MEMORIA PRÁCTICA 3

Miguel Laseca Espiga, Pablo Marcos Manchón

1. Apartados realizados e implementación

En esta práctica hemos optado por implementar todos los apartados solicitados, intentando utilizar cuanto más código del usado de partida posible. Las decisiones de diseño para el motor de búsqueda proximal y el índice posicional se describen más adelante.

1.1. Búsqueda proximal

Para la implementación del modelo de búsqueda proximal se ha creado la clase *es.uam.eps.bmi.search.proximity.ProximityEngine*, la cual extiende a *es.uam.eps.bmi.search.AbstractEngine*.

Para la búsqueda proximal en primer lugar se utiliza un Heap, el cual se utiliza para ordenar los postings obtenidos de las palabras de la consulta por docID, de manera similar a como se hacía en el modelo orientado a documentos, tras esto se calcula la puntuación basada en proximidad de los documentos que contienen todas las palabras de la consulta.

Para la implementación del algoritmo de cálculo de los intervalos mínimos se han empleado directamente los iteradores de enteros, empleando 2 listas para el almacenamiento de los índices actuales y siguientes de posiciones de cada documento.

Por simplicidad no se han tenido en cuenta palabras repetidas en una consulta, eliminando estas de la consulta.

1.2. Búsqueda Literal

Cuando se realiza una búsqueda entrecomillada utilizando el motor *ProximityEngine* para calcular la puntuación en lugar de utilizar el algoritmo de búsqueda proximal calculamos el número de veces que aparecen las palabras en orden y consecutivas en el documento.

En este tipo de búsqueda se permite que una palabra aparezca repetida en una consulta, es decir, podría realizarse la consulta “a b c a b” sin ningún tipo de problema.

2. Índice Posicional

Se han creado las clases *PositionalIndex* y *PositionalIndexBuilder* en el paquete *es.uam.eps.bmi.search.index.impl*.

Se ha reutilizado la implementación proporcionada de el índice serializado en RAM, de hecho solo ha sido necesario hacer modificaciones en el constructor del índice. Se ha tenido en cuenta que en esta práctica no se ha necesitado usar grandes colecciones de datos, por lo que la dimensión del índice nunca llega a ser preocupante como para implementarla en disco.

Para la implementación se han tenido que crear estructuras auxiliares para el almacenamiento de los índices posicionales que no dependieran de Lucene y permitieran ir añadiendo posiciones a los postings durante el parseo, para ello se han creado las siguientes clases:

- *es.uam.eps.bmi.search.index.structure.impl.PositionalHashDictionary*
- *es.uam.eps.bmi.search.index.structure.positional .PositionalPostingEditable*
- *es.uam.eps.bmi.search.index.structure.positional .PositionalPostingsListEditable*

Las cuales heredan de sus respectivos análogos no posicionales.

El diccionario con los postings posicionales se almacena en RAM utilizando la serialización de Java, reutilizando toda la lógica implementada en SerializedRAMIndex.

2. Page Rank

Hemos implementado el algoritmo de PageRank en el paquete *es.uam.eps.bmi.search.graph* de la práctica, implementando la interfaz DocumentFeatureMap del código que se nos dió en clase.

Nuestra implementación utiliza dos tablas hash para el almacenamiento de los links, aunque para el cómputo de las puntuaciones son utilizados arrays. Utilizamos el algoritmo iterativo visto en clase, con un número máximo de iteraciones como criterio de parada. Decidimos no escalar las puntuaciones, de forma que sumen 1, para que la salida se corresponda con el fichero de output proporcionado.

3. Crawling

Más adelante también implementamos el crawler que se nos pidió en la práctica. El crawler sigue la implementación básica solicitada, utilizando una serie de URLs seed para generar un grafo que se guardará en el fichero plano que más adelante utilizará PageRank para generar el ranking del mismo. El crawler genera un índice utilizando un IndexBuilder pasado al constructor, y recorriendo los documentos enlazados por las URLs del fichero de texto plano. El procedimiento es el siguiente:

- Recorremos la lista de URLs e intentamos normalizarlas, si alguna nos falla no se añadirá al Index generado. La normalización nos permitirá no reindexar páginas entrando en bucles.
- En cuanto encontremos una URL válida, usando Jsoup recorremos el documento en busca de links a otras páginas que podemos incorporar al grafo. Se ha añadido un máximo de urls a añadir a la cola por página, por defecto a 10.
- Durante la exploración se van indexando los documentos y escribiendo en un fichero los enlaces entre páginas webs para poder ser utilizado por pagerank.

Para probar que el crawler funciona correctamente y medir su tiempo de ejecución para un listado de ejemplo hemos creado una clase de test adicional Crawling en el paquete *es.uam.eps.bmi.search.test*. Tras probar la ejecución comprobamos que genera correctamente un Index recorriendo los 100 documentos de prueba en un tiempo de 72,77 segundos, un rendimiento satisfactorio a nuestro juicio.

Se utiliza una cola FIFO para la exploración de páginas, por lo que el grafo crecerá siguiendo una búsqueda en anchura.

4. CombinedEngine

Generalmente en la práctica se suele combinar con otros motores de búsqueda como la búsqueda vectorial. Esta idea de combinar distintos motores de búsqueda para combinar sus ventajas es la que inspira al motor de búsqueda híbrido.

En nuestra práctica hemos implementado la clase *es.uam.eps.bmi.search.CombinedEngine*, que utiliza un conjunto de motores distintos para comparar sus búsquedas.

- En nuestra implementación cada motor de los utilizados realiza una búsqueda individualmente, y almacenamos sus distintos rankings.
- Se estandarizan las puntuaciones de los rankings se normalizan al intervalo [0,1] empleando una transformación afín.

- Finalmente se utilizan los paths para obtener identificadores de los documentos que aparecen en los rankings y poder identificar los documentos de los distintos rankigs y se asignan identificadores a los documentos para después poner obtener los paths a partir de estos utilizando la interfaz del combined engine.
- La puntuación de cada documento es la suma de las puntuaciones normalizadas en los distintos rankings.

Para probar el motor de búsqueda combinado probamos a combinar los motores de búsqueda proximal, VSM basado en documentos y estático, utilizando un índice distinto para cada uno.