KD-Tree: Estructura de Datos para Búsqueda Eficiente en Espacios Multidimensionales

1st Tamayo H. Maria, 2nd Cayllahua H. Joel, 3rd García L. Bruno, 4th Ortega H. Jhonatan.
{maria.tamayo, joel.cayllahua, bruno.garcia, jhonatan.ortega}@utec.edu.pe
Departamento de Ciencias de la Computación
Universidad de Ingeniería y Tecnología (UTEC), Lima, Perú

Kd trees are widely used data structures to accelerate spatial queries such as nearest neighbor search and range queries in multidimensional spaces. In this work, we present a self-contained overview of the k-d tree structure and its core algorithms: balanced construction, exact k-NN search, and orthogonal range queries. We analyze how design decisions—axis selection policy, stopping criteria, and effective dimensionality—affect practical performance in terms of build time and query latency. We also review recent variants tailored to real-world scenarios, including incremental versions for dynamic streams, parallel designs with batch updates, and compact cache-friendly representations. Finally, we propose a reproducible experimental protocol to compare k-d trees, Ball Trees, and a linear baseline, measuring indexing time, query throughput, and memory usage under different dataset sizes, dimensionalities, and values of k.

***Index Terms*—k-d tree, nearest neighbor search, multidimensional indexing, range queries, spatial data structures, performance evaluation**

I. Introduction

LOS *K*d trees son estructuras de datos espaciales diseñadas para acelerar consultas sobre puntos en espacios R^k, como *k*-vecinos más cercanos (*k*-NN) y rangos ortogonales. Desde la propuesta seminal de Bentley [6] y el algoritmo clásico de búsqueda de Friedman–Bentley–Finkel [7], los k-d trees se han consolidado como un pilar para indexación multidimensional en visión por computador, gráficos, SIG y robótica. Su principio básico consiste en particionar recursivamente el espacio con hiperplanos ortogonales, buscando mantener el árbol cercano al balance por mediana para lograr tiempos de consulta sublineales en el promedio.

A pesar de su popularidad, el rendimiento real de un k-d tree depende de tres factores: (i) **la estrategia de construcción**, que controla el balance y el costo de indexado; (ii) **la política de corte por eje**, que incide en la calidad de la poda durante las búsquedas; y (iii) **el régimen de uso**, estático o dinámico. Clásicamente, la construcción balanceada tiene costo $O(n \log n)$ y permite consultas promedio $O(\log n)$, pero el desempeño se degrada con la dimensión efectiva del dato ("maldición de la dimensionalidad") y con distribuciones muy irregulares [8], [12]. Para mitigar estos efectos, la literatura ha explorado políticas de eje basadas en varianza o rango [9], [10], así como esquemas de construcción más eficientes [11].

El auge de aplicaciones *en línea* y de tiempo real (por ejemplo, SLAM y odometría LiDAR) ha expuesto las limitaciones de reconstruir el árbol desde cero ante flujos continuos de puntos. En este contexto, surgen variantes **dinámicas e incrementales** como *ikd-Tree*, que introducen actualizaciones locales (inserciones/eliminaciones), monitorización del balance y *reconstrucción parcial* para sostener latencias en el orden de milisegundos sin sacrificar exactitud [1]. En dominios críticos en seguridad (p. ej., aeroespacial), además, se demanda **determinismo y no recursión** —evitando asignación dinámica y aproximaciones— como enfatiza el trabajo del JPL [2].

En plataformas multinúcleo, la necesidad de throughput impulsa diseños **paralelos con actualizaciones por lotes**. El *Pkd-tree* propone un enfoque de *skeleton* de altura limitada, *sieving* de puntos hacia *buckets* y *rebuild* local bajo un balance relajado, con cotas de trabajo óptimo $O(n \log n)$ y *span* $O(\log^2 n)$ [4]. En escenarios estáticos con presión de memoria, representaciones **compactas y cache-friendly** como *cKd-tree* almacenan el árbol implícitamente en arreglos y reducen la huella sin penalizar la navegación [3].

Aportes de este trabajo. (1) Presentamos una síntesis autocontenida de la estructura del k-d tree y sus algoritmos base (construcción, *k*-NN exacto y consultas de rango); (2) discutimos decisiones de diseño (políticas de eje, complejidad y efectos de la dimensión); (3) revisamos avances recientes —*ikd-Tree*, *Pkd-tree* y *cKd-tree*— destacando sus escenarios de uso; y (4) proponemos un **protocolo experimental reproducible** para comparar tiempo de construcción, rendimiento de consulta (k-NN: QPS y P95/P99) y uso de memoria entre KDTree, BallTree y un baseline lineal sobre datasets sintéticos controlados. El resto del artículo se organiza como sigue: Sec. II presenta el fundamento teórico; Sec. III describe estructuras y algoritmos (incluida la construcción clásica); Sec. IV analiza variantes modernas; Sec. V discute complejidad y compromisos; Sec. VI propone el diseño experimental; y Sec. VII concluye.

II. Fundamento Teórico

A. Estructura de un KD-Tree

Un KD-Tree (K-dimensional tree) es una estructura de datos utilizada para organizar puntos en un espacio K-dimensional. Cada nodo del árbol contiene un punto de K dimensiones. Los nodos no hoja actúan como hiperplanos, dividiendo el espacio en dos partes: un subárbol izquierdo con puntos cuyas coordenadas en el eje de partición son menores y un subárbol

derecho con puntos con coordenadas mayores. En cada nivel del árbol, el eje de partición cambia de forma alternada: primero el eje X, luego el eje Y, y así sucesivamente [6], [8].

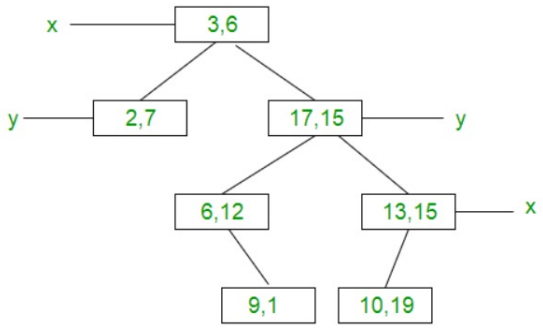

Fig. 1. Estructura de un KD-Tree en un espacio 2D [6], [8].

La Figura 1 ilustra la estructura de un KD-Tree en un espacio bidimensional, donde los nodos actúan como hiperplanos que dividen el espacio en dos subárboles: izquierdo y derecho. En cada nivel del árbol, el espacio se divide utilizando de forma alternada los ejes X y Y. El punto central en cada partición se selecciona como la mediana, y el árbol se construye recursivamente alternando entre los ejes X y Y [6], [8].

B. División del Espacio Multidimensional

En un KD-Tree, el espacio multidimensional se divide recursivamente en semiespacios utilizando planos alineados con las dimensiones del espacio. En el primer nivel, el nodo raíz divide el espacio usando el eje X. Los puntos cuyo valor en esa dimensión es menor que el valor del nodo raíz se colocan en el subárbol izquierdo, y los puntos con valores mayores o iguales se colocan en el subárbol derecho. A medida que se desciende en el árbol, el eje de partición alterna entre X y Y, lo que permite una eficiente distribución de los puntos, optimizando las consultas de búsqueda [6], [8].

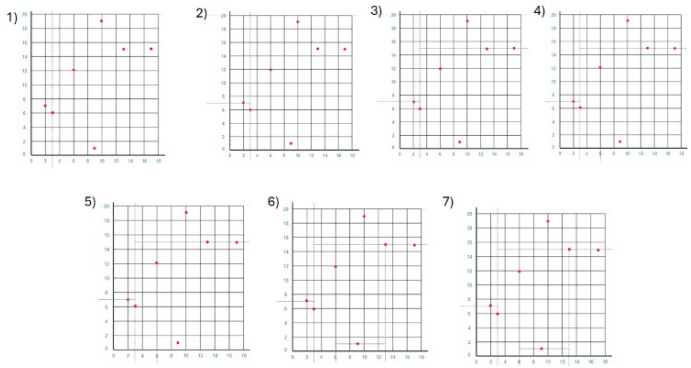

Fig. 2. Partición del espacio multidimensional en un plano 2D utilizando un KD-Tree [6], [8].

La Figura 2 muestra cómo un KD-Tree divide el espacio bidimensional alternando entre los ejes X y Y. En cada nivel del árbol, el espacio se divide en dos subespacios, optimizando las búsquedas de vecinos cercanos [6]–[8].

C. Propiedades Clave

1) Balanceo del Árbol

La estructura del KD-Tree asegura que los puntos se distribuyan de manera balanceada a lo largo de los ejes, lo que mejora la eficiencia de las búsquedas. Al utilizar la mediana como punto de partición en cada nivel, se garantiza que ambos subárboles tengan aproximadamente el mismo número de nodos, resultando en una altura de $O(\log n)$ [**?**].

2) Eficiencia en las Búsquedas

Gracias a la forma en que se dividen los puntos, las búsquedas en el árbol son más rápidas, ya que se reduce el espacio de búsqueda a medida que se desciende por los nodos. Esta propiedad permite alcanzar una complejidad temporal de $O(\log n)$ en el caso promedio para operaciones de búsqueda [**?**], [7].

3) Ventajas y Desventajas

Ventajas:

- Los árboles KD son muy eficientes para consultas de k vecinos más cercanos (KNN) y consultas de rango.
- Son ampliamente utilizados en aprendizaje automático y sistemas de recomendación debido a su eficiencia computacional [12].
- Proporcionan un balance óptimo entre costo de construcción y rendimiento de consultas en espacios de baja a media dimensionalidad.

Desventajas:

- Los árboles KD pueden volverse ineficientes a medida que la dimensionalidad del espacio aumenta, debido a la "maldición de la dimensionalidad" [12].
- En espacios de alta dimensión (típicamente $k > 20$), el rendimiento puede degradarse hasta aproximarse a una búsqueda lineal.
- La estructura rígida del árbol puede dificultar la captura de relaciones geométricas precisas entre puntos en distribuciones de datos complejas [8], [12].

D. Ejemplo Paso a Paso en 2D

Para concretar las definiciones anteriores, consideremos el conjunto de puntos en 2D

$$P = \{(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)\}.$$

1) Construcción del árbol

1) **Nivel 0 (raíz, eje X):** se ordenan los puntos por la coordenada X: $(2,3), (4,7), (5,4), (7,2), (8,1), (9,6)$. La mediana es $(7,2)$, que se almacena en la raíz. Los puntos con $x \leq 7$ forman el subárbol izquierdo y el resto el derecho.
2) **Nivel 1 (hijos, eje Y):**
   - Izquierda: $P_L = \{(2,3), (4,7), (5,4)\}$. Ordenados por Y: $(2,3), (5,4), (4,7)$; la mediana es $(5,4)$, nodo interno con eje Y.
   - Derecha: $P_R = \{(8,1), (9,6)\}$. Ordenados por Y: $(8,1), (9,6)$; la mediana es $(9,6)$.
3) **Nivel 2 (nietos, eje X):**
   - Subárbol de $(5,4)$: a la izquierda queda $(2,3)$ y a la derecha $(4,7)$, ambos como hojas.

- Subárbol de $(9, 6)$: a la izquierda queda $(8, 1)$ y no hay hijo derecho.

El resultado es un KD-Tree balanceado de altura $O(\log n)$, coherente con la construcción por medianas [6], [7].

2) Búsqueda del vecino más cercano

Supongamos ahora que queremos encontrar el vecino más cercano del punto query $q = (9, 2)$:

1) **Descenso inicial:**
   - En la raíz $(7, 2)$, con eje X, comparamos $q_x = 9$ con 7; descendemos al subárbol derecho.
   - En $(9, 6)$, eje Y, comparamos $q_y = 2$ con 6; descendemos al subárbol izquierdo, llegando a la hoja $(8, 1)$.
2) **Inicialización de la mejor distancia:** se calcula $d(q, (8, 1))$ y se guarda como mejor vecino actual.
3) ***Backtracking* y poda:**
   - Se retrocede a $(9, 6)$ y se comprueba si la bola de radio igual a la mejor distancia intersecta el plano $Y = 6$. Si no lo hace, se puede podar el subárbol contrario.
   - Se retrocede a la raíz $(7, 2)$; se verifica de nuevo si la bola intersecta el hiperplano $X = 7$. Si la distancia de q al plano es menor que la mejor distancia, podría haber un vecino mejor en el subárbol izquierdo y se explora; en caso contrario, se poda.

En este ejemplo, el algoritmo visita solo una fracción de los nodos, ilustrando la eficiencia del k-d tree frente a una búsqueda lineal.

III. Estructuras de Datos y Algorítmos

A. Construcción del KD-Tree desde los Datos

La construcción de un KD-Tree a partir de un conjunto de datos multidimensionales es un proceso recursivo que divide el espacio en subespacios mediante hiperplanos perpendiculares a los ejes coordenados. Este enfoque fue propuesto originalmente por Bentley [6], quien estableció las bases para las estructuras de búsqueda multidimensional. El algoritmo básico de construcción se basa en la selección de un punto pivote (típicamente la mediana) y la división del conjunto de datos en dos subconjuntos según su posición relativa al pivote respecto al eje seleccionado [7].

El proceso de construcción sigue los siguientes pasos:

1) **Selección del eje**: Se determina el eje de partición (generalmente alternando entre dimensiones o usando criterios de varianza) [8].
2) **Elección del punto pivote**: Se selecciona la mediana de los puntos según el eje elegido para garantizar un árbol balanceado [11].
3) **Partición del conjunto**: Los puntos se dividen en dos subconjuntos: aquellos menores o iguales al pivote y aquellos mayores.
4) **Construcción recursiva**: Se aplica el mismo proceso recursivamente a cada subconjunto hasta que se alcanza un criterio de parada (típicamente un solo punto o conjunto vacío).

Algorithm 1 Construcción de KD-Tree

Require: Conjunto de puntos P, profundidad actual $depth$, dimensión k
Ensure: Nodo raíz del KD-Tree

BuildKDTree$P, depth, k$
if $P = \emptyset$ **then**
  return $null$
end if
$axis \leftarrow depth \mod k$
Ordenar P según la coordenada $axis$
$median \leftarrow$ punto mediano de P
$node.point \leftarrow median$
$node.axis \leftarrow axis$
$P_{left} \leftarrow \{p \in P : p[axis] \leq median[axis]\}$
$P_{right} \leftarrow \{p \in P : p[axis] > median[axis]\}$
$node.left \leftarrow$ BuildKDTree$(P_{left}, depth + 1, k)$
$node.right \leftarrow$ BuildKDTree$(P_{right}, depth + 1, k)$
return $node$

El Algoritmo 1 presenta el pseudocódigo para la construcción del KD-Tree basado en el método clásico [6].

La complejidad temporal de este algoritmo es $O(n \log^2 n)$ cuando se ordena el conjunto completo en cada nivel, o $O(n \log n)$ si se utiliza el algoritmo de selección de mediana en tiempo lineal (quickselect) [7], [11]. La complejidad espacial es $O(n)$ para almacenar los n puntos en el árbol.

Un aspecto crítico de la construcción es mantener el balance del árbol. Al elegir la mediana como punto pivote en cada nivel, se garantiza que ambos subárboles tengan aproximadamente el mismo número de nodos, resultando en una altura de $O(\log n)$, lo cual es óptimo para operaciones de búsqueda [8]. Estudios recientes han propuesto variantes incrementales [1] y versiones compactas [3] que mejoran la eficiencia en aplicaciones específicas como robótica y procesamiento de datos masivos.

B. Elección del Eje para la Partición

La selección del eje de partición en cada nivel del KD-Tree es fundamental para la eficiencia del árbol resultante [8]. Existen varias estrategias para realizar esta elección, cada una con sus ventajas y desventajas según el tipo de datos y la aplicación [12].

1) Alternancia Cíclica

La estrategia más común y simple es alternar cíclicamente entre las dimensiones disponibles, propuesta en el trabajo original de Bentley [6]. En un espacio k-dimensional, el eje se selecciona mediante:

$$axis = depth \mod k \tag{1}$$

donde $depth$ es la profundidad actual en el árbol. Esta estrategia garantiza que todas las dimensiones sean utilizadas equitativamente y es computacionalmente eficiente ($O(1)$) [7].

Ventajas:

- Simplicidad de implementación
- Costo computacional mínimo
- Distribución uniforme entre dimensiones

Desventajas:

- No considera la distribución real de los datos
- Puede resultar en particiones desbalanceadas con datos no uniformes [9]

2) Máxima Varianza

Una estrategia más sofisticada, propuesta por Moore [9], consiste en seleccionar el eje con mayor varianza en el conjunto de puntos actual. La varianza para la dimensión i se calcula como:

$$\sigma_i^2 = \frac{1}{n}\sum_{j=1}^{n}(p_j[i] - \mu_i)^2 \tag{2}$$

donde μ_i es la media de la coordenada i y n es el número de puntos.

Ventajas:

- Adaptativa a la distribución de datos
- Mejores particiones en espacios no uniformes
- Reduce el overlap entre regiones [10]

Desventajas:

- Mayor costo computacional: $O(nk)$ por nivel
- Complejidad total de construcción aumenta a $O(nk \log n)$ [9]

3) Máximo Rango

Esta estrategia selecciona el eje con el mayor rango (diferencia entre valor máximo y mínimo) [8]:

$$axis = \arg\max_{i \in [1,k]} (\max_j p_j[i] - \min_j p_j[i]) \tag{3}$$

Es computacionalmente menos costosa que la varianza ($O(nk)$ por nivel) y proporciona resultados similares en muchos casos prácticos [10].

4) Comparación de Estrategias

La Tabla I resume las características de cada estrategia según el análisis de diversos estudios [8], [12]:

TABLE I
COMPARACIÓN DE ESTRATEGIAS DE SELECCIÓN DE EJE

Estrategia	Complejidad	Calidad	Uso
Alternancia Cíclica	$O(1)$	Media	General
Máxima Varianza	$O(nk)$	Alta	Datos no uniformes
Máximo Rango	$O(nk)$	Media-Alta	Compromiso

En la práctica, la alternancia cíclica es suficiente para la mayoría de aplicaciones debido a su simplicidad y eficiencia [6], [7]. Sin embargo, para conjuntos de datos con distribuciones muy irregulares o alta dimensionalidad, las estrategias basadas en varianza o rango pueden proporcionar mejoras significativas en el rendimiento de búsqueda [12], justificando el costo adicional de construcción. Implementaciones recientes como [1] y [4] han explorado enfoques híbridos que combinan diferentes estrategias según las características dinámicas de los datos.

C. Búsqueda k-NN y Consultas por Rango

1) Búsqueda k-NN

La búsqueda de los k vecinos más cercanos (*k-NN*) en un KD-Tree sigue el esquema propuesto por Friedman et al. [7]: un descenso guiado por los planos de partición seguido de una fase de retroceso con poda basada en cotas.

Se mantiene una estructura de datos de tamaño acotado k (típicamente un heap máximo) que almacena los mejores candidatos actuales. La distancia del peor vecino en el heap define el radio de la "bola de búsqueda".

Algorithm 2 Búsqueda k-NN en un KD-Tree

Require: Nodo actual $node$, punto consulta q, número de vecinos k, heap H
if $node = null$ **then**
  return
end if
$axis \leftarrow node.axis$
$d \leftarrow \|q - node.point\|$ {Distancia según la métrica elegida}
if $|H| < k$ **then**
  Insertar $(d, node.point)$ en H
else if $d <$ distancia_máxima(H) **then**
  Reemplazar el peor elemento de H por $(d, node.point)$
end if
if $q[axis] \leq node.point[axis]$ **then**
  $first \leftarrow node.left$, $second \leftarrow node.right$
else
  $first \leftarrow node.right$, $second \leftarrow node.left$
end if
{Descenso hacia el lado más prometedor}
KNN_SEARCH$(first, q, k, H)$
{Prueba de solapamiento con la bola}
if $|q[axis] - node.point[axis]| <$ distancia_máxima(H) **then**
  KNN_SEARCH$(second, q, k, H)$
end if

2) Consultas por Rango Ortogonal

En una consulta por rango ortogonal, se busca el conjunto de puntos p tales que p se encuentre dentro de una caja axis-aligned $R = [a_1, b_1] \times \cdots \times [a_k, b_k]$. El algoritmo recorre solo aquellos nodos cuyo hipercubo asociado intersecta R [8].

3) Complejidad Algorítmica

Para un KD-Tree aproximadamente balanceado en baja dimensionalidad, la búsqueda k-NN exacta tiene complejidad esperada

$$O(\log n + k),$$

donde n es el número total de puntos y k el número de vecinos reportados [7], [8]. En la práctica, el número de nodos visitados suele ser casi independiente de n para dimensiones moderadas.

Las consultas por rango ortogonal tienen coste

$$O(\log n + T),$$

donde T es el número de puntos que caen dentro del rango consultado. Sin embargo, en el *peor caso* —datos muy ad-

Algorithm 3 Consulta por rango ortogonal en un KD-Tree

Require: Nodo actual $node$, rango R, lista de resultados S

if $node = null$ **then**
  return
end if
if $node.point \in R$ **then**
  Añadir $node.point$ a S
end if
$axis \leftarrow node.axis$
if $R_{\text{min}}[axis] \leq node.point[axis]$ **then**
  RANGE_QUERY($node.left, R, S$)
end if
if $R_{\text{max}}[axis] \geq node.point[axis]$ **then**
  RANGE_QUERY($node.right, R, S$)
end if

versos o dimensionalidad alta— ambas operaciones pueden degradarse a $O(n)$, ya que la poda se vuelve inefectiva [12].

IV. Análisis Comparativo y de Aplicación

A. Algoritmo de búsqueda más cercana

El algoritmo de búsqueda del vecino más cercano (nearest neighbor search, NNS) en un KD-Tree aprovecha la estructura jerárquica del árbol para evitar revisar todos los puntos, a diferencia de una búsqueda lineal. La idea central es combinar un *descenso guiado* por los planos de partición con una fase de *retroceso* en la que se podan ramas completas usando cotas de distancia [7].

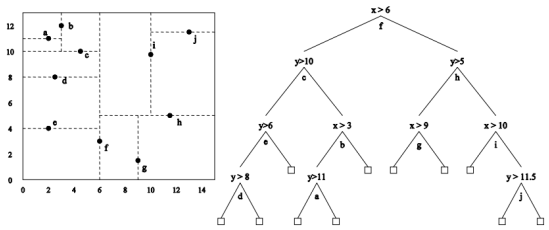

Fig. 3. Recorrido de la búsqueda del vecino más cercano en un KD-Tree [13].

El proceso puede describirse en tres etapas:

1) **Descenso hacia la hoja más prometedora.** A partir de la raíz, en cada nodo se compara la coordenada del punto consulta q en el eje discriminador $axis$ con la coordenada del punto almacenado. Si $q[axis] \leq node.point[axis]$ se desciende al subárbol izquierdo; en caso contrario, al derecho. Este proceso continúa recursivamente hasta llegar a una hoja.
2) **Inicialización del mejor candidato.** En la hoja alcanzada se calculan las distancias de q a los puntos almacenados y se actualiza una estructura de tamaño acotado k (por ejemplo, un *heap* máximo) que contiene los k mejores vecinos encontrados hasta el momento. La distancia del peor vecino en el *heap* define el radio actual de la "bola de búsqueda".
3) ***Backtracking* con poda geométrica.** Se retrocede por el camino recorrido en el árbol. En cada nodo se verifica si el hiperplano de partición puede contener puntos más cercanos que los ya encontrados. Esta prueba se realiza comparando la distancia de q al plano con el radio de la bola: si

$$|q[axis] - node.point[axis]| < r_{\text{actual}},$$

entonces la bola intersecta el subespacio del subárbol opuesto y es necesario explorarlo; en caso contrario, dicho subárbol se *poda* y no se visita. Durante este recorrido inverso se siguen actualizando el *heap* y el radio de búsqueda.

Como se ilustra en la Figura 3, el recorrido combina un descenso inicial hacia la hoja más prometedora con una fase de *backtracking* en la que se decide qué ramas podar según la distancia al plano de partición.

El algoritmo termina cuando no quedan nodos por visitar que puedan mejorar la solución. En escenarios típicos de baja o media dimensionalidad, la poda geométrica permite visitar sólo una fracción de los nodos, de modo que el número de distancias evaluadas es muy inferior a n.

B. Tiempo computacional y comparación con otros algoritmos de búsqueda

Para un KD-Tree aproximadamente balanceado y en baja dimensionalidad, la búsqueda de los k vecinos más cercanos tiene una **complejidad esperada**

$$T_{\text{kNN}}(n, k) = O(\log n + k),$$

donde n es el número total de puntos y k el número de vecinos devueltos [7], [8]. El término $\log n$ proviene del descenso por la altura del árbol y el término k del mantenimiento del *heap* de candidatos. En la práctica, el número de nodos visitados se mantiene casi constante al crecer n mientras la dimensión efectiva sea moderada.

Las **consultas por rango ortogonal** $R = [a_1, b_1] \times \cdots \times [a_k, b_k]$ siguen un patrón similar: se desciende por el árbol y sólo se exploran los nodos cuyo hipercubo asociado intersecta R. Su coste esperado es

$$T_{\text{rango}}(n, T) = O(\log n + T),$$

donde T es el número de puntos que caen dentro del rango consultado [8]. El término $\log n$ corresponde a localizar la región y el término T a reportar los puntos.

En el **peor caso**, tanto k-NN como las consultas por rango pueden degradarse a $O(n)$: por ejemplo, cuando la dimensión es alta o la distribución de los datos hace que casi todos los nodos sean "potencialmente relevantes" y la poda sea inefectiva [12]. En esos escenarios, el comportamiento se aproxima al de una búsqueda lineal.

Si comparamos con otros índices espaciales, en un **Quadtree** (2D, factor de ramificación $b = 4$) la complejidad promedio de búsqueda también es $O(\log N)$ para distribuciones uniformes, pero se deteriora hacia $O(N)$ con datos muy sesgados o altamente concentrados [8]. Por su parte, el **R-Tree** ofrece tiempos promedio $O(\log N + K)$ para consultas de rango, donde K es el número de resultados devueltos, y está mejor adaptado a regiones solapadas y objetos extendidos [8].

Estas diferencias motivan la elección de KD-Tree principalmente para búsqueda k-NN exacta en puntos de baja a media dimensionalidad, mientras que Quadtrees y R-Trees resultan más adecuados para ciertos escenarios 2D o datos espaciales con rectángulos solapados.

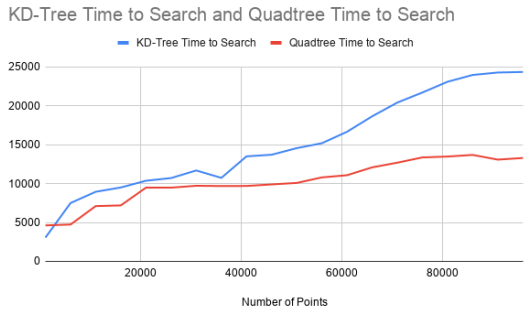

Fig. 4. Comparación experimental del tiempo de búsqueda entre KD-Tree y Quadtree [14].

La Figura 4 muestra una comparación experimental del tiempo de búsqueda entre KD-Tree y Quadtree, donde se aprecia que el KD-Tree mantiene mejores tiempos en distribuciones cercanas a uniformes, mientras que el Quadtree puede degradarse con datos muy sesgados [8].

C. Ventajas y Limitaciones de los KD-Trees

1) Ventajas

- **Eficiencia en búsquedas exactas**: en espacios de baja a media dimensionalidad, los KD-Trees permiten resolver consultas k-NN y de rango en tiempo esperado $O(\log N + k)$, visitando solo una fracción de los nodos del árbol [6]–[8].
- **Árboles relativamente balanceados**: la construcción basada en la mediana mantiene la altura en $O(\log N)$ y asegura subárboles con tamaños comparables, lo que se traduce en latencias bajas y predecibles para índices estáticos [11].
- **Compatibilidad con múltiples métricas**: el esquema de poda solo requiere cotas superiores e inferiores sobre la distancia, por lo que puede adaptarse a distintas funciones de disimilitud (euclidiana, Manhattan, etc.) siempre que se disponga de una cota triangular adecuada [12].
- **Buenas propiedades prácticas en visión y robótica**: en aplicaciones como correspondencia de características en visión por computador o registro de nubes de puntos, los KD-Trees siguen siendo una opción estándar para k-NN exacto antes de recurrir a métodos aproximados [1], [12].

2) Limitaciones

- **Costo de construcción**: la construcción balanceada típica requiere entre $O(N \log N)$ y $O(k\, N \log N)$ operaciones, dependiendo de la estrategia de selección de eje y mediana, lo que puede ser costoso para flujos de datos continuos o reindexaciones frecuentes [1], [11].
- **Maldición de la dimensionalidad**: en dimensiones altas, la mayoría de los nodos no se pueden podar porque muchas regiones quedan "cerca" del punto consulta; el número de nodos visitados se aproxima a N y el rendimiento se degrada al de una búsqueda lineal [12].
- **Sensibilidad a la distribución de datos**: datos muy desbalanceados, con clusters densos y regiones vacías, pueden generar subárboles descompensados y particiones poco informativas. En estos casos, estructuras alternativas como Ball Trees o R-Trees pueden ofrecer un mejor compromiso [8], [12].
- **Actualizaciones dinámicas complejas**: inserciones y eliminaciones locales tienden a degradar el balance del árbol; mantener buenas propiedades asintóticas suele requerir reconstrucciones parciales o estrategias incrementales más sofisticadas, como las propuestas en *ikd-Tree* y *Pkd-tree* [1], [4].

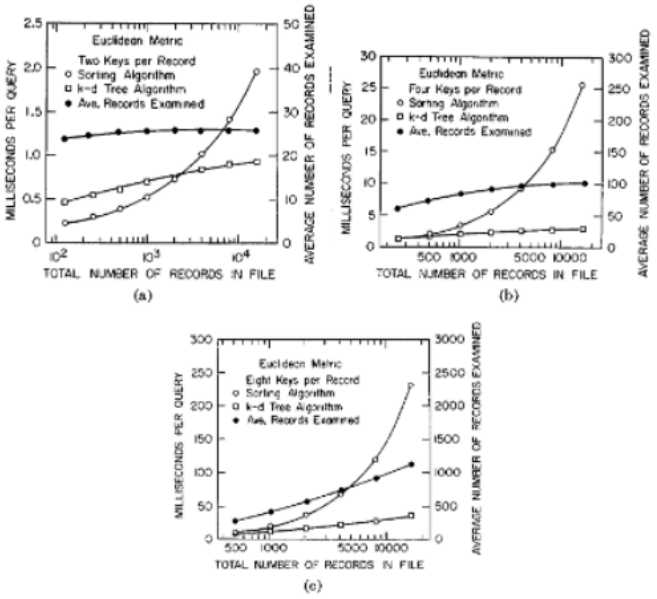

Fig. 5. Cantidad de cálculo computacional y número promedio de registros examinados en función del tamaño del archivo y la dimensión [7].

D. Escalabilidad del árbol k-d en alta dimensión

A medida que la dimensión aumenta, la eficencia disminuye, este problema esconocido como la maldición de la dimensionalidad. Las regiones delimitadas por los nodos del árbol tienden a volverse menos discriminatorias, ya que los buckets y las regiones que contienen registros se contraen y se vuelven menos distintivos, esto provoca efectos como aumento en el número de buckets examinados, la perdida de la logarítmica escala de búsqueda y reducción de la efectividad del particionado, en la siguiente imagen se analiza porque la cantidad de dimensiones afecta el efectividad del árbol [7]

Tal como se observa en la Figura 5, al incrementar la dimensión efectiva crece tanto el costo computacional como el número promedio de registros examinados, acercando el comportamiento al de una búsqueda lineal [7].

En conclusión, el árbol k-d se vuelve menos escalable en espacios de muchas dimensiones, creciendo la cantidad de trabajo para buscar el vecino mas cercano, porque por factores mensionados anteriormente se ejecuta un comportamiento casi

lineal, por ese este algoritmo es más util en dimensiones más cortas como 2D o 3D.

V. Conclusión

Los k-d trees ofrecen un compromiso sólido entre costo de construcción y rapidez de consulta en espacios de baja a media dimensionalidad, siempre que se preserve un buen balance del árbol. Su rendimiento práctico está gobernado por tres factores: (i) la estrategia de construcción y de selección de eje (alternancia, varianza o rango), (ii) la distribución y dimensión efectiva de los datos, y (iii) el régimen de uso (estático vs. dinámico). En datos moderadamente uniformes, la construcción por mediana con alternancia cíclica suele bastar; ante sesgos marcados, los ejes por varianza o rango mejoran la poda; y en flujos en línea, variantes incrementales con reconstrucción parcial sostienen latencias estables sin reindexar desde cero.

No obstante, conforme crece la dimensión efectiva, la poda se debilita y el comportamiento se acerca al lineal, lo que sugiere considerar estructuras alternativas o reducción de dimensionalidad previa. En términos prácticos, recomendamos: construcción balanceada para índices estáticos, diseños compactos *cache-friendly* cuando la memoria es crítica, y paralelismo con actualizaciones por lotes para aumentar *throughput*. Finalmente, el protocolo experimental propuesto —tiempo de construcción, QPS y latencias P50/P95/P99, y huella de memoria bajo variaciones de n, k y k en k-NN— ofrece una base reproducible para comparar KDTree, BallTree y la línea base lineal, guiando elecciones informadas según los requisitos de la aplicación.

Acknowledgment

Este trabajo fue desarrollado como parte del curso de Algoritmos y Estructuras de Datos en la Universidad de Ingeniería y Tecnología (UTEC). Los autores agradecen al profesor Heider Ysaias Sánchez Enríquez por su orientación y retroalimentación.

References

[1] Z. Cai, W. Xu, and M. Zhang, "ikd-Tree: An incremental k-d tree for robotic applications," *arXiv preprint arXiv:2102.10808*, Feb. 2021.

[2] I. Naim, "Deterministic iteratively built kd tree with knn search for exact applications," Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA, Tech. Rep., Jun. 2020.

[3] G. Gutiérrez and Á. Rojas, "ckd-tree: A compact kd-tree," *IEEE Access*, vol. 12, pp. 24089–24103, 2024.

[4] S. Men, Z. Shen, Y. Gu, and Y. Sun, "Parallel kd-tree with batch updates (pkd-tree)," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2025, pp. 1–14.

[5] Anonymous, "Ckd-tree: Construction with lightweight coresets for fast classification," in *Proc. Workshop Efficient Large-Scale Learning*, 2021, pp. 1–8.

[6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sep. 1977.

[8] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA, USA: Addison-Wesley, 1990.

[9] A. W. Moore, "An inductive algorithm for fast computation of multidimensional kd-trees," Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU-RI-TR-91-02, 1991.

[10] R. F. Sproull, "Refinements to nearest-neighbor searching in k-dimensional trees," *Algorithmica*, vol. 6, no. 1-6, pp. 579–589, Jun. 1991.

[11] R. A. Brown, "Building a balanced k-d tree in O(kn log n) time," *J. Comput. Graph. Tech.*, vol. 4, no. 1, pp. 50–68, 2015.

[12] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2227–2240, Nov. 2014.

[13] A. G. Yabo, "Figura 4.21: Generación de un árbol k-d," disponible en ResearchGate, 2016. [En línea]. Disponible: https://www.researchgate.net/ Accedido: Oct. 26, 2025.

[14] A. Amay-Kadre and S. Alavandar, "SpatialSearch: A comparison of spatial search using K-D Trees and Quadtrees," GitHub repository, 2020. [En línea]. Disponible: https://github.com/amay12/SpatialSearch Accedido: Oct. 26, 2025.