GRAFOS 2

¿CÓMO SON LAS ARISTAS?

Las aristas de la matriz pueden tener un valor o no, ya que hay grafos donde la existencia de la arista da información. Una de las diferencias entre los datos es si las aristas están ponderadas o no ponderadas. Las aristas no ponderadas las utilizaremos para resolver problemas desde el punto de vista mas abstracto. Hay otro tipo mas que son las anotadas donde la diferencia es que no es un valor único sino un conjunto de valores abstractos.

En un grafo, puedes encontrarte con que el grafo sea dirigido o no. Lo que indica es que las relaciones no son inepotentes es decir que la relación entre nodo a y nodo b no es la misma que nodo b a nodo a. Hay dos relaciones distintas y por tanto necesito saber hacia donde van las relaciones. Hay que tener en cuenta cual es el grado que tiene un brazo. El grado de un grafo son los grados de sus nodos o vértices y esto tiene 2 visiones, cuando es no dirigido y cuando si es dirigido. Cuando es no dirigido será el numero de aristas, cuando es dirigido tendrás un grado de entrada y otro de salida que representara que aristas llegan y cuales salen de ti.

Que haya ciclos nos dice que de alguna manera yo soy capaz de establecer un camino que me lleve al nodo de inicio y hay unos ciclos especiales que son los reflexivos que son propios y hay un tipo de grafo especial que son los DAG que son acíclicos, no hay manera de llegar de un nodo al mismo. Los acíclicos son muy importantes para muchas cosas por ejemplo un árbol ya que dice que es un grafo donde no hay caminos donde no puedo volver a atrás y representan relaciones de dependencia. Si quiero ver si existe relaciones de interdependencia es un grafo cíclico.

La representación de datos va a depender de las características que quiero usar en ese grafo. Un grafo no dirigido puedo hacerlo con un elemento que tiene su dato y tendrá las aristas que son relaciones autónicas, si es no dirigido, con tener un aspecto de nodo raíz me sirve para poder volver. Equivalente a los árboles. Nos limita a la hora de usar un grafo. Tengo un grao no dirigido acíclico, todos los nodos tienen la misma importancia, si la tienen entonces significa que desde un nodo D debo ser capad de ir al B y A, pero en el árbol no sucede y eso significa que captura la información de una forma determinada que va a serme más difícil definir ciertas operaciones.

Necesito poder implementar la estructura de un grafo que pueda tener ciclos, que sea dirigido, que sea ponderado y anotado. Modelamos la realidad de nuestro grafo mediante nodos.

El nodo tenga una clase de datos, además va a tener conexiones a otros nodos. Vamos a tener una lisa llamada salida y otra lista de aristas que llamamos entrada, estamos diciendo que un nodo tiene aristas que salen de el y por tanto estamos diciendo q son dirigidas y también dicen que hay aristas q me llegan que vienen de otros nodos.

```
Nodo{
        D dato;
        Lista<Aristas> salida;
        Lista<Aristas> entrada;
}
```

ARISTA<DA, D> {

DA anotación;

Nodo <D> origen;

Nodo <D> destino;

}

Puedo declarar un nodo A, B, C, nodos sin aristas es decir no conexo, a la hora de conectar generas una arista y dices que sale de A y llega a C y esta en la lista de A.salida y a su vez en c.entrada. El problema es que son objetos distintos pueden tener datos distintos, si 2 datos distintos representan la misma información debo asegurarme de que son iguales porque sino va haber incoherencias.

Para solucionarlo hay que tener un concepto, información igual a un dato y solo 1 para evitar las incoherencias, pero tiene sus problemas como de facilidad de uso, de rendimiento, etc.

Hay que meter a la parte de aristas :

Void setOrigenDestino(Nodo<D>, Nodo<D>, dest)

…..

O.salida.add(this)

Dest.entrada.add(this)

}

Esta opción nos permite navegar por el grafo. A partir del nodo me puedo mover y si es conexo es decir si existen caminos que conecten cualquier par de nodos significa que con tener uno puedo moverme por el grafo sin problemas. La cosa es cuando tengo un grafo y los nodos no están conectados.

La solución en este caso es la clase grafo.

GRAFO <D, DA>{

Lista<Nodo> Nodos;

Lista <Arista> Aristas;

}

Ahora si voy a tener siempre todos mis nodos y aristas y nos permite tener conjunto de nodos inconexos.

El problema es el mismo que al principio porque cada vez que tengo un nodo, hay que meterlo a la lista y cada vez que meto una lista hay que meterla también al grafo por lo cual necesito pasarle la referencia, el contexto al grafo.

Una vez que tenemos nodos, aristas y un grafo que mantiene toda la información ya tenemos cubierto todo. Ahor hay que ver como operamos aquí dentro, será parecido como los árboles.

Las funciones de utilidad irán dentro del grafo donde se puedan hacer búsquedas, poder recoger nodos, recoger aristas desde un determinado nodo, insertar nodo, insertar arista. A

nivel básico son las funcionalidad que vbamos a necesitar y en estas funciones vamos a buscar la gestión de las listas de ndoos y de aristas como las conexiones que tienen cada una de ellas. De momento todo lo que hemos hecho ha sido pensar desde el punto de vista matemático o abstracto, pero ahora toda desde el punto de vista informático y ahora tenemos un posible problema ya que yo lo que no tengo son los nodos identificados de alguna manera. ¿Puedo meter un nodo duplicado en mi grafo? No debería ya que esa es la gracia de los grafos. Lo mismo pasa con las aristas. ¿Cómo las identifico?

Todo esto me llega a pensar que desde el punto de vista práctico me falta algún tipo de identificador como por ejemplo un entero o estableces como hacíamos con los últimos nodos de los árboles donde poníamos clave, valor.

Existe un acrítica a lo que hemos hecho que es, si tienes un nodo, esa instancia tiene un identificador en memoria, ¿porque pones otro? Porque un grafo no está siempre en memoria, cuando se guarda a disco, al volver a levantarlo las referencias ya no son las mismas.

¿CUALES SON LAS REPRESENTACIONES A NIVEL DE DATOS QUE PODEMOS TENER?

La representación matricial se usa normalmente en problemas matemáticos y que en implementación se ha utilizado, pero en la actualidad no tanto porque estas matrices requieres espacio. Para meterla en un ordenador necesita reservar espacio continuo de memoria para toda la matriz y es poco práctico además de que los grafos son dinámicos y por tanto la matriz debería ser capaz de crecer, pero las matrices no crecen, cuando las hacemos crecer necesitan coger más memoria y necesitamos que esa memoria sea continua. Por lo tanto, el enfoque de usar matrices falla un poco.

Dependiendo de las características que tengamos dentro de los grafos y lo que se permite hacer

GRAFOS 3

DEFINICIÓN:

Estructura de datos compuesta por nodos(vértices) y aristas (conexiones)

TIPOS:

- Dirigidos: todas las aristas con dirección dirigida
- No dirigidos: conexiones bidireccionales entre nodos
- Ponderado: las aristas tienen peso o coste asociados
- No ponderados: todas las conexiones con mismo peso(distancia)

DIRIGIDOS NO DIRIGIDOS PONDERADO NO PONDERADO

REPRESENTACIÓN MATRICIAL:

La matriz adyecente es un array bidimensional donde matriz [i][j] conexión entre nodo i,j.

EJEMPLO:

```
Class MatrizAdyecencia{
        Private int [][]matriz;
        Private int numVertices;
        Public MatrizAdyacencia(int numVertices){
                This.numVertices=numVertices;
                Matriz=new int[numVertices] [numVertices] ;
        }
        Public void agregarAristas (int ori, int dest){
                Matriz[ori][dest] =1 ;
                Matriz[dest][ori] =1 ;
        }
        Public void mostrarMatriz(){
                For( int[] fila :mat5riz)
                        System.out.println(Array.toString(fila)) ;
        }
```

LISTA ADYECENCIA :

Cada nodo tiene una lista de vecinos (Hash Map)

EJEMPLO:

```
Class Grafo {
   protected HashMap<String, List<String>> adyacencia;
   public Grafo() {
      adyacencia = new HashMap<>();
   }
   public void addVertice(String vertice) {
      adyacencia.putIfAbsent(vertice, new ArrayList<>());
   }
   public void addArista(String origen, String destino) {
      adyacencia.putIfAbsent(origen, new ArrayList<>());
      adyacencia.putIfAbsent(destino, new ArrayList<>());
```

```
        adyacencia.get(origen).add(destino);
    }
}
```

BUCLES:

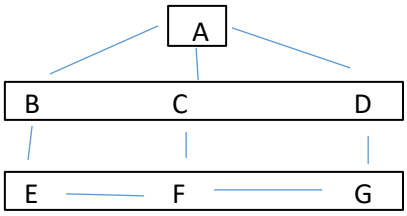
Se usan para construir grafos o crear caminos (aristas) entre nodos.

```
public class grafos {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, List<String>> grafo = new HashMap<>();

        while ( !(linea = sc.nextLine()).equals("fin")) {
            String[] partes = line.split("  ");
            If (partes.length != 2) continue ;
            String origen = partes[0],
                    destino = partes[1];

            grafo.computeIfAbsent(origen, k -> new ArrayList<>()).add(destino);
            grafo.computeIfAbsent(destino, k -> new ArrayList<>()).add(orgen) ;
        }
    }
}
```

ALGORITMOS EN GRAFOS:

- DFS(Depth First Search): recorrido en profundidad
- BFS(Breath First Search): recorrido en anchura
- Dijkstra: recorrido más corto en graos ponderados
- Krustal y Prim: conectar todos los nodos con el menor pero total sin ciclos.

EJEMPLO:



- DFS: A → B → E → C → D → G

- BFS: A → B → C → D→ E → F → G

- Dijkstra: A → D (3), D → G (1) Total: 4

    - A asigna distancia 0

    - Revisa vecinos y actualiza distancias

    - Visita nodo no visitado y menor distancia y repite el proceso

- Kruskal y Prim:

  - Kruskal: Ordena aristas por peso y las añade si no forman ciclos.

  - Prim: Comienza en un nodo y va añadiendo su vecino más barato respecto peso de aristas.

TPOS ESPECIALES:

• Gráfico conexo: Hay camino entre todas las parejas de nodos.

• Gráfico cíclico: Tiene ciclos.

• Gráfico acíclico: No tiene ciclos.

• Gráfico completo: Todos los nodos conectados entre sí.

• Árbol: Gráfico acíclico y conexo (n-1 aristas para n vértices).

• Gráfico dirigido acíclico (DAG): Aristas sin dirección y sin ciclos.

ALMACENAMIENTO EN DATA:

Usamos JDBC (Java Database Connectivity) para insertar datos en bases de datos.

EJEMPLO :

```
Connection conn = DriverManager.getConnection(url, user, password);

PreparedStatement stmt = conn.prepareStatement("INSERT INTO");

stmt.setString(1, "Laptop");

stmt.setInt(2, 2);
```

stmt.setDouble(3, 1200.00);

stmt.executeUpdate();

LEER REGISTROS DE UN ARCHIVO DE LOGS:

contienen información de eventos dentro de un sistema.

- Datos semiestructurados: **archivos JSON**
- Datos no estructurados: **imágenes y videos**

EJEMPLO:

```
BufferedReader br = new BufferedReader(new FileReader("log.txt"));

while ((line = br.readLine()) != null) {

   if(line.contains("ERROR")) System.out.println("Error " + line);

}
```

ANÁLISIS DATOS CON JAVA STREAM:

análisis información eficientemente y detallada.

EJEMPLO :

```
List<Product> products = Arrays.asList(new Product("Laptop", 10, 1200),new Product("Tablet", 15, 500));

double totalSales = products.stream().mapToDouble(p -> p.getCantidad() * p.getPrecio()).sum();

System.out.println(Total ventas  + totalSales);
```

ANÁLISIS DATOS CON OLAP (ONLINE ANALYTICAL PROCESSING):

analizar grandes cantidades de datos de manera rápida y eficiente.

EJEMPLO :

```
Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("Select product");

while (rs.next()) System.out.println("Product");
```

ALGPORITMO DE DIJKSTRA

un grafo con un nodo la matriz es 0, con dos nodos es medianamente fácil, con tres nodos ya he tenido que seguir la secuencia.

¿Con 100 nodos? = 100 x 100= 10000

Ejemplo:

Pueblos de Guadalajara 498

Pueblos de España 18000

Cuantes calles? ….

Por lo tanto, las matrices de adyacencias son un problema. Cuando las matrices crecen, quedan muchas cosas vacías y son muy largas.

Uno de los problemas es si quiero ir desde el pueblo A hasta el pueblo C. En la matriz, aparece que no hay nada porque la matriz representa la adyacencia (vecinos directos) que tiene los nodos.

Esto hay que transfórmalo a otra “cosa” es decir a otra matriz, pero esta vez matriz de distancias. Dijkstra es el proceso por el cual paso de una matriz de adjaciencia a la matriz de distancia. Hay otros, además. Esta transformación puede venir dada por la matriz de adjacencia o por el propio grafo. Dijkstra no hace la matriz de distancias completa (algoritmo de floid), lo que hace es el cálculo de distancias desde un nodo elegido como por ejemplo A hasta el resto.

Como las matrices crecen, cuesta mas guardarlas porque van aumentando en tamaño. Vamos a hacer un método determinista que es que dada una entrada siempre va a salir una salida, pero dicha salida siempre va a buscar la mas optima es decir la mejor de las salidas. Cuando las matrices son tan grandes encontrar el optimo es un problema porque lleva muchísimo tiempo.

El proceso para encontrar el óptimo no es cuadrático sino exponencial. Hoy en día por encima de 100 posibles destinos no se intenta resolver el sistema, sino que se usan otros métodos, heurísticos y metaheurísticos. Los heurísticos te dan la solución que es suficiente mente buena.

Los algoritmos nol estan pensados para que te den la ruta sino la distancia mas corta.

Primero usas distea para encontrar la primera fila de la matriz de distancia. Para calcular la matriz completa se necesita meter un bucle.

Hemos transformado el grafo en un árbol almacenando hacia atrás los caminos d los nodos.

Rutas= 1 por nodo que visito.