



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: TISTA GARCÍA EDGAR

Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS II

Grupo: 5

Número de Práctica(s): Guía práctica de estudio 6: Algoritmos de Grafos. Parte II

Integrante(s): MURRIETA VILLEGAS ALFONSO

*Núm. De Equipo de
cómputo empleado :* 38

Semestre : 2019 - 1

Fecha de entrega: 2 DE OCTUBRE DE 2018

Observaciones:

CALIFICACIÓN: _____

ALGORITMOS DE GRAFOS. PARTE II

OBJETIVOS DE LA PRÁCTICA

- El estudiante conocerá e identificará las características de la estructura no lineal árbol.

INTRODUCCIÓN

En la presente práctica se verá la estrategia de recorrido **Depth First Search DFS** o recorrido por profundidad. Este método explora sistemáticamente las aristas del grafo, de manera que los primeros visitados son los adyacentes a los visitados más recientemente.

Se da inicio desde un vértice cualquiera y el recorrido se da por alguno de sus vértices cualquiera, luego un adyacente de este nodo, siendo el método repetitivo y terminará el recorrido cuando llegue a un nodo que ya no tenga nodos adyacentes que visitar.

```
DFS-VISITAR( $G, u$ )
Inicio
    tiempo=tiempo +1
     $u.d$ =tiempo
     $u.color$ =gris
    Para cada vértice  $v$  que pertenece a la lista de adyacencia de  $u$ 
        Inicio
            Si  $v.color == \text{blanco}$ 
                 $v.pred = u$ 
                DFS-VISITAR( $G, v$ )
            Fin Si
        Fin Para
     $u.color$ =negro
    tiempo=tiempo+1
     $u.f$ =tiempo
Fin
```

Imagen 1: Pseudo-código del algoritmo de recorrido de grafos-DFS

Este método de recorrido regresa a un nodo anterior cuando llega al final, esto para checar los nodos adyacentes restantes del nodo anterior. Este método hace uso de la estrategia de **backtrack**, ya que regresa a una solución anterior para checar las posibles soluciones o caminos no recorridos. Esto continúa hasta haber recorrido todos los vértices alcanzables desde el vértice origen.

Durante el recorrido se debe verificar si el dato o vértice ya ha sido leído por el método, si este no lo ha leído entonces se utilizará como punto para el siguiente nodo adyacente, si ya fue tomado, se evitará ya no se contará en los nuevos nodos, pero servirá si este nodo tiene vértices adyacentes aún no recorridos.

Una propiedad de este algoritmo es que se puede usar para la clasificación de arista de grafo de entrada, información que es útil al implementar otros algoritmos como para verificar si un grafo es acíclico, o sea que no se forman caminos que regresen al mismo punto de partida.

CONCEPTOS PREVIOS

Se pueden definir 4 tipos de aristas: Aristas de Árbol:

Árbol: Aristas de un árbol de búsqueda en profundidad. La arista (u,v) es una arista de árbol si v se alcanzó por primera vez al recorrido de dicha arista.

Retorno: Arista (u,v) que se conectan al vértice u con un antecesor v de un árbol de búsqueda. Las aristas cíclicas se consideran aristas de retorno.

Avance: Son arista que no pertenecen al árbol de búsqueda. Pero conectan con un vértice u con un descendiente v que pertenece al árbol de búsqueda en profundidad.

Cruce: Son todas las otras aristas, que pueden conectar vértices en el mismo árbol de búsqueda o en 2 árboles de búsqueda diferentes

Actividad 1. Ejercicios del Manual

Para los grafos existen principalmente 2 algoritmos de búsqueda, BFS y DFS, en el caso de esta práctica se llevará a cabo el caso de DFS.

CONOCIMIENTOS PREVIOS

La programación orientada a objetos es un paradigma donde todo se lleva a cabo mediante la abstracción de objetos o modelos a través de lo que se conoce como clases, donde además influyen otras características más que a continuación serán explicadas:

Clases: Las clases son modelos o plantillas sobre los cuales se construirán objetos, estos están formados por atributos y métodos. En Python una clase se define con la palabra reservada class.

Atributos: Son características propias de un objeto y modifican el estado de este, existen principalmente 2 tipos de variables o atributos las de tipo instancia y las de tipo clase o global.

Métodos: Los métodos o también conocidos como funciones son aquellos encargados de llevar acciones propias de los objetos.

CÓDIGO

Para llevar a cabo el algoritmo de DFS en Python fue necesario llevar a cabo el uso de clases dentro de este, es por ello que, mediante los conocimientos previos de POO, se tuvo primero que llevar a cabo un diagrama UML para poder considerar los requisitos que debía tener el programa, a continuación, se muestra el diagrama:

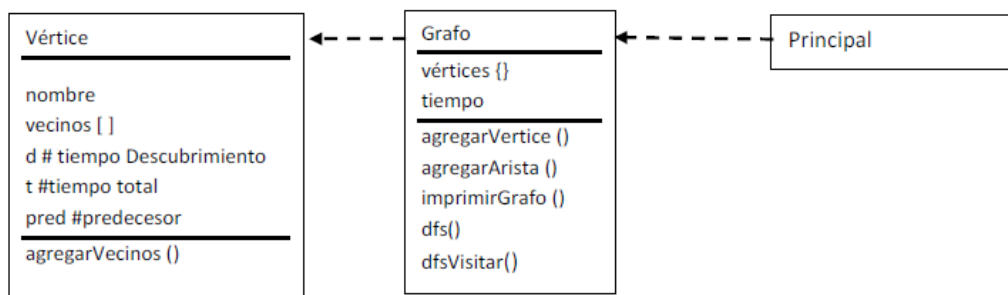


Imagen 2: Diagramas UML – Diagrama de Clases del código previo del algoritmo DFS en Python

ANÁLISIS DEL CÓDIGO

En este ejercicio se dio previamente el código del método DFS, para iniciar se dio a la creación de las clases de vértices y del grafo. A continuación, se explican brevemente cada uno de los métodos:

CLASE VERTICE

Sus atributos son el nombre, la lista de vecinos, el tiempo de inicio en donde se descubrió el nodo y cuando vuelve al mismo nodo al terminar el *backtracking*, el color que tendrá la bandera de si el vértice ya fue leído o aún no y el nodo predecesor a este. Dentro de este existen solo 2 métodos el constructor y el método agregar vecino.

CLASE GRAFO

Sus atributos son los vértices que contiene el grafo y el tiempo en el que fue recorrido. A su vez sus métodos son:

Agregar Vértices:

Checará si el vértice dado es de la instancia Vértice y a su vez si este nodo aún no está en la lista. Si cumple con esto el nodo se agregará a la lista de vértices.

Agregar Aristas:

El cual checará si el nodo de salida y entrada están en la lista de nodos pertenecientes al grafo. Si la llave de la lista es igual a u o v, entonces el vértice será agregado como vecino del vértice que se obtiene del objeto value. Si realiza todo este procedimiento regresará un verdadero, si no es así, será falso, o sea que no hay conexión entre los nodos checados en ese momento. Se creará el método de impresión

Método de Impresión

Obtiene el nodo y mostrará en pantalla el tiempo en el que se recorrió y en donde vuelve al final.

Método de DFS

Obtiene el nodo por donde se iniciará el recorrido, si iniciará el tiempo en el que ingresa la búsqueda por cada nodo. Se recorrerá la lista de nodos, se checará cada nodo, si el nodo es blanco entonces el nodo se enviará al método *dfsVisitar*.

Método de DFSVisitar

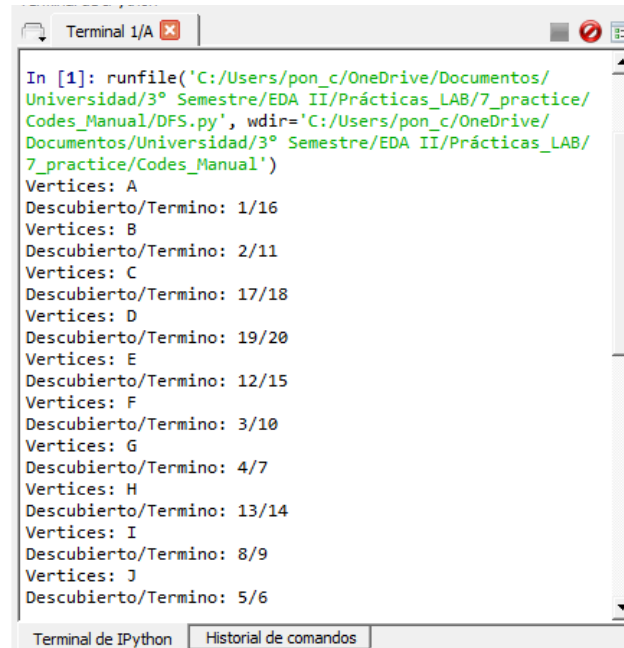
Este método obtiene el vértice y revisa cada elemento de la lista de vecinos, sin antes marcar a este nodo como gris, para que ya no vuelva a ser considerado para los otros nodos. Si el nodo adyacente es de color blanco, entonces el vértice inicial se ingresará como el predecesor del nuevo nodo a checar y el método se volverá a llamar para checar los vecinos de este nuevo nodo, terminará cuando el último nodo adyacente no tenga nodos nuevos y se vuelvan a checar los del nodo anterior.

NOTA: Cuando termina la parte recursiva el método, el nodo final se tornará en negro y el tiempo de recorrido se incrementará en 1

Para el uso de este método se dio uso del **método controlador** que se utilizó en la práctica anterior. Y el método iniciará desde el nodo a (Se puede hacer desde cualquier nodo).

SALIDA DEL PROGRAMA

A continuación, se muestran la salida del programa empleando como entrada el grafo con los siguientes nodos: edges=['AB','AE','BF','CG','DE','DH','EH','FG','FI','FJ','GJ'] :



```
In [1]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3º Semestre/EDA II/Prácticas_LAB/7_practice/
Codes_Manual/DFS.py', wdir='C:/Users/pon_c/OneDrive/
Documentos/Universidad/3º Semestre/EDA II/Prácticas_LAB/
7_practice/Codes_Manual')
Vertices: A
Descubierto/Termino: 1/16
Vertices: B
Descubierto/Termino: 2/11
Vertices: C
Descubierto/Termino: 17/18
Vertices: D
Descubierto/Termino: 19/20
Vertices: E
Descubierto/Termino: 12/15
Vertices: F
Descubierto/Termino: 3/10
Vertices: G
Descubierto/Termino: 4/7
Vertices: H
Descubierto/Termino: 13/14
Vertices: I
Descubierto/Termino: 8/9
Vertices: J
Descubierto/Termino: 5/6
```

Imagen 3: Salida del programa empleando el algoritmo de recorrido de grafos DFS.

Actividad 2. Ejercicios del laboratorio

2.A Compilar y ejecutar proyecto (Grafo propuesto y grafos de la clase)

Para este apartado simplemente se creó un proyecto en NetBeans donde la clase principal “Murrieta_practica7” la cual a través de su método main fuera la encargada de llamar a la clase “Graph” donde a través de sus métodos implementaría la lógica tanto de la creación de grafos como del recorrido de este, a continuación, se explicarán los métodos correspondientes y sus respectivas salidas del programa:

DEFINICIÓN DE MÉTODOS GENÉRICOS:

A continuación, se presentande manera explícita y particular las definiciones de cada uno de los métodos de esta clase, esto con el fin de poder explicar con el mayor detalle posible el programa:

NOTA: Variables globales

Debe considerarse que como variables globales de la clase se tiene un valor entero asociado a los vértices del grafo, además de una lista ligada o LinkedList la cual es la estructura de datos que será empleada para poder llevar a cabo la representación en computadora de un grafo.

1] Método constructor Graph

Es un método dedicado a la construcción de la clase (Para saber que es lo necesario o que debe realizarse al momento de instanciar un objeto de esta clase) que como parámetros necesarios requiere un valor entero asociado a la cantidad de vértices que tendrá el grafo.

Dentro de este método a su vez se llevará a cabo mediante un for la construcción del grafo a través de la implementación de una lista ligada o linkedList.

2] addEdge

Es un método void encargado de la conexión de los vértices y de la asignación de estos a sus listas de adyacencia, para ello lo que necesita es que se le pasen dos valores enteros que representan los vértices.

Para poder llevar a cabo el apartado de la conexión y asignación lo que se hace es simplemente agregar a ambas listas tanto los valores que conectan de un vértice al otro y así de manera inversa en la siguiente línea de código

3] printGraph

Es un método del tipo void al que se le pasa como parámetro una clase del tipo Graph. Debido a que para llevar a cabo los grafos es necesario un LinkedList es necesario llevar a cabo

DEFINICIÓN DE MÉTODOS (DFS):

1] Método DFS()

El método DFS es un método del tipo void que como parámetros requiere solamente pasarle un entero que es el referente al nodo de donde se empezará a recorrer el grafo.

Dentro de este método se tiene un arreglo booleano que será empleado para marcar si están o no visitados los nodos. Por último, este método llama al método DFSUtil al cual se le pasa como parámetros el valor del nodo como el arreglo de booleanos.

2] Método DFSUtil()

Es el método que realmente lleva toda la lógica del programa, como se mencionó previamente requiere como parámetros el nodo de inicio y un arreglo de booleanos.

Dentro de este método se declara nuevamente un arreglo de booleanos con la posición o en el nodo en el que se ha mandado al método, obviamente una vez hecho esto se marcar inmediatamente o se le da el valor de True para ya no considerarlo dentro del recorrido.

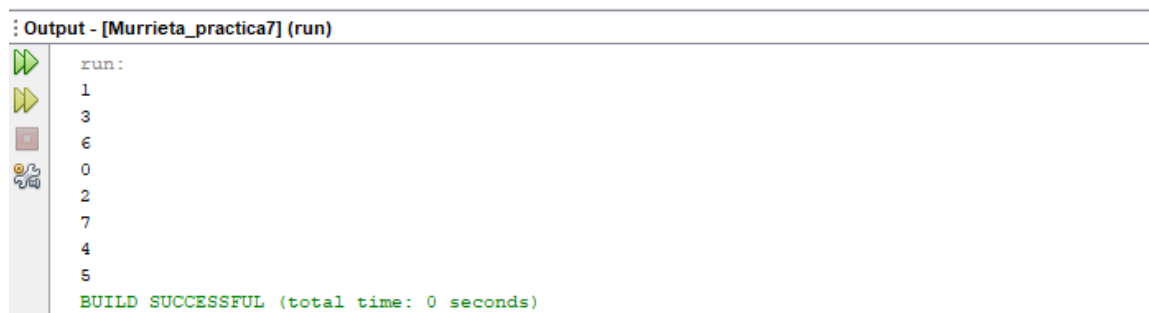
NOTA: Inmediatamente es en esta parte es cuando se imprime el nodo marcado

Por último, nuevamente hacemos uso del iterador que se declaró como variable global al inicio de la clase, el cual servirá como dice su nombre, ir recorriendo el grafo una vez dentro del ciclo while que tiene el método, dentro del ciclo while lo único que hace es simplemente es ir recorriendo el grafo, sin embargo, la parte crucial del algoritmo es en la última condición if donde a través de la recursividad del mismo método es cómo es posible el recorrido de todo el grafo.

NOTA: Al igual que en la versión en Python previa, DFS se llevó a través de su forma recursiva donde se manda tanto el nodo de inicio como el arreglo de booleanos, lo que realmente pasa es que debe iniciarse desde el nodo de inicio debido a la lógica de DFS que es ir hasta el fondo del grafo con la condición de no hacer un ciclo (esa parte se encarga el ciclo while)

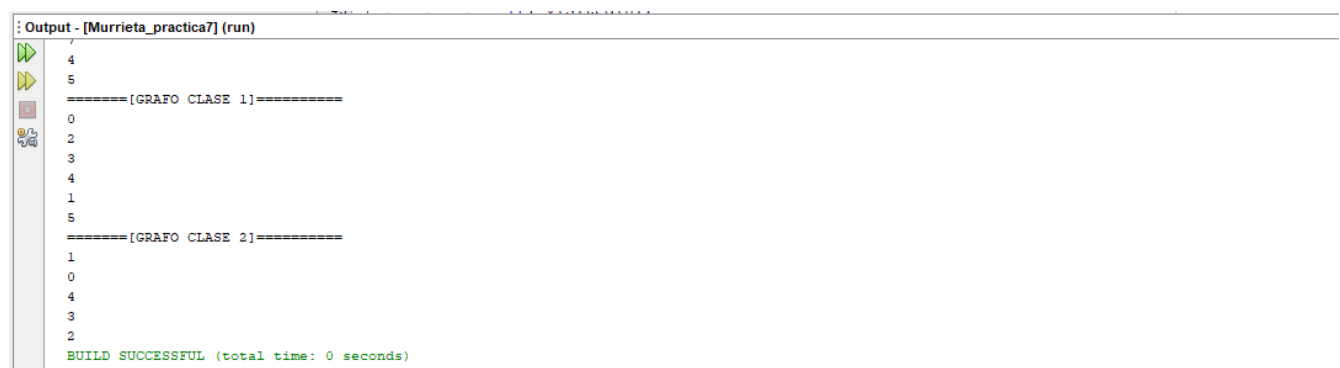
APLICACIÓN EN CÓDIGO Y SALIDA DEL PROGRAMA:

A continuación, se muestran capturas de pantalla correspondientes a las salidas de cada uno de los métodos de la clase presente y de los grafos respectivos:



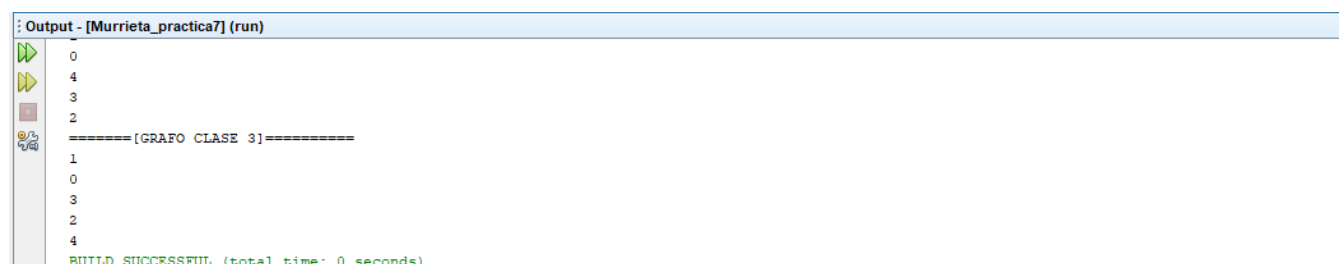
```
Output - [Murrieta_practica7] (run)
run:
1
3
6
0
2
7
4
5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 4: Salida del programa empleando el algoritmo de recorrido de grafos DFS del grafo propuesto en clase.



```
Output - [Murrieta_practica7] (run)
4
5
=====GRAFO CLASE 1=====
0
2
3
4
1
5
=====GRAFO CLASE 2=====
1
0
4
3
2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 5: Salida del programa empleando el algoritmo de recorrido de grafos DFS de los 2 grafos vistos en la clase de teoría.



```
Output - [Murrieta_practica7] (run)
0
4
3
2
=====GRAFO CLASE 3=====
1
0
3
2
4
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 6: Salida del programa empleando el algoritmo de recorrido de grafos DFS del tercer grafo

2.B Grafo ponderado e Implementación de Primm

Para este apartado lo que se pidió fue emplear el código previo de los grafos ponderados (Práctica 6) y posteriormente emplear el algoritmo de árbol de expansión mínimo PRIMM, para ello se nos proporcionó un pseudocódigo. A continuación, se plantea la forma en que se abordó el algoritmo mediante sus respectivos métodos y clases.

NOTA: Debido a la incompatibilidad de la forma empleada para llevar los grafos ponderados en la práctica 6 se decidió partir nuevamente desde cero para de esta forma emplear listas de adyacencia en vez de la matriz de adyacencia.

DEFINICIÓN DE CLASES:

1] CLASE ARISTA

La clase Arista es realmente una forma auxiliar en la que se abordó la manera en que realmente se iban a llevar a cabo el apartado de guardado de cada una de las aristas del grafo, como variables globales se consideraron los 3 elementos que comparten todas las aristas en un grafo ponderado.

- 1) El nodo del que se parte
- 2) El nodo en donde termina
- 3) El valor que tiene la arista

NOTA: Solo como apartado extra de POO y por conveniencia al momento de instanciar, se decidió realizar un método constructor para poder declarar la firma del constructor.

2] CLASE VERTICE

Al igual que la clase arista, es una clase realmente auxiliar que se encarga de guardar los valores de cada uno de los vértices como los vértices o nodos vecinos que tiene.

Para poder llevar la lógica de esta clase, lo que se realizó simplemente fue emplear una lista donde se guardara los vértices vecinos o adyacentes del vértice del que se trata y a su vez mediante una variable entera guardar el valor de este.

NOTA: Solo como apartado extra de POO y por conveniencia al momento de instanciar, se decidió realizar un método constructor para poder declarar la firma del constructor.

3] CLASE VERTICES- ADYACENTES (AUXILIAR)

Esta clase se desarrolló con el propósito de poder llevar a cabo las relaciones y comparaciones del estado, conexión y el valor (Ponderación) entre los vértices que son adyacentes con otros vértices.

Lo primero que se realizó fue la declaración de los 3 elementos que debía considerar esta clase para poder llevar a cabo las relaciones al momento de que se recorriera el árbol mediante PRIMM y es que recordemos que la forma en que se debe recorrer es a través de un nodo llegan al otro nodo y considerando que los valores de las aristas sumen o den el mínimo total del peso del árbol. Como variables globales se determinaron dos objetos del tipo vértice que iban a ser tanto el vértice con el que está conectado y a su vez el mismo vértice donde se encuentra en el recorrido, por último, una variable entera encargada de guardar los valores de la ponderación entre ambos vértices.

NOTA: Esta clase está heredando mediante una interfaz (Por eso el implements en vez del extends) la colección comparable de Java la cual sirve para ordenar elementos o en este caso objetos en el orden en que se van utilizando (Con el fin de llevar a cabo la cola de prioridad).

NOTA 2: Por conveniencia al momento de instanciar, se decidió realizar un método constructor para poder declarar la firma del constructor.

MÉTODOS

1] getters

Los getters son métodos empleados sobre todo para poder acceder a variables que son privadas (Conceptos de POO), sin embargo, en este caso realmente los getters no los utilicé para eso, Debido a que esta clase está heredando mediante una interfaz es por ello que para no confundir variables fue mejor emplear los getters directamente al llamar o utilizar estas propiedades.

2] ToString

Es un método muy conocido en el uso de clases en POO debido a que de esta forma directamente la clase ya tiene un método de salida, el cual como dice el nombre regresa una cadena.

3] compareTo

Es un método de la clase que se está heredando el cual funciona para comparar elementos, en este caso se empleó para poder llevar a cabo la inserción de vértices adyacentes dentro de la cola de prioridad que se encuentre en la siguiente clase. La lógica que lleva es la siguiente (Basado en el algoritmo de Prim – Libro de Java (referencia abajo)):

```
if (ponderacion>vert.getPonderacion()) {  
    return 1;  
}  
else if (ponderacion<vert.getPonderacion()) {  
    return -1;  
}  
else {  
    return 0;  
}
```

Imagen 7: Líneas de código encargado de la anexión y comparación de los vértices adyacentes

NOTA: Debemos aclarar que al ser heredada mediante una implementación de una interfaz es por ello que debe sobre-escribirse el método (Recordemos que se trata de una clase no abstracta).

4] CLASE GRAFO PONDERADO

Esta clase es la versión 2 de grafos ponderados, debe aclararse que debido a la poca compatibilidad de la forma en que se abordó la solución en la práctica anterior, tuvo que necesariamente realizar esta clase desde cero, entre sus características están el poder emplear todas las clases previamente comentadas para poder llevar de una forma más cómoda el algoritmo de Prim y los grafos ponderados.

VARIABLES GLOBALES

Se emplearon listas ligadas o linkedlist para guardar los valores ponderados y la lista de los vértices, a su vez dos variables encargadas de guardar los valores fundamentales, por un lado la cantidad total de las aristas y por otro un arreglo entero encargado del guardado de los valores de ponderación.

MÉTODOS

1] Constructor

A diferencia de los anteriores métodos constructores de las otras clases, en este caso este método es a su vez el encargado de llevar el apartado de guardado dentro de ambas listas ligadas y a su vez el que asigna los valores a todas las variables globales previamente mencionadas

2] addedges

Este método como menciona su nombre es el que lleva el apartado de asignación de los valores de los ponderados y a su vez de los vértices y sus respectivos vértices adyacentes, esto con el motivo de poder llevar el conteo y apartado de vértices adyacentes del vértice en el que se encuentra el algoritmo.

NOTA: Como se puede ver aquí también se guardan los valores de los ponderados en la variable global ponderados que era un arreglo de dos dimensiones (matriz)

3] printGraph

Al igual que en la práctica 6, en este caso simplemente se anidaron dos ciclos for para realizar la impresión de la lista de adyacencia en vez de la matriz de adyacencia que se había hecho. Cabe destacar que los cambios que sufrió este método solamente fueron cambiar el arreglo que se había hecho por la lista ligada que es en este caso la variable global empleada para el guardado de los vértices.

MÉTODOS EMPLEADOS PARA EL ALGORITMO DE PRIMM

Cabe destacar y aclarar que este apartado fue realmente la parte más compleja de llevar sobre todo porque uno de los principales requisitos que, aunque parecía inofensivo no lo era fue el llevar el guardado de guardado y comparación dentro de la cola de prioridad.

Para llevar el algoritmo de Prim en general requirió de los siguientes 2 métodos sin embargo cabe destacar que todas las previas clases que se emplearon para auxiliar el ordenamiento realmente son la base de esto.

1] Prim

Es un método de tipo void encargado de llevar la parte del recorrido del grafo mediante el algoritmo de árbol de expansión mínimo Prim (MTS), en la primera parte están declarados todos los objetos y variables que fueron necesarios:

NOTA: A este método es necesario pasarle el grafo y su respectivo nodo o vértice en el que se iniciará el recorrido.

VARIABLES y OBJETOS

- 1) Una lista de booleanos como auxiliar de si ya había pasado o no por un vértice (nodo)
- 2) la cola de prioridad a través de una colección de la biblioteca Java Util
- 3) La lista de los valores que se irán guardando para el MTS
- 4) Una clase del tipo vértice la cual sirve para guardar el nodo del que se iniciará el recorrido
- 5) Un arreglo de los valores de los vértices
- 6) Por último, el iterador que en todas las versiones de grafos se ha empleado (Iterator)
- 7) Por último, una variable entera para guardar el peso o tamaño del árbol de expansión mínimo.

Posteriormente, a través tanto de ciclos `for` como se fue llevando a cabo la respectiva lógica del algoritmo basado en el pseudocódigo que se dio en el laboratorio.

En el ciclo `while` solamente se llevó a cabo el guardado y recorrido de los vértices de adyacencia y del guardado de estos en la cola de prioridad.

NOTA: Es importante denotar que antes de llegar al siguiente ciclo es donde se llama al método principal de todo el algoritmo de prim que es "UtilPrim"

Por otro lado, el ciclo `for` es el que hace realmente el apartado de impresión del MTS además de que ahí convenientemente se decidió meter la variable `peso` para poder dar el total del MTS a través de la suma de las ponderaciones de los vértices de la lista de Prim.

2] UtilPrim

Este método cabe destacar que está directamente basado en otro código encontrado en GitHub, cabe destacar que a su vez se dan los créditos y la dirección web del código dentro del mismo código.

UtilPrim es un método del tipo void el cual se le pasan prácticamente todas las variables declaradas en el método anterior que es `prim`, realmente la razón de porque todas las líneas de código de este método no están en el método anterior es debido a que para poder llevar la verificación de que la cola de prioridad esté vacía es necesario llevar varias veces el recorrido es por ello que el método en su última parte a través de una condición `if` se vuelve recursivo.

SALIDA DEL PROGRAMA

A continuación, se muestran tanto la lista de adyacencia como el árbol de expansión mínimo de un grafo ponderado.

```
Output - [Murrieta_practica7] (run)
LISTA DE ADYACENCIA [GRAFO]
[Para: 0]
0 -> 2 Valor: 1
0 -> 3 Valor: 3
[Para: 1]
1 -> 3 Valor: 5
1 -> 5 Valor: 2
1 -> 2 Valor: 3
[Para: 2]
2 -> 0 Valor: 1
2 -> 1 Valor: 3
2 -> 4 Valor: 2
2 -> 5 Valor: 1
[Para: 3]
3 -> 0 Valor: 3
3 -> 1 Valor: 5
3 -> 4 Valor: 6
[Para: 4]
4 -> 2 Valor: 2
4 -> 3 Valor: 6
[Para: 5]
5 -> 1 Valor: 2
5 -> 2 Valor: 1
```

Imagen 8: Salida del programa donde se puede ver la lista de adyacencia del grafo además de los respectivos valores de las aristas

```
Output - [Murrieta_practica7] (run)
MST - PRIMM
Nodo de inicio: 0
->
INICIO [ 2 ] FIN [ 0 ] Valor: 1
->
INICIO [ 5 ] FIN [ 2 ] Valor: 1
->
INICIO [ 1 ] FIN [ 5 ] Valor: 2
->
INICIO [ 4 ] FIN [ 2 ] Valor: 2
->
INICIO [ 3 ] FIN [ 0 ] Valor: 3

El total del recorrido es: 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 9: Salida del programa donde se puede ver el árbol de expansión mínimo MTS del grafo anterior como se puede ver se dan los vértices, los valores de las aristas y además el peso o el total del recorrido.

CONCLUSIONES

En la presente práctica se llevó a cabo la implementación del algoritmo de recorrido de grafos DFS y además el algoritmo de árbol de expansión mínimo Primm , donde a través de la ayuda de los conceptos de orientado a objetos tanto en Java como en Python es como se pudieron realizar los métodos y clases necesarias para codificar los algoritmos.

Cabe destacar que en esta primera entrega solamente se pudo concluir el primer ejercicio en todos sus apartados, debido a que en el caso del segundo ejercicio el llevar a cabo el algoritmo de Primm el primer reto fue llevar a cabo las colas de prioridad lo cual cabe destacar que para fortuna propia ya existe objetos de esas características en la biblioteca de Java.Util (PriorityQueue), sin duda alguna, para la segunda entrega será más que definitivo la entrega de lo faltante de esta práctica.

Sin duda alguna uno de los programas más complicados de llevar a cabo, realmente fue muy complicado este algoritmo sobre todo porque tuve que partir nuevamente para el desarrollo del grafo ponderado, lamentablemente la versión de la práctica previa fue bastante ineficiente e incompleta.

Por otro lado, realmente uno de los factores que ayudó muchísimo fueron todos los conocimientos que he aprendido en POO como fue la creación de clases auxiliares y por ejemplo la herencia a través de interfaces para poder emplear métodos de clases mediante la sobre escritura.

También es necesario destacar que uno de los factores realmente lamentables fue que no pude realizar de manera propia el recorrido mediante Prim sobre todo por todos los problemas que se tuvieron con la cola de prioridad, es por ello que esta vez necesariamente tuve que recurrir a otras fuentes como stackoverflow o github para poder ver como otros desarrolladores llevaron a cabo el algoritmo de Prim.

Por último, nuevamente es necesario mencionar que algo realmente destacable es la gran ayuda que ofrece Java al momento de llevar a cabo el uso de bastantes estructuras de datos o colecciones como son las listas ligadas, las colas de prioridad y entre muchas otras.

REFERENCIAS

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). *Problem Solving with Algorithms and Data Structures using Python*. Segunda Edición.

Elba Karen Saenz García. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos II*. UNAM, Facultad de Ingeniería.

Recuperado el 5 de octubre del 2018, de <https://github.com/search?q=PRIMM+ALGORITHM>