ESCUELA POLITECNICA NACIONAL

FACULTAD DE INGENIERÍA DE SISTEMAS

ICCD442-ESTRUCTURA DE DATOS Y ALGORITMOS II-GR3SW

GRUPO: No. 3

Deber No: 1

Alumnos:

Huilca Villagómez Fernando Eliceo

Quisilema Flores Juan Mateo

Simbaña Guarnizo Mateo Nicolás

PROFESORA: Dra. María Pérez

FECHA DE ENTREGA: 18-10-2024

Laboratorio de:

Estructura de Datos y Algoritmos II

Práctica No.:

1

Tema:

Representación de grafos y algoritmos de búsqueda en grafos

Objetivos:

1. Comprender los conceptos básicos de grafos y sus componentes.
2. Implementar las diferentes representaciones de grafos.
3. Dominar los algoritmos de recorrido en profundidad (DFS) y en anchura (BFS) y aplicarlos a problemas prácticos.

Marco teórico:

Definición

Un grafo es una estructura de datos matemática que consta de un conjunto de vértices (también llamados nodos) y un conjunto de aristas (también llamadas bordes) que conectan estos vértices. [1]

Componentes

- **Nodos (vértices):** Son elementos individuales en el grafo. Cada vértice puede representar entidades como personas, lugares, o cualquier objeto que se esté modelando. [1]
- **Relaciones (aristas):** Son las conexiones entre los vértices. Estas aristas pueden ser direccionales o no direccionales, dependiendo del tipo de relación que se quiera representar. [1]

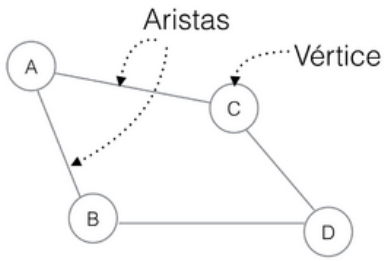
Tipos de grafos

- **Grafos dirigidos:** Cada arista tiene una dirección asociada. Esto significa que la relación entre dos vértices es unidireccional, y se representa mediante una flecha que indica la dirección de la conexión. La arista va desde un vértice inicial (fuente) hacia un vértice final (destino). [2]
- **Grafos no dirigidos:** Las aristas no tienen dirección. Esto significa que la relación entre dos vértices es bidireccional y simétrica. La conexión entre dos nodos no tiene una orientación específica, y se representa mediante una línea que conecta los vértices. [2]
- **Grafos ponderados:** Se dice que estos grafos están etiquetados cuando sus aristas contienen datos. En particular se dice que un grafo tiene peso si cada arista tiene un valor numérico no negativo que le proporciona condiciones de peso o longitud. [3]

Grafo simple

Un grafo simple se define como una estructura en la que un conjunto no vacío de vértices está conectado mediante aristas, donde cada arista une dos vértices distintos y no hay múltiples aristas entre los mismos pares de vértices ni bucles. [3]

En la siguiente imagen se muestra un grafo simple, compuesto por nodos, que contienen un valor alfabético y se representan mediante círculos. Además, tiene aristas que conectan los vértices sin una dirección específica, por lo que se clasifica como un grafo no dirigido.

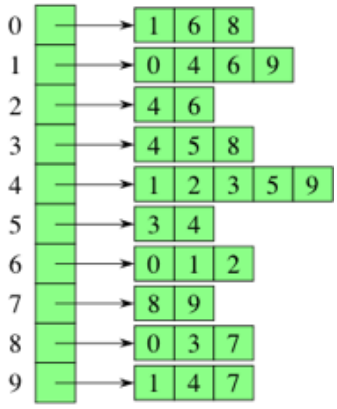


Representaciones de grafos

- **Lista de adyacencia**

Representar un grafo con listas de adyacencia combina las matrices de adyacencia con las listas de aristas. Para cada vértice i, almacena un arreglo de los vértices adyacentes a él. Típicamente tenemos un arreglo de |V| listas de adyacencia, una lista de adyacencia por vértice. [4]

Los números del vértice en una lista de adyacencia no están obligados a aparecer en ningún orden en particular, aunque a menudo es conveniente enumerarlos en orden ascendente. [4]



- **Matriz de adyacencia:**

En un grafo con |V| vértices, la matriz de adyacencia es una matriz de tamaño |V| x |V| que contiene ceros y unos, donde el vértice origen se encuentra en la fila y el vértice final se encuentra en la columna. La entrada en la fila i y la columna j es 1 si y solo si existe una arista entre los vértices i y j en el grafo. [4]

En el caso de un grafo no dirigido, la matriz de adyacencia es simétrica: la entrada en la fila i, columna j es 1 si y solo si la entrada en la fila j, columna i también es 1. En un grafo dirigido, la matriz no necesita ser simétrica. [4]

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Importancia de los recorridos en los grafos

Es importante porque permite comprender las relaciones entre los vértices y las interacciones dentro del grafo. Los recorridos, como el de la ruta más corta, son útiles para realizar cálculos importantes.

En redes sociales, los grafos ayudan a analizar las interacciones entre usuarios. Cada usuario se representa como un nodo, y sus conexiones (amigos o seguidores) como las aristas. Al recorrer estos nodos, se puede simular cómo se propaga la información, cómo se forman comunidades o cómo se difunden tendencias.

Por ejemplo, cuando un nodo se activa (un usuario recibe una noticia), se puede observar cómo esa información se distribuye a través de la red.

Recorrido en profundidad (DFS)

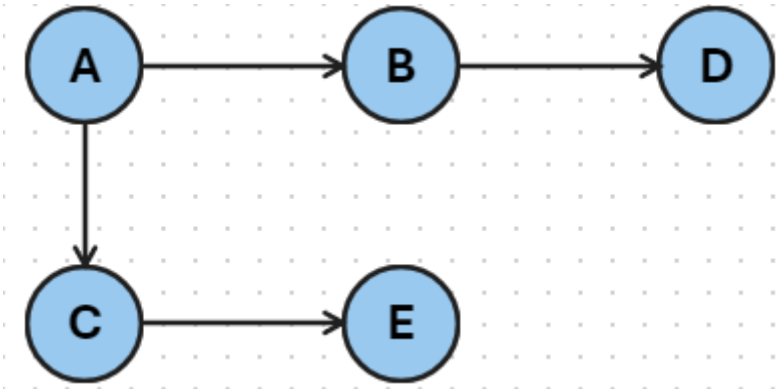
El recorrido explora el grafo siguiendo un camino hasta llegar al final, y retrocede cuando no puede avanzar más. Utiliza una pila como estructura de datos para almacenar los nodos pendientes por explorar, lo que permite regresar y continuar la exploración desde puntos no visitados. [5]

Se usa en una amplia variedad de aplicaciones, desde juegos hasta la resolución de problemas complejos en inteligencia artificial. [5]

El procedimiento es el siguiente:

1. Se inicializa una pila vacía y se empuja el nodo inicial.
2. Mientras la pila no esté vacía, se extrae el nodo de la cima y se verifica si ha sido visitado.
3. Si no ha sido visitado, se marca como visitado y se empujan sus vecinos no visitados a la pila.

Ejemplo:



Si se comienza el DFS en el nodo A, el algoritmo seguirá este recorrido:

1. Se inicia en el nodo A y se marca como visitado. A se apila en la estructura de la pila.
2. Se mueve al siguiente nodo adyacente no visitado (en este caso, B), se marca como visitado y se apila.
3. Desde B, se mueve al nodo adyacente no visitado (D), lo marca como visitado y lo apila.
4. Desde D, como no hay más nodos adyacentes no visitados, el algoritmo desapila D y retrocede al nodo B.
5. Desde B, no hay más nodos por explorar, así que el algoritmo desapila B y regresa a A.
6. Desde A, se mueve al siguiente nodo adyacente no visitado, que es C, lo marca como visitado y lo apila.
7. Desde C, se mueve al nodo adyacente no visitado E, lo marca como visitado y lo apila.
8. Como no hay más nodos adyacentes a E, el algoritmo desapila E, luego desapila C, y finalmente desapila A, completando el recorrido.

De modo que, el orden de visita de los nodos sería: A → B → D → C → E

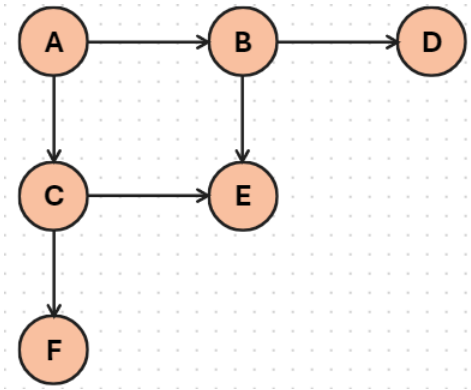
Recorrido en anchura (BFS)

Explora todos los nodos de un grafo desde un nodo inicial en orden de niveles, es decir, explora primero todos los vecinos directos antes de pasar a los vecinos de los vecinos. Esto lo convierte en un algoritmo no recursivo que se basa en una estructura de datos clave: la cola. [5]

A continuación, se describe el proceso:

1. Se comienza con un nodo inicial que se agrega a la cola.
2. Se marca el nodo como visitado para evitar volver a explorarlo.
3. Se extrae el primer nodo de la cola y se exploran todos sus vecinos.
4. Los vecinos no visitados se agregan a la cola para ser procesados más tarde.
5. El proceso continúa hasta que la cola esté vacía, lo que indica que se han explorado todos los nodos accesibles.

Ejemplo:



Si se aplica el BFS empezando en el nodo A, el recorrido sería el siguiente:

1. Se inicia en A y se agregan los vecinos B y C a la cola.
2. Se extrae B de la cola y se agregan sus vecinos D y E a la cola.
3. Se extrae C de la cola y se agrega su vecino F a la cola.
4. El siguiente en la cola es D, pero no tiene vecinos, por lo que se extrae sin agregar más nodos.
5. Se extraen E y F, completando el recorrido.

De manera que, el orden de visita de los nodos sería: A → B → C → D → E → F

Diferencias entre DFS y BFS

Aspecto	BFS	DFS
Estructura	Cola (FIFO)	Pila (LIFO)
Exploración	Nivel por nivel	Profundidad hasta el final
Utilidad	Encontrar caminos más cortos	Detectar ciclos y explorar todas las rutas

System.nanoTime ()

Devuelve el valor actual del temporizador de alta precisión en nanosegundos. Este método mide el tiempo transcurrido de manera muy precisa y es adecuado para medir intervalos cortos de tiempo, como la duración de la ejecución de un bloque de código. [6]

Ofrece precisión en nanosegundos, pero la exactitud depende del temporizador del sistema subyacente. No está vinculado a la hora del sistema o reloj, por lo que el resultado reflejado no tiene ningún punto de referencia fijo, según la documentación de Java. [6]

Ideal para comparar tiempos y medir la duración entre dos eventos en una aplicación. [6]

System.currentTimeMillis ()

Devuelve la cantidad de milisegundos transcurridos desde la medianoche del 1 de enero de 1970 UTC (también conocido como "epoch"). Este método está ligado al reloj del sistema y se usa para obtener la hora actual. [6]

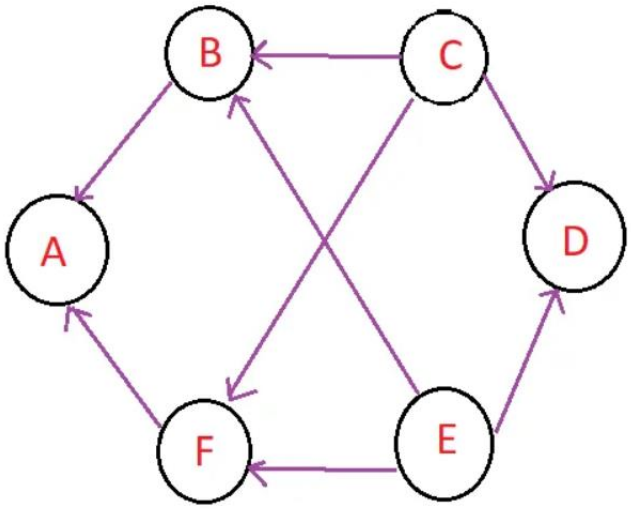
Proporciona una resolución de milisegundos, sin embargo, puede dar resultados incorrectos si el usuario cambia la hora del sistema, golpea un segundo salto o hay cambios en la sincronización NTP. [6]

Se utiliza cuando se necesita registrar la hora actual o hacer cálculos basados en la hora del sistema. [6]

Desarrollo de la práctica:

1. **Ejercicio 1: Construcción de un grafo**

Grafo Propuesto:



Lista de Adyacencia:

A	→ null			
B	→ A	→ null		
C	→ B	→ D	→ F	→ null
D	→ null			
E	→ B	→ D	→ F	→ null
F	→ A	→ null		

Matriz de Adyacencia:

	A	B	C	D	E	F	Grado de salida
A	0	0	0	0	0	0	0
B	1	0	0	0	0	0	1
C	0	1	0	1	0	1	3
D	0	0	0	0	0	0	0
E	0	1	0	1	0	1	3
F	1	0	0	0	0	0	1
Grado de llegada	2	2	0	2	0	2	

Implementación en código

Creamos los nodos y asignamos la información dada:

```
NodoGrafo<String> nodo11 = new NodoGrafo<String>( infoNodo: "A");
NodoGrafo<String> nodo22 = new NodoGrafo<~>( infoNodo: "B");
NodoGrafo<String> nodo33 = new NodoGrafo<~>( infoNodo: "C");
NodoGrafo<String> nodo44 = new NodoGrafo<~>( infoNodo: "D");
NodoGrafo<String> nodo55 = new NodoGrafo<~>( infoNodo: "E");
NodoGrafo<String> nodo66 = new NodoGrafo<~>( infoNodo: "F");
```

Agregamos los nodos creados:

```
//Agregamos los nodos
grafoDirigido1.agregarNodo(nodo11);
grafoDirigido1.agregarNodo(nodo22);
grafoDirigido1.agregarNodo(nodo33);
grafoDirigido1.agregarNodo(nodo44);
grafoDirigido1.agregarNodo(nodo55);
grafoDirigido1.agregarNodo(nodo66);
```

Definimos las aristas como se planteó en el ejercicio:

```
//Dirigimos los nodos
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 1,  nodoDestino: 0);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 2,  nodoDestino: 1);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 2,  nodoDestino: 3);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 2,  nodoDestino: 5);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 4,  nodoDestino: 3);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 4,  nodoDestino: 1);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 4,  nodoDestino: 5);
grafoDirigido1.dirigirAristaANodo( nodoOrigen: 5,  nodoDestino: 0);
```

Lista que nos arroja nuestro programa en consola:

```
_____________________ LISTA DE ADYACENCIA GRAFO DIRIGIDO N# 0 ____________
A → null
B → A → null
C → B → D → F → null
D → null
E → D → B → F → null
F → A → null
```

Matriz que nos arroja nuestro programa:

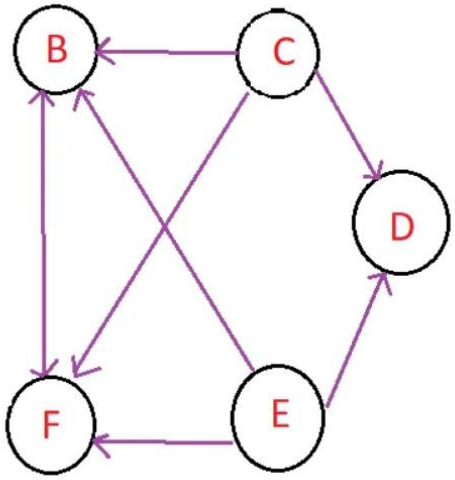
```
_____________ MATRIZ DE ADYACENCIA GRAFO DIRIGIDO N# 0 ___________
 false  false  false  false  false  false
 true  false  false  false  false  false
 false  true  false  true  false  true
 false  false  false  false  false  false
 false  true  false  true  false  true
 true  false  false  false  false  false
```

Recorrido en profundidad:

```
CASO 6 Imprimir recorrido en profundidad grafo Dirigido _________________
 A

 B

 C
 D C
 C
 F C
 C

 E

La salida del recorrido es la siguiente: [A, B, D, F, C, E]
```

2. Ejercicio 2: Modificación de grafos

El grafo del ejercicio 1 se modificó de la siguiente manera:



Lista de adyacencia:

B	→ F	→ null		
C	→ B	→ D	→ F	→ null
D	→ null			
E	→ B	→ D	→ F	→ null
F	→ B	→ null		

Matriz de adyacencia:

	B	C	D	E	F	Grado de salida
B	0	0	0	0	1	1
C	1	0	1	0	1	3
D	0	0	0	0	0	0
E	1	0	1	0	1	3
F	1	0	0	0	0	1
Grado de llegada	3	0	2	0	0	

Implementación en código

Creamos los nodos:

```
NodoGrafo<String> nodo22 = new NodoGrafo<~>( infoNodo: "B");
NodoGrafo<String> nodo33 = new NodoGrafo<~>( infoNodo: "C");
NodoGrafo<String> nodo44 = new NodoGrafo<~>( infoNodo: "D");
NodoGrafo<String> nodo55 = new NodoGrafo<~>( infoNodo: "E");
NodoGrafo<String> nodo66 = new NodoGrafo<~>( infoNodo: "F");
```

Apuntamos los nodos:

```
//Dirigimos los nodos
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 0,  nodoDestino: 2);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 2,  nodoDestino: 3);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 2,  nodoDestino: 5);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 3,  nodoDestino: 4);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 5,  nodoDestino: 1);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 4,  nodoDestino: 0);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 6,  nodoDestino: 3);
```

Lista de adyacencia:

```
_____________________ LISTA DE ADYACENCIA GRAFO DIRIGIDO N# 0 ______
B → F → null
C → B → D → F → null
D → null
E → D → F → B → null
F → B → null
```

Matriz de adyacencia:

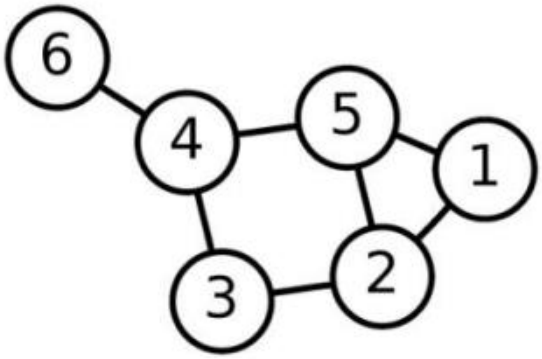
```
_____________ MATRIZ DE ADYACENCIA GRAFO DIRIGIDO N# 0 ___________
 false  false  false  false  true
 true  false  true  false  true
 false  false  false  false  false
 true  false  true  false  true
 true  false  false  false  false
```

Recorrido en profundidad:

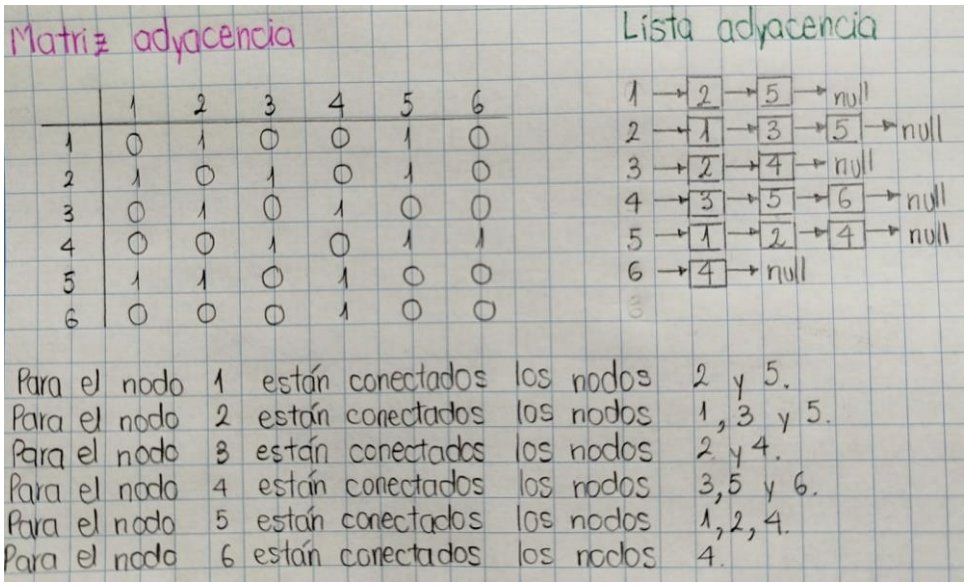
```
CASO 6 Imprimir recorrido en profundidad grafo Dirigido ________
 B
 F B
 B

 C
 D C
 C

 E

La salida del recorrido es la siguiente: [F, B, D, C, E]
```

3. Ejercicio 3: Identificación de conexiones

Dado el siguiente grafo:



Se realizó la matriz y lista de adyacencia para determinar cuántos nodos están conectados a un nodo específico, teniendo como resultado lo que se muestra a continuación.



Implementación en código

Creamos los nodos:

```
NodoGrafo<String> nodo1 = new NodoGrafo<String>( infoNodo: "1");
NodoGrafo<String> nodo2 = new NodoGrafo<String>( infoNodo: "2");
NodoGrafo<String> nodo3 = new NodoGrafo<String>( infoNodo: "3");
NodoGrafo<String> nodo4 = new NodoGrafo<String>( infoNodo: "4");
NodoGrafo<String> nodo5 = new NodoGrafo<String>( infoNodo: "5");
NodoGrafo<String> nodo6 = new NodoGrafo<String>( infoNodo: "6");
```

Apuntamos los nodos:

```
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 0,  nodoDestino: 1);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 0,  nodoDestino: 4);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 1,  nodoDestino: 2);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 1,  nodoDestino: 4);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 2,  nodoDestino: 3);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 3,  nodoDestino: 4);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 3,  nodoDestino: 5);
```

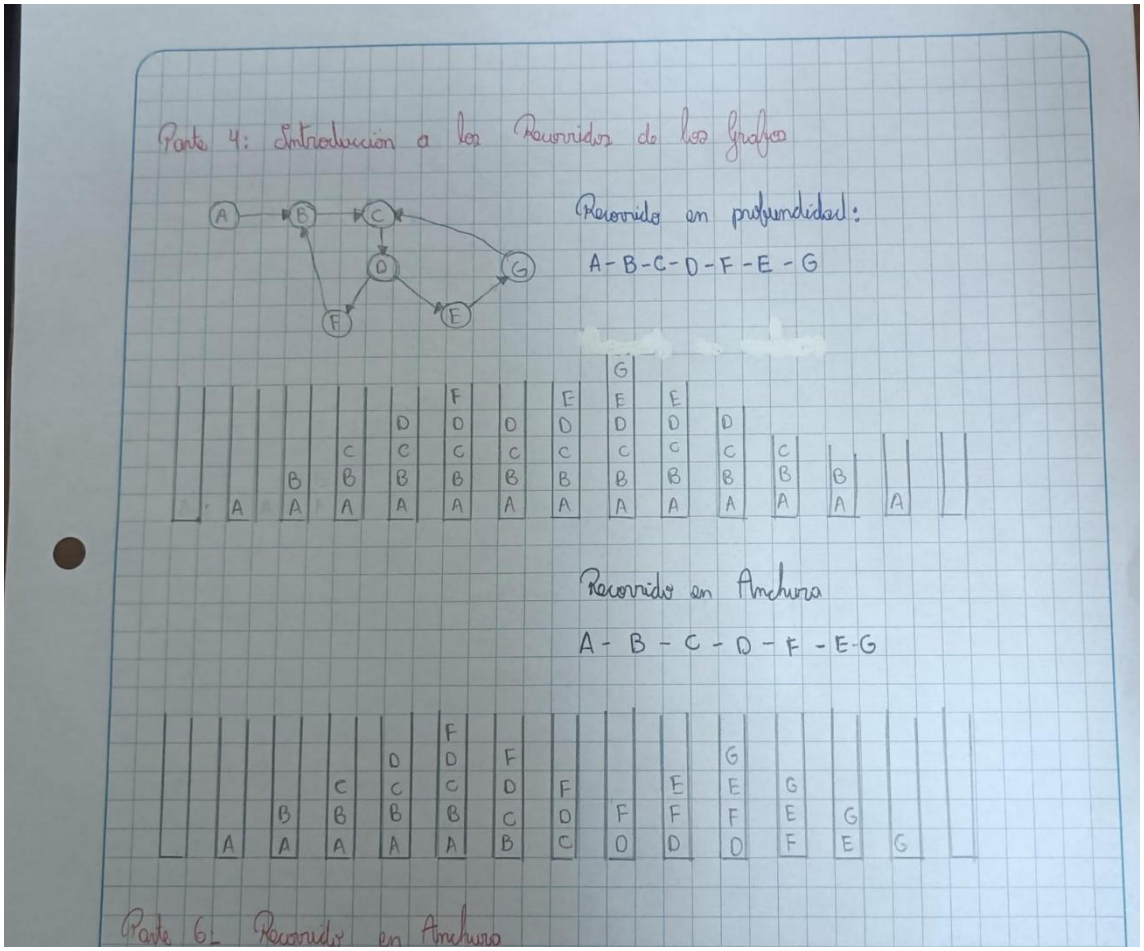
Lista de adyacencia:

```
____________________ LISTA DE ADYACENCIA GRAFO NO DIRIGIDO N# 0 ____________________
1 → 2 → 5 → null
2 → 1 → 3 → 5 → null
3 → 2 → 4 → null
4 → 3 → 5 → 6 → null
5 → 1 → 2 → 4 → null
6 → 4 → null
```

Matriz de adyacencia:

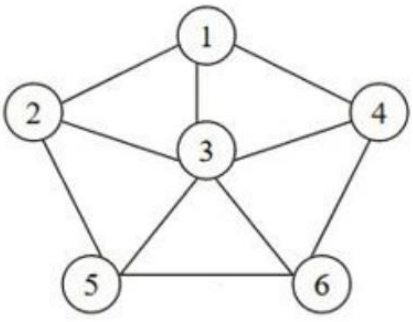
```
_____________ MATRIZ DE ADYACENCIA GRAFO NO DIRIGIDO N# 0 ____________________
false  true  false  false  true  false
true  false  true  false  true  false
false  true  false  true  false  false
false  false  true  false  true  true
true  true  false  true  false  false
false  false  false  true  false  false
```

4. Ejercicio 4: Recorridos de grafos



5. Ejercicio 5: Recorrido en Profundidad (DFS)

Dado el siguiente grafo, se realizó un recorrido en profundidad a partir del nodo 1.



Implementación en Código:
Creamos los nodos:

```
NodoGrafo<String> nodo1 = new NodoGrafo<~>( infoNodo: "1");
NodoGrafo<String> nodo2 = new NodoGrafo<~>( infoNodo: "2");
NodoGrafo<String> nodo3 = new NodoGrafo<~>( infoNodo: "3");
NodoGrafo<String> nodo4 = new NodoGrafo<~>( infoNodo: "4");
NodoGrafo<String> nodo5 = new NodoGrafo<~>( infoNodo: "5");
NodoGrafo<String> nodo6 = new NodoGrafo<~>( infoNodo: "6");
```

Añadimos los nodos y dirigimos sus aristas:

```
//Agregamos los nodos
grafoNoDirigido0.agregarNodo(nodo1);
grafoNoDirigido0.agregarNodo(nodo2);
grafoNoDirigido0.agregarNodo(nodo3);
grafoNoDirigido0.agregarNodo(nodo4);
grafoNoDirigido0.agregarNodo(nodo5);
grafoNoDirigido0.agregarNodo(nodo6);
//Dirigimos los nodos
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 0,  nodoDestino: 1);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 0,  nodoDestino: 2);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 0,  nodoDestino: 3);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 1,  nodoDestino: 0);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 1,  nodoDestino: 2);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 1,  nodoDestino: 4);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 3,  nodoDestino: 2);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 3,  nodoDestino: 0);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 3,  nodoDestino: 5);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 5,  nodoDestino: 3);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 5,  nodoDestino: 2);
grafoNoDirigido0.dirigirAristaANodo( nodoOrigen: 5,  nodoDestino: 4);
```

Lista de Adyacencia:

```
-------------------- LISTA DE ADYACENCIA GRAFO NO DIRIGIDO N# 0 --------------------
1 → 2 → 3 → 4 → null
2 → 1 → 3 → 5 → null
3 → 1 → 2 → 4 → 6 → null
4 → 1 → 3 → 6 → null
5 → 2 → 6 → null
6 → 4 → 3 → 5 → null
```

Matriz de adyacencia:

```
_____________ MATRIZ DE ADYACENCIA GRAFO NO DIRIGIDO N# 0 _______
 false  true  true  true  false  false
 true  false  true  false  true  false
 true  true  false  true  false  true
 true  false  true  false  false  true
 false  true  false  false  false  true
 false  false  true  true  true  false
```

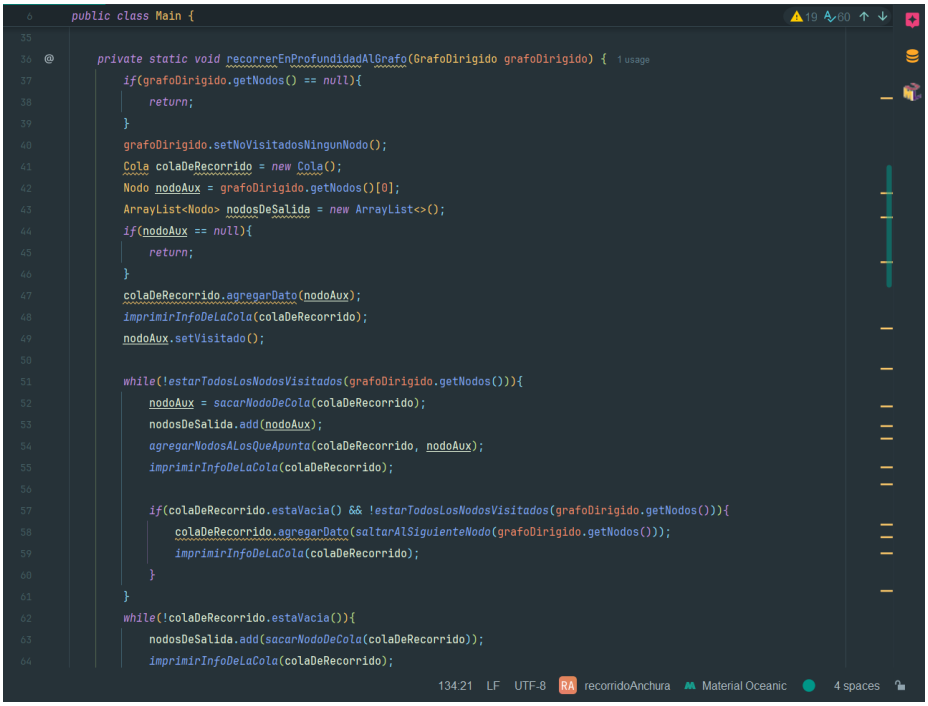
Ejercicio de Pila:

Resultado en consola del recorrido:

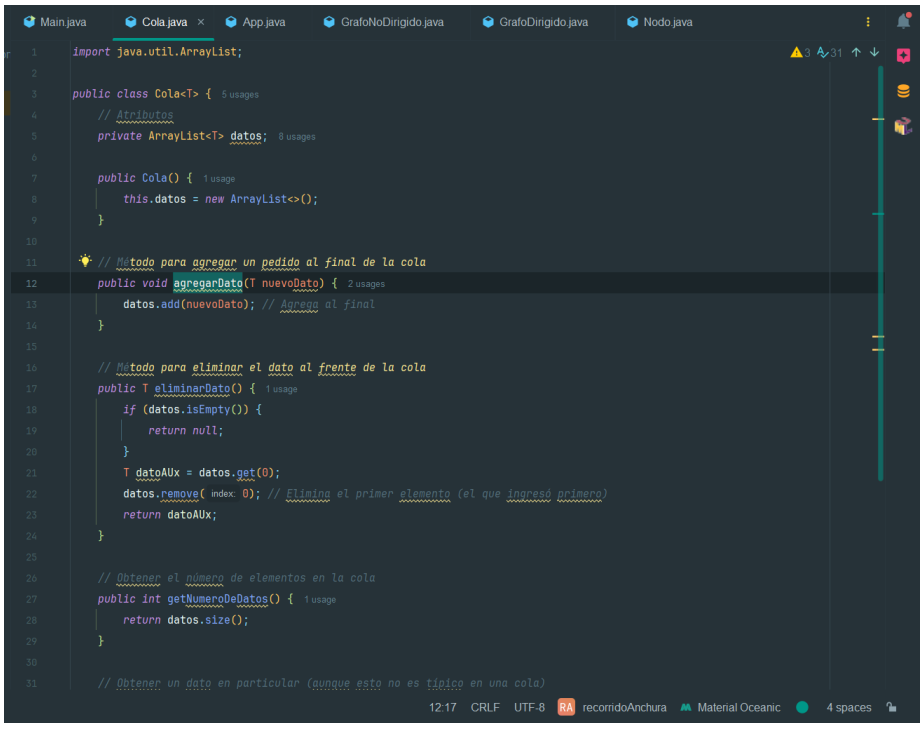
```
CASO 7 Imprimir recorrido en profundidad grafo No Dirigido _________________
 1
 2 1
 3 2 1
 4 3 2 1
 6 4 3 2 1
 5 6 4 3 2 1
 6 4 3 2 1
 4 3 2 1
 3 2 1
 2 1
 1

La salida del recorrido es la siguiente: [5, 6, 4, 3, 2, 1]
```

6. Ejercicio 6: Recorrido en anchura

El algoritmo realizado para simular este recorrido se muestra a continuación.

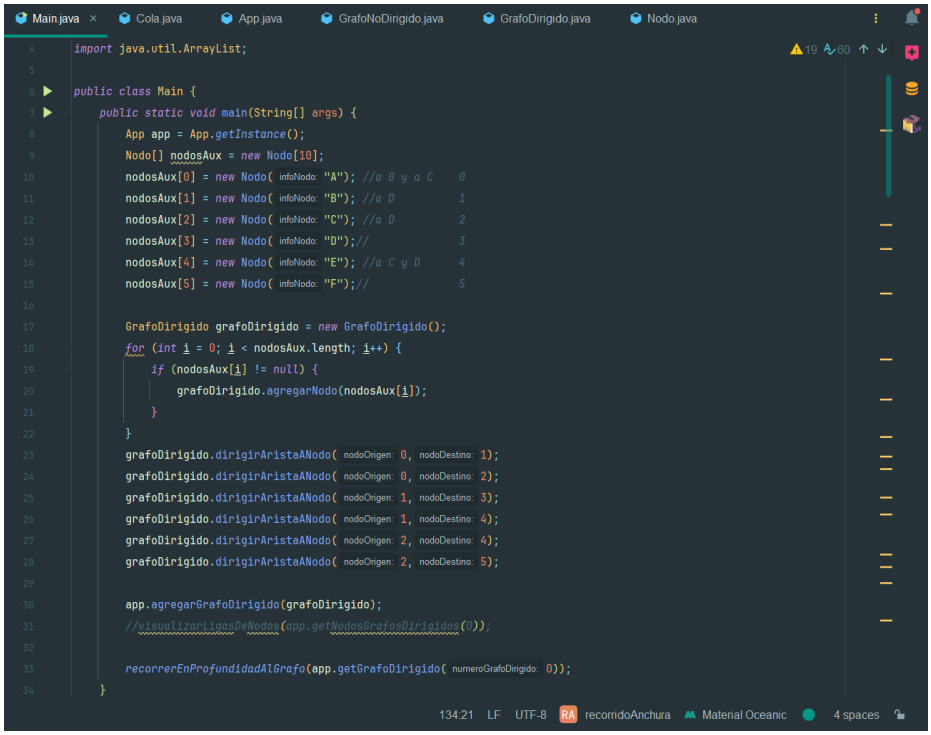


```java
public class Main {

    private static void recorrerEnProfundidadAlGrafo(GrafoDirigido grafoDirigido) {
        if(grafoDirigido.getNodos() == null){
            return;
        }
        grafoDirigido.setNoVisitadosNingunNodo();
        Cola colaDeRecorrido = new Cola();
        Nodo nodoAux = grafoDirigido.getNodos()[0];
        ArrayList<Nodo> nodosDeSalida = new ArrayList<>();
        if(nodoAux == null){
            return;
        }
        colaDeRecorrido.agregarDato(nodoAux);
        imprimirInfoDeLaCola(colaDeRecorrido);
        nodoAux.setVisitado();

        while(!estarTodosLosNodosVisitados(grafoDirigido.getNodos())){
            nodoAux = sacarNodoDeCola(colaDeRecorrido);
            nodosDeSalida.add(nodoAux);
            agregarNodosALosQueApunta(colaDeRecorrido, nodoAux);
            imprimirInfoDeLaCola(colaDeRecorrido);

            if(colaDeRecorrido.estaVacia() && !estarTodosLosNodosVisitados(grafoDirigido.getNodos())){
                colaDeRecorrido.agregarDato(saltarAlSiguienteNodo(grafoDirigido.getNodos()));
                imprimirInfoDeLaCola(colaDeRecorrido);
            }
        }
        while(!colaDeRecorrido.estaVacia()){
            nodosDeSalida.add(sacarNodoDeCola(colaDeRecorrido));
            imprimirInfoDeLaCola(colaDeRecorrido);
```



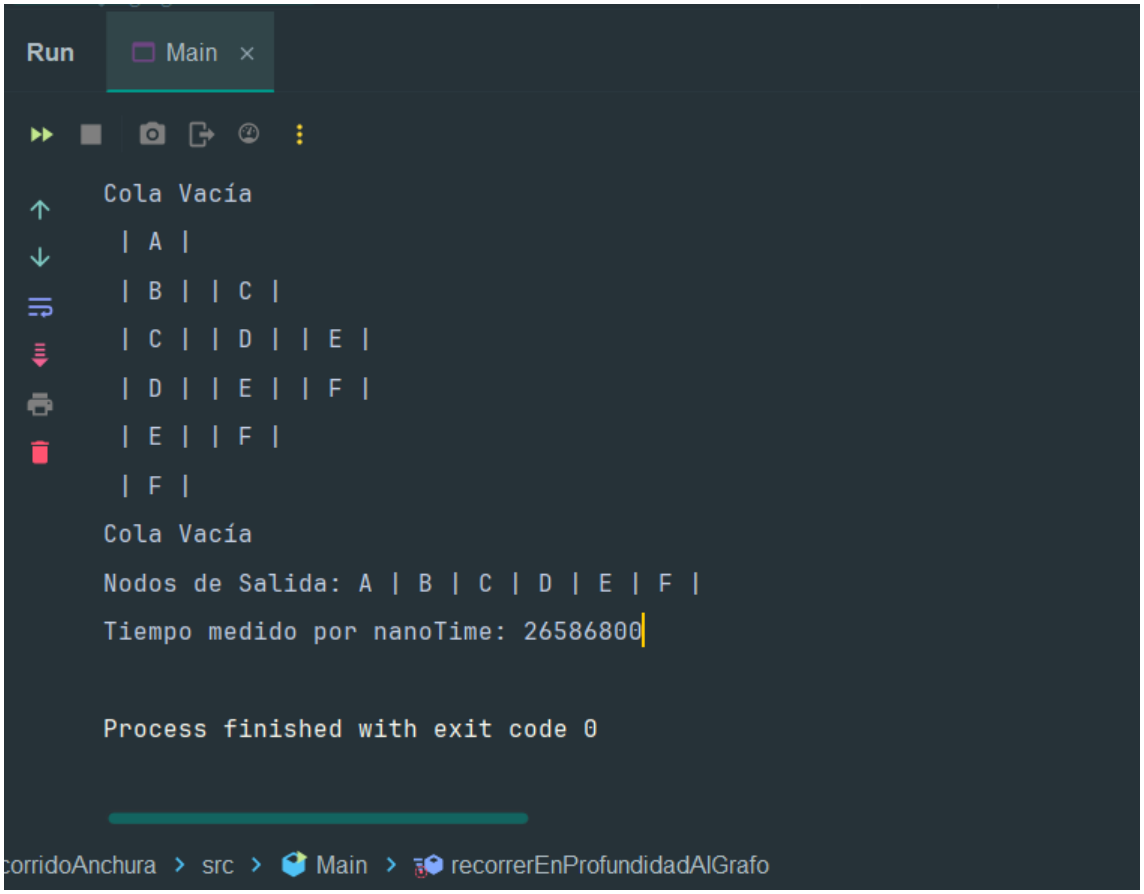
```java
import java.util.ArrayList;

public class Cola<T> {
    // Atributos
    private ArrayList<T> datos;

    public Cola() {
        this.datos = new ArrayList<>();
    }

    // Método para agregar un pedido al final de la cola
    public void agregarDato(T nuevoDato) {
        datos.add(nuevoDato); // Agrega al final
    }

    // Método para eliminar el dato al frente de la cola
    public T eliminarDato() {
        if (datos.isEmpty()) {
            return null;
        }
        T datoAUx = datos.get(0);
        datos.remove( index: 0); // Elimina el primer elemento (el que ingresó primero)
        return datoAUx;
    }

    // Obtener el número de elementos en la cola
    public int getNumeroDeDatos() {
        return datos.size();
    }

    // Obtener un dato en particular (aunque esto no es típico en una cola)
```

Al momento de colocar los datos para comprobar el recorrido en anchura se puede evidenciar el correcto funcionamiento de la cola de igual forma que cada uno de los nodos es visitado en el orden correcto del algoritmo visto en clase.



Primero se ingresa los nodos que va a tener el grafo dirigido y se los enlaza de acuerdo con el ejercicio anterior.

A continuación, se puede ver como los elementos van entrando a la cola y saliendo para recorrer todos los nodos y guardando el orden de salida de cada uno.



De igual forma se analizo en base a la función System.nanoTime() la duración precisa en la que el método puede realizar su función de esta manera corroborando el gasto computacional.

Análisis de resultados:

La matriz de adyacencia utiliza una estructura bidimensional donde cada fila y columna representan los nodos del grafo. Si hay una arista entre dos nodos, la celda correspondiente se marca, lo que facilita la representación de grafos tanto dirigidos como no dirigidos.
Esta representación es particularmente útil para visualizar la conexión directa entre nodos, lo que puede ser beneficioso en problemas que requieren la identificación de relaciones inmediatas entre elementos.

Con respecto al cálculo de los nanosegundos y milisegundos del código tenemos los siguientes resultados:

	Lista de Adyacencia	Matriz de Adyacencia	Recorrido Profundidad	Recorrido Anchura
Nanosegundos	4049700	1418700	1262600	3272900
Milisegundos	4	1	1	3

Comparando la ejecución de los métodos de recorrido llegamos a la conclusión de que el recorrido a profundidad es mas eficiente porque se demora mucho menos comparado con el de anchura que se demora 3 veces más en su ejecución.

La lista de adyacencia consiste en un arreglo donde cada índice representa un nodo y contiene una lista de nodos adyacentes. Esta estructura permite una representación más dinámica de las conexiones del grafo. Por lo que, es útil en aplicaciones donde se necesita acceder a los vecinos de un nodo de manera eficiente, facilitando operaciones como inserciones y eliminaciones de aristas.

Con respecto al recorrido en los grafos, el DFS explora los nodos comenzando desde un nodo inicial y avanzando hacia un nodo vecino antes de retroceder. Este algoritmo es útil en situaciones que requieren el descubrimiento de todos los nodos conectados a partir de un nodo inicial, como en la detección de componentes conectados o en la búsqueda de ciclos.

Mientras que, el BFS explora todos los nodos a un nivel antes de proceder al siguiente. Utiliza una cola para mantener el orden de los nodos a explorar. Este algoritmo es efectivo para encontrar el camino más corto en grafos no ponderados, siendo particularmente útil en contextos como la búsqueda en redes sociales o en aplicaciones de navegación.

Conclusiones y recomendaciones:

Conclusiones

En conclusión, un grafo es una estructura compuesta por nodos (vértices) y aristas (conexiones entre los nodos), y que estos componentes permiten modelar diversos problemas. Se reconoció que los grafos pueden ser dirigidos o no dirigidos, ponderados o no ponderados, dependiendo del contexto. Estos conceptos son fundamentales para abordar la resolución de problemas relacionados con redes, rutas y relaciones entre entidades, donde cada tipo de grafo puede proporcionar diferentes enfoques.

Al implementar las diferentes representaciones de grafos (matriz de adyacencia y lista de adyacencia), se notaron ventajas y desventajas en ambas. La matriz de adyacencia es intuitiva y fácil de entender para grafos pequeños, ya que proporciona un acceso directo a la existencia de una arista entre dos nodos. Sin embargo, para grafos dispersos (es decir, con pocas aristas), esta representación consume mucho espacio, lo que resulta ineficiente. Por otro lado, la lista de adyacencia es más eficiente en términos de memoria para grafos grandes y dispersos, pero requiere más tiempo para verificar si existe una arista entre dos nodos. En situaciones donde el grafo tiene muchas conexiones, una matriz de adyacencia sería más apropiada, mientras que, en grafos dispersos, una lista de adyacencia sería preferible por su ahorro de espacio.

Al implementar y aplicar los algoritmos de búsqueda DFS y BFS, se identificó que cada uno tiene aplicaciones distintas según el tipo de problema. El DFS es más adecuado para problemas que requieren explorar caminos completos (como en la búsqueda de soluciones en profundidad o detección de ciclos), ya que prioriza la exploración exhaustiva de cada rama antes de retroceder. El BFS, por otro lado, es útil para encontrar el camino más corto en grafos no ponderados, ya que explora todos los vecinos de un nodo antes de profundizar. Aunque BFS resultó más intuitivo

para comprender cómo se expande a través de los nodos nivel por nivel, DFS es más fácil de implementar recursivamente. Por lo tanto, BFS es ideal para problemas de optimización de rutas cortas, mientras que DFS es preferible para exploración o detección de conexiones profundas.

Recomendaciones

Se recomienda utilizar la matriz de adyacencia en grafos con muchas conexiones, ya que facilita un acceso rápido a las aristas. En grafos dispersos, es preferible optar por la lista de adyacencia debido a su eficiencia en el uso de memoria.

Se sugiere emplear DFS para explorar caminos en profundidad o detectar ciclos, y usar BFS cuando se necesite encontrar la ruta más corta en grafos no ponderados.

Es recomendable implementar ambos algoritmos y representaciones en problemas de la vida real, como redes sociales o rutas de transporte, para mejorar la comprensión de su comportamiento en distintos escenarios y de la utilidad de los grafos en la resolución de problemas complejos.

Se recomienda utilizar las funciones System.nanoTime() o System.currentTimeMillis() para medir el costo computacional y evaluar la viabilidad del código, lo que facilitará la identificación de áreas de mejora para futuras optimizaciones.

Bibliografía:

[1] A. Castro, «saberpunto.com,» 24 Septiembre 2024. [En línea]. Available: https://saberpunto.com/programacion/que-es-el-recorrido-de-grafos-recorrido-en-profundidad-y-anchura/#Recorrido_en_Anchura_(BFS). [Último acceso: 18 Octubre 2024].

[2] L. Llamas, «Luis Llamas,» 22 Abril 2024. [En línea]. Available: https://www.luisllamas.es/que-es-un-grafo/. [Último acceso: 17 Octubre 2024].

[3] ApInEm, «ApInEm Web,» 22 Febrero 2024. [En línea]. Available: https://www.apinem.com/grafos-en-programacion/#3-1-grafos-dirigidos. [Último acceso: 17 Octubre 2024].

[4] Grapheverywhere, «Graph Everywhere,» 8 Julio 2019. [En línea]. Available: https://www.grapheverywhere.com/tipos-de-grafos/. [Último acceso: 17 Octubre 2024].

[5] Khan Academy, «Khan Academy,» 18 Noviembre 2014. [En línea]. Available: https://es.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs. [Último acceso: 17 Octubre 2024].

[6] GeeksforGeeks, «GeeksforGeeks,» 12 Abril 2018. [En línea]. Available: https://www.geeksforgeeks.org/java-system-nanotime-vs-system-currenttimemillis/. [Último acceso: 18 Octubre 2024].