Universidad Nacional del Altiplano

FACULTAD DE INGENIERÍA MECÁNCA ELÉCTRICA, ELECTRÓNCA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

RESUMEN DE AVANCE DEL CURSO ESTRUCTURAS DE DATOS AVANZADAS

Estudiante: Widvar Gustavo Condori Coaquira

Docente: Ing. Fredy Collanqui Martinez

Semestre: VI **Grupo:** C

Contenido

1. INTRODUCCIÓN 2
2. KD- Trees 2
3. Representación de Datos Multidimensionales: 3
4. Aplicaciones de los KD-Trees: 4
5. Algoritmo para la Construcción de un KD-Tree: 4
6. Consideraciones en la Construcción de KD-Trees: 4
7. Introducción al Algoritmo Nearest Neighbour: 5
8. Simulación: 6
Visualización del Proceso 9

1. INTRODUCCIÓN

MÉTODOS DE ACCESO ESPACIAL

- **Definición:** Los métodos de acceso espacial son técnicas especializadas para gestionar y consultar datos que tienen una dimensión espacial, como coordenadas geográficas. Estos métodos se utilizan principalmente en bases de datos espaciales donde es necesario almacenar, buscar y analizar datos que están ubicados en un espacio físico o conceptual.

- **Importancia:** Estos métodos son cruciales porque permiten realizar búsquedas y análisis de manera eficiente sobre datos espaciales. Esto es esencial en aplicaciones como los sistemas de información geográfica (GIS), donde se necesita manejar grandes volúmenes de datos geoespaciales, bases de datos espaciales para almacenar y recuperar información geográfica, y en la ciencia de datos para analizar patrones y relaciones espaciales.

2. KD- Trees

- **Definición:** Un KD-Tree (K-Dimensional Tree) es una estructura de datos que organiza puntos en un espacio de K dimensiones. Es similar a un árbol binario, pero en lugar de dividir datos en una única dimensión, los divide en múltiples dimensiones, alternando entre ellas en cada nivel del árbol. Esto facilita la búsqueda de puntos en un espacio multidimensional de manera eficiente.

- **Estructura:** En un KD-Tree, cada nodo representa un punto en el espacio y divide ese espacio en dos mitades. La dimensión en la que se divide alterna en cada nivel del árbol.

Por ejemplo, en un espacio bidimensional, el primer nivel puede dividir el espacio en función de la coordenada x y el segundo nivel en función de la coordenada y, y así sucesivamente.

- **Construcción:**

Pasos:

- **Seleccionar la dimensión para dividir el espacio:** Se elige una de las dimensiones (x, y, z, etc.) para dividir los puntos. Esta selección suele alternar en cada nivel del árbol.
- **Ordenar los puntos en función de la dimensión seleccionada:** Los puntos se ordenan según sus coordenadas en la dimensión seleccionada.
- **Elegir el punto mediano y establecerlo como nodo raíz:** El punto que queda en el medio de los puntos ordenados se convierte en el nodo actual (raíz del subárbol).
- **Recursivamente aplicar el mismo proceso a las dos mitades restantes:** El proceso se repite para los subconjuntos de puntos a la izquierda y a la derecha del punto mediano, creando subárboles.

Ejemplo: En un espacio bidimensional (K=2), el primer nivel del árbol podría dividir el espacio según la coordenada x. Los puntos se ordenan por sus valores de x y el punto mediano se convierte en el nodo raíz. El siguiente nivel divide el espacio según la coordenada y, y así sucesivamente, alternando las dimensiones en cada nivel del árbol.

3. Representación de Datos Multidimensionales:

Descripción: Los KD-Trees permiten representar y organizar datos en múltiples dimensiones, lo cual es esencial para realizar consultas eficientes. Por ejemplo, se pueden usar para encontrar rápidamente el punto más cercano a un punto de consulta (nearest neighbor search) o para buscar todos los puntos dentro de un rango específico.

Ventajas:

- **Eficiencia:** Los KD-Trees son especialmente eficientes para realizar consultas en espacios multidimensionales, reduciendo significativamente el tiempo de búsqueda comparado con métodos de búsqueda exhaustivos.
- **Flexibilidad:** Pueden ser utilizados en diversas aplicaciones que requieren manejar datos espaciales, desde gráficos computacionales, donde ayudan a gestionar objetos en un espacio tridimensional, hasta bases de datos espaciales, donde facilitan la consulta y organización de grandes volúmenes de datos geoespaciales.

4. Aplicaciones de los KD-Trees:

- **Sistemas de Información Geográfica (GIS):** Utilizados para organizar y consultar grandes conjuntos de datos geoespaciales, permitiendo la búsqueda rápida de ubicaciones y características espaciales.
- **Ciencias de Datos y Machine Learning:** Aplicados en la búsqueda de vecinos más cercanos, clustering (agrupamiento de datos) y otros algoritmos que requieren manejo de datos multidimensionales.
- **Computación Gráfica:** Empleados en la gestión de objetos en un espacio tridimensional para mejorar la eficiencia de renderizado y detección de colisiones.

5. Algoritmo para la Construcción de un KD-Tree:

Pasos del Algoritmo:

1. **Selección de la Dimensión:** Se selecciona una dimensión de las K disponibles para dividir los puntos. Esta selección alterna en cada nivel del árbol.
2. **Ordenación de Puntos:** Los puntos se ordenan según la dimensión seleccionada.
3. **Elección del Punto Mediano:** El punto mediano se elige como el nodo actual del árbol, dividiendo el conjunto de puntos en dos subconjuntos.
4. **División Recursiva:** El proceso se aplica recursivamente a los subconjuntos izquierdo y derecho hasta que todos los puntos se hayan insertado en el árbol.

- **Pseudocódigo del Algoritmo:**

```
function construirKDTree(puntos, profundidad):
    si puntos está vacío:
        return None
    eje = profundidad mod K
    ordenar puntos por coordenada eje
    mediano = tamaño(puntos) // 2
    nodo = nuevo Nodo(puntos[mediano])
    nodo.izquierdo = construirKDTree(puntos[0:mediano], profundidad
+ 1)
    nodo.derecho = construirKDTree(puntos[mediano + 1:],
profundidad + 1)
    return nodo
```

6. Consideraciones en la Construcción de KD-Trees:

Consideraciones en la Construcción de KD-Trees:

- **Equilibrio del Árbol:** Un KD-Tree equilibrado tiene una profundidad logarítmica en función del número de puntos, lo que garantiza una búsqueda eficiente.

- **Puntos Duplicados:** Manejo de puntos duplicados y cómo afectan la estructura del árbol. Se debe decidir si se permiten duplicados y cómo se representan.
- **Dimensiones Desiguales:** Tratamiento de dimensiones con diferentes escalas o rangos, asegurando que ninguna dimensión domine las divisiones del árbol.

7. Introducción al Algoritmo Nearest Neighbour:

Definición: El algoritmo Nearest Neighbour (NN) es una técnica utilizada para encontrar el punto más cercano a un punto de consulta en un espacio multidimensional. Es una de las aplicaciones más comunes de los KD-Trees y se utiliza ampliamente en diferentes áreas de la informática y el análisis de datos.

Importancia: Es fundamental en aplicaciones como la búsqueda de información, donde se necesita encontrar datos similares; reconocimiento de patrones, como en el reconocimiento facial; y en el aprendizaje automático, donde es esencial para algoritmos de clasificación y regresión.

2. Funcionamiento del Algoritmo Nearest Neighbour en KD-Trees:

Búsqueda Recursiva:

- La búsqueda comienza desde la raíz del KD-Tree.
- En cada nodo, se compara la distancia del punto de consulta con el punto del nodo actual. Dependiendo de esta comparación, se decide si explorar el subárbol izquierdo o derecho.
- Este proceso se repite recursivamente, moviéndose a través de los nodos del árbol.

Podas y Optimización:

- Se utilizan estrategias de poda para descartar subárboles completos que no pueden contener el punto más cercano al punto de consulta. Esto se hace comparando la distancia mínima encontrada hasta el momento con la distancia entre el punto de consulta y los límites de los subárboles.
- Esto reduce significativamente el número de comparaciones necesarias, haciendo la búsqueda mucho más eficiente.

Aplicaciones del Algoritmo Nearest Neighbour:

- **Reconocimiento de Patrones:** Utilizado en sistemas de recomendación, reconocimiento facial y clasificación de imágenes, donde es crucial identificar rápidamente el patrón más similar.
- **Análisis de Datos:** Aplicado en la búsqueda de registros similares en bases de datos y en la detección de anomalías, facilitando el análisis de grandes volúmenes de datos.
- **Robótica y Navegación:** Utilizado en la planificación de rutas y la evitación de obstáculos, ayudando a los robots a navegar de manera segura y eficiente.

- **Machine Learning:** Fundamental en métodos como K-Nearest Neighbours (K-NN) para la clasificación y regresión, donde se determina la clase o valor de un nuevo dato basado en sus vecinos más cercanos.

3. Algoritmo de Búsqueda Nearest Neighbour:

- **Pseudocódigo del Algoritmo:**

```
plaintext
Copiar código
function buscarNN(nodo, puntoConsulta, profundidad, mejorNodo,
mejorDistancia):
    si nodo es None:
        return mejorNodo, mejorDistancia
    eje = profundidad mod K
    distanciaActual = distanciaEuclidiana(nodo.punto,
puntoConsulta)
    if distanciaActual < mejorDistancia:
        mejorNodo = nodo
        mejorDistancia = distanciaActual
    if puntoConsulta[eje] < nodo.punto[eje]:
        mejorNodo, mejorDistancia = buscarNN(nodo.izquierdo,
puntoConsulta, profundidad + 1, mejorNodo, mejorDistancia)
        if abs(puntoConsulta[eje] - nodo.punto[eje]) <
mejorDistancia:
            mejorNodo, mejorDistancia = buscarNN(nodo.derecho,
puntoConsulta, profundidad + 1, mejorNodo, mejorDistancia)
    else:
        mejorNodo, mejorDistancia = buscarNN(nodo.derecho,
puntoConsulta, profundidad + 1, mejorNodo, mejorDistancia)
        if abs(puntoConsulta[eje] - nodo.punto[eje]) <
mejorDistancia:
            mejorNodo, mejorDistancia = buscarNN(nodo.izquierdo,
puntoConsulta, profundidad + 1, mejorNodo, mejorDistancia)
    return mejorNodo, mejorDistancia
```

8. Simulación:

- **Construcción Paso a Paso:**

Se implementó un árbol KD en Python y visualizarlo junto con un conjunto de puntos en un espacio bidimensional, simulando el funcionamiento:

Componentes del Código:

1. **Clase Node (Nodo):**
   - La clase `Node` representa un nodo en el árbol KD.
   - Cada nodo tiene un punto asociado en el espacio bidimensional, una dimensión de división y referencias a los nodos hijos izquierdo y derecho.
2. **Función `build_kd_tree`:**

- Esta función construye el árbol KD recursivamente a partir de un conjunto de puntos dados.
- Divide el conjunto de puntos en cada nivel del árbol utilizando la mediana en la dimensión actual como punto del nodo.

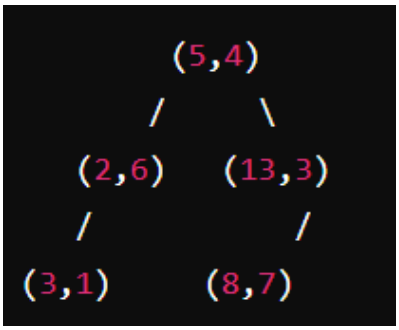
3. **Función `plot_kd_tree`:**
   - Esta función visualiza el árbol KD utilizando la biblioteca matplotlib.
   - Recorre recursivamente el árbol y dibuja líneas verticales u horizontales para representar las divisiones en el espacio.
4. **Función `plot_points`:**
   - Esta función visualiza los puntos en el espacio bidimensional utilizando la biblioteca matplotlib.
5. **Ejemplo de Uso:**
   - Se proporciona un ejemplo de uso que crea un árbol KD a partir de un conjunto de puntos dados y visualiza el árbol junto con los puntos.

EJEMPLIFICACIÓN:

Paso 1: Construcción del KD-Tree

Dividir los puntos en dimensiones alternadas y construir el árbol:

1. **Primer nivel (raíz):**
   - Dimensión: x
   - Puntos: (5,4), (2,6), (13,3), (8,7), (3,1)
   - Ordenar por x: [(2,6), (3,1), (5,4), (8,7), (13,3)]
   - Punto mediano: (5,4)
   - El nodo raíz es (5,4).
2. **Segundo nivel (hijos del nodo raíz):**
   - Dimensión: y
   - Izquierda de (5,4): [(2,6), (3,1)]
     - Ordenar por y: [(3,1), (2,6)]
     - Punto mediano: (2,6)
     - El hijo izquierdo de (5,4) es (2,6).
     - Subdividir más:
       - (3,1) es el hijo izquierdo de (2,6).
   - Derecha de (5,4): [(8,7), (13,3)]
     - Ordenar por y: [(13,3), (8,7)]
     - Punto mediano: (13,3)
     - El hijo derecho de (5,4) es (13,3).
     - Subdividir más:
       - (8,7) es el hijo izquierdo de (13,3).



Buscar el vecino más cercano al punto objetivo (9,4):

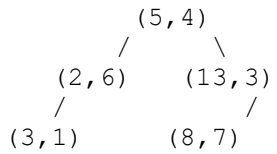
1. **Inicio en la raíz (5,4):**
   - Distancia euclidiana: sqrt((9-5)^2 + (4-4)^2) = 4
   - Mejor distancia hasta ahora: 4
   - Mejor punto hasta ahora: (5,4)
   - Comparar x (9 vs. 5): Ir a la derecha.
2. **Nodo (13,3):**
   - Distancia euclidiana: sqrt((9-13)^2 + (4-3)^2) = sqrt(16 + 1) = sqrt(17) ≈ 4.12
   - Mejor distancia sigue siendo: 4 (del nodo (5,4))
   - Mejor punto sigue siendo: (5,4)
   - Comparar y (4 vs. 3): Ir a la izquierda.
3. **Nodo (8,7):**
   - Distancia euclidiana: sqrt((9-8)^2 + (4-7)^2) = sqrt(1 + 9) = sqrt(10) ≈ 3.16
   - Mejor distancia actualizada: 3.16
   - Mejor punto actualizado: (8,7)
   - No hay más nodos en este subárbol.
4. **Volver a (13,3):**
   - No es necesario explorar más subárboles de (13,3) porque las distancias ya verificadas en el subárbol izquierdo son menores que la distancia actualizada (3.16).
5. **Volver a (5,4):**
   - Evaluar si necesitamos explorar el subárbol izquierdo de (5,4).
   - Comparar la distancia del límite del subárbol izquierdo (distancia en la coordenada x = 5 a la línea x = 9) que es 4. La mejor distancia encontrada hasta ahora (3.16) es menor, por lo que no es necesario explorar el subárbol izquierdo de (5,4).

Resultado Final

El punto más cercano al punto objetivo (9,4) es (8,7) con una distancia de aproximadamente 3.16.

Visualización del Proceso

1. **KD-Tree:**



2. **Búsqueda del Vecino Más Cercano (9,4):**
   - Raíz (5,4): Mejor distancia = 4
   - (13,3): Mejor distancia sigue siendo 4
   - (8,7): Mejor distancia actualizada a 3.16

Conclusión: El vecino más cercano al punto (9,4) es (8,7).

Espero que esta explicación detallada te haya ayudado a entender cómo funciona el KD-Tree para buscar el vecino más cercano.

CODIGO:

```python
import numpy as np
import matplotlib.pyplot as plt

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

def construir_arbol_kd(puntos, profundidad=0):
    if len(puntos) == 0:
        return None

    k = len(puntos[0])
    split_dim = profundidad % k

    puntos.sort(key=lambda x: x[split_dim])
    indice_mediano = len(puntos) // 2
    punto_mediano = puntos[indice_mediano]

    puntos_izquierda = puntos[:indice_mediano]
    puntos_derecha = puntos[indice_mediano + 1:]

    return Nodo(
        punto=punto_mediano,
        split_dim=split_dim,
        izquierda=construir_arbol_kd(puntos_izquierda, profundidad + 1),
        derecha=construir_arbol_kd(puntos_derecha, profundidad + 1)
```

```
    )

def distancia_cuadrada(punto1, punto2):
    return np.sum((np.array(punto1) - np.array(punto2)) ** 2)

def nearest_neighbor_search(arbol, punto_objetivo, profundidad=0,
mejor=None):
    if arbol is None:
        return mejor

    k = len(punto_objetivo)
    split_dim = profundidad % k

    next_branch = None
    opposite_branch = None

    if punto_objetivo[split_dim] < arbol.punto[split_dim]:
        next_branch = arbol.izquierda
        opposite_branch = arbol.derecha
    else:
        next_branch = arbol.derecha
        opposite_branch = arbol.izquierda

    mejor = nearest_neighbor_search(next_branch, punto_objetivo,
profundidad + 1, mejor)

    if mejor is None or distancia_cuadrada(punto_objetivo, arbol.punto) <
distancia_cuadrada(punto_objetivo, mejor):
        mejor = arbol.punto

    if (punto_objetivo[split_dim] - arbol.punto[split_dim]) ** 2 <
distancia_cuadrada(punto_objetivo, mejor):
        mejor = nearest_neighbor_search(opposite_branch, punto_objetivo,
profundidad + 1, mejor)

    return mejor

def graficar_arbol_kd(arbol, punto_minimo, punto_maximo, ax=None):
    if ax is None:
        fig, ax = plt.subplots()

    if arbol is None:
        return

    k = len(punto_minimo)
    split_dim = arbol.split_dim

    if split_dim == 0:
        ax.plot([arbol.punto[0], arbol.punto[0]], [punto_minimo[1],
punto_maximo[1]], color='black')
    else:
        ax.plot([punto_minimo[0], punto_maximo[0]], [arbol.punto[1],
arbol.punto[1]], color='black')

    graficar_arbol_kd(arbol.izquierda, punto_minimo, arbol.punto, ax)
    graficar_arbol_kd(arbol.derecha, arbol.punto, punto_maximo, ax)
```

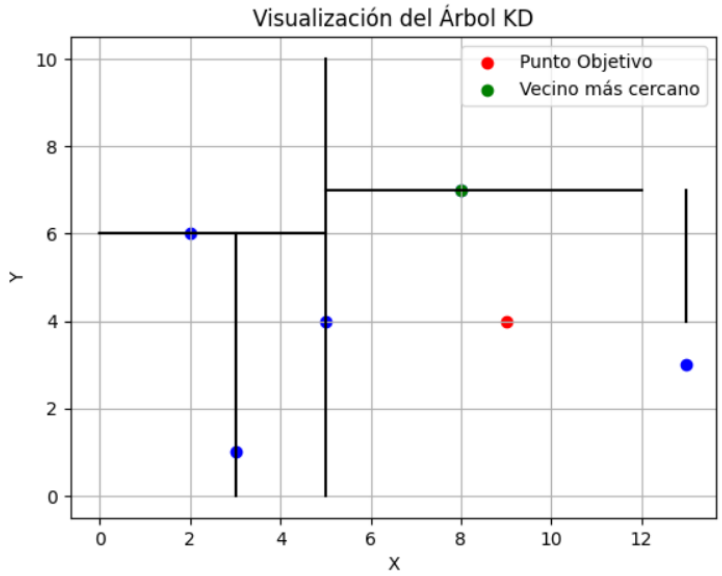
```python
def graficar_puntos(puntos, ax=None):
    if ax is None:
        fig, ax = plt.subplots()

    x = [p[0] for p in puntos]
    y = [p[1] for p in puntos]
    ax.scatter(x, y, color='blue')

# Ejemplo de uso
puntos = [(5,4),(2,6),(13,3),(8,7),(3,1)]
punto_objetivo = (9,4)
arbol_kd = construir_arbol_kd(puntos)

# Visualización
fig, ax = plt.subplots()
graficar_arbol_kd(arbol_kd, (0, 0), (12, 10), ax)
graficar_puntos(puntos, ax)
ax.scatter(punto_objetivo[0], punto_objetivo[1], color='red',
label='Punto Objetivo')
vecino_mas_cercano = nearest_neighbor_search(arbol_kd, punto_objetivo)
ax.scatter(vecino_mas_cercano[0], vecino_mas_cercano[1], color='green',
label='Vecino más cercano')
ax.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Visualización del Árbol KD')
plt.grid(True)
plt.show()
```

VISUALIZACION:



9. ENLACE DEL REPOSITORIO (GITHUB)

https://github.com/gustavowc/EDA.git