|  |  |  |
| --- | --- | --- |
|  |  |  |

FUNDAMENTOS DE ESTRUCTURAS DE DATOS

Y ANÁLISIS DE ALGORITMOS

**INFORME**

**“Proyecto Semestral”**

Julio 2023

**ERICH GERMÁN GRÜTTNER DÍAZ**

**Índice**

[1. Introducción 3](#_Toc139797987)

[2. Descripción de las estructuras de datos 5](#_Toc139797988)

[2.1 Matriz de adyacencia 5](#_Toc139797989)

[2.2 Listas de adyacencia 5](#_Toc139797990)

[3. Implementación 7](#_Toc139797991)

[3.1 Carga de archivo XML de gran tamaño 7](#_Toc139797992)

[3.2 Creación de índice de autores 7](#_Toc139797993)

[3.3 Implementación de matriz de adyacencia 7](#_Toc139797994)

[3.3 Implementación listas de adyacencia 7](#_Toc139797995)

[3.4 Medición de tiempos de ejecución 8](#_Toc139797996)

[3.5 Construcción del código 8](#_Toc139797997)

[3.6 Generación de gráficos 8](#_Toc139797998)

[4. Experimentación 9](#_Toc139797999)

[4.1 Archivos para graficar 9](#_Toc139798000)

[3.1 Tiempos de ejecución 10](#_Toc139798001)

[3.2 Cantidad de memoria utilizada 10](#_Toc139798002)

[3.3 Complejidad de la implementación 10](#_Toc139798003)

[5. Conclusiones 12](#_Toc139798004)

[5.1 Comparación de creación de estructuras 12](#_Toc139798005)

[5.2 Comparación de cálculo de coautoría 12](#_Toc139798006)

[5.3 Conclusiones generales 12](#_Toc139798007)

[6. Referencias 14](#_Toc139798008)

# 1. Introducción

Este informe tiene como objetivo presentar un análisis sobre dos soluciones a la problemática de calcular el grafo de coautoría de una comunidad científica: matriz de adyacencia y listas de adyacencia.

Se revisarán sus características, como fueron implementadas en código y el rendimiento que tuvieron en su ejecución.

Adicionalmente, este informa aborda la complejidad de la manipulación de grandes archivos de datos, como la base de datos DBLP que, en su versión más reciente, pesa casi 4GB. Para ello, se describirá la utilización de la librería RapidXML que, como sus siglas lo sugieren, realiza una rápida carga de datos, creando internamente el DOM para poder recorrer el documento.

En este informe se verá también el cálculo de conexiones entre coautores, lo que deriva en información de cantidad de aristas par o impar de publicaciones. Estos datos permiten, dado cierto contexto, inferir eventualmente si es que existe algún tipo de colusión entre autores.

El proceso de cálculo se divide en 3 partes: lectura de archivo en memoria (y creación de grafo), creación de estructuras y finalmente el cálculo de aristas par e impar.

Para el proceso de experimentación se utilizó la misma base de datos original, pero parcelada por cantidad de registros. En particular, desde 10000 hasta 50000 autores.

En el apartado de resultados experimentales, se muestran datos y gráficos resultantes de la ejecución de las diversas tareas en el software. Para ello se utilizó la librería “Chrono” de C++. En tanto que los gráficos se obtuvieron a través de Python usando la librería “Matplotlib”.

Adicionalmente se muestran algunas capturas de pantalla de la ejecución de la librería Networkx, para Python que permite navegar en al grafo generado, a partir de una matriz de adyacencia generada.

Finalmente se muestran las conclusiones que se enfocan principalmente en la comparativa entre soluciones. El uso de matriz de adyacencia, si bien es muy simple de implementar, tiene un alto costo espacial y se verá que en rendimiento, para este caso, tampoco justifica su uso. Por el contrario, las listas de adyacencias tuvieron un rendimiento mucho mejor al esperado.

Este informe se complementa con un repositorio Github en donde se puede encontrar tanto el código utilizado en los diferentes experimentos, como los diferentes archivos de salida y gráficos de análisis correspondientes.

# 2. Descripción de las estructuras de datos

## 2.1 Matriz de adyacencia

En teoría de grafos y ciencias de la computación, una matriz de adyacencia es una matriz cuadrada utilizada para representar un grafo finito. Los elementos de la matriz indican que pares de vértices son adyacentes o no en el grafo.

Para el presente caso, cada vértice representará un autor y cada arista representará una relación de co-autoría entre trabajos académicos.

Normalmente se utiliza una matriz con 0 y 1 para indicar la presencia o ausencia de la relación (arista), pero en este caso se reemplazará el 1 por la cantidad de veces que han realizado, los autores, algún trabajo en conjunto.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | A0 | A1 | A2 | A3 | A4 | A5 | A6 | | A0 | 0 | 1 | 1 | 2 | 1 | 0 | 1 | | A1 | 0 | 0 | 1 | 1 | 2 | 2 | 0 | | A2 | 0 | 0 | 0 | 2 | 1 | 1 | 0 | | A3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | | A4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | | A5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | A6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |  |  | | --- | --- | | Id Autor An | Nombre | | 0 | Hai He | | 1 | Weiyi Meng | | 2 | Yiyao Lu | | 3 | Clement T. Yu | | 4 | Zonghuan Wu | | 5 | Grüttner | | 6 | Barbay | |

Figura 1 - Ejemplo de matriz de adyacencia y listado de autores correspondientes

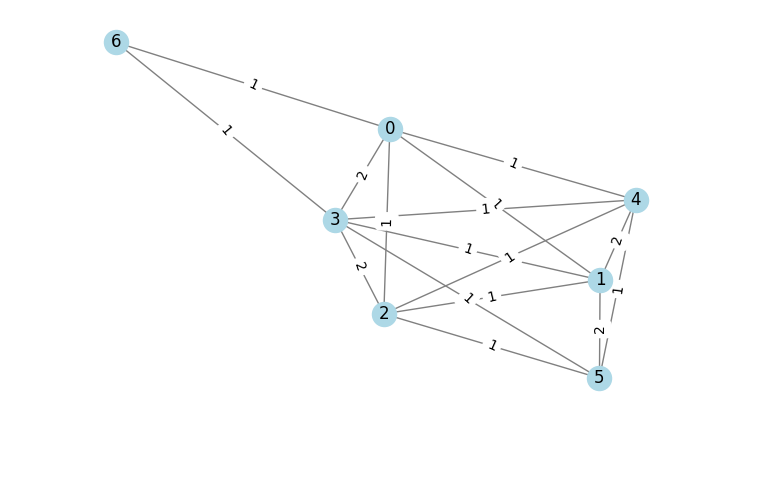


Figura 2 - Grafo resultante de cálculo de coautoría

El algoritmo para realizar la carga de la matriz de adyacencia es el siguiente:

|  |  |
| --- | --- |
| **Algoritmo 1**: Carga de matriz de adyacencia | |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22: | **Previamente:**  ***Doc*** ← *DBLP.XML //Carga de archivo XML en memoria, variable Doc*  **Function** CargaMatrizAdyacencia **is**  **Leer *Doc*** hasta **EOF**  *bloque* ← *Doc.registro*  **If** *bloque*.nombre==”article” or “inproceedings” **Then**  *hijo* ← *bloque.hijo*  **If** *hijo*.nombre==”author” **Then**  *autor\_raiz* ← *hijo.valor*  **Insert** *lista\_autores (autor\_raiz)* → posición = id\_autor1  **While** (*nieto* ← *hijo.hijo) //Mientras hijo tenga más hijos queda en nodo nieto*  **If** *nieto*.nombre==”author” **Then //**Es coautor  coautor ← *nieto.valor*  **Insert** *lista\_autores (nieto\_valor)* → posición = id\_autor2  **MatrizAdyacencia** (id\_autor1,id\_autor2)++; //Marca relación, aumenta en 1 si es que ya existía  **End If**  **End While**  **End If**  **End if**  **Leer siguiente registro de Doc**  **End** |

Notas adicionales:

* Al momento de insertar un nuevo autor a la lista, se busca si es que está registrado devolviendo su posición. Si es nuevo, se inserta al final y se entrega la posición en la que quedó.
* Inicialmente la matriz de adyacencia está con valores 0 en todas sus filas y columnas.
* Para marcar en la matriz se valida que id\_autor1<id\_autor2, sino se insertan al revés. Para que la relación solamente aparezca una vez
* Para la búsqueda solamente se consideró “article” o “improceeding”, pero basta con dejar el código paramétrico para permitir cualquier tipo de bloque de lectura
* El costo esperado de la “marca” sería simplemente O(1), pero se verá en el siguiente capítulo que la complejidad de tiempo se verá afectada por el procesamiento del gran archivo de ingreso

El algoritmo para realizar el cálculo de coautoría es el siguiente:

|  |  |
| --- | --- |
| **Algoritmo 2**: Cálculo de coautoría (pares/impares) | |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24: | **Function** CalculaCoautoría **is**  *pares* ← *0;*  *impares* ← *0;*  *aux* ← *0;*  *max\_cantidad\_autoria* ← *0;*  *id\_autor1* ← *0;*  *id\_autor2* ← *0;*  **For** *i* ← 0 **to** *num\_autores* **do**  **For** *j* ← 0+aux **to** *num\_autores* **do**  **If** (*MatrizAdyacencia*[i][j] mod 2 == 0) //Calcula pares o impares  **If** (*MatrizAdyacencia*[i][j] != 0) //Para no contar los 0  *pares++;*  **End If**  **Else**  *impares++;*  **End If**  **If** (*MatrizAdyacencia*[i][j] > *max\_cantidad\_autoria*) //Calcula mayor relación de coautoría  *max\_cantidad\_autoria* ← *MatrizAdyacencia*[i][j]*;*  *id\_autor1* ← *i;*  *id\_autor2* ← *j;*  **End If**  **End**  *aux++; //Para que solamente busque en la diagonal superior*  **End**  **End** |

* Complejidad de tiempo para construcción
* Complejidad de tiempo para funciones de búsqueda

## 2.2 Listas de adyacencia

En teoría de grafos y en ciencia de la computación, una lista de adyacencia es una colección de listas desordenadas utilizadas para representar un grafo finito. Cada lista desordenada dentro de una lista de adyacencia describe el set de vecinos de un particular vértice en el grafo.

Para el presente caso, el vector central representará la lista de autores, y sus listas asociadas corresponderán a otros autores (coautoría) y la cantidad de veces que han colaborado.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Id Autor An | Lista de Adyacencia  (An, peso arista) | | 0 | (1, 1) (2, 1) (3, 2) (4, 1) (6, 1) | | 1 | (2, 1) (3, 1) (4, 2) (5, 2) | | 2 | (3, 2) (4, 1) (5, 1) | | 3 | (4, 1) (5, 1) (6, 1) | | 4 | (5, 1) | | |  |  | | --- | --- | | Id Autor An | Nombre | | 0 | Hai He | | 1 | Weiyi Meng | | 2 | Yiyao Lu | | 3 | Clement T. Yu | | 4 | Zonghuan Wu | | 5 | Grüttner | | 6 | Barbay | |

Figura 3 - Ejemplo de listas de adyacencia y listado de autores correspondientes

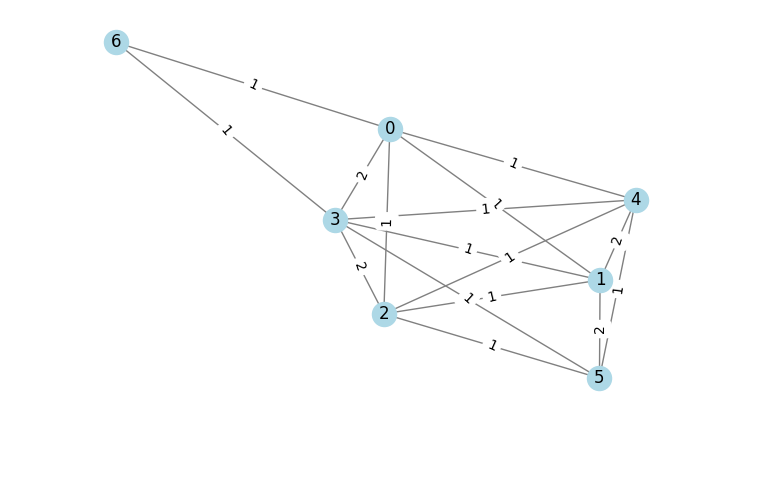


Figura 4 - Grafo resultante de cálculo de coautoría

El algoritmo para realizar la carga de la lista de adyacencia es el siguiente:

|  |  |
| --- | --- |
| **Algoritmo 3**: Carga de lista de adyacencia | |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22: | **Previamente:**  ***Doc*** ← *DBLP.XML //Carga de archivo XML en memoria, variable Doc*  **Function** CargaListaAdyacencia **is**  **Leer *Doc*** hasta **EOF**  *bloque* ← *Doc.registro*  **If** *bloque*.nombre==”article” or “inproceedings” **Then**  *hijo* ← *bloque.hijo*  **If** *hijo*.nombre==”author” **Then**  *autor\_raiz* ← *hijo.valor*  **Insert** *lista\_autores (autor\_raiz)* → posición = id\_autor1  **While** (*nieto* ← *hijo.hijo) //Mientras hijo tenga más hijos queda en nodo nieto*  **If** *nieto*.nombre==”author” **Then //**Es coautor  coautor ← *nieto.valor*  **Insert** *lista\_autores (nieto\_valor)* → posición = id\_autor2  **ActualizaListaAdyacencia** (id\_autor1,id\_autor2);  **End If**  **End While**  **End If**  **End if**  **Leer siguiente registro de Doc**  **End** |

* Ejemplo de implementación
* El algoritmo de construcción fue el siguiente
* El algoritmo para buscar pares e impares
* Complejidad de tiempo para construcción
* Complejidad de tiempo para funciones de búsqueda

# 3. Implementación

El código de la aplicación construida se encuentra en el repositorio [[5]](#cinco)

## 3.1 Carga de archivo XML de gran tamaño

## 3.2 Creación de índice de autores

## 3.3 Implementación de matriz de adyacencia

Se utilizó la librería “AVL-Tree”, desarrollada por Kadir Emre Oto en 2018 [[6]](#seis)

Según su autor, el AVL Tree que implementa esta librería corresponde a un árbol binario auto-balanceado, que garantiza un complejidad de tiempo O(Log N) para la inserción, borrado y búsqueda. Para mostrar el árbol y limpiarlo, en cambio, se garantiza una complejidad de tiempo de O(n).

Permite almacenar cualquier tipo de dato que permita la operación “mayor que (>)”. En particular para este informe se utilizarán long y string.

## 3.3 Implementación listas de adyacencia

Se utilizó un código base generado por ChatGPT y luego modificado y adaptado a las necesidades de la tarea.

Fueron creadas clases separadas para manejar User\_Id y User\_Name, con sus tipos de datos propios (long y string, respectivamente).

Para el caso de string, la función transforma cada elemento del dato de entrada a int multiplicando por 31 (número primo). Con el resultado se realiza la división entera contra el tamaño de la tabla. Lo que es considerado un Hash de división.

Se utilizó una tabla de tamaño 10000.

# 4. Experimentación

4.1 Datasets

Se utilizaron dos datasets: el archivo oficial dblp.xml (3,91 GB) y un dataset propio denominado test.xml (5 KB), que contiene extractos de información para realizar pruebas. Ambos datasets comparten el archivo de definición dblp.dtd.

Una vez realizada la carga en memoria del archivo, internamente se realizaron pseudo datasets, parcelando la carga en estructuras con parámetros limitantes. Por ejemplo: sólo se hicieron cargas de tags “asdasd” y “dsfsdfd”. Y también se limitó la carga por cantidad de registros.

El máximo de registros que se pudieron cargar con matriz fue

El máximo de registros que se pudieron cargar con lista fue

Para el caso de datos no “encontrables” se aplicó el siguiente algoritomo

* Si se procesa User\_Id, se toma el dato desde el registro de datos “encontrables” y se le aumenta un número de su valor
* Si se procesa User\_Name, se toma el dato desde el registro datos “encontrables” y se le concatena una letra “X” para que no aparezca en el resultado

## 3.4 Medición de tiempos de ejecución

Se utilizó la librería “Chrono” de STL, para medir la ejecución del código.

Cada tarea ejecutada fue medida en base a un promedio de ejecución de 10 intentos.

Cada ejecución se almacena en un archivo CSV con los tiempos para cada tamaño de dataset.

## 3.5 Construcción del código

Se realizó en base a **make**, lo que permite realizar ejecuciones programáticas en base a parámetros de entrada del programa.

Adicionalmente, se incorporó un archivo de tipo Bash para ejecutar otras tareas como limpieza y carga del programa.

## 3.6 Generación de gráficos

Finalmente, con los datos obtenidos en formato CSV, se utilizó un script en lenguaje Python, que utiliza la librería **MatPlotLib** [**[7]**](#siete) y que permite generar gráficos customizados de acuerdo a las pruebas requeridas por el proyecto.

## 4.1 Archivos para graficar

Para facilitar la generación de gráficos para la observación del rendimiento de los algoritmos, la ejecución del programa genera archivos del tipo CSV con el siguiente formato:

|  |
| --- |
| tarea\_results.csv |
| **Tamaño muestra, Tiempo[ms]** |
| **valor 1, tiempo 1** |
| **…** |
| **valor n, tiempo n** |

Estos archivos se pueden acceder en el repositorio en [[9]](#nueve)

## 3.1 Tiempos de ejecución

Para la realización de las pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria. El chip M1 tiene 8 núcleos (4 de alta eficiencia a 3.2 GHz + 4 de alto rendimiento a 2.0 GHz) y una velocidad de transferencia de 50Gb por segundo.

Los núcleos de alto performance tienen un caché de instrucciones L1 de 192 KB, un caché de datos L1 de 128 KB, y comparten un caché L2 de 12 MB. Mientras que los núcleos de alta eficiencia energética tienen caché de instrucciones L1 de 128 KB, un caché de datos L1 de 64KB y comparten un caché L2 de 4MB. El SoC también tiene un caché a nivel de sistema, de 8MB que es compartido por el GPU.

## 3.2 Cantidad de memoria utilizada

Se seleccionaron los libros “Alice’s Adventures in Wonderland”, de Lewis Carroll (3760 líneas) [[10]](#diez) y “Metarmorphosis”, de Frank Kafka [[11]](#once), con 2267 líneas.

Inicialmente se intentó comparar los libros completos, pero dado el alto costo espacial (matriz m\*n), que se traduce en requerimiento de memoria RAM, no pudo ser realizado.

Por lo tanto se dividieron los libros en 5 bloques proporcionales a su tamaño total. Cada ejecución tomó una versión de cada libro y procesó la distancia utilizando ambos algoritmos. El dataset está disponible en [[12]](#doce)

## 3.3 Complejidad de la implementación

Se utilizó como base el libro “Alice’s Adventures in Wonderland”, reducido a 100 líneas. Luego se prepararon 20 copias con el siguiente cambio: insertar un espacio y cambiar un letra. El cambio se realizó de manera incremental. El dataset está disponible en [[12]](#doce)

Si bien no es posible determinar una curva de tendencia a priori, sí es notable la diferencia de tiempo hacia el rendimiento de la función adaptativa, que en este escenario se muestra muy superior.

Ambas soluciones utilizan una matriz de enteros (int), de tamaño (m+1) x (n+1), donde m=largo string S y n=largo string T.

Inicialmente se intentó comparar ambos libros, “Alicia” (3758 líneas, 164015 caracteres) con “Metamorfosis” (2266 líneas, 138407 caracteres). Esta combinación requiere de una matriz de enteros de 164016 x 138407. Asumiendo que el tamaño del tipo de datos **int** es de 4 bytes:

* Tamaño en bytes = 164016 x 138407 x 4 = 905,093,275,904 bytes
* Tamaño en megabytes = 863,286.8 MB
* Tamaño en gigabytes = 842.2 GB

Estos requerimientos de tamaño hacen prácticamente imposible realizar el cálculo en un computador tradicional. Es por ello que se parcelaron los archivos en trozos más pequeños de hasta máximo 500 líneas: “Alicia” (25041 caracteres) y “Metamorfosis” (30042). Lo que genera una matriz de int de 25042 x 30043

* Tamaño en bytes = 25042 x 30043 x 4 = 3,002,545,368 bytes
* Tamaño en megabytes = 2,864.6 MB
* Tamaño en gigabytes = 2.8 GB

Que, evidentemente, es mucho más manejable.

Para el resto de las pruebas se utilizaron archivos más pequeños de 100 líneas.

# 5. Conclusiones

En base a las hipótesis planteadas en el comienzo, y a las que se fueron generando en el transcurso del desarrollo del presente informe, se describen conclusiones en varios aspectos relevantes de la experimentación.

## 5.1 Comparación de creación de estructuras

|  |  |
| --- | --- |
|  |  |

* Se aprecia una diferencia de rendimiento entre árbol y tabla hash (con hashing abierto y cerrado). Entre la búsqueda de un dataset pequeño y uno grande hay un costo casi 4 veces superior.

## 5.2 Comparación de cálculo de coautoría

## 5.3 Conclusiones generales

* El impacto del uso de una clave de tipo long (User\_Id) vs string (User\_Name) depende exclusivamente de la implementación en el código. En las pruebas realizadas solamente se observó un efecto en la inserción del árbol binario de búsqueda.
* El proceso de inserción y de búsqueda en Tablas Hash presenta un mejor rendimiento que en un árbol binario de búsqueda. La diferencia se hace mayor utilizando muestras más grandes de datos.
* Para la implementación utilizada, no se ven grandes diferencias de rendimiento entre Tabla Hash con hashing abierto y hashing cerrado, salvo, por supuesto, a nivel de complejidad de espacio, donde la primera necesita memoria adicional para las listas enlazadas, mientras que la segunda opción requiere una tabla de al menos el tamaño de la muestra a probar.
* Existen más variaciones que se podrían probar a futuro como diferentes funciones de hashing, otro tipo de árbol u otras funciones de sondeo.

# 6. Referencias

[1] Wikipedia, Árbol binario. [En línea]. Disponible: https://es.wikipedia.org/wiki/Árbol\_binario

https://es.wikipedia.org/wiki/Matriz\_de\_adyacencia

[2] Wikipedia, AVL Tree. [En línea]. Disponible: https://en.wikipedia.org/wiki/AVL\_tree

<https://en.wikipedia.org/wiki/Adjacency_list>

[3] Wikipedia, Árbol rojo-negro. [En línea]. Disponible: https://es.wikipedia.org/wiki/Árbol\_rojo-negro

[4] Wikipedia, Tabla Hash. [En línea]. Disponible: https://es.wikipedia.org/wiki/Tabla\_hash

[5] Github, Código fuente. [En línea]. Disponible: https://github.com/egruttner/FEDA-Informe3/code

[6] Github, AVL-Tree. [En línea]. Disponible: https://github.com/KadirEmreOto/AVL-Tree

[7] Matplotlib. [En línea]. Disponible: https://matplotlib.org

[8] Github, Datasets. [En línea]. Disponible:

https://github.com/egruttner/FEDA-Informe3/code/datasets

[9] Github, Archivos CSV. [En línea]. Disponible: https://github.com/egruttner/FEDA-Informe3/code/csv

[10] Freecodecamp, TimeComplexityChart. [En línea]. Disponible: https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/