



Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la
Computación
CC4102 - Diseño y Análisis de Algoritmos

Tarea 2

KD-Trees

Autores: Claudio Berroeta
Sebastián Ferrada
Profesor: Pablo Barceló
Auxiliar: Miguel Romero
Ayudantes: Javiera Born
Giselle Font
5 de junio de 2014

Índice

1. Introducción	3
1.1. Hipótesis	3
1.2. Ambiente Operacional	4
2. Diseño de la Estructura y Experimentos	5
2.1. Estructura del Árbol	5
2.2. Construcción del Árbol	7
2.3. Consultas de Vecino más cercano	7
2.4. Diseño de los Experimentos	9
3. Resultados	10
4. Conclusiones	11
5. Anexos	12
5.1. Merge function	12
5.2. Partition function	13
5.3. Experimentos	13
5.3.1. Experimento 1	13
5.3.2. Experimento 2	13
5.3.3. Experimento 3	13
5.3.4. Experimento 4	13
5.4. Resultados	13
5.4.1. Resultado 1	13
5.4.2. Resultado 2	13
5.4.3. Resultado 3	13
5.4.4. Resultado 4	13

1. Introducción

En el presente informe se pretende mostrar un análisis sobre la performance de la construcción y consultas a Árboles KD, los cuales alojan puntos del plano y los organizan de forma tal que se pueda consultar fácilmente por el vecino más cercano a un punto dado. Este problema de vecino más cercano es particularmente útil, por ejemplo, si se tiene una cadena de locales de pizza, saber desde qué local despachar la pizza, dada una dirección de pedido.

La idea es, dado un conjunto de puntos, particionarlos según una recta $x = c$. El valor de la constante de la recta, c , puede ser la media aritmética o la mediana entre las coordenadas x de los puntos. Particionamos el conjunto de puntos según si están a la derecha o a la izquierda de la recta y proseguimos recursivamente particionando los dos conjuntos obtenidos, pero esta vez, particionando respecto a una recta $y = c'$.

Se estudiarán los tiempos de construcción, altura del árbol construido, espacio utilizado por el árbol y el tiempo que toma una consulta sobre el árbol en 4 situaciones concretas, dadas por las combinaciones entre árbol que particiona por mediana o árbol que particiona por media y puntos aleatorios o puntos de baja discrepancia.

1.1. Hipótesis

1. Se espera que la altura de los árboles sea $O(\log(n))$, pues cada árbol intentará particionar el conjunto de puntos en dos mitades con la misma cantidad de elementos cada una.
2. Dado lo anterior, se espera que la construcción tome tiempo $O(n \log(n))$, pues recorreremos los n puntos en los $\log(n)$ niveles que se espera tener.
3. Dada la topología de la estructura, se espera que las consultas de vecino más cercano, se ejecuten en tiempo $O(\log(n))$.
4. Además, se espera que el espacio utilizado por el árbol no sea superior a $O(n)$.
5. Dado que tanto la media, como la mediana son medidas de centralización casi igualmente certeras, no se espera que la elección de uno por sobre el otro cause diferencias importantes en cuanto a la eficiencia de la construcción del árbol o del balanceo de la estructura resultante, lo que implica que tampoco habría una optimización en el tiempo de consulta ni en la altura del árbol ni en el espacio utilizado.
6. El tener puntos de baja discrepancia, se espera que nos permita descartar ramas completas del árbol en el segundo recorrido en la consulta de punto más cercano, pues hay una menor probabilidad de intersección/colisión respecto a una distribución aleatoria de puntos, por lo que se espera que las consultas en árboles contruidos sobre puntos de baja discrepancia sean más rápidas.

7. Claramente, los experimentos en memoria secundaria demorarán mayor tiempo en ejecutarse, con $O(n)$ accesos a disco, en promedio durante la construcción del árbol (n escrituras) y cantidad logarítmica de accesos para la búsqueda (Asumiendo un nodo por bloque, sin buffer).

1.2. Ambiente Operacional

Los experimentos fueron ejecutados en un Notebook con Sistema Operativo Windows 8 de 64 bits, 8 Gb de memoria RAM y procesador intel Core i7 de 3.630 GHz. Para la implementación se utilizó el Java Development Kit v1.7.

2. Diseño de la Estructura y Experimentos

El diseño con el que se implementó la estructura, es bastante clásico, un árbol que contiene una raíz, un nodo, y una serie de nodos que guardan punteros a sus hijos y un valor, el cual puede ser una recta divisoria o el valor de un punto, propiamente tal.

2.1. Estructura del Árbol

Un árbol tiene un nodo raíz, sabe como construirse a partir de un set de puntos y es capaz de recibir consultas de vecino más cercano a un punto. Este árbol debe “decidir” cómo particionar los nodos, es por esto que se crearon dos tipos de árboles, un árbol que particiona por mediana y otro por media aritmética. Entonces usando un Template Pattern, el árbol KD genérico sabe cómo construirse y son los hijos los que definen el criterio para seleccionar la recta de partición, según el siguiente diagrama de clases:

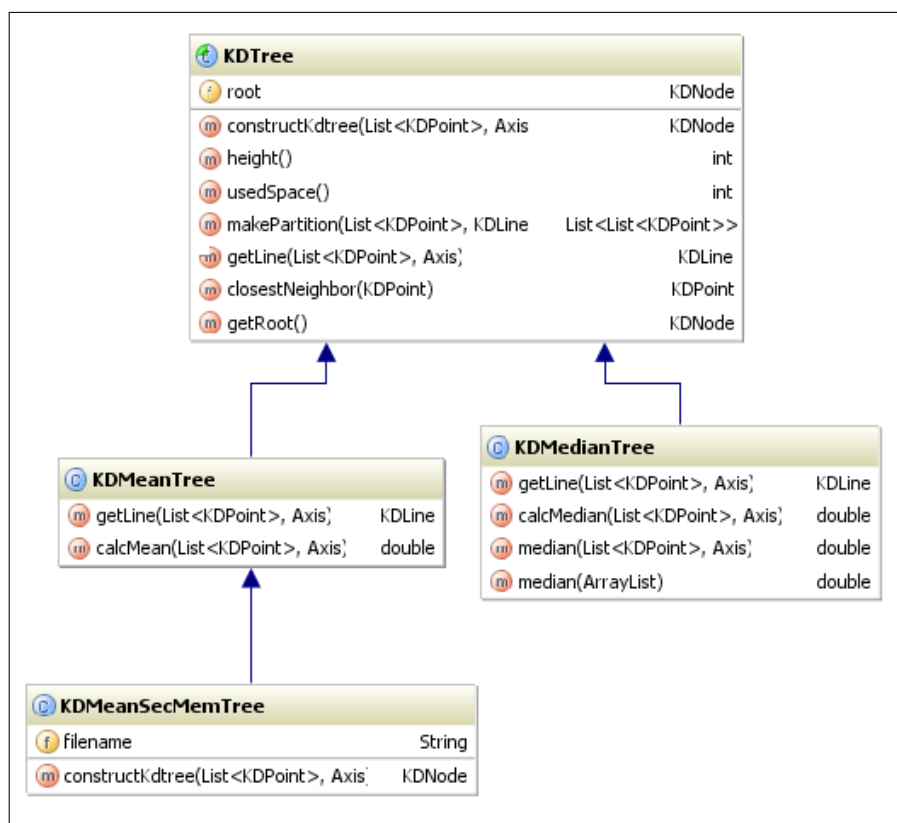


Figura 1: UML para las clases que implementan la estructura

Por otro lado, son Nodos quienes guardan la información del árbol, pues este solo tiene un puntero al nodo raíz. Hay nodos de dos tipos, los internos guardan las rectas que particionan el espacio y las hojas guardan la información de los puntos, por lo que se tiene el siguiente diagrama de clases:

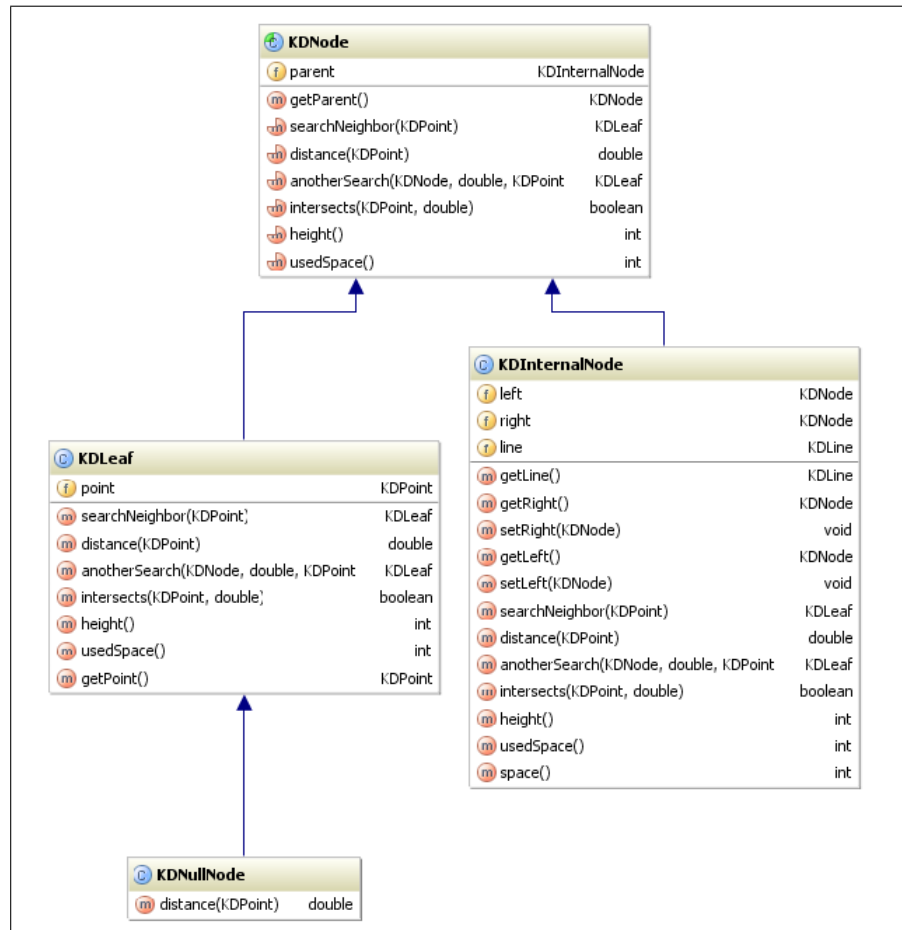


Figura 2: UML para las clases que implementan los nodos

Los nodos también son los responsables de recorrer el árbol durante las consultas, hacia los hijos y hacia el padre, además son los que calculan la intersección entre los sectores que delimitan y el círculo de consulta.

2.2. Construcción del Árbol

La construcción se hace recursivamente, con caso base un conjunto de puntos de tamaño 1. En cada fase se particiona el conjunto de acuerdo al criterio de mediana o media, intercalando el eje en cada nivel. El código es bastante simple y se presenta a continuación:

```
public KNode constructKdtree(List<KDPoint> points, Axis axis){
    if(points.size() == 1 ){
        return new KDLeaf(points.get(0));
    }
    KDLine line = getLine(points, axis) ; //This line depends on the Tree criteria
    List<List<KDPoint>> partition = makePartition(points,line);
    return new KDInternalNode(line,
                               constructKdtree(partition.get(0),axis.negated()),
                               constructKdtree(partition.get(1),axis.negated()));
}
```

Las diferentes implementaciones de `getLine` y el método `partition` pueden encontrarse en los anexos

2.3. Consultas de Vecino más cercano

Las consultas buscan descendentemente desde la raíz, el nodo del cuadrante que incluye al punto de consultas. Recordemos que el árbol crea una suerte de rejilla sobre el espacio, entonces la primera búsqueda le indica cual es el cuadrante del punto consultado y se calcula la distancia entre este y el punto del árbol de ese cuadrante.

Con esta distancia volvemos a subir por el árbol y vemos si el cuadrante que enmarca el otro hijo del nodo intersecta el círculo con centro el punto de consulta y radio la distancia que tenemos guardada. Así vamos subiendo y si encontramos alguna distancia mejor, la guardamos. Este comportamiento se registra en el siguiente código:

```
public KDPoint closestNeighbor(KDPoint q){
    KDLeaf currentBest = root.searchNeighbor(q); //looks for node, same sector than q
    double currentDistance = currentBest.distance(q);
    KDNode actual = currentBest.getParent();
    KDNode prev = currentBest;

    while(actual != null){
        KDLeaf temp = actual.anotherSearch(prev, currentDistance, q); //looks for a
            best fit
        if(temp.distance(q) < currentDistance){
            currentBest = temp;
            currentDistance = currentBest.distance(q);
        }
        prev = actual;
        actual = actual.getParent();
    }
    return currentBest.getPoint();
}
```

Los métodos searchNeighbor y anotherSearch se pueden encontrar en los anexos

2.4. Diseño de los Experimentos

Los experimentos consistieron en construir árboles KD con arreglos de puntos de tamaños sucesivamente más grandes (tamaño $2i$, $i \in \{10, \dots, 20\}$) en las diferentes combinaciones de criterio de partición y de distribución de puntos y medimos su tiempo, la altura del árbol y la cantidad de espacio en memoria que utiliza. Luego a cada sabor de árbol le hacemos consultas y medimos el tiempo que toma en contestar. Cada experimento se repite las veces necesarias para que el error sea menor al 5 % (i.e., hasta que $\frac{stdv}{mean} \leq 0,05$). Para detalles de la implementación de la batería de experimentos, referirse al código fuente adjunto.

3. Resultados

very successful

4. Conclusiones

WOW

5. Anexos

5.1. Merge function

```
static private int[] merge(int[] A, int[] B){
    int i=0, j=0, n=A.length+B.length;
    int[] res = new int[n];

    for(int k=0; k<n; k++){
        if(i<A.length && j<B.length){
            if(A[i]<=B[j]){
                res[k] = A[i];
                i++;
            }else{
                res[k] = B[j];
                j++;
            }
        } else if(i == A.length){ //si se acaban los elems de B
            for(;k<n;){
                res[k] = B[j];
                k++; j++;
            }
        }else{
            for(;k<n;){ //si se acaban los elems de A
                res[k] = A[i];
                k++; i++;
            }
        }
    }
    return res;
}
```

5.2. Partition function

```
private static int partition(int[] A, int start, int end) {  
    int index = start + new Random().nextInt(end-start);  
    int pivot = A[index];  
    int i = start, j = end;  
    while(i<j){  
        while(A[i] < pivot) i++;  
        while(A[j] > pivot) j--;  
        swap(A, i, j);  
    }  
    return i;  
}
```

5.3. Experimentos

5.3.1. Experimento 1

5.3.2. Experimento 2

5.3.3. Experimento 3

5.3.4. Experimento 4

5.4. Resultados

5.4.1. Resultado 1

5.4.2. Resultado 2

5.4.3. Resultado 3

5.4.4. Resultado 4