SEMANA 15 (julio 08, 10 ,12)

FUNCIONES CONTÍNUAS

Espacios Métricos

Un espacio métrico es un conjunto de elementos con una métrica (o función de distancia) que define la distancia entre cualquier par de elementos en el conjunto.

Ejemplo:

Un conjunto de puntos en un plano 2D con la distancia euclidiana como métrica.

```
function euclideanDistance(point1, point2) {
    return Math.sqrt(point1.reduce((sum, value, index) => sum + (value -
point2[index]) ** 2, 0));
}

const point1 = [1, 2];
const point2 = [4, 6];
console.log(euclideanDistance(point1, point2));  // Output: 5
```

Funciones Continuas

En el contexto de espacios métricos, una función continua asegura que pequeños cambios en el valor de entrada resulten en pequeños cambios en el valor de salida, preservando la proximidad entre los puntos. Una función continua puede referirse a funciones que miden distancias o similitudes entre elementos de un espacio métrico de una manera suave y sin saltos bruscos.

Ejemplo:

Una función que calcula la distancia euclidiana entre dos puntos es continua.

Funciones Continuas en Espacios Métricos

En un espacio métrico, una función continua es aquella que mantiene la proximidad entre puntos. Esto es crucial para garantizar que las relaciones de similitud o distancia entre los puntos se preserven de manera consistente.

Definición Formal

Dada una función $f: X \to Y$ entre dos espacios métricos $(X,\ d_X)$ y $(Y,\ d_Y)$, la función f es continua si para todo $\varepsilon > 0$, existe un $\delta > 0$ tal que si $d_X(x, x') < \delta$, entonces $d_Y\big(f(x), f(x')\big) < \varepsilon$.

En términos más simples, pequeños cambios en el valor de entrada x resultan en pequeños cambios en el valor de salida $f(x)$.

Árboles de Vantage Point (VP Trees)

Un VP tree es una estructura de datos que permite realizar búsquedas de proximidad eficientes en espacios métricos. Se construye eligiendo un punto de referencia (vantage point) y particionando el espacio en función de las distancias a este punto.

Ejemplo:

Un VP tree que organiza puntos en un plano 2D.

```
class VPTree {
    constructor(points) {
        this.root = this.buildTree(points);
    }

    buildTree(points) {
        if (points.length === 0) return null;
        const vp = points[Math.floor(Math.random() * points.length)];
        const distances = points.map(point => euclideanDistance(point, vp));
        const median = this.median(distances);
        const leftPoints = points.filter((point, index) => distances[index] <
median);
        const rightPoints = points.filter((point, index) => distances[index] >=
median);
        return {
            vp,
            median,
            left: this.buildTree(leftPoints),
            right: this.buildTree(rightPoints)
        };
    }

    median(values) {
        values.sort((a, b) => a - b);
        const mid = Math.floor(values.length / 2);
        return values.length % 2 === 0 ? (values[mid - 1] + values[mid]) / 2 :
values[mid];
    }
}
```

Multi-VP Trees

Un multi-VP tree es una extensión del VP tree que utiliza múltiples puntos de referencia en cada nodo, permitiendo una partición del espacio más efectiva y mejorando la eficiencia de las búsquedas en espacios de alta dimensionalidad.

Ejemplo:

Construcción de un multi-VP tree con dos puntos de referencia por nodo.

```
 class MultiVPTree {
   constructor(points, numReferences = 2, leafSize = 10) {
      this.numReferences = numReferences;
      this.leafSize = leafSize;
      this.root = this.buildTree(points);
   }

   buildTree(points) {
      if (points.length <= this.leafSize) {
         return { points };
      }

      const references = [];
      for (let i = 0; i < this.numReferences; i++) {
         references.push(points[Math.floor(Math.random() * points.length)]);
      }

      const distances = points.map(point => references.map(ref =>
euclideanDistance(point, ref)));
      const medianDistances = references.map((_, i) => this.median(distances.map(d
=> d[i])));
      const leftPoints = points.filter((_, idx) => distances[idx].every((d, i) =>
d <= medianDistances[i]));
      const rightPoints = points.filter((_, idx) =>
!leftPoints.includes(points[idx]));

      return {
         references,
         medianDistances,
         left: this.buildTree(leftPoints),
         right: this.buildTree(rightPoints)
      };
   }

   search(query, k = 1) {
      return this._search(this.root, query, k);
   }

   _search(node, query, k) {
      if (node.points) {
         const distances = node.points.map(point => euclideanDistance(query,
point));
```

```
         return node.points.map((point, idx) => [point, distances[idx]]).sort((a,
b) => a[1] - b[1]).slice(0, k);
      }

      const queryDistances = node.references.map(ref => euclideanDistance(query,
ref));
      const exploreLeft = queryDistances.every((d, i) => d <=
node.medianDistances[i]);
      const bestBranch = exploreLeft ? node.left : node.right;
      const otherBranch = exploreLeft ? node.right : node.left;
      let bestResults = this._search(bestBranch, query, k);

      const bestDistances = bestResults.map(result => result[1]);
      if (queryDistances.some((d, i) => d <= node.medianDistances[i] +
Math.max(...bestDistances))) {
         const otherResults = this._search(otherBranch, query, k);
         bestResults = bestResults.concat(otherResults).sort((a, b) => a[1] -
b[1]).slice(0, k);
      }

      return bestResults;
   }

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

   median(values) {
      values.sort((a, b) => a - b);
      const mid = Math.floor(values.length / 2);
      return values.length % 2 === 0 ? (values[mid - 1] + values[mid]) / 2 :
values[mid];
   }
}
```

Selección de Puntos de Referencia

La selección de puntos de referencia es crucial para la eficacia de los multi-VP trees. Los puntos de referencia deben elegirse de manera que maximicen la discriminación entre los datos y optimicen la partición del espacio.

Ejemplo:

Seleccionar puntos de referencia al azar o utilizando una heurística específica.

Partición del Espacio

La partición del espacio se realiza en función de las distancias a los puntos de referencia. En un multi-VP tree, esto implica calcular las distancias de todos los puntos a múltiples puntos de referencia y dividir el espacio en subconjuntos basados en estas distancias.

Ejemplo:

Partición del espacio utilizando las distancias calculadas.

Búsqueda en el Multi-VP Tree

La búsqueda en un multi-VP tree implica navegar por el árbol y utilizar las distancias a los puntos de referencia para decidir qué ramas explorar. La poda de ramas ineficaces es crucial para la eficiencia de la búsqueda.

Ejemplo:

Implementación de una función de búsqueda en el multi-VP tree.

```
// Continuación del código de MultiVPTree para la función de búsqueda

const points = [
  [0.1, 0.2],
  [0.3, 0.4],
  [0.5, 0.6],
  [0.7, 0.8],
  [0.9, 1.0],
  [1.1, 1.2]
];  // Ejemplo de puntos de características de imágenes

const tree = new MultiVPTree(points);
const query = [0.2, 0.3];
const results = tree.search(query, 3);

results.forEach(result => {
  console.log(` Punto: ${result[0]}, Distancia: ${result[1]}` );
});
```

Análisis de Rendimiento

Es importante analizar el rendimiento de los multi-VP trees en términos de tiempo de construcción, tiempo de búsqueda y eficiencia en comparación con otras estructuras de datos, como k-d trees.

Ejemplo:

Comparar el tiempo de búsqueda de un multi-VP tree con un k-d tree utilizando un conjunto de datos de prueba.

Relevancia en Métodos de Acceso Métrico

1. Búsqueda de Proximidad:

Las funciones continuas son fundamentales en algoritmos de búsqueda de proximidad, donde necesitamos encontrar elementos cercanos a un punto dado en un espacio métrico. La continuidad asegura que puntos cercanos en el espacio original también sean considerados cercanos por la función de distancia.

2. Eficiencia de Búsqueda:

Estructuras de datos como los VP trees (Vantage-Point trees) y los Multi-VP trees se benefician de funciones continuas para particionar el espacio de datos de manera eficiente. Estas estructuras de datos utilizan la continuidad de la función de distancia para determinar cómo dividir el espacio y organizar los datos para búsquedas rápidas.

EJEMPLO PRÁCTICO

Supongamos que estamos trabajando con un conjunto de datos de imágenes donde cada imagen está representada por un vector de características. Queremos construir una estructura de datos que nos permita buscar imágenes similares de manera eficiente.

Construcción del Multi-VP Tree

1. **Representación de Imágenes como Vectores de Características:** Cada imagen se representa como un vector de características en un espacio métrico.
2. **Función de Distancia:** Utilizamos una función continua, como la distancia euclidiana, para medir la similitud entre dos vectores de características.

Implementación en JavaScript

A continuación, se presenta una implementación simplificada de un Multi-VP Tree en JavaScript utilizando una función continua para medir distancias.

```
class MultiVPTree {
    constructor(points, numReferences = 2, leafSize = 10) {
        this.numReferences = numReferences;
        this.leafSize = leafSize;
        this.root = this.buildTree(points);
    }

    // Función para construir el árbol
    buildTree(points) {
        if (points.length <= this.leafSize) {
            return { points: points };
```

```
        }

        // Seleccionar puntos de referencia aleatoriamente
        const references = [];
        for (let i = 0; i < this.numReferences; i++) {
            const index = Math.floor(Math.random() * points.length);
            references.push(points[index]);
        }

        // Calcular distancias a los puntos de referencia
        const distances = points.map(point => references.map(ref =>
this.distance(point, ref)));

        // Particionar puntos en subconjuntos
        const medianDistances = references.map((_, i) =>
this.median(distances.map(d => d[i])));
        const leftPoints = points.filter((_, idx) => distances[idx].every((d, i)
=> d <= medianDistances[i]));
        const rightPoints = points.filter((_, idx) =>
!leftPoints.includes(points[idx]));

        // Crear nodo actual
        return {
            references: references,
            medianDistances: medianDistances,
            left: this.buildTree(leftPoints),
            right: this.buildTree(rightPoints)
        };
    }

    // Función para buscar imágenes similares
    search(query, k = 1) {
        return this._search(this.root, query, k);
    }

    _search(node, query, k) {
        if (node.points) {
            // Caso base: nodo hoja
            const distances = node.points.map(point => this.distance(query,
point));
            return node.points.map((point, idx) => [point,
distances[idx]]).sort((a, b) => a[1] - b[1]).slice(0, k);
        }

        // Calcular distancias a los puntos de referencia
        const queryDistances = node.references.map(ref => this.distance(query,
ref));

        // Determinar qué rama explorar
```

```
        const exploreLeft = queryDistances.every((d, i) => d <=
node.medianDistances[i]);
        const bestBranch = exploreLeft ? node.left : node.right;
        const otherBranch = exploreLeft ? node.right : node.left;

        // Explorar la rama más prometedora
        let bestResults = this._search(bestBranch, query, k);

        // Posiblemente explorar la otra rama
        const bestDistances = bestResults.map(result => result[1]);
        if (queryDistances.some((d, i) => d <= node.medianDistances[i] +
Math.max(...bestDistances))) {
            const otherResults = this._search(otherBranch, query, k);
            bestResults = bestResults.concat(otherResults).sort((a, b) => a[1] -
b[1]).slice(0, k);
        }

        return bestResults;
    }

    // Función para calcular la distancia euclidiana entre dos puntos
    distance(a, b) {
        return Math.sqrt(a.reduce((sum, ai, i) => sum + (ai - b[i]) ** 2, 0));
    }

    // Función para calcular la mediana de un conjunto de valores
    median(values) {
        values.sort((a, b) => a - b);
        const mid = Math.floor(values.length / 2);
        return values.length % 2 === 0 ? (values[mid - 1] + values[mid]) / 2 :
values[mid];
    }
}

// Ejemplo de uso
const points = [
    [0.1, 0.2],
    [0.3, 0.4],
    [0.5, 0.6],
    [0.7, 0.8],
    [0.9, 1.0],
    [1.1, 1.2]
];  // Ejemplo de puntos de características de imágenes

const tree = new MultiVPTree(points);
const query = [0.2, 0.3];
const results = tree.search(query, 3);

results.forEach(result => {
```

```
    console.log(`Punto: ${result[0]}, Distancia: ${result[1]}`);
});
```

Explicación del Código

1. **Representación de Imágenes como Vectores de Características:**
    - Las imágenes se representan como puntos en un espacio de características (vectores).
2. **Función de Distancia Continua:**
    - La distancia euclidiana es una función continua utilizada para medir la similitud entre los puntos (vectores de características).
3. **Construcción del Multi-VP Tree:**
    - La función `buildTree` construye recursivamente el árbol seleccionando puntos de referencia aleatorios y particionando los puntos en subconjuntos basados en las distancias a estos puntos de referencia.
4. **Búsqueda de Imágenes Similares:**
    - La función `search` realiza una búsqueda de las imágenes más similares a una imagen de consulta utilizando las distancias euclidianas continuas.

Conclusión

Las funciones continuas y métodos de acceso métrico son esenciales para preservar la proximidad entre puntos y asegurar búsquedas eficientes. En el ejemplo práctico de un multi-VP tree en JavaScript, la continuidad de la función de distancia (distancia euclidiana) juega un papel crucial en la partición del espacio de datos y en la eficiencia de las búsquedas.

PROBLEMA PRÁCTICO: MULTI-VANTAGE-POINT TREES

Supongamos que estamos desarrollando una aplicación para encontrar imágenes similares en una base de datos de imágenes utilizando características visuales.

Problema

Queremos construir una base de datos que permita buscar imágenes similares basadas en sus características visuales. Cada imagen se representa como un vector en un

espacio de características de alta dimensionalidad. Necesitamos una estructura de datos eficiente para realizar búsquedas rápidas de imágenes similares.

Solución

Utilizaremos un multi-VP tree para indexar y buscar las imágenes. La estructura del multi-VP tree nos permitirá dividir el espacio de características en regiones manejables y realizar búsquedas eficientes.

Pasos para Implementar el Multi-VP Tree en JavaScript

1. **Representar Imágenes como Vectores de Características**

   Cada imagen se representa como un vector de características. Por simplicidad, asumiremos que ya tenemos estos vectores.

2. **Construir el Multi-VP Tree**

   Crearemos un multi-VP tree utilizando múltiples puntos de referencia en cada nodo para particionar el espacio de características.

3. **Buscar Imágenes Similares**

   Implementaremos una función para buscar las imágenes más similares a una imagen de consulta utilizando el multi-VP tree.

Implementación en JavaScript

A continuación, se presenta una implementación simplificada en JavaScript:

```
class MultiVPTree {
   constructor(points, numReferences = 2, leafSize = 10) {
      this.numReferences = numReferences;
      this.leafSize = leafSize;
      this.root = this.buildTree(points);
   }

   // Función para construir el árbol
   buildTree(points) {
      if (points.length <= this.leafSize) {
         return { points: points };
      }

      // Seleccionar puntos de referencia aleatoriamente
      const references = [];
      for (let i = 0; i < this.numReferences; i++) {
         const index = Math.floor(Math.random() * points.length);
         references.push(points[index]);
      }
```

```
      // Calcular distancias a los puntos de referencia
      const distances = points.map(point => references.map(ref =>
this.distance(point, ref)));

      // Particionar puntos en subconjuntos
      const medianDistances = references.map((_, i) => this.median(distances.map(d
=> d[i])));
      const leftPoints = points.filter((_, idx) => distances[idx].every((d, i) =>
d <= medianDistances[i]));
      const rightPoints = points.filter((_, idx) =>
!leftPoints.includes(points[idx]));

      // Crear nodo actual
      return {
         references: references,
         medianDistances: medianDistances,
         left: this.buildTree(leftPoints),
         right: this.buildTree(rightPoints)
      };
   }

   // Función para buscar imágenes similares
   search(query, k = 1) {
      return this._search(this.root, query, k);
   }

   _search(node, query, k) {
      if (node.points) {
         // Caso base: nodo hoja
         const distances = node.points.map(point => this.distance(query, point));
         return node.points.map((point, idx) => [point, distances[idx]]).sort((a,
b) => a[1] - b[1]).slice(0, k);
      }

      // Calcular distancias a los puntos de referencia
      const queryDistances = node.references.map(ref => this.distance(query,
ref));

      // Determinar qué rama explorar
      const exploreLeft = queryDistances.every((d, i) => d <=
node.medianDistances[i]);
      const bestBranch = exploreLeft ? node.left : node.right;
      const otherBranch = exploreLeft ? node.right : node.left;

      // Explorar la rama más prometedora
      let bestResults = this._search(bestBranch, query, k);

      // Posiblemente explorar la otra rama
      const bestDistances = bestResults.map(result => result[1]);
```

```
      if (queryDistances.some((d, i) => d <= node.medianDistances[i] +
Math.max(...bestDistances))) {
         const otherResults = this._search(otherBranch, query, k);
         bestResults = bestResults.concat(otherResults).sort((a, b) => a[1] -
b[1]).slice(0, k);
      }

      return bestResults;
   }

   // Función para calcular la distancia euclidiana entre dos puntos
   distance(a, b) {
      return Math.sqrt(a.reduce((sum, ai, i) => sum + (ai - b[i]) ** 2, 0));
   }

   // Función para calcular la mediana de un conjunto de valores
   median(values) {
      values.sort((a, b) => a - b);
      const mid = Math.floor(values.length / 2);
      return values.length % 2 === 0 ? (values[mid - 1] + values[mid]) / 2 :
values[mid];
   }
}

// Ejemplo de uso
const points = [
   [0.1, 0.2],
   [0.3, 0.4],
   [0.5, 0.6],
   [0.7, 0.8],
   [0.9, 1.0],
   [1.1, 1.2]
];  // Ejemplo de puntos de características de imágenes

const tree = new MultiVPTree(points);
const query = [0.2, 0.3];
const results = tree.search(query, 3);

results.forEach(result => {
   console.log(`Punto: ${result[0]}, Distancia: ${result[1]}`);
});
```

Explicación del Código

3. Representación de Imágenes como Vectores de Características

- Las imágenes se representan como puntos en un espacio de características, en este caso, puntos en un espacio 2D.

4. Construcción del Multi-VP Tree

- La función `buildTree` construye recursivamente el árbol seleccionando puntos de referencia aleatorios y particionando los puntos basados en las distancias a estos puntos de referencia.

5. Búsqueda de Imágenes Similares

- La función `search` realiza una búsqueda de las imágenes más similares a una imagen de consulta utilizando las distancias euclidianas.

6. Distancia Euclidiana y Mediana

- La función `distance` calcula la distancia euclidiana entre dos puntos.
- La función `median` calcula la mediana de un conjunto de valores, lo que se utiliza para particionar los puntos en subconjuntos.

Conclusión

Este ejemplo muestra cómo se puede aplicar un multi-VP tree para resolver el problema de buscar imágenes similares en una base de datos. El enfoque basado en problemas nos permite ver cómo las estructuras de datos y las funciones continuas se pueden utilizar en aplicaciones del mundo real para mejorar la eficiencia y efectividad de nuestras soluciones.