Universidad de Concepción
Facultad de Ingeniería
Dpto. Ingeniería Civil Informática y Ciencias de la Computación

Mejorando las representaciones compactas de grafos planares

Alexander Irribarra Cortés

Tesis presentada a la Facultad de Ingeniería para optar al grado de
Magíster en Ciencias de la computación

Profesores guía:
Diego Seco Naveiras
José Fuentes Sepúlveda

17 de agosto de 2021

1. Introducción

La capacidad de los computadores de procesar y almacenar información ha crecido exponencialmente con el pasar de los años. Sin embargo, el volumen de datos generados y capturados por la humanidad igualmente está creciendo de manera exponencial, e incluso a tasas mayores que el avance de la tecnología, lo que ha motivado el surgimiento de técnicas de procesamiento y almacenamiento de datos cada vez más sofisticadas. Una de las soluciones existentes para esta problemática es el uso de estructuras de datos compactas [25], las cuales reducen el espacio utilizado para almacenar ciertos datos, a la vez que proveen soporte para ciertas operaciones, aunque generalmente con un *trade-off* entre espacio y velocidad en las operaciones.

Esto es importante no sólo por el crecimiento en la generación de datos, sino también, porque cada vez se trata de procesar datos en dispositivos más pequeños, pero cercanos a donde los datos son producidos; esto es el caso en el paradigma *edge computing* [10, 4], debido a que la tendencia es a utilizar dispositivos más portables y con capacidad limitada de almacenamiento como lo son celulares inteligentes, sensores para *IoT* o *wearables*. Las estructuras de datos compactas ya se han utilizado exitosamente en dominios tales como Sistemas de Información Geográfica [13, 12], bioinformática [8, 33], recuperación de información [2, 1], por nombrar algunos ejemplos.

Reducir el espacio tiene como resultado que es posible trabajar con la estructura a niveles en la jerarquía de memoria que se encuentran más próximos al procesador, lo cual no se limita a pasar desde disco a memoria principal, sino que se puede extender incluso a pasar desde memoria principal a algún nivel de caché. De esta manera, es posible en la práctica compensar el *trade-off*, pudiéndose lograr operaciones más rápidas que con métodos convencionales, siempre y cuando los datos no quepan en memoria principal usando los métodos convencionales. Los ejemplos más satisfactorios se han logrado al ser capaces de situar en memoria principal estructuras que antes sólo se podían almacenar en disco, dada la diferencia de órdenes de magnitud en el tiempo de acceso que existe entre ambos niveles de la jerarquía de memoria.

Una aplicación de estructuras de datos compactas, y en lo que se enfoca este trabajo, es en la representación de *planar embeddings*, los que se generan al tomar un grafo planar y darle un orden en particular a las aristas de manera que estas no se crucen al ser dibujadas. Notar que dado un grafo planar, este puede tener más de un *embedding*, como se ve ejemplificado en la Figura 1. Algunas aplicaciones incluyen la representación de mapas y diseño de circuitos.

En este trabajo se propone utilizar como base el trabajo propuesto por Ferres *et al.* [11], el que se explicará en detalle durante la revisión bibliográfica. A modo introductorio, esta representación extiende el trabajo de Turán [34], en el que originalmente

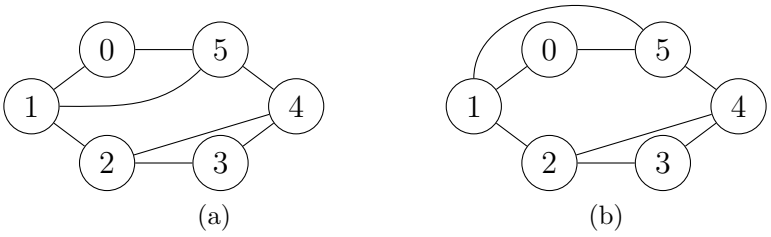


Figura 1: Dos planar embeddings correspondientes al mismo grafo planar.

se presenta una codificación obtenida en base a un *spanning tree* T del grafo planar, el que induce un *spanning tree* T' sobre el grafo dual consistente de las aristas que cortan a las no pertenecientes a T, y a un recorrido en profundidad que visita las aristas del árbol T en sentido anti-horario. En este recorrido, al encontrar una arista perteneciente a T, se escribe el símbolo '(' ó el símbolo ')' dependiendo si es la primera o segunda vez que se visita esa arista respectivamente, y se sigue por ese camino. Si es que la arista no pertenece a T, se escribe el símbolo '[' o el símbolo ']' dependiendo si es la primera o segunda vez que se visita. De esta manera, la codificación tiene un largo de $2m$ y utiliza un espacio de $4m$ bits, donde m es la cantidad de aristas en el grafo. En la Figura 2 se muestra un ejemplo de la representación.

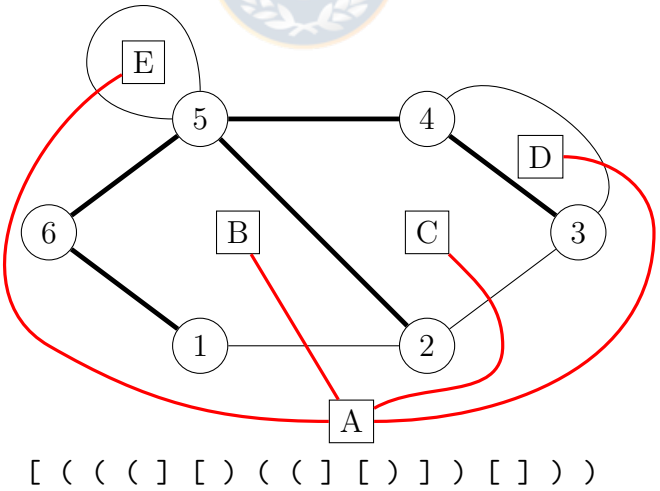


Figura 2: Ejemplo de la descomposición hecha en la representación partiendo desde la arista (1,2). En negrita está el árbol T y en rojo el árbol T'. En la parte inferior se muestra la secuencia de símbolos correspondiente a la codificación de Turán.

Originalmente, Turán solo presenta cómo se obtiene esta codificación y decodificación al *planar embedding original*, además de demostrar el espacio que utiliza, sin

otorgar soporte a operaciones útiles sobre el *embedding*, como la obtención de vecinos de un nodo, del grado de un nodo o el recorrido de una cara. Posteriormente, Ferres *et al.* [11] aumenta esta codificación, de manera que da soporte a estas operaciones, además de mostrar un algoritmo paralelo para la construcción. Para lograr esto, primero se observa que los paréntesis codifican al árbol T mientras que los corchetes codifican al árbol T', de manera que ahora la codificación se separa en tres cadenas de bits A, B y B^*. La cadena A clasifica a las aristas de acuerdo a si pertenecen a T ó T'; lo que indica cómo se entrelazan estos dos árboles. Por su parte, B y B^* corresponden a T y T' respectivamente codificados como secuencias de paréntesis balanceados [24]. Estas tres secuencias se almacenan utilizando estructuras de datos compactas [25], las cuales dan soporte a ciertas operaciones básicas que son utilizadas en conjunto para permitir navegar sobre el *embedding* de manera eficiente. Una de estas estructuras es el *Range Min-Max Tree* [27], el que aumenta una codificación compacta de árboles como secuencias de paréntesis balanceados almacenando cierta redundancia para poder responder consultas de navegación sobre el árbol en tiempo constante en teoría, aunque en la práctica se han logrado tiempos logarítmicos con respecto a la cantidad de nodos. Esto se hace almacenando algunos estadísticos útiles asociados a la secuencia de paréntesis.

Una de las operaciones que soporta el *Range Min-Max Tree*, y que es importante en la representación compacta de *planar embeddings*, es encontrar el paréntesis o corchete complementario a uno dado en las secuencias codificadas. Se conoce que el tiempo de esta operación es logarítmico a la distancia entre pares de paréntesis complementarios (pares de paréntesis que se abren y cierran mutuamente) [25], por lo que es deseable que esas distancias sean lo más pequeñas posibles. En este trabajo se verifica que esto se consigue al utilizar árboles con menor altura.

En la representación de Ferres *et al.* existe la libertad de utilizar cualquier *spanning tree* del grafo planar, por lo que se pueden utilizar árboles de menor altura, como puede ser el obtenido con un recorrido en anchura sobre este grafo. Un desafío es que dado que en esta representación se almacena tanto un árbol T sobre el primal como el árbol T' inducido sobre el dual, disminuir la altura en uno de estos árboles puede llevar a aumentar la altura sobre el otro, lo que puede conllevar a que ciertas consultas se hacen más rápidas mientras que otras más lentas, en particular esto significa un *trade-off* entre consultas sobre el primal contra consultas sobre el dual.

Por otro lado, también se prueban ideas más recientes sobre la representación de Ferres *et al.* como el *Range min tree* [18], el cual es una variante del *Range Min-Max Tree* que utiliza menos espacio y es capaz de conseguir mejores tiempos de consulta. Además, dado que el *Range min tree* se implementa de forma similar al *Range Min-Max Tree*, los resultados observados acerca de la influencia de la topología también aplican en este caso. Combinando ambas técnicas, se consigue una representación

compacta de *planar embeddings* más compacta y rápida que las existentes en el estado del arte.

El resto de este documento se encuentra organizado de la siguiente forma. Primero, en la Sección 2 se repasan conceptos fundamentales sobre estructuras de datos compactas y conceptos preliminares pertinentes sobre *planar embeddings*, para así continuar con las soluciones existentes a este problema. En la Sección 3 se presentan los métodos desarrollados y luego en la Sección 4 se presentan los experimentos, sus resultados y una discusión de ellos, para finalmente en la Sección 5 presentar las conclusiones obtenidas de este trabajo.

2. Revisión bibliográfica

Entropía. Uno de los conceptos fundamentales de la teoría de la información y utilizado dentro del campo de las estructuras compactas es el de entropía. Este concepto acuñado por Shannon [32] mide el nivel de "incertidumbre" o "sorpresa" en una fuente de información. Sea U el conjunto de todos los valores que puede generar la fuente de información, X una variable aleatoria que con resultados posibles $x_i \in U$, cada uno con probabilidad $p(x_i)$ de ocurrir, la entropía $H(X)$ de la fuente de información se calcula como:

$$H(X) = \sum_{x_i \in U} p(x_i) \log_2 \frac{1}{p(x_i)}$$

Además, utilizando la Desigualdad de Kraft [9] se puede demostrar que $H(X)$ es una cota inferior para el largo esperado de cualquier codificación unívocamente decodificable de X, de manera que se utilizan $H(X)$ bits por símbolo. Existe el caso particular en que cada uno de los resultados posibles en la fuente de información tienen la misma probabilidad de ocurrir, situación en la que la entropía se maximiza. De esa manera, todos los elementos tienen asociados codificaciones del mismo largo. Este valor es llamado entropía de peor caso y se expresa como:

$$H_{wc}(X) = \log_2 |U|$$

Entropía empírica. Sea $S[1..n]$ una cadena sobre un alfabeto σ en la que cada símbolo $s \in \sigma$ se repite c_s veces, se define la entropía empírica de orden cero $H_0(S)$ de la siguiente manera [14]:

$$H_0(S) = \sum_{s \in \sigma} \frac{c_s}{n} \log_2 \frac{n}{c_s}$$

Estructuras de datos compactas. El objetivo general de las estructuras de datos compactas es representar datos utilizando una cantidad de bits cercana al óptimo Z, pero permitiendo realizar ciertas consultas de manera eficiente [20]. Por ejemplo, las estructuras de datos sucintas son una categoría particular de estructuras de datos compactas donde se utilizan $Z + o(Z)$ bits.

Varias estructuras de datos compactas para la representación de objetos más complejos se basan en el uso de estructuras de datos compactas más simples, combinando las operaciones soportadas por estas estructuras simples para generar operaciones más complejas. Las más importantes para este trabajo, pero que también es común ver su uso en otras representaciones, corresponden a representaciones sucintas de bitvectors y árboles sucintos.

Bitvectors. Un bitvector [25] X corresponde a un arreglo de bits con soporte para las siguientes operaciones:

- $access(X, i)$**:** retorna el valor del i-ésimo bit.
- $rank_v(X, i)$**:** retorna el número de bits con valor v entre las posiciones 0 e i.
- $select_v(X, i)$**:** retorna la posición del i-ésimo bit con valor v.

Existen representaciones que dan soporte a estas operaciones tanto si son planas [19, 17, 28] – *que no están comprimidas a su entropía* – como si fueran [30, 26, 16].

Árboles sucintos. Con respecto a árboles sucintos, una manera de representarlos es codificándolos como secuencias de paréntesis balanceados [24]. Para obtener la codificación de un árbol se realiza un recorrido en profundidad del mismo desde la raíz. En este recorrido, al visitar a un nodo primero se escribe el símbolo '(', luego se prosigue visitando a los hijos y tras visitarlos se escribe el símbolo ')'. Cada símbolo escrito en la secuencia utiliza un solo bit, es decir, los símbolos (' podrían corresponder a bits con valor 1 y los símbolos ')' a bits con valor 0. Las principales soluciones existentes son LOUDS [19, 5], DFUDS [5] y Fully Functional [27]. En este trabajo nos enfocaremos en la representación Fully Functional, la cual se implementa por medio de la estructura de datos compacta Range min-Max Tree o RMM-TREE. Esta representación ha demostrado ser la solución más eficiente en la práctica [3].

LOUDS [19, 5]. En esta representación un árbol T de n nodos se codifica en un bitvector $B[0, 2n]$ en el que primero se escribe la secuencia 10[1] en B y luego se realiza un recorrido en anchura del árbol T iniciado desde la raíz. Cada vez que se visita un nodo v en este recorrido se escribe la secuencia $1^{deg(v)}0$ en B, donde $deg(v)$ corresponde al grado del nodo v. Esta representación soporta operaciones sobre los nodos del árbol T en base a consultas de *rank* y *select* sobre B.

DFUDS [5]. Esta representación codifica un árbol T de n nodos en un bitvector $B[0, 2n + 1]$ de manera similar a LOUDS. Primero, se escribe la secuencia 1 en B y luego se realiza un recorrido *preorder* del árbol T iniciado desde su raíz. Al visitar un nodo v en este recorrido se escribe la secuencia $1^{deg(v)}0$ en B. Esta representación soporta más operaciones que LOUDS, pero requiere de soporte a operaciones sobre paréntesis, las cuales son más costosas en la práctica.

Fully Functional [27]. Esta representación codifica un árbol como una secuencia de paréntesis balanceados y da soporte a operaciones sobre el árbol y la secuencia

[1]Esta secuencia 10 codifica una súper-raíz, y permite evitar casos borde.[25, Sección 8.1]

de paréntesis utilizando la estructura de datos RMM-TREE. Se tiene que para una secuencia de paréntesis $P[0, n-1]$ se define la operación $excess(i)$ como $excess(i) = excess(i-1) + 1$ si $P[i] = (, excess(i) = excess(i-1) - 1$ si $P[i] =)$, y para $i = 0, excess(-1) = 0$. La secuencia P se divide en bloques de tamaño b, los que corresponden a los nodos hoja del RMM-TREE. Recursivamente se van generando más nodos en los niveles superiores, los que agrupan dos nodos consecutivos en el siguiente nivel inferior. Por ejemplo, si tenemos un nodo u correspondiente a la subsecuencia $P[s, m]$ y un nodo v correspondiente a la subsecuencia $P[m+1, e]$ con $s < m < e$, entonces podemos agruparlos en un nodo w que correspondería a la subsecuencia $P[s, e]$, siendo u y v los hijos izquierdo y derecho respectivamente. De esta manera, se van generando más nodos en los niveles superiores hasta que se tiene un solo nodo en el nivel superior, el que corresponde a la raíz del RMM-TREE. Un nodo v que representa a la subsecuencia $P[s, e]$ contiene la siguiente información:

- $v.e = excess(e) - excess(s-1)$
- $v.min = \text{mín}_{i \in [s,e]}\{excess(i) - excess(s-1)\}$
- $v.max = \text{máx}_{i \in [s,e]}\{excess(i) - excess(s-1)\}$

En la Figura 3 se ve un ejemplo de RMM-TREE junto al árbol que codifica. Se definen las siguientes primitivas como base para generar consultas sobre la secuencia de paréntesis.

- $fwd\_search(i, d)$: retorna la mínima posición $j > i$ tal que $excess(i) + d = excess(j)$, $d < 0$. Por ejemplo, $fwd\_search(1, -1) = 6$.
- $bwd\_search(i, d)$: retorna la máxima posición $j < i$ tal que $excess(i) + d = excess(j)$, $d < 0$. Por ejemplo, $bwd\_search(4, -2) = 0$.

Para resolver la primitiva $fwd\_search(i, d)$, primero se escanea el bloque que contiene a la posición $i+1$, verificando si la posición j se encuentra en este y retornándola en ese caso. En caso de no encontrarla en ese bloque se procede a recorrer el RMM-TREE desde las hojas hacia arriba, partiendo por el nodo que corresponde al bloque ya escaneado. Para esto, primero se obtiene el valor d' como el exceso desde la posición $i+1$ hasta el final del bloque. Este valor en un momento dado corresponde al desplazamiento en el exceso desde la posición $i+1$ hasta lo que se haya recorrido, por lo que se debe ir actualizando en los casos necesarios.

Cuando se recorre un nodo v en la subida, primero se verifica si es hijo izquierdo o derecho de su padre. En el caso de que sea hijo derecho, simplemente se procede a continuar el recorrido por su padre. En caso de que sea hijo izquierdo, se verifica si el exceso buscado está en su hermano derecho u, es decir, se verifica si $d < d' + u.min$. Si esto se cumple, significa que el exceso d no es cubierto por el nodo u, por lo que se hace

la actualización $d' \leftarrow d' + u.e$ y se sigue subiendo por el padre hasta encontrar un nodo izquierdo cuyo hermano derecho u cubra el exceso buscado, es decir $d \geq d' + u.min$.

Al momento de bajar, esto se hace con la intención de encontrar el nodo hoja más hacia la izquierda que contenga el exceso buscado. Para esto, al visitar un nodo interno, primero se verifica si su hijo izquierdo v contiene el exceso verificando $d \geq d' + v.min$. Si es así, se baja por el hijo izquierdo. En caso contrario se baja por el hijo derecho y se actualiza $d' \leftarrow d' + v.e$. Cuando finalmente se llegue a un nodo hoja, se sabe que el exceso buscado está dentro del bloque que representa, por lo que simplemente se recorre y retorna la posición buscada.

En la Figura 3 se muestra un ejemplo de un RMM-TREE con el árbol del que se construye, y también los datos accedidos en una consulta de ejemplo.

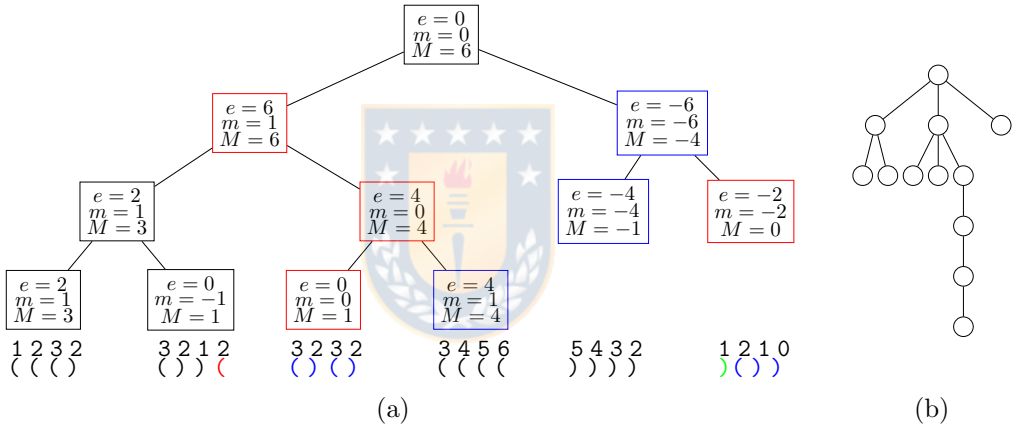


Figura 3: En 3a se muestra el RMM-TREE correspondiente al árbol en 3b usando un tamaño de bloque $b = 4$, junto a la secuencia de paréntesis codificada y los excesos. Se ejemplifica además la consulta $match(7) = fwd\_search(7, -1)$ de la siguiente manera: En la parte inferior se indica en rojo el paréntesis consultado, en azul los bloques revisados y en verde la respuesta $j = 20$. En la parte superior, los nodos indicados en rojo corresponden a aquellos que solo son visitados, mientras que los indicados en azul corresponden a aquellos que además son revisados.

El caso de la primitiva $bwd\_search(i, d)$ es simétrico a la ya descrita, con la salvedad de que el bloque inicial a escanear es aquel que contiene a i en lugar de $i+1$. Además, dado que el recorrido es ahora de derecha a izquierda hay que realizar cambios respecto al mismo. Lo primero es que se sube hasta encontrar un hermano izquierdo que contenga el valor d, y en esta subida el valor d' se actualiza a $d' - v.e$ cuando se sube desde un hijo derecho u con hermano izquierdo v. También, al momento de bajar, se busca el bloque más a la derecha que contenga a d, de manera que se da prioridad

a bajar por el hijo derecho cuando este contiene a d, y cuando se baja por un hijo izquierdo u con hermano derecho v, d' se actualiza a $d' - u.e$. Finalmente, al llegar a una hoja, se escanea el bloque correspondiente de derecha a izquierda.

Algunas de las operaciones sobre la secuencia soportadas por esta representación son las siguientes:

- $match(i)$**:** retorna la posición del paréntesis que abre/cierra al paréntesis en la posición i.

$$match(i) = \begin{cases} fwd\_search(i, -1) & \text{si } P[i] = (\\ bwd\_search(i, 0) + 1 & \text{en otro caso} \end{cases}$$

- $parent(i)$**:** retorna la posición del paréntesis que abre asociado al nodo que es padre del nodo correspondiente al paréntesis en la posición i.

$$parent(i) = \begin{cases} bwd\_search(i, -2) + 1 & \text{si } P[i] = (\\ parent(match(i), -2) + 1 & \text{en otro caso} \end{cases}$$

Range min tree. Existe una variante del RMM-TREE llamada *Range min tree* [18] en la cual no se almacena el máximo en los nodos, de manera que se reduce el espacio. Además se utiliza una estrategia distinta para realizar la búsqueda dentro de un bloque, la cual resulta más rápida. Al momento de hacer la búsqueda dentro de un bloque del RMM-TREE se utiliza una tabla que almacena soluciones para cada posible combinación de byte y exceso buscado. Por otro lado, para el *Range min tree* se propone almacenar una tabla donde se guarda para cada posible byte el exceso mínimo dentro de éste, junto a otra tabla donde para cada byte se guarda el exceso total en el mismo. A esta solución se le llama *RangeMin/Loop*.

Otra solución que se propone es dividir los bloques en palabras de máquina, de manera que, usando las tablas descritas anteriormente, para cada palabra w se calcula:

- Una palabra m_8, en la cual cada i-ésimo byte almacena el mínimo exceso dentro del i-ésimo byte de w.
- Una palabra c_8, en la cual cada i-ésimo byte almacena la cantidad de 1s dentro del i-ésimo byte de w.

Utilizando estos valores y aplicando técnicas de programación a bajo nivel [22] se puede encontrar rápidamente la respuesta dentro de un bloque.

Planar embeddings. Un *planar embedding* corresponde a un "dibujo" en el plano de un grafo planar tal que las aristas no se cruzan, de manera que las aristas tienen un orden particular dado por su orientación. Los *planar embeddings* tienen directa

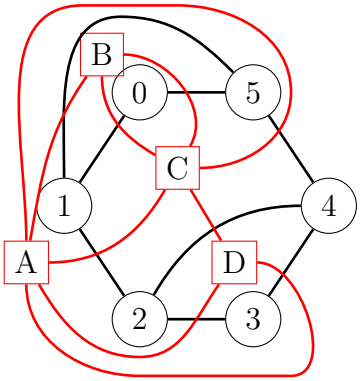


Figura 4: Un *planar embedding*. En color negro está dibujado el grafo primal, mientras que en color rojo está dibujado el grafo dual.

relación con *planar maps*, que corresponden a una subdivisión de la esfera en regiones delimitadas por curvas. Se conoce que existen $\frac{2(2m)!3^m}{m!(m+2)!}$ *rooted planar maps* (mapas planares donde a una arista en particular se le da dirección) con m aristas [35], de manera que la entropía de peor caso de los *planar embeddings* es $m \log 12 \approx 3{,}58m$ [25]. Esto quiere decir que para representar cualquier *planar embedding* se necesitan al menos 3,58 bits por arista. Además, para un *planar embedding* cuyo grafo primal es G se define el grafo dual G' como aquel donde las caras del *embedding* son vértices en G', y las aristas en G' cortan a las aristas de G delimitando las caras. Esto se ve ejemplificado en la Figura 4.

Compact planar embeddings. Para representar *planar embeddings* de manera compacta existen diversas soluciones cada una con sus ventajas y desventajas. Turán [34] presenta una codificación que, como se menciona en la introducción de este documento, utiliza $4m$ bits para representar un *planar embedding* en base al recorrido de un *spanning tree* del grafo. Esta representación utiliza un espacio cercano al óptimo, sin embargo no soporta operaciones de navegación por si sola. Luego, Jacobson [19] presenta una codificación para grafos planares basada en *book embeddings* [6] que soporta operaciones utilizando una cantidad de bits lineal a la cantidad de nodos. Keeler y Westbrook [21] presentan una codificación en espacio óptimo, pero sin soporte para operaciones. Munro y Raman [23] estiman que la representación de Jacobson utiliza $64n$ bits y presentan una representación también basada en *book embeddings* que utiliza $2m + 8n + o(m)$ bits con soporte a operaciones. Blelloch y Farzan [7] presentan una representación de espacio óptimo basada en técnicas utilizadas para grafos separables [7], sin embargo, hasta donde sabemos, no existen implementaciones prácticas de dicha solución. Ferres *et al.* [11] presenta una extensión al trabajo de Turán, la que codifica un árbol sobre el grafo primal y el árbol sobre el dual que este

induce, además del entrelazamiento entre estos, de manera que utilizando estructuras compactas para bitvectors y árboles se puede agregar navegación. Así, se logra una representación para *planar embeddings* con espacio cercano al óptimo y operaciones de navegación.

A continuación se detalla la manera de obtener la representación de Turán [34], la cual corresponde a una codificación de largo $2m$ compuesta de paréntesis y corchetes. Primero se obtiene cualquier *spanning tree* T del grafo G, cuyo complemento T' se corresponde con un *spanning tree* sobre el grafo dual [31]. Luego, comenzando desde una arista incidente en la cara externa del *embedding*, se realiza un recorrido en profundidad del árbol T en el que se recorren las aristas de G en sentido anti-horario. Cuando se visita una arista perteneciente a T, se escribe un paréntesis mientras que si la arista pertenece a T' se escribe un corchete. Cada arista es visitada dos veces, de manera que el paréntesis o corchete escrito abre cuando la arista que le corresponde es visitada por primera vez, mientras que cierra cuando es visitada por segunda vez.

La extensión presentada por Ferres *et al.* [11] corresponde a, en primer lugar, separar la secuencia en tres cadenas: un bitvector A donde $A[i] = 1$ si y solo si la i-ésima arista visitada en el recorrido pertenece a T, y $A[i] = 0$ en caso contrario, una secuencia de paréntesis balanceados B correspondiente a los paréntesis redondos y una secuencia de paréntesis balanceados B^* correspondiente a los corchetes. Las secuencias B y B^* codifican a los árboles T y T' respectivamente, pero en el recorrido no se codifican sus raíces, por lo que se añaden después del recorrido. En la Figura 5 se muestran los árboles codificados para el *embedding* en la Figura 2.

(a) (b)

Figura 5: Los árboles codificados usando la representación de Ferres *et al.* para el *embedding* de la Figura 2. La figura 5a corresponde al árbol sobre el dual, mientras que 5b corresponde al árbol sobre el primal.

Las secuencias se almacenan utilizando estructuras compactas, de manera que

además se da soporte a operaciones como las mencionadas previamente en este capítulo. Algunas de las primitivas definidas por esta representación son:

- $first(v)$: retorna la posición i en A de la primera arista visitada al procesar el nodo v durante la construcción, es decir:
$$first(v) = A.select_1(B.select_0(v-1)) + 1$$
- $last(v)$: retorna la posición i en A de la última arista visitada al procesar el nodo v durante la construcción, es decir:
$$last(v) = A.select_1(B.match(B.select_0(v-1))$$
- $mate(i)$: retorna la posición j en A de la arista (v, w) cuando la arista i en A es (w, v).
- $next(i)$: retorna la posición j en A de la arista (v, w) siguiente en sentido anti-horario a la arista (v, w') de posición i en A.

$$next(i) = \begin{cases} i+1 & \text{si } A[i] = 0 \\ mate(i)+1 & \text{en otro caso} \end{cases}$$

A partir de estas primitivas es posible realizar operaciones más complejas, como encontrar los vecinos de un vértice o recorrer las aristas de una cara. Una de las peculiaridades de esta representación es que si se invierten los bits en A y se intercambian las secuencias B y B^*, la codificación obtenida corresponde a la del grafo dual, con la diferencia de que el recorrido es en sentido horario en lugar de anti-horario. También debido a esto es posible utilizar esta representación para responder consultas sobre el dual, por ejemplo adyacencia de caras. En particular, esto tiene aplicaciones, por ejemplo, en el campo de Sistemas de Información Geográfica, pues permite implementar el modelo topológico [13].

Ferres *et al.* [11] utiliza un recorrido en profundidad para obtener el *spanning tree* usado en la construcción. Sin embargo, como se muestra en esta tesis, esto tiene un efecto negativo en los tiempos de las consultas, ya que esta clase de árbol tiene una altura relativa al vértice desde el cual se inicia el DFS mucho mayor que la obtenida, por ejemplo, por un recorrido en anchura. Los tiempos se ven afectados en la práctica porque aumentar la altura implica también aumentar las distancias entre pares de paréntesis, lo que hace que al momento de realizar consultas sobre las secuencias de paréntesis balanceados se deban visitar más nodos en el RMM-TREE. Esto se explica en detalle en el siguiente capítulo, pues es la base del método propuesto.

3. Mejorando la representación de Turán

3.1. Efecto de la topología de los árboles

En la representación de un árbol como una secuencia de paréntesis balanceados, cada nodo es codificado como un par de paréntesis que se complementan. De esta manera, navegar por el árbol se traduce en navegar por la secuencia de paréntesis, lo que generalmente equivale a encontrar el par de paréntesis correspondientes a un nodo. En el caso del RMM-TREE, esta operación se puede realizar en la práctica con una complejidad temporal de $O(\log \frac{n}{\log n})$, siempre y cuando se utilicen bloques de tamaño $\Theta(\log n)$. Sin embargo, al tomar en cuenta las posiciones a y b del par de paréntesis, esta operación de *matching* toma tiempo $O(\log \frac{b-a}{\log n})$. En consecuencia de esto, es deseable la elección de árboles con una menor distancia promedio entre paréntesis complementarios.

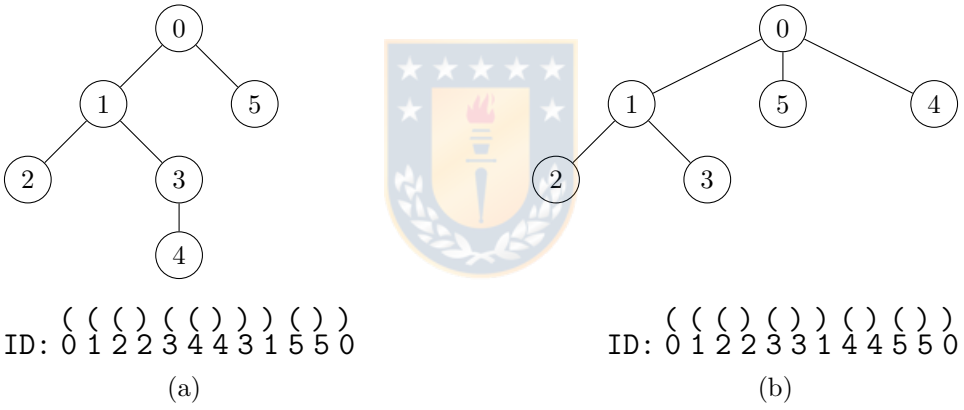


Figura 6: Dos árboles con sus respectivas secuencias de paréntesis balanceados indicando a qué nodo pertenece cada par de paréntesis. El árbol en 6b corresponde al árbol 6a tras reubicar al nodo 4 como hijo de la raíz. Como efecto de esto las distancias de los paréntesis complementarios asociados a los nodos 1 y 3 bajan de 7 a 5 y de 3 a 1 respectivamente.

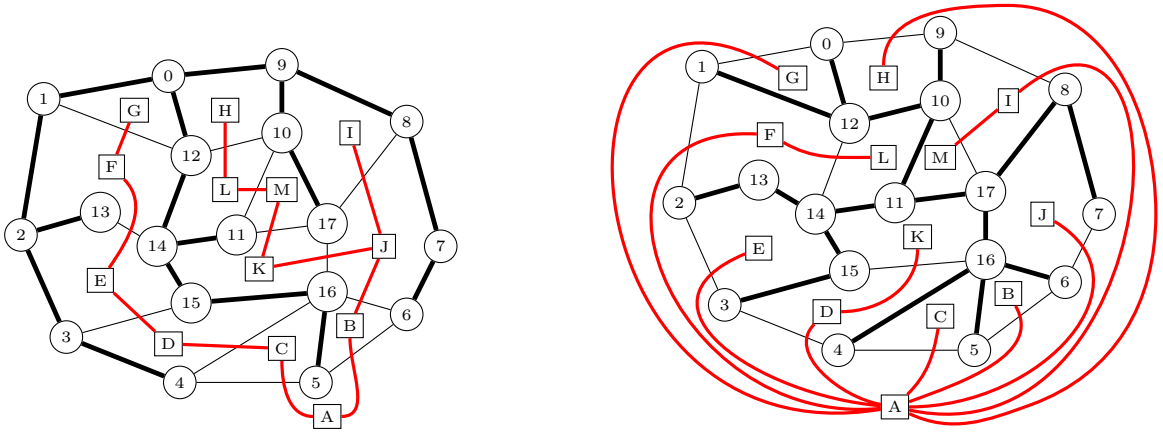
Se puede notar que, dado que la distancia entre el par de paréntesis correspondiente a un nodo depende de su número de descendientes, la distancia promedio entre pares de paréntesis disminuye en conjunto con la altura del árbol. Esto se ve ejemplificado en la Figura 6. Así, un primer enfoque para mejorar los tiempos en la representación de Turán es aplicar métodos de cómputo de los *spanning trees* donde las alturas disminuyan con respecto al *baseline*.

3.2. Computación de árboles de menor altura

Es importante destacar que en la representación de Turán, modificar la topología de uno de los *spanning trees* cambia también la topología de su complementario. Como consecuencia de esto, disminuir la distancia promedio entre pares de paréntesis de un árbol puede aumentar la distancia promedio en el otro. Evaluamos cinco métodos para construir los *spanning trees* de la representación de Turán:

- $\texttt{DFS}_{\texttt{primal}}$: Obtenemos el *spanning tree* sobre el primal como un recorrido en profundidad de los vértices del *planar embedding*, iniciado en algún vértice incidente a la cara externa. Este método es el *baseline*, correspondiente a lo usado por Ferres *et al.* [11].
- $\texttt{BFS}_{\texttt{primal}}$: Obtenemos el *spanning tree* sobre el primal como un recorrido en anchura de los vértices del *planar embedding*, iniciado en algún vértice incidente a la cara externa.
- $\texttt{BFS}_{\texttt{dual}}$: Obtenemos el *spanning tree* sobre el dual como un recorrido en anchura de las caras del *planar embedding*, iniciado en la cara externa. El *spanning tree* sobre el primal se obtiene como el complemento del *spanning tree* sobre el dual.
- $\texttt{DFS}_{\texttt{x}}$: El *spanning tree* sobre el primal se obtiene como un recorrido en profundidad sobre los vértices del *planar embedding* limitado a una altura de `x` veces la altura obtenida por un recorrido en anchura sobre el grafo primal. En caso de que no se consiga visitar todos los vértices, se continua con un recorrido en anchura desde los vértices en el último nivel visitado para reducir la altura.
- `Alt`: Los *spanning trees* se construyen alternando recorridos en anchura sobre el grafo primal y el grafo dual. Es decir, primero expandimos un nivel del árbol sobre los vértices, luego un nivel del árbol sobre las caras y así sucesivamente.
- `DegHeur`: Se utiliza un *heap* H de vértices, donde la prioridad está dada por el grado de cada vértice. Inicialmente, se inserta a H el vértice incidente a la cara externa de mayor grado. En caso de que haya más de un vértice con el mismo mayor grado, se escoge uno de estos arbitrariamente. Luego, mientras H no esté vacío, sacamos de H el vértice de mayor grado u. Por cada vecino v de u que no haya sido visitado aun, agregamos la arista (u, v) al árbol T e ingresamos v a H. Finalizado este recorrido, se tiene un *spanning tree* T válido.

En la Figura 10 se muestra el impacto que tienen los métodos propuestos sobre las topologías de los *spanning trees* sobre los *datasets* `tiger_map` y `PE25M`, los cuales se describen en la Sección 4. En los otros *datasets*, `PE1M`, `PE5M` y `PE10M`, se obtienen resultados similares. Esencialmente, mientras más rápido un método alcanza una frecuencia relativa acumulada de 1 es mejor, pues esto indica una distribución de



(a) BFS sobre el primal

(b) BFS sobre el dual

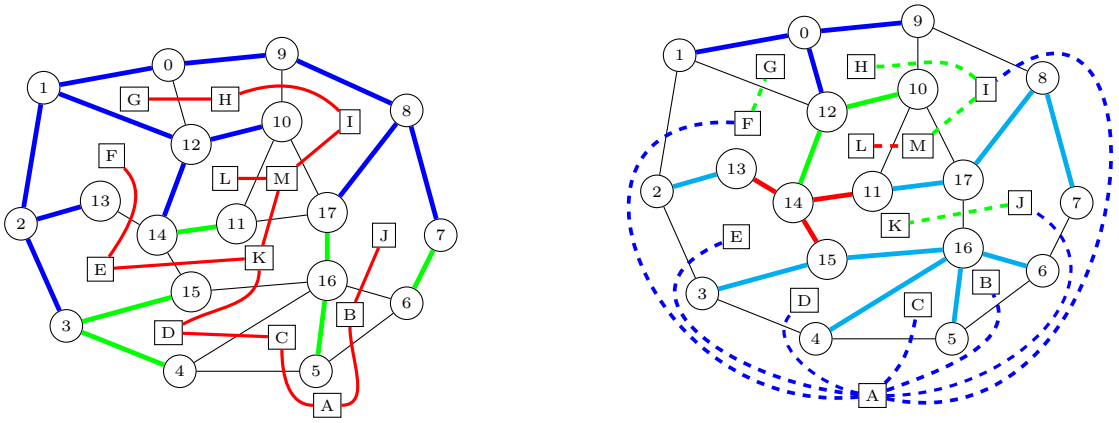
Figura 7: Los árboles T (en negrita) y T' (en rojo) obtenidos los métodos $\texttt{BFS}_{\texttt{primal}}$ y $\texttt{BFS}_{\texttt{dual}}$. En ambos casos la raíz de T es 0 y la de T' es A.

distancias ajustada a valores más pequeños.

Dataset	$\texttt{DFS}_{\texttt{primal}}$		$\texttt{BFS}_{\texttt{primal}}$		$\texttt{BFS}_{\texttt{dual}}$		Alt		$\texttt{DFS}_2$		DegHeur	
	B	B^*	B	B^*	B	B^*	B	B^*	B	B^*	B	B^*
PE1M	263.96	345.84	0.57	5.65	3.11	1.30	0.57	5.65	2.34	6.42	4.78	12.30
PE5M	1261.52	1612.13	1.17	12.07	6.03	2.68	1.17	12.07	5.12	13.22	7.39	33.49
PE10M	2547.27	3265.26	1.86	20.55	9.69	4.29	1.86	20.55	7.01	16.41	11.88	47.00
PE25M	6440.49	8246.12	2.87	32.22	15.16	6.66	2.87	32.22	11.23	25.67	19.39	102.99
tiger_map	1882.50	2482.99	0.27	26.06	5.80	1.96	0.27	26.06	2.03	28.15	2.07	44.82

Tabla 1: Distancias promedio entre pares de paréntesis en B y B^* utilizando cada método. Los valores están multiplicados por 10^{-3} para mejorar la visibilidad.

De manera complementaria, la Tabla 1 indica las distancias promedio obtenidas por los métodos propuestos para los *datasets* descritos en la Sección 4. A pesar del conflicto entre las topologías de B y B^*, se puede notar que todos los métodos propuestos consiguen mejorar considerablemente las distancias promedio de ambos árboles con respecto al *baseline*. En particular, los métodos $\texttt{BFS}_{\texttt{primal}}$ y $\texttt{BFS}_{\texttt{dual}}$ obtienen las mejores distancias para las secuencias B y B^* respectivamente. En la Sección 4 se mostrará cómo estas mejoras en las distancias producen mejoras en los tiempos de las operaciones de la representación.



(a) DFS limitado

(b) BFS alternado

Figura 8: Los árboles T y T' obtenidos utilizando los métodos $\texttt{DFS}_\texttt{x}$ (limitado a una altura de 3) y `Alt`, iniciados desde el vértice 0. En la Figura 8a las aristas en azul son aquellas marcadas en el DFS limitado, las verdes son aquellas marcadas luego de no haber alcanzado todos los vértices, mientras que en rojo se marca T'. En la Figura 8b los colores indican el orden en que se expanden los niveles: azul, verde, rojo y celeste. Las lineas continuas corresponden a T y las discontinuas a T'.

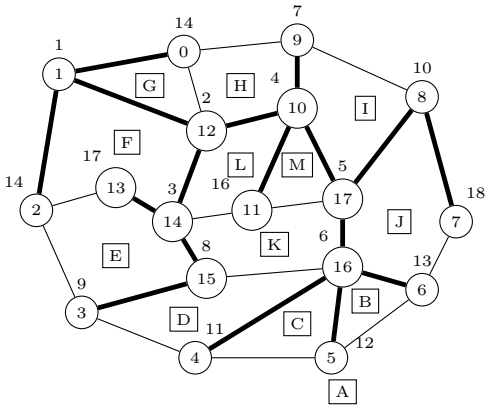
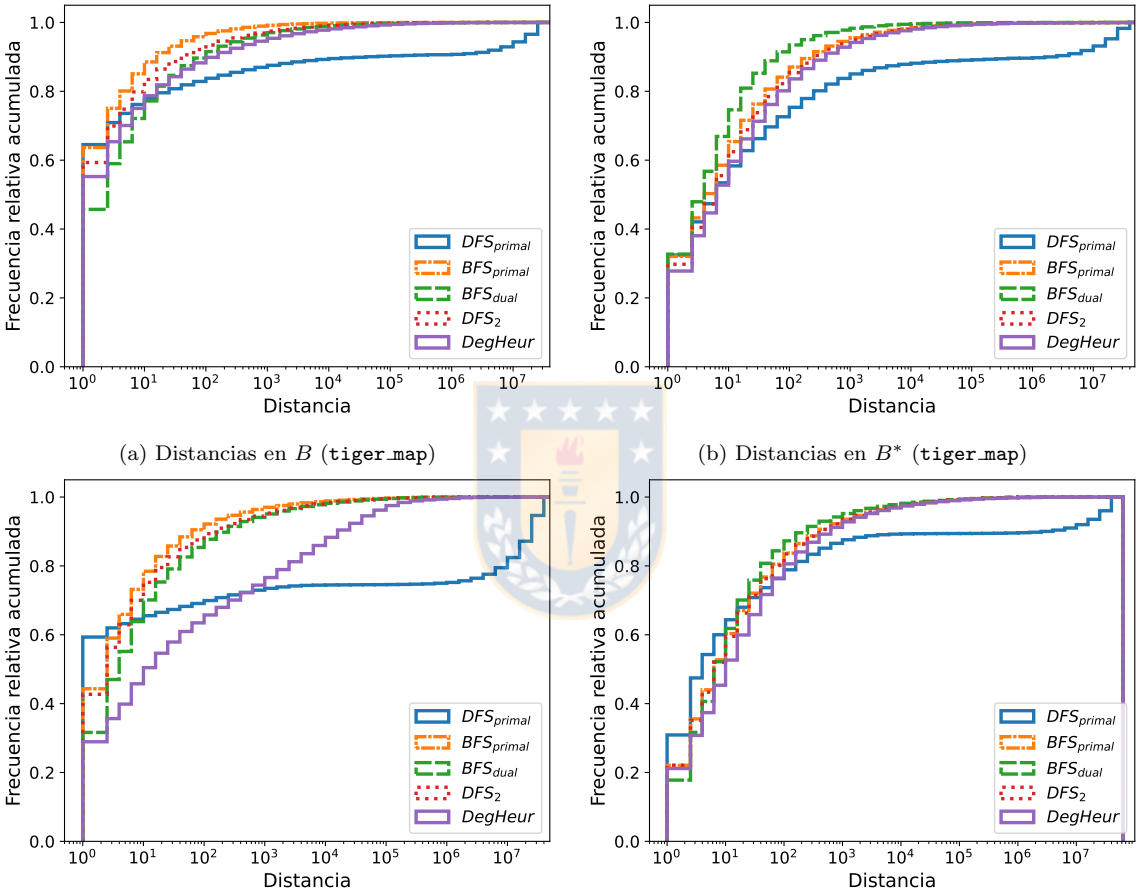


Figura 9: El árbol T, en negrita, obtenido al utilizar el método `DegHeur`, incicando el orden en que los vértices son visitados. No se muestra el árbol T' por visibilidad.



(a) Distancias en B (`tiger_map`)

(b) Distancias en B^* (`tiger_map`)

(c) Distancias en B (`PE25M`)

(d) Distancias en B^* (`PE25M`)

Figura 10: Histogramas de frecuencia relativa acumulada sobre las distancias entre pares de paréntesis correspondientes. `Alt` no se despliega, ya que se comporta de manera similar a $\texttt{BFS}_{\texttt{primal}}$.

3.3. Uso de estructuras de datos alternativas

Si bien ya es posible obtener mejoras con respecto al estado del arte a través del uso de las técnicas anteriormente descritas, también se puede explorar el uso de estructuras de datos para representaciones de paréntesis balanceados más recientes, tal como el *range min tree*[18] descrito en la Sección 2. Eliminar el máximo no afecta a la correctitud de la estructura, pues si $j = fwd\_search(i, d)$, con $d \leq 0$, el exceso acumulado en las posiciones entre i y j son mayores que el exceso acumulado en j, de manera que el primer bloque que tenga un mínimo menor al valor buscado debe contener ese valor buscado. En la práctica, el dejar de almacenar el máximo no solamente reduce el espacio, sino que además mejora los tiempos de las operaciones, tanto por efectos de caché como por realizar menos comparaciones. Esto lo probamos, además de utilizando la implementación del *range min tree* disponible en la biblioteca Succinct [29], creando una variación del *range min-max tree* de la biblioteca SDSL en la que no se almacenan los máximos. Notar que esta última variante no es una implementación completa del *range min tree*, pues no utiliza las tablas precomputadas propuestas por Grossi y Ottaviano [18].

Aplicando estas técnicas en conjunto a los métodos descritos en la Sección 3.2, se puede llegar a una representación todavía más rápida en sus operaciones y que potencialmente utiliza menos espacio, como veremos en la siguiente sección.

4. Análisis experimental

Configuración: Modificamos la implementación de Ferres *et al.* para insertar las modificaciones propuestas en la Sección 3. Nuestra implementación base utiliza la biblioteca SDSL [15].

Para evaluar el desempeño del *range min tree*, utilizamos la implementación disponible en la biblioteca SUCCINCT [29]. El código es compilado utilizando GCC 4.8.4 y el *flag* de optimización -O3. El tiempo es medido utilizando la función `clock`. Los experimentos se ejecutan en una máquina con un procesador Intel Core i7-3820 a 3.60 GHz. Cada núcleo del procesador tiene cachés L1 y L2 de 32KB y 256KB respectivamente, compartiendo una caché L3 de 10MB. La máquina ejecuta Linux 3.13.0-86-generic y posee 32GB de RAM DDR3.

Datasets: En nuestros experimentos utilizamos tanto *datasets* sintéticos como reales con diferentes números de vértices. Los *datasets* son mostrados en la Tabla 2. Los *datasets* sintéticos `pe1M`, `pe5M`, `pe10M` y `pe25M` corresponden a puntos generados al

Dataset	Vértices	Aristas
PE1M	1.000.000	2.999.978
PE5M	5.000.000	14.999.983
PE10M	10.000.000	29.999.979
PE25M	25.000.000	74.999.979
tiger_map	19.785.187	43.903.023

Tabla 2: Datasets utilizados en el análisis experimental

azar en el plano euclideano a través de la función `rnorm` de `R`.[2] Luego, sobre estos puntos se obtiene una triangulación de Delaunay con *Triangle*[3], que es un software para generación de mallas y triangulaciones. Finalmente, los *planar embeddings* se obtienen de estas triangulación con *Edge Addition Planarity Suite*.[4] Adicionalmente, se utilizó el dataset TIGER,[5] provisto por la Oficina del Censo de los Estados Unidos. Este dataset provee información geográfica y cartográfica de las divisiones administravas del territorio de los Estados Unidos. En particular, `tiger_map` corresponde a la parte continental de los Estados Unidos.

Implementaciones: Implementamos múltiples variantes de la representación de Ferres *et al.* La notación utilizada para nombrar dichas variantes es la siguiente: Primero, se define el método utilizado para obtener el *spanning tree*, el cual puede ser $\texttt{BFS}_{\texttt{primal}}$, $\texttt{BFS}_{\texttt{dual}}$ o $\texttt{DFS}_{\texttt{primal}}$, donde este último corresponde al *baseline*. Luego, se describen las estructuras de datos que soportan las operaciones en A, B y B^*. `Succinct` significa que la implementación utiliza solamente las estructuras disponibles en la biblioteca SUCCINCT, mientras que `SDSL` utiliza las implementaciones disponible en la biblioteca SDSL para bitmaps y el *range min-max tree*. Definimos también `minTree` como un método híbrido en el que se utilizan estructuras de datos de la biblioteca SDSL para A y el *range min tree* de la biblioteca `succinct` para B y B^*. Las otras variantes son `v5`, que implementa el bitmap A con *rank_support_v5* de la biblioteca SDSL para el soporte de operaciones *rank*; y `min`, que corresponde a una versión modificada del *range min-max tree* de la biblioteca SDSL, en la cual no se almacenan

[2]La función `rnorm` genera números aleatorios con una distribución normal dada una media y desviación estandar. En este caso, las componentes x e y se generaron usando una media de 0 y deviación estandar de $10,000$. Para más información de esta funcioón, visitar `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html`

[3]Disponible en `http://www.cs.cmu.edu/~quake/triangle.html` http://www.cs.cmu.edu/ quake/triangle.html. Las triangulaciones se obtienen con las opciones `-cezCBVPNE`.

[4]Disponible en `https://github.com/graph-algorithms/edge-addition-planarity-suite`. Los *embeddings* se generaron con las opciones `-s -q -p`.

[5]TIGER dataset, versión 2019. `https://www2.census.gov/geo/tiger/TIGER2019/`.

los excesos máximos. Así, por ejemplo, $\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{v5}}$ obtiene los *spanning trees* a través del método $\texttt{BFS}_{\texttt{primal}}$, implementa el bitmap A con *rank_support_v5* y las secuencias de paréntesis balanceados B y B^* con el *range min-max tree* de la biblioteca SDSL.

Resultados: Realizamos experimentos para las siguientes operaciones:

- *degree*: Obtener el grado de un vértice o cara.
- *neighbors*: Listar los vecinos de un vértice o cara.
- *face*: Dada una arista e, listar las aristas perteneciente a la cara en la que e incide en un recorrido antihorario.
- *dfs*: Realizar un recorrido en profundidad del grafo planar.

Las operaciones se realizan tanto sobre el grafo primal como en el grafo dual. Para las operaciones *degree* y *neighbors* se aplica cada operación sobre todos los vértices del grafo y se reporta el promedio de 15 repeticiones. Análogamente, para la operación *face* se repite la operación sobre todas las aristas del grafo y se reporta el promedio de 15 repeticiones. Finalmente, para la operación *dfs* se escogen cinco vértices iniciales al azar y se calcula el promedio. Todas estas mediciones se repiten tres veces, haciendo variar la arista inicial utilizada para la construcción del árbol, y reportamos medianas.

En la Tabla 3 se muestra el desempeño de los diferentes métodos de obtención de los *spanning trees*, utilizando la variante `SDSL`. Aquí se puede ver que los métodos propuestos se comportan más rápido que el *baseline*, correspondiente a $\texttt{DFS}_{\texttt{primal}}$ alcanzando incluso ser hasta el doble de rápidos. Para los datasets sintéticos, el método `DegHeur` destaca principalmente para el recorrido DFS en el grafo primal y el grafo dual, así como también consigue tiempos cercanos a los mejores sobre las consultas *degree* y *neighbors* en el dual, los cuales son obtenidos por el método $\texttt{BFS}_{\texttt{dual}}$, mientras que para las consultas *degree*, *neighbors* y *face* sobre el primal obtiene resultados similares a $\texttt{BFS}_{\texttt{primal}}$, aunque siguen siendo los mejores. En el caso del dataset *tiger_map*, las diferencias entre los tiempos de ejecución con los métodos $\texttt{BFS}_{\texttt{primal}}$, *Alt* y `DegHeur` son más estrechas, destacando estos métodos para las consultas sobre el primal, mientras que $\texttt{BFS}_{\texttt{dual}}$ lo hace para las consultas sobre el dual.

De manera general, se puede ver que los métodos $\texttt{BFS}_{\texttt{dual}}$ y $\texttt{BFS}_{\texttt{primal}}$ ofrecen un *trade–off* entre tiempos para las consultas sobre el grafo primal y sobre el grafo dual. Los métodos $\texttt{DFS}_{\texttt{x}}$ y `Alt` no ofrecen mayores mejoras para el balance entre consultas sobre el grafo primal y el grafo dual, mientras que `DegHeur` obtiene resultados interesantes para los datasets artificiales.

En la Tabla 4 y en la Tabla 5 se despliegan resultados para los *datasets* `tiger_map`

y PE25M respectivamente, haciendo variar tanto el método de obtención de los *spanning tree* como las estructuras de datos subyacentes de la representación. La principal conclusión es que la mayoría de los nuevos métodos son más rápidos que el *baseline*, utilizan menos espacio, o ambas. Respecto al uso de espacio, los mejores resultados se obtienen al usar nuestra versión modificada del *range min-max tree* de la biblioteca SDSL, la cual almacena solo los valores mínimos, en conjunto a *rank_support_v5* para A. Esta propuesta reduce aproximadamente un 8 % el espacio utilizado por el *baseline*. Notar que utiliza ligeramente menos espacio que las implementaciones utilizando la biblioteca Succinct, la cual incluye la implementación original del *range min tree* de Grossi y Ottaviano [18], pero esto es debido a las diferencias en el soporte a operaciones de *rank* y *select* proporcionadas por las bibliotecas. Con respecto a los tiempos de ejecución, los mejores tiempos para consultas sobre el primal se obtienen con $\texttt{BFS}_{\texttt{primal}}$ `minTree` y $\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min}}$, mientras que para el grafo dual, por $\texttt{BFS}_{\texttt{dual}}$ `minTree` y $\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{min}}$.

Para simplificar la comprensión de estos resultados, seleccionamos cuatro consultas representativas y graficamos los *trade–off* entre tiempos de consulta y uso de memoria en la Figura 11 y la Figura 12. En cada una de estas figuras, las dos gráficas superiores muestran el desempeño de los métodos al ejecutar la misma consulta, *degree*, sobre el grafo primal (Figura 11a y Figura 12a) o sobre el grafo dual (Figura 11b y Figura 12b). Por lo general, se puede ver que las consultas sobre el grafo primal son más rápidas al realizarlas en una estructura de datos construida usando un *spanning tree* obtenido con un recorrido en anchura en el grafo primal, mientras que las consultas sobre el grafo dual son más rápidas al obtener el *spanning tree* con un recorrido en anchura sobre el grafo dual. Por otro lado, se puede ver que la ventaja que el método `DegHeur` tiene en la variante `SDSL` se pierde para las otras variantes. Las dos gráficas inferiores corresponden a los tiempos de realizar un recorrido en profundidad en el grafo primal y la consulta de vecinos en el grafo dual. Notar que ambas consultas son más costosas que *degree*, especialmente *dfs*.

		Consultas primal				Consultas dual		
Dataset	Método	dfs	degree	neighbors	face	dfs	degree	neighbors
PE1M	$\texttt{BFS}_{\texttt{primal}}$	1.22	0.170	0.889	0.487	1.46	0.327	1.131
	$\texttt{DFS}_{\texttt{primal}}$	2.16	0.300	1.904	1.090	2.30	0.463	1.991
	$\texttt{DFS}_2$	1.25	0.175	0.919	0.504	1.47	0.326	1.136
	$\texttt{BFS}_{\texttt{dual}}$	1.24	0.181	0.924	0.512	1.45	**0.311**	**1.086**
	Alt	1.22	0.170	0.892	0.488	1.46	0.327	1.129
	DegHeur	**1.14**	**0.164**	**0.872**	**0.473**	**1.43**	0.319	1.106
PE5M	$\texttt{BFS}_{\texttt{primal}}$	6.19	0.172	0.905	0.495	7.38	0.331	1.147
	$\texttt{DFS}_{\texttt{primal}}$	10.77	0.299	1.900	1.090	11.52	0.464	1.993
	$\texttt{DFS}_2$	6.40	0.178	0.940	0.518	7.44	0.329	1.154
	$\texttt{BFS}_{\texttt{dual}}$	6.28	0.183	0.942	0.523	7.45	**0.313**	**1.098**
	Alt	6.23	0.172	0.906	0.496	7.38	0.330	1.146
	DegHeur	**5.71**	**0.164**	**0.876**	**0.475**	**7.19**	0.320	1.111
PE10M	$\texttt{BFS}_{\texttt{primal}}$	12.32	0.172	0.906	0.496	14.79	0.330	1.146
	$\texttt{DFS}_{\texttt{primal}}$	21.83	0.303	1.926	1.104	23.27	0.468	2.015
	$\texttt{DFS}_2$	13.15	0.183	0.977	0.541	15.05	0.332	1.173
	$\texttt{BFS}_{\texttt{dual}}$	12.55	0.183	0.943	0.524	14.75	**0.314**	**1.101**
	Alt	12.33	0.172	0.904	0.495	14.76	0.330	1.145
	DegHeur	**11.39**	**0.164**	**0.874**	**0.473**	**14.38**	0.320	1.112
PE15M	$\texttt{BFS}_{\texttt{primal}}$	18.62	0.172	0.905	0.495	22.18	0.330	1.148
	$\texttt{DFS}_{\texttt{primal}}$	33.26	0.306	1.954	1.122	35.41	0.474	2.046
	$\texttt{DFS}_2$	19.78	0.184	0.978	0.542	22.62	0.331	1.174
	$\texttt{BFS}_{\texttt{dual}}$	18.86	0.183	0.944	0.524	22.30	**0.314**	**1.099**
	Alt	18.72	0.172	0.906	0.496	22.15	0.330	1.147
	DegHeur	**17.13**	**0.164**	**0.875**	**0.474**	**21.64**	0.321	1.116
PE20M	$\texttt{BFS}_{\texttt{primal}}$	24.91	0.172	0.907	0.497	29.64	0.331	1.151
	$\texttt{DFS}_{\texttt{primal}}$	44.06	0.307	1.945	1.115	46.62	0.467	2.016
	$\texttt{DFS}_2$	27.22	0.189	1.018	0.566	30.43	0.332	1.189
	$\texttt{BFS}_{\texttt{dual}}$	25.18	0.183	0.945	0.526	29.71	**0.314**	**1.102**
	Alt	24.80	0.172	0.906	0.496	29.60	0.331	1.149
	DegHeur	**22.96**	**0.165**	**0.879**	**0.476**	**28.84**	0.321	1.115

Tabla 3: Tiempos promedio de consulta utilizando las estructuras de datos originales, es decir, la variante `SDSL`. Todos los tiempos están medidos en microsegundos, salvo *dfs* que está medido en segundos. Se destaca en negrita el mejor tiempo para cada consulta en cada dataset.

		Consultas primal				Consultas dual		
Dataset	Método	dfs	degree	neighbors	face	dfs	degree	neighbors
PE25M	$\texttt{BFS}_{\texttt{primal}}$	31.99	0.166	0.919	0.504	37.77	0.328	1.173
	$\texttt{DFS}_{\texttt{primal}}$	58.25	0.299	2.056	1.189	60.96	0.466	2.121
	$\texttt{DFS}_2$	32.57	0.179	0.959	0.527	37.59	0.335	1.174
	$\texttt{BFS}_{\texttt{dual}}$	32.40	0.178	0.964	0.537	38.02	0.321	1.132
	Alt	31.18	0.172	0.906	0.496	37.16	0.338	1.163
	DegHeur	**28.58**	**0.164**	**0.874**	**0.474**	**35.93**	**0.320**	**1.114**
tiger_map	$\texttt{BFS}_{\texttt{primal}}$	16.59	0.142	0.621	0.718	18.89	0.197	0.724
	$\texttt{DFS}_{\texttt{primal}}$	27.43	0.205	1.177	1.309	29.67	0.268	1.265
	$\texttt{DFS}_2$	17.80	0.151	0.675	0.777	19.36	0.201	0.755
	$\texttt{BFS}_{\texttt{dual}}$	18.18	0.156	0.677	0.767	**17.89**	**0.179**	**0.664**
	Alt	**16.49**	0.143	**0.615**	**0.712**	18.47	0.199	0.715
	DegHeur	**16.49**	**0.140**	0.639	0.752	18.94	0.196	0.723

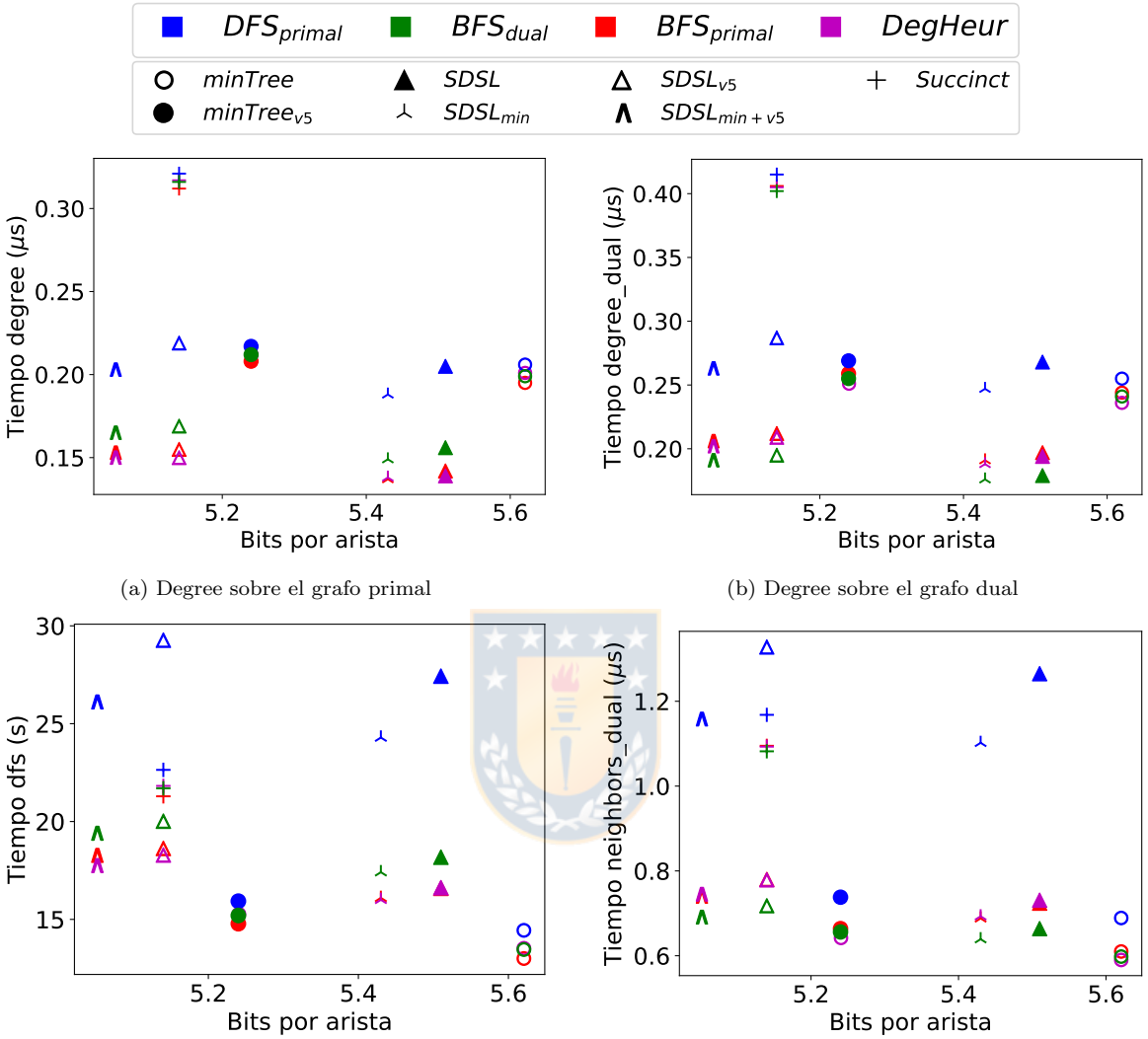
Tabla 3 (continuación): Tiempos promedio de consulta utilizando las estructuras de datos originales, es decir, la variante SDSL. Todos los tiempos están medidos en microsegundos, salvo *dfs* que está medido en segundos. Se destaca en negrita el mejor tiempo para cada consulta en cada dataset.

Método	Grafo primal				Grafo dual			Espacio
	dfs	degree	neighbors	face	dfs	degree	neighbors	
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{minTree}_{\texttt{v5}}$	14.78	0.208	0.590	0.668	15.92	0.259	0.664	28.78
$\texttt{BFS}_{\texttt{primal}}$ minTree	13.00	0.195	0.540	0.608	14.09	0.244	0.610	30.83
$\texttt{BFS}_{\texttt{primal}}$ SDSL	16.59	0.142	0.621	0.718	18.89	0.197	0.724	30.25
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min}}$	16.10	0.137	0.594	0.688	18.05	0.191	0.688	29.78
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min+v5}}$	18.27	0.153	0.646	0.753	20.09	0.206	0.740	27.72
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{v5}}$	18.63	0.155	0.678	0.792	20.79	0.212	0.780	28.20
$\texttt{BFS}_{\texttt{primal}}$ Succinct	21.29	0.312	0.986	1.076	23.18	0.406	1.095	28.23
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{minTree}_{\texttt{v5}}$	15.93	0.217	0.662	0.743	17.43	0.269	0.738	28.78
$\texttt{DFS}_{\texttt{primal}}$ minTree	14.44	0.206	0.617	0.687	15.79	0.255	0.689	30.83
$\texttt{DFS}_{\texttt{primal}}$ SDSL	27.43	0.205	1.177	1.309	29.67	0.268	1.265	30.25
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min}}$	24.29	0.188	1.013	1.133	26.39	0.247	1.102	29.78
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min+v5}}$	26.11	0.203	1.064	1.198	28.39	0.263	1.158	27.72
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{v5}}$	29.27	0.219	1.234	1.386	31.72	0.287	1.328	28.20
$\texttt{DFS}_{\texttt{primal}}$ Succinct	22.64	0.321	1.059	1.152	24.74	0.415	1.168	28.23
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{minTree}_{\texttt{v5}}$	15.20	0.212	0.615	0.687	15.87	0.255	0.656	28.78
$\texttt{BFS}_{\texttt{dual}}$ minTree	13.46	0.199	0.561	0.620	13.98	0.241	0.598	30.83
$\texttt{BFS}_{\texttt{dual}}$ SDSL	18.18	0.156	0.677	0.767	17.89	0.179	0.664	30.25
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{min}}$	17.41	0.149	0.638	0.728	17.26	0.176	0.639	29.78
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{min+v5}}$	19.40	0.165	0.689	0.792	19.33	0.191	0.691	27.72
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{v5}}$	20.02	0.169	0.730	0.835	19.89	0.195	0.718	28.20
$\texttt{BFS}_{\texttt{dual}}$ Succinct	21.70	0.316	1.005	1.087	22.98	0.402	1.082	28.23
DegHeur $\texttt{minTree}_{\texttt{v5}}$	15.27	0.212	0.609	0.678	15.96	0.251	0.642	28.78
DegHeur minTree	13.52	0.201	0.561	0.623	14.01	0.236	0.590	30.83
DegHeur SDSL	16.62	0.139	0.644	0.770	18.89	0.194	0.731	30.25
DegHeur $\texttt{SDSL}_{\texttt{min}}$	16.00	0.138	0.613	0.719	18.13	0.188	0.693	29.78
DegHeur $\texttt{SDSL}_{\texttt{min+v5}}$	17.75	0.150	0.660	0.776	20.08	0.202	0.745	27.72
DegHeur $\texttt{SDSL}_{\texttt{v5}}$	18.29	0.150	0.689	0.818	20.69	0.209	0.779	28.20
DegHeur Succinct	21.82	0.317	1.013	1.092	23.17	0.405	1.093	28.23

Tabla 4: Resultados de los experimentos para el *dataset* `tiger_map`. Los valores desplegados corresponden a microsegundos, salvo para las columnas **dfs** y **Espacio**, cuyos valores están en segundos y Megabytes respectivamente. El *baseline* corresponde a la fila $\texttt{DFS}_{\texttt{primal}}$ SDSL, la cual está coloreada. Los mejores resultados para cada columna están subrayados.

Método	Grafo primal				Grafo dual			Espacio
	dfs	degree	neighbors	face	dfs	degree	neighbors	
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{minTree}_{\texttt{v5}}$	24.82	0.227	0.754	0.397	30.04	0.418	1.000	49.10
$\texttt{BFS}_{\texttt{primal}}$ minTree	21.75	0.214	0.693	0.362	26.50	0.392	0.911	52.62
$\texttt{BFS}_{\texttt{primal}}$ SDSL	31.99	0.166	0.919	0.504	37.77	0.328	1.173	51.60
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min}}$	30.58	0.163	0.869	0.474	36.36	0.317	1.116	50.80
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min+v5}}$	34.13	0.175	0.931	0.511	40.27	0.342	1.182	47.28
$\texttt{BFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{v5}}$	35.25	0.179	0.982	0.539	41.63	0.348	1.253	48.09
$\texttt{BFS}_{\texttt{primal}}$ Succinct	35.10	0.323	1.273	0.645	44.17	0.691	1.673	48.22
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{minTree}_{\texttt{v5}}$	26.47	0.245	0.880	0.471	32.26	0.433	1.110	49.10
$\texttt{DFS}_{\texttt{primal}}$ minTree	23.87	0.232	0.821	0.437	29.13	0.408	1.021	52.62
$\texttt{DFS}_{\texttt{primal}}$ SDSL	58.25	0.299	2.056	1.189	60.96	0.466	2.121	51.59
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min}}$	49.91	0.266	1.727	0.986	53.55	0.422	1.827	50.78
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{min+v5}}$	52.18	0.276	1.769	1.014	56.94	0.448	1.891	47.27
$\texttt{DFS}_{\texttt{primal}}$ $\texttt{SDSL}_{\texttt{v5}}$	60.88	0.312	2.113	1.220	64.65	0.486	2.207	48.07
$\texttt{DFS}_{\texttt{primal}}$ Succinct	37.39	0.340	1.399	0.719	46.79	0.707	1.782	48.22
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{minTree}_{\texttt{v5}}$	24.57	0.233	0.784	0.415	31.01	0.417	1.017	49.10
$\texttt{BFS}_{\texttt{dual}}$ minTree	21.52	0.217	0.710	0.375	27.20	0.389	0.903	52.62
$\texttt{BFS}_{\texttt{dual}}$ SDSL	32.40	0.178	0.964	0.537	38.02	0.321	1.132	51.60
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{min}}$	31.02	0.173	0.911	0.503	36.88	0.307	1.079	50.80
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{min+v5}}$	34.02	0.185	0.967	0.537	40.70	0.330	1.140	47.28
$\texttt{BFS}_{\texttt{dual}}$ $\texttt{SDSL}_{\texttt{v5}}$	35.42	0.191	1.025	0.570	41.77	0.331	1.203	48.09
$\texttt{BFS}_{\texttt{dual}}$ Succinct	35.06	0.327	1.292	0.657	44.43	0.686	1.666	48.22
DegHeur $\texttt{minTree}_{\texttt{v5}}$	24.50	0.231	0.771	0.410	31.05	0.414	0.978	49.10
DegHeur minTree	21.65	0.219	0.710	0.375	27.22	0.391	0.906	52.62
DegHeur SDSL	29.12	0.164	0.892	0.486	36.58	0.320	1.141	51.60
DegHeur $\texttt{SDSL}_{\texttt{min}}$	27.77	0.161	0.834	0.455	34.43	0.307	1.062	50.80
DegHeur $\texttt{SDSL}_{\texttt{min+v5}}$	30.50	0.173	0.894	0.488	38.48	0.329	1.140	47.28
DegHeur $\texttt{SDSL}_{\texttt{v5}}$	32.30	0.175	0.951	0.519	40.68	0.343	1.209	48.09
DegHeur Succinct	35.04	0.328	1.295	0.658	44.42	0.690	1.669	48.22

Tabla 5: Resultados de los experimentos para el *dataset* PE25M. El formato de la tabla es el mismo al usado en la Tabla 4.



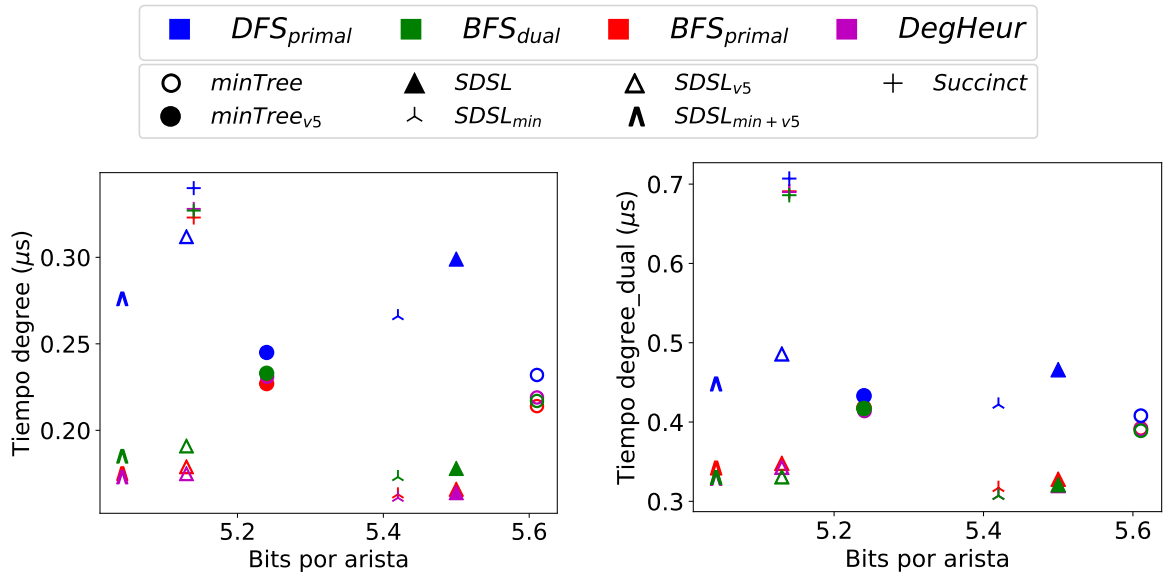
(a) Degree sobre el grafo primal

(b) Degree sobre el grafo dual

(c) DFS sobre el grafo primal

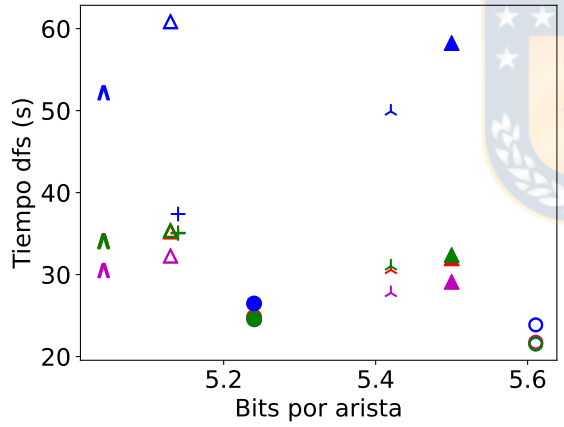
(d) Neighbors sobre el grafo dual

Figura 11: Gráficas de *trade–off* entre tiempo y memoria para las consultas *degree* (11a), *degree_dual* (11b), *dfs* (11c) y *neighbors_dual* (11d) en el dataset `tiger_map`. El baseline corresponde a $\texttt{DFS}_{\texttt{primal}}$ `SDSL`, que es mostrado como un triángulo completamente pintado azul.

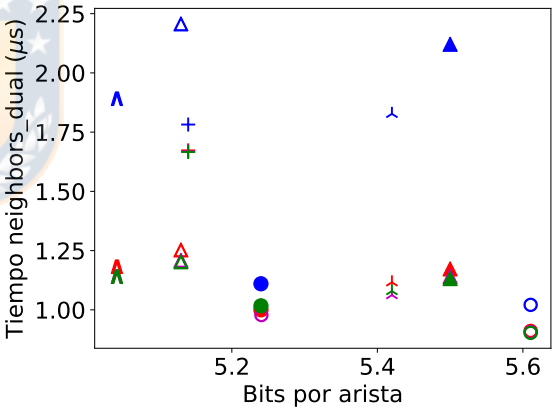


(a) Degree sobre el grafo primal

(b) Degree sobre el grafo dual



(c) DFS sobre el grafo primal



(d) Neighbors sobre el grafo dual

Figura 12: Gráficas de *trade–off* entre tiempo y memoria para las consultas *degree* (12a), *degree_dual* (12b), *dfs* (12c) y *neighbors_dual* (12d) en el *dataset* `PE25M`. El baseline corresponde a $\texttt{DFS}_{\texttt{primal}}$ `SDSL`, que es mostrado como un triángulo completamente pintado azul.

5. Conclusiones y trabajo futuro

En esta tesis se muestra que la topología de los *spanning trees* usados en la representación compacta de grafos planares tiene un impacto en los tiempos de consulta. Esto es consecuencia de que los árboles de menor altura tienen menor distancia pro-

medio entre pares de paréntesis complementarios que árboles de mayor altura, lo cual puede ser aprovechado en representaciones de secuencias de paréntesis balanceados, tales como el *range min-max tree* y el *range min tree*. En base a esto, se generan múltiples métodos de construcción de los *spanning trees* usados en la representación de Ferres *et al.* [11], junto con múltiples variantes de esta estructura de datos, variando las implementaciones utilizadas para las estructuras de datos subyacentes. En general, los métodos propuestos pueden reducir hasta en un 8 % el uso de memoria, mientras que por el lado de los tiempos de ejcución, alcanzamos ser hasta 2.7 y 2.3 veces más rápidos que el *baseline* para los recorrido DFS sobre el grafo primal y grafo dual respectivamente, 1.85, 2.96 y 3.28 veces más rápido para las consultas *degree*, *neighbors* y *face* sobre el primal respectivamente, y finalmente 1.52 y 2.35 veces más rápidos que el baseline para las consultas *degree* y *neighbors* sobre el dual.

Por lo general, las consultas sobre el grafo primal, es decir sobre los vértices, son más rápidas cuando los *spanning trees* se obtienen con un recorrido en anchura sobre el grafo primal. Asimismo, las consultas sobre el grafo dual, es decir sobre las caras, suelen ser más rápidas cuando los *spanning trees* se obtienen con un recorrido en anchura sobre el grafo dual.

Podría ser interesante en un futuro buscar una manera de encontrar un punto de balance entre las alturas de los *spanning trees* sobre el grafo primal y grafo dual. Si bien los métodos heurísticos propuestos en este trabajo no consiguen este objetivo, el problema puede ser formalizado y analizado desde el punto de vista de la optimización. Asimismo, también se podría analizar la construcción de *spanning trees* que en lugar de mejorar los tiempos promedios de consulta, enfoquen sus mejoras hacia un conjunto de vértices o caras en particular. Para esto, un punto de inicio sería generar los *spanning trees* de modo que los vértices o caras de interés queden codificadas en las hojas del árbol respectivo.

Otra posible línea de investigación, sería adaptar esta representación para grafos planares con ciertas características en particular. Un ejemplo podría ser para grafos planares que tengan algún camino hamiltoniano que comience en algún vértice incidente a la cara externa. En este caso, se podría utilizar este camino hamiltoniano como el *spanning tree* sobre el primal. En este caso, bastaría con que la representación almacene únicamente el entrelazado entre los árboles (el *bitvector* A) y el *spanning tree* sobre el dual, dado que la representación del *spanning tree* como secuencia de paréntesis balanceados está completamente definida por su topología (una lista de nodos). Esto permitiría, por un lado, reducir el espacio considerablemente y, por otro lado, potencialmente también se reducirían tiempos de consulta, pues las operaciones sobre una de las secuencias de paréntesis queda definida aritméticamente.

Referencias

[1] Sandra Álvarez-García, Nieves Brisaboa, Javier D. Fernández, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems*, 44(2):439–474, August 2014.

[2] D. Arroyuelo, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto. Worst-case optimal graph joins in almost no space. In *ACM International Conference on Management of Data (SIGMOD)*, pages 102–114, 2021.

[3] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97. SIAM, 2010.

[4] Saurabh Bagchi, Muhammad-Bilal Siddiqui, Paul Wood, and Heng Zhang. Dependability in edge computing. *Communications of the ACM*, 63(1):58–66, 2019.

[5] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[6] Frank Bernhart and Paul C Kainen. The book thickness of a graph. *Journal of Combinatorial Theory, Series B*, 27(3):320–331, 1979.

[7] Guy E Blelloch and Arash Farzan. Succinct representations of separable graphs. In *Annual Symposium on Combinatorial Pattern Matching*, pages 138–150. Springer, 2010.

[8] Dustin Cobas, Veli Mäkinen, and Massimiliano Rossi. Tailoring r-index for document listing towards metagenomics applications. In *International Symposium on String Processing and Information Retrieval*, pages 291–306. Springer, 2020.

[9] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[10] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[11] Leo Ferres, José Fuentes-Sepúlveda, Travis Gagie, Meng He, and Gonzalo Navarro. Fast and compact planar embeddings. *Computational Geometry*, page 101630, 2020.

[12] José Fuentes-Sepúlveda, Diego Gatica, Gonzalo Navarro, M Andrea Rodríguez, and Diego Seco. Compact representation of spatial hierarchies and topological relationships. In *2021 Data Compression Conference (DCC)*, pages 113–122. IEEE, 2021.

[13] José Fuentes-Sepúlveda, Gonzalo Navarro, and Diego Seco. Implementing the topological model succinctly. In *International Symposium on String Processing and Information Retrieval*, pages 499–512. Springer, 2019.

[14] Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[15] 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.

[16] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

[17] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[18] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *Journal of Experimental Algorithmics (JEA)*, 19:1–1, 2015.

[19] Guy Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554, Washington, DC, USA, 1989. IEEE Computer Society.

[20] Guy Joseph Jacobson. Succinct static data structures. 1988.

[21] Kenneth Keeler and Jeffery Westbrook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58(3):239–252, 1995.

[22] Donald E Knuth. The art of computer programming, volume 3: Searching and sorting. *Addison-Westley Publishing Company: Reading, MA*, 1973.

[23] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[24] G. Navarro and K. Sadakane. Compressed tree representations. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 397–401. Springer, 2nd edition, 2016.

[25] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[26] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *International Symposium on Experimental Algorithms*, pages 295–306. Springer, 2012.

[27] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):1–39, 2014.

[28] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70. SIAM, 2007.

[29] Giuseppe Ottaviano. Succinct. `https://github.com/ot/succinct`. Last accessed: May 25, 2021.

[30] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43–es, 2007.

[31] T. R. Riley and W. P. Thurston. The absence of efficient dual pairs of spanning trees in planar graphs. *The Electronic Journal of Combinatorics*, 13(1), August 2006.

[32] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[33] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

[34] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[35] William Thomas Tutte. A census of planar maps. *Canadian Journal of Mathematics*, 15:249–271, 1963.