	Carátula para entrega de prácticas
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 II -1317

Grupo: 10

No. de práctica(s): 6-7

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 26

Semestre: 2023-1

Fecha de entrega: 30 de octubre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 6-7.

Algoritmos de grafos

OBJETIVOS

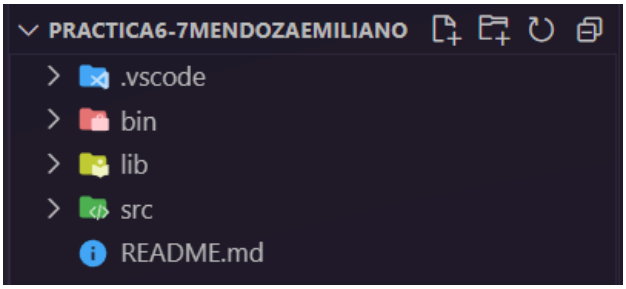
- **Objetivo general:** El conocerá las formas de representar un grafo e identificará las características necesarias para comprender el algoritmo de búsqueda por expansión y profundidad.
- **Objetivo de la clase:** El alumno será capaz de implementar y comprender los grafos, así como los recorridos.

DESARROLLO

Ejercicio 1. Implementación de Grafos en Java

Descripción de actividades realizadas

1. Se creó un nuevo proyecto llamado *Practica6-7MendozaEmiliano* usando Visual Studio Code.



2. Se creó la clase *Graph*, implementando el código proporcionado en el manual de práctica.

3. Se realizaron las siguientes modificaciones en el código de *Graph*:

- Primero, se modificó la función que imprime el grafo para que se visualicen las listas de adyacencia en una sola línea.
- También se realizó un cambio en la estructura de datos de la lista de adyacencia con el fin de evitar los warnings que se presentaron, sin embargo, esto no altera la salida del programa.

  El warning indica: `type safety: the expression of type LinkedList needs unchecked conversion to conform to LinkedList<Integer>[]`. Esto se puede solucionar usando un ArrayList en lugar de usar un arreglo de genéricos.
- A partir de este punto se agregaron modificadores de acceso a los atributos y métodos de todas las clases que conforman al proyecto.

Modificación del programa

```
// LinkedList<Integer> adjArray[];
ArrayList<LinkedList<Integer>> adjArray;
```

Nota: Al hacer uso de la clase *ArrayList*, se agregó su respectivo *import.*

```
import java.util.ArrayList;
```

Como puede observarse, se sustituye el uso de un arreglo por un *ArrayList*.

```
//adjArray = new LinkedList[v];
adjArray = new ArrayList<>();
```

Además, las listas ligadas contenidas en el *ArrayList* no son genéricas.

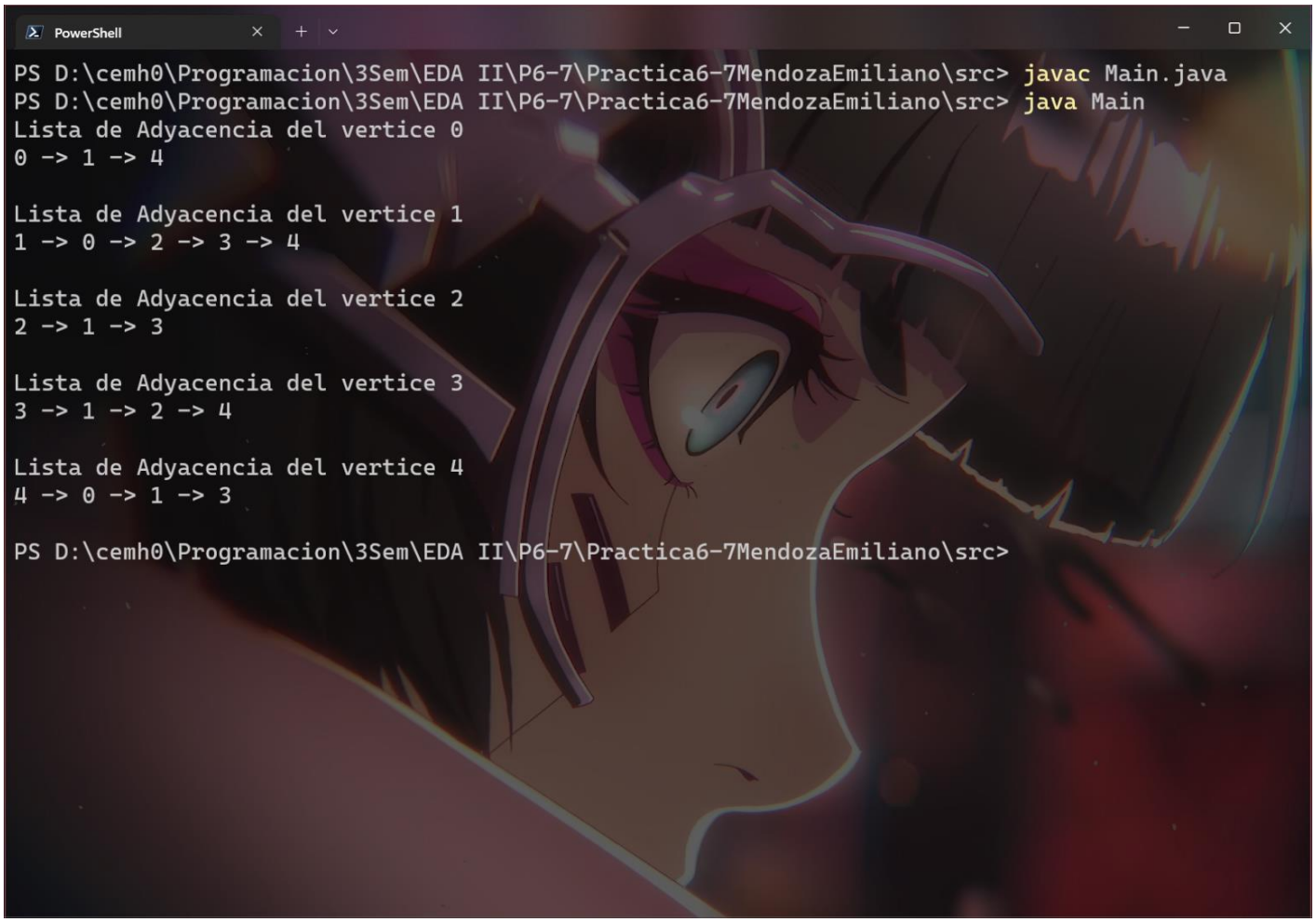
```
//adjArray[i] = new LinkedList();
adjArray.add(i, new LinkedList<Integer>());
```

Por último, se adaptaron las sentencias referentes al uso del arreglo anterior de manera que funcionen para *ArrayList*

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

4. Se creó la clase principal *Main* implementando el código proporcionado en el manual de práctica.
5. Se compiló y se ejecutó el proyecto desde la terminal.

Ejecución en *PowerShell*



***Observación*:** En esta implementación de grafos, los vértices se nombran de acuerdo con su posición en la lista de adyacencia. Por lo tanto, un grafo de 5 vértices nombra a sus vértices como 0, 1, 2, 3 y 4. Esto implica que para cada grafo de *n* vértices creado con esta implementación, sus vértices se etiquetan como 0,1,2,…,*n*.

Análisis de la implementación del código de *Graph*

Dentro de la clase *Graph* se observa que tiene dos atributos, un constructor y dos métodos:

- **Atributos:**

```
int V;
ArrayList<LinkedList<Integer>> adjArray;
```

V: es el número o cantidad de nodos en el grafo.

adjArray: es una lista de adyacencia que representa al grafo.

- **Constructor:**

```
Graph (int v) {
        V = v;
        adjArray = new ArrayList<>();
        for (int i = 0; i < v; ++i)
            adjArray.add(i, new LinkedList<Integer>());
}
```

Parámetro *v*: Es el número de nodos o vértices con el que se inicializará el grafo.

Dado como parámetro la cantidad de nodos del grafo, este constructor inicializa un *ArrayList* donde, para cada nodo del grafo, inicializa a su vez una *LinkedList* de números enteros.

- **Métodos:**

I. *addEdge*

```
public void addEdge(int v, int w) {
        adjArray.get(v).add(w);
        adjArray.get(w).add(v);
}
```

Parámetro *v*: Un nodo del grafo al cual se le quiere agregar una arista, partiendo de este (en realidad no importa de qué nodo parte en este caso porque es un grafo no dirigido).

Parámetro *w*: Un nodo del grafo al cual se le quiere agregar una arista, llegando a este.

Como puede observarse, primero se toma el nodo *v* como cabeza de lista y se agrega la adyacencia con el nodo *w*; luego se hace lo mismo, pero tomando al nodo *w* como cabeza de lista y agregando la adyacencia al nodo *v*.

II. *printGraph*

```
public void printGraph (Graph graph) {
        for (int v = 0; v < graph.V; v++) {
            System.out.println("Lista de Adyacencia del vertice " + v);
            System.out.print(v);
            for (Integer node : graph.adjArray.get(v)) {
                System.out.print(" -> " + node);
            }
            System.out.println("\n");
        }
}
```

Este método imprime la lista de adyacencia del grafo.

Ejercicio 2. Grafos dirigidos

6. Se implementó una nueva clase llamada *Digraph*. Esta clase representará grafos dirigidos y permitirá al usuario crear grafos indicando la cantidad de nodos y aristas.

***Observación*:** La implementación de un grafo dirigido es bastante similar a la de un grafo no dirigido. La única diferencia se presenta en el método *addEdge*.

```
public void addEdge(int v, int w) {
        adjArray.get(v).add(w);
        // adjArray.get(w).add(v);
}
```

Para que el grafo solo “apunte” en una dirección, basta con quitar la sentencia que añade la segunda arista (“la de regreso”).

Ejercicio 3. Grafos ponderados

7. Se creó una nueva clase llamada *WGraph*. Que servirá para representar grafos ponderados.

***Observación*:** En esta implementación de grafos por medio de la lista de adyacencia, para un grafo ponderado, se debe almacenar junto con cada arista el valor del peso que contiene. Una forma de lograr esto es usando la estructura de datos *HashMap*, en la cual las claves pueden representar el nodo adyacente y el valor del peso de la arista.

En comparación con la lista de adyacencia de las clases de grafos anteriores, que se construye con un *ArrayList* de *LinkedLists*, la lista de adyacencia de los grafos ponderados se construye con un *ArrayList* de *HashMaps* que contiene los pares (*nodo adyacente, valor de la arista*).

- **Atributos:**

```
int V;
ArrayList<LinkedHashMap<Integer, Integer>> adjArray;
```

- **Constructor:**

Su funcionamiento es bastante similar a los constructores de las dos clases anteriores, pero se hizo la adaptación para inicializar la lista de adyacencia con *HashMaps*.

- **Métodos:**

En esta clase se implementan 3 métodos: *addEdge*, *printGraph* y *prim*, cuyas funciones son agregar una arista, imprimir el grafo, y crear un árbol de expansión mínimo usando el algoritmo de Prim. Para el método de *addEdge*, se modificó la firma del método agregando el parámetro *weight*. *Weight* guarda el peso de la arista agregada. Al igual que con el método *addEdge*, se modificó el método de *printGraph* para imprimir grafos ponderados (para cada arista se imprime su valor ponderado).

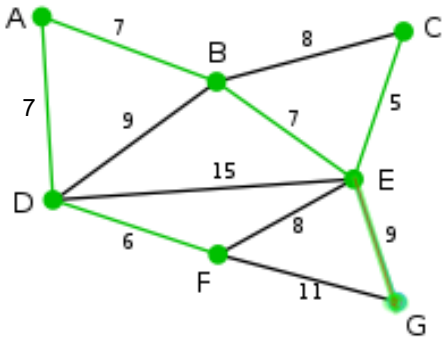
8. A continuación se detalla más el funcionamiento del método *prim().*

- El valor de retorno de este método es un objeto de tipo *Graph*, que representa un grafo no dirigido y que es el árbol de expansión mínima de esta instancia que llama al método.
- El parámetro *s* es el nodo inicial para iniciar el algoritmo.
- Se crea la variable *current* (de tipo entero) que servirá para guardar temporalmente el nodo de origen de la arista que se está revisando. Inicialmente toma el valor del nodo donde se inicia el algoritmo.
- Se crea el arreglo de booleanos *visited* (del tamaño de nodos del grafo) para marcar como visitados los nodos revisados.
- Se crea una instancia de la clase *Graph* llamada *MST*. La referencia a este objeto será el valor de retorno al final de la función, pero inicialmente, se inicializa con la misma cantidad de nodos que el grafo que invoca al método.
- Se crea también una estructura *ArrayList* de Strings llamada *pq* que será la cola de prioridad. ***Nota:*** A pesar de que Java cuenta con implementaciones de colas de prioridad, no es (para mí) tan intuitivo su funcionamiento al usar *HashMaps* como nodos en la lista de adyacencia (además de que no hay un criterio por defecto para ordenar este tipo de estructuras). Entonces, para representar la cola de prioridad de esta manera se siguió el siguiente análisis: Dada la necesidad de ordenar los nodos en una cola de prioridad donde el criterio de prioridad es el peso de las aristas (a menor peso, mayor prioridad), se pensó en un formato que condense la información de un nodo de la siguiente manera:

  Peso=nodo de origen-nodo de destino

  Para cada *String* con este formato, agregando las aristas y ordenándolas en orden ascendente, se puede simular el funcionamiento de una cola de prioridad.

Ejemplo: Se tiene el siguiente grafo al cual se le aplica el algoritmo de Primm empezando por el nodo *B*.



En teoría, encolando las aristas en una *PriorityQueue* se tendría:

$$PQ: \{(7, B-A), (7, B-E), (8, B-C), (9, B-D)\}$$

Se desencola de la *PQ* y se encolan las aristas adyacentes a *A*:

$$PQ: \{(7, B-E), (7, A-D), (8, B-C), (9, B-D)\}$$

En el caso de un *ArrayList* de *Strings* con el formato antes mencionado se tendría, después de agregar las aristas y de ordenar con un algoritmo de ordenamiento:

$$AL: \{7 = B-A, 7 = B-E, 8 = B-C, 9 = B-D\}$$

Se remueve la cabeza de *AL* y se agregan las aristas adyacentes a *A*, después de ordenarlas se tiene:

$$AL: \{7 = A-D, 7 = B-E, 8 = B-C, 9 = B-D\}$$

Al principio puede parecer que se obtienen resultados diferentes, sin embargo, el orden en que se hace el recorrido de las aristas es irrelevante dado que una vez visitados, ya no se agrega al *MST* otra arista hacia el mismo nodo. Se observa, entonces, que el funcionamiento de la cola de prioridad se puede simular usando un *ArrayList* de *Strings* (donde cada *String* contiene la información necesaria de algún nodo) ordenando la lista después de agregar los nodos adyacentes a un nodo según se recorren en el grafo.

- Dado el nodo inicial, lo siguiente que se hace es iterar sobre todas las claves de la sublista (de la lista de adyacencia del grafo) referente nodo inicial. Estas claves son todos los nodos que tengan una arista adyacente con el nodo inicial.
- Se construyen los *Strings* (Peso=origen-destino) con todas las aristas adyacentes al nodo inicial. Se agregan a *pq*.
- Después de agregar las aristas se ordena *pq* y se marca el nodo inicial (*current*) como visitado.
- Se entra al ciclo *while* que itera mientras *pq* tenga elementos.
    - Se usa la variable *dest* para guardar temporalmente el nodo al que se dirige la arista que se va a desencolar (Esto es equivalente a recuperar el último dígito de la cadena con el formato “Peso=origen-destino” en el *head* de *pq*).

      Nota**:** En esta implementación del algoritmo, esto último restringe el funcionamiento del programa a grafos con máximo 10 nodos (del 0 al 9). Hay maneras de recuperar más de un dígito por medio de ciclos y comparaciones, pero para mantener el código simple se optó por no eliminar esta restricción.
    - Se “desencola” de *pq*.
    - Si el nodo al que se está llegando no está marcado, se agrega esa arista al árbol de expansión mínimo *MST*, se encolan en *pq* las aristas adyacentes a nodos que aun no se han visitado y se marca el nodo como visitado.
    - Se actualiza la variable *current* con el siguiente nodo que se está revisando.

      Nota**:** Se hace una validación donde se recupera el valor del nodo origen en el *head* de *pq*. Es decir, se extrae el dígito correspondiente a origen de “Peso=origen-destino” de la próxima arista que se va a revisar.

Ejercicio 4. BFS

9. En la clase *Graph*, se agregó el método *BFS*.

Análisis de la implementación del código de *BFS*

- Se crea un arreglo de booleanos del tamaño de nodos en el grafo, *visited*, donde se marcarán los nodos que se revisan como visitados.
- Se crea una cola *queue*.
- Se marca el nodo inicial *s* como visitado y se encola.
- Mientras haya elementos en la cola:
    - Se desencola y se actualiza *s* con el valor desencolado.
    - Se imprime *s*.
    - Se crea un objeto iterable que recorre los nodos adyacentes al nodo *s* (esto es, los elementos de la s-ésima sublista de la lista de adyacencia del grafo *adjArray*).
    - Mientras haya un nodo para la próxima iteración:
        - Se crea la variable *n* y se le asigna el valor del nodo siguiente.
        - Si el nodo no está visitado se encola y se marca como visitado.

El funcionamiento de esta implementación es bastante similar al pseudocódigo que se revisó en clase salvo algunas diferencias mínimas:

- El método imprime la lista de visitados en lugar de devolver la lista.
- Se utiliza la clase *Iterator* para crear un objeto iterable de los nodos adyacentes para cada nodo que se revisa.

Ejercicio 5. DFS

10. En la clase *Graph*, se agregó el método *DFS*.

Análisis de la implementación del código de *DFS*

- El algoritmo en sí se desarrolla en el método *DFSUtil*, sin embargo, dado que se utiliza la recursividad para este algoritmo, es necesario crear la lista de visitados desde "fuera"; para ello se crea otra función desde la cual se crea la lista de visitados y ahora sí, se llama al método recursivo *DFSUtil* pasando la lista de visitados como parámetro.
- En el método *DFSUtil*, primero, se marca el nodo especificado *v* como visitado.
- Se imprime el nodo especificado *v*.
- Se crea un objeto iterable que recorre los nodos adyacentes al nodo *v* (esto es, los elementos de la *v*-ésima sublista de la lista de adyacencia del grafo *adjArray*).
- Mientras haya un nodo para la próxima iteración:
    - Se crea la variable *n* y se le asigna el valor del nodo siguiente.
    - Si el nodo no está visitado se vuelve a llamar a *DFSUtil* con el nodo actual *n*.

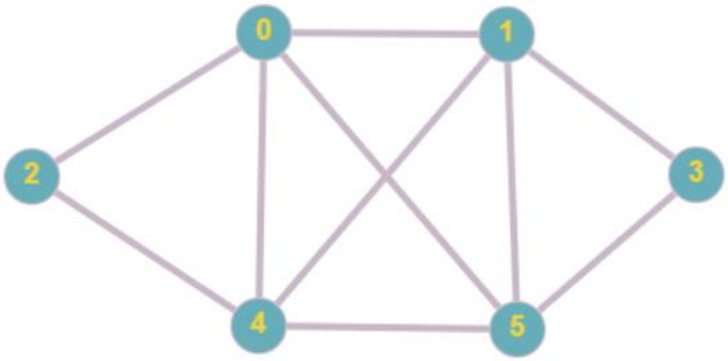
El funcionamiento de esta implementación es bastante similar al pseudocódigo que se revisó en clase salvo algunas diferencias mínimas:

- Como se mencionó anteriormente, en la implementación del algoritmo fue necesario crear otro método encargado de crear la lista de visitados y la llamada inicial al método *DFSUtil*.
- Se utiliza la clase *Iterator* para crear un objeto iterable de los nodos adyacentes para cada nodo que se revisa.

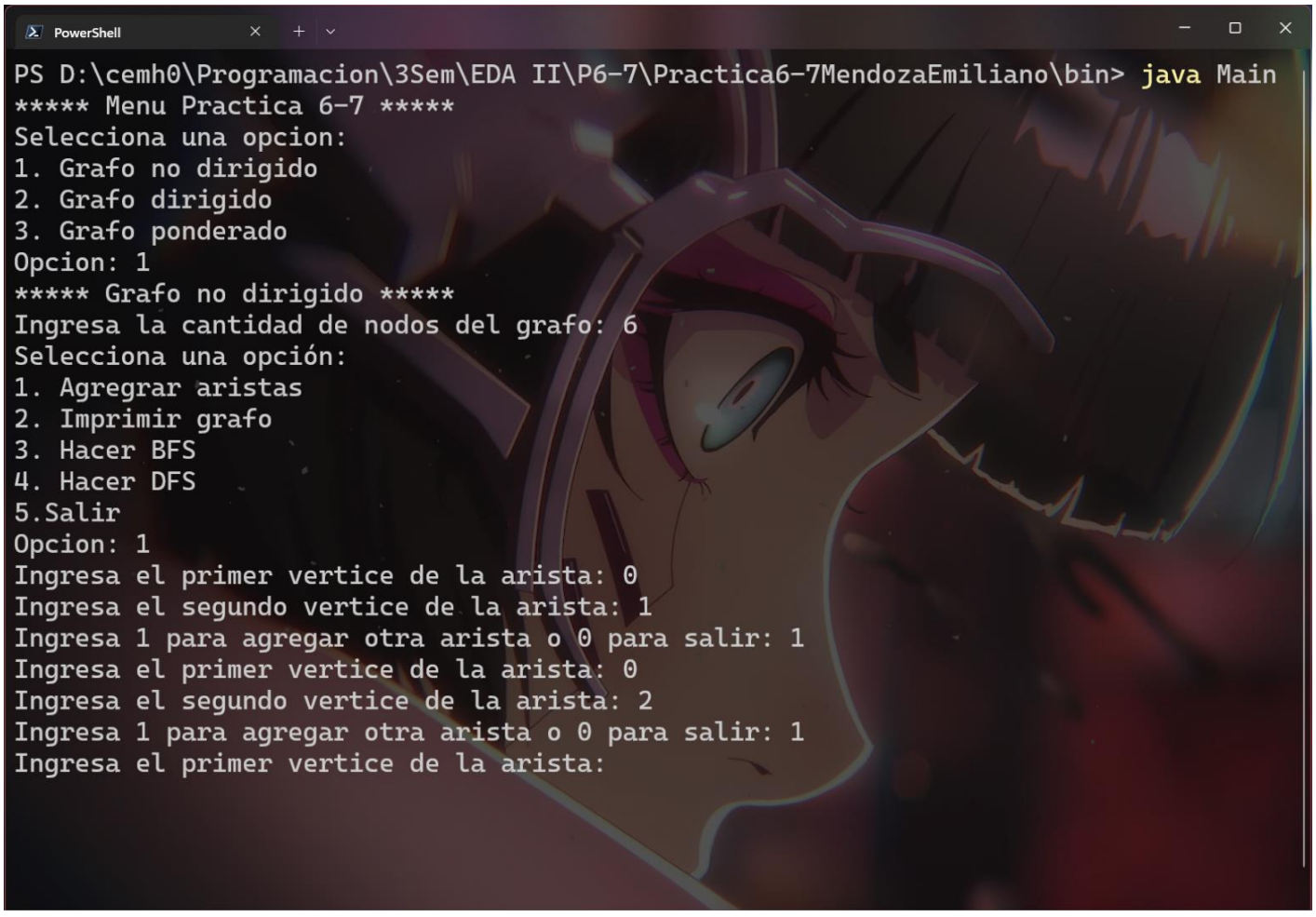
***Observación*:** Estas diferencias, tanto como en *DFS* como en *BFS* respecto a los pseudocódigos vistos en clase, realmente no alteran el funcionamiento del algoritmo y son, por el contrario maneras de aplicar o codificar las líneas que no son tan explícitas en el pseudocódigo.

RESULTADOS OBTENIDOS

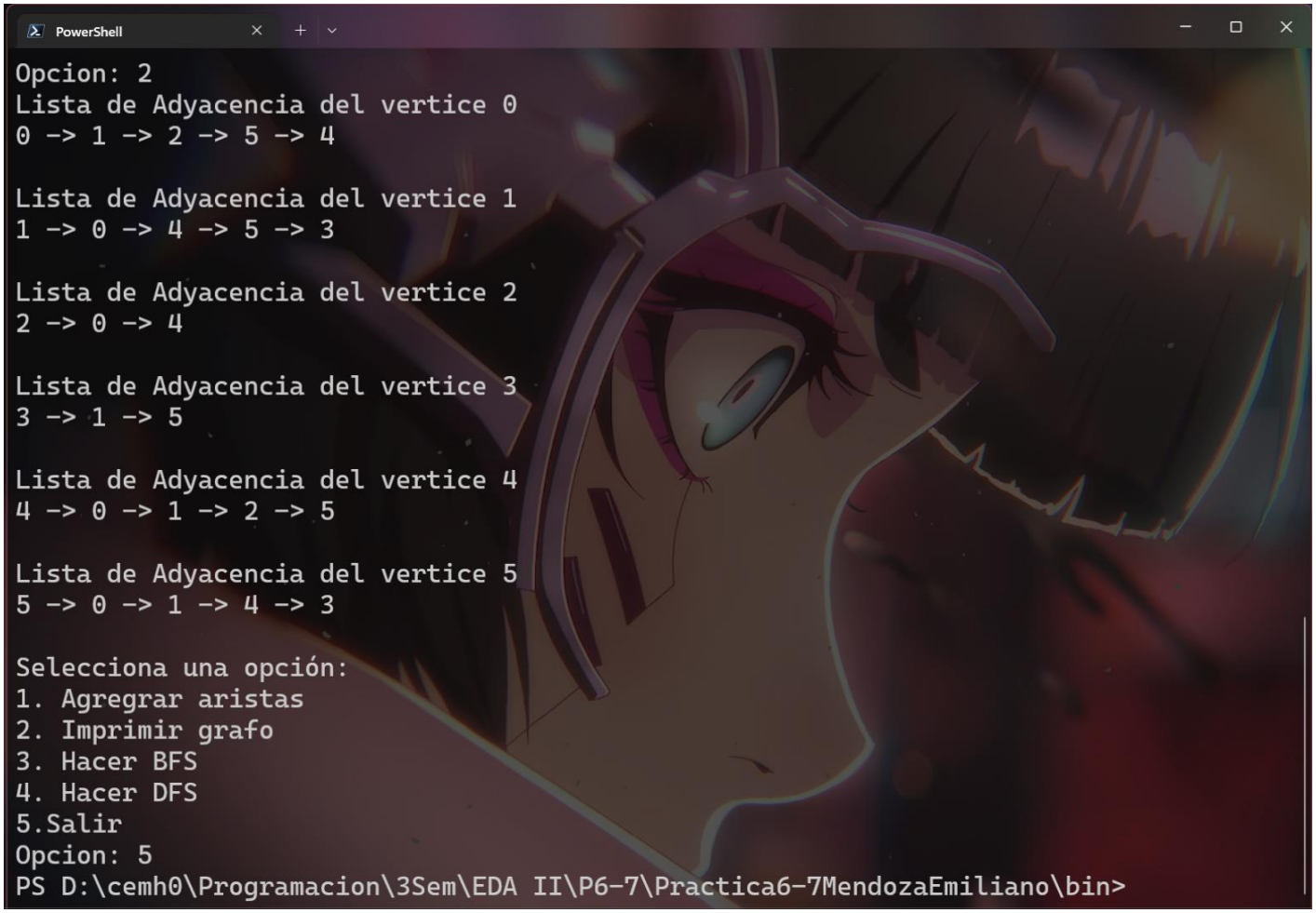
Creando un grafo no dirigido desde la interfaz del programa



Se selecciona la opción de grafo no dirigido en el menú; a continuación, se selecciona la opción de agregar aristas y se agregan de la siguiente manera:

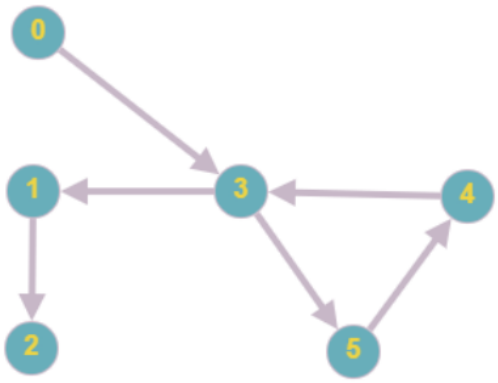


Después de agregar todas las aristas, se selecciona la opción de imprimir grafo y se obtiene la siguiente lista de adyacencia:



***Observación*:** Al comparar la lista de adyacencia con el dibujo del grafo, se corrobora que la lista de adyacencia es coherente con el grafo, la cual verifica que el programa funciona correctamente:

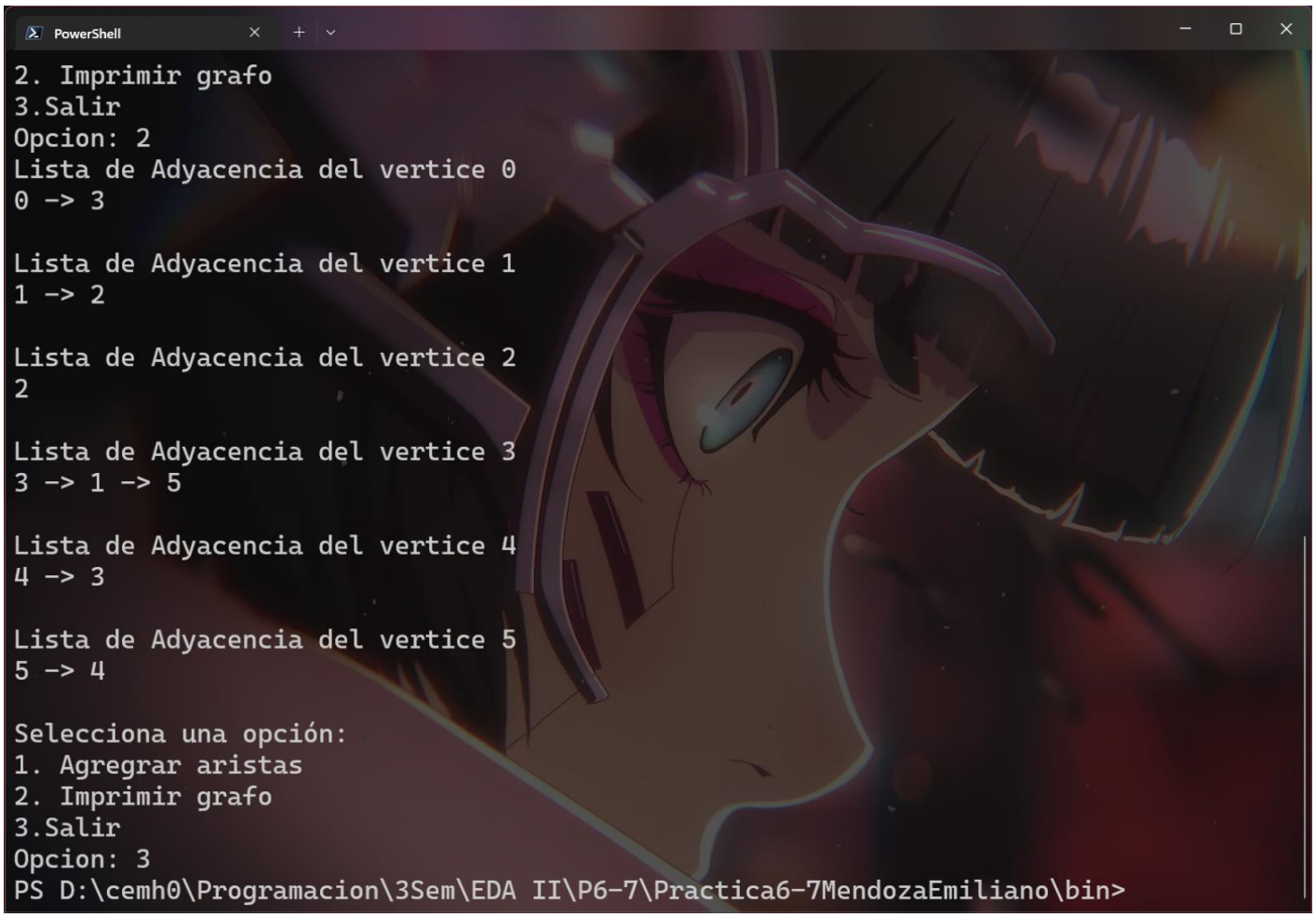
Creando un grafo dirigido desde la interfaz del programa



Se selecciona la opción de grafo dirigido en el menú; a continuación, se selecciona la opción de agregar aristas y se agregan de la siguiente manera:

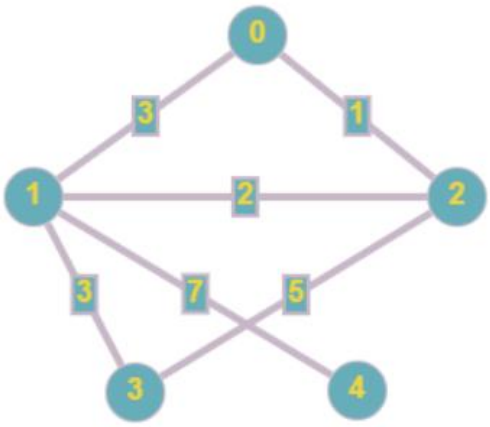
```
PowerShell
PS D:\cemh0\Programacion\3Sem\EDA II\P6-7\Practica6-7MendozaEmiliano\bin> java Main
***** Menu Practica 6-7 *****
Selecciona una opcion:
1. Grafo no dirigido
2. Grafo dirigido
3. Grafo ponderado
Opcion: 2
***** Grafo dirigido *****
Ingresa la cantidad de nodos del grafo: 6
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3.Salir
Opcion: 1
Ingresa el vertice de origen de la arista: 0
Ingresa el vertice de destino de la arista: 3
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el vertice de origen de la arista: 1
Ingresa el vertice de destino de la arista: 2
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el vertice de origen de la arista: 3
Ingresa el vertice de destino de la arista: 1
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el vertice de origen de la arista: 4
Ingresa el vertice de destino de la arista: 3
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el vertice de origen de la arista: 3
```

Después de agregar todas las aristas, se selecciona la opción de imprimir grafo y se obtiene la siguiente lista de adyacencia, la cual verifica que el programa funciona correctamente:



***Observación*:** Al comparar la lista de adyacencia con el dibujo del grafo, se corrobora que la lista de adyacencia es coherente con el grafo, la cual verifica que el programa funciona correctamente:

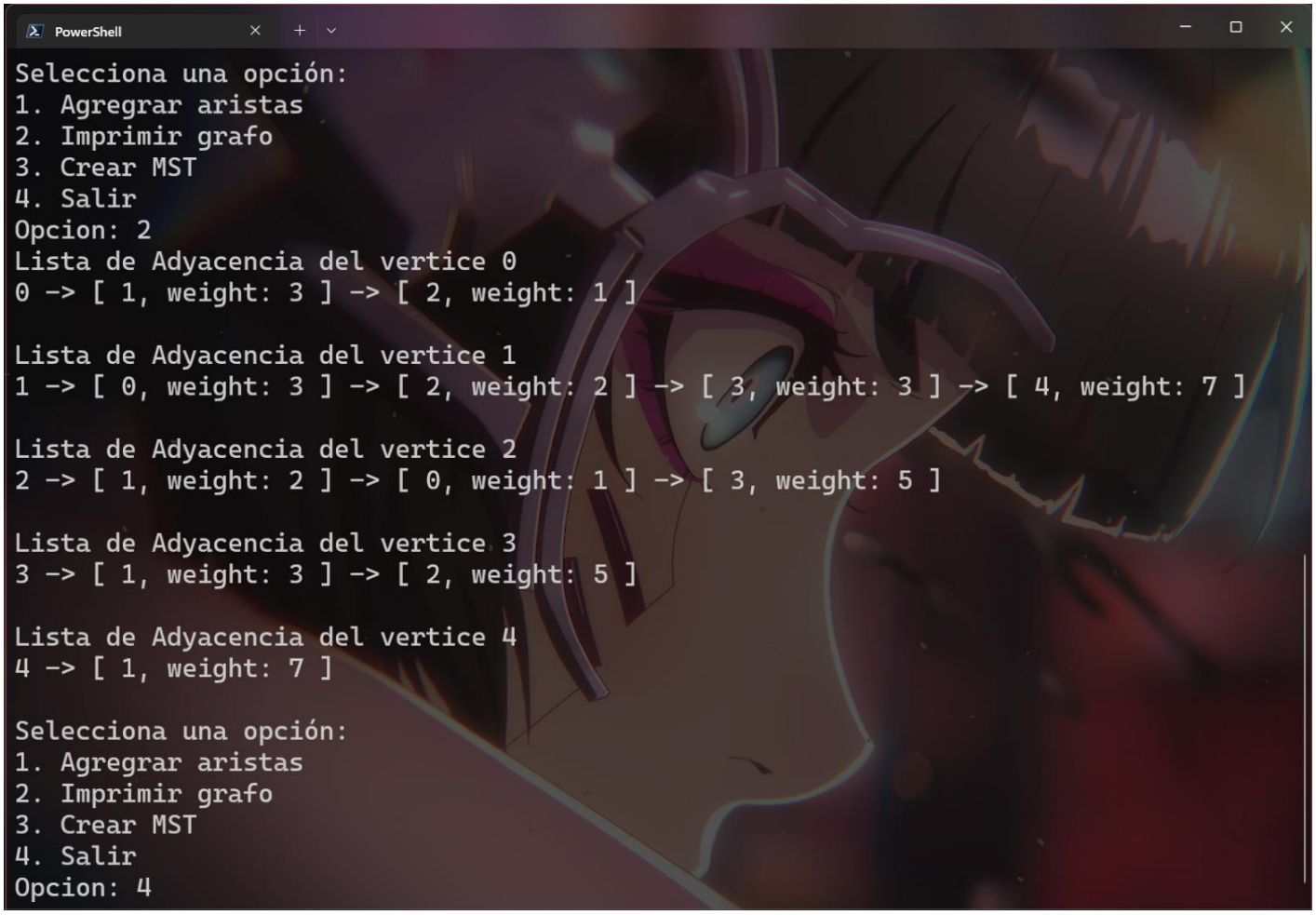
Creando un grafo ponderado desde la interfaz del programa



Se selecciona la opción de grafo dirigido en el menú; a continuación, se selecciona la opción de agregar aristas y se agregan de la siguiente manera:

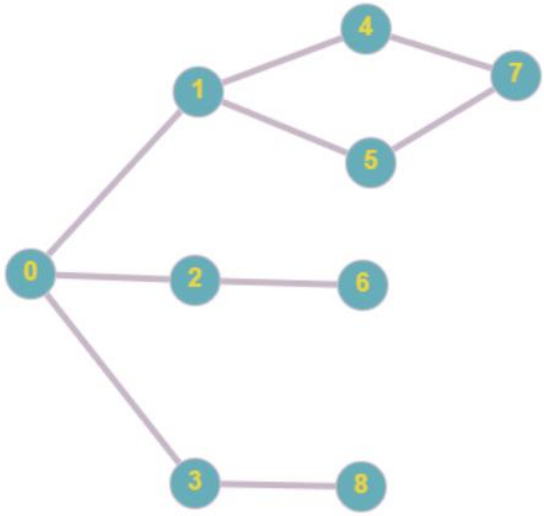
```
PowerShell
PS D:\cemh0\Programacion\3Sem\EDA II\P6-7\Practica6-7MendozaEmiliano\bin> java Main
***** Menu Practica 6-7 *****
Selecciona una opcion:
1. Grafo no dirigido
2. Grafo dirigido
3. Grafo ponderado
Opcion: 3
***** Grafo ponderado *****
Ingresa la cantidad de nodos del grafo: 5
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Crear MST
4. Salir
Opcion: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 1
Ingresa el valor del peso de la arista: 3
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 1
Ingresa el segundo vertice de la arista: 2
Ingresa el valor del peso de la arista: 2
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 2
Ingresa el valor del peso de la arista: 1
Ingresa 1 para agregar otra arista o 0 para salir: 1
```

Después de agregar todas las aristas, se selecciona la opción de imprimir grafo y se obtiene la siguiente lista de adyacencia, la cual verifica que el programa funciona correctamente:



***Observación*:** Al comparar la lista de adyacencia con el dibujo del grafo, se corrobora que la lista de adyacencia es coherente con el grafo, la cual verifica que el programa funciona correctamente:

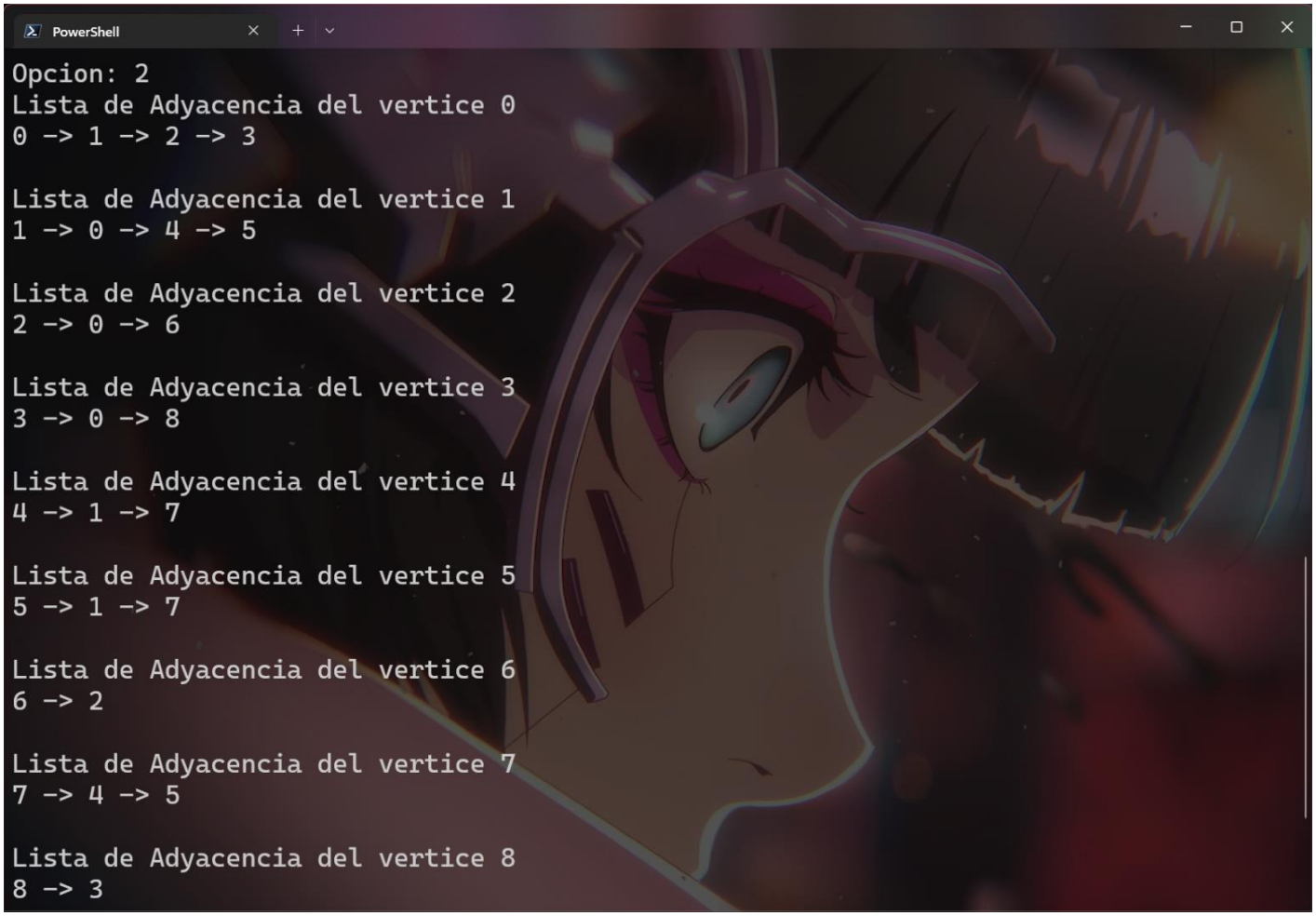
Verificando el funcionamiento de BFS con el grafo:



Se selecciona la opción de grafo no dirigido en el menú para crear un grafo con 9 vértices:

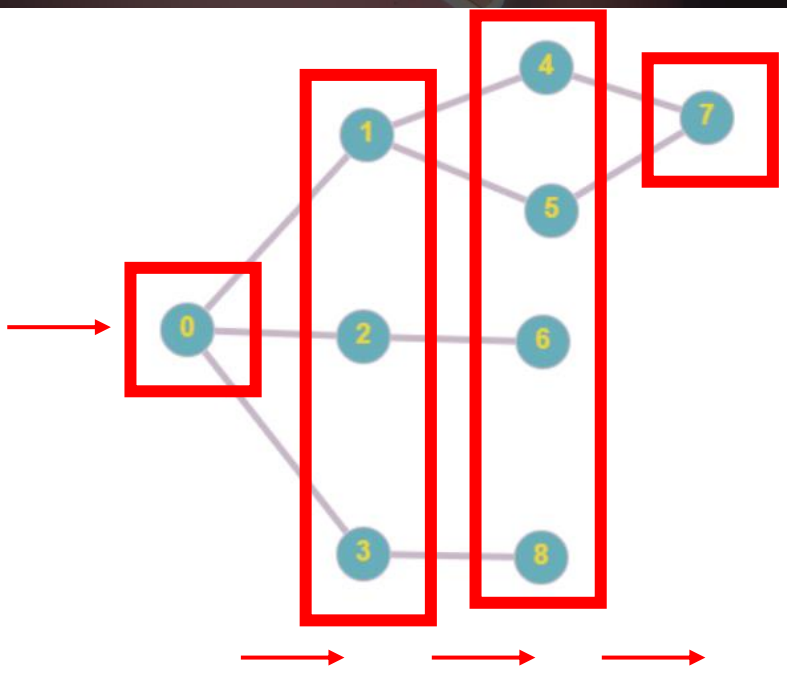
```
PS D:\cemh0\Programacion\3Sem\EDA II\P6-7\Practica6-7MendozaEmiliano\bin> java Main
***** Menu Practica 6-7 *****
Selecciona una opcion:
1. Grafo no dirigido
2. Grafo dirigido
3. Grafo ponderado
Opcion: 1
***** Grafo no dirigido *****
Ingresa la cantidad de nodos del grafo: 9
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 1
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 2
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 3
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 1
Ingresa el segundo vertice de la arista: 4
```

Después de agregar todas las aristas se tiene la lista de adyacencia:

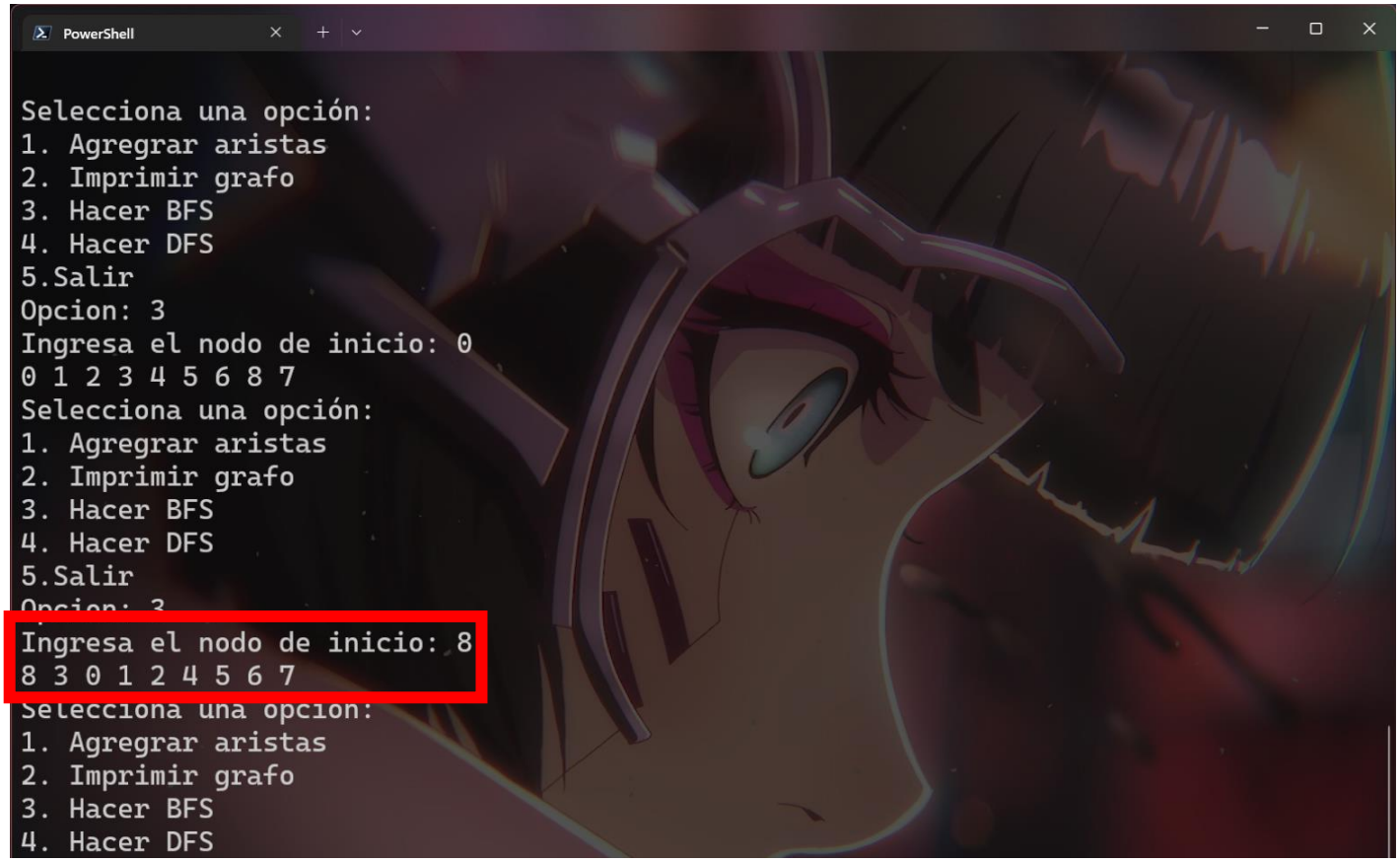


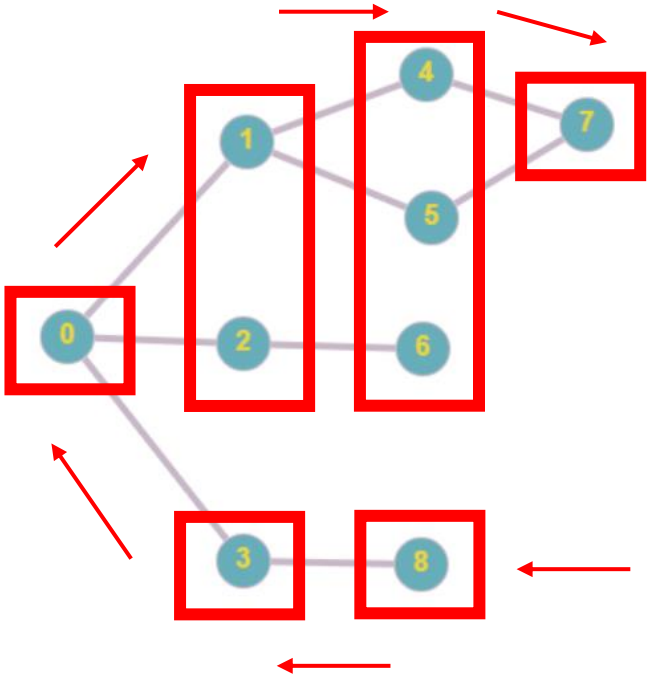
Se selecciona la opción de hacer recorrido BFS iniciando por el nodo 0:

```
PowerShell
5 -> 1 -> 7

Lista de Adyacencia del vertice 6
6 -> 2

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

Lista de Adyacencia del vertice 8
8 -> 3

Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 3
Ingresa el nodo de inicio: 0
0 1 2 3 4 5 6 8 7
Selecciona una opcion:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
```

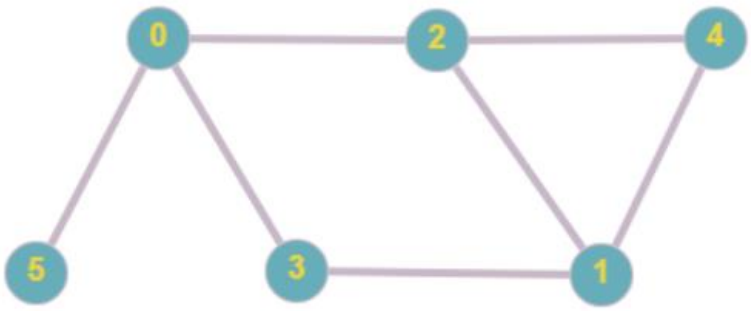


Se selecciona la opción de hacer recorrido BFS iniciando por el nodo 8 y se observa el siguiente recorrido:





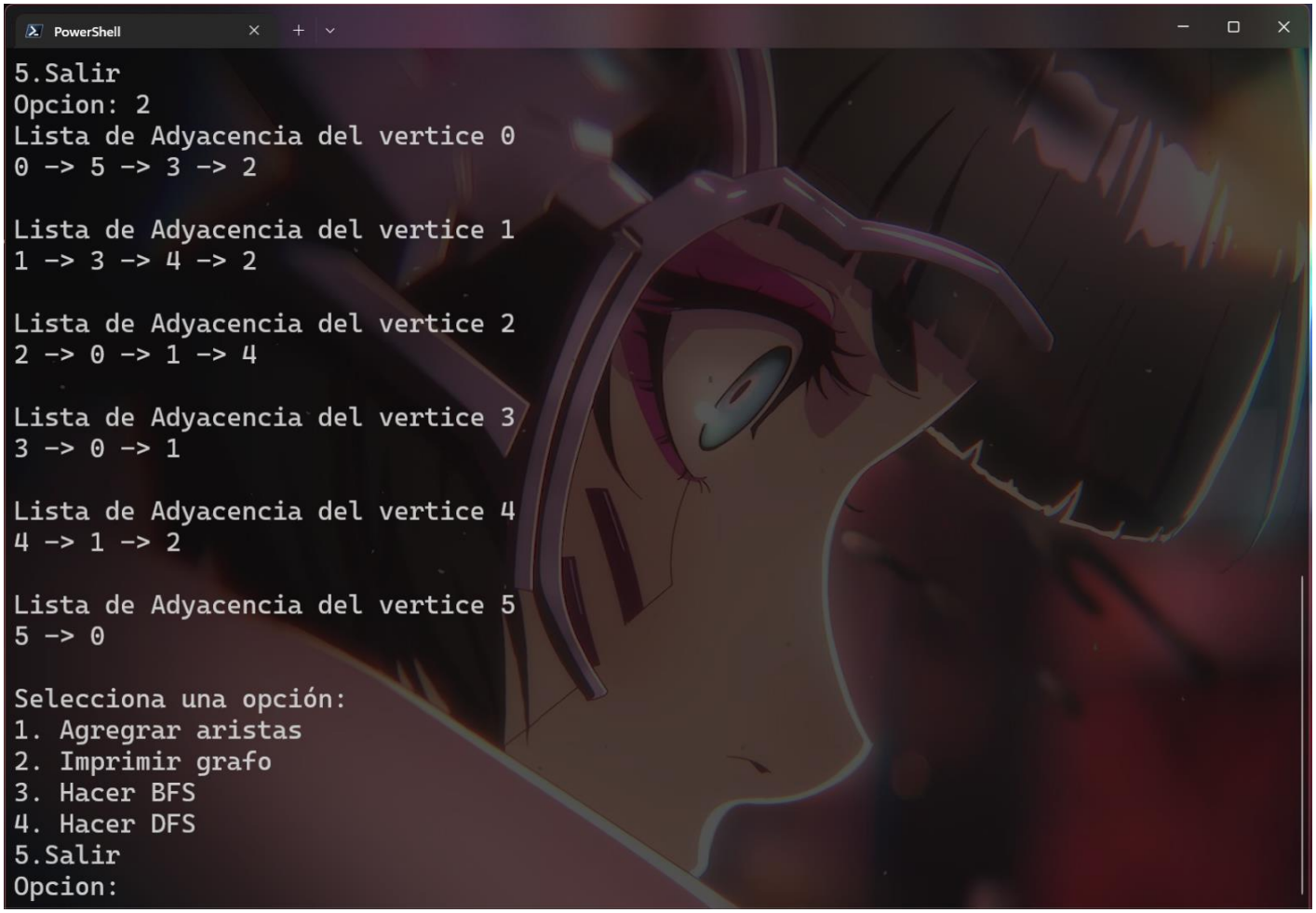
Verificando el funcionamiento de BFS con el grafo:



Se selecciona la opción de grafo no dirigido en el menú para crear un grafo con 6 vértices:

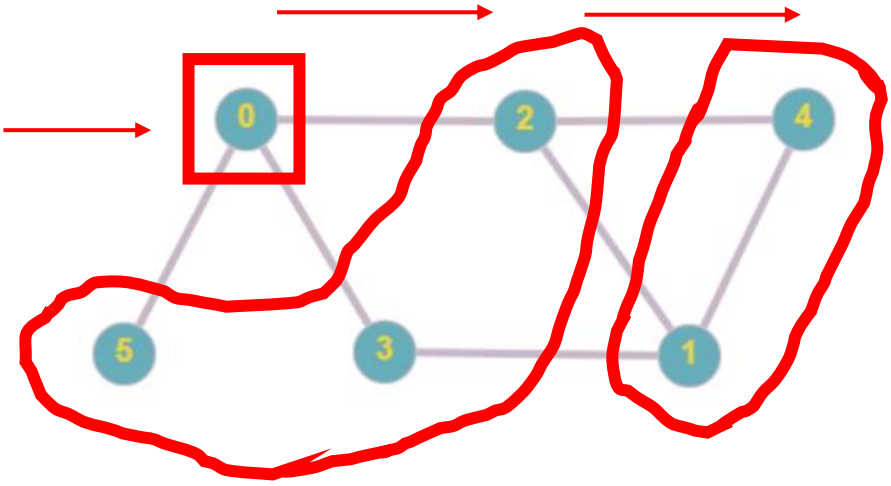
```
PS D:\cemh0\Programacion\3Sem\EDA II\P6-7\Practica6-7MendozaEmiliano\bin> java Main
***** Menu Practica 6-7 *****
Selecciona una opcion:
1. Grafo no dirigido
2. Grafo dirigido
3. Grafo ponderado
Opcion: 1
***** Grafo no dirigido *****
Ingresa la cantidad de nodos del grafo: 6
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 5
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 3
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 0
Ingresa el segundo vertice de la arista: 2
Ingresa 1 para agregar otra arista o 0 para salir: 1
Ingresa el primer vertice de la arista: 1
Ingresa el segundo vertice de la arista: 3
```

Después de agregar todas las aristas se tiene la lista de adyacencia:



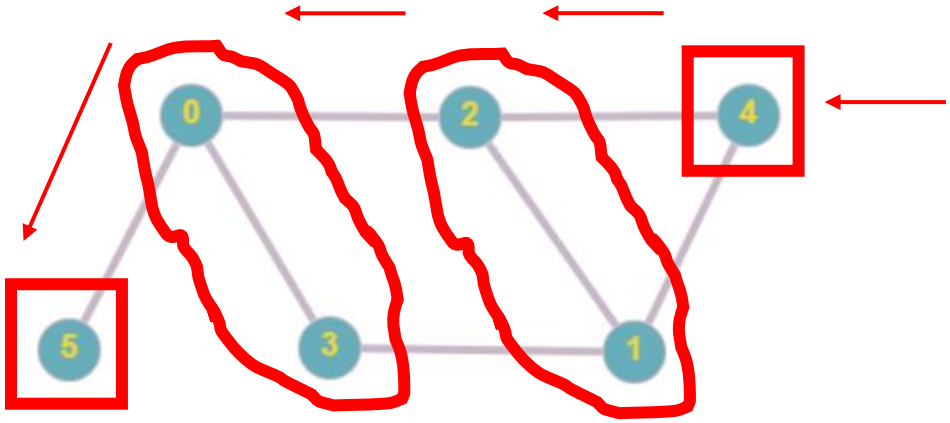
Se selecciona la opción de hacer recorrido BFS iniciando por el nodo 0:

```
PowerShell
2 -> 0 -> 1 -> 4

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

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

Lista de Adyacencia del vertice 5
5 -> 0

Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 3
Ingresa el nodo de inicio: 0
0 5 3 2 1 4
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion:
```



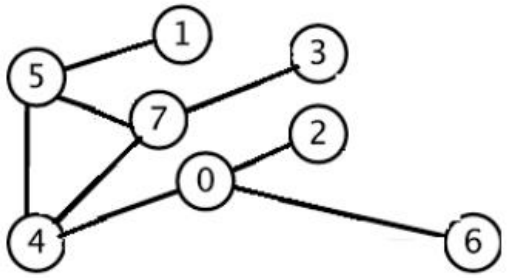
Se selecciona la opción de hacer recorrido BFS iniciando por el nodo 8 y se observa el siguiente recorrido:

```
PowerShell
5 -> 0

Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 3
Ingresa el nodo de inicio: 0
0 5 3 2 1 4
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 3
Ingresa el nodo de inicio: 4
4 1 2 3 0 5
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion:
```

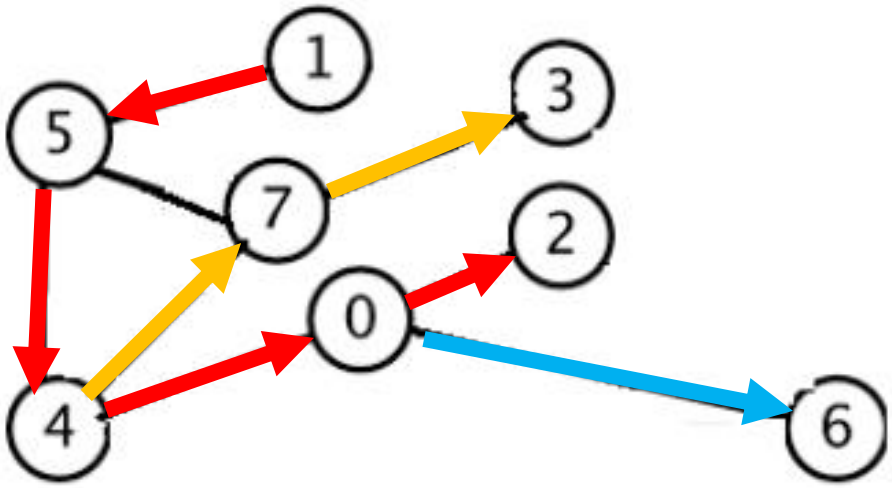


Observaciones: En ambas ejecuciones puede notarse el recorrido “en capas” que hace el algoritmo, sin embargo, no siempre resulta ser un recorrido “corto” dependiendo del nodo de inicio y de las conexiones entre los nodos. Todas las listas de visitados son válidas para recorridos *BFS* de los grafos mostrados, por lo tanto, se corrobora que el algoritmo funciona correctamente.

Verificando el funcionamiento de DFS con el grafo:

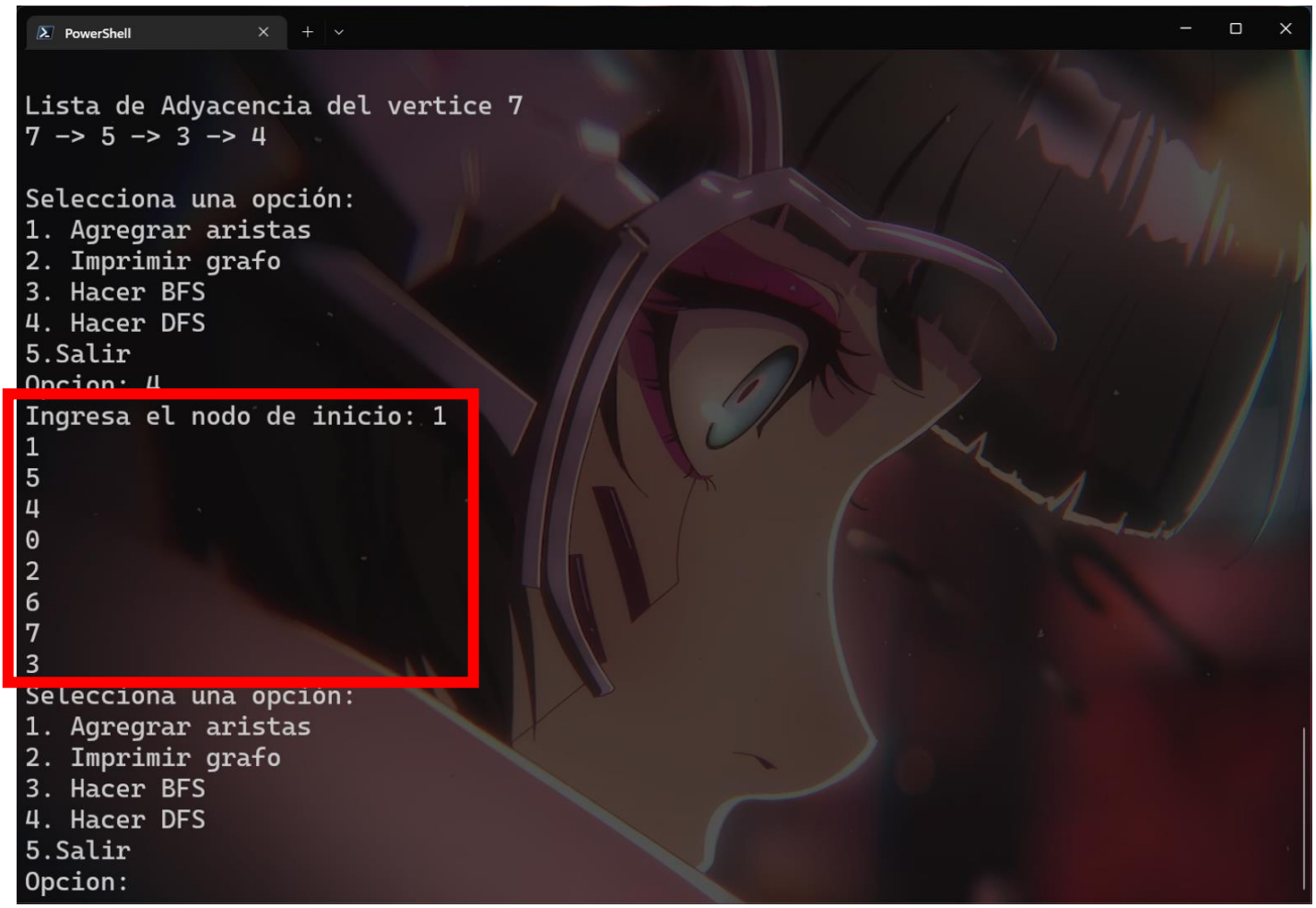


1) **Iniciando en el nodo 1**

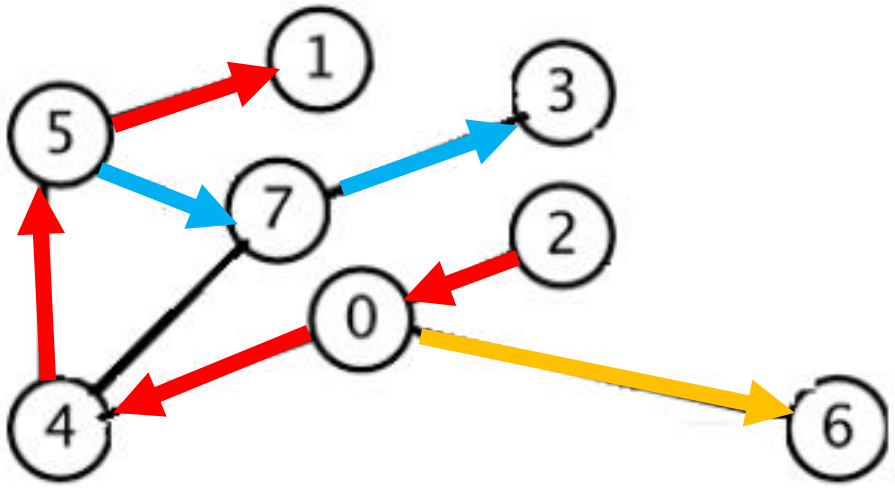


$$Visitados: \{1,5,4,0,2,6,7,3\}$$

Ejecutando en el programa el resultado es:



2) Iniciando en el nodo 2

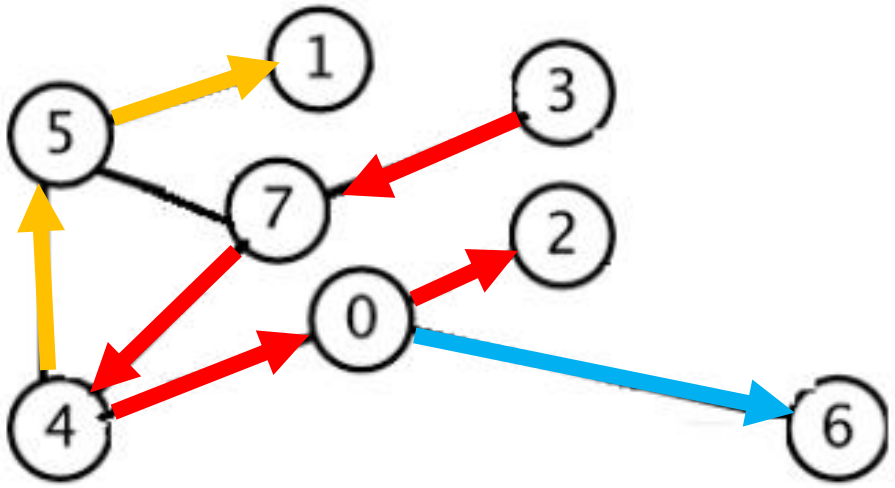


$$Visitados: \{2,0,4,5,1,7,3,6\}$$

Ejecutando en el programa el resultado es:

```
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 4
Ingresa el nodo de inicio: 2
2
0
4
5
1
7
3
6
Selecciona una opcion:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion:
```

3) Iniciando en el nodo 3



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

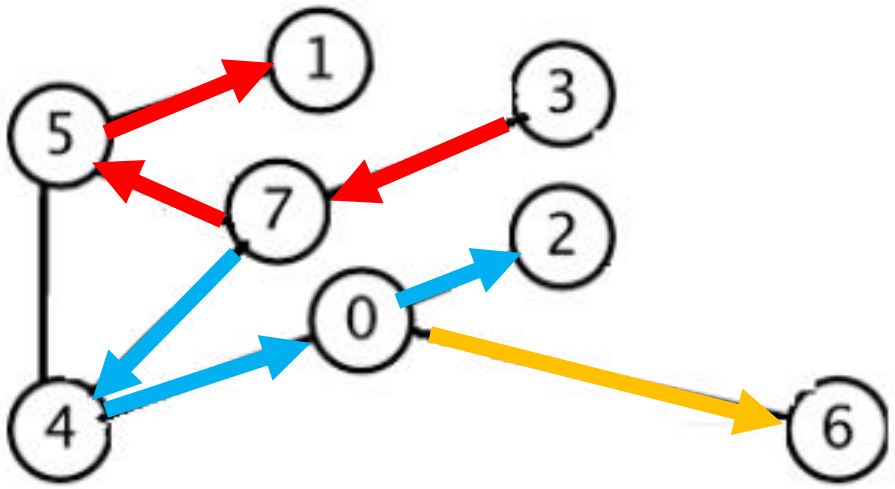
Ejecutando en el programa el resultado es:

```
Selecciona una opción:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion: 4
Ingresa el nodo de inicio: 3
3
7
5
1
4
0
2
6
Selecciona una opcion:
1. Agregrar aristas
2. Imprimir grafo
3. Hacer BFS
4. Hacer DFS
5.Salir
Opcion:
```

Observaciones: En los primeros dos recorridos, se obtuvo la misma lista de visitados con el programa y al hacerlo manualmente. Sin embargo, en el último recorrido no coincide la lista de visitados del programa con la que se obtuvo manualmente.

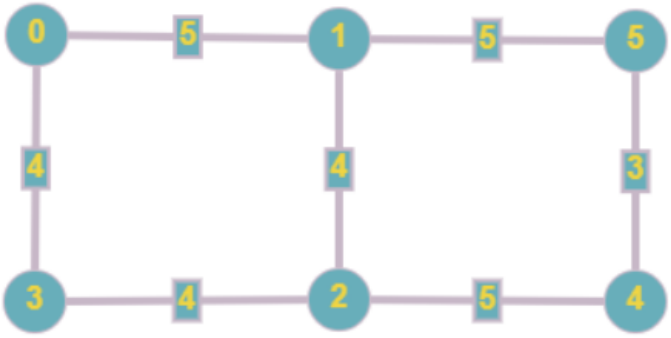
Esto se debe a que al hacer el recorrido manualmente, se consideró como regla interna visitar los nodos en orden numérico ascendente. Realmente, no considerar esta regla interna no altera el hecho de que se haga el recorrido correctamente dado que no es relevante el orden en que se visitan los nodos.

Como puede observarse la lista de visitados también es un recorrido *DFS*. Habiendo pasado todas las pruebas, se corrobora que el algoritmo funciona correctamente.

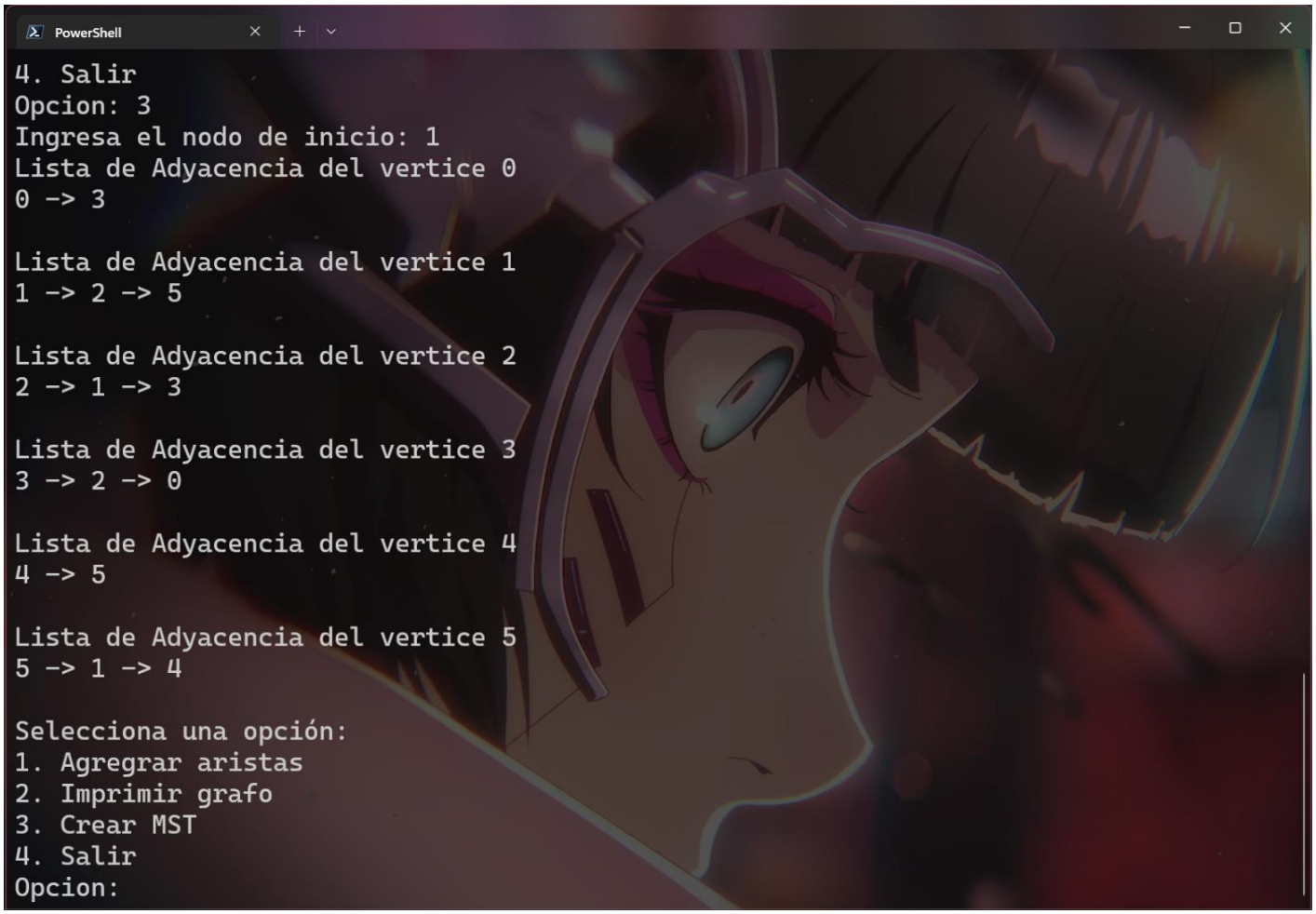


$$Visitados: \{3,7,5,1,4,0,2,6\}$$

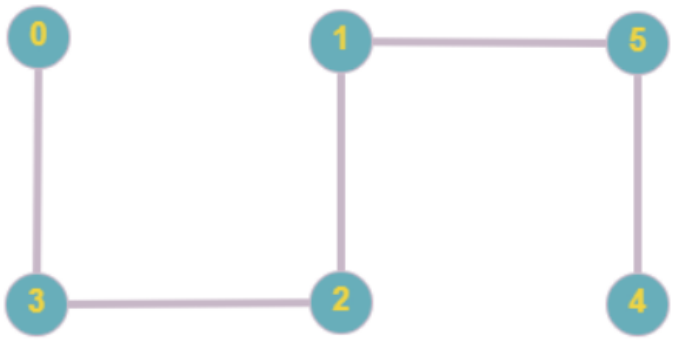
Creando un MST del siguiente grafo iniciando por el nodo 1:



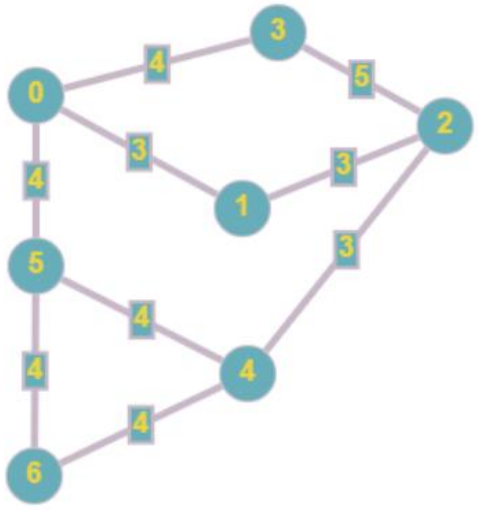
Se ingresa el grafo en el programa y se selecciona la opción de crear *MST*:



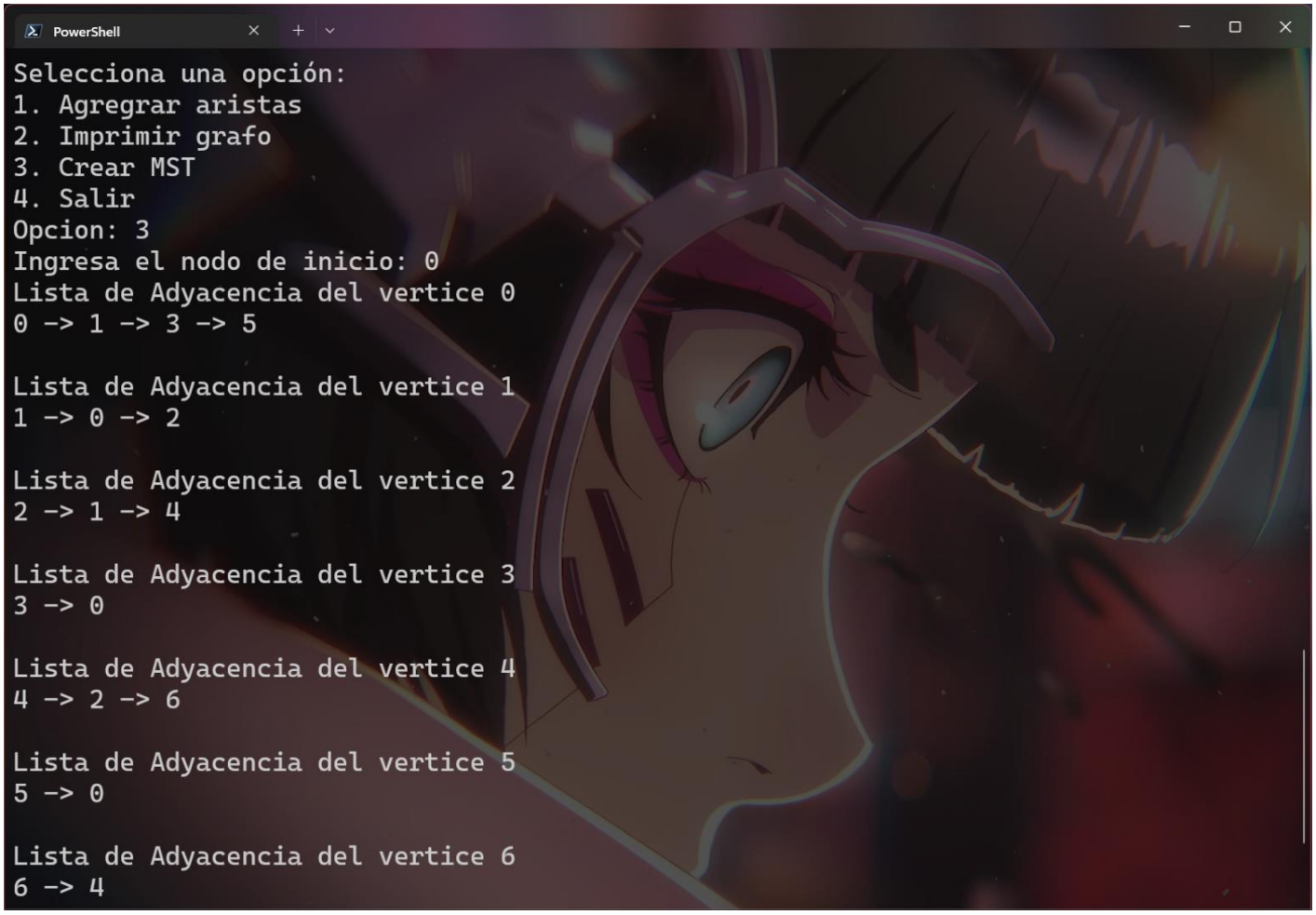
MST:



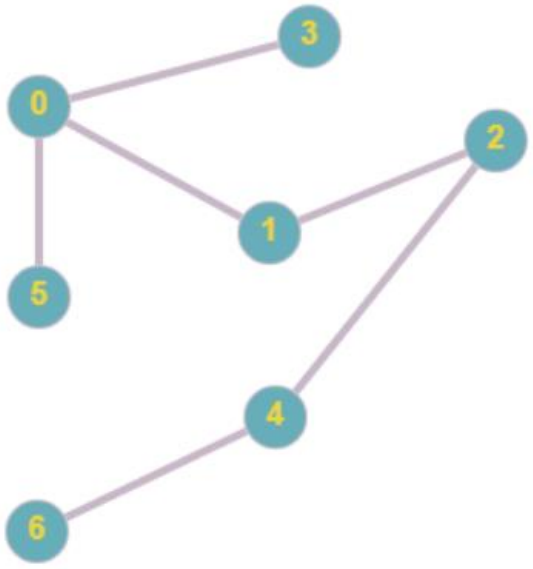
Creando un MST del siguiente grafo iniciando en el nodo 0:



Se ingresa el grafo en el programa y se selecciona la opción de crear *MST*:



MST:



Observaciones: Al usar el método *prim*, se imprime la lista de adyacencia de un grafo no dirigido y no ponderado que representa al árbol de expansión mínimo del grafo con el que se llamó el método. Ambos árboles son correctos, por lo tanto, se verifica el funcionamiento de este algoritmo.

CONCLUSIONES

Algunos puntos que se concluyen de la primera actividad son los siguientes:

- Una forma de implementar grafos en el lenguaje Java puede ser representando los grafos por medio de su lista de adyacencia.
- La lista de adyacencia de un grafo está compuesta esencialmente por una lista de listas.
- Para agregar una arista a un grafo no dirigido se debe agregar la adyacencia de los nodos “en ambos sentidos”.

Para el segundo ejercicio se puede concluir que:

- Partiendo de la implementación anterior de grafos no dirigidos es muy sencillo modificarla para representar grafos dirigidos.
- Para agregar una arista a un grafo dirigido se debe tomar en cuenta desde que nodo se parte y a cuál nodo se dirige y agregar la adyacencia en un solo sentido.

Al realizar el ejercicio 3 se llegó a las siguientes conclusiones:

- Modificando, una vez más, la implementación de grafos no dirigidos, se pueden representar grafos ponderados agregando un espacio para cada arista que guarde el peso de sí misma.
- Con una estructura tipo *HashMap* fue posible guardar pares <destino, peso> y así modificar la lista de adyacencia para grafos ponderados.

Además, de los ejercicios 4 y 5 se concluye lo siguiente:

- El recorrido *BFS* “avanza” en el grafo seccionándolo por capas para cada nivel de nodos adyacentes.
- El recorrido *DFS* se pude implementar con recursividad.
- En *DFS* se hace el recorrido a profundidad y cuando se llega al “tope” y se regresa para probar por otra ruta.

Dicho todo lo anterior, considero que se ha cumplido con cada uno de los objetivos planteados para el desarrollo de esta práctica dado que se conocieron e identificaron las características necesarias para implementar y comprender los grafos, así como los algoritmos de búsqueda por expansión y profundidad. Además, todo lo anterior se realizó con un enfoque orientado a objetos.

Como comentario final, considero que trabajar con grafos puede ser muy complejo dado que no hay un orden secuencial para recorrer sus elementos. Al no haber un orden implica que se pueden tener diferentes resultados incluso siguiendo el mismo algoritmo de recorrido. Otra característica que añade complejidad a los grafos respecto a las estructuras de datos lineales son las aristas ponderadas, esto da pie a que sea de interés obtener el árbol de expansión mínimo o la ruta más corta entre nodos.