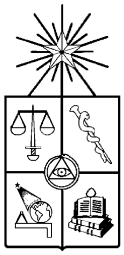


Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Indexación Efectiva de Espacios Métricos usando Permutaciones

Por

Karina Mariela Figueroa Mora

Tesis para optar al grado de Doctor en Ciencias
mención Computación

Profesores guía	:	**Gonzalo Navarro Badino**
	:	**Edgar Chávez González**
Miembros de la comisión	:	Claudio Domingo Gutiérrez Gallardo
	:	Marcelo Arenas Saavedra
	:	Enrique Vidal Ruiz (profesor externo, Universidad Politécnica de Valencia, España)

Santiago - Chile
Junio de 2007

Este trabajo fue financiado por Núcleo Milenio Centro de Investigación de la Web, Mideplan, Chile y la Universidad Michoacana de San Nicolás de Hidalgo, México.

RESUMEN DE LA TESIS
PARA OPTAR AL GRADO DE DOCTOR
EN CIENCIAS, MENCIÓN COMPUTACIÓN
POR: KARINA MARIELA FIGUEROA MORA
FECHA: 13-06-2007
PROF. GUÍAS : Sr. GONZALO NAVARRO
: Sr. EDGAR CHÁVEZ

Indexación Efectiva de Espacios Métricos usando Permutaciones

En muchas aplicaciones multimedia y de reconocimiento de patrones es necesario hacer consultas por proximidad a grandes bases de datos modelándolas como un espacio métrico, donde los elementos son los objetos de la base de datos y la proximidad se mide usando una distancia, generalmente costosa de calcular. El objetivo de un índice es preprocesar la base de datos para responder consultas haciendo el menor número de evaluaciones de distancia.

Los índices métricos existentes hacen uso de la desigualdad triangular para responder consultas de proximidad, ya sea partiendo el espacio en regiones compactas o utilizando distancias precalculadas a un conjunto distinguido de elementos. En esta tesis presentamos una nueva manera de resolver el problema, representando los elementos como permutaciones. La permutación se obtiene eligiendo un conjunto de objetos, llamados *permutantes*, y considerando el orden relativo en el que se ven los permutantes desde cada elemento a indexar.

Nuestra contribución principal es el haber descubierto que la proximidad entre elementos se puede predecir con mucha precisión midiendo la distancia entre las permutaciones que representan esos elementos.

Una aplicación directa de nuestra técnica deriva en un método probabilístico simple y eficiente: Se ordena la base de datos por proximidad de las permutaciones de los elementos a la permutación de la consulta, y se recorre en ese orden. De la comparación experimental de esta técnica contra el estado del arte, en diversos espacios reales y sintéticos, se concluye que las permutaciones son mucho mejores predictores de proximidad que las técnicas hasta ahora usadas, sobre todo en dimensiones altas. Generalmente basta revisar una pequeña fracción de la base de datos para tener un alto porcentaje de la respuesta correcta.

Otra aplicación menos directa de nuestra técnica consiste en modificar el algoritmo exacto AESA, que por 20 años ha sido el índice más eficiente, en términos de cálculos de distancia, para buscar en espacios métricos. Nuestra variante, iAESA, utiliza las permutaciones para determinar el siguiente candidato a compararse contra la consulta. Los resultados experimentales muestran que es posible mejorar el desempeño de AESA hasta en 35 %. Esta técnica es adaptable a otros algoritmos existentes.

Se aplicó nuestra técnica al problema de identificación de rostros en imágenes, y se lograron resultados hasta ahora no alcanzados por los típicos algoritmos vectoriales usados en estas aplicaciones. Asimismo, dado que nuestra técnica no aplica explícitamente la desigualdad triangular, la probamos en algunos espacios de similaridad no métrica, obteniendo un índice que permite la búsqueda por proximidad con resultados semejantes al caso de los espacios métricos.

A mis padres

Agradecimientos

No quiero cometer el gran pecado de omitir a alguien en esta sección, aunque eso sea una labor muy difícil. El orden de mención no refleja ni modifica la importancia y lo que las personas son en mi vida. De una u otra forma cada una contribuyó durante la realización de este trabajo en la que el crecimiento no sólo fue de índole académica.

Agradezco a todo el personal del DCC por su valiosa ayuda en cada momento y trámite de mi estancia. Gracias por esa disposición y sobre todo la amabilidad de sus atenciones.

A mi asesor y una gran persona Gonzalo Navarro, quien es un admirable líder. Gracias por haberme concedido la oportunidad de trabajar bajo tu guía.

A mi co-asesor Edgar Chávez por tu apoyo incondicional, por tus enseñanzas, por la gran motivación inyectada en esta etapa de mi vida.

Quiero manifestar mi agradecimiento a la comisión por sus acertados comentarios los cuales contribuyeron a enriquecer este trabajo. De manera especial agradezco a Javier Ruiz del Solar por el apoyo recibido para el desarrollo de uno de los capítulos de esta tesis.

Definitivamente es imposible no estar eternamente agradecida con Karina Muñoz Cofré y su hermosa Familia. Gracias por hacer de mi estancia en Chile una segunda casa, y por recibirme como parte de esa familia.

A todos mis compañeros del DCC: Gilberto Gutierrez, Rodrigo Paredes, Olivier Motelet, Renzo Angles, George Drupet, Javier Bustos, Cuauhtemoc Rivera, Manuel Varas, gracias por hacer de cada día de trabajo un momento inolvidable. De manera especial quiero agradecer a mi inigualable compañero de oficina Gilberto Gutierrez, gracias por tus sabios consejos, por permitirme aprender mucho de ti, y por cada momento en que compartimos logros y tropiezos (sobre todo aquel día decisivo en nuestras respectivas investigaciones).

Gracias a grandes amigos como Milton, Mariela, Francia, Oscar, Florian, Juanma, Benoit por su cariño, por su apoyo y especialmente por su valiosa amistad.

Un agradecimiento muy especial es para Tonathiu Sánchez, por la motivación e impulso que siempre le diste a este sueño y sobre todo por haber sido un pilar muy importante en este proyecto.

Gracias a la vida por darme la dicha de conocer a cada uno de ustedes (y a los que seguramente estoy omitiendo).

Índice general

1. Introducción **2**

1.1. Aplicaciones 4

1.2. Aporte de la Tesis 6

1.3. Organización de la Tesis 7

I CONCEPTOS BÁSICOS **10**

2. Conceptos Básicos **11**

2.1. Consultas por Proximidad 12

2.1.1. Consultas de los *K* Vecinos más Cercanos 13

2.2. Espacios Vectoriales 14

2.2.1. Tipos de Distancias 14

2.2.2. Índices 15

2.3. Efectos de la Dimensión Intrínseca Alta 16

2.4. Concepto de Índice Métrico 17

2.5. Definiciones 18

2.5.1. Tipos de Relaciones 19

2.5.2. Cadenas y Árboles (Tries) 20

2.6. Bases de Datos de Prueba 21

2.6.1. Vectores en el Cubo Unitario 21

2.6.2. Diccionarios 21

2.6.3. Documentos 22

3. Estado del arte **25**

3.1. Clasificación de Algoritmos 25

3.1.1. Algoritmos Basados en Pivotes 25

3.1.2. Algoritmos Basados en Particiones Compactas 28

3.2. Descripción de los Algoritmos 30

3.3. Comparación del Estado del arte 39

3.4. Algoritmos Inexactos 42

II NUESTRA TÉCNICA **48**

4. Técnica Basada en Inversiones **49**

4.1. Definiciones 50

4.2. Comparación de las Inversiones contra las Familias Existentes 53

4.2.1. Inversiones frente a los Algoritmos de Pivotes 53

4.2.2. Inversiones frente a las Particiones Compactas 54

4.3. Uso de la Técnica Basada en Inversiones 56

4.3.1. Indexación 56

4.3.2. Proceso de Búsqueda 58

4.4. Experimentación 59

4.4.1. Diccionario 60

4.4.2. Cubo unitario 60

5. Algoritmo Basado en Permutaciones **63**

5.1. Comparación entre Permutaciones 64

5.2. Buscando las Permutaciones más Similares 65

5.2.1. Empleando un Trie 66

5.2.2. Ordenando el Conjunto 68

5.3. Experimentación 70

5.3.1. Documentos TREC 72

5.3.2. Comparación entre Exactos y Probabilísticos 73

5.4. Selección de Permutantes 73

III APLICACIONES 82

6. Nuevo Algoritmo Exacto: iAESA 83

6.1. AESA 83

6.1.1. Proceso de Búsqueda de AESA 84

6.2. iAESA 86

6.2.1. Proceso de Búsqueda de iAESA 86

6.2.2. Comparando AESA con iAESA 86

6.2.3. Combinando AESA e iAESA 88

6.3. Versiones Probabilísticas 89

6.3.1. Limitando el Trabajo 90

6.3.2. Relajando las Condiciones de Búsqueda 90

6.4. Resultados Experimentales 90

6.4.1. iAESA Exacto 91

6.4.2. Algoritmos Probabilísticos 92

6.5. Cálculo Dinámico del Footrule 97

6.5.1. Experimentación 107

7. Aplicaciones: Identificación de Caras 111

7.1. Reconocimiento de Caras 111

7.1.1. Transformación de Objetos 112

7.1.2. Reducción Dimensional 113

7.2. Descripción de las Bases de Datos 114

7.2.1. Descripción de la Base de Datos *BD-762* 115

7.2.2. Descripción de la Base de Datos *BD-7327* 115

7.3. Resultados en *BD-762* 115

7.3.1. Experimentos 118

7.4. Resultados en *BD-7327* 119

7.4.1. Experimentos 119

8. Bases de Datos No Métricas **125**

8.1. Norma Fraccionaria L_p 125

8.1.1. Experimentos 126

9. Conclusiones **132**

9.1. Trabajo Futuro 134

A. Nubes de Puntos **144**

A.1. Cubo Unitario 145

A.1.1. Dimensión 8 145

A.1.2. Dimensión 128 145

A.2. Documentos 145

A.3. Base de datos de Caras FERET 151

A.4. Diccionario 151

A.5. Espacios no Métricos 151

A.5.1. L_p con $p = 0,2$ 151

A.5.2. L_p con $p = 0,8$ 151

Índice de figuras

1.1. Proyección de un objeto complejo. 3

2.1. Tipos de consultas por proximidad. 13
2.2. Ejemplo de histogramas de distancias. 17
2.3. Proceso de consulta en la búsqueda por similaridad. 18
2.4. Ejemplo de un trie. 21
2.5. Histograma de distancias del cubo unitario. 22
2.6. Histograma de distancias de las palabras. 23
2.7. Histograma de distancias de los documentos. 24

3.1. Ejemplo de un algoritmo basado en pivotes. 27
3.2. Ejemplo de un algoritmo basado en particiones compactas. 29
3.3. Familias de algoritmos del estado de arte. 31
3.4. Ejemplo de AESA. 32
3.5. Ejemplo del GNAT. 33
3.6. Ejemplo de BKT, FQT, FHQT Y FQA. 35
3.7. Ejemplo de un BKT. 36
3.8. Ejemplo de un VPT. 37
3.9. Ejemplo de un BST. 39
3.10. Ejemplo de un SAT. 40
3.11. Estado de arte en el tiempo. 41
3.12. Estado de arte. Dimensión 8 y 20. 44
3.13. Diferencia entre algoritmos exactos y aproximados. 46

4.1. Ejemplo de órdenes en un espacio $\mathbb{R}^2$. 50
4.2. Ejemplo de las inversiones en $\mathbb{R}^2$. 51
4.3. Comparación entre los algoritmos basados en particiones compactas y los basados en inversiones. 55
4.4. Comparación entre las particiones de Voronoi frente a los permutantes. 55
4.5. Trie de permutaciones. 57
4.6. Ejemplo del proceso de búsqueda en un trie basado en permutaciones. 58
4.7. Desempeño de la técnica de las inversiones en un diccionario de palabras. 61
4.8. Desempeño de la técnica de las inversiones en el cubo unitario. 62

5.1. Ejemplo de la técnica basada en permutaciones. 65
5.2. Tiempo de procesamiento de la técnica de las permutaciones con un Trie. 69
5.3. Tiempo de procesamiento de la técnica basada en permutaciones usando qsort e IQS. 75
5.4. Tiempo de procesamiento para la técnica de las permutaciones (usando PowerPC). 76
5.5. Tiempo de procesamiento para la técnica de las permutaciones (usando Intel). . . . 77
5.6. Desempeño de los algoritmos probabilísticos en dimensiones altas. 78
5.7. Comparando diferentes medidas de similaridades entre permutaciones. 79
5.8. Comparación de las métricas L_1 y L_∞ como base de ordenamiento. 79
5.9. Desempeño de la técnica basada en permutaciones en un espacio de documentos. . 80
5.10. Comparación del desempeño entre algoritmos exactos e inexactos. 80
5.11. Diferentes heurísticas para la selección de permutantes. 81

6.1. Ejemplo del proceso de AESA e iAESA. 89
6.2. Comparando iAESA contra AESA en distintas dimensiones. 91
6.3. Comparación de AESA e iAESA para documentos. 92
6.4. Comparación en tiempo entre AESA e iAESA. 93
6.5. Desempeño de AESA, iAESA e iAESA2 probabilísticos en dimensión 128. 94
6.6. Error relativo de las distancias en las consultas infructuosas. 95
6.7. Comparación de AESA, iAESA e iAESA2 en dimensión 128. 95
6.8. Comparación de iAESA, iAESA2 y ordenamiento usando permutaciones en dimensión 128. 96

6.9. Comparación de AESA, iAESA e iAESA2 probabilístico en dimensión 8. 98
6.10. Comparación de AESA, iAESA e iAESA2 aproximado en dimensión 16. 99
6.11. Desempeño de la técnica basada en permutaciones. Dimensiones 8 y 16. 100
6.12. Comparación entre la métrica dinámica del Footrule y la secuencial. 108
6.13. Comparación de tiempos de procesamiento de iAESA. 110

7.1. Sistema general de Reconocimiento de Caras. 113
7.2. Histogramas de la Base de Datos FERET (real y proyectado). 116
7.3. Histograma de la Base de Datos BD-7327 (real y proyectado). 117
7.4. Comparación del reconocimiento de caras sobre la base de datos FERET. 120
7.5. Tiempo de procesamiento para la base de datos de caras FERET. 121
7.6. Comparación del reconocimiento de caras sobre la base de datos FERET. 123
7.7. Tiempo de procesamiento para la base de datos de caras FERET. 124

8.1. Histograma de la desigualdad triangular. 126
8.2. Desempeño de las permutaciones bajo una norma fraccionaria, $p = 0,2$ y $p = 0,8$ dimensión 8. 128
8.3. Desempeño de las permutaciones bajo una norma fraccionaria, $p = 0,2$. 129
8.4. Desempeño de las permutaciones bajo una norma fraccionaria, $p = 0,8$. 130
8.5. Desempeño de las permutaciones bajo una norma fraccionaria. 131

A.1. Nubes de puntos: Cubo unitario, dimensión 8. 146
A.2. Nubes de puntos: Cubo unitario, dimensión 128. 147
A.3. Acercamiento en nubes de puntos: Cubo unitario, dimensión 128. 148
A.4. Nubes de puntos: Base de datos de documentos. 149
A.5. Acercamiento en nubes de puntos: Base de datos de documentos. 150
A.6. Nubes de puntos: Base de datos de caras. 152
A.7. Acercamiento en nubes de puntos: Base de datos de caras. 153
A.8. Nubes de puntos: Diccionario. 154
A.9. Acercamiento en nubes de puntos: Diccionario. 155
A.10. Nubes de puntos: Cubo unitario usando una norma fraccionaria L_p con $p = 0,2$. . . 156

A.11. Acercamiento nubes de puntos: Cubo unitario usando una norma fraccionaria L_p con $p = 0,2$. 157

A.12. Nubes de puntos: Cubo unitario usando una norma fraccionaria L_p con $p = 0,8$. . . 158

A.13. Acercamiento nubes de puntos: Cubo unitario usando una norma fraccionaria L_p con $p = 0,8$. 160

Índice de cuadros

3.1. Comparación de los algoritmos para la búsqueda por proximidad en espacios métricos. 43

Lista de algoritmos

1. Algoritmo-pivotero-indexado 27
2. Algoritmo-pivotero-consulta 27
3. Búsqueda-trie(T trie, d_{max}, *nivel*) 59
4. Probar-hoja(T trie, d_{max}, *nivel*) 60
5. Trie_búsqueda(Trie R, Consulta q) 67
6. Sort-Búsqueda(Π_q, q) 67
7. AESA 85
8. iAESA 87
9. Footrule-dinámico 105
10. Footrule-dinámico-estimado() 109

Capítulo 1

Introducción

La necesidad de procesar conjuntos de datos para obtener información ha estado presente en toda la historia de la humanidad. En 1963 nació el concepto de bases de datos, y en los setentas el *modelo relacional* para organizarlas [Cod70]. Este esquema continúa utilizándose en las bases de datos tradicionales, donde es posible estructurar los datos en tuplas y después hacer búsquedas por igualdad de campos en las tuplas. El modelo relacional ha resultado inmensamente exitoso por su capacidad de modelar correctamente la realidad en muchas aplicaciones de procesamiento de datos durante décadas.

Sin embargo, con la mayor cantidad y variedad de información disponible digitalmente de las últimas décadas, han comenzado a aparecer aplicaciones que no se adaptan fácilmente al modelo relacional, pues necesitan manipular datos llamados “no estructurados”. Entre los nuevos tipos de datos podemos mencionar huellas digitales, secuencias de audio, música, secuencias biológicas, etc. A las bases de datos capaces de almacenar este tipo de información las llamaremos “no tradicionales”.

En las bases de datos no tradicionales las búsquedas por igualdad ya no son posibles o simplemente no tienen sentido porque difícilmente los objetos calzarán exactamente, por lo tanto, la alternativa es la *búsqueda por similaridad o proximidad*. Este tipo de búsquedas consiste en recuperar de una base de datos los objetos más semejantes o relevantes a uno de consulta dado.

Cuando se trata de miles o millones de objetos y/o la comparación entre ellos es costosa (en tiempo y/o recursos), hacer una *búsqueda secuencial* resulta impensable, una alternativa ante esto son los índices, que permiten localizar rápidamente los elementos relevantes.

Un claro ejemplo de la utilidad de un índice es un diccionario físico, donde se tienen miles de palabras que sería muy lento de encontrar si estuvieran desordenadas, en lugar de estar ordenadas alfabéticamente. Lo mismo ocurre en las bases de datos en general, aunque construir un índice no sea tan simple como ordenar lexicográficamente.

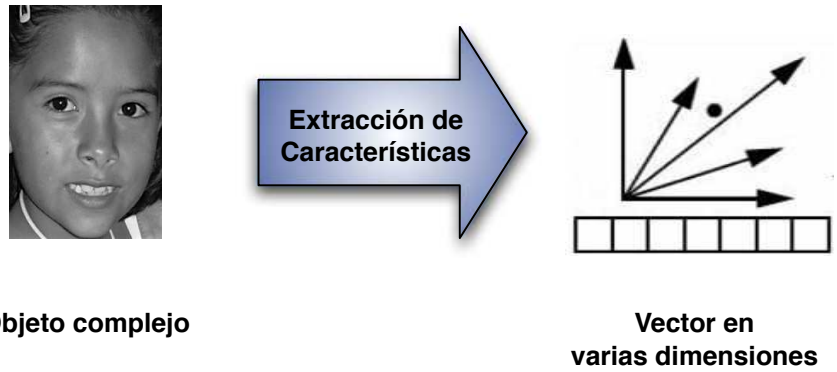


Figura 1.1: Trasformación de un objeto (Obj) a un vector de varias dimensiones R^m.

Veamos un ejemplo donde se aplica la búsqueda por similaridad y los índices. Piense en el proceso que debe realizar la policía internacional cuando quiere saber si un individuo (un sospechoso, regularmente) ha entrado al país. Una forma de hacerlo es buscar su rostro en la base de datos de todas las personas que entran al país. Considere que en la fotografía del sospechoso puede haber gestos que hacen variar la imagen entre dos imágenes del mismo individuo, y por lo cual es indispensable la búsqueda por similaridad. En un caso así, la base de datos tiene millones de registros y comparar la base de datos completa (*búsqueda secuencial*) es mucho más lento que una búsqueda con ayuda de un índice (*búsqueda indexada*).

Una parte importante de las bases de datos no tradicionales son las que almacenan objetos multimediales como audio, video e imágenes. 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 alta dimensión. Esto se conoce como extracción de características. Definimos la extracción de características δ como una transformación de un objeto (Obj) a un vector de múltiples 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 se transforma en encontrar los vectores con mayor proximidad entre sí.

Generalmente, en las BDMD se emplean índices que aprovechan las coordenadas 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 un intento por evitar esa maldición se ha propuesto otro tipo de índices como los basados en las distancias entre objetos: *índices para espacios métricos*.

Antes de definir un espacio métrico es necesario enfatizar que existen bases de datos donde simplemente no es posible transformar los elementos a espacios vectoriales. Un buen ejem-

plo de son las secuencias de ADN, donde el único recurso para la búsqueda es la semejanza entre una secuencia y otra.

Un espacio métrico está constituido por un conjunto de datos (por ejemplo: caras, secuencias de audio, secuencias de ADN, etc.) y una función de distancia que evalúa la semejanza entre dos elementos cualesquiera. Usualmente, la función de distancia es costosa de calcular (en tiempo y/o recursos), comparada con el tiempo de CPU empleado en recorrer un índice y decidir cuáles elementos son relevantes. De aquí que el objetivo de los algoritmos para la búsqueda por similaridad en espacios métricos es resolver las consultas utilizando la menor cantidad posible de comparaciones de distancia.

En los espacios métricos la función de distancia debe cumplir, entre otras propiedades, con la desigualdad triangular, la cual es empleada para evitar comparaciones de distancia al momento de resolver una consulta. En la práctica también son de interés los espacios no métricos, donde la desigualdad del triángulo no se garantiza, y por lo tanto no es posible evitar comparaciones, sin embargo, es posible aproximar una respuesta. En esta tesis consideraremos ambos tipos de espacios.

1.1. Aplicaciones

Existen muchas aplicaciones que utilizan el concepto de búsqueda por similaridad. Entre ellas podemos mencionar reconocimiento de patrones, recuperación de información y biología computacional.

- *Reconocimiento de patrones*. El propósito de un programa de reconocimiento de patrones es analizar una escena en el mundo real tomada por un dispositivo (como un escáner) y llegar a una descripción de la escena útil para completar alguna tarea. Por ejemplo, la aplicación podría ser en una oficina de correos; la descripción, una serie de números que identifican el código postal [1]; y la tarea, ordenar la correspondencia por regiones geográficas para su distribución.

  Básicamente el reconocimiento de patrones se realiza en tres etapas: la primera es la segmentación de los objetos útiles en la entrada (por ejemplo sobre imágenes). La siguiente etapa usualmente está destinada a eliminar el ruido de esos objetos que pudiera interferir en su reconocimiento. En la etapa final se lleva a cabo la clasificación del objeto en una o más categorías, y es aquí donde se requiere la búsqueda por similaridad, pues los objetos más parecidos entre sí deberían tener la misma categoría.

[1] Los números pueden ser tomados con un escáner e identificados usando reconocimiento de caracteres *OCR* (Optical Character Recognition)

Dentro del reconocimiento de patrones, existen un sinfín de aplicaciones en cuya etapa de clasificación es indispensable la búsqueda por proximidad, entre ellas [DHS73] reconocimiento de voz, identificación de oradores, reconocimiento de firmas, de caras, de letras, etc.

- *Recuperación de información*. La operación central en el proceso de recuperación de información (RI) es buscar en registros *no estructurados* (o semi-estructurados) aquellos más semejantes a uno de consulta. Un claro ejemplo de registro no estructurado es un documento en lenguaje natural. Cuando se tienen grandes volúmenes de información en forma de texto, las búsquedas usando índices son relevantes.

  Un ejemplo típico de RI es un buscador en internet. Generalmente el usuario proporciona un conjunto de palabras, que conforman una especie de "*documento de consulta*" y obtiene como respuesta un conjunto de documentos ordenados por su relevancia (o similaridad) para la consulta. Una forma de modelar un documento es verlo como un conjunto de palabras, donde los pesos de los términos forman un vector y una medida de semejanza entre ellos es la distancia coseno entre dos vectores [BYRN99].

  Una aplicación muy importante en la recuperación de información se da en bases de datos multimediales. El ejemplo perfecto para ilustrar el problema es la búsqueda de imágenes en internet. Generalmente las búsquedas se realizan utilizando etiquetas asignadas por el usuario y el nombre del archivo con que se accede a ellas, de esta manera se aprovecha la única información fácilmente disponible. Sin embargo, aún no es posible identificar los objetos dentro de las imágenes, y realizar una busqueda por similaridad de éstas.

- *Biología computacional*. En esta área se utilizan secuencias que definen objetos básicos, por ejemplo proteínas, ADN, etc. Si estas secuencias son modeladas como texto, entonces el problema es llevado a una búsqueda de cadenas de texto similares. En este tipo de búsquedas es poco probable que exista una secuencia idéntica a otra. Dependiendo de lo que se quiera evaluar al comparar dos secuencias, se utilizan distintas medidas de similitud entre cadenas, por ejemplo éstas pueden considerar las probabilidades de mutaciones, la similaridad entre estructuras tridimensionales, etc. [SK83, Wat95].

  Algunas aplicaciones en esta área son árboles filogenéticos [ZP65], identificar comportamientos en especies, clasificación de especies, identificación de restos humanos, etc.

  En este tipo de aplicaciones existe la posibilidad de errar en la respuesta pues el más ligero cambio en ciertas áreas importantes de cadenas es decisivo en la identificación. Sin embargo, esto es aceptable pues generalmente se realizan varias muestras que deberían confirmar o rechazar el resultado.

1.2. Aporte de la Tesis

En este trabajo se propone un enfoque distinto a los algoritmos existentes para la búsqueda por similaridad en espacios métricos, el cual muestra un muy buen desempeño en tiempo y calidad de las respuestas ante las consultas a una base de datos. A continuación describiremos los principales aportes de esta tesis.

- Un enfoque novedoso para la búsqueda por proximidad (o similaridad) en espacios métricos, donde básicamente cada elemento en la base de datos genera un *ordenamiento*. Los ordenamientos son el resultado de cómo un elemento *percibe* a cierto subconjunto de la base de datos (ordenados por cercanía). La idea novedosa consiste en utilizar la semejanza entre ordenamientos para predecir la similaridad. Esta idea se divide en dos alternativas de solución: inversiones y permutaciones.

- El algoritmo basado en inversiones consiste básicamente en encontrar *intercambios* o *inversiones* entre los elementos de dos *ordenamientos*. Las inversiones garantizan una cierta distancia entre dos elementos. Las inversiones que se producen entre una consulta y los elementos de la base de datos permiten descartar algunos de ellos sin compararlos. Este es un algoritmo exacto o determinístico, pues recupera el 100 % del conjunto respuesta a la consulta.

- El algoritmo basado en permutaciones es un algoritmo probabilístico. En éste el usuario limita el número de comparaciones para responder una consulta. Este algoritmo también usa el modelo donde cada elemento ordena un subconjunto de la base de datos, pero sólo se evalúa la *semejanza* entre ordenamientos para predecir la *promisoriedad* de cada elemento. Básicamente, el algoritmo consiste en ordenar los elementos de la base de datos por su promisoriedad para la consulta y recorrer en ese orden. Con este orden rápidamente se encuentra un alto porcentaje de la respuesta. Este algoritmo resulta ser imbatible en dimensiones altas.

- El algoritmo basado en permutaciones fue aplicado en bases de datos reales. Una de éstas fue la colección de fotografías con rostros humanos (FERET [PWHR98]). Los resultados previos de búsqueda usando índices en este tipo de aplicaciones han sido nulos por ser una base de datos en *alta dimensión*. En cambio, nuestro algoritmo basado en permutaciones muestra un buen desempeño en esta base de datos, llegando a ahorrar un 90 % de trabajo con respecto a los algoritmos exactos.

- La técnica basada en permutaciones se aplicó a bases de datos no métricas. Nuestra propuesta no transforma el espacio no métrico a uno métrico para utilizar un algoritmo para espacios métricos (como generalmente se hace). El desempeño es tan bueno como en las bases de datos métricas.

- Entre los algoritmos existentes, el mejor algoritmo exacto conocido fue superado usando la idea de las permutaciones, con lo cual se estableció un nuevo mínimo en el número de comparaciones necesarias para resolver una consulta por proximidad en espacios métricos.

Los artículos obtenidos a partir de esta tesis son:

- Resumen: “Detección de Inversiones para búsqueda por proximidad en espacios métricos”. Consorcio Doctoral. Encuentro Nacional de Computación 2004, Colima, México. Edgar Chávez, Karina Figueroa y Gonzalo Navarro.
- Artículo: “Proximity Searching in High Dimensional Spaces with a Proximity Preserving Order”. En MICAI’05 (5th Mexican International Conference in Artificial Intelligence), Monterrey, México. Noviembre 2005. Páginas 405-414. LNAI 3789. Edgar Chávez, Karina Figueroa y Gonzalo Navarro. *Obtuvo el Premio al tercer mejor artículo entre 120*.
- Artículo: “On the Least Cost For Proximity Searching in Metric Spaces”. En WEA’06 (5th Workshop Experimental Algorithms), Menorca, España. Mayo 2006. Páginas 270-290, LNCS 4007, Springer. Karina Figueroa, Edgar Chávez, Gonzalo Navarro y Rodrigo Paredes. Este artículo fue seleccionado para la edición especial del “ACM Journal of Experimental Algorithmics”, en un número especial dedicado a los mejores artículos de WEA’06.
- Versión de revista (enviado a IEEE Transactions on Pattern Recognition and Machine Intelligence) “Effective Proximity Retrieval by Ordering Permutations”. Edgar Chávez, Karina Figueroa y Gonzalo Navarro.
- Versión de revista (Invitado a ACM Journal of Experimental Algorithmics) “Speeding up Spatial Approximation Search in Metric Spaces”. Karina Figueroa, Edgar Chávez, Gonzalo Navarro y Rodrigo Paredes.
- Resumen: “A Novel Index for Proximity Searching in Metric Spaces”. Consorcio Doctoral. Encuentro Nacional de Computación 2006, San Luis Potosí, México. Karina Figueroa, Edgar Chávez y Gonzalo Navarro.

1.3. Organización de la Tesis

Este documento está organizado en tres partes: conceptos básicos, la parte algorítmica y la parte de las aplicaciones. En la primera parte se comienza presentando los conceptos básicos utilizados a lo largo de esta tesis, explicando detalladamente los problemas, los objetivos y alcances de esta tesis. Posteriormente, se hace una revisión del estado del arte de la búsqueda en espacios

métricos. En la segunda parte se presenta el enfoque propuesto en esta tesis, los algoritmos derivados con sus pruebas experimentales. Por último, en la tercera parte se presentan aplicaciones reales en las que fueron usados los algoritmos propuestos en esta tesis y cuyos resultados muestran una importante contribución en esos problemas.

A continuación se resumen los capítulos de esta tesis.

PARTE I. CONCEPTOS BÁSICOS

- Capítulo 2. Se explican los conceptos básicos, los problemas y desafíos en las búsquedas por proximidad en espacios métricos. En este capítulo también se describen las bases de datos de pruebas usadas a lo largo de la tesis.

- Capítulo 3.- Se hace una revisión de los algoritmos para la búsqueda por similaridad en espacios métricos. El estado del arte está dividido en dos familias: basadas en *pivotes* y basadas en *particiones compactas*. En cada una se presentan las bases formales, y las soluciones existentes. Para cada una de estas familias de soluciones se analiza la forma de procesar los datos, el tiempo de respuesta y la memoria ocupada. Este capítulo termina con una comparación entre las familias existentes y nuestra propuesta.

PARTE II. ALGORITMOS

- Capítulo 4.- Se presenta formalmente la nueva técnica basada en *inversiones*. La idea general es que cada elemento de la base de datos forme un *orden* según cómo percibe el espacio (o un subconjunto de éste), y use esa información para *deducir* qué objetos no son relevantes para las consultas. Esta técnica da lugar a una nueva familia de algoritmos para espacios métricos.

- Capítulo 5.- En este capítulo se presenta la versión probabilística de la técnica basada en el concepto de permutaciones (introducida en el capítulo 4), a esta técnica la llamaremos basada en permutaciones. La idea general ahora está enfocada a priorizar los elementos de la base de datos por *promisoriedad* adosada al orden ya descrito. Esta técnica resulta ser imbatible en la práctica, y su desempeño se muestra en bases de datos reales y sintéticas. En particular, en las bases de datos reales muestra un desempeño notable comparado contra todas las alternativas existentes.

PARTE III. APLICACIONES

- Capítulo 6.- La técnica presentada permite mejorar algunos de los algoritmos exactos existentes para espacios métricos. Presentamos un resultado interesante, el cual muestra que es posible reducir el número de comparaciones de distancia realizadas por el algoritmo que

marcaba la pauta del menor número de comparaciones posibles en las búsquedas por proximidad en los espacios métricos.

- Capítulo 7.- Una base de datos comúnmente usada en el reconocimiento de patrones es la base de datos FERET. Esta es un conjunto de imágenes faciales con alta dimensión. En este capítulo se estudia el desempeño de nuestra técnica y su precisión. En términos de precisión es posible alcanzar un 99.98 % de la precisión máxima alcanzable comparando sólo un 17 % de la base de datos.

- Capítulo 8.- En este capítulo se estudia el desempeño de nuestra técnica en las bases de datos no métricas. Este tipo de bases de datos se caracterizan porque su distancia no cumple con la desigualdad del triángulo (algo indispensable en los algoritmos existentes para espacios métricos). Los resultados en este tipo de bases de datos son tan notables como en las bases de datos métricas.

Finalmente, el Capítulo 9 resume nuestras contribuciones y muestra las líneas de investigación y trabajos futuros en esta área.

Parte I

CONCEPTOS BÁSICOS

Capítulo 2

Conceptos Básicos

En esta capítulo se explican algunos conceptos básicos utilizados a los largo de la tesis y en la última sección se describen los tipos de espacios que fueron usados en los experimentos de los capítulos posteriores para mostrar el desempeño de las técnicas presentadas.

El primer concepto que se quiere introducir 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 general de esta medida dado que está estrechamente ligada a la aplicación y a las características que se quieran evaluar.

Formalmente, sea $(\mathbb{X}, d)$ un espacio métrico, donde $\mathbb{X}$ es el universo de objetos válidos, y d es una función de distancia, $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ que 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: $d(x,x) = 0$

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

Si la distancia no satisface la propiedad de positividad estricta, entonces el espacio es llamado *pseudo-métrico*. En un espacio pseudo métrico pueden existir objetos diferentes que estén

a distancia cero entre sí, por ejemplo, si $d((x_1,x_2),(y_1,y_2)) = |x_1 - y_1|$, entonces dos puntos diferentes como $d((2,3),(2,5))$ tienen distancia cero. Este tipo de espacios puede ser fácilmente adaptado a los algoritmos presentados en este trabajo, identificando a todos los objetos cuya distancia sea cero como un mismo objeto, mediante una *clase de equivalencia*. Esto es posible porque, por las otras propiedades, vale que $d(x,y) = 0 \Rightarrow \forall\ z,\ d(x,z) = d(y,z)$.

En los casos en los que la propiedad de simetría no se cumple, el espacio se llama *cuasi-métrico*. Un ejemplo de este tipo de espacios es la distancia recorrida por un auto en una ciudad para ir de un punto a otro, considerando las esquinas como objetos, y la distancia como el recorrido por las calles tomando en cuenta el sentido del tráfico (la existencia de calles en un sólo sentido hace que la distancia sea asimétrica). Existen técnicas para derivar una nueva distancia simétrica desde una asimétrica, por ejemplo: $d'(x,y) = d(x,y) + d(y,x)$.

En el resto de esta tesis usaremos un conjunto finito de objetos $\mathbb{U} \subseteq \mathbb{X}$ de tamaño $n = |\mathbb{U}|$. Al conjunto de elementos de interés $\mathbb{U}$ lo llamaremos diccionario, base de datos, o simplemente conjunto de elementos. El término *distancia* será usado para referirnos a una métrica.

2.1. Consultas por Proximidad

Existen básicamente dos tipos de consultas por similaridad. Sea $q \in \mathbb{X}$ el elemento de consulta.

- **Consulta de rango** .- Consiste en recuperar todos los elementos en un radio r, es decir, $(q,r)_d = \{u \in \mathbb{U} \mid d(q,u) \leq r\}$.
- **Consulta de los K vecinos más cercanos** $KNN(q)_d$.- Recuperar los K elementos en $\mathbb{U}$ más cercanos a q. Esto es $A \subseteq \mathbb{U}$ tal que $\forall\ u \in A, \forall\ v \in \mathbb{U} - A,\ d(q,u) \leq d(q,v),\ K = |A|$.

El caso más común en muchas aplicaciones es cuando $K = 1$, es decir, se quiere recuperar al vecino más cercano (*NN* por sus siglas en inglés de *Nearest Neighbor*). Una variante a estas búsquedas es limitar el radio de interés r^*, esto significa que los vecinos más cercanos que no estén dentro del radio de interés r^* no serán reportados.

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 $(q,r)_d$ y en el derecho la consulta de los dos vecinos más cercanos (2NN).

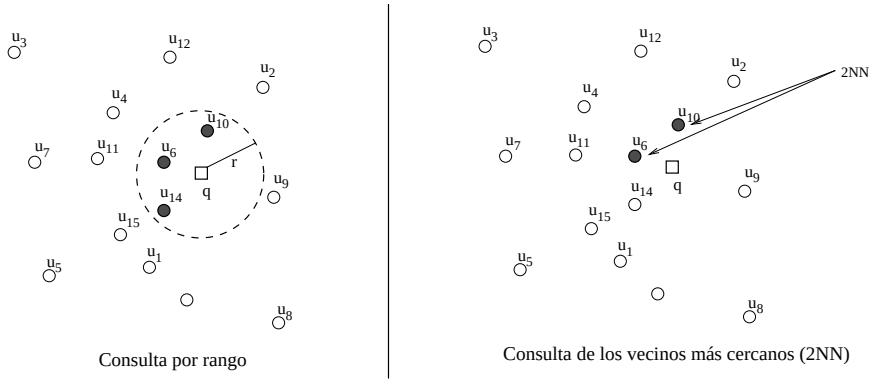


Figura 2.1: Tipos de consultas por proximidad en espacios métricos.

2.1.1. Consultas de los K Vecinos más Cercanos

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

El algoritmo para la búsqueda del vecino más cercano con esta técnica está basado en usar un algoritmo de rango como sigue: La búsqueda de q con radio fijo $r = a^i\epsilon$ ($a > 1$), inicia con $i = 0$ y se incrementa i hasta que al menos el número deseado de elementos (K) estén dentro del radio de consulta $r = a^i\epsilon$. Después hay un refinamiento del radio entre $r = a^i\epsilon$ y $r = a^{i-1}\epsilon$.

Dado que la complejidad de las búsquedas de rango crece abruptamente con el radio de búsqueda, el costo de este método es semejante al costo de una búsqueda por rango con el radio apropiado (cuando se conoce originalmente). El incremento de a puede ser muy pequeño ($a \to 1$) para evitar que el radio de consulta sea mucho más grande de lo necesario.

Backtracking con Reducción del Radio

Una búsqueda más elaborada del vecino más cercano con esta técnica (en principio para $K = 1$) consiste en iniciar la búsqueda con $r^* = \infty$. Cada vez que la consulta q es comparada contra un elemento p en la base de datos, se actualiza el radio de búsqueda $r^* = min(r^*, d(q, p))$ y continúa la búsqueda con el radio reducido.

A medida que se encuentran elementos más cercanos a la consulta, el radio se reduce y se refleja en una búsqueda menos difícil. Por esta razón es importante tratar de encontrar rápidamente los elementos más cercanos (note que esto no es relevante para las consultas por rango). Para este proceso de conocer rápidamente los elementos cercanos, los diferentes algoritmos dependen

directamente de la estructura de datos usada.

Resolver búsquedas de los K vecinos más cercanos es una extensión al algoritmo descrito anteriormente. Se mantienen los K elementos más cercanos conocidos hasta ahora y el radio se reduce a la distancia máxima entre aquellos K elementos. Cada vez que se compara un elemento contra la consulta, éste podría reemplazar a algún elemento de los candidatos y por ende debería actualizarse el radio de consulta.

Priorizando el *Backtracking*

Esta técnica consiste en recorrer de manera más inteligente los elementos. Por simplicidad usaremos el ejemplo de recorrido de un árbol para explicar esta técnica. La técnica propone seguir el recorrido de las ramas de un árbol con más libertad. La meta es incrementar la probabilidad de encontrar rápidamente los elementos más cercanos a q y reducir r^*. Esta técnica ha sido usada varias veces en espacios vectoriales y métricos [Uhl91, HS95, CPZ97, Nav02b].

La decisión de qué subárbol procesar está normalmente definida por la cota mínima de la distancia entre q y los elementos del subárbol; siempre eligiendo los subárboles con menor cota mínima. La mejor forma de implementar esto es usando una cola de prioridad ordenada por la cota mínima, donde se insertan y remueven subárboles. Inicialmente, la cola está vacía y se agrega sólo la raíz. El proceso es iterativo, se remueve de la cola el mejor subárbol (que se encuentre en ésta), se procesa y se insertan los subárboles relevantes hasta que la menor cota inferior sea mayor que r^*, donde ya es posible "podar" los subárboles sin perder respuestas relevantes.

2.2. Espacios Vectoriales

Un caso particular de los espacios métricos son los espacios vectoriales m-dimensionales. En los espacios vectoriales los objetos son identificados con m coordenadas de valores reales $(x_1, x_2, \ldots, x_m)$ y la distancia es la métrica L_p.

2.2.1. Tipos de Distancias

En el caso de los espacios vectoriales, existen algunas distancias usadas muy comúnmente dependiendo de la aplicación. Supongamos u y q como dos vectores con dimensión m, es decir, $u = (u_1, \ldots, u_m)$ y $q = (q_1, \ldots, q_m)$. La métrica comúnmente usada es la Minkowski de orden p o métrica L_p, definida como:

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

donde algunos casos especiales son:

Manhattan

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

Euclidiana

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

Máxima

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

2.2.2. Índices

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, algo no disponible en espacios métricos generales donde los algoritmos sólo usan la distancia entre objetos.

Los métodos para espacios vectoriales son conocidos como *Spatial Access Methods SAM* [BBK01b]. Entre los más populares están *KD-Tree* [Ben75, Ben79], *R-Tree* [Gut84] y *X-Tree* [BKK96], por mencionar algunos [HS00, BBK+01a]. En general, estas técnicas hacen uso exhaustivo de la información de las coordenadas para agrupar y clasificar los objetos en el espacio.

- *KD-Tree*.- Es el método para espacios vectoriales más popular. El *KD-Tree* es un árbol binario, donde cada nivel del árbol fue dividido usando un hiperplano perpendicular a una dimensión del vector. Esto es, en el nodo raíz el espacio se divide usando un hiperplano perpendicular a la primera dimensión; en el segundo nivel, a la segunda dimensión; así sucesivamente. El proceso puede ser recursivo cuando se han usado todas las dimensiones. Usualmente, este método deja de ser competitivo a partir de la dimensión 4 [Ben75, Ben79]. El *KD-Tree* se usa para memoria principal, una versión modificada para memoria secundaria es *K-D-B Tree* [Rob81].

- *R-Tree*.- Este tipo de árbol usa el concepto de hiper-rectángulos, cada nodo es un rectángulo que encierra todos los nodos hijos contenidos. Este método es muy similar a los B-Trees. El *R-Tree* almacena sus nodos internos al menos a la mitad de su capacidad y es un árbol balan-

ceado multiario donde todos los nodos hojas están al mismo nivel. El principal problema de este método es la intersección entre los nodos internos (la cual produce muchos caminos de búsqueda [Gut84]); y la dependencia del orden de inserción de los objetos. Este algoritmo también está diseñado para dimensiones bajas. A diferencia del *KD-Tree*, el *R-Tree* se usa para memoria secundaria. Una modificación donde tratan de reducir el traslape entre nodos minimizando el tamaño de los rectángulos es el *R*-Tree*.

- *X-Tree*.- La base del *X-Tree* es el *R*-Tree*. La novedad en este método es el incremento de la capacidad de cada nodo, al que llaman "super nodo". Sin embargo, el *X-tree* sigue siendo dependiente del orden de inserción de los datos y de la dimensión de éstos. Este método deja de ser competitivo en dimensiones superiores a 16.

Las técnicas existentes en espacios vectoriales son muy sensibles a la dimensión del espacio. En general tienen una dependencia exponencial con la dimensión, lo que se conoce como "*la maldición de la dimensionalidad*". En términos prácticos, se considera intratable el espacio de más de 20 dimensiones [CNBYM01]. En [YY85, BBKK97, BBK$^+$01a, BGRS99] se muestran ejemplos de métodos que comprueban la ineficiencia de un índice cuando crece la dimensión.

2.3. Efectos de la Dimensión Intrínseca Alta

En los espacios vectoriales puede existir una gran diferencia entre la dimensión *representada* y la dimensión *intrínseca*. La dimensión intrínseca representa el número real de dimensiones en las cuales los puntos pueden ser embebidos mientras mantienen (con poca distorsión) la distancia entre ellos. Por ejemplo, un plano embebido en un espacio de 50 dimensiones tiene dimensión intrínseca 2, mientras que su dimensión representada es de 50. Éste es un claro ejemplo de cómo difieren la dimensión real e intrínseca. Basados en esto varios autores han propuesto el uso de técnicas de indexación basadas en distancias (técnicas para espacios métricos), en un intento por evitar la *maldición de la dimensionalidad*. Estas técnicas usan sólo la distancia entre puntos y evitan cualquier referencia a coordenadas.

Al usar sólo la información de la distancia es importante conocer el efecto que sufren los algoritmos cuando se incrementa la dimensión intrínseca. La característica típica en espacios de dimensión intrínseca alta es que la distribución de las distancias entre elementos tiene un histograma concentrado, con una media mayor a medida que la dimensión crece. En un caso extremo tenemos un espacio donde $d(x,x) = 0$ y $\forall x \neq y$, $d(x,y) = 1$; allí es imposible evitar cálculos de distancia en una consulta, pues la distancia sólo nos dice si el elemento comparado es o no la respuesta.

En [CNBYM01] proponen una manera de definir la dimensión intrínseca de un espacio

métrico, como $\rho = \frac{\mu^2}{2\sigma^2}$, donde μ y σ^2 son la media y la varianza, respectivamente, de su histograma de distancias.

Para ilustrar el problema con la dimensión intrínseca alta, véase la figura 2.2, donde se muestran dos histogramas de distancias de dos bases de datos distintas, con respecto a un elemento distinguido $p \in \mathbb{X}$. El histograma del lado derecho (dimensión alta) está mucho más concentrado que el del lado izquierdo (dimensión baja). Piense que una consulta de rango q con radio r se aplica a estos dos espacios; y que la distancia entre p y q, representada por $d(p,q)$, se ubica al centro del histograma, que por cierto es su posición con mayor probabilidad. En ambos espacios las zonas sombreadas corresponden a los elementos que podrían ser descartados por la desigualdad triangular, sin compararse contra la consulta usando un índice (como se describe en la sección 3.1). Note que el número de elementos que podrían ser descartados es mayor en dimensión baja que en dimensión alta.

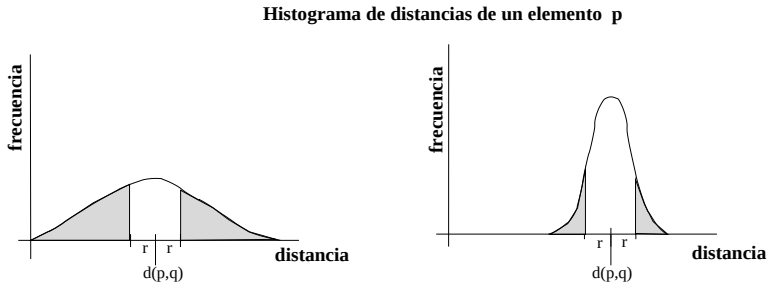


Figura 2.2: Histogramas de distancias de una consulta $(q,r)_d$ hacia un elemento p en dimensión baja y alta (izquierda y derecha, respectivamente). Las zonas sombreadas corresponden a los elementos que podrían ser descartados sin compararse contra la consulta usando un índice y la desigualdad triangular.

2.4. Concepto de Índice Métrico

En espacios métricos los algoritmos que resuelven consultas por proximidad generalmente 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 un conjunto de elementos (base de datos).

Generalmente, un algoritmo que resuelve consultas por proximidad tiene dos fases: la de preprocesamiento y la de consulta. En la figura 2.3 se muestra el proceso de preprocesamiento y consulta a una base de datos para obtener el conjunto respuesta. Inicialmente tenemos la base de datos, ésta se *proyecta* en un nuevo espacio que permite crear un índice (fase de preprocesamiento). El índice se crea una vez y se almacena, por lo tanto, el costo de su construcción no se considera al momento de resolver consultas. 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) y los elementos que pueden ser descartados de manera segura. En la lista de candidatos hay elementos que no son relevantes para la consulta, pero que no pudieron ser distinguidos por el índice. El caso ideal es que todos los elementos en la lista de candidatos sean relevantes para la consulta.

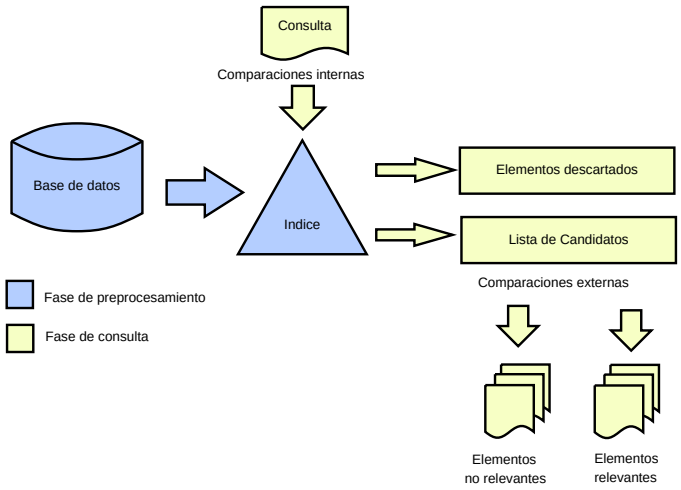


Figura 2.3: Proceso de tratamiento de la base de datos para resolver consultas usando un índice.

Una proyección de un espacio métrico puede verse como una reducción natural, un ejemplo es proyectar el espacio original en un espacio vectorial, mediante una función ϕ. De esta manera cada elemento en el espacio original es representado como un punto en el nuevo espacio vectorial. 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 $D(\phi(x),\phi(y))$, donde $x,y \in \mathbb{X}$.

Si la proyección es *contractiva* ($D(\phi(x),\phi(y)) \leq d(x,y)$), entonces es posible hacer búsquedas por rango en el espacio proyectado con el radio original. Sin embargo, debido a que pueden ser seleccionados algunos falsos positivos en el espacio proyectado, el resultado obtenido en este espacio es conocido como la lista de candidatos, la que deberá ser verificada en el espacio original para obtener la lista real del conjunto respuesta (comparaciones externas). En un índice, incrementar las comparaciones internas mejora cualitativamente el poder de filtro y reduce las comparaciones externas, por lo tanto buscamos un balance óptimo entre ambas.

2.5. Definiciones

Durante el desarrollo de la tesis se usarán tres conceptos muy importantes: relación, orden y árboles de búsqueda. En esta sección haremos una breve descripción de estos conceptos.

2.5.1. Tipos de Relaciones

El concepto de relación implica la idea de correspondencia entre los elementos de dos o más conjuntos. Un ejemplo de relación son las binarias. Sea A un conjunto cualquiera, se dice que $\sim$ una relación binaria en A si:

$$\sim \subseteq A \times A$$

Las relaciones binarias pueden tener algunas de estas propiedades:

- Si $\forall x \in A$, $x \sim x$, entonces la relación es reflexiva.
- Si $\forall x \in A$, x no se relaciona consigo mismo, $\sim$ es antirreflexiva.
- Si siempre que $x \sim y$ se cumple $y \sim x$, entonces $\sim$ es simétrica.
- Si siempre que $x \sim y$ e $y \sim x$ se tiene que $x = y$, entonces la relación es antisimétrica.
- Si siempre que $x \sim y$ e $y \sim z$ se cumple $x \sim z$, entonces la relación es transitiva.

Un tipo de relación interesante para generar particionamientos son las relaciones de equivalencia. Una *relación de equivalencia* sobre A es una relación binaria $\sim$ que cumple las propiedades de: reflexividad, simetría y transitividad. Las relaciones de equivalencia definen subconjuntos de A denominados *clases de equivalencia*. Asimismo, el conjunto de todas las clases de equivalencia se denomina *conjunto cociente*. Un ejemplo básico de relación de equivalencia es la igualdad.

Sea $\sim$ una relación de equivalencia y A el conjunto sobre el que está definida, llamaremos *clase de equivalencia* del elemento $a \in A$ al subconjunto $[a]_\sim \subseteq A$ formado por todos los elementos de A que están relacionados con a por $\sim$. Esto es: $[a]_\sim = \{x \in A : x \sim a\}$. Entonces, el conjunto cociente de A por $\sim$ es $A/\sim = \{[a]_\sim, a \in A\}$.

Otro tipo de relación es el orden entre elementos. De esta manera es posible decir si un elemento es menor, igual o mayor a otro. Los distintos tipos de órdenes son:

- Preorden. En este caso, la relación binaria cumple con la reflexividad y la transitividad. Un preorden $\leq$ induce una equivalencia $\equiv$, $a \equiv b$ si y solo si $a \leq b$ y $b \leq a$, e induce un orden (parcial) en $A/\equiv$. Un ejemplo de una relación de preorden es la relación basada en la distancia Euclidiana a un punto p, decimos que $a \leq b$ sii $d(a, p) \leq d(b, p)$. Este preorden induce la relación de equivalencia: $a \sim b$ sii $d(a, p) = d(b, p)$.
- Orden parcial. Aquellas relaciones que cumplen con: reflexividad, antisimetría y transitividad. Por ejemplo, la relación *ser divisor de*, genera un orden parcial en el conjunto.

- Orden parcial estricto. Es una relación binaria que es antireflexiva y transitiva.
- Orden total. Es un orden parcial $\leq$ donde para todo a y b vale o bien $a \leq b$ o bien $b \leq a$ (condición de totalidad). Un orden total estricto es un orden parcial estricto que cumple con esta misma condición de totalidad. Por ejemplo, la relación de orden alfabético establece un orden total en un conjunto de palabras.

2.5.2. Cadenas y Árboles (Tries)

Una estructura que se emplea en esta tesis es el *Trie* [Fre60]. Esta estructura permite agilizar la búsqueda sobre un conjunto de palabras (*cadenas*). Para su descripción usaremos los conceptos de palabras, prefijos y sufijos descritos en breve.

Una *palabra* o *cadena* es una secuencia de símbolos de la forma $S = c_1c_2c_3 \dots c_m$ donde cada c_i es un carácter de la palabra S. Un *prefijo* de la cadena S es una cadena de la forma $c_1 \dots c_k$ donde $0 < k < m$, por ejemplo un prefijo de *cadena* es "*cad*". Un sufijo de la cadena S es una cadena de la forma $c_{m-k+1} \dots c_m$, por ejemplo, un sufijo de *cadena* es "*dena*".

Un trie es un árbol de prefijos, el cual almacena cadenas sobre un alfabeto. La aridad del árbol depende del tamaño del alfabeto y la altura máxima es el largo de las cadenas. La principal característica de un trie es que almacena en un nodo aquellos elementos con un mismo prefijo [Fre60]. En la figura 2.4 mostramos un trie construido sobre un conjunto de palabras (enumeradas del 1 al 9). Note que todas las palabras cuyo prefijo es *A* (del 1 al 8) están en el subárbol izquierdo de la raíz, y descendiendo por éste, las palabras cuyo segundo carácter es una *L* se colocaron en su subárbol izquierdo del nodo etiquetado por *A* (numeradas del 1 al 5). Las hojas están indicadas por un cuadrado cuya etiqueta corresponde al número de la palabra correspondiente.

La búsqueda de una palabra sobre un trie inicia en la raíz y desciende por el primer nodo correspondiente a la primera letra de la palabra. Por ejemplo, sea "AMA" la palabra a buscar en el trie. Inicialmente, partiendo por la raíz, se tienen dos ramas, las palabras cuyo prefijo es *A* y aquellas con *D*. En este caso, se descenderá por el nodo izquierdo que corresponde a las palabras cuyo prefijo es *A*, que también es el prefijo de la palabra de consulta. A partir de ese nodo, se tienen 3 caminos, cuyos prefijos serían *AL*, *AM* y *AR*. De la misma forma, se continúa descendiendo por la rama cuyo prefijo es igual al prefijo de la cadena de consulta *AM*, y así sucesivamente. En este caso es una búsqueda infructuosa pues el prefijo *AMO* no es un prefijo de *AMA*. En los casos en que se llega a una hoja, los elementos de ésta son comparados directamente contra la consulta.

Una variante de un trie evita prolongar los caminos unarios, es decir, cuando se tiene un sólo elemento por la rama, ésta se contrae. Por ejemplo, en la figura 2.4 la rama correspondiente a la cadena *DAR* podría contraerse hasta el primer nodo con la etiqueta *D*, ahí se colocaría la hoja (en este caso, con la etiqueta 9). Este proceso evitaría crear los nodos *A* y *R* para esta rama.

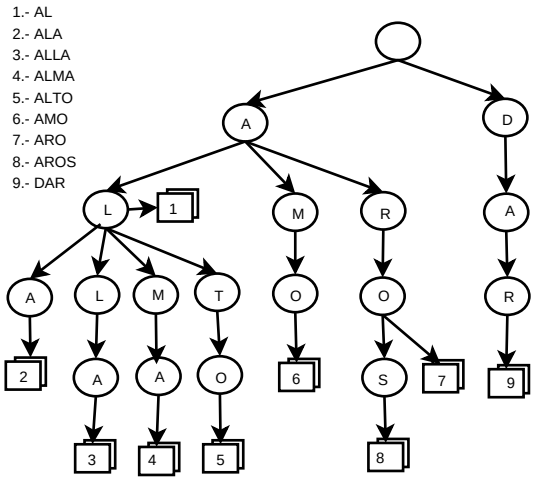


Figura 2.4: Ejemplo de un trie construido sobre un conjunto de nueve palabras.

2.6. Bases de Datos de Prueba

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

2.6.1. 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 [BBK01b]. 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 (véase la página 15).

Las bases de datos generadas fueron de tamaño 2.000 hasta 20.000. Las dimensiones tratadas varían entre 4 y 20 para algoritmos exactos y entre 128 y 1.024 para algoritmos probabilísticos. En los experimentos se promediaron 500 consultas para generar cada punto. En la figura 2.5 se muestra el histograma de distancias para varias dimensiones (8, 32, 128 y 1024).

2.6.2. 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 [Lev66]. Esta distancia mide

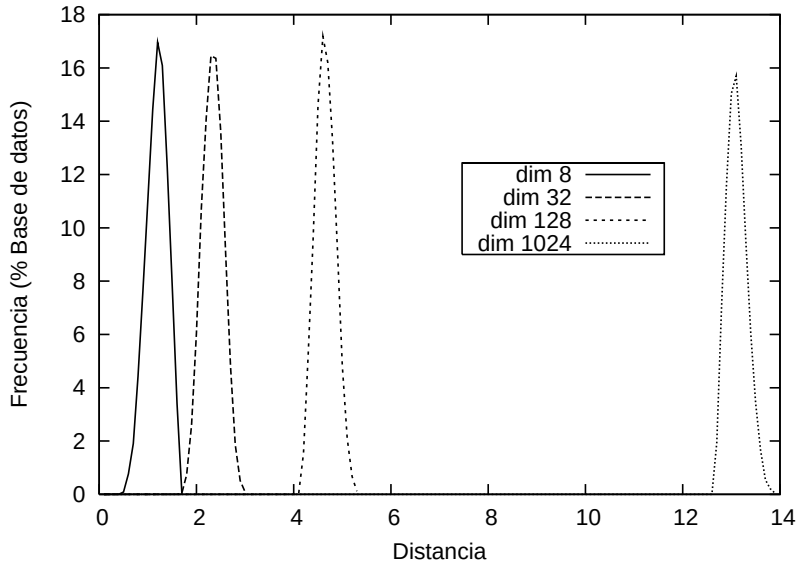


Figura 2.5: Histograma de distancias de bases de datos en el cubo unitario en distintas dimensiones. 2.000 Objetos.

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_1c_1 representa la cadena s_1 concatenada con c_1. Los símbolos c_1 y/o c_2 pueden también representar la cadena vacía.

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

La dimensión intrínseca de esta base de datos es de 12 y en la figura 2.6 puede verse su histograma de distancias. Este fue calculado sobre un conjunto de pares seleccionados de manera aleatoria.

2.6.3. Documentos

Una base de datos real e interesante de analizar es un conjunto de documentos. Una aplicación para este tipo de bases de datos en general es en la recuperación de información, donde cada documento es modelado como un punto en un espacio métrico. El conjunto de palabras que se proporcionan durante la búsqueda es visto también como un documento. Por lo tanto, la búsqueda por similaridad en este tipo de espacios consiste en encontrar documentos relevantes. Éstos son más relevantes en la medida que comparten más términos con la consulta.

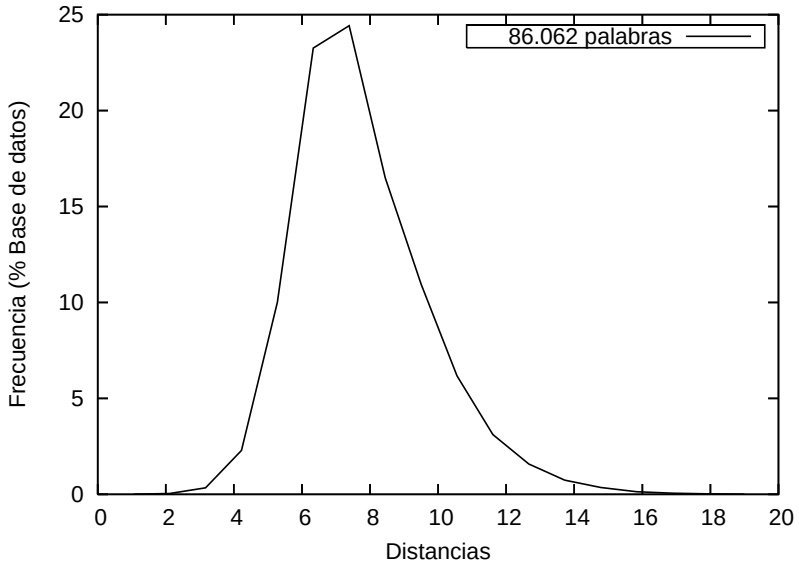


Figura 2.6: Histograma de distancias del diccionario de palabras en español.

Una distancia típica entre documentos consiste en modelar cada palabra de un documento como un término [1] [BYRN99]. De esta forma, sea $\{t_1, \ldots, t_k\}$ el conjunto de términos y $\{d_1, \ldots, d_N\}$ el de documentos. Un documento d_i se modela como un vector:

$$\overrightarrow{d_i} = (w(t_1, d_i), \ldots, w(t_k, d_i))$$

donde $w(t_r, d_i)$ es el peso del término t_r en el documento d_i.

Hay varias fórmulas para calcular los pesos, una de las más populares es:

$$w(t_r, d_i) = w_{r,i} = \frac{\phi_{r,i} \cdot \varphi_r}{|\overrightarrow{d_i}|} = \frac{\phi_{r,i} \log \frac{N}{N_r}}{\sqrt{\sum_{s=1}^{k} \left(\phi_{s,i} \log \frac{N}{N_s}\right)^2}}$$

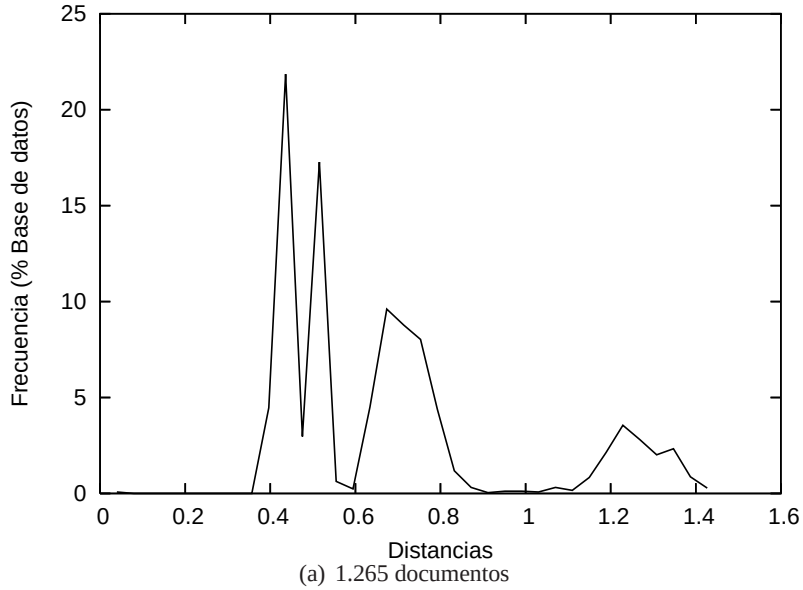
donde $\phi_{r,i}$ (frecuencia del término) es la cantidad de veces que t_r aparece en d_i y N_r es la cantidad de documentos donde aparece t_r. Por otro lado φ_r es el logaritmo del inverso de la frecuencia de documentos en los que aparece el término t_r. Con esto lo que se intenta medir es qué tan importante es un término en un documento. La distancia entre dos documentos es el arco coseno del ángulo entre sus vectores, es decir, el arco coseno del producto interno entre vectores [BYRN99].

$$d(d_i, d_j) = \arccos(\overrightarrow{d_i} \cdot \overrightarrow{d_j}) = \arccos\left(\sum_{r=1}^{k} w_{r,i} \cdot w_{r,j}\right)$$

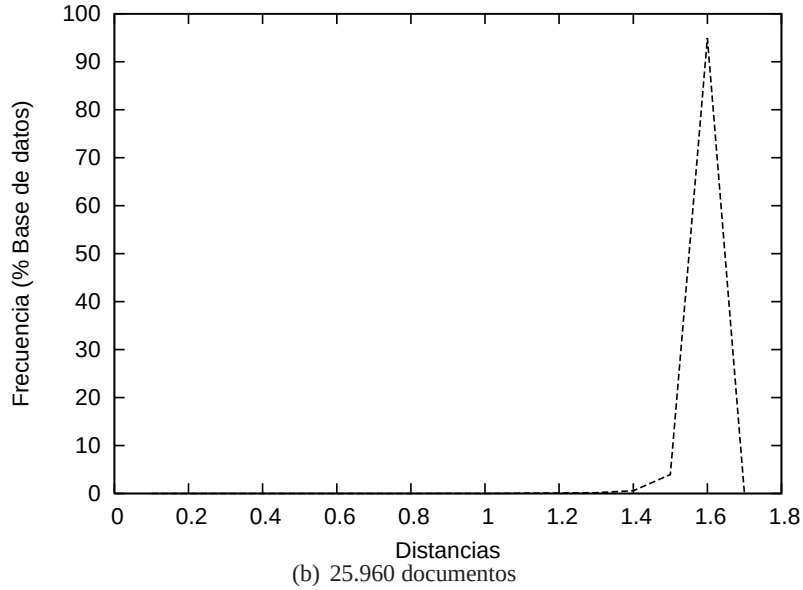
[1] Excepto *stopwords* o palabras vacías, por ejemplo, los artículos: *la*, *el*, etc.

En las pruebas realizadas en esta tesis se utilizaron dos colecciones del TREC-3 [Har95], una con 25.960 documentos y otra con 1.265 documentos. Ambas bases de datos usan los mismos documentos, la diferencia es el tamaño de cada documento: en la primera son colecciones de 1MB aproximadamente, mientras que en la segunda son de 1KB.

En la figura 2.7 se muestran los histogramas de distancias de estas bases de datos. Para el caso de los 25.960 documentos el valor de la dimensión intrínseca fue 800 y para el de 1.265 fue de 3,65.



(a) 1.265 documentos



(b) 25.960 documentos

Figura 2.7: Histograma de distancias de las dos bases de datos de documentos.

Capítulo 3

Estado del arte

En este capítulo describiremos los algoritmos existentes para la búsqueda por proximidad en espacios métricos.

Las técnicas existentes emplean índices durante las consultas con el propósito de reducir el costo de la búsqueda (generalmente evitando comparaciones de distancia). 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.

En esta revisión del estado del arte mostraremos cómo trabajan los índices existentes y mostraremos que toda la investigación realizada en este tema puede verse como una búsqueda de qué información de distancias almacenar dado cierto espacio de memoria disponible.

Básicamente el estado de arte se divide en dos familias de algoritmos [CNBYM01, HS03]: los basados en pivotes y los basados en particiones compactas. Primero se mostrarán las bases de cada familia y posteriormente se analizarán los algoritmos.

3.1. Clasificación de Algoritmos

3.1.1. Algoritmos Basados en Pivotes

La idea general para construir un índice basado en pivotes consiste en elegir un conjunto $\mathbb{P} = \{p_1, p_2, \ldots, p_k\} \subseteq \mathbb{U}$, 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 $\{d(p_1,u), d(p_2,u), \ldots, d(p_k,u)\}, \forall u \in \mathbb{U}$, 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

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

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 $\exists p \in \mathbb{P}$, $|d(q,p) - d(p,u)| > r$, pues es claro que si $|d(q,p) - d(p,u)| > r$, entonces $d(q,u) > r$. Los elementos no descartados se comparan directamente contra la consulta (comparaciones externas). El algoritmo 1 muestra el proceso realizado durante el indexación, y el algoritmo 2 muestra el proceso de una consulta en esta familia de índices.

En la figura 3.1 se muestra un ejemplo del uso de esta técnica, donde p es un pivote. El anillo formado por los círculos centrados en p, a distancias $d(p,q) + r$ y $d(p,q) - r$, contiene a los elementos que no podrían ser descartados por p, estos son: $u_2, u_3, u_4, u_5, u_6, u_7$. El resto de la base de datos sí es descartada, por ejemplo, en el caso de u_1, se cumple que $|d(p,q) - d(p,u_1)| > r$, lo mismo sucede con el resto.

En [BY97, BYN98] y recientemente en [MSMO02] muestran que existe un número óptimo de pivotes que depende no sólo de dimensión del espacio sino también del número de

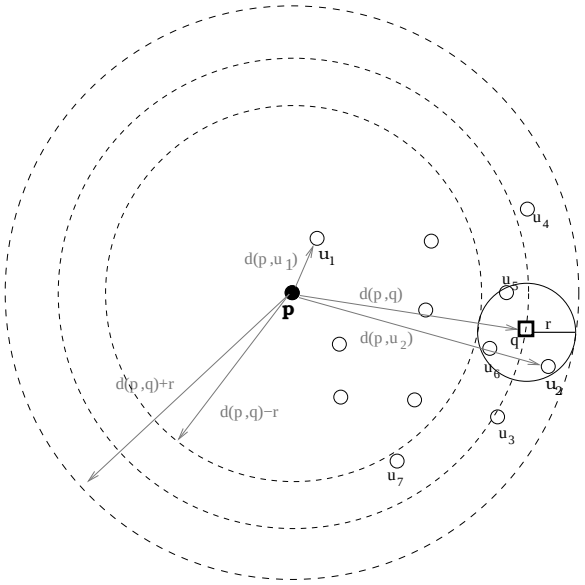


Figura 3.1: Ejemplo del uso de un algoritmo basado en pivotes para una consulta de rango (q,r).

Algoritmo 1 Algoritmo-pivotero-indexado

```
1: M matriz de distancias, tamaño n×k
2: for all u ∈ 𝕌 do
3:    for all p ∈ ℙ do
4:       M[u,p] ← d(u,p)
5:    end for
6: end for
7: Guardar M
```

Algoritmo 2 Algoritmo-pivotero-consulta

```
1: M matriz de distancias, tamaño n×k
2: 𝔽 ← 𝕌
3: for all p ∈ ℙ do
4:    d ← d(p,q)          // comparaciones internas
5:    for all u ∈ 𝔽 do
6:       if |M[u,p] − d| > r then
7:          𝔽 ← 𝔽 − u          // Objetos filtrados
8:       end if
9:    end for
10: end for
11: for all u ∈ 𝔽 do
12:                          // Lista de candidatos, comparaciones externas
13:    if d(q,u) ≤ r then
14:       Reportar u          // Relevantes
15:    end if
16: end for
```

vecinos más cercanos (K) pues a medida que K aumenta, las comparaciones de distancia (externas) también aumentan.

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 $p_1, \ldots, p_k \subseteq \mathbb{U}$ y dividir el espacio, es decir, distribuir los demás elementos entre las k zonas pertenecientes a cada p_i. Estos p_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 p_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 $u, p \in \mathbb{U}, q \in \mathbb{X}$, si p 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* $\max\{d(q,p) - r_c, 0\} \leq d(q,u) \leq d(q,p) + r_c$.

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

$$\begin{aligned} d(p,q) &\leq d(p,u) + d(u,q)\,, \\ d(p,u) &\leq r_c. \end{aligned}$$

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

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

Durante una consulta por rango $(q,r)_d$, esta familia de algoritmos descarta aquellas zonas de centro p tales que $d(p,q) - r_c > r$, pues usando el lema 2 sabemos que cualquier elemento u en esa zona cumple con $d(q,u) > r$. En las zonas no descartadas se repite el proceso hasta que se llega a zonas sin divisiones. En éstas los elementos deben revisarse secuencialmente.

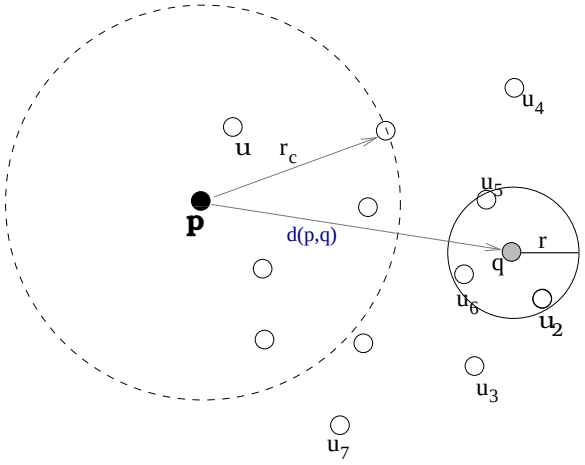


Figura 3.2: Ejemplo del uso de un algoritmo basado en particiones compactas. El tipo de consulta es de rango (q,r).

En la figura 3.2 se muestra un ejemplo del uso de esta técnica. El radio de cobertura del centro p está representado por r_c. En la figura toda la zona del centro p puede ser descartada puesto que todos los elementos en ella cumplen con $d(q,p) - r_c \leq d(q,u)$ y $d(q,p) - r_c > r$.

En una búsqueda usando el criterio de los hiperplanos, es posible establecer una cota para $d(q,u)$.

Lema 3 *Sea $u \in \mathbb{U}$ un objeto más cercano a p_1 que a p_2 o que equidista, es decir, $d(p_1,u) \leq d(p_2,u)$. Dados $d(q,p_1)$ y $d(q,p_2)$, podemos establecer la cota* $\max\left\{\frac{d(q,p_1)-d(q,p_2)}{2}, 0\right\} \leq d(q,u)$.

Demostración De la desigualdad triangular sabemos que

$$d(q,p_1) \quad \leq \quad d(q,u) + d(p_1,u)\,,$$

lo cual lleva a

$$d(q,p_1) - d(q,u) \quad \leq \quad d(p_1,u).$$

Del mismo modo tenemos que

$$d(p_2, u) \quad \leq \quad d(q, p_2) + d(q, u)\,,$$

y por hipótesis

$$d(p_1, u) \quad \leq \quad d(p_2, u).$$

Combinando en las ecuaciones anteriores obtenemos

$$\begin{aligned} d(q, p_1) - d(q, u) \;\leq\; d(p_1, u) \leq\; d(p_2, u) \;\leq\; d(q, p_2) + d(q, u)\,, \\ d(q, p_1) - d(q, p_2) \qquad\qquad \leq \qquad\qquad 2d(q, u). \end{aligned}$$

Finalmente, sabemos que $d(q, u) \geq 0$, así la cota inferior es

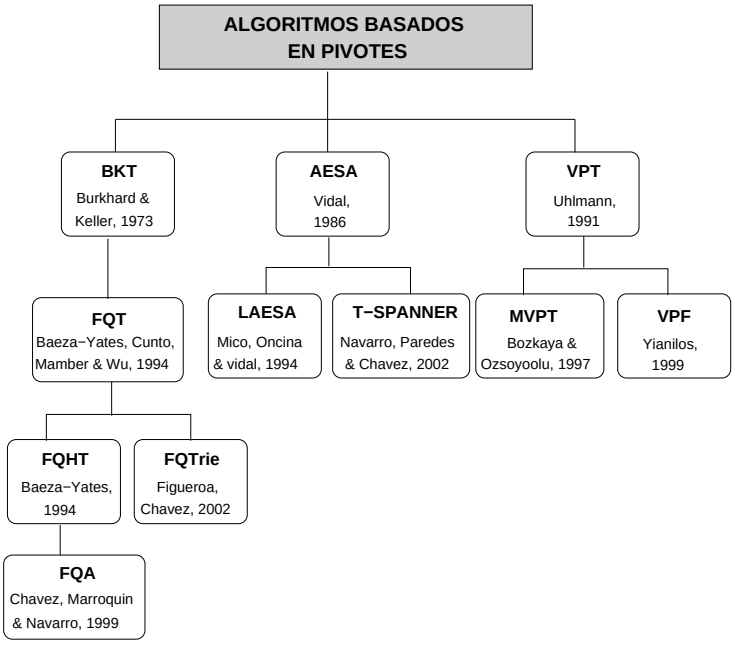
$$\text{máx}\left\{ \frac{d(q, p_1) - d(q, p_2)}{2},\, 0 \right\} \quad \leq \quad d(q, u). \qquad \blacksquare$$

Usando sólo el criterio de hiperplanos no es posible determinar una cota superior, porque los objetos pueden estar muy cerca o muy lejos de p_1 y p_2. En particular, una zona p_j debe ser revisada si $\frac{d(q,p_j)-d(q,p_i)}{2} \leq r$, y q pertenece a la zona de p_i. El mismo cálculo se repite para decidir visitar o descartar las otras zonas de la base de datos.

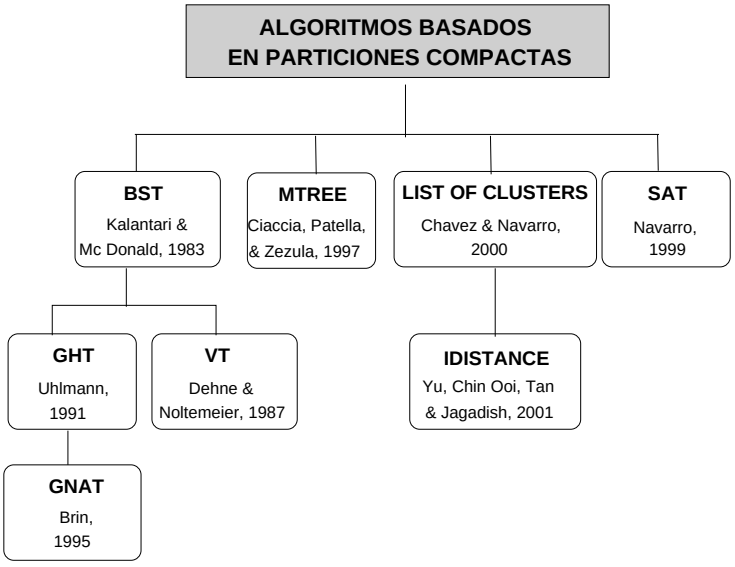
3.2. Descripción de los Algoritmos

A continuación 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 [CNBYM01]. Para ver una división por familia véase la figura 3.3, y por fecha de aparición en la figura 3.11 (página 41).

- *AESA* Approximating Eliminating Search Algorithm [Vid86]. Este algoritmo almacena todas las $n^2/2$ 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



(a) Algoritmos pivoteros



(b) Algoritmos basados en particiones compactas

Figura 3.3: Esquema de las familias de los algoritmos para la búsqueda por proximidad en espacios métricos.

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.

Un ejemplo de AESA se muestra en la figura 3.4. El primer pivote seleccionado fue u_{11}, en la segunda iteración el pivote puede ser cualquiera del resto de los objetos sin contar a u_2, u_9 y u_8. Esto es una ventaja porque un pivote cercano a q aproxima mejor las distancias de q a la base de datos; el ejemplo más claro de esto es cuando q mismo es un pivote.

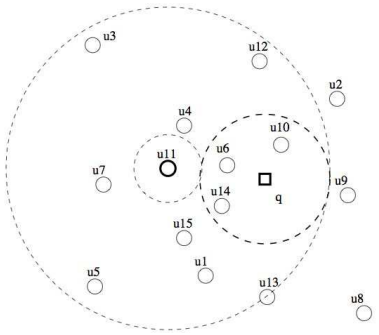


Figura 3.4: Ejemplo de la primera iteración de AESA. Los elementos entre los dos anillos serán los candidatos en la segunda iteración.

- *t-Spanner* [Par02, NPC02]. 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 $d(u,v) \leq D_G(u,v) \leq t \cdot d(u,v)\ \forall u,v \in \mathbb{U}$, donde $t > 1$ es un parámetro de construcción. El índice intenta almacenar las mínimas distancias necesarias para garantizar la condición sobre t. El espacio utilizado por el índice es empíricamente $O\left(n^{\left(1+\frac{0,1...0,2}{t-1}\right)}\right)$. Cada vértice guarda información de las aristas y los elementos conectados a él. Para responder una consulta, este algoritmo usa el criterio de AESA para decidir qué vértice seguir. Este algoritmo puede verse como una simulación de AESA usando menos espacio.

- *GNAT* Geometric Near-neighbor Access Tree [Bri95]. En un GNAT de aridad m, se seleccionan m centros $p_1, \ldots, p_m$, y se define $U_i = \{u \in \mathbb{U} \mid d(p_i,u) < d(p_j,u), \forall j \neq i\}$. 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

se almacena para cada nodo una tabla $rango_{i,j} = [\text{mín}_{u \in \mathbb{U}_j}(p_i, u), \text{máx}_{u \in \mathbb{U}_j}(p_i, u)]$, 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 $d(q, p_i) \pm r$ 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 recursivamente en los subárboles no descartados. En la figura 3.5 se muestra un ejemplo de este algoritmo en su primer nivel. Los autores muestran buenos resultados en aridades grandes (50 y 100).

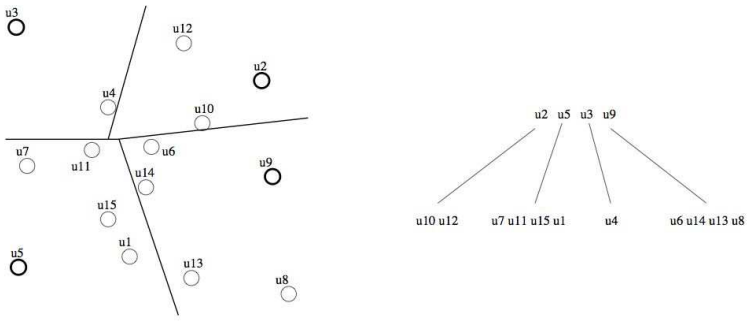


Figura 3.5: Ejemplo del primer nivel de un GNAT con $m = 4$.

- *LAESA* Linear AESA [MOV94]. 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)* [MOC96, TYM06], el cual resuelve las consultas con un costo sublineal aunque duplica, en promedio, los cálculos de distancia. En [MSMO02] los autores muestran una extensión de LAESA para encontrar los K vecinos más cercanos (K-LAESA).

- *FQT* Fixed-Queries Tree, *FHQT* Fixed Height FQT [BYCMW94] y FQTrie [CF04]. 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.

Nótese que la idea de los FQT es esencialmente una tabla con h pivotes tipo LAESA, donde se compara q contra todos los pivotes y el árbol se usa para reducir el tiempo extra de CPU necesario para encontrar la lista de candidatos. Un ejemplo puede verse en la figura 3.6.

En el FHQT todas las hojas tienen la misma profundidad h. Cada pivote almacena todas las distancias hacia los elementos de la base de datos. El espacio ocupado se encuentra entre $O(n)$ y $O(nh)$. En la práctica el h óptimo es $O(\log n)$, el cual rara vez puede ser alcanzado por limitaciones de espacio en bases de datos grandes. Otro índice que usa esta misma estructura es el FQTrie [CF04]. Los autores proponen el uso de un trie y una tabla de búsqueda para agilizar el proceso.

- *FQA* Fixed-Queries Array [CMN99, CMN01]. 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.

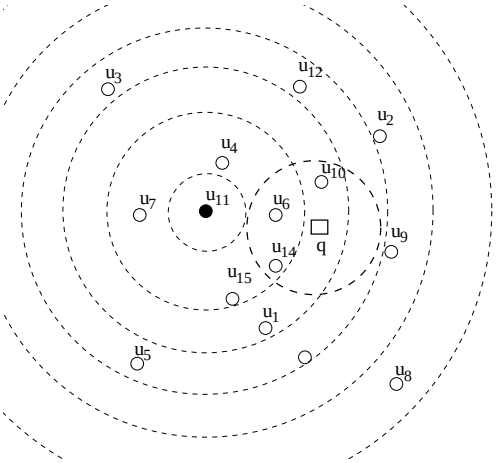
  Como resultado el FQA está almacenado en un número de hb bits. La ventaja de este índice es que usa menos memoria, $O(nhb)$ bits, para obtener los mismos resultados que el FHQT.

  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 3.6 se muestra una comparación entre estos índices. En el caso de FQA se podría perder precisión por el manejo de bits.

- *BKT* Burkhard-Keller Tree [BK73]. 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.



(a) Base de datos

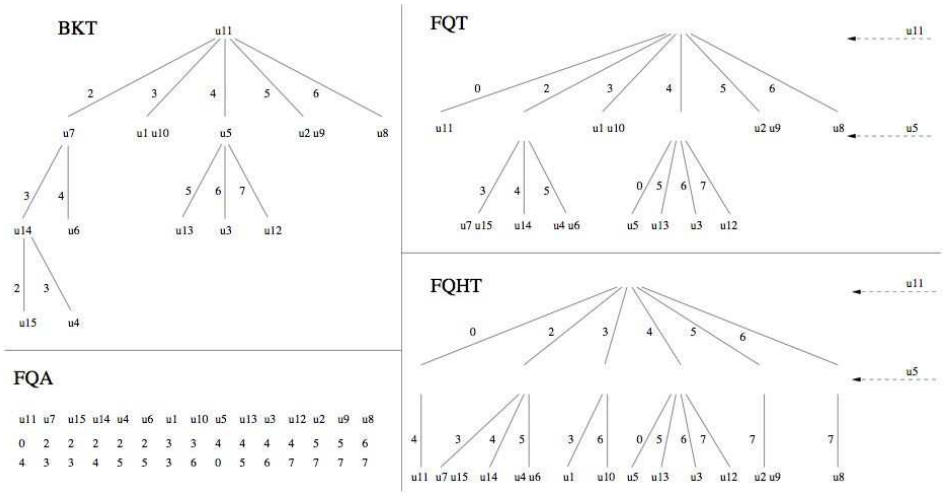


Figura 3.6: Ejemplo comparativo entre los índices BKT, FQT, FHQT y FQA para el conjunto de puntos mostrado en la parte superior.

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 $i \notin [d(q,p)-r, d(q,p)+r]$. Se desciende por las ramas que no fueron descartadas aplicando el mismo proceso. En las figuras 3.6 y 3.7 se muestra un ejemplo de este índice. En la figura 3.6 se puede apreciar la diferencia con respecto a la familia de los FQT.

Nótese que a medida que la dimensión del espacio crece la mayoría de los elementos se concentran en un misma rama (pues la varianza disminuye y la media aumenta). Este fenómeno hace que el árbol degenere en una lista enlazada y la búsqueda en un proceso casi secuencial.

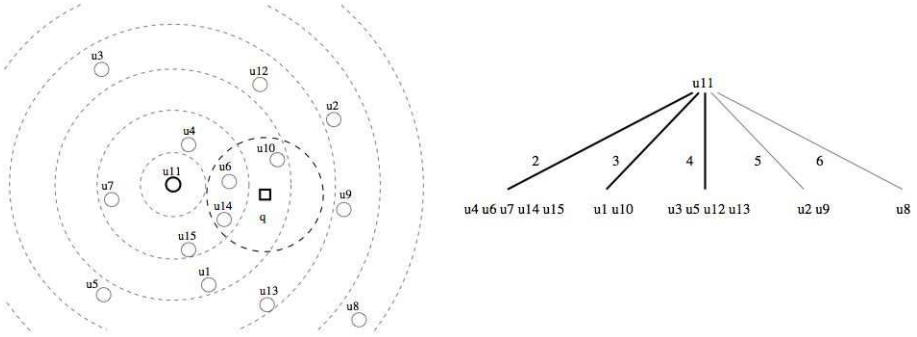


Figura 3.7: En el lado izquierdo, se muestra a u_{11} como pivote y la división del espacio creado por éste. En el lado derecho, se muestra el primer nivel del BKT con u_{11} como raíz. Para la consulta q, las ramas que se seguirían son $2,3,4$, que corresponden a los anillos intersectados en el lado izquierdo por el radio de la consulta.

- *VPT* Vantage Point Tree [Uhl91]. 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 del conjunto de todas las distancias en el subárbol. Inicialmente, sea p el elemento en el nodo raíz, $M = mediana\{u \in \mathbb{U} | \, d(p,u)\}$, los elementos $d(p,u) \leq M$ serán insertados en el subárbol izquierdo, el resto en el subárbol derecho. En esencia este índice es como un BKT donde sólo existen dos etiquetas $i \leq M$ e $i > M$. El resultado de seleccionar la mediana como criterio de división es un árbol balanceado. El espacio que ocupa este índice es $O(n)$.

  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 3.8 se muestra un ejemplo de la construcción de este índice.

  En espacios de dimensión alta, la gran mayoría de los elementos, así como la consulta, se encontrarán muy cerca de la mediana, por lo que será muy probable descender por ambas ramas de esta estructura.

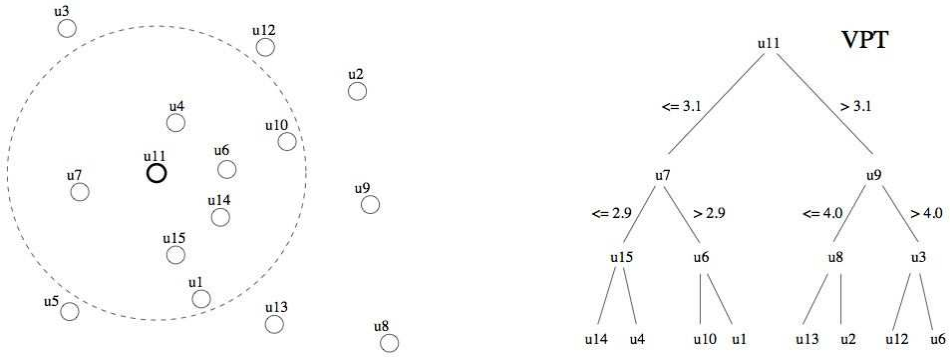


Figura 3.8: Ejemplo del algoritmo VPT cuya raíz es u_{11}. El círculo en el lado izquierdo representa el radio M usado para construir el árbol.

- *MVPT* Multi Vantage-Point Tree [BO97]. 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 [Yia99a]. 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 (véase la figura 2.2, página 17). 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 [CPZ97]. 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 [Gut84].

Cada elemento almacena la distancia a su padre, lo que hace que este algoritmo use $O(n)$ espacio. El criterio de descarte de ramas es el del radio de cobertura.

- *LC* List of Clusters [CN00, CN05], *iDistance* Indexing the Distance[YOTJ01]. Estos índices tienen muy buen desempeño en dimensiones altas. El algoritmo de construcción (en ambos) consiste en seleccionar un centro p 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 $O(n)$.

  La diferencia entre estos dos algoritmos es que LC recorre la lista secuencialmente mientras que *iDistance* asocia una "llave" y a cada elemento dependiendo de la zona (i) en que fue agrupado; y es calculada como: $y = i \times c + d(u, p_i)$, donde c es una constante (grande) que trata de evitar el traslape entre llaves. Finalmente, las llaves son indexadas con un B^+-tree para su localización. Este índice muestra mejores tiempos de respuesta que el M-Tree, pero empeora cuando se recuperan muchos vecinos.

  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 [KM83].- Es un árbol binario que se construye recursivamente. Para cada nodo del árbol se eligen dos centros p_1 y p_2. Los objetos más cercanos a p_1 que a p_2 se insertan en el subárbol izquierdo, y los objetos más cercanos a p_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_{c_1} y r_{c_2}. La memoria utilizada es $O(n)$, pues cada elemento almacena información de la cercanía a su centro.

  Durante una búsqueda del objeto q, se pueden excluir todos aquellos subárboles donde se cumpla que $d(q, p_i) - r_{c_i} > r$ (criterio de radio de cobertura, lema 2). Un ejemplo de la primera división de este índice está en la figura 3.9. Note que para esa consulta q deben revisarse ambos subárboles pues el radio de la consulta intersecta ambas regiones.

- *GHT* Generalized-Hyperplane Tree [Uhl91].- 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 $O(n)$. El criterio para recorrer el subárbol izquierdo es $d(q, p_1) - r < d(q, p_2) + r$ y para entrar al subárbol derecho es $d(q, p_2) - r \leq d(q, p_1) + r$. Un ejemplo de este algoritmo se muestra en la figura 3.9.

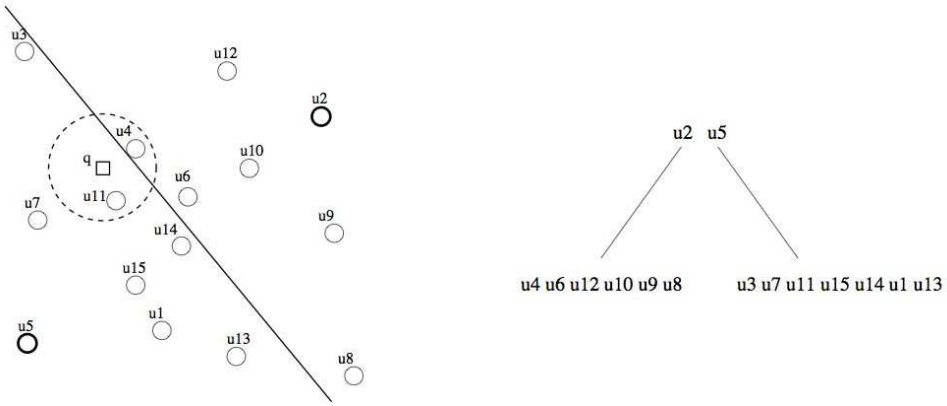


Figura 3.9: Ejemplo del primer nivel del BST o GHT. En este caso, note que para la consulta q deben revisarse ambos subárboles.

- *VT* Voronoi Tree [DN87] .- Este algoritmo es una mejora al BST. El *VT* tiene 2 ó 3 elementos (e hijos) por nodo. Cuando se crea un nuevo nodo, se insertará en él 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 [Nav99, Nav02a]. 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. El criterio de descarte son los hiperplanos y el radio de cobertura (lemas 2 y 3). Un ejemplo puede verse en la figura 3.10.

3.3. Comparación del Estado del arte

Nótese 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 de-

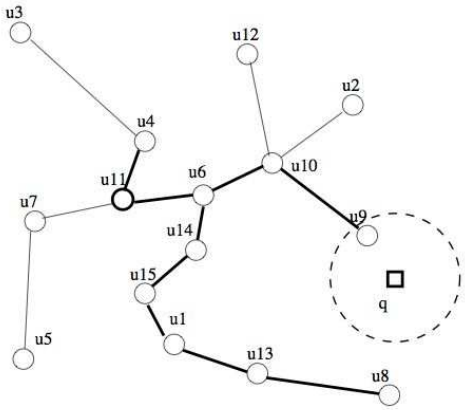


Figura 3.10: Ejemplo de un SAT. El recorrido que se sigue para esa consulta se muestra en la figura con la línea más oscura, iniciando por u_{11}.

sempeño de AESA pero almacenando mucha menos memoria. La manera de usar menos memoria es almacenando sólo unas columnas de esa matriz (el caso de los algoritmos pivoteros), algunas áreas ó manteniendo cotas inferiores y/o superiores (los basados en particiones compactas). Otra comparación interesante 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 [BY97, BYN98] 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. Algunas técnicas de optimización son: eliminar información redundante, compactar la información guardada, etc.

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.

La tabla 3.1 resume las principales características de los algoritmos descritos anteriormente. En la primera y tercera columna se tiene el nombre del algoritmo y el espacio ocupado por su índice.

La cuarta columna describe cómo se seleccionan los Objetos Representantes (OR, como pivotes, centros, etc.) durante la fase de construcción del índice. Dicha selección se generaliza como sigue:

- ***OR Fijos***. Algunos índices primero escogen los OR y en base a ellos se construye la estructura de su índice.

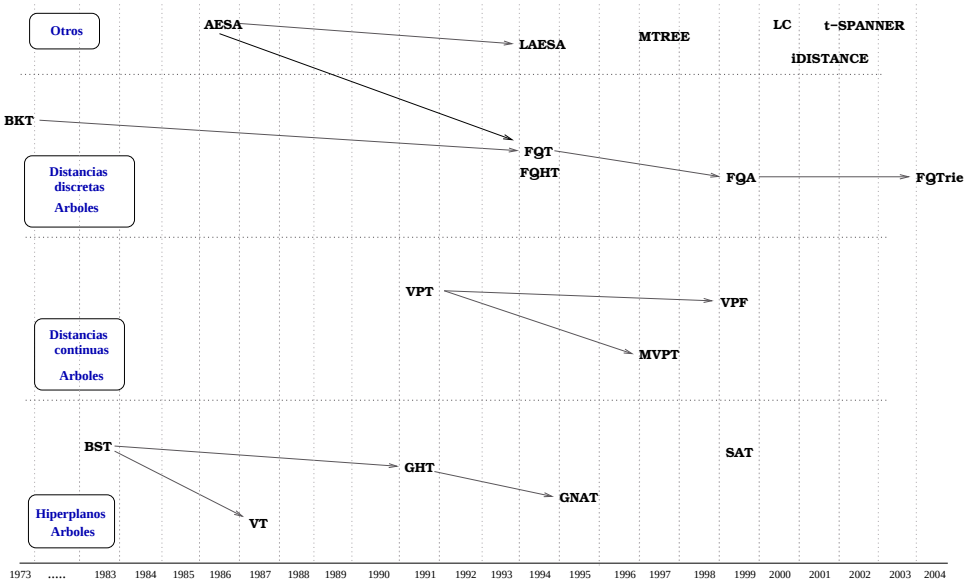


Figura 3.11: Estado de arte de los algoritmos para búsqueda en espacios métricos a través del tiempo.

- ***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.

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. Nótese que 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.

En la quinta columna de la tabla 3.1 se indica si el índice se mantiene balanceado en dimensiones altas, pues es una condición que influye en el rendimiento del algoritmo. Al balancear una estructura se busca que, en el mejor de los casos, sea posible eliminar al menos la mitad de los elementos en la búsqueda. Sin embargo, en dimensiones altas se tiene mayor probabilidad de descender por ambas (o más) ramas (o divisiones), dependiendo de la consulta y de su radio. Note que LC y VPF no tienen una estructura balanceada en dimensiones altas, pero esta característica es una ventaja en dimensión alta debido a su estructura.

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.

El resto de las columnas tiene información citada por los autores [CNBYM01] sobre el desempeño de su respectivo algoritmo.

Algunos algoritmos presentados pueden mejorar su rendimiento usando más memoria. El caso límite es AESA pues requiere toda la información de las distancias disponible para la búsqueda. En la figura 3.12(a) se muestra el desempeño de algunos de los algoritmos para la búsqueda por proximidad en dimensión 8, esto es, los cálculos de distancia necesarios para responder una consulta contra el espacio ocupado por el índice. El espacio tratado fue un conjunto sintético de distribución uniforme (descrito en la sección 2.6.1) con 100.000 elementos. Las consultas recuperan el 0,01 % de la base de datos. Los casos en que es posible emplear más espacio en el índice y tener un mejor rendimiento, se muestran con una línea (LAESA, GNAT y FQA).

En la figura 3.12(b) se refleja el efecto que provoca el aumento de la dimensión del espacio y que es conocido como "la maldición de la dimensionalidad" (explicada en la sección 2.3). En general, cuando la dimensión aumenta todos los algoritmos para la búsqueda por proximidad tienden a hacer búsqueda secuencial, como se muestra en la figura. El extremo derecho del mapa corresponde a hacer búsqueda secuencial.

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.

3.4. Algoritmos Inexactos

Los algoritmos presentados en la sección anterior fueron algoritmos exactos, los cuales recuperan correctamente los elementos de la base de datos que se encuentran en la bola $(q,r)_d$. A continuación 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 [VRCB88, VL88, TYM06]. 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 cuán 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 se mueve en la dirección contraria. De esta manera, la precisión está perfectamente

Algoritmo	Familia	Espacio ocupado	Criterio de selección de pivotes o centros	Balanceo en dim. alta	Complejidad de la consulta según los autores	Tiempo extra de CPU
AESA	Pivotes	n^2 dist	*Todos OR*	-	$O(1)$[d]	$n \dots n^2$
t-spanner	Pivotes	$m = \left(n^{(1+\frac{0,1 \dots 0,2}{t-1})}\right)$ ptrs	*Todos OR*	-	no analizado	$m \log n \dots nm \log n$
GNAT	Part. compactas	nm^2 dist	*De cada subconj.*	Si	no analizado	-
LAESA	Pivotes	kn dist	*Fijos*	-	$k + O(1)$[d]	$n \dots kn$
FQT	Pivotes	$n \dots nh$ ptrs	*Fijos*	No	$O(n^\alpha)$	-
FHQT	Pivotes	$n \dots nh$ ptrs	*Fijos*	No	$O(\log n)$[b]	-
FQA	Pivotes	$nb \dots nhb$ bits	*Fijos*	No	$O(\log n)$[b]	$(\log n) *$ distancias
BKT	Pivotes	n ptrs	*De cada subconj.*	No	$O(n^\alpha)$	-
VPT	Pivotes	n ptrs	*De cada subconj.*	Si	$O(\log n)$[c]	-
MVPT	Pivotes	n ptrs	*De cada subconj.*	Si	$O(\log n)$[c]	-
MT	Part. compactas	n ptrs	*De cada subconj.*	Si	no analizado	-
VPF	Pivotes	n ptrs	*De cada subconj.*	No	$O(n^{1-\alpha} \log n)$	-
LC	Part. compactas	n ptrs	*De cada subconj.*	No	no analizado	-
BST	Part. compactas	n ptrs	*De cada subconj.*	Si	no analizado	-
GHT	Part. compactas	n ptrs	*De cada subconj.*	Si	no analizado	-
VT	Part. compactas	n ptrs	*De cada subconj.*	Si	no analizado	-
SAT	Part. compactas	n ptrs	*De cada subconj.*	Si	$O(n^{1-\Theta(1/\log \log n)})$	-

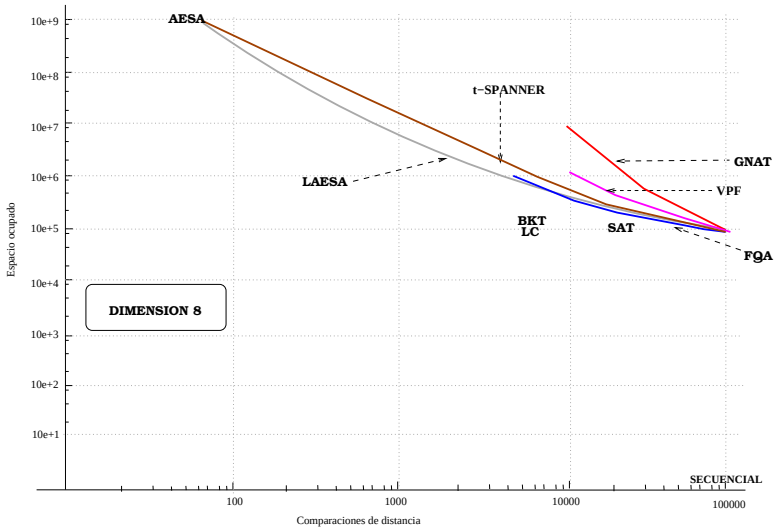
Cuadro 3.1: Comparación de los algoritmos para la búsqueda por proximidad en espacios métricos.

[a] Las complejidades sólo toman en cuenta el valor de n, no están incluídos otros parámetros como la dimensión. El espacio ocupado descrito es el máximo almacenado. Se empleó *ptrs* para "punteros" y *dist* para "distancias". α es un número entre 0 y 1, diferente para cada estructura.

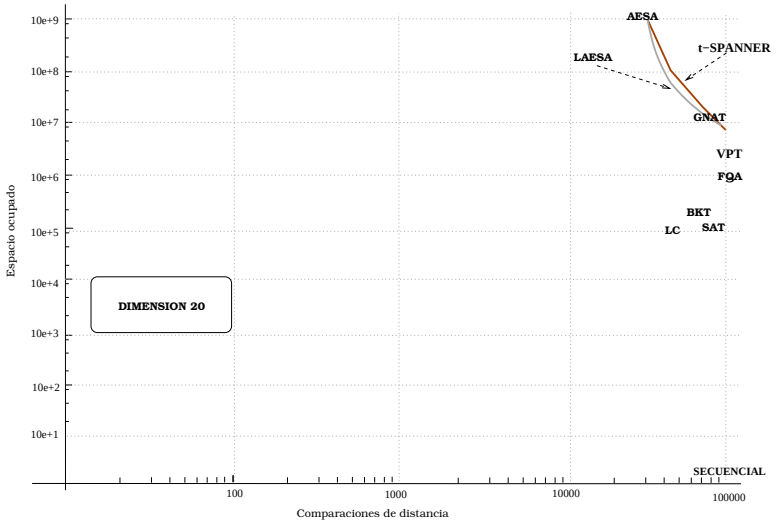
[b] Si $h = \log n$

[c] Sólo válido para radios de búsqueda muy pequeños.

[d] Conclusiones empíricas sin análisis.



(a) Dimensión 8



(b) Dimensión 20

Figura 3.12: Estado de arte de los algoritmos para búsqueda en espacios métricos, para dos dimensiones (8 y 20). El gráfico compara espacio ocupado por los índices contra cálculos de distancia promedio para responder una consulta (escala logarítmica).

controlada.

Esta alternativa para la búsqueda por similaridad exacta es llamada búsqueda por similaridad inexacta, y abarca los algoritmos aproximados y probabilísticos. En [WJ96] se presentan algunos algoritmos inexactos para la búsqueda por similaridad.

- *BBD-Tree* [AMN+94]. Este es un algoritmo aproximado para búsqueda del vecino más cercano en un espacio vectorial usando la métrica L_p. La estructura propuesta está inspirada en el *KD-Tree* para encontrar "$(1+\epsilon)$ vecinos más cercanos": en lugar de encontrar un u tal que $d(u,q) \leq d(v,q)$, $\forall v \in \mathbb{U}$, los autores encuentran un elemento u^*, tal que $d(u^*,q) \leq (1+\epsilon)\ d(v,q)\ \forall v \in \mathbb{U}$.

  La idea esencial de este algoritmo es localizar la consulta q en una celda (cada hoja en el árbol está asociada con una celda). Cada punto dentro de la celda es procesada para obtener el vecino más cercano (u) en la celda. La búsqueda se detiene cuando no hay más celdas promisorias, esto es, cuando el radio de alguna bola centrada en q y su intersección a alguna celda no vacía excede el radio $d(q,p)/(1+\epsilon)$. El tiempo de consulta es $O(\lceil 1+6k/\epsilon \rceil^k k \log n)$.

- *Aggressive Pruning* [Yia99b]. 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.

- $M(\mathbb{U}, Q)$ [Cla99]. 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 $O(\alpha n \rho)$ espacio y $O(\alpha \rho \log n)$ tiempo de búsqueda. Aquí ρ es el logaritmo del radio entre los pares de puntos más alejados y más cercanos en la unión de $\mathbb{U}$ y del conjunto de entrenamiento.

- *Streching* [CN01]. Este algoritmo está basado en "reducir" la desigualdad triángular. La propuesta es general, aunque los autores la mostraron para los algoritmos basados en pivotes.

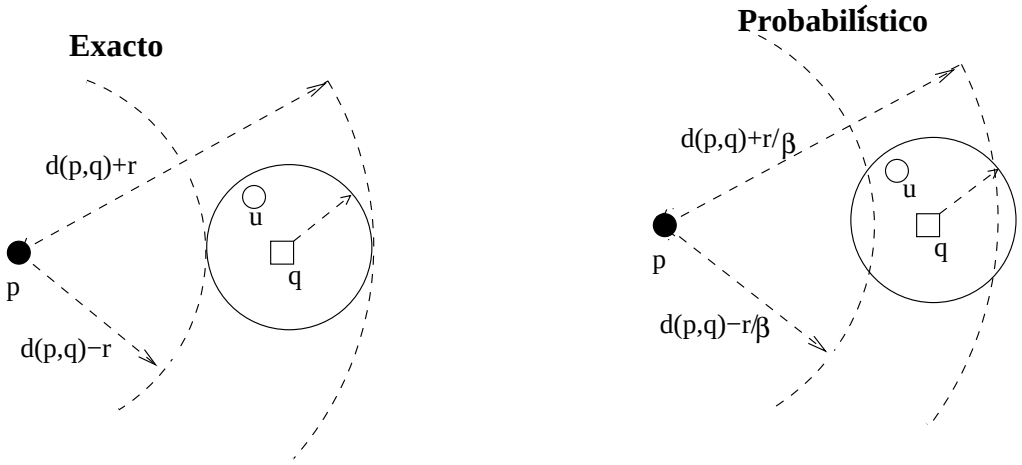


Figura 3.13: Diferencia entre un algoritmo exacto y uno aproximado, ambos desarrollados para un algoritmo basado en pivotes.

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). Aún con una pequeña reducción, se obtiene una gran mejora en el tiempo de búsqueda con una baja probabilidad de error. El factor β puede ser seleccionado en tiempo de búsqueda, así el índice puede ser construido de antemano y después es posible decidir el nivel de exactitud deseado y velocidad del algoritmo. Como el factor es usado sólo para descartar elementos, sólo los elementos a distancia mayor que r/β de q se pueden perder durante la búsqueda. En la práctica, todos los elementos que satisfacen $|d(p_i,u) - d(p_i,q)| > r/\beta$ para algún p_i son descartados. La figura 3.13 muestra la idea de este algoritmo.

- Algoritmo Probabilístico [BN02]. 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* [MSMO03]. 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. El

orden lo determina la distancia entre la consulta q y un elemento u de la base de datos, $L_\infty = \max_{p \in \mathbb{P}} |d(p,u) - d(p,q)|$. 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, note que no depende del valor de K. Esto último hace que la respuesta sea aproximada, aunque su precisión sí está relacionada con el valor de K, pues empeora cuando K aumenta.

- *TLAESA Aproximado* [TYM06]. 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 $L_\infty > \alpha \cdot d_{min}$, 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. Este algoritmo también fue probado para la búsqueda de los K vecinos más cercanos.

Resumiendo, existen varias propuestas para la búsqueda inexacta:

- Aproximada. En este tipo de búsquedas el usuario define un valor $0 < \epsilon \leq 1$ 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.
- Probabílistica. Otra forma de búsqueda inexacta son los algoritmos probabílisticos 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.

Parte II

NUESTRA TÉCNICA

Capítulo 4

Técnica Basada en Inversiones

En esta tesis planteamos un enfoque original respecto al estado del arte en búsqueda en espacios métricos. A manera de resumen diremos que los índices conocidos usan las distancias (o cotas de distancia) entre elementos, para procesar la base de datos (ver Capítulo 2). En nuestro nuevo enfoque cada elemento *ordena* el espacio de acuerdo a cómo lo *percibe*. En este caso ese orden es por cercanía (en orden ascendente). Si bien para generar ese orden también usamos las distancias, la diferencia radica en que almacenamos otro tipo de información sobre esas distancias.

Como vimos en el capítulo 2, las técnicas existentes tienen dos fases: la fase de pre-procesamiento o indexación y la de consultas. Nuestra técnica también tiene estas dos fases. En la primera, crearemos nuestro índice; y en la segunda, mostraremos cómo usar ese índice para realizar las consultas.

La idea esencial es seleccionar un conjunto $\mathbb{P} = \{p_1, \ldots, p_k\} \subseteq \mathbb{U}, |\mathbb{P}| = k$; a los elementos en ese conjunto los llamaremos ***permutantes***. En la indexación, cada elemento $u \in \mathbb{U}$ ordena los elementos de $\mathbb{P}$ por distancia creciente a u. Nótese que cada elemento de la base de datos, una vez que ordena el conjunto $\mathbb{P}$, podría distinguirse respecto a otro u' usando sólo ambos órdenes. Esto claramente podría ser útil para acotar la búsqueda.

Al momento de responder una consulta q, también se genera un orden en el conjunto de permutantes $\mathbb{P}$ de acuerdo a su cercanía a q. De la diferencia entre el orden producido en $\mathbb{P}$ por q y por u, se puede inferir si u es relevante para q, y descartarlo sin haberlo comparado directamente contra q, o al menos estimar la semejanza entre u y q (sin posibilidad de descartarlo). En este capítulo se explica cómo podría usarse esa diferencia en los órdenes para descartar elementos sin compararlos directamente contra la consulta. En el capítulo 5 se explica cómo estimar la relevancia entre los elementos a partir de la semejanza de los órdenes.

En la figura 4.1 se muestra un espacio métrico en $\mathbb{R}^2$. El conjunto de permutantes $\mathbb{P}$ son los círculos rellenos (u_1 a u_4, llamados p_1 a p_4 respectivamente). En el ejemplo se seleccionaron

dos elementos para ilustrar la idea de los órdenes, u_7 y u_{12}. El elemento u_7 tiene más cerca a p_4, después a p_3, luego p_1, y finalmente p_2. El elemento u_{12} tiene más cerca a p_4 después a p_2, p_3, p_1. Note que los órdenes de u_7 y u_{12} son distintos. Cuando llega una consulta (llamada q en la figura), ésta también formará su orden del conjunto $\mathbb{P}$. La idea central es que estos órdenes nos den suficiente información para resolver las consultas por similaridad.

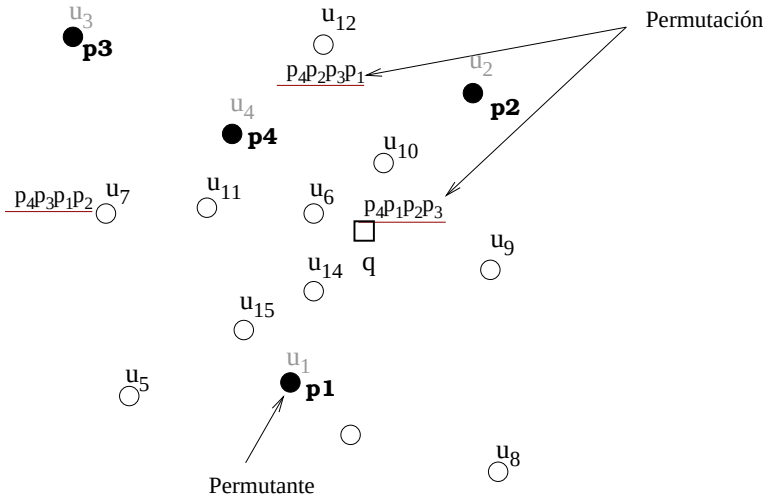


Figura 4.1: Los elementos u_7 y u_{12} perciben de manera diferente al conjunto de permutantes $\mathbb{P}$. El orden que establecen los elementos en la base de datos sobre el conjunto $\mathbb{P}$ será por cercanía (en orden creciente).

4.1. Definiciones

Definición 1 *Cada elemento $x \in \mathbb{X}$ define un preorden $\leq_x$ en $\mathbb{P}$ dado por la distancia a x. Este se define como: dados $p, p' \in \mathbb{P}$:*

$$p \leq_x p' \quad \Leftrightarrow \quad d(x,p) \leq d(x,p')$$

La relación $\leq_x$ es un preorden y no un orden porque varios elementos pueden estar a la misma distancia de x, y entonces $\exists p \neq p'$ tales que $p \leq_x p' \ \wedge \ p' \leq_x p$.

Definición 2 *Para cada x en $\mathbb{X}$ y $r > 0$, denotaremos la relación $\leq_{x,r}$ como un orden parcial estricto en $\mathbb{P}$ dado por:*

$$p \leq_{x,r} p' \quad \Leftrightarrow \quad d(x,p') - d(x,p) \geq r$$

La relación es un orden parcial estricto y no un orden porque no vale $p \leq_{x,r} p$ para ningún p. De

hecho, para cualquier par p, p' tal que $d(p, p') < r$, no vale ni $p \leq_{x,r} p'$ ni $p' \leq_{x,r} p$. Nótese que $\leq_x$ es equivalente a $\leq_{x,0}$ por definición, aunque hemos reservado $\leq_{x,r}$ para los $r > 0$.

Definición 3 *Para cada x en $\mathbb{X}$ y $r > 0$, denotaremos la relación $<_{x,r}$ como un orden parcial estricto en $\mathbb{P}$ dado por:*

$$p <_{x,r} p' \quad \Leftrightarrow \quad d(x, p') - d(x, p) > r$$

La relación es otro orden parcial estricto porque tampoco vale $p <_{x,r} p$ para ningún p. De hecho, para cualquier par p, p' tal que $d(p, p') \leq r$, no vale ni $p <_{x,r} p'$ ni $p' <_{x,r} p$.

Diremos que el par p, p' distingue a x si p y p' son comparables bajo $<_{x,r}$. La figura 4.2 (un espacio métrico en el plano con la distancia Euclidiana) muestra cómo p y p' dividen el espacio en tres áreas. Si u esta en el área que contiene p o en el área que contiene p', entonces p y p' distinguen a u.

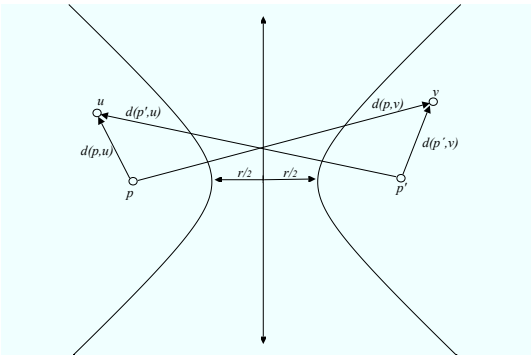


Figura 4.2: Ejemplo en el plano donde los permutantes p y p' distinguen a los objetos u y v los cuales tienen una r-inversión.

A continuación probamos que el orden parcial estricto de los elementos nos puede dar información de la distancia entre ellos.

Teorema 1 *Sean $<_{q,r_1}, \leq_{u,r_2}$ los órdenes parciales estrictos inducidos por $q, u \in \mathbb{U}$, respectivamente, para $r_1, r_2 \geq 0$. Si existen $p, p' \in \mathbb{P}$ tales que $p \leq_{u,r_2} p'$ y $p' <_{q,r_1} p$, entonces $d(q, u) > \frac{r_1 + r_2}{2}$.*

Demostración Del orden parcial estricto sabemos:

$$p \leq_{u,r_2} p' \quad \Leftrightarrow \quad d(u, p') - d(u, p) \geq r_2 \,, \tag{4.1}$$

$$p' <_{q,r_1} p \quad \Leftrightarrow \quad d(q, p) - d(q, p') > r_1 \,, \tag{4.2}$$

usando la desigualdad del triángulo:

$$\begin{aligned} d(u,p') &\leq d(u,q)+d(q,p')\,, \\ d(q,p) &\leq d(q,u)+d(u,p)\,, \end{aligned}$$

reemplazando ambas ecuaciones en (4.1) y (4.2), tenemos:

$$\begin{aligned} d(u,q)+d(q,p')-d(u,p) &\geq r_2\,, \\ d(u,q)+d(u,p)-d(q,p') &> r_1\,, \end{aligned}$$

sumando ambas ecuaciones:

$$\begin{aligned} 2d(u,q) &> r_2+r_1\,, \\ d(u,q) &> \frac{r_1+r_2}{2} \quad \blacksquare \end{aligned} \tag{4.3}$$

El teorema vale también obviamente si $\leq_{u,r_2}$ realmente es $<_{u,r_2}$.

Con estas definiciones se puede definir una *r-inversión*. Una *r*-inversión puede ser gráficamente ilustrada usando otra vez la figura 4.2: *v* está en el área que contiene p', entonces existe una *r*-inversión entre *v* y *u*.

Definición 4 *Existe una* **r-inversión** *entre $u,q \in \mathbb{X}$ cuando existen otros dos objetos $y,z \in \mathbb{P}$ y dos números r_1, r_2 tal que $r_1+r_2 \geq 2r$, tal que los objetos tienen un intercambio en los órdenes inducidos por u,q, es decir $y \leq_{u,r_2} z \quad y \quad z <_{q,r_1} y$.*

Veamos un ejemplo de cómo usar esta técnica. Inicialmente escogemos $\mathbb{P} \subseteq \mathbb{U}$ y $r_2 > 0$ para indexar la base de datos (preprocesamiento). Almacenamos sólo los órdenes parciales estrictos $\leq_{u,r_2}$ inducidos en $\mathbb{P}$ por cada elemento $u \in \mathbb{U}$. Supongamos que el orden parcial estricto inducido sobre $\mathbb{P}$ por *u* es:

$$p_1 \leq_{u,r_2} p_2 \leq_{u,r_2} p_3 \leq_{u,r_2} \cdots \leq_{u,r_2} p_k\,,$$

y que el orden parcial estricto sobre el mismo conjunto $\mathbb{P}$ inducido por una consulta $q \in \mathbb{X}$ es:

$$p_2 <_{q,r} p_1 <_{q,r} p_3 <_{q,r} \ldots <_{q,r} p_k\,,$$

donde para ejemplificar hemos supuesto que $\mathbb{P}$ queda totalmente ordenado en ambos casos.

Nótese que existe una *r*-inversión en los órdenes, en particular en este caso $p_1 \leq_{u,r_2} p_2$ y $p_2 <_{q,r} p_1$. De acuerdo al teorema 1 es posible asegurar que $d(q,u) > r$ y por lo tanto descartar el elemento *u* sin haber calculado su distancia contra la consulta. El mismo proceso se aplica para cada $u \in \mathbb{U}$ usando el mismo orden parcial estricto para la consulta pues éste sólo depende de $\mathbb{P}$.

Nótese que el orden parcial estricto de la consulta sólo se calcula una vez.

4.2. Comparación de las Inversiones contra las Familias Existentes

En esta sección veremos que las *r*-inversiones son necesariamente inferiores a la técnica basada en pivotes, sin embargo son las bases para la técnica basada en permutaciones que veremos en el siguiente capítulo.

4.2.1. Inversiones frente a los Algoritmos de Pivotes

Una pregunta natural es cómo se compara la técnica basada en inversiones y la técnica basada en pivotes, si se usa el mismo conjunto de pivotes/permutantes. Formalmente mostraremos que las *r*-inversiones no son superiores a los algoritmos basados en pivotes.

Teorema 2 *Sean* p *y* p' *tal que* $p \leq_{u,r_1} p'$ *y* $p' <_{q,r_2} p$. *Entonces* $|d(u,p) - d(q,p)| > \frac{r_1+r_2}{2}$ *ó* $|d(u,p') - d(q,p')| > \frac{r_1+r_2}{2}$.

Demostración Del enunciado tenemos

$$\begin{aligned} d(u,p) + r_1 &\leq d(u,p'), \\ d(q,p) &> d(q,p') + r_2, \end{aligned}$$

restando ambas ecuaciones:

$$d(u,p) + r_1 - d(q,p) < d(u,p') - d(q,p') - r_2,$$

esto es,

$$[d(u,p') - d(q,p')] + [d(q,p) - d(u,p)] > r_1 + r_2.$$

Por lo tanto, esto implica que se cumple alguna de las ecuaciones

$$\begin{aligned} |d(u,p') - d(q,p')| &> \frac{r_1+r_2}{2} \quad \text{ó} \\ |d(q,p) - d(u,p)| &> \frac{r_1+r_2}{2} \end{aligned}$$

∎

De este teorema se puede concluir que los elementos que pueden ser descartados con los permutantes también serán descartados por un algoritmo basado en pivotes.

Sin embargo almacenar sólo el orden de los permutantes bastan unos cuantos bits (dependiendo del número de permutantes), esto es $\log_2 |\mathbb{P}|$ bits por permutante, lo que se traduce en utilizar mucho menos espacio que la técnica de pivotes para la misma cantidad de pivotes o utilizar más permutantes para la misma cantidad de espacio, lo que podría lograr mejores resultados. Es decir, para k permutantes usados, el espacio requerido es $kn\lceil \log_2 k \rceil$ bits para almacenar las n permutaciones. En el caso de usar k pivotes, se requiere $kn\lceil \log_2 N \rceil$ bits para guardar kn distancias, donde N es el número de valores de distancia posibles. Usualmente, N es mucho mas grande que k. En las práctica, las distancias con valores reales pueden usar hasta 4 bytes para su representación, esto es, $32|\mathbb{P}|$ bits, mientras que los permutantes $|\mathbb{P}|\log_2 |\mathbb{P}|$. Es decir, para que ambos algoritmos usen la misma cantidad de memoria se necesitan 2^{32} permutantes/pivotes. Por otro lado, se pueden usar 256 permutantes con sólo 1 byte por celda. Esto significa que 256 permutantes usarían la misma cantidad de memoria que 64 pivotes.

4.2.2. Inversiones frente a las Particiones Compactas

El caso donde no es evidente que las particiones compactas y las inversiones difieren es cuando las particiones dividen el espacio usando hiperplanos, es decir, cada elemento pertenece a la zona de su centro más cercano (ver página 28). En este caso, si los permutantes son esos centros. En cada zona los elementos tienen distintos preórdenes. Si u está en la zona de p_1, entonces $p_1 \leq_u p_2$ para todo centro p_2, es decir p_1 es el mínimo de $\mathbb{P}$ de acuerdo al preorden $\leq_u$. Si u' pertenece a la zona de p_2, siempre habrá una inversión en el orden entre $\leq_u$ y $\leq_{u'}$, pues $p_1 \leq_u p_2$ y $p_2 \leq_{u'} p_1$. Sin embargo, podría haber otras inversiones en el preorden dentro de una misma zona.

Los algoritmos basados en particiones compactas discriminan por el primer elemento del orden (es decir por su centro más cercano). Nuestra técnica permite detectar inversiones incluso dentro de una zona, cuando ocurren entre otros elementos de $\mathbb{P}$ que no son el primero.

Para ilustrar lo anterior, observe la figura 4.3. Nótese que todos los elementos en la zona donde se ubica el elemento de consulta q, tienen un preorden distinto. Esto hace que los elementos en la zona aún puedan ser discriminados y se pueda reducir cálculos de distancia.

Una comparación interesante es respecto a los diagramas de Voronoi (explicados en la página 3.2). En los diagramas de Voronoi los centros dividen el espacio creando una región donde se cumple que los elementos en esa zona están más cercanos al centro de esa zona que a ningún otro. En la figura 4.4 se muestran las particiones de Vonoroi formadas a partir de 3 centros y las particiones formadas a partir de los mismos tres centros vistos como permutantes. La técnica basada en permutaciones crea una región por cada permutación distinta.

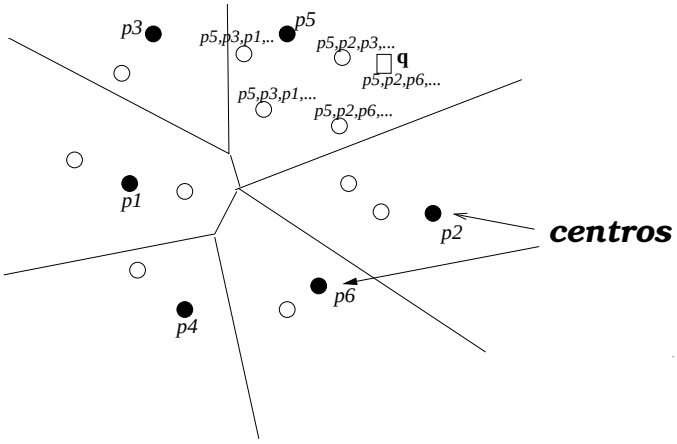


Figura 4.3: Ejemplo de una división producida por los algoritmos basados en particiones compactas. Nótese cómo los elementos de una zona tienen distintos pre-órdenes. Esto permite diferenciar entre los elementos de la misma zona.

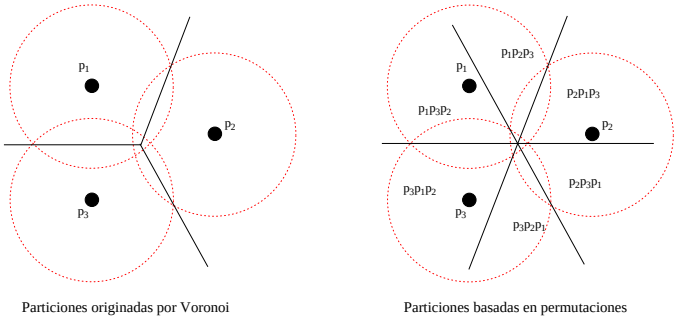


Figura 4.4: Ejemplo de las particiones generadas por Voronoi y las particiones generadas por los permutantes.

4.3. Uso de la Técnica Basada en Inversiones

Entonces las r-inversiones dan un método para descartar elementos en un consulta de rango. Un algoritmo basado en búsqueda exacta en r-inversiones es presentado en esta sección por medio de un índice que utiliza esta técnica.

En base al teorema 1, sabemos que, a partir de dos órdenes parciales estrictos (entre q y u con un radio r), si éstos tienen una r-inversión entonces la distancia entre q y u es mayor que el radio r. Cada $u \in \mathbb{U}$ podría formar el orden parcial estricto dado un radio r ($\leq_{u,r}$) durante la fase de preprocesamiento o indexación. Si al momento de una consulta q calculásemos el orden parcial estricto $<_{q,r}$, entonces podríamos usar el teorema 1 para descartar aquellos elementos en $\mathbb{U}$ que tienen una r-inversión en su orden parcial estricto, con respecto a q. Esto tiene dos problemas: el primero es que necesitamos conocer r al momento de indexar, y el segundo es que almacenar un orden $\leq_{u,r}$ es complejo, pues habrían muchas posibles combinaciones de elementos en $\mathbb{P}$ que cumplan con $\leq_{u,r}$, y se podría requerir una costosa comparación individual de $<_{q,r}$ contra cada $\leq_{u,r}$ para detectar r-inversiones. Buscamos en cambio una técnica que nos permita descartar varios elementos de una vez y no dependa de r al momento de indexar.

Una solución al problema de usar los órdenes parciales estrictos es remplazarlos por los pre-órdenes $\leq_u$ en los elementos de la base de datos, de modo que el índice no dependa de r. Sin embargo, al momento de la consulta, para filtrar elementos usando el teorema 1 es necesario tener el orden parcial estricto de la consulta y el preorden de los elementos en la base de datos, como en el teorema 1.

De acuerdo al preorden $\leq_u$ se puede asociar $\leq_u$ a una ***permutación*** Π_u de $p_1, p_2, \ldots, p_k$, donde, $\forall i$, $\Pi_u[i] \leq_u \Pi_u[i+1]$. Esto es, los elementos en $\mathbb{P}$ quedan ordenados de manera creciente en base a la distancia a u, siendo $\Pi_u[i]$ el elemento en la posición i-ésima dentro de la permutación de u. Los elementos a la misma distancia de u toman un orden arbitrario pero consistente. Así, veremos la permutación de cada u como una cadena de símbolos que indexaremos en el preprocesamiento.

Por ejemplo, sea u un elemento cuyas distancias son: $d(p_1,u) = 6$, $d(p_2,u) = 4$, $d(p_3,u) = 1$, $d(p_4,u) = 9$, $d(p_5,u) = 7$, $d(p_6,u) = 3$, su permutación será $\Pi_u = p_3, p_6, p_2, p_1, p_5, p_4$. Así $\Pi_u[1] = p_3$, $\Pi_u[2] = p_6$, etc. Véase la figura 4.1.

4.3.1. Indexación

A partir de la idea de usar el preorden en los elementos de la base de datos y el orden parcial estricto en la consulta, surge la interrogante de cómo procesar la base de datos para que la consulta no se compare contra todas las permutaciones una a una. Una alternativa es usar un trie (véase la sección 2.5.2), puesto que es una estructura de datos diseñada para buscar en cadenas y las permutaciones pueden ser vistas como cadenas o palabras. De esta manera podemos filtrar de

una vez todos los elementos que comparten un prefijo sobre el cual se detectó una *r*-inversión.

En la figura 4.5 se muestra un ejemplo de un trie construido sobre un conjunto de permutaciones de tamaño *k*. El tamaño del índice ocupado por esta técnica será en el peor caso $O(kn)$, donde *n* es el tamaño de la base de datos. Nótese que nuestro trie unifica caminos cuando todos los elementos comparten la misma permutación (una hoja puede tener profundidad menor que *k*, si todos los elementos que contiene comparten una misma permutación). De esta forma los elementos en cada hoja del trie comparten la misma permutación.

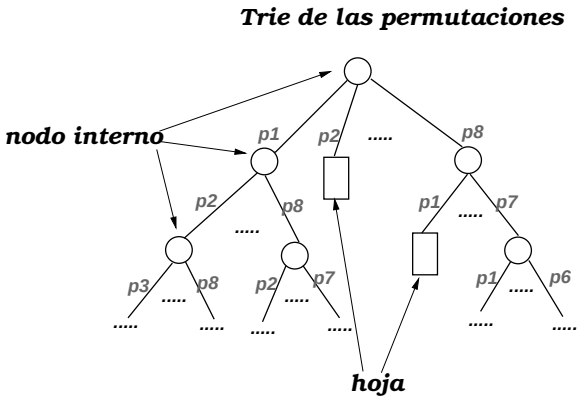


Figura 4.5: Nuestro índice es el conjunto de permutaciones ordenadas en un trie. En este ejemplo $|\mathbb{P}| = 8$.

De las observaciones más importantes de resaltar en el trie, tenemos:

- Al variar el número de permutantes cambiará la aridad del trie
- La altura de un árbol es $O(\log_{aridad} n)$, cuando éste es balanceado.
- Para valores grandes de *k*, probablemente el trie no tendrá una profundidad muy alta pues a medida que aumente el número de permutantes es más improbable que los elementos compartan permutaciones hasta una cierta profundidad. Sin embargo puede haber unos pocos pares de elementos que tengan una permutación casi igual y difieran al final.
- En el caso contrario, cuando hay pocos permutantes, la profundidad del árbol será mayor y es posible descartar más elementos con menos operaciones de recorrido por el trie, porque las permutaciones compartirán más prefijos. Sin embargo, con pocos permutantes, el número de combinaciones entre permutantes que favorezcan encontrar *r*-inversiones disminuye.

Por lo tanto, debe existir un balance entre los permutantes usados que minimice el número de comparaciones en una consulta. Esta comparación se verá al final del capítulo en la sección de experimentación.

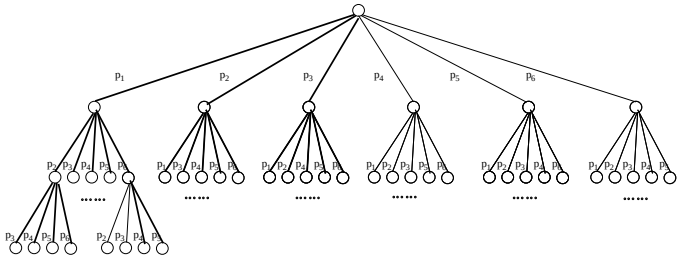


Figura 4.6: Ejemplo del proceso de búsqueda en un trie basado en permutaciones.

4.3.2. Proceso de Búsqueda

En esta sección detallaremos cómo procesar una consulta usando su permutación y recorriendo el trie de las permutaciones.

Inicialmente, construímos la permutación de la consulta Π_q, esto es, calculamos $|\mathbb{P}| = k$ distancias (llamadas comparaciones internas en la sección 2.4).

Una vez con esta permutación podemos recorrer el trie partiendo desde la raíz, sólo se descenderá (recursivamente) por aquellas ramas en las que no se produzca una r-inversión (con respecto a algún ancestro en el camino al nodo). Cada vez que se descarte una rama se descartarán todos los elementos en ella pues comparten un mismo prefijo. El proceso continúa hasta llegar a una hoja, donde se comparan directamente los elementos en esa hoja contra la consulta (comparaciones externas).

El proceso de búsqueda será explicado para una consulta por rango $(q,r)_d$. Si existe un par de permutantes $\Pi_q[i], \Pi_q[j]$ para $i, j \leq k$, tal que $d(\Pi_q[i], q) > d(\Pi_q[j], q) + 2r$ (nótese que $j < i$ pues los elementos en la permutación están ordenados por distancia), entonces debemos filtrar los objetos u tales que Π_u satisface $d(\Pi_q[j], u) \geq d(\Pi_q[i], u)$ (esto es, el elemento $\Pi_q[i]$ se encuentra antes que $\Pi_q[j]$ en la permutación Π_u), pues significa que hemos encontrado una r-inversión.

Por ejemplo: sea $r = 1$,

$$\begin{aligned} \Pi_q &= p_1, p_2, p_3, p_4, p_5, p_6 \ , \\ \Pi_u &= p_3, p_6, p_2, p_1, p_5, p_4 \end{aligned}$$

y las distancias son $d(p_i, q) = i$. Los pares de elementos que cumplen con la condición de r-inversión $d(\Pi_q[i], q) > d(\Pi_q[j], q) + 2r$ son: (p_1, p_4), pues $d(p_4, q) > d(p_1, q) + 2$, (p_1, p_5), (p_1, p_6), (p_2, p_5), (p_2, p_6), (p_3, p_6). Si revisamos Π_u veremos que existe una r-inversión entre (p_6, p_2) (para este ejemplo existe otra r-inversión (p_6, p_1)), por lo tanto podemos asegurar $d(q, u) > r$ y en este caso u no es relevante para la consulta. En la figura 4.6 se muestra gráficamente este proceso. Inicialmente es posible descartar p_4, p_5 y p_6 pues necesariamente habrá una r-inversión respecto a p_1.

El algoritmo 3 describe el proceso de búsqueda mediante el recorrido del trie. Las condiciones iniciales son: T es la raíz del árbol, $d_{max} = d(\Pi_q[1], q)$ y $nivel = 0$. Se utilizó $d(\Pi_q[1], q)$ pues es la mínima distancia a la que se encuentran los pivotes y con esta distancia es posible saber si habrá una r-inversión desde el primer pivote. En el ejemplo del párrafo anterior aquellas permutaciones que empiecen por el pivote p_4, p_5 o p_6 serán descartadas en el primer nivel (línea 11). En cada nivel se desciende por aquellas ramas (T') que cumplan con $d_{max} - d(p,q) \leq 2r$, donde $p \in \mathbb{P}$ es el permutante por el que se desciende a T'. El proceso es recursivo en los subárboles usando la rama T', $\max(d_{max}, d(p,q))$ y $nivel + 1$. En cada trayectoria es necesario mantener la distancia máxima del permutante encontrado (en esa trayectoria) pues sólo así se detectarán r-inversiones con seguridad. Mantener d_{max} es equivalente a recordar cual es el permutante más alejado que se ha encontrado en esa trayectoria.

Cuando se llega a una hoja antes de comparar los elementos en ésta, se revisa la permutación de la hoja completa. Si no se detecta una r-inversión, aún revisando la permutación completa, entonces se comparan por distancia los elementos en la hoja contra la consulta reportando los que son relevantes.

Algoritmo 3 Búsqueda-trie(T trie, d_{max}, $nivel$)

```
1:  if T ≠ NULL then
2:    if T es una hoja then
3:      if Probar-hoja(T, nivel)= TRUE then
4:        for u ∈ T do
5:          if d(u,q) ≤ r then reportar u
6:        end for
7:      end if
8:    else
9:      for T' hijo de T do
10:       Sea p el permutante que conecta T con T'
11:       if d_max − d(p,q) ≤ 2r then
12:         Búsqueda-trie(T', máx(d_max, d(p,q)), nivel + 1)
13:       end if
14:     end for
15:   end if
16: end if
```

4.4. Experimentación

En esta sección mostraremos el desempeño de nuestra técnica en bases de datos reales y sintéticas. La base de datos real fue un diccionario de palabras y la base de datos sintética fueron vectores en el cubo unitario. En particular, nos interesa mostrar la cantidad de comparaciones de distancia necesarias (internas y externas), para responder una consulta usando la técnica basada en

Algoritmo 4 Probar-hoja(T trie, d_{max}, *nivel*)

```
1: Sea Π_u la permutación de la hoja
2: for i ← nivel to k do
3:    if d_max − d(Π_u[i], q) > 2r then
4:       return FALSE
5:    end if
6:    d_max ← máx(d_max, d(Π_u[i], q))
7: end for
8: return TRUE
```

inversiones, modificando algunos parámetros tales como el número de permutantes y la dimensión de los datos. Para detectar las r-inversiones se utilizó el trie explicado en la sección anterior.

4.4.1. Diccionario

Los experimentos en la base de datos real se realizaron usando un diccionario de 86.062 palabras en español (véase la sección 2.6.2). Para este experimento se hicieron búsquedas por rango con $r = 1$. El desempeño de la técnica de las inversiones se muestra en la figura 4.7, donde ya están consideradas las comparaciones internas y externas. En esta figura se compara la técnica de las inversiones con un algoritmo clásico de pivotes que usa la misma cantidad de pivotes que permutantes usa nuestra técnica. Nótese la relación entre el número de permutantes y el desempeño de nuestra técnica. Al aumentar el número de permutantes (comparaciones internas) hay mayor probabilidad de encontrar r-inversiones (es decir, se disminuyen las comparaciones externas) y eso se refleja en los experimentos. A partir de 40 permutantes se debe revisar el 1 % de la base de datos. Cuando se usan 80 permutantes se alcanza casi la misma respuesta que un algoritmo basado en pivotes.

4.4.2. Cubo unitario

Otros experimentos interesantes en el estudio de la técnica de las inversiones es cómo afecta la dimensión de los datos al desempeño de ésta. Para este nuevo experimento se usaron conjuntos sintéticos distribuidos en el cubo unitario (detalles en la sección 2.6.1). El rango de dimensiones fue $4 \leq dim \leq 12$. El tamaño de la base de datos fue de 10.000 objetos y las consultas recuperaron el 0,02 % de la base de datos en consultas por rango.

El desempeño de la técnica de las inversiones en distintas dimensiones se muestra en la figura 4.8. En esta figura se muestran las comparaciones internas más las externas para resolver las consultas. Nótese que en dimensión baja (por ejemplo, 4) son suficientes $k = 16$ permutantes para resolver la consulta comparando menos del 3 % de la base de datos. Sin embargo, a medida que la dimensión aumenta, se requiere un mayor número de permutantes para mejorar el desempeño de

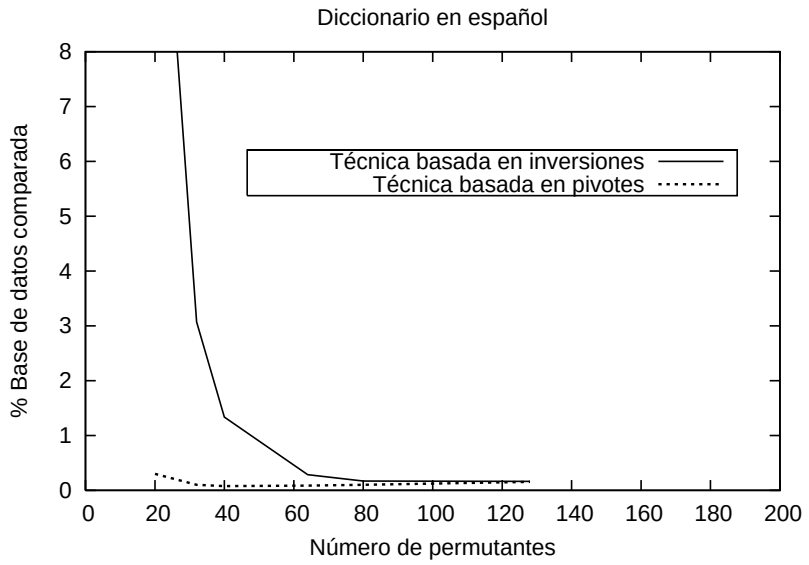


Figura 4.7: Desempeño de la técnica de las inversiones, para un diccionario de palabras, comparado contra un algoritmo clásico de pivotes.

la técnica basada en inversiones, pues esta técnica es sensible a la dimensión.

La sensibilidad a la dimensión se debe a que los rangos de búsqueda dependen de la dimensión. Es decir, a mayor dimensión los rangos de búsqueda son mayores ya que las distancias entre elementos son mayores y las diferencias de distancias son menores. La consecuencia directa es que entre más alta sea la dimensión menos elementos son capaces de cumplir la condición de una *r*-inversión (teorema 1).

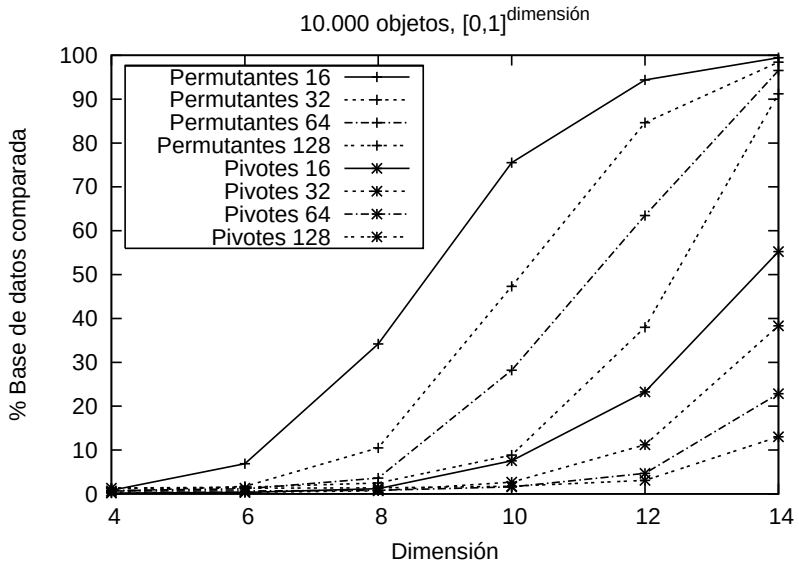


Figura 4.8: Desempeño de la técnica de las inversiones en diferentes dimensiones. 10.000 vectores aleatorios distribuidos uniformemente en el cubo unitario.

Capítulo 5

Algoritmo Basado en Permutaciones

En este capítulo describiremos una propuesta probabilística para resolver el problema de la búsqueda por similaridad en espacios métricos. La técnica que presentaremos está estrechamente relacionada a la presentada en el capítulo anterior, donde cada elemento forma su permutación sobre un conjunto de elementos (permutantes) y el orden de la permutación de ese conjunto es dado por distancia creciente al elemento.

A diferencia de la técnica de las inversiones, en este capítulo no definimos una manera de filtrar elementos, sino que simplemente aprovechamos la semejanza entre permutaciones para identificar aquellos elementos con mayor probabilidad de ser cercanos a la consulta.

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.

La idea general de las permutaciones es: seleccionamos un conjunto $\mathbb{P} = \{p_1, \ldots, p_k\} \subseteq \mathbb{U}$, llamados permutantes, en principio de manera aleatoria. Para cada $u \in \mathbb{U}$, calculamos su preorden $\leq_u$ en $\mathbb{P}$ y formamos su permutación Π_u (véase la figura 4.1 y la definición 4.1, página 50). Los empates se deciden de forma arbitraria pero consistente.

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 $d(u, v) = 0$ (u puede ser distinto de v en un espacio pseudo-métrico), las permutaciones Π_u y Π_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.

El resto del capítulo está organizado de la siguiente manera: primero se presentará el proceso de búsqueda describiendo la forma de comparar dos permutaciones y cómo ordenarlas para su evaluación contra la consulta. Posteriormente se analizará cuál es el método más rápido,

en tiempo, para conocer las permutaciones más semejantes a la permutación de la consulta. Éste será el método ocupado en la sección de experimentos, donde se verá el buen desempeño de esta técnica probabilística en diferentes tipos de espacios métricos. Por ejemplo, en dimensión 128 en el cubo unitario se puede obtener el 90 % de la respuesta revisando sólo el 5 % de la base de datos. Finalizamos el capítulo mostrando heurísticas para seleccionar permutantes, en un intento por mejorar los resultados.

5.1. Comparación entre Permutaciones

Al momento de la consulta, calculamos Π_q para la consulta q usando $\leq_q$ sobre el mismo conjunto $\mathbb{P}$. Luego, para saber en qué orden revisar los elementos, ordenamos los $\Pi_u, u \in \mathbb{U}$ 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 usamos Spearman Rho [FKS03], la cual denotaremos 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, $\Pi_u^{-1}(p_i)$ y $\Pi_q^{-1}(p_i)$, respectivamente. Formalmente [a],

$$S_\rho(\Pi_q, \Pi_u) = \sum_{i=1}^{k} |\Pi_u^{-1}(p_i) - \Pi_q^{-1}(p_i)|^2 \tag{5.1}$$

A continuación daremos un ejemplo de cómo se calcula $S_\rho(\Pi_q, \Pi_u)$. Sea $\Pi_q = p_1, p_2, p_3, p_4, p_5, p_6$ la permutación de la consulta, y sea $\Pi_u = p_3, p_6, p_2, p_1, p_5, p_4$ la permutación de un elemento u. El elemento p_3 dentro de la permutación Π_u está desfasado dos posiciones con respecto a su posición en Π_q. Las diferencias entre las permutaciones para cada elemento son: $|1-3|, |2-6|, |3-2|, |4-1|, |5-5|, |6-4|$, y la suma de los cuadrados de todas las diferencias es $S_\rho(\Pi_q, \Pi_u) = 34$.

En la figura 5.1 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.).

Existen otras medidas de similaridad entre permutaciones [FKS03], tales como Ken-

[a] La definición de Spearman Rho incluye la operación $()^{\frac{1}{2}}$ en el cálculo de S_ρ presentado, pero no incluimos esta operación por eficiencia y porque no altera el orden.

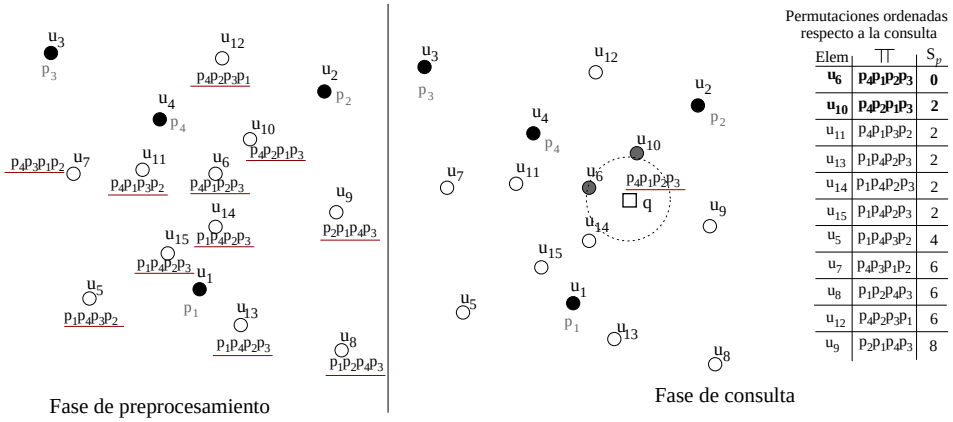


Figura 5.1: Ejemplo de las dos fases de un algoritmo basado en permutaciones. Las permutaciones fueron ordenadas por valor de S_ρ respecto a la permutación de la consulta.

dall Tau y Spearman Footrule. La métrica de Kendall Tau entre permutaciones está definida como sigue. Para cada par $\{p_i, p_j\} \in \mathbb{P}$, si p_i y p_j están en el mismo orden en Π_u y Π_q, (esto es $\Pi_u^{-1}(p_i) < \Pi_u^{-1}(p_j) \Leftrightarrow \Pi_q^{-1}(p_i) < \Pi_q^{-1}(p_j)$) entonces $K_{p_i,p_j}(\Pi_u, \Pi_q) = 0$; sino $K_{p_i,p_j}(\Pi_u, \Pi_q) = 1$. Kendall Tau está dada por $K(\Pi_u, \Pi_q) = \sum_{\{p_i,p_j\}\in\mathbb{P}} K_{p_i,p_j}(\Pi_u, \Pi_q)$. Esto es semejante a contar el número de intercambios necesarios en un ordenamiento con el método de la burbuja para convertir una permutación en la otra.

Por otro lado, Spearman Footrule es la distancia L_1 entre dos permutaciones. Formalmente,

$$F(\Pi_u, \Pi_q) = \sum_{i=1}^{k} |\Pi_u^{-1}(p_i) - \Pi_q^{-1}(p_i)| \tag{5.2}$$

En la sección 5.3 mostraremos que estas dos medidas, para nuestro propósito, son inferiores a Spearman Rho en distintos aspectos, por lo que esta última será la medida que usaremos en la parte experimental.

5.2. Buscando las Permutaciones más Similares

En esta sección consideramos cómo encontrar las permutaciones de la base de datos más similares a la de la consulta. Como se vió en la sección anterior los elementos con las permutaciones más similares tendrán altas probabilidades de ser parte del conjunto respuesta (elementos promisorios).

Para el proceso de búsqueda de las permutaciones más similares a Π_q distinguimos tres alternativas. Una es usar un trie con las permutaciones y recorrerlo calculando un valor parcial de S_ρ hasta llegar a una hoja; las otras dos consisten en calcular todos los S_ρ y ordenar toda la base

de datos por ese valor en forma creciente (con un ordenamiento parcial o total). En esta sección presentaremos un análisis de estas tres alternativas.

5.2.1. Empleando un Trie

Al igual que en la técnica basada en inversiones (página 58), es posible formar un trie con las permutaciones. Esta vez el objetivo es recorrer las hojas con las permutaciones más similares a la permutación de la consulta Π_q sin pasar por las otras. Para recorrer el trie usaremos valores parciales de S_ρ (el valor parcial se calcula usando sólo un prefijo de Π_u). La idea principal es usar una cola de prioridad con los subárboles y hojas cuya similaridad parcial $S_\rho(\Pi_q, \Pi_u)$ ya ha sido calculada pero que aún no han sido procesados. La cola de prioridad mantendrá ordenados los elementos por el valor parcial de S_ρ y en caso de empates decidirá por mayor profundidad en el trie. La ventaja de usar un trie es que permite revisar los elementos en $\mathbb{U}$, empezando por aquellos con menor S_ρ, sin haber calculado todos los S_ρ, además de que su costo de procesamiento para alcanzar una hoja $O(\log_k n)$.

El proceso de búsqueda en el trie consiste en insertar tuplas de la forma (T, l, S_ρ) en la cola de prioridad, siendo T un nodo u hoja del trie, l la profundidad del nodo, y S_ρ su valor parcial de Spearman Rho. Inicialmente, insertamos $(R, 0, 0)$, donde R es la raíz del trie. Como su permutación es la cadena vacía, el valor parcial de S_ρ es cero y su nivel 0. El proceso de búsqueda repetidamente extrae y procesa al primer elemento en la cola de prioridad, (T, l, S_ρ), procediendo según el caso:

1. **Caso 1**. *Si T es un nodo interno del trie*, cada hijo T' de T será insertado en la cola de prioridad, con su nuevo el valor parcial S'_ρ. Se puede calcular la similaridad parcial S'_ρ de cada nodo del trie usando sólo una operación. Para cada nodo T' descendiente de T por el permutante p, S'_ρ parcial para T' es $S'_\rho = S_\rho + |(l+1) - \Pi_q^{-1}(p)|^2$. Entonces insertamos $(T', l+1, S'_\rho)$ en la cola de prioridad.

2. **Caso 2**. *Si T es un hoja* de profundidad menor que k ($l < k$), entonces todos sus elementos compartirán una misma permutación, sin embargo no debemos procesar aún estos elementos sino tratarlos como caminos unarios de T. Sea la permutación restante, $p_{i_{l+1}}, \ldots, p_{i_k}$. Reinsertaremos T en la cola de prioridad como $(T, l+1, S_\rho + |(l+1) - \Pi_q^{-1}(p_{i_{l+1}})|^2)$.

3. **Caso 3**. *Si T es una hoja* de profundidad k, entonces los elementos en esa hoja serán comparados contra la consulta.

Como S'_ρ se incrementa de padres a hijos, es claro que los elementos en $\mathbb{U}$ serán visitados en el mismo orden que si hubieran sido ordenados por el valor total de S_ρ.

El pseudocódigo del recorrido del trie está dado en el algoritmo 5. En el proceso *Trie_búsqueda* la condición del algoritmo en el ciclo *while* de la línea 3 podría cambiarse por un cierto porcentaje de la base de datos comparado (para sólo comparar un porcentaje de la base de datos). Existen otras alternativas que generalmente son usadas para detener las búsquedas en los algoritmos que almacenan distancias (por ejemplo, cota mínima, un parámetro de precisión, etc [b]), sin embargo, en este caso no es posible su uso pues no podemos garantizar una cierta distancia a partir de un valor de similaridad entre permutaciones.

Finalmente, en la función *ProcesarHoja* los elementos son comparados directamente contra la consulta (comparaciones externas).

Algoritmo 5 Trie_búsqueda(Trie R, Consulta q)

1: Sea C_p una cola de prioridad
2: insertar($C_p, R, 0, 0$)
3: **while** C_p no está vacía **do**
4: $(T, l, S_\rho) \leftarrow$ extraer_mínimo(C_p)
5: **if** T es un nodo interno **then**
6: // Caso 1
7: **for** T' hijo de T vía símbolo p **do**
8: insertar $(C_p, T', l+1, S_\rho + |(l+1) - \Pi_q^{-1}(p)|^2)$
9: **end for**
10: **else if** T es una hoja y $l < k$ **then**
11: // Caso 2
12: Sea $p_{i_1} \dots p_{i_k}$ la permutación
13: insertar $(C_p, T, l+1, S_\rho + |(l+1) - \Pi_q^{-1}(p_{i_{l+1}})|^2)$
14: **else**
15: // Caso 3
16: *ProcesarHoja*(T, q) // comparaciones externas
17: **end if**
18: **end while**

Algoritmo 6 Sort-Búsqueda(Π_q, q)

1: Sea $A[1, n]$ un arreglo
2: Sea $m \leq n$
3: $i \leftarrow 1$
4: **for** $u \in \mathbb{U}$ **do**
5: $A[i] \leftarrow S_\rho(\Pi_u, \Pi_q)$
6: $i \leftarrow i + 1$
7: **end for**
8: sort(A) // por valor creciente de S_ρ.
9: **for** $i \leftarrow 1$ to m **do**
10: Comparar $A[i]$ con la consulta q. //comparaciones externas
11: **end for**

[b] Una explicación detallada de cota mínima se hizo en la página 42

5.2.2. Ordenando el Conjunto

Otras alternativas para buscar las permutaciones más cercanas consisten en ordenar total o parcialmente la base de datos respecto al valor S_ρ. Este valor debe ser calculado previamente para cada elemento en la base de datos. Tal proceso se muestra en el algoritmo 6. El ordenamiento se realizó con: *QuickSort* o *BucketSort* para ordenar la base de datos completa; e *IQS* para un orden parcial.

En el caso de un ordenamiento completo toda la base de datos quedará ordenada y se podrá comparar elemento a elemento de la base de datos contra la consulta, hasta que se decida dejar de comparar elementos (línea 8 del algoritmo 4). Para este ordenamiento se usó la implementación de *QuickSort* (qsort) de la librería estándar de lenguaje *C* cuya complejidad promedio es $O(n \log n)$, y BucketSort, cuya complejidad es $O(n + k^2)$.

Un ordenamiento parcial consiste en recuperar uno a uno los elementos de la base de datos más similares (primero el de S_ρ más pequeño, después el segundo más pequeño, etc.) sin tener que ordenar el conjunto completo. Una ventaja de este tipo de algoritmos es que ordenan sólo una parte de los datos. El algoritmo de ordenamiento parcial utilizado fue IQS [PN06], debido a que tiene los mejores tiempos de procesamiento para el ordenamiento parcial.

IQS es básicamente un QuickSort. Su principal diferencia consiste en ordenar sólo la parte del arreglo que se requiere y no todo el arreglo como es el caso del QuickSort. Su complejidad de ordenamiento es $O(n + m \log m)$, siendo m la cantidad de elementos ordenados requeridos.

La figura 5.2 se muestra la comparación del desempeño de las alternativas de ordenamiento (Trie, QuickSort, BucketSort e IQS) en una base de datos formada por vectores aleatorios en el cubo unitario en dimensión 128, usando un procesador PowerPC G4 a 1.33 GHz con 768 MB RAM y 512 KB L2 de cache, con un bus de 167 MHz y corriendo Mac OS X 10.4.7 . Se puede ver que la alternativa de ordenamiento completo de la base de datos con QuickSort es más rápida que usar un trie. La razón es que el QuickSort toma tiempo $O(n \log n)$, más los cálculos de S_ρ, que toman tiempo $O(kn)$. Por otro lado, al recorrer el trie usando una cola de prioridad se pueden llegar a procesar casi todos los nodos internos (que pueden ser casi n) antes de llegar a una hoja. Esto ocurre si la mayoría de los nodos internos tiene un valor parcial de S_ρ más pequeño que el S_ρ (completo) de cualquier hoja. Por lo tanto, un elemento de $\mathbb{U}$ puede provocar hasta k (el largo de la permutación) inserciones y extracciones de la cola de prioridad, lo que ocasiona que, en el peor caso, el trie pueda tomar tiempo $O(kn \log n)$ más el costo $O(kn)$ de calcular el valor de S_ρ.

Este algoritmo basado en permutaciones se comparó (en tiempo de procesamiento, por el momento) contra un algoritmo basado en pivotes. Este último también puede volverse probabilístico si se ordena la base de datos respecto a la distancia máxima (véase la ecuación 5.3, página 71) y posteriormente se comparan en ese orden contra la consulta. Es decir, los elementos u en la

base de datos son ordenados por $L_\infty(q,u)$ y comparados contra la consulta en ese orden, como en [BN02]. Con esta comparación mostramos que la técnica basada en permutaciones es más rápida que la técnica basada en pivotes. En las secciones siguientes se mostrará el desempeño en cuanto al grado de recuperación de ambas técnicas. Los algoritmos de ordenamiento para la técnica basada en pivotes también fueron QuickSort e IQS. La parte inferior de la figura 5.2 se amplió en la figura 5.3(a).

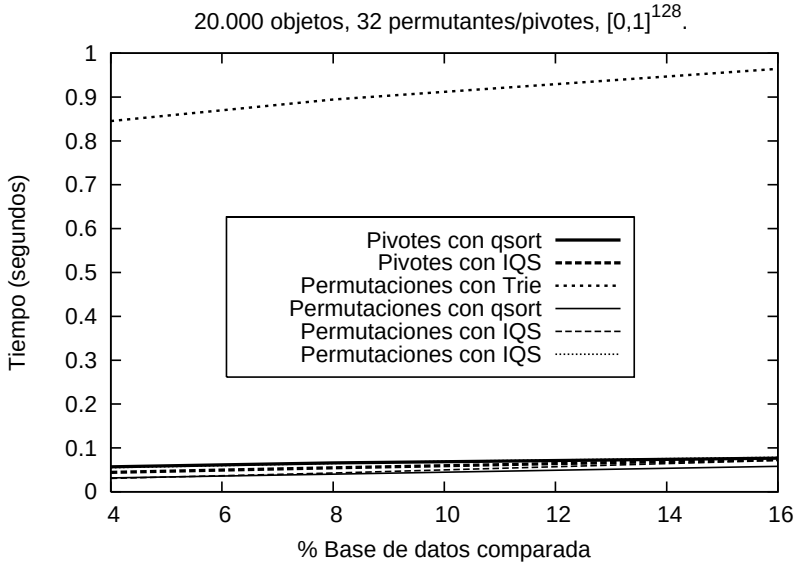


Figura 5.2: Tiempo de procesamiento de los algoritmos propuestos: la técnica de las permutaciontes y los algoritmos pivoteros. El criterio de ordenamiento fue Spearman Rho y L_∞, respectivamente. Los métodos de ordenamiento fueron trie, QuickSort e IQS. El tamaño de la base de datos fue 10.000 elementos y la dimensión 128.

En las figuras 5.3(a) y 5.3(b) se muestra el efecto en el tiempo de procesamiento al incrementar el número de permutantes/pivotes (32 y 128, (a) y (b), respectivamente). Nótese que resolver el mismo problema usando la técnica de los algoritmos basados en pivotes es más lento que usando la técnica basada en permutaciones. Esta diferencia se incrementa al usar más permutantes/pivotes. IQS tiene un mejor desempeño que el QuickSort cuando se quiere una fracción muy pequeña de los elementos ordenados. En las figuras esta fracción es aproximadamente el 4 % de la base de datos para 32 pivotes/permutantes. Si la fracción a compararse es más grande, entonces usar el QuickSort es la mejor alternativa.

En las figuras 5.4 y 5.5 se compara el desempeño de diferentes alternativas que se han discutido para ordenar la base de datos y recuperar sólo el primer porcentaje de los datos ordenados. Para este experimento se usaron 100.000 vectores aleatorios en dimensión 128 y 32 permutantes. En esta figura se muestran los tiempos de procesamiento usando dos máquinas distintas: en el lado izquierdo usando un procesador PowerPC G4 a 1.33 GHz con 768 MB RAM y 512 KB L2 de

cache, con un bus de 167 MHz y corriendo Mac OS X 10.4.7 ; en el lado derecho, un Procesador Xeon Intel a 2.60 GHz con 3 GB RAM y 512 KB de cache, corriendo Red Hat Linux 3.2.2-10. En ambos casos se usó `gcc` con la opción de optimización `-O9`.

Como se puede apreciar en las figuras 5.4 y 5.5 la alternativa basada en el trie es mucho más lenta que las otras por las razones explicadas anteriormente. Este experimento ayuda a ilustrar el problema del trie al tener que procesar una gran mayoría de nodos internos antes de recuperar la *primera* hoja. Por ejemplo, con $k = 32$ permutantes, se procesaron el 51 % de los nodos internos antes de comparar la primer hoja; en el caso de $k = 128$ este porcentaje fue del 77 %.

En las figuras 5.4 y 5.5 es interesante observar que la comparación entre los algoritmos de ordenamiento depende del procesador. IQS es una alternativa eficiente en ambos casos. La comparación entre BucketSort y QuickSort depende fuertemente de la máquina.

Los resultados muestran que, a pesar de haber empleado una técnica que evite el procesamiento de la base de datos completa, el método de ordenamiento más rápido en la práctica continúa siendo un procesamiento secuencial. En la práctica se recorrerá sólo una fracción de la base de datos, en este experimento se discute solamente el costo extra de CPU.

5.3. Experimentación

Independientemente del tipo de alternativa usada para ordenar de la base de datos por valor creciente de S_ρ, los elementos serán comparados en el mismo orden (comparaciones externas). La diferencia entre los métodos de ordenamiento es el tiempo extra de CPU. En esta sección estudiaremos el desempeño de la técnica basada en permutaciones.

En esta sección presentaremos el desempeño de nuestra propuesta de orden en la búsqueda usando la técnica basada en permutaciones en diferentes espacios métricos tales como cubo unitario y base de datos de documentos.

Cubo Unitario

Para estos experimentos se utilizó un conjunto sintético distribuido de manera uniforme en el cubo unitario (descrito en la sección 2.6.1). El tamaño de la base de datos fue de 10.000 objetos y las consultas buscaban recuperar el 0,05 % de la base de datos. Otro parámetro de nuestro algoritmo es el número de permutantes, k. En este conjunto de experimentos, se usó $k = 128$ y $k = 256$. Las dimensiones de los conjuntos variaron entre 128 y 1.024.

Nuestra técnica fue comparada contra un algoritmo clásico de pivotes que usa la misma cantidad de pivotes, aún cuando esto representa usar 4 veces más memoria que la ocupada por nuestro algoritmo. Si restringiésemos ambos algoritmos a ocupar la misma cantidad de memoria,

nuestros resultados comparativos serían aún mejores.

En el caso del algoritmo basado en pivotes, para ordenar los elementos por promisoriedad, calculamos $\forall p \in \mathbb{P},\ d(p,q)$ y $\forall u \in \mathbb{U}$,

$$L_\infty(q,u) = \max_{p \in \mathbb{P}} |d(q,p) - d(p,u)|$$

que es la cota mínima de la distancia entre un elemento y la consulta, si se usan como vectores $(d(u,p_1),\ldots,d(u_{p_k}))$ y $(d(q,p_1),\ldots,d(q_{p_k}))$ sta distancia fue explicada en la sección 2.2.1. La figura 5.6 muestra la comparación entre ambas técnicas: basada en pivotes (explicada en la página 27 y usada anteriormente en [BN02]) y basada en permutaciones. Usamos permutaciones con 128 (5.6(a)) y 256 elementos (5.6(b)). El eje *x* corresponde a la fracción de la base de datos examinada, y el eje *y* al porcentaje de la respuesta recuperada. Las series con la palabra *piv* se refieren al algoritmo clásico basado en pivotes.

El porcentaje de recuperación en una consulta está dado por:

$$\%\ \text{Recuperación} = \frac{|R_{Alg} \cap R_{Opt}|}{|R_{Opt}|} \times 100 \tag{5.3}$$

donde:
R_{Alg} es el conjunto respuesta entregado por el algoritmo
R_{Opt} es el conjunto respuesta correcto. En el caso de consultas de los K vecinos más cercanos $|R_{Opt}| = K$. En el caso de consultas de rango $(q,r)_d$, se cumple $\forall u \in R_{Opt},\ d(q,u) \leq r$.

Para la mayoría de aplicaciones basadas en búsquedas por proximidad, recuperar el 90 % de las respuestas es un resultado aceptable. En el caso del algoritmo basado en permutantes este porcentaje puede obtenerse si se examina el 10 % de la base de datos con $k = 128$ en dimensión 128. En el caso del algoritmo pivotero es necesario revisar un 60 % de la base de datos para obtener ese porcentaje. Observe que, a medida que la dimensión crece, para obtener la misma precisión es necesario revisar una fracción más grande de la base de datos. En ambos algoritmos ocurre lo mismo, aunque no en la misma proporción.

Por otro lado, es posible revisar un porcentaje menor de la base de datos si se incrementa el número de permutantes k. En la figura 5.6(b), se usaron 256 permutantes y examinando el 10 % de la base de datos se recuperó el 99 % de los resultados. Observe que el algoritmo basado en pivotes necesita comparar el 85 % de la base de datos para recuperar el mismo porcentaje.

En las figuras 5.6(a) y 5.6(b) se puede ver que en dimensión 1024 la técnica basada en pivotes no aporta nada acerca del orden en que deben revisarse los elementos. El resultado es semejante a una comparación en orden aleatorio de los elementos.

En la figura 5.7 comparamos otras medidas de similaridad entre permutaciones como predictores de cercanía, como Spearman Footrule y Kendall Tau. Se puede ver que Spearman Rho y Kendall Tau tienen el mismo desempeño, pero el primero es menos costoso de calcular. Por otro lado, Spearman Footrule es más barato de evaluar que los otros dos pero su desempeño en cuanto a proximidad es más pobre. El rango de la figura en el eje x es para resaltar la comparación entre las medidas de similaridad, ya que los 256 permutantes corresponden al 2,5 % de la base de datos, y hasta el 3,5 % no existen grandes diferencias entre ellas.

AESA [Vid86] está considerado como el mejor algoritmo exacto para la búsqueda por proximidad en términos de comparaciones de distancia. Este algoritmo usa la métrica L_1, esto es $L_1(q,u) = \sum_{p\in\mathbb{P}} |d(q,p) - d(p,u)|$ como un oráculo para seleccionar el siguiente mejor candidato como respuesta. De aquí que una comparación obligada es usar L_1 para ordenar la base de datos en lugar de L_∞. Sin embargo, vale la pena probar ambas métricas L_1 y L_∞ para comprobar su desempeño. En la figura 5.8, se puede observar la comparación entre ambas distancias L_1 y L_∞. En la primera parte (recorriendo entre el 30 % y el 50 % de la base de datos) la distancia L_1 tuvo un mayor porcentaje de recuperación comparado con L_∞, para dimensión 128. A medida que se revisa un porcentaje mayor de la base de datos, L_∞ pasa a ser mejor oráculo. Se observa el mismo comportamiento en todas las dimensiones consideradas. Nótese que, de cualquier manera, los resultados están muy lejos de los obtenidos con nuestra técnica.

En el apéndice A, sección A.1 se muestra otra forma distinta de comparar el desempeño de la técnica basada en permutaciones con respecto a L_1 y L_∞. La comparación se realiza por medio de una nube de puntos representando la posición dada por el orden de los diferentes criterios usados y la posición dada por la distancia real.

5.3.1. Documentos TREC

En esta sección usamos un subconjunto de la colección del TREC-3 (descrita en la sección 2.6.3) para comparar el desempeño de nuestra técnica contra el mejor algoritmo probabilístico reportado [BN02]. La base de datos consiste de 25.960 documentos y de éstos seleccionamos 1.000 consultas de manera aleatoria, recuperando el 0,035 % de la base de datos. En la figura 5.9 se puede ver que la técnica de las permutaciones compara el 2 % de la base de datos para recuperar el 90 % de las respuestas, mientras que el algoritmo basado en pivotes necesita revisar casi el 18 % de la base de datos para alcanzar el mismo porcentaje de la respuesta.

En [BN02] se propone otro criterio para ordenar la base de datos llamado *Dynamic Beta*, el cual necesita revisar, al menos, el 11 % de la base de datos para alcanzar el mismo 90 % de las respuestas. Dynamic Beta está basado en un algoritmo de particiones compactas (ver sección 3.1.2) cuyo criterio de selección consiste en reducir el radio de búsqueda en un factor β dependiendo de la zona, es decir $\beta(d(q,p) - cr(p))$, donde $1/\beta \in [0 \dots 1]$ y $mrc(p)$ es el radio de cobertura de

una zona p. Básicamente, la idea es reducir el radio en aquellas zonas con un radio de cobertura grande, esto es, $\beta = 1/\left(1 - \frac{mrc(p)}{mcr}\right)$, donde mrc es el radio de cobertura máximo de todas las zonas.

En el apéndice A, sección A.2 se compara el desempeño de la técnica basada en permutaciones con respecto a L_1 y L_∞ por medio de una nube de puntos representando la posición dada por el orden de los diferentes criterios usados y la posición dada por la distancia real.

5.3.2. Comparación entre Exactos y Probabilísticos

En la figura 5.10 se muestra la comparación entre nuestro algoritmo probabilístico y distintos métodos exactos (AESA, Lista de Clusters y un algoritmo pivotero, descritos en la sección 3.1.1) en un espacio de distribución uniforme en el cubo unitario (véase la sección 2.6.1). Como se puede ver en la figura, los algoritmos probabilísticos son una buena alternativa cuando la dimensión de los datos aumenta. Nótese que usando 128 permutantes es posible obtener un 90 % de la respuesta calculando la distancia hacia menos del 10 % de la base de datos, mientras que las técnicas exactas deben comparar el 100 % de la base de datos para reportar las respuestas correctas, esto es un resultado esperado en algoritmos exactos debido al problema de la maldición de la dimensionalidad (véase la sección 2.3).

Finalmente, una vez que hemos analizado el buen desempeño de la técnica basada en permutaciones podemos decir que los resultados experimentales han mostrado que *ordenar* la base de datos usando la semejanza entre permutaciones es un buen predictor de cercanía entre elementos. De manera indirecta, todos los algoritmos para espacios métricos funcionan a base de un predictor de cercanía (que hasta ahora ha sido L_1 o L_∞ generalmente) tratando de encontrar los elementos más cercanos para rápidamente filtrar el resto. En lo sucesivo llamaremos *ordenamiento basado en permutaciones*, a la técnica aquí presentada. Y generalmente será comparada contra ordenar la base de datos de acuerdo a L_1 y a L_∞ por la relevancia de estas métricas para los algoritmos existentes.

5.4. Selección de Permutantes

Hasta ahora los permutantes han sido seleccionados de manera aleatoria. En esta sección estudiaremos algunas opciones de selección de permutantes, analizando cómo influyen en el desempeño del algoritmo probabilístico mostrado en este capítulo.

Las heurísticas propuestas consisten en seleccionar como permutantes aquellos que minimicen o maximicen el valor de Spearman Rho, respecto a los permutantes que han sido seleccionados.

En el caso de seleccionar como permutantes los que minimicen el Spearman Rho,

significa que éstos estarán muy cerca entre ellos y con esto, las permutaciones serán más sensibles a pequeños cambios de posición en el espacio. Formalmente, seleccionaremos aquel $p \in \mathbb{U}$ tal que:

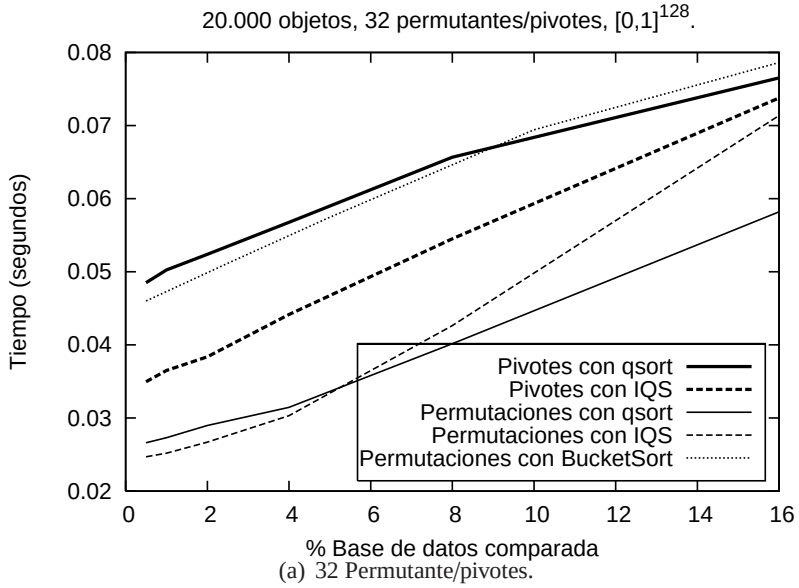
$$p = \arg \min_{p_i \in \mathbb{U}-\mathbb{P}} \left(\sum_{p_j \in \mathbb{P}} S_\rho(p_i, p_j) \right)$$

Si se seleccionan aquellos que maximizan el valor del Spearman Rho significa que los permutantes serán muy distintos entre ellos y podría ayudar a que pequeños cambios en una permutación indiquen mayor distancia entre elementos, sobre todo si existen agrupamientos en los datos.

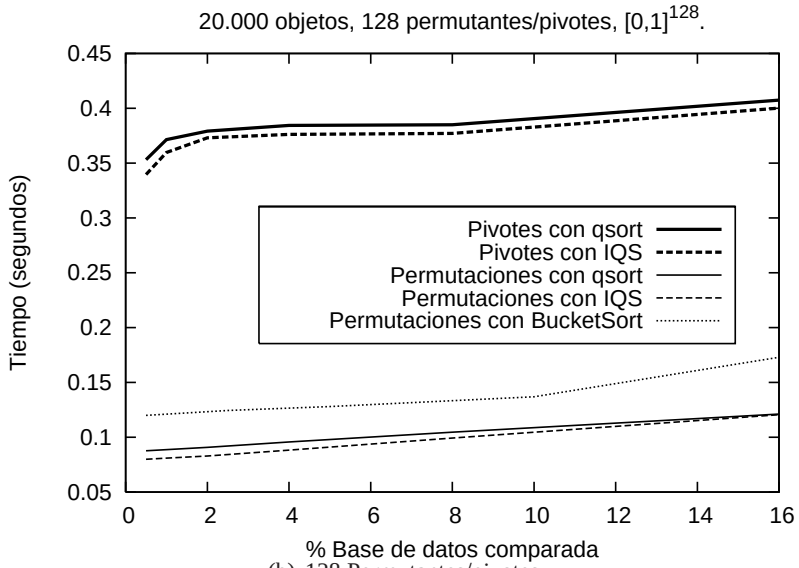
$$p = \arg \max_{p_i \in \mathbb{U}-\mathbb{P}} \left(\sum_{p_j \in \mathbb{P}} S_\rho(p_i, p_j) \right)$$

En la figura 5.11 se muestra el desempeño de la selección de permutantes usando tres métodos distintos: de manera aleatoria, buscando el mínimo o el máximo valor de Spearman Rho. El espacio empleado fueron vectores distribuidos de manera uniform en el cubo unitario (veáse la sección 2.6.1). En la figura 5.11(a) se presenta el desempeño para dimensión 128, y en la 5.11(b) para 256. En ambos casos el número de permutantes fueron 32, 64 y 128 (con líneas de más delgadas a más gruesas al aumentar el número de permutantes). Note que en las diferencias son mínimas entre una técnica y otra, aunque el mejor desempeño se tiene cuando los permutantes seleccionados son los que minimizan el valor del Spearman Rho.

La dimensión de los datos también afecta el desempeño de los permutantes, pues en dimensiones altas se minimiza la diferencia entre usar un permutante u otro. En particular, en estos experimentos se aprecia que usando pocos permutantes es más visible el efecto de cómo fueron seleccionados, aunque sea pequeña la contribución.

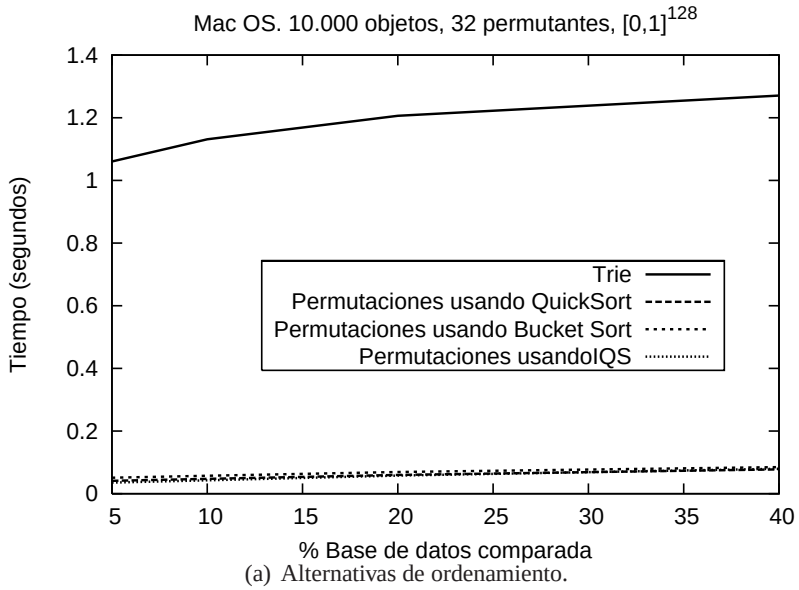


(a) 32 Permutante/pivotes.

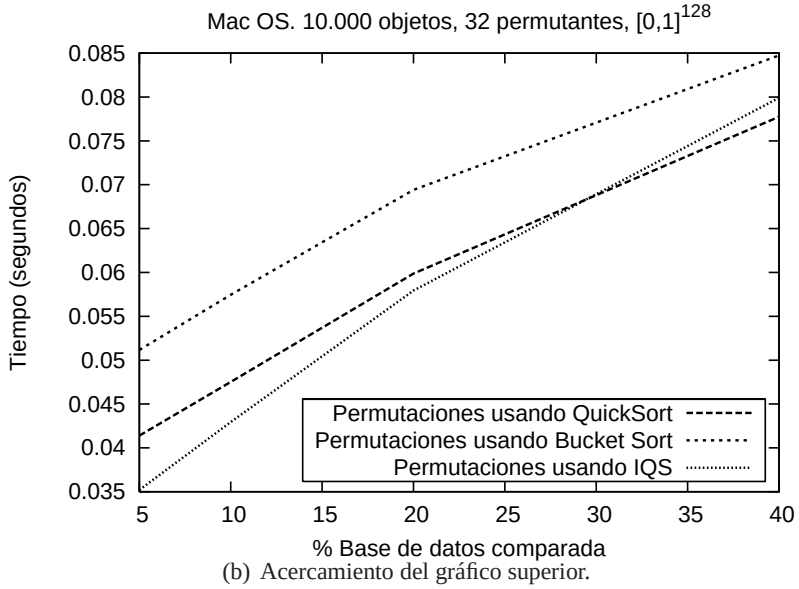


(b) 128 Permutantes/pivotes.

Figura 5.3: Tiempo de procesamiento de los algoritmos propuestos usando dos métodos de ordenamiento (QuickSort e IQS) y variando el número de permutantes/pivotes, arriba 32, abajo 128. El tamaño de la base de datos fue 10.000 elementos y la dimensión 128.

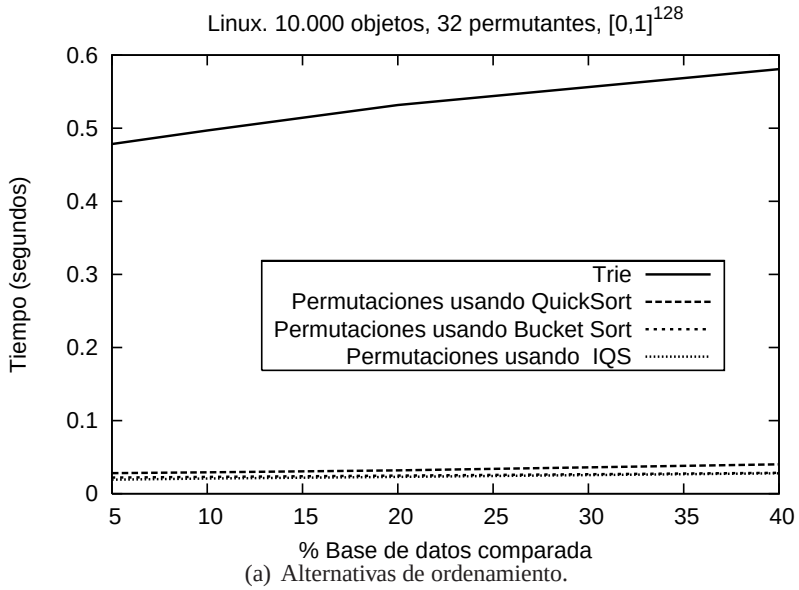


(a) Alternativas de ordenamiento.

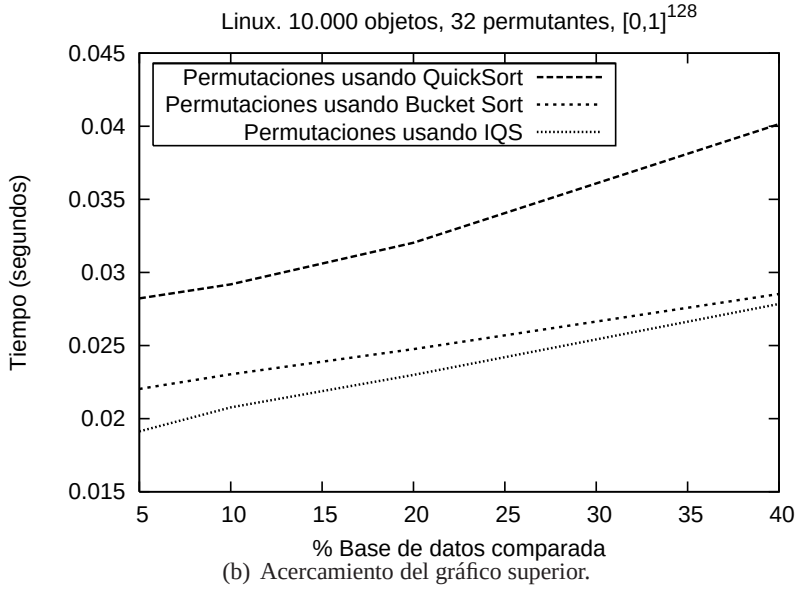


(b) Acercamiento del gráfico superior.

Figura 5.4: Tiempo de procesamiento para diferentes algoritmos para obtener el primer porcentaje de la base de datos. El procesador usado fue un PowerPC. En la parte inferior se muestra un acercamiento en los algoritmos más rápidos.

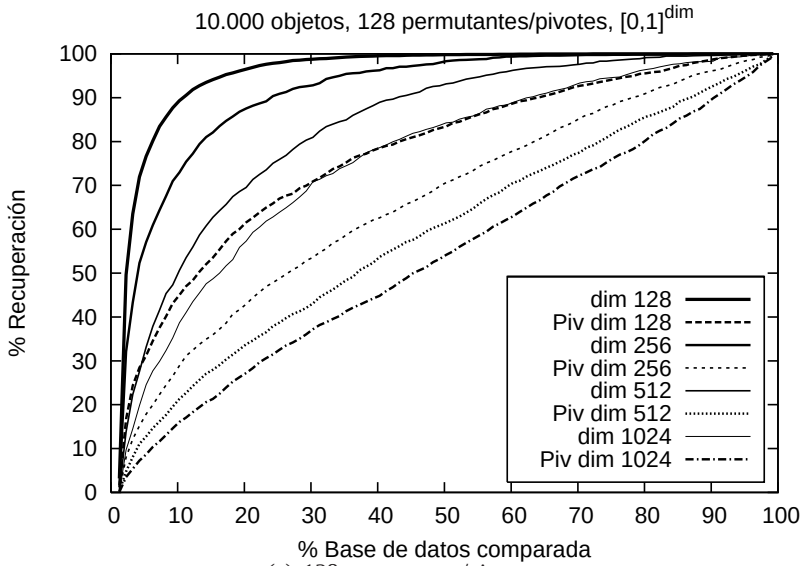


(a) Alternativas de ordenamiento.

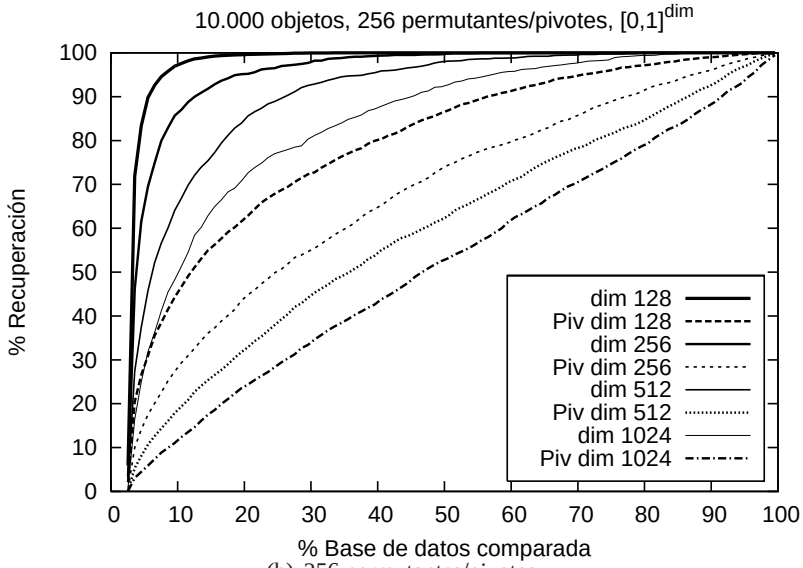


(b) Acercamiento del gráfico superior.

Figura 5.5: Tiempo de procesamiento para diferentes algoritmos para obtener el primer porcentaje de la base de datos. El procesador usado fue un Intel. En la parte inferior se muestra un acercamiento en los algoritmos más rápidos.



(a) 128 permutantes/pivotes.



(b) 256 permutantes/pivotes.

Figura 5.6: Desempeño de los algoritmos basados en ordenamientos (permutantes y L_∞) en dimensiones altas, para 10.000 objetos y la consulta busca recuperar el 0,05 % de la base de datos.

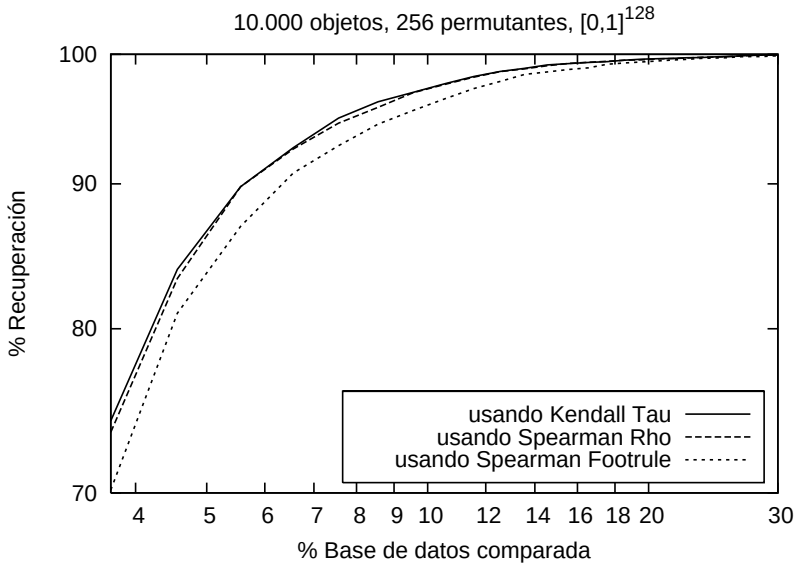


Figura 5.7: Desempeño de diferentes medidas de similaridad entre permutaciones: *Spearman Footrule, Spearman Rho* y *Kendall Tau*, dimensión 128, 256 permutantes y 10.000 objetos, la consulta busca recuperar el 0,05 % de la base de datos (escala logarítmica).

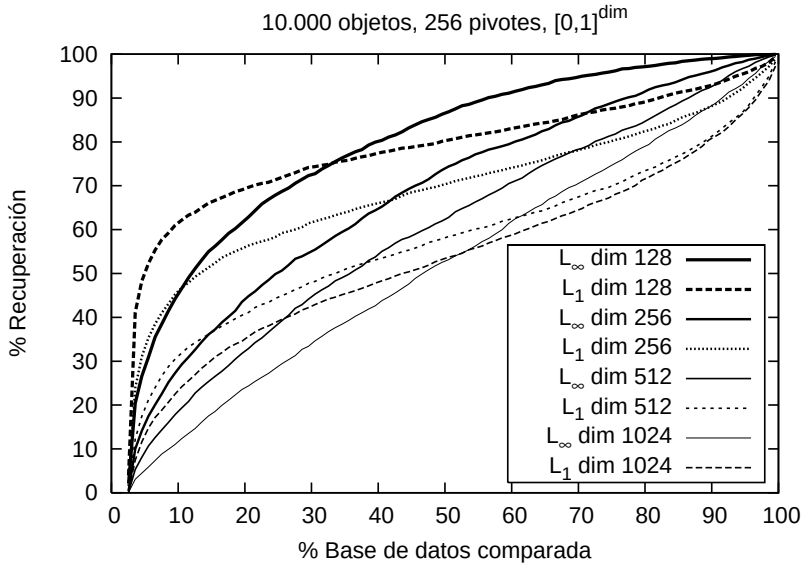


Figura 5.8: Comparación entre las métricas Minkowski L_1 y L_∞ para ordenar la base de datos en un algoritmo basado en pivotes. 256 pivotes, 10.000 objetos, la consulta busca recuperar el 0,05 % de la base de datos.

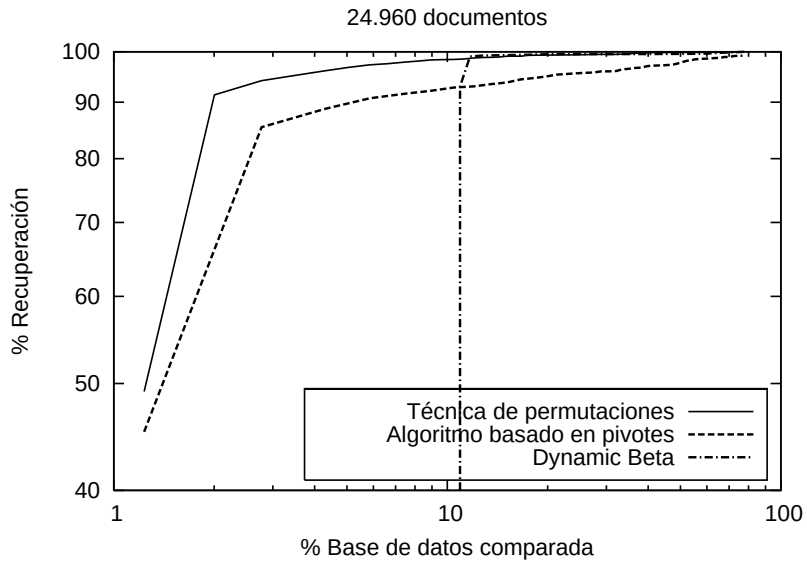


Figura 5.9: Comparación de nuestra técnica en el espacio de documentos (colección TREC-3 de 24.960 documentos) con 64 pivotes/permutantes. La consulta busca recuperar el 0,035 % de la base de datos por consulta. El algoritmo basado en pivotes fue usado con L_∞ (escala logarítmica).

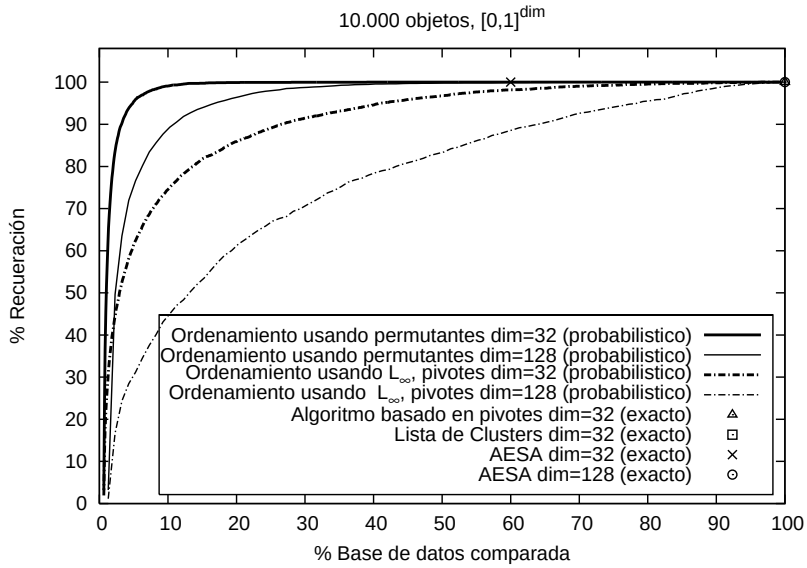
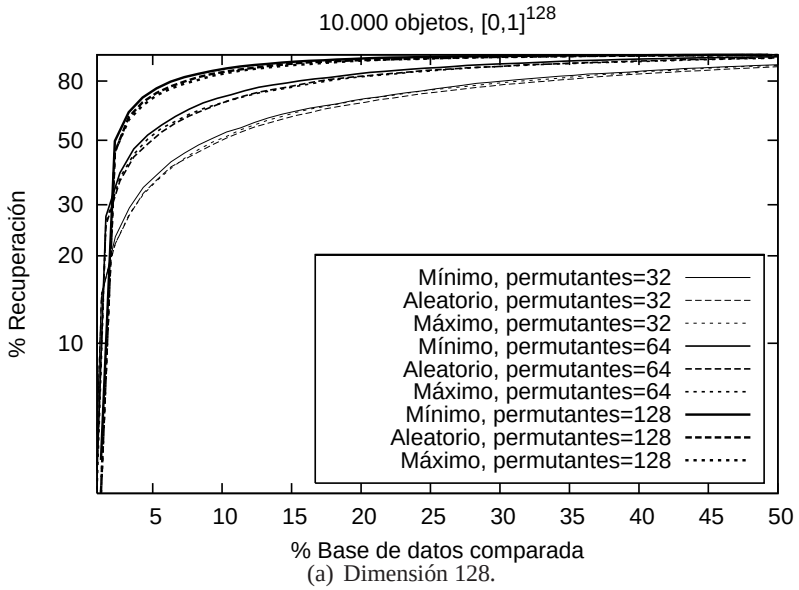
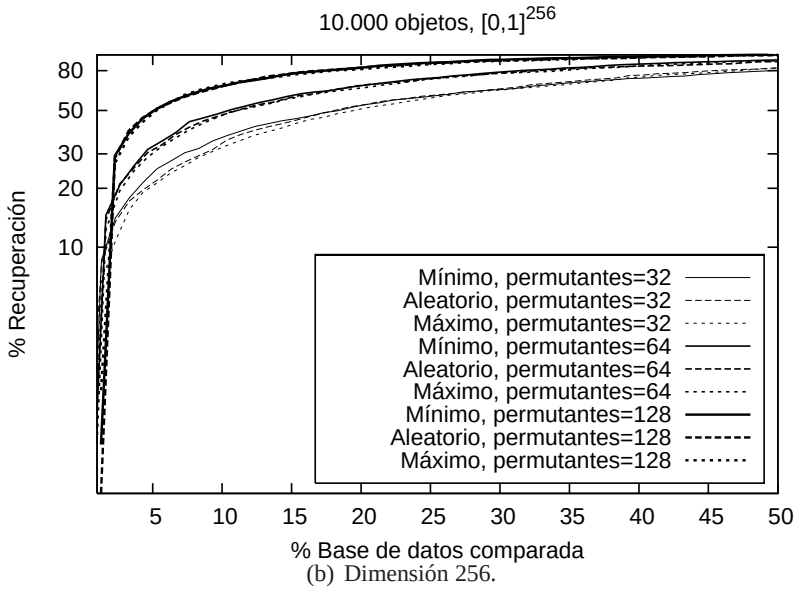


Figura 5.10: Comparación entre algunos algoritmos exactos y los probabilísticos presentados. El tamaño de la base de datos fue 10.000 elementos, se emplearon 128 pivotes/permutantes para la dimensión 128 y 64 pivotes/permutantes para la dimensión 32. Todos los algoritmos exactos (excepto AESA para dimensión 32) están en el mismo punto en la figura, pues recuperaron el 100 % de la respuesta pero debieron comparar el 100 % de la base de datos.



(a) Dimensión 128.



(b) Dimensión 256.

Figura 5.11: Diferentes heurísticas para la selección de permutantes. El tamaño de la base de datos fue 10.000, y se recuperó el 0,05 % de la base de datos.

Parte III

APLICACIONES

Capítulo 6

Nuevo Algoritmo Exacto: iAESA

En este capítulo mostraremos cómo la técnica de las permutaciones presentada en el capítulo 5, además de ser un algoritmo muy competitivo por sí mismo, puede mejorar el desempeño de otros algoritmos. En particular, aplicando nuestra técnica se logró reducir el número de evaluaciones de distancia hecho por AESA. Este último es el algoritmo exacto que realiza, la menor cantidad de comparaciones para las búsquedas por proximidad en espacios métricos. En este capítulo presentaremos reducciones de hasta un 35 % en las comparaciones de distancia realizadas por AESA.

6.1. AESA

El algoritmo que realiza la menor cantidad de comparaciones para resolver la búsqueda por proximidad es, sin duda, AESA (Approximating and Eliminating Search Algorithm), el cual fue presentado en 1986 [Vid86]. El índice que AESA utiliza es una matriz con cada distancia $d(u,v)$, $\forall u,v \in \mathbb{U}$, esto es $O(n^2)$ distancias. Esta matriz se usa para descartar elementos usando la desigualdad del triángulo. Durante el proceso de búsqueda, se escoge un elemento ("pivote") del conjunto de candidatos, este será comparado contra la consulta. AESA propone un método para seleccionar el siguiente elemento (pivote) más promisorio [a] a compararse contra la consulta. El algoritmo se describe en la sección 6.1.1.

A pesar de que $O(n^2)$ pueda ser una gran cantidad de memoria, existen aplicaciones con bases de datos lo suficientemente pequeñas (hasta pocos miles de objetos) donde es posible manejar todas las $O(n^2)$ distancias. Para este tipo de aplicaciones, AESA es todavía una solución práctica y la que hasta ahora calcula el menor número de evaluaciones de distancia.

En el caso de bases de datos grandes, donde no es posible almacenar $O(n^2)$ distan-

[a] promisorio a ser el vecino más cercano

cias, es posible combinar una técnica de particionamiento sobre la base de datos (por ejemplo: LC, BST, etc., descritos en el capítulo 3) y aplicar AESA en cada partición [Fre07], pues cada particionamiento guarda sólo un subconjunto de $\mathbb{U}$ y generalmente tendrá pocos elementos.

El núcleo de AESA consiste en cómo escoger los buenos candidatos a compararse contra la consulta, puesto que todos los elementos en la base de datos pueden ser pivotes. Un buen criterio de selección debe aproximarse rápidamente a la consulta, pues de esta manera logrará filtrar más elementos.

6.1.1. Proceso de Búsqueda de AESA

AESA soluciona consultas del vecino más cercano en $O(1)$ comparaciones, seleccionando aquel candidato $u \in \mathbb{U}$ más promisorio para comparar contra q (comparación interna). Con la distancia calculada $d(q,u)$ se filtra la mayor cantidad posible de candidatos de $\mathbb{U}$, y se repite el proceso hasta que todos los candidatos hayan sido comparados o descartados, por lo que no existen comparaciones externas. AESA propone seleccionar el siguiente pivote a comparar contra la consulta como aquel u que minimiza la siguiente ecuación, la cual corresponde a la métrica L_1:

$$D(u) = L_1 = \sum_{p \in \mathbb{P}} |d(u,p) - d(p,q)|, \tag{6.1}$$

donde $\mathbb{P}$ son aquellos pivotes que ya han sido comparados contra q. El objetivo de minimizar $D(u)$ es utilizar pivotes cercanos a q, pues filtran mejor ya que estiman mejor la distancia entre la consulta y los elementos en la base de datos. Aunque en [Vid94] proponen usar la métrica L_∞ en lugar de L_1, basandonos en la figura 5.8 consideramos que L_1 obtiene resultados superiores a L_∞ entre los primeros elementos de la base de datos, donde AESA generalmente trabaja. Básicamente, el algoritmo que resuelve consultas del vecino más cercano $NN(q)_d$ se resume en cinco pasos.

1. **Inicialización**. Se inicializan vacíos el conjunto de pivotes $\mathbb{P}$ y los elementos filtrados $\mathbb{F}$. Sea $D(u) \leftarrow 0$ para $u \in \mathbb{U}$, $Dmax(u) \leftarrow L_\infty \leftarrow 0$ (cota mínima, ecuación 2.4) y $r \leftarrow \infty$. Los pasos 2-5 se repiten hasta que cada elemento en $\mathbb{U}$ ha sido comparado o filtrado.

2. **Aproximación**. En este paso se selecciona un nuevo pivote p de acuerdo a la ecuación 6.1. Esto es $p \leftarrow argmin_{u \in \mathbb{U}-\mathbb{F}-\mathbb{P}}\ D(u)$.

3. **Evaluación de distancia**. El elemento p es comparado contra la consulta q. El nuevo p es agregado al conjunto de pivotes $\mathbb{P}$.

4. **Actualizar el vecino más cercano**. Si $d(q,p) < r$, se actualizan r y el vecino más cercano ($p*$) hasta el momento. Cada objeto u en $\mathbb{U} - \mathbb{F} - \mathbb{P}$ actualiza el criterio de aproximación de acuerdo a las ecuaciones 6.1 y 2.4. Esto es $D(u) \leftarrow D(u) + |d_{u,p} - d(p,q)|$ y

$Dmax(u) \leftarrow \text{máx}(Dmax(u), |d_{u,p} - d(p,q)|)$, donde $d_{u,p}$ es una entrada de la matriz de distancias correspondiente a $d(p,u)$.

5. **Eliminación**. Aquellos $u \in \mathbb{U} - \mathbb{F} - \mathbb{P}$ tales que $Dmax(u) > r$ son descartados usando la desigualdad triangular. Aquellos elementos filtrados en este paso se agregan al conjunto $\mathbb{F}$. El proceso continúa en el paso 2.

El pseudocódigo de este proceso se muestra en el algoritmo 7. El proceso de consultas por rango puede ser implementado de manera similar, manteniendo fijo el valor del radio r y reportando cada p encontrado que cumpla $d(p,q) \leq r$.

En el caso de búsqueda de KNN donde $K > 1$, una propuesta es mantener los vecinos más cercanos en una cola de prioridad (heap) [Wil64]. En cada distancia calculada se verifica si ésta es menor al radio de los vecinos más cercanos almacenados en el heap (sólo comparando el mayor del heap). De ser así, se reduce el radio y el elemento pasa a formar parte de los K vecinos más cercanos almacenados en el heap, sacando al más lejano del heap, esto toma $O(\log K)$ operaciones en el heap. En [AJE95, JVA98] se presentan técnicas para resolver K vecinos más cercanos con AESA ($K-AESA$). Los autores mantienen una lista ordenada de los vecinos más cercanos, donde para insertar un nuevo elemento requieren $O(K)$ operaciones.

Algoritmo 7 AESA

```
1:  Sea ℙ ← ∅ conjunto de pivotes                          // Inicialización
2:  Sea 𝔽 ← ∅ conjunto de elementos filtrados
3:  r ← ∞
4:  For u ∈ 𝕌, D(u) ← 0, Dmax(u) ← 0
5:  while 𝕌 ≠ ℙ ∪ 𝔽 do
6:     p ← argmin_{u∈𝕌−ℙ−𝔽} D(u)                           // Aproximación
7:     ℙ ← ℙ ∪ {p}
8:     d ← d(q, p)                  //Cálculo de distancia
9:     if d < r then
10:       r ← d                     // Actualización del vecino más cercano
11:       p* ← p
12:    end if
13:    for u ∈ 𝕌 − ℙ − 𝔽 do
14:       Dmax(u) ← máx(Dmax(u), |d − d_{u,p}|)
15:       if Dmax(u) > r then
16:          𝔽 ← 𝔽 ∪ {u}                                  // Eliminación
17:       else
18:          D(u) ← D(u) + |d − d_{u,p}|
19:       end if
20:    end for
21: end while
22: return p*                        // regresa el vecino más cercano
```

6.2. iAESA

La sección 6.1.1 muestra que el único punto heurístico en AESA es el criterio de aproximación (paso 2). En esta sección proponemos un método diferente para seleccionar el siguiente pivote. Éste consiste en usar las permutaciones introducidas en el capítulo 5. Seleccionaremos como siguiente pivote al elemento cuya permutación sea la más similar a la permutación de la consulta.

6.2.1. Proceso de Búsqueda de iAESA

El nuevo algoritmo de búsqueda basado en AESA consiste en remplazar el criterio de aproximación (ecuación 6.1), por la similaridad entre permutaciones (métrica del Footrule, ecuación 5.2, página 65). Esto es, en lugar de elegir el que minimice $D(u)$, se usará el que minimice la diferencia entre permutaciones $F(\Pi_q, \Pi_u)$. Las permutaciones (explicadas en el capítulo 5) se forman con los pivotes usados en cada iteración, por lo tanto las permutaciones se actualizan cada vez que se selecciona un pivote.

El algoritmo es prácticamente el mismo que el de AESA: se inicializa a cero el valor del Footrule de cada elemento; cada vez que se escoge un nuevo pivote p se actualiza la permutación de todos los elementos y de la consulta; se recalcula el valor del Footrule; y el resto del proceso es el mismo. El costo de la búsqueda es el mismo que AESA más el costo por mantener actualizadas las permutaciones en cada inserción.

En este algoritmo se usó Footrule y no Spearman Rho porque este último tiene ligeramente mayor tiempo de procesamiento, además de que los resultados experimentales no mostraron una mejora importante. En la figura 8 se muestra el pseudocódigo de iAESA. Note que las líneas distintas a AESA son: 6, 8, 19 y 20.

6.2.2. Comparando AESA con iAESA

En la figura 6.1 se compara AESA contra iAESA usando el mismo ejemplo que el mostrado en [Vid86]. El ejemplo recupera el vecino más cercano. En la figura (lado izquierdo) los objetos son $p_1, \ldots, p_7$ y q es la consulta; las flechas continuas son los sumandos de la ecuación 6.1 para cada elemento usando p, o sea $|d(p,q) - d(p,u)|$, $\forall p \in \mathbb{P}$; las flechas discontinuas indican el orden de selección del siguiente pivote (etiquetados paso-1, paso-2, paso-3); el círculo y los semicírculos son las distancias desde un pivote a la consulta, y son etiquetados con el paso en que se midió esa distancia.

En el ejemplo, la secuencia de selección, de acuerdo a AESA, fue la siguiente: primero se selecciona de manera aleatoria p_1, y el radio de la consulta al vecino más cercano es infinito.

Algoritmo 8 iAESA

1: Sea $\mathbb{P} \leftarrow \emptyset$ conjunto de pivotes // Inicialización
2: Sea $\mathbb{F} \leftarrow \emptyset$ conjunto de elementos filtrados
3: $r \leftarrow \infty$, $\Pi_q \leftarrow <>$
4: For $u \in \mathbb{U}$, $F(u) \leftarrow 0, \Pi_u \leftarrow <>, Dmax(u) \leftarrow 0$
5: **while** $\mathbb{U} \neq \mathbb{P} \cup \mathbb{F}$ **do**
6: $p \leftarrow \text{argmin}_{u \in \mathbb{U}-\mathbb{P}-\mathbb{F}} F(u)$ // Aproximación
7: $\mathbb{P} \leftarrow \mathbb{P} \cup \{p\}$
8: insertar p en Π_q
9: $d \leftarrow d(q,p)$ //Cálculo de distancia
10: **if** $d < r$ **then**
11: $r \leftarrow d$ // Actualización del vecino más cercano
12: $p* \leftarrow p$
13: **end if**
14: **for** $u \in \mathbb{U} - \mathbb{P} - \mathbb{F}$ **do**
15: $Dmax(u) \leftarrow \text{máx}\,(Dmax(u), |d - d_{u,p}|)$
16: **if** $Dmax(u) > r$ **then**
17: $\mathbb{F} \leftarrow \mathbb{F} \cup \{u\}$ // Eliminación
18: **else**
19: insertar p en Π_u
20: $F(u) \leftarrow F(\Pi_q, \Pi_u)$
21: **end if**
22: **end for**
23: **end while**
24: **return** $p*$ // regresa el vecino más cercano

Con la distancia calculada se actualiza el vecino más cercano (NN) y el radio a éste ($r \leftarrow d(p_1, q)$), también se descartan algunos elementos que, de acuerdo a la desigualdad triangular, no son parte de la respuesta, aunque en este ejemplo el radio de la consulta aún es muy grande para descartar algún elemento. El resto de los elementos se ordena por el criterio de proximidad, y se selecciona aquél con menor $D(u)$, que en nuestro ejemplo es p_2 (paso-1). Se calcula $d(p_2, q)$, se actualiza el NN y ahora p_2 es el NN. El radio de la consulta se redujo a $d(p_2, q)$. El proceso se repite hasta que todos los elementos hayan sido revisados o descartados. En el ejemplo, el resto del proceso está indicado por el paso 2 y 3. Una vez seleccionado el p_4 el resto de la base de datos puede ser filtrado usando la desigualdad del triángulo y la matriz de distancias almacenadas en el índice (estos son p_5, p_6 y p_7).

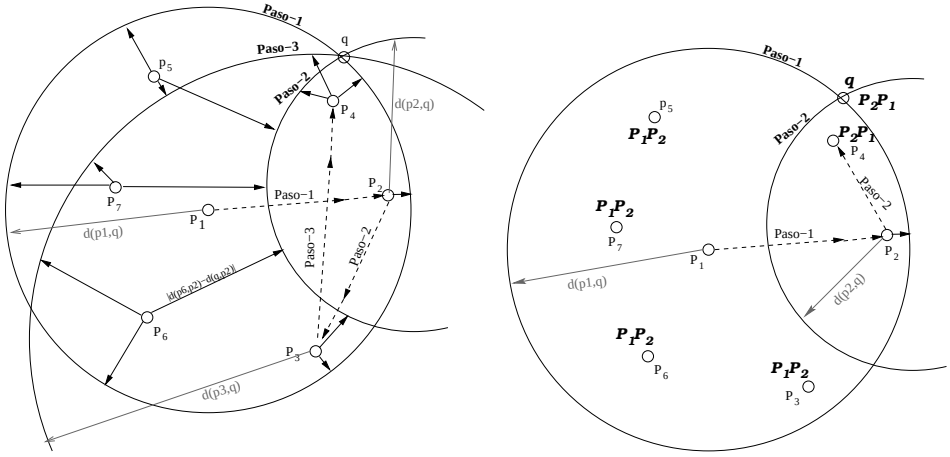
Nótese cómo a medida que los elementos se aproximan más a la consulta el radio se reduce, y por lo tanto se pueden filtrar más elementos. De aquí que sea de gran relevancia un buen criterio de proximidad para localizar rápidamente los elementos más cercanos a q.

Por otro lado, en el proceso de iAESA (lado derecho de la figura 6.1), se inicia seleccionando p_1, se mide la distancia entre $d(p_1, q)$, al igual que en AESA. La distancia $d(p_1, q)$ es usada para reducir el radio al NN y para filtrar los elementos que de acuerdo a la desigualdad del triángulo no puedan ser parte de la respuesta (aunque en este ejemplo no pueda ser filtrado ninguno). Con el resto de la base de datos se empieza a formar la permutación (inicialmente $\{p_1\}$). Nótese que el segundo permutante puede ser cualquiera pues toda la base de datos tiene la misma permutación. Se selecciona otro elemento, en este caso p_2 (paso 1), se mide la distancia a la consulta $d(p_2, q)$ y se reduce el radio al NN a $d(p_2, q)$. El proceso se repite: filtrar los elementos posibles, actualizar la permutación insertando a p_2 en los sobrevivientes y la consulta. Con dos permutantes ya es posible seleccionar permutaciones con mayor semejanza a la consulta. En este ejemplo, p_4 tiene la misma permutación que $\Pi_q = p_2, p_1$, por lo tanto p_4 es el siguiente y último permutante escogido, pues el resto de la base de datos es descartado. En este ejemplo iAESA usa los pivotes (p_1, p_2, p_4), uno menos que AESA. Es importante notar que no siempre los pivotes escogidos en un algoritmo y otro serán iguales.

La complejidad de tiempo de CPU de AESA es $\sum_1^{|\mathbb{P}|} i * n(i)$, donde $n(i)$ es la cantidad de sobrevivientes a la i-ésima iteración. Para el caso de iAESA la complejidad de tiempo de CPU es $\sum_1^{|\mathbb{P}|} i^2 * n'(i)$, donde $n'(i)$ se refiere a la cantidad de sobrevivientes al usar la técnica de los permutantes. Si los permutantes hicieran un trabajo muy bueno, podría ser que un $n'(i)$ mucho menor que $n(i)$ compensara el i^2 frente al i.

6.2.3. Combinando AESA e iAESA

Una segunda modificación a AESA es combinar ambos criterios: el de AESA y el de iAESA. Al resultado lo llamaremos iAESA2. Básicamente, la idea es modificar el criterio de



(a) AESA. (b) iAESA.

Figura 6.1: En el lado izquierdo, el ejemplo mostrado en [Vid86] para explicar AESA. En el lado derecho, el proceso de iAESA para el mismo conjunto de elementos. El orden de selección de cada pivote se muestra en las leyendas, paso-1, paso-2, y paso-3. En este ejemplo iAESA usa un pivote menos que AESA.

aproximación de iAESA (la similaridad entre permutaciones) usando el criterio de aproximación de AESA, $D(u)$, cuando hay empates en $F(u)$. En general este criterio ayudará mucho cuando se tienen pocos permutantes, pues existe un mayor porcentaje de permutaciones con empates.

En otras palabras, iAESA2 usa dos criterios de aproximación: el primero está dado por el menor valor según la métrica Spearman Footrule (es decir, la permutación más similar) y el segundo está dado por el menor valor de $D(u)$, en caso de empates. La complejidad de esta técnica es también $O(|\mathbb{P}'|^2 \cdot n)$. Nótese que para seleccionar al segundo pivote siempre se usará el criterio de $D(u)$, pues todos los elementos tienen la misma permutación.

6.3. Versiones Probabilísticas

En esta sección detallaremos las modificaciones a los algoritmos anteriores (AESA, iAESA e iAESA2) para convertirlos en versiones probabilísticas. Estas modificaciones consisten en:

1. **Limitar el trabajo de los algoritmos**, esto es revisar sólo una fracción de la base de datos y reportar los objetos relevantes para la consulta encontrados en esa fracción.

2. **Relajar las condiciones de búsqueda**. Usar los criterios de *relajación* descritos en [JV00, TYM06] principalmente y reportar los objetos recuperados.

6.3.1. Limitando el Trabajo

Esta modificación en los algoritmos 7 y 8 para convertir estos algoritmos exactos en probabilísticos consiste en cambiar la condición del **while** (línea 5 para ambos algoritmos) por **while** $|\mathbb{P}| < m$ donde $m \leq n$ es el número de elementos de la base de datos a compararse directamente contra la consulta. La ventaja de esta técnica es que podemos limitar el trabajo que cada algoritmo realizará.

6.3.2. Relajando las Condiciones de Búsqueda

La segunda parte consiste en relajar las condiciones de búsqueda como lo describen en [TYM06], donde se propone una manera de ganar velocidad de respuesta evitando comparaciones de distancia a cambio de disminuir la precisión de la respuesta de forma controlada por un parámetro de *holgura* α. Aunque los autores lo usaron para TLAESA su propuesta es aplicable a cualquier método exacto. En particular en estos experimentos fue aplicado a AESA (logrando una versión probabilística) para la búsqueda de los K vecinos más cercanos. Esta modificación a AESA también fue usada con iAESA en la parte experimental.

Básicamente la idea es *relajar* el radio de los elementos candidatos a ser los K-vecinos más cercanos, esto es $\alpha \cdot d_{min}$, con $0 < \alpha \leq 1$ y donde d_{min} es la distancia K-ésimo elemento candidato. Es decir, para que un elemento u reemplace a un candidato su distancia deberá ser $d(u,q) < \alpha \cdot d_{min}$.

En [JVA98] se presenta otra manera de relajar las condiciones de búsqueda. La idea consiste en mantener una lista ordenada de los K candidatos más cercanos y su distancia a la consulta. Cuando se selecciona un nuevo pivote y se calcula su distancia, si ésta es menor que la menor distancia entre los candidatos encontrados hasta ese momento, entonces se actualiza la lista sustituyendo al K-ésimo candidato por el nuevo pivote y se reordena la lista desplazando al nuevo elemento hacia la izquierda hasta su nueva posición. Una desventaja de este método es encontrar rápidamente al vecino más cercano cuando $K > 1$ pues en la lista podría tenerse almacenado cualquier elemento y no serán remplazados por ningún otro. Esta idea no será mostrada en los experimentos debido a la pobre precisión del método que se observaron en los experimentos realizados.

6.4. Resultados Experimentales

Se realizaron experimentos con bases de datos métricas: vectores aleatorios en el cubo unitario distribuidos uniformemente (véase la sección 2.6.1) y documentos TREC-3 (base de datos real, véase la sección 2.6.3).

6.4.1. iAESA Exacto

Cubo Unitario

El conjunto de datos usado fue descrito en la sección 2.6.1. Las dimensiones en estos experimentos fueron desde 6 hasta 14, el tamaño de la base de datos fue de 5.000 hasta 20.000.

El desempeño de nuestra técnica respecto al número de comparaciones de distancia en distintas dimensiones se aprecia en la figura 6.2. Nótese que a medida que la dimensión de los datos crece, se requieren más comparaciones para responder las consultas. En la figura se puede apreciar que iAESA tiene un mejor comportamiento que AESA cuando la dimensión crece. Con nuestra técnica, en este espacio, es posible reducir el trabajo hecho por AESA hasta un 17 % (dimensión 12, n=20.000). iAESA2 tuvo el mismo comportamiento que iAESA, por lo que fue omitido en esta figura.

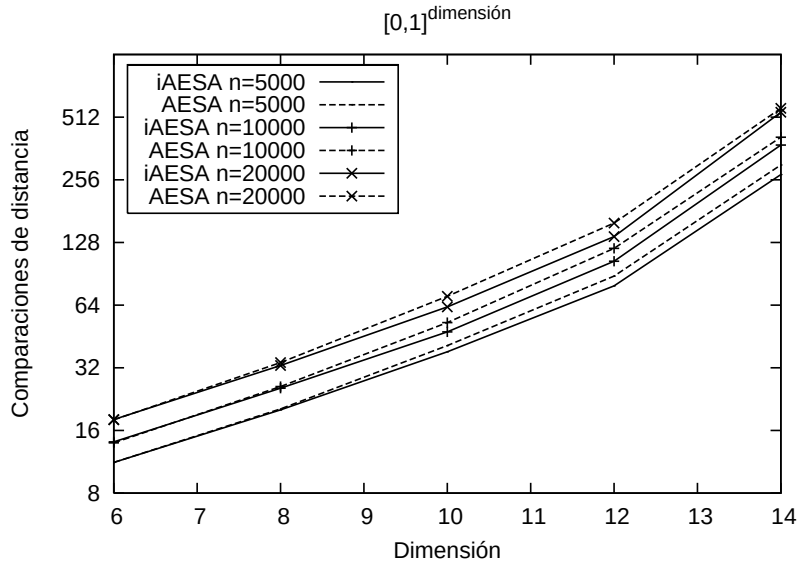


Figura 6.2: Desempeño de iAESA contra AESA para diferentes dimensiones. Los experimentos recuperaron los dos vecinos más cercanos (escala logarítmica en el eje *y*).

Documentos

En el caso del conjunto de documentos de prueba se usó el conjunto de tamaño 1265 (sección 2.6.3). La comparación que puede verse en la figura 6.3, se realizó en base a la cantidad de comparaciones de distancia cuando varía el número de vecinos más cercanos recuperados. Los algoritmos fueron AESA, iAESA e iAESA2. En general, iAESA2 es el algoritmo con el mejor desempeño en este tipo de base de datos. En el caso de usar sólo el criterio de permutantes (iAESA)

se mejora el algoritmo original AESA cuando el número de vecinos más cercanos crece sobre 7. En este espacio se mejora AESA con iAESA2 hasta un 35 % (cuando se recuperan 11 vecinos más cercanos), y con iAESA hasta un 16 %.

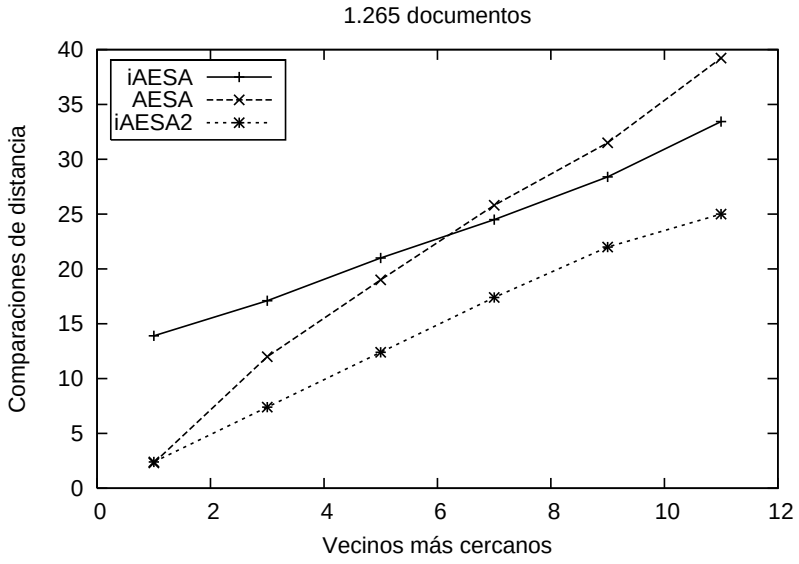


Figura 6.3: Comparando el desempeño de nuestra técnica contra AESA en la base de datos de documentos (colección TREC-3, 1.265 documentos).

Este tipo de base de datos es un ejemplo típico de espacios con distancias costosas de evaluar. Nuestra técnica requiere más tiempo de CPU por distancia calculada, pero este costo se justifica en este tipo de bases de datos, como se ve en la figura 6.4. El tiempo medido corresponde al experimento de la figura 6.3. AESA ocupa más tiempo debido al alto costo computacional de la distancia pues realiza más comparaciones que los otros algoritmos.

Por sanidad, se compararon estos resultados contra una heurística cuya selección de pivote fue aleatoria. Los resultados realizan hasta 4 veces más evaluaciones de distancia y mínimo un 50 % más. Y, como se esperaba, demuestran la importancia de un buen criterio de selección de pivote (o criterio de aproximación).

6.4.2. Algoritmos Probabilísticos

En esta sección mostramos el desempeño de AESA, iAESA e iAESA2 cuando son empleados como algoritmos probabilísticos.

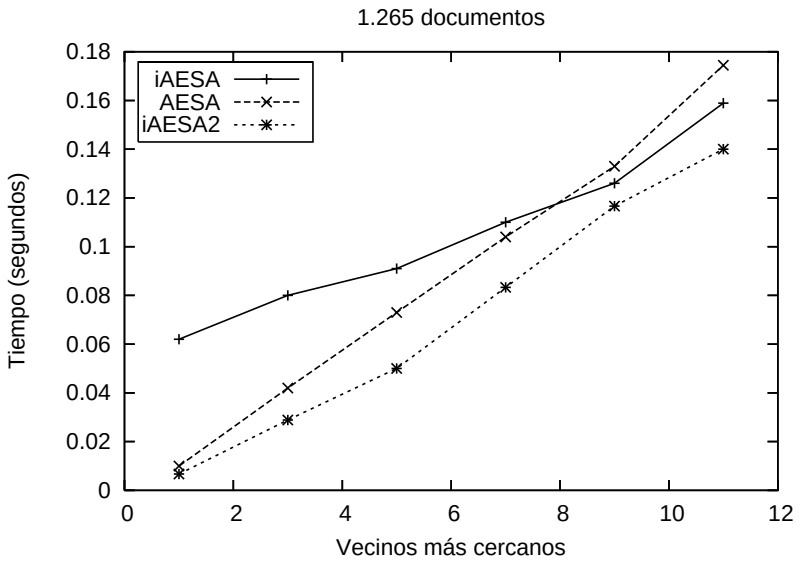


Figura 6.4: Comparación del tiempo empleado por cada algoritmo en la figura 6.3 (colección TREC-3, 1.265 documentos).

Cubo Unitario: Limitando el Trabajo

Para estos experimentos se usaron 3.000 vectores aleatorios en el cubo unitario con dimensión 128 (descritos en la sección 2.6.1). El desempeño de los algoritmos en dimensiones altas se puede ver en la figura 6.5, donde se muestra el porcentaje de consultas que recuperaron correctamente sus K vecinos más cercanos después de comparar una fracción de la base de datos. Para este caso se comparó sólo el 10 % de la base de datos. iAESA recupera sus K vecinos más cercanos mucho más rápido que AESA. Note que iAESA, comparando ese 10 %, ya encontró a todos los KNN (para $K = 1$ y $K = 2$), mientras que AESA encuentra sólo una fracción (85 % para $K = 1$ y 62 % para $K = 2$). En este experimento AESA obtiene el 95 % de las respuestas correctas (para $K = 1$) revisando el 15 % de la base de datos.

Por otro lado, también se analizó el error relativo de las distancias en las respuestas infructuosas del gráfico 6.5. El error relativo de las distancias (ERD) es una medida para saber la imprecisión de una respuesta incorrecta. Este se define como:

$$ERD = \frac{\text{distancia al } K\text{-ésimo vecino más cercano reportado}}{\text{distancia real al } K\text{-ésimo vecino más cercano}}$$

El error relativo de las distancias se muestra en la figura 6.6. Note que usando iAESA2 y revisando sólo el 6 % de la base de datos (para $K = 2$), tenemos el 88 % de las respuestas correctas, y el 12 % búsquedas infructuosas, cuyo error relativo es del 0,1 %, lo que significa que

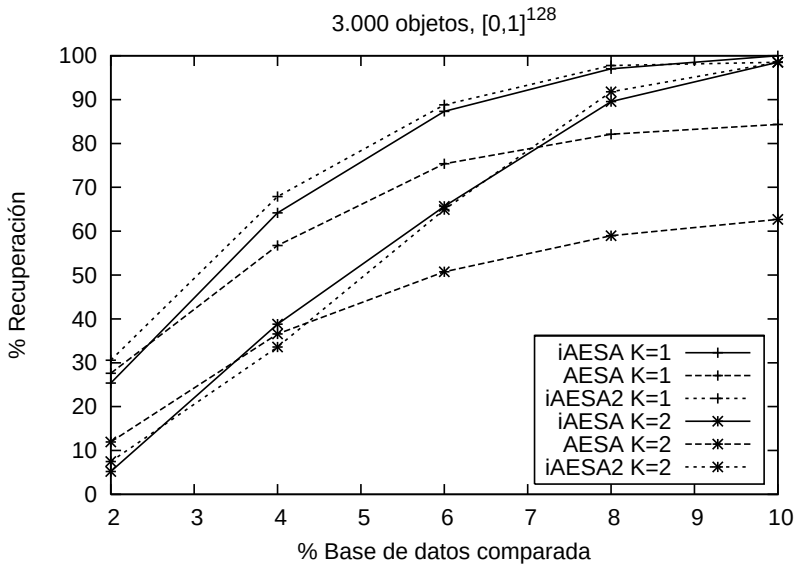


Figura 6.5: Desempeño de AESA, iAESA e iAESA2 probabilísticos. Búsqueda de los K vecinos más cercanos usando vectores aleatorios distribuidos en el cubo unitario con 128 dimensiones y 3.000 objetos.

las respuestas reportadas estaban muy cerca de las reales. Para $K = 2$ con iAESA2 y revisando sólo el 6 %, se tienen el 64 % de las respuestas correctas y el 36 % infructuosas con ERD=0,3 %. Para este caso AESA por distancia calculada recupera el 52 % de las respuestas correctas y el otro 48 % (las infructuosas) tiene el 0,5 % de ERD.

Finalmente, en la figura 6.7 se muestra el desempeño de nuestra técnica cuando varía el número de vecinos más cercanos recuperados en dimensión 128 con 3.000 objetos en el cubo unitario. Para este experimento se comparó sólo el 6 % de la base de datos. En la figura se puede ver que nuestra técnica mantiene más del 80 % de las respuestas correctas con ese porcentaje de comparaciones; aún variando el número de vecinos más cercanos buscados. AESA, a pesar de su buen desempeño, depende de la eficacia del predictor para que rápidamente pueda reducir el radio de búsqueda, es por esto que sólo revisar un porcentaje de la base de datos no favorece su desempeño sobre todo en dimensión alta.

Una comparación obligada de iAESA es contra el algoritmo basado en permutaciones mostrado en el capítulo 5. El desempeño de estos algoritmos se muestra en la figura 6.8. Para el ordenamiento usando permutaciones se usaron 128 permutantes (comparaciones internas), en el gráfico están consideradas estas comparaciones y por eso este algoritmo comienza a reportar respuestas después del 4 %. Esta comparación es interesante pues mientras iAESA e iAESA2 usan $O(n^2)$ distancias almacenadas, el algoritmo basado en permutaciones, en este caso, usa solo $O(kn)$ con $k = 128$ permutantes almacenados (1 permutación por elemento) y obtiene el 85 % de las

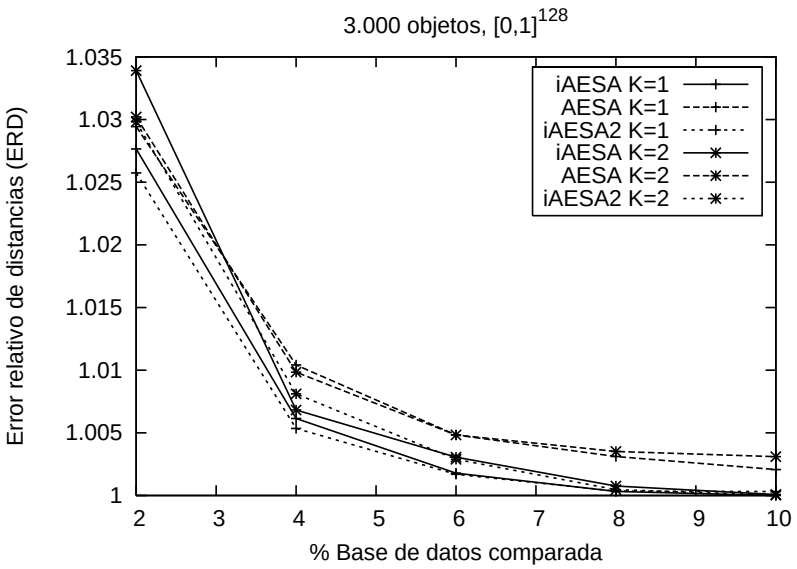


Figura 6.6: Error relativo de las distancias en las consultas infructuosas.

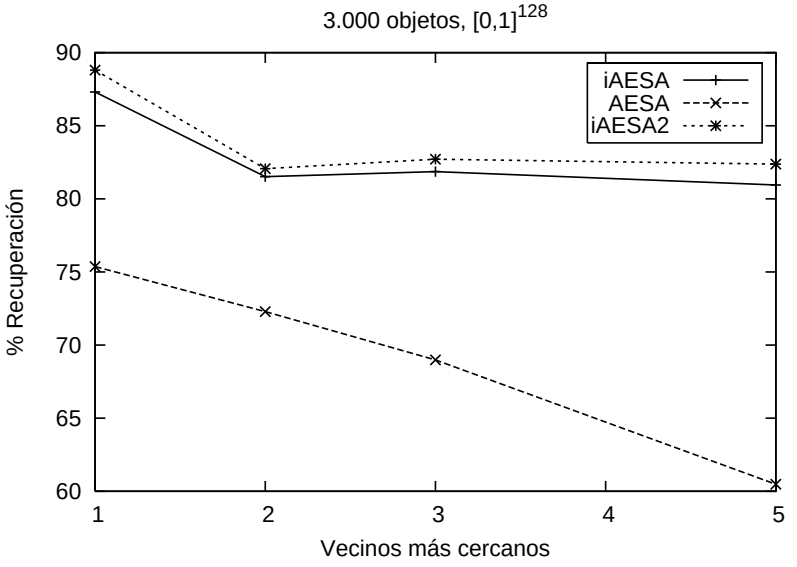


Figura 6.7: Comparación de AESA, iAESA e iAESA2 al variar la cantidad de vecinos más cercanos recuperados, y comparando sólo el 6 % de la base de datos. Dimensión 128 con 3000 objetos.

respuestas al haber comparado el 10 % de la base de datos.

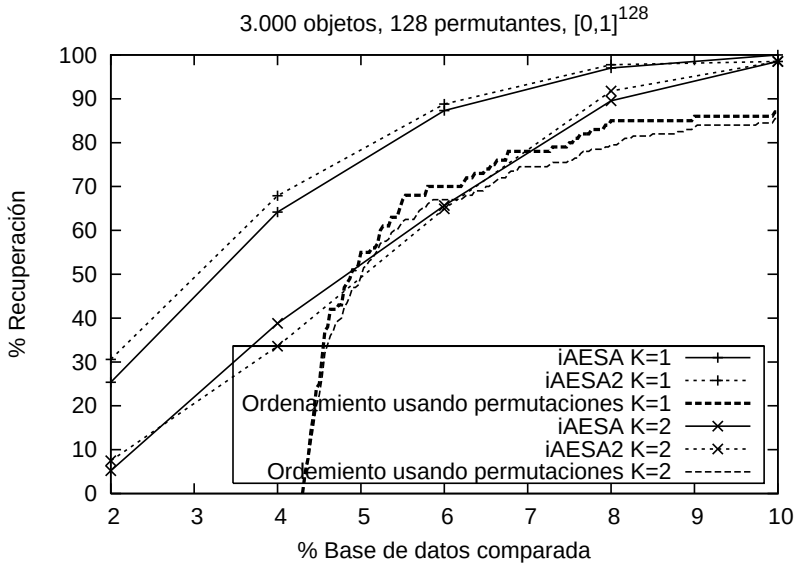


Figura 6.8: Comparación de iAESA, iAESA2 y el ordenamiento basado en permutaciones comparando sólo el 10 % de la base de datos. Dimensión 128 con 3000 objetos.

Cubo Unitario: Relajando las Condiciones de Búsqueda

Para esta comparación se usaron 10.000 vectores distribuidos uniformemente en el cubo unitario (ver sección 2.6.1) en dimensión 8 y 16.

El porcentaje de error en las respuestas entregadas por el algoritmo (de manera semejante se encuentra en [TYM06]) se define de la siguiente manera:

$$E[\%] = \frac{|R_{Alg} - R_{Opt}|}{|R_{Opt}|} \tag{6.2}$$

donde, usando la notación de la ecuación 5.3 (página 71), R_{Alg} son las respuestas entregadas por el algoritmo y R_{Opt} son el conjunto de las K respuestas correctas.

Note que el porcentaje de error es el complemento del porcentaje de recuperación para consultas de tipo K vecinos más cercanos. % Error = 100 % - % Recuperación. Una recuperación exitósa 100 %, tiene 0 % error, una recuperación infructuosa 0 % tiene 100 % de error.

En las figuras 6.9 y 6.10 se muestra el desempeño de AESA como algoritmo probabilístico. Para esta comparación se usaron dos versiones de AESA [Vid86, Vid94]: con L_1 y con L_∞; estas modificaciones consisten en usar L_1 ó L_∞ como criterio de selección de pivotes (o criterio de

promisoriedad). Para iAESA las modificaciones consistieron en usar Footrule o Spearman Rho como medida de similaridad entre permutaciones y relajar las condiciones de búsqueda de la misma forma que para AESA. En estas figuras se aprecia que todos los algoritmos tienen un desempeño semejante, excepto AESA usando L_∞ pues se nota ligeramente peor que el resto.

En las figuras se muestra el problema típico de los espacios métricos, la mayoría de los índices tienen un buen desempeño en dimensiones bajas, a medida que aumenta la dimensión se degradan. Por ejemplo en la figura 6.10 todas las versiones de AESA ya deben revisar cerca del 14 % de la base de datos para tener un 90 % de las respuestas correctas, mientras que en dimensión 8 no revisan más que el 0.7 % de la base de datos.

Una comparación interesante de estos algoritmos es contra el algoritmo basado en permutaciones. Esta comparación se puede ver en la figura 6.11. Debido a que en los permutantes no tenemos un factor α sólo mostramos su desempeño ante el mismo conjunto de datos analizando el porcentaje de error contra las comparaciones de distancia. Por ejemplo, en dimensión 8, revisando el 0.75 % de la base de datos los permutantes tienen un error del 14 % en su respuesta, mientras que las versiones de AESA tiene un 6 % de error. Sin embargo, en dimensión 16 revisando un 4 % de la base de datos los permutantes tienen 6 % de error, mientras que las versión de AESA revisando un 4 % de la base de datos ($\alpha = 0{,}4$) tienen un 90 % de error en su respuesta.

6.5. Cálculo Dinámico del Footrule

La técnica iAESA adolece del problema de tener mayor costo de CPU que AESA. Esto se debe al costo de inserción de cada nuevo pivote en todas las permutaciones de los elementos sin filtrar y de recalcular el valor del Footrule en cada inserción. En esta sección presentamos una forma de disminuir el costo de CPU de iAESA, reduciendo el número de cálculos para actualizar el nuevo valor del Footrule.

Para el primer problema de cómo insertar un nuevo elemento p en la permutación Π_u para $u \in \mathbb{U}$, la mejor forma es mover los p_i desde el final de la permutación (del pivote más lejano al más cercano) de manera que el hueco donde se insertará p sea desplazado hasta la posición correcta en la permutación.

El segundo problema es recalcular el valor del Footrule en cada inserción (que se origina por cada pivote comparado), por lo que aprovecharemos el recorrido de la inserción para modificar el valor del Footrule.

Por ejemplo, supongamos que se tienen dos permutaciones $\Pi_q = p_1, p_2$, y $\Pi_u = p_2, p_1$, $F(\Pi_q, \Pi_u) = 2$. Si insertamos p_3 podríamos tener $\Pi_q = p_1, p_2, p_3$, y $\Pi_u = p_2, p_3, p_1$, $F(\Pi_q, \Pi_u) = 4$. Note que el pivote que cambió en Π_u sólo fue p_1, pues $d(p_1, u) > d(p_3, u)$, mientras que p_2 no fue desplazado en la permutación.

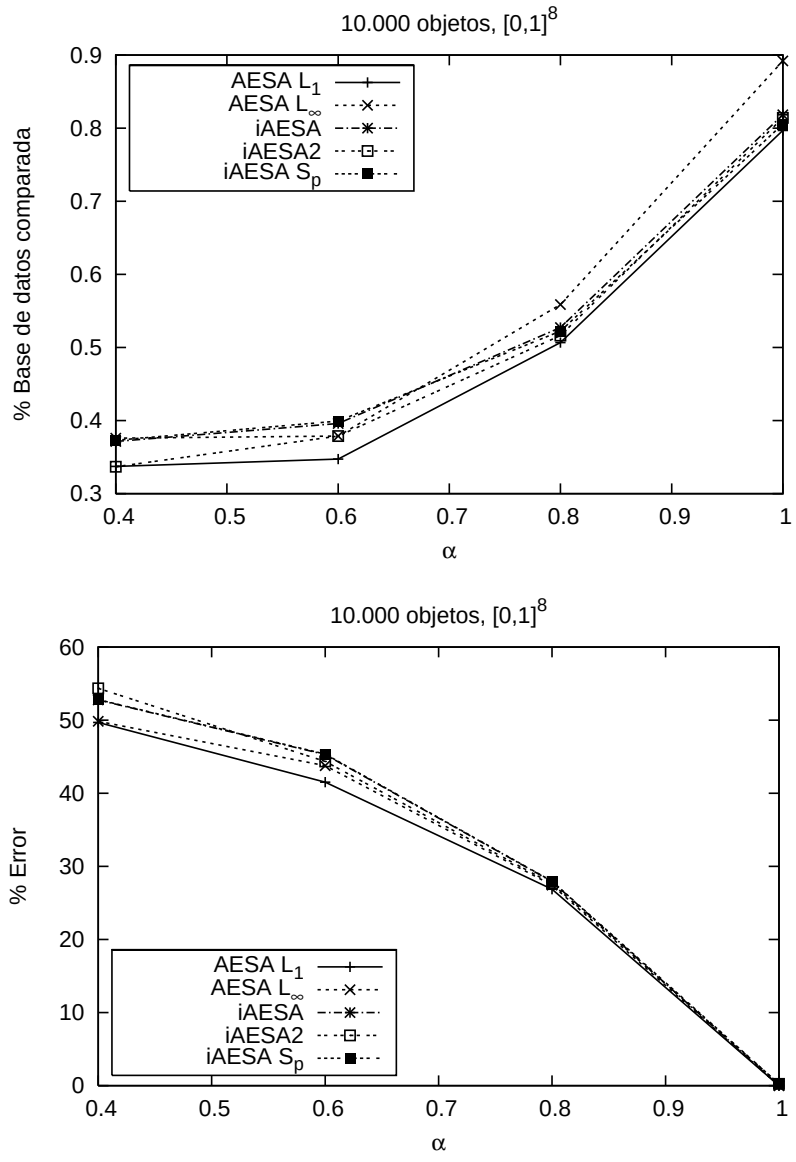


Figura 6.9: Comparación de AESA usando L_1, AESA usando L_∞, iAESA e iAESA2 aproximados. Dimensión 8 con 10.000 objetos.

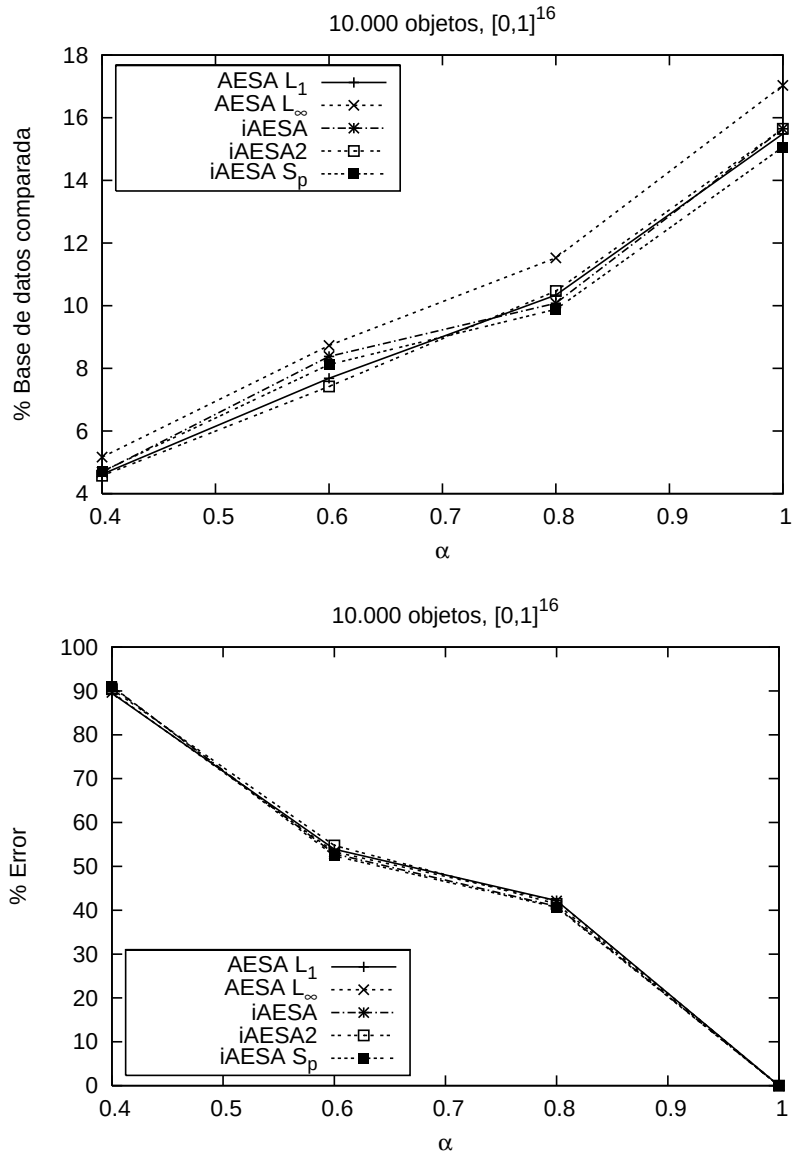


Figura 6.10: Comparación de AESA usando L_1, AESA usando L_∞, iAESA e iAESA2 aproximados. Dimensión 16 con 10.000 objetos.

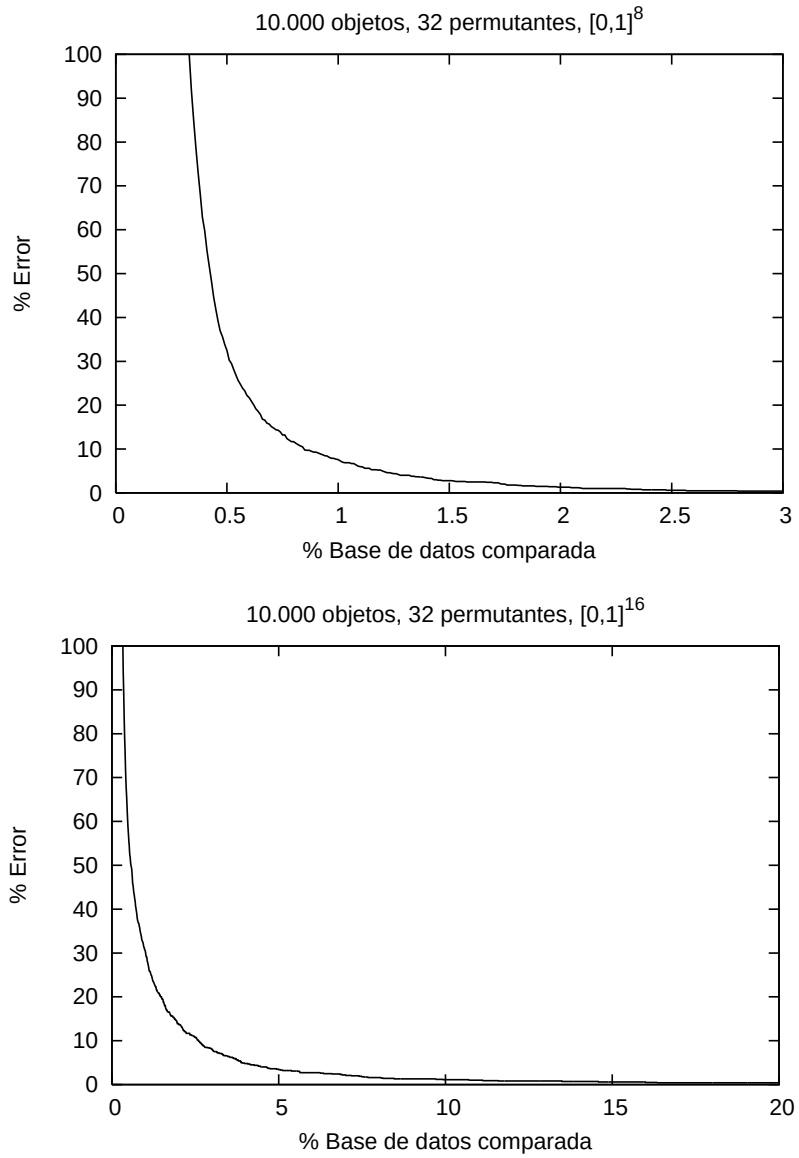
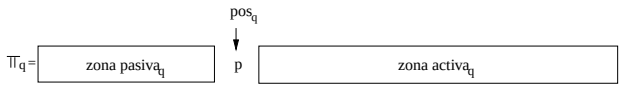


Figura 6.11: Desempeño de las permutaciones. Dimensión 8 y 16, con 10.000 objetos.

Para explicar el algoritmo para el cálculo dinámico del valor del Footrule, supondremos que la permutación de la consulta es Π_q y el nuevo pivote a insertar es p. La posición de p en la permutación Π_q después del proceso de inserción será pos_q (similarmente pos_u es la posición de p en Π_u). Esto es, $d(\Pi_q[i], q) < d(p, q)$, si y sólo si $i < pos_q$. En cada permutación diremos que existen dos zonas: *la activa* y *la pasiva*. El límite entre una y otra lo define la posición del nuevo pivote que se insertó en la permutación, como se ve en el siguiente dibujo para Π_q.



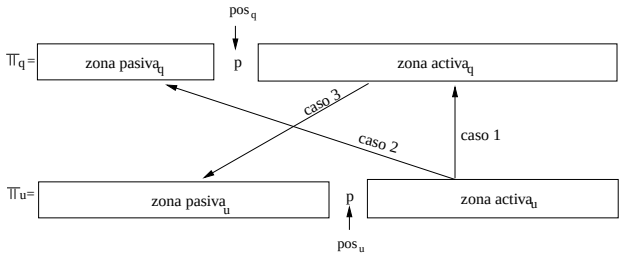
La zona pasiva de q es la de los $i < pos_q$, o lo que es lo mismo,

$$d(\Pi_q[i], q) < d(p, q)$$

La zona activa de q es la de los $i > pos_q$, o lo que es lo mismo,

$$d(\Pi_q[i], q) > d(p, q)$$

Las zonas activas y pasivas se definen de manera similar para Π_u, $\forall\, u \in \mathbb{U}$. El cálculo dinámico del valor de Footrule que se propone entre Π_q y Π_u consistirá en identificar los elementos que pertenecen a cada zona. En general, un elemento p' modificará el valor del Footrule si pertenece a zonas distintas en ambas permutaciones. Sólo existen tres combinaciones de zonas posibles para cada p' que afectarán el valor del Footrule anterior [b]. Cada una de estas combinaciones será tratada como un caso.



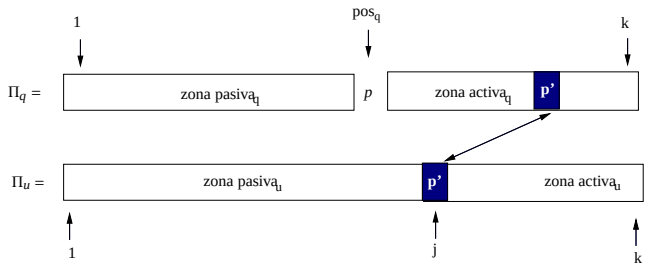
- **Caso 1**. $zona - activa_q$ con $zona - activa_u$
- **Caso 2**. $zona - pasiva_q$ con $zona - activa_u$
- **Caso 3**. $zona - activa_q$ con $zona - pasiva_u$

[b] La cuarta combinación debería ser cuando algún p' pertenece a la zona pasiva de ambas permutaciones. Esta combinación no modifica el valor del Footrule pues p' no fue desplazado en ninguna de las dos permutaciones al insertar p, por lo que conserva la diferencia de posiciones previa.

Supongamos que se seleccionó un nuevo pivote p. El proceso de iAESA indica que p se insertará en la permutación de la consulta Π_q. Para los cálculos posteriores se requerirá la permutación inversa Π_q^{-1}, la cual puede ser (re)calculada al insertar p en Π_q. Luego, para cada elemento candidato $u \in \mathbb{U} - \mathbb{P} - \mathbb{F}$ primero se verifica si éste será filtrado usando $Dmax(u)$ (línea 16 en el pseudocódigo del algoritmo de iAESA). Si u no fue descartado, entonces el nuevo pivote p debe ser insertado en Π_u (línea 19).

El cálculo del nuevo valor del Footrule se realizará durante la inserción de p en Π_u, hasta su posición final pos_u. Un elemento $\Pi_u[j]$ es desplazado una posición, hacia la derecha de la permutación, si $d(p,u) < d(\Pi_u[j],u)$. Si $\Pi_u[j]$ será desplazado, significa que pertenece a la zona activa de u y puede afectar el valor anterior del Footrule (desde ahora denotado por $F(\Pi_q,\Pi_u)_{ant}$), de la siguiente manera.

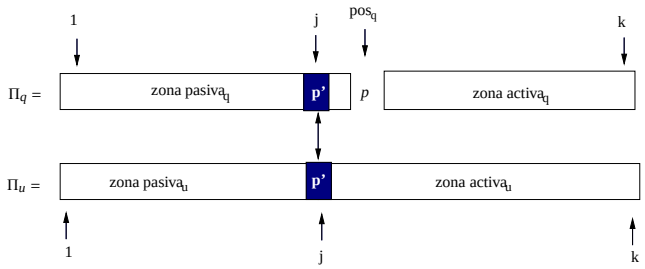
- **CASO 1. zona – activa$_q$ con zona – activa$_u$**. Sea $p' = \Pi_u[j]$. Si $\Pi_q^{-1}(p') > pos_q$, significa que en ambas permutaciones p' está en la zona activa. En este caso, este pivote no cambia $F(\Pi_q,\Pi_u)_{ant}$, pues en ambas permutaciones su desplazamiento fue de 1 y su diferencia calculada en $F(\Pi_q,\Pi_u)_{ant}$ es la misma. Sin embargo, a pesar de que no altera el Footrule anterior este caso, sí es necesario saber cuántos elementos caen en este caso, por lo que lo contabilizaremos en **elemento_sin_aporte**. En la siguiente figura el cuadro oscuro representa p', es decir, el pivote en la posición j en Π_u, es decir $\Pi_u[j]$. Note que en la figura siguiente el elemento p aún no ha sido insertado pues aún no conocemos su ubicación en la permutación, sin embargo, ya se sabe que p' estará en la zona activa pues $d(p,u) < d(p',u)$.



Por ejemplo, sea $\Pi_q = p_1, p_2, p_3$ y $\Pi_u = p_2, p_1, p_3$. Después de insertar p tenemos $\Pi_q = p_1, p_2, p, p_3$ y $\Pi_u = p_2, p_1, p, p_3$. Note que p_3 esta en ambas zonas activas y no cambió su diferencia respecto al Footrule anterior.

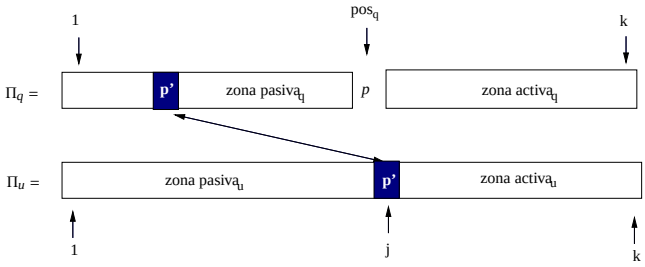
- **CASO 2. zona – pasiva$_q$ con zona – activa$_u$**. Si $\Pi_q^{-1}(p') < pos_q$ significa que en una permutación (Π_u) p se encuentra en la zona activa y en la otra (Π_q) en la pasiva. Por lo tanto puede haber dos posibilidades.

  - **CASO 2a**. Si $j \leq \Pi_q^{-1}(p')$ significa que, luego de desplazarse a la derecha en Π_u, las posiciones del pivote quedaron más cerca que antes, como se observa en el siguiente dibujo. Por lo tanto, lo contaremos como **pérdida_de_diferencia**.



Para este caso, presentaremos el siguiente ejemplo, sea $\Pi_q = p_1, p_2, p_3$ y $\Pi_u = p_2, p_1, p_3$. Después de insertar p tenemos $\Pi_q = p_1, p_2, p, p_3$ y $\Pi_u = p, p_2, p_1, p_3$. Note que p_2 tenía diferencia de 1 y después de la inserción de p su diferencia es cero.

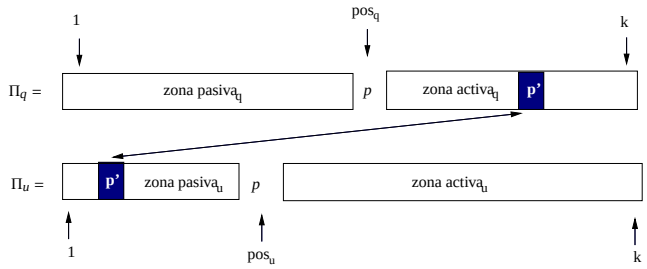
- **CASO 2b**. Si $j > \Pi_q^{-1}(p')$, entonces las posiciones de p' se han alejado luego del desplazamiento en Π_u, por lo que este elemento incrementará en 1 el valor del $F(\Pi_q, \Pi_u)_{ant}$. En este caso, lo contaremos como **incremento_de_valor**. El siguiente dibujo representa esta situación.



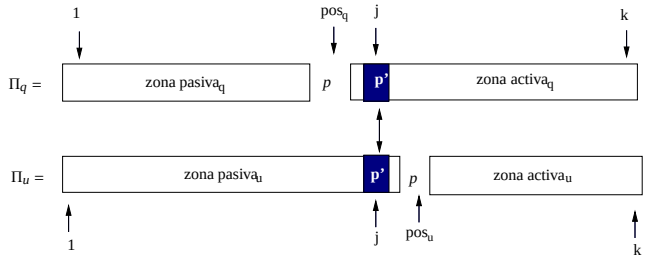
El ejemplo de este caso será $\Pi_q = p_1, p_2, p_3$ y $\Pi_u = p_2, p_1, p_3$. Después de insertar p tenemos $\Pi_q = p_1, p_2, p, p_3$ y $\Pi_u = p_2, p, p_1, p_3$. El pivote p_1 incrementará su diferencia.

Una vez que el pivote p se insertó en su posición final pos_u en Π_u, sólo nos falta contabilizar aquellos elementos que pertenecen a la zona activa de q, y a la pasiva de u. El interés por estos elementos se debe a que también podrían modificar el valor del Footrule. Estos elementos serán tratados en el caso 3.

- **CASO 3. $\mathbf{zona - activa_q}$ con $\mathbf{zona - pasiva_u}$**. Inicialmente consideraremos que todos los elementos de Π_u se ubican en este caso (lo que equivale a decir que cambiaron de zona) e incrementaremos el valor del Footrule en $k - pos_q$. Luego, se restarán los elementos que corresponden al caso 1 (que ya han sido identificados). Ahora sólo nos queda corregir las excepciones. Estas excepciones dan origen al caso 3a.



- **CASO 3a**. Por último, sólo falta considerar los elementos que se acercan, de modo de perder diferencia en vez de ganar (como en el caso 2a). Los únicos elementos que pueden estar en este caso son aquellos entre pos_u y pos_q. Por lo tanto basta revisar los elementos en Π_u que se encuentren en esta zona. Estos son los j tales que $pos_q < \Pi_q^{-1}(\Pi_u[j]) \leq j < pos_u$. En este caso, debemos considerar este caso incrementando **pérdida_de_diferencia** en 2; uno por perder diferencia entre los pivotes con respecto a $F(\Pi_q, \Pi_u)_{ant}$, y otro para descontar el incremento que tuvo por el caso 3.



Por ejemplo, sea $\Pi_q = p_1, p_2, p_3$ y $\Pi_u = p_1, p_3, p_2$. Después de insertar p tenemos $\Pi_q = p_1, p, p_2, p_3$ y $\Pi_u = p_1, p_3, p_2, p$. En este caso el pivote p_2 sufre un alineamiento y pierde su diferencia respecto al Footrule anterior.

El algoritmo 9 detalla este proceso. Una vez que tenemos identificadas las modificaciones que intervienen en el nuevo valor de $F(\Pi_q, \Pi_u)$ $\forall p' \in \mathbb{P}$ en ambas permutaciones, podemos calcular el nuevo valor del Footrule. Esto es:

$$
\begin{aligned}
F(\Pi_q, \Pi_u) = \quad & F(\Pi_q, \Pi_u)_{ant} & \\
+ \quad & |pos_q - pos_u| & \text{Diferencia del nuevo pivote } p \text{ en las permutaciones} \\
+ \quad & (k - pos_q) & \text{Caso 3} \\
- \quad & perdida\_de\_diferencia & \text{Caso 2a o 3a} \\
- \quad & elemento\_sin\_aporte & \text{Caso 1} \\
+ \quad & incremento\_de\_valor & \text{Caso 2b}
\end{aligned}
$$

Algoritmo 9 Footrule-dinámico

1: $j \leftarrow k$, $p' \leftarrow \Pi_u[j]$ // La permutación será recorrida hacia atrás
2: **while** $d(p,u) < d(p',u)$ **y** $j > 0$ **do**
3: **if** $\Pi_q^{-1}(p') > pos_q$ **then**
4: $elementos\_sin\_aporte \leftarrow elementos\_sin\_aporte + 1$ // caso 1, misma zona
5: **else if** $\Pi_q^{-1}(p') \geq j$ **then**
6: $perdida\_de\_diferencia \leftarrow perdida\_de\_diferencia + 1$ // caso 2a
7: **else**
8: $incremento\_de\_valor \leftarrow incremento\_de\_valor + 1$ // caso 2b
9: **end if**
10: $j \leftarrow j - 1$
11: $p' \leftarrow \Pi_u[j]$
12: **end while**
13: // caso 3a
14: **for** $j \leftarrow pos_u - 1$ to $j < pos_q$ **do**
15: $p' \leftarrow \Pi_u[j]$
16: **if** $j \geq \Pi_q^{-1}(p') > pos_q$ **then**
17: $perdida\_de\_diferencia \leftarrow perdida\_de\_diferencia + 2$
18: **end if**
19: **end for**
20: $F(\Pi_q, \Pi_u) = F(\Pi_q, \Pi_u)_{ant}$
21: $+|pos_q - pos_u|$
22: $+(k - pos_q)$
23: $-perdida\_de\_diferencia$
24: $-elemento\_sin\_aporte$
25: $+incremento\_de\_valor$

A continuación mostraremos un ejemplo del cálculo descrito anterior mente. Sea p_{11} el nuevo pivote a insertar en las permutaciones. Las permutaciones antes de insertar p_{11} son:

$$\begin{aligned} \Pi_q &= p_7, p_9, p_8, p_5, p_4, p_6, p_{10}, p_0, p_3, p_2, p_1 \\ \Pi_u &= p_2, p_{10}, p_6, p_0, p_9, p_4, p_7, p_3, p_8, p_5, p_1 \\ F(\Pi_q, \Pi_u) &= 44 \end{aligned}$$

después de insertar p_{11},

$$\begin{aligned} \Pi_q &= p_7, p_9, p_8, p_5, \mathbf{p_{11}}, p_4, p_6, p_{10}, p_0, p_3, p_2, p_1 \\ \Pi_u &= p_2, p_{10}, p_6, p_0, p_9, p_4, p_7, p_3, \mathbf{p_{11}}, p_8, p_5, p_1 \end{aligned}$$

A continuación analizaremos el proceso de inserción de p_{11}. Iniciamos con el elemento p_1, el cual pertenece al caso 1 (zona activa en ambas permutaciones). p_1 no cambió su diferencia y por lo tanto no modifica en nada al nuevo valor del Footrule. Los elementos p_8 y p_5 tienen un cambio de zona e incrementarán el valor de Footrule (caso 2b).

En este punto del algoritmo ya se insertó el nuevo pivote (p_{11}) en su posición en Π_u, ahora sólo falta considerar los elementos tales que $pos_q < j < pos_u$. Para este ejemplo los elementos que cumplen esta condición son p_4, p_6 y p_{10}, sin embargo, los elementos que se analizarán están en Π_u y estos son p_3, p_7 y p_4 (ciclo en la línea 13 del algoritmo), pues se conoce Π_q^{-1} y no Π_u^{-1}.

Para este ejemplo, sólo el elemento p_4 genera una excepción y será contabilizada por el caso 3a. Antes de insertar p_{11}, la diferencia de p_4 era 1, al insertar p_{11} éste se alineó en ambas permutaciones, por lo tanto, incrementaremos **perdida_de_diferencia** en 2 (1 por caso 3 y otro por pérdida de diferencia). Los valores de las variables mencionadas en el algoritmo son:

$$\begin{aligned} elementos\_sin\_aporte &\leftarrow 1(p_1) \\ perdida\_de\_diferencia &\leftarrow 2(p_4) \\ incremento\_de\_valor &\leftarrow 2(p_8, p_5) \\ F(\Pi_q, \Pi_u)_{ant} &\leftarrow 44 \\ F(\Pi_q, \Pi_u) &\leftarrow 44 + |5-9| + |12-5| - 2 - 1 + 2 = 54 \end{aligned}$$

6.5.1. Experimentación

Para medir el número de operaciones ahorradas en el cálculo del Footrule se realizaron experimentos con vectores distribuidos uniformemente en el cubo unitario (véase la sección 2.6.1), variando la dimensión de los datos entre 4 y 14. Las bases de datos fueron de tamaño 2.000 a 4.000 elementos. En la figura 6.12(a) se muestra el porcentaje de operaciones realizadas con la técnica dinámica y la secuencial. Este porcentaje fue calculado de la siguiente manera:

$$\%\text{operaciones realizadas} = \frac{\#\text{ operaciones con el Footrule dinámico}}{\#\text{ operaciones con el Footrule}} * 100$$

donde, una operación es: cada comparación en el caso de la inserción, y cada *suma* en el caso del cálculo del Footrule. Por ejemplo: Insertar el *i*-ésimo pivote a una permutación realiza $i - 1 - pos_u$ comparaciones para alcanzar su posición. El cálculo del Footrule secuencial se calcula con i.

En la figura 6.12(b) se muestra el tiempo empleado por ambas técnicas en el experimento mostrado en la figura 6.12(a): la dinámica y la secuencial.

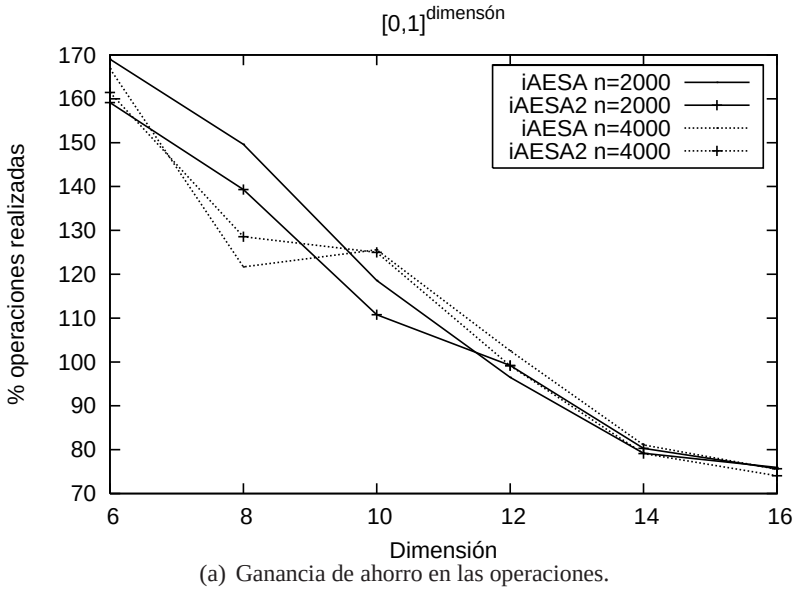
De la figura 6.12 se puede ver que cuando la dimensión de los datos aumenta, el número de operaciones que realiza el cálculo dinámico es menor que el cálculo secuencial. Esto sucede porque, en dimensión baja, las permutaciones son pequeñas y las operaciones ahorradas son compensadas con las comparaciones extra del algoritmo 9. Por lo tanto, el principal ahorro de este algoritmo ocurre cuando el tamaño de las permutaciones aumenta, como es el caso de las dimensiones altas, donde precisamente se tiene un mayor costo de CPU. La ganancia en tiempo de usar el cálculo dinámico del Footrule también se percibe en dimensiones altas, como se ve en la figura 6.12(b).

Cálculo Estimado del Footrule

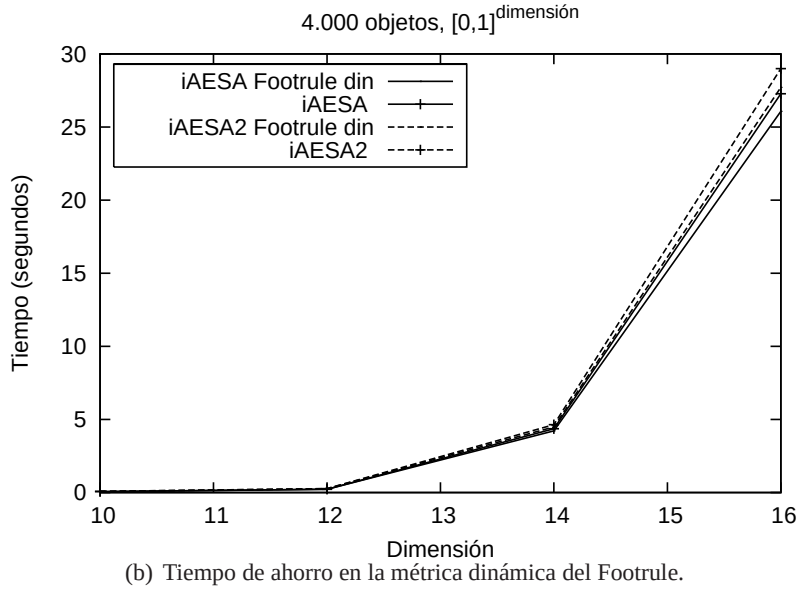
Finalmente, otra propuesta para reducir el tiempo de CPU necesario para el cálculo del Footrule durante el proceso de iAESA es no computar completamente el Footrule en cada permutación. Esto se puede lograr si sólo se estima el nuevo valor del Footrule cada vez que se agrega un pivote y después de varias estimaciones se calcula el Footrule completo en cada permutación.

El proceso para esta estimación es el siguiente: cada vez que se calcula la distancia hacia un nuevo pivote p, y al insertar p en cada permutación se suma al cálculo del nuevo Footrule sólo la diferencia de posición de p. Después de iterar un cierto número de veces entonces se recalcula el Footrule completo.

En el algoritmo 10 se muestra el cálculo del Footrule aproximado entre dos permutaciones, donde pos_u y pos_q representan la posición del nuevo pivote en las permutaciones de u y



(a) Ganancia de ahorro en las operaciones.



(b) Tiempo de ahorro en la métrica dinámica del Footrule.

Figura 6.12: Porcentaje del cociente entre el cálculo dinámico y el secuencial de la métrica de Footrule.

q respectivamente; k es el número de pivotes calculados (y por lo tanto el largo de las permutaciones); UMBRAL es el número de cálculos de Footrule estimados que serán permitidos antes de re-calcular el Footrule.

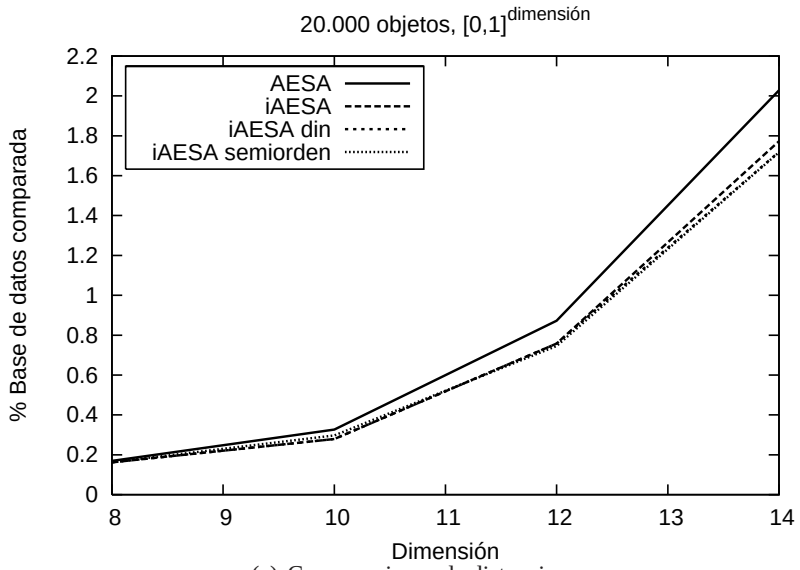
Algoritmo 10 Footrule-dinámico-estimado()

if (k%UMBRAL) = 0 **then**
  return $Footrule(\Pi_q, \Pi_u)$
else
  return $F(\Pi_q, \Pi_u)_{ant} + |pos_u - pos_q|$
end if

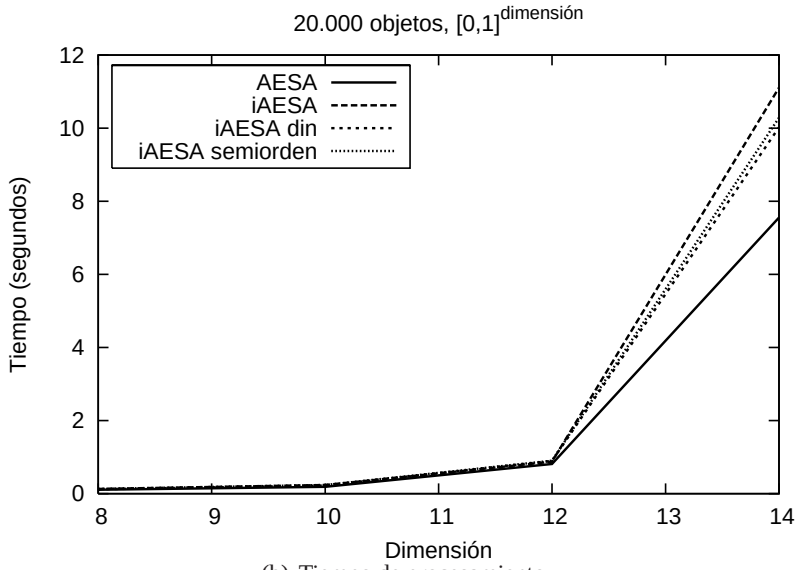
Para medir el desempeño de esta nueva propuesta para el cálculo del Footrule se realizaron experimentos sobre un conjunto de 20,000 objetos distribuidos uniformemente en el cubo unitario (ver sección 2.6.1). Los resultados se muestran en la figura 6.13. En la parte superior 6.13(a) se muestra el número de cálculos de distancia para este conjunto de datos y en la parte inferior 6.13(b) se muestra el tiempo consumido por estos experimentos. Nótese que como era de esperarse, iAESA realiza menos cálculos de distancia que AESA. Respecto al tiempo se puede observar que el cálculo del Footrule de la sección 6.5 es el más rápido para computar iAESA; a su vez este nuevo cómputo del Footrule también consume menos tiempo que iAESA y ligeramente más tiempo que el cálculo dinámico del Footrule.

En los experimentos el valor del UMBRAL fue de 5. Este parámetro es delicado de seleccionar ya que un valor muy grande producirá valores de Footrule muy distorsionados y por lo tanto menor efectividad para la selección de pivotes. A su vez, un valor muy pequeño producirá un poco ahorro en tiempo ya que se re-calcularán Footrule completos con mucha frecuencia.

Ninguno de los métodos presentados (el cálculo dinámico y el estimado) para reducir el tiempo de cálculo del Footrule muestra una mejora importante en dimensiones bajas. En dimensiones altas puede valer la pena usar alguna de las propuestas para reducir el tiempo de procesamiento de iAESA. En todo caso nuestros intentos dejan en claro que este es un problema no trivial que requiere mucha más investigación.



(a) Comparaciones de distancia.



(b) Tiempo de procesamiento.

Figura 6.13: Comparación de tiempos de procesamiento de iAESA realizando cálculos dinámicos para computar la métrica Footrule.

Capítulo 7

Aplicaciones: Identificación de Caras

El objetivo primordial de un índice métrico es su aplicación en bases de datos reales; es indispensable entonces caracterizar el desempeño de los algoritmos basados en permutaciones, tanto desde el punto de vista práctico (el tiempo total empleado en satisfacer una consulta) como desde el punto de vista teórico (cual es la cantidad de cálculos de distancia empleados al satisfacer una consulta). El número de cálculos de distancia es relevante porque nos da una idea de la escalabilidad del algoritmo, nos permite medir la complejidad que tiene un índice. En una aplicación real la complejidad adicional puede oscurecer la ventaja aparente de hacer menos cálculos de distancia. En general si la distancia es barata de calcular y la dimensión intrínseca de los datos es alta, incluso una búsqueda secuencial puede superar cualquier índice.

En este capítulo usaremos dos bases de datos con imágenes faciales de características y tamaños distintos. Para poner al lector en contexto, primero se explicará el proceso general de recuperación de caras, después se describirán los detalles de cada base de datos y finalmente se mostrará el desempeño de los algoritmos desarrollados en esta tesis para este caso de estudio.

7.1. Reconocimiento de Caras

Como todo proceso de reconocimiento de patrones en el reconocimiento de caras es necesario extraer características, que idealmente deberían ser estadísticamente independientes, para obtener sólo la información más relevante de las imágenes.

Lo anterior equivale a proyectar los objetos reales a un espacio conceptual representado mediante vectores de dimensión muy alta; una dimensión por cada característica considerada. Este proceso *separa* los datos haciéndolos más fácil de clasificar; sin embargo, hace más costosa la clasificación en sí. Note que un objeto sin proyectar contiene toda la información de éste y el error de precisión obtenido es el mínimo, sin embargo, la dimensión real puede ser muy alta. En nues-

tro caso de estudio cada cara es representada por un vector de dimensión 16.384. Es interesante hacer notar que el problema se "reduce" a clasificar vectores de dimensión "más baja", de "sólo" 761 componentes (por mencionar un ejemplo). El proceso completo de reconocimiento se detalla a continuación mediante un sistema de espacios propios .

7.1.1. Transformación de Objetos

Espacios Propios

Un sistema de reconocimiento basado en *espacios propios* [TP91a] corresponde a la tecnología más exitosa para reconocimiento de caras en imágenes digitales (una descripción más detallada se encuentra en [TP91b, GCZ+04]), y éste se muestra en el diagrama de bloques de la figura 7.1.

La idea esencial es bajar la dimensión representacional de los objetos mediante una proyección que conserve las características geométricas (relevantes para la clasificación) de los objetos en el espacio original. Esto generalmente significa hacer una transformación aproximadamente isométrica, en donde las relaciones de distancia se mantienen entre todos los objetos de la base de datos. Estos métodos tienen dos fases: una de entrenamiento, en donde se encuentra la proyección deseada y otra de reconocimiento, en donde el objeto a clasificar es proyectado de acuerdo a la proyección obtenida en la fase de entrenamiento.

En la fase de entrenamiento se obtiene una *matriz de proyección* ($\mathbb{W} \in \mathbb{R}^{N \times m}$) mediante la cual se proyectan los vectores originales a un espacio de dimensión menor. Esta matriz de proyección se puede obtener mediante cualquier método de reducción dimensional; por ejemplo, *análisis de componentes principales* [Pea01, SK87], *discriminante lineal de Fisher* [BHK97], *proyección aleatoria*, o algún otro. En este caso N es la dimensión del espacio original y m la dimensión del espacio proyectado. Todo método de reducción dimensional busca preservar las propiedades geométricas de los objetos relevantes para la construcción de un clasificador; por ejemplo proximidad relativa, distribución espacial, etc. El objetivo es que el espacio proyectado conserve las mismas propiedades que el espacio original; pero que sea computacionalmente más fácil de calcular distancias; por ello se requiere que $m << N$. En esta fase se calcula el centroide de las caras como vectores, también llamado *cara media* y denotado aquí por c en el espacio original. Esta cara media será relevante para la normalización.

Normalizar los objetos es un procedimiento común en el proceso de reconocimiento de patrones. Conceptualmente la normalización elimina las diferencias no tan esenciales de los objetos; aquellas diferencias que desde el punto de vista de una persona no son relevantes para la clasificación. En el método que estamos describiendo, el proceso de normalización consiste en sustraer la cara media de la cara de consulta (como vectores q de consulta y c cara media, N

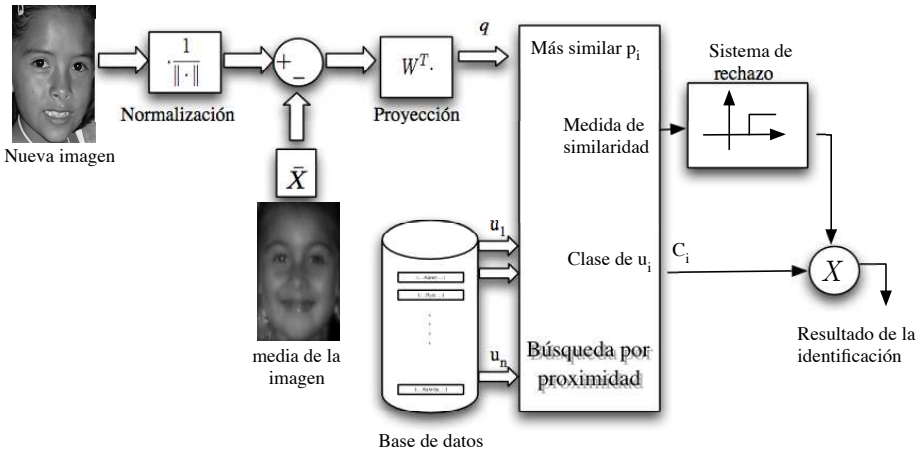


Figura 7.1: Sistema general de reconocimiento de caras basado en espacios propios.

dimensionales).

En la fase de reconocimiento se proyecta el vector normalizado $q - c$ mediante una multiplicación por la matriz de proyección W. El clasificador empleado es el método de K vecinos más cercanos, el cual se aplica buscando los K vecinos más cercanos a $q - c$ en el espacio proyectado $\mathbb{R}^m$; es decir buscamos el conjunto $NN_K(q-c)$ bajo algún criterio de similaridad; por ejemplo, la distancia Euclidiana. El resultado de la consulta es la clase del vector más semejante al de la consulta identificada por votación entre los vecinos más cercanos. En ocasiones se usa un sistema de verificación para caras no identificadas, usualmente esto ocurre cuando se quiere sintonizar el clasificador probando con distancias que pueden resultar no confiables[GMM97].

Una descripción del proceso de proyección puede verse en [Pea01], donde se observa que el error en el reconocimiento depende fuertemente del número de componentes (dimensiones) de las imágenes proyectadas, las cuales varían entre 200 y 1.050. Esta es una conclusión más o menos lógica pues hay menos distorsión geométrica en la proyección a medida que se toman más componentes.

7.1.2. Reducción Dimensional

Análisis de Componentes Principales

El análisis de componentes principales [Pea01, SK87] (PCA por sus siglas en inglés) es un método general para identificar la base (el conjunto de direcciones) en la que puede ser representado un vector dada una nube de datos. Dado que la matriz de proyección juega un papel determinante en la tasa de reconocimiento, esta matriz se calcula con una nube de datos *confiable*;

es decir, minimizando el ruido y los artefactos. Esta matriz será pues independiente de la base de datos a clasificar y se obtiene en una fase de entrenamiento donde se seleccionaron cuidadosamente las caras a utilizar como patrones. Dado que nuestro interés es realizar las consultas eficientemente consideraremos que nuestro trabajo comienza en el espacio proyectado o el espacio original; los cuales son dados. No es nuestro objetivo seleccionar adecuadamente el método de proyección o la distancia utilizada.

Proyección Aleatoria

Otro tipo de proyección eficiente cuando se usa la distancia Euclidiana se muestra en [Vem04]; dicha proyección consiste en seleccionar de manera aleatoria $O(\log n)$ componentes (donde n es el tamaño de la base de datos). Los autores muestran que la distancia usando esta selección aleatoria es muy semejante a la distancia real, su diferencia sólo es un factor pequeño.

Debido a los interesantes resultados de la proyección aleatoria, otro aporte de esta tesis (y que aún no ha sido publicado) es utilizar una nueva relación de orden (o predictor, como se ha llamado en esta tesis) para establecer en qué orden deben ser revisados los elementos de la base de datos. Este nuevo predictor consiste en utilizar la distancia Euclidiana en un espacio proyectado aleatoriamente. En este caso, la distancia entre dos objetos será mucho menos costosa que la distancia original, pues se emplean muchas menos componentes, siendo posible comparar toda la base de datos contra el elemento de consulta con esta nueva distancia y ordenar los objetos de acuerdo a ese valor. Note que la proyección aleatoria se usará sólo como predictor de orden.

7.2. Descripción de las Bases de Datos

En las bases de datos descritas a continuación se usó tanto el espacio original como dos espacios proyectados usando proyección aleatoria (con 40 componentes) [a] y PCA. La distancia Euclidiana en ambos casos fue tomada como medida de similaridad (véase la página 15).

Para los experimentos mostrados a continuación se usaron dos bases de datos: una con pocos elementos y alta dimensión intrínseca (BD-762) y otra con muchos elementos y baja dimensión intrínseca (BD-7327). En la primera el objetivo es analizar el comportamiento con pocos elementos para ver la escalabilidad en dimensión muy alta. Las dimensiones intrínsecas son diferentes debido al método de proyección y el conjunto de entrenamiento utilizado (explicado en la sección 7.1.1). Ambas bases de datos tienen originalmente dimensión real de 16.384 componentes. Es interesante hacer notar que en la literatura se especula que la complejidad del problema de búsqueda con índices métricos depende fundamentalmente de la dimensión intrínseca; por lo

[a] se usaron 40 componentes en lugar de 8 que es el resultado de $\log n$.

que el número de cálculos de distancia en el espacio original y el espacio proyectado deberá ser semejante.

7.2.1. Descripción de la Base de Datos *BD-762*

Esta base de datos es una colección de fotografías con rostros humanos FERET [PWHR98] que comúnmente es utilizada en las investigaciones de reconocimiento de caras. En estos experimentos se usó una base de datos que consiste en 762 imágenes faciales [NdS02] las cuales constan de 254 clases distintas (3 imágenes frontales por clase), y un conjunto de consultas de 254 imágenes (1 imagen por clase).

Esta base de datos fue usada tanto en su espacio original (con 16.384 componentes) como en dos espacios proyectados (PCA y proyección aleatoria). En el caso del espacio proyectado las caras fueron transformadas mediante el método basado en espacios propios (*eigenspaces*) usando PCA como matriz de proyección y la dimensión del espacio es de 761 componentes. El segundo método de proyección fue proyección aleatoria usando 40 componentes. La distancia entre las caras fue la Euclidiana en todos los espacios.

Se midió la dimensión de acuerdo a lo propuesto en [CNBYM01], tanto en el espacio original, como en los espacios proyectados: en el original es de 40,82, en el proyectado con PCA es de 39,82 y con proyecciones aleatorias de 39,60. En la figura 7.2 se muestra un histograma de distancias de esta base de datos en el espacio real y en el proyectado con PCA.

7.2.2. Descripción de la Base de Datos *BD-7327*

Esta base de datos consiste de una colección de imágenes de caras CAS-PEAL [GCZ+04], la cual tiene 7.327 objetos obtenidos de cuatro conjuntos distintos con imágenes de caras frontales (estos conjuntos pertenecen a colecciones de imágenes normales, con expresiones, imágenes con más brillo y con accesorios) correspondientes a 1.040 individuos. Esta base de datos también fue proyectada con PCA y la dimensión resultante fue de 2.152 componentes. El histograma de distancias de esta base de datos se muestra en la figura 7.3 en el espacio original (arriba) y en el espacio con proyección aleatoria (abajo). La dimensión intrínseca de los datos en el espacio original es de 9,08, en el espacio proyectado con PCA es de 8,52 y en el espacio con proyección aleatoria de 9,08. Note que ambos histogramas de distancia y las dimensiones intrínsecas son muy semejantes.

7.3. Resultados en *BD-762*

En este primer experimento es muy importante hacer notar que el número de elementos de la base de datos es muy pequeño; no es representativo de un problema real. En un caso

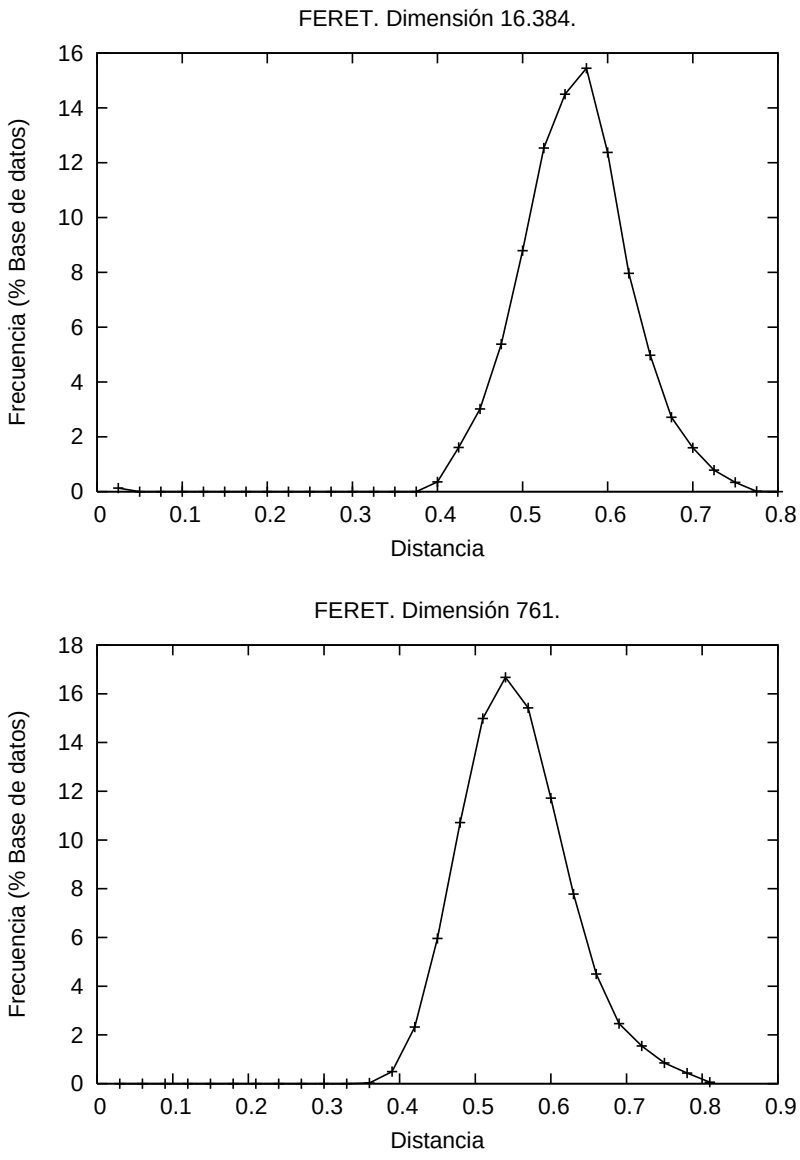


Figura 7.2: Histograma de la Base de datos BD-762 en el espacio real (arriba) y en el proyectado con PCA (abajo).

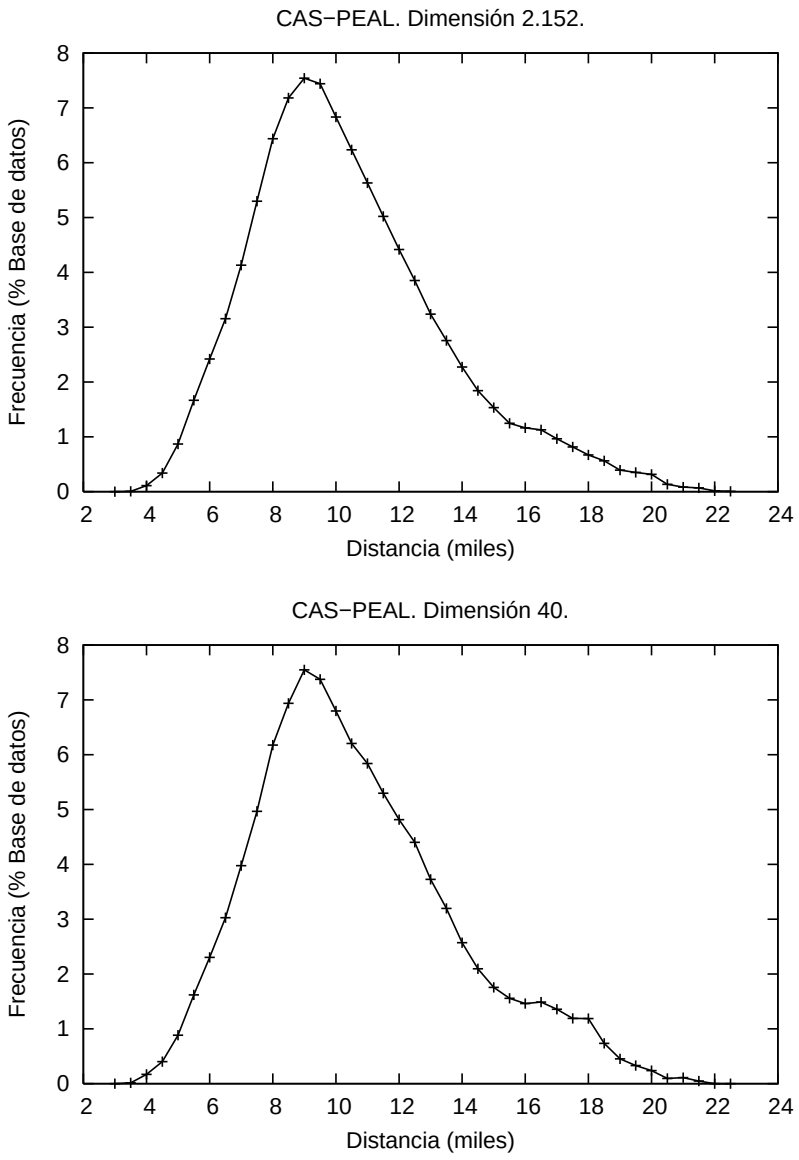


Figura 7.3: Histograma de la Base de datos BD-7327, espacio original y espacio con proyección aleatoria, arriba y abajo respectivamente.

extremo, con pocos elementos, incluso la búsqueda secuencial puede ser mucho más eficiente que cualquier índice si sólo se considera el tiempo empleado en satisfacer la consulta (por supuesto no es el caso para nuestros experimentos). Es por esta razón que nos enfocaremos en el número de cálculos de distancia empleados; más que en el tiempo utilizado. Con esto estaremos extrapolando el comportamiento de los índices métricos que sí escalan con la dimensión intrínseca y el número de elementos de la base de datos. A esta base de datos se le aplicaron todos los índices descritos y desarrollados en esta tesis.

7.3.1. Experimentos

Cota inferior en búsquedas de K-vecinos

Para resolver el problema de los K vecinos más cercanos se utilizó el algoritmo basado en permutaciones (descrito en el capítulo 5), el cual establece una relación de orden en la que la base de datos es revisada secuencialmente hasta haber encontrado los K vecinos más cercanos.

El desempeño de la relación de orden establecida por la técnica basada en permutaciones en bases de datos reales se comparó contra otras relaciones de orden basadas en las distancias L_1 y L_∞ con 64 pivotes (descritas en la sección 5.3).

Es importante hacer notar que para estos algoritmos existe un costo, el de ordenamiento (discutido en la sección 5.2.2), que implica calcular $O(n)$ distancias más baratas que la distancia original: Spearman Footrule, L_1, L_∞ ó en el espacio con proyección aleatoria.

Como base de comparación se utilizaron tres algoritmos (exactos) estándar, AESA[b], *KD-Tree* (explicados en el capítulo 2, sección 2.2.2) y búsqueda secuencial.

En los experimentos mostrados en esta sección están reportados el menor número de comparaciones para obtener la respuesta correcta de cada algoritmo.

El desempeño comparativo de todos los algoritmos en la base de datos de 762 caras se puede ver en la figura 7.4, con y sin reducción dimensional (abajo y arriba respectivamente). En este caso se muestra el número de comparaciones de distancia necesarias para encontrar los K vecinos más cercanos (variando K entre 1 y 20). En el espacio original se muestran los siguientes algoritmos: AESA, búsqueda secuencial, *KD-Tree*, ordenamientos basados en pivotes con L_1 y L_∞ y usando las permutaciones con Spearman Footrule. Para todos los algoritmos de ordenamiento se tomaron en cuenta las comparaciones a los 64 pivotes/permutantes (comparaciones internas). En el caso del espacio proyectado, además de utilizar los mismos algoritmos usados en el espacio original, también se incluyó el experimento con un espacio proyectado de manera aleatoria. En este espacio proyectado se aplicó: el ordenamiento por distancia Euclidiana con 40 componentes y la

[b]El algoritmo AESA e iAESA tienen el mismo desempeño en esta dimensión (véase el capítulo 6).

idea de las permutaciones aplicada a este espacio (40 componentes) usando 64 permutantes. En la figura 7.5 se muestra el tiempo de procesamiento respecto a todos los algoritmos de la figura 7.4.

De la figura 7.4 se observa que las permutaciones como relaciones de orden son superiores al resto de los algoritmos, por ejemplo, se reduce hasta un 50 % el trabajo del *KD-Tree* en ambos espacios (sin y con proyección). Sin embargo, usando proyecciones aleatorias se logra reducir hasta 55 % el trabajo del *KD-Tree*.

En cuanto al tiempo de procesamiento en el espacio original (figura 7.5 arriba) se aprecia que los algoritmos de ordenamiento son hasta 4,8 veces más rápidos que el *KD-Tree*, no así en el caso del espacio proyectado donde sólo se consigue el doble de velocidad (como es el caso de la distancia L_∞). Esto es debido a que es una base de datos pequeña y que es mínimo el costo debido a la distancia.

En estas figuras es claro que AESA y la búsqueda secuencial tienen mayor costo para resolver consultas de este tipo, por lo que no son algoritmos competitivos en este caso. En los sucesivo se omitirá AESA y la búsqueda secuencial. Para estos algoritmos, el *KD-Tree* se muestra superior debido a que está aprovechando la información de las coordenadas (el *KD-Tree* tienen buen desempeño en espacios proyectados con PCA) y AESA sólo la distancia entre objetos.

7.4. Resultados en *BD-7327*

7.4.1. Experimentos

Nuestra segunda base de datos, descrita en la sección 7.2.2, tiene muchos más elementos y menor dimensión intrínseca respecto a la primera. Para esta base de datos sólo se realizaron experimentos con el *KD-Tree* (usando todas las componentes y con PCA), el orden basado en permutaciones y en las distancias L_1 y L_∞ (usando todas las componentes y PCA) y el orden establecido en el espacio con proyección aleatoria y nuevamente las permutaciones en este espacio de proyección aleatoria.

En estas bases de datos es mucho más evidente el ahorro en cálculos de distancia respecto al *KD-Tree*, en el espacio original. El desempeño puede verse en la figura 7.6, en la parte superior se muestra en el espacio original y en la parte inferior en el espacio proyectado.

El tiempo de procesamiento empleado por esas consultas se muestra en la figura 7.7. Las técnicas presentadas en esta tesis son notoriamente más rápidas que los algoritmos típicos para el reconocimiento de patrones.

En la figura 7.6 (abajo) se puede observar el buen rendimiento que se tiene al usar proyecciones aleatorias (y como relación de orden la distancia en ese espacio), aunque el tiempo empleado es mayor que los basados en permutaciones y las métricas L_1 y L_∞.

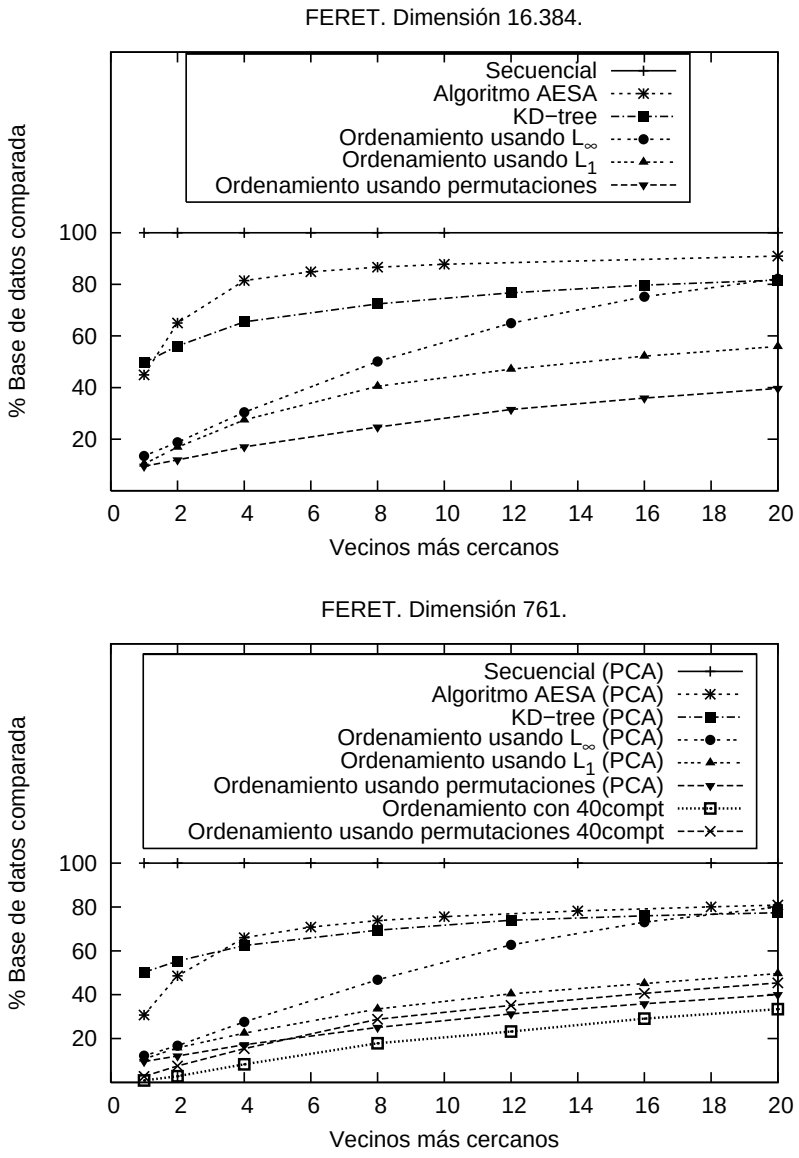


Figura 7.4: Comparación del reconocimiento de caras con AESA, búsqueda secuencial, *KD-Tree* y usando criterios de ordenamiento (64 pivotes/permutantes): basados en pivotes usando L_1 y L_∞, basado en permutaciones y con proyección aleatoria (sólo este caso usa 40 componentes).

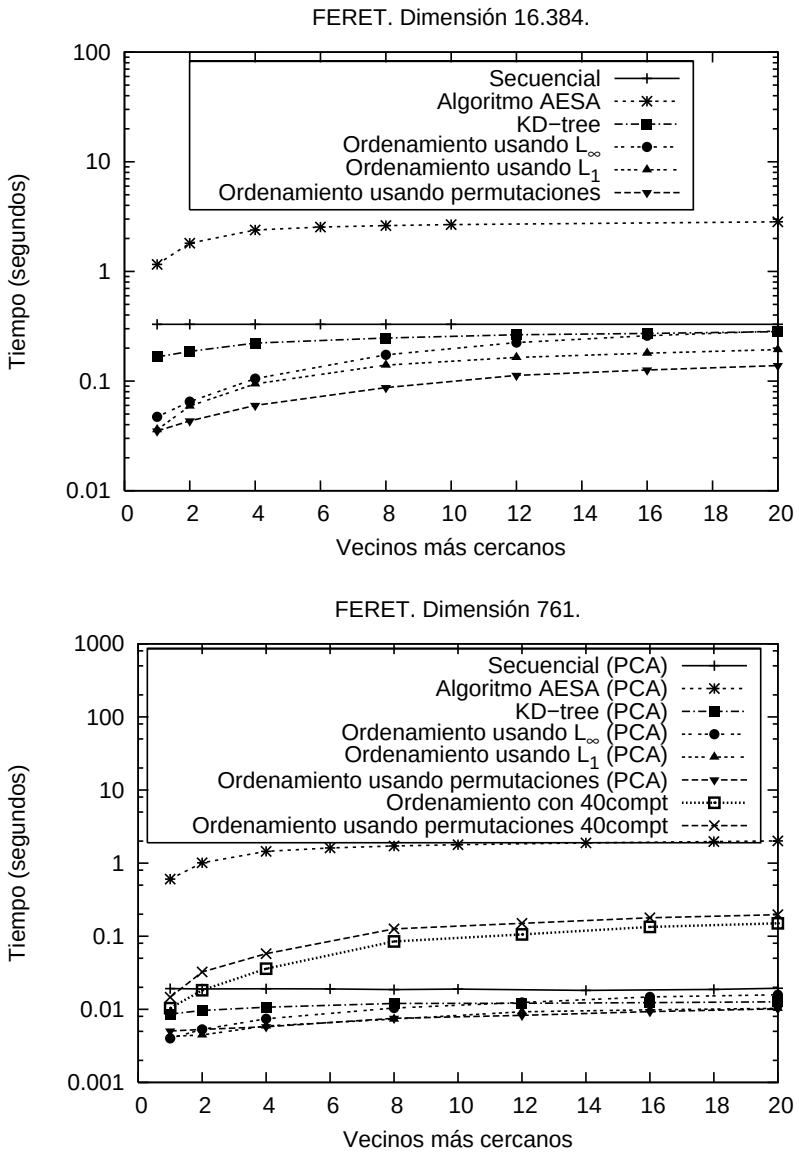


Figura 7.5: Tiempo empleado por los algoritmos AESA, búsqueda secuencial, *KD-Tree* y usando criterios de ordenamiento (64 pivotes/permutantes): basados en pivotes usando L_1 y L_∞, basado en permutaciones y con proyección aleatoria (sólo este caso usa 40 componentes).

Una conclusión muy interesante de estos experimentos es que las consultas por proximidad en bases de datos reales pueden ser agilizadas al ser tratadas como espacios métricos.

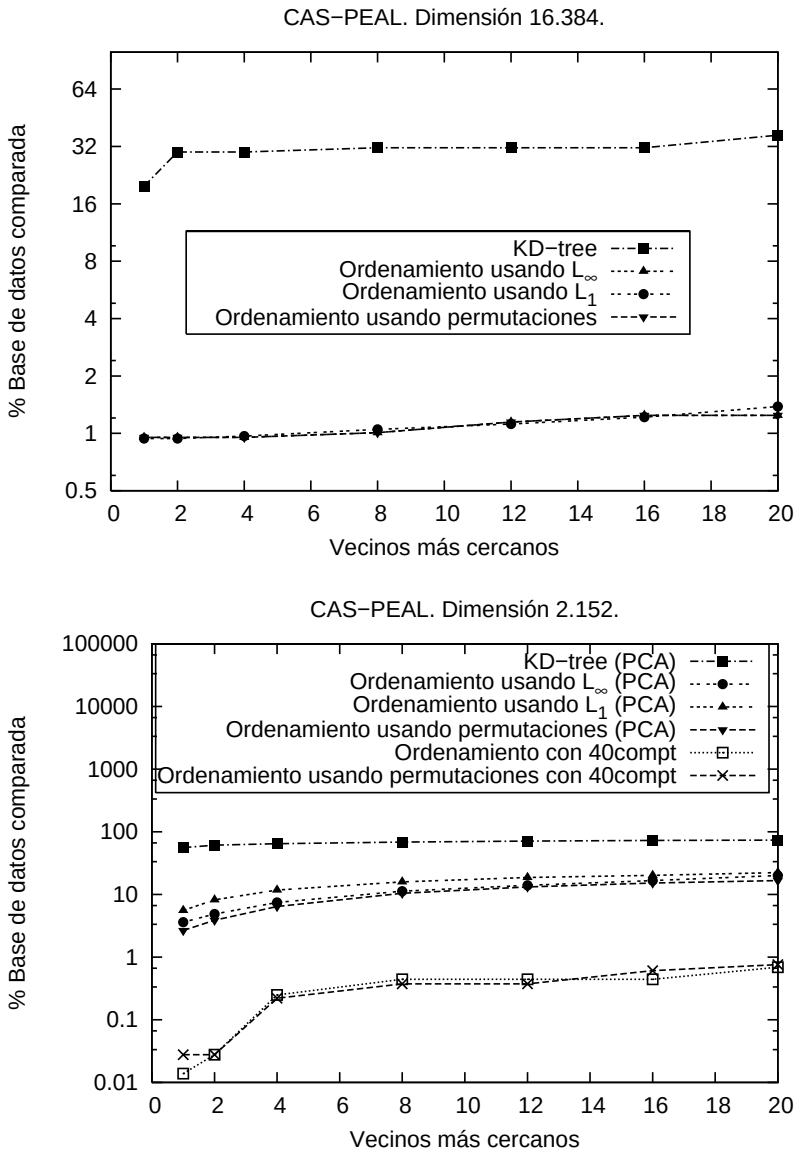


Figura 7.6: Comparación del reconocimiento de caras con *KD-Tree* y usando criterios de ordenamiento: basados en pivotes (64) usando L_1 y L_∞, basado en permutaciones (64 permutantes) y con proyección aleatoria (sólo este caso usa 40 componentes).

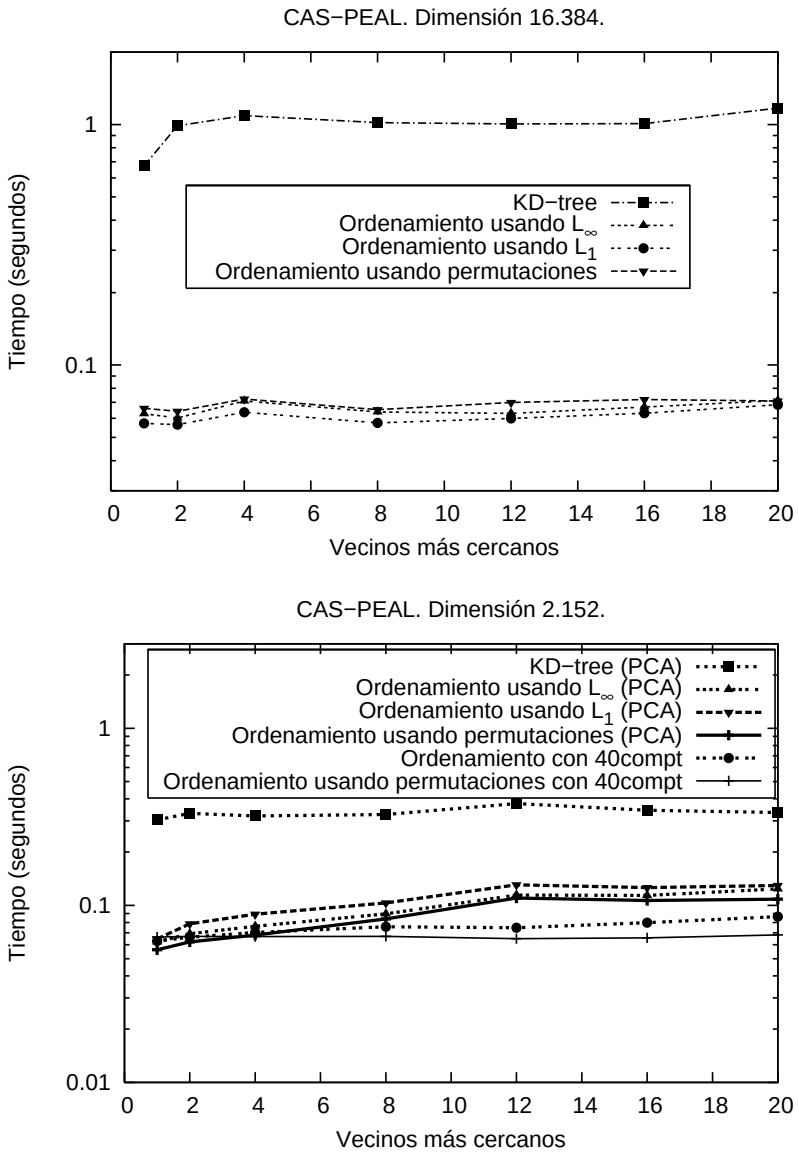


Figura 7.7: Tiempo empleado por los algoritmos *KD-Tree* y usando criterios de ordenamiento: basados en pivotes (64) usando L_1 y L_∞, basado en permutaciones (64 permutantes) y con proyección aleatoria (sólo este caso usa 40 componentes).

Capítulo 8

Bases de Datos No Métricas

Es interesante hacer notar que no hay nada en el método de ordenamiento por permutaciones que use explícitamente la desigualdad del triángulo para resolver consultas. Esto es importante porque tanto los índices basados en pivotes como los basados en particiones compactas utilizan de manera fundamental la desigualdad del triángulo para evitar hacer cálculos de distancia adicionales. Ha sido más o menos dogmático en la literatura el asumir que sólo se pueden construir índices utilizando la desigualdad del triángulo. Probaremos en este capítulo que es posible utilizar los métodos basados en permutaciones para indexar bases de datos no métricas, con excelentes resultados. Tomaremos como ejemplo una base de datos de interés práctico cuya distancia no cumple con la desigualdad triangular. Esta usa la norma fraccionaria L_p (con $0 < p < 1$).

8.1. Norma Fraccionaria L_p

En los espacios vectoriales las métricas más utilizadas para comparar vectores son las L_p, descritas en la ecuación 8.1; en particular con parámetro $p = 2$. En aplicaciones de análisis de datos [HR05] esta métrica en particular se ve muy afectada a medida que aumenta la dimensión de los datos y todos los pares se vuelven indistinguibles [BGRS99]. Para evitar este fenómeno de concentración de la medida, algunos autores sugieren el uso de normas fraccionales o L_p con $0 < p < 1$ [HR05]. Sin embargo, las normas fraccionales no cumplen la desigualdad del triángulo; es decir, para u y q dos vectores de dimensión m

$$d_p(u,q) = \left(\sum_{i=1}^{m} |u_i - q_i|^p \right)^p \tag{8.1}$$

cuando $0 < p < 1$, no cumple con la desigualdad del triángulo.

En la figura 8.1 se muestra el histograma de la desigualdad triangular de triplas en un

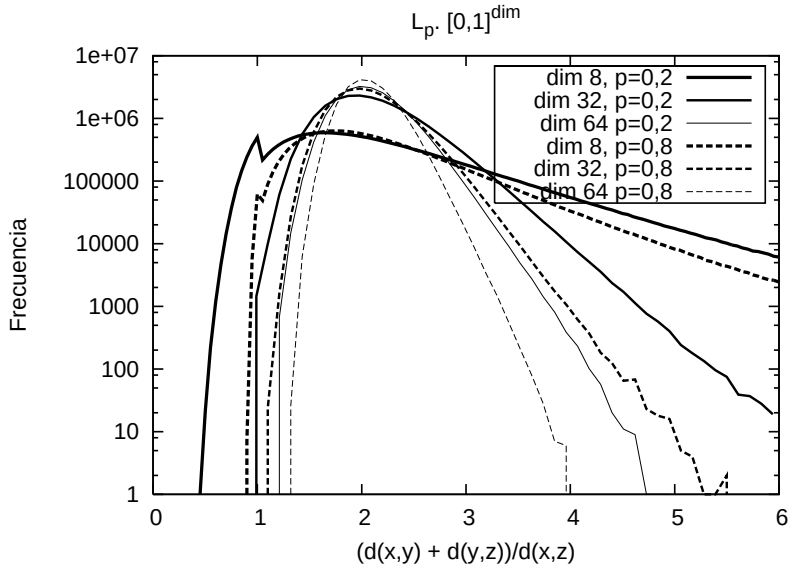


Figura 8.1: Histograma de la desigualdad triangular usando L_p para distintas triplas, donde $x, y, z \in [0,1]^{dim}$.

espacio del cubo unitario de distribución uniforme usando la norma fraccionaria L_p. En el eje x está el cociente $(d(x,y)+d(y,z))/d(x,z)$, los valores menores que uno significan que no cumplen la desigualdad triangular. Note que al disminuir la dimensión de los datos, con esta distancia, aumenta la cantidad de triplas que no cumplen con la desigualdad del triángulo. Sin embargo en dimensión alta (≥ 64) todas las triplas analizadas cumplen con la desigualdad triangular.

Desde el punto de vista algorítmico, en un índice métrico, sin la desigualdad del triángulo no sería posible descartar elementos de manera segura, es decir, no se podría tener la garantía de que los elementos descartados están fuera de la zona de interés (una bola de radio dado r centrada en la consulta), lo cual es permitido en un algoritmo probabilísco [a]. Nuestro método de búsqueda está basado en predecir la proximidad y no utiliza la desigualdad del triángulo para esto. Haremos algunos experimentos para determinar si las permutaciones son tan efectivas para predecir proximidad con normas fraccionales como lo han sido con otras métricas; en particular con métricas L_p con $p \geq 1$ en espacios vectoriales.

8.1.1. Experimentos

Los experimentos de esta sección consistieron en establecer una relación de orden de la base de datos (como en el capítulo 5) y medir la porcentaje de recuperación como se describió en la ecuación 5.3. Una vez que se tiene ordenada la base de datos se comparan los elementos en

[a] A menos que se usen heurísticas (que dependen de la base de datos) para corregir la desigualdad triangular.

ese orden contra la consulta. En estas figuras se muestran tres criterios distintos para ordenar la base de datos: usando la similaridad entre permutaciones (Spearman Rho, ecuación 5.1), usando L_1 (ecuación 2.2) y usando L_∞ (ecuación 2.4). Note que usar las dos típicas relaciones de orden usadas por los algoritmos existentes en espacios métricos L_1 y L_∞ es comparable a conocer el desempeño de esos algoritmos.

El conjunto de datos sobre el que se experimentó usando una norma fraccionaria fue el de vectores en el cubo unitario descrito en la sección 2.6.1. Para este caso se emplearon bases de datos con 3.000 objetos en distintas dimensiones (8, 32, 64 y 128), usando $k = 16$ y $k = 32$ permutantes para dimensión 8 y para el resto de las dimensiones 128 y 256 permutantes.

En las figuras 8.2(a) y 8.2(b) se muestra el desempeño de los distintos ordenamientos en dimensión 8 usando $p = 0,2$ y $p = 0,8$, respectivamente. De las figuras se puede apreciar que L_1 es ligeramente más efectivo que las permutaciones usando 16 permutantes. Usando más permutantes ($k = 32$) L_1 y las permutaciones tienen un desempeño semejante. L_∞ no muestra ser una buena alternativa en ambos casos.

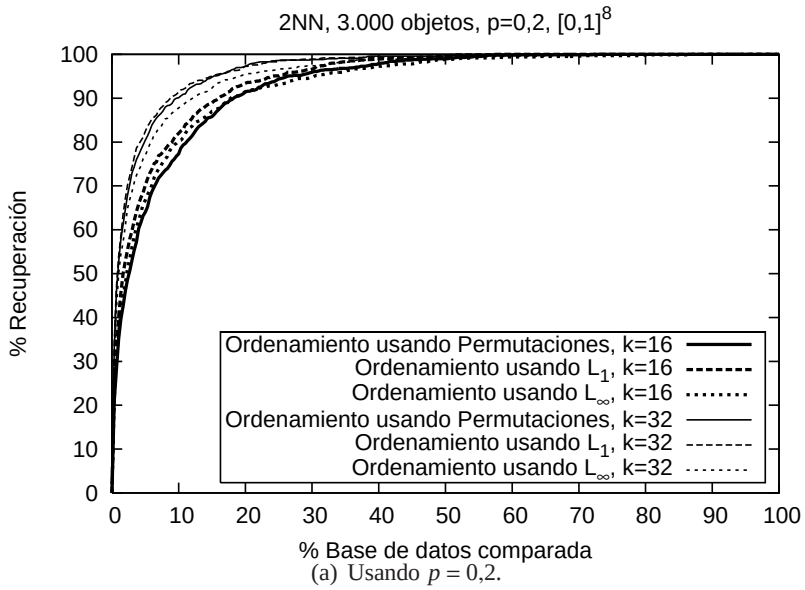
El desempeño de la técnica basada en permutaciones en dimensiones altas con 128 y 256 permutantes se puede observar en las figuras 8.3 y 8.4 (para las normas fraccionarias se utilizaron los parámetros $p = 0,2$ y $p = 0,8$).

En las figuras 8.3 y 8.4 se puede ver que la distancia L_1 ordena mejor la base de datos que L_∞, sin embargo, ordenar la base de datos usando las permutaciones es mucho más eficiente que ambas alternativas. La eficiencia de las permutaciones sobre L_1 y L_∞ se vuelve más notoria cuando la dimensión de los datos aumenta, puesto que en dimensiones bajas L_1 y las permutaciones tienen desempeños semejantes. Note que L_∞ no es una buena opción aún en dimensiones bajas para este espacio.

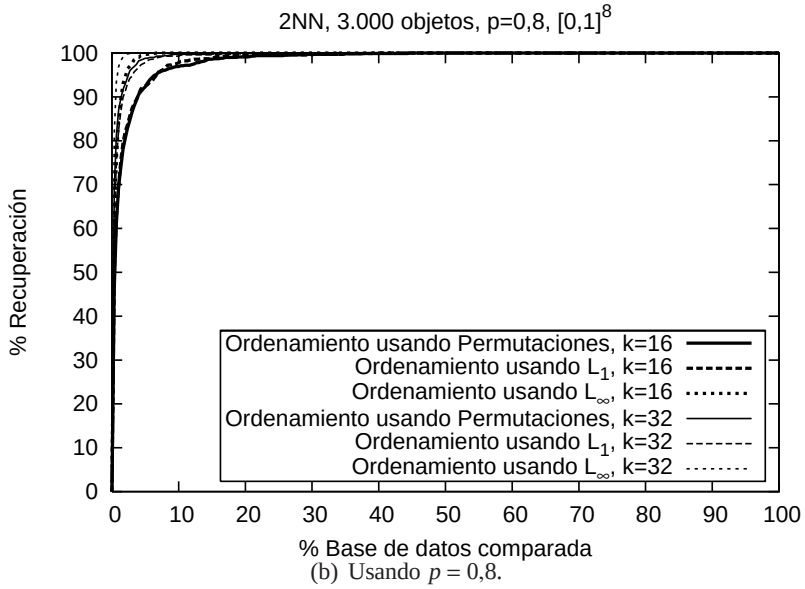
En las figuras se muestra que revisando un 10 % de la base de datos y usando 128 permutantes, es posible tener alrededor del 95 % de las respuestas en dimensión 32. Se puede apreciar que incrementar el número de permutantes incrementa de manera monótona el desempeño de la predicción basada en permutaciones.

Por otro lado, a medida que p se aproxima a 1 la norma fraccionaria tiene más semejanza a una métrica (las bolas en la norma fraccionaria tienden a ser convexas). Como se esperaba, el criterio de las permutaciones tiene un mejor desempeño a medida que p se aproxima a 1. Lo anterior es claramente reflejado en la figura 8.5, donde se muestra el desempeño de las permutaciones usando la norma fraccionaria L_p con $0 < p \leq 1$. Este experimento se realizó en dimensión 8 pues de acuerdo a la figura 8.1 en esta dimensión la base de datos realmente no cumple con la desigualdad triangular. El tamaño de la base de datos fue de 5.000 elementos.

Para verificar la calidad de la aproximación de las permutaciones en este espacio no métrico, ésta se analizó con la técnica de nubes de puntos en el apéndice A, sección A.5.



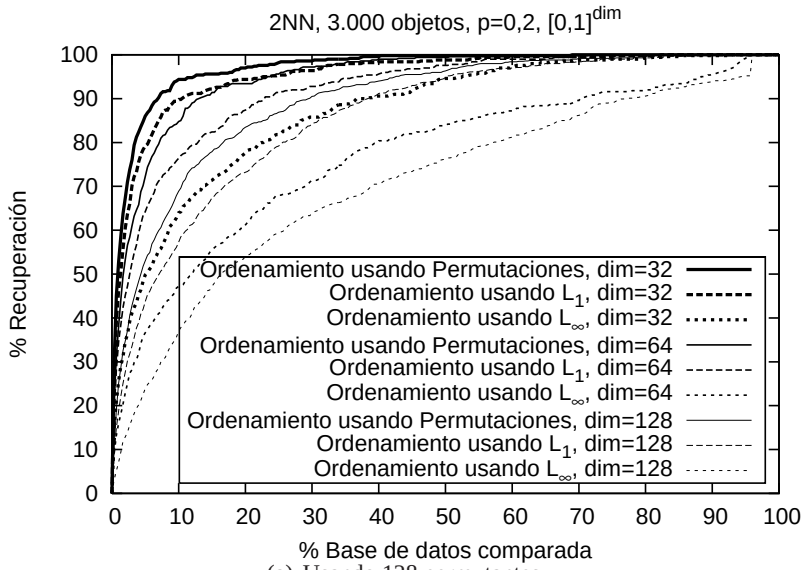
(a) Usando $p = 0{,}2$.



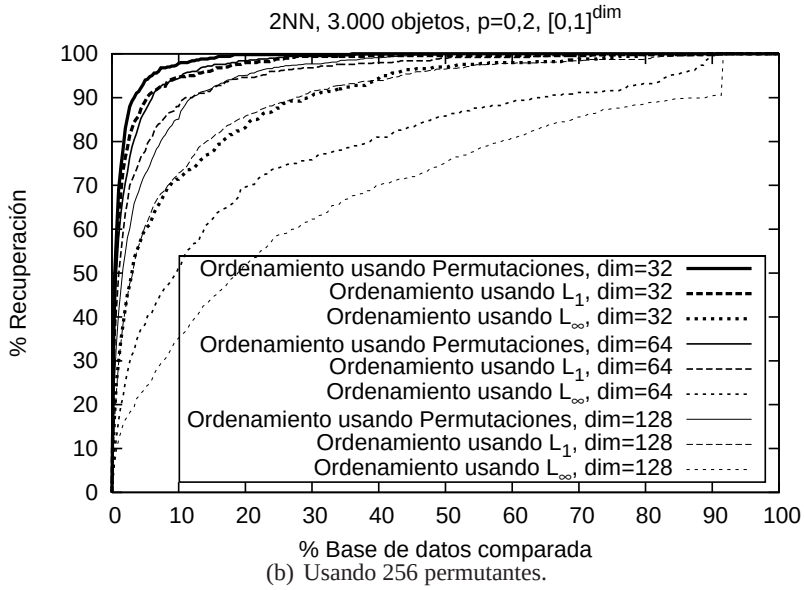
(b) Usando $p = 0{,}8$.

Figura 8.2: Comparación la técnica basada en permutaciones en vectores distribuidos uniformemente en el cubo unitario usando la distancia L_p con $p = 0,2$ y $p = 0,8$, en dimensión 8.

.

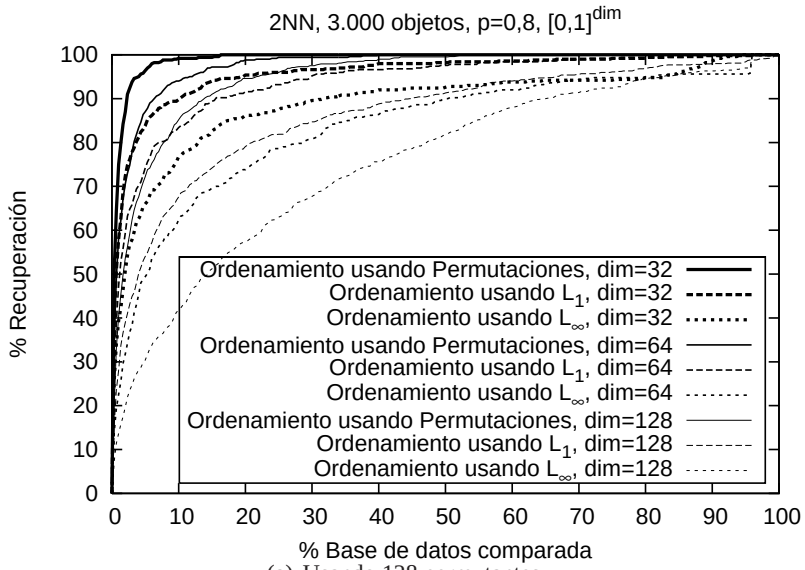


(a) Usando 128 permutantes.

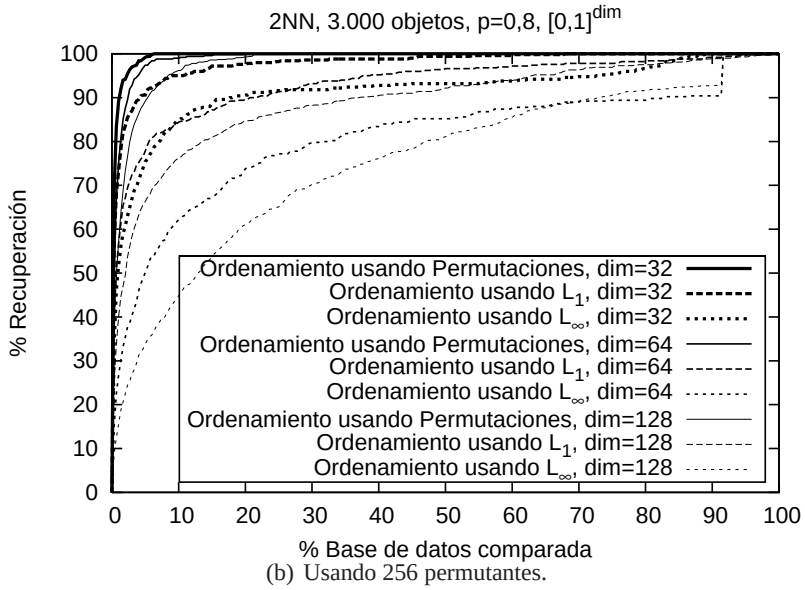


(b) Usando 256 permutantes.

Figura 8.3: Comparación la técnica basada en permutaciones en vectores distribuidos uniformemente en el cubo unitario usando la distancia L_p con $p = 0,2$.



(a) Usando 128 permutantes.



(b) Usando 256 permutantes.

Figura 8.4: Comparando la técnica basada en permutaciones en vectores distribuidos uniformemente en el cubo unitario usando la norma fraccionaria L_p con $p = 0,8$.

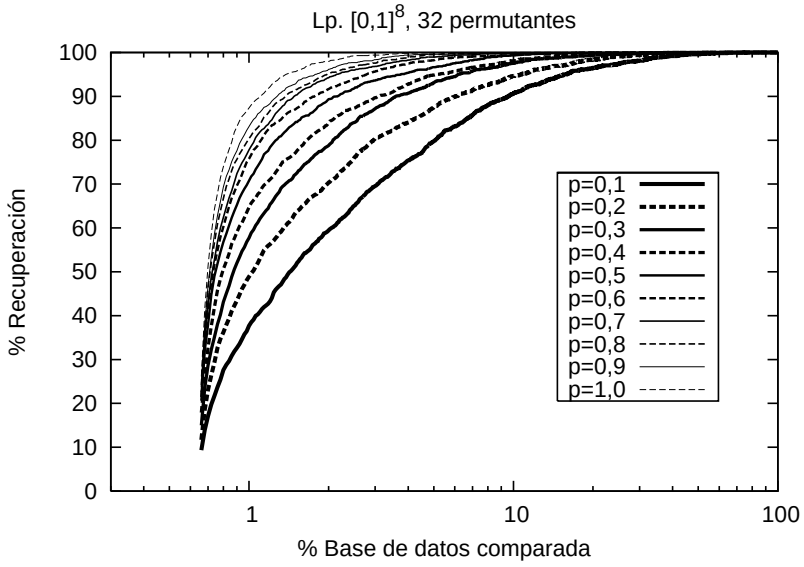


Figura 8.5: Desempeño de la técnica basada en permutaciones en vectores distribuidos uniformemente en el cubo unitario usando la norma fraccionaria L_p.

Capítulo 9

Conclusiones

En este trabajo se presentó una nueva técnica para la búsqueda por proximidad o similaridad en espacios métricos. La idea consiste en representar cada elemento por la secuencia en que se ven los elementos de un subconjunto del espacio métrico. Es decir, cada elemento induce un orden en este conjunto de elementos basado en la distancia a éstos. Esta es una manera de indexar cómo cada elemento percibe el espacio. La proximidad en el espacio métrico es predicha por la semejanza en los órdenes inducidos. Operativamente, cada elemento de la base de datos se ve como una *permutación* y la semejanza entre elementos estará dada por la distancia entre las permutaciones.

En este trabajo se mostró que las permutaciones de los elementos en la base de datos son muy buenas para *predecir* la relevancia de un objeto hacia una consulta. Un aporte de este trabajo fue un algoritmo probabilístico basado en esta técnica, el cual requiere comparar sólo una pequeña fracción de la base de datos para obtener rápidamente los elementos más relevantes ante una consulta. Nuestra técnica se mostró, por mucho, muy competitiva respecto al estado de arte en esta área. Se presentaron caminos no explorados con anterioridad, los cuales hicieron posible entender mejor el mecanismo de algoritmos de búsqueda muy antiguos, como AESA, mejorando su desempeño.

La propuesta aquí presentada es muy simple de implementar y tiene aplicaciones inmediatas a muchos problemas de reconocimiento de patrones y otras áreas donde es aceptable tener (muy buenas) aproximaciones a las soluciones exactas. Un ejemplo son las aplicaciones con alta dimensión, donde los algoritmos inexactos son la única alternativa.

Más precisamente, en esta tesis hemos presentado dos herramientas de búsqueda por proximidad en espacios métricos: un algoritmo exacto (inversiones) y un algoritmo probabilístico (basado en permutaciones). Como una consecuencia directa de usar nuestra técnica mostramos, además, una mejora importante de ahorro en tiempo empleado en espacios vectoriales.

- Inversiones. Decimos que existe una *inversión* en una permutación que representa a un elemento, cuando al menos dos elementos de la permutación están en orden diferente y la distancia entre ellos es mayor que $2r$, siendo r el radio de búsqueda. En este caso se puede garantizar que si existe una inversión entonces la distancia entre los elementos es mayor que r. Note que esto lo hace ser muy sensible a la dimensión. El algoritmo es muy interesante y sirve como motivación para el desarrollo de un algoritmo basado *sólo* en la semejanza entre permutaciones, la contribución más fuerte de este trabajo.

- Basado en permutaciones. En el caso de que en dos permutaciones no se pueda garantizar que la distancia entre elementos sea mayor que $2r$, entonces la idea que perseguimos es la suposición de que aún cuando no se pueda garantizar la distancia mínima, podría ser un buen predictor de la distancia original. Mostramos en la tesis que efectivamente la distancia entre permutaciones es un excelente predictor de la distancia original; mucho mejor que las normas L_p sobre el espacio de pivotes. El hecho anterior deriva en dos algoritmos, uno inexacto (aproximado) y otro una mejora sustancial al algoritmo que había permanecido como límite inferior de complejidad por más de 20 años, AESA, dando lugar a iAESA. Más aún, fue posible extender el ámbito de aplicación de la predicción de proximidad a bases de datos en donde no hay una métrica; sino una (dis)similaridad.

- Ahorro de tiempo en espacios vectoriales. En los espacios vectoriales se puede utilizar la información de las coordenadas para acotar el espacio de búsqueda. En estos espacios se tienen algoritmos de complejidad logarítmica para búsqueda de proximidad, como el *KD-Tree*. Introducimos en esta tesis la idea de proyecciones aleatorias, que permite predecir la distancia Euclidiana de manera muy eficiente. Usando permutaciones sobre la distancia en proyecciones aleatorias se logra reducir de manera muy importante la complejidad extra del predictor basado en permutaciones.

Los resultados obtenidos empíricamente en distintos espacios métricos que permiten medir las posibilidades del método basado en permutaciones se pueden resumir así:

- Cubo unitario. En un espacio sintético compuesto por vectores en el cubo unitario de dimensión 128, basta comparar el 15 % de la base de datos para recuperar el 99 % de las respuestas, usando 256 permutantes. Los algoritmos actuales deben revisar casi el 85 % de la base de datos para tener el mismo desempeño.

- Documentos (TREC). Esta base de datos es comúnmente usada en recuperación de la información. Usando el algoritmo propuesto, basta revisar un 2 % de la base de datos para recuperar el 90 % de la base de datos usando 64 permutantes. Los algoritmos existentes deben revisar al menos el 18 % para tener el mismo resultado.

- Imágenes faciales (FERET). En esta base de datos usando proyecciones aleatorias (en el espacio original, 16.384 componentes) se logró reducir el tiempo empleado hasta 4.8 veces que el usado por el *KD-Tree*. Esta base de datos es comúnmente utilizada en el reconocimiento de patrones, sin embargo cuenta con pocos elementos (762 caras).

- Imágenes faciales (CAS-PEAL). Ésta es otra base de datos real sobre la que se tuvo un excelente desempeño usando proyecciones aleatorias y permutaciones. En este caso, la base de datos fue mucho mayor (7.327 caras) y se logró reducir de manera muy importante el tiempo empleado y la reducción en los cálculos de distancia tanto en el espacio sin proyectar (16.384 componentes) como en el proyectado (2.152 componentes).

Por su parte, al cambiar el predictor de distancia de L_1 a distancia entre permutaciones en AESA para dar lugar a iAESA, se tuvieron los siguientes resultados en diferentes espacios métricos.

- Cubo unitario. Se disminuyó hasta 17 % las comparaciones realizadas por AESA en dimensión 14, recuperando los 2 vecinos más cercanos.

- Documentos (TREC). En el caso de documentos se logró disminuir hasta en un 35 % las comparaciones realizadas por AESA.

9.1. Trabajo Futuro

El método probabilístico aquí presentado es muy simple de implementar y tiene aplicaciones inmediatas en muchos problemas de reconocimiento de patrones, así como en otras áreas que emplean búsqueda por proximidad y aceptan aproximaciones a la solución exacta ante consultas de proximidad. A continuación mostramos varios aspectos de nuestra técnica que requieren investigación a fondo.

1. En los algoritmos probabilísticos se presentó sólo una cota de la máxima eficiencia por alcanzar; sin embargo, la suposición es que se sabe dónde terminar de hacer la búsqueda para garantizar que se tiene un cierto porcentaje de la respuesta. Un tema de relevancia práctica y que es un tópico de trabajo futuro es cómo determinar en qué momento deberíamos detener la revisión en la base de datos. Ésto está definido cuando se limita el trabajo (o tiempo) durante la revisión, de manera que sólo sean reportados los resultados obtenidos dentro de este límite. También podría ser recomendable trabajar hasta que se tenga cierta cantidad de respuestas conocidas o un error relativo muy bajo. Un criterio simple para detener la búsqueda es: trabajar hasta que la *tasa de esfuerzo* (número de cambios en el conjunto respuesta

dividido por el número de comparaciones) sea lo suficientemente pequeño. Por ejemplo, si en las últimas 100 comparaciones de distancia no se han producido cambios en el conjunto respuesta, y dado que las curvas suelen ser convexas, se podría suponer que se ha alcanzado una tasa de esfuerzo menor a 0.01, y entonces detener la revisión.

2. Otro tema de estudio es cómo seleccionar los buenos permutantes. Hasta ahora la selección ha sido aleatoria. Nuestras exploraciones preliminares para seleccionar permutantes aún no han aportado resultados significativos, pero confiamos en que, tal como se demostró en [BNC03], una buena selección mejore el desempeño del algoritmo.

3. Un reto importante en el algoritmo probabilístico presentado es reducir el tiempo de CPU, pues nuestra mejor solución toma tiempo proporcional al tamaño de la base de datos por el ordenamiento (con una constante pequeña). Una pregunta abierta es: será posible emplear sólo el tiempo proporcional a la parte que se revisará en la base de datos?.

   Una posibilidad es lograrlo por medio de índices en cascada. Hasta ahora el problema de búsqueda por proximidad en un espacio métrico fue trasladado a un problema de búsqueda por similaridad en permutaciones. Sin embargo, si estas permutaciones fueran vistas como un nuevo espacio métrico, donde cada permutación es un objeto de la base de datos, sería posible hacer búsqueda aproximada en las permutaciones, con la finalidad de encontrar rápidamente las permutaciones semejantes.

4. El concepto de permutaciones se puede emplear de manera distinta, sin tener un conjunto de referencia para representar a cada elemento como una permutación de ese conjunto. Se puede pensar en general en que la permutación se obtiene escribiendo el orden en el que se percibe a cada elemento en un conjunto de k órdenes distintos. Un ejemplo de estas permutaciones sería: Selecionar k órdenes totales distintos $\leq_1, \cdots, \leq_k$, y construir una permutación de tamaño k para cada objeto de acuerdo a la posición en la cual aparece en cada orden. Por ejemplo, considere tres órdenes y una base de datos de seis elementos. Digamos que $\leq_1 = (u_1, u_2, u_3, u_5, u_6, u_4)$, $\leq_2 = (u_3, u_2, u_6, u_4, u_1, u_5)$, y $\leq_3 = (u_4, u_1, u_5, u_2, u_3, u_6)$, entonces el orden asociado a u_1 será $(1, 3, 2)$ debido a que u_1 aparece primero en $\leq_1$ (posición 1), después en $\leq_3$ (posición 2) y finalmente en $\leq_2$ (posición 5). Después de obtener la permutación para los elementos y la consulta, se pueden aplicar directamente todos los desarrollos presentados en este trabajo.

5. El principal cuello de botella en iAESA es el reordenamiento que se hace sobre las permutaciones de los elementos que no han sido descartados. Se intentaron varias heurísticas (descritas en las secciones 6.5 y 6.5.1) sin tener grandes resultados y lo cual muestra que no es un problema simple. Otras alternativas propuestas son:

   - Basándose en el hecho de que al incrementarse un pivote la distancia sólo puede aumen-

tar, se podría ordenar sólo un porcentaje de la base de datos. Este porcentaje se obtendrá como los primeros en el orden anterior.

- Otra propuesta, para dimensiones altas especialmente, donde el tamaño de la permutación es más largo, consiste en reiniciar a la cadena vacía las permutaciones de mayor longitud (controlado por un parámetro). Con esto sería posible reducir el costo de ordenamiento en permutaciones muy largas.

6. Un problema interesante es el de obtener los k vecinos inversos de un elemento; es decir, aquellos elementos que tienen como vecino más cercano al elemento dado. Resultados preliminares permiten suponer que las permutaciones pueden ser muy eficientes para obtener los k vecinos inversos aproximados.

7. Una contribución directa, si el problema anterior se resuelve es mantener de manera dinámica el grafo aproximado de los k vecinos, incluyendo inserciones y borrados. Este problema tiene aplicaciones importantes en sistemas de recomendación, e incluso índices para búsqueda por proximidad basados basados en el grafo de los k vecinos más cercanos [PC05]. Nuestros resultados preliminares indican que obtenemos casi siempre el grafo correcto de k-NN a muy bajo costo comparado con los algoritmos de construcción exactos tales como [PCFN06].

Bibliografía

[AJE95] P. Aibar, A. Juan, and E.Vidal. Extension to the approximating and eliminating search algorithm (aesa) for finding k-nearest-neighbours. *speech Recognition and Coding: New Advances and Trends*, 147, 1995. NATO ASI Series F: Computer and Systems Sciences.

[AMN+94] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proceedings 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 573–583, 1994.

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

[BBK01b] 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.

[BBKK97] S. Berchtold, C. Böhm, D. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. *Symposium on Principles of Database Systems*, pages 78–86, 1997.

[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.

[BGRS99] K. Beyer, J. Goldstein, J. Ramakrishnan, and U. Shaft. When is "nearest neighbor' meaningful?'. *Proceedings International Conference on Database Theory*, pages 217–235, 1999.

[BHK97] P. Belhumeur, J. Hespanha, and D. J. Kriegman. Eigenfaces vs. Fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):711–720, 1997.

[BK73] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

[BKK96] S. Berchtold, D. Keim, 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.

[BN02] 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.

[BNC03] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[BO97] 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).

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proceedings 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[BY97] 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.

[BYCMW94] 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.

[BYN98] 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.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ISBN: 020139829X. Addison-Wesley, 1999.

[CF04] 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.

[Cla99] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[CMN99] E. Chávez, J. Marroquín, and G. Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, pages 57–64, 1999.

[CMN01] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.

[CN00] 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.

[CN01] E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01)*, volume 2153 of *Lecture Notes in Computer Science*, pages 147–160, 2001.

[CN05] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

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

[Cod70] E. Codd. A relational model of data for large shared data banks. In Eprint, editor, *Communications of the ACM*, volume 13, pages 377–387, 1970.

[CPZ97] 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.

[DHS73] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley, 2nd edition, 1973.

[DN87] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[FKS03] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 9(3):490–500, 1960.

[Fre07] K. Fredriksson. Engineering Efficient Metric Indexes. *Pattern Recognition Letters*, 28(2):75–84, 2007.

[GCZ+04] W. Gao, B. Cao, S. Zhou, X. Zhang, and D. Zhao. The CAS-PEAL large scale chinese face database and evaluation protocols. Technical report, Join Reserarch & Development Laboratory, CAS, 2004. JDL_TR_04_FR_001.

[GMM97] M. Golfarelli, D. Maio, and D. Maltoni. On the error-reject trade-off in biometric verification systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):786–796, 1997.

[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.

[Har95] D. Harman. Overview of the Third Text REtrieval Conference. In *Proceedings Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[HR05] P. Howarth and S. M. Rüger. Fractional distance measures for content-based image retrieval. In *Advances in Information Retrieval, 27th European Conference on IR Research, ECIR 2005.*, volume 3408 of *Lecture Notes in Computer Science*, pages 447–456. Springer, 2005.

[HS95] G. Hjaltason and H. Samet. Ranking in spatial databases. In *Fourth International Symposium on Large Spatial Databases*, pages 83–95, 1995.

[HS00] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. *Technical Report TR 4199. Department of Computer Science, University of Maryland*, 2000.

[HS03] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions Database Systems*, 28(4):517–580, 2003.

[JV00] A. Juan and E. Vidal. On the use of edit distances and an efficient k-NN search technique (k-AESA) for fast and accurate string classification. In *Proceedings of the 5th International Conference on Pattern Recognition (ICPR 2000)*, volume 2, pages 2676–2679, 2000.

[JVA98] A. Juan, E. Vidal, and P. Aibar. Fast k-nearest-neighbours searching through extended versions of the approximating and eliminating search algorithm (aesa). In *International Conference on Pattern Recognition ICPR'98*, pages 828–830, 1998.

[KM83] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *Transactions on Software Engineering*, 9(5), 1983.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics - Doklady*, volume 10, pages 707–710, 1966.

[MOC96] 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.

[MOV94] 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.

[MSMO02] 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.

[MSMO03] 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.

[Nav99] G. Navarro. Searching in metric spaces by spatial approximation. In *String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

[Nav02a] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[Nav02b] G.Ñavarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[NdS02] P. Navarrete and J. Ruiz del Solar. Analysis and comparison of eigenspace-based face recognition approaches. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 16(7):817–830, 2002.

[NPC02] 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.

[Par02] 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.

[PC05] R. Paredes and E. Chávez. Using the *k*-nearest neighbor graph for proximity searching in metric spaces. In *12th International Symposium on String Processing and Information Retrieval (SPIRE 2005)*, volume 3772 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 2005.

[PCFN06] R. Paredes, E. Chavez, K. Figueroa, and G. Navarro. Practical construction of *k*-nearest neighbor graphs in metric spaces. In *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, volume 4007 of *Lecture Notes in Computer Science*, pages 85–97. Springer, 2006.

[Pea01] K. Pearson. On lines and planes of closest fit to system of points in space. *Philophical Magazine*, 2(6):559–572, 1901.

[PN06] R. Paredes and G. Navarro. Optimal incremental sorting. In *Proceedings 8th Workshop on Algorithm Engineering and Experiments (ALENEX 2006)*, pages 171–182. SIAM Press, 2006.

[PWHR98] P. Phillips, H. Wechsler, J. Huang, and P. Rauss. The FERET database and evaluation procedure for face recognition algorithms. *Image and Vision Computing Journal*, 16(5):295–306, 1998.

[Rob81] John T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM Press, 1981.

[SK83] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[SK87] L. Sirovich and M. Kirby. A low-dimensional procedure for the characterization of human faces. *Journal of Optical Society of America*, 4:519–524, 1987.

[TP91a] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Congnitive Neuroscience*, 3(1):71–86, 1991.

[TP91b] M. Turk and A. Pentland. Face recognition using eigenfaces. In *Proceedings IEEE Conference Computer Vision and Pattern Recognition*, pages 586–591, 1991.

[TYM06] 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.

[Uhl91] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[Vem04] S. Vempala, editor. *The random Projection Method*. AMS, Dimacs/65.S, 2004.

[Vid86] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[Vid94] E. Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa). *Pattern Recognition Letters*, 15:1–7, 1994.

[Vil95] J. Vilar. Reducing the overhead of the AESA metric-space nearest neighbor searching algorithm. *Information Processing Letters*, 56:256–271, 1995.

[VL88] 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.

[VRCB88] 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.

[Wat95] M. Waterman. *Introduction to Computational Biology*. Chapman and Hall, 1995.

[Wil64] J. Williams. *Algorithms 232 Heapsort*, volume 7, chapter 3, pages 378–348. CACM, 1964.

[WJ96] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, 1996.

[Yia99a] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, 1999.

[Yia99b] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. Technical report, NEC Research Institute, 1999.

[YOTJ01] 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.

[YY85] A. Yao and F. Yaho. A general approach to D-dimensional geometric queries. In *Proceedings ACM Symposium on Theory Computing*, volume 12, pages 163–168, 1985.

[ZP65] E. Zuckerkandl and L. Pauling. Evolving genes and proteins. *Academic Press*, 8(357):97–166, 1965.

Apéndice A

Nubes de Puntos

En este apéndice se mostrará una comparación entre el criterio basado en permutaciones y otros criterios (L_1 y L_∞) que también sirven para ordenar la base de datos por promisoriedad, y que hasta ahora han sido los ocupados en los algoritmos para espacios métricos.

Para cada espacio, usado en la parte experimental a lo largo de esta tesis, se graficaron nubes de puntos donde se puede comparar el orden dado por un criterio de promisoriedad con el orden dado por la distancia real. Los criterios de promisoriedad fueron: la similitud entre permutaciones (spearman rho), L_1 y L_∞. Para cada experimento se graficaron 100 consultas; para cada consulta se graficaron los elementos de la base de datos en el orden correspondiente. Con estos experimentos se pretende mostrar la efectividad de cada método de ordenamiento usado.

En los gráficos que presentaremos a continuación el caso ideal es una línea de (0,0) a (100,100), pues ésta indicaría que el orden establecido por el criterio de promisoriedad es también el orden dado por la distancia real, y que nunca se compararía un elemento innecesario. Las técnicas están lejos de lograr el ideal, pero los gráficos que se asemejan más a una línea indican un mejor criterio para ordenar la base de datos.

En la recuperación de información lo más importante es revisar sólo un pequeño porcentaje de la base de datos y encontrar todas la respuestas. Ésto es conocido como recuperación y precisión. La recuperación es el porcentaje que se ha revisado y la precisión es la cantidad de respuestas correctas en cierto porcentaje revisado. En nuestro caso, nos interesa recorrer un pequeño porcentaje de la base de datos y tener la mejor precisión posible. Para esta parte del análisis se hizo un acercamiento en las imágenes de las nubes del 10 % (tanto en x como en y, lo que equivale a evaluar la recuperación y la precisión). En los gráficos de precisión lo ideal es que la nube de puntos esté concentrada en el eje x y para el caso de la recuperación la nube debería concentrarse en el eje y.

A.1. Cubo Unitario

Para este tipo de vectores distribuídos de manera uniforme en el cubo unitario (descritos en la sección 2.6.1) se emplearon distintas dimensiones. En este caso, sólo se mostrará para dos dimensiones distintas, 8 y 128. La distancia empleada fue la Euclidiana.

A.1.1. Dimensión 8

En la figura A.1 se muestran las nubes de puntos con los tres criterios de ordenamiento L_1, L_∞ y permutaciones (de arriba hacia abajo). En los gráficos las permutaciones tienen un desempeño notoriamente superior.

A.1.2. Dimensión 128

En el caso de vectores en el cubo unitario en dimensión 128 se usaron 254 pivotes/permutantes. Los gráficos correspondientes a esta dimensión se muestran en la figura A.2. En este espacio es muy interesante ver que el criterio de L_∞ no establece ningún orden útil. El criterio de L_1 establece un ligero orden en la base de datos. Note que de estos tres criterios el mejor desempeño lo tiene nuevamente el criterio de las permutaciones, aunque se ha degenerado con respecto a la dimensión 8.

Los acercamientos de los gráficos (en el eje x y en el eje y) mostrados en la figura A.2 corresponden a la precisión y recuperación y se muestran en la figura A.3. En estos gráficos es más evidente que el mejor criterio de ordenamiento es el basado en permutaciones.

A.2. Documentos

Las nubes de puntos correspondientes a la base de datos de 1.265 documentos se muestran en la figura A.4. Estos gráficos se ven fundamentalmente distintos de los anteriores. Esto se debe a los agrupamientos (clusters) que presenta este espacio (véase el histograma en la figura 2.7, página 24). Sin embargo, note que sólo el criterio de permutaciones es un buen ordenamiento de los clusters. En los gráficos donde se usó L_∞ note que se aprecia una línea, la cual corresponde a un cluster que se ha ordenado mal y queda repartido por toda la base de datos ordenada.

Los acercamientos de los gráficos (en el eje x y en el eje y) de la figura A.4 y que corresponden a la recuperación y precisión se muestran en la figura A.5. Note que el criterio con mejor recuperación y precisión es el basado en permutaciones.

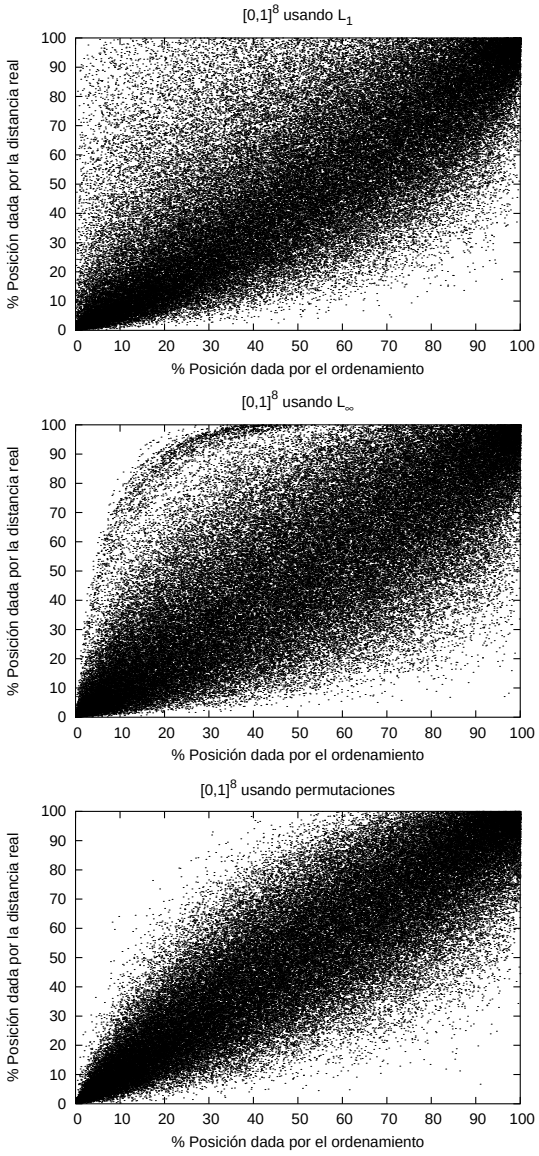


Figura A.1: Nubes de puntos para un espacio distribuido uniformemente en el cubo unitario con dimensión 8 y usando 64 pivotes/permutantes. De arriba hacia abajo, L_1, L_∞ y permutaciones.

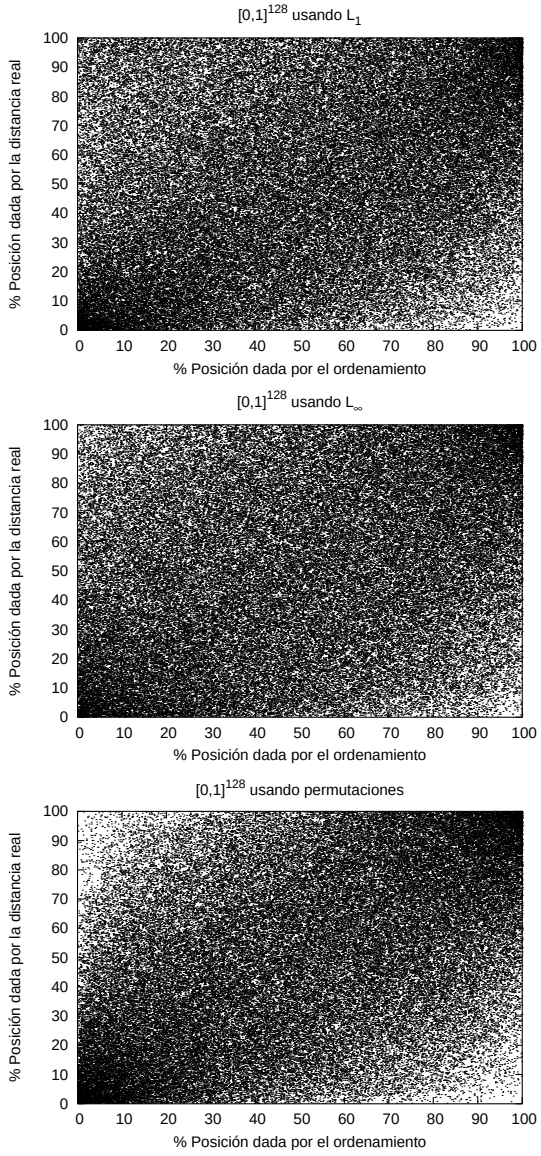


Figura A.2: Nubes de puntos para un espacio distribuido uniformemente en el cubo unitario con dimensión 128 y 254 pivotes/permutantes. De arriba hacia abajo, L_1, L_∞, y permutaciones.

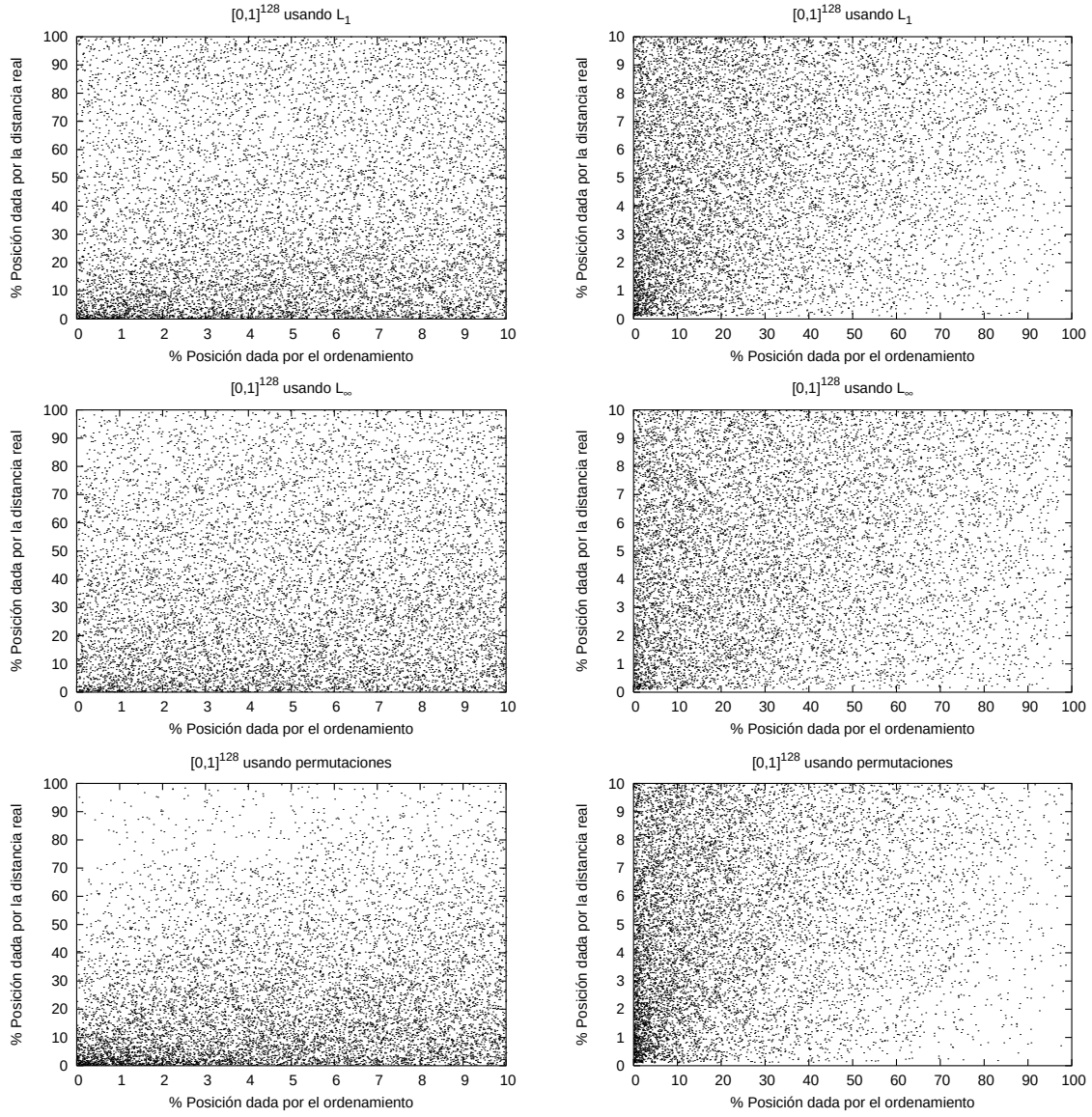


Figura A.3: Acercamiento de las nubes de puntos dimensión 128 y 256 pivotes/permutantes (figura A.2). Las figuras de la izquierda corresponden a la precisión y los de la derecha a la recuperación. De arriba hacia abajo, L_1, L_∞, y permutaciones.

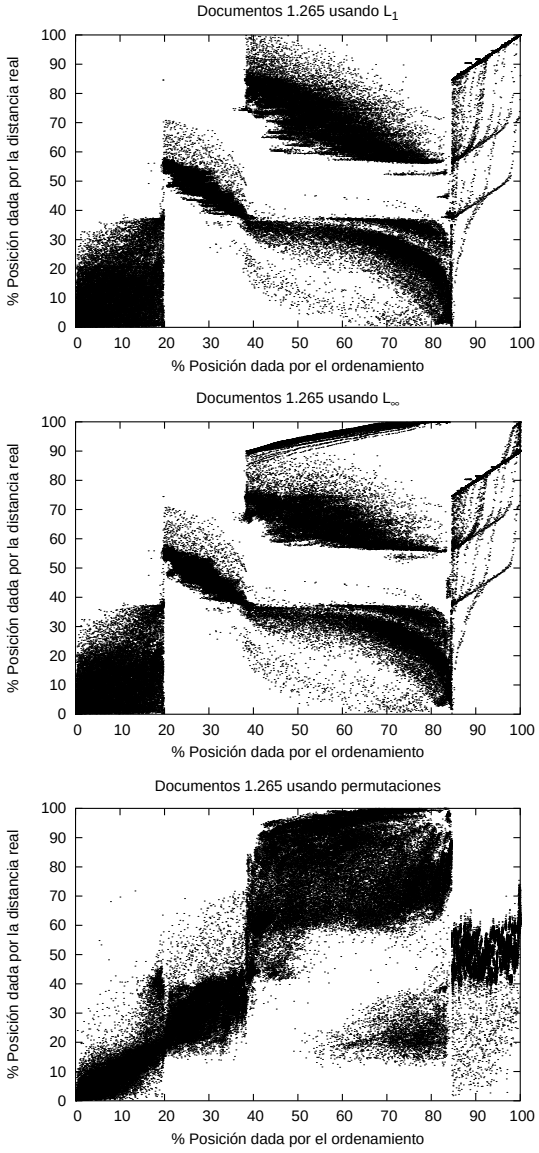


Figura A.4: Nubes de puntos para la base de datos de documentos, usando 128 pivotes/permutantes. De arriba hacia abajo, L_1, L_∞, y permutaciones.

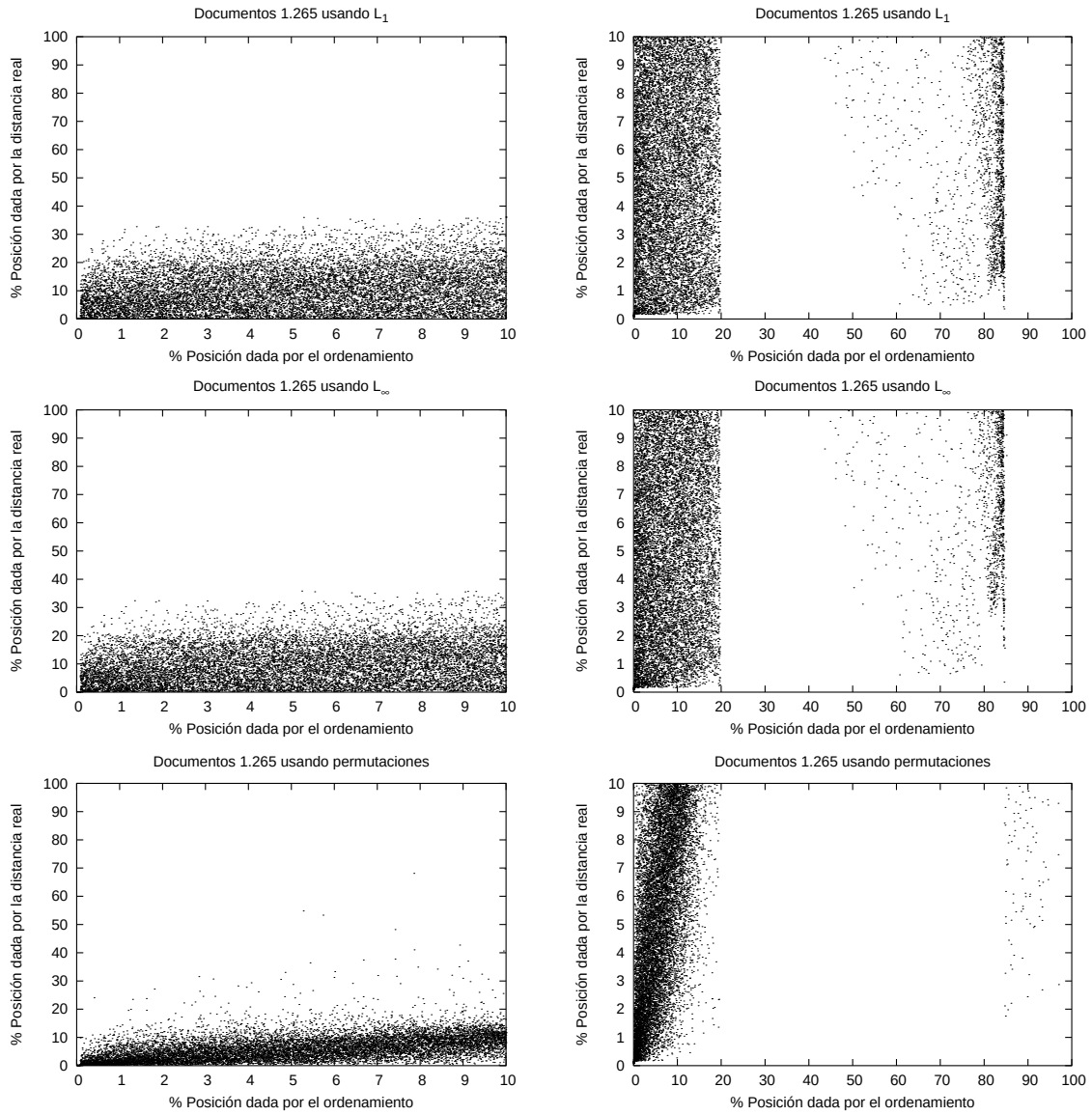


Figura A.5: Acercamiento de las nubes de puntos para la base de datos de documentos, usando 128 pivotes/permutes (figura A.4). Se muestran los gráficos de Precisión (lado izquierdo) y recuperación (lado derecho). De arriba hacia abajo L_1, L_∞, y permutaciones.

A.3. Base de datos de Caras FERET

En el caso de la base de datos de caras, se analizó sólo la que tiene 762 elementos y 761 componentes (descrita en la sección 7.2.1). Las nubes de puntos con los tres criterios manejados se muestra en la figura A.6. En este caso se usaron 64 pivotes/permutantes.

Los gráficos correspondientes a la precisión y recuperación de la figura A.6 se muestran en las figura A.7.

A.4. Diccionario

La nube de puntos para el espacio del diccionario de palabras en español bajo la distancia de edición se muestra en la figura A.8. En estos gráficos, nuevamente, el criterio de las permutaciones tiene un mejor desempeño que los otros dos, de hecho es el caso más evidente. La forma de líneas en los gráficos seguramente es generada debido al uso de una distancia discreta.

Los gráficos de precisión y recuperación de la figura A.8 se muestran en la figura A.9, lado izquierdo y derecho, respectivamente.

A.5. Espacios no Métricos

En el caso de los espacios no métricos se utilizaron nuevamente vectores en el cubo unitario distribuidos uniformemente, ahora bajo la norma fraccionaria L_p, con $0 < p < 1$. La dimensión manejada en este espacio fue 32, por considerarse intermedia a las presentadas.

A.5.1. L_p con $p = 0,2$

La norma fraccionaria usada en esta sección fue L_p con $p = 0,2$. Las nubes de puntos correspondientes se muestran en la figura A.10. Se usaron 64 pivotes/permutantes y los tres criterios de ordenamiento.

Los gráficos de precisión y recuperación de la figura A.10 se muestran en la figura A.11. Estos evidencian la superioridad de las permutaciones para ordenar esta base de datos.

A.5.2. L_p con $p = 0,8$

En los experimentos presentados en esta tesis también se usó $p = 0,8$ en la norma fraccionaria. En esta sección analizamos la nube de puntos de este espacio bajo los tres criterios de ordenamiento, en la figura A.12.

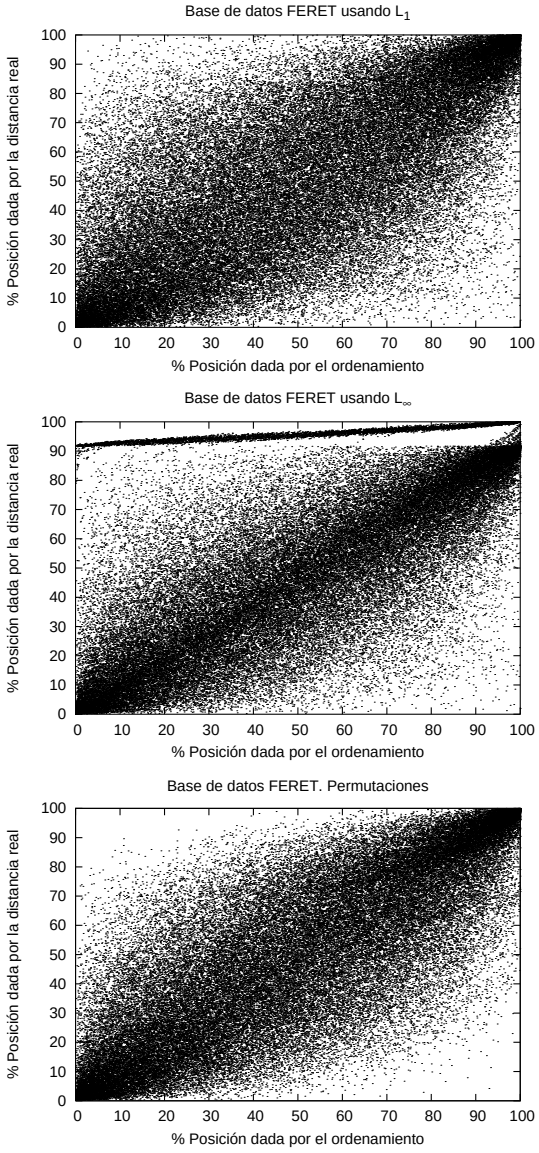


Figura A.6: Nubes de puntos para la base de datos de caras, usando 64 pivotes/permutantes. De arriba hacia abajo, L_1, L_∞, y permutaciones.

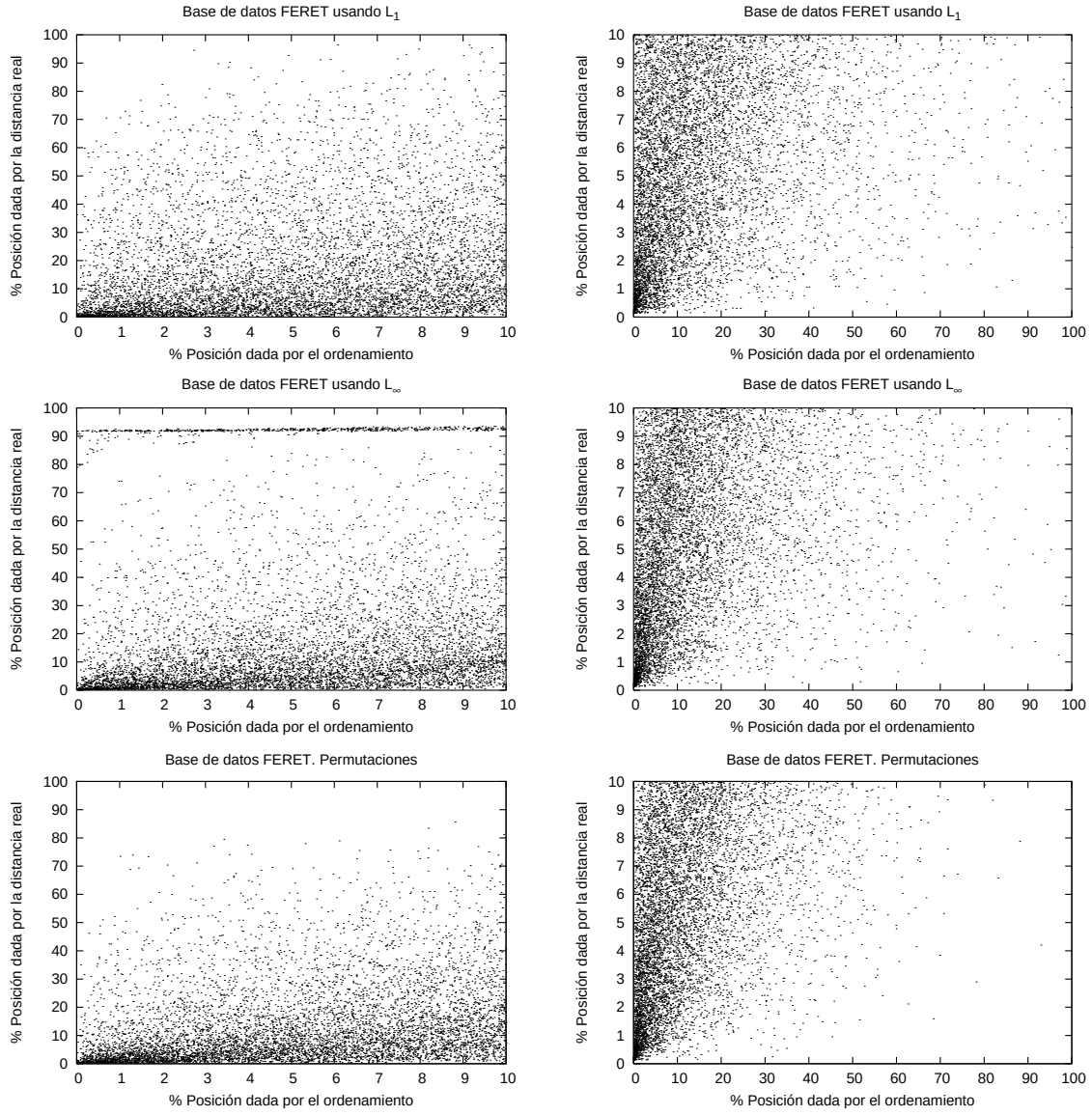


Figura A.7: Acercamiento de las nubes de puntos para la base de datos de caras, usando 64 pivotes/permutantes (figura A.6). Precisión (lado izquierdo) y recuperación (lado derecho). De arriba hacia abajo, L_1, L_∞, y permutaciones.

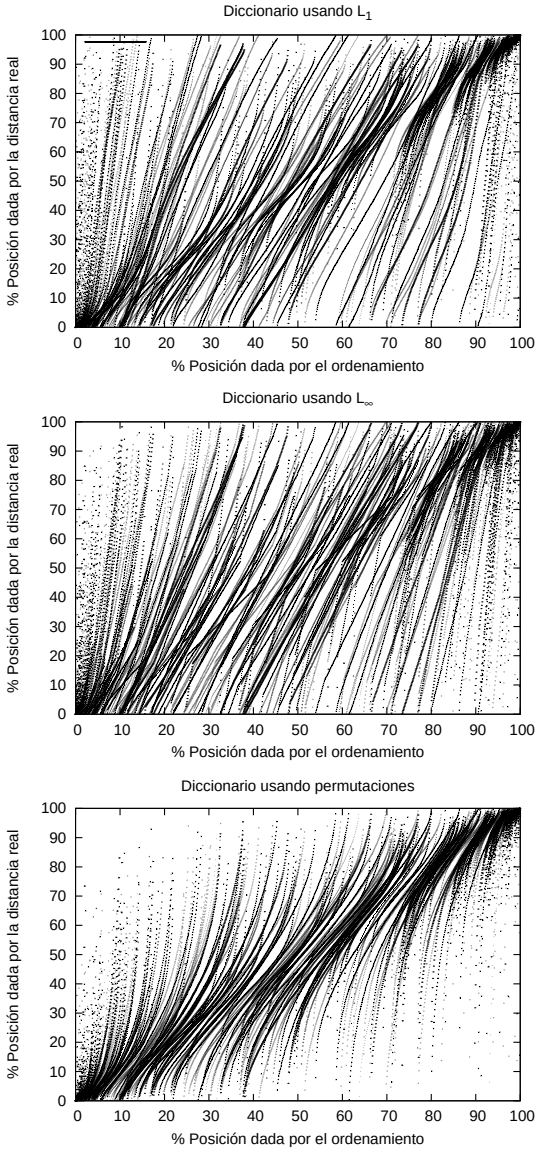


Figura A.8: Nubes de puntos para la base de datos del diccionario, usando 128 pivotes. De arriba hacia abajo, L_1, L_∞, y permutaciones.

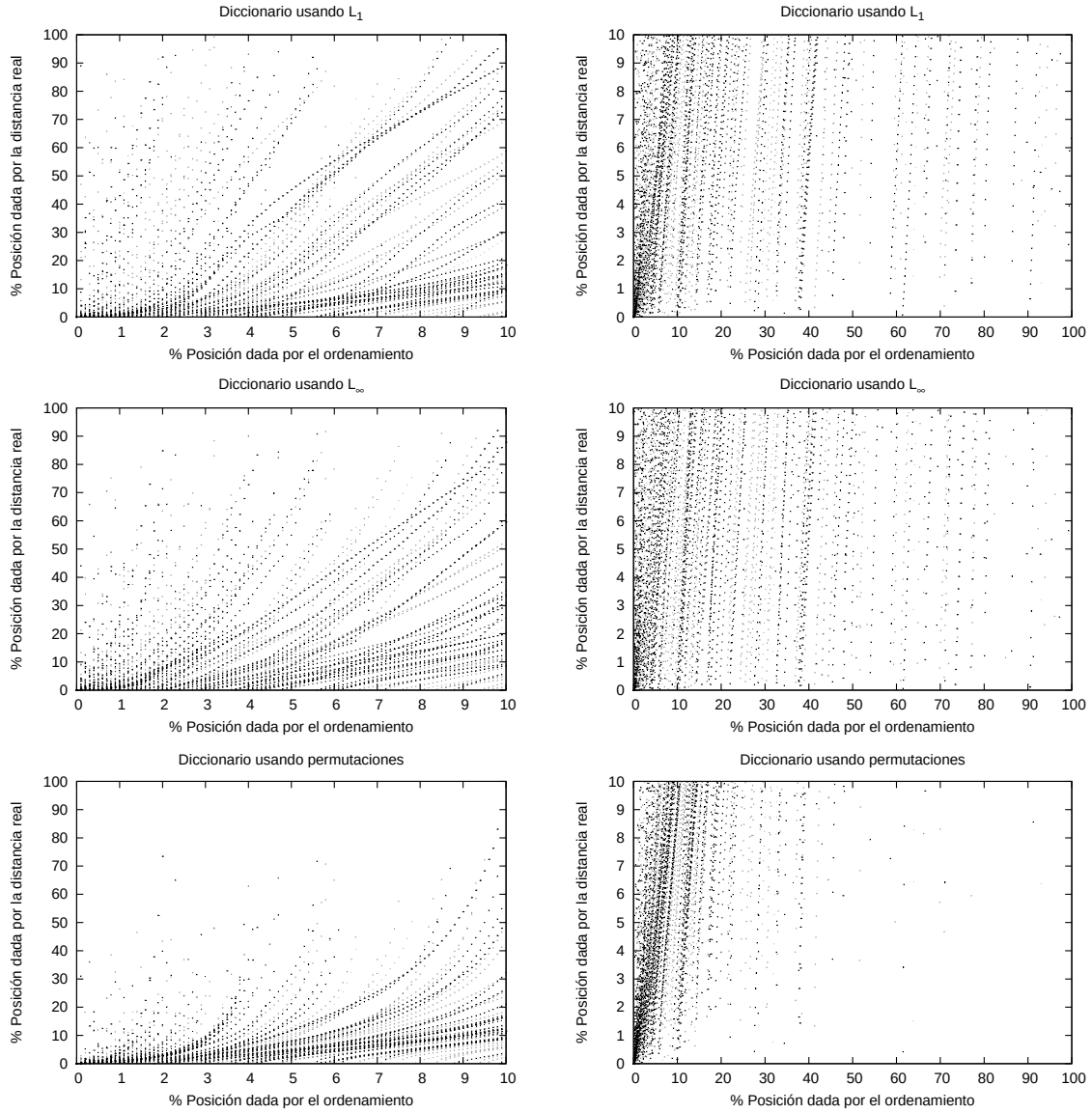


Figura A.9: Acercamiento de las nubes de puntos para la base de datos del diccionario, usando 128 pivotes/permutantes (figura A.8). Se muestran los gráficos de precisión (lado izquierdo) y recuperación (lado derecho). De arriba hacia abajo L_1, L_∞, y permutaciones.

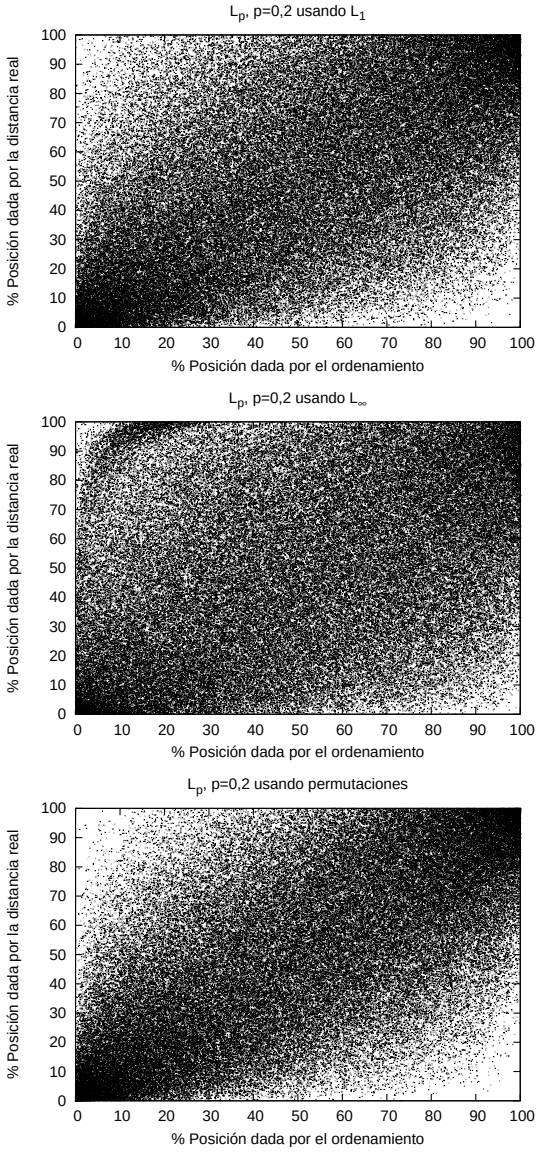


Figura A.10: Nubes de puntos para un espacio distribuido uniformemente en dimensión 32, usando 64 pivotes/permutantes. De arriba hacia abajo L_1, L_∞, y permutaciones.

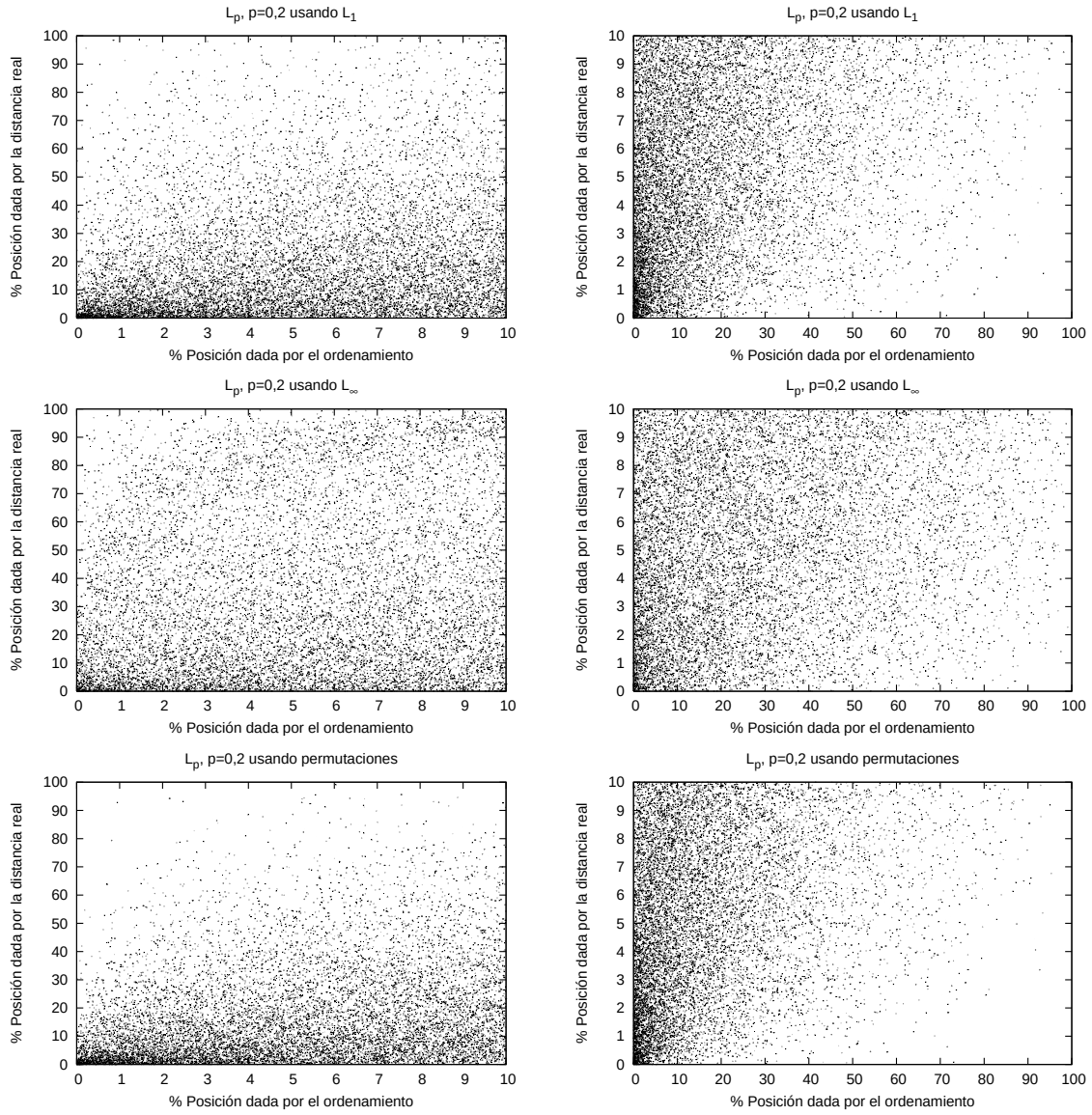


Figura A.11: Acercamiento de las nubes de puntos para un espacio distribuido uniformemente y la norma fraccionaria L_p, $p = 0,2$, usando 64 pivotes/permutantes y dimensión 32 (figura A.10). Se muestran los gráficos de precisión (lado izquierdo) y recuperación (lado derecho). De arriba hacia abajo L_1, L_∞, y permutaciones.

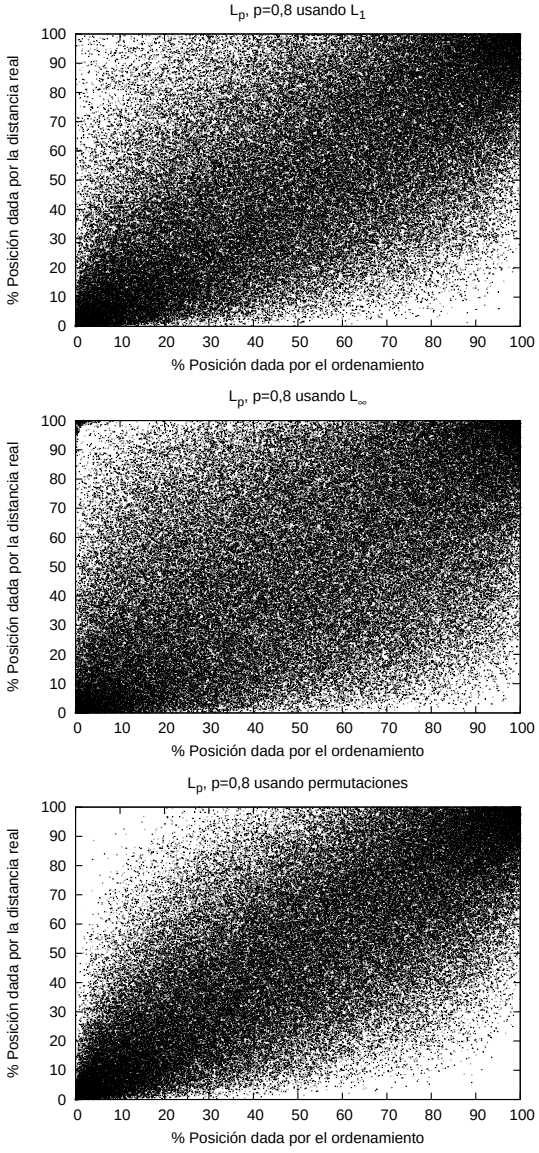


Figura A.12: Nubes de puntos para un espacio distribuido uniformemente en dimensión 32, usando 64 pivotes/permutantes, usando L_p con $p = 0,8$. De arriba hacia abajo L_1, L_∞, y permutaciones.

El acercamiento de los gráficos la figura A.12 que corresponden a la precisión y recuperación, se muestran en la figura A.13. En estos espacios y para esta norma fraccionaria el criterio de permutaciones tiene un desempeño superior a los otros dos criterios L_1 y L_∞.

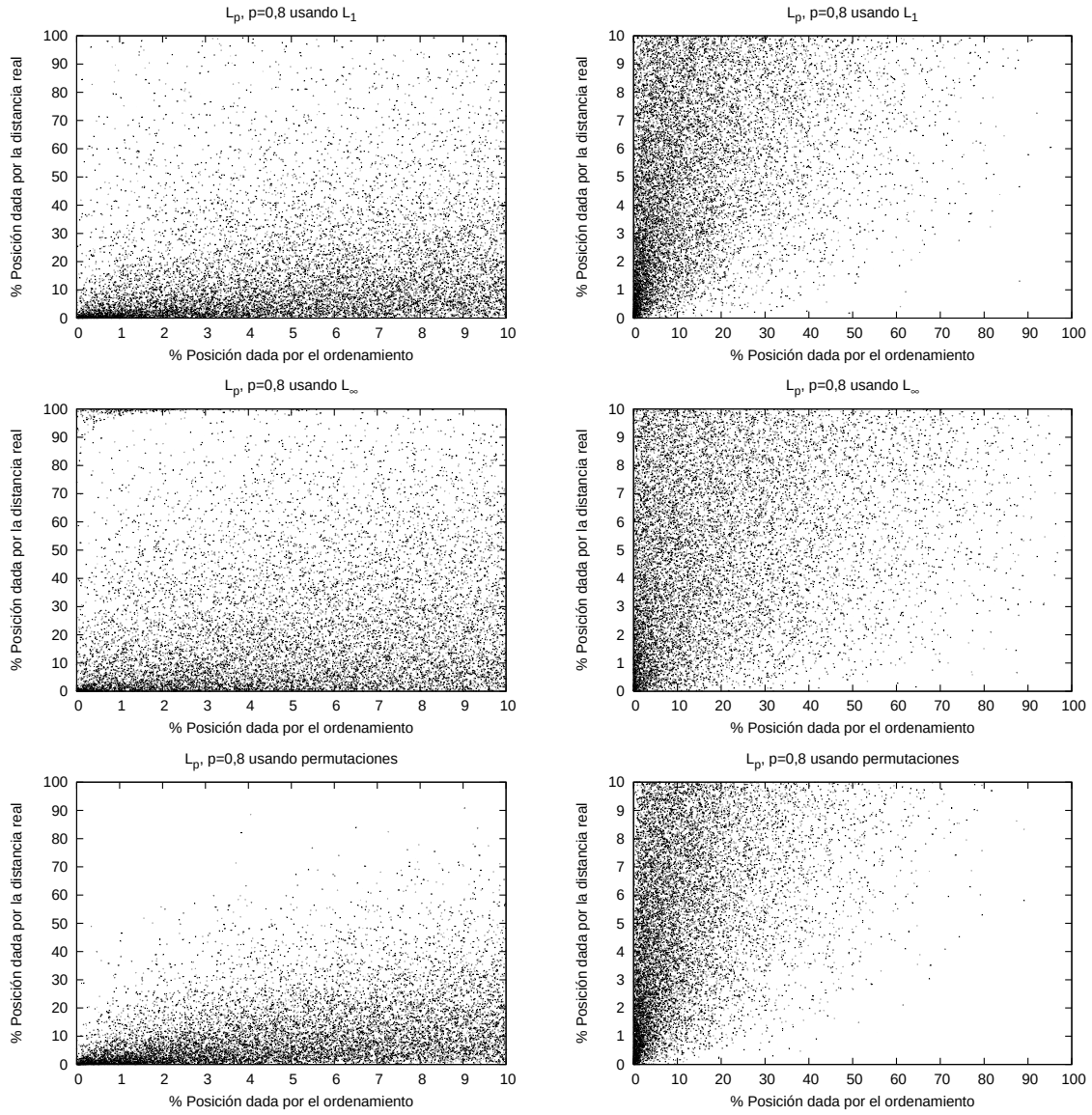


Figura A.13: Acercamiento de las nubes de puntos para un espacio distribuido uniformemente en el cubo unitario, dimensión 32 y usando 64 pivotes/permutantes (figura A.12). Se muestran los gráficos de precisión (lado izquierdo) y recuperación (lado derecho). De arriba hacia abajo L_1, L_∞, y permutaciones.