Count Word

Andres Ricardo Caceres Ortiz 506222067

***Abstract*—Se ha propuesto desarrollar algoritmos para el procesamiento de texto. En este taller se propondrán dos ejemplos de algoritmos para la búsqueda y enumeración de las palabras más recurrentes en un documento previamente definido. No solo se identificarán las palabras más repetidas, sino también se mostrará la ubicación precisa de cada palabra en el documento. También se calculará la complejidad de cada algoritmo, en este caso se hallarán la complejidad espacial y temporal de cada algoritmo, de este modo garantizar una eficiencia óptima tanto en términos de tiempo como de espacio.**

I. Introduction

La manipulación y el análisis de grandes cantidades de texto se han vuelto esenciales en una variedad de aplicaciones, desde motores de búsqueda en la web hasta la minería de datos y la inteligencia artificial. Es por ello que se ha propuesto la creación de dos algoritmos, capaces de separar cada palabra y almacenarla en un arreglo, así mismo este deberá mostrar la ubicación exacta y las veces que se repite. Antes de desarrollar el taller se deben tener en cuenta algunas definiciones como:
HashMap: es una estructura de datos en programación que representa una colección de pares clave-valor, donde cada clave está asociada a un valor.

Arreglo o Array: Es un tipo de dato que almacena datos, dichos datos son del mismo tipo (String, int etc), todo esto de manera estructurada.
Array List: Es una clase similar a los Array, con la diferencia de que para un ArrayList no es necesario declarar su tamaño.

II. Algoritmo 1: Memorización para la Clasificación de Frecuencia de Palabras

Calcular las frecuencias de palabras puede ser computacionalmente costoso, especialmente cuando se trabaja con conjuntos de datos voluminosos. Para abordar este problema, podemos emplear un algoritmo que utiliza la memorización, una técnica de caché que almacena resultados previamente calculados para evitar cálculos redundantes.

El primer paso consiste en crear un HashMap para almacenar pares de palabra-frecuencia. A continuación, recorremos cada documento, lo dividimos en palabras y actualizamos las frecuencias de las palabras en el HashMap. Después de procesar todos los documentos, tenemos un HashMap que contiene pares de palabra-frecuencia. Para identificar las palabras más repetidas, ordenamos el HashMap por valores (frecuencias) en orden descendente. Este enfoque garantiza que las palabras más frecuentes aparezcan en la parte superior de la clasificación.

La caché de memorización almacena el HashMap de palabra-frecuencia calculado, evitando la necesidad de recalcularlo para los mismos datos de entrada en el futuro. Esta optimización no solo ahorra recursos computacionales, sino que también acelera el proceso de identificación de términos clave dentro del texto. A continuacion el desarrollo del algoritmo:

```
1
2  import java.util.*;
3
4  public class DocumentSearch {
5
6      private static Map<String, List<
           Integer>> invertedIndex = new
           HashMap<>();
7      private static Map<String, List<
           Integer>> cache = new HashMap<>();
8      public static void buildInvertedIndex
           (String[] documents) {
9          for (int i = 0; i < documents.
               length; i++) {
10             String document = documents[i
                   ];
               String[] words = document.
                   split("\\s+");
12             for (String word : words) {
13                 word = word.toLowerCase()
                       ;
14                 if (!invertedIndex.
                       containsKey(word)) {
15                     invertedIndex.put(
                           word, new
                           ArrayList<>());
16                 }
17                 invertedIndex.get(word).
                       add(i);
18             }
19         }
20     }
21
22     public static List<Integer>
           searchWord(String word) {
23         if (cache.containsKey(word)) {
24             return cache.get(word);
25         }
26
27         word = word.toLowerCase();
28         List<Integer> result =
```

```
            invertedIndex.getOrDefault(
            word, new ArrayList<>());

        cache.put(word, result);

        return result;
    }

    public static void main(String[] args
        ) {
        String[] myDocuments = {
            "La programaci n en Python
               es clave para el trabajo
               con datos",
            "Los programadores en Java
               tienen un alto inter s en
                pasar a Python",
            "La optimizaci n de
               algoritmos es fundamental
               en el desarrollo de
               software",
            "..."
        };

        buildInvertedIndex(myDocuments);

        String searchWord = "desarrollo";
        List<Integer> documentIndices =
            searchWord(searchWord);

        System.out.print(searchWord + ":
            [");
        for (int i = 0; i <
            documentIndices.size(); i++) {
            System.out.print(
                documentIndices.get(i)+1);
            if (i < documentIndices.size
                () - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }
}
```

A. *Complejidad Temporal y Espacial*

- Complejidad Temporal: En el peor caso, este algoritmo debe recorrer todos los documentos en la colección, lo que resulta en una complejidad de $O(n)$, donde n es el número de documentos. La búsqueda en cada documento se realiza en tiempo lineal en función de la longitud del documento y la longitud de la palabra objetivo, por lo que podemos considerar la búsqueda dentro de un documento como $O(m)$, donde m es la longitud máxima del documento o la palabra objetivo más larga.
- Complejidad Espacial: El espacio adicional utilizado por este algoritmo es principalmente para almacenar la lista de documentos que contienen la palabra objetivo. En el peor caso, podría ser $O(n)$, donde n es el número de documentos que contienen la palabra objetivo.

III. Algoritmo 2: Índice Invertido para Recuperación de Documentos

El segundo algoritmo que exploraremos realiza la búsqueda de documentos que contienen una palabra específica en un conjunto de documentos. La caché juega un papel crucial en este algoritmo. Una vez construido el índice invertido, podemos almacenar en caché los resultados, asegurando que las búsquedas posteriores de las mismas palabras clave sean extremadamente rápidas. Cuando se solicita una búsqueda, primero verificamos la caché y, si se encuentra el resultado, lo devolvemos directamente.

```
import java.util.*;
import java.util.stream.Collectors;

public class WordFrequencyRanking {
    private static Map<String[], Map<
        String, Integer>> cache = new
        HashMap<>();

    public static Map<String, Integer>
        getWordFrequency(String[]
        documents) {
        if (cache.containsKey(documents))
            {
            return cache.get(documents);
        }
        Map<String, Integer>
            wordFrequency = new HashMap
            <>();
        for (String document : documents)
            {
            String[] words = document.
                split("\\s+");
            for (String word : words) {
                word = word.toLowerCase()
                    ;
                wordFrequency.put(word,
                    wordFrequency.
                    getOrDefault(word, 0)
                    + 1);
            }
        }
        Map<String, Integer>
            sortedWordFrequency =
            wordFrequency.entrySet()
                .stream()
                .sorted(Map.Entry.<String
                    , Integer>
```

```
                    comparingByValue().
                    reversed())
                .collect(Collectors.toMap
                    (Map.Entry::getKey,
                    Map.Entry::getValue, (
                    e1, e2) -> e1,
                    LinkedHashMap::new));


        cache.put(documents,
            sortedWordFrequency);

        return sortedWordFrequency;
    }

    public static void main(String[] args
        ) {
        String[] myDocuments = {
          "La programaci n en Python
              es clave para el trabajo
              con datos",
          "Los programadores en Java
              tienen un alto inter s en
               pasar a Python",
          "La optimizaci n de
              algoritmos es fundamental
              en el desarrollo de
              software",
          "..."
        };

        Map<String, Integer>
            wordFrequency =
            getWordFrequency(myDocuments);


        int rank = 1;
        for (Map.Entry<String, Integer>
            entry : wordFrequency.entrySet
            ()) {
            System.out.println(entry.
                getKey() + " " + entry.
                getValue());
            rank++;
            if (rank > 10) {
                break;
            }
        }
    }
}
```

A. *Complejidad Temporal*

Construcción del Índice Invertido (buildInvertedIndex): O(N * M), donde N es el número de documentos y M es el número promedio de palabras en un documento. Búsqueda de Palabras (searchWord): O(1) en promedio debido al uso de la caché. Sin embargo, en el peor caso, donde la palabra no está en la caché y debe buscarse en el Índice Invertido, la complejidad sería O(N), donde N es el número promedio de documentos donde aparece la palabra.

B. *Complejidad Espacial*

Espacio para el Índice Invertido: O(W * D), donde W es el número total de palabras únicas en todos los documentos y D es el número de documentos. Espacio para la Caché: O(W), donde W es el número total de palabras únicas.

IV. Conclusiones

En este artículo, hemos explorado dos algoritmos para buscar palabras en una colección de documentos y hemos discutido su complejidad temporal y espacial. La elección del algoritmo adecuado depende de varios factores, como el tamaño de la colección de documentos y las operaciones de búsqueda previstas. La Búsqueda Lineal es simple pero puede volverse ineficiente para colecciones grandes, mientras que la Búsqueda con Mapa es más eficiente en términos de tiempo para búsquedas repetidas.

En la práctica, el uso de estructuras de datos como mapas o índices invertidos es común para mejorar la eficiencia en la búsqueda de palabras en grandes conjuntos de documentos. La elección del algoritmo y la estructura de datos adecuados depende de los requisitos específicos de su aplicación y de la cantidad de documentos que necesita procesar.