DRAFT COPY Printed 31 de julio de 2019 16:30

Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería - Programa de Magíster en Ciencias de la Computación

DISEÑO E IMPLEMENTACIÓN DE MÉTODO DE COMPRESIÓN DE GRAFOS BASADO EN CLUSTERING DE CLIQUES MAXIMALES

Tesis para optar al grado de
MAGÍSTER EN CIENCIAS DE LA COMPUTACIÓN

POR
Felipe Alberto Glaría Grego
CONCEPCIÓN, CHILE
Mayo, 2019

Profesor guía: Lilian Salinas Ayala
Profesor co-guía: Cecilia Hernández Rivas
Departamento de Ingeniería Informática y Ciencias de la Computación
Facultad de Ingeniería
Universidad de Concepción

A mi familia...

AGRADECIMIENTOS

Quisiera agradecer en primer lugar a mi hija Olivia, mi madre Milena, a Paulina y Claudia, mujeres elementales en mi vida durante el transcurso de este trabajo, quienes entregaron su apoyo incondicional en todo momento, y por ello les agradezco profundamente.

También agradezco a las profesoras Cecilia Hernández y Lilian Salinas. Su soporte, ayuda y comprensión en el desarrollo de esta tesis es invaluable.

De igual manera, agradezco a mis compañeros de generación del programa de Magister, con quienes compartimos y generamos una comunidad de aprendizaje fundamental para los seres sociales que somos, y así poder avanzar de manera grata y fructíera.

Resumen

La compresión y manejo eficiente de grandes grafos se vuelve cada día más fundamental en distintos ámbitos, desde lo macro de la representación de la Web y las redes sociales, hasta lo micro de representar componentes biológicos. Su crecimiento ha tenido un aumento constante, y con esto se eleva la exigencia en recursos para almacenarlos, junto con la complejidad de los algoritmos para procesar consultas sobre ellos.

Una alternativa a este problema es buscar una opción de representar estos grafos de manera comprimida que, además de usar menos espacio en disco, permita responder dichas consultas en el menor tiempo posible.

En este trabajo se desarrolla un método de compresión para grafos no dirigidos, que aprovecha la superposición de cliques maximales en la generación de una estructura compacta que permite manejar de manera eficiente dichos grafos, ahorrando en espacio de disco pero pagando su costo en tiempos de acceso.

Si bien el problema de obtener los cliques maximales de un grafo es muy complejo (NP-completo), existen algoritmos para grafos esparsos que logran generar el listado en poco tiempo. Además es un costo que se debe pagar una sola vez, luego de generar la estructura compacta se puede obtener nuevamente el listado de cliques directo de ella.

Finalmente se prueba el rendimiento de esta estructura propuesta con diferentes grafos, estudiando qué características deben tener para aprovechar sus ventajas, y se compara con otros algoritmos relevantes del estado del arte, comprobando su rendimiento tanto en la compresión de grafos como en resolver las consultas sobre ella.

Se identifica que para grafos altamente clusterizados la compresión logra su máxima eficiencia, ocupando menos de un bit por arco, y siempre menos que los demás algoritmos. En cuanto a tiempos de acceso, los resultados más competitivos se logran en acceso aleatorio, y se proponen líneas de investigación para mejorar esto.

Capítulo 1

INTRODUCCIÓN

En los últimos años se ha visto un gran crecimiento en grafos de redes sociales y de la web, junto a otros de características similares. Por ejemplo, el número de sitios indexados por los principales motores de búsqueda en la web se estima actualmente en al menos 5,68 miles de millones[1], o la cantidad de usuarios activos diarios en la red social Facebook es de 1,56 mil millones, con un crecimiento anual de un 8 % [2].

El gran tamaño de los grafos de la Web y redes sociales, trae consigo varios problemas, siendo uno de los más importantes el alto costo en recursos que demanda su procesamiento. Este costo está dado principalmente por el espacio requerido en memoria, donde la jerarquía de memoria de los sistemas computacionales modernos penaliza los tiempos de acceso a medida que los datos se alejan de las unidades de procesamiento. Este problema ha motivado a la comunidad científica a proponer estructuras comprimidas que no sólo requieran menos espacio de almacenamiento, sino también proporcionen resolución de consultas que permitan la navegación del grafo a través de consultas básicas, tales como acceso a vecinos. El objetivo de estas representaciones comprimidas es permitir la simulación de algoritmos de procesamiento de grafos usando mucho menos espacio en memoria que las representaciones sin comprimir.

En el contexto de estructuras de datos que ocupan muy poco espacio, el área de estructuras compactas ha tenido un desarrollo importante desde el punto de vista teórico y práctico en los últimos años. Actualmente, existen estructuras compactas que permiten representar secuencias de bits, bytes y símbolos con soporte de consultas básicas, así como otras estructuras compactas más complejas, como árboles y grafos con soporte de navegación.

Existen varias propuestas para comprimir grafos. Sin embargo, aún existen algunas características de representación de grafos que no se han explorado. En particular, en este trabajo se propone que es posible definir un método de clustering eficiente para construir una estructura compacta de grafos, basada en la superposición de vértices de los cliques maximales que componen el grafo. En este caso, la compresión del grafo se logra mediante la representación implícita de aristas mediante la representación de los vértices que definen los cliques.

[1] `http://www.worldwidewebsize.com/`, consultado el 09 de mayo del 2019.

[2] https://investor.fb.com, informe de resultados del primer trimestre del 2019 de Facebook.

Capítulo 2

MARCO TEÓRICO

En este capítulo se presentan las definiciones de grafos y cliques maximales, y codificaciones ocupadas en este trabajo.

2.1. Grafos, cliques maximales y métricas de clusterización

Se define un **grafo** $G = (V, E)$ como el conjunto finito de *vértices* o *nodos* V y el conjunto de *aristas* $E \subseteq V \times V$ (arcos). La expresión $V(G)$ representa el conjunto de sus vértices y $E(G)$ el conjunto de sus aristas. El *orden* de un grafo corresponde al total de sus vértices $|V(G)|$, mientras que el *tamaño* de un grafo corresponde al total de sus aristas $|E(G)|$.

Dos vértices v_1 y $v_2 \in V(G)$ son **adyacentes** o **vecinos** si $(v_1, v_2) \in E(G)$ y $v_1 \neq v_2$. Un grafo es **no dirigido** cuando la arista conlleva ambos sentidos, quiere decir que $(v_1, v_2) = (v_2, v_1)$, ambos vértices son vecinos directos entre sí. Caso aparte son los grafos **dirigidos**, donde las aristas tienen un solo sentido, y $(v_1, v_2) \neq (v_2, v_1)$. En este caso, v_2 es llamado **vecino directo** de v_1, mientras v_1 es llamado **vecino inverso o reverso** de v_2. Un **grafo denso** es aquel que su número de aristas es cercano al máximo. Este trabajo está enfocado a utilizar grafos no dirigidos y poco densos.

El **grado de un vértice** $d(v)$ se define como la cantidad de vértices en $V(G)$ que son adyacentes con v. La **matriz de adyacencia** de un grafo G corresponde a una matriz binaria cuadrada $|V(G)| \times |V(G)|$ donde cada bit representa si un par de vértices v_1 y $v_2 \in V(G)$ son vecinos o no. En resumen, todos sus valores son ceros a menos que haya una arista entre dichos vértices.

(a) (b)

Figura 2.1: Ejemplo de grafos bipartitos. (a) Grafo bipartito. (b) Grafo bipartito completo o biclique.

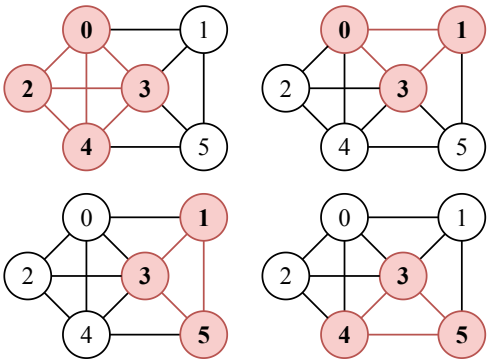


Figura 2.2: Ejemplo de grafo y sus cliques maximales.

Un grafo **k-degenerate** es no dirigido, donde cada subgrafo tiene un vértice con grado a lo más **k**. El índice de **degeneracy** de un grafo, $D(G)$, es el menor valor **k** para el cual el grafo es **k-degenerate**.

Un grafo es **bipartito** cuando sus vértices se pueden separar en dos conjuntos separados U y W, tal que se cumple $U \cup W = V$ y $U \cap W = \varnothing$. Un grafo es **bipartito completo** o **biclique**, cuando todos los vértices de un conjunto son vecinos directos de todos los vértices del otro conjunto. En la Figura 2.1 se ilustra un ejemplo de grafo bipartito y un biclique.

Un **clique** es un subgrafo donde todos los vértices son adyacentes entre sí, es decir, $\exists V' \subseteq V(G), \forall v_1, v_2 \in V', (v_1, v_2) \in E(G)$. Un **clique maximal** no puede extenderse incluyendo otro vértice adyacente, es decir, no es subconjunto de otro clique más grande. En la Figura 2.2 se presenta ejemplo de un grafo y sus cliques maximales.

Un **triángulo** es un subgrafo de tres vértices y tres aristas. Se define $\lambda(v)$ como la cantidad de triángulos donde participa un nodo v, y $\lambda(G)$ como la cantidad de triángulos de un grafo, y se calcula sumando el cálculo individual para cada vértice, y dividiendo el total en tres (por cada triángulo se cuentan 3 veces los vértices), como lo muestra la siguiente ecuación

$$\lambda(G) = \frac{1}{3} \sum_{v \in V} \lambda(v) \tag{2.1}$$

Un **triplete** es un subgrafo de tres vértices y dos aristas, donde las aristas comparten un vértice común. Se define $\tau(v)$ como la cantidad de tripletes donde v es el vértice común, y $\tau(G)$ como la cantidad de tripletes de un grafo.

$$\tau(G) = \sum_{v \in V} \tau(v) \tag{2.2}$$

El **coeficiente de clusterización** de un vértice indica cuánto está conectado con sus vecinos, y se define como $c(v) = \lambda(v)/\tau(v)$. El coeficiente de clusterización de un grafo $(C(G))$ es el promedio del coeficiente de todos los nodos del grafo, y se define como:

$$C(G) = \frac{1}{|V'|} \sum_{v \in V'} c(v) \tag{2.3}$$
$$V' = \{v \in V | d(v) \geq 2\}$$

donde V' es el conjunto de vértices con un grado mayor a dos. Su rango es entre $[0, 1]$, mientras más cercano a 1 indica más conexión entre vértices.

La **transitividad** de un grafo ($T(G)$) es la probabilidad que un par de nodos adyacentes estén interconectados, y se define como:

$$T(G) = \frac{3\lambda(G)}{\tau(G)} \tag{2.4}$$

y su rango también va entre $[0, 1]$, siendo 1 cuando todos los nodos están interconectados con todos.

Tanto el coeficiente de clusterización como la transitividad son métricas que permiten vislumbrar cuán conectados o clusterizados están los vértices de un grafo, y de sus ecuaciones se puede notar que están relacionados.

2.2. Codificaciones

Existen distintos tipos de codificaciones, según la aplicación. En esta sección se resumen algunas de relevancia para este trabajo, como algunos códigos universales o la codificación Huffman.

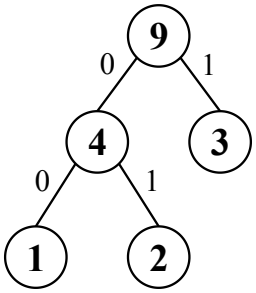
2.2.1. Códigos universales

Los códigos universales para enteros son un tipo de códigos que transforman enteros positivos en secuencias de bits, donde el largo de la secuencia final de bits tiene relación con el entero a codificar. Existen varios códigos de este tipo, algunos son:

- **Código unario**: Se representa un entero x por una secuencia de $1^{x-1}0$, donde el 0 indica el término de la secuencia. Por ejemplo, el número 5 se representa por la secuencia 111110. Este código no es muy eficiente por si solo, pero se usa de base para otro tipo de códigos.
- **Código gamma (γ)**: Se representa un entero x en un par concatenado de *largo* y *offset*. *Offset* es la representación binaria de x, pero sin el primer 1. Por ejemplo, para $x = 5$ su representación binaria es 101, por tanto su *offset* es 01. *Largo* codifica el largo de *offset* en código unario. Para $x = 5$, el largo de *offset* es 2 bits, por tanto *largo* es 110. Concatenando ambas, el código γ para $x = 5$ es 11001.
- **Código delta (δ)**: Este código es una extensión del código γ para enteros largos. Básicamente hacen lo mismo, pero el *largo* lo representan en código γ en vez de código unario. El código δ para $x = 5$ es 10001.

323312313
(a)

1011100011001
(b)



(c)

Figura 2.3: Ejemplo de codificación Huffman. (a) Secuencia de entrada. (b) Secuencia de salida. (c) Diagrama de vocabulario en árbol binario.

2.2.2. Codificación Huffman

Una codificación muy usada es la propuesta por Huffman[24], que retorna códigos de largo variable. Su conversión se basa en la frecuencia de los símbolos en la secuencia de entrada, codificando los más frecuentes con menos bits. Esta representación requiere tanto la secuencia codificada junto a un vocabulario de símbolos para recuperar la secuencia original.

En la Figura 2.3 se muestra un ejemplo de esta codificación, donde (a) es la secuencia de entrada, (b) es la secuencia de salida, y (c) es el vocabulario de símbolos dispuesto en un árbol binario. Primero se cuenta la frecuencia de cada símbolo en la secuencia de entrada, dos veces 1, dos veces 2, y cinco veces 3. Luego se crea el árbol binario, partiendo por las hojas de los símbolos menos frecuentes, 1 y 2, y creando un nodo con la frecuencia total de ambos casos, en total cuatro. Luego esa rama se añade a otro nodo junto con la hoja del símbolo más probable, 3, y ambas suman la frecuencia total de nueve símbolos. Así, el símbolo 3 puede representarse tan solo con un bit en 1, y para los otros casos se necesitarán dos bits.

Para recuperar la secuencia original, se debe recorrer la de salida bit a bit de izquierda a derecha, e ir avanzando en el árbol binario hasta llegar a una hoja. Una desventaja de este método es que necesita decodificar de manera secuencial todo el código para obtener la secuencia original.

Capítulo 3

ESTADO DEL ARTE

En este capítulo se realiza una revisión del trabajo previo realizado en compresión de grafos, se profundiza sobre las posibles estructuras compactas a utilizar, y se detalla el problema y la solución a utilizar para la detección de cliques maximales.

3.1. Compresión de grafos

El problema de compresión de grafos ha sido abordado de distintas maneras en las últimas décadas. En esta sección se revisan los trabajos más relevantes del área.

3.1.1. The WebGraph Framework, *Boldi y Vigna*

Uno de los primeros trabajos en la materia es *WebGraph* de Boldi y Vigna [4], apuntado a comprimir grafos dirigidos como el grafo de la Web, aprovechando la distribución potencial de las diferencias entre vecinos sucesivos, reflejados en dos características de sus enlaces ordenados por su *URL*, **localidad** (hipervínculos donde sus *URL* tienen un prefijo en común y si se ordenan lexicográficamente en una lista estarán muy cerca entre ellos) y **similitud** (los sitios que tienden a estar juntos en esa lista lexicográfica también tienden a tener muchos sucesores en común). Así, codifican las listas de adyacencias basadas en otras listas de adyacencias y cuán similar sean entre ellas.

Primero, cada nodo se numeran los N nodos del grafo de 0 a $N-1$, ordenados de manera lexicográfica según sus *URL*. En una primera aproximación, cada nodo tiene asociado su grado de salida (*Outdegree*) y su listado de adyacencia o sucesores asociado $S(x)$. Luego, aprovechando la localidad de los nodos en dichas listas, se pueden representar usando las diferencias entre sus nodos, quiere decir si $S(x) = (s_1, s_2, ..., s_k)$ es el listado de sucesores del nodo x con k vecinos, se codifica como $(s_1 - x, s_2 - s_1 - 1, ..., s_k - s_{k-1} - 1)$. En la Tabla 3.1 se muestran ambos casos, usando listado de sucesores y usando la diferencia. Para evitar tener que lidiar con números negativos, el primer número en esta nueva secuencia se codifica de la siguiente manera:

$$w(x) = \begin{cases} 2x & x \geq 0 \\ 2|x| - 1 & x < 0 \end{cases} \tag{3.1}$$

Avanzando en el modelo de compresión, cada nodo tiene un entero r llamado referencia, si $r = 0$ la lista no está comprimida usando una referencia, y para $r > 0$ la lista x está definida por la diferencia de la lista $x - r$. Un bitmap llamado *copy list* codifica los

Tabla 3.1: Representación de Webgraph usando listado de sucesores directo y con brechas.

Nodo	Outd.	Sucesores	Usando brechas
...	...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034	3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041	1, 0, 0, 4, 0, 0, 290, 0, 0, 2723
17	0		
18	5	13, 15, 16, 17, 50	9, 1, 0, 0, 32
...	...	...	...

Tabla 3.2: Representación de Webgraph usando copy list.

Nodo	Outd.	Ref.	Copy list	Nodos extra
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Tabla 3.3: Representación de Webgraph usando copy blocks.

Nodo	Outd.	Ref.	# blocks	Copy blocks	Nodos extra
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, 317, 3041
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

sucesores que deben ser copiados a la lista, con un 1 si el nodo referenciado esta presente en dicha lista o no. Adicionalmente se usa una lista extra para agregar todos los nodos remanentes. Las copy list se codifican en *copy blocks*, donde el primer block es 0 si la copy list comienza con un 0. Un bloque se representa por l largo de 0 o 1 en la lista menos uno, y el último bloque se omite. En las Tablas 3.2 y 3.3 se ilustra un ejemplo para ambos casos.

Como se puede apreciar de los ejemplos, la consecutividad es frecuente en el listado de nodos extra. Este hecho se puede aprovechar en un paso previo a la compresión por brecha, aislando las subsecuencias correspondientes a intervalos de enteros. Sólo los intervalos de largo no menor a un cierto umbral L_{min} son considerados. Entonces, cada listado de nodos extra se comprime de la siguiente manera:

- Un listado de intervalos de enteros. Se representa cada intervalo por su valor extremo izquierdo y su largo. Su valor extremo izquierdo se comprime usando la diferencia entre si mismo y el previo extremo derecho menos dos, ya que debe haber al menos un entero entre el final de un intervalo y el inicio del siguiente. Al largo del intervalo se le resta el umbral L_{min}.

- Una lista de nodos residuales, los que no son parte de los intervalos anteriores, comprimida usando la diferencia.

Tabla 3.4: Representación de Webgraph usando intervalos, con umbral $L_{min} = 2$.

Nodo	Outd.	Ref.	# blocks	Copy blocks	# intervalos	Ext. izq.	Largo	Residuales
...	...	...	...	...	...	...	...	...
15	11	0			2	0, 2	3, 0	5, 189, 111, 718
16	10	1	7	0, 0, 2, 1, 1, 0, 0	1	600	0	12, 3018
17	0							
18	5	3	1	4	0			50
...	...	...	...	...	...	...	...	...

Finalmente, en la Tabla 3.4 se puede apreciar la representación comprimida resultante, con un umbral $L_{min} = 2$.

En un trabajo posterior Boldi et. al. [3], usando la matriz de adyacencia y basados en aplicar permutaciones a sus filas, logran reordenar y generar una nueva matriz donde las filas, si son similares (contienen 1s en posiciones muy comunes), deben ser consecutivas o en una vecindad acotada. En otro trabajo propusieron un nuevo algoritmo llamado *Layered Label Propagation* [2] (propagación de etiquetas por capas). Su objetivo era poder ocupar las técnicas desarrolladas anteriormente para grafos de redes sociales, donde los vértices no pueden ser ordenados de manera lexicográfica. Usando la matriz de adyacencia, junto con descomponer en tareas el re-ordenamiento de la matriz y aprovechar los procesadores multi-core, logran muy buenos resultados.

3.1.2. BFS, *Apostolico y Drovandi*

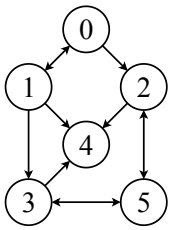
Otra alternativa de compresión bastante competitiva, también para grafos dirigidos, es la que presentan Apostolico y Drovandi [1], basado en la topología del grafo de la Web en vez de las *URL* subyacentes. En vez de asignarle índices a los nodos según el orden lexicográfico de sus *URL*, realizan un recorrido por *breath-first* o búsqueda en anchura del grafo, numerando cada nodo según el orden en que se expanden. Este proceso y su compresión inducida lo llaman *Fase 1*, y la compresión de las aristas remanentes como *Fase 2*.

En la Fase 1, al expandir un nodo $v_i \in V$ se le asignan índices enteros consecutivos a sus k_i vecinos, y se guarda el valor k_i. Cuando el recorrido del grafo se completa, todas las aristas que pertenecen al árbol de búsqueda por anchura quedan codificadas en la secuencia $\{k_1, k_2, ..., k_{|V|}\}$ llamada *traversal list* (lista de recorrido). En la Figura 3.1 se presenta un ejemplo para la Fase 1, donde en (a) se presenta el orden de los nodos asignados por BFS, y en (b) las aristas restantes junto con el listado de recorrido.

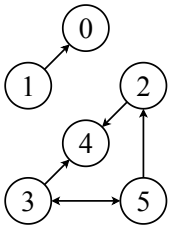
Luego comprimen por separado trozos consecutivos de l nodos, siendo l un valor específico que define el nivel de compresión. Cada trozo comprimido C, conformado por los nodos $v_i, v_{i+1}, ..., v_{i+l-1}$, lleva prefijado la secuencia $\{k_i, k_{i+1}, ..., k_{i+l-1}\}$.

En la Fase 2, codifican la lista de adyacencia A_i de cada nodo $v_i \in V$ de un trozo C en orden creciente. Cada lista codificada consiste en la diferencia entre elementos adyacentes en la lista y un indicador tipo del set $\{\alpha, \beta, \chi, \phi\}$. Con A_i^j indicando el elemento j de la lista A_i, distinguen tres casos:

1. $A_{i-1}^j \leq A_i^{j-1} < A_i^j$: el código es $\phi \cdot (A_i^j - A_i^{j-1} - 1)$.



(a)



$T = \{2, 2, 1, 0, 0, 0\}$

(b)

Figura 3.1: Ejemplo de Fase 1 de BFS. (a) Índices asignados a los nodos. (b) Aristas restantes después de BFS, junto listado de recorrido T.

Tabla 3.5: Lista de adyacencia para BFS, con v_i siendo el primer nodo de un trozo.

Nodo	Grado	Adyacentes
...	...	...
i	8	13, 15, 16, 17, 20, 21, 23, 24
i + 1	9	13, 15, 16, 17, 19, 20, 25, 31, 32
i + 2	0	
i + 3	2	15, 16
...	...	...

2. $A_i^{j-1} < A_{i-1}^j \leq A_i^j$: el código es $\beta \cdot (A_i^j - A_{i-1}^j)$.

3. $A_i^{j-1} < A_i^j < A_{i-1}^j$: se subdivide en dos subcasos:

   a) Si $A_i^j - A_i^{j-1} - 1 \leq A_{i-1}^j - A_i^j - 1$: el código es $\alpha \cdot (A_i^j - A_i^{j-1} - 1)$.

   b) De otro modo: el código es $\chi \cdot (A_{i-1}^j - A_i^j - 1)$.

Los tipo α y ϕ codifican la diferencia con respecto al elemento previo de la lista (A_i^{j-1}), mientras β y χ con respecto al elemento en la misma posición de la lista de adyacencia del nodo previo (A_{i-1}^j). Cuando A_{i-1}^j no existe se reemplaza por A_k^j, donde $k(k < i-1 \wedge v_k \in C)$ es el índice más cercano a i para cual el grado de v_k no es menor que j, o por un código tipo ϕ en caso que un nodo así no exista.

En la Tabla 3.5 se ilustra un ejemplo de listado de adyacencia, y en la Tabla 3.6 su codificación basada en los casos ya mencionados.

Luego, aprovechan distintos tipos de redundancias en los listados de adyacencia, como se puede ver en la Tabla 3.7, distinguiendo cuatro casos:

1. Un conjunto de líneas idénticas (ver bloque ancho amarillo en la Tabla 3.7) se codifican asignando un multiplicador a la primera línea de la secuencia.

2. Los intervalos con grado constante de nodos (ver bloque consecutivo de 9s en la Tabla 3.7), se codifican por su diferencia.

Tabla 3.6: Codificación BFS del listado de adyacencia en la Tabla 3.5.

Nodo	Grado	Adyacentes
...	...	...
i	8	$\phi 13$, $\phi 1$, $\phi 0$, $\phi 0$, $\phi 2$, $\phi 0$, $\phi 1$, $\phi 0$
i + 1	9	$\beta 0$, $\beta 0$, $\beta 0$, $\beta 0$, $\chi 0$, $\alpha 0$, $\beta 2$, $\phi 5$, $\phi 0$
i + 2	0	
i + 3	2	$\beta 2$, $\alpha 0$
...	...	...

Tabla 3.7: Ejemplo de redundancias a explotar en listado de adyacencia de BFS.

Grado	Adyacentes									
...	...									
0										
9	$\beta 1$,	$\phi 1$,	$\phi 1$,	$\phi 1$,	$\phi 0$,	$\phi 1$,	$\phi 1$,	$\phi 1$,	$\phi 1$,	
9	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 2$,	
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 1$,	$\phi 903$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 223$,	$\phi 900$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 1$,	$\alpha 0$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 1$,	$\beta 0$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 1$,	$\beta 0$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 1$,	$\beta 0$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 1$,	$\beta 0$
10	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\alpha 76$,	$\alpha 232$
9	$\beta 0$,	$\beta 1$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$,	$\beta 0$	
...	...									

3. Una secuencia de largo $\mathcal{L}_{min}$ de elementos idénticos (ver el bloque de $\phi 1$s en la Tabla 3.7), se codifica por su largo.

4. Un bloque de filas idénticas (ver gran bloque azul en la Tabla 3.7) que superen un umbral $\mathcal{A}_{min}$, se codifica por su largo.

Para ello introducen un nuevo símbolo Σ, seguido de un indicador Σ_F toma valor 2 si la redundancia es de tipo 3, 3 si es de tipo 4, o 1 si son ambas. Dependiendo de la redundancia a codificar, el código se expresa como '$tipo\, \Sigma\, \Sigma_F\, gap\, l$' o '$tipo\, \Sigma\, \Sigma_F\, gap\, l\, w\, h$', siendo *tipo* uno de los símbolos del set $\{\alpha, \beta, \chi, \phi\}$, *gap* un entero indicando la diferencia, l la cantidad de elementos idénticos en la misma línea, w y h el ancho y alto de las columnas y filas idénticas. En la Tabla 3.8 se ilustra esta codificación para el caso ejemplo de la Tabla 3.7.

Finalmente, usan códigos Huffman para codificar $\alpha, \beta, \chi, \phi$, y proponen una nueva codificación π para cifrar diferencias, Σ, y otros enteros.

¿Es muy extendida esta explicación? Quizás podría detallarse el método un poco menos.

Tabla 3.8: Ejemplo de redundancias codificadas de BFS para la Tabla 3.7.

Líneas	Grado	Enlaces
...	...	...
0	0	
0	9	β7, $\phi\,\Sigma\,2\,1\,1$, ϕ0, $\phi\,\Sigma\,2\,1\,2$
0	0	$\beta\,\Sigma\,3\,0\,7\,5$, β1, $\beta\,\Sigma\,2\,0\,4$, β2
0	1	β1, ϕ903
0	0	β223, ϕ900
0	0	β1, α0
3	0	β1, β0
0	0	α76, α232
0	-1	β0
...	...	...

Tabla 3.9: Ejemplo de Re-Pair. Las reglas en la tabla conforman el diccionario asociado a la compresión.

Reglas	String
	singing.do.wah.diddy.diddy.dum.diddy.do
$A \rightarrow .d$	*singingAo.wahAiddyAiddyAumAiddyAo*
$B \rightarrow dd$	*singingAo.wahAiByAiByAumAiByAo*
$C \rightarrow Ai$	*singingAo.wahCByCByAumCByAo*
$D \rightarrow By$	*singingAo.wahCDCDAumCDAo*
$E \rightarrow CD$	*singingAo.wahEEAumEAo*
$F \rightarrow in$	*sFgFgAo.wahEEAumEAo*
$G \rightarrow Ao$	*sFgFgG.wahEEAumEG*
$H \rightarrow Fg$	*sHHG.wahEEAumEG*

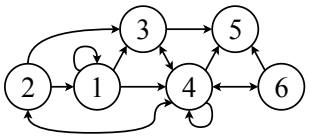
3.1.3. Using Re-Pair, *Claude y Navarro*

El trabajo de Claude y Navarro [11] propone usar Re-Pair [25], un método de compresión basado en gramática, para comprimir el grafo de la Web. Re-Pair consiste en buscar el par de símbolos más frecuente en una secuencia y reemplazar cada ocurrencia por un nuevo símbolo, el cual se añade a un diccionario como nueva regla. En la Tabla 3.9 se muestra un ejemplo simple de Re-Pair, donde las reglas conforman el diccionario para descomprimir la secuencia.

Claude y Navarro [11] aplican esta idea a los listados de adyacencia de un grafo dirigido. Si $V = \{v_1, v_1, ..., v_n\}$ es el listado de n nodos del grafo G, $adj(v_i) = \{v_{i,1}, v_{i,2}, ..., v_{i,a_i}\}$ el listado de a_i nodos adyacentes a v_i, y $\overline{v_i}$ una alternativa de notación para el nodo v_i, proponen concatenar todos los listados de adyacencia con la siguiente notación:

$$T = T(G) = \overline{v_1}\ v_{1,1}, v_{1,2}, ..., v_{1,a_1}, \overline{v_2}\ v_{2,1}, v_{2,2}, ..., v_{2,a_2}, ..., \overline{v_n}\ v_{n,1}, v_{n,2}, ..., v_{n,a_n} \tag{3.2}$$

donde se debe cumplir con $v_{i,j} < v_{i,j+1}$ para cualquier $1 \leq i \leq n$ y $1 \leq j \leq a_i$. Esto significa



(a)

B1	111101
B2	1111001

(c)

$T(G)$	-1	1	3	4	-2	1	3	4	-3	4	5	-4	3	4	5	6	-5	-6	4	5
$7 \rightarrow 4\ 5$	-1	1	3	4	-2	1	3	4	-3	7	-4	3	7	6	-5	-6	7			
$8 \rightarrow 1\ 3$	-1	8	4	-2	8	4	-3	7	-4	3	7	6	-5	-6	7					
$9 \rightarrow 8\ 4$	-1	9	-2	9	-3	7	-4	3	7	6	-5	-6	7							
Borrar < 0	9	9	7	3	7	6	7													

(b)

Figura 3.2: Ejemplo para Re-Pair aplicado a grafos por Claude y Navarro. (a) Grafo de ejemplo. (b) Listado concatenado $T(G)$ y resultado final luego de tres reemplazos y eliminar nodos de referencia. (c) Bitmaps indicadores de nodos de referencia removidos.

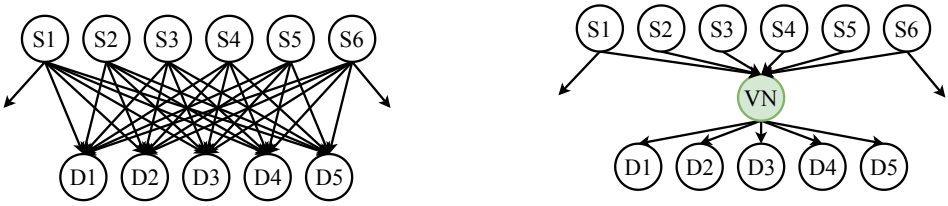
que cada lista de adyacencia concatenada debe partir con su nodo referencia, y todos los nodos de ella deben estar ordenados de menor a mayor. Luego se van reemplazando recursivamente, sin incluir los nodos de referencia, los pares de símbolos consecutivos más frecuentes por uno nuevo, hasta que no sea conveniente, y cada reemplazo se guarda como regla.

Finalmente, se eliminan los nodos de referencia con el cuidado de codificar en dos bitmaps: En $B1$ si tienen nodos adyacentes, y en $B2$ la ubicación de dichos nodos en el arreglo final. En la Figura 3.2 se ilustra un ejemplo, donde en (a) se tiene el grafo de ejemplo, en (b) la concatenación $T(G)$ de todos los listados de adyacencia ordenados de menor a mayor, los reemplazos necesarios para su compresión y el arreglo resultante, y en (c) los bitmaps indicadores de los nodos de referencia. Se debe notar que la notación alternativa $\overline{v_i}$ se reemplaza por $-v_i$ en el arreglo. También presentan otras mejoras, como usar diferencias para codificar los listados de adyacencia o reordenar los nodos para aprovechar mejor Re-Pair.

Una propuesta de compresión más reciente, de Maneth y Peternek [28], también generaliza Re-Pair para comprimir grafos, específicamente hipergrafos dirigidos.

3.1.4. Virtual Node Mining, *Buehrer y Chellapilla*

Buehrer y Chellapilla [10] aprovechan que muchos grupos de nodos en el grafo de la Web consisten en conjuntos de sitios que comparten los mismas vecinos directos, lo que genera un grafo bipartito completo. Su propuesta reduce la cantidad de aristas, definiendo nodos virtuales que son agregados artificialmente al grafo entre los dos conjuntos del biclique. Esto reduce la cantidad total de aristas, ya que cada biclique une un set de U nodos con uno de W nodos, genera en total $|U \times W|$ aristas, y usando un nodo virtual



(a) (b)

Figura 3.3: Ejemplo de reemplazo por nodo virtual. (a) Biclique. (b) Biclique con reemplazo de aristas por nodo virtual V.

disminuye a $|U + W|$, un gran ahorro cuando los bicliques son grandes. En la Figura 3.3 se ejemplifica este reemplazo, cambiando de las iniciales 30 aristas del biclique en (a), 19 por un nodo virtual VN en (b), quedando solo 11 aristas restantes.

Este proceso se repite iterativamente hasta que ya no se reducen significativamente más aristas. Luego aplican codificación delta (ver sección 2.2.1) sobre el grafo reducido. La propuesta logra una buena compresión, pero no reportan los tiempos de acceso. Además, su resultado no se ve afectado por el orden de numeración de los nodos, y permite actualizaciones.

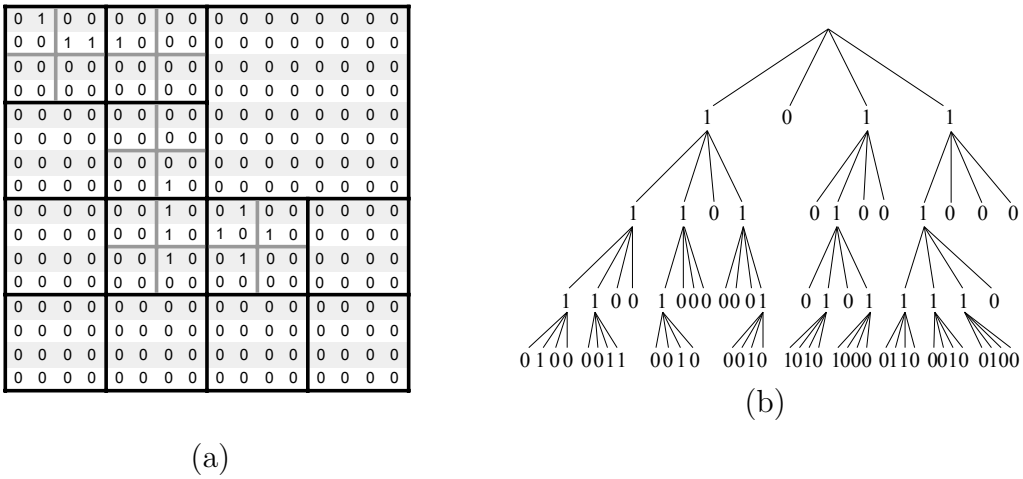
Anh y Moffat [**?**] también aprovechan esta localidad y similaridad en los listados de adyacencia, dividiendo en grupos de h listas consecutivas. Luego reducen las listas aplicando reglas gramáticas como Re-Pair [25] de manera recursiva, y finalmente aplican codificaciones como el código ζ [**?**].

3.1.5. k2-tree, *Brisaboa, Ladra y Navarro*

Una propuesta que aprovecha lo dispersa y agrupada que es la matriz de adyacencia del grafo de la Web es la propuesta por Brisaboa, Ladra y Navarro [8] donde proponen una estructura llamada k2-tree para ahorrar espacio y poder responder consultas tanto de vecinos directos como reversos. Esto último significa que este método, si bien fue pensado para grafos dirigidos, también se puede aplicar para no dirigidos. En un trabajo posterior, lograron mejorar sus resultados aplicando el ordenamiento por BFS antes de crear su estructura [9].

Este modelo propone representar la matriz de adyacencia con un árbol k_2-nario de altura $h = \lceil(\log_k n)\rceil$, donde n es el número de vértices en el grafo. Luego subdivide la matriz de adyacencia en k^2 submatrices de tamaño n^2/k^2, si una de ellas contiene solo ceros se representa solo con un bit en 0, de lo contrario se marcan con un 1 y se vuelven a subdividir de manera recursiva. Esta estructura soporta las consultas de vecinos directos y reversos de manera simétrica, ya que significa revisar las filas o columnas de la matriz. En la Figura 3.4 se presenta (a) un ejemplo de una matriz de adyacencia y sus subdivisiones para k2tree, y (b) un diagrama de la estructura final.

Finalmente, la compresión se realiza representando el árbol usando dos arreglos de bits, un bitmap T para representar la estructura del árbol, y un bitmap L para representar



(a) (b)

Figura 3.4: Ejemplo de k2-tree. (a) Matriz de adyacencia. (b) Diagrama de estructura.

las hojas, que representan las celdas de la matriz. Además usan un bitmap adicional para acelerar la resolución de consultas.

En 2011, Hernández y Navarro [**?**] propusieron combinar varios métodos: Primero reducir la cantidad de aristas con nodos virtuales [10], y luego comprimir usando las propuestas de k2-tree [8] y el trabajo anterior basado en Re-Pair [11], logrando resultados de compresión bastante competitivos, pero sacrificando tiempos de acceso. En 2012, Hernandez y Navarro [22] proponen una estructura bastante similar, pero los bicliques son representados usando estructuras sucintas como wavelet trees y bitmaps comprimidos (ver Sección 3.2). Esta representación comprimida proporciona resolución de consulta de vecinos directos y reversos.

3.1.6. List Merging, *Grabowski y Bieniecki*

Grabowski y Bieniecki proponen agrupar los listados de adyacencia en bloques de h listas cada uno, de manera similar a lo propuesto por Anh y Moffat [**?**]. Cada bloque luego es descompuesto en dos arreglos: el primero es *long list*, un arreglo de todos los índices de los vértices presentes en el bloque de listados de adyacencia, sin repetir ninguno. El segundo es *flags*, un arreglo de bits que permite la reconstrucción de las listas agrupadas.

El arreglo de enteros *long list* es reducido usando las diferencias, terminada en cero y codificada en bytes. El arreglo de bits *flags* indica la pertenencia de los índices en *long list* a sus listas de adyacencias asociadas. El número de bits por cada índice es h, definido como un múltiplo de 8. su largo queda definido por el largo de *long list*. Este arreglo puede no comprimirse (en cual caso lo llaman *LM-bitmap*), o codificarse usando las diferencias entre los 1 sucesivos, escritas en bytes individuales (*LM-diff*).

Ambas secuencias, *long list* y ya sea *LM-bitmap* o *LM-diff*, luego son concatenadas y

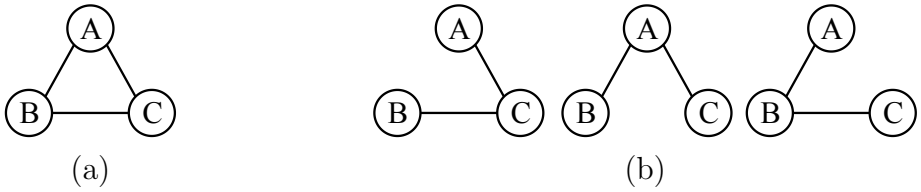


Figura 3.5: Ejemplos de árboles de expansión. (a) Grafo simple. (b) Posibles árboles de expansión para el grafo.

comprimidas usando el algoritmo Deflate. Este algoritmo consiste en una serie de bloques, todos precedidos por una cabecera de 3 bits, que indican lo siguiente:

- Primer bit:
  - 0: No es el último bloque de la secuencia.
  - 1: Es el último bloque de la secuencia.
- Segundo y tercer bit:
  - 00: Bloque sin codificar.
  - 01: Bloque codificado con Huffman, con un árbol de Huffman ya definido.
  - 10: Bloque codificado con Huffman y su árbol asociado.
  - 11: Reservado; No usar.

Para más detalles sobre la codificación Huffman, ver la Sección 2.2.2.

3.1.7. GLOUDS, *Fischer y Peters*

Fischer y Peters [**?**] proponen una nueva estructura de datos sucintos para comprimir grafos con forma de árbol, llamada *GLOUDS (Graph Level Order Unary Degree Sequence)*. La idea del algoritmo es representar mediante otra estructura de datos sucintos para árboles de expansión, llamada *LOUDS (Level Order Unary Degree Sequence)* [**?**], y agregarle información adicional para los nodos extra.

Un árbol de expansión T de un grafo G con n vértices y m aristas, es un árbol que cubre todos los n vértices de G con la mínima cantidad de aristas posibles, sin generar ciclos. En la Figura 3.5 se muestran algunos ejemplos de árboles de expansión para un grafo simple. Un grafo con forma de árbol no tiene una cantidad de aristas adicionales muy alta, con respecto a un árbol de expansión.

La estructura *LOUDS* [**?**] está diseñada para comprimir árboles de expansión. Primero, por conveniencia, agrandan el árbol con un nodo *super-raíz* artificial conectado al nodo raíz original. Luego van construyendo un bitmap B, avanzando por el árbol mediante BFS. Por cada nodo con k hijos, se agregan 1^k0 bits a B (con $k = 3$ sería 1110). Esto representa a cada nodo del árbol dos veces: una vez con un 1 cuando aparece el nodo como

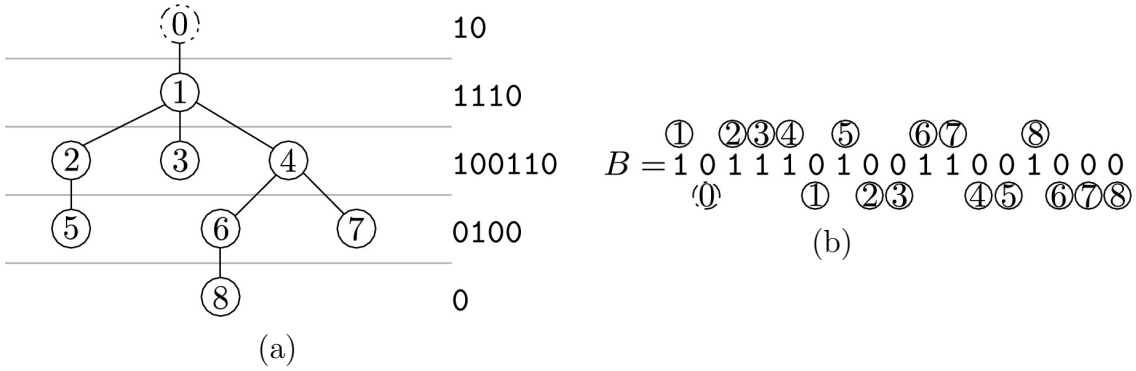


Figura 3.6: Ejemplo de un árbol de expansión y su LOUDS. (a) Árbol de expansión y los bits generados por cada nivel de BFS. (b) Bitmap B de LOUDS.

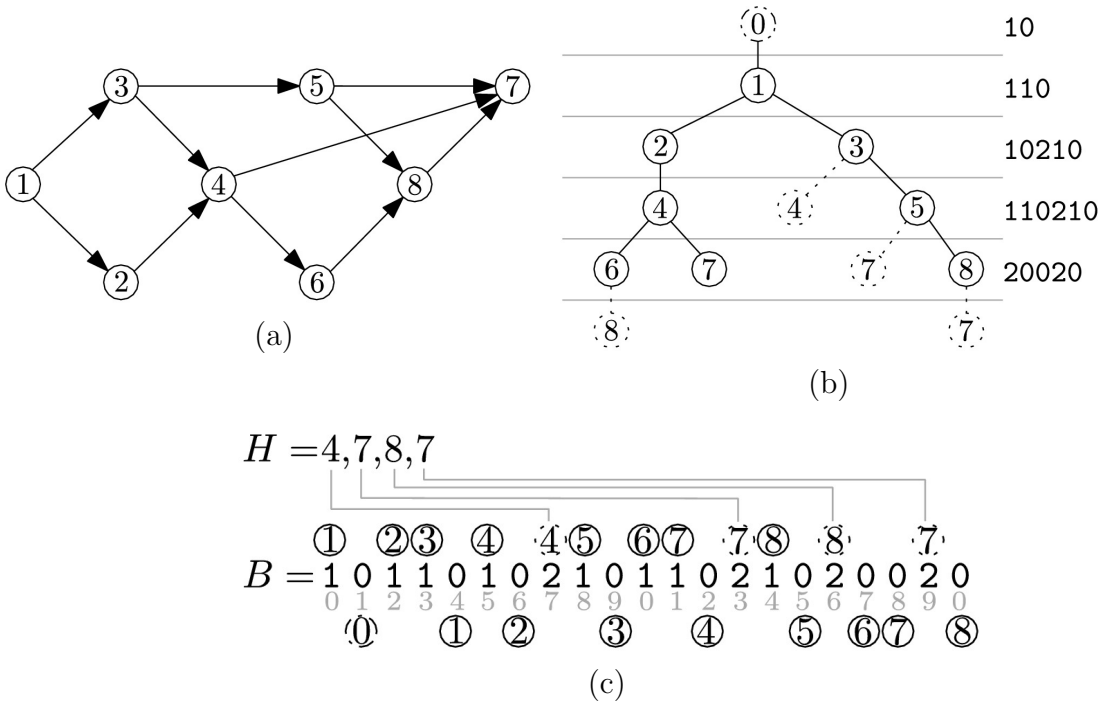
hijo, y con un 0 cuando aparece como padre. Para n nodos, el bitmap B tiene $2n + 1$ bits de largo. En la Figura 3.6 se muestra un ejemplo de esta representación. Los nodos quedan identificados según el nivel de BFS donde aparecen.

Basado en esta idea, *GLOUDS* [**?**] propone una nueva estructura para grafos con forma de árbol. Para esto define las siguientes características de un grafo G:

- $c \leq n$, la cantidad de nodos raíz de G (el tamaño mínimo del set de nodos desde donde se alcanzan todos los nodos restantes).
- $k = m - n + 1$, la cantidad de aristas adicionales, con respecto al árbol de expansión T de G.
- $h \leq k$, la cantidad de nodos adicionales.

Por simpleza, parten asumiendo que existe un nodo r desde donde se pueden alcanzar todos los nodos restantes ($c = 1$). Desde r, realizan el recorrido por BFS de G, lo que llaman BFT, y T_G^{BFT} el árbol resultante. Agrandan T_G^{BFT} tal que por cada vértice w ya visitado durante la BFT del nodo v en una etapa anterior, realizan una copia llamada *nodo sombra* y la agregan como hijo al nodo v en T_G^{BFT}. Finalmente agregan el nodo *super-raíz*, y llaman al árbol final T_G, que tiene exactamente $m + 2$ nodos. En la Figura 3.7 se ilustra un ejemplo de GLOUDS, en (a) el grafo G, y en (b) el árbol T_G final.

Luego proponen una representación de T_G similar a la de LOUDS, pero que permita diferenciar los nodos sombra de los demás. Para esto, cambian el bitmap por una secuencia de trits, con posibles valores del conjunto $\{0, 1, 2\}$. Luego, la secuencia B se genera de la misma manera, pero cuando aparece un nodo sombra en el recorrido BFS se agrega un 2. Estos nodos no se vuelven a representar con un 0. Le llaman al vector de trits B CLOUDS, y tiene un largo de $n + m + c$ trits. En la Figura 3.6 (c) se muestra el resultado para el ejemplo ya mencionado. Además, necesita un arreglo $H[0, k)$ que anota los nodos sombra que aparecen en B.



(a) (b) (c)

Figura 3.7: Ejemplo de la generación de GLOUDS. (a) Grafo G. (b) Árbol T_G. (c) GLOUDS B y el arreglo H de nodos sombra.

3.2. Estructuras compactas

En esta sección se detallan las posibles estructuras compactas basadas en secuencias que se evaluarán para la compresión de la estructura planteada. Las operaciones básicas que permiten son *Rank*, *Select* y *Access*, las cuales se detallan a continuación:

- $Rank_S(a, i)$: Cuenta las ocurrencias del símbolo a hasta la posición i en la secuencia S.
- $Select_S(a, i)$: Encuentra la posición de la ocurrencia i del símbolo a en la secuencia S.
- $Access_S(i)$: Retorna el símbolo en la posición i de la secuencia S.

No tengo claridad de cómo presentar los siguientes trabajos, sobre todo las secuencias binarias, sin tener que entras más en detalle de la teoría.

3.2.1. Secuencias binarias

Existen varias propuestas de estructuras compactas para secuencias binarias, siendo una de las más usadas la de Raman, Raman, Rao [31], que logran las funciones de *Rank* y *Select* en tiempo constante. Además, con $B[1, n]$ un bitmap de largo n con n_0 ceros y

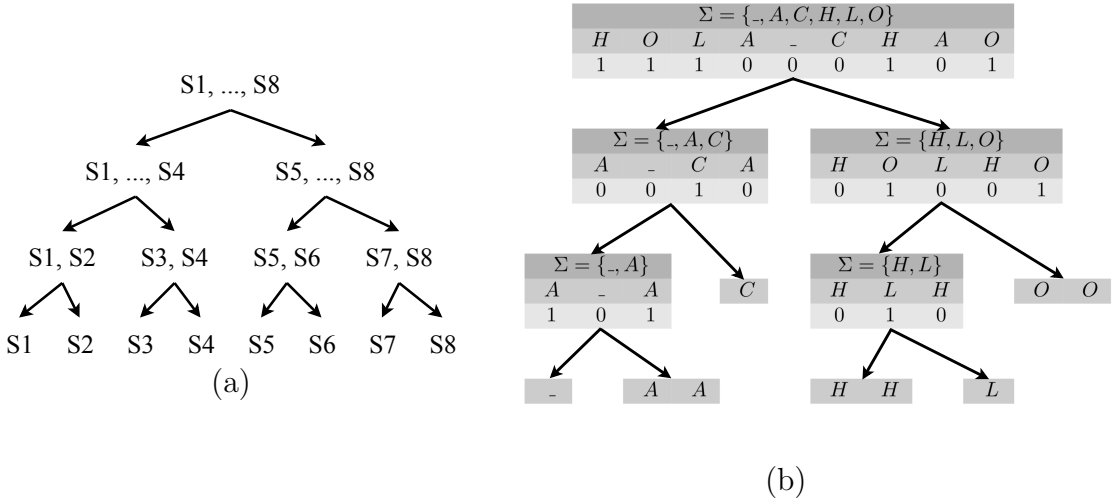


Figura 3.8: Ejemplos de wavelet-tree. (a) Ejemplo básico de subdivisión de secuencia ordenada. (b) Ejemplo práctico con alfabetos y bitmaps por nodo.

n_1 unos, en espacio requiere $nH_0(B) + o(n)$ bits, siendo $H_0(B)$ la entropía de orden cero de B:

$$H_0(B) = \frac{n_0}{n} \log \frac{n}{n_0} + \frac{n_1}{n} \log \frac{n}{n_1} \tag{3.3}$$

Okanohara y Sadakane [29] proponen una mejora que elimina el costo de $o(n)$ en bits para bitmaps poco densos, y si bien el costo para la operación *Select* se mantiene, aumenta para *Rank*.

3.2.2. Wavelet Tree y Wavelet Matrix

Grossi, Gupta y Vitter [20] proponen una estructura llamada **wavelet tree** para codificar secuencias, la cual consiste en un árbol binario donde la raíz y cada nodo interno son bitmaps. Si el símbolo en la secuencia pertenece a la primera mitad del alfabeto de la secuencia de entrada, se marca con un 0 en su correspondiente bit del bitmap asociado, y se escribe en el nodo hijo izquierdo. Si pertenece a la segunda mitad del alfabeto, se marca con un 1 en el bitmap y se escribe en el nodo hijo derecho. En la Figura 3.8 se muestran dos ejemplos de wavelet-tree. En (a) se tiene un ejemplo simple de la subdivisión de una secuencia ordenada. En (b) se presenta un ejemplo práctico de los alfabetos, la subdivisión de la secuencia y los bitmaps por nodo del wavelet-tree.

Se construye de la siguiente manera. Dada una secuencia S de largo $n = |S|$, donde $S[i] \in \Sigma$ y $\sigma = |\Sigma|$, se tiene: La raíz del árbol consiste en un bitmap B donde $B[i] = 0$ si $S[i] \in [0, \frac{\sigma}{2}]$ y $B[i] = 1$ si $S[i] \in [\frac{\sigma}{2} + 1, \sigma]$. El siguiente nivel del árbol se construye basado en los símbolos asociados al nodo por el bitmap B padre. Para $B[i] = 0$, se crea el bitmap hijo de la izquierda, y el alfabeto asociado a esos símbolos nuevamente se divide en dos, asignando valor a dicho bitmap siguiendo el mismo procedimiento descrito

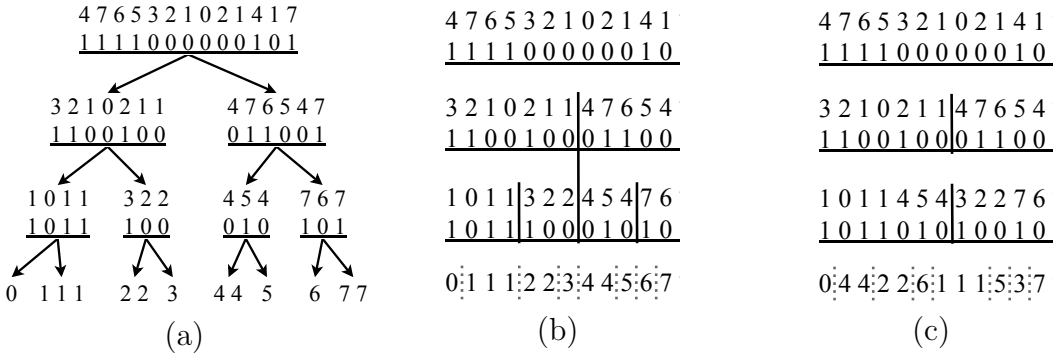


Figura 3.9: Ejemplos para wavelet-matrix. (a) Un wavelet tree. (b) El mismo wavelet tree sin punteros. (c) La wavelet matrix correpondiente.

anteriormente. De manera similar, para $B[i] = 1$ se crea el bitmap derecho y se sidue el mismo procedimiento. Esto se repite por cada nodo hasta llegar a símbolos únicos en cada nodo terminal. Se requiere guardar los punteros a cada nodo hijo izquierdo y derecho, con los que se permite la navegación por el árbol.

Mejorando la propuesta de wavelet tree [20], Claude, Navarro y Ordóñez [12] proponen una nueva estructura llamada **wavelet matrix**. Primero crean una versión de wavelet tree sin punteros, reemplazando cada bitmap por nodo por solo un bitmap por nivel, y contando la cantidad de ceros pueden determinar las subdivisiones pertinentes en cada nivel.

Luego, para crear la wavelet matrix, liberan a la estructura de la suposición que los hijos de un nodo v deben ir alineados. Esto les permite diseñar un mecanismo de clasificación mas sencillo entre un nivel y otro: todos los ceros pasan a la izquierda y todos los unos a la derecha. Por cada nivel, guardan en un entero z_ℓ la cantidad de ceros del nivel ℓ. En la Figura 3.9 se presentan en (a) un wavelet tree de ejemplo, en (b) su correspondiente wavelet tree sin punteros, y en (c) su wavelet matrix, donde las líneas verticales representan el valor de z_ℓ para cada nivel.

También prueban otras alternativas de representación, como usar Huffman [24] tanto para la representación sin punteros del wavelet-tree como para la wavelet matrix. Su resultado apunta a la superioridad tanto en tiempo como espacio en disco de wavelet matrix sobre wavelet-tree, lo que se tendrá en consideración a la hora de evaluarlos como alternativas de compresión.

3.2.3. SDSL - Succinct Data Structure Library

Gog, Beller, Moffat y Petri [19] desarrollan la librería `SDSL`[1] (Succinct Data Structure Library), desarrollada en `C++11` y registrada bajo `GPLv3`, donde se pueden utilizar las estructuras planteadas en las secciones anteriores, entre muchas otras más.

[1] `https://github.com/simongog/sdsl-lite`

3.3. Detección de cliques maximales

La detección de cliques maximales en un grafo es un problema NP-Hard [**?**]. Se han buscado una solución desde varios enfonques [**?**, 13, 21, 6, **?**, **?**]. siendo la más destacada para este trabajo lo propuesto por Eppstein y Strash [**?**], basado en el trabajo de Eppstein, Löffler y Strash [**?**], enfocado a grafos poco densos.

Eppstein et al. [**?**] presentan una modificación al algoritmo de Bron–Kerbosch [**?**], que permite encontrar los cliques maximales de un grafo poco denso, con n nodos y *degeneracy* d, en un tiempo $O(dn3^{d/3})$.

Antes de detallar el funcionamiento de los algoritmos, primero es necesario definir lo siguiente: Sea un grafo $G = (V, E)$, con n vértices y m aristas. Para un vértice v se define $\Gamma(v)$ como el set $\{w|(v, w) \in E\}$, llamado la *vecindad* de v, y similarmente para un subset $W \subset V$ se define $\Gamma(W)$ como el set $\cap_{w \in W}\Gamma(w)$, llamado la *vecindad común* de todos los vértices en W.

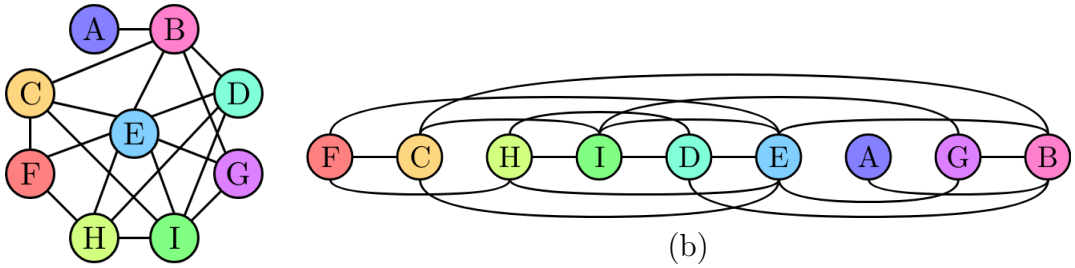
El algoritmo de Bron–Kerbosch [**?**] es un algoritmo *backtracking* recursivo, sencillo y muy usado para encontrar el listado de cliques maximales de un grafo. Una llamada recursiva entrega tres sets separados de nodos: R, P, y X. R es un clique (posiblemente no maximal), y $P \cup X = \Gamma(R)$ son todos los vértices adyacentes a cada vértice en R. Los nodos en P son aquellos que serán considerados para añadirse a R, y los pertenecientes a X deben ser excluidos del clique. El algoritmo elige un candidato en $v \in P$ para añadirlo al clique R, realiza una llamada recursiva con v ya movido de P a R, y con X restringido a los vecinos de v. Cuando la llamada recursiva retorna, v se mueve a X para evitar trabajo redundante. Cuando la recursión llega al punto donde P y X están vacíos, R es reportado como un clique maximal. Para obtener todos los cliques maximales, se debe iniciar la recursión con P igual a todos los nodos del grafo, y R con X vacíos.

También describen la heurística llamada *pivoting*, que limita la cantidad de llamadas recursivas realizadas por el algoritmo. Para cualquier nodo $u \in P \cup X$, llamado *pivot*, cualquier clique maximal debe contener algún nodo no vecino de u, incluido si mismo. Por tanto, se retrasa que los vértices en $P \cap \Gamma(u)$ sean añadidos al clique, beneficiando realizar menos llamadas recursivas. Tomita et al. [**?**] demuestran que eligiendo el *pivot* u para maximizar $|P \cap \Gamma(u)|$, se garantiza un orden de tiempo $O(3^{n/3})$.

Eppstein et al. [**?**] prueban que el orden de procesamiento de los vértices de G por el algoritmo de Bron–Kerbosch también es importante. Lo primero que hacen es un ordenamiento por *degeneracy* de los nodos del grafo, y en ese orden hacen las llamadas recursivas del algoritmo, usando la regla de *pivot* de Tomita. En la Figura 3.10 se ilustra un ejemplo de ordenamiento por *degeneracy*. Gracias a esto, garantizan que su algoritmo propuesto logra listar todos los cliques maximales en un tiempo $O(dn3^{d/3})$.

3.4. PENDIENTES

Francisco et al. [18] discute algunos algoritmos de compresión de grafos que permiten explotar en forma amigable la computación de matrices, que además son usados en algoritmos de ranking como PageRank [30].



(a) (b)

Figura 3.10: Ejemplo de ordenamiento por *degeneracy*. (a) Grafo con *degeneracy* 3. (b) Posible ordenamiento de nodos por *degeneracy*.

La propuesta de Hernández y Navarro [32], proporciona un buen compromiso entre espacio y tiempo de acceso a vecinos directos. En dicho trabajo, se propone una transformación del grafo dirigido original donde se reduce el número de arcos originales usando nodos virtuales para conectar subgrafos densos representados por grafos bipartitos completos, que incluyen cliques, y luego se apliacan distintos algoritmos de ordenamientos de vértices sobre el grafo transformado. En particular, se obtienen mejores resultados aplicando ordenamiento LLP y compresión [2] que ordenamiento BFS (como lo reportado en [23]).

Capítulo 4

MÉTODO DE COMPRESIÓN PROPUESTO

En esta sección se procede a desarrollar el algoritmo de compresión de grafos dispersos, usando estructuras compactas y aprovechando la redundancia de vértices del grafo en sus cliques maximales.

Esto consta de tres etapas. La primera consta de listar todos los cliques maximales del grafo, utilizando el algoritmo de Eppstein et. al. [15, 14]. Luego se define una heurística eficiente para agrupar o particionar los cliques, aprovechando la superposición entre ellos. Finalmente se define una estructura compacta basada en secuencias para almacenar las particiones.

4.1. Detección de cliques maximales

La representación del grafo mediante su grafo de cliques (ver Definición 4.1) conlleva un problema, listar los cliques maximales de un grafo. Enumerarlos todos es un problema complejo desde un punto de vista teórico y práctico.

Eppstein et. al. [15, 14] proponen un algoritmo rápido para listar cliques maximales de grafos poco densos. La complejidad de su algoritmo es $O(dn3^{d/3})$ en tiempo y $O(n+m)$ en espacio, siendo d el índice de *degeneracy*, n la cantidad de vértices, y m la cantidad de aristas del grafo (ver Sección 3.3).

Este algoritmo está disponible en el repositorio **Quick Cliques**[1], implementado por los mismos autores. Luego, el problema se concentra en encontrar un método eficiente para dividir en particiones el grafo de cliques, que permita tanto ahorrar espacio como responder consultas sobre el grafo de manera rápida.

Con el listado de cliques maximales, se puede obtener el grafo de cliques del grafo, el cual se define a continuación.

Definición 4.1. *Gafo de cliques*

Dado un grafo $G = (V, E)$ y $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ el conjunto de tamaño N de cliques maximales que cubren G, se tiene $CG_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ un grafo de cliques donde:

1. $V_{\mathcal{C}} = \mathcal{C}$
2. $\forall c, c' \in \mathcal{C}, (c, c') \in E_{\mathcal{C}} \iff c \cap c' \neq \varnothing$

En la Figura 4.1 (a) se muestra un grafo no dirigido de ejemplo, en la Figura 4.1 (b) su listado de cliques maximales, y en la Figura 4.1 (c) el grafo de cliques resultante.

[1] `https://github.com/darrenstrash/quick-cliques`

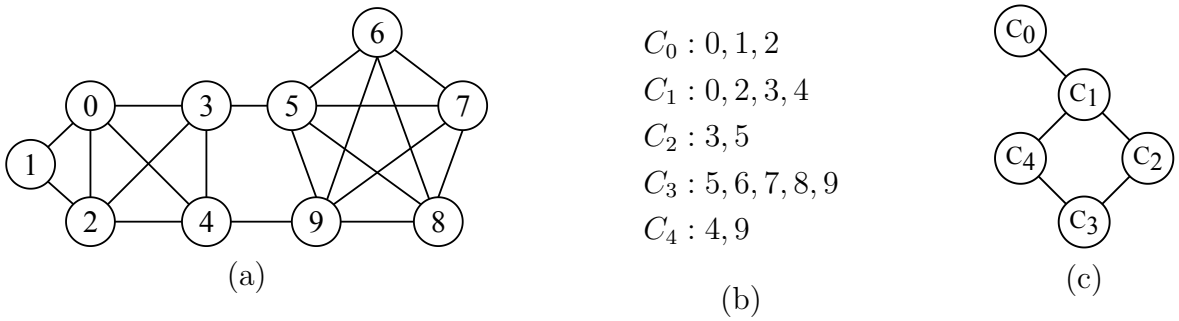


Figura 4.1: (a) Grafo no dirigido. (b) Lista de cliques maximales. (c) Grafo de cliques.

4.2. Particionamento del grafo de cliques

Teniendo el grafo de cliques maximales, es necesario definir una heurística que permita agruparlos en particiones de manera eficiente, pensando tanto en el espacio que ocuparán las secuencias comprimidas como en tiempos de acceso secuencial y aleatorio.

Se desean encontrar particiones del grafo de cliques que exploten dicha redundancia de vértices en los cliques maximales, y permita agrupar en una misma partición a cliques que tengan una cantidad razonable de vértices en común, y los que no la tengan queden separados en otras particiones. El problema de encontrar particiones de cliques se define a continuación.

Problema 4.1. *Encontrar particiones de cliques para el grafo de cliques $CG_{\mathcal{C}}$.*

Dado un grafo de cliques $CG_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$, encontrar un set de particiones de cliques $\mathcal{CP} = \{cp_1, cp_2, ..., cp_M\}$ de $CG_{\mathcal{C}}(V_{\mathcal{C}}, E_{\mathcal{C}})$ con $M \geq 1$, tal que

1. $\bigcup_{i=1}^{M} cp_i = CG_{\mathcal{C}}$
2. $cp_i \cap cp_j = \varnothing$ *para* $i \neq j$
3. *cualquier* $cp_i \in \mathcal{CP}$ *es un subgrafo de* $CG_{\mathcal{C}}(V_{\mathcal{C}}, E_{\mathcal{C}})$ *inducido por el subset de vértices en* cp_i

Esto indica que cada partición es un subgrafo del grafo de cliques maximales del grafo $G(V, E)$. El punto 2 es importante, ya que prohíbe que un clique se repita en una partición, no así un subset de vértices de grafo $G(V, E)$ que sí puede estar en varias particiones a la vez.

A continuación se plantea una heurística que, basada en el listado de cliques maximales $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ y funciones de ranking, genere el particionamiento del grafo de cliques sin necesidad de generar dicho grafo.

4.3. Algoritmo de particionamiento o clustering

En esta sección se procede a describir el algoritmo para generar las particiones del grafo de cliques. Para ello, en la Definición 4.2 se define una función de ranking, que valoriza cada vértice según ciertas características. También se detallan ciertas funciones de ranking basadas en la cantidad y tamaño de los cliques maximales donde un vértice se encuentre.

Definición 4.2. *Función de ranking*
Dado un grafo $G = (V, E)$ y $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ el conjunto de tamaño N de cliques maximales que cubren G, una función de ranking es una función $r : V \to \mathbb{R}^+$ que retorna un valor de puntuación para cada vértice $v \in V$.

La heurística de clustering se describe en el Algoritmo 4.1. La salida del cálculo de ranking son los arreglos D y R (Algoritmo 4.1 línea 1), donde D contiene los índices de los cliques donde cada vértice del grafo G participan, y R contiene el valor de puntuación para cada vértice en G. La complejidad del algoritmo de cálculo de ranking se compone primero por pasar por todos los vértices en G en el conjunto de cliques maximales $\mathcal{C}$, y luego ordenar R de mayor a menor. La complejidad total del algoritmo es de $O(L \log L)$, donde $L = \sum_{c_i \in \mathcal{C}} |c_i|$.

Luego se crea un arreglo de bits Z de largo $N = |\mathcal{C}|$ iniciando cada bit en cero, el cual servirá para mantener revisado si un clique ya fue incluido o no en una partición. Se recorre el arreglo R y por cada vértice u, se obtienen los índices de los cliques donde u participa según $D[u]$ y se añaden el índice id de cada clique a la partición pertinente ($cpid$) solo si $Z[id] = 0$. Si el índice id fue exitosamente agregado, se cambia el valor de $Z[id] = 1$. Si la partición $cpid$ contiene al menos un índice de clique, la partición es agregada a la colección $\mathcal{CP}$, y se continúa procesando vértices en R. La complejidad del algoritmo para este paso es de $O(N + V)$. Finalmente, el algoritmo retorna la colección de particiones $\mathcal{CP}$, donde cada partición contiene un set de los índices de cliques que las componen.

Las funciones de ranking (Definición 4.2) que se proponen toman en cuenta la cantidad y el tamaño de los cliques donde participa cada vértice del grafo $G(V, E)$. Primero se define el conjunto $C(u)$ para cada vértice $u \in V$ como $C(u) = \{c \in \mathcal{C} | u \in c\}$, luego las funciones de rankings son las siguientes:

$$r_f(u) = |C(u)| \tag{4.1}$$

$$r_c(u) = \sum_{c \in C(u)} |c| \tag{4.2}$$

$$r_r(u) = \frac{r_c(u)}{r_f(u)} \tag{4.3}$$

La función $r_f(u)$ (ec. 4.1) indica en cuántos cliques está presente el vértice u, la función $r_c(u)$ (ec. 4.2) entrega la suma del tamaño de los cliques donde está presente el vértice u,

Algorithm 4.1 Algoritmo de particionamiento del grafo de cliques.

Require: $\mathcal{C}$ maximal clique collection ($N = |\mathcal{C}|$), ranking function $r(u)$
Ensure: Returns clique-graph partition collection $\mathcal{CP}$

```
(D, R) ← computeRanking(r, C) (array D y R, ∀u ∈ V )
Initialize bit array Z of size N and set each bit to 0
for u ∈ R do
  cpid ← ∅
  for id ∈ D[u] and D[u] = 0 do
    Z[id] ← 1
    cpid ← cpid ∪ {id}
  end for
  if cpid ≠ ∅ then
    CP ← CP : cpid
  end if
end for
return CP
```

$u \in G$	0	1	2	3	4	5	6	7	8	9
R_{rf}	2,0	1,0	2,0	2,0	2,0	1,0	1,0	1,0	1,0	2,0
R_{rc}	7,0	3,0	7,0	6,0	6,0	7,0	5,0	5,0	5,0	7,0
R_{rr}	3,5	3,0	3,5	3,0	3,0	3,5	5,0	5,0	5,0	3,5

(a)

$\mathcal{CP}_{rf}$	C_0 C_1	C_2		C_4	C_3	
$\mathcal{CP}_{rc}$	C_0 C_1	C_2 C_3		C_4		
$\mathcal{CP}_{rr}$	C_3	C_0 C_1		C_2	C_4	

(b)

Figura 4.2: Resultados de las funciones de ranking. (a) Puntaje final. (b) Particiones de cliques.

y la función $r_r(u)$ (ec. 4.3) es la razón entre $r_c(u)$ y $r_f(u)$. En la Figura 4.2 se muestra el resultado de las funciones de ranking para el caso ejemplo, y las particiones de cliques resultantes para cada una.

4.4. Representación en estructuras compactas

En esta sección se detalla la estructura compacta para representar $G(V, E)$ usando las particiones $\mathcal{CP}$ obtenida en la Sección 4.3. Se consideran estructuras de datos compactas basadas en símbolos y secuencias de bits, con soporte para las operaciones de *rank()*, *select()* y *access()*.

4.4.1. Secuencias de la representación de las particiones

La representación de las particiones consta de cuatro elementos, dos secuencias de enteros **X** e **Y**, un mapa de bits **B**, y una secuencia de bytes **BB**, las cuales se describen a continuación.

- La secuencia de enteros **X** consiste en las listas concatenadas de los vértices presentes en los cliques de cada partición.
- El mapa de bits **B** contiene un bit por cada elemento en **X** inicializados en cero, indicando el inicio de las particiones con un uno. Además se agrega un bit extra en uno al final de la secuencia para indicar su final.
- La secuencia de bytes **BB** codifica en qué cliques está presente cada vértice, marcando un 1 en cada bit de cada byte por clique si el vértice pertenece a ese clique.
- La secuencia de enteros **Y** indica cuántos bytes omitir en **BB** para acceder rápidamente a la partición deseada.

La definición formal de la estructura se presenta en la Definición 4.3. Se puede observar de la ecuación 4.6 que $BB_p \in BB$ es una matriz de bytes, donde cada fila representa un vértice u en $X_p \in X$, y las columnas corresponden a los bytes usados por los vértices para marcar los cliques donde participan en la partición.

También se debe notar el caso especial, cuando un clique maximal queda solo en una partición, no ocupa espacio en su BB_p correspondiente. Para poder reconocer estos casos, se agrupan al final de la estructura compacta todas estas particiones, y con esto se puede ahorrar espacio tanto en BB como en Y.

Definición 4.3. *Representación compacta del grafo $G(V, E)$.*

Dado $\mathcal{CP} = \{cp_0, ..., cp_{M-1}\}$, $cp_p \in \mathcal{CP}$, y $cp_p = \{c_0, ..., c_{m_r-1}\}$. Se especifica $bpu_p = \lceil \frac{m_r-1}{8} \rceil$ como la cantidad de bytes por vértice u en X_p, y se definen las secuencias X_p, B_p, BB_p, Y_p como sigue:

$$X_p = \{u \in c | c \in cp_p\} = \{u_0, ...u_{|X_p|-1}\} \tag{4.4}$$

$$B_p = 1 : 0^{|X_p|-1} \tag{4.5}$$

$$BB_p = bb[|X_p|][bpu_p] \tag{4.6}$$

$$bb[i][j] = \begin{cases} \sum_{k=0}^{7} 2^k (u_i \in c_{8j+k}), & bpu_p \neq 0 \\ \emptyset, & otherwise \end{cases}$$

$$Y_p = \begin{cases} |X_p| \times bpu_p + Y_{p-1}, & bpu_p \neq 0 \\ \emptyset, & otherwise \end{cases} \tag{4.7}$$

Cambié la definición de Y, para dejar en claro que si una partición no tiene bytes en BB, tampoco tendrá un Y asociado.

$\mathcal{CP}_{rr}$

C_0	C_1	C_3	C_2	C_4

(a)

X:	0	1	2	3	4	5	6	7	8	9	3	5	4	9
B:	1	0	0	0	0	1	0	0	0	0	1	0	1	0	1
BB:	3	1	3	2	2									
Y:	5													

(b)

Figura 4.3: Ejemplo de reordenamiento y estructura compacta. (a) $\mathcal{CP}_{rr}$ reordenado. (b) Estructura compacta reordenada.

En la Figura 4.3 se presenta la estructura resultante del ejemplo, usando las particiones $\mathcal{CP}_{rr}$ reordenadas. Como se puede apreciar, solo la primera partición tiene dos cliques, por tanto será la única que agregue bytes en la secuencia BB, codificando la pertenencia de cada clique en un bit del byte, requiriendo entonces solo un byte por vértice en X.

La secuencia X se conforma por todos los vértices que conforman los cliques en cada partición, ordenados de menor a mayor. La secuencia B escribe un 1 en cada inicio de una partición más uno extra para indicar el final. Para la secuencia BB, los cliques involucrados son $C_0 : \{0, 1, 2\}$ y $C_1 : \{0, 2, 3, 4\}$, ambos contienen los vértices 0 y 2, codificado con sus bytes en 3, el vértice 1 solo está presente en C_0 y se codifica con su respectivo byte en 1, y los vértices 3 y 4 solo participan en C_1 y sus bytes toman el valor 2. Finalmente la secuencia Y se inicia con un 5 por la cantidad de cliques presentes en la primera partición, y como las siguientes solo tienen un clique, no se agregan más enteros.

4.4.2. Algoritmos de consulta

A continuación se presentan los algoritmos de consulta que soporta la estructura compacta. El Algoritmo 4.3 reconstruye el grafo $G(V, E)$ recorriendo de manera secuencial la estructura compacta. El Algoritmo 4.4 recupera el listado de vecinos para un vértice cualquiera u del grafo $G(V, E)$. El Algoritmo 4.5 revisa si dos vértices son vecinos. El Algoritmo 4.6 recupera el listado de cliques maximales $\mathcal{C}$ del grafo $G(V, E)$.

El algoritmo secuencial (Algoritmo 4.3) consiste en recorrer secuencialmente la estructura compacta, revisando los vecinos de cada partición. Si una partición contiene un solo clique entonces todos los vértices asociados son vecinos. Si contiene más de un clique, para cada vértice en X se comparan sus bytes asociados en BB con todos los demás, y si el resultado es distinto de cero, son vecinos. La cantidad de cliques se determina rápidamente al comparar el valor de la secuencia Y de cada partición con la anterior; si es el mismo valor significa que hay un solo clique, si cambió es que hay más de uno.

Para esto, primero se obtiene la cantidad P de particiones, contando la cantidad de unos en la secuencia B. Para cada una de ellas, se obtiene el índice del inicio (s) y final (e) de la partición en B. Luego se calcula en bpu_p la cantidad de bytes por vértice en la secuencia X, y se copia el listado de vértices de la partición actual a RAM para un rápido acceso. Por cada vértice, se revisan los vértices restantes; si la cantidad de bytes por vértice es cero, se agregan todos los pares de vértices a la reconstrucción del grafo. De lo contrario, se comparan todos los bytes por vértice correspondientes, y si dicha

comparación da algo distinto a cero, se agrega esa arista a ambos vértices involucrados, y se continúa con el siguiente vértice. Cuando se revisan todas las combinaciones de pares de vértices posibles, se prosigue con la partición siguiente. Finalmente retorna el grafo completo G. La complejidad de este algoritmo es $O(P_0 \cdot N^2)$ cuando bpu_p es igual a cero, de lo contrario $O(P_1 \cdot N^2 \cdot bpu_p)$, siendo P_0 el número de particiones con cero bytes por vértice, P_1 las particiones que sí tienen bytes por vértice, y N el largo de las particiones.

El algoritmo para encontrar vecinos de vértices aleatorios (Algoritmo 4.4) primero detecta las particiones donde participa el vértice u en la secuencia X, y luego revisa cada partición detectada. Gracias a las funciones de acceso *rank()*, *select()* y *access()* que soporta la estructura compacta, esta tarea se realiza de manera eficiente.

Primero se cuentan las ocurrencias del vértice u, y por cada una se obtiene el inicio y final de las particiones donde está presente, junto con la cantidad de bytes por vértice y la copia a RAM de los vértices que tiene dicha partición. Luego, por cada vértice en la partición y posible vecino, si bpu_p es cero se agrega directamente dicho vértice al listado de vecinos de u. Si no lo es, se comparan uno a uno los bytes por vértice de u con su posible vecino, y si alguna comparación es distinta de cero, se agrega el vértice en evaluación al listado final y se continúa al siguiente posible. Finalmente retorna el listado de vecinos $N(u)$. La complejidad del algoritmo es $O(M_0 \cdot N)$ cuando bpu_p es igual a cero, y $O(M_1 \cdot N \cdot bpu_p)$ cuando no lo es, siendo M_0 la cantidad de particiones que contienen al vértice en la secuencia X con cero bytes por vértice, M_1 el resto de particiones con bytes por vértice distinto de cero, y N el largo de las particiones.

El algoritmo para revisar si dos nodos son vecinos (Algoritmo 4.5), primero cuenta las ocurrencias de ambos nodos en la secuencia X, y luego revisa de manera ordenada en qué particiones se encuentra cada una de ellas. Si dos ocurrencias coinciden en una partición, revisa si existe algún bit en común entre sus correspondientes bytes de BB, si lo hay entonces son vecinos, de lo contrario continúa buscando otra partición donde vuelvan a encontrarse ambos nodos. La complejidad del algoritmo es $O(M_1 + M_2)$ cuando bpu_p es igual a cero, y $O((M_1 + M_2) \cdot bpu_p)$ cuando no lo es, siendo M_1 el número de particiones que contienen al vértice u_1, y M_2 el número de particiones que contienen al vértice u_2.

El algoritmo para recuperar el listado de cliques maximales (Algoritmo 4.6) también recorre la estructura compacta de manera secuencial, y va recreando los cliques representados por los bytes en la secuencia BB de cada partición.

Al recorrer las particiones de manera secuencial, comienza de igual manera que el Algoritmo 4.3. Primero obtiene la cantidad de particiones P, luego por cada una de ellas obtiene sus índices de inicio (s) y final (e) en B, calcula la cantidad de bytes por vértice bpu_p, y copia a RAM los vértices de la partición. Si bpu_p es cero, quiere decir que todos los vértices pertenecen al mismo clique, por tanto los agrega como un clique directamente. Y si bpu_p es distinto de cero, revisa por cada vértice y cada bit de cada byte la pertenencia de dicho vértice a un clique maximal; si el bit es uno lo agrega, y si es cero lo omite. Finalmente, agrega cada clique detectado al listado final de cliques maximales. La complejidad del algoritmo es $O(P_0 \cdot N)$ cuando bpu_p es igual a cero, y $O(P_1 \cdot N \cdot 8 \cdot bpu_p)$ cuando no lo es, siendo P_0 el número de particiones con cero bytes por vértice, P_1 las particiones que sí tienen bytes por vértice, y N el largo de las particiones.

Las notaciones de complejidad de los algoritmos me causan duda, separando siempre cuando hay o no bytes en BB. Por favor revisar el algoritmo de consulta si dos nodos son vecinos. Además, realicé algunos cambios menores en los demás algoritmos, por favor revisar igual.

Algorithm 4.3 Algoritmo secuencial para reconstruir $G(V, E)$.

Require: X, B, BB, Y
Ensure: Returns $G(V, E)$
Initialize empty graph G
$P \leftarrow rank_1(B, |B|)$
for $p = 1$ **to** P **do**
  $s \leftarrow select_1(B, p)$
  $e \leftarrow select_1(B, p + 1)$
  $bpu_p \leftarrow \frac{Y_{p+1} - Y_p}{e - s}$
  $X_p \leftarrow X[s..e]$
  for $j = 0$ **to** $|X_p|$ **do**
    for $k = j + 1$ **to** $|X_p|$ **do**
      if $bpu_p = 0$ **then**
        Insert (unoriented) edges $(X_p[j], X_p[k])$ into G
      else
        $BB_p \leftarrow HuffmanToBytes(BB[Y_p], BB[Y_{p+1}])$
        $iBBj \leftarrow bpu_p \cdot j$
        $iBBk \leftarrow bpu_p \cdot k$
        for $b = 1$ **to** bpu_p **do**
          if $BB_p[iBBj + b]$ & $BB_p[iBBk + b] \neq 0$ **then**
            Insert (unoriented) edge $(X_p[j], X_p[k])$ into G
            break
          end if
        end for
      end if
    end for
  end for
end for
return G

Algorithm 4.4 Algoritmo para recuperar vecinos $N(u)$ de un vértice $u \in V$.

Require: u, X, B, BB, Y
Ensure: Returns $N(u)$

Initialize empty graph $N(u)$
$occur \leftarrow rank_u(X, |X|)$
for $i = 1$ **to** $occur$ **do**
  $u_p \leftarrow select_u(X, i)$
  $p \leftarrow rank_1(B, u_p)$
  $s \leftarrow select_1(B, p)$
  $e \leftarrow select_1(B, p + 1)$
  $bpu_p \leftarrow \frac{Y_{p+1} - Y_p}{e-s}$
  $X_p \leftarrow X[s..e]$
  for $j = 0$ **to** $|X_p|$ **do**
    if $X_p[j] \neq u$ **then**
      if $bpu_p = 0$ **then**
        Insert $X_p[j] \neq u$ to $N(u)$
      else
        $BB_p \leftarrow HuffmanToBytes(BB[Y_p], BB[Y_{p+1}])$
        $iBBj \leftarrow bpu_p \cdot j$
        $iBBx \leftarrow bpu_p \cdot (u_p - s)$
        for $b = 1$ **to** bpu_p **do**
          if $BB_p[iBBx + b]$ & $BB_p[iBBj + b] \neq 0$ **then**
            Insert $X_p[j]$ into $N(u)$
            break
          end if
        end for
      end if
    end if
  end for
end for
return $N(u)$

Algorithm 4.5 Algoritmo para consultar si dos nodos $u_1, u_2 \in V$ son vecinos.

Require: u_1, u_2, X, B, BB, Y
Ensure: Returns if $(u_1, u_2) \in E$
1: $occur_1 \leftarrow rank_{u_1}(X, |X|)$
2: $occur_2 \leftarrow rank_{u_2}(X, |X|)$
3: $ySize \leftarrow |Y|$
4: $u1_p \leftarrow select_{u_1}(X, 1)$
5: $p1 \leftarrow rank_1(B, u1_p)$
6: $i1 \leftarrow 1$
7: **for** $i2 = 1$ **to** $occur_2$ **do**
8: $u2_p \leftarrow select_{u_2}(X, i2)$
9: $p2 \leftarrow rank_1(B, u2_p)$
10: **while** $p1 < p2$ **do**
11: $i1 \leftarrow i1 + 1$
12: **if** $i1 > occur_1$ **then**
13: **return false**
14: **end if**
15: $u1_p \leftarrow select_{u_1}(X, i1)$
16: $p1 \leftarrow rank_1(B, u1_p)$
17: **end while**
18: **if** $p1 = p2$ **then**
19: **if** $ySize < p1$ **then**
20: **return true**
21: **end if**
22: $s \leftarrow select_1(B, p1)$
23: $e \leftarrow select_1(B, p1 + 1)$
24: $bpu_p \leftarrow \frac{Y_{p1+1} - Y_{p1}}{e-s}$
25: $BB_p \leftarrow HuffmanToBytes(BB[Y_p], BB[Y_{p+1}])$
26: $iBB1 \leftarrow bpu_p \cdot (u1_p - s)$
27: $iBB2 \leftarrow bpu_p \cdot (u2_p - s)$
28: **for** $b = 1$ **to** bpu_p **do**
29: **if** $BB_p[iBB1 + b]$ & $BB_p[iBB2 + b] \neq 0$ **then**
30: **return true**
31: **end if**
32: **end for**
33: **end if**
34: **end for**
35: **return false**

Algorithm 4.6 Algoritmo para recuperar listado de cliques maximales $\mathcal{C}$ de $G(V, E)$.

Require: X, B, BB, Y
Ensure: Returns collection of maximal cliques $\mathcal{C}$

$\mathcal{C} \leftarrow \emptyset$
$P \leftarrow rank_1(B, |B|)$
for $p = 1$ **to** P **do**
  $s \leftarrow select_1(B, p)$
  $e \leftarrow select_1(B, p + 1)$
  $bpu_p \leftarrow \frac{Y_{p+1} - Y_p}{e-s}$
  $X_p \leftarrow X[s..e]$
  if $bpu_p = 0$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup X_p[e..s]$
  else
    $BB_p \leftarrow HuffmanToBytes(BB[Y_p], BB[Y_{p+1}])$
    $CC \leftarrow \emptyset$
    for $j = 0$ **to** $|X_p| - 1$ **do**
      $cluster \leftarrow 0$
      $iBBj \leftarrow bpu_p \cdot j$
      for $b = 1$ **to** bpu_p **do**
        for $k = 1$ **to** 8 **do**
          $CC[cluster] \leftarrow \emptyset$
          if $BB_p[iBBj + b][k] = 1$ **then**
            $CC[cluster] \leftarrow CC[cluster] \cup X_p[j]$
          end if
          $cluster \leftarrow cluster + 1$
        end for
      end for
    end for
  end if
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{CC[1], CC[2], \cdots, CC[cluster]\}$
end for
return $\mathcal{C}$

Capítulo 5

RESULTADOS

En esta sección se presentan las características de los grafos $G(V, E)$ y de la estructura compacta utilizada para evaluar la propuesta, y luego se compara tanto el nivel de compresión como los tiempos de acceso secuencial y aleatorio de los algoritmos propuestos contra otros algoritmos relevantes del área.

Los algoritmos a comparar son actuales en el estado del arte para compresión de grafos, incluyendo la última versión de WebGraph [2], Apostolico and Drovandi [1], y k2tree [9].

Todas las pruebas y experimentos se realizaron en una computadora con un procesador Intel i7 2.70GHz CPU con 12GB de RAM, y los algoritmos fueron implementados con el compilador g++ 8.2.1 con la opción de optimización O3.

5.1. Grafos

Los grafos a evaluar son todos no densos y no dirigidos. Se consideran siguientes los ocho grafos:

- `dblp-2010` y `dblp-2011` de WebGraph[1].
- `snap-dblp` y `snap-amazon` de SNAP[2].
- `marknewman-astro` y `marknewman-condmat` de Quick-Cliques[3].
- `coPapersDBLP` junto a `coPapersCiteseer` de Network repository[4].

En la Tabla 5.1 se muestran la cantidad de vértices ($|\boldsymbol{V}|$), aristas ($|\boldsymbol{E}|$), cliques maximales ($|\boldsymbol{\mathcal{C}}|$), grado medio ($\overline{\boldsymbol{d}}$) y máximo ($\boldsymbol{d_{max}}$) de los vértices de los grafos, valor de degeneracy ($\boldsymbol{D(G)}$), coeficiente de clusterización ($\boldsymbol{C(G)}$) y transitividad ($\boldsymbol{T(G)}$). De ella se pueden apreciar varias características importantes de los grafos.

En cuanto a tamaño, `marknewman-astro` y `marknewman-condmat` son los más pequeños, no superan los 50.000 vértices. Los demás tienen un tamaño bastante similar, siendo `dblp-2011` el más grande con 986.324 vértices.

Con respecto al número de aristas, `marknewman-astro` y `marknewman-condmat` también son los menores con menos de 400.000, luego `dblp-2010`, `dblp-2011`, `snap-dblp` y

[1] `http://law.di.unimi.it/datasets.php`
[2] `https://snap.stanford.edu/data/`
[3] `http://www.dcs.gla.ac.uk/~pat/jchoco/clique/enumeration/quick-cliques/doc/`
[4] `http://networkrepository.com/`

Tabla 5.1: Cantidad de vértices, aristas, cliques, grado medio y máximo de los vértices, degeneracy, coeficiente de clusterización y transitividad de los grafos a comprimir.

Grafo	$\|V\|$	$\|E\|$	$\|\mathcal{C}\|$	$\overline{d}$	d_{max}	$D(G)$	$C(G)$	$T(G)$
marknewman-astro	16.706	242.502	15.794	14,51	360	56	0,66	0,42
marknewman-condmat	40.421	351.386	34.274	8,69	278	29	0,64	0,24
dblp-2010	326.186	1.615.400	196.434	4,95	238	74	0,61	0,39
dblp-2011	986.324	6.707.236	806.320	6,80	979	118	0,63	0,20
snap-dblp	317.080	2.099.732	257.551	6,62	2.752	113	0,63	0,30
snap-amazon	403.394	4.886.816	1.023.572	12,11	343	10	0,41	0,16
coPapersDBLP	540.486	30.491.458	139.340	56,41	3.299	336	0,80	0,65
coPapersCiteseer	434.102	32.073.440	86.303	73,88	1.188	844	0,83	0,77

`snap-amazon` entre 1 y 7 millones, y finalmente `coPapersDBLP` junto a `coPapersCiteseer` con más de 30 millones de aristas.

En cantidad de cliques maximales, la mayoría tiene una cantidad proporcional a su número de vértices, a excepción de tres grafos: `snap-amazon` posee más del doble de cliques que vértices, y tanto `coPapersDBLP` y `coPapersCiteseer` tienen menos de la mitad de cliques que vértices. Esto quiere decir que su clusterización es distinta a los demás, lo que se confirma estudiando los indicadores restantes.

Con respecto al grado medio y máximo de los vértices en los grafos, se destacan `snap-dblp` con 6,62 de media pero 2.752 de máxima, que contrasta con `dblp-2011` que tiene cerca del triple de vértices, aristas y cliques, pero una media similar y un grado máximo tres veces menor. Y `coPapersDBLP` con `coPapersCiteseer` que presentan 56,41 y 73,88 de grado medio, y 3.299 con 1.188 de grado máximo, respectivamente.

Finalmente en los indicadores de clusterización, los mismos grafos `coPapersDBLP` y `coPapersCiteseer` presentan los valores más altos, lo que podría dar un indicio que los resultados de compresión y tiempos de acceso serán distintos a los demás.

La distribución del grado de los vértices para cada grafo se presenta en la Figura 5.1. Se puede apreciar que todos los grafos presentan una distribución similar, donde muchos vértices tienen pocos vecinos, y pocos vértices tienen muchos vecinos.

La distribución de los tamaños de los cliques maximales para cada grafo se muestra en la Figura 5.2. Es importante notar que el gráfico del grafo `coPapersCiteseer` está truncado para efectos de comparación, pero tiene muy pocos cliques por sobre el límite fijado gráficamente.

La mayoría de los grafos contienen muchos cliques con menos de 50 vértices, a excepción de los grafos `snap-amazon`, `coPapersDBLP` y `coPapersCiteseer`. El primero contiene solo cliques pequeños, de no más de 20 vértices. Los otros dos grafos contienen una cantidad considerable de cliques de hasta 100 vértices. Esto permitirá contrastar los resultados del método propuesto entre grafos con distintas cantidades de cliques maximales grandes.

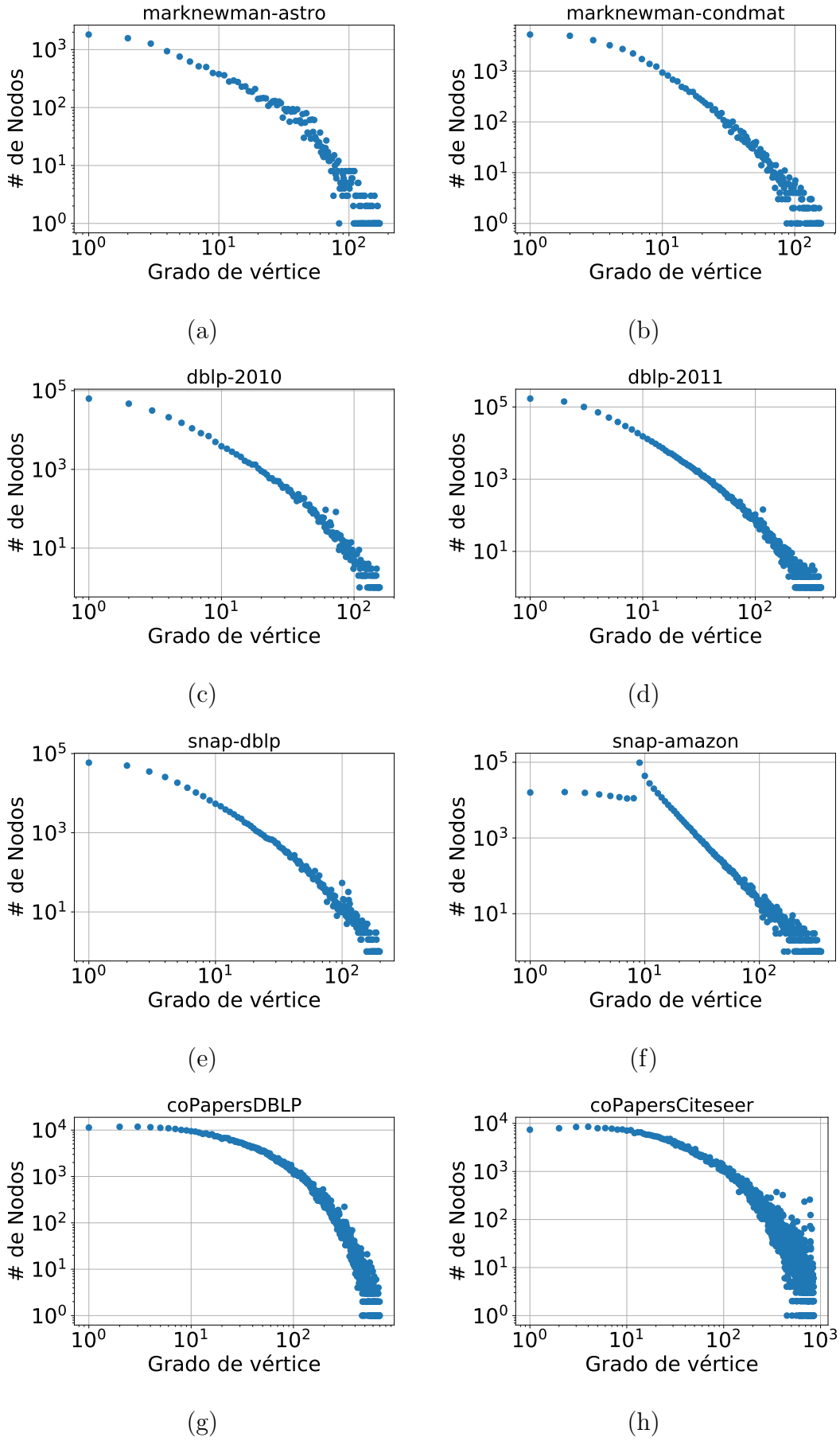
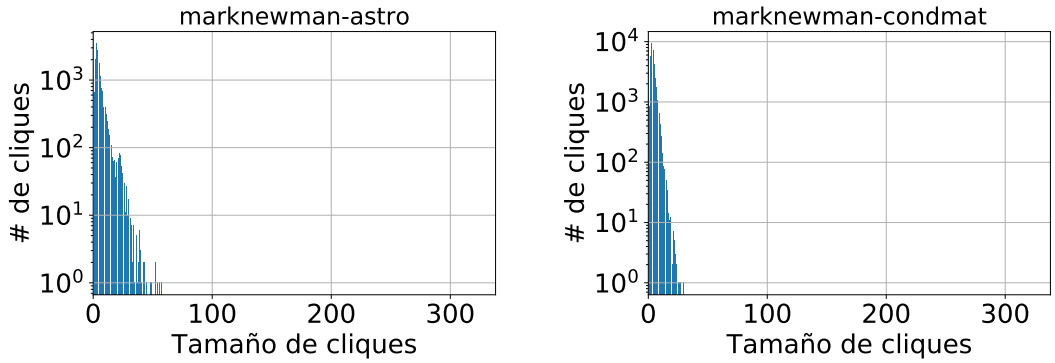
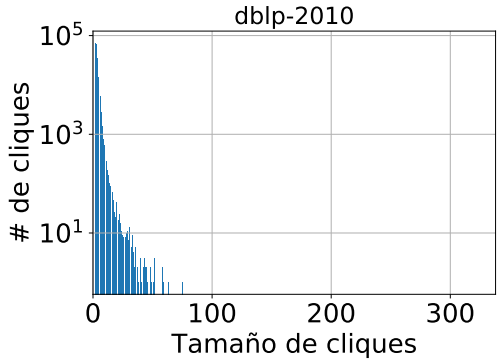


Figura 5.1: Distribución del grado de los vértices para cada grafo.

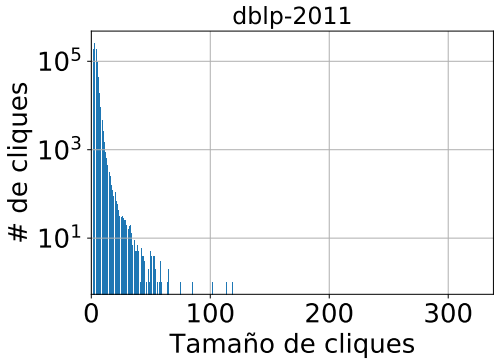


(a)

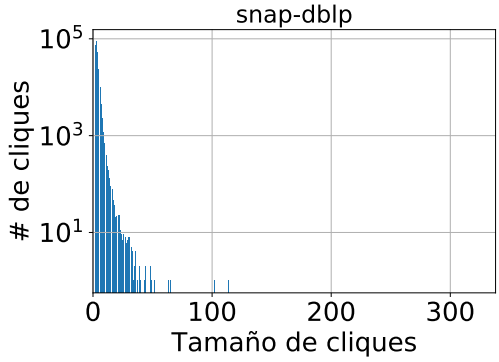
(b)



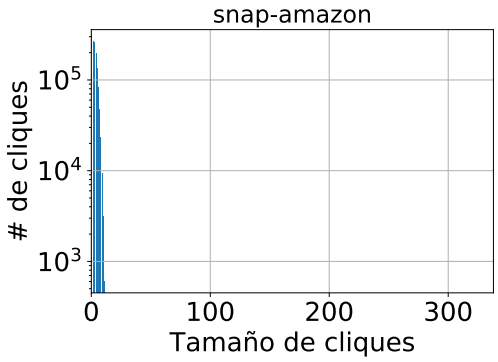
(c)



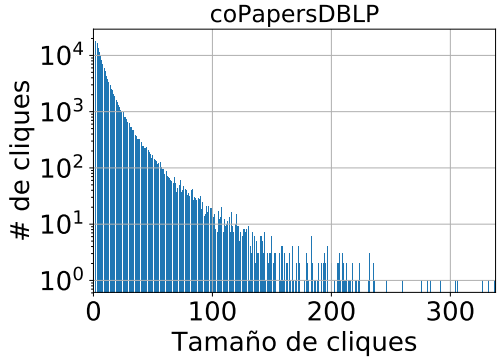
(d)



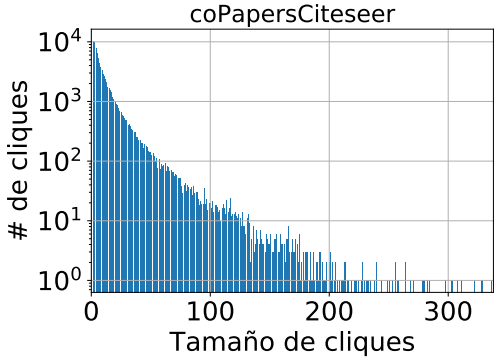
(e)



(f)



(g)



(h)

Figura 5.2: Distribución del tamaño de los cliques maximales para cada grafo.

5.2. Estructura compacta

Para generar la estructura compacta se usa `SDSL`[5] desarrollada por Gog et al.[19]. Las estructuras de datos sucintas a evaluar dependen del tipo de secuencia a compactar.

- Para las secuencias de símbolos, se consideran las estructuras basadas en wavelet matrix (*wm*) [12] y wavelet tree (*wt*) [20].

- Para la secuencia de bits, se consideran las estructuras basadas en bitmaps comprimidos de Raman, Raman y Rao (*rrr*) [31], y Okanohara y Sadakane (*sdb*) [29].

La secuencia de bytes BB se comprime usando código Huffman[24] directamente (ver Sección 2.2.2). Esto significa que la secuencia final de BB se transforma en una secuencia de bits, lo que requiere actualizar los índices de inicio de sus particiones en la secuencia Y, y así poder decodificar directamente los bytes de las particiones correspondientes. Esto también conlleva que, por cada consulta a una partición, primero hay que decodificar todos los bytes correspondientes de esa partición antes de poder usarlos para detectar si los nodos asociados son vecinos o no.

Creo suficiente no entrar más en detalle sobre este cambio, debido a que no es muy complejo.

Se toman como factores de selección el nivel de compresión en bits y tanto el tiempo de reconstrucción secuencial, usando el Algoritmo 4.3, como el tiempo de acceso aleatorio al recuperar los vecinos de un millón de nodos, usando el Algoritmo 4.4, para cada grafo y cada función de ranking con cada una de las estructuras antes planteadas.

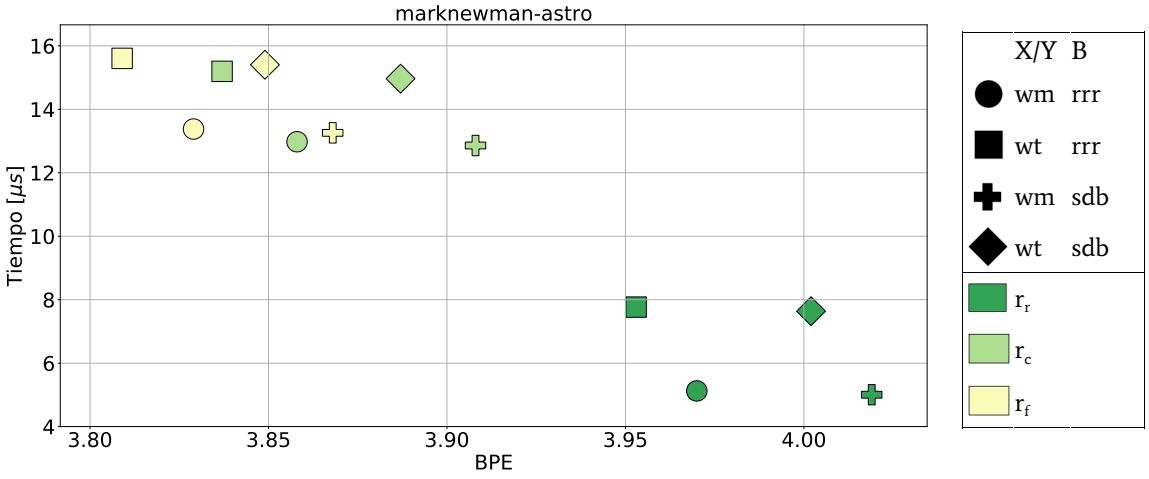
Aquí antes mostraba las tablas con los tamaños en bytes de cada estructura compacta. Las eliminé, ya que creo no aportan en mucho comparado con la comparativa entre BPE y espacio.

Para ello, se decide construir la estructura compacta para cada grafo y cada función de ranking, con todas las posibles combinaciones para las secuencias. En la Figura 5.7, Figura 5.8, Figura 5.9, y Figura 5.10, se compara para cada grafo el BPE de las ocho posibles combinaciones para cada función de ranking con respecto al tiempo secuencial de reconstrucción del grafo. Y en la Figura 5.3, Figura 5.4, Figura 5.5 y Figura 5.6, el BPE con respecto al tiempo de acceso aleatorio al recuperar los vecinos de un millón de nodos.

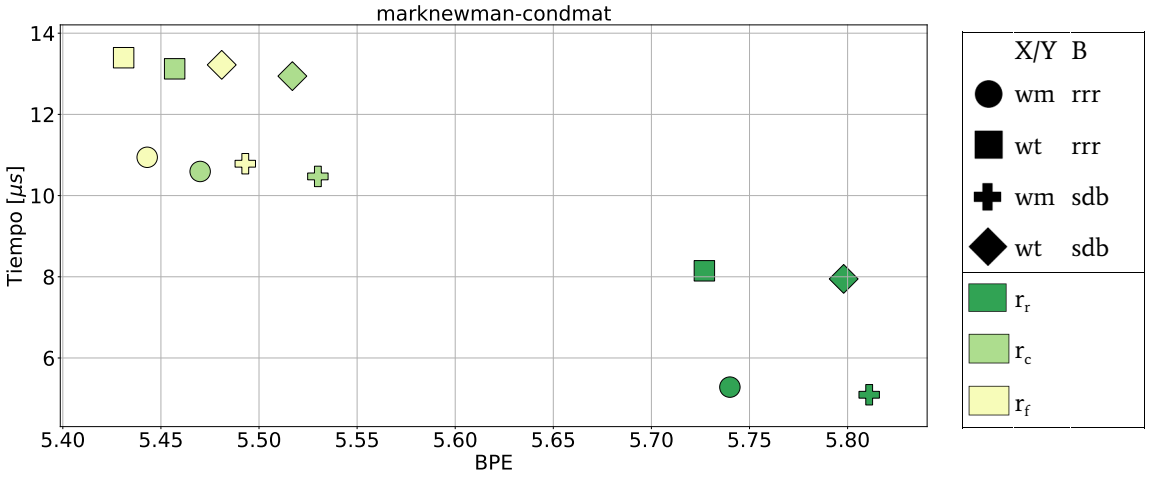
No estoy seguro que esta sea la mejor manera de anotar tantas figuras.

En la mayoría de los casos, la estructura que presenta la mejor relación entre BPE y tiempos es la de secuencias de símbolos con *wm*, secuencia de bytes con *hutu*, y secuencia de bits con *sdb*. Las opciones que presentan menores tiempos aumentan en BPE, y viceversa. Por tanto, se elige esta combinación de estructuras de secuencias como la estructura compacta a desarrollar.

[5] `https://github.com/simongog/sdsl-lite`

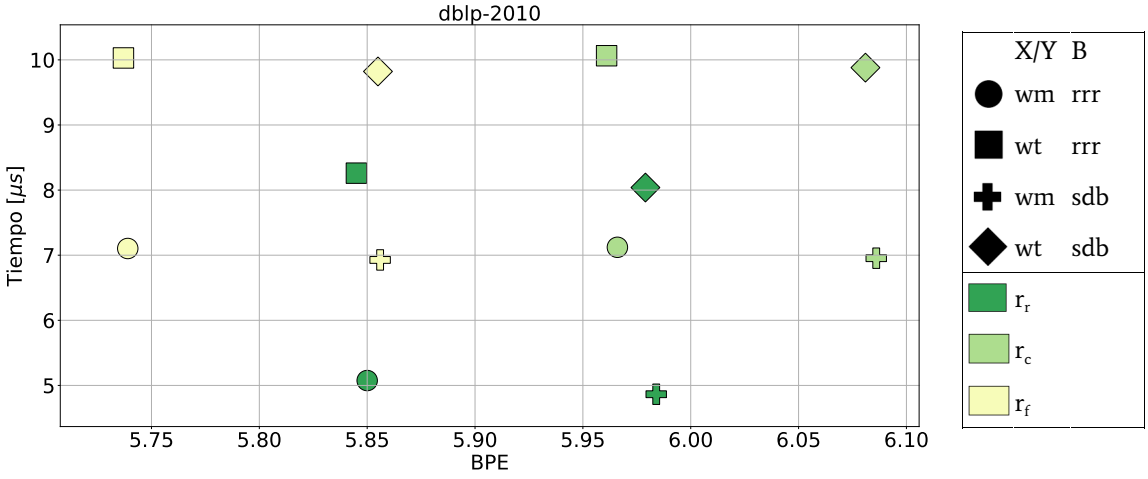


(a)

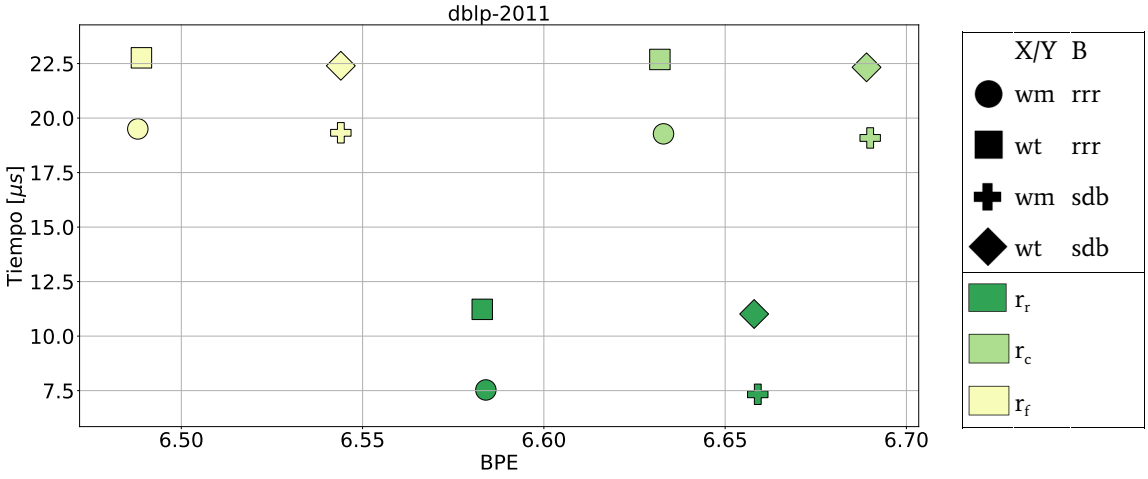


(b)

Figura 5.3: BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking, para los grafos marknewman-astro y marknewman-condmat.

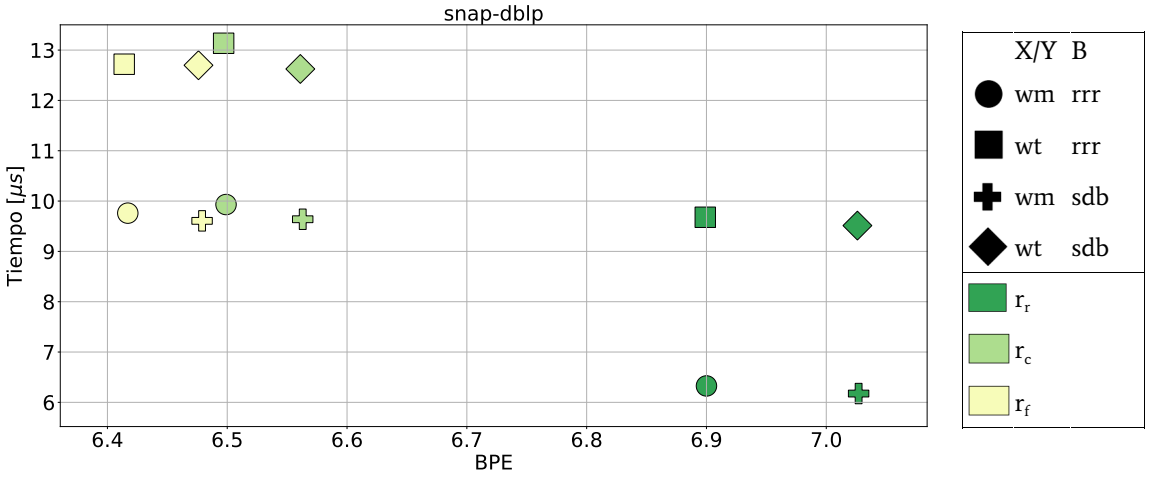


(a)

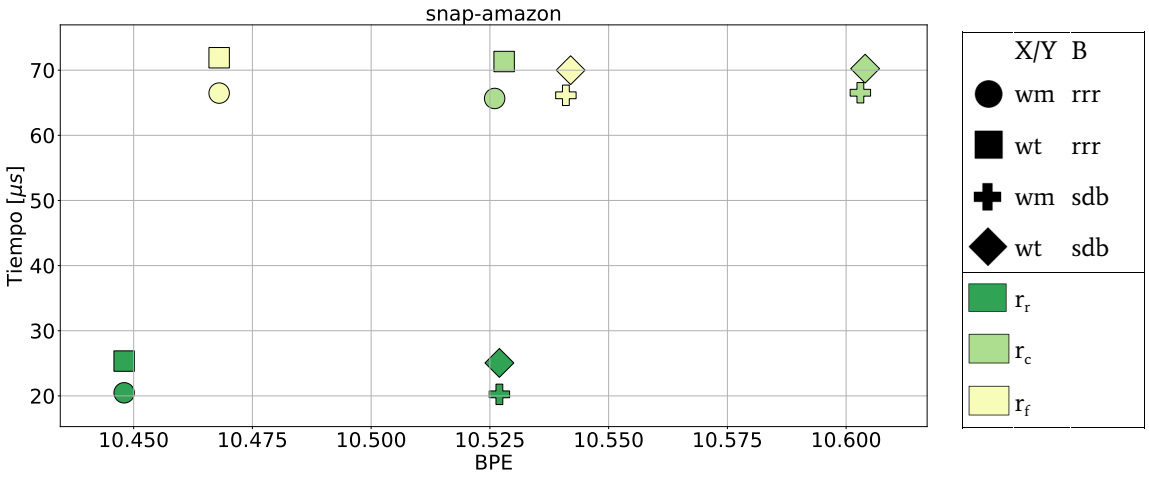


(b)

Figura 5.4: BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking, para los grafos dblp-2010 y dblp-2011.

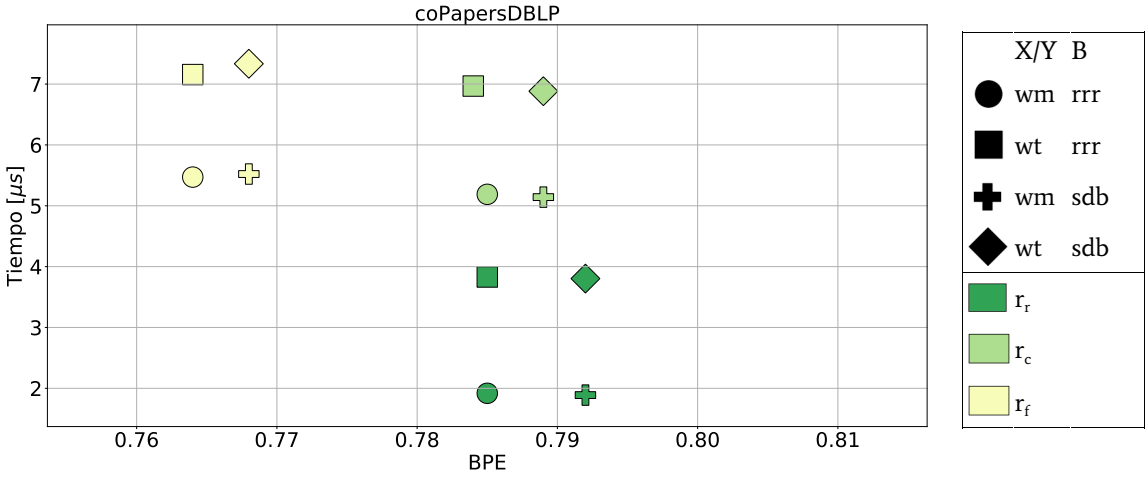


(a)

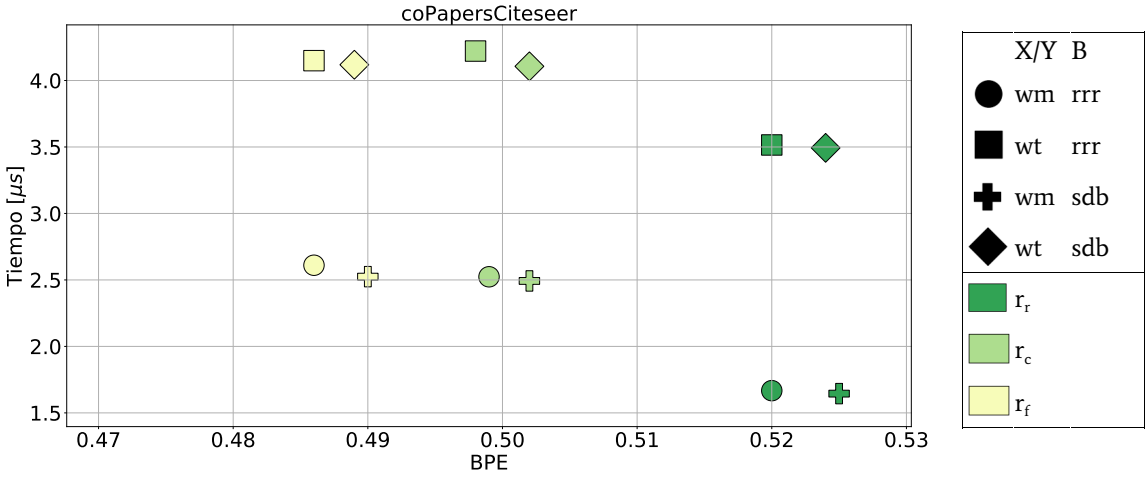


(b)

Figura 5.5: BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking, para los grafos snap-dblp y snap-amazon.

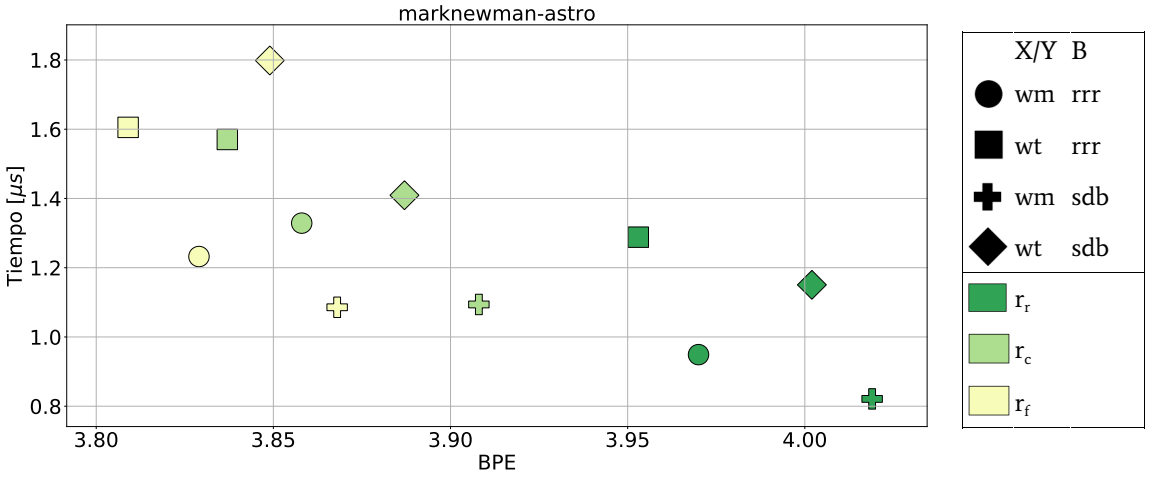


(a)

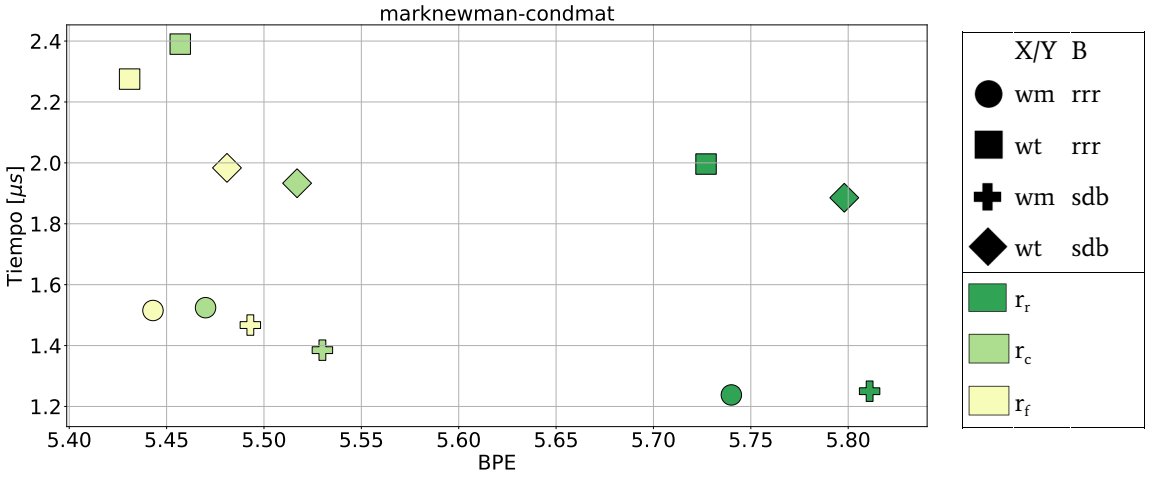


(b)

Figura 5.6: BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking, para los grafos coPapersDBLP y coPapersCiteseer.

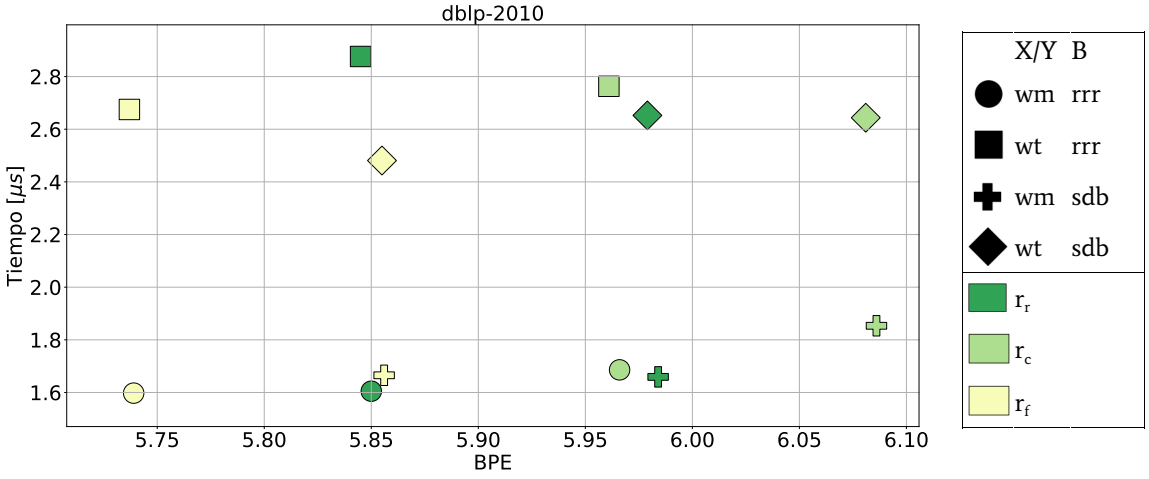


(a)

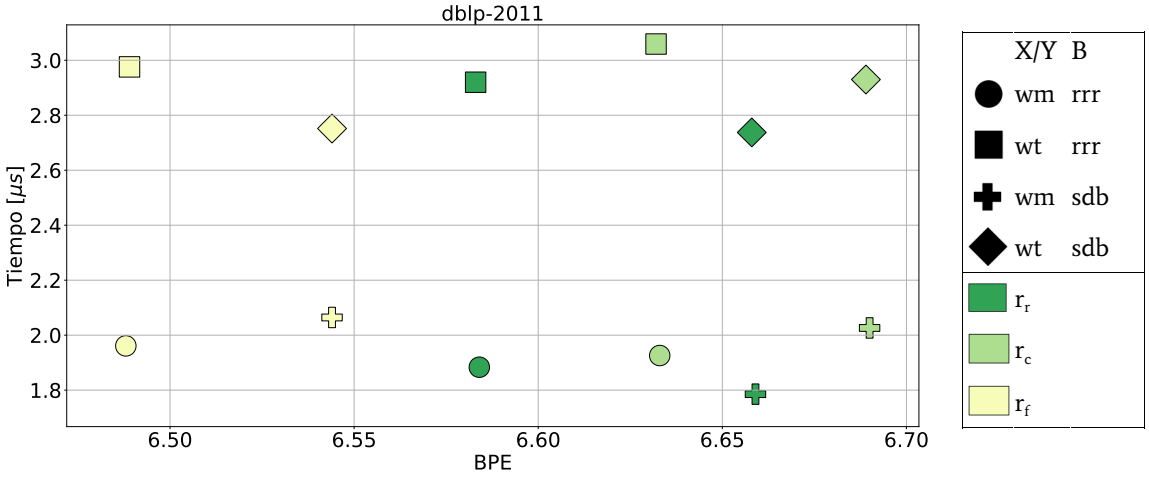


(b)

Figura 5.7: BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking, para los grafos marknewman-astro y marknewman-condmat.

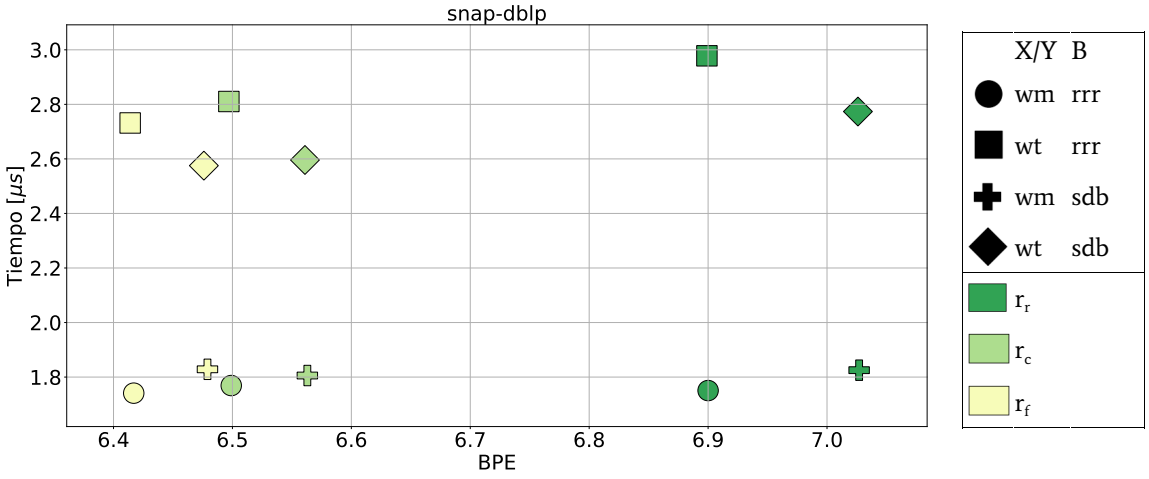


(a)

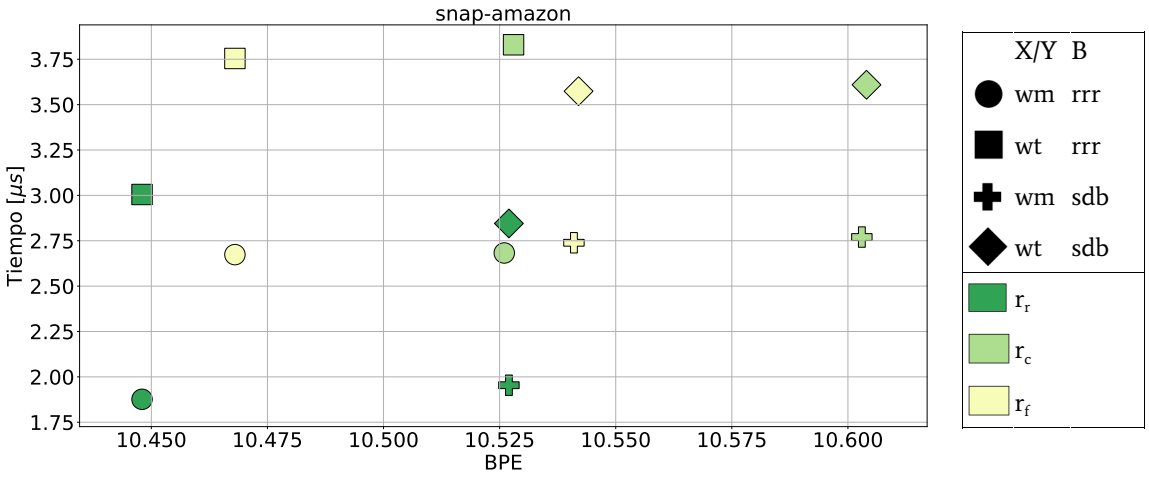


(b)

Figura 5.8: BPE y Tiempo de acceso secuencial medio para posibles estructuras compactas, por cada función de ranking, para los grafos dblp-2010 y dblp-2011.

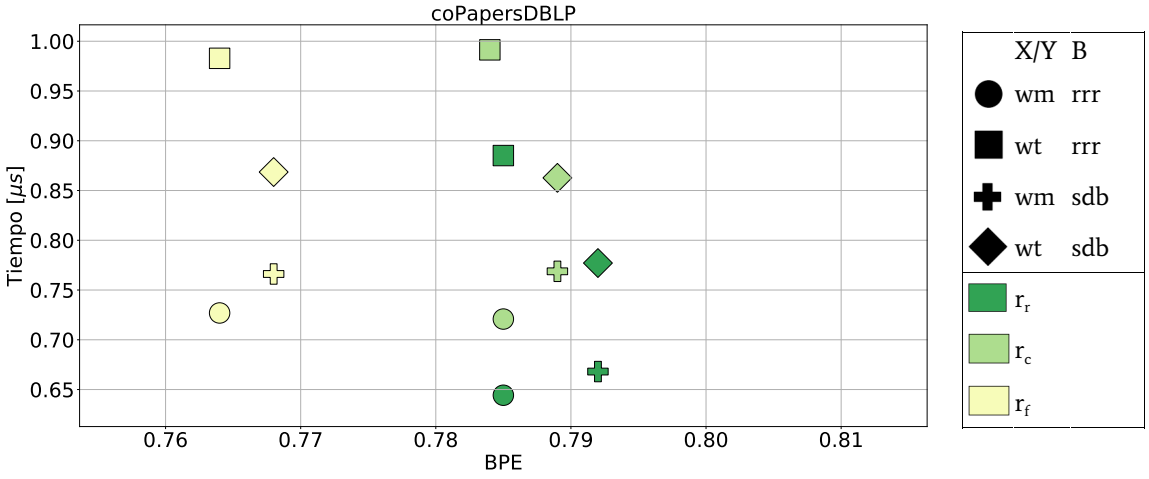


(a)

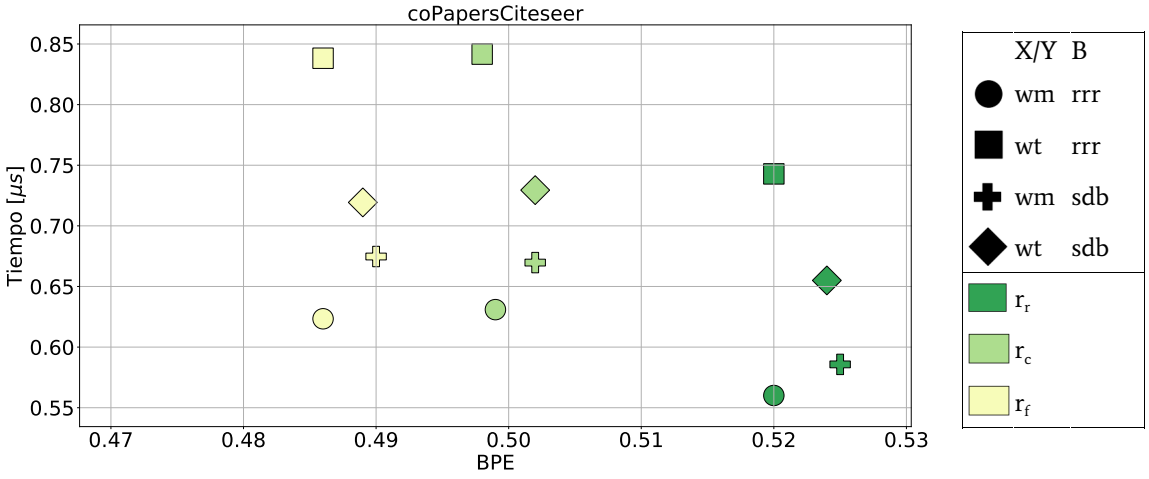


(b)

Figura 5.9: BPE y Tiempo de acceso secuencial medio para posibles estructuras compactas, por cada función de ranking, para los grafos snap-dblp y snap-amazon.



(a)



(b)

Figura 5.10: BPE y Tiempo de acceso secuencial medio para posibles estructuras compactas, por cada función de ranking, para los grafos coPapersDBLP y coPapersCiteseer.

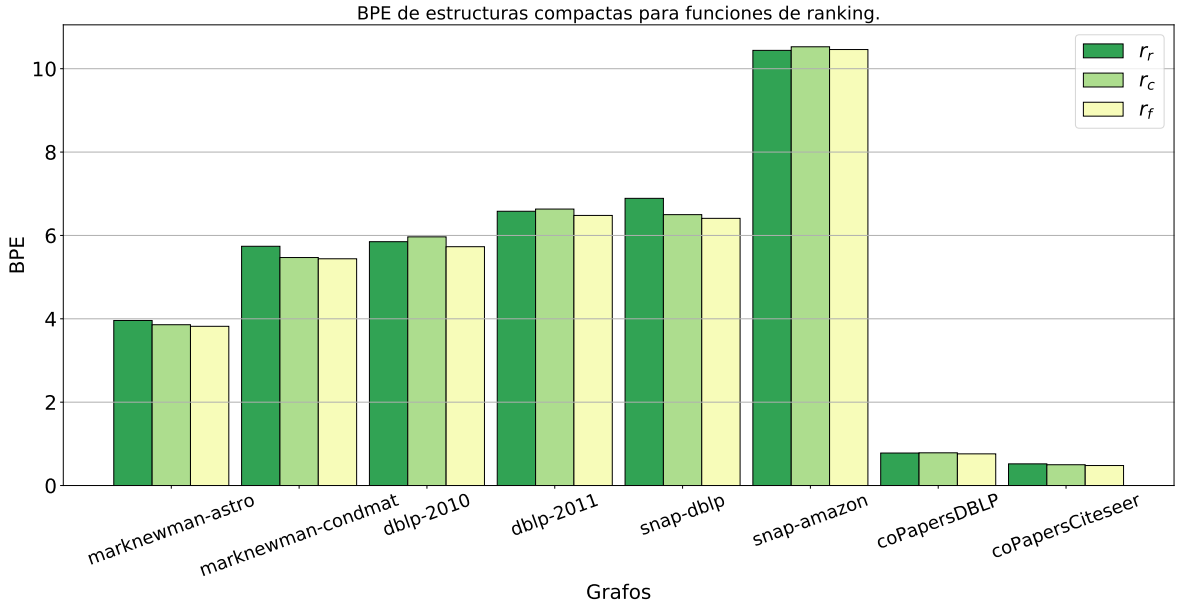


Figura 5.11: BPE de las estructuras compactas para las funciones de ranking.

5.3. Comparación de funciones de ranking

A continuación se compararán las estructuras compactas resultantes, usando las tres funciones de ranking basadas en la frecuencia del vértice en los cliques maximales $r_f(u)$, en la cantidad de vecinos en los cliques del vértice $r_c(u)$, y la razón entre ambas funciones $r_r(u)$.

En la Figura 5.11 se muestran los BPE de las estructuras compactas finales, aplicando las funciones de ranking en la heurística de clusterización. Se puede apreciar que tanto $r_f(u)$ como $r_c(u)$ logran un nivel de compresión muy similar con un BPE parecido en todos los casos, y si bien $r_r(u)$ obtiene un BPE mayor siempre, tampoco es tan lejano al resultado de las demás funciones. Si este parámetro fuera el único a considerar para elegir la mejor función, $r_f(u)$ es siempre la que obtiene el menor BPE, pero es necesario profundizar en la conformación de la estructura.

En la Figura 5.12 se presentan la cantidad de particiones que contiene las estructuras compactas para cada función de ranking. Se aprecia con bastante claridad que la función $r_r(u)$ genera más particiones que las otras dos, y dado que el BPE expuesto en la Figura 5.11 no refleja esta diferencia, se puede intuir que las particiones son más pequeñas. Para profundizar en este punto, se estudiará la composición de las secuencias de las estructuras compactas para cada función de ranking.

En la Figura 5.13(a) se ilustra la proporción de bits para cada secuencia dentro de la estructura compacta, para cada función de ranking. En la Figura 5.13(b) se ilustra la misma proporción normalizada. Como se puede observar, la secuencia que más aporta para todos los casos es la de vértices X, seguida casi siempre de la secuencia de bytes BB,

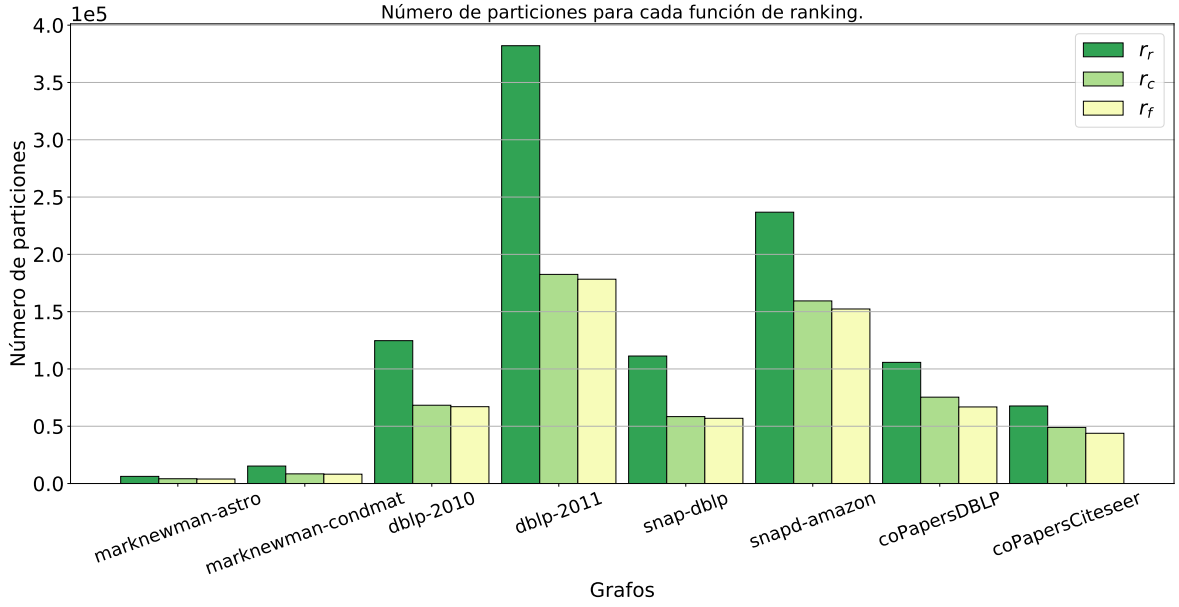


Figura 5.12: Número de particiones en las estructuras compactas para las funciones de ranking.

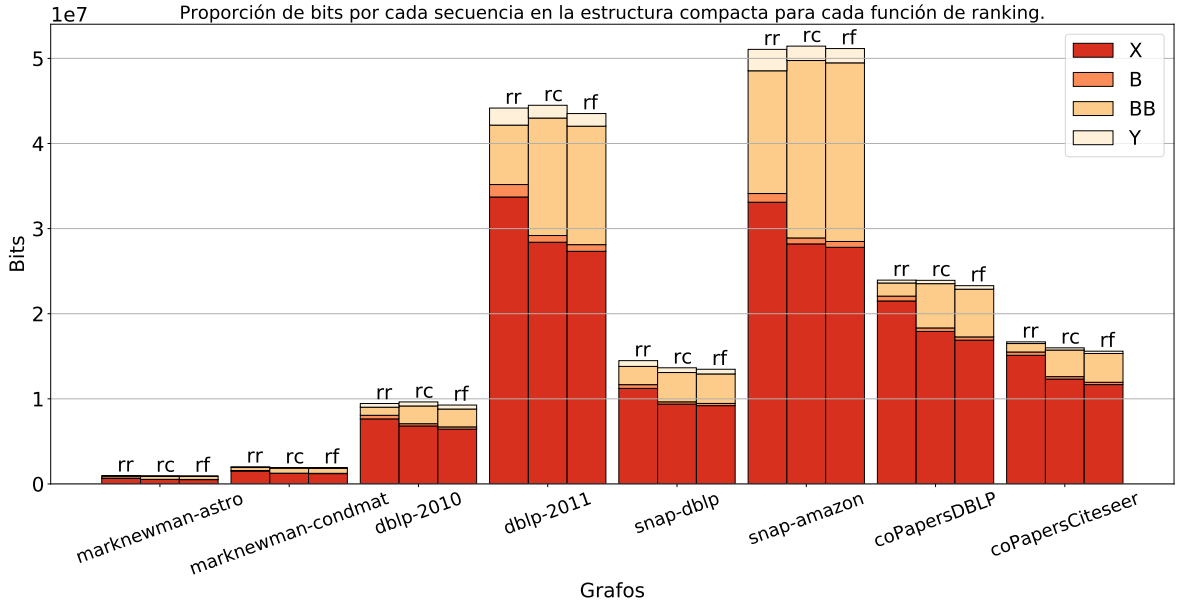
luego Y y finalmente la secuencia de bits B. Nuevamente las funciones de ranking $r_f(u)$ y $r_c(u)$ tienen resultados similares, y para la función $r_r(u)$ la secuencia X aumenta su proporción mientras que BB disminuye. Esto podría afectar positivamente el tiempo de respuesta de los algoritmos propuestos, ya que todos requieren comparar los bytes de BB de cada partición entre ellos, y si hay menos bytes requerirá menos tiempo.

Para estudiar la composición de las particiones para cada función de ranking, en la Figura 5.14 se ilustran la cantidad máxima de vértices en la secuencia X por partición, y en la Figura 5.15 la cantidad máxima de bytes por vértice en la secuencia BB de las estructuras compactas resultantes. Como se puede apreciar, la función $r_r(u)$ presenta consistentemente los menores valores entre las tres funciones, lo que permite asegurar que es la que mejor agrupa y usa el espacio de las particiones en la estructura compacta.

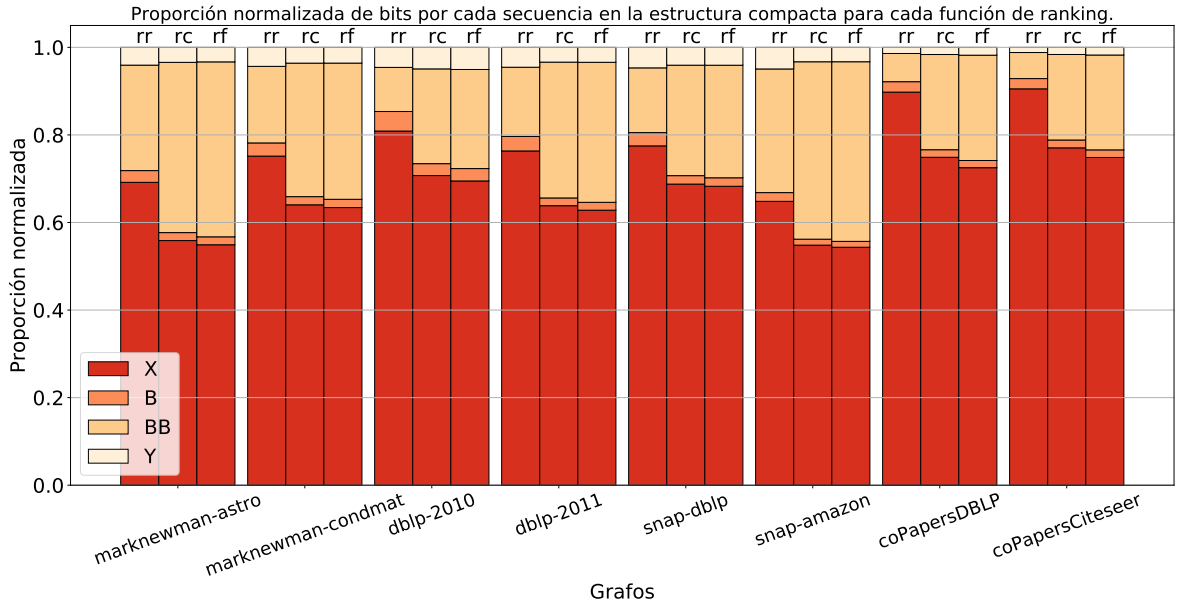
En la Figura 5.16 se puede estudiar la función de distribución acumulativa (CDF) para la cantidad de bytes por vértice en la estructura compacta, para cada función de ranking. Se confirma que para la función $r_r(u)$ las particiones contienen menos bytes en la secuencia BB, ya que la cantidad de bytes por vértice es significativamente menor.

En la Figura 5.17 se ven los tiempos de acceso aleatorio para la obtención de vecinos de cualquier vértice $u \in G(V, E)$ desde las estructuras compactas generadas con las tres funciones de ranking en comparación. Como se esperaba, los tiempos menores se obtienen usando la estructura basada en la función $r_r(u)$, ya que no tiene que comparar tantos bytes por particiones como las otras dos.

Entonces, considerando la gran ventaja en tiempo de acceso y la leve diferencia en compresión, se concluye que la mejor alternativa entre las tres funciones de ranking es



(a)



(b)

Figura 5.13: (a) Proporción de bits por cada secuencia en la estructura compacta, para cada función de ranking. (b) Proporción normalizada.

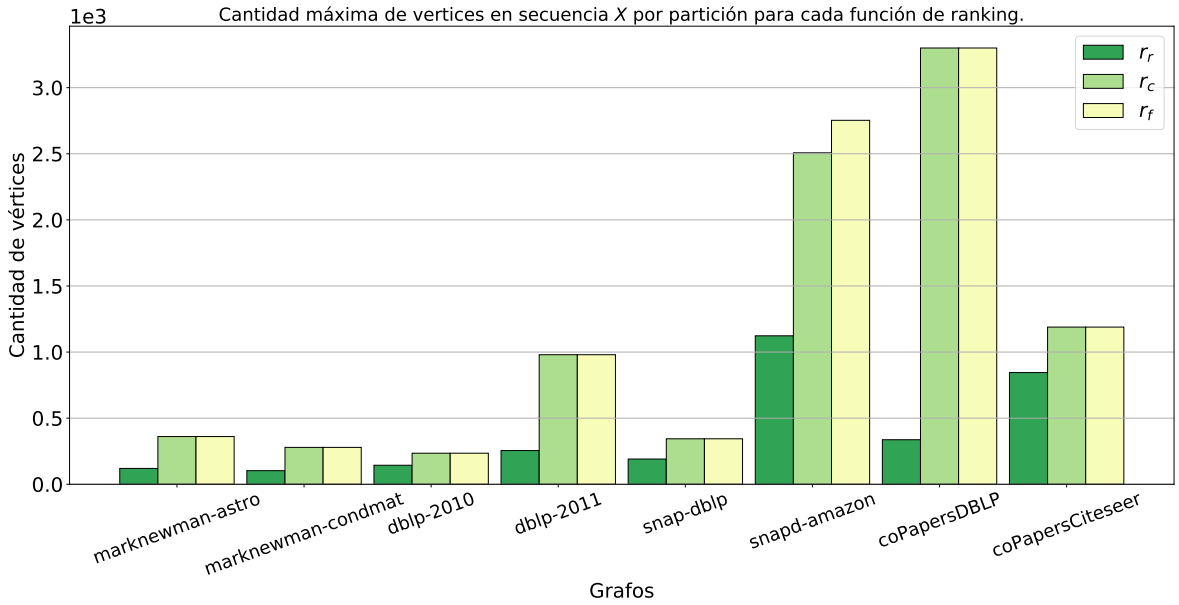


Figura 5.14: Número máximo de vértices en la secuencia X para las funciones de ranking.

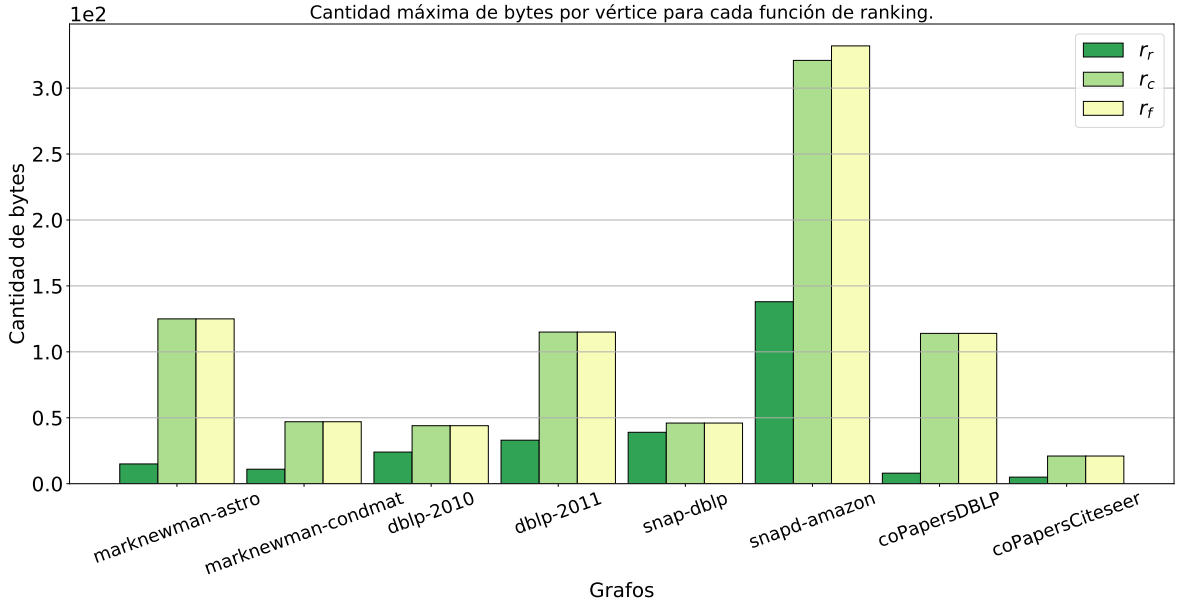


Figura 5.15: Número máximo de bytes por nodo para las funciones de ranking.

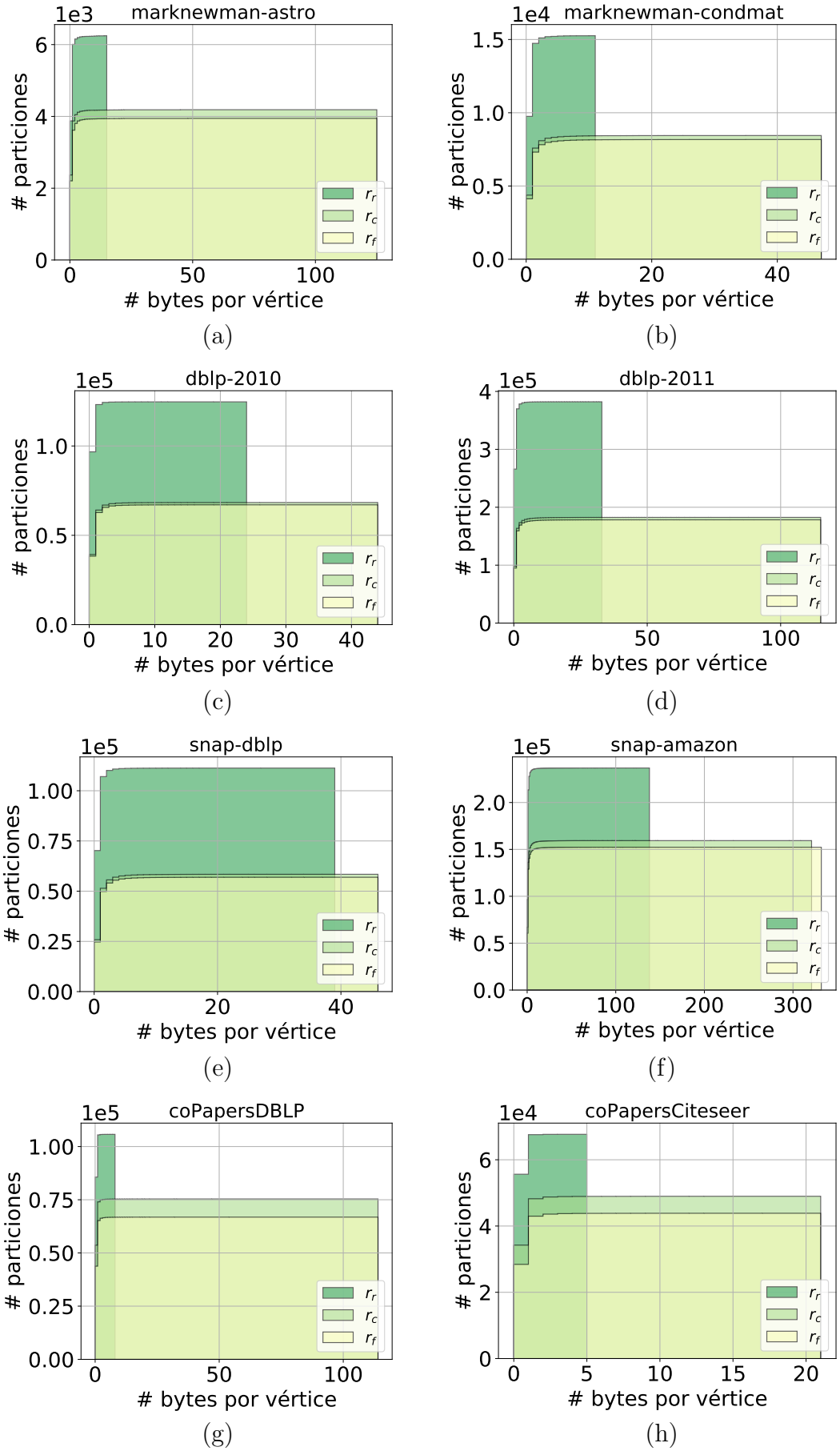


Figura 5.16: CDF para bytes por vértice en estructuras compactas para cada función de ranking.

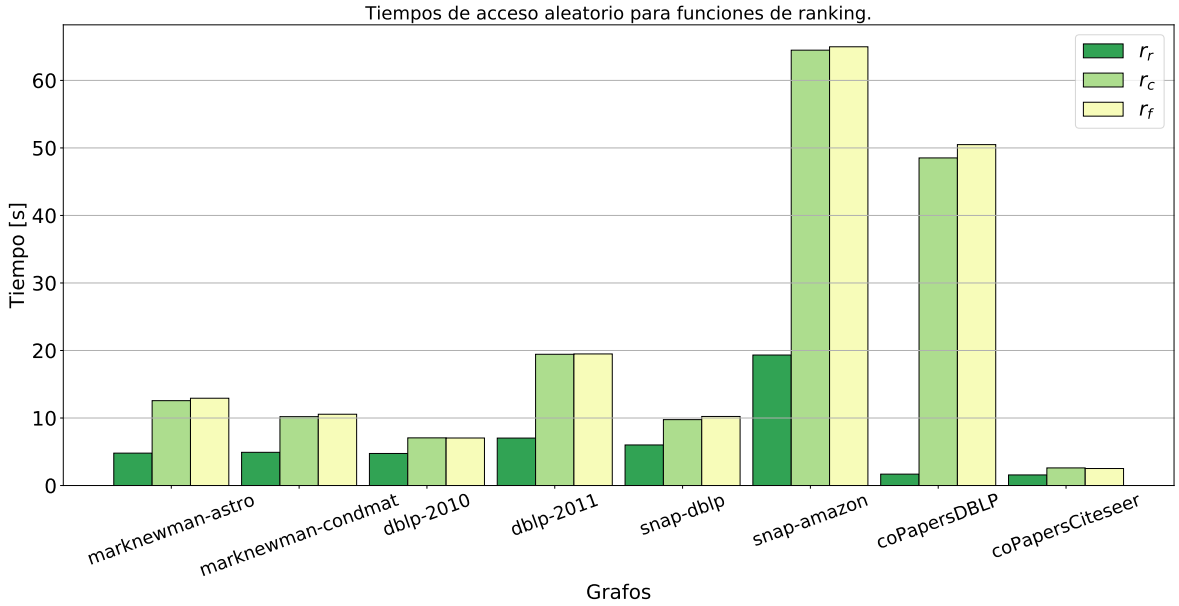


Figura 5.17: Tiempos de acceso aleatorio para las funciones de ranking.

$r_r(u)$.

En la Tabla 5.2 se muestran los tiempos en segundos de la generación del listado de cliques maximales $\mathcal{C}$ ($t_{\mathcal{C}}$) directo del grafo, el tiempo de generar la estructura compacta desde el listado de cliques (t_{CS}), el tiempo total de generar la estructura compacta ($t_T = t_{\mathcal{C}} + t_{CS}$) y el tiempo para recuperar el listado de cliques $\mathcal{C}$ desde la estructura compacta ($t'_{\mathcal{C}}$)) usando el Algoritmo 4.6.

Se debe notar que el tiempo para recuperar el listado de cliques desde la estructura compacta es menor al requerido desde el grafo directamente. Si bien se puede argumentar que para llegar a la estructura compacta se debe generar el listado desde el grafo, por tanto $t_{\mathcal{C}}$ es necesario pagarlo ineludiblemente, una vez generada la estructura se puede obtener $\mathcal{C}$ de ella, en menor tiempo y sin tener que descomprimir el grafo. Se destaca este contraste en los grafos `coPapersDBLP` y `coPapersCiteseer`, donde desde la estructura compacta es sobre 10 veces más rápida.

A continuación se procede a comparar la opción de compresión seleccionada con los algoritmos del estado del arte ya mencionados.

5.4. Comparando con estado del arte

En esta sección se compara el nivel de compresión y los tiempos de acceso de la estructura compacta usando la función de ranking $r_r(u)$ seleccionada en la sección anterior, con los algoritmos más recientes de WebGraph [2], Apostolico and Drovandi (AD) [1], y k2tree [9].

A continuación se detallan las notaciones a usar en el resto de la sección.

Tabla 5.2: Tiempos de obtención de listado de cliques maximales y construcción de la estructura compacta, en segundos.

Grafo	$t_{\mathcal{C}}$	t_{CS}	t_T	$t'_{\mathcal{C}}$
marknewman-astro	0,18	0,28	0,46	0,05
marknewman-condmat	0,28	0,40	0,68	0,11
dblp-2010	1,12	1,46	2,58	0,57
dblp-2011	5,58	7,30	12,88	2,77
snap-dblp	1,68	2,30	3,98	0,86
snap-amazon	5,93	8,44	14,37	3,01
coPapersDBLP	17,96	3,44	21,40	1,39
coPapersCiteseer	26,70	4,70	31,40	0,96

- Para la estructura compacta propuesta, basada en las superposición de cliques maximales, se anotará como $clique_{rr}$.
- En el caso de Webgraph, se diferencia el caso de acceso aleatorio (WG_a) de acceso secuencial (WG_s). Esto es necesario ya que para el acceso aleatorio el algoritmo genera una estructura adicional.
- Para BFS de Apostolico y Drovandi, la notación corresponderá a AD.
- En el caso de k2tree, se diferencia cuando el algoritmo usa el orden del grafo original ($k2tree$) del orden por BFS ($k2tree_{BFS}$).

Es importante recordar que los algoritmos Webgraph y AD están orientados a comprimir grafos dirigidos. Esto requiere que cada arco de los grafos no dirigidos a evaluar deben ser anotados en ambos sentidos antes de ser comprimidos.

Mención especial requiere k2-tree, donde la autora proporcionó una versión mejorada del algoritmo orientado específicamente a grafos no dirigidos, que reduce el espacio necesario para representar el grafo considerando la mitad de la matriz de adyacencia, junto con la capacidad de dicho modelo de entregar los listados de vecinos directos como reversos.

Con esto presente, en la Tabla 5.3 se comparan los BPE de todos los casos, resaltando los mejores resultados. Para dos de los grafos comprimidos, `marknewman-astro` y `coPapersDBLP`, la propuesta es la que obtiene el menor resultado, seguido muy de cerca por la versión de k2-tree con BFS. Con el grafo `coPapersCiteseer` se obtiene un resultado muy cercano al mejor caso, y en los demás, los dos mejores resultados los obtienen alguna de las dos versiones de k2-tree, seguido siempre del método propuesto. tanto Webgraph como AD no logran competir en compresión con los demás.

Esto confirma que la la propuesta es competitiva con el estado del arte, solo considerando el nivel de compresión. A continuación se evaluara el rendimiento en tiempos de acceso.

Para comparar el tiempo de acceso aleatorio, se prueba el Algoritmo 4.4 recuperando los vecinos de un millón de vértices aleatorios de $G(V, E)$, y se divide el tiempo que demora

Tabla 5.3: BPE de algoritmos de compresión.

Grafo	$clique_{rr}$	$k2tree$	$k2tree_{BFS}$	AD	WG_a	WG_s
marknewman-astro	**3,96**	4,89	4,34	5,67	8,10	7,30
marknewman-condmat	5,74	6,28	**5,60**	7,86	11,78	10,45
dblp-2010	5,84	**4,23**	4,30	6,71	8,67	6,91
dblp-2011	6,58	**5,48**	5,89	9,67	10,13	8,71
snap-dblp	6,89	5.88	**5.23**	8,14	11,80	10,17
snap-amazon	10,44	8,02	**6,38**	10,96	14,50	13,35
coPapersDBLP	**0,78**	1,67	0,94	1,81	2,71	2,48
coPapersCiteseer	0,52	1,21	**0,45**	0,85	1,79	1,63

Tabla 5.4: Tiempos de acceso aleatorio, en microsegundos por arco.

Grafo	$clique_{rr}$	$k2tree$	$k2tree_{BFS}$	AD	WG_a
marknewman-astro	2,67	2,58	1,33	1,79	**0,052**
marknewman-condmat	3,16	5,53	2,81	2,32	**0,063**
dblp-2010	3,70	5,55	4,84	2,15	**0,097**
dblp-2011	4,66	11,43	10,69	2,36	**0,114**
snap-dblp	4,07	10,35	6,93	2,30	**0,125**
snap-amazon	6,99	13,97	7,13	2,47	**0,087**
coPapersDBLP	1,51	1,89	1,16	0,73	**0,045**
coPapersCiteseer	1,30	0,95	0,50	0,45	**0.037**

dicha solicitud por la cantidad de aristas recuperadas. En la Tabla 5.4 se presentan los resultados, sin considerar el caso de Webgraph secuencial (WG_s), ya que solo se considera acceso aleatorio.

Como se puede apreciar, Webgraph presenta los menores tiempos de acceso entre todos los algoritmos, y AD lo sigue siempre en segundo lugar. Luego para los grafos `dblp-2010`, `dblp-2011`, `snap-dblp` y `snap-amazon`, el método propuesto logra mejores resultados que ambas versiones de k2-tree, con especial atención a `dblp-2011` donde logra ser el doble de rápido. Para el resto de los casos, el resultado es bastante competitivo entre esos métodos.

El tiempo de reconstrucción secuencial se midió usando el Algoritmo 4.3. En la Tabla 5.5 se presentan los resultados obtenidos para los algoritmos evaluados. En este caso, si bien el método propuesto es más lento que los demás, para casos como `marknweman-astro`, `marknewman-condmat`, `dblp-2010` y `snap-dblp`, la diferencia es menor a un segundo. Para el resto de los casos, los grafos tienen una cantidad considerable de arcos, `dblp-2011` y `snap-amazon` tienen la mayor cantidad de cliques, y `coPapersDBLP` con `coPapersCiteseer` tienen muchos cliques de hasta 100 vértices, y presentan una relación no proporcional entre cantidad de vértices y cantidad de cliques, como lo muestra la Tabla 5.1, lo que afecta bastante a la hora de recuperar dichos grafos.

Para evaluar mejor la competitividad, en la Figura **??**, Figura 5.19, Figura 5.20, y

Tabla 5.5: Tiempos de reconstrucción secuencial del grafo, en segundos.

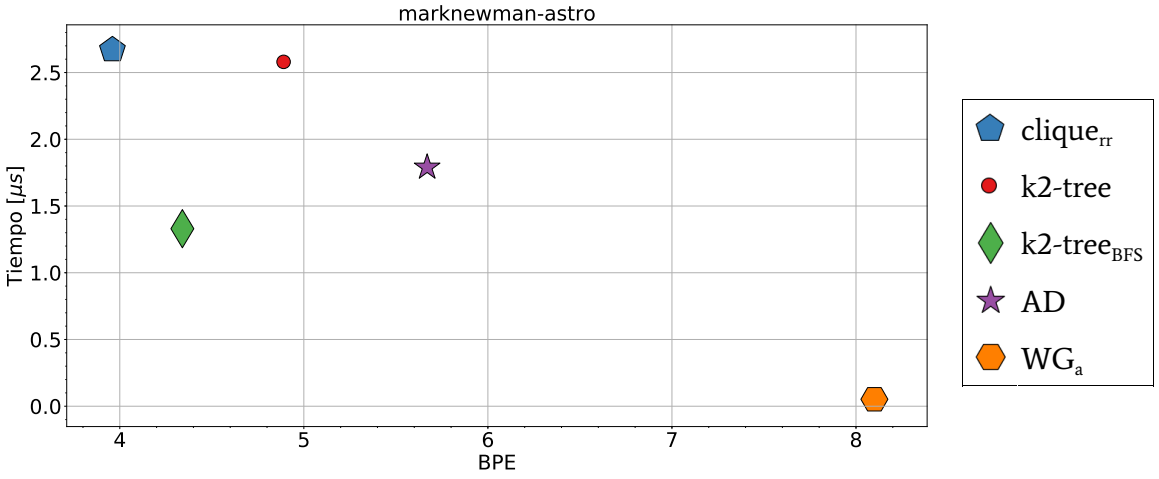
Grafo	$clique_{rr}$	$k2tree$	$k2tree_{BFS}$	WG_s
marknewman-astro	0,09	0,03	**0,02**	0,28
marknewman-condmat	0,16	0,07	**0,04**	0,52
dblp-2010	0,82	0,18	**0,16**	1,09
dblp-2011	4,45	**1,10**	1,31	2,41
snap-dblp	1,26	0,58	**0,35**	1,20
snap-amazon	4,53	1,36	**1,13**	1,30
coPapersDBLP	5,81	1,45	**1,01**	1,59
coPapersCiteseer	5,46	1,33	**0,65**	1,56

Figura 5.21 se grafican los BPE con respecto a los tiempo de acceso aleatorio en microsegundos, y en la Figura 5.22, Figura 5.23, Figura 5.24, y Figura 5.25 los BPE con respecto a los tiempos de reconstrucción secuencial en segundos, obtenidos para cada algoritmo y cada grafo.

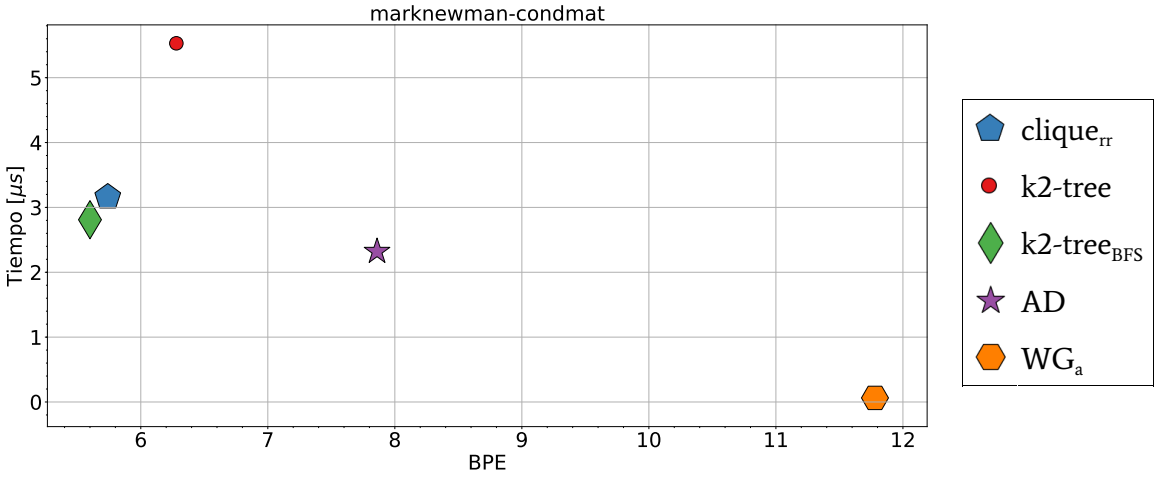
Se puede apreciar que, para el caso aleatorio, si bien el método de compresión siempre se ubica en el cuadrante de menor BPE y mayor tiempo, otros logran una ubicación de menor calidad, como los casos de los algoritmos de k2tree para los grafos `dblp-2010`, `dbpl-2011` y `snap-dblp` en la Figura 5.18(c)(d)(e) respectivamente.

En el caso secuencial, el método propuesto también se encuentra siempre en el mismo cuadrante, pero muy alejado en tiempo de los demás algoritmos, como ya se había estudiado de la Tabla 5.5, y tanto en el caso de `marknewman-astro` (Figura **??**(a)) como `marknewman-condmat` (Figura **??**(b)) presentan los mejores resultados.

En cuanto a detectar si dos nodos son vecinos o no, la estructura planteada es la única que tiene dicha consulta implementada y no requiere descomprimir para responder directamente. Tanto Webgraph como AD, al usar diferencias para codificar los listados de adyacencia, primero requieren obtener el listado de vecinos de un nodo y luego revisar si el segundo nodo pertenece o no a dicha lista. La propuesta de k2tree podría responder dicha consulta, ya que codifica la matriz de adyacencia de tal manera que podría revisar directamente la vecindad de dos nodos, pero no la tiene implementada.

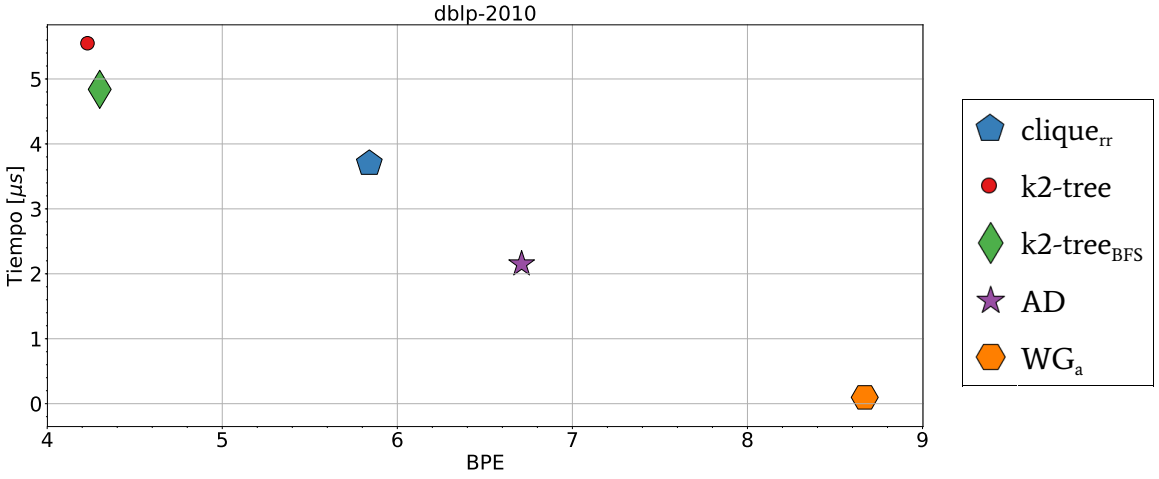


(a)

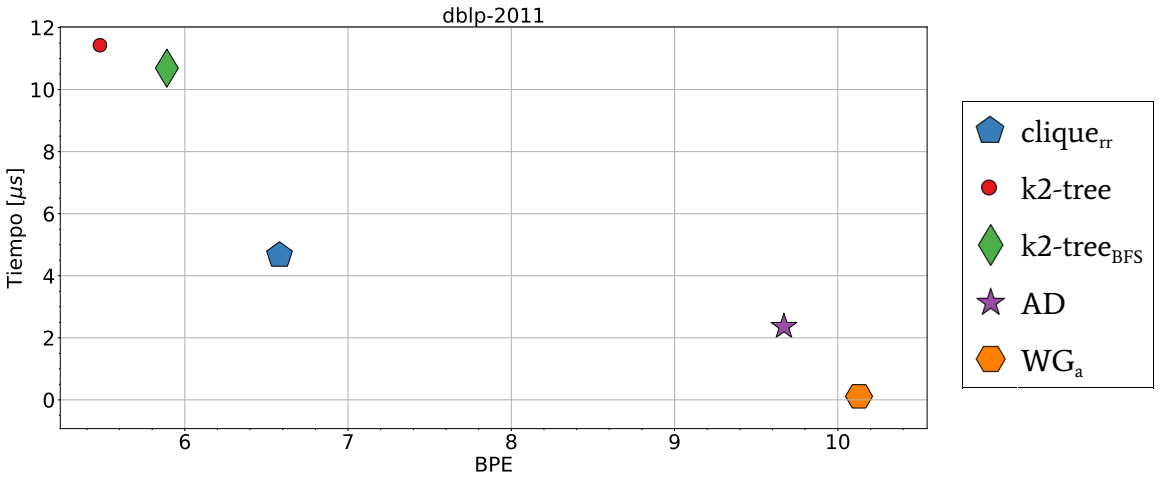


(b)

Figura 5.18: BPE y tiempo de acceso aleatorio en microsegundos de cada algoritmo, para los grafos marknewman-astro y marknewman-condmat.

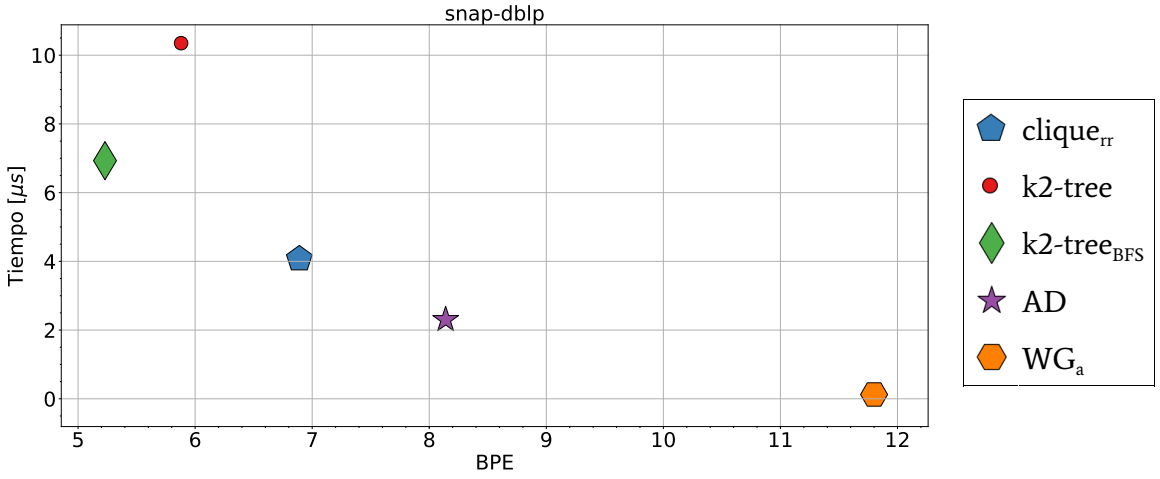


(a)

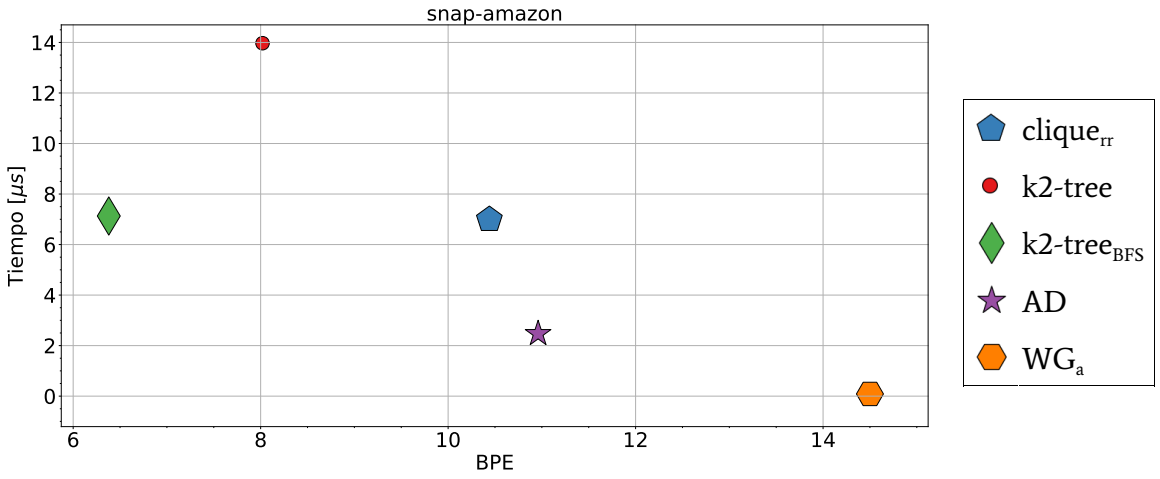


(b)

Figura 5.19: BPE y tiempo de acceso aleatorio en microsegundos de cada algoritmo, para los grafos dblp-2010 y dblp-2010.

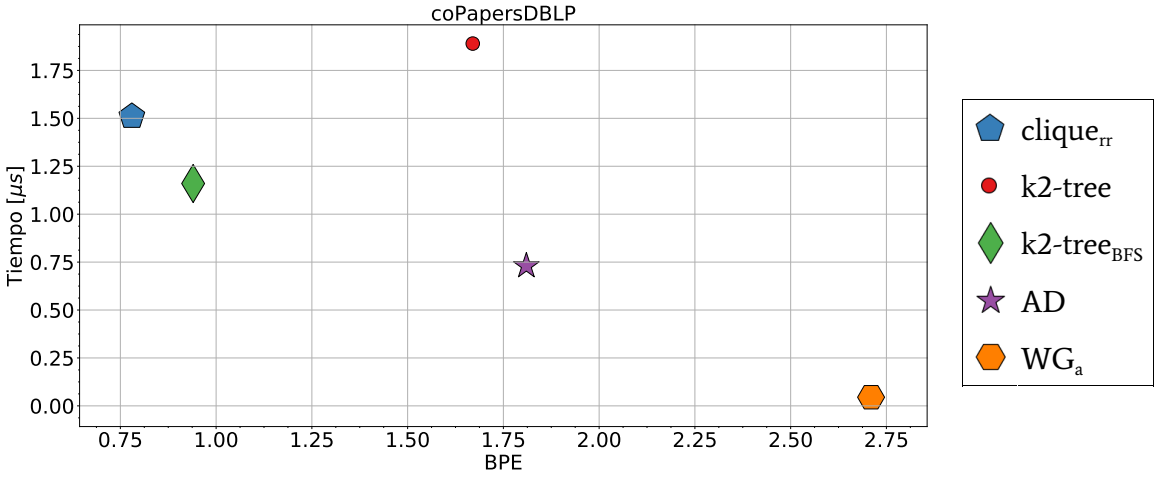


(a)

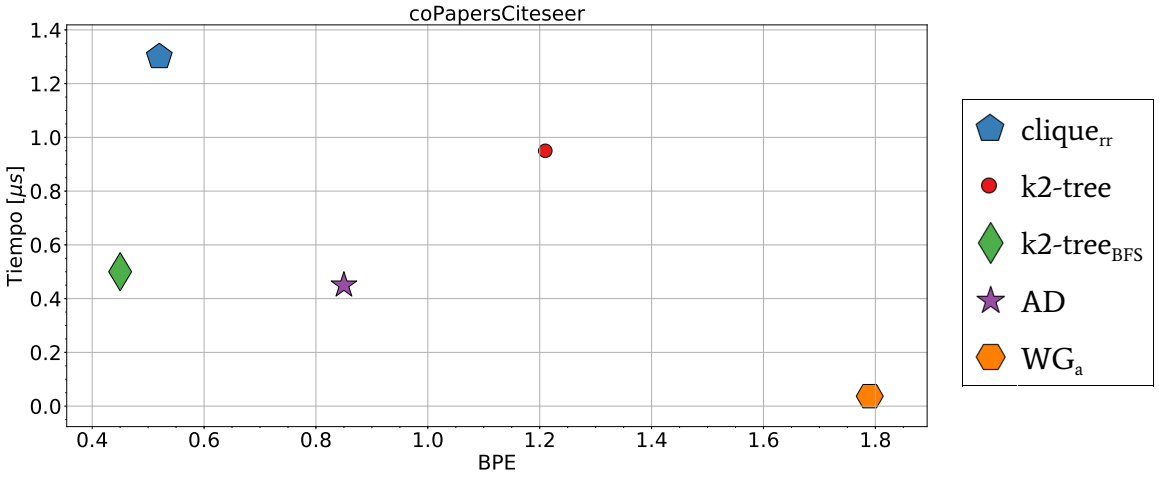


(b)

Figura 5.20: BPE y tiempo de acceso aleatorio en microsegundos de cada algoritmo, para los grafos snap-dblp y snap-amazon.

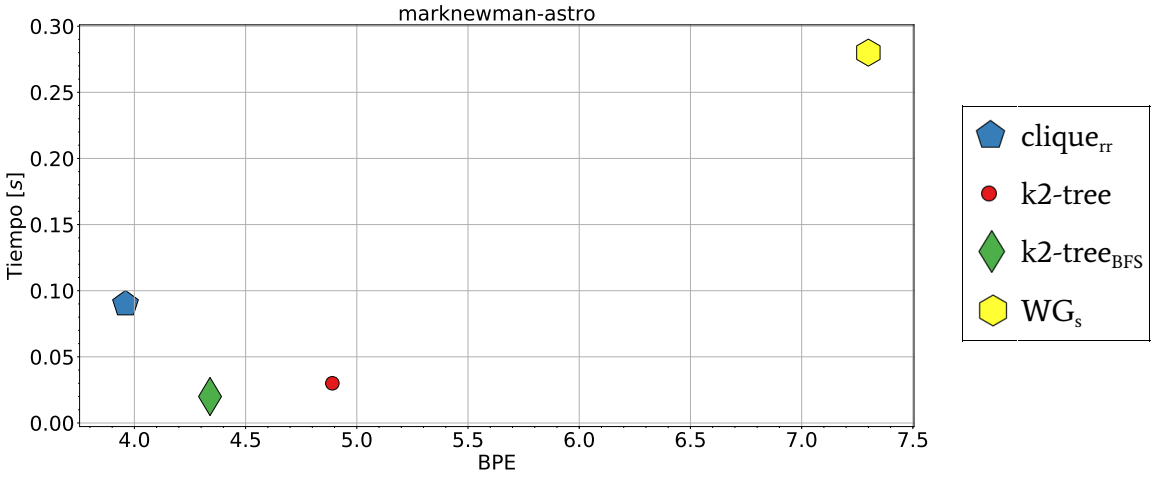


(a)

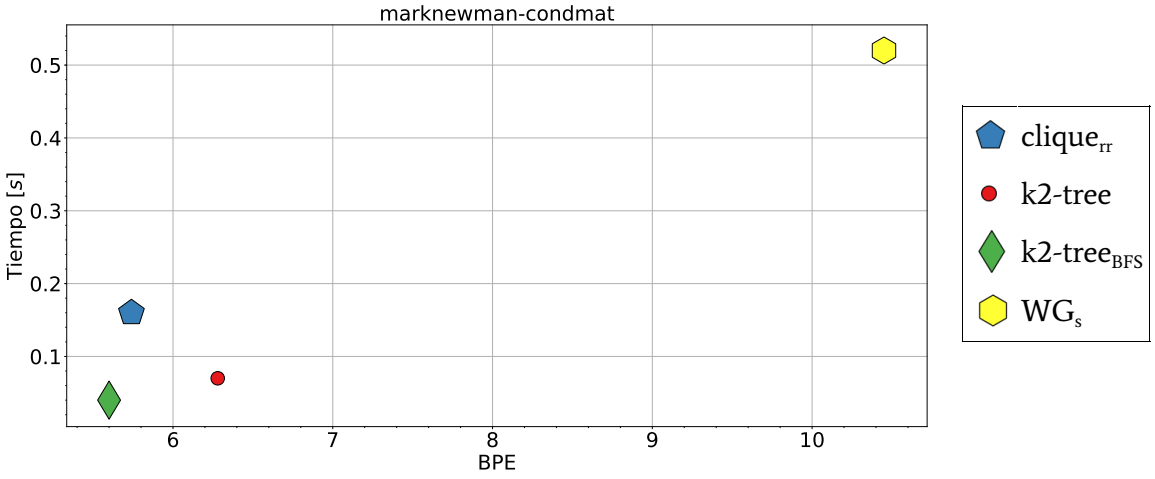


(b)

Figura 5.21: BPE y tiempo de acceso aleatorio en microsegundos de cada algoritmo, para los grafos coPapersDBLP y coPapersCiteseer.

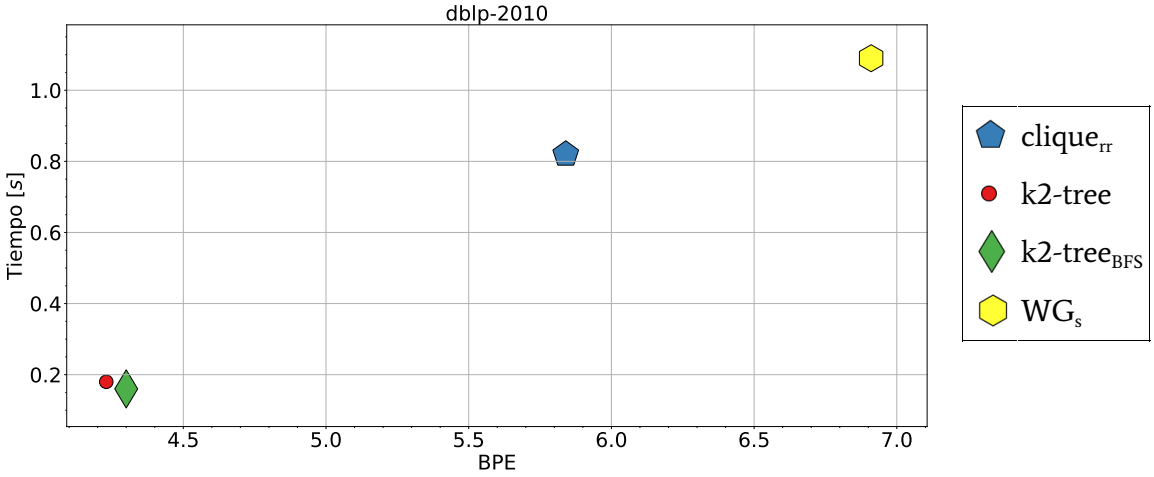


(a)

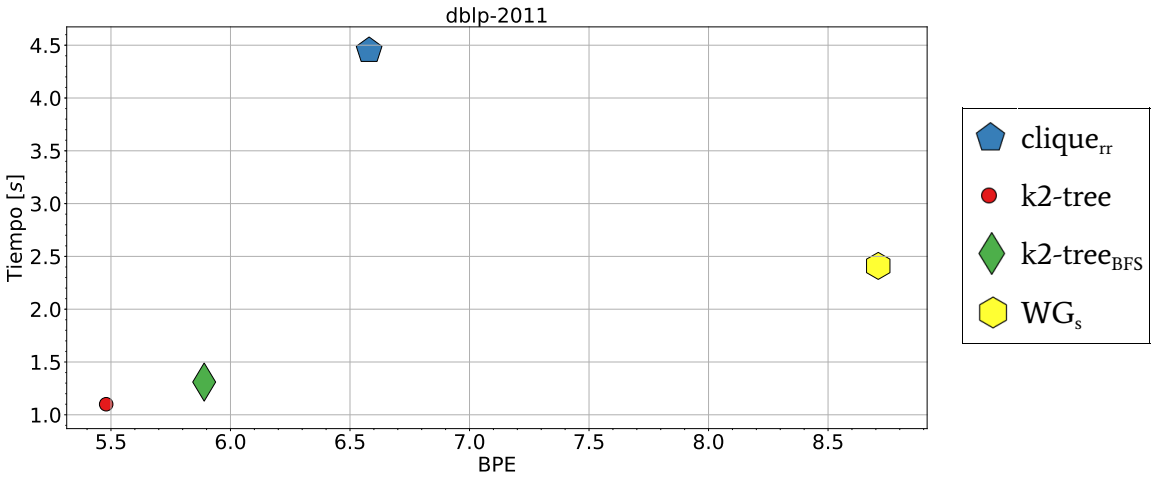


(b)

Figura 5.22: BPE y tiempo de reconstrucción secuencial en segundos de cada algoritmo, para los grafos marknewman-astro y marknewman-condmat.

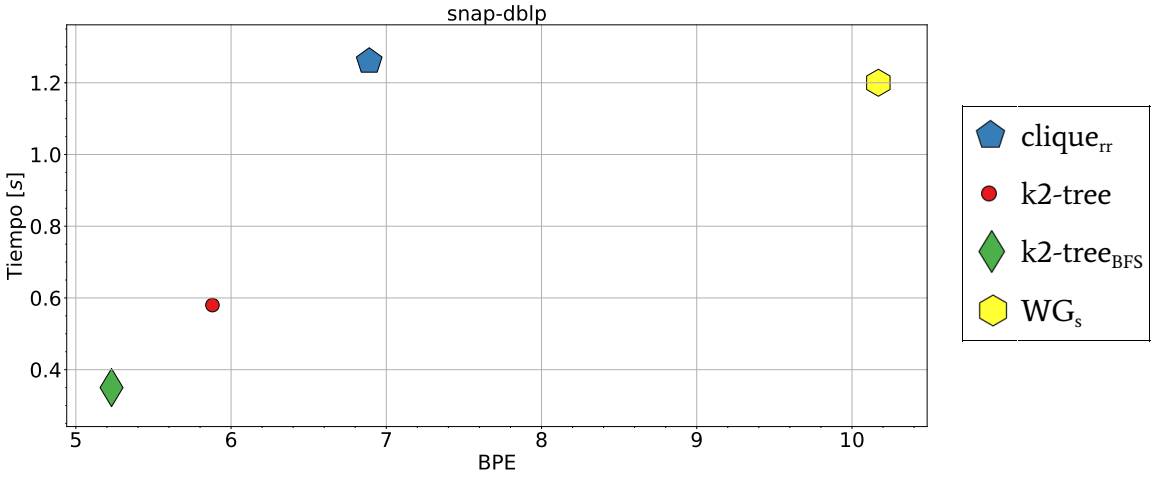


(a)

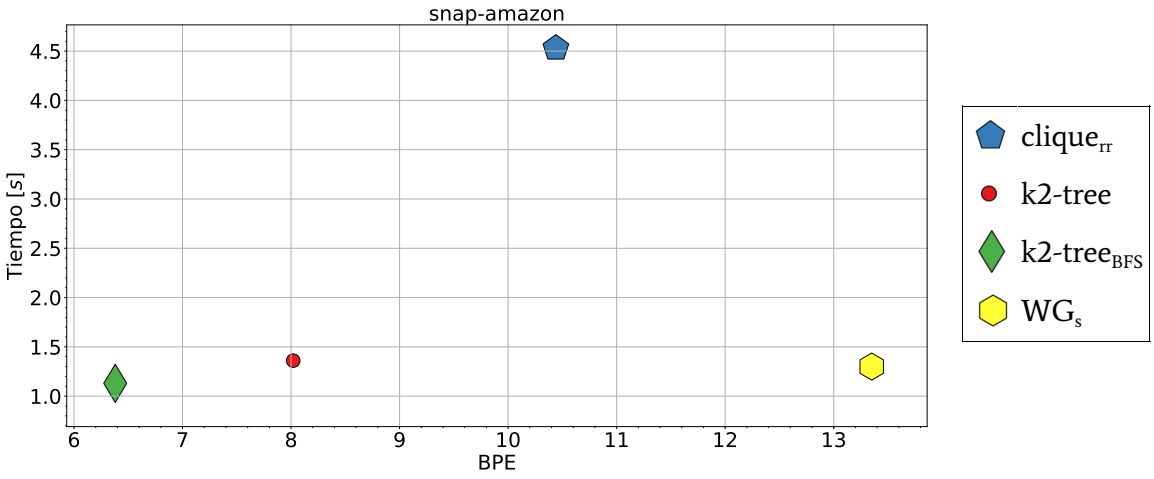


(b)

Figura 5.23: BPE y tiempo de reconstrucción secuencial en segundos de cada algoritmo, para los grafos dblp-2010 y dblp-2011.

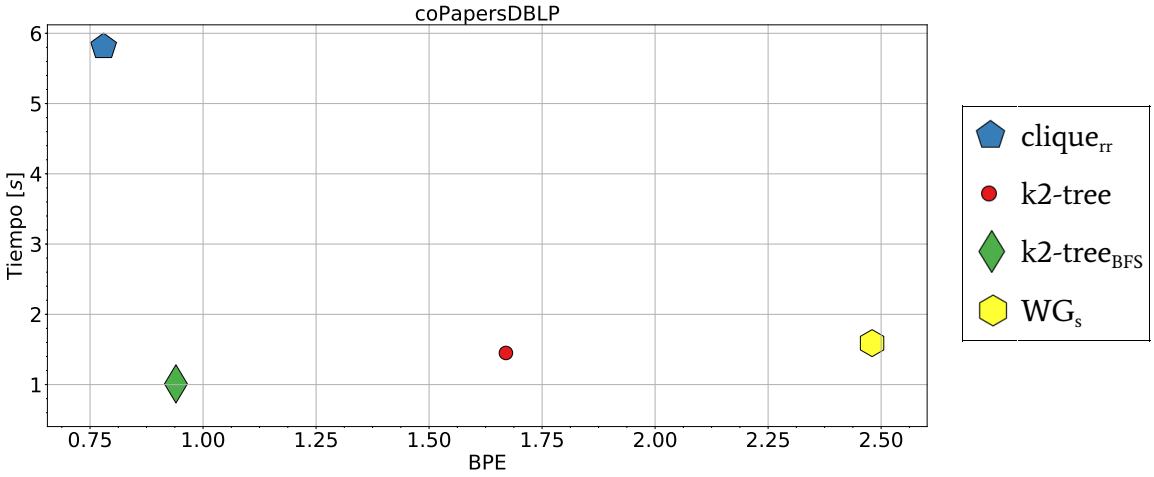


(a)

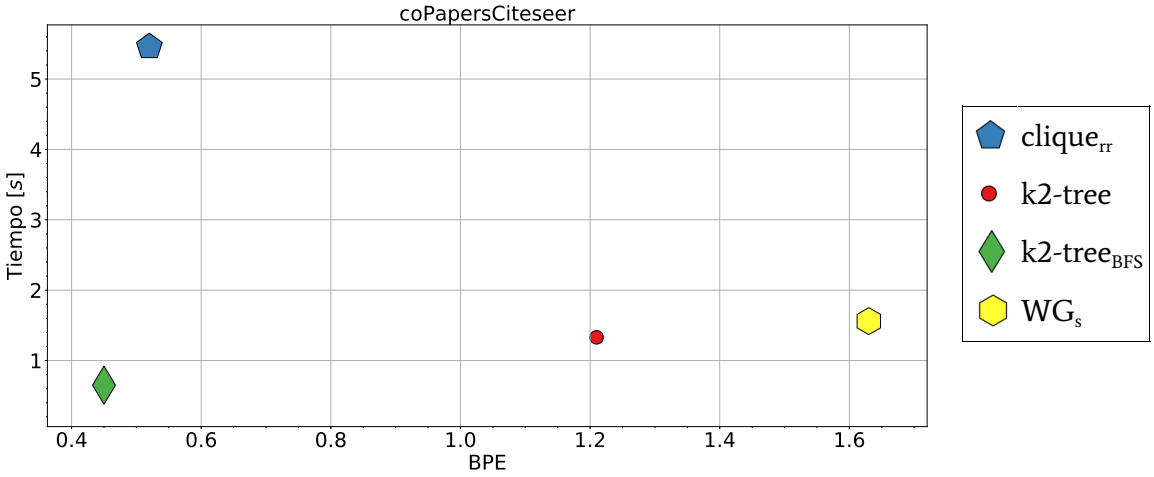


(b)

Figura 5.24: BPE y tiempo de reconstrucción secuencial en segundos de cada algoritmo, para los grafos snap-dblp y snap-amazon.



(a)



(b)

Figura 5.25: BPE y tiempo de reconstrucción secuencial en segundos de cada algoritmo, para los grafos coPapersDBLP y coPapersCiteseer.

Capítulo 6

CONCLUSIONES

Este trabajo se enfoca en desarrollar un método de compresión de grafos basado en clustering de cliques maximales, apuntado a grafos no dirigidos. Se logra llegar a una estructura compacta final que comprime un grafo y permite responder consultas sin tener que descomprimir para ello.

Entrando en detalle, el nivel de compresión logrado, medido en BPE, es mejor al estado del arte (ver Tabla 5.3), superando en todos los casos estudiados a los otros algoritmos. Incluso puntualmente para el caso de los grafos `coPapersDBLP` y `coPapersCiteseer`, los cuales poseen los coeficiente de clusterización (0,80 y 0,83) y transitividad (0,65 y 0,77) más altos (ver Tabla 5.1, se logran los BPE de 0,80 y 0,52 respectivamente, lo que es muy eficiente. Si bien este resultado en la compresión es muy positivo, es necesario reconocer el efecto que conlleva en los tiempos de acceso

El tiempo de acceso aleatorio, medido usando el Algoritmo 4.4 recuperando vecinos para un millón de nodos, en varios casos se logran mejores resultados que k2tree con orden del grafo original, pero no usando BFS, donde solo para el grafo `dblp-2010` logra un tiempo similar, en los demás es mayor. Más lejos aún comparando con AD o Webgraph, donde en el mejor de los casos los tiempos logrados con respecto a AD son del orden del doble, para `marknewman-condmat`, `dblp-2010` y `coPapersDBLP` (ver Tabla 5.4).

Para el tiempo de reconstrucción secuencial, medido usando el Algoritmo 4.3, solo para los grafos pequeños `marknewman-astro` y `marknewman-condmat` se obtienen resultados mejores con respecto a WebGraph, en ambos casos casi la mitad. Pero en general, tampoco se logra un tiempo mejor a los algoritmos en comparación, y para los grafos `coPapersDBLP` y `coPapersCiteseer` que presentan la mejor compresión, se obtienen tiempos mucho más altos (ver Tabla 5.5).

Con esto en consideración, una buena aplicación para el método propuesto son dispositivos donde el espacio para guardar el grafo sea limitado, como dispositivos móviles con poca RAM y espacio en disco, donde se pueden almacenar los grafos usando una compresión muy eficiente, y que permitirá responder consultas sin ocupar mucho espacio extra y en un tiempo algo mayor. Esto no es menor, ya que sin esta opción de compresión, y pese a su costo en tiempos de acceso, no se podría almacenar los grafos en dichos dispositivos de otra manera.

Además, esta estructura compacta permite obtener el listado de cliques maximales directamente de ella, sin tener que descomprimir el grafo completo, y pese a que se necesita generar este listado antes de su construcción, una vez comprimido permite listar los cliques de manera rápida (ver Tabla 5.2). Esto, junto con el nivel de compresión ya mencionado, sirve para trabajar con dispositivos de memoria acotada en variadas aplicaciones biológicas

[13, 21], entre otras [6].

Como trabajo futuro, se puede explorar cómo mejorar los tiempos de acceso de esta estructura, por ejemplo encontrar nuevas funciones de ranking en la heurística de clusterización para mejorar la relación entre compresión y tiempos de acceso. Otra opción es explotar el potencial de paralelismo que posee la estructura compacta, ya que cada partición se puede acceder de manera simultánea, y cada comparación de bytes dentro de las particiones se puede optimizar usando instrucciones paralelas, como SIMD[1].

[1]SIMD: Single Instruction, Multiple Data. Una Instrucción, múltiples datos.

Bibliografía

[1] Alberto Apostolico and Guido Drovandi. Graph compression by bfs. *Algorithms*, 2(3):1031–1044, 2009.

[2] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, pages 587–596. ACM, 2011.

[3] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Permuting web graphs. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 116–126. Springer, 2009.

[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. ACM, 2004.

[5] Paolo Boldi and Sebastiano Vigna. The webgraph framework ii: Codes for the world-wide web. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, page 528. IEEE, 2004.

[6] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999.

[7] Nieves R Brisaboa, Rodrigo Cánovas, Francisco Claude, Miguel A Martínez-Prieto, and Gonzalo Navarro. Compressed string dictionaries. In *International Symposium on Experimental Algorithms*, pages 136–147. Springer, 2011.

[8] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k 2-trees for compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 18–30. Springer, 2009.

[9] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.

[10] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, WSDM '08, pages 95–106, New York, NY, USA, 2008. ACM.

[11] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transactions on the Web (TWEB)*, 4(4):16, 2010.

[12] Francisco Claude, Gonzalo Navarro, and Alberto Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[13] John D Eblen, Charles A Phillips, Gary L Rogers, and Michael A Langston. The maximum clique enumeration problem: algorithms, applications, and implementations. In *BMC bioinformatics*, volume 13, page S5. BioMed Central, 2012.

[14] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *J. Exp. Algorithmics*, 18:3.1:3.1–3.1:3.21, November 2013.

[15] David Eppstein and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, pages 364–375, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] Otmar Ertl. Bagminhash - minwise hashing algorithm for weighted sets. *CoRR*, abs/1802.03914, 2018.

[17] Johannes Fischer and Daniel Peters. Glouds: Representing tree-like graphs. *Journal of Discrete Algorithms*, 36:39 – 49, 2016. WALCOM 2015.

[18] Alexandre Francisco, Travis Gagie, Susana Ladra, and Gonzalo Navarro. Exploiting computation-friendly graph compression methods for adjacency-matrix multiplication. In *2018 Data Compression Conference*, pages 307–314. IEEE, 2018.

[19] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[20] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[21] William Hendrix, Matthew C Schmidt, Paul Breimyer, and Nagiza F Samatova. Theoretical underpinnings for maximal clique enumeration on perturbed graphs. *Theoretical Computer Science*, 411(26-28):2520–2536, 2010.

[22] Cecilia Hernández and Gonzalo Navarro. Compressed representation of web and social networks via dense subgraphs. In *International Symposium on String Processing and Information Retrieval*, pages 264–276. Springer, 2012.

[23] Cecilia Hernández and Gonzalo Navarro. Compressed representations for web and social graphs. *Knowledge and Information Systems*, 40(2):279–313, Aug 2014.

[24] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[25] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[26] Fengying Li, Qi Zhang, Tianlong Gu, and Rongsheng Dong. Optimal representation for web and social network graphs based on k^2-tree. *IEEE Access*, 7:52945–52954, 2019.

[27] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005.

[28] Sebastian Maneth and Fabian Peternek. Compressing graphs by grammars. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 109–120. IEEE, 2016.

[29] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. *CoRR*, abs/cs/0610001, 2006.

[30] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[31] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

[32] Cecilia Hernández Rivas. Managing massive graphs. Universidad de Chile, 2014. Tesis de doctorado.