Trabajo Práctico Especial
2da Entrega

Programación III - TUDAI

Facultad de Cs. Exactas
Univ. del Centro de la Pcia. de Bs. As.

GRUPO: **IGARTUA, S. Lorena**
lorenaigartua@gmail.com

MUJICA, Martin S.
mujicamartin@gmail.com

Tandil, 06 de junio de 2018

INTRODUCCIÓN:

Conforme el requerimiento, la herramienta a diseñar debe ser útil para realizar un análisis del comportamiento de los usuarios y sus gustos relacionados con los géneros o categorías de libros que lee.
Nos entregaron 4 archivos .csv, que contenían distintas cantidades de consultas realizadas. Así, por cada fila, nos encontrábamos con una cierta cantidad de géneros de Libros que habían sido consultados por un usuario en particular, y la secuencia en la que habían sido indexados.
Con esta información, se debía implementar una estructura tipo Grafo, en el que cada genero iba a ser representado por un Nodo o Vértice, y el Peso o la Arista hacia el siguiente genero consultado indicaría la cantidad de veces que fue consultado inmediatamente a continuación.
Armada la estructura, se requería implementar tres tipos de servicios, a saber:
- Obtener los N géneros más buscados luego de buscar por el género A.
- Obtener todos los géneros que fueron buscados luego de buscar por el género A.
- Obtener el grafo únicamente con los géneros afines, es decir, que se vinculan entre sí (pasando o no por otros géneros).

DESARROLLO DEL TRABAJO:
El primer paso, y la primera decisión que se debía tomar, era modelar el grafo que contendría la información contenida en los archivos .cvs, considerando que debía ser lo suficientemente escalable, en tanto el ultimo archivo contenía un millón de búsquedas realizadas, y mas de cinco millones de géneros consultados en distinto orden.

Para implementar el grafo de tipo dirigido solicitado, se analizaron 2 de las representaciones más usuales: lista de adyacencias y matrices de adyacencias.
Con una matriz de adyacencia, podemos averiguar si una arista está presente en un tiempo constante, solo buscando la entrada correspondiente en la matriz, luego de haber obtenido la ubicación del Nodo en cuestión. Esto tiene como contracara, que ocupa un espacio [n] * [n], o sea, O(n^2), situación que es aún más desventajosa, cuando el grafo es disperso (con relativamente pocas aristas), como es el caso de las primeras secuencias de consultas de

usuarios que debemos analizar. En otras palabras, para un grafo disperso utilizamos mucho espacio para representar solo algunas aristas. Otra desventaja es que, si necesitamos averiguar que vértices son adyacentes a un nodo X, debemos consultar todas las entradas que lo incluyen ([n]), incluso si solo un pequeño número de vértices son adyacentes al nodo buscado.

Por otro lado, en la lista de adyacencia, se puede acceder a cada nodo en un tiempo constante, porque solo deberíamos indexar en un arreglo. Para averiguar si una arista (X,Y) está presente en el grafo, vamos a la lista de adyacencia de X en un tiempo constante y luego buscamos Y en su lista de adyacencias. Esto, en el peor de los casos, tiene una demora de O(d), siendo d la cantidad de adyacencias de X. La principal ventaja de utilizar lista de adyacencias, es que requiere un espacio proporcional a la suma del número de nodos más el número de aristas, haciendo un buen uso de la memoria.

Luego de este análisis de ventajas y desventajas, decidimos implementar nuestro grafo, representándolo con Listas de Adyacencias.

Se diseño la clase Nodo, que contendría, además del género, una lista de Aristas, que va a representar la cantidad de géneros consultados, en diversas oportunidades, a continuación de la categoría representada por el nodo en cuestión.

Luego, se diseño la clase Arista. Y aquí se planteó otra disyuntiva. Si la Arista debía representar el peso hacia un Nodo, o hacia un String (que sería el parámetro para indexar el Nodo en cuestión). Nos inclinamos por el atributo String, que representaría el género destino o genero inmediato siguiente consultado. Esta decisión se baso en que el usuario de la herramienta, para realizar consultas, utilizaría un String.

Por último, la clase Grafo, que va a estar dada por una lista de Nodos. El método principal aquí, lo definimos insertarGenero (String genero1, String genero2).

```
public void insertarGenero ( String genero1, String genero2) {
            if (!existeNodo(genero1))
                  this.insertarNodo(genero1);
             if (!existeNodo(genero2))
                  this.insertarNodo(genero2);
             if (!this.getVertice(genero1).contieneLaArista(genero2)) {
                  this.getVertice(genero1).insertarConexion(genero2);
             }
             else {
                  this.getVertice(genero1).incrementarPeso(genero2);
             }
          }
```

Este método incluye una serie de paso, a saber:

1ro, controlaría que en el Grafo, existiera un Nodo cuyo getGenero() coincida con el String ingresado como parametro1. De no ser así, lo crearía e insertaría en la lista de Nodos del Grafo.
2do, mismo control y comportamiento, con el String ingresado como parametro2.
3ro, ya seguros de que los Nodos existen, se controla, luego, la existencia de una Arista desde el Nodo origen al Nodo destino. Si no existe la conexión, se inserta, creando la Arista con peso 1, y agregándola al Nodo origen.
4to, si la relación ya existía, el peso se aumenta en 1, a fin de que represente y contemple la nueva serie de consultas efectuada por el usuario.

Ya con el grafo en funcionamiento, y preparado para almacenar las consultas almacenadas en los archivos administrados, procedimos a implementar los servicios solicitados, a saber:
- Obtener los N géneros más buscados luego de buscar por el género A.
Para implementar este servicio, se analizaron diversas alternativas. Una es la que sigue:

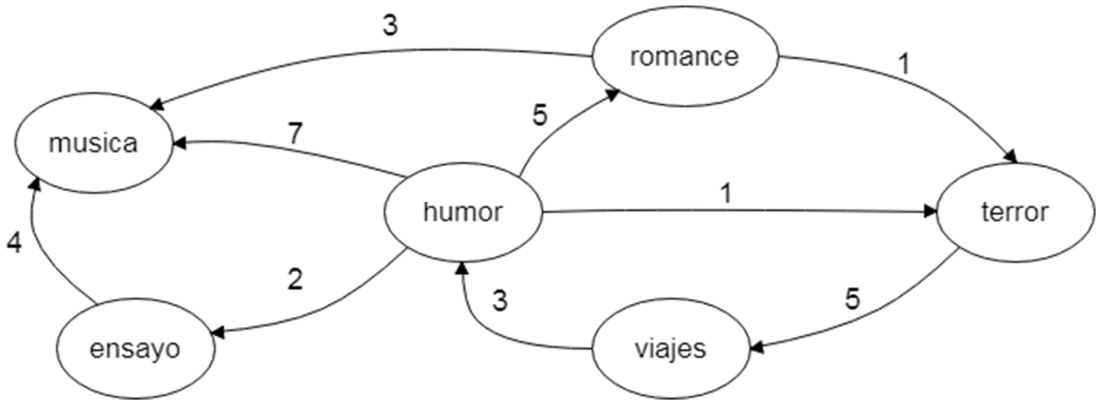
```
public void obtenerNGenerosSiguientes (String nodo, int n) {
	ArrayList<Arista> aux = this.getVertice(nodo).getConexiones();
	System.out.println( "Los "  + n + " generos mas buscados luego de " + nodo
+ " son:");
		if (n > aux.size())
			n = aux.size();
		for (int i = 0; i < n; i++) {
		    Arista max = null;
			for (Arista a : aux ) {
				if  (max == null)
					max = a;
				if (a.getPeso() > max.getPeso() )
							max = a;
				}
		System.out.println( max.getDestino() + ": " + max.getPeso() + "
veces.");
				aux.remove(max);
			}
		 }
```

Este procedimiento, iba a recorrer la lista de adyacencias del genero requerido, tantas veces como el tamaño de N ingresado, y retornaría, en cada iteración, el genero de mayor peso presente en el listado analizado.
Se concluyo que, si bien en una cantidad de datos reducida, y en un grafo disperso, con nodos con pocas aristas, esta función no generaría mayores costos, cuando esas condiciones cambien, el costo O (N*n) afectaría el rendimiento.
Por esto, se optó por una opción mas eficiente, que primero ordene la lista en función del peso registrado en cada arista, e imprima por pantalla, las posiciones hasta la cantidad N ingresada en la consulta.

```
public void obtenerNGenerosSiguientes (String nodo, int n) {
	LinkedList<Arista> aux = this.getVertice(nodo).getConexiones();
	ordenarLista (aux);

	System.out.println( "Los "  + n + " generos mas buscados luego de " + nodo
+ " son:");
		if (n > aux.size())
			n = aux.size();

		for (int i = 0; i < n; i++) {
			System.out.println( aux.get(i).getDestino() + ": " +
aux.get(i).getPeso() + " veces.");
		}
 }
```

Con esto, en un grafo como el que sigue, y la consulta:

`obtenerNGenerosSiguientes (“humor”, 3)`



El método va, en un primer paso, ordenar la lista de adyacencias del Nodo humor, de esta manera:

Arista String Música peso 7

Arista String Romance peso 5

Arista String Ensayo peso 2

Arista String Terror peso 1

E imprimiría en pantalla los 3 mayores: Música, Romance y Ensayo.

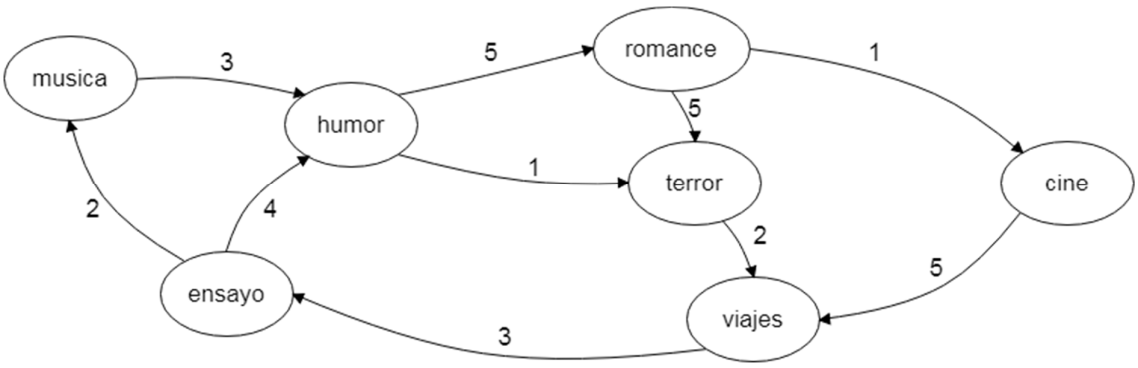
- Obtener todos los géneros que fueron buscados luego de buscar por el género A.

```
public void obtenerTodosLosGeneros(String nodo) {
        this.setearEstadoVertices();

        LinkedList<Nodo> hijos = obtenerHijos(nodo);
              for (Nodo n : hijos ) {
                   System.out.print(n.getGenero() + ", ");
             }
}


private LinkedList<Nodo> obtenerHijos(String nodo) {
        this.setearEstadoVertices();

        LinkedList<Nodo> cola = new LinkedList<Nodo>();
        LinkedList<Nodo> retorno = new LinkedList<Nodo>();
        this.getVertice(nodo).setEstado("AMARILLO");
        cola.add(this.getVertice(nodo));

        while (!cola.isEmpty()) {
              Nodo aux = cola.remove(0);
              retorno.add(aux);
                    for (Arista a : aux.getConexiones()) {
                          Nodo actual = this.getVertice(a.getDestino());
                                if (actual.getEstado() == "BLANCO") {
                                      actual.setEstado("AMARILLO");
                                      cola.add(actual);
                                    }
                          }
             }
       return retorno;
}
```

Este método hace un recorrido en anchura (BFS), partiendo del Nodo relacionado con el String ingresado como parámetro. El recorrido comienza, entonces, en ese elemento raíz seleccionado, y se exploraran todos los vecinos de este nodo. A continuación, para cada uno de los vecinos, se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra toda la estructura.

Con esto, en un grafo como el que sigue, y la consulta:

obtenerTodosLosGeneros (“humor”)



El método va, en un primer paso, a agregar en su cola, al Nodo relacionado con el String Humor. La cola va a tomar ese elemento, y por cada uno de sus Nodos adyacentes, les cambiara su estado (para evitar sean visitados en otro turno), y los agregara en la cola, para, luego, analizar a sus adyacentes. En tanto la cola de trabajo no esté vacía, significara que hay más Nodos pendientes de visitar.
Por pantalla, se imprimirá lo siguiente:
Humor
Romance, Terror ----→ que representan el primer anillo de adyacencias
Cine, Viajes ----→ que representan el segundo anillo de adyacencias
Ensayo ----→ que representan el ultimo anillo de adyacencias

- Obtener el grafo únicamente con los géneros afines, es decir, que se vinculan entre sí (pasando o no por otros géneros).

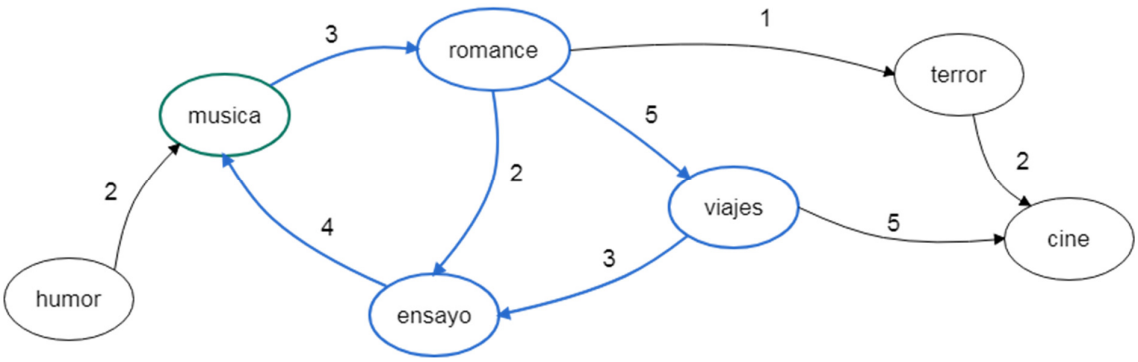
```
public  void obtenerGenerosAfines (String origen) {
       this.setearEstadoVertices ();
       LinkedList<Nodo> cola = new LinkedList<Nodo>();
       LinkedList<String> afines = new LinkedList<String>();

       cola = obtenerhijos(origen);

       while (!cola.isEmpty()) {
           Nodo aux = cola.remove(0);
           if ( this.esAfin(origen,aux.getGenero(),retorno))
                afines.add(aux.getGenero());
         }
       for (String n : afines ) {
              System.out.print(n + " ,");
       }
}
```

Este método tiene varias fases. Primero, con el método privado obtenerHijos(origen), arma una cola de Nodos que debe analizar, por estar dentro del recorrido en anchura (BFS) del nodo indicado como parámetro. Luego el método comienza a analizar, uno por uno, esos Nodos, determinando su afinidad o no con el género origen (si son afines, se agregan a una lista de retorno, que luego se plasmara por pantalla).
Para determinar su afinidad o no con el Genero buscado, se debe analizar si el Nodo actual tiene, por sí mismo o por sus vecinos, adyacencia con el Genero Origen (se verifica con el método esAfinPorOrigen(origen,nodo)), o con alguno de los Géneros que ya resultan

adyacentes a este, almacenados en la lista retorno (se verifica con el método esAfinPorHijo(nodo,afines)).

Con esto, en un grafo como el que sigue, y la consulta:

```
obtenerGenerosAfines (“musica”)
```



El método va, en un primer paso, a agregar en su cola, todos los Nodos relacionados con el String Musica, a saber: romance, ensayo, viajes, terror, cine.

Por cada uno de estos Nodos, va a analizar su afinidad con Música. Por ejemplo, Romance, no es esAfinPorOrigen(“musica”, “romance”), ni esAfinPorHijo (“romance”, afines)), por eso, va a analizar en profundidad, cada uno de sus adyacentes, o sea, Viajes y Ensayo y Terror, y así sucesivamente. Cuando sea el turno de analizar esAfinPorOrigen(“musica”, “ensayo”), como es afín, se agregara a la lista de afines. Cuando analice esAfinPorHijo (“viajes”, afines)), como unos de los adyacentes de Viajes es Ensayo, y esta dentro de la lista de afines, también será incluido en este listado.

CONCLUSIÓN:

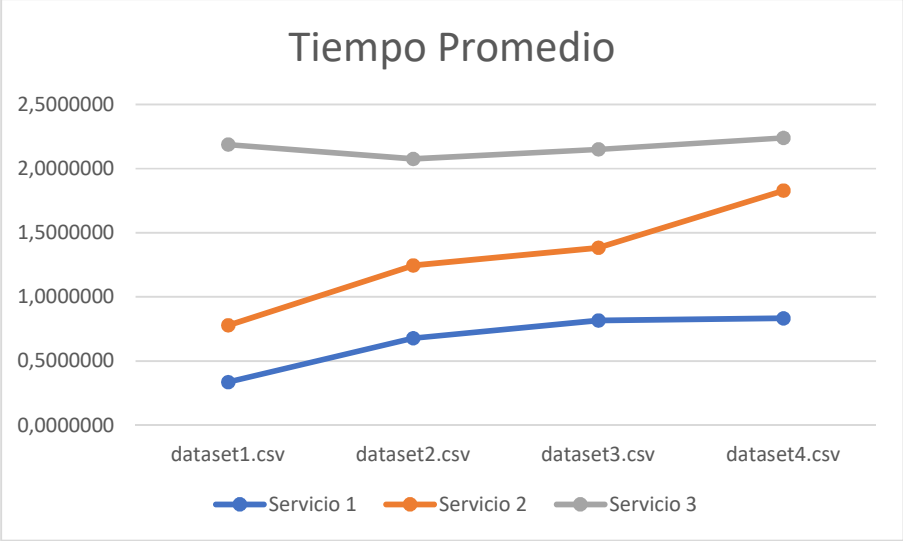
Este tipo de estructuras de datos, fuera de la complejidad que implica su implementación, reporta beneficios durante el procesamiento de los datos, el cual es altamente eficiente.

Los archivos .csv administrados, iban aumentando la cantidad de consultas de usuarios, desde 20 secuencias, en el dataset1, a 1.000.000 en el dataset4.

Sin embargo, sin considerar la demora de la aplicación en generar la estructura del grafo, una vez éste estaba preparado para ser consultado, no se perciben grandes diferencias en el costo de resolver los servicios de búsqueda implementados, independientemente del peso de las aristas. Se agrega a continuación, cuadro comparativo del tiempo promedio (luego de 10 iteraciones), en nanosegundos, que conlleva cada uno de los 3 servicios implementados, y por cada archivo .csv analizado.

Archivo	Servicio 1 buscarNMayores(humor,15)	Servicio 2 buscarSiguientes(humor)	Servicio 3 buscarAfines(humor)
dataset1.csv	0,3361960 nanosegundos	0,7782700 nanosegundos	2,1878470 nanosegundos
dataset2.csv	0,6771350 nanosegundos	1,2448370 nanosegundos	2,0756500 nanosegundos
dataset3.csv	0,8158000 nanosegundos	1,3823180 nanosegundos	2,1495260 nanosegundos
dataset4.csv	0,8331830 nanosegundos	1,8275520 nanosegundos	2,2392060 nanosegundos

Tal como se observa, no existe una gran diferencia de tiempo consumido entre el dataset1 y el dataset4, por ejemplo, considerando que hay una diferencia de 999.980 de registros almacenados.



La estructura seleccionada proporciona un eficiente uso de memoria. Esto será más palpable, cuantos más datos debamos procesar.