Universidad Nacional de San Agustín de Arequipa

Estructuras de Datos Avanzadas

Kd-Tree

Integrantes del GRUPO 2A:
- Portocarrero Espirilla, Diego Armando
- Coayla Zuñiga, Gonzalo Eduardo
- Jara Huilca, Arturo Jesús
- Pucho Zevallos, Kelvin Paul

Profesor :
- Machaca Arceda Vicente Enrique

10 de noviembre de 2020

Índice

1. Introducción

Kd-Tree es una estructura de datos de particionado del espacio que organiza los puntos en un Espacio euclídeo de k dimensiones. Un árbol kd emplea sólo planos perpendiculares a uno de los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. Además, todos los nodos de un árbol kd, desde el nodo raíz hasta los nodos hoja, almacenan un punto. Mientras tanto, en los árboles BSP son las hojas los únicos nodos que contienen puntos (u otras primitivas geométricas). Como consecuencia, cada plano debe pasar a través de uno de los puntos del árbol kd.

1.1. Repositorio

https://github.com/khannom/EDA-Grupo/tree/master/Kdtree

2. Ejercicios

2.1. Creacion del archivo main.html.

```
<html>
    <head>
        <title >Kd tree </title>
        <script src = "p5.min.js" ></script>
        <script src = "kdtree.js"></script>
        <script src = "sketch.js" ></script>
        <!-- Librerias para graficar el arbol -->
        <!-- <script
            src="../../d3-graphviz/node_modules/d3/dist/d3.js">
        </script>
        <script
            src="../../d3-graphviz/node_modules/@hpcc-js/wasm/
            dist/index.js" type="application/javascript">
        </script>
        <script
            src="../../d3-graphviz/build/d3-graphviz.js">
        </script> -->

    </head>
    <body >
        <div id="graph" style="text-align: center;"></div>
    </body>
</html>
```

2.2. Creacion del archivo sketch.js

```
function setup() {
    var width = 250;
    var height = 200;
    createCanvas(width, height);

    background(0);

    // Ejercicio 2
    /* var data = [
        [40, 70],
        [70, 130],
        [90, 40],
        [110, 100],
        [140, 110],
        [160, 100],
    ];
    var point = [140, 90]; */

    // Ejercicio 3
    var data = [
        [40, 70],
        [70, 130],
        [90, 40],
        [110, 100],
        [140, 110],
        [160, 100],
        [150, 30],
    ];
    var point = [140, 90];
    for (let i = 0; i < data.length; i++) {
        fill(225, 225, 225);
        circle(data[i][0], 200 - data[i][1], 7);
        textSize(8);
        text(data[i][0] + "," + data[i][1], data[i][0] + 5, 200 - data[i][1]);
}
fill(255, 0, 0);
circle(point[0], 200 - point[1], 7);
textSize(8);
text(point[0] + "," + point[1], point[0] + 5, 200 - point[1]);

    var root = build_kdtree(data, 0);
```

```
    console.log(root);
    var dotString = generate_dot(root);
    console.log(dotString);
    console.log("Queried point: " + point);
    console.log("Brute force: " + closest_point_brute_force(data, point));
    console.log("Naive closest point: " + naive_closest_point(root, point));
    console.log("Closest point: " + closest_point(root, point));

    // Ejercicio 5
    let arr = new Array();
    var n = 3;
    k_nearest_neighbor(root, point, arr);
    arr.sort((a, b) => distanceSquared(point, a) - distanceSquared(point, b));
    console.log(n + " puntos más cercanos: ");
    for (i = 0; i < n; i++) console.log(arr[i]);

    // Se necesita d3-graphviz para ejecutar la siguiente línea
    // caso contrario, comentar la siguiente línea
    /* d3.select("#graph").graphviz().renderDot(dotString); */
}
```

2.3. Creacion del archivo kdtree.js.

```
k = 2;
class Node {
    constructor(point, axis) {
        this.point = point;
        this.left = null;
        this.right = null;
        this.axis = axis;
    }
}
```

2.3.1. Implementacion de la funcion build_kdtree.

```
function build_kdtree(points, depth = 0,father = null) {
    if (!points.length) return null;
    n = points.length;
    m = points[0].length;

    eje = depth % m;

    points.sort((a, b) => a[eje] - b[eje]);
```

```
    median = Math.ceil((points.length - 1) / 2);

    let izq = [];
    let der = [];
    for (let i = 0; i < median; i++) izq.push(points[i]);
    for (let i = median + 1; i < n; i++) der.push(points[i]);

    let node = new Node(points[median], eje);

    /*****************creacion de sectores***************/
    var width = 250;
    var height = 200;
    var c = color(255, 204, 0);
    stroke(c);
    if(eje == 1){
        var y = node.point[eje];
        if(node.point[father.axis] < father.point[father.axis]){
            line(0, 200-y, father.point[father.axis], 200-y);
        }else{
            line(father.point[father.axis], 200-y, width, 200-y);
        }
    }else if( eje == 0){
        var x = node.point[eje];
        if(!father){
            line(x, 0, x, height);
        }else{
            if(node.point[father.axis] < father.point[father.axis]){
                line(x, 200 - father.point[father.axis], x, height);
            }else{
                line(x, 0, x,200 - father.point[father.axis]);
            }
        }
    }
    father = node;
    /*****************************************************/

    node.left = build_kdtree(izq, depth + 1,father);
    node.right = build_kdtree(der, depth + 1,father);

    return node;
}
```

2.3.2. Implementacion de la funcion getHeight.

```
function getHeight(node) {
    if (!node)
        return -1;
    else
        return (1 + Math.max(getHeight(node.left), getHeight(node.right)));
}
```

2.3.3. Implementacion de la funcion generate_dot

```
function recursive_generate_dot(node) {
    let txt = "";
    if (node) {
        if (node.left) {
            txt = txt + '\t"';
            txt = txt + node.point;
            txt = txt + '" -> "';
            txt = txt + node.left.point;
            txt = txt + '";\n';
            txt = txt + recursive_generate_dot(node.left);
            }
        if (node.right) {
            txt = txt + '\t"';
            txt = txt + node.point;
            txt = txt + '" -> "';
            txt = txt + node.right.point;
            txt = txt + '";\n';
            txt = txt + recursive_generate_dot(node.right);
            }
    }
    return txt;
}

function generate_dot(node) {
    string = "digraph G {\n";
    string = string + recursive_generate_dot(node);
    string = string + "}\n";

    return string;
}
```

2.3.4. Implementación de la función closes_point_brute_force.

```
function distanceSquared(point1, point2) {
```

```
    var distance = 0;
    for (var i = 0; i < k; i++) distance += Math.pow(point1[i]
    point2[i], 2);
    return Math.sqrt(distance);
}


function closest_point_brute_force(points, point) {
    if (points.length < 2) return null;

    var min = distanceSquared(point, points[0]);
    var minPoint = points[0];

    for (let i = 1; i < points.length; i++) {
        var distance = distanceSquared(point, points[i]);
        if (distance < min) {
            min = distance;
            minPoint = points[i];
        }
    }

    return minPoint;
}
```

2.3.5. Implementación de la función closest_naive_closest_point.

```
function distanceSquared(point1, point2) {
    var distance = 0;
    for (var i = 0; i < k; i++) distance += Math.pow(point1[i]
    point2[i], 2);
    return Math.sqrt(distance);
}

function naive_closest_point(node, point, depth = 0, best = null) {
    if (!node) return best;

    if (!depth) {
        best = node.point;
    } else {
        if (distanceSquared(node.point, point) < distanceSquared(best, point)) {
            best = node.point;
        }
    }
```

```
    var axis = depth % node.point.length;

    if (point[axis] < node.point[axis]) {
        return naive_closest_point(node.left, point, depth + 1, best);
    } else {
        return naive_closest_point(node.right, point, depth + 1, best);
    }
}
```

2.3.6. Averigue e implemente una función KNN, que retorna los k puntos mas cercanos a un punto.

```
function k_nearest_neighbor(node, point,arr, depth = 0, best = null) {
    if (!node) return best;

    if (!depth) {
        best = node.point;
    } else {
        if (distanceSquared(node.point, point) < distanceSquared(best, point)) {
            best = node.point;
        }
    }
    var axis = depth % node.point.length;
    arr.push(node.point);
    if (point[axis] < node.point[axis]) {
        best = k_nearest_neighbor(node.left, point,arr, depth + 1, best);

        if (
            Math.abs(point[axis] - node.point[axis]) <
            distanceSquared(point, best)
        )
            best = k_nearest_neighbor(node.right, point,arr, depth + 1, best);
        } else {
        best = k_nearest_neighbor(node.right, point,arr, depth + 1, best);
        if (
            Math.abs(point[axis] - node.point[axis]) <
            distanceSquared(point, best)
        )
            best = k_nearest_neighbor(node.left, point,arr, depth + 1, best);
    }
    return best;
}
```

```
function k_closest_order(arr,k,point,res){
    if(arr.lenght<k){
        k=arr.lenght;
    }
    for (i = 0; i < k; i++) {
        var tmp = arr[0];
        var pos = 0;
        for(j=1;j<arr.length;j++){
            if (distanceSquared(arr[j], point) < distanceSquared(tmp, point)) {
                tmp = arr[j];
                pos = j;
            }
        }
        arr.splice(pos,1);
        res.push(tmp);
    }
}
```

3. Resultados

3.1. Evaluación de los resultados de las funciones implementadas anteriormente.

Pruebas a las funciones:

- closest_point_brute_force
- naive_closest_point
- closest_point
- k_nearest_neighbor

3.1.1. Ejercicio 1

- Conjunto de datos

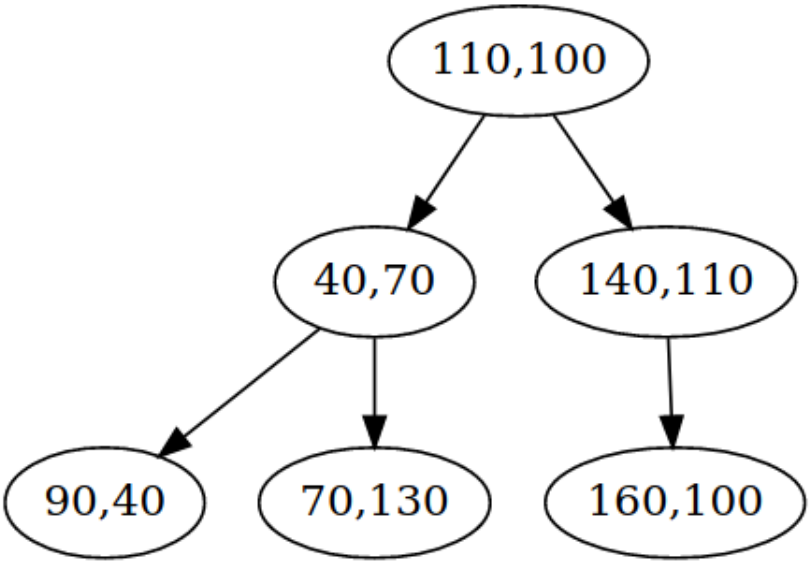
```
var data = [
[40 ,70] ,
[70 ,130] ,
[90 ,40] ,
[110 , 100] ,
[140 ,110] ,
[160 , 100]
```

```
];
var point = [140 ,90];
```

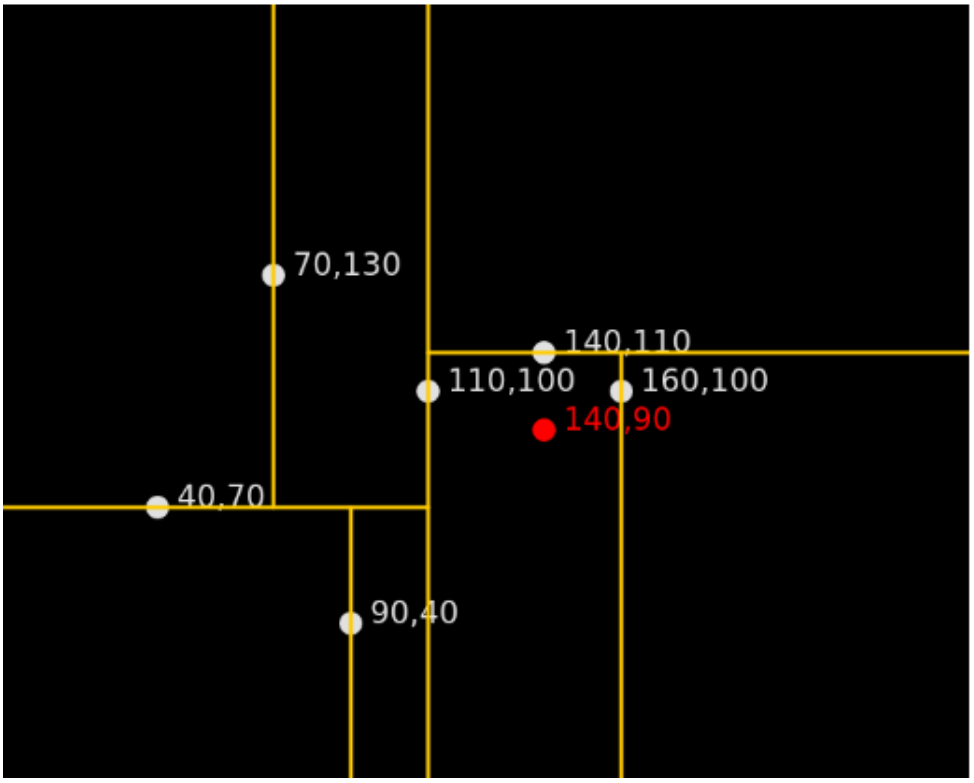
- Arbol en formato dot

```
digraph G {
        "110,100" -> "40,70";
        "40,70" -> "90,40";
        "40,70" -> "70,130";
        "110,100" -> "140,110";
        "140,110" -> "160,100";
}
```

- Grafico del grafo generado



- Grafico multidimensional KD-Tree.



- Resultados en la consola

```
Queried point: 140,90
Brute force: 140,110
Naive closest point: 140,110
Closest point: 140,110
Más cercanos:
▸ Array [ 140, 110 ]
▸ Array [ 160, 100 ]
▸ Array [ 110, 100 ]
```

3.1.2. Ejercicio 2

- Conjunto de datos

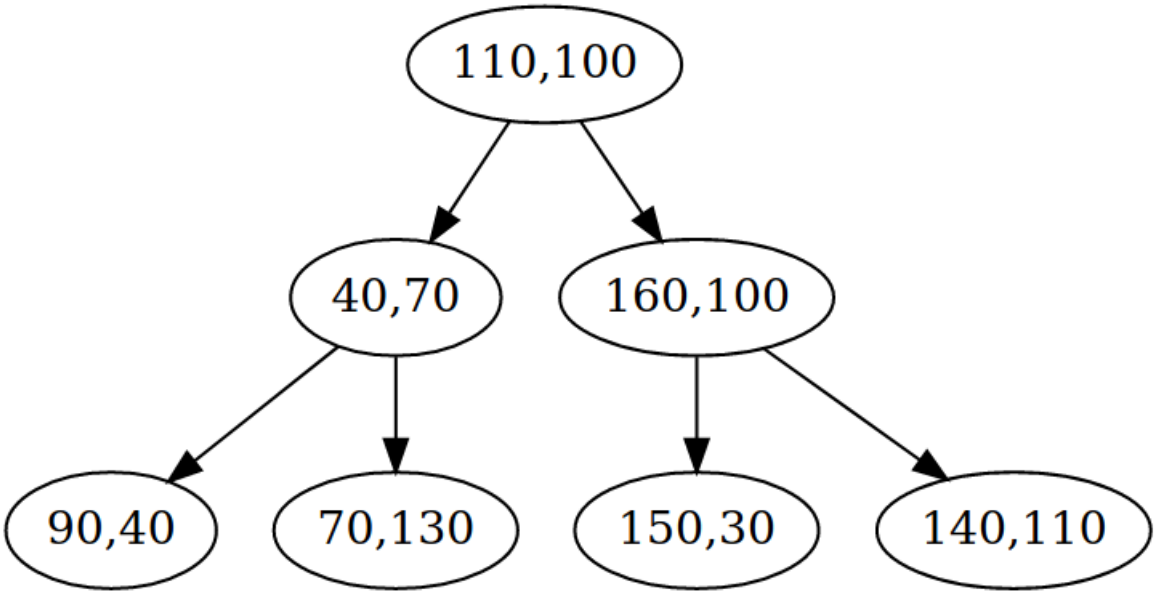
```
var data = [
[40 ,70] ,
[70 ,130] ,
[90 ,40] ,
[110 , 100] ,
[140 ,110] ,
[160 , 100] ,
```

```
[150 , 30]
];
var point = [140 ,90];
```

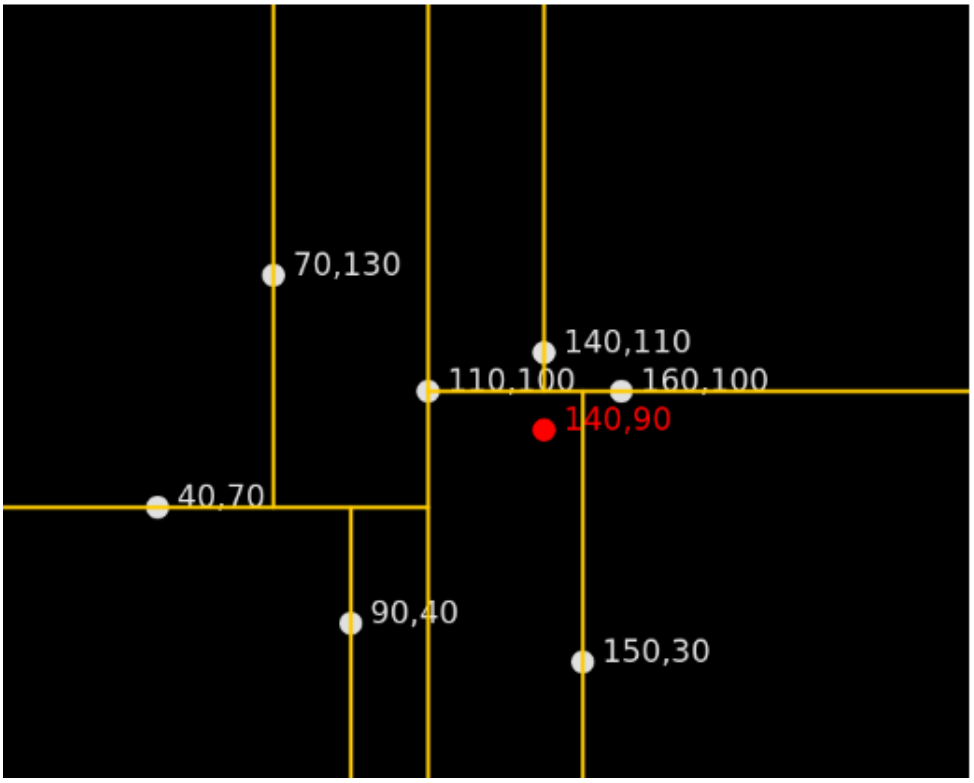
- Arbol en formato dot

```
digraph G {
        "110,100" -> "40,70";
        "40,70" -> "90,40";
        "40,70" -> "70,130";
        "110,100" -> "160,100";
        "160,100" -> "150,30";
        "160,100" -> "140,110";
}
```

- Grafico del grafo generado



- Grafico multidimensional KD-Tree.



- Resultados en la consola

```
Queried point: 140,90
Brute force: 140,110
Naive closest point: 160,100
Closest point: 140,110
Más cercanos:
▸ Array [ 140, 110 ]
▸ Array [ 160, 100 ]
▸ Array [ 110, 100 ]
```