Universidad Simón Bolivar
Septiembre-Diciembre 2018
Laboratorio de Algorimos y Estructuras III CI-2693

Jesús De Aguiar 15-10360
Wilfredo Graterol 15-10639

Informe - Proyecto I: TAD Grafo

Diseño de la estructura:

Este proyecto consiste en implementar una estructura de datos de Grafo como una lista de adyacencias. En este caso decidimos utilizar las siguientes interfaces y clases abstractas:

1. Graph 2.Node 3.GraphEdges 4.Transformer

Al implementar la interfaz *Graph* se creó las clases *DirectedGraph* y *UndirectedGraph.* Al implementar la clase abstracta *Node,* creamos las clases *NodeD* y *NodeU,* las cuales serán los tipos de nodos que se utilizarán en los digrafos y grafos no dirigidos respectivamente. Además para representar los lados se crearon las clases *Arc* y *Edge* para representar arcos y aristas respectivamente. La manera en la que se decidió guardar los nodos y los lados en los grafos fue la siguiente: En cada una de las clases *DirectedGraph* y *UndirectedGraph* se guarda un *HashSet* en el que se guardara todos los nodos del grafo, un par de enteros que representan el numero de lados y nodos que hay en el grafo, un par de *HashMaps* para relacionar los nombres de cada nodo y los lado con ese elemnto, y un par de *Transformers* que permitiran parsear los datos especiales al momento de agregar nodos y lados al cargar el grafo de un archivo. Todos los nodos tienen como atributo un String que representa su identificador y un *Double* que representa su peso. Cada *DNode* guarda como atributo su grado interno y externo, un par de *HashSets* para sus sucesores y predecesores y un par de *ArrayList* para los arcos que salen y los arcos que entran en él. Para el caso de los *UNode* es similar, con la diferencia de que solo se guarda un entero para el grado, un *ArrayList* para los lados incidentes y un *HashSet* para los nodos adyacentes. Los lados solo guardan su nodo inicial y final, su dato y su peso.

En la implementación se le dá prioridad a las operaciones de inserción, búsqueda y query, pagando el precio de hacer la operación de eliminación costosa en cuanto a tiempo de ejecución. De la misma manera, se le da prioridad a la velocidad de ejecución sobre el uso de memoria (Aunque se busca que el uso de memoria sea simplemente lo necesario para que esto se cumpla). Esta decisión fue tomada dado que, esperamos utilizar la operación de eliminación muy poco en comparación con las demás. Al mantener todas estas estructuras alternativas podemos saber de manera rápida, si un nodo o lado pertenece o no en el grafo o si es adyacente a otro, entre otras operaciones. Por ejemplo, al usar un *HashMap* podemos verificar de manera simple si un identificador de un nodo o un lado ya pertenece al grafo. O por ejemplo, al usar un *HashSet* para representar la lista de nodos y lados, las operaciones de inserción y eliminación pueden tomar un tiempo cercano al constante (Dependiendo de la función de hash)

El cliente:

Para correr el cliente es necesario desempaquetar el archivo Codigos.tar en una carpeta y correr *javac *.java* en ese directorio, una vez hecho eso debe correr en el terminal java *ClienteGrafo*. Al correr el comando se iniciará el menú en el cuál se tendrá acceso a todas las operaciones básicas de la estructura, solo debe escribir el numero de la opción que desee y marcar *enter*. El menú se vuelve a mostrar al final de cada selección, a menos que el usuario seleccione la opción de salir o alguna excepcion sea arrojada debido a algún error. Al inicio del main del cliente se inicializan dos grafos, uno dirigido y uno no dirigido, esto se hizo pues se generaban errores de compilación crear el grafo como *Graph* (Clase abstracta), e inicializarlo luego como dígrafo o grafo no dirigido (Clases concretas) y utilizar operaciones únicas de estos.

Sin embargo, en el cliente solo se permite mantener un grafo a la vez, al crearse uno, el anterior se deja de utilizar y pasa a ser eliminado por el recolector de basura de Java, y solo se utiliza el nuevo para todas las operaciones. Si intenta usar operaciones únicas de digrafos en un grafo no dirigido (como calcular el grado interno de un nodo), el programa lanzara una excepción y finalizara su ejecución. Cualquier entrada de un tipo de dato no valido al agregar un nodo o un lado también provocará que el programa lance una excepción. Todas las operaciones del cliente se basan en las operaciones de las clases de los grafos. En particular las opciones para cargar un grafo desde un archivo, agregar un nodo y agregar un lado se apoyan de unos atributos que implementan la interfaz *Transformer*. Estos atributos se inicializan al leer el archivo o al crear el grafo y se encargan de parsear los datos especiales de los nodos y lados una vez definido este tipo.

Casos de prueba usados:

Para probar cargar un grafo desde un archivo probamos con dos archivos adjunto en el proyecto, uno es un archivo llamado “dirigido” el cual carga en el programa un digrafo y contiene lo siguiente:

```
D
B
D
7
10
cero 0.5 0
uno 1.5 1
dos 2.5 2
tres 3.5 3
cuatro 4.5 4
cinco 5.5 5
seis 6.5 6
ceroconuno false 0 cero uno
unocondos true 1 uno dos
cerocondos false 2 cero dos
dosconcero false 2 dos cero
tresconcuatro true 3 tres cuatro
cuatroconcinco true 3 cuatro cinco
seisconseis true 6 seis seis
cincocondos true 5 cinco dos
trescondos true 2 tres dos
ceroconcuatro false 0 cero cuatro
```

El otro archivo, llamado “nodirigido” es similar pero carga en el programa un grafo no dirigido. En las pruebas no se detecto ningún error con ninguno de los dos archivos, tanto al cargarlos como al realizar operaciones sobre ellos. También se realizaron pruebas iniciando el grafo mediante las operaciones del menú y tampoco se detecto ningún error.