Universidad Nacional de San Agustín de Arequipa

Escuela Profesional de Ciencia de la Computación
Estructura de Datos Avanzados

Laboratorio 5: KDTree

Presentado por:
Cahuana Nina, Melany Maria
Zavalaga Orozco, Rushell Vanessa
Muñoz Curi, Giomar Danny

Docente :
Rolando Jesús Cárdenas Talavera

1. Algoritmos Implementados

En este trabajo se implementan dos versiones del algoritmo de búsqueda de vecinos más cercanos (KNN): (i) un enfoque de fuerza bruta y (ii) un enfoque utilizando un KD-Tree. Se analizan los costos computacionales de inserción y búsqueda, y se comparan los tiempos de ejecución para distintos conjuntos de datos y valores del parámetro k.

1.1. KNN Fuerza Bruta

El enfoque de fuerza bruta calcula las distancias entre el punto de consulta y todos los puntos del conjunto de datos.

Algorithm 1 Función `distance`

Require: `coord1`, `coord2`: listas de coordenadas de puntos en un espacio de dimensión `dim`.
Ensure: `dist`: distancia euclidiana entre `coord1` y `coord2`.

```
dist ← 0
for i desde 0 hasta dim −1 do
    dist ← dist + (coord1[i] − coord2[i])^2
end for
return √dist
```

A continuación, se presenta la función de KNN usando fuerza bruta, que imprime los k vecinos más cercanos al punto de consulta.

Algorithm 2 Función `bruteForce`

Require: `nodes`: lista de nodos, `coord`: coordenada del punto de consulta, `k`: número de vecinos a buscar.
Ensure: Vecinos más cercanos al punto de consulta.

```
Crear una cola de prioridad pq (máximo de tamaño k)
for node en nodes do
    bestDist ← distance(node.coord, coord)
    if pq tiene menos de k elementos then
        Insertar (bestDist, node) en pq
    else if bestDist es menor que la distancia máxima en pq then
        Eliminar el nodo con distancia máxima de pq
        Insertar (bestDist, node) en pq
    end if
end for
return pq
```

1.2. KNN usando KD-Tree

El algoritmo KNN utilizando un KD-Tree permite realizar búsquedas eficientes en espacios multidimensionales.

Algorithm 3 Función `searchKNN` (Recursiva)

Require: `coord`: coordenada del punto de consulta, `ind`: índice de la dimensión actual, `node`: nodo actual, `k`: número de vecinos a buscar, `pq`: cola de prioridad.
Ensure: Cola de prioridad `pq` con los k vecinos más cercanos.

```
if node es nulo then
    return pq
end if
dist ← distance(coord, node.coord)
if El tamaño de pq es menor que k then
    Insertar (dist, node) en pq
else if dist es menor que la distancia máxima en pq then
    Eliminar el nodo con distancia máxima de pq
    Insertar (dist, node) en pq
end if
closeBranch ← nodo hijo más cercano
farBranch ← nodo hijo más lejano
if coord[ind] es menor que node.coord[ind] then
    closeBranch ← node.L
    farBranch ← node.R
else
    closeBranch ← node.R
    farBranch ← node.L
end if
pq ← searchKNN(coord, (ind + 1) % dim, closeBranch, k, pq)
if La distancia entre coord[ind] y node.coord[ind] es menor que la distancia máxima en pq o el tamaño de pq es menor que k then
    pq ← searchKNN(coord, (ind + 1) % dim, farBranch, k, pq)
end if
return pq
```

Algorithm 4 Función `searchKNN`

Require: `coord`: coordenada del punto de consulta, `k`: número de vecinos a buscar.
Ensure: Cola de prioridad `pq` con los k vecinos más cercanos.

1: Crear una cola de prioridad `pq`
2: Insertar `(INT_MAX, root)` en `pq`
3: `pq` ← `searchKNN(coord, 0, root, k, pq)`
4: **if** La distancia máxima en `pq` es `INT_MAX` **then**
5: Imprimir "No se encontraron los k vecinos más cercanos"
6: **end if**
7: **return** `pq`

Ejemplo de ejecucion de funcion Print, NearestNeighbour Y KNearestNeighbours con (1,2,3)

```
Printing tree
N A: 6 6 8
|  LN B: 4 4 3
|  |  LN T: 4 1 4
|  |  RN C: 5 5 5
|  |  |  LN K: 1 5 3
|  |  |  |  RN M: 2 7 4
|  |  |  RN G: 3 8 5
|  |  |  |  LN I: 0 5 5
|  |  |  |  RN J: 4 6 6
|  RN D: 9 2 8
|  |  RN E: 7 8 9
|  |  |  LN H: 7 6 1
|  |  |  |  RN N: 9 6 6
|  |  |  |  |  LN O: 7 5 4
|  |  |  |  |  |  LN P: 9 2 2
|  |  |  |  |  RN Q: 8 6 2
|  |  |  |  |  |  RN S: 9 7 6
|  |  |  RN F: 7 5 9
|  |  |  |  LN R: 6 5 9
|  |  |  |  RN L: 9 9 9
Best Node: K with 1 5 3
Best Distance: 3
Node: M with distance: 5.19615
2 7 4
Node: I with distance: 3.74166
0 5 5
Node: B with distance: 3.60555
4 4 3
Node: T with distance: 3.31662
4 1 4
Node: K with distance: 3
1 5 3
```

2. Costo Computacional

2.1. Costo de Inserción en KD-Tree

El costo computacional de la función de inserción en un KD-Tree puede ser analizado considerando diferentes escenarios y características del árbol:

- **Costo Promedio:** En el caso promedio, el KD-Tree se mantiene balanceado, lo que implica que cada nodo tiene aproximadamente el mismo número de puntos en sus subárboles izquierdo y derecho.

  En este caso, la altura del árbol es aproximadamente $O(\log n)$, donde n es el número de puntos en el KD-Tree.

  La inserción de un nuevo punto implica recorrer la altura del árbol para encontrar la posición adecuada, por lo tanto, el costo promedio es $O(\log n)$.

- **Costo en el Peor Caso:** Si los puntos se insertan de forma desbalanceada (por ejemplo, si se insertan puntos ya ordenados), el árbol se convierte en una lista enlazada.

  En este caso, la altura del árbol es $O(n)$ y el costo de inserción también es $O(n)$.

  Este caso se puede mitigar aplicando técnicas de re-balanceo o mediante la inserción aleatoria de puntos.

En resumen, el costo computacional de la inserción en un KD-Tree es:

$$\text{Costo promedio: } O(\log n), \quad \text{Costo en el peor caso: } O(n)$$

2.2. Costo de Búsqueda

El costo computacional de la búsqueda de k-vecinos más cercanos (k-Nearest Neighbors, KNN) varía según el método utilizado. A continuación, se analiza el costo para los métodos de fuerza bruta y utilizando KD-Tree:

- **Fuerza Bruta:**
  - En el enfoque de fuerza bruta, se recorren todos los puntos del conjunto de datos para calcular la distancia a cada punto de consulta.
  - La complejidad es $O(n \times d)$, donde n es el número de puntos y d es la dimensión del espacio. Esto se debe a que se realiza una comparación con cada punto y se calculan d distancias parciales.
  - Este método es ineficiente para conjuntos de datos grandes, ya que la complejidad aumenta linealmente con el número de puntos.

- **KD-Tree:**
  - El KD-Tree permite realizar búsquedas más eficientes dividiendo recursivamente el espacio en regiones, lo que reduce el número de puntos que deben ser evaluados.
  - En el caso promedio, el costo de la búsqueda es $O(\log n + k)$:
    - $O(\log n)$ corresponde al costo de recorrer la altura del árbol para localizar la región de interés.
    - $O(k)$ representa el costo de mantener los k-vecinos más cercanos en una estructura como una cola de prioridad (heap).
  - En el peor caso, el costo puede ser $O(n)$, especialmente si el árbol está desbalanceado o si la consulta cae en una región que abarca muchos puntos, lo que obliga a explorar más nodos.
  - A medida que aumenta la dimensión del espacio (d), el rendimiento del KD-Tree puede degradarse debido a la "maldición de la dimensionalidad", lo que aumenta la probabilidad de tener que explorar ambos subárboles durante la búsqueda.

En resumen, el costo computacional es:

Fuerza Bruta: $O(n \times d)$, KD-Tree (promedio): $O(\log n + k)$, KD-Tree (peor caso): $O(n)$

3. Análisis de Desempeño

Se utilizaron los archivos `testX.csv`, donde X representa el número de puntos en el espacio 3D. Los datos contienen puntos con coordenadas (x, y, z).

Se generaron puntos de consulta aleatorios dentro del mismo rango de coordenadas que los puntos del conjunto de datos para realizar la búsqueda de vecinos.

3.1. Comparación del Tiempo de Ejecución

Se compararon los tiempos de ejecución de la búsqueda KNN usando fuerza bruta y KD-Tree para diferentes valores de k. Los resultados se muestran en la Tabla 1.

Cuadro 1: Comparación del tiempo de búsqueda KNN para fuerza bruta y KD-Tree con k = 10 consultas

Archivo	N° de Puntos	Fuerza Bruta (ms)	KD-Tree (ms)
`test1000.csv`	1000	0.000866	0.000171
`test5000.csv`	5000	0.003641	0.000451
`test10000.csv`	10000	0.00379	0.00028

Aqui vemos los 10 puntos calculados con los diferentes algoritmos:

```
Tree 1:
Time taken by KDTree: 0.000171 microseconds
Node (228, 181, 139, ) with distance: 184.103
Node (183, 182, 175, ) with distance: 169.765
Node (85, 245, 101, ) with distance: 160.814
Node (50, 160, 210, ) with distance: 155.737
Node (44, 230, 64, ) with distance: 149.753
Node (96, 202, 162, ) with distance: 144.395
Node (187, 158, 67, ) with distance: 126.886
Node (41, 177, 2, ) with distance: 125.18
Node (39, 147, 32, ) with distance: 87.647
Node (109, 159, 110, ) with distance: 84.2259

Time taken by Brute Force: 0.000866 microseconds
Node (228, 181, 139, ) with distance: 184.103
Node (183, 182, 175, ) with distance: 169.765
Node (85, 245, 101, ) with distance: 160.814
Node (50, 160, 210, ) with distance: 155.737
Node (44, 230, 64, ) with distance: 149.753
Node (96, 202, 162, ) with distance: 144.395
Node (187, 158, 67, ) with distance: 126.886
Node (41, 177, 2, ) with distance: 125.18
Node (39, 147, 32, ) with distance: 87.647
Node (109, 159, 110, ) with distance: 84.2259
```

```
Tree 2:
Time taken by KDTree: 0.000451 microseconds
Node (127, 76, 13, ) with distance: 78.3071
Node (85, 143, 25, ) with distance: 77.1816
Node (62, 46, 138, ) with distance: 75.9078
Node (139, 38, 66, ) with distance: 74.5721
Node (17, 98, 62, ) with distance: 68.7386
Node (96, 40, 125, ) with distance: 67.7422
Node (107, 24, 76, ) with distance: 66.4906
Node (102, 63, 26, ) with distance: 59.0847
Node (72, 94, 31, ) with distance: 47.9687
Node (86, 89, 42, ) with distance: 35.2562

Time taken by Brute Force: 0.003641 microseconds
Node (127, 76, 13, ) with distance: 78.3071
Node (85, 143, 25, ) with distance: 77.1816
Node (62, 46, 138, ) with distance: 75.9078
Node (139, 38, 66, ) with distance: 74.5721
Node (17, 98, 62, ) with distance: 68.7386
Node (96, 40, 125, ) with distance: 67.7422
Node (107, 24, 76, ) with distance: 66.4906
Node (102, 63, 26, ) with distance: 59.0847
Node (72, 94, 31, ) with distance: 47.9687
Node (86, 89, 42, ) with distance: 35.2562
```

```
Tree 3:
Time taken by KDTree: 0.00028 microseconds
Node (692, 2863, 1320, ) with distance: 3102.85
Node (1127, 2475, 992, ) with distance: 2763.06
Node (1936, 1953, 194, ) with distance: 2633.06
Node (1150, 2318, 954, ) with distance: 2624.77
Node (2348, 1109, 118, ) with distance: 2485.65
Node (499, 193, 2240, ) with distance: 2205.24
Node (29, 1795, 1243, ) with distance: 2069.58
Node (154, 1694, 1286, ) with distance: 2013.05
Node (956, 27, 1534, ) with distance: 1699.55
Node (370, 143, 1440, ) with distance: 1394.05

Time taken by Brute Force: 0.003792 microseconds
Node (692, 2863, 1320, ) with distance: 3102.85
Node (1127, 2475, 992, ) with distance: 2763.06
Node (1936, 1953, 194, ) with distance: 2633.06
Node (1150, 2318, 954, ) with distance: 2624.77
Node (2348, 1109, 118, ) with distance: 2485.65
Node (499, 193, 2240, ) with distance: 2205.24
Node (29, 1795, 1243, ) with distance: 2069.58
Node (154, 1694, 1286, ) with distance: 2013.05
Node (956, 27, 1534, ) with distance: 1699.55
Node (370, 143, 1440, ) with distance: 1394.05
```

3.2. Análisis del Impacto del Parámetro k

En esta sección se analiza cómo afecta el incremento del parámetro k en los tiempos de búsqueda. Se observa que para valores altos de k, el tiempo de búsqueda con fuerza bruta aumenta significativamente en comparación con el KD-Tree. El gráfico de la Figura 1 muestra esta relación.

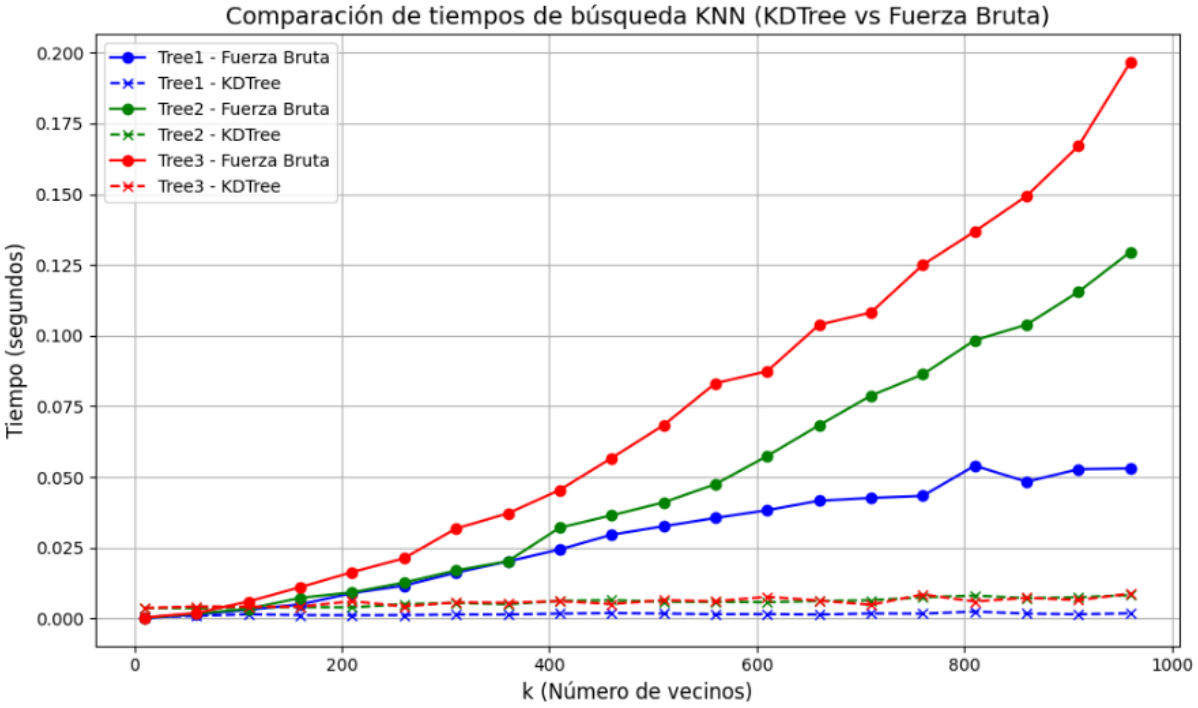

Figura 1: Comparación del tiempo de búsqueda según el parámetro k

4. Conclusiones

- El algoritmo de *fuerza bruta* presenta un rendimiento adecuado para conjuntos de datos pequeños, pero su eficiencia disminuye considerablemente a medida que el tamaño del conjunto de datos aumenta, debido a su complejidad cuadrática en función del número de puntos y vecinos.
- El *KD-Tree* demuestra un rendimiento notablemente superior en términos de tiempo de búsqueda, especialmente cuando se trabaja con conjuntos de datos grandes y valores pequeños de k. Esto se debe a su capacidad para reducir el espacio de búsqueda mediante la división recursiva de los datos.
- A medida que el valor de k aumenta, el tiempo de búsqueda se incrementa de manera más pronunciada para el algoritmo de *fuerza bruta*, ya que el número de comparaciones crece proporcionalmente. En contraste, el *KD-Tree* muestra un aumento más gradual, lo que resalta su ventaja en términos de escalabilidad.

- En general, el uso del *KD-Tree* es recomendable para aplicaciones que requieren búsquedas rápidas en grandes volúmenes de datos, mientras que el algoritmo de *fuerza bruta* sigue siendo útil en escenarios con conjuntos de datos pequeños o cuando la implementación simple es preferible.

Referencias

- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509-517.
- Hastie, T., Tibshirani, R., Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.