



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
Departamento de Ingeniería Informática y Ciencias de la Computación
Facultad de Ingeniería
Universidad de Concepción

©

Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica del documento.

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 fructífera.

Resumen

La compresión y utilización 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.

En este trabajo se pretende aportar con un método que aprovecha la superposición de cliques maximales en la generación de una estructura compacta, que permita manejar de manera eficiente y efectiva dichos grafos, ahorrando tanto en espacio como en tiempos de acceso, y entregar una herramienta que facilite dicho uso en dispositivos con memoria acotada.

Tabla de Contenido

Resumen	v
Índice de Tablas	vii
Índice de Figuras	viii
Capítulo 1 INTRODUCCIÓN	1
Capítulo 2 ESTADO DEL ARTE	2
Capítulo 3 MARCO TEÓRICO	5
3.1 Notación y definiciones	5
3.1.1 Grafos y cliques maximales	5
Capítulo 4 MÉTODO DE COMPRESIÓN PROPUESTO	8
4.1 Detección de cliques maximales	8
4.2 Particionamiento del grafo de cliques	9
4.3 Algoritmo de particionamiento o clustering	9
4.4 Representación en estructuras compactas	11
4.4.1 Secuencias de la representación de las particiones	12
4.4.2 Algoritmos de consulta	13
Capítulo 5 RESULTADOS	17
5.1 Grafos y estructura compacta	17
5.2 Comparación de funciones de ranking	21
5.3 Comparando con estado del arte	30
Capítulo 6 CONCLUSIONES	35
Bibliografía	37

Índice de Tablas

5.1	Cantidad de vértices, aristas, cliques, grado medio y máximo de los vértices, coeficiente de clusterización y transitividad de los grafos a comprimir.	20
5.2	Espacio en bits de las estructuras de datos sucintos para la función $r_f(u)$	21
5.3	Espacio en bits de las estructuras de datos sucintos para la función $r_c(u)$	21
5.4	Espacio en bits de las estructuras de datos sucintos para la función $r_r(u)$	25
5.5	Tiempos de obtención de listado de cliques maximales y construcción de la estructura compacta, en segundos.	29
5.6	BPE de algoritmos de compresión.	30
5.7	Tiempos de acceso aleatorio, en microsegundos por arco.	31
5.8	Tiempos de reconstrucción secuencial del grafo, en segundos.	34

Índice de Figuras

3.1	Ejemplo de grafo y sus cliques maximales.	6
4.1	(a) Grafo no dirigido. (b) Lista de cliques maximales. (c) Grafo de cliques.	9
4.2	Resultados de las funciones de ranking. (a) Puntaje final. (b) Particiones de cliques.	11
4.3	Estructura compacta para particiones \mathcal{CP}_{rr}	13
5.1	Distribución del grado de los vértices para cada grafo.	18
5.2	Distribución del tamaño de los cliques maximales para cada grafo.	19
5.3	BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking para cada grafo.	22
5.4	BPE y Tiempo de reconstrucción secuencial medio para posibles estructuras compactas, por cada función de ranking para cada grafo.	23
5.5	BPE de las estructuras compactas para las funciones de ranking.	24
5.6	Número de particiones en las estructuras compactas para las funciones de ranking.	24
5.7	(a) Proporción de bits por cada secuencia en la estructura compacta, para cada función de ranking. (b) Proporción normalizada.	26
5.8	Número máximo de vértices en la secuencia X para las funciones de ranking.	27
5.9	Número máximo de bytes por nodo para las funciones de ranking.	27
5.10	CDF para bytes por vértice en estructuras compactas para cada función de ranking.	28
5.11	Tiempos de acceso aleatorio para las funciones de ranking.	29
5.12	BPE y tiempo de acceso aleatorio en microsegundos de cada algoritmo, para cada grafo.	32
5.13	BPE y tiempo de reconstrucción secuencial en segundos de cada algoritmo, para cada grafo.	33

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 millones¹, 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

ESTADO DEL ARTE

El problema de compresión de grafos ha sido abordado de distintas maneras en las últimas décadas.

Uno de los primeros trabajos en la materia es *WebGraph* de Boldi y Vigna [4, 5] el año 2003, donde comprimen el grafo de la web aprovechando dos características de sus enlaces, **localidad** (hipervínculos donde sus *url* tienen un prefijo en común y si se ordenan lexicográficamente en una lista estarán muy cerca entre ellos) y **similaridad** (los sitios que tienden a estar juntos en esa lista lexicográfica también tienden a tener muchos sucesores en común). Gracias a esto se puede aprovechar las pequeñas brechas entre enlaces en la lista de adyacencia para codificarlos y así comprimir el grafo.

El 2009 Boldi, Santini y Vigna [3] usaron la matriz de adyacencia, y basados en aplicar permutaciones a sus filas, se logran reordenar y generar una nueva matriz donde las filas, si son similares (contienen unos en posiciones muy comunes), deben estar cerca (ser consecutivas o en una vecindad acotada). El 2011 los mismos autores et al. 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 red social, donde los vértices no pueden ser ordenados de manera lexicográfica. Usando la matriz de adyacencia, junto con **descomponer** en tareas el reordenamiento de la matriz y aprovechar los procesadores multi-core, logran muy buenos resultados.

La mayoría de las propuestas de compresión de grafos proporcionan consultas para accesar los vecinos directos de un vértice, es decir, la lista de vértices a los que apunta un vértice en el grafo. Sin embargo, existen propuestas para accesar los vecinos directos y reversos como lo son k2-tree propuesta por Brisaboa, Ladra y Navarro [8], y la propuesta por Hernández y Navarro [22]. Una consulta a vecinos reversos corresponde a obtener la lista de vecinos que apuntan a un determinado vértice. Soportar varios tipos de consultas es relevante para varios tipos de aplicaciones de grafo, sobre todo considerando que la compresión puede ayudar a hacer **mas** eficiente la implementación de algoritmos de grafos más complejos, a través de mejor uso de la memoria en el procesamiento. 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 [29].

La compresión propuesta por Brisboa et al.[8] propone una representación que explota el hecho que las matrices de adyacencia de muchos grafos son esparsas, es decir, contienen muchos ceros. Junto con esta característica, este artículo explota

además el hecho que existen grupos de ceros juntos. Para ello utiliza una idea basada en el particionamiento del quadtree, estructura de datos de árbol que consiste en subdividir el espacio en cuatro hijos en forma recursiva. Usando esto, representa la matriz de adyacencia con un árbol con k^2 hijos 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 , y cada submatriz se subdivide recursivamente hasta que una submatriz sólo contenga ceros, la cual se representa en el árbol con una hoja en cero, o se representa un nodo interno con un uno y continúa la recursión hasta llegar a los elementos que contengan unos en la matriz como nodos hojas en el árbol. Esta representación comprimida usa un bitmap T para representar la estructura del árbol, y un bitmap L para representar las hojas, además de un bitmap adicional para acelerar la resolución de consultas. La estructura del k2tree se puede mejorar usando definición de orden en los nodos usando recorrido BFS en el grafo [9]. Una propuesta muy reciente, por Li, Zhang et. al.[25] mejora considerablemente k2-tree proponiendo una manera distinta de subdividir la matriz de adyacencia.

La compresión de Hernandez y Navarro [22] propone una estructura compuesta por un componente que contiene los subgrafos bipartitos completos, y el resto del grafo usando otra representación comprimida existente como k2tree. Los subgrafos bipartitos completos son encontrados en el grafo original y son representados usando estructuras sucintas como wavelet trees y bitmaps comprimidos [19]. Esta representación comprimida proporciona resolución de consulta de vecinos directos y reversos con tiempos de acceso similares.

Una propuesta de compresión más reciente utiliza gramáticas [27]. Esta propuesta generaliza Re-Pair [24], que consiste en iterativamente reemplazar los *digrams* (un par de letras consecutivas en un string) por un nuevo símbolo hasta que no se pueden seguir reemplazando. La idea de usar gramáticas también ha sido explotada en otros enfoques. Claude y Navarro [11] proponen buscar los pares de arcos vecinos más frecuentes y reemplazarlos por nuevos símbolos en forma iterativa hasta que se cumpla un umbral, representando el grafo y el diccionario de símbolos en forma comprimida. En esta propuesta, un *digram* consiste en un par de hipervínculos. Para lograr esta representación, los autores deben encontrar digrams con un número máximo de ocurrencias que no se sobrepongan (que no tengan arcos en común). Encontrar estas ocurrencias es un problema conocido como *maximum matching problem* y es caro computacionalmente ($O(|V|^2 \times |E|)$). Independiente de eso y al igual que k2tree, la estructura que proponen mejora con los algoritmos para ordenar los nodos en el grafo, y consiguen mejores resultados en grafos RDF con orden BFS.

El trabajo de Fisher y Peters [17] propone un esquema de compresión que consiste en representar el grafo como un árbol donde se representan los vértices repetidos como nodos sombra (shadow nodes). La compresión efectiva de esta representación consiste en representar el árbol usando bitmaps comprimidos y los nodos sombra se representan con una secuencia de símbolos usando wavelet trees (usando la biblioteca *SDSL* [20]). Sin embargo, esta representación ve limitado su nivel de compresión cuando los grafos a comprimir contienen muchos componentes densos, dado

que en este caso la secuencia de nodos sombra puede crecer mucho.

Entre las mejores propuestas de compresión de grafos que especialmente ofrecen accesos a vecinos directos, se encuentran la de Boldi, Rosa, Santini y Vigna [2] y la de Grabowski y Bieniecki [27]. Grabowski y Bieniecki usa bloques contiguos de listas de vecinos y consigue muy buen uso de disco, pero aumenta el tiempo de acceso a vecinos directos a medida que aumenta el tamaño del bloque. Desde un punto de vista de tiempo de acceso a vecinos directos, la propuesta de Boldi y Vigna sigue siendo una de las mas rápidas. La propuesta de Hernández y Navarro [31], 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 aplican 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]).

Por otro lado, en el contexto de clustering de grafos masivos, se ha usado como heurística el método de minhashing. Por ejemplo, para encontrar subgrafos bipartitos completos y agrupar listas de **adyacencia** con similitud de jaccard (Buerher y Chellapilla [10], Hernández [22]). Ambos trabajos aplican minhashing a la listas de adyacencia para crear una matriz de $n \times k$ valores de hash. Cada valor de hash en una fila de la matriz corresponde a un mapeo de la lista de adyacencia. Luego, para cada fila de la lista de adyacencia (vecinos directos de un vértice en el grafo) se aplica minhashing k veces. La implementación de Buerher y Chellapilla usa $k = 8$ y luego encuentra clusters buscando listas de valores hash que comparten columnas para encontrar clusters. La implementación de Hernández [22] sólo usa $k = 2$ para comprimir grafos. El reciente trabajo de Ertl [16] propone un algoritmo que permite expandir las posibilidades de uso a sets con peso.

Un detalle no menor en este trabajo, es la necesidad de generar el listado de cliques maximales de un grafo. Esto ha sido abordado en varios trabajos, tanto por su complejidad como por su utilidad[13, 21, 6], y de las propuestas más recientes se destaca el trabajo de Eppstein, Strash et. al. [15, 14], dirigido a encontrar dicho listado para grafos con matrices de adyacencia ralas.

Capítulo 3

MARCO TEÓRICO

3.1 Notación y definiciones

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

3.1.1 Grafos y cliques maximales

Se define un **grafo** $G = (V, E)$ como el conjunto finito de **vértices** V (nodos) 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* de v_2 . Este trabajo está enfocado a utilizar grafos no dirigidos.

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.

Un **clique** es un subgrafo donde todos los vértices son adyacentes entre sí, es decir, $\exists V' \subseteq V(G), \forall u, v \in V', (u, v) \in E(G)$ puede extenderse incluyendo otro vértice adyacente, es decir, no es subconjunto de otro clique más grande. Un **grafo denso** es aquel que su número de aristas es cercano al máximo. En la Figura 3.1 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,

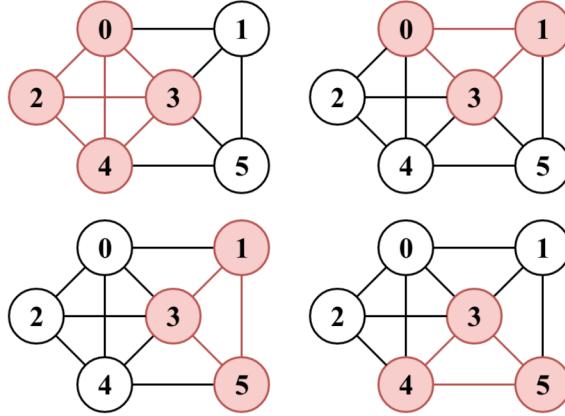


Figura 3.1: Ejemplo de grafo y sus cliques maximales.

$$\lambda(G) = \frac{1}{3} \sum_{v \in V} \lambda(v) \quad (3.1)$$

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.

$$\tau(G) = \sum_{v \in V} \tau(v) \quad (3.2)$$

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 es el promedio del coeficiente de todos los nodos del grafo, y se define como :

$$C(G) = \frac{1}{|V'|} \sum_{v \in V'} c(v) \quad (3.3)$$

$$V' = \{v \in V | d(v) \geq 2\}$$

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 es la probabilidad que un par de nodos adyacentes estén interconectados, y se define como :

$$T(G) = \frac{3\lambda(G)}{\tau(G)} \quad (3.4)$$

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.

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, Strash 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 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, Strash et. al. [15, 14] proponen un algoritmo rápido para listar cliques maximales de grafos dispersos.

En este trabajo, se asume que este algoritmo puede entregar el listado de cliques maximales de un grafo. Luego, el problema a resolver se concentra en encontrar un método eficiente para particionar 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. Grafo de cliques

Dado un grafo $G = (V, E)$ y $\mathcal{C} = \{c_1, c_2, \dots, c_N\}$ 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. $\forall c, c' \in \mathcal{C}, (c, c') \in E_{\mathcal{C}} \iff c \cap c' \neq \emptyset$

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.

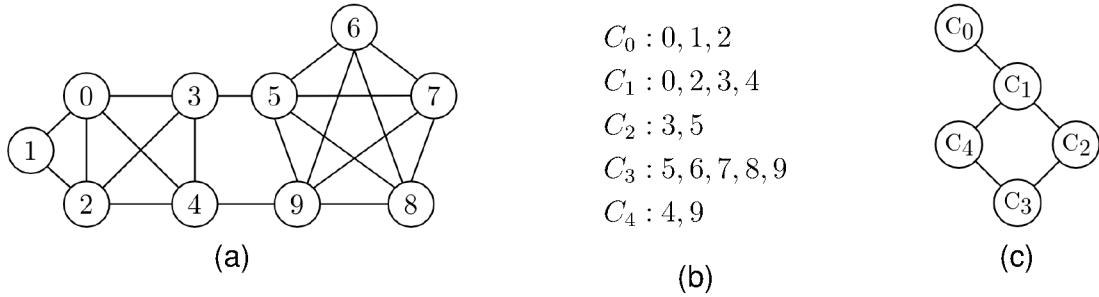


Figura 4.1: (a) Grafo no dirigido. (b) Lista de cliques maximales. (c) Grafo de cliques.

4.2 Particionamiento del grafo de cliques

Teniendo el listado de los cliques maximales, es necesario definir una heurística que permita agruparlos en particiones de manera eficiente, pensando tanto en espacio en disco como en tiempos de acceso.

Se desea encontrar un particionamiento del grafo de cliques que explote 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_C .

Dado un grafo de cliques $CG_C = (V_C, E_C)$, encontrar un set de particiones de cliques $\mathcal{CP} = \{cp_1, cp_2, \dots, cp_M\}$ de $CG_C(V_C, E_C)$ con $M \geq 1$, tal que

$$1. \bigcup_{i=1}^M cp_i = CG_i$$

$$2. cp_i \cap cp_j = \emptyset \text{ para } i \neq j$$

$$3. \text{cualquier } cp_i \in \mathcal{CP} \text{ es un subgrafo de } CG_C(V_C, E_C) \text{ inducido por el subset de vértices en } cp_i$$

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** $\mathcal{C} = \{c_1, c_2, \dots, c_N\}$ 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 $\mathcal{C} = \{c_1, c_2, \dots, c_N\}$ el conjunto de tamaño N de cliques maximales que cubren G , una función de ranking es una función $r : V \rightarrow \mathbb{R}_{>0}$ 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.

Algorithm 4.1 Algoritmo de particionamiento del grafo de cliques.

Require: Listado \mathcal{C} de cliques maximales ($N = |\mathcal{C}|$), función de ranking $r(u)$

Ensure: Colección de particiones \mathcal{CP}

```

1:  $(D, R) \leftarrow computeRanking(r, \mathcal{C})$  (Arreglos D y R,  $\forall u \in V$ )
2: Crear arreglo de bits  $Z$  de tamaño  $N$  e iniciar cada bit en 0
3: for ( $u \in R$ ) do
4:    $cpid \leftarrow \emptyset$ 
5:   for ( $id \in D[u]$  and  $D[u] = 0$ ) do
6:      $Z[id] \leftarrow 1$ 
7:      $cpid \leftarrow cpid \cup \{id\}$ 
8:   end for
9:   if ( $cpid \neq \emptyset$ ) then
10:     $\mathcal{CP} \leftarrow \mathcal{CP} : cpid$ 
11:   end if
12: end for
13: return  $\mathcal{CP}$ 

```

$u \in G$	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.

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.

$$r_f(u) = |C(u)| \quad (4.1)$$

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

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

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 , 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** lista los vértices presentes en los cliques de cada partición.
- El mapa de bits **B** indica dónde comienza una nueva partición, con un bit por cada elemento en **X** donde un 1 indica el inicio de la partición, más un bit extra al final en 1 para indicar el 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** contiene 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 la BB_p correspondiente, manteniendo el valor anterior de Y_p como lo indica la ecuación 4.7 con $bpu_p = 0$.

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

Dado $\mathcal{CP} = \{cp_0, \dots, cp_{M-1}\}$, $cp_p \in \mathcal{CP}$, y $cp_p = \{c_0, \dots, c_{m_r-1}\}$. 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:

$$X_p = \{u \in c | c \in cp_p\} = \{u_0, \dots, u_{|X_p|-1}\} \quad (4.4)$$

$$B_p = 1 : 0^{|X_p|-1} \quad (4.5)$$

$$BB_p = bb[|X_p|][bpu_p] \quad (4.6)$$

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

$$Y_p = |X_p| \times bpu_p + Y_{p-1}, \quad Y_0 = 0 \quad (4.7)$$

En la Figura 4.3 se presenta la estructura final resultante del ejemplo, usando las particiones \mathcal{CP}_{rr} . Como se puede apreciar, solo la segunda partición tiene dos cliques,

\mathcal{CP}_{rr}

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

Figura 4.3: Estructura compacta para particiones \mathcal{CP}_{rr} .

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 cero (un solo clique en partición 1), luego se agrega 5 por la cantidad de vértices presentes en la segunda partición, y como las siguientes particiones también solo tienen un clique, se repite el mismo valor para ellas.

4.4.2 Algoritmos de consulta

A continuación se presentan los algoritmos de consulta que soporta la estructura compacta. El Algoritmo 4.2 reconstruye el grafo $G(V, E)$ recorriendo secuencialmente la estructura compacta. El Algoritmo 4.3 recupera el listado de vecinos para un vértice cualquiera u del grafo $G(V, E)$. El **Algoritmo 4.1** recupera el listado de cliques maximales \mathcal{C} del grafo $G(V, E)$.

4.4?

El algoritmo secuencial (Algoritmo 4.2) 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.

El algoritmo para encontrar vecinos de vértices aleatorios (Algoritmo 4.3) 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.

El algoritmo para recuperar el listado de cliques maximales (Algoritmo 4.4) también recorre la estructura compacta de manera secuencial, y va recreando los cliques **representados** por los bytes en la secuencia BB de cada partición.

Algorithm 4.2 Algoritmo secuencial para reconstruir $G(V, E)$.

Require: X, B, BB, Y

Ensure: Retorna $G(V, E)$

```

1: Inicializa grafo vacío  $G$ 
2: for ( $i = 1$  to  $rank_B(1, |B|)$ ) do
3:    $bpu_p \leftarrow \frac{Y_p[i] - Y_p[i-1]}{select_B(1, i+1) - select_B(1, i)}$ 
4:   for ( $j = 0$  to  $|X_p|$  and  $bpu_p = 0$ ) do
5:     for ( $k = j + 1$  to  $|X_p|$  and  $u \neq X_p[k]$ ) do
6:       Insert edges  $(X_p[j], X_p[k])$  and  $(X_p[k], X_p[j])$  to  $G$ 
7:     end for
8:   end for
9:   for ( $j = 0$  to  $|X_p|$  and  $bpu_p \neq 0$ ) do
10:    for ( $k = j + 1$  to  $|X_p|$ ) do
11:      for ( $b = 0$  to  $bpu_p$ ) do
12:        if ( $BB_p[bpu_p * j + b]$  and  $BB_p[bpu_p * k + b]$ ) then
13:          Insert edges  $(X_p[j], X_p[k])$  and  $(X_p[k], X_p[j])$  to  $G$ 
14:          break
15:        end if
16:      end for
17:    end for
18:  end for
19: end for
20: return  $G$ 
  
```

Algorithm 4.3 Algoritmo para recuperar vecinos de un vértice $u \in G$.

Require: u, X, B, BB, Y

Ensure: Retorna $N(u)$

```

1: Initialize empty graph  $N(u)$ 
2: for ( $i = 1$  to  $rank_X(u, |X|)$ ) do
3:    $u_p \leftarrow select_X(u, i)$ 
4:    $p \leftarrow rank_B(1, u_p + 1) - 1$ 
5:    $bpu_p \leftarrow \frac{Y_p[p+1] - Y_p[p]}{select_B(1, p+2) - select_B(1, p+1)}$ 
6:   for ( $j = 0$  to  $|X_p|$  and  $bpu_p = 0$ ) do
7:     Insert  $X_p[j] \neq u$  to  $N(u)$ 
8:   end for
9:   for ( $j = 0$  to  $|X_p|$  and  $bpu_p \neq 0$ ) do
10:    for ( $b = 0$  to  $bpu_p$ ) do
11:      if ( $BB_p[bpu_p * j + b]$  and  $BB_p[bpu_p * u_p + b]$ ) then
12:        Insert  $X_p[j] \neq u$  to  $N(u)$ 
13:        break
14:      end if
15:    end for
16:   end for
17: end for
18: return  $N(u)$ 

```

Algorithm 4.4 Algoritmo para recuperar listado de cliques \mathcal{C} de $G(V, E)$.

Require: X, B, BB, Y

Ensure: Retorna listado de cliques maximales \mathcal{C}

```

1:  $\mathcal{C} \leftarrow \emptyset$ 
2: for ( $i = 1$  to  $rank_B(1, |B|)$ ) do
3:    $s \leftarrow select_B(1, i)$ 
4:    $e \leftarrow select_B(1, i + 1)$ 
5:    $bpu_p \leftarrow \frac{Y_p[i] - Y_p[i-1]}{e-s}$ 
6:   if  $bpu_p \neq 0$  then
7:      $\mathcal{C} \leftarrow \mathcal{C} : X_p[e..s]$ 
8:     continue
9:   end if
10:  for ( $j = 0$  to  $|X_p|$  and  $bpu_p \neq 0$ ) do
11:     $currentbit \leftarrow 0$ 
12:    for ( $b = 0$  to  $bpu_p$ ) do
13:      for ( $k = 0$  to 7) do
14:        if ( $BB_p[bpu_p * j + b][k]$ ) then
15:          Insert vertex  $X_p[j]$  to  $C[currentbit]$ 
16:        end if
17:      end for
18:    end for
19:  end for
20:   $\mathcal{C} \leftarrow \mathcal{C} : C$ 
21: end for
22: return  $\mathcal{C}$ 
  
```

Capítulo 5

RESULTADOS

En esta sección se presentan las características de los grafos $G(V, E)$ y de la estructura compacta que 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 memoria RAM, y los algoritmos fueron implementados con el compilador g++ 8.2.1 con la opción de optimización O3.

5.1 Grafos y estructura compacta

Los grafos a evaluar son todos no densos y no dirigidos. Se consideran los grafos dblp-2010 y dblp-2011 de WebGraph¹, snap-dblp y snap-amazon de SNAP², marknewman-astro y marknewman-condmat de Quick-Cliques³, y ca-coauthors de Network repository⁴. En la Tabla 5.1 se muestran la cantidad de vértices, aristas, cliques maximales, grado medio (\bar{d}) y máximo (d_{max}) de los vértices de los grafos, su coeficiente de clusterización y su transitividad.

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. En general los grafos contienen una gran mayoría de cliques pequeños y algunos pocos de gran tamaño, a excepción del grafo snap-amazon que no presenta cliques muy grandes. Mención especial requiere ca-coauthors el cual, aún cuando presenta la misma tendencia que los demás, contiene una cantidad considerable de cliques de hasta ~~tamaño~~ 100 vértices, a diferencia de los demás grafos donde la mayoría de sus cliques no superan los 25 vértices.

Para generar la estructura compacta se usa la librería sds-lite⁵ desarrollada por

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/>

⁵<https://github.com/simongog/sds-lite>

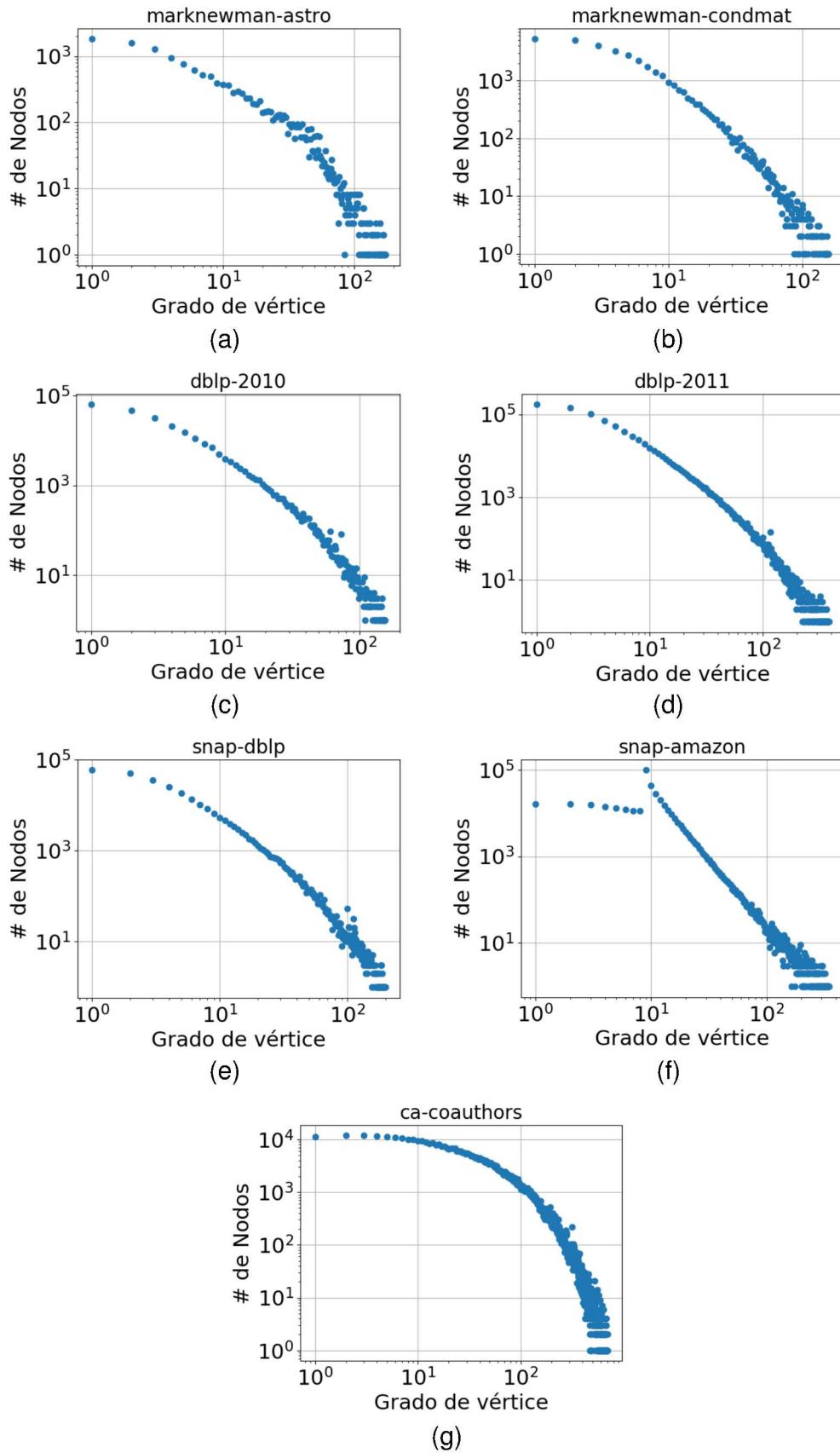


Figura 5.1: Distribución del grado de los vértices para cada grafo.

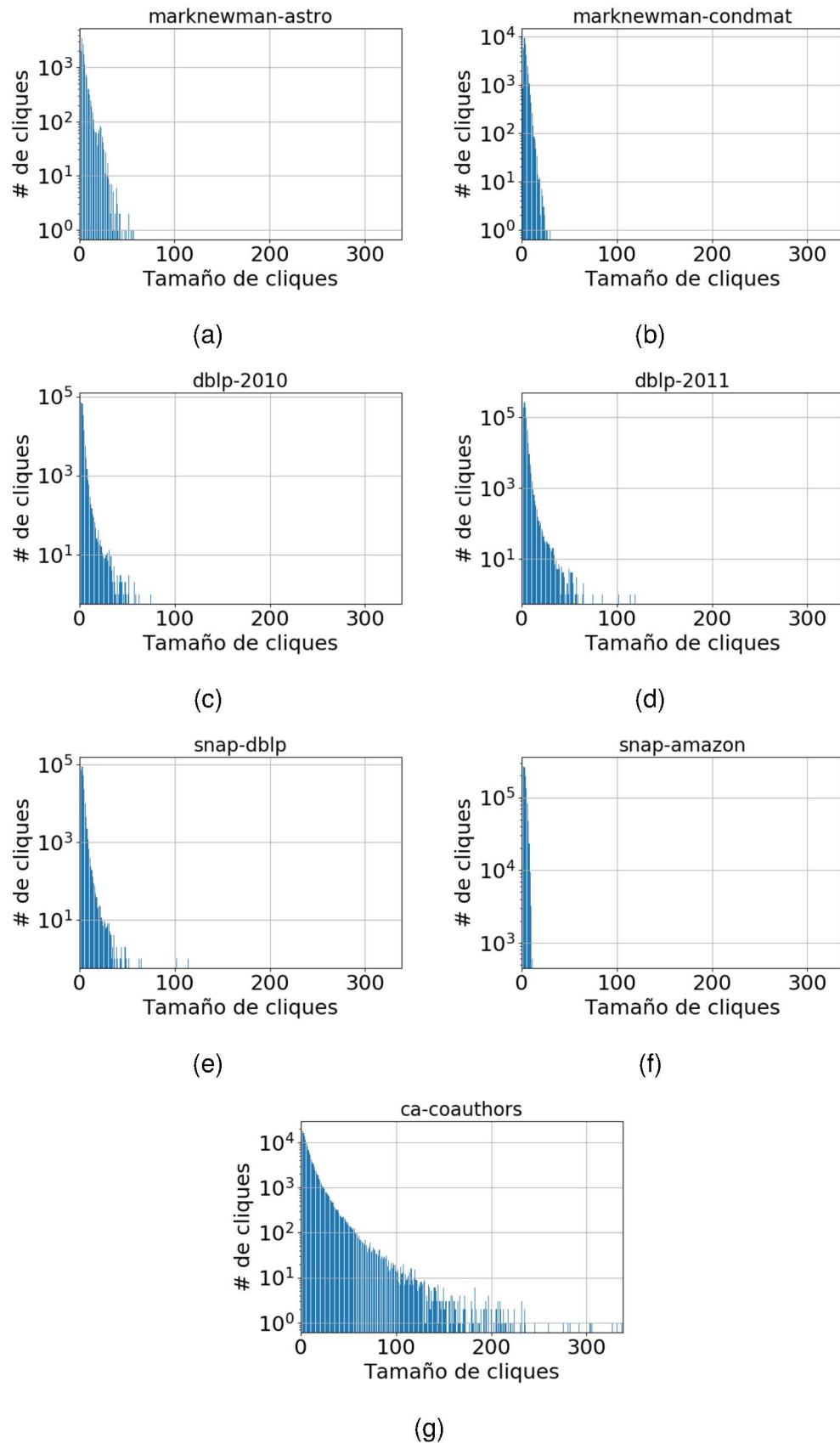


Figura 5.2: Distribución del tamaño de los cliques máximos para cada grafo.

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

Grafo	$ V $	$ E $	$ \mathcal{C} $	\bar{d}	d_{max}	$C(G)$	$T(G)$
marknewman-astro	16.706	242.502	15.794	14,51	360	0,66	0,42
marknewman-condmat	40.421	351.386	34.274	8,69	278	0,64	0,24
dblp-2010	326.186	1.615.400	196.434	4,95	238	0,61	0,39
dblp-2011	986.324	6.707.236	806.320	6,80	979	0,63	0,20
snap-dblp	317.080	2.099.732	257.551	6.62	2.752	0,63	0,30
snap-amazon	403.394	4.886.816	1.023.572	12,11	343	0,41	0,16
ca-coauthors	540.486	30.491.458	139.340	56,41	3.299	0,80	0,65

Gog et al.[20]. Las estructuras de datos sucintos 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*) [].
- Para la secuencia de bits, se consideran las estructuras basadas en bitmaps comprimidos de Raman, Raman y Rao (*rrr*) [30], y Okanohara y Sadakane (*sdb*) [28].
- Para la secuencia de bytes, se consideran las estructuras basadas en Hu-Tucker (*hutu*) [7], y Huffman (*huff*) [26].

Se toman como factores de selección el nivel de compresión en bits y el tiempo de reconstrucción secuencial, usando el Algoritmo 4.2, para cada grafo y cada función de ranking con cada una de las estructuras antes planteadas.

En la Tabla 5.2 se presentan los bits que ocupa cada estructura de datos sucintos para la función de ranking $r_f(u)$, en la Tabla 5.3 lo equivalente para la función $r_c(u)$, y en la Tabla 5.4 para la función $r_r(u)$. Se puede apreciar que en la mayoría de los casos, para las secuencias de símbolos *wt* requiere menos bits, para las secuencias de bits *rrr* requiere un poco menos de la mitad de bits que *sdb*, y para las secuencias de bytes *hutu* requiere menos bits que *huff*. Pero es necesario incorporar el análisis de tiempo de reconstrucción antes de seleccionar las mejores estructuras para cada secuencia.

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 5.4 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 el mismo BPE con respecto al tiempo de acceso aleatorio al recuperar los vecinos de un millón de nodos.

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

Tabla 5.2: Espacio en bits de las estructuras de datos sucintos para la función $r_f(u)$.

Grafo	X_{wm}	X_{wt}	B_{rrr}	B_{sdb}	BB_{hutu}	BB_{huff}	Y_{wm}	Y_{wt}
marknewman-astro	507.512	504.568	17.240	26.416	450.888	453.128	44.856	42.744
marknewman-condmat	1.206.392	1.203.960	36.568	53.504	710.888	712.424	93.176	91.704
dblp-2010	6.376.120	6.373.560	264.536	452.072	2.233.960	2.237.288	724.664	726.200
dblp-2011	27.129.144	27.120.312	792.408	1.155.008	13.417.192	13.468.392	2.249.848	2.262.648
snap-dblp	9.157.240	9.150.200	264.152	391.248	3.717.672	3.718.760	724.024	725.560
snap-amazon	27.688.056	27.681.592	701.272	1.045.264	18.264.104	18.376.168	2.194.616	2.211.320
ca-coauthors	16.764.152	16.753.208	388.504	518.376	4.709.416	4.757.608	810.104	813.240

Tabla 5.3: Espacio en bits de las estructuras de datos sucintos para la función $r_c(u)$.

Grafo	X_{wm}	X_{wt}	B_{rrr}	B_{sdb}	BB_{hutu}	BB_{huff}	Y_{wm}	Y_{wt}
marknewman-astro	520.504	517.880	17.368	29.056	443.720	446.280	48.184	45.880
marknewman-condmat	1.225.016	1.222.584	36.504	57.048	704.520	706.760	96.696	94.904
dblp-2010	6.721.336	6.715.640	264.792	455.976	2.237.544	2.243.368	762.296	762.552
dblp-2011	28.130.424	28.117.304	800.664	1.172.680	13.362.280	13.419.688	2.347.192	2.359.928
snap-dblp	9.337.976	9.329.784	267.864	398.040	3.710.120	3.712.424	756.792	760.952
snap-amazon	28.092.216	28.087.160	720.600	1.074.328	18.155.816	18.265.640	2.298.040	2.316.216
ca-coauthors	17.791.928	17.784.056	419.992	554.488	4.341.608	4.370.536	912.184	915.448

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.

5.2 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 máximales $r_f(u)$, 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 5.5 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 5.6 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 5.5 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 5.7(a) se ilustra la proporción de bits para cada secuencia dentro de la estructura compacta, para cada función de ranking. En la Figura 5.7(b) se ilustra

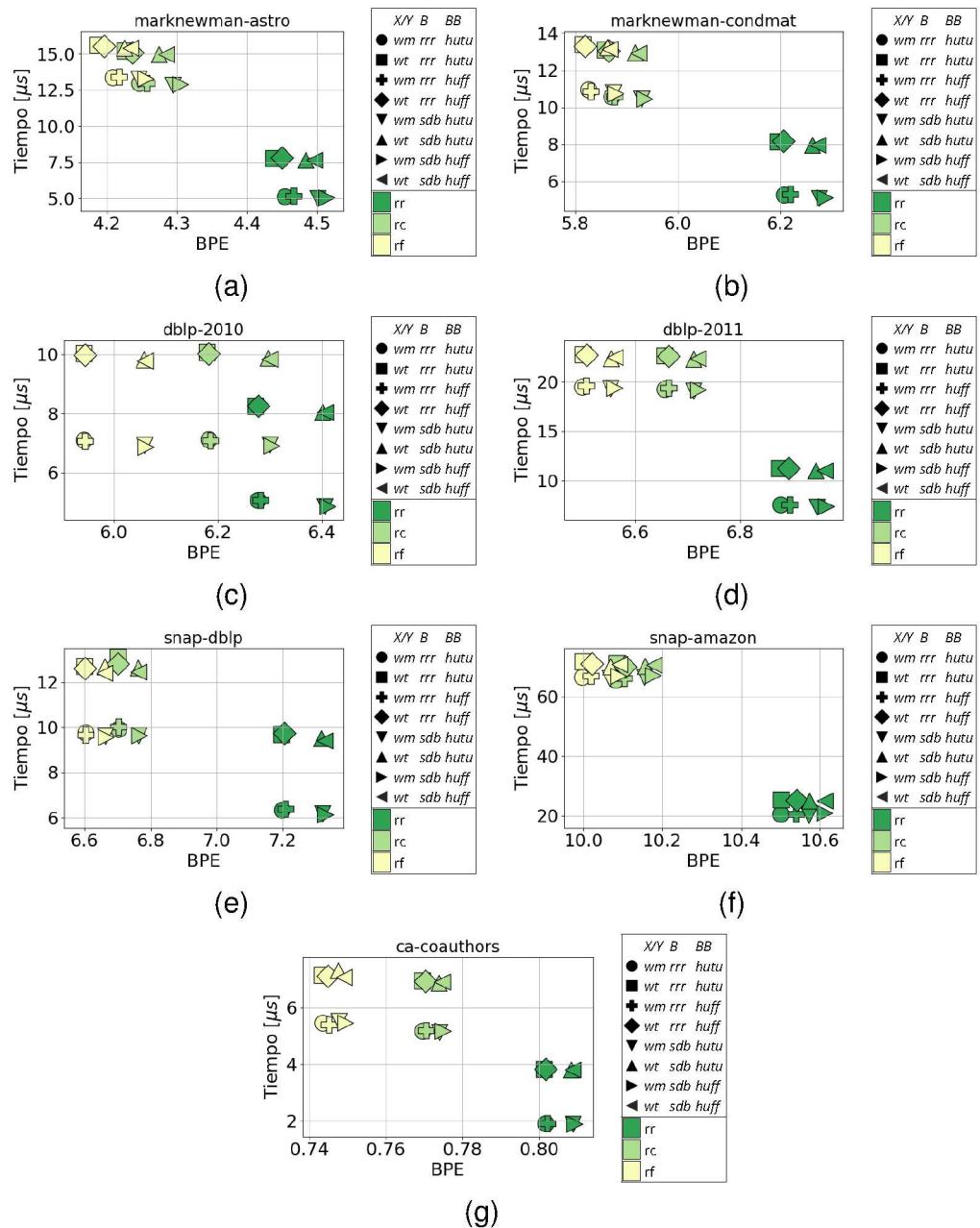


Figura 5.3: BPE y Tiempo de acceso aleatorio medio para posibles estructuras compactas, por cada función de ranking para cada grafo.

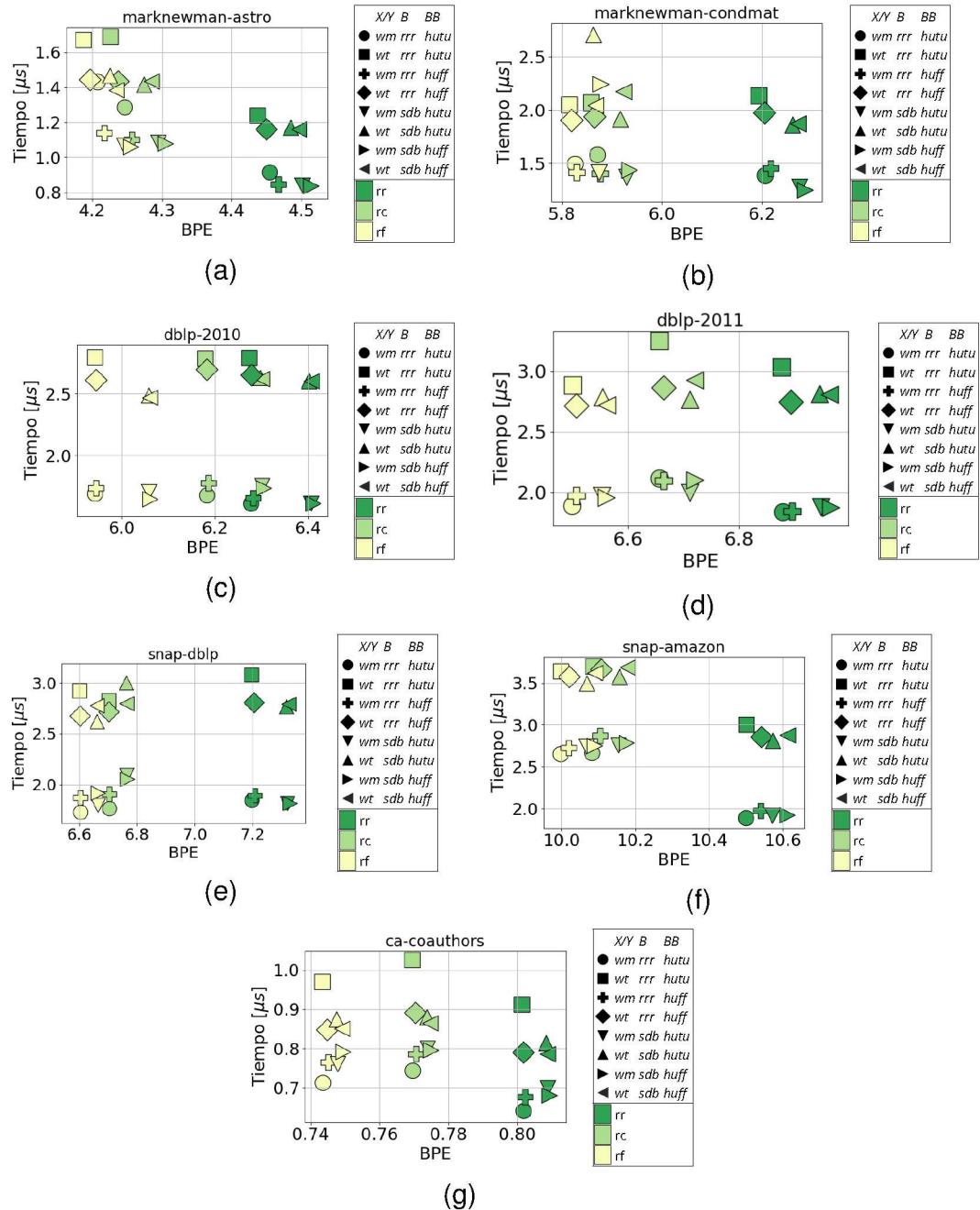


Figura 5.4: BPE y Tiempo de reconstrucción secuencial medio para posibles estructuras compactas, por cada función de ranking para cada grafo.

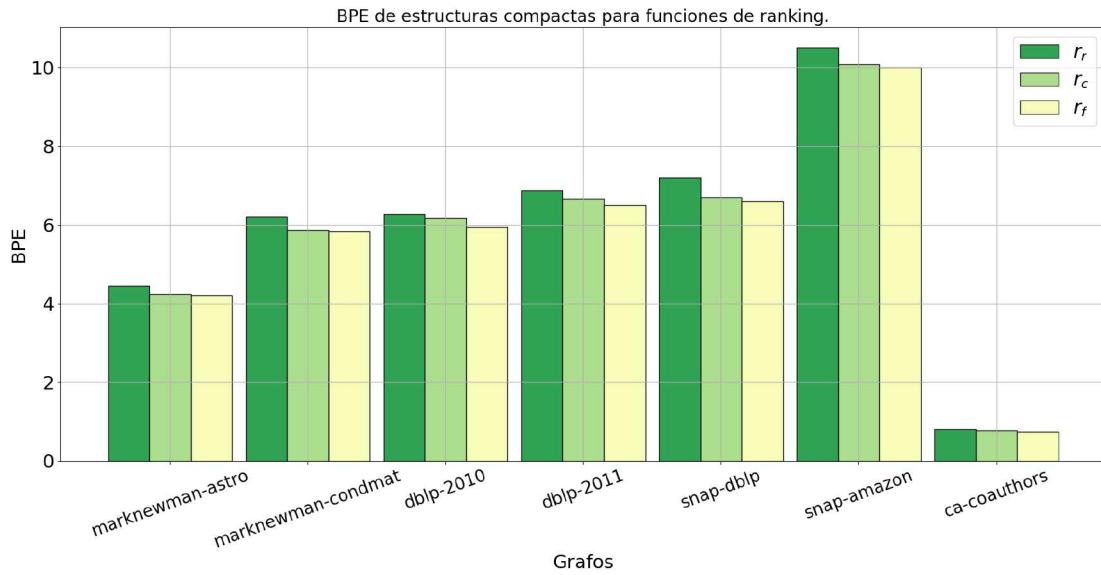


Figura 5.5: BPE de las estructuras compactas para las funciones de ranking.

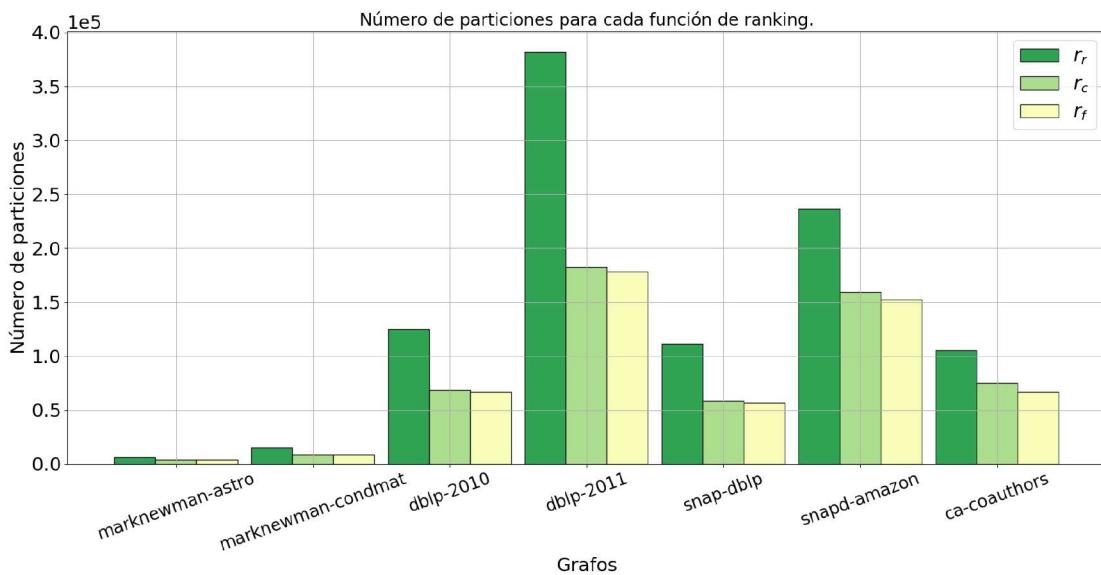


Figura 5.6: Número de particiones en las estructuras compactas para las funciones de ranking.

Tabla 5.4: Espacio en bits de las estructuras de datos sucintos para la función $r_r(u)$.

Grafo	X_{wm}	X_{wt}	B_{rrr}	B_{sdb}	BB_{hutu}	BB_{huff}	Y_{wm}	Y_{wt}
marknewman-astro	657.528	655.032	26.200	37.568	331.912	334.920	64.568	62.904
marknewman-condmat	1.494.392	1.492.024	61.720	85.536	466.472	470.504	158.328	156.152
dblp-2010	7.412.280	7.407.480	430.424	638.616	1.088.424	1.097.128	1.207.864	1.206.520
dblp-2011	32.853.944	32.828.536	1.499.800	1.963.152	7.279.144	7.397.224	4.497.976	4.508.216
snap-dblp	11.034.360	11.027.576	450.520	706.304	2.321.960	2.341.736	1.308.280	1.308.728
snap-amazon	32.730.680	32.719.032	1.052.056	1.402.544	14.134.568	14.330.472	3.394.104	3.410.680
ca-coauthors	21.342.328	21.329.848	573.528	788.080	1.439.880	1.453.192	1.091.192	1.089.528

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 , 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 5.8 se ilustran la cantidad máxima de vértices en la secuencia X por partición, y en la Figura 5.9 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 5.10 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 BB , ya que la cantidad de bytes por vértice es significativamente menor.

En la Figura 5.11 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 $r_r(u)$.

En la Tabla 5.5 se muestran los tiempos en segundos de la generación del listado de cliques maximales \mathcal{C} (t_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 ($t_T = t_C + t_{CS}$) y el tiempo para recuperar el listado de cliques \mathcal{C} desde la estructura compacta (t'_C) usando el Algoritmo 4.4.

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

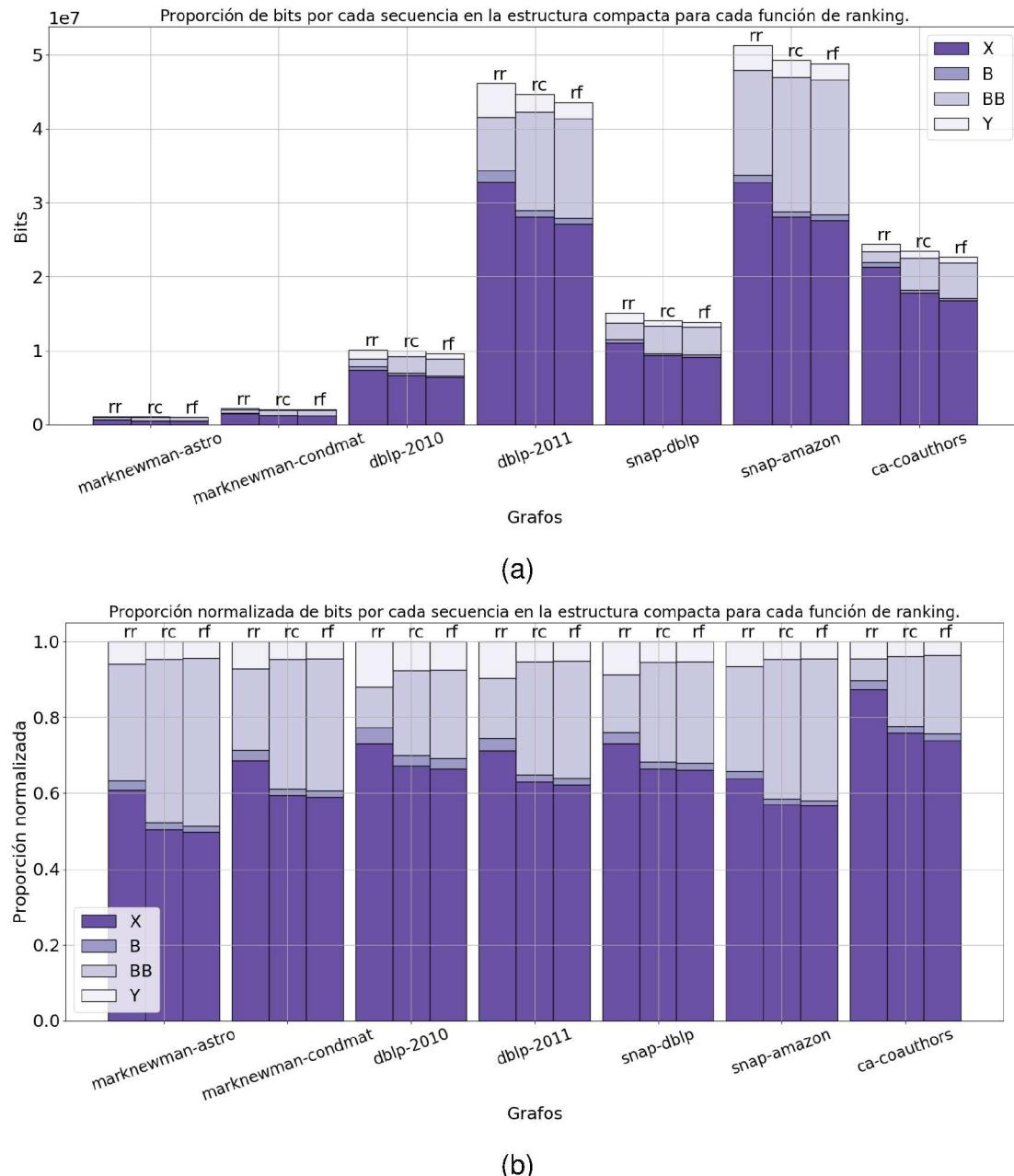


Figura 5.7: (a) Proporción de bits por cada secuencia en la estructura compacta, para cada función de ranking. (b) Proporción normalizada.

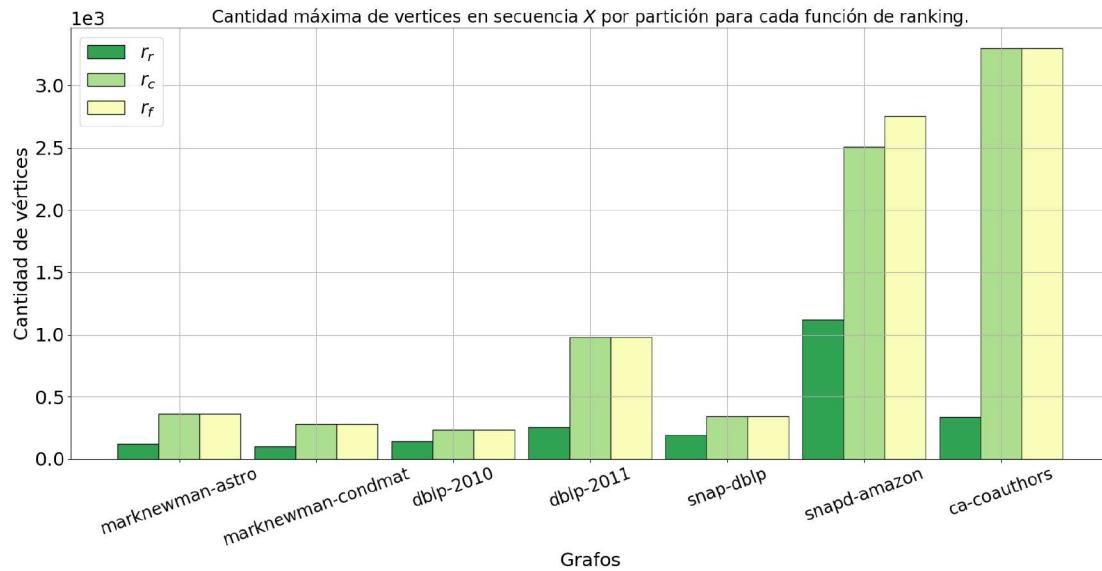


Figura 5.8: Número máximo de vértices en la secuencia X para las funciones de ranking.

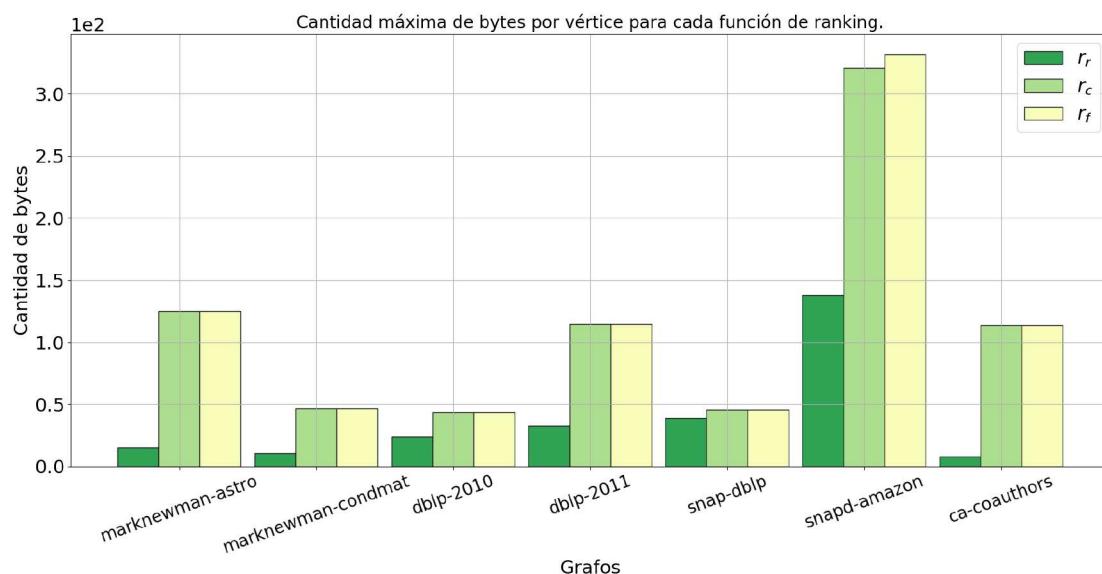


Figura 5.9: Número máximo de bytes por nodo para las funciones de ranking.

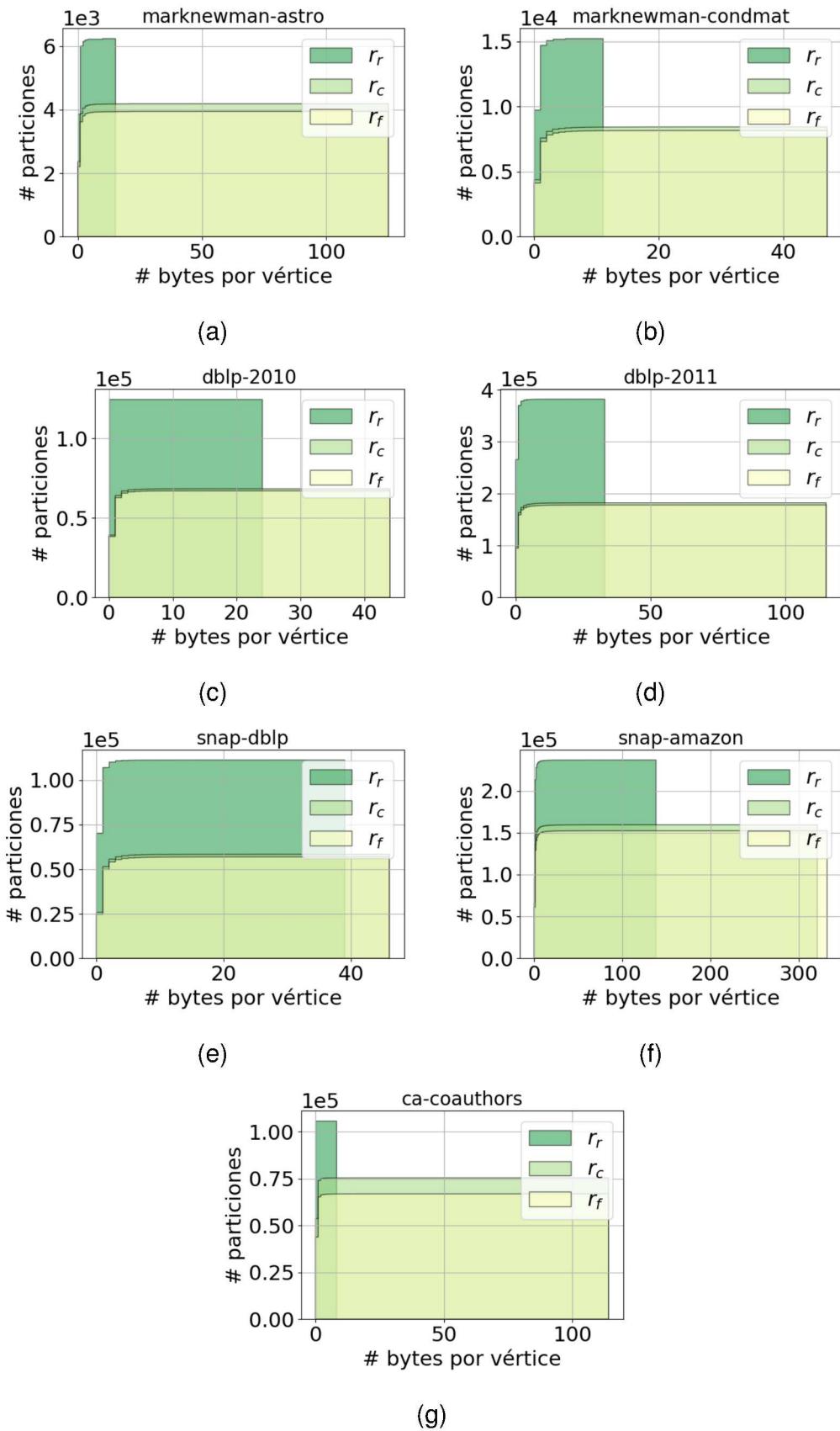


Figura 5.10: CDF para bytes por vértice en estructuras compactas para cada función de ranking.

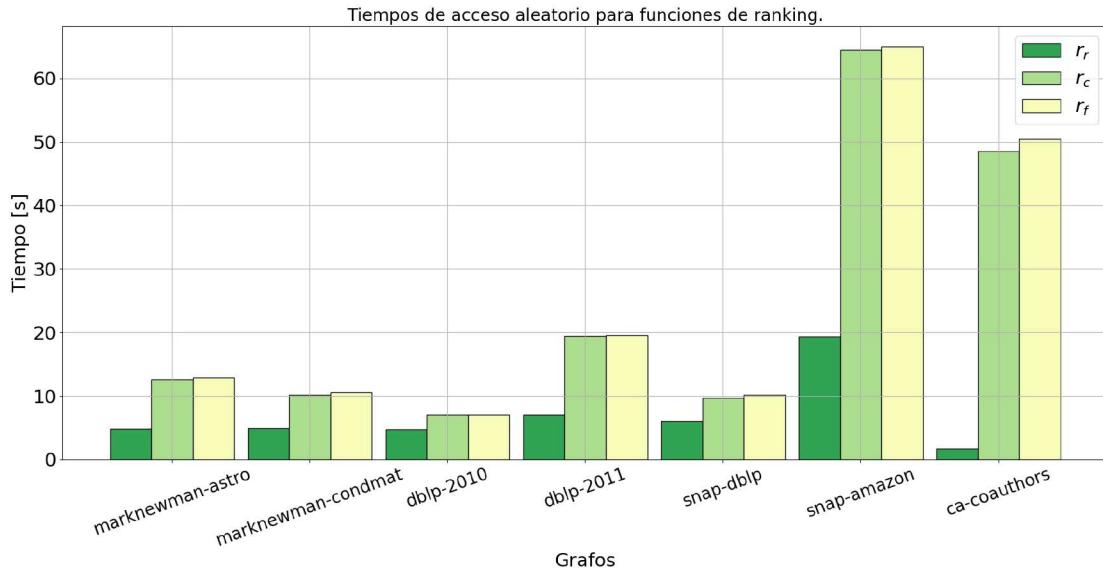


Figura 5.11: Tiempos de acceso aleatorio para las funciones de ranking.

argumentar que para llegar a la estructura compacta se debe generar el listado desde el grafo, por tanto t_c es necesario pagarla ineludiblemente, una vez generada la estructura se puede obtener \mathcal{C} directamente de ella, en menor tiempo y sin tener que descomprimir el grafo.

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

Tabla 5.5: Tiempos de obtención de listado de cliques maximales y construcción de la estructura compacta, en segundos.

Grafo	t_c	t_{CS}	t_T	t'_c
marknewman-astro	0,18	0,28	0,46	0,10
marknewman-condmat	0,28	0,40	0,68	0,18
dblp-2010	1,12	1,46	2,58	0,76
dblp-2011	5,58	7,30	12,88	3,55
snap-dblp	1,68	2,30	3,98	1,12
snap-amazon	5,93	8,44	14,37	4,12
ca-coauthors	17,96	3,44	21,40	1,60

Tabla 5.6: BPE de algoritmos de compresión.

Grafo	$clique_{rr}$	$k2tree_O$	$k2tree_{BFS}$	AD	WG_a	WG_s
marknewman-astro	4,45	9,28	8,05	5,67	8,10	7,30
marknewman-condmat	6,20	12,06	10,43	7,86	11,78	10,45
dblp-2010	6,28	7,45	8,00	6,71	8,67	6,91
dblp-2011	6,87	10,18	11,37	9,67	10,13	8,71
snap-dblp	7,19	11,21	9,92	8,14	11,80	10,17
snap-amazon	10,49	15,66	12,33	10,96	14,50	13,35
ca-coauthors	0,80	3,29	1,83	1,81	2,71	2,48

5.3 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 $r(u)$ 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.

- Para la estructura compacta propuesta, basada en las superposición de cliques maximales, se anotará como $clique_{rr}$.
- En el caso de k2tree, se diferencia cuando el algoritmo usa el orden del grafo original ($k2tree_O$) del orden por BFS ($k2tree_{BFS}$).
- Para Apostolico and Drovandi, la notación corresponderá a AD .
- 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 secuencial el algoritmo genera una estructura adicional.

Con esto presente, en la Tabla 5.6 se comparan los BPE de todos los casos, resaltando los mejores resultados. Para todos los grafos en estudio, la estructura compacta presenta el mejor nivel de compresión, alcanzando particularmente para el grafo *ca-coauthors* un BPE de 0,80, quiere decir que necesita menos de un bit por arco para realizar la compresión.

Esto además confirma que la elección de la función de ranking $r(u)$ es correcta, ya que si bien presenta un BPE más alto de entre las tres funciones sigue siendo el más bajo en la comparativa. Además, el tiempo de acceso es notablemente menor entre las tres opciones, lo que será muy importante al comparar con otros algoritmos, como veremos a continuación.

Para comparar el tiempo de acceso aleatorio, se prueba el Algoritmo 4.3 recuperando los vecinos de 1.000.000 de vértices aleatorios de $G(V, E)$, y se divide el tiempo que demora dicha solicitud por la cantidad de aristas recuperadas. En la

Tabla 5.7: Tiempos de acceso aleatorio, en microsegundos por arco.

Grafo	$clique_{rr}$	$k2tree_O$	$k2tree_{BFS}$	AD	WG_a
marknewman-astro	4,80	2,54	1,31	1,79	0,052
marknewman-condmat	4,92	5,46	2,75	2,32	0,063
dblp-2010	4,75	5,38	4,74	2,15	0,097
dblp-2011	7,03	11,63	11,08	2,36	0,114
snap-dblp	6,01	10,33	7,15	2,30	0,125
snap-amazon	19,33	14,53	7,33	2,47	0,087
ca-coauthors	1,69	1,94	1,17	0,73	0,045

Tabla 5.7 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. Pero se puede decir que el resultado para esta propuesta sí es competitivo, por ejemplo contra k2tree original logra mejores tiempos en casi todos los casos, y para el grafo ca-coauthors la mayor diferencia es de 1,645 microsegundos, lo que se considera un buen resultado.

El tiempo de reconstrucción secuencial se midió usando el Algoritmo 4.2. En la Tabla 5.8 se presentan los resultados obtenidos para los algoritmos evaluados. En este caso, si bien el método propuesto es mayoritariamente más lento que los demás, para casos como marknweman-astro o marknewman-condmat la diferencia es menor a un segundo, y en otros casos como dblp-2010 o snap-dblp es a lo más cercano a dos segundos. El resto de los casos, los grafos tienen una cantidad considerable de arcos, y tanto snap-dblp como ca-coauthors presentan una relación no proporcional entre cantidad de vértices y cantidad de cliques, como lo muestra la Tabla 5.1.

Para evaluar mejor la competitividad, en la Figura 5.12 se grafica el BPE con respecto al tiempo de acceso aleatorio en microsegundos, y en la Figura 5.13 el BPE con respecto al tiempo de reconstrucción secuencial en segundos obtenidos para cada algoritmo en 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 5.12(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 5.8, y tanto en el caso de marknewman-astro (Figura 5.13(a)) como marknewman-condmat (Figura 5.13(b)) presentan los mejores resultados.

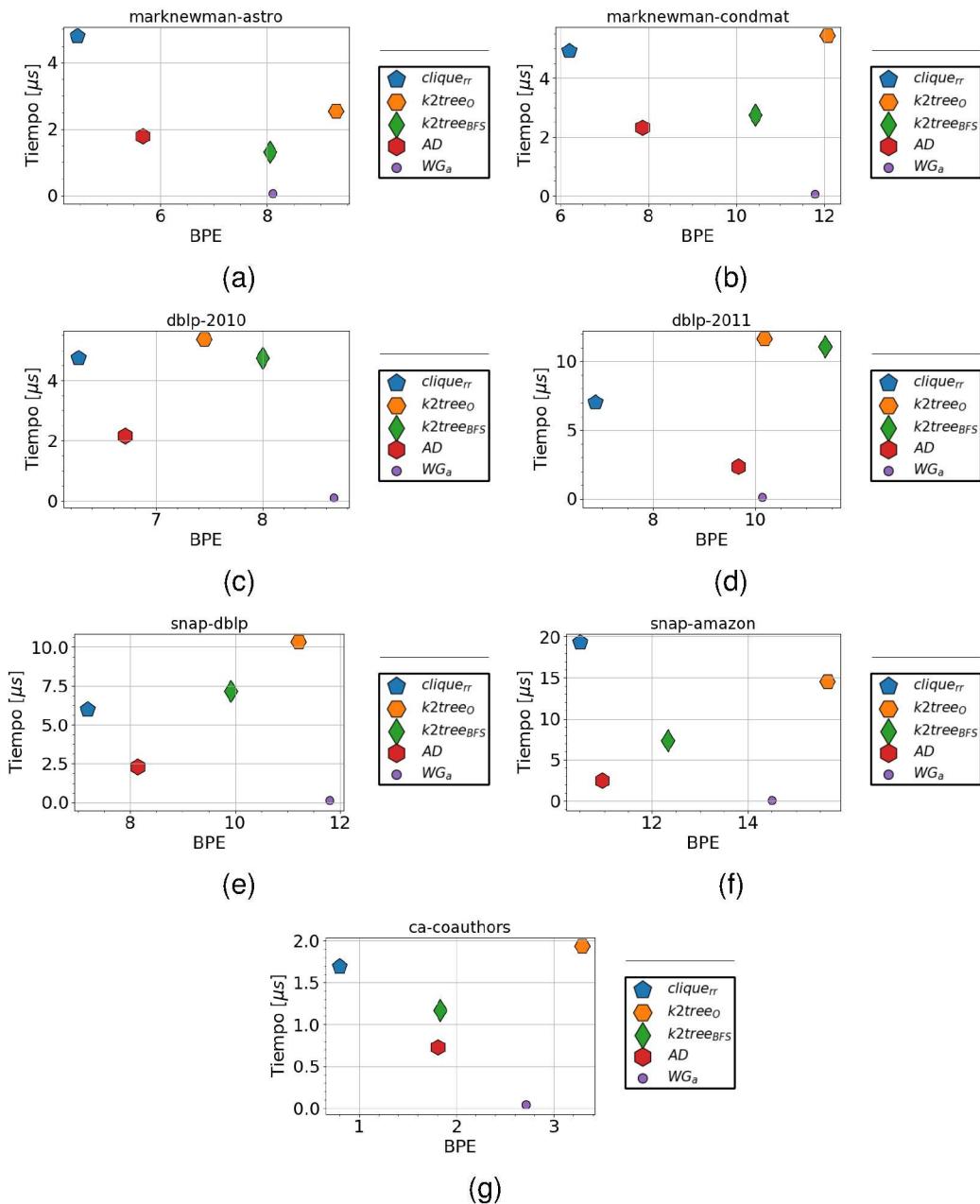


Figura 5.12: BPE y tiempo de acceso aleatorio en microsegundos de cada algoritmo, para cada grafo.

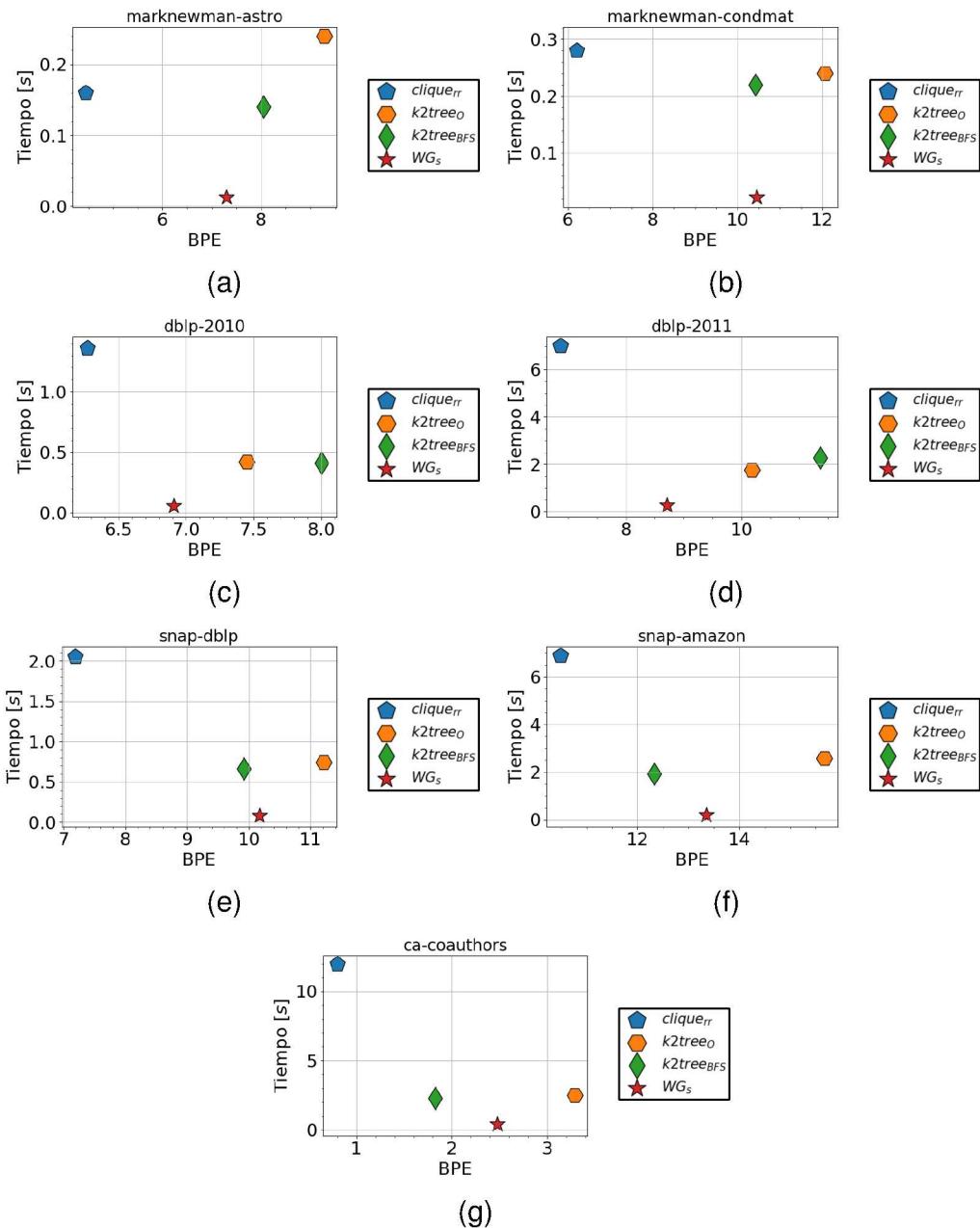


Figura 5.13: BPE y tiempo de reconstrucción secuencial en segundos de cada algoritmo, para cada grafo.

Tabla 5.8: Tiempos de reconstrucción secuencial del grafo, en segundos.

Grafo	$clique_{rr}$	$k2tree_O$	$k2tree_{BFS}$	WG_s
marknewman-astro	0,16	0,24	0,14	0,013
marknewman-condmat	0,28	0,24	0,22	0,022
dblp-2010	1,36	0,42	0,41	0,059
dblp-2011	7,01	1,76	2,28	0,254
snap-dblp	2,05	0,74	0,66	0,083
snap-amazon	6,90	2,57	1,92	0,183
ca-coauthors	11,96	2,50	2,27	0,401

Capítulo 6

CONCLUSIONES

En este trabajo se apunta a 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 ~~descomprimir~~ 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 5.6), superando en todos los casos estudiados a los otros algoritmos. Incluso puntualmente para el caso del grafo ca-coautors, el cual posee el coeficiente de clusterización (0,80) y transitividad (0,65) más altos (ver Tabla ??) se logra un BPE de 0,80, quiere decir que se utiliza menos de un bit por arco, lo que es muy eficiente. Pero no hay gran beneficio que venga sin costo, y en este caso se paga en tiempos de acceso.

El tiempo de acceso aleatorio, medido usando el Algoritmo 4.3 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. Y menos comparando con AD o Webgraph, donde en el mejor de los casos el tiempo logrado es del orden del doble (ver Tabla 5.7).

Y para el tiempo de reconstrucción secuencial, medido usando el Algoritmo 4.2, solo para el grafo marknewman-astro se obtienen resultados mejores con respecto a k2tree con orden original, y similar con BFS. Para el caso del grafo marknewman-condmat el resultado es del mismo orden comparado con los mismos dos algoritmos. En todos los otros casos, Webgraph es muy superior, tanto comparando el algoritmo propuesto como con k2tree, lo que mantiene su superioridad en tiempos de acceso (ver Tabla 5.8).

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 baja memoria 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 al costo en tiempo, puede que no se pueda 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 5.5). 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 explotando el potencial de paralelismo que posee, ya que cada partición en las estructuras compactas se puede acceder de manera simultánea, y cada comparación de bytes dentro de las particiones se puede optimizar usando instrucciones paralelas, como SIMD¹.

¹SIMD: 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 Ordóñez. 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 *International Symposium on Experimental Algorithms*, pages 326–337. Springer, 2014.
- [20] 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.
- [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] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
- [25] 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.
- [26] 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.
- [27] Sebastian Maneth and Fabian Peternek. Compressing graphs by grammars. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 109–120. IEEE, 2016.
- [28] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. *CoRR*, abs/cs/0610001, 2006.
- [29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- [30] 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.
- [31] Cecilia Hernández Rivas. Managing massive graphs. Universidad de Chile, 2014. Tesis de doctorado.