Estructuras de Datos y Algoritmos: Reto 02

Juan Sebastián Ortega (Estudiante 1- js.ortergar1@uniandes.edu.co - 202021703) y Yesid Camilo Almanza (Estudiante 2 - y.almanza@uniandes.edu.co - 201921773)

Departamento de Ingeniería de Sistemas y Computación

UNIVERSIDAD DE LOS ANDES

14 de abril de 2021

I. Introducción

Con base en los resultados obtenidos en el anterior *Reto 01*, se busca seguir trabajando sobre la base de datos de *Trending YouTube Video Statistics* (específicamente en los archivos *category-id.csv* y *videos-large.csv*). En este caso, se busca reemplazar la estructura de datos principal con la cual se carga y filtra la información del catalogo de vídeos con el objetivo de optimizar el tiempo de ejecución del algoritmo.

Previamente se había utilizado el TAD Lista de tipo *ARRAY LIST* para cargar todo el catalogo de videos en una lista principal y en base a esta lista se iban generando sub-listas notablemente más pequeñas puesto que eran filtradas por categoría, país y tag. Sin embargo, este modelo de filtrado era ineficiente puesto que se necesitaba iterar múltiples veces sobre el catalogo de videos para generar cada sub-lista filtrada. Este Reto 02 plantea el TAD Map como una alternativa más eficiente puesto que permite realizar un filtrado de complejidad constante (O(k)) a la par que el catalogo es cargado una única vez (O(n)).

La optimización temporal radica en las tablas de hash que el TAD MAP es capaz de interpretar, de forma que el filtrado puede realizarse mediante la asignación de parejas llave-valor, donde la llave representa cada posible requerimiento de filtrado y el valor contiene directamente a la sub lista de videos relacionados con dicha llave. De esta forma, los requerimientos de filtrado como el nombre del país o de la categoría son establecidos como llaves desde que se realiza la carga de datos. En consecuencia, cuando se desee acceder a la información de un país o categoría en específico, solo sera necesario acceder a su respectiva llave, sin la necesidad de iterar.

Requerimientos a cumplir:

1. Encontrar los n videos con más reproducciones (views) que son tendencia para un país y una categoría especifica.
2. Conocer el video con más días como tendencia en un país.
3. Conocer el video con más días como tendencia según una categoría.
4. Conocer los n videos con más Likes (likes) en un país y con una etiqueta (tag) especifica.

En el momento de cumplir con los requerimientos se tuvo en mente la complejidad temporal que iba a adquirir el algoritmo en cada uno de estos, de forma que se tomaron las siguientes decisiones para optimizar la eficiencia:

- Todos los ordenamientos se realizarán mediante la aplicación de **MERGE SORTING** puesto que es el algoritmo de ordenamiento más efectivo dentro de la *DISClib* de acuerdo a pruebas de eficiencia realizadas previamente.
- Todas las listas serán instanciadas como **ARRAY_LIST** debido a que se prioriza la utilización de las funciones $lt.addLast()$ y $lt.getElement()$ las cuales cuentan con una complejidad de $O(K)$ en este tipo de estructura.
- Con el objetivo de prevenir la mayor cantidad de colisiones posibles, se estableció que todos los mapas serian creados de tipo **CHAINING**, teniendo un factor de carga de entre 2.0 y 4.0 (dependiendo de la cantidad de datos).
- El número de elementos en cada mapa se determinó individualmente aproximando la cantidad de llaves que este guardaría y siguiendo la fórmula $mapsize = nextprime(N)$, donde N es el número de llaves a alamacenar. Esto se hizo con el objetivo de evitar re-hash's y ahorrar la mayor cantidad de tiempo de ejecución posible.

II. Comparación de funciones principales con el Reto 01

El algoritmo de la aplicación fue diseñado siguiendo la misma metodología del Reto 01 de tal forma que los diferentes requerimientos comparten las mismas funciones (en su mayoría). Por lo tanto, antes de realizar el análisis general de la complejidad temporal de cada requerimiento es necesario analizar cada función principal individualmente. En este caso, algunas funciones de filtrado presentes en el Reto 01 han sido totalmente desechadas puesto que el TAD Map permite ahorrar el tiempo de su ejecución.

II-A. Funciones de filtro

- filterCatalog: **O(n) —— Desechada**

```
def filterCatalog(catalog, column_1, value_1, column_2=None, value_2=None):
    """Filtra el catalogo dejando solo los videos con el valor especificado
    para maximo 2 columnas determinadas."""


    filtered_catalog = lt.newList("ARRAY_LIST")
    filtered_catalog["videos"] = lt.newList("ARRAY_LIST")

    for video in lt.iterator(catalog['videos']):

        if column_2 is not None:

            if video[column_1] == "_"+value_1 and video[column_2] == value_2:

                lt.addLast(filtered_catalog["videos"], video)
        else:
            if video[column_1] == value_1:
                lt.addLast(filtered_catalog["videos"], video)

    return filtered_catalog
```

Esta función, encargada de filtrar hasta dos columnas a la vez y con una complejidad de **O(n)** ha sido totalmente desechada puesto que el filtrar sobre el catalogo después de su carga ahora es innecesario.

El Reto 02 ahora puede acceder a la lista de vídeos relacionados a un valor de columna específico de país o categoría mediante las lineas de complejidad *O(k)*: $mp.get(catalog["countries"], country-name)$ y $mp.get(catalog["categories"], category-name)$.

- newUniqueCatalog: **O(2n) —— Desechada**

```
def newUniqueCatalog(catalog):
    unique_dict = {"videos": None}
    unique_dict["videos"] = {}
    unique_catalog = lt.newList("ARRAY_LIST")
    pos = 0
    for video in lt.iterator(catalog):
        pos += 1
        try:
            video_info = unique_dict["videos"][video["title"]]
            new_day = lt.getElement(video_info, 1) + 1
            lt.changeInfo(video_info, 1, new_day)
        except:
            unique_dict["videos"][video["title"]] = lt.newList("ARRAY_LIST")
            lt.addLast(unique_dict["videos"][video["title"]], 1)
            lt.addLast(unique_dict["videos"][video["title"]], pos)
            lt.addLast(unique_dict["videos"][video["title"]], video)

    for i in unique_dict["videos"]:
        lt.addLast(unique_catalog, unique_dict["videos"][i]["elements"])


    return unique_catalog
```

Esta función que iteraba sobre todos los videos cargados (complejidad temporal O(n)) y generaba una sub-lista únicamente con los videos únicos ha sido totalmente desechada.

El Reto 02 al cargar los datos por primera vez es capaz de generar la sub-lista de videos únicos en la llave $["unique\_videos"]$ **y de añadirles la información respecto a la cantidad de dias que se repite cada uno. De esta forma, para acceder a los videos únicos solo se hace necesario una operación de complejidad temporal O(k) para acceder a la pareja llave, valor.**

- filterTag: **O(n) —— Conservada**

```
def filterTag(catalog, tag):
    filter_tags=lt.newList("ARRAY_LIST")
    filter_tags["videos"]=lt.newList("ARRAY_LIST")

    for video in lt.iterator(catalog['videos']):

        video_tags=video["tags"]

        if tag in video_tags:
            lt.addLast(filter_tags["videos"], video)

    return filter_tags
```

En esta función se utiliza un *forLoop* para iterar a través de todos los videos, lo que le da al algoritmo una complejidad de *O(n)* que depende de la cantidad de videos cargados. Adicionalmente se analiza si un sub-string (tag) está dentro de un string mayor para lo que se utiliza la función de Python *.split()* la cual dependerá de la cantidad de caracteres (c) en el string, dando una complejidad adicional de *O(c)*. Para efectos prácticos, se tendrá en cuenta el límite de 400 caracteres que tiene la casilla de caracteres en YouTube (siendo este una constante en el peor de los casos), por lo que podrá ser descartada la complejidad O(c).

Esta función permaneció en el Reto 02 ya que a diferencia del método de filtrado de carga aplicado con la categorías y los países (con nombres predefinidos), los tags representan un requisito de filtrado excesivamente diverso al poder tomar literalmente el valor de cualquier string posible. Por lo tanto, el haber añadido una llave para cada tag analizado hubiera resultado en un consumo de espacio exorbitante, por lo que se opto por dejar el método de filtrado en lista del Reto 01.

II-B. Funciones de ordenamiento

- sortVideos: $\mathbf{O(n \cdot log_2(n))}$ **—— Conservada**

```
def sortVideos(catalog, size, cmpFunction):

    if cmpFunction == "sortByViews":
        sub_list = lt.subList(catalog['videos'], 1, size)
        sub_list = sub_list.copy()
        start_time = time.process_time()
        sorted_list = mergesort.sort(sub_list, cmpVideosByViews)
        stop_time = time.process_time()
        elapsed_time_mseg = (stop_time - start_time)*1000
        return elapsed_time_mseg, sorted_list

    elif cmpFunction == "sortByDays":
        sub_list = lt.subList(catalog, 1, size)
        sub_list = sub_list.copy()
        start_time = time.process_time()
        sorted_list = mergesort.sort(sub_list, cmpVideosByDays)
        stop_time = time.process_time()
        elapsed_time_mseg = (stop_time - start_time)*1000
        return elapsed_time_mseg, sorted_list

    elif cmpFunction == "sortByLikes":

        sub_list = lt.subList(catalog['videos'], 1, size)
        sub_list = sub_list.copy()
        start_time = time.process_time()
        sorted_list = mergesort.sort(sub_list, cmpVideosByLikes)
        stop_time = time.process_time()
        elapsed_time_mseg = (stop_time - start_time)*1000
        return elapsed_time_mseg, sorted_list
```

Esta función se encuentra fragmentada en 3 condicionales, no obstante, todos tienen la misma complejidad ya que el único cambio es el valor con el que se realizará la comparación al momento de organizar los datos. Como es posible observar se utiliza **MERGE_SORT** para organizar una lista con n elementos (la cantidad de videos cargados) lo que tiene una complejidad de $O(n \cdot log_2(n))$.

En el Reto 02 se conserva la función de ordenamiento principal del algoritmo. Esto se debe a que a pesar de que los videos ahora se carguen en sub-listas ya filtradas, dichas sub-listas siguen estado

desordenadas. Por lo tanto, en el aspecto de la organización de los datos no existe ninguna otra forma más efectiva de realizar este procedimiento.

III. Requerimiento 01

```
def execute_req1(catalog, req_category, req_country, n_sample):
    """Ejecuta el requerimiento 1"""

    filter_category_entry = mp.get(catalog["categories"], req_category)

    filter_category = me.getValue(filter_category_entry)

    filter_country_entry = mp.get(filter_category["videos"], req_country)

    filter_country = me.getValue(filter_country_entry)

    sorted_catalog = sortVideos(filter_country, lt.size(filter_country), "sortByViews")[1]

    n_sample = validateNSample(n_sample, sorted_catalog)

    filter_nsample = lt.subList(sorted_catalog, 1, n_sample)

    return filter_nsample
```

En el requerimiento 01 se requiere hacer dos filtrados iniciales: país y categoría específica. Como se mostró anteriormente, la sola carga del catalogo se encarga de filtrar todos los casos posibles de forma autónoma por lo tanto solo se hace necesario acceder a la pareja llave valor que contiene la sub-lista del país y categoría especificados, dando una complejidad temporal de **O(K)**. Ya que sigue siendo necesario realizar un ordenamiento de las sub-listas, sigue existiendo el componente de complejidad $\mathbf{O(n \cdot log_2(n))}$ que tiene la función sortVideos. De esta forma, la complejidad temporal del requerimiento 01 se reduce únicamente al componente de organización de los datos, a comparación de su contra parte en el Reto 01 que debía realizar dos iteraciones extra sobre todos los videos.

Complejidad temporal final: $\mathbf{O(n \cdot log_2(n))}$

IV. Requerimiento 02 - Estudiante 1

```
def execute_req2(catalog, req_country):
    """Ejecuta el requerimiento 2"""


    filter_country_entry = mp.get(catalog["countries"], req_country)

    filter_country_map = me.getValue(filter_country_entry)["unique_videos"]

    filter_country = mp.valueSet(filter_country_map)

    sorted_catalog = sortVideos(filter_country, lt.size(filter_country), "sortByDays")[1]


    filter_first_element = lt.subList(sorted_catalog, 1, 1)


    filter_first_item = lt.getElement(filter_first_element,1)

    filter_first_video = lt.getElement(filter_first_item,1)
    filter_first_day = lt.getElement(filter_first_item,2)

    return (filter_first_video, filter_first_day)
```

En el requerimiento 02 se hace necesario filtrar mediante una categoría específica y mediante los videos únicos de forma que se pueda llevar una cuenta del número de apariciones de cada video (dias en tendencia). Afortunadamente, el TAD Map permite realizar esta acción en la carga de datos por lo que en el requerimiento en sí solamente es necesario acceder a la tabla de hash (complejidad O(k)). Nuevamente la complejidad es definida por el algoritmo de organización de los datos.

Complejidad temporal final: $\mathbf{O(n \cdot log_2(n))}$

V. Requerimiento 03 - Estudiante 2

```
def execute_req3(catalog, req_category):
    """Ejecuta el requerimiento 3"""

    filter_category_entry = mp.get(catalog["categories"], req_category)

    filter_category_map = me.getValue(filter_category_entry)["unique_videos"]

    filter_category = mp.valueSet(filter_category_map)

    sorted_catalog = sortVideos(filter_category, lt.size(filter_category), "sortByDays")[1]

    filter_first_element = lt.subList(sorted_catalog, 1, 1)


    filter_first_item = lt.getElement(filter_first_element,1)
    filter_first_video = lt.getElement(filter_first_item,1)
    filter_first_day = lt.getElement(filter_first_item,2)

    return (filter_first_video, filter_first_day)
```

El requerimiento 03 es bastante similar al requerimiento 02, con la única diferencia que se accede a la una pareja de llave valor que contiene un país en específico en vez de una categoría específica. Nuevamente la complejidad es determinada por el algoritmo de ordenamiento puesto que las funciones de filtrado adquieren una complejidad constante.

Complejidad temporal final: $\mathbf{O(n \cdot log_2(n))}$

VI. Requerimiento 04

```
def execute_req4(catalog, req_country ,req_tag, n_sample):
    """Ejecuta el requerimiento 4"""


    filter_country_entry = mp.get(catalog["countries"], req_country)

    filter_country = me.getValue(filter_country_entry)["videos"]

    filter_tag = filterTag(filter_country, req_tag)

    sorted_catalog = sortVideos(filter_tag, lt.size(filter_tag), "sortByLikes")[1]

    n_sample = validateNSample(n_sample, sorted_catalog)

    filter_nsample = lt.subList(sorted_catalog, 1, n_sample)

    return (filter_nsample)
```

En el caso del requerimiento 04 únicamente se optimiza una de las funciones (la que está relacionada al filtrado de los países), por otra parte, la función filterTag permanece dentro del código así como la función de ordenamiento sortVideos. Como fue mencionado anteriormente, la decisión de mantener filterTag esta relacionada al hecho de que utilizar un TAD Map para modelar todos los tags posibles resultaría exhaustivo para la memoria del sistema por lo que este fue el único caso donde se prefirió mantener el modelo anterior.

Complejidad temporal final: $\mathbf{O(n \cdot log_2(n)) + O(n)}$

VII. Gráficas de complejidad entre máquinas

Con el propósito de realizar un análisis de la complejidad del algoritmo desarrollado en un sistema real, se utilizó la librería *time* y la librería *tracenmalloc* para obtener los datos reales de tiempo de ejecución y espacio ocupado en ambas máquinas. En adición, se implementó la librería *matplotlib* para obtener una representación gráfica de los datos obtenidos.

Para realizar estas pruebas, todos los requerimientos en ambas máquinas fueron ejecutados siguiendo los siguiente parámetros de prueba:

- País: USA
- Categoría: Music

- Número de elementos: 4
- Tag: 2018

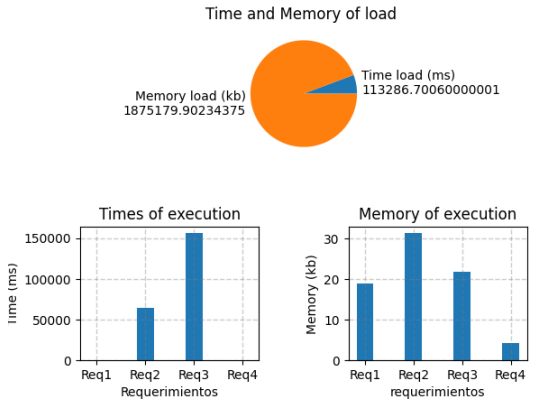

Figura 1. Complejidad Temporal y Espacial: Máquina 1

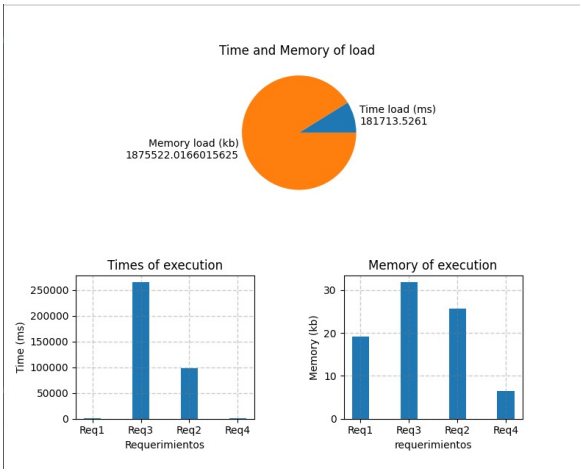

Figura 2. Complejidad Temporal y Espacial: Máquina 2

- **Análisis Complejidad Temporal**: Como se puede observar existe una diferencia notable entre el tiempo de ejecución de los requerimientos. Mientras que los requerimientos 1 y 4 cuentan con una complejidad temporal casi instantánea, los requerimientos 2 y 3 toman considerablemente más tiempo en realizarse. Esto puede estar relacionado el hecho de que estos requerimientos necesitan organizarse comparando un parámetro añadido externamente, por lo que puede existir cierta complicación temporal en el momento de acceder a este valor. En ambas máquinas es posible observar los patrones ya mencionados.

- **Análisis Complejidad Espacial**: En cuanto a la complejidad espacial de cada requerimiento es posible concluir que esta depende directamente de la cantidad de datos que hayan sido filtrados por requerimiento. En el caso de ambas máquinas el requerimiento que utiliza menos espacio es el requerimiento 4, relacionado con el hecho de que este tiene un filtro general por país y posteriormente un filtro bastante específico por tag, lo que hace que se opere con una sub-lista de tamaño muy reducido. Por otra parte, los requerimientos 2 y 3 son los que más espacio utilizan ya que estos únicamente son filtrados por categoría o por país, operando con las sub-listas más grandes.

VIII. Conclusión

En conclusión, es posible afirmar que la implementación de TAD Maps es adecuada para lograr optimizar la complejidad temporal de un algoritmo que necesite filtrar datos. Esto se debe a que como se vio en el contraste entre el Reto 01 y el Reto 02, las tablas de hash (componente interno del TAD Map) permiten acceder a información ya filtrada en la carga de datos mediante una función de búsqueda con complejidad constante. Sin embargo, también es necesario analizar que existe una correlación inversa entre la cantidad de datos utilizados y la reducción de la complejidad temporal. A medida que se deseen tener sub-conjuntos de elementos cada vez más específicos, va a ser necesario generar más sub-listas internamente para que se relacionen con las llaves de filtrado. Por lo tanto, en una implementación de este tipo es completamente necesario encontrar un balance entre la complejidad temporal y la complejidad espacial.

Referencias

[1] R. Sedgewick and K. Wayne, Algorithms, 4th Edition. Addison-Wesley, 2011.