Soluciones OIFem 2020
Entrenamiento 8

Diámetro

Teoría

- Diámetro de un árbol

Solución

Usamos la solución de dos colas vista en clase y lo resolvemos con el código de los apuntes.

Código

C++

```cpp
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;

int diametro(vector<vector<int>> & grafo) {
    // grafo contiene la lista de adyacencia del árbol
    int N = (int) grafo.size();
    queue<int> bfsCola;
    bfsCola.push(0);
    vector<int> dist(N, 1e9);
    dist[0] = 0;
    int u = 0;
    while(!bfsCola.empty()) {
        u = bfsCola.front();
        bfsCola.pop();
        for (int v: grafo[u]) {
            if (dist[v] > dist[u]+1) {
                dist[v] = dist[u]+1;
                bfsCola.push(v);
            }
        }
    }
    // u va a ser el nodo más lejano
```

```
    bfsCola.push(u);
    dist.assign(N, 1e9);
    dist[u] = 0;
    int nodoLejano = u;
    while(!bfsCola.empty()) {
        u = bfsCola.front();
        bfsCola.pop();
        nodoLejano = u;
        for (int v: grafo[u]) {
            if (dist[v] > dist[u]+1) {
                dist[v] = dist[u]+1;
                bfsCola.push(v);
            }
        }
    }
    // nodoLejano va a ser el nodo más lejano
    return dist[nodoLejano];
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int T, N, x, y;
    vector<vector<int>> g;
    cin >> T;
    while(T--) {
        cin >> N;
        g = vector<vector<int>>(N);
        for (int i = 0; i < N-1; i++) {
            cin >> x >> y;
            g[x].push_back(y);
            g[y].push_back(x);
        }
        cout << diametro(g) << '\n';
    }
    return 0;
}
```

Minimum Spanning Tree

Teoría

- Árboles recubridores mínimos
- Algoritmo de Kruskal
- Conjuntos distintos (UFDS)

Solución

Usaremos el código de Kruskal de los apuntes, retornando el coste de las aristas seleccionadas.

Código

C++

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

int numeroConjuntos;
vector<int> padre, alturaAprox;

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

bool mismoConjunto(int a, int b) {
    return encontrarRaiz(a) == encontrarRaiz(b);
}

void unirConjuntos(int a, int b) {
    int raiz_a, raiz_b;
    raiz_a = encontrarRaiz(a);
    raiz_b = encontrarRaiz(b);
    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 kruskalMST(int N, vector<pair<int, pair<int, int>>> & aristas) {
```

```
37      sort(aristas.begin(), aristas.end());
38      padre.assign(N, 0);
39      for (int i = 1; i < N; i++) padre[i] = i;
40      alturaAprox.assign(N, 0);
41      numeroConjuntos = N;
42      int j = 0, u, v, respuesta = 0;
43      while(numeroConjuntos > 1) {
44          u = aristas[j].second.first;
45          v = aristas[j].second.second;
46          if (mismoConjunto(u, v)) {
47              j++;
48              continue;
49          }
50          respuesta += aristas[j].first;
51          unirConjuntos(u, v);
52          j++;
53      }
54      return respuesta;
55  }
56
57  int main() {
58      int T, N, M;
59      vector<pair<int, pair<int, int>>> aristas;
60      cin >> T;
61      while(T--) {
62          cin >> N >> M;
63          aristas = vector<pair<int, pair<int, int>>>(M);
64          for (int i = 0; i < M; i++)
65              cin >> aristas[i].second.first >> aristas[i].second.second >>
                ↪  aristas[i].first;
66          cout << kruskalMST(N, aristas) << '\n';
67      }
68      return 0;
69  }
```

Grupos

Teoría

- Conjuntos distintos (UFDS)

Solución

Usaremos la estructura de datos de conjuntos distintos, donde si dos personas se hacen amigas fusionamos/unimos sus conjuntos y, para comprobar si son del mismo grupo, comprobamos que sus raíces sean las mismas.

Código

C++

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

int numeroConjuntos;
vector<int> padre, alturaAprox;

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

bool mismoConjunto(int a, int b) {
    return encontrarRaiz(a) == encontrarRaiz(b);
}

void unirConjuntos(int a, int b) {
    int raiz_a, raiz_b;
    raiz_a = encontrarRaiz(a);
    raiz_b = encontrarRaiz(b);
    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 main() {
```

```
38      int T, N, M, a, b, c;
39      cin >> T;
40      while(T--) {
41          cin >> N >> M;
42          padre.assign(N, 0);
43          for (int i = 1; i < N; i++) padre[i]=i;
44          alturaAprox.assign(N, 0);
45          while(M--) {
46              cin >> c >> a >> b;
47              if (c == 1)
48                  unirConjuntos(a, b);
49              else {
50                  if (mismoConjunto(a, b)) cout << "Si\n";
51                  else cout << "No\n";
52              }
53
54          }
55      }
56      return 0;
57  }
```

Traffic Congestion

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
```

```
29          dfs(0, -1, listaAdyacencia, DP);
30
31          // por defecto, elegimos 0 como ciudad arbitrariamente y encontramos la
            ↪  máxima congestión
32          int ciudadElegida = 0;
33          ll maximaConcentracionCiudadElegida = 0;
34          for (int vecinoRaiz: listaAdyacencia[0]) {
35                  maximaConcentracionCiudadElegida =
                    ↪  max(maximaConcentracionCiudadElegida, DP[vecinoRaiz]);
36          }
37
38          ll maximaConcentracionCiudadActual;
39          for (int ciudadActual = 1; ciudadActual < N; ciudadActual++) {
40                  // 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
41                  maximaConcentracionCiudadActual = DP[0] - DP[ciudadActual];
42                  for (int v: listaAdyacencia[ciudadActual]) {
43                          // 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
44                          if (DP[v] < DP[ciudadActual])
45                                  maximaConcentracionCiudadActual =
                                    ↪  max(maximaConcentracionCiudadActual, DP[v]);
46                  }
47
48                  if (maximaConcentracionCiudadActual <
                    ↪  maximaConcentracionCiudadElegida) {
49                          // la ciudad es un nuevo récord -> actualizamos
50                          maximaConcentracionCiudadElegida =
                            ↪  maximaConcentracionCiudadActual;
51                          ciudadElegida = ciudadActual;
52                  }
53          }
54          return ciudadElegida;
55  }
```

War

Teoría

- Conjuntos distintos

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++

```cpp
#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;
        }
        else {
                padre[raiz_a] = raiz_b;
                alturaAprox[raiz_b]++;
        }
}

bool sonamigos(int a, int b) {
        int raiz_a = raiz(a), raiz_b = raiz(b);

        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 }
```

Habitaciones

Teoría

- Componentes fuertemente conexas (CFC en español/SCC en inglés)
- Ordenamiento topológico
- DAGs (grafos digidos acíclicos)
- Conjuntos desordenados

Solución

En este problema, el primer paso es modelar las amistades como aristas en el grafo. Es decir, si la persona u confía en v para darle a v la otra copia de su llave, habrá una arista de la persona v a la persona u. Por lo tanto, sabemos que todas las aristas serán de grado 1, ya que cada persona tiene 1 llave que dejar. Sin embargo, pueden salir hasta $N-1$ aristas de un mismo nodo si este guarda el resto de llaves.

Una vez construido el grafo, nos fijamos en que, dentro de la misma componente fuertemente conexa, tienen que todos sus nodos perder su llave para que haya alguien dentro de la componente que se quede fuera de la habitación. Por lo tanto, el primer paso para resolver el problema será encontrar las componentes fuertemente conexas del grafo.

Usando el método ya visto anteriormente, retornamos el número de componentes, además de la componente a la que pertenece cada nodo, modificando ligeramente la función de los apuntes para que se empiece con el índice 0, y no el 1, a nombrar las componentes.

Si ahora construimos un grafo donde cada nodo represente a una componente fuertemente conexa en el grafo inicial, este será un DAG (tiene que ser así porque de haber un ciclo, esto formaría una CFC más grande en el grafo original). Esta construcción es el paso final, ya que para que una persona no pueda acceder a su habitación, tiene que perder la llave su componente entera, además de las componentes que desembocan en la componente de esta.

Modificamos el algoritmo de ordenamiento topológico para visitar las componentes en orden, es decir, que si la componente A desemboca en la B, visitemos la A antes. De esta forma, ya habremos calculado cuántas llaves han de perderse para que se pierdan las llaves de la componente A cuando visitemos la B. Visitamos las componentes en orden, usando una cola, solo que en vez de añadir las componentes a un orden, calculamos las pérdidas necesarias, posteriormente quitando este nodo/componente del DAG. Inicialmente, diremos que las llaves que ha de perder cada componente son iguales al tamaño de la componente, calculado previamente. De manera constructiva, vamos sumando las pérdidas de componentes anteriores a las futuras hasta visitar todas las componentes.

Código

C++

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

```

```cpp
void dfsComponente(int nodo, int ccActual, vector<int> & componente,
↪  vector<vector<int>> & listaAdyacencia) {
    componente[nodo] = ccActual;
    for (int conexion: listaAdyacencia[nodo]) {
        if (componente[conexion] == -1)
            dfsComponente(conexion, ccActual, componente, listaAdyacencia);
    }
}

void dfsCFC(int nodo, vector<bool> & visitado, stack<int> & S, vector<vector<int>>
↪  & listaAdyacencia) {
    visitado[nodo] = true;
    for (int conexion: listaAdyacencia[nodo]) {
        if (!visitado[conexion])
            dfsCFC(conexion, visitado, S, listaAdyacencia);
    }
    S.push(nodo);
}

int componentesFuertementeConexas(vector<vector<int>> & listaAdyacencia,
↪  vector<int> & componente) {
    int N = (int) listaAdyacencia.size();
    // Paso 1: DFS guardando cada nodo tras visitar todas sus conexiones de mayor
    ↪  profundidad
    stack<int> S;
    vector<bool> visitado(N, false);
    for (int i = 0; i < N; i++) {
        if (!visitado[i])
            dfsCFC(i, visitado, S, listaAdyacencia);
    }
    // Paso 2: Invertir el grafo
    vector<vector<int>> grafoInv(N, vector<int>());
    for (int i = 0; i < N; i++) {
        for (int u: listaAdyacencia[i]) {
            grafoInv[u].push_back(i);
        }
    }
    // Paso 3: Obtener las componentes
    int numeroComponentes = 0, nodoActual;
    while (!S.empty()) {
        nodoActual = S.top();
        S.pop();
        if (componente[nodoActual] == -1) {
            dfsComponente(nodoActual, numeroComponentes, componente, grafoInv);
            numeroComponentes++;
        }
    }
    return numeroComponentes;
}

void resolver(vector<vector<int>> & listaAdyacencia) {
    int N = (int) listaAdyacencia.size();
    // con la función ya vista, encontramos las componentes fuertemente conexas
    // sabremos el total de CFCs y a cuál pertenece cada nodo
    vector<int> componente(N, -1);
```

```
59      int numeroComponentes = componentesFuertementeConexas(listaAdyacencia,
        ↪  componente);
60      // calculamos el tamaño de cada CFC
61      vector<int> tamanoComponente(numeroComponentes, 0);
62      for (int i = 0; i < N; i++)
63          tamanoComponente[componente[i]]++;
64      // encontramos el grado de cada CFC
65      vector<int> gradoComponente(numeroComponentes, 0);
66      // construimos una lista de adyacencia entre componentes
67      // lo hacemos con un set para guardar solo una arista entre cada pareja de
        ↪  componentes conectadas
68      // esto nos permitirá encontrar el grado de cada componente correctamente
69      vector<unordered_set<int>> adyCFC(numeroComponentes, unordered_set<int>());
70      for (int i = 0; i < N; i++) {
71          for (int u: listaAdyacencia[i]) {
72              if (componente[i] == componente[u])
73                  continue;
74              if (adyCFC[componente[i]].find(componente[u]) ==
                ↪  adyCFC[componente[i]].end()) {
75                  gradoComponente[componente[u]]++;
76                  adyCFC[componente[i]].insert(componente[u]);
77              }
78          }
79      }
80      // inicializamos para cada componente el número de llaves perdidas necesarias,
81      // que de momento será igual al tamaño de la componente
82      vector<int> perdidasNecesarias(numeroComponentes);
83      for (int i = 0; i < numeroComponentes; i++)
84          perdidasNecesarias[i] = tamanoComponente[i];
85      // ahora, modificaremos el algoritmo de ordenamiento topológico
86      // de forma que visitaremos las CFC en orden, ya que sabemos que el grafo
        ↪  entre componentes tiene que ser DAG
87      // así, cuando lleguemos a una CFC, habremos pasado ya (y contabilizado las
        ↪  pérdidas necesarias) por los nodos anteriores
88      queue<int> colaBFS;
89      for (int i = 0; i < numeroComponentes; i++) {
90          if (gradoComponente[i] == 0)
91              colaBFS.push(i);
92      }
93      vector<bool> visitado(numeroComponentes, false);
94      while(!colaBFS.empty()) {
95          int cfcActual = colaBFS.front();
96          colaBFS.pop();
97          if (visitado[cfcActual])
98              continue;
99          visitado[cfcActual] = true;
100         for (unordered_set<int>::iterator it = adyCFC[cfcActual].begin(); it !=
            ↪  adyCFC[cfcActual].end(); it++) {
101             int vecino = *it;
102             // añadimos las pérdidas del padre a su vecino
103             perdidasNecesarias[vecino] += perdidasNecesarias[cfcActual];
104             // quitamos la arista que los une
105             gradoComponente[vecino]--;
106             if (gradoComponente[vecino] == 0)
107                 colaBFS.push(vecino); // añadimos el vecino a la cola si procede
```

```
108             }
109         }
110         // imprimimos para cada nodo las pérdidas de su CFC, ya que son las mismas
            ↪ para todos los nodos de la CFC
111         for (int i = 0; i < N; i++)
112             cout << perdidasNecesarias[componente[i]] << ' ';
113         cout << '\n';
114     }
115
116     int main() {
117         ios::sync_with_stdio(0);
118         cin.tie(NULL);
119         int N, a;
120         vector<vector<int>> listaAdyacencia;
121         while(cin >> N) {
122             listaAdyacencia.assign(N, vector<int>());
123             for (int i = 0; i < N; i++) {
124                 // añadimos una arista del amigo que tiene la llave al inquilino de la
                    ↪ habitación
125                 cin >> a;
126                 listaAdyacencia[a].push_back(i);
127             }
128             resolver(listaAdyacencia);
129         }
130         return 0;
131     }
```

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;
                }
        }

        // damos un índice a cada raíz
        unordered_map<int, int> raices;
        int c = 0;
        for (int i = 0; i < N; i++) {
                if (padre[i] == i)
                        raices[i] = c++;
        }

        // guardamos las componentes conexas
        vector<vector<int>> CCs(numCCs);
        for (int i = 0; i < N; i++)
                CCs[raices[raiz(i, padre)]].push_back(i);

        int CCtamano, u, v, n, raiz_u, raiz_v;

        vector<vector<int>> resultadoFinal(N, vector<int>(N, 0)), serpientes;

        for (int i = 0; i < numCCs; i++) {
                CCtamano = (int) CCs[i].size();
                if (CCtamano == 1) continue;

                // encontramos las "serpientes" de esta componente conexa
```

```
                n = CCtamano;
                for (auto node: CCs[i]) {
                        padre[node] = node;
                        alturaAprox[node] = 0;
                }
                for (int j = 0; j < CCtamano; j++) {
                        u = CCs[i][j];
                        raiz_u = raiz(u, padre);
                        for (int k = j+1; k < CCtamano; k++) {
                                v = CCs[i][k];
                                if (p[u][v] == 1)
                                        unirConjuntos(u, v, padre, alturaAprox,
                                        ↪  n);
                                else if (raiz_u == raiz(v, padre))
                                        return 0;
                        }
                }

                // nos aseguramos de que no hay incongruencias en esta componente
                ↪  conexa
                if (n == 2) return 0;

                raices.clear();
                c = 0;
                for (int j = 0; j < CCtamano; j++) {
                        u = CCs[i][j];
                        raiz_u = raiz(u, padre);
                        if (raiz_u == u)
                                raices[u] = c++;
                        for (int k = j+1; k < CCtamano; k++) {
                                v = CCs[i][k];
                                raiz_v = raiz(v, padre);
                                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
```

```
120                 for (int j = 0; j < n-1; j++)
121                         resultadoFinal[serpientes[j][0]][serpientes[j+1][0]] =
                            ↪  resultadoFinal[serpientes[j+1][0]][serpientes[j][0]] =
                            ↪  1;
122
123                 if (n > 1)
124                         resultadoFinal[serpientes[0][0]][serpientes[n-1][0]] =
                            ↪  resultadoFinal[serpientes[n-1][0]][serpientes[0][0]] =
                            ↪  1;
125         }
126         build(resultadoFinal);
127         return 1;
128 }
```

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;
}
```