SEMANA 13 (junio 24, 26 ,28)

ÁRBOLES PARA FUNCIONES DE DISTANCIAS CONTINUAS

Para estudiar árboles para funciones de distancias continuas, es fundamental comprender varios conceptos clave que afectan cómo se construyen, organizan y utilizan estos árboles. Aquí hay una lista de los conceptos más importantes:

1. Espacio Métrico

Un espacio métrico es un conjunto de elementos con una función de distancia (métrica) que cumple con las siguientes propiedades:

- **No negatividad**: La distancia entre dos puntos es siempre no negativa.
- **Identidad**: La distancia entre un punto y sí mismo es cero.
- **Simetría**: La distancia entre dos puntos A y B es la misma que la distancia entre B y A.
- **Desigualdad triangular**: La distancia directa entre dos puntos es menor o igual que la suma de las distancias pasando por un tercer punto.

2. Función de Distancia

La función de distancia es crucial para los árboles métricos. En el contexto de distancias continuas, las métricas comunes incluyen:

- **Distancia Euclidiana**: Medida en línea recta entre dos puntos en un espacio euclidiano.
- **Distancia Manhattan**: Suma de las diferencias absolutas de sus coordenadas.
- **Distancia de Minkowski**: Generaliza las distancias Euclidiana y Manhattan.
- **Distancia Coseno**: Medida de la similitud entre dos vectores.

3. Particionamiento del Espacio

El particionamiento del espacio es cómo se dividen los datos en subespacios para organizar eficientemente las búsquedas. En árboles métricos, esto puede implicar:

- **Puntos de Referencia (Vantage Points)**: Puntos seleccionados para dividir el espacio en regiones basadas en distancias.
- **Ejes de Partición**: Selección de una dimensión y un valor de corte para dividir el espacio (usado en árboles k-d).

4. Estructuras de Árbol Comunes

Varios tipos de árboles métricos se utilizan para manejar distancias continuas, cada uno con sus propias ventajas y aplicaciones:

1. Árboles k-d (K-Dimensional Trees)

- Utilizan ejes de partición para dividir el espacio.
- Eficientes en espacios de baja dimensionalidad.
- Búsqueda de vecinos más cercanos y consultas de rango.

2. Árboles VP (Vantage Point Trees)

- Utilizan puntos de referencia para dividir el espacio.
- Adecuados para espacios de alta dimensionalidad.
- Búsqueda de vecinos más cercanos y consultas de rango.

3. Árboles Cover (Cover Trees)

- Utilizan niveles jerárquicos de cobertura para organizar puntos.
- Eficientes en espacios de alta dimensionalidad y grandes conjuntos de datos.

4. Árboles Ball (Ball Trees)

- Utilizan hiperesferas (balls) para dividir el espacio.
- Eficientes para problemas de alta dimensionalidad y consultas de rango.

5. Operaciones de Búsqueda

Las operaciones de búsqueda son fundamentales para los árboles métricos y determinan su eficiencia. Las operaciones comunes incluyen:

- **Búsqueda de Vecinos Más Cercanos (k-NN)**: Encontrar los k puntos más cercanos a un punto de consulta.
- **Consultas de Rango**: Encontrar todos los puntos dentro de un radio específico desde un punto de consulta.
- **Búsqueda Aproximada**: Sacrificar precisión por velocidad en búsquedas de grandes conjuntos de datos.

6. Algoritmos de Construcción

El proceso de construcción del árbol es crítico para su eficiencia en operaciones de búsqueda. Los algoritmos de construcción varían según el tipo de árbol, pero generalmente implican:

- **Selección de Puntos de Partición**: Elegir puntos o ejes para dividir el espacio.
- **Balanceo del Árbol**: Asegurar que el árbol esté balanceado para optimizar la búsqueda.
- **Inserción y Eliminación**: Métodos para agregar y eliminar puntos mientras se mantiene la estructura del árbol.

7. Aplicaciones

Comprender las aplicaciones prácticas de los árboles métricos ayuda a contextualizar su importancia:

- **Bases de Datos Espaciales**: Consultas de proximidad en datos geográficos.
- **Visión por Computadora**: Búsqueda de imágenes similares basadas en características métricas.
- **Biología Computacional**: Análisis de secuencias genómicas y búsqueda de secuencias similares.
- **Sistemas de Recomendación**: Encontrar elementos similares en sistemas de recomendación basados en métricas de similitud.

EJEMPLO PRÁCTICO DE ARBOLES PARA FUNCIONES DE DISTANCIAS CONTINUAS

Problema 1

Imagina que trabajas para una empresa que desarrolla un sistema de recomendación de música. La empresa quiere ofrecer a los usuarios recomendaciones de canciones similares a las que ya les gustan. Para ello, necesita una forma eficiente de encontrar canciones similares basándose en diversas características (como tempo, pitch, duración, etc.).

Objetivo

Utilizar un Árbol VP (Vantage Point Tree) para realizar búsquedas eficientes de canciones similares en un espacio métrico continuo.

Ejemplo Práctico en JavaScript

Definir las Características de las Canciones

Cada canción será representada por un punto en un espacio métrico continuo. Por ejemplo, una canción puede ser representada por un vector de características como [tempo, pitch, duración].

```
const songs = [
    { id: 1, features: [120, 55, 210] }, // [tempo, pitch, duración]
    { id: 2, features: [130, 60, 200] },
    { id: 3, features: [110, 53, 180] },
    { id: 4, features: [125, 58, 240] },
    { id: 5, features: [140, 65, 220] }
];
```

Definir la Función de Distancia

Usaremos la distancia Euclidiana para medir la similitud entre dos canciones.

```
function euclideanDistance(a, b) {
    return Math.sqrt(a.reduce((sum, val, i) => sum + (val - b[i]) ** 2, 0));
}
```

Implementar el Nodo y el Árbol VP

Cada nodo del Árbol VP contendrá una canción, un radio que define la distancia para dividir las canciones, y referencias a los nodos hijos.

```
class Node {
   constructor(song) {
       this.song = song;
       this.radius = 0;
       this.inside = null;
       this.outside = null;
   }
}

class VPTree {
   constructor(songs, distanceFunc) {
       this.root = this.buildTree(songs);
       this.distanceFunc = distanceFunc;
   }

   buildTree(songs) {
       if (songs.length === 0) return null;
       const index = Math.floor(Math.random() * songs.length);
       const song = songs[index];
       const node = new Node(song);
       songs.splice(index, 1);

       if (songs.length === 0) return node;

       const distances = songs.map(s => this.distanceFunc(song.features, s.features));
       const median = this.median(distances);
       node.radius = median;
```

```
        const insideSongs = songs.filter((s, i) => distances[i] <= median);
        const outsideSongs = songs.filter((s, i) => distances[i] > median);

        node.inside = this.buildTree(insideSongs);
        node.outside = this.buildTree(outsideSongs);

        return node;
    }

    median(values) {
        values.sort((a, b) => a - b);
        const mid = Math.floor(values.length / 2);
        return values[mid];
    }

    search(song, maxResults, node = this.root, neighbors = []) {
        if (!node) return neighbors;

        const dist = this.distanceFunc(song.features, node.song.features);
        if (neighbors.length < maxResults || dist < neighbors[0].distance) {
            neighbors.push({ song: node.song, distance: dist });
            neighbors.sort((a, b) => b.distance - a.distance);
            if (neighbors.length > maxResults) neighbors.shift();
        }

        const checkInsideFirst = dist < node.radius;

        if (checkInsideFirst) {
            this.search(song, maxResults, node.inside, neighbors);
            if (neighbors.length<maxResults || Math.abs(node.radius - dist) < neighbors[0].distance) {
                this.search(song, maxResults, node.outside, neighbors);
            }
        } else {
            this.search(song, maxResults, node.outside, neighbors);
            if (neighbors.length<maxResults || Math.abs(node.radius - dist) < neighbors[0].distance) {
                this.search(song, maxResults, node.inside, neighbors);
            }
        }

        return neighbors;
    }
}
```

Usar el Árbol VP para Encontrar Canciones Similares

Construimos el Árbol VP con las canciones disponibles y realizamos una búsqueda de las canciones más similares a una canción de consulta.

```
const tree = new VPTree(songs, euclideanDistance);

const querySong = { id: 6, features: [128, 59, 210] };
const maxResults = 2;
```

```
const similarSongs = tree.search(querySong, maxResults);
console.log('Songs similar to query:', similarSongs);
```

Explicación del Ejemplo Práctico

8. **Características de las Canciones:** Las canciones se representan como vectores de características en un espacio métrico.
9. **Función de Distancia:** La distancia Euclidiana mide la similitud entre las características de las canciones.
10. **Árbol VP:** Se construye un árbol VP que organiza las canciones en base a distancias continuas.
11. **Búsqueda de Similitud:** El Árbol VP se utiliza para encontrar las canciones más similares a una canción de consulta.

EJEMPLO PRÁCTICO DE ARBOLES PARA FUNCIONES DE DISTANCIAS CONTINUAS

Problema 2

Tu empresa debe desarrollar un sistema de búsqueda de imágenes similares. La empresa quiere ofrecer a los usuarios la capacidad de buscar imágenes basándose en sus características visuales, como el color promedio, la textura, y la forma. Para ello, necesitan una forma eficiente de encontrar imágenes similares basándose en diversas características continuas.

Objetivo

Utilizar un Árbol VP (Vantage Point Tree) para realizar búsquedas eficientes de imágenes similares en un espacio métrico continuo.

Ejemplo Práctico en JavaScript

5. Definir las Características de las Imágenes

Cada imagen será representada por un punto en un espacio métrico continuo. Por ejemplo, una imagen puede ser representada por un vector de características como [color promedio (RGB), textura, forma].

```
const images = [
    { id: 1, features: [128, 64, 64, 0.5, 0.8] }, // [R, G, B, textura, forma]
```

```
    {    id:    2,    features:    [130,    60,    70,    0.6,    0.7]    },
    {    id:    3,    features:    [120,    65,    60,    0.4,    0.9]    },
    {    id:    4,    features:    [125,    70,    80,    0.7,    0.6]    },
    {    id:    5,    features:    [140,    55,    75,    0.8,    0.5]    }
];
```

2. Definir la Función de Distancia

Usaremos la distancia Euclidiana para medir la similitud entre dos imágenes.

```
function euclideanDistance(a, b) {
    return Math.sqrt(a.reduce((sum, val, i) => sum + (val - b[i]) ** 2, 0));
}
```

3. Implementar el Nodo y el Árbol VP

Cada nodo del Árbol VP contendrá una imagen, un radio que define la distancia para dividir las imágenes, y referencias a los nodos hijos.

```
class Node {
    constructor(image) {
        this.image = image;
        this.radius = 0;
        this.inside = null;
        this.outside = null;
    }
}

class VPTree {
    constructor(images, distanceFunc) {
        this.root = this.buildTree(images);
        this.distanceFunc = distanceFunc;
    }

    buildTree(images) {
        if (images.length === 0) return null;

        const index = Math.floor(Math.random() * images.length);
        const image = images[index];
        const node = new Node(image);
        images.splice(index, 1);

        if (images.length === 0) return node;

        const distances = images.map(img => this.distanceFunc(image.features,
img.features));
        const median = this.median(distances);
        node.radius = median;
```

```
        const insideImages = images.filter((img, i) => distances[i] <= median);
        const outsideImages = images.filter((img, i) => distances[i] > median);

        node.inside = this.buildTree(insideImages);
        node.outside = this.buildTree(outsideImages);

        return node;
    }

    median(values) {
        values.sort((a, b) => a - b);
        const mid = Math.floor(values.length / 2);
        return values[mid];
    }

    search(image, maxResults, node = this.root, neighbors = []) {
        if (!node) return neighbors;

        const dist = this.distanceFunc(image.features, node.image.features);
        if (neighbors.length < maxResults || dist < neighbors[0].distance) {
            neighbors.push({ image: node.image, distance: dist });
            neighbors.sort((a, b) => b.distance - a.distance);
            if (neighbors.length > maxResults) neighbors.shift();
        }

        const checkInsideFirst = dist < node.radius;

        if (checkInsideFirst) {
            this.search(image, maxResults, node.inside, neighbors);
            if (neighbors.length < maxResults || Math.abs(node.radius - dist) <
neighbors[0].distance) {
                this.search(image, maxResults, node.outside, neighbors);
            }
        } else {
            this.search(image, maxResults, node.outside, neighbors);
            if (neighbors.length < maxResults || Math.abs(node.radius - dist) <
neighbors[0].distance) {
                this.search(image, maxResults, node.inside, neighbors);
            }
        }

        return neighbors;
    }
}
```

4. Usar el Árbol VP para Encontrar Imágenes Similares

Construimos el Árbol VP con las imágenes disponibles y realizamos una búsqueda de las imágenes más similares a una imagen de consulta.

```
const tree = new VPTree(images, euclideanDistance);

const queryImage = { id: 6, features: [129, 63, 65, 0.55, 0.75] };
const maxResults = 2;

const similarImages = tree.search(queryImage, maxResults);
console.log('Images similar to query:', similarImages);
```

Explicación del Ejemplo Práctico

12. **Características de las Imágenes:** Las imágenes se representan como vectores de características en un espacio métrico.
13. **Función de Distancia:** La distancia Euclidiana mide la similitud entre las características de las imágenes.
14. **Árbol VP:** Se construye un árbol VP que organiza las imágenes en base a distancias continuas.
15. **Búsqueda de Similitud:** El Árbol VP se utiliza para encontrar las imágenes más similares a una imagen de consulta.