UNIVERSIDAD DE CONCEPCIÓN

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Multiplicación de matrices booleanas comprimidas mediante bicliques

Nicolás Araya Durán

Tesis presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar grado de Magíster en Ciencias de la Computación

Profesor Guía:

Cecilia Hernández Rivas

Diciembre 2025

Concepción, Chile

Agradecimientos

Agradezco profundamente a mi familia, en especial a mis padres, por su constante esfuerzo y sacrificio, que me permitieron llegar hasta este punto, y por enseñarme siempre la importancia de la perseverancia y de aprender de los errores. Agradezco también a mis abuelos, *Flor y René*, por su inmenso amor y apoyo. Aunque ya no estén físicamente presentes para ver este final, sé que continúan acompañándome con el mismo cariño y respaldo que me brindaron desde el comienzo. Dondequiera que estén, les agradezco todo, pues sin ellos no habría sido posible alcanzar este logro.

Agradezco asimismo a las amistades que me han acompañado a lo largo de los años y que espero sigan presentes en el tiempo. Un agradecimiento especial a *José Zagal* por su amistad, su apoyo y su ayuda durante toda esta travesía; a los profesores que me impulsaron y enriquecieron con su conocimiento para llegar hasta aquí. En particular, agradezco cordialmente a la profesora *Cecília Hernández* por su enseñanza y su constante apoyo, y al profesor *Gonzalo Navarro* por su valiosa colaboración en el desarrollo de este proyecto. Finalmente, agradezco al proyecto ANID Fondecyt Iniciación N.º 11240971 por el apoyo brindado.

Tarda en llegar, y al final hay recompensa.

Resumen

La multiplicación booleana de matrices es una operación fundamental en numerosas aplicaciones basadas en grafos, como el análisis de redes web y sociales, la recuperación de información y la evaluación de consultas sobre grafos. No obstante, las representaciones dispersas convencionales, como *CSR* y *CSC*, siguen implicando costos significativos de espacio y tiempo cuando se aplican a grafos de gran escala. En este trabajo, hemos desarrollado una representación basada en bicliques que comprime matrices booleanas mediante su descomposición en subgrafos bipartitos densos, capturando así grandes conjuntos de aristas de manera implícita. En primer lugar, mejoramos el algoritmo iterativo de extracción de bicliques de Hernández y Navarro mediante la incorporación de una adaptación de parámetros basada en percentiles, la cual ajusta dinámicamente dichos parámetros según la distribución de grados, reduciendo el tiempo de cómputo de extracción y aumentando la tasa de compresión. Luego, extendemos el algoritmo de multiplicación de Schoor para operar directamente sobre matrices codificadas con bicliques, permitiendo una forma composicional de multiplicación en la que la salida también puede representarse como una colección de bicliques. Nuestros resultados experimentales demuestran que nuestro enfoque logra reducciones en espacio y tiempo de cómputo, alcanzando aceleraciones de hasta 200× en comparación con implementaciones basadas en *CSR*/*CSC*, y una competitiva cantidad de bits por arista comparado con el k^2-tree. Estos resultados demuestran que la compresión basada en bicliques ofrece un equilibrio efectivo entre compresión y operatividad algebraica para el procesamiento de matrices booleanas a gran escala.

Índice general

Índice de figuras

Índice de cuadros

1 Introducción

Los grafos desempeñan un papel fundamental en la modelación de una amplia gama de problemas en diversas áreas, incluidas las aplicaciones científicas, los algoritmos de grafos y las bases de datos orientadas a grafos. Por ejemplo, es posible modelar redes sociales, la estructura de Internet [12], grafos web y el intercambio de genes entre conjuntos de genomas [20]. Muchas aplicaciones emplean grafos dispersos, como los grafos web y sociales [4], las bases de datos de grafos [2] y las aplicaciones bioinformáticas. Una representación común para los grafos es la matriz de adyacencia, que permite realizar operaciones algebraicas entre grafos mediante aritmética matricial. En estos contextos, el cálculo de la transpuesta, la suma de grafos, la multiplicación matriz-vector dispersa y la multiplicación matriz-matriz son operaciones fundamentales para tareas como el análisis de gramáticas libres de contexto o el procesamiento de consultas en bases de datos de grafos.

Entre las representaciones eficientes para matrices dispersas se encuentra el formato *Compressed Sparse Row* (CSR), que almacena los elementos distintos de cero junto con los índices de fila y la posición inicial de cada una. El formato *Compressed Sparse Column* (CSC) es la estructura análoga para indexar columnas. Aunque la multiplicación de matrices booleanas dispersas requiere menos cómputo que la multiplicación general de matrices, el espacio y el tiempo necesarios para manejar grafos y matrices grandes siguen siendo considerables. Para abordar este problema, la comunidad científica ha propuesto diversas soluciones a lo largo del tiempo. Una contribución destacada es el algoritmo de Schoor, que alcanza una complejidad promedio de $O(\frac{xy}{N})$ para multiplicar matrices $X \times Y$ de dimensiones $M \times N$ y $N \times L$, donde x y y son los números de elementos no nulos en las matrices X y Y, respectivamente. Recientemente, Arroyue-

lo *et al.* [2] desarrollaron una implementación compacta del algoritmo de Schoor que logra una complejidad promedio de $O(\frac{xy \log(N)}{N})$ sobre representaciones CSR/CSC. Sin embargo, debido al crecimiento continuo de los datos, resulta cada vez más importante diseñar representaciones espaciales eficientes que mantengan operaciones matriciales rápidas.

Más allá de los formatos tradicionales CSR y CSC, varios enfoques emplean esquemas de compresión de grafos, como los propuestos por Boldi y Vigna [4] o los basados en patrones de bicliques [12], para reducir los tiempos de ejecución en operaciones matriz-vector [10]. Las técnicas de compresión basadas en cliques o bicliques máximales capturan regiones altamente conectadas que aparecen con frecuencia en redes sociales y web, donde las comunidades o los clústeres de conectividad inducen una alta densidad de aristas [11]. Al descomponer los grafos en representaciones compactas basadas en estas subestructuras densas, los métodos de compresión representan las aristas de forma implícita, conservando la capacidad de reconstruir el grafo original.

En este trabajo proponemos una mejora a la compresión basada en bicliques, propuesta por Hernandez y Navarro [12] que permite aumentar la compresión y además mejora el tiempo de extracción de bicliques. Así como también un enfoque basado en bicliques que descompone matrices de adyacencia dispersas con el fin de realizar operaciones de multiplicación booleana directamente sobre la representación comprimida, mejorando tanto el uso de espacio como el tiempo de cómputo.

1.1 Hipótesis

Es posible reducir el espacio y tiempo de cómputo de operaciones de potencia y multiplicación entre matrices mediante una representación basada en bicliques

1.2 Objetivo general

El objetivo general es diseñar e implementar una estructura comprimida basada en bicliques que permita soportar las operaciones de multiplicación y potencia entre

matrices.

1.2.1 Objetivos específicos

- Analizar parámetros del extractor de bicliques y su incidencia en el tiempo y espacio de la representación reducida del grafo.
- Definir un método adaptativo de extracción de bicliques con el objetivo de mejorar la compresión alcanzada y reducir el tiempo de cómputo.
- Analizar la degradación de los bicliques extraídos a lo largo de múltiples operaciones de multiplicación o potencia. Estudiar el impacto en el espacio y tiempo de cómputo requeridos por las operaciones sobre matrices booleanas.

1.3 Metodología

El desarrollo del proyecto está basado en diseñar, implementar y evaluar experimental y teóricamente los algoritmos que se desarrollen usando grafos dirigidos.

1. **Recolección de Datos.** Los grafos fueron obtenidos de distintos repositorios: Laboratory for Web Algorithmics [4, 3] y SNAP Network Datasets [14]
2. **Implementaciones.** La implementación de todos los algoritmos se desarrollaron en `C++`, usando el estandar `C++17` y sin dependencias de bibliotecas externas a la STL[1]
3. **Experimentación**
   - Comparamos la compresión de nuestro algoritmo con los enfoques k^2-tree [6] utilizando la implementación realizada por Arroyuelo *et al.* [2], Framework WebGraph [4] y $REPAIR32$ [9]
   - Comparamos el rendimiento de la multiplicacion matriciales con otras representaciones que soportan operaciones matriciales, como es el caso del

[1] https://github.com/nicolasaraya/BicliqueMultiplication

algoritmo mejorado de Schoor [18], y k^2-tree [6] ambas implementaciones de Arroyuelo *et al.* [2].

2 Marco teórico

Este capitulo presenta algunas notaciones y definiciones fundamentales para el desarrollo de esta tesis. En particular, se abordan conceptos esenciales relacionados con grafos, matrices de adyacencia y sus representaciones en memoria. Asimismo, se describen estructuras de datos comprimidas de otros trabajos que permiten almacenar grafos dispersos de forma eficiente.

2.1 Definiciones

2.1.1 Grafos dirigidos

Sea $G = (V, E)$ un grafo dirigido y simple, donde:

1. V es un conjunto finito y no vacío de vértices.
2. E es un conjunto de pares ordenados (s, t) llamados aristas dirigidas, donde $s, t \in V$. Cada arista (s, t) indica una conexión desde el vértice s hacia el vértice t.
3. $n = |V|$ es el número de vértices del grafo.
4. $m = |E|$ es el número de aristas del grafo.

La Figura 2.1 muestra un ejemplo de grafo dirigido, mientras que la Figura 2.2 presenta su definición formal.

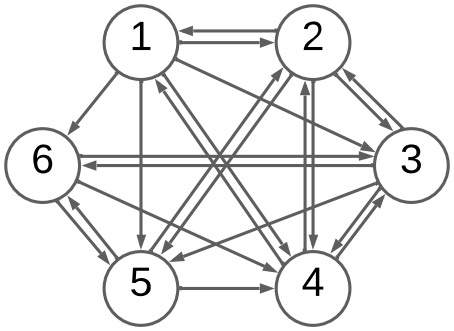

Figura 2.1: Representación gráfica del grafo dirigido G con sus vértices y aristas.

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

Figura 2.2: Definición formal del grafo dirigido G.

2.1.2 Matriz de Adyacencia

Dado $G = (V, E)$, un grafo dirigido con $n = |V|$ nodos y $m = |E|$ aristas, es posible representarlo mediante una matriz de adyacencia X de tamaño $n \times n$. En esta matriz, la fila i-ésima describe las conexiones salientes desde el vértice i. Cada entrada $a_{i,j}$ se define como:

$$
a_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E, \\ 0 & \text{en caso contrario.} \end{cases}
$$

La Figura 2.3 presenta la matriz de adyacencia correspondiente al grafo G mostrado anteriormente.

$$X = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figura 2.3: Matriz de adyacencia binaria del grafo dirigido G. Un valor 1 indica la presencia de una arista, mientras que un valor 0 indica su ausencia.

2.1.3 Compressed Sparse Row

La matriz de adyacencia es una representación directa de un grafo dirigido, pero requiere $O(n^2)$ espacio. En grafos dispersos, donde la mayoría de las entradas son cero, esta representación se vuelve poco eficiente. Para reducir el espacio utilizado, se emplean estructuras compactas que almacenan únicamente las posiciones de las entradas no nulas. Dos de las más usadas son *Compressed Sparse Row* (CSR) y *Compressed Sparse Column* (CSC). Ambas permiten representar grafos grandes utilizando una fracción del espacio requerido por la matriz completa.

Compressed Sparse Row (CSR) almacena únicamente las posiciones de los valores no nulos (igual a 1 en una matriz de adyacencia). La estructura se organiza por filas y utiliza dos arreglos principales:

1. `col_ind`: contiene los índices de las columnas donde existe una arista, en el orden en que aparecen al recorrer la matriz fila por fila.
2. `row_ptr`: almacena la posición inicial de cada fila dentro del arreglo `col_ind`.

Opcionalmente, se puede incluir un vector `row_id` si se desea almacenar únicamente las filas con entradas no nulas, útil en matrices extremadamente dispersas.

2.1.4 Compressed Sparse Column

Compressed Sparse Column (CSC) es análogo a CSR, pero almacena la información por columnas. Es especialmente útil para operaciones que requieren acceso a los

predecesores de un vértice o recorridos columna–por–columna.

CSC también utiliza dos arreglos principales:

1. `row_ind`: índices de las filas donde existe una arista.
2. `col_ptr`: posición inicial de cada columna dentro del arreglo `row_ind`.

De manera opcional, puede agregarse `col_id` para registrar únicamente las columnas con entradas no nulas.

`col_ind:`	2	3	4	5	6	1	3	4	5				
	2	4	5	6	1	2	3	2	4	6	3	4	5
`row_ptr:`	0	5	9	13	16	19	22						
`row_id:`	1	2	3	4	5	6							

Figura 2.4: Representación en CSR de X.

`row_ind:`	2	4	1	3	4	5	1	2	4	6		
	1	2	3	5	6	1	2	3	6	1	3	5
`col_ptr:`	0	2	6	10	15	19	22					
`col_id:`	1	2	3	4	5	6						

Figura 2.5: Representación en CSC de X.

La Figura 2.4 muestra la representación CSR de la matriz X presentada en la Figura 2.3, mientras que la Figura 2.5 presenta su versión en formato CSC.

2.1.5 Densidad de un grafo

La densidad de un grafo dirigido describe la proporción de aristas presentes en relación con el número máximo posible de aristas entre sus nodos. Dado un grafo dirigido G con n nodos y m aristas, su densidad se define mediante la Ecuación 2.1:

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

El término $n(n-1)$ corresponde al número máximo de aristas posibles en un grafo dirigido con n nodos, considerando que entre cada par de nodos pueden existir dos

aristas dirigidas (una en cada sentido). La densidad toma valores entre 0 y 1: una densidad igual a 0 indica ausencia total de aristas, mientras que una densidad igual a 1 implica que todas las posibles aristas dirigidas están presentes.

2.1.6 Producto cartesiano

El producto cartesiano de dos conjuntos es una operación que genera todas las combinaciones posibles entre sus elementos mediante pares ordenados. Para dos conjuntos A y B, el producto cartesiano se define como

$$A \times B = \{(a, b) \mid a \in A,\, b \in B\}.$$

Por ejemplo, si

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

entonces el producto cartesiano es

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

Este operador se utiliza posteriormente para definir aristas generadas a partir de subconjuntos de nodos en estructuras como los bicliques.

2.1.7 Compresión con bicliques

Definición 1 *Sea $G = (V, E)$ un grafo dirigido. Un* biclique *en G es un subgrafo bipartito completo definido por un par de conjuntos disjuntos de vértices $S_i, C_i \subseteq V$. Cada biclique se denota como*

$$b_i(S_i, C_i),$$

y define un subgrafo bipartito completo

$$B_i = (S_i \cup C_i,\ S_i \times C_i),$$

donde el conjunto de aristas $S_i \times C_i$ corresponde al producto cartesiano entre S_i y C_i.

El conjunto de todos los bicliques extraídos desde G se denota por

$$B = \{b_1(S_1, C_1),\, b_2(S_2, C_2),\, \ldots,\, b_N(S_N, C_N)\}.$$

La representación basada en bicliques del grafo G se expresa mediante el par (B, R), donde:

- $\bigcup_{i=1}^{N} S_i \times C_i$ *corresponde al conjunto de aristas cubiertas por los bicliques.*
- $R = E - \bigcup_{i=1}^{N} S_i \times C_i$ *es el conjunto de aristas residuales, es decir, aquellas que no pertenecen a ningún biclique.*

El grafo original puede interpretarse así como la combinación implícita de las aristas representadas por los bicliques y las aristas explícitas contenidas en R.

Procedimiento de extracción de bicliques

Una alternativa a la matriz de adyacencia es la representación mediante listas de adyacencia. Aplicando este método al grafo de la Figura 2.1 presentado anteriormente, obtenemos la representación mostrada en la Tabla 2.1.

Nodo	**Adyacentes**
1:	2, 3, 4, 5, 6
2:	1, 3, 4, 5
3:	2, 4, 5, 6
4:	1, 2, 3
5:	2, 4, 6
6:	3, 4, 5

Tabla 2.1: Representación en listas de adyacencia del grafo G.

Esta representación permite almacenar únicamente las aristas existentes, evitando el espacio desperdiciado de una matriz completa. Sin embargo, todavía es posible reducir el espacio aprovechando la similitud que existe entre las listas de adyacencia

de distintos nodos. Una forma de capturar estas similitudes es mediante la extracción de bicliques, como se describe en las siguientes secciones.

La extracción de bicliques permite identificar patrones repetitivos en las listas de adyacencia de un grafo, lo que posibilita una representación más compacta. El proceso descrito a continuación corresponde a una adaptación y generalización del método presentado por Hernández y Navarro [12], reimplementado en `C++` y extendido para permitir grafos de hasta 2^{64} nodos.

1. Estimación de similitud en listas de adyacencia

   El primer paso consiste en agrupar nodos cuyas listas de adyacencia son similares. Debido al tamaño potencialmente grande de los grafos, no es eficiente aplicar directamente funciones hash a todas las aristas. En su lugar, se utilizan funciones hash especializadas que permiten detectar patrones repetidos de manera eficiente y con bajo consumo de memoria.

   El algoritmo MinHash [7, 8] permite estimar la similitud entre conjuntos de manera eficiente mediante la construcción de *signatures*. En este contexto, su objetivo no es medir similitud exacta, sino identificar elementos comunes entre listas de adyacencia.

   Para ello, cada lista se representa mediante P *signatures*, obtenidas a partir de *k-shingles* [16]: subconjuntos de tamaño k a los que se les aplica una función hash obteniendo un *ShingleID*.

   Las P funciones hash se definen así:

   a) Se elige un número primo suficientemente grande para reducir colisiones.

   b) Para cada función hash H_i, con $i = 1, \ldots, P$, se eligen dos constantes aleatorias A_i y B_i.

   $$H_i(\text{ShingleID}) = (A_i \cdot \text{ShingleID} + B_i) \bmod prime.$$

   c) Para cada lista de adyacencia, se define un valor de signature s_i como el mínimo valor obtenido al aplicar H_i sobre todos los *ShingleID* de la lista.

El resultado final es una matriz de n filas y P columnas, denominada *matriz de signatures*, como se observa en la Tabla 2.2. El costo temporal de esta etapa es $O(P|E|)$.

Nodo	Signatures
1:	$s_1, s_2, \ldots, s_P$
2:	$s_1, s_2, \ldots, s_P$
.	.
.	.
.	.
n:	$s_1, s_2, \ldots, s_P$

Tabla 2.2: Matriz de signatures para n nodos y P funciones hash.

Para el grafo de la Figura 2.1 y usando $P = 2$, la matriz resultante se muestra en la Tabla 2.3.

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

Tabla 2.3: Matriz de signatures del grafo G con $P = 2$.

2. Clustering

   Luego, generamos los clusters en base a los signatures ordenados de cada nodo. La Figura 2.6 muestra un ejemplo de Cluster generado por esta etapa, construido en base a la similitud de las filas 1, 2 y 6, dado por el signature A. El costo temporal de esta etapa es $O(P|V| \log |V|)$.

3. Reordenamiento de aristas

   Para cada cluster, se calcula la frecuencia de aparición de cada arista en sus listas de adyacencia. Esto permite reordenar cada lista desde las aristas más frecuentes

Nodo	Signatures	Adyacentes
1:	A, F	2,3,4,5,6
2:	A, B	1,3,4,5
6:	A, B	3,4,5

Figura 2.6: Ejemplo de Cluster obtenido por coincidencia en la primera signature.

hacia las menos frecuentes, facilitando la extracción de bicliques más grandes mediante el árbol de prefijos.

Histograma Aristas	
(3):	3
(4):	3
(5):	3
(1):	1
(6):	1
(2):	1

(a) Frecuencias del Cluster 1.

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

(b) Listas reordenadas.

Figura 2.7: Proceso de reordenamiento del Cluster 1.

El costo temporal de esta etapa es $O(|E| \log |E|)$.

4. Selección de bicliques

   A partir de las listas reordenadas, se construye un árbol de prefijos. Cada nodo del árbol corresponde a un vértice vecino y almacena el conjunto de vértices desde los cuales dicha secuencia de aristas aparece. Cada camino desde la raíz hasta un nodo hoja representa un biclique potencial.

   La fracción de ganancia de un biclique se define como:

$$k = \frac{|S| \cdot |C|}{|S| + |C|}.$$

   El biclique con mayor valor de k se considera el mejor.

El proceso es iterativo: luego de extraer un biclique, sus aristas se eliminan temporalmente y se continúa buscando otros bicliques hasta que no existan candidatos que

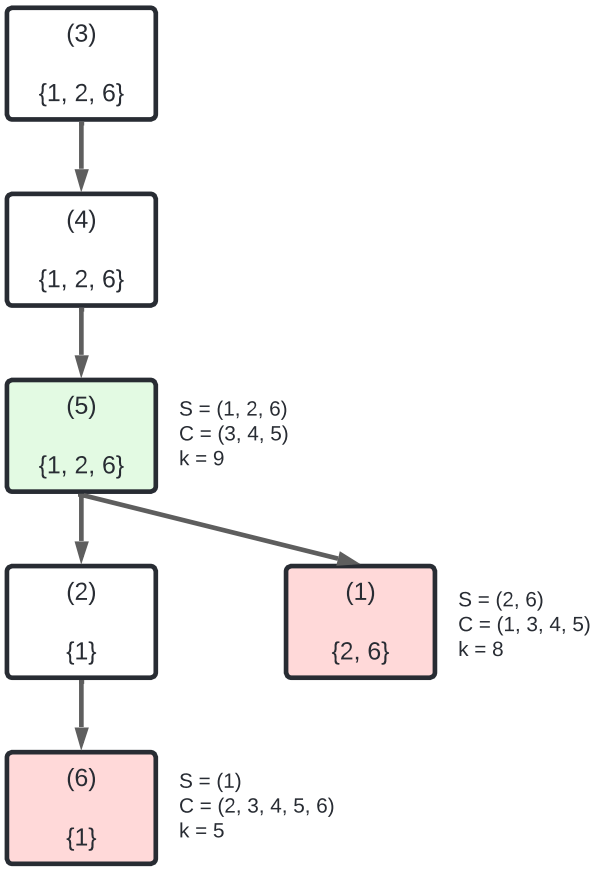

Figura 2.8: Árbol de prefijos del Cluster 1. Los nodos coloreados representan candidatos a bicliques.

superen un umbral mínimo de k.

El conjunto final de bicliques del grafo G se muestra en la Tabla 2.4. El mejor biclique se destaca visualmente.

	S	C
b_1	{1,2,6}	{3,4,5}
b_2	{1,5}	{2,6}
b_3	{4,6}	{1,2}

Tabla 2.4: Conjunto B de todos los bicliques extraídos del grafo G. El biclique b_1 se encuentra destacado por ser el mejor según la métrica de ganancia.

La matriz de adyacencia asociada a todos los bicliques extraídos se muestra en la Figura 2.9.

El residual del grafo se obtiene como $R_x = X - B_x$, cuya matriz de adyacencia se

$$B_x = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Figura 2.9: Matriz B_x que contiene todas las aristas cubiertas por los bicliques extraídos de X.

muestra en la Figura 2.10.

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 2.10: Matriz R_x correspondiente a la matriz residual del grafo luego de extraer todos los bicliques.

2.1.8 k²-tree

El k^2-*tree* es una estructura de datos compacta diseñada para representar matrices binarias dispersas mediante una partición recursiva del espacio [6]. En este trabajo utilizamos el caso estándar $k = 2$, para ello dividimos repetidamente en cuatro cuadrantes. Cada uno de estos cuadrantes se codifica con un bit 1 en caso de contener alguna arista. En caso contrario, se codifica con el bit 0.

El árbol conceptual resultante se almacena utilizando dos bitmaps:

- T: almacena todos los niveles internos del árbol, excepto el último;
- L: almacena el nivel final (las hojas), correspondientes a bloques no subdivididos.

La Figura 2.11, mustra el processo de construcción de k^2-tree para la matriz X, la cual ha sido ampliada a una matriz de 8×8

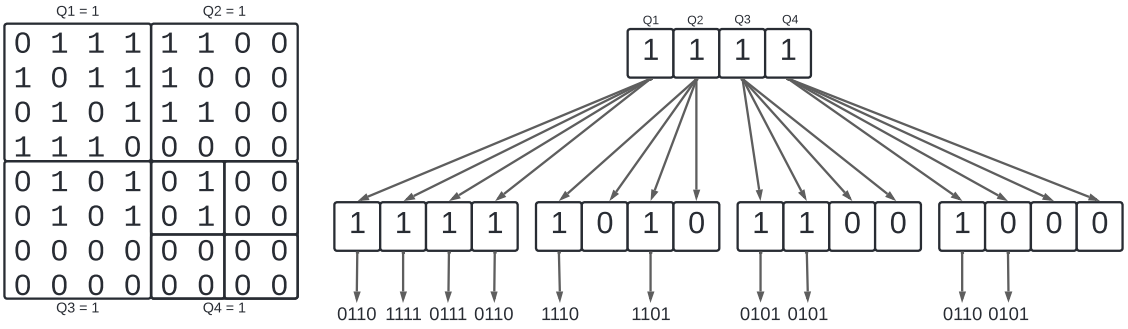

Figura 2.11: Proceso de construcción de k^2-tree

De esta forma, la matriz A queda en la siguiente forma compacta:

$$T = \texttt{1111 1111 1010 1100 1000}$$

$$L = \texttt{0110 1111 0111 0110 1110 1101 0101 0101 0110 0101}$$

Este par (T, L) codifica completamente la matriz binaria X del grafo G usando la estructura k^2-tree.

3 Estado del arte

Uno de los problemas a tratar en la representación de grafos es el gran tamaño utilizado para representarlos. Es importante mejorar el espacio y tiempo de procesamiento de operaciones en grafos de gran tamaño. Desde hace años se utilizan representaciones alternativas a las matrices de adyacencia convencionales que permiten ahorrar espacio en su representación y además permiten mantener o mejorar el tiempo de cómputo de operaciones. Pinar y Heath [15] utilizaron una forma comprimida de representar la matriz utilizando como estructura de datos *Compress Row Storage* (CRS) la cual permite ahorrar espacio aprovechando las características dispersas del grafo, dado que se limita a almacenar los valores distintos de cero. Sin embargo, esta representación está condicionada a la naturaleza dispersa del grafo, por lo cual además proponen un enfoque de re-ordenamiento de filas de la matriz utilizando heurísticas que permiten aumentar la compresibilidad del grafo. Además el hacer esto les permite reducir el tiempo de cómputo de las operaciones multiplicación matriz-vector.

Otros trabajos se han enfocado en mejorar el espacio y la eficiencia de las representaciones comprimidas, las cuales también permiten realizar consultas de manera eficiente sobre los grafos. Uno de los trabajos más relevantes es el framework WebGraph [4] y el k^2-tree [6]. El framework WebGraph por un lado, aprovecha la similitud, la localidad y el reordenamiento de las listas de adyacencia para utilizarse sobre grafos web y de redes sociales. Por otro lado, k^2-tree aprovecha la dispersión del grafo y el clustering sobre los elementos nulos de la matriz de adyacencia.

Otros métodos se basan en la búsqueda de comunidades densas dentro de los grafos, como es en el trabajo de Hernández y Navarro [12], que se enfocan en la búsqueda de bicliques apuntando a la compresión mediante una representación implícita de aris-

tas del grafo. Para la búsqueda de estos bicliques se utiliza la técnica de Min-Hash [7] para estimar la similitud de listas de adyacencia y clusterizar, tras ello se realiza la extracción de los bicliques de los clústers obtenidos. El enfoque utiliza una estructura compacta para los bicliques y el resto del grafo lo comprime utilizando la estructura k^2-tree. Más tarde, Glaria *et al.* [11], siguiendo con la misma idea de búsqueda de patrones repetitivos, propone utilizar clustering de cliques maximales para comprimir grafos no dirigidos. 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.

En cuanto a operaciones entre matrices, con el fin de mejorar el rendimiento de operaciones de multiplicación y potencia de matrices, Schoor [18] propuso un nuevo enfoque de multiplicación de matrices dispersas de gran tamaño, restringiéndose únicamente a almacenar los valores distintos de cero. De esta forma se consigue reducir el espacio utilizado por el algoritmo a $5DMN + M + N$ para una matriz de dimensión $M \times N$ y densidad D. Además, el algoritmo reduce el tiempo de computo hasta $O(\frac{x \cdot y}{N})$, con x e y la cantidad de valores distintos de cero de las matrices X e Y respectivamente. Recientemente, Arroyuelo *et al.* [2] implementaron el algoritmo propuesto por Schoor, mejorando el espacio utilizando como base las representaciones Compress Sparse Row y Compress Sparse Column, logrando reducir la complejidad de tiempo a $O(\frac{xy \log(n)}{n})$.

También se han desarrollado trabajos en los cuales se aprovechan representaciones comprimidas las cuales permiten realizar operaciones eficientes de matrices. Francisco *et al.* [10] redujo los tiempos de cómputo de la multiplicación matriz-vector usandoet dos enfoques distintos, en primer lugar utilizó el framework WebGraph, propuesto por Boldi y Vigna [4] el cual plantea un enfoque de compresión basado en la similitud entre filas y en segundo lugar el enfoque basado en bicliques de Hernández y Navarro [12]. En ambos casos, consiguió aprovechar la compresión para optimizar la operación matriz-vector del cómputo del algoritmo PageRank [5], demostrando que es posible reducir el tiempo de manera proporcional a la compresión realizada. Además el trabajo realizado por Francisco *et al.* sirvió de base para el estudio de Tosoni [19] en el cual se establece que

utilizar formatos de compresión de matrices ofrecen distintas compensaciones entre el uso de espacio, el tiempo de ejecución y el consumo de energía. En particular, al emplear el formato de compresión adecuado, puede reducir significativamente el consumo de energía.

Ferragina *et al.* [9] siguiendo con la idea de utilizar representaciones comprimidas para realizar operaciones eficientes entre matrices, propone un enfoque distinto al visto por Francisco *et al.* [10] para la multiplicación matriz-vector, centrándose en la compresión basada en gramática, sin enfatizar como tal en la naturaleza densa o dispersa del grafo. Para ello, desarrollaron una variante del *Compress Sparse Row*, denominada *Compress Sparse Row-Value*, similar a algoritmos de compresión basadas en gramática como RePair [13]. La representación propuesta 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, los autores recomiendan considerar el reordenamiento de las columnas con el fin de potenciar la compresión de gramáticas y facilitar la identificación de columnas similares.

De manera similar, Alves *et al.* [1] propusieron un nuevo formato con el fin de mejorar la multiplicación de matrices binarias utilizadas en el contexto de Redes Neuronales de Grafos (GNN). Este formato llamado *Compressed Binary Matrix* permite reducir el número de operaciones necesarias para realizar la multiplicación en comparación a otros formatos de compresión al representar unicamente las diferencias entre filas de la matriz de adyacencia. Esto disminuye el esfuerzo computacional y permite multiplicaciones de matrices más rápidas, lo que acelera tanto las etapas de entrenamiento como de inferencia en modelos GNN.

4 Identificación y extracción adaptativa de bicliques

4.1 Definición del problema

La implementación original de Hernández y Navarro [12] extrae bicliques de manera iterativa utilizando distintos parámetros, tales como el tamaño mínimo de los clusters, el tamaño mínimo $S \times C$ de los bicliques a descubrir, el número máximo de iteraciones permitido y el tamaño mínimo de nodos adyacentes (`minAdyNodes`), el cual permite restringir a un mínimo de nodos adyacentes para omitir nodos con poca cantidad de vecinos. Sin embargo, al procesar grafos de gran escala, estos parámetros fijos pueden conducir a tiempos de ejecución elevados, y al no conocer de antemano las características del grafo, la elección de parámetros tiende a ser arbitraria y no garantiza buenos resultados.

4.2 Extracción con parámetros adaptativos

Con el fin de mitigar este problema, proponemos un esquema de ajuste adaptativo de parámetros basado en la distribución de grados del grafo de entrada y en estadísticas recolectadas durante las iteraciones de extracción. Este ajuste dinámico permite reducir el tiempo de cómputo y mejorar la eficiencia de la compresión, favoreciendo la detección de bicliques más grandes.

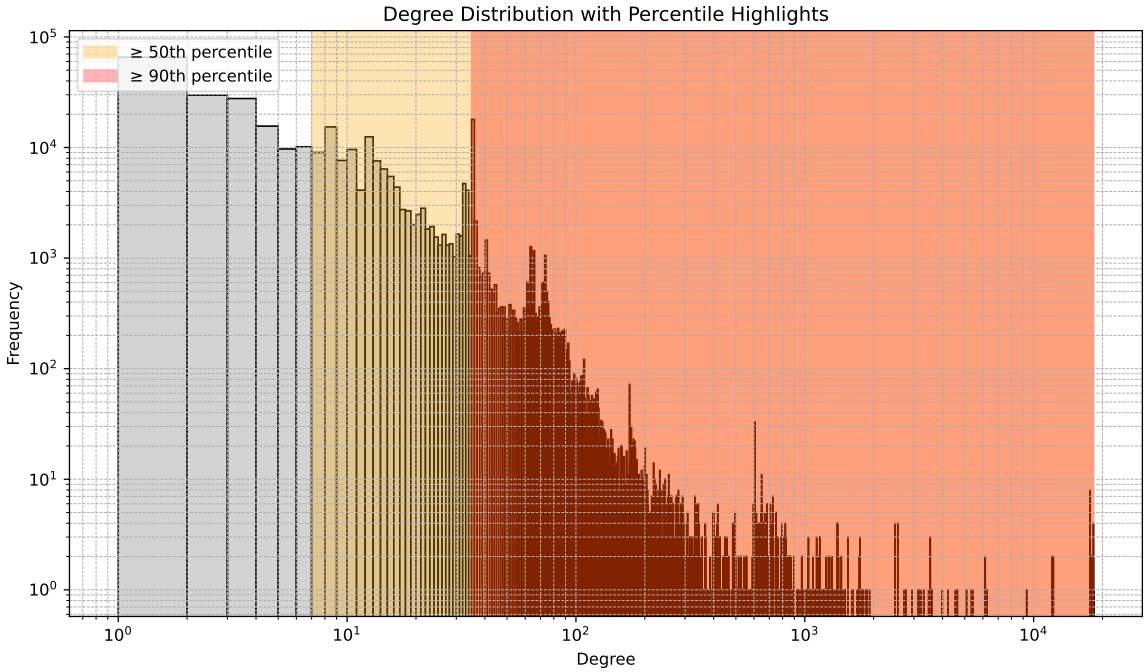

Figura 4.1: Distribución de grados de salida del grafo *cnr-2000-hc*, agrupada por percentiles. Las regiones de mayor densidad corresponden a nodos potencialmente relevantes para la extracción de bicliques.

Un proceso eficaz para priorizar la detección de bicliques con una gran cantidad de aristas consiste en concentrarse inicialmente en los nodos con mayor grado y descartar temporalmente aquellos con pocos vecinos. Esto reduce la cantidad de clusters y acelera cada etapa del proceso. Una forma práctica de aplicar este filtrado es mediante umbrales definidos por percentiles de la distribución de grados. Por ejemplo, establecer como umbral el percentil 90 restringe la búsqueda al 10 % de los nodos más conectados.

La Figura 4.1 muestra la distribución de grados del grafo *cnr-2000-hc*[4, 3], el cual se describe en la Tabla 6.1, segmentada por percentiles. Las zonas destacadas permiten identificar rangos de nodos con alta conectividad que son candidatos naturales para la extracción de bicliques de tamaño significativo. Esta información guía la parametrización inicial y evita explorar regiones del grafo con escasa contribución potencial.

Algorithm 1 Proceso de extracción adaptativa de bicliques

Require: Grafo G

Ensure: Grafo residual G', conjunto de bicliques B_G

Inicializar: `unordered_map<int,int>distribution`

▷ Etapa 1: Construcción de la distribución de grados

for all nodo $v \in G$ **do**

    `distribution[grado(`v`)]` $\leftarrow$ `distribution[grado(`v`)]` $+ 1$

end for

▷ Etapa 2: Inicialización de parámetros

`minAdyNodes` $\leftarrow$ `getPercentile`(90)

`currIt` $\leftarrow 0$

`totalBicliques` $\leftarrow 0$

`threshold` $\leftarrow 0{,}8$

`maxIt` $\leftarrow 10$

$B_G \leftarrow$ *vector vaco*

▷ Etapa 3: Extracción iterativa

while `currIt` $<$ `maxIt` **do**

    `signatures` $\leftarrow$ `computeMinHashes(minAdyNodes)`

    `clusters` $\leftarrow$ `clusterize(signatures)`

    `bicliquesExtracted` $\leftarrow$ `extractBicliques(clusters)`

    $B_G \leftarrow B_G +$ `bicliquesExtracted`

▷ Etapa 4: Ajuste adaptativo

    `avgBicliques` $\leftarrow |B_G| / \max(1,$ `currIt`$)$

    if $|$`bicliquesExtracted`$| <$ `avgBicliques` $\cdot$ `threshold` **then**

        `minAdyNodes` $\leftarrow \lfloor$`minAdyNodes` $\cdot$ `threshold`$\rfloor$

    end if

    `currIt` $\leftarrow$ `currIt` $+ 1$

end while

El Algoritmo 1 resume el proceso propuesto. El procedimiento comienza con la construcción de la distribución de grados, la cual sirve para estimar el parámetro inicial `minAdyNodes`. Este valor corresponde al percentil 90, de modo que solo se consideren inicialmente los nodos más conectados, donde es más probable encontrar bicliques grandes.

Posteriormente se ejecuta el ciclo iterativo de extracción, siguiendo la metodología descrita en las secciones previas. Tras cada iteración se evalúa la productividad del proceso comparando el número de bicliques extraídas con el promedio histórico. Si la productividad disminuye más de lo permitido por el umbral `threshold`, es decir si es que la cantidad de bicliques encontrados es inferior al promedio de bicliques encontrados, con una variación del 20 %, se reduce el valor de `minAdyNodes`, permitiendo

incorporar nodos con grados inferiores en las iteraciones siguientes.

Este mecanismo implementa una estrategia de búsqueda segmentada que prioriza regiones densas del grafo, reduce significativamente el costo computacional y mejora la calidad de las bicliques obtenidas al mantener alta la coherencia estructural dentro de cada cluster.

5 Multiplicación de matrices usando bicliques

5.1 Definición del problema

En esta sección abordamos el problema de cómo mejorar la multiplicación entre matrices de adyacencia utilizando la descomposición basada en bicliques presentada en el capítulo anterior.

Recordemos que, para un grafo dirigido G_x, su matriz de adyacencia booleana X puede expresarse como:

$$X = R_x + B_x,$$

donde:

- B_x es la matriz de adyacencia asociada al conjunto de bicliques extraído del grafo,
- R_x es la matriz residual, que contiene únicamente las aristas no cubiertas por bicliques.

Esta descomposición permite reescribir distintas operaciones matriciales separando los aportes del residual y de los bicliques.

$$X \times Y = (R_x \times R_y) + (R_x \times B_y) + (B_x \times R_y) + (B_x \times B_y). \tag{5.1}$$

La Ecuación 5.1 muestra que la operatoria de la multiplicación de las matrices X e Y puede descomponerse en cuatro términos independientes, cada uno asociado a una combinación distinta entre residual y bicliques.

Esta formulación separa la operatoria completa en cuatro componentes, lo que permite diseñar algoritmos especializados para cada producto parcial.

5.2 Operaciones

A continuación para describir las operaciones, utilizaremos el grafo G presentado la Figura 2.1, el cual fue utilizado anteriormente en los ejemplos de extracción de bicliques.

Para complementar el desarrollo, presentamos la matriz Y que se muestra en la Figura 5.1, la cual cual cuenta con los bicliques presentados en la Tabla 5.1, generando a su vez la matriz R_y presentada en la Figura 5.2 la cual se utilizará en los ejemplos a continuación.

$$Y = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figura 5.1: Matriz de adyacencia booleana Y correspondiente al grafo G_y.

	S	C
b_1	{1,2,3}	{5,6}
b_2	{5,6}	{2,3,4}
b_3	{4,6}	{1,5}

Tabla 5.1: Conjunto B_y de todos los bicliques extraídos de la matriz Y.

5.2.1 Multiplicación de matrices

La operación *Matriz* $\times$ *Matriz* utilizada para computar $R_x \times R_y$ se basa en el algoritmo de Schoor [18]. Este método se diseñó específicamente para matrices esparsas, y su eficiencia proviene de evitar el recorrido completo de filas y columnas, operando únicamente sobre las posiciones no nulas.

$$R_y = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 5.2: Matriz de adyacencia booleana R_y correspondiente a la matriz residual de Y.

Para aplicar el algoritmo de Schoor, la matriz residual R_x debe representarse en formato *Compressed Sparse Column* (CSC), mientras que R_y debe presentarse en *Compressed Sparse Row* (CSR). CSC permite acceder rápidamente a los predecesores de un nodo, mientras que CSR entrega sus sucesores. La multiplicación se expresa como:

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

(a) $CSC(R_x)$

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

(b) $CSR(R_y)$

Figura 5.3: Representaciones de las matrices R_x y R_y en *CSC* y *CSR* respectivamente

El algoritmo procede en cuatro pasos:

1. **Intersección de índices.** Se intersectan los valores del vector `colId` de la representación $CSC(R_x)$ con los valores del vector `rowId` de la representación $CSR(R_y)$. Cada elemento común t identifica un índice intermedio que contribuye a la multiplicación.

`row_ind:`	2	3	4	3	5	3	3
`col_ptr:`	0	1	2	3	5	6	7
`col_id:`	**1**	2	**3**	**4**	**5**	6	

(a) $CSC(R_x)$

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

(b) $CSR(R_y)$

Figura 5.4: Intersecciones entre R_x y R_y

En el ejemplo de la Figura 5.4b, la intersección resulta en el conjunto $\{1, 3, 4, 5\}$, cuyos elementos se destacan con distintos colores.

2. **Obtención de filas y columnas incidentes.** Para cada valor t en la intersección, el algoritmo obtiene:

   - el conjunto de filas asociado a la columna t de X, accediendo a `colPtr` y `rowIdx` de $CSC(X)$;
   - el conjunto de columnas asociado a la fila t de Y, accediendo a `rowPtr` y `colIdx` de $CSR(Y)$.

   En el ejemplo, se obtienen las siguientes intersecciones:

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

   Cada una de estas intersecciones define un producto cartesiano pendiente de procesar.

3. **Min-Heap de filas (`HeapByRow`).** Todas las intersecciones obtenidas se insertan en un *min-heap*, denominado `HeapByRow`, el cual se ordena según el primer elemento del conjunto de filas. Esto garantiza que las intersecciones que comparten la misma fila se procesen de forma consecutiva.

   Para el ejemplo considerado, el contenido inicial de `HeapByRow` es:

HeapByRow		
(2)	×	(4)
(3)	×	(1)
(3,5)	×	(2,3)
(4)	×	(1,4)

Figura 5.5: Min-Heap de filas con las intersecciones

4. **Min-Heap de columnas (`HeapByCol`) y productos cartesianos.** El algoritmo extrae iterativamente elementos de `HeapByRow`. Para cada fila procesada, se crea un *min-heap* auxiliar `HeapByCol`, el cual ordena los pares resultantes según la columna.

   En la primera iteración, se procesa la fila 2, computando el producto cartesiano $(2) \times (4) = (2, 4)$, el cual se inserta en `HeapByCol(2)`.

HeapByCol(2)
(2,4)

Figura 5.6: Min-Heap de columnas para la fila 2

   En las siguientes iteraciones se obtiene la fila 3. Como esta fila es distinta de la anterior, `HeapByCol(2)` se vacía y sus elementos se incorporan a la representación final en formato *CSR*. Luego se computan los productos:

$$(3) \times (1) = (3, 1), \quad (3, 5) \times (2, 3) = (3, 2), (3, 3),$$

   los cuales se insertan en `HeapByCol(3)`.

HeapByCol(3)
(3,1)
(3,2)
(3,3)

Figura 5.7: Min-Heap de columnas para la fila 3

   Dado que la intersección $(3, 5) \times (2, 3)$ aún tiene filas pendientes, se reinserta en `HeapByRow` como $(5) \times (2, 3)$.

HeapByRow		
(4)	×	(1,4)
(5)	×	(2,3)

Figura 5.8: Min-Heap de filas con las intersecciones tras reinsertar

5. **Construcción de la matriz resultante.** Cada vez que se detecta un cambio de fila al extraer un elemento de `HeapByRow`, el heap `HeapByCol` asociado a la fila previa se vacía. A medida que sus elementos son extraídos, se construye incrementalmente la matriz resultante en formato *CSR*.

   El proceso continúa hasta que ambos heaps quedan vacíos, completando así la construcción de $CSR(X \times Y)$.

Una vez se procesen todos los elementos de `HeapByRow` y `HeapByCol`, se obtiene la matriz en formato CSR de $R_x \times R_y$, como se muestra en la Figura 5.9

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

Figura 5.9: Representación en *Compress Sparse Row* de la matriz resultante, $CSR(R_x \times R_y)$

$$R_x \times R_y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 5.10: Matriz resultante $R_x \times R_y$ obtenida a partir de los pares únicos de los *min-heap*.

La Figura 5.10 muestra $R_x \times R_y$ en representación matriz de adyacencia.

5.2.2 Multiplicación de matriz por biclique

Al igual que en el caso anterior, esta operación utiliza una representación basada en *Compressed Sparse Column* (CSC) para representar la matriz R_x.

Cada biclique de Y presentados en la Tabla 5.1 lo representaremos de manera individual con sus componentes S y C, como se observa en la Figura 5.11.

b_1

S:	1	2	3
C:	5	6	

b_2

S:	5	6	
C:	2	3	4

b_3

S:	4	6
C:	1	5

Figura 5.11: Bicliques de Y.

Para computar $R_x \times B_y$ se procede de la siguiente manera:

1. **Intersección de índices.** Se intersecta el vector `col_id` de $CSC(R_x)$ con los vectores `S` de cada b_i.

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

Figura 5.12: Representación en *Compress Sparse Column*, $CSC(R_x)$.

b_1

S:	1	2	3
C:	5	6	

b_2

S:	5	6	
C:	2	3	4

b_3

S:	4	6
C:	1	5

2. **Obtención de filas y columnas incidentes.** Por cada intersección obtener el contenido de la columna de $CSC(R_x)$ y el contenido de C de cada biclique b_i

   Para b_1:

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

Para b_2:

$$\mathbf{5} \times \mathbf{5}: \quad (3) \quad \times \quad (2,3,4)$$
$$\mathbf{6} \times \mathbf{6}: \quad (3) \quad \times \quad (2,3,4)$$

Para b_3:

$$\mathbf{4} \times \mathbf{4}: \quad (3,5) \quad \times \quad (1,\ 5)$$
$$\mathbf{6} \times \mathbf{6}: \quad (3) \quad \times \quad (1,\ 5)$$

3. **Concatenación.** Por cada bicliques obtendremos bicliques nuevos. Para ello debemos concatenar los elementos obtenidos de $CSC(R_x)$, ordenarlos y eliminar repetidos, estos elementos seran parte del vector S del nuevo biclique. El vector C, simplemente se heredará del biclique original.

   Para b_1:

   b_1

S:	2	3	5
C:	5	6	

   Para b_2:

   b_2

S:	3		
C:	2	3	4

Para b_3:

b_3

S:	3	5
C:	1	5

5.2.3 Multiplicación de biclique por biclique

	S	C
b_{x1}	{1,2,6}	{3,4,5}
b_{x2}	{1,5}	{2,6}
b_{x3}	{4,6}	{1,2}

Tabla 5.2: Conjunto B_x de todos los bicliques extraídos de la matriz X

	S	C
b_{y1}	{1,2,3}	{5,6}
b_{y2}	{5,6}	{2,3,4}
b_{y3}	{4,6}	{1,5}

Tabla 5.3: Conjunto B_y de todos los bicliques extraídos de la matriz Y.

Para realizar la operación de multiplicación biclique por biclique, utilizamos los bicliques B_x y B_y, presentados anteriormente en las Figuras 2.4 y 5.1 respectivamente y que se muestran en las Figuras 5.2 y 5.3

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, como se muestra en la Figura 5.14.

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

1. Para cada biclique b_{x_i}, se busca cada elemento de C_{x_i} en la tabla $marks(B_y)$.

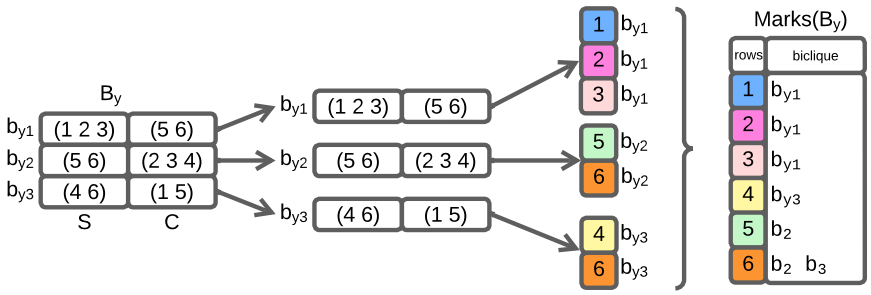

Figura 5.14: Marks de B_y, cada nodo a la izquierda pertenece a uno o más bicliques de la columna de la derecha.

b_{x1}

S:	1	2	6
C:	**3**	**4**	**5**

b_{x2}

S:	1	5
C:	**2**	**6**

b_{x3}

S:	4	6
C:	**1**	**2**

Figura 5.15: Bicliques de X, en negritas se encuentran los elementos que son encontrados en $Marks(B_y)$.

2. Si el elemento se encuentra en $marks(B_y)$, se obtiene su contenido asociado (los conjuntos C_{y_j} correspondientes). Nótese que esto implica que el biclique (S_{x_i}, C_{y_j}) pertenece al resultado.

b_{x1}

S:	1	2	6
C:	c_{y1}	c_{y3}	c_{y2}

b_{x2}

S:	1	5
C:	c_{y2}	c_{y3} c_{y2}

b_{x3}

S:	4	6
C:	c_{y1}	c_{y1}

Figura 5.16: Bicliques de X, referenciando a los C_i de B_y.

3. Así, para cada biclique b_{x_i} que presenta al menos una intersección, se genera un nuevo biclique (S_{x_i}, C), donde la secuencia C se forma combinando los valores C_{y_j} obtenidos desde $marks(B_y)$.

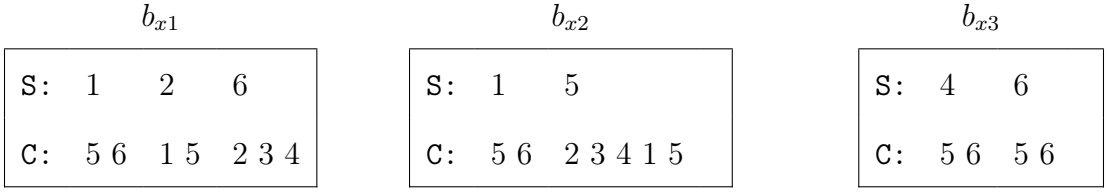

Figura 5.17: Bicliques de X, reemplazando el contenido de C_x por los elementos de C_y.

4. Finalmente, se eliminan los elementos duplicados en el nuevo conjunto C de cada biclique generado.

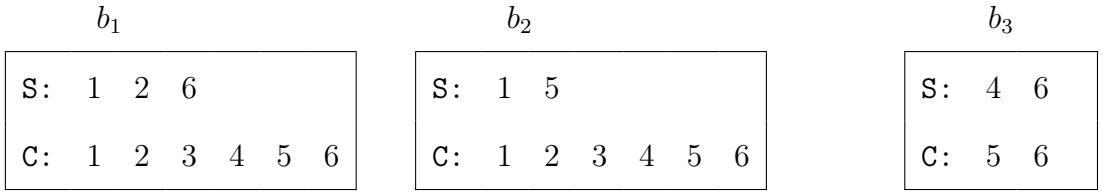

Figura 5.18: Nuevo conjunto de bicliques, tras la eliminación de elementos duplicados

De esta forma, obtenemos un resultado en forma de biclique, como se muestra en la Tabla 5.4.

	S	C
b_1	{1,2,6}	{1, 2, 3, 4, 5, 6}
b_2	{1,5}	{1, 2, 3, 4, 5, 6}
b_3	{4,6}	{5, 6}

Tabla 5.4: Conjunto de bicliques obtenidos al operar $B_x \times B_y$.

5.2.4 Multiplicación de biclique por matriz

Esta operación es un caso similar al anterior, para ello se toman los bicliques B_x de la matriz X, los cuales se muestran en 5.19 y la representación en *Compress Sparse Row* de R_x que se presenta en la Figura 5.20.

Luego para computar $B_x \times R_y$ se debe operar de la siguiente manera:

1. Se calcula la intersección entre el arreglo C_{x_i} de cada biclique b_{x_i} y el arreglo `rowId` de $CSR(R_y)$.

b_{x1}

S:	1	2	6
C:	3	4	5

b_{x2}

S:	1	5
C:	2	6

b_{x3}

S:	4	6
C:	1	2

Figura 5.19: Bicliques de X.

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

Figura 5.20: $CSR(R_y)$

Figura 5.21: Representacion de R_y en CSR

b_{x1}

S:	1	2	6
C:	3	4	5

b_{x2}

S:	1	5
C:	2	6

b_{x3}

S:	4	6
C:	1	2

Figura 5.22: Elementos intersectados en los Bicliques de X.

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

Figura 5.23: $CSR(R_y)$

Figura 5.24: Elementos intersectados de $CSR(R_y)$

2. Para cada elemento intersectado $\texttt{rowId}[k] \in C_{x_i}$, se recupera el contenido de columnas desde $\texttt{colIdx}[\texttt{rowPtr}[k]..\texttt{rowPtr}[k+1]-1]$ de $CSR(R_y)$, junto con el conjunto de filas S_{x_i} del biclique b_{x_i}.

   Para b_1:

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

Para b_2: No hay intersección entre `row_id` y C_x.

Para b_3:

$$\mathbf{1} \times \mathbf{1}: \quad (4,6) \quad \times \quad (4)$$

3. El biclique de salida es (S_{x_i}, C). Para construir C, se concatenan en una secuencia todos los valores de columnas válidos obtenidos desde `colIdx` de $CSR(R_y)$.

   Para b_1:

   b_1

`S:`	1	2	6		
`C:`	1	4	2	3	1

   Para b_3:

   b_3

`S:`	4	6
`C:`	4	

4. Finalmente, se eliminan los elementos duplicados en C para los bicliques generados.

Figura 5.25: Bicliques resultantes tras la operación $B_x \times R_y$.

De esta forma, obtenemos un resultado en forma de biclique, como se muestra en el Cuadro 5.5.

	S	C
b_1	{1,2,6}	{1, 2, 3, 4}
b_3	{4,6}	{4}

Tabla 5.5: Conjunto de bicliques obtenidos de $B_x \times R_y$.

5.3 Representación del Resultado de la Multiplicación

Una de las principales ventajas de la multiplicación matricial basada en bicliques es que el resultado de la operación puede representarse en el mismo formato comprimido que las matrices de entrada. En particular, asumimos que las matrices de entrada X e Y están representadas mediante pares (B_x, R_x) y (B_y, R_y), donde B corresponde al conjunto de bicliques y R al grafo residual en formato *CSR*.

Como se mostró en la sección anterior, las operaciones $(R_x \times B_y)$, $(B_x \times R_y)$ y $(B_x \times B_y)$ producen resultados que pueden expresarse naturalmente como conjuntos de bicliques, mientras que la operación $(R_x \times R_y)$ genera un resultado que se representa de forma explícita en formato *CSR*. De este modo, el resultado de la multiplicación $Z = X \times Y$ puede expresarse como el par (B_z, R_z), definido como:

$$\begin{aligned} B_z &= R_x \times B_y \cup B_x \times R_y \cup B_x \times B_y, \\ R_z &= R_x \times R_y. \end{aligned}$$

Cabe notar que, bajo esta representación, el conjunto residual R_z puede contener aristas que también estén implícitamente representadas en B_z. Sin embargo, esto no afecta la corrección del resultado: la posible redundancia se elimina únicamente cuando la representación (B_z, R_z) se materializa en formato *CSR*, por ejemplo, durante una operación de recuperación o combinación final.

Con el fin de reducir el espacio utilizado, y dado que el costo de construir $CSC(R_x)$ a partir de $CSR(R_x)$ es despreciable en comparación con el costo de la multiplicación, almacenamos únicamente el residual en formato *CSR*. La representación *CSC* se genera bajo demanda durante la multiplicación, cuando es necesaria. Este enfoque contrasta con trabajos previos, como el de Arroyuelo et al. [2], donde ambas representaciones se almacenan explícitamente.

Una propiedad fundamental de esta representación es su *composicionalidad*: el resultado de la multiplicación tiene exactamente el mismo formato que las matrices de entrada. Esto permite encadenar operaciones de manera directa, dando lugar a un álgebra matricial en la que las matrices se mantienen siempre en forma de bicliques más residual. En este marco, operaciones adicionales como la transposición o la suma pueden incorporarse de forma natural y eficiente.

Recompresión Un posible inconveniente de esta estrategia es que la representación (B_z, R_z) obtenida tras la multiplicación no necesariamente corresponde a una descomposición óptima en términos de compresión. Por ejemplo, bicliques degenerados, como aquellos con $|S_i| = 1$ o $|C_i| = 1$, pueden ser más eficientemente integrados en el residual.

Para mejorar la calidad de la representación comprimida del resultado Z, se puede aplicar un proceso de *recompresión*. Este consiste en recuperar la matriz $CSR(\mathrm{Z})$ a partir de (B_z, R_z) y, posteriormente, extraer nuevamente bicliques siguiendo el procedimiento descrito en la Sección 4.1. En la Sección 4.2 se muestra cómo realizar esta etapa de forma más eficiente y con mejores resultados.

La recompresión puede considerarse como una operación adicional dentro del álgebra matricial, aplicable en cualquier momento en que se desee mejorar la calidad de la compresión. Una representación más adecuada en términos de bicliques no solo

reduce el espacio ocupado, sino que también acelera operaciones posteriores sobre la matriz resultante. Para aplicar este proceso, es necesario convertir previamente la representación basada en bicliques a formato *CSR*, procedimiento que se describe en la Sección 5.4.

Bicliques como mecanismo de aceleración Finalmente, las mejoras en el tiempo de extracción de bicliques presentadas en la Sección 4.2 permiten utilizar bicliques no solo como mecanismo de compresión, sino también exclusivamente como una herramienta para acelerar la multiplicación matricial. En este escenario, se asume que las matrices de entrada y salida se representan únicamente en formato *CSR*.

En este caso, se extraen bicliques de las matrices X e Y antes de realizar la multiplicación, y tras completar la operación, el resultado se convierte directamente a formato *CSR* utilizando el procedimiento descrito en la Sección 5.4.

5.4 Recuperar matriz a partir de bicliques

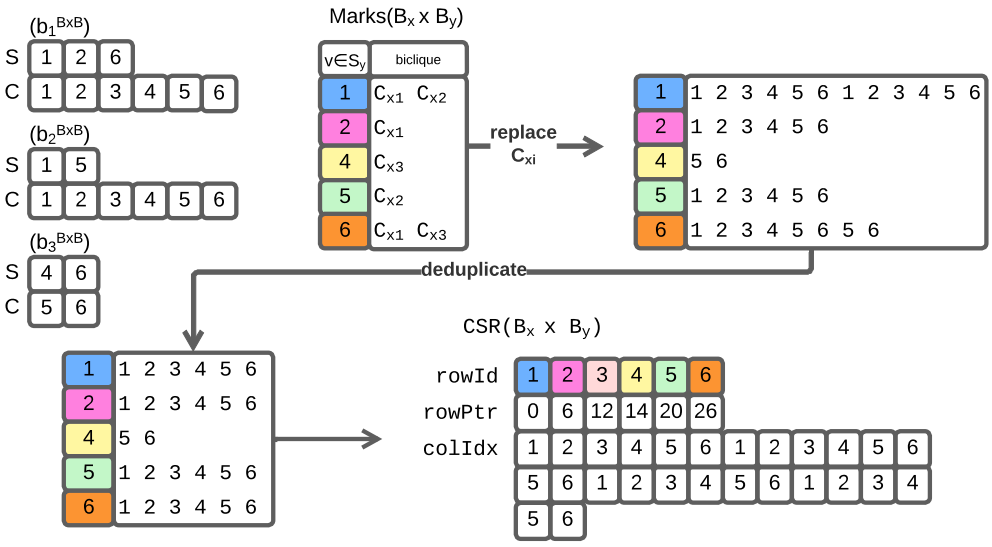

Figura 5.26: Proceso de convertir la salida de $B_x \times B_y$ al formato *CSR*.

Para obtener la matriz en formato *CSR* a partir de los bicliques computados en las operaciones descritas anteriormente, se vuelve a utilizar la tabla *marks*. A partir de ella,

se construye la representación en *CSR* del siguiente modo: se concatenan los elementos de los componentes C_{z_i} asociados a cada fila v en *marks*, se eliminan los duplicados de dicha concatenación, y el resultado corresponde a los elementos del arreglo `colIdx` asociados a $v \in$ `rowId`.

La Figura 5.26 ilustra este proceso para el componente $B_x \times B_y$; de manera análoga, los bicliques generados por $B_x \times R_y$ y $R_x \times B_y$ se combinan utilizando el mismo procedimiento. Finalmente, el resultado se fusiona (eliminando duplicados) con la representación en *CSR* del componente restante, $R_x \times R_y$.

6 Resultados

Implementamos la propuesta de Arroyuelo et al. [2] en `C++`, extendiendo el algoritmo original para incluir las operaciones basadas en bicliques. Para la evaluación, medimos el tiempo de ejecución y el uso de memoria, comparando nuestros resultados con el algoritmo de Schoor mejorado e implementado por Arroyuelo et al. [2], el cual definimos como *Baseline*. También incluimos una comparación con la implementación del k^2-tree desarrollada por los mismos autores.

Los conjuntos de datos utilizados son de acceso público y provienen de diversas fuentes reconocidas, como el *Laboratory of Web Algorithmics* (LAW), recopilados y utilizados por Boldi y Vigna [4]; la plataforma *Stanford Network Analysis Platform* (SNAP) [14]; y el repositorio *Network Repository* [17]. La Tabla 6.1 muestra las estadísticas principales de los grafos empleados, incluyendo el número de vértices $|V(G)|$, el número de aristas $|E(G)|$ y el grado medio de los nodos, denotado como Average $dg(v)$.

Todas las pruebas se ejecutaron en implementaciones secuenciales sobre un servidor Linux con procesador *AMD EPYC 9354*, 256 GB de memoria RAM y 32 núcleos físicos con dos hilos por núcleo.

Grafo	**|V(G)|**	**|E(G)|**	**Avg dg(v)**
web-Stanford	281,903	2,312,497	8.20
cnr-2000-hc	325,557	3,126,152	9.87
wikipedia_link_lmo	52,214	3,623,678	64.40
web-Google	875,713	5,105,039	5.83
coPapersCiteseer	434,102	32,073,436	73.88
indochina-2011	7,414,866	194,109,311	26.17
arabic-2005	22,744,080	639,999,458	28.13

Tabla 6.1: Estadísticas principales de los grafos utilizados en los experimentos, incluyendo el número de vértices, número de aristas y grado medio.

6.1 Extracción de Bicliques y Compresión

Primero comparamos la compresión en espacio y el rendimiento temporal entre las versiones adaptativa (Sección 4.2) y no adaptativa (la implementación original [12], véase la Sección 4.1) del algoritmo de extracción de bicliques. Para realizar esta comparación, definimos el *ratio de compresión* como el porcentaje de espacio ahorrado mediante la compresión (mientras mayor, mejor):

$$Ratio = 100 \left(1 - \frac{(\sum_i |S_i| + |C_i|) + |R|}{|E|}\right).$$

Dataset	**Ratio (%)**		**BicEx (s)**		**bpe(B+R)**	
	No adapt.	Adapt.	No adapt.	Adapt.	No adapt.	Adapt.
web-Stanford	21.64	**52.35**	2.54	**2.40**	32.46	**23.21**
cnr-2000-hc	41.32	**59.60**	2.71	**1.17**	23.61	**17.88**
wikipedia-link-lmo	**67.15**	65.95	2.95	**1.35**	**11.43**	11.83
web-Google	2.01	**40.98**	**8.40**	13.58	40.59	**28.88**
coPapersCiteseer	49.94	**64.47**	47.38	**13.69**	16.89	**12.29**
indochina-2011	73.96	**78.33**	173.12	**89.73**	10.18	**8.85**
arabic-2005-hc	70.75	**74.13**	951.66	**441.95**	11.17	**10.18**

Tabla 6.2: Comparación entre la extracción de bicliques con parámetros adaptativos y no adaptativos. La columna *BicEx* muestra el tiempo de ejecución en segundos del proceso de extracción, mientras que *bpe* indica los bits por arista de las representaciones resultantes. Los mejores valores se destacan en negrita.

La Tabla 6.2 compara la compresión obtenida y los tiempos de ejecución del método de extracción de bicliques adaptativo frente a la alternativa no adaptativa, la cual utiliza parámetros fijos y un número constante de iteraciones. Observamos que la versión adaptativa supera a la no adaptativa tanto en ratio de compresión como en velocidad en casi todos los casos. En promedio, la versión adaptativa requiere aproximadamente el 45 % del tiempo de la versión no adaptativa, llegando incluso al 29 % en algunos conjuntos de datos. Solo en un caso resulta más lenta, alcanzando el 162 % del tiempo; sin embargo, en dicho escenario la versión no adaptativa prácticamente no logra comprimir, con un ratio de solo 2 %, frente al 41 % obtenido por la versión adaptativa.

En términos de espacio, la representación comprimida obtenida mediante el mé-

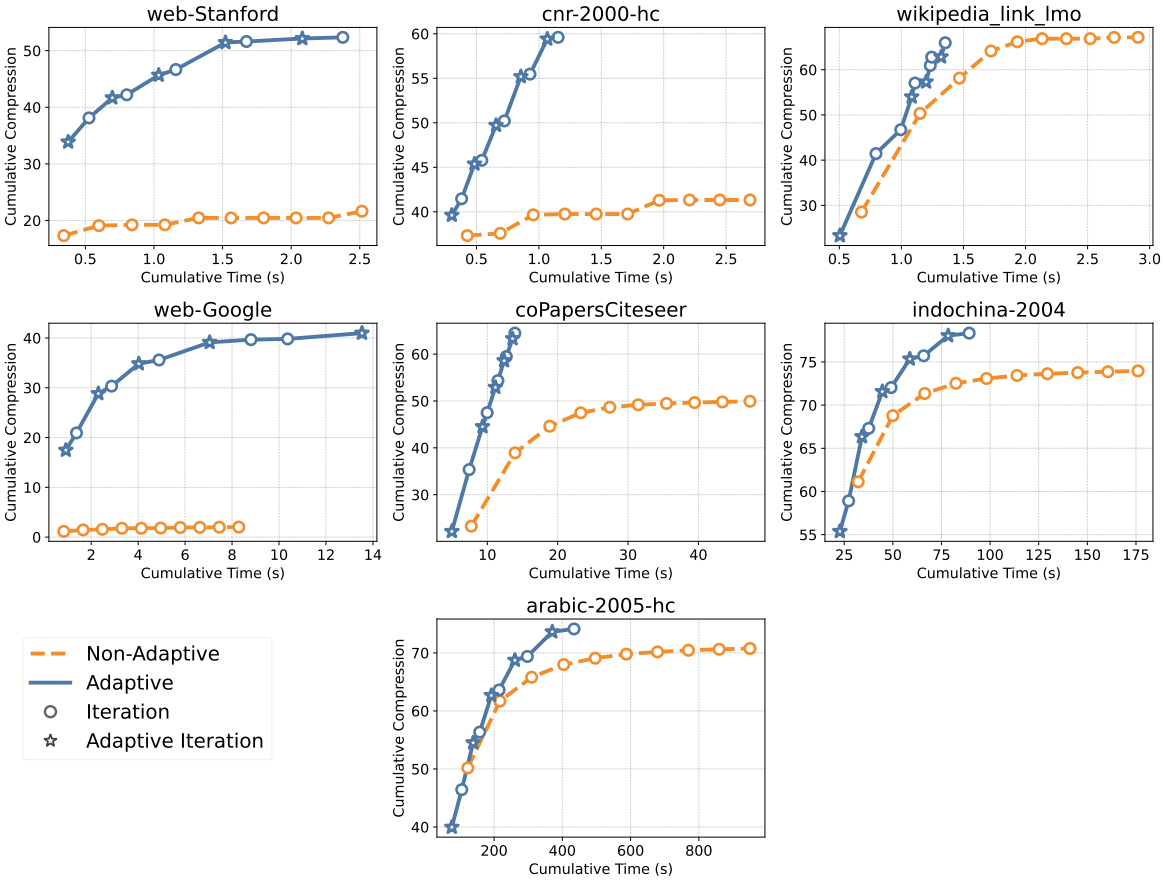

Figura 6.1: Compresión acumulada por segundo para la extracción de bicliques adaptativa y no adaptativa en cada conjunto de datos.

todo adaptativo ocupa típicamente entre el 70 % y el 75 % del espacio requerido por la versión no adaptativa, siendo mayor únicamente en un conjunto de datos, donde alcanza el 103 %.

La Figura 6.1 muestra cómo evoluciona el ratio de compresión acumulado a lo largo del tiempo y de las sucesivas iteraciones, tanto para la implementación adaptativa como para la no adaptativa. En todos los casos, la versión adaptativa alcanza una mayor compresión acumulada por segundo, lo que indica que el ajuste dinámico de los parámetros acelera la convergencia del algoritmo sin sacrificar eficiencia en la compresión. Las iteraciones inmediatamente posteriores a cada adaptación de parámetros se marcan con un símbolo de estrella, observándose que estas iteraciones mejoran efectivamente la compresión sin incrementar significativamente los tiempos de ejecución en la mayoría de los conjuntos de datos.

Dataset	**bpe**(G_M)	**bpe**($B+R$)	**bpe**(G_{k^2})	**bpe**(G_{RP32})	**bpe**(G_{WG})
web-Stanford	76.68	23.21	26.37	20.00	19.74
cnr-2000-hc	73.62	17.88	3.49	13.46	2.89
wikipedia-link-lmo	65.41	11.83	2.87	3.64	1.42
web-Google	79.66	28.88	33.57	24.66	23.88
coPapersCiteseer	65.54	12.29	3.24	4.24	1.63
indochina-2011	68.17	8.85	2.40	5.52	1.45
arabic-2005	68.08	10.18	2.74	5.72	1.84

Tabla 6.3: Comparación de bits por arista entre distintos métodos de compresión: bpe(G_M) corresponde a la representación base, bpe($B+R$) a nuestra técnica, bpe(G_{k^2}) al k^2-tree, bpe(G_{RP32}) a RePair32, y bpe(G_{WG}) a WebGraph.

A continuación, comparamos la eficiencia de compresión de nuestra representación basada en bicliques (utilizando construcción adaptativa) con otras técnicas de compresión. La Tabla 6.3 presenta los valores de bpe para nuestra representación bpe($B+R$), y los contrasta con bpe(G_M), correspondiente a la representación base utilizada por Arroyuelo et al. [2], así como con su implementación de k^2-tree, bpe(G_{k^2}). También se incluyen bpe(G_{RP32}), obtenidos mediante RePair32 [9], y bpe(G_{WG}), correspondientes al framework WebGraph [4].

Como se observa, WebGraph logra la mejor compresión, seguido por k^2-tree y RePair32, los cuales superan ampliamente a nuestra técnica en términos de espacio. No obstante, entre las representaciones que soportan directamente la multiplicación matricial, nuestra compresión basada en bicliques reduce sustancialmente el espacio de la representación base en todos los conjuntos de datos, utilizando entre un 13 % y un 36 % de su tamaño. Incluso si la representación base se optimizara almacenando únicamente su componente en formato *CSR*, esta aún requeriría entre un 26 % y un 72 % del espacio original. El k^2-tree también soporta multiplicación matricial; sin embargo, como se mostrará más adelante, puede resultar órdenes de magnitud más lento que nuestra representación basada en bicliques.

6.2 Tiempos de Ejecución y Uso de Espacio en la Multiplicación de Matrices

Esta sección estudia el escenario en el que las matrices se almacenan y se representan exclusivamente en formato *CSR*, el cual corresponde tanto al formato de entrada como de salida del algoritmo de multiplicación. En este contexto, la representación basada en bicliques se utiliza únicamente como un mecanismo para acelerar la multiplicación, y no con fines de compresión. Por lo tanto, la extracción de bicliques forma parte del propio proceso de multiplicación, y su costo se contabiliza explícitamente en los tiempos reportados.

Dado que las matrices de los conjuntos de datos considerados tienen distintos tamaños, medimos el tiempo necesario para calcular X^2 para cada matriz individual X, con el fin de evaluar el rendimiento de multiplicaciones del tipo $X \times Y$. Todos los tiempos de ejecución se miden como si se tratara de la multiplicación de dos matrices distintas; por ejemplo, no se aprovecha el hecho de que en el caso de X^2 los bicliques podrían extraerse una sola vez.

La Tabla 6.4 presenta los tiempos de ejecución de nuestro método, desglosados según las distintas etapas necesarias para realizar la multiplicación. La columna *BicEx* muestra el tiempo de ejecución de la extracción de bicliques para ambas matrices, el cual en la práctica se midió como el doble del tiempo de extracción sobre la matriz X. A continuación, se reporta el tiempo requerido para construir la representación *CSC* a partir del formato *CSR* de la matriz residual R_x, necesaria para computar el producto $R_x \times R_y$. Luego, se muestran los tiempos de multiplicación correspondientes a cada uno de los cuatro componentes, el tiempo requerido para fusionar los resultados en una representación basada en bicliques, y finalmente el tiempo necesario para convertir el resultado al formato *CSR*. La columna *Total* corresponde a la suma de todos estos tiempos y puede compararse directamente con la última columna, que reporta el tiempo de ejecución del método *Baseline*.

Los resultados muestran que nuestro enfoque puede ser significativamente más rápido que el método *Baseline* en los conjuntos de datos de mayor tamaño, alcanzando

Dataset	Nuestro enfoque (s)									Baseline (s)
	BicEx	CSC(R_x)	$R_x \times R_y$	$R_x \times B_y$	$B_x \times R_y$	$B_x \times B_y$	Merge	a CSR	Total	
web-Stanford	4.80	0.04	0.16	0.03	0.04	0.17	0.47	0.85	6.56	1.27
cnr-2000-hc	2.34	0.03	0.19	0.01	0.03	0.09	0.12	0.76	3.57	1.34
wikipedia-link-lmo	2.70	0.02	0.66	0.02	0.05	0.20	0.01	0.46	4.12	9.52
web-Google	27.16	0.22	0.58	0.14	0.18	0.56	1.83	2.28	32.95	2.43
coPapersCiteseer	27.38	0.19	4.39	2.35	2.50	12.13	2.85	7.09	58.88	91.99
indochina-2011	179.46	1.29	8.68	0.78	1.62	11.34	7.33	51.35	261.85	8982.77
arabic-2005	883.90	4.89	44.29	3.53	7.55	45.43	36.62	220.08	1246.29	2175.63

Tabla 6.4: Multiplicación matricial utilizando bicliques como mecanismo de aceleración, manteniendo la representación en formato *CSR*. Los tiempos se reportan en segundos. El tiempo total incluye la extracción de bicliques (una vez por matriz), la conversión a *CSC* de la matriz residual R_x, las cuatro operaciones de multiplicación, la etapa de fusión para obtener la representación $(B + R)$ y una etapa final para convertir el resultado a formato *CSR*. También se reporta el tiempo de ejecución del método *Baseline*.

aceleraciones de entre 1,75× y 34,3×. En contraste, el método *Baseline* suele ser más rápido en los conjuntos de datos pequeños. De este modo, nuestra técnica tiene un impacto claro en las multiplicaciones más costosas, cuyo tiempo de ejecución se mide en minutos u horas, mientras que la pérdida de rendimiento en las multiplicaciones más simples, que toman solo segundos, resulta menos relevante en términos absolutos.

Las mejoras introducidas en el rendimiento de la extracción de bicliques son claves para que este enfoque sea viable, ya que el costo adicional de dicha extracción se ve ampliamente compensado por una multiplicación considerablemente más rápida.

6.2.1 Multiplicación de matrices usando la representación con bicliques

Consideramos ahora el segundo escenario de uso, en el cual los bicliques se utilizan para representar matrices ocupando menos espacio; por lo tanto, el algoritmo de multiplicación opera utilizando nuestra representación basada en bicliques tanto en la entrada como en la salida. En este caso, no se realiza extracción de bicliques sobre las matrices de entrada, puesto que estas ya se encuentran en el formato (B, R); sin embargo, sí aplicamos *recompresión* sobre la matriz de salida, extrayendo bicliques desde el resultado materializado en formato *CSR*. En este escenario comparamos el espacio y el tiempo de ejecución de nuestro enfoque con los del *Baseline* y del k^2-tree. Asumiendo que la matriz ya se encuentra en el formato requerido.

La Tabla 6.5 muestra que nuestro enfoque con recompresión mejora de manera

Dataset	Baseline		k^2-tree		Nuestro enfoque			
	bpe	time(s)	bpe	time(s)	bpe	Mult(s)	BicEx2(s)	Total(s)
web-Stanford	65.47	1.27	20.45	283.44	7.92	1.76	6.05	7.81
cnr-2000-hc	64.96	1.34	2.61	6.63	5.86	1.23	6.60	7.83
wikipedia-link-lmo	64.26	9.52	3.73	24.30	9.73	1.42	7.91	9.33
web-Google	66.63	2.43	27.99	1199.00	12.47	5.79	14.49	20.28
coPapersCiteseer	64.19	91.99	3.17	324.62	8.31	31.50	113.85	145.35
indochina-2011	64.43	8982.77	2.47	4519.20	5.74	82.39	488.15	570.54
arabic-2005	64.32	2175.64	2.68	3579.26	6.16	362.39	2435.97	2798.36

Tabla 6.5: Espacio (en bpe) y tiempo de multiplicación (en segundos) usando nuestro método con el formato $R + B$ como entrada y salida, incluyendo recompresión. La columna *Mult* reporta el tiempo total para multiplicar las representaciones basadas en bicliques y producir el resultado en formato *CSR*, tal como se presenta en la Tabla 6.4. La columna *BicEx2* indica el tiempo adicional necesario para aplicar extracción de bicliques sobre el resultado, y la columna *Total* entrega el tiempo total. Con fines comparativos, también se reportan el bpe y el tiempo de multiplicación del *Baseline* y del *k^2-tree*.

sustancial la compresión del resultado en comparación con el *Baseline*, utilizando entre un 9 % y un 19 % de su espacio (y entre un 18 % y un 38 % si se optimizara el *Baseline* almacenando únicamente su componente en formato *CSR*). En las matrices más grandes, donde el rendimiento temporal es más relevante, nuestra técnica es competitiva con el *Baseline*, siendo solo entre un 30 % y un 60 % más lenta (y hasta 16 veces más rápida en un caso). En las matrices pequeñas, en cambio, somos más lentos por un factor de hasta 8. Por otra parte, nuestro enfoque es casi siempre más rápido que el k^2-tree, con aceleraciones de hasta 59 veces; el k^2-tree solo es más rápido en un conjunto de datos pequeño, por un 18 %. En términos de espacio, el resultado del k^2-tree suele ser más compacto que nuestra representación por un factor de 2.3–2.6, aunque esta relación se invierte en un par de conjuntos de datos.

Dataset	$E(X)$	$E(X^2)$	$E(X^4)$
web-Stanford	2,312,497	20,811,442	429,584,958
cnr-2000-hc	3,216,152	34,174,066	683,240,962
wikipedia-link-lmo	3,623,678	19,997,655	549,950,901
web-Google	5,105,039	29,710,164	585,573,462
coPapersCiteseer	32,073,436	264,584,716	12,248,543,674
indochina-2011	194,109,311	1,952,630,542	-
arabic-2005	639,999,458	8,323,612,632	-

Tabla 6.6: Número de aristas del grafo que representa X, X^2 y X^4.

Al comparar las columnas *Mult* y *BicEx2*, se observa que la necesidad de recom-

primir la salida vuelve considerablemente más lenta nuestra multiplicación basada en bicliques. La Tabla 6.6 explica por qué extraer bicliques sobre el resultado es mucho más costoso que hacerlo sobre las matrices de entrada: en nuestro escenario, las matrices resultantes tienden a ser significativamente más densas (pues su tamaño está relacionado con el número de caminos de largo 2 en el grafo representado). Esto motiva un tercer escenario, en el cual se utilizan directamente como salida los bicliques producidos por el algoritmo de multiplicación, sin aplicar recompresión.

6.2.2 Multiplicación de matrices con bicliques sin recompresión

Dataset	**Baseline**		**Recompresión**		**Sin recompresión**	
	bpe	time(s)	bpe	time(s)	bpe	time(s)
web-Stanford	65.47	1.27	7.92	7.81	11.03	0.91
cnr-2000-hc	64.96	1.34	5.86	7.83	8.22	0.47
wikipedia-link-lmo	64.26	9.52	9.73	9.33	11.74	0.96
web-Google	66.63	2.43	12.47	20.28	17.06	3.51
coPapersCiteseer	64.19	91.99	8.31	145.35	7.99	24.41
indochina-2011	64.43	8982.77	5.74	570.54	6.15	31.04
arabic-2005	64.32	2175.64	6.16	2798.36	7.47	142.31

Tabla 6.7: Comparación de espacio (en bpe) y tiempo de multiplicación (en segundos) con y sin recompresión. La columna *Recompresión* reporta el espacio y el tiempo cuando la multiplicación es seguida por extracción de bicliques sobre el resultado. La columna *Sin recompresión* muestra el rendimiento cuando el resultado se mantiene directamente en el formato $B + R$ producido por la multiplicación basada en bicliques, sin recomprimir.

Finalmente evaluamos el escenario en el cual se utilizan directamente como salida los bicliques producidos por la multiplicación, de modo que la representación basada en bicliques del resultado se obtiene sin aplicar extracción adicional. Esto conduce a un algoritmo de multiplicación mucho más rápido, ya que nunca se extraen bicliques desde el resultado. Aparece, sin embargo, un nuevo inconveniente: la calidad de la representación puede degradarse tras una multiplicación (por ejemplo, aumentando el número de bicliques y su tamaño promedio disminuye, o creciendo el residual), en comparación con lo que se obtendría mediante recompresión. Medimos estos efectos y destacamos que, aun así, siempre es posible aplicar recompresión en cualquier momento si se estima que la calidad de la representación ha disminuido.

La Tabla 6.7 compara el espacio y el tiempo de ejecución con y sin recompresión. Se observa que, en la mayoría de los casos, el espacio comprimido se degrada entre un 7 % y un 40 % cuando no se aplica recompresión (curiosamente, en un caso la recompresión empeora levemente el espacio). A cambio, la multiplicación sin recompresión es mucho más rápida, por un factor de hasta 20. Comparado con el *Baseline*, esta alternativa utiliza solo entre un 10 % y un 26 % de su espacio (o entre un 20 % y un 52 % si se mejorara el espacio del *Baseline*) y es casi siempre más rápida, con aceleraciones de hasta 289 (siendo más lenta solo en un caso, por un 44 %).

Nuestro experimento final busca medir cómo la degradación en la calidad de los bicliques puede afectar el rendimiento temporal de operaciones futuras sobre la matriz. Para ello, elevamos al cuadrado matrices X para obtener X^2 y luego volvemos a elevar al cuadrado el resultado para obtener X^4. Realizamos este procedimiento con y sin una recompresión intermedia. En el primer caso, recomprimimos X^2 antes de volver a multiplicar, con la expectativa de que este costo adicional sea compensado por una multiplicación posterior más rápida gracias a la presencia de bicliques de mayor calidad.

Dataset	**Baseline (s)**		**k²-tree (s)**		**Sin recompresión (s)**				**Con recompresión (s)**				
	X^4	bpe	X^4	bpe	X^2	X^4	Total	bpe	X^2	BicEx2	X^4	Total	bpe
web-Stanford	81.78	64.07	8,898.64	12.65	0.91	22.74	23.65	9.00	1.76	6.05	7.38	**15.19**	4.93
cnr-2000-hc	198.85	64.05	455.96	1.70	0.47	28.96	29.43	5.82	1.23	6.60	6.75	**14.58**	3.56
wikipedia-link-lmo	335.09	64.01	1,008.50	2.41	0.96	79.49	80.45	16.45	1.42	7.91	45.00	**54.33**	14.57
web-Google	87.32	64.14	19,879.91	19.14	3.51	52.88	56.39	15.14	5.79	14.49	16.79	**37.07**	8.05
coPapersCiteseer	13,398.54	64.00	26,775.85	2.40	24.41	2,948.08	2,972.49	7.23	31.50	113.85	1,039.51	**1,184.86**	6.63

Tabla 6.8: Comparación de tiempos de ejecución (en segundos) y espacio para calcular X^4 con y sin recompresión intermedia.

La Tabla 6.8 presenta el espacio y los tiempos de ejecución para ambas estrategias. También se incluyen los resultados del *Baseline* y del k^2-tree. La columna *Sin recompresión* considera las dos multiplicaciones y el tiempo total. La columna *Con recompresión* incluye además la recompresión de X^2 (pero no la de X^4). Como se observa, ambas alternativas de nuestro enfoque son más rápidas que el *Baseline* (por un factor de hasta 13.6) y que el k^2-tree (por un factor de hasta 586). Por otra parte, recomprimir X^2 hace que el proceso completo sea casi el doble de rápido que no recomprimirlo; es decir, la recompresión intermedia se justifica en tiempo.

El resultado de este experimento se entiende mejor considerando que las matrices tienden a volverse considerablemente más densas tras multiplicarse. La Tabla 6.6 mues-

tra el número total de aristas de la matriz de entrada X, de X^2 y de X^4. Se observa que el número de aristas crece aproximadamente un orden de magnitud cada vez que se eleva la matriz al cuadrado. La tabla omite las dos matrices más grandes, ya que no fue posible calcular X^4 debido a limitaciones de memoria. Además, el hecho de que superemos al *Baseline* por un margen mucho mayor en espacio sugiere que las matrices que son producto de otras tienden a ser más compresibles mediante bicliques, lo cual es coherente con la intuición detrás del algoritmo de Schoor.

Finalmente, la Figura 6.2 muestra la mejora de rendimiento obtenida por nuestro método en comparación con el *Baseline* y el k^2-tree al computar X^2 y X^4. Definimos la aceleración (*speedup*) como la razón entre el tiempo de ejecución del método comparado y el tiempo requerido por nuestra mejor alternativa (mostrada en la Tabla 6.8). Un valor igual a uno indica tiempos equivalentes; valores mayores indican cuántas veces nuestra solución es más rápida.

Observamos que en X^2 el *Baseline* es más rápido en dos conjuntos pequeños, mientras que somos más rápidos que el k^2-tree en todos los casos. Sin embargo, para X^4 somos más rápidos que ambos métodos para todos los conjuntos de datos, obteniendo un *speedup* entre 2 y 8 respecto del *Baseline* y entre 10 y más de 300 respecto del k^2-tree.

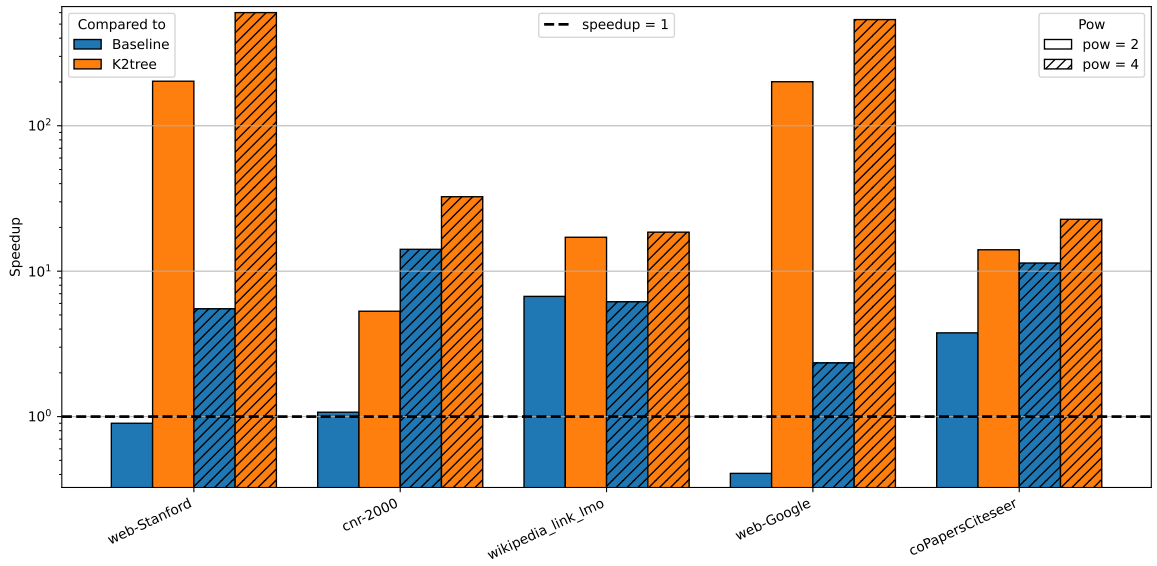

Figura 6.2: Speedup obtenido para X^2 y X^4 comparado con el *Baseline* y el k^2-tree.

Cabe destacar que la representación basada en bicliques no solo reduce el uso de espacio, sino que además mejora o mantiene los tiempos de cómputo en este escenario, particularmente en operaciones de multiplicación costosas.

7 Conclusión y trabajo futuro

7.1 Conclusiones

En este trabajo se abordó el problema de la compresión y multiplicación eficiente de matrices booleanas asociadas a grafos dispersos de gran escala, proponiendo un enfoque basado en bicliques que permite reducir tanto el uso de espacio como el costo computacional de las operaciones algebraicas.

El primer objetivo consistió en analizar los parámetros del algoritmo de extracción de bicliques propuesto por Hernández y Navarro [12]. Para ello, se realizó un estudio fundamentado en las distribuciones de grado de los grafos, con el fin de identificar configuraciones que permitan obtener una compresión efectiva sin requerir conocimiento previo de la estructura del grafo. Como resultado de este análisis, se propuso un método adaptativo de extracción de bicliques guiado por percentiles, capaz de ajustar dinámicamente sus parámetros según las características del grafo de entrada. Este enfoque permitió reducir significativamente el tiempo de extracción, manteniendo e incluso mejorando el ratio de compresión, cumpliendo así el segundo objetivo planteado.

En relación con el tercer objetivo, se desarrolló un método que aprovecha la representación comprimida basada en bicliques para acelerar la multiplicación de matrices booleanas. En particular, se extendió el algoritmo de Schoor para operar directamente sobre matrices representadas mediante bicliques y matrices residuales, descomponiendo la multiplicación en cuatro componentes: matriz–matriz, matriz–biclique, biclique–matriz y biclique–biclique. Esta formulación permite expresar el resultado de la multiplicación como una nueva representación en componentes ($M + B$), preservando la estructura comprimida.

Los resultados experimentales muestran que este enfoque logra reducciones sustanciales en los tiempos de cómputo, especialmente en matrices de gran tamaño, donde se obtienen aceleraciones de hasta dos órdenes de magnitud respecto a implementaciones basadas exclusivamente en formatos *CSR*/*CSC*. Asimismo, se observa que las matrices resultantes de las multiplicaciones tienden a ser aún más compresibles mediante bicliques, lo que refuerza la utilidad de esta representación para operaciones algebraicas encadenadas. En este contexto, la aplicación de una recompresión posterior permite mejorar la calidad de la representación y acelerar operaciones posteriores, como el cálculo de potencias superiores, por ejemplo X^4.

En conjunto, los resultados obtenidos demuestran que la representación basada en bicliques no solo constituye un mecanismo efectivo de compresión, sino que también habilita una álgebra matricial eficiente en el dominio comprimido, ofreciendo una base sólida para el procesamiento escalable de grafos dispersos masivos.

7.2 Trabajo futuro

Como líneas de trabajo futuro, resulta relevante profundizar en el análisis de las propiedades estructurales de los grafos con el objetivo de refinar el proceso de extracción adaptativa de bicliques. Actualmente, este método depende de la selección de percentiles y del parámetro `threshold`, cuyos valores se determinan de manera heurística; una caracterización más precisa de estos parámetros podría conducir a una mayor eficiencia en compresión y a mejores tiempos de ejecución.

Otra línea de investigación consiste en extender las técnicas desarrolladas hacia implementaciones paralelas y en GPU, con el fin de aprovechar arquitecturas de cómputo masivamente paralelas y escalar aún más el rendimiento de la multiplicación basada en bicliques. Esto se abordó en la reimplementación del extractor de bicliques, sin embargo ocurria que al trabajar clusters de distintos tamaños no se conseguía una aceleración tan significativa. Con el enfoque ade extracción adaptativa es posible que hayan mejoras en este ámbito.

Asimismo, resulta de interés generalizar el modelo propuesto más allá de matri-

ces booleanas, incorporando soporte para grafos ponderados y dinámicos, donde las estructuras de bicliques evolucionan en el tiempo. Finalmente, se plantea como trabajo futuro explorar la aceleración de otras operaciones algebraicas sobre grafos, tales como el cierre transitivo, utilizando la representación basada en bicliques, con el objetivo de completar un marco de álgebra booleana eficiente en formato comprimido.

Bibliografía

[1] João NF Alves, Samir Moustafa, Siegfried Benkner, Alexandre P Francisco, Wilfried N Gansterer, and Luís MS Russo. Accelerating graph neural networks using a novel computation-friendly matrix compression format. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1091–1103. IEEE, 2025.

[2] Diego Arroyuelo, Adrián Gómez-Brandón, and Gonzalo Navarro. Evaluating regular path queries on compressed adjacency matrices. *The VLDB Journal*, 34(1):2, 2025.

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[4] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[6] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval*, pages 18–30, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[8] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336, 1998.

[9] Paolo Ferragina, Travis Gagie, Dominik Köppl, Giovanni Manzini, Gonzalo Navarro, Manuel Striani, and Francesco Tosoni. Improving matrix-vector multiplication via lossless grammar-compressed matrices. *Proc. VLDB Endow. 15(10): 2175-2187*, 2022.

[10] Alexandre P. Francisco, Travis Gagie, Dominik Köppl, Susana Ladra, and Gonzalo Navarro. Graph compression for adjacency-matrix multiplication. *SN Computer Science*, page 193, 2022.

[11] Felipe Glaria, Cecilia Hernández, Susana Ladra, Gonzalo Navarro, and Lilian Salinas. Compact structure for sparse undirected graphs based on a clique graph partition. *Information Sciences*, pages 485–499, 2021.

[12] Cecilia Hernández and Gonzalo Navarro. Compressed representations for web and social graphs. *Knowledge and information systems*, pages 279–313, 2014.

[13] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2002.

[14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[15] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

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

[17] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[18] Amir Schoor. Fast algorithm for sparse matrix multiplication. *Information Processing Letters*, pages 87–89, 1982.

[19] Francesco Tosoni, Philip Bille, Valerio Brunacci, Alessio De Angelis, Paolo Ferragina, and Giovanni Manzini. Toward greener matrix operations by lossless compressed formats. *IEEE Access*, 2025.

[20] Andrew K Watson, Romain Lannes, Jananan S Pathmanathan, Raphaël Méheust, Slim Karkar, Philippe Colson, Eduardo Corel, Philippe Lopez, and Eric Bapteste. The methodology behind network thinking: graphs to analyze microbial complexity and evolution. *Evolutionary genomics: Statistical and computational methods*, pages 271–308, 2019.