadElegida =
                        ↪  maximaConcentracionCiudadActual;
                        ciudadElegida = ciudadActual;
                }
        }
        return ciudadElegida;
}
```

Is Graph Bipartite?

Teoría

- DFS

Solución

Para saber si un grafo es bipartito, coloreamos un nodo escogido arbitrariamente con uno de los dos colores, escogido también arbitrariamente. Lo más común suele ser pintar el nodo 0 de color 0. Iniciamos

entonces una DFS, pintando sus vecinos de 1, los vecinos de estos de 0, etc. Si en algún momento nos encontramos una contradicción, el grafo no es bipartito y guardamos esta información en la variable `isPossible`.

Código

```cpp
class Solution {
public:
    bool isPossible;

    void dfs(int x, int col, vector<int> & color, vector<vector<int>>& graph) {
        color[x] = col;
        for (int neigh: graph[x]) {
            if (color[neigh] == color[x])
                isPossible = false;
            else if (color[neigh] == -1)
                dfs(neigh, 1-col, color, graph);
        }
    }

    bool isBipartite(vector<vector<int>>& graph) {
        isPossible = true;
        vector<int> color(graph.size(), -1);
        for (size_t i = 0; i < graph.size(); i++) {
            if (color[i] == -1)
                dfs(i, 0, color, graph);
        }
        return isPossible;
    }
};
```

Swim in Rising Water

Teoría

- UFDS
- Mapas

Solución

Para resolver este problema, podemos simularlo de la siguiente manera: empezando el contador de tiempo en 0, vamos dejando que pase el tiempo y, cuando el tiempo sea igual a t, unimos el cuadrado que tenga valor t con sus vecinos con un valor menor. Retornaremos el momento en el que todos los cuadrados formen parte del mismo conjunto.

Código

```cpp
class Solution {
public:
    vector<int> par, rnk;
    int root(int a) {
        if (a == par[a]) return a;
        return par[a] = root(par[a]);
    }
    void merge(int a, int b) {
        int rA = root(a), rB = root(b);
        if (rA == rB) return;
        if (rnk[rA] != rnk[rB]) {
            if (rnk[rA] < rnk[rB]) par[rA] = rB;
            else par[rB] = rA;
        } else {
            rnk[rB]++;
            par[rA] = rB;
        }
    }
    int swimInWater(vector<vector<int>> & grid) {
        int N = (int) grid.size();
        if(N <= 1) return 0;
        map<int, vector<int>> timePos;
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                timePos[grid[i][j]].push_back(i*N + j);
        par = rnk = vector<int>(N*N, 1);
        for (int i = 0; i < N*N; i++) par[i] = i;
        map<int, vector<int>>::iterator it = timePos.begin();
        int t = 0, r, c;
        while(it != timePos.end()) {
            if (root(0) == root(N*N-1)) break;
            t = it->first;
            for (auto pos: it->second) {
                r = pos/N;
                c = pos%N;
                if (r > 0 && grid[r - 1][c] <= t) merge(pos, pos-N);
                if (r < N-1 && grid[r + 1][c] <= t) merge(pos, pos+N);
                if (c < N-1 && grid[r][c + 1] <= t) merge(pos, pos+1);
                if (c > 0 && grid[r][c - 1] <= t) merge(pos, pos-1);
            }
            it++;
        }
        return t;
    }
};
```

Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree

Teoría

- UFDS
- Priority queues
- Kruskal MST

Solución

Encontramos el MST con Kruskal. Procedemos a quitar una a una (y devolverlas) las aristas del grafo, calculando el MST del grafo sin la i-ésima arista, para todo i. Así, podíamos encontrar las aristas críticas. Si una arista no era crítica, le restábamos uno de longitud y probabámos a encontrar el MST, a ver si se mantenía o no.

Código

```cpp
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int, int> pii;
typedef pair<int, pii> triple;

class Solution {
public:
    vi par, sz;
    int numSets, maxW;

    int root(int a) {
        if (a == par[a]) return a;
        return par[a] = root(par[a]);
    }

    void merge(int a, int b) {
        int rA = root(a), rB = root(b);
        if (rA == rB) return;
        numSets--;
        if (sz[rA] <= sz[rB]) {
            par[rA] = rB;
            sz[rB] += sz[rA];
        } else {
            par[rB] = rA;
            sz[rA] += sz[rB];
        }
    }

    // encuentra el MST sin incluir la arista i
    int KruskalAvoid(int n, vvi & edges, int i) {
        priority_queue<triple> pq1, pq2;
        int cost = 0, u, v, w;
        maxW = 0;
        for (int i = 0; i < n; i++) par[i] = i;
```

```
        sz.assign(n, 1);
        numSets = n;
        for (int j = 0; j < (int) edges.size(); j++) {
            if (i == j) continue;
            if (j & 1) pq1.push({-edges[j][2], {edges[j][0], edges[j][1]}});
            else pq2.push({-edges[j][2], {edges[j][0], edges[j][1]}});
        }
        while(pq1.size() && pq2.size()) {
            if (numSets == 1) break;
            if (pq1.top().first >= pq2.top().first) {
                u = pq1.top().second.first;
                v = pq1.top().second.second;
                w = -pq1.top().first;
                pq1.pop();
            } else {
                u = pq2.top().second.first;
                v = pq2.top().second.second;
                w = -pq2.top().first;
                pq2.pop();
            }
            if (root(u) == root(v)) continue;
            merge(u, v);
            cost += w;
            maxW = w;
        }
        while(pq1.size()) {
            if (numSets == 1) break;
            u = pq1.top().second.first;
            v = pq1.top().second.second;
            w = -pq1.top().first;
            pq1.pop();
            if (root(u) == root(v)) continue;
            merge(u, v);
            cost += w;
            maxW = w;
        }
        while(pq2.size()) {
            if (numSets == 1) break;
            u = pq2.top().second.first;
            v = pq2.top().second.second;
            w = -pq2.top().first;
            pq2.pop();
            if (root(u) == root(v)) continue;
            merge(u, v);
            cost += w;
            maxW = w;
        }
        return cost;
    }

    // encuentra el MST incluyendo la arista i pero restándole 1 de longitud
    int KruskalSubtract(int n, vvi & edges, int i) {
        priority_queue<triple> pq1, pq2;
        int cost = 0, u, v, w;
        for (int i = 0; i < n; i++) par[i] = i;
```

```
        sz.assign(n, 1);
        numSets = n;
        for (int j = 0; j < (int) edges.size(); j++) {
            if (i == j) edges[i][2]--;
            if (j & 1) pq1.push({-edges[j][2], {edges[j][0], edges[j][1]}});
            else pq2.push({-edges[j][2], {edges[j][0], edges[j][1]}});
        }
        edges[i][2]++;
        while(pq1.size() && pq2.size()) {
            if (numSets == 1) break;
            if (pq1.top().first >= pq2.top().first) {
                u = pq1.top().second.first;
                v = pq1.top().second.second;
                w = -pq1.top().first;
                pq1.pop();
            } else {
                u = pq2.top().second.first;
                v = pq2.top().second.second;
                w = -pq2.top().first;
                pq2.pop();
            }
            if (root(u) == root(v)) continue;
            merge(u, v);
            cost += w;
        }
        while(pq1.size()) {
            if (numSets == 1) break;
            u = pq1.top().second.first;
            v = pq1.top().second.second;
            w = -pq1.top().first;
            pq1.pop();
            if (root(u) == root(v)) continue;
            merge(u, v);
            cost += w;
        }
        while(pq2.size()) {
            if (numSets == 1) break;
            u = pq2.top().second.first;
            v = pq2.top().second.second;
            w = -pq2.top().first;
            pq2.pop();
            if (root(u) == root(v)) continue;
            merge(u, v);
            cost += w;
        }
        return cost;
    }
    vvi findCriticalAndPseudoCriticalEdges(int n, vvi & edges) {
        vvi ret = {{}, {}};
        par = vi(n);
        int MST = KruskalAvoid(n, edges, -1), cost;
        for (int i = 0; i < (int) edges.size(); i++) {
            cost = KruskalAvoid(n, edges, i);
            if (cost > MST || numSets != 1) ret[0].push_back(i);
            else {
```

```
                if (maxW < edges[i][2]) continue;
                cost = KruskalSubtract(n, edges, i);
                if (cost < MST) ret[1].push_back(i);
            }
        }
        return ret;
    }
};
```

War

Teoría

- UFDS

Solución

Basaremos nuestra solución en guardar para cada nodo el índice de un enemigo suyo. Esto será suficiente, ya que sabemos que solo hay dos conjuntos distintos (los dos bandos de la guerra) en el mundo real. El código se explica solo. Basta con tener en cuenta que el enemigo de un nodo será -1 si aún no hemos encontrado tal enemigo.

Código

C++

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> padre;
vector<int> alturaAprox;
vector<int> enemigos;
int N;

int raiz(int a) {
        if (a == padre[a]) return a;
        return padre[a] = raiz(padre[a]);
}

void unir(int a, int b) {
        int raiz_a, raiz_b;
        raiz_a = raiz(a);
        raiz_b = raiz(b);

        if (raiz_a == raiz_b) return;

        if (alturaAprox[raiz_a] != alturaAprox[raiz_b]) {
                if (alturaAprox[raiz_a] > alturaAprox[raiz_b]) padre[raiz_b] =
                ↪  raiz_a;
                else padre[raiz_a] = raiz_b;
        }
```

```
26          else {
27                  padre[raiz_a] = raiz_b;
28                  alturaAprox[raiz_b]++;
29          }
30  }
31
32  bool sonamigos(int a, int b) {
33          int raiz_a = raiz(a), raiz_b = raiz(b);
34
35          if (raiz_a == raiz_b)
36                  return true;
37
38          // min(enemigos[raiz_a], enemigos[raiz_b]) != -1 significa que ambos
            ↪  tienen ya un enemigo
39          return (min(enemigos[raiz_a], enemigos[raiz_b]) != -1 &&
            ↪  (raiz(enemigos[raiz_a]) == raiz(enemigos[raiz_b])));
40  }
41
42  bool sonenemigos(int a, int b) {
43          if (sonamigos(a, b))
44                  return false;
45
46          int raiz_a = raiz(a), raiz_b = raiz(b);
47
48          if (max(enemigos[raiz_a], enemigos[raiz_b]) == -1)
49                  return false;
50
51          if (enemigos[raiz_a] == -1)
52                  return sonamigos(raiz_a, enemigos[raiz_b]);
53
54          if (enemigos[raiz_b] == -1)
55                  return sonamigos(raiz_b, enemigos[raiz_a]);
56
57          return (raiz(enemigos[raiz_a]) == raiz_b || raiz(enemigos[raiz_b]) ==
            ↪  raiz_a);
58  }
59
60  void haceramigos(int a, int b) {
61          int raiz_a = raiz(a), raiz_b = raiz(b);
62
63          if (sonenemigos(a, b) || (enemigos[raiz_a] != -1 && enemigos[raiz_b] != -1
            ↪  && sonenemigos(enemigos[raiz_a], enemigos[raiz_b]))) {
64                  cout << "-1\n";
65                  return;
66          }
67
68          if (sonamigos(a, b))
69                  return;
70
71          unir(a, b);
72
73          if (min(enemigos[raiz_a], enemigos[raiz_b]) != -1)
74                  unir(enemigos[raiz_a], enemigos[raiz_b]);
75
76          if (max(enemigos[raiz_a], enemigos[raiz_b]) == -1)
```

```
77                  return;
78
79          if (enemigos[raiz_a] == -1)
80                  enemigos[raiz_a] = raiz(enemigos[raiz_b]);
81
82          if (enemigos[raiz_b] == -1)
83                  enemigos[raiz_b] = raiz(enemigos[raiz_a]);
84  }
85
86  void hacerenemigos(int a, int b) {
87          int raiz_a = raiz(a), raiz_b = raiz(b);
88
89          if (sonamigos(a, b) || (enemigos[raiz_a] != -1 &&
            ↪  sonenemigos(enemigos[raiz_a], b)) || (enemigos[raiz_b] != -1 &&
            ↪  sonenemigos(enemigos[raiz_b], a))) {
90                  cout << "-1\n";
91                  return;
92          }
93
94          if (sonenemigos(a, b))
95                  return;
96
97          if (enemigos[raiz_a] != -1)
98                  unir(enemigos[raiz_a], b);
99          else
100                 enemigos[raiz_a] = raiz_b;
101
102         if (enemigos[raiz_b] != -1)
103                 unir(enemigos[raiz_b], a);
104         else
105                 enemigos[raiz_b] = raiz_a;
106 }
107
108 int main() {
109         int a, b, c;
110         cin >> N;
111         enemigos.assign(N, -1);
112         alturaAprox.assign(N, 0);
113         padre.assign(N, 0);
114         for (int u = 0; u < N; u++)
115                 padre[u] = u;
116
117         while(true) {
118                 cin >> c >> a >> b;
119                 if (!c) break;
120                 if (c == 1)
121                         haceramigos(a, b);
122                 else if (c == 2)
123                         hacerenemigos(a, b);
124                 else if (c == 3)
125                         cout << sonamigos(a, b) << "\n";
126                 else
127                         cout << sonenemigos(a, b) << "\n";
128         }
129         return 0;
130 }
```

Dreaming

Teoría

- Diámetro
- Árboles

Solución

Nota: La eccentricidad de un nodo es la mayor distancia desde ese nodo al nodo más lejano de la componente conexa en la que se encuentra.

Para conseguir los 100 puntos, basta con una solución que conecte los árboles del bosque en una forma de estrella, con una componente en el centro y conectando esta componente con el resto de componentes con una arista para cada uno.

Lo fundamental del problema es el decidir cuál de las componentes será la del centro de la estrella. Para ello, calcularemos para todos los nodos cuál es su eccentricidad. Elegiremos como centro la componente conexa cuyo nodo con eccentricidad mínima sea el máximo del resto de componentes. Esta solución tiene una complejidad de O(n).

Código

C++

```cpp
#include <vector>
#include <queue>
#include "dreaming.h"
using namespace std;

pair<int,int> calcDist(int i, vector<vector<pair<int, int>>> & listaAdyacencia, vector<int> & dist1, vector<int> & dist2, vector<int> & dist3) {
        // calcularemos el diámetro con el método de las dos colas, que primero busca el nodo más lejano de la raíz i (nodoMax)
        // después, rellenamos dist2 con la distancia entre nodoMax y el resto de nodos de la componente conexa
        // así, encuentra el nodo más lejano de nodoMax (nodoMax2). El diámetro es la distancia entre nodoMax y nodoMax2.
        // finalmente, rellenamos dist3 con la distancia entre nodoMax2 y el resto de nodos de la componente conexa
        // la eccentricidad de cada nodo será el máximo entre la distancia con nodoMax y la distancia con nodoMax2
        queue<int> Q;
        Q.push(i);
        int u, v, t, nodoMax = i;
        vector<int> nodos = {}; // aquí guardaremos los nodos de la componente conexa
        dist1[i] = 0;
        while(!Q.empty()) {
                u = Q.front();
                Q.pop();
                nodos.push_back(u); // añadimos u a la lista de nodos de la componente conexa
```

```
                if (dist1[u] > dist1[nodoMax]) // actualizamos nodoMax si procede
                        nodoMax = u;
                for (auto conexion: listaAdyacencia[u]) { // procesamos las
                ↪  conexiones de u
                        v = conexion.first;
                        t = conexion.second;
                        if (dist1[v] > dist1[u]+t) { // v no visitado
                                dist1[v] = dist1[u]+t; // actualizamos la
                                ↪  distancia de v en base a la de u
                                Q.push(v);
                        }
                }
        }

    // hacemos lo mismo para nodoMax
        dist2[nodoMax] = 0;
        Q.push(nodoMax);
        int nodoMax2 = nodoMax;
        while(!Q.empty()) {
                u = Q.front();
                Q.pop();
                if (dist2[u] > dist2[nodoMax2])
                        nodoMax2 = u;
                for (auto conexion: listaAdyacencia[u]) {
                        v = conexion.first;
                        t = conexion.second;
                        if (dist2[v] > dist2[u]+t) {
                                dist2[v] = dist2[u]+t;
                                Q.push(v);
                        }
                }
        }

    // hacemos lo mismo para nodoMax2
        dist3[nodoMax2] = 0;
        Q.push(nodoMax2);
        while(!Q.empty()) {
                u = Q.front();
                Q.pop();
                for (auto conexion: listaAdyacencia[u]) {
                        v = conexion.first;
                        t = conexion.second;
                        if (dist3[v] > dist3[u]+t) {
                                dist3[v] = dist3[u]+t;
                                Q.push(v);
                        }
                }
        }

        int ecc = 1e9;
        for (auto nodo: nodos) {
                ecc = min(ecc, max(dist2[nodo], dist3[nodo])); // calculamos la
                ↪  eccentricidad de nodo y actualizamos el mínimo
        }
        return make_pair(ecc, dist2[nodoMax2]);
```

```cpp
}

int travelTime(int N, int M, int L, int A[], int B[], int T[]) {
        // primero, pasamos las listas de aristas a un grafo en formato "adjacency list"
        // guardamos cada arista en la lista de sus dos nodos, ya que el grafo es bidireccional
        vector<vector<pair<int, int>>> listaAdyacencia(N);
        for (int i = 0; i < M; i++) {
                listaAdyacencia[A[i]].push_back(make_pair(B[i], T[i]));
                listaAdyacencia[B[i]].push_back(make_pair(A[i], T[i]));
        }
        int CC = 0; // número de componentes conexas del grafo (un bosque)
        vector<int> diam; // diámetro de las componentes conexas
        vector<int> ecc; // guarda para cada componente conexa la eccentricidad del nodo con menor eccentricidad del grafo
        vector<int> dist1; // guarda la distancia entre la raíz de la componente conexa y cada uno de sus nodos
    vector<int> dist2; // guarda la distancia entre el nodo más lejano de la raíz de la componente conexa y el resto de sus nodos
        vector<int> dist3; // guarda la distancia entre el nodo que define dist2 y el resto de nodos de la componente conexa
    dist1 = dist2 = dist3 = vector<int>(N, 1e9); // asignamos "infinito" como valores iniciales para las distancias
    pair<int, int> curCC; // aquí guardamos el valor devuelto de la componente conexa
    for (int i = 0; i < N; i++) {
            if (dist1[i] == 1e9) { // nodo i no visitado aún
                curCC = calcDist(i, listaAdyacencia, dist1, dist2, dist3); // {ecc, diam} de la componente conexa de i
                ecc.push_back(curCC.first);
                diam.push_back(curCC.second);
                CC++;
            }
    }
        // para minimizar el camino más largo, guardaremos el grafo como una estrella, conectando los centros de cada
        // componente conexa todas al centro de la componente conexa con mayor eccentricidad
        // así, la respuesta será una de las siguientes:
        // 1. la componente conexa con el mayor diámetro
        // 2. la mayor distancia entre el centro de las dos componentes con mayor eccentricidad
        // 3. la distancia entre la segunda y la tercera mayor eccentricidad, pasando por el centro medio

        // guardamos en ecc[0] la eccentricidad mayor
        for (int i = 1; i < CC; i++) {
                if (ecc[i] > ecc[0])
                        swap(ecc[0], ecc[i]);
        }
        int ans = 0;
        if (CC > 1) {
                // guardamos en ecc[1] la sedunda eccentricidad mayor
                for (int i = 2; i < CC; i++) {
```

```
                        if (ecc[i] > ecc[1])
                                swap(ecc[i], ecc[1]);
                }
                ans = ecc[0]+ecc[1]+L; // posibilidad 2
                if (CC > 2) {
                        // guardamos en ecc[2] la tercera eccentricidad mayor
                        for (int i = 3; i < CC; i++) {
                                if (ecc[i] > ecc[2])
                                        swap(ecc[i], ecc[2]);
                        }
                        ans = max(ans, ecc[1]+ecc[2]+2*L); // posibilidad 3
                }
        }

        for (int i = 0; i < CC; i++) {
                ans = max(ans, diam[i]); // posibilidad 1
        }

        return ans;
}
```

Connecting Supertrees

Teoría

- Conjuntos distintos
- Árboles

Solución

Para resolver este problema, separaremos el grafo en sus distintas componentes conexas, usando conjuntos distintos. Hay que tener en cuenta que si hay cualquier pareja con 3 conexiones, es imposible encontrar tal grafo.

Por lo tanto, terminamos con una serie de componentes conexas, cuyos nodos tienen o un camino o dos entre ellos. Si una componente conexa puede construirse según las especificaciones, tiene que cumplirse los siguiente:

1. Todos los nodos de la componente conexa deben de tener por lo menos un camino entre ellos ($p[i][j] > 0$ para todo i, j en la componente conexa).
2. La componente conexa se puede partir en una serie de "serpientes". Los nodos dentro de la misma serpiente tienen un camino que los conecta y dos con los del resto de serpientes ($p[i][j] = 1$ si i y j comparten serpiente y $p[i][j] = 2$ si no es el caso.

Por lo tanto, construimos las serpientes también con conjuntos distintos, asegurándonos de que no haya incongruencias respecto a estas condiciones.

Finalmente, conectamos los elementos dentro de la serpiente en línea recta. Así, solo hay un camino posible entre cualquier pareja dentro de la serpiente.

Conectamos la cabeza de todas las serpientes en un "círculo de cabezas", consiguiendo así que haya dos formas de conectarse con cualquier nodo ajeno a la serpiente.

Código

C++

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include "supertrees.h"
using namespace std;

int raiz(int a, vector<int> & padre) {
    if (padre[a] == a) return a;
    return padre[a] = raiz(padre[a], padre);
}

void unirConjuntos(int a, int b, vector<int> & padre, vector<int> & alturaAprox,
 ↪  int & numeroConjuntos) {
    int raiz_a, raiz_b;
    raiz_a = raiz(a, padre);
    raiz_b = raiz(b, padre);
    if (raiz_a == raiz_b) return;
    numeroConjuntos--;
    if (alturaAprox[raiz_a] > alturaAprox[raiz_b]) {
        padre[raiz_b] = raiz_a;
    } else if (alturaAprox[raiz_b] > alturaAprox[raiz_a]) {
        padre[raiz_a] = raiz_b;
    } else {
        padre[raiz_a] = raiz_b;
        alturaAprox[raiz_b]++;
    }
}

int construct(vector<vector<int>> p) {
        int N = (int) p.size(), numCCs = N;
        vector<int> padre(N), alturaAprox(N, 0);

        for (int i = 0; i < N; i++)
                padre[i] = i;

        // encontramos las componentes conexas
        for (int i = 0; i < N; i++) {
                for (int j = i+1; j < N; j++) {
                        if (p[i][j]) {
                                if (p[i][j] == 3) {
                                        return 0;
                                }
                                unirConjuntos(i, j, padre, alturaAprox, numCCs);
                        } else if (raiz(i, padre) == raiz(j, padre))
                                return 0;
                }
        }

```

```
48          // damos un índice a cada raíz
49          unordered_map<int, int> raices;
50          int c = 0;
51          for (int i = 0; i < N; i++) {
52                  if (padre[i] == i)
53                          raices[i] = c++;
54          }
55
56          // guardamos las componentes conexas
57          vector<vector<int>> CCs(numCCs);
58          for (int i = 0; i < N; i++)
59                  CCs[raices[raiz(i, padre)]].push_back(i);
60
61          int CCtamano, u, v, n, raiz_u, raiz_v;
62
63          vector<vector<int>> resultadoFinal(N, vector<int>(N, 0)), serpientes;
64
65          for (int i = 0; i < numCCs; i++) {
66                  CCtamano = (int) CCs[i].size();
67                  if (CCtamano == 1) continue;
68
69                  // encontramos las "serpientes" de esta componente conexa
70                  n = CCtamano;
71                  for (auto node: CCs[i]) {
72                          padre[node] = node;
73                          alturaAprox[node] = 0;
74                  }
75                  for (int j = 0; j < CCtamano; j++) {
76                          u = CCs[i][j];
77                          raiz_u = raiz(u, padre);
78                          for (int k = j+1; k < CCtamano; k++) {
79                                  v = CCs[i][k];
80                                  if (p[u][v] == 1)
81                                          unirConjuntos(u, v, padre, alturaAprox,
                                            ↪  n);
82                                  else if (raiz_u == raiz(v, padre))
83                                          return 0;
84                          }
85                  }
86
87                  // nos aseguramos de que no hay incongruencias en esta componente
                    ↪  conexa
88                  if (n == 2) return 0;
89
90                  raices.clear();
91                  c = 0;
92                  for (int j = 0; j < CCtamano; j++) {
93                          u = CCs[i][j];
94                          raiz_u = raiz(u, padre);
95                          if (raiz_u == u)
96                                  raices[u] = c++;
97                          for (int k = j+1; k < CCtamano; k++) {
98                                  v = CCs[i][k];
99                                  raiz_v = raiz(v, padre);
100                                 if (raiz_u != raiz_v) {
```

```
                                        if (p[u][v] != 2) {
                                                return 0;
                                        }
                                } else if (p[u][v] != 1) {
                                        return 0;
                                }
                        }
                }

                // conectamos los elementos dentro de una serpiente
                serpientes = vector<vector<int>>(n);
                for (int j = 0; j < CCtamano; j++)
                        serpientes[raices[raiz(CCs[i][j],
                        ↪  padre)]].push_back(CCs[i][j]);
                for (int j = 0; j < n; j++) {
                    for (int k = 0; k < (int) serpientes[j].size() - 1; k++)
                        resultadoFinal[serpientes[j][k]][serpientes[j][k+1]] =
                        ↪  resultadoFinal[serpientes[j][k+1]][serpientes[j][k]]
                        ↪  = 1;
                }

                // conectamos el ciclo de cabezas de serpiente
                for (int j = 0; j < n-1; j++)
                        resultadoFinal[serpientes[j][0]][serpientes[j+1][0]] =
                        ↪  resultadoFinal[serpientes[j+1][0]][serpientes[j][0]] =
                        ↪  1;

                if (n > 1)
                        resultadoFinal[serpientes[0][0]][serpientes[n-1][0]] =
                        ↪  resultadoFinal[serpientes[n-1][0]][serpientes[0][0]] =
                        ↪  1;
        }
        build(resultadoFinal);
        return 1;
}
```