UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

GRADO EN INGENIERÍA INFORMÁTICA

Técnicas de Programación Avanzada

Práctica 4 Parcial 2. Temas 5 y 6

Manuel Araújo Baño

Daniel Gutiérrez Torres

CURSO 2024-2025

Daniel Gutiérrez Torres - Manuel Araujo Baño

Índice

1 - Introducción **2**
2 - Objetivos **2**
3 - Fundamento teórico **3**
4 - Arquitectura del código **3**
5 - Desarrollo **4**
Ejercicio 1: 4
Ejercicio 2 5
Ejercicio 3 6
Ejercicio 4 7
Ejercicio 5 8
6 - Pruebas y resultados **9**
Ejercicio 1 9
Ejercicio 2 9
Ejercicio 3 10
Ejercicio 4 10
Ejercicio 5 11
7 - Conclusiones **11**

1 - Introducción

En esta práctica, se desarrollarán diferentes ejercicios centrados en el uso de estructuras de datos relacionales grafos y la aplicación de estrategias algorítmicas clásicas como divide y vencerás, avance rápido y vuelta atrás.

A través de la implementación de funciones específicas en la clase Practica4.java, se desarrollarán recorridos en grafos en anchura y profundidad, el cálculo del grado de un grafo, la identificación de las componentes conexas y la planificación óptima de rutas con restricciones de autonomía.

2 - Objetivos

Los objetivos de esta práctica son los siguientes:

- Implementación de funciones que trabajen con grafos dirigidos utilizando las clases Grafo.java, Lista.java y Par.java.
- Aplicar estrategias algorítmicas como:
    - Avance rápido
    - Vuelta atrás
    - Divide y vencerás
- Realizar recorridos en grafos:
    - Recorrido en anchura
    - Recorrido en profundidad
    - Calcular el grado de un grafo.
- Determinar el número de componentes conexos de un grafo.
- Diseñar una ruta óptima entre ciudades minimizando repostajes, considerando la autonomía del vehículo.

3 - Fundamento teórico

- **Grafo dirigido**: Estructura que representa relaciones asimétricas entre elementos. Cada arista tiene una dirección, que va desde un vértice origen hacia un vértice destino.

- **Recorrido en anchura**: Técnica que recorre los vértices por niveles, comenzando desde un nodo inicial. Utiliza una cola para procesar primero los nodos más cercanos, siendo útil para hallar distancias mínimas entre vértices.

- **Recorrido en profundidad**: Estrategia que explora lo más posible a lo largo de una rama antes de retroceder. Es útil para detectar componentes conexos, ciclos y se emplea en algoritmos de ordenación y detección.

- **Componentes conexos**: Conjunto de vértices que están interconectados entre sí. Identificarlas permite comprender la estructura global del grafo y resulta clave en análisis de redes o segmentación de rutas.

- **Grado de un vértice:** En grafos dirigidos, el grado de entrada indica cuántas aristas llegan a un vértice y el de salida cuántas salen. El grado total, se usará para evaluar la importancia o centralidad de un nodo.

- **Avance rápido**: Método que busca avanzar al máximo con los recursos disponibles, evitando retrocesos o paradas innecesarias. Se emplea en la planificación de rutas eficientes sin necesidad de explorar todas las combinaciones posibles.

- **TAD Lista como cola**: clase lista como una cola, accediendo sólo al primer y último elemento. Esto permite simular el comportamiento FIFO, necesario para el recorrido en anchura.

4 - Arquitectura del código

El proyecto se compone de una única clase principal llamada Practica4.java, que incluye las funciones requeridas (gradoGrafo, recorridoAnchura, componentesConexas, calcularViaje) y la función main para realizar pruebas.

Se utilizan las clases proporcionadas Grafo.java, Lista.java y Par.java, tal como indica el enunciado. Estas permiten trabajar con grafos dirigidos y estructuras auxiliares como listas.

Adicionalmente se han implementado funciones auxiliares para simplificar el código. Todo el desarrollo se ha realizado sin paquetes y siguiendo las restricciones impuestas por la práctica.

5 - Desarrollo

Ejercicio 1:

El objetivo de este ejercicio es implementar una función que calcula el grado máximo de un grafo dirigido. Para ello, se recorre el conjunto de vértices del grafo y se calcula para cada uno, la suma de su grado de entrada y grado de salida. La función mantiene un valor máximo que se actualiza si se encuentra un vértice con mayor grado.

La solución utiliza un bucle do-while, controlado por un contador llamado counter y una variable booleana seguir, que determina cuándo finalizar el recorrido. Esta estructura permite recorrer los vértices de forma segura y ordenada, sin usar estructuras de control prohibidas como break o return.

```
static public <Clave, InfoV, Coste> int gradoGrafo(Grafo<Clave,InfoV,Coste>
grafo) {
   int counter = 1;
   boolean seguir = true;
   int maxGrado = 0;

   Lista<Clave> listaVertices = grafo.listaVertices();

   do {
      if (counter <= listaVertices.longitud()) {

         Clave claveVertice = listaVertices.consultar(counter);

         int gradoVertice = grafo.gradoEntrada(claveVertice) +
         grafo.gradoSalida(claveVertice);

         if (gradoVertice > maxGrado) {
                maxGrado = gradoVertice;
         }

         counter++;
      } else {
         seguir = false;
      }
   } while (counter <= listaVertices.longitud() && seguir);
      return maxGrado;
}
```

Ejercicio 2

En este ejercicio se implementa una función que realiza un recorrido en anchura a partir de un vértice dado en un grafo dirigido. De esta forma se imprimen todos los vértices que son alcanzables desde el vértice de origen utilizando esta estrategia.

Para ello se simula el comportamiento de una cola mediante la clase Lista, usando exclusivamente inserciones al final y extracciones desde el principio. Se mantiene una lista de vértices ya visitados para evitar procesar los mismos nodos más de una vez y durante la ejecución, se imprimen los vértices conforme se visitan, comenzando desde el nodo inicial. Esta implementación garantiza que los vértices se recorren por niveles y que no se repiten vértices ya procesados.

Además, se incluye una comprobación inicial para asegurarse de que el vértice de partida existe en el grafo, lo que evita errores de ejecución.

```
public static <Clave, InfoV, Coste> void recorridoAnchura(
Grafo<Clave,InfoV,Coste> grafo, Clave v) {
   if (!grafo.existeVertice(v)) {
       System.out.println("El vértice " + v + " no existe en el grafo");
       return;
   }

   Lista<Clave> visitados = new Lista<Clave>();

   Lista<Clave> cola = new Lista<Clave>();

   cola.insertar(cola.longitud()+1, v);
   visitados.insertar(visitados.longitud()+1, v);

   System.out.println("Recorrido en anchura desde " + v + ":");
   System.out.println(v);

   while (!cola.esVacia()) {
       Clave actual = cola.consultar(1);
       cola.borrar(1);

        Lista<Clave> sucesores = grafo.listaSucesores(actual);

        for (int i = 1; i <= sucesores.longitud(); i++) {
            Clave sucesor = sucesores.consultar(i);

            if (visitados.buscar(sucesor) == 0) {

               cola.insertar(cola.longitud()+1, sucesor);
```

```
                visitados.insertar(visitados.longitud()+1, sucesor);

                System.out.println(sucesor);
            }
        }
    }
}
```

Ejercicio 3

En este ejercicio calculamos el número de componentes conexas presentes en un grafo. Para ello se parte de la premisa de que cada componente puede ser identificado a través de un recorrido en profundidad a partir de un vértice no visitado.

La función utiliza la lista de vértices del grafo como referencia para controlar cuáles aún no han sido explorados. Mientras existan vértices sin visitar se irá seleccionando el primero de ellos como punto de partida y se aplicará un recorrido en profundidad mediante la función auxiliar profREC que marca todos los vértices de su componente como visitados, eliminándolos de la lista. Cada vez que se ejecuta este recorrido, se incrementa en uno el contador de componentes encontradas.

Esta técnica garantiza que se identifican todas las zonas conectadas del grafo, separadas entre sí sin procesar ningún vértice más de una vez. También permite comprobar si el grafo está completamente conectado o dividido en varios subconjuntos aislados.

```
public static <Clave, InfoV, Coste> int componentesConexas(Grafo<Clave, InfoV,
Coste> grafo) {
   if (grafo.esVacio()) {
        return 0;
   }

   Lista<Clave> noVisitados = grafo.listaVertices();
   int numComponentes = 0;

   while (!noVisitados.esVacia()) {
        Clave vertice = noVisitados.consultar(1);
        System.out.println("TEST");
        Grafo.profREC(grafo, vertice, noVisitados);
        numComponentes++;
   }
   return numComponentes;
}
```

Ejercicio 4

En este ejercicio se implementa una función que planifica un viaje desde un punto de origen hasta un destino final, minimizando el número de paradas necesarias para repostar, teniendo en cuenta la autonomía máxima del coche. Para ello, se aplica una estrategia de avance rápido, que consiste en recorrer la mayor distancia posible con el combustible disponible antes de detenerse.

La función recibe tres parámetros: un array de nombres de ciudades, un array de distancias entre cada tramo y la autonomía máxima del camión. El recorrido comienza desde un punto A no incluido en el array de ciudades y se evalúa en cada tramo si es posible continuar sin repostar. En caso contrario se añade la ciudad anterior como parada y se reinicia la autonomía.

Las paradas necesarias se almacenan en un vector llamado resultado, como indica el enunciado. Al finalizar el recorrido, la función imprime las ciudades donde se realizaron paradas mediante un bucle for, y finalmente muestra cuántos kilómetros de autonomía quedaron disponibles tras llegar al destino.

Esta estrategia permite optimizar el viaje al reducir el número total de paradas, sin evaluar todas las combinaciones posibles,mejorando el rendimiento.

```
private static <Clave, InfoV, Coste> void profundidadComponente(Grafo<Clave,
InfoV, Coste> grafo,Clave inicio, Lista<Clave> noVisitados) {
    int pos = noVisitados.buscar(inicio);
    if (pos != 0) {
        noVisitados.borrar(pos);
    }

    Lista<Clave> sucesores = grafo.listaSucesores(inicio);
    for (int i = 1; i <= sucesores.longitud(); i++) {
        Clave sucesor = sucesores.consultar(i);
        if (noVisitados.buscar(sucesor) != 0) {
             profundidadComponente(grafo, sucesor, noVisitados);
        }
    }
}
```

Ejercicio 5

En este ejercicio se implementa la función calcularViaje, cuyo objetivo es planificar un trayecto desde un origen A hasta un destino B, pasando por una serie de ciudades intermedias, definiendo en qué ciudades detenerse para repostar, de forma que se realicen el menor número posible de paradas, respetando la autonomía máxima del camión.

La función toma como parámetros una lista ordenada de ciudades desde el primer punto intermedio hasta el destino final, unas distancias entre los tramos sucesivos, una autonomía total del camión en kilómetros y se utiliza una estrategia de avance rápido, evaluando en cada paso cuántos tramos consecutivos se pueden recorrer sin repostar. Si el siguiente tramo excede la autonomía restante se detendrá en la ciudad anterior y se reiniciará el contador de autonomía.

Las ciudades donde el camión debe detenerse se almacenan en una lista y al final del recorrido se imprimirá el nombre de cada parada, así como la autonomía restante al llegar al destino.

Esta solución evita decisiones innecesarias, haciendo avanzar al camión lo máximo posible en cada iteración, y logra una planificación eficiente del viaje con un mínimo de paradas.

```
public static void calcularViaje(String[] ciudades, int[] distancias, int
autonomia) {
   Lista<String> resultado = new Lista<String>();

   int autonomiaRestante = autonomia;
   int posicion = 0;
   int N = ciudades.length;

   while (posicion < N) {
      int alcance = posicion;
      int suma = 0;

      while (alcance < N && suma + distancias[alcance] <= autonomiaRestante) {
         suma += distancias[alcance];
         alcance++;
      }

      if (alcance == posicion) {
         autonomiaRestante = autonomia - distancias[posicion];

         if (posicion < N) {
             resultado.insertar(resultado.longitud() + 1, ciudades[posicion]);
         }
         posicion++;
```

```
      } else {
         autonomiaRestante -= suma;
         posicion = alcance;

         if (posicion < N) {
             resultado.insertar(resultado.longitud() + 1, ciudades[posicion-1]);
             autonomiaRestante = autonomia;
         }
      }
   }

   for (int i = 0; i <= resultado.longitud() - 1; i++) {
      System.out.println("Me detengo en: " + resultado.consultar(i + 1));
   }

   System.out.println("autonomía sobrante: " + autonomiaRestante);
}
```

6 - Pruebas y resultados

Ejercicio 1

Se crea un grafo con los vértices A, B, C y D, y se añaden aristas dirigidas: A→B, A→C, B→C, y C→D. La función gradoGrafo se llama sobre este grafo para determinar si se devuelve ell vértice con mayor grado.

```
int gradoMaximo = gradoGrafo(grafo);
System.out.println("Grado máximo del grafo: " + gradoMaximo);
```

- **Resultado esperado**: 3 (el vértice C tiene 2 entradas y 1 salida)
- **Resultado obtenido**: 3
- **Conclusión**: Éxito

Ejercicio 2

Usando el mismo grafo del ejercicio anterior, se llama a la función recorridoAnchura con el vértice A como punto de inicio. Esta función imprimirá los vértices alcanzables siguiendo una estrategia de recorrido en anchura, por lo que comprobaremos que lo hace de forma correcta.

```
recorridoAnchura(grafo, "A");
```

- **Resultado esperado**: A, B, C, D (orden de visita en recorrido en anchura)
- **Resultado obtenido**: A, B, C, D
- **Conclusión**: Éxito

Ejercicio 3

Se crea un nuevo grafo con 6 vértices numerados del 1 al 6 que se conectan entre sí formando dos componentes independientes 1, 2, 3 y 4, 5, 6. Las aristas serán bidireccionales y comprobaremos si devuelve el número correcto de componentes conexas), para asegurar el correcto cálculo de de las componentes conexas se añaden las aristas en ambas direcciones para simular que no es dirigido y poder calcular correctamente el número de componentes

```
int numComponentes = componentesConexas(grafo2);
System.out.println("Número de componentes conexas: " + numComponentes);
```

- **Resultado esperado**: 2 componentes conexas
- **Resultado obtenido**: 2
- **Conclusión**: Éxito

Ejercicio 4

Se define un array de ciudades que representan una ruta desde un punto A a B, y un array de distancias entre tramos consecutivos. Se llama a la función calcularViaje con una autonomía de 200 km. De esta forma comprobaremos que la función determinará de frma correcta en qué ciudades el camión debe detenerse a repostar y con que autonomía sobrante.

```
String[] ciudades = {"Toledo", "Ciudad Real", "Jaén", "Alcalá la Real",
"Granada"};
int[] distancias = {75, 120, 173, 70, 55};
int autonomia = 200;

calcularViaje(ciudades, distancias, autonomia);
```

- **Resultado esperado**: Toledo, Jaén, Granada, 145 km (parada1, parada2, parada3, autonomía sobrante)

- **Resultado obtenido**: Toledo, Jaén, Granada, 145 km
- **Conclusión**: Éxito

Ejercicio 5

```
String[] ciudades = {"Toledo", "Ciudad Real", "Jaén", "Alcalá la Real",
"Granada"};
int[] distancias = {75, 120, 173, 70, 55};
int autonomia = 200;

calcularViaje(ciudades, distancias, autonomia);
```

- **Resultado esperado**: Toledo, Jaén,llego a granada 55km (parada1, parada2, final, autonomía sobrante)
- **Resultado obtenido**: Toledo, Jaén, llego a granada, sobra 55km
- **Conclusión**: Éxito

7 - Conclusiones

La realización de esta práctica nos ha permitido aplicar de forma práctica conceptos de grafos y estrategias algorítmicas avanzadas, reforzando los contenidos vistos en los temas 5 y 6 de la asignatura. A través del desarrollo de las funciones de los diferentes ejercicios, se han desarrollados recorridos en anchura y puesto en práctica estrategias de avance rápido, se han calculado grados en grafos dirigidos, se han identificado componentes conexas y se ha planificado de forma óptima las rutas con restricciones de autonomía para un viaje.

En conjunto, la experiencia ha contribuido no solo al desarrollo de habilidades de programación estructurada y algorítmica, sino también a mejorar la capacidad para analizar problemas, diseñar soluciones eficientes y validar su funcionamiento mediante pruebas claras y controladas.