

31 de Julio de 2021



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

PROYECTO FINAL DE ESTRUCTURAS DE DATOS

AUTOR:

Axel Daniel Malváez Flores

Título del proyecto :

MOTOR DE BÚSQUEDA

Profesor : Pedro Ulises Cervantes González

Ayudantes : Yessica Janeth Pablo Martínez, América Monserrat
García Coronado, Emmanuel Cruz Hernández, Adrián Felipe
Vélez Rivera

NOTA

Los pasos a seguir para la ejecución del programa y los pasos se encuentran detalladamente en el README.

Explicación de los archivos y carpetas que contiene la carpeta *ProyectoFinalMalvaezDaniel*:

- build.xml
El archivo que nos permitirá ejecutar el programa sin la GUI, para ejecutar la Interfaz Gráfica vea el siguiente punto.
- interfaz/
Esta es la carpeta que contiene todo lo necesario para ejecutar la interfaz del proyecto, dado que no pude ejecutar todo con ANT sin la necesidad de tener instalado el IDE NetBeans (pues requería librerías en NetBeans y no sé si usted que está leyendo tenga el IDE), es por eso que se las incluí ya compilada y hecha .jar aquí, por lo que para ejecutar la GUI solo basta entrar a *interfaz/MiniGoogle.jar* aquí simplemente dan doble click en el archivo *MiniGoogle.jar* y automáticamente se ejecutará el GUI, Además dentro de esta carpeta incluí las carpetas *lib/* y *InterfacesJavaProjectCodigo/*, en la primera se encuentra un archivo con extensión .jar que sin éste mismo, no se abre la interfaz gráfica, en la segunda carpeta es en donde podrán encontrar el código completo y adaptado para la interfaz.
- README.md
README del proyecto, implementación y los pasos a seguir.
- src/
Carpeta donde se encuentran alojadas las clases y los documentos necesarios para compilar el programa este sin GUI.

ESTRUCTURAS

- ArrayList<K>
La siguiente estructura fue utilizada debido a su implementación basada en arreglos, pues el acceder a un elemento tiene complejidad $O(1)$ y dado que estaremos accediendo a cada término de listas con longitud inimaginable, es por eso que nos sirve muchísimo esta estructura y así ahorrarnos el tiempo de recorrerla para encontrar algún elemento. Además dado que agregar al final se hace de manera constante, y no requerimos en el proyecto de agregar elementos en el medio de una lista, esta operación tiene una complejidad $O(1)$, así las únicas operaciones que necesitamos para el proyecto se hacen en $O(1)$, es por eso que decidí utilizar esta estructura.
 1. Reader class:
El ArrayList fue muy utilizado al leer los documentos, pues, cada palabra que se encuentra en el texto será guardada en un "Nodo" o en una "casilla", así al agregar al final en la lista, dado que se hace de manera constante, el tiempo solo se ve afectado por la cantidad de veces que agregaremos al final, es decir el número de palabras que ese encuentran en el documento, es así que el pasar un texto a una lista se hace de manera $O(n)$, sobre el número de palabras.
 2. TFIDFcalculator class:
Dentro de esta clase y en la mayoría de las clases, utilicé los ArrayLists<k>para interpretar los documentos y dado que en la clase Reader ya pasamos un documento a una lista, esta idea siempre se mantuvo, por lo que en la mayoría de métodos se utilizó un Arreglo de ArrayLists<K>para hacer alusión con el arreglo al conjunto de archivos y el ArrayList<K>a un simple documento de ellos.
- Hashtable<K, V>
Al principio había utilizado un RedBlackTree para almacenar el valor IDF de cada término en todos los documentos, sin embargo este proceso era muy tardado, dado que agregar y buscar se hacía en $O(\log n)$ decidí utilizar un Hashtable<K, V>ya implementado por java para así poder encontrar un elemento dada su clave en $O(1)$ o bien agregar igualmente en tiempo $O(1)$. Dado que todas sus operaciones se hicieron

en `TFIDFcalculator` class, entonces su clave del `Hashtable` es un `String`, es decir una palabra y su valor almacenado es un objeto de tipo `Pair<String, Double>` siendo el `String` la misma palabra y el `Double` su valor `IDF` previamente ya calculado, para posteriormente al momento de añadir, simplemente verifique si la palabra ya ha sido almacenada y de no ser así la almacena. Dado que todo este proceso lo hace tantas veces como palabras tengamos todos los documentos, entonces su complejidad es $O(n)$. Es por ello que decidí hacer este cambio de `RedBlackTree` a `Hashtable`.

■ `Pair<V, K>`

La primer estructura implementada fue una tupla, clase implementada por mí (basada en la clase `Pair<V, K>` de java). Esta clase es una clase genérica y por lo tanto almacena dos elementos de cualquier tipo, es así que fue muy utilizada en la mayoría de las clases, para asociar un valor con una respectiva clave al estilo `Hashtable`, sin embargo en este caso era un único objeto.

1. `TFIDFcalculator` class:

Dado que cada documento fue procesado y almacenado en una lista, se utilizó para asociar a cada palabra con un valor de verdad. Posteriormente analizando este valor de verdad que estaba asociado con cada palabra se analizó la ocurrencia de cada palabra en un documento.

Es así que al momento del cálculo del `IDF` se utilizaron objetos de tipo `Pair<String, Double>` para guardar el valor `IDF` de cada término ya fuese del documento o de la consulta.

En el `TF` igual se utilizó la misma idea de asociar el valor `TF` con cada término del documento o de la consulta.

Y finalmente para el cálculo `TF-IDF` igual se utilizó para asociar el término con su valor `TF-IDF` en los documentos o en la búsqueda.

Para la similitud se utilizó un arreglo de tipo `Pair` para asociar cada documento con la similitud dada una búsqueda.

Se utilizó por su versatilidad para la manipulación y obtención de datos así como para poder asociar términos con algún valor, ya sea ocurrencia en el término, o sus respectivos valores `TF`, `IDF` o `TF-IDF`.

CONCLUSIÓN

En general me sentí muy bien realizando este proyecto y el hecho de haberme animado a hacer la GUI, me alentó a seguir haciendo proyectos por mi cuenta. Siento que las estructuras que utilicé en la segunda versión de mi proyecto, ayudó a mejorar el tiempo de ejecución en los casos grandes y por tanto me siento bien de haber entendido los conceptos y saber cuándo conviene una estructura sobre otra, si bien me hubiera gustado agregar un `MAXHEAP`, debido al tiempo que me hubiera tomado implementarlo, no puede agregarlo y creo que este le hubiera bajado aún más la complejidad pues hubiera sustituido el tener `ArrayLists` y en uno ordenarlo y pasar los 10 primeros al segundo, pues implementé `insertionSort`.

