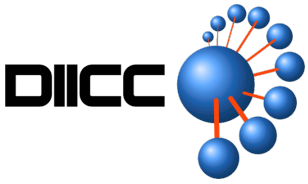


FUNDAMENTOS DE ESTRUCTURAS DE DATOS Y ANÁLISIS DE ALGORITMOS

INFORME

"Proyecto Semestral"

Julio 2023

ERICH GERMÁN GRÜTTNER DÍAZ

Índice

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 informe 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. Soluciones implementadas

2.1 Matriz de adyacencia (Básica)

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. [1]

Para el presente caso, cada vértice representará un autor y cada arista representará una relación de coautoría 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.

	A_0	A_1	A_2	A_3	A_4	A_5	A_6
A_0	0	1	1	2	1	0	1
A_1	0	0	1	1	2	2	0
A_2	0	0	0	2	1	1	0
A_3	0	0	0	0	1	1	1
A_4	0	0	0	0	0	1	0
A_5	0	0	0	0	0	0	0
A_6	0	0	0	0	0	0	0

Id Autor A_n	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

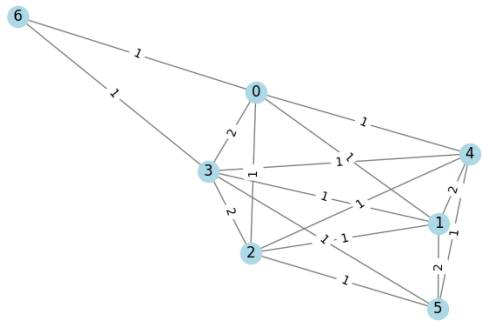


Figura 2 - Grafo resultante de cálculo de coautoría

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

Algoritmo 1: Carga de matriz de adyacencia

```
1:  Previamente:
2:  Doc ← DBLP.XML //Carga de archivo XML en memoria, variable Doc
3:  Function CargaMatrizAdyacencia is
4:    Leer Doc hasta EOF
5:      bloque ← Doc.registro
6:      If bloque.nombre=="article" or "inproceedings" Then
7:        hijo ← bloque.hijo
8:        If hijo.nombre=="author" Then
9:          autor_raiz ← hijo.valor
10:         Insert lista_autores (autor_raiz) → posición = id_autor1
11:         While (nieto ← hijo.hijo) //Mientras hijo tenga más hijos queda en nodo nieto
12:           If nieto.nombre=="author" Then //Es coautor
13:             coautor ← nieto.valor
14:             Insert lista_autores (nieto_valor) → posición = id_autor2
15:            MatrizAdyacencia (id_autor1,id_autor2)++; //aumenta en 1 si ya existía
16:           End If
17:         End While
18:       End If
19:     End if
20:   Leer siguiente registro de Doc
21: End
```

Notas adicionales:

- Al momento de insertar un nuevo autor a la lista de autores, 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.
- 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
- Para medir la complejidad de tiempo, debe considerarse los siguientes procesos:
    - Carga del XML en memoria = O(n)
    - Recorrer cada bloque del XML, y luego cada autor dentro del bloque = O(n) x O(n-1) x O(n-2) x .... ≈ $O(n^2)$
    - Marcar en matriz el peso = O(1)

  Por lo tanto se puede indicar un costo dentro de $O(n^2)$

El algoritmo para realizar el cálculo de coautoría con matriz de adyacencia es el siguiente:

Algoritmo 2: Cálculo de coautoría (pares/impares)

```
1:  Function CalculaCoautoríaMatriz  is
2:    pares ← 0;
3:    impares ← 0;
4:    aux ← 0;
5:    max_cantidad_autoria ← 0;
6:    id_autor1 ← 0;
7:    id_autor2 ← 0;
8:    For i ← 0 to num_autores do
9:      For j ← 0+aux to num_autores do
10:       If (MatrizAdyacencia[i][j] mod 2 == 0) //Calcula pares o impares
11:         If (MatrizAdyacencia[i][j] != 0) //Para no contar los 0
12:           pares++;
13:         End If
14:       Else
15:         impares++;
16:       End If
17:
18:       If (MatrizAdyacencia[i][j] > max_cantidad_autoria) //Calcula mayor relación de
19: coautoría
20:           max_cantidad_autoria ← MatrizAdyacencia[i][j];
21:           id_autor1 ← i;
22:           id_autor2 ← j;
23:       End If
24:     End
25:     aux++; //Para que solamente busque en la diagonal superior
26:   End
27: End
```

Notas adicionales:

- Además de calcular la cantidad de coautorías pares/impares, este algoritmo busca la máxima cantidad de coautorías y sus respectivos autores.
- Se utiliza la variable aux para poder realizar un cálculo de la diagonal superior de la matriz.
- La complejidad de tiempo de este algoritmo está dentro de $O(n^2)$, ya que recorre la matriz de tamaño n x n.

2.2 Listas de adyacencia (Avanzada)

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. [2]

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 A_n	Lista de Adyacencia (A_n, 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 A_n	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

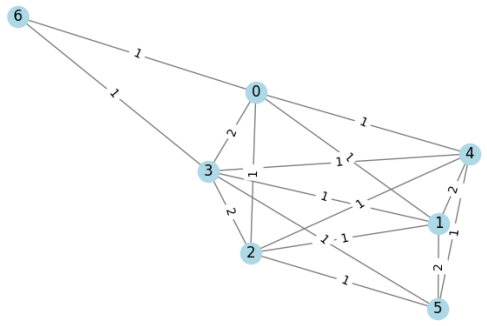


Figura 4 - Grafo resultante de cálculo de coautoría

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

Algoritmo 3: Creación de lista de adyacencia

```
1:  Previamente:
2:  Doc ← DBLP.XML //Carga de archivo XML en memoria, variable Doc
3:  Function CargaListaAdyacencia  is
4:    Leer Doc hasta EOF
5:      bloque ← Doc.registro
6:      If bloque.nombre==”article” or “inproceedings” Then
7:        hijo ← bloque.hijo
8:       If hijo.nombre==”author” Then
9:         autor_raiz ← hijo.valor
10:        Insert lista_autores (autor_raiz) → posición = id_autor1
11:        While (nieto ← hijo.hijo) //Mientras hijo tenga más hijos queda en nodo nieto
12:          If nieto.nombre==”author” Then //Es coautor
13:            coautor ← nieto.valor
14:            Insert lista_autores (nieto_valor) → posición = id_autor2
15:           ActualizaListaAdyacencia (id_autor1,id_autor2);
16:          End If
17:        End While
18:       End If
19:     End if
20:    Leer siguiente registro de Doc
21: End
```

Notas adicionales:

- Al momento de insertar un nuevo autor a la lista de autores, 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.
- Cada nodo de la lista contiene el valor del autor asociado más la cantidad de conexiones (peso).
- La función ActualizaListaAdyacencia realiza las siguientes tareas: busca al autor1 en la lista y luego al autor2 en los nodos asociados. Si no existe la relación, crea un nodo con el autor 2 y el peso 1, y si existe simplemente aumenta el peso en 1.
- Para medir la complejidad de tiempo, debe considerarse los siguientes procesos:
    - Carga del XML en memoria = O(n)
    - Recorrer cada bloque del XML, y luego cada autor dentro del bloque = O(n) x O(n-1) x O(n-2) x …. = $O(n^2)$
    - Llamar a ActualizaListaAdyacencia = O(1) + O(m) donde m=número de nodos, y m <<<< n.

  Por lo tanto se puede indicar un costo dentro de $O(n^2)$

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

Algoritmo 4: Cálculo de coautoría (pares/impares) para lista de adyacencia

```
1:  Function CalculaCoautoría is
2:    pares ← 0;
3:    impares ← 0;
4:    max_cantidad_autoria ← 0;
5:    id_autor1 ← 0;
6:    id_autor2 ← 0;
7:    For i ← 0 to num_autores do
8:      For coautor ← nodo de ListaAdyacencia do
9:        If (coautor.second mod 2 == 0) //Calcula pares o impares
10:         If (coautor.second != 0) //Para no contar los 0
11:           pares++;
12:         End If
13:       Else
14:         impares++;
15:       End If
16:
17:       If (coautor.second > max_cantidad_autoria) //Calcula mayor relación de coautoría
18:         max_cantidad_autoria ← coautor.second;
19:         id_autor1 ← i;
20:         id_autor2 ← j;
21:       End If
22:     End
23:   End
24: End
```

Notas adicionales:

- Además de calcular la cantidad de coautorías pares/impares, este algoritmo busca la máxima cantidad de coautorías y sus respectivos autores.
- En este caso, el cálculo se realiza recorriendo todo el listado de autores y contando la información almacenada en sus nodos asociados.
- La complejidad de tiempo está dentro de O(n x m), donde n=número de autores y m=número de nodos, y m, en el tiempo, para este caso tiene a ser mucho más pequeño que n. Por lo tanto no alcanzaría a estar dentro de $O(n^2)$

3. Análisis y comparación de las soluciones implementadas

El código de la aplicación construida se encuentra en el repositorio [6]

3.1 Carga de archivo XML de gran tamaño

Luego de analizar varias librerías tales como TinyXML, irrXML, Xerces-C++, y pugiXML, entre otras, se optó por la librería RapidXML [3], del autor Marcin Kalicinski, principalmente por su rapidez y facilidad de integración al código.

Esta librería carga el archivo XML e internamente crea el árbol DOM asociado al documento, el cual se puede recorrer con métodos dispuestos para ello.

En el sitio web oficial se dan las razones de la rapidez:

- Parseo in-situ. No hace copias de strings, como nombres y valores del nodo, sino que guarda punteros al interior de la fuente de texto.
- Uso extensivo de tablas de búsqueda (lookup tables) para parseo
- Librería “optimizada a mano” para operar con lo más populares CPUs

Esta librería tiene una complejidad de tiempo, para parsear un documento XML, típicamente lineal O(n), donde n es el tamaño del documento XML. Acceder a una rama tiene un costo de O(1).

3.2 Datasets

Se utilizaron tres datasets: el archivo oficial dblp.xml (3,91 GB), una versión del 02-03-2015 del mismo archivo (1,7 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. [4] [5]

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 “article” y “inproceedings”.

Se realizaron pruebas con cargas de 10000, 20000, 30000, 40000 y 5000 autores.

3.3 Medición de tiempos de ejecución

Se utilizó la librería “Chrono” de STL, para medir la ejecución del código.
Cada ejecución se almacena en un archivo CSV con los tiempos para cada tamaño de dataset.

3.4 Pruebas de dataset y gráficos

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.

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 [6]

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

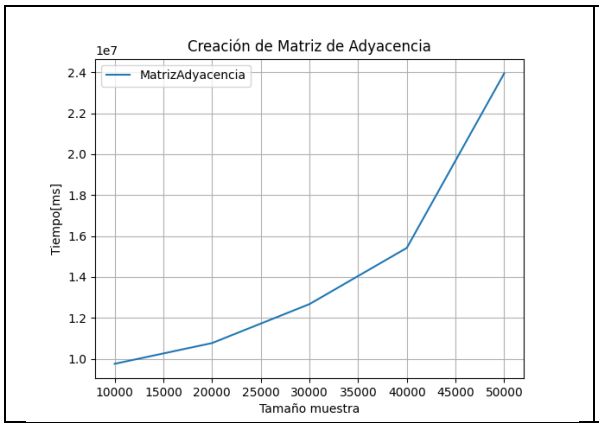
3.5 Equipo de prueba

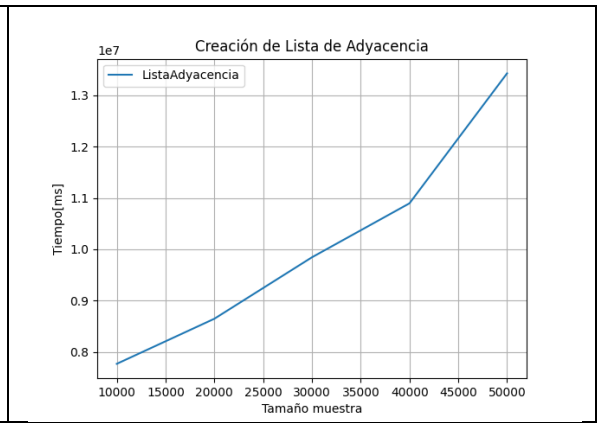
Para la realización de las pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria. El chip M1 tiene 8 núcleos (4 de alta eficiencia a 3.2 GHz + 4 de alto rendimiento a 2.0 GHz) y una velocidad de transferencia de 50Gb por segundo.

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.

3.6 Tiempos de ejecución

3.6.1 Creación de estructuras

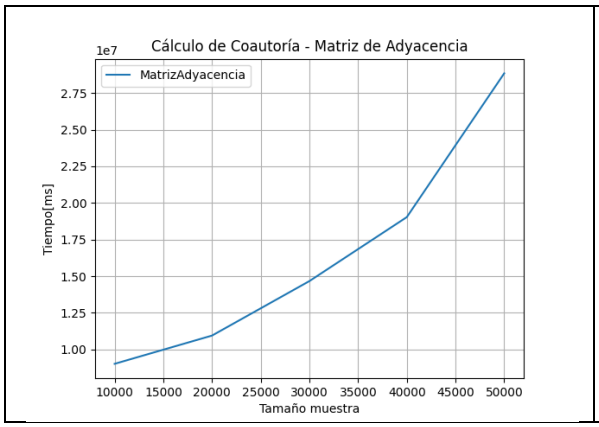


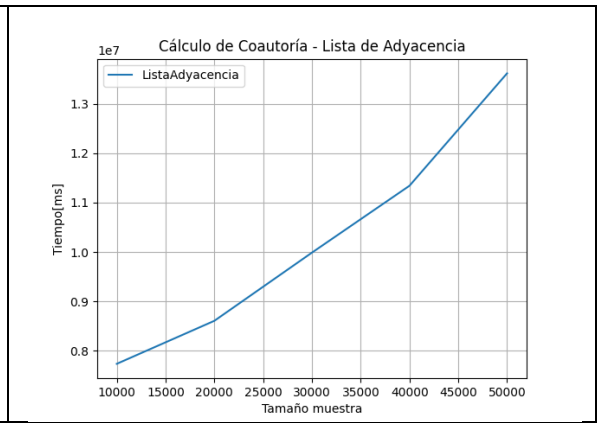


Tamaño muestra	Tiempo[ms]
10000	2.01488e+07
20000	5.49517e+07
30000	1.15201e+08
40000	1.97256e+08
50000	3.0313e+08

Tamaño muestra	Tiempo[ms]
10000	7.85176e+06
20000	8.56871e+06
30000	9.75848e+06
40000	1.10931e+07
50000	1.35427e+07

3.6.2 Cálculo de coautoría





Tamaño muestra	Tiempo[ms]
10000	2.04755e+07
20000	5.55855e+07
30000	1.18073e+08
40000	1.99931e+08
50000	3.16137e+08

Tamaño muestra	Tiempo[ms]
10000	8.75075e+06
20000	9.21321e+06
30000	1.04093e+07
40000	1.18513e+07
50000	1.44347e+07

3.7 Cantidad de memoria usada

Para realizar este cálculo, deben ser consideradas todas las partes que componen la solución:

- **Archivo base**: va desde los 7KB (archivo de prueba) hasta los **3,91GB** (archivo oficial). Toda esa información se carga en memoria antes de empezar
- **Vector de autores**: se almacenan todos los autores en un vector de strings, considerando la máxima carga de 50000 datos.
  Tamaño del vector = tamaño de un elemento x número de elementos
  = (20 bytes + 8 bytes) x 50000
  = 28 bytes x 50000
  = 1,400,000 bytes
  = 1.4 megabytes
- **Matriz de adyacencia**: se almacenan todos los autores en un vector de strings, considerando la máxima carga de 50000 datos.
  Tamaño de la matriz = tamaño de un elemento x número de elementos
  = 4 bytes x 50000 x 50000
  = 10000000000 bytes
  = 10 gigabytes
- **Lista de adyacencia:** corresponde a un vector de enteros más nodos asociados. Para 50000 datos, se estiman no más de 40 nodos:
  Tamaño del vector de enteros = tamaño de un elemento x número de elementos
  = 4 bytes x 50000
  = 200000 bytes
  = **200 kilobytes**
  Tamaño de las listas de adyacencia = tamaño de un elemento x número de elementos x número de nodos
  = 4 bytes x 10 x 40
  = 1600 bytes
  = **1.6 kilobytes**
  Uso total de memoria = Tamaño del vector de enteros + Tamaño de las listas de adyacencia
  = 200 kilobytes + 1.6 kilobytes
  = **201.6 kilobytes**

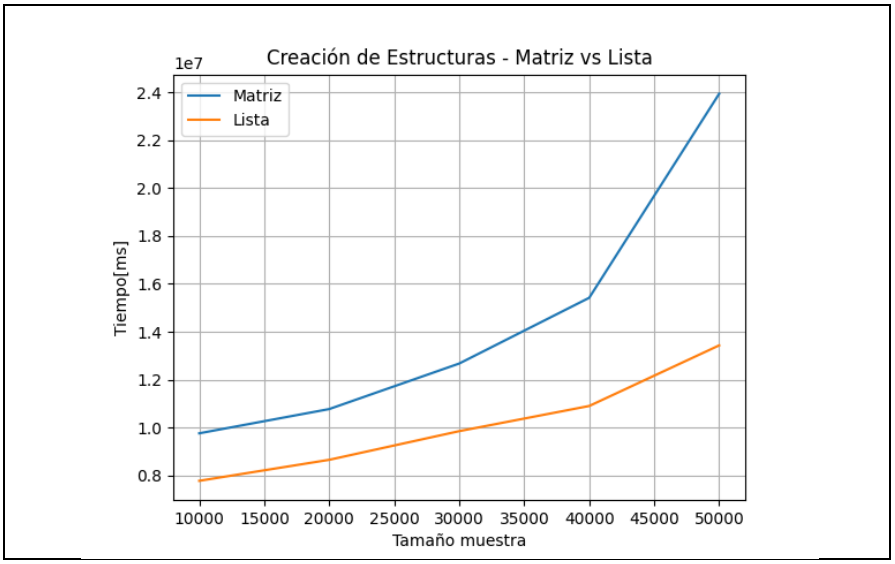
3.8 Complejidad de implementación

- Ambas estructuras fueron implementadas en C++ utilizando clases separadas, donde cada una contiene sus métodos de creación y cálculo de coautoría.
- La construcción del código de matriz de adyacencia fue más sencillo debido a que no requiere tantas búsquedas para procesar.
- Pero sin duda el aspecto más complejo fue la carga y procesamiento de un gran volumen de datos. La memoria disponible siempre se mostró insuficiente para procesar toda la información. De hecho, ese fue el motivo para parcelar los datos.
- La complejidad espacial que requiere el uso de la matriz de adyacencia hace necesario contar con una gran cantidad de memoria RAM.
- Fue muy importante comprender la estructura DOM del documento XML, para ello fue fundamental estudiar el archivo DBLP.DTD, que contiene definiciones de valores y atributos.

4. 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.

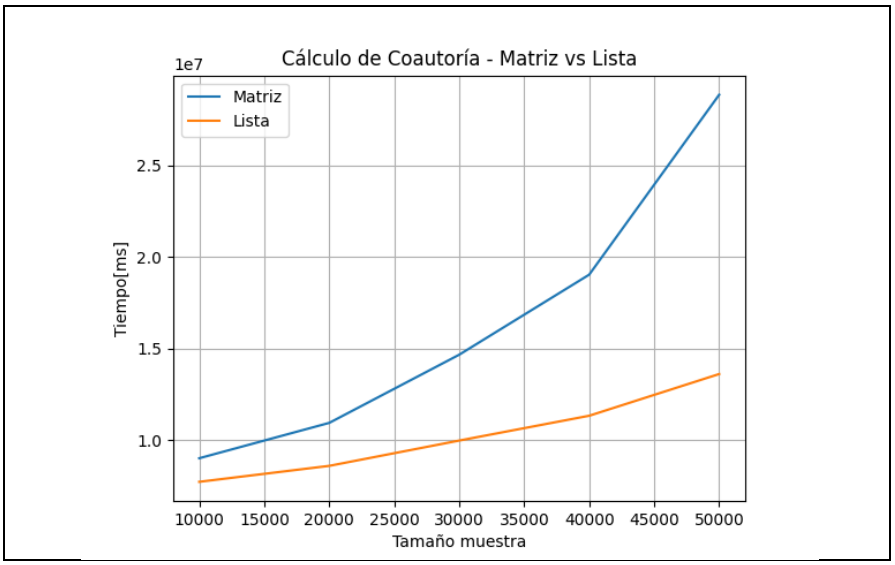
4.1 Comparación de creación de estructuras



- Si bien ambas sugieren una curva más o menos próxima a $O(n^2)$, es probable que sea el resultado de la carga y lectura del árbol DOM del XML, que tiene ese orden. La ejecución básica de actualizar la matriz o la lista quedan contenidas dentro de esa complejidad de tiempo.

- El desempeño de la lista de adyacencia es sorprendentemente mejor que el de la matriz de adyacencia. Se esperaba que la “marca” que se hace en la matriz fuese menos costosa que la operación que se hace sobre la lista, pero en vista a los resultados se ve que es más costoso ubicar una coordenada en una matriz que pesa 16Gb.

4.2 Comparación de cálculo de coautoría



- De acuerdo a lo esperado, el rendimiento de la lista de adyacencia es muy superior, en la medida que aumenta el tamaño de la muestra.
- Aunque el algoritmo de cálculo de matriz de adyacencia utiliza la mitad de la matriz, igualmente de denota su característica cuadrática, mientras que el algoritmo de listas de adyacencia, al contener tantos nodos, se aleja de ella.

4.3 Conclusiones generales

- La lista de adyacencia tiene un desempeño mejor tanto en la creación de la estructura como en el cálculo de la coautoría.
- Respecto al costo espacial, también resulta mejor la opción de lista de adyacencia, ya que, si bien también requiere una lista de autores de tamaño n, la cantidad de nodos asociados nunca es tan grande como la dimensión m de la matriz de adyacencia.
- Se demostró de que, pese a que existen herramientas especializadas para el procesamiento de grandes volúmenes de datos, como ETL o Bases de datos, es posible construir una solución en C++ que haga algo similar.
- Es posible pensar en una tercera solución que implica utilizar solamente la librería RapidXML. El software recorrería las ramas y agregaría un atributo de peso. Con el árbol “pesado” solamente bastaría con recorrerlo nuevamente por completo y determinar pares/impares.

5. Referencias

[1] Wikipedia, Matriz de Adyacencia. [En línea]. Disponible: https://en.wikipedia.org/wiki/Adjacency_matrix

[2] Wikipedia, Lista de Adyacencia. [En línea]. Disponible: https://en.wikipedia.org/wiki/Adjacency_list

[3] RapidXML, Librería. [En línea]. Disponible: https://rapidxml.sourceforge.net

[4] Dblp, Computer Science Bibliography. [En línea]. Disponible: https://dblp.org

[5] Dblp, Database Releases. [En línea]. Disponible: https://dblp.org/xml/release/

[6] Github, Proyecto Semestral. [En línea]. Disponible: https://github.com/egruttner/FEDA-Proyecto-Semestral

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