UNIVERSIDAD DE CONCEPCIÓN
DEPARTAMENTO INGENIERÍA INFORMÁTICA Y
CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA

Explorando el efecto de la topología en representaciones compactas de grafos planares

por
Alexander Nicolás Irribarra Cortés

Memoria presentada para la obtención del título de
Ingeniero Civil Informático

Patrocinante: Diego Seco Naveiras
Co-patrocinante: José Sebastián Fuentes Sepúlveda

Concepción, 1 de Octubre de 2020

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

Para mis padres,
a Rufina y Gustavo.
Gracias totales.

Resumen

El campo de las estructuras de datos compactas ha ganado importancia en los últimos años no solo por permitir almacenar y operar con una mayor cantidad de datos, sino también por permitir hacerlo en dispositivos con memoria limitada. Un ejemplo de esto es la representación de Ferres *et al.* para *planar embeddings*. En esta memoria de título se explora el efecto que tiene en la eficiencia de las consultas la topología de los árboles usados en la construcción de dicha estructura de datos. Los resultados obtenidos en la evaluación experimental muestran que se puede mejorar la velocidad de las operaciones entre un 20 % y un 50 % respecto a la propuesta original al usar árboles de menor altura.

Índice de cuadros

1. Caracterización de los datasets usados en los experimentos 17
2. Caracterización de los métodos implementados para el análisis experimental. 17
3. Tiempos promedio de consulta. 20
4. Reducción del tiempo respecto al baseline. 21
5. Tiempos promedio de construcción 23

Índice de figuras

1. Ejemplo de la descomposición de Turán 2
2. Ejemplo de RMM-TREE y consultas sobre él. 8
3. Árboles codificados por la representación de Ferres *et al.* 10
4. Consultas sobre un RMM-TREE 12
5. Comparación de distancias entre pares de paréntesis para dos árboles similares 13
6. Árboles sobre el primal y dual obtenidos con BFS sobre el primal y dual. 16
7. Árboles sobre el primal y dual obtenidos con heurísticas. 17
8. Histogramas de frecuencia acumulada sobre las distancias. 18
9. Gráficas de distancia promedio vs tiempo de consulta 22

Índice

Resumen I

Índice de tablas II

Índice de figuras III

1. Introducción 1

2. Revisión bibliográfica 5

3. Método desarrollado 11

4. Experimentos y resultados 16

5. Conclusiones y trabajo futuro 22

Referencias 24

A. Anexo 26

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 exponencialmente e incluso a tasas mayores que la tecnología, lo que ha motivado el surgimiento de técnicas de procesamiento y almacenamiento de datos más sofisticadas. Una de las soluciones existentes es a través de las estructuras de datos compactas [15], 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* [5], 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, bioinformática, recuperación de información, por nombrar algunos ejemplos.

Reducir el espacio tiene como consecuencia 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 hasta lograr operaciones más rápidas que con métodos convencionales, considerando las limitaciones actuales de hardware. Los ejemplos más satisfactorios se han logrado al ser capaces de situar en memoria principal, estructuras que antes sólo se podían ejecutar en disco, dada la diferencia de órdenes de magnitud en tiempo de acceso que existen entre ambos niveles de la jerarquía.

Una aplicación de estructuras de datos compactas, y en lo que se propone trabajar, es en la representación de *planar embeddings*, los que se generan al tomar un grafo planar y darle un orden a las aristas de manera que estas no se crucen. Notar que dado un grafo planar, este puede tener más de un *embedding*. Algunas aplicaciones son la representación de mapas y diseño de circuitos. Lo que se propone es utilizar como base el trabajo propuesto por Ferres *et al.* [6], el que se explicará en detalle durante la revisión bibliográfica. A modo introductorio, esta representación expande el trabajo de Turán [22], en el que originalmente se presenta una codificación obtenida en base a un árbol de cobertura T del grafo planar, el que induce un árbol 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 utiliza $4m$ bits, donde m es la cantidad de aristas en el grafo. En la Figura 1 se muestra un ejemplo de la representación.

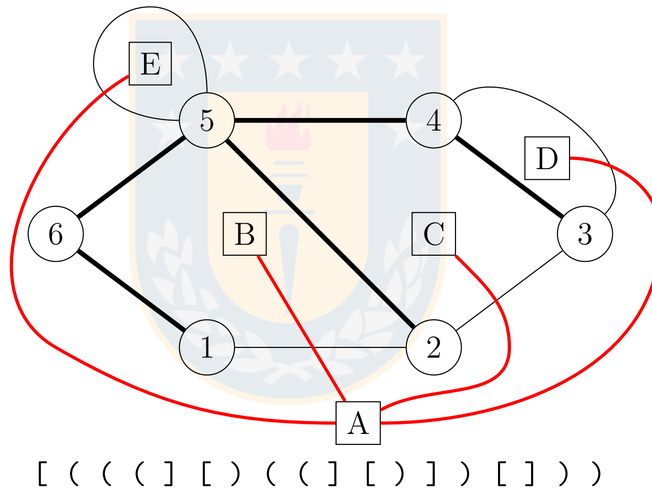


Figura 1: Ejemplo de la descomposición hecha en la representación partiendo desde la arista (1,2). En negrita está el árbol T y en rojo el árbol T'. En la parte inferior se muestra la secuencia de símbolos correspondiente a la codificación de Turán.

Originalmente, Turán solo presenta cómo se obtiene esta codificación y demuestra 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.* [6] 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 3 cadenas de bits A, B y B^*, donde A clasifica a las aristas de acuerdo a si pertenecen a T ó T'; lo que indica cómo se entrelazan estos dos árboles, B y B^* corresponden a T y T' respectivamente codificados como secuencias de paréntesis balanceados [14]. Estas tres secuencias se almacenan utilizando estructuras de datos compactas [15], 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* [17], 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 tiempos

logarítmicos con respecto a la cantidad de nodos, esto almacenando valores mínimos y máximos del exceso o altura asociados a cada 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 complementarios [15], 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 en que se basa este trabajo existe la libertad de utilizar cualquier árbol de cobertura del grafo planar, por lo que se pueden utilizar árboles de menor altura, como puede ser el obtenido con un recorrido BFS sobre este grafo. El inconveniente de esto es que dado que 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 conlleva 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.

El objetivo general de esta memoria de título es estudiar el efecto que tiene la topología del árbol T utilizado en la construcción sobre los tiempos de consulta de una representación compacta de *planar embeddings*. Para alcanzar esta meta, se definen los siguientes objetivos específicos:

1. Definir conjuntos de datos con los que se trabajará para el análisis experimental.
2. Obtener estadísticos relevantes al problema, como distancia promedio entre pares de paréntesis tanto para el árbol de cobertura sobre el primal como el árbol de cobertura sobre el dual.
3. Modificar la implementación existente para que se construya sobre un árbol obtenido con un BFS sobre el primal y el dual, además de variantes de DFS con altura limitada.
4. Medir los tiempos de cada consulta en las variantes.
5. Comparar las soluciones y discutir los resultados obtenidos.

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* y pertinentes a este trabajo en particular, para así continuar con las soluciones existentes a este problema. Luego, en la sección 3 se plantea la hipótesis de este trabajo y se detalla el razonamiento detrás de ella junto a los resultados que se esperan. También en esta sección se explican los cambios propuestos en la construcción de la representación de

Ferres *et al.* En la sección 4 se explica cómo se implementaron los métodos propuestos en la sección 3 y los experimentos realizados para verificar la hipótesis, junto a sus resultados y análisis de los mismos. Finalmente, en la sección 5 se resume el trabajo realizado y las conclusiones que se obtienen del mismo, para luego dar paso al trabajo futuro.

2. Revisión bibliográfica

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 [21] 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 posibilidad $p(x_i)$ de ocurrir, la entropía $H(X)$ de la fuente de información se calcula como:

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

Este valor nos indica el largo promedio que tiene el código asociado a cada elemento de U si es que se codifica de manera óptima. 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:

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

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 [11]. Por ejemplo, las estructuras de datos sucintas son una categoría particular de estructuras de datos compactas donde se utilizan $Z + o(Z)$ bits.

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 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 [23], de manera que la entropía de peor caso de los *planar embeddings* es $m \log 12 \approx 3{,}58m$ [15]. 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 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.

Varias estructuras de datos compactas para la representación de objetos más complejos – *como es el caso de los embeddings planares* – 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.

Un bitvector [15] 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.
- $select_v(X, v)$**:** retorna la posición del i-ésimo bit con valor v.

Existen representaciones que dan soporte a estas operaciones tanto en representaciones no compactas [10, 9, 18] – *que no están comprimidas a su entropía* – como representaciones compactas [19, 16, 8].

Con respecto a árboles sucintos, estos se codifican como secuencias de paréntesis balanceados [14]. 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 (' corresponden a bits con valor 1 y los símbolos ')' corresponden a bits con valor 0 o viceversa. Las principales soluciones existentes son LOUDS [10, 2], DFUDS [2] y Fully Functional [17]. 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 [1].

Tomando como foco la representación que utiliza el 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 $P[i] = \texttt{(}$, $excess(i) = excess(i-1) - 1$ si $P[i] = \texttt{)}$, y para $i = 0, excess(-1) = 0$. Además, se divide la secuencia P 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 aglomeran 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 aglomerarlos 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:

- $v.e = excess(e) - excess(s-1)$

- $v.min = \min_{i \in [s,e]} excess(i) - excess(s-1)$
- $v.max = \max_{i \in [s,e]} excess(i) - excess(s-1)$

En la Figura 2 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 es así, se comienza a bajar desde el hermano u, en caso contrario se hace la actualización $d' \leftarrow d' + u.e$ y se sigue subiendo por el padre hasta encontrar un nodo izquierdo cuyo hermano derecho cumpla la condición.

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 $d \geq d' + v.m$. 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 2 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.

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

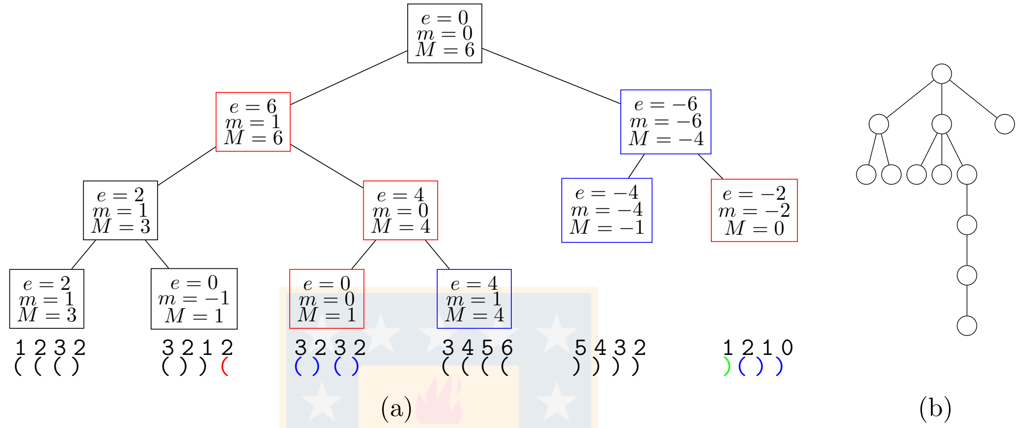


Figura 2: En 2a se muestra el RMM-TREE correspondiente al árbol en 2b usando un tamaño de bloque $b = 4$, junto a la secuencia de paréntesis codificada y los excesos. Se ejemplifica además la consulta $match(7) = fwd_search(7, -1)$ de la siguiente manera: En la parte inferior se indica en rojo el paréntesis 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.

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.

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

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

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

Retomando el caso particular de grafos y *planar embeddings*, existen diversas soluciones cada una con sus ventajas y desventajas. Turán [22] 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 árbol de cobertura del grafo. Esta representación utiliza un espacio cercano al óptimo, sin embargo no soporta operaciones de navegación por si sola. Luego, Jacobson [10] presenta una codificación para grafos planares basada en *book embeddings* [3] que soporta operaciones utilizando una cantidad de bits lineal a la cantidad de nodos. Keeler y Westbrook [12] presentan una codificación en espacio óptimo, pero sin soporte para operaciones. Munro y Raman [13] 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 [4] presentan una representación de espacio óptimo basado en técnicas utilizadas para grafos separables [4], sin embargo no ha habido éxito en implementarla por lo que su valor es más teórico que práctico. Ferres *et al.* [6] 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 [22], la cual corresponde a una codificación de largo $2m$ compuesta de paréntesis y corchetes. Primero se obtiene cualquier árbol de cobertura T del grafo G, cuyo complemento T' se corresponde con un árbol de cobertura sobre el grafo dual [20]. 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.* [6] corresponde a, en primer lugar, separar la secuencia en 3 cadenas: un bitvector A donde $A[i] = 1$ si y solo si la i-ésima arista visitada en el recorrido pertenece a T, una secuencia de paréntesis balanceados B correspondiente a los paréntesis redondos y una secuencia de paréntesis balanceados B^* correspondiente a los paréntesis cuadrados. 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 3 se muestran los árboles codificados para el *embedding* en la Figura 1.

Las secuencias se almacenan utilizando estructuras compactas, de manera que además se da soporte a operaciones como las mencionadas previamente en este capítu-

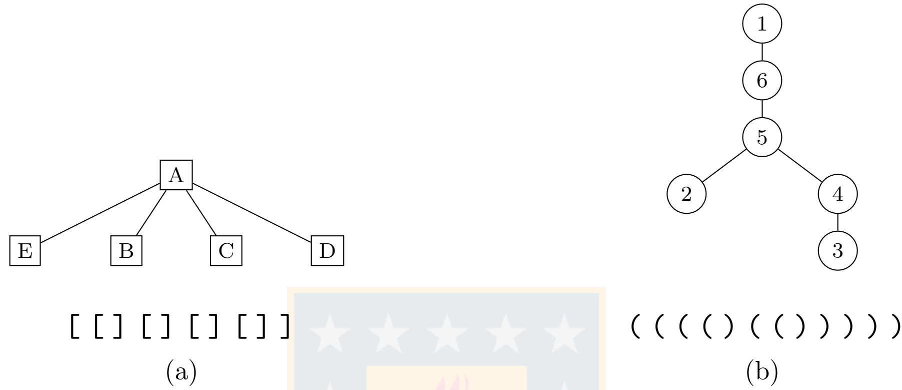


Figura 3: Los árboles codificados usando la representación de Ferres *et al.* para el *embedding* de la Figura 1. 3a corresponde al árbol sobre el dual, mientras que 3b corresponde al árbol sobre el primal.

lo. 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.
$$first(v) = A.select_1(B.select_0(v-1)) + 1$$

- $last(v)$: retorna la posición i en A de la última arista visitada al procesar el nodo v durante la construcción.
$$last(v) = A.select_1(B.match(B.select_0(v-1))$$

- $mate(i)$: 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.

Ferres *et al.* [6] utiliza un DFS para obtener el árbol de cobertura usado en la construcción. Sin embargo, esto tiene un efecto negativo en los tiempos de las consultas, ya que esta clase de árbol tiene una altura mucho mayor que la obtenida, por ejemplo, por un recorrido BFS. Los tiempos se ven afectados 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. Método desarrollado

La hipótesis propuesta en este trabajo es que la topología del *spanning tree* utilizado en la construcción de la representación de Ferres *et al.* tiene un impacto en la velocidad de las consultas sobre el *planar embedding*, siendo posible así obtener una estructura de datos más rápida tras reemplazar el método de construcción.

El comportamiento esperado es que al usar *spanning trees* de menor altura la velocidad de consulta sobre el *planar embedding* tienda a subir. Esto tiene directa relación con la forma en que funciona en RMM-TREE. De manera más general, se puede observar el efecto que tiene la distancia entre pares de paréntesis al realizar consultas en el RMM-TREE, recordando que en el RMM-TREE un nodo del árbol se representa por un par de paréntesis. Se puede ver por ejemplo, el efecto de la distancia entre pares de paréntesis para la consulta $match(i)$ con resultado j. El caso más simple corresponde cuando la posición j está en el primer bloque a revisar, pues se evita recorrer el RMM-TREE. En el caso de tener que recorrerlo, a modo de intuición se puede notar que al subir se va verificando si j está contenido en un intervalo, de manera que en cada nodo visitado se va ampliando hacia la derecha el intervalo total visitado. Debido a esto, mientras menor sea la distancia entre i y j, menor será la cantidad de nodos visitados. Esto se ve ejemplificado en la Figura 4.

Dado que la representación de Ferres *et al.* utiliza principalmente la operación *match*, es que resulta deseable que la distancia entre pares de paréntesis sea la menor posible, ya que mejoras en esta primitiva causan mejoras en los tiempos de las operaciones más complejas que la utilizan. Se puede notar que en una representación de árbol como secuencia de paréntesis balanceados, cada subárbol queda representado en una subsecuencia. Esto significa que el subárbol que contiene a un nodo v y sus descendientes está representado en la subsecuencia que inicia en el paréntesis que abre de v y termina en el paréntesis que le cierra. De esta manera se puede ver el efecto que tiene la ubicación de un nodo en la distancia entre los pares de paréntesis correspondientes a sus ancestros. Este efecto es el aumento en dos posiciones de la distancia asociada a cada uno de los ancestros. Esto se ejemplifica en la Figura 4. En

Figura 4: Resumen de los recorridos sobre el RMM-TREE para las consultas $match(0)$ (en rojo) y $match(10)$ (en azul). Se puede apreciar que en la consulta $match(0)$ es necesario visitar más nodos.

consecuencia, se tienen menores tiempos de consulta en promedio cuando hay una mayor concentración de nodos cerca de la raíz, o en otras palabras, cuando la altura es menor.

No hay que olvidar que en la representación de Ferres *et al.* se codifican dos árboles T y T', cuyas topologías son dependientes entre sí. Además, T se ocupa principalmente para responder consultas sobre los vértices del *embedding*, mientras que T' se utiliza principalmente para responder consultas sobre las caras. También T tiene la restricción de que su raíz debe ser incidente en la cara externa, mientras que para T' su raíz debe ser la cara externa. Dada la interdependencia entre las topologías de T y T', es que no siempre es posible mejorar las distancias de uno de los árboles sin empeorar las distancias en el otro. De manera inicial se explican las técnicas enfocadas a mejorar la topología de T, sin embargo, se destaca que en los resultados experimentales se mejora considerablemente tanto T como T' con respecto a la construcción original.

Si nos enfocamos en realizar mejoras a T, podemos ver que tras haber escogido una raíz u se quisiera ubicar tantos nodos en el siguiente nivel como sea posible, ya que como se explicó previamente hay una relación entre la altura del árbol y las distancias promedio. Naturalmente, solo se pueden agregar a ese nivel los vecinos de u, y así mismo para cada nivel $i > 0$ solo se pueden agregar nodos que sean vecinos a aquellos en el nivel $i-1$. Sea V_i el conjunto de nodos ubicados en el i-ésimo nivel, con $V_0 = \{u\}$ y un operador booleano binario $vec(v, w)$ que retorne verdadero si y solo

```
    ( ( ( ) ( ( ) ) ) ( ) )
ID: 0 1 2 2 3 4 4 3 1 5 5 0
```

(a)

```
    ( ( ( ) ( ) ) ( ) ( ) )
ID: 0 1 2 2 3 3 1 4 4 5 5 0
```

(b)

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

si los nodos v y w son vecinos, en términos formales esto se expresa de la siguiente manera:

$$V_i = \{v : (\exists w \in V_{i-1} : vec(v,w)) \wedge (\nexists w \in V_j : vec(v,w) \wedge j < i-1), i > 0\}$$

Esta distribución de nodos se puede conseguir realizando un recorrido en anchura (BFS) desde la raíz u. Como consecuencia de esto también se tiene que es la distribución donde los nodos se ubican lo más cerca posible de la raíz u. Si bien existe más de una opción posible para la raíz u, los resultados experimentales indican que no existe un cambio considerable al escoger una sobre otra. El algoritmo de construcción se presenta en el Algoritmo 1.

Por otro lado, si se quisiera que las mejoras estén enfocadas en T', se puede utilizar la misma estrategia de realizar un recorrido BFS, pero ahora sobre el grafo dual iniciado en la cara externa. Para esto se debe previamente computar la cara que cada arista delimita en sus dos sentidos como se describe en el Algoritmo 2 (en Anexo A), y así se puede proseguir con el BFS sobre el dual como se describe en el Algoritmo 3 (en Anexo A).

Con estas dos construcciones es posible minimizar las distancias en T o T' respectivamente, lo que reduce las consultas sobre el grafo primal o sobre el grafo dual respectivamente. Con el fin de tener construcciones donde las alturas en T y T' no sean necesariamente óptimas, pero que estén más balanceadas, se crean las siguientes heurísticas:

Algorithm 1 BFS sobre el primal

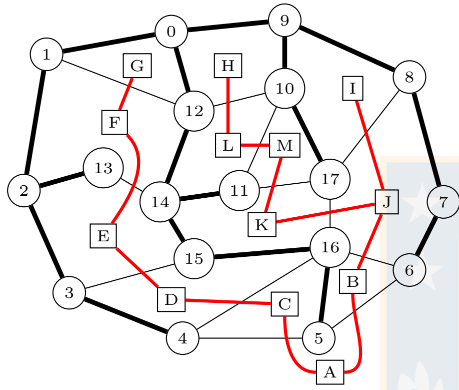
```
 1: procedure bfsPrimal(o, G)   ▷ Retorna el spanning tree con BFS iniciado en o
 2:     T ← Una lista de aristas inicialmente vacía
 3:     q ← Una cola de vértices vacía
 4:     V ← Un arreglo booleano de tamaño G.vertices iniciado en false
 5:     q.push(o)
 6:     V[o] ← true
 7:     while q no está vacio do
 8:         u ← q.pop()
 9:         for cada vecino v de u do
10:             if V[v] = false then
11:                 V[v] = true
12:                 Agregar aristas (v, u) y (u, v) a T
13:                 q.push(v)
14:             end if
15:         end for
16:     end while
17:     return T
18: end procedure
```

- **DFS limitado:** Sea h un parámetro que limite la altura, se obtiene un árbol de cobertura realizando un recorrido en profundidad (DFS) desde la raíz u en el que no se agrega ningún nodo con distancia desde la raíz mayor a h. Si al final de este DFS quedan vértices sin procesar, se continúa con un BFS multi-origen desde los nodos con altura h para así reducir la diferencia entre la altura obtenida h' y h, agregando los nodos restantes. Esto se describe en detalle en el Algoritmo 4 (en Anexo A).
- **BFS alternado:** En esta construcción primero se expande un nivel de T desde la raíz u, luego se expande un nivel de T' desde la cara externa y así sucesivamente se alterna la expansión de niveles entre T y T'. Se tiene cuidado de que en ningún momento se agregue a T' una arista perteneciente a T ni viceversa. Esto se describe en detalle en el Algoritmo 5 (en Anexo A).

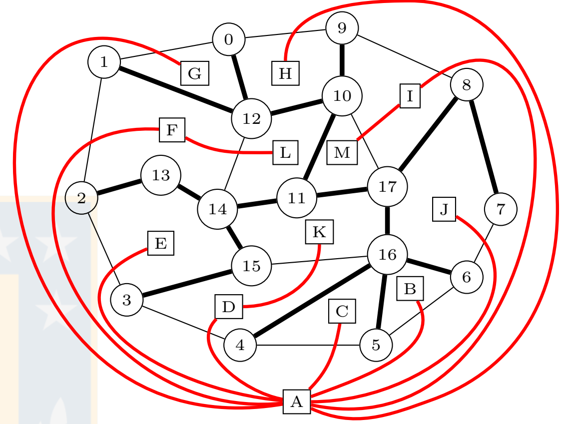
La intuición detrás del DFS limitado es que se fuerza a que T aumente su altura de manera controlada, de manera que T' potencialmente pueda disminuir su altura. Respecto al BFS alternado, la intuición es que alternar la generación de niveles entre los dos árboles podría permitir que cada uno de los árboles mantenga altura baja sin perjudicar mayormente a la altura del otro.

Algorithm 2 Marcado de aristas

procedure MARCARARISTAS(G)
  $M \leftarrow$ Un arreglo de tamaño G.edges*2, inicializado en -1
  $F \leftarrow$ Un arreglo dinámico inicialmente vacio
  $currFaceId \leftarrow 0$
  for cada arista e de G **do** ▷ Cada arista (u,v) aparece duplicada como (v,u)
    if $M[e]$ = -1 **then** ▷ Se procede a recorrer la cara en sentido anti-horario
      $visiting \leftarrow e$
      $F.push_back(e)$
      repeat
        $M[visiting] \leftarrow currFaceId$
        $visCmp \leftarrow cmp(visiting)$ ▷ $cmp(u, v) = (u, v)$
        $visiting \leftarrow nextCCW(visCmp)$ ▷ $nextCCW$ retorna la siguiente arista en sentido anti-horario
      until $visiting = e$
      $currFaceId \leftarrow currFaceId + 1$
    end if
  end for
  return $< M, F >$ ▷ M es tal que $M[e]$ es la cara correspondiente a la arista e, F es tal que $F[i]$ tiene una arista representativa de la i-ésima cara descubierta.
end procedure



(a) BFS sobre el primal



(b) BFS sobre el dual

Figura 6: Los árboles T (en negrita) y T' (en rojo) obtenidos variando la construcción. En ambos casos la raíz de T es 0 y la raíz de T' es A. En la Figura 6a la altura de T es 6 y la de T' es 7, mientras que en la Figura 6b la altura de T es 7 y la de T' es 3.

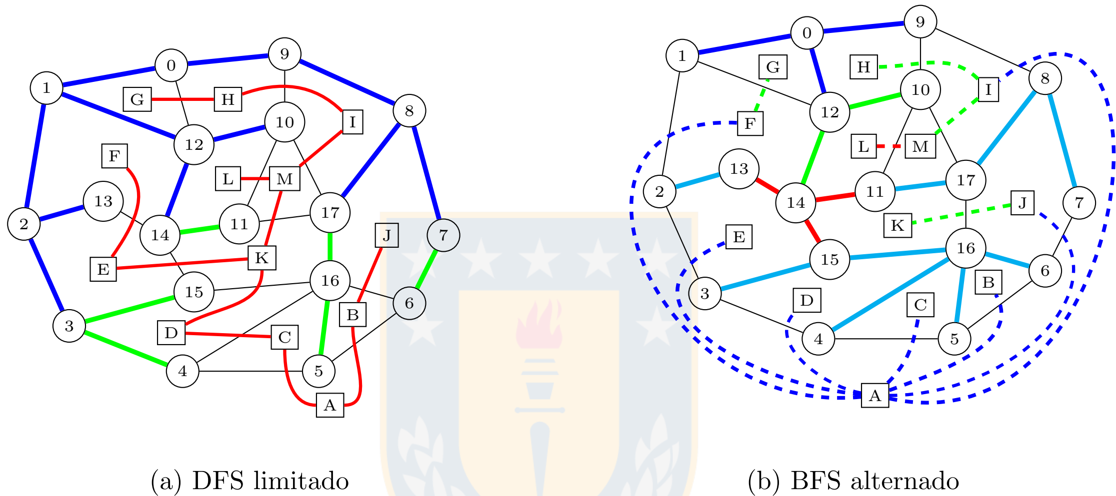
4. Experimentos y resultados

La implementación se realizó a través de un *fork* al repositorio puesto a disposición por Ferres *et al.* y modificado para que utilice los métodos de construcción propuestos. Esta implementación utiliza la biblioteca SDSL desarrollada por Simon Gog [7]. Los experimentos se realizaron en una máquina con procesador Intel® Core™ i7-3820 @ 3.60GHz y con 32 GB de memoria RAM. Cada uno de los 4 núcleos del procesador tiene 32 kB de caché L1 y 256 kB de caché L2. Los núcleos comparten 10 MB de caché L3. Sobre esta máquina se ejecuta GNU/Linux 3.13.0-86-generic.

En los experimentos se utilizan datasets obtenidos en `http://www.inf.udec.cl/~jfuentess/datasets/graphs.php`, los que se caracterizan en la Tabla 1. El dataset *tiger_map* se obtiene del mapa de Estados Unidos, mientras que el resto de los datasets se obtienen con triangulaciones de puntos generados al azar.

Los experimentos se realizan sobre los métodos presentados en la Tabla 2.

Dado que las distancias entre pares de paréntesis tienen un impacto en los tiempos de consulta, resulta importante analizar cómo estas se distribuyen al utilizar distintos algoritmos de construcción. En la Figura 8 se presentan los histogramas de frecuencias relativas acumuladas de las distancias utilizando DFS_{primal}, BFS_{primal} y BFS_{dual} res-



(a) DFS limitado (b) BFS alternado

Figura 7: Los árboles T (en negrita) y T' (en rojo) obtenidos variando la construcción. En ambos casos la raíz de T es 0 y la raíz de T' es A. En 6a la altura de T es 6 y la de T' es 7, mientras que en 6b la altura de T es 7 y la de T' es 3.

Tabla 1: Caracterización de los datasets usados en los experimentos

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: Caracterización de los métodos implementados para el análisis experimental.

Método	Descripción
DFS_{primal}	DFS sobre el primal, corresponde al *baseline* implementado por Ferres *et al.*
BFS_{primal}	BFS sobre el primal
BFS_{dual}	BFS sobre el dual
DFS_{Nx}	DFS acotado sobre el primal con altura mínima de N veces la altura obtenida con BFS
HEUR	BFS alternado entre primal y dual

pectivamente para los datasets *tiger_map* y *PE25M*. Se debe recordar que DFS_{primal} corresponde al *baseline* disponible en el estado del arte.

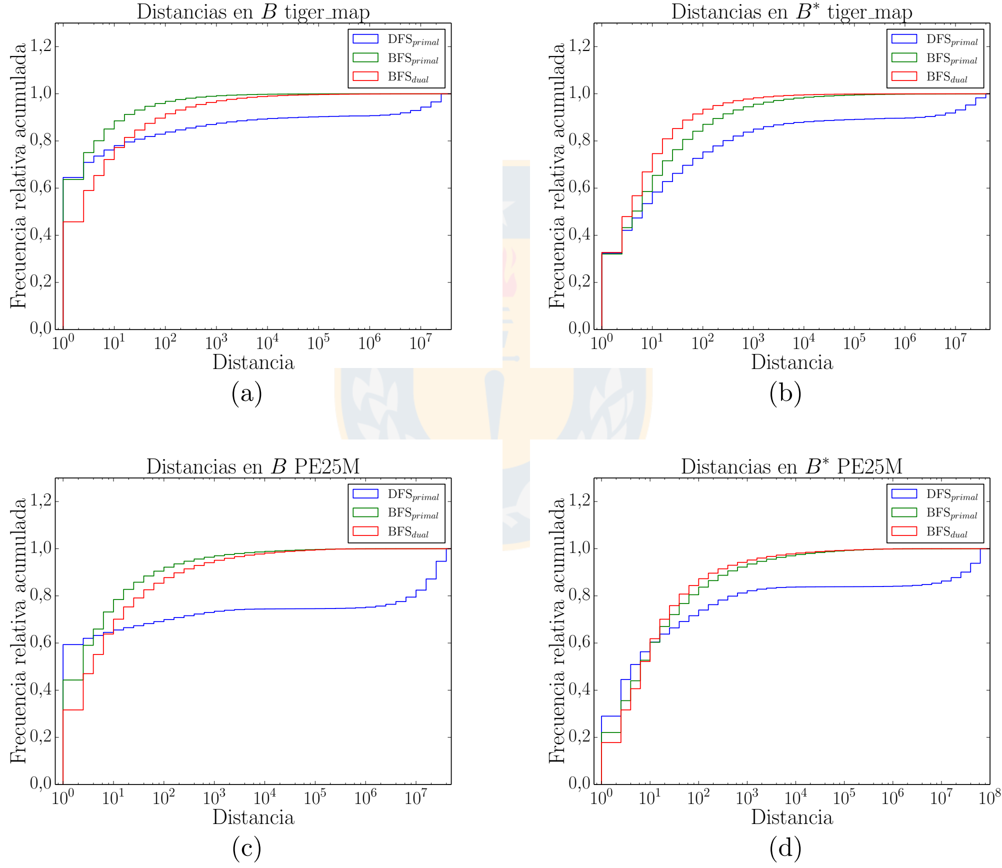


Figura 8: Histogramas de frecuencia relativa acumulada para las distancias en B y B^* utilizando los métodos DFS_{primal}, BFS_{primal} y BFS_{dual}. Los métodos DFS_{Nx} y HEUR se comportan de manera similar al método BFS_{primal}, por lo que se decidió omitirlos. Las Figuras 8a y 8b corresponden al dataset *tiger_map*, mientras que las Figuras 8c y 8d corresponden al dataset *PE25M*.

De la Figura 8 se puede destacar que utilizar BFS_{primal} o BFS_{dual} genera que un mayor porcentaje de nodos tenga menores distancias entre sus paréntesis correspondientes tanto en B como B^*, mientras que por otro lado al usar DFS_{primal} hay un porcentaje no menor de nodos con distancias muy altas. En particular, se puede ver que casi la totalidad de los nodos tiene distancia menor a 10^4 usando BFS_{primal} o BFS_{dual}, mientras que alrededor de un 10 % de los nodos tiene distancia mayor a 10^6 en el dataset *tiger_map* al usar DFS_{primal}, lo que aumenta alrededor del 30 % y 20 % respectivamente para B y B^* en el dataset *PE25M*. También se puede ver que usar

BFS_{primal} reduce ligeramente las distancias en B con respecto a BFS_{dual}, mientras que sucede lo contrario en B^*.

De esta manera, se espera que los tiempos de consulta al construir la estructura de datos utilizando BFS_{primal} o BFS_{dual} sean menores que al utilizar DFS_{primal}. Asimismo también es esperable que al utilizar BFS_{primal} se obtengan los mejores tiempos en consultas sobre el grafo primal, mientras que al utilizar BFS_{dual} se obtengan los mejores tiempos en consultas sobre el grafo dual. Estas hipótesis las validamos a continuación.

Para medir el efecto de cada construcción en el rendimiento de la estructura se miden los tiempos promedio de consultas de realizar un recorrido DFS sobre la representación compacta (*dfs*), obtener el grado de un vértice (*degree*), listar los vecinos de un vértice (*list*) y de recorrer la cara a la que pertenece una arista (*face*). Para las operaciones de grado y vecinos se mide el tiempo total de 30 consultas sobre cada nodo y calcula el promedio. Para la operación de recorrido de cara se hace lo mismo con 30 consultas sobre cada arista. Para el recorrido DFS sobre la representación compacta, se mide el tiempo promedio de realizarlo desde 10 vértices al azar. Todas estas mediciones se repiten 5 veces para el dataset *tiger_map* y 3 veces para el resto (debido a que son triangulaciones) haciendo variar la arista desde la que se inicia la construcción. Recordar que la construcción de la estructura de datos se puede iniciar desde cualquier arista de la cara externa. Los resultados se presentan en la Tabla 3.

		Consultas Primal				Consultas Dual			
Dataset	Método	dfs	degree	list	face	dfs	degree	list	face
PE1M	BFS_{primal}	**1.51**	0.245	**1.411**	**1.469**	1.82	3.638	0.480	1.745
	DFS_{primal}	2.89	0.417	2.841	3.194	3.04	6.729	0.660	2.965
	DFS_{2x}	1.55	0.252	1.452	1.521	1.83	3.638	0.479	1.751
	DFS_{4x}	1.59	0.258	1.495	1.576	1.85	3.704	0.480	1.777
	DFS_{8x}	1.66	0.270	1.566	1.664	1.87	3.732	0.479	1.793
	BFS_{dual}	1.52	0.261	1.465	1.542	**1.79**	**3.504**	**0.458**	**1.675**
	HEUR	**1.51**	**0.244**	1.413	1.470	1.82	3.643	0.492	1.752
PE5M	BFS_{primal}	**7.69**	0.248	1.435	1.494	9.23	3.689	0.486	1.768
	DFS_{primal}	14.43	0.417	2.841	3.194	15.12	6.737	0.660	2.968
	DFS_{2x}	7.91	0.254	1.476	1.550	9.23	3.697	0.483	1.771
	DFS_{4x}	8.47	0.274	1.601	1.706	9.46	3.805	0.484	1.817
	DFS_{8x}	11.10	0.359	2.165	2.423	10.62	4.372	0.495	2.053
	BFS_{dual}	7.75	0.264	1.491	1.573	**9.12**	**3.565**	**0.462**	**1.696**
	HEUR	7.72	**0.247**	**1.434**	**1.493**	9.23	3.701	0.497	1.776

Dataset	Método	Consultas primal				Consultas dual			
		dfs	degree	list	face	dfs	degree	list	face
PE10M	BFS_{primal}	**15.26**	**0.247**	**1.435**	**1.494**	18.43	0.485	1.768	3.686
	DFS_{primal}	29.18	0.421	2.871	3.229	30.55	0.664	2.990	6.803
	DFS_{2x}	16.42	0.263	1.534	1.620	18.74	0.484	1.798	3.758
	DFS_{4x}	20.75	0.338	2.014	2.232	20.52	0.491	1.984	4.199
	DFS_{8x}	24.35	0.393	2.386	2.702	22.17	0.500	2.149	4.604
	BFS_{dual}	15.52	0.265	1.493	1.576	**18.21**	**0.463**	**1.700**	**3.568**
	HEUR	15.34	**0.247**	1.440	1.501	18.54	0.500	1.783	3.713
PE25M	BFS_{primal}	**38.43**	**0.247**	**1.435**	**1.496**	46.17	0.486	1.772	3.698
	DFS_{primal}	73.60	0.424	2.897	3.264	76.97	0.667	3.013	6.851
	DFS_{2x}	40.29	0.257	1.503	1.583	46.90	0.487	1.796	3.775
	DFS_{4x}	49.35	0.316	1.890	2.069	50.90	0.492	1.959	4.110
	DFS_{8x}	59.33	0.382	2.311	2.608	54.89	0.497	2.125	4.535
	BFS_{dual}	39.07	0.264	1.500	1.584	**45.99**	**0.463**	**1.705**	**3.590**
	HEUR	38.50	**0.247**	1.436	1.498	46.64	0.498	1.791	3.727
tiger_map	BFS_{primal}	**20.06**	**0.210**	**0.952**	**1.970**	22.67	0.295	1.107	**3.325**
	DFS_{primal}	34.23	0.299	1.688	3.535	36.79	0.389	1.823	6.301
	DFS_{2x}	21.74	0.225	1.042	2.159	23.80	0.299	1.164	4.913
	DFS_{4x}	22.62	0.231	1.087	2.252	24.45	0.304	1.197	5.075
	DFS_{8x}	24.04	0.244	1.163	2.427	25.39	0.308	1.246	5.308
	BFS_{dual}	21.66	0.238	1.048	2.106	**21.32**	**0.268**	**1.031**	4.129
	HEUR	20.12	**0.210**	**0.952**	**1.970**	22.73	0.301	1.112	3.338

Tabla 3: Tiempos promedio de consulta. 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.

Se puede ver que para todas las consultas sobre todos los datasets el tiempo más alto siempre se obtiene al utilizar el baseline DFS_{primal}, tanto en consultas sobre el grafo primal como sobre el grafo dual. Por otro lado, en las consultas sobre el grafo primal los tiempos más bajos se obtienen al construir con BFS_{primal} o usando el método HEUR, mientras que las consultas sobre el grafo dual se resuelven más rápido con BFS_{dual} excepto para la consulta *faces* en el dataset *tiger_map*, en la que se obtiene el mejor tiempo con BFS_{primal}. El método HEUR no tiene diferencia considerable con BFS_{primal}, mientras que el método de altura acotada no muestra ventajas respecto a los métodos con BFS, al menos para los valores de altura que fueron probados.

Para facilitar la comparación, se presenta en la Tabla 4 un resumen de los porcen-

tajes de mejora con respecto al baseline, que es el método DFS_{primal}.

Tabla 4: Reducción del tiempo obtenida al cambiar la construcción con DFS_{primal} al indicado para cada dataset. Todos los valores corresponden a porcentajes. Se destaca en negrita el método que obtuvo la mayor mejora para cada operación en cada dataset.

		Consultas primal				Consultas dual			
Dataset	Método	dfs	degree	list	face	dfs	degree	list	face
PE1M	BFS_{primal}	**47.8**	**41.2**	**50.3**	**54.0**	40.1	45.9	27.3	41.1
	BFS_{dual}	47.4	37.4	48.4	51.7	**41.1**	**47.9**	**30.6**	**43.5**
PE5M	BFS_{primal}	**46.7**	**40.5**	**49.5**	**53.2**	39.0	45.2	26.4	40.4
	BFS_{dual}	46.3	36.7	47.5	50.8	**39.7**	**47.1**	**30.0**	**42.9**
PE10M	BFS_{primal}	**47.7**	**41.3**	**50.0**	**53.7**	39.7	27.0	40.9	45.8
	BFS_{dual}	46.8	37.1	48.0	51.2	**40.4**	**30.3**	**43.1**	**47.6**
PE25M	BFS_{primal}	**47.8**	**41.7**	**50.5**	**54.2**	40.0	27.1	41.2	46.0
	BFS_{dual}	46.9	37.7	48.2	51.5	**40.2**	**30.6**	**43.4**	**47.6**
tiger_map	BFS_{primal}	**41.4**	**29.8**	**43.6**	**44.3**	38.4	24.2	39.3	**47.2**
	BFS_{dual}	36.7	20.4	37.9	40.4	**42.0**	**31.1**	**43.4**	34.5

Se puede ver que para los datasets utilizados las mejoras obtenidas van desde un 20 % hasta un 54 % utilizando cualquiera de los dos métodos con BFS. Más allá de eso, se puede ver que utilizar BFS_{primal} reduce más los tiempos de consulta sobre el primal, por ejemplo para la consulta *degree* sobre el primal en el dataset *tiger_map* se obtiene un 29.8 % de mejora usando BFS_{primal}, mientras que al usar BFS_{dual} solo es un 20.4 % de mejora. De la misma forma, se obtienen mejores tiempos de consulta sobre el dual al usar BFS_{dual}. También se puede ver que la diferencia entre usar un BFS_{primal} y BFS_{dual} se ve acentuada en el dataset *tiger_map*, lo que da a entender que la topología del *embedding* representado también tiene un efecto en las mejoras que estos métodos pueden traer.

Se analiza también el efecto que tiene la distancia promedio entre pares de paréntesis en B con los tiempos de consulta sobre el grafo primal. En la Figura 9 se grafican estas métricas con los resultados obtenidos para los datasets *tiger_map* y *PE25M* con los métodos propuestos a excepción del construido con DFS. Se puede ver como por lo general el tiempo de consulta crece en conjunto a la distancia promedio.

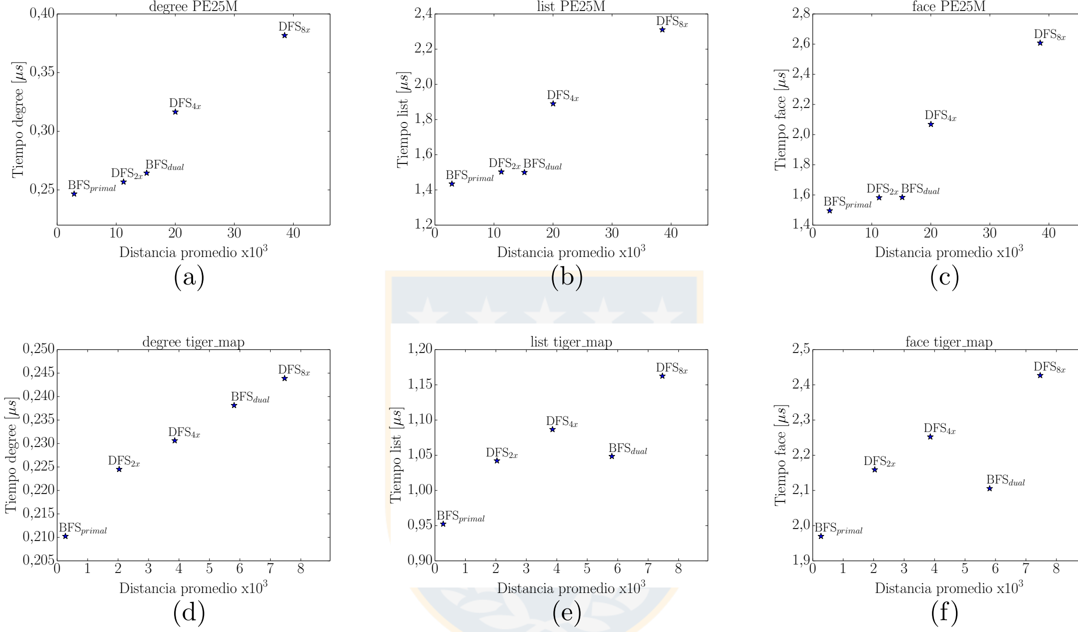


Figura 9: Gráficas de distancia promedio vs tiempo de consulta para los datasets *tiger_map* (Figuras 9d, 9e y 9f) y *PE25M* (Figuras 9a, 9b y 9c), para las consultas *degree*, *neighbors* y *face* respectivamente. No se incluye la medición obtenida con DFS_{primal} ni HEUR.

Adicionalmente, se mide el tiempo de construcción promedio para los dos datasets más grandes. Esto se hace repitiendo la construcción 100 veces variando la arista inicial, de manera que luego se obtienen los promedios desplegados en la Tabla 5. El mejor tiempo es obtenido por BFS_{primal}, que es ligeramente menor al obtenido por DFS_{primal}. Los peores tiempos son obtenidos por BFS_{dual} y HEUR, debido a que en estos métodos es necesario identificar previamente qué caras conecta cada arista.

Respecto al espacio utilizado por la representación, hay un aumento de menos del 0,03 % al utilizar métodos basados en BFS con respecto a la construcción basada en DFS. En particular, este aumento sucede en la estructura auxiliar para dar soporte a operaciones en B y B^*. Esto se podría deber a la pérdida de ciertas regularidades presentes al utilizar DFS, pero de todas maneras se considera algo marginal.

5. Conclusiones y trabajo futuro

En esta memoria de título se estudió el efecto que tiene la topología del árbol utilizado en la construcción de la representación de *planar embeddings* de Ferres *et*

Tabla 5: Tiempos promedio de construcción para los datasets tiger_map (izquierda) y PE25M (derecha). Todos los valores están en segundos.

Método	Tiempo
DFS_{primal}	5.85
BFS_{primal}	5.40
BFS_{dual}	11.36
HEUR	12.78
DFS_{2x}	6.32
DFS_{4x}	6.30
DFS_{8x}	6.22

Método	Tiempo
DFS_{primal}	17.59
BFS_{primal}	15.98
BFS_{dual}	39.74
HEUR	47.28
DFS_{2x}	18.76
DFS_{4x}	18.31
DFS_{8x}	18.43

al. [6] en la velocidad de las consultas, verificando que utilizar árboles de menor altura contribuye a disminuir los tiempos de consulta en promedio entre un 20 % y 54 %. Para esto se reemplazó el uso de un DFS sobre el primal por BFS sobre el primal o el dual. Es posible además mejorar los tiempos de consulta sobre el grafo primal entre un 29 % y 54 % al construir con un BFS en el primal, o mejorar entre un 30 % y 48 % los tiempos de consulta sobre el grafo dual al construir con un BFS sobre el dual. Todo esto se logró con efectos despreciables sobre el espacio utilizado por la estructura.

A pesar de que este trabajo estuvo enfocado en una representación compacta para *planar embeddings*, vale la pena mencionar que cualquier otra representación compacta que utilice por debajo árboles sucintos puede verse acelerada al forzar la construcción de árboles menos profundos, siempre y cuando exista la libertad de escoger algún árbol en particular a codificar.

Como trabajo futuro, dado que en Ferres *et al.* [6] se presenta un algoritmo en paralelo para la construcción, se plantea aplicar una construcción paralela con los métodos propuestos en este trabajo. Esto significaría utilizar algoritmos paralelos eficientes para la realización de BFS e incorporarlo al *pipeline* de la construcción original. Adicionalmente, se plantea explorar otras representaciones compactas basadas en árboles sucintos en las que se puedan realizar las mismas modificaciones o similares, con el fin de mejorar los tiempos de consulta. Otra propuesta es expandir este trabajo a otras representaciones sucintas de árboles, como por ejemplo LOUDS [10].

Referencias

[1] 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.

[2] 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.

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

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] Guy Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554, 1989.

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

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

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

[14] G. Navarro and K. Sadakane. *Compressed Tree Representations*, pages 397–401. Springer, 2nd edition, 2016.

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

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

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

[18] 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.

[19] 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.

[20] TR Riley and William P Thurston. The absence of efficient dual pairs of spanning trees in planar graphs. *arXiv preprint math/0511493*, 2005.

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

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

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

A. Anexo

Algorithm 3 BFS sobre el dual

1: **procedure** BFSDUAL($extEdge, G$)
2: $T' \leftarrow$ Una lista de aristas inicialmente vacía
3: $q \leftarrow$ Una cola de caras vacía
4: $V \leftarrow$ Un arreglo booleano de tamaño $2 + G.edges - G.vertices$ iniciado en $false$
5: $< M, F > \leftarrow marcarAristas(G)$
6: $initFace \leftarrow M[extEdge]$
7: $q.push(initFace)$
8: $V[extFace] \leftarrow true$
9: **while** q no está vacía **do**
10: $currFace \leftarrow q.pop()$
11: $visEdge \leftarrow F[currFace]$ ▷ Obtenemos una arista de la cara que se está visitando
12: **repeat**
13: $visCmp \leftarrow cmp(visEdge)$
14: $cmpFace \leftarrow M[visCmp]$ ▷ Obtenemos a que cara pertenece la arista complementaria
15: **if** $V[cmpFace] = false$ **then**
16: $q.push(cmpFace)$
17: $V[cmpFace] \leftarrow true$
18: $T' \leftarrow T' \cup \{visEdge, visCmp\}$
19: **end if**
20: $visEdge \leftarrow nextCCW(visCmp)$
21: **until** $visEdge = F[currFace]$
22: **end while**
23: **return** $G - T'$
24: **end procedure**

Algorithm 4 DFS limitado

```
1: procedure dfsLimitado(o, n, G)  ▷ Retorna el spanning tree con DFS limitado a n niveles iniciado en o
2:     T ← Una lista de aristas inicialmente vacía
3:     q ← Una cola de vértices vacía
4:     p ← Una pila de vértices vacía
5:     V ← Un arreglo booleano de tamaño G.vertices iniciado en false
6:     H ← Un arreglo de enteros de tamaño G.vertices iniciado en 0  ▷ La altura de los nodos
7:     p.push(o)
8:     V[o] ← true
9:     while p no está vacío do
10:        u ← p.pop()
11:        h ← H[u]
12:        for cada vecino v de u do
13:            if V[v] = false then
14:                V[v] = true
15:                H[v] = H[u] + 1
16:                Agregar aristas (v, u) y (u, v) a T
17:                if H[v] < n then          ▷ Todavía quedan niveles por agregar
18:                    p.push(v)
19:                else                      ▷ Está en el límite, se agrega a la cola
20:                    q.push(v)
21:                end if
22:            end if
23:        end for
24:    end while
25:    while q no está vacío do
26:        u ← q.pop()
27:        for cada vecino v de u do
28:            if V[v] = false then
29:                V[v] = true
30:                Agregar aristas (v, u) y (u, v) a T
31:                q.push(v)
32:            end if
33:        end for
34:    end while
35:    return T
36: end procedure
```

Algorithm 5 BFS alternado

```
1: procedure bfsHeur(extEdge, G)
2:     T ← Una lista de aristas inicialmente vacía
3:     qVertex ← Una cola de vértices vacía
4:     qFaces ← Una cola de caras vacía
5:     vVertex ← Un arreglo booleano de tamaño G.vertices iniciado en false
6:     vFaces ← Un arreglo booleano de tamaño 2 + G.edges − G.vertices iniciado
   en false
7:     vEdges ← Un arreglo booleano de tamaño 2 · G.edges iniciado en false
8:     hVertex ← Un arreglo de enteros de tamaño G.vertices iniciado en 0
9:     hFaces ← Un arreglo de enteros de tamaño 2 + G.edges − G.vertices iniciado
   en 0
10:    < M, F >← marcarAristas(G)
11:    initFace ← M[extEdge]
12:    qFaces.push(initFace)
13:    vFaces[extFace] ← true
14:    initVertex ← extEdge.src                ▷ src es el vértice origen de la arista.
15:    qVertex.push(initVertex)
16:    vVertex[initVertex] ← true
17:    onPrimal ← true
18:    while qVertex no está vacía o qFaces no está vacía do
19:        if onPrimal then
20:            currHeight ← hVertex[qVertex.front()]
21:            while qVertex no está vacía y currHeight =
   hVertex[qVertex.front()] do
22:                u ← qVertex.pop()
23:                for cada arista e saliendo de u do
24:                    v ← e.tgt
25:                    if vVertex[v] = false y vEdges[e] = false then
26:                        vVertex[v] = true
27:                        T ← T ∪ {e, e.cmp}
28:                        vEdges[e] ← true
29:                        vEdges[e.cmp] ← true
30:                        qVertex.push(v)
31:                        hVertex[v] ← currHeight + 1
32:                    end if
33:                end for
34:            end while
```

Algorithm 5 BFS alternado (continuación)

35: **else**
36: $currHeight \leftarrow hFaces[qFaces.front()]$
37: **while** $qFaces$ no está vacía y $currHeight = hFaces[qFaces.front()]$ **do**
38: $currFace \leftarrow qFaces.pop()$
39: $visEdge \leftarrow F[currFace]$
40: **repeat**
41: $visCmp \leftarrow cmp(visEdge)$
42: $cmpFace \leftarrow M[visCmp]$
43: **if** $vEdges[visEdge] = false$ y $vFaces[cmpFace] = false$ **then**
44: $qFaces.push(cmpFace)$
45: $vFaces[cmpFace] \leftarrow true$
46: $vEdges[visEdge] \leftarrow true$
47: $vEdges[visCmp] \leftarrow true$
48: $hFaces[cmpFace] \leftarrow currHeight + 1$
49: **end if**
50: $visEdge \leftarrow nextCCW(visCmp)$
51: **until** $visEdge = F[currFace]$
52: **end while**
53: **end if**
54: $onPrimal \leftarrow \neg(onPrimal)$
55: **end while**
56: **return** T
57: **end procedure**