KD-Tree y Adaptive KD-Tree

Gaona Briceño, Leonardo Gustavo
Muñoz Curi, Giomar Danny
Philco Puma, Josue Samuel
Rodríguez Lima, Kevin André

5 de noviembre de 2024

1. Introducción

Vamos a explorar la estructura y la funcionalidad del **KD-Tree** y **Adaptive KD-Tree**, son dos importantes estructuras de datos espaciales que van a ser usadas para la búsqueda eficiente en espacios de múltiples dimensiones. Primero comenzaremos desde el origen y creador del **KD-Tree**, que propósito tiene y que operaciones fundamentales soporta. Luego, veremos el **Adaptive KD-Tree** que mejora ciertos aspectos de adaptabilidad y eficiencia en la búsqueda. Finalmente se verán aplicaciones con estas estructuras para ciertos casos.

2. KD-Tree

2.1. Definición del KD-Tree

Los KD-Trees son estructuras para particionar un espacio multidimensional con diferentes variaciones, principalmente según como se vayan manejando dos aspectos: La ubicación de las particiones y la elección del eje de partición. Al ir construyendo el KD-Tree, los nodos van a ir dividiendo el espacio en dos. Este tipo de árbol se utiliza comúnmente para organizar los datos en varias dimensiones.

2.2. Origen y Creador del KD-Tree

En los años 70 estaba el crecimiento del análisis de datos y también problemas geométricos en la computación, surge la necesidad de estructuras de datos que puedan manejar eficientemente puntos en el espacio multidimensional para tareas como búsqueda de vecinos más cercanos, clasificación por rango, etc.

La estructura **KD-Tree** fue inventada en el año 1975 por **Jon Louis Bentley** que buscaba estructuras de datos que fueran eficientes en espacios multidimensionales. Ahora, los `KD` en el **KD-Tree** significa `K-Dimensional` que hace referencia a la capacidad de los árboles para dividir y organizar datos en múltiples dimensiones.

2.3. Propósito del KD-Tree

El KD-Tree se creó como una solución eficiente para organizar y buscar datos en espacios de múltiples dimensiones. En esencia, su propósito es reducir la complejidad de las búsquedas en problemas multidimensionales. Problemas clásicos como encontrar puntos cercanos a un punto de referencia o clasificar datos en un rango específico en 2D, 3D o incluso dimensiones superiores fueron las motivaciones fundamentales.

2.4. Estructura del KD-Tree

Para la estructura del **KD-Tree** tenemos lo siguiente:

- Cada nivel va a tener una dimensión de corte.
- Recorre las dimensiones a medida que vamos bajando por el árbol.
- Cada nodo va a contener un Punto en el espacio `P(x, y)`.
- Si queremos encontra un punto `(x', y')` debemos comparar el punto `x', y'` con el nodo actual y así decidir si vamos hacia el subárbol derecho o subárbol izquierdo.

2.5. Aplicaciones del KD-Tree

- **Búsqueda de puntos más cercanos:** Los KD-Tree's permiten realizar consultas de proximidad de forma eficiente. Por ejemplo, si deseamos saber el punto más cercano a otro punto, el KD-Tree reducirá la cantidad de comparaciones.
- **Sistemas de información geográfica (GIS):** Se trabajan con datos de localización multidimensionales. Los KD-Tree's facilitan los que es la indexación y la búsqueda de datos permitiendo consultar puntos en un área específica.
- **Compresión de imágenes y visión por computadora:** En la visión por computadora, los KD-Tree's se van a utilizar para tareas de coincidencia de características y la compresión.

3. Operaciones del KD-Tree

3.1. Inserción

La inserción en el **KD-Tree** consiste en agregar un nuevo punto en el árbol manteniendo su estructura y sus divisiones en el espacio definidas por las dimensiones.

3.1.1. Proceso de la Inserción

Para realizar la inserción en el KD-Tree debemos tener claro siempre va a alternar en sus dimensiones (en el caso de 2D lo va alternando en las dimensiones `(x, y)`) asegurando la división equitativa en el espacio en cada dirección.

- **Caso inicial:** Si el árbol se encuentra vacío, se va a insertar el primer nodo como la raíz del árbol.
- **Proceso recursivo:** La inserción se va realizando de forma recursiva comenzando en la raíz. En cada nivel, se decide si el nuevo punto se coloca en el subárbol izquierdo o derecho del nodo actual.
- **Proceso de comparaciones:**
  - En la raíz (nivel 0), vamos a comparar con la primera dimensión (`x`) del nuevo punto con la dimensión del nodo actual. Si la coordenada `x` del nuevo punto es menor, se va al subárbol izquierdo; en caso de que sea mayor se va al subárbol derecho.
  - En el siguiente nivel (nivel 1), se va a comparar con la segunda dimensión (`y`) del nuevo punto con el nodo actual. Si la coordenada `y` del nuevo punto es menor, se va al subárbol izquierdo; en caso de que sea mayor se va al subárbol derecho.
- **Inserción en el nodo hoja:** El proceso se va repitiendo alternando en las dimensiones de cada nivel hasta llegar al nodo vacío (null). Una vez llegado ahí, se crea un nuevo nodo y se inserta el punto en esa posición.

3.1.2. Complejidad de la Inserción

- **Complejidad en el Mejor Caso:** En el mejor caso para la inserción es que tengamos el árbol vacío, es decir, al insertar un nuevo punto lo hará de forma directa como la raíz del árbol. Entonces, la complejidad en el mejor caso es constante, es decir, $O(1)$.
- **Complejidad en el Peor Caso:** Esta complejidad dependerá mucho de la distribución de los puntos, si hacemos que los puntos se vayan insertando en orden, básicamente estaremos haciendo inserciones como una lista enlazada. Entonces, si hacemos unas inserciones ordenadas la complejidad se vuelve lineal por visitar todos los nodos para insertar uno nuevo, es decir, $O(n)$ donde n es la cantidad de puntos a insertar.
- **Complejidad en el Caso Promedio:** En este caso, la estructura del árbol se puede decir que está razonablemente balanceada, lo que permite descender por el árbol realizando divisiones eficientes en los nodos hojas. Entonces, la complejidad en Caso Promedio es aproximadamente $O(\log n)$ para realizar las inserciones.

En resumen, la complejidad esperada por la inserción del **KD-Tree** es $O(\log n)$, esto es para evitar un árbol totalmente desbalanceado y evitar el caso lineal.

3.2. Búsqueda

La búsqueda en el **KD-Tree** permite encontrar un punto exacto en el árbol mediante comparaciones sucesivas en cada dimensión. Este va a seguir la misma lógica de la inserción usando la estructura jerárquica para reducir las comparaciones necesarias.

3.2.1. Proceso de la Búsqueda

- **Comienzo en la raíz:** Primero iniciamos la búsqueda en la raíz del árbol, Si el nodo raíz es el punto que buscamos lo retornamos; de caso contrario se explora el subárbol izquierdo o subárbol derecho.
- **Proceso recursivo:** Al igual que la función de inserción, este proceso se realiza de forma recursiva desde la raíz.
- **Proceso de comparaciones:**
  - En el primer nivel (`x`) comparamos con su primera dimensión. Si el valor de `x` del nodo a buscar es menor al del nodo actual se dirige al subárbol izquierdo; de caso contrario, se dirige al subárbol derecho.
  - En el segundo nivel (`y`) comparamos con su segunda dimensión. Si el valor de `y` del nodo a buscar es menor al del nodo actual se dirige al subárbol izquierdo; de caso contrario, se dirige al subárbol derecho.
- **Búsqueda del punto exacto:** Este proceso va repitiendo en cada nivel alternando en sus dimensiones hasta encontrar el el nodo con el punto exacto, lo que indica que cierto punto si existe en el árbol.
- **Caso de no coincidencia:** Si se alcanza a un nodo vacío (null) sin encontrar el punto, significa que el punto buscado no se encuentra presente en el árbol.

3.2.2. Complejidad de la Búsqueda

- **Complejidad en el Mejor caso:** En el mejor caso es que el punto que estemos buscando sea la raíz del árbol o en los primeros niveles del árbol logrando encontrar el punto rápidamente sin necesidad de recorrer la mayoría de los nodos. Entonces, la complejidad en este caso sería constante, es decir, es $O(1)$.
- **Complejidad en el Peor Caso:** En el peor caso, al igual que la inserción, es que nuestro árbol se encuentre muy desbalanceado y parezca una lista enlazada donde si debemos recorrer todos los nodos. Entonces, la complejidad sería lineal, es decir, es $O(n)$.
- **Complejidad en el Caso Promedio:** Para este caso es cuando el árbol se encuentra razonablemente balanceado, entonces acá la búsqueda va a empezar a descender aproximadamente la áltura del árbol que es $\log n$ antes de encontrar el punto exacto. Entonces, la complejidad es aproximadamente $O(\log n)$.

En resumen, la complejidad que se espera para la búsqueda de un punto exacto es de $O(\log n)$ aprovechando la estructura en la partición del árbol reduciendo el número de comparaciones necesarias.

3.3. Eliminación

Está función puede ser muy compleja para el **KD-Tree**, lo que hace es encontrar el nodo que se va a eliminar sin romper la estructura de partición espacial del árbol.

3.3.1. Proceso de la eliminación

- **Búsqueda del nodo a eliminar:** Al igual que un árbol binario, este proceso se va a realizar de forma recursiva. Primero debemos buscar el nodo que vamos a eliminar.
- **Eliminar el nodo:** Una vez encontrado el nodo que queremos eliminar, no vamos a eliminar el nodo como si nada, debemos tener en cuenta unos casos especiales para mantener la estructura del árbol.

- **Casos para la eliminación del nodo:** Al encontrar el nodo que vamos a eliminar, vamos a manejar los siguientes casos:

  - **Caso 1 (Si el nodo es una hoja):** Si el nodo que queremos eliminar es un nodo hoja, se elimina directamente, no se requiere una reestructura al árbol.
  - **Caso 2 (Si el nodo tiene subárbol derecho):** Si el nodo que se eliminará tiene subárbol derecho, vamos a encontrar el nodo mínimo en la dimensión actual en el subárbol derecho. Una vez encontrado el mínimo, vamos a reemplazar el nodo actual con el nodo mínimo y eliminar recursivamente el nodo mínimo del subárbol derecho.
  - **Caso 3 (Si el nodo tiene subárbol izquierdo):** Si el nodo que vamos a eliminar tiene subárbol izquierdo, vamos a encontrar el nodo mínimo en el subárbol izquierdo. Una vez que encontramos el nodo mínimo, vamos a reemplazar el nodo actual con el nodo mínimo y eliminar recursivamente el nodo mínimo del subárbol izquierdo.

3.3.2. Complejidad de la eliminación

- **Mejor Caso:** Esto solo ocurrirá si es que el árbol contiene un solo nodo, que sería la raíz, en este caso la eliminación lo hace directo sin verificar los otros 2 casos. Entonces, la complejidad es $O(1)$.

- **Peor Caso:** En caso de que el árbol este muy desbalanceado simulando una lista enlazada, buscar el nodo mínimo puede tomar tiempo lineal y eliminarlo recursivamente sería constante. Entonces, la complejidad seria lineal es decir, es $O(n)$.

- **Caso Promedio:** Esto se debe a que el árbol KD tiene una estructura equilibrada con niveles $\log n$, y las operaciones de inserción, eliminación y búsqueda recorren el árbol de una manera que reduce el espacio de búsqueda a la mitad en cada nivel. Entonces, la complejidad en el caso promedio sería de $O(\log n)$.

3.4. Búsqueda por Rango

La búsqueda por rango en el **KD-Tree** encuentra los puntos que se van a encontrar en una región rectangular determinada. Por ejemplo puede ser en los rangos [10, 10] y [40, 50].

3.4.1. Proceso de la búsqueda por rango

- **Definiendo el rango:** Primero debemos especificar en que rangos vamos a buscar, dentro de ese rango se van a encontrar los puntos buscados.

- **Comienzo de la búsqueda:** La búsqueda empezará desde la raíz, si este se encuentra en el rango se agrega al resultado; de caso contrario, vamos a buscar en el subárbol izquierdo y subárbol derecho.

- **Buscar recursivamente:** Una vez partido de la raíz buscamos en el subárbol izquierdo y derecho verificando si el punto del nodo se encuentra dentro del rango. De ser así se agrega al resultado.

3.4.2. Complejidad de la búsqueda por rango

- **Mejor Caso:** El mejor caso solo se va a dar cuando el rango va a ser más pequeño para abarcar muy pocos puntos. En este caso, la complejidad sería $O(\log n)$.

- **Peor caso:** Para el peor caso ocurre cuando el rango de búsqueda es muy grande o si el árbol esta desequilibrado y esto va a ocasionar que debamos explorar todos los nodos del árbol. Entonces, la complejidad se vuelve lineal, es decir, es $O(n)$.

- **Caso Promedio:** Este se da si es que el árbol se encuentra medianamente equilibrado, cada nodo puede descartar aproximadamente la mitad del espacio de búsqueda en cada nivel. Entonces, la complejidad sería $O(\sqrt{n} + k)$.

3.5. Visualización del KD-Tree usando matplolib y graphviz

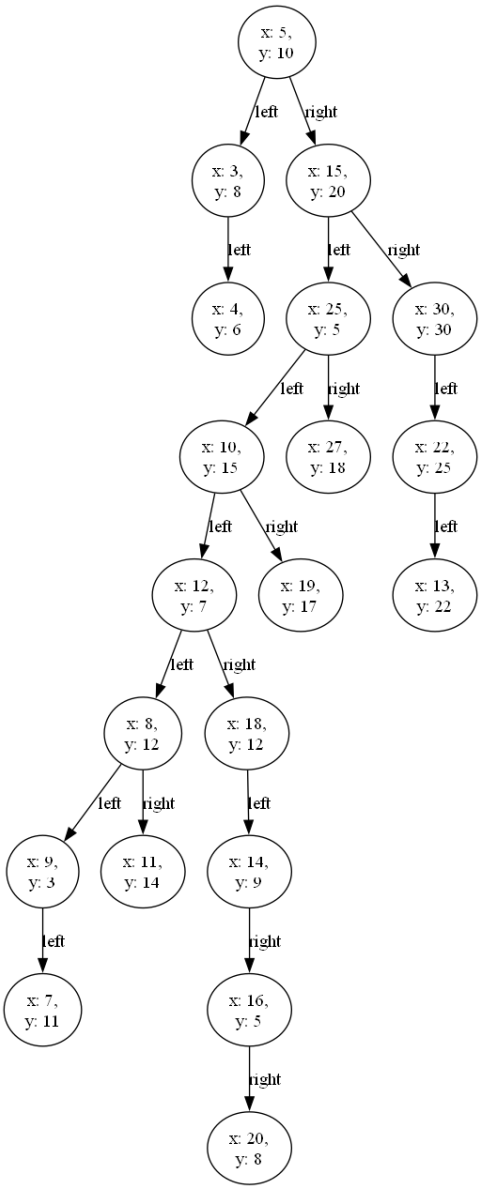

Figura 1: Árbol generado usando Graphviz

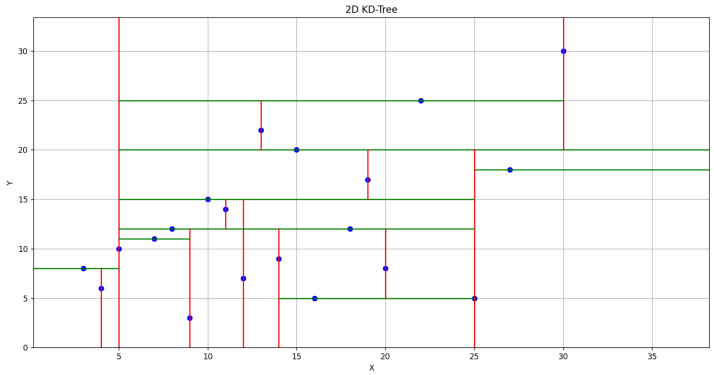

Figura 2: Gráfico 2D del KD-Tree

4. Adaptive KD-Tree

4.1. Definición del Adaptive KD-Tree

En el Adaptive KD-Tree, la elección del eje de división se basa en la dispersión de los puntos en cada dimensión. Esto se calcula, por ejemplo, usando la diferencia entre los valores máximo y mínimo (o el rango) en cada dimensión. Al seleccionar el eje con la mayor dispersión, el árbol se divide de manera que maximiza la equidad en la distribución de los datos, evitando una estructura desbalanceada y mejorando la eficiencia en búsquedas.

4.2. Características principales

- **División Adaptativa:** En un Adaptive KD-Tree, la elección de la dimensión para dividir los datos no es fija. En su lugar, se adapta según la variabilidad y la distribución de los puntos en el espacio, lo que puede mejorar significativamente el rendimiento en comparación con un KD-Tree estándar.

- **Manejo de altas dimensiones:** Esta estructura es especialmente útil en contextos donde se trabaja con datos de alta dimensión, como en aplicaciones de aprendizaje automático y visión por computadora. La adaptabilidad del árbol permite mitigar el problema conocido como *curse of dimensionality*, que puede dificultar la búsqueda eficiente en espacios multidimensionales.

- **Eficiencia en Búsquedas:** Al permitir divisiones más informadas y específicas, el Adaptive KD-Tree puede reducir el espacio de búsqueda en cada nivel del árbol. Esto resulta en tiempos de consulta más rápidos, ya que se pueden descartar más rápidamente grandes partes del espacio de búsqueda que no contienen los puntos cercanos al objetivo.

4.3. Aplicaciones con Adaptive KD-Tree

- **Sistemas de recomendación y motores de búsqueda:** Al usar vectores de características para representar usuarios o productos, el Adaptive KD-Tree va a facilitar las búsquedas en espacios de alta dimensión logrando optimizar consultas de similitud.

- **Filtros de imágenes:** En una imagen si queremos detectar áreas similares (tonos y texturas) en una zona específica, el Adaptive KD-Tree lo que va a hacer es ajustar la partición del espacio en función a sus densidades de los píxeles similares.

- **Videojuegos de mapas abiertos:** Al ser juegos de mapas grandes en un videojuego, algunos puntos se concentran en un espacio. El Adaptive KD-Tree puede lograr una búsqueda de enemigos o puntos de interés cercanos al jugador de forma más rápida en ese tipo de espacios.

5. Especificación Algebraica del Adaptive KD-Tree

Esta sección presenta la especificación algebraica para un Adaptive KD-Tree, definiendo sus operaciones fundamentales y axiomas.

5.1. Operaciones

$$
\begin{aligned}
\texttt{make}: &\ \rightarrow \texttt{AdaptiveKDTree} \\
\texttt{insert}: &\ \texttt{AdaptiveKDTree} \times \texttt{point} \rightarrow \texttt{AdaptiveKDTree} \\
\texttt{search}: &\ \texttt{AdaptiveKDTree} \times \texttt{point} \rightarrow \texttt{Boolean} \\
\texttt{search}: &\ \texttt{AdaptiveKDTree} \times \texttt{range} \rightarrow \texttt{Set(point)}
\end{aligned}
$$

Axiomas:

$$
\begin{aligned}
\texttt{make()} &= \texttt{AdaptiveKDTree} \\
\texttt{insert(make(), p)} &= \texttt{AdaptiveKDTree} \\
\texttt{search(make(), p)} &= \texttt{False} \\
\texttt{search(insert(AdaptiveKDTree, p), p)} &= \texttt{True} \\
\texttt{search(insert(AdaptiveKDTree, p), q)} &= \texttt{search(t, q)}, \quad \text{si } p \neq q
\end{aligned}
$$

5.2. Descripción de las Operaciones

- **make:** Crea un árbol KD vacío. Es la operación de inicialización que devuelve un `AdaptiveKDTree` sin ningún punto insertado.
- **insert:** Inserta un punto en el árbol. Esta operación toma un `AdaptiveKDTree` y un punto (representado como `point`) y devuelve un nuevo árbol que incluye el punto insertado.
- **search (punto):** Busca un punto específico en el árbol. La operación toma un `AdaptiveKDTree` y un punto, y devuelve un valor booleano indicando si el punto está presente o no en el árbol.
- **search (rango):** Realiza una búsqueda de puntos dentro de un rango en el árbol. La operación toma un `AdaptiveKDTree` y un rango (representado como `range`), y devuelve un conjunto de puntos que están dentro del rango especificado.

6. Operaciones del Adaptive KD-Tree

6.1. Inserción

Para insertar los nodos vamos a depender de la carácteristica principal del **Adaptative KD-Tree**, este se encarga de determinar el eje con una mayor dispersión en los puntos. Esto va a permitir que el árbol divida los puntos de manera más eficiente.

6.1.1. Proceso de la inserción

- **Caso base:** Si el árbol no contiene ningún punto significa que se encuentra vacío.
- **Calcular el eje de división:** Vamos a analizar la dispersión de los puntos en cada dimensión para determinar el eje con la mayor diferencia entre el valor máximo y mínimo. Ese eje va a ser seleccionado como eje de división actual.
- **Ordenar los puntos por eje de división:** Una vez seleccionado el eje de mayor dispersión, vamos a ordenar los puntos en función del eje de división.
- **Dividir y recursión:** Ahora con los puntos ordenados seleccionamos el punto medio y creamos el nodo con ese punto. El conjunto de puntos se divide en 2 subconjuntos (izquierdo y derecho) representando los puntos a la izquierda y derecha del punto medio. Finalmente, la función se llama recursivamente en ambos subconjuntos.

6.1.2. Complejidad de la inserción

- **Mejor Caso:** Los puntos estan distribuidos de manera uniforme en todas las dimensiones lo que permite que la construcción del árbol sea balanceada. Acada nivel, el conjunto de puntos se reduce a la mitad generando $O(\log n)$ niveles de profundidad. En cada nivel, debemos calcular el eje de mayor dispersión que puede tomar $O(n)$ y ordenando los puntos sería $O(n \log n)$. Entonces en el mejor caso la complejidad es $O(n \log^2 n)$.
- **Peor caso:** Aunque el Adaptive KD-Tree esta diseñado para mejorar el balance del árbol, no siempre garantiza que esté balanceado, esto ya depende en la forma en que se insertaron los puntos provocando divisiones desiguales. En este caso, la complejidad puede llegar a ser $O(n^2)$
- **Caso Promedio:** Es cuando los puntos no están perfectamente uniformes pero se permite una división razonablemente balanceada para la mayoria de los niveles. Al igual que el mejor caso, la complejidad de este será $O(n \log^2 n)$.

6.2. Búsqueda y Búsqueda por rango

En el Adaptive KD-Tree, las funciones de búsqueda y búsqueda por rango comparten una estructura similar con el KD-Tree, pero presentan diferencias clave en la forma de realizar las divisiones en el espacio. A continuación, se describen estas diferencias y sus implicaciones en la eficiencia.

6.2.1. Search en Adaptive KD-Tree y KD-Tree

- **Adaptive KD-Tree:** En cada nivel, el Adaptive KD-Tree elige el eje de división basado en la mayor variación de los puntos en ese nivel, permitiendo adaptarse mejor a la distribución de los datos. Esto puede reducir el número de comparaciones necesarias para encontrar un punto específico, especialmente en distribuciones de datos no uniformes.

- **KD-Tree:** El KD-Tree utiliza un patrón fijo para dividir el espacio (por ejemplo, alterna entre `(x, y)` y en 2D). Esta estructura es menos adaptable a la distribución real de los datos, lo cual puede resultar en una eficiencia menor en ciertos casos, aunque mantiene una simplicidad que puede ser beneficiosa en escenarios con datos distribuidos uniformemente.

6.2.2. Range Search en Adaptive KD-Tree y KD-Tree

En la búsqueda por rango, el enfoque en las divisiones también difiere:

- **KD-Tree**: La búsqueda por rango en un KD-Tree sigue divisiones fijas según el nivel del nodo, alternando entre los ejes de manera predeterminada. Esto simplifica la búsqueda, pero puede resultar en exploraciones innecesarias cuando los datos están distribuidos de forma desigual.

- **Adaptive KD-Tree**: La elección adaptativa del eje permite una partición más acorde con la distribución de los datos. Al adaptar cada división a la mayor dispersión, el Adaptive KD-Tree tiene el potencial de limitar el área de búsqueda de manera más eficiente, aunque en la práctica los tiempos de búsqueda por rango pueden ser similares al KD-Tree en ciertos escenarios.

6.2.3. Complejidad de búsqueda y búsqueda por rango

Ambas funciones van a cumplir con el mismo objetivo siguiendo los mismo puntos de las funciones de búsqueda del **KD-Tree**, siendo así que la complejidad de estas funciones serían las siguientes:

- **Función de búsqueda**
  - **Mejor Caso:** $O(1)$.
  - **Peor Caso:** $O(n)$.
  - **Caso Promedio:** $O(\log n)$.
- **Función de búsqueda por rango**
  - **Mejor Caso:** $O(1)$.
  - **Peor Caso:** $O(n)$.
  - **Caso Promedio:** $O(\log n)$.

6.3. Visualización del Adaptive KD-Tree usando matplolib y graphviz

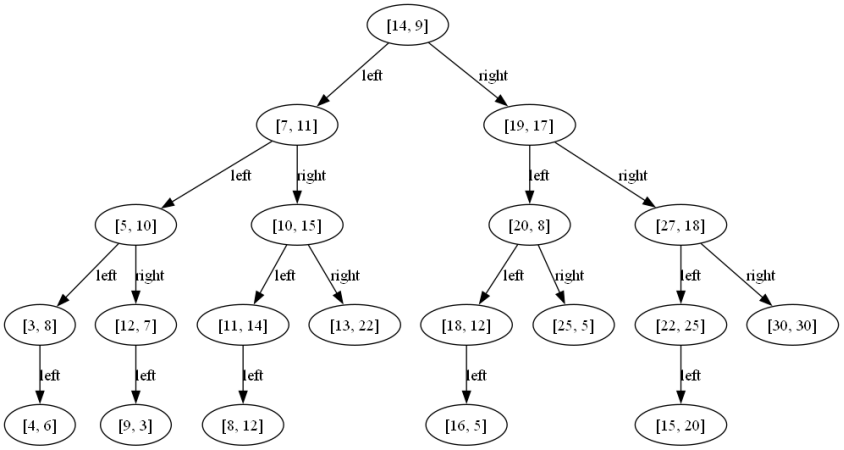

Figura 3: Árbol generado con Graphviz

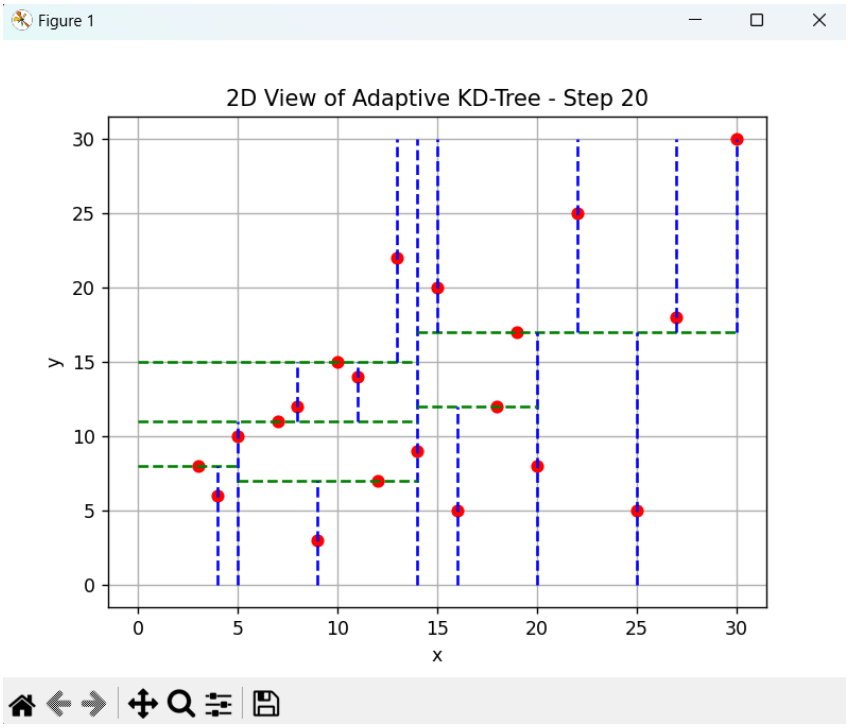

Figura 4: Gráfico 2D del Adaptive KD-Tree

7. Aplicaciones computacionakes con KD-Tree y Adaptive KD-Tree

7.1. Segmentación de Imágenes con KD-Tree

7.1.1. Introducción

El KD-Tree es una estructura de datos eficiente para realizar búsquedas en espacios multidimensionales, lo cual lo hace ideal para aplicaciones de procesamiento de imágenes, como la segmentación. Este proceso divide una imagen en regiones basadas en similitudes, facilitando la identificación de objetos o características relevantes. La segmentación se realiza comparando la similitud de colores o intensidades

entre píxeles, una tarea que el KD-Tree optimiza al reducir el número de comparaciones necesarias en grandes conjuntos de datos.

7.1.2. Descripción del Proceso

En la segmentación de imágenes, el KD-Tree organiza los píxeles de la imagen en un espacio de color, como RGB. Utilizando esta estructura, es posible agrupar píxeles con colores similares en regiones contiguas, mejorando la eficiencia en consultas de búsqueda y clasificación de píxeles. Por ejemplo, dado un color de referencia, el KD-Tree permite identificar rápidamente los píxeles más similares en la imagen.

Figura 5: Segmentación de imagen basada en color usando KD-Tree. Izquierda: Segmentación por color rojo. Derecha: Segmentación por color amarillo.

7.1.3. Ventajas del KD-Tree en Segmentación

La estructura jerárquica del KD-Tree permite que la segmentación sea considerablemente más rápida en comparación con métodos de búsqueda lineal, especialmente en imágenes de alta resolución. Además, el KD-Tree se adapta bien a espacios de color tridimensionales, haciendo que la segmentación basada en atributos de color sea eficiente incluso en conjuntos de datos grandes y complejos.

7.2. Aplicación del Adaptive KDTree: Comparativa con KDTree Tradicional

7.2.1. Introducción

En esta sección se presenta una comparación entre el Adaptive KDTree y el KDTree tradicional, utilizando como caso de estudio un sistema de recomendación de canciones. El objetivo principal es demostrar la superioridad del Adaptive KDTree en términos de eficiencia en la búsqueda de los vecinos más cercanos en un conjunto de datos musicales. Esta comparativa se basa en el tiempo de ejecución y la calidad de los resultados obtenidos por ambos árboles al realizar la misma tarea de recomendación.

7.2.2. Descripción del Código

El código desarrollado permite realizar recomendaciones de canciones similares a partir de un conjunto de datos que contiene atributos como BPM, energía, danza, y valencia. Para ello, se construyeron dos estructuras: un KDTree tradicional y un Adaptive KDTree.

El proceso comienza con la normalización de los atributos de las canciones nombrados previamente, lo cual facilita la comparación equitativa entre diferentes métricas. Luego, ambas estructuras se utilizan para encontrar canciones similares a una dada, mostrando sus atributos y comparando los tiempos de ejecución. El Adaptive KDTree muestra una mejora significativa en los tiempos de búsqueda al optimizar la forma en que se crean los nodos, haciendo énfasis en la adaptabilidad a la distribución del conjunto de datos.

7.2.3. Resultados

Los resultados obtenidos muestran que el Adaptive KDTree supera en eficiencia al KDTree tradicional, especialmente al trabajar con conjuntos de datos de alta dimensionalidad y distribuciones no uniformes. La adaptabilidad del Adaptive KDTree le permite realizar búsquedas de vecinos más rápidas, lo cual se refleja en menores tiempos de ejecución durante el proceso de recomendación.

```
Seleccione el índice de la cancion para obtener recomendaciones: 475

Canciones similares a: Despacito por Luis Fonsi ft. Daddy Yankee (KDTree)
Atributos de la canción seleccionada: BPM: 172, Energía: 0.88, Danza: 0.63, Valencia: 0.22

 -> Dance Monkey por Tones and I (Género: Clásica)
  Atributos: BPM: 164, Energía: 0.83, Danza: 0.56, Valencia: 0.26
 -> Faded por Alan Walker (Género: Pop)
  Atributos: BPM: 159, Energía: 0.85, Danza: 0.53, Valencia: 0.26
 -> Shallow por Lady Gaga & Bradley Cooper (Género: Hip-Hop)
  Atributos: BPM: 170, Energía: 0.83, Danza: 0.54, Valencia: 0.32
 -> Love Yourself por Justin Bieber (Género: Rock)
  Atributos: BPM: 162, Energía: 0.85, Danza: 0.51, Valencia: 0.24
 -> Circles por Post Malone (Género: Pop)
  Atributos: BPM: 155, Energía: 0.9, Danza: 0.74, Valencia: 0.2
Tiempo de bisqueda con KDTree: 0.003890 segundos

Canciones similares a: Despacito por Luis Fonsi ft. Daddy Yankee (Adaptive KDTree)
Atributos de la canción seleccionada: BPM: 172, Energía: 0.88, Danza: 0.63, Valencia: 0.22

 -> Dance Monkey por Tones and I (Género: Clásica)
  Atributos: BPM: 164, Energía: 0.83, Danza: 0.56, Valencia: 0.26
 -> Faded por Alan Walker (Género: Pop)
  Atributos: BPM: 159, Energía: 0.85, Danza: 0.53, Valencia: 0.26
 -> Shallow por Lady Gaga & Bradley Cooper (Género: Hip-Hop)
  Atributos: BPM: 170, Energía: 0.83, Danza: 0.54, Valencia: 0.32
 -> Love Yourself por Justin Bieber (Género: Rock)
  Atributos: BPM: 162, Energía: 0.85, Danza: 0.51, Valencia: 0.24
 -> Circles por Post Malone (Género: Pop)
  Atributos: BPM: 155, Energía: 0.9, Danza: 0.74, Valencia: 0.2
Tiempo de busqueda con Adaptive KDTree: 0.001019 segundos
```

Figura 6: Resultado