	<b>Carátula para entrega de prácticas</b>	
Facultad de Ingeniería	Laboratorio de docencia	

# Laboratorios de computación salas A y B

---

*Profesor: M.I. EDGAR TISTA GARCÍA*

*Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS 2*

*Grupo: 8*

*No de Práctica(s): 6 Y 7] ALGORITMOS DE GRAFOS*

*Integrante(s): ADOLFO ROMÁN JIMENEZ*

*No. de Equipo de cómputo empleado: TRABAJO EN CASA*

*No. de Lista o Brigada:*

*Semestre: 2022 - 1*

*Fecha de entrega: 3 DE NOVIEMBRE DE 2021*

**CALIFICACIÓN:** \_\_\_\_\_

# Practica 6 y 7

## Algoritmos de Grafos

Adolfo Roman Jimenez

November 3, 2021

### **1 Objetivo General**

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

### **2 Objetivo de Clase**

El alumno será capaz de implementar y comprender los grafos, así como los recorridos.

## 3 Actividades

### 3.1 Ejercicio 1.1

#### 3.1.1 Desarrollo

La clase *Graph* primeramente comienza importando a *LinkedList* para poder usarla dentro del programa.

Ya dentro de la clase, el programa primero declara una variable entera **V** y una referencia a una lista de listas de enteros llamada **adjArray**.

Para el constructor de la clase llamado *Graph()* este constructor usa como parametro una variable entera **v** y ya dentro del constructor, primeramente se iguala la variable **v** a la variable **V** y se crea el objeto de listas ligadas que se asocia a la referencia en **adjArray**, despues de esto, por medio de un *for-loop* que itera hasta **v** se insertan dentro de cada uno de los indices en **adjArray** un numero **v** de listas ligadas que contendran a numeros enteros.

Despues de esto, la clase contiene tambien al metodo *addEdge()* el cual usa de parametros a las variables enteras **v** y **w** y lo que este metodo hace, es que toma, dentro de **adjArray**, primeramente el indice de la lista en **v** y agrega el entero **w**, despues de forma inversa, toma el indice de la lista en **w** y agrega el entero **v**.

Por ultimo el metodo *printGraph()* es un *for-loop* que itera desde 0 hasta **V** y que imprime la leyenda "**Lista de adyacencia del vertice x**" siendo **x** el nodo a imprimir.

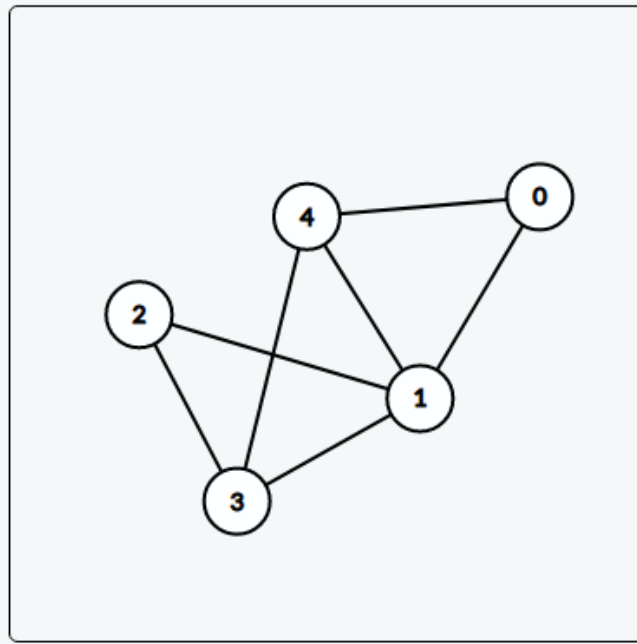
Despues de esto el programa vuelve a imprimir el numero del nodo y con un *enhanced loop* que imprime los enteros dentro del nodo **v** imprime en pantalla cada uno de los elementos que el nodo contenga.

Para la clase *Main* en *main()* primeramente se declara una variable entera **V** que se iguala a 5 y acto seguido se crea una variable de la clase *Graph* llamada **graph** cuyo constructor contiene como parametro el valor en **V**.

Acto seguido, a traves del metodo *graph.addEdge()* la clase *Main* agrega cada uno de los elementos en cada nodo generado, desde 0 a 4.

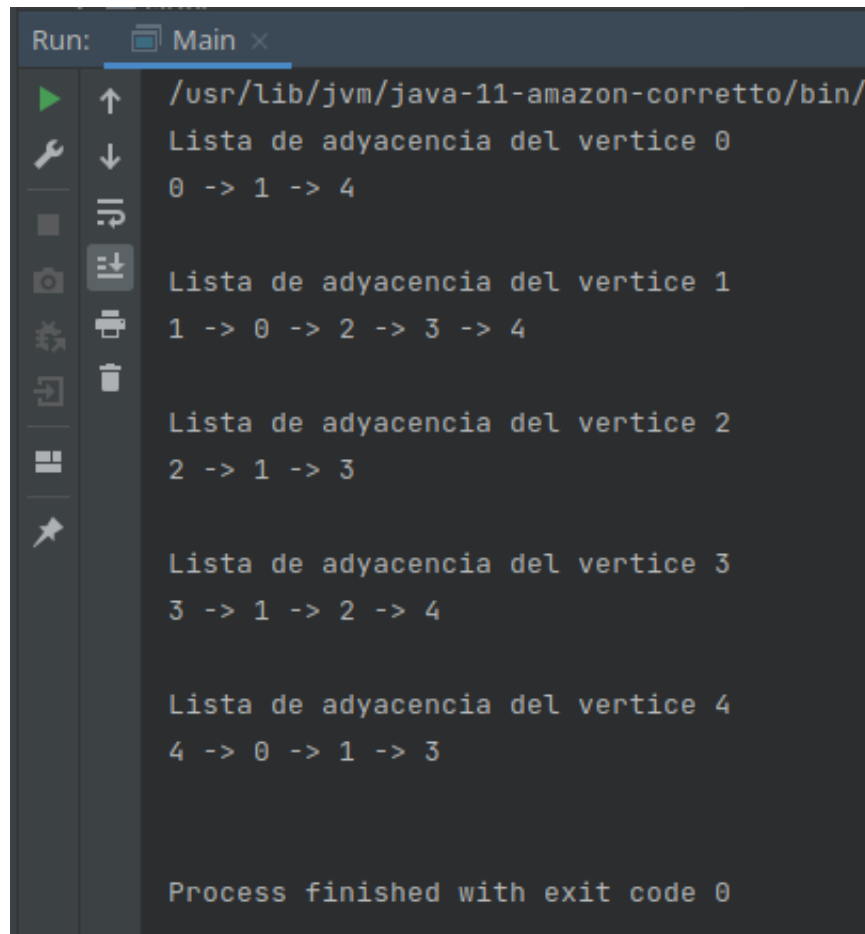
Por ultimo se imprime la lista del grafo por medio del metodo *graph.printGraph()*

La grafica resultante tendria que ser la siguiente:



Ejercicio 1.1: Grafica resultante

### 3.1.2 Ejecucion



```
Run: Main x
/usr/lib/jvm/java-11-amazon-corretto/bin/
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

Process finished with exit code 0
```

Ejercicio 1: Ejecucion de Ejercicio 1

## 3.2 Ejercicio 1.2

### 3.2.1 Desarrollo

Para este ejercicio se creo un menu en la clase principal *Main* y 3 submenus para cada una de las opciones. Esto fue debido a que las diferentes opciones del menu principal necesitan usar metodos especificos que no se pueden aglutinar en un solo menu de opciones.

El menu principal contenido en el metodo *menu1()* contiene las siguientes opciones:

1. Grafo No Dirigido
2. Grafo Dirigido
3. Grafo Ponderado No Dirigido
4. Grafo Ponderado Dirigido
5. Salir

Para la opcion 1 que es la de *Grafo No Dirigido* a traves de un control *Switch()* accede al metodo *option1()*.

Inmediatamente despues de que el usuario accede a este metodo, el mismo crea una instancia de la clase *Graph* la cual lleva el nombre de **uGraph**. Al momento de crear la instancia, el constructor de la clase *Graph* le pregunta al usuario, cuantos nodos desea crear para el grafo. El usuario ingresa el numero de nodos que se depositan en la variable privada **V** de la clase *Graph* y acto seguido el constructor asigna una *LinkedList* a la variable **nodes** la cual es una lista ligada de instancias de la clase *Node*.

Despues de esto, por medio de un *for-loop* se itera hasta el numero de nodos que el usuario desea crear, creando instancias de la clase *Node* que dentro de su estructura incluyen el numero de nodo creado dependiendo de el iterador en ese momento y por medio del metodo *add()* los inserta a la lista **nodes**.

Despues de esto, a traves de una variable entera **option1** se asigna una opcion por medio del metodo *menu2*, el cual devuelve un entero dependiendo de la opcion que el usuario escoja.

El metodo *menu2()* contiene las siguientes opciones:

1. Agregar Conexiones
2. Imprimir Lista de Adyacencia
3. Imprimir BFS

#### 4. Imprimir DFS

#### 5. Salir

El metodo *option1()* de igual forma, usa una funcion *switch()* para llevar a cabo la opcion que el usuario haya escogido dentro del submenu. Para la opcion numero 1 que es la de *Agregar conexiones* este *switch()* invoca al metodo *setEdges()* y le asigna dos parametros que son dos variables *string*, "**u**" y "**r**" las cuales sirven para indicar que las conexiones a ingresar corresponden a un grafo no dirigido "**u**" por *undirected* y que es un grafo no ponderado, "**r**" para *regular*.

Dentro del metodo *setEdges()* primero evaluamos que el grafo tenga mas de 1 nodo, de no ser asi, entonces termina el metodo puesto que no existen conexiones para realizar. Si esto es falso, entonces el metodo procede a declarar las variables enteras **v**, **w**, **edges**, **e**, **maxedges** y la variable *boolean* **flag**.

Acto seguido se procede a un *switch()* para determinar, dependiendo del tipo de grafo a crear, el numero maximo de conexiones entre nodos que el usuario puede crear. Para el caso de un grafo no dirigido se iguala entonces la variable **maxedges** al resultado de la ecuacion  $V * (V - 1) / 2$  donde **V** es el numero de nodos que el grafo contiene.

Una vez terminado esto, se le solicita al usuario el numero total de conexiones que desea crear y dentro de un *while-loop* se verifica que el usuario ingrese un numero que no sea mayor al total maximo de conexiones posibles para el nodo, valor que esta contenido en **maxedges**.

Una vez verificado esto, el programa permite al usuario ingresar las conexiones entre nodos que desee agregar, el metodo verifica que el usuario agregue conexiones entre nodos que existan y que no agregue un nodo que sea mayor a la cantidad de nodos existentes.

Este metodo de igual forma verifica que el nodo no contenga ya la conexion que el usuario desea insertar, si esta conexion ya existe dentro del nodo, el programa le pide al usuario ingresar un valor diferente y le menciona que esa conexion ya se encuentra en el nodo.

Por ultimo el metodo verifica tambien que el nodo en el que desea ingresar el usuario la conexion no este ya lleno, esto quiere decir que ya se encuentre totalmente conectado a todos los nodos posibles dentro de su campo, si esto sucede, el programa le menciona al usuario que el nodo ya esta lleno.

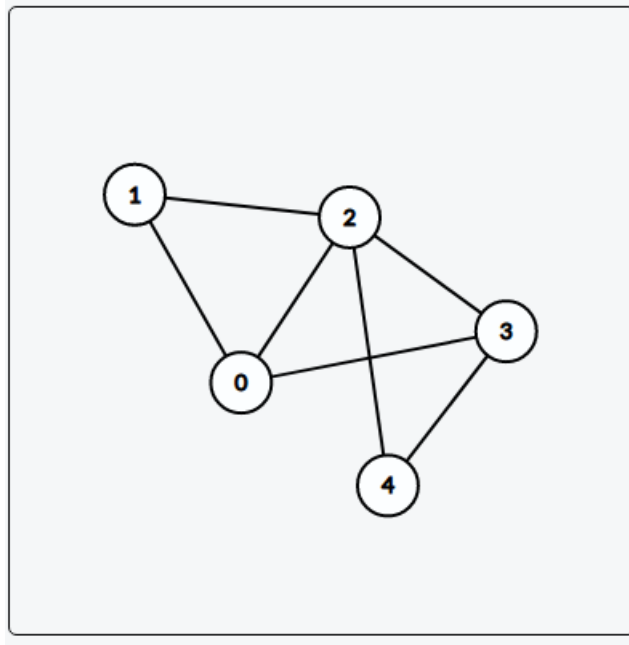
Si no ha habido ningun problema en este aspecto, entonces se procede a un nuevo *switch()* que verifica cual es el tipo de grafo con el que se esta trabajando. Para este caso como tenemos un grafo no dirigido y no ponderado, el *switch()* invoca al metodo *uGraphNodeAdd()*.

Este metodo contiene los parametros enteros **v** y **w** que incluyen los nodos que el usuario pretende conectar. Lo que hace el metodo es que simplemente invoca dos veces al metodo *dGraphNodeAdd()* con los parametros invertidos. El metodo *dGraphNodeAdd()* obtiene de la lista **nodes.next** el indice **v** y dentro de este agrega la referencia al nodo **w** a su lista ligada. De igual manera lo hace pero ahora de forma invertida despues cuando ahora obtiene el nodo que esta en la lista **nodes.next** en el indice **w** y agrega la referencia al nodo **v** dentro de esa lista.

De esta forma ambos nodos quedan interconectados pues tanto uno como otro contienen su propia informacion.

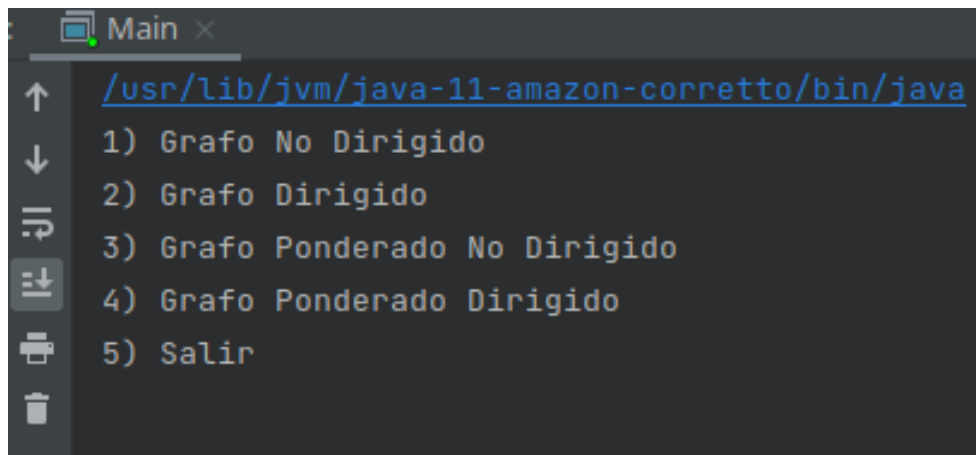
Al finalizar, el usuario puede escoger la opcion 2 *Imprimir Lista de Adyacencia* para corroborar el funcionamiento del programa.

### 3.2.2 Ejecucion



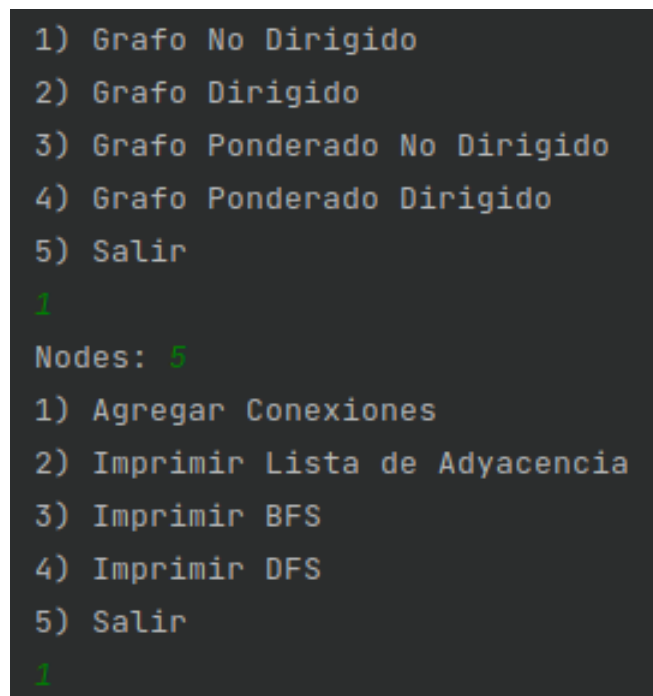
Ejercicio 1.2: Grafo a representar





```
Main x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
```

Ejercicio 1.2: Menu Principal



```
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
1
Nodes: 5
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
1
```

Ejercicio 1.2: submenu2, se agregan nodos

```
No. de conexiones: 7
Node -> Node
1. 0 1
2. 0 2
3. 0 3
4. 3 4
5. 4 2
6. 3 2
7. 1 2
```

Ejercicio 1.2: Opcion 1, se agregan conexiones

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
2
```

Ejercicio 1.2: Se imprimen la lista de adyacencia

```
Lista de adyacencia del vertice 0
0 -> 1 -> 2 -> 3

Lista de adyacencia del vertice 1
1 -> 0 -> 2

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

Lista de adyacencia del vertice 3
3 -> 0 -> 4 -> 2

Lista de adyacencia del vertice 4
4 -> 3 -> 2
```

Ejercicio 1.2: Se visualiza la lista

## 3.3 Ejercicio 2

### 3.3.1 Desarrollo

Para este ejercicio como ya se ha descrito anteriormente, se modifico el metodo *setEdges()* para que en este caso generara grafos dirigidos.

Se sigue la misma logica del ejercicio anterior, pero en este caso ahora, en el menu principal, el usuario debe seleccionar la opcion 2 que corresponde a *Grafo Dirigido*.

Al seleccionar esta opcion, el usuario invoca al metodo *option2()* cuyo menu contiene exactamente las mismas opciones que el ejercicio anterior, de igual manera se crea un grafo al tiempo que el usuario ingresa al metodo *option2()* pero en este caso, el grafo que el programa crea lleva el nombre de **dGraph**.

La diferencia principal es que al momento de escoger en el *menu2()* la opcion de *Crear Conexiones*, ahora el *switch()* de *option2()* invocara al metodo *setEdges()* pero con los parametros "**d**" y "**r**" que corresponden a "*directed*" y "*regular*".

Ya dentro de el metodo *setEdges()* el funcionamiento es similar, pero al momento de otorgar un valor a la variable **maxedges** el programa evalua  $V*(V-1)$  unicamente, puesto que son el total de conexiones que puede obtener un grafo no dirigido.

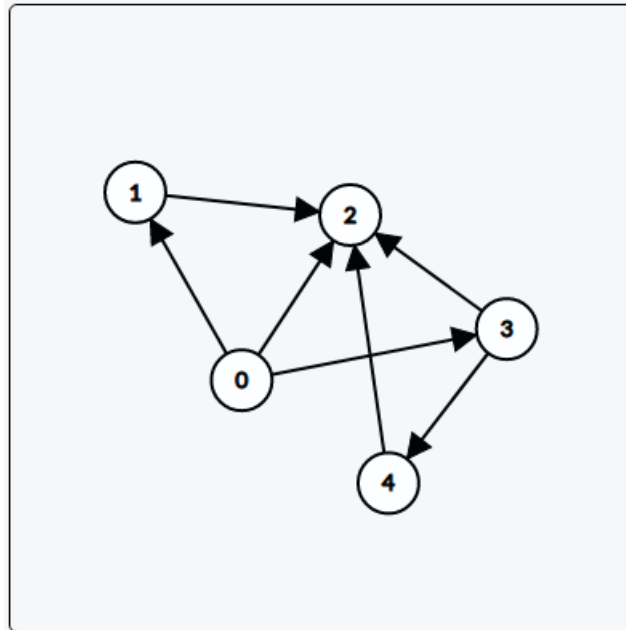
Ya despues de solicitar los nodos a conectar y haberlos ingresado, el programa dentro del *switch()* de *setEdges()* de acuerdo con los parametros de la funcion, invoca al metodo *dGraphAddNodes()* que toma a los parametros **v** y **w** que contienen los nodos a conectar de (Nodo **v** a Nodo **w**) y toma en la lista **Node.next** el indice de **v** y le agrega la referencia a su lista ligada interna del nodo **w**.

Esto se ejecuta hasta que se ha terminado de iterar en las conexiones que el usuario desea crear.

Para la ejecucion vamos a tomar el mismo grafo anterior, pero ahora lo haremos como grafo dirigido.

Al finalizar la ejecucion, podremos apreciar como las listas de adyacencia tanto del primer ejercicio como del segundo, son totalmente distintas. De hecho se aprecia al finalizar la ejecucion en este ejercicio, como donde en el ejercicio 1.2 todos los grafos contienen conexiones, en el ejercicio 2 existen grafos que carecen de adyacencias.

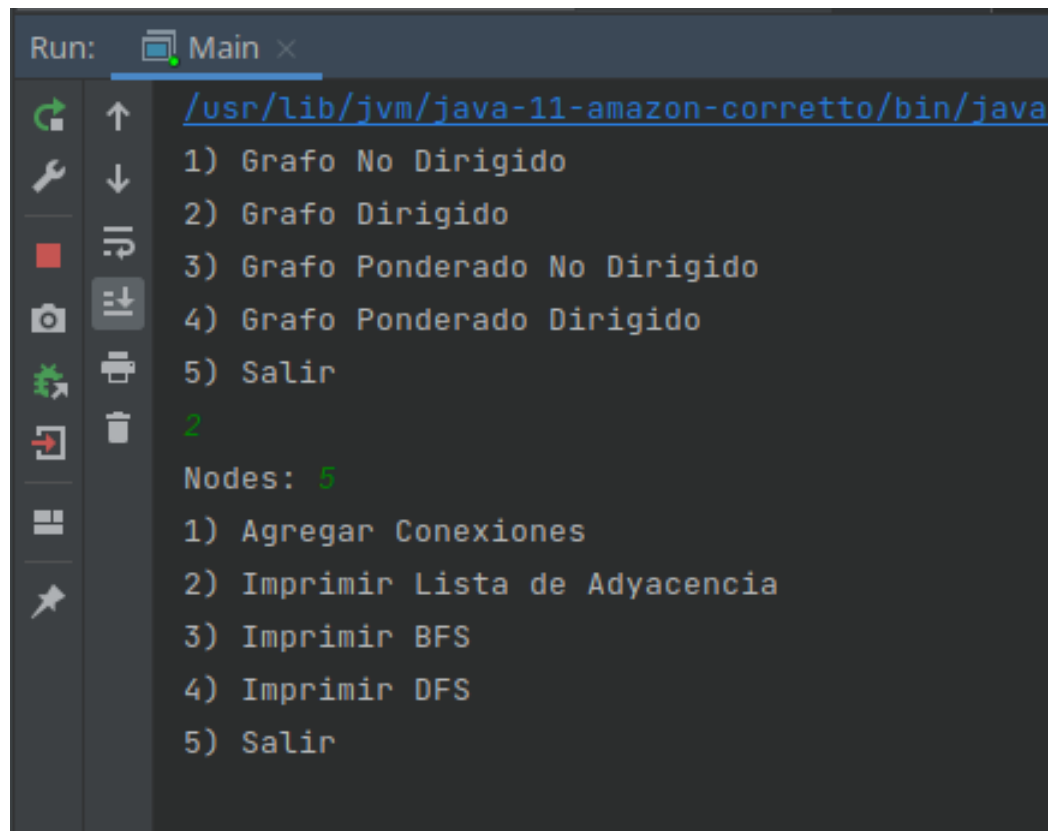
### 3.3.2 Ejecucion



Ejercicio 2: Grafo a representar

```
h: Main x
  /usr/lib/jvm/java-11-amazon-corretto/bin/java
  1) Grafo No Dirigido
  2) Grafo Dirigido
  3) Grafo Ponderado No Dirigido
  4) Grafo Ponderado Dirigido
  5) Salir
  |
```

Ejercicio 2: Menu Principal



The screenshot shows a Java IDE's console window with the title "Run: Main x". The command `/usr/lib/jvm/java-11-amazon-corretto/bin/java` is at the top. The program's output is as follows:

```
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
2
Nodes: 5
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
```

Ejercicio 2: submenu2, se agregan nodos

```
1
No. de conexiones: 7
Node -> Node
1. 0 1
2. 0 2
3. 0 3
4. 3 4
5. 4 2
6. 3 2
7. 1 2
```

Ejercicio 2: Opcion 1, se agregan conexiones

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
2
```

Ejercicio 2: Se imprime la lista de adyacencia

```
Lista de adyacencia del vertice 0  
0 -> 1 -> 2 -> 3
```

```
Lista de adyacencia del vertice 1  
1 -> 2
```

```
Lista de adyacencia del vertice 2  
2 -> /
```

```
Lista de adyacencia del vertice 3  
3 -> 4 -> 2
```

```
Lista de adyacencia del vertice 4  
4 -> 2
```

Ejercicio 1.2: Se visualiza la lista



## 3.4 Ejercicio 3

### 3.4.1 Desarrollo

Para este ejercicio una de las implementaciones que hice fue dentro de la clase *Node* fue modificar su estructura a la siguiente:

- *private LinkedList<Node> next*
- *private LinkedList<Integer> edges*
- *private int num*
- *private int N*
- *private int E*

La lista **next** contiene las **referencias** a otros nodos a los que hace adyacencia.

La lista **edges** contiene el valor de los aristas a cada uno de esos nodos.

La variable **num** contiene el numero que le corresponde a ese nodo.

La variable **N** contiene la cantidad de nodos a los que hace adyacencia el nodo.

La variable **E** contiene el mismo numero de nodos pero su distancia o costo en caso de que el grafo sea ponderado.

Una de las cosas que hice, es que sobrecargue el metodo *addNode()* dentro de la clase *Node* para los grafos ponderados, de tal manera que al momento de crear un nodo, en vez de requerir unicamente el nodo a agregar dentro de la lista **next**, el metodo tambien requiere el valor del arista que se agrega junto con ese nodo y que inserta dentro de la lista **edges** al momento de ingresar el nodo.

De igual manera para el metodo *setEdges()* cuando se trata de grafos ponderados, en su segundo parametro utilizo la palabra "**p**" que se refiere a ponderado y al momento de hacer esto, el programa contiene instrucciones, para que en caso de que en el segundo parametro se encuentre esta palabra, entonces en vez de solicitar 2 variables que serian las de los grafos **v** y **w** tambien solicite la variable **e** que corresponde al valor de la arista.

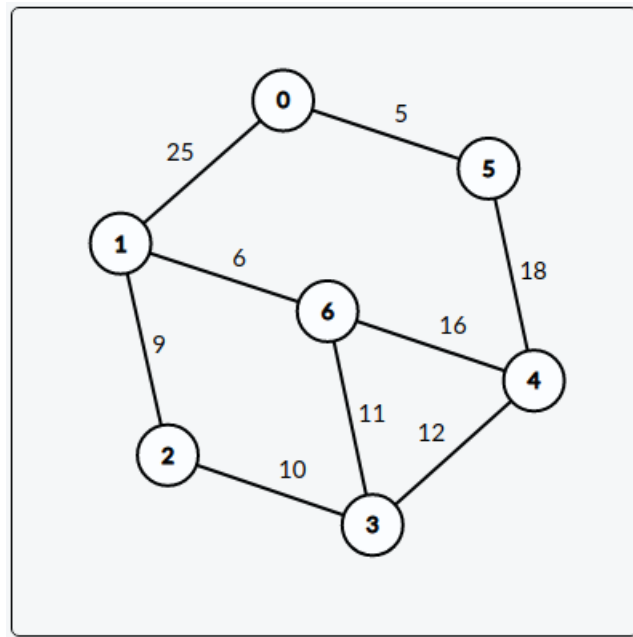
El programa tambien imprime antes de solicitar esta informacion la leyenda *Node -> Distancia -> Node* que explica de que manera se debe de ingresar la informacion cuando se trata de un grafo ponderado, osea, debemos primero de ingresar la informacion del nodo a conectar, despues de la distancia o costo que

se encuentra entre ambos nodos y al finalizar el nodo objetivo.

Cuando el metodo *setEdges()* identifica que el programa contiene una "p" como segundo argumento, entonces utiliza de igual forma, una de las versiones sobrecargadas de los metodos (*uGraphNodeAdd()*) o *dGraphNodeAdd()* las cuales contienen 3 parametros en vez de 2 que por su puesto, incluyen a la distancia entre los nodos.

### 3.4.2 Ejecucion

#### Grafo Ponderado No Dirigido



Ejercicio 3: Grafo a representar

```
Main x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
3
```

Ejercicio 2: Menu Principal

```
Main x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
3
Nodes: 7
```

Ejercicio 3: submenu3, se agregan nodos

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Primm's Algorithm
6) Salir
1
No. de conexiones: 9
```

Ejercicio 3: Opcion 1, se agregan conexiones

```
Node -> Distance -> Node
1. 0 5 5
2. 5 18 4
3. 4 16 6
4. 4 12 3
5. 3 11 6
6. 3 10 2
7. 2 9 1
8. 6 6 1
9. 1 25 0
```

Ejercicio 3: Se incluyen los nodos y entre ambos la distancia

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Primm's Algorithm
6) Salir
2
```

Ejercicio 3: Opcion 2 para visualizar lista de adyacencia

```
Lista de adyacencia del vertice 0  
0 -(5)-> 5 -(25)-> 1
```

```
Lista de adyacencia del vertice 1  
1 -(9)-> 2 -(6)-> 6 -(25)-> 0
```

```
Lista de adyacencia del vertice 2  
2 -(10)-> 3 -(9)-> 1
```

```
Lista de adyacencia del vertice 3  
3 -(12)-> 4 -(11)-> 6 -(10)-> 2
```

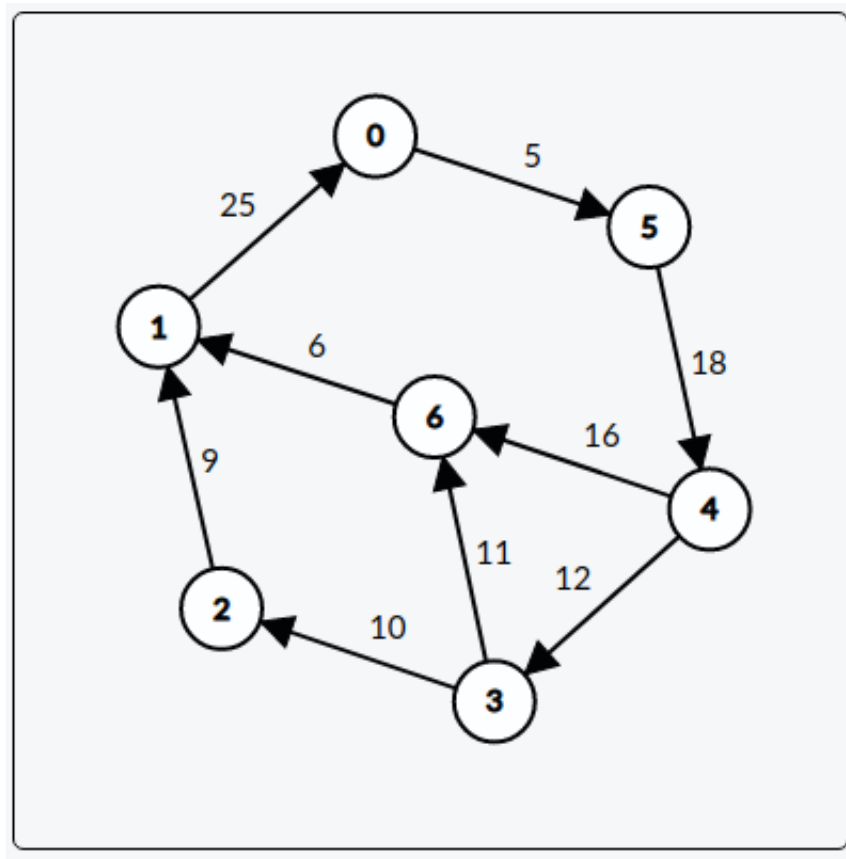
```
Lista de adyacencia del vertice 4  
4 -(18)-> 5 -(16)-> 6 -(12)-> 3
```

```
Lista de adyacencia del vertice 5  
5 -(5)-> 0 -(18)-> 4
```

```
Lista de adyacencia del vertice 6  
6 -(16)-> 4 -(11)-> 3 -(6)-> 1
```

Ejercicio 3: Se visualiza la lista con distancia a cada Nodo

### Grafo Ponderado Dirigido



Ejercicio 3: Grafo ponderado dirigido a representar

```
Main x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
4
Nodes: 7
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Primm's Algorithm
6) Salir
1
No. de conexiones: 9
```

Ejercicio 2: Menu principal y se agregan nodos, menu3

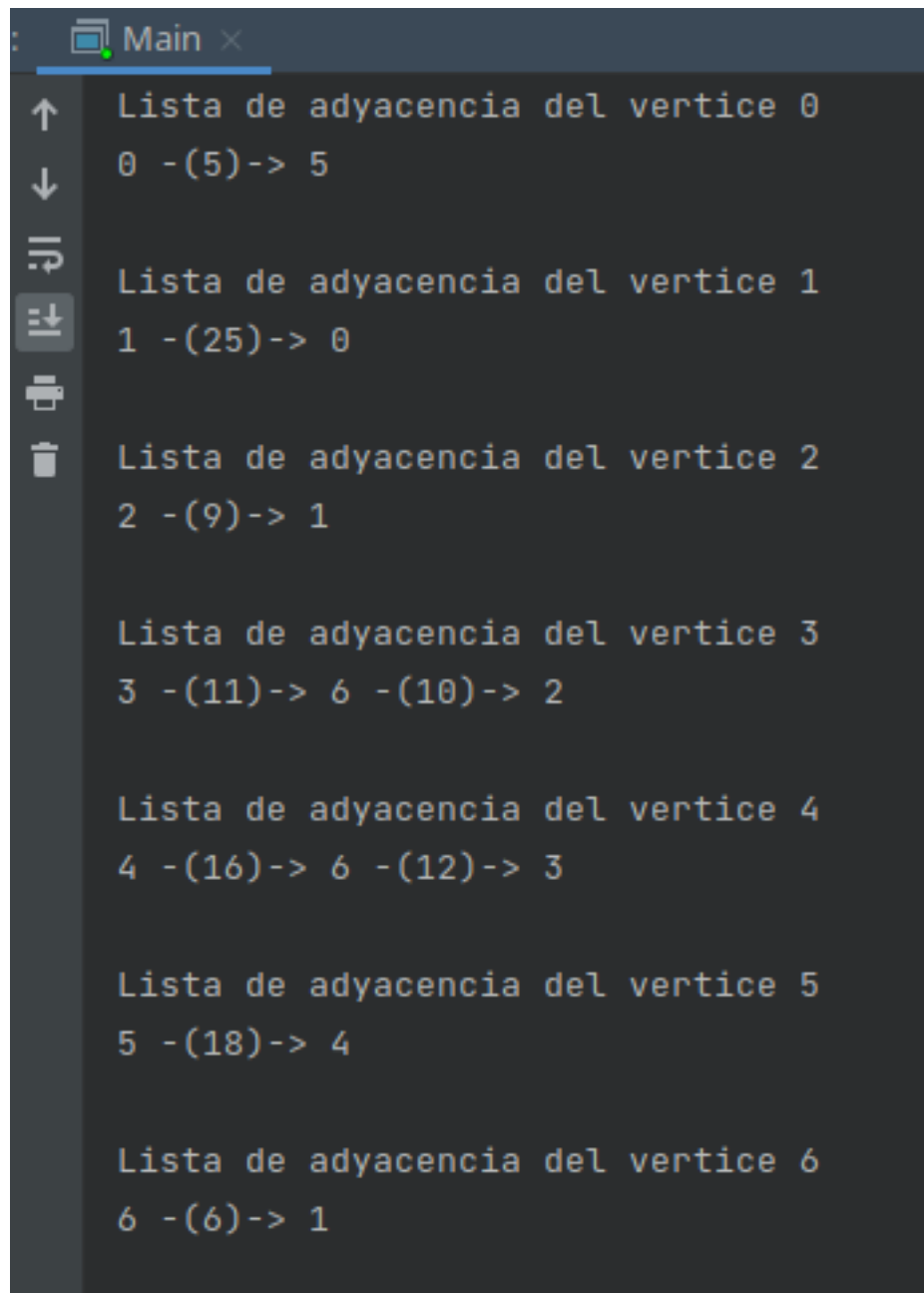


```
Node -> Distance -> Node
1. 0 5 5
2. 5 18 4
3. 4 16 6
4. 4 12 3
5. 3 11 6
6. 3 10 2
7. 2 9 1
8. 6 6 1
9. 1 25 0
```

Ejercicio 3: Se agregan conexiones con distancia

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Primm's Algorithm
6) Salir
2
```

Ejercicio 3: Opcion 2 para imprimir lista de adyacencia



The image shows a terminal window titled "Main" with a dark background and light-colored text. On the left side of the terminal, there is a vertical toolbar with icons for navigation (up, down), search, and other standard terminal functions. The main area of the terminal displays the following text:

```
Lista de adyacencia del vertice 0  
0 -(5)-> 5  
  
Lista de adyacencia del vertice 1  
1 -(25)-> 0  
  
Lista de adyacencia del vertice 2  
2 -(9)-> 1  
  
Lista de adyacencia del vertice 3  
3 -(11)-> 6 -(10)-> 2  
  
Lista de adyacencia del vertice 4  
4 -(16)-> 6 -(12)-> 3  
  
Lista de adyacencia del vertice 5  
5 -(18)-> 4  
  
Lista de adyacencia del vertice 6  
6 -(6)-> 1
```

Ejercicio 3: Lista de adyacencia para grafo ponderado no dirigido

## 3.5 Ejercicio 4

### 3.5.1 Desarrollo

Para el desarrollo de este ejercicio tuve que modificar el algoritmo mostrado en la practica para que fuera compatible con la estructura de datos que cree tanto para mi clase *Nodo* como para mi clase *Graph*.

Una de las cosas que agregue, fue codigo en el metodo *BFS()* para que este desde el interior del metodo, pregunte al usuario cual es el nodo por donde desea iniciar el recorrido del grafo por medio de un *do-while loop* que verifica que el usuario ingrese un grafo existente para poder recorrerlo. De esta manera el metodo ya no usa ningun parametro y se vuelve mas dinamico.

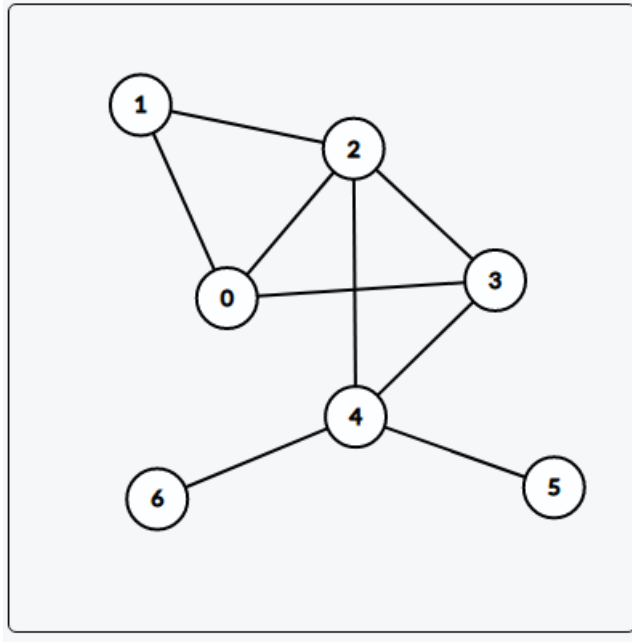
Despues, debido a la estructura de datos que implemente, removi el iterador porque tuve algunos problemas al momento de implementarlo y mejor lo sustitui por un *enhanced for-loop* que itera dentro de la lista `nodes.get(s).getNext()`, *getNext()* es el "getter" de la lista ligada de Nodos que el nodo contiene (sus conexiones) asi, ee codigo, devuelve una lista de nodos contenidos en el nodo con indice `s` de la lista de la clase *Graph*.

El funcionamiento de *BFS()* me parecio muy sencillo, despues de escoger a partir de que nodo quieres comenzar a recorrer el la grafica. Se crea un arreglo de variables tipo *boolean* del tamaño de la cantidad de nodos contenidos en el grafo. Despues de esto se crea una lista ligada que contiene variables de tipo entero **queue** y se agrega dentro del indice `s` que previamente escogio el usuario, de el arreglo **visited** se iguala a *true* indicando que ese nodo ya ha sido visitado.

Por medio del metodo *add()* se ingresa el valor numerico del nodo a la lista **queue** y se ingresa a un *while-loop* que no termina sino hasta que el tamaño de la lista **queue** sea igual 0.

Dentro de este *while-loop* se remueve el primer indice de **queue** por medio del metodo *poll()* y se agrega este valor, nuevamente a la variable `s`. Despues de esto, como ya se habia comentado, se itera con un *enhanced for-loop* a la lista de nodos contenidos dentro del nodo `s` y se obtiene el numero de cada nodo por medio del metodo *getter* `node.getNum()`, despues de esto se verifica si este nodo se encuentra dentro del arreglo **visited** en el indice que le corresponde, si no es asi, entonces se iguala ese indice a *true* y se agrega a la **queue** por medio del metodo *add()*. El ciclo termina de iterar cuando ya todos los espacios en **visited** son iguales a *true* y consecuentemente no quedan nodos dentro de **queue**.

### 3.5.2 Ejecucion



Ejercicio 4: Grafo a representar

```
Main x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
1
Nodes: 7
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
1
No. de conexiones: 9
```

Ejercicio 4: Menu Principal agregamos nodos y conexiones

```
Node -> Node
1. 0 1
2. 0 2
3. 0 3
4. 1 2
5. 2 3
6. 4 3
7. 4 2
8. 4 5
9. 4 6
```

Ejercicio 4: Se agregan conexiones

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
3
BFS nodo inicio: 0
BFS: 0 1 2 3 4 5 6
```

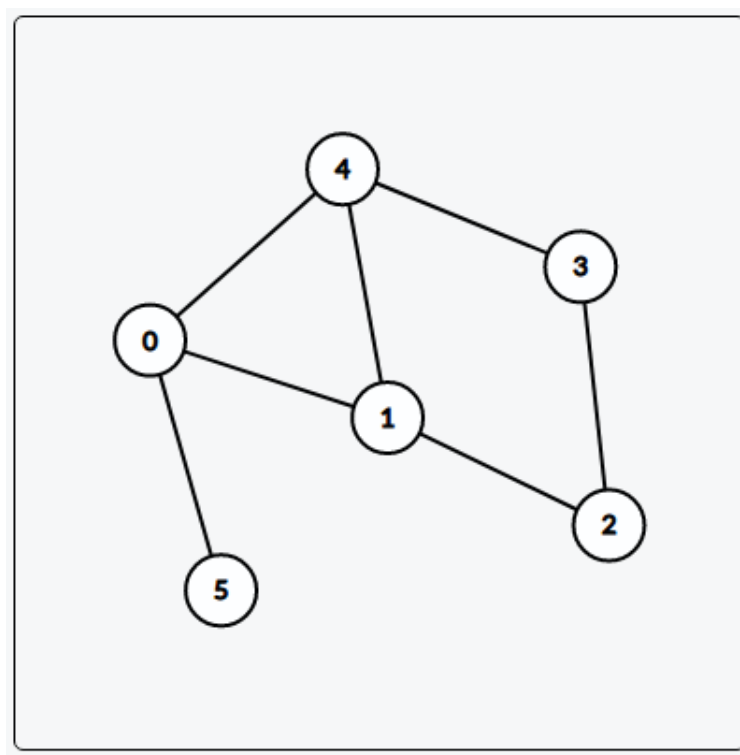
Ejercicio 4: BFS a partir de nodo 0

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
3
BFS nodo inicio: 6
BFS: 6 4 3 2 5 0 1
```

Ejercicio 4: BFS a partir de nodo 6

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
3
BFS nodo inicio: 3
BFS: 3 0 2 4 1 5 6
```

Ejercicio 4: BFS a partir de nodo 3



Ejercicio 4: Grafo no 2

```
1) Grafo No Dirigido
2) Grafo Dirigido
3) Grafo Ponderado No Dirigido
4) Grafo Ponderado Dirigido
5) Salir
1
Nodes: 6
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
1
No. de conexiones: 7
```

Ejercicio 4: Menu Principal y agregamos nodos y conexiones

```
Node -> Node
1. 0 1
2. 0 5
3. 0 4
4. 4 1
5. 1 2
6. 2 3
7. 4 3
```

Ejercicio 4: Se asocian nodos



```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
3
BFS nodo inicio: 0
BFS: 0 1 5 4 2 3
```

Ejercicio 4: BFS desde nodo 0

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
3
BFS nodo inicio: 5
BFS: 5 0 1 4 2 3
```

Ejercicio 4: BFS desde nodo 5

```
1) Agregar Conexiones
2) Imprimir Lista de Adyacencia
3) Imprimir BFS
4) Imprimir DFS
5) Salir
3
BFS nodo inicio: 3
BFS: 3 2 4 1 0 5
```

Ejercicio 4: BFS desde nodo 3

## 3.6 Ejercicio 5

### 3.6.1 Desarrollo

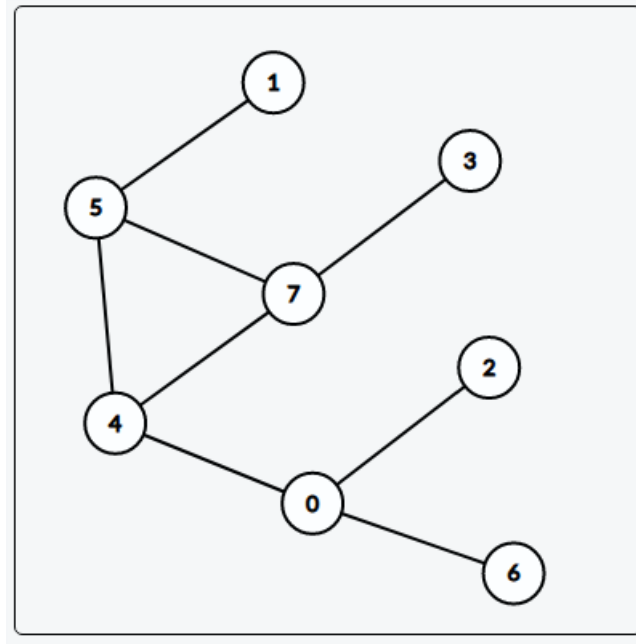
Para la implementacion de el metodo *DFS()* las modificaciones que le tuve que hacer fueron similares a las de (*BFS()*). De igual manera el usuario tiene la opcion de escoger el nodo a partir del cual quiere comenzar a recorrer el grafo de forma dinamica y dentro del cuerpo de *DFSUtil()* tambien sustitui el iterador por un *enhanced for-loop* que opera de la misma forma que en *BFS()*.

Algo que me llamo la atencion de *BFS()* es que este opera de forma recursiva dentro del metodo *DFSUtil()*.

El metodo comienza con *DFS()* que como ya lo mencionamos, pide al usuario el nodo por el cual quiere comenzar, de igual manera que *DFS()* se crea un arreglo de *boolean* llamado **visited** pero en este caso, tanto el nodo escogido por el usuario **v** como el arreglo **visited** se transfieren como parametros al metodo *DFSUtil()*.

Ya dentro de *DFSUtil()* de igual forma se agrega dentro de **visited** en el indice **v** se iguala a *true* y se comienza a iterar dentro del indice **v** de los nodos de la lista **nodes**, al igual que en el anterior, se declara una variable entera **n** que se iguala a **node.getNum()** que proporciona el numero del nodo en cuestion. Si la variable **n** como nodo no esta contenida (no esta igualada a *true*) dentro del indice **n** que le corresponde, entonces se invoca recursivamente a *DFSUtil()* pero ahora con **n** como parametro para continuar el proceso de tal forma que se va iterando dentro de cada nodo como si fuera una linea hasta llegar a lo mas profundo de este a diferencia de **BFS()** que unicamente alcanza un nivel por debajo del nodo en cuestion para revisar sus contenidos.

### 3.6.2 Ejecucion



Ejercicio 5: Grafo a ingresar

```
Run: Main x Ejercicio5 x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
DFS nodo inicio: 0
DFS: 0 4 5 1 7 3 2 6
DFS nodo inicio: 7
DFS: 7 4 5 1 0 2 6 3
DFS nodo inicio: 4
DFS: 4 5 1 7 3 0 2 6

Process finished with exit code 0
```

Ejercicio 5: Ejecucion DFS desde 3 nodos

## 3.7 Ejercicio 6: Adicional

### 3.7.1 Desarrollo

Aunque intente hacerlo por mi cuenta y sin verificar informacion de internet, no pude completar el algoritmo de Primm, pero seguire trabajando en el ya que dudo que haya estado lejos de comenzar a conseguir algo con el.

El codigo se encuentra como metodo de la clase *Graph*, el metodo se llama *primm()* y se encuentra en la linea 176 de la clase.

## 4 Conclusiones

Esta practica me gusto mucho porque me puso a pensar muchisimo sobre los grafos y como se itera sobre ellos.

Ya tenia yo una vaga idea sobre como debia de implementar los grafos en codigo debido a que alguna vez me puse a pensar en ello y decidi llevar manos a la obra durante la practica y ponerme a experimentar muchisimo ya que algo que me agrada es poder construir estructuras de datos a partir de nada y sin ayuda de muchos metodos predefinidos.

Fue por esta razon que durante la practica me di a la tarea de implementar la idea que yo tenia de los elementos que un Nodo tenia que contener y por esta razon le implemente una lista ligada de otros nodos a los cuales pudiera tener referencia.

Algo en lo que tambien me base fue en la definicion de grafos la cual menciona que es un conjunto  $S = \{G, E\}$  donde **G** es el conjunto de grafos y **E** el conjunto de aristas. Fue en esto que me base para la clase *Graph* que es precisamente lo que ahi representa **S** como el conjunto donde todos los grafos pertenecientes a un conjunto habitan.

Lo que si me di cuenta de la practica es que ya no voy a estar experimentando tanto, porque hice bastantes metodos porque me gusta mi codigo genial, estructurado y eficiente. Pero me tardo mucho en hacerlo y luego ya no me da tiempo de acabar la practica por escrito y ando a las prisas. Una lastima porque me gusta mucho programar y experimentar con mi codigo.

Espero que le guste mi practica y gracias por leerme