Sistemas de Almacenamiento y Recuperación de Información
Práctica 4: Buscador web

Roberto Ortiz Sanz
Francisco Miralles Ferrer
Carlos Ruiz Aguirre
David Esteso Calatrava

20 de mayo de 2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSINF
Grado en Ingeniería Informática

Índice

1 Introducción

1.1 Coordinación de trabajo

Durante la ejecución del proyecto se ha podido mantener un seguimiento de versiones mediante la herramienta de control de versiones git, esto ha permitido trabajar de forma concurrente.

Además, se ha realizado la mayoría del proyecto juntos en llamada mediante la aplicación de comunicación Discord, por lo que se ha mantenido una coordinación adecuada y se ha podido resolver dudas entre los métodos que realizaban los demás compañeros.

1.2 Dificultades encontradas

Afrontamos dificultades debido a la modularidad del proyecto, ya que inicialmente no teníamos una comprensión clara de las dependencias entre las diferentes partes. Esto complicó la distribución inicial del trabajo, ya que no sabíamos qué módulos iban a depender más de otros ni qué métodos podrían resultar más costosos en términos de carga de trabajo. La división de métodos consideramos que ha sido equitativa ya que en caso de que algunos compañeros hayan realizado mayor cantidad de métodos, estos no contienen la dificultad de otros.

Además, encontramos dificultades al probar métodos como el parser, ya que la gran mayoría de entradas de Wikipedia se actualizaron después de que se realizara el crawling de ejemplo. Esto generó discrepancias entre los datos obtenidos durante el crawling y los datos actuales, lo que complicó la evaluación precisa de la funcionalidad del parser.

2 SAR_Crawler_Lib.py

El objetivo de esta sección es implementar un sistema de crawling para descargar artículos de Wikipedia y guardarlos en formato JSON. Este sistema permite iniciar el crawling desde una URL o archivo de URLs, procesar y estructurar el contenido de las páginas, y controlar la profundidad del crawling. Es fundamental para la indexación y consulta de artículos en la segunda parte del proyecto.

2.1 start_crawling() - David

El método `start_crawling` captura contenido de Wikipedia desde una lista inicial de URLs hasta alcanzar un número máximo de documentos o agotar las URLs. Utiliza estructuras de datos para rastrear URLs visitadas, pendientes y en proceso. Descarga y analiza el contenido de cada URL, añadiendo nuevos enlaces válidos para procesar. El contenido textual descargado se analiza mediante una llamada a `self.parse_wikipedia_textual_content`, que genera un diccionario estructurado del artículo. Los documentos capturados se guardan periódicamente en lotes si se especifica un tamaño de lote, o al finalizar el proceso. Este enfoque asegura la captura de los datos, respetando los límites de documentos y profundidad establecidos.

2.2 parse_wikipedia_textual_content() - Roberto

El método `parse_wikipedia_textual_content` convierte el texto crudo de un artículo de Wikipedia en una estructura organizada y fácil de manejar. A continuación, se describen los pasos y componentes clave del método:

Primero, se definen dos argumentos para el método: `text`, que es el texto en crudo del artículo de Wikipedia, y `url`, que es la URL del artículo que se añade como un campo en el resultado.

El objetivo del método es transformar el texto crudo en un diccionario que contenga la URL del artículo, el título, un resumen y una lista de secciones del artículo, donde cada sección puede tener subsecciones.

El proceso comienza con la definición de una función `clean_text` que elimina líneas vacías del texto. Luego, se utiliza una expresión regular para intentar extraer el título y el resumen del texto. Si no encuentra título ni resumen, el método devuelve `None`.

A continuación, se inicializa un diccionario con los campos `url`, `title`, `summary` y `sections` (inicialmente vacío). Luego, se encuentran las secciones en el texto restante utilizando expresiones regulares. Para cada sección, se identifica el nombre y el contenido, y se crea un diccionario que incluye su nombre, contenido y una lista de subsecciones (inicialmente vacía).

Dentro de cada sección, se encuentran las subsecciones. Para cada subsección, se identifica el nombre y el contenido, se crea un diccionario para la subsección y se añade a la lista de subsecciones de la sección correspondiente.

Finalmente, se añaden las secciones (con sus subsecciones) al diccionario del artículo. El método devuelve un diccionario estructurado con la información del artículo, o `None` si no se pudo extraer título ni resumen.

3 SAR_lib.py

El objetivo de esta parte es crear un índice invertido de artículos de Wikipedia en JSON. Requiere un directorio de artículos y un nombre de índice como entrada. Procesa los artículos, asigna identificadores únicos y crea un índice invertido por término. La información se guarda en disco junto con estadísticas del proceso.

3.1 Parte 1: Indexación

3.1.1 index_dir() - David

Se ha editado el método `index_dir` para que permita la generación de los índices permuterm y stemming si se pide en la ejecución del indexador.

3.1.2 index_file() - Carlos

Para **indexar el contenido de un fichero** o documento el **primer paso es asignarle un identificador único**, este corresponderá al número de documentos ya almacenados en ese momento, así conseguimos que el primer documento tenga un *docID*=0, el segundo tenga un *docID*=1...

En resumen, el *docID* del documento n-ésimo analizado será n-1.

A continuación, trataremos cada línea del documento como un artículo distinto, pues en cada línea encontramos un artículo en formato JSON.

Es prioritario tener en cuenta antes de la indexación si el documento ya ha sido indexado, por lo que mediante el método `self.already_in_index()` lo podemos verificar para no considerar este documento.

Tras esto, asignamos un *artID* de la misma forma que para cada documento; considerando la cantidad de artículos indexados.

Es en este momento cuando empieza el proceso de indexación:

1. Se *tokeniza* el contenido del campo en cuestión (el conjunto de campos considerados para *tokenización* depende de si la opción *multifield* está activa).

2. Para cada *token*, **se añade** en la posting list correspondiente **el artID** en el que ha aparecido o una tupla (**artID, pos**) con el artID y la posición que ocupa el token dentro del texto dependiendo de si la opción de índice posicional está activa.

   Antes de añadir una nueva entrada a la posting list, **se comprueba si**, en el caso de que la indexación no sea posicional, **este token ya ha aparecido** en el documento actual para evitar añadir un duplicado. **De la misma forma si** la indexación es posicional y **un token ya ha aparecido** en un documento se tiene en cuenta y no se añade la tupla, sino que **se añade la posición dentro del segundo elemento de la tupla** para almacenar todas las posiciones en las que un token aparece dentro de un documento.

3. Finalmente, se añade la URL del artículo al conjunto de URLs, gracias a este conjunto somos capaces de determinar si un artículo ya ha sido indexado.

3.1.3 make_stemming() - Carlos

A la hora de crear el índice de *stemming* para los términos de todos los índices debemos de iterar por todos los campos indexados (depende de la opción multifield) para extraer su *stem* y almacenar todos aquellos términos que comparten stem en el diccionario `self.sindex`.

Ejemplo: Contenido de `self.sindex['all']['program']` tras la indexación de la carpeta '200' (de sus ficheros):
['programación', 'programa', 'programadora', 'programas', 'programar', 'programador', 'programadores', 'programarlas', 'programadas', 'programada', 'programables', 'programando', 'program', 'programable', 'programarse', 'programados', 'programó', 'programado', 'programaban', 'programan', 'programe', 'programacion']

3.1.4 make_permuterm() - Carlos

A la hora de crear el índice de *permuterms* para los términos de todos los índices debemos de iterar por todos los campos indexados (depende de la opción multifield) y calcular el índice de permuterms del término con las rotaciones de cada subcadena con

el símbolo '$' de la forma vista en teoría. Hemos decidido emplear una función llamada `generate_permuterms()` para externalizar el propio cálculo del permuterm. Deberemos almacenar el término generado por su permuterm en `self.ptindex`.

Ejemplo: Contenido de `self.ptindex['all']['sa$ca']` tras la indexación de la carpeta '200' (de sus ficheros):
casa

3.1.5 show_stats() - Carlos

Para este apartado debemos mostrar la longitud de los distintos diccionarios generados durante el proceso de indexación, así como el número de archivos y artículos indexados. Debe tener este formato:

```
========================================
Number of indexed files: 3
----------------------------------------
Number of indexed articles: 589
----------------------------------------
TOKENS:
        # of tokens in 'all': 103358
        # of tokens in 'title': 638
        # of tokens in 'summary': 11474
        # of tokens in 'section-name': 3397
        # of tokens in 'url': 589
----------------------------------------
PERMUTERMS:
        # of permuterms in 'all': 888793
        # of permuterms in 'title': 5095
        # of permuterms in 'summary': 101524
        # of permuterms in 'section-name': 30396
        # of permuterms in 'url': 28155
----------------------------------------
STEMS:
        # of stems in 'all': 71944
        # of stems in 'title': 606
        # of stems in 'summary': 7130
        # of stems in 'section-name': 2614
        # of stems in 'url': 588
----------------------------------------
Positional queries are allowed
========================================
Time indexing: 3.67s.
Time saving: 0.84s.
```

- La sección PERMUTERMS solo se muestra si se ha realizado el cálculo de *permuterms*.
- La sección STEMS solo se muestra si se ha realizado el cálculo de *stems*.

- En una sección se muestran los campos 'all', 'title', 'summary', 'section-name', 'url' si se ha activado la opción de indexación *multifield*, en caso de no haberse activado solo se realiza el cálculo de 'all' y se muestra el campo.
- Además, se indica si se admiten consultas posicionales, esto varía en función de si se ha realizado la indexación con índices posicionales.

3.2 Parte 2.1: Recuperación

3.2.1 solve_query() - David

El método `solve_query` resuelve una consulta desglosando y evaluando términos y operadores. Primero, normaliza la consulta con `normalize_query`, convirtiéndola en una lista de tokens, para después obtener las posting lists de los términos y almacenarlas junto con los operadores en una lista de tokens. Después aplica el siguiente algoritmo: se utilizan dos pilas, una para operadores y otra para operandos, y se recorre la lista de tokens de la consulta para evaluarla. Para darle prioridad a los paréntesis, cuando se encuentra un paréntesis de cierre, el algoritmo procesa los operadores y operandos hasta que encuentra el paréntesis de apertura correspondiente, evaluando así la subexpresión. En caso de ser precedida por un *NOT*, se evalúa. Esto asegura que las subexpresiones entre paréntesis se evalúen antes que otras partes de la consulta. Para los operadores *AND* y *OR*, evalúa los operadores que se habían almacenado previamente en la pila antes de añadir el nuevo operador, ya que leemos las consultas de izquierda a derecha. Posteriormente, añade el operador a la pila. Si el operador *NOT* es seguido por una subexpresión entre paréntesis, lo añade a la pila; de lo contrario, aplica el operador al siguiente operando. Los términos se añaden directamente a la pila de operandos como sus posting lists correspondientes. El proceso continúa hasta que todos los tokens se procesan y cualquier operador restante se evalúa, obteniendo el resultado final de la pila de operandos.

3.2.2 normalize_query() - David

El objetivo del método `normalize_query` es convertir la consulta en una lista compuesta de operadores y operandos. Primero, se recorre la consulta para separarla en palabras, a menos que se encuentren dentro de comillas dobles, indicando una consulta posicional, en cuyo caso los espacios dentro de las comillas no separan las palabras. Además, se asegura de que los paréntesis estén correctamente separados de los operandos en la consulta normalizada. Posteriormente, se recorre nuevamente la consulta ya normalizada para insertar operadores lógicos *AND* entre los operandos cuando el usuario no los haya incluido explícitamente. Esto garantiza que la consulta esté estructurada de manera clara y que los operadores lógicos estén presentes donde sean necesarios.

3.2.3 evaluate() - David

Método que se encarga de evaluar las operaciones con los operandos *AND* y *NOT*.

3.2.4 get_posting() - Fran

Obtiene una lista de artículos relacionados con los parámetros de entrada del método. Los parámetros son: el término que se consulta y el campo que indica el diccionario indexado en el que buscar, siendo opcional (si no se introduce, se busca en el campo *all*).

Dependiendo del término de entrada, se hacen llamadas a los métodos:

- `get_positionals()`: si el término es un término posicional (contiene espacios).
- `get_stemming()`: si se ha indexado con stemming.
- `get_permuterm()`: si el término contiene comodines.

En cualquier otro caso, se obtiene la *posting list* mediante el diccionario *index*, devolviendo una lista vacía si el término no ha sido indexado.

Por último, el método elimina las posiciones de las *posting list* en el caso de que hayan, ya que solo debe devolver los artículos.

El método devuelve la *posting list* relacionada con el término y campo.

3.2.5 get_positionals() - Fran

Devuelve una *posting list* con aquellos artículos que contienen el sintagma pasado como parámetro término y relacionado con el campo.

Para ello, se hace una primera búsqueda de los artículos comunes para todos los términos del sintagma, para esto, se realiza *and* de las *postings lists*. Esto agiliza la computación del bucle posterior.

Una vez teniendo la lista de artículos comunes para todos los términos, se obtiene la referencia de la *posting list* del primer término y se recorre hasta encontrar un elemento cuyo artículo sea el mismo que alguno de la lista de artículos comunes. Al encontrar alguno, se hace una búsqueda a través de sus posiciones y se busca si los demás términos en el mismo artículo se encuentran en una posición sucesora. Aquellas iteraciones que acaben, que hayan recorrido todos los términos y hayan encontrado posición sucesora, almacenarán su artículo en el resultado a devolver.

3.2.6 get_stemming() - Fran

En este método, se obtiene el *stem* asociado al término de entrada con un atributo *stemmer* del *Indexer*. Luego se obtienen los términos indexados con el diccionario *sindex* a partir del *stem* y el campo de entrada.

Una vez obtenidos todos los términos, se hace una búsqueda de sus *posting list* en el índice *index* dependiendo también del campo de entrada para dar funcionalidad al *multifield*, acabando con la operación *OR* de todas las listas obtenidas. De esta forma devolveremos todos los artículos en los que aparece el *stem* del término de entrada.

3.2.7 get_permuterm() - Roberto

El método recibe un término con un comodín `*` o `?` y el campo donde buscar.

El primer paso es realizar la conversión a *wild card query*, añadiendo un `$` al final del término y permutando su posición hasta dejar el comodín a la derecha.

Una vez realizado, se obtienen todas las claves del índice permuterm `ptindex` y se hace matching de cada clave con el término teniendo en cuenta que:

- Si el comodín es `*`, el matching se realiza si la clave contiene como prefijo al término dado.
- Si el comodín es '?', el matching se realiza en todas las claves que empiecen por el término dado y tengan una longitud de una unidad superior.

Una vez se tienen todas las claves que hacen matching, se encuentran las palabras relacionadas con las *postings lists* obteniendo el valor de la clave de `ptindex` con las claves. Por último, se realiza el *OR* de todas las *postings lists* y se devuelve.

Ejemplo: Para una búsqueda con comodín, como `ca*`, se realiza el siguiente proceso:

- Convertir `ca*` a una *wild card query*, añadiendo `$` al final: `ca*$`.
- Generar permutaciones hasta tener el comodín a la derecha: `ca*$` ->`$ca*`.
- Buscar en el índice permuterm las claves que comiencen con `$ca`.

Si el comodín fuera `ca?`, se buscarían claves que comiencen con `$ca` y tengan un carácter adicional después de `ca`.

3.2.8 reverse_posting() - Fran

Como parámetros contiene una lista de artículos, la cual se quiere conseguir su negado respecto a todo el conjunto de artículos indexados.

Se utiliza para realizar las consultas con el operando *NOT*. Para conseguir esta funcionalidad, se obtienen todos los id de los artículos mediante el diccionario `articles`. Se ordenan y se devuelve el resultado de la operación `minus_posting()` explicada en los siguientes apartados con los parámetros de todas las claves y la lista pasada por parámetros. Esto devolverá aquellos artículos que no se encuentran en la lista pasada por parámetros.

3.2.9 and_posting() - Fran

Se le pasan por parámetros dos listas las cuales se les aplica un algoritmo que da funcionalidad a las consultas con el operando *AND*.

Las listas se ordenan y se eliminan duplicados, para impedir posibles errores ya que el algoritmo es funcional solo si las listas se encuentran ordenadas ascendentemente.

Se hace un recorrido de ambas listas hasta que alguna de las dos se vacíe. En cada iteración se recupera el primer elemento de las listas. Si este coincide, se añade al resultado y avanzan ambas listas. En caso de no coincidir, avanza la lista cuyo elemento sea menor.

3.2.10 or_posting() - Fran

Similar al algoritmo de `and_posting()`, recibe dos listas por parámetros, ordenándolos y eliminando repetidos como primer paso.

El recorrido se realiza mientras que ambas listas no estén vacías, obteniendo en cada iteración los primeros elementos de las listas. Si son iguales, se añade uno de los dos y avanzan ambas listas en su recorrido. Si son distintos, se añade el menor elemento y avanza la lista cuyo elemento se ha añadido. Al terminar el bucle, se añaden los elementos de aquella lista que no está vacía.

3.2.11 minus_posting() - Fran

Este método es usado como auxiliar por `reverse_posting()` para aportar funcionalidad al operando *NOT*.

Se hace un recorrido hasta que alguna de las listas se vacíe. En cada iteración, si el elemento es el mismo en ambas, no se añade ninguno al resultado y se avanza en ambas listas. Si el primer elemento es menor al segundo, se añade el elemento al resultado y se avanza en la lista. Si es el segundo el elemento el menor, solamente se avanza en la lista. Al terminar el bucle, si la longitud de la primera lista aún contiene elementos, se añaden todos estos al resultado.

Con este funcionamiento, se representa la eliminación de los elementos de una lista sobre otra.

3.3 Parte 2.2: Mostrar resultados

3.3.1 solve_and_show() - Roberto

El método `solve_and_show` se encarga de resolver una consulta y mostrar los resultados junto con el número de artículos recuperados. A continuación, se describen los pasos y componentes del método:

Primero, se resuelve la consulta utilizando el método `solve_query`, lo que devuelve una lista de IDs de artículos que coinciden con la consulta. Luego, se inicializan tres listas: `indexed_urls` para almacenar las URLs de los artículos, `titles` para almacenar los títulos, y `snippets` para almacenar fragmentos de texto relevantes si la opción de mostrar snippets está habilitada.

Para cada ID de artículo en la lista de resultados, se obtiene el `docID` y la URL correspondiente. Se añade la URL a la lista `indexed_urls`. Luego, se abre el archivo correspondiente al `docID` y se lee línea por línea. Si se encuentra un artículo cuya URL coincide, se añade su título a la lista `titles` y, si la opción de mostrar snippets está habilitada, se genera un snippet del contenido del artículo y se añade a la lista `snippets`.

Una vez procesados todos los artículos, se crea una lista de resultados combinando las URLs, los títulos y los IDs de los artículos. Si la opción de mostrar snippets está habilitada, se imprime cada resultado con su URL, título y snippet correspondiente. De lo contrario, se imprime solo la URL y el título de cada artículo, deteniéndose después de 10 resultados si la opción `show_all` no está habilitada.

Finalmente, se imprime el número total de resultados y se devuelve este número como salida del método.

3.3.2 Cálculo de Snippets - David y Carlos

Finalmente, vamos a exponer cómo hemos realizado el cálculo de `snippets`, el cual se emplea en el método `solve_and_show()` si se ha activado la opción de mostrar `snippets`. Para realizar el cálculo llamamos a la función `make_snippet()` pasando un artículo, una lista de términos y un tamaño de ventana. Obtenemos una lista de las ocurrencias de los términos de la consulta y determinamos el inicio y el final del `snippet` empleando la posición mínima y máxima de los índices anteriores, asegurándonos de que no exceda los límites del texto y que tenga el tamaño de ventana especificado (por defecto 50 unidades).