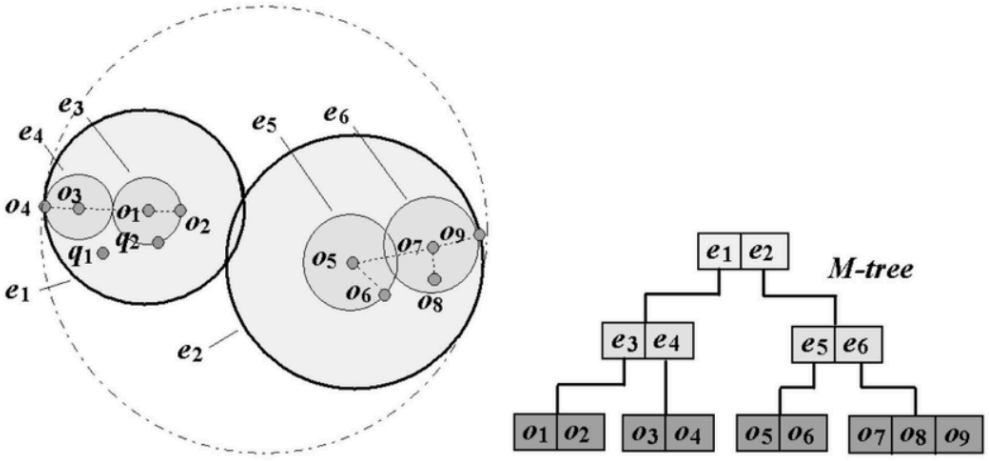

Algoritmos de Construcción de M-Trees: Diseño y Análisis.

CC4102 Diseño y Análisis de Algoritmos. Otoño 2024.

Profesores de Cátedra: Benjamín Bustos y Gonzalo Navarro
Profesores Auxiliares: Sergio Rojas y Diego Salas

Autores: Valentina Alarcón Y, Naomí Cautivo B y Máximo Flores Valenzuela.

Índice

Introducción

Un M-Tree es un tipo de estructura de datos dinámica que forma un árbol, el cual nace de la necesidad de indexar sobre diversos tipos de datos en los cuales se posen diferentes características, como por ejemplo, bases de datos multimedia.

Este tipo de estructuras requiere de ser dotado de una métrica (lo que le da la "`M`" de `M-Tree`), la cual cumple dentro del espacio de los datos el rol de medir similitud o diferencia entre ellos. Para este trabajo, ocuparemos la distancia euclideana $\|\cdot\|_2$ en $\mathbb{R}^2$, para puntos en el intervalo $[0, 1]$.

Para construir estos árboles, se pueden utilizar diversos métodos de construcción, los cuales varían en su complejidad como estructura, en este caso se utilizarán los métodos de Ciaccia-Patella y de Sexton-Swinbank, los cuales serán utilizados para construir y posteriormente buscar en ellos, donde se medirá su complejidad temporal en relación a diversos tamaños de prueba para las entradas. Cabe destacar que se utilizó el lenguaje de programación `C++` para esto.

La hipótesis planteada es que el método de construcción más eficiente en términos de tiempo será Ciaccia-Patella debido a que Sexton-Swinbank realiza una forma de *clustering* sobre los puntos de entrada, que ya de por sí son procesos pesados en temas de tiempo.

Desarrollo

Algoritmo de Ciaccia-Patella

Para implementar este algoritmo se implementaron una serie de funciones auxiliares.

Cálculo de distancias

```C++
double distancia_cuadrado(const Point& punto1, const Point& punto2)
```

Para el cálculo de la distancia entre dos puntos se tomó en cuenta la distancia *euclideana* al cuadrado. Esto último fue así pues se pensó que el sacar la raíz cuadrada podría implicar costos extras innecesarios, puesto que la operación intrínsecamente es más costosa. Esto mantiene el invariante por la propiedad de positividad de la distancia *euclideana* ($\forall x, y \in \mathbb{R}, d(x, y) \geq 0$) y la monotonía creciente estricta de la función cuadrado en su rama positiva, es decir, para cualesquiera $x, y > 0$, se tiene que $x < y \Rightarrow x^2 < y^2$.

Distribución de un *set* de puntos en los k (o menos) conjuntos existentes

```C++
void punto_mas_cercano(const set<Point>& points, map<Point, set<Point>>& mapa)
```

Luego de tener el conjunto original F de samples (formado por los k puntos aleatorios seleccionados al inicio), se debe asignar cada punto del *set* de puntos recibido como *input* en el método de construcción a un punto P perteneciente a él. Esto se hace de acuerdo a cuál de los puntos P es más cercano, y se finaliza teniendo k conjuntos (Paso 3). Un proceso similar de distribución se hace luego de eliminar un p_{f_j} de F tras identificar que su conjunto asociado tiene menos de $\frac{B}{2}$ entradas (Paso 4). Esta función se encarga de la distribución o redistribución, de acuerdo a lo que corresponda.

Esta función auxiliar recibe un set de puntos `points`, y el mapa `mapa` que contiene a samples como sus llaves (donde su valor relacionado es el conjunto de puntos que lo consideraron como su P más cercano).

Si la función se llama antes de distribuir cualquier cosa, entonces `points` es el mismo del *input* del método de construcción. Por otro lado, si se llama en el contexto del paso 4, `point` es el conjunto asociado al p_{f_j} descartado. En ambos casos `map` es el mismo mapa, solo que si estamos en el contexto del paso 3 todos los conjuntos inician estando vacíos.

Para cada punto, calcula la distancia entre él y cada punto del conjunto samples, seleccionando cada vez el que sea menor al actualmente definido. Esta función tiene un orden de $O(n \cdot |F|)$.

Búsqueda de subárboles de altura h

```C++
void busqueda_h(Nodo *nodo, const int h, map<Point, Nodo*>& arboles)
```

En el paso 9 descrito en el enunciado se describe el procedimiento en caso de que T_j no tenga altura h. Esto detona una búsqueda exhaustiva de los subárboles de T_j cuya altura sí sea h. Esta función toma un nodo, el entero h y un mapa donde se guardarán todas las respuestas correctas. Este último almacenará como clave los puntos raíz correspondiente a un subárbol, y el subárbol asociado.

Si el nodo recibido no es un puntero nulo ni es hoja, entonces debería tener nodos hijos. Cabe destacar que el nodo que recibe la función como *input* es aquel que ya se sabe que no tiene altura h. Luego, y dado que los nodos tienen su altura almacenada, se revisan las alturas de los hijos de `nodo`. Si es que estas son h, se toma el punto p de nodo como clave, y el nodo a como su subárbol. Luego, en el caso de que el hijo tenga altura menor a h, se ignora; y si es mayor, se busca recursivamente.

Es importante notar que, en realidad, no debería ocurrir que esta función se llame con nodo `nullptr` ni con un nodo hoja, puesto que h (la altura mínima encontrada entre los T_j) es siempre al menos 1, que es precisamente la altura *seteada* para los nodos asociados a las entradas hojas.

Setear radios cobertores

```C++
1  double setear_radio_cobertor(Entry& entry)
```

Como último paso del método de construcción, se conectan las hojas de T_{sup} con los subárboles T_j. Durante este proceso, se deben actualizar los radios cobertores de los nodos de T_{sup}. Para lograr esto, se llama a la función `setear_radio_cobertor` para cada entrada de los nodos de T_{sup}.

Esta función recibe una entrada, y toma su valor a. Para `entrada.a`, nodo al cual llamaremos `hijo`, revisaremos sus entradas. El radio cobertor a setear para `entry` será el máximo de todos los cálculos realizados de la suma entre:

1. La distancia entre el punto `entrada.p` y el punto `entry.p`.
2. El radio cobertor del hijo, `entrada.cr`.

Conexión T_{sup} y subárboles T_j

```C++
1  void conectar_arboles(Nodo* nodo, map<Point, Nodo*>& subarboles)
```

Esta función recibe como input un nodo que se denominará `padre`, el cual en la primera llamada será el nodo raíz de T_{sup}, y el mapa `subarboles`, el cual tiene como claves los puntos que se encuentran en las hojas de T_{sup}, y como valores los subárboles a los cuales se les va a asociar. Por otro lado, su algoritmo incluye tanto la conexión entre ambos, el *seteo* de las nuevas alturas de los nodos de T_{sup}, y el detonamiento de la función `setear_radio_cobertor` para cada entrada de sus nodos.

Para lograr lo mencionado previamente, primero se chequea la primera entrada del nodo padre. Si esta entrada tiene como valor a, un puntero nulo, esto significa que `padre` es una hoja. Luego, podemos conectar directamente estas entradas con los subárboles T_j. Para hacer esto, modificamos el valor a al nodo (al cual llamaremos `hijo`) que encontraremos en subarboles utilizando la clave `entrada.p`. Además, la altura del padre se *seteará* como 2, que es uno más que la altura de una hoja.

Por otro lado, si no es hoja, se llama recursivamente la función para todas sus entradas y se *setea* la altura acorde a la mayor altura encontrada en sus subárboles. También se *setea* el radio cobertor de cada una.

El algoritmo Ciaccia-Patella

```C++
Nodo* crear_MTree_CCP(const set<Point> points)
```

La función asociada a la creación de los árboles utilizando el algoritmo de Ciaccia-Patella utiliza los métodos mencionados previamente. Algunos aspectos relevantes a mencionar sobre decisiones que se tomaron en la implementación son los siguientes.

1. Se definió la altura de las hojas como 1.
2. El conjunto F no existe como un set separado. Este conjunto se ve usualmente como las claves de algunos mapas que se crearon, y es recién al final de la función (previo a la conexión entre T_{sup} y los subárboles T_j) donde se agrupan los puntos p_f en un set llamado `keys`.
3. Inicialmente, para el caso $n > B$, se tiene un mapa denominado `samples`, el cual asocia los puntos de F con su conjunto de `samples` asociado. La primera interacción con él es el agregar los k puntos (seleccionados aleatoriamente del *input* previamente) como claves, teniendo asociados un set de puntos vacío. Luego se llama a `punto_mas_cercano`, lo que *setea* sus valores asociados. Este mapa se sigue utilizando para la etapa de redistribución, y se itera sobre él para la detonación de la recursión sobre sus conjuntos.
4. Más tarde en la función se migra a la utilización del mapa `subarboles` el cual, a diferencia de su predecesor, asocia los puntos p_{f_j} con el subárbol al que debería apuntar. Este se arma luego de las llamadas recursivas a `crear_MTree_CCP`, y se actualiza tras la búsqueda de los subárboles de altura h si así se requiere.

Algoritmo Sexton-Swinbank

A diferencia del algoritmo de Ciaccia-Patella, ahora se debe hacer *clustering* sobre el conjunto de puntos. Para ello, se implementó una función que en un paso ocupa el *min-max split policy*, descrito a continuación. La idea de la función de *clustering* es particionar al conjunto de puntos S que vienen del *input* en una colección $\{S_i\}_{i \in I \subseteq \mathbb{N}}$, donde cada $S_i \subseteq S$.

Min-Max Split Policy

Se usó una estrategia que minimizara la cantidad de comparaciones que se basa en el siguiente algoritmo. Por simplicidad, diremos que P es el conjunto de puntos del *input*, y además, $|P| = n$.

1. Se modela el *mapeo* de puntos a índices como una función biyectiva $f : \{0, \ldots, n-1\} \to P$ tal que $f(i) = p$. Esto sólo requiere $\Omega(n)$ operaciones, pues sólo se recorre el conjunto P asignando puntos a los índices de manera creciente.

   Sea $v \to [\,]$ **e** $i \to 0$
   Para cada $p \in P$
   $\quad v[i] \to p$
   $\quad i \to i + 1$
   Regresar v

   La estructura que genera a v puede agregar elementos en $O(|v|)$ y acceder a ellos en $O(1)$. En C++, esta estructura fue extraída del header `<vector.h>`.

2. Se calculan todas las distancias necesarias. Se puede demostrar que el óptimo de este paso se alcanza en $\Omega\left(\binom{n}{2}\right)$ comparaciones por la definición del coeficiente binomial.

Demostración

Ocupando el argumento del adversario. Suponemos por contradicción que hacemos menos de $\binom{n}{2}$ comparaciones, entonces habrá algún punto p que no fue comparado con algún otro q, y esta distancia $d(p,q)$ puede ser mejor que todas las comparaciones previas de p.

Para hacer las $\binom{n}{2}$ comparaciones, necesitamos una estructura iterativa:

1 **Sea** $S \to [\emptyset, ..., \emptyset]$ **tal que** $|S| = n$
2 **Sea** $v \to [f(0), f(1), ..., f(n-1)]$
3 **Para cada** $i \in \{0, ..., n-1\}$
4 $p_i \to v[i]$
5 **Para cada** $j \in \{i+1, ..., n-1\}$
6 $p_j \to v[j]$
7 $S[i] \to S[i] \cup \{(d(p_i, p_j), j)\}$
8 $S[j] \to S[j] \cup \{(d(p_i, p_j), i)\}$
9 **Regresar** S

Notemos que para obtener v es necesario haber pasado por el paso 1, pues v posee la información sobre el punto asociado a cada índice, que será útil para definir el conjunto S. Formalmente, una entrada $S[i]$ se define de la siguiente forma:

$$S[i] = \{(e_1, e_2) \in \mathbb{R}^+ \times \{0, ..., n-1\} \setminus \{i\} \mid \exists j \in \{0, ..., n-1\} \setminus \{i\},\ e_1 = d(v[i], v[j]) \wedge e_2 = j\}$$

donde $d(\cdot, \cdot)$ es la distancia euclideana, e i y j son las representaciones indexadas de dos puntos distintos $p_i, p_j \in P$. En otras palabras, $S[i]$ es el conjunto de pares ordenados formados por las distancias del punto p_i con algún otro punto p_j, y en la segunda coordenada se guarda cuál es el índice que representa al punto que indujo esa métrica. Inicialmente todo $S[i]$ es un conjunto vacío, pues no hemos hecho comparaciones.

El invariante es en cada paso aprovechar la simetría de la distancia euclideana, es decir, para cualesquiera $x, y \in \mathbb{R}^2$, $d(x,y) = d(y,x)$ para realizar las $\binom{n}{2}$ comparaciones. Además, no es necesario guardar las distancias de un punto con sí mismo, pues sabemos de antemano que $\forall x \in \mathbb{R}^2, d(x,x) = 0$.

La estructura de S tiene complejidad $O(|S|)$ al crearla y $O(1)$ al actualizar o acceder a cualquiera de sus datos. Además, cada índice de S está dotado de otra estructura T con complejidad de inserción y borrado $O(\log|T|)$ y complejidad de acceso a sus datos $O(|T|)$. T se comporta igual que un conjunto matemático, es decir, si $\exists p, p \in T \Rightarrow T \cup \{p\} = T$ (no puede tener elementos repetidos).

Una propiedad destacable de T es que ordena automáticamente –y de manera ascendente– sus elementos al realizar una inserción. En el caso de la inserción de pares ordenados

como lo es el del algoritmo (se inserta un par $(d(p_i, p_j), i)$ y $(d(p_i, p_j), j)$ en cada iteración) el ordenamiento se realiza por la primera coordenada, que justamente es la distancia. Nos aprovechamos de esta propiedad para generar un *ranking* de los pares más cercanos a cada punto.

En particular, S está programado como un `vector` (`<vector>`) de C++, y T está programado como un `set` (`<set>`) de C++.

Proposición
El algoritmo propuesto hace exactamente $\Theta\left(\binom{n}{2}\right)$ comparaciones.

Demostración

Por la estructura del algoritmo, podemos calcular el número de pasos como:

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1$$

y la suma de unos es conocida, en particular, la suma interna queda de la siguiente manera: $(n-1) - (i+1) + 1 = n - i - 1$. Ahora bien, reescribiendo tendremos:

$$\sum_{i=0}^{n-1} (n-1) - i = \sum_{i=0}^{n-1} (n-1) - \sum_{i=0}^{n-1} i = (n-1) \cdot n - \frac{(n-1) \cdot n}{2} = \binom{n}{2}$$

Para el resto del algoritmo, se usó la implementación ya conocida, dado que ya se posee el *ranking* de distancias hacia cada punto. Para marcar los puntos que ya fueron ocupados, se ocupó otro vector (`<vector>`) que en la i-ésima posición (representante del punto p_i, pues recordemos que $\forall p \in P, f(i) = p$ con f biyectiva) se pone un 1 si ya se hizo una comparación, y 0 si no.

Experimentación

Para las búsquedas se ocupó el archivo `queries.txt` que contiene 100 pares ordenados de la forma (x, y) donde $x, y \in [0, 1]$ son del tipo `double`. Este archivo se generó usando una función programada en C++ que usa la librería `<random>` con algoritmos que generan números pseudo-aleatorios que luego fueron escritos en el `.txt` usando la librería `<fstream>`.

```
(0.0182257, 0.581204)
(0.0319226, 0.340618)
...
(0.990259, 0.612797)
```

Este archivo puede ser recuperado desde la siguiente página (*click* acá).

De manera similar, para cada $n \in \{2^{10}, \ldots, 2^{25}\}$ se crearon archivos de la forma `pow2<pot>.txt` donde `<pot>` es la potencia de 2 correspondiente. Un ejemplo para 2^{15} se puede recuperar desde la siguiente página (*click* acá).

En particular, para el algoritmo de Sexton-Swinbank se probó con $n \in \{2^{10}, \ldots, 2^{15}\}$, y los *inputs* usados fueron los mismos para ambos algoritmos en este rango.

De ahora en adelante, entenderemos un acceso del algoritmo X como la lectura de un nodo, ya sea interno o externo. Para contar los accesos que realizaba cada método en cada valor de n, se dotó a la función de búsqueda de un vector v (extraído desde `<vector>`) y un índice i, ambos pasados por referencia (es decir, no se creaba una copia de los parámetros en cada llamado recursivo) con la notación "`&`". Con esta información, cada vez que se accedía a un nodo, se le sumaba una unidad a $v[i]$, donde i era el punto actual del archivo `queries.txt`, $i \in \{1, \ldots, 100\}$.

Las pruebas se realizaron usando el sistema operativo Windows, pero desde WSL2 ("Subsistema de Windows para Linux"). Para el caso de Ciaccia-Patella, se ocupó la distribución Ubuntu, y para Sexton-Swinbank, Debian. Como estos dos sistemas operativos comparten sus herramientas, principios y estructuras subyacentes, las diferencias de tiempo –en caso de existir– no son significativas al evaluar rendimiento como para considerarlas en los resultados.

Los tamaños del caché del sistema operativo son los siguientes (obtenidos con el comando de Linux `lscpu | grep "cache"`) para Sexton-Swinbank y Ciaccia-Patella respectivamente:

```
L1d cache:                       288 KiB (6 instances)
L1i cache:                       192 KiB (6 instances)
L2 cache:                        7.5 MiB (6 instances)
L3 cache:                        12 MiB (1 instance)
```

```
L1d cache:                    128 KiB
L1i cache:                    256 KiB
L2 cache:                     2 MiB
L3 cache:                     4 MiB
```

Y la RAM del equipo para ejecutar el algoritmo de Sexton-Swinbank fue de 8 GB. Además, la RAM del equipo para ejecutar Ciaccia-Patella fue de 16 GB. Esto se decidió de esta manera pues el último método mencionado requería de más memoria para el caso con $n = 2^{25}$. Sin embargo, la

única diferencia es la memoria extra, puesto que al compararlo con resultados obtenidos para $n = 2^k$ con $k < 25$ se obtuvieron resultados similares.

Para valores pequeños de n en el algoritmo de Sexton-Swinbank, es decir, $n \in \{2^{10}, 2^{11}, 2^{12}\}$ se realizaron 3 *tests* con las 100 iteraciones del archivo `queries.txt`. Sin embargo, para los valores críticos $n \in \{2^{14}, 2^{15}\}$ se realizó sólo 1 *test* con las 100 iteraciones del archivo mencionado. Esto fue porque para $n = 2^k$ con $k > 15$ los tiempos de ejecución de Sexton-Swinbank eran considerablemente altos (del orden de horas) para poder realizar consistentemente más pruebas.

En el caso del algoritmo Ciaccia-Patella, y debido a que este tenía un tiempo de ejecución muy inferior a su contraparte, se pudo testear 4 veces considerando el rango desde $n = 2^{10}$, hasta 2^{25}. Es decir, las 100 búsquedas se realizaron 4 veces cada una. Sin embargo, esto incluye solamente los *tests* de búsquedas, puesto que la construcción del árbol tuvo su revisión propia, lo que incluyó una mayor cantidad de *tests*.

Para cada valor de n, dado que se realizaron 100 consultas, y se reportaron los siguientes intervalos de confianza del 95% para $\mu = \frac{1}{n}\sum_i x_i$ (con $\alpha = 0,05$) en cuanto a los accesos a memoria:

Valor de k, con $n = 2^k$	Accesos de Ciaccia Patella	Accesos de Sexton-Swinbank
10	$(3.41055, 3.86945)$	$(3.55391205, 3.92608795)$
11	$(4.92355, 5.41645)$	$(3.74140386, 4.13859614)$
12	$(4.12048, 4.69952)$	$(3.93353781, 4.36646219)$
13	$(4.09405, 4.52595)$	$(4.37713418, 4.82286582)$

Gráficos

A continuación se presentan los gráficos realizados en base a los experimentos descritos anteriormente.

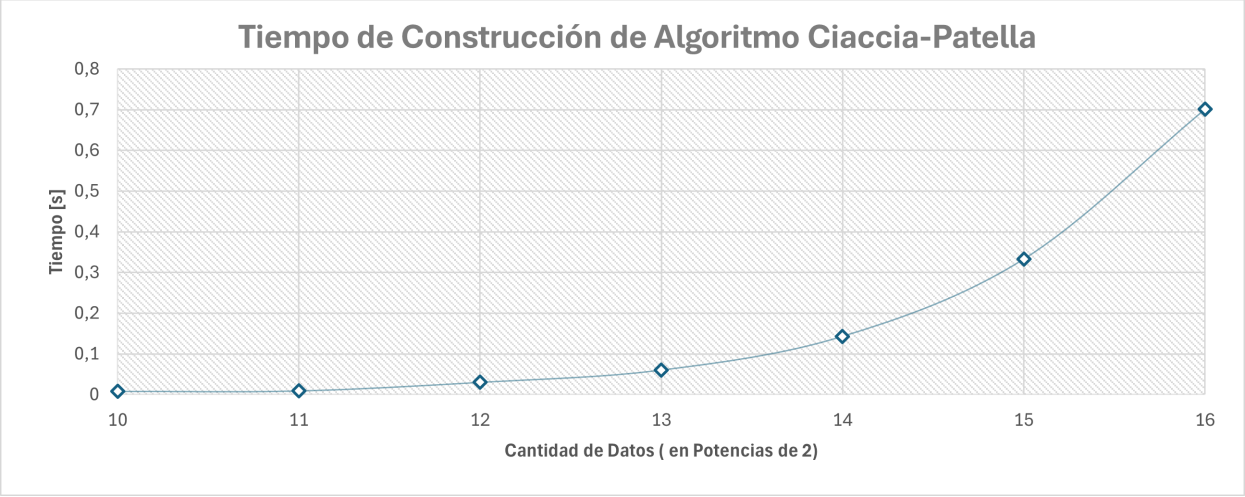

Figura 2: Tiempo de construcción de Ciaccia-Patella, para $n \in \{2^{10}, \ldots 2^{16}\}$

En la primera figura, se muestra el crecimiento de tiempo de construcción del algoritmo Ciaccia-Patella desde 2^{10} a 2^{16}, esto se utilizó para ver en mayor detalle la comparativa de tiempo

presente en la experimentación, se puede ver que existe un crecimiento exponencial con el aumento de los datos.

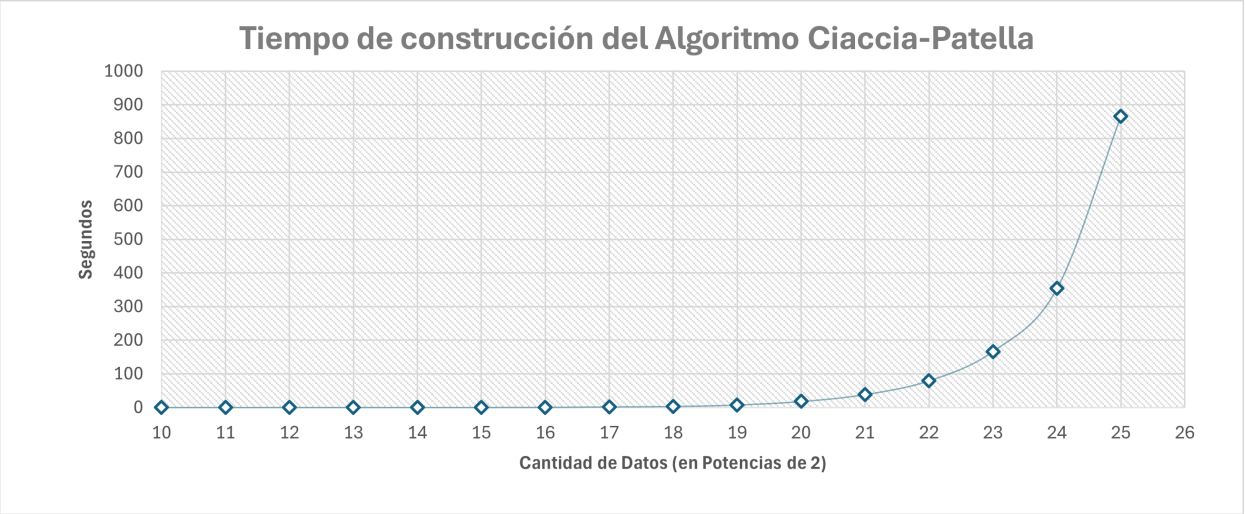

Figura 3: Tiempo de construcción de Ciaccia-Patella, para $n \in \{2^{10}, ..., 2^{25}\}$

En la presente figura, la relación exponencial vista anteriormente se mantiene a lo largo de los experimentos, aumentando a medida que el `input` crece.

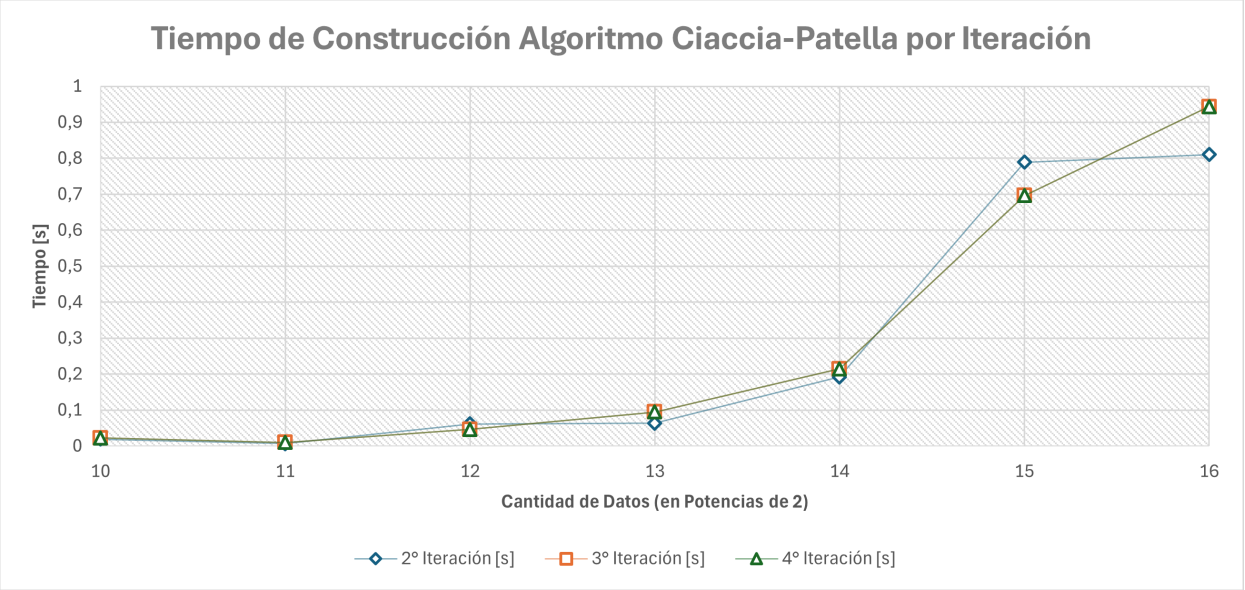

Figura 4: Tiempo de construcción de Ciaccia-Patella, para cada iteración, con $n \in \{2^{10}, ..., 2^{16}\}$

Como se mencionó dentro del apartado, el experimento de Ciaccia-Patella fue iterado en cuatro ocasiones y en esta imagen, se ve la comparativa desde la segunda iteración hasta la cuarta y última, notando que existe un comportamiento temporal muy similar entre la tercera y la última al menos en el subconjunto de $[2^{10}, 2^{16}]$ datos.

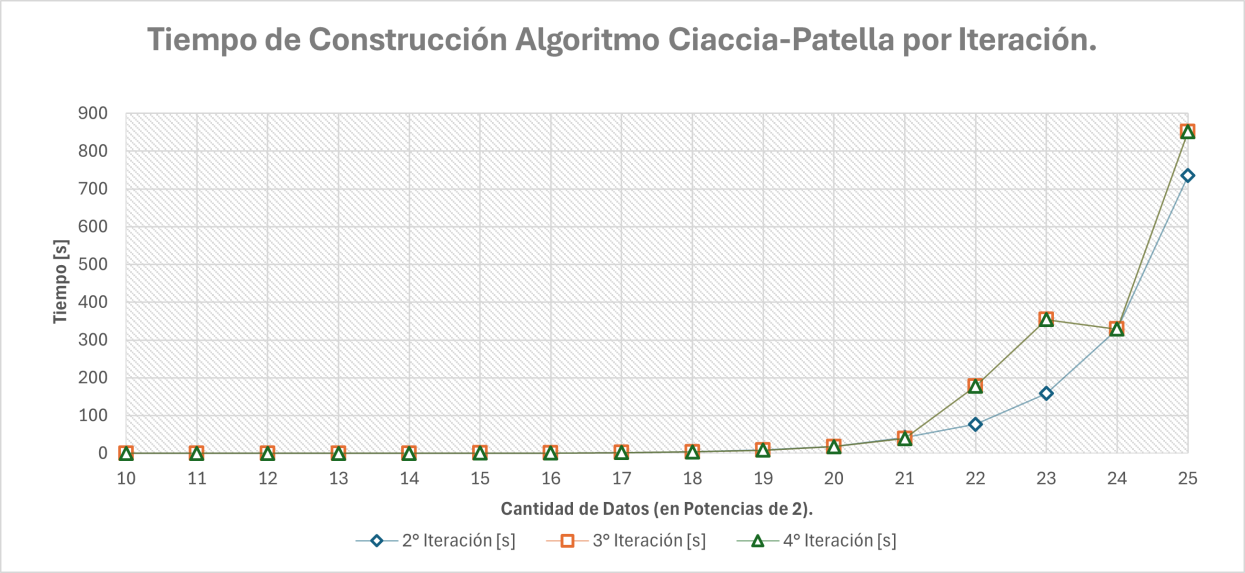

Figura 5: Tiempo de construcción de Ciaccia-Patella, para cada iteración, con $n \in \{2^{10}, ..., 2^{25}\}$

Luego, se comparó en la totalidad de datos los experimentos, viendose así la confirmación de la similitud en todo el conjunto de experimentación tendiendo a una curva exponencial, sin embargo entre $[2^{23}$ y $2^{24}]$, se presenta una anomalía en esa curva, la segunda iteración en este caso fue la más eficiente en tiempo mirando su curva.

Ahora, viendo el algoritmo Sexton-Swinbank:

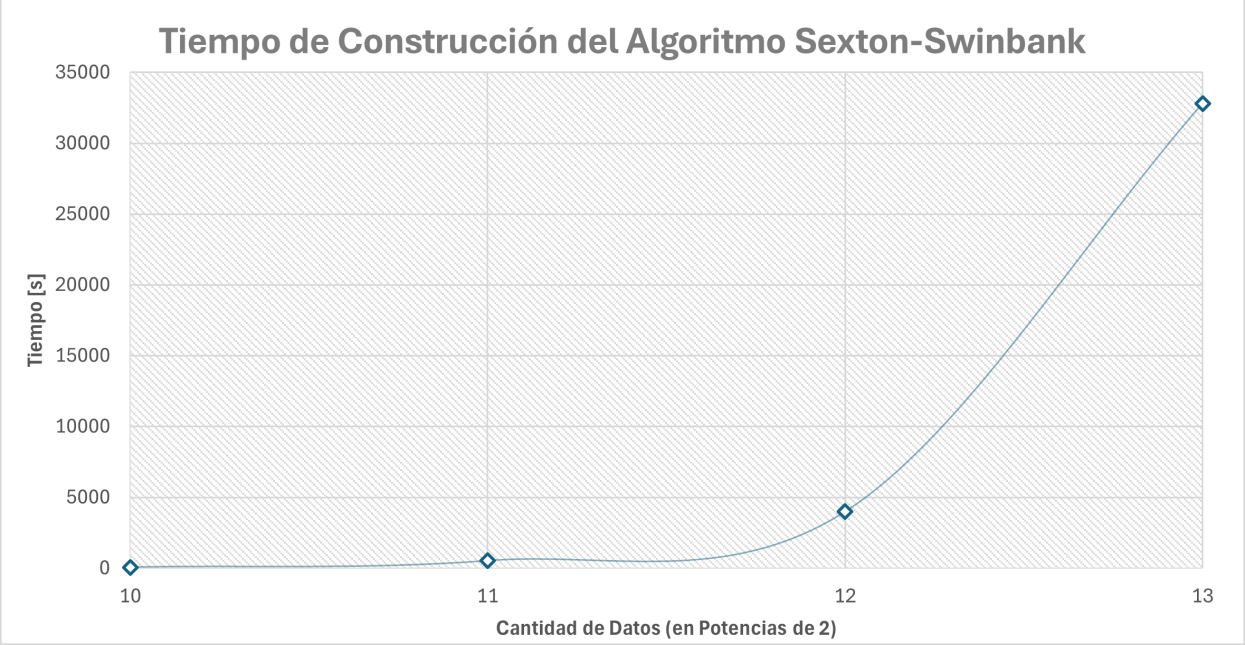

Figura 6: Tiempo de construcción para Sexton-Swinbank, con $n \in \{2^{10}, ..., 2^{13}\}$

Se puede ver para este caso que el algoritmo,presenta un crecimiento exponencial a medida que aumenta el tamaño del `input`.

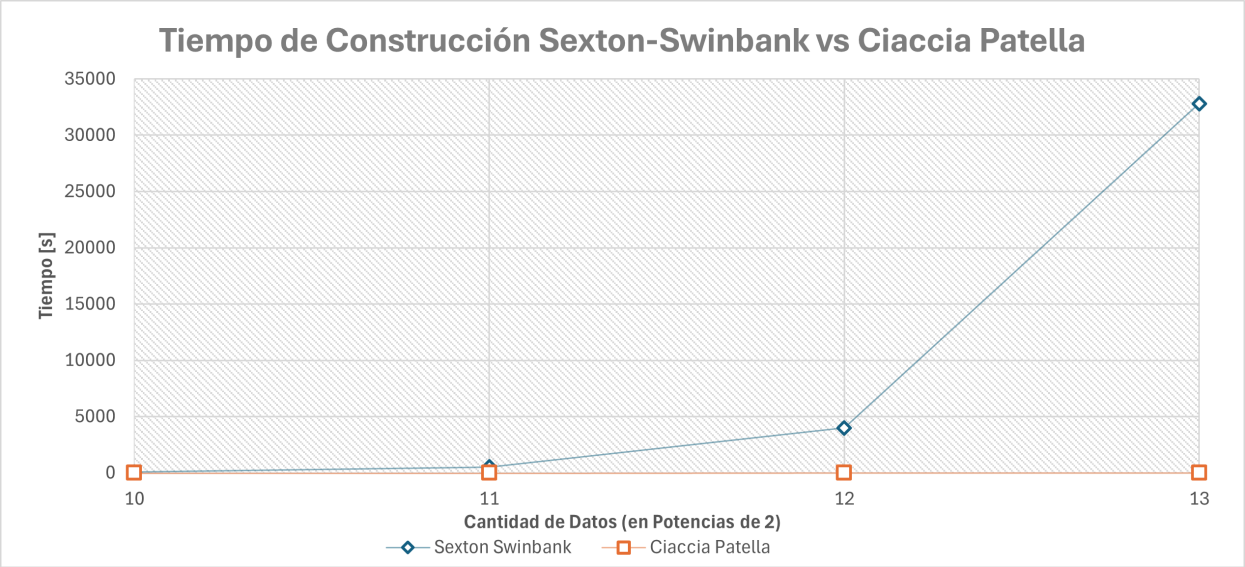

Figura 7: Comparativa de tiempos de construcción, para $n \in \{2^{10}, \ldots, 2^{13}\}$

Viendo los tiempos de construcción, se puede visualizar en la presente figura que Sexton-Swinbank aumenta de manera visiblemente superior a Ciaccia-Patella en la comparativa.

Ya se ha interpretado la diferencia de tiempos en construcción, a continuación se comparará en búsqueda.

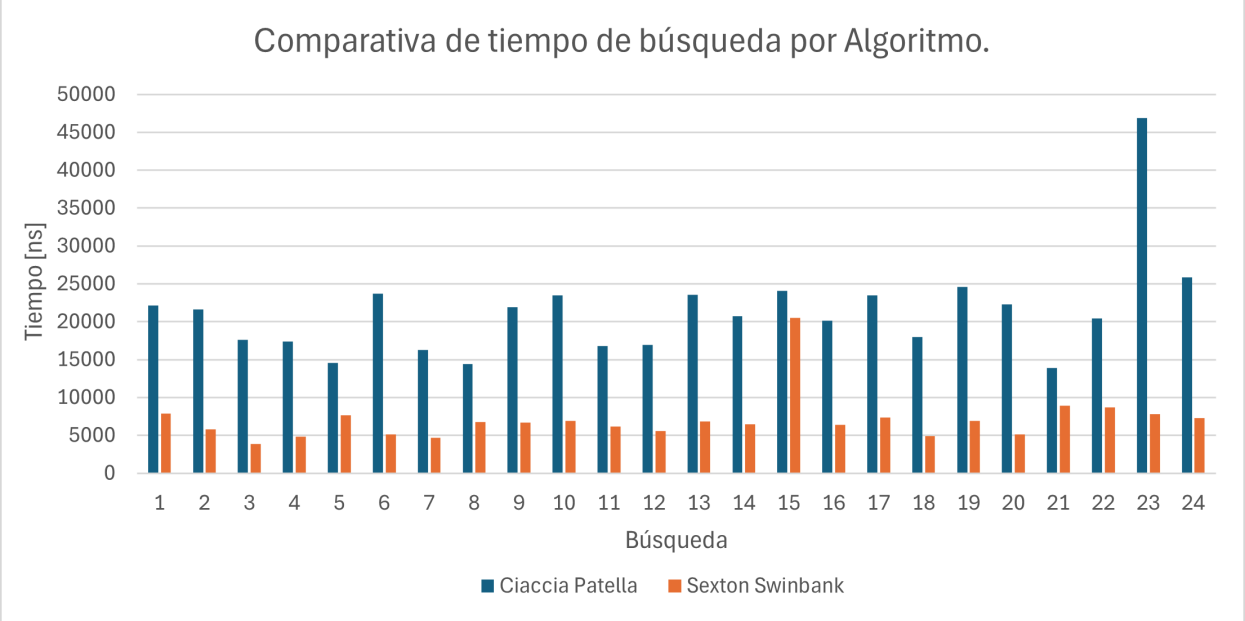

Figura 8: Comparativa de tiempo de búsqueda para los dos algoritmos, para las búsquedas $q = 2, \ldots, 25$ con $n = 2^{13}$

Dentro de este subconjunto de búsquedas, se puede interpretar la existencia de una diferencia entre ambos algoritmos, siendo Sexton-Swinbank el que posee en general menor tiempo de búsqueda dentro de las consultas vistas.

La selección de este intervalo, se debe a que se consideraron las primeras 25 búsquedas, sin embargo se presentó un aumento considerable de tiempo entre la primera búsqueda y las siguientes, por tanto se consideró como `outlier` y se grafica a continuación.

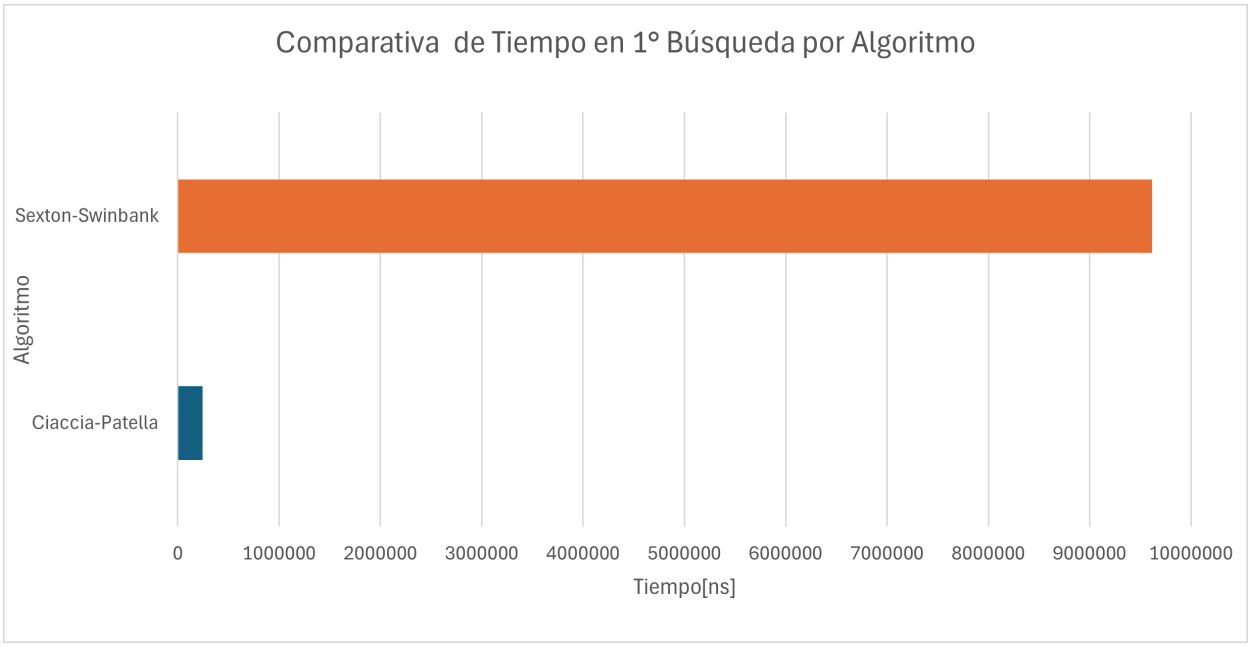

Figura 9: Comparativa del *outlier* obtenido para $n = 2^{13}$ en la primera búsqueda ($q = 1$)

Para este caso, Sexton-Swinbank posee un mayor tiempo de búsqueda con respecto a Ciaccia-Patella.

Una vez que se documenta la relación temporal, es importante notar los accesos a memoria de cada algoritmo según el caso.

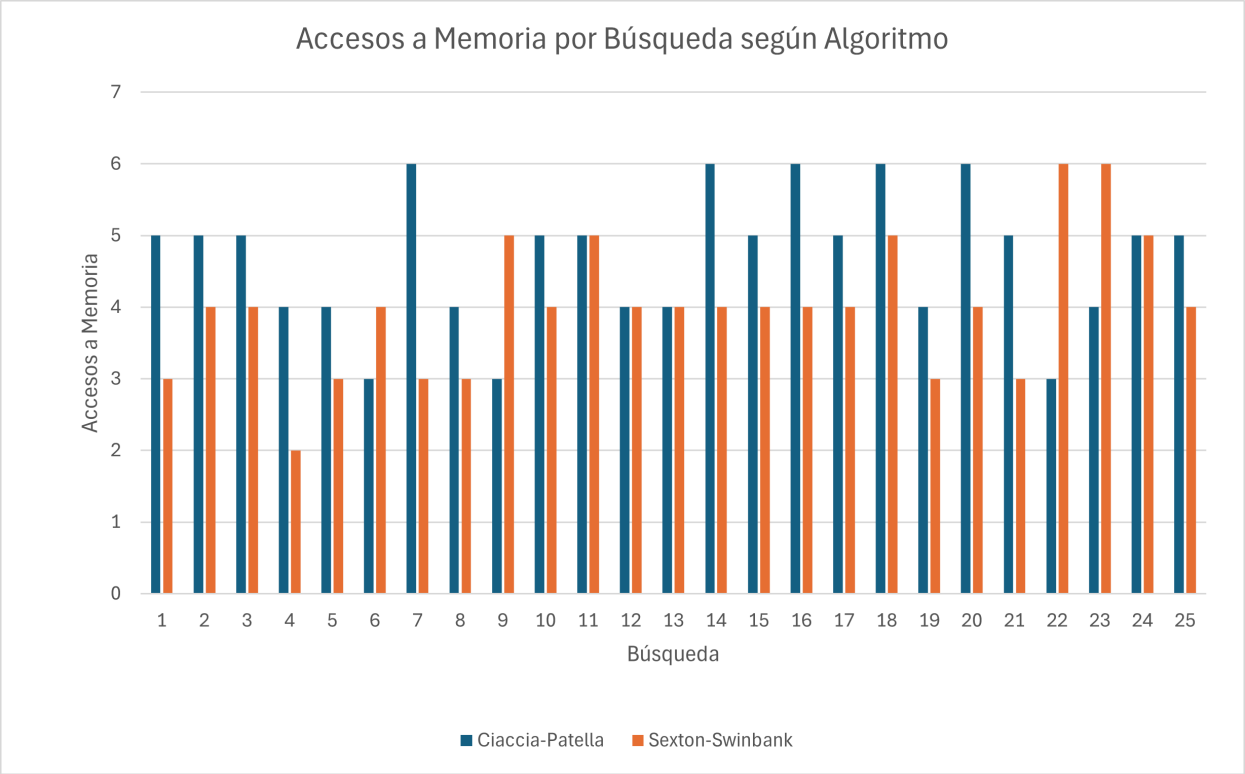

Figura 10: Comparativa de accesos a nodos para las búsquedas $q = 2, \ldots, 25$ con $n = 2^{13}$

En este caso, se puede visibilizar una clara diferencia entre accesos de memoria entre Ciaccia-Patella y Sexton-Swinbank, la cual es en mayoría a favor de Sexton-Swinbank.

Resultados

Para el análisis de los resultados, se puede notar que el método de Ciaccia-Patella es mucho más rápido en temas de construcción del M-Tree, sin embargo, el método de Sexton-Swinbank es más rápido en temas de búsqueda. Esto concuerda con lo investigado en el siguiente paper, pues intrínsecamente Sexton-Swinbank nace como una "extensión" de Ciaccia-Patella para optimizar las *queries*, mas no la construcción.

Se puede inferir que esta diferencia radica justamente en el método de construcción, pues también se observa que Ciaccia-Patella realiza más consultas a la hora de buscar sobre el árbol, esto implica un mayor tiempo de búsqueda.

De la Figura 2 y Figura 3 se puede inferir que para valores de $n = 2^k$ con $k > 25$ el algoritmo Ciaccia-Patella no permitirá una ejecución en un tiempo razonable, dado el crecimiento que presenta la función de complejidad. Además, tiene sentido que este comportamiento sea no acotado en función del tiempo, pues hay pasos del algoritmo que son al menos polinómicos en función de la entrada.

En la Figura 4 y Figura 5 no se ven grandes diferencias en el tiempo de construcción de Ciaccia-Patella con respecto a la iteración realizada, entendiéndose por iteración una nueva ejecución con las 100 búsquedas solicitadas. Esto es porque el algoritmo posee una complejidad fija, independiente de cuándo se ejecute. Las diferencias que se ven pueden ser porque el tiempo de ejecución sí depende de los recursos disponibles de la máquina en un cierto instante temporal, entonces, si se puede ejecutar dándole la máxima cantidad de recursos del computador, el tiempo de ejecución debería ser menor.

De la Figura 6 y Figura 7 se puede inferir que el algoritmo de Sexton-Swinbank es mucho más ineficiente en temas de construcción, lo que confirma nuestra hipótesis. Esta diferencia se da porque el método de Sexton-Swinbank prioriza las búsquedas por sobre la construcción, en el sentido que los recursos usados para maximizar la distribución óptima de puntos en el árbol hace que ejecutar consultas por sobre esta estructura sea menos costoso.

En la Figura 8 se confirma lo del párrafo anterior, es decir, Sexton-Swinbank sacrifica recursos en construcción para que la búsqueda sea más fácil.

La Figura 9 muestra un *outlier* para la primera búsqueda en el árbol (un tiempo muy grande). Esto se debe a que la primera vez que se llama a una función (en particular, a la función de búsqueda) hay una serie de pasos de interpretación que para llamadas que vienen después no ocurren. Este comportamiento en particular no se dio sólamente con tendencia hacia Sexton-Swinbank como se puede visualizar, sino que se dio para ambos. En particular, dicha figura es sólo un ejemplo.

De la Figura 10 se puede inferir que, si bien Sexton-Swinbank realiza búsquedas sobre el M-Tree con un tiempo de ejecución menor, en cuanto a accesos a memoria existe un balance, o no hay gran diferencia, con respecto a los accesos que realiza Ciaccia-Patella, sin embargo, esta diferencia existe.

Conclusiones

Luego de todas las implementaciones y su comparativa tanto en tiempo como memoria, se confirma la hipótesis inicial, en efecto Ciaccia-Patella es el algoritmo más eficiente en términos de tiempo para construir este tipo de estructuras, sin embargo a nivel de accesos a memoria y tiempo de búsqueda, Sexton-Swinbank presenta una mejoría con respecto a este.

Se estima que esta diferencia se debe a las formas que ambos algoritmos poseen para particionar los datos del `input`, ya que Sexton-Swinbank particiona de manera más costosa en tiempo, pero más eficiente en memoria por su política de *clustering* basada en *min-max split policy* vista anteriormente, la cual permite la optimización en búsqueda.

Además, se concluye que el algoritmo Sexton-Swinbank implementado por el equipo posee diferentes formas de optimización para reducir la complejidad temporal de este, dentro de ello, se sugiere optimizar la búsqueda del punto más cercano, ya que este puede ser $O(n \log n)$ si es óptimamente implementado.

Cerrando así, que para M-Trees, la forma más eficiente de construirlos es Ciaccia-Patella, sin embargo la forma más eficiente de almacenarlos es Sexton-Swinbank, pues las búsquedas o consultas tardan menos.