Práctica 9

Gráficas

Matú Hernández Diana

Rojo Mata Daniel

Complejidad en tiempo y espacio

- **eliminarVertice.**

```
1  public void eliminarVertice(String identificador) throws Exception {
2      Vertice verticeAEliminar = null;
3      for (Vertice vertice : vertices) {
4          if (vertice.darIdentificador().equalsIgnoreCase(identificador)) {
5              verticeAEliminar = vertice;
6              break;
7          }
8      }
9      if (verticeAEliminar != null) {
10         LinkedList<Arista> nuevasAristas = new LinkedList<>();
11         for (Arista arista : aristas) {
12             if (!arista.darU().equalsIgnoreCase(identificador) &&
13             !arista.darV().equalsIgnoreCase(identificador)) {
14                     nuevasAristas.add(arista);
15                 }
16             }
17         aristas = nuevasAristas;
18         vertices.remove(verticeAEliminar);
19     } else {
```

```
20            throw new Exception("No se encontró el vértice");
21        }
22    }
```

Complejidad en Tiempo

Acceder a los campos de un objeto no toma tiempo, sin embargo, en el bloque `if-else 4-6` se hace un comparativo[1] de una cadena con el `identificador` que se ingresa como parámetro. Esto es, hay una operación elemental. Luego, en la línea 5, se hace una asignación, por lo que, dentro del bloque antes mencionado, se realizan 2 operaciones elementales.

Las operaciones mencionadas se realizan en el cuerpo de un ciclo de la forma `for(Vertice vertice : vertices)` (líneas 3-8), esto quiere decir que se realizan a lo más, el número de elementos que tiene la lista *vertices*; esto es, el grado de la gráfica.

Es por ello que, en el peor de los casos, el ciclo se ejecuta n veces, siendo n el grado de la gráfica.

Es así que hasta este punto, se tiene el siguiente número de operaciones:

$$T_1(n) = 2n + 1$$

Para el bloque `if-else 9-21` se realiza lo siguiente.

La línea 9 hace un comparativo, por lo que se tiene una operación elemental. La condición de la línea 12 y 13 contabiliza un caso comparativo (como en el bloque anterior) y una negación. Estas operaciones las realiza en 2 ocasiones, además de que se usa el operador &&. Es así que se registran 5 operaciones. Nuevamente las operaciones antes mencionadas se realizan en el cuerpo de un ciclo de la forma `for(Vertice vertice : vertices)` (líneas 11-16), esto quiere decir que se realizan a lo más, el número de elementos que tiene la lista *vertices*; esto es, el grado de la gráfica.

En la línea 14 se hace una asignación y en la línea 18 se hace uso de la función `add` implementada de `LinkedList`.

Finalmente, en las líneas 17 y 18, se hace una asignación y se utiliza la función `remove`.

[1] Formalmente se tendrían que contar el número de operaciones elementales que realiza la función `equalsIgnoreCase`, la cual compara dos strings para ver si son iguales ignorando las diferencias entre mayúsculas y minúsculas. Se está suponiendo que esto contabiliza solo una operación elemental.

Se tiene:

$$T_2(n) = 1 + 5(n + T_{add}) + 1 + 1 + T_{remove}$$

Es así que el tiempo de ejecución final es de la forma:

$$T(n) = T_1(n) + T_2(n) = 2n + n(5 + T_{add}) + 3 + T_{remove}$$

$$T(n) = 7n + 3 + 5T_{add} + T_{remove}$$

Siendo T_{add} y T_{remove} el tiempo de ejecución de las funciones `add` y `remove` respectivamente. [2]

Considerando que $T_{add} \in O(1)$ y que $T_{remove} \in O(n)$, se tiene que: $T(n) \in O(n)$.

Complejidad en Memoria

En la línea 2 del algoritmo se crea un nuevo nodo denominado `verticeAEliminar` y se inicializa con `null`. Esto, al ser un nodo externo creado, consume solo un espacio de memoria adicional. En la línea 10 se crea una lista ligada llamada `nuevasAristas`, en donde en el peor de los casos, ésta consta de `n` elementos, es decir, de todos los vértices de la gráfica.

De esta manera, la función que mide la complejidad en memoria total es de la forma:

$$M(n) = n + 1$$

De aquí se concluye que $M(n) \in O(n)$, es decir, la complejidad en memoria es lineal.[3]

[2] Por lo realizado en la práctica 6, se llegó a la conclusión de que el método `remove` (eliminar) tiene complejidad en tiempo del orden lineal, mientras que `add` (insertar), orden constante; esto es $O(n)$ y $O(1)$ resp.

[3] Por lo realizado en la práctica 6, se llegó a la conclusión de que el método `remove` (eliminar) y `add` (insertar) tienen complejidad de memoria $O(1)$, es por ello que no altera el resultado final.

- **eliminarArista**

```
1  public void eliminarArista(String u, String v) throws Exception {
2      Arista aristaAEliminar = null;
3      for (Arista arista : this.aristas) {
4          if ((arista.darU().equalsIgnoreCase(u) &&
5              arista.darV().equalsIgnoreCase(v)) ||
6              (arista.darU().equalsIgnoreCase(v) &&
7              arista.darV().equalsIgnoreCase(u))) {
8              aristaAEliminar = arista;
9              break;
10         }
11     }
12     if (aristaAEliminar != null) {
13         this.aristas.remove(aristaAEliminar);
14         this.tamano--;
15     } else {
16         throw new Exception("No se encontró la arista");
17     }
18 }
```

Complejidad en Tiempo

En el bloque `if 4-10` se realizan 4 comparaciones del tipo `equalsUgnoredCase`, 2 operaciones del tipo && y 1 como ||, dando un total de 7 operaciones y una asignación en la línea 8. Dentro del bloque se realizan 8 operaciones elementales. Éstas se harán a lo más un número n de veces al estar dentro de un ciclo de la forma `for(Vertice vertice : vertices)`.

En este caso, se realizan $8n$ operaciones, dentro del ciclo 3-11.

Ahora, en el bloque `if 12-15` se realizan 3 operaciones elementales (comparación con `null` y disminución `-`, que cuenta como 2) y se aplica el método `remove`. Por lo que en general se tienen: $T_{remove} + 3$ operaciones.

Es así que el tiempo de ejecución es de la forma:

$$T(n) = 8n + T_{remove} + 3 = 9n + 3$$

Pero como $T_{remove} \in O(n)$ se concluye que $T(n) \in O(n)$. Esto es, el tiempo de ejecución es lineal.

Complejidad en Espacio

En la línea 2 se crea una nueva arista llamada `aristaAEliminar`, tomando un elemento extra de memoria, mientras que en la línea 13 se utiliza el método `remove`.

Por lo que la función $M(n)$ es de la forma:

$$M(n) = c$$

Esto es, la complejidad en espacio es constante, $O(1)$.

- **darVecindad**

```
1  public LinkedList<String> darVecindad(String identificador) throws Exception {
2      LinkedList<String> vecinos = new LinkedList<>();
3      if (buscarVertice(identificador)) {
4          for (Arista arista : aristas) {
5              if (arista.darU().equals(identificador) ||
6                  arista.darV().equals(identificador)) {
7                  String vecino = arista.darU().equals(identificador) ?
8                  arista.darV() : arista.darU();
9                  vecinos.add(vecino);
10             }
11         }
12     } else {
13         throw new Exception("No se encontró el vértice");
14     }
```

```
15      return vecinos;
16  }
```

subsection*Complejidad en Tiempo

En el bloque `if 3-11` se realizan 3 comparaciones del tipo `equals`, 1 operación del tipo ? y 1 como ||, dando un total de 5 operaciones y una asignación en la línea 7, esto significa que dentro del bloque se realizan 6 operaciones elementales. Éstas se harán a lo más un número m de veces al estar dentro de un ciclo de la forma `for(Arista arista : aristas)`. Siendo m el tamaño de la gráfica.

En este caso, se realizan $6m$ operaciones, dentro del ciclo 3-11. Sin embargo, se debe contabilizar la función $T_{buscarVertice}$[4]

Es así que el tiempo de ejecución es de la forma:

$$T(n) = 6m + T_{buscarVertice}$$

Pero como $T_{remove} \in O(n)$ y se sabe que $m \leq \frac{n(n-1)}{2}$, entonces, a lo sumo, $T(n) \in O(n^2)$.

Complejidad en Espacio

En la línea 2 se crea una lista ligada llamada `vecinos`, en donde en el peor de los casos, ésta consta de `n` elementos, es decir, de todos los vértices de la gráfica. Además de que se utiliza la función `add`, pero ésta, presenta complejidad $O(1)$.

De esta manera, la función que mide la complejidad en memoria total es de la forma:

$$M(n) = n$$

De aquí se concluye que $M(n) \in O(n)$, es decir, la complejidad en memoria es lineal.

[4]Sin entrar en detalle, esta función pertenece a $O(n)$, pues en esencia, este método verifica si existe un vértice con el identificador dado en la gráfica mediante un ciclo for.

- **darGrado**

```
1  public int darGrado(String identificador) throws Exception {
2      if (buscarVertice(identificador)) {
3              LinkedList<String> vecindad = darVecindad(identificador);
4          return vecindad.size();
5      } else {
6          throw new Exception("No se encontró el vértice");
7      }
8  }
```

Complejidad en tiempo

Comienza con una declaración if que llama a buscarVertice(identificador) para verificar si existe un vértice con el identificador proporcionado en el grafo. Esto tiene complejidad en tiempo de 1, es decir es constante. Si se encuentra el vértice, se llama al método darVecindad(identificador) que devuelve una lista enlazada (LinkedList) de identificadores de los vértices adyacentes al vértice con el identificador dado. Esto también tiene complejidad en tiempo constante de 1. Luego, se devuelve el tamaño de la lista vecindad, que representa el grado del vértice, es decir, la cantidad de aristas que inciden en él. Si no se encuentra el vértice con el identificador dado, se lanza una excepción con el mensaje "No se encontró el vértice". Hasta este punto sin tomar en cuenta los métodos llamados la complejidad es constante. Sin embargo se deben de tomar en cuenta y como sabemos el metodo buscarVertice(identificador) pertenece a una complejidad en tiempo lineal O(n) y el metodo darVecindad(identificador) tiene complejidad en tiempo cuadratico $O(n^2)$. Con la información anterior podemos concluir que la complejidad en tiempo de este metodo será de $T(n) \in O(n^2)$.

Complejidad en espacio

En la linea 3 se hace el uso de la variable vecindad, la cual es una lista enlazada que esta haciendo uso de memoria, así que la complejidad en enpacio es de $T(m) \in O(deg(v))$

- **darVertice**

```
1  public Vertice darVertice(String identificador) throws Exception {
2      if (buscarVertice(identificador)) {
3          Vertice verticeBuscado = null;
4          for (int i = 0; i < this.vertices.size(); i++) {
5              if (this.vertices.get(i).darIdentificador().
6                  equalsIgnoreCase(identificador)) {
7                  verticeBuscado = this.vertices.get(i);
8                  break;
9              }
10         }
11         return verticeBuscado;
12     } else {
13         throw new Exception("No se encontró el vértice");
14     }
15 }
```

Complejidad en tiempo

El método inicia con una declaración if que llama a buscarVertice(identificador) Esto verifica si existe un vértice con el identificador en la colección de vértices. Si es verdadero, continúa con la búsqueda del vértice. Después se declara una variable verticeBuscado inicializada como null, que se utilizará para almacenar el vértice encontrado. Esto tiene una complejidad en tiempo constante de 1. Ya que se tiene donde almacenar el vértice se inicia un bucle for el cual recorrerá todos los vertices. Está línea de código tiene una complejidad en tiempo de 5. Dentro del bucle, compara si el identificador del vértice en la posición i coincide con el identificador proporcionado como argumento. Si la comparación es verdadera, asigna el vértice actual a la variable ‘verticeBuscado‘ y sale del bucle. Si no se encuentra un vértice con el identificador dado, se lanza una excepción con el mensaje "No se encontró el vértice". Así que la funcion de complejidad en tiempo es:

T(n) = n+7

Y como sabemos $T(n) \in O(n)$. Lo cual es tiempo de ejecución lineal.

Complejidad en espacio

En este motodo no se está ocupando ningun tipo dato para guardar cosas, lo unico que se creo fue una variable ,así que T(m) = 1, por lo tanto la complejidad en memoria es constante.