Soluciones OIFem 2020
Entrenamiento 9

Building Teams

Teoría

- Grafos bipartitos

Solución

Este problema es una aplicación directa del código visto en clase.

Código

C++

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

bool esPosible;

void dfs(int nodo, vector<vector<int>> & listaAdyacencia, vector<int> & equipo) {
        if (!esPosible)
                return;
        for (int amigo: listaAdyacencia[nodo]) {
                if (equipo[amigo] == -1) {
                        equipo[amigo] = 1 - equipo[nodo];
                        dfs(amigo, listaAdyacencia, equipo);
                } else if (equipo[amigo] == equipo[nodo]) {
                        // imposible!!
                        esPosible = false;
                        return;
                }
        }
}


void partirClase(vector<vector<int>> & listaAdyacencia) {
        int N = (int) listaAdyacencia.size();
```

```
        vector<int> equipo(N, -1);
        esPosible = true;
        for (int i = 0; i < N; i++) {
                if (equipo[i] == -1) {
                        equipo[i] = 0;
                        dfs(i, listaAdyacencia, equipo);
                        if (!esPosible)
                                break;
                }
        }
        if (esPosible) {
                for (int i = 0; i < N; i++)
                        if (equipo[i] == 0) equipo[i] = 2;
                cout << equipo[0];
                for (int i = 1; i < N; i++)
                        cout << " " << equipo[i];
                cout << '\n';
        } else cout << "IMPOSSIBLE\n";
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int N, M, a, b;
    cin >> N >> M;
    vector<vector<int>> listaAdyacencia(N);
    while(M--) {
        cin >> a >> b;
        a--;
        b--;
        listaAdyacencia[a].emplace_back(b);
        listaAdyacencia[b].emplace_back(a);
    }
    partirClase(listaAdyacencia);
    return 0;
}
```

Patinete eléctrico

Teoría

- Dijkstra

Solución

Usamos el algoritmo de Dijkstra partiendo del nodo 0 y retornamos la distancia del nodo 0 al nodo 1.

Código

C++

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

int Dijkstra(vector<vector<pair<int, int>>> & adjList) {
    int N = (int) adjList.size(), inf = 1e9;
    vector<int> dist(N, inf);
    dist[0] = 0;
    vector<int> visited(N, 0);
    priority_queue<pair<int, int>> pq;
    pq.push({0, 0});
    while(!pq.empty()) {
        pair<int, int> cur = pq.top();
        pq.pop();
        int v = cur.second, cost = -cur.first;
        if (visited[v])
            continue;
        visited[v] = 1;
        dist[v] = cost;
        for (auto edge: adjList[v]) {
            if (!visited[edge.second])
                pq.push(make_pair(-dist[v] - edge.first, edge.second));
        }
    }
    return dist[1];
}


int main() {
    int t, n, m, a, b, c;
    vector<vector<pair<int, int>>> adjList;
    cin >> t;
    while(t--) {
        cin >> n >> m;
```

```
        adjList = vector<vector<pair<int, int>>>(n);
        while(m--) {
            cin >> a >> b >> c;
            adjList[a].push_back(make_pair(c, b));
            adjList[b].push_back(make_pair(c, a));
        }
        cout << Dijkstra(adjList) << '\n';
    }
    return 0;
}
```

Wormholes

Teoría

- Bellman Ford

Solución

Este problema es una aplicación directa del algoritmo de Bellman Ford visto en clase.

Código

C++

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

bool bellmanFord(int raiz, vector<vector<pair<int, int>>> & listaAdyacencia) {
    // las parejas de la lista son {nodo, coste}
    int N = (int) listaAdyacencia.size();
    vector<int> distancia(N, 1e9);
    distancia[raiz] = 0;
    // relajamos costes N-1 veces
    for (int i = 0; i < N - 1; i++)
        for (int u = 0; u < N; u++)
            for (pair<int, int> vecino: listaAdyacencia[u]) {
                int v = vecino.first, coste = vecino.second;
                distancia[v] = min(distancia[v], distancia[u] + coste);
            }
    for (int u = 0; u < N; u++)
        for (pair<int, int> vecino: listaAdyacencia[u]) {
            int v = vecino.first, coste = vecino.second;
            if(distancia[u] + coste < distancia[v])
                return true; // hemos encontrado un ciclo negativo
        }
    return false; // no hay ningún ciclo negativo
}

int main() {
        int T, N, M, A, B, C;
        vector<vector<pair<int, int>>> adj_list;
        cin >> T;
        while(T--) {
                cin >> N >> M;
                adj_list = vector<vector<pair<int, int>>>(N);
                for (int m = 0; m < M; m++) {
                        cin >> A >> B >> C;
                        adj_list[A].push_back({B, C});
                }
                if (bellmanFord(0, adj_list))
                    cout << "possible\n";
                else
                    cout << "not possible\n";
```

```
41          }
42          return 0;
43  }
```

Commandos

Teoría

- Floyd-Warshall's

Solución

Este problema es una aplicación directa del algoritmo aprendido.

Código

C++

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

vector<vector<int>> adj_matrix;

int main() {
        int T, N, R, u, v, S, D, res;
        vector<int> empt;
        cin >> T;
        for (int cas = 1; cas <= T; cas++) {
                cin >> N >> R;
                empt.assign(N, 10000000);
                adj_matrix.assign(N, empt);
                for (int i = 0; i < N; i++) adj_matrix[i][i] = 0;
                for (int r = 0; r < R; r++) {
                        cin >> u >> v;
                        adj_matrix[u][v] = 1;
                        adj_matrix[v][u] = 1;
                }
                cin >> S >> D;
                for (int k = 0; k < N; k++) {
                        for (int i = 0; i < N; i++) {
                                for (int j = 0; j < N; j++) {
                                        adj_matrix[i][j] = min(adj_matrix[i][j],
                                        ↪  adj_matrix[i][k] + adj_matrix[k][j]);
                                }
                        }
                }
                res = 0;
                for (int i = 0; i < N; i++) {
                        res = max(adj_matrix[S][i] + adj_matrix[i][D], res);
                }
                cout << "Case " << cas << ": " << res << endl;
        }
        return 0;
}
```

Transporte carísimo

Teoría

- Dijkstra

Solución

Hacemos Dijkstra ordenando primero en base al coste base y luego al coste con impuestos en el caso de empate. Así, guardamos en todo momento el récord de ambos.

Código

C++

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

vector<vector<pair<ll, ll>>> listaAdyacencia;
vector<ll> costeBase, costeConImpuestos;
ll plusINF = 2e18;
int N;

void Dijkstra() {
    costeBase.assign(N, plusINF);
    costeConImpuestos.assign(N, plusINF);
    costeBase[0] = 0;
    costeConImpuestos[0] = 0;
    priority_queue<pair<pair<ll, ll>, ll>> pq;
    pq.push(make_pair(make_pair(0, 0), 0)); // costeBase, costeConImpuestos, v
    ll costeBase_u, costeConImpuestos_u, coste_u_v, u, v;
    while(!pq.empty()) {
        costeBase_u = -pq.top().first.first;
        costeConImpuestos_u = -pq.top().first.second;
        u = pq.top().second;
        pq.pop();
        if (costeBase_u > costeBase[u] && costeConImpuestos_u >
        ↪  costeConImpuestos[u])
            continue;
        for (pair <ll, ll> cur_v: listaAdyacencia[u]) {
            v = cur_v.first;
            coste_u_v = cur_v.second;
            if (costeConImpuestos_u + coste_u_v + costeBase_u <
            ↪  costeConImpuestos[v]) {
                costeConImpuestos[v] = costeConImpuestos_u + coste_u_v
                ↪  +costeBase_u;
                costeBase[v] = costeBase_u+coste_u_v;
                pq.push(make_pair(make_pair(-costeBase[v], -costeConImpuestos[v]),
                ↪  v));
```

```
            } else if (costeBase_u + coste_u_v < costeBase[v]) {
                pq.push(make_pair(make_pair(-(costeBase_u + coste_u_v),
                ↪  -(costeConImpuestos_u + coste_u_v + costeBase_u)), v));
            }
        }
    }
    for (int D = 1; D < N; D++) {
        if (costeConImpuestos[D] == plusINF) cout << "-1 ";
        else cout << costeConImpuestos[D] << " ";
    }
    cout << "\n";
}

int main() {
    int T, M;
    ll C, A, B;
    vector<pair<ll, ll>> empt;
    cin >> T;
    while(T--) {
        cin >> N >> M;
        listaAdyacencia.assign(N, empt);
        while(M--) {
            cin >> A >> B >> C;
            listaAdyacencia[A].push_back({B, C});
        }
        Dijkstra();
    }
    return 0;
}
```

Caza de brujas

Teoría

- Grafos bipartitos

Solución

Este problema lo vimos en clase. Adjunto aquí mi código.

Código

C++

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

bool esPosible;
int equipo0, equipo1;

void dfs(int nodo, vector<vector<int>> & amigos, vector<vector<int>> & enemigos,
 ↪  vector<int> & equipo) {
    if (!esPosible)
        return;

    if (equipo[nodo] == 0)
        equipo0++;
    else
        equipo1++;

    for (int amigo: amigos[nodo]) {
        if (equipo[amigo] == 1-equipo[nodo]) {
            esPosible = false;
            return;
        } else if (equipo[amigo] == -1) {
            equipo[amigo] = equipo[nodo];
            dfs(amigo, amigos, enemigos, equipo);
        }
    }
    for (int enemigo: enemigos[nodo]) {
        if (equipo[enemigo] == equipo[nodo]) {
            esPosible = false;
            return;
        } else if (equipo[enemigo] == -1) {
            equipo[enemigo] = 1 - equipo[nodo];
            dfs(enemigo, amigos, enemigos, equipo);
        }
    }
}
```

```cpp

int brujas(vector<vector<int>> & amigos, vector<vector<int>> & enemigos) {
    int N = (int) amigos.size();
    vector<int> equipo(N, -1);
    int totalBrujas = 0;
    esPosible = true;
    for (int i = 0; i < N; i++) {
        if (equipo[i] == -1) {
            equipo0 = equipo1 = 0;
            equipo[i] = 0;
            dfs(i, amigos, enemigos, equipo);
            if (!esPosible)
                return -1;
            totalBrujas += min(equipo0, equipo1);
        }
    }
    return totalBrujas;
}

int main() {
    int T, N, M, a, b, r;
    vector<vector<int>> enemigos, amigos;
    cin >> T;
    while(T--) {
        cin >> N >> M;
        enemigos = vector<vector<int>>(N);
        amigos = vector<vector<int>>(N);
        while(M--) {
            cin >> r >> a >> b;
            if (r == 1) {
                amigos[a].push_back(b);
                amigos[b].push_back(a);
            } else {
                enemigos[a].push_back(b);
                enemigos[b].push_back(a);
            }
        }
        cout << brujas(amigos, enemigos) << '\n';
    }
    return 0;
}
```

La pachanga

Teoría

- Grafos bipartitos
- Bisección

Solución

Para este problema, usaremos el método de bisección. Para cada posible k, probamos a ver si se puede construir un grafo bipartito. Seguimos hasta encontrar el mínimo k que cumpla esto.

Código

C++

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

vector<vector<pair<int, int>>> conexiones; // conexiones[jugador1] = [(jugador2, odio2), (jugador3, odio3)...]
vector<int> equipo;
/* equipo[jugador] = -1 -> equipo no asignado
equipo[jugador] = 0 -> jugador está en el primer equipo
equipo[jugador] = 1 -> el jugador está en el segundo equipo */

int N; // número de jugadores
bool esPosible; // usaremos esta variable como símbolo para cortar rápido la prueba de un valor de k si ya sabemos que no es posible


void designaEquipo(int jugador1, int kDePrueba) {
    // designaEquipo revisa las conexiones de jugador1 y pone en el equipo contrario a todo jugador al que odie con una intensidad superior a kDePrueba
    if (!esPosible) return; // paramos si ya sabemos que kDePrueba es demasiado pequeño
    int jugador2, odio;
    for (auto pareja: conexiones[jugador1]) {
        jugador2 = pareja.first;
        odio = pareja.second;
        if (odio > kDePrueba) {
            // jugador1 y jugador2 tienen que estar en diferentes equipos
            if (equipo[jugador2] == -1) {
                // jugador2 no tiene equipo asignado
                equipo[jugador2] = 1-equipo[jugador1]; // lo asignamos al equipo contrario
                designaEquipo(jugador2, kDePrueba); // repetimos el proceso para los jugadores conectados con jugador2
            } else if (equipo[jugador2] == equipo[jugador1]) {
                // jugador1 y jugador2 están en el mismo equipo -> kDePrueba es demasiado pequeño
```

```
                esPosible = false;
                return;
            }
        }
    }
}

bool divisionPosible(int kDePrueba) {
    // devuelve si es posible hacer una división con k <= kDePrueba
    esPosible = true;
    equipo.assign(N, -1); // empezamos la prueba con todos los jugadores por asignar
    for (int jugador = 0; jugador < N; jugador++) {
        // vamos jugador a jugador, viendo si ya tiene equipo asignado
        if (equipo[jugador] == -1) {
            // si está sin asignar, ningún jugador de su componente conexa (grupo de jugadores conectados entre sí por odios superiores a kDePrueba) ha sido asignado
            equipo[jugador] = 1; // por lo tanto, lo asignamos al primer equipo (sería igual asignarlo al segundo)
            designaEquipo(jugador, kDePrueba); // miramos a los jugadores que odia más de kDePrueba y los ponemos en el equipo contrario
            if (!esPosible) // no es posible dividir a la componente conexa de jugador
                return false;
        }
    }
    return true; // dividimos con éxito todas las componentes conexas
}

int main() {
    int T, M, A, B, O, k, bajo, medio, alto;
    cin >> T;
    while (T--) {
        cin >> N >> M;
        conexiones.assign(N, vector<pair<int, int>>());
        // creamos el grafo de conexiones como "adjacency list" con pesos
        while (M--) {
            cin >> A >> B >> O;
            conexiones[A-1].push_back({B-1, O});
            conexiones[B-1].push_back({A-1, O});
        }
        // para encontrar el valor de k más bajo, hacemos una búsqueda binaria entre todas las posibilidades
        k = 0;
        bajo = 0;
        alto = 1e9+10;
        while(bajo <= alto) {
            medio = bajo + (alto - bajo)/2;
            if (divisionPosible(medio)) {
                // se puede partir a los jugadores en dos equipos cuyo odio máximo no supera medio
                k = medio; // actualizamos k
                alto = medio-1;
            } else {
```

```
77                  // es imposible partir a los jugadores en dos equipos cuyo odio
                    ↪  máximo no supere medio
78                  bajo = medio+1;
79              }
80          }
81          cout << k << '\n';
82      }
83      return 0;
84  }
```

Crocodile

Teoría

- Dijkstra

Solución

Antes de leer la solución, prueba esta **demo** para ver si se te ocurre.

La solución es una variación de Dijkstra muy bien explicada en las soluciones oficiales del concurso del que fue extraído este problema: **IOI 2011**.

Graph

Teoría

- Grafos bipartitos

Solución

Este problema es una variación sobre el algoritmo para ver si un grafo es bipartito algo compleja y se esperaba que obtuvierais puntos parciales, pero programarla es buena práctica una vez leída la explicación. La solución oficial la explica bien: **BOI 2020**.