	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

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: 17/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 como 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 que objeto llamado a ciertos métodos y/o modifica ciertos atributos.
   La forma en como 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 la 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 agregarVecino(), 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 Gráfo, que modela grafos no dirigidos, encontramos tres métodos: agregarVertice(), agregarArista() e imprimeGrafo() y un miembro llamado vertices, en el cual se almacenarán los vértices existentes n el grafo. Es importante remarcar que este miembro vertices 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 vertices.
   - 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 vertices, 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 el. Ya que se terminó de explorar un vértice, se desencola un vértice de la cola auxiliar y ahora, se encolarán 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.

```
Antes de la busqueda
Vertice A Sus vecinos son ['B', 'E']
Distancia de A a A es: 9999
Vertice B Sus vecinos son ['A', 'F']
Distancia de A a B es: 9999
Vertice C Sus vecinos son ['G']
Distancia de A a C es: 9999
Vertice D Sus vecinos son ['E', 'H']
Distancia de A a D es: 9999
Vertice E Sus vecinos son ['A', 'D', 'H']
Distancia de A a E es: 9999
Vertice F Sus vecinos son ['B', 'G', 'I', 'J']
Distancia de A a F es: 9999
Vertice G Sus vecinos son ['C', 'F', 'J']
Distancia de A a G es: 9999
Vertice H Sus vecinos son ['D', 'E']
Distancia de A a H es: 9999
Vertice I Sus vecinos son ['F']
Distancia de A a I es: 9999
Vertice J Sus vecinos son ['F', 'G']
Distancia de A a J es: 9999
```

```
Despues de la busqueda a partir de A
Vertice A Sus vecinos son ['B', 'E']
Distancia de A a A es: 9999
Vertice B Sus vecinos son ['A', 'F']
Distancia de A a B es: 10000
Vertice C Sus vecinos son ['G']
Distancia de A a C es: 10003
Vertice D Sus vecinos son ['E', 'H']
Distancia de A a D es: 10001
Vertice E Sus vecinos son ['A', 'D', 'H']
Distancia de A a E es: 10000
Vertice F Sus vecinos son ['B', 'G', 'I', 'J']
Distancia de A a F es: 10001
Vertice G Sus vecinos son ['C', 'F', 'J']
Distancia de A a G es: 10002
Vertice H Sus vecinos son ['D', 'E']
Distancia de A a H es: 10001
Vertice I Sus vecinos son ['F']
Distancia de A a I es: 10002
Vertice J Sus vecinos son ['F', 'G']
Distancia de A a J es: 10002
```

Para este caso, observamos en las 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.

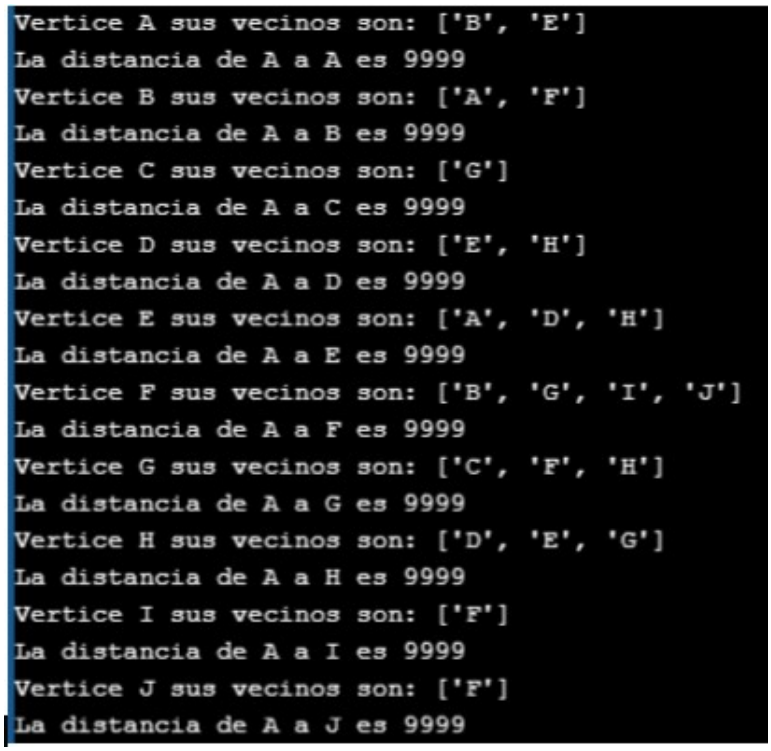
Prepractica (Lesly)

Ejercicios del manual de prácticas:

- Ejercicio 1: Es proporcionado el código de las clases Grafo y Vértice, mediante estas clase, 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']
```

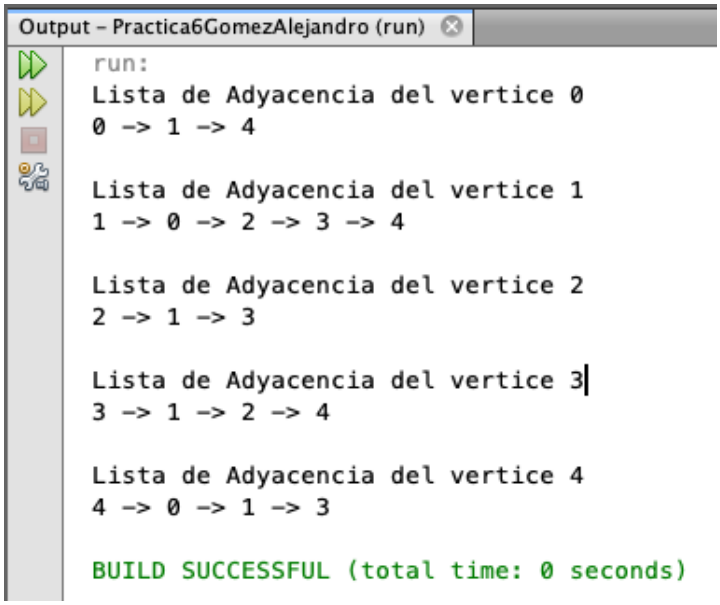
- **Ejercicio 2:** Tomando como base el código anterior, se modificarán los atributos de la clase Vértice, agregando color, distancia y predecesor, además a la clase Grafo es agregado el método BFS, el cual nos brindará una búsqueda primero en anchura, esto quiere decir que al momento de buscar nodos, lo hará primero en una cierta distancia k y ya que haya terminado con ea distancia, procederá a la distancia k-1. El atributo color es utilizado, para uq 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 todo los nodos.
  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.



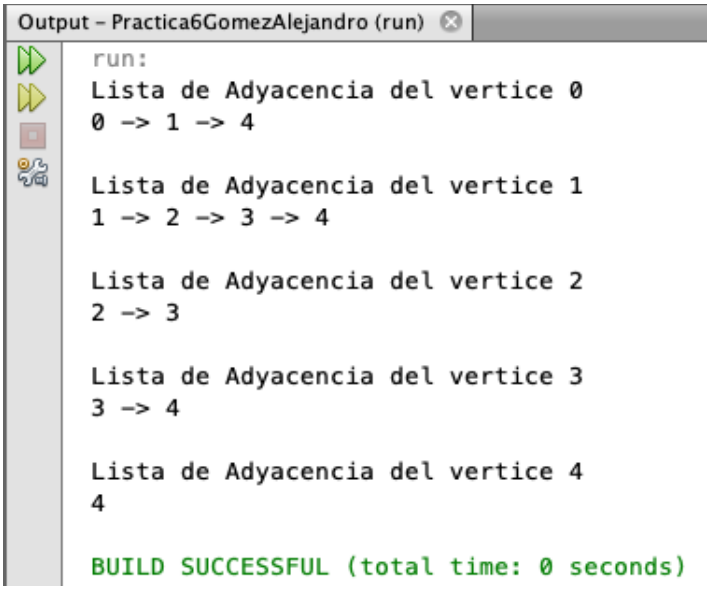
Desarrollo

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.



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.



b. Ejercicio 2: Se crea una clase llamada WightedGraph en la cual el usuario podrá crear su grafo ponderado dirigido. En dicha clase, utilizamos herencia con la clase Graph para acceder a los métodos de dicha clase. Se ha declarado una tabla hash en donde se guardarán las aristas tanto la inicial como final en la clave (para ello se tomaran las claves como tipo String, para el valor dentro de la clave, será el mimo valor de la arista. entonces en el método addEdge() usamos

sobrecargar, pues ahora las aristas tienen un valor especificado.

Conclusiones