



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:* TISTA GARCÍA EDGAR

*Asignatura:* ESTRUCTURAS DE DATOS Y ALGORITMOS II

*Grupo:* 5

*No de Práctica(s):* 06 -07

*Integrante(s):* GÓMEZ LUNA ALEJANDRO  
ANDRADE LÓPEZ LESLY

*No. de Equipo de  
cómputo empleado:* 49,50

*No. de Lista o Brigada:* 16

*Semestre:* 2020-1

*Fecha de entrega:* 24/SEPTIEMBRE/2019

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_

## Objetivo de la práctica

EL ESTUDIANTE CONOCERÁ LAS FORMAS DE REPRESENTAR UN GRAFO E IDENTIFICARÁ LAS CARACTERÍSTICAS NECESARIAS PARA COMPRENDER EL ALGORITMO DE BÚSQUEDA POR EXPANSIÓN.

### Prepractica (Alejandro)

- a) Ejercicios de la guía: En la guía se presentaba el antecedente de cómo surgieron los grafos, los problemas en los que se aplican los grafos y las definiciones importantes para poder entender bien el concepto de grafo.

Además, se proporcionaban conceptos clave de la programación orientada a objetos en el lenguaje de programación Python. Esto es importante porque la implementación de los grafos en Python se realizó a través de listas de adyacencia y con un enfoque orientado a objetos. Lo que cabe destacar de Python, es el uso de la palabra reservada `self`, ya que, a grandes rasgos, es necesaria para poder identificar qué objeto llamado a ciertos métodos y/o modifica ciertos atributos.

La forma en cómo se representaron los grafos para esta práctica fue mediante lista de adyacencia.

Para la primera actividad se definieron la clase `Vértice`, `Grafo` y `Controladora`.

En la clase `Vértice` encontramos su constructor al que se le pasa como parámetro el nombre del vértice y se crea una lista vacía. La lista representa los vértices que se pueden visitar desde este vértice. De igual manera, encontramos el método `agregar Vecino()`, el cual será usado por la clase `Grafo` para poder añadir aquellos vértices próximos al vértice actual en su lista de vecinos, siempre verificando que dichos vértices no hayan sido agregados con anterioridad.

Por otro lado, para la clase `Grafo`, que modela grafos no dirigidos, encontramos tres métodos: `agregar Vertice()`, `agregar Arista()` e `imprime Grafo()` y un miembro llamado `vértices`, en el cual se almacenarán los vértices existentes en el grafo. Es importante remarcar que este miembro `vértices` es un diccionario, donde la clave es el nombre del vértice y el valor es la dirección en memoria. En cuanto a los métodos:

- `agregarVertice()`: Toma como parámetro el vértice que se añadirá al grafo. Antes de ser añadido, se verifica si no se encuentra con anterioridad. Si se cumple esta condición entonces se agrega al atributo `vértices`.
- `agregarArista()`: Toma como parámetros el nombre de dos vértices, pues se pretende que este método una dichos nodos con una arista. Primero, verifica si dichos vértices se encuentran en el diccionario de vértices, para después ir comparando cada llave del diccionario con estos vértices. Cuando se encuentra la clave que coincide con el primer vértice, a la clave se le añade el segundo vértice. Cuando la clave coincide con el segundo vértice, a la clave se le añade el primer vértice. Para realizar esto, se usa la dirección en memoria que almacena el diccionario para el nombre de cada vértice en el grafo, con la finalidad de que los resultados se vean reflejados verdaderamente en cada vértice.
- `imprimeGrafo()`: Imprime cada vértice en el grafo como una lista de adyacencia, es decir, para cada vértice, se imprimen sus nodos vecinos.

Por último, la clase Controladora sirve para crear un objeto de la clase grafo y añadirle sus respectivos elementos como vértices y aristas. Al final, se imprime el grafo, mostrando en pantalla lo siguiente:

```
Vertice A Sus vecinos son['B', 'E']
Vertice B Sus vecinos son['A', 'F']
Vertice C Sus vecinos son['G']
Vertice D Sus vecinos son['E', 'H']
Vertice E Sus vecinos son['A', 'D', 'H']
Vertice F Sus vecinos son['B', 'G', 'I', 'J']
Vertice G Sus vecinos son['C', 'F', 'J']
Vertice H Sus vecinos son['D', 'E']
Vertice I Sus vecinos son['F']
Vertice J Sus vecinos son['F', 'G']
```

Para la segunda actividad, se modificó la clase Vertice. En este caso, el constructor de la clase Vertice ahora inicializa nuevos atributos, como es la distancia, color y predecesor.

Para la clase Grafo, se agregó el método bsf() y se modificó el método imprimeGrafo(). El método bfs() que significa Breadth-first search, sirve para que, dado un vértice inicial, se busquen todos los vértices a los que se puede llegar desde dicho vértice inicial, así como la distancia que existe a partir del vértice inicial con los demás en el grafo. Para realizarlo, se usan colores, blanco significa que no ha sido visitado el vértice, gris que está próximo a un nodo visitado y negro que ese nodo ya ha sido visitado. A grandes rasgos, se empieza por el vértice inicial y, con ayuda de una cola auxiliar, se adicionan aquellos vértices adyacentes a él. Ya que se terminó de explorar un vértice, se desencola un vértice de la cola auxiliar y ahora, se encolan los vértices adyacentes a este. Para cada vértice nuevo visitado, su predecesor es el vértice en el que nos encontramos. Se continúa este proceso hasta terminar de explorar el grafo.

En cuanto a el método imprimeGrafo(), lo único que se modificó fue que, además de imprimir los vértices adyacentes a cada vértice, se muestran las distancias entre vértices, las cuales se modifican una vez que se usa el método bfs() con respecto a un vértice.

Para este caso, observamos en la captura de pantalla, el valor de las distancias en el grafo antes de bfs() y después bfs(), el cual se realiza a partir del nodo A.

Debido a que las distancias se inicializan con un valor de 9999 para cada vértice, es que podemos observar el mismo valor de distancia para todos los vértices antes de aplicar el método bfs(). Una vez que se aplica este método, observamos que a este valor por defecto 9999 se va incrementando en uno dependiendo del vértice a partir del cual se realiza bfs() y que tan «lejos» se encuentre de este vértice inicial.

<p>Antes de la búsqueda</p> <p>Vertice A Sus vecinos son ['B', 'E']</p> <p>Distancia de A a A es: 9999</p> <p>Vertice B Sus vecinos son ['A', 'F']</p> <p>Distancia de A a B es: 9999</p> <p>Vertice C Sus vecinos son ['G']</p> <p>Distancia de A a C es: 9999</p> <p>Vertice D Sus vecinos son ['E', 'H']</p> <p>Distancia de A a D es: 9999</p> <p>Vertice E Sus vecinos son ['A', 'D', 'H']</p> <p>Distancia de A a E es: 9999</p> <p>Vertice F Sus vecinos son ['B', 'G', 'I', 'J']</p> <p>Distancia de A a F es: 9999</p> <p>Vertice G Sus vecinos son ['C', 'F', 'J']</p> <p>Distancia de A a G es: 9999</p> <p>Vertice H Sus vecinos son ['D', 'E']</p> <p>Distancia de A a H es: 9999</p> <p>Vertice I Sus vecinos son ['F']</p> <p>Distancia de A a I es: 9999</p> <p>Vertice J Sus vecinos son ['F', 'G']</p> <p>Distancia de A a J es: 9999</p>	<p>Despues de la búsqueda a partir de A</p> <p>Vertice A Sus vecinos son ['B', 'E']</p> <p>Distancia de A a A es: 9999</p> <p>Vertice B Sus vecinos son ['A', 'F']</p> <p>Distancia de A a B es: 10000</p> <p>Vertice C Sus vecinos son ['G']</p> <p>Distancia de A a C es: 10003</p> <p>Vertice D Sus vecinos son ['E', 'H']</p> <p>Distancia de A a D es: 10001</p> <p>Vertice E Sus vecinos son ['A', 'D', 'H']</p> <p>Distancia de A a E es: 10000</p> <p>Vertice F Sus vecinos son ['B', 'G', 'I', 'J']</p> <p>Distancia de A a F es: 10001</p> <p>Vertice G Sus vecinos son ['C', 'F', 'J']</p> <p>Distancia de A a G es: 10002</p> <p>Vertice H Sus vecinos son ['D', 'E']</p> <p>Distancia de A a H es: 10001</p> <p>Vertice I Sus vecinos son ['F']</p> <p>Distancia de A a I es: 10002</p> <p>Vertice J Sus vecinos son ['F', 'G']</p> <p>Distancia de A a J es: 10002</p>
---	--

## Prepractica (Lesly)

### Ejercicios del manual de prácticas:

- **Ejercicio 1:** Es proporcionado el código de las clases Grafo y Vértice, mediante estas clases, para implementarlas en otra clase llamada Controladora, es posible crear un Grafo instanciando la clase grafo y añadiendo esta instancia al momento de instanciar a Vértice. Se proporciona una lista las cuales contienen las aristas del grafo y mediante un for, son añadidas en el objeto grafo, finalmente se imprime el grafo.

```
Vertice A sus vecinos son: ['B', 'E']
Vertice B sus vecinos son: ['A', 'F']
Vertice C sus vecinos son: ['G']
Vertice D sus vecinos son: ['E', 'H']
Vertice E sus vecinos son: ['A', 'D', 'H']
Vertice F sus vecinos son: ['B', 'G', 'I', 'J']
Vertice G sus vecinos son: ['C', 'F', 'H']
Vertice H sus vecinos son: ['D', 'E', 'G']
Vertice I sus vecinos son: ['F']
Vertice J sus vecinos son: ['F']
```

- **Ejercicio 2:** Tomando como base el código anterior, se modificarán los atributos de la clase Vértice, agregando color, distancia y predecesor, además a la clase Grafo es agregado el método BFS, el cual nos brindará una búsqueda primero en anchura, esto quiere decir que al momento de buscar nodos, lo hará primero en una cierta distancia k y ya que haya terminado con esa distancia, procederá a la distancia k-1. El atributo color es utilizado, para que en el momento que se está haciendo la búsqueda de cada nodo, si se encuentra un nodo en blanco, este aún no ha sido revisado, se cambia el color de este a gris, posteriormente se calcula la distancia y ya terminada estas funciones, el nodo cambia su color a negro. Este procedimiento se repite hasta haber encontrado todos los nodos.

```

Vertice A sus vecinos son: ['B', 'E']
La distancia de A a A es 9999
Vertice B sus vecinos son: ['A', 'F']
La distancia de A a B es 9999
Vertice C sus vecinos son: ['G']
La distancia de A a C es 9999
Vertice D sus vecinos son: ['E', 'H']
La distancia de A a D es 9999
Vertice E sus vecinos son: ['A', 'D', 'H']
La distancia de A a E es 9999
Vertice F sus vecinos son: ['B', 'G', 'I', 'J']
La distancia de A a F es 9999
Vertice G sus vecinos son: ['C', 'F', 'H']
La distancia de A a G es 9999
Vertice H sus vecinos son: ['D', 'E', 'G']
La distancia de A a H es 9999
Vertice I sus vecinos son: ['F']
La distancia de A a I es 9999
Vertice J sus vecinos son: ['F']
La distancia de A a J es 9999

```

Para el método imprimirGrafo, se le ha añadido la impresión de la distancia del nodo principal, al nodo del cual se está midiendo la distancia.

#### Desarrollo

##### a) Ejercicios propuestos:

- a. Ejercicio 1: Para la codificación de los grafos en este ejercicio, se utilizaron las listas de adyacencia. Lo importante a resaltar en esta parte es que, dentro de la clase Graph, encontramos una lista ligada representada como un arreglo, pues se tienen los corchetes.  
En el constructor de la clase Graph, se inicializa cada posición de la lista ligada con otra lista ligada, es decir, una lista ligada de listas ligadas. Además, al constructor se le pasa como parámetro el total de vértices que tendrá el grafo. Respecto a los métodos, encontramos addEdge(), al cual se le pasan dos parámetros, los cuales son los nodos que se conectarán por una arista entre ellos. Por otro lado, está el método printGraph(), el cual, a través de un ciclo for, va recorriendo la lista modelada como arreglo y, en cada posición de dicha, se usa un for each para imprimir en pantalla cada elemento de la lista en esa posición. Esto simula como se conectan los vértices con sus respectivas aristas en el grafo.

```

Output - Practica6GomezAlejandro (run)
run:
Lista de Adyacencia del vertice 0
0 -> 1 -> 4

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

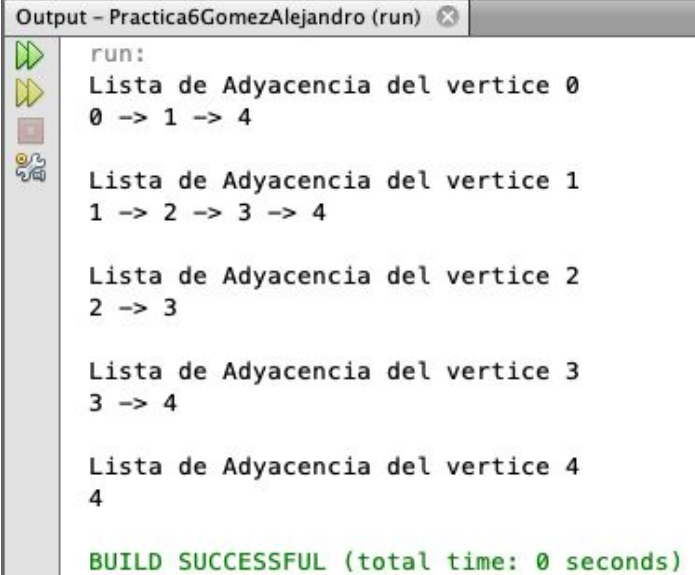
Lista de Adyacencia del vertice 2
2 -> 1 -> 3

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

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

```

Para la cuestión de los grafos dirigidos, basta con eliminar la segunda instrucción dentro del método `addEdge()`, esto debido a que solamente se agregará la arista dirigida desde el primer vértice al segundo y, no en sentido contrario.



```
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)
```

- b. Ejercicio 2: Se crea una clase llamada `WeightedGraph` en la cual el usuario podrá crear su grafo ponderado dirigido. En dicha clase, utilizamos herencia con la clase `Graph` para utilizar el constructor de esta, así como los atributos referentes al número de vértices y lista de adyacencia. Se ha declarado una tabla hash como atributo de esta clase, en donde se guardarán el valor del vértice del que parte hacia el que incide como clave (para ello se tomarán las claves como tipo `String`) y, en cuanto al valor correspondiente a cada clave, será el valor de la arista asignado por el usuario. De igual forma, se añaden las aristas en la lista de adyacencia.

Para imprimir el grafo, se utilizó el método `toString()` dentro de la colección `Hashtable`, para que todo se visualiza como un cadena y así imprimir en pantalla más fácilmente el valor de las aristas. Para las listas de adyacencia, se mostraron como en el método `printGraph()` de la clase padre.

Por último, se implementó un pequeño menú en donde el usuario fuera adicionando vértices hasta que decidiera salir.



```
Output - Practica6GomezAlejandro (run) X
Ingrese la cantidad de vertices que tendra el grafo
5
Ingrese la opcion a realizar:
1)Crear grafo dirigido
2)Crear grafo no dirigido
3)Crear grafo ponderado diridigo
4)Salir
3
Ingrese el nodo inicial:
0
Ingrese el nodo final:
1
Ingrese el valor de la arista que los une:
56
Si desea salir del programa oprima 1
0
Ingrese el nodo inicial:
4
Ingrese el nodo final:
1
Ingrese el valor de la arista que los une:
89
Si desea salir del programa oprima 1
0
Ingrese el nodo inicial:
3
Ingrese el nodo final:
2
Ingrese el valor de la arista que los une:
88
```

```
Output - Practica6GomezAlejandro (run) X
Si desea salir del programa oprima 1
0
Ingrese el nodo inicial:
2
Ingrese el nodo final:
0
Ingrese el valor de la arista que los une:
99
Si desea salir del programa oprima 1
1
Lista de Adyacencia del vertice 0
0 -> 1

Lista de Adyacencia del vertice 1
1

Lista de Adyacencia del vertice 2
2 -> 0

Lista de Adyacencia del vertice 3
3 -> 2

Lista de Adyacencia del vertice 4
4 -> 1

Valor de las aristas:
{2 a 0 =99, 4 a 1 =89, 3 a 2 =88, 0 a 1 =56}
```

- c. Ejercicio 3: Para este ejercicio, se creó una clase llamada GrafosMatrices. Esta sigue básicamente el principio de la clase Graph, con algunas diferencias. La primera de ellas es que se utilizará una matriz de dimensión  $V \times V$ , donde  $V$  es el número de vértices que tiene el grafo. Posteriormente, los métodos de añadir una arista, tanto dirigida como no dirigida. Para añadir una arista no dirigida, solamente se le asigna el valor de uno a la intersección entre el valor del primer vértice a manera de renglón con el valor del segundo vértice a manera de columna, y después se repite este proceso intercambiando el valor de la columna con el del renglón. En cuanto a la arista dirigida, se realiza el proceso descrito anteriormente con la diferencia de, sin intercambiar el valor de la columna con el del renglón.
- Para imprimir este tipo de representación, se utilizaron dos for, uno anidado del otro para recorrer en cada fila sus columnas.
- Se utilizaron los vértices proporcionados a manera de ejemplo en el principio de la práctica, con la finalidad de apreciar mejor el cambio de representación. Los resultados en pantalla para aristas dirigidas como para aristas no dirigidas son los siguientes, respectivamente:

```
Output - Practica6GomezAlejandro (run) X
run:
Matriz de Adyacencia:
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
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
Output - Practica6GomezAlejandro (run) X
run:
Matriz de Adyacencia:
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
BUILD SUCCESSFUL (total time: 0 seconds)
```

- d. Ejercicio 4: Es proporcionado el método para la implementación de búsqueda por anchura o expansión (BFS), dicho método debe agregarse a la clase Graph. La funcionalidad que cumple este método comienza al momento de ser seleccionado un nodo de un grafo. Inicialmente se necesita un arreglo de la cantidad de los vértices del grafo y una lista ligada. El arreglo es declarado del tipo booleano, puesto que servirá para llevar un control de cuáles nodos ya han sido visitados y cuáles no. Como el arreglo tiene el tamaño del número de vértices, se tendrá en cuenta que cada posición del arreglo le pertenece a cada vector. Se comienza en la posición del nodo que ha sido elegido y como es el primero que se está visitando, se marca como verdadero, de la misma manera el nodo se introduce a la lista (se encola). Las siguientes instrucciones son llevadas a cabo, mientras que la lista no se encuentra vacía. Se desencola el valor correspondiente de la lista y se imprime. A continuación se usa el método iterator() con el atributo adjArray[ ] de la clase Graph y se comprobará que mientras haya aún valores consecutivos en este arreglo, se comprobará que el nodo que se está visitando no ha sido visitado, si este es el caso, se llevará la comprobación del while previamente declarado, en caso contrario se marcará ese nodo ahora como visitado y será añadido a la lista. Estas acciones se seguirán repitiendo hasta que ya no existan valores en la lista creada. La estructura de resolución del algoritmo lleva la misma secuencia que el algoritmo visto en clase, a diferencia que en el algoritmo de clase se



implementan 2 listas (pues en el algoritmo de la práctica se implementa un arreglo de tipo booleano) . Como en un algoritmo se implementa una lista y en otro un arreglo, la diferencia se nota cuando en una, es necesario revisar si el valor de una lista está en la otra, si ya se encuentra, no se escribe, pero si o, se introduce valor. Mientras que en el arreglo se comprueba si está ese dato si en la posición del nodo se encuentra marcado como true o false.

Para implementar este algoritmo en la clase principal, cree un método en la clase Graph llamada UsBFS, en la que se pregunta al usuario si desea implementar la búsqueda por expansión, si su respuesta es verdadera, se le solicitará el nodo por el cual desea empezar y comenzará la búsqueda, en caso contrario no se efectuará nada y se regresará al menú donde contiene las formas de crear el grafo.

```

EJEMPLO:
Ingrese la cantidad de vertices que tendra el grafo
5
Ingrese la opcion a realizar:
1)Crear grafo dirigido
2)Crear grafo no dirigido
3)Crear grafo ponderado dirigido
4)Salir
1
Ingrese el nodo inicial:
2
Ingrese el nodo final:
3
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
4
Ingrese el nodo final:
3
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
2
Ingrese el nodo final:
1
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
4
Ingrese el nodo final:
3
Si desea salir del programa oprima 1
3
Ingrese el nodo inicial:
2
Ingrese el nodo final:
1
Ingrese el nodo inicial:
2
Ingrese el nodo final:
1
Si desea salir del programa oprima 1
1
Lista de Adyacencia del vertice 0
0

Ingrese el nodo inicial:
1
Ingrese el nodo final:
2
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
1
Ingrese el nodo final:
2
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
1
Ingrese el nodo final:
2
Si desea salir del programa oprima 1
1
Lista de Adyacencia del vertice 0
0

```

```

1
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
1
Ingrese el nodo final:
4
Si desea salir del programa oprima 1
2
Ingrese el nodo inicial:
2
Ingrese el nodo final:
4
Si desea salir del programa oprima 1
1
Lista de Adyacencia del vertice 0
0

Lista de Adyacencia del vertice 1
1 -> 4

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

Lista de Adyacencia del vertice 3
3

Lista de Adyacencia del vertice 4
4 -> 3

Desea implementar la busqueda por anchura (BFS)
1. Si
2. No
1
Dame el nodo inicial
2
2 3 1 4 Ingrese la opcion a realizar:
1)Crear grafo dirigido

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

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

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

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

Desea implementar la busqueda por anchura (BFS)
1. Si
2. No
1
Dame el nodo inicial
1
1 4 3 2 Ingrese la opcion a realizar:
1)Crear grafo dirigido
2)Crear grafo no dirigido
3)Crear grafo ponderado dirigido
4)Salir
1

```

## Conclusiones

- **Alejandro:** En esta práctica se lograron resolver los ejercicios propuestos, así como analizar las principales características de los grafos y las maneras en cómo se pueden representar. Además de esto, se pudo comprender la manera en cómo recorrer un grafo a partir de un vértice mediante el uso del método Breadth-Depth Search. A grandes rasgos, este método hace uso de una cola para ir explorando los diversos vértices en el grafo, en donde, cada que se descubre un nuevo vértice se adiciona a la cola provocando que primero se exploren todos aquellos vértices que se vayan encontrando.

Por otro lado, dentro de las características de los grafos, podemos encontrar la presencia de los vértices y las aristas. Los vértices contendrán un cierto valor, mientras que las aristas se encargará de conectar a los vértices en el grafo. Dependiendo de la relación entre los vértices, el grafo será de cierto tipo. Si las aristas tienen un valor, el grafo es ponderado. Si las aristas tienen una dirección, el grafo será direccionado. Si las aristas no tienen dirección, el grafo será no direccionado. Entre estos casos, pueden existir diversas combinaciones.

Los grafos son una herramienta bastante útil, pues nos permiten de cierta manera jerarquizar cierto tipo de información y observar como se relacionan los elementos dentro de ella, dando como resultado la resolución de diversas problemáticas, como aquel conocido problema del viajero.

Se logró cumplir el objetivo, porque ahora se tiene una noción buena de las formas en como funciona un grafo, sus características, dos formas en como representarlo y una forma en como se pueden recorrer todos los vértices de dicho.

La práctica nos permite cumplir el objetivo propuesto, además, nos permite analizar otra manera en como visualizar los grafos sin usar una lista de adyacencia, la cual es mediante su matriz de adyacencia. Una buena práctica para poder profundizar mejor en el uso de grafos.

**Lesly:** En esta práctica se lleva a cabo la primera implementación con código de los grafos, en este caso se implementaron grados dirigidos, no dirigidos y ponderados dirigidos. Al momento de crear un grafo, es necesario saber cuántos vértices contendrá, para que posteriormente se implementen la cantidad de aristas que se requieren para conectar los vértices. Al momento de compilar el algoritmo de grafo no dirigido, pude comprender mucho mejor la teoría vista en clase, gracias a ello pude aportar información a mi compañero para realizar las actividades siguientes de la práctica. La parte que llamó mi atención es la creación de la matriz de adyacencia, pues teniendo en cuenta el número de vértices, es fácil construir la matriz donde indicará la cantidad de aristas que unen al vértice.

## Practica 7

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

### Pre Práctica (Alejandro)

- Ejercicios de la guía: Para esta práctica, se nos pidió realizar otro tipo de «búsqueda» en el grafo, la cual es Depth First Search. A comparación de BFS, este tipo de recorrido del grafo utiliza la recursividad para recorrer el grafo. Otra forma de verlo o de realizarlo es mediante una estructura de pila. Debido a la utilización de la recursividad o la pila, siempre que se vayan descubriendo nuevos vértices en el grafo, estos se dejarán para una exploración posterior, por lo que solo nos iremos centrando en aquellos vértices que acaban de ser descubiertos, provocando que se recorra todo el grafo mediante diversos caminos, donde, cada que ya no se pueda continuar recorriendo un camino, se retrocederá hasta el vértice que tenga otro camino por recorrer y se realizará el procedimiento anterior.

Se nos presentó la implementación de DFS en el lenguaje de programación Python, donde se implementa mediante los métodos `bfs()` y `dfsVisitar()` dentro de la clase `Grafo`. El método `dfs()` simplemente llama al método `dfsVisitar()` para todos los nodos que aún no han sido visitados. Para ir marcando aquellos nodos que ya han sido visitados, se utiliza la misma nomenclatura que en la implementación de BFS, la cual es modificando los atributos de color en cada vértice, variando entre blanco, gris y negro.

En cuanto al método `dfsVisitar()`, vemos que es de tipo recursivo, y que, conforme se van visitando nuevos vértices, se modifica una variable global que lleva la cuenta de en que momento se descubrieron por primera vez cada vértice y cuanto tiempo se ha terminado en recorrer todos sus vértices vecinos. La función se vuelve a llamar siempre que, estando en un vértice, uno de sus nodos vecinos aún no ha sido visitado.

En cuanto a la clase controladora, se creó un grafo igual al del ejemplo usado en la guía, por lo que, se observan los resultados obtenidos en la guía.

```
class Controladora:
    def main(self):
        g=Grafo()
        a=Vertice('u')
        g.agregarVertice(a)

        for i in range(ord('u'),ord('{')):
            g.agregarVertice(Vertice(chr(i)))

        g.agregarArista('u','v');
        g.agregarArista('u','x');
        g.agregarArista('v','y');
        g.agregarArista('w','y');
        g.agregarArista('w','z');
        g.agregarArista('x','v');
        g.agregarArista('y','x');
        g.agregarArista('z','z');
        print("***Antes de realizar DFS***")
        g.imprimeGrafo()

        print("***Despues de realizar DFS***")
        g.dfs('u')
        g.imprimeGrafo()

obj=Controladora()
obj.main()
```

```
***Antes de realizar DFS***
Vertice u
Descubierto/Termino:0/0
Vertice v
Descubierto/Termino:0/0
Vertice w
Descubierto/Termino:0/0
Vertice x
Descubierto/Termino:0/0
Vertice y
Descubierto/Termino:0/0
Vertice z
Descubierto/Termino:0/0
***Despues de realizar DFS***
Vertice u
Descubierto/Termino:1/8
Vertice v
Descubierto/Termino:2/7
Vertice w
Descubierto/Termino:9/12
Vertice x
Descubierto/Termino:4/5
Vertice y
Descubierto/Termino:3/6
Vertice z
Descubierto/Termino:10/11
```

## Pre Práctica (Lesly)

- **Ejercicio 1:** Se proporcionan los códigos para llevar la implementación de la búsqueda por profundidad de un grafo (DFS). La diferencia entre este algoritmo y el implementado en la práctica anterior (BFS) es que en este algoritmo se seguirá recorriendo la cantidad necesarios de nodos enlazados, este recorrido se detiene cuando ya no hay un nodo que visitar. Puede llegarse al caso en que un nodo, se encuentre enlazado con 2 o más nodos, puede verse esto como si se tuviesen más de un camino por visitar, por lo que se implementa “backtrack”, se lleva a cabo de la siguiente manera; Al momento de llegar a un nodo que ya no tenga un nodo enlazado independientemente de los que ya han sido visitados, se debe retroceder y verificar si en el nodo al que se ha retrocedido contiene otro nodo enlazado, si esto se cumple se procederá a visitar el nodo, en caso contrario, se volverá a retroceder un nodo y se verificará de nuevo la existencia de un nodo que no se haya visitado anteriormente. Es por ello que le llaman búsqueda profunda, pues primero se busca la mayor cantidad de nodos que están entrelazados y ya que esto se haya completado, procede a buscar los posibles nodos que aún no hayan sido visitado.

Lo Mencionado anteriormente solo ha sido enfocado al momento de búsqueda de los nodos, pero no se ha hablado de donde se guardan los nodos, o como es que se lleva a cabo el conteo de los nodos que han sido visitados y de los que no. Al momento de implementar este algoritmo, se deben considerar 1 lista( que tendrá función de tipo pila) donde contiene los nodos creados con anterioridad, color, así como del tiempo. La referencia del tiempo, es referida a que se le asignará un valor según el número de nodo que haya sido encontrado, pero también debe de tenerse una referencia final de tiempo, pues así se tendrá mayor certeza del recorrido. Para el color del nodo, como se presentó en la práctica pasada, es a causa de cuando se crea el nodo se toma color inicial blanco, si es recorrido por primera vez, se cambia el color a gris, pero si ya no se revisará más, será asignado el color negro, por lo que significa que ya no es necesario revisarlo, pues la labor con ese nodo ha terminado. Cada vez que un nodo es visitado por primera vez, este es introducido a la lista de vértices donde se guardarán los datos.

Como en la práctica no venía implementado el método main, me guie de la práctica pasada creando una clase llamada controladora, así que primero creo un objeto de tipo Grafo, posteriormente uno de tipo vértice tomando como principal la letra 'A'. Para las aristas del grafo, considere las implementadas en la práctica pasada, ya que son agregados los vértices y las aristas al grafo, se implementa el método dfs, para que finalmente se imprimen los resultados con el método imprimir Grafo().

```
        if self.vertices[u].color=='white':
            self.dfsVisitar(self.vertices[u])
    def dfsVisitar(self,vert):
        global tiempo
        tiempo=tiempo+1
        vert.d=tiempo
        vert.color='gris'
        for v in vert.vecinos:
            if self.vertices[v].color=='white':
                self.vertices[v].pred=vert
                self.dfsVisitar(self.vertices[v])
        vert.color='black'
        tiempo=tiempo+1
        vert.f=tiempo
class Controladora:
    def main(self):
        g=Grafo()
        a=Vertice('F')
        g.agregarVertice(a)
        for i in range(ord('A'),ord('K')):
            g.agregarVertice(Vertice(chr(i)))
        edges=['AB','AE','BF','CG','DE','DH','EH','FG','FI','FJ','GJ']
        for edge in edges:
            g.agregarArista(edge[:1],edge[1:])
        g.dfs(a)
        g.imprimeGrafo()
obj=Controladora()
obj.main()
```

## Ejercicios del laboratorio:

### • Ejercicio 1:

#### NODO 5

```

DFSutil(5, visited [])
visited [5] =true
i= [ 1,4, 7]
n=1
visited [1] !=true *
    DFSutil(1,visited[])
    i=[5]
    n=5
    visited [5] !=true **
    (termina la recursividad)

n=4
visited[4] !=true *
    DFSutil(4, visited [])
    visited [4] = true
    i= [ 0,5,7]
    n=0
    visited [0] !=true *
        DFSutil (0, visited [])
        visited [0] =true
        i= [2,4,6]
        n=2
        visited [2] !=true *
            DFSutil(2,visited[])
            visited[2]=true
            i=[0]
            n=0
            visited[0] !=true **
            (termina la recursividad)

        N=4
        visited [4] !=true **
        n=6
        visited [6] !=true *
            DFSutil (6, visited [])
            visited [6] =true
            i= [0]
            n=0
            visited[0] !=true **
            (termina la recursividad)

    n=5
    visited [5] !=true *
        DFSutil (5, visited [])
        visited [5] =true
        i= [1,4,7]
        n=1
        visited [1] !=true **
        n=4
        visited [4] !=true **
        n=7
        visited [7] !=true *
            DFSutil (7, visited [])
            visited [7] =true
            i= [3,4,5]
            n=3
            visited[3] !=true *
                DFSutil (3, visited [])
                visited [3] =true
                i= [7]
                n=7
                visited [7] !=true **
                (termina la recursividad)

            n=4
            visited[4] !=true **
            n=5
            visited[5] !=true **
            (termina la recursividad)

        n=7
        visited [7] !=true **
        (termina la recursividad)

n=7
visited [7] !=true **
(termina el programa)

```

\*\*Se imprime por pantalla\*\*

5 1 4 0 2 6 7 3

#### NODO 3

```

DFSutil(3, visited [])
visited [3] =true
i= [7]
n=7
visited [7] !=true *
    DFSutil(7,visited[])
    i=[3,4,5]
    n=3
    visited [3] !=true **
    n=4
    visited[4] !=true *
        DFSutil(4, visited [])
        visited [4] = true
        i= [ 0,5,7]
        n=0
        visited [0] !=true *
            DFSutil (0, visited [])
            visited [0] =true
            i= [2,4,6]
            n=2
            visited [2] !=true *
                DFSutil(2,visited[])
                visited[2]=true
                i=[0]
                n=0
                visited[0] !=true **
                (termina la recursividad)

            n=4
            visited [4] !=true **
            n=6
            visited [6] !=true *
                DFSutil (6, visited [])
                visited [6] =true
                i= [0]
                n=0
                visited[0] !=true **
                (termina la recursividad)

        n=5
        visited [5] !=true *
            DFSutil (5, visited [])
            visited [5] =true
            i= [1,4,7]
            n=1
            visited [1] !=true *
                DFSutil (1, visited [])
                visited [1] =true
                i= [5]
                n=5
                visited[5] !=true **
                (termina la recursividad)

            n=4
            visited [4] !=true **
            n=7
            visited [7] !=true **
            (termina la recursividad)

        N=7
        visited [7] !=true **
        (termina la recursividad)

    N=5
    visited [5] !=true **
    (termina la recursividad)

```

\*\*Se imprime por pantalla\*\*

3 7 4 0 2 6 5 1

```

NODO 4
DFSutil(4, visited [])
visited [4] =true
i= [0,5,7]
n=0
visited [0] !=true *
    DFSutil(0,visited[])
    i=[2,4,6]
    n=2
    visited[2] !=true *
        DFSutil(2, visited[])
        visited [2] = true
        i= [ 0]
        n=0
        visited [0] !=true **
        (termina la recursividad)

    n=4
    visited [4] !=true **
    n=6
    visited [6] !=true *
        DFSutil (6, visited [])
        visited [6] =true
        i= [0]
        n=0
        visited [2] !=true **
        (termina la recursividad)

n=5
visited [5] !=true *
    DFSutil (5, visited [])
    visited [5] =true
    i= [1,4,7]
    n=1
    visited[1] !=true *
        DFSutil (1, visited [])
        visited [1] =true
        i= [5]
        n=5
        visited [5] !=true **

    n=4
    visited[4] !=true **
    n=7
    visited[7] !=true *
        DFSutil (7, visited [])
        visited [7] =true
        i= [3,4,5]
        n=3
        visited[3] !=true *
            DFSutil (3, visited [])
            visited [3] =true
            i= [7]
            n=7
            visited[7] !=true **
            (termina la recursividad)

        n=4
        visited[4] !=true **
        n=5
        visited[5] !=true **
        (termina la recursividad)

n=7
visited [7] !=true **
(termina el programa)

**Se imprime por pantalla**
4 0 2 6 5 1 7 3

```

**NOTA:** En las implementaciones manuales se observa que en algunas instrucciones se muestra \* o \*\* en diferentes casos. La causa de ello es que un \* significa que es verdadero o que cumple la condición, mientras que \*\* significa que es falso o que no cumple la condición.

```

UDF con arista 5
5 1 7 4 0 2 6 3
UDF con arista 3
3 7 5 1 4 0 2 6
DF con arista 4
4 5 1 7 3 0 2 6 B

```



La razón por la cual la impresión de pantalla anterior no cumple con la aplicación manual del algoritmo, se debe a que en la implementación manual se usó el criterio de ; primero los valores más pequeños y posteriormente los grandes. Mientras que el código toma los valores según sean introducidos al momento de enlazar un nodo con el otro. Ahora bien, teniendo cuidado de lo mencionado y conectando los nodos, de manera que primero se conectan los vértices con las aristas más pequeñas hasta llegar a los valores mayores, se tuvo el siguiente resultado:

UDF con arista 5	UDF con arista 3	DF con arista 4
5 1 4 0 6 2 7 3	3 7 4 0 2 6 5 1	4 0 2 6 5 1 7 3 E

La impresión de pantalla muestra que la implementación manual es correcta.

```
Dame el vertice con el que implementarás DFS
3
Lista de Adyacencia del vertice 0
0 -> 4 -> 2 -> 2

Lista de Adyacencia del vertice 1
1 -> 4 -> 5 -> 2 -> 3 -> 4 -> 5 -> 4

Lista de Adyacencia del vertice 2
2 -> 0 -> 0 -> 3 -> 1 -> 5 -> 4

Lista de Adyacencia del vertice 3
3 -> 2 -> 5 -> 1 -> 3 -> 3

Lista de Adyacencia del vertice 4
4 -> 0 -> 5 -> 1 -> 1 -> 1 -> 2

Lista de Adyacencia del vertice 5
5 -> 4 -> 1 -> 3 -> 1 -> 2

Busqueda DFS con nodo 3
3 2 0 4 5 1 BUILD SUCCESSFUL (total time: 36 sec)

~
3 2 0 4 5 1 Lista de Adyacencia del vertice 0
0 -> 4 -> 2 -> 2

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

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

Lista de Adyacencia del vertice 3
3 -> 2 -> 5 -> 2 -> 4

Lista de Adyacencia del vertice 4
4 -> 0 -> 5 -> 1 -> 5 -> 3 -> 2

Lista de Adyacencia del vertice 5
5 -> 4 -> 1 -> 3 -> 4 -> 1

UDF con arista 5
5 4 0 2 3 1

Dame el vertice con el que implementarás DFS
3
3 2 0 4 5 1 7 Lista de Adyacencia del vertice 0
0 -> 4 -> 2 -> 2 -> 7 -> 1

Lista de Adyacencia del vertice 1
1 -> 4 -> 5 -> 5 -> 0

Lista de Adyacencia del vertice 2
2 -> 0 -> 0 -> 3 -> 5 -> 3

Lista de Adyacencia del vertice 3
3 -> 2 -> 5 -> 7 -> 2

Lista de Adyacencia del vertice 4
4 -> 0 -> 5 -> 1 -> 7

Lista de Adyacencia del vertice 5
5 -> 4 -> 1 -> 3 -> 1 -> 2

Lista de Adyacencia del vertice 6
6

Lista de Adyacencia del vertice 7
7 -> 3 -> 0 -> 4

Lista de Adyacencia del vertice 8
8

UDF con arista 3
5 4 0 2 3 7 1 BUILD SUCCESSFUL (total time: 24 sec)
```

Se lleva a cabo la busque por profundidad DFS. Para ello es necesario tener en cuenta el número de vértices del grafo, pues así también se inicializará el tamaño del arreglo de tipo booleano que nos servirá para llevar un control de los nodos que han sido visitados, de los que no.

Cuando es seleccionado el nodo por el cual se iniciará, se imprime en pantalla principal y posteriormente se crea un iterador de donde se encuentran almacenadas la forma en que los nodos fueron guardados, como dicha lista, lleva el orden según fue llevada la conexión de los mismos, se deberá realizar lo siguiente mientras haya un sucesor del nodo que se ha elegido: Se contempla el número que aparece en el iterador y se verificará que no se haya visitado, si se cumple, se implementará recursividad ahora con el valor tomado del iterador, se marcará como visitado, se creará su lista iterator, en caso de que su lista sea vacía, se termina la recursividad y conforme al primer iterator creado, se implementará el segundo valor de esta. Por otro lado, si esto no se cumple, se volverá a realizar recursividad hasta que la lista del segundo valor (en dado caso que algunos de los valores de este nuevo iterador, tengan su propio iterator, deben implementar recursividad en ellos hasta que ya se hayan visitados dichos nodos). En algunas listas formadas por iterator, se muestra que se repiten algunos valores, pues existen diversas conexiones entre varias aristas, esto es solucionado gracias al arreglo booleano que se tiene, como ahí se va llevando el control de cuales nodos han sido visitados y cuales no, se evita una múltiple recursividad en el código.

La diferencia que se encuentra respecto al algoritmo visto en clase se nota en el arreglo booleano que es marcado como verdadero cuando el nodo ha sido visitado, mientras que en el algoritmo visto en clase usa una lista para llevar el control de los nodos que ya han sido visitados. Aunque en ella algoritmo no especifica cómo se lleva a cabo la forma de marcar los nodos que ya han sido visitados, podría considerarse que lo realiza de la misma manera que el arreglo booleano del algoritmo visto en el laboratorio. Además se menciona en el algoritmo visto en clase que se implementa una lista tipo LIFO, donde se van añadiendo cada nodo que ha sido visitado tomando a consideración primero las aristas de menor valor y cada vez que se ha llegado hasta el último nodo de una rama, son desencolados los nodos necesarios para visitar otra rama y así sucesivamente hasta que la lista queda vacía, esto indica que se han revisado todas las aristas.

- **Ejercicio 2:** Para este ejercicio, se nos pedía realizar la implementación del algoritmo de Prim, para un grafo ponderado dirigido. Para poder realizarlo, se empezó por crear un nuevo método dentro de la clase de WeightedGraph. En este método se crea un arreglo de booleans, el cual sirve para llegar el control de los vértices que ya han sido visitados.

Después, se crea una arrayList, donde se irán almacenando las aristas pertenecientes al árbol de expansión mínimo (minimum spanning tree).

El nodo inicial se marca como visitado en el arreglo de booleans y, se procede a crear un comparador y una cola de prioridad. El comparador nos permitirá decidir el criterio con el cual querremos ir almacenando las aristas vecinas a un vértice en la cola de prioridad. Para este caso, siempre que se vaya a agregar una nueva arista a la cola de prioridad, se verifica en la hashtable su valor asociado (recordando que la llave es el nombre de la arista) y así, determinar en qué posición de la cola de prioridad insertarlo. El comparador es una interfaz, en la cual se encuentra el método compare, el cual es sobrescrito para que realice el ordenamiento como nosotros le indiquemos.

Una vez creadas estas dos cuestiones, se procede a recorrer la lista de adyacencia del vértice inicial, e ir agregando estas aristas a la cola de prioridad.

Ahora, mientras la cola de prioridad no esté vacía, se irán desencolando las aristas y revisando el vértice al que se llega. Si no se encuentra marcado en el arreglo de booleans, se procede a agregarlo al m.s.t y a marcarlo como visitado. Además, se agregan todas las aristas del vértice al que se llega haciendo uso de su lista de adyacencia.

```
Output - Practica7 (run) x
Lista de Adyacencia del vertice 0
0 -> 1 -> 2 -> 3

Lista de Adyacencia del vertice 1
1 -> 3

Lista de Adyacencia del vertice 2
2 -> 3 -> 5

Lista de Adyacencia del vertice 3
3 -> 4

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

Lista de Adyacencia del vertice 5
5

Lista de Adyacencia del vertice 6
6

Valor de las aristas:
{03=2, 02=7, 01=2, 25=4, 23=5, 46=3, 42=2, 13=4, 34=1}

M.S.T: 0-1 0-3 3-4 4-2 4-6 2-5 BUILD SUCCESSFUL (total time: 0 seconds)
```

## Conclusiones

- **Alejandro:** De esta práctica, pudimos observar la nueva forma en cómo recorrer un grafo, la cual es a través de DFS (Depth-First Search), en donde, a grandes rasgos, primero vamos siguiendo un camino para ir explorando los vértices del grafo. Si se llega a un camino donde ya todos los vértices vecinos hayan sido visitados, entonces se va retrocediendo en el mismo camino hasta llegar a un punto donde aún existen vértices sin visitar. Esto se asemeja un poco a los algoritmos back-track, los cuales van recorriendo cierto camino hasta llegar a un punto donde ya no pueden avanzar más, entonces retroceden a un punto anterior en el mismo camino de donde volver a partir, repitiendo lo anterior hasta encontrar la solución de un problema.  
En el caso de DFS, se utiliza una pila de recursividad, mientras que en BFS se utiliza una cola, en esto radica la principal diferencia entre estos dos tipos de recorridos.  
Por otro lado, se implementó el algoritmo de Prim, el cual sirve para encontrar el árbol de expansión mínima de un grafo ponderado a partir de un vértice inicial. Para este caso, se realizó en un grafo ponderado y dirigido, sin embargo, el algoritmo no cambia si se trata de un grafo no dirigido, siempre y cuando sea ponderado.  
Es importante conocer las maneras más comunes de recorrer todos los nodos de un grafo, ya que nos puede ayudar a determinar si un grafo es conectado, además de que sirven como base para algoritmos como el de Prim, el cual utiliza una cola de prioridad.  
Se logró el objetivo, pues se observaron y analizaron las características de un grafo, además de las posibles variantes que existen, así como la forma en que se implementan en Java. De igual forma, se analizaron los dos tipos de búsqueda más comunes en grafos: DFS y BFS.  
Al igual en prácticas anteriores, se fue un poco más que el objetivo propuesto, ya que también se vio el algoritmo de Prim, lo que permite observar la importancia de la forma en cómo se recorre un grafo. Una práctica bastante interesante y un tanto difícil, por la parte que indica implementar el algoritmo de Prim, sin embargo, permite obtener un mayor aprendizaje en cuanto a los grafos ponderados.

- **Lesly:** Inicialmente la práctica tiene como objetivo implementar la búsqueda mediante el algoritmo DFS, que se basa en llegar hasta el último nodo conectado en una sucesión de aristas conectadas, como se menciona anteriormente, para llevar a cabo este algoritmo se usa la recursividad, donde permite revisar todos los nodos que se encuentran conectado al nodo que está siendo analizado. Es de suma importancia llevar un control de los nodos que han sido visitados, sino se generará un bucle infinito en el programa y nunca llegaría a la solución deseada. Aunque, también se llevó a cabo el algoritmo de Primm visto en clase, fue un tanto complicado plantear la forma de plasmar el algoritmo en java, pues se tenía que tener la consideración una cola de prioridad de los valores de la arista que enlazaba a cada una de las aristas. Además tener en cuenta que la cola de prioridad tenía que ser verificada en cada uno de los datos que se iban introduciendo, debido a que podría tenerse la situación de introducir el valor de un vértice mucho menor que los vértices que ya se encontraban en la cola de prioridad. Para el ejercicio 2 se permite hacer uso de los grafos ponderados, pues estos poseen un valor en su vértice, dicho valor fue tomado como el valor de prioridad para la cola.