



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:* Tista García Edgar Ing.

*Asignatura:* Estructura de Datos y Algoritmos II

*Grupo:* 5

*No de Práctica(s):* 6-7

*Integrante(s):* Téllez González Jorge Luis

*No. de Equipo de  
cómputo empleado:* ---

*No. de Lista o Brigada:* X

*Semestre:* 2020-2

*Fecha de entrega:* 30/04/2020

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_



# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>3</b>
<b>3. Implementación de grafos en Java</b>	<b>4</b>
3.1. Grafos dirigidos . . . . .	5
3.2. Creación de nuevos grafos . . . . .	7
3.3. Grafos ponderados . . . . .	8
3.3.1. Funcionamiento . . . . .	9
3.4. Matrices de adyacencia . . . . .	10
<b>4. Breadth First Search</b>	<b>13</b>
4.1. Verificación de funcionamiento . . . . .	14
<b>5. Depth First Search</b>	<b>16</b>
5.1. Análisis de la implementación . . . . .	19
5.2. Verificación de funcionamiento . . . . .	20
<b>6. Conclusiones</b>	<b>22</b>

# 1. Introducción

La resolución de problemas cotidianos en diversos momentos de la historia ha traído consigo la aparición de nuevas áreas del conocimiento: tal es el caso de la *teoría de grafos*. La ciudad de Königsberg fue una ciudad prusiana del siglo XVIII de la cuál surgió el siguiente problema: Königsberg tiene al río *Pregel*, el cuál cruzaba la ciudad, a dos islas que se encontraban en el mismo río y a siete puentes que conectaban las dos partes de la ciudad con el resto.

El problema consistía en comenzar en un punto arbitrario, pasar por cada uno de los siete puntos sin repetir alguno, y volver al punto de partida.

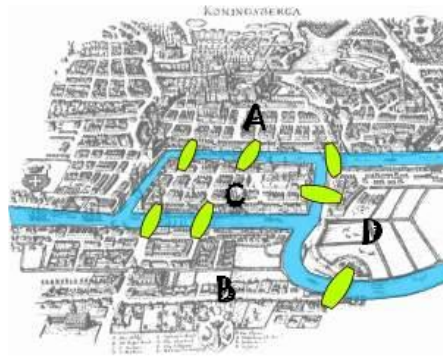


Figura 1: *El problema de los puentes de Königsberg.*

Tras diversos intentos de solucionar el problema, se llegó a la conclusión empírica de la irresolubilidad del problema. Sin embargo, el matemático *Leonhard Euler* simplificó el problema a únicamente puntos y líneas; convirtiendo cada uno de los territorios en los que los puentes dividieron la ciudad en **vértices** y los puentes mismos se convirtieron en conexiones o **aristas**.

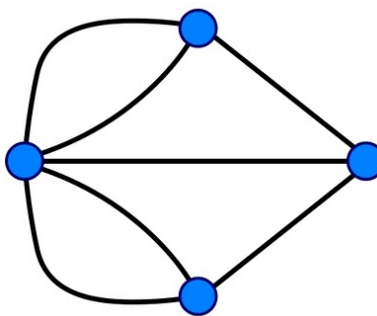


Figura 2: *Grafo representativo del problema.*

Euler, por medio de esta abstracción, llegó a la conclusión de que, para poder recorrer tal sistema, los vértices intermedios deben de poseer un número par de aristas asociadas, es decir, *deben tener una*

*entrada y una salida*. Esto no incluye a los vértices de inicio y fin, pues nunca se 'entra' al vértice inicial y tampoco se 'sale' al vértice de llegada. La generalización de Euler concluye en que, debido a que los vértices intermedios tienen un número impar de aristas, es imposible cumplir la tarea propuesta por el problema.

A raíz del trabajo realizado por Euler, surgieron diversas investigaciones matemáticas que dieron forma a la actual *Teoría de Grafos*. Gracias al trabajo desarrollado en esta teoría, la humanidad ha tenido un enorme avance en el análisis de grandes volúmenes de datos y representa una parte fundamental en la solución de problemas complejos; como la elección de la mejor ruta a seguir en una determinada ciudad.

Entre otras aplicaciones, se encuentran las siguientes:

- Representación computacional.
- Ámbito bancario y detección de fraudes.
- Análisis en redes sociales.

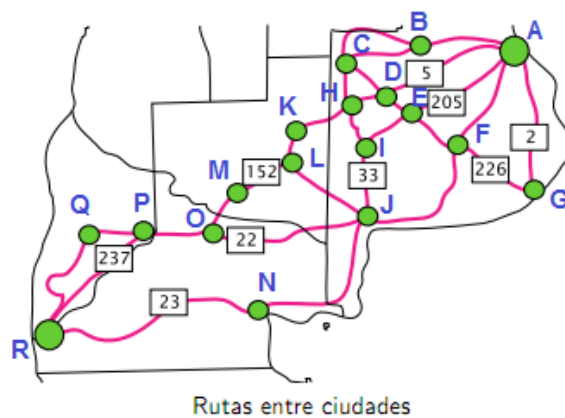


Figura 3: Análisis de grafos aplicado a las rutas entre ciudades.

En el siguiente reporte se implementarán y analizarán diversas representaciones para un grafo: no-dirigido, dirigido o ponderado. Así mismo, se abordarán 2 algoritmos de *Búsqueda por Expansión*: **Breadth First Search** y **Depth First Search**.

## 2. Objetivos

- El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para comprender el algoritmo de búsqueda por expansión.

- El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para comprender el algoritmo de búsqueda en profundidad.

### 3. Implementación de grafos en Java

El proyecto creado en **NetBeans** contiene a la clase principal y a la clase *Grafo*:

- La clase *Grafo* contiene dos atributos: un entero  $V$  que representa los vértice o el número total de nodos que serán insertados en un objeto creado de tipo *Grafo* y una referencia a una lista ligada de enteros que funcione como estructura auxiliar en el proceso de creación.

```
int V;  
LinkedList<Integer> adjArray[];
```

Figura 4: Atributos de *Grafo*.

- El método constructor recibe un entero, que representa el número total de vértices, y lo asigna al atributo  $V$  del objeto creado. Posteriormente, se crea una lista ligada denominada *adjArray* que contiene en su interior otra lista ligada.

```
for (int i = 0; i < v; ++i) {  
    adjArray[i] = new LinkedList();  
}
```

Figura 5: Anidación de listas.

- El método **addEdge** recibe un nodo entero *head* y un entero  $w$  que representa al vértice que será ligado al vértice *head* recibido. Para esto, se utiliza la lista aninada para crear una relación entre los nodos.

```
adjArray[v].add(w);  
adjArray[w].add(v);
```

Figura 6: Proceso de ligado entre vértices.

- El método **printGraph** recibe un objeto de tipo *Grafo* e, iterando en todos los vértices por medio de un ciclo **for**, se imprime la lista de adyacencia de cada vértice, es decir, a qué nodos conecta el nodo *head* actual.

```
for (Integer node: graph.adjArray[v]) {  
    System.out.print(" -> "+node);  
}
```

Figura 7: Iteración *for-each* sobre la lista de adyacencia del vértice *head* actual.

Al compilar y ejecutar el código proporcionado, se obtiene la siguiente salida:

```
Output - Practica6[TellezJorge] (run)
run:
Lista de Adyacencia del vertice 0
0
-> 1 -> 4

Lista de Adyacencia del vertice 1
1
-> 0 -> 2 -> 3 -> 4

Lista de Adyacencia del vertice 2
2
-> 1 -> 3

Lista de Adyacencia del vertice 3
3
-> 1 -> 2 -> 4

Lista de Adyacencia del vertice 4
4
-> 0 -> 1 -> 3

BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 8: Salida de la lista de adyacencia de cada nodo presente en el grafo.

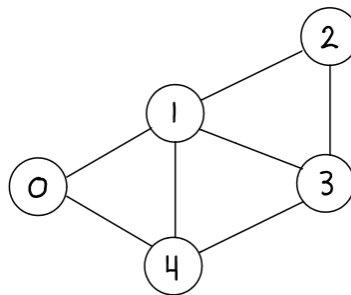


Figura 9: Representación gráfica del grafo creado.

Es importante destacar que este grafo **no es dirigido**. Por otra parte, la lista de adyacencia únicamente garantiza la conexión entre el head y los nodos de la lista; no implica que los nodos de la lista de adyacencia se encuentren conectados entre sí.

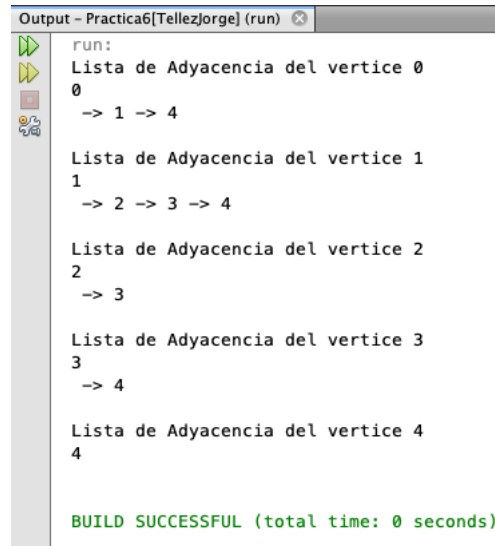
### 3.1. Grafos dirigidos

La implementación anterior puede volverse **dirigida** únicamente eliminando una línea de código en el método **addEdge**. De esta forma, la lista de adyacencia únicamente mostrará conexiones en una única dirección; de acuerdo a la forma en la que se realizaron las conexiones por medio del método **addEdge**:

`addEdge(0,1)`, por ejemplo, crea una conexión que va del vértice 0 al vértice 1, pero no del vértice 1 al vértice 0.

```
void addEdge(int v, int w) {  
    adjArray[v].add(w);  
}
```

Figura 10: *Relación múltiple eliminada entre nodos.*



```
Output - Practica6[Tellezjorge] (run) x  
run:  
Lista de Adyacencia del vertice 0  
0  
-> 1 -> 4  
  
Lista de Adyacencia del vertice 1  
1  
-> 2 -> 3 -> 4  
  
Lista de Adyacencia del vertice 2  
2  
-> 3  
  
Lista de Adyacencia del vertice 3  
3  
-> 4  
  
Lista de Adyacencia del vertice 4  
4  
  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 11: *Salida obtenida con la modificación realizada.*

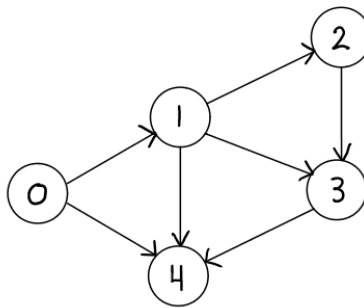
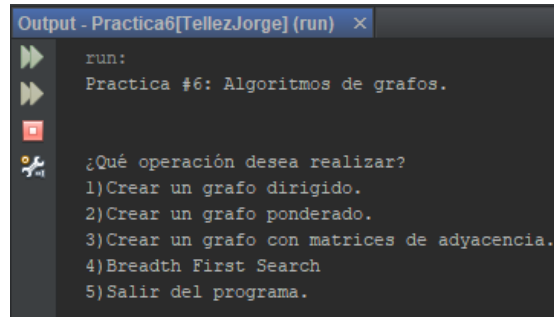


Figura 12: *Versión dirigida del grafo creado anteriormente.*

### 3.2. Creación de nuevos grafos

Se ha añadido la opción de permitir al usuario del programa crear un nuevo grafo dirigido, indicando la cantidad de nodos y aristas que lo conectarán.

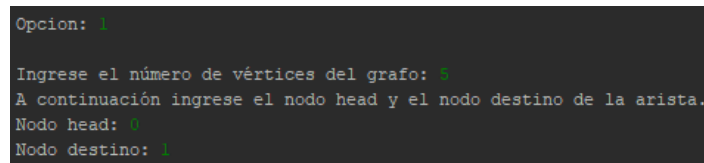


```
Output - Practica6[TellezJorge] (run) X
run:
Practica #6: Algoritmos de grafos.

¿Qué operación desea realizar?
1) Crear un grafo dirigido.
2) Crear un grafo ponderado.
3) Crear un grafo con matrices de adyacencia.
4) Breadth First Search
5) Salir del programa.
```

Figura 13: *Menú de selección creado.*

Para ello, se solicita en primer lugar el número total de vértices que formarán parte del grafo. Posteriormente, se crea un objeto de tipo *Grafo* y, a continuación, se solicita al nodo *head* y el nodo *destino* de conexión.



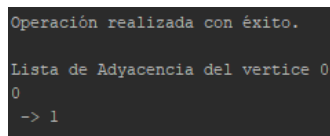
```
Opcion: 1
Ingrese el número de vértices del grafo: 5
A continuación ingrese el nodo head y el nodo destino de la arista.
Nodo head: 0
Nodo destino: 1
```

Figura 14: *Creación de aristas o 'conexiones' entre vértices.*

Verificando por medio de una condicional que los valores introducidos sean coherentes con el número de vértices introducidos al inicio, se utiliza el método **addEdge** para crear una arista entre ambos nodos. Finalmente, se imprime la lista de adyacencia del grafo creado y se brinda la opción de introducir una nueva arista.

```
if (head <= V && nodoDestino <= V) {
    grafoDirigido.addEdge(head, nodoDestino);
}
```

Figura 15: *Condición necesaria para crear nuevas aristas en el grafo.*



```
Operación realizada con éxito.
Lista de Adyacencia del vertice 0
0
-> 1
```

Figura 16: *Impresión de la lista de adyacencia tras crear una arista.*



```
¿Desea ingresar otra arista?
Ingrese cualquier valor para continuar o introduzca (0) para salir.
Opcion:
```

Figura 17: Opción de añadir nuevas aristas tras crear una.

### 3.3. Grafos ponderados

Un grafo ponderado comparte muchas características con respecto a los grafos trabajados con anterioridad; siendo el valor o costo asignado a las aristas la única diferencia notable. Es decir, los grafos ponderados **heredan** características de los objetos de tipo *Grafo*. Con lo anterior en mente, se crea una nueva clase denominada *GrafoPonderado* que, por medio de herencia, recibirá los mismos atributos y métodos definidos para la clase *Grafo*.

```
public class GrafoPonderado extends Grafo{

    public GrafoPonderado(int v) {
        super(v);
    }
}
```

Figura 18: Relación de herencia entre la clase creada y la clase padre.

El constructor para la clase creada es idéntico al de la clase *Grafo*: en el interior del constructor se hace referencia directa al constructor de la clase padre por medio de la palabra reservada **super**, enviando como parámetro el número de vértices introducido al crear un objeto *GrafoPonderado*.

Se modificaron los siguientes métodos en la nueva clase:

- En el método **addEdge** se introdujo una nueva instrucción **add** tal que, tras añadir a la lista de adyacencia de un vértice dado un nodo, a continuación se introduce el *costo* o *peso* de la arista introducida; valor que es solicitado al usuario.

```
adjArray[v].add(w);
adjArray[v].add(peso);
```

Figura 19: 'Asignación de costo' a la arista creada.

- El método **printGraph** funciona de manera similar con respecto a los grafos anteriores. Sin embargo, la diferencia radica en que, tras imprimirse la lista de adyacencia de cada vértice, se imprimen los costos asociados a cada arista creada.

```
for(int i=0; i<graph.adjArray[v].size(); i=i+2){
    System.out.print("\n -> "+graph.adjArray[v].get(i)+
        " Costo del arista: "+graph.adjArray[v].get(i+1));
}
```

Figura 20: Líneas de impresión de la lista de adyacencia y el costo de cada arista creada.

Debido a que el costo se almacena en el índice siguiente al vértice destino, la variable iteradora se incrementa en dos unidades para imprimir cada conexión y su costo asociado de forma sucesiva.

### 3.3.1. Funcionamiento

Con el fin de poner a prueba las modificaciones realizadas, se ha creado el mismo grafo propuesto en el código proporcionado, añadiendo un costo asociado a cada arista presente.

```
Lista de Adyacencia del vertice 0
0

-> 1 Costo del arista: 1
-> 4 Costo del arista: 2

Lista de Adyacencia del vertice 1
1

-> 2 Costo del arista: 3
-> 3 Costo del arista: 4
-> 4 Costo del arista: 5

Lista de Adyacencia del vertice 2
2

-> 3 Costo del arista: 6

Lista de Adyacencia del vertice 3
3

-> 4 Costo del arista: 7

Lista de Adyacencia del vertice 4
4
```

Figura 21: Impresión de la lista de adyacencia y el costo asociado de cada arista.

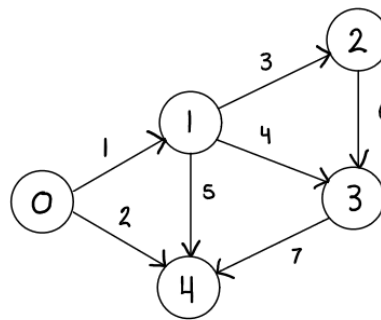


Figura 22: Representación gráfica del grafo creado junto con el costo de cada una de sus aristas.

### 3.4. Matrices de adyacencia

En la clase *Grafo* se ha introducido un nuevo método denominado **matrizGrafo**, el cual recibe el número de vértices que componen al grafo a representar matricialmente. El método comienza creando e inicializando un arreglo multidimensional denominado *matrizAdyacencia* de orden  $[Vertices][Vertices]$  con un valor 0 en cada una de sus posiciones.

```
Integer matrizAdyacencia[][] = new Integer[Vertices][Vertices];  
  
for (int i = 0; i < Vertices; i++) {  
    for (int j = 0; j < Vertices; j++) {  
        matrizAdyacencia[i][j] = 0;  
    }  
}
```

Figura 23: Inicialización de la matriz de adyacencia.

Hecho lo anterior, se le pregunta al usuario si desea añadir una arista al grafo; conexiones que quedarán representadas en la matriz creada. Por medio de un **if**, se puede proceder a introducir la arista o salir de la ejecución del método.

```
Desea agregar una arista?  
Presione (1) para confirmar o ingrese cualquier otro valor para salir.
```

Figura 24: Selección: crear una nueva arista o salir de la opción.

A continuación, se procede a solicitar el vértice o nodo *head* y el *nodoDestino* de conexión de forma similar a las implementaciones anteriores. Una vez que se verifica la validez de los datos introducidos, se procede a incrementar en una unidad la matriz en la posición: *matrizAdyacencia[head][nodoDestino]*. Finalmente, se imprime la matriz de adyacencia por medio de ciclos **for** anidados y un listado de adyacencia ubicada en la parte inferior haciendo uso de la propia matriz para recuperar la adyacencia entre nodos.

```
for (int i = 0; i < Vertices; i++) {  
    System.out.print("El vértice " + i + " se encuentra conectado a: ");  
    for (int j = 0; j < Vertices; j++) {  
        if (matrizAdyacencia[i][j] == 1) {  
            System.out.print(j + " -> ");  
        }  
    }  
}
```

Figura 25: Anidación utilizada para crear el listado de adyacencia utilizando la matriz generada.

Una vez que se introduce una arista, el usuario tiene la opción de agregar otra arista o salir de la ejecución si así lo desea. Por otra parte, si se introduce un vértice fuera del rango introducido inicialmente, el programa arrojará un mensaje de error.

```

Ingrese el número de vértices del grafo: 5
Matriz de adyacencia generada.

Desea agregar una arista?
Presione (1) para confirmar o ingrese cualquier otro valor para salir.

Opcion: 1
A continuación ingrese el nodo head y el nodo destino de la arista.
Nodo head: 6
Nodo destino: 7
Error: Ingrese un vértice existente en el grafo actual.

```

Figura 26: Mensaje de error obtenido tras introducir vértices inexistentes.

La implementación descrita anteriormente implementa un grafo dirigido. Con el fin de probar la validez de la misma, se utilizará como ejemplo el grafo utilizado en secciones anteriores.

```

A continuación ingrese el nodo head y el nodo destino de la arista.
Nodo head: 3
Nodo destino: 4
Matriz de adyacencia actual:

0 1 0 0 1
0 0 1 1 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0

El vértice 0 se encuentra conectado a: 1 -> 4 ->
El vértice 1 se encuentra conectado a: 2 -> 3 -> 4 ->
El vértice 2 se encuentra conectado a: 3 ->
El vértice 3 se encuentra conectado a: 4 ->
El vértice 4 se encuentra conectado a:

¿Desea ingresar otra arista? Ingrese cualquier valor para continuar o introduzca (0) para salir.

```

Figura 27: Impresión de la matriz de adyacencia y su lista de adyacencia.

	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>
<u>0</u>	0	1	0	0	1
<u>1</u>	0	0	1	1	1
<u>2</u>	0	0	0	1	0
<u>3</u>	0	0	0	0	1
<u>4</u>	0	0	0	0	0

Figura 28: Matriz de adyacencia del grafo propuesto.

La creación de grafos **no dirigidos** por medio de esta implementación se logra añadiendo una condicional tal que, si el vértice head es diferente al vértice o nodo destino, también se incrementa en una unidad `matrizAdyacencia[nodoDestino][head]`. Esta modificación se incluye en un método denominado **matrizGrafoND**.

```
if (head <= Vertices && nodoDestino <= Vertices) {
    matrizAdyacencia[head][nodoDestino]++;

    if (head != nodoDestino) {
        matrizAdyacencia[nodoDestino][head]++;
    }
}
```

Figura 29: *Matriz de adyacencia del grafo propuesto.*

El resultado, introduciendo el mismo grafo en su versión no dirigida, es el siguiente:

```
A continuación ingrese el nodo head y el nodo destino de la arista.
Nodo head: 3
Nodo destino: 4
Matriz de adyacencia actual:

0 1 0 0 1
1 0 1 1 1
0 1 0 1 0
0 1 1 0 1
1 1 0 1 0

El vértice 0 se encuentra conectado a: 1 -> 4 ->
El vértice 1 se encuentra conectado a: 0 -> 2 -> 3 -> 4 ->
El vértice 2 se encuentra conectado a: 1 -> 3 ->
El vértice 3 se encuentra conectado a: 1 -> 2 -> 4 ->
El vértice 4 se encuentra conectado a: 0 -> 1 -> 3 ->
```

Figura 30: *Grafo no dirigido en representación matricial: equivalente a [8].*

Finalmente, el último caso implementado corresponde a un grafo **dirigido y ponderado**. En la clase *GrafoPonderado* se ha sobrescrito el método **matrizGrafo** tal que, en vez de incrementarse en una unidad la posición `[head][nodoDestino]`, a esa posición se le asigna el costo de la arista que une a los vértices. Por otra parte, se ha creado e inicializado otra matriz a la que se le realizan las operaciones de incremento en una unidad presentes en los grafos dirigidos.

El resultado obtenido consiste en una matriz de adyacencia ponderada y una lista de adyacencia anexa a la matriz ponderada.

```

A continuación ingrese el nodo head y el nodo destino de la arista.
Nodo head: 3
Nodo destino: 4
Ingrese el costo de la arista: 7
Matriz de adyacencia actual:

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

El vértice 0 se encuentra conectado a: 1 -> 4 ->
El vértice 1 se encuentra conectado a: 2 -> 3 -> 4 ->
El vértice 2 se encuentra conectado a: 3 ->
El vértice 3 se encuentra conectado a: 4 ->
El vértice 4 se encuentra conectado a:

```

Figura 31: Grafo ponderado en representación matricial: equivalente a [21].

## 4. Breadth First Search

Se ha agregado a la clase *Grafo* el método **BFS**, el cual tiene la siguiente estructura:

- El método recibe en primer lugar el nodo o vértice a partir del cuál comenzará a recorrer el grafo en forma de capas. Posteriormente, se crea un arreglo booleano denominado *visited* de tamaño equivalente al número de vértices presentes en el grafo: este arreglo tiene como propósito marcar aquellos vértices o nodos que han sido visitados; por defecto todos los índices del arreglo tienen un valor de verdad *false*.

Hecho lo anterior, se crea una lista ligada llamada *queue*, la cual funcionará como estructura secundaria que almacenará un nodo marcado previamente como visitado.

```

visited[s] = true;
queue.add(s);

```

Figura 32: Se marca al nodo actual como visitado, y a continuación, se 'encola' al nodo.

A continuación, y siempre y cuando el tamaño de la lista *queue* sea diferente de 0, el método procede a desencolar el primer vértice en la cola y lo imprime.

Haciendo uso de la clase *Iterator* específica para iterar sobre envoltorios *Integer*, se utiliza la lista de adyacencia del grafo **adjArray** en su índice *s* y se comienza a iterar sobre la misma por medio del método **listIterator**. Mientras el iterador contenga un elemento siguiente, se almacena tal elemento en un entero *n*. Finalmente, si el vértice coincidente con el valor de *n* tiene asociado un valor *false*,

se marca el índice correspondiente al vértice como visitado en el arreglo booleano y se procede a encolar el vértice en la lista.

```
if (!visited[n]) {  
    visited[n] = true;  
    queue.add(n);  
}
```

Figura 33: Se marca al nodo siguiente como visitado, y se procede a encarlo.

## 4.1. Verificación de funcionamiento

El funcionamiento de la implementación de **BFS** proporcionada se puede resumir en los siguientes pasos:

1. Se inicia a partir de un vértice  $s$  dado de un grafo que se marca como visitado y se añade a la cola.
2. Se desencola el elemento introducido anteriormente y se imprime.
3. Se itera sobre los vértices adyacentes del elemento  $s$  desencolado. Si uno de los nodos adyacentes no ha sido visitado, se procede a marcarlo como visitado y a encarlo. Este proceso se repite hasta que la cola quede vacía.

Las diferencias con respecto al enfoque visto en clase son las siguientes:

- Esta implementación no devuelve una lista de nodos visitados, como el ejemplo teórico. En cambio, imprime los nodos conforme se desencolan de la lista *queue* y no devuelve nada.
- En la lista de nodos *visitados*, se asocia la posición entera de cada nodo con un valor *true* o *false*, con el fin de indicar si el nodo ha sido visitado o no.

Con el fin de probar el funcionamiento de la implementación, se han creado 2 grafos de prueba:

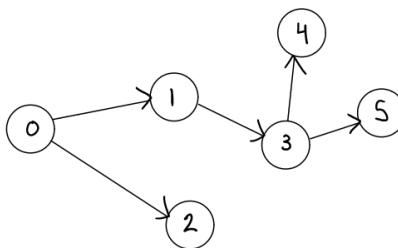


Figura 34: Primer grafo dirigido creado.

Con el grafo anterior, se ha utilizado el método **BFS** iniciando con el vértice 0 y el vértice 1. A continuación, se muestra la salida obtenida.

```
Creando un nuevo grafo...  
  
BFS: Grafo 1 - Vertice 0.  
  
[0]  
0 1 2 3 4 5  
BFS: Grafo 1 - Vertice 1.  
  
[1]  
1 3 4 5
```

Figura 35: *Impresión de nodos desencolados y 'visitados'.*

El segundo grafo creado es el siguiente:

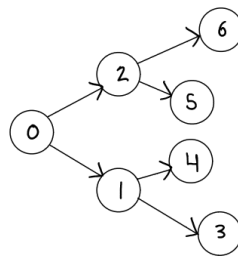


Figura 36: *Segundo grafo dirigido creado.*

```
Creando otro grafo...  
  
BFS: Grafo 2 - Vertice 0.  
  
[0]  
0 1 2 3 4 5 6  
BFS: Grafo 2 - Vertice 1.  
  
[1]  
1 3 4  
BFS: Grafo 2 - Vertice 2.  
  
[2]  
2 5 6
```

Figura 37: *Impresión de nodos desencolados y 'visitados' en el segundo grafo.*

Es importante destacar el segundo caso observado, el cuál corresponde a un tipo particular de grafo denominado **árbol**. El algoritmo **BFS**, iniciando a partir del vértice 0, recorre en orden ascendente cada uno de los vértices. Si se inicia a partir del vértice 1 o 2, la impresión en lista resultará en los vértices conectados a 1 y 2, o visto de otra forma, a sus nodos **hijos**.



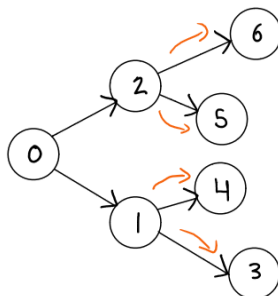


Figura 38: Ruta de impresión seguida por el algoritmo.

## 5. Depth First Search

Con el fin de probar el funcionamiento de la implementación proporcionada, se ha creado un nuevo proyecto en NetBeans denominado **Practica7[Tellez,Jorge]** y en el mismo se ha copiado la clase *Grafo* creada en el proyecto anterior y se ha anexado en el código de *DFS*.

En la clase principal se ha creado el siguiente grafo y se le ha aplicado **DFS** a partir de los vértices: 0, 3 Y 7. A continuación se muestra el grafo creado, su lista de adyacencia y el recorrido **DFS** a partir de los vértices señalados.

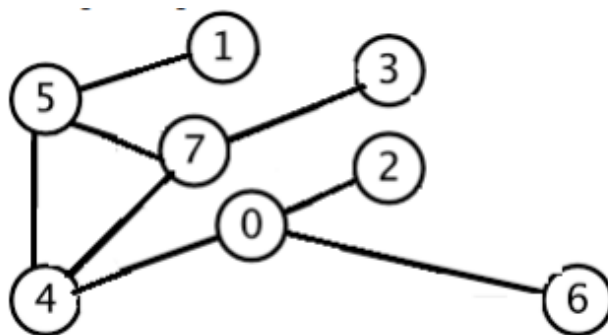


Figura 39: Grafo a recorrer por DFS.

```
Recorrido desde el vertice 0: 0 2 4 5 1 7 3 6
Recorrido desde el vertice 3: 3 7 4 0 2 6 5 1
Recorrido desde el vertice 7: 7 3 4 0 2 6 5 1
```

Figura 40: Recorrido DFS en el grafo a partir de 3 vértices distintos.

```

Lista de Adyacencia del vertice 0
0
-> 2 -> 4 -> 6

Lista de Adyacencia del vertice 1
1
-> 5

Lista de Adyacencia del vertice 2
2
-> 0

Lista de Adyacencia del vertice 3
3
-> 7

Lista de Adyacencia del vertice 4
4
-> 0 -> 5 -> 7

Lista de Adyacencia del vertice 5
5
-> 1 -> 4 -> 7

Lista de Adyacencia del vertice 6
6
-> 0

Lista de Adyacencia del vertice 7
7
-> 3 -> 4 -> 5

```

Figura 41: *Lista de adyacencia del grafo creado.*

A continuación, se aplicará el algoritmo **DFS** manualmente con el fin de corroborar la validez de la implementación:

Recorrido DFS:

+ Vértice 0:

Visitados:  $\{0, 2, 4, 5, 1, 7, 3, 6\}$

$S = 0$ * DFS(2) ✓ * DFS(4) ✓ DFS(6) ✓	$S = 2$ --- ✓	$S = 4$ * DFS(5) ✓ DFS(7) ✓
$S = 5$ ✓ * DFS(1) ✓ * DFS(7) ✓	$S = 1$ ✓ --- ✓	$S = 7$ * DFS(3) ✓
$S = 3$ ✓ --- ✓	$S = 6$ ✓ --- ✓	

Figura 42: *DFS a partir del vértice 0.*

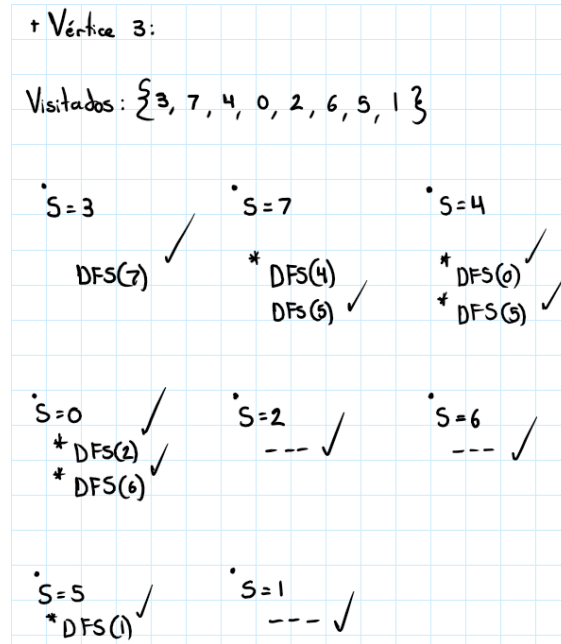


Figura 43: DFS a partir del vértice 3.

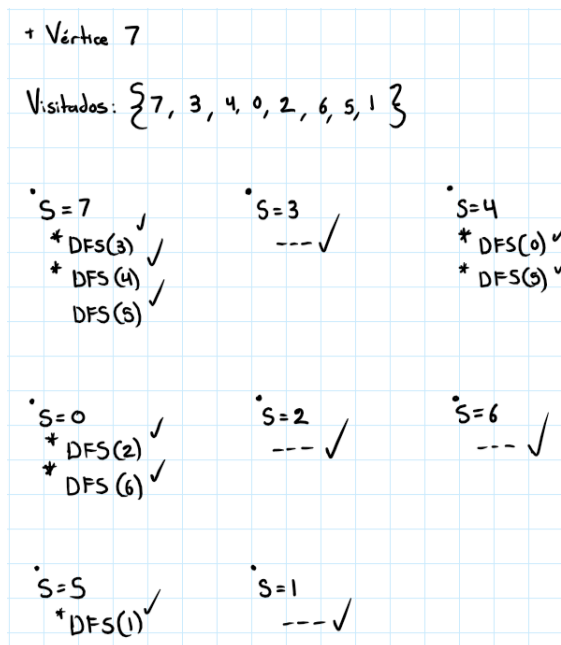


Figura 44: DFS a partir del vértice 7.

Los resultados obtenidos por medio de la aplicación manual del algoritmo coinciden con los presentados por la implementación en [40].

## 5.1. Análisis de la implementación

El código proporcionado contiene dos métodos, los cuáles serán detallados a continuación:

- El método **DFS** recibe un entero  $v$  que representa el vértice sobre el que se desea iniciar el recorrido **DFS** en el grafo. A continuación, se crea una lista booleana llamada *visited* inicializada con un tamaño igual al valor del atributo  $V$  del grafo; que indica el total de vértices del mismo.

Una diferencia a destacar con respecto al algoritmo teórico es que la lista creada asocia cada índice con un valor *true* o *false*. Así mismo, cada índice se encuentra asociado a su respectivo vértice numérico en el grafo.

```
boolean visited[] = new boolean[V];  
DFSUtil(v, visited);
```

Figura 45: *DFS a partir del vértice 7.*

- **DFSUtil** recibe el vértice  $v$  recibido previamente por el método **DFS** y la lista booleana *visited* creada. A continuación, se marca el vértice de inicio como visitado por medio de la lista booleana en el índice  $v$  y a continuación se imprime el vértice o nodo visitado junto con un espacio vacío.

Posteriormente, se crea un objeto de la clase *Iterator* que es utilizado para iterar sobre los nodos adyacentes a  $v$ . Por medio de un ciclo *while* y el método **hasNext()**, se recupera cada nodo adyacente  $n$  y, en caso de que no se encuentre marcado con *true* el índice  $n$  del arreglo *visited*, se realiza una llamada recursiva a **DFSUtil**, ahora recibiendo al vértice  $n$  y la lista *visited* actualizada.

```
Iterator<Integer> i = adjArray[v].listIterator();  
while (i.hasNext()) {  
    int n = i.next();  
    if (!visited[n]) {  
        DFSUtil(n, visited);  
    }  
}
```

Figura 46: *DFS a partir del vértice 7.*

El procedimiento seguido por la implementación es semejante al algoritmo teórico proporcionado aunque, a diferencia de la versión teórica, esta implementación **no devuelve** la lista de visitados.

```
Lista DFS (Nodo s):  
  
Lista visitados  
Marcar s como visitado  
visitados.add(s)  
  
Para todos (Nodos adyacentes w de s)  
  
Si w no está marcado  
  
DFS(w)  
  
return visitados
```

Figura 47: Algoritmo DFS.

## 5.2. Verificación de funcionamiento

Con el objetivo de validar una vez más la implementación de **DFS**, se han creado tres grafos diferentes en la clase principal. El primer grafo creado corresponde al mostrado en [34] en su versión no dirigida y se ha realizado el recorrido **DFS** a partir de los vértices 0, 3, 5:

```
Recorrido desde el vertice 0: 0 1 3 4 5 2  
Recorrido desde el vertice 3: 3 1 0 2 4 5  
Recorrido desde el vertice 5: 5 3 1 0 2 4
```

Figura 48: Recorrido DFS en el primer grafo creado a partir de 3 vértices distintos.

A continuación, se muestra el recorrido **DFS** manual a partir del vértice 0. El resultado corrobora la validez de la implementación:

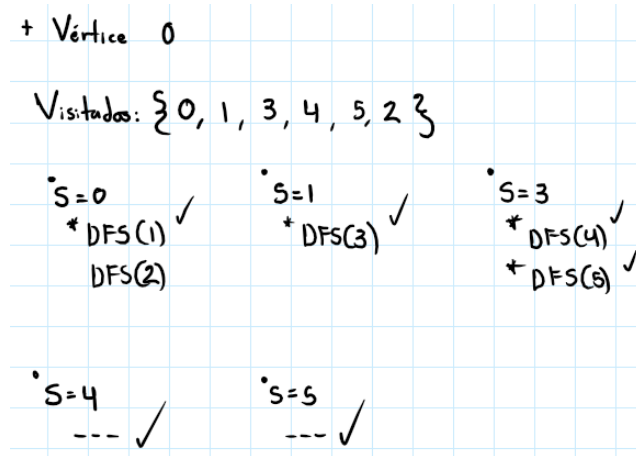


Figura 49: Recorrido DFS manual en el primer grafo a partir del vértice 0.

El segundo grafo creado corresponde al presentado en [36] en su forma no dirigida, y se ha recorrido con **DFS** a partir de los vértices 0, 3 y 5:

```
Recorrido desde el vertice 0: 0 1 3 4 2 5 6
Recorrido desde el vertice 3: 3 1 0 2 5 6 4
Recorrido desde el vertice 5: 5 2 0 1 3 4 6
```

Figura 50: Recorrido DFS en el segundo grafo creado a partir de 3 vértices distintos.

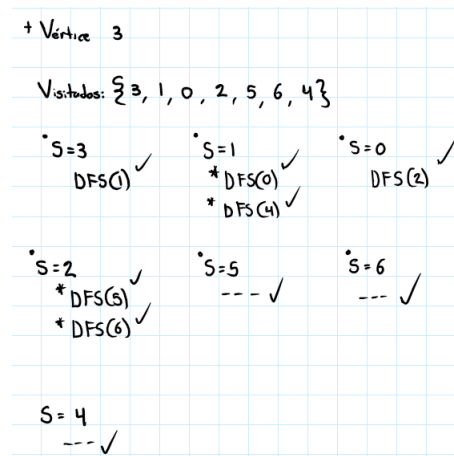


Figura 51: Recorrido DFS manual en el segundo grafo a partir del vértice 3.

El último grafo creado corresponde al presentado en [9], y se ha recorrido con **DFS** a partir de los vértices 0, 3 y 4:

```

Recorrido desde el vertice 0: 0 1 2 3 4
Recorrido desde el vertice 3: 3 1 0 4 2
Recorrido desde el vertice 4: 4 0 1 2 3

```

Figura 52: Recorrido DFS en el tercer grafo creado a partir de 3 vértices distintos.

+ Vértice 4

Visitados: { 4, 0, 1, 2, 3 }

S = 4  
 \* DFS(0) ✓  
 DFS(1) ✓  
 DFS(3) ✓

S = 0  
 \* DFS(1) ✓

S = 1  
 \* DFS(2) ✓  
 DFS(3) ✓

S = 2  
 \* DFS(3) ✓

S = 3  
 --- ✓

Figura 53: Recorrido DFS manual en el tercer grafo a partir del vértice 4.

## 6. Conclusiones

El trabajo desarrollado me ha permitido desarrollar a un mayor grado mis habilidades de abstracción y programación en Java. Una de las partes más complicadas a desarrollar estuvo en la representación matricial de los grafos; ya que requirió un análisis detallado del problema y, una vez programado, requirió diversas pruebas para verificar su adecuado funcionamiento. Con respecto al resto de representaciones, no se presentaron mayores complicaciones y se logró llegar a una serie de implementaciones correctas y que cumplen con las especificaciones solicitadas.

El algoritmo de búsqueda por expansión **BFS** me resultó complicado de entender en un inicio, ya que desconocía a detalle el funcionamiento de la clase *Iterator*; lo cual me obligó a consultar la documentación específica de la clase. A pesar de estos problemas iniciales, y revisando las notas teóricas del curso, se logró realizar un análisis adecuado de la implementación y verificar su funcionamiento.

Por último, y a raíz de la experiencia obtenida, el análisis de **DFS** resultó ser mucho más sencillo. A pesar de que este algoritmo en específico no fue abordado en la clase teórica debido a la cuarentena, no fue complicando comprender su funcionamiento. Como resultado, fue verificable con facilidad la validez de la implementación proporcionada para la *Práctica 7*.

La experiencia obtenida con estas 2 prácticas desarrolladas sin duda me ha permitido comprender



de mejor forma las diversas formas de representar un grafo, así como las formas de recorrerlos mediante los algoritmos por expansión. Con el adecuado funcionamiento de las representaciones implementadas y el punto expresado anteriormente, es posible afirmar que los objetivos propuestos para ambas prácticas han sido cumplidos en su totalidad.

Por desgracia, la situación actual no permite que las prácticas se desarrollen de la forma en la que venían realizándose, sin embargo, espero que las próximas prácticas a desarrollar en el curso puedan seguir complementando el contenido teórico del curso y se logre concluir exitosamente el temario.

## Referencias

- [1] Los puentes de KÖNIGSBERG: El comienzo de la Teoría de Grafos. Recuperado de: <https://www.gaussianos.com/los-puentes-de-konigsberg-el-comienzo-de-la-teoria-de-grafos/>. Fecha de consulta: 02/04/2020.
- [2] Teoría de Grafos. Recuperado de: <https://www.grapheverywhere.com/teoria-de-grafos/>. Fecha de consulta: 02/04/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©