Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 1

M-Tree

Integrantes: Felipe Avendaño A.
J. Paz Fredes C.
Roberto Rivera C.
Profesor: Benjamín Bustos
Gonzalo Navarro
Auxiliar: Sergio Rojas
Diego Salas

Fecha de entrega: 12 de mayo de 2024.
Santiago de Chile

Índice de Contenidos

1. Introducción

En el presente informe se detallarán las implementaciones y estudio de la estructura de datos M-Tree, la cuál es similar a otros árboles como R-tree o B-tree. Esta estructura se construye utilizando alguna métrica y se basa en la desigualdad triangular; en particular para este trabajo se utiliza la distancia euclidiana y los datos a almacenar serán puntos en el plano cartesiano.

Se detallará la implementación realizada tanto del algoritmo de búsqueda en M-Tree como de dos métodos de construcción, cada uno con un enfoque distinto; Ciaccia-Patella y Sexton-Swinbank, así como los experimentos realizados para probar cada uno de estos métodos de cosntrucción y los resultados obetenidos de dichos experimentos.

Se espera que el método Sexton-Swinbank tome más tiempo de construcción en los experimentos dado el costo del algoritmo para generar clusters que utiliza el cuál ejecuta varias veces; en particular la subrutina para encontrar pares más cercanos es de una complejidad computacional estrictamente mayor a $O(n^2)$ en términos asintóticos. Además, los propios autores de estas implementaciones las hicieron con enfoques distintos; mientras Sexton-Swinbank se centra en optimizar las búsquedas, Ciaccia-Patella lo hace en reducir el tiempo de construcción. Es por esto último también que se espera que Sexton-Swinbank tenga mejor rendimiento al resolver consultas.

2. Desarrollo

El objetivo consiste en la comparación de dos métodos de construcción para la estructura R-Tree, en términos de tiempo y accesos al disco. Se usaría además la implementación de un método de búsqueda en el M-Tree, con el propósito de ver la variación del rendimiento de este método dependiendo del solapamiento de regiones espaciales de los nodos del árbol, lo cual depende de la manera en la que es construído.

Para el desarrollo del trabajo se decidió optar por el lenguaje de programación C++ dadas las facilidades y utilidades que otorgan sus diversas librerías en el manejo de estructuras de datos, y, dado que se asume que los inputs no contendrán puntos repetidos, se decidió en general trabajar los conjuntos de puntos como vectores.

Se definieron estructuras de datos comunes para ambos métodos de construcción y la búsqueda las cuales son: *Point* que representa un punto en el plano cartesiano dando sus coordenadas con valores reales de doble precisión, *Node* que representa un nodo del árbol y contiene un vector de entradas llamado *Entries*, *Entry* que es una entrada y contiene un punto, radio cobertor y un puntero que apunta hacia el nodo hijo. Finalmente se define *MTree* como una estructura que tiene un puntero a un nodo raíz y que almacena la altura como un entero.

2.1. Sexton-Swinbank (SS)

Este método se basa en la generación de grupos de clusters los cuales se utilizarán para armar el M-Tree, es por ello que se define la estructura *Cluster*, la cual almacena un vector que contiene el conjunto de puntos asociados y el punto medoide.

Además, la implementación de los clusters requiere de definir funciones auxiliares para calcular el medoide de un conjunto de puntos, establecer el medoide de un cluster, crear un cluster dado un conjunto de puntos, unir dos clusters, y generar aleatoriamente un vector de puntos, arreglo de puntos, y un vector de clusters singletons (que contienen un único punto y por tanto el medoide será él mismo).

La función principal que inicializa el proceso de construcción corresponde a *Algorithm_SS*. Ésta recibe como input un grupo de puntos (representando por un vector de puntos), a partir de de los cuales se desea construir el árbol, realizándose dentro de ella el proceso de iteración recursiva, el cual será descrito a continuación.

- Inicia del caso base en que el tamaño del conjunto de puntos sea menor o igual a B, escenario en que se procede a llamar a la *Entry* OutputLeaf() para crear un nodo de hoja que contenga aquellos puntos, calculando el medoide y radio del cluster para devolver una estructura de entrada que contenga esta información.
- En el caso contrario, se llama a la función *Cluster_SS* que recibe este mismo conjunto de puntos, con el propósito de agruparlos en clusters, procesándolos de manera recursiva para armar el árbol. Hace uso del método de búsqueda *nearestPairBrute()* para encontrar los clusters más cercanos en cada iteración. Hecho esto, se decide si fusionar los clusters o dividirlos en función de su tamaño. Si al sumarlos, el tamaño del cluster resultante es menor o igual a B, se fusionan en un nuevo cluster. Sino, se dividen utilizando el método *MinMax*.

- Para cada cluster se crea una *Entry* para un nodo interno u hoja. En el primer caso se utiliza OutputInternal(), mientras que en el segundo caso se utiliza OutputLeaf() como fue mencionado anteriormente. La iteración se repite hasta que el número de entradas en el nodo actual es menor o igual a B. Una vez completado el proceso, se retorna el puntero al nodo raíz del árbol.

2.2. Ciaccia-Patella (CP)

Este método utiliza un tipo distinto de clustering al presentado previamente. Este clustering se realiza a partir de una cantidad n de puntos, obteníendose un árbol M balanceado cuyos nodos cumplen con estar definidos dentro de las capacidades permitidas.

La implementación de este método se hizo en varios pasos, ordenados y ejecutados por la función *Algorithm_CP*, la cual recibe como input un set de puntos (representado en código como un vector de puntos), retornando el puntero hacia el nodo raíz del árbol construido. A continuación se revisará, los pasos realizados para armarlo.

- Se comienza definiendo el caso base, en el cual el número de puntos sea menor o igual a B, creándose un nodo hoja que contiene todos los puntos. Éstos se ordenan dentro del nodo para facilitar las búsquedas posteriores. Debido al ordenamiento de las entradas, la complejidad de este paso es de $\theta(n \log n)$

- En el caso contrario, se seleccionan k muestras aleatorias de los puntos dados, haciendo uso de la función *randomSample* del módulo *sample_utils.h*. Estas muestras subdividen el grupo el conjunto de puntos en grupos más pequeños.

- Los puntos se asignan a los clusters basados en la muestra más cercana a cada punto, las cuales son determinadas utilizando *nearestSample()* del módulo *sample_utils.h*

- Si algún cluster tiene menos de b puntos (que es un valor definido como capacidad mínima de un cluster), éstos se redistribuyen entre los clusters restantes. Se utiliza la función *erase()* de la clase vector para eliminar muestras del conjunto de muestras, y la función *push_back()* para agregar los puntos a los clusters. Si sólo queda una muestra, se repite el proceso de selección de muestras. En el caso contrario en que se cumple la condición, se utiliza *goto* para saltar al siguiente paso.

- Se utiliza *Algorithm_CP* recursivamente para la construcción de sub-árboles para cada cluster. Se chequea la existencia de *underfill* en los sub-árboles construídos, es decir, aquellos que no alcanzan el tamaño mínimo b son eliminados. Para esto se utiliza nuevamente la función *erase()*, y la función *push_back()* para agregar sub-árboles y puntos a una lista.

- Se empieza el proceso de balanceamiento usando la función *getHeights()* para determinar las alturas de los sub-árboles. Luego, se prepara el conjunto de puntos y sub-árboles que se utilizarán para construir el árbol final, haciendo uso de las funciones *getLeaves()* y *getNodesUpToDepth()* para obtener las hojas y sub-árboles necesarios.

- Se llama recursivamente *Algorithm_CP* para armar el árbol final con las estructuras preparadas en el paso anterior, adjuntando sus hojas las raíces de los sub-árboles construídos, las cuales son encontradas haciendo uso de *Node_binarySearch()*.

- Se calculan y establecen los radios cobertores para las hojas del árbol final usando las funciones *calculateCR()*, *getLeaves()* y *getOuterPoints()*. Finalmente, se retorna la raíz del árbol construido.

Cabe agregar que en la implementación de este método, la complejidad computacional de las operaciones de búsqueda, redistribución y balanceamiento está influenciada por el uso de funciones auxiliares como *BFS (Breadth-First Search)*, que se utiliza para obtener los niveles de los árboles, tanto para encontrar los sub-árboles de altura h como para la liberación de memoria, y *DFS (Depth-First Search)*, que es utilizada para actualizar el radio de cada entrada agregada, mediante un proceso de búsqueda inversa, revisando los puntos fuera del radio anterior (usando la función *getOuterPoints()* buscarlos).

2.3. Búsqueda

Finalmente, para la realización del experimento se implementa el método de búsqueda *MSearch* con el objetivo de medir el tiempo que se demora dependiendo del método de construcción utilizado para cada árbol. *MSearch* recibe como input el nodo actual donde se realiza la búsqueda, una query y un vector *result* donde se almacenan los puntos encontrados en el radio cobertor de la query.

- Se comienza el proceso inicializando un contador *counter* en 1, el cual se utiliza para contabilizar el número de nodos visitados durante la búsqueda, considerando esto como el número simulado de accesos al disco.
- Empezando a recorrer iterativamente el vector *entries*, el cual contiene los nodos. Se identifican dos casos. El primer caso en que el nodo corresponda a una hoja, es decir, su puntero a es *nullptr*, se verifica la si la distancia entre el punto del input y el punto de la consulta es menor o igual al radio de la consulta al cuadrado. Si es así, el punto se agrega con *push_back()* al vector *result*.
- En el caso contrario, el nodo se considera interno y se verifica igualmente la distancia entre el punto del input y el punto de consulta es menor igual al radio de consulta al cuadrado, y en el caso en que se cumpla esta condición, se realiza una búsqueda recursiva en el nodo hijo y se incrementa el número de accesos en el contador.
- La función retorna el número entero almacenado en *counter*, es decir, retorna los nodos visitados en total/accesos al disco.
- Luego, se define la función *MSearch* que recibe un MTree y una query, con el propósito de llamar recursivamente la función definida anteriormente para que recorra los nodos del árbol, empezando por el nodo raíz. La llamada a esta función se hace de forma *MSearch(*tree.root, query, result)*. Finalmente, esta función retorna el total de nodos visitados al buscar en el árbol entero.

3. Experimentos y Resultados

3.1. Datos Utilizados

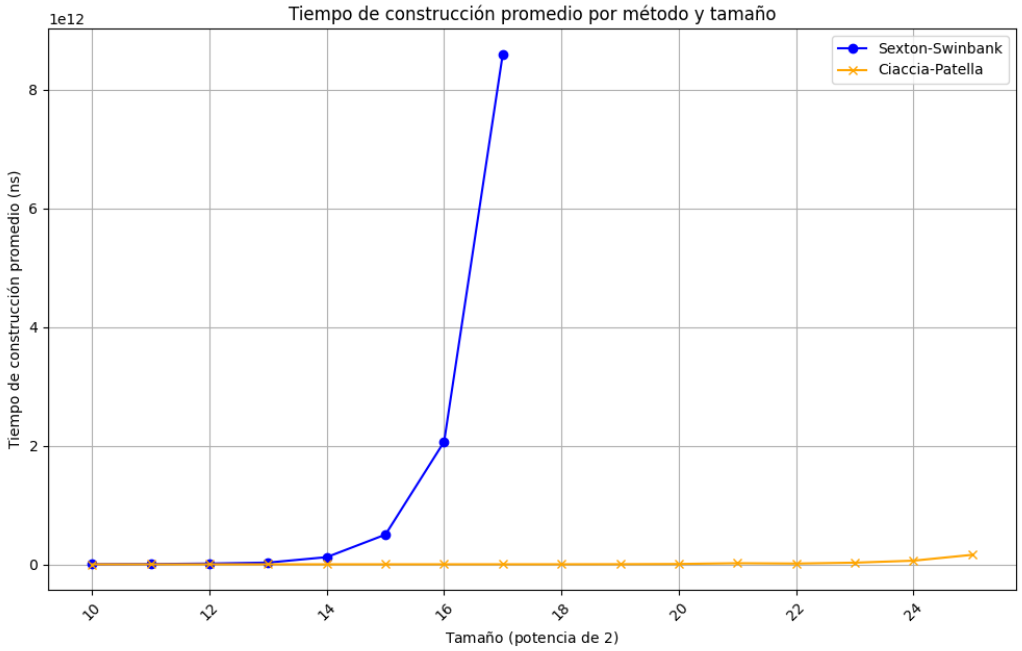
Para la realización del experimento se utilizaron como datos; conjuntos de puntos y consultas generados mediante una semilla de aleatoriedad para garantizar la estabilidad y reproductibilidad de los resultados. En particular, se generó un único conjunto aleatorio de 100 consultas para todas las pruebas, y para cada cantidad $n \in \{2^{10}, 2^{11}, \ldots, 2^{25}\}$ de puntos, se generó en una iteración esa cantidad de puntos aleatorios. Dado el riesgo de correr todas las iteraciones seguidas, se decidió ejecutar por separado, primero todos los n hasta 2^{15} para ambos métodos (considerando que este era el mínimo de datos solicitados para el método de Sexton-Swinbank) y después de 2^{16} hasta 2^{20} y desde 2^{21} hasta 2^{25} exclusivamente para el método Ciaccia-Patella. Finalmente, se intentó ejecutar los experimentos faltantes para Sexton-Swinbank pero este solo llegó a completar hasta 2^{17}.

En particular los experimentos se realizaron en un computador con sistema operativo Windows 11 con 12GB de RAM y 8MB de caché de CPU.

3.2. Resultados

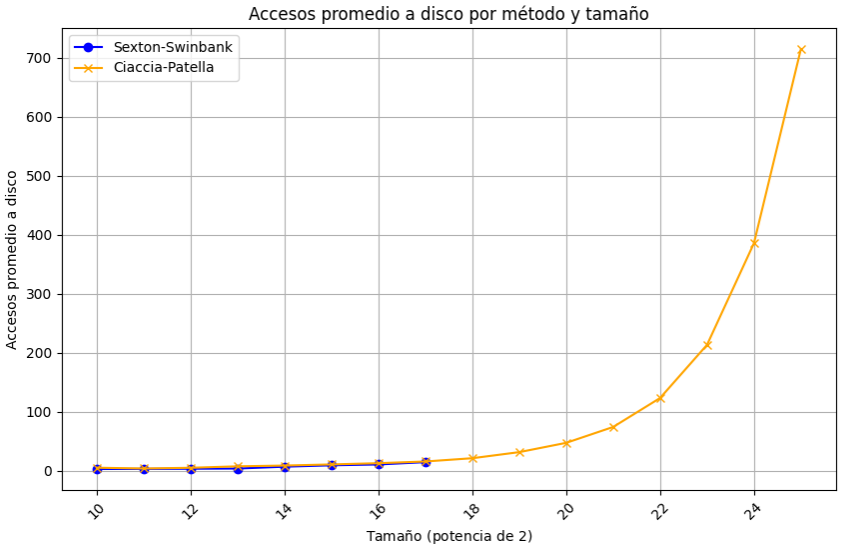
A continuación se representarán gráficamente y observarán los resultados obtenidos a partir de la experimentación. Estos resultados se compararán en relación al método de construcción utilizado, con el objetivo de determinar las diferencias en términos de tiempo y complejidad entre ellos.

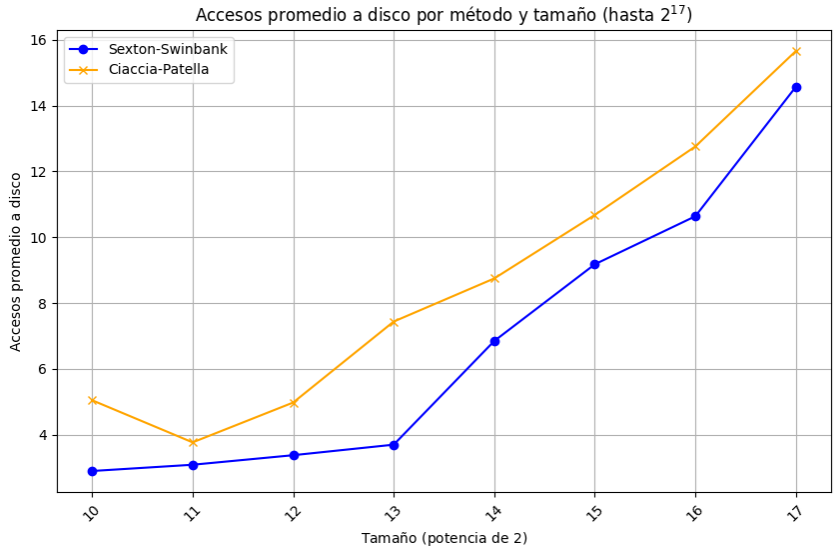
3.2.1. Tiempos de Construcción



Se aprecia una diferencia notable en los tiempos de construcción pues el método Sexton-Swinbank con 2^{14} puntos demora más que casi cualquier iteración de Ciaccia-Patella, solo comparable a lo que demora este último con 2^{25} puntos. Además, mientras el mayor tiempo registrado para SS, que corresponde a 2^{17} es de 8598786545000 nanosegundos (aproximadamente 2.4 horas), el mayor tiempo de construcción para CP, con 2^{25} puntos, es de 159328436000 nanosegundos (2.6 minutos). La forma que tiene de crecer el tiempo de construcción de SS es similar a una función exponencial.

3.2.2. Accesos a disco



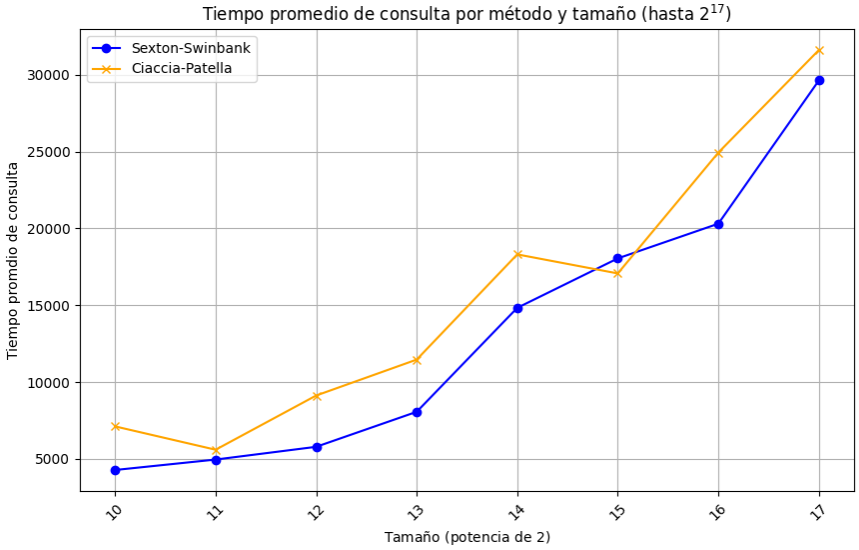


Respecto a la cantidad de accesos promedio a disco para las 100 consultas, se puede apreciar que Sexton-Swinbank requiere siempre menos, teniendo la mayor brecha de diferencia con 2^{13} puntos. Los intervalos de confianza del 95 % para cada iteración y método son:

Método	**Iteración**	**AVG**	**STD**	**Intervalo de Confianza**
SS	10	2.9	0.798	[2.744, 3.056]
SS	11	3.09	0.740	[2.945, 3.235]
SS	12	3.38	0.838	[3.216, 3.544]
SS	13	3.7	0.870	[3.529, 3.871]
SS	14	6.85	1.520	[6.552, 7.148]
SS	15	9.18	1.783	[8.831, 9.529]
SS	16	12.76	2.441	[12.281, 13.238]
SS	17	15.66	2.641	[15.142, 16.177]
CP	10	5.05	1.772	[4.703, 5.397]
CP	11	3.77	1.230	[3.529, 4.011]
CP	12	4.98	1.348	[4.716, 5.244]
CP	13	7.44	2.690	[6.913, 7.967]
CP	14	8.75	2.148	[8.329, 9.171]
CP	15	10.68	2.291	[10.231, 11.129]
CP	16	12.76	2.442	[12.281, 13.239]
CP	17	15.66	2.641	[15.142, 16.178]
CP	18	21.21	3.508	[20.522, 21.898]
CP	19	31.54	5.478	[30.466, 32.614]
CP	20	47.31	6.538	[46.029, 48.591]
CP	21	74.2	9.710	[72.297, 76.103]
CP	22	123.33	13.559	[120.672, 125.988]
CP	23	213.45	23.510	[208.842, 218.058]
CP	24	385.8	39.778	[378.003, 393.597]
CP	25	715.02	72.470	[700.816, 729.224]

3.2.3. Tiempos de Consulta

Finalmente, respecto a los tiempos de consulta se tienen los siguientes datos:



Se aprecia que los tiempos de consulta promedios de Sexton-Swinbank suelen ser menores salvo para el caso de 2^{15} puntos. Los intervalos de confianza del 95 % para cada iteración y método son:

Método	Iteración	AVG	STD	Intervalo de Confianza
SS	10	4270.0	1780.109	[3921.099, 4618.901]
SS	11	4950.0	1924.720	[4572.755, 5327.245]
SS	12	5780.0	1967.309	[5394.408, 6165.592]
SS	13	8060.0	2827.784	[7505.754, 8614.246]
SS	14	14830.0	5144.488	[13821.680, 15838.320]
SS	15	18050.0	4957.140	[17078.401, 19021.599]
SS	16	20310.0	4823.542	[19364.585, 21255.414]
SS	17	29639.344	7418.519	[28185.314, 31093.374]
CP	10	7110.0	3786.993	[6367.749, 7852.251]
CP	11	5590.0	2005.523	[5196.918, 5983.082]
CP	12	9120.0	3811.943	[8372.859, 9867.141]
CP	13	11460.0	7145.557	[10059.471, 12860.529]
CP	14	18310.0	13356.567	[15692.113, 20927.887]
CP	15	17070.0	7055.702	[15687.082, 18452.918]
CP	16	24940.0	7532.851	[23463.561, 26416.439]
CP	17	31620.0	5584.575	[30525.423, 32714.577]
CP	18	43500.0	9347.932	[41667.805, 45332.195]
CP	19	61990.0	11872.636	[59662.963, 64317.037]
CP	20	111720.0	24073.994	[107001.497, 116438.503]
CP	21	222490.0	57032.065	[211311.715, 233668.285]
CP	22	258190.0	29028.963	[252500.323, 263879.677]
CP	23	446320.0	50412.216	[436439.206, 456200.794]
CP	24	765860.0	84683.728	[749261.989, 782458.011]
CP	25	1894640.0	448433.244	[1806747.084, 1982532.916]

4. Comentarios y Conclusiones

A partir de los datos observados, es posible afirmar que el método Sexton-Swinbank es más eficiente para resolver consultas que el método Ciaccia-Patella, pues toma en general menos tiempo y además siempre menos accesos simulados a disco al momento de resolver estas.

La mayor diferencia registrada de accesos a disco promedio es en el caso de los árboles de 2^{13} puntos con alrededor de 4 accesos menos para el método Sexton-Swinbank, disminuyendo luego esta diferencia para números más grandes de puntos. Si bien los tiempos promedio de consulta para cada método son similares, se debe tener en cuenta que todo el experimento fue realizado en memoria principal, que siempre es más rápida, por lo que estas diferencias sí podrían ser notables si se trabajara en memoria secundaria donde un acceso a disco extra sería considerablemente más costoso. Es por ello que se decide no darle tanta importancia a los tiempos de consulta como sí a la cantidad de accesos a disco.

Respecto a los tiempos de construcción en función de la cantidad de puntos, el del método Sexton-Swinbank empieza a crecer explosivamente, similar a una exponencial, desde el valor $n = 2^{14}$ puntos, llegando a tener una diferencia de hasta 4 órdenes de magnitud con el tiempo de construcción de Ciaccia-Patella tan solo en el caso de 2^{17} puntos, donde se detiene el experimento. No sería sorprendente que esta diferencia llegase incluso a 10 órdenes de magnitud en las siguientes iteraciones que no se completaron.

Se observa que el método Ciaccia-Patellla es considerablemente más rápido en su construcción, especialmente para cantidades de puntos mayores a 2^{14}.

Se concluye que si bien ambos métodos de construcción pueden ser prácticos, lo serán en distintos contextos: Ciaccia-Patella es más útil cuando se quiere poder construir un árbol rápidamente, y quizás en caso de querer tener un árbol que en cualquier momento podría ser borrado y reemplazado por otro mientras Sexton-Swinbank es más conveniente cuando se quieren almacenar datos por largos períodos de tiempo para ser consultados (pues las consultas cuestan menos accesos a disco) o incluso si se quiere que estos datos puedan ser reescritos, pues pueden ser encontrados más rápidamente desde la raíz, pero para saber esto último habría que implementar y testear la inserción y el borrado para M-trees.

5. Recapitulación

Se implementaron ambos métodos de construcción solicitados efectivamente y se realizaron los experimentos pertinentes, cuyos resultados se condicen con la hipótesis planteada en la introducción: el método Sexton-Swinbank efectivamente demora más tiempo en la construcción a cambio de en general tomar menos tiempo y accesos a disco en resolver las consultas, lo cuál se corresponde también al enfoque planteado por sus creadores.

Esto lleva a que ambos métodos de construcción sean útiles en distintos contextos, según se quiera o bien una construcción rápida y quizás desechable de la estructura o si se espera mantener esta por un período de tiempo considerable y realizar consultas sobre ella.

Podría además terminar de probarse el método Sexton-Swinbank para los $n \in \{2^{18}, \ldots, 2^{25}\}$ si se dejara un computador dedicado ejecutando durante varios días, quizás incluso más de una semana, pero no disponemos de dicha capacidad. Por otro lado, un desafío interesante sería implementar la inserción y el borrado para analizar si en estas operaciones efectivamente también se ahorra tiempo al ejecutarlas sobre un árbol construido con el método de Sexton-Swinbank.