Carátula para entrega de prácticas	
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jesús Cruz Navarro
Asignatura:	Estructuras de Datos y Algoritmos 2
Grupo:	1
No de Práctica(s):	7
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	32
Semestre:	2021-1
Fecha de entrega:	18 de noviembre de 2020
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA 7: GRAFOS 2

Objetivo: El estudiante conocerá e identificará las características necesarias para entender el algoritmo de búsqueda por profundidad en un grafo.

a) Diseñar e implementar las clases **Vértice** y **Grafo**, con los métodos **AgregarVertice** y **AgregarArista**, **BreadthFirstSearch**, y **DepthFirstSearch** como se vio en clase *(usando como parámetros los nombres de los vértices, en lugar de pasar un objeto de tipo Vértice)* y modificar la clase vértice para poder utilizar los algoritmos **BFS** y **DFS**.

b) Diseñar e implementar el método **EncontrarCaminoBFS**(*nombreVerticeInicial, nombreVerticeFinal*) que, utilizando el algoritmo **Breadth First Search**. **IMPRIMA** los nombres de los vértices del camino que hay que seguir para llegar desde un vértice Inicial a un vértice Final. Esto se realiza iterando sobre los vértices Padre del nodo final, hasta que un vértice padre sea nulo. Además, imprima el número de estaciones de la ruta.

c) Diseñar e implementar el método **EncontrarCaminoDFS**(*nombreVerticeInicial, nombreVerticeFinal*) que, utilizando el algoritmo **Depth First Search** (el pseudocódigo lo pueden encontrar en la práctica de la coordinación), **IMPRIMA** los nombres de los vértices del camino que hay que seguir para llegar desde un vértice Inicial a un vértice Final. Esto se realiza iterando sobre los vértices Padre del nodo final, hasta que un vértice padre sea nulo. Además, imprima el número de estaciones de la ruta.

d) Desarrolle un programa que genere un grafo con todas las líneas del Metro (una lista con todas las estaciones de metro se encuentra en el archivo **metro.py**). Las estaciones serán los nodos del grafo y la interconexión entre cada estación con otras serán los vértices del grafo.

e) Imprima para los Caminos encontrados con **BFS** y **DFS** para los siguientes trayectos:

   a. *Aquiles Serdán - Iztapalapa*

   b. *San Antonio - Aragón*

   c. *Vallejo – Insurgentes*

Se utilizaron las clases realizadas en la práctica pasada, y se implementaron los métodos correspondientes de la forma analizada en clase. Nótese que el método **DFS** se implementó tomando como referencia el pseudocódigo presente en el libro **Introduction to Algorithms**. Es decir, en éste se genera un **bosque** de árboles **DFS**, en lugar de solamente un **árbol**, y, además, no se toma un nodo inicial, ya que esta tarea corresponde al método **DFS_visit**.

A continuación, se muestra la implementación de los algoritmos correspondientes:

```
def BFS(self, sourceNodeName):
    if not self.hasVertex(sourceNodeName):
        raise KeyError("Source node does not exist")
    self.__restore()
    source = self.getVertex(sourceNodeName)
    self.__BFSAux(source)

def __BFSAux(self, source):
    source.distance = 0
    source.color = Color.GRAY
    queue = Queue()
    queue.put(source)

    while not queue.empty():
        current = queue.get()
        for neighbor in current.neighbors:
            if neighbor.color is Color.WHITE:
                neighbor.color = Color.GRAY
                neighbor.parent = current
                neighbor.distance = current.distance + 1
                queue.put(neighbor)
        current.color = Color.BLACK
```

Implementación de BFS

```python
def DFS(self):
    self.__restore()
    for node in self.vertices.values():
        if node.color is Color.WHITE:
            self.__DFS_visit(node)

def __DFS_visit(self, node):
    self.time += 1
    node.discovered = self.time
    node.color = Color.GRAY
    node.distance = 0 if node.parent is None else node.parent.distance + 1

    for neighbor in node.neighbors:
        if neighbor.color is Color.WHITE:
            neighbor.parent = node
            self.__DFS_visit(neighbor)

    node.color = Color.BLACK
    self.time += 1
    node.finished = self.time
```

Implementación de DFS

Posteriormente, se implementaron los métodos para obtener los caminos correspondientes. En el método **findPathAux**, se realiza la búsqueda con el algoritmo correspondiente y se imprime el camino obtenido, utilizando el método **getPathString**, que retorna una cadena con el camino. Éste último se encuentra implementado de forma recursiva, es decir, primero se obtiene recursivamente el camino al nodo padre del nodo actual, y, finalmente, se imprime el nombre del nodo actual.

Por último, podemos observar los *"wrappers"* del método **findPathAux**, que indican el algoritmo a utilizar en cada caso.

```python
def __findPathAux(self, sourceNodeName, finalNodeName, searchAlg, algName):
    if not self.hasVertex(sourceNodeName):
        raise KeyError("Source node does not exist")

    if not self.hasVertex(finalNodeName):
        raise KeyError("Final node does not exist")

    self.__restore()
    source = self.getVertex(sourceNodeName)
    final = self.getVertex(finalNodeName)
    searchAlg(source)

    if final.distance is math.inf:
        return print("No existe ningún camino")

    print("Camino", algName, "de", sourceNodeName, "a", finalNodeName)
    print("Número de estaciones:", final.distance + 1)
    print(self.__getPathString(final))

def __getPathString(self, node):
    if node.parent is None:
        return node.name
    base = self.__getPathString(node.parent)
    return base + " → " + node.name

def findPathBFS(self, sourceNodeName, finalNodeName):
    self.__findPathAux(sourceNodeName, finalNodeName, self.__BFSAux, "BFS")

def findPathDFS(self, sourceNodeName, finalNodeName):
    self.__findPathAux(sourceNodeName, finalNodeName, self.__DFS_visit, "DFS")
```

Implementación de los métodos para encontrar los caminos

A continuación, se muestran los caminos y distancias obtenidos para los casos solicitados:

```
Camino BFS de Aquiles Serdán a Iztapalapa
Número de estaciones: 21
Aquiles Serdán → Camarones → Refinería → Tacuba → San Joaquín → Polanco → Auditorio –
Camino DFS de Aquiles Serdán a Iztapalapa
Número de estaciones: 52
Aquiles Serdán → El Rosario → Tezozómoc → Azcapotzalco → Ferrería/Arena Ciudad de Méxic

Camino BFS de San Antonio a Aragón
Número de estaciones: 16
San Antonio → San Pedro de los Pinos → Tacubaya → Patriotismo → Chilpancingo → Centro
Camino DFS de San Antonio a Aragón
Número de estaciones: 25
San Antonio → San Pedro de los Pinos → Tacubaya → Juanacatlán → Chapultepec → Sevilla

Camino BFS de Vallejo a Insurgentes
Número de estaciones: 11
Vallejo → Instituto del Petróleo → Autobuses del Norte → La Raza → Tlatelolco → Guerre
Camino DFS de Vallejo a Insurgentes
Número de estaciones: 26
Vallejo → Norte 45 → Ferrería/Arena Ciudad de México → Azcapotzalco → Tezozómoc → El R
```

Salida del programa

CONCLUSIONES

Como se discutió en la práctica anterior, los grafos son una de las estructuras matemáticas más importantes en las Ciencias de la Computación, ya que nos permiten modelar y resolver una gran variedad de problemas. Los algoritmos de búsqueda en grafos representan una forma de recorrer estas estructuras de forma eficiente, y, además, cuentan con ciertas características o rasgos que, dependiendo del problema a resolver, pueden ser óptimos para el objetivo deseado.

El algoritmo **BFS**, como se analizó en clase, recorre el grafo desde un nodo inicial, *"avanzando un nivel"* al añadir todos los vecinos del nodo a una cola, y repitiendo lo anterior hasta haber visitado todos los nodos alcanzables desde el nodo inicial. Debido a que se "avanza" una unidad de distancia en cada iteración, se descubren todos los nodos presentes a cierta distancia del nodo inicial, y, por ende, se obtienen los caminos más cortos respectivos.

Por otro lado, en el algoritmo **DFS**, el recorrido se realiza de una forma distinta: al visitar recursivamente todos los nodos alcanzables desde un nodo inicial, en realidad, implícitamente se terminan de visitar primero los *“últimos”* nodos, es decir, aquellos en los que ya no es posible descubrir algún nuevo nodo en sus vecinos. Después, se realiza un **“backtracking”** y se terminan de visitar los nodos anteriores. El proceso anterior no se ejecuta en un orden específico, es decir, es posible iniciar por visitar cualquier rama del nodo inicial. A partir de este análisis, podemos concluir que la estrategia tomada por este algoritmo no garantiza que los caminos generados por éste sean los más cortos al nodo objetivo.