MEMORIA PRÁCTICA 3 - BUSCADOR
EXPLOTACIÓN DE LA INFORMACIÓN

Alejandro García Gil 77042008N

Introducción

El código que presento a continuación lo he desarrollado como parte de un conjunto de clases que conforman la estructura principal de un buscador. Todo el sistema se basa en estructuras que yo mismo he generado previamente mediante un tokenizador, un indexador y una clase *stemmer* que también he implementado, y que se encarga de adaptar el procesamiento según el idioma del texto. Las clases que no detallo aquí ya las expliqué en prácticas anteriores.

En este documento describo las mejoras que he incorporado, las funcionalidades que he añadido a la clase, y explico los algoritmos que he diseñado e implementado para realizar la búsqueda de términos. Además, analizo la complejidad computacional (a nivel práctico) del algoritmo que he utilizado.

Análisis de la solución implementada

```
93  bool Buscador::Buscar(const int& numDocumentos){
94      unordered_map<string, InfDoc>::const_iterator iterator;
95      priority_queue<ResultadoRI> emptyQueue;
96      swap(docsOrdenados, emptyQueue);
97
98      try{
99          if(pregunta.size()<=0 || indicePregunta.size()<=0){
100             return false;
101         }
102         if(formSimilitud==0){
103             for(iterator=indiceDocs.begin(); iterator!=indiceDocs.end(); iterator++){//dfr
104                 docsOrdenados.push(ResultadoRI(SimilitudDFR(iterator->second), iterator->second.getIdDoc(), 0));
105             }
106         }else{
107             double media = 0;
108             for(iterator = indiceDocs.begin(); iterator != indiceDocs.end(); iterator++){
109                 media += iterator->second.getNumPalSinParada();
110             }//bm25
111             media=media / indiceDocs.size();
112
113             for(iterator= indiceDocs.begin(); iterator!=indiceDocs.end(); iterator++){
114                 docsOrdenados.push(ResultadoRI(SimilitudBM25(iterator->second, media), iterator->second.getIdDoc(), 0));
115             }
116         }
117     }catch(bad_alloc& e){
118         cerr<<"ERROR: Falta de memoria:"<<iterator->second<<'\n';
119         return false;
120     }
121     return true;
122 }
123
```

El método Buscar que he desarrollado se encarga de recorrer las estructuras que previamente he indexado para clasificar los documentos según la similitud con los términos de búsqueda. El cálculo de esa similitud, que utilizo para ordenar los resultados en la cola de prioridad docsOrdenados, lo realizo a través de dos modelos de recuperación de información (aunque solo aplico uno cada vez, en función de las especificaciones que se indiquen). Ambos modelos están integrados en el código que he implementado y son los siguientes:

BM25:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}\left(q_i\right) \cdot \frac{f\left(q_i, D\right) \cdot \left(k_1 + 1\right)}{f\left(q_i, D\right) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

He implementado el modelo de similitud **BM25**, un algoritmo de recuperación de información que se basa en la frecuencia de los términos y en la longitud de los documentos. Para calcular la similitud entre un documento y una consulta, utilizo una ponderación de términos basada en la frecuencia inversa de los documentos. Además, tengo en cuenta aspectos como la longitud media de los documentos y la longitud de la propia consulta, lo que me permite ajustar mejor la relevancia de los resultados.

BM25 es un modelo muy utilizado en motores de búsqueda, y he comprobado que ofrece buenos resultados en la recuperación de documentos relevantes dentro de mi buscador.

Los elementos que he integrado en el modelo son los siguientes:

- **IDF(q_i)**: la frecuencia inversa de documento del término q_i en la colección.
- **f(q_i, D)**: la frecuencia del término q_i dentro del documento *D*.
- **k_1** y **b**: son parámetros ajustables que he definido según las características de mi colección.
- **|D|**: el número de palabras (excluyendo las de parada) en el documento *D*.
- **avgdl**: la longitud promedio de todos los documentos en la colección.

También he desarrollado una variante del algoritmo que limita el análisis a un subconjunto de documentos seleccionados en función de una muestra de preguntas específica. Como la lógica del modelo no cambia y únicamente se modifica el tamaño del conjunto de datos utilizado, no considero necesario profundizar más en esta versión.

DFR:

$$\mathrm{sim}(q,d) = \sum_{i=1}^{k} w_{i,q} * w_{i,d}$$

También he implementado el modelo de similitud Deviation From Randomness (DFR) como otro enfoque para medir la relevancia de los documentos respecto a una consulta. Este modelo parte de la idea de que un documento relevante debería contener términos poco frecuentes dentro del conjunto general de documentos.

Lo que hago es comparar la distribución real de los términos en un documento con la distribución que se esperaría si los documentos fueran completamente aleatorios. La desviación entre ambas distribuciones me sirve para evaluar la similitud y determinar qué documentos son más relevantes para una consulta concreta.

Los elementos que forman parte de este modelo y que he incluido en mi implementación son:

- $W_{i,\varphi}$: el peso del término *i* dentro de la consulta *q*. Lo calculo utilizando la fórmula:
  Wi,q=ftqkWi,q=kftq
  donde ft□,φ representa el número de veces que el término *t* aparece en la consulta *q*, y k es el número total de términos (excluyendo palabras vacías) en la consulta.

También he definido el $W_{i,d}$, que representa el peso del término *i* (presente en la consulta) dentro del documento *d*. Para calcularlo, utilizo la siguiente fórmula basada en el modelo DFR:

Wi,d=−log2(λt)·ftd1+ftdWi,d=−log2(λt)·1+ftdftd

donde:

- λ□ es la razón entre la frecuencia total del término *t* en la colección y el número total de documentos. Es decir:

λt=ftNλt=Nft

- ft es el número total de veces que el término *t* aparece en toda la colección.
- ft_{d} es la frecuencia del término *t* dentro del documento *d*.
- n□ es el número de documentos en los que aparece el término *t*.

Con este cálculo, consigo reflejar cuánto se desvía la aparición de un término en un documento concreto respecto a lo que sería esperable si los términos **se**

distribuyeran aleatoriamente. Cuanto mayor es esta desviación, mayor es la relevancia que asigno al documento en relación con la consulta.

Justificación de la solución elegida

Para implementar el algoritmo, opté por una solución modular en la que delego los cálculos de similitud (que determinan la prioridad de los documentos en la cola) a diferentes clases, en función del modelo de recuperación que se esté utilizando.

Tomé esta decisión para mejorar la coherencia y la reutilización del código, ya que los modelos que empleo tienen cierta complejidad y requieren extraer múltiples datos a partir de la indexación de los documentos. Además, durante el desarrollo me encontré con algunos problemas relacionados con los cálculos de similitud, que surgían por la forma en que se instancian los objetos derivados del proceso de indexación. En algunos casos me vi obligado a crear objetos en puntos del código poco adecuados, lo que podría haber generado problemas de memoria más adelante. Modularizar la lógica me permitió evitar estos conflictos y mantener una arquitectura más limpia y mantenible.

Análisis de las mejoras realizadas en la práctica

Dado que la estructura del algoritmo de búsqueda no presenta gran complejidad, no me he encontrado con problemas relevantes que merezcan ser destacados en esa parte concreta. Sin embargo, la interacción del algoritmo con las estructuras de indexación sí me llevó a realizar ajustes importantes en el código de las clases IndexadorHash e IndexadorInformación.

En particular, tuve que modificar aspectos relacionados con la creación y destrucción de objetos. A lo largo del desarrollo, detecté que la falta de flexibilidad en el diseño original de esas clases provocaba que algunos objetos utilizados durante el proceso de búsqueda pudieran causar fugas de memoria. Estas fugas fueron confirmadas mediante Valgrind, lo que me obligó a revisar y reescribir parte del código heredado de la práctica anterior para garantizar una gestión de memoria más segura y eficiente.

Análisis de eficiencia computacional (no teórica)

Debido a que trabajo con un sistema macOS, no pude ejecutar correctamente el archivo memory.cpp proporcionado para el análisis de memoria. Esta limitación ya la

comuniqué al profesor con antelación. Aun así, antes de disponer de la versión definitiva del ejecutable, decidí probar el programa en los ordenadores del aula para poder obtener una estimación del consumo de memoria durante la prueba de estrés.

En esas pruebas preliminares, los resultados que obtuve eran muy similares a los presentados en el análisis proporcionado por el profesorado, aunque en mi caso el consumo de memoria fue ligeramente superior.

La prueba de estrés y el análisis espacial de memoria permiten evaluar la eficiencia del sistema, tanto en términos temporales como espaciales. En el informe oficial se muestra que la memoria total utilizada fue de 52902 Kbytes. En mi caso, esa cifra era algo mayor, lo que puede deberse a diferencias en el momento de ejecución, condiciones del entorno o incluso pequeños cambios intermedios en el código antes de la versión final.

La mayor parte de la memoria se destina a datos —estructuras, variables y otros elementos necesarios para el funcionamiento del sistema—, mientras que la pila almacena principalmente variables locales y contextos de ejecución de funciones. Esta diferencia es esperable y depende de cómo se gestiona la información en el programa.

También se midió el tiempo que tarda el algoritmo en realizar la clasificación y búsqueda sobre el corpus proporcionado, con 423 documentos relevantes. Como es lógico, gran parte del trabajo computacional lo realiza el indexador, por lo que los tiempos obtenidos son similares a los de la práctica anterior. Sin embargo, en esta prueba concreta, el buscador también fue sometido a un escenario de carga elevada, con 83 preguntas distintas procesadas sobre una colección extensa de documentos. Esto provocó que los tiempos de ejecución aumentaran de forma notable, especialmente por la cantidad de términos generados y el tamaño de la colección, lo que resulta coherente con un crecimiento exponencial del procesamiento.