Universidad de Concepción

UNIVERSIDAD DE CONCEPCIÓN

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA Y CIENCIAS DE LA COMPUTACIÓN

MEMORIA DE TÍTULO:

MULTIPLICACIÓN DE MATRICES BOOLEANAS COMPRIMIDAS USANDO BICLIQUES

Autor: José Zagal Vallejos

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

Profesor Guía:
Cecilia Hernández Rivas

Marzo 2024
Concepción, Chile

Resumen

Las matrices booleanas son ampliamente utilizadas en campos como la informática, electrónica, estadística y biología para representar relaciones binarias entre los elementos. El constante crecimiento de datos generado a lo largo de los años, impulsado por la expansión de internet, el auge de las redes sociales y el incremento en el número de páginas web, ha dado lugar a una cantidad masiva de información. Una forma de representarlos, es mediante la utilización de grafos, donde dependiendo de la densidad del grafo se puede representar de distintas formas con el objetivo de reducir el espacio de almacenamiento. La reducción de espacio de representación puede, a su vez, permitir computar operaciones en menor tiempo. A partir de esto, se ve interesante estudiar nuevas representaciones que permitan reducir el espacio y tiempo de cómputo de operaciones de interés.

En particular, Hernández y Navarro propusieron un algoritmo para encontrar subgrafos densos compuestos por cliques y bicliques, donde utilizan grafos de hasta 32 *bits*.

Esta memoria de título busca lograr una mejora al extractor de bicliques utilizado por Hernández y Navarro, soportando grafos masivos de 64 *bits*. Los resultados obtenidos muestran un buen nivel de compresión, llegando a comprimir mas del 50 % del grafo en todos los casos, consiguiendo que disminuya la cantidad de *bits* utilizados por arista en todos los casos.

El segundo objetivo es el diseño e implementación de la multiplicación de matrices ralas o dispersas booleanas utilizando sus bicliques. Para esto se definieron dos posibles formas de obtener una matriz resultante, la primera obtenemos la matriz completa y la segunda forma obtenemos la matriz mas una colección de bicliques. Dependiendo del coeficiente de compresión obtenido la multiplicación puede ser una muy buena opción a considerar, o en caso contrario, los bicliques generan una ralentización del algoritmo.

Agradecimientos

Agradezco profundamente a mi mamá, Emilia Vallejos y a mi papá, Hugo Zagal. Siempre estuvieron a mi lado durante los momentos difíciles que atravesé, brindándome aliento y motivación antes de cada desafío. Son mi más grande fuente de inspiración, sin ellos no tengo idea si habría llegado hasta este punto en mi trayectoria académica.

Agradezco también a mi profesora guía, la Dra. Cecilia Hernández, cuyo respaldo fue invaluable durante todo este proceso educativo. Su disposición y orientación fueron cruciales para mi desarrollo. Le estoy profundamente agradecido por su dedicación y apoyo incondicional. Por otro parte, también le doy mis agradecimientos al profesor de la Universidad de Chile, el Dr. Gonzalo Navarro, cuya orientación fue crucial para el desarrollo de este trabajo.

Agradezco sinceramente a mi compañero, Nicolás Araya, quien fue el primero en introducirme en este fascinante tema de memoria. Además de ser una fuente confiable para solicitar opiniones en diversas situaciones, su apoyo ha sido inestimable.

Por ultimo, se agradece al centro de excelencia en investigación CeBiB por el apoyo financiero de la memoria.

Índice general

Capítulo 1

Introducción

1.1. Motivación

Las matrices booleanas son ampliamente utilizadas en campos como la informática, electrónica, estadística y biología para representar relaciones binarias entre los elementos. Algunas relaciones incluyen enlaces dirigidos entre páginas de la web, representación de usuarios en las redes sociales, expresiones lógicas, presencia de voltaje circuitos digitales y para resumir la compartición de genes entre conjuntos de genomas [23].

El constante crecimiento de datos generado a lo largo de los años, impulsado por la expansión de internet, el auge de las redes sociales y el incremento en el número de páginas web, ha dado lugar a una cantidad masiva de información. Una forma de representarlos es mediante la utilización de grafos, donde dependiendo de la densidad del grafo se puede representar de distintas formas con el objetivo de reducir el espacio de almacenamiento. La reducción de espacio de representación puede, a su vez, permitir computar operaciones en menor tiempo. A partir de esto, se ve interesante estudiar nuevas representaciones que permitan reducir el espacio y tiempo de cómputo de operaciones de interés.

El tiempo de procesamiento de estos grafos aumenta rápidamente con el número

de nodos y aristas, lo que puede dificultar su manejo y computo eficiente de los algoritmos. Esto puede llevar a tiempos de ejecución excesivos y a la necesidad de técnicas de optimización y reducción de la complejidad del grafo para un análisis eficiente.

Una de las operaciones mas importantes que se utilizan en el procesamiento de estos es la multiplicación, dado que a medida que aumenta el tamaño del grafo, aumenta de forma cúbica el tiempo de cálculo de esta operación. Algunas funcionalidades son encontrar diferentes caminos para una ruta [24], recomendaciones de amistad dentro de las redes sociales, alcance de navegación en las páginas web entre otros. Todas estas aplicaciones tienen un problema en común, gran cantidad de ceros dentro de su representación matricial.

Luego, una pregunta de investigación asociada a esta problemática es: ¿Se puede utilizar de alguna forma la gran cantidad de ceros dentro de la matriz para conseguir una ganancia en espacio de almacenamiento ,y a la vez, en tiempo al momento de realizar multiplicaciones? Estas matrices se llaman matrices ralas (sparse) o de baja densidad, las cuales suelen representarse como listas de adyacencia, dado que poseen muy pocos unos dentro de ella, estas listas ahorraran espacio al no representar los ceros de estas matrices ralas. Por otra parte se podría emplear la búsqueda de aristas repetidas dentro de la matriz para obtener una matriz aún más rala, logrando una mayor eficiencia de espacio y encontrar alguna forma para realizar operaciones algebraicas.

1.2. Contenido del documento

El documento se estructura de la siguiente manera. En el capítulo 2 se presentan las definiciones y conceptos que ayudan a la lectura y entendimiento del documento. El capítulo 3 describe las distintas metodologías empleadas tanto para la compresión de grafos como para operaciones algebraicas, incluyendo enfoques con y sin compresión. En el capítulo 4 es donde encontraremos una forma de utilizar bicliques para la multiplicación, además de cómo se obtienen y cómo se diferencia de la multiplicación

normal de matrices ralas. En el capítulo 5 se analizan los resultados obtenidos a partir de la multiplicación, además de compararlos con otros métodos. Por último, en el capítulo 6 se presentan las conclusiones finales del trabajo elaborado.

A continuación se presentan el objetivo general y específicos del trabajo realizado.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar e implementar la multiplicación para matrices booleanas, utilizando el extractor de bicliques con distintos parámetros para analizar el impacto que logra en tiempo y espacio de almacenamiento.

1.3.2. Objetivos específicos

1. Implementar la identificación y extracción de bicliques en grafos con soporte de 64 bits.
2. Diseñar e implementar multiplicación de matrices booleanas utilizando bicliques.
3. Comparar resultados con el algoritmo base y el $k^2 - tree$ descrito en trabajo relacionado.

Capítulo 2

Definiciones y conceptos previos

2.1. Grafo

Un grafo es un par ordenado $G = (V, E)$ donde V es un conjunto finito, no vacío de elementos llamados vértices y E es un conjunto de pares no ordenados de V llamado aristas, es decir, $E \subseteq V \times V$ y $\forall e \in E$, $|e| = 2$. Por otra parte, dado G, $V(G)$ se refiere al conjunto de vértices de G. Análogamente $E(G)$ se refiere al conjunto de aristas. El orden del grafo esta considerado por el número de vértices $|V(G)|$ y el tamaño esta dado por el número de aristas $|E(G)|$.

Dos vértices $u, v \in E$ son vértices adyacentes o vecinos si $\exists e \in E, e = uv$. Además, para este caso u tiene como vecino directo a v, por otra parte, v tiene como vecino indirecto a u. Sea $G = (V, E)$ un grafo y $v \in V$ se define la vecindad de v como:

$$N_G(v) = \{u \in V : uv \in E\} = \{u \in V : u \text{ es vértice adyacente a } v \text{ en G }\} \quad (2.1)$$

2.1.1. Grafos de la web

Un grafo de la web representa una porción de la compleja red de hipervínculos que constituye la estructura de la web en un momento específico. Este grafo se construye

como un conjunto de nodos y aristas, donde cada nodo representa una página web identificada por su *URL*, y cada arista indica la existencia de un hipervínculo en una página web que dirige hacia otra página.

2.1.2. Representación matricial

Cualquier grafo finito tiene representación matricial. Es una de las representaciones mas fáciles de entender y construir, además que son muy utilizadas en distintas aplicaciones. Esta representación matricial también es llamada matriz de adyacencia. Para esto utilizaremos la figura 2.1 como ejemplo.

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	1	0	0	0

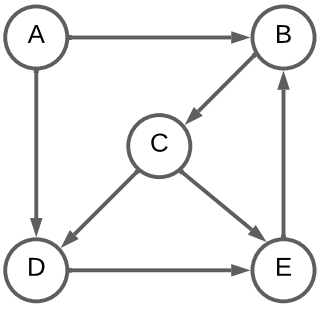


Figura 2.1: Forma matricial

2.1.3. Bicliques o grafos bipartitos completos

Sea $G = (V, E)$ un grafo cualquiera. Definimos $\mathbf{B}$ como un conjunto de bicliques de tamaño b, donde b representa el número de bicliques encontrados por el algoritmo extractor de bicliques utilizando el grafo G.

$$\mathbf{B} = \{B_1, B_2, ..., B_b\} \tag{2.2}$$

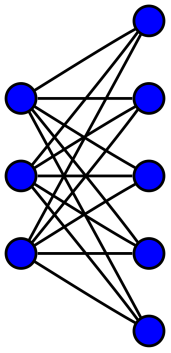


Figura 2.2: Ejemplo de un biclique B_i. Los vértices de la izquierda representan el conjunto S_i y los de la derecha representan el conjunto C_i.

Un biclique B_i se define como un par (S_i, C_i), donde S_i y C_i son conjuntos de vértices disjuntos. Todos los vértices de S_i tienen aristas que se dirigen hacia todos los vértices de C_i. Esta definición se puede visualizar mediante un ejemplo en la Figura 2.2. Dado la definición anterior y la ecuación dada en 2.2, podemos considerar $\mathbf{B}$ como:

$$\mathbf{B} = \{(S_1, C_1), (S_2, C_2), ..., (S_b, C_b)\} \tag{2.3}$$

Estrella

Una estrella es una forma específica de un biclique. Todos los bicliques que sean un árbol son llamados estrellas. Algunos ejemplos se pueden ver en la figura 2.3.

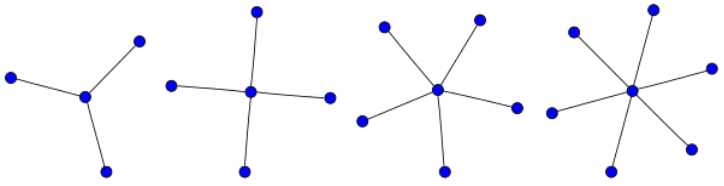


Figura 2.3: Estrellas de tamaño 3, 4, 5 y 6 respectivamente.

2.2. Multiplicación de matrices booleanas

Dadas dos matrices A y B, con $A \in \{0,1\}^{m\times n}$ y $B \in \{0,1\}^{n\times p}$, la multiplicación de matrices booleanas esta dada por:

$$C_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge B_{kj} \tag{2.4}$$

donde obtenemos la matriz C, con $C \in \{0,1\}^{m\times p}$. Este tipo de multiplicación es muy lenta usando el algoritmo mas simple, dado que es O(n^3), el cual es general sin importar que tan rala es la matriz. Sin embargo, existen algoritmos que reducen la complejidad, como es el caso del algoritmo de Strassen [22], cuya complejidad es $O(n^{2,81})$ y el algoritmo de Coppersmith-Winograd cuya complejidad es $O(n^{2,37}$ [9]. Actualmente, se intenta otras formas de definir mejores algoritmos tanto para matrices generales y especialmente matrices ralas las cuales aparecen en muchas aplicaciones.

2.3. Producto cartesiano

El producto cartesiano es una operación entre 2 conjuntos A y B, el cual su resultado es otro conjunto. Este conjunto esta compuesto por todos los pares ordenados que puedan formarse, tomando como primer elemento del par ordenado sea del primer conjunto A y el segundo elemento pertenezca al segundo conjunto B. Por ejemplo, considerando los conjuntos:

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

El producto entre A y B nos queda de la siguiente manera:

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

Analógicamente el producto entre B y A:

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

Capítulo 3

Trabajo relacionado

Esta sección se separa en dos partes. Primero analizaremos diversas técnicas de compresión de matrices. Después estudiaremos como distintos trabajos utilizaron compresión de matrices para lograr operaciones algebraicas, los cuales lograron obtener una mejora en tiempos de cómputo y/o de espacio de almacenamiento.

3.1. Compresión de matrices

Usar matrices booleanas no siempre es la mejor opción para representar los grafos, dado que si la matriz es demasiada rala o dispersa, tendremos muchos ceros que no nos aportan información. Para esto se han buscado distintas alternativas de representar estos grafos, con el fin de mantener la misma información ahorrando espacio. El formato *Compressed Sparse Row* fue utilizado al menos desde la década de 1960, pero su primera aparición formal fue en 1967 [8]. Este formato aprovecha la dispersión de los datos para solamente representar los valores distintos de ceros, lo que lo hace una estructura perfecta para matrices con estas características. Esto lo hace disponiendo de 3 vectores, el primero para guardar los punteros de las filas, el segundo guarda las columnas del primer vector y el tercero guarda los valores. A pesar que fue pensado para representar matrices con pesos, puede adaptarse para matrices sin pesos, tal como lo hacen en [2].

Un trabajo muy interesante es el de Brisaboa et al[5], en el cual emplearon técnicas para comprimir grafos de la web. Ellos explotaban la dispersión y el clustering de la matriz de adyacencia para reducir espacio e incluso navegar a través de ella usando la estructura llamada k^2tree. El k^2tree representa la matriz de adyacencia de $n \times n$ por un árbol k^2-ario de altura $h = \lceil log_k n \rceil$. Cada nodo está representado por un bit, 1 si tiene nodos internos o 0 en caso contrario, y cada nodo interno (valor 1) tiene k^2 hijos. Esta estructura fue recientemente utilizada en [2] para evaluar operaciones algebraicas en *regular path queries*, los cuales son caminos que se representan como expresiones regulares. Dentro de las operaciones utilizadas están la transpuesta, suma, multiplicación y clausura transitiva. A pesar de no lograr una mejora en tiempo con respecto a su método base, logran un grado de compresión cuatro veces mas pequeño y solo unos segundos de diferencias con respecto a los demás métodos de compresión tales como *Ring*, *Jena*, *Virtuoso* y *Blazegraph*.

Por último, se ha utilizado la identificación y extracción de bicliques para la compresión de grafos [15, 16] . Esto se debe a la idea de que un biclique puede evitar la repetición de aristas, lo que elimina la redundancia en el grafo. Para obtener estos bicliques, se emplean diversas técnicas. Primero se utiliza el método *min hashing* para encontrar similitudes entre listas de adyacencia. A partir de estos resultados, se crean clústeres para luego introducir cada uno en un *prefix tree*, permitiendo así la extracción de los bicliques.

3.2. Operaciones algebraicas utilizando compresión

La multiplicación de matrices es un problema fundamental en ciencias de la computación [1]. Un trabajo se enfoca en la compresión mediante gramática [13], el cual realiza esta compresión a una versión modificada de *Compressed Sparse Row* el cual llaman *Compressed Sparse row/value*. Esto les permite realizar multiplicación matriz-vector en tiempo proporcional al número de reglas de reemplazo. Esto logra una mejora de tiempo y espacio con respecto a su estado del arte *Compressed Lineal*

Algebra[11, 12].

Otro trabajo interesante es el de Karante C. [17], el cual utiliza nodos virtuales para la compresión de la matriz de adyacencia, y para esto lo hace mediante bicliques. Dado un biclique (S, T) donde la cantidad de aristas representadas esta dado por $|S|\times|T|$, se crea un nodo virtual w, el cual no pertenece a la matriz. Se eliminan todas las aristas de S hacia T, luego todos los nodos de S apuntan hacia w y w apunta a los nodos de T. A partir de esto proponen la multiplicación matriz-vector, definiendo x cómo vector a multiplicar. Para obtener el vector y resultante, los autores ocupan operaciones de empuje. El valor almacenado en el nodo u en x es empujado a través de la arista uv:

$$y[v] = \sum_{uv \in G} x[u]$$

A partir de esta fórmula, modifican el algoritmo para matrices comprimidas con nodos virtuales.

Francisco [14] busca acelerar el tiempo de realizar la multiplicación matriz $\times$ vector en proporción a la compresión del grafo, para esto utilizó 2 trabajos distintos de compresión. Primero utilizó el *framework* de *WebGraph* [3]. Para calcular el vector y resultante se optaron por 2 posibles formas de computarlo. La primera es multiplicar de forma convencional $A \cdot x^T$. La segunda forma es restando un vector v a la matriz A, obteniendo A', quedando la multiplicación $A' \cdot x^T + w^T$, donde $w = v \cdot x^T$. Por otra parte utiliza la compresión mediante bicliques [16], el cual para grandes matrices con muchos bicliques obtuvo una mejora en tiempo. Este trabajo muestra una reducción en espacio y tiempo de ejecución del algoritmo de PageRank usando la estructura comprimida. PageRank [4] fue desarrollado por Google en 1998 para evaluar la relevancia de las páginas web en función de su importancia en la web, es decir, cuántas páginas enlazan hacia ellas y la importancia de esas páginas.

Capítulo 4

Multiplicación de matrices mediante compresión de bicliques

4.1. Compresión mediante bicliques

Dado que un biclique B_i es un par (S_i,C_i), podemos observar que este representa $|S| \cdot |C|$ aristas usando $O(|S| + |C|)$. Este método es muy competente al momento de ahorrar espacio, dado que mientras mas grande la matriz, y mayor sea la cantidad de bicliques, mayor es la compresión. Caso contrario, en el peor caso, con un biclique de tamaño 2×2 no hay ahorro de espacio.

4.2. Extracción de bicliques

A continuación, se explican los distintos procesos que componen el algoritmo extractor de bicliques que representa la Figura 4.1. Para llevar a cabo esta explicación, nos apoyaremos en el grafo representado en la Tabla 4.1, el cual servirá como ejemplo para cada fase. El primer paso consiste en computar y representar cada lista de adyacencia del grafo con *signatures* usando *Min Hashing*. A partir de ellas se obtienen clusters encontrando *signatures* similares. Luego, para cada cluster se procesan las listas de adyacencia usando un árbol de prefijos para identificar y extraer los

bicliques.

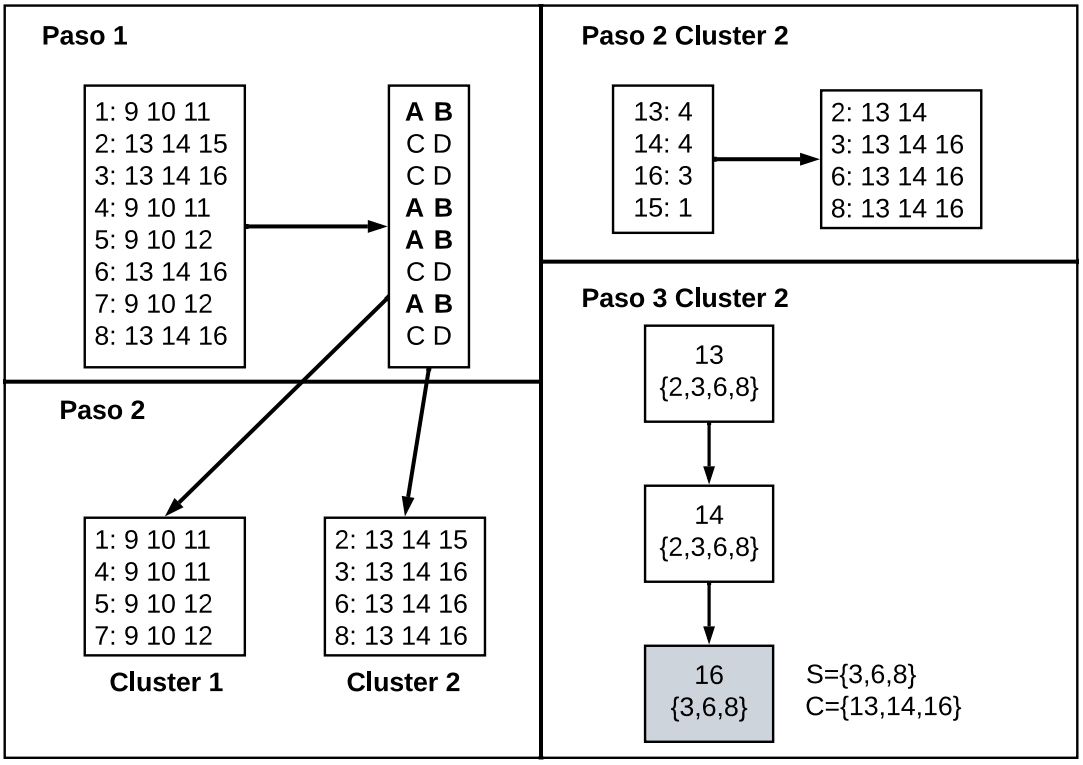


Figura 4.1: Etapas para extracción de bicliques

4.2.1. Min Hashing

Min Hash es una técnica que es utilizada principalmente para estimar rápidamente que tan similares son dos conjuntos. Esta técnica fue propuesta en 1997 [6] para detectar páginas web duplicadas con el fin de eliminarlas [7].

Normalmente, si uno quiere comparar todos los pares posibles de n conjuntos requiere un total de n^2 operaciones. Luego, para el problema de encontrar la similitud de todos los pares posibles de listas de adyacencia de grafos masivos hacer todas las comparaciones es computacionalmente muy costoso.

En este trabajo se utiliza la técnica de Min-hash para evitar comparar las listas de adyacencia directamente, dado Min-hash permite estimar la similitud entre las listas

Vértice	Arista 1	Arista 2	Arista 3
1	9	10	11
2	13	14	15
3	13	14	16
4	9	10	11
5	9	10	12
6	13	14	16
7	9	10	12
8	13	14	16

Tabla 4.1: Lista de adyacencia a extraer bicliques.

de adyacencia de los vértice del grafo usando las signatures. Esta técnica utiliza una función hash para la implementación de la técnica. En particular, la función hash usada está definida de la siguiente manera:

$$M(x) = (Ax + B)\,\%p, \tag{4.1}$$

donde p es un número primo lo suficientemente grande. En este caso, se utilizó el número primo de Mersenne $p = 2^{61} - 1$. Por otra parte, A y B corresponden a números aleatorios entre 0 y el número primo elegido, y x es un número que es el id asociado al *k-shingle*. Esta función se aplica a cada *k-shingle*, el cual es una concatenación de k vértices contiguos en la lista de adyacencia de x. Una vez que se procesan las listas de adyacencia, se conservan los P valores hash mas pequeños, que son las Min-hashes y que representan la *signature* (P valores hash mas pequeños) de ese vértice. Por ejemplo, la Tabla 4.2 muestra los signatures obtenidos para el grafo anterior.

Vértice	Signature 1	Signature 2
1	1760843776	189794601
2	1625095119	16656699
3	1625095119	16656699
4	1760843776	189794601
5	1760843776	189794601
6	1625095119	16656699
7	1760843776	189794601
8	1625095119	16656699

Tabla 4.2: Representación de Min-hashes (Signatures) de un grafo.

4.2.2. Clustering

Los algoritmos de *clustering* tienen como finalidad agrupar objetos en conjuntos o grupos por similitud o cercanía en términos de distancia. Luego, los elementos dentro de un mismo grupo compartan características comunes. Existen múltiples áreas donde se pueden aplicar esta técnica y existen muchos algoritmos con diversos enfoques de solución, tales como particionamiento (K-Means), densidad (DBSCAN), por niveles (Jerárquico), y otros mas específcos a la aplicación.

Para la extracción de bicliques, una vez se obtiene el conjunto de signatures, se busca obtener los clusters. Para eso se ordenan los elementos por columnas, con el fin de agrupar los vértices que tengan listas de adyacencia similares. Entre mas columnas coincidan significa que habrá mas similitud entre las aristas de los vértices. Este paso esta representado en las tablas 4.3a y 4.3b. Luego, por cada cluster obtenido, se obtiene la frecuencia de las aristas de cada vértice en el cluster. A partir de esta frecuencia se ordenan las listas de adyacencia de mayor a menor frecuencia de las aristas en el cluster, es decir, dejando primero los vértices que tengan la arista con mayor frecuencia (en caso de empate se considera el identificador del vértice). Por otra parte, se descartan los vértices que tengan aristas con frecuencia 1.

Vértice	Valor hash 1	Valor hash 2	Valor hash 3
1	1760843776	189794601	2367512841
4	1760843776	189794601	2367512841
5	1760843776	189794601	3100360591
7	1760843776	189794601	3100360591

(a) Ejemplo 1 de un posible cluster.

Vértice	Valor hash 1	Valor hash 2	Valor hash 3
2	1625095119	16656699	2367512841
3	1625095119	16656699	2405980371
6	1625095119	16656699	2405980371
8	1625095119	16656699	2405980371

(b) Ejemplo 2 de un posible cluster.

Tabla 4.3: Posibles clusters

4.2.3. Árbol de prefijos

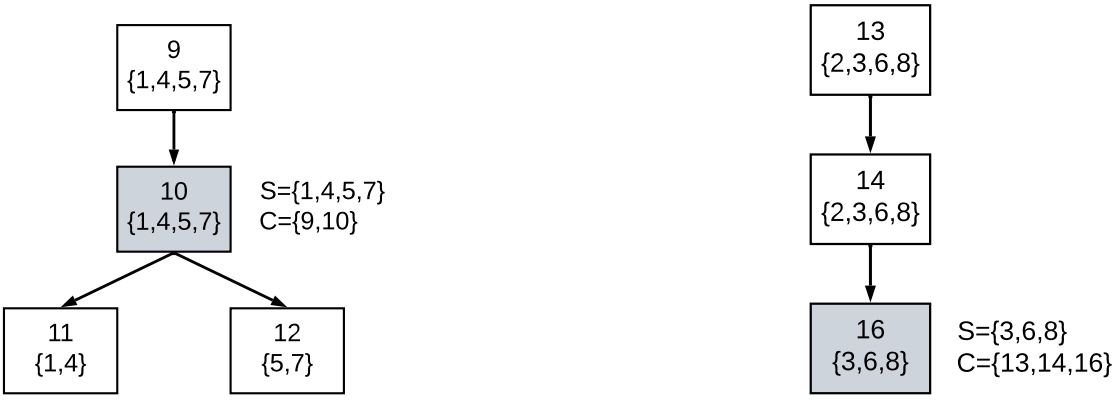
Un *árbol de prefijos* es un tipo de árbol k-ario, una estructura de datos que utiliza símbolos en cada nodo para posteriormente lograr búsquedas de secuencias de símbolos. Se puede utilizar para representar distintos tipos de símbolos y secuencias como, por ejemplo, caracteres y palabras o números enteros para encontrar una secuencia de números.

Para esta aplicación se crea un árbol de prefijos para cada cluster encontrado, donde la construcción consiste en procesar las listas de adyacencia ordenadas por frecuencia del cluster. Para cada lista de adyacencia, si la arista no se encuentra en la rama actual se crea un nodo, se agrega el id del vértice a un vector en el nodo en el árbol indicando que el vértice tiene ese arista. Además se agrega una arista del nodo padre en la rama al nodo creado. Por otro lado, si en la rama actual la arista procesada ya se encuentra en el árbol solo se agrega el vértice al vector en el nodo, indicando que ese vértice también tiene la arista siendo procesada. Una vez construido el árbol de

prefijos se recorre el árbol obtieniendo las ramas que contienen mas vértices en común y se extraen los bicliques del árbol. El procedimiento mas detallado se proporciona a continuación:

1. Cada hoja del árbol tiene como etiqueta la arista correspondiente a la última arista (con menor frecuencia) en alguna lista de adyacencia y como raíz la arista con mayor frecuencia (primera posición). Además cada nodo en el árbol tiene un vector con los vértices que poseen la arista.

2. Una vez construido el árbol de prefijos, se procede a encontrar el mejor biclique presente. Para ello se recorre el árbol buscando aquella hoja que tenga el mejor coeficiente de compresión. Esta se define como $Z(x) = depth \times vsize$, donde $depth$ es la profundidad de la hoja dentro del árbol y $vsize$ es el tamaño del vector de vértices. Esto se traduce a que mayor profundidad y mayor cantidad de vértices en una arista profunda, es mas grande el biclique encontrado. La Figura 4.2 proporciona un ejemplo donde se observa lo descrito.



(a) Cluster 1. (b) Cluster 2.

Figura 4.2: Árbol de prefijos de cada cluster, el biclique del primer cluster tiene tamaño 8 (2·4) y el segundo cluster tiene tamaño 9 (3·3).

4.3. Multiplicación sin bicliques

4.3.1. Algoritmo de Schoor

La multiplicación que propone Schoor es para dos matrices ralas o dispersas, el cual tiene una complejidad de tiempo promedio mejor que los algoritmos ya conocidos. Sea A una matriz de $M \times N$ y B otra matriz de $N \times K$, la multiplicación entre ambas matrices tiene un tiempo promedio $O(\frac{a \cdot b}{N})$, donde a es la cantidad de aristas de la matriz A y b la cantidad de aristas de B.

El algoritmo ocupa una lista enlazada ortogonal como estructura de datos para las matrices. Para la matriz A, por cada elemento A_{ij} que no sea cero es representada por un nodo $a(i, j)$ que contiene los siguientes campos: el valor de A_{ij}, el índice de la fila i, el índice de la columna j y dos punteros a los siguientes elementos no cero en la fila y la columna respectivamente. También existe un vector de punteros **Filas_A**, el cual apunta al primer elemento no cero de cada fila. Análogamente existe un vector de punteros **Columnas_A**. Esta estructura se utiliza de igual manera para la matriz B. Cabe destacar que el espacio utilizado es $5 \cdot a + M + N$ para la matriz A.

El algoritmo sigue una idea simple, por cada elemento A_{ij} que no sea cero, es multiplicado por todos los elementos no cero de la j-esima fila de B. De forma mas general, por cada elemento A_{it}, el algoritmo busca la t-ésima fila de B y multiplica cada elemento B_{tj} que no sea cero de esa fila. Cada producto es agregado al nodo $c(i, j)$ de la matriz resultante C.

Adicionalmente, para acceder y actualizar la matriz C mas rápido, se utilizan vectores de punteros auxiliares **Final_columna** y **Final_fila**, los cuales apuntan al último elemento no cero de cada columna y cada fila respectivamente. Al inicio del algoritmo, estos vectores se inicializan en nulo. Este algoritmo se puede ver en Algoritmo 1.

Algoritmo 1: Multiplicación de matrices ralas de Schoor

Entrada: Matrices A y B

Salida : Matriz C

```
1  para i ← 1 a M hacer
2  |  Fila_C[i] ← Final_fila[i] ← nulo
3  para j ← 1 a K hacer
4  |  Columna_C[i] ← Final_columna[i] ← nulo
5  para i ← 1 a M hacer
6  |  para cada elemento A_{i,t} en Fila_A[i] hacer
7  |  |  para cada elemento B_{t,j} en Fila_B[t] hacer
8  |  |  |  si la fila del elemento de C apuntado por Final_columna[j]=i
   |  |  |   entonces
9  |  |  |  |  C_ij ← C_ij + A_it · B_tj
10 |  |  |  en otro caso
11 |  |  |  |  se crea un nuevo nodo C_ij ← A_it · B_tj
12 |  |  |  |  si Final_columna[j] = nulo entonces
13 |  |  |  |  |  Columna_C[j] ← C_ij
14 |  |  |  |  en otro caso
15 |  |  |  |  |  se enlaza C_ij como el siguiente elemento en la columna de
   |  |  |  |  |   Final_columna[j]
16 para j ← 1 a K hacer
17 |  para por cada elemento C_ij en Columna_C[j] hacer
18 |  |  si Final_fila[i] = nulo entonces
19 |  |  |  Fila_C[i] ← C_ij
20 |  |  en otro caso
21 |  |  |  colocamos el nodo apuntado por Final_fila[i], apuntar hacia C_ij
   |  |  |   como el siguiente elemento de la fila
22 |  |  Final_fila[i] ← C_ij
```

4.3.2. Multiplicación de Schoor utilizando CSR y CSC

El algoritmo implementado es el del artículo de Arroyuelo [2], el cual es una adaptación del algoritmo de Schoor [21] vista en la sub-sección anterior 4.3.1 que combina los formatos CSR y CSC [[20], Sec. 3.4] para la representación de las matrices. Esta implementación es de Gonzalo Navarro y está optimizada en espacio y tiempo, donde la complejidad de tiempo es $O(\frac{a \cdot b \cdot \log N}{N})$. Este algoritmo utiliza la representación CSR consistiendo en tres vectores de enteros. El primer vector **Filas** almacena las filas no vacías, el segundo **FilasPos** registra las posiciones iniciales del tercer vector y este último **ColumnasPorFila** guarda, para cada fila no vacía, las columnas no vacías ordenadas de manera ascendente. Análogamente, para la representación CSC se usan tres vectores adicionales que almacenan la misma información, pero para las columnas en lugar de las filas (**Columnas**, **ColumnasPos** y **FilasPorColumna**), esta es la representación de la transpuesta. La Figura 4.3 presenta un ejemplo de la representación de una matriz booleana y su representación usando la estructura CSR y CSC. Además, la matriz completa, con ambas representaciones se pueden ver en la estructura 2.

Algoritmo 2: Estructura utilizada para representar la matriz como CSR y CSC.

```
1  Estructura matrix contiene
2      uint64_t elems
3      uint width, height
4      uint nrows
5      uint *rowids
6      uint64_t *rowpos
7      uint *colsbyrow
8      uint ncols
9      uint *colids
10     uint64_t *colpos
11     uint *rowsbycol
```

Para la operación de multiplicación consideremos lo siguiente:

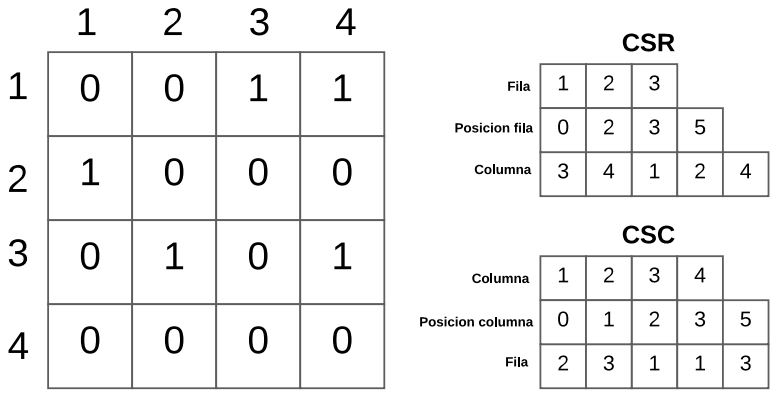


Figura 4.3: Ejemplo de una matriz representada con CSR y CSC adaptadas para matrices booleanas.

Sea A una matriz $M \times N$, la cual tiene m filas no vacías y n columnas no vacías, y sea B otra matriz de $N \times K$, con n filas no vacías y k columnas no vacías. Definimos la multiplicación en 3 partes, donde las dos primeras se centran principalmente en obtener la representación *CSR* y la última construye el *CSC*.

1. **Intersección y generación de productos cartesianos:** Primero, se identifican las columnas de la matriz A que se intersectan con las filas de la matriz B. Esta intersección ocurre cuando encontramos una columna t en el vector **Columnas** de A y una fila t en el vector **Filas** de B. El resultado de este proceso nos proporciona r intersecciones. Para encontrar estas intersecciones se utiliza búsqueda lineal o exponencial, dependiendo de la diferencia de tamaño de los vectores.

   Cuando encontramos una intersección en t, encontramos la columna y fila donde se origina el producto $A_{it} \cdot B_{tj}$. A partir de esto podemos obtener todas las filas i de A y todas las columnas j de B, mediante los vectores **ColumnasPos**, **FilasPorColumnas** de A y **FilasPos**, **ColumnasPorFila** de B respectiva-

mente, y así también obtener el tamaño del vector. Al tener varias filas y columnas, obtenemos un producto cartesiano. Por cada intersección encontrada, utilizaremos una estructura llamada *task*.

La estructura *task* se encarga de almacenar todos los productos cartesianos mediante punteros, lo que lo hace eficiente en espacio al momento de procesarlos. Además ayuda en el siguiente paso para procesar las aristas.

2. **Procesamiento de productos cartesianos:** Como se describe en el paso anterior, los productos cartesianos obtenidos serán las aristas que tendrá nuestra matriz resultante. Aquí puede resultar un problema con las aristas, dado que una arista i, j puede encontrarse en uno, dos o en varios productos cartesianos, por lo que no aporta valor en el contexto de matrices booleanas y genera dificultad para procesarla.

   Para esto se utilizan dos *Min-heaps*, uno para procesar las filas y otro para procesar las columnas de la fila a tratar. Esta estructura puede verse en el Algoritmo 4. El *heap* de filas es de tamaño correspondiente a la cantidad de intersecciones (r), donde la llave es la primera fila de una *task* y el dato es el índice de la *task*. Al tener todas las *task* dentro del *heap* de filas, empezamos a procesar fila por fila.

   Primero, obtenemos el primer elemento del *Min-heap* de filas, sacamos la fila y gracias a que guardamos el índice de la *task* asociada, guardamos las columnas en el *heap* de columnas junto al índice de las *task*. Luego, avanzamos a la siguiente fila del elemento obtenido y obtenemos nuevamente el primer elemento del *Min-heap* de filas, si la fila obtenida es igual a la anterior, seguimos agregando columnas al *heap* de columnas. Caso contrario, terminamos el proceso de construir el *heap* de columnas. Cabe destacar que el elemento del heap se queda sin columnas, y luego se elimina este objeto. Al final de este proceso se guarda en un vector resultante **Filas** la fila procesada.

   Una vez que obtenemos todas las columnas a procesar, hacemos el mismo proceso para las columnas, y cada vez que aparezca una nueva columna la

agregamos al vector resultante **ColumnasPorFila**. Después de procesar todas las columnas de una fila, modificamos el vector de posiciones **FilasPos** para que concuerde la fila obtenida con sus respectivas columnas.

Por otra parte, un proceso intermedio que se hace es contar cuantas veces aparece las columnas, esto se hace con un vector **colc**, el cual ayuda para el siguiente paso de construir la transpuesta. Todo este paso se puede observar en el algoritmo 5.

3. **Construcción de CSC:** Finalmente, una vez obtenida la representación *CSR*, usando el vector **colc** y los vectores del *CSR* podemos construir esta representación en tiempo proporcional al resultado obtenido anteriormente. Para esto podemos revisar el algoritmo 6.

Algoritmo 3: Estructura utilizada para las task.

```
1 Estructura task contiene
2     uint ncols, nrows; // tamaño de los vectores
3     uint *cbyr, *rbyc; // vectores columnas de las filas y filas de las
        columnas
4     uint pcbyr; // variable auxiliar que ayuda a procesar el heap
```

Algoritmo 4: Estructura utilizada para el Min-Heap.

```
1 Estructura heap contiene
2     uint key;
3     uint data;
```

Algoritmo 5: Construcción CSR

Entrada: $Hr, Hc, task, colc$ // Hr = heap row, Hc = heap column

Salida : $rowids, rowpos, colsbyrow, colc$

```
1  pr ← 0, prc ← 0 // contador del vector rowids y colsbyrow
2  mientras p hacer // Mientras existan task
3      m ← 0
4      hacer// Inicializamos el heap de columnas
5          hr ← Hr.top()
6          i ← hr.data
7          Hc[m].key ← task[i].colums
8          Hc[m].data ← i
9          task[i].pcbyr ← 1
10         m++
11         si quedan filas por procesar entonces
12             Hr.reemplazarMin(p, task[i],rbyc++,i)
13         en otro caso
14             Hr.extraerMin(p−−)
15     mientras p and min(Hr).key = hr.key;
16     rowids[pr] ← hr.key, rowpos[pr] ← prc
17     prc++
18     mientras m hacer
19         hc ← min(Hc)
20         i ← hr.data
21         colsbyrow[prc++] ← k
22         colc[k]++
23         hacer
24             si task[i].pcbyr < task[i].ncols entonces
25                 Hc.reemplazarMin(m,task[i].cbyr[task[i].pcbyr++],i)
26             en otro caso
27                 Hc.extraerMin(m−−)
28             hc ← Hc.findMin(m)
29             i ← hc.data
30         mientras hc.key = k;
```

Algoritmo 6: Construcción CSC

Entrada: $prc, colc, rowids, rowpos, colsbyrow$

Salida : $colids, colpos, rowsbycol$

```
1  pc ← 0
2  para i ← 0 a n hacer
3  |  si colc[i] entonces
4  |  |  colids[pc++] ← i
5  colspos[0] ← 0
6  para i ← 1 a pc hacer
7  |  colpos[i] ← colpos[i − 1] + colc[colids[i − 1]]
8  colpos[pc] ← prc
9  para pr ← 0 a nrows hacer
10 |  f ← rowpos[pr + 1]
11 |  r ← rowids[pr]
12 |  para prc ← rowpos[pr] a f hacer
13 |  |  rowsbycol[colc[colsbyrow[prc]]++] ← r
```

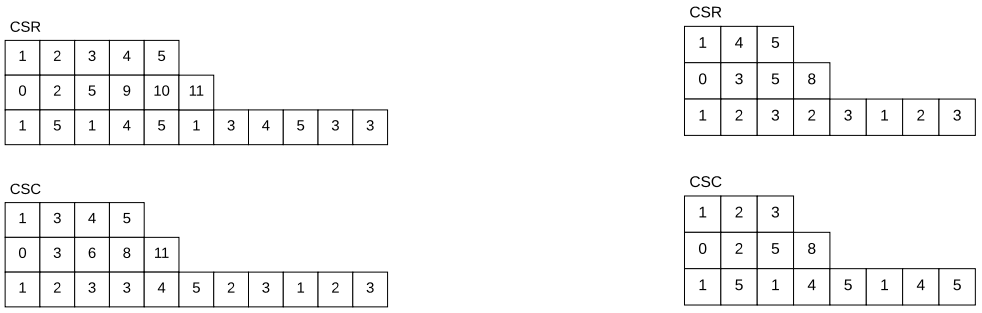
	1	2	3	4	5
1	1	0	0	0	1
2	1	0	0	1	1
3	1	0	1	1	1
4	0	0	1	0	0
5	0	0	1	0	0

	1	2	3	4	5
1	1	1	1	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	1	1	0	0
5	1	1	1	0	0

Figura 4.4: Matrices A1' y A2' respectivamente.

Considerando las matrices $A1'$ y $A2'$ mostradas en la figura 4.4, se presentan los pasos a seguir para obtener el resultado de una multiplicación:

1. Considerando estas matrices, tenemos tres intersecciones, que son el 1, 4 y el



(a) Matriz A. (b) Matriz B.

Figura 4.5: Matrices A1' y A2' con su representación CSR Y CSC.

5. A partir de estas intersecciones, obtenemos tres productos cartesianos, por el lado izquierdo tendremos las filas de la columna intersectada y por el otro lado tendremos las columnas de la fila. En este caso tendremos tres productos:

- 1 2 3 × 1 2 3
- 2 3 × 2 3
- 1 2 3 × 1 2 3

Este paso y el anterior, se pueden observar en las figuras 4.6 y 4.7.

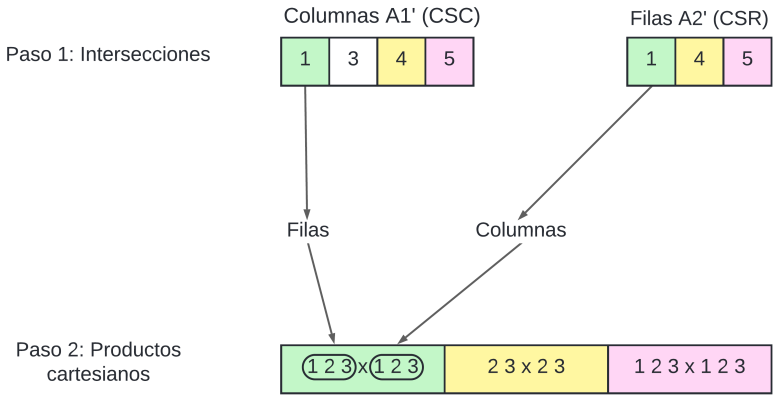


Figura 4.6: Obtención de productos cartesianos.

	1	2	3	4	5
1	1	0	0	0	1
2	1	0	0	1	1
3	1	0	1	1	1
4	0	0	1	0	0
5	0	0	1	0	0

	1	2	3	4	5
1	1	1	1	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	1	1	0	0
5	1	1	1	0	0

Figura 4.7: Productos cartesianos, vista matricial de A1' y A2'.

2. Como se puede observar, tenemos redundancia en la repetición de aristas como se menciono anteriormente. Debemos eliminarla de alguna forma, para esto iremos procesando en orden las filas y por cada fila ordenamos las columnas. La primera fila a procesar es la fila 1, dado que es la menor. A partir de esto, tomamos todos los productos cartesianos que tengan esta fila y comenzaremos a tomar la columna menor y se la agregamos al CSR, luego la segunda y así sucesivamente hasta pasar por todas las columnas. Posteriormente eliminamos esa fila y continuamos con la siguiente hasta que no queden mas filas por procesar. Esto se puede ver en las figuras 4.8, 4.9 y 4.10.

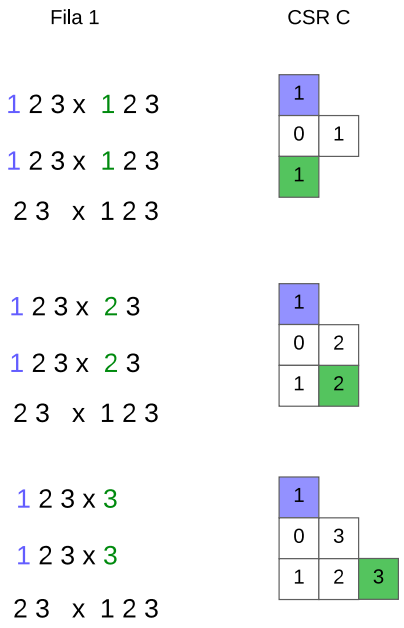


Figura 4.8: Construcción CSR Fila 1.

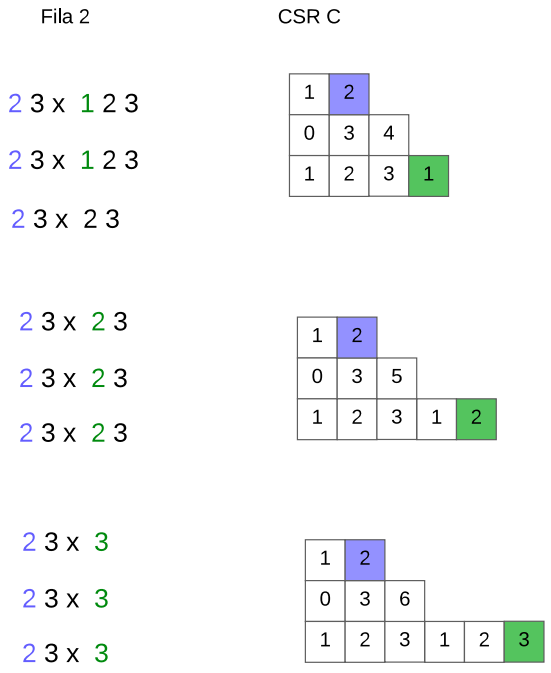


Figura 4.9: Construcción CSR Fila 2.

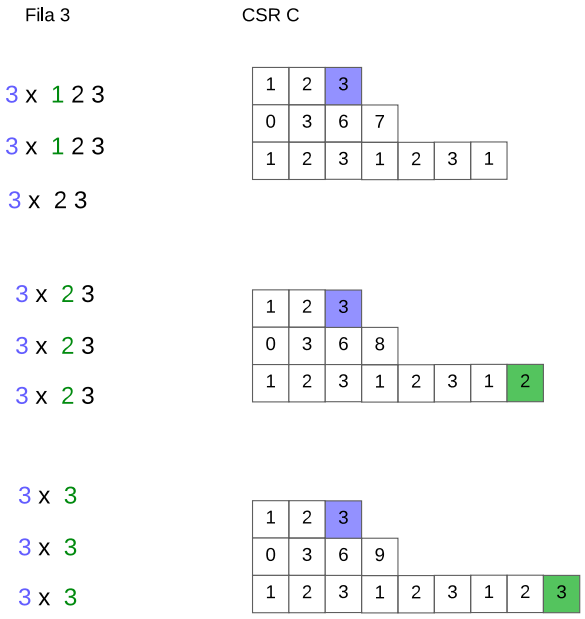


Figura 4.10: Construcción CSR Fila 3.

3. En este ejemplo específico, el *CSC* nos queda de la misma manera que el *CSR* obtenido anteriormente.

4.4. Multiplicación con bicliques

Así como anteriormente multiplicamos dos matrices $A1'$ y $A2'$, también podemos considerar multiplicar utilizando sus bicliques para obtener una reducción en espacio dado a la compresión obtenida por la representación implícita de aristas. Para esto redefinimos la matriz A como

$$A = A' + B \tag{4.2}$$

donde B esta determinado como 2.2, y A' es el resto de la matriz, o visto de otra forma

$$A' = A - \{B_1, B_2, ..., B_n\} \tag{4.3}$$

Tomando en cuenta las matrices A_1 y A_2, con B_1 y B_2 sus respectivos bicliques y A'_1 y A'_2 sus restos de la matriz, definimos la multiplicación como

$$\begin{aligned} A_1 \cdot A_2 &= (A'_1 + B_1) \cdot (A'_2 + B_2) \\ &= A'_1 \cdot A'_2 + A'_1 \cdot B_2 + B_1 \cdot A'_2 + B_1 \cdot B_2 \\ &= A'_1 \cdot A'_2 + A'_1 \cdot (B_{21} + B_{22}+, ..., +B_{2m}) + (B_{11} + B_{12}+, ..., +B_{1n}) \cdot A'_2 \\ &+ (B_{11} + B_{12}+, ..., +B_{1n}) \cdot (B_{21} + B_{22}+, ..., +B_{2m}) \end{aligned} \tag{4.4}$$

Para esta aplicación, vamos a considerar que A'_1 y A'_2 de la figura 4.4 son el resto de las matrices $A1$ y $A2$ respectivamente. Además, A_1 se le han extraído los bicliques (4,5 - 1,2), (4,5 - 4,5) y para A_2 (1,2,3 - 4,5), (4,5 - 4,5), esto se puede ver en la figura 4.11. En este caso, como tenemos 2 bicliques en cada matriz, b1 = 2 y b2 = 2, obteniendo:

$$A'_1 \cdot A'_2 + A'_1 \cdot (B_{21} + B_{22}) + (B_{11} + B_{12}) \cdot A'_2 + (B_{11} + B_{12}) \cdot (B_{21} + B_{22}) \tag{4.5}$$

Algoritmo 7: Estructura utilizada para representar la matriz como CSR y CSC, incluyendo sus bicliques.

```
1  Estructura matrix_with_bicliques contiene
2      uint64_t elems
3      uint width, height
4      uint nrows
5      uint *rowids
6      uint64_t *rowpos
7      uint *colsbyrow
8      uint ncols
9      uint *colids
10     uint64_t *colpos
11     uint *rowsbycol
12     biclique bics
13     uint nbics
14     bool truebic
```

Algoritmo 8: Estructura que representa un biclique.

```
1  Estructura biclique contiene
2      uint *rows;
3      uint *cols;
4      uint nrows;
5      uint ncols;
```

En general, para todas las operaciones que veremos a continuación utilizaremos los pasos vistos en la sección 4.3.2, donde el único paso que se diferencia entre operaciones es el paso de intersección (paso 1).

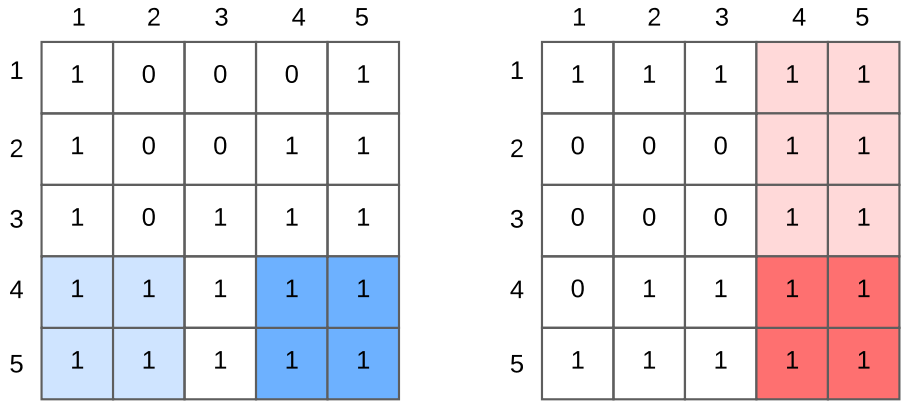


Figura 4.11: Matrices A1 y A2 respectivamente, donde se representan los bicliques extraídos en color azul y rojo.

4.4.1. Matriz × Matriz

Sean X e Y, dos matrices de $M \times N$ y $N \times K$ respectivamente. Luego, podemos ocupar exactamente todos los pasos vistos en la sección 4.3.2, dado que se mantiene la operación de multiplicar matrices.

Para la multiplicación $A'_1 \cdot A'_2$, obtenemos el mismo resultado visto en la figura 4.10.

4.4.2. Matriz × Biclique

Sea X una matriz $M \times N$ y B_1 un biclique que representa $|S_1| \cdot |C_1|$ aristas. Para esta operación, buscamos las intersecciones entre las columnas del vector **Columnas** de X y las filas del biclique B1(S_1). Por cada intersección t encontrada, obtenemos el producto cartesiano (filas de la columna t × todas las columnas del biclique (C_1)). En caso de tener mas de un biclique, simplemente sumamos los productos cartesianos obtenidos.

En este ejemplo tenemos 2 multiplicaciones que resolver, $A'_1 \cdot B_{21}$ y $A'_1 \cdot B_{22}$, o visto de otra forma $A'_1 \cdot ((1,2,3),(4,5))$ y $A'_1 \cdot ((4,5),(4,5))$. Para el primer producto intersectamos las columnas de la matriz A'_1 (**Columnas**) y las filas del biclique B_{21} (S_{21}), donde obtenemos dos productos cartesianos a partir de 1 y 3. Los productos cartesianos están compuestos a partir de las filas de la matriz A'_1, específicamente de

las columnas intersectadas y las columnas serán los C del biclique (C_{21}) para ambos casos. Como resultado nos quedan como productos cartesianos:

1. $1, 2, 3 \times 1, 2$
2. $3, 4, 5 \times 1, 2$

Análogamente, intersectamos las columnas de la matriz A'_1 (**Columnas**) y las filas del biclique B_{22} (S_{22}), acá vemos que las intersecciones se dan en 4 y 5. Para este caso tenemos los siguientes productos cartesianos.

1. $2, 3 \times 4, 5$
2. $1, 2, 3 \times 4, 5$

A partir de estos cuatro productos cartesianos obtenidos, podemos aplicar los pasos 2 y 3 de la multiplicación sin bicliques, en la sección 4.3.2, para obtener como resultado una matriz con formato CSR y CSC.

4.4.3. Biclique × Matriz

Esta operación es muy similar a la anterior, dado que involucra tanto a la matriz como al biclique, pero este caso es diferente, dado que la multiplicación de matrices no es conmutativa. Sea B_2 un biclique que representa $|S_2| \cdot |C_2|$ aristas e Y una matriz de tamaño $N \times K$, buscamos intersectar todas las columnas del biclique B_2 (C_2) con todas las filas del vector **Filas** de la matriz Y. Por cada intersección, obtenemos como resultado el producto cartesiano (todas las filas del biclique (S_2) × columnas de la fila t).

En esta situación tenemos $B_{11} \cdot A'_2 + B_{12} \cdot A'_2$ o $((4, 5), (1, 2)) \cdot A'_2 + ((4, 5), (4, 5)) \cdot A'_2$. Primero obtenemos las intersecciones de las columnas del biclique B_{11} (C_{11}) con las filas de la matriz A'_2 (CSR), donde obtenemos una única intersección en 1. Por otra parte, para el segundo producto aplicamos lo mismo con las columnas del biclique B_{12} (C_{12}). De acá se obtienen dos productos cartesianos de 4 y 5, dándonos en total tres para la operación completa.

1. $4, 5 \times 1, 2, 3$

2. $4, 5 \times 2, 3$
3. $4, 5 \times 1, 2, 3$

4.4.4. Biclique × Biclique

Sean B_1 un biclique que representa $|S_1| \cdot |C_1|$ aristas y B_2 un biclique que representa $|S_2| \cdot |C_2|$ aristas. Para esta multiplicación buscamos intersectar las columnas del primer biclique (C_1) con las filas del segundo (S_2). Para este caso, nos basta solamente con encontrar una sola intersección, dado que si lo hacemos varias veces, crearemos varios productos cartesianos igual, ralentizando el algoritmo general. En caso de haber intersección, obtenemos el producto $S_1 \times C_2$.

Siguiendo el ejemplo, nos queda los productos entre bicliques. Desarrollando nos quedan estos cuatro productos:

$$(B_{11} + B_{12}) \cdot (B_{21} + B_{22}) = B_{11} \cdot B_{21} + B_{11} \cdot B_{22} + B_{12} \cdot B_{21} + B_{12} \cdot B_{22} \quad (4.6)$$

Tomando en consideración la primera multiplicación $((4, 5), (1, 2)) \cdot ((1, 2, 3), (4, 5))$ se intersecta $(1, 2)$ (C_{11}) con $(1, 2, 3)$ (S_{21}). Como si hay intersección, obtenemos el producto cartesiano $S_{11} \times C_{21}$, caso contrario, no hay producto y pasamos a la siguiente multiplicación.

4.4.5. Multiplicación completa

En caso de que queramos obtener el resultado completo, simplemente agrupamos todos los productos cartesianos obtenidos a partir de matriz×matriz, matriz×biclique, biclique×matriz, biclique×biclique en un solo conjunto. Una vez que tenemos todos los productos cartesianos juntos, aplicamos el algoritmo visto en la sección anterior (algoritmo 5 y 6). A esta operación, la cual nos da como resultado solamente una matriz la llamaremos m.

También podemos acelerar el proceso de la multiplicación dejando unos productos cartesianos como bicliques, esto siempre y cuando sean lo suficientemente grandes.

Para esto, definimos un parámetro *gb* que elige si un producto es un biclique atractivo o se procesa junto a los demás. Este parámetro se regirá por multiplicación entre el número de filas y de columnas de un producto. Además, las estrellas nunca serán consideradas un buen biclique. Esta forma de dejar expresado el resultado como una matriz mas una colección de bicliques será definida como

$$(mb)$$

En el caso que se quiera transformar el resultado de una matriz con bicliques a una sola matriz, basta simplemente con agrupar los producto cartesianos de la matriz con los productos de los bicliques y aplicar nuevamente el algoritmo 5 y 6. Los productos cartesianos de una matriz serán cada vértice $\times$ sus respectivas aristas. Esta transformación sera llamada T.

4.5. Conteo de triángulos

En esta sección se utiliza el algoritmo de multiplicación de matrices para el conteo de triángulos.

Dado un grafo no dirigido $G = (V, E)$, el problema de conteo de triángulos busca computar el número de triángulos que contiene un grafo, el cual es importante para computar el coeficiente de *clustering* global. Cabe destacar que se aplica solamente para matrices cuadradas.

El algoritmo para contar triángulos usado se compone principalmente de dos pasos. Primero se calcula el cuadrado de la matriz y posteriormente se multiplican los valores de la matriz original con el cuadrado ($A_{ij} \cdot A_{ij}^2$). Antes que nada, hay que tomar un par de consideraciones antes de entender el algoritmo completo. En este caso, el cuadrado almacena la suma, por lo que se asemeja a lo que es la multiplicación con pesos, aunque en este caso sean todos los valores 1 (ver ec. 4.7). Por otro lado, la multiplicación $A_{ij} \cdot A_{ij}^2$ puede considerase como una búsqueda en ambas matrices donde coincidan ambos índices i, j, y solamente sumar el valor de A_{ij}^2, dado que el valor de A_{ij} es 1.

$$(A \cdot A)_{ij} = \sum_{k=1}^{n} A_{ik} \cdot A_{kj} \tag{4.7}$$

Algoritmo 9: Conteo de triángulos

Entrada: matriz A de M × N

Salida : cantidad de triángulos encontrados

```
1 count ← 0
2 A² ← A · A
3 para i ← 1 a n hacer
4 |   para j ← 1 a n hacer
5 |   |   count ← count + A_ij · (A²)_ij
6 devolver count/6
```

El algoritmo se muestra en el pseudo-código 9. Podemos observar que para el cuadrado, podemos ocupar las dos multiplicaciones vistas anteriormente, multiplicar con o sin bicliques. Para ambos casos necesitamos editar la estructura para la matriz resultante, agregando un vector, el cual guarde el valor de la celda de la matriz resultante. Además, dado que solamente queremos saber el número de triángulos de la matriz, podemos quitar el CSC de la estructura. Esto se puede ver en la estructura 10.

Algoritmo 10: Estructura utilizada para representar la matriz como CSR, donde se agrega un vector **count** para guardar la suma de la multiplicación.

```
1 Estructura matrix_triangles contiene
2 |   uint64_t elems
3 |   uint width, height
4 |   uint nrows
5 |   uint *rowids
6 |   uint64_t *rowpos
7 |   uint *colsbyrow
8 |   uint *count
```

4.5.1. Versión sin bicliques

Para esta versión, el algoritmo de multiplicación queda exactamente igual. Sin embargo, al momento de encontrar por primera vez una nueva columna, se establece la suma en 0 en el vector **count[i]** y se suma 1 cada vez que se repita la columna. En este contexto, i es el índice de la columna procesada en el vector **ColumnaPorFila**.

4.5.2. Versión con bicliques

Para este método, consideramos todas las intersecciones vistas anteriormente, a excepción de biclique × biclique, dado que en esta solamente consideramos una intersección. Para este caso debemos considerar todas las intersecciones, y para no aumentar demasiado la cantidad de *task*, provocando la ralentización del proceso de creación de la matriz, agregaremos un atributo *nreps* a la estructura que cuente la cantidad de veces que se repite un producto cartesiano.

Algoritmo 11: Estructura utilizada para las task al contar triangulos.

```
1 Estructura task contiene
2 |  uint nreps; uint ncols, nrows; // tamaño de los vectores
3 |  uint *cbyr, *rbyc; // vectores columnas de las filas y filas de las
  |   columnas
4 |  uint pcbyr; // variable auxiliar que ayuda a procesar el heap
```

Para los productos matriz × matriz, matriz × biclique y biclique × matriz, cuando se cree una *task*, se coloca *nreps* en 1. En el caso de biclique × biclique, *nreps* indicara la cantidad de intersecciones que se encuentran y por lo tanto, la cantidad de veces que se repite el producto cartesiano obtenido.

Finalmente, para ambas versiones se buscan los índices i, j que coincidan en ambas, en caso de existir, se agrega el valor de esa arista a la suma.

Para ejemplificar como funciona el algoritmo, utilizaremos la matriz que se aprecia en la figura 4.12. Primero, se computa el cuadrado y posteriormente se encuentran las celdas que comparten con la matriz original. En la figura 4.13 observamos que

en color amarillo se muestran las celdas mencionadas. Finalmente sumamos todas las celdas amarillas que encontramos en el cuadrado y dividimos por 6. Esto nos da un total de 3 triángulos, lo cual observando el grafo en 4.12, podemos confirmar que es correcto.

	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	0	1
3	1	1	0	1	1
4	1	0	1	0	1
5	0	1	1	1	0

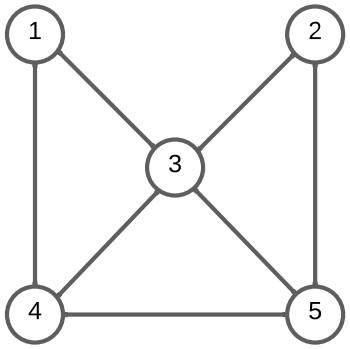


Figura 4.12: Matriz que se usará para contar los triángulos, con su respectiva representación gráfica, el cual contiene 3 triángulos.

	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	0	1
3	1	1	0	1	1
4	1	0	1	0	1
5	0	1	1	1	0

	1	2	3	4	5
1	2	1	1	1	2
2	1	2	1	2	1
3	1	1	4	2	2
4	1	2	2	3	1
5	2	1	2	1	3

Figura 4.13: Matriz original y su respectiva potencia, además se marca en color amarillo las celdas que comparten.

Capítulo 5

Resultados

5.1. Configuración de los experimentos

A continuación se presentan las condiciones y entorno bajo los cuales se realizaron los experimentos.

5.1.1. Máquina

Las pruebas y experimentos fueron desarrolladas en combinación de C y C++ utilizando los compiladores gcc y g++, con -std=c++17 -O3 como *flags* de compilación en el caso de C++, -O3 solamente en el caso de C. Además, para su ejecución, se utilizó un servidor del Laboratorio VLSI del Departamento de Ingeniería Eléctrica de la Universidad de Concepción que cuenta con un Intel(R) Xeon(R) Gold 5118 CPU @2.30GHz junto con 92GB de memoria principal utilizables.

5.1.2. Datasets

Para la evaluación de la extracción y multiplicación se utilizaron datos reales y de acceso público. Para esto, se recurrió a 3 páginas web que contienen una gran cantidad de datasets. Primero tenemos a *The Laboratory of Web Algorithmics* (LAW), luego a *Stanford Network Analysis Platform* (SNAP) [19] y por último *Koblenz Network*

Collection (KONECT) [18].Todos los datasets son dirigidos y pueden observarse en la tabla 5.1.

Dataset	$\|V(G)\|$	$\|E(G)\|$
web-Stanford	281.903	2.312.497
cnr-2000-hc	325.557	3.216.152
wikipedia link lmo	52.214	3.623.678
web-Google	875.713	5.105.039
indochina-2004	7.414.866	194.109.311
arabic-2005-hc	22.744.080	639.999.458

Tabla 5.1: Datasets a utilizar en los resultados.

En cambio, para la evaluación del conteo de triángulos se utilizó principalmente una página. Para esto se utilizó *The SuiteSparse Matrix Collection* [10] para obtener datasets reales. Todos los grafos utilizados son no dirigidos y se pueden observar en la figura 5.2.

Dataset	$\|V(G)\|$	$\|E(G)\|$	N triangulos
msc23052	23.052	1.154.814	5.995.917
msc10848	10.848	1.229.778	14.839.758
crankseg_1	16.656	1.430.006	18.063.405
sme3Dc	42.930	3.148.656	23.177.851
crankseg_2	36.849	4.894.630	75.417.540
hood	220.542	10.768.436	55.567.372
coPapersCiteseer	434.102	32.073.440	156.212.415
audikw	943.695	77.651.847	602.326.705

Tabla 5.2: Datasets a utilizar en los resultados de conteo de triángulos.

5.2. Evaluación del algoritmo extractor de bicliques

5.2.1. Metodología de evaluación

La metodología incluye el estudio de distintos parámetros para ver el rendimiento en la extracción de bicliques, observando el impacto en el tiempo de ejecución y en la compresión del grafo. En esta sección primero se muestran los resultados de los experimentos realizados con una configuración que busca extraer bicliques lo mas grandes posibles. Posteriormente se utilizan los parámetros que intentan sacar una mayor cantidad y mejor ganancia de aristas en bicliques, sin importar el tamaño del biclique extraído, y finalmente ver como afecta mas tarde a la multiplicación.

Para la extracción de bicliques de cada grafo, se utiliza un tamaño de biclique inicial *TBI*, donde si el tamaño del biclique a extraer es menor a *TBI*, este no es considerado. Al terminar una iteración, se sacan los bicliques del grafo y el algoritmo continua con este grafo de menor tamaño. Por otro lado, si la cantidad de bicliques no supera un valor *threshold*, disminuirá el tamaño del biclique a extraer *DTB*. También existe la posibilidad de modificar la cantidad de signatures a ocupar, el número de *shingles* y el tamaño mínimo del cluster para ser considerado.

Los valores a considerar para ambas pruebas son:

- Número de *signatures*: 2
- Mínimo del tamaño del cluster: 4
- Tamaño del *Shingle*: 1
- *Threshold*: 100

El tamaño del biclique inicial y la disminución dependerá completamente de cada grafo y este estará especificado en las tablas 5.4 y 5.6. Aun así, uno de los experimentos tiene un *TBI* del 10 % del tamaño del otro para todos los datasets, es decir, se busca encontrar mas bicliques aunque sean mas pequeños.

Los resultados obtenidos a partir de los experimentos son agrupados en tres tablas diferentes:

1. La primera tabla muestra estadísticas únicamente enfocadas en los bicliques extraídos, tales como la cantidad de bicliques, la cantidad de vértices representados en S y en C. También se encuentra la suma de ambos grupos, que representa la compresión hecha por los bicliques, y por último la multiplicación de estos grupos es la cantidad de aristas originales las cuales son comprimidas.

2. En la segunda tabla encontraremos los parámetros que fueron variados para estos experimentos tales como TBI y DTB, los cuales fueron mencionados anteriormente. Por otra parte tendremos dos coeficientes de compresión distintos. El primero es K, el cual es la división entre la suma de S y C, y la multiplicación entre ambas $\frac{S \times C}{S+C}$.

3. Por último, podemos comparar la compresión realizada en disco observando los bits por aristas *bpe*, esto nos dará una idea de que tanto llegamos a comprimir en relación al grafo original.

5.2.2. Resultados

Las Tablas 5.3 y 5.5 muestran que la cantidad de bicliques del segundo experimento es muchísimo mayor a la del primero. Como consecuencia, también aumenta la cantidad de S, C, la suma $S + C$ y la multiplicación $S \times C$. Sin embargo, en la mayoría de los casos la multiplicación es la única que no aumenta significativamente a excepción del *cnr-2000-hc*.

Las tablas 5.4 y 5.6 muestran los resultados con distintos parámetros para la extracción. Como se observa, el número de iteraciones que requiere el algoritmo extractor de bicliques cambia. Se observa que a menor tamaño de biclique inicial y menor disminución, mayor cantidad de iteraciones requiere el algoritmo para converger y a su vez, mayor es el tiempo en el cual termina su ejecución. Esto pasa para todos los datasets, sin importar su tamaño. Aún así, mientras mas grande el dataset, mayor diferencia habrá en los tiempos de extracción.

	N Bicliques	$\|S\|$	$\|C\|$	$\|S+C\|$	$\|S\times C\|$
web-Stanford	1.915	134.165	47.624	181.789	1.211.879
cnr-2000-hc	1.462	118.012	57.227	175.239	1.973.025
wikipedia link lmo	2.640	66.184	81.786	147.970	3.121.873
web-Google	6.270	191.432	64.102	255.534	1.433.625
indochina-2004	14.234	3.738.985	700.129	4.439.114	149.471.673
arabic-2005-hc	21.263	12.094.741	1.259.690	13.354.431	42.4245.229

Tabla 5.3: Estadísticas de los bicliques extraídos del experimento 1, definido por los parámetros usados en el extractor.

	TBI	DTB	K	% AC	Iteraciones	Tiempo[s]
web-Stanford	500	100	6,67	52,41	11	21
cnr-2000-hc	1.000	400	11,26	61,35	8	13
wikipedia link lmo	500	100	21,10	86,15	15	14
web-Google	500	100	5,61	28,08	18	149
indochina-2004	10.000	1.000	33,67	77,00	40	7.765
arabic-2005-hc	30.000	3.000	31,76	66,29	56	54.119

Tabla 5.4: Parámetros y coeficientes de compresión del experimento 1 (definido por los parámetros usados en el extractor en Tabla 5.3).

	N Bicliques	$\|S\|$	$\|C\|$	$\|S+C\|$	$\|S\times C\|$
web-Stanford	11.345	194.211	146.273	340.484	1.555.843
cnr-2000-hc	10.898	200.729	163.473	364.202	2.494.755
wikipedia link lmo	7.889	92.374	160.217	252.591	3.344.339
web-Google	45.630	447.446	353.366	800.812	2.818.698
indochina-2004	82.534	5.246.900	2.319.894	7.566.794	167.674.121
arabic-2005-hc	61.827	15.391.910	2.984.468	18.376.378	483.364.171

Tabla 5.5: Estadísticas de los bicliques extraídos del experimento 2.

La Figura 5.1 muestra que se obtiene un mayor coeficiente de compresión K de acuerdo a los parámetros de primer experimento de extracción para todos los datasets.

	TBI	DTB	K	% AC	Iteraciones	Tiempo[s]
web-Stanford	50	10	4,56	67,28	28	66
cnr-2000-hc	100	10	6,85	77,57	37	55
wikipedia link lmo	50	10	13,24	92,29	18	16
web-Google	50	10	3,51	55,21	52	550
indochina-2004	1.000	100	22,15	86,38	100	34.196
arabic-2005-hc	3.000	300	26,3	75,53	100	185.050

Tabla 5.6: Parámetros y coeficientes de compresión del experimento 2 (definido por los parámetros usados en el extractor en Tabla 5.4).

Por otra parte, la Figura 5.2 se observa todo lo contrario. El segundo experimento tiene una mayor cantidad de aristas comprimidas. Esto quiere decir que el primer experimento obtiene una mayor compresión a nivel local, dado que los bicliques que se encuentran son de mayor tamaño y el segundo experimento logra una mayor compresión global dado que comprime mayor cantidad del grafo, a costo de unos bicliques de menor tamaño.

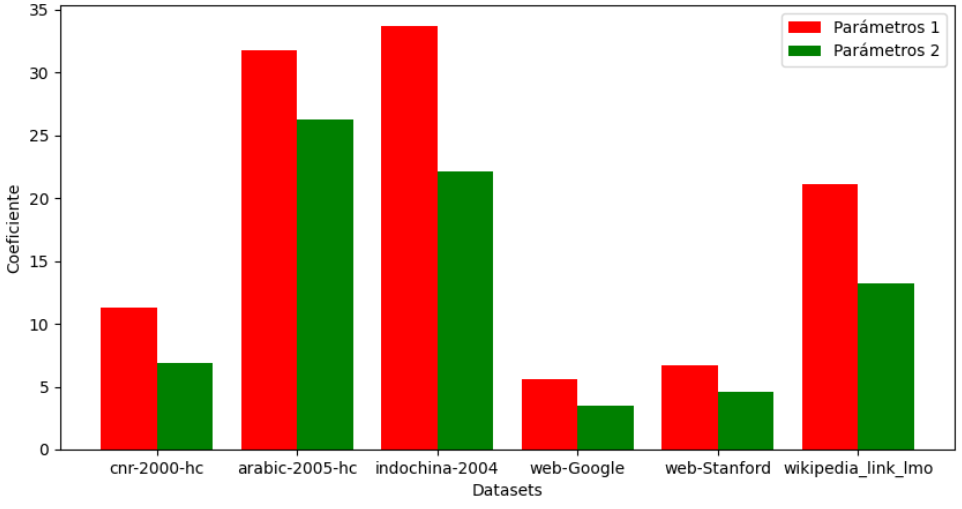


Figura 5.1: Comparación de coeficientes entre ambos experimentos.

Por último, podemos observar en la tabla 5.7 la medida de compresión mediante los

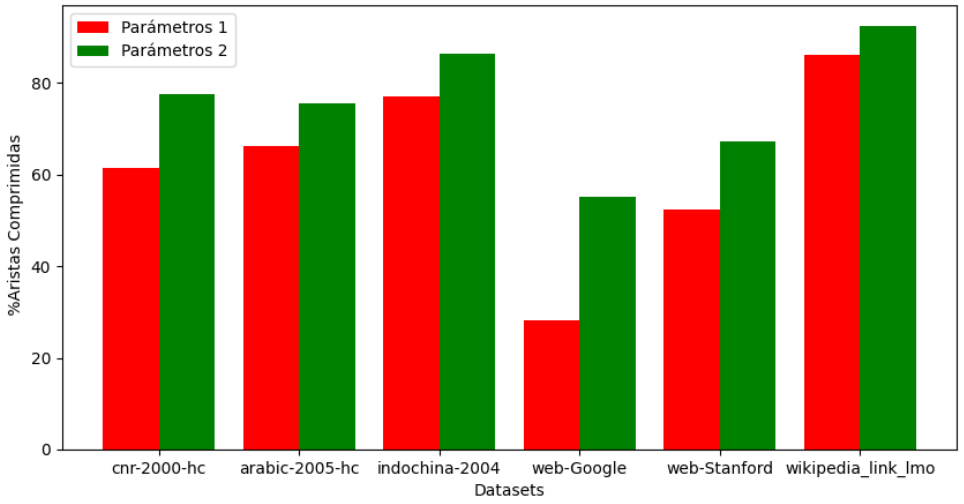


Figura 5.2: Comparación de porcentaje de aristas comprimidas entre ambos experimentos.

bits por arista *bpe* entre el grafo sin comprimir, el experimento 1, 2 y el uso de la estructura compacta k^2-*tree* ([5]). El k^2-*tree* es una estructura compacta que se basa en la representación de matriz de adyacencia de un grafo y usa un árbol k^2-ario que le permite representar submatrices con ceros con un solo 0-bit. La implementación de 5.1 utilizada en este trabajo del k^2-*tree* es una versión mejorada de la implementación usada en el artículo Arroyuelo [2].

Podemos observar que el k^2-*tree*, en la mayoría de casos ocupa mucho menos espacio que usando bicliques, con la excepción de las datasets *web-Google* y *web-Stanford*. Sin embargo, la representación con bicliques muestra una gran mejora en comparación a la representación original, logrando una compresión de mas del 50 % en la mayoría de los casos, lo que lo hace un método bastante eficiente.

	Grafo C.	R. Grafo+B (exp.1)	R. Grafo+B (exp.2)	$k^2 - tree$
web-Stanford	35,89	21,67	19,50	26,37
cnr-2000-hc	35,23	16,85	13,84	3,49
wikipedia link lmo	32,45	6,51	5,70	2,87
web-Google	36,63	29,23	24,37	33,57
indochina-2004	33,00	9,14	6,92	2,40
arabic-2005-hc	32,97	12,53	9,54	2,74

Tabla 5.7: Comparación de *bpe* entre el grafo completo sin comprimir, resto del grafo con sus bicliques (experimento 1), grafo con sus bicliques (experimento 2) y el $k^2 - tree$ respectivamente.

5.3. Evaluación de la multiplicación usando bicliques

5.3.1. Metodología de evaluación

Para la multiplicación utilizando bicliques, tenemos distintas formas de expresar nuestro resultado. Primero, podemos utilizar el parámetro *gb* antes mencionado para definir que es un buen biclique y dejar expresado el resultado como Matriz + Biclique *mb*. Además, este tipo de resultado conlleva a obtener otro tipo de información, como el número de *task* producidas (*N Task*) y el número de bicliques expresados (*N Bicliques*).

Por otra parte, se podría decir que obtener una matriz más bicliques no es comparable con multiplicar el grafo completo consigo mismo, el cual produce una matriz completa y sin repeticiones de aristas. Para esto se mide el tiempo de transformar la matriz + biclique a una sola matriz sin repeticiones (T). Por lo tanto, el tiempo total que se demora en obtener la matriz es la suma entre la multiplicación con bicliques y la transformación ($(M + B)^2_{mb} + T$).

Finalmente, podemos comparar el tiempo de ejecución total como la suma descrita anteriormente con la multiplicación al no utilizar *gb*, dado que esta también nos

entrega una matriz completa sin bicliques.

En esta ocasión, solamente tenemos un único parámetro a ajustar, *gb* el cual se define como 100 para todos los casos, dado que es el mínimo tamaño de bicliques para los grafos de menos magnitud.

5.3.2. Resultados

Observando las tablas 5.8 y 5.9, podemos analizar que en ambos casos el tiempo de ejecución de la multiplicación con bicliques obteniendo solamente una matriz es muy cercano al obtener una matriz con bicliques y posteriormente transformarla.

Por otro lado, comparándolas, podemos ver que el rendimiento del experimento que tenia mejor coeficiente K, pero menor compresión de aristas es muy superior, tanto que en algunos casos es 5 veces mas rápido. Esto se debe a que al tener menor cantidad de bicliques de mayor tamaño, tiene como consecuencia que se requieran menor cantidad de productos cartesianos con tamaños mas grandes. Esto se ve reflejado en la columna *N Task*. Además, también podemos observar que la cantidad de bicliques en la mayoría de casos se duplica, obteniendo una mayor repetición de aristas.

Por último, analizando la tabla 5.10, podemos ver que k^2-*tree* es el algoritmo mas lento de todos y pierde en todos los casos. Sin embargo, el k^2-*tree* es el que ofrece mayor compresión. Luego vemos que la multiplicación con bicliques del segundo experimento es superior a la multiplicación sin usar compresión en los grafos mas grandes, pero es mas lento en los grafos mas pequeños. Por otra parte, el primer escenario es el que supera a todos los métodos, excepto en *web-Google* y *web-Stanford* donde usando el algoritmo sin bicliques es el mejor. A pesar de eso, la diferencia no es mucha y el algoritmo con bicliques es entre dos y cinco veces mas rápido en los grafos mas grandes tales como *arabic-2005-hc* y el *indochina-2004*.

	$(M+B)^2_{mb}$[s]	$(M+B)^2_{mb}$+T[s]	$(M+B)^2_{m}$[s]	Task	Bicliques
web-Stanford	1,66	3,56	3,58	339.998	48.269
cnr-2000-hc	0,67	2,47	2,38	346.938	47.104
wikipedia link lmo	0,53	5,61	5,54	57.158	148.081
web-Google	10,03	13,37	13,33	598.778	143.621
indochina-2004	89,96	239,84	243,38	6.892.791	2.626.260
arabic-2005-hc	431,77	1.403,21	1352,74	22.323.924	8.868.143

Tabla 5.8: Resultados de la multiplicación del experimento 1.

	$(M+B)^2_{mb}$[s]	$(M+B)^2_{mb}$+T[s]	$(M+B)^2_{m}$[s]	Task	Bicliques
web-Stanford	14,60	17,62	17,47	516.314	80.342
cnr-2000-hc	5,05	6,6	6,64	558.880	46.793
wikipedia link lmo	2,65	9,57	9,44	182.207	201.285
web-Google	175,89	182,08	180,28	1.117.326	238.873
indochina-2004	789,37	992,78	1.013,09	8.659.238	4.303.182
arabic-2005-hc	1.529,27	2.720,82	2.702,14	24.776.254	11.442.001

Tabla 5.9: Resultados de la multiplicación del experimento 2.

	M^2[s]	$(M+B)^2_{m}$(exp. 1)[s]	$(M+B)^2_{m}$(exp. 2)[s]	k^2-tree[s]
web-Stanford	2,36	3,58	17,47	586,29
cnr-2000-hc	2,52	2,38	6,64	22,54
wikipedia link lmo	25,04	5,54	9,44	82,47
web-Google	4,08	13,33	180,28	2325,77
indochina-2004	22.122,4	243,38	1.013,09	19.241,6
arabic-2005-hc	6.487,75	1.352,74	2.702,14	13.479,2

Tabla 5.10: Comparación de los tiempos de las multiplicaciones, donde el resultado es solo una matriz obtenida de 4 formas distintas. Primero es la multiplicación del grafo completo, luego el experimento 1 y 2, y finalmente el k^2-tree.

5.4. Evaluación del conteo de triángulos

5.4.1. Metodología de evaluación

Para el conteo de triángulos, primero tendremos la extracción de bicliques de cada dataset. En esta evaluación solamente analizaremos como afecta el coeficiente K y el porcentaje de aristas comprimidas con respecto a la multiplicación de matrices.

Por otro lado, para la multiplicación se utiliza el método que nos da como resultado directamente la matriz completa. Por lo tanto, tendremos dos resultados, la operación del cuadrado y la operación completa de conteo, la cual contempla el cuadrado y el doble ciclo descrito en el Algoritmo 10.

5.4.2. Resultados

Analizando la tabla 5.12, vemos que el coeficiente K se mantiene bajo en todos los datasets incluso en los mas grandes. Además, el porcentaje de aristas comprimidas en todos los casos siempre es mayor al 65 %, por lo que se logra una buena compresión del grafo.

	N Bicliques	$\|S\|$	$\|C\|$	$\|S+C\|$	$\|S\times C\|$
msc23052	2.362	24.846	94.753	119.599	843.483
msc10848	1.463	17.712	104.481	122.193	1.101.924
crankseg_1	1.497	17.174	106.951	124.125	1.055.389
sme3Dc	5.474	50.078	353.769	403.847	2.070.206
crankseg_2	4902	61325	378069	439394	4182725
hood	28.036	246.470	1.249.836	1.496.306	9.844.443
coPapersCiteseer	24.305	439.150	1.303.825	1.742.975	23.967.945
audikw	126.571	1.189.509	8.632.728	9.822.237	50.958.684

Tabla 5.11: Estadísticas de los bicliques extraídos, definidos por los parámetros usados por el extractor

En cuanto a la tabla 5.12, observamos que los primeros datasets, tanto el cuadrado

como la operación completa de conteo, es mas eficiente utilizando bicliques. En cambio, para los datasets de mas abajo la multiplicación sin bicliques es mejor. Estos son los datasets de mayor tamaño y tiene sentido que sean mas lento, dado que el coeficiente K es bajo para la cantidad de aristas que tiene. El *coPapersCiteseer* es la excepción a los datasets grandes, dado que su coeficiente K es mas grande e igual logra una compresión del 75 % de las aristas.

A partir de estos resultados, nos podemos dar cuenta que mientras sea alto el coeficiente K y el porcentaje de aristas comprimidas para los datasets de mayor magnitud, la multiplicación es mejor utilizando sus bicliques.

	TBI	TDB	K	% AC	Iteraciones	Tiempo[s]
msc23052	1.000	200	7,05	73,04	9	5
msc10848	1.000	200	9,02	89,60	9	3
crankseg_1	2.000	200	8,50	73,80	14	12
sme3Dc	1.000	200	5,13	65,75	15	29
crankseg_2	2.000	200	9,52	85,46	24	53
hood	1.000	200	6,58	91,42	12	47
coPapersCiteseer	1.000	100	13,75	74,73	100	649
audikw	1.000	200	5,19	65,62	100	4531

Tabla 5.12: Parámetros y coeficientes de compresión (definido por los parámetros usados en el extractor en la Tabla 5.11).

Nos podemos dar cuenta que la operación mas costosa es calcular el cuadrado de la matriz. En general, el tiempo que toma calcular el doble ciclo es bastante bajo y en el caso de los datasets pequeños casi inexistente, por lo tanto, podemos decir que para las matrices que tengan bicliques grandes, se obtiene una mejora al conteo de triángulos.

Por ultimo, ejemplificaremos usando el grafo *audikw*, el cual es el que tiene mayor cantidad de nodos y aristas. Utilizaremos dos *TDB* distintos, 200 y 100 respectivamente, observando como afecta en la compresión, coeficiente K y el tiempo del cuadrado.

	A^2[t]	$(A+B)^2$[t]	Conteo[t]	Conteo con bic.[t]
msc23052	1,61	1,37	1,62	1,41
msc10848	4,81	1,79	4,83	1,84
crankseg_1	6,49	3,15	6,51	3,21
sme3Dc	16,65	18,63	16,76	18,85
crankseg_2	27,87	12,21	27,95	12,40
hood	14,43	40,58	14,51	40,98
coPapersCiteseer	238,44	121,97	239,27	123,77
audikw	258,64	1.545,12	260,05	1.550,21

Tabla 5.13: Resultados de la multiplicación y algoritmo completo de conteo de triángulos, comparando el uso de bicliques.

	N Bicliques	$\|S\|$	$\|C\|$	$\|S+C\|$	$\|S \times C\|$
audikw(exp. 1)	126.571	1.189.509	8.632.728	9.822.237	50.958.684
audikw(exp. 2)	34.072	569.955	2.549.751	3.119.706	23.666.859

Tabla 5.14: Estadísticas de los bicliques extraídos para el dataset audikw, definidos por los parámetros usados por el extractor

	TBI	DTB	K	% AC	Iteraciones	Tiempo[s]
audikw(exp. 1)	1.000	200	5,19	65,62	100	4531.08s
audikw(exp. 2)	1.000	100	7,59	30,48	100	5561.73s

Tabla 5.15: Parámetros y coeficientes de compresión (definido por los parámetros usados en el extractor en la Tabla 5.14).

Observando la Tabla 5.14, vemos principalmente la diferencia entre el numero de bicliques. Por otra parte, en la Tabla 5.15 vemos que el coeficiente K es mayor para el escenario que usa DTB con el valor 100, pero tiene una menor compresión de aristas. Analizando la ultima tabla 5.16, vemos la gran diferencia de tiempo en el cuadrado de la matriz, esto es dado principalmente a la cantidad de bicliques, dado que el algoritmo deberá realizar mayor cantidad de intersecciones y, a la vez,

obtendremos mayor cantidad de productos cartesianos, ralentizando el algoritmo por partes distintas.

	A^2[t]	$(A+B)^2$[t]	Conteo[t]	Conteo con bic.[t]
audikw(exp. 1)	258,64	1.545,12	260,05	1.550,21
audikw(exp. 2)	258,64	387,13	260,05	391,22

Tabla 5.16: Resultados de la multiplicación y algoritmo completo de conteo de triángulos, comparando el uso de bicliques para el dataset audikw.

Capítulo 6

Discusión final

6.1. Conclusiones

La multiplicación propuesta se ve efectiva para grafos que cumplan la condición de tener grandes bicliques y que obtienen una mayor compresión, logrando una mejora de tiempo en algunos casos de mas del 50 %. Además, aunque puede haber una variación de parámetros al momento de encontrar los bicliques, aun así, se logra esta mejora en tiempo y logrando una mayor compresión.

Así, se cumple el primer objetivo de esta memoria donde se implementó la extracción de bicliques para grafos masivos. Gracias a esto, logramos encontrar bicliques dentro de matrices ralas o dispersas, para posteriormente multiplicar estas matrices de una forma mas rápida, aprovechando la compresión del grafo. Esto hace que se cumplan a la vez el segundo y tercer objetivo dado que a pesar de no lograr una mejora de espacio con respecto al k^2-*tree*, si logramos una compresión considerable con respecto al baseline y obtenemos una gran mejora con respecto al tiempo de cómputo de las operaciones de cuadrado de matrices y conteo de triángulos.

Si bien se utilizaron mayormente grafos de la web para este trabajo, no queda descartado que puedan utilizarse para cualquier otro, siempre y cuando tenga la característica de tener bicliques de gran tamaño. Los grafos de redes sociales o los

genómicos pueden ser un muy buen punto a tomar en consideración, dado que el tiempo de computo de la multiplicación es clave en ambas partes.

6.2. Trabajo Futuro

Para la extracción de bicliques, buscar parámetros específicos, que cumplan con la naturaleza del grafo puede llegar a ser algo tedioso y confuso. Para esto podría investigarse una forma de encontrar los mejores parámetros para cada grafo mediante un algoritmo, o incluso, podría utilizarse modelos de *Deep Learning* con modelos predictivos como lo son las redes neuronales, los cuales tengan como entrada el grafo a extraer bicliques, y como salida los parámetros que mejor se ajusten.

Por otra parte, al momento de computar la multiplicación con bicliques, y obtenemos como resultado una matriz con bicliques. Un potencial trabajo futuro es buscar la forma de re-comprimir la matriz, quitando la redundancia que generan los bicliques y posteriormente, volver a ejecutar la multiplicación sobre la matriz. Esto ayudaría a otras operaciones tales como la clausura transitiva con bicliques.

Índice de tablas

Índice de figuras

Índice de algoritmos

Bibliografía

[1] Josh Alman y Virginia Vassilevska Williams. «Further limitations of the known approaches for matrix multiplication». En: *arXiv preprint arXiv:1712.07246* (2017).

[2] Diego Arroyuelo, Adrián Gómez-Brandón y Gonzalo Navarro. «Evaluating Regular Path Queries on Compressed Adjacency Matrices». En: *International Symposium on String Processing and Information Retrieval*. Springer. 2023, págs. 35-48.

[3] 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.

[4] Sergey Brin y Lawrence Page. «The Anatomy of a Large-Scale Hypertextual Web Search Engine». En: *Computer Networks* 30.1-7 (1998), págs. 107-117.

[5] 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.

[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] Andrei Z Broder et al. «Min-wise independent permutations». En: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, págs. 327-336.

[8] Aydin Buluç et al. «Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks». En: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 2009, págs. 233-244.

[9] Don Coppersmith y Shmuel Winograd. «Matrix multiplication via arithmetic progressions». En: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, págs. 1-6.

[10] Timothy A Davis y Yifan Hu. «The University of Florida sparse matrix collection». En: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), págs. 1-25.

[11] Ahmed Elgohary et al. «Compressed linear algebra for declarative large-scale machine learning». En: *Communications of the ACM* 62.5 (2019), págs. 83-91.

[12] Ahmed Elgohary et al. «Compressed linear algebra for large-scale machine learning». En: *The VLDB Journal* 27.5 (2018), págs. 719-744.

[13] Paolo Ferragina et al. «Improving matrix-vector multiplication via lossless grammar-compressed matrices». En: *arXiv preprint arXiv:2203.14540* (2022).

[14] Alexandre P Francisco et al. «Graph Compression for Adjacency-Matrix Multiplication». En: *SN Computer Science* 3.3 (2022), pág. 193.

[15] Cecilia Hernández y Gonzalo Navarro. «Compressed representation of web and social networks via dense subgraphs». En: *String Processing and Information Retrieval: 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings 19*. Springer. 2012, págs. 264-276.

[16] Cecilia Hernández y Gonzalo Navarro. «Compressed representations for web and social graphs». En: *Knowledge and information systems* 40.2 (2014), págs. 279-313.

[17] Chinmay Karande, Kumar Chellapilla y Reid Andersen. «Speeding up algorithms on compressed web graphs». En: *Proceedings of the Second ACM International Conference on Web Search and Data Mining*. 2009, págs. 272-281.

[18] Jérôme Kunegis. «Konect: the koblenz network collection». En: *Proceedings of the 22nd international conference on world wide web*. 2013, págs. 1343-1350.

[19] Jure Leskovec y Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. Jun. de 2014.

[20] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[21] Amir Schoor. «Fast algorithm for sparse matrix multiplication». En: *Information Processing Letters* 15.2 (1982), págs. 87-89.

[22] Volker Strassen. «Gaussian elimination is not optimal». En: *Numerische mathematik* 13.4 (1969), págs. 354-356.

[23] Andrew K Watson et al. «The methodology behind network thinking: graphs to analyze microbial complexity and evolution». En: *Evolutionary genomics: Statistical and computational methods* (2019), págs. 271-308.

[24] Uri Zwick. «All pairs shortest paths using bridging sets and rectangular matrix multiplication». En: *Journal of the ACM (JACM)* 49.3 (2002), págs. 289-317.