UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Compresión para multiplicación de matrices mediante bicliques en grafos con pesos

Nicolás Araya Durán

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Informático

Profesor Guía:
Cecilia Hernández Rivas
Comisión Evaluadora:
Jérémy Barbay
Javier Vidal

Marzo 2024
Concepción, Chile

Agradecimientos

Quiero agradecer en primer lugar a mi Familia, por apoyarme durante esta aventura en la universidad. A mis padres por su esfuerzo y dedicación durante toda mi vida para que yo pudiera llegar a este punto. A enseñarme que las cosas buenas nunca son fáciles, pero siempre vale la pena intentar y luchar por ello.

Agradecer a las personas que me acompañaron durante esta etapa de mi vida, a *José Zagal* por participar junto a mi en este proyecto y por la ayuda brindada. Agradecer a cada uno de los profesores de la universidad y del colegio que me nutrieron de conocimiento para ser la persona que soy ahora. En especial a la *Profesora Cecília Hernández* por su enseñanza y por confiar en mí para llevar este proyecto.

Agradecer al equipo de *DFT* de *Synopsys*, en especial a *Rubén*, *Felipe* y *Nelson* por darme la oportunidad de formar parte del equipo durante mi práctica profesional y por el apoyo brindado.

Y finalmente, a mis abuelos les dedico en especial esto, que me vieron y se alegraron de que entrara a la universidad, aunque probablemente no supieran mucho de que iba la carrera que elegí, estuvieron ahí apoyándome desde el principio. Y aunque ahora no estén para verlo, sé que de alguna manera están presentes con su cariño y apoyo igual que durante toda mi vida.

Sumario

Las operaciones sobre matrices constituyen un área crucial en las ciencias de la computación, especialmente debido a su uso extensivo en diversas disciplinas para representar datos, y al constante crecimiento de estos datos. Por ende, resulta fundamental proponer algoritmos eficientes para operar sobre conjuntos de datos cada vez más grandes.

Entre las aplicaciones destacadas de las matrices se encuentra la representación de grafos. En un trabajo anterior realizado por Hernández y Navarro [1] en 2014, propusieron un método para la compresión de grafos utilizando representaciones matriciales. Su enfoque se centró en la búsqueda y extracción de bicliques con el objetivo de eliminar subgrafos densos dentro del grafo, lo que resultó en una representación comprimida eficiente del mismo.

Esta memoria de título se inspira en el enfoque propuesto por Hernández y Navarro [1], ampliando su aplicación a grafos de mayor tamaño y a grafos con pesos o ponderados. Se busca también aprovechar esta compresión para reducir el tiempo de cómputo en operaciones entre matrices, proponiendo un método alternativo para la multiplicación matriz-matriz.

Para realizar la evaluación se utilizaron grafos generados artificialmente. Se busca analizar tanto las características de los grafos como de los bicliques, con el fin de medir la capacidad de búsqueda y extracción del algoritmo propuesto así como también medir el impacto que pueden en las operaciones matriciales.

Los resultados muestran por un lado que el algoritmo es capaz de recuperar las aristas de los bicliques de los grafos y por otro lado muestran que los tiempos de multiplicación de matrices si se ven beneficiados por la compresión obtenida, dando mejores resultados en escenarios donde la compresión en mayor.

Índice general

Índice de cuadros

Índice de figuras

Índice de algoritmos

Capítulo 1

Introducción

1.1. Motivación

El estudio de grafos es un área en continuo crecimiento, mediante estos es posible modelar datos de diversas disciplinas. Algunas posibles aplicaciones con grafos incluyen las interacciones entre grupos de personas o comunidades, la estructura de internet, caminos entre ciudades, problemas de optimización, análisis de datos, aprendizaje de máquina, procesamiento de imágenes, análisis de redes y teoría de grafos.

Considerando el crecimiento de internet, el manejo de grandes volúmenes de datos es cada vez más común. Por lo tanto, encontrar métodos que permitan trabajar sobre estos datos de manera más eficiente es un área atractiva de estudio. No es coincidencia que junto con esto, las publicaciones relacionadas a su estudio hayan aumentado en los últimos años [2].

Una de las representaciones más comunes y utilizadas para representar grafos es utilizar matrices de adyacencia. Esto extiende más aún el análisis grafos permitiendo también realizar operaciones algebraicas entre grafos, como por ejemplo el cálculo de la matriz transpuesta o la posibilidad de definir un grafo como suma de otros sub-grafos [3, 4].

En especifico una de las operaciones de continuo estudio es la multiplicación de matrices. Obtener implementaciones eficaces para este cálculo en matrices de gran tamaño es un enorme desafío. ¿Es posible aprovechar las características de los valores almacenados en una matriz para mejorar los tiempos de cómputo de operaciones algebraicas entre matrices? ¿Se puede mejorar la representación de los valores de las matrices de tal forma que permita reducir el espacio de almacenamiento y el tiempo de procesamiento de operaciones algebraicas? El caso de grafos dispersos donde su representación matricial contiene muchos ceros se podría aprovechar la información esencial presente en este en búsqueda de patrones repetitivos entre filas y columnas con el fin de obtener representaciones alternativas que aprovechen de mejor manera el espacio y además permitan realizar operaciones en menor tiempo de cómputo.

1.2. Objetivos

El objetivo de esta memoria de título es proponer un estructura compacta que reduzca el espacio de un grafo con pesos, y que permita realizar operaciones de multiplicación con tiempo menor a la multiplicación eficiente para matrices dispersas.

Los objetivos específicos son:

- O1: Extender el descubrimiento de bicliques en la implementación de Hernández y Navarro [1] para grafos con pesos.
- O2: Definir una estructura para el cálculo de multiplicación de matrices generales utilizando bicliques.
- O3: Evaluación experimental de los enfoques propuestos.

Capítulo 2

Notación y definiciones

2.1. Grafos dirigidos

Sea $G = (V, E)$ un grafo dirigido con pesos, donde:

1. V es un conjunto finito y no vacío, cuyos elementos llamamos vértices.
2. E es conjunto de ternas (s, t, w) ordenadas la cuales llamaremos aristas y representan el nodo origen, destino y peso de la arista respectivamente, donde $s, t \in V$ y $w \in \mathbb{N}$.
3. $n = |V|$ es el número de nodos del grafo.
4. $m = |E|$ es el número de aristas del grafo.

En la Figura 2.1 se puede ver una representación gráfica de un grafo G de ejemplo, junto con su definición el la Figura 2.2.

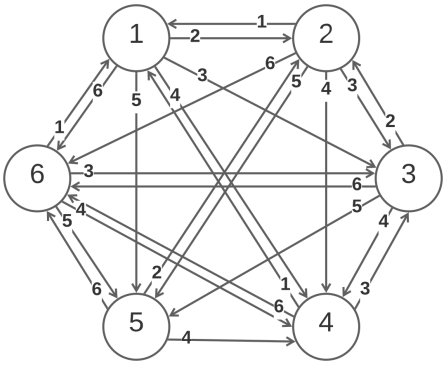

Figura 2.1: Representación gráfica del grafo G con sus respectivos nodos y aristas dirigidas con peso.

$$
\begin{aligned}
G &= (V, E) \\
V &= \{1, 2, 3, 4, 5, 6\} \\
E &= \{(1,2,2), (1,3,3), (1,4,4), (1,5,5), (1,6,6), \\
&\quad (2,1,1), (2,3,3), (2,4,4), (2,5,5), (2,6,6), \\
&\quad (3,2,2), (3,4,4), (3,5,5), (3,6,6), \\
&\quad (4,1,1), (4,3,3), (4,6,6), \\
&\quad (5,2,2), (5,4,4), (5,6,6), \\
&\quad (6,1,1), (6,3,3), (6,4,4), (6,5,5)\}
\end{aligned}
$$

Figura 2.2: Ejemplo de definición del Grafo G.

2.2. Matriz de adyacencia

Dado $G = (V, E)$, un grafo con $n = |V|$ nodos y $m = |E|$ aristas, es posible representar un grafo utilizando una una matriz de adyacencia A de tamaño $n \times n$, donde la fila i-ésima representa a los nodos adyacentes del nodo i. De este modo, cada elemento de la matriz $a_{i,j}$ se relaciona con un elemento $(s, t, w) \in E$, con $i = s$, $j = t$ y $a_{i,j} = w$. La Figura 2.3 presenta un ejemplo de matriz de adyacencia utilizando el grafo G del ejemplo anterior.

$$A = \begin{bmatrix} a_{1,1}=0 & a_{1,2}=2 & a_{1,3}=3 & a_{1,4}=4 & a_{1,5}=5 & a_{1,6}=6 \\ a_{2,1}=1 & a_{2,2}=0 & a_{2,3}=3 & a_{2,4}=4 & a_{2,5}=5 & a_{2,6}=6 \\ a_{3,1}=0 & a_{3,2}=2 & a_{3,3}=0 & a_{3,4}=4 & a_{3,5}=5 & a_{3,6}=6 \\ a_{4,1}=1 & a_{4,2}=0 & a_{4,3}=3 & a_{4,4}=0 & a_{4,5}=0 & a_{4,6}=6 \\ a_{5,1}=0 & a_{5,2}=2 & a_{5,3}=0 & a_{5,4}=4 & a_{5,5}=0 & a_{5,6}=6 \\ a_{6,1}=1 & a_{6,2}=0 & a_{6,3}=3 & a_{6,4}=4 & a_{6,5}=5 & a_{6,6}=0 \end{bmatrix}$$

Figura 2.3: Ejemplo de matriz de adyacencia A del grafo G con una matriz de 6×6, en 0 las casillas que no existen dentro del grafo, y con otro valor las aristas que existen en el grafo.

2.3. Densidad de un grafo

La densidad de un grafo dirigido se refiere a la proporción de aristas dirigidas presentes en el grafo en relación con el número máximo posible de aristas dirigidas entre sus nodos. Formalmente, la densidad de un grafo dirigido G con n nodos y m aristas dirigidas se calcula mediante la fórmula presentada en la Ecuación 2.1.

$$D(G) = \frac{m}{n(n-1)} \tag{2.1}$$

Donde:

- m es el número de aristas dirigidas en el grafo.
- n es el número de nodos en el grafo.

La fórmula $n(n-1)$ representa el máximo número de aristas dirigidas en un grafo dirigido con n nodos, teniendo en cuenta que cada par de nodos puede estar conectado por una arista dirigida en ambas direcciones. La densidad de un grafo dirigido varía en el rango de 0 a 1. Un valor de densidad igual a 0 indica que no hay aristas dirigidas en el grafo, mientras que un valor de densidad igual a 1 indica que todas las posibles aristas dirigidas entre los nodos están presentes en el grafo.

2.4. Producto cartesiano

El producto cartesiano es una operación que combina elementos de dos conjuntos para formar pares ordenados. Cada elemento del primer conjunto se combina con todos los elementos del segundo conjunto, generando un nuevo conjunto de pares ordenados que representa todas las posibles combinaciones entre los elementos de ambos conjuntos.

El producto cartesiano de dos conjuntos A y B, denotado como $A \times B$, es el conjunto de todos los pares ordenados (a, b), donde $a \in A$ y $b \in B$.

Dados los conjuntos,

$$A = \{1, 2, 3\}, \quad B = \{a, b, c\}$$

El producto cartesiano $A \times B$ es:

$$A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$$

2.5. Bicliques de un grafo

Se define el conjunto B^G de bicliques de G, como un conjunto de subgrafos bipartitos completos de G. Cada biclique $b_i \in B^G$ se define como $b_i(S_i, C_i)$, donde $S_i, C_i \in V$ son un subconjunto de vértices de V, y tiene como grafo asociado $B_i = (S_i \cup C_i, S_i \times C_i)$, con $S_i \cup C_i$ el conjunto de vértices de B_i y $S_i \times C_i$ el conjunto de aristas de B_i, que corresponde a todas las posibles aristas entre los el conjunto de vértices S_i y C_i que además están presentes en E. Además, cuando un biclique cumple que $|S_i| = |C_i|$ se dice que es un *biclique balanceado*

	S	C
b_1	{1,3,5}	{(2,2) (6,6)}
b_2	{1,2,6}	{(3,3) (4,4) (5,5)}

Cuadro 2.1: Representación de bicliques y sus componentes, a la izquierda el conjunto de nodos origen s, y al derecha el conjunto de nodos con sus pesos (t, w). Las aristas se obtienen realizando el producto cartesiano entre ambos conjuntos.

En el Cuadro 2.1 se ve un ejemplo de un conjunto B^G de bicliques del grafo G, cuyos subgrafos están representados en las Figuras 2.4 y 2.5 respectivamente.

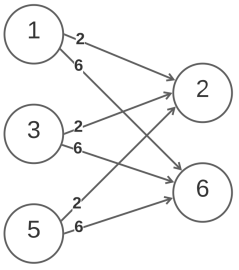

Figura 2.4: Representación gráfica de las aristas del biclique B_1.

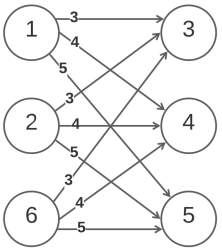

Figura 2.5: Representación gráfica de las aristas del biclique B_2.

Además se define la matriz de adyacencia B como la representación matricial de todos los bicliques o la matriz asociada a B^G, presentada en la Figura 2.6.

$$B = \begin{bmatrix} 0 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 3 & 4 & 5 & 0 \\ 0 & 2 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 6 \\ 0 & 0 & 3 & 4 & 5 & 0 \end{bmatrix}$$

Figura 2.6: Matriz de adyacencia B correspondiente a todas las aristas del conjunto de bicliques B^G.

2.6. Descomposición de grafos

Es posible definir la matriz A', como la diferencia entre la matriz de adyacencia A del grafo G y la matriz de bicliques B, cuyo grafo asociado G' lo llamaremos "resto del grafo"de G.

$$A' = A - B \tag{2.2}$$

$$A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 5 & 0 \\ 1 & 0 & 3 & 0 & 0 & 6 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 2.7: Matriz de adyacencia A', correspondiente a la diferencia entre la matriz del grafo completo y la matriz de bicliques.

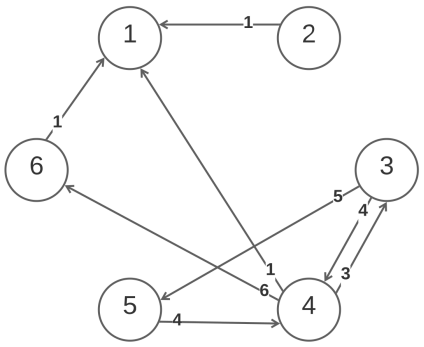

Figura 2.8: Representación gráfica de G', resto del grafo de G

En la Figura 2.7 se muestra la representación de A', dando como resultado una matriz dispersa o matriz esparsa, donde hay mayor cantidad de ceros y menor redundancia entre filas en comparación a la matriz A. La Figura 2.8 muestra una representación gráfica de G' donde se puede apreciar la reducción de aristas respecto a G.

2.7. Representaciones alternativas

Además de la representación de un grafo mediante una matriz de adyacencia, existen otras formas alternativas de representación que son ampliamente utilizadas en el ámbito de las ciencias de la computación. Estas representaciones son conocidas por su capacidad para reducir significativamente el espacio en disco, lo cual es fundamental en aplicaciones donde se manejan grandes volúmenes de datos. A continuación, se presentan dos de estas representaciones, las cuales son conocidas como *Compressed Sparse Row* (CSR) y *Compressed Sparse Column* (CSC). Estas técnicas de compresión son especialmente útiles cuando se trabaja con grafos dispersos, donde la mayoría de las entradas en la matriz de adyacencia son cero.

2.7.1. Compress Sparse Row

Compress Sparse Row (CSR) es un formato de representación de matrices dispersas que almacena únicamente los valores no nulos y sus índices correspondientes por

fila. Esto permite reducir significativamente la cantidad de memoria necesaria para almacenar una matriz dispersa, ya que no se guardan los ceros.

En la representación CSR, se utilizan tres arreglos para almacenar la matriz dispersa, de manera auxiliar se utiliza un vector `row_id` para almacenar los índices de las filas que están representadas.

1. `values:` contiene los valores no nulos de la matriz (sin los ceros).
2. `col_ind:` guarda los índices de columna de estos valores.
3. `row_ptr:` almacena los índices de inicio de cada fila en los dos arreglos anteriores.

2.7.2. Compress Sparse Column

Compress Sparse Column (CSC) es otro formato de representación de matrices dispersas que almacena los valores no nulos y sus índices por columna en lugar de por fila. Al igual que en CSR, de manera auxiliar se puede utilizar un vector `col_id` para almacenar los índices de las columnas que están representadas.

Al igual que en CSR, CSC utiliza tres arreglos para representar la matriz dispersa:

1. `values:` contiene los valores no nulos de la matriz (sin los ceros).
2. `row_ind:` guarda los índices de fila de estos valores.
3. `col_ptr:` almacena los índices de inicio de cada columna en los dos arreglos anteriores.

La matriz A' se muestra en su representación *Compress Sparse Row* y *Compress Sparse Column* en la Figura 2.9a y 2.9b respectivamente. [1]

[1] En el vector `row_ptr` y `col_ptr` se dejaron espacios en blanco para que el valor coindcida con la posición a la que hace referencia en el vector `col_ind` y `row_ind` respectivamente.

values:	1	4	5	1	3	6	4	1
col_ind:	1	4	5	1	3	6	4	1
row_ptr:	0	1		3			6	7	8
row_id:	2	3		4			5	6

(a) *Compress Sparse Row* $CSR(A')$

values:	1	1	1	3	4	4	5	6
row_ind:	2	4	6	4	3	5	3	4
col_ptr:	0			3	4		6	7	8
col_id:	1			3	4		5	6

(b) *Compress Sparse Column* $CSC(A')$

Figura 2.9: Representaciones alternativas de A'

Capítulo 3

Estado del arte

3.1. Compresión de grafos

El principal problema del uso de la representación de grafos mediante matrices de adyacencia es el gran espacio que utiliza, dado que representa con un cero la inexistencia de una arista. Luego, para grafos muy grandes y dispersos la representación utiliza mucho espacio. Desde hace años se trabajan sobre alternativas que permitan ahorrar espacio en su representación, y además permitan mantener o mejorar el tiempo de cómputo de operaciones. Pinar y Heath [3] utilizaron una forma comprimida de representar la matriz utilizando como estructura de datos *Compressed Row Storage* (CRS) la cual permite ahorrar espacio aprovechando las características del grafo disperso, dado que no necesita representar los ceros de la matriz. Sin embargo, esta representación esta condicionada a la naturaleza del grafo por lo que también proponen un método de re-ordenamiento de las filas de la matriz utilizando heurísticas con el fin de aumentar la compresión de el mismo. Además el hacer esto les permite reducir el tiempo de cómputo de las operaciones multiplicación matriz-vector.

Otros enfoques más recientes aprovechan las propiedades de los grafos dispersos y

lo combinan con la búsqueda de patrones repetitivos dentro de estos. El trabajo de Glaria et al. [5] es un ejemplo de esto, que busca aprovechar la redundancia usando clustering de particionamiento de los cliques maximales que cubren el grafo. En el trabajo desarrollado, los autores proponen un método que permite encontrar una partición de cliques maximales. Mediante este método pueden realizar compresión sobre el grafo utilizando una estructura de datos compacta. La idea consiste en encontrar una descomposición del grafo en términos de particiones de cliques.

Por otro lado, en el trabajo de Hernández y Navarro [1], se presenta la idea de compresión mediante la búsqueda de "comunidades densas" en grafos web. La idea de "comunidades densas" dentro del contexto de un grafo hacen referencia a un biclique (sub-grafo bipartito completo). En términos de la representación utilizando matrices de adyacencia, en un biclique todos los nodos tendrían un conjunto nodos adyacentes en común. Por lo tanto la compresión se centraría en eliminar esta redundancia entre nodos. Para la búsqueda de estos bicliques se utiliza la técnica de Min-Hash [6] combinado con Prefix-Tree, para hacer una búsqueda bicliques de manera rápida y sin tener tanto impacto en memoria RAM. Esta representación permite comprimir grafos dispersos que contienen bicliques. La estructura permite representar en forma implícita las aristas representadas en los bicliques. El enfoque usa una estructura compacta para los bicliques y el resto del grafo lo comprime utilizando la estructura `k2-tree`. Como resultado, obtienen una mayor compresión que otros métodos.

Otro problema relacionado con bicliques corresponde a la búsqueda del biclique máximo de un grafo, es decir, el biclique de mayor tamaño presente en un grafo, con el fin de aprovechar la información que representa el biclique de mayor tamaño de un grafo. En el trabajo desarrollado por Bingqing Lyu et al. [7] proponen un framework para abordar este problema, ya que en este caso la búsqueda de este biclique maximo (biclique mas grande) no apunta a obtener una representación comprimida, si no que buscan aprovechar la información que representa el biclique maximo en un grafo. Sin embargo, este es un problema mucho más complejo que el anterior, y el framework que proponen los autores no es escalable a grafos con grandes cantidades de nodos y aristas.

Otro problema que va de la mano con este último es la búsqueda del biclique máximo balanceado. En el trabajo realizado por Chen et al. [8], los autores proponen un algoritmo para abordar este problema en tiempo polinomial en grafos densos y en grafos dispersos de gran tamaño.

Si bien los trabajos mencionados se enfocan en la búsqueda de bicliques, los métodos utilizados no son equivalentes. En el trabajo desarrollado por Hernández y Navarro [1] se apunta a encontrar bicliques de distintos tamaños en un tiempo razonable, aunque no busca garantizar encontrar el mayor biclique ni el mayor biclique balanceado que es donde apuntan los trabajos realizados por Bingqing Lyu et al. [7] y Chen et al. [8] respectivamente.

3.2. Compresión para realizar operaciones

Además de buscar comprimir grafos para ahorrar espacio, muchas investigaciones pretenden aprovechar esta compresión para mejorar tiempos de cómputo en el álgebra de matrices de gran tamaño.

Para el caso de la operación de multiplicación de matrices, o multiplicación vector-matriz, actualmente existen diversos algoritmos de operaciones de matrices. El algoritmo estándar de multiplicación matricial tiene complejidad $O(n^3)$. Por otro lado podríamos utilizar algoritmos más eficientes como el desarrollado por Strassen [9] para reducir la complejidad del cómputo. Una de las ventajas de este último es que se trata de un algoritmo de «Dividir para conquistar» [10], por lo tanto se han propuesto múltiples enfoques utilizando computación paralela para mejorar los tiempos de cómputo [11]. Estos trabajos apuntan a resolver el problema de multiplicación de matrices generales, esto es, incluyen matrices densas y dispersas.

Mohades [12] demostró que es posible aprovechar las propiedades de grafos dispersos para reducir la complejidad de algunas operaciones, apuntando a la reducción de operaciones repetitivas. De igual manera en el trabajo de Buluc y Gilbert [13] se aprovechan las propiedades dispersas para proponer un algoritmo paralelo en la computo de la multiplicación matriz-matriz.

Otros autores además de aprovechar las propiedades dispersas, buscan aprovechar representaciones comprimidas, usando CSR, para mejorar los tiempos de operaciones matriciales [14]. En el trabajo realizado por Francisco et al. [15] se busca computar el producto de la multiplicación matriz por vector en tiempo proporcional al grado de compresión de su representación comprimida, para ello se utilizan dos representaciones para la representación comprimida. Por un lado se utiliza el framework *WebGraph* [16], la idea que se plantea es computar un vector y, producto de la multiplicación de la matriz A y el vector x^T utilizando dos enfoques:

1. De manera habitual, multiplicando $A \times x^T = y$
2. Descomponer A en A', donde A' es el resultado de restar un vector v a cada fila de A. De esta manera, para obtener y se debe realizar la siguiente operación $y = A' \times x^T + z^T$, donde $z = v \times x^T$.

Los resultados obtenidos por los autores muestran que el segundo enfoque proporciona mejores tiempos de computo en todos los datasets que utilizaron.

Por otro lado, utilizaron el enfoque de bicliques visto por Hernández y Navarro [1] la representación comprimida, obteniendo resultados favorables cuando se trabajan con gran cantidad de bicliques.

El estudio realizado por Ferragina et al. [17] presenta un enfoque diferente para la compresión de grafos. En lugar de enfocarse en la naturaleza densa o dispersa del grafo, se centra en la compresión mediante gramática. En primer lugar, se aplica una compresión utilizando una variante adaptada de *Compress Sparse Row*, denominada Compressed Sparse Row/Value, y luego se implementa una compresión basada en gramática similar a algoritmos como RePair. La representación resultante permite realizar operaciones matrix-vector en tiempo proporcional al número de reglas de reemplazo generadas por la compresión. Además, para mejorar la eficiencia de estos algoritmos, es recomendable considerar el reordenamiento de las columnas. Esto puede potenciar la compresión de gramáticas y facilitar la identificación de columnas similares.

Tanto el trabajo de Francisco et al. [15], como el de Ferragina et al. [17] aprovechan

representaciones comprimidas de grafos que no requieren hacer descompresión para trabajar sobre ellos, lo cual favorece el rendimiento de ambos.

Los autores en los trabajos anteriormente mencionados buscan realizar mejoras en las operaciones entre matrices, comprimir para ahorrar espacio, o bien para mejorar el rendimiento de algunos algoritmos como PageRank. [15]. En cambio, el trabajo de Arroyuelo et al. [18] se centra en un problema más especifico que aprovecha operaciones de matrices para su resolución. Enfocándose en bases de datos orientada a grafos (graph database), apuntan a encontrar Regular path queries, que son caminos que pueden ser expresados por una expresión regular. Para ello trabajan sobre un grafo dirigido con etiquetas en sus aristas G, el cual se compone de tríos (s, p, o) que codifican que el nodo s esta conectado al nodo o por el «camino» con etiqueta p. Como base de la representación utilizaron una versión de `k2-tree` enfocada en grafos web [19].

Capítulo 4

Identificación y extracción de bicliques

4.1. Definición del problema

La representación de grafos mediante matrices de adyacencia es útil y ampliamente utilizada. Sin embargo cuando el número de nodos de un grafo aumenta, también lo hace la matriz de manera cuadrática, por lo que trabajar con un grafo grande, digamos billones de nodos, esta no es la mejor representación para ello.

En matrices de adyacencia se representan las $n \times n$ aristas posibles, aún cuando no existen en el grafo. Entonces, cuando la cantidad de aristas es mucho menor a $n \times n$ se desperdicia una gran cantidad de espacio al representarlas como matriz de adyacencia.

Una manera alternativa de representar el grafo es utilizar listas de adyacencia. Tomando el grafo G, de la Figura 4.1, se obtiene la representación con listas de adyacencias del Cuadro 4.1.

Con esta representación se consigue ahorrar espacio para trabajar con grafos de gran tamaño al representar únicamente las aristas que existen en el grafo y omitiendo las que no existen. Sin embargo, aún es posible reducir el espacio aprovechando la

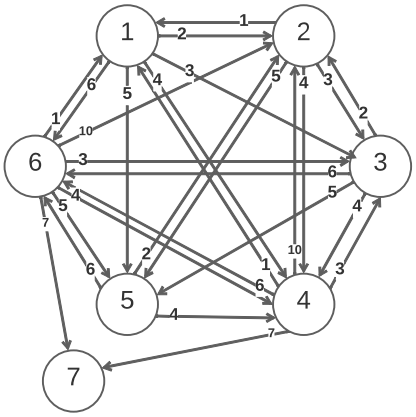

Figura 4.1: Grafo G utilizado de ejemplo en el proceso de extracción de bicliques

$$
\begin{aligned}
G &= (V, E) \\
V &= \{1, 2, 3, 4, 5, 6, 7\} \\
E &= \{(1,2,2), (1,3,3), (1,4,4), (1,5,5), (1,6,6), \\
&\quad (2,1,1), (2,3,3), (2,4,4), (2,5,5), \\
&\quad (3,2,2), (3,4,4), (3,5,5), (3,6,6), \\
&\quad (4,1,1), (4,2,10), (4,3,3), (4,6,6), (4,7,7) \\
&\quad (5,2,2), (5,4,4), (5,6,6), \\
&\quad (6,1,1), (6,2,10), (6,3,3), (6,4,4), (6,5,5), (6,7,7)\}
\end{aligned}
$$

Figura 4.2: Representación formal del Grafo G de ejemplo

similitud que hay entre las listas de adyacencia de cada grafo. Es decir extrayendo los bicliques que hay presente en el grafo.

Nodo	Adyacentes
1:	(2,2) (3,3) (4,4) (5,5) (6,6)
2:	(1,1) (3,3) (4,4) (5,5)
3:	(2,2) (4,4) (5,5) (6,6)
4:	(1,1) (2,10) (3,3), (6,6) (7,7)
5:	(2,2) (4,4) (6,6)
6:	(1,1) (2,10) (3,3) (4,4), (5,5) (7,7)

Cuadro 4.1: Representación en lista de adyacencia de Grafo G

4.2. Extracción de bicliques

A continuación, se describe una generalización del proceso de extracción de bicliques a grafos con pesos, basada en el trabajo previo realizado por Hernández y Navarro [1], que ha sido reimplementado utilizando `C++`. Esta implementación admite además grafos de mayor tamaño, con capacidad para hasta 2^{64} nodos.

4.2.1. Clustering

El primer paso en la extracción de bicliques implica identificar los bicliques potenciales. Dado que se pretende manejar grafos con un gran volumen de nodos y aristas, aplicar funciones hash a las listas de adyacencia de cada arista no resulta la estrategia más eficiente debido al espacio que requiere. En su lugar, se busca detectar patrones repetitivos mediante funciones hash específicas diseñadas para esta tarea, las cuales son mucho más amigables en términos de consumo de memoria.

MinHash

El algoritmo MinHash [6] es una técnica utilizada en la minería de datos y la recuperación de información para estimar la similitud entre conjuntos de elementos [20]. Su enfoque se basa en el uso de funciones hash sensibilizadas por la localidad para crear signatures compactas que representan conjuntos grandes de manera eficiente.

Estas signatures permiten comparar rápidamente la similitud entre dos conjuntos, incluso cuando estos conjuntos son muy grandes y contienen elementos repetidos o no ordenados.

En términos simples, MinHash es una herramienta poderosa para medir la similitud entre conjuntos de datos de manera aproximada pero eficiente, lo que lo hace especialmente útil en aplicaciones donde el cálculo de la similitud exacta entre conjuntos grandes sería computacionalmente costoso.

En este caso no se busca en sí medir la similitud, si no que aprovechar el algoritmo para encontrar elementos comunes en las listas de adyacencias. Para ello el algoritmo se encarga de representar las listas de adyacencia de cada nodo con P *signatures*. Generando de esta manera una matriz de $n = |V|$ filas y P columnas. Para ello, se define el concepto de *k-shingle* [21], que consiste en tomar k elementos de una lista de adyacencia y aplicarles una función hash, cuyo resultado denominamos *ShingleID*.

Luego, se definen las P funciones hash de la siguiente manera:

1. Se elige un número primo *prime* lo suficientemente grande para evitar colisiones.

2. Por cada $signature_i$ $i \in 1, 2, ..., P$ se eligen dos números aleatorios A_i, B_i.

$$\begin{matrix} A_1 & A_2 & ... & A_P \\ B_1 & B_2 & ... & B_P \end{matrix}$$

3. Por cada *k-shingle*, con un *ShingleID* respectivo, computar

$$H_i = (A_i \cdot ShingleID + B_i) \bmod prime \tag{4.1}$$

4. De esta manera se obtienen P resultados por cada *k-shingle*, pero se quiere obtener solo P por cada lista de adyacencia. Por lo tanto se definen P valores s por cada lista de adyacencia, para los cuales se guarda el menor valor obtenido con la i-ésima función aplicada a algún *shingleID* de la lista de adyacencia de ese nodo.

El resultado de este proceso es la Matriz de Signatures de tamaño n filas y P columnas, como se ilustra en el Cuadro 4.2. Este paso requiere tiempo $O(P|E|)$.

Nodo	Signatures
1:	s_1, s_2, ... , s_P
2:	s_1, s_2, ... , s_P
.	.
.	.
.	.
n:	s_1, s_2, ... , s_P

Cuadro 4.2: Matriz de signatures, para n nodos y con P signatures por cada lista de adyacencia.

Volviendo al grafo G de ejemplo de la Figura 4.1, la matriz de signatures del G se presenta en el Cuadro 4.3, utilizando $P = 2$.

Nodo	Signatures
1:	A, F
2:	A, B
3:	C, F
4:	D, G
5:	E, F
6:	A, B

Cuadro 4.3: Matriz de signatures para el Grafo G, con $n = 6$ y $P = 2$.

Construcción de clusters

Una vez computada la matriz de signatures del grafo, se debe formar los clusters. Para ello, se recorren las P columnas de la Matriz, para el caso del grafo G del ejemplo se tienen los clusters presentados en la Figura 4.3 y la Figura 4.4.

Nodo	Signatures	Adyacentes
1:	A, F	(2,2) (3,3) (4,4) (5,5) (6,6)
2:	A, B	(1,1) (3,3) (4,4) (5,5)
6:	A, B	(1,1) (2,10) (3,3) (4,4), (5,5) (7,7)

Figura 4.3: Cluster 1 obtenido por la similitud de su primera columna de signatures

Nodo	Signatures	Adyacentes
3:	C, F	(2,2) (4,4) (5,5) (6,6)
4:	D, G	(1,1) (2,10) (3,3), (6,6) (7,7)
5:	E, F	(2,2) (4,4) (6,6)

Figura 4.4: Cluster 2 obtenido por los nodos restantes de la matriz de signatures

El Cluster 1 de la Figura 4.3 es construido por la similitud en la primera columna de Signatures, mientras que el Cluster 2 de la Figura 4.4 si bien no mantiene la similitud en la primera columna, es un cluster formado por los nodos descartados, ya que podría existir similitud en una columna posterior. Este paso requiere tiempo $O(P|V|log(|V|))$.

4.2.2. Reordenamiento de aristas

En la siguiente etapa, por cada cluster se computa la frecuencia de las aristas adyacentes, para reordenar de arista más frecuente a arista menos frecuentes las listas de adyancencia del cluster. Para ello se recorren las listas de adyacencia y se construye

un histograma con la aparición de las aristas. El objetivo del reordenamiento es poder capturar bicliques mas grandes mediante el árbol de prefijos que se describe en la siguiente sección.

Histograma Aristas	
(3,3):	3
(4,4):	3
(5,5):	3
(1,1):	2
(6,6):	2
(2,2):	1
(2,10):	1
(7,7):	1

(a) Tabla de frecuencias de los nodos adyacentes del Cluster 1

Nodo	Adyacentes
1:	(3,3), (4,4), (5,5), (6,6)
2:	(3,3), (4,4), (5,5), (1,1)
6:	(3,3), (4,4), (5,5), (1,1)

(b) Listas de adyacencias despues del reordenamiento por frecuencias

Figura 4.5: Proceso de reordenamiento del Cluster 1

La Tabla 4.5a corresponde al histograma del Cluster 1 4.3, luego en la Tabla 4.5b se presenta las listas de adyacencia resultantes luego de reordenamiento. Este paso requiere tiempo $O(|E|log(|E|))$.

4.2.3. Árbol de prefijos

Por cada cluster obtenido, después de reordenar las aristas, se crea un árbol de prefijos a partir de las listas de adyacencia. Para ello, cada nodo del árbol tiene una etiqueta con el *id* de una arista (vértice vecino) en la lista de adyacencia y un vector que almacena los vértices que lo apuntan. Un biclique en el árbol está formado por una secuencia de *ids* de los nodos en una rama del árbol y que constituye el set C del biclique. Por otro lado, el set S del biclique lo conforma el vector de

vértices almacenadas en el nodo en el árbol. Luego, el algoritmo recorre el árbol para identificar y extraer los bicliques con mayor representación de aristas dada por $\frac{|S|\times|C|}{|S|+|C|}$ (fracción de ganancia).

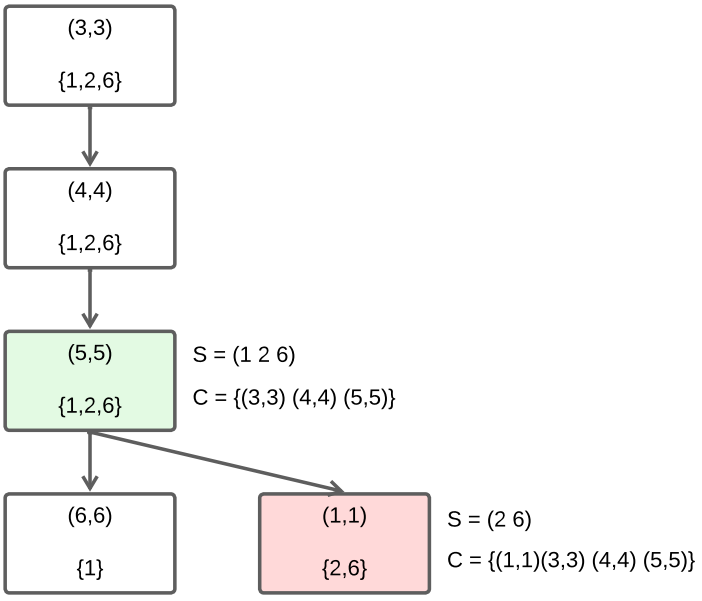

Figura 4.6: Árbol de prefijos a partir del Cluster 1, el elemento superior de cada nodo representa una arista, mientras que los elementos abajo representan los nodos a los cuales pertenece esa arista y todas las aristas superiores hasta la raíz. Los nodos en color representan candidatos a posibles bicliques

La Figura 4.6 representa el árbol de prefijos del Cluster 1, el nodo hoja en verde y el nodo hoja en rojo representan posibles bicliques.

El nodo hoja en verde es el mejor biclique que se puede extraer de ese árbol, ya que mediante se están representando 9 aristas, con un fracción de ganancia $\frac{|S|\times|C|}{|S|+|C|} = 1{,}5$, mientras que con el nodo hoja rojo solo se representan 8 aristas y presenta una fracción de ganancia $\frac{|S|\times|C|}{|S|+|C|} = 1{,}35$. Es decir, se busca el nodo hoja que maximice $(|S| \times |C|)$ y también la fracción de ganancia. El Biclique b_1 extraído del árbol de prefijos se representa en el Cuadro 4.4.

	S	C
b_1	{1,2,6}	{(3,3) (4,4), (5,5)}

Cuadro 4.4: Mejor biclique extraído del árbol de prefijos del Cluster 1, correspondiente al nodo en color verde en base a su fracción de ganancia.

El proceso de descubrimiento de bicliques es iterativo. En cada iteración se actualizan las listas de adyacencias para encontrar nuevos bicliques, hasta el punto en el que no se encuentran bicliques, o bien, los bicliques encontrados no cumplen con un umbral dado por la fracción de ganancia.

Para el caso del grafo G, los bicliques extraídos al final del proceso completo se presentan en el Cuadro 4.5, cuya matriz B está representada en la Figura 4.7.

	S	C
b_1	{1,2,6}	{(3,3) (4,4), (5,5)}
b_2	{1,3,5}	{(2,2) (6,6)}
b_3	{4,6}	{(1,2) (2,10), (7,7)}

Cuadro 4.5: Conjunto de todos los bicliques que fueron extraídos del grafo G, repitiendo el proceso descrito.

La resto del grafo G, denotado por la matriz de Adyacencia A' se muestra en la Figura 4.8, donde se puede ver que se trata de una matriz dispersa donde hay menos información redundante.

$$B = \begin{bmatrix} 0 & 2 & 3 & 4 & 5 & 6 & 0 \\ 0 & 0 & 3 & 4 & 5 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 6 & 0 \\ 1 & 10 & 0 & 0 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 & 0 & 6 & 0 \\ 1 & 10 & 3 & 4 & 5 & 0 & 7 \end{bmatrix}$$

Figura 4.7: Matriz de adyacencia B de todas las aristas de los bicliques extraídos del grafo G.

$$A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 4.8: Matriz de adyacencia A' del resto del grafo G despues de la extracción de bicliques.

Para el desarrollo se cuentan con los siguientes parámetros para ajustarse a diversos grafos, que permiten encontrar bicliques de menor o menor tamaño según se requiera:

1. `shingle_size`: Valor k para de los *k-shingles*.
2. `num_signatures`: Número de funciónes hash para computar MinHash.
3. `min_ady_nodes`: Mínimo de nodos adyacentes en un nodo para decidir si evaluarlo o no en MinHash.
4. `min_cluster_size`: Número de nodos mínimo que deben constituir un cluster para computar su árbol de prefijos.
5. `biclique_size`: Tamaño mínimo ($|S| \times |C|$) que debe tener el biclique al momento de ser extraído del árbol de prefijos.
6. `threashold`: Cuando la cantidad de bicliques obtenidas en la iteración es bajo este umbral, se disminuye `biclique_size` para encontrar bicliques más pequeños.
7. `bs_decrease`: Valor de decremento en el `biclique_size` cuando el `threashold` no es alcanzado.
8. `iterations`: Número máximo de iteraciones.

Capítulo 5

Multiplicación de matrices usando bicliques

5.1. Definición del problema

El siguiente problema a abordar es el de la multiplicación entre matrices y como puede ser mejorado aprovechando los bicliques extraídos en el problema anterior. Según lo presentado en el capitulo anterior, en términos matriciales la extracción de bicliques de un grafo puede expresarse en una descomposición de matrices. Siendo A la matriz del grafo original, A' la matriz del grafo sin bicliques y B la matriz del conjunto de bicliques extraídos.

$$A = A' + B \tag{5.1}$$

La Ecuación 5.1 representa la extracción de bicliques como una suma de matrices. De ese modo, se puede pensar la potencia de la matriz A de la siguiente manera, Expresando la potencia de A^2 como la potencia de un binomio según se muestra en la Ecuación 5.2.

$$A^2 = (A' + B)^2 = (A')^2 + (A' \times B) + (B \times A') + B^2 \tag{5.2}$$

Para el caso de una multiplicación general, con A_1 y A_2 como matrices generales, cuya descomposición con bicliques viene dada por la Ecuación 5.3 y 5.4 respectivamente

$$A_1 = A'_1 + B_1 \tag{5.3}$$

$$A_2 = A'_2 + B_2 \tag{5.4}$$

En este caso, la multiplicación matricial $A_1 \times A_2$ es posible expresarla en términos de una multiplicación entre dos binomios como se expresa en la Ecuación 5.6

$$(A_1 \times A_2) = (A'_1 + B_1) \times (A'_2 + B_2) \tag{5.5}$$

$$(A_1 \times A_2) = (A'_1 \times A'_2) + (A'_1 \times B_2) + (B_1 \times A'_2) + (B_1 \times B_2) \tag{5.6}$$

De esta manera, para obtener la multiplicación utilizando bicliques se debe desarrollar una operatoria para $(Matriz \times Matriz)$, $(Matriz \times Biclique)$, $(Biclique \times Matriz)$ y $(Biclique \times Biclique)$ y luego sumar las matrices resultantes de cada una de las operaciones.

5.2. Operaciones

Por simplicidad, se considera el caso de la potencia de una matriz para su explicación, siguiendo con el ejemplo del grafo G del capítulo anterior. La Figura 4.1 muestra dicho grafo, cuya matriz de adyacencia se presenta en la Figura 5.1.

$$A = \begin{bmatrix} 0 & 2 & 3 & 4 & 5 & 6 & 0 \\ 1 & 0 & 3 & 4 & 5 & 0 & 0 \\ 0 & 2 & 0 & 4 & 5 & 6 & 0 \\ 1 & 10 & 3 & 0 & 0 & 6 & 7 \\ 0 & 2 & 0 & 4 & 0 & 6 & 0 \\ 1 & 10 & 3 & 4 & 5 & 0 & 7 \end{bmatrix}$$

Figura 5.1: Matriz de adyacencia A del Grafo G

.

5.2.1. Multiplicación de matrices

La multiplicación $Matriz \times Matriz$ utilizada se basa en el algoritmo desarrollado por Schoor [14], el cual para una matriz A de tamaño $M \times N$ y una matriz B de tamaño $N \times K$, tiene una complejidad de $O(D(A) \cdot D(B) \cdot M \cdot N \cdot K)$, donde $D(A)$ y $D(B)$ es la densidad de las matrices a operar. Entre menor sea la densidad, o mayor cantidad de ceros mejor comportamiento tiene.

Para realizar esta operatoria, se debe representar la matriz dispersa A' del resto del grafo G utilizando *Compress Sparse Column* (CSC) y *Compress Sparse Row* (CSR).

$$A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 5.2: Matriz de adyacencia A' del resto del grafo G

$$A' \times A' = CSC(A') \times CSR(A') \tag{5.7}$$

values:	1	3	4	4	5	6	
row_ind:	2	4	3	5	3	4	
col_ptr:	0	1	2		4	5	6
col_id:	1	3	4		5	6	

(a) $CSC(A')$

values:	1	4	5	3	6	4	
col_ind:	1	4	5	3	6	4	
row_ptr:	0	1		3		5	6
row_id:	2	3		4		5	

(b) $CSR(A')$

Figura 5.3: Representaciones alternativas de la matriz A'

Con esta representación ya no se realizan operaciones de multiplicación fila por columna sobre la matriz de adyacencia A', sino que se debe operar $CSC(A') \times CSR(A')$ como se muestra en la Ecuación 5.7. El procedimiento se muestra en el siguiente ejemplo para la potencia de la matriz A'.

1. Intersectar el vector `col_id` con el vector `row_id`

values:	1	3	4	4	5	6	
row_ind:	2	4	3	5	3	4	
col_ptr:	0	1	2		4	5	6
col_id:	1	**3**	**4**		**5**	6	

values:	1	4	5	3	6	4	
col_ind:	1	4	5	3	6	4	
row_ptr:	0	1		3		5	6
row_id:	2	**3**		**4**		**5**	

Figura 5.4: Elementos intersectados del vector `col_id` y `row_id` resaltados en color.

2. Por cada intersección obtener el contenido de la columna y fila respectivamente

$$\begin{aligned} \mathbf{3} \times \mathbf{3}: &\quad (4,3) \times (4,4)\ (5,5) \\ \mathbf{4} \times \mathbf{4}: &\quad (3,4)\ (5,4) \times (3,3)\ (6,6) \\ \mathbf{5} \times \mathbf{5}: &\quad (3,5) \times (4,4) \end{aligned}$$

3. Realizar producto cartesiano entre las intersecciones, almacenando en una *priority queue* los resultados. Para los pares obtenidos de $CSC(A')$, su primer elemento corresponde al *id* de la fila de la matriz resultante, mientras que los pares obtenidos de $CSR(A')$, su primer elemento corresponde al *id* de columna de la matriz resultante. Para obtener el valor resultante, se debe obtener el producto del segundo elemento de ambos pares.

Priority queue	
(3,3,12)	(3,4) × (3,3)
(3,4,20)	(3,5) × (4,4)
(3,6,24)	(3,4) × (6,6)
(4,4,12)	(4,3) × (4,4)
(4,5,15)	(4,3) × (5,5)
(5,3,12)	(5,4) × (3,3)
(5,6,24)	(5,4) × (6,6)

Figura 5.5: Priority queue con los elementos obtenidos de las intersecciones, con las aristas a la izquierda ya ordenadas.

La Figura 5.5 muestra la *priority queue* ya construida, a la izquierda se muestra una terna donde cada valor representa $fila$, *columna* y *valor* respectivamente. A la derecha se muestra los pares que de los cuales se hizo el producto para obtener aquel resultado.

4. Finalmente, para obtener la matriz resultante se debe vaciar la *priority queue* la cual irá entregando los elementos de la matriz de manera ordenada. El orden viene dado por el primer valor de la terna, y en caso de empate, se considera el segundo valor de manera ascendente. En caso de que en la *priority queue* exista una terna cuyo valor de *fila* y*columna* se repita, se debe sumar los valores de ambas ternas como se muestra en el Cuadro 5.1. De este modo, la *priority queue* entregara las filas ordenadas de menor a mayor, y con sus respectivas

columnas también ordenadas.

$$\begin{matrix}(a,b,c)\\(a,b,c)\end{matrix} \quad \Longrightarrow \quad (a,b,2c)$$

Cuadro 5.1: Ejemplo de elementos repetidos en la *priority queue*, se suman los pesos de ambos elementos conservando un único trío por arista.

`values:`	12	20	24	12	15	12	24	
`col_ind:`	3	4	6	4	5	3	6	
`row_ptr:`	0			3		5		7
`row_id:`	3			4		5		

Figura 5.6: Representación en *Compress Sparse Row* de la matriz resultante, $CSR(A' \times A')$

$$A' \times A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 20 & 0 & 24 & 0 \\ 0 & 0 & 0 & 12 & 15 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 24 & 0 \end{bmatrix}$$

Figura 5.7: Matriz de adyacencia resultante $A' \times A'$, con las aristas extraídas de la *priority queue*.

Las Figuras 5.6 y 5.7 muestran $A' \times A'$ en representación *Compress Sparse Row* y matriz de adyacencia respectivamente.

El caso de la multiplicación general viene dado por la Ecuación 5.8. La multiplicación de matrices no es conmutativa por lo que el primer factor debe estar en su representación CSC mientras que el segundo factor debe estar en su representación CSR.

$$A_1 \times A_2 = CSC(A_1) \times CSR(A_2) \tag{5.8}$$

Estructuras de datos

A continuación se presentan las estructuras de datos utilizadas para la implementación de la operación descrita.

```
Struct CSC {
    ▪ vector<int>values
    ▪ vector<int>row_ind
    ▪ vector<int>col_ptr
    ▪ vector<int>col_id
}
```

Figura 5.8: Estructura Compress Sparse Column para matrices.

```
Struct CSR {
    ▪ vector<int>values
    ▪ vector<int>col_ind
    ▪ vector<int>row_ptr
    ▪ vector<int>row_id
}
```

Figura 5.9: Estructura Compress Sparse Row para matrices.

```
Struct Intersection {
    ▪ int start_col
    ▪ int end_col
    ▪ int start_row
    ▪ int end_row
    ▪ int value_col
    ▪ int value_row
}
```

Figura 5.10: Estructura Intersección, `start_col` y `end_col` son los intervalos de la columna intersectada correspondiente en el vector `row_ind` de CSC, análogamente `start_row` y `end_row` son los intervalos para la fila en el vector `col_ind` de CSR, por último `value_col` y `value_row` son los valores al principio de la columna y la fila respectivamente, para ser comparadas en al *priority queue*.

Algoritmos

A continuación se presentan los *pseudocódigos* de los pasos descritos anteriormente. El Algoritmo 1 corresponde a los pasos 1 y 2, mientras que el Algoritmo 2 corresponde a los pasos 3 y 4, dando como resultado la matriz en formato CSR.

Algorithm 1: Get Intersections

Entrada: `csc_a, csr_b`: Matriz A y B en representación CSC y CSR

Salida : `Hr`: Heap de Intersections

```
1  Hr ← empty Heap
2  i ← 0
3  j ← 0
4  while i < csc_a.col_id.size() and j < csr_b.row_id.size() do
5      if csc_a.col_id[i] == csr_b.row_id[j] then
6          inter ← Struct Intersection
7          inter.start_col ← csc_a.col_ptr[i]
8          inter.end_col ← csc_a.col_ptr[i+1]
9          inter.start_row ← csr_b.row_ptr[j]
10         inter.end_row ← csr_b.row_ptr[j+1]
11         inter.value_col ← csc_a.row_ind[inter.start_col]
12         inter.value_row ← csr_b.col_ind[inter.start_row]
13         Hr.push(inter)
14         i ← i + 1
15         j ← j + 1
16     end
17     else if csc_a.col_id[i] > csr_b.row_id[j] then
18         j ← j + 1
19     end
20     else
21         i ← i + 1
22     end
23 end
24 return Hr
```

Algorithm 2: Multiplicación Matriz × Matriz

Entrada: csc_a, csr_b: Matriz A y B en representación CSC y CSR

Salida : csr_res: Matriz resultante en CSR

```
1  Hr ← getIntersections(csc_a, csr_b)                          // Heap por value_row
2  Hc ← empty Heap                                              // Heap por value_col
3  csr_res ← empty CSR Matrix while not Hr.empty() do
4      elem ← Hr.top()
5      Hr.pop()
6      Hc.push(elem)
7      if Hr.empty() or Hr.top() ≠ elem.value_col then
8          sum ← 0
9          while not Hc.empty() do
10             temp ← Hc.top()
11             Hc.pop()
12             sum ← sum + csc_a.values[temp.start_col]
13                               × csr_b.values[temp.start_row]
14             if Hc.empty() or temp.value_row ≠ Hc.top().value_row then
15                 csr_res.insert(csc_a.row_ind[temp.start_col],
16                                csr_b.col_ind[inter.start_row], sum)
17                 sum ← 0
18             end
19             if temp.start_row < temp.end_row - 1 then
20                 temp.start_row ← temp.start_row + 1
21                 temp.value_row ← csr_b.col_ind[temp.start_row]
22                 Hc.push(temp)
23             end
24         end
25     end
26     if elem.start_col < elem.end_col - 1 then
27         elem.start_col ← temp.start_col + 1
28         elem.value_col ← csc_a.row_ind[elem.start_col]
29         Hr.push(elem)
30     end
31 end
32 return csr_res
```

5.2.2. Multiplicación de matriz por biclique

Al igual que en el caso anterior, se trata una representación basada en *Compress Sparse Column* y *Compress Sparse Row*. El conjunto de bicliques B^G extraídos de G, se presentan en el Cuadro 4.5, cuya matriz de adyacencia B se presenta en la Figura 5.12. Cada uno de los bicliques se representa de manera individual en *Compress Sparse Row*, como se muestra en la Figura 5.11. En este caso, *Compress Sparse Row* no requiere añadir el vector `row_ptr`.

b_1

`values:`	3	4	5
`col_ind:`	3	4	5
`row_id:`	1	2	6

(a) $CSR(b_1)$

b_2

`values:`	2	6	
`col_ind:`	2	6	
`row_id:`	1	3	5

(b) $CSR(b_2)$

b_3

`values:`	1	10	7
`col_ind:`	1	2	7
`row_id:`	4	6	

(c) $CSR(b_3)$

Figura 5.11: Bicliques en representación $CSR(B^G)$.

$$B = \begin{bmatrix} 0 & 2 & 3 & 4 & 5 & 6 & 0 \\ 0 & 0 & 3 & 4 & 5 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 6 & 0 \\ 1 & 10 & 0 & 0 & 0 & 0 & 7 \\ 0 & 2 & 0 & 0 & 0 & 6 & 0 \\ 1 & 10 & 3 & 4 & 5 & 0 & 7 \end{bmatrix}$$

Figura 5.12: Matriz de adyacencia B de las aristas de bicliques extraídos.

De esta forma, la multiplicación $A' \times B$ queda expresada en la Ecuación 5.10.

$$A' \times B = CSC(A') \times CSR(B) \tag{5.9}$$

$$A' \times B = CSC(A') \times (\sum_{i=0} CSR(b_i)), \quad b_i \in B^G \tag{5.10}$$

Para computar $A' \times B$ se debe operar de la siguiente manera:

1. Intersectar el vector `col_id` de $CSC(A')$ con cada uno de los vectores `row_id` de $CSR(b_i)$

`values:`	1	3	4	4	5	6
`row_ind:`	2	4	3	5	3	4
`col_ptr:`	0	1	2		4	5	6
`col_id:`	**1**	**3**	4		**5**	**6**	

Figura 5.13: Representación en *Compress Sparse Column*, $CSC(A')$.

b_1

`values:`	3	4	5
`col_id:`	3	4	5
`row_id:`	**1**	2	**6**

(a) $CSR(b_1)$

b_2

`values:`	2	6	
`col_id:`	2	6	
`row_id:`	**1**	**3**	**5**

(b) $CSR(b_2)$

b_3

`values:`	1	10	7
`col_id:`	1	2	7
`row_id:`	4	**6**	

(c) $CSR(b_3)$

2. Por cada intersección obtener el contenido de la columna de $CSC(A')$ y el contenido del biclique b_i

   Para b_1:

1 $\times$ **1**: (2,1) $\times$ (3,3) (4,4) (5,5)

6 $\times$ **6**: (4,6) $\times$ (3,3) (4,4) (5,5)

Para b_2:

$$\begin{aligned} \mathbf{1} \times \mathbf{1}: &\quad (2,1) \quad \times \quad (2,2)\ (6,6) \\ \mathbf{3} \times \mathbf{3}: &\quad (4,3) \quad \times \quad (2,2)\ (6,6) \\ \mathbf{5} \times \mathbf{5}: &\quad (3,5) \quad \times \quad (2,2)\ (6,6) \end{aligned}$$

Para b_3:

$$\begin{aligned} \mathbf{4} \times \mathbf{4}: &\quad (3,4) \quad \times \quad (1,1)\ (2,10)\ (7,7) \\ \mathbf{6} \times \mathbf{6}: &\quad (4,6) \quad \times \quad (1,1)\ (2,10)\ (7,7) \end{aligned}$$

3. Realizar producto cartesiano entre las intersecciones, almacenando en una *priority queue* los resultados como se ve en la Figura 5.16.

4. Construir la matriz resultante vaciando la *priority queue*. La Figura 5.15 y 5.17 muestra $A' \times B$ en representación *Compress Sparse Row* y matriz de adyacencia respectivamente

`values:`	2	3	4	5	6	10	30	6	18	24	30	18	
`col_ind:`	2	3	4	5	6	2	6	2	3	4	5	6	
`row_ptr:`	0					5		7					12
`row_id:`	2					3		4					

Figura 5.15: Representación en *Compress Sparse Row*, $CSR(A' \times B)$.

Las estructuras de datos y algoritmos 1, 2 descritos en la operación $Matriz \times Matriz$ también son utilizados en la operación descrita.

Priority queue	
(2,2,2)	(2,1) × (2,2)
(2,3,3)	(2,1) × (3,3)
(2,4,4)	(2,1) × (4,4)
(2,5,5)	(2,1) × (5,5)
(2,6,6)	(2,1) × (6,6)
(3,2,10)	(3,5) × (2,2)
(3,6,30)	(3,5) × (6,6)
(4,2,6)	(4,3) × (2,2)
(4,3,18)	(4,6) × (3,3)
(4,4,24)	(4,6) × (4,4)
(4,5,30)	(4,6) × (5,5)
(4,6,18)	(4,3) × (6,6)

Figura 5.16: Priority queue de $A' \times B$, a la izquierda los trios que representan las aristas ya ordenadas.

$$A' \times B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 4 & 5 & 6 & 0 \\ 0 & 10 & 0 & 0 & 0 & 30 & 0 \\ 0 & 6 & 18 & 24 & 30 & 18 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 5.17: Matriz de adyacencia con las aristas de $A' \times B$, despues de ser extraídas de la *priority queue*.

5.2.3. Multiplicación de biclique por biclique

	S	C
b_1	{1,2,6}	{(3,3) (4,4), (5,5)}
b_2	{1,3,5}	{(2,2) (6,6)}
b_3	{4,6}	{(1,2) (2,10), (7,7)}

Cuadro 5.2: Conjunto de bicliques extraídos de G

Para realizar la operación de multiplicación biclique por biclique, se deben expresar los bicliques utilizando *Compress Sparse Column* y *Compress Sparse Row*. Por lo que la multiplicación $B \times B$ queda expresada en la Ecuación 5.12.

$$B \times B = CSC(B) \times CSR(B) \tag{5.11}$$

$$B \times B = (\sum_i CSC(b_i)) \times (\sum_i CSC(b_i)), \quad b_i \in B^G \tag{5.12}$$

La representación en *Compress Sparse Column* y *Compress Sparse Row* de los bicliques se presenta en la Figura 5.18 y 5.19 respectivamente.

b_1

`values:`	3	4	5
`row_ind:`	1	2	6
`col_id:`	3	4	5

(a) $CSC(b_1)$

b_2

`values:`	2	6	
`row_ind:`	1	3	5
`col_id:`	2	6	

(b) $CSC(b_2)$

b_3

`values:`	1	10	7
`row_ind:`	4	6	
`col_id:`	1	2	7

(c) $CSC(b_3)$

Figura 5.18: Bicliques en representación *Compress Sparse Column* $CSC(B^G)$.

Si bien a simple vista puede parecer que para obtener la representación en CSC de un biclique solo se deben intercambiar de lugar los vectores `col_ind` y `row_id`

b_1

`values:`	3	4	5
`col_ind:`	3	4	5
`row_id:`	1	2	6

(a) $CSR(b_1)$

b_2

`values:`	2	6	
`col_ind:`	2	6	
`row_id:`	1	3	5

(b) $CSR(b_2)$

b_3

`values:`	1	10	7
`col_ind:`	1	2	7
`row_id:`	4	6	

(c) $CSR(b_3)$

Figura 5.19: Bicliques en representación *Compress Sparse Row* $CSR(B^G)$.

de su representación en CSR. En realidad la representación en CSC expresa la información de manera distinta. En la representación en CSR cada elemento i del vector `col_ind` estaba relacionado con su correspondiente i en el vector `values`. Mientras que el caso de CSC, cada elemento i del vector `col_id` esta relacionado con su correspondiente i en el vector `values`, por lo que referencia a cada una de las filas en el vector `row_ind` con un valor igual para todas.

Adicionalmente, se construye la tabla `marks`, donde a la izquierda se muestran todas las filas que están representadas en los bicliques, y a la derecha los bicliques a los cuales pertenece esa fila, ya que una fila puede pertenecer a varios bicliques, como se muestra en la Figura 5.20.

Marks	
1	b_1 b_2
2	b_1
3	b_2
4	b_3
5	b_2
6	b_1 b_3

Figura 5.20: Marks de bicliques, cada Nodo a la izquierda pertenece a uno o más bicliques de la columna de la derecha.

Ahora para computar $B \times B$ se debe operar de la siguiente manera:

1. Intersectar el vector `col_id` de la representación en CSC de cada biclique con el vector `row_id` de la representación en CSR de cada biclique, siempre y cuando representen bicliques distintos.

b_1

`values:`	3	4	5
`row_ind:`	1	2	6
`col_id:`	**3**	**4**	**5**

(a) $CSC(b_1)$

b_2

`values:`	2	6	
`row_ind:`	1	3	5
`col_id:`	**2**	**6**	

(b) $CSC(b_2)$

b_3

`values:`	1	10	7
`row_ind:`	4	6	
`col_id:`	**1**	**2**	7

(c) $CSC(b_3)$

Figura 5.21: Elementos intersectados del vector `col_id`.

b_1

`values:`	3	4	5
`col_ind:`	3	4	5
`row_id:`	**1**	**2**	**6**

(a) $CSR(b_1)$

b_2

`values:`	2	6	
`col_ind:`	2	6	
`row_id:`	**1**	**3**	**5**

(b) $CSR(b_2)$

b_3

`values:`	1	10	7
`col_ind:`	1	2	7
`row_id:`	**4**	**6**	

(c) $CSR(b_3)$

Figura 5.22: Elementos intersectados del vector `row_id`.

2. Por cada intersección $CSC(b_i) \cap CSR(b_j)$, obtener la suma de elementos del vector `values` de $CSC(b_i)$ que se intersectan y obtener el contenido de $CSR(b_j)$.

   Para b_1:

$$CSC(b_1) \times CSR(b_2) \rightarrow \quad (\mathbf{3}, \mathbf{5}) \quad \times \quad (\mathbf{3}, \mathbf{5}): \quad w = 3 + 5 \quad \times \quad (2,2)\ (6,6)$$

$$CSC(b_1) \times CSR(b_3) \rightarrow \quad (\mathbf{4}) \quad \times \quad (\mathbf{4}): \quad w = 4 \quad \times \quad (1,1)\ (2,10)\ (7,7)$$

Para b_2:

$$\begin{array}{llllll} CSC(b_2) \times CSR(b_1) \rightarrow & (\mathbf{2}, \mathbf{6}) & \times & (\mathbf{2}, \mathbf{6}): & w = 2+6 & \times \quad (3,3)\ (4,4)\ (5,5) \\ CSC(b_2) \times CSR(b_3) \rightarrow & (\mathbf{6}) & \times & (\mathbf{6}): & w = 6 & \times \quad (1,1)\ (2,10)\ (7,7) \end{array}$$

Para b_3:

$$\begin{array}{llllll} CSC(b_3) \times CSR(b_1) \rightarrow & (\mathbf{1}, \mathbf{2}) & \times & (\mathbf{1}, \mathbf{2}): & w = 1+10 & \times \quad (3,3)\ (4,4)\ (5,5) \\ CSC(b_3) \times CSR(b_2) \rightarrow & (\mathbf{1}) & \times & (\mathbf{1}): & w = 1 & \times \quad (2,2)\ (6,6) \end{array}$$

3. Por cada intersección, multiplicar el valor w, con el segundo elemento de cada par

b_1:

$$\begin{array}{lllll} w = 8 & \times & (2,2)\ (6,6) & = & (2,16)\ (6,48) \\ w = 4 & \times & (1,1)\ (2,10)\ (7,7) & = & (1,4)\ (2,40)\ (7,28) \end{array}$$

b_2:

$$\begin{array}{lllll} w = 2+6 & \times & (3,3)\ (4,4)\ (5,5) & = & (3,24)\ (4,32)\ (5,40) \\ w = 6 & \times & (1,1)\ (2,10)\ (7,7) & = & (1,6)\ (2,60)\ (7,42) \end{array}$$

b_3:

$$\begin{array}{lllll} w = 1+10 & \times & (3,3)\ (4,4)\ (5,5) & = & (3,33)\ (4,44)\ (5,55) \\ w = 1 & \times & (2,2)\ (6,6) & = & (2,2)\ (6,6) \end{array}$$

4. Finalmente, se debe concatenar los resultados de cada intersección, sumar si es que hay repeticiones y ordenar.

b_1:

$$\begin{array}{l} (2,16)\ (6,48) \\ (1,4)\ (2,40)(7,28) \end{array} \Longrightarrow \quad (1,4)\ (2,56)\ (6,48)\ (7,28)$$

b_2:

$$\begin{matrix}(3,24)\ (4,32)\ (5,40)\\(1,6)\ (2,60)\ (7,42)\end{matrix} \implies (1,6)\ (2,60)\ (3,24)\ (4,32)\ (5,40)\ (7,42)$$

b_3:

$$\begin{matrix}(3,33)\ (4,44)\ (5,55)\\(2,2)\ (6,6)\end{matrix} \implies (2,2)\ (3,33)\ (4,44)\ (5,55)\ (6,6)$$

De esta forma, obtenemos un resultado en forma de biclique, como se muestra en el Cuadro 5.3.

	S	C
b_1	{1,2,6}	{(1,4) (2,56) (6,48) (7,28)}
b_2	{1,3,5}	{(1,6) (2,60) (3,24) (4,32) (5,40) (7,42)}
b_3	{4,6}	{(2,2) (3,33) (4,44) (5,55) (6,6)}

Cuadro 5.3: Conjunto de bicliques obtenidos al operar $B \times B$.

Luego utilizando la tabla `marks` podemos obtener las listas de adyacencia de cada nodo como se ve en la Figura 5.23, la cual se puede utilizar para obtener la matriz de adyacencia B^2 representada en la Figura 5.24.

Lista de adyacencia de B^2		
1	$b_1 + b_2$	(1,10) (2,116) (3,24) (4,32) (5,40) (6,48) (7,70)
2	b_1	(1,4) (2,56) (6,48) (7,28)
3	b_2	(1,6) (2,60) (3,24) (4,32) (5,40) (7,42)
4	b_3	(2,2) (3,33) (4,44) (5,55) (6,6)
5	b_2	(1,6) (2,60) (3,24) (4,32) (5,40) (7,42)
6	$b_1 + b_3$	(1,4) (2,58) (3,33) (4,32) (5,55) (6,54) (7,28)

Figura 5.23: Listas de adyacencia obtenidas a partir de la tabla `marks`.

$$B^2 = \begin{bmatrix} 10 & 116 & 24 & 32 & 40 & 48 & 70 \\ 4 & 56 & 0 & 0 & 0 & 48 & 28 \\ 6 & 60 & 24 & 32 & 40 & 0 & 42 \\ 0 & 2 & 33 & 44 & 55 & 6 & 0 \\ 6 & 60 & 24 & 32 & 40 & 0 & 42 \\ 4 & 58 & 33 & 44 & 55 & 54 & 28 \end{bmatrix}$$

Figura 5.24: Matriz de adyacencia resultante B^2.

Estructuras de datos

A continuación se presentan las estructuras de datos utilizadas en para la implementación de la operación *Biclique* × *Biclique*

`Struct CSC_bicl{`

- `vector<int>values`
- `vector<int>row_ind`
- `vector<int>col_id`

}

Figura 5.25: Estructura Compress Sparse Column para bicliques.

`Struct CSR_bicl{`

- `vector<int>values`
- `vector<int>col_ind`
- `vector<int>row_id`

}

Figura 5.26: Estructura Compress Sparse Row para bicliques.

`Struct Intersection_bicl{`

- `vector<int> S`
- `vector<int,int> C`

}

Figura 5.27: Estructura Intersección con bicliques. Cuando se intersectan dos bicliques en CSC y CSR, en `S` se almacena el `row_ind` de CSC, mientras que en `C` se almacenan los indices intersectados de `col_ind` en CSR.

Algoritmos

A continuación se presentan los algoritmos en *pseudocódigo*, los pasos 1, 2 y 3 descritos son realizados por el Algoritmo 3, mientras que el paso 4 es realizado por el Algoritmo 4, dando el resultado en forma de biclique. Luego es posible construir la matriz resultante, utilizando la tabla `marks` como se mencionó anteriormente.

Algorithm 3: Get Intersections_Bicl

Entrada: `csc_a, csr_b`: Bicliques en CSC y CSR respectivamente

Salida : `Inters`: vector Intersections_bicl

```
1  Inters ← vector Intersection_bicl
2  i ← 0
3  j ← 0
4  while i < csc_a.col_id.size() and j < csr_b.row_id.size() do
5      if csc_a.col_id[i] == csr_b.row_id[j] then
6          inter ← Struct Intersection_bicl
7          inter.S ← csc_a.col_id
8          k ← 0
9          while k < csr_b.col_ind.size() do
10             inter.C.push(csr_b.col_ind[k],
11                 csr_b.values[k] × csc_a.values[j])
12             k ← k + 1
13         end
14         i ← i + 1
15         j ← j + 1
16     end
17     else if csc_a.col_id[i] > csr_b.row_id[j] then
18         j ← j + 1
19     end
20     else
21         i ← i + 1
22     end
23 end
24 return Inters
```

Algorithm 4: Multiplicación Biclique × Biclique

Entrada: `v_csc_a, v_csr_b`: Vector de bicliques en representación CSC y CSR

Salida : `res`: Vector de resultados por biclique

```
1  for csc_a : v_csc_a do
2      for csr_b : v_csr_b do
3          inter ← getInters_bicl(csc_a, csr_b)
4          if inter.size() > 0 then
5              temp ← Empty vector
6              for i : inter.C do
7                  if temp.empty() or temp.back().first != i.first then
8                      temp.push(i)
9                  end
10                 else
11                     temp.back().second ← temp.back().second + i.second
12                 end
13             end
14             inter.C ← temp
15         end
16         res.push(inter)
17     end
18 end
19 return res
```

5.2.4. Multiplicación de biclique por matriz

b_1

values:	3	4	5
row_ind:	1	2	6
col_id:	3	4	5

(a) $CSC(b_1)$

b_2

values:	2	6	
row_ind:	1	3	5
col_id:	2	6	

(b) $CSC(b_2)$

b_3

values:	1	10	7
row_ind:	4	6	
col_id:	1	2	7

(c) $CSC(b_3)$

Figura 5.28: Bicliques en representación $CSC(B^G)$.

values:	1	4	5	3	6	4	
col_ind:	1	4	5	3	6	4	
row_ptr:	0	1		3		5	6
row_id:	2	3		4		5	

Figura 5.29: Representación en *Compress Sparse Row* $CSR(A')$.

Esta operación es un caso similar al anterior, para ello se toma la representación en *Compress Sparse Column* del conjunto de bicliques B^Gcomo se muestra en la Figura 5.28 y la representación en *Compress Sparse Column* de A' de la Figura 5.29. La multiplicación se $B \times A'$ queda expresada en la Ecuación 5.14

$$B \times A' = CSC(B) \times CSR(A') \tag{5.13}$$

$$B \times A' = (\sum_i CSC(b_i)) \times A', \quad b_i \in B^G \tag{5.14}$$

Luego para computar $B \times A'$ se debe operar de la siguiente manera:

1. Intersectar el vector `col_id` de la representación CSC de cada biclique con el vector `row_id`.

b_1

values:	3	4	5
row_ind:	1	2	6
col_id:	**3**	4	**5**

(a) $CSC(b_1)$

b_2

values:	2	6	
row_ind:	1	3	5
col_id:	**2**	6	

(b) $CSC(b_2)$

b_3

values:	1	10	7
row_ind:	4	6	
col_id:	1	**2**	7

(c) $CSC(b_3)$

Figura 5.30: Elementos intersectados de los bicliques del vector `col_id`.

values:	1	4	5	3	6	4	
col_ind:	1	4	5	3	6	4	
row_ptr:	0	1		3		5	6
row_id:	**2**	**3**		**4**		**5**	

Figura 5.31: Elementos intersectados de del vector `row_id`.

2. Por cada intersección $CSC(b_i) \cap CSR(A')$, obtener la suma de elementos del vector `values` de $CSC(b_i)$ que se intersectan y obtener el contenido de la fila de $CSR(A')$

   Para b_1:

   (3) $\times$ **(3)**: $w = 3$ $\times$ (4,4) (5,5)

   (4) $\times$ **(4)**: $w = 4$ $\times$ (3,3) (6,6)

   (5) $\times$ **(5)**: $w = 5$ $\times$ (4,4)

   Para b_2:

   (2) $\times$ **(2)**: $w = 2$ $\times$ (1,1)

   Para b_3:

   (2) $\times$ **(2)**: $w = 10$ $\times$ (1,1)

3. Por cada intersección, multiplicar el valor w, con el segundo elemento de cada par.

   Para b_1:

$$\begin{aligned} w = 3 &\times (4,4)\ (5,5) &&= (4,12)\ (5,15) \\ w = 4 &\times (3,3)\ (6,6) &&= (3,12)\ (6,24) \\ w = 5 &\times (4,4) &&= (4,20) \end{aligned}$$

   Para b_2:

$$w = 2 \times (1,1) = (1,2)$$

   Para b_3:

$$w = 10 \times (1,1) = (1,10)$$

4. Finalmente, se debe concatenar los resultados de cada intersección, sumar si es que hay repeticiones y ordenar.

   b_1:

$$\begin{array}{l} (4,12)\ (5,15) \\ (3,12)\ (6,24) \\ (4,20) \end{array} \implies (3,12)\ (4,32)\ (5,15)\ (6,24)$$

b_2:

$$(1,2) \implies (1,2)$$

b_3:

$$(1,10) \implies (1,10)$$

De esta forma, obtenemos un resultado en forma de biclique, como se muestra en el Cuadro 5.3.

	S	C
b_1	{1,2,6}	{(3,12) (4,32) (5,15) (6,24)}
b_2	{1,3,5}	{(1,2)}
b_3	{4,6}	{(1,10)}

Cuadro 5.4: Conjunto de bicliques obtenidos de $B \times A'$.

Luego utilizando la tabla `marks` como en el caso anterior podemos obtener las listas de adyacencia de cada nodo como se ve en la Figura 5.32, la cual se puede utilizar para obtener la matriz de adyacencia B^2 representada en la Figura 5.33

Lista de adyacencia de B^2		
1	$b_1 + b_2$	(1,2) (3,12) (4,32) (5,15) (6,24)
2	b_1	(3,12) (4,32) (5,15) (6,24)
3	b_2	(1,2)
4	b_3	(1,10)
5	b_2	(1,2)
6	$b_1 + b_3$	(1,10) (3,12) (4,32) (5,15) (6,24)

Figura 5.32: Listas de adyacencia obtenida a partir de la tabla `marks`.

$$B \times A' = \begin{bmatrix} 2 & 0 & 12 & 32 & 15 & 24 & 0 \\ 0 & 0 & 12 & 32 & 15 & 24 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 0 & 12 & 32 & 15 & 24 & 0 \end{bmatrix}$$

Figura 5.33: Matriz de adyacencia resultante $B \times A'$.

Las estructuras de datos y los algoritmos presentados para la operación *Biclique* × *Biclique* son equivalentes para la operación actual.

Una vez computadas todas las operaciones solo se deben sumar las matrices para obtener A^2, dando como resultado la matriz de la Figura 5.34

$$A^2 = \begin{bmatrix} 12 & 116 & 36 & 64 & 55 & 72 & 70 \\ 4 & 58 & 15 & 36 & 20 & 78 & 28 \\ 12 & 110 & 36 & 52 & 40 & 54 & 70 \\ 16 & 68 & 51 & 80 & 100 & 24 & 42 \\ 12 & 100 & 36 & 32 & 40 & 24 & 70 \\ 14 & 58 & 45 & 76 & 70 & 78 & 28 \end{bmatrix}$$

Figura 5.34: Matriz de adyacencia resultante A^2, obtenida por la suma de todas las operaciones.

Capítulo 6

Resultados

Todas las pruebas y experimentos fueron desarrolladas en `C++` utilizando el compilador `g++` junto con `-std=c++20 -O3` como flags de compilación. Además para su ejecución se utilizó un servidor del laboratorio de VLSI del Departamento Eléctrico de la Universidad de Concepción que cuenta con un Intel(R) Xeon(R) Gold 5118 CPU @2.30GHz junto con 128GB de memoria RAM.

6.1. Generador de grafos

Como parte del desarrollo se trabajó en un Generador de Grafos Sintéticos, con tal de medir las capacidades del Extractor de Bicliques del Capítulo 4 y la Multiplicación de Matrices del Capítulo 5.

El Generador de Grafos permite generar grafos con una cantidad definida de aristas, donde un porcentaje de ellas forman bicliques de distintos tamaño. Los parámetros para el generador se presentan a continuación:

- `num_edges`: Cantidad de aristas totales del grafo.
- `compression_percentage`: Porcentaje de aristas presentes en bicliques.
- `SxC_average`: Cantidad de aristas promedio por biclique.

Por ejemplo, para los siguientes parámetros:

- `num_edges=` 100,000
- `compression_percentage:` 40 %
- `SxC_average:` 400

Se genera un grafo con 100,000 aristas, de las cuales 40 % pueden ser representadas en bicliques, esto es 40,000 aristas en bicliques. Además, el tamaño promedio de un biclique es de 400 aristas.

El Generador de Grafos además establece una *Distribución Normal* con una *Desviación estándar* igual al 20 % del parámetro `SxC_average`. Luego, los tamaños de los bicliques estarían entre 320 y 480 aristas. De esta manera, la cantidad de bicliques a generar es variable al igual que los tamaños de éstos y se generan hasta que se supere o iguale la cantidad de aristas que deben estar en total entre todos los bicliques (40,000 en este ejemplo). Finalmente se completará el grafo hasta llegar a la cantidad de aristas requeridas.

Si bien el tamaño de los bicliques es variable, el Generador de Grafos siempre producirá bicliques *balanceados*, es decir, $|S| = |C|$. Para ello una vez el generador establece la cantidad de aristas que contiene el biclique, se computa el tamaño de $|S|$ y $|C|$, según la Ecuación 6.1.

$$|S| = |C| = \sqrt{|S \times C|} \tag{6.1}$$

Luego el Generador de Grafos entrega 3 archivos de salida, el primero un archivo con los bicliques del grafo (B^G), el segundo es el grafo sin los bicliques (A') y el tercero es el grafo completo incluyendo bicliques (A).

6.2. Extracción de bicliques

En esta sección se evalúa el rendimiento de la identificación y extracción de bicliques del Capítulo 4 sobre grafos generados. Para la extracción en cada instancia se utilizaron los parámetros dados a continuación:

Parámetros:

- `shingle_size`: 1
- `num_signatures`: 2
- `min_ady_nodes`: 30
- `min_cluster_size`: 50
- `biclique_size`: $|S \times C|$ *average*
- `threashold`: 100
- `bs_decrease`: $\frac{\texttt{biclique\_size}}{5}$
- `iterations`: 100

Con la excepción del parámetro `biclique_size`, el cual para cada instancia se estableció el mismo valor que el tamaño promedio de los bicliques, y ($|S \times C|$ *average*), y el parámetro `bs_decrease` el cual se estableció en un valor equivalente a un 20 % del `biclique_size`.

Para evaluar el capacidad del Extractor de Bicliques, se generaron distintos grafos, manteniendo constante la cantidad de aristas entre todos ellos, y se varió el porcentaje de compresión, es decir, la cantidad de aristas que pueden ser representadas en bicliques. También por cada grado de compresión, se varió el tamaño promedio de los bicliques del grafo, con el propósito de medir la incidencia del tamaño de los bicliques en el extractor. La columna `Grafo` muestra la información general del grafo. El Cuadro 6.1 muestra los resultados obtenidos en este experimento.

Grafo			Bicliques Originales		Bicliques Extraídos				Cobertura	
Aristas	%	$\lvert S \times C \rvert$ avg	Num Bicl	Aristas en Bicl	Num Bicl	Aristas en Bicl	Iter	Time(s)	Num Bicl	Aristas (%)
100.000.000	20	1024	19350	18006873	26938	18006874	28	419,451	139,47 %	100,00 %
		4096	4770	19899395	9362	19899396	17	295,033	196,62 %	100,00 %
		16384	1182	19963022	2421	19963023	12	214,034	204,82 %	100,00 %
		65536	292	20037909	580	20037909	8	189,282	198,63 %	100,00 %
	40	1024	38747	35019338	51351	35019339	35	844,405	132,52 %	100,00 %
		4096	9601	39776857	18792	39776857	19	589,749	195,72 %	100,00 %
		16384	2373	39910894	4956	39910895	13	372,888	208,84 %	100,00 %
		65536	587	40009488	1227	40009488	10	366,194	209,02 %	100,00 %
	60	1024	58040	50821146	72850	50821146	36	1042,95	125,51 %	100,00 %
		4096	14268	59665741	27944	59665741	21	817,022	195,85 %	100,00 %
		16384	3529	59856390	7427	59856391	13	484,656	210,45 %	100,00 %
		65536	882	59933828	1844	59933828	11	507,135	209,07 %	100,00 %
	80	1024	77353	64808063	91488	64808064	34	1047,98	118,27 %	100,00 %
		4096	19052	79537183	37087	79537183	23	1048,31	194,66 %	100,00 %
		16384	4732	79815200	9861	79815200	13	567,969	208,38 %	100,00 %
		65536	1188	79915462	2540	79915463	12	612,432	213,80 %	100,00 %

Cuadro 6.1: Estadísticas del EXTRACTOR DE BICLIQUES sobre grafos generados con distinta cantidad de bicliques y distinto tamaño de bicliques.

La columna `Bicliques Originales` muestra información sobre los bicliques que contiene el grafo generado. Se especifica tanto la cantidad de bicliques totales que contiene el grafo, así como también la cantidad de aristas totales que se representan en todos los bicliques.

Los bicliques extraídos se presentan en la columna `Bicliques Extraídos`, en ella se presentan la cantidad de bicliques obtenidos por cada instancia, la cantidad de aristas totales entre todos los bicliques obtenidos, la cantidad de iteraciones requeridas para obtener los bicliques, y el tiempo total de ejecución.

Finalmente la columna `Cobertura` presenta estadísticas de los bicliques extraídos comparado con los bicliques originales del grafo. Para ello se define la *Cobertura de Num de Bicliques* y la *Cobertura de Aristas* dados por la Ecuación 6.2 y 6.3

respectivamente.

$$CoberturaNumBicliques = \frac{NumBicliquesObtenidos}{NumBicliquesOriginales} \times 100 \tag{6.2}$$

$$CoberturaAristas = \frac{NumAristasObtenidas}{NumAristaOriginales} \times 100 \tag{6.3}$$

La *Cobertura de Num de Bicliques* es una medida que indica que tan preciso es el extractor de bicliques en términos de número de bicliques, mientras que la *Cobertura de Aristas* mide la capacidad del extractor de recuperar las aristas presentes en bicliques.

Los resultados presentados en el Cuadro 6.1 muestran que en cada caso el extractor de bicliques es capaz de recuperar el 100 % de las aristas de los bicliques originales. Sin embargo, también existe un incremento en la cantidad de bicliques obtenidos, esto se debe principalmente a la primera etapa de clustering, debido a que MinHash es una técnica probabilística para estimar similitud entre dos conjuntos y a que solo se almacenan dos signatures, afectando así la capacidad del clustering para detectar las listas de adyacencia similares.

En ciertos casos, se puede ver que la cantidad de aristas obtenidas en bicliques difiere de la cantidad de aristas originales en bicliques. Esto se da por las características del Generador de Grafos, que agrega aristas de manera aleatoria, por lo que puede darse el caso de que bicliques pequeños tengan otra aristas adicionales y sean descubiertas por el extractor.

También se puede apreciar la influencia que tienen los distintos tamaños de los bicliques en la extracción de éstos. A medida que el tamaño promedio de bicliques crece, la cantidad de iteraciones requeridas para recuperar disminuye en todos los casos, al igual que el tiempo con algunas excepciones. De igual forma, la cantidad de bicliques obtenidos aumenta a medida que aumenta el tamaño de los bicliques.

Es decir, entre mayor es el tamaño de los bicliques, menos probable es obtener el biclique completo.

Grafo			Bicliques Originales				Bicliques Extraídos							
Aristas	%	$\|S \times C\|$ Avg	$\|S\|$ $\|C\|$ Min	$\|S\|$ $\|C\|$ Avg	$\|S\|$ $\|C\|$ Max	Desv Std	$\|S\|$ Min	$\|S\|$ Avg	$\|S\|$ Max	$\|S\|$ Desv Std	$\|C\|$ Min	$\|C\|$ Avg	$\|C\|$ Max	$\|C\|$ Desv Std
100.000.000	20	1024	9	32	56	**6,44**	2	20	54	**13,89**	10	32	56	**5,90**
		4096	5	64	112	**12,99**	2	32	111	**27,72**	10	64	112	**12,78**
		16384	31	128	209	**25,38**	2	63	197	**59,33**	31	128	209	**25,40**
		65536	103	256	366	**52,20**	2	127	365	**124,48**	103	253	366	**52,37**
	40	1024	4	32	57	**6,42**	2	20	52	**14,07**	10	33	57	**5,79**
		4096	13	64	113	**12,82**	2	32	109	**28,20**	8	64	113	**12,63**
		16384	37	128	209	**25,56**	2	61	198	**59,98**	37	128	209	**25,64**
		65536	50	256	424	**54,16**	2	126	412	**127,34**	50	258	424	**54,26**
	60	1024	7	32	56	**6,45**	2	20	54	**14,31**	10	34	56	**5,55**
		4096	16	64	109	**12,87**	2	32	104	**28,78**	11	65	109	**12,55**
		16384	29	128	218	**25,56**	2	61	204	**60,86**	29	129	218	**25,37**
		65536	95	256	428	**51,58**	2	122	428	**127,55**	95	258	428	**51,11**
	80	1024	6	32	59	**6,43**	2	20	56	**14,71**	8	34	59	**25,47**
		4096	14	64	122	**12,70**	2	32	118	**29,28**	10	65	153	**12,34**
		16384	36	128	216	**25,47**	2	61	214	**61,80**	36	129	262	**25,29**
		65536	71	256	410	**50,15**	2	120	401	**126,84**	71	256	410	**49,46**

Cuadro 6.2: Estadísticas de tamaños entre los bicliques originalmente generados y los bicliques encontrados, entre mayor es el tamaño del biclique original más se aleja el extractor de recuperar ese biclique.

Además, el Cuadro 6.2 muestra información sobre el tamaño de los bicliques extraídos. La columna `Bicliques Originales` muestra información sobre los tamaños de los conjuntos S y C de los bicliques. Como los bicliques generados por el Generador de Grafos son *bicliques balanceados*, las estadísticas sobre los tamaños de S y C son equivalentes. La columna $|S||C|Min$ entrega el tamaño de S y C del biclique más pequeño presente en el grafo, mientras que $|S||C|Max$ del biclique más grande. $|S||C|Avg$ es el tamaño promedio de S y C de los bicliques, el cual esta

relacionado al parámetro $|S \times C|Avg$ que se utiliza para generar el grafo.

La columna `Bicliques Extraídos` muestra información sobre los tamaños de los bicliques obtenidos por el Extractor de Bicliques, se puede ver que las estadísticas para la columnas de $|S|$ en todos los datos obtenidos se tienen valores por debajo de los originales. Mientras que las estadísticas obtenidas para $|C|$ mantienen una cercanía con los originales.

Los resultados obtenidos complementan los resultados anteriores en los que se obtenían más bicliques que los que habían originalmente, si se tiene que $|S|Avg$ es menor en los bicliques obtenidos y además tiene una desviación estándar mayor. Esto refleja que los bicliques obtenidos tienden a tener menos nodos, por lo cual el Extractor de Bicliques utiliza más bicliques para representar las aristas de un único biclique original. Un ejemplo de este artefacto se muestra en la Figura 6.1.

	S	C
b_1	$\{s_1, s_2, s_3, s_4, s_5, s_6\}$	$\{c_1, c_2, c_3\}$

(a) Biclique original

	S	C
b_1	$\{s_1, s_4\}$	$\{c_1, c_2, c_3\}$
b_2	$\{s_2, s_5\}$	$\{c_1, c_2, c_3\}$
b_3	$\{s_3, s_6\}$	$\{c_1, c_2, c_3\}$

(b) Bicliques obtenidos

Figura 6.1: Comparación bicliques originales vs. obtenidos, a la izquierda un ejemplo de un biclique generado, a la derecha una posible descomposición hecha por el Extractor de Bicliques.

6.3. Multiplicación de matrices

En esta sección se evalúa el rendimiento de las distintas operaciones descritas en el Capitulo 5 sobre grafos generados.

En este caso, para comparar la multiplicación con bicliques y sin bicliques se generaron grafos con distintas cantidades de aristas y distintos grados de compresión. En todos los casos se mantuvo constante el tamaño promedio de bicliques, establecido por defecto en $|S \times C|Avg = 16384$. El cuadro 6.3 muestra los resultados obtenidos en este experimento.

La columna `Grafo` del cuadro muestra la información respectiva del grafo generado, mientras que la columna `Bicliques` muestra la información de los bicliques presentes en el grafo en cuestión para utilizarlos para obtener la multiplicación entre ellos.

Los tiempos de las distintas etapas de la multiplicación como bicliques se muestran en la columna `Etapas mult c/ bicliques`, en ella además se muestra la subcolumna `Sum`, que corresponde a la operación de construir la matriz resultante total sumando todas las matrices parciales de cada una de las operaciones desarrolladas.

Finalmente la columna `Tiempo` muestra los tiempos que requiere el algoritmo para obtener la matriz resultante sin utilizar bicliques, utilizando bicliques y el porcentaje de mejora con respecto al otro. El porcentaje de mejora se define según la Ecuación 6.4.

$$Mejora = \frac{TiempoNoBicliques - TiempoBicliques}{TiempoNoBicliques} \times 100 \tag{6.4}$$

Los resultados obtenidos en la columna `No Bicl` de `Tiempo` corresponden realizar la operación $Matriz \times Matriz$ utilizando el dataset original, para obtener el cuadrado de la matriz, es decir $A \times A$. Mientras que la columna `Bicl` de `Tiempo` es el tiempo para obtener el cuadrado de la matriz utilizando bicliques, para ello utiliza la representación que incluye los bicliques extraídos B, y A' como la matriz restante.

Grafo		Bicliques		Etapas mult c/ bicliques (s)					Tiempo(s)		
Aristas	%	Num Bicl	Aristas en Bicl	$A' \times A'$	$A' \times B$	$B \times A'$	$B \times B$	Sum	No Bicl	Bicl	Mejora %
10.000.000	20	122	2015486	1,175	0,030	0,052	0,006	0,096	1,494	1,430	**4,28 %**
	40	232	4025545	0,630	0,038	0,060	0,026	0,081	0,937	0,869	**7,25 %**
	60	347	6013015	0,269	0,040	0,061	0,051	0,065	0,789	0,494	**37,38 %**
	80	471	8004932	0,071	0,028	0,051	0,086	0,056	0,368	0,2865	**22,14 %**
20.000.000	20	239	4006051	4,315	0,081	0,125	0,026	0,414	6,608	5,583	**15,51 %**
	40	466	8019916	2,215	0,123	0,161	0,092	0,355	4,378	3,430	**21,65 %**
	60	709	12005423	0,977	0,116	0,163	0,209	0,320	3,642	1,960	**46,18 %**
	80	924	16010103	0,249	0,081	0,130	0,332	0,264	2,169	1,071	**50,62 %**
40.000.000	20	463	8004924	15,068	0,323	0,345	0,091	1,571	24,183	20,820	**13,90 %**
	40	933	16001640	10,346	0,527	0,529	0,326	1,525	19,563	12,305	**37,10 %**
	60	1447	24010876	4,727	0,487	0,558	0,720	1,497	13,678	8,107	**40,72 %**
	80	1909	32019312	0,982	0,291	0,389	1,308	1,397	10,739	4,646	**56,73 %**
60.000.000	20	694	12002843	31,214	0,674	0,738	0,191	4,262	55,774	42,242	**24,26 %**
	40	1398	24004676	21,520	1,061	1,135	0,748	3,279	42,939	29,652	**30,94 %**
	60	2138	36004010	8,761	1,145	1,152	1,728	3,376	35,429	17,912	**49,44 %**
	80	2861	48002497	2,540	0,735	0,813	2,983	3,252	25,660	10,757	**58,07 %**
80.000.000	20	949	16010871	53,472	1,329	1,266	0,327	6,777	68,517	62,427	**8,88 %**
	40	1925	32005200	32,272	2,062	2,012	1,297	5,940	63,062	46,501	**26,26 %**
	60	2845	48015938	17,603	2,238	2,110	3,033	6,231	54,775	31,197	**43,04 %**
	80	3746	64010736	4,561	1,456	1,458	5,415	6,014	40,750	19,835	**51,32 %**
100.000.000	20	1182	19963022	66,301	2,049	1,964	0,535	10,302	95,409	93,234	**2,27 %**
	40	2373	39910894	42,538	3,294	2,869	2,177	10,802	91,552	73,116	**20,13 %**
	60	3529	59856390	21,664	3,117	3,020	4,732	10,977	93,269	53,359	**42,79 %**
	80	4732	79815200	6,096	2,209	2,272	8,685	9,844	71,635	38,785	**45,85 %**

Cuadro 6.3: Estadísticas de multiplicación con y sin bicliques, evalúados en grafos de distinto tamaño con distintos grados de compresión y con el detalle del tiempo de computo de cada operación.

El tiempo obtenido corresponde a la suma de los tiempos de cada una de las operaciones $(A' \times A')$, $(A' \times B)$, $(B \times A')$ y $(B \times B)$ y sumado además el tiempo de la operación de `Sum`.

Se puede ver que en todos los casos hay mejora al utilizar la multiplicación con bicliques, llegando hasta a un 58 % en el mejor caso obtenido. Cabe mencionar que los bicliques no solo tienen una influencia de mejora en la multiplicación con bicliques propiamente tal, sino que también se puede ver que lo grafos cuyas aristas están representadas en bicliques la multiplicación normal tiende a tener menores tiempos. Esto puede deberse principalmente al hecho de representar la matriz como *Compress Sparse Row*, y al tener aristas repartidas en menos nodos, es decir, filas con más información.

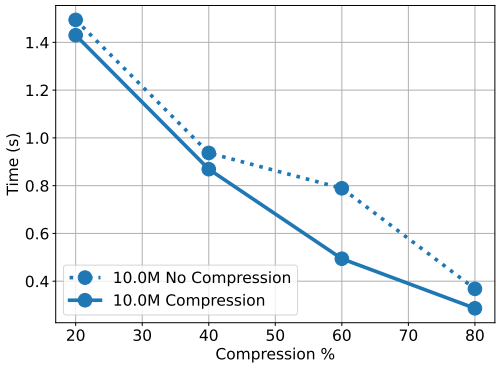

Figura 6.2: Comparación de tiempos con distintos grados de compresión con 10.000.000 de aristas.

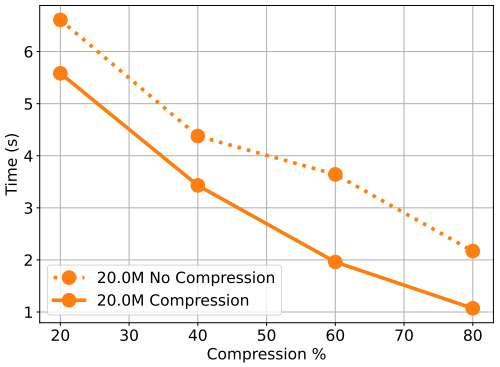

Figura 6.3: Comparación de tiempos con distintos grados de compresión con 20.000.000 de aristas.

Las Figuras 6.2, 6.3, 6.4, 6.5, 6.6 y 6.7 muestran una comparativa de los tiempos obtenidos la multiplicación con compresión y sin compresión para 10.000.000, 20.000.000, 40.000.000, 60.000.000, 80.000.000 y 100.000.000 respectivamente. Se puede apreciar que la pendiente de la curva de la multiplicación con compresión es favorable en cada caso obtenido y se ve mayor diferencia en el caso de la multiplicación con 100.000.000 de aristas.

La Figura 6.8 muestra una comparativa de todos los tiempos de todos los grafos generados utilizando escala logarítmica para el eje vertical de tiempo, donde se evidencia aún más que la compresión juega un factor a favor a la hora de multiplicación en comparación a la multiplicación de matrices tradicional.

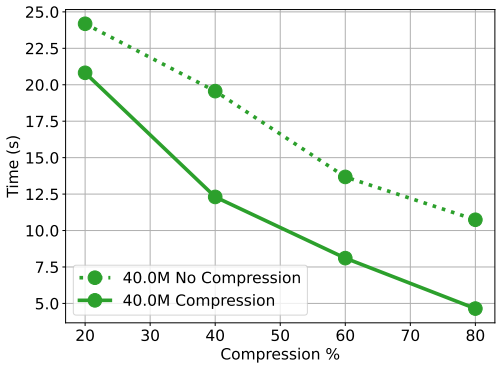

Figura 6.4: Comparación de tiempos con distintos grados de compresión con 40.000.000 de aristas.

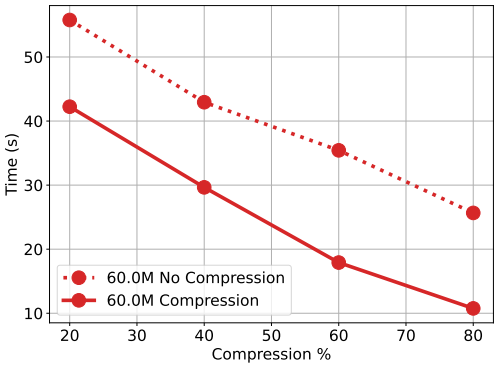

Figura 6.5: Comparación de tiempos con distintos grados de compresión con 60.000.000 de aristas.

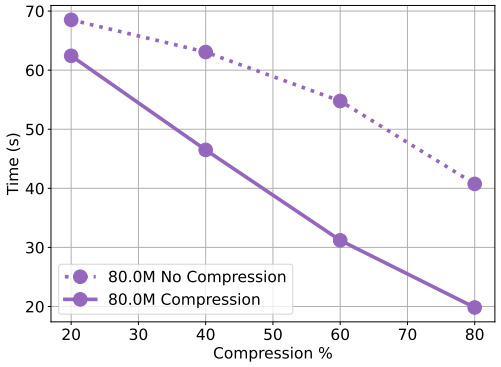

Figura 6.6: Comparación de tiempos con distintos grados de compresión con 80.000.000 de aristas.

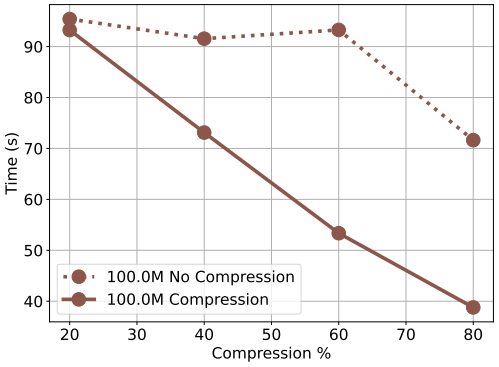

Figura 6.7: Comparación de tiempos con distintos grados de compresión con 100.000.000 de aristas.

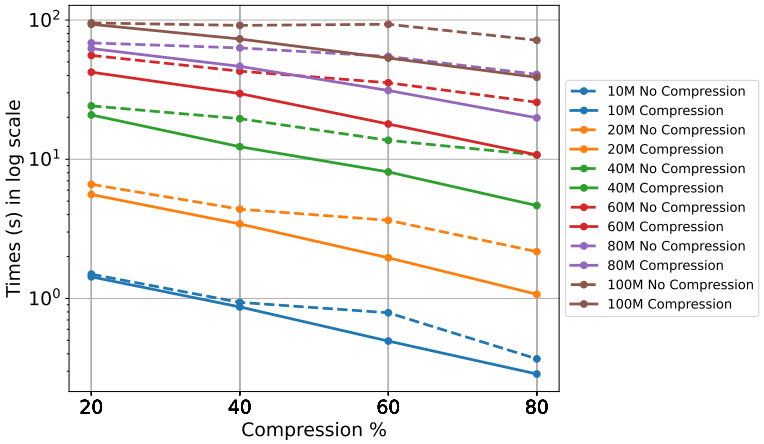

Figura 6.8: Comparativa de los tiempos obtenidos de todos los grafos en escala logaritmica.

También se realizó el experimento utilizando distintos tamaños de bicliques con el fin de medir el impacto que pueda conllevar en el tiempo de las operaciones, manteniendo constante el tamaño del grafo y la compresión. El cuadro 6.4 muestra el experimento realizado, se tomó como base el dataset con mayor cantidad de aristas (100.000.000) y mayor grado de compresión (80 %).

Graph			Bicliques		Mult stages w/ Bicliques (s)					Times (s)		
Edges	%	$\|S \times C\|$ Avg	Num Bicl	Edges in Bicl	$A' \times A'$	$A' \times B$	$B \times A'$	$B \times B$	*Sum*	No Bicl	Bicl	Improv %
100.000.000	80	1024	77589	80000848	7,214	4,674	4,035	14,574	8,20	75,653	47,399	35,92 %
	80	4096	19089	80002286	7,109	3,136	2,994	11,650	7,74	73,011	41,151	43,63 %
	80	16384	4741	80013598	7,133	2,374	2,308	10,522	8,08	71,635	38,785	45,85 %
	80	65536	1196	80054797	6,937	2,005	2,034	10,065	7,85	70,661	37,154	47,41 %
	80	262144	293	80311225	5,982	1,793	1.881	7.569	7,635	60,019	24,860	58,57 %
	80	1098204	77	80348231	5.969	1.708	1.830	7.579	7,497	58,390	24,583	57,89 %
	80	4194304	19	80602275	5.899	1.641	1.831	8.424	9,064	65,770	26,859	59,16 %

Cuadro 6.4: Estadísticas de multiplicación con y sin bicliques para un grafo de 100.000.000 de aristas, 80 % de aristas en bicliques y distinta configuración de tamaño de bicliques.

En los resultados presentados se puede evidenciar el impacto que tienen el tamaño

de los bicliques en la multiplicación, aumentando el margen de mejora a medida que estos crecen. Esto se puede explicar principalmente por el hecho de que al trabajar con este tipo de bicliques, se disminuye la cantidad de veces que se deben realizar intersecciones entre ellos. Por lo tanto, para obtener mejores tiempos es deseable trabajar con bicliques de mayor tamaño.

Capítulo 7

Conclusión y trabajo futuro

7.1. Conclusiones

El primer objetivo establecido para este trabajo fue expandir la propuesta de Hernández y Navarro [1] para el descubrimiento de bicliques en grafos con pesos y de gran tamaño. Para ello se trabajó con una reimplementación del código utilizando `C++`. Utilizando el Generador de Grafos fue posible caracterizar grafos y bicliques y evaluar la calidad de bicliques obtenidos así como también bajo que condiciones el Extractor de bicliques obtiene mejores resultados. Los resultados obtenidos son favorables en términos de compresión logrando recuperar en su totalidad la cantidad de aristas en bicliques, considerando que el algoritmo no busca garantizar obtener los mejores bicliques si no obtener bicliques lo suficientemente buenos en tiempo competitivo.

En cuanto al segundo objetivo establecido que busca definir una estructura que permita el cálculo de multiplicación de matrices generales utilizando bicliques. Se obtuvo una implementación en `C++` que aprovecha los bicliques extraídos para reducir tiempo de computo en operaciones de multiplicación, además con la evaluación experimental se pudo evidenciar que tipos de bicliques favorecen en la multiplicación más allá del porcentaje de compresión que se pueda obtener.

7.2. Trabajo futuro

Entre las líneas de investigación futuras, se puede mencionar el análisis sobre el Extractor de Bicliques en grafos reales, a los cuales no se le conoce de antemano los bicliques presentes y por lo tanto no se conocen los parámetros óptimos para utilizar. En ese sentido, se podría realizar un preprocesamiento del grafo para conocer por ejemplo el grado promedio de sus nodos, la distribución del numero de aristas, entre otros y a partir de ahí establecer parámetros que podrían ser prometedores. O bien buscar patrones utilizando Machine Learning para encontrar aquellos parámetros que favorezcan al extractor.

También, apuntando esta vez hacia la compresión de grafo sería interesante profundizar en estructuras compactas que permitan reducir más aún el espacio utilizado pero que permitan de igual manera soportar realizar operaciones como la multiplicación de matrices en tiempo competitivo.

Por otro lado, sería interesante evaluar el comportamiento de la Multiplicación de Matrices con Bicliques con un enfoque más especifico y ver como podría mejorar tiempos de computo de algoritmos que utilizan estas operaciones, considerando por una parte el tiempo de la extracción de bicliques y por otra parte el tiempo de la multiplicación matricial.

Bibliografía

[1] Cecilia Hernández y Gonzalo Navarro. «Compressed representations for web and social graphs». En: *Knowledge and information systems* (2014), págs. 279-313.

[2] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu y Yizhuo Wang. «A systematic survey of general sparse matrix-matrix multiplication». En: *ACM Computing Surveys* (2023), págs. 1-36.

[3] Ali Pinar y Michael T Heath. «Improving performance of sparse matrix-vector multiplication». En: *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. 1999.

[4] Ravindra B. Bapat. *Graphs and matrices*. Springer, 2010, págs. 70-71.

[5] Felipe Glaria, Cecilia Hernández, Susana Ladra, Gonzalo Navarro y Lilian Salinas. «Compact structure for sparse undirected graphs based on a clique graph partition». En: *Information Sciences* (2021), págs. 485-499.

[6] Andrei Z Broder. «On the resemblance and containment of documents». En: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE. 1997, págs. 21-29.

[7] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian y Jingren Zhou. «Maximum biclique search at billion scale». En: *Proceedings of the VLDB Endowment* (2020).

[8] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu y Jianxin Li. «Efficient exact algorithms for maximum balanced biclique search in bipartite graphs». En: *Proceedings of the 2021 International Conference on Management of Data*. 2021, págs. 248-260.

[9] Volker Strassen. «Gaussian elimination is not optimal». En: *Numerische mathematik* (1969), págs. 354-356.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein. *Introduction to algorithms*. MIT press, 2022, págs. 76-90, 1214-1222.

[11] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz y Oded Schwartz. «Communication-optimal parallel algorithm for strassen's matrix multiplication». En: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 2012, págs. 193-204.

[12] Ali Mohades y Johannes Lederer. «Reducing Computational and Statistical Complexity in Machine Learning Through Cardinality Sparsity». En: (2023).

[13] Aydin Buluç y John R. Gilbert. «Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments». En: *SIAM Journal on Scientific Computing* (2012), págs. C170-C191.

[14] Amir Schoor. «Fast Algorithm for Sparse Matrix Multiplication». En: *Information Processing Letters* (1982), págs. 87-89.

[15] Alexandre P. Francisco, Travis Gagie, Dominik Köppl, Susana Ladra y Gonzalo Navarro. «Graph compression for adjacency-matrix multiplication». En: *SN Computer Science* (2022), pág. 193.

[16] Paolo Boldi y Sebastiano Vigna. «The webgraph framework I: compression techniques». En: *Proceedings of the 13th international conference on World Wide Web*. 2004, págs. 595-602.

[17] Paolo Ferragina, Travis Gagie, Dominik Köppl, Giovanni Manzini, Gonzalo Navarro, Manuel Striani y Francesco Tosoni. «Improving matrix-vector multiplication via lossless grammar-compressed matrices». En: *arXiv preprint arXiv:2203.14540* (2022).

[18] Diego Arroyuelo, Adrián Gómez-Brandón y Gonzalo Navarro. «Evaluating Regular Path Queries on Compressed Adjacency Matrices». En: (2023).

[19] Nieves R. Brisaboa, Susana Ladra y Gonzalo Navarro. «k2-trees for compact web graph representation». En: *International symposium on string processing and information retrieval*. Springer. 2009, págs. 18-30.

[20] Andrei Z. Broder, Moses Charikar, Alan M. Frieze y Michael Mitzenmacher. «Min-wise independent permutations». En: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, págs. 327-336.

[21] Anand Rajaraman y Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.