# Práctica 3 BMI

## Memoria

12/04/2020

Adrián Rubio Pintado
Jorge Muñoz Aguado

A continuación, se explicarán las tareas realizadas.

# Tarea 1 - Motor de búsqueda proximal.

En esta tarea se nos pide crear un motor de búsqueda proximal, el cual tenga en cuenta la distancia entre las palabras.

## Tarea 1.1 - Búsqueda proximal.

La programación de este código se ha basado en el algoritmo visto en clase de teoría. Para ello, antes es necesario *parsear* la entrada y generar las estructuras de las que se vaya a hacer uso.

Además, se han creado dos funciones que facilitarán la implementación del código. La primera función `getMaxPosition` busca la posición más alta dentro en la que todos los términos están incluidos. La segunda de función, `avanzaHastaValor`, avanza los valores de los valores de los términos hasta un valor dado, en este caso se corresponderá con *b*. También se encarga de devolver el mínimo.

El algoritmo consiste en que primero se busca la posición máxima de los valores con la función `getMaxPosition`, lo cual será el primer valor de *b*. Tras esto, se entra en un bucle que no acaba hasta que *b* no pueda tomar un valor. En este bucle se calculará *a* con la segunda función `avanzaHastaValor`. Y a continuación se calculará el *score* con una tercera función adicional `calculaScore`.

Los resultados del ranking se guardarán en la clase `RankingImpl` y serán devueltos.

## Tarea 1.2 - Búsqueda literal.

La búsqueda literal debe buscar las palabras en el orden determinado y deben estar juntas. Esta segunda característica se consigue comprobando que el score parcial de cada rango *a* – *b* tiene el valor 1.

La primera característica mencionada se comprueba en la función `avanzaHastaValor`, donde se comprueba que las posiciones de palabras sigan un orden estricto.

Se hará uso del mismo código que la *búsqueda proximal*, pero añadiendo las características anteriormente descritas. También será necesario *parsear* la entrada con tal de encontrar las dobles comas, que exigirían que se utilizara esta *búsqueda literal*.

# Tarea 2 - Índice posicional.

Para la implementación del índice posicional (y su builder) se ha tomado como referencia y adaptado el índice SerializedIndexRAM proporcionado en los archivos de la práctica 3.

Las clases creadas para este apartado han sido:

**PositionalIndex (RAM)**extiende de AbstractIndex. Utiliza las estructuras para manejar postings posicionales para implementar un índice posicional. En esencia, es una modificación del índice SerializedIndexRam convertido en índice posicional.

**PositionalIndexBuilder** Constructor del indice. Extiende de AbstractIndexBuilder.

Y como adaptación para las estructuras posicionales necesarias:

**PositionalPosting**: Clase que hereda de Posting. Alamcena la lista de posiciones de un término en un documento.

**PositionalPostingList** Estructura de lista de PositionalPostings. Implementa la interfaz PostingList y gestiona con coherencia las inserciones/actualizaciones de datos.

**PositionalDictionary** Implementa la interfaz Dictionary. Estructura que mapea términos con listas de positional postings(PositionalPostingList). Gestiona coherentemente las inserciones/actualizaciones de datos como le corresponde.

*Se pueden ver sus relaciones más detalladamente en el diagrama de clases adjunto.*

# Tarea 3 - PageRank.

*PageRank* es un algoritmo que busca calificar la relevancia de las páginas webs en la búsqueda. Para ello hace uso de una estructura de grafo, con la cual calculará los rankings a cada nodo del mismo.

La codificación del algoritmo se ha basado en el pseudocódigo cedido por los profesores en las presentaciones de teoría.

También se ha creado una clase *Graph* que permite guardar la información necesaria para posteriormente calcular los scores.

En el constructor, primero se leerá el fichero con la función `loadGraph`. Esta función devolverá un objeto *Graph* con la información extraída del fichero. Este grafo se pasará a la función `calculateScore`, la cual calculará el score de cada vértice haciendo uso del pseudocódigo anteriormente mencionado. El resultado de esta función será un *HashMap* con los scores.

El resto de las funciones harán uso de la información generada por el constructor para dar una respuesta.

# Tarea 4 - Crawling.

Para este ejercicio se ha implementado la clase *WebCrawler*. Se incluye un tester de demostración de su funcionamiento en el archivo *WebCrawlerTester*.

Para la implementación, se ha usado la estructura de grafo implementada en el ejercicio anterior (*Graph*), para construir el grafo de descubrimiento. El crawler toma como constructor, entre otros, un IndexBuilder, para construir el índicen y "maxDocsToIndex", el número máximo de documentos que pasará para que se indexen.

En la implementación del crawler, se usa una cola de prioridad. Para ello se ha creado un clase simple dentro de WebCrawler: Tupla. Que almacena el path junto con la prioridad de cada enlace. Esta implementación solo se usa para la cola de prioridad. Cabe destacar, que para ganar rapidez y sencillez, como pedía el ejercicio, no se ha tomado un criterio para añadir una prioridad mayor a unos documetos sobre otros. Sin embargo, la dicha implementación, con simples variaciones, conseguiría resultados deseados con pocas modificaciones del código.

En esencia, el crawler inserta las semillas en la cola. De la cola saca el documento, lo indexa, analiza sus enlaces y añade al grafo tanto el nodo(documento), como sus enlaces a otros documentos(para ello inserta también los nodos), y los inserta además en la cola de prioridad. Este proceso se repite hasta que se alcanza el máximo de documentos a indexar, o hasta que ya no hay más enlaces.

De los problemas encontrados a la hora de construir el crawler, nos encontramos varios. Uno de ellos, es el control de duplicados. En una web, varios enlaces dentro de sí misma enlazan a la misma página. Si no hacemos control de los documentos ya recorridos, muy probablemente el crawler entre en bucle en un circuito de links. Dado que recorrer el grafo es muy costoso en cada inserción, para solucionar dicho error se creó un HashSet para almacenar a modo de lista negra los documentos ya explorados. Eso es eficiente suponiendo que dicho crawler no se usara de manera muy muy masiva.

Otra de las dificultades que se observan en el testeo del crawler, es la cantidad de enlaces dentro de un sitio web que no son deseables. Esto es, encontrar como distintos documentos uno mismo dado que tienen una vista distinta. Por ejemplo, la versión móvil o de escritorio de un mismo documento. Este problema es difícil de solucionar dado que normalmente los sitios web utilizan diferente notación en las urls con sus CGI, al generar paginas dinámicas.

Otro problema es, por ejemplo, los enlaces que son referencias a apartados de la web. Actualmente se tratan como documentos distintos (muy comúnmente en pruebas con Wikipedia).

Aquí algunos resultados de las pruebas del tester:

**PRUEBA 1 (5 semillas:urls.txt)**

---

--> 54s 886ms

Semillas: 5

Documentos indexados: 100

Numero de webs en la frontera: 9186

**PRUEBA 2 (1 semilla:reddit.com)**

---

--> 47s 370ms

Semillas: 1

Documentos indexados: 100

Numero de webs en la frontera: 7863

**PRUEBA 3 (5 semillas:urls.txt)**

---

--> 5min 35s 914ms

Semillas: 5

Documentos indexados: 1000

Numero de webs en la frontera: 68819

Observamos como el número de webs en la frontera crece con gran fuerza a medida que elevamos el límite de documentos a indexar.

En el último ejemplo, vemos como para indexar 1000 documentos a partir de 5 semillas, tardamos 5minutos y medio. Lo que nos da una idea, de como de costoso debe de ser de crawlear la web entera para los grandes buscadores.

****Se adjunta el diagrama de clases de la practica con los elementos mas representativos de la practica en el fichero "diagrama_clases_p3.png"*