Pontificia Universidad Javeriana
Facultad de Ingeniería
Ingeniería de Sistemas y Computación
Estructuras de datos

TIPO ABSTRACTO DE DATO: GRAFO

Profesor : Carlos Alberto Ramírez Restrepo
Autores : Jeffrey Steven García Gallego
Mauricio Cortés Diaz
Fecha : 29/11/2017

Índice

1. Introducción

El proyecto del curso consiste en la implementación del Tipo Abstracto de Dato (TAD): Grafo, realizando previamente el proceso de abstracción del TAD y posteriormente implementando nuestro diseño de grafo en el lenguaje de programación $C++$. En adición a lo anterior y acorde a lo propuesto en el enunciado del proyecto se implementaron una serie de operaciones sobre el diseño de grafo implementado anteriormente, los cuales se basan en los siguientes algoritmos: Breadth First Search (Búsqueda primero en amplitud), Depth First Search (Búsqueda primero en profundidad) y Dijkstra.

Para la realización de este proyecto se utilizaron conceptos, métodos y explicaciones vistas en clase, por lo tanto, se pusieron en práctica los temas y objetivos del curso.

2. Implementaciones

2.1. Implementación TAD Grafo

Para la implementación del TAD grafo se optó por trabajar únicamente con vectores para realizar las respectivas representaciones acordes al modelo de "listas de adyacencia". Principalmente se eligieron los vectores porque según el diseño que se había implementado, el acceso a elementos era una parte importante para el correcto y seguro funcionamiento del TAD. Los vectores ofrecen facilidades para trabajar de esta manera, como la sobrecarga del operador [] y además el acceso a elementos es constante. Sin embargo, en este caso, operaciones como eliminar e insertar en alguna posición sería de complejidad constante amortizada, debido a las realocaciones que debe hacer al eliminar o insertar un elemento, que en comparación con otros contenedores como listas, serían de complejidad constante. El diseño se enfocó también en la capacidad para trabajar con distintos tipos de datos, para lograr esta característica se recurrió al uso de plantillas o *templates*.

Siguiendo el modelo de listas de adyacencia se crearon tres vectores: el primero contiene los vértices o nodos del grafo; el segundo, las aristas de cada nodo, teniendo un vector para cada nodo en la misma posición en la que ese nodo se encuentra en el vector de vértices, teniendo un mayor control sobre la relación de un nodo con sus adyacentes; por último, el tercero, donde se encuentran las ponderaciones de cada arista de cada nodo, siguiendo siempre

un orden con la posición del nodo y con la posición de la arista relacionada al nodo. Para un mayor control, se establecieron dos atributos, los cuales indican el número de aristas y vértices en el grafo. Este diseño fue similar al implementado en la clase "grafo con dirección", los cambios fueron realizados únicamente sobre el manejo de las ponderaciones de cada arista y abstractamente sobre el orden de los parámetros de los métodos.

2.2. Implementación Operaciones Grafo

Para la implementación de las operaciones que se pueden realizar con el diseño implementado de grafos fueron utilizadas las siguientes estructuras de datos: colas, pilas y colas de prioridad. La implementación de cada uno de los algoritmos fue en base al pseudocódigo de cada algoritmo. Se hizo uso del polimorfismo de métodos para que éstos fueran compatibles tanto para grafos sin dirección como para grafos con dirección.

Para la implementación del algoritmo Dijsktra fue creada una nueva clase que serviría para almacenar dos tipos de datos distintos en la cola de prioridad utilizada. Así como también fue necesario para el correcto funcionamiento de la cola de prioridad un functor que hiciera la comparación entre cada elemento de la cola.

3. Complejidades

3.1. TAD Grafo

3.1.1. Grafo()

El constructor por defecto para clase Grafo tiene una complejidad constante $O(1)$, debido a que los vectores se inicializan como vacíos y la asignación se realiza en tiempo constante.

3.1.2. Grafo(nodos)

La complejidad del constructor que recibe como parámetro un vector de nodos, tiene una complejidad lineal sobre el número de elementos del vector de nodos $O(n)$, porque se debe recorrer el vector para agregar cada nodo al vector de nodos de la clase grafo.

3.1.3. AgregarArista(n1, n2, *valor*)

La complejidad de éste método es lineal sobre el número de nodos en el grafo sumado al número de aristas de los nodos en concreto $O(|V| + |E|)$. Se debe acceder al vector de nodos para conocer las posiciones de cada uno de los dos nodos, donde el peor caso sería $|V|$ y posteriormente se hace la verificación que debe recorrer el vector con las aristas correspondientes a cada nodo, lo cual se repetiría en el peor de los casos $|E|$ veces.

3.1.4. AgregarNodo(n1)

La complejidad de este método es constante $O(1)$. Se crean vectores vacíos, que, como se había afirmado antes, se hace en un tiempo constante. Posteriormente se agregan elementos al vector con una operación de éste, el cual también la realiza en tiempo constante porque se tiene un acceso constante a la última posición del vector.

3.1.5. ModificarArista(n1, n2, valor)

La complejidad de este método es lineal sobre el número de nodos en el grafo sumado al número de aristas de los nodos en concreto $O(|V|+|E|)$. Esto es porque se debe acceder a la posición de los nodos, lo cual, en el peor caso se tendría que repetir $|V|$ veces. Posteriormente se debe hallar la posición de la arista que se quiere modificar, para esto es necesario recorrer el vector que contiene las aristas de un nodo en específico, lo cual se tendría que repetir en el peor caso $|E|$ veces.

3.1.6. EliminarArista(n1, n2)

La complejidad de este método es lineal amortizado sobre el número de nodos en el grafo sumado al número de aristas de los nodos en concreto $O(|V| + |E|)$. Debido a que se debe acceder a la posición de los nodos recorriendo el vector de nodos de la clase y posteriormente buscar la posición de la arista que se quiere eliminar. El vector debe posicionar las demás aristas al haber una eliminación, todo esto ocurre a nivel de memoria.

3.1.7. EliminarNodo(n1)

La complejidad de este método es lineal amortizado sobre el número de nodos en el grafo, debido a que se debe encontrar la posición del nodo que se quiere eliminar, lo cual tendría una complejidad de $O(|V|)$. El vector debe posicionar los demás nodos al haber una eliminación, todo esto ocurre a nivel de memoria.

3.1.8. ObtenerListaAdy(n1)

La complejidad de este método es lineal sobre el número de nodos en el grafo $O(|V|)$, debido a que se debe hallar la posición del nodo al que se le quiere conocer la lista de adyacencia.

3.1.9. ObtenerNumVertices(), ObtenerNumAristas(), ObtenerVertices()

La complejidad de estos métodos es constante $O(1)$, debido a que únicamente acceden a atributos de la clase y posteriormente los retornan.

3.1.10. ObtenerPonderaciones(n1)

La complejidad de este método es lineal sobre el número de nodos en el grafo $O(|V|)$, porque debe encontrar la posición del nodo al que se le quiere conocer el vector de ponderación asignado, lo cual, en el peor caso llevaría recorrer todo el vector de nodos.

3.2. Operaciones Grafo

3.2.1. Búsqueda Primero en Amplitud (BFS)

La implementación del algoritmo BFS, en este caso presenta una complejidad $O(|V|^2 * |E|)$. Esto es debido a que, primero, se debe comprobar que la cola no esté vacía, y la máxima cantidad de elementos que puede tener la cola es de $|V|$, por lo que, en el peor caso este ciclo se repetiría $|V|$ veces. Posteriormente se comprueba si un nodo de la lista de adyacencia está visitado o no, para realizar la comprobación primero se recorre la lista de adyacencia del nodo y después se tiene que verificar con otro ciclo que el nodo sea igual al que se encuentra en la lista de adyacencia y que su estado este como no visitado en el vector de visitado, para así tener acceso también a la posición

que ocupa el nodo en el vector de visitado. Todo este proceso tiene un costo de $|V| * |E|$ en el peor caso.

3.2.2. Búsqueda Primero en Profundidad (DFS)

La implementación de este algoritmo DFS, en este caso presenta una complejidad $O(|V|^2 * |E|)$ ya que se trabaja con listas de adyacencia, por lo cual este algoritmo debe hacer un recorrido de todos los nodos marcando cada nodo recorrido como visitado, a su vez se realiza la verificación de la lista de nodos conectados al nodo padre y así sucesivamente hasta encontrar el final de una rama, para posteriormente continuar desde el padre de los nodos visitados hacia nodos no visitados.Para este algoritmo se usa una pila en la cual se ingresan los nodos visitados y que faltan por verificar completamente los nodos a los cuales esta conectado, cuando ya se verifican totalmente los nodos adyacentes a este y se marcan como visitados, este nodo sale de la pila; el algoritmo finaliza una vez que la pila este vacía (todos los nodos visitados). Por todo este proceso toma tal complejidad la implementación del algoritmo.

3.2.3. Dijkstra

La implementación del algoritmo Djikstra, en este caso presenta una complejidad $O(|V|^2 * |E|)$. Esto es debido a que se debe verificar que la cola no está vacía, y mientras ésta no esté vacía se debe acceder a la lista de adyacencia del nodo actual y compararlo con los nodos en la lista de vértices de grafo, donde, en el peor caso, la complejidad sería de $O(|V| * |E|)$. A esta complejidad se le multiplican las veces que entra en el primer ciclo, que serían $|V|$ veces en el peor caso, generando así la complejidad mencionada de esta implementación.

4. Conclusiones

Todas las implementaciones del proyecto resultaron con un balance positivo, por su funcionamiento y versatilidad, gracias al uso templates, y al diseño de los métodos para cada implementación, los cuales a su vez brindan una mayor comodidad y familiaridad al utilizar la clase grafo, ya que se asemeja a como se trabaja con los contenedores de STL. También fue utilizado el polimorfismo para lograr una mayor unificación en todo el proyecto. Sin embargo, al obviar la clase *Nodo* en la implementación del TAD grafo

se presentaron ciertos inconvenientes que afectaron a la complejidad de las operaciones sobre grafos. Ya que para realizar la comprobación de si el nodo había sido visitado o no hacía falta que esa operación se hiciese en tiempo constante teniendo un atributo en la clase nodo para ello, pero en el diseño principal no se tuvieron en cuenta estas necesidades para la implementación de las operaciones sobre grafos.

A través de la enseñanza y tipos de estructuras de datos vistos en clase se pueden lograr implementar tipos abstractos de datos como lo es en este caso el TAD grafo en el cual se utilizan la gran mayoría de estructuras que son en gran medida útiles para poder implementar con precisión este tipo de abstracto de dato; además de la implementación del TAD grafo se logra la implementación de recorridos en grafos siendo realmente útiles tanto como en el proyecto como en siguientes cursos, ya que estos tipos de algoritmos son utilizados en muchas aplicaciones de diferentes índoles en la programación.