Estructuras de Datos

Grafos

import java.util.\*;

public class GraphForCA {

class Node {

List<Edge> ady;

}

class Edge {

int ini;

int end;

int weight;

}

**// con matriz de adyacencia**

static int[][] matrizAdy;

**// para floyd-warshall**

static int[][] matrizAdyFW;

**// con lista adyacencia**

static Node[] listaAdy;

**// para dijkstra**

static int[] distancia;

static int[] predecesor;

static boolean[] visited;

**// para min tree**

static List<List<Integer>> sets = new ArrayList<List<Integer>>();

static int[][] matrizAdyMST;

static Node[] listaAdyMST;

private static void **initGraphMA**(int cnodes) {

matrizAdy = new int[cnodes][cnodes];

matrizAdyFW = new int[cnodes][cnodes];

}

private static void **initGraphLA**(int cnodes) {

listaAdy = new Node[cnodes];

distancia = new int[cnodes];

predecesor = new int[cnodes];

visited = new boolean[cnodes];

}

private static void **insertEdgeMA**(int ini, int end, int w) {

matrizAdy[ini][end] = w;

}

private static void **insertEdgeLA**(int ini, int end, int w) {

if (listaAdy[ini] == null) {

listaAdy[ini] = new Node();

listaAdy[ini].ady = new ArrayList<Edge>();

}

Edge e = new Edge();

e.ini = ini;

e.end = end;

e.weight = w;

listaAdy[ini].ady.add(e);

}

private static void **dfsMA**(int start) {

byte[] estado = new byte[matrizAdy.length];

// 1 preparado

// 2 espera

// 3 procesado

Arrays.fill(estado, (byte) 1);

Deque<Integer> pila = new ArrayDeque<Integer>();

pila.push(start);

estado[start] = 2;

while (!pila.isEmpty()) {

int toProcess = pila.pop();

estado[toProcess] = 3;

// for para poner los vecinos de toProcess en la pila

for (int i = 0; i < estado.length; i++) {

if (matrizAdy[toProcess][i] > 0) {

if (estado[i] == 1) {

pila.push(i);

estado[i] = 2;

}

}

}

// operaciones

}

}

private static void **bfsMA**(int start) {

byte[] estado = new byte[matrizAdy.length];

// 1 preparado

// 2 espera

// 3 procesado

Arrays.fill(estado, (byte) 1);

Deque<Integer> cola = new ArrayDeque<Integer>();

cola.push(start);

estado[start] = 2;

while (!cola.isEmpty()) {

int toProcess = cola.pop();

estado[toProcess] = 3;

// for para poner los vecinos de toProcess en la cola

for (int i = 0; i < estado.length; i++) {

if (matrizAdy[toProcess][i] > 0) {

if (estado[i] == 1) {

cola.addLast(i);

estado[i] = 2;

}

}

}

// aqui se realiza la operacion deseada

// con el nodo actualmente

// visitado

}

}

private static void **dfsLA**(int start) {

byte[] estado = new byte[listaAdy.length];

// 1 preparado

// 2 espera

// 3 procesado

Arrays.fill(estado, (byte) 1);

Deque<Integer> pila = new ArrayDeque<Integer>();

pila.push(start);

estado[start] = 2;

while (!pila.isEmpty()) {

int toProcess = pila.pop();

estado[toProcess] = 3;

for (int i = 0; i < listaAdy[toProcess].ady.size(); i++) {

if (estado[listaAdy[toProcess].ady.get(i).end] == 1) {

pila.push(listaAdy[toProcess].ady.get(i).end);

estado[listaAdy[toProcess].ady.get(i).end] = 2;

}

}

// operaciones

}

}

private static void **bfsLA**(int start) {

byte[] estado = new byte[listaAdy.length];

// 1 preparado

// 2 espera

// 3 procesado

Arrays.fill(estado, (byte) 1);

Deque<Integer> cola = new ArrayDeque<Integer>();

cola.push(start);

estado[start] = 2;

while (!cola.isEmpty()) {

int toProcess = cola.pop();

estado[toProcess] = 3;

for (int i = 0; i < listaAdy[toProcess].ady.size(); i++) {

if (estado[listaAdy[toProcess].ady.get(i).end] == 1) {

cola.addLast(listaAdy[toProcess].ady.get(i).end);

estado[listaAdy[toProcess].ady.get(i).end] = 2;

}

}

// operaciones

}

}

private static void **MAtoLA**() {

initGraphLA(matrizAdy.length);

for (int i = 0; i < matrizAdy.length; i++) {

for (int j = 0; j < matrizAdy.length; j++) {

if (matrizAdy[i][j] > 0) {

insertEdgeLA(i, j, matrizAdy[i][j]);

}

}

}

}

private void **warshall**() {

for (int i = 0; i < matrizAdy.length; i++) {

for (int j = 0; j < matrizAdy.length; j++) {

if (matrizAdy[i][j] == 0) {

matrizAdyFW[i][j] = 99999999;

} else {

matrizAdyFW[i][j] = matrizAdy[i][j];

}

}

}

for (int k = 0; k < matrizAdy.length; k++) {

for (int i = 0; i < matrizAdy.length; i++) {

for (int j = 0; j < matrizAdy.length; j++) {

matrizAdyFW[i][j] = Math.min(matrizAdyFW[i][j],

matrizAdyFW[i][k] + matrizAdyFW[k][j]);

}

}

}

}

private static void **djikstraMA**(int stat, long[] dist) {

visited = new boolean[matrizAdy.length];

int best;

for (int i = 0; i < dist.length; i++)

dist[i] = (int) 1e10;

dist[stat] = 0;

for (int i = 0; i < dist.length - 1; i++) {

best = -1;

for (int j = 0; j < dist.length; j++)

if (!visited[j] && (best == -1 || dist[j] < dist[best]))

best = j;

if (best == -1)

break;

visited[best] = true;

for (int j = 0; j < dist.length; j++)

if (matrizAdy[best][j] != -1 && !visited[j])

if (dist[best] + matrizAdy[best][j] < dist[j]){

dist[j] = dist[best] + matrizAdy[best][j];

predecesor[j] = best;

}

}

}

private static void **djikstraLA**(int stat, long[] dist) {

visited = new boolean[listaAdy.length];

int best;

for (int i = 0; i < dist.length; i++)

dist[i] = (int) 1e10;

dist[stat] = 0;

for (int i = 0; i < dist.length - 1; i++) {

best = -1;

for (int j = 0; j < dist.length; j++)

if (!visited[j] && (best == -1 || dist[j] < dist[best]))

best = j;

if (best == -1)

break;

visited[best] = true;

for (int j = 0; j < listaAdy[best].ady.size(); j++) {

int des = listaAdy[best].ady.get(j).end;

if (listaAdy[best].ady.get(j).weight != -1 && !visited[des])

if (dist[best] + listaAdy[best].ady.get(j).weight < dist[des])

dist[des] = dist[best] + listaAdy[best].ady.get(j).weight;

}

}

}

static private void **minTree**() {

cNodes();

tree = new Integer[3][nodes.length - 1];

Integer[] sort = adyList[2].clone();

Arrays.sort(sort);

List<List<Integer>> tempAdyList = new ArrayList<List<Integer>>();

tempAdyList.add(Arrays.asList(adyList[0]));

tempAdyList.add(Arrays.asList(adyList[1]));

tempAdyList.add(Arrays.asList(adyList[2]));

int edgesCount = 0;

int i = 0;

while (edgesCount < nodes.length - 1 || i < sort.length) {

int index = tempAdyList.get(2).indexOf(sort[i]);

int origen = tempAdyList.get(0).get(index);

int dest = tempAdyList.get(1).get(index);

int weigth = tempAdyList.get(2).get(index);

List<Integer> lo = tempAdyList.remove(0);

List<Integer> ld = tempAdyList.remove(0);

List<Integer> lw = tempAdyList.remove(0);

lo.remove(index);

ld.remove(index);

lw.remove(index);

tempAdyList.add(lo);

tempAdyList.add(ld);

tempAdyList.add(lw);

if (addToSets(origen, dest, weigth)) {

edgesCount++;

}

i++;

}

}

static private boolean **addToSets**(int origen, int dest, int weigth) {

int origenSet = -1;

for (int i = 0; i < sets.size(); i++) {

if (sets.get(i).contains(origen)) {

origenSet = i;

break;

}

}

int destSet = -1;

for (int i = 0; i < sets.size(); i++) {

if (sets.get(i).contains(dest)) {

destSet = i;

break;

}

}

if (origenSet == destSet && origenSet == -1) {

sets.add(0, new ArrayList<Integer>());

sets.get(0).add(origen);

sets.get(0).add(dest);

tree[0][insert2] = origen;

tree[1][insert2] = dest;

tree[2][insert2++] = weigth;

return true;

} else if (origenSet == -1) {

sets.get(destSet).add(origen);

tree[0][insert2] = origen;

tree[1][insert2] = dest;

tree[2][insert2++] = weigth;

return true;

} else if (destSet == -1) {

sets.get(destSet).add(dest);

tree[0][insert2] = origen;

tree[1][insert2] = dest;

tree[2][insert2++] = weigth;

return true;

} else if (origenSet == destSet && origenSet >= 0) {

return false;

} else {

while (sets.get(destSet).size() > 0) {

sets.get(origenSet).add(sets.get(destSet).remove(0));

}

tree[0][insert2] = origen;

tree[1][insert2] = dest;

tree[2][insert2++] = weigth;

return true; } } }

Para encontrar camino óptimo de I a J

**import** java.util.Arrays;

**import** java.util.PriorityQueue;

**import** java.util.StringTokenizer;

**class** Pair **implements** Comparable<Pair> {

**public** **int** first;

**public** **int** second;

**public** Pair(**int** first, **int** second) {

**this**.first = first;

**this**.second = second;

}

**public** **int** compareTo(Pair p) {

**if** (**this**.second != p.second)

**return** **this**.second < p.second ? -1 : 1;

**if** (**this**.first != p.first)

**return** **this**.first < p.first ? -1 : 1;

**return** 0;

}

}

**public** **class** Main\_lightOJ\_1019 {

**private** **static** **int**[][] *graph*;

**private** **static** **int** distance(**int** x, **int** y) {

PriorityQueue<Pair> pq = **new** PriorityQueue<Pair>();

**boolean**[] visited = **new** **boolean**[101];

pq.add(**new** Pair(x, 0));

**while** (!pq.isEmpty()) {

Pair par = pq.poll();

**if** (visited[par.first] == **false**) {

visited[par.first] = **true**;

**if** (par.first == y)

**return** par.second;

**for** (**int** i = 1; i < *graph*[par.first].length; i++) {

**if** (*graph*[par.first][i] > -1)

pq.add(**new** Pair(i, par.second + *graph*[par.first][i]));

}

}

}

**return** -1;

}

**public** **static** **void** main(String[] args) **throws** java.lang.Exception {

java.io.BufferedReader br = **new** java.io.BufferedReader(**new** java.io.InputStreamReader(System.***in***));

**int** cases = Integer.*parseInt*(br.readLine().trim());

**int** no = 0;

**while** (cases-- > 0) {

br.readLine();

StringTokenizer stk = **new** StringTokenizer(br.readLine());

**int** N = Integer.*parseInt*(stk.nextToken());

**int** M = Integer.*parseInt*(stk.nextToken());

*graph* = **new** **int**[N + 1][N + 1];

**for** (**int** i = 0; i < *graph*.length; i++) {

Arrays.*fill*(*graph*[i], -1);

}

**int** begin = 1, end = N;

**for** (**int** i = 0; i < M; i++) {

StringTokenizer st = **new** StringTokenizer(br.readLine());

**int** k, m, n;

k = Integer.*parseInt*(st.nextToken());

m = Integer.*parseInt*(st.nextToken());

n = Integer.*parseInt*(st.nextToken());

**if** (*graph*[k][m] == -1 || *graph*[k][m] > n) {

*graph*[k][m] = n;

*graph*[m][k] = n;

}

}

**int** d = *distance*(begin, end);

**if** (d == -1)

System.***out***.println("Case " + (++no) + ": Impossible");

**else**

System.***out***.println("Case " + (++no) + ": " + d);

}

}

}

Conjuntos Disjuntos

**public** **class** MisConjustosDisjuntos {

**private** **static** **int**[] *P*;

// private static int[] rank ;

**private** **static** **void** create(**int** x) {

*P*[x] = x;

// rank[x] = 0;

}

**private** **static** **int** find(**int** x) {

// if(x != P[x])

// P[x] = find(P[x]);

**return** *P*[x];

}

**private** **static** **void** merge(**int** x,**int** y){

**int** px = find(x);

**int** py = find(y);

**if**(px>py){

**for** (**int** i = 0; i < P.length; i++) {

**if**(P[i]==py){

P[i]=px;

}

}

}

**else**{

**for** (**int** i = 0; i < P.length; i++) {

**if**(P[i]==px){

P[i]=py;

}

}

}

}

**Articulation vertex**

#define MAXV 50005

#define MAXE 100005

struct node {

int v, next;

} L[MAXE];

int V, E, cnt, top, subtrees;int x, y;

int ptr[MAXV];int dfsnum[MAXV], low[MAXV];

bool mk[MAXV], ap[MAXV];

void DFS( int u ) {

mk[u] = true;

dfsnum[u] = low[u] = ++cnt;

for ( int i = ptr[u]; i >= 0; i = L[i].next ) {

int v = L[i].v;

if ( !mk[v] ) {

if ( !u ) subtrees++;

DFS( v );

if ( low[v] >= dfsnum[u] ) ap[u] = true;

low[u] <?= low[v];

} else low[u] <?= dfsnum[v]; }}

int main() {

memset( ptr, -1, sizeof( ptr ) );

scanf( "%d %d", &V, &E );

for ( int i = 0; i < E; i++ ) {

scanf( "%d %d", &x, &y );

x--; y--;

L[i] = ( node ) { y, ptr[x] };

ptr[x] = i;

L[ i + E ] = ( node ) { x, ptr[y] };

ptr[y] = i + E;

}

// Find Articulation Points

DFS( 0 );

if ( subtrees < 2 ) ap[0] = false;

for ( int i = 0; i < V; i++ )

if ( ap[i] ) printf( "%d\n", i + 1 );}

**Bridge**

#define MAXV 10000

#define MAXE 30000

#define BACK( x ) ( ( x < E ) ? x + E : x - E )

struct edge {

int v, next;

} edges[ 2 \* MAXE ];

int V, E;int u, v;int \_time;

int p[MAXV];int dfsn[MAXV], low[MAXV];

bool cut[ 2 \* MAXE ];vector< pair< int, int > > bridges;

void dfs( int u ) {

dfsn[u] = low[u] = ++\_time;

for ( int i = p[u]; i != -1; i = edges[i].next ) {

int v = edges[i].v;

if ( !dfsn[v] ) {

cut[ BACK( i ) ] = 1;

dfs( v );

low[u] <?= low[v];

if ( low[v] > dfsn[u] )

bridges.push\_back( make\_pair( u, v ) );

} else if ( !cut[i] ) low[u] <?= dfsn[v]; }}

int main() {

memset( p, -1, sizeof( p ) );

scanf( "%d %d", &V, &E );

for ( int i = 0; i < E; i++ ) {

scanf( "%d %d", &u, &v );

u--; v--;

edges[i] = ( edge ) { v, p[u] };

p[u] = i;

edges[ i + E ] = ( edge ) { u, p[v] };

p[v] = i + E; }

dfs( 0 );

int size = bridges.size();

printf( "%d\n", size );

for ( int i = 0; i < size; i++ )

printf( "(%d,%d)\n", bridges[i].first + 1, bridges[i].second + 1 );}

**Chequear si un grafo es bipartito**

#define MAXN 10001

vector<int> g[MAXN];

int part[MAXN];

bool possible, multiples;int p;

void check(int nod) {

if (!possible)

return;

if (part[nod] == -1) {

if (p > 1)

multiples = true;

part[nod] = p;

p += 2;}

for (int i = 0; i < (int) g[nod].size(); ++i) {

int v = g[nod][i];

if(v == nod)

continue;

if (part[v] == -1) {

part[v] = (part[nod] % 2) ? part[nod] - 1 : part[nod] + 1;

check(v);

} else if (part[v] == part[nod]) {

possible = false;

break;}}}

**Floyd warshall**

void Floyd(int V, int[][] c) {

int w;

for (int k = 0; k < V; k++)

for (int i = 0; i < V; i++)

if (c[i][k] < oo)

for (int j = 0; j < V; j++) {

if (c[k][j] < oo) {

w = c[i][k] + c[k][j];

if (w < c[i][j]) {

c[i][j] = w; }}}}

**MST Kruskal**

// Se hace el disjoint set primero

**struct** edge {

**int** u, v, w;

**bool** **operator<**(**const** edge& e) **const** {

**return** w < e.w;

}

};

vector<edge> e;

sort(e.begin(), e.end());

**int** cnt1 = 0, max\_v = 0, mst = 0;

**for** (size\_t i = 0; i < e.size() && cnt1 < n - 1; ++i) {

**int** u = e[i].u, v = e[i].v, w = e[i].w;

**int** set1 = find\_set(u);

**int** set2 = find\_set(v);

**if** (set1 != set2) {

merge(set1, set2);

cnt1++;

max\_v = max(max\_v, w); // lo que se quiere

mst += w;

}

}

**Segment Tree**

**int** array[n];

**int** tree[4 \* n], prop[4 \* n];

**void** **initialize**(**int** nod, **int** lo, **int** hi) {

//llegada a una hoja del tree

**if** (lo == hi) {

tree[nod] = array[lo];

**return**;

}

**int** mid = (hi + lo) / 2;

initialize(2 \* nod, lo, mid);

initialize(2 \* nod + 1, mid + 1, hi);

tree[nod] = tree[2 \* nod] + tree[2 \* nod + 1];

}

**void** **update**(**int** nod, **int** lo, **int** hi, **int** slo, **int** shi, **int** v) {

**if** (slo > hi || lo > shi)

**return**;

**int** tmpR = max(lo, slo), tmpL = min(hi, shi);

tree[nod] += v \* (tmpL - tmpR + 1);

**int** mid = (hi + lo) / 2, r = 2 \* nod, l = r + 1;

**if** (prop[nod]) {

prop[r] += prop[nod];

prop[l] += prop[nod];

tree[r] += prop[nod] \* (mid - lo + 1);

tree[l] += prop[nod] \* (hi - (mid + 1) + 1);

prop[nod] = 0;

}

**if** (slo <= lo && hi <= shi) {

prop[nod] += v;

**return**;

}

update(r, lo, mid, slo, shi, v);

update(l, mid + 1, hi, slo, shi, v);

}

**int** **query**(**int** nod, **int** lo, **int** hi, **int** slo, **int** shi) {

**if** (slo > hi || lo > shi)

**return** 0;

**int** mid = (hi + lo) / 2, r = 2 \* nod, l = r + 1;

**if** (prop[nod]) {

prop[r] += prop[nod];

prop[l] += prop[nod];

tree[r] += prop[nod] \* (mid - lo + 1);

tree[l] += prop[nod] \* (hi - (mid + 1) + 1);

prop[nod] = 0;

}

**if** (slo <= lo && hi <= shi) {

**return** tree[nod];

}

**int** rr = query(r, lo, mid, slo, shi);

**int** ll = query(l, mid + 1, hi, slo, shi);

**return** rr + ll;

}