Soluciones OIFem 2020
Entrenamiento 7

Componentes conexas

Teoría

- Componentes conexas

Solución

Este problema es una aplicación directa del algoritmo visto en clase, por lo que puede resolverse con el código de los apuntes. Simplemente hay que quitar el `cout` de la función de DFS y retornar el número de componentes.

Código

C++

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

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);
    }
}

int componentesConexas(vector<vector<int>> & listaAdyacencia) {
    int N = (int) listaAdyacencia.size();
    vector<int> componente(N, -1);
    int numeroComponentes = 0;
    for (int i = 0; i < N; i++) {
        if (componente[i] == -1) {
            numeroComponentes++;
            dfsComponente(i, numeroComponentes, componente, listaAdyacencia);
```

```
        }
    }
    return numeroComponentes;
}

int main() {
    int T, N, M, a, b;
    vector<vector<int>> listaAdyacencia;
    cin >> T;
    while(T--) {
        cin >> N >> M;
        listaAdyacencia.assign(N, vector<int>());
        while(M--) {
            cin >> a >> b;
            listaAdyacencia[a].push_back(b);
            listaAdyacencia[b].push_back(a);
        }
        cout << componentesConexas(listaAdyacencia) << '\n';
    }
    return 0;
}
```

Número de Erdös

Teoría

- BFS

Solución

Tal y como vimos en clase, usamos una llamada a BFS para guardar la distancia mínima entre Erdös y el resto de nodos. Para un nodo u, esta distancia mínima será 1 más la distancia mínima entre Erdös y v para todo vecino v de u, ya que hay que sumar la arista que conecta v y u. Usamos mapas para convertir entre strings y enteros, además de para guardar la lista de adyacencia, ya que no sabemos cuántos autores va a haber en total en cada caso.

Código

C++

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

vector<int> numeroErdos(int indiceErdos, unordered_map<int, vector<int>> &
    listaAdyacencia) {
    int N = (int) listaAdyacencia.size();
    vector<int> distErdos(N, -1);
    distErdos[indiceErdos] = 0;
    queue<int> indicesBFS;
    indicesBFS.push(indiceErdos);
    int nodo;
    while(indicesBFS.size()) {
        nodo = indicesBFS.front();
        indicesBFS.pop();
        for (int conexion: listaAdyacencia[nodo]) {
            if (distErdos[conexion] == -1 || distErdos[conexion] >
                distErdos[nodo]+1) {
                distErdos[conexion] = distErdos[nodo]+1;
                indicesBFS.push(conexion);
            }
        }
    }
    return distErdos;
}


int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int T, numeroArticulos, numeroAutores, indiceErdos;
```

```
    unordered_map<string, int> diccionario;
    unordered_map<int, vector<int>> listaAdyacencia;
    vector<int> autores;
    vector<string> autorNombres;
    string nombre;
    cin >> T;
    while(T--) {
        cin >> numeroArticulos;
        listaAdyacencia.clear();
        indiceErdos = -1;
        autorNombres.clear();
        diccionario.clear();
        while(numeroArticulos--) {
            cin >> numeroAutores;
            autores.clear();
            for (int i = 0; i < numeroAutores; i++) {
                cin >> nombre;
                if (diccionario.find(nombre) == diccionario.end()) {
                    diccionario[nombre] = (int) diccionario.size();
                    autorNombres.push_back(nombre);
                    listaAdyacencia[diccionario[nombre]] = vector<int>();
                    if (nombre == "erdos")
                        indiceErdos = (int) diccionario.size()-1;
                }
                autores.push_back(diccionario[nombre]);
            }
            for (int i = 0; i < numeroAutores; i++) {
                for (int j = i+1; j < numeroAutores; j++) {
                    // i -> j
                    listaAdyacencia[autores[i]].push_back(autores[j]);
                    // j -> i
                    listaAdyacencia[autores[j]].push_back(autores[i]);
                }
            }
        }
        sort(autorNombres.begin(), autorNombres.end());
        if (indiceErdos == -1) {
            for (int i = 0; i < (int) autorNombres.size(); i++) {
                cout << autorNombres[i] << " -1\n";
            }
        } else {
            vector<int> distErdos = numeroErdos(indiceErdos, listaAdyacencia);
            for (int i = 0; i < (int) autorNombres.size(); i++) {
                cout << autorNombres[i] << " " <<
                ↪  distErdos[diccionario[autorNombres[i]]] << '\n';
            }
        }
    }
    return 0;
}
```

Componentes fuertemente conexas

Teoría

- Componentes fuertemente conexas

Solución

Basta con usar el algoritmo y el código explicados en los apuntes.

Código

C++

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

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);
}

void componentesFuertementeConexas(vector<vector<int>> & listaAdyacencia) {
    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>());
```

```cpp
    for (int i = 0; i < N; i++) {
        for (int u: listaAdyacencia[i]) {
            grafoInv[u].push_back(i);
        }
    }
    // Paso 3: Obtener las componentes
    vector<int> componente(N, -1);
    int numeroComponentes = 0, nodoActual;
    while (!S.empty()) {
        nodoActual = S.top();
        S.pop();
        if (componente[nodoActual] == -1) {
            numeroComponentes++;
            dfsComponente(nodoActual, numeroComponentes, componente, grafoInv);
        }
    }
    cout << numeroComponentes << '\n';
}

int main() {
    int T, N, M, a, b;
    vector<vector<int>> listaAdyacencia;
    cin >> T;
    while(T--) {
        cin >> N >> M;
        listaAdyacencia.assign(N, vector<int>());
        while(M--) {
            cin >> a >> b;
            listaAdyacencia[a].push_back(b);
        }
        componentesFuertementeConexas(listaAdyacencia);
    }
    return 0;
}
```

Aristas y vértices de corte

Teoría

- Aristas y vértices de corte

Solución

De nuevo, este problema es una aplicación directa del algoritmo aprendido en el addendum.

Código

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> listaAdyacencia;
vector<int> ordenDescubrimiento, bajo; // empiezan como (N, -1)
vector<bool> nodoCorte; // empieza como (N, false)
int contador, N, vertices, aristas, hijos;

void articulacion(int u, int par) {
    ordenDescubrimiento[u] = bajo[u] = contador++;
    for (int v: listaAdyacencia[u]) {
        if (ordenDescubrimiento[v] == -1) {
            articulacion(v, u); // recurrimos
            if (par == -1)
                hijos++; // v es hijo de u, la raíz, en el árbol de DFS
            if (bajo[v] > ordenDescubrimiento[u]) {
                // no hay forma de llegar a ningún ancestor de u desde v salvo por la arista u-v
                nodoCorte[u] = true;
                // u -> nodo de corte
                aristas++;
                // u-v -> arista corte
            }
            else if (bajo[v] == ordenDescubrimiento[u]) {
                // no hay forma de llegar de vuelta a ningún ancestor de u desde el subárbol de v salvo a través de u
                // pero sí a u (y por tanto la arista u-v no es de corte pero u sí)
                nodoCorte[u] = true;
                // u -> nodo de corte
            }
            bajo[u] = min(bajo[u], bajo[v]);
        } else if (v != par)
            bajo[u] = min(bajo[u], ordenDescubrimiento[v]);
    }
}
```

```
37
38  void corte() {
39      // aquí contamos con los valores de N y la lista de adyacencia ya rellenos
40      contador = 0;
41      aristas = 0;
42      ordenDescubrimiento.assign(N, -1); // ordenDescubrimiento guarda el orden en
        ↪  el que el DFS encuentra cada nodo
43      bajo.assign(N, -1); // bajo[u] guarda el orden de descubrimiento más pequeño
        ↪  que pertenece a un nodo alcanzable desde el subárbol de u (incluyendo el
        ↪  de u)
44      nodoCorte.assign(N, false);
45      for (int i = 0; i < N; i++) {
46          hijos = 0;
47          if (ordenDescubrimiento[i] == -1) {
48              // aún no hemos explorado esta componente -> tratamos al nodo i como
                ↪  su raíz
49              articulacion(i, -1); // hacemos DFS
50              nodoCorte[i] = hijos > 1; // i será vértice de corte solo en el caso
                ↪  de que tenga más de un hijo en el árbol DFS
51          }
52      }
53      vertices = count(nodoCorte.begin(), nodoCorte.end(), true);
54      cout << vertices << " " << aristas << '\n';
55  }
56
57  int main() {
58      ios::sync_with_stdio(false);
59      cin.tie(NULL);
60      int T, M, a, b;
61      cin >> T;
62      while(T--) {
63          cin >> N >> M;
64          listaAdyacencia.assign(N, vector<int>());
65          while(M--) {
66              cin >> a >> b;
67              listaAdyacencia[a].push_back(b);
68              listaAdyacencia[b].push_back(a);
69          }
70          corte();
71      }
72      return 0;
73  }
```

Moocast 1

Teoría

- Componentes conexas

Solución

En este problema, invocaremos DFS con cada nodo de 0 a $N-1$ como raíz y guardaremos el número de nodos alcanzados con dicha invocación, llevando la cuenta del máximo hasta el momento. Este número de nodos se encuentra contando el número de casillas con valor de `true` en el vector `visitado`. Consideraremos que hay una arista desde el nodo u hasta el nodo v si la potencia del walkie-talkie de u al cuadrado es mayor o igual a la distancia al cuadrado entre los dos nodos. Hay que tener en cuenta que este es un grafo dirigido.

Código

C++

```cpp
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

int dist(pair<int, int> A, pair<int, int> B) {
        return (A.first-B.first)*(A.first-B.first) +
        ↪  (A.second-B.second)*(A.second-B.second);
}

void dfs(int S, vector<bool> & visitado, vector<vector<int>> & listaAdyacencia) {
        visitado[S] = true;
        for (int v: listaAdyacencia[S]) {
                if (!visitado[v])
                        dfs(v, visitado, listaAdyacencia);
        }
}

int moocast(vector<pair<int, int>> & coords, vector<int> & power) {
        int N = (int) power.size();
        vector<vector<int>> listaAdyacencia(N, vector<int>());
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if (i == j) continue;
                        if (dist(coords[i], coords[j]) <= power[i]*power[i]) {
                                listaAdyacencia[i].push_back(j);
                        }
                }
        }
        vector<bool> visitado;
        int respuesta = 0;
        for (int i = 0; i < N; i++) {
                visitado.assign(N, false);
                dfs(i, visitado, listaAdyacencia);
```

```
                respuesta = max(respuesta, (int) count(visitado.begin(),
                ↪  visitado.end(), true));
        }
        return respuesta;
}


int main() {
        ifstream fin;
        fin.open("moocast.in");
        ofstream fout;
        fout.open("moocast.out");
        int N;
        fin >> N;
        vector<pair<int, int>> coords(N, pair<int, int>());
        vector<int> power(N);
        for (int i = 0; i < N; i++) {
                fin >> coords[i].first >> coords[i].second >> power[i];
        }
        fout << moocast(coords, power) << '\n';
        fin.close();
        fout.close();
        return 0;
}
```

Moocast 2

Teoría

- Método de bisección
- DFS

Solución

Usamos el método de bisección para encontrar el mínimo valor D que deben gastar para estar todas comunicadas. Para cada valor de prueba, hacemos un DFS considerando que hay una arista entre dos nodos si la distancia entre ellos al cuadrado es menor o igual a D.

Extra: Sin embargo, este DFS comprueba todos los nodos i cada vez que visita un nodo u. Se puede hacer de forma más rápida este problema con cualquiera de las dos técnicas que veremos la clase que viene: árboles recubridores mínimos (MST) o conjuntos distintos (UFDS). ¿Sabrías programar estas dos variaciones?

Código

C++

```cpp
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

int N;
vector<vector<int>> dist;

void dfs(int u, int D, vector<bool> & visitado) {
        visitado[u] = true;
        for (int i = 0; i < N; i++) {
                if (!visitado[i] && dist[u][i] <= D)
                        dfs(i, D, visitado);
        }
}

bool esPosible(int D) {
        vector<bool> visitado(N, false);
        dfs(0, D, visitado);
        return (int) count(visitado.begin(), visitado.end(), true) == N;
}

int minimoD() {
        int lo = 1, hi = 2e9, mid, respuesta = 2e9;
        while (lo <= hi) {
                mid = lo + (hi - lo) / 2;
                if (esPosible(mid)) {
                        respuesta = mid;
                        hi = mid - 1;
                } else
                        lo = mid + 1;
```

```
        }
        return respuesta;
}

int main() {
        ifstream fin;
        ofstream fout;
        fin.open("moocast.in");
        fout.open("moocast.out");
        fin >> N;
        vector<int> x(N), y(N);
        for (int i = 0; i < N; i++) fin >> x[i] >> y[i];
        dist.assign(N, vector<int>(N, 0));
        for (int i = 0; i < N; i++) {
                for (int j = i+1; j < N; j++) {
                        dist[i][j] = dist[j][i] = (x[i] - x[j])*(x[i] - x[j]) +
                        ↪  (y[i] - y[j])*(y[i] - y[j]);
                }
        }
        fout << minimoD() << '\n';
        fin.close();
        fout.close();
        return 0;
}
```

Topological sorting

Teoría

- Ordenamiento topológico

Solución

Para comprobar si el orden dado es correcto, seguimos un algoritmo muy similar al que usamos para crear un ordenamiento propio. Un ordenamiento es correcto si, cada vez que llegamos a un nodo dentro de la lista, este tiene 0 aristas apuntando a él (grado = 0). Por lo tanto, vamos nodo a nodo asegurándonos de que su grado es 0 y eliminando las aristas que salen de este. Si llegamos al final del orden y no ha habido ninguna contradicción (un nodo con grado superior a 0), sabremos que el orden es correcto.

Código

C++

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

bool ordenamientoTopologico(vector<vector<int>> & listaAdyacencia, vector<int> & propuesta) {
        int N = (int) listaAdyacencia.size();
        vector<int> grado(N, 0);
        for (int i = 0; i < N; i++) {
                for (int vecino: listaAdyacencia[i])
                        grado[vecino]++;
        }
        for (int v: propuesta) {
                // comprobamos que no haya contradicciones
                if (grado[v] > 0)
                        return false;
                // eliminamos las aristas que salen de v
                for (int vecino: listaAdyacencia[v]) grado[vecino]--;
        }
        return true;
}

int main() {
        int N, M, a, b;
        cin >> N >> M;
        vector<vector<int>> listaAdyacencia(N, vector<int>());
        vector<int> ordenPropuesto(N);
        while(M--) {
                cin >> a >> b;
                listaAdyacencia[a-1].push_back(b-1);
        }
        for (int i = 0; i < N; i++) {
                cin >> ordenPropuesto[i];
                ordenPropuesto[i]--;
```

```
35          }
36          if(ordenamientoTopologico(listaAdyacencia, ordenPropuesto))
37                  cout << "YES\n";
38          else
39                  cout << "NO\n";
40          return 0;
41  }
```

Dirigiendo las carreteras de Grafolandia

Teoría

- Aristas y vértices de corte
- DFS

Solución

Las únicas aristas cuya dirección no podremos fijar serán las aristas de corte (mirad el addendum para saber más). Hacemos, por tanto, el DFS modificado que usamos para encontrar aristas de corte y alteramos la dirección del resto de aristas, designando su dirección como la encontrada por el DFS. O sea, si una arista $u - v$ es de corte, la dejamos tal cual, pero si no lo es y en nuestro DFS u es padre de v, designamos su dirección como $u - v$. Vamos llevando la cuenta del número de carreteras cuya dirección hemos cambiado. Si al final del DFS hemos cambiado al menos k carreteras de dirección, imprimimos las primeras k cambiadas. Si no, diremos que no es posible cumplir con los requisitos.

Código

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> listaAdyacencia;
vector<int> ordenDescubrimiento, bajo; // empiezan como (N, -1)
vector<bool> nodoCorte; // empieza como (N, false)
int contador, N;
vector<pair<int, int>> carreteras;

void articulacion(int u, int par) {
    ordenDescubrimiento[u] = bajo[u] = contador++;
    for (int v: listaAdyacencia[u]) {
        if (ordenDescubrimiento[v] == -1) {
            articulacion(v, u);
            bajo[u] = min(bajo[u], bajo[v]);
            if (bajo[v] <= ordenDescubrimiento[u])
                carreteras.push_back(make_pair(u, v)); // u-v no es arista de
                ↪ corte
        } else if (v != par) {
            bajo[u] = min(bajo[u], ordenDescubrimiento[v]);
            if (ordenDescubrimiento[v] < ordenDescubrimiento[u])
                carreteras.push_back(make_pair(u, v)); // u-v no es arista de
                ↪ corte
        }
    }
}

void corte(int K) {
    // aquí contamos con los valores de N y la lista de adyacencia ya rellenos
```

```
31      contador = 0;
32      ordenDescubrimiento.assign(N, -1); // ordenDescubrimiento guarda el orden en
        ↪  el que el DFS encuentra cada nodo
33      bajo.assign(N, -1); // bajo[u] guarda el orden de descubrimiento más pequeño
        ↪  que pertenece a un nodo alcanzable desde el subárbol de u (incluyendo el
        ↪  de u)
34      nodoCorte.assign(N, false);
35      carreteras.clear();
36      for (int i = 0; i < N; i++) {
37          if (ordenDescubrimiento[i] == -1) {
38              // aún no hemos explorado esta componente -> tratamos al nodo i como
                ↪  su raíz
39              articulacion(i, -1); // hacemos DFS
40          }
41      }
42      if ((int) carreteras.size() >= K) {
43          // es posible
44          cout << "SI\n";
45          // imprimimos las primeras K carreteras que cambiamos de dirección
46          for (int i = 0; i < K; i++)
47              cout << carreteras[i].first << ' ' << carreteras[i].second << '\n';
48      } else cout << "NO\n";
49  }
50
51  int main() {
52      ios::sync_with_stdio(false);
53      cin.tie(NULL);
54      int T, M, a, b, K;
55      cin >> T;
56      while(T--) {
57          cin >> N >> M >> K;
58          listaAdyacencia.assign(N, vector<int>());
59          while(M--) {
60              cin >> a >> b;
61              listaAdyacencia[a].push_back(b);
62              listaAdyacencia[b].push_back(a);
63          }
64          corte(K);
65      }
66      return 0;
67  }
```