UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

FACULTAD DE CIENCIAS FÍSICO-MATEMÁTICAS

DISEÑO Y CREACIÓN DE MEDIDAS DE DISIMILARIDAD ENTRE PERMUTACIONES PARA BÚSQUEDAS POR PROXIMIDAD

TESIS

QUE PARA OBTENER EL GRADO DE
LICENCIADO EN CIENCIAS FÍSICO MATEMÁTICAS
PRESENTA

LUIS DAVID TRUJILLO SILVA

ASESORES

DRA. KARINA MARIELA FIGUEROA MORA

DR. JOSÉ ANTONIO CAMARENA IBARROLA,

Morelia,

Michoacán, agosto de 2018

Este trabajo fue financiado por la UMSNH

AGRADECIMIENTOS

Primeramente quiero dedicar esta tesis a mi familia, en particular a mi madre que me ha apoyado en todos y cada una de las decisiones que he tomado. A mis hermanos Daniel, Nancy e Ivan por toda la confianza que me otorgaron, también agradecer a la Dra. Karina Figueroa por la paciencia, comprensión y apoyo que me brindo al realizar este trabajo. A mi tío Guillermo por haberme recibido como un hijo, Arturo y Mónica por el apoyo que me brindaron y no pueden faltar mis primos Alejandro (oso), Israel (viejon), Lady y Estefania por todas esas aventuras que hemos tenido y las que nos faltan, Vanessa (euro) y su Familia. Que me han recibido con gran afecto y cariño.

Mencionando también con quienes compartimos los miércoles santos con tantas aventuras Alyabra, Juan, Fernando, Henry, Aron, Raul, Felipe, Isai, Paco, mis compañeros Rodrigo, Nadia, liz, Ciclaly, Sajida, Chelis, Gustavo, Javi, con quienes hemos tenido experiencias que no olvido. Chucho que me salvo en muchas ocasiones me ayudo en las materias, a mis compañeros de trabajo Anahi, Antonio, Marimar, Alejandra, Alma, Karen, a los 5 Juanes que han hecho un ambiente de trabajo muy bueno a todos aquellos con quienes hemos compartido algo (lo que sea) se los agradezco, espero seguir conviviendo con TODOS, pasar muchos y mejores momentos.

Resumen

Las búsquedas por proximidad o similaridad consisten en recuperar los elementos de la base de datos mas parecidos a uno de consulta. Este tipo de búsquedas es escencial en nuestros días pues actualmente se tienen grandes volúmenes de datos digitales donde la búsqueda exacta no tiene mucha relevancia, en cambio la búsqueda por proximidad es la única alternativa. Por ejemplo, buscar la fotografía de una persona en un repositorio de imágenes, el rostro de la persona cambia con sus facciones.

Una estrategia para resolver las búsquedas es comparando todos los elementos de una base datos, esto se conoce como búsqueda secuencial. Se asume que la comparación sea una función de distancia costosa y determine que tanto se parece un objeto a otro. Otra alternativa es construir un índice para hacer mas eficiente esta búsqueda y evitar la búsqueda secuencial.

En este trabajo, se propone modelar el problema como un espacio métrico (i.e. una base de datos y una función de distancia) y modificar un algoritmo existente para mejorar el procesamiento de la respuesta. Nuestra propuesta consiste en modificar la función de distancia empleada en los algoritmos basados en permutaciones (PBA) [12] para comparar las permutaciones.

Los PBA funcionan de las siguiente manera, seleccionan un conjunto de la base de datos (es decir, permutantes) y el resto de los elementos determina el orden de cercanía de los permutantes, a esto se le llama permutación. Las consultas siguen el mismo procedimiento y los elementos que forman parte de la respuesta serán aquellos cuya permutación es muy parecida a la de la consulta. Los resultados que presentamos en este trabajo muestran que es posible mejorar la técnica de PBA y contribuir en el estado del arte de este problema.

Palabras clave: Búsqueda por similitud, vecinos más cercanos, recuperación de información, reconocimiento de patrones,algoritmos.

Abstract

Similarity searching or proximity searching consists in retrieving those objects that are similar of a database from a given query. Nowdays, this kind of searching is essential because there are huge digital databases where exact matching is almost impossible. Instead, the unique alternative can be the proximity searching.

An strategy to solve this kind of queries is comparing all the elements in a database (sequential scan). The comparison is using a distance function which allows to measure how similar objects are. Another alternative is building an index in order to solve the query without making a sequential scan.

In this work, problem is modeled as metric space (i.e. a database and a distance function) and improving an existing algorithm in order to get better processing time for answering. Our proposal consists in modify the distance function used in permutation-based algorithm (PBA) [12] in order to compare permutations.

PBA works as follow, first, a set of elements are chosen (i.e. set of permutants), the rest of the database sort these permutants in increasing order, it is called permutation. Queries follows the same process and elements which permutation is similar to the query's permutation they are part of the answer. In this work, we show that our proposal improves the PBA and contributes to the state of art.

Índice general

Índice de figuras

Capítulo 1

Introducción

La búsqueda es uno de los problemas fundamentales en las ciencias de la computación, además de ser el núcleo de casi todas las aplicación de computación, a medida que la aplicación es mas sofisticada mas díficil es la búsqueda. Un caso típico son las bases de datos, donde básicamente, se tiene un conjunto de datos y los usuarios proporcionan una consulta que la aplicación deberá constesta si la consulta tiene una respuesta o no en esa base de datos.

El caso típico de bases de datos (BD) son las tradicionales, o que cumplen con el modelo relacional (el modelo más utilizado actualmente para implementar las BD ya planificadas), requieren que los datos tengan una división clara por características; por ejemplo, en una BD de nombres de personas la separación obvia es: *nombre, apellido materno, apellido paterno, año de nacimiento*, etc. El tipo de búsqueda que se emplean se considera exacta pues debe coincidir letra por letra, además de que es claro en qué característica se buscará: por ejemplo, quiero saber quiénes son las personas llamadas "David", bastaría con buscar en la característica llamada *nombre* y aquellos nombres que tengan "David" serán parte de la respuesta. Sin embargo, hoy en día, las bases de datos multimedia no tienen esa división claramente definida, piense, por ejemplo, en cómo dividiría una imagen o una grabación de audio, por mencionar algunos.

Las bases de datos multimedia (BDM) se encuentran en casi todos los aspectos de la vida cotidiana y realizar búsquedas en ellas es esencial para muchas ciencias (i.e. biología, computación, etc). Sin embargo, en este tipo de bases de datos no tiene mucho sentido la búsqueda exacta (como se explicó en el párrafo anterior), en cambio, el único recurso para estas búsquedas es la consulta por similaridad o proximidad, por ejemplo, buscar una canción en una colección musical mediante el sonido emitido mientras se tarareaba la canción. Note que no

tiene sentido hacer una búsqueda exacta.

Una estrategia para resolver cualquier búsqueda en una base de datos es haciendo una comparación secuencial de los datos contra la consulta, aunque esto sería prácticamente imposible en bases de datos gigantescas. Cuando hablamos de la *comparación (o distancia)* debemos dejar claro que ésta corresponde al dominio de los datos y que se espera que indique que tanto se parecen dos objetos de ese dominio. Otra estrategia es generar una estructura de datos que nos permita establecer un *orden* en los datos, de manera que, al momento de hacer la búsqueda sea posible tener una respuesta sin necesidad de comparar toda la base de datos. Este tipo de estructura de datos se le conoce como *índice*.

En general, es una vez que se tiene una función de distancia y los datos, es posible modelar el problema como un espacio métrico (ver el capítulo 2). La ventaja de modelarlo de esta forma es que se tiene un modelo genérico para bases de datos donde sólo se requiere conocer la función de distancia entre objetos y ninguna otra información adicional [17]. La función de distancia en algunas BDM puede ser computacionalmente costosa de calcular de aquí que nuestra meta sea reducir el número de cálculos de distancia a evaluar para obtener la respuesta.

Nuestra propuesta consiste en mejorar uno de los algoritmos propuestos en la literatura [11, 12], el cual, permite resolver la búsqueda en BD gigantescas, realizar un menor número de comparaciones para tener la respuesta, utilizar de manera eficiente los recursos físicos de la computadora, compactar el índice y por supuesto, mejorar el estado del arte en este tipo de búsquedas.

1.1. Aplicaciones de búsquedas por similitud

Algunas aplicaciones de este tipo de búsquedas por similitud son en las siguientes áreas: recuperación de información, estadísticas, geometría computacional, reconocimiento de patrones, minería de datos, entre otros. A continuación se describirán algunos de estos.

- Reconocimiento de patrones. En estas aplicaciones se tienen 3 etapas: la segmentación, la caracterización y clasificación. La primera de ellas debe tomar los datos de entrada y eliminar el ruido de ellos (por ejemplo, una fotografía tiene un objeto de interés y el resto es ruido). La segunda etapa es la caracterización, la cual toma los elementos mas importantes del objeto de interés y que pueden ser utilizados para diferenciar a los objetos. Por último, la clasificación, es esta etapa donde se requiere resolver el problema de la

búsqueda por similitud, pues de ésto depende el tipo de clasificación y la velocidad en la que se puede responder.

- Recuperación de información. El objetivo en estas aplicaciones es recuperar registros semejantes a uno de consulta. Un ejemplo típico es un buscador en internet, el usuario proporciona un conjunto de palabras (consulta) y obtiene como respuesta documentos ordenados por relevancia (o similitud) con respecto a la consulta.

Algunos otros problemas específicos son: agrupación de documentos de texto por temas, búsquedas de cadenas similares de acuerdo a su ADN, búsqueda en imágenes similares, reconocimiento facial por características similares, etc. donde estos problemas son especificos y aunque no los veremos a fondo esto no quiere decir que no son importantes solo que para fines de este trabajo solo los mencionare

1.2. Organización

En este trabajo se propone una nueva forma de establecer un orden parcial en un índice conocido como: algoritmos basados en permutaciones.

Este documento está organizado de la siguiente manera.

PARTE 1. CONCEPTOS BÁSICOS

Capítulo 2: Conceptos Básicos. En este capítulo de explicarán las definiciones formales, conceptos básicos, los problemas típicos en este tipo de problemas y los desafíos en la búsqueda por proximidad en espacios métricos.

Capítulo 3: Estado de Arte. En este capítulo se presenta una revisión del estado del arte de los algoritmos para búsquedas por proximidad en espacios métricos. Cabe mencionar que dicho estado del arte se encuentra dividido en tres grandes familias: algoritmos basados en pivotes, algoritmos basados en particiones compactas y algoritmos basados en permutaciones.

PARTE 2. PROPUESTA

Capítulo 4: Propuesta. Los detalles de la propuesta se darán en este capítulo. La idea general es mejorar el orden parcial que establecían los algoritmos basados en permutaciones.

Capítulo 5: Experimentación. El desempeño de la propuesta se muestra de manera experimental con distintas bases de datos, entre ellas: un conjunto

de bases de datos de vectores generados sintéticamente y distribuídos de manera uniforme en el cubo unitario; otras bases de datos utilizadas fueron con datos reales (imágenes y diccionarios de palabras). En este capítulo se muestra que se consigue tener hasta un 30 % de mejora respecto al algoritmo original.

Capítulo 6: Conclusiones y Trabajo a Futuro. La tesis termina mostrando conclusiones y el trabajo futuro como resultado de la propuesta.

Parte I

CONCEPTOS BÁSICOS

Capítulo 2

Conceptos Básicos

En este capítulo se explican algunos conceptos utilizados a lo largo de la tesis. El primer concepto que se necesita es el de espacio métrico. Éste consta de un espacio (conjunto de datos) y una medida de semejanza entre los elementos de ese espacio. No existe una definición genérica de esta medida pues está estrechamente ligada a la aplicación y a las características que se quieran evaluar en los elementos.

Formalmente, sea (X,d) un espacio métrico, donde X es el universo de objetos válidos, y d es la función de distancia, $d : X \times X \rightarrow R^+$ que denotará la medida de semejanza entre los elementos de X. Cuanto más pequeño es el valor de d, más parecidos entre sí son los elementos. La función de distancia debe satisfacer las siguientes propiedades:

♣ Positividad estricta:$d(x,y) > 0$ si $x \neq y$.

♣ Simetría:$d(x,y) = d(y,x)$

♣ Reflexividad:$d(x,y) = 0$

Hasta ahora, las propiedades mencionadas sólo aseguran una definición consistente de la función de distancia y no pueden ser usadas para evitar comparaciones en una búsqueda por proximidad. La siguiente propiedad es la que permite descartar elementos sin ser comparados directamente contra la consulta.

♣ Desigualdad triangular:$d(x,z) \leq d(x,y) + d(y,z)$

En esta tesis usaremos un conjunto finito de objetos lo llamaremos base de datos $\mathbb{U} \subseteq \mathbb{X}$ de tamaño $n = \mathbb{U}$. El conjunto de elementos lo llamaremos

diccionario, base de datos o simplemente conjunto de elementos. El término distancia será usado para referirnos a la métrica entre los elementos de la base de datos.

2.1. Consultas por Proximidad

Básicamente hay dos:

las consultas por rango: que consiste en recuperar todos los elementos en un radio r, esto es: $R_d(q,r) = \{u \in \mathbb{U} | d(q,u) \leq r\}$.

las consultas de los K vecinos mas cercanos: Este tipo de búsqueda consiste en recuperar los vecinos mas cercanos en la base de datos a la consulta. Esto es, $A \subseteq \mathbb{U}$ tal que, $A = KNN(q) = \{u \in A \mid \forall u \in \{\mathbb{U} - A\}, d(q,u) \leq d(q,v),\ K = |A|\}$

El caso mas común en muchas aplicaciones es cuando se quiere recuperar el vecino mas cercano es decir $K = 1$ problema es conocido por sus siglas en inglés $1NN$(Nearest Neighbor).

Las consultas para encontrar los vecinos más cercanos pueden ser solucionadas con los algoritmos que resuelven consultas de rango. Esto se logra mediante distintas técnicas: De radio incremental, backtracking con reducción del radio, priorizando el backtracking.

Gráficamente, estas consultas se muestran en la figura: 2.1 (un espacio métrico en el plano, considerando la distancia Euclidiana), en el lado izquierdo la consulta por rango $R_d(q,r)$ y en el derecho la consulta de los dos vecinos más cercanos ($2NN$).

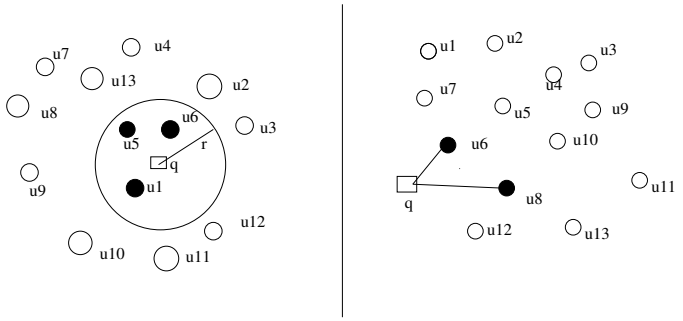


Figura 2.1: Tipos de consultas por proximidad en espacios métricos. Izquierdo: búsqueda por rango. Derecho: búsqueda de los 2 vecinos mas cercanos.

2.1.1. Tipos de Distancias

Un caso particular de los espacios métricos son los llamados espacios vectoriales con m-dimensiones. Un elemento de un espacio vectorial se vería como $(x_1, x_2, \ldots, x_m)$. Por ejemplo, sean $\vec{u}$, $\vec{q}$ vectores.

La distancia puede ser la métrica L_P (La métrica de Minkowski de orden p) definida como:

$$L_p = d_p(\vec{u}, \vec{q}) = \sqrt[p]{\sum_{i=1}^{m} |u_i - q_i|^p} \tag{2.1}$$

donde algunos casos especiales son:

Manhattan

$$L_1 = d_1(\vec{u}, \vec{q}) = \sum_{i=1}^{m} |u_i - q_i| \tag{2.2}$$

Euclidiana

$$L_2 = d_2(\vec{u}, \vec{q}) = \sqrt[2]{\sum_{i=1}^{m} |u_i - q_i|^2} \tag{2.3}$$

Máxima

$$L_\infty = d_\infty(\vec{u}, \vec{q}) = max_{i=1\ldots m}|u_i - q_i| \tag{2.4}$$

2.2. Concepto de Índice Métrico

En espacios métricos los algoritmos que resuelven consultas por proximidad casi siempre se emplean índices para conocer la respuesta sin comparar toda la base de datos como lo haría una búsqueda secuencial.

Un índice puede verse como una estructura de datos que facilita la búsqueda en una base de datos.

Usualmente, un algoritmo que resuelve consultas por proximidad tiene dos fases: la de pre-procesamiento y la de consulta.

Inicialmente se tiene la base de datos y la función de distancia. La proyección de la base de datos en un nuevo espacio permitirá crear un índice (esta es la fase de preprocesamiento). El índice se crea una sola vez y se almacena (para ser utilizado muchas veces). El costo de esta fase no se consideran durante la siguiente fase: consulta.

En la fase de consulta se recorre el índice para conocer la lista de candidatos que deberán ser comparados directamente contra la consulta (*comparaciones externas*). En esta lista de candidatos hay elementos que podrían no ser reelevantes a la consulta (por ejemplo, en una consulta por rango, serían los que están mas alejados que el radio dado).

El caso ideal es que todos los elementos de la lista de candidatos sean relevantes para la consulta.

Una proyección de un espacio métrico se puede ver como una reducción natural. En el caso de los espacios vectoriales, esta proyección podría ser representada como un punto en el nuevo espacio mediante una función τ. El costo de cada proyección es el número de *comparaciones internas*. Los dos espacios están relacionados por dos distancias: la original $d(x,y)$ y la distancia en el nuevo espacio $D(\tau(x),\tau(y))$, donde $x,y \in \mathbb{X}$ [17].

Capítulo 3

Estado de arte

En este capítulo describimos los algoritmos que existen para la búsqueda por proximidad en espacios métricos.

Los índices son estructuras de datos usadas para recorrer con cierto orden los datos y poder tomar decisiones sobre estos, cuando es posible: compararlos o descartarlos. Las propuestas existentes para construir índices consisten en usar ciertos elementos de la base de datos (seleccionados en su mayoría por heurísticas), precalcular y almacenar las distancias entre estos y el resto de la base de datos.

3.1. Algoritmos Existentes y su Clasificación

3.1.1. Algoritmos Basados en Pivotes

La idea general para construir un índice basado en pivotes consiste en elegir un conjunto $\mathbb{P} = \{p_1, p_2, ..., p_k\} \subseteq \mathbb{U}$ de tamaño $k = |\mathbb{P}|$ de pivotes. Para cada elemento de la base de datos se precalculan las distancias hacia los elementos del conjunto $\mathbb{P}$. Este conjunto de distancias $d(p_1,u), d(p_2,u), ..., d(p_k,u),\ \forall u \in\ \mathbb{U}$, es el que finalmente conformará el índice.

Dada una consulta q, se calcula $d(p_i,q),\ \forall p_i \in \mathbb{P}$ (comparaciones internas). Con esto puede acotarse la distancia entre q y cualquier $u \in \mathbb{U}$ tal como se demuestra en el siguiente lema.

Lema 1 *Dados tres objetos $q \in \mathbb{X},\ u \in \mathbb{U}, p \in \mathbb{P}$, sabemos que $d(q,p)$ - $d(p,u) \leq d(q,u) \leq d(q,p) + d(p,u)$.*

Demostración: El límite superior se obtiene directamente de la desigualdad triangular

$$d(q,u) \leq d(q,p) + d(p,u).$$

en el caso del límite inferior, de acuerdo a la desigualdad triangular, se tiene que:

$$d(p,u) \leq d(p,q) + d(q,u),$$

$$d(p,q) \leq d(p,u) + d(u,q),$$

estas desigualdades implican

$$d(p,u) - d(p,q) \leq d(q,u),$$

$$d(p,q) - d(p,u) \leq d(u,q),$$

combinando estas desigualdades y usando la simetría, obtenemos que

$$|d(p,u) - d(p,q)| \leq d(q,u). \clubsuit$$

Este tipo de algoritmos utiliza el índice de la siguiente forma: durante una consulta por rango, usando el lema 1, se pueden descartar todos los elementos de la base de datos $u \in \mathbb{U}$ tales que $\exists\ p \in \mathbb{P}$, $|d(q,p)\text{-}d(p,u)| > r$,pues es claro que si $|d(q,u)\text{-}d(p,u)| > r$, entonces $d(q,u) > r$. Los elementos no descartados se comparan directamente contra la consulta (comparaciones externas).

3.1.2. Algoritmos Basados en Particiones Compactas

Estos algoritmos dividen el espacio en zonas tan compactas como sea posible. La idea general es seleccionar un conjunto de objetos $\mathbb{C} = \{c_1, \ldots, c_k\} \subseteq \mathbb{U}$y dividir el espacio, es decir, distribuir los demás elementos entre las k zonas pertenecientes a cada c_i. Estos c_i son llamados " centros" de su zona. El índice está compuesto por los centros, los elementos que pertenecen a cada zona y, en algunos casos, información adicional sobre las distancias. Un criterio de distribución, entre varios, es asignar cada u a la zona cuyo c_i sea el más cercano. En cada zona se realiza el mismo procedimiento recursivamente hasta que no sea posible o deseable dividir el espacio.

Este tipo de algoritmos se dividen de acuerdo a su procedimiento de búsqueda en: los que usan un radio de cobertura r_c, los que usan los hiperplanos, y los que usan ambos criterios. El radio de cobertura r_c es la distancia máxima del centro de la zona a los elementos en ella. El hiperplano es la frontera entre dos zonas cuando cada elemento se asigna a su centro más cercano.

Durante la búsqueda usando el criterio del radio de cobertura, es posible acotar la distancia entre la consulta y los elementos en la base de datos de la siguiente forma.

lema 2 Dados tres objetos $u, c \in \mathbb{U}, q \in \mathbb{X}$ si c es el centro de la zona a la que pertenece u, con radio de cobertura r_c , sabemos que. $d(q,u)$ está acotada por máx$\{d(c,r) - rc, 0\} \leq d(c,u) \leq d(d,q) + r_c$.

Demostración Ambos límites los obtenemos de la desigualdad triangular y de la cota superior:

$$d(p,q) \leq d(p,u) + d(u,q),$$

$$d(p,u) \leq r_c$$

usando estas dos ecuaciones tenemos

$$d(p,q) - d(q,u) \leq d(p,u) \leq r_c,$$

esto implica que

$$d(p,q) - r_c \leq d(q,u).$$

por otro lado, sabemos que $d(q,u) \geq 0$, por lo tanto el límite inferior es

$$max\ \{d(q,p) - r_c, 0\}\ \leq d(q,u).$$

el límite superior lo obtenemos de la desigualdad triangular

$$d(q,u) \leq d(q,p) + d(p,u) \leq d(q,p) + r_c. \clubsuit$$

Durante la consulta por rango $R_d(q,r)$, esta familia de algoritmos descarta aquellas zonas de centro c tales que: $d(c,q) - r_c > r$, pues usando el lema 2 sabemos que cualquier elemento u en esa zona cumple con $d(q,u) > r$. En las zonas no descartas se repite el proceso hasta que llega a zonas finales o sin división. En estas últimas zonas solo queda hacer una revisión secuencial.

En la figura 3.1 se muestra un ejemplo del uso de esta técnica. El radio de cobertura del centro c está representado por r_c . En la figura toda la zona del centro c puede ser descartada puesto que todos los elementos en ella cumplen con $d(q,p) - r_c \leq d(q,u) y d(q,c) - r_c > r$.

3.1.3. Bases de Datos de Prueba

El rendimiento de los algoritmos presentados en esta tesis fue evaluado usando las bases de datos descritas a continuación.

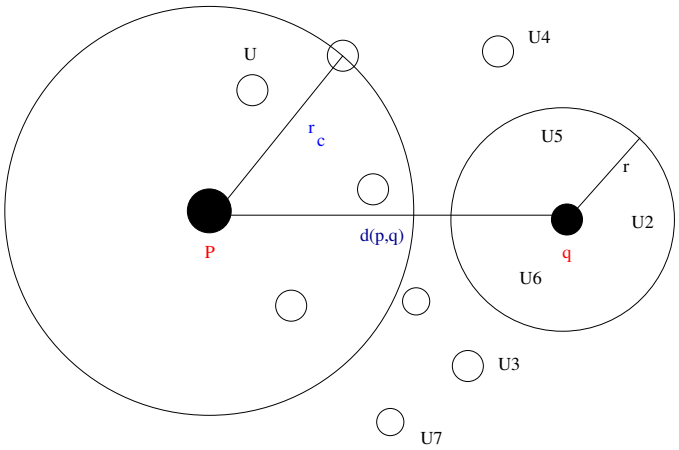


Figura 3.1: Ejemplo del uso de un algoritmo basado en particiones compactas para una consulta de rango $R_d(q,r)$. Los elementos en la zona del centro c pueden ser descartados.

Vectores en el Cubo Unitario

El desempeño de los algoritmos para la búsquedas por proximidad en espacios métricos (consultas de rango o las consultas de los K vecinos más cercanos), decae a medida que la dimensión del espacio crece [5]. Por lo tanto, es interesante experimentar en espacios donde sea posible variar la dimensión.

Una manera de controlar la dimensión del espacio es generar un conjunto uniformemente distribuido en el cubo unitario, y usar este conjunto como un espacio métrico abstracto. La medida de distancia usada en estos espacios fue la Euclidiana.

Las bases de datos generadas fueron de tamaño 10,000. Las dimensiones tratadas varían entre 8 y 128. En los experimentos se promediaron 500 consultas para generar cada punto.

Diccionarios

Otro tipo de espacio usado es un diccionario de 86,062 palabras en español. La distancia entre palabras usada es la distancia de edición o Levenshtein [30]. Esta distancia mide la similaridad entre dos cadenas s_1 y s_2 como el mínimo número de transformaciones que requiere s_1 para convertirse en s_2 o viceversa. Las transformaciones pueden ser: cambiar, insertar o borrar una letra. La siguiente recurrencia define esta distancia entre las dos cadenas. La notación $s_1\ c_1$ representa la

cadena s_1 concatenada con c_1 . Los símbolos c_1 y/o c_2 pueden también representar la cadena vacía.

$$d(s_1c_1, s_2c_2) = \begin{cases} d(s_1, s_2) + 0 \ si \ c_1 = c_2, \\ d(s_1, s_2) + 1 \ si \ c_1 \neq c_2, \end{cases} \tag{3.1}$$

La dimensión intrínseca de esta base de datos es de 12.

3.2. Descripción de los Algoritmos

En esta parte haremos un breve análisis general de los algoritmos existentes. El orden en el que mostraremos estos algoritmos será de acuerdo a la cantidad de memoria que emplean(de mayor a menor). Explicaremos, para cada algoritmo, qué información almacena y cómo la utiliza para descartar elementos durante la consulta. La mayoría de los ejemplos presentados en esta sección fueron tomados de [17].

♣ **AESA Approximating Eliminating Search Algorithm** [42].

Este algoritmo almacena todas las $\frac{n^2}{2}$ distancias entre los elementos de la base de datos. Es de lejos el algoritmo que menos evaluaciones de distancia realiza, pero sólo es aplicable a bases de datos muy pequeñas (del orden de algunos miles). Una característica importante es que el algoritmo elige de manera dinámica los pivotes. Éstos deben ser idealmente aquellos que en cada iteración se vayan aproximando a la consulta q. Para aproximarse a esto se utiliza la desigualdad triangular y los pivotes empleados hasta el momento. Este algoritmo es interesante pues fue el primero que toma en cuenta a la consulta para elegir cuáles y cuántos pivotes usar.Una ventaja de AESA sobre los algoritmos típicos de pivotes, es que los elementos escogidos como pivotes en cada iteración son parte del conjunto no descartado hasta ese momento. Aunque procesar este conjunto es principalmente su mayor costo durante la consulta, una propuesta para reducirlo es ROAESA (Reduced Overhead AESA) [Vil95]. El cual mantiene el mismo cálculo de distancias que AESA, pero seleccionan el próximo pivote de un subconjunto más pequeño de la base de datos.

En la figura 3.2 se muestra un ejemplo de la primera iteración de AESA.

♣ **t-Spanner [39, 38] .**

En este algoritmo se utilizan algunas distancias entre los elementos de la base de datos para construir un grafo. Cada $u \in \mathbb{U}$ representa un vértice en el grafo, y las distancias entre pares de objetos son las aristas. Con la estructura formada se pueden acotar las distancias reales usando caminos mínimos. A la distancia estimada con el grafo la llamaremos D_G . La principal característica de este algoritmo es que garantiza que $d(u,v) \leq D_G(u,v) \leq t\ d(u,v)$, donde t > 1 es un parámetro de

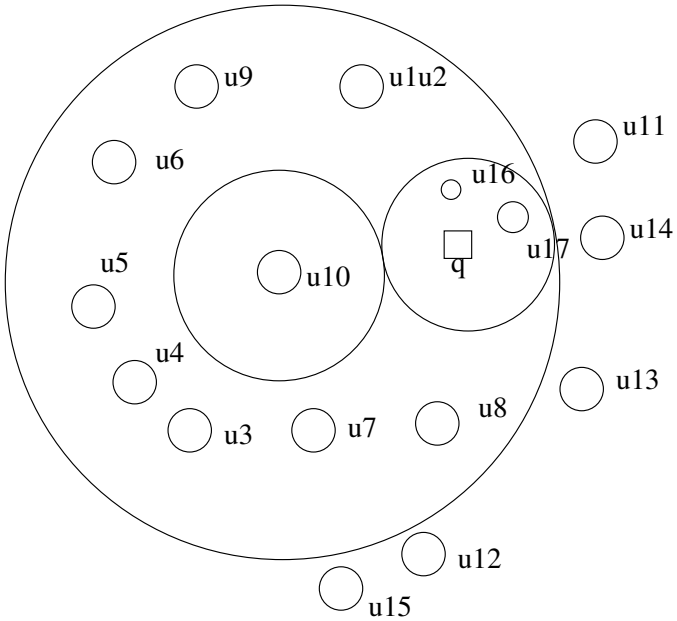


Figura 3.2: Ejemplo de la primera iteración de AESA. Los elementos entre los dos anillos serán los candidatos en la segunda iteración.

construcción. El índice intenta almacenar las mínimas distancias necesarias para garantizar la condición sobre t. El espacio utilizado por el índice es empíricamente $O\left(n^{(1+\frac{0,1-0,2}{t-1})}\right)$. Cada vértice guarda información de las aristas y los elementos conectados a él. Para responder una consulta, este como una simulación de AESA usando menos espacio.

♣ GNAT Geometric Near-neighbor Access Tree [7].

En un GNAT de aridad m, se seleccionan m centros $p_1 ... p_m$, y se define $U_i = u \in \mathbb{U} \mid d(p_i, u) < d(p_j, u)$, para todo $j \neq i$. Esto es, U_i son los elementos más cercanos a p_i que a algún otro p_j . El espacio requerido es $O(nm^2)$ pues e almacena para cada nodo una tabla $rango_{(i,j)} = [\text{mín}_{(u \in U_j)}(p_i, u), \text{máx}_{u \in U_j}(p_i, u)]$, es decir, las distancias mínimas y máximas de cada centro a los otros U_j.

Durante la búsqueda, una consulta q se compara contra algún centro p_i y se descartan otros centros p_j tales que $d(q, p_i)$ mas menos r no intersecte $rango_{i,j}$, de modo que es posible descartar todo el subárbol U_j por la desigualdad triangular. Este proceso se repite tomando otro centro aleatorio hasta que no queden más centros por seleccionar. La búsqueda continúa recursivamenteen los subárboles no descartados. El primer nivel de un GNAT puede verse en la figura 3.3

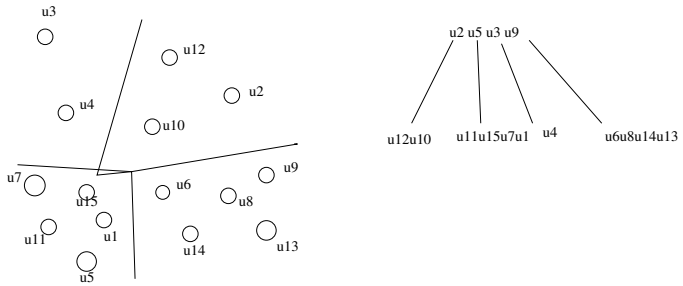


Figura 3.3: Ejemplo del primer nivel de un GNAT con m=4

♣ LAESA Linear AESA [32].

Básicamente este algoritmo es AESA ocupando menos espacio. Los autores proponen utilizar sólo un subconjunto de k pivotes fijos, por lo tanto la memoria utilizada se reduce a $O(kn)$ en lugar de $O(n^2)$ que utiliza AESA. En este conjunto de k elementos se busca el mejor pivote en cada iteración. Una versión mejorada de LAESA fue implementada en un árbol llamado Tree LAESA (TLAESA) [31, 40], el cual resuelve las consultas con un costo sublineal aunque duplica en promedio, los cálculos de distancia. En [34] los autores muestran una extensión de LAESA para encontrar los K vecinos más cercanos (K-LAESA).

♣ FQT Fixed-Queries Tree, FHQT Fixed Height FQT [3] y FQTrie [10].

FQT es un índice diseñado para espacios discretos y su proceso de construcción es el siguiente. Tomamos un conjunto de k pivotes, inicialmente, con el pivote p_1 , y para cada distancia i >0 determinamos el subconjunto de objetos que se encuentran a distancia i de p_1. Para cada subconjunto no vacío se genera una rama con etiqueta i. En cada hijo, con los elementos en éste, se crea un FQT, tomando como raíz el siguiente pivote p_2 . Nótese que todos los subárboles de un mismo nivel usan el mismo pivote como raíz, la altura del árbol es de tamaño k. La información de los elementos de la base de datos está almacenada en las hojas. Para una consulta q y un radio r, se calcula la distancia de q hacia la raíz p_1 , y se descartan aquellas ramas cuya etiqueta $i \notin [d(q,p_1)$ - $r, d((q,p_1)$ $+r]$. Se desciende por las ramas que no fueron descartadas aplicando el mismo procedimiento.

♣ FQA Fixed-Queries Array [16, 15].

Este índice no es propiamente un árbol, sino sólo una representación compacta del FHQT. Imagine que se tiene construido un FHQT de altura h. Si se recolectan las hojas de ese árbol de izquierda a derecha y se colocan los elementos en un arreglo, el resultado es un FQA. Para cada objeto se tienen h números que representan las ramas del árbol desde la raíz para llegar a las hojas. Cada uno de estos h números se codifica en b bits y es posible representar los h números de cada objeto en un número, concatenando los bits de cada distancia. Así los primeros b bits más significativos en realidad representan la distancia de la raíz al primer nivel del árbol, los siguientes b bits representan el siguiente nivel y así sucesivamente.

Como resultado el FQA está almacenado en un número de hb bits. La ventaja de este índice es que usa menos memoria, $O(nhb)$ bits, para obtener los mismos resultados que el FHQT. Este índice es importante porque reduce la memoria usada, y por lo tanto permite incrementar el número de pivotes, teniendo mejor desempeño en la búsqueda. Estos cuatro índices, FQT, FHQT, FQTrie y FQA, usan el mismo criterio para descartar elementos, de hecho si se emplearan los mismos pivotes, los primeros tres índices resolverían la consulta con los mismos cálculos de distancia. La diferencia es el tiempo extra de CPU, siendo el FQTrie el más rápido.

En la figura 3.4 se muestra un pivote p_{11} y la definición de las zonas dadas por las distancias. Cada círculo indica que los elementos ahí están al menos a esa distancia.

♣ BKT Burkhard-Keller Tree [8].

Este índice está diseñado para funciones de distancia discretas. El proceso de construcción es igual que el FQT, la diferencia es que el pivote es distinto en cada nodo del árbol, y se selecciona del conjunto de elementos en su subárbol. Con esto, el pivote es más local al conjunto que indexa y filtra mejor.

En este índice cada pivote p conoce las distancias a sus hijos, es decir, hay muchos pivotes pero cada uno sólo conoce un subconjunto de todas las distancias. Con esto se logra ocupar $O(n)$ espacio, aún teniendo muchos pivotes. El algoritmo de consulta es esencialmente el mismo que para el FQT. Cuando llega una consulta q, se mide la distancia al elemento de la raíz (que ahora es distinto en cada nodo aún en el mismo nivel) y se descartan aquellas ramas cuya etiqueta $i \notin [d(q,p)\text{-}r, d(q,p)\text{+}r]$.

En la figura 3.5 se muestra cómo quedaría la base de datos de la figura 3.4 para cada uno de los índices explicados (FQT, FQA, BKT, FQHT).

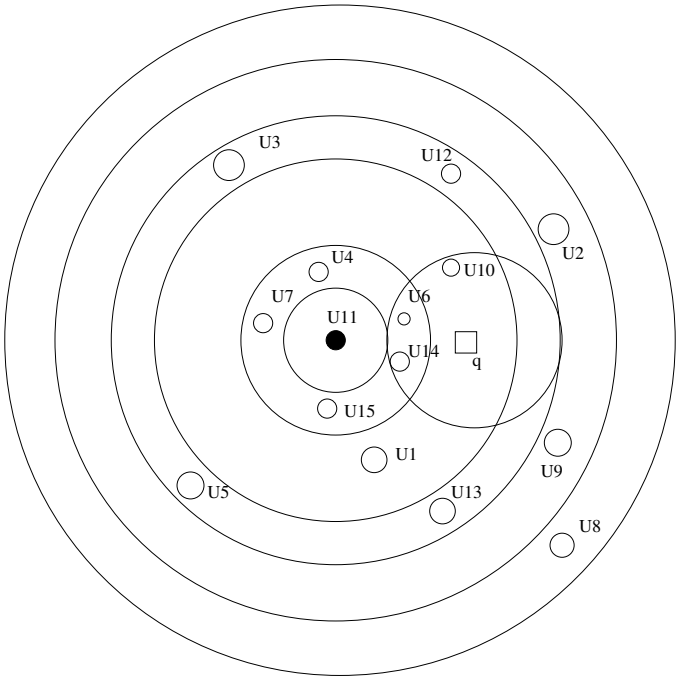


Figura 3.4: En la figura se representan las zonas definidas por un rango de distancias. La consulta (representada con un cuadrado) intersecta 3 zonas. En ellas deberán buscar de manera secuencial.

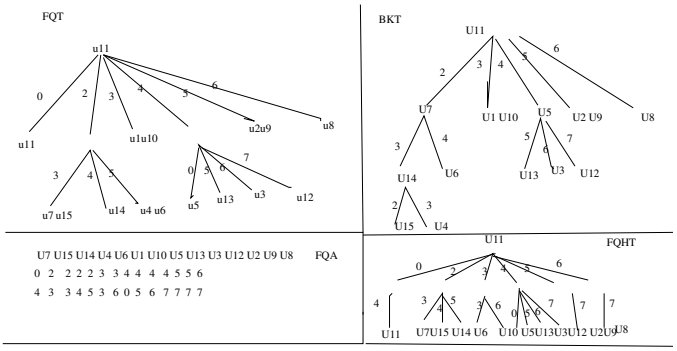


Figura 3.5: Ejemplo la construcción de los índices FQT, FQA, BKT y FQHT [17] para el ejemplo de la figura 3.4.

♣ VPT Vantage Point Tree [41].

Este índice fue diseñado para funciones de distancia continuas. El VPT es un árbol binario; el elemento en cada nodo interno se selecciona de manera aleatoria entre los elementos del subárbol, el criterio de división es la mediana M del conjunto de todas las distancias en el subárbol. Una propiedad interesante de esta estructura es que cada elemento sólo almacena información para saber si la distancia es mayor que M o menor igual a M. La desventaja es que habrá elementos que no fueron descartados tempranamente por no contar con la información exacta.

En la figura 3.6 se presenta la construcción del VPT (lado izquierdo un nivel), lado derecho 3 niveles. El circulo del lado izquierdo representa la distancia a la mediana de los datos.

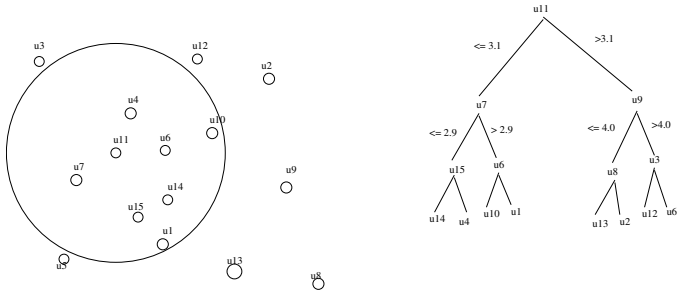


Figura 3.6: Ejemplo del algoritmo VPT cuya raíz es u_{11} . El círculo en el lado izquierdo representa el radio M usado para construir el árbol.

♣ MVPT Multi Vantage-Point Tree [6].

Este índice es una extensión al VPT. Se construye un árbol m-ario usando $m-1$ percentiles uniformes en lugar de sólo la mediana. Los autores mostraron ligeras mejoras al VPT. La memoria utilizada sigue siendo $O(n)$, pues los elementos son almacenados en las hojas. Nótese que los pivotes de este algoritmo también indexan sólo a un subconjunto de elementos, pues son tomados del conjunto de objetos en el subárbol al que pertenecen. Las mejoras reportadas requieren muchos elementos por nodo,lo que hace muy costoso el recorrido en el árbol por más de una rama.

♣ VPF Excluded Middle Vantage Point Forest [46].

Este índice es otra generalización de los VPT, y está diseñado para

responder consultas con un radio limitado (búsquedas de los vecinos más cercanos dentro un radio). El método consiste en excluir, para cada árbol, los elementos que se encuentren en las zonas concentradas. Con los elementos no excluidos se construye un VPT. Con los elementos que fueron excluidos se repite el proceso y se forma un segundo árbol, y así sucesivamente, hasta formar un bosque. Cuando se tiene una consulta se busca sólo en aquellos árboles que tienen intersección con la bola de la consulta.

♣ MT M-Tree [18].

Los objetivos de este algoritmo son reducir el número de cálculos de distancia para responder las búsquedas por proximidad, ser dinámico, tener un buen desempeño en operaciones de I/O. Los autores proponen la construcción de un árbol, donde se escoge un conjunto de objetos representantes, y cada uno forma un subárbol con los elementos cercanos a él. La estructura tiene una cierta semejanza con un GNAT donde los elementos cercanos a un representante son colocados en un subárbol. La principal diferencia de este algoritmo está en cómo son insertados los elementos: en el M-Tree se insertan siempre en el "mejor" subárbol, no siempre en el más cercano como es el caso del GNAT. Los autores consideran mejor subárbol aquel con menor expansión en su radio de cobertura al insertar un elemento. Los elementos son almacenados en las hojas y cuando éstas llenan su capacidad se produce una separación en dos nodos y un elemento es promovido hacia el padre, como en el B-tree o un R-tree [28].

♣ LC List of Clusters [14, 13], iDistance Indexing the Distance [47].

Estos índices tienen muy buen desempeño en dimensiones altas. El algoritmo de construcción (en ambos) consiste en seleccionar un centro c y agrupar sus m elementos más cercanos. El conjunto restante aplica el mismo proceso hasta que todos los elementos hayan sido agrupados. Cada centro almacena un radio de cobertura, que después es usado para descartar elementos en una consulta. El espacio que ocupa la estructura es $O(n)$. Los autores en LC emplean dos criterios para agrupar a los elementos: fijando en m el número de elementos más cercanos, o usando un radio de tolerancia. Los mejores resultados los obtuvieron cuando limitan el número de elementos más cercanos.

♣BST Bisector Tree [29].

Es un árbol binario que se construye recursivamente. Para cada nodo del árbol se eligen dos centros c_1 y c_2 . Los objetos más cercanos a c_1 que a c_2 se insertan en el subárbol izquierdo, y los objetos más cercanos a c_2 se colocan en el subárbol derecho. Para ambos centros se almacena la información de su radio de cobertura respectivo, al que representaremos por r_{c1} y r_{c2} . La memoria utilizada es $O(n)$, pues cada elemento almacena información de la cercanía a su centro.

♣ GHT Generalized-Hyperplane Tree [41].

Es idéntico en construcción a un BST. Sin embargo, el algoritmo utiliza los hiperplanos como condición para descartar ramas del árbol, en lugar del radio de cobertura. El espacio requerido por esta estructura es $O(n)$. La construcción de este árbol y de un BST se puede ver en la figura 3.7, donde u_2, u_5 son los centros escogidos.

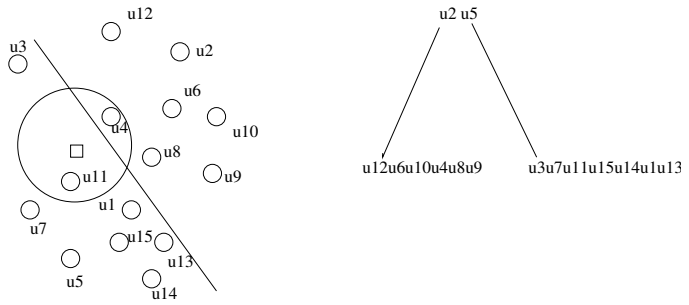


Figura 3.7: Ejemplo del primer nivel del BST o GHT. En este caso, note que para la consulta q deben revisarse ambos subárboles.

♣ VT Voronoi Tree [20].

Este algoritmo es una mejora al BST. El VT tiene 2 ó 3 elementos por nodo. Cuando se crea un nuevo nodo, se insertará en aquél elemento más cercano que pertenece al padre. Este algoritmo tiene la propiedad de que el radio de cobertura se reduce a medida que se desciende por el árbol. Los autores muestran que el VT es superior y mejor balanceado que el BST. El espacio requerido es $O(n)$ pues los pivotes indexan localmente.

♣ **SAT Spatial Approximation Tree [37, 36].**

Este algoritmo no usa centros para separar la base de datos, sino la aproximación espacial. Se selecciona un elemento p como raíz del árbol y a éste se conectan todos los vecinos en $\mathbb{U}$ tales que están más cerca a p que a otro vecino. Los demás se insertan en su vecino más cercano. El algoritmo es recursivo para los elementos que se van insertando en cada vecino. La cantidad de memoria utilizada es $O(n)$ pues cada elemento almacena cuáles son sus vecinos y a qué distancia se encuentran. La figura 3.8 muestra un SAT construído y la línea marcada es el recorrido que haría para conocer el vecino mas cercano de la consulta dada q.

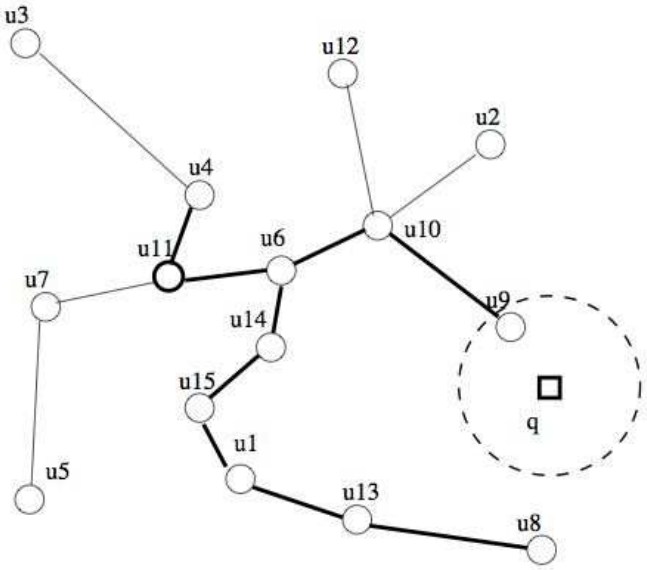


Figura 3.8: Ejemplo de un SAT. El recorrido que se sigue para esa consulta se muestra en la figura con la línea más oscura,

3.3. Comparación del Estado del arte

Es importante resaltar cómo los algoritmos expuestos usan las distancias precalculadas, y guardadas en el índice, al momento de realizar consultas por proximidad. De esta manera, algunos elementos de la base de datos serán descartados sin que se mida su distancia contra la consulta y así se reducirán los cálculos de distancia para responder las consultas.

La comparación entre todos estos algoritmos es evidente al notar que

mientras AESA usa toda la matriz de distancias, el resto de los algoritmos tratan de acercarse lo más posible al desempeño de AESA pero almacenando mucha menos memoria. La manera de usar menos memoria es almacenando sólo unas columnas de esa matriz (algoritmos pivoteros), algunas áreas ó manteniendo cotas inferiores y/o superiores (particiones compactas). Otra comparación es que en AESA no existen comparaciones externas, como en el resto de los algoritmos, pues el próximo pivote siempre es seleccionado de la lista de candidatos.

En el caso de los algoritmos basados en pivotes, se ha demostrado [2, 4] que existe un número óptimo de pivotes para usarse en el índice. La desventaja es que ese número puede no ser alcanzable por falta de memoria (incluso en bases de datos de tamaño moderado).Por lo tanto, optimizar la memoria que ocupan los algoritmos en el índice permite acercarse al número óptimo de pivotes y tener el mejor rendimiento posible.

Durante el análisis de los distintos algoritmos se mostraron características diversas, que son importantes para comprender por qué nuestro interés en usar los índices centrados en los datos para optimizar la memoria ocupada.

Objetos representativos (OR) Fijos Algunos índices primero escogen los OR y en base a ellos se construye la estructura de su índice.

De cada subconjunto. Significa que, después del primer particionamiento (zonas o subárboles), los OR son seleccionados de los elementos que pertenecen a cada partición para continuar recursivamente el proceso.

Todos OR. Algunos algoritmos usan todos los objetos como OR, como AESA.

Durante la fase de consulta, este criterio refleja el tipo de información que se empleará en el recorrido del índice. En el caso de los fijos se compara la consulta contra todos los OR, y ésta es toda la información que se usará durante el recorrido por el índice, sin que alguna distancia extra calculada en el proceso agregue más información. De antemano se sabe contra quién se comparará la consulta. En el caso De cada subconjunto, de acuerdo a la naturaleza de la consulta, ésta se comparará contra diferentes OR y cada comparación irá contribuyendo al proceso de búsqueda. Finalmente, en el caso Todos OR, sucede algo similar al caso anterior, la diferencia radica en que cualquier elemento de la base de datos puede ser un OR.

Cabe destacar que una gran mayoría de los algoritmos presentados selecciona sus OR de manera aleatoria. Asimismo, en ninguno de los algoritmos se puede pertenecer a dos subárboles o agrupaciones a la vez.

Algunos algoritmos presentados pueden mejorar su rendimiento usan-

do más memoria. El caso límite es AESA pues requiere toda la información de las distancias disponible para la búsqueda.

Algunos índices fueron omitidos, debido a que su desempeño es muy semejante a otro, por ejemplo, el FQT, FHQT y FQTrie con respecto al FQA; o porque son casos particulares de otro algoritmo, por ejemplo, VPT, MVPT del VPF, el BST, GHT del GNAT.

3.3.1. Algoritmos Inexactos

La parte anterior fueron los algoritmos exactos los cuales recuperan correctamente los elementos de la base de datos que se encuentran en la bola $(q, r)_d$. Se hará una descripción de los algoritmos inexactos para espacios métricos. Este tipo de algoritmos relaja la precisión de la consulta y con esto gana velocidad en los tiempos de respuesta [44, 43, 40]. Esto generalmente es razonable en muchas aplicaciones debido a que modelar como espacios métricos ya involucra una aproximación a la respuesta verdadera y por lo tanto podría ser aceptable una segunda aproximación durante la búsqueda.

Adicionalmente a la consulta se especifica un parámetro de precisión ϵ el cual controla cuál lejos (o relajada) se quiere la respuesta de la solución correcta a la consulta. Un comportamiento razonable para este tipo de algoritmos es una tendencia asintótica a la respuesta correcta cuando ϵ tiende a cero y complementariamente acelerar el algoritmo, perdiendo precisión, a medida que ϵ mueve en la dirección contraria. De esta manera, la precisión está perfectamente.

Esta alternativa para la búsqueda por similaridad exacta es llamada búsqueda por similaridad inexacta, y abarca los algoritmos aproximados y probabilísticos. En [45] se presentan algunos algoritmos inexactos para la búsqueda por similaridad.

♣ Aggressive Pruning [46].

Este es un algoritmo probabílistico para espacios vectoriales. La estructura es muy semejante a un KD-Tree, usando un "recorte agresivo de hojas" para mejorar el desempeño. La idea es incrementar el número de ramas filtradas (o recortadas) a expensas de perder algunos candidatos en el proceso. Este proceso es controlado, de esta manera, siempre se conoce la probabilidad de éxito. La estructura de datos es útil sólo para encontrar vecinos más cercanos en un radio limitado, esto es, vecinos más cercanos dentro de una distancia fija a la consulta.

♣ M(U, Q) [19].

Este es un ejemplo de un algoritmo probabilístico para espacios métricos generales. La intención original es construir una estructura de datos como la de Voronoi para un espacio métrico. En general esto no es posible porque no existe suficiente información de las características de las futuras consultas. Para solucionar este problema, los autores seleccionaron entre las consultas un " conjunto de entrenamiento " y construyen una estructura de datos capaz de responder correctamente sólo las consultas que pertenecen al conjunto de entrenamiento. La idea es que con esto sea posible resolver consultas arbitrarias con alta probabilidad. Bajo algunas suposiciones en la distribución de las consultas, mostraron que la probabilidad de no encontrar el vecino más cercano es $O((\log n)2/\alpha)$, donde α es un parámetro que permite controlar la probabilidad de falla. La estructura ocupa $O(\alpha np)$ espacio y $O(\alpha p \log n)$ tiempo de búsqueda. Aquí p es el logaritmo del radio entre los pares de puntos más alejados y más cercanos en la unión de U y del conjunto de entrenamiento.

♣ Streching [15].

Este algoritmo está basado en "reducir" la desigualdad triangular. La propuesta es general, aunque los autores la mostraron para los algoritmos basados en pivotes. La idea es reducir el radio de búsqueda por un factor β, que debe crecer cuando el espacio se vuelve más difícil (cuando la dimensión intrínseca aumenta).

♣ Algoritmo Probabilístico [9].

Los autores proponen dos técnicas probabilísticas. La primera es un adaptación al algoritmo de búsqueda de vecinos más cercanos priorizando el backtracking para los algoritmos basados en pivotes, y la segunda usa el ranking de zonas para algoritmos basados en particiones. La segunda técnica mostró un mejor desempeño que la primera en dimensiones altas.

La idea esencial en este tipo de algoritmos consiste en ordenar la base de datos con algún criterio propicio y comparar los elementos en ese orden. El criterio para no comparar más elementos se establece generalmente usando heurísticas. El caso ideal en estos algoritmos es que los primeros elementos del orden establecido en la base de datos sean la respuesta correcta, en el caso de los K vecinos más cercanos.

♣ **AK-LAESA [35].**

Este algoritmo es una extensión al algoritmo exacto LAESA para encontrar los K vecinos más cercanos. Básicamente tiene dos modificaciones, la primera consiste en proponer un orden en el que los elementos deben ser comparados. La segunda modificación es que los autores detienen las comparaciones cuando $L_\infty > d_{NN}$, donde d_{NN} es la distancia al vecino más cercano,

♣ **M(U, Q) [19].**TLAESA Aproximado [40].

Este algoritmo aproximado mejora LAESA recorriendo un árbol de caminos múltiples por la mejor rama. La prioridad de una rama está dada por L_∞, y la condición para podar una rama es que $L_\infty > \alpha \dot{d}_{min}$, donde d_{min} es la distancia al vecino más cercano encontrado, y $0 < \alpha \leq 1$ es un parámetro para controlar la aproximación de la consulta a la respuesta correcta.

En fin existen varias propuestas para búsquedas inexacta las cuales son:

Aproximada. En este tipo de búsquedas el usuario define un valor $0 <\in\leq 1$ que indica qué tan relajada acepta una solución respecto a la respuesta correcta. Muchos algoritmos exactos usan esta estrategia para conseguir mejorar su desempeño a cambio de perder precisión.

Probabilística. Otra forma de búsqueda inexacta son los algoritmos probabilísticos donde se especifica la probabilidad de error en la respuesta. Una variante de este tipo de algoritmos son los basados en ordenamientos. En este tipo de heurísticas se propone un orden (por promisoriedad) en que deben ser comparados los elementos de la base de datos. La idea es identificar rápidamente los elementos prometedores y a medida que se degrade la calidad de la respuesta dejar de comparar elementos en la base de datos.

En la figura 3.9 se muestra gráficamente la diferencia entre un algoritmo exacto y uno probabilístico.

3.4. Algoritmo Basado en Permutaciones

Esta propuesta se basa en modificar uno de los índices mas recientes en esta área, los algoritmos basados en permutaciones. Los autores en [11, 12] presentaron la propuesta de índice basado en los datos. La idea general es: cada elemento de la base de datos guardará el orden (creciente) en que se encuentran los

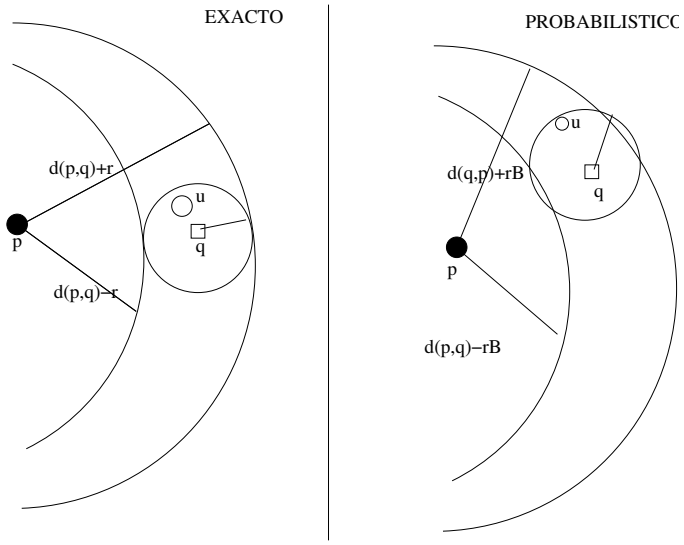


Figura 3.9: : Diferencia entre un algoritmo exacto y uno aproximado, ambos desarrollados para un algoritmo basado en pivotes.

objetos representantes.

Formalmente, sea $\mathbb{P} = \{p_1, \ldots, p_k\}$ un conjunto de objetos representantes al que llamaremos *permutantes*, donde $\mathbb{P} \subseteq \mathbb{U}$. Cada elemento $u \in \mathbb{U}$ calcula su distancia al conjunto de permutantes, esto es $d(u, p_1), \ldots, d(u, p_k)$ y las ordena de manera ascendente. u guardará este orden conservando solo el número de permutante, las distancias se descartan. A esto se le llamará permutación, Π_u. Por ejemplo, $\Pi_u = \{3, 2, 4, 5, 1, 6\}$, significa que el permutante p_3 está mas cerca de u que cualquier otro permutante. Este fase se conocerá como preprocesamiento.

La ventaja de esta técnica de permutaciones es que es posible obtener rápidamente un alto porcentaje de la respuesta correcta, lo que da como resultado un algoritmo probabilístico.

Intuitivamente, si dos elementos u y v son cercanos entre sí, sus permutaciones serán similares. En particular, si dos elementos u, v son tales que $d(u, v) = 0$, las permutaciones $\Pi_u y \Pi_v$ serán iguales. Basándose en esta observación se quiere revisar primero aquellas permutaciones con mayor semejanza de la consulta, esperando que las permutaciones sean un buen predictor de cercanía entre elementos. Por lo tanto, el objetivo de esta técnica es identificar a los elementos más cercanos entre sí usando la semejanza entre las permutaciones.

Al momento de la fase de consulta, calculamos Π_q para la consulta q sobre el mismo conjunto $\mathbb{P}$. Luego, para saber en qué orden revisar los elementos,

ordenamos los Π_u, $u \in \mathbb{U}$ de mayor a menor similaridad con respecto a Π_q . De esta forma, esperamos que los elementos con permutaciones muy similares a Π_q estén espacialmente cerca de q, y sean los más cercanos a la consulta.

Como medida de similaridad entre permutaciones los autores propusieron usar Spearman Rho [22], la cual esta denotada por $S_\rho(\Pi_q, \Pi_u)$. S_ρ es la suma de los cuadrados de las diferencias en las posiciones relativas de cada elemento en las dos permutaciones. Para cada $p_i \in \mathbb{P}$ calculamos su posición en Π_u y Π_q, esto es por medio de su permutación inversa, $\Pi_u^{-1}(p_i)$ y $\Pi_q^{-1}(p_i)$, respectivamente. Formalmente

$$S_\rho(\Pi_q, \Pi_u) = \sqrt{\sum_{i=1}^{k} |\Pi_u^{-1} - \Pi_q^{-1}|^2} \tag{3.2}$$

Una alternativa con desempeño semejante es Spearman Footrule. Formalmente, esta función es:

$$S_\rho(\Pi_q, \Pi_u) = \sum_{i=1}^{k} |\Pi_u^{-1} - \Pi_q^{-1}| \tag{3.3}$$

La principal diferencia es la potencia de la diferencia que calcula Spearman Rho, note que Spearman Footrule no realiza este cálculo ni la raíz cuadrada.

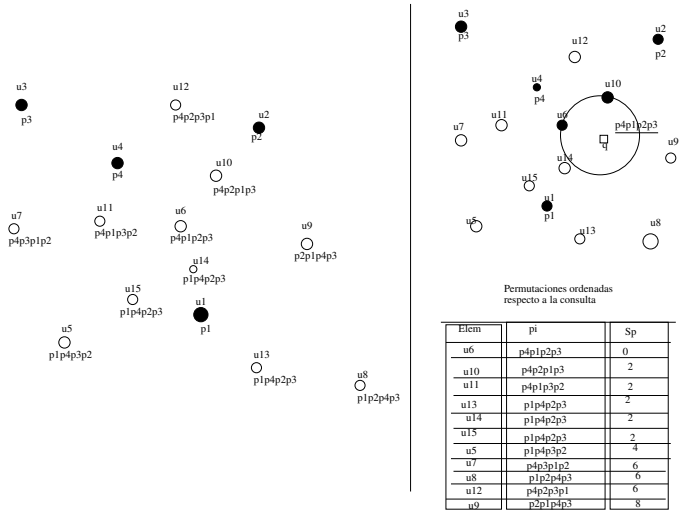
Recientemente se han propuesto diferentes modificaciones a la técnica de permutaciones, por ejemplo, en [1] los autores proponen un índice invertido para conocer las permutaciones semejantes con un costo mínimo de la distancia de Spearman footrule.

Otras propuestas han variado desde: propuestas de selección de permutantes [24], o considerar grupos de permutantes en lugar de usar solo uno fue propuesto en [33], incluso propuestas de generar un suffix tree para indexar las permutaciones puede verse en [21].

Las propuestas mas recientes de este índice son: FQT basado en permutaciones [27], en [26] proponen usar mas información discretizada sobre las distancias que no se almacenan. Cada algoritmo de estos están basados en Sperman Footrule o en Spearman Rho.

Ejemplo de la Técnica Basada en Permutaciones

En la figura 3.10 se muestra un ejemplo de las dos fases de un algoritmo basado en permutaciones; la fase de preprocesamiento (lado izquierdo) y la de consulta (lado derecho). En la fase de preprocesamiento se calculan las permutaciones para todos los elementos de la base de datos. En la fase de consulta, primero se calcula la permutación para q. Los elementos son ordenados respecto a la similaridad entre permutaciones (usando S_ρ) y finalmente comparados secuencialmente en ese orden (primero u_6 , luego u_{10} , etc.).



Elem	pi	Sp
u6	p4p1p2p3	0
u10	p4p2p1p3	2
u11	p4p1p3p2	2
u13	p1p4p2p3	2
u14	p1p4p2p3	2
u15	p1p4p2p3	2
u5	p1p4p3p2	4
u7	p4p3p1p2	6
u8	p1p2p4p3	6
u12	p4p2p3p1	6
u9	p2p1p4p3	8

Figura 3.10: Ejemplo de las dos fases de un algoritmo basado en permutaciones. Las permutaciones fueron ordenadas por valor de S_ρ respecto a la permutación de la consulta.

Parte II

PROPUESTA

Capítulo 4

Otras Medidas de Disimilaridad entre Permutaciones

En este capítulo se presenta la propuesta de este trabajo. Como se mencionó en el capítulo anterior, todo el estado del arte de los algoritmos basados en permutaciones utilizan Spearman Footrule 3.3 o Spearman Rho 3.2, vistos en el capítulo 3, sección 3.4. Actualmente no existe ninguna otra propuesta para modificar la forma de comparar las permutaciones, y ésta es la idea central de este trabajo.

4.1. Idea Principal

Dado que buscamos permutaciones similares, por ser una buena referencia sobre el espacio que indexan, para esta propuesta nos basaremos en que tanto se mueve un permutante en dos permutaciones a compararse. La motivación de esto es que si un elemento tiene como permutante mas cercano al p_i, y la permutación de la consulta tiene a ese permutante como el mas alejado de todos, significaría que este elemento u es poco promisorio para ser parte de la respuesta. En nuestra propuesta esta inversión será penalizada con mayor valor.

Por otro lado, imagine que u tiene dos permutantes en este orden p_i, p_j, y que la consulta q los tiene en este orden p_j, p_i. Con esta inversión no refleja si es promisorio o no dicho elemento. En nuestra propuesta esta inversión debería tener una penalización con menor valor.

Con esta propuesta pretendemos, penalizar las inversiones grandes (i.e. la diferencia de posiciones de un elemento en dos permutaciones) y no castigar

tanto las inversiones pequeñas.

4.2. Medidas de Disimilaridad Propuestas

En esta sección daremos formalmente nuestra propuesta, sea la función f:

$$f(\Pi_q, \Pi_u) = \sum_{i=1}^{k} |\Pi_u^{-1} - \Pi_q^{-1}|^{\alpha} \tag{4.1}$$

Note que en la ecuación 4.1 introducimos una variable α que nos permitirá penalizar las inversiones grandes.

Definiremos :

$$\varphi_i = |\Pi_u^{-1}(i) - \Pi_q^{-1}(i)| \tag{4.2}$$

Reescribiendo la ecuación 4.1, tendríamos:

$$f(\Pi_q, \Pi_u) = \sum_{i=1}^{k} \beta_i \tag{4.3}$$

Nuestra propuesta consiste en evaluar *phi* y dependiendo del valor, penalizar o no. Para medir este valor se usará un umbral μ.

Considerando un Umbral μ para φ y $\alpha = 2$

Esta medida consiste básicamente de la ecuación 4.3, evaluando $\alpha = 2$, para un valor de φ mayor de umbral determinado experimentalmente.

$$\beta_i = \begin{cases} \varphi_i^{\alpha} & si\ \varphi > \mu \\ \varphi_i & si\ \varphi \leq \mu \end{cases} \tag{4.4}$$

Considerando un Umbral μ para φ y $\alpha = \log_{10} \varphi$

Esta medida consiste de la ecuación 4.3 darle los parámetros parecidos a la anterior es decir si φ es mayor a un umbral μ, entonces α sería $\log_{10}(\varphi)$.

$$\beta_i = \begin{cases} \varphi_i^{\log_{10}(\varphi_i)} & si\ \varphi > \mu \\ \varphi_i & si\ \varphi \leq \mu \end{cases} \tag{4.5}$$

Considerando un Umbral μ para φ y $\alpha = 2$ sin centro

Esta medida de similaridad es muy parecida a la ecuación 4.4 solo que la diferencia es que en esta no tomamos en cuenta lo que quede en el centro de la permutación es decir añadimos una nueva condición, por ejemplo, sea $\Pi_q = \{4,6,5,3,1,2\}$

$$\Pi_q = [\ \underbrace{4,6}_{\theta}\ ,\ \underbrace{5,3}_{\gamma}\ ,\ \underbrace{1,2}_{\delta}\ ] \tag{4.6}$$

En este caso, γ sería eliminado de las permutaciones. La razón de quitar a la parte central está fundamentada en el histograma de las distancias de los permutantes a los elementos de la base de datos. Necesariamente i debería estar entre los primeros o últimos elementos de la permutación.

$$\beta_i = \begin{cases} \varphi_i^{\alpha} & si\ \varphi > \mu \\ \varphi_i & si\ \varphi \leq \mu \end{cases} \tag{4.7}$$

Considerando un Umbral μ para φ y $\alpha = \log_{10}\varphi$, sin centro

Esta medida de similaridad es muy parecida a la presentada en la ecuación 4.5 solo que al igual que en la anterior añadimos la misma condición para descartar la parte central de las permutaciones

$$\beta_i = \begin{cases} \varphi_i^{\log_{10}(\varphi_i)} & si\ \varphi > \mu \\ \varphi_i & si\ \varphi \leq \mu \end{cases} \tag{4.8}$$

En el siguiente capítulo presentaremos el desempeño de estas nuevas medidas de disimilaridad

Capítulo 5

Experimentación

En esta sección presentaremos el comportamiento de nuestra propuesta de manera experimental. Primero comenzaremos con bases de datos sintéticas generadas para controlar la dimensión y la cantidad de los datos, y después seguiremos con bases de datos reales. La idea general es primero controlar los datos para estudiar los parámetros, y con este conocimiento pasar a las bases de datos reales. Los parámetros estudiados fueron: cantidad de permutantes usados y el valor del umbral.

5.1. Bases de Datos Sintéticas

Para este tipo de bases de datos se generaron sintéticamente vectores distribuídos uniformemente en el cubo unitario. La distancia utilizada fue la Euclideana. Se generaron bases de datos con 100,000 objetos en distintas dimensiones (32, 64, 128).

5.1.1. Dimensión 128

La primera de las bases de datos usadas fue de dimensión 128.

En la fig: 5.1 podemos apreciar que en una dimensión alta a las medidas de similitud ya existentes (Footrule y SpearmanRho) son superadas por las nuevas propuestas ($\alpha = 2, \mu \geq 15$).

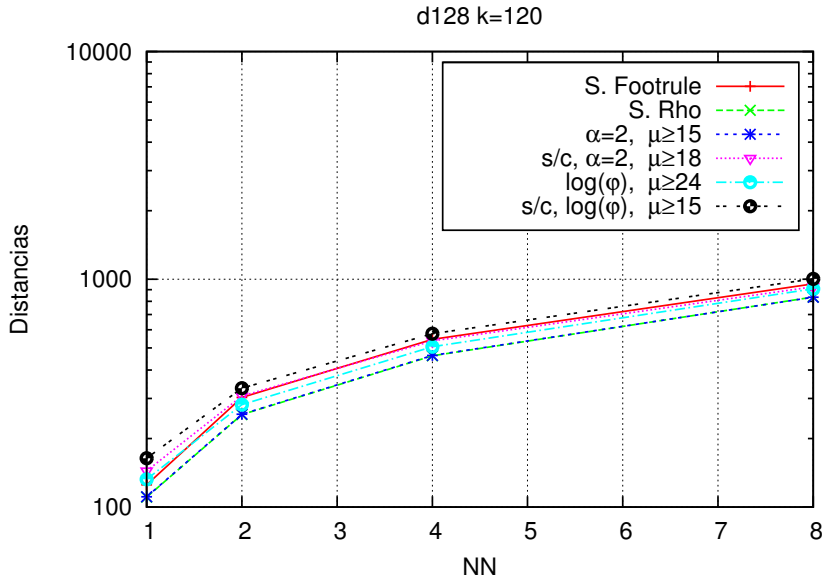


Figura 5.1: La cantidad de distancias que calcula variando los vecinos mas cercanos en una dimencion de 128 con 120 permutantes.

5.1.2. Dimensión 64

Otra base de datos sintética usada fue variando la dimensión a solo 64.

Variando los Vecinos mas Cercanos

En la Figura 5.3 se muestra el comportamiento de las propuestas variando el número de vecinos buscados en dimensión 64 con 57 permutantes. Note que Spearman Rho tiene el mejor desempeño junto con usar $\log_{10}(\phi)$ con $\mu \geq 24$.

En dimensión 64 usando solo 16 permutantes también se consiguen buenos resultados como se muestra en la figura 5.2. Note que se tiene una ligera mejoría respecto a la técnica original.

Las figuras 5.4, 5.5, 5.6, 5.7, 5.8 y 5.9 muestran el comportamiento de las nuevas propuestas en dimensión 64 variando la cantidad de permutantes utilizados. Note que la mejor estrategia es usando $\alpha = 2$ y μ entre 15 y 27. El valor del umbral está directamente relacionado con la cantidad de permutantes usados.

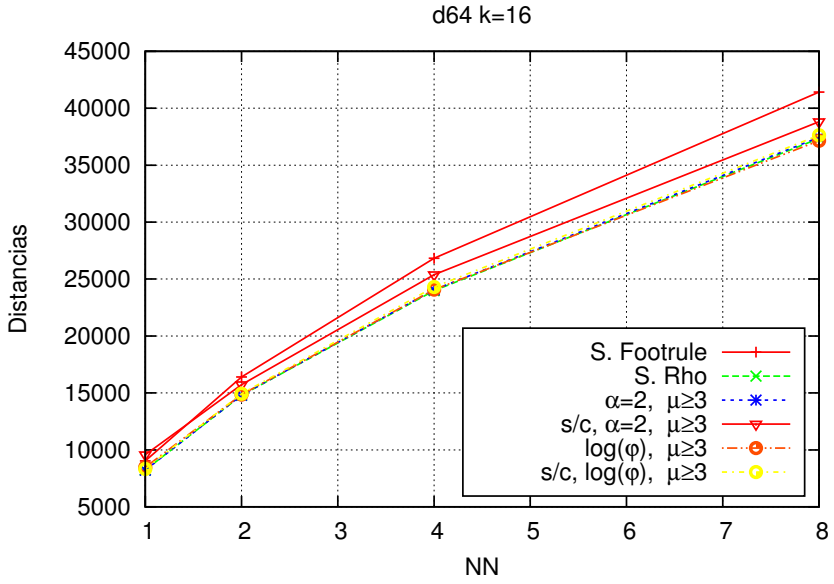


Figura 5.2: La cantidad de distancias necesarias para encontrar vecinos mas cercanos en una dimencion de 64 con 16 permutantes.

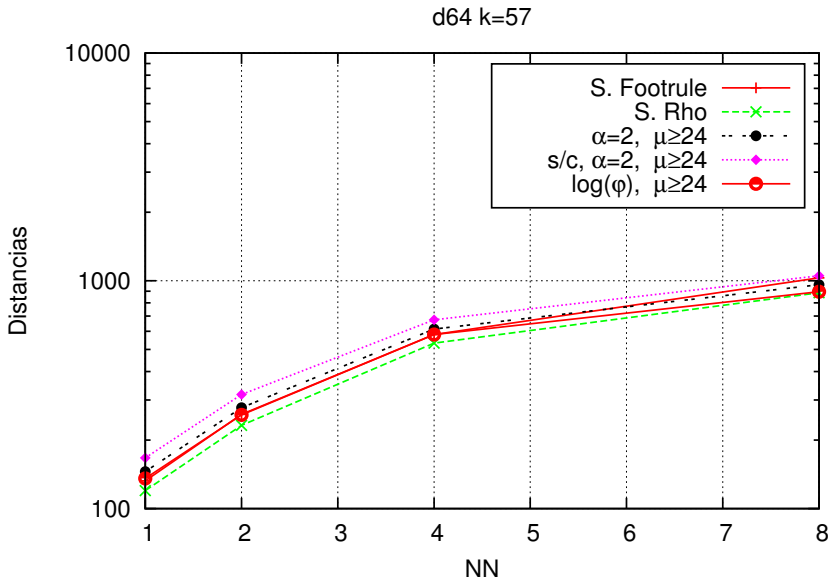


Figura 5.3: La base de datos de vectores variando la cantidad de vecinos mas cercanos.

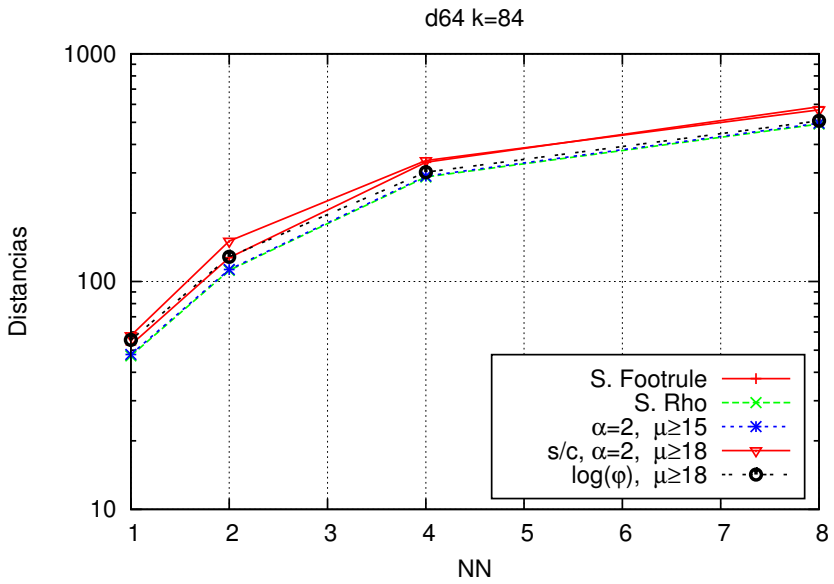


Figura 5.4: La base de datos de Vectores calculando la cantidad de distancias variando los 8 vecinos mas cercanos en una dimension de 64 con 84 permutantes.

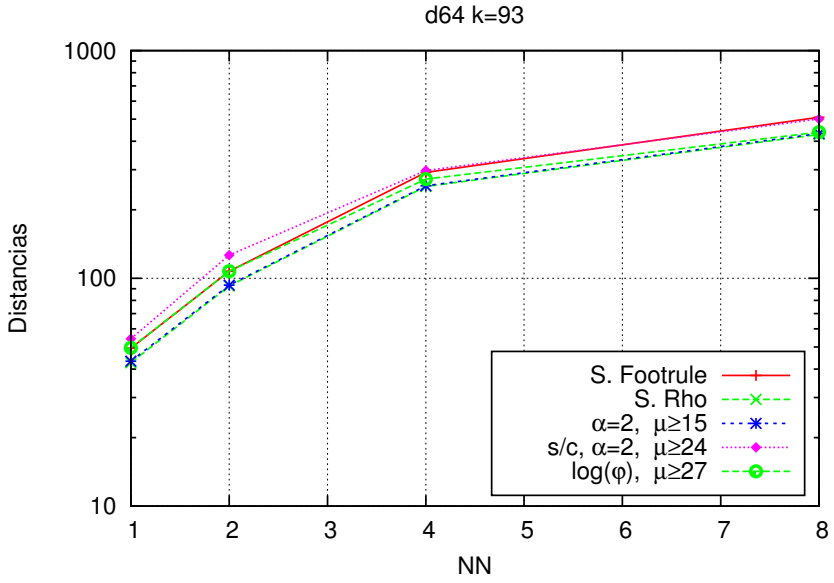


Figura 5.5: La base de datos de Vectores calculando la cantidad de distancias variando los 8 vecinos mas cercanos en una dimension de 64 con 93 permutantes.

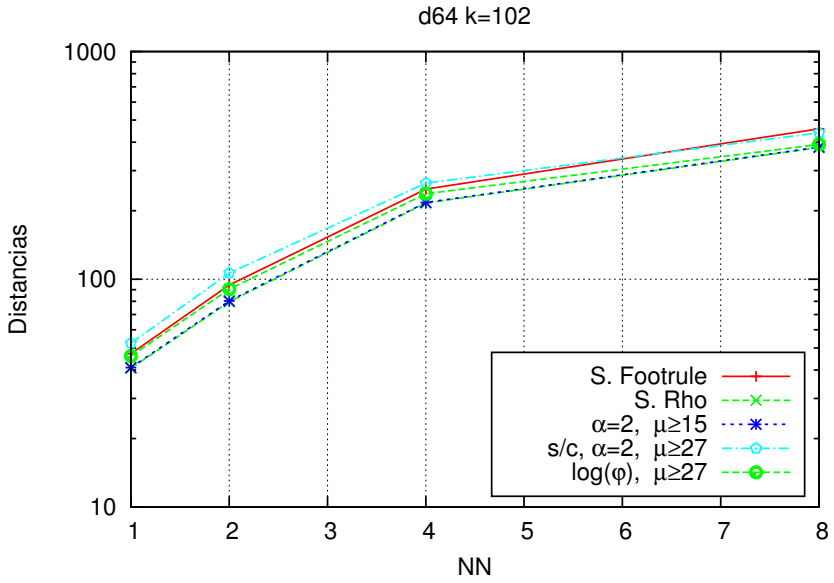


Figura 5.6: La base de datos de Vectores calculando la cantidad de distancias variando los 8 vecinos mas cercanos en una dimension de 64 con 102 permutantes.

Variando el Número de Permutantes

En las siguientes gráficas se fijó el número de vecinos a recuperar y se variaron la cantidad de permutantes. Véanse las figuras 5.10 (para 1NN), 5.11 (para 2NN), 5.12 (para 4NN) y 5.13 (para 8NN). Note que en general se mejora el número de cálculos de distancia cuando incrementa la cantidad de permutantes. En esta dimensión la propuesta que hace la menor cantidad de distancias es usando $\alpha = 2$.

5.2. Bases de Datos Reales

En esta sección se muestra el desempeño de la propuesta en bases de datos reales. Se presentarán 2 bases de datos reales: un histograma de colores de imágenes obtenidas (la llamaremos COLORs) y otra base de datos con imágenes de la NASA. En cada sección se darán detalles de estas BD.

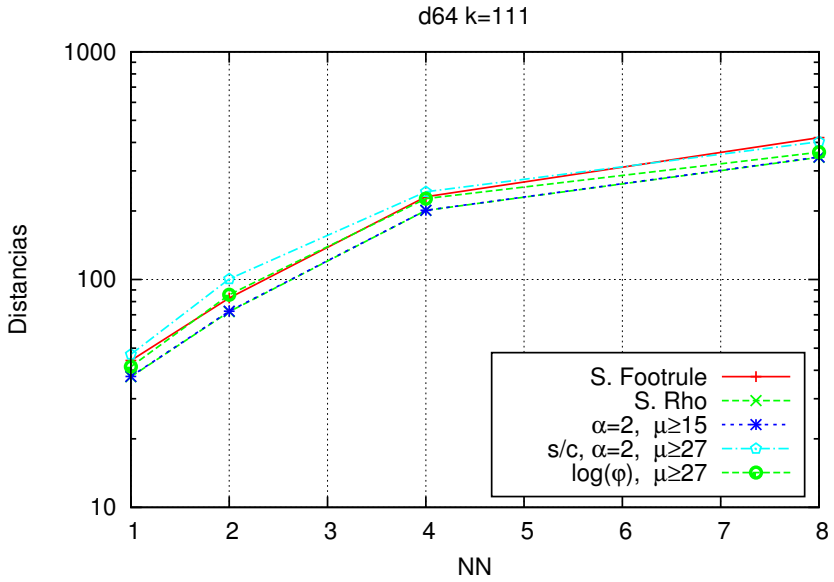


Figura 5.7: La base de datos de Vectores calculando la cantidad de distancias variando los 8 vecinos mas cercanos en una dimension de 64 con 111 permutantes.

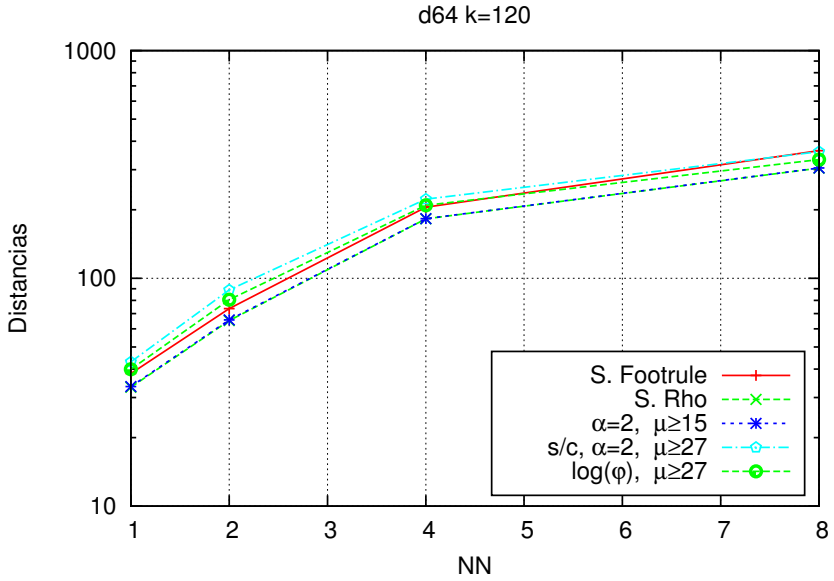


Figura 5.8: La base de datos de Vectores calculando la cantidad de distancias variando los 8 vecinos mas cercanos en una dimension de 64 con 120 permutantes.

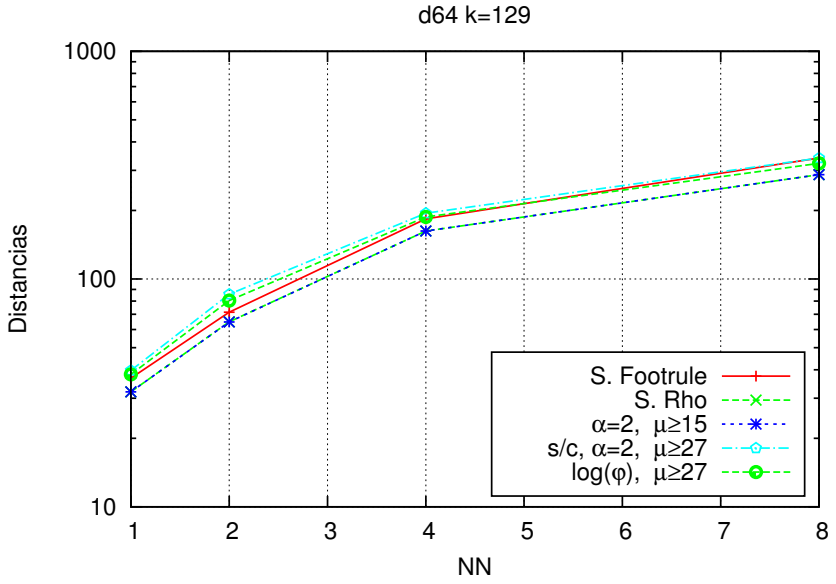


Figura 5.9: La base de datos de Vectores calculando la cantidad de distancias variando los 8 vecinos mas cercanos en una dimension de 64 con 129 permutantes.

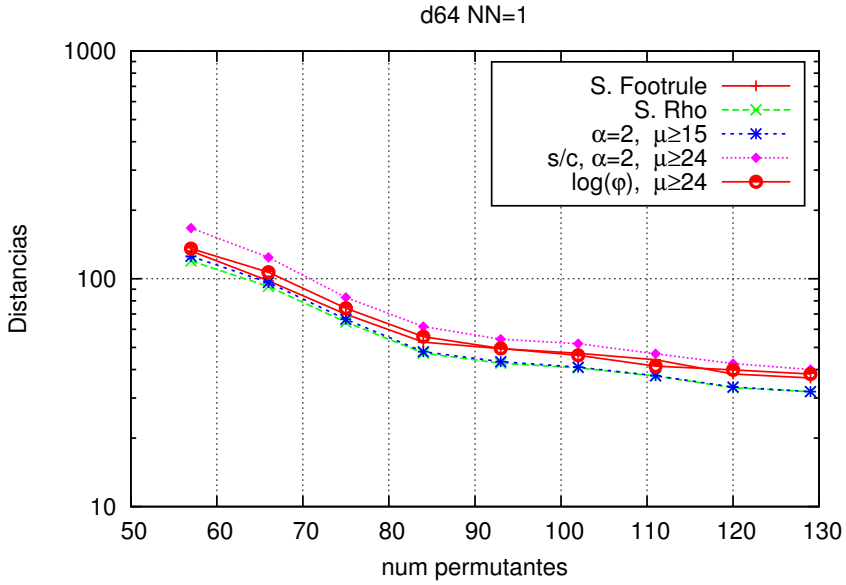


Figura 5.10: Número de cálculos de distancias para resolver 1NN en una Base de datos sintéticas en una dimension de 64.

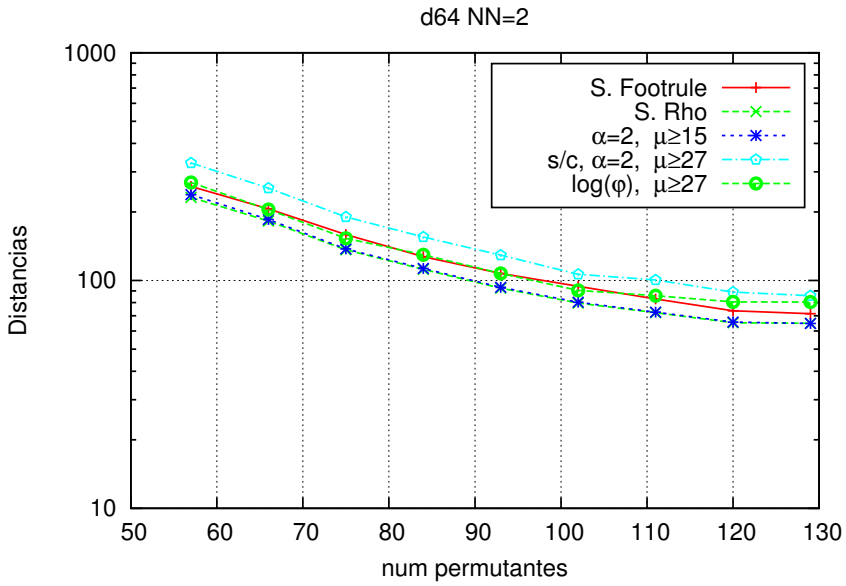


Figura 5.11: Número de cálculos de distancias para resolver 2NN en una Base de datos sintéticas en una dimension de 64.

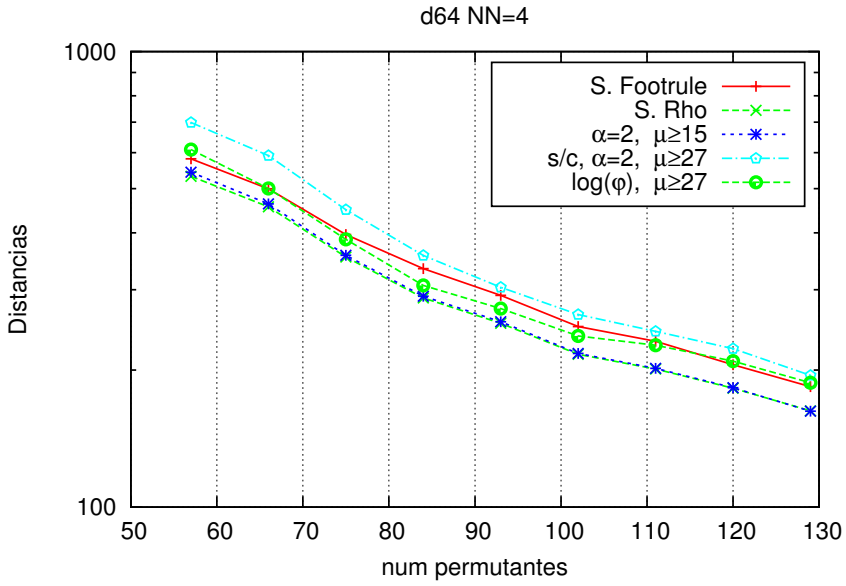


Figura 5.12: Número de cálculos de distancias para resolver 4NN en una Base de datos sintéticas en una dimension de 64.

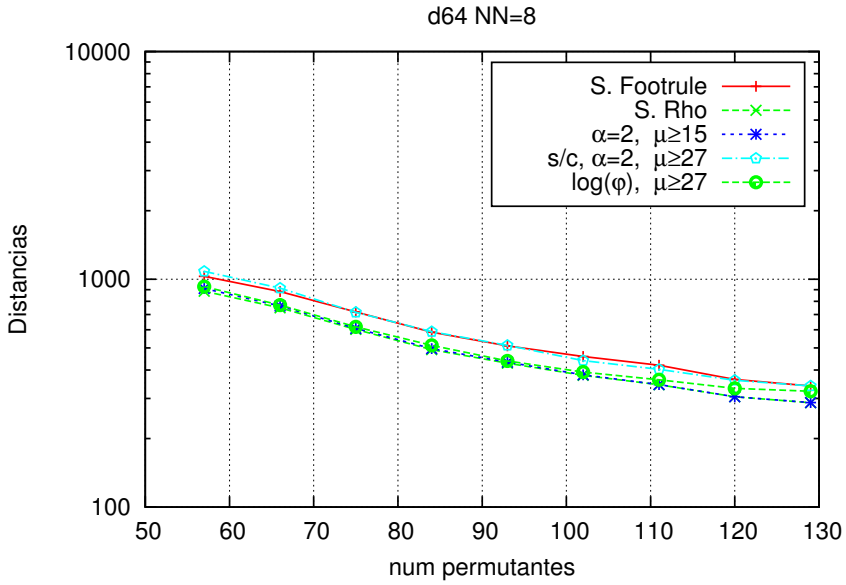


Figura 5.13: Número de cálculos de distancias para resolver 8NN en una Base de datos sintéticas en una dimension de 64.

5.2.1. Histogramas de Colores

Esta base de datos consta de 112,682 histogramas de color, representados como 112 dimensiones vectores de características. Este conjunto de datos se obtuvo del espacio métrico del proyecto SISAP [25]. Elegimos 500 histogramas aleatoriamente como conjunto de prueba, para consultas, y el resto como nuestra base de datos para indexar. En estos experimentos mostraremos que la métrica Footrule y Spearman Rho se pueden mejorar.

Variando los Vecinos mas Cercanos

Como podemos apreciar en las Figuras 5.14, 5.15, 5.16 y 5.17 se puede mejorar las métricas existentes con la propuesta de $\log_{10}(\phi)$ y con $\mu \geq 15$. En estas gráficas se usaron 66 y 75 permutantes.

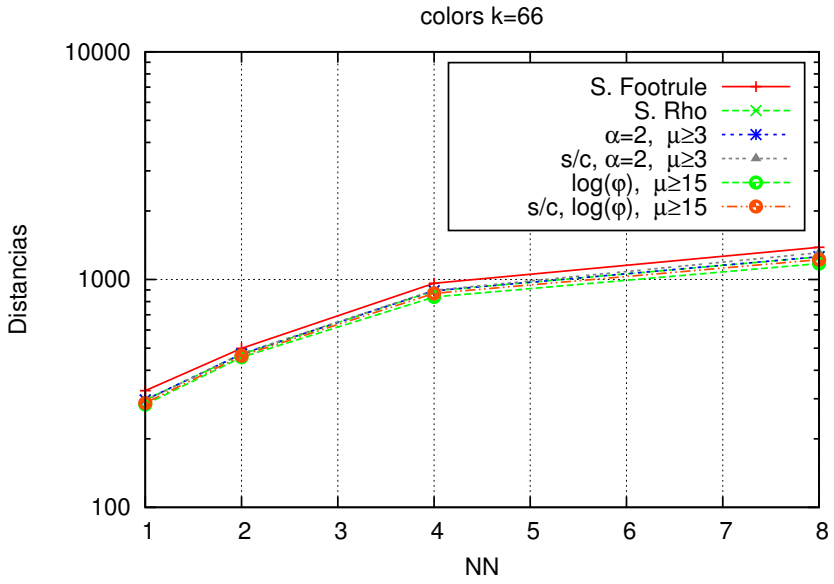


Figura 5.14: La base de datos de Colores calculando la cantidad de distancias variando la cantidad de vecinos a encontrar, con 66 permutantes.

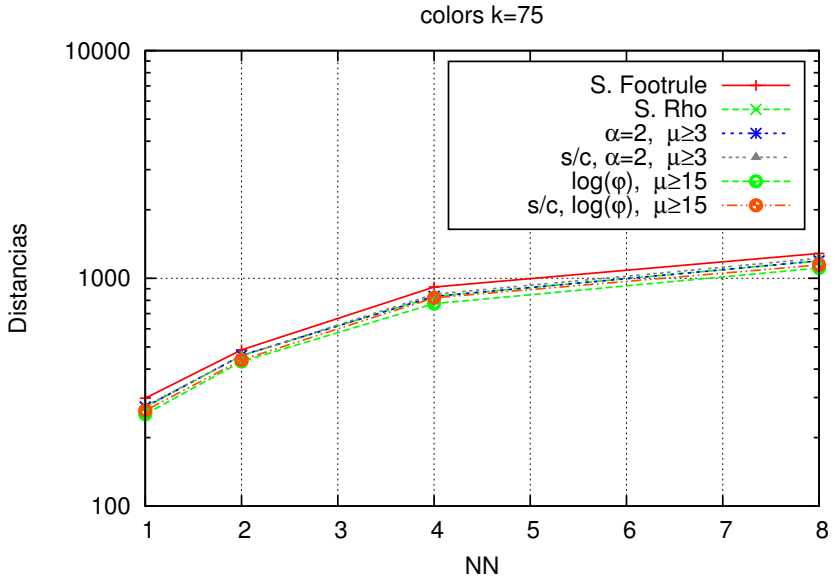


Figura 5.15: La base de datos de Colores calculando la cantidad de distancias variando la cantidad de vecinos a encontrar, con 75 permutantes.

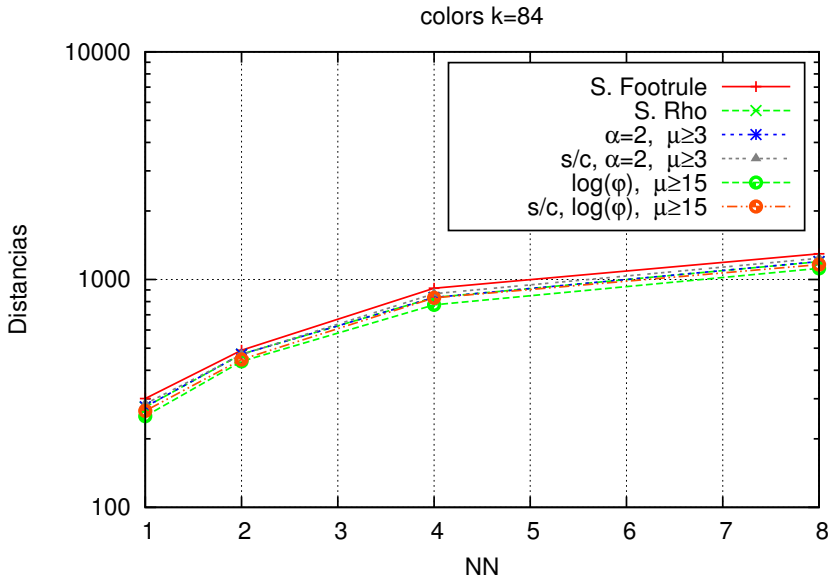


Figura 5.16: La base de datos de Colores calculando la cantidad de distancias variando la cantidad de vecinos a encontrar, con 84 permutantes.

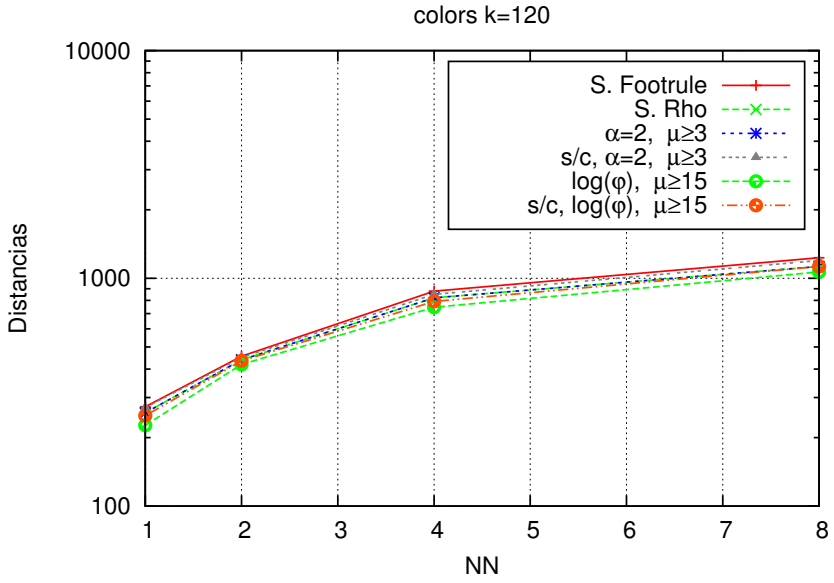


Figura 5.17: La base de datos de Colores calculando la cantidad de distancias variando la cantidad de vecinos a encontrar, con 120 permutantes.

Variando el Número de Permutantes

Como podemos apreciar en las Figuras 5.18, 5.19, 5.20 y 5.21 se pueden mejorar las métricas existentes con la propuesta de $\log_{10}(\phi)$ y con $\mu \geq 15$. En estas gráficas se buscaron 1, 2, 4 y 8 vecinos mas cercanos.

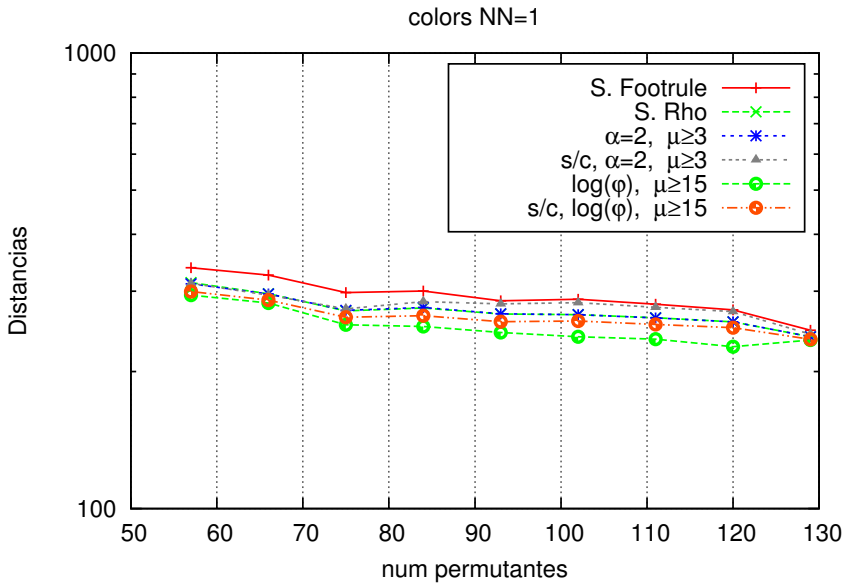


Figura 5.18: La base de datos de COLORs calculando la cantidad de distancias variando la cantidad de permutantes a usar y buscando 1NN.

5.3. NASA

Este conjunto de datos consta de 40.150 vectores de características en R 20. Estos vectores de 20 dimensiones Los generadores se generaron a partir de imágenes descargadas de la NASA [1], se duplicaron los vectores eliminado Usamos la distancia euclidiana para comparar los vectores de características de esta colección de imágenes.

[1] disponibles en (http://-www.dimacs.rutgers.edu/Challenges/Sixth/software.html

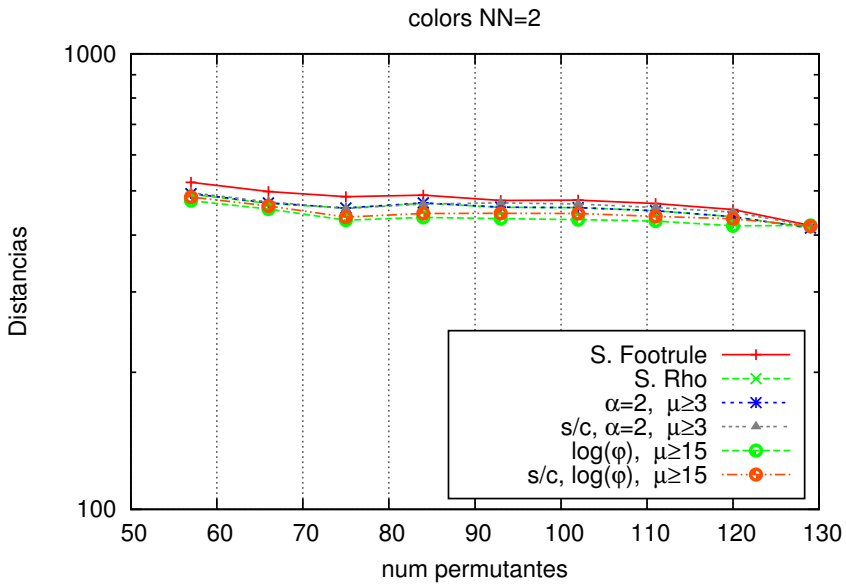


Figura 5.19: La base de datos de COLORs calculando la cantidad de distancias variando la cantidad de permutantes a usar y buscando 2NN

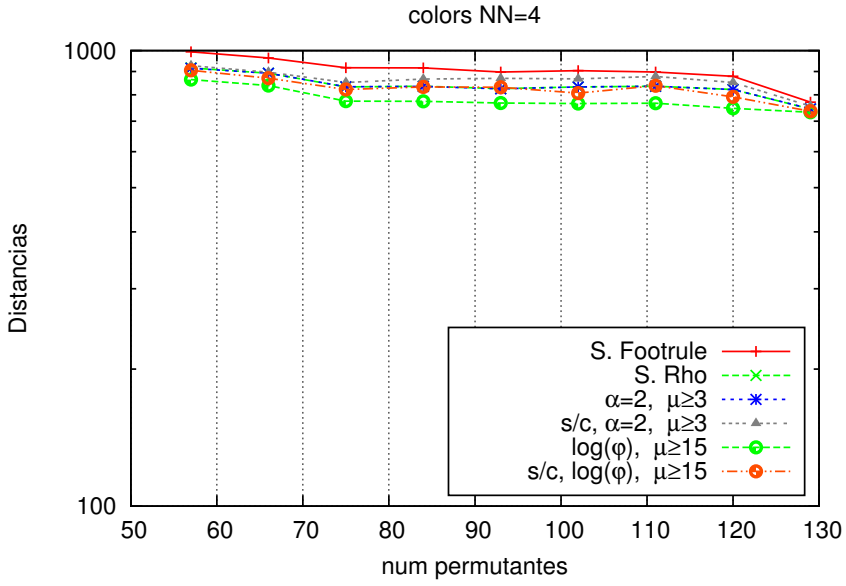


Figura 5.20: La base de datos de COLORs calculando la cantidad de distancias variando la cantidad de permutantes a usar y buscando 4NN

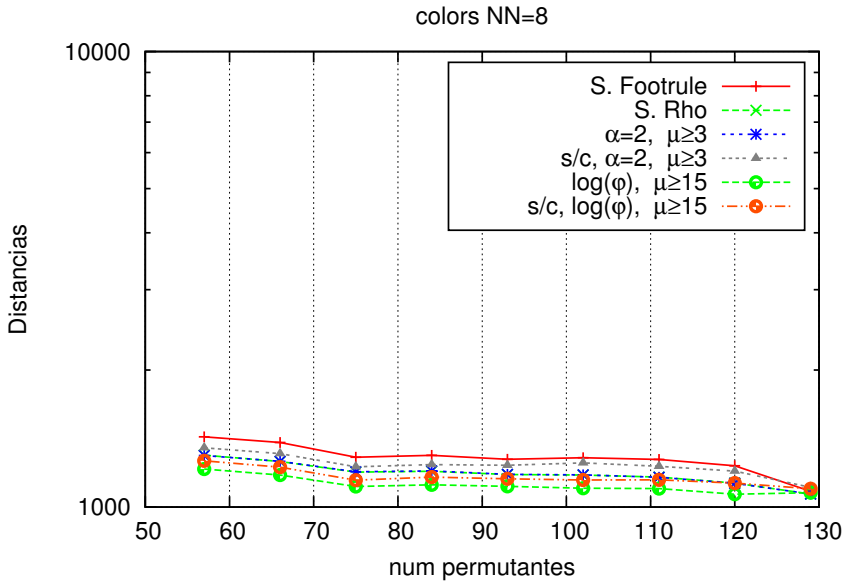


Figura 5.21: La base de datos de COLORs calculando la cantidad de distancias variando la cantidad de permutantes a usar y buscando 8NN.

Variando los Vecinos Mas Cercanos

En las Figuras 5.22, 5.23 y 5.24 se muestra el desempeño de la propuesta al variar la cantidad de vecinos mas cercanos.

Variando el Número de Permutantes

En las Figuras 5.26, 5.27 y 5.28 se muestra el desempeño variando la cantidad de permutantes y dejamos los vecinos fijos. Note que las propuestas reducen la cantidad de distancias necesarias para resolver el problema.

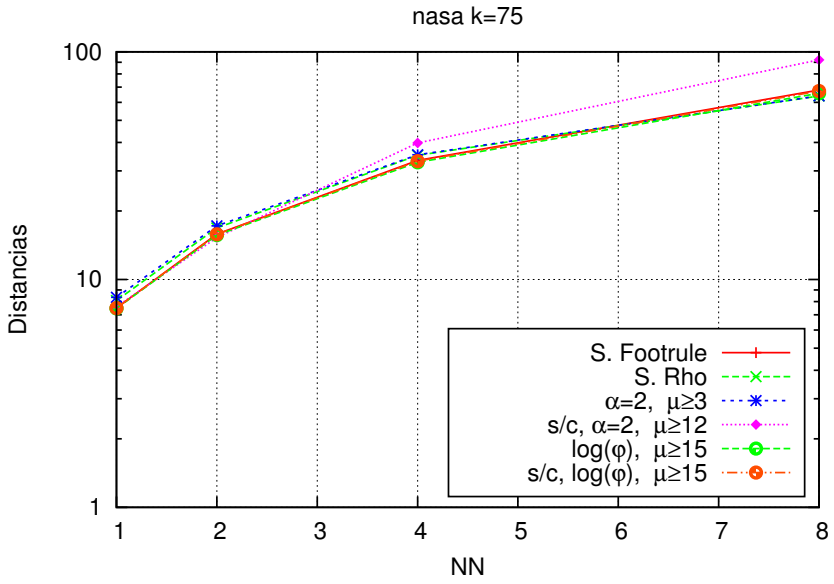


Figura 5.22: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de vecinos mas cercanos con 75 permutantes.

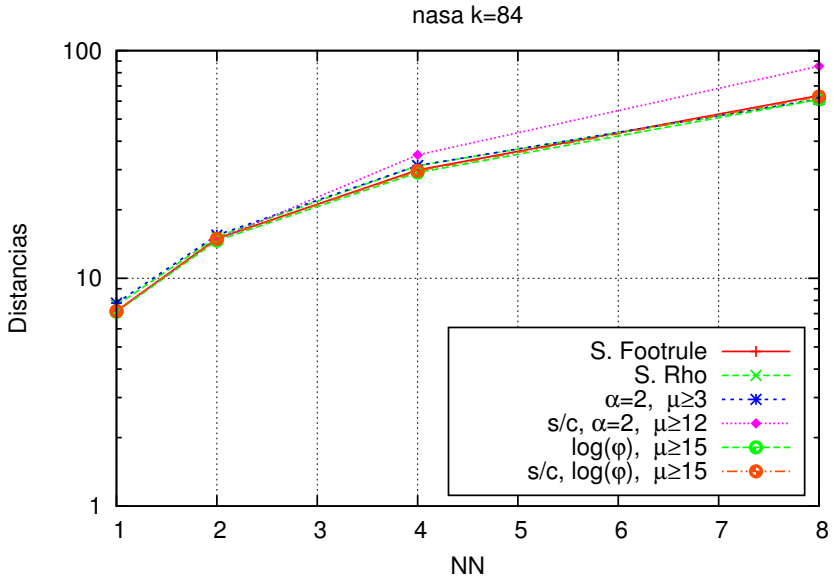


Figura 5.23: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de vecinos mas cercanos con 84 permutantes.

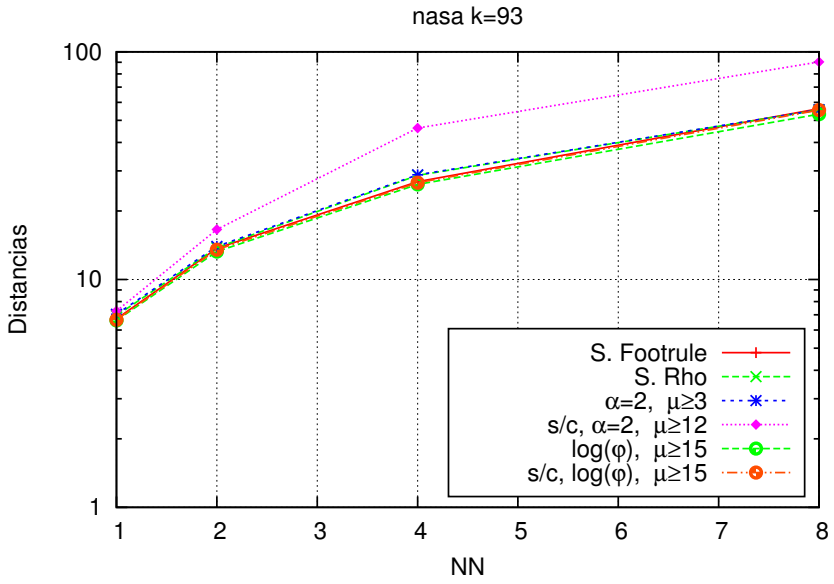


Figura 5.24: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de vecinos mas cercanos con 93 permutantes.

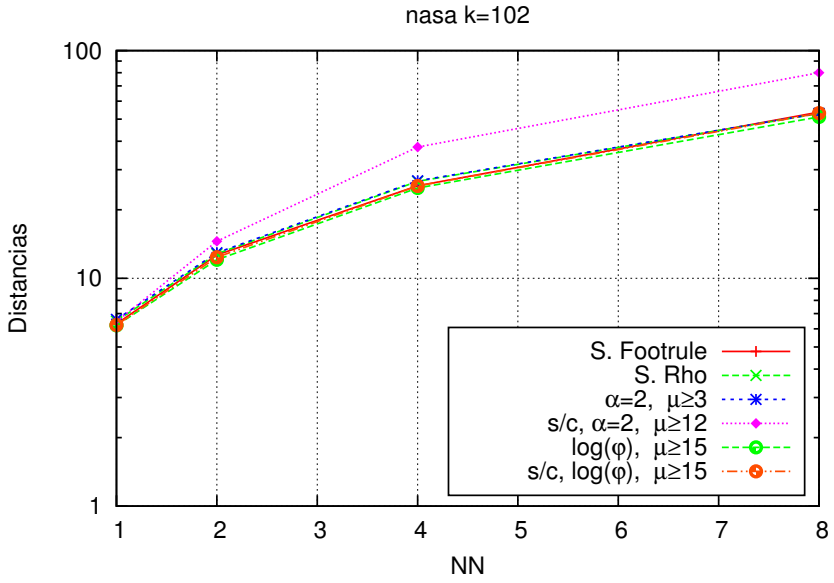


Figura 5.25: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de vecinos mas cercanos con 102 permutantes.

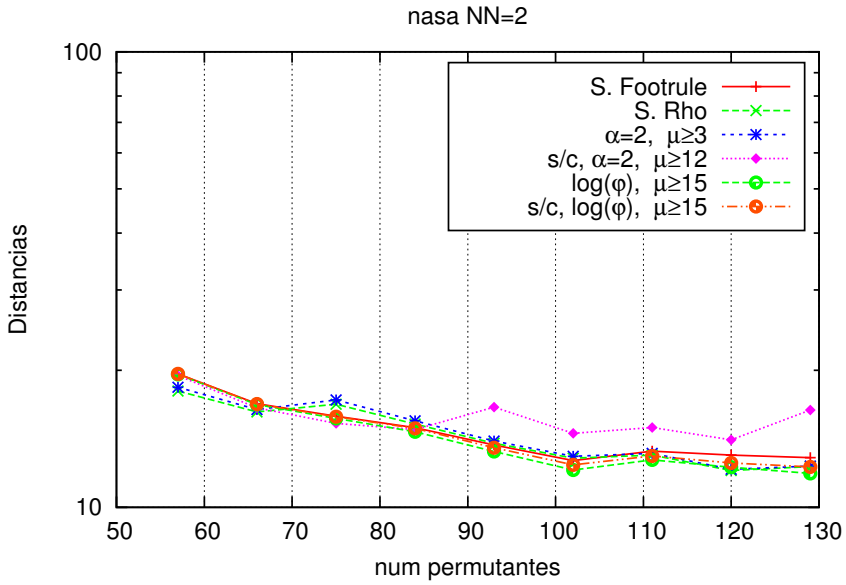


Figura 5.26: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de permutantes con 2 vecinos mas cercanos.

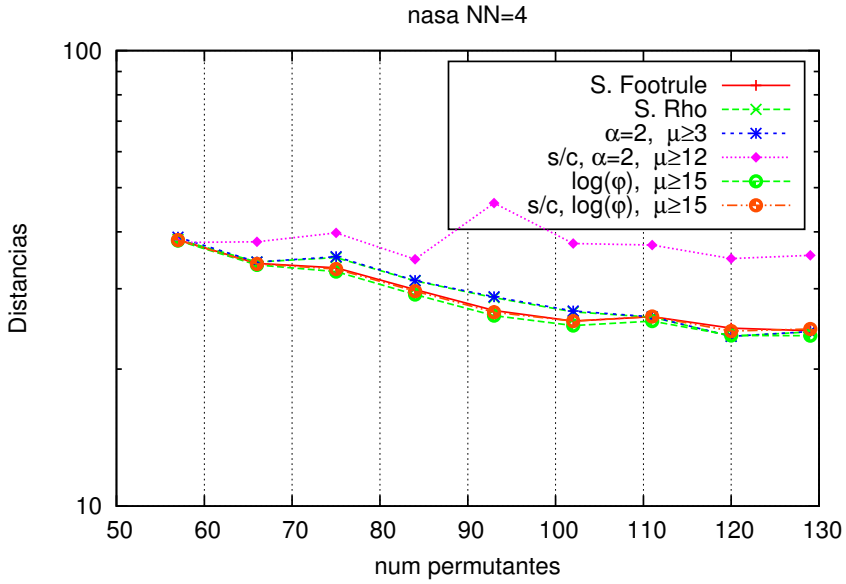


Figura 5.27: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de permutantes con 4 vecinos mas cercanos.

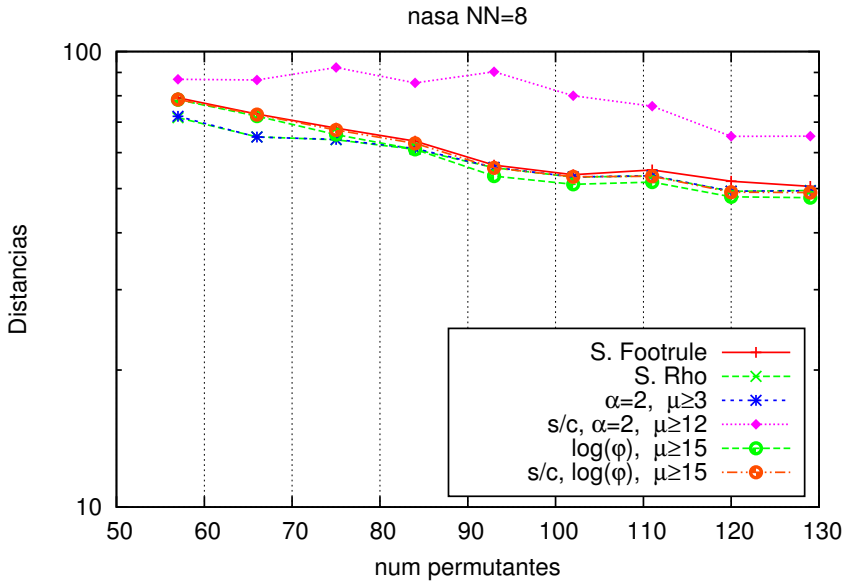


Figura 5.28: La base de datos de NASA calculando la cantidad de distancias variando la cantidad de permutantes con 8 vecinos mas cercanos.

Capítulo 6

Conclusiones y Trabajos a Futuro

6.1. Conclusiones

La búsqueda es una operación indisipensable en cualquier ámbito. En especial, en las ciencias de la computación, las búsquedas son el núcleo de distintas áreas como: inteligencia artificial, biología computacional, redes adhoc, etc. Las búsquedas por similaridad o proximidad consisten en recuperar de una base de datos aquellos elementos mas parecidos a uno de consulta.

En este trabajo se presentó una revisión del estado del arte de los algoritmos de búsqueda por similaridad en espacios métricos. En particular se detalló una técnica reciente conocida como algoritmos basados en permutaciones [12].

Los algoritmos basados en permutaciones consisten tomar un subconjunto de elementos, cada elemento restante, mide su distancia a dicho subconjunto y lo ordena de manera ascendente. Este ordenamiento será llamado permutación. Cabe mencionar que hasta hoy en día solo existía una propuesta para medir la diferencia entre permutaciones [12] esto es: Spearman Footrule o Spearman Rho. Esto significa que cualquier algoritmo en el estado del arte de este tema usaba una de estas medidas.

La propuesta de este trabajo consistió en modificar la medida de similitud entre permutaciones. En la tesis se presentaron 4 propuestas basadas en penalizar qué tanto se desplazaba un elemento en ambas permutaciones. Las 4 propuestas son:

- Elevar la diferencia al cuadrado, si es que la diferencia era mayor a un umbral establecido.

- Elevar la diferencia al logaritmo base 10 de esa diferencia, siempre que la diferencia fuera mayor a un umbral dado.
- Elevar la diferencia al cuadrado siempre que el elemento no se encontrara en el centro de la permutación.
- Elevar la diferencia al logaritmo base 10 de esa diferencia siempre que el elemento no se encontrara en el centro de la permutación.

Posteriormente se analizó su desempeño de manera experimental. Las bases de datos empleadas en la experimentación fueron: un grupo de bases de datos generadas sintéticamente en el cubo unitario de manera uniforme; el resto de las bases de datos fueron obtenidas con datos reales: Colors (histogramas de imágenes) y NASA (imágenes de la NASA).

El desempeño de nuestra propuesta fue competitivo en bases de datos reales, en algunas se reduce en un 30 % el valor de las comparaciones necesarias para resolver la consulta. Las bases de datos sintéticas muestran muy poca o ninguna mejoría.

6.2. Trabajo a Futuro

Las medidas presentadas en esta propuesta podrían mejorar todos los algoritmos existentes que hayan sido basados en permutaciones. Como trabajo a futuro se tiene:

- Implementar estas medidas de disimilaridad en propuestas de emplear grupos de permutantes [33].
- Considerar usar estas medidas en índices como iAESA [23].
- Implementar y experimentar si el índice invertido empleado en [1] puede ser mejorado con las medidas de esta propuesta.

Bibliografía

[1] Giuseppe Amato and Pasquale Savino. Approximate similarity search in metric spaces using inverted files. In *3rd Intl. ICST Conf. on Scalable Information Systems, INFOSCALE 2008, Vico Equense, Italy, June 4-6, 2008*, page 28. ICST / ACM, 2008.

[2] R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

[3] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proceedings 5th Combinatorial Pattern Matching (CPM'94)*, volume 807 of *Lecture Notes in Computer Science*, pages 198–212, 1994.

[4] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proceedings 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE CS Press, 1998.

[5] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[6] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 357–368, 1997. Sigmod Record 26(2).

[7] S. Brin. Near neighbor search in large metric spaces. In *Proceedings 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[8] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

[9] B. Bustos and G. Navarro. Probabilistic proximity searching algorithms based on compact partitions. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, volume 2476 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2002.

[10] E. Chávez and K. Figueroa. Faster proximity searching in metric data. In *Proceedings of the Mexican International Conference in Artificial Intelligence (MICAI)*, LNAI, pages 222–231. Springer, 2004.

[11] E. Chávez, K. Figueroa, and G. Navarro. Proximity searching in high dimensional spaces with a proximity preserving order. In *MICAI 2005: Advances in Artificial Intelligence*, volume 3789 of *Lecture Notes in Computer Science*, pages 405–414, 2005.

[12] E. Chávez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI)*, 30(9):1647–1658, 2009.

[13] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[14] E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In IEEE CS Press, editor, *7th International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, pages 75–86, 2000.

[15] E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01)*, volume LNCS 2153, pages 147–160, 2001.

[16] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. Technical Report TR/DCC-99-3, Dept. of Computer Science, Univ. of Chile, 1999. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/survmetric.ps.gz`.

[17] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[18] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[19] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[20] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[21] Andrea Esuli. Use of permutation prefixes for efficient and scalable approximate similarity search. *Inf. Process. Manage.*, 48(5):889–902, September 2012.

[22] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[23] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. On the lower cost for proximity searching in metric spaces. *5th International Workshop on Experimental Algorithms*, 4007:270–290, May 2006.

[24] K. Figueroa and R. Paredes. An effective permutant selection heuristic for proximity searching in metric spaces. In *Proc. 6th Mexican Conf. on Pattern Recognition (MCPR'14)*, LNCS 8495, pages 102–111. Springer, 2014.

[25] Karina Figueroa, Gonzalo Navarro, and Edgar Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

[26] Karina Figueroa, Rodrigo Paredes, Antonio Camarena-Ibarrola, and Héctor Tejeda Villela. Improving the permutation-based proximity searching algorithm using zones and partial information. *Pattern Recognition Letters*, 95:29–36, 2017.

[27] Karina Figueroa, Rodrigo Paredes, José Antonio Camarena Ibarrola, and Nora Reyes. Fixed height queries tree permutation index for proximity searching. In *Pattern Recognition - 9th Mexican Conference, MCPR 2017, Huatulco, Mexico, June 21-24, 2017, Proceedings*, pages 74–83, 2017.

[28] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[29] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *Transactions on Software Engineering*, 9(5), 1983.

[30] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics - Doklady*, volume 10, pages 707–710, 1966.

[31] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996.

[32] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[33] Karina Figueroa Mora, Rodrigo Paredes, and Roberto Rangel. Efficient group of permutants for proximity searching. In *Pattern Recognition - Third Mexican Conference, MCPR 2011, Cancun, Mexico, June 29 - July 2, 2011. Proceedings*, pages 42–49, 2011.

[34] F. Moreno-Seco, L. Micó, and J. Oncina. Extending LAESA fast nearest neighbour algorithm to find the *k*-nearest neighbours. In *Lecture Notes in Artificial Intelligence*, volume 2396 of *Lecture Notes in Computer Science*, pages 691–699. Springer, 2002.

[35] F. Moreno-Seco, L. Micó, and J. Oncina. A modification of the LAESA algorithm for approximated *k*-nn clasification. *Pattern Recognition Letters*, 24:47–53, 2003.

[36] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[37] G. Navarro. Searching in metric spaces by spatial approximation. In *String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

[38] G. Navarro, R. Paredes, and E. Chávez. *t*-spanners as a data structure for metric space searching. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, volume 2476 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2002.

[39] R. Paredes. Uso de *t*-spanners para búsqueda en espacios métricos. Master's thesis, Departamento de Ciencias de la Computacón Universidad de Chile, Santiago de Chile., 2002.

[40] K. Tokoro, K. Yamaguchi, and S. Masuda. Improvements of tlaesa nearest neighbour search algorithm and extension to approximation search. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 77–83, 2006.

[41] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[42] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[43] E. Vidal and M. J. Lloret. Fast speaker-independent dtw recognition of isolated words using a metric-space algorithm (aesa). *Speech Commun.*, 7(4):417–422, 1988.

[44] E. Vidal, H. Rulot, F. Casacuberta, and J.M. Benedi. On the use of a metric-space search algorithm (AESA) for fast DTW-based recognition of isolated words. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(5), 1988.

[45] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, 1996.

[46] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, 1999.

[47] C. Yu, B. Chin Ooi, K.L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, 2001*, pages 421–430, 2001.