Taller Tres Y Cuatro Estructura De Datos

Lisseth Tatiana Quilindo Patiño
Fundación Universitaria Konrad Lorenz

***Resumen*—El informe se ejecuto un algoritmo donde aplicando Memoization se mostraron las clasificaciones de las palabras más repetidas y se desarrollo un algoritmo (tipo búsqueda) que, al pasar una palabra como parámetro da una función que, devuelva una nueva lista con los documentos que contienen las mismas palabra analizando cada linea del algoritmo usando la notación Big-O.**

I. Introducción

El informe detalla un proyecto en el que se ha implementado un algoritmo que utiliza la técnica de Memoization para llevar a cabo dos tareas principales: calcular las clasificaciones de las palabras más repetidas en un conjunto de documentos y desarrollar una función de búsqueda de palabras en estos documentos. Ambas tareas tienen aplicaciones significativas en la manipulación y análisis de texto.

En la primera parte del proyecto, se ha aplicado la técnica de Memoization para calcular las clasificaciones de las palabras más repetidas en los documentos proporcionados. Esto implica contar la frecuencia de cada palabra en los documentos y determinar cuáles son las palabras más comunes. El uso de Memoization permite almacenar los resultados de los cálculos para evitar recálculos innecesarios, mejorando así la eficiencia del algoritmo.

En la segunda parte del proyecto, se ha desarrollado un algoritmo de búsqueda de palabras. Dado un parámetro que representa una palabra, la función de búsqueda devuelve una nueva lista que contiene los documentos que contienen esa palabra. Este tipo de búsqueda es útil en aplicaciones de recuperación de información y procesamiento de texto.

El análisis de ambos algoritmos se ha llevado a cabo utilizando la notación Big-O. Esto ha permitido evaluar la eficiencia computacional de cada algoritmo en términos de complejidad espacial (uso de memoria) y complejidad temporal (tiempo de ejecución). Se han considerado diversos escenarios, como la variación en el tamaño de los documentos y la cantidad de palabras únicas presentes en ellos, para comprender cómo los algoritmos se comportan en diferentes situaciones.

El objetivo de este análisis es determinar cómo los algoritmos escalan con respecto al tamaño de los datos de entrada y cuántos recursos computacionales consumen. Esto es esencial para tomar decisiones informadas sobre la eficiencia y la capacidad de manejo de grandes conjuntos de datos.

II. Resultados

III. Análisis de la Complejidad Algorítmica

En este análisis, se descompne el código en sus partes principales y se evalua la complejidad de cada una.

III-A. Inicialización del diccionario cache

La inicialización del diccionario cache se realiza una sola vez y es una operación de tiempo constante, $O(1)$.

III-B. Iteración a través de cada documento en my_documents

El bucle exterior itera a través de los documentos en `my_documents`. Si hay N documentos en `my_documents`, este bucle se ejecutará N veces. Por lo tanto, la complejidad de esta parte es $O(N)$.

III-C. Procesamiento de palabras en cada documento

Para cada documento, se divide el texto en palabras, lo que implica recorrer el texto. Si el promedio de palabras en un documento es M, el bucle interno se ejecutará M veces en promedio por cada documento. Por lo tanto, la complejidad de esta parte es $O(M)$ en promedio por documento.

III-D. Verificación y asignación en el diccionario cache

Dentro del bucle interno, se realizan operaciones de búsqueda y asignación en el diccionario cache. La operación de búsqueda en un diccionario en Python es de tiempo constante, $O(1)$. La asignación también es de tiempo constante, $O(1)$. En el peor caso, se ejecutan $N \cdot M$ operaciones de búsqueda y asignación (una por cada palabra en cada documento). Por lo tanto, la complejidad de esta parte es $O(N \cdot M)$ en el peor caso.

III-E. Impresión del diccionario

La impresión del diccionario cache implica recorrer todos los pares clave-valor en el diccionario. En el peor caso, habría $N \cdot M$ pares clave-valor en el diccionario, por lo que la impresión sería $O(N \cdot M)$ en el peor caso.

III-F. Complejidad total del código

Para obtener la complejidad total, combinamos todos estos elementos:

La inicialización del diccionario es $O(1)$.
La iteración a través de cada documento es $O(N)$.
El procesamiento de palabras en cada documento es $O(M)$ en promedio por documento.
Las operaciones de búsqueda y asignación en el diccionario son $O(N \cdot M)$ en el peor caso.
La impresión del diccionario es $O(N \cdot M)$ en el peor caso.

En general, la complejidad total del código sería $O(N \cdot M)$ en el peor caso, ya que las operaciones de búsqueda y asignación en el diccionario y la impresión del mismo son las partes que más contribuyen a la complejidad. La complejidad también depende del tamaño y contenido de los documentos en `my_documents`.

III-G. código

```
[2] my_documents = [
        "La programación en Python es clave para el trabajo con datos",
        "Los programadores en Java tienen un alto interés en pasar a Python",
        "La optimización de algoritmos es fundamental en el desarrollo de software",
        "Las bases de datos relacionales son esenciales para muchas aplicaciones",
        "El paradigma de programación funcional gana popularidad",
        "La seguridad informática es un tema crucial en el desarrollo de aplicaciones web",
        "Los lenguajes de programación modernos ofrecen abstracciones poderosas",
        "La inteligencia artificial está transformando diversas industrias",
        "El aprendizaje automático es una rama clave de la ciencia de datos",
        "Las interfaces de usuario intuitivas mejoran la experiencia del usuario",
        "La calidad del código es esencial para mantener un proyecto exitoso",
        "La agilidad en el desarrollo de software permite adaptarse a cambios rápidamente",
        "Las pruebas automatizadas son cruciales para garantizar la estabilidad del software",
        "La modularización del código facilita la colaboración en equipos de programadores",
        "El control de versiones es necesario para rastrear cambios en el código",
        "La documentación clara es fundamental para que otros entiendan el código",
        "La programación orientada a objetos promueve la reutilización de código",
        "La resolución de problemas es una habilidad esencial en la programación",
        "La optimización prematura puede llevar a código complicado y difícil de mantener",
        "El diseño de interfaces de usuario atractivas mejora la usabilidad de las aplicaciones",
        "El código limpio es esencial para facilitar el mantenimiento",
        "Los patrones de diseño son soluciones probadas para problemas comunes",
        "Las pruebas unitarias garantizan el correcto funcionamiento de las partes del código",
        "El desarrollo ágil prioriza la entrega continua de valor al cliente",
```

Figura 1. Asignando una lista de cadenas de texto a la variable `my_documents`.

```
[5] cache = {}
    for index, document in enumerate(my_documents):
      words = document.lower().split()
      for word in words:
        if word in cache:
          cache[word].append(index)
        else:
          cache[word] = [index]

    print(cache)

    {'la': [0, 2, 5, 7, 8, 9, 10, 11, 12, 13, 13, 15, 16, 16, 17, 17, 18, 19, 23, 25, 25, 27, 28, 28, 30, 31, 31, 32, 35,
```

Figura 2. Devuelve la longitud de una lista.

```
[6] def buscar(palabra):
      cache = {}
      for index, document in enumerate(my_documents):
        words = document.lower().split()
        for word in words:
          if word in cache:
            cache[word].append(index)
          else:
            cache[word] = [index]
      return cache.get(palabra, "No encontrada")

    resultado = buscar("problemas")
    print("Resultados de la busqueda:", resultado)

    Resultados de la busqueda: [17, 21, 65]
```

Figura 3. Crea un índice invertido simple que muestra en qué documentos aparece cada palabra, utilizando un diccionario.

```
[7] len(cache["problemas"])

    3
```

Figura 4. Devuelve la lista de índices de documentos donde aparece la palabra proporcionada como argumento.

```
[8] cache = {}
    for index, document in enumerate(my_documents):
      words = document.lower().split()
      count = 0
      for word in words:
        if word in cache:
          cache[word] = cache.get(word,0) + 1
        else:
          cache[word] = index

    print(cache)

    {'la': 57, 'programación': 8, 'en': 23, 'python': 1, 'es': 19, 'clave': 2, 'para': 22, 'el': 39, 'trabajo': 1, 'co
```

Figura 5. Cuenta cuántas veces aparece cada palabra en los documentos y almacenará esa información en el diccionario.

```
[9] diccionario_ordenado = dict(sorted(cache.items(), key=lambda item: item[1], reverse=True))

[10] print(diccionario_ordenado)

     {'proceso': 68, 'constante': 68, 'mitigación': 68, 'riesgos': 68, 'masivos': 66, '(big': 66, 'data)': 66, 'abre':
```

Figura 6. Un nuevo diccionario de orden.

IV. Conclusiones

- En conclusion podemos afirmar que este informe detalla la implementación de un algoritmo que utiliza la técnica de Memoization para calcular las clasificaciones de las palabras más repetidas en documentos y desarrollar una función de búsqueda de palabras en estos documentos. Se ha realizado un análisis de la complejidad algorítmica del código, concluyendo que su complejidad total en el peor caso es de O(N * M), donde N es la cantidad de documentos y M es el promedio de palabras por documento. Esto proporciona información valiosa sobre cómo el algoritmo se comporta en diferentes situaciones y su capacidad para manejar conjuntos de datos de diversos tamaños.

Referencias

[1] Oscar Alexander Mendez Aguirre.