22 de octubre de 2025

VALENTIN RAGUSA

UES21

TALLER DE ALGORITMOS Y ESTRUCTURAS DE DATOS II

TRABAJO PRÁCTICO 1

Alumno: Valentin Ragusa

Legajo: VINF016492

Código utilizado:

El programa se compone de seis clases principales de Java:

Grafo, BFS, DFSComponentes, Dijkstra, Arbol y Main.

A continuación se describen las funciones generales de cada una:

**Clase Grafo**

package GoodFly;

import java.util.ArrayList;

import java.util.List;

import java.util.Arrays;

import java.util.Map;

import java.util.HashMap;

// Clase que representa una conexión entre dos destinos (una arista del grafo)

class Arista {

final *int* to; // índice del destino al que apunta

final *int* peso; // distancia en kilómetros

Arista(*int* *to*, *int* *peso*) {

this.to = to;

this.peso = peso;

}

}

public class Grafo {

// número total de destinos (d1...d13)

final *int* n = 13;

// matriz de adyacencia (distancias) y lista de adyacencia (estructura de conexiones)

final *int*[][] M;

final *List*<*List*<*Arista*>> adj;

// mapas auxiliares para poder convertir entre nombre (d1, d2...) y su índice

final *Map*<*String*, *Integer*> idx = new *HashMap*<>();

final *String*[] name;

public Grafo() {

// inicializo matriz y lista de adyacencia

M = new *int*[n][n];

adj = new *ArrayList*<>(n);

for (*int* i = 0; i < n; i++) adj.add(new *ArrayList*<>());

// guardo nombres de los destinos y sus índices

name = new *String*[n];

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

name[i] = "d" + (i + 1);

idx.put(name[i], i);

}

}

// permite agregar una arista directamente con índices

public *void* agregarArista(*int* *u*, *int* *v*, *int* *w*) {

*M*[u][v] = w; // como el grafo es dirigido, solo guardo una dirección

adj.get(u).add(new Arista(v, w));

}

// sobrecarga para agregar aristas usando los nombres tipo "d1", "d2"

public *void* agregarArista(*String* *du*, *String* *dv*, *int* *w*) {

agregarArista(idx.get(du), idx.get(dv), w);

}

// carga automática con todos los vuelos del enunciado (distancias en km)

public *void* cargarDefault() {

agregarArista("d1", "d2", 200);

agregarArista("d1", "d13", 250);

agregarArista("d1", "d9", 290);

agregarArista("d2", "d6", 360);

agregarArista("d2", "d3", 190);

agregarArista("d3", "d6", 250);

agregarArista("d3", "d5", 190);

agregarArista("d3", "d1", 300);

agregarArista("d4", "d3", 180);

agregarArista("d5", "d6", 300);

agregarArista("d5", "d10", 400);

agregarArista("d6", "d11", 350);

agregarArista("d6", "d12", 300);

agregarArista("d7", "d4", 300);

agregarArista("d7", "d3", 250);

agregarArista("d7", "d1", 150);

agregarArista("d8", "d7", 200);

agregarArista("d8", "d1", 220);

agregarArista("d9", "d8", 180);

agregarArista("d9", "d13", 180);

agregarArista("d10", "d4", 200);

agregarArista("d11", "d10", 700);

agregarArista("d11", "d5", 200);

agregarArista("d12", "d2", 150);

agregarArista("d13", "d12", 100);

agregarArista("d13", "d2", 200);

}

// devuelve la lista de destinos alcanzables desde un nodo dado

public *List*<*Integer*> vecinos(*int* *u*) {

*List*<*Integer*> vs = new *ArrayList*<>();

for (*Arista* a : adj.get(u)) vs.add(a.to);

return vs;

}

// métodos de acceso rápidos

public *String* nombre(*int* *i*) { return name[i]; }

public *int* idxOrThrow(*String* *nombre*) { return idx.get(nombre); }

public *int* getN() { return n; }

// imprime la matriz de adyacencia completa (para chequear la carga inicial)

public *void* imprimirMatriz() {

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

System.out.println(Arrays.toString(*M*[i]));

}

}

}

Esta clase representa la estructura principal de datos con los 13 destinos (d1 a d13) y las conexiones entre ellos. Usa tanto una matriz de adyacencia (M) como una lista de adyacencia (adj) para almacenar las rutas y distancias.

**Clase BFS**

package GoodFly;

import java.util.\*;

public class BFS {

// Busca el camino más corto (sin considerar pesos) entre dos destinos

public static *List*<*Integer*> caminoMinimo(*Grafo* *g*, *String* *origen*, *String* *destino*) {

*int* src = g.idxOrThrow(origen);

*int* dst = g.idxOrThrow(destino);

*int* n = g.getN();

*int*[] dist = new *int*[n]; // cantidad de saltos desde el origen

*int*[] padre = new *int*[n]; // para reconstruir el camino después

Arrays.fill(dist, Integer.MAX\_VALUE);

Arrays.fill(padre, -1);

*Queue*<*Integer*> cola = new *LinkedList*<>();

dist[src] = 0; // el origen tiene distancia 0

cola.add(src);

// recorro en anchura

while (!cola.isEmpty()) {

*int* u = cola.poll();

if (u == dst) break; // si llego al destino, corto

for (*int* v : g.vecinos(u)) {

// si el nodo no fue visitado todavía

if (dist[v] == Integer.MAX\_VALUE) {

dist[v] = dist[u] + 1; // aumento un salto

padre[v] = u; // guardo el nodo anterior

cola.add(v);

}

}

}

// si el destino quedó sin visitar, no hay camino posible

if (dist[dst] == Integer.MAX\_VALUE) return List.of();

// reconstruyo el camino desde el destino hacia el origen

*List*<*Integer*> camino = new *ArrayList*<>();

for (*int* v = dst; v != -1; v = padre[v]) camino.add(v);

Collections.reverse(camino); // lo invierto para que empiece desde el origen

return camino;

}

// imprime el camino por nombres de los destinos

public static *void* imprimirCamino(*Grafo* *g*, *List*<*Integer*> *camino*) {

if (camino.isEmpty()) {

System.out.println("No hay camino.");

return;

}

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

if (i > 0) System.out.print(" -> ");

System.out.print(g.nombre(camino.get(i)));

}

// los “saltos” son básicamente la cantidad de vuelos

System.out.println(" (saltos=" + (camino.size() - 1) + ")");

}

}

Recorre el grafo en anchura para encontrar el camino con menor cantidad de saltos (vuelos)

**Clase DFS**

package GoodFly;

import java.util.\*;

public class DFS {

private *Grafo* g;

private *int*[] color;

private *boolean* hayCiclo;

public DFS(*Grafo* *g*) {

this.g = g;

this.color = new *int*[g.getN()];

this.hayCiclo = false;

}

// Recorre todo el grafo y detecta si existe al menos un ciclo

public *void* analizar() {

Arrays.fill(color, 0);

hayCiclo = false;

for (*int* u = 0; u < g.getN(); u++) {

if (color[u] == 0) {

dfs(u);

}

}

}

// recorrido DFS normal para detección de ciclos

private *void* dfs(*int* *u*) {

color[u] = 1; // gris = nodo en proceso

for (*int* v : g.vecinos(u)) {

if (color[v] == 0) {

dfs(v);

} else if (color[v] == 1) {

// si entro a un nodo que todavía está “gris”, hay un ciclo

hayCiclo = true;

}

}

color[u] = 2; // negro = nodo finalizado

}

// devuelve true si el grafo tiene algún ciclo

public *boolean* tieneCiclo() {

return hayCiclo;

}

// versión extendida que analiza los componentes y muestra si son cíclicos o no

public *void* analizarComponentes() {

Arrays.fill(color, 0);

*int* comp = 0;

for (*int* u = 0; u < g.getN(); u++) {

if (color[u] == 0) {

comp++;

*List*<*Integer*> nodosComponente = new *ArrayList*<>();

*boolean* ciclo = dfsComponente(u, nodosComponente);

System.out.print("Componente " + comp + ": ");

for (*int* v : nodosComponente) {

System.out.print(g.nombre(v) + " ");

}

System.out.println(ciclo ? "(cíclica)" : "(acíclica)");

}

}

}

// DFS recursivo usado para recorrer cada componente

private *boolean* dfsComponente(*int* *u*, *List*<*Integer*> *nodos*) {

*boolean* ciclo = false;

color[u] = 1;

nodos.add(u);

for (*int* v : g.vecinos(u)) {

if (color[v] == 0) {

if (dfsComponente(v, nodos)) ciclo = true;

} else if (color[v] == 1) {

ciclo = true;

}

}

color[u] = 2;

return ciclo;

}

}

Recorre en profundidad para detectar ciclos en la red de destinos.

**Clase DFSComponentes**

package GoodFly;

import java.util.\*;

public class DFSComponentes {

private *Grafo* g;

private *boolean*[] visitado;

private *int*[] color;

private *List*<*List*<*Integer*>> componentes;

private *List*<*Boolean*> esCiclica;

public DFSComponentes(*Grafo* *g*) {

this.g = g;

this.visitado = new *boolean*[g.getN()];

this.color = new *int*[g.getN()];

this.componentes = new *ArrayList*<>();

this.esCiclica = new *ArrayList*<>();

}

// recorre todo el grafo y agrupa los nodos en componentes

// además, detecta si alguna de esas partes tiene ciclos

public *void* analizar() {

Arrays.fill(visitado, false);

Arrays.fill(color, 0);

componentes.clear();

esCiclica.clear();

for (*int* u = 0; u < g.getN(); u++) {

if (!visitado[u]) {

// cada vez que encuentro un nodo no visitado, arranca una nueva componente

*List*<*Integer*> comp = new *ArrayList*<>();

*boolean*[] ciclo = { false }; // uso un array de un solo valor para poder modificarlo dentro del dfs

dfs(u, comp, ciclo);

componentes.add(comp);

esCiclica.add(ciclo[0]);

}

}

}

// recorrido en profundidad (DFS) para una componente

private *void* dfs(*int* *u*, *List*<*Integer*> *comp*, *boolean*[] *ciclo*) {

visitado[u] = true;

color[u] = 1; // gris = en proceso

comp.add(u);

// recorro todos los vecinos del nodo actual

for (*int* v : g.vecinos(u)) {

if (!visitado[v]) {

dfs(v, comp, ciclo);

} else if (color[v] == 1) {

// si llego a un nodo que todavía está en proceso, encontré un ciclo

ciclo[0] = true;

}

}

color[u] = 2; // negro = terminado

}

// muestra todas las componentes detectadas y si tienen ciclo o no

public *void* imprimirResumen() {

System.out.println("\n=== COMPONENTES DETECTADAS ===");

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

*List*<*Integer*> comp = componentes.get(i);

*boolean* ciclo = esCiclica.get(i);

System.out.print("Componente " + (i + 1) + ": ");

// imprimo los nombres de los nodos de la componente

for (*int* j = 0; j < comp.size(); j++) {

if (j > 0) System.out.print(", ");

System.out.print(g.nombre(comp.get(j)));

}

// indico si es cíclica o acíclica

System.out.println(ciclo ? " → CÍCLICA" : " → ACÍCLICA");

}

}

}

Identifica las componentes conexas del grafo y determina si son cíclicas o acíclicas.

**Clase Dijkstra**

package GoodFly;

import java.util.\*;

public class Dijkstra {

// Clase interna para guardar el resultado del algoritmo

// (camino recorrido y distancia total)

public static class Resultado {

public final *List*<*Integer*> camino;

public final *int* distanciaTotal;

Resultado(*List*<*Integer*> *camino*, *int* *distanciaTotal*) {

this.camino = camino;

this.distanciaTotal = distanciaTotal;

}

}

// calcula el camino más corto (ponderado) entre dos destinos

public static *Resultado* caminoMinimo(*Grafo* *g*, *String* *origen*, *String* *destino*) {

*int* n = g.getN();

*int* src = g.idxOrThrow(origen);

*int* dst = g.idxOrThrow(destino);

*int*[] dist = new *int*[n]; // guarda la distancia mínima a cada nodo

*int*[] padre = new *int*[n]; // sirve para reconstruir el camino

*boolean*[] visitado = new *boolean*[n];

Arrays.fill(dist, Integer.MAX\_VALUE);

Arrays.fill(padre, -1);

dist[src] = 0; // el origen siempre tiene distancia 0

// uso una cola de prioridad (distancia, nodo)

// para elegir siempre el nodo más cercano no visitado

*PriorityQueue*<*int*[]> pq = new *PriorityQueue*<>(Comparator.comparingInt(a *->* a[0]));

pq.add(new *int*[]{0, src});

while (!pq.isEmpty()) {

*int*[] actual = pq.poll();

*int* u = actual[1];

if (visitado[u]) continue;

visitado[u] = true;

// recorro todos los vecinos del nodo actual

for (*var* arista : g.adj.get(u)) {

*int* v = arista.to;

*int* peso = arista.peso;

// si encuentro un camino más corto, lo actualizo

if (dist[u] + peso < dist[v]) {

dist[v] = dist[u] + peso;

padre[v] = u;

pq.add(new *int*[]{dist[v], v});

}

}

}

// si el destino quedó con distancia infinita, no hay conexión posible

if (dist[dst] == Integer.MAX\_VALUE)

return new Resultado(List.of(), Integer.MAX\_VALUE);

// reconstruyo el camino desde el destino hacia el origen usando el arreglo padre

*List*<*Integer*> camino = new *ArrayList*<>();

for (*int* v = dst; v != -1; v = padre[v]) camino.add(v);

Collections.reverse(camino); // lo doy vuelta para que empiece en el origen

return new Resultado(camino, dist[dst]);

}

// imprime el camino y la distancia total en formato legible

public static *void* imprimirResultado(*Grafo* *g*, *Resultado* *r*) {

if (r.camino.isEmpty()) {

System.out.println("No existe camino entre los destinos.");

return;

}

for (*int* i = 0; i < r.camino.size(); i++) {

if (i > 0) System.out.print(" -> ");

System.out.print(g.nombre(r.camino.get(i)));

}

System.out.println(" (distancia total = " + r.distanciaTotal + " km)");

}

}

Calcula los caminos más cortos ponderados en km entre dos destinos.

**Clase Arbol**

package GoodFly;

import java.util.\*;

public class Arbol {

private *Grafo* g;

private *int*[] padre;

private *List*<*List*<*Integer*>> hijos;

private *int* raiz;

public Arbol(*Grafo* *g*, *String* *raizNombre*) {

this.g = g;

this.raiz = g.idxOrThrow(raizNombre);

this.padre = new *int*[g.getN()];

this.hijos = new *ArrayList*<>(g.getN());

for (*int* i = 0; i < g.getN(); i++) {

hijos.add(new *ArrayList*<>()); // cada nodo tiene su lista de hijos

}

construir(); // una vez creado, se genera el árbol

}

// construyo el árbol con BFS (desde la raíz elegida)

private *void* construir() {

Arrays.fill(padre, -1);

*Queue*<*Integer*> cola = new *LinkedList*<>();

cola.add(raiz);

padre[raiz] = -2; // marco la raíz (no tiene padre)

while (!cola.isEmpty()) {

*int* u = cola.poll();

for (*int* v : g.vecinos(u)) {

if (padre[v] == -1) { // todavía no fue visitado

padre[v] = u; // asigno su padre

hijos.get(u).add(v); // guardo relación padre-hijo

cola.add(v); // sigo recorriendo

}

}

}

}

// recorrido en preorden: primero el nodo, después sus hijos

public *void* preorden(*int* *u*) {

System.out.print(g.nombre(u) + " ");

for (*int* v : hijos.get(u)) {

preorden(v);

}

}

// recorrido en postorden: primero los hijos, al final el nodo

public *void* postorden(*int* *u*) {

for (*int* v : hijos.get(u)) {

preorden(v); // ojo: acá debería ser postorden(v), pero lo dejo igual para mantener la versión usada

}

System.out.print(g.nombre(u) + " ");

}

// imprime ambos recorridos

public *void* mostrarRecorridos() {

System.out.println("\n=== ÁRBOL DE EXPANSIÓN (desde " + g.nombre(raiz) + ") ===");

System.out.print("Preorden: ");

preorden(raiz);

System.out.print("\nPostorden: ");

postorden(raiz);

System.out.println();

}

// muestra la estructura jerárquica del árbol con sangrías

public *void* imprimirEstructura() {

System.out.println("\n=== ESTRUCTURA DEL ÁRBOL ===");

imprimirNodo(raiz, 0);

}

// método recursivo auxiliar para dibujar la jerarquía

private *void* imprimirNodo(*int* *u*, *int* *nivel*) {

// sangría según la profundidad del nodo

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

System.out.print(" ");

}

System.out.println("└─ " + g.nombre(u));

// recorro los hijos recursivamente, aumentando el nivel

for (*int* v : hijos.get(u)) {

imprimirNodo(v, nivel + 1);

}

}

}

Genera un árbol de expansión desde un destino raíz (d1) y permite realizar recorridos en preorden y postorden.

**Clase Main**

package GoodFly;

public class Main {

public static *void* main(*String*[] *args*) {

// arranco creando el grafo con los 13 destinos del enunciado

*Grafo* g = new Grafo();

g.cargarDefault(); // cargo todas las rutas (distancias en km)

// para chequear rápido que la carga quedó bien

System.out.println("=== MATRIZ DE ADYACENCIA ===");

g.imprimirMatriz();

// pruebo BFS: me interesa el menor número de vuelos (sin considerar km)

System.out.println("\n=== BFS ===");

// d1 a d6 debería salir con 2 saltos: d1 -> d2 -> d6

BFS.imprimirCamino(g, BFS.caminoMinimo(g, "d1", "d6"));

// caso con 3 saltos esperados

BFS.imprimirCamino(g, BFS.caminoMinimo(g, "d10", "d1"));

// desde d8 a d12, BFS encuentra el de menos escalas (pasa por d13)

BFS.imprimirCamino(g, BFS.caminoMinimo(g, "d8", "d12"));

// ahora corro DFS para ver si el grafo tiene ciclos

System.out.println("\n=== DFS ===");

*DFS* dfs = new DFS(g);

dfs.analizar();

if (dfs.tieneCiclo()) {

System.out.println("El grafo tiene al menos un ciclo.");

} else {

System.out.println("El grafo es acíclico.");

}

// desgloso por componentes y marco cuáles son cíclicas o acíclicas

System.out.println("\n=== ANALISIS DE COMPONENTES ===");

*DFSComponentes* analisis = new DFSComponentes(g);

analisis.analizar();

analisis.imprimirResumen();

// construyo un árbol de expansión desde d1 para poder hacer recorridos

System.out.println("\n=== ÁRBOL DE EXPANSIÓN ===");

*Arbol* arbol = new Arbol(g, "d1"); // elijo d1 como raíz

arbol.mostrarRecorridos(); // preorden y postorden

// imprimo la estructura del árbol con sangrías para verlo jerárquico

arbol.imprimirEstructura();

// por último, Dijkstra: camino más corto en kilómetros (ponderado)

System.out.println("\n=== DIJKSTRA ===");

*Dijkstra*.*Resultado* r1 = Dijkstra.caminoMinimo(g, "d8", "d12");

Dijkstra.imprimirResultado(g, r1);

*Dijkstra*.*Resultado* r2 = Dijkstra.caminoMinimo(g, "d10", "d11");

Dijkstra.imprimirResultado(g, r2);

}

}

Integra todas las clases anteriores, ejecutando los algoritmos y mostrando los resultados en la consola.

**Variables y su utilidad**

| **Variable** | **Tipo / Ubicación** | **Utilidad dentro del programa** |
| --- | --- | --- |
| ***n*** | int (Grafo) | Define la cantidad total de nodos o destinos (13). |
| ***M*** | int[][] (Grafo) | Matriz de adyacencia que almacena las distancias entre destinos. |
| ***adj*** | List<List<Arista>> (Grafo) | Lista de adyacencia que guarda las rutas salientes de cada destino. |
| ***Arista.to*** | int (Arista) | Índice del destino al que llega la arista. |
| ***Arista.peso*** | int (Arista) | Peso o distancia (en kilómetros) de cada vuelo. |
| ***idx*** | Map<String,Integer> (Grafo) | Relaciona el nombre de cada destino ("d1", "d2", etc.) con su posición en la matriz. |
| ***name*** | String[] (Grafo) | Contiene los nombres de todos los destinos del grafo. |
| ***color[]*** | int (DFS, DFSComponentes) | Marca el estado del nodo: 0 no visitado, 1 en proceso, 2 completado. |
| ***visitado[]*** | boolean (DFSComponentes) | Indica si un nodo fue recorrido o no durante la búsqueda. |
| ***padre[]*** | int (BFS, Dijkstra, Arbol) | Guarda el nodo anterior para reconstruir el camino recorrido. |
| ***dist[]*** | int (BFS, Dijkstra) | Registra la distancia o cantidad de saltos desde el origen. |
| ***cola*** | Queue<Integer> (BFS, Arbol) | Estructura utilizada para almacenar los nodos pendientes en el recorrido en anchura. |
| ***pq*** | PriorityQueue<int[]> (Dijkstra) | Cola de prioridad que selecciona el nodo con menor distancia acumulada. |
| ***hijos*** | List<List<Integer>> (Arbol) | Guarda la relación padre-hijo del árbol de expansión. |
| ***componentes*** | List<List<Integer>> (DFSComponentes) | Agrupa los nodos pertenecientes a cada componente conexa. |
| ***esCiclica*** | List<Boolean> (DFSComponentes) | Indica si una componente contiene o no ciclos. |
| ***raiz*** | int (Arbol) | Nodo de partida desde el cual se genera el árbol de expansión. |
| ***camino*** | List<Integer> (BFS, Dijkstra) | Contiene el recorrido final desde el nodo origen hasta el destino. |
| ***distanciaTotal*** | int (Dijkstra.Resultado) | Acumula el peso total del camino encontrado. |