Universidad Nacional del Altiplano

FACULTAD DE INGENIERÍA MECÁNICA ELÉCTRICA, ELECTRÓNICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

ESTRUCTURAS DE DATOS AVANZADAS

" KD-Tree"

ALUMNO:

Mamani Quispe Saul Rodrigo

DOCENTE: Ing. COLLANQUI MARTINEZ FREDY

SEMESTRE: VI **GRUPO:** C

Contenido

1.Introducción

Los métodos jerárquicos son técnicas de organización de datos que estructuran la información en múltiples niveles o jerarquías. Estas técnicas son particularmente útiles en la gestión y análisis de grandes volúmenes de datos, ya que permiten una búsqueda y recuperación eficiente de información. Dividen los datos en subconjuntos más pequeños y manejables, facilitando así el acceso y la manipulación de datos específicos dentro de un conjunto más grande. Entre las aplicaciones más comunes de los métodos jerárquicos se encuentran las bases de datos, la minería de datos, la informática gráfica y los sistemas de información geográfica.

Hablando de informática y ciencia de datos, los métodos jerárquicos son utilizados para resolver problemas complejos de clasificación, agrupamiento y búsqueda. Estos métodos permiten realizar operaciones eficientes, como la búsqueda de vecinos más cercanos, la detección de anomalías y la clasificación de objetos.

Dentro de estos métodos, los árboles KD (k-dimensional trees) son una estructura de datos jerárquica especialmente diseñada para gestionar y manipular datos en espacios multidimensionales. Introducidos por Jon Louis Bentley en 1975, los árboles KD son una extensión de los árboles binarios de búsqueda y se utilizan ampliamente para problemas que involucran datos multidimensionales, como la búsqueda de vecinos más cercanos, la búsqueda de rangos y la clasificación.

Estos organizan los datos dividiendo recursivamente el espacio en regiones más pequeñas mediante la selección de puntos de división a lo largo de diferentes dimensiones en cada nivel del árbol. Cada nodo del árbol representa una partición del espacio de datos y almacena un punto de datos junto con dos hijos, que representan las subdivisiones de ese espacio. Esta estructura permite que los árboles KD sean

extremadamente eficientes para la búsqueda y recuperación de datos en múltiples dimensiones.

Los árboles KD son especialmente útiles en aplicaciones que requieren una búsqueda rápida y eficiente de datos cercanos en un espacio multidimensional. Ejemplos de tales aplicaciones incluyen la recuperación de imágenes basada en contenido, la clasificación de patrones en reconocimiento de voz y la búsqueda de rutas en sistemas de navegación.

Los ambos métodos ofrecen una forma poderosa y eficiente de organizar y buscar datos multidimensionales, proporcionando una base sólida para muchas aplicaciones avanzadas en informática y ciencia de datos.

2.Estructura de un kd-tree

2.1 Nodo

En un KD-Tree, cada nodo representa un punto en el espacio k-dimensional y almacena la siguiente información:

- **Punto (Point)**: Este es el dato principal almacenado en el nodo. En un espacio k-dimensional, un punto puede ser representado como un vector de k elementos. Por ejemplo, en un espacio tridimensional, un punto se representaría como (x,y,z)(x, y, z)(x,y,z). Este punto es crucial porque es el elemento que se utiliza para comparar y dividir el espacio en subregiones.
- **Izquierdo (Left)**: Una referencia al subárbol izquierdo. Este subárbol contiene todos los puntos cuya coordenada en la dimensión del discriminador es menor o igual que la del punto en el nodo actual. En otras palabras, todos los puntos en el subárbol izquierdo están "a la izquierda" del punto actual en la dimensión del discriminador.

- **Derecho (Right)**: Una referencia al subárbol derecho. Este subárbol contiene todos los puntos cuya coordenada en la dimensión del discriminador es mayor que la del punto en el nodo actual. Todos los puntos en el subárbol derecho están "a la derecha" del punto actual en la dimensión del discriminador.

El nodo es la estructura fundamental del KD-Tree, y cada nodo sigue la misma estructura, permitiendo la recursividad necesaria para construir y navegar el árbol.

2.2 Discriminador (Discriminator)

El discriminador es el criterio utilizado para dividir los puntos en el espacio. Este criterio cambia en cada nivel del árbol y se elige de manera cíclica entre las dimensiones del espacio k-dimensional. Aquí se explica cómo funciona este mecanismo:

- **Nivel 0 (Discriminador x)**: En el nivel más alto del árbol (la raíz), se selecciona la primera dimensión, comúnmente la dimensión x. Esto significa que el punto en el nodo raíz divide el espacio en dos mitades basándose en su coordenada x. Todos los puntos con una coordenada x menor o igual que la del punto en el nodo raíz irán al subárbol izquierdo, y los que tienen una coordenada x mayor irán al subárbol derecho.
- **Nivel 1 (Discriminador y)**: En el siguiente nivel, se selecciona la segunda dimensión, generalmente la dimensión y. Ahora, cada punto en este nivel divide el espacio en dos mitades basándose en su coordenada y. Este proceso se continúa de manera cíclica.
- **Nivel 2 (Discriminador z)**: Si el espacio es tridimensional, en el siguiente nivel se seleccionará la tercera dimensión, la dimensión z, y así sucesivamente.

El discriminador cambia en cada nivel del árbol de manera cíclica (x, y, z, x, y, z, ...). Esto asegura que el árbol esté equilibrado y que las divisiones sean consistentes a lo largo de todas las dimensiones.

2.3 Subárboles (Subtrees)

Cada nodo en un KD-Tree tiene dos subárboles, que se derivan del punto en el nodo actual y su discriminador:

- **Subárbol izquierdo (Left Subtree)**: Este subárbol contiene todos los puntos que tienen una coordenada en la dimensión del discriminador menor o igual que la del punto del nodo. Por ejemplo, si el discriminador es la dimensión x, el subárbol izquierdo contendrá todos los puntos con una coordenada x menor o igual que la del punto en el nodo actual. Esta estructura recursiva permite que el árbol divida el espacio de manera eficiente, facilitando operaciones como la búsqueda de vecinos más cercanos y la búsqueda de rango.
- **Subárbol derecho (Right Subtree)**: Este subárbol contiene todos los puntos que tienen una coordenada en la dimensión del discriminador mayor que la del punto del nodo. Siguiendo con el ejemplo anterior, si el discriminador es la dimensión x, el subárbol derecho contendrá todos los puntos con una coordenada x mayor que la del punto en el nodo actual.

La clave de los subárboles es que permiten que el KD-Tree divida el espacio de manera jerárquica, donde cada nivel del árbol refina aún más la partición del espacio. Esto es fundamental para la eficiencia del KD-Tree en operaciones de búsqueda.

2.4 Pseudocódigo para la construcción de un KD-Tree

```
class Nodo:
    def __init__(self, punto, discriminador=None, izquierda=None,
derecha=None):
        self.punto = punto
        self.discriminador = discriminador
        self.izquierda = izquierda
        self.derecha = derecha

def construir_kd_tree(puntos, profundidad=0):
    if not puntos:
        return None

    k = len(puntos[0])  # número de dimensiones
    discriminador = profundidad % k

    puntos.sort(key=lambda punto: punto[discriminador])
    mediana = len(puntos) // 2

    return Nodo(
        punto=puntos[mediana],
        discriminador=discriminador,
        izquierda=construir_kd_tree(puntos[:mediana], profundidad +
1),
        derecha=construir_kd_tree(puntos[mediana + 1:], profundidad +
1)
    )
```

3. Construcción de un KD-Trees

Para la construcción de un KD-Tree, consideremos un ejemplo con un conjunto de puntos en un espacio bidimensional (2D). Supongamos que tenemos los siguientes puntos:

Puntos = {(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)}

3.1 Pasos para Construir el KD-Tree

1. Nodo Raíz

- **Nivel 0**: El discriminador es la dimensión xxx.

- Ordenamos los puntos según su coordenada xxx: (2,3),(4,7),(5,4),(7,2),(8,1),(9,6)(2, 3), (4, 7), (5, 4), (7, 2), (8, 1), (9, 6)(2,3),(4,7),(5,4),(7,2),(8,1),(9,6).
- Seleccionamos el punto de la mediana como el nodo raíz: (7,2)(7, 2)(7,2).

2. Subárbol Izquierdo del Nodo Raíz

- **Nivel 1**: El discriminador es la dimensión yyy.
- Consideramos los puntos a la izquierda del nodo raíz: (2,3),(4,7),(5,4)(2, 3), (4, 7), (5, 4)(2,3),(4,7),(5,4).
- Ordenamos estos puntos según su coordenada yyy: (2,3),(5,4),(4,7)(2, 3), (5, 4), (4, 7)(2,3),(5,4),(4,7).
- Seleccionamos el punto de la mediana: (5,4)(5, 4)(5,4), que será el nodo izquierdo del nodo raíz.

3. Subárbol Derecho del Nodo Raíz

- **Nivel 1**: El discriminador es la dimensión yyy.
- Consideramos los puntos a la derecha del nodo raíz: (8,1),(9,6)(8, 1), (9, 6)(8,1),(9,6).
- Ordenamos estos puntos según su coordenada yyy: (8,1),(9,6)(8, 1), (9, 6)(8,1),(9,6).
- Seleccionamos el punto de la mediana: (9,6)(9, 6)(9,6), que será el nodo derecho del nodo raíz.

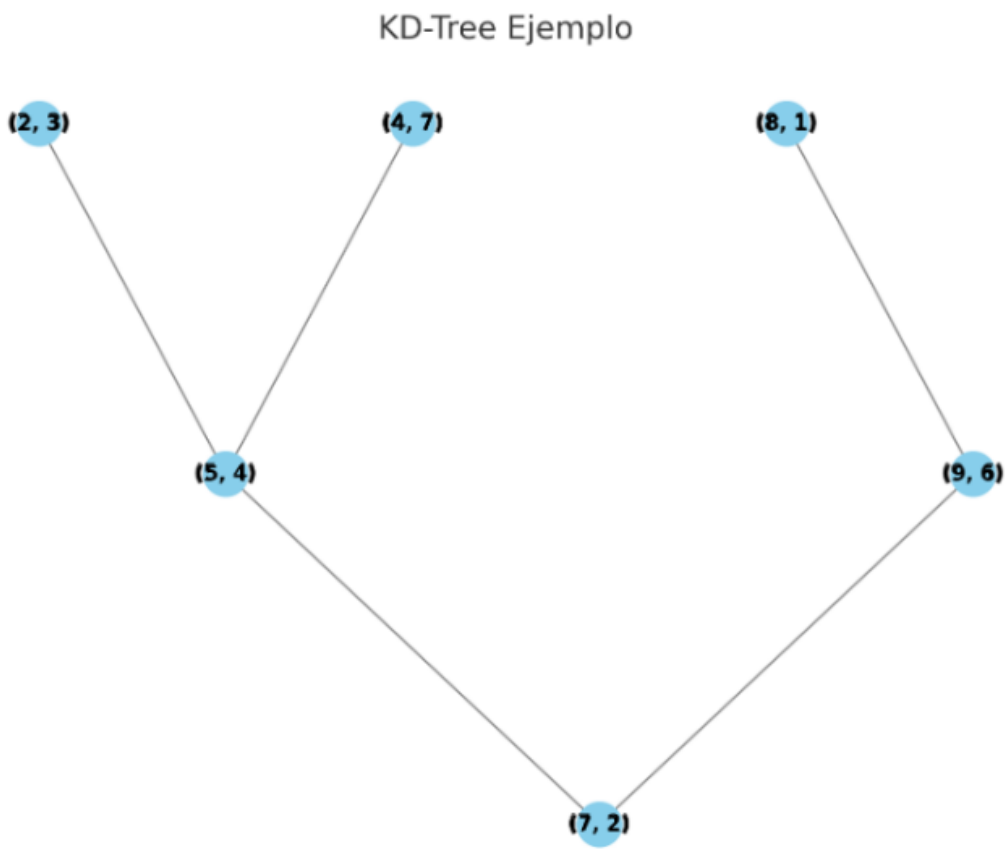
4. Subárboles del Nodo Izquierdo (5,4)(5, 4)(5,4)

- **Nivel 2**: El discriminador es la dimensión xxx.

- Para los puntos a la izquierda de (5,4)(5, 4)(5,4): (2,3)(2, 3)(2,3), seleccionamos este punto directamente ya que solo queda un punto.
- **Nivel 2**: El discriminador es la dimensión xxx.
- Para los puntos a la derecha de (5,4)(5, 4)(5,4): (4,7)(4, 7)(4,7), seleccionamos este punto directamente ya que solo queda un punto.

5. Subárboles del Nodo Derecho (9,6) (9, 6) (9,6)

- **Nivel 2**: El discriminador es la dimensión xxx.
- Para los puntos a la izquierda de (9,6)(9, 6)(9,6): (8,1)(8, 1)(8,1), seleccionamos este punto directamente ya que solo queda un punto.



3.2 Pseudocódigo para la Construcción de un KD-Tree

```
class Nodo:
    def __init__(self, punto, discriminador=None, izquierda=None,
derecha=None):
        self.punto = punto
        self.discriminador = discriminador
        self.izquierda = izquierda
        self.derecha = derecha

def construir_kd_tree(puntos, profundidad=0):
    if not puntos:
        return None

    k = len(puntos[0])  # número de dimensiones
    discriminador = profundidad % k

    # Ordenar puntos según la dimensión del discriminador
    puntos.sort(key=lambda punto: punto[discriminador])
    mediana = len(puntos) // 2

    # Crear nodo y construir recursivamente subárboles
    return Nodo(
        punto=puntos[mediana],
        discriminador=discriminador,
        izquierda=construir_kd_tree(puntos[:mediana], profundidad +
1),
        derecha=construir_kd_tree(puntos[mediana + 1:], profundidad +
1)
    )

# Ejemplo de uso
puntos = [(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)]
kd_tree = construir_kd_tree(puntos)
```

4. Algoritmos asociados a kd-trees

4.1 Inserción

La inserción de un punto en un KD-Tree sigue un proceso recursivo que asegura que el nuevo punto se coloque en el lugar correcto, respetando las divisiones del espacio multidimensional en base a los discriminadores.

Proceso de Inserción:

1. **Iniciar en la Raíz**: Comienza en el nodo raíz del árbol.

2. **Seleccionar Dimensión de Comparación**: En cada nivel del árbol, selecciona la dimensión de comparación basada en la profundidad del nodo. La dimensión se elige cíclicamente (por ejemplo, en un espacio 2D, las dimensiones se alternan entre xxx y yyy).
3. **Comparar y Desplazar**: Compara el nuevo punto con el punto en el nodo actual en la dimensión de comparación:
    - Si el valor del nuevo punto en la dimensión de comparación es menor que o igual al valor del punto del nodo, desplázate al subárbol izquierdo.
    - Si el valor del nuevo punto es mayor, desplázate al subárbol derecho.
4. **Encontrar una Posición Vacía**: Continua el proceso recursivo hasta encontrar un nodo vacío, donde se insertará el nuevo punto.
5. **Crear un Nuevo Nodo**: Inserta el nuevo punto creando un nodo en la posición vacía encontrada.

Ventajas de la Inserción en KD-Tree:

- **Balance del Árbol**: Insertar puntos siguiendo esta metodología ayuda a mantener el árbol equilibrado, lo que es crucial para la eficiencia de las búsquedas.
- **Eficiencia**: La inserción se realiza en tiempo O(log⁡N)O(\log N)O(logN) en promedio para un árbol balanceado, donde NNN es el número de puntos en el árbol.

4.1.1 Pseudocódigo para Inserción

```
class Nodo:
    def __init__(self, punto, discriminador=None, izquierda=None,
derecha=None):
        self.punto = punto
        self.discriminador = discriminador
        self.izquierda = izquierda
        self.derecha = derecha

def insertar_kd_tree(nodo, punto, profundidad=0):
    if nodo is None:
        return Nodo(punto, discriminador=profundidad % len(punto))

    dim = nodo.discriminador

    if punto[dim] < nodo.punto[dim]:
        nodo.izquierda = insertar_kd_tree(nodo.izquierda, punto,
profundidad + 1)
    else:
        nodo.derecha = insertar_kd_tree(nodo.derecha, punto,
profundidad + 1)

    return nodo

# Ejemplo de uso
puntos = [(7, 2), (5, 4), (9, 6), (2, 3), (4, 7), (8, 1)]
kd_tree = None
for punto in puntos:
    kd_tree = insertar_kd_tree(kd_tree, punto)
```

4.2 Búsqueda de Vecinos Más Cercanos (k-NN)

La búsqueda de vecinos más cercanos (k-NN) en un KD-Tree es un algoritmo que permite encontrar los puntos más cercanos a un punto de consulta en el espacio k-dimensional. Este algoritmo es fundamental en aplicaciones como la clasificación y la regresión en aprendizaje automático.

4.2.1 Proceso de Búsqueda k-NN:

1. **Inicializar**: Inicia en la raíz del árbol y establece un heap (cola de prioridad) para almacenar los k vecinos más cercanos encontrados hasta el momento.
2. **Recorrer el Árbol**: Desciende recursivamente por el árbol, seleccionando las ramas a explorar en base a la distancia del punto de consulta:

- Calcula la distancia euclidiana (o la distancia cuadrada) entre el punto de consulta y el punto en el nodo actual.
- Mantén un registro de los mejores k resultados encontrados en el heap.

3. **Comparar y Actualizar el Heap**:
   - Si el heap tiene menos de k elementos, agrega el nodo actual.
   - Si el heap está lleno y el nuevo punto es más cercano que el más lejano en el heap, reemplaza el más lejano.
4. **Explorar Subárboles Potenciales**: Recurre primero en el subárbol que es más probable que contenga el punto más cercano. Si la distancia al otro subárbol podría contener un punto más cercano (basado en la dimensión de comparación), también recurre en ese subárbol.
5. **Terminar la Recursión**: El proceso termina cuando todos los subárboles relevantes han sido explorados.

Ventajas del Algoritmo k-NN en KD-Trees:

- **Eficiencia**: El algoritmo es más eficiente que la búsqueda exhaustiva, especialmente en espacios de baja a media dimensionalidad, y se ejecuta en tiempo O(log⁡N)O(\log N)O(logN) para la búsqueda de vecinos más cercanos en el mejor de los casos.
- **Precisión**: Proporciona una manera eficaz de encontrar los puntos más cercanos, crucial para muchas aplicaciones prácticas.

4.2.2 Pseudocódigo para Búsqueda de Vecinos Más Cercanos

```
import heapq

def distancia_cuadrada(p1, p2):
    return sum((x - y) ** 2 for x, y in zip(p1, p2))

def k_nn(kd_tree, punto, k=1):
    mejor_distancia = float('inf')
```

```
    mejor_nodo = None
    heap = []

    def buscar(nodo, profundidad=0):
        nonlocal mejor_distancia, mejor_nodo

        if nodo is None:
            return

        distancia = distancia_cuadrada(punto, nodo.punto)

        if len(heap) < k:
            heapq.heappush(heap, (-distancia, nodo.punto))
        elif distancia < -heap[0][0]:
            heapq.heappushpop(heap, (-distancia, nodo.punto))

        dim = nodo.discriminador
        if punto[dim] < nodo.punto[dim]:
            buscar(nodo.izquierda, profundidad + 1)
            if abs(punto[dim] - nodo.punto[dim]) < mejor_distancia:
                buscar(nodo.derecha, profundidad + 1)
        else:
            buscar(nodo.derecha, profundidad + 1)
            if abs(punto[dim] - nodo.punto[dim]) < mejor_distancia:
                buscar(nodo.izquierda, profundidad + 1)

    buscar(kd_tree)

    return [punto for distancia, punto in heap]

# Ejemplo de uso
punto_consulta = (5, 5)
vecinos_mas_cercanos = k_nn(kd_tree, punto_consulta, k=3)
```

4.3 Eliminación

La eliminación de un nodo en un KD-Tree es una operación que requiere atención para mantener la estructura del árbol. A diferencia de la inserción y la búsqueda, la eliminación puede implicar la reconstrucción parcial del árbol.

Proceso de Eliminación:

1. **Encontrar el Nodo a Eliminar**: Comienza en la raíz y desciende recursivamente por el árbol para encontrar el nodo que contiene el punto a eliminar, usando el discriminador cíclico.
2. **Eliminar el Nodo**:
    - **Nodo Hoja**: Si el nodo a eliminar es una hoja, simplemente se elimina.

- **Nodo con un Subárbol**: Si el nodo tiene solo un hijo, ese hijo reemplaza al nodo.
- **Nodo con Dos Subárboles**: Si el nodo tiene dos hijos, se debe encontrar un reemplazo adecuado para mantener la estructura del árbol.
    - **Encontrar el Mínimo en el Subárbol Derecho**: Encuentra el nodo con el valor mínimo en la misma dimensión del nodo a eliminar en el subárbol derecho.
    - **Reemplazar y Eliminar Recursivamente**: Reemplaza el punto del nodo a eliminar con el punto mínimo encontrado y elimina recursivamente el nodo mínimo en el subárbol derecho.

3. **Reconstrucción Parcial**: Puede ser necesario reequilibrar el árbol o ajustar las referencias de los nodos hijos y padres para mantener la propiedad del KD-Tree.

Ventajas del Algoritmo de Eliminación en KD-Trees:

- **Mantenimiento de la Estructura del Árbol**: Garantiza que la estructura del árbol se mantenga balanceada y funcional después de la eliminación.

Flexibilidad: El proceso permite la eliminación de cualquier nodo, no solo las hojas, haciendo el KD-Tree adecuado para aplicaciones dinámicas donde los datos pueden cambiar frecuentemente.

4.3.1 Pseudocódigo para Eliminación:

```
def encontrar_minimo(nodo, dim, profundidad=0):
    if nodo is None:
        return None

    nodo_dim = profundidad % len(nodo.punto)

    if nodo_dim == dim:
        if nodo.izquierda is None:
            return nodo
        return encontrar_minimo(nodo.izquierda, dim, profundidad + 1)
```

```
    izquierda_min = encontrar_minimo(nodo.izquierda, dim, profundidad
+ 1)
    derecha_min = encontrar_minimo(nodo.derecha, dim, profundidad + 1)

    minimo = nodo
    if izquierda_min and izquierda_min.punto[dim] < minimo.punto[dim]:
        minimo = izquierda_min
    if derecha_min and derecha_min.punto[dim] < minimo.punto[dim]:
        minimo = derecha_min

    return minimo

def eliminar_kd_tree(nodo, punto, profundidad=0):
    if nodo is None:
        return None

    dim = profundidad % len(nodo.punto)

    if nodo.punto == punto:
        if nodo.derecha is not None:
            minimo = encontrar_minimo(nodo.derecha, dim, profundidad +
1)
            nodo.punto = minimo.punto
            nodo.derecha = eliminar_kd_tree(nodo.derecha,
minimo.punto, profundidad + 1)
        elif nodo.izquierda is not None:
            minimo = encontrar_minimo(nodo.izquierda, dim, profundidad
+ 1)
            nodo.punto = minimo.punto
            nodo.derecha = eliminar_kd_tree(nodo.izquierda,
minimo.punto, profundidad + 1)
            nodo.izquierda = None
        else:
            return None
    elif punto[dim] < nodo.punto[dim]:
        nodo.izquierda = eliminar_kd_tree(nodo.izquierda, punto,
profundidad + 1)
    else:
        nodo.derecha = eliminar_kd_tree(nodo.derecha, punto,
profundidad + 1)

    return nodo

# Ejemplo de uso
kd_tree = eliminar_kd_tree(kd_tree, (5, 4))
```

5. Aplicación de kd-trees

5.1 Búsqueda Espacial

Los KD-Trees son extremadamente útiles en aplicaciones de búsqueda espacial, donde se necesita encontrar rápidamente puntos cercanos en un espacio multidimensional.

Algunas aplicaciones específicas incluyen:

- **Sistemas de Información Geográfica (SIG)**: Utilizados para buscar y clasificar puntos de interés (POI), como restaurantes, gasolineras, o cualquier tipo de localización geográfica en mapas.
- **Gestión de Datos Espaciales**: En bases de datos espaciales, los KD-Trees ayudan a realizar consultas eficientes de vecinos cercanos, como buscar todos los puntos dentro de una cierta distancia desde un punto dado.
- **Indexación Geoespacial**: Para mejorar la velocidad de acceso y manipulación de grandes conjuntos de datos geoespaciales, los KD-Trees proporcionan una estructura eficiente para organizar y buscar datos.

5.2 Procesamiento de Imágenes

En el procesamiento de imágenes, los KD-Trees son útiles para diversas tareas que implican manipulación y análisis de datos multidimensionales:

- **Detección de Bordes y Características**: KD-Trees ayudan a acelerar la búsqueda de puntos de interés o características específicas en imágenes, como bordes, esquinas, o regiones de interés.
- **Compresión de Imágenes**: Los KD-Trees se pueden utilizar para vector quantization, donde se dividen los datos de la imagen en regiones para reducir el tamaño de los datos sin perder demasiada calidad.
- **Registro de Imágenes**: En tareas donde se necesita alinear dos o más imágenes, los KD-Trees pueden acelerar la búsqueda de correspondencias entre puntos de control en diferentes imágenes.

5.3 Machine Learning

En el ámbito del machine learning, los KD-Trees se aplican en algoritmos de clasificación y regresión, así como en técnicas de reducción de dimensionalidad:

- **Algoritmos k-NN**: Los KD-Trees son una estructura de datos subyacente en la implementación eficiente de algoritmos k-Nearest Neighbors (k-NN), que son utilizados para tareas de clasificación y regresión.
- **Agrupamiento (Clustering)**: KD-Trees facilitan la implementación de algoritmos de agrupamiento, como k-means, al acelerar la búsqueda de centroides más cercanos a los puntos de datos.
- **Reducción de Dimensionalidad**: En técnicas como t-SNE (t-distributed Stochastic Neighbor Embedding), los KD-Trees ayudan a acelerar las búsquedas de vecinos más cercanos en el espacio de alta dimensionalidad.

5.4 Robótica

En robótica, los KD-Trees se utilizan para planificación de movimientos, percepción, y navegación autónoma:

- **Planificación de Trayectorias**: Los KD-Trees ayudan a los robots a encontrar trayectorias eficientes evitando obstáculos y navegando en entornos complejos, mediante la búsqueda rápida de vecinos y puntos de paso.
- **SLAM (Simultaneous Localization and Mapping)**: En el problema de SLAM, donde el robot necesita construir un mapa de su entorno mientras se localiza a sí mismo en ese mapa, los KD-Trees son útiles para procesar y asociar rápidamente características del entorno.

- **Percepción y Reconocimiento de Entornos**: Los sensores de los robots generan datos multidimensionales que se pueden gestionar y analizar eficientemente utilizando KD-Trees, permitiendo al robot reconocer objetos y características del entorno rápidamente.

6. Ventajas y desventajas

6.1 Ventajas

Eficiencia en Búsquedas Multidimensionales:

Los KD-Trees proporcionan una estructura eficiente para búsquedas de vecinos más cercanos y consultas de rango en espacios multidimensionales. Las búsquedas suelen tener una complejidad de O(log⁡N)O(\log N)O(logN) en árboles balanceados, lo cual es significativamente mejor que la búsqueda lineal.

Flexibilidad:

Los KD-Trees pueden manejar datos en cualquier número de dimensiones, lo que los hace aplicables a una amplia variedad de problemas, desde la búsqueda espacial hasta el procesamiento de imágenes y el machine learning.

Capacidad de Manejar Grandes Conjuntos de Datos:

Los KD-Trees son capaces de organizar y manejar eficientemente grandes conjuntos de datos, lo que los hace adecuados para aplicaciones que requieren la manipulación de grandes volúmenes de datos, como la indexación geoespacial y los sistemas de información geográfica.

Soporte para Inserciones y Eliminaciones Dinámicas:

A diferencia de otras estructuras de datos como los árboles de barrido, los KD-Trees permiten la inserción y eliminación de puntos de manera dinámica, lo que es crucial para aplicaciones en tiempo real como la robótica y los sistemas de navegación autónoma.

Reducción de Costos Computacionales:

Al organizar los puntos de datos de manera jerárquica, los KD-Trees pueden reducir drásticamente el número de comparaciones necesarias para encontrar vecinos cercanos o realizar consultas de rango, optimizando así los recursos computacionales.

6.2 Desventajas

1. **Dificultad para Mantener el Equilibrio**:

   Los KD-Trees pueden desbalancearse con inserciones y eliminaciones sucesivas, lo que puede degradar la eficiencia de las operaciones a O(N)O(N)O(N) en el peor de los casos. Mantener un árbol balanceado puede requerir reconstrucciones periódicas, que son costosas en términos de tiempo de computación.

2. **Ineficiencia en Altas Dimensionalidades**:

   Los KD-Trees pierden su eficiencia en espacios de alta dimensionalidad (comúnmente referido como "la maldición de la dimensionalidad"). En estos casos, las divisiones del espacio no son tan efectivas y la eficiencia de las búsquedas puede acercarse a la de una búsqueda lineal.

3. **Complejidad de Implementación**:

   Implementar KD-Trees correctamente puede ser complejo, especialmente en comparación con otras estructuras de datos como los

árboles binarios de búsqueda o los arrays. Esto puede aumentar el tiempo de desarrollo y la probabilidad de errores en el código.

4. **Sobrecarga de Almacenamiento**:

   Los KD-Trees requieren almacenamiento adicional para los punteros a los nodos izquierdo y derecho, lo que puede ser una desventaja en sistemas con memoria limitada.

5. **Dificultad en la Implementación de Algoritmos Complejos**:

   Algoritmos más avanzados como la eliminación o la búsqueda de k vecinos más cercanos en un KD-Tree pueden ser difíciles de implementar y optimizar correctamente, requiriendo un profundo conocimiento de la estructura y sus propiedades.

7. Optimización y balanceo de KD-trees

7.1 Técnicas de Balanceo

Para mantener un KD-Tree eficiente, es fundamental que el árbol esté balanceado. Un árbol balanceado distribuye uniformemente los puntos de datos en todas las ramas, lo que permite que las operaciones de búsqueda e inserción se realicen en tiempo O(log⁡N)O(\log N)O(logN) en promedio. Aquí se presentan algunas técnicas para balancear KD-Trees:

Construcción desde un Conjunto de Datos Estático:

Mediana de Medianas: Al construir un KD-Tree desde cero, seleccionar la mediana de las coordenadas de los puntos en la dimensión correspondiente en cada nivel garantiza un buen balance inicial. La mediana divide el conjunto de datos en dos mitades aproximadamente iguales.

Ordenamiento y Selección: Ordenar los puntos en la dimensión actual y elegir la mediana también asegura un balance adecuado, aunque es menos eficiente que la mediana de medianas en términos de complejidad de tiempo.

Reinserción Total o Parcial:

Reinserción Total: En aplicaciones donde la estructura del KD-Tree se degrada significativamente, puede ser beneficioso reconstruir el árbol desde cero periódicamente. Aunque es costoso en términos de tiempo de computación, esta técnica garantiza un árbol bien balanceado.

Reinserción Parcial: En lugar de reconstruir todo el árbol, se pueden identificar y reconstruir subárboles específicos que se hayan desbalanceado.

Inserción Balanceada:

Inserciones Alternas: Mantener un balance durante la inserción mediante técnicas como la inserción alterna, donde se insertan puntos en subárboles alternos, puede ayudar a evitar desbalances severos.

Estrategias de Carga Balanceada: En aplicaciones de inserciones continuas, implementar estrategias que insertan puntos en el subárbol menos cargado (con menos nodos) puede ayudar a mantener un balance más uniforme.

7.2 Rebalanceo Dinámico

El rebalanceo dinámico es crucial en aplicaciones donde los datos cambian con frecuencia y el árbol se puede desbalancear debido a inserciones y eliminaciones continuas. Algunas técnicas para el rebalanceo dinámico incluyen:

Rebalanceo por Inserción y Eliminación:

- **Verificación y Rebalanceo**: Después de cada inserción o eliminación, verificar la altura del subárbol afectado. Si la diferencia de alturas entre los subárboles de un nodo supera un umbral predefinido, se puede realizar un rebalanceo local.
- **Rotaciones**: Aplicar rotaciones simples o dobles (similares a las utilizadas en los árboles AVL) para rebalancear el árbol. Las rotaciones reorganizan los nodos de manera que se minimiza la altura del subárbol desbalanceado.

1. **Rebalanceo de Subárboles**:
    - **Selección de Subárboles Desbalanceados**: Identificar y seleccionar subárboles que se hayan desbalanceado más allá de un límite específico. Esto se puede hacer mediante una revisión periódica del árbol o mediante la acumulación de estadísticas durante las operaciones normales.
    - **Reconstrucción Local de Subárboles**: Reconstruir subárboles desbalanceados utilizando técnicas de selección de mediana para garantizar que el subárbol reconstruido esté balanceado.
2. **Algoritmos de Balanceo Incremental**:
    - **Balanceo con Árboles Auxiliares**: Mantener árboles auxiliares temporales durante inserciones o eliminaciones intensivas y fusionarlos con el árbol principal periódicamente puede ayudar a mantener el equilibrio sin la necesidad de reconstrucciones completas frecuentes.
    - **Ajuste de Umbrales de Balance**: Ajustar dinámicamente los umbrales utilizados para desencadenar operaciones de rebalanceo basados en la frecuencia de operaciones y la tasa de cambio en los datos.

Ejemplo Práctico de Rebalanceo Dinámico

Para ilustrar el rebalanceo dinámico, consideremos un escenario donde un KD-Tree se usa en una aplicación de navegación robótica que recibe datos de sensores en tiempo real. Aquí, es esencial mantener el árbol balanceado para garantizar respuestas rápidas a las consultas de vecinos cercanos (k-NN) y de ruta más corta.

1. **Inserción y Verificación**:
    - Después de cada inserción de un nuevo punto de datos (por ejemplo, una nueva posición detectada por el sensor), se verifica la altura del subárbol afectado.
    - Si la altura del subárbol izquierdo y derecho de cualquier nodo difiere en más de un valor umbral (por ejemplo, 2 niveles), se activa el rebalanceo.
2. **Aplicación de Rotaciones**:

    Si se detecta un desbalance, se aplica una rotación simple o doble para rebalancear el subárbol. Por ejemplo, si el subárbol izquierdo es más alto, se puede aplicar una rotación derecha al nodo desbalanceado.
3. **Reconstrucción de Subárboles**:

    Si se detecta que un subárbol completo se ha desbalanceado (por ejemplo, después de múltiples inserciones o eliminaciones en una región específica), se selecciona el subárbol y se reconstruye utilizando la técnica de selección de mediana.

8. Comparación con Otras Estructuras de Datos

Para entender mejor las fortalezas y debilidades de los KD-Trees, es útil compararlos con otras estructuras de datos comúnmente utilizadas para la gestión de datos espaciales: Quad-Trees, R-Trees y Hashing Espacial.

8.1 KD-Trees

Ventajas:

- **Eficiencia en Búsquedas Multidimensionales**: Los KD-Trees son eficientes para realizar búsquedas de vecinos más cercanos y consultas de rango en espacios de baja a mediana dimensionalidad.
- **Flexibilidad**: Pueden manejar datos en cualquier número de dimensiones, siendo útiles en aplicaciones como el machine learning y la robótica.
- **Inserción y Eliminación Dinámica**: Permiten la inserción y eliminación de puntos de manera dinámica, aunque puede requerir técnicas de rebalanceo para mantener la eficiencia.

Desventajas:

- **Maldición de la Dimensionalidad**: Su eficiencia disminuye significativamente en espacios de alta dimensionalidad.
- **Balanceo Complejo**: Mantener el árbol balanceado puede ser complicado y costoso en términos de tiempo de computación.
- **Complejidad de Implementación**: Requieren un esfuerzo considerable para implementar y optimizar correctamente.

8.2 Quad-Trees

Ventajas:

- **Simplicidad**: Son más simples de implementar en comparación con los KD-Trees.
- **Eficiencia en Espacios 2D**: Excelentes para manejar datos en espacios bidimensionales, como imágenes y mapas.

- **Eficiente en Consultas de Área**: Las consultas que requieren seleccionar áreas específicas (rectángulos o cuadrados) son rápidas y eficientes.

Desventajas:

- **Escalabilidad Limitada**: Su eficiencia disminuye en espacios de más de dos dimensiones.
- **Desbalanceo**: Pueden volverse desbalanceados rápidamente con inserciones desiguales, lo que afecta la eficiencia de las operaciones.
- **Fragmentación del Espacio**: La subdivisión del espacio puede llevar a una gran cantidad de nodos vacíos o casi vacíos.

8.3 R-Trees

Ventajas:

- **Eficiencia en Consultas Espaciales**: Son muy eficientes para consultas de rango y búsquedas de vecinos más cercanos en espacios multidimensionales.
- **Balanceo Automático**: Los R-Trees mantienen automáticamente un balance del árbol durante inserciones y eliminaciones.
- **Aplicaciones Prácticas**: Muy utilizados en bases de datos espaciales y sistemas de información geográfica (GIS).

Desventajas:

- **Complejidad de Implementación**: Son más complejos de implementar en comparación con los Quad-Trees y KD-Trees.
- **Rendimiento Variabl**e: El rendimiento puede variar dependiendo de cómo se manejan las operaciones de división y fusión de nodos.

8.4 Hashing Espacial

Ventajas:

- **Eficiencia de Búsqueda:** Ofrece una búsqueda de tiempo constante O(1)O(1)O(1) en la práctica, especialmente en conjuntos de datos grandes y distribuidos uniformemente.
- **Simplicidad:** Es fácil de implementar y muy eficiente para operaciones de búsqueda exacta.
- **Escalabilidad:** Se escala bien con el aumento de datos, ya que el rendimiento de las operaciones de hash es constante.

Desventajas:

- **Consultas de Rango Ineficientes:** No es eficiente para consultas de rango o búsquedas de vecinos más cercanos, ya que requiere escanear múltiples celdas de hash.
- **Fragmentación del Espacio:** Puede sufrir problemas de fragmentación y colisiones, lo que requiere estrategias adicionales para manejarlas.
- **Limitado a Búsquedas Exactas:** No es adecuado para aplicaciones que requieren búsqueda aproximada o consultas de vecinos más cercanos.

8.5 Conclusión

Cada una de estas estructuras de datos tiene sus propias ventajas y desventajas, haciéndolas más o menos adecuadas para diferentes tipos de aplicaciones:

- KD-Trees son ideales para aplicaciones que requieren búsquedas eficientes en espacios multidimensionales de baja a media dimensionalidad, como machine learning y robótica.
- Quad-Trees son óptimos para manejar datos espaciales en dos dimensiones, como en procesamiento de imágenes y mapas.
- R-Trees son altamente eficientes para consultas espaciales y son ampliamente utilizados en bases de datos geoespaciales y sistemas GIS.
- Hashing Espacial es excelente para búsquedas exactas y se escala bien con grandes volúmenes de datos, aunque no es adecuado para consultas de rango o vecinos más cercanos.

La elección de la estructura de datos depende de las necesidades específicas de la aplicación, el tipo de datos que se maneja y las operaciones que se deben realizar con mayor frecuencia.

9. Consideraciones de implementación

9.1 Complejidad Temporal y Espacial

Inserción y Búsqueda: La operación principal en un KD-Tree es la inserción y búsqueda de puntos. En promedio, tanto la inserción como la búsqueda tienen una complejidad de O(log⁡n)O(\log n)O(logn), donde nnn es el número de puntos en el árbol. Sin embargo, en el peor caso, especialmente si el árbol no está balanceado adecuadamente, la complejidad puede llegar a ser O(n)O(n)O(n). Mantener el árbol balanceado mediante técnicas como rotaciones y ajustes es crucial para asegurar que las operaciones se realicen en tiempo logarítmico.

Espacio: El espacio requerido por un KD-Tree depende directamente del número de puntos almacenados y de la estructura del árbol. Cada nodo en el árbol KD almacena un

punto y referencias a los subárboles izquierdo y derecho. En dimensiones más altas, cada punto tiene más coordenadas, lo que aumenta el espacio requerido para almacenar el árbol. Además, si se utiliza un KD-Tree para almacenar datos adicionales en cada nodo (como información adicional sobre los puntos), el espacio ocupado puede incrementarse significativamente.

9.2 Manejo de Datos de Alta Dimensionalidad

Maldición de la Dimensionalidad: Uno de los desafíos principales de los KD-Trees es la "maldición de la dimensionalidad". En espacios de alta dimensionalidad, los puntos tienden a estar más dispersos y las distancias entre puntos se vuelven más similares entre sí. Esto provoca que las estructuras basadas en distancia, como los KD-Trees, pierdan eficiencia ya que la diferenciación entre puntos cercanos y lejanos se vuelve menos significativa. Como resultado, el rendimiento de las operaciones de búsqueda y consulta puede deteriorarse considerablemente.

Selección de Dimensiones: La selección adecuada de la dimensión para dividir en cada nivel del árbol KD puede tener un impacto significativo en su rendimiento. Estrategias comunes incluyen alternar entre dimensiones (por ejemplo, usando la dimensión kkk en el nivel k%Dk \% Dk%D, donde DDD es el número total de dimensiones) o elegir la dimensión con mayor varianza entre los puntos del nodo actual. La elección de una estrategia de selección de dimensiones efectiva puede ayudar a mitigar los efectos adversos de la alta dimensionalidad y mejorar la eficiencia de las operaciones del árbol.

9.3 Selección de Dimensiones

Criterios de Selección: La forma en que se selecciona la dimensión para dividir en cada nivel del árbol KD es crucial para optimizar su rendimiento. La estrategia de alternancia simple es fácil de implementar y puede funcionar bien en muchos casos, pero en situaciones donde las distribuciones de datos son desiguales o las dimensiones no están

correlacionadas, estrategias más sofisticadas como la selección basada en la varianza pueden ser preferibles. La varianza puede indicar qué dimensiones son más informativas para dividir los datos de manera que se optimice la estructura del árbol.

9.4 Escalabilidad y Eficiencia

Balanceo del Árbol: Mantener un KD-Tree balanceado es fundamental para garantizar la eficiencia en las operaciones. Dado que las inserciones y eliminaciones pueden desequilibrar el árbol, es importante implementar técnicas de rebalanceo como rotaciones o reorganizaciones de nodos para mantener la altura del árbol en un nivel logarítmico respecto al número de nodos nnn. Sin un balance adecuado, las operaciones pueden degradarse a O(n)O(n)O(n), perdiendo así la ventaja de la estructura logarítmica.

Optimización del Espacio: En aplicaciones donde el espacio es un recurso crítico, como en sistemas embebidos o bases de datos con grandes conjuntos de datos espaciales, se pueden aplicar técnicas de optimización del espacio en los KD-Trees. Estas técnicas pueden incluir poda de nodos redundantes o innecesarios, compresión de estructuras de datos adicionales almacenadas en cada nodo, o el uso de variantes de KD-Trees que minimicen el espacio utilizado sin comprometer significativamente la eficiencia de las consultas.

9.5 Optimizaciones Específicas de Aplicación

Personalización para Aplicaciones Específicas: Los KD-Trees pueden adaptarse y optimizarse para diversas aplicaciones y escenarios específicos:

En aplicaciones de aprendizaje automático, como la búsqueda de vecinos más cercanos en algoritmos de clasificación, se pueden implementar variaciones del KD-Tree que prioricen eficiencia en operaciones específicas.

En entornos de robótica y sistemas de información geográfica (GIS), los KD-Trees pueden personalizarse para manejar eficientemente datos de sensores multidimensionales o consultas geoespaciales complejas.

Las optimizaciones pueden incluir ajustes en la estrategia de selección de dimensiones, el manejo de datos desbalanceados o la implementación de técnicas de poda adaptativas que mejoran el rendimiento específico de la aplicación sin comprometer la estructura general del árbol.

10. Casos de estudio y ejemplos prácticos

10.1. Reconocimiento de Imágenes

En el campo del reconocimiento de imágenes, los KD-Trees juegan un papel crucial al permitir una búsqueda eficiente de vecinos más cercanos en espacios de características de imágenes. Imagina un sistema de búsqueda de imágenes basado en contenido que necesita comparar una imagen de consulta con una base de datos de millones de imágenes. Utilizando un KD-Tree para indexar características extraídas de cada imagen (como descriptores de textura, color o forma), podemos reducir drásticamente el tiempo necesario para encontrar imágenes similares. Esto se logra dividiendo recursivamente el espacio de características en cada dimensión y almacenando los puntos de datos en nodos hoja del árbol. La búsqueda se realiza rápidamente navegando a través del árbol para encontrar vecinos cercanos en el espacio de características, lo que permite respuestas rápidas incluso en grandes conjuntos de datos.

10.2 Búsqueda Espacial

En aplicaciones de sistemas de información geográfica (GIS) o navegación, los KD-Trees son esenciales para manejar y consultar datos geoespaciales eficientemente. Por

ejemplo, en un sistema de navegación GPS que necesita buscar los puntos de interés (como restaurantes, gasolineras, etc.) más cercanos a la ubicación actual del usuario, un KD-Tree puede indexar rápidamente estos puntos utilizando coordenadas geográficas. La estructura del árbol permite realizar consultas de vecinos más cercanos de manera eficiente, facilitando la visualización de resultados relevantes en mapas interactivos y sistemas de navegación.

10.3 Clasificación y Regresión

En el contexto de machine learning, los KD-Trees son útiles para optimizar algoritmos de clasificación y regresión que dependen de la búsqueda eficiente de vecinos más cercanos en el espacio de características. Por ejemplo, en un problema de reconocimiento de dígitos escritos a mano, un KD-Tree puede acelerar la clasificación de nuevos dígitos comparándolos rápidamente con dígitos almacenados en una base de datos mediante características como la intensidad de píxeles. Esto mejora la precisión y la velocidad de los modelos predictivos, permitiendo una clasificación rápida incluso en grandes conjuntos de datos de entrenamiento**.**

10.4 Procesamiento de Nubes de Puntos

En aplicaciones de visión por computadora y gráficos 3D, los KD-Trees facilitan el manejo y procesamiento eficiente de nubes de puntos, utilizadas en la reconstrucción de modelos 3D, simulaciones físicas y realidad aumentada. Por ejemplo, en la representación de objetos tridimensionales en un entorno virtual, un KD-Tree puede indexar rápidamente los puntos que forman los modelos 3D, permitiendo operaciones como la detección de colisiones o la interacción precisa entre objetos virtuales. Esto optimiza la experiencia del usuario al garantizar una respuesta en tiempo real y una representación precisa del entorno virtual.

10.5 Búsqueda de Colisiones en Simulaciones Físicas

En el desarrollo de videojuegos y simulaciones físicas, los KD-Trees son esenciales para detectar colisiones entre objetos virtuales de manera eficiente. Por ejemplo, en un simulador de carreras donde múltiples vehículos interactúan entre sí y con el entorno, un KD-Tree puede indexar rápidamente las posiciones y dimensiones de cada vehículo. Esto permite detectar de manera eficiente si dos vehículos están en riesgo de colisión, optimizando la simulación y garantizando una experiencia de juego fluida y realista.

11. Herramientas y librerías para kd trees

Los KD-Trees son implementados en diversas bibliotecas y herramientas en diferentes lenguajes de programación. Aquí te presento algunas opciones comunes y una comparación de rendimiento entre ellas en los lenguajes populares como C++, Python y Java:

11.1 Implementaciones en Lenguajes Populares

1. **C++**
    - **CGAL (Computational Geometry Algorithms Library):** CGAL proporciona una implementación eficiente de KD-Trees y otras estructuras de datos geométricas. Es conocida por su robustez y velocidad en algoritmos geométricos.
    - **FLANN (Fast Library for Approximate Nearest Neighbors):** Aunque FLANN está más enfocada en búsqueda de vecinos cercanos aproximados, también incluye una implementación eficiente de KD-Trees para búsqueda exacta de vecinos más cercanos.
2. **Python**

- **scikit-learn**: Esta biblioteca de machine learning en Python incluye una implementación de KD-Trees como parte de su módulo neighbors. Es fácil de usar y está optimizado para trabajar con grandes conjuntos de datos.
- **SciPy**: La biblioteca SciPy también ofrece una implementación de KD-Trees en su módulo spatial. Es útil para aplicaciones científicas y de análisis de datos que requieren operaciones espaciales eficientes.

3. **Java**
    - **JTS (Java Topology Suite)**: JTS es una biblioteca Java para operaciones geométricas que incluye soporte para estructuras de datos espaciales como KD-Trees. Es utilizada en aplicaciones de GIS y análisis geoespacial.
    - **ELKI (Environment for Developing KDD-Applications Supported by Index-Structures):** ELKI proporciona implementaciones eficientes de KD-Trees y otras estructuras de datos para minería de datos y aplicaciones de descubrimiento de conocimiento.

11.2 Comparación de Rendimiento

La elección de una implementación de KD-Tree puede depender del lenguaje de programación utilizado y de los requisitos específicos de rendimiento y escalabilidad de la aplicación. Aquí hay algunos puntos a considerar al comparar el rendimiento:

- Eficiencia Temporal: La velocidad de inserción, búsqueda y eliminación de datos en el KD-Tree.
- Complejidad Espacial: La cantidad de memoria utilizada por la estructura de datos, especialmente importante en grandes conjuntos de datos.

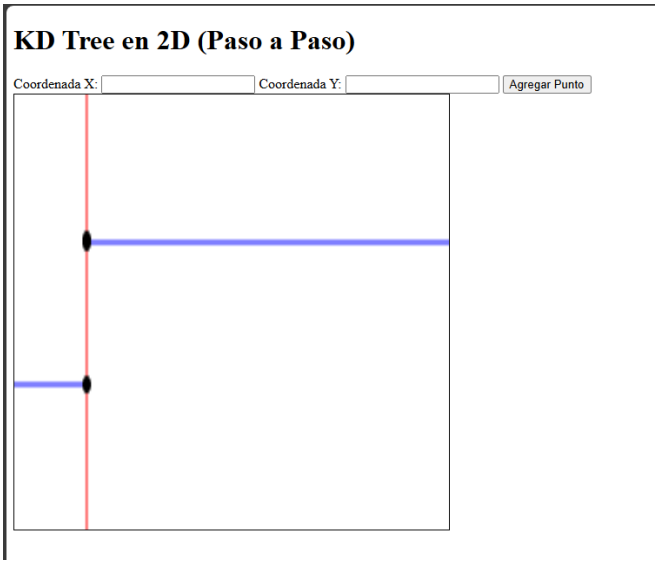
- Escalabilidad: Cómo maneja la estructura el aumento en el número de dimensiones o el tamaño de los conjuntos de datos.
- Facilidad de Uso: La claridad de la API y la documentación disponible para la biblioteca o herramienta.

12. Código

```
<!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 (Paso a Paso)</title>
<style>
    #canvas {
        width: 500px;
        height: 500px;
        border: 1px solid black;
    }
</style>
</head>
<body>
<h1>KD Tree en 2D (Paso a Paso)</h1>
<div>
    <label for="x">Coordenada X:</label>
    <input type="number" id="x" min="0">
    <label for="y">Coordenada Y:</label>
    <input type="number" id="y" min="0">
    <button onclick="addPoint()">Agregar Punto</button>
</div>
<canvas id="canvas"></canvas>

<script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');
    let points = [];
    let kdTree = null;

    // Clase para representar un punto
    class Point {
        constructor(x, y) {
            this.x = x;
            this.y = y;
        }
    }

    // Clase para representar un nodo del árbol KD
    class KDNode {
        constructor(point, axis, left, right) {
            this.point = point;
            this.axis = axis; // 0 para X, 1 para Y
            this.left = left;
            this.right = right;
        }
    }
```

```
    // Función para construir un árbol KD a partir de una lista de
puntos
    function buildKDTree(points, depth = 0) {
        if (points.length === 0) {
            return null;
        }

        const axis = depth % 2; // Alternar entre X y Y

        // Ordenar puntos según el eje actual
        points.sort((a, b) => (axis === 0 ? a.x - b.x : a.y - b.y));

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

        const leftPoints = points.slice(0, medianIndex);
        const rightPoints = points.slice(medianIndex + 1);

        return new KDNode(
            median,
            axis,
            buildKDTree(leftPoints, depth + 1),
            buildKDTree(rightPoints, depth + 1)
        );
    }

    // Función para dibujar un árbol KD
    function drawKDTree(node, minX, maxX, minY, maxY) {
        if (!node) {
            return;
        }

        // Dibujar la división vertical u horizontal según el eje del
nodo
        if (node.axis === 0) { // Eje X
            ctx.beginPath();
            ctx.moveTo(node.point.x, minY);
            ctx.lineTo(node.point.x, maxY);
            ctx.strokeStyle = 'red';
            ctx.stroke();
            ctx.closePath();

            drawKDTree(node.left, minX, node.point.x, minY, maxY);
            drawKDTree(node.right, node.point.x, maxX, minY, maxY);
        } else { // Eje Y
            ctx.beginPath();
            ctx.moveTo(minX, node.point.y);
            ctx.lineTo(maxX, node.point.y);
            ctx.strokeStyle = 'blue';
            ctx.stroke();
            ctx.closePath();

            drawKDTree(node.left, minX, maxX, minY, node.point.y);
            drawKDTree(node.right, minX, maxX, node.point.y, maxY);
        }
    }

    // Función para dibujar los puntos
    function drawPoints(points) {
        points.forEach(point => {
```

```
            ctx.beginPath();
            ctx.arc(point.x, point.y, 3, 0, Math.PI * 2);
            ctx.fillStyle = 'black';
            ctx.fill();
            ctx.closePath();
        });
    }

    // Función para agregar un punto con las coordenadas ingresadas
    function addPoint() {
        const xInput = document.getElementById('x');
        const yInput = document.getElementById('y');
        const x = parseInt(xInput.value);
        const y = parseInt(yInput.value);

        if (!isNaN(x) && !isNaN(y)) {
            const point = new Point(x, y);
            points.push(point);
            xInput.value = '';
            yInput.value = '';

            kdTree = buildKDTree(points);
            draw();
        } else {
            alert('Por favor ingresa coordenadas válidas.');
        }
    }

    // Función principal para dibujar
    function draw() {
        ctx.clearRect(0, 0, canvas.width, canvas.height);
        if (kdTree) {
            drawKDTree(kdTree, 0, canvas.width, 0, canvas.height);
        }
        drawPoints(points);
    }

    // Dibujar al cargar la página
    draw();
</script>
</body>

</html>
```



Para la simulación se utilizaron las siguientes coordenadas:

(X, Y)

(50,100)

(50,50)

(50,51)

12.1 Explicación del código por partes

12.1.1 Definición de elementos HTML:

```
<!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 (Paso a Paso)</title>
<style>
    #canvas {
        width: 500px;
        height: 500px;
        border: 1px solid black;
    }
</style>
</head>
<body>
<h1>KD Tree en 2D (Paso a Paso)</h1>
<div>
    <label for="x">Coordenada X:</label>
    <input type="number" id="x" min="0">
    <label for="y">Coordenada Y:</label>
    <input type="number" id="y" min="0">
    <button onclick="addPoint()">Agregar Punto</button>
</div>
<canvas id="canvas"></canvas>
```

Aquí se define la estructura básica de la página HTML, se crea un formulario simple para ingresar las coordenadas X e Y de un punto. Hay un botón "Agregar Punto" que llama a la función addPoint() cuando se hace clic, se utiliza un elemento <canvas> para dibujar los puntos y las divisiones del árbol KD.

12.1.2 Funciones JavaScript:

```
<script>
    const canvas = document.getElementById('canvas');
    const ctx = canvas.getContext('2d');
    let points = [];
    let kdTree = null;
```

Se obtiene una referencia al elemento <canvas> y se obtiene su contexto de dibujo 2D, se declaran dos variables globales: points para almacenar los puntos ingresados y kdTree para almacenar el árbol KD.

```
// Clase para representar un punto
    class Point {
        constructor(x, y) {
            this.x = x;
            this.y = y;
        }
    }
```

Se define una clase Point para representar un punto en 2D con coordenadas X e Y.

```
    // Clase para representar un nodo del árbol KD
    class KDNode {
        constructor(point, axis, left, right) {
            this.point = point;
            this.axis = axis; // 0 para X, 1 para Y
            this.left = left;
            this.right = right;
        }
    }
```

Se define una clase KDNode para representar un nodo en el árbol KD. Cada nodo tiene un punto asociado, un eje de división (0 para X, 1 para Y), y referencias a los nodos izquierdo y derecho.

```
    // Función para construir un árbol KD a partir de una lista de
puntos
    function buildKDTree(points, depth = 0) {
        if (points.length === 0) {
            return null;
        }

        const axis = depth % 2; // Alternar entre X y Y

        points.sort((a, b) => (axis === 0 ? a.x - b.x : a.y - b.y));

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

        const leftPoints = points.slice(0, medianIndex);
        const rightPoints = points.slice(medianIndex + 1);

        return new KDNode(
            median,
            axis,
            buildKDTree(leftPoints, depth + 1),
            buildKDTree(rightPoints, depth + 1)
        );
    }
```

Se define la función buildKDTree para construir un árbol KD recursivamente a partir de una lista de puntos. La función calcula el eje de división alternando entre X e Y en cada nivel del árbol. Los puntos se ordenan según el eje actual antes de dividirlos, la mediana de los puntos ordenados se elige como punto de división, y se construyen los subárboles izquierdo y derecho con los puntos restantes.

```
// Función para dibujar un árbol KD
function drawKDTree(node, minX, maxX, minY, maxY) {
    if (!node) {
        return;
    }

    if (node.axis === 0) { // Eje X
        ctx.beginPath();
        ctx.moveTo(node.point.x, minY);
        ctx.lineTo(node.point.x, maxY);
        ctx.strokeStyle = 'red';
        ctx.stroke();
        ctx.closePath();

        drawKDTree(node.left, minX, node.point.x, minY, maxY);
        drawKDTree(node.right, node.point.x, maxX, minY, maxY);
    } else { // Eje Y
        ctx.beginPath();
        ctx.moveTo(minX, node.point.y);
        ctx.lineTo(maxX, node.point.y);
        ctx.strokeStyle = 'blue';
        ctx.stroke();
        ctx.closePath();

        drawKDTree(node.left, minX, maxX, minY, node.point.y);
        drawKDTree(node.right, minX, maxX, node.point.y, maxY);
    }
}
```

Se define la función drawKDTree para dibujar un árbol KD en el lienzo. La función utiliza el algoritmo de recorrido preorden para dibujar los nodos del árbol. Se dibuja una línea vertical u horizontal para representar la división en cada nodo, dependiendo del eje de división

```
// Función para dibujar los puntos
function drawPoints(points) {
    points.forEach(point => {
        ctx.beginPath();
        ctx.arc(point.x, point.y, 3, 0, Math.PI * 2);
        ctx.fillStyle = 'black';
        ctx.fill();
        ctx.closePath();
    });
}
```

Se define la función drawPoints para dibujar los puntos ingresados en el lienzo. La función dibuja un círculo en cada punto con un radio de 3 píxeles.

```
// Función para agregar un punto con las coordenadas ingresadas
function addPoint() {
    const xInput = document.getElementById('x');
    const yInput = document.getElementById('y');
    const x = parseInt(xInput.value);
    const y = parseInt(yInput.value);

    if (!isNaN(x) && !isNaN(y)) {
        const point = new Point(x, y);
        points.push(point);
        xInput.value = '';
        yInput.value = '';

        kdTree = buildKDTree(points);
        draw();
    } else {
        alert('Por favor ingresa coordenadas válidas.');
    }
}
```

Se define la función addPoint para agregar un nuevo punto a la lista de puntos, se obtienen las coordenadas X e Y del formulario y se crea un nuevo objeto Point, se valida que las coordenadas ingresadas sean números válidos antes de agregar el punto, después de agregar el punto, se reconstruye el árbol KD y se actualiza la visualización en él.

13. Conclusiones

En conclusión, los árboles KD (KD-Trees) representan una hx|erramienta poderosa y eficiente para la organización y búsqueda de datos multidimensionales. Su capacidad para dividir recursivamente el espacio de datos y realizar búsquedas eficientes del vecino más cercano los hace indispensables en una variedad de campos, desde la visión por computadora hasta la minería de datos.

Hemos explorado los conceptos fundamentales de los KD-Trees, incluyendo su estructura, algoritmos de construcción y búsqueda, así como sus aplicaciones prácticas en diversos campos. Queda claro que los KD-Trees ofrecen una solución elegante y efectiva para problemas que involucran datos multidimensionales.

Sin embargo, es importante destacar que aún existen desafíos y áreas de investigación abiertas en el campo de los KD-Trees. Por ejemplo, la optimización de algoritmos para espacios de alta dimensionalidad y la adaptación de KD-Trees a entornos de big data son áreas que merecen una atención continua.

Los KD-Trees representan una contribución significativa al campo de la organización de datos multidimensionales y ofrecen un gran potencial para futuras investigaciones y aplicaciones innovadoras.

14. Referencias

-De Berg, M., Van Kreveld, M., Overmars, M., & Schwarzkopf, O. (2008). Computational Geometry: Algorithms and Applications. Springer.

-Preparata, F. P., & Shamos, M. I. (1985). Computational Geometry: An Introduction. Springer.

-O'Rourke, J. (1998). Computational Geometry in C. Cambridge University Press.