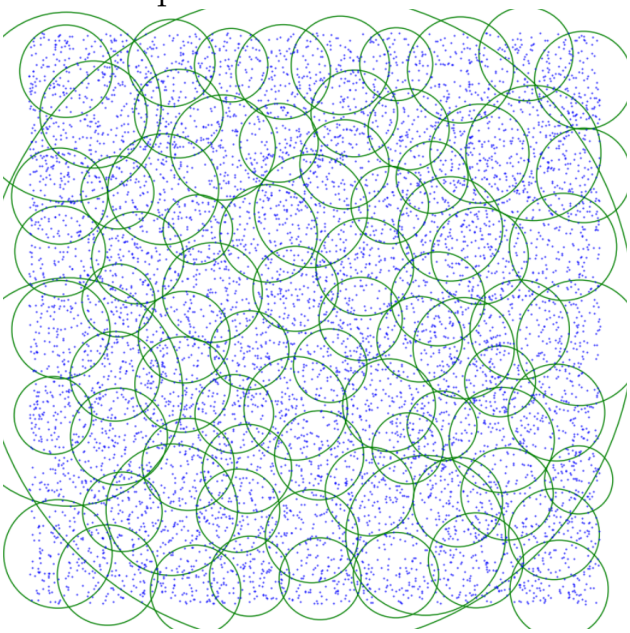
Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 1

Implementación M-Trees



Integrantes: Jean Duchens
Franco Gonzalez
Edgar Morales
Profesores: Gonzalo Navarro
Benjamin Bustos
Auxiliares: Diego Salas
Sergio Rojas

Fecha de realización: 13 de mayo de 2024
Fecha de entrega: 13 de mayo de 2024
Santiago de Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

El M-Tree, o árbol métrico, es una estructura de datos dinámica y balanceada, diseñada para optimizar el almacenamiento y la recuperación de objetos en un espacio métrico. Este tipo de árbol es especialmente útil para manejar conjuntos de datos donde las relaciones entre objetos se pueden definir a través de una métrica que cumpla con la desigualdad triangular. En el contexto de esta investigación, los objetos de estudio son puntos en un plano bidimensional, y la métrica utilizada para medir la distancia entre ellos es la distancia euclidiana.

El rendimiento del M-Tree depende críticamente del grado de superposición entre las regiones espaciales representadas por sus nodos. La minimización de esta superposición es fundamental y constituye un eje central en el diseño y las optimizaciones implementadas en el M-Tree y sus variantes.

En este informe, se explorarán dos métodos distintos de construcción de M-Trees: el Método Ciaccia-Patella (CP) y el Método Sexton-Swinbank (SS). Basados en el análisis de pseudocódigos específicos para cada método, se formuló la hipótesis que a grandes volúmenes de datos el Método SS podría exhibir un rendimiento inferior al Método CP debido a diferencias en sus estrategias de construcción y manejo de superposiciones.

A continuación, se detallarán las implementaciones de ambos métodos en el lenguaje de programación C++. Se analizarán los algoritmos y las estructuras de datos utilizadas, para luego proceder a una comparación exhaustiva de su rendimiento, en tiempo de ejecución, los accesos a disco (I/0's) de forma simulada (trabajaremos en RAM) y tiempos de búsqueda. Esta evaluación se realizará considerando varios tamaños de entrada, lo que permitirá validar la hipótesis planteada y obtener conclusiones sobre la eficacia relativa de estos métodos en la construcción de M-Trees.

2. Presentación de Algoritmos y Estructuras

2.1. Implementación y Funcionalidad

2.1.1. Estructura del M-Tree

- **Uso de Vectores:** Los vectores son utilizados por su capacidad para ajustar dinámicamente su tamaño, lo cual es esencial para manejar la variabilidad en el número de entradas por nodo a medida que el árbol crece o se modifica. Además, ofrecen accesos rápidos a los elementos, lo que es crucial para operaciones como inserciones y eliminaciones.
- **Node (Nodo):** Los nodos del M-Tree almacenan un vector de entradas, cada una de las cuales consta de un punto, un radio cobertor, y un puntero a un nodo hijo. La altura de cada nodo, denotada por h, es crucial para determinar la profundidad del árbol y facilitar la búsqueda por niveles.
- **Entry (Entrada):** Las entradas encapsulan tres componentes críticos:
  1. **Punto (p):** Representa el dato o la referencia espacial que gestiona el nodo.
  2. **Radio Cobertor (cr):** Esencial para determinar la necesidad de explorar un subárbol durante las búsquedas, indica la distancia máxima dentro del subárbol.
  3. **Puntero al Nodo Hijo (a):** Enlaza estructuralmente al nodo padre con sus nodos hijos, manteniendo la integridad del árbol.
- **M-Tree:** Puntero al nodo raíz del M-Tree.

2.1.2. Búsqueda en M-Tree Utilizando BFS

BFS es adecuado para navegar por el árbol nivel por nivel, garantizando que todos los nodos a una profundidad específica sean examinados antes de avanzar al siguiente nivel. Esto resulta conveniente en los M-Trees, donde la distancia entre puntos (definida por el radio cobertor) es determinante para decidir la exploración de un nodo hijo.

Implementación de BFS: En la práctica, BFS se implementa utilizando una cola para gestionar los nodos que deben explorarse. Esta técnica es compatible con la dinámica de los vectores en C++, facilitando la inclusión y exclusión de nodos en la cola de manera eficiente.

2.2. Métodos de construcción del M-Tree

Se presentaron dos algoritmos de construcción para la estructura, uno propuesto por Paolo Ciaccia y Marco Patella (Ciaccia-Patella), y otro propuesto por Alan P. Sexton y Richard Swinbank (Sexton-Swinbank), ambos se utilizan para la creación de M-Trees, pero con distintas finalidades.

2.2.1. Método Ciaccia-Patella (CP)

Este método se centra en el clustering de puntos y la construcción eficiente de un M-Tree, asegurando que cada nodo esté dentro de las capacidades permitidas y manteniendo un equilibrio estructural adecuado. Para esto se utilizan las siguientes funciones:

- **bulk_loading:** Esta función es responsable de generar un M-Tree de forma eficiente a partir de un conjunto de entries, utilizando un proceso que implica la selección de muestras, la creación de subárboles, la redistribución de puntos y el balanceo de alturas para garantizar una estructura de árbol óptima. Primero realiza ciclos, dentro de un while, para la redistribucion de puntos entre grupos aleatorios, terminando solo cuando se consiguen más de 2 conjuntos con tamaño mayor a b, para esto se hace uso de las funciones de asignacion y redistribución de puntos. El primer filtro, que elimina árboles con menos de los puntos necesarios, simplemente revisa, con un for, aquellos hijos que sirven y los agrega al conjunto. El segundo filtro utiliza la funcion find_node_h que retorna todos los subárboles que cumplen con la altura deseada, que es la menor de entre todos los hijos. Finalmente, luego de hacer la llamada recursiva para el conjunto F, agregar los T' simplemente consiste en conectar el las hojas los subárboles correspondientes, lo cual se hace con ayuda de funciones auxiliares que permiten encontrar los valores correspondientes efficientemente.

- **assign_to_nearest:** Función auxiliar encargada de realizar la asignación a los puntos al sample mas cercano.

- **redistribution:** Reasigna puntos de los conjuntos que tienen menos de la mitad de la capacidad B, lo cual ayuda a mantener un equilibrio adecuado en el tamaño de los nodos y asegura que todos los nodos estén suficientemente poblados para optimizar el rendimiento de búsqueda.

- **find_node_h:** Realiza un recorrido en anchura (BFS) en el M-Tree, buscando subárboles que tengan una altura h, examina cada nodo en el árbol y verifica si su altura coincide. Si es así, añade el nodo junto con su punto asociado a un vector de salida. En caso contrario, explora los subárboles del nodo y continúa la búsqueda. Al final, devuelve un vector que contiene todos los subárboles encontrados con la altura deseada, junto con el punto que representa al subárbol.

- **append_to_leaf:** Esta función empareja cada subárbol en un conjunto con su hoja correspondiente en una lista ordenada de entradas, usando un punto de referencia. Primero ordena los puntos de las entradas en base a sus coordenadas, luego se busca en el vector de entradas, utilizando una búsqueda binaria, la entrada a la que corresponde el punto representante del subárbol, luego simplemente se conecta el subárbol con la entrada.

- **update_radius, update_height:** Funciones que facilitan el mantenimiento de la estructura del árbol, desde la conexión de nodos de la misma altura hasta la actualización de radios y alturas tras modificaciones. Estas funciones se aprovechan de otras como get_all_entries para obtener facilmente la información del T_{sup} y de esta forma no perder la información de los nodos y entradas originales. Esto permite que al actualizar la información de radio y altura solo se tenga que actualizar para aquellos que cambiaron, que son aquellos que estaban originalmente en T_{sup}.

- **get_all_entries, get_all_leaf_entries, get_all_levels:** Proporcionan herramientas para extraer y organizar información sobre la estructura del árbol, facilitando el acceso y la optimización de los nodos y entradas. get_all_leaf_entries retornará punteros a todas las entradas de las hojas de T_{sup}, útil para conectar los T' a T_{sup}. get_all_entries retornará todas las entradas, útil para calcular los radios solo a las entradas del T_{sup} original. get_all_levels retornará todos los nodos separados por su altura, de forma que al actualizar las alturasa se haga desde las hojas hasta la raíz del T_{sup} original.

2.2.2. Método Sexton-Swinbank (SS)

Este método se centra en la gestión dinámica de clusters y en la minimización de los radios cobertores mediante una política de división MinMax, lo cual es crucial para datos que naturalmente forman grupos densos.

- **ss_algorithm:** Función encargada de la construcción completa del M-Tree. Si el conjunto inicial de puntos es pequeño, directamente crea un nodo hoja. Si no, organiza los puntos en clusters y procede a construir el árbol de manera recursiva, optimizando la estructura para búsquedas eficientes. Este algoritmo coordina la transición entre diferentes fases de la construcción del árbol, desde la creación de nodos hoja hasta la consolidación de nodos internos.

- **output_hoja:** Responsable de crear un nodo hoja en el M-Tree a partir de un cluster de puntos. Esta función establece el medoide del cluster como el punto de referencia del nodo y calcula el radio cobertor en base a la distancia más grande entre el medoide y otros puntos del cluster. Este nodo hoja encapsula los puntos de manera que se minimice el radio cobertor, facilitando búsquedas más eficientes.

- **output_interno:** Construye un nodo interno a partir de varios subárboles o entradas. Utiliza el medoide del conjunto de entradas como punto de referencia y ajusta los radios

cobertores para abarcar adecuadamente todos los subárboles incluidos. Esta función es clave para mantener la integridad estructural y la eficiencia del árbol durante las operaciones de búsqueda y navegación.

- **create_clusters:** Inicialmente considera cada punto como un clúster individual y, progresivamente, combina estos clusters basándose en la proximidad de sus medoides. Este proceso es iterativo y continúa hasta que el número de clusters es manejable o cumple con los criterios especificados para la formación de nodos. La eficacia de esta función es crucial para el éxito del método SS, ya que una buena formación de clusters reduce significativamente los radios cobertores necesarios. Para aumentar la eficiencia, se mantienen guardados los valores de los medoides primarios de cada clusters para evitar tener que volver a calcularlos para cada comparación, y solo se calcula el medoide de un cluster cuando se hace merge de 2 clusters y se necesita agregar este nuevo cluster al conjunto C.

- **find_medoid:** Identifica el medoide de un cluster, que es el punto cuyo nombramiento como centro minimiza el radio cobertor del cluster. Esta selección es vital para reducir la distancia promedio a otros puntos en el cluster, optimizando la estructura del árbol para consultas rápidas.

- **closest_neighbor:** Encuentra el cluster más cercano a un cluster dado, utilizando el medoide como referencia. Esta función es fundamental para poder encontrarle el cluster más cercano al último cluster del conjunto C.

- **closest_clusters:** Implementa un algoritmo modificado de dividir y reinar para identificar el par de clusters más cercanos dentro de un conjunto, pasando de complejidad $O(n^2)$ (fuerza bruta) a $O(nlogn)$. Este algoritmo es necesario para el algoritmo de creación de clusters, ya que se crean los cluster a partir de los 2 clusters más cercanos del conjunto. La función que entrega los puntos más cercanos es una modificación de un algoritmo de closest-pair.

- **point_bin_search:** Esta función optimiza la búsqueda de posiciones dentro de listas ordenadas de puntos o entradas. Es crucial para el manejo eficiente de las estructuras de datos ordenadas en el M-Tree, especialmente durante la inserción de nuevos puntos o la reorganización de entradas existentes.

Cada una de estas funciones contribuye significativamente a la eficacia del método SS en la construcción y mantenimiento de M-Trees, asegurando que el árbol esté bien balanceado y sea eficiente para consultas basadas en la proximidad.

2.2.3. Comparación

Tanto CP como SS son métodos diseñados para la eficiente construcción de M-Trees, si bien se destacan en diferentes tipos de datos. CP es óptimo para datos uniformemente distribuidos, permitiendo construcciones rápidas, mientras que SS sobresale en el manejo de

datos agrupados, ofreciendo tiempos de búsqueda optimizados y una gestión eficiente de los clusters.

El algoritmo CP inicia la construcción del árbol desde la raíz y procede de manera recursiva para generar los subárboles, utilizando conjuntos aleatorios. Su complejidad radica en la necesidad de filtrar y reconectar los subárboles generados de forma recursiva.

Por otro lado, el algoritmo SS comienza desde la base del árbol y construye el árbol hasta llegar a la raíz. Aunque su implementación es menos compleja, es computacionalmente más costoso que CP debido a operaciones más intensivas, como la búsqueda de los dos puntos más cercanos.

3. Resultados Experimentales

Entorno de Ejecución: Los experimentos se realizaron en una computadora con sistema operativo GNU/Linux, equipada con 16 GB de memoria RAM y 12 MB de memoria caché. Las visualizaciones y análisis se llevaron a cabo utilizando Python y Jupyter Notebooks.

Datos Utilizados: Los datos para los experimentos se generaron mediante funciones específicas diseñadas para crear puntos aleatorios y conjuntos de datos reproducibles. A continuación, se describen las funciones principales utilizadas:

- **init_random()**: Inicializa un generador de números aleatorios (`std::mt19937 gen`) usando una semilla proporcionada por el dispositivo si no ha sido inicializado antes. Utiliza `std::random_device rd;` para generar un número aleatorio no determinístico que se usa como semilla para `std::mt19937 gen(rd());` y también para `srand(rd());`, que establece la semilla del generador de números aleatorios (`rand()`). Asegura que la inicialización solo ocurra una vez mediante el uso de una variable estática `random_init`.

- **random_point()**: Genera un punto aleatorio utilizando la función `rand()` para crear valores de x e y. Los valores generados son floats entre 0 y 1, obtenidos dividiendo `rand()` por `RAND_MAX-1`. Devuelve un punto (`Point`) con las coordenadas x e y generadas aleatoriamente.

- **random_sample_generator(int size, int seed)**: Genera un vector de puntos aleatorios de tamaño especificado (`size`). Utiliza una semilla específica (`seed`) para inicializar el generador de números aleatorios con `srand(seed);`, lo que permite la reproducibilidad de los resultados. Dentro de un bucle, genera cada punto aleatorio como en `random_point()` y los almacena en un vector.

- **random_k_sample(std::vector<Point> points, int k)**: Selecciona aleatoriamente `k` puntos de un vector de puntos dado. Utiliza `std::shuffle()` para barajar el vector de puntos de manera aleatoria. `std::shuffle()` requiere un generador de números aleatorios, en este caso, se utiliza el `std::mt19937 gen` inicializado en `init_random()`. Devuelve un nuevo vector que contiene los primeros `k` elementos del vector barajado, que son la selección aleatoria de puntos.

Procedimiento Experimental: Cada experimento se repitió varias veces para asegurar la precisión de los resultados. Los tests se realizaron con tamaños de muestra desde 2^{10} hasta 2^{25} para el método CP y hasta 2^{18} para el método SS, debido a limitaciones que se discutirán en secciones posteriores.

```
Método CP

Size Sample = 2^10, I/OS Mean:     4.01, I/OS 95% CI: (    3.954963,     4.062615), Size Mean:     1.26
Size Sample = 2^11, I/OS Mean:     4.62, I/OS 95% CI: (    4.569241,     4.673923), Size Mean:     2.50
Size Sample = 2^12, I/OS Mean:     5.14, I/OS 95% CI: (    5.077618,     5.200703), Size Mean:     5.06
Size Sample = 2^13, I/OS Mean:     5.59, I/OS 95% CI: (    5.524405,     5.656259), Size Mean:    10.07
Size Sample = 2^14, I/OS Mean:     7.95, I/OS 95% CI: (    7.876908,     8.029342), Size Mean:    20.24
Size Sample = 2^15, I/OS Mean:    10.91, I/OS 95% CI: (   10.810359,    11.003117), Size Mean:    40.42
Size Sample = 2^16, I/OS Mean:    13.23, I/OS 95% CI: (   13.118627,    13.331568), Size Mean:    80.99
Size Sample = 2^17, I/OS Mean:    15.61, I/OS 95% CI: (   15.489634,    15.738881), Size Mean:   161.89
Size Sample = 2^18, I/OS Mean:    19.99, I/OS 95% CI: (   19.867320,    20.112172), Size Mean:   323.75
Size Sample = 2^19, I/OS Mean:    27.42, I/OS 95% CI: (   27.259016,    27.580827), Size Mean:   647.36
Size Sample = 2^20, I/OS Mean:    41.02, I/OS 95% CI: (   40.819129,    41.220910), Size Mean:  1294.48
Size Sample = 2^21, I/OS Mean:    64.35, I/OS 95% CI: (   64.064885,    64.625544), Size Mean:  2589.63
Size Sample = 2^22, I/OS Mean:   105.35, I/OS 95% CI: (  104.923386,   105.767044), Size Mean:  5179.46
Size Sample = 2^23, I/OS Mean:   181.77, I/OS 95% CI: (  181.109242,   182.440563), Size Mean: 10359.37
Size Sample = 2^24, I/OS Mean:   320.93, I/OS 95% CI: (  319.806004,   322.050442), Size Mean: 20722.91
Size Sample = 2^25, I/OS Mean:   590.23, I/OS 95% CI: (  588.256204,   592.212546), Size Mean: 41445.20
```

Figura 1: Resultados del Método CP.

```
Método SS

Size Sample = 2^10, I/OS Mean:     3.45, I/OS 95% CI: (    3.407751,     3.483850), Size Mean:     1.26
Size Sample = 2^11, I/OS Mean:     3.85, I/OS 95% CI: (    3.806001,     3.887358), Size Mean:     2.50
Size Sample = 2^12, I/OS Mean:     4.14, I/OS 95% CI: (    4.094115,     4.189089), Size Mean:     5.06
Size Sample = 2^13, I/OS Mean:     4.46, I/OS 95% CI: (    4.420952,     4.500923), Size Mean:    10.07
Size Sample = 2^14, I/OS Mean:     6.12, I/OS 95% CI: (    6.056827,     6.178524), Size Mean:    20.24
Size Sample = 2^15, I/OS Mean:     7.27, I/OS 95% CI: (    7.204514,     7.334548), Size Mean:    40.42
Size Sample = 2^16, I/OS Mean:     9.36, I/OS 95% CI: (    9.293858,     9.432705), Size Mean:    80.99
Size Sample = 2^17, I/OS Mean:    14.20, I/OS 95% CI: (   14.104634,    14.288921), Size Mean:   161.89
Size Sample = 2^18, I/OS Mean:    17.43, I/OS 95% CI: (   17.322933,    17.541325), Size Mean:   323.75
```

Figura 2: Resultados del Método SS.

Procederemos a mostrar algunas visualizaciones sobre las distintas métricas en ambos métodos para facilitar la comparativa.

3.1. Promedio de I/O's vs. Tamaño de Datos

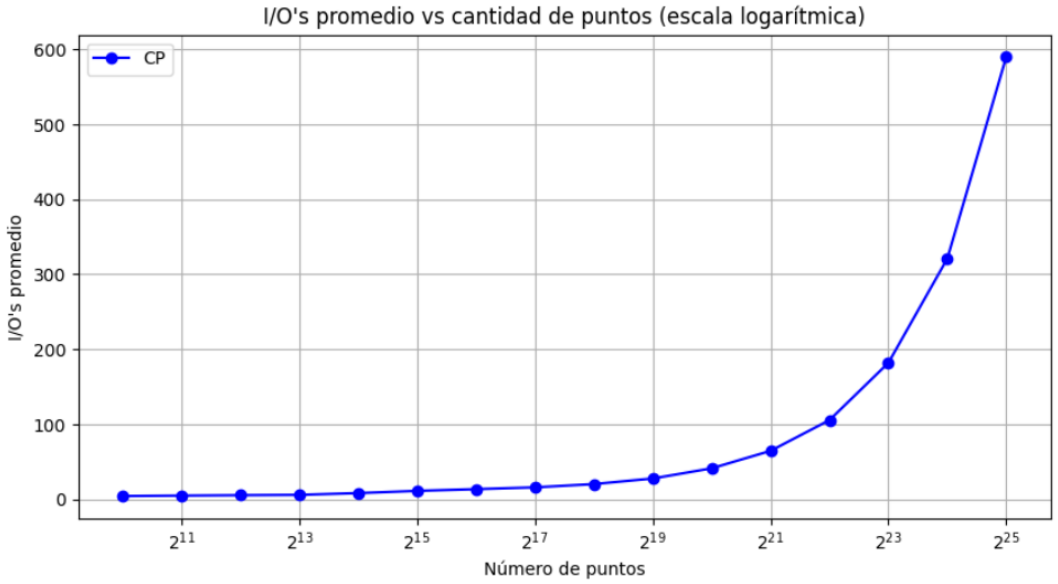


Figura 3: Promedio de I/O's con respecto al tamaño de datos para el Método CP.

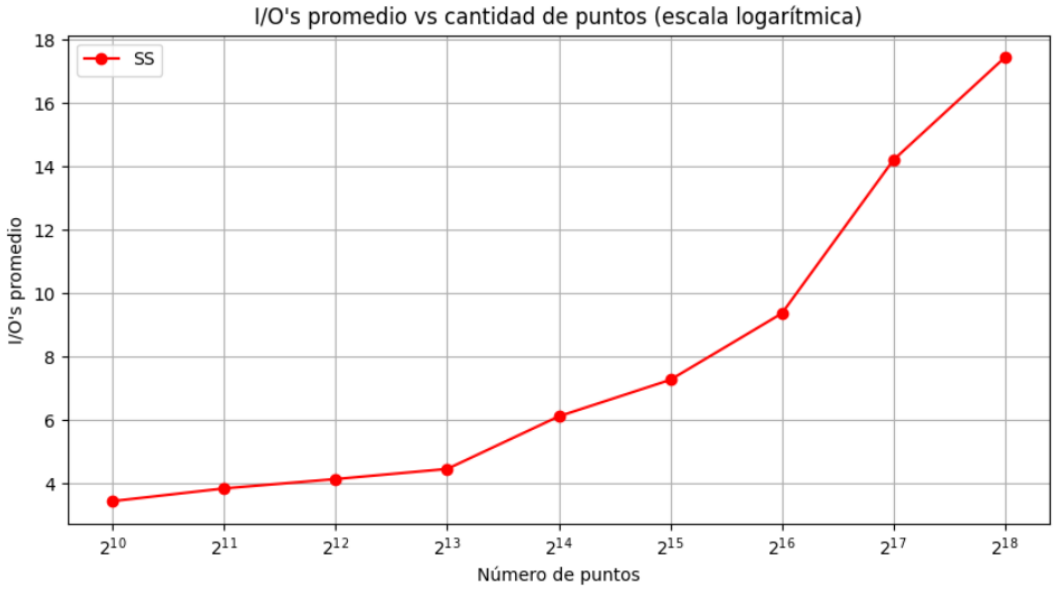


Figura 4: Promedio de I/O's con respecto al tamaño de datos para el Método SS.

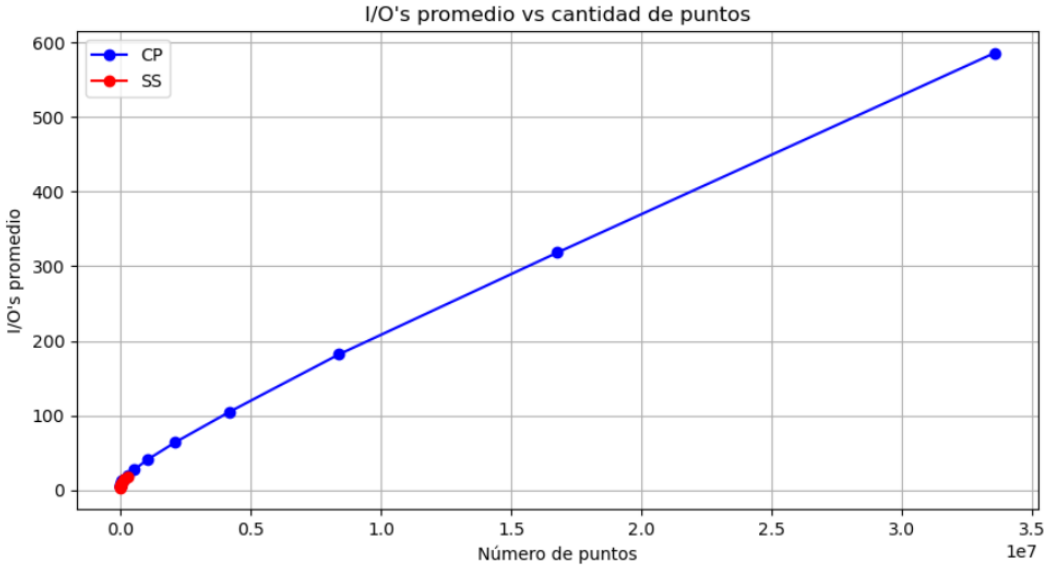


Figura 5: Promedio de I/O's con respecto al tamaño de datos para ambos métodos.

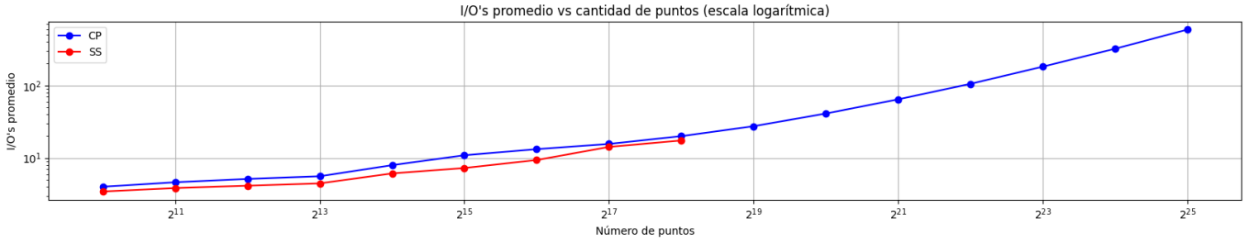


Figura 6: Comparativa de I/O's promedio vs. tamaño de datos para ambos métodos.

3.2. Cantidad de I/O's e Intervalos de Confianza

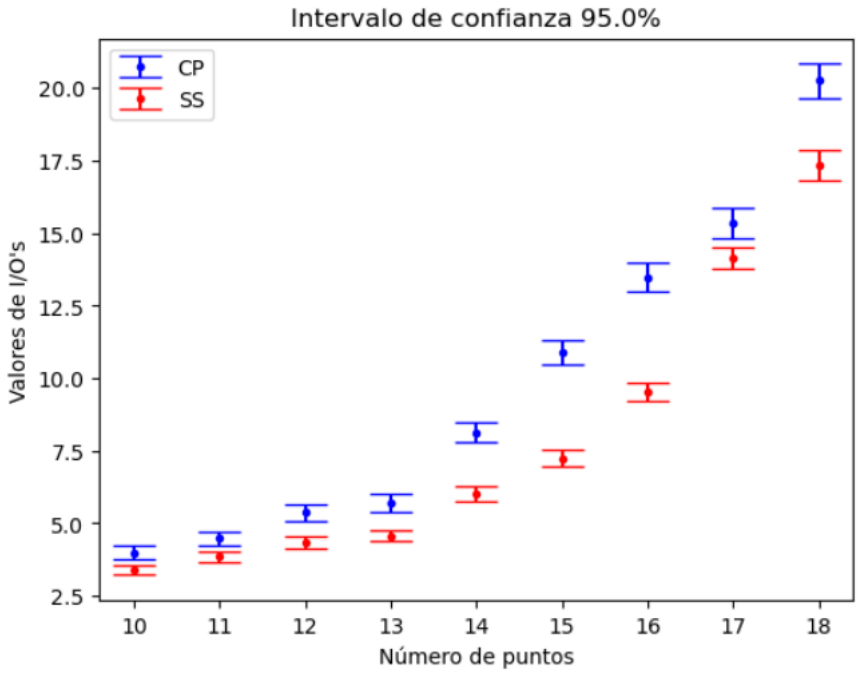


Figura 7: Cantidad de I/O's con respecto al número de puntos a un 95 % de confianza

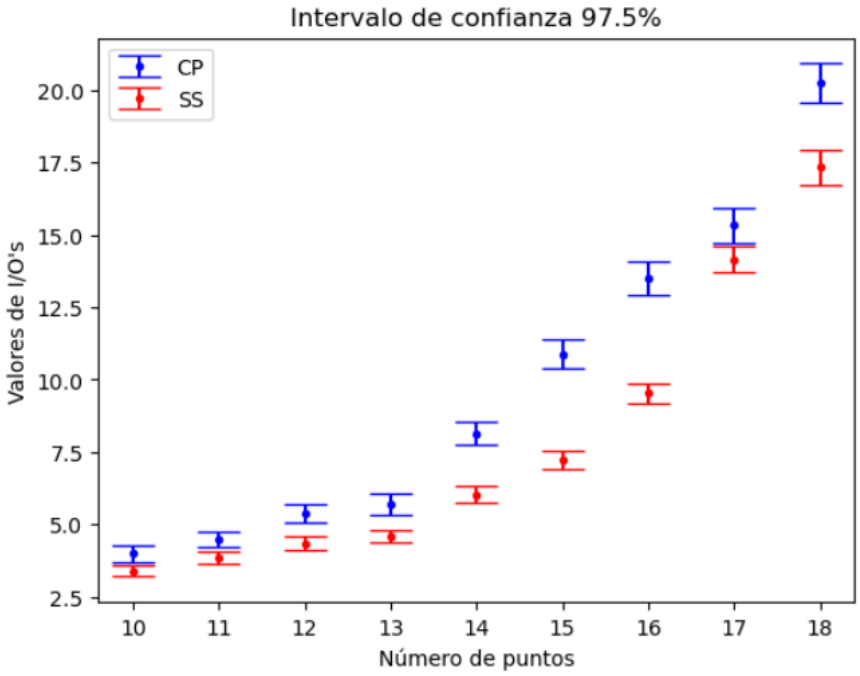


Figura 8: Cantidad de I/O's con respecto al número de puntos a un 97.5 % de confianza

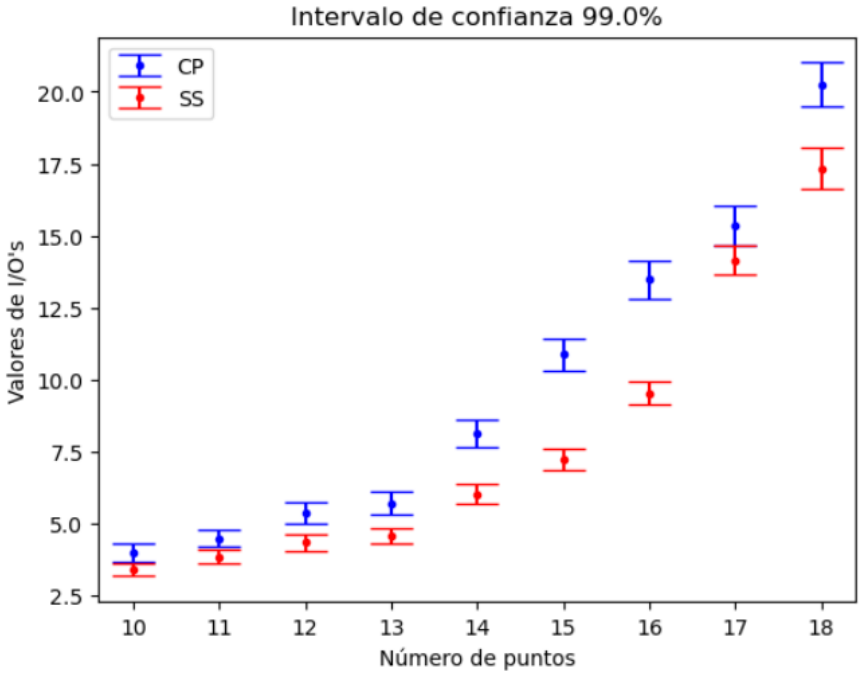


Figura 9: Cantidad de I/O's con respecto al número de puntos a un 99 % de confianza

3.3. Tiempo de ejecución vs Tamaño de Datos

Figura 10: Comparación en el tiempo de ejecución para ambos métodos

3.4. Tiempo de Búsqueda vs. Tamaño de Datos

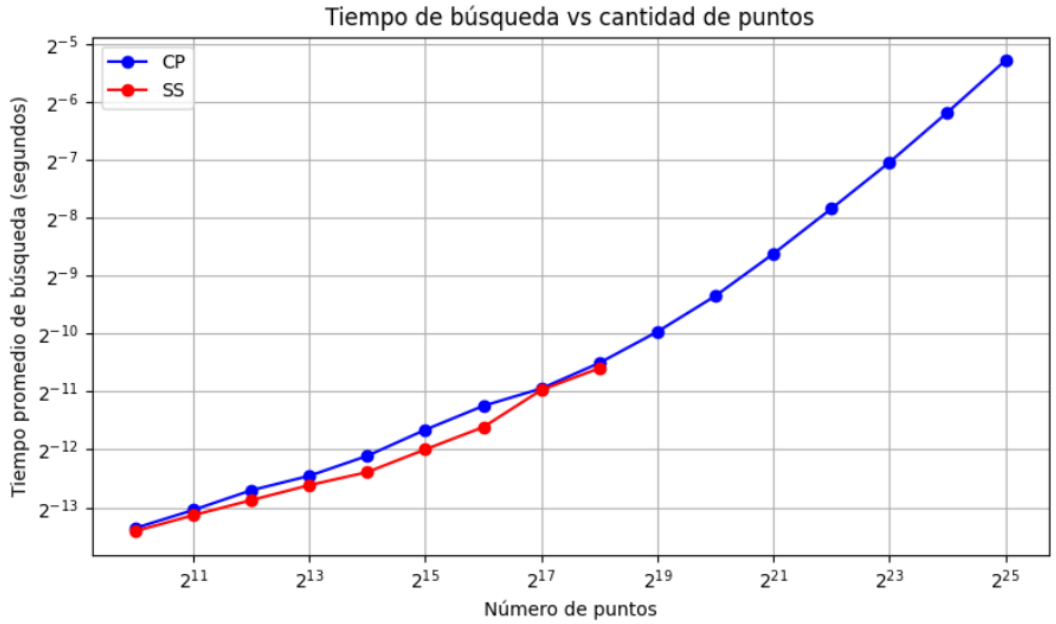


Figura 11: Comparación en el tiempo de búsqueda para ambos métodos

3.5. Análisis de Operaciones de Entrada/Salida (I/O's)

En los gráficos analizados, observamos diferentes comportamientos en las operaciones de entrada/salida (I/Os) entre los dos métodos evaluados. El Método CP exhibe un incremento más pronunciado en las I/Os a medida que aumenta el tamaño de los datos (ver Figura 3). En contraste, el Método SS muestra un crecimiento más moderado y lineal en las I/Os (ver Figura 4).

Además, al evaluar ambos métodos, se aprecia que las I/Os para ambos métodos crecen de manera logarítmica, lo que refleja un aumento progresivo pero controlado en la carga de acceso a disco con el aumento en el tamaño de los datos (ver Figura 5).

Al comparar el promedio de las I/Os entre los dos métodos, se puede concluir que el Método SS mantiene un promedio más bajo de I/Os a lo largo de las pruebas realizadas, demostrando una eficiencia superior en la gestión de recursos del sistema frente al Método CP (ver Figura 6).

3.5.1. Análisis de Intervalos de Confianza

Los intervalos de confianza proporcionan una medida estadística de la precisión de las estimaciones del rendimiento medio de los métodos CP y SS. Observamos que los intervalos de confianza para el método SS son consistentemente más estrechos que para el método CP a través de diferentes tamaños de muestra.

Al observar los intervalos de confianza al 97.5 % para las operaciones de I/Os, notamos que los intervalos para el método SS siguen siendo más estrechos en comparación con los del método CP a través de los diferentes tamaños de muestra. Además, los intervalos de confianza de SS no se superponen con los de CP en puntos críticos, especialmente para muestras más grandes.

3.6. Comparación de Tiempos de Construcción

La comparativa de tiempos de construcción versus el tamaño de datos revela diferencias significativas entre los dos métodos. Mientras que el Método CP muestra una escalabilidad razonable hasta 2^{25}, el Método SS experimenta un aumento exponencial en el tiempo de construcción que comienza mucho antes, alrededor de 2^{17} (ver figura 10).

3.7. Análisis de Tiempos de Búsqueda

El análisis de los tiempos de búsqueda en relación con el tamaño de los datos revela que ambos métodos, CP y SS, experimentan un aumento en los tiempos de búsqueda a medida que crece el tamaño de los datos. El gráfico correspondiente muestra un incremento casi lineal en el tiempo de búsqueda para ambos métodos. No obstante, el Método SS requiere consistentemente menos tiempo en comparación con el Método CP en los tamaños de datos evaluados (ver Figura 11).

Además, es relevante comentar sobre el tamaño de los resultados retornados durante las búsquedas. A pesar del incremento en los tiempos de búsqueda con mayores volúmenes de

datos, las búsquedas efectuadas por ambos métodos son precisas, retornando aproximadamente el 0.12 % del total de los datos. Esto indica no solo que el tiempo de búsqueda aumenta con el tamaño del conjunto de datos, sino también que la eficiencia de la búsqueda en términos de precisión se mantiene, validando la efectividad de ambos métodos en el retorno de resultados pertinentes y específicos bajo diversas escalas de datos.

4. Comentarios y Conclusiones

A continuación se presenta un análisis detallado basado en los resultados obtenidos para los métodos CP y SS:

4.1. I/Os vs. Tamaño de Datos

El Método SS, mantiene un número menor de I/Os en comparación con el Método CP en todas las escalas de datos examinadas. Este comportamiento indica una mayor eficiencia y optimización en el manejo de accesos a datos.

Con un nivel de confianza del 97.5 %, la falta de superposición en los intervalos de confianza entre los métodos CP y SS sugiere que la diferencia en el rendimiento es estadísticamente significativa y no atribuible al azar. Por tanto, podemos concluir con mayor certeza que el Método SS supera al Método CP en términos de eficiencia de operaciones de I/O's. Además, los datos indican que muestra una mayor estabilidad y previsibilidad en su rendimiento a través de diferentes tamaños de muestra.

4.2. Tiempo de Construcción vs. Tamaño de Datos

Ambos métodos muestran un aumento en el tiempo de construcción a medida que el tamaño de los datos aumenta. Sin embargo, el Método SS muestra un incremento exponencial más pronunciado que el CP a partir de 2^{17}, lo que puede limitar su aplicabilidad para bases de datos de gran tamaño.

4.3. Tiempo de Búsqueda vs. Tamaño de Datos

Ambos métodos demuestran un incremento en el tiempo de búsqueda conforme crece el tamaño de los datos. No obstante, el Método SS generalmente requiere menos tiempo que el CP, lo cual es ventajoso para aplicaciones que necesitan una rápida recuperación de la información.

4.4. Inferencias de los Resultados

- **Escalabilidad:** El Método SS muestra limitaciones notables en su capacidad de escalabilidad, evidenciadas por su rendimiento hasta un tamaño máximo de 2^{18} en los conjuntos de datos procesados. Esta restricción podría atribuirse al proceso intensivo de calcular repetidamente el par de puntos más cercanos para determinar los medoides de los clusters. Dicha operación, que se realiza múltiples veces durante el proceso de clusterización, implica un coste computacional considerable, afectando así la eficiencia general del método al manejar grandes volúmenes de datos. En contraste, el Método CP, aunque menos eficiente, muestra una capacidad superior para manejar incrementos en el tamaño de los datos.

- **Optimización de Recursos:** La eficiencia en el uso de recursos, particularmente la memoria, es crucial para mejorar la escalabilidad de ambos métodos. Esto podría implicar la exploración de técnicas de compresión de datos o algoritmos de indexación más eficientes.
- **Aplicabilidad:** Dependiendo del tamaño del conjunto de datos y los requisitos de tiempo de respuesta, el Método SS podría ser preferible para conjuntos de datos más pequeños y medianos donde la eficiencia de búsqueda y la reducción de I/Os son críticas. Para conjuntos de datos muy grandes, el Método CP podría ser más apropiado, aunque con un costo de eficiencia.

Los resultados indican que mientras el Método SS es superior en eficiencia operativa para tamaños de datos moderados, enfrenta limitaciones significativas a medida que aumenta el tamaño de los datos. Por otro lado, el Método CP, aunque menos eficiente, ofrece mejor escalabilidad, haciendo necesario un equilibrio entre eficiencia operativa y capacidad de manejo de grandes volúmenes de datos en la elección del método más apropiado para aplicaciones específicas.

5. Recapitulación y Conclusiones Finales

5.1. Resumen de Actividades

En este estudio, hemos implementado y evaluado dos métodos para la gestión de clusters de datos: el Método CP y el Método SS. Se han generado datos de manera controlada utilizando funciones específicas para asegurar la reproducibilidad y la aleatoriedad. Los experimentos se llevaron a cabo en un sistema con características definidas, como 16 GB de RAM y un sistema operativo GNU/Linux, para medir el rendimiento de los algoritmos en términos de accesos a disco y tiempos de ejecución en función del tamaño de los datos.

5.2. Evaluación de Resultados

Los resultados obtenidos indican que mientras el Método CP maneja mejor los grandes volúmenes de datos, el Método SS es más eficiente en términos de tiempo de búsqueda y menor número de I/Os, además de comportarse de manera más estable y previsible, pero enfrenta graves limitaciones de escalabilidad. Estas observaciones sugieren que el Método CP es preferible para aplicaciones que involucran grandes conjuntos de datos, mientras que el Método SS podría ser más adecuado para conjuntos de datos más pequeños donde la eficiencia de búsqueda es crucial.

5.3. Verificación de Hipótesis

La hipótesis inicial, que anticipaba que el Método CP superaría al Método SS en la gestión de grandes volúmenes de datos, se ha confirmado en gran medida. Esta superioridad del Método CP se ha reflejado claramente en los tiempos de ejecución observados. Sin embargo, el Método SS ha demostrado ser ligeramente más eficiente en términos de tiempo de búsqueda y operaciones de entrada/salida.

A pesar de esta mejora operativa en el Método SS, consideramos, como equipo de investigación, que el Método CP es superior en términos generales. La pequeña ganancia en eficiencia de búsqueda y de I/O del Método SS no compensa las notables ventajas en tiempo de construcción que ofrece el Método CP. Concluimos, por tanto, que el Método CP es preferible en la mayoría de los casos prácticos (por no decir casi todos), pues si quisieramos aplicar el Método SS en BigData, no terminaría en el tiempo requerido.

No obstante, la elección del método adecuado siempre dependerá del contexto específico de su aplicación. En situaciones donde una búsqueda rápida es crucial y cada milisegundo cuenta, y se maneja un volumen de datos moderado, el Método SS podría ser preferible. Esta consideración es esencial para optimizar el rendimiento en escenarios donde la velocidad de búsqueda es más crítica que el tiempo de construcción o el manejo de grandes volúmenes de datos.

5.4. Oportunidades de Mejora

Para futuras versiones de estos métodos, sería beneficioso explorar las siguientes mejoras:

- Optimizar el cálculo de pares de puntos más cercanos en el Método SS para reducir el costo computacional y mejorar la escalabilidad.
- Investigar y posiblemente implementar algoritmos de indexación más eficientes que podrían mejorar la gestión de I/Os y tiempos de búsqueda en grandes conjuntos de datos.
- Aumentar la capacidad de memoria del sistema (hardware) de prueba podría permitir un manejo más eficiente de mayores volúmenes de datos y proporcionar una evaluación más completa de los algoritmos bajo condiciones extremas.
- Implementación de paralelización utilizando threads. Esto permitiría que varias operaciones de lectura y escritura se ejecuten simultáneamente, reduciendo significativamente el tiempo de procesamiento y aumentando la eficiencia del sistema.
- Investigar la posibilidad de mezclar ambos métodos de construcción, dado que el método SS divide los grupos de manera más eficaz, mientras que el método CP optimiza la ejecución una vez que los puntos están distribuidos adecuadamente. Esta fusión podría potenciar las fortalezas de cada método, mejorando tanto la segmentación como la eficiencia en la ejecución.

Conclusión: Este estudio proporciona una comprensión de los desafíos y ventajas de los Métodos CP y SS en la gestión de datos.