Universidad Michoacana de San Nicolás de Hidalgo

FACULTAD DE INGENIERÍA ELÉCTRICA

“GRUPOS EFICACES DE PERMUTANTES PARA BUSQUEDAS POR SIMILITUD”

TESIS

QUE PARA OBTENER EL GRADO DE
INGENIERO EN COMPUTACIÓN

S U S T E N T A

ROBERTO RANGEL HERAS

ASESORA DE TESIS

DRA. KARINA MARIELA FIGUEROA MORA

MORELIA, MICHOACÁN DE OCAMPO

OCTUBRE 2011

RESUMEN

Las bases de datos de la actualidad tienen diversos formatos, muy distintos a los que se contenían en sus inicios. En un principio en una base de datos se almacenaba información estructurada (i.e. por números o palabras), ahora tenemos bases de datos multimediales (BDM) compuestas por imágenes, música, videos, etc. En estas bases de datos la búsqueda por igualdad ya no tiene sentido, pues difícilmente 2 objetos serán iguales.

Una propuesta para las búsquedas en bases de datos de este tipo, es modelando el problema como un espacio métrico. Un espacio métrico consiste en una base de datos y una función de distancia entre los elementos que generalmente es costosa de calcular, por lo que el objetivo será disminuir dicho costo.

La propuesta aquí planteada pretende mejorar uno de los algoritmos recientes (algoritmos basados en permutaciones) en espacios métricos. La contribución principal es haber logrado reducir el número de comparaciones respecto a la técnica original (de hasta 35 %) y tener índices más compactos sin usar mas memoria para la construcción del índice (hasta un 50 %).

Agradecimientos

Al igual que en una multiplicación, el orden de los factores no altera el producto (excepto en matrices), el orden en el que mencione a las personas que influyeron en mi vida, no quiere decir que sean mas o menos importantes para mi, toda la gente que conocí siempre será importante para mi por igual.

A mis padres, por el apoyo que me dieron y por haberme educado a pesar de todo, muchas gracias a mi mamá Ofelia Heras del Rio, por soportarme tantos años y seguir haciendolo. A mi padre, Eduardo Rangel Vieyra, que espero que le vaya muy bien en el camino que eligió.

Para mi hermano Juan Antonio Rangel Heras (Tonopower) del cual me siento muy orgulloso y aunque no se lo he dicho, espero hacerlo pronto, al menos antes de que lea los agradecimientos.

A mi asesora la Dra. Karina Mariela Figueroa Mora, por todo el apoyo que me brindo en todos los sentidos, siempre estaré agradecido. Muchas gracias por haberme recibido como su tesista.

Para Rodrigo Paredes por su apoyo aportado a través de mi asesora, y por la colaboración para obtener el artículo derivado de la tesis.

Mis maestros que me brindaron conocimientos a lo largo de la carrera y estuvieron ahí para aclarme mis dudas y mostrarme el camino correcto. Gracias por su sabiduria.

Para mi mejor amiga Claudia Paz Servin, mis amigos y amigas, Beatriz Flores Vega, Paulina Rodríguez, Sonia Tonatzi Morales, Miguel Bravo Orozco (“El miko“), Leonardo Ramírez Ortiz, Josué Ulises Suarez Soria, Geovanni Nambo Herrejon (”El canas”), Humberto Rene Farías Rojas (“El O.g.t.“) y muchos más que no incluyo no porque no sean importantes, sino porque necesitaría varias hojas para mencionarlos. Mi historia la escriben los que tengo cerca, gracias amigos por su apoyo incondicional.

Gracias por todo.

Publicaciones

El artículo obtenido apartir de esta tesis se puede consultar en: *Efficient Group of Permutants for Proximity Searching*, en *Mexican Conference on Pattern Recognition*, 2011. LNCS 6718. Páginas 42–49. Autores: Karina Figueroa, Rodrigo Paredes y Roberto Rangel.

Índice general

Índice de figuras

Índice de cuadros

Lista de algoritmos

Capítulo 1

Introducción

Las bases de datos han ido evolucionando a lo largo del tiempo. Al inicio, una base de datos tradicional estaba compuesta por números y palabras, ahora tenemos bases de datos compuestas por todo tipo de datos, desde secuencias de ADN y patrones biológicos hasta videos. Para buscar dentro de una base de datos, se deben estructurar los datos o realizar un índice. Un índice en una base de datos es conceptualmente parecido al índice de un libro, es decir, en un libro se puede consultar el índice para saber en cual página buscar cierto tema, de la misma manera un índice en una base de datos nos ayuda a encontrar los datos más rápido. Para bases de datos compuestas por números o palabras, la construcción del índice es bastante clara, ya que para los números sólo basta ordenarlos de menor a mayor y en el caso de las palabras se pueden ordenar alfabéticamente. Sin embargo con bases de datos compuestas por otros tipos de datos, como música, videos, etcétera los problemas son de otro tipo pues los datos no tienen un orden como los alfabéticos por ejemplo, a estas bases de datos les llamaremos bases de datos multimedia (BDM).

1.1. Antecedentes

Las búsquedas dentro de una BDM difícilmente pueden ser búsquedas exáctas. Una alternativa son las búsquedas por similaridad las cuales consisten en recuperar objetos similares a la consulta donde dichos objetos serán elementos de una BDM.

Las propuestas existentes para tratar este tipo de información están, en su mayoría, basadas en extraer la información importante de los objetos, y representar esta información usando vectores de dimensión alta. Esto se conoce como extracción de características. Definimos la extracción de características δ como una transformación de un objeto (Obj) a un vector de dimensión m

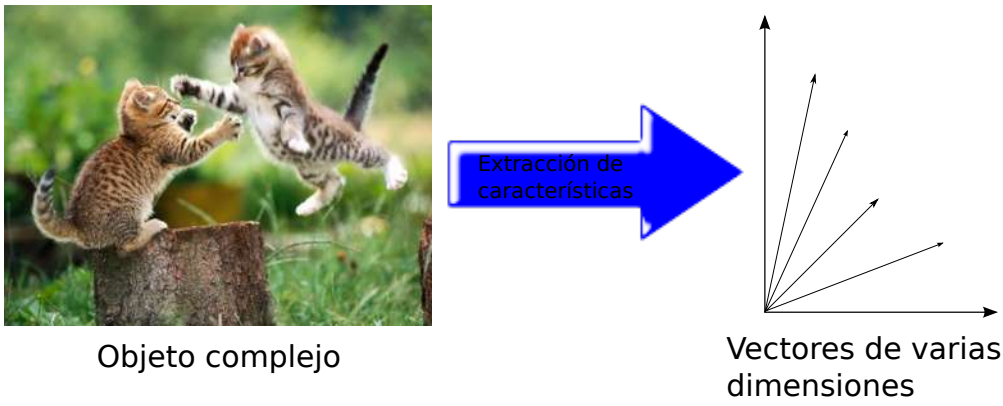

Figura 1.1: A un objeto complejo, en este caso a una imagen, se le extraen las características y se modela como vectores de varias dimensiones.

($\mathbb{R}^m$), esto es, $\delta : Obj \rightarrow \mathbb{R}^m$ (véase la Figura 1.1). Las bases de datos multidimensionales (BDMD) almacenan conjuntos de estos vectores.

En las BDMD, una vez que se tienen los vectores asociados a cada objeto, la búsqueda por similaridad consiste en encontrar los vectores con mayor proximidad entre sí.

Generalmente, en las BDMD se emplean índices que aprovechan las coordendas de cada vector (llamados índices para espacios multidimensionales o vectoriales). Estos índices presentan un buen rendimiento en dimensiones bajas (<20). A medida que la dimensión aumenta, su rendimiento se degrada rápidamente. Esto es conocido como la maldición de la dimensionalidad [CNBYM01]. En los espacios vectoriales existen algoritmos que aprovechan el hecho de que la similaridad se interprete geométricamente. En general, los algoritmos para espacios vectoriales se basan en la geometría del espacio y la información de las coordenadas.

Los métodos para espacios vectoriales son conocidos como *Spatial Access Methods* SAM [BBK01a]. Entre los más populares están KD-Tree [Ben75, Ben79], R-Tree [Gut84]y X-Tree [SBK96], por mencionar algunos [HS95, BBK$^+$01b]. En general, estas técnicas hacen uso exhaustivo de la información de las coordenadas para agrupar y clasificar los objetos en el espacio. Otra alternativa es modelar las bases de datos como espacios métricos (veáse el capítulo 2 sección 2.1 donde se define formalmente este concepto).

Para llevar a cabo las búsquedas por similaridad en BDM modeladas como espacios métricos, se llevan a cabo en 2 partes: preprocesamiento y consultas. En el preprocesamiento se construye un índice, éste permitirá agilizar encontrar la respuesta al momento de la consulta. Existen tres familias de este tipo de índices:

- Los algoritmos basados en pivotes,
- los algoritmos basados en particiones compactas y,

- los algoritmos basados en permutantes (ABPe);

de los cuales se hablará mas en el Capítulo 2 Sección 2.2.

1.2. Objetivo

El objetivo básico de esta tesis es mejorar la recuperación de datos en índices más compactos al momento de realizar las búsquedas por rango o búsquedas de los vecinos más cercanos en los algoritmos basados en permutaciones. La propuesta es formar grupos de permutantes, es decir, se tomarán varios permutantes para conformar un grupo y las permutaciones ahora se harán por grupo, con esto se pretende tener índices más compactos y tener al menos la misma recuperación que en un algoritmo de permutantes común.

1.3. Metodología

La metodología a seguir será análisis - experimental. Primero se estudiarán las propuestas para mejorar la idea de los algoritmos basados en permutaciones y después se desarrollará un programa en lenguaje C que implemente el algoritmo para sus pruebas en bases de datos de dos tipos: sintéticas y reales. Las bases de datos sintéticas se usarán para controlar la dimensión del espacio y conocer el funcionamiento de ciertos parámetros del algoritmo.

1.4. Contenido de la tesis

En el Capítulo 1 se da una breve introducción del presente trabajo, se explica que son las BDM y la necesidad de la creación de índices para las mismas. También se presenta a grandes rasgos la idea propuesta. Por último se plantea la metodología a seguir.

En el Capítulo 2 se presentan conceptos básicos sobre espacios métricos, búsquedas por proximidad, se dan a conocer las 3 familias en las que se organizan los algoritmos para espacios métricos.

En el Capítulo 3 se desarrolla la propuesta de los grupos de permutantes. Inicialmente se explican los problemas a resolver en el algoritmo planteado y después se mostrarán los aportes de esta tesis y los detalles técnicos de la implementación.

En el Capítulo 4 se presentan las pruebas y resultados en la fase experimental. En esta parte se presentan 2 tipos de bases de datos: sintéticas y reales. Las sintéticas permiten conocer el funcionamiento del algoritmo en un ambiente controlado; en las bases de datos reales, se muestra

el funcionamiento de la técnica en un espacio real.

En el Capítulo 5 se darán las conclusiones de la tesis y los trabajos futuros propuestos a partir de esta tesis.

Capítulo 2

Conceptos básicos

2.1. Espacios métricos

Un espacio métrico $(\mathbb{X}, d)$ está formado por un universo de objetos y una función de distancia, donde $\mathbb{X}$ es el universo de objetos válidos. $\mathbb{U} \subseteq \mathbb{X}$ es la base de datos, tal que $|\mathbb{U}| = n$. La función de distancia es definida como $d : \mathbb{X} \times \mathbb{X} \to [0, \infty)$ y denotará la medida de semejanza entre los elementos de $\mathbb{X}$. 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: $d(x, x) = 0$
- Desigualdad triangular: $d(x, z) \leq d(x, y) + d(y, z)$

2.2. Estado del Arte

Los índices para espacios métricos básicamente pueden clasificarse en 3 familias, las cuales serán descritas a continuación [CNBYM01].

2.2.1. Algoritmos basados en pivotes

En esta familia un índice se construye de la siguiente forma:

1. Elegir un conjunto $\mathbb{P} = \{p_1, p_2, \ldots, p_k\} \subseteq \mathbb{U}, k = |\mathbb{P}|$

2. Calcular y almacenar con alguna estructura (árboles usualmente) $\{d(p_1,u), d(p_2,u), \ldots, d(p_k,u)\}, \forall u \in \mathbb{U}$

Una vez construido un índice es posible realizar las búsquedas y ya que los algoritmos basados en pivotes y los algoritmos basados en particiones compactas realizan la búsqueda de manera similar, se explicará en la siguiente subsección.

2.2.2. Algoritmos basados en particiones compactas

Esta familia divide el espacio en zonas tan compactas como sea posible, seleccionando un conjunto de objetos (centros) $\mathbb{P} = \{p_1, \ldots, p_k\} \subseteq \mathbb{U}$ y distribuyendo los demás elementos entre las k zonas de acuerdo a su cercanía. El índice está compuesto por: los centros, los elementos que pertenecen a cada zona; y, en algunos casos, información adicional sobre algunas distancias (por ejemplo la distancia al centro de cada elemento tal que $u \in \mathbb{U}$ y $u \notin \mathbb{P}$).

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,

$$d(q,u) \leq d(q,p) + d(p,u). \tag{2.1}$$

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

$$d(p,u) \leq d(p,q) + d(q,u), \tag{2.2}$$

$$d(p,q) \leq d(p,u) + d(u,q), \tag{2.3}$$

Estas desigualdades implican, despejando $d(q,u)$ de 2.4 y $d(u,q)$ de 2.5 respectivamente, se obtiene:

$$d(p,u) - d(p,q) \leq d(q,u), \tag{2.4}$$

$$d(p,q) - d(p,u) \leq d(u,q), \tag{2.5}$$

combinando 2.4 y 2.5 y usando la simetría, obtenemos que

$$|d(p,u) - d(p,q)| \leq d(q,u). \blacksquare \tag{2.6}$$

De esta forma, es posible evitar cálculos de distancia para responder la consulta.

En ambas familias, dada una consulta q, de acuerdo al índice usado, q se compara contra los pivotes o los centros (según sea el caso). Con ayuda de la desigualdad triangular y el índice, es posible acotar inferiormente la distancia entre q y cualquier $u \in \mathbb{U}$, $|d(p_i,q) - d(p_i,u)| > r$, si esto se cumple se puede descartar u porque si $|d(p_i,q) - d(p_i,u)| > r$ entonces $d(p_i,u) > r$. Veáse Lema 1.

2.2.3. Algoritmos basados en permutaciones

La técnica de los algoritmos Basados en Permutantes (ABPe) es la siguiente: sea $\mathbb{P} \subseteq \mathbb{U}$ un conjunto de objetos, que llamaremos permutantes, $\mathbb{P} = \{p_1, p_2, \ldots, p_k\}$. Cada elemento del espacio, $u \in \mathbb{U}$, define una permutación Π_u, donde los elementos de $\mathbb{P}$ están escritos en orden ascendente de la distancia hacia u, es decir, definimos Π_u como una permutación de $(1 \ldots k)$ de manera que, para todos $1 \leq i < k$ contiene también $d(p_{\Pi_{u(i)}}, u) < d(p_{\Pi_{u(i+1)}}, u)$, o $d(p_{\Pi_{u(i)}}, u) = d(p_{\Pi_{u(i+1)}}, u)$ y $\Pi_u(i) < \Pi_u(i+1)$ [Fig07], donde $p_{\Pi_{u(i)}}$ es el permutante en la posición i-ésima, es decir el elemento en la base de datos y $\Pi_{u(i)}$ es sólo la posición en la que se encuentra el permutante en la base de datos.

Los empates se rompen con cualquier orden coherente, por ejemplo, el orden de los elementos de $\mathbb{P}$. Los elementos de $\mathbb{P}$ se seleccionaron de manera aleatoria, en [FP10] se explican otras formas de seleccionar los permutantes. Una permutación inversa Π_u^{-1} nos indica en que posición se encuentra cada permutante. Un ejemplo de permutación y permutación inversa se ilustra en el Cuadro 2.1 y en el Cuadro 2.2, donde Π_u y Π_u^{-1} se obtienen de la segunda fila de cada cuadro.

Cuadro 2.1: Permutación Π_u

Posición (i)	1	2	3	4	5	6
Permutante	6	2	3	1	4	5

Cuadro 2.2: Permutación inversa Π_u^{-1}

Permutante	1	2	3	4	5	6
Posición (i)	4	2	3	5	6	1

La permutación obtenida apartir del Cuadro 2.1 es $\Pi_u = \{6,2,3,1,4,5\}$ y la permutación inversa obtenida del Cuadro 2.2 es $\Pi_u^{-1} = \{4,2,3,5,6,1\}$.

Dos permutanciones Π_u y Π_q pueden ser comparadas usando Spearman Footrule. Esta se define como:

$$S_p(\Pi_u, \Pi_q) = \sum_{1 \leq i < k} (|\Pi_u^{-1}(i) - \Pi_q^{-1}(i)|) \tag{2.7}$$

Donde Π_u^{-1} y Π_q^{-1} son las permutaciones inversas del objeto u y q respectivamente.

Un ejemplo de $S_p(\Pi_u, \Pi_q)$ es con $\Pi_q = 6,2,3,1,4,5$ y su permutanción inversa $\Pi_q^{-1} = 4,2,3,5,6,1$, y $\Pi_u = 3,6,2,1,5,4$ y su permutación inversa $\Pi_u^{-1} = 4,3,1,6,5,2$. Un elemento particular p_3 en la permutación Π_u se encuentra a dos posiciones con respecto a la posición en Π_q. Utilizando Spearman Footrule tenemos: $4-4, 3-2, 3-1, 5-6, 6-5, 1-2$, y la suma es $S_p(\Pi_u, \Pi_q) = 5$. El resultado de $S_p(\Pi_u, \Pi_q)$ nos indica que tan similares son dos permutaciones, si $S_p(\Pi_u, \Pi_q) = 0$ quiere decir que las permutaciones son iguales. Entre más pequeño es el resultado entonces las permutaciones son más similares entre sí. Si el resultado es muy grande significa que las permutaciones son muy diferentes.

Hay otras medidas de similaridad entre permutaciones, sin embargo se mostró en [CFN09] que tienen un rendimiento similar pero en especial Spearman Footrule tiene un balance entre exactitud y tiempo de cálculo.

Descripción del problema de esta técnica

Los ABPe son algoritmos aproximados debido a que a pesar de que tienen un muy buen rendimiento en dimensiones altas, existen algunos falsos negativos que de encontrarlos rápidamente se mejoraría la técnica. En esta sección se explicará una posible causa de este problema.

Los ABPe tienen una frontera entre los elementos cercanos a un permutante y a otro, observe la Figura 2.1 a), la línea (frontera) delimita la mitad entre 2 permutantes (cuadros azules). A pesar de que q y u_1 son cercanos, tienen una permutación invertida en esos dos permutantes, idealmente las permutaciones deberían ser iguales porque se encuentran muy cercanos entre sí, esto convierte al elemento u_1 en un falso negativo, ya que no es reportado como relevante entre los primeros elementos comparados. En la Figura 2.1 b) se puede observar que las permutaciones son iguales, ya que los elementos se encuentran encima de la frontera y no se ven afectados.

La propuesta de esta tesis esta basada en una heurística para evitar el problema de frontera expuesto en esta sección.

2.3. Búsquedas por proximidad

Básicamente existen dos tipos de consultas, por rango y K vecinos más cercanos (KNN por sus siglas en inglés). La primera consiste en recuperar los objetos de una consulta dada q dentro de un radio r, esto es $R(q,r) = \{d(u,q) \leq r \,|\forall u \in \mathbb{U}\}$, el segundo es recuperar los K elementos de $\mathbb{U}$ más cercanos a la consulta q, esto es $A \subseteq \mathbb{U}$ tal que $\forall u \in A$, $\forall v \in \mathbb{U}$, $d(q,u) \leq d(q,v), K = |A|$.

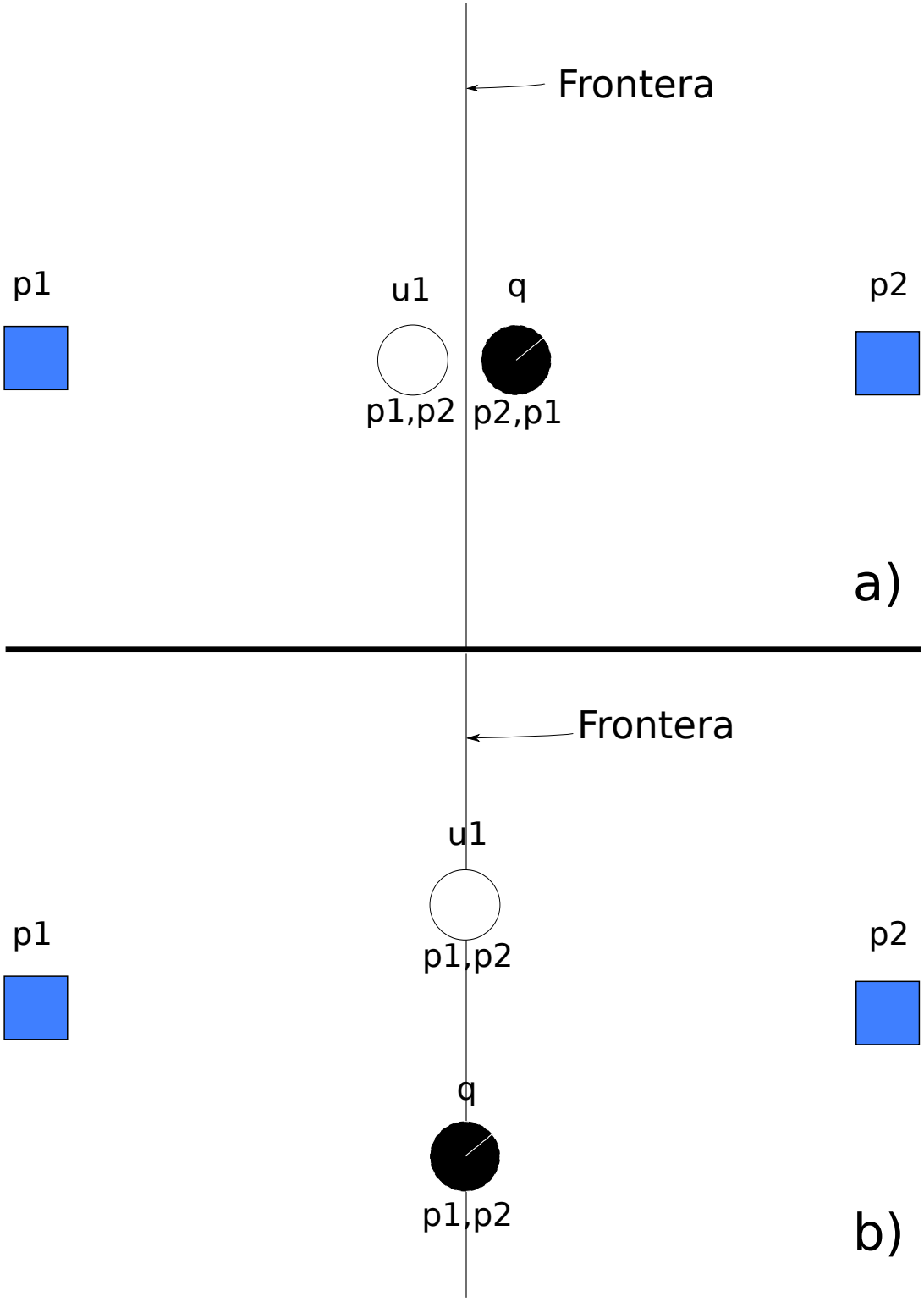

Figura 2.1: En a) se ilustra gráficamente como afecta la frontera a q y en b) se muestra que la frontera no afecta en ciertas zonas.

La mayoría de los enfoques existentes para resolver el problema de la búsqueda son algoritmos que recuperan exactamente los elementos de $\mathbb{U}$ ante una consulta. En [CFN05] se muestra que estos enfoques trabajan bien en dimensiones bajas (2 a 8), sin embargo, en dimensiones altas comparan la base de datos completa. Una alternativa en dimensiones altas son los algoritmos basados en permutantes.

En este capítulo se describió el estado del arte y las 3 familias de índices que utilizan espacios métricos, así como el problema que tienen los ABPe, además se describieron 2 tipos de búsqueda por proximidad, estos serán utilizados en el Capítulo 3.

Capítulo 3

Grupos de permutantes

En este capítulo se explicará la propuesta de esta tesis que consiste, básicamente, en usar un *grupo de permutantes* en lugar de un sólo permutante por cada componente en la permutación. De esta manera, se usará la misma cantidad de espacio que en el ABPe pero se espera tener más precisión. Además, la frontera que puede inestabilizar al ABPe podría estabilizarse. Las funciones se programarán en lenguaje C utilizando como apoyo librerías existentes de espacios métricos, estas se encuentran disponibles en International Workshop on Similarity Search and Applications (SISAP)[1], además hay un repositorio de bases de datos reales (Nasa, Colors) y permite crear bases de datos sintéticas (DBS) que nos servirán para realizar las pruebas necesarias.

3.1. Grupo de permutantes

La propuesta consiste en, seleccionar un grupo de permutantes $\mathbb{G} = \{G_1, G_2, \ldots, G_k\}$ donde $\mathbb{G} \subseteq \mathbb{U}$ y $G_i \cap G_j = \emptyset$, $\forall i, j, 1 \leq i \leq k$, $1 \leq j \leq k$, $i \neq j$ y $|\mathbb{G}| = k$, es decir, k es el número de grupos de permutantes. Cada grupo esta compuesto por m permutantes. Entonces, $\forall u \in \mathbb{U}$ y $u \notin \mathbb{G}$, comparamos u contra los grupos, esto es, $D_i(u, G_i)$, $1 \leq i \leq k$ y ordenamos D_i por proximidad a u. Las permutaciones ahora se harán respecto a los grupos y no a un sólo permutante como se usa en los ABPe comúnes.

3.2. Conformación de grupos de permutantes

Para conformar un grupo de permutantes, al igual que en un ABPe, se toman elementos aleatorios de $\mathbb{U}$ y se utilizan como permutantes. En este caso son $k * m$ ya que cada grupo de permutantes está conformado por m permutantes y tenemos k grupos. Por lo tanto, nuestros per-

[1] `http://sisap.org`

mutantes serían $\mathbb{G} \subseteq \mathbb{U}$ y $\mathbb{G} = \{u_1, \ldots, u_{k*m}\}$, y como $\mathbb{G} = \{G_1, \ldots, G_k\}$ entonces, $G_1 = \{u_1, \ldots, u_m\}$, $G_2 = \{u_{m+1}, \ldots, u_{2*m}\}, \ldots, G_k = \{u_{(k-1)*m}, \ldots, u_{k*m}\}$. Así estan conformados los grupos de permutantes para determinar, en primer lugar, la mejor forma de calcular la proximidad hacia un grupo de permutantes D_i. Mas adelante, en la Sección 3.5 se verán otras maneras de conformar los grupos.

3.3. Proximidad a un grupo

Una vez que tenemos conformados nuestros grupos de permutantes, un factor importante del rendimiento de la técnica es cómo decidir la proximidad hacia los grupos. La propuesta es tratar cada grupo como un cluster y usar los métodos para asociar cada elemento a un grupo.

- Enlace individual: Esto es, se considera la menor distancia hacia todos los objetos en el grupo, $D_{i_{min}}(u, G_i) = min_{\forall_{p \in G_i}} d(p, u)$.
- Enlace completo: En este caso se considera la distancia mayor hacia todos los objetos en el grupo, $D_{i_{max}}(u, G_i) = max_{\forall_{p \in G_i}} d(p, u)$.
- Promedio: La tercer propuesta es el promedio de las distancias, esto es, $D_{i_{avg}}(u, G_i) = \frac{\sum_{\forall_{p \in G_i}} d(p,u)}{|G_i|}$.

Por simplicidad, en lo siguiente solo se usará *Dmin*, *Dmax* y *Davg*. La Figura 3.1 describe dos de los criterios por proximidad hacia los grupos. El punto azul es un elemento de la base de datos y los permutantes son los puntos negros y los grupos que están organizados de la siguiente manera $G_1 = \{p_1, p_2, p_3\}$, $G_2 = \{p_4, p_5, p_6\}$ y $G_3 = \{p_7, p_8, p_9\}$. La Figura 3.1 a) muestra la permutación del elemento utilizando *Dmin* como criterio, en la Figura 3.1 b) se puede observar que la permutación utilizando *Dmax* es la misma que usando *Dmin*, pero en la Figura 3.1 c) las permutaciones usando *Dmin* y *Dmax* las permutaciones cambian, esto nos indica que la permutación puede cambiar dependiendo del criterio usado. Por ultimo, el criterio *Davg* no se muestra en la figura, ya que depende del promedio de las distancias y sería complicado explicarlo en una imagen, por el contrario, sólo confundiría la idea.

3.4. Detalles de implementación

En esta sección se mostrarán los detalles más relevantes de la implementación del programa de esta tesis, el código fuente se muestra en los apéndices.

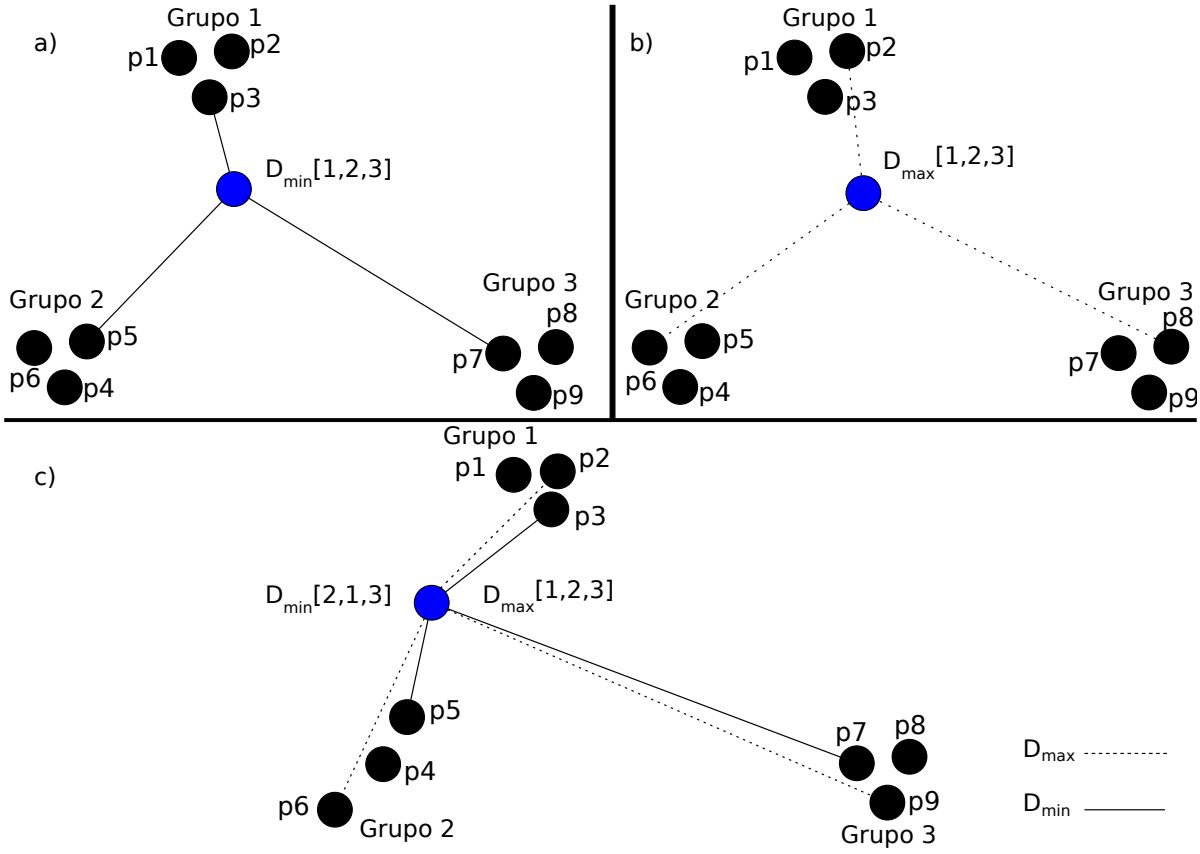

Figura 3.1: La línea continua es para la distancia mínima (Dmin) hacia el grupo y la línea discontinua es para la distancia máxima (Dmax). a) Muestra *Dmin*, b) Muestra *Dmax* y c) Muestra *Dmin* y *Dmax*

3.4.1. Construcción del índice

La primera parte de la implementación consistió en crear el índice para resolver las consultas de similaridad. Para esto se usó el pseudocódigo del Algoritmo 1. Este algoritmo recibe como parámetros la base de datos $\mathbb{U}$, el número de permutantes por grupo m y el número de grupos k y la función regresará el índice I. El primer ciclo asigna los permutantes que conformarán los grupos, se puede observar que usa $k * m$ elementos para conformarlos. En la línea 6 de se utiliza $CalcularD$ que calcula la distancia de un elemento u hacía los grupos y cambiará según el criterio elegido. Dis es una estructura que contiene tuplas con las distancias hacia cada grupo $D_i(u, G_i)$ e id, id es el identificador de cada grupo, estas distancias se ordenan de menor a mayor con el propósito de formar la permutación de cada elemento (el método de ordenamiento puede ser $qsort$ dado su buen desempeño) y en caso de que las distancias sean iguales, se toma en cuenta el id del grupo de permutantes para romper el empate. Para todos los ordenamientos que se llevan a cabo a lo largo del algoritmo se usó la implementación de Quick sort (qsort) de la librería estándar de lenguaje C cuya complejidad promedio es $O(n \log n)$. Estas permutaciones se almacenan en un índice I además de otros parámetros importantes como $\mathbb{G}$, m y k.

Algoritmo 1 ConstruirElIndice

Entrada: $\mathbb{U}$,m,k
Salida: I

```
1:  Sea Dis un arreglo de tuplas con id y d
2:  for i = 1 → k * m do
3:      G[i] ← Random(i)
4:  end for
5:  for all u ∈ U y u ∉ G do
6:      Dis ← CalcularD(G, u, m, k)
7:      Dis ← qsort(Dis)
8:      for i = 1 → k do
9:          Π_u[i] ← Dis[i].id
10:     end for
11: end for
12: return I
```

3.4.2. Consultas

Una vez construido el índice se procede a hacer las consultas, ya sean por radio o por KNN. En esta fase se utilizarán el Algoritmo 2 y el Algoritmo 3 para la búsqueda por radio y de KNN respectivamente. Note que estas funciones dependen de la creación del índice y deberá usarse el mismo criterio de cercanía. Para ambas funciones se calcula Π_q^{-1}, es decir, la permutación inversa de q, esta nos será útil para evaluar Spearman Footrule.

El Algoritmo 2 regresa el total de elementos dentro del radio r. Primero se calcula la cercanía de Π_q hacia todos los Π_u, esta cercanía se calcula usando Π_q^{-1} y Π_u. Una vez que se calcularon las cercanías entre Π_q y $\mathbb{U} - \mathbb{G}$ se ordenan de menor a mayor, ya que las que tienen menor distancia son más similares. Esto nos permitirá establecer el orden en que deberán revisar los elementos en la base de datos, es decir, $u \in \mathbb{U}$ más relevantes para q. Cada elemento es comparado usando la distancia $d(u,q)$. Observe que sólo se comparará un porcentaje de elementos que nuestro algoritmo propone como candidatos, si este candidato propuesto se encuentra dentro del radio r, entonces se reporta el objeto. $Aprox.id$ contiene la posición de cada elemento de $\mathbb{U}$, así sabemos a que objeto nos referimos. En la sección de pruebas se explicará porqué solo se utiliza un porcentaje de elementos.

El Algoritmo 3 regresa la distancia a la que se encuentra el K-ésimo vecino más cercano. En este algoritmo se utilizó un arreglo ordenado de tamaño K. Al igual que en el Algoritmo 2, se calcula la cercanía entre Π_q y todos los Π_u. También se ordenan de menor a mayor y así se establecen los elementos más relevantes. Después se realizan las comparaciones externas para encontrar a los KNN. Si un elemento tiene menor distancia que cualquiera de los que están en el arreglo, entonces se desecha el elemento mas alejado (mayor distancia hacia q) y se introduce el nuevo elemento. La función *radio* regresa la distancia hacia el K-ésimo vecino mas cercano.

En ambas funciones se utilizó la función *SpearmanFootrule* la cual básicamente calcula la similaridad entre permutaciones, según lo descrito en la Ecuación 2.7.

Algoritmo 2 BusquedaPorRadio

Entrada: I,q,r
Salida: *cont*

```
1:  Sea Aprox un arreglo con tuplas (entero, entero)
2:  cont ← 0
3:  Π_q^{-1} ← CalculaQInv(I, q)
4:  for all u ∈ U y u ∉ G do
5:      Aprox[u].d ←SpearmanFootrule(Π_q^{-1}, Π_u)
6:      Aprox[u].id ← posicion(u)
7:  end for
8:  Aprox ← qsort(Aprox)
9:  for i to n do
10:     d ← d(q, Aprox[i].id)
11:     if d ≤ r then
12:         reportar Aprox[i].id
13:         cont ← cont + 1
14:     end if
15: end for
16: return cont
```

Los tres criterios utilizados para conocer la cercanía entre un elemento u y un grupo

Algoritmo 3 BusquedaDeKNN

Entrada: I,q,K
Salida: r

1: $r \leftarrow -1$
2: $KNN \leftarrow$
3: Sea C un arreglo de tamaño K
4: Sea $Aprox$ un arreglo con tuplas (entero, entero)
5: $cont \leftarrow 0$
6: $\Pi_q^{-1} \leftarrow CalculaQInv(I,q)$
7: **for all** $u \in \mathbb{U}$ y $u \notin \mathbb{G}$ **do**
8: $\quad Aprox[u].d \leftarrow SpearmanFootrule(\Pi_q^{-1}, \Pi_u)$
9: $\quad Aprox[u].id \leftarrow posicion(u)$
10: **end for**
11: $Aprox \leftarrow qsort(Aprox)$
12: **for** i to n **do**
13: $\quad d \leftarrow d(q, Aprox[i].id)$
14: $\quad$ **if** $d \leq r$ OR $r = -1$ **then**
15: $\quad\quad KNN \leftarrow push(C, Aprox[i].id, d)$
16: $\quad\quad$ **if** $i \geq K$ **then**
17: $\quad\quad\quad r \leftarrow radio(C)$
18: $\quad\quad$ **end if**
19: $\quad$ **end if**
20: **end for**
21: **return** r

G_i $Dmin$, $Dmax$ y $Davg$ permiten construir la permutación de la consulta, de la misma forma que se hizo para cada elemento en la base de datos (en el Algoritmo 1). Estos criterios serán sometidos a pruebas para comprobar su efectividad en el siguiente capítulo.

Una parte importante en el desempeño de este algoritmo es la selección de buenos permutantes [FP10], por lo que la última parte de este capítulo esta orientada a explicar cual es la propuesta para seleccionar buenos permutantes por grupo.

3.5. Selección de buenos permutantes por grupo

Como se pudo ver en la Figura 3.1, los permutantes dentro de cada grupo se encontraban cercanos entre sí, pero si se eligen los permutantes de manera aleatoria es muy difícil que esto ocurra. Los criterios que se tomarán en cuenta para conformar nuestros grupos de permutantes serán los siguientes:

- Aleatorios (ADG). Se escogen elementos de manera aleatoria para formar un grupo (como se explico en la Sección 3.2).

- Cercanos dentro del grupo (CDG). Se escoge a un permutante p y se toman los más cercanos a p para formar el resto del grupo.
- Alejados dentro del grupo (LDG). Se escoge a un permutante p y se toman los más lejanos a p para formar el resto del grupo.

El criterio ADG fue explicado en la Sección 3.2, aquí se centrará en explicar los dos criterios restantes.

El criterio CDG toma $k * m$ permutantes y elige k permutantes "centrales", y los grupos se conforman con el resto de los permutantes, tomando los mas cercanos a cada permutante central como se observa en la Figura 3.2. El Cuadro 3.1 muestra $\mathbb{G}$ antes de reagrupar, es decir $\mathbb{G} = \{1,2,3,4,5,6\}$ y $G_1 = \{1,2\}$, $G_2 = \{3,4\}$ y $G_3 = \{5,6\}$. El Cuadro 3.2 muestra cuando ya estan agrupados, por lo que quedaría $\mathbb{G} = \{1,6,3,2,5,4\}$ y $G_1 = \{1,6\}$, $G_2 = \{3,2\}$ y $G_3 = \{5,4\}$.

El criterio LDG sigue el mismo procedimiento que el criterio CDG, la única diferencia es que los grupos se forman con los más alejados a cada permutante central.

Cuadro 3.1: Seleccionando los permutantes centrales para cada grupo

1	2	**3**	4	**5**	6

Cuadro 3.2: Reordenando los permutantes según el criterio CDG.

1	6	**3**	2	**5**	4

Hasta ahora hemos visto diferentes criterios para evaluar la cercanía entre los elementos y los grupos de permutantes. También hemos visto 3 diferentes criterios para conformar los grupos, esto con el fin de evaluar los diferentes desempeños de cada uno y así poder tomar el que entregue mejores resultados. Esto se verá en el siguiente capítulo.

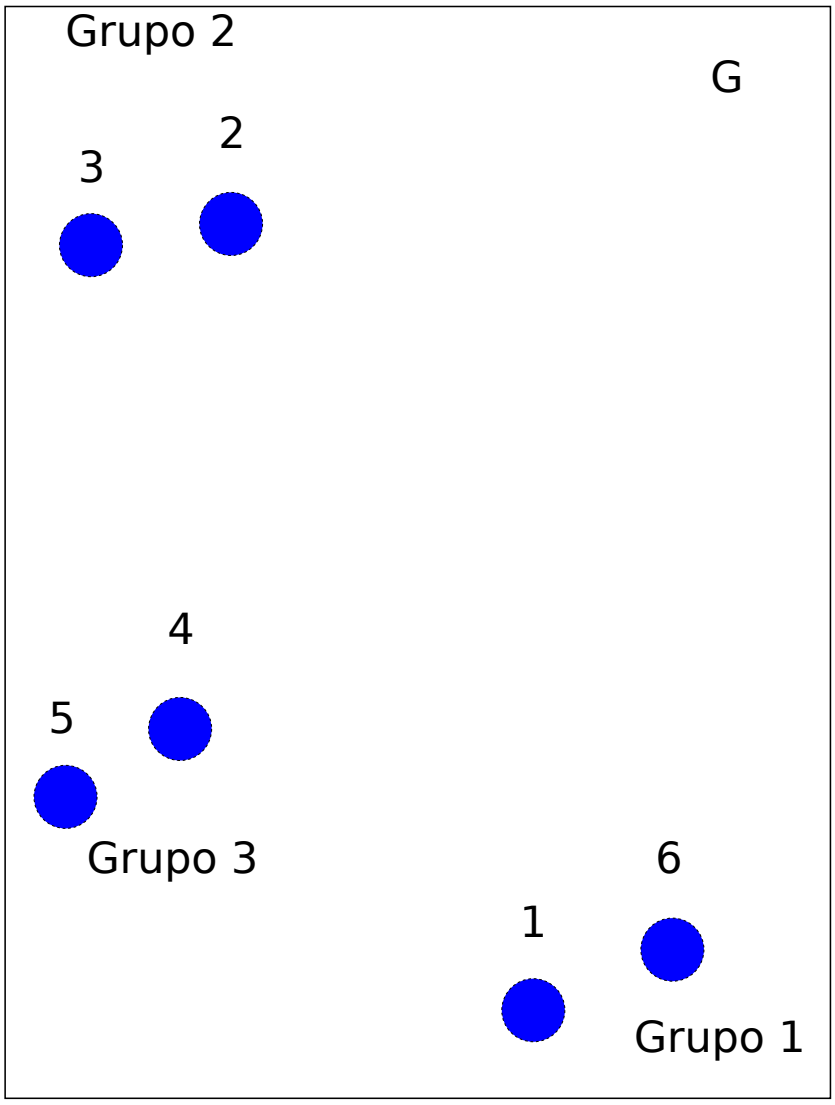

Figura 3.2: En esta figura se muestra como quedarían conformados los grupos usando el criterio CDG.

Capítulo 4

Pruebas y resultados

En este capítulo se expone la fase experimental que consta de 2 partes: las pruebas realizadas en bases de datos sintéticas (BDS) y las pruebas realizadas a bases de datos reales (Nasa, Colors). Primero se explicarán las condiciones de las bases de datos, después se mostrarán los resultados experimentales de seleccionar la cercanía hacia los grupos y finalmente se presentarán los experimentos del desempeño de seleccionar permutantes de acuerdo a un criterio.

4.1. Base de datos sintéticas y reales

La diferencia al aplicar pruebas en bases de datos reales y en BDS es que en las primeras no podemos controlar la dimensión intrínseca de los objetos que componen las bases de datos y en las BDS si. Las BDS son vectores uniformemente distribuídos en el cubo unitario, de esta manera sí podemos controlar la dimensión intrínseca de los datos.

La base de datos de objetos reales que llamaremos *colors*, es un conjunto de 112,544 histogramas de colores (vectores de dimensión 112) de una base de datos de imágenes [1]. La distancia usada entre objetos de esta base de datos fue la Euclidiana.

La base de datos de objetos reales que llamaremos *nasa* es un conjunto de 40,150 vectores (vectores de dimensión 20), generados de imágenes descargadas de la NASA [2]. La distancia usada entre objetos de esta base de datos fue la Euclidiana.

Las pruebas fueron realizadas en una computadora con las siguientes características:

- Procesador: Intel® Pentium® T4300 (2.10GHz), Mobile Intel® GL40 Express Chipset.
- Memoria: 2GB DDR2 SDRAM.

[1] en `http://www.dbs.informatik.uni-muenchen.de/šeidl/DATA/histo112.112682.gz`
[2] en `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`

- Sistema operativo: Mandriva 2010.1

4.2. Selección de la medida de cercanía hacia los grupos (fase 1)

Aquí se presentarán las pruebas realizadas para saber que medida de distancia entre grupos D_i es la más adecuada y se utilizó permutantes aleatorios (ADG) para formar los grupos de permutantes.

4.2.1. Pruebas en bases de datos sintéticas

A pesar de que se realizaron muchas pruebas, en este capítulo sólo se mostrarán las más relevantes. En la Figura 4.1 se muestran los rendimientos de los diferentes criterios de cercanía usados dentro del grupo de permutantes, que son: *Davg*, *Dmin* y *Dmax*. Las etiquetas se conforman de dos partes $m = X$ dice cuantos permutantes por grupo y *avg* para representar *Davg* por ejemplo. En la figura se puede apreciar que el rendimiento usando *Davg* es mejor respecto a los otros 2 criterios utilizados, ya que realizan menos comparaciones de distancia y es lo que se pretende reducir. Véase que en $m = 2, avg$ para recuperar el 100 % de los elementos, se utilizaron 4000 comparaciones, mientras que en $m = 4, may$ para obtener el 100 % de los elementos se utilizaron 6000 comparaciones, esto es 20 % más respecto a $m = 2, avg$.

La Figura 4.2 muestra el máximo número de comparaciones realizadas en las dimensiones de 16, 32 y 64 utilizando de 1 a 4 permutantes por grupo, en dicha gráfica se puede apreciar que usando un solo permutante tiene casi el mismo desempeño que por grupos.

4.2.2. Pruebas en bases de datos reales

Al igual que en la sección anterior, se realizaron exhaustivas pruebas a bases de datos reales (Nasa y Colors) y a continuación se muestran los resultados más sobresalientes.

En la Figura 4.3 se muestra una búsqueda de los 10 vecinos más cercanos en la base de datos Nasa. Se puede apreciar que a medida que se usan más permutantes por cada grupo, las comparaciones iniciales aumentan. También se puede observar, que a medida que se recuperan los vecinos más cercanos, los grupos de permutantes que tienen más de un permutante los recuperan más rápido que el que solo tiene un permuntante. Véase que la línea marcada como $m = 4$, cuando

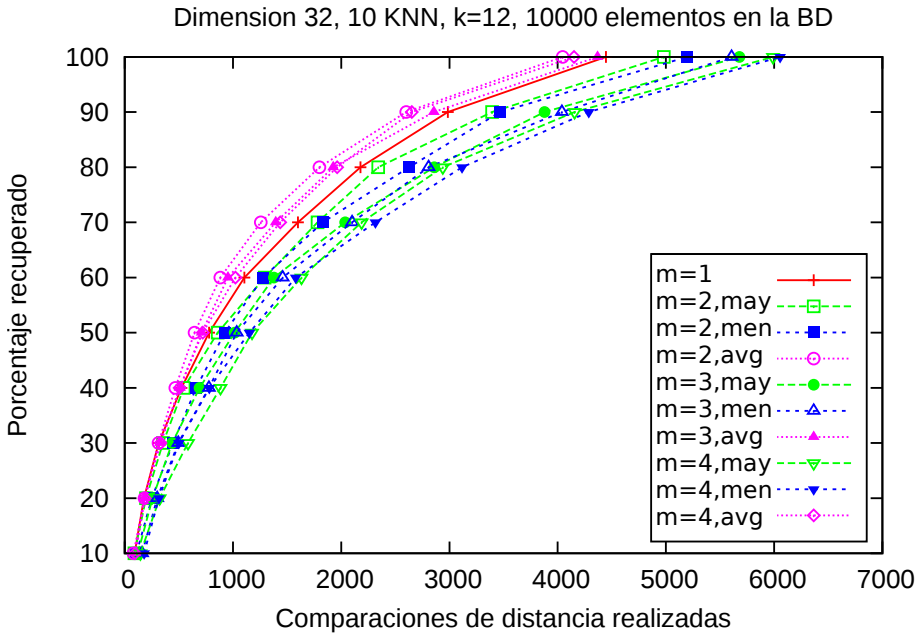

Figura 4.1: En esta figura se muestran las comparaciones de distancia realizadas por cada uno de los criterios.

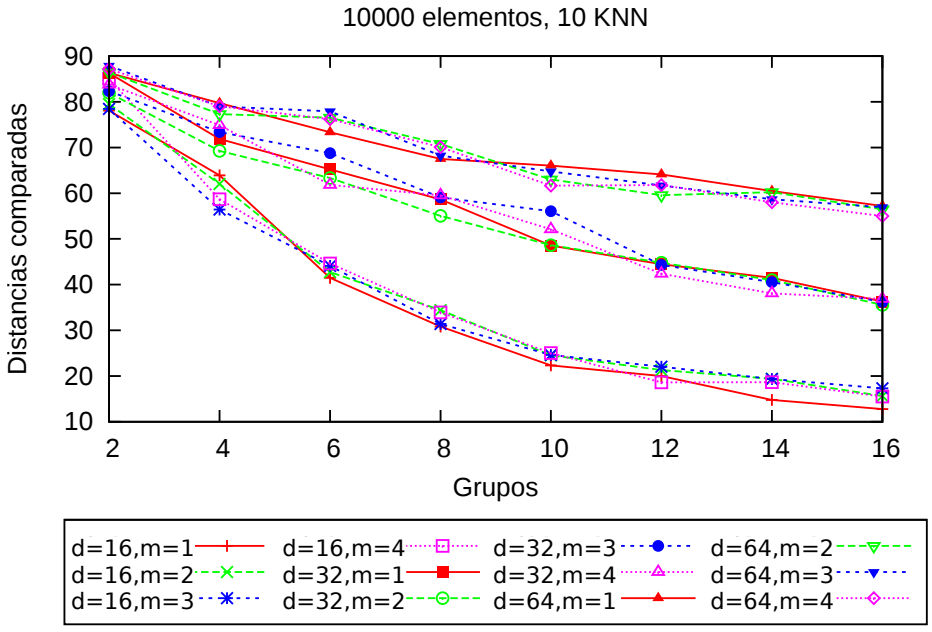

Figura 4.2: En esta figura se muestran las comparaciones máximas realizadas contra el número de grupos usados en cada consulta.

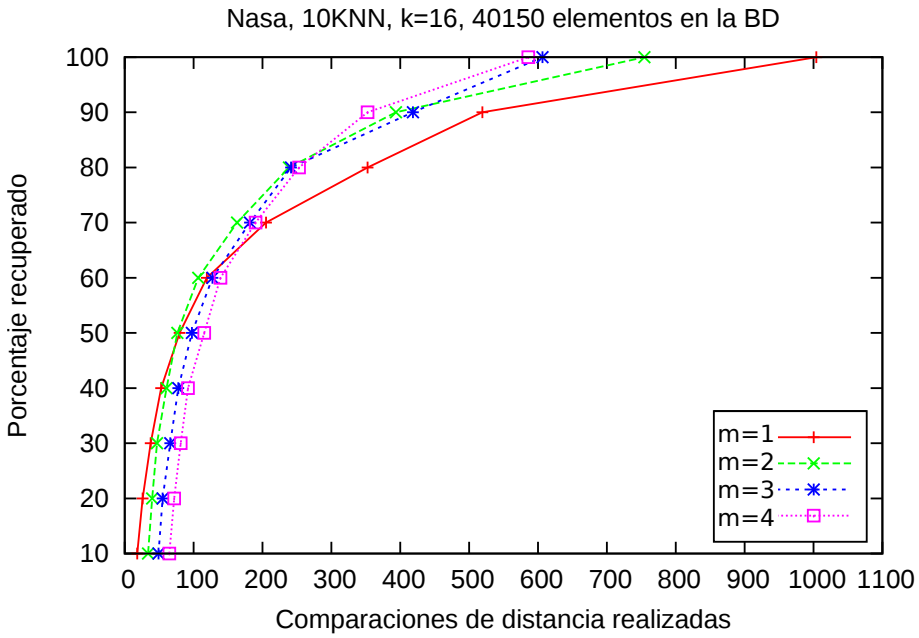

Figura 4.3: En esta figura se muestran las comparaciones de distancia realizadas usando el criterio del promedio.

recupera el 100 % de los elementos ha realizado menos de 600 comparaciones y la línea marcada como $m = 1$ al recuperar la misma información realizó poco más de 1000 comparaciones, lo cual implica 400 comparaciones más.

La Figura 4.4 muestra el resultado de usar grupos de permutantes en la base de datos de Colors. Se puede apreciar en la gráfica que el rendimiento usando grupos de permutantes es mejor comparado con usar un solo permutante. Véase que al utilizar 4 permutantes por grupo (línea $m = 4$) recupera el 90 % aproximadamente en un 5 % de las comparaciones de distancia, mientras que usando un solo permuntante para obtener la misma recuperación se utiliza 15 % de comparaciones de distancia, esto es 10 % mas comparaciones de distancias.

Después de seleccionar el criterio con mejor rendimiento (*Davg*), ahora se mostrarán las pruebas realizadas para obtener la mejor manera de agrupar dentro del grupo de permutantes, tomando en cuenta ADG, CDG y LDG.

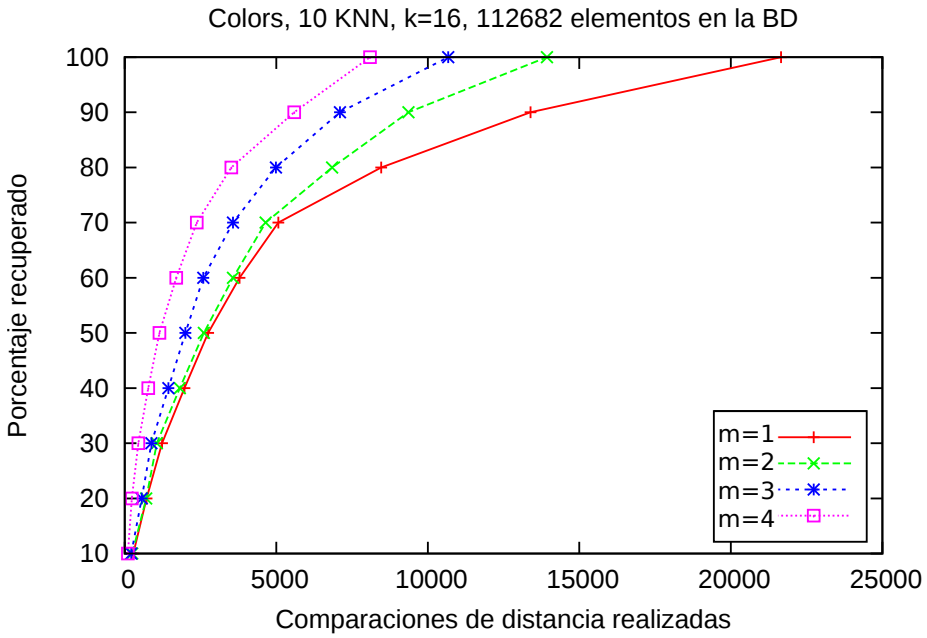

Figura 4.4: En esta figura se muestran las comparaciones de distancia realizadas usando el criterio del promedio.

4.3. Seleccionando buenos permutantes por grupo (fase 2)

En esta sección se presentarán las pruebas obtenidas al seguir diferentes criterios para agrupar los permutantes dentro de un grupo.

4.3.1. Pruebas en bases de datos sintéticas

En la Figura 4.5 se muestra el rendimiento de los dos criterios antes mencionados comparados contra la agrupación de permutantes aleatorios. Véase que el criterio utilizado tomando en cuenta la distancia más alejada hacia los permutantes centrales siempre realiza mas cálculos de distancia que usando un solo permutante por grupo, lo que nos indica que son mejores los otros dos criterios. Por otro lado, se observa que realiza menos comparaciones utilizando permutantes aleatorios para 2 y 4 permutantes por grupo.

4.3.2. Pruebas en bases de datos reales

Al igual que en la sección anterior, se realizaron exhaustivas pruebas a bases de datos reales (Nasa y Colors) y a continuación se muestran los resultados más sobresalientes.

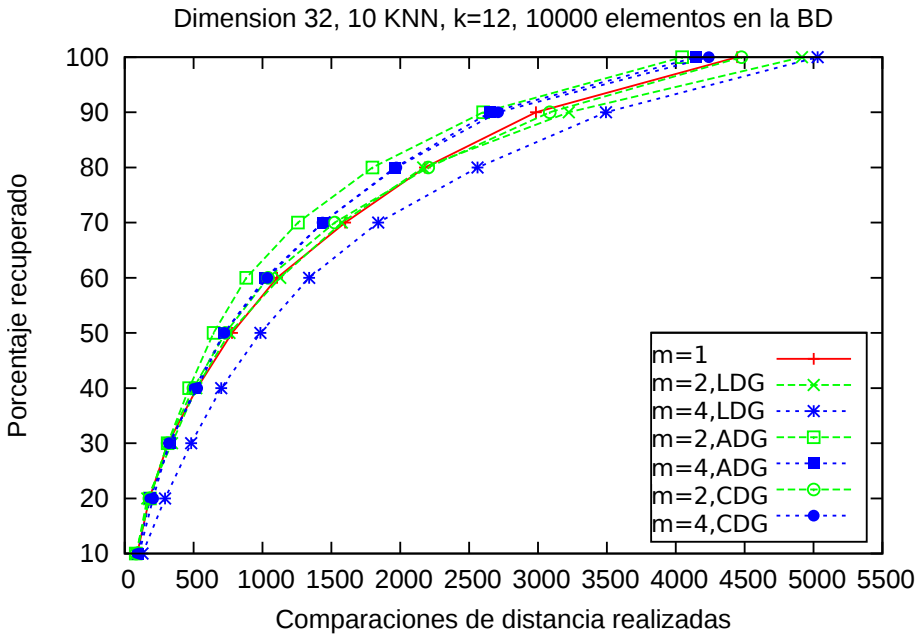

Figura 4.5: En esta figura se muestran las comparaciones de distancia realizadas por cada uno de los criterios ADG, CDG y LDG.

En la Figura 4.6 se muestra una búsqueda de los 10 vecinos más cercanos en la base de datos Nasa. Se puede apreciar que aún seleccionando buenos permutantes dentro del grupo, el rendimiento sin seleccionar los permutantes sigue siendo un poco mejor. Véase que en la línea $m = 4, ADG$ al recuperar el 100 % de los elementos, se utilizaron poco menos de 600 comparaciones, mientras que en la línea $m = 3, LDG$, al tener la misma recuperación se utilizaron aproximadamente 700 comparaciones, esto es 100 comparaciones más. A pesar de esto, utilizando este criterio sigue siendo mejor que usar un sólo permutante por grupo puesto que, en la linea $m = 1$ se observa que para tener la misma recuperación se utilizan aproximadamente 1000 comparaciones, que son 300 más que las que utiliza la linea $m = 3, LDG$ y 400 más que las que usa $m = 4, ADG$.

La Figura 4.7 muestra el resultado de usar grupos de permutantes en la base de datos colors. Se puede apreciar en la gráfica que el rendimiento usando grupos de permutantes con 2 permutantes por grupo, es menos eficiente que utilizando 3 y 4 permutantes por grupo. Además, al seleccionar permutantes por grupo, el rendimiento sigue siendo mejor utilizando permutantes aleatorios que utilizando un algoritmo para seleccionarlos, aunque el rendimiento basado en grupos de permutantes sigue siendo mejor que utilizando un solo permutante. Véase que las líneas $m = 4, ADG$ y $m = 4, LDG$ tienen una recuperación similar, pero la primera sigue siendo mucho mejor respecto a la segunda.

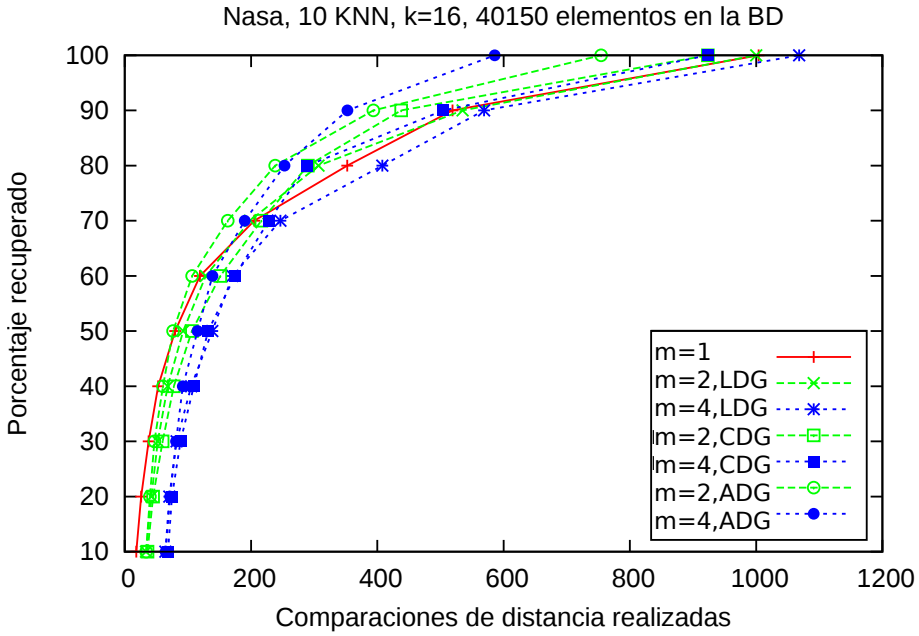

Figura 4.6: En esta figura se muestran las comparaciones de distancia realizadas usando los 3 criterios LDG,CDG y ADG.

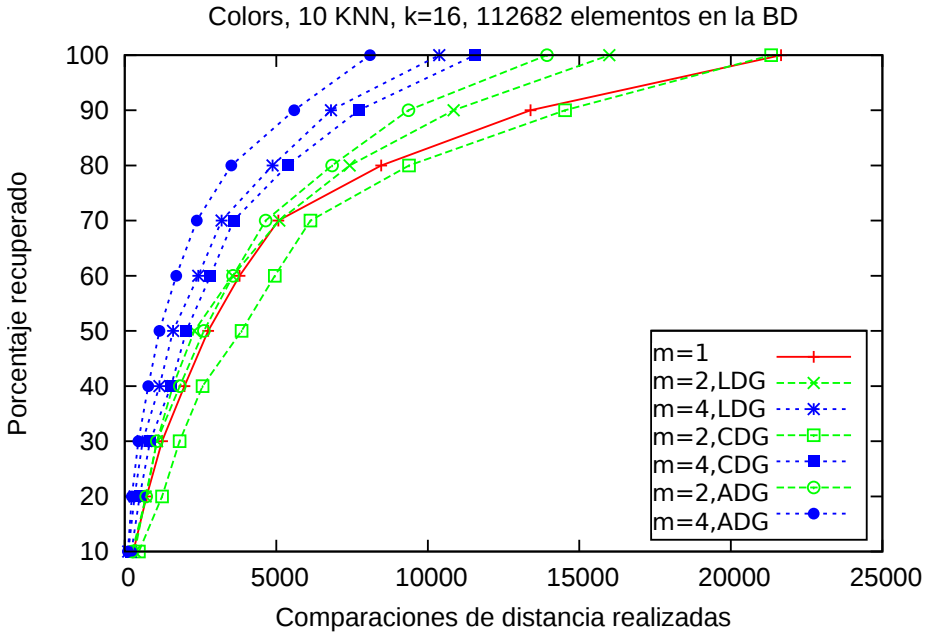

Figura 4.7: En esta figura se muestran las comparaciones de distancia realizadas usando los 3 criterios LDG,CDG y ADG.

En este capítulo se presentaron algunas de las pruebas realizadas, algunos de los resultados se esperaban, mientras que otros no. Un caso particular era el criterio *Davg*, que era el que se esperaba con un mejor rendimiento respecto a los otros 2 *Dmin* y *Dmax*. Por otro lado, utilizando el criterio CDG para agrupar, se esperaba un mejor rendimiento respecto a utilizar ADG, pero la realidad en las graficas es que ADG sigue siendo mejor. Una posible explicación a esto pudiera ser que, como se mostro en la Figura 3.2, a medida que los primeros permutantes centrales conforman su grupo, los demás permutantes se van quedando sin opciones.

Capítulo 5

Conclusiones y trabajos futuros

En esta tesis se presentó una alternativa para las búsquedas por similaridad en bases de datos multimediales. La propuesta es una modificación a un algoritmo existente y que modela la base de datos como un espacio métrico. Un espacio métrico consiste en una base de datos y una función de distancia entre los elementos que generalmente es costosa de calcular, por lo que el objetivo será disminuir este cálculo.

La propuesta aquí planteada pretende mejorar uno de los algoritmos recientes (algoritmos basados en permutaciones) en espacios métricos. La contribución principal es haber logrado reducir el número de comparaciones respecto a la técnica original y tener índices más compactos sin usar más memoria para la construcción del índice. En el Capítulo 4 se muestra que en algunos casos se tienen reducciones importantes de hasta un 35 % como en la Figura 4.3.

Algunos trabajos pendientes consisten en mejorar los algoritmos planteados para seleccionar los permutantes dentro del grupo. Un ejemplo claro sería el cómo seleccionar el permutante central para formar mejores agrupaciones. Por otro lado, se plantea también la idea de utilizar la varianza en un conjunto más grande que el que va a conformar los permutantes que se vayan a usar. Esto para conocer los permutantes más cercanos entre sí y utilizar esos permutantes para poder conformar los grupos, siguiendo nuevamente algoritmos para conformar los grupos. Otra propuesta para conformar los grupos es utilizando el algoritmo de “k-medias” el cuál recibe el conjunto de objetos y los agrupa en “k” grupos por cercanía.

Bibliografía

[BBK01a] S. Berchtold, C. Böhm, and D. Keim. Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, pages 322–373, 2001.

[BBK+01b] S. Berchtold, C. Böhm, D. Keim, F. Krebs, and H.P. Kriegel. On optimizing nearest neighbor queries in high-dimensional data spacies. *In 8th International Conference Database Theory (ICDT), volume 1973*, pages 435–449, 2001.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben79] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, 1979.

[CFN05] 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.

[CFN09] 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.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[Fig07] K. Figueroa. *Indexación Efectiva de Espacios Métricos usando Permutantes.* PhD thesis, Facultad de Ciencias Físicas y Matemáticas, Departamento de Ciencias de la Computación, Universidad de Chile, Junio 2007.

[FP10] K. Figueroa and R. Paredes. Finding good permutants. *IEEE International Conference on Information Security and Artificial Intelligence, ISBN: 978-1-4244-8870-4*, IEEE Catalog Number: CFP1058L-PRT(1), 2010.

[Gut84] 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.

[HS95] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. *In Fourth Inter*, pages 28–39, 1995.

[SBK96] D. Keim S. Berchtold and H. Kriegel. The x-tree: an index structure for high-dimensional data. *In Proceedings 22nd Conference on Very Large Databases (VLDB'96)*, pages 28–39, 1996.

Apéndice A

Cabecera del programa principal

```
#ifndef CABECERASINCLUDED
#define CABECERASINCLUDED

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/times.h>
#include <stdlib.h>
#include <math.h>
#include "../../../basics.h"
#include "../../../obj.h"
#include "../../../index.h"
#include <time.h>
#include <sys/times.h>

#define MAX_PAL 255
#define MENOR 1
#define MAYOR 2
#define MEDIA 3
#define PROMEDIO 4

typedef struct t_miIndex
{
  int tam;
  char *dbname;
  int piv;
  int *permutantes;
  int grupos;
}miIndex;

typedef struct t_dist
{
  Tdist d;
  int id;
}dist;

int **tabla;
int *matriz;
int *objetos;
int *grupos;

void sortarr(int *ar2,int tam);
void build_table(Index A);
int compara(const void *a, const void *b);
Tdist distanceGroup(int pos,int *qinv,miIndex *a);
void printPerm(int *perm, int tam);

#endif
```

Apéndice B

Programa principal

```
#include "setPermutants2.h"

int compara(const void *a, const void *b){
  dist *a1=(dist *)a;
  dist *b1=(dist *)b;
  if(a1->d<b1->d) return -1;
  if(a1->d>b1->d) return 1;
  if(a1->id<b1->id) return -1;
  if(a1->id>b1->id) return 1;
  return 0;
}

int compara2(const void *a, const void *b){
  dist *a1=(dist *)a;
  dist *b1=(dist *)b;
  if(a1->d<b1->d) return 1;
  if(a1->d>b1->d) return -1;
  if(a1->id<b1->id) return -1;
  if(a1->id>b1->id) return 1;
  return 0;
}

void sortarr(int *ar2,int tam){
  int i;
  for(i=1;i<tam;i++){
    int j=i;
    while(j>0 && ar2[j]<ar2[j-1]){
      int temp=ar2[j];
      ar2[j]=ar2[j-1];
      ar2[j-1]=temp;
      j--;
    }
  }
}

int pertenece(int pos,int *permutantes,int tam){
  int i;
  for(i=0;i<tam;i++)
    if(pos==permutantes[i])
      return 0;
  return 1;
}

void build_table_pro(Index A){
  miIndex *a = (miIndex *)A;
  int k,i,p,rows=a->tam,piv=a->piv,grupos=a->grupos;
  dist *ar2;
  ar2=malloc(grupos * sizeof(dist));
  for(i=grupos*piv+1;i<=rows;i++){
    tabla[i-(grupos*piv+1)] = matriz + ((i-(grupos*piv+1)) * grupos);
    for(k=0;k<grupos;k++){
      ar2[k].d=(Tdist)0.0;
      ar2[k].id=k;
      for(p=0;p<piv;p++)
ar2[k].d+=distance(a->permutantes[k*piv+p],i);
      ar2[k].d=piv;
    }
    qsort(ar2,grupos,sizeof(dist),compara);
    for(k=0;k<grupos;k++)
      tabla[i-(grupos*piv+1)][k]=ar2[k].id;
  }
```

```
}

void varianza(miIndex *a) {
  int long i, j, cont;
  dist *elemVar;
  int tam=a->tam10;
  int grupos=a->grupos;
  int piv=a->piv;
  elemVar = malloc(sizeof(dist) * a->tam10);
  for (i = 1; i <= tam; i++) {
    float s = 0.0, s2 = 0.0, mu, sigma2, aproxDim;
    for (j = 1; j <= a->tam; j++) {
      Tdist d = distance(i,j);
      s += d;
      s2 += d*d;
    }
    cont = a->tam;
    mu = scont, sigma2 = (s2 - cont*mu*mu)(cont-1), aproxDim = mu*musigma22;
    elemVar[i-1].d = aproxDim;
    elemVar[i-1].id = i;
  }
  qsort(elemVar, tam, sizeof(dist), compara);
  for(i=tam-grupos*piv; i< tam; i++)
    a->permutantes[i-(tam-grupos*piv)] = elemVar[i].id;
}

int buscaPos(int perm,int permutantes[],int tam){
  int i;
  for(i=0;i<tam;i++){
    if(perm==permutantes[i]) return i;
  }
  return 0;
}

void makegroup(miIndex *a) {
  int i,j,k,pos;
  if(a->piv==1) return;
  dist *arr;
  int marcas[a->grupos*a->piv];
  arr=malloc(sizeof(dist)*(a->grupos*a->piv-a->grupos));
  for(i=0;i<a->grupos*a->piv;i++){
    if(i%a->piv==0) continue;
    marcas[i]=0;
  }
  for(i=0;i<a->grupos*a->piv;i+=a->piv){
    pos=0;
    for(j=1;j<a->grupos*a->piv;j++){
      if(j%a->piv==0) continue;
      arr[pos].d=distance(a->permutantes[i],a->permutantes[j]);
      arr[pos].id=a->permutantes[j];
      pos++;
    }
    qsort(arr,a->grupos*a->piv-a->grupos,sizeof(dist),compara2);
    k=1;
    pos=0;
    while(k<a->piv){
      if(marcas[arr[pos].id-1]==0){
marcas[arr[pos].id-1]=1;
int temp=a->permutantes[i+k];
a->permutantes[i+k]=arr[pos].id;
int posperm=buscaPos(arr[pos].id,a->permutantes,a->grupos*a->piv);
a->permutantes[posperm]=temp;
k++;
      }
      pos++;
    }
  }
}

Index build (char *dbname, int n, int *argc, char ***argv) {
  miIndex *a;
  a = (miIndex *)malloc(sizeof(struct t_miIndex));

  if(*argc < 2)
  {
    printf("sintaxis: ........ group permutants\n");
    exit(1);
  }
  a->dbname = malloc(strlen(dbname)+1);
  strcpy(a->dbname, dbname);
  a->tam = openDB(dbname);
  a->grupos = atoi(argv[0][0]);
  argc--;
  a->piv = atoi(argv[0][1]);
  argc--;
  int rows=a->tam,grupos=a->grupos;
  matriz=malloc((rows-a->grupos*a->piv) * grupos * sizeof(int));
  tabla=malloc((rows-a->grupos*a->piv) * sizeof(int *));
```

```
  a->permutantes = (int *) malloc(sizeof(int) * a->grupos*a->piv);
  int i;
  for(i=0;i<a->grupos*a->piv; i++)
    a->permutantes[i]=i+1;
  makegroup(a);
  build_table_pro(a);
  return (Index)a;
}

Tdist distanceGroup(int pos,int *qinv,miIndex *a){
  long int dist=0;
  long int tot;
  int i;
  for(i=0;i<a->grupos;i++){
    tot=qinv[tabla[pos][i]]-i;
    tot=tot > 0 ? tot : -tot;
    dist+=tot;
  }
  return (Tdist)dist;
}

void calculaDistancias(Obj obj,dist *aprox,int *qinv,miIndex *a){
  int i;
  for(i=a->grupos*a->piv+1;i<=a->tam;i++){
      aprox[i-(a->grupos*a->piv+1)].id=i;
      aprox[i-(a->grupos*a->piv+1)].d=distanceGroup(i-(a->grupos*a->piv+1),qinv,a);
  }
  qsort(aprox, a->tam-a->piv*a->grupos, sizeof(dist), compara);
}

void calculaPermInvQProm(Obj obj,int *perm,int *perminvq,miIndex *a, Tcelem *res, int k){
  dist *ar2;
  ar2=malloc(a->grupos * sizeof(dist));
  int p,j,r=-1;
  int grupos=a->grupos,piv=a->piv;
  if(res==NULL){
    for(k=0;k<grupos;k++){
      ar2[k].id=k;
      ar2[k].d=0.0;
      for(p=0;p<piv;p++)
ar2[k].d+=distance(a->permutantes[k*piv+p],obj);
      ar2[k].d=piv;
    }
  }
  else{
    for(k=0;k<grupos;k++){
      ar2[k].id=k;
      ar2[k].d=0.0;
      for(p=0;p<piv;p++){
Tdist d=distance(a->permutantes[k*piv+p],obj);
if(d<=r||r==-1){
  addCelem(res,(Obj)(a->permutantes[k*piv+p]),d,(k*piv+p));
  if(res->csize>=k)
    r=radCelem(res);
}
ar2[k].d+=d;
      }
      ar2[k].d=piv;
    }
  }
  qsort(ar2,a->grupos,sizeof(dist),compara);
  for(j=0;j<a->grupos;j++){
      perminvq[ar2[j].id]=j;
      perm[j]=ar2[j].id;
  }
}

int search(Index S, Obj obj, Tdist r, bool show) {
  miIndex *a = (miIndex *)S;  los objetos empiezan en 1
  int i, cont=0;
  int *perminvq;
  int *perm;
  perminvq=(int *)malloc(a->grupos*sizeof(int));
  perm=(int *)malloc(a->grupos*sizeof(int));
  calculaPermInvQProm(obj,perm,perminvq,a,NULL,0);
  dist *Tabla_aprox;
  Tabla_aprox=(dist *)malloc((a->tam-a->grupos*a->piv)*sizeof(dist));
  calculaDistancias(obj,Tabla_aprox,perminvq,a);
  for (i=a->grupos*a->piv+1 ; i<= a->tam; i++) {
    Tdist d=distance(obj, Tabla_aprox[i-1].id);
    if(d <= r)
    {
      cont++;
      printf("%d\n", i);
    }
  }
  free(Tabla_aprox);
  return cont;
```

```
}


void prnstats(Index S) {
  imprime estadisticas (opcional)
}

void printPerm(int *p,int n){
    int i;
    for(i=0;i<n;i++)
      printf("%d ",p[i]);
    printf("\n" );
}

Tdist searchNN (Index S, Obj obj, int k, bool show) {
  miIndex *a = (miIndex *)S;
  Tcelem res = createCelem(k);
  int i,*perminvq,*perm;
  perminvq=(int *)malloc(a->grupos*sizeof(int));
  perm=(int *)malloc(a->grupos*sizeof(int));
  calculaPermInvQProm(obj,perm,perminvq,a,&res,0);
  Tdist r = radCelem(&res);
  dist *Tabla_aprox;
  Tabla_aprox=malloc((a->tam-a->grupos*a->piv)*sizeof(dist));
  calculaDistancias(obj,Tabla_aprox,perminvq,a);
  for (i=a->grupos*a->piv+1; i<=a->tam; i++) {
    Tdist d = distance(obj,Tabla_aprox[i-(a->grupos*a->piv+1)].id);
    if(d <= r ) {
      addCelem (&res,(Obj)Tabla_aprox[i-(a->grupos*a->piv+1)].id,d,i);
      if(res.csize >= k)
r = radCelem(&res);
    }
  }
  free(perminvq);
  free(perm);
  free(Tabla_aprox);
  int *pos;
  pos=malloc(k*sizeof(int));
  for(i=0;i<k;i++)
    pos[i]=res.elems[i].calc_real;
  sortarr(pos,k);
  for(i=0;i<res.csize;i++)    {
    printf("%d %g\n",pos[i],(float)((i+1)*100k));
    printf("%g\n",(float)pos[i]);
  }
  freeCelem(&res);
  return (Tdist) r;
}

void freeIndex(Index S, bool libobj) {
  free(matriz);
  free(tabla);
  miIndex *a = (miIndex *)S;
  free(a->permutantes);
  if (libobj) closeDB();
}

void saveIndex(Index S, char *fname) {
  int j;
  char src[MAX_PAL];
  int i;
  FILE *fp;
  miIndex *a = (miIndex *)S;
  int cols, rows;
  cols=a->grupos*a->piv;
  rows=a->tam;
  if( (fp=fopen(fname,"wb")) == NULL )   {
    fprintf(stderr,"Error al abrir el archivo para guardar");
    exit(-1);
  }
  strcpy(src, a->dbname);
  fwrite(src,MAX_PAL,1,fp);
  fwrite(&a->tam, sizeof(int), 1, fp);
  fwrite(&a->piv, sizeof(int), 1, fp);
  fwrite(&a->grupos, sizeof(int), 1, fp);
  for(j=0;j<a->grupos*a->piv;j++)
    fwrite(&a->permutantes[j],sizeof(int),1,fp);
  for(i=a->grupos*a->piv+1;i<=rows;i++)
    for(j=0;j<a->grupos;j++)
      fwrite(&tabla[i-(a->grupos*a->piv+1)][j],sizeof(int),1,fp);
  fclose(fp);
}

Index loadIndex(char *fname) {
  int j;
  char str[MAX_PAL];
  int i;
  FILE *fp;
```

```
  miIndex *a;
  a = (miIndex *)malloc(sizeof(struct t_miIndex));
  if( (fp=fopen(fname,"r")) == NULL )   {
    fprintf(stderr,"Error al leer el archivo\n");
    exit(-1);
  }
  fread(str,MAX_PAL,1,fp);
  a->dbname = malloc(strlen(str)+1);
  strcpy(a->dbname, str);
  fread(&a->tam, sizeof(int), 1,  fp);
  fread(&a->piv, sizeof(int), 1,  fp);
  fread(&a->grupos, sizeof(int), 1, fp);
  matriz=malloc((a->tam-a->piv*a->grupos) * a->grupos * sizeof(int));
  tabla=malloc((a->tam-a->piv*a->grupos) * sizeof(int *));
  a->permutantes = (int *) malloc(sizeof(int) * a->grupos*a->piv);
  objetos=(int *)malloc(sizeof(int) * (a->tam-a->grupos*a->piv));
  for(j=0;j<a->grupos*a->piv;j++)
    fread(&a->permutantes[j],sizeof(int),1,fp);
  for(i=a->grupos*a->piv+1;i<=a->tam;i++){
    tabla[i-(a->grupos*a->piv+1)] = matriz + ((i-(a->grupos*a->piv+1)) * a->grupos);
    for(j=0;j<a->grupos;j++)
      fread(&tabla[i-(a->grupos*a->piv+1)][j],sizeof(int),1,fp);
  }
  fclose(fp);
  openDB(a->dbname);
  return (Index) a;
}
```