UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

Cuantización Vectorial Utilizando Árboles k-Dimensionales

VÍCTOR BASTIÁN AYALA MEDINA

Profesor Guía: CÉSAR ALEJANDRO ASTUDILLO HERNÁNDEZ

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
Enero, 2018

TALCA UNIVERSIDAD CHILE

CONSTANCIA

La Dirección del Sistema de Bibliotecas a través de su encargado Biblioteca Campus Curicó certifica que el autor del siguiente trabajo de titulación ha firmado su autorización para la reproducción en forma total o parcial e ilimitada del mismo.

Curicó, 2019

Dedicado a mi familia, amigos y compañeros. En especial a mis abuelos, padres, hermanos y pareja,
por su apoyo incondicional.

AGRADECIMIENTOS

Agradezco a todas las personas que fueron un apoyo importante en este proceso, a las personas que dieron todo por que yo pudiera estar en este instante y lograr todo lo que tengo, ya sea en el ámbito académico, laboral y sentimental.

Gracias a mis padres por estar en todo momento, entregando todas las herramientas necesarias para lograr mis objetivos, ayudándome cuando estuve derrotado, y por compartir tantos momentos inolvidables.

Gracias a mi pareja y su familia, a mis hermanos, amigos y compañeros, los cuales me ayudaron en todos estos años de estudio, por aportar desde un grano de arena hasta lo impensado, con el fin de apoyarme, ya que fue necesario y muy importante para poder construir mi carrera, o por el solo motivo de estar ahí muchas gracias.

Gracias a mi profesor guía Cesar Astudillo y a todos los profesores, por la paciencia, apoyo, guía y orientación en todo este proceso universitario.

Por último pero no menos importante, gracias a los momentos que me ha entregado la vida, la cual me ha dado la gran oportunidad de compartir con todas las personas antes mencionadas.

gratias ago.

RESUMEN

En computación, la compresión de datos se refiere a la capacidad de reducir el volumen de datos, representando de forma sucinta los datos originales, con el fin de optimizar el espacio empleado para guardar estos valores o mejorar la velocidad de comunicación. Existen dos familias de métodos para realizar este trabajo, LOSSLESS en la cual se almacena íntegramente la información y LOSSY en la que se pierde parte de la información original. Cuantización vectorial (VQ) es un algoritmo de compresión de datos perteneciente a la familia LOSSY.

VQ ha sido aplicado con éxito en gran número de aplicaciones en los campos de compresión y transmisión de datos multimedia, reconocimiento del habla, marca de agua digital, recuperación de imágenes y audio entre otros, ya que posee un ratio de compresión sumamente alto, entregando grandes beneficios al mundo de la ingeniería. Pero este algoritmo posee una dificultad, la cual radica en su alto tiempo de codificación y decodificación al procesar grandes volúmenes de información.

En este documento se presenta el desarrollo de una variante del algoritmo VQ, el cual es un algoritmo híbrido, que utiliza las migraciones de los vectores como se realiza en VQ y se estructura estos valores migrados en un árbol k-dimensional (KD-TREE-VQ). Este último, con la finalidad de disminuir la complejidad de ejecución de VQ, mejorando los tiempo de codificación y decodificación al procesar volúmenes grandes de información.

Con la finalidad de verificar si el nuevo algoritmo KD-TREE-VQ logra mejorar la complejidad temporal, se realizaron pruebas con archivos de distintos tamaños, comparando el tiempo de codificación, decodificación y la calidad de la respuesta entregada, llegando a la siguiente conclusión: El tiempo de decodificación empleado en el algoritmo KD-TREE-VQ es mejor que en VQ, por otro lado, KD-TREE-VQ logra tener una calidad de respuesta muy cercana a la de VQ y ésta la entrega en un tiempo de ejecución menor que VQ. Logrando así mejorar la complejidad temporal del algoritmo VQ en forma experimental.

Cuantización Vectorial, árbol, kd-tree, Compresión de datos, VQ.

TABLA DE CONTENIDOS

página

Dedicatoria I

Agradecimientos II

Tabla de Contenidos III

Índice de Figuras V

Índice de Tablas VI

Resumen VII

1. Introducción **8**

1.1. Descripción de la Propuesta 8
1.2. Hipótesis 10
1.3. Objetivo General 10
1.4. Objetivos Específicos 10
1.5. Alcances 11

2. Antecedentes **12**

2.1. Introducción 12
2.2. Cuantización Vectorial 12
2.3. Árbol K-Dimensional 15
2.4. Métodos Alternativos 18
2.4.1. Árbol de Partición Binaria del Espacio 18
2.4.2. Árbol Cuaternario 19
2.5. Trabajo Relacionado 20
2.5.1. TSVQ in Parallel Machine 20
2.5.2. DTIE-TSVQ 21
2.5.3. HVQ 22
2.5.4. CTAS-ISOC 23
2.5.5. MCTAS 24

2.5.6. RVQ-Lloyd y RVQ-KD 25
2.5.7. FMIR 27
2.5.8. Cuadro Comparativo 28

3. Cuantización Vectorial utilizando KD-TREE **29**
3.1. Introducción 29
3.2. KD-TREE-VQ 2.0 29
3.2.1. Análisis Primario 32
3.2.2. Análisis Secundario 34
3.2.3. Migración de un Nodo 36
3.3. Bosque de Búsqueda en KD-TREE-VQ 36

4. Análisis de los Resultados **38**
4.1. Introducción 38
4.2. Plan de Pruebas 38
4.3. Análisis y Resultados Experimentales 41
4.3.1. Estudio con Volumen Pequeño de Datos 41
4.3.2. Estudio con Volumen Grande de Datos 42
4.3.3. Estudio con División de Dimensiones en los Datos 43

5. Conclusión y Trabajo Futuro **46**
5.1. Conclusión 46
5.2. Trabajo Futuro 48

Glosario **49**

Bibliografía **51**

Anexos

A: Primeros Pasos **54**

ÍNDICE DE FIGURAS

página

2.1. Migración del punto W(t) hasta el punto W(t+1) con respecto al punto X(t). 15
2.2. Divisiones del plano y representación en el árbol. 16
2.3. Secuencia de construcción de un BSP-Tree. 18
2.4. QuadTree de regiones. 19
2.5. QuadTree de puntos. 20

3.1. Representación de prototipos en un árbol k-dimensional. 30
3.2. Definición de límites superior he inferior 32
3.3. Árbol completo a sub dividir de 4 dimensiones 37
3.4. Bosque de búsqueda 37

A.1. Hiperplano donde se realizan las divisiones. 56
A.2. Visualización de un árbol de altura 3, por niveles. 58
A.3. Seguimiento de algoritmo de dibujo y árbol asociado. 58
A.4. Nodo fuera de rango 59
A.5. Árbol con error en nodo 23 60

ÍNDICE DE TABLAS

página

2.1. TSVQ in parallel machine vs KD-TREE-VQ. 21
2.2. DTIE-TSVQ vs KD-TREE-VQ. 22
2.3. HVQ vs KD-TREE-VQ. 23
2.4. CTAS-ISOC vs KD-TREE-VQ. 24
2.5. MCTAS vs KD-TREE-VQ. 25
2.6. RVQ-LLoyd y RVQ-KD vs KD-TREE-VQ. 27
2.7. Cuadro comparativo general. 28

4.1. Archivos de datos. 41
4.2. Análisis archivo diabetes. H:Altura árbol – P: Prototipos VQ 42
4.3. Análisis de archivo abalone. 42
4.4. Skin NomSkin4. 43
4.5. covtype 10 dimensiones. 43
4.6. Análisis archivo diabetes. H: Altura sub-árbol 44
4.7. covtype 10 dimensiones. 44
4.8. MiniBooNE 50 dimensiones. 45

1. Introducción

El avance mejora el mundo.

1.1. Descripción de la Propuesta

El algoritmo de cuantización vectorial (VQ) es un algoritmo de compresión de datos, el cual produce una representación sucinta de los datos originales. VQ ha encontrado un gran número de aplicaciones en los campos de compresión y transmisión de datos multimedia. Es por esto que VQ actualmente ha sido utilizado en una gran cantidad de aplicaciones en diversas áreas de la ingeniería tales como el reconocimiento del habla, marca de agua digital, recuperación de imágenes y audio entre otros [8].

VQ trabaja identificando el vector que mejor representa a los datos de entrada. Esto se logra a través de un proceso denominado como la búsqueda de la mejor unidad (Best Matching Unit) [1], la cual se realiza inspeccionando a cada uno de los vectores para determinar cuál es el de menor distancia al estímulo asociado. Este desarrollo requiere que se realice un número de iteraciones proporcional a la cantidad de vectores de referencias. A pesar que el algoritmo tiene grandes beneficios, podemos observar algunos problemas conocidos. En particular, la dificultad para procesar grandes volúmenes de información, ya que el tiempo de respuesta es elevado producto a la complejidad (Descrita en el Capitulo 2.2) inherente del algoritmo.

Para poder mejorar el tiempo de respuesta de VQ con grandes volúmenes de datos, se propone definir el diseño de un nuevo algoritmo, utilizando VQ y una estructura que enlace los vectores de referencia. Esta estructura podría ser lineal, una malla o en general un grafo. Sin embargo, la estructura adecuada para resolver este problema es la de árbol, porque esta ha sido ampliamente estudiada, y en

determinadas circunstancias se asocian a complejidades temporales eficientes; más específicamente podemos utilizar las ventajas de tiempo que entrega un árbol de búsqueda binario (BST) [15], asociado a la complejidad logarítmica que se aprecia en este tipo de estructura y así poder fusionarlo con el algoritmo VQ. Para poder realizar esto observamos que la relación entre un BST y VQ no es obvia. BST típicamente recurre a claves que son utilizadas para efectuar el recorrido en el árbol. Estas claves requieren de una propiedad matemática conocida como orden parcial, que permite especificar que un elemento es mayor que otro. En el caso de los vectores multidimensionales no es directo el criterio para decir que un vector en el espacio es mayor a otro.

La solución encontrada consiste en utilizar un tipo de estructura de datos conocida como árboles *k-d* [13]. Este tipo de estructura establece árboles de búsqueda binaria para puntos en el espacio multidimensional evaluando una sola dimensión por cada nivel del árbol y ha demostrado utilidad en la búsqueda del vecino más cercano. A su vez, el problema del vecino más cercano es en esencia la generalización del problema de la neurona más cercana en VQ, y por lo tanto, compatible con los requerimientos del problema que estamos estudiando. Nuestro objetivo es adaptar el algoritmo VQ para que procese muchos datos de manera eficiente. Nuestra aproximación consiste en definir un algoritmo jerárquico basado en árboles *k-d*. La idea es que el nuevo algoritmo, realice un recorrido selectivo a través de los distintos niveles de profundidad en el árbol hasta llegar a las hojas, sin la necesidad de verificar la posición de todos los vectores. Como resultado de esta nueva operación esperamos realizar consultas en tiempo logarítmico, acelerando sustancialmente la complejidad de VQ.

La salida del nuevo algoritmo propuesto $KD\text{-}TREE\text{-}VQ$ es radicalmente diferente a los métodos ya definidos en la literatura, los cuales entregan una cantidad de vectores de referencia como resultado. En nuestro caso el árbol generado por el algoritmo en sí es el resultado, ya que cada nodo del árbol contiene sólo un valor único referente a una sola dimensión de los datos. Esto nos permite identificar distintos vectores a medida que se analicen cada uno de los recorridos hasta las hojas del árbol. Pudiendo así analizar las dimensiones de los vectores por separado y que no dependan de múltiple vectores de referencia, ya que VQ analiza todas las dimensiones del vector en conjunto.

Sin embargo, existe un sacrificio en nuestra propuesta y es que a diferencia de

VQ, nuestro algoritmo de búsqueda del vector más cercano es **aproximado**. En referencia a esto, el supuesto estudiado, es que el vector *identificado* como el BMU, en muchos casos coincide con el vector más cercano y de no ser así, se encuentra "razonablemente cerca". Este último concepto es relativo, y para nosotros importa más la posición final de los vectores después del proceso de aprendizaje, que pequeños errores en la identificación del vector más cercano, los cuales podrían verse corregidos a largo plazo.

1.2. Hipótesis

Al aumentar las capacidades del algoritmo cuantización vectorial (VQ) mediante la incorporación de árboles k-dimensionales (K-D trees), es posible:

- Mejorar los tiempos de respuesta en comparación al algoritmo VQ
- Aprender de los datos de manera más eficaz en comparación al algoritmo VQ.

1.3. Objetivo General

Diseño e implementación de una variante del algoritmo VQ, con el fin de estudiar si mejora la complejidad temporal con grandes volumenes de datos en relación al algoritmo original y si realiza un aprendizaje automático de manera más eficiente y eficaz.

1.4. Objetivos Específicos

- Diseño e implementación de una variante del algoritmo VQ que utilice árboles k-dimensionales.
- Evaluar si KD-TREE-VQ posee menor complejidad en tiempo de ejecución en comparación con VQ.
- Evaluar y comparar el desempeño de VQ con el algoritmo propuesto, utilizando conjuntos de datos del mundo real.
- Evaluar la eficiencia del nuevo algoritmo utilizando grandes volúmenes de datos.

1.5. Alcances

- El algoritmo se debe construir con la fusión de los algoritmos VQ y KD-Tree.
- Se analiza 5 datasets de un conjunto de datos estándar pertenecientes al mundo real.
- Se analiza el algoritmo con un conjunto de gran volumen de información.
- Se realiza una implementación de código abierto que utilice R y C++, el cual esté testeado y documentado, cumpliendo con los requerimientos.
- Los resultados experimentales deben estar adecuadamente documentados.

2. Antecedentes

2.1. Introducción

En esta sección se realiza una revisión literaria de todas las componentes que se abordan para el desarrollo del proyecto propuesto. Este apartado se divide en cuatro partes, la primera hace referencia a la descripción del algoritmo de cuantización vectorial y cuáles son sus abordajes. La segunda parte continua con la definición de la estructura de datos del KD-Tree el cual es utilizado para mejorar las funciones del algoritmo de cuantización vectorial. En tercer lugar se discuten métodos alternativos que se han utilizado en estudios posteriores, los cuales tienen una funcionalidad parecida a KD-Tree. Finalmente la cuarta sección del apartado habla sobre el trabajo relacionado con las mejoras y aplicaciones de VQ.

2.2. Cuantización Vectorial

En computación la compresión de datos se refiere a la capacidad de reducir el volumen de datos, representando de forma sucinta los datos originales, con el fin de optimizar el espacio empleado para guardar estos valores o mejorar la velocidad de comunicación. Para realizar esto, existen dos familias de métodos, una en la cual se pierde parte de la información original y otra en la cual se almacena íntegramente la información [6]. La técnica en la cual no existe pérdida de información se denomina LOSSLESS y es utilizada para comprimir archivos que sus datos no pueden ser degradados o perdidos, manteniendo un duplicado exacto del documento original. Esta técnica se aplica en sistemas de compresión como RAR y PNG, entre otros. La segunda técnica, que acepta un porcentaje de pérdida de la información original, con el objetivo de aumentar la compresión de datos se llama LOSSY y no permite la

reconstrucción total de la secuencia de datos original. La técnica LOSSY apunta a eliminar datos que se consideran desechables con la finalidad de reducir el volumen del archivo resultante. La técnica LOSSY se utiliza con gran frecuencia en compresión de archivos de audio como MP3, de vídeo como MP4 y de imágenes como JPG. La mayor ventaja de la técnica LOSSY es que el razón o porcentaje de compresión del archivo resultante es mayor a la de LOSSLESS.

Cuantización vectorial (VQ) fue concebido como un algoritmo de compresión de datos, produciendo una representación sucinta de los datos originales. VQ ha sido utilizado con éxito en gran número de aplicaciones en los campos de compresión y transmisión de datos multimedia, reconocimiento del habla, marca de agua digital, recuperación de imágenes y audio entre otros [8].

VQ es un algoritmo de tres pasos [2]: el primero es la creación del codebook (Conjunto de vectores prototipos de k-dimensiones). Luego, en una segunda etapa se realiza la codificación de los datos donde se aplica el paradigma de "La búsqueda del vecino más cercano", que analiza vectores de k-dimensiones, para poder identificar la densidad de los datos. En esta fase en vez de trasmitir el dato original, lo que se transmite el identificador del prototipo más cercano. Finalmente en la etapa de decodificación, el receptor, que conoce de antemano cual es la identificación de cada uno de los prototipos, puede reconstruir el mensaje original con una pérdida de información aceptable. La codificación de VQ puede ser realizada de diferentes formas dependiendo del uso que se quiera dar.

La filosofía de este algoritmo consiste en separar la densidad de los datos en k particiones, asignando a cada una un único prototipo que representa a esa región. A continuación en 1 se describe el método VQ.

Algorithm 1 VQ ($\mathcal{X}$, K, T, α_i, α_f)

Input:
$\mathcal{X}$, conjunto de datos.
K, número de prototipos.
T, número máximo de iteraciones.
α_i, α_f, factor de aprendizaje inicial y final, respectivamente.
Ouput:
$\mathcal{W}$, conjunto de prototipos analizados.
Method:

1: Inicializar conjunto de prototipos $\mathcal{W}$, seleccionando elementos al azar en $\mathcal{X}$.
2: **for** t ←1 to T **do**
3: $\mathbf{x} \leftarrow$ random-select($\mathcal{X}$) ▷ seleccionar un dato de $\mathcal{X}$ al azar.
4: $\mathbf{w} \leftarrow$ BMU($\mathbf{x}$) ▷ identificar BMU.
5: Migrar el prototipo $\mathbf{w}$, usando regla de actualización de VQ.
6: Actualizar factor de aprendizaje.
7: **return** $\mathcal{W}$.

Se observa que en la primera línea de código se inicializan los prototipos (W) los cuales realizaran una representación de los datos a analizar, estos prototipos se les asignan valores aleatorios de conjunto de datos a evaluar. Luego se realiza el proceso de iterativo. El primer paso de este proceso es seleccionar un pivote aleatorio de los datos a analizar (Línea 3), seguido de esto se buscar el prototipo más cercano al pivote (Línea 4), con la finalidad de migrar este prototipo (Línea 5) utilizando la regla de actualización definida a continuación:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha(\mathbf{x}(t) - \mathbf{w}(t)) \quad (2.1)$$

Esta migración de datos se realiza para encontrar el centro de los datos que se están a analizando. Podemos apreciar en la Figura 2.1 cómo se migra un punto W(t) hasta el punto W(t+1) tomando la regla de actualización de la ecuación (2.1).

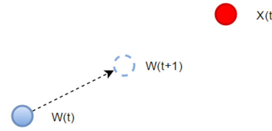


Figura 2.1: Migración del punto W(t) hasta el punto W(t+1) con respecto al punto X(t).

Finalmente se actualiza el factor de aprendizaje α, el cual se utiliza en la ecuación 2.1 con la finalidad de definir cuanto se migrará el prototipo en dirección al pivote seleccionado. El factor de aprendizaje α se inicializa en 1 y disminuye $1/T$ en cada iteración hasta llegar a 0. Este criterio de actualización permite que en las primeras iteraciones los prototipos se migren completamente al pivote seleccionado y la migración disminuya en cada iteración, estableciendo que cada prototipo se establezca en una región diferente, pudiendo realizar una representación de los datos estudiados.

VQ se considera como un potente algoritmo de compresión por su simple arquitectura y alto factor de compresión, pero requiere un enorme tiempo y carga computacional [2][3]. Esto se debe a su complejidad algorítmica $O(t \times k \times d)$ en la que t corresponde al número de iteraciones, k el número de prototipos y d a la dimensionalidad del conjunto de datos.

Por lo anterior, podemos observar que al estudiar problemas que envuelven archivos con un gran volumen de datos, en términos de cardinalidad o dimensión, se traducen a tiempo de ejecución intratables. Es por este motivo que se han realizado varios estudios dedicados a la reducción del tiempo de ejecución y carga computacional de VQ, con el fin de mejorar algunas de las tres etapas centrales del algoritmo, es decir la creación, codificación o decodificación del codebook.

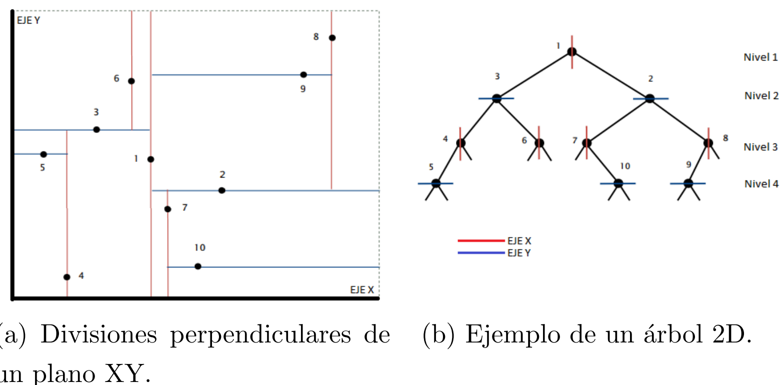
2.3. Árbol K-Dimensional

Un árbol de búsqueda binaria (BST) se define como una estructura de datos en la cual cada nodo tiene dos enlaces que se llaman hijo derecho e hijo izquierdo, los cuales pueden estar con un valor o vacíos. Además, todos los nodos contiene un valor asociado comparable con otro dato y tiene la restricción de que todos los valores menores a la clave raíz del árbol o sub-árbol se almacenan en el lado izquierdo de esta, por ende todos los mayores se almacenan a la derecha del sub-árbol [15].

Este método es utilizado para realizar consulta y ordenamiento de datos en una sola dimensión. Existen varias aplicaciones como la detección de colisiones (empleado en videojuego), búsqueda del vecino mas cercano [15] o consultas de rango ortogonal [4], en las cuales se pueden representar los datos en una estructura de árbol, pero resulta complejo estructurarlo en un BST estándar, ya que tienen datos con más de una dimensión para analizar. Los árboles k-dimensionales son estructuras de datos empleadas para solucionar este problema y poder utilizar una estructura de árbol.

Un árbol k-dimensional (KD ó KD-Tree), es una estructura que realiza divisiones del plano dentro de un espacio euclídeo de k dimensiones, donde k es la cantidad de dimensiones que contiene el conjunto de datos que se analizan, estas divisiones se almacenan en un BST. La estructura del KD-Tree está basada en un BSP-Tree (ver Sección 2.4.1) que realiza divisiones de forma arbitraria al espacio utilizado, a diferencia a KD-Tree que realiza las divisiones de forma perpendicular al plano en el cual se está trabajando, como se representa en la Figura 2.2a.

Al trabajar con KD-Tree en más de una dimensión, las representación de los datos dentro de la estructura de un árbol binario es completamente diferente. Esto se debe a que cada nivel del árbol representa una dimensión completamente distinta a la anterior, no pudiendo comparar los nodos hijos con los nodos padre en la Figura 2.2b podemos apreciar la distribución de un árbol KD de dos dimensiones.



(a) Divisiones perpendiculares de un plano XY.

(b) Ejemplo de un árbol 2D.

Figura 2.2: Divisiones del plano y representación en el árbol.

La construcción de un árbol KD está relacionada al obtener el valor de la mediana de la dimensión en análisis, con esto el algoritmo divide el conjunto de datos de entrada en dos grupos con la misma cantidad de datos [4]. A continuación se muestra

el cálculo de la mediana con una lista de datos.

- Se ordenan los datos de menor a mayor.
- La mediana de un conjunto de datos impar es el dato central.
- La mediana de un conjunto de datos par es el promedio entre los dos datos centrales.

Mediante este cálculo de la mediana, se establece un método recursivo para genera un árbol KD:

Algorithm 2 KDTree ($\mathcal{X}$, $profundidad$, k)

Input:
$\mathcal{X}$, conjunto de datos.
$profundidad$, nivel del árbol.
k, dimensiones.
Ouput:
node, Raíz del árbol o sub-árbol
Method:

1: **if** $\mathcal{X}$ es vació **then**
2:     **return** Null
3: **else**
4:     eje ← $profundidad$ **mod** k
5:     *node* ← *mediana*(*eje*) ▷ Seleccionamos la *mediana* de la dimensión eje como pivote.
6:     *node.izq* ← KDTree($\mathcal{X}/2$, $profundidad + 1$, k) ▷ Pasamos los datos a la izquierda de la mediana
7:     *node.der* ← KDTree($\mathcal{X}/2$, $profundidad + 1$, k) ▷ Pasamos los datos a la derecha de la mediana
8:     **return** *node*

El tiempo de construcción que se emplea en la creación de un árbol KD esta dada por: [4]

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ O(n \log n) + 2T([n/2]) & \text{si } n > 1 \end{cases} \tag{2.2}$$

Esto es por que en la construcción del árbol se debe recorrer todos los nodos para almacenar los valores ($O(n)$), y en cada nodo visitado se tiene que ordenar de menor

a mayor el conjunto de elementos $O(n \log n)$, siendo para cada nivel del árbol $n/2$ datos que el nivel anterior $(2T([n/2]))$, con la excepción de la raíz, ya que se analizar n datos de entrada. Lo que se reduce a:

$$T(n) = O(n \log^2 n) \tag{2.3}$$

El algoritmo de consulta es similar al empleado en un BST, con la gran diferencia que para cada nivel del árbol de debe cambiar la dimensión del dato para realizar la comparación y saber por que mitad del árbol continuar realizando la búsqueda. Esta búsqueda tiene una complejidad igual a la de un BST $O(\log n)$.

2.4. Métodos Alternativos

2.4.1. Árbol de Partición Binaria del Espacio

Un árbol de partición binaria del espacio (BSP-Tree) es un método empleado para realizar divisiones arbitrarias del espacio recursivamente utilizando hiperplanos. Este método se pensó inicialmente para incrementar la eficiencia del renderizado, con el objetivo lograr manejar estructuras complejas en gráficos 3D [9].

Podemos observar en la Figura 2.3, que para utilizar árboles binarios se necesita entender que cada nodo del árbol es la representación de un hiperplano. La raíz almacena la primera división que se realiza en el algoritmo, dividiendo la figura en dos y cada hijo del nodo almacena el lado correspondiente. Por consiguiente por cada hijo se realiza el mismo proceso de división arbitraria en la nueva figura [5].

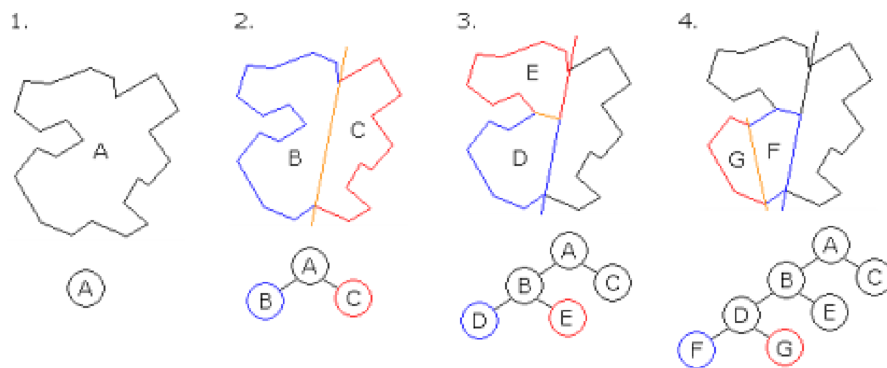


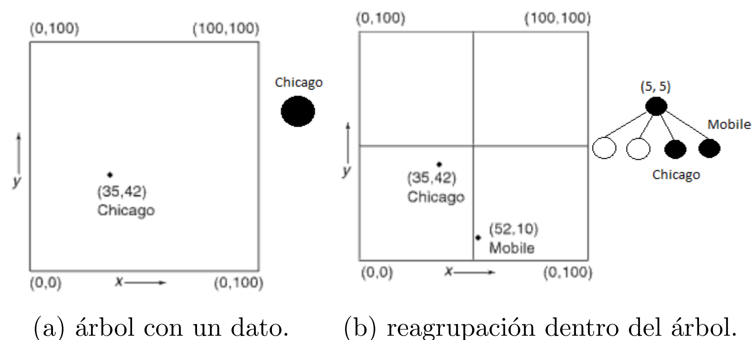
Figura 2.3: Secuencia de construcción de un BSP-Tree.

Al finalizar la construcción del BSP-Tree se puede apreciar que cada hoja del árbol generado contiene una de la figuras creadas por la partición de los hiperplanos.

2.4.2. Árbol Cuaternario

Un árbol cuaternario o QuadTree, descrito en [13], es una estructura de datos que usa una técnica de división espacial, la cual divide un plano en 4 regiones; con el fin de representar datos jerárquicos de dos dimensiones, para generalizar a más dimensiones se utilizan otras estructuras ejemplo el OctaTree. QuadTree genera una estructura de árbol, en el cual cada nodo contiene de 0 a 4 hijos. Para realizar la construcción de un árbol QuadTree revisaremos dos métodos.

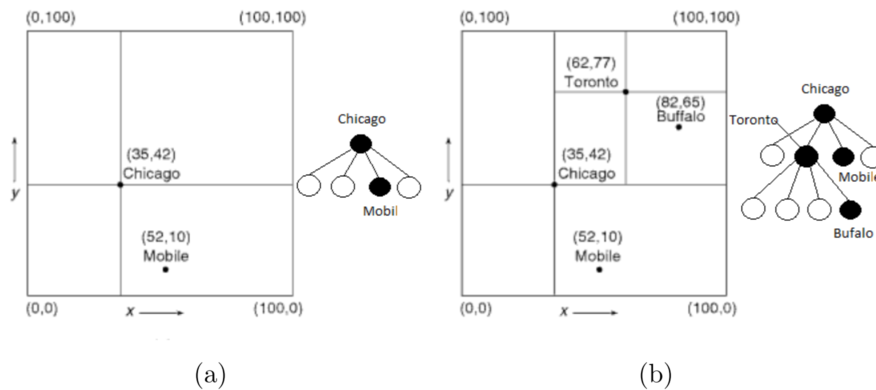
El primero, QuadTree de regiones, se limita a dividir el plano en 4 partes iguales, donde los nodos hojas representan el valor del dato buscado. Podemos ver en la Figura 2.4a que cuando sólo existe un valor dentro del árbol no se realiza división alguna; pero al agregar un segundo valor podemos apreciar cómo se realiza la división del plano. Por ende cada vez que se agrega un valor, si la región donde debe ir el dato está vacía se agrega al nodo que representa esta región, de lo contrario se divide la región a la cual pertenece y se reposiciona los datos en los hijos de la división realizada, ver Figura 2.4b.



(a) árbol con un dato. (b) reagrupación dentro del árbol.

Figura 2.4: QuadTree de regiones.

EL segundo método es QuadTree de puntos, el cual divide al plano en 4 regiones tomando como referencia el punto ingresado. Cuando existe sólo un punto dentro del árbol este se representa de igual forma que el método anterior ver Figura 2.4a. Pero al ingresar un segundo valor, éste se almacena en una de las 4 regiones en la que divide el plano el punto anterior, esto lo podemos ver en la Figura 2.5.



(a) (b)

Figura 2.5: QuadTree de puntos.

2.5. Trabajo Relacionado

2.5.1. TSVQ in Parallel Machine

Un cuantificador vectorial estructurado por árbol (TSVQ) es una reestructuración del agoritmo VQ en un árbol de búsqueda binaria, el cual utiliza índices en los nodos del árbol para representar los prototipos.

Lo que plantea [6] es analizar y mejorar el rendimiento de un (TSVQ) implementándolo en una máquina paralela MP-1 MasPar.

TSVQ in parallel machine realiza de forma normal la estructura de TSVQ con la diferencia que el trabajo de creación y codificación del codebook se realiza en una máquina paralela utilizando el algoritmo de "Instrucción individual, datos múltiples"(SIMD) cuyo control se basa en la secuencia a través de datos de entrenamiento en lugar de la estructura de árbol [6]. Al crear el codebook, generando los prototipos para realizar la codificación, una vez creado el codebook se comienza con la codificación. En la codificación se asignan los índices que representan cada vector del árbol.

El estudio realizado en [6] demuestra que TSVQ es viable en ordenadores paralelos, además de mejorar el proceso de ejecución de TSVQ.

Al comparar los dos algoritmos (KD-TREE-VQ v/s TSVQ in parallel machine) se puede apreciar que ambos utilizan una estructura de árbol, con la diferencia que [6] utiliza una máquina paralela, e índices en el árbol para representar el vector, teniendo que analizar el vector completo para realizar una modificación; en comparación de

KD-TREE-VQ que utiliza valores unidimensionales a procesar, los cuales existen automáticamente en el árbol y se genera de forma secuencial.

TSVQ in parallel machine	KD-TREE-VQ
Realiza la creación y codificación del codebook por separado.	Realiza la creación del codebook a medida que se realiza la codificación.
Trabaja con todas las dimensiones de los vectores.	Trabaja con una dimensión del vector a la vez.
La salida entregada es un árbol con índice los cuales apuntan a los vectores.	La salida es un árbol con valores unidimensionales, que al recorrer el árbol de forma descendente se generan los prototipos.
Aplica un algoritmo SIMD para desarrollar el algoritmo con programación en paralelo.	Se desarrolla de manera secuencial.

Cuadro 2.1: TSVQ in parallel machine vs KD-TREE-VQ.

2.5.2. DTIE-TSVQ

TSVQ utiliza una enorme carga computacional a la hora de realizar una búsqueda (decodificación). En [3] se emplea la fusión entre la eliminación dinámica de la desigualdad triangular (DTIE) y un TSVQ (DTIE-TSVQ) con el fin de realizar una precisión de búsqueda del 100 % y al mismo tiempo reducir la carga computacional que conlleva esta.

DTIE-TSVQ trabaja con esquema de árbol donde cada hoja además de su dato clave contiene cuatro valores candidatos, los cuales son datos de referencia de búsqueda, que representan cuatro grupos cercanos de datos claves, aplicando el algoritmo DTIE. Con esto se analiza si existe otro dato más cercano al dato buscado aplicando el algoritmo DTIE, logrando de esta manera representar una búsqueda más exacta y sin ocupar tantos recursos. Los resultandos entregados por [3] muestran una búsqueda mejor empleada en las pruebas realizadas logrando el 100 % de precisión y reducir la carga computacional considerablemente.

DTIE-TSVQ mejora la decodificación del algoritmo VQ en general, a diferencia de KD-TREE-VQ que se centra en mejorar la creación y codificación, mediante un conjunto de datos en los nodos que se utilizan para mejorar la precisión de la

búsqueda, a diferencia de nuestro algoritmo que no utiliza ese conjunto.

DTIE-TSVQ	KD-TREE-VQ
Mejora la decodificación.	Mejora creación, codificación y decodificación.
Utiliza una estructura de árbol combinada con el Algoritmo TIE.	Utiliza una estructura de árbol.

Cuadro 2.2: DTIE-TSVQ vs KD-TREE-VQ.

2.5.3. HVQ

En [2] se analizo un nuevo algoritmo en base a VQ ya que posee un alto grado de compresión, lo que resulta adecuado para datos multimedia en los dispositivos móviles y redes inalámbricas. En [2], se entabla una forma de mejorar tiempo y recursos comunicacionales altos que utiliza VQ, mediante la creación de un VQ híbrido (HVQ), utilizando una estructura de árbol combinada con un diagrama de VORONOI (VD). El enfoque que se muestra en [2] es en mejorar la creación del codebook con la finalidad de mejorar la calidad de codificación y decodificación de VQ.

Para la construcción del HVQ se aplica la estructura de árbol, luego el DV se utiliza para ampliar el espacio de búsqueda y finalmente se aplica un algoritmo de búsqueda codicioso para buscar la clave óptima en el DV [2]. La estructura de árbol se basa en el análisis de componentes principales (PCA) el cual se utiliza para normalizar las claves obteniendo una matriz de covarianza. El diagrama de VORONOI se genera con la finalidad de ampliar el espacio de búsqueda y disminuir la distorsión de los datos entregados utilizando el PCA. Este diagrama interpreta las relaciones con las regiones vecinas, teniendo en cuenta que cada región del diagrama contiene un prototipo clave; mientras más dimensiones contenga la estructura que se utiliza, el diagrama se relaciona con más vecinos mejorando la búsqueda. Luego, se realiza la búsqueda encontrando rutas a través del árbol que lleguen a un nodo hoja, el que contenga un prototipo inicial vinculado al DV, y finalmente se utiliza el diagrama para mejorar la precisión de la búsqueda.

Las pruebas realizadas en [2] entregaron como resultado que el algoritmo funcionaba mejor utilizando más dimensiones en contraste con aumentar la altura del árbol,

por que ocupa más almacenamiento. HVQ mejoró el rendimiento de VQ además de mejorar la precisión y la carga computacional.

HVQ utiliza una cantidad de dimensiones diferentes, pero sigue estableciendo prototipos de k-dimensiones dentro de los nodos, en contraste con KD-TREE-VQ, que para cada nodo utiliza valores unidimencionales. Además HVQ se enfoca sólo en mejorar la búsqueda de VQ utilizando el diagrama de VORONOI, en diferencia de KD-TREE-VQ que busca mejorar la complejidad del tiempo de construcción y búsqueda, implementando un árbol k-dimensional.

HVQ	KD-TREE-VQ
Trabaja con todas las dimensiones de los vectores.	Trabaja con una dimensión del vector.
Utiliza el DV en conjunto con la estructura de árbol.	Utiliza solo la estructura de árbol.
Mejora la decodificación.	Mejora la creación, codificación y decodificación.

Cuadro 2.3: HVQ vs KD-TREE-VQ.

2.5.4. CTAS-ISOC

En [16] se establece el estudio de un nuevo algoritmo para mejorar las capacidades de VQ. El algoritmo del esquema de asignación de árbol de codificación con una codificación de orden de búsqueda mejorada (CTAS-ISOC) se basa en una comparación y mejora del algoritmo de codificación de orden de búsqueda (SOC) el cual se utiliza para explotar la correlación entre bloques en el dominio de índice en lugar de el dominio de píxeles [16]. CTAS-ISOC mejora el rendimiento de SOC mediante la incorporación de un análisis de correlación de bloques vecinos incorporando patrones de par izquierdo y par superior.

CTAS-ISOC trabaja en dos fases, primero genera la creación del codebook mediante el algoritmo de Linde-Buzo-Gray (LBG) [7] y después la codificación de los datos que se realiza en tres bloques: el bloque uno se refiere a la asignación de código de índice vecino (NICA), el cual se utiliza para verificar si el índice que se está analizando es el mismo que uno de sus vecinos. De no haber coincidencia se utiliza el bloque dos, codificación de orden de búsqueda de pares izquierdos (LSOC), donde se

comparan los índice par izquierdo o el bloque tres codificación de orden de búsqueda de pares superiores (USCO), en el que se compara el índice par derecho con los pares de índices previos en una ruta de búsqueda predefinida. Estos dos últimos bloques logran una mejor codificación que SOC. Luego de encontrar el índice indicado se reconstruye el mapa de índices. El algoritmo CTAS-ISOC logra un mejor rendimiento de compresión que el algoritmo SOC.

EL algoritmo CTAS-ISOC trabaja con la creación y luego codificación en un ámbito separado a diferencia del algoritmo propuesto KD-TRRE-VQ, que a medida que se realiza la codificación el codebook se va creando, además el algoritmo CTAS-ISOC utiliza varios algoritmos en conjunto para mejorar el tiempo de ejecución, en comparación de KD-TREE-VQ que utilizamos sólo la combinación de VQ y KDTREE.

CTAS-ISOC	KD-TREE-VQ
Realiza la creación y codificación del codebook por separado.	Realiza la creación del codebook a medida que se realiza la codificación.
Trabaja con todas las dimensiones de los vectores.	Trabaja con una dimensión del vector.
Utiliza combinaciones de distintos algoritmos.	Es un VQ híbrido entre VQ y KD-Tree.
Utiliza una tabla de índices para la codificación.	Utiliza un árbol con valores unidimensionales, los cuales representan la codificación.

Cuadro 2.4: CTAS-ISOC vs KD-TREE-VQ.

2.5.5. MCTAS

En [8] se establece el estudio de un nuevo algoritmo, esquema de asignación de árbol de codificación modificado (MCTAS), para mejorar las capacidades de VQ. El algoritmo MCTAS se basa en una comparación y mejora del algoritmo CTAS-ISOC, el cual se utilizó para mejorar la codificación de VQ mediante un análisis de correlación de los bloques vecinos utilizando los patrones de par izquierdo y par superior en el dominio del índice [8]. MCTAS está orientado de mejorar el rendimiento de CTAS-ISOC empleado la técnica de codificación de tabla de índice dinámico

(DICT), que utiliza la correlación de los bloques vecinos en una tabla de índices construida de forma temporal dentro de la ejecución del algoritmo.

El algoritmo se trabaja en dos fases, la primera es la creación del codebook, la cual se realiza con el algoritmo de LBG [7] y como segunda fase, utiliza el algoritmo MCTAS, que se enfoca en tres aspectos. El primer aspecto utiliza el algoritmo CTAS con el fin de seleccionar un esquema apropiado para realizar la codificación de los índices. El segundo aspecto aplica la técnica NICA que se utiliza para verificar si el índice el cual se está analizando es el mismo que uno de sus vecinos. Por último, se emplea el algoritmo DICT, con el fin de mejorar la tabla de índices original. El algoritmo MCTAS baja la duración de la complejidad en comparación con CTAS-ISOC.

MCTAS trabaja con la creación y luego codificación en un ámbito separado a diferencia de KD-TREE-VQ que a medida que se realiza la codificación el codebook se va creando. Por otro lado, el algoritmo MTCAS utiliza varios algoritmos en conjunto para mejorar el tiempo de ejecución, a comparación de KD-TREE-VQ que emplea sólo la combinación de VQ y KDTREE.

MCTAS	KD-TREE-VQ
Realiza la creación y codificación del codebook por separado.	Realiza la creación del codebook a medida que se realiza la codificación.
Trabaja con todas las dimensiones de los vectores.	Trabaja con una dimensión del vector.
Utiliza combinaciones de distintos algoritmos.	Es un VQ híbrido entre VQ y KD-Tree.
Utiliza una tabla de índices para la codificación.	Utiliza un árbol con valores unidimensionales, los cuales representan la codificación.

Cuadro 2.5: MCTAS vs KD-TREE-VQ.

2.5.6. RVQ-Lloyd y RVQ-KD

En [14], se analiza cómo reducir la complejidad de búsqueda dentro de los canales de redes inalámbricos con el fin de mejorar la rapidez de respuesta sin comprometer el rendimiento. Para esto se define un codebook aleatorio mediante un Random VQ

(RVQ), el cual requiere una búsqueda exhaustiva para localizar la entrada seleccionada. Este codebook es el punto de comparación para definir los nuevos enfoques, los cuales se basan en mejorar RVQ utilizando el algoritmo de Lloyd y KD-Tree. Estos estudios se realizan en dos modelos de sistema multi-entrada multi-salida (MIMO) y acceso múltiple por división de código (CDMA) [14].

Primero se define un nuevo codebook mediante la intervención del algoritmo de Lloyd (RVQ-LLoyd), que divide el codebook en dos y se inserta un pivote en la raíz, luego la mitad izquierda del codebook se analiza en el sub-árbol izquierdo y la mitad derecha en el sub-árbol derecho. Esta secuencia se realiza hasta que la división del codebook deje todos los nodos del árbol con un vector definido y no se puede seguir sub dividiendo el codebook. Este proceso genera un árbol no equilibrado ya que la división del codebook no es equitativa.

En el segundo estudio se utiliza KD-Tree (RVQ-KD), que al implementarse también produce un árbol no equilibrado. Esto sucede ya que se agrupan los vectores analizando dimensiones distintas en cada etapa del árbol. Para la construcción del árbol RVQ-KD, primero se analiza la mediana para el primer elemento de todos los vectores (análisis de la primera dimensión) y se agrega a la raíz el elemento más cercano al pivote encontrado. Luego se divide el codebook en dos, los cuales se utilizan para los sub-árboles izquierdo y derecho de la raíz. Luego calculamos los pivotes para los nodos hijos, pero esta vez utilizamos la segunda dimensión para hacer la búsqueda y separación, al encontrar el pivote se realiza lo mismo que en la raíz. Este proceso se realiza para cada dimensión y si no quedan dimensiones que analizar se parte con la primera dimensión inicializando el ciclo. El proceso finaliza una vez que todos los elementos del codebook estén almacenados en un nodo del árbol. RVQ-KD mejora la rapidez de ejecución al analizar sólo una dimensión del vector por nivel. Una variación del RVQ-KD es realizar los mismos pasos que se establecieron con anterioridad, con la diferencia de que se utiliza una matriz de covarianza para determinar el pivote.

Para la simulación de CDMA y MIMO, los algoritmos de RVQ-Lloyd y RVQ-KD tienen un rendimiento cercano a RVQ. Lo que conlleva que en algunos casos es mejor o en otros peor, manteniendo la puerta abierta a nuevos análisis y esquemas para mejorar el rendimiento de RVQ.

El algoritmo de RVQ-Lloyd y RVQ-KD sólo reestructura el codebook dentro de un árbol, con la diferencia que RVQ-KD analiza los vectores en una dimensión a la vez

y utiliza una matriz de covarianza. KD-TREE-VQ trabaja de igual forma analizando una dimensión a la vez con la diferencia que en la construcción del árbol se utiliza la migración de los puntos del árbol y se construye el codebook automáticamente en el árbol.

<table>
<tr><th>RVQ-LLoyd</th><th>RVQ-KD</th><th>KD-TREE-VQ</th></tr>
<tr><td colspan="2">Busca mejorar la búsqueda de VQ.</td><td>Busca mejorar la búsqueda y creación de VQ.</td></tr>
<tr><td>Analiza los vectores completos para la creación del árbol.</td><td colspan="2">Analiza una dimensión a la vez para la creación del árbol.</td></tr>
<tr><td colspan="2">Utilizado para mejorar comunicación de redes inalámbricas MIMO y CMDA.</td><td>Utilizado para mejorar tiempo ejecución de datos en general.</td></tr>
</table>

Cuadro 2.6: RVQ-LLoyd y RVQ-KD vs KD-TREE-VQ.

2.5.7. FMIR

EL objetivo de [10] es presentar el método de recuperación difusa de imágenes médicas (FMIR), con el fin de encontrar una cantidad de imágenes médicas con gran similitud, en referencia a una imagen de entrada [10]. FMIR utiliza a VQ con fin de crear el codebook de las imágenes médicas que se consultarán, trasformando las imágenes a vectores. Luego se crean firmas a cada imagen para realizar la comparación de imágenes similares y área de interés dentro de la imagen y se organizan en una estructura de árbol. Finalmente, mediante una imagen médica de entrada se busca una lista de imágenes con la mayor similitud posible para analizar y comparar médicamente.

El algoritmo de VQ se utiliza principalmente para crear el codebok y convertir las imágenes a vectores. Este proceso se realiza sólo una vez, ya que es el que presenta la mayor complejidad dentro del método; luego se utiliza para convertir la imagen de entrada en un vector para poder ser comparada con las demás imágenes, además de la creación de su firma.

Este método se compara con el algoritmo de distancia de compresión normalizada (NCD), que utiliza del mismo modo el algoritmo de VQ, pero a diferencia de FMIR puede crear lista de casos similares respecto a las imágenes, pero no puede encontrar

el área de interés respecto a la imagen. El estudio realizado muestra que FMIR arroja mejores imágenes semejantes que NCD.

2.5.8. Cuadro Comparativo

Podemos apreciar una tabla comparativa en función de todos los algoritmos antes descritos. A continuación se describen las columnas de la tabla.

- Árbol: Utiliza una estructura de árbol.
- Codificación: Se centra en mejorar la codificación más que la decodificación.
- Decodificación: Se centra en mejorar la decodificación más que la codificación.
- Estructuras: Utiliza más de una estructura para realizar el estudio.
- N° D: Número de dimensiones que analiza dentro del algoritmo.
- Referencia: Referencia del algoritmo dentro del documento.

Algoritmos	Árbol	Codificación	Decodificación	Estructuras	N° D	Referencia
KD-TREE-VQ	✓	✓	✓		1	3
TSVQ-Parallel	✓	✓		✓	d	2.1
DTIE-TSVQ	✓		✓	✓	d	2.2
HVQ	✓		✓	✓	2 a d	2.3
CTAS-ISOC		✓		✓	d	2.4
MCTAS		✓		✓	d	2.5
RVQ-LLOYD	✓	✓	✓		d	2.6
RVQ-KD	✓	✓	✓	✓	1	2.6
FMIR				✓	1	2.5.7

Cuadro 2.7: Cuadro comparativo general.

3. Cuantización Vectorial utilizando Árboles K-Dimensionales

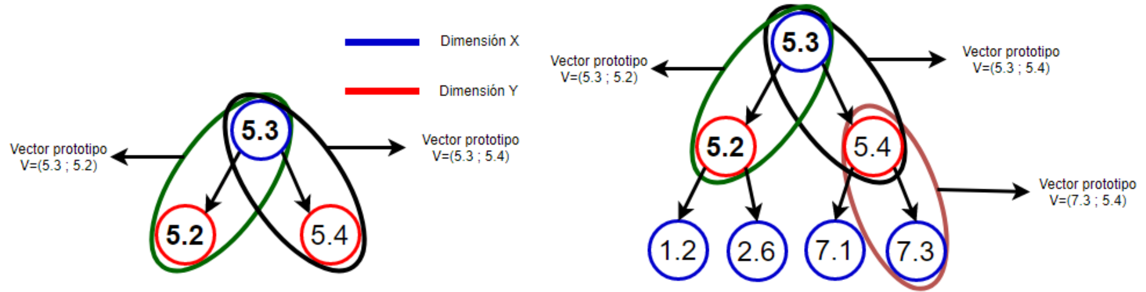
3.1. Introducción

En esta sección se explica la construcción de KD-TREE-VQ.

3.2. KD-TREE-VQ 2.0

Para realizar la unión entre la estructura de árbol k-dimensional y el compresor de datos VQ (KD-TREE-VQ), se estableció un estudio previo para saber como representar un codebook dentro del árbol utilizando valores claves unidimensionales en los nodos. Esto se logró mediante la combinación de los valores siguiendo el camino desde la raíz del árbol hasta una hoja (Figura 3.1a), cuando la altura del árbol es igual a la cantidad de dimensiones las cuales se están analizando ($h = d$). Esto presentó nuestra primera restricción dentro del método KD-TREE-VQ, donde h no puede ser menor que d, con el fin de interpretar los prototipos de forma correcta y coherente. Lo antes planteado muestra que si $h = d$ la cantidad de prototipos representada por el árbol será igual a la cantidad de hojas que tenga la estructura, esto se define como 2^{h-1}. Por otro lado si queremos definir más prototipos para realizar el análisis, se debe aumentar la altura del árbol, con él fin de obtener más combinaciones de valores. Esta composición de datos se realiza de forma secuencial a medida que se desciende en una rama por el árbol (Figura 3.1b), obteniendo como resultado una cantidad de prototipos igual a $\sum_{i=d-1}^{h-1} 2^i$, lo que representa la cantidad de nodos desde el primer nivel que simboliza la última dimensión, hasta el último

nivel del árbol. Estos cálculos sirven para cualquier $d \geq 2$, lo que modifica la primera restricción dejando $h \geq d$.



(a) Árbol de altura 2; datos de 2 dimensiones; prototipos $= 2^{h-1} = 2$.

(b) Árbol de altura 3; datos de 2 dimensiones; prototipos $= \sum_{i=d-1}^{h-1} 2^i = 6$.

Figura 3.1: Representación de prototipos en un árbol k-dimensional.

Al tener una representación unidimensional, en la estructura del KD-TREE-VQ se puede realizar migraciones de un conjunto de vectores prototipos en un mismo ciclo, analizando sólo una dimensión de los vectores y disminuyendo la búsqueda de BMU considerablemente. Esto se debe a la estructura del KD-TREE-VQ, ya que realiza la búsqueda en tiempo logarítmico $\log_2 n$ donde n es la cantidad de datos que contiene el árbol.

Para realizar la creación del árbol, se optó por representar la estructura mediante un arreglo, el cual contiene los valores a analizar. El Algoritmo 3 representa los pasos principales de cómo se construye el árbol.

Todos los nodos están inicialmente desactivados, con el fin de activarlos en el momento de asignarles un valor y saber si seguir recorriendo el árbol por esa rama, ya que al no estar con un dato asociado, ningún nodo el cual desciende de él debe estar activado (Líneas 1-2). Siguiendo con el algoritmo, se comienza a realiza las iteraciones definidas para hacer las migraciones; cuando comience una ciclo se define i (posición del nodo que se está analizando) como la raíz (posición 0), luego se selecciona un dato al azar del conjunto de datos que se quiere analizar o comprimir (Líneas 5-6). Después, se inicia el primer análisis, en el cual se recorre una rama del árbol de forma descendente buscando el BMU (Algoritmo 4). Una vez terminado el primer estudio se prosigue con el análisis secundario, en donde se efectúa la migración de los nodos de forma ascendente desde la última hoja visitada hasta llegar a la raíz

(Algoritmo 5). Por último se actualiza el factor de aprendizaje del algoritmo y se inicializan los valores auxiliares para proceder con el análisis de otra rama del árbol (Líneas 9-10). Al finalizar todas las iteraciones requeridas, se entrega como resultado la lista A que representa el árbol con todos los nodos creados y migrados.

Algorithm 3 KD-TREE-VQ ($\mathcal{X}$, h, T, α_i, α_f)

Valores de entrada:
$\mathcal{X}$, Colección de datos $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}, \mathbf{x_i} \in \mathbb{R}^d$.
h, Altura del árbol.
T, Número máximo de iteraciones.
α_i, α_f, Factor de aprendizaje inicial y final, respectivamente.
Datos de salida:
A, arreglo $\{a_1, a_2, \ldots, a_{2^h-1}\}$, $a_i \in \mathbb{R}$, Contiene puntos de corte del árbol KD binario completo.
Method:

1: **for** b ←1 to $|A|$ **do** ▷ Para cada nodo del árbol.
2: B[b] ← false ▷ Nodo inicialmente está desactivado.
3: ▷ La cantidad de elementos del arreglo A es la cantidad de nodos que posee un árbol binario completo de altura h.
4: **for** $t \leftarrow 1$ to T **do**
5: **i** ←0 ▷ Posición en el árbol que actualmente está siendo analizado.
6: $\mathbf{x}$ ← random-select($\mathcal{X}$) ▷ Seleccionar un dato de $\mathcal{X}$ al azar.
7: AnalisisPrimario() ▷ Recorremos el árbol desde la raíz hasta una hoja (Algoritmo 4).
8: AnalisisSecundario() ▷ Recorremos el árbol desde la última hoja analizada hasta la raíz (Algoritmo 5).
9: α_t ← decay(α_i,α_f,t, T) ▷ Actualizar el factor de aprendizaje.
10: Inicializar$MMaux()$ ▷ Inicializamos la matriz auxiliar para recorrer otra rama del árbol.
11: **return** A ▷ Retorna el arreglo con los datos migrados.

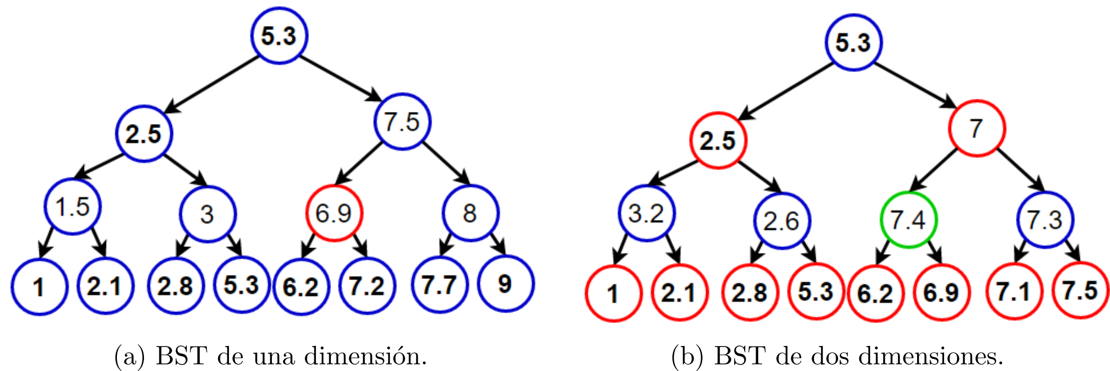
Podemos apreciar mediante el Algoritmo 3 que, teóricamente, la complejidad asociada a la construcción del árbol está dada por:

$$t * 2 * h \tag{3.1}$$

Ya que t es la cantidad de iteraciones que utiliza el algoritmo para construir el árbol, h es la altura asociada al árbol, y el 2 porque en cada iteración se recorre desde la raíz a la hoja y desde la hoja hasta la raíz.

3.2.1. Análisis Primario

Para realizar el análisis primario se emplearon variables auxiliares. Estas permiten tener un control estructural dentro del árbol, con el propósito de no romper el esquema de un BST. Estas variables hacen referencia a una nueva restricción la cual está implícita en un BST, pero es más difícil aplicarla dentro de un árbol donde se migran los valores con cada iteracion. Para un nodo del BST todos los nodos hijos, hermanos y ancestros izquierdos deben ser menores al nodo que estamos analizando y para los hijos, hermanos y ancestros derechos tienen que ser mayores. Esto podemos verlo en la Figura 3.2a, en que al lado derecho del nodo con valor 6,9 tiene desde el nodo 7,2 hasta el 9, los cuales todos son mayores que él y al lado izquierdo desde el 1 hasta el 6,2 los cuales todos son menores. Para KD-TREE-VQ, esto cambia ligeramente, ya que los nodos que se tienen que revisar deben ser sólo los hijos y ancestros que posean la misma dimensión, el sub-árbol hermanos no representan restricción ya que no analizan el mismo plano.



(a) BST de una dimensión. (b) BST de dos dimensiones.

Figura 3.2: Definición de límites superior he inferior

Estudiando la Figura 3.2 podemos observar que cada nodo tiene un nodo menor el cual esta más cerca de su valor. En la Figura 3.2b, el nodo raíz 5,3 su valor menor

más próximo o limite inferior sería el 3,2. Mediante esto podemos analizar que si se debe migran el nodo raíz no podemos disminuirlo en un valor inferior a 3,2. Esto mismo sucede para el valor mayor más próximo o límite superior, que en este caso sería 7,4, no pudiendo aumentar el valor final de la migración en un valor superior a 7,4. Con este estudio tenemos nuestra segunda restricción en el algoritmo KD-TREE-VQ, "al realizar la migración de un nodo esta no puede salirse de los límites superiores he inferiores especificados". Estos parámetros en el algoritmo KD-TREE-VQ se representan mediante una matriz de $n \times 4$; en la cual se almacenan los límites y de que nodo provienen. A continuación se definen el algoritmo de análisis primario y las variables auxiliares empleadas.

- MM, Matriz $n \times 4$, Guarda el valor de los límites superiores e inferiores de todos los nodos en el árbol y de qué nodo provienen los valores.
- $MMaux$, Matriz $d \times 4$, Guarda el valor de los límites superiores e inferiores de la rama que se analiza en el árbol y de qué nodo provienen los valores.

Este análisis recorre de forma descendente el árbol, buscando la rama asociada al BMU. Al iniciar el ciclo lo primero que hace es obtener la dimensión que debe estudiar, luego actualiza el valor de los límites inferiores y superiores del nodo (Líneas 4-9). Esto se hace si y sólo si la cantidad de iteraciones que lleva es mayor a la cantidad de dimensiones que se están analizando, ya que los valores auxiliares ($MMaux$), no se han inicializado. Luego se verifica si el nodo que se está analizando tiene un valor asociado. De no tenerlo se le asigna el valor aleatorio de la dimensión correspondiente, se activa y se finaliza el descenso por la rama en esta posición para realizar el análisis secundario (Líneas 10-15). De no ser este el caso se continua con el algoritmo. Se decide por que rama del árbol continuar (Líneas 16 o 21), si se continua por el sub-árbol izquierdo se actualiza el límite superior auxiliar. Si se continua por el sub-árbol derecho se actualiza el límite inferior auxiliar y se continua con el siguiente ciclo. Al finalizar la última iteración i queda con un valor hijo inválido, ya que supera la altura del árbol y obtiene un valor no existente, esto se corrige en la primera iteración del análisis secundario.

Algorithm 4 AnalisisPrimario()

Method:

1: **for** j ←0 to $h-1$ **do** ▷ Recorrer el árbol desde las raíz hasta las hojas.
2: **k** ← j mod d ▷ Dimensión a procesar con respecto a la profundidad actual del árbol.
3: **if** j ≥ d **then** ▷ Los d primeros niveles no requieren actualización en este análisis.
4: **if** (MM[i][0] < $MMaux$[k][0] AND $MMaux$[k][0]< A[i]) OR MM[i][2] = $MMaux$[k][2] **then**
5: MM[i][0] ← $MMaux$[k][0] ▷ Se actualiza el límite inferior para el nodo.
6: MM[i][2] ← $MMaux$[k][2] ▷ Se define el nodo que posee el límite inferior.
7: **if** (MM[i][1] > $MMaux$[k][1] AND $MMaux$[k][1]> A[i]) OR MM[i][3] = $MMaux$[k][3] **then**
8: MM[i][1] ← $MMaux$[k][1] ▷ Se actualiza el límite superior para el nodo.
9: MM[i][3] ← $MMaux$[k][3] ▷ Se define el nodo que posee el límite superior.
10: **if** B[i] = false **then** ▷ Si el nodo no está activado se ingresa.
11: A[i] ← $\mathbf{x}_k$ ▷ Asignamos la componente k del vector al nodo actual.
12: B[i] ← true ▷ Activa nodo actual del árbol.
13: i ← 2*i + 1 ▷ Asignamos la posición del hijo izquierdo de la posición actual.
14: j+1 ▷ Aumentamos el contador para inicial el ascenso en la dimensión correcta.
15: **break** ▷ Finalizamos el análisis primario.
16: **if** A[i] > $\mathbf{x}_k$ **then** ▷ Si la condición es válida se prosigue por el sub-árbol izquierdo.
17: **if** A[i]< $MMaux$[k][1] **then**
18: $MMaux$[k][1] ← A[i] ▷ Actualiza el límite superior para el nodo que viene.
19: $MMaux$[k][3] ← i ▷ Se define el nodo que posee el límite superior.
20: i ← 2*i + 1 ▷ Asignamos la posición del hijo izquierdo de la posición actual.
21: **else** ▷ Si la condición es válida se prosigue por el sub-árbol derecho.
22: **if** A[i] > $MMaux$[k][0] **then**
23: $MMaux$[k][0] ← A[i] ▷ Actualiza el límite inferior para el nodo que viene.
24: $MMaux$[k][2] ← i ▷ Se define el nodo que posee el límite inferior.
25: i ← 2*i+ 2 ▷ Asignamos la posición del hijo derecho de la posición actual.

3.2.2. Análisis Secundario

El análisis secundario se enfoca en realizar las actualizaciones de los límites inferiores y superiores después de que se realice las migraciones. En la primera iteración

se corrige el valor de i inválido tomando la posición del padre. Para todas los ciclos siguientes, esta operación se refiere a seleccionar el nodo siguiente de la rama (Línea 4). Luego se obtiene la dimensión que se debe estudiar. Realizado esto, se actualizan los límites superiores e inferiores del nodo (Líneas 7-17). Una vez actualizados los límites del nodo se prosigue a realizar la migración de este de ser posible, esto se detalla en el Algoritmo 6. Por último se actualizan los límites superiores e inferiores auxiliares, para actualizar los límites del nodo siguiente a ser evaluado.

Algorithm 5 AnalisisSecundario()

Method:

1: j- = 1 ▷ Establecemos j en la posición de la hoja para recorrer el árbol de forma ascendente.
2: jj = j ▷ Inicializamos la variable jj para analizar límites inferiores y superiores.
3: **for** j to 0 **do** ▷ Recorre de la hoja hasta la raiz
4: i = (i-1)/2
5: **k** ← j *mod* d ▷ Dimensión a procesar con respecto a la profundidad actual del árbol.
6: **if** j ≤ jj-d **then** ▷ Los d últimos niveles no requieren actualización en este análisis.
7: **if** (MM[i][0] < $MMaux$[k][0] AND $MMaux$[**k**][0]< A[i]) OR MM[i][2] = $MMaux$[**k**][2] **then**
8: MM[i][0] ← $MMaux$[**k**][0] ▷ Se actualiza el límite inferior para el nodo.
9: MM[i][2] ← $MMaux$[**k**][2] ▷ Se define el nodo que posee el límite inferior.
10: **if** (MM[i][1]< $MMaux$[**k**][1] AND $MMaux$[**k**][1]< A[i]) OR MM[i][3] = $MMaux$[**k**][3] **then**
11: MM[i][1] ← $MMaux$[**k**][1] ▷ Se actualiza el límite superior para el nodo.
12: MM[i][3] ← $MMaux$[**k**][3] ▷ Se define el nodo que posee el límite superior.
13: **else**▷ Si están definidos los límites de los últimos d niveles se actualizan con el nuevo valor del nodo que definió el límite
14: **if** MM[i][2] ≠ -1 **then**
15: MM[i][0] ← A[MM[i][2]] ▷ se actualiza el límite inferior.
16: **if** MM[i][3] ≠ -1 **then**
17: MM[i][1] ← A[MM[i][3]] ▷ se actualiza el límite superior.
18: Migracion() ▷ Se analiza si se migra el punto (algoritmo 6).
19: **if** A[i]> $MMaux$[k][0] **then**
20: $MMaux$[k][0] ← A[i] ▷ actualizamos el límite inferior para el nodo que viene.
21: $MMaux$[k][2] ← i ▷ Se define el nodo que posee el límite inferior.
22: **if** A[i]< $MMaux$[k][1] **then**
23: $MMaux$[k][1] ← A[i] ▷ actualizamos el límite superior para el nodo que viene.
24: $MMaux$[k][3] ← i ▷ Se define el nodo que posee el límite superior.

3.2.3. Migración de un Nodo

Para realizar las migraciones de los nodos existen tres casos válidos. En el primer caso se revisa si la migración del nodo no se puede realizar dentro de los límites inferior y superior del nodo (Líneas 2-3). Si el nodo migrado no respeta uno de estos valores, se estudia la probabilidad de que los límites estén inicializando sus valores, en este caso no se migra y se prosigue con el siguiente nodo (Líneas 5-6). Si los límites están bien definidos se prosigue a analizar el siguiente caso. Segundo caso, cuando el valor de la migración es menor que el límite inferior del nodo que se está evaluando, se establece el valor de la migración que realizara el nodo con al valor del límite inferior. Para el caso tres, donde la migracón toma un valor mayor al límite superior del nodo evaluado, se establece el valor de la migración que realizara el nodo evaluado con al valor del límite superior (Líneas 7-11).

Algorithm 6 Migracion()

Method:

```
1:  migracion ← update-rule(A[i],x_k,α_t)     ▷ Valor de la migración que se debe realizar al nodo.
2:  if MM[i][0] ≤ migracion ≤ MM[i][1] then
3:      A[i] ← migracion                       ▷ Si la migración esta dentro de los límites, se realiza.
4:  else
5:      if MM[i][0] > MM[i][1] then
6:          Fin del metodo.    ▷ No se migra el nodo, por que los parámetros se están inicializando
7:      else
8:          if migracion < MM[i][0] then   ▷ Si la migración realizada es menor el límite inferior,
9:              A[i] ← MM[i][0]        ▷ Se establece el valor del nodo con el valor del límite inferior.
10:         else                           ▷ Si la migración realizada es mayor a límite superior,
11:             A[i] ← MM[i][1]        ▷ Se establece el valor del nodo con el valor del límite superior.
```

3.3. Bosque de Búsqueda en KD-TREE-VQ

Con el fin de evaluar una mejora en la exploración de un árbol de múltiples dimensiones, se construyó un bosque de búsqueda, el cual se centra en dividir la creación y exploración de un codebook de n dimensiones en x sub-arboles de n/x dimensiones, donde x es la cantidad de arboles que se utilizan para el bosque de árboles. Las dimensiones se distribuyen en forma aleatoria, dejando n/x dimensiones en cada árbol las cuales no se repiten. En la Figura 3.3 podemos observar un árbol de 4 dimensiones, construido para una búsqueda normal.

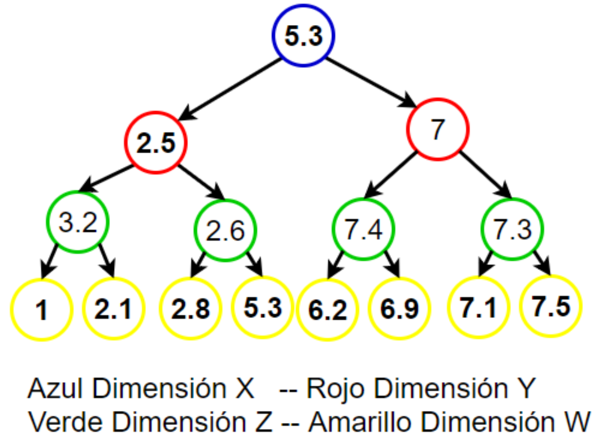
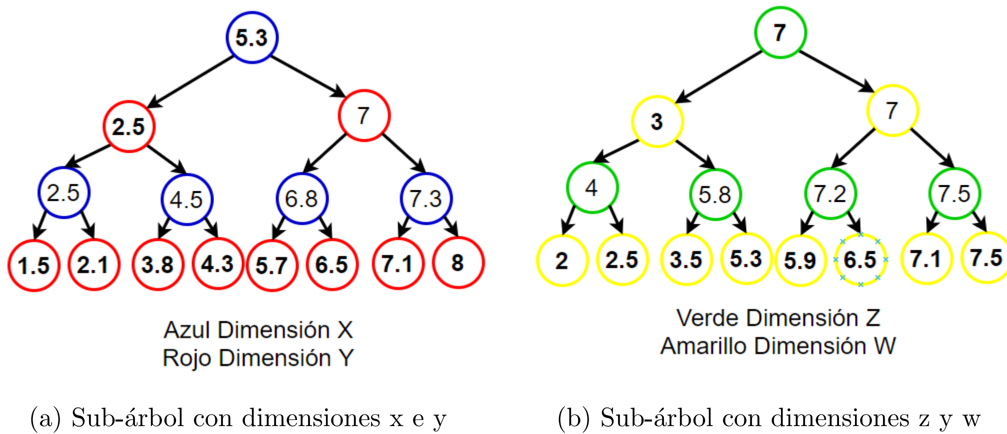


Figura 3.3: Árbol completo a sub dividir de 4 dimensiones

En la Figura 3.4 podemos apreciar la diferencia de construcción para un bosque de búsqueda, generado a partir de 2 sub-árboles, obteniendo sub-árboles de n/x dimensiones (4/2=2).



(a) Sub-árbol con dimensiones x e y

(b) Sub-árbol con dimensiones z y w

Figura 3.4: Bosque de búsqueda

Una vez creado el bosque de búsqueda, para realizar una exploración se analizan todos los árboles generados, utilizando una búsqueda normal en cada sub-árbol y al final de la ejecución combinando las soluciones entregadas por cada uno, obteniendo un vector de n dimensiones.

4. Análisis de los Resultados

4.1. Introducción

En esta sección se desarrolla el análisis experimental del algoritmo KD-Tree-VQ, con el objetivo de obtener resultados concretos referentes al tiempo de ejecución empleados para distintos archivos de prueba y la calidad de la respuesta final. Todo este análisis es realizado en comparación con el algoritmo VQ. La sección de análisis se divide en dos etapas: la primera describe el plan de pruebas que se utilizó para el análisis del KD-TREE-VQ. La segunda etapa muestran los resultados experimentales y sus análisis de los dos algoritmos en referencia a lo descrito en la sección anterior.

4.2. Plan de Pruebas

El plan de pruebas define los aspectos que se evaluaron y cómo se realizan estas evaluaciones. Para realizar la evaluación de KD-TREE-VQ se definieron dos aspectos a estudiar: la cantidad de tiempo que se demora en entregar una respuesta y la calidad de la respuesta entregada. Estos aspectos se compararon a los resultados entregados por VQ.

Para comparar la calidad de las respuestas y tiempo de ejecución entre KD-TREE-VQ vs VQ, se seleccionaron cinco archivos de prueba, los cuales fueron normalizados antes de la ejecuciones, estableciendo los valores de los datos entre 0 y 1, con el fin de obtener resultados con magnitudes pequeñas y las cuales se pueden analizar con facilidad. Los archivos fueron obtenidos de las plataformas **Tudefl**[17] y **UCI Machine Learning Repository**[11]. Estos archivos se analizaron por los dos algoritmos, entregando los siguientes datos de análisis.

Error Cuadrático medio (RMSE). Es una medida estadística que se utiliza para medir la diferencia entre dos valores dentro de una muestra o población en un modelo. En este estudio se utiliza para analizar el error promedio de los datos buscados dentro del codebook generado por VQ y KD-TREE-VQ, la fórmula se describe a continuación:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (x_i - y_i)^2}{n}} \tag{4.1}$$

donde n es la cantidad de elementos a buscar, x_i el elemento i-ésimo a buscar y y_i es el BMU del elemento buscado.

Desviación estándar (S). Es una medida estadística la cual es utilizada para medir la dispersión de un conjunto de datos en relación al valor promedio de los datos analizados. En este estudio se utiliza con el fin de medir la dispersión del error promedio de los datos buscados en el codebook. La fórmula se describe a continuación:

$$S = \sqrt{\frac{\sum_{i=1}^{n} \left(x_i - \bar{x}\right)^2}{n}} \tag{4.2}$$

donde n es la cantidad de elementos a analizar, x_i es el error o distancia entre el vector buscado y su BMU, y $\bar{x}$ es el promedio de los datos analizados.

Porcentaje promedio de error (PPE). Al estar los datos normalizados, sus valores varían entre 0 y 1, siendo así el valor máximo que puede tomar la magnitud de un vector igual a $\sqrt{d}$. Esto se debe a que todos los valores del vector son igual a 1, y el valor mínimo que puede tomar un vector es igual a 0. Con esto podemos apreciar que la distancia entre el vector de magnitud máxima y mínima es de $\sqrt{d}$. Mediante este análisis, si tomamos el vector máximo como el elemento buscado y el vector mínimo como el BMU el error máximo de búsqueda en el codebook sera de $\sqrt{d}$, lo que observa como el 100 % de error. Mediante esto obtenemos que el PPE para cualquier caso, el cual está dado por:

$$PPE = \frac{RMSE}{\sqrt{d}} \tag{4.3}$$

Mientras menor sea el PPE, mejor es la búsqueda promedio de elementos dentro del codebook.

Como últimos datos de análisis, se entrega el tiempo en segundos que se demoran los algoritmos en generar los codebook entrenados (TC), y la cantidad de tiempo que se demoran en buscar los elementos dentro de este (TD).

Para entregar estos datos de análisis se modificaron los parámetros de entrada de los algoritmos para cada archivo, con el fin de saber cuales eran los mejores parámetros en cada uno. Los distintos parámetros que se modificaron son:

VQ(iteraciones, alfa inicial, alfa final, prototipos, archivo analizado, archivo de búsqueda)

KD-Tree-VQ(iteraciones, alfa inicial, alfa final, altura, archivo analizado, archivo de búsqueda)

En donde las iteraciones son la cantidad de veces que se ejecuta el algoritmo, el alfa inicial es el valor con el cual comienza el factor de aprendizaje, este decae en cada iteración llegando hasta el valor del alfa final establecido. El archivo analizado (dataset) hace alusión al archivo con el cual se genera el codebook, por último el archivo de búsqueda contiene una colección de datos del dataset original.

La diferencia al ejecutar estos algoritmos es que a VQ se le definen la cantidad de prototipos (P) que se desea encontrar, mientras que a KD-Tree-VQ se establecen la alturas (H) que se desean definir en los árboles, además de la cantidad de árboles a utilizar (Sub-árboles), la que por defecto es 1.

Se puede observar que cada uno de los algoritmos, puede tener el mismo tipo de respuestas, pero con cantidad de iteraciones diferentes. Por otro lado, como se explicó en la descripción del algoritmo la salida de los dos algoritmos es diferente. Teniendo en cuenta esto la cantidad de prototipos necesarios para realizar una representación de los datos puede variar, y por último analizar si para cantidades altas de dimensiones, los algoritmos son capaces de dar una respuesta rápida.

El entorno de trabajo que se utilizó para realizar las pruebas fue **RStudio**, el cual es un software profesional de código abierto creado el 2008 [12]. Este software trabaja con **R**, un lenguaje estadístico de código abierto, utilizado para amplias investigaciones por la comunidad científica, ya que posee una gran cantidad de herramientas gráficas, de cómputo, modelamiento, base de datos, entre otras herramientas, las que ayudan a dar estudio a diversas áreas. **RStudio** ademas de trabajar con **R** incorpora la librería **Rcpp**, que provee una interacción directa entre **R** y **C++**, pudiendo intercambiar variables dinamicamente entre estos dos lenguajes. Esta interacción con **C++** se crea ya que este es un lenguaje multiparadigma que tiene una gran rapidez

de cómputo, la cual aprovecha **RStudio**, con el fin de acelerar el proceso estadístico de **R**.

El computador utilizado para realizar las pruebas fue un Samsung NP300V4A, con un sistema operativo Windows 10 de 64 bits, con una capacidad de 8 GB de RAM y un procesador Intel(R) Core(TM) i5-2430M CPU 2.4GHz.

4.3. Análisis y Resultados Experimentales

Las pruebas realizadas se dividen en tres partes, la primera consiste en observar cómo funciona KD-TREE-VQ en comparación con VQ al ser utilizado con archivos de volumen pequeño. En segundo lugar, apreciar el funcionamiento de KD-TREE-VQ con archivos de gran volumen de dato. Por último analizar distintos tipos de archivos dividiendo las dimensiones en diferentes árboles. Los resultados experimentales se desarrollaron en base a los siguientes datasets de prueba:

Archivo	Instancias	Dimensiones
Diabetes	768	8
Abalone	4177	10
Skin_NonSkin4	245057	4
CovType	581012	10
MiniBooNE_PID	130064	50

Cuadro 4.1: Archivos de datos.

4.3.1. Estudio con Volumen Pequeño de Datos

Como podemos observar en la Tabla 4.1 los archivos Abalone y Diabetes son documentos de tamaño pequeño, los cuales no sobrepasan las 100,000 instancias. Las Tablas 4.2 y 4.3 reflejan los resultados experimentales de ambos archivos con los dos métodos KD-TREE-VQ y VQ. Estas Tablas muestran que el algoritmo de VQ tiene un PPE más bajo con un tiempo de respuesta corto en comparación con KD, donde podemos observar en la tabla 4.2 una diferencia porcentual del 0,07 % y con un TC de un segundo aproximadamente para ambos métodos.

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)
VQ	512	0,03	0,03	0,009	0,58	73,72
KD-TREE-VQ	8	0,41	0,19	0,15	0	0,53
	16	0,22	0,15	0,08	1,02	0,99

Cuadro 4.2: Análisis archivo diabetes.
H:Altura árbol – P: Prototipos VQ

Por otro lado, en la Tabla 4.3 podemos ver una diferencia más baja del 0,02 % pero un tiempo de ejecución en KD-TREE-VQ que supera los 20,32 segundos duplicando el tiempo de ejecución de VQ que es de 10,85 segundos.

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)
VQ	512	0,03	0,01	0,01	10,85	77,45
	1024	0,02	0,01	0,007	21,48	154,74
	2048	0,015	0,01	0,005	42,94	309,36
KD-TREE-VQ	10	0,6	0,11	0,19	0,08	0,76
	15	0,18	0,12	0,06	0,44	1,18
	20	0,09	0,06	0,03	21,32	1,66

Cuadro 4.3: Análisis de archivo abalone.

En conclusión, podemos decir que para conjuntos de pequeños volúmenes de datos, VQ sigue siendo la opción más favorable a utilizar a la hora de analizar este tipo de archivos.

4.3.2. Estudio con Volumen Grande de Datos

Para la realización del estudio con grandes archivos de datos se analizaron dos archivos los cuales superan las 100 mil instancias, Skin y CovType los que se puede apreciar en Tabla 4.1.

La Tabla 4.4 muestra una mejora al utilizar KD-TREE-VQ en el archivo Skin, ya que muestra una obtención del 0.12 % en el PPE, con un tiempo de ejecución de 398,57 segundos en comparación con VQ, el cual no disminuye del 0,17 % y alcanza niveles de tiempo que superan las horas en referencias al TC (6056,54 Segundos).

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)
VQ	8192	0,35	0,47	0,17	122,9	9
	16384	0,35	0,47	0,17	236,6	18,1
	32768	0,35	0,47	0,17	487,63	39,84
	65536	0,35	0,26	0,17	6056,54	71,1
KD-TREE-VQ	4	0,49	0,37	0,24	0,004	0,003
	12	0,32	0,32	0,16	0,46	0,02
	20	0,31	0,40	0,15	19,55	0,04
	24	0,24	0,29	0,12	398,57	0,06

Cuadro 4.4: Skin NomSkin4.

Por otro lado al analizar el archivo CovType observado en la Tabla 4.5, la cual muestra que VQ tuvo méjor resultado, entregando un PPE del 0.01 % en un tiempo de ejecución de 173,6 segundos, a diferencia de KD-TREE-VQ que obtuvo un PPE del 0.06 % en un tiempo mas prolongado que VQ (228,19 segundos), no logrando mejorar el tiempo de ejecución y obteniendo un error promedio más alto.

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)
VQ	8192	0,04	0,01	0,01	173,6	15,3
	16384	0,03	0,001	0,01	248,91	24,86
KD-TREE-VQ	10	0,4	0,1	0,13	85,11	0,04
	15	0,27	0,17	0,08	143,41	0,02
	20	0,18	0,11	0,06	228,19	0,02
	24	0,18	0,11	0,06	603,54	0,03

Cuadro 4.5: covtype 10 dimensiones.

4.3.3. Estudio con División de Dimensiones en los Datos

Mediante los resultados que arrojó el estudio para archivo de gran tamaño, se realizó un análisis en el cual las dimensiones de un archivo se dividieron en distintos árboles con la finalidad de disminuir el porcentaje promedio de error y observar si el tiempo de ejecución de este estudio era parecido o menor que el de VQ.

Podemos apreciar en la Tabla 4.6 que se analizó nuevamente el archivo diabetes, con el archivo dividido en distintos sub-árboles 2 y 4 respectivamente, el cual arrojo

resultados favorables, alcanzando una diferencia porcentual de 0,006 % con un tiempo de ejecución más favorable para KD. Por otro lado, KD-TREE-VQ alcanzo un PPE menor que VQ, con una diferencia del 0,002 %, pero con un TC más alto, este aumento se ve compensando por el TD que es considerablemente menor que el de VQ, teniendo una diferencia de 70 y 68 segundos.

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)	Sub-árboles
VQ	512	0,03	0,03	0,009	0,58	73,72	-
KD-TREE-VQ	10	0,16	0,1	0,06	0,06	1,6	2
	15	0,09	0,09	0,03	1,22	2,39	2
	10	0,04	0,05	0,015	0,11	3,44	4
	15	0,02	0,2	0,007	2,42	5,1	4

Cuadro 4.6: Análisis archivo diabetes.
H: Altura sub-árbol

En la Tabla 4.7 se visualiza el estudio del archivo covtype, con sub-árboles de 2 y 5. Este nuevo estudio demostró una mejora sustancialmente del algoritmo KD-TREE-VQ llegando al mismo PPE del 0,001 porciento con un tiempo de decodificación en menos de un segundo en relación a VQ. Además de tener un tiempo de decodificación ampliamente inferior que el de VQ.

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)	Sub-Árboles
VQ	8192	0,04	0,01	0,01	173,6	15,3	-
	16384	0,03	0,001	0,01	248,91	24,86	-
KD-TREE-VQ	6	0,42	0,11	0,13	0,16	0,01	2
	10	0,21	0,11	0,06	0,17	0,02	2
	15	0,09	0,05	0,03	1,12	0,03	2
	6	0,09	0,06	0,03	0,12	0,04	5
	10	0,03	0,02	0,009	0,14	0,06	5

Cuadro 4.7: covtype 10 dimensiones.

Como último estudio, se puede observar en la Tabla 4.8 que se analizó el archivo mini el cual contiene una cantidad de dimensiones alta, en comparación a los archivos anteriormente observados. Los sub-árboles utilizados para dividir este archivo fueron 5 y 10 respectivamente. Entregando resultados los cuales tienen una diferencia porcentual máxima del 0,035 %. VQ logra un PPE del 0,02 % en 3,55 segundos, mientras

que KD-TREE-VQ logra este PPE en un TC de 4,68 segundos, no logrando mejorar el tiempo de ejecución de VQ, pero esto se compensa con la baja complejidad de búsqueda que genera KD, ya que a diferencia de VQ que su TD es de 9,35 segundos, el TD de KD-TREE-VQ fluctúa entre los 0,07 y los 0,2 segundos.

Algoritmo	H/P	RMSE	S	PPE	TC (s)	TD (s)	Sub-árboles
VQ	4096	0,15	0,09	0,02	3,55	9,35	-
	8192	0,11	0,07	0,015	726,16	35,4	-
KD-TREE-VQ	12	0,37	0,13	0,05	0,26	0,07	5
	17	0,33	0,13	0,05	1,15	0,08	5
	6	0,37	0,14	0,05	0,09	0,09	10
	10	0,25	0,11	0,03	0,2	0,12	10
	15	0,17	0,09	0,02	4,68	0,17	10

Cuadro 4.8: MiniBooNE 50 dimensiones.

5. Conclusión y Trabajo Futuro

En esta sección se explican las conclusiones experimentales obtenidas del desarrollo del algoritmo KD-TREE-VQ y del análisis de los resultados experimentales.

Finalmente se explican cuáles son los puntos que se deben mejorar y estudiar en un futuro.

5.1. Conclusión

VQ es un algoritmo de compresión de datos, el cual tiene un tiempo de ejecución muy elevado al momento de procesar archivos de gran volumen de información. A lo largo de la literatura se puede encontrar un alto grado de estudios referente a mejorar la complejidad temporal de VQ. En este documento en particular, se desarrolló y estudió la implementación del algoritmo KD-TREE-VQ, el cual utiliza la migración de los valores, de forma dinámica, a medida que se construye el árbol k-dimensional. Este algoritmo estrega como resultado, un codebook en estructura de árbol con valores unidimensionales y que en cada nivel de este representa una dimensión del archivo en análisis.

En la sección de análisis y resultados experimentales Sección 4.3, podemos ver que KD-TREE-VQ para el estudio de archivos pequeños, no logra un resultado experimental aceptable y sigue siendo más favorable utilizar el algoritmo de VQ. Esto se demuestra porque KD-TREE-VQ para lograr un PPE cercano a VQ o más bajo emplea un tiempo de ejecución mayor.

Por otro lado, en el análisis de archivos de gran volumen de información, podemos observar que para archivos donde VQ no logra obtener una respuesta en un corto lapso de tiempo, KD-TREE-VQ obtiene una respuesta mejor en un pequeño intervalo

de tiempo. Esto deja las puertas abiertas, a mejorar el algoritmo KD-TREE-VQ, ya que para algunos tipos de archivo VQ sigue teniendo mejor TC.

Al no obtener resultados experimentales 100 % favorables, se modificó la ejecución del algoritmo KD-TREE- VQ, dividiendo los datos en grupos más pequeños, con el fin de evaluar distintas dimensiones en distintos arboles y obtener respuestas rápidas y certeras. A esta ejecución se le denominó bosque de búsqueda descrita en la Sección 3.3. Este método logró mejorar considerablemente el tiempo de ejecución, al mismo tiempo de obtener valores bajos y mejores de PPE que VQ, con archivos de pequeño y gran volumen de información

Como síntesis, en la práctica KD-TREE-VQ al tener sus datos estructurados en un BST multidimensional y sub dividir el problema en sub-árboles, logra mejorar la complejidad temporal de VQ, al momento de codificar y decodificar archivos de pequeño y gran volumen de información, logrando en algunos casos tener mejores respuestas que VQ y en otros casos se acepta el sacrificio de tener un BMU aproximado, sabiendo que la disminución en el tiempo de ejecución es considerablemente mejor.

5.2. Trabajo Futuro

El algoritmo KD-TREE-VQ utilizando sólo la construcción de un árbol, al no lograr mejorar en todos los casos la complejidad temporal de VQ, se debe estudiar más, con el fin de mejorar la aproximación de los vectores prototipos, reduciendo el error cuadrático medio (RMSE), en un tiempo bajo, para archivos de distintos tamaños.

Por otro lado, el estudio de la construcción de bosque, utilizando el algoritmo de KD-TREE-VQ, mejoró considerablemente la respuesta del BMU, pero se debe estudiar si existe una forma más eficiente de dividir las dimensiones dentro de los arboles, con el objetivo de mejorar el PPE del algoritmo y disminuir más el tiempo de ejecución del algoritmo.

Existen muchos desarrollos y estudios de algoritmos referentes a mejorar la complejidad temporal de VQ. En este documento se construyó y analizó el algoritmo de KD-TREE-VQ en comparación con VQ, pero hace falta determinar en una próxima investigación si este nuevo algoritmo es capaz de mejorar la complejidad temporal frente a otros estudios y algoritmos como los descritos en la Sección 2.5, con la finalidad de saber en qué áreas donde VQ se ha implementado, KD-TREE-VQ será un aporte.

Bibliografía

[1] César A. Astudillo and B. John Oommen. Approximating the best matching unit in self-organizing maps utilizin random hyperplane search trees. *IEEE*, 13:1–12, 2014.

[2] Yeou-Jiunn Chen, Shih-Chung Chen, and Jiunn-Liang Wu. A hybrid vector quantization combining a tree structure and a voronoi diagram. *MATHEMATICAL PROBLEMS IN ENGINEERING*, 2014:1–9, 2014.

[3] Chao-Ping Chu, Cheng-Yu Yeh, and Shaw-Hwa Hwang. An efficient tree-structured vector quantization using dynamic triangular inequality elimination. *IEEJ TRANSACTIONS ON ELECTRICAL AND ELECTRONIC ENGINEERING*, 9:S70–S72, 2014.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, Heidelberg, 2008.

[5] Comba J. and Naylor B. *Conversion of Binary Space Partitioning Trees to Boundary Representation*. Spatial Labs Inc, 1997.

[6] Robert G. Palmer Jr., Howard Jay Siegel, Janet M. Siegel, and John K. Antonio. *Implementation of a Tree-Structured Vector Quantizer for Image Compression on the MasPar MP-1 Parallel Machine*. IEEE Xplore, 2002.

[7] Yoseph Linde, Andres Buzo, and Robert M. Gray. An algorithm for vector quantizer design. *IEEE*, 28:84 – 95, 1980.

[8] Yung-Chih Liu, Gwo-Her Lee, Jinshiuh Taur, and Chin-Wang Tao. Index compression for vector quantisation using modified coding tree assignment scheme. *IET IMAGE PROCESSING*, 8:173–182, 2014.

[9] Bruce F. Naylor. *A Tutorial on Binary Space Partitioning Trees.* Spatial Labs Inc, 2005.

[10] Jana Nowakova, Michal Prilepok, and Vaclav Snasel. Medical image retrieval using vector quantization and fuzzy s-tree. *JOURNAL OF MEDICAL SYSTEMS*, 41:18, 2017.

[11] University of California. UCI, Machine Learning Repository. http://archive.ics.uci.edu/ml/index.php.

[12] RStudio Team. *RStudio: Entorno de desarrollo integrado para R.* RStudio, Inc., Boston, MA, 2015.

[13] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers, 2006.

[14] Wiroonsak Santipach and Kritsada Mamat. Tree-structured random vector quantization for limited-feedback wireless channels. *IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS*, 10:3012–3019, 2012.

[15] Robert Sedgewick and Kevin Wayne. Algorithms, 4th edition. http://algs4.cs.princeton.edu/home/.

[16] J. S. Taur, Y. C. Liu, G. H. Lee, and C. W. Tao. Vector quantisation index compression based on a coding tree assignment scheme with improved search-order coding algorithms. *IET IMAGE PROCESSING*, 6:318–326, 2012.

[17] Technische Universiteit Delft. TUDelft. http://homepage.tudelft.nl/n9d04/occ/.

Glosario

VQ Vector Quantization

KD-TREE k-Dimensional Tree

KD-TREE-VQ Vector Quantization Using k-Dimensional Tree

TSVQ Tree Structured Vector Quantization

SIMD Single Intruccion, Multiple Data

DTIE Dynamic Triangular Inequality Elimination

DV Voronoi Diagram

LBG Algorithm Linde-Buzo-Grey

PCA Principal Component Analysis

FSVQ Full-Search Vector Quantization

NICA Neighbouring Index Code Assignment

CTAS-ISOC Coding Tree Assigment Scheme With Improved Search-Order Coding

LSOC Left-Pair Search-Order Coding

USOC Upper-Pair Search-Order Coding

SOC Search-Order Coding

MCTAS Modified Coding Tree Assignment Scheme

DITC Dynamic Index Table Coding

RVQ Random Vector Quantization

MIMO Milti-Input Multi-Output

CMDA Code Division Multiple Access

RMSE Root-Mean-Square Error

ANEXOS

A. Primeros Pasos

Al inicio del estudio para realizar la combinación entre VQ y KD-TREE se creó la primera versión del algoritmo KD-TREE-VQ, el cual implementa lo más cercano a la realidad la construcción de un árbol k-dimensional y las migraciones de algoritmo de cuantización vectorial. Este se enfoca en crear los valores de los nodos recorriendo el árbol de forma ascendente y migrando los nodos a medida que se desciende en la rama. El algoritmo 7 describe la construcción del árbol y la migración de los datos.

Todos los nodos están inicialmente desactivados, con el fin de activarlos en el momento de asignarles un valor y saber si seguir recorriendo el árbol por esa rama, ya que al no estar con un dato asociado, ningún nodo el cual desciende de él, debe estar activado (Líneas 1-2). Siguiendo con el algoritmo se comienza a realizar las iteraciones definidas para hacer las migraciones. Cuando comience un ciclo se define i (posición del nodo que se está analizando) como la raíz (posición 0). Luego se selecciona un dato al azar del conjunto de datos que se quiere analizar o comprimir (Líneas 5-6). Después se inicia el descenso en el árbol desde la raíz hasta un nodo hoja (Línea 7). Lo primero que se realiza es seleccionar la dimensión que se analiza con respecto a la profundidad del árbol que se está estudiando. Luego se verifica si el nodo que se está analizando tiene un valor asociado, de no tenerlo se le asigna un valor aleatorio de la dimensión correspondiente, se activa y se finaliza el descenso por la rama en esta posición y se inicia una nueva iteración (Líneas 8-12). De no ser este el caso se continua con la ejecución normal del algoritmo, en la cual se selecciona el nodo hijo con el que se continua y se migra el nodo en análisis (Líneas 13 -17), y se continua el descenso en el árbol. Finalmente una vez terminado el descenso se actualiza el factor de aprendizaje y se continua con la siguiente iteración. Al finalizar todo el proceso, se obtiene un árbol completo, con los nodos migrados en referencia al archivo analizado.

Algorithm 7 KdT-VQ ($\mathcal{X}$, h, T, α_i, α_f)

Input:

$\mathcal{X}$, Colección de datos $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}, \mathbf{x_i} \in \mathbb{R}^d$.

h, Altura del árbol.

T, Número máximo de iteraciones.

α_i, α_f, Factor de aprendizaje inicial y final, respectivamente.

Ouput:

A, arreglo $\{a_1, a_2, \ldots, a_{2^h-1}\}$, $a_i \in \mathbb{R}$, que contiene puntos de corte del árbol KD binario completo.

Method:

1: **for** b ←1 to $|A|$ **do** ▷ Para cada nodo del árbol
2: B[b] ← false ▷ Nodo inicialmente esta desactivado.
3: ▷ La cantidad de elementos del arreglo A, es la cantidad de nodos que posee un árbol binario completo de altura H.
4: **for** t ← 1 to T **do**
5: **i** ←0, ▷ Posición en el árbol que actualmente esta siendo analizado.
6: **x** ← random-select($\mathcal{X}$) ▷ Seleccionar un dato de $\mathcal{X}$ al azar.
7: **for** j ←0 to $h-1$ **do** ▷ Recorrer desde las raíz hasta las hojas
8: **k** ← j %d ▷ Dimensión a procesar con respecto a la profundidad actual del árbol.
9: **if** B[i] = false **then**
10: A[i] ← $\mathbf{x}_k$ ▷ Asignamos la componente k del vector al nodo actual
11: B[i] ← true ▷ Activa nodo actual del árbol
12: **break**
13: **if** A[i] ≤ $\mathbf{x}_k$ **then** ▷ Identificar la rama del árbol a seguir, $\mathbf{x}_k \in \mathbb{R}$.
14: i ← 2*i + 1 ▷ asignamos la posición del hijo izquierdo de la posición actual.
15: **else**
16: i ← 2*i+ 2 ▷ asignamos la posición del hijo derecho de la posición actual.
17: update-rule(A[(i-1)/2],$\mathbf{x}_k$,α_t) ▷ Migración el nodo.
18: α_t ← decay(α_i,α_f,t, T) ▷ Actualizar el factor de aprendizaje.

Para corroborar que el algoritmo entrega un resultado correcto, se implementó un algoritmo de visualización utilizado para graficar un árbol KD. Como se explicó en la Sección 2.3, utiliza los valores de los nodos para dibujar una línea en la dimensión a la que pertenece un nodo dentro de un plano euclidiano, respetando los límites superiores e inferiores que tiene un BST. EL algoritmo se describe a continuación:

En el Algoritmo 8 podemos observar la implementacion del algoritmo de visualización de KD-Tree para dos dimensiones, el cual es un algoritmo recursivo que toma como primera entrada, la posición de la raíz del árbol dentro del arreglo ($i = 0$), la profundidad del árbol que se está analizando ($depth = 0$), para saber en que dimensión dibujar la línea y los límites superiores $[X_{Max}, Y_{Max}]$ e inferiores $[X_{Min}, Y_{Min}]$ de un hiperplano en el cual se dibujarán las líneas. En este estudio, al tener los datos normalizados, los límites inferiores y superiores se encuentran en el rango entre 0 y 1, como se puede apreciar en la Figura A.1 donde se forma un rectángulo con limites [0,0] y [1,1].

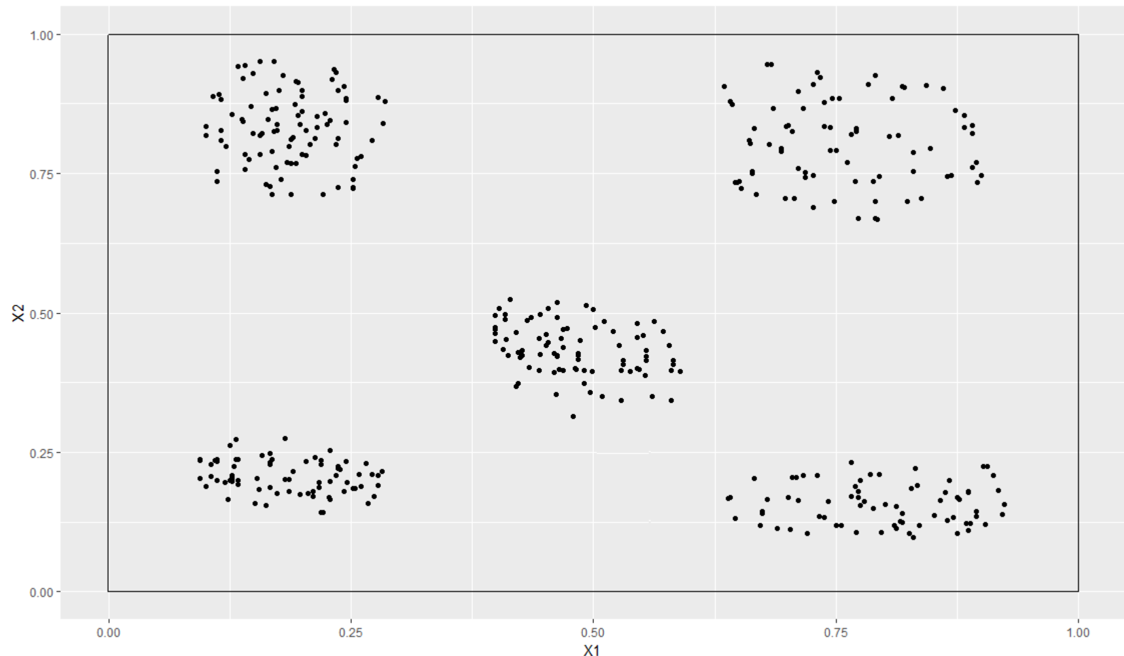


Figura A.1: Hiperplano donde se realizan las divisiones.

Luego se prosiguen a dibujar las líneas analizando el valor de los nodos de la siguiente forma: primero verificamos en que dimensión se encuentra nuestro nodo a analizar mediante la profundidad donde se encuentra (Línea 1), si este está en la

dimensión X se verifica si este tiene hijos, de ser así se inicia el proceso recursivo (Líneas 3-6), modificando el límite superior (X_{Max}) para el hijo izquierdo y el límite inferior (X_{Min}) con el valor del nodo actualmente analizado, finalmente se dibuja la línea modificando los límites de la dimensión analizada (X) con el valor actual del nodo (Línea 7). Este proceso se repite para el análisis de la dimensión Y (Líneas 8-13).

Algorithm 8 KDTree.visualization(i, X_{min}, Y_{min}, Y_{max}, Y_{max}, profundidad)

Input:

i índice del nodo en el arreglo A que representa al árbol binario.

X_{min} Posición X mínima en el lienzo de dibujo.

Y_{min} Posición Y mínima en el lienzo de dibujo.

X_{max} Posición X máxima en el lienzo de dibujo.

Y_{max} Posición Y máxima en el lienzo de dibujo.

$profundidad$ Profundidad del árbol.

Global:

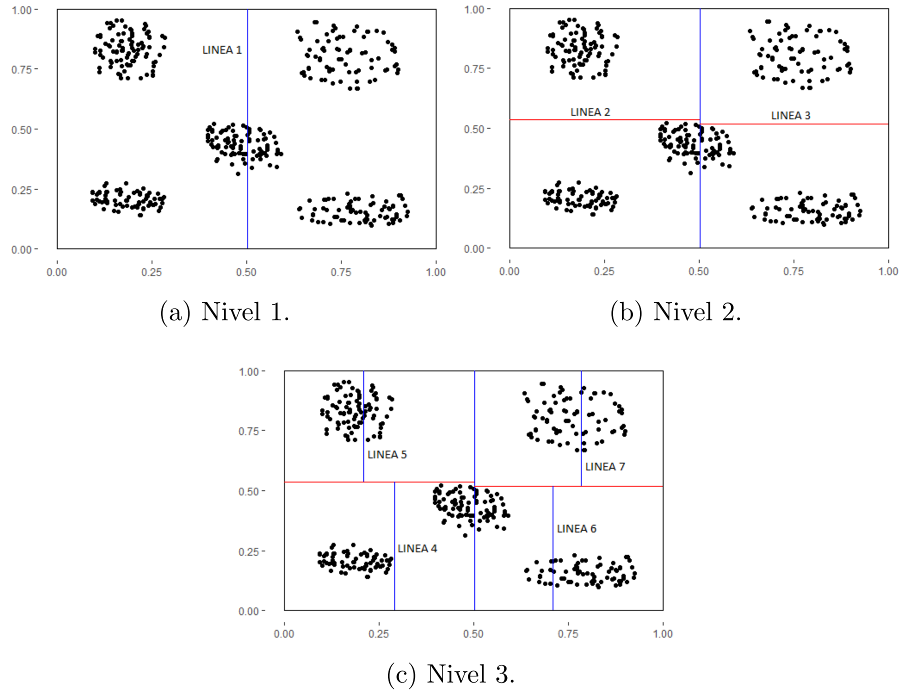
A arreglo $\{a_1, a_2, \ldots, a_{2^h-1}\}$, $a_i \in \mathbb{R}$, que contiene puntos de corte del árbol KD.

Method:

1: dimension ← profundidad %2 ▷ Se selecciona la dimensión que se analizara para realizar el dibujo de la linea dependiendo de la profundidad donde se encuentra el nodo.
2: **if** dimension=0 **then** ▷ Dimensión X
3:     **if** $i * 2 + 1 < |A|$ **then** ▷ modificamos el valor X_{Max} del hijo izquierdo con el valor del padre
4:         KDTree.visualization($i * 2 + 1$, X_{min}, Y_{min}, $A[i]$, Y_{max}, depth $+ 1$) ▷ modificamos Xmax
5:     **if** $i * 2 + 2 < |A|$ **then** ▷ modificamos el valor X del hijo derecho con el valor del padre
6:         KDTree.visualization($i * 2 + 2$, $A[i]$, Y_{min}, X_{max}, Y_{max}, depth $+ 1$) ▷ modificamos Xmin
7:     Dibujar($A[i]$, Y_{min}, $A[i]$, Y_{max}) ▷ Dibujamos la línea en el gráfico
8: **else** ▷ Dimensión Y
9:     **if** $i * 2 + 1 < |A|$ **then** ▷ modificamos el valor y del hijo izquierdo con el valor del padre
10:         KDTree.visualization($i * 2 + 1$, X_{min}, Y_{min}, X_{max}, $A[i]$, depth $+ 1$) ▷ modificamos Ymax
11:     **if** $i * 2 + 2 < |A|$ **then** ▷ modificamos el valor Y del hijo derecho con el valor del padre
12:         KDTree.visualization($i * 2 + 2$, X_{min}, $A[i]$, X_{max}, Y_{max}, depth $+ 1$) ▷ modificamos Ymin
13:     Dibujar(X_{min}, $A[i]$, X_{max}, $A[i]$) ▷ Dibujamos la línea en el gráfico

En la Figura A.2 podemos ver un ejemplo de un árbol de altura 3 de dos dimensiones, el cual podemos apreciar por niveles como se va graficando, un árbol

k-dimensional, utilizando el algoritmo de visualización antes descrito.



(a) Nivel 1. (b) Nivel 2.

(c) Nivel 3.

Figura A.2: Visualización de un árbol de altura 3, por niveles.

En la Figura A.3 podemos observar cómo se modifican los parámetros para realizar los dibujos de las líneas que conforman el gráfico antes descrito, y cómo se distribuyen los valores dentro del árbol que se está dibujando.

Índice	Dimension	Padre	Valor	Xmin	Ymin	Xmax	Ymax	Linea (x1,y1) - (x2,y2)
Valores Iniciales				0	0	1	1	Rectángulo [0,0][1,1]
0	X	-	0.5	0.5	0	0.5	1	Linea 1 (0.5; 0) - (0.5, 1)
1	Y	0	0.54	0	0.54	0.5	0.54	Linea 2 (0, 0.54) - (0.5, 0.54)
2	Y	0	0.52	0.5	0.52	1	0.52	Linea 3 (0.5, 0.52) - (1, 0.52)
3	X	1	0.3	0.3	0	0.3	0.54	Linea 4 (0.3, 0) - (0.3, 0.54)
4	X	1	0.21	0.21	0.54	0.21	1	Linea 5 (0.21, 0.54) - (0.21, 1)
5	X	2	0.7	0.7	0	0.7	0.52	Linea 6 (0.7, 0) - (0.7, 0.52)
6	X	2	0.78	0.78	0.52	0.78	1	Linea 7 (0.78, 0.52) - (0.78, 1)

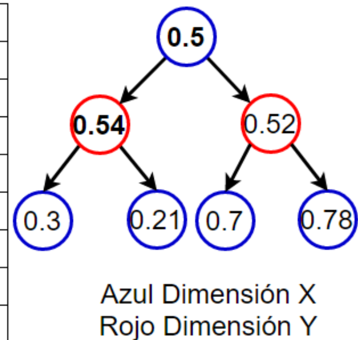


Figura A.3: Seguimiento de algoritmo de dibujo y árbol asociado.

Mediante ese algoritmo de visualización se lógro observar el problema que trae la migración de los nodos de forma dinámica en la estructura de un BST.

Al obtener el gráfico de un árbol con 5 niveles se pudo observar que el dibujo de algunas líneas no se encontraban trazadas correctamente, quedando fuera de rango con los nodos antecesores, interfiriendo en la estructura de un árbol binario, exhibiendo que el algoritmo inicial sufre algún error.

En primer lugar se analizó si el algoritmo de visualización tenía algún defecto, analizándolo con árboles creados correctamente, los cuales al ser dibujados por el algoritmo antes descrito se graficaron correctamente. Por conclusión el algoritmo KD-TRRE-VQ debía tener algún error.

Como podemos observar en la Figura A.4, el problema se centra en que el nodo hijo adquiere un valor mayor o menor a los nodos antecesores, dejando a este fuera de rango, en este caso se muestra un nodo hijo mayor que el nodo antecesor, en el momento que debe ser menor.

Figura A.4: Nodo fuera de rango

Para expresarlo de mejor manera en la Figura A.5 podemos apreciar que el nodo 23, que representa a la linea rojo de la Figura A.4 está fuera de rango, ya que sus nodos derechos antecesores, por regla estricta de un árbol BST debe ser mayor, y en este caso el nodo 5 tiene un valor menor que el nodo 23, por lo cual deja en claro que existe un problema en el algoritmo **KD-TREE-VQ**.

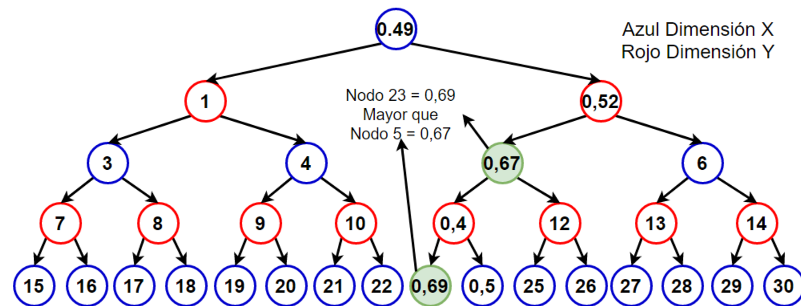


Figura A.5: Árbol con error en nodo 23

Este problema se soluciona mediante un arduo seguimiento del error en el algoritmo, y definiendo que cada nodo debe tener un límite inferior y superior en el momento de migrar un nodo de forma dinámica dentro del árbol. Agregando al algoritmo **KD-TREE-VQ 2.0** las variables auxiliares MM y $MMaux$.