UNIVERSIDAD NACIONAL DEL ALTIPLANO

Facultad Ingeniería Mecánica, Eléctrica, Electrónica y Sistemas

Escuela Profesional de Ingeniería de Sistemas

Trabajo 1 :

KD-Trees

DOCENTE:

ING. COLLANQUI MARTINEZ FREDY

PRESENTADO POR:

CHOQUE CHURA OLIVER FRAN

SEXTO SEMESTRE 2024-I

PUNO - PERÚ

2024

Índice

1. Introducción **3**
2. Definición de KD-Tree **4**
3. Estructura de un KD-Tree **5**
3.1 Nodo 5
3.2 Discriminador 5
3.3 Subárboles 6
4. Ejemplo de Construcción de un KD-Tree **6**
5. Algoritmos Asociados a KD-Trees **8**
5.1 Inserción 8
5.2 Búsqueda de Vecinos Más Cercanos (k-NN) 8
5.3 Eliminación 9
6. Aplicaciones de KD-Trees **9**
6.1 Búsqueda espacial 9
6.2 Procesamiento de imágenes 10
6.3 Machine Learning 10
6.4 Robótica 10
7. Ventajas y Desventajas **11**
7.1 Ventajas 11
7.2 Desventajas 11
8. Optimización y Balanceo de KD-Trees **12**
8.1 Técnicas de balanceo 12
8.2 Rebalanceo dinámico 12
9. Comparación con Otras Estructuras de Datos **13**
9.1 Quad-Trees 13
9.2 R-Trees 13
9.3 Hashing Espacial 14
9. 4 Comparación 14
10. Consideraciones de Implementación **14**
10.1 Complejidad temporal y espacial 14
10.2 Manejo de datos de alta dimensionalidad 15
10.3 Selección de Dimensiones 15
10.4 Escalabilidad y Eficiencia 15
10.5 Optimizaciones Específicas de Aplicación 15
11. Casos de Estudio y Ejemplos Prácticos **16**
11.1 Reconocimiento de Imágenes 16
11.2 Búsqueda Espacial 16
11.3 Clasificación y Regresión 16
11.4 Procesamiento de Nubes de Puntos 17
11.5 Búsqueda de Colisiones en Simulaciones Físicas 17
12. Herramientas y Librerías para KD-Trees **17**
12.1 Implementaciones en lenguajes populares (C++, Python, Java) 17
12.2 Comparación de rendimiento 19
13. Conclusión **19**
14. Referencias Bibliográficas **20**

1. Introducción

En el mundo de la informática y el análisis de datos, gestionar y consultar datos multidimensionales de manera eficiente es un desafío crítico. Las estructuras de datos tradicionales a menudo se quedan cortas cuando se extienden a dimensiones superiores, lo que requiere métodos más sofisticados. Uno de esos métodos es KD-Tree (abreviatura de árbol k-dimensional), una estructura de datos versátil y poderosa diseñada para manejar datos multidimensionales de manera eficiente.

Los KD-Trees ofrecen una forma estructurada de dividir el espacio, lo que los hace particularmente útiles para tareas que involucran datos espaciales. Esto incluye aplicaciones en campos como los gráficos por ordenador, donde se emplean para el trazado de rayos y la detección de colisiones; robótica, donde ayudan a encontrar caminos y evitar obstáculos; y el aprendizaje automático, donde sustentan algoritmos como los k vecinos más cercanos (KNN). Además, los KD-Trees se utilizan ampliamente en sistemas de información geográfica (SIG) para búsquedas eficientes de rangos y consultas de vecinos más cercanos.

El concepto fundamental detrás de KD-Trees es dividir recursivamente el espacio de datos en regiones más pequeñas utilizando hiperplanos perpendiculares a los ejes del espacio. Esta partición permite una rápida búsqueda, inserción y eliminación de puntos, y las operaciones suelen mostrar una complejidad de tiempo logarítmica en condiciones equilibradas. La estructura del árbol se adapta bien a varios tipos de consultas, incluida la búsqueda del vecino más cercano a un punto determinado y la recuperación de todos los puntos dentro de un rango específico.

A pesar de su eficiencia y versatilidad, los KD-Trees presentan ciertos desafíos. Mantener el equilibrio durante las inserciones y eliminaciones puede ser complejo y su

rendimiento puede degradarse en dimensiones superiores debido a la maldición de la dimensionalidad. Sin embargo, sus beneficios en el manejo eficiente de datos dimensionales bajos a moderados los convierten en un elemento básico en geometría computacional y bases de datos espaciales.

En este informe, profundizaremos en la estructura detallada y la construcción de KD-Trees, exploraremos las diversas operaciones que se pueden realizar en ellos y examinaremos sus aplicaciones en diferentes campos. También discutiremos las ventajas y limitaciones del uso de KD-Trees, brindando una descripción general completa de esta estructura de datos esencial. A través de esta exploración, nuestro objetivo es resaltar por qué los KD-Trees son una herramienta fundamental en la representación y manipulación eficiente de datos multidimensionales.

2. Definición de KD-Tree

Los KD-Trees son estructuras de datos especialmente diseñadas para manejar eficientemente conjuntos de datos en espacios de múltiples dimensiones. Su nombre, "K-Dimensional Trees", refleja su capacidad para manejar datos en cualquier número de dimensiones. La idea fundamental detrás de un KD-Tree es organizar los puntos en el espacio de manera que las operaciones de búsqueda y manipulación sean eficientes.

Un KD-Tree se construye mediante la subdivisión recursiva del espacio multidimensional. Cada nodo del árbol representa un punto en el espacio y está asociado con una dimensión específica. La subdivisión se realiza alternando entre las dimensiones en cada nivel del árbol, de manera que los nodos se dividen a lo largo de la dimensión que maximiza la varianza de los puntos.

La estructura de un KD-Tree permite realizar operaciones como la búsqueda de vecinos más cercanos, la inserción y la eliminación de puntos de manera eficiente, lo que lo convierte en una herramienta invaluable en una variedad de aplicaciones donde la eficiencia en el manejo de datos multidimensionales es crucial.

3. Estructura de un KD-Tree

3.1 Nodo

Cada nodo en un KD-Tree representa un punto en el espacio multidimensional. Está compuesto por dos elementos principales:

- **Punto:** Este es el punto en el espacio que el nodo representa.
- **Discriminador:** También conocido como el "splitting value", indica la coordenada en la que se divide el espacio en ese nivel del árbol. El nodo se divide en dos subárboles basados en el valor de esta coordenada.

3.2 Discriminador

En cada nivel del árbol, se selecciona una dimensión específica como discriminador para dividir el espacio. Por ejemplo, en un KD-Tree en dos dimensiones (2D), la dimensión podría ser x o y. En dimensiones más altas, se seleccionan subconjuntos de dimensiones para optimizar la subdivisión del espacio.

3.3 Subárboles

Cada nodo tiene dos subárboles: uno que contiene los puntos con coordenadas menores que el discriminador y otro con coordenadas mayores. Estos subárboles a su vez pueden tener nodos que dividen el espacio aún más, lo que permite una búsqueda eficiente en el árbol.

La estructura de un KD-Tree es fundamental para su funcionamiento, ya que determina cómo se organiza y divide el espacio multidimensional para facilitar las operaciones de búsqueda, inserción y eliminación de puntos.

4. Ejemplo de Construcción de un KD-Tree

Para comprender mejor cómo se construye un KD-Tree, consideremos un ejemplo práctico. Supongamos que tenemos un conjunto de puntos en un espacio bidimensional:

- (3, 6)

- (17, 15)

- (13, 15)

- (6, 12)

- (9, 1)

- (2, 7)

- (10, 19)

Comenzamos con el primer punto de la lista y lo seleccionamos como la raíz del árbol. A continuación, alternamos entre las dimensiones x e y para dividir el espacio en cada nivel del árbol.

1. Nivel 1 (Dimensión x): Seleccionamos la mediana de los valores de x como el discriminador. En este caso, la mediana de los valores de x es 6. Dividimos los puntos en aquellos con x < 6 y x >= 6.

- Subárbol izquierdo: (3, 6), (2, 7), (9, 1)

- Subárbol derecho: (17, 15), (13, 15), (6, 12), (10, 19)

2. Nivel 2 (Dimensión y): Para el subárbol izquierdo, seleccionamos la mediana de los valores de y como el discriminador. En este caso, la mediana de los valores de y es 6. Dividimos los puntos en aquellos con y < 6 y y >= 6.

- Subárbol izquierdo del subárbol izquierdo: (9, 1)

- Subárbol derecho del subárbol izquierdo: (3, 6), (2, 7)

3. Nivel 2 (Dimensión y): Para el subárbol derecho, seleccionamos la mediana de los valores de y como el discriminador. En este caso, la mediana de los valores de y es 15. Dividimos los puntos en aquellos con y < 15 y y >= 15.

- Subárbol izquierdo del subárbol derecho: (6, 12)

- Subárbol derecho del subárbol derecho: (17, 15), (13, 15), (10, 19)

El proceso continúa recursivamente hasta que todos los puntos estén incluidos en el KD-Tree. Este ejemplo ilustra cómo se construye un KD-Tree dividiendo el espacio en subconjuntos más pequeños a lo largo de diferentes dimensiones.

5. Algoritmos Asociados a KD-Trees

Los KD-Trees son acompañados por varios algoritmos que permiten realizar operaciones eficientes en la estructura. Entre los algoritmos más importantes se encuentran:

5.1 Inserción

El algoritmo de inserción en un KD-Tree se realiza de manera recursiva. Comienza en la raíz y desciende por el árbol hasta encontrar el lugar adecuado para insertar el nuevo punto. Una vez encontrado, se inserta el punto en un nodo hoja y se ajusta la estructura del árbol según sea necesario para mantener las propiedades de un KD-Tree.

5.2 Búsqueda de Vecinos Más Cercanos (k-NN)

La búsqueda de vecinos más cercanos en un KD-Tree es un proceso fundamental en muchas aplicaciones, como la clasificación y la recuperación de información. El algoritmo k-NN en un KD-Tree implica encontrar los k puntos más cercanos a un punto de consulta dado. Esto se logra de manera eficiente utilizando técnicas de búsqueda en el árbol y aprovechando la estructura jerárquica para descartar subárboles enteros cuando sea posible.

5.3 Eliminación

La eliminación de un punto de un KD-Tree puede ser un proceso complejo debido a la necesidad de mantener la estructura balanceada del árbol. El algoritmo de eliminación implica encontrar el nodo que contiene el punto a eliminar, eliminarlo y ajustar la estructura del árbol según sea necesario para preservar las propiedades de un KD-Tree, como el balance y la correcta subdivisión del espacio.

Estos algoritmos son fundamentales para el funcionamiento de los KD-Trees y permiten realizar operaciones de manipulación y consulta de datos de manera eficiente en espacios de múltiples dimensiones.

6. Aplicaciones de KD-Trees

Los KD-Trees tienen una amplia gama de aplicaciones en diversos campos debido a su eficiencia en el manejo de datos multidimensionales. Algunas de las aplicaciones más comunes incluyen:

6.1 Búsqueda espacial

En sistemas de información geográfica (GIS), mapas interactivos y motores de búsqueda basados en ubicación, los KD-Trees se utilizan para realizar búsquedas eficientes de puntos dentro de áreas delimitadas o cercanas a un punto de referencia.

6.2 Procesamiento de imágenes

En aplicaciones de visión por computadora, reconocimiento de patrones y búsqueda de características, los KD-Trees se emplean para buscar puntos similares en imágenes, emparejar objetos o encontrar regiones de interés en el espacio de características.

6.3 Machine Learning

Los KD-Trees se utilizan en algoritmos de aprendizaje automático, como los clasificadores k-NN, para buscar rápidamente los vecinos más cercanos a un punto de consulta en el espacio de características y tomar decisiones de clasificación o regresión.

6.4 Robótica

En aplicaciones de robótica, como la planificación de trayectorias y la navegación autónoma, los KD-Trees se emplean para buscar eficientemente puntos de referencia en el entorno circundante y evitar obstáculos en el camino del robot.

Estas son solo algunas de las muchas aplicaciones de los KD-Trees en la práctica. Su capacidad para manejar eficientemente datos multidimensionales los hace invaluables en una amplia gama de aplicaciones en la informática y la ingeniería.

7. Ventajas y Desventajas

Los KD-Trees ofrecen varias ventajas y desventajas que deben tenerse en cuenta al considerar su aplicación en diferentes contextos. A continuación, se detallan algunas de las principales:

7.1 Ventajas

1. **Eficiencia en Búsqueda:** Los KD-Trees permiten búsquedas rápidas y eficientes de puntos en espacios multidimensionales, lo que los hace ideales para aplicaciones que requieren recuperación rápida de datos.
2. **Simplicidad de Implementación:** Comparados con otras estructuras de datos complejas, los KD-Trees son relativamente simples de implementar y entender, lo que facilita su uso en una variedad de aplicaciones.
3. **Flexibilidad:** Los KD-Trees admiten una variedad de operaciones, incluida la inserción, la eliminación y la búsqueda de vecinos más cercanos, lo que los hace versátiles para diferentes escenarios de aplicación.

7.2 Desventajas

1. **Desbalanceo:** Con inserciones y eliminaciones frecuentes, los KD-Trees tienden a desbalancearse, lo que puede afectar negativamente el rendimiento de las operaciones de búsqueda y consulta.
2. **Maldición de la Dimensionalidad:** En espacios de alta dimensionalidad, la eficiencia de los KD-Trees puede disminuir significativamente debido al fenómeno conocido como la "maldición de la dimensionalidad".

3. **Costo de Construcción:** La construcción inicial de un KD-Tree puede ser costosa en términos de tiempo y recursos computacionales, especialmente para conjuntos de datos grandes o de alta dimensionalidad.

8. Optimización y Balanceo de KD-Trees

Dado que los KD-Trees pueden volverse desbalanceados con inserciones y eliminaciones frecuentes, es importante considerar técnicas de optimización y balanceo para garantizar un rendimiento óptimo. Algunas de las estrategias comunes incluyen:

8.1 Técnicas de balanceo

1. **Rotación de Nodos:** Se pueden aplicar rotaciones en el árbol para reorganizar su estructura y mejorar el balance. Esto implica intercambiar nodos dentro del árbol para redistribuir los puntos de manera más uniforme.
2. **Reorganización de Puntos:** Algunas implementaciones de KD-Trees pueden reorganizar los puntos dentro de los nodos para mejorar el balance y la eficiencia de las operaciones de búsqueda.

8.2 Rebalanceo dinámico

1. **Rebalanceo Automático:** Se pueden implementar algoritmos que monitorean la estructura del árbol y aplican operaciones de rebalanceo automáticamente cuando sea necesario, manteniendo así el árbol en un estado óptimo.

2. **División y Fusión de Nodos:** Se pueden dividir o fusionar nodos según ciertos criterios para mantener el balance del árbol durante la inserción y eliminación de puntos.

9. Comparación con Otras Estructuras de Datos

Los KD-Trees son solo una de varias estructuras de datos utilizadas para manejar conjuntos de datos multidimensionales. Es importante comparar los KD-Trees con otras estructuras para comprender mejor sus fortalezas y debilidades. Algunas de las estructuras de datos más comunes para este propósito incluyen:

9.1 Quad-Trees

Los Quad-Trees son árboles de cuadrantes utilizados principalmente en aplicaciones espaciales donde el espacio se divide recursivamente en cuadrantes. Aunque son eficientes para áreas bidimensionales, pueden volverse ineficaces en dimensiones más altas debido al crecimiento exponencial del número de cuadrantes.

9.2 R-Trees

Los R-Trees son árboles de regiones utilizados para indexar datos espaciales en varias dimensiones. A diferencia de los KD-Trees, que dividen el espacio de manera equitativa, los R-Trees agrupan los puntos en regiones rectangulares para optimizar las búsquedas en áreas específicas.

9.3 Hashing Espacial

El hashing espacial utiliza funciones de hash para asignar puntos a celdas en una cuadrícula multidimensional. Si bien puede ser eficiente para búsquedas de vecinos cercanos, el rendimiento puede variar dependiendo de la distribución de los puntos y la función de hash utilizada.

9. 4 Comparación

- Los KD-Trees son eficientes para buscar puntos en dimensiones más altas y pueden adaptarse bien a datos distribuidos de manera desigual.
- Los Quad-Trees son útiles para representar estructuras jerárquicas en áreas bidimensionales, pero pueden ser menos eficientes en dimensiones más altas.
- Los R-Trees son ideales para consultas de rango en áreas específicas, pero pueden tener problemas con la sobrecarga de datos y la fragmentación.
- El hashing espacial puede ser rápido para búsquedas de vecinos cercanos, pero puede tener problemas con la colisión de hash y la resolución de celdas.

10. Consideraciones de Implementación

Al implementar KD-Trees en una aplicación, es importante tener en cuenta varias consideraciones que pueden afectar su rendimiento y eficacia. Algunas de estas consideraciones incluyen:

10.1 Complejidad temporal y espacial

Es fundamental comprender la complejidad temporal y espacial de las operaciones en un KD-Tree. La búsqueda, inserción y eliminación de puntos

tienen complejidades que dependen del número de puntos en el árbol y la dimensionalidad del espacio.

10.2 Manejo de datos de alta dimensionalidad

Los KD-Trees pueden sufrir de la "maldición de la dimensionalidad" en espacios de alta dimensionalidad, donde la eficiencia de las operaciones puede verse significativamente afectada. Es importante considerar técnicas de reducción de dimensionalidad o utilizar estructuras alternativas en tales casos.

10.3 Selección de Dimensiones

La selección de la dimensión adecuada para subdividir el espacio en cada nivel del árbol puede afectar el rendimiento de las operaciones. Es importante evaluar diferentes estrategias de selección de dimensiones y su impacto en el rendimiento global del árbol.

10.4 Escalabilidad y Eficiencia

A medida que el tamaño del conjunto de datos crece, es crucial garantizar que el KD-Tree pueda escalar de manera eficiente para manejar conjuntos de datos más grandes. Esto puede implicar técnicas de división de nodos y rebalanceo dinámico para mantener un rendimiento óptimo.

10.5 Optimizaciones Específicas de Aplicación

Dependiendo de la aplicación específica, puede ser necesario implementar optimizaciones adicionales para adaptar el KD-Tree a los requisitos y restricciones del problema en cuestión. Esto puede incluir técnicas de

preprocesamiento de datos, caching o paralelización para mejorar el rendimiento.

11. Casos de Estudio y Ejemplos Prácticos

Los KD-Trees han sido ampliamente utilizados en una variedad de campos y aplicaciones. Algunos casos de estudio y ejemplos prácticos de su aplicación incluyen:

11.1 Reconocimiento de Imágenes

En sistemas de reconocimiento de imágenes, los KD-Trees se utilizan para indexar y buscar características visuales en grandes bases de datos de imágenes. Por ejemplo, en la búsqueda de imágenes similares o en la recuperación de información visual.

11.2 Búsqueda Espacial

En aplicaciones de sistemas de información geográfica (SIG), los KD-Trees se emplean para realizar consultas espaciales eficientes, como la búsqueda de puntos de interés cercanos o la búsqueda de rutas óptimas en mapas.

11.3 Clasificación y Regresión

En el aprendizaje automático, los KD-Trees se utilizan en algoritmos de clasificación y regresión basados en vecinos más cercanos (k-NN), donde los KD-Trees se utilizan para buscar rápidamente los vecinos más cercanos a un punto de consulta en el espacio de características.

11.4 Procesamiento de Nubes de Puntos

En aplicaciones de visión por computadora y realidad aumentada, los KD-Trees se utilizan para manejar nubes de puntos tridimensionales, lo que permite realizar consultas y operaciones eficientes en datos de escaneo 3D.

11.5 Búsqueda de Colisiones en Simulaciones Físicas

En simulaciones físicas y videojuegos, los KD-Trees se utilizan para detectar colisiones entre objetos en un entorno tridimensional, lo que permite una simulación precisa y eficiente de interacciones físicas.

Estos casos de estudio y ejemplos prácticos ilustran la versatilidad y utilidad de los KD-Trees en una variedad de aplicaciones del mundo real, desde el procesamiento de imágenes hasta la simulación física y más allá.

12. Herramientas y Librerías para KD-Trees

Para implementar y trabajar con KD-Trees, existen varias herramientas y librerías disponibles en diferentes lenguajes de programación. Algunas de las más populares incluyen:

12.1 Implementaciones en lenguajes populares (C++, Python, Java)

C++

- **CGAL:** Computational Geometry Algorithms Library (CGAL) ofrece implementaciones eficientes de estructuras de datos geométricas,

incluidos KD-Trees, para aplicaciones en computación geométrica y ciencia computacional.

- **FLANN:** Fast Library for Approximate Nearest Neighbors (FLANN) proporciona implementaciones rápidas y eficientes de KD-Trees y otras estructuras de datos para búsqueda de vecinos más cercanos en grandes conjuntos de datos.

Python

- **scikit-learn:** La biblioteca scikit-learn ofrece una implementación fácil de usar de KD-Trees y otras estructuras de datos para aprendizaje automático y minería de datos en Python. Es especialmente útil para algoritmos de clasificación y regresión basados en vecinos más cercanos.
- **SciPy:** La biblioteca SciPy proporciona funciones para la manipulación y el procesamiento de datos científicos en Python, incluida una implementación de KD-Tree que se utiliza para realizar consultas espaciales y análisis de vecinos más cercanos.

Java

- **JTS:** Java Topology Suite (JTS) es una biblioteca de código abierto para realizar operaciones espaciales y geométricas en Java. Ofrece soporte para KD-Trees y otras estructuras de datos geométricas para aplicaciones en SIG y procesamiento de datos espaciales.

12.2 Comparación de rendimiento

Al comparar el rendimiento de diferentes implementaciones de KD-Trees, es importante tener en cuenta varios factores, como la eficiencia en tiempo y espacio, la escalabilidad y la facilidad de uso. Algunas consideraciones a tener en cuenta al comparar el rendimiento incluyen:

- Velocidad de construcción del árbol: El tiempo necesario para construir el KD-Tree a partir de un conjunto de datos dado.
- Tiempo de búsqueda de vecinos más cercanos: El tiempo necesario para encontrar los vecinos más cercanos a un punto de consulta en el árbol.
- Eficiencia en espacios de alta dimensionalidad: La capacidad del KD-Tree para manejar eficientemente conjuntos de datos en dimensiones más altas.
- Uso de memoria: La cantidad de memoria necesaria para almacenar el KD-Tree y realizar operaciones en él.

La comparación de rendimiento puede variar según el tipo de datos, el tamaño del conjunto de datos y las operaciones específicas realizadas en el KD-Tree. Es importante realizar pruebas exhaustivas en diferentes escenarios para determinar la mejor implementación para una aplicación particular.

13. Conclusión

Los KD-Trees son estructuras de datos poderosas y versátiles que permiten organizar y manipular eficientemente conjuntos de datos en espacios multidimensionales. A lo largo de este informe, hemos explorado en detalle los

principios fundamentales, la estructura, los algoritmos asociados, las aplicaciones y las consideraciones prácticas relacionadas con los KD-Trees.

A través de su capacidad para realizar operaciones como búsqueda de vecinos más cercanos, inserción y eliminación de manera eficiente, los KD-Trees encuentran aplicaciones en una amplia gama de campos, incluyendo sistemas de información geográfica, procesamiento de imágenes, aprendizaje automático, robótica y más.

Sin embargo, también es importante tener en cuenta las limitaciones y desafíos asociados con los KD-Trees, como la necesidad de balanceo y optimización, la "maldición de la dimensionalidad" en espacios de alta dimensionalidad y las consideraciones de implementación específicas del problema.

En resumen, los KD-Trees son una herramienta poderosa y valiosa para el manejo de datos multidimensionales en aplicaciones del mundo real. Al comprender sus principios y aplicaciones, así como las mejores prácticas para su implementación y optimización, los desarrolladores pueden aprovechar al máximo el potencial de los KD-Trees para resolver una amplia variedad de problemas computacionales y científicos.

14. Referencias Bibliográficas

Bentley, JL (1975). Árboles de búsqueda binarios multidimensionales utilizados para búsqueda asociativa. Comunicaciones de la ACM, 18(9), 509-517.

Friedman, JH, Bentley, JL y Finkel, RA (1977). Un algoritmo para encontrar las mejores coincidencias en el tiempo esperado logarítmico. Transacciones ACM sobre software matemático (TOMS), 3(3), 209-226.

Algoritmo para la construcción de un KD-Tree.

1. **Inicializar:** Comienza con una lista de puntos en un espacio K-dimensional.

2. **Dividir y Conquistar:** Utiliza un enfoque de dividir y conquistar para construir el árbol recursivamente.

3. **Elegir la Dimensión de División:** Selecciona una dimensión (eje) para dividir los puntos. Generalmente, se elige de manera cíclica (por ejemplo, en una dimensión de 3, el primer nivel es x, el segundo y, el tercero z, y luego se repite).

4. **Ordenar los Puntos:** Ordena los puntos según la dimensión de división seleccionada.

5. **Mediana:** Encuentra la mediana de los puntos en la dimensión de división. La mediana será el nodo de división en el árbol actual.

6. **Subdivisión:** Los puntos a la izquierda de la mediana irán al subárbol izquierdo y los puntos a la derecha irán al subárbol derecho.

7. **Repetir:** Repite el proceso recursivamente para los subárboles izquierdo y derecho hasta que no queden puntos.

En JavaScript

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

function buildKDTree(points, depth = 0, k = 2) {
    if (points.length === 0) {
```

```
        return null;
    }

    const axis = depth % k;

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

    const median = Math.floor(points.length / 2);

    return new Node(
        points[median],
        buildKDTree(points.slice(0, median), depth + 1, k),
        buildKDTree(points.slice(median + 1), depth + 1, k)
    );
}
function printKDTree(node, depth = 0) {
    if (node !== null) {
        console.log("  ".repeat(depth) + node.point);
        printKDTree(node.left, depth + 1);
        printKDTree(node.right, depth + 1);
    }
}

// Ejemplo de uso
const points = [
    [2, 3],
    [5, 4],
    [9, 6],
    [4, 7],
    [8, 1],
    [7, 2]
];
const kdTree = buildKDTree(points);
printKDTree(kdTree);
```

GITHUB

https://github.com/OliverChoque/KD-Tree.git

Implementación de KD-tree en 2D en JavaScript

Pedidos ChatGPT

- Aplicación de KD-Tree
- Aplicación de KD-Tree, puntos generados con el click derecho del mouse.
- Árbol de KD-Tree
- El resto de modificación es Propia y el ajuste al programa.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>KD-Tree en 2D</title>
    <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.
css" rel="stylesheet">
    <style>
        body {
            background-color: rgb(2, 11, 36);
            color: white;
            margin: 0;
            padding: 0;
        }
        #canvas, #treeCanvas {
            border: 5px solid #448ada;
            display: block;
            margin: auto;
            background-color: white;
        }
        #canvas-container {
            display: flex;
            justify-content: center;
            align-items: center;
            height: calc(90vh - 50px);
```

```
        }
        h1 {
            margin-top: 15px;
        }
    </style>
</head>
<body>
    <div class="container-fluid text-center">
        <h1>KD-Tree en 2D</h1>
        <div id="canvas-container">
            <canvas id="canvas" width="800" height="800"></canvas>
            <canvas id="treeCanvas" width="800" height="800"></canvas>
        </div>
    </div>

    <script>
        class Node {
            constructor(point, axis) {
                this.point = point;
                this.left = null;
                this.right = null;
                this.axis = axis;
            }
        }

        class KDTree {
            constructor() {
                this.root = null;
            }

            insert(point) {
                const insertRec = (node, point, depth) => {
                    if (node === null) {
                        return new Node(point, depth % 2);
                    }
                    const axis = node.axis;
                    if (point[axis] < node.point[axis]) {
                        node.left = insertRec(node.left, point, depth +
1);
                    } else {
```

```
                    node.right = insertRec(node.right, point, depth
+ 1);
                }
                return node;
            };

            this.root = insertRec(this.root, point, 0);
        }

        draw(ctx, node, xMin, xMax, yMin, yMax) {
            if (node === null) {
                return;
            }

            const axis = node.axis;

            ctx.fillStyle = 'black';
            ctx.beginPath();
            ctx.arc(node.point[0], node.point[1], 3, 0, Math.PI * 2,
true);
            ctx.fill();

            if (axis === 0) {
                ctx.strokeStyle = 'red';
                ctx.beginPath();
                ctx.moveTo(node.point[0], yMin);
                ctx.lineTo(node.point[0], yMax);
                ctx.stroke();

                this.draw(ctx, node.left, xMin, node.point[0], yMin,
yMax);
                this.draw(ctx, node.right, node.point[0], xMax,
yMin, yMax);
            } else {
                ctx.strokeStyle = 'blue';
                ctx.beginPath();
                ctx.moveTo(xMin, node.point[1]);
                ctx.lineTo(xMax, node.point[1]);
                ctx.stroke();
```

```
                    this.draw(ctx, node.left, xMin, xMax, yMin,
node.point[1]);
                    this.draw(ctx, node.right, xMin, xMax,
node.point[1], yMax);
                }
            }

            drawTree(ctx, node, x, y, spacingX, spacingY, depth = 0) {
                if (node === null) {
                    return;
                }

                const leftX = x - spacingX;
                const rightX = x + spacingX;
                const nextY = y + spacingY;
                const colors = ['#FF7F50', '#6495ED', '#90EE90',
'#FFD700', '#DDA0DD'];

                ctx.fillStyle = colors[depth % colors.length];
                ctx.beginPath();
                ctx.arc(x, y, 5, 0, Math.PI * 2, true);
                ctx.fill();

                ctx.fillStyle = 'white';
                ctx.textAlign = 'center';
                ctx.fillRect(x - 30, y - 25, 60, 20);
                ctx.fillStyle = 'black';
                ctx.fillText("(" + node.point[0] + ", " + node.point[1]
+ ")", x, y - 10);

                if (node.left !== null) {
                    ctx.strokeStyle = 'black';
                    ctx.beginPath();
                    ctx.moveTo(x, y);
                    ctx.lineTo(leftX, nextY);
                    ctx.stroke();
                    this.drawTree(ctx, node.left, leftX, nextY, spacingX
/ 2, spacingY, depth + 1);
                }

                if (node.right !== null) {
```

```
                    ctx.strokeStyle = 'black';
                    ctx.beginPath();
                    ctx.moveTo(x, y);
                    ctx.lineTo(rightX, nextY);
                    ctx.stroke();
                    this.drawTree(ctx, node.right, rightX, nextY,
spacingX / 2, spacingY, depth + 1);
                }
            }

            clearCanvas(ctx, width, height) {
                ctx.clearRect(0, 0, width, height);
            }
        }

        const canvas = document.getElementById('canvas');
        const treeCanvas = document.getElementById('treeCanvas');
        const ctx = canvas.getContext('2d');
        const treeCtx = treeCanvas.getContext('2d');

        const tree = new KDTree();

        canvas.addEventListener('click', (event) => {
            const rect = canvas.getBoundingClientRect();
            const x = event.clientX - rect.left;
            const y = event.clientY - rect.top;

            tree.insert([x, y]);
            tree.clearCanvas(ctx, canvas.width, canvas.height);
            tree.clearCanvas(treeCtx, treeCanvas.width,
treeCanvas.height);
            tree.draw(ctx, tree.root, 0, canvas.width, 0,
canvas.height);

            tree.drawTree(treeCtx, tree.root, treeCanvas.width / 2, 50,
treeCanvas.width / 4, 70);
        });
    </script>
</body>
</html>
```