Soluciones OIFem II Nivel 2

Entrenamiento 6

Island Perimeter

Teoría

- Grafos implícitos

Solución

Este era un problema de implementación. Íbamos casilla a casilla viendo si tenía bordes y, en tal caso, incrementábamos un contador.

Código

```cpp
class Solution {
public:
    int islandPerimeter(vector<vector<int>>& grid) {
        int perim = 0;
        for (size_t i = 0; i < grid.size(); i++) {
            for (size_t j = 0; j < grid[0].size(); j++) {
                if (grid[i][j] == 1) {
                    if (i == 0 || grid[i-1][j] == 0)
                        perim++;
                    if (j == 0 || grid[i][j-1] == 0)
                        perim++;
                    if (i+1 == grid.size() || grid[i+1][j] == 0)
                        perim++;
                    if (j+1 == grid[0].size() || grid[i][j+1] == 0)
                        perim++;
                }
            }
        }
        return perim;
    }
};
```

Flood Fill

Teoría

- BFS

Solución

Iniciamos BFS en la celda que nos dan y vamos pintando todas las de su componente conexa.

Código

```cpp
class Solution {
public:
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {
        vector<vector<int>> filled = image;
        queue<pair<int, int>> Q;
        Q.push(make_pair(sr, sc));
        filled[sr][sc] = newColor;
        while(!Q.empty()) {
            int curRow = Q.front().first;
            int curCol = Q.front().second;
            Q.pop();
            if (curRow > 0 && image[curRow-1][curCol] == image[curRow][curCol] && filled[curRow-1][curCol] != newColor) {
                filled[curRow-1][curCol] = newColor;
                Q.push(make_pair(curRow-1, curCol));
            }
            if (curCol > 0 && image[curRow][curCol-1] == image[curRow][curCol] && filled[curRow][curCol-1] != newColor) {
                filled[curRow][curCol-1] = newColor;
                Q.push(make_pair(curRow, curCol-1));
            }
            if (curRow+1 < (int) image.size() && image[curRow+1][curCol] == image[curRow][curCol] && filled[curRow+1][curCol] != newColor) {
                filled[curRow+1][curCol] = newColor;
                Q.push(make_pair(curRow+1, curCol));
            }
            if (curCol+1 < (int) image[0].size() && image[curRow][curCol+1] == image[curRow][curCol] && filled[curRow][curCol+1] != newColor) {
                filled[curRow][curCol+1] = newColor;
                Q.push(make_pair(curRow, curCol+1));
            }
        }
        return filled;
    }
};
```

Surrounded Regions

Teoría

- Componentes conexas

Solución

Encontramos las componentes conexas del grafo de forma constructiva. En este código lo hice con UFDS (clase 6), pero se puede hacer también con una lista de adyacencia. Una vez construidas las componentes, basta con comprobar que no haya ninguna casilla de la componente que choque con un borde y, si esto se cumple, pintar la componente.

Código

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

    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;
        }
        if (isBord[rA] || isBord[rB]) isBord[rA] = isBord[rB] = 1;
    }

    void solve(vector<vector<char>> & board) {
        int R = (int) board.size();
        if (!R) return;
        int C = (int) board[0].size();
        par = rnk = isBord = vector<int>(R*C, 0);
        for (int i = 1; i < R*C; i++) par[i] = i;
        for (int i = 0; i < R*C; i++) {
            if (i % C == 0 || i % C == C-1 || i / C == 0 || i / C == R-1)
            ↪  isBord[root(i)] = 1;
            if (i - C >= 0 && board[i/C][i%C] == board[i/C - 1][i%C]) merge(i, i -
            ↪  C);
            if (i % C > 0 && board[i/C][i%C] == board[i/C][i%C - 1]) merge(i,
            ↪  i-1);
        }
        vector<int> pos;
        for (int i = 0; i < R*C; i++) {
```

```
            if (board[i/C][i%C] == 'O' && (!isBord[root(i)])) pos.push_back(i);
        }
        for (auto p: pos) board[p/C][p%C] = 'X';
    }
};
```

Number of Islands

Teoría

- DFS

Solución

Hacemos DFS para encontrar las componentes conexas y contamos cuántas son.

Código

```
class Solution {
public:
    vector<vector<int>> vis;
    int R, C;
    void DFS(int r, int c, vector<vector<char>> & grid) {
        vis[r][c] = 1;
        if (r && grid[r-1][c] == '1' && vis[r-1][c] == false) DFS(r-1, c, grid);
        if (c && grid[r][c-1] == '1' && vis[r][c-1] == false) DFS(r, c-1, grid);
        if (r < R-1 && grid[r+1][c] == '1' && vis[r+1][c] == false) DFS(r+1, c,
        ↪  grid);
        if (c < C-1 && grid[r][c+1] == '1' && vis[r][c+1] == false) DFS(r, c+1,
        ↪  grid);
    }

    int numIslands(vector<vector<char>> & grid) {
        R = (int) grid.size();
        if (R == 0) return 0;
        C = (int) grid[0].size();
        if (C == 0) return 0;
        int ans = 0;
        vis.assign(R, vector<int>(C, 0));
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                if (grid[i][j] == '1' && !vis[i][j]) {
                    ans++;
                    DFS(i, j, grid);
                }
            }
        }
        return ans;
    }
};
```

Redundant Connection

Teoría

- Componentes conexas

Solución

Vamos añadiendo aristas al grafo en el orden en el que viene la entrada hasta que vemos que una arista crea un ciclo. Retornamos esa arista.

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;
        }
    }
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        rnk.assign(1001, 0);
        par = vector<int>(1001);
        for (int i = 1; i < 1001; i++) par[i] = i;
        for (int i = 0; i < (int) edges.size(); i++) {
            if (root(edges[i][0]) == root(edges[i][1])) return edges[i];
            merge(edges[i][0], edges[i][1]);
        }
        return edges[0];
    }
};
```

Flower Planting With No Adjacent

Teoría

- DFS

Solución

Vamos descubriendo las componentes conexas usando DFS y a medida que descubrimos nuevas flores conectadas a flores ya pintadas, las pintamos de un color que aún no tienen sus vecinos.

Código

```cpp
class Solution {
public:
    vector<vector<int>> adj;
    vector<int> col;

    void dfs(int node) {
        vector<bool> valid(4, true);
        for (int neighbour: adj[node])
            if (col[neighbour] != -1)
                valid[col[neighbour]] = false;
        for (int i = 0; i < 4; i++) {
            if (valid[i]) {
                col[node] = i;
                break;
            }
        }
        for (int neighbour: adj[node])
            if (col[neighbour] == -1)
                dfs(neighbour);
    }

    vector<int> gardenNoAdj(int n, vector<vector<int>>& paths) {
        adj.assign(n, vector<int>());
        col.assign(n, -1);
        for (vector<int> path: paths) {
            adj[path[0]-1].push_back(path[1]-1);
            adj[path[1]-1].push_back(path[0]-1);
        }
        for (int i = 0; i < n; i++)
            if (col[i] == -1)
                dfs(i);
        for (int i = 0; i < n; i++)
            col[i]++;
        return col;
    }
};
```

Smallest String With Swaps

Teoría

- Componentes conexas
- Sorting

Solución

Encontramos las componentes conexas que forman los índices de la palabra y ordenamos cada componente en orden alfabético.

Código

```cpp
class Solution {
public:
    vector<int> par, rnk;
    int numSets;
    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 (rnk[rA] != rnk[rB]) {
            if (rnk[rA] < rnk[rB]) par[rA] = rB;
            else par[rB] = rA;
        } else {
            rnk[rB]++;
            par[rA] = rB;
        }
    }
    string smallestStringWithSwaps(string & s, vector<vector<int>> & pairs) {
        int n = (int) s.length();
        if (!n) return s;
        numSets = n;
        par = rnk = vector<int>(n, 0);
        for (int i = 1; i < n; i++) par[i] = i;
        for (int i = 0; i < (int) pairs.size(); i++) merge(pairs[i][0],
        ↪  pairs[i][1]);
        if (numSets == 1) {
            sort(s.begin(), s.end());
            return s;
        }
        map<int, vector<int>> pos;
        map<int, int> used;
        for (int i = 0; i < n; i++) pos[root(i)].push_back(s[i] - 'a');
        map<int, vector<int>>::iterator it = pos.begin();
        while(it != pos.end()) {
            used[it->first] = 0;
```

```
            sort(it->second.begin(), it->second.end());
            it++;
        }
        for (int i = 0; i < n; i++) {
            s[i] = 'a' + pos[root(i)][used[root(i)]];
            used[root(i)]++;
        }
        return s;
    }
};
```

Moocast (Silver)

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++

```
#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;
}
```

Course Schedule

Teoría

- Ordenamiento topológico

Solución

Tratamos de construir un ordenamiento topológico de los cursos. Si esto es posible, retornamos `true`. Si no, retornamos `false`.

Código

```cpp
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> listaAdyacencia(numCourses);
        vector<int> grado(numCourses, 0);
        for (vector<int> edge: prerequisites) {
            listaAdyacencia[edge[1]].push_back(edge[0]);
            grado[edge[0]]++;
        }
        queue<int> Q;
        vector<bool> escogido(numCourses, 0);

        for (int i = 0; i < numCourses; i++) {
            if (grado[i] == 0) {
                Q.push(i);
                escogido[i] = true;
            }
        }

        while(Q.size()) {
            int nodoActual = Q.front();
            Q.pop();
            for (int vecino: listaAdyacencia[nodoActual]) {
                grado[vecino]--;
                if (grado[vecino] == 0) {
                    escogido[vecino] = true;
                    Q.push(vecino);
                }
            }
        }

        return count(escogido.begin(), escogido.end(), true) == numCourses;
    }
};
```

Longest Increasing Path in a Matrix

Teoría

- DFS
- Programación dinámica

Solución

El camino más largo empezando en la celda con coordenadas (r, c) tiene longitud $1 + l$, donde el camino más largo empezando por una celda contigua a (r, c) y con un valor más alto tiene longitud l. Usando esta relación, podemos modificar DFS para que nos dé la respuesta que buscamos. Tenemos que guardar resultados intermedios en una DP para evitar TLE.

Código

```
class Solution {
public:
    vector<vector<int>> DP;
    int dfs(int row, int col, vector<vector<int>>& matrix) {
        if (DP[row][col] != -1)
            return DP[row][col];
        int best = 1;
        if (row > 0 && matrix[row-1][col] > matrix[row][col])
            best = max(best, 1 + dfs(row-1, col, matrix));
        if (col > 0 && matrix[row][col-1] > matrix[row][col])
            best = max(best, 1 + dfs(row, col-1, matrix));
        if (row + 1 < (int) matrix.size() && matrix[row+1][col] >
        ↪  matrix[row][col])
            best = max(best, 1 + dfs(row+1, col, matrix));
        if (col + 1 < (int) matrix[0].size() && matrix[row][col+1] >
        ↪  matrix[row][col])
            best = max(best, 1 + dfs(row, col+1, matrix));
        return DP[row][col] = best;
    }
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        DP.assign(matrix.size(), vector<int>(matrix[0].size(), -1));
        int best = 0;
        for (int i = 0; i < (int) matrix.size(); i++) {
            for (int j = 0; j < (int) matrix[0].size(); j++) {
                best = max(best, dfs(i, j, matrix));
            }
        }
        return best;
    }
};
```

Moocast (Gold)

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;
}
```

Course Schedule II

Teoría

- Ordenamiento topológico

Solución

Modificamos el código de *Course Schedule* para retornar el ordenamiento si existe.

Código

```cpp
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> listaAdyacencia(numCourses);
        vector<int> grado(numCourses, 0);
        for (vector<int> edge: prerequisites) {
            listaAdyacencia[edge[1]].push_back(edge[0]);
            grado[edge[0]]++;
        }
        queue<int> Q;
        vector<int> orden;

        for (int i = 0; i < numCourses; i++) {
            if (grado[i] == 0) {
                Q.push(i);
                orden.push_back(i);
            }
        }

        while(Q.size()) {
            int nodoActual = Q.front();
            Q.pop();
            for (int vecino: listaAdyacencia[nodoActual]) {
                grado[vecino]--;
                if (grado[vecino] == 0) {
                    orden.push_back(vecino);
                    Q.push(vecino);
                }
            }
        }

        if (orden.size() == numCourses)
            return orden;

        return {};
    }
};
```

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  }
```