Índice

1. Objetivos

- Objetivo P6: El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para comprender el algoritmo de búsqueda por expansión.
- Objetivo P7: El estudiante conocerá e identificará las características necesarias para entender el algoritmo de búsqueda por profundidad en un grafo.

**Nota: Para el desarrollo de la práctica usaré las herramientas:*

- *Visual Studio Code: para crear e implementar los programas en Java.*
- *Overleaf: para crear este documento en LaTeX del reporte.*

2. Desarrollo

2.1. Análisis de la Implementación de Grafos en Java

En la práctica se presentan implementaciones diferentes de grafos

Grafo No ponderado:
Atributos:

- Una lista tipo *List* de vértices (nodos del grafo)
- Una lista de listas de tipo entero que es la lista de adyacencia
- Un booleano que determina si está dirigido o no

Métodos:

- **Constructor** que recibe como parámetro un booleano, dentro se inicializan los atributos *vertices* y *listaAdyacencia* con un nuevo *ArrayList*, y al atributo *dirigido* se le asigna el parámetro booleano que recibe el constructor

- ***agregarVertice(String nombre)*:** recibe como parámetro un *String* llamado *nombre*
  Si la lista de *vertices* NO contiene el *nombre* que se ingresó, se añade este string a la lista de *vertices* y se crea una nueva *listaAdyacencia* para este vertice

- ***agregarArista(String v1, String v2)*:**
  se recibe como string los vértices que está conectando la arista, *i1* guarda el índice de *v1* en la lista de *vertices* e *i2* guarda el indice de *v2*, luego, si ambos indices son diferentes a -1 (-1 querría decir que no se encuentran en la lista de vértices o sea que no existen en el grafo): a la lista de adyacencia de *v1* se añade el indice de *v2*, luego, si el grafo NO es dirigido, a la lista de *v2* se agrega el indice de *v1*

Nota: entonces si el grafo SI es dirigido, hay que añadir los vertices en orden: origen –>destino, pero si es NO dirigido, no importa el orden en que se ingresan los vertices al metodo

- ***public void imprimirGrafo()*:**
  - Con un *for* recorre la lista de vértices usando su tamaño con el metodo *size()*
  - imprime el vértice en el indice que se está revisando
  - con un *for each* con la variable auxiliar *vecino* se recorre la lista de adyacencia del vertice que se esta revisando y se imprime concatenando espacios

Grafo Ponderado
Atributos:

- Una lista tipo *List* de vértices (nodos del grafo)
- Una lista de listas de tipo entero que es la lista de adyacencia
- Un booleano que determina si está dirigido o no

Métodos:

- **Constructor** que recibe como parámetro un booleano, dentro se inicializan los atributos *vertices* y *listaAdyacencia* con un nuevo *ArrayList*, y al atributo *dirigido* se le asigna el parámetro booleano que recibe el constructor

- ***agregarVertice(String nombre)*:** recibe como parámetro un *String* llamado *nombre*
  Si la lista de *vertices* NO contiene el *nombre* que se ingresó, se añade este string a la lista de *vertices* y se crea una nueva *listaAdyacencia* para este vertice

- ***public void agregarArista(String v1, String v2, int peso)*:** recibe como parámetros los nodos que conecta y el peso de la arista (su valor) *i1* guarda el indice de *v1* en la lista de *vertices* e *i2* guarda el indice de *v2*, luego, si ambos indices son diferentes a -1 (-1 querría decir que no se encuentran en la lista de vértices o sea que no existen en el grafo): a la lista de adyacencia de *v1* se añade un nuevo objeto de tipo *Arista* que recibe como parámetro en su constructor el indice de *v2* y el peso que se recibio en los argumentos del metodo, si el grafo no es dirigido, se agrega otro objeto de tipo *Arista* a la lista de adyacencia de *v2*, pasándole como argumento a su constructor el indice de *v1* como destino y el peso que se puso en los argumentos del metodo.

Nota: entonces si el grafo SI es dirigido, hay que añadir los vertices en orden: origen –>destino, pero si es NO dirigido, no importa el orden en que se ingresan los vertices al metodo

- ***public void imprimirGrafo()*:**
  - Con un *for* recorre la lista de vértices usando su tamaño con el metodo *size()*
  - imprime el vértice en el índice que se está revisando
  - **con un *for each* con el objeto auxiliar *arista* de tipo *Arista* se recorre la lista de adyacencia del vertice que se esta revisando y se imprime: la lista de vertices en la posicion de *arista* y se accede a su atributo *destino*, y también su *peso***

- **¿Cómo se determina si un grafo es dirigido o no dirigido en el programa proporcionado?**

Con el atributo booleano *dirigido*

- **¿Te parece adecuada esta manera de implementarlo?, ¿De qué otra forma se podría implementar esto?**

Si me parece adecuada, pero creo que se podría hacer que la clase *Grafo* fuera la clase padre de *GrafoPonderado* ya que son parecidos en la mayor parte de su implementación, solo tendrían que sobreescribirse los métodos haciendo uso de la clase *Arista* en lugar de solo los indices de los vertices.

- **¿Para que sirve la clase Arista?**

La usa la clase *GrafoPonderado* para que este objeto guarde el destino y el valor o peso de la arista que conecta dos vértices.

- **Dibuja los grafos que se crean en la clase Principal**

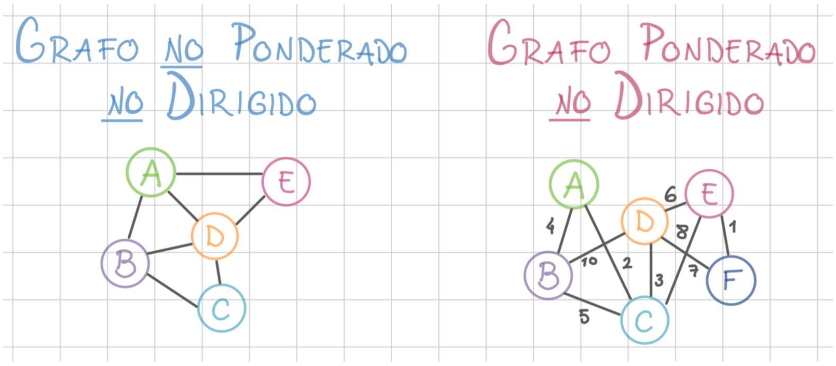
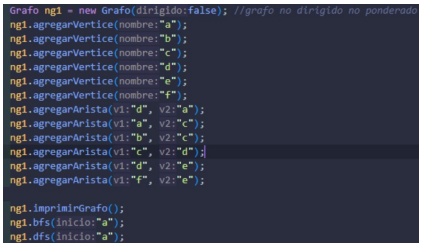


Figura 1: Grafos precargados

2.2. Nuevos grafos

Agregué los dos grafos con los mismos datos, para ver las diferencias de los métodos cuando son de tipo diferente pero con los mismos nodos y aristas.



(a)

(b)

Figura 2: Agregar otros grafos

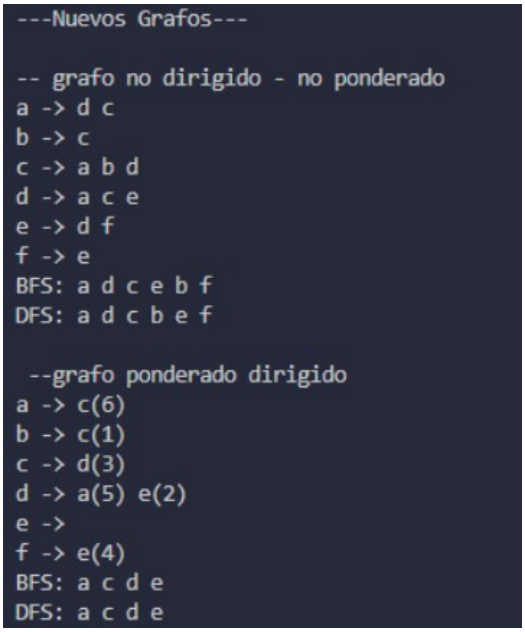


Figura 3: Resultado de los nuevos grafos

Principales diferencias:

en las impresiones lo que se imprime realmente son las listas de adyacencia de ambos grafos, que como vimos en clase son diferentes si son para dirigido y no dirigido, por eso podemos ver que por ejemplo a y d tienen una arista, en el NO dirigido, *d* está en la lista de *a*, y *a* está en la lista de *d*, pero en el que SÍ es dirigido, podemos ver que *d* no está en la lista de *a*, pero *a* sí está en la lista de *d*, esto debido a que en el método *agregarArista* se ingresó:

ng2.agregarArista("d", .ª", 5);

entonces el programa recibe al primer vertice como origen y el segundo como destino entonces queda d→a

Se puede ver tambien que DFS y BFS como vimos en la clase NO garantizan visitar todos los nodos en un grafo dirigido, ya que hay algunas aristas que no tienen aristas de entrada, solo de salida, por lo que no se puede acceder a ellas

2.3. Algoritmos BFS y DFS

2.3.1. Explica detalladamente las diferencias entre BFS Y DFS implementadas en la clase Grafo con respecto a las vistas en clase

La diferencia está en la *representación de los datos*.

Manejo de Nodos: Los algoritmos BFS y DFS se explicaron en la clase usando "nodos" o "vértices".
En la implementación no se trabaja directamente con los nombres de los nodos, usa dos listas:
- *private List<String>vertices*
- *private List<List<Integer* listaAdyacencia
En BFS o DFS, se usa el índice en la lista *vertices* de el nodo a revisar. Toda la lógica de BFS y DFS (la cola, la pila, el arreglo de *visitado*) usa los **índices**, no los nombres de los nodos.

Análisis de BFS (Búsqueda por Expansión): Usa una **Cola** para recorrer el grafo por niveles. Visita un nodo, luego encola a *todos* sus vecinos no visitados, y después procesa el siguiente nodo de la cola. En la implementación se usan las iteraciones sobre el arreglo DESDE EL INICIO para que el primero en entrar sea el primero en salir

- Usa una *Queue* de enteros (una Cola con indices).
- Usa un arreglo de booleanos basado en los índices de los *visitados*
- El *while while (!cola.isEmpty())* procesa los nodos nivel por nivel, exactamente como dicta el algoritmo.

Análisis de DFS (Búsqueda por Profundidad):

- Teoría: Recorre a profundidad implementando una Pila.
- Implementación: El método *dfs* usa la implementación recursiva. Llama a un método auxiliar *recorrer(int actual, boolean[] visitado)*
- Dentro de *recorrer*, marca el nodo actual como visitado y hace una llamada recursiva para el primer vecino no visitado que encuentra (*if (!visitado[vecino]) recorrer(vecino, visitado);*).

2.3.2. Explica detalladamente las diferencias entre BFS Y DFS de Grafo con respecto a GrafoPonderado

La diferencia es en cómo se accede a los vecinos, en *Grafo* se accede directamente con *vecino*, y *GrafoPonderado* usa el atributo de *Arista* llamado *destino*.

Tanto en *Grafo.java* como en *GrafoPonderado.java*, los algoritmos BFS y DFS: En ambos usa un *boolean[] visitado* y un método *recorrer*. La diferencia es cómo implementan el método de recorrer

La diferencia está en la estructura de la listaAdyacencia:

Clase Grafo:
La lista es listaAdyacencia de tipo List de enteros;
Almacena directamente los índices de los vecinos.
Clase GrafoPonderado:
La lista es listaAdyacencia de tipo List de Aristas
No almacena índices, sino objetos de tipo *Arista*. Cada objeto *Arista* contiene el *destino* (el índice del vecino) y el *peso*.
Para obtener un vecino, el bucle debe acceder a los atrbutos del objeto *Arista*

2.4. Menú de usuario

En la clase PrincipalMenu.java la que contiene el método main.

Usa un while para mantener el menú hasta que el usuario elija la opción 5 (Salir). usa una estructura *switch* para ejecutar el código correspondiente con opción seleccionada.

Cada llamada crea un objeto de la clase Grafo o GrafoPonderado y lo añade a la lista correspondiente.

- **Case 1 y 2: Uso de la Clase Grafo (No Ponderado)**
  Estas opciones utilizan la clase Grafo.java.
  - **case 1:** Crear un grafo dirigido
    Crea un objeto de la clase Grafo.
    Se pasa true al constructor
    Llamada a Métodos:

grafoDir.agregarVertice(nomNodo): Llama al método para añadir un vértice a la lista vertices y crear una sublista vacía en listaAdyacencia.
grafoDir.agregarArista(origen, destino): Llama al método para añadir una arista, añade la arista origen ->destino. Como this.dirigido es true, no añade la arista inversa (destino ->origen)

- **case 2:** Crear un grafo NO dirigido
  Se crea un objeto de la clase Grafo con *false* en su constructor.
  Llamada a Métodos:
  grafoNoDir.agregarVertice(nomNodo): funciona igual que en el caso 1. *grafoNoDir.agregarArista(origen, destino)*: Llama al mismo método. Pero el método añade ambas aristas: origen ->destino y destino ->origen porque no es dirigido

- **Case 3 y 4: Uso de la Clase GrafoPonderado (Ponderado)** Estas opciones utilizan las clases GrafoPonderado.java y Arista.java

- **case 3:** Crear un grafo ponderado
  Se crea un objeto de la clase GrafoPonderado, se pasa false al constructor
  Llamada a Métodos:
  grafoPondNoDir.agregarVertice(nomNodo): la listaAdyacencia de esta clase es una List<List<Arista. **grafoPondNoDir.agregarArista(origen, destino, peso):** Llama al método de GrafoPonderado, dentro de él se crea un objeto Arista y se le pasa omo atributos los pedidos en el scanner.

- **case 4:** Crear un grafo ponderado dirigido
  Se crea un objeto GrafoPonderado con true en el constructor.
  Llamada a Métodos:
  grafoPondDir.agregarVertice(nomNodo): como en el caso 3.
  grafoPondDir.agregarArista(origen, destino, peso): llama al método, y crea y añade el objeto Arista a la lista del origen, pero como es dirigido, no se añade del destino al origen.

case 5: Salir
Muestra un mensaje y con la verificación del *while* sale del bucle y termina la ejecución.

default:
Muestra un mensaje de error pero no sale del bucle porque no fue el caso 5, sólo se vuelve a imprimir el menú y solicita otra opción.

2.5. Resultados

```
---Menu de grafos---
Opciones:
1. Crear un grafo dirigido
2. Crear un grafo NO dirigido
3. Crear un grafo ponderado
4. Crear un grafo ponderado dirigido
5. Salir
Ingresa la opción que deseas:
1
--- Grafo Dirigido (No Ponderado) ---
De cuantos nodos será tu grafo?
3
Ingresa el nombre del 1 nodo
A
Ingresa el nombre del 2 nodo
B
Ingresa el nombre del 3 nodo
C
cuantas aristas deseas agregar?
2
Arista 1
origen: A
destino: B
B
Arista 2
origen: C
destino: B

Grafo Dirigido creado:
A -> B
B ->
C -> B
```

Figura 4: Caso 1. Grafo dirigido

```
---Menu de grafos---
Opciones:
1. Crear un grafo dirigido
2. Crear un grafo NO dirigido
3. Crear un grafo ponderado
4. Crear un grafo ponderado dirigido
5. Salir
Ingresa la opción que deseas:
2
--- Grafo NO Dirigido (No Ponderado) ---
De cuántos nodos será tu grafo?
3
Ingresa el nombre del 1 nodo:
X
Ingresa el nombre del 2 nodo:
Y
Ingresa el nombre del 3 nodo:
Z
¿Cuántas aristas deseas agregar?
2
Arista 1
Vértice 1: X
Vértice 2: Y
Arista 2
Vértice 1: Y
Vértice 2: Z

Grafo NO Dirigido creado:
X -> Y
Y -> X Z
Z -> Y
```

Figura 5: Caso 2. Grafo no dirigido

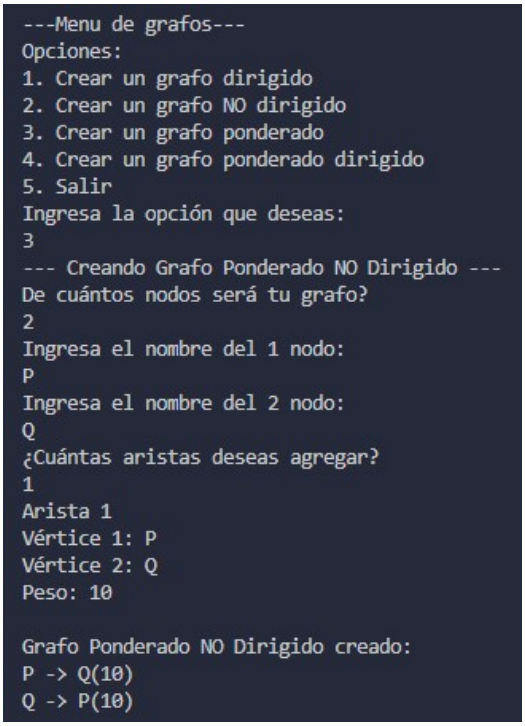


Figura 6: Caso 3. Grafo ponderado

```
---Menu de grafos---
Opciones:
1. Crear un grafo dirigido
2. Crear un grafo NO dirigido
3. Crear un grafo ponderado
4. Crear un grafo ponderado dirigido
5. Salir
Ingresa la opción que deseas:
4
--- Grafo Ponderado Dirigido ---
De cuántos nodos será tu grafo?
2
Ingresa el nombre del 1 nodo:
M
Ingresa el nombre del 2 nodo:
N
¿Cuántas aristas deseas agregar?
1
Arista 1
Origen: M
Destino: N
Peso: 7

Grafo Ponderado Dirigido creado:
M -> N(7)
N ->
```

Figura 7: Caso 4. Grafo ponderado dirigido

```
---Menu de grafos---
Opciones:
1. Crear un grafo dirigido
2. Crear un grafo NO dirigido
3. Crear un grafo ponderado
4. Crear un grafo ponderado dirigido
5. Salir
Ingresa la opción que deseas:
9
Opción no válida

---Menu de grafos---
```

Figura 8: Caso default

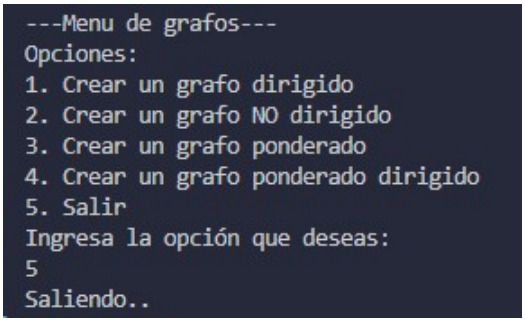


Figura 9: Caso 5. Salir

3. Conclusiones

Los objetivos de la práctica fueron cumplidos ya que pude ver cómo se implementan los grafos en un programa, que fue una de las dudas que me surgió cuando vimos el tema en la clase teorica, ya que yo había pensado si se representaban en una lista o algo así, pero igual pensé que no podía ser porque como vimos en clase, los grafos son estructuras no lineales y no jerárquicas; pero si noté que Java da la facilidad de tratar los nodos como un objetco con atributos y métodos, entonces es mucho más facil verlo de esa manera, también comprendí cómo se emplean los algoritmos BFS y DFS en los grafos y cómo se comportan ya que pude ver cómo no son funcionales para los grafos dirigidos ya que no es posible llegar a los nodos qque sólo son de SALIDA.

4. Referencias