UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE
INGENIERÍA MECÁNICA ELÉCTRICA, ELECTRÓNICA Y SISTEMAS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

TRABAJO ENCARGADO

KD-TREE

CURSO:

ESTRUCTURA DE DATOS AVANZADAS

DOCENTE:

ING. COLLANQUI MARTINEZ FREDY

PRESENTADO POR:

BELTRAN EDWIN MAMANI MAMANI

PUNO- PERU

2024

INDICE

DEFINICION DE KD-TREES

Un KD-Tree (K-dimensional tree) es una estructura de datos en forma de árbol utilizada para organizar puntos en un espacio K-dimensional. Se emplea en aplicaciones que requieren búsquedas eficientes en espacios de alta dimenionalidad, como la búsqueda de vecinos más cercanos, la búsqueda de rango y la clasificación de patrones.

Los KD-Trees son una generalización de los árboles binarios de búsqueda que manejan claves multidimensionales, pero presentan complicaciones en los algoritmos de actualización. Para simplificar estos algoritmos, se introduce el discriminante, creando los relaxed KD-Trees, los cuales utilizan aleatorización para mantener la eficiencia y evitar la degeneración del árbol con sucesivas inserciones y borrados.

ESTRUCTURA DE UN KD-TREE

Un KD-Tree es una estructura de datos que organiza puntos en un espacio k-dimensional. Su objetivo principal es permitir búsquedas rápidas, tales como encontrar el vecino más cercano o realizar consultas de rango.

La estructura del KD-Tree está compuesta por nodos que dividen recursivamente el espacio en semi-espacios usando diferentes dimensiones como discriminadores.

Nodo

- **Datos**: Cada nodo en un KD-Tree representa un punto en el espacio k-dimensional. Este punto es un vector de k dimensiones que define una posición en el espacio.

Discriminador

El discriminador es el eje o dimensión que se utiliza para dividir el espacio en el nodo actual. Este índice rota entre las dimensiones (de 0 a k−1k-1k−1) en cada nivel del árbol. Por ejemplo, en un espacio 3D, los discriminadores rotarían entre las dimensiones xxx, yyy, y zzz en niveles consecutivos del árbol. La rotación asegura que cada dimensión sea utilizada de manera equitativa para particionar el espacio.

- **Valor del Discriminador**: Es el valor del punto en la dimensión del discriminador. Este valor se utiliza para decidir cómo se dividen los puntos en los subárboles izquierdo y derecho. Los puntos con un valor menor que este se colocan en el subárbol izquierdo, y los puntos con un valor mayor o igual se colocan en el subárbol derecho.

Subárboles

- **Subárbol Izquierdo**: Contiene todos los puntos cuyo valor en la dimensión del discriminador es menor que el valor del discriminador del nodo actual. Este subárbol representa el semi-espacio a la izquierda del plano definido por el valor del discriminador en la dimensión actual.
- **Subárbol Derecho**: Contiene todos los puntos cuyo valor en la dimensión del discriminador es mayor o igual que el valor del discriminador del nodo actual. Este subárbol representa el semi-espacio a la derecha del plano definido por el valor del discriminador en la dimensión actual.

EJEMPLO: CONSTRUCCIÓN DE UN KD-TREE

Vamos a construir un KD-Tree con un ejemplo paso a paso utilizando un conjunto de puntos en un espacio 2D. El proceso incluye seleccionar los puntos, determinar el discriminador en cada nivel, y dividir el espacio hasta que todos los puntos estén insertados en el árbol.

Descripción del Código

1. **KDTree Class**:
   - Implementa un k-d tree que organiza los puntos en un espacio 2D.
   - El constructor divide los puntos recursivamente en nodos, alternando entre los ejes X y Y.
2. **Generación de Puntos**:

- generateRandomPoints crea un conjunto de puntos aleatorios en el lienzo para visualización.

3. **Funciones de Dibujo**:

- drawPoint dibuja un punto en el lienzo.
- drawLine dibuja una línea que representa la división de espacio en el k-d tree.

4. **Construcción y Visualización del KD-Tree**:

- buildKDTree limpia el lienzo, dibuja los puntos iniciales y construye el k-d tree.
- drawKDTree dibuja recursivamente las divisiones del k-d tree en el lienzo.

Cómo Funciona

- **Puntos Negros**: Representan los puntos aleatorios iniciales.
- **Puntos Rojos**: Representan los nodos del k-d tree en cada nivel de la división.
- **Líneas Rojas**: Representan las divisiones a lo largo del eje X.
- **Líneas Azules**: Representan las divisiones a lo largo del eje Y.

```
1                   < script >
2                       clase KDTree {
3                           constructor(puntos, profundidad =  0 ) {
4                               if (puntos.longitud ===  0 ) {
5                                   this .node =  null ;
6                                   este .left =  nulo ;
7                                   este .right =  nulo ;
8                                   devolver ;
9                               }
10
11                              eje constante = profundidad %  2 ; // Alternar entre eje
12          x (0) y eje y (1)
```

```
13                          points.sort((a, b) => a[eje] - b[eje]); // Ordenar puntos
14          según el eje actual
15                          const mediana = Math .floor(points.length / 2 );
dieciséis
17                          este .nodo = puntos[mediana];
18                           this .left = new KDTree(points.slice( 0 , mediana),
19          profundidad + 1 );
20                           this .right = new KDTree(points.slice(mediana + 1 ),
21          profundidad + 1 );
22                      }
23                  }
24
25                  const lienzo = documento .getElementById ( 'lienzo' );
26                   const ctx = lienzo.getContext( '2d' );
27
28                  puntos constantes = generarPuntosRandom( 10 ,
29          lienzo.ancho, lienzo.alto);
30
31                  función generarPuntosRandom(n, ancho, alto) {
32                      puntos constantes = [];
33                      para ( sea i = 0 ; i < n; i ++ ) {
34                          puntos.push([ Math .random() * ancho, Math
35          .random() * alto]);
36                      }
37                      puntos de retorno ;
```

```
38                      }
39
40                      función dibujarPunto(punto, color = 'negro' ) {
41                          ctx.beginPath();
42                          ctx.arc(punto[ 0 ], punto[ 1 ], 5 , 0 , 2 * Matemáticas .PI);
43                          ctx.fillStyle = color;
44                          ctx.fill();
45                      }
46
47                      función dibujarLínea(desde, hasta, color = 'azul' ) {
48                          ctx.beginPath();
49                          ctx.moveTo(desde[ 0 ], desde[ 1 ]);
50                          ctx.lineTo(a[ 0 ], a[ 1 ]);
51                          ctx.strokeStyle = color;
52                          ctx.stroke();
53                      }
54
55                      función drawKDTree(árbol, minX, maxX, minY, maxY,
56              profundidad = 0 ) {
57                          if ( ! tree.node) return ;
58
59                          const [x, y] = árbol.nodo;
60                          drawPoint([x, y], 'rojo' );
61
62                          if (profundidad % 2 === 0 ) { // Dividir según el eje X
```

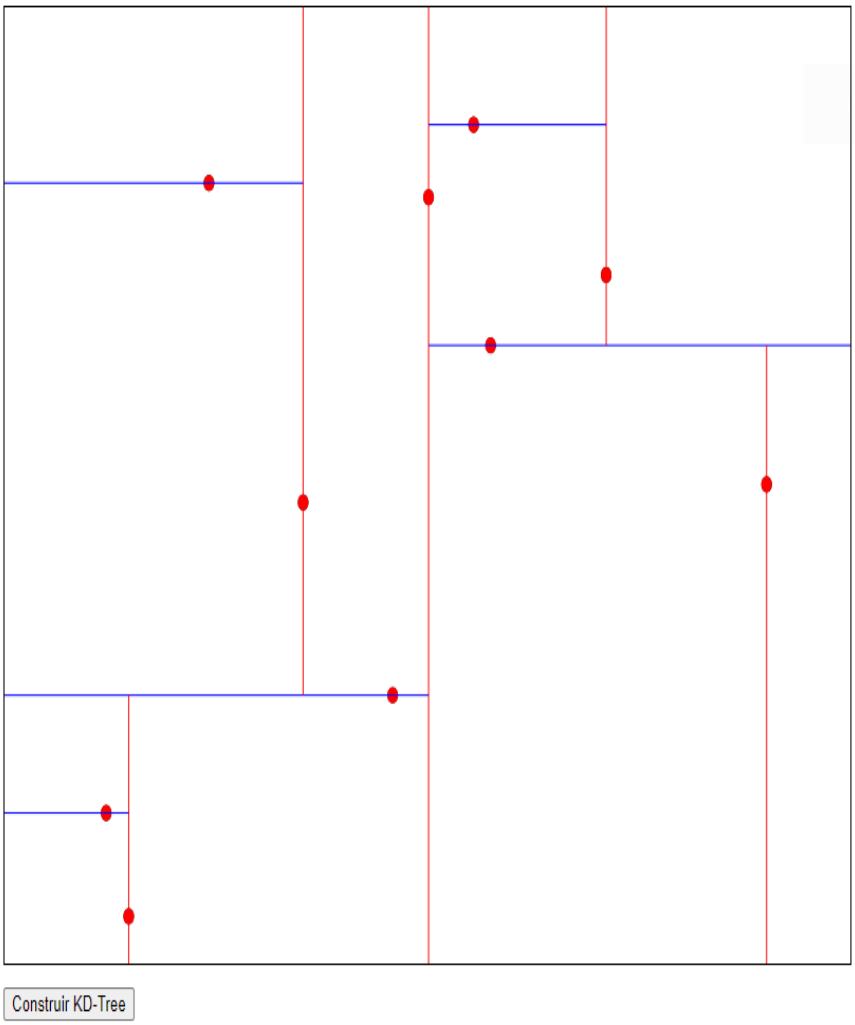
```
63                          drawLine([x, minY], [x, maxY], 'red' );
64                          drawKDTree(tree.left,  minX,  x,  minY,  maxY,
sesenta     y profundidad +  1 );
cinco                       drawKDTree(árbol.derecha,  x,  maxX,  minY,  maxY,
66              profundidad +  1 );
67                        } else { // Dividir según el eje Y
68                          drawLine([minX, y], [maxX, y], 'blue' );
69                          drawKDTree(tree.left,  minX,  maxX,  minY,  y,
70              profundidad +  1 );
71                          drawKDTree(tree.right,  minX,  maxX,  y,  maxY,
72              profundidad +  1 );
73                        }
74                    }

                    función construirKDTree() {
                      ctx.clearRect( 0 , 0 , lienzo.ancho, lienzo.alto); // Limpiar
                    lienzo

                      puntos.forEach(punto => drawPoint(punto, 'negro' )); //
                    dibujar puntos iniciales

                      const kdTree =  nuevo KDTree(puntos);
                      drawKDTree(kdTree, 0 , lienzo.ancho, 0 , lienzo.alto); //
                    Dibujar KD-Tree
                    }
```

< /guión>

Ejecución

1. **Paso 1**: Guarda el código en un archivo index.html.
2. **Paso 2**: Abre index.html en tu navegador.
3. **Paso 3**: Haz clic en el botón "Construir KD-Tree" para ver la construcción paso a paso del árbol y sus divisiones en el lienzo.

Este código proporciona una visualización clara de cómo se construye un k-d tree y cómo se divide el espacio 2D en nodos utilizando los ejes X y Y.



ALGORITMOS ASOCIADOS A KD-TREES

Estos algoritmos son fundamentales para realizar operaciones eficientes sobre datos espaciales en espacios multidimensionales. La estructura y los algoritmos de KD-Trees se utilizan en aplicaciones que van desde la búsqueda de vecinos más cercanos en bases de datos espaciales hasta la optimización en algoritmos de búsqueda y recuperación de información.

Inserción en un KD-Tree:

- **Proceso:** Se realiza una búsqueda recursiva descendiendo por el árbol desde la raíz hasta una hoja.
- **Decisión:** En cada nivel del árbol, se compara el nuevo punto con el punto actual del nodo utilizando el discriminador actual.
- **Inserción:** Se decide si el nuevo punto debe ir al subárbol izquierdo o derecho del nodo actual según su valor en el discriminador.
- **Finalización:** La inserción se completa cuando se alcanza un nodo hoja adecuado para el nuevo punto.

Búsqueda de Vecinos Más Cercanos (k-NN) en un KD-Tree:

- **Algoritmo:** Se utiliza una búsqueda recursiva para encontrar los k vecinos más cercanos a un punto objetivo.
- **Comparación:** En cada nodo, se compara el punto objetivo con el punto del nodo utilizando el discriminador actual.
- **Actualización:** Se mantiene una lista de los k puntos más cercanos encontrados hasta el momento.
- **Exploración:** El algoritmo explora primero el subárbol que podría contener puntos más cercanos al objetivo basándose en el valor del discriminador.

- **Optimización:** Se minimiza la exploración mediante el cálculo de distancias y la actualización de la lista de vecinos más cercanos según sea necesario.

EJEMPLO BÚSQUEDA DE VECINOS MÁS CERCANOS

Aquí tienes un ejemplo simple de cómo utilizar un KD-Tree para la búsqueda del vecino más cercano en JavaScript dentro de HTML. En este ejemplo, se genera un conjunto de puntos aleatorios en un espacio 2D y se implementa la búsqueda del vecino más cercano utilizando un KD-Tree.

```
1
                < script >
2
                    clase KDTree {
3
                        constructor(puntos, profundidad =  0 ) {
4
                            if (puntos.longitud ===  0 ) {
5
                                this .node =  null ;
6
                                este .left =  nulo ;
7
                                este .right =  nulo ;
8
                                devolver ;
9
                            }
10
11
                            eje constante = profundidad %  2 ;
12
                            puntos.sort((a, b) => a[eje] - b[eje]);
13
                            mediana constante =  Math .floor(puntos.longitud /  2 );
14
15
                            este .nodo = puntos[mediana];
                            this .left =  new KDTree(points.slice( 0 , mediana),
dieciséis
                profundidad +  1 );
17
```

```
18                      this .right =  new KDTree(points.slice(mediana +  1 ),
19      profundidad +  1 );
20                  }
21
22                  vecino más cercano (punto, profundidad =  0 , mejor =  nulo
23      , mejorDist =  infinito ) {
24                      si ( este .nodo ===  nulo ) devuelve mejor;
25
26                      eje constante = profundidad %  2 ;
27                      const nextBranch = punto [eje] <  este .nodo [eje] ?  esta
28      .izquierda :  esta .derecha;
29                      const rama opuesta = punto [eje] <  este .nodo [eje] ?  esta
30      .derecha :  esta .izquierda;
31
32                      let más cercano = mejor;
33                      let Dist más cercana = mejorDist;
34
35                      distancia constante =  esta .distancia (punto, este .nodo);
36                      si (distancia < distanciamáscercana) {
37                          más cercano =  este .nodo;
38                          Dist más cercano = distancia;
39                      }
40
41                      más  cercano  =  nextBranch.nearestNeighbor(punto,
42      profundidad +  1 , más cercano, más cercanoDist);
```

```
43
44                          if ( Math .abs(punto[eje] - este .nodo[eje]) <
45      distanciamáscercana) {
46                              más cercano = opuestoBranch.nearestNeighbor(punto,
47      profundidad + 1 , más cercano, más cercanoDist);
48                          }
49
50                          volver más cercano;
51                      }
52
53                      distancia(punto1, punto2) {
54                          return Math .sqrt((punto1[ 0 ] - punto2[ 0 ]) ** 2 +
55      (punto1[ 1 ] - punto2[ 1 ]) ** 2 );
56                      }
57                  }
58
59                  const lienzo = documento .getElementById ( 'lienzo' );
60                  const ctx = lienzo.getContext( '2d' );
61
62                  // Genera puntos aleatorios
63                  const points = generateRandomPoints( 50 , canvas.width,
64      canvas.height);
sesenta
y cinco             // Crea el KD-Tree
66                  const kdTree = new KDTree(points);
```

```
67
68                  // Punto de prueba para buscar el vecino más cercano
69                  const testPoint = [ 400 , 300 ];
70
71                  // Función para generar puntos aleatorios
72                  function generateRandomPoints(n, width, height) {
73                      const points = [];
74                      para ( sea i =  0 ; i < n; i ++ ) {
75                          puntos.push([ Math .random() * ancho, Math .random() *
76      alto]);
77                      }
78                      puntos de retorno ;
79                  }
80
81                  // Función para dibujar un punto en el lienzo
82                  function drawPoint(point, color =  'black' ) {
83                      ctx.beginPath();
84                      ctx.arc(punto[ 0 ], punto[ 1 ], 5 , 0 , 2  *  Matemáticas .PI);
85                      ctx.fillStyle = color;
86                      ctx.fill();
87                  }
88
89                  // Función para dibujar una línea entre dos puntos en el lienzo
90                  function drawLine(from, to, color =  'blue' ) {
91                      ctx.beginPath();
```

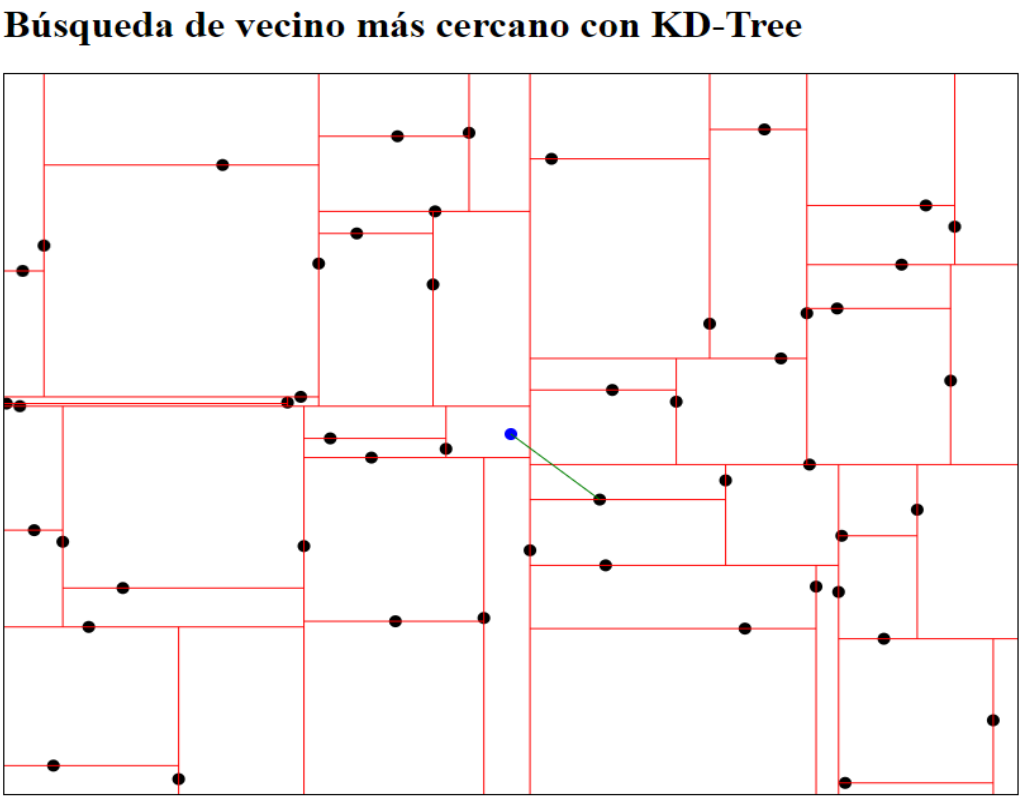
```
92                      ctx.moveTo(desde[ 0 ], desde[ 1 ]);
93                      ctx.lineTo(a[ 0 ], a[ 1 ]);
94                      ctx.strokeStyle = color;
95                      ctx.stroke();
96                  }
97
98                  // Función para dibujar el KD-Tree en el lienzo
99                  function drawKDTree(tree, minX, maxX, minY, maxY, Depth
100     =  0 ) {
101                     if ( ! tree.node) return ;
102                     const [x, y] = árbol.nodo;
103                     si (profundidad %  2  ===  0 ) {
104                       drawLine([x, minY], [x, maxY], 'rojo' );
105                       drawKDTree(tree.left, minX, x, minY, maxY, profundidad
106     +  1 );
107                       drawKDTree(árbol.derecha,  x,  maxX,  minY,  maxY,
108     profundidad +  1 );
109                     } demás {
110                       drawLine([minX, y], [maxX, y], 'rojo' );
111                       drawKDTree(tree.left, minX, maxX, minY, y, profundidad
112     +  1 );
113                       drawKDTree(tree.right,  minX,  maxX,  y,  maxY,
114     profundidad +  1 );
115                     }
116                 }
```

```
117
118                 // Dibuja los puntos generados aleatoriamente
                    points.forEach(point => drawPoint(point));

                    // Dibuja el KD-Tree en el lienzo
                    drawKDTree(kdTree, 0 , canvas.width, 0 , canvas.height);

                    // Dibuja el punto de prueba
                    drawPoint(testPoint, 'blue' );

                    // Busca el vecino más cercano al punto de prueba usando el KD-
            Tree
                    const near = kdTree.nearestNeighbor(testPoint);

                    // Dibuja la línea desde el punto de prueba al vecino más cercano
            encontrado
                    drawLine(testPoint, near, 'green' );

                < /guión>
```

En este código:

- **KDTree**: La clase KDTree implementa la estructura del árbol KD para la búsqueda del vecino más cercano en un espacio 2D.
- **generateRandomPoints**: Genera puntos aleatorios en un espacio definido por el tamaño del lienzo HTML.

- **drawPoint** y **drawLine**: Funciones auxiliares para dibujar puntos y líneas en el lienzo.
- **drawKDTree**: Función para dibujar el KD-Tree en el lienzo, dividiendo el espacio según el eje X o Y en cada nivel del árbol.
- **nearestNeighbor**: Método de la clase KDTree que busca el vecino más cercano a un punto dado utilizando recursión y el árbol KD.
- **Canvas y Contexto**: Se utilizan para dibujar los puntos aleatorios, el árbol KD y la línea que conecta el punto de prueba con su vecino más cercano



Eliminación en un KD-Tree:

- **Proceso:** Se busca el punto a eliminar a través de una búsqueda descendente desde la raíz, utilizando el discriminador para guiar las decisiones.
- **Reestructuración:** Dependiendo de si el nodo a eliminar es una hoja o tiene hijos, se decide cómo reorganizar el árbol para mantener la estructura balanceada.
- **Estrategias:** Puede implicar reemplazar el punto a eliminar con el punto más cercano en términos de distancia euclidiana, seguido de la eliminación de ese punto de reemplazo.
- **Ajustes:** Es posible que se requieran ajustes adicionales para mantener las propiedades del KD-Tree después de la eliminación.

APLICACIONES DE KD-TREES

Estas aplicaciones se benefician de las propiedades de los KD-Trees, como su capacidad para dividir el espacio en regiones y encontrar puntos más cercanos de manera eficiente.

Los KD-Trees son estructuras de datos fundamentales que permiten realizar operaciones eficientes en espacios de varias dimensiones. Desde la búsqueda rápida de vecinos más cercanos hasta la organización y búsqueda de datos en aprendizaje automático y procesamiento de imágenes, estos árboles juegan un papel crucial en diversas aplicaciones tecnológicas. Su capacidad para manejar datos espaciales y optimizar operaciones complejas los convierte en una herramienta esencial en áreas como la robótica, el procesamiento de imágenes y el análisis de grandes volúmenes de datos multidimensionales.

Búsqueda Espacial:

- **Búsqueda de Vecinos Más Cercanos:** Utilizado para identificar rápidamente el punto más cercano a un punto de consulta en espacios de varias dimensiones.

- **Búsqueda por Rango:** Permite obtener todos los puntos dentro de una región específica definida en el espacio multidimensional.

Procesamiento de Imágenes:

- **Recuperación de Imágenes Basada en Contenido:** Emplea KD-Trees para encontrar imágenes similares mediante características visuales extraídas.
- **Detección y Seguimiento de Objetos:** Facilita la localización rápida de regiones de interés en imágenes basadas en características visuales.

Machine Learning:

- **Aprendizaje Supervisado y No Supervisado:** Utilizado para organizar y buscar ejemplos de entrenamiento en espacios de características de alta dimensionalidad.
- **Agrupamiento (Clustering) de Datos:** Facilita la agrupación de datos similares basados en características espaciales.
- **Búsqueda de Vecinos Más Cercanos:** Esencial en algoritmos de clasificación y regresión para encontrar puntos cercanos en el espacio de características.

Robótica:

- **Localización y Mapeo Simultáneo (SLAM):** Utilizado para organizar y buscar puntos de referencia espacial en entornos 3D o 2D.
- **Planificación de Rutas:** Facilita la búsqueda de rutas óptimas basadas en la distancia y restricciones del entorno.
- **Visión por Computadora:** Utilizado en tiempo real para identificar y seguir objetos utilizando características espaciales.

EJEMPLO

Imagina un robot autónomo que utiliza un kd-tree para planificar su ruta evitando obstáculos en un entorno desconocido. Utilizando el árbol para buscar rápidamente áreas libres de obstáculo y calcular la ruta más segura hacia su destino.

El k-d tree es una estructura de datos para particionar el espacio 2D. Aquí hay una implementación básica:

```
1                clase KDTree {
2                  constructor(puntos, profundidad = 0 ) {
3                    este .eje = profundidad % 2 ;
4                    if (puntos.longitud === 0 ) {
5                      este .nodo = nulo ;
6                    } demás {
7                      puntos.sort((a, b) => a[ este .eje] - b[ este .eje]);
8                      mediana constante = Math .floor(puntos.longitud / 2 );
9                      este .nodo = puntos[mediana];
10                     this .left = new KDTree(points.slice( 0 , mediana),
11       profundidad + 1 );
12                     this .right = new KDTree(points.slice(mediana + 1 ),
13       profundidad + 1 );
14                   }
15                 }
dieciséis
17                 Vecino más cercano (punto, profundidad = 0 , mejor = nulo ) {
18                   si ( este .nodo === nulo ) devuelve mejor;
19                   let más cercano = mejor;
```

```
20                  dejar Dist más cercano = mejor ?  this .distance(punto, mejor)
21      :  Infinito ;
22                  distancia constante =  esta .distancia (punto, este .nodo);
23                  si (distancia < distanciamáscercana) {
24                    más cercano =  este .nodo;
25                    Dist más cercano = distancia;
26                  }
27                  eje constante = profundidad %  2 ;
28                  const nextBranch = punto [eje] <  este .nodo [eje] ?  esta
29      .izquierda :  esta .derecha;
30                  const rama opuesta = punto [eje] <  este .nodo [eje] ?  esta
31      .derecha :  esta .izquierda;
32                  más cercano = nextBranch.nearestNeighbor(punto, profundidad
33      +  1 , más cercano);
34                  if  (  Math  .abs(punto[eje]  -  este  .nodo[eje])  <
35      distanciamáscercana) {
36                    más  cercano  =  opuestoBranch.nearestNeighbor(punto,
37      profundidad +  1 , más cercano);
                    }
                    volver más cercano;
                }

                distancia(punto1, punto2) {
                  return  Math .sqrt((punto1[ 0 ] - punto2[ 0 ]) **  2  + (punto1[
      1 ] - punto2[ 1 ]) **  2 );
```

```
    }
}
```

Simulación del Entorno y Planificación de Ruta

Ahora, vamos a crear el entorno y la planificación de ruta, considerando que el robot se moverá en una grilla de puntos.

```
const lienzo =  documento .getElementById ( 'lienzo' );
 const ctx = lienzo.getContext( '2d' );

puntos  constantes  =  generarPuntosAleatorios(  100  );     // Genera
puntos de obstáculos
const kdTree =  new KDTree(points);
 inicio constante = [ 50 , 50 ];
 final constante = [ 750 , 550 ];

función generarPuntosRandom(n) {
    puntos constantes = [];
    para ( sea i =  0 ; i < n; i ++ ) {
      puntos.push([ Math .random() * canvas.width, Math .random()
* canvas.height]);
    }
  puntos de retorno ;
}

función dibujarPunto(punto, color =  'negro' ) {
```

```
20                  ctx.beginPath();
21                  ctx.arc(punto[ 0 ], punto[ 1 ], 3 , 0 , 2  *  Matemáticas .PI);
22                  ctx.fillStyle = color;
23                  ctx.fill();
24              }
25
26              función dibujarLínea(desde, hasta, color =  'azul' ) {
27                  ctx.beginPath();
28                  ctx.moveTo(desde[ 0 ], desde[ 1 ]);
29                  ctx.lineTo(a[ 0 ], a[ 1 ]);
30                  ctx.strokeStyle = color;
31                  ctx.stroke();
32              }
33
34              función    drawKDTree(árbol,    minX,    maxX,    minY,    maxY,
35      profundidad =  0 ) {
36                  if ( ! tree.node) return ;
37                  const [x, y] = árbol.nodo;
38                  si (profundidad %  2  ===  0 ) {
39                    drawLine([x, minY], [x, maxY], 'rojo' );
40                    drawKDTree(tree.left, minX, x, minY, maxY, profundidad +  1
41      );
42                    drawKDTree(árbol.derecha,    x,    maxX,    minY,    maxY,
43      profundidad +  1 );
44                  } demás {
```

```
45                      drawLine([minX, y], [maxX, y], 'rojo' );
46                      drawKDTree(tree.left, minX, maxX, minY, y, profundidad +  1
47          );
48                      drawKDTree(tree.right, minX, maxX, y, maxY, profundidad +
49          1 );
50                  }
51              }
52
53              función planRoute(inicio, fin) {
54                  let current = start;
55                  while (actual[ 0 ] !== fin[ 0 ] || actual[ 1 ] !== fin[ 1 ]) {
56                      const más cercano = kdTree.nearestNeighbor(actual);
57                      drawLine(actual, más cercano, 'verde' );
58                      actual = más cercano;
59                  }
60                  drawLine(actual, final, 'verde' );
61              }


                // Dibuja puntos, árbol y ruta
                points.forEach(point => drawPoint(point));
                drawPoint(inicio, 'azul' );
                drawPoint(fin, 'azul' );
                drawKDTree(kdTree, 0 , lienzo.ancho, 0 , lienzo.alto);
                planRoute(inicio, fin);
```

Explicación

1. **K-D Tree**: El árbol k-d se utiliza para dividir el espacio en regiones y buscar el vecino más cercano a un punto dado rápidamente.

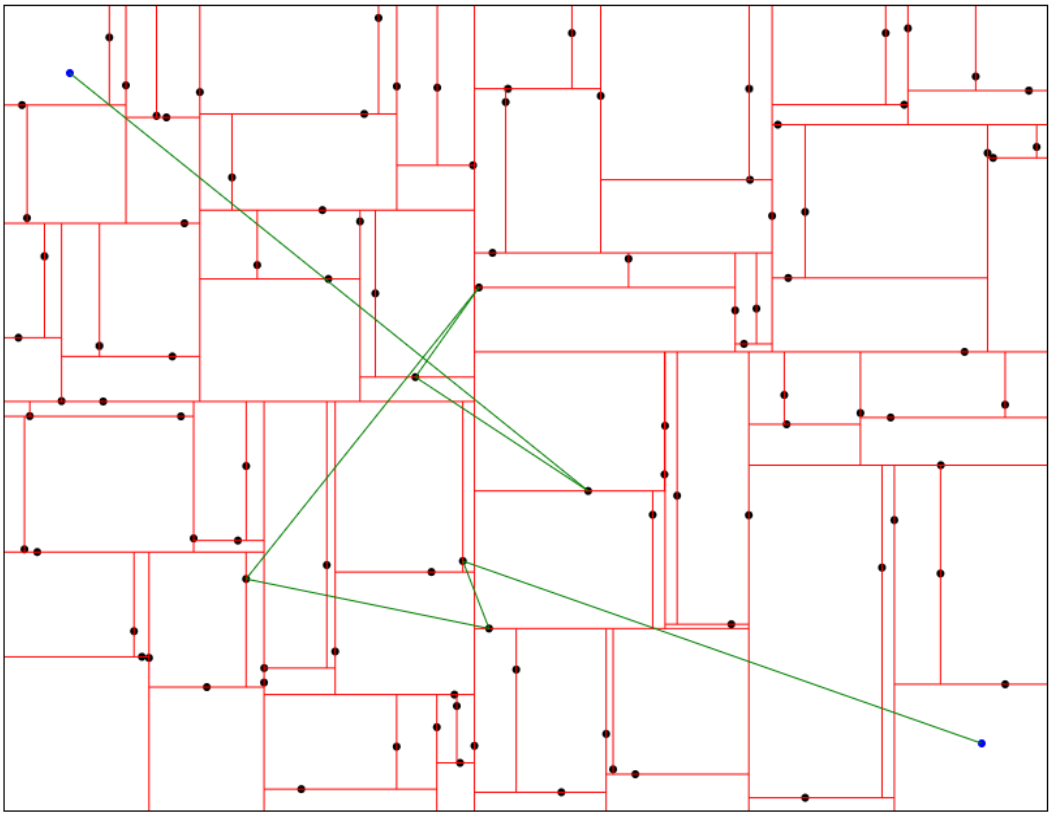
2. **Generación de Obstáculos**: generateRandomPoints crea puntos aleatorios que actúan como obstáculos.

3. **Dibujo en Canvas**: drawPoint y drawLine dibujan puntos y líneas en el lienzo.

4. **Planificación de Ruta**: planRoute encuentra la ruta más segura desde el inicio hasta el final utilizando el k-d tree.

Resultado

Al abrir este archivo HTML en un navegador, deberías ver una visualización del entorno con puntos representando obstáculos y una ruta verde que el robot sigue desde el inicio hasta el destino, evitando los obstáculos utilizando el k-d tree.



VENTAJAS Y DESVENTAJAS DE KD-TREES

En resumen, los KD-Trees tienen varias ventajas, como su eficiencia en búsquedas y manejo de datos de alta dimensionalidad, pero también tienen desventajas, como su complejidad de construcción y limitaciones en la búsqueda de vecinos.

Ventajas de los KD-Trees:

1. **Eficiencia en la Búsqueda Espacial:**
   - Los KD-Trees permiten búsquedas rápidas de vecinos más cercanos y de puntos dentro de un rango específico en espacios de alta dimensionalidad.
   - Son más eficientes que los métodos de búsqueda lineal, especialmente cuando se manejan grandes volúmenes de datos.
2. **Estructura Jerárquica y Organización de Datos:**
   - Organizan los datos de manera jerárquica, lo que facilita la búsqueda y la inserción de datos en espacios multidimensionales.
   - Proporcionan una forma estructurada de subdividir el espacio en regiones más pequeñas.
3. **Aplicaciones Versátiles:**
   - Son aplicables en una amplia gama de problemas de búsqueda espacial, procesamiento de imágenes, machine learning, robótica y más.
   - Permiten realizar operaciones avanzadas como búsqueda de vecinos más cercanos, clustering y recuperación de datos basada en contenido.
4. **Flexibilidad en la Implementación:**
   - Pueden ser adaptados y extendidos para manejar diferentes tipos de datos y requisitos específicos de aplicación.
   - Permiten ajustar parámetros como la profundidad del árbol y los criterios de división para optimizar el rendimiento.

Desventajas de los KD-Trees:

1. **Maldición de la Dimensionalidad:**

- Su eficiencia de búsqueda puede disminuir significativamente en espacios de alta dimensionalidad debido a la maldición de la dimensionalidad.
- En espacios con muchas dimensiones, las distancias entre puntos tienden a volverse uniformes, lo que reduce la eficacia del árbol.

2. **Construcción y Mantenimiento Costoso:**

- La construcción inicial de un KD-Tree puede ser costosa computacionalmente, especialmente para grandes conjuntos de datos.
- Rebalancear o actualizar el árbol puede ser complejo y costoso en términos de tiempo y recursos computacionales.

3. **Dependencia de la Distribución de los Datos:**

- La eficiencia del KD-Tree puede depender de la distribución y densidad de los datos.
- Datos con distribuciones irregulares pueden llevar a árboles mal balanceados que afectan negativamente el rendimiento de las consultas.

4. **Limitaciones en Casos Específicos:**

- No son ideales para todos los tipos de datos y aplicaciones. Por ejemplo, en conjuntos de datos muy dinámicos donde los puntos se insertan y eliminan frecuentemente, otras estructuras como los R-Trees podrían ser más adecuadas.

OPTIMIZACIÓN Y BALANCEO DE KD-TREES

Los KD-Trees son optimizados y balanceados mediante diversas técnicas para mejorar su rendimiento y eficiencia:

Técnicas de Balanceo de KD-Trees:

1. **Balanceo durante la Construcción:**

   - Durante la construcción inicial del KD-Tree, se asegura que las subdivisiones del espacio sean equitativas.
   - Se alterna el eje de división en cada nivel para distribuir los puntos de manera uniforme en cada subárbol generado.

2. **Rebalanceo Posterior:**

   - Después de la construcción inicial, se realiza el rebalanceo si las operaciones de inserción o eliminación afectan la estructura del árbol.
   - Implica redistribuir los nodos y posiblemente reconstruir parcial o totalmente el árbol para mantenerlo balanceado y optimizado.

3. **División Ponderada (Weighted Split):**

   - Asigna un peso a cada punto basado en su importancia o frecuencia de acceso.
   - Durante la construcción o el rebalanceo, los puntos con mayor peso se colocan estratégicamente para minimizar la profundidad del árbol y mejorar la eficiencia de búsqueda.

4. **División Median-of-Median (Mediana de Medianas):**

   - En lugar de usar solo la mediana para dividir los puntos, se divide en grupos medianos más pequeños.
   - Puede conducir a árboles más balanceados y reducir situaciones de peores casos en las búsquedas.

Rebalanceo Dinámico:

El rebalanceo dinámico es crucial en entornos dinámicos donde los datos cambian con frecuencia:

- **Inserción y Eliminación Eficientes:** Implementación de algoritmos eficientes para insertar y eliminar puntos sin afectar la estructura del árbol de manera drástica.

- **Monitoreo de la Profundidad del Árbol:** Establecimiento de umbrales para la profundidad del árbol y realización de operaciones de rebalanceo cuando se superan estos umbrales.
- **Reconstrucción Incremental:** Utilización de técnicas de reconstrucción que minimicen la interrupción del acceso a los datos durante el rebalanceo.

Ejemplo Implementación Práctico

A continuación te muestro un ejemplo básico de como podrías implementar alguna de estas técnicas en JavaScript para construir y revalancear un kd-trees.

```
1
                clase KDTree {
2
                  constructor(puntos = [], profundidad =  0 ) {
3
                    este .nodo =  nulo ;
4
                    este .left =  nulo ;
5
                    este .right =  nulo ;
6
                    este .build(puntos, profundidad);
7
                  }
8
9
                  construir(puntos, profundidad) {
10
                    si (puntos.longitud ===  0 ) retorno ;
11
12
                    eje constante = profundidad %  2 ;
13
                    puntos.sort((a, b) => a[eje] - b[eje]);
14
                     mediana constante =  Math .floor(puntos.longitud /  2 );
15
dieciséis
                    este .nodo = puntos[mediana];
17
```

```
18                      this .left =  new KDTree(points.slice( 0 , mediana),
19          profundidad + 1 );
20                      this .right =  new KDTree(points.slice(mediana + 1 ),
21          profundidad + 1 );
22                  }
23
24                  insertar (punto, profundidad = 0 ) {
25                      si ( este .nodo === nulo ) {
26                          este .nodo = punto;
27                          este .left = nuevo KDTree();
28                          este .right = nuevo KDTree();
29                          devolver ;
30                      }
31
32                      eje constante = profundidad % 2 ;
33                      if (punto[eje] < este .nodo[eje]) {
34                          este .left.insert(punto, profundidad + 1 );
35                      } else {
36                          this .right.insert(punto, profundidad + 1 );
37                      }
38                  }
39
40                  aPuntos() {
41                      si ( este .nodo === nulo ) devuelve [];
42
```

```
43                          return [... este .left.toPoints(), este .nodo, ... este
44              .right.toPoints()];
45                      }
46                  }
47
48                  const lienzo =  documento .getElementById ( 'lienzo' );
49                   const ctx = lienzo.getContext( '2d' );
50                   let puntos = generarPuntosRandom( 10 , lienzo.ancho,
51              lienzo.alto);
52                   let kdTree =  new KDTree(puntos);
53
54                  función generarPuntosRandom(n, ancho, alto) {
55                      puntos constantes = [];
56                      para ( sea i =  0 ; i < n; i ++ ) {
57                          puntos.push([ Math .random() * ancho, Math .random() *
58              alto]);
59                      }
60                      puntos de retorno ;
61                  }
62
63                  función dibujarPunto(punto, color =  'negro' ) {
64                      ctx.beginPath();
sesenta     y           ctx.arc(punto[ 0 ], punto[ 1 ], 5 , 0 , 2  *  Matemáticas .PI);
cinco                   ctx.fillStyle = color;
66                      ctx.fill();
```

```
67                  }
68
69                  función dibujarLínea(desde, hasta, color =  'azul' ) {
70                      ctx.beginPath();
71                      ctx.moveTo(desde[ 0 ], desde[ 1 ]);
72                      ctx.lineTo(a[ 0 ], a[ 1 ]);
73                      ctx.strokeStyle = color;
74                      ctx.stroke();
75                  }
76
77                  función  drawKDTree(árbol,  minX,  maxX,  minY,  maxY,
78              profundidad =  0 ) {
79                      if ( ! tree.node) return ;
80
81                      const [x, y] = árbol.nodo;
82                      drawPoint([x, y], 'rojo' );
83
84                      if (profundidad %  2  ===  0 ) { // Dividir según el eje X
85                          drawLine([x, minY], [x, maxY], 'red' );
86                          drawKDTree(tree.left, minX, x, minY, maxY, profundidad +
87              1 );
88                          drawKDTree(árbol.derecha,  x,  maxX,  minY,  maxY,
89              profundidad +  1 );
90                      } else { // Dividir según el eje Y
91                          drawLine([minX, y], [maxX, y], 'blue' );
```

```
92                  drawKDTree(tree.left, minX, maxX, minY, y, profundidad +
93          1 );
94                  drawKDTree(tree.right, minX, maxX, y, maxY, profundidad
95          + 1 );
96              }
97          }
98
99          función addRandomPoint() {
100             const newPoint = [ Math .random() * lienzo.ancho, Math
101         .random() * lienzo.alto];
102             puntos.push(nuevoPunto);
103             kdTree.insert(nuevoPunto);
104             volver a dibujar();
105         }
106
107         función reequilibrarKDTree() {
            kdTree = nuevo KDTree(puntos);
            volver a dibujar();
            }


            función redibujar() {
            ctx.clearRect( 0 , 0 , lienzo.ancho, lienzo.alto); // Limpiar lienzo
            puntos.forEach(punto => drawPoint(punto, 'negro' )); // Dibujar
        puntos iniciales
```

```
                drawKDTree(kdTree, 0 , canvas.width, 0 , canvas.height); //
        Dibujar KD-Tree
            }

            // Dibuja puntos iniciales y KD-Tree
            volver a dibujar();
```

Descripción del Código

1. **Clase KDTree**:

- **Constructor**: Inicializa el árbol y construye usando build.
- **build**: Construye el k-d tree a partir de un conjunto de puntos.
- **insert**: Inserta un nuevo punto en el k-d tree.
- **toPoints**: Devuelve todos los puntos en el árbol (útil para reequilibrar).

2. **Generación de Puntos**:

- **generateRandomPoints**: Crea un conjunto de puntos aleatorios en el lienzo.

3. **Funciones de Dibujo**:

- **drawPoint**: Dibuja un punto en el lienzo.
- **drawLine**: Dibuja una línea que representa la división en el k-d tree.
- **drawKDTree**: Dibuja recursivamente las divisiones del k-d tree.

4. **Funciones de Control**:

- **addRandomPoint**: Agrega un punto aleatorio al k-d tree y redibuja.
- **reequilibrarKDTree**: Reequilibra el k-d tree a partir de los puntos actuales y redibuja.

5. **Redibujar**:

- **redraw**: Limpia el lienzo y redibuja todos los puntos y el k-d tree.

Cómo Funciona

- **Puntos Negros**: Representan los puntos en el espacio.
- **Puntos Rojos**: Representan los nodos del k-d tree.
- **Líneas Rojas**: Representan las divisiones a lo largo del eje X.
- **Líneas Azules**: Representan las divisiones a lo largo del eje Y.

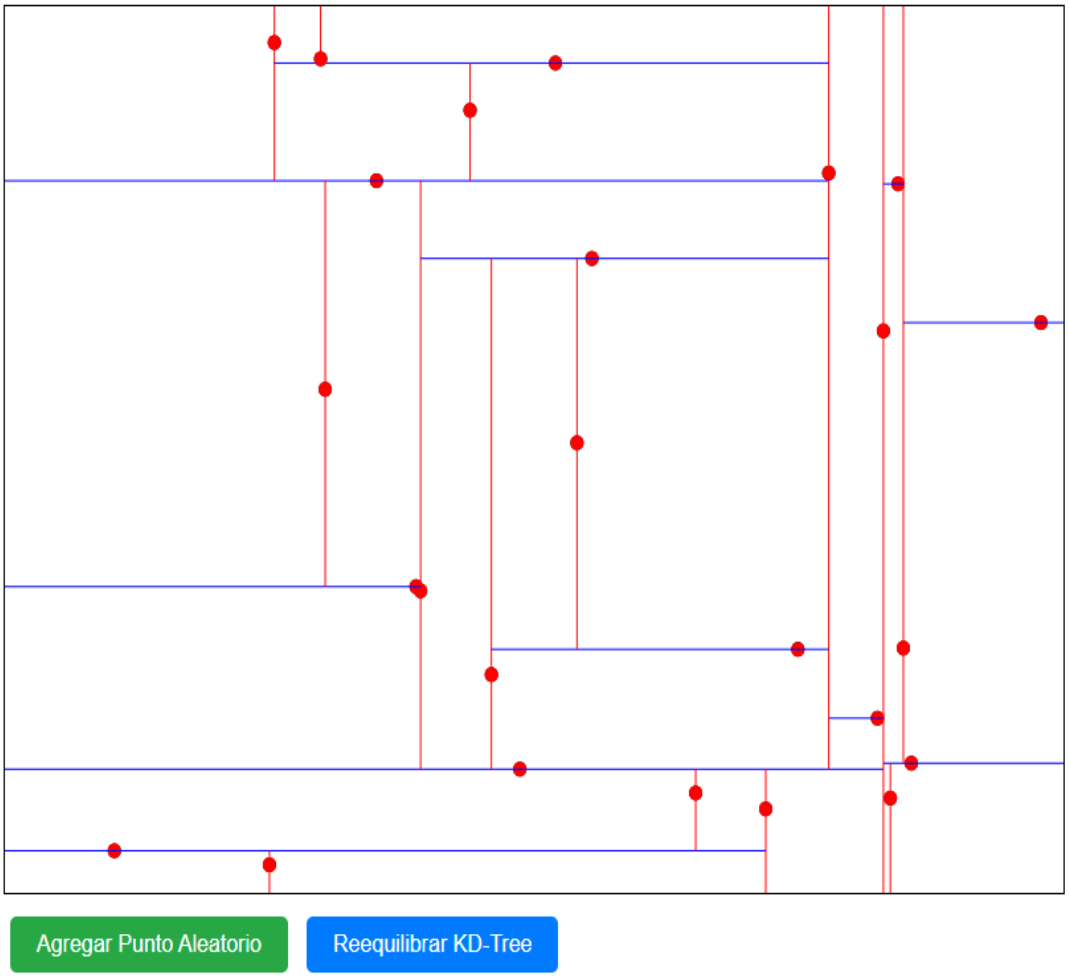
Ejecución

Paso 1: Guarda el código en un archivo index.html.

Paso 2: Abre index.html en tu navegador.

Paso 3: Haz clic en "Agregar Punto Aleatorio" para añadir puntos al k-d tree.

Paso 4: Haz clic en "Reequilibrar KD-Tree" para reconstruir el árbol con los puntos actuales.



Este ejemplo proporciona una manera interactiva de entender cómo se construye y reequilibra un k-d tree en JavaScript y cómo se visualizan las divisiones en un espacio 2D.

COMPARACIÓN CON OTRAS ESTRUCTURAS DE DATOS

La elección entre estas estructuras de datos debe basarse en la naturaleza específica del problema, las características de los datos y los requisitos de rendimiento de la aplicación. Cada una tiene sus fortalezas y se adapta mejor a diferentes contextos y tipos de datos.

Quad-Trees

Definición: Un Quad-Tree es una estructura de datos que divide el espacio en cuadrados, cada uno de los cuales puede contener un punto o un subárbol.

Ventajas: Los Quad-Trees son útiles para manejar datos de alta dimensionalidad y pueden ser más eficientes que los KD-Trees en algunos casos.

Desventajas: Los Quad-Trees pueden ser más complejos de construir y mantener que los KD-Trees, especialmente en espacios de alta dimensionalidad.

R-Trees

Definición: Un R-Tree es una estructura de datos que divide el espacio en rectángulos, cada uno de los cuales puede contener un punto o un subárbol.

Ventajas: Los R-Trees son útiles para manejar datos de alta dimensionalidad y pueden ser más eficientes que los KD-Trees en algunos casos.

Desventajas: Los R-Trees pueden ser más complejos de construir y mantener que los KD-Trees, especialmente en espacios de alta dimensionalidad.

Hashing Espacial

Definición: El Hashing Espacial es una técnica que utiliza hash functions para mapear los puntos en un espacio multidimensional a una clave única.

Ventajas: El Hashing Espacial es rápido y eficiente para realizar búsquedas espaciales, especialmente en espacios de baja dimensionalidad.

Desventajas: El Hashing Espacial puede ser menos eficiente que los KD-Trees y R-Trees en espacios de alta dimensionalidad y puede tener problemas con la colisión de claves.

VENTAJAS Y DESVENTAJAS COMPARACIÓN CON OTRAS ESTRUCTURAS DE DATOS

Ventajas:

Eficiencia en Búsquedas: Los KD-Trees permiten realizar búsquedas de vecinos más cercanos de manera rápida y eficiente en espacios multidimensionales.

Manejo de Datos de Alta Dimensionalidad: Los KD-Trees son útiles para manejar datos de alta dimensionalidad, donde la "curva de dimensionalidad" puede hacer que las búsquedas sean lentas.

Búsqueda Espacial: Los KD-Trees se utilizan comúnmente para búsquedas espaciales, como encontrar el punto más cercano a un punto de búsqueda en un espacio multidimensional.

Desventajas:

Desempeño que se degrade con el aumento de dimensiones: El rendimiento de los KD-Trees puede degradarse significativamente cuando el número de dimensiones aumenta, especialmente si los datos no están uniformemente distribuidos.

No apto para conjuntos de datos dinámicos: La inserción y eliminación de puntos en un KD-Tree pueden ser costosas y requerir reorganizar el árbol, lo que puede ser un problema para conjuntos de datos dinámicos.

Complejidad de la Construcción: La construcción de un KD-Tree puede ser compleja y requerir un proceso recursivo para dividir el espacio en regiones.

Limitaciones en la Búsqueda de Vecinos: Los KD-Trees pueden tener limitaciones en la búsqueda de vecinos más cercanos, especialmente si los datos están muy dispersos o no están bien distribuidos.

CONSIDERACIONES DE IMPLEMENTACIÓN PARA KD-TREES

Estas consideraciones aseguran que los KD-Trees sean implementados de manera óptima y eficiente, adaptándose a diferentes necesidades y garantizando un rendimiento adecuado en entornos variables y desafiantes.

Complejidad Temporal y Espacial:

- **Construcción:** La construcción de un KD-Tree tiene una complejidad de O (nlogf_0n)O(n \log n)O(nlogn), donde nnn es el número de puntos. Esto se debe a la subdivisión recursiva del espacio en cada dimensión.
- **Búsqueda:** En espacios de baja dimensionalidad, la búsqueda de vecinos más cercanos tiene complejidad promedio O(logf_0n)O(\log n)O(logn), pero puede degradarse hasta O(n)O(n)O(n) en dimensiones altas debido a la maldición de la dimensionalidad.
- **Espacio de Almacenamiento:** El KD-Tree consume espacio adicional para almacenar los nodos y sus relaciones jerárquicas. Aunque eficiente en términos de espacio comparado con estructuras más densas, como Hash Tables, puede ser lineal en el peor de los casos.

Manejo de Datos de Alta Dimensionalidad:

- **Maldición de la Dimensionalidad:** En dimensiones altas (>10), la eficiencia del KD-Tree puede disminuir significativamente debido a la uniformidad de las distancias entre puntos. Alternativas como R-Trees o hashing espacial pueden ser más adecuadas en estos casos.

Selección de Dimensiones:

- **Estrategias de División:** La selección adecuada del eje de división en cada nivel del árbol es crucial. Métodos como la alternancia de dimensiones, la mediana de medianas o la selección basada en la varianza pueden mantener el árbol balanceado y optimizado.

Escalabilidad y Eficiencia:

- **Rebalanceo Dinámico:** Implementar técnicas de rebalanceo dinámico es esencial para manejar actualizaciones frecuentes en los datos. Esto incluye reconstrucción incremental del árbol y ajuste dinámico de umbrales de profundidad para mantener la eficiencia.
- **Optimización de Consultas:** Aplicar técnicas como el preordenamiento de puntos según proximidad espacial antes de la inserción en el árbol, y técnicas de poda durante la búsqueda para reducir comparaciones, puede mejorar significativamente el rendimiento.

Optimizaciones Específicas de Aplicación:

- **Aplicaciones Específicas:** Ajustar parámetros como la profundidad del árbol, criterios de división y estrategias de búsqueda según los requisitos específicos de cada aplicación.
- **Implementación Eficiente:** Utilizar estructuras de datos eficientes para representar nodos y relaciones en el árbol. Considerar implementaciones paralelas o distribuidas para manejar grandes volúmenes de datos y entornos de alto rendimiento.

HERRAMIENTAS Y LIBRERÍAS PARA KD-TREES

Para trabajar eficientemente con KD-Trees (árboles K-dimensionales) en diferentes entornos de desarrollo, existen varias herramientas y librerías disponibles en lenguajes populares como C++, Python y Java. Estas librerías ofrecen implementaciones robustas y optimizadas para la manipulación y consulta de datos en espacios multidimensionales

Implementación En Lenguajes Populares

C++

- **CGAL (Computational Geometry Algorithms Library)**

- ✓ **Características:** Ofrece implementaciones robustas de estructuras de datos geométricas, incluyendo KD-Trees.
- ✓ **Uso:** Ideal para aplicaciones de geometría computacional y procesamiento de datos espaciales.
- ✓ **Enlace:** Sitio web de CGAL

- **FLANN (Fast Library for Approximate Nearest Neighbors)**

- ✓ **Características:** Biblioteca para búsquedas eficientes de vecinos más cercanos utilizando KD-Trees y otras estructuras.
- ✓ **Uso:** Útil para aplicaciones que requieren búsqueda rápida en grandes conjuntos de datos.
- ✓ **Enlace:** Repositorio FLANN en GitHub

Python

- **scikit-learn**

- ✓ **Características:** Biblioteca de aprendizaje automático que incluye implementaciones de KD-Trees para búsqueda de vecinos más cercanos.
- ✓ **Uso:** Ampliamente utilizado en aprendizaje automático y minería de datos.
- ✓ **Enlace:** Sitio web de scikit-learn

- **SciPy**

- ✓ **Características:** Biblioteca científica que ofrece funciones para cálculos numéricos y científicos, incluyendo implementaciones de KD-Trees.
- ✓ **Uso:** Ideal para aplicaciones científicas y análisis de datos.
- ✓ **Enlace:** Sitio web de SciPy

Java

- **JTS (Java Topology Suite)**

- **Características:** Biblioteca para el procesamiento de datos espaciales que incluye estructuras como KD-Trees para consultas espaciales eficientes.
- **Uso:** Utilizado en aplicaciones GIS y análisis espacial.
- **Enlace:** Sitio web de JTS

- **EJML (Efficient Java Matrix Library)**
  - **Características:** Biblioteca para operaciones matriciales eficientes en Java, útil para implementar estructuras como KD-Trees.
  - **Uso:** Ideal para aplicaciones que requieren manipulación de datos multidimensionales y búsquedas eficientes.
  - **Enlace:** Repositorio EJML en GitHub

COMPARACIÓN DE RENDIMIENTO

La elección de la librería para KD-Trees puede depender de varios factores críticos:

- **Tiempo de Construcción:** Evaluar la eficiencia en la construcción del árbol a partir de conjuntos de datos específicos.
- **Eficiencia en Consultas:** Medir el tiempo de respuesta en operaciones como búsqueda de vecinos más cercanos y consultas de rango en diferentes configuraciones de datos.
- **Escalabilidad:** Probar el rendimiento con grandes volúmenes de datos y dimensiones variables para asegurar la escalabilidad de la solución.
- **Uso de Memoria:** Comparar el consumo de memoria de cada implementación, especialmente importante en entornos con limitaciones de recursos.

Consideraciones Adicionales

- **Documentación y Soporte:** La calidad de la documentación y el soporte disponible juegan un papel crucial en la selección de la librería adecuada.
- **Facilidad de Uso:** Evaluar la integración y la facilidad de uso de la librería en el contexto específico de la aplicación.

CASOS DE ESTUDIO Y EJEMPLOS PRÁCTICOS DE KD-TREES

Los KD-Trees son herramientas poderosas en el ámbito de la informática gráfica, aprendizaje automático y simulaciones físicas debido a su capacidad para manejar eficientemente grandes volúmenes de datos en espacios multidimensionales, facilitando operaciones críticas como búsqueda de vecinos más cercanos, clasificación, y detección de colisiones.

Reconocimiento de Imágenes

En el campo del reconocimiento de imágenes, los KD-Trees son utilizados para:

- **Comparación de Características:** Permiten buscar imágenes similares utilizando descriptores como histogramas de color o texturas, facilitando la recuperación rápida de imágenes basada en contenido visual.

Búsqueda Espacial

Los KD-Trees son esenciales en aplicaciones que requieren búsqueda eficiente en espacios multidimensionales:

- **Búsqueda de Vecinos más Cercanos:** Usados en geolocalización, sistemas de recomendación basados en ubicación y en bases de datos para encontrar rápidamente puntos cercanos a un punto de consulta.

Clasificación y Regresión

En aprendizaje automático y minería de datos, los KD-Trees tienen aplicaciones clave:

- **Clasificación Basada en Vecinos:** Permiten clasificar nuevos datos basados en la mayoría de votos de los vecinos más cercanos, siendo útiles en problemas de clasificación con datos etiquetados.
- **Regresión:** Utilizados para predecir valores continuos basados en la proximidad a puntos conocidos en el espacio multidimensional.

Procesamiento de Nubes de Puntos

En aplicaciones de visión por computadora y gráficos 3D, los KD-Trees facilitan:

- **Renderizado y Visualización:** Son cruciales para la renderización eficiente de objetos 3D y la detección rápida de colisiones entre objetos representados como nubes de puntos en espacios tridimensionales.

Búsqueda de Colisiones en Simulaciones Físicas

En simulaciones físicas y videojuegos, los KD-Trees son utilizados para:

- **Detección de Colisiones:** Permiten identificar rápidamente qué objetos están en proximidad unos de otros, crucial para la prevención de colisiones en simulaciones físicas interactivas.

CONCLUCION

En conclusión, los KD-Trees son una herramienta poderosa y eficiente para la organización y búsqueda de datos espaciales en espacios multidimensionales. A pesar de las limitaciones en su construcción y su desempeño en altas dimensiones, los KD-Trees se destacan en muchas aplicaciones prácticas, proporcionando una base sólida para operaciones de búsqueda espacial y partición de datos en varias dimensiones.

BIBLIOGRAFÍA

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

"Spatial Indexing" by Markus Schneider.

"Multidimensional Data Structures: Algorithms and Applications" by Hanan Samet.

"Algorithms in C++ Part 5: Graph Algorithms" by Robert Sedgewick.

Wikipedia: KD-Tree.

"K-D Trees for Semantics-Based Access to Images" by Henry Fuchs, Kenneth P. Smith.