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

Tarea 1

M-tree

Integrantes: Vicente Leyton
Juan Molina
Andrés Salazar
Profesor: Gonzalo Navarro
Auxiliar: Diego Salas

Fecha: 12 de mayo de 2024

Índice de Contenidos

1. Introducción **1**

2. Desarrollo **2**
2.1. Ciaccia-Patella (CP) 2
2.2. Sexton-Swinbank (SS) 3

3. Experimentos **4**

4. Conclusiones **6**

5. Recapitulación **7**

Índice de Códigos

1. Encabezados y definiciones importantes. 5
2. Ejemplo de un print inicial de los experimentos. 6

1. Introducción

En el siguiente informe se describe un experimento realizado con el fin de comparar dos métodos de construcción de un M-tree, mediante algoritmos bulk-loading, es decir, donde el input es un conjunto de elementos. En este caso, se utilizan puntos en el plano cartesiano. El M-tree es una estructura de tipo árbol, donde los nodos contienen entradas, cada una con un punto, un radio cobertor y una dirección al nodo hijo de su nodo correspondiente. La idea de este informe es comparar la eficiencia de dos algoritmos con base en la cantidad de accesos a discos que hace cada uno. Esto se menciona con el fin de que no se malentienda que cada operación entre puntos tiene un costo, ya que estas se realizan en memoria principal. El costo en este caso se mide según los accesos a disco, o lo que es equivalente, los accesos a cada nodo del árbol, pero cada uno de estos accesos realiza muchas operaciones.

Los algoritmos realizan clustering de un conjunto de puntos, lo cual se describe más detalladamente en los papers donde son propuestos. Estos son conocidos como Ciaccia-Patella y Sexton-Swinbank por los nombres de sus respectivos autores. Para sintetizar, los identificaremos en el informe como algoritmos CP y SS respectivamente.

Nuestra hipótesis es que el segundo algoritmo, SS, realiza menos accesos que el algoritmo CP. La razón de esto radica en la forma en que se realiza la división del conjunto de puntos. El primer algoritmo selecciona un conjunto k de puntos aleatorios, mientras que el segundo busca siempre los puntos o clusters más cercanos, incluso en algunas ocasiones se llama a la función MinMaxSplitPolicy, que calcula cuál es la mejor forma de dividir un conjunto de puntos, construyendo así siempre el mejor árbol posible para el conjunto de puntos dado. Entonces, al ser aleatorios los puntos para la primera forma, existe una probabilidad de que el conjunto de clusters seleccionado no sea apropiado y cree un árbol donde deba realizar más accesos, ya que se crean nodos con más solapamiento. Por esto, el algoritmo SS creará un árbol el cual realice menos accesos para la búsqueda.

Cabe recalcar que este segundo árbol tiene un mayor costo de construcción, dado que al comparar todas las divisiones posibles de conjuntos, este probablemente realizará operaciones extra habiendo ya encontrado la mejor división posible. El otro método, en cambio, puede encontrar una división más apropiada antes gracias a la aleatoriedad y por ende, tomar menos tiempo en su construcción. Sin embargo, el fin de esta tarea es comparar accesos a disco en la búsqueda, por lo que nuestra hipótesis se aplica para el árbol ya construido, y por lo mencionado anteriormente, el algoritmo SS construirá un árbol con menor solapamiento.

2. Desarrollo

Tal como se mencionó, esta tarea consiste en comparar dos algoritmos de construcción del M-tree. Los algoritmos son del tipo bulk-loading, es decir, reciben un input con un conjunto de datos (en este caso, puntos en el plano) y el tamaño del conjunto, y construyen el árbol con los datos en cuestión, almacenándolos en las entradas de cada nodo del árbol.

2.1. Ciaccia-Patella (CP)

El método CP, fue implementado siguiendo secuencialmente los pasos del algoritmo indicados en el enunciado de la tarea y su paper adjunto. Se trabajó con punteros a arreglos de C, haciendo las alocaciones y realocaciones de memoria correspondientes con las funciones malloc y realloc de la librería estándar de C. Para el primer paso se verifica si hay una cantidad menor o igual de B puntos, si es así, se crea un nodo hoja y se insertan todos los puntos en las entradas, mediante un ciclo for secuencial. Luego, desde el paso 2 al 4 y sus subetapas se implementan dentro de un bloque do-while, donde la condición del while es que el tamaño del conjunto de samples sea 1. Esto debido a que el paso 5 indica que se debe volver al paso 2 en caso de que se cumpla esa condición. Se determina el tamaño k y se guarda como F_size. Dentro del bloque do-while, se calculan los puntos únicos y aleatorios desde el conjunto inicial de puntos del algoritmo, se guardan en un arreglo F y se utiliza una estructura llamada SubsetStructure para guardar un punto central del sample f_j en F, un puntero a un arreglo de samples que representan los F_j y el número de elementos en dicho arreglo, además de un indicador si se sigue trabajando con estos F_j o no, en caso de que sus puntos hayan sido redistribuidos en el paso 4. Cada vez que se elimina un punto de F, se utiliza la función auxiliar DeletePointInF, la cual busca y elimina el punto perteneciente a F_j. En cambio, cuando se agregaba un punto a algún F_j en el arreglo de estructuras SubsetStructure, se llamaba a la función que se creó llamada AddPointToArray, la cual insertaba el punto dentro del F_j, realizando la respectiva realocación de memoria, y actualizaba el tamaño del arreglo. Luego, como se debía aplicar recursivamente el algoritmo CP para cada F_j, se creó un arreglo T donde se irían guardando los T_j obtenidos por la recursión. Para ello, se creó una estructura llamada PointAndNode donde se guardaba cada árbol, su punto perteneciente en F y su altura para no realizar cálculos repetidos en los pasos 8 y 9, para lo cual se utilizó la función auxiliar creada treeHeight. Por otro lado, el arreglo T_prime representa a T', cuyo arreglo, al igual que el arreglo T, comienza vacío y cada vez que se debía insertar un árbol, se realocaba su tamaño y el árbol correspondiente era insertado, utilizando la función auxiliar addPointAndNode. En este arreglo solo se insertaban los árboles cuya altura calculada y seteada anteriormente eran igual a h, donde h se define como la altura mínima entre los árboles en T_prime. Luego de aplicarle el algoritmo CP a F y obtener T_{sup}, se unió cada elemento en T' utilizando la función auxiliar joinTj, la cual buscaba el punto correspondiente del elemento de T' dentro de las hojas de T_{sup} e insertaba el subárbol en dicha hoja. Finalmente, para setear los radios cobertores se utilizó

la función auxiliar setCoveringRadius, la cual para cada entrada del árbol calculaba su radio cobertor de manera recursiva sobre cada nodo, estando así el árbol completo ya formado y devolviendo así el algoritmo CP un puntero al árbol construido.

2.2. Sexton-Swinbank (SS)

Para el método SS, se crearon las distintas funciones propuestas en el enunciado, las cuales ayudarían más tarde a construir el árbol. Además, se crearon algunas estructuras de datos, siendo estas Cluster, ClusterArray, EntryArray y EntryArrayArray, dónde se explicará a continuación el uso de cada una.

En primer lugar, se implementó la función cluster, la cual recibe como input un Cluster, el cual contiene un arreglo de puntos con su tamaño, y retorna un ClusterArray, el cual contiene un arreglo de clusters con su tamaño también. Siguiendo el enunciado, la función verifica que el tamaño del input sea mayor que b, luego cada punto del input se convierte en un cluster de un único punto y con la función closest_pair obtenemos los clusters más cercanos. Esta función auxiliar realiza un algoritmo de fuerza bruta, de complejidad $O(n^2)$, haciendo ineficiente la solución. No obstante, al obtener los clusters más cercanos, se analiza, a partir de sus tamaños, si pueden unirse en un solo cluster (el tamaño de la unión debe ser menor que B). Si es así, se agrega la unión al ClusterArray que se retornará, si no, solo se agrega el cluster de mayor tamaño. Tras hacer esto con todos los clusters, nos quedará uno que aún no se agrega al ClusterArray que vamos a devolver. Así que buscamos en este último su vecino más cercano. Si el tamaño de la unión entre estos es menor que B, se añade la unión al output. Si no, la unión se dividirá en dos clusters mediante la función MinMaxSplitPolicy. Estos últimos dos clusters se añaden al output y finalmente retornamos.

La segunda función que se creó fue la de OutputHoja, nuevamente siguiendo paso a paso las indicaciones del enunciado. Esta función retorna una tupla, que para nuestro caso es de tipo Entry, ya que tiene un punto, un radio y una dirección al hijo. Además, esta función también recibe como input un cluster, que sería un conjunto de puntos con su tamaño. Entonces, primero se calcula el medoide primario del cluster recibido y se crea un nodo de tipo Node. Luego, por cada punto en el cluster, se crea una nueva entrada con radio cobertor 0.0 e hijo $NULL$, se añade esta entrada al nodo y se calcula su radio cobertor. Finalmente, devolvemos una entrada con el medoide primario, el radio cobertor y el puntero al nodo que se creó.

La tercera función corresponde a OutputInterno, que sigue una idea muy parecida a la anterior, solo que en este caso, recibe un EntryArray, un arreglo de entradas. Entonces, primero toma todos los puntos de las entradas y calcula con estos, el medoide primario. Luego, creamos un nodo, le agregamos cada entrada que recibimos y calculamos el radio cobertor. Finalmente, devolvemos una tupla con el medoide primario, el radio cobertor y un puntero al nodo creado.

La última función corresponde a Sexton-Swinbank, la cual recibe un arreglo de puntos con su tamaño y se encarga de utilizar las funciones definidas anteriormente para construir el M-tree. Primero se crea la estructura de tipo Cluster con el input y se revisa la cantidad de puntos, si son menos de B, se crea el nodo hoja con OutputHoja, si no, hacemos un clustering con la función cluster a los puntos. Luego, se crea el nodo de cada cluster y su entrada que le apunta. Si la cantidad de clusters que quedan son menores a B, creamos el único nodo interno que necesitamos con la función OutputInterno ocupando las entradas que obtuvimos luego del clustering y retornamos el nodo raíz. Si la cantidad de clusters que se obtuvieron en un inicio fueron mayores a B, se tienen que crear más de un nodo interno, por lo que ahora se realizará nuevamente un clustering, pero con los medoides primarios de cada cluster que ya se tienen. Luego, con estos nuevos clusters, que contienen a los anteriores, se crean los nodos internos para estos. Se sigue revisando entonces la cantidad de clusters que se obtuvieron, si son mayores a B, hay que seguir creando nuevos nodos internos haciendo clustering, que contengan a los anteriores. Finalmente, creamos el nodo raíz con OutputInterno cuando la cantidad sea menor a B y retornamos su dirección.

En cuanto a las diferencias entre ambos algoritmos, podemos mencionar que el método de construcción Ciaccia-Patella emplea un enfoque recursivo para dividir su conjunto de datos (puntos) en subconjuntos más pequeños, en contraste con el método de construcción Sexton-Swinbank, el cual no utiliza una técnica de recursión. Además, la manera en escoger los puntos es diferente, ya que el primer método los selecciona de manera al azar entre el conjunto de datos inicial, mientras que el segundo método realiza MinMax Split Policy sobre el conjunto de datos entregados al algoritmo. Por otro lado, en cuanto a la estructura de cada método de construcción, podemos notar que el método CP solo ocupa una tarea principal, la cual se ejecuta sobre cada subconjunto de puntos en cada recursión. En cambio, el método SS utiliza sub-tareas, las cuales se emplean dentro de su función principal.

3. Experimentos

Para la experimentación se detallan algunos aspectos técnicos. Las implementaciones de los algoritmos y sus test se realizaron en el lenguaje C. Estas fueron probadas en un sistema virtualizado de la distribución Debian 12 de Linux, además de Windows 10 y 11. Se probaron distintos sistemas para evaluar si existían diferencias, más no fue el caso. Los caches L2 y L3 tenían un tamaño de 2048 kb y 6144 kb respectivamente. Los experimentos fueron siempre ejecutados con 12 Gb de RAM. Para la experimentación, se crearon una serie de archivos, cuya forma de compilación y ejecución se encuentra detalla en el archivo README.md adjunto a este informe y a todos los archivos de código.

El archivo main es donde se ejecutan los experimentos, contiene la asignación de los conjuntos P de puntos y Q de consultas. La idea era probar cada método con n puntos, para cada $n \in \{2^{10}, 2^{11}, ..., 2^{25}\}$. Para entender el cómo se implementó esto se deben considerar

una serie de encabezados de estructuras y funciones definidas en el código. Lo que representa cada una se encuentra comentado en el código fuente insertado en el informe:

Código 1: Encabezados y definiciones importantes.

```
// Estructuras que forman parte de un M-tree
typedef struct point Point; // Punto del plano
typedef struct node Node; // Nodo del M-tree
typedef struct entry Entry; // Entrada de un nodo
typedef struct query Query; // Consulta

// Funciones para los métodos de construcción
Node *ciacciaPatella(Point *P, int P_size);
Node *sextonSwinbank(Point *P, int P_size);

// Otras estructuras creadas con el fin de facilitar la implementación
typedef struct subsetstructure SubsetStructure; // Conjunto de samples para método CP
typedef struct cluster Cluster; // Cluster
typedef struct clusterarray ClusterArray; // Conjunto de Clusters
typedef struct entryarray EntryArray; // Conjunto de entradas
typedef struct entryarrayarray EntryArrayArray; // Conjunto de conjuntos de entradas
```

Para efectos de esta tarea, se considerará el tamaño de un nodo como el tamaño de un bloque en disco. En otras palabras, si el tamaño de una entrada es e bytes y el tamaño de un bloque es B, se considerará que un bloque tiene tamaño $B \cdot e$ en bytes y un acceso a disco como el acceso a las B entradas.

Nuestra estructura entrada contiene un punto del tipo Point, el cual a su vez consiste de dos doubles, y además contiene el radio cobertor que también corresponde a un valor de tipo double y un puntero a un nodo. Todos estos valores utilizan 8 bytes en memoria, luego en total, una entrada pesa 32 bytes en memoria. Con esto, dado que cada nodo corresponde a un bloque en memoria y la memoria de cada bloque es de 4096 bytes, tenemos que en cada nodo habrá $B = 4096/sizeof(Entry)$ entradas, es decir $B = 4096/32 = 128$. Luego el valor de B estimado es 128 y $b = B/2$ será 64.

Los puntos aleatorios para las queries y para los conjuntos de test se crearon con la función rand() de la librería estándar de C. Se creó una función random_double() la cual retorna un real de doble precisión aleatorio en el rango [0,1] y se asignó en la estructura Point en sus campos x e y para cada punto en los sets.

Las funciones de los métodos reciben un puntero a un arreglo de puntos (Representados por la estructura Point) y la cantidad de puntos en dicho arreglo. Retorna un puntero al nodo raíz del árbol construido con dichos puntos.

En total hay 4 archivos junto con el archivo README.md que contiene los comandos para poder compilar y ejecutar el código. Hay dos archivos, ss.c y cp.c que cada uno corresponde

a un método de construcción y sus funciones y estructuras auxiliares. Luego está el archivo mtree.c que contiene el código con las estructuras que representan al árbol, tales como los Nodos, Entradas, etc. Por último está el archivo mtree-test.c que contiene la función main que crea los conjuntos de puntos aleatorios y de queries, y ejecuta los experimentos. Basta con compilar y ejecutar este último archivo para poder ejecutar todo el código. Un ejemplo de algo que imprime el código sin los resultados es lo siguiente:

Código 2: Ejemplo de un print inicial de los experimentos.

```
Entry size: 32 bytes
B: 128

Query 1 - Point: (0.612354, 0.418806)
Query 2 - Point: (0.928251, 0.105228)
Query 3 - Point: (0.483230, 0.047456)
Query 4 - Point: (0.484207, 0.613330)
Query 5 - Point: (0.925138, 0.340892)

Array 1 size: 1024

Point 1: (0.742729, 0.304788)
Point 2: (0.270943, 0.578906)
Point 3: (0.044832, 0.949919)
Point 4: (0.409711, 0.754784)
Point 5: (0.297739, 0.179937)
```

El print anterior muestra los puntos de las primeras 5 queries creadas, el tamaño del primer set de puntos y los primeros 5 puntos de este set. Dado que no se pudo compilar y ejecutar el código a tiempo, en esta sección no se muestran resultados.

4. Conclusiones

Dada la complejidad que toma construir un árbol y limitaciones de tiempo, no pudimos obtener resultados relevantes que puedan ser estudiados. Sabemos que el algoritmo SS realiza la construcción, ya que en ningún momento deja de ejecutarse el código compilado. Nos hubiese gustado tener más tiempo para realizar las pruebas, ya que se perdió mucho tiempo en programar las implementaciones.

Se estima que el algoritmo SS sí hubiese mostrado resultados, pero dado el alto costo que implica construir este árbol con este método, no tuvimos tiempo para construirlo y por ende obtener conclusiones satisfactorias.

5. Recapitulación

En este informe se investigó y analizó dos algoritmos para la construcción de un M-tree: el Método Ciaccia-Patella (CP) y el Método Sexton-Swinbak (SS). En primer lugar, se habló sobre qué era un M-tree y cómo estaba compuesto. Además, se realizaron hipótesis previas sobre los resultados de los métodos. Luego, se investigó en detalle como era el funcionamiento de cada uno de ellos y se explicó a detalle cómo fueron implementados, y se discutió sobre las diferencias en el funcionamiento que poseían entre ellos. También se comentó sobre el entorno y aspectos técnicos en los cuales fueron implementados ambos algoritmos, y sobre cómo compilarlos y ejecutarlos. El objetivo era comparar los accesos simulados a disco, lo cual no fue posible dado a la falta de tiempo para evaluar detalladamente cada algoritmo debido al gran tiempo que tomaba la construcción de los árboles con cada método.

Como mejora para una versión futura, el método Sexton-Swinbank ocupa una función auxiliar que lo hace muy ineficiente, la función closest_pair, ocupando casi todo el tiempo que se demora en construir el M-tree. Esta función tiene complejidad $O(n^2)$ y se buscó información que decía que este problema, buscar el par de puntos más cercanos, puede resolverse en $O(n * \log(n))$. Lamentablemente, no se pudo implementar esta función, por lo que los tiempos de construcción del árbol eran muy grandes. Otro punto a mejorar, podría ser el uso de otro lenguaje de programación como C++. Esto facilitaría el uso de memoria más complejo que tiene C, además de las librerías que contiene. Por último, quizás sería mejor ocupar otros métodos en vez de MinMaxSplitPolicy, por ejemplo, usar insertion splitting policy (la utilizada en su paper).

No se pudo resolver correctamente el método CP, y no se pudo resolver eficientemente el SS, por esto la falta de resultados. Además, en el método SS, se analizó que devolvía solo un acceso a disco por query para las cantidades de puntos más baja, lo cual está incorrecto y no se pudo resolver.

Para extender el código, se podría recibir como función un método distinto para calcular la distancia entre puntos, así no solo se ocuparía la distancia euclidiana. También se podría indicar la dimensión de los puntos, y no trabajar solamente en dos dimensiones.