Índice General

Capítulo 1

Introducción

La búsqueda es un problema fundamental en las ciencias computacionales, y está presente virtualmente en cada aplicación de computadora. Las aplicaciones simples poseen problemas de búsqueda simples, mientras que una aplicación más compleja podría requerir, en general, un tipo de búsqueda más sofisticado.

La operación de búsqueda ha sido tradicionalmente aplicada a "datos estructurados", un ejemplo de esto es la información numérica o alfabética la cual es buscada en forma exacta; es decir, dada una consulta, deben recuperarse los números o cadenas los cuales son *exactamente iguales* a la consulta. Las bases de datos tradicionales son construidas en torno al concepto de búsqueda exacta, y éstas están divididas en registros; cada registro tiene una llave completa comparable. Las consultas a la base de datos regresan todos los registros cuyas llaves coinciden con la llave de búsqueda. Las búsquedas más sofisticadas tales como consultas de rango en llaves numéricas o búsquedas de prefijos en llaves alfabéticas, aún confían en el concepto de que dos llaves son o no iguales. En años recientes, cuando las bases de datos han incluido la habilidad para guardar nuevos tipos de datos tales como imágenes, sonido, etc., la búsqueda todavía ha sido hecha sobre un número predeterminado de llaves numéricas o alfabéticas.

Con la evolución de la información y las tecnologías de comunicación, los bancos de información no estructurada han emergido. Los nuevos tipos tales como texto libre, imágenes, audio y video tienen que ser consultados, y esto ya no es posible para la información estructurada en llaves y registros, por ser muy difícil estructurar este tipo de datos y por restringir de antemano el tipo de consulta que se pueda realizar. Cuando es posible implementar una estructura clásica, las nuevas aplicaciones tales como la minería de datos que requieren accesar a la base de datos por algún campo (no solo a los marcados como llaves), se enfrentan a serios problemas. He aquí la necesidad de contar con nuevos modelos de búsqueda en bancos de información no estructurados, como lo es la búsqueda por proximidad.

1.1 Introducción al problema

El concepto de "búsqueda de similaridad" ó "búsqueda de proximidad", consiste en buscar a los elementos de la base de datos que sean similares o cercanos a un elemento de consulta dado. La similaridad es modelada con una función de distancia [NBY00] que satisface la desigualdad del triángulo, y el conjunto de objetos es llamado espacio métrico.

En muchas aplicaciones, el problema en espacios métricos generales es trasladado a un espacio vectorial (por ejemplo, los objetos están representados como puntos en m-dimensiones con alguna interpretación geométrica de similaridad). Desafortunadamente los algoritmos existentes son muy sensibles a la dimensión del espacio vectorial, es decir, tienen una dependencia exponencial en la dimensión del espacio (esto es llamado "la maldición de la dimensión"). En términos prácticos, es considerado que el problema es intratable en más de 20 dimensiones. Por esta razón, varios autores han propuesto el uso de técnicas de indexamiento basadas en *distancias*, las cuales usan solo la distancia entre puntos y evitan cualquier referencia a coordenadas, en un intento por evitar la maldición de la dimensión.

El problema de los K vecinos más cercanos aparece en diversas áreas en las cuales las técnicas computacionales son aplicadas, por ejemplo para encontrar agrupamientos en nubes de datos (clustering) [Chá97], precondicionamiento de algoritmos de indexamiento, matrices de confusión para probar el algoritmo KNN (K-Nearest Neighbors), algoritmos de Ranking, pruebas de hipótesis y estimación de densidades.

En este trabajo se presenta una solución al problema de *todos* los K vecinos más cercanos. Este problema consiste en que dada una base de datos de n puntos o elementos en un espacio métrico, se necesita encontrar para cada elemento sus K vecinos más cercanos (la figura 1.1, muestra el resultado de encontrar todos los K vecinos de una base de datos). La complejidad del problema es medida en cálculos de distancia, de antemano se sabe que usando la fuerza bruta el problema es solucionado con n^2 cálculos de distancia (debido a que cada elemento calcularía la distancia hacia toda la base de datos).

La solución planteada para encontrar los K vecinos mas cercanos consiste en hacer uso de un algoritmo de indexamiento para reducir los cálculos de distancia; el algoritmo de indexamiento usado es el FQTrie (Fixed Query Trie), el cual una combinación de lo mejor de los algoritmos FQT (Fixed Query Tree) y FQA (Fixed

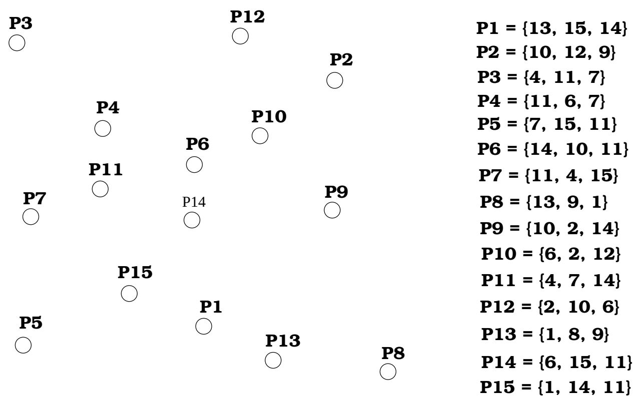


Figura 1.1: Ejemplo de una base de datos, donde cada elemento ya conoce a sus K vecinos más cercanos

Query Array).

1.2 Motivación

Existen muchas aplicaciones que utilizan el concepto de búsqueda de similaridad, entre ellas podemos mencionar: Consulta por contenido en bases de datos estructuradas, búsqueda por contenido en datos multimedia, recuperación de texto, biología computacional, reconocimiento de patrones, compresión de audio y video, etc.; no se ahondará en ellas por no ser el tema central de este trabajo. Ahora bien entre las aplicaciones que usan los K vecinos más cercanos podemos mencionar las siguientes:

- ***Agrupamientos***

  Este tipo de aplicaciones agrupan datos con algunas características comunes, y son aplicadas a la clasificación de resultados en recuperación de información [Chá97, MB96], por ejemplo, supóngase que estamos recuperando documentos de texto y que hemos obtenido miles de documentos que satisfacen nuestros parámetros de búsqueda; nuestro objetivo es agruparlos en documentos que tratan sobre temas semejantes. En este punto cabe hacer notar que la parte más importante en la resolución de este problema consiste en la localización de una buena medida de similaridad entre documentos.

  A continuación se presenta un ejemplo ilustrativo (no técnico) de agrupamientos (cúmulos); el síndrome de Tarzán. Tarzán es un niño que es dejado por accidente en la selva, y es criado por las bestias. En la selva hay multitud de objetos, de los cuales desconoce el nombre. Algunos de estos objetos son comestibles, otros son venenosos, curativos, peligrosos, inofensivos, etc. ¿Como

Tarzán sobrevivió hasta la edad adulta?, es de suponer que encontró una manera de clasificarlos sin morir en el intento. Es claro que si no hablamos el mismo lenguaje no vamos a asignar las mismas "etiquetas" a los mismos objetos; más aún, Tarzán puede encontrar una larga serie de subdivisiones en las etiquetas que nosotros podemos asignar. Una última observación: para efectos de sobrevivencia es irrelevante el tipo de etiqueta (números, letras, símbolos diversos) que se le asigne a cada clase de objetos, siempre y cuando sean distintas para cada una de ellas y puedan ser organizadas.

- ***Precondicionamiento de algoritmos de indexamiento***

  La búsqueda de los K vecinos más cercanos se puede utilizar para realizar el preprocesamiento de algoritmos de indexamiento, como los árboles desbalanceados multiraices (multi root unbalanced tree). La idea es hacer un agrupamiento de los datos para resolver un problema de indexamiento.

- ***Matrices de confusión para probar el algoritmo KNN***

  La clasificación supervisada usada en la minería de datos es un método empleado para el análisis de grandes colecciones de información, si se construye una matriz de confusión sin necesidad de una supervisión buscando *todos* los K vecinos más cercanos, se logra una clasificación automática y con esta se podrían generar árboles de decisión más eficientes [AGG].

- ***Algoritmos de Ranking***

  Los algoritmos de Ranking representan un técnica para recuperación de documentos. Una de las ventajas de este método es que se elimina la necesidad de entender modelos teóricos, como en otros algoritmos; los algoritmos de Ranking están orientados al usuario final los cuales pueden recuperar información usando el lenguaje natural, otra característica es que los resultados están ordenados por un *Ranking* basados en una co-ocurrencia de los términos de la consulta. En una de las técnicas que usan los algoritmos Ranking "podar (*pruning*)" su principal cuello de botella en el proceso de búsqueda es ordenar las acumulaciones para grandes conjuntos de datos [Yaz], el cual puede ser solucionado encontrando *todos* los KNN.

- ***Pruebas de hipótesis***

  La búsqueda de los K vecinos más cercanos se usa en la prueba de hipótesis o estimación de densidades para conocer los puntos que están alrededor de un punto en una posición dada debido a que estos puntos contribuirán a la función de densidad en esa posición. Cuando se hace un análisis con agrupamiento se detecta el máximo de la función de densidad local [USC].

Capítulo 2

Conceptos Básicos

Todas las aplicaciones presentadas en las secciones previas comparten una estructura común, la cual es en esencia encontrar los objetos más cercanos bajo una función conveniente de similaridad, entre un conjunto finito de objetos. En este capítulo se explican los principales conceptos que se utilizarán a lo largo de este trabajo.

2.1 Definiciones

Espacios métricos y consultas de proximidad

A continuación se muestra la notación básica del problema de búsquedas, las cuales satisfacen la proximidad y el modelo usado [CNBYM99].

El conjunto X denotará el universo de objetos válidos. Un subconjunto finito de ellos, U, de tamaño $n = |\mathrm{U}|$, es el conjunto de objetos donde se aplicará la búsqueda. U podría ser llamado diccionario, base de datos o simplemente nuestro conjunto de puntos. La función

$d : X \times X \to R$

Denotará una medida de distancia entre objetos (por ejemplo la distancia más pequeña, la más cercana o más similar a la que están los objetos). La función de distancia [NBY00] tiene las siguientes propiedades:

a) $\forall x, y \in X, d(x, y) \geq 0$ **Positiva**

b) $\forall x, y \in X, d(x, y) = d(y, x)$ **Simetrica**
c) $\forall x \in X, d(x, x) = 0$ **Reflexiva**
y en algunos de los casos
d) $\forall x, y \in X, x \neq y \Rightarrow d(x, y) > 0$ **Positividad estricta**

Las propiedades de similaridad mencionadas arriba solo garantizan una definición consistente de la función y no pueden ser usadas para guardar comparaciones en una consulta de proximidad. Si d es una distancia, y si satisface la desigualdad del triángulo, es decir,

e) $\forall x, y, z \in X, d(x, y) \leq d(x, z) + d(z, y)$

entonces el par (X, d) es llamado un espacio métrico.

Si la función de distancia no satisface la propiedad de estrictamente positiva (d) entonces el espacio es llamado *espacio pseudo-métrico* Sin embargo por simplicidad no consideramos espacios pseudo-métricos en este trabajo, todas las técnicas presentadas son fácilmente adaptadas a ellos por sencillez identificamos todos los objetos a distancia cero son el mismo objeto, tomando en cuenta esto, tenemos que

$$d(x, y) = 0 \Rightarrow \forall z, d(x, z) = d(y, z)$$

Existen básicamente tres tipos de consultas de interés en espacios métricos

a) Consulta de rango o de proximidad.- La cual consiste en recuperar todos los elementos los cuales están dentro de una distancia r de q. Esto es $\{u \in U \mid d(q, u) \leq r\}$. Denotaremos esta consulta por $(q, r)_d$.

b) Consulta del vecino más cercano.- Este tipo de consulta recupera el elemento más cercano a q, en U, Esto es, $\{u \in U \mid \forall v \in U, d(q, u) \leq d(q, v)\ \}$ Podemos dar una distancia máxima r$*$ tal que si el elemento más cercano esta a una distancia mayor que r$*$ no queremos ningún reporte.

c) Consulta de los k-vecinos más cercanos.- En la cual se recuperan los K elementos más cercanos a q en U. Esto es, recuperar un conjunto $A \subset U$ tal que $\mid A \mid = k$ y $\forall u \in A, v \in U - A, d(q, u) \leq d(q, v)$.

El tipo básico de consulta es el correspondiente al inciso (a), en la figura 2.1 se ilustra este tipo de consulta. Una consulta de proximidad será por lo tanto, un par

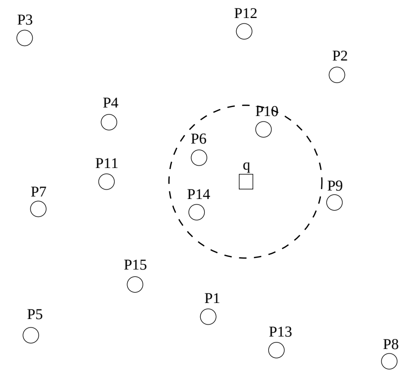


Figura 2.1: Ejemplo de una consulta de rango (q) en un conjunto de puntos ($p_1 - p_{15}$)

$(q, r)_d$. El conjunto $\{u \in U, d(q, u) \leq r\}$ será llamado resultado de la consulta de proximidad.

Existen otros dos tipos de consultas que son solucionados usando una variante de la consulta de rango. Por ejemplo, la consulta de tipo (b) la cual es normalmente solucionada con una consulta de rango donde el radio r inicialmente es infinito y este es reducido con los elementos más cercanos a la consulta a medida que son encontrados. Normalmente se trata de obtener los elementos más cercanos tan rápido como sea posible (el problema es siempre más fácil para radios pequeños). El otro tipo de consulta (el tipo c) es normalmente solucionado como una variante del tipo (b), donde los K elementos más cercanos son guardados y la r activa es la distancia más larga de estos elementos a la consulta (Los elementos más lejanos no son de interés).

Es claro que cualquier tipo de consulta puede ser resuelta examinando el diccionario completo U. De hecho, si no pudiéramos preprocesar los datos, por ejemplo construir una estructura de datos indexada, entonces una examinación exhaustiva es la única manera de proceder. En nuestro modelo la función de distancia d es costosa y por lo tanto, el número de cálculos de distancia procesadas será la medida de complejidad del algoritmo.

Ahora bien un algoritmo de indexamiento es un procedimiento independiente para construir una estructura de datos (o índice), diseñada para ahorrar cálculos de distancia cuando se responde a una consulta de proximidad. Esta estructura de datos puede ser costosa de construir, pero ahorrará cálculos de distancia sobre muchas consultas a la base de datos. Todos los algoritmos de indexamiento trabajan en base a descartar elementos usando la desigualdad del triángulo (de hecho la única propiedad para ahorrar cálculos de distancia).

Finalmente, el objetivo de este trabajo, es realizar un algoritmo que sea capaz de encontrar a todos los K vecinos más cercanos, utilizando un índice de complejidad lineal en su construcción, y con esto reducir el número de cálculos de distancia, que se compara con la solución al problema aplicando fuerza bruta, el cual es resuelto usando n^2 cálculos de distancia.

2.1.1 Espacio Vectorial

Si los elementos de un espacio métrico (X, d) son tuplas de números reales (de hecho tuplas en cualquier campo) entonces el par es llamado *espacio vectorial dimensional finito*, o espacio vectorial.

Un espacio vectorial m-dimensional es un espacio métrico particular donde los objetos son identificados con m coordenadas de valores reales de aquí que existan numerosas opciones para usar una función de distancia, entre ellas se encuentran la distancia de Manhatan, la distancia euclidiana, la distancia de Minkowski [CNBYM99] por mencionar algunas. En este trabajo se usó la distancia euclidiana, la cual consiste en nuestra noción de distancia espacial. Los puntos a la misma distancia r a un punto p dado, forman una esfera de diámetro $2r$ centrado en el punto.

Capítulo 3

Estado del arte

En este capítulo se presentará una revisión del estado del arte de los algoritmos que solucionan el problema de los K vecinos más cercanos, pero antes se describirá de manera breve el estado del arte de los algoritmos que realizan la construcción de un índice.

3.1 Algoritmos que usan el diagrama de Voronoi

Los diagramas de Voronoi fueron por primera vez discutidos por Peter Lejeune-Dirichlet en 1850, pero fue hasta mucho después en 1908 que estos diagramas fueron escritos por Voronoi, de ahí su nombre [RI].

Un diagrama de Voronoi es una estructura geométrica que representa información de proximidad acerca de un conjunto de puntos u objetos es decir, el diagrama de Voronoi de un conjunto de objetos geométricos es una partición del espacio en celdas cada una de la cuales contiene una colindancia con sus puntos más cercanos, obsérvese la figura 3.1.

Los diagramas de Voronoi tienen numerosas aplicaciones en diversas áreas que van desde la arqueología hasta la zoología. Por ejemplo, los diagramas de Voronoi son esenciales en la solución de un problema de comunicación sin hilos. Dado un conjunto de subestaciones con n-micro celdas. ¿Cómo pueden ser pre-procesados de modo que para que cualquier llamada celular arbitraria la estación más cercana lleve la llamada?.

Los grafos aproximados o diagramas de Voronoi, han sido usados para consultas de proximidad en espacios vectoriales y son una estructura fundamental en la geometría computacional para solucionar el problema de puntos más cercanos. Aunque

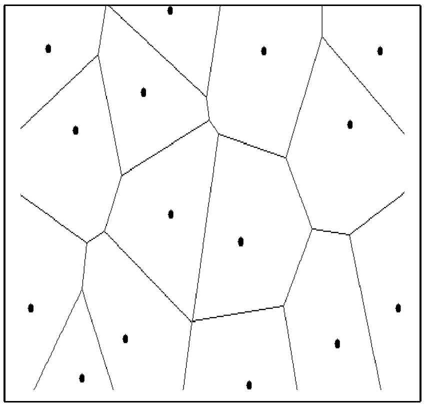


Figura 3.1: Diagrama de Voronoi para algunos puntos

es realmente un desafío generalizarlo a espacios métricos, porque los algoritmos a construir dependen fuertemente de la información de las coordenadas.

3.2 Triangulación de Voronoi

El dual de los diagramas de Voronoi, o también llamado "triangulación de Voronoi" donde una línea es dibujada entre cada punto y todos los puntos cercanos a este. También se le conoce como "la triangulación de Delaunay", véase la figura 3.2, cabe mencionar que en las figuras 3.1 y 3.2 el conjunto de puntos es el mismo y en ellas se pretende mostrar la construcción de un diagrama de Voronoi y la triangulación de Delaunay para los mismos puntos.

El concepto de Voronoi por sí mismo ha inspirado varias técnicas construyendo, aproximaciones más o menos afines al grafo de Voronoi o su dual, la triangulación de Delaunay [Che]. En esta línea se puede mencionar un algoritmo reciente [Nav99a], donde la idea principal en éste y otros algoritmos basados en la triangulación de Delaunay es dividir el espacio para buscar por técnicas espaciales la consulta.

3.3 Algoritmos basados en pivotes

Los algoritmos que usan pivotes, son herramientas efectivas para búsquedas de proximidad en espacios métricos. Ellos permiten negociar entre el espacio ocupado y el número de cálculos de distancia desarrollados en tiempo de consulta.

Una vista abstracta de los algoritmos basados en pivotes es la siguiente. Seleccio-

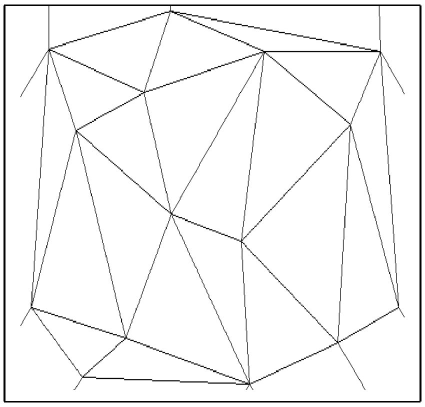


Figura 3.2: Formación de la triangulación de Delaunay sobre los puntos de la figura 3.1

namos un conjunto de P pivotes $\{p_1, ..., p_P\}$. En tiempo de indexamiento, para cada elemento de la base de datos a calculamos y guardamos $\Phi(a) = (d(a,p_1)...d(a,p_P))$. En tiempo de la consulta, dada una consulta $(q,r)_d$, calculamos $\Phi(q) = (d(q,p_1), ..., (q,p_P))$. Ahora podemos descartar cada $a \in U$ tal que, para algún pivote p_i, $\mid d(q,p_i) - d(a,p_i) \mid > r$, o lo que es lo mismo, descartamos cada a tal que

$$\max_{1 \leq i \leq k} \mid d(q,p_i) - d(a,p_i) \mid = L_\infty(\Phi(a), \Phi(q)) > r$$

Esto muestra que los algoritmos basados en pivotes pueden ser vistos como un mapeo Φ del espacio métrico original (X,d) a un espacio vectorial P-dimensional con la distancia L_∞, es decir (R^P, L_∞).

Los algoritmos kd-trees [Ben75] desarrollan una descomposición binaria jerárquica del espacio vectorial. Para cada nivel las ramificaciones izquierdas y derechas, cuentan con puntos a la izquierda o a la derecha de un umbral en una coordenada particular. Las coordenadas se alternan a cada nivel. Para espacios métricos generales la auscencia de coordenadas hizo urgente el diseño de reglas alternativas para descomposición de espacios y localización de objetos. Una familia entera de algoritmos son directamente descendientes de la estructura de los kd-trees. En lugar de usar las coordenadas directamente, estos algoritmos usan la distancia a un conjunto de objetos distinguidos de la base de datos llamados llaves, puntos ventajosos o *pivotes*. Combinando esto con la desigualdad del triángulo se obtiene una regla de descarte similar a la de los kd-trees.

La mayoría de los esquemas descendientes de los kd-trees son estructuras de datos basados en árboles que definen una descomposición jerárquica donde las celdas espaciales coinciden con las hojas en los árboles. Un ejemplo simple es el Burkhard-

Keller Tree (BKT) [BK73], el cual es una estructura de datos diseñada para una función de distancia con valores discretos. Cada nodo del árbol corresponde a un pivote p diferente, y cada rama descendiente está a cierta distancia desde p. Esto es, todos los elementos a distancia i de p son puestos en el i-th subárbol del nodo correspondiente. Los subárboles son construídos recursivamente con la misma regla. Para una consulta $(q,r)_d$, descendemos en el árbol entrando sólo en los subárboles que cumplan con $(q,p)_d - r < (q,r)_d < (q,p)_d + r$, los otros elementos son descartados por la desigualdad del triángulo.

3.4 Indices y particiones

Dado un conjunto X, una *partición* $\pi(X) = \{\pi_1, \pi_2, ...\}$ es un subconjunto del conjunto potencia $P(X)$ tal que cada elemento del conjunto pertenece exactamente a una partición, es decir, $\cup\pi_i = X$ y $\forall i \neq j$, $\pi_i \cap \pi_j = 0$

Una relación, es denotada por $\sim$, es un subconjunto del producto cruz $X \times X$ (el conjunto de pares ordenados). Dos elementos x, y se dice que están relacionados, denotado por $x \sim y$, si el par (x, y) están en el mismo subconjunto. Una relación $\sim$ se dice que es una *relación de equivalencia* si satisface, para todo $x, y, z \in X$, las propiedades de:

a) $(x \sim x)$	**Reflexividad**
b) $(x \sim y \Leftrightarrow y \sim x)$	**Simetria**
c) $(x \sim y \bigwedge y \sim z \Rightarrow x \sim z)$	**Transitividad**

Se muestra que cada partición $\pi(X)$ induce una relación de equivalencia $\sim$ e inversamente cada relación de equivalencia induce una partición. Dos elementos están relacionados si ellos pertenecen a la misma partición [Chi95]. Cada elemento π_i de la partición es entonces llamado una *clase de equivalencia*. Una clase de equivalencia es a menudo llamada representativa (cualquier elemento de π_i puede ser tomado como un representativo). Una definición alternativa de una clase de equivalencia de un elemento x es el conjunto de todos y tales que $x \sim y$. Denotaremos la clase de equivalencia de x como $[x] = \{y : x \sim y\}$.

Por ejemplo, dado un conjunto de números reales R, la relación $x \sim y \Leftrightarrow [x] = [y]$ es una relación de equivalencia. Un ejemplo de una clase equivalente en esta partición es el conjunto de todos los números reales tales que su suelo es uno. Las clases de equivalencia del número 1 ([1]) y del número 1.5 ([1.5]) son la misma (lo mismo ocurre en el intervalo entre los números 1 y 2 representado como $[1, 2)$), hemos cambiado sólo su representación.

Dado un conjunto X y una relación de equivalencia $\sim$, obtenemos el cociente $\pi(X) = X/\sim$. Esto indica que el conjunto de clases equivalentes o co-conjuntos, son obtenidos cuando aplicamos la relación de equivalencia al conjunto X. En el ejemplo citado, los co-conjuntos obtenidos de los números reales con la relación de equivalencia dado son los intervalos $[i, i+1)$, para todos los enteros i.

La relevancia de las clases de equivalencia en este trabajo viene de la posibilidad de usarlas en un espacio métrico así que el nuevo espacio métrico es derivado del conjunto cociente. Este nuevo espacio métrico es una versión engrosada del original.

Las clases de equivalencia obtenidas con una relación de equivalencia de un espacio métrico pueden ser consideradas por sí mismas como puntos en un nuevo espacio métrico, así que definimos una función de distancia en su nuevo espacio métrico.

Introducimos una nueva función $Do: \pi(X) \times \pi(X) \longrightarrow R$ definida ahora en el cociente. Dado que Do, está definida entre clases equivalentes, una selección natural es $Do([x], [y]) = \inf_{x \in [x], y \in [y]}\{d(x, y)\}$, así que esto da los valores máximos posibles que mantienen el mapeo contractivo (Por ejemplo $Do([x], [y]) \leq d(x, y)$ para algún x, y). Desafortunadamente, Do no satisface la desigualdad del triángulo, solo las propiedades de positivo, positividad estricta y reflexividad. Por lo tanto, Do por sí misma no es conveniente para propósitos de indexamiento.

Podemos usar cualquier distancia D que satisfaga las propiedades del espacio métrico y su límite inferior Do (Por ejemplo $D([x], [y]) \leq Do([x], [y])$). Llamamos D la *extensión* de d. De aquí que D es una distancia, $(X \mid\sim D)$ es un espacio métrico y sin embargo podemos hacer consultas en el co-conjunto de la misma manera que hemos hecho en el conjunto. Redefinimos la salida de una consulta en el co-conjunto como $([q], r)_D = \{u \in U, D([u], [q]) \leq r\}$ (aunque formalmente deberíamos recuperar las clases, no los elementos). Dado que el mapeo es contractivo ($D([x], [y]) \leq d(x, y)$) podemos convertir un problema de búsqueda en otro, esperemos que en un más simple. Para una consulta dada $(q, r)_d$ descubrimos cuales clases de equivalencia pertenecen al punto de consulta q (por ejemplo $[q]$). Entonces, usando la nueva función de distancia D la consulta es transformada en $([q], r)_D$. Como el mapeo es contractivo, tenemos $(q, r)_d \subseteq ([q], r)_D$. Esto es, $([q], r)_D$ es en efecto una lista candidata, así esto es suficiente para desarrollar una búsqueda exhaustiva en la lista candidata (ahora usando la distancia original), para obtener la salida real de la consulta $(q, r)_d$.

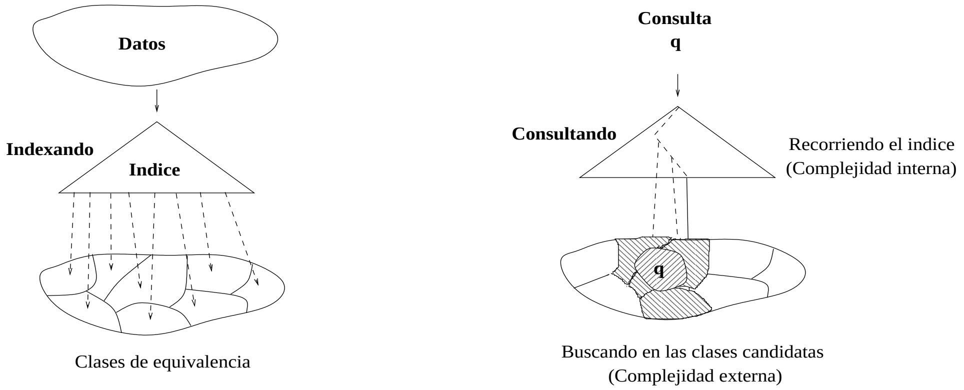


Figura 3.3: Modelo usado para indexar y consultar en espacio métricos

El procedimiento antes citado es de hecho usado en cada algoritmo de indexamiento. En otras palabras.

Todos los algoritmos de indexamiento para búsqueda de proximidad consisten en construir un conjunto de clases de equivalencia, descartando algunas clases, y buscando exhaustivamente en el resto.

En la figura 3.3 podemos ver esquemáticamente un ejemplo de esta idea. Dividimos el espacio en varias regiones (clases de equivalencia). Los puntos dentro de cada región son indistinguibles. Podemos considerarlos como puntos en un nuevo espacio métrico. Para encontrar la respuesta, en lugar de examinar exhaustivamente el diccionario completo, solo examinamos las clases que contienen puntos potencialmente interesantes, en la imagen del lado derecho en la figura 3.3 estas regiones de interés se muestran sombreadas. En otras palabras, si una clase puede contener un punto que debe ser regresado en la salida de una consulta, entonces la clase será examinada.

3.5 Complejidad interna y externa

Una vez que hemos determinado la relación de equivalencia a usar (por ejemplo los P pivotes), procesamos el conjunto pero guardando para cada elemento de U, sus P coordenadas (es decir la distancia a los P pivotes). Esto toma $O(Pn)$ tiempo de preprocesamiento y espacio en memoria principal. El índice puede ser visto como una tabla de n renglones y P columnas como se muestra en la parte izquierda de la figura 3.4.

En tiempo de consulta, primero comparamos la consulta q contra los P pivotes, entonces obtenemos sus P coordenadas $(y_1, ...y_P)$ en el espacio. Esto corresponde a

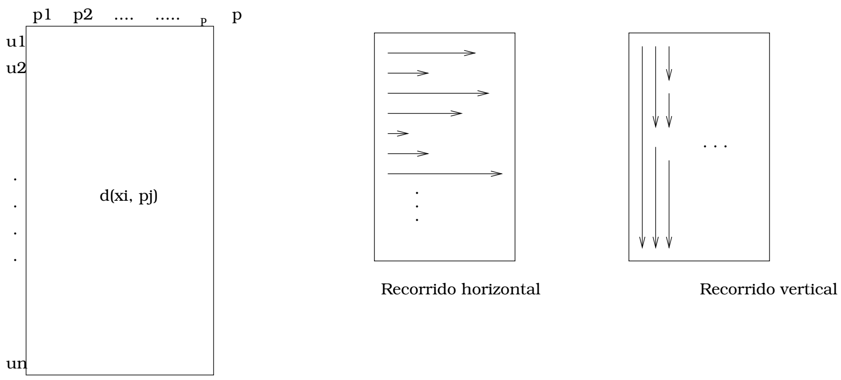


Figura 3.4: Vista esquemática de un índice usando P pivotes, así como un recorrido horizontal y vertical

determinar cuales clases equivalentes pertenecen a la consulta. El costo de estas P evaluaciones de la función de distancia d, corresponden a la *complejidad interna* de la búsqueda. Ahora tenemos que determinar, en el espacio, cuales clases pueden ser relevantes a la consulta (por ejemplo cuales están a distancia r o menos) recorriendo el índice, esto puede tomar un costo extra de CPU. Finalmente, los elementos en las clases calificadas (aquellos que no pudieron ser descartados después de considerar los P pivotes) son directamente comparados contra q (*complejidad externa*). La complejidad interna y externa son usadas para comparar el desempeño de un algoritmo.

3.6 Estado del arte de la construcción del índice

Probablemente la primera solución general para buscar en espacios métricos fue presentada en [BK73], donde se propone un árbol llamado *"Burkhard-Keller Tree"* o ***BKT***, el cual es recomendable para cuando se usan funciones de distancia discretas. Un desarrollo más sobresaliente sobre los BKTs es ***FQT*** *"Fixed Query Tree"* [BYCMW94]. Este árbol es básicamente un BKT donde todas las ramas del mismo nivel, tienen el mismo pivote (por supuesto que no necesariamente pertenecen al conjunto guardado en el subárbol). Los elementos restantes están guardados en las hojas. La ventaja de tal construcción es que las comparaciones entre la consulta y los nodos (pivotes) solo se necesitan una vez y estas ayudan en el recorrido del árbol. Si visitamos muchos nodos del mismo nivel, no se necesita calcular más que una comparación porque todos los pivotes en este nivel son el mismo. En [BYCMW94] los autores proponen una variante llamada ***FQHT*** *"Fixed-Height FQT"*, donde todas las hojas están a la misma profundidad h, independientemente del tamaño del bucket. Esto hace que algunas hojas estén más profundas de lo necesario, lo cual tiene sentido porque podemos haber desarrollado las comparaciones entre la consulta y el pivote de un nivel intermedio, y por lo tanto eliminamos libremente una hoja, sin haber calculado su distancia. En [CMN99] se presenta el ***FQA*** *arreglo de consultas fijas* (Fixed

Query Array), aunque no propiamente un árbol, no es más que una representación compacta de los FQHT. Imagine que un FQHT de altura fija h es construida en un conjunto. Si recorremos las hojas del árbol de izquierda a derecha y ponemos los elementos en un arreglo, el resultado es el FQA. Los elementos del arreglo (renglones) son concatenados en un número que después es ordenado lexicográficamente, cada subárbol del FQHT corresponde a un intervalo en el FQA, y cada movimiento en el FQHT es simulado con dos búsquedas binarias en el FQA. Usando la misma memoria, la simulación del FQA es capaz de usar muchos más pivotes que el original FQHT, con lo cual prueba su eficiencia.

La primera técnica diseñada para funciones de distancia continuas fue llamada "árboles métricos" en [Uhl91]. Un trabajo más completo con la misma idea [Yia93] son ***VPT***s *"Vantage Point Tree"*, en el cual se construye un árbol binario recursivamente, tomando algún elemento como la raíz p y tomando la mediana del conjunto de todas las distancia, $M = mediana\{d(p,u) \mid u \in U\}$. Los elementos u tales que $d(p,u) < M$ son insertados en el subárbol izquierdo, mientras que aquellos tales que $d(p,u) > M$ son insertados en el subárbol derecho. Los VPT pueden ser extendidos a árboles $m-arios$ usando los $m-1$ percentiles uniformes en lugar de solo la mediana, esto es sugerido en [Bri95, BO97]. En [BO97] se presentó el *"Multi-Vantage Point Tree* ***MVPT*** el cual proponen el uso de muchos elementos en un nodo. Otra generalización de los VPT está dada por el ***VPF*** (Vantage Point Forest) [Yia93]. Este algoritmo está diseñado para búsqueda del vecino más cercano con un radio limitado (una consulta de tipo (b) con radio máximo r^*). El método consiste en excluir, a cada nivel los elementos a distancia intermedia a sus pivotes. Si r_o y r_n corresponden a los puntos más cercanos y más lejanos al pivote p, los elementos $u \in U$ tales que $d(p,r_o)+\delta \leq d(p,u) \leq d(p,r_n)-\delta$ son excluidos del árbol. Un segundo árbol es construido con la parte excluida del primer árbol, y así se continúa para obtener un bosque.

Otro algoritmo presentado en [Uhl91] es ***GHT*** *árbol de generalización hiperplana* (Generalize Hiperplane-Tree) el cual es un árbol binario construido recursivamente como sigue. Para cada nodo, dos puntos p_1 y p_2 son seleccionados. Los puntos más cercanos a p_1 que a p_2 van dentro del subárbol izquierdo y aquellos más cercanos a p_2 van en el subárbol derecho. El GHT es extendido en [Bri95] a un árbol $m-ario$, llamado **GNAT** *"Geometric Near-Neighbor Access Tree"*. Se selecciona, para el primer nivel P pivotes $p_1, ..., p_P$, y se define $U_i = \{u \in U, d(p_i,u) < d(p_j,u), \forall j \neq i\}$. Esto es, U_i son los puntos más cercanos a p_i que a algún otro p_j. Finalmente dan algún criterio para seleccionar los p_i elementos los suficientemente lejanos.

Un algoritmo el cual encierra muchas de las ideas presentadas pero se desarrolla sorpresivamente mucho mejor por un orden de magnitud es **AESA** [Vid86] ("Aproximating Eliminating Search Algorithm"). La estructura no es un árbol, pero la idea es muy similar, se selecciona un punto $p \in U$ aleatorio y miden $r_p = d(p,q)$, eliminando todos los elementos $u \in U$ los cuales no satisfacen $r_p - r \leq d(u,p) \leq r_p + r$. Esto se repite hasta que hay pocos elementos resultantes en el conjunto y son comparados contra q directamente.

3.7 Estado del arte para el problema de los K vecinos más cercanos

Pravin M. Vaidya [VAnl89]. Este algoritmo resuelve el problema de los K vecinos más cercanos. Es un algoritmo para espacios vectoriales el cual tiene una complejidad de $O(\log n)$, y es muy sensible a la dimensión.

Kenneth L. Clarkson [Cla83]. Es un algoritmo para espacios métricos que soluciona el problema de todos los vecinos más cercanos en $O(n \log \sigma)$ tiempo, donde σ es el radio de la distancia entre el par de puntos más lejanos a la distancia entre el par de puntos más cercanos. Este algoritmo construye un *celltree*, una compresión de un *trie* digital, en $O(n \log n)$ tiempo probabilístico, finalmente su complejidad es $O(n(\log n)^2(\log \eta)^2))$ donde η es la constante de empacamiento.

Capítulo 4

FQA: Fixed Query Array

El FQA [CMN99] es índice ó estructura de datos simple, con el cual se facilitan las consultas, por reducir el número de cálculos de distancia. Entre sus características está el que ocupa muy poca memoria y permite negociar el número de pivotes para la precisión y de aquí optimizar la cantidad de memoria que puede ser usada.

4.1 La estructura del FQA

Para empezar la descripción del FQA, primero asumiremos que las distancias son valores discretos. Para cada elemento de la base de datos, se guarda una lista de las distancias a los P pivotes. En el FQA, esta lista es considerada como una secuencia de P enteros. La estructura simplemente guarda los elementos de la base de datos ordenados lexicográficamente (obsérvese la figura 4.2, tabla "a") por esta secuencia de distancias, esto es, los elementos son primero ordenados por su distancia al primer pivote (columna 1 de la tabla "a", figura 4.2), aquellos que tienen la misma distancia al primer pivote son ordenados por su distancia al segundo pivote (columna 2 de la tabla "a", figura 4.2)y así. A medida que más y más pivotes son agregadas el arreglo está y más ordenado.

El resultado del FQA tiene una fuerte relación al FHFQT (Fixed Height FQT, veáse la figura 4.1) de altura P. Si las hojas del FHFQT son recorridas en pre-orden, la salida es precisamente el orden impuesto en el FQA, más aún el algoritmo de búsqueda del FHFQT es inherente al FQA. Cada nodo del FHFQT corresponde a un rango de celdas en el FQA. Si un nodo desciende desde otro en el árbol, este rango es un subrango de otros en el arreglo. De aquí que, para cada vez, que el algoritmo del árbol se mueve desde un nodo a uno de sus hijos en el árbol, simulamos el movimiento en el arreglo, por búsqueda binaria al nuevo rango dentro del actual. Esta búsqueda binaria no desarrolla cálculos de distancia extra, solo compara secuencias de enteros.

La complejidad de construcción es $O(Pn)$ cálculos de distancia más el tiempo para ordenar el arreglo lexicográficamente. Esto es $O(k \log n)$ [CMN99].

Para hacer una idea más clara, mostramos explícitamente el algoritmo de búsqueda. Dada una consulta q a ser buscada con tolerancia r y P pivotes $p_1 ... p_P$, medimos $d_1 = (q, p_1), d_2 = (q, p_2), d_n = (q, p_n)$. Ahora, para cada i en el rango $d_i - r$ a $d_i + r$, hacemos una búsqueda binaria en el arreglo del rango donde la primer coordenada es i. Una vez que el rango es calculado, para cada i continuamos recursivamente la búsqueda en el subarreglo encontrado, desde el pivote p_2. Esto es equivalente a recursivamente visitar en el i^{esimo} nivel del subárbol en el FHFQT. La búsqueda termina cuando usamos los P pivotes, y los puntos del subarreglo resultante son revisados secuencialmente. El procedimiento recursivo obviamente termina antes cuando el subarreglo resultante esta vacío.

4.2 Un ejemplo del FQA

En esta sección se presentará un ejemplo del proceso de búsqueda en el FQA. Considere el FHFQT de la figura 4.1. Cada ramificación de la raíz representa una distancia al pivote p_1. Las ramificaciones de los nodos del segundo nivel se refieren a la distancia a p_2 y así sucesivamente.

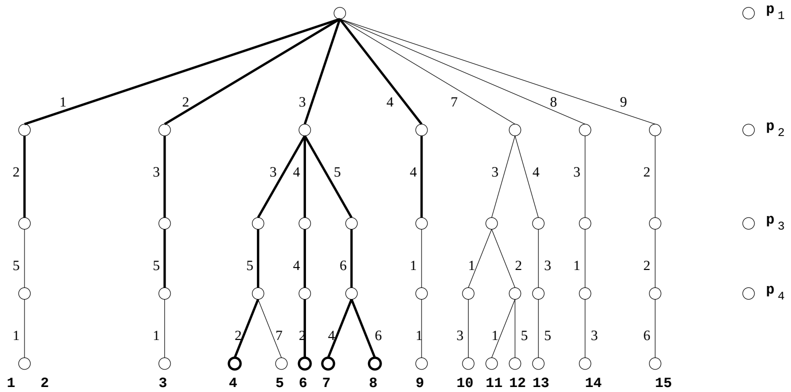


Figura 4.1: Representación de la búsqueda en FHFQT (Fixed Height FQT)

El equivalente FQA a partir del FHFQT de la figura 4.1 guarda los elementos en el orden de izquierda a derecha, guardando las cuatro distancias para cada elemento, es decir, para llegar al elemento 1 debemos recorrer la rama de los elementos a distancia 1 y descender por esa rama, hasta las hojas, con lo cual se obtiene $\{1\ 2\ 5\ 1\}$ (la primer línea de la tabla (a) de la figura 4.2. La figura 4.2 en la tabla (a) ilustra este proceso, cabe mencionar que las tablas (b), (c), (d) representan los pasos seguidos para una búsqueda en cada nivel del FHFQT, es decir, la tabla (b) representa con

(1,5)

	(	a	)
1	**2**	**5**	**1**
1	**2**	**5**	**1**
2	**3**	**5**	**1**
3	**3**	**5**	**2**
3	**3**	**5**	**7**
3	**4**	**4**	**2**
3	**5**	**6**	**4**
3	**5**	**6**	**6**
4	**4**	**1**	**1**
7	3	1	3
7	3	2	1
7	3	2	5
7	4	3	5
8	3	1	3
9	2	2	6

(2,6)

	(	b	)
1	**2**	**5**	**1**
1	**2**	**5**	**1**
2	**3**	**5**	**1**
3	**3**	**5**	**2**
3	**3**	**5**	**7**
3	**4**	**4**	**2**
3	**5**	**6**	**4**
3	**5**	**6**	**6**
4	**4**	**1**	**1**
7	3	1	3
7	3	2	1
7	3	2	5
7	4	3	5
8	3	1	3
9	2	2	6

(3,7)

	(	c	)
1	**2**	**5**	**1**
1	**2**	**5**	**1**
2	**3**	**5**	**1**
3	**3**	**5**	**2**
3	**3**	**5**	**7**
3	**4**	**4**	**2**
3	**5**	**6**	**4**
3	**5**	**6**	**6**
4	**4**	1	1
7	3	1	3
7	3	2	1
7	3	2	5
7	4	3	5
8	3	1	3
9	2	2	6

(2,6)

	(	d	)
1	**2**	**5**	1
1	**2**	**5**	1
2	**3**	**5**	1
3	**3**	**5**	**2**
3	**3**	**5**	7
3	**4**	**4**	**2**
3	**5**	**6**	**4**
3	**5**	**6**	**6**
4	**4**	1	1
7	3	1	3
7	3	2	1
7	3	2	5
7	4	3	5
8	3	1	3
9	2	2	6

Figura 4.2: Representación de la búsqueda en FQA (Fixed Query Array).

negritas las ramas que serán consideradas en el siguiente paso de la búsqueda. Tenemos cuatro pivotes, y cada renglón en las cuatro tablas (a), (b), (c) y (d) representa una rama en el árbol; con la distancia de los puntos de la base de datos al pivote apropiado. Para una consulta q calculamos el vector $(d(q, p_1)...d(q, p_4))$, es decir, calculamos $d(q, p_i)$, en este caso este cálculo obtiene $(3, 4, 5, 4)$, si radio de búsqueda es 2, tenemos que los intervalos de búsqueda son $(\{1, 5\}, \{2, 6\}, \{3, 7\}, \{2, 6\})$ en cada nivel del árbol respectivamente, Las ramificaciones etiquetadas como $[1, 2, 3, 4]$ en el primer nivel van a ser examinadas y, recursivamente, todas las ramificaciones con búsqueda binaria se encuentran los intervalos en la primera columna, en cada una de las cuatro tablas, mostramos a los candidatos con negritas para cada paso. La tabla (a) es equivalente al primer nivel en el 7 árbol, la tabla (b) para el segundo nivel, y así para el resto. Podemos fácilmente verificar que los intervalos para la búsqueda binaria en cada columna es equivalente a limitar la búsqueda en los niveles apropiados en el árbol.

Debe observarse que el orden lexicográfico nos permite usar la búsqueda binaria en las columnas subsecuentes. Considerando por ejemplo las columnas que empiezan con un 3: los elementos de la segunda columna son también ordenados en orden creciente y así, la figura 4.2 muestra este proceso de búsqueda.

Es claro que la lista de candidatos usando cualquier representación no cambia. En la búsqueda basada en el arreglo tenemos que pagar $O(log\ n)$ (el costo de la búsqueda binaria) para simular la visita a cada ramificación. Si visitamos m nodos en el árbol, usamos $O(m \log n)$.

Capítulo 5

FQTrie: Fixed Query Trie

5.1 Introducción

Una de las contribuciones de este trabajo es el algoritmo llamado FQTrie (Fixed Query Trie) el cual es una estructura de datos basada en pivotes que servirá para indexar la base de datos. El FQTrie es un árbol con profundidades variables y guarda solo una parte de la información de cada elemento. Esta estructura servirá para satisfacer el problema de todos los K vecinos más cercanos. Cabe mencionar que este algoritmo es una modificación al FQA (Fixed Query Array).

5.2 La estructura del FQTrie

Inicialmente se tiene una base de datos con n elementos, de ahí se toman P elementos aleatorios como pivotes, se calculan las distancias de todos los elementos a los pivotes, y con esta información se construye la firma de cada elemento, una vez que se tienen todas las firmas se ordenan lexicográficamente como en el FQA, y finalmente se construye un árbol usando estas firmas, cabe mencionar que cuando se tiene solo un elemento en una rama del árbol, en ese momento esa rama es terminada y se coloca su hoja con la información correspondiente a ese elemento (la cual es solo una parte de la firma), esto es básicamente la idea de la estructura del FQTrie, más adelante se describe cada paso de manera detallada.

El resultado de este algoritmo tiene una fuerte relación con el FQA y con el FHFQT, a diferencia de estos algoritmos, el FQTrie no es un arreglo completamente, ni tiene profundidad fija en cada hoja.

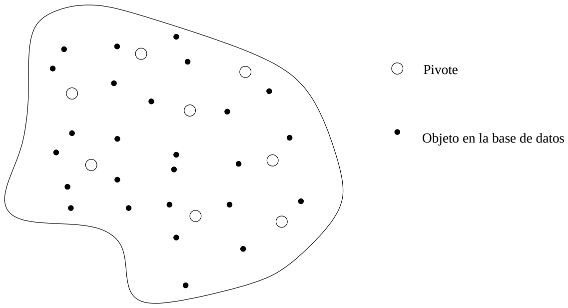


Figura 5.1: Base de datos marcando la selección de los pivotes

5.2.1 Indexamiento

En esta sección se explicará como se realiza el indexamiento de la base de datos, es decir, la construcción del FQTrie; el modelo que se utilizó fue basado en el FQA y el modelo de un árbol, obteniendo una estructura la cual es una parte árbol y una parte arreglo. La implementación de este algoritmo es el siguiente:

Dado una base de datos con objetos (aleatorios) los cuales pueden ser palabras, imágenes, etc.; para construir la estructura del FQTrie se llevan acabo los siguientes pasos:

- ***Selección de pivotes***.- Los primeros N objetos de la base de datos (aleatorios) los llamaremos pivotes y servirán para construir las firmas necesarias para la construcción del árbol. Véase la figura 5.1.

- ***Cálculos de distancia a los pivotes***.- Una vez que se seleccionaron los pivotes, se realiza el cálculo de distancia (en el apéndice A.1 se presenta un ejemplo de lo que podría ser un cálculo de distancia entre palabras) de cada uno de todos los pivotes a los elementos de la base de datos incluyendo a los pivotes restantes, para ilustrar este paso obsérvese la figura siguiente 5.2. Cada una de las distancias calculadas es almacenada (para cada pivote) conservando el número del elemento con el cual fue medido.

- ***Generación de firmas***.- Una vez que se tienen todos los cálculos de distancia de cada pivote, se ordenan estas distancias (para cada pivote), y se establecen 2^{bits} divisiones, lo que representa a generar 2^{bits} anillos en torno al pivote, obsérvese las figuras 5.3, 5.4, 5.5, 5.6, cada uno de estos anillos tiene una etiqueta asignada donde cada una de estas nuevas etiquetas (00, 01, 10,....) corresponden a los bits (el número de bits es uno de los parámetros dados por el usuario, por ejemplo, para la etiqueta 00 el parámetro fue 2 bits) que se escribirán en un byte para cada elemento. Cada elemento tiene un número de bytes destinado para almacenar la firma, el cual depende del número de pivotes

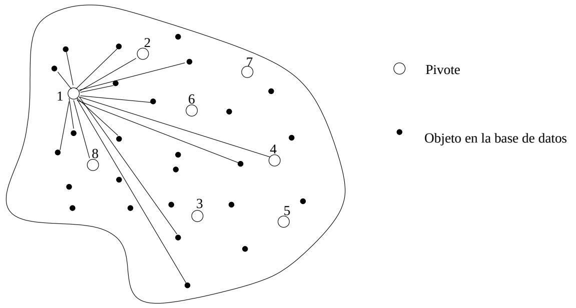


Figura 5.2: Cálculo de distancia de un pivote a toda la Base de Datos (por simplicidad no se muestran todas la líneas)

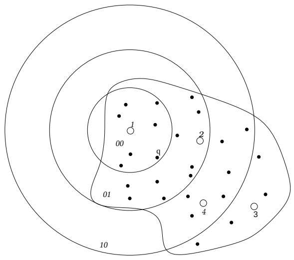


Figura 5.3: Construcción de la firma. En la figura se muestran los anillos generados para el pivote 1

y el número de bits. El diagrama de la firma para el punto q de la figura 5.3 se muestra en la tabla 5.1 (usando 4 pivotes y 2 bits), donde los dos primeros *bits* corresponden al anillo en el que se encuentra este elemento del pivote 1, los dos siguientes bits al pivote 2 y así sucesivamente, si el número de bits por el número de pivotes es mayor que 8, serán necesarios más de un byte para la firma (no aplica para el caso de ejemplo). Finalmente cuando se tienen ya estas firmas para cada elemento, se ordenan lexicográficamente y esto será utilizado en el siguiente paso.

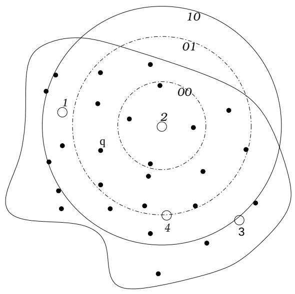


Figura 5.4: Construcción de la firma. En la figura se muestran los anillos generados para el pivote 2

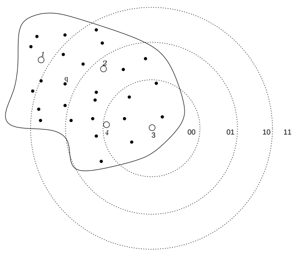


Figura 5.5: Construcción de la firma. En la figura se muestran los anillos generados para el pivote 3

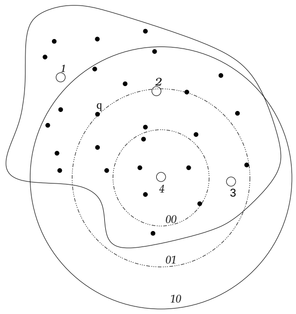


Figura 5.6: Construcción de la firma. En la figura se muestran los anillos generados para el pivote 4

01	01	10	10
pivote 1	pivote 2	pivote 3	pivote 4

Tabla 5.1: Esquema del uso de un byte para almacenar una firma usando 4 pivotes y 2 bits

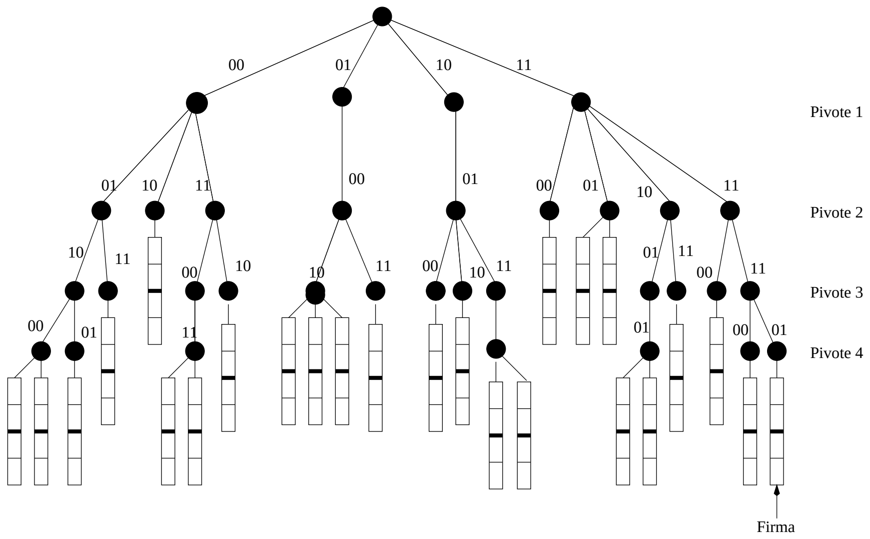


Figura 5.7: Representación gráfica de un FQTrie (Fixed Query Trie).

- ***Construcción del árbol***.- Una vez que se tienen las firmas de todos los elementos y que éstas están ordenadas, la construcción del árbol se realiza de la siguiente manera. Para el primer nivel del árbol se toma en cuenta solo el pedazo de la firma correspondiente al primer pivote de todos los elementos (en la tabla 5.1 corresponde a los dos primeros bits), es decir, todos los elementos que estén en el mismo anillo de un pivote estarán en la misma rama, para cada nivel del árbol se sigue este mismo procedimiento usando solo los siguientes pivotes, es decir, para el segundo nivel se usa el pivote 2 y así sucesivamente. Una vez que en alguna de las ramas, en cualquier nivel del árbol, solo queda un elemento, en esta rama se crea un arreglo con el resto de la firma de ese elemento. Para ilustrar esto obsérvese el ejemplo de la figura 5.7, una vez que se tiene construido el árbol, ya se pueden realizar consultas de la siguiente manera.

- ***Recorrido en el árbol***.- Dada una consulta q, antes de hacer el recorrido a través del árbol se calcula la firma de q como se hizo para cada elemento de la base de datos. Inicialmente se tiene un radio r, así que se visitan solo aquellas ramas que cumplen con $d(q,p_i) \pm r$, donde p_i es pivote correspondiente en un nivel del árbol. Una vez que se llega a una hoja se analiza la firma de q contra la firma de la hoja es decir solo se comparan las dos firmas (que finalmente son solo secuencias de bits en un byte) y se determina si la firma del elemento(s) es próspera, de ser así, se calcula q directamente contra ese elemento. Este procedimiento termina una vez que son recorridas todas la ramas potenciales.

Capítulo 6

Encontrando todos los K vecinos más cercanos

Dada un conjunto de puntos U, tal que $\mid U \mid = n$, en un espacio métrico, se desea conocer para cada elemento ($\{u \in U\}$) sus K vecinos más cercanos. La complejidad del problema se mide en cálculos de distancia; se sabe que usando la fuerza bruta el problema es resuelto en n^2 cálculos de distancia.

En esta sección se planteará el desarrollo del algoritmo, el cual consiste en indexar la base de datos en tiempo lineal, después, hacer n consultas de rango con un radio seguro para conocer los K vecinos más cercanos, donde un radio seguro es aquel que contiene al menos K vecinos más cercanos; esta heurística se puede mejorar con una estructura auxiliar llamada *heap* ó cola de prioridad, en la cual se mantienen al menos K elementos más cercanos para cada punto durante el desarrollo del algoritmo, con esta estructura se pretende reducir los radios de cada punto cada vez que se pueda.

6.1 Desarrollo del algoritmo

Introducimos una técnica que acelera el proceso para encontrar los K vecinos más cercanos de *todos* los elementos de una base de datos, la idea es indexar este conjunto de puntos con un algoritmo de pivotes como el FQTrie conservando las ventajas del FQA y mejorando algunas de ellas, además la complejidad de este algoritmo es lineal en su construcción, posteriormente usar está para encontrar los K vecinos más cercanos de cada punto.

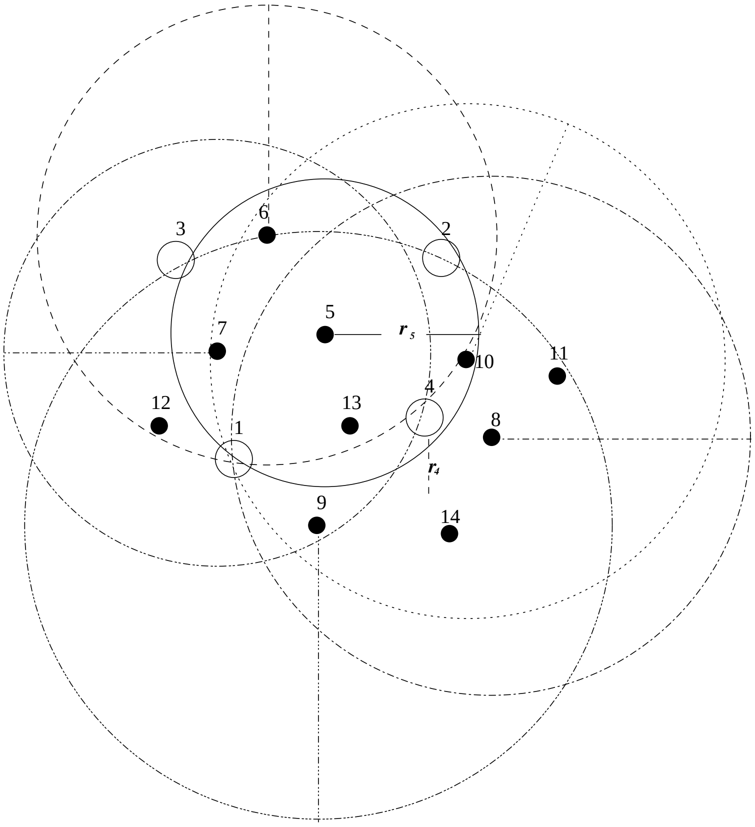
La idea principal del algoritmo es, realizar una consulta de rango con un radio r, para cada elemento, es decir, se hacen n consultas de rango con radio r, es importante enfatizar que para cada punto se utilizan los mismos pivotes usados para generar el índice.

La estructura auxiliar (cola de prioridad) usada para el desarrollo del algoritmo consiste en mantener para cada punto al menos los K elementos más cercanos durante el proceso, manteniendo un radio máximo, el cual corresponde al elemento más alejado y es precisamente el que esta en el tope de la cola de prioridad. Como primer paso la cola de prioridad es inicializada usando los pivotes (puesto que estos puntos conocen todas las distancias a los demás puntos, véase la figura 5.2) y cada una de las colas de prioridad para los puntos que no son pivotes son rellenadas con los K pivotes más cercanos véase esto en la figura 6.2, con esto se asegura que los K vecinos más cercanos estén dentro del radio inicial, el cual está acotado por los pivotes, véase la figura 6.1

A continuación se presentan de manera detallada las ideas mencionadas anteriormente.

- ***Inicialización de la cola de prioridad***.- En esta primera etapa, cada elemento de la base de datos, no cuenta con ningún elemento en su cola de prioridad, y su radio de búsqueda inicial es infinito, así que se toman todos los pivotes con los que se generó el FQTrie y se inicializan las colas de prioridad de todos los elementos, debido a que estos conocen las distancias a todos los elementos de la base de datos, con esto se reduce el radio máximo en la cola de prioridad, es decir, de un radio infinito a un radio acotado por los pivotes, inicialmente se tienen al menos K posibles elementos a ser los K vecinos más cercanos en la cola de prioridad, es decir, todos los puntos que nos pivotes, en su cola de prioridad tienen a sus K pivotes más cercanos (esto es solo inicialmente). Esta etapa puede verse en la figura 6.1.

- ***Consulta de rango para cada punto***.- Una vez que se cuenta con un radio inicial en la cola de prioridad de cada elemento, se inicia la construcción de los K vecinos más cercanos. Primero, se toma un punto a la vez, y se hace una consulta de rango sobre el índice construido con este elemento y su radio máximo en la cola de prioridad, usando los pivotes con los que se construyó el índice; esto se hace así para cada elemento de la base de datos, es importante enfatizar que a medida que una consulta de rango es aplicada a un elemento, todos los cálculos de distancia realizados en esta consulta, sirven para actualizar el radio máximo de las otras colas de prioridad, no solo la del elemento actual. Cada vez que se encuentra un elemento que mejora la distancia máxima (es decir, es una distancia más pequeña) de alguna cola de prioridad, se saca el elemento en el tope de la cola de prioridad y se inserta el elemento encontrado, con esto obviamente se reduce el radio máximo de la cola de prioridad. El



4 pivots
K = 3

	radius	heap		
1	$\boldsymbol{r_1}$	7	9	12
2	$\boldsymbol{r_2}$	4	5	10
3	$\boldsymbol{r_3}$	5	7	6
4	$\boldsymbol{r_4}$	13	10	8
→ 5	r_5	1	2	4
6	r_6	1	2	3
7	r_7	4	1	3
8	r_8	1	2	4
9	r_9	2	4	1
10	r_{10}	1	2	4
11	r_{11}	1	2	4
12	r_{12}	4	3	1
13	r_{13}	2	1	4
14	r_{14}	2	1	4

$r_5 = d(5,1)$

Figura 6.1: Cola de prioridad inicializada. En la figura se muestran algunos puntos (parte izquierda) con sus radios máximos en la cola de prioridad (parte derecha), con círculos entorno al elemento.

algoritmo termina cuando a todos los elementos de la base de datos se les aplico esta consulta, y como resultado todos los elementos ya tienen sus K vecinos más cercanos.

Es importante enfatizar que, debido a las características del algoritmo, este puede ser detenido en cualquier momento y se tendría una muy buena aproximación de todos los K vecinos más cercanos, puesto que se mantienen siempre al menos K elementos más cercanos durante el desarrollo del algoritmo.

6.2 Ejemplo

En esta sección presentaremos el método que se sigue para encontrar los K vecinos más cercanos, paso a paso, esto se ilustrará con una serie de imágenes presentadas en esta sección.

Los 14 puntos mostrados en la figura 6.2 corresponden a una base de datos, donde los primeros 4 puntos (círculos no rellenos del 1 al 4), corresponden a los puntos seleccionados como pivotes (cabe mencionar que estos puntos son escogidos al azar), los 10 puntos restantes de la imagen (círculos rellenos del número 5 al 14) corresponden a los puntos de la base de datos que no son pivotes, también se puede apreciar en el lado derecho de esta imagen, que cada punto cuenta con una cola de prioridad (su descripción puede verse en el apéndice B.1), donde se almacenarán los K (para este ejemplo K=3) vecinos más cercanos y la distancia máxima a la que se encuentra el elemento más lejano con respecto a ese punto, esta cola de prioridad esta ordenada de acuerdo a la distancia, por lo tanto el punto más a la izquierda en la cola de prioridad (heap) de cada uno, en la imagen corresponde al punto más alejado, caso contrario para el punto que se encuentra en el lado derecho.

En la figura 6.1 se pueden observar que los puntos que no son pivotes *inicialmente* contienen en su cola de prioridad los K *pivotes* más cercanos al punto en cuestión, la explicación a este criterio fue explicada anteriormente, nótese que para este cálculo no es necesario hacer ningún cálculo de distancia extra. En la figura 6.2 se puede observar los puntos 1, 2, 3 y 4 (pivotes), tiene a sus K vecinos más cercanos en su cola de prioridad, debido a que por ser pivotes, estos puntos saben exactamente a que distancia están todos los elementos de cada uno de ellos, y el radio en la cola de prioridad (por ejemplo r_1 es la distancia al punto 7) corresponde al radio del punto más alejado; finalmente el circulo (línea punteada) que se muestra en esta figura en torno a cada elemento, tiene como radio, el valor máximo del radio en la cola de prioridad.

Como se puede apreciar en la figura 6.3 el circulo formado cuyo centro es el punto

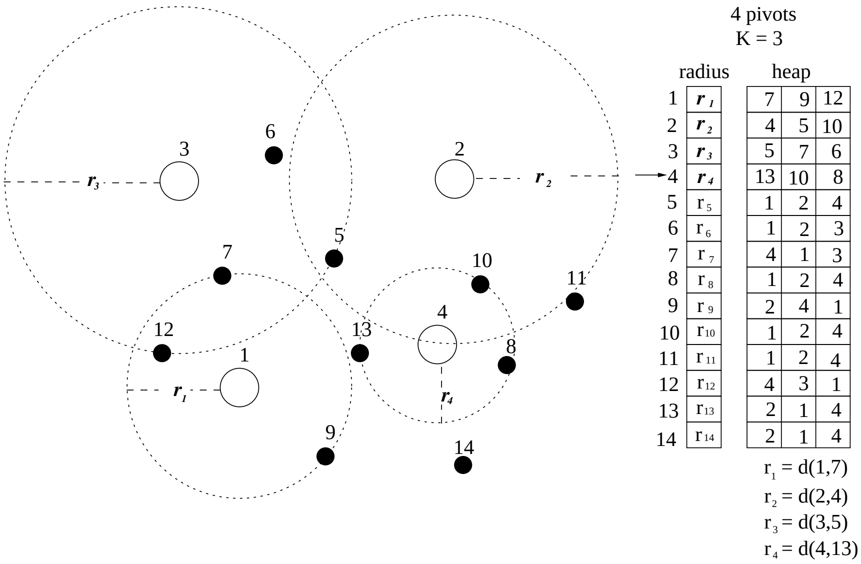


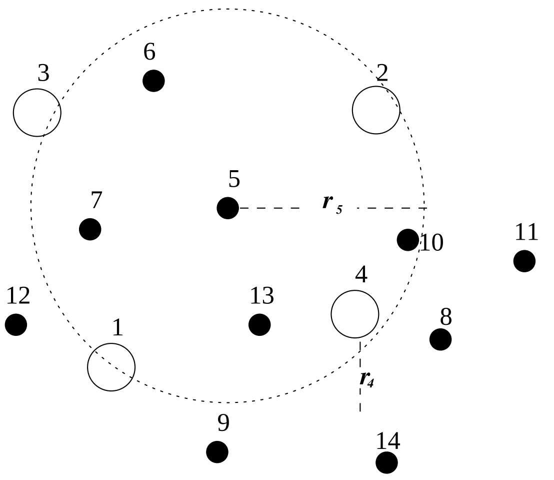
Figura 6.2: Ejemplo de la búsqueda de los K vecinos más cercanos. Puntos iniciales

5, tiene como radio máximo la distancia al punto 1, obsérvese que los elementos que se encuentran en la cola de prioridad corresponden (por estar inicializado así la cola de prioridad) solo a los pivotes, no a los K vecinos más cercanos. Resulta obvio que si los elementos *iniciales* en la cola de prioridad son aquellos K pivotes más cercanos, los K vecinos más cercanos es seguro que se encuentren dentro del *radio inicial máximo*, esto es sin realizar ningún cálculo de distancia extra.

En la siguiente figura 6.4 se muestra el resultado de aplicar una consulta de rango al mismo punto que en la imagen 6.3 (punto número 5), el cual ya tiene en su cola de prioridad sus K vecinos más cercanos, como se puede observar el radio máximo fue modificado (r_{5a}) y ahora ese punto contiene ya a todos sus K vecinos más cercanos, algo que es importante enfatizar, es que no solo fue modificado el radio del punto 5, sino también el radio de los puntos 6, 7, y 13 debido a que todos los cálculos de distancia realizados son aprovechados al máximo, es decir, si un cálculo de distancia realizado mejora el radio máximo de algún otro punto, la cola de prioridad de ese otro punto es actualizado, como es el caso de los puntos 6, 7 y 13.

La siguiente figura 6.5 muestra el radio máximo inicial (circulo mas grande) y el radio máximo final (circulo más pequeño) del punto 6, así como también sus K vecinos más cercanos, como puede observarse solo se realizó un cálculo de distancia para conocer a todos sus K vecinos más cercanos y la única cola de prioridad actualizada fue la de este punto.

En la figura 6.6, se puede observar que el elemento tratado es el número 7, y se realizaron 2 cálculos de distancia extras para conocer los K vecinos más cercanos de este punto, nótese que la cola de prioridad del otro elemento (12) también fue modificada por haber realizado un cálculo de distancia que lo involucraba.

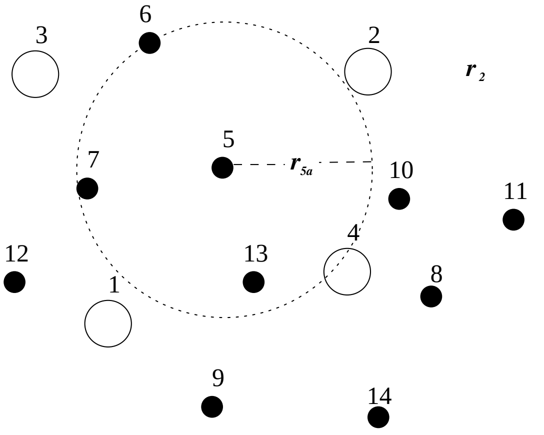


4 pivots
K = 3

	radius	heap		
1	$\boldsymbol{r_1}$	7	9	12
2	$\boldsymbol{r_2}$	4	5	10
3	$\boldsymbol{r_3}$	5	7	6
4	$\boldsymbol{r_4}$	13	10	8
→ 5	r_5	1	2	4
6	r_6	1	2	3
7	r_7	4	1	3
8	r_8	1	2	4
9	r_9	2	4	1
10	r_{10}	1	2	4
11	r_{11}	1	2	4
12	r_{12}	4	3	1
13	r_{13}	2	1	4
14	r_{14}	2	1	4

$r_5 = d(5,1)$

Figura 6.3: Ejemplo de la búsqueda de los K vecinos más cercanos. Estado inicial del punto 5

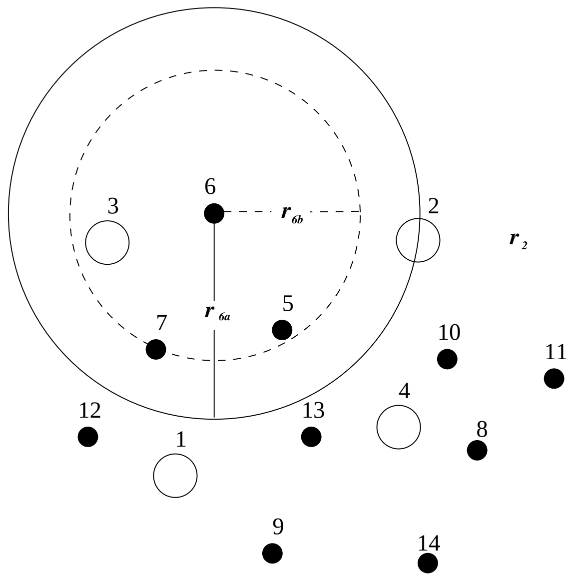


4 pivots
K = 3

	radius	heap		
1	$\boldsymbol{r_1}$	7	9	12
2	$\boldsymbol{r_2}$	4	5	10
3	$\boldsymbol{r_3}$	5	7	6
4	$\boldsymbol{r_4}$	13	10	8
→ 5	r_{5a}	6	7	13
6	r_{6a}	2	5	3
7	r_{7a}	5	1	3
8	r_8	1	2	4
9	r_9	2	4	1
10	r_{10}	1	2	4
11	r_{11}	1	2	4
12	r_{12}	4	3	1
13	r_{13a}	1	5	4
14	r_{14}	2	1	4

$r_{5a} = d(5,6)$
$r_{6a} = d(6,2)$
$r_{7a} = d(7,5)$
$r_{13a} = d(13,1)$

Figura 6.4: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 5

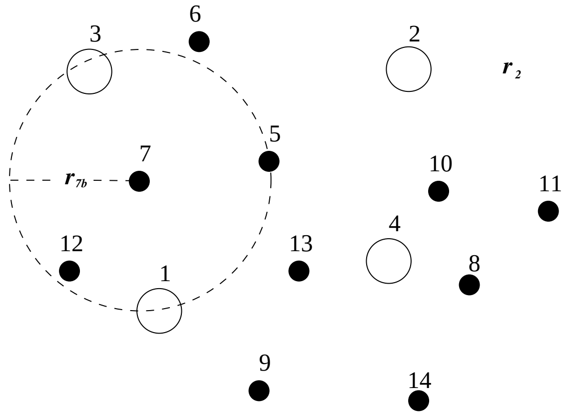


4 pivots
K = 3

	radius	heap		
1	r_1	7	9	12
2	r_2	4	5	10
3	r_3	5	7	6
4	r_4	13	10	8
5	r_{5a}	6	7	13
6	r_{6b}	7	5	3
7	r_{7a}	5	1	3
8	r_8	1	2	4
9	r_9	2	4	1
10	r_{10}	1	2	4
11	r_{11}	1	2	4
12	r_{12}	4	3	1
13	r_{13a}	1	5	4
14	r_{14}	2	1	4

$r_{6b} = d(6,7)$

Figura 6.5: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 6. Radio inicial (círculo exterior), radio final (círculo interior.



4 pivots
K = 3

	radius	heap		
1	r_1	7	9	12
2	r_2	4	5	10
3	r_3	5	7	6
4	r_4	13	10	8
5	r_{5a}	6	7	13
6	r_{6b}	7	5	3
7	r_{7b}	1	3	12
8	r_8	1	2	4
9	r_9	2	4	1
10	r_{10}	1	2	4
11	r_{11}	1	2	4
12	r_{12a}	3	7	1
13	r_{13a}	1	5	4
14	r_{14}	2	1	4

$r_{7b} = d(7,1)$
$r_{12a} = d(12,3)$

Figura 6.6: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 7

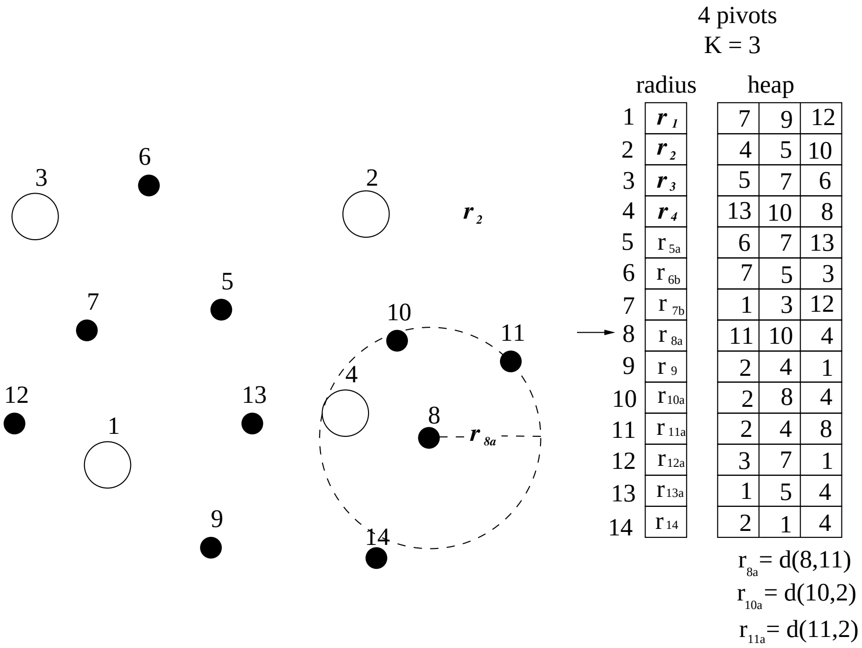


Figura 6.7: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 8

Para algunos puntos es necesario hacer más cálculos de distancia que para otros, esto es debido a la posición en la que se encuentren con respecto a los pivotes, para el caso del punto número 8, se realizaron tres cálculos de distancia extras, y estos cálculos son usados para modificar las colas de prioridades de los elementos 10 y 11, así finalmente este elemento tiene sus K vecinos más cercanos. Como puede verse en la figura 6.7.

Para el punto número 9 de esta base de datos, se calcularon tres distancias extras para conocer sus K vecinos más cercanos y como en los puntos anteriores, estos cálculos sirvieron para modificar la cola de prioridad de otros puntos, en este caso son los puntos 13 y 14, para ilustrar esto obsérvese la figura 6.8

En el punto 10 mostrado en la figura 6.9 solo fue necesario realizar un cálculo de distancia extra y al igual que en los otros casos, se modifica la cola de prioridad del elemento 11.

Como puede observarse en las figuras 6.10, 6.11, 6.12 no se realizo ningún cálculo de distancia extra para conocer los K vecinos más cercanos de cada punto, debido a que la cola de prioridad de cada uno de estos elementos ya contenía sus K vecinos más cercanos.

Finalmente para el punto 14, solo se tuvo que realizar un cálculo de distancia extra, esto puede obsérvese en la figura 6.13

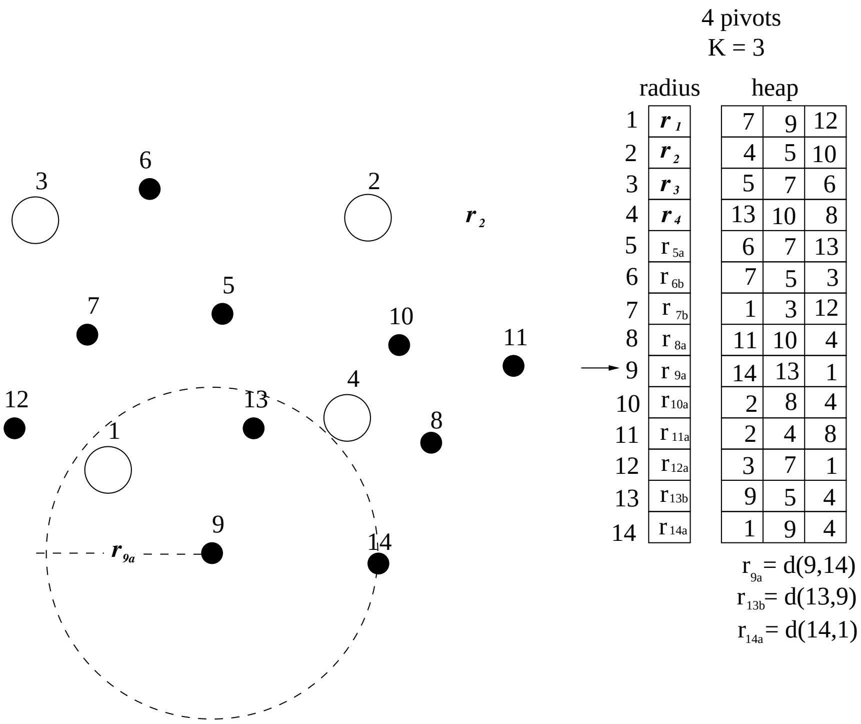


Figura 6.8: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 9

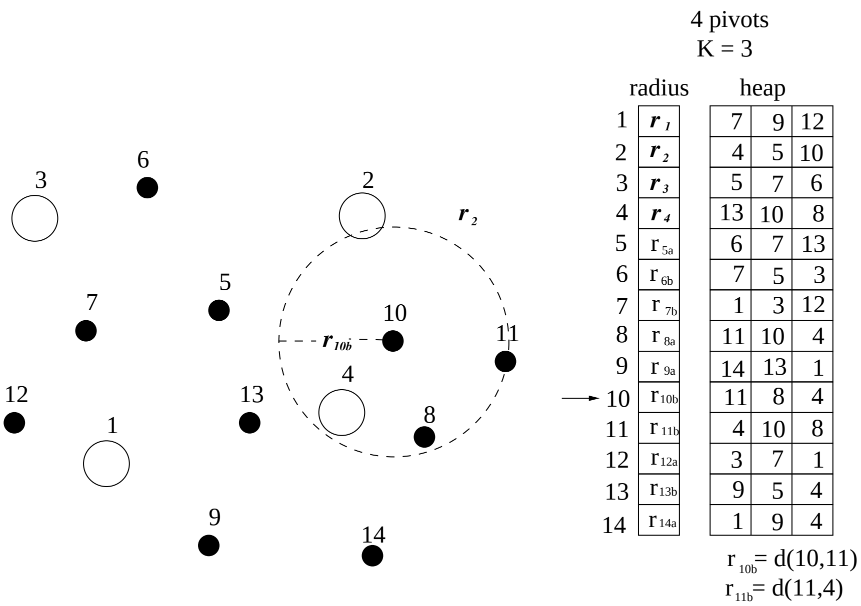
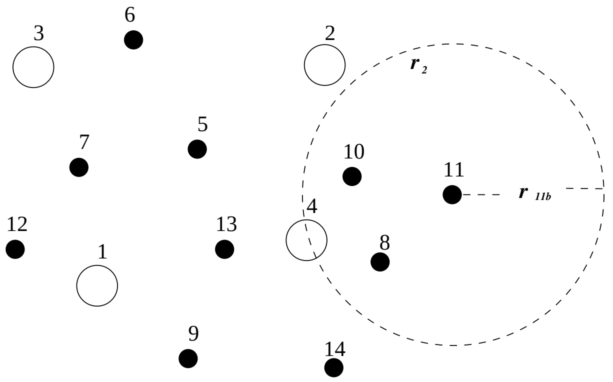


Figura 6.9: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 10

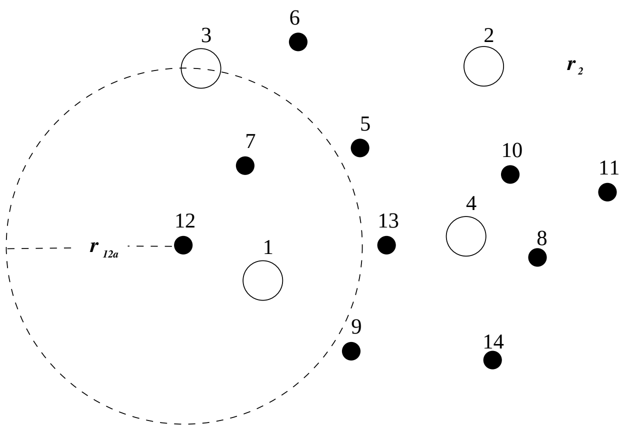


4 pivots
K = 3

	radius	heap		
1	r_1	7	9	12
2	r_2	4	5	10
3	r_3	5	7	6
4	r_4	13	10	8
5	r_{5a}	6	7	13
6	r_{6b}	7	5	3
7	r_{7b}	1	3	12
8	r_{8a}	11	10	4
9	r_{9a}	14	13	1
10	r_{10b}	11	8	4
→ 11	r_{11b}	4	10	8
12	r_{12a}	3	7	1
13	r_{13b}	9	5	4
14	r_{14a}	1	9	4

$r_{11b} = d(11,4)$

Figura 6.10: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 11

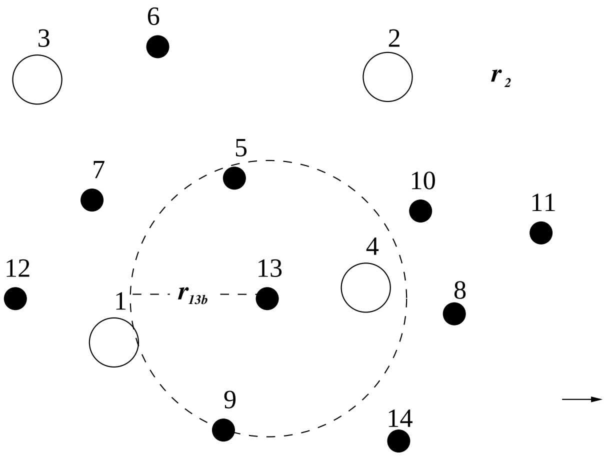


4 pivots
K = 3

	radius	heap		
1	r_1	7	9	12
2	r_2	4	5	10
3	r_3	5	7	6
4	r_4	13	10	8
5	r_{5a}	6	7	13
6	r_{6b}	7	5	3
7	r_{7b}	1	3	12
8	r_{8a}	11	10	4
9	r_{9a}	14	13	1
10	r_{10b}	11	8	4
11	r_{11b}	4	10	8
→ 12	r_{12a}	3	7	1
13	r_{13b}	9	5	4
14	r_{14a}	1	9	4

$r_{12a} = d(12,3)$

Figura 6.11: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 12

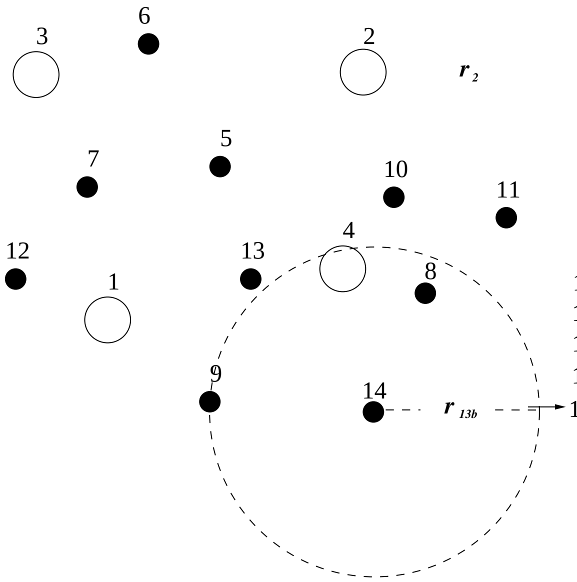


4 pivots
K = 3

	radius	heap		
1	r_1	7	9	12
2	r_2	4	5	10
3	r_3	5	7	6
4	r_4	13	10	8
5	r_{5a}	6	7	13
6	r_{6b}	7	5	3
7	r_{7b}	1	3	12
8	r_{8a}	11	10	4
9	r_{9a}	14	13	1
10	r_{10b}	11	8	4
11	r_{11b}	4	10	8
12	r_{12a}	3	7	1
13	r_{13b}	9	5	4
14	r_{14a}	1	9	4

$r_{13b} = d(13,9)$

Figura 6.12: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 13



4 pivots
K = 3

	radius	heap		
1	r_1	7	9	12
2	r_2	4	5	10
3	r_3	5	7	6
4	r_4	13	10	8
5	r_{5a}	6	7	13
6	r_{6b}	7	5	3
7	r_{7b}	1	3	12
8	r_{8a}	11	10	4
9	r_{9a}	14	13	1
10	r_{10b}	11	8	4
11	r_{11b}	4	10	8
12	r_{12a}	3	7	1
13	r_{13b}	9	5	4
14	r_{14b}	9	4	8

$r_{14b} = d(14,9)$

Figura 6.13: Ejemplo de la búsqueda de los K vecinos más cercanos. Punto número 14

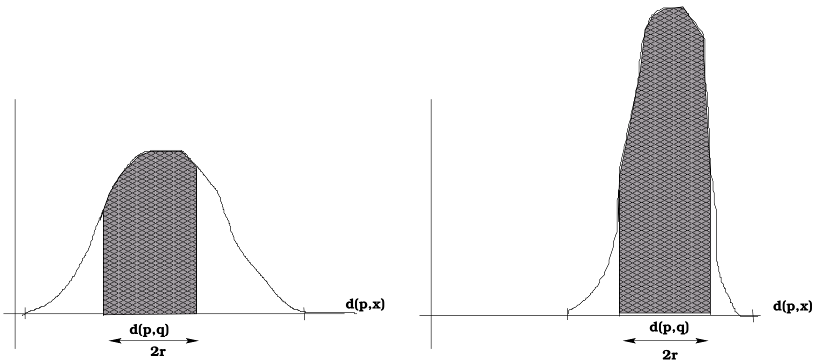


Figura 6.14: Un histograma de distancias con dimensión baja (izquierda) y dimensión alta (derecha), mostramos que en dimensión alta virtualmente todos los elementos son candidatos para una evaluación exhaustiva

6.3 Resultados experimentales

En esta sección se mostrarán los resultados experimentales del desempeño del algoritmo, mediante algunas gráficas representativas.

6.3.1 Desempeño del algoritmo

Todos lo algoritmos de búsqueda tratan de capturar las propiedades geométricas del espacio, las cuales son complejas y costosas de evaluar y algunas son difíciles de extender a espacios métricos [CNBYM99]. Muchos autores han propuesto usar los histogramas de distancias para caracterizar la dificultad de la búsqueda en un espacio métrico arbitrario. Si se considera el histograma de distancias entre puntos en un espacio métrico X, usando un diccionario U, se puede dar una explicación intuitiva del porque el problema de búsqueda es más difícil cuando el histograma esta concentrado (véase la figura 6.14). Si consideramos una consulta aleatoria q y un esquema de indexamiento basado en pivotes, entonces las posibles distancias entre q y un pivote p son distribuidas de acuerdo al histograma de esta figura. La regla de eliminación dice que podemos descartar cualquier punto u tal que $d(p,u) \notin [d(p,q) - r\ ,\ d(p,q) + r]$. El área gris 6.14 en la figura muestra los puntos que *no pueden ser descartados.* Conforme el histograma esta más y más concentrado al punto q, menos y menos puntos pueden ser descartados usando la información dada por $d(p,q)$.

Para observar el desempeño del algoritmo se probó este algoritmo en un *espacio dimensional completo*, variando el número de pivotes, variando el número de vecinos más cercanos (K), y por supuesto variando también el número de elementos en el conjunto de puntos, en las siguientes secciones se presentan los resultados experimentales.

En las gráficas mostradas se nota que hay un cambio de tener un algoritmo subcuadrático a uno sublineal si medimos solo los cálculos de distancia externas (no contando el costo de construcción del índice). Esto es posible por que en algunos casos (muy común para los últimos elementos de la base de datos) podemos resolver las consultas de proximidad casi sin ningún cálculo de distancia, cuando el máximo radio en la cola de prioridad coincide con la lista actual de vecinos más cercanos (K).

En las siguientes secciones se mostrarán los resultados experimentales de este algoritmo, es importante enfatizar que en cada una de las gráficas mostradas a continuación es mucho mejor que usar la fuerza bruta y las gráficas presentadas son comparadas con esta heurística.

6.3.2 Gráficas variando la dimensión

En las gráficas de las figuras 6.15, 6.16 y 6.17 se muestran el número de cálculos de distancia realizados para 250,000, 100,00 y 30,000 puntos respectivamente, y en cada una de estas gráficas se presenta el comportamiento del algoritmo (es decir, el número de cálculos de distancia realizados para un punto, esto es para las gráficas de la 6.15 a la 6.25) variando la dimensión, nótese como a medida que crece la dimensión crece el número de cálculos de distancia realizados; estas gráficas fueron obtenidas usando 32 pivotes, 8 bits para buscar 16 vecinos más cercanos. Como puede observarse para los últimos elementos casi no realiza ningún cálculo de distancia.

Variando el tamaño de la base de datos

Si se varía el número de objetos en una misma dimensión, usando 32 pivotes, 8 bits y 16 vecinos más cercanos, los resultados se muestran en la gráficas de las figuras 6.18, 6.19, 6.20 y 6.21, donde la primer gráfica es en dimensión 8, 10, 14 y 18 respectivamente. En estas gráficas el número de cálculos de distancia se muestra en porcentaje debido a que depende del tamaño del diccionario, el valor máximo de los cálculos de distancia (el valor máximo de cálculos de distancia es el valor de n).

6.3.3 Gráficas variando los K vecinos más cercanos

Resulta claro que si tenemos un número grande de pivotes y un número pequeño de K vecinos más cercanos a encontrar, el radio inicial en la cola de prioridad será mucho más pequeño que en el caso contrario; por ser los K vecinos más cercanos una parámetro importante en este trabajo, en las siguientes secciones se muestran como influye el número de pivotes, la dimensión, para diferentes valores de K.

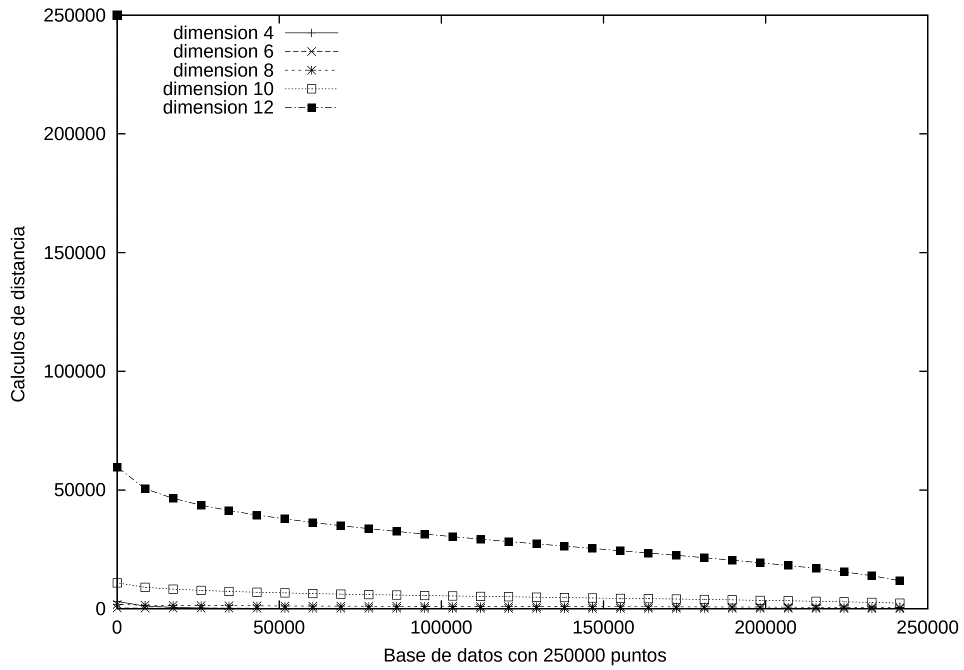


Figura 6.15: Gráfica con 250,000 objetos, usando 32 pivotes, 8 bits, K=16, para diferentes dimensiones

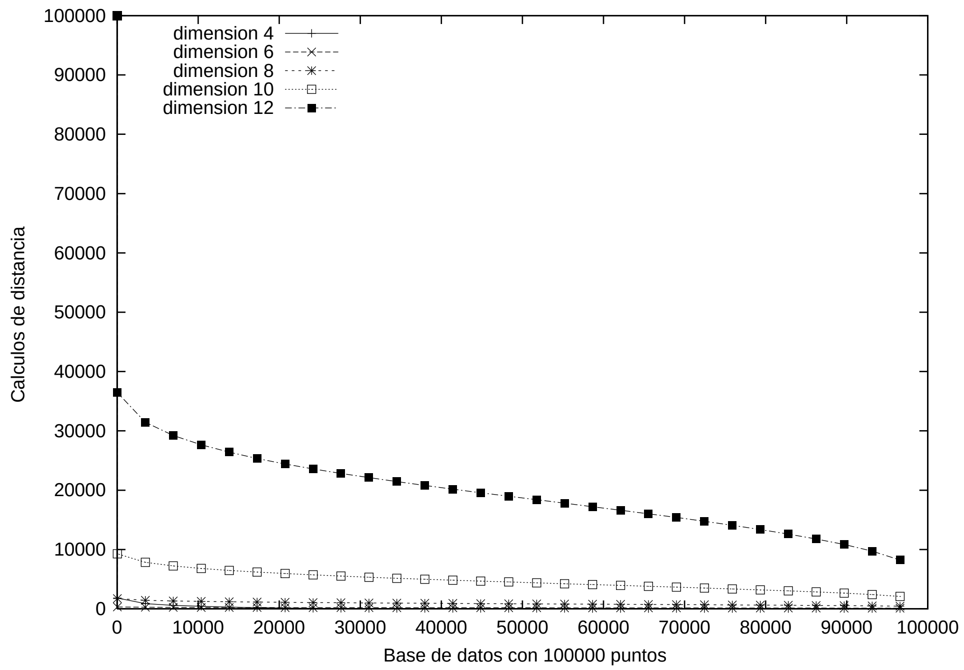


Figura 6.16: Gráfica con 100,000 objetos, usando 32 pivotes, 8 bits, K=16, para diferentes dimensiones

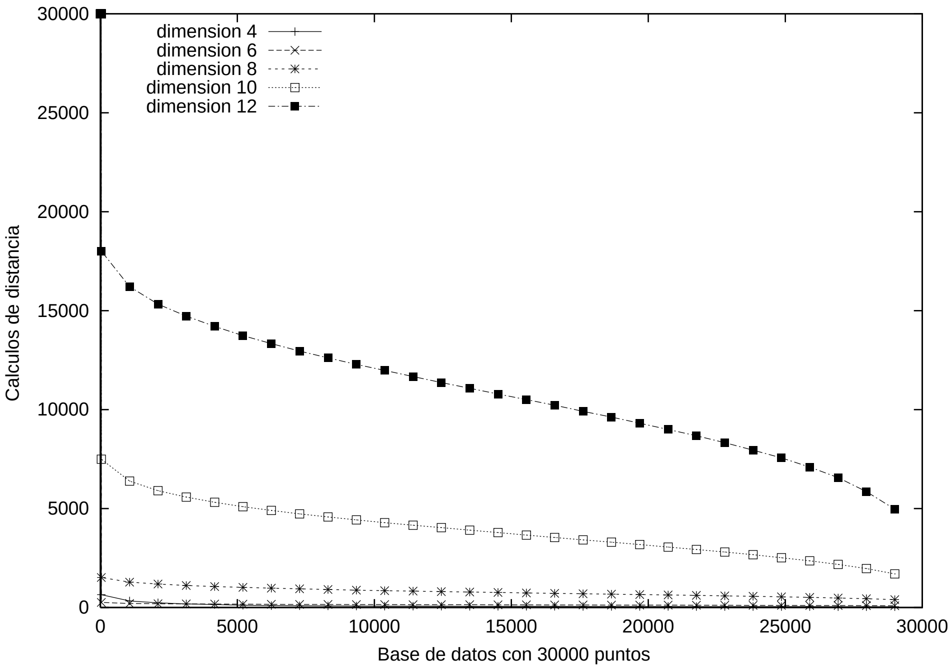


Figura 6.17: Gráfica con 30,000 objetos, usando 32 pivotes, 8 bits, K=16, para diferentes dimensiones

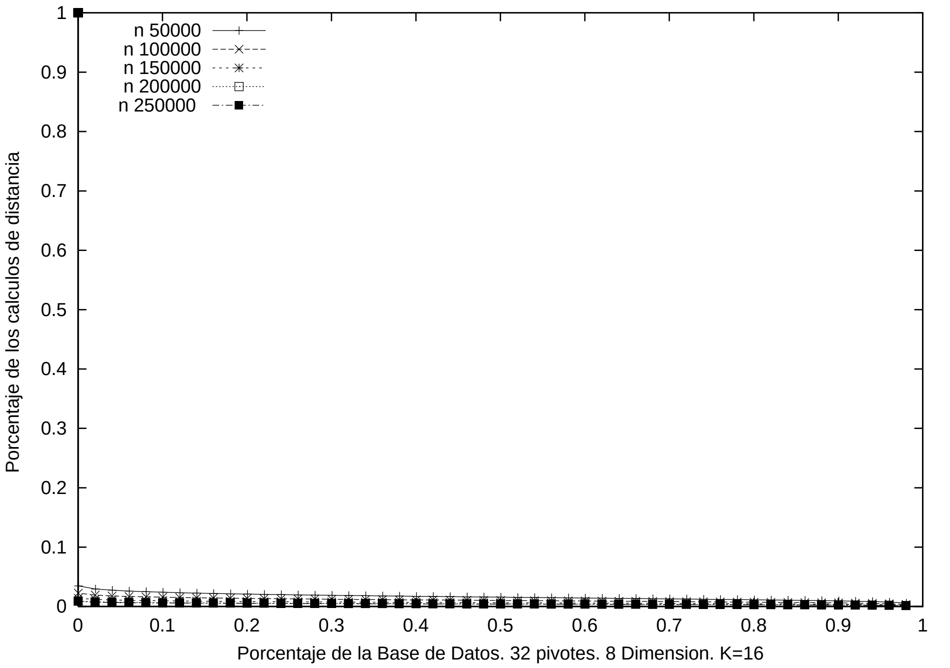


Figura 6.18: Gráfica con diferentes tamaños de la Base de datos, usando 32 pivotes, 8 bits, en dimensión 8 y K=16

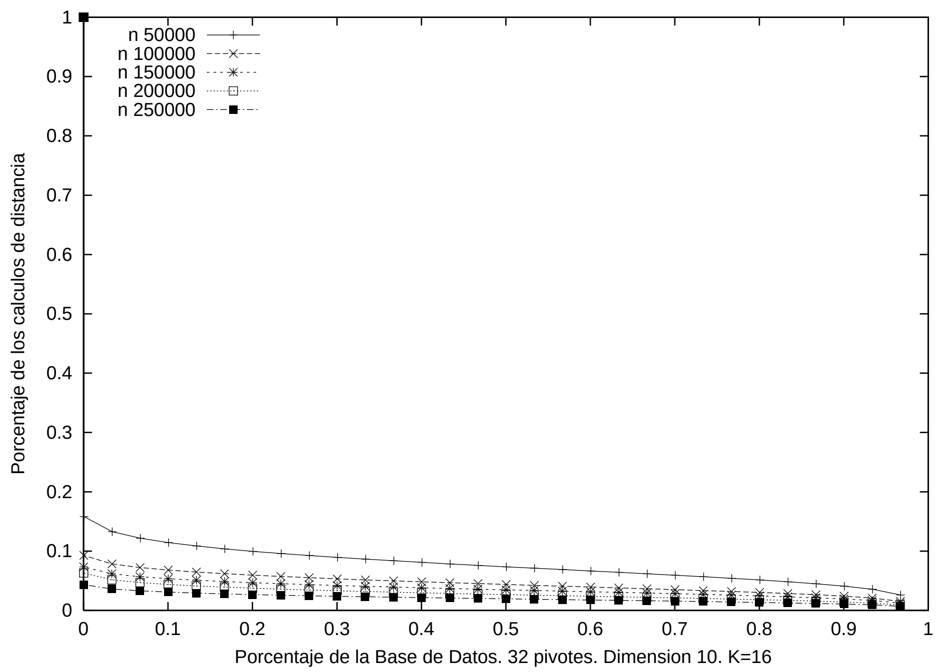


Figura 6.19: Gráfica con diferentes tamaños de la Base de datos, usando 32 pivotes, 8 bits, en dimensión 10 y K=16

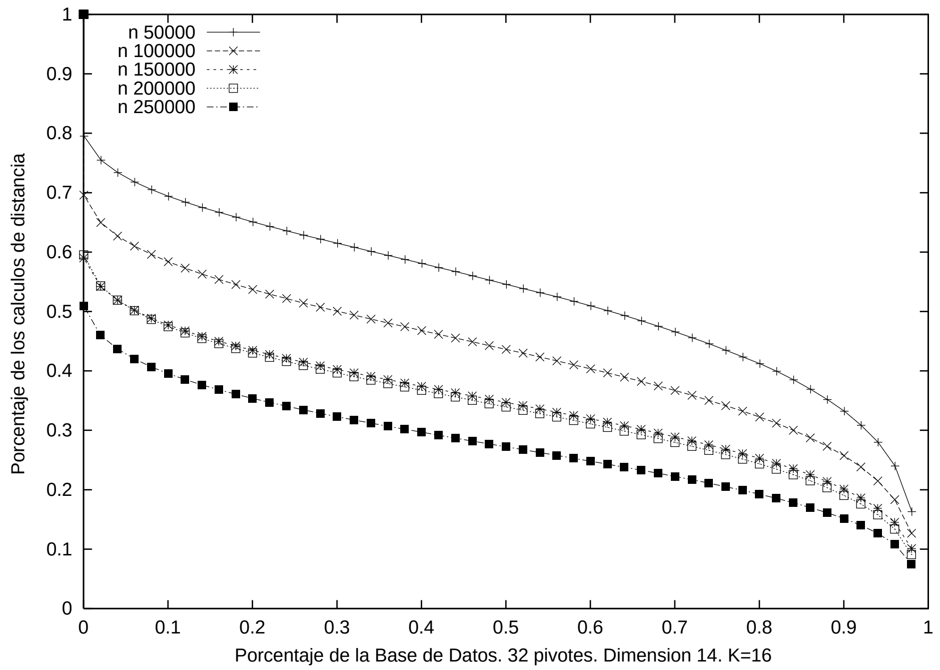


Figura 6.20: Gráfica con diferentes tamaños de la Base de datos, usando 32 pivotes, 8 bits, en dimensión 14 y K=16

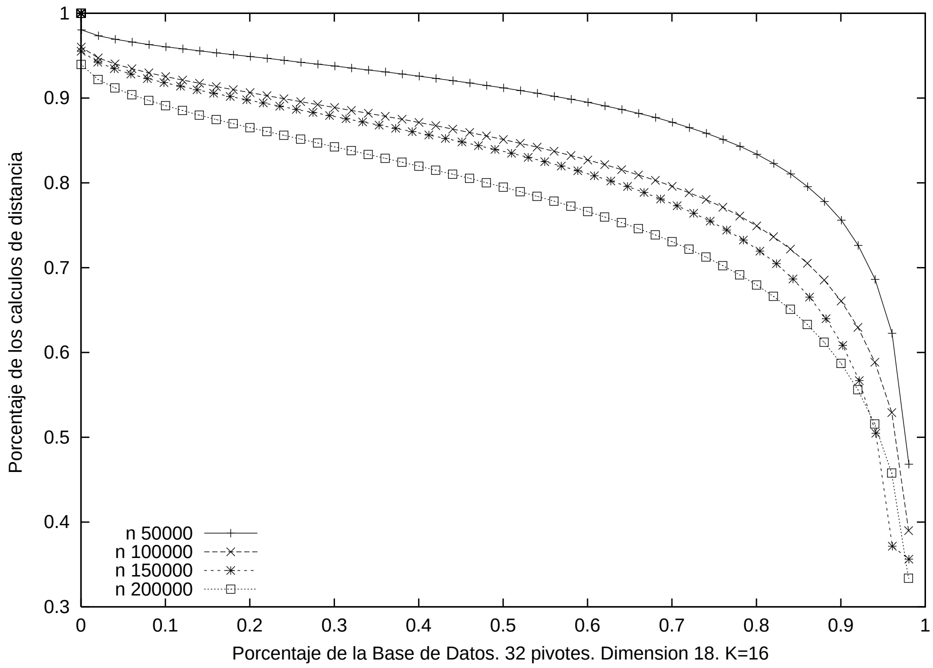


Figura 6.21: Gráfica con diferentes tamaños de la Base de datos, usando 32 pivotes, 8 bits, en dimensión 18 y K=16

Diferentes pivotes

En las gráficas de las figuras 6.22, 6.23, 6.24 y 6.25 muestran el comportamiento del algoritmo para diferentes K y como influye el número de pivotes con este comportamiento, las cuatro gráficas fueron desarrolladas para una dimensión de 8. En estas gráficas es importante notar que a medida que se usan más y más pivotes se reduce el número de cálculos de distancia necesarios para encontrar los K vecinos más cercanos, sin embargo es claro que a medida que crece el número de pivotes, se necesita más memoria para construir el índice.

Diferentes dimensiones

En las figuras 6.26 y 6.27 se muestra como influye la dimensión para el cálculo de los K vecinos más cercanos, las diferentes gráficas que se muestran presentan el número de cálculos de distancia contra diferente cantidad de puntos, para 32 y 64 vecinos más cercanos respectivamente, usando 8 pivotes. para las gráficas de las figuras 6.28 y 6.29, se usaron 32 pivotes y finalmente para las gráficas de las figuras 6.30 y 6.31, se usaron 64 pivotes. En estas gráficas se observa que a medida que crece la dimensión crece también el número de cálculos de distancia necesarios para satisfacer la consulta.

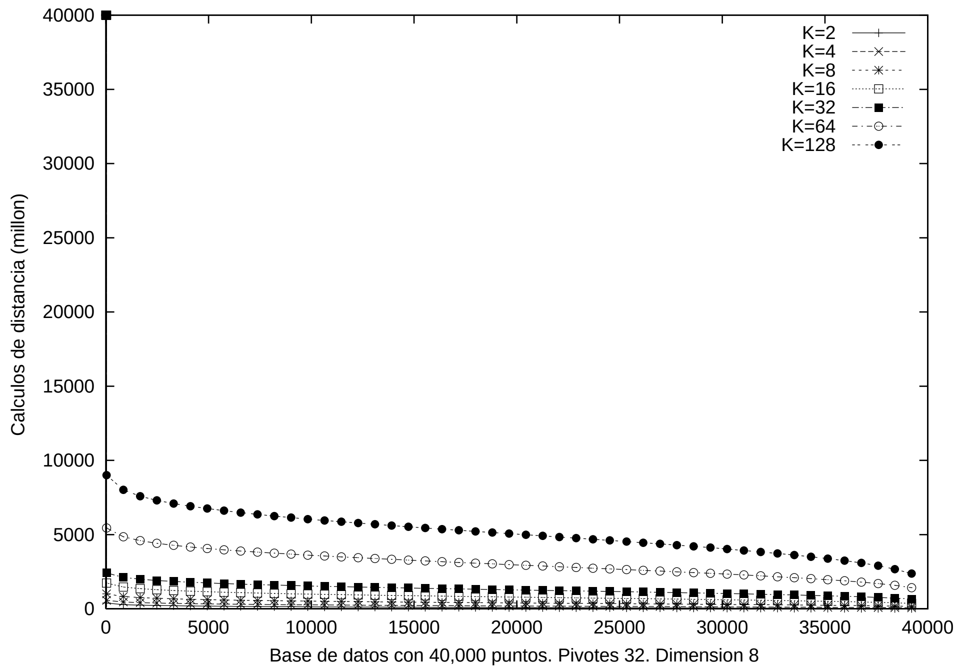


Figura 6.22: Gráfica con 40,000 objetos, en dimensión 8, con 32 pivotes, 8 bits, para diferentes K

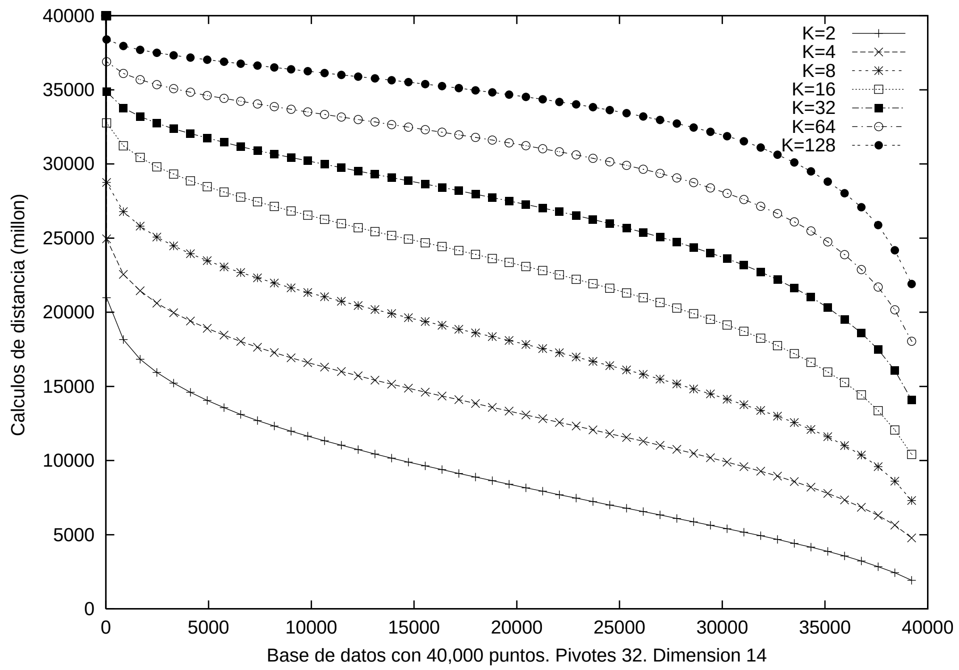


Figura 6.23: Gráfica con 40,000 objetos, en dimensión 14, con 32 pivotes, 8 bits, para diferentes K

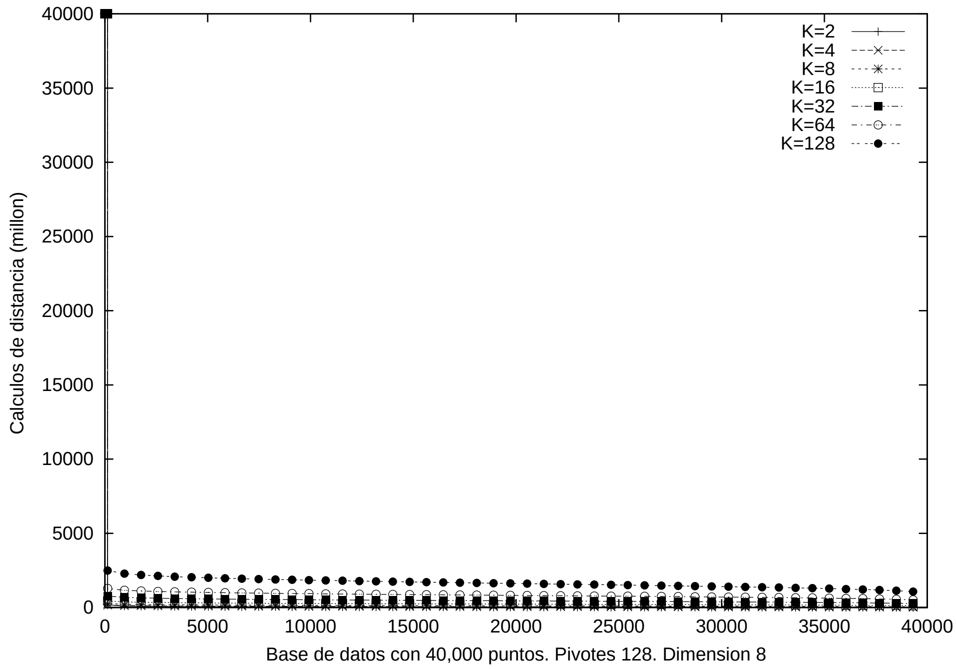


Figura 6.24: Gráfica con 40,000 objetos, en dimensión 8, con 128 pivotes, 8 bits, para diferentes K

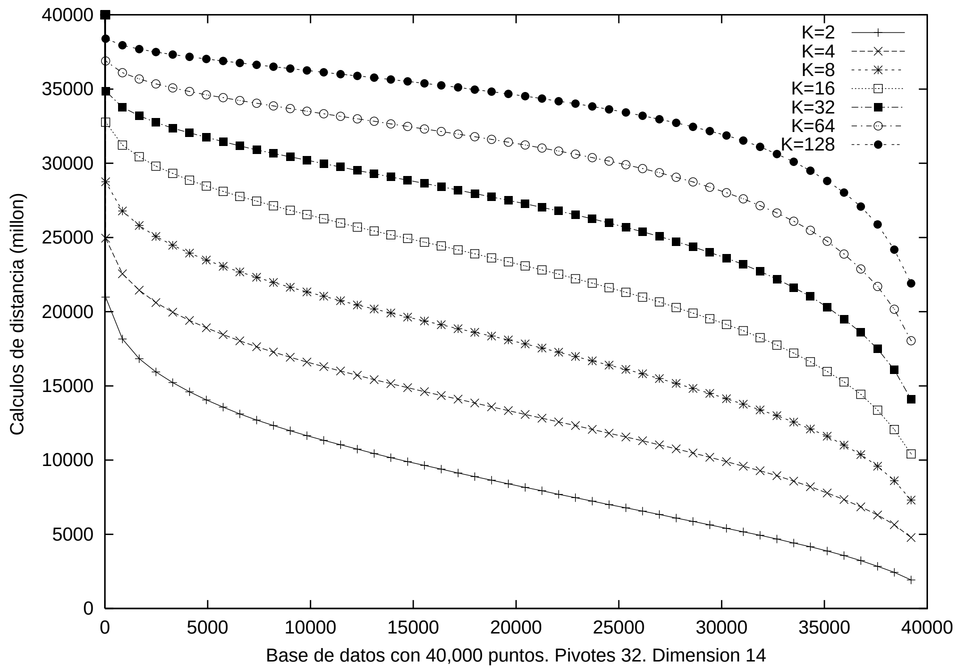


Figura 6.25: Gráfica con 40,000 objetos, en dimensión 14, con 128 pivotes, 8 bits, para diferentes K

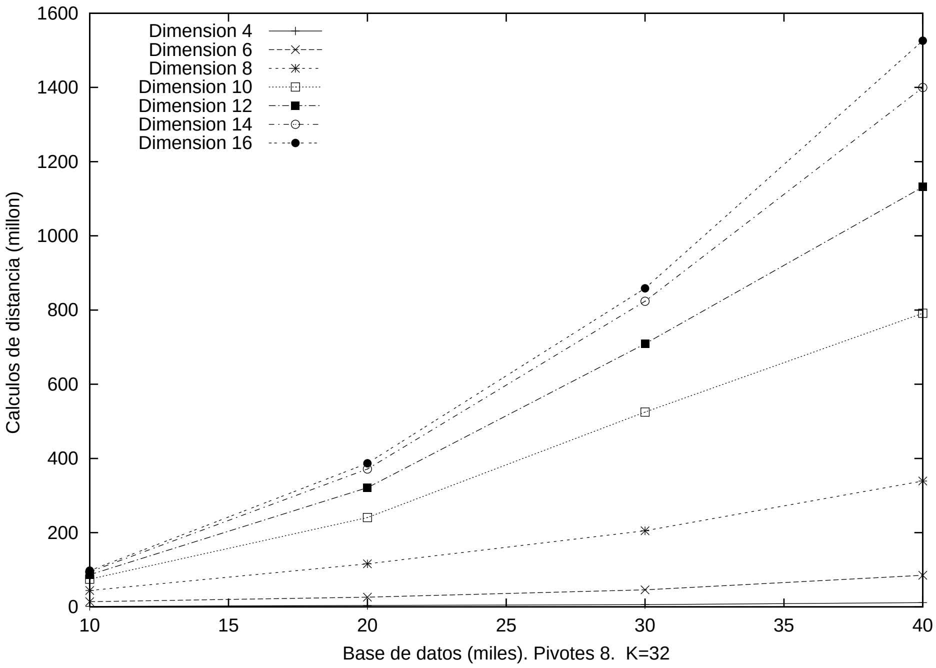


Figura 6.26: Gráfica con diferentes bases de datos, variando la dimensión, para 8 pivotes, 8 bits y K=32

Es importante mencionar que a partir de la gráfica 6.26 cada punto corresponde al número total de cálculos de distancia, es decir, cada punto es una corrida completa.

Variando los K vecinos más cercanos

Si mantenemos constante el número de puntos y la dimensión, podremos obtener gráficas como las mostradas en esta sección donde se presentan el número cálculos de distancia contra valores de K vecinos más cercanos, cada línea de la gráfica corresponde a un número de pivotes.

En las gráficas de las figuras 6.32, 6.33, 6.34 y 6.35, las dos primeras son para 20,000 puntos con dimensión 8 y 12 respectivamente, las siguientes dos son para 40,000 puntos con dimensión 14 y 16 respectivamente.

En las gráficas de las figuras 6.32, 6.33, 6.34 y 6.35 se puede apreciar, como a medida que aumenta el número de pivotes disminuye notablemente el número de cálculos de distancia, también se puede ver el efecto que causa la dimensión para estos cálculos, y finalmente, se puede apreciar, como importa también el número de los K vecinos más cercanos.

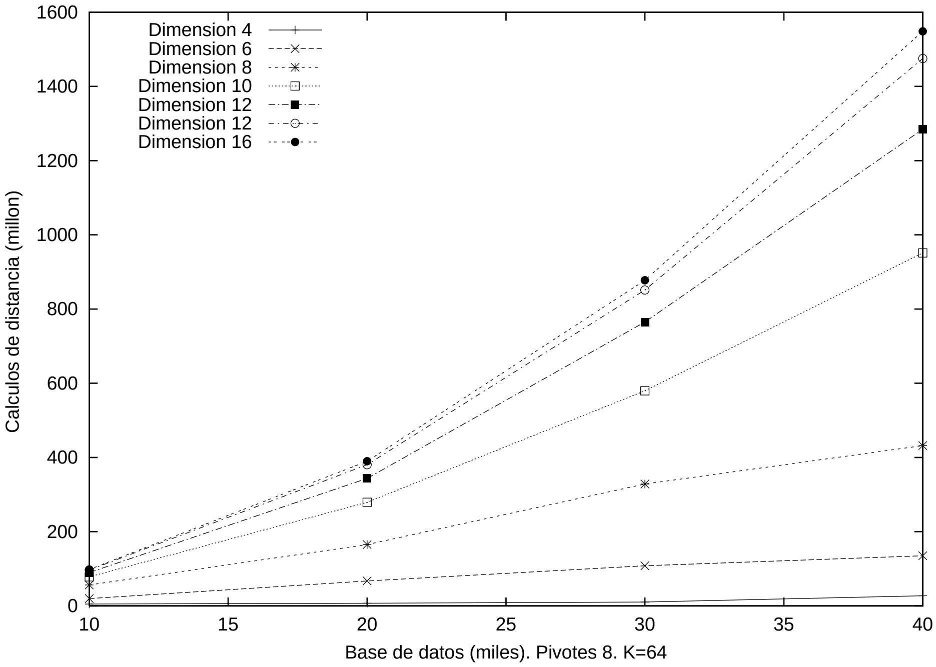


Figura 6.27: Gráfica con diferentes bases de datos, variando la dimensión, para 8 pivotes, 8 bits y K=64

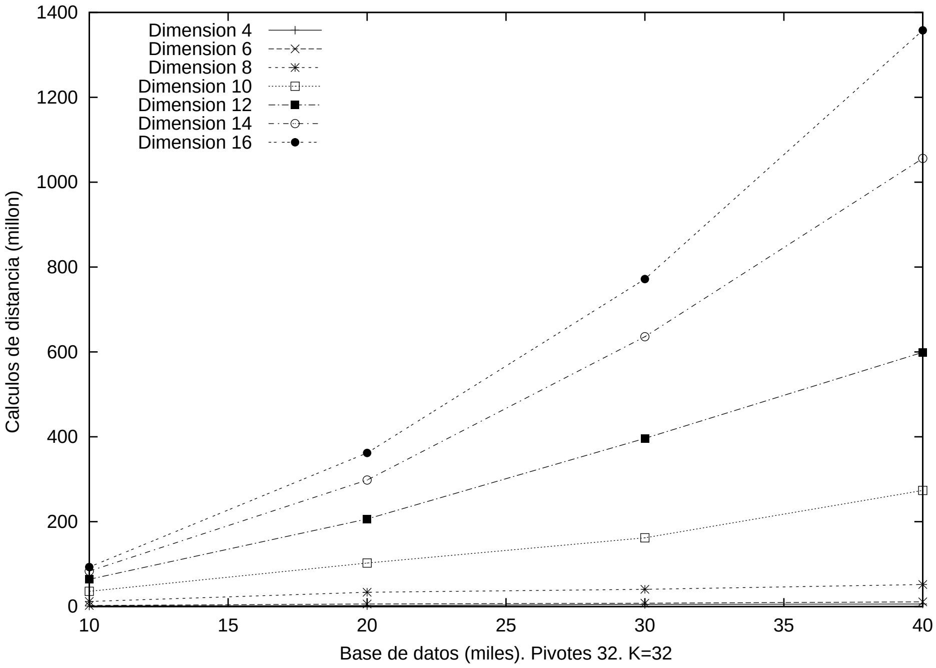


Figura 6.28: Gráfica con diferentes bases de datos, variando la dimensión, para 32 pivotes, 8 bits y K=32

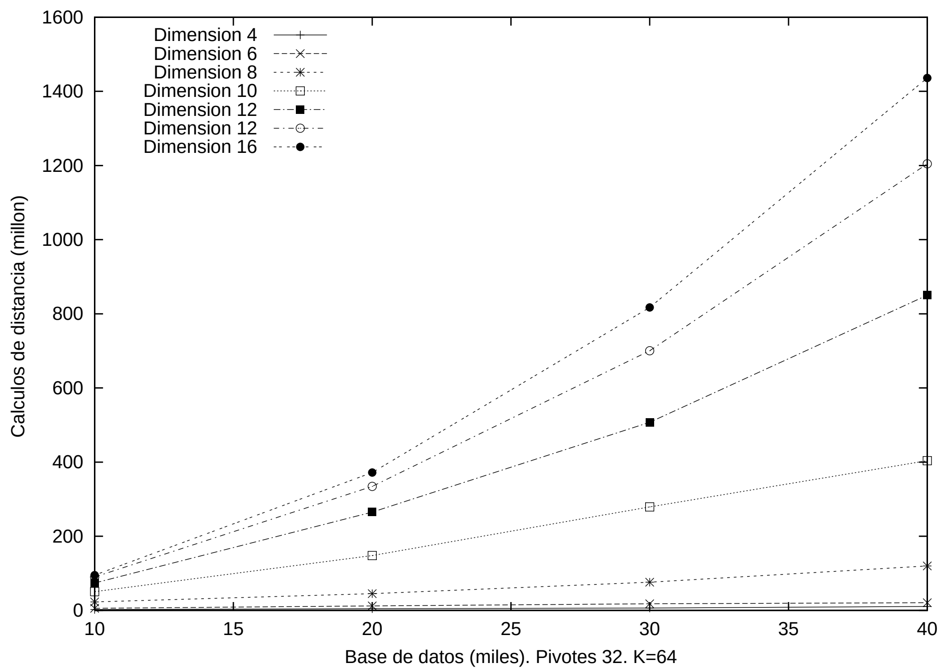


Figura 6.29: Gráfica con diferentes bases de datos, variando la dimensión, para 32 pivotes, 8 bits y K=64

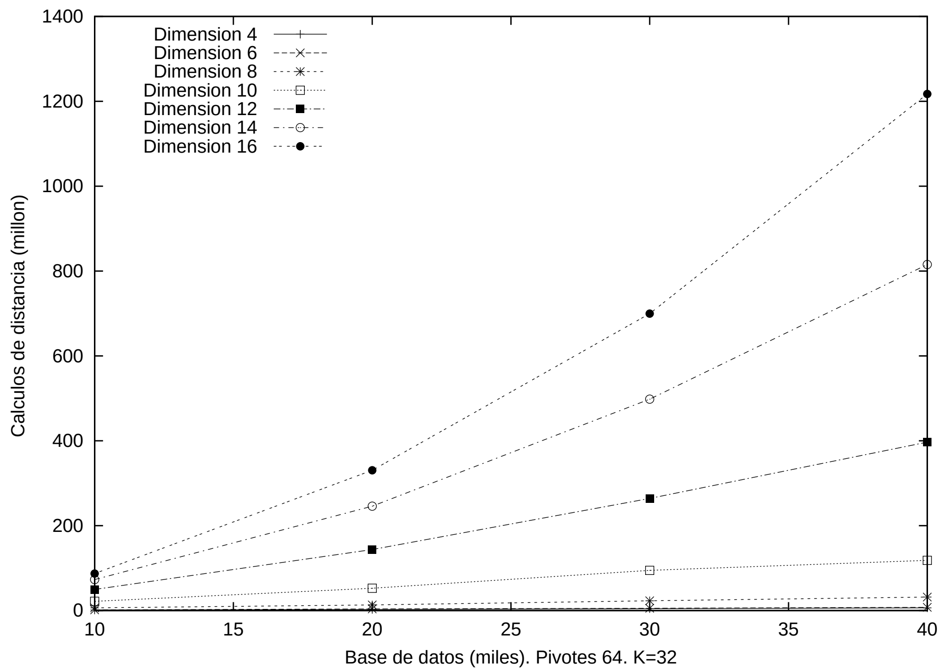


Figura 6.30: Gráfica con diferentes bases de datos, variando la dimensión, para 64 pivotes, 8 bits y K=32

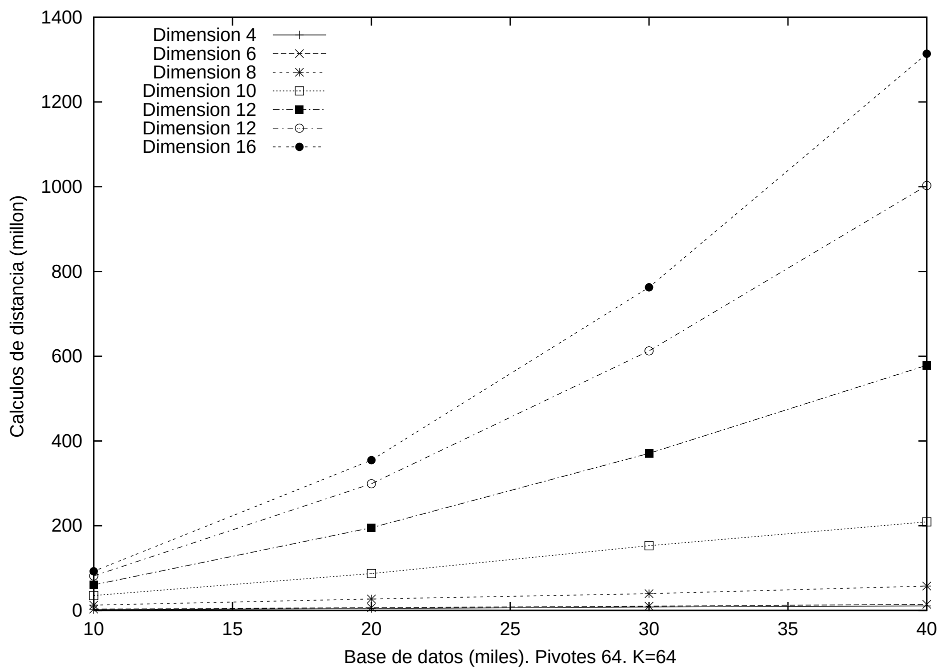


Figura 6.31: Gráfica con diferentes bases de datos, variando la dimensión, para 64 pivotes, 8 bits y K=64

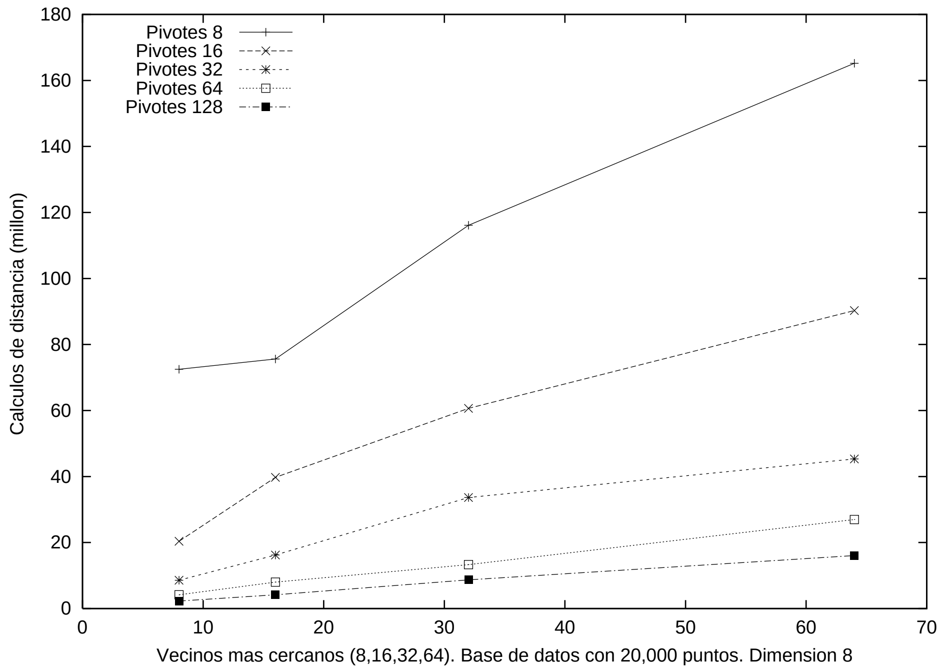


Figura 6.32: Distancias contra K vecinos más cercanos, para 20,000 puntos en dimensión 8, variando el número de pivotes

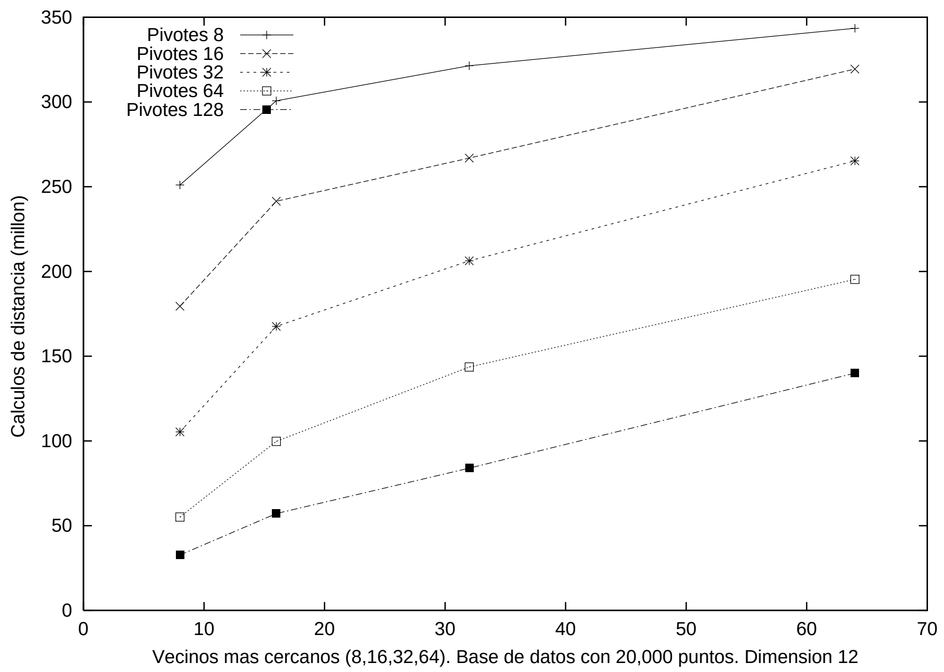


Figura 6.33: Distancias contra K vecinos más cercanos, para 20,000 puntos en dimensión 12, variando el número de pivotes

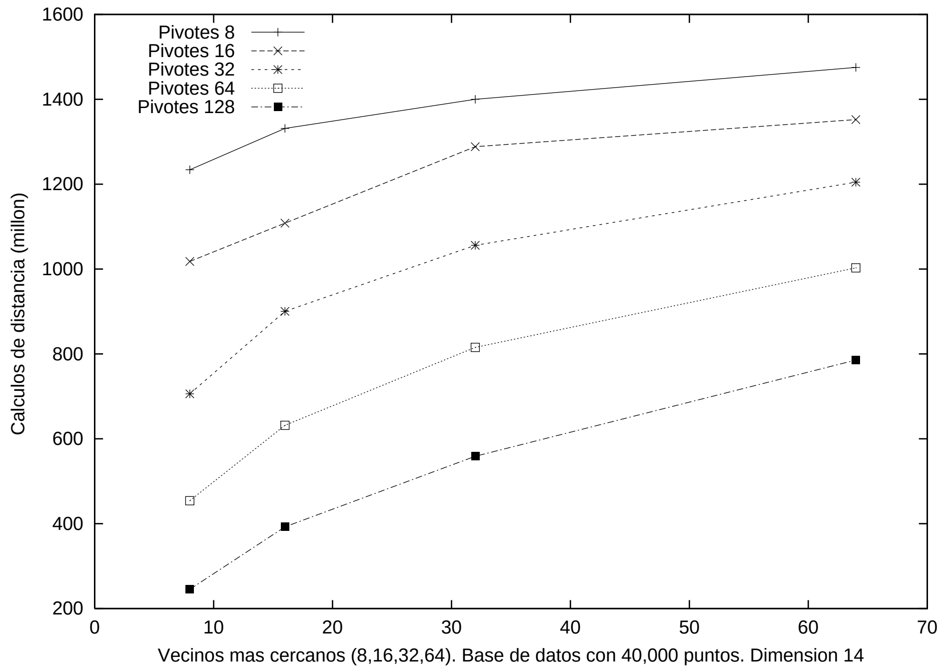


Figura 6.34: Distancias contra K vecinos más cercanos, para 40,000 puntos en dimensión 14, variando el número de pivotes

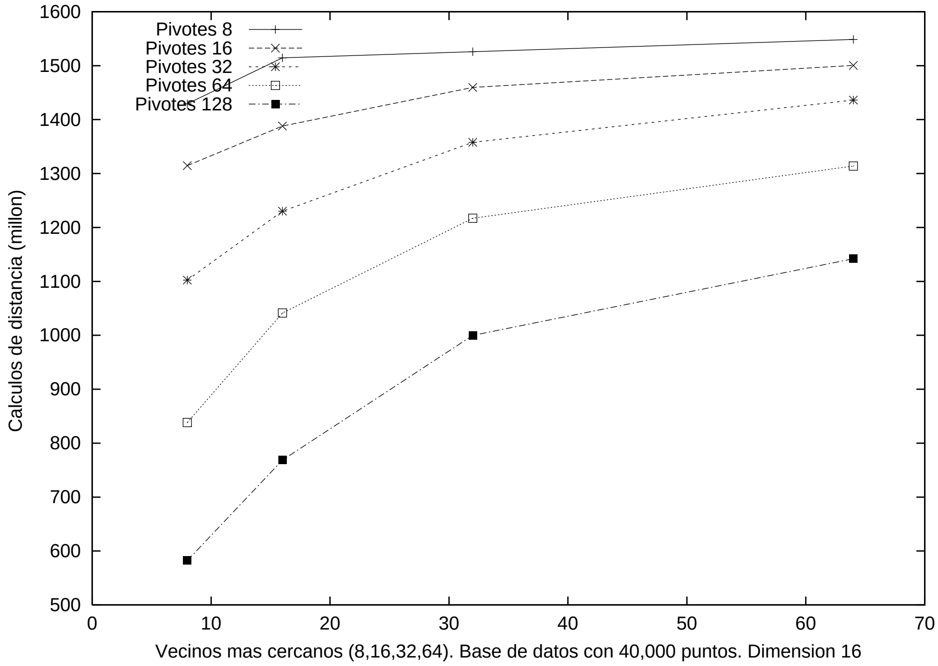


Figura 6.35: Distancias contra K vecinos más cercanos, para 40,000 puntos en dimensión 16, variando el número de pivotes

6.3.4 Gráfica variando los pivotes

Las gráficas mostradas en las figuras 6.36, 6.37 y 6.38 varían el número de pivotes, es decir, cada línea de la gráfica corresponde a un número de pivotes diferente, y en la parte inferior se varía la cantidad de puntos y se varían también el número de K vecinos más cercanos, para una dimensión de 6, en las figuras 6.39, 6.40 y 6.41 la dimensión ocupada es 12, con los mismos parámetros.

Obsérvese que si la dimensión crece, aumenta los cálculos de distancia, si aumenta el número de pivotes, disminuye el número de cálculos de distancia al momento de realizar la búsqueda de todos los K vecinos más cercanos, sin embargo es importante mencionar que aumenta el número de cálculos de distancia que se deben realizar para construir el índice.

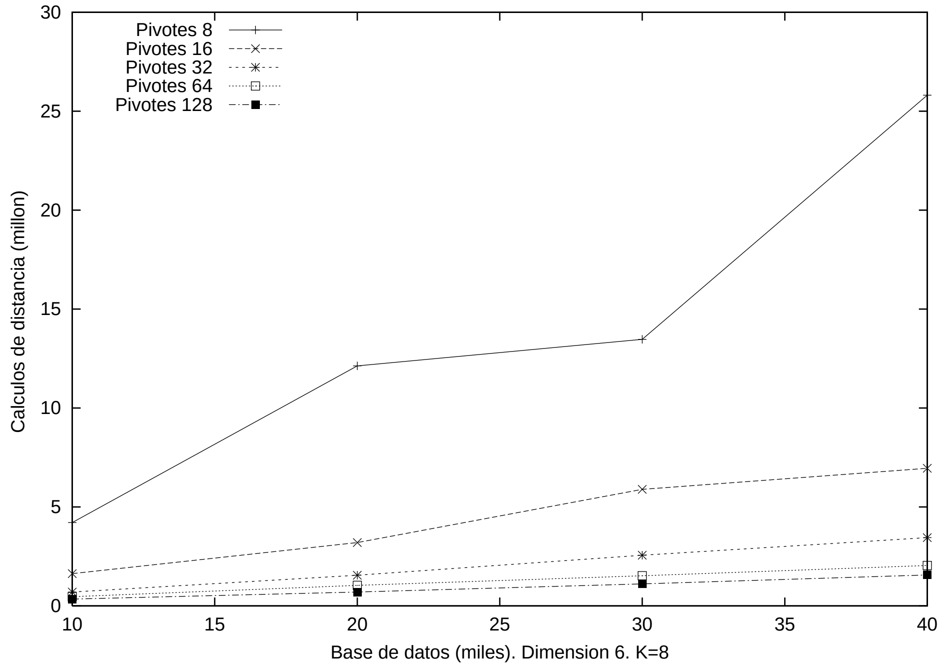


Figura 6.36: Gráfica con diferentes bases de datos, variando el número de pivotes, en dimensión 6, para K=8

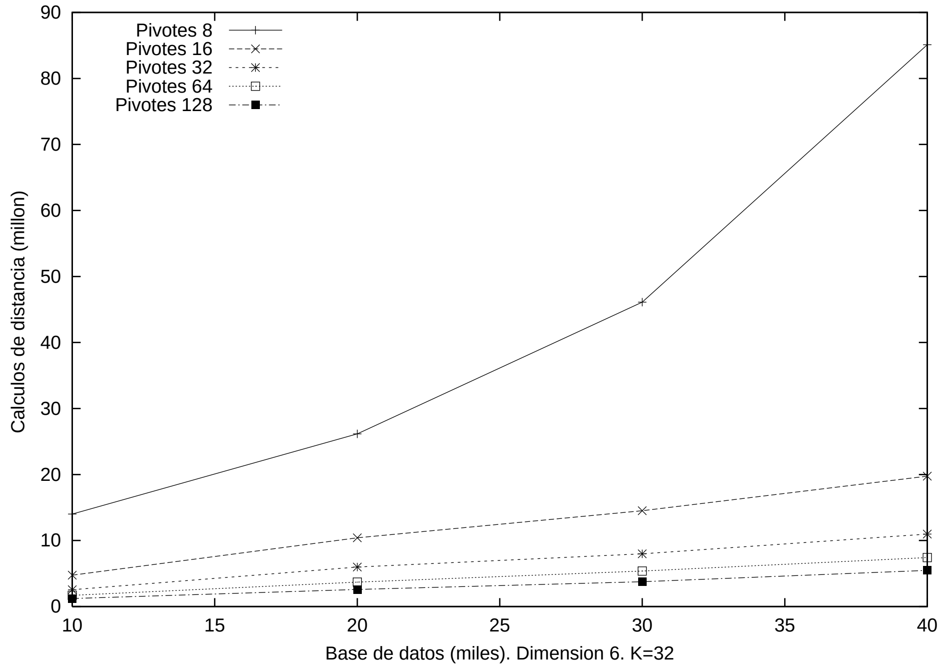


Figura 6.37: Gráfica con diferentes bases de datos, variando el número de pivotes, en dimensión 6, para K=32

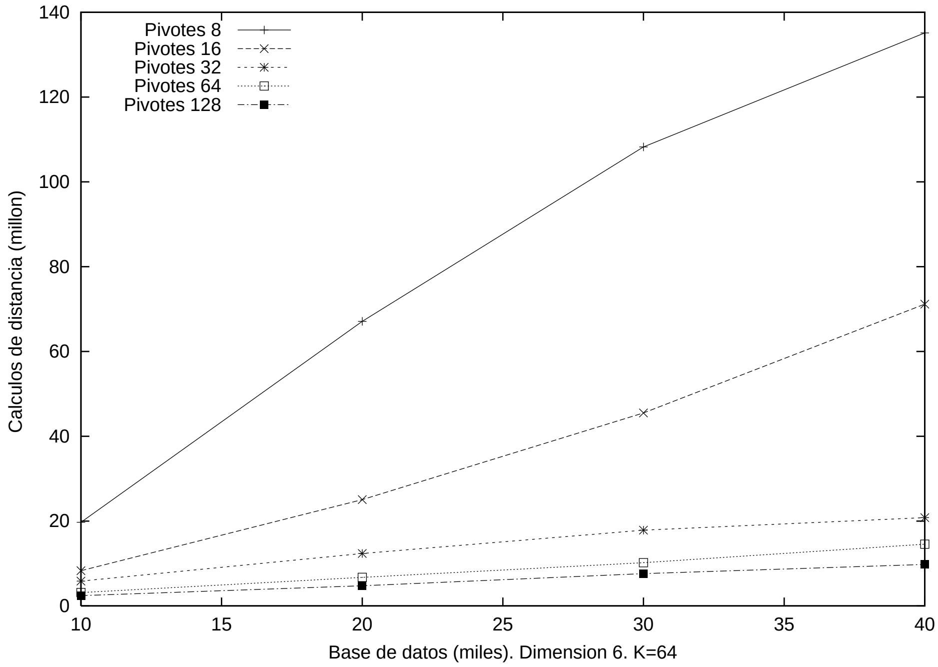


Figura 6.38: Gráfica con diferentes bases de datos, variando el número de pivotes, en dimensión 6, para K=64

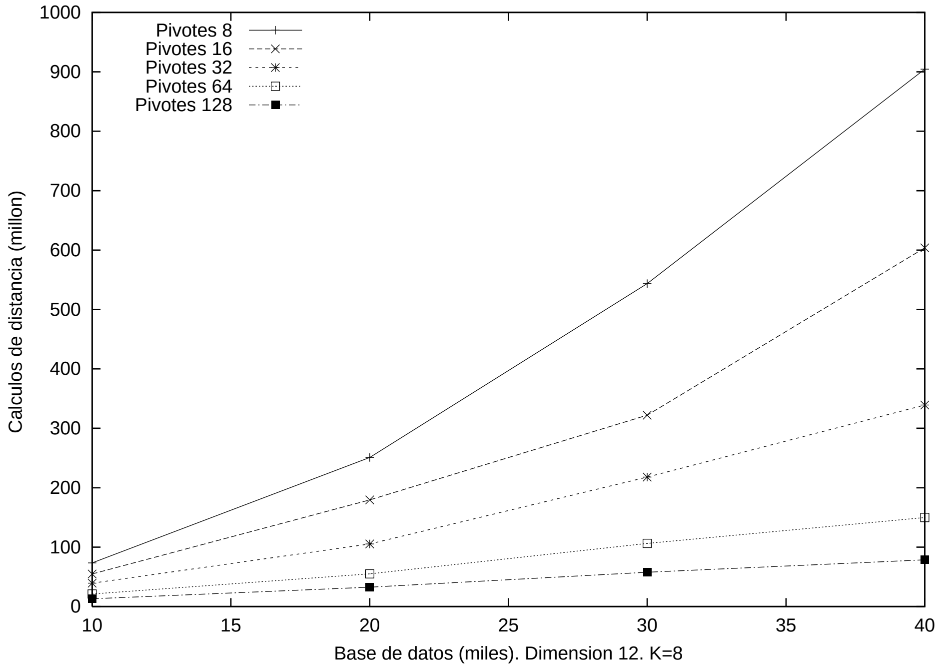


Figura 6.39: Gráfica con diferentes bases de datos, variando el número de pivotes, en dimensión 12, para K=8

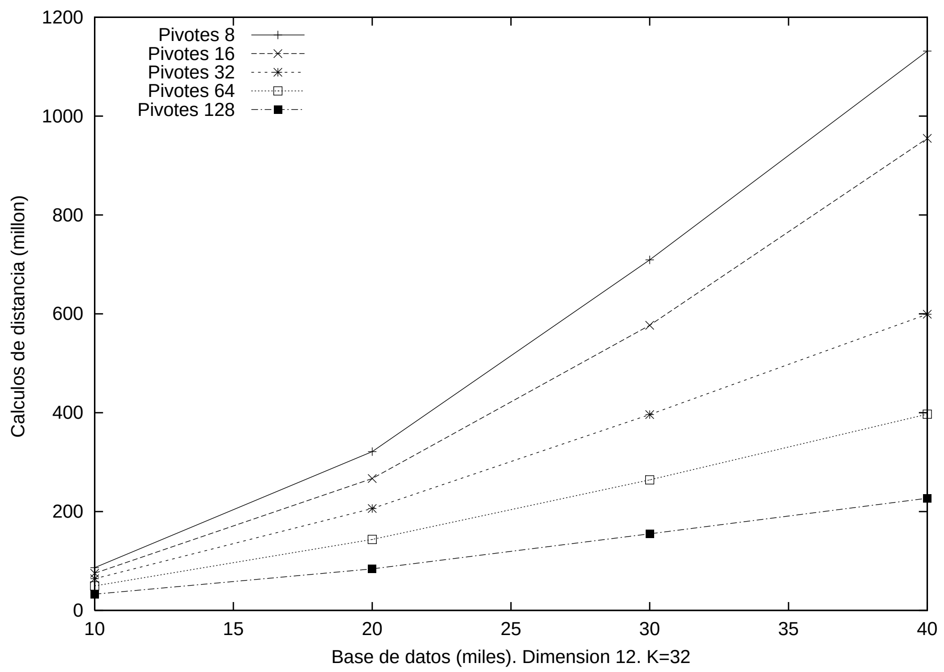


Figura 6.40: Gráfica con diferentes bases de datos, variando el número de pivotes, en dimensión 12, para K=32

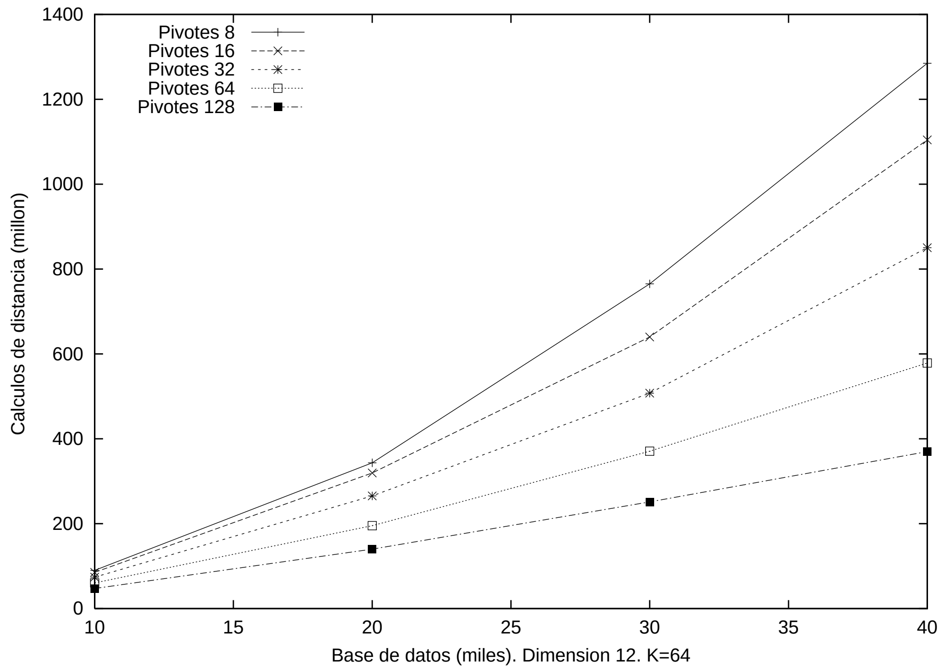


Figura 6.41: Gráfica con diferentes bases de datos, variando el número de pivotes, en dimensión 12, para K=64

Capítulo 7

Conclusiones y trabajos futuros

Finalmente, antes de presentar las conclusiones técnicas de este trabajo, se mencionarán algunas conclusiones prácticas. Durante la realización de este trabajo, se analizaron muchos algoritmos, se implementaron algunos de ellos, de los cuales algunos resultaron sin el éxito esperado, sin embargo las aportaciones de estos fueron muchas, entre los algoritmos que se desarrollaron están, implementar la búsqueda binaria al FQA, la construcción del FQT, un algoritmo para resolver el problema de la disparidad, un algoritmo dinámico semejante a SAT [Nav99b] llamado DSAT [CFG99], un algoritmo de búsqueda con los resultados arrojados por el algoritmo de esta tesis (todos los K vecinos más cercanos).

7.1 Conclusiones

Como fruto de este trabajo, se presentó un algoritmo de indexamiento llamado FQTrie, el cual es una combinación de los algoritmos FQA y FQT, obteniendo como resultado una estructura mitad árbol y mitad arreglo, con esta estructura se mejora la memoria necesaria en el FQT y se mejora el tiempo la búsqueda del FQA, conserva las ventajas de los algoritmos basados en pivotes y el tiempo empleado en el indexamiento es lineal en n (tamaño de la base de datos). Cabe mencionar que este algoritmo es necesario en el desarrollo del algoritmo que soluciona el problema de los K vecinos más cercanos. La decisión de usar este algoritmo para solucionar el problema principal radica en sus ventajas: su tiempo de indexamiento, la eficiencia de la búsqueda que se puede realizar en él, y la cantidad de memoria utilizada.

Antes de abordar las ventajas del algoritmo presentado, se mencionará la importancia de la dimensión intrínseca en un algoritmo de indexamiento, para entender este problema se propone utilizar un histograma de distancias para caracterizar la dificultad de la búsqueda en un espacio métrico arbitrario, en la sección 6.3.1 , se dio una explicación intuitiva del porque el problema es más difícil a medida que la dimensión crece. Cuando se habla de la dimensión se refiere a la dimensión representativa de los datos y cuando se habla de la dimensión intrínseca esta última trata de capturar la dimensión real del espacio en la que están los datos.En [CN99] se propone una medida para cuantificar la dimensión intrínseca de un algoritmo basado en pivotes.

Según la definición dada en [CN99], la dimensión intrínseca de un espacio métrico esta definido como $\rho = \frac{\mu^2}{2\sigma^2}$, donde μ y σ^2 son la media y la varianza del histograma de distancias correspondiente.

En los algoritmos basados en pivotes, cada vez que se realiza una consulta, se hacen un número de cálculos de distancia y es posible saber cuantos cálculos de distancia se realizarán, en [CN99] se demuestra que el límite inferior para conocer el costo promedio de búsqueda usando un algoritmo basado en pivotes es:

$$costo \geq N_P + n\left(1 - \frac{2\sigma^2}{r^2}\right)^2 \tag{7.1}$$

donde N_P es el número de pivotes utilizado, σ^2 es la varianza de los elementos de la base de datos y r es radio promedio de las búsquedas aplicadas a un algoritmo basado en pivotes, una descripción más detallada, se puede observar en el apéndice C.1.

En este trabajo se presentó un algoritmo que encuentra los K vecinos más cercanos de todos los elementos en una base de datos, las ventajas de este algoritmo son las enlistadas a continuación.

- Parámetros de ajuste.- Entre las cosas que deben tomarse en cuenta en una aplicación se encuentran la memoria y el tiempo de ejecución. En algunas ocasiones el usuario está dispuesto a gastar memoria, por tiempo de ejecución, y viceversa. Un parámetro de ajuste, es precisamente la libertad que se le concede al usuario de decidir entre tiempo de ejecución y memoria. El algoritmo de todos los K vecinos más cercanos presentado, tiene algunos parámetros de ajuste, los cuales modifican el desempeño del algoritmo, a continuación se presentan dichos parámetros.
  - Número de pivotes.- Permite seleccionar la cantidad de pivotes utilizados, tanto en la construcción del FQTrie, como en la búsqueda de los K vecinos más cercanos. Las conclusiones acerca de este parámetro son: a medida que incrementa el número de pivotes aumenta el tiempo de construcción del FQTrie linealmente y disminuye el tiempo de respuesta en la búsqueda de todos los K vecinos más cercanos.

    Siendo un parámetro tan importante el número de pivotes, cabe mencionar que la mejor manera, para encontrar los puntos que serán pivotes es de manera aleatoria, debido a que con los puntos seleccionados de esta manera (observe en las gráficas mostradas en el capítulo anterior) se obtiene un buen rendimiento del algoritmo y no es necesario hacer ningún cálculo de distancia extra, si se implementara alguna heurística para conocer los mejores puntos como pivotes, el costo del algoritmo sería, el costo de la construcción del índice más el costo de la selección de pivotes más el costo del algoritmo para encontrar los K vecinos y además el tiempo que tarda cada uno de estos procesos, por lo tanto se recomienda que esta selección sea aleatoria.

En [CN99] se sugiere el número óptimo de pivotes es:

$$P = \frac{\ln n + \ln \ln(1/t)}{\ln(1/t)} \tag{7.2}$$

donde n es el tamaño de la base de datos, y t es $1 - \frac{2\sigma^2}{r_0^2}$, donde σ^2 es la varianza de la base de datos y r_0 es el radio promedio de consulta, la demostración de 7.2 puede verse en el apéndice C.1.

Un valor muy importante es el número de vecinos para cada punto, este valor varía de acuerdo tipo de problema a resolver, es decir, el número de los vecinos más cercanos depende directamente del problema que se pretende resolver. Como sugerencia, el número de vecinos más cercanos debe ser menor que el número de pivotes, debido a las características del algoritmo.

- Número de Bits.- El manejo de memoria siempre es importante en cualquier algoritmo, con este parámetro es posible seleccionar la cantidad de memoria empleada en el indexamiento, es claro, que a medida que aumenta el número de bits, mayor es la memoria necesaria para realizar el indexamiento, sin embargo, si aumenta el número de bits, disminuye el número de cálculos de distancia al momento de la consulta. Como sugerencia, se recomienda que sean 8 bits.

- En algunas aplicaciones solo se requiere una buena aproximación de los vecinos más cercanos, para esto, es importante mencionar, que este algoritmo, puede detenerse en el momento que se desee, y cada elemento tendría una aproximación buena de sus vecinos más cercanos. Si se requiere saber con exactitud los vecinos más cercanos de todos los puntos, entonces si es necesario esperar a que el algoritmo termine. Debido a las características del algoritmo al momento de consultar un elemento para conocer sus vecinos más cercanos, se modifican los vecinos más cercanos de otros puntos, esta es la razón por la que es posible detener el algoritmo en cualquier momento.

Una vez que se obtuvo el algoritmo, se hicieron algunas pruebas, acerca de como empezar la búsqueda de los vecinos mas cercanos, esto con la finalidad de probar, si importa el orden de los puntos en que se empieza la búsqueda de los vecinos más cercanos, se observó que no existen diferencias significativas en el desempeño del algoritmo; las dos heurísticas planteadas para realizar esta pruebas fueron:

- Radios máximos primeros.- Empezar la búsqueda de los vecinos mas cercanos primero con los elementos cuyo radio inicial fuera el mayor de todos, esto con el fin de que muchos más puntos fueran beneficiados con la búsqueda de un elemento. La desventaja de esta heurística, es que se necesita una cola de prioridad para mantener ordenados los puntos de acuerdo al radio máximo, con esto, es obvio que se necesita más memoria, y más tiempo de procesamiento. Los resultados de esta prueba, no incrementan significativamente el desempeño del algoritmo.

- Radios mínimos primeros.- Se planteó empezar la búsqueda con los elementos que tuvieran el radio inicial mínimo, pretendiendo con esto que los puntos con un radio más grande, se redujera y al momento de consultar estos otros puntos su radio ya no fuera tan grande. Si un radio inicial esta muy próximo al correspondiente radio que encierra los K vecinos, se realizaran menos cálculos de distancia. La desventaja de esta heurística es que se necesita una cola de prioridad para mantener los elementos ordenados por su distancia menor, lo que implica, incrementar la memoria usada y un incremento al tiempo de procesamiento. Esta otra heurística, tampoco muestra un incremento considerable en el desempeño del algoritmo.

Es importante para todo algoritmo, que se realicen comparaciones con respecto a los algoritmos existentes, en misma circunstancias, así que, las pruebas finales aplicadas a este algoritmo, fueron tratar de compararlo con respecto a los algoritmos existentes, por desgracia, no fue posible conseguir el algoritmo realizado por Vaidya, así que esta comparación no se llevo a cabo, el otro algoritmo desarrollado por Clarkson, además de que tampoco fue posible conseguirlo, no tiene parámetros de ajuste, por lo tanto, el algoritmo presentado en este trabajo podría superarlo modificando sus parámetros de ajuste.

7.2 Análisis del algoritmo

Las gráficas presentadas en la sección anterior muestran el comportamiento del algoritmo al variar algunos parámetros, es por eso que se considera necesario analizar cada uno de estos parámetros y la repercusión que tienen sobre el algoritmo. A continuación se en listan estos parámetros y sus efectos de acuerdo a su variación.

- El número de pivotes
  - Si se incrementa el número de pivotes, disminuye el número de cálculos de distancia, al momento de buscar los K vecinos más cercanos
  - Si se incrementa el número de pivotes, aumenta la memoria utilizada en el indexamiento

dimensión	valor de $1 + \alpha$ (n = 40,000)
6	1.025
8	1.128
10	1.208
12	1.493
14	1.718

Tabla 7.1: Valores aproximados para $1 + \alpha$, en diferentes dimensiones

- La dimensión de los datos
  - Si aumenta la dimensión, aumenta el número de cálculos de distancia, al momento de buscar los K vecinos más cercanos
  - Si aumenta la dimensión, aumenta el tiempo empleado
- El tamaño de la base de datos
  - Si disminuye el tamaño de la base de datos, disminuye el costo del algoritmo, por depender directamente de este valor (si el costo del algoritmo es: $O(n^{1+\alpha})$, a medida que aumenta n aumenta el costo del algoritmo y viceversa)
- El número de vecinos más cercanos
  - A medida que aumenta K, varía insignificativamente los cálculos de distancia a realizar, debido a que la complejidad del algoritmo depende del valor de la α, veáse la tabla 7.1.

Como una conclusión de este trabajo se puede decir a partir de un análisis empírico, que el algoritmo tiene una complejidad de $O(n^{1+\alpha})$, donde $0 \leq \alpha \leq 1$, la cual depende directamente de la dimensión. Esto se deduce visiblemente de las gráficas presentadas en el capítulo anterior. Para calcular el valor de α, se aplicó el método de mínimos cuadrados a la gráfica 6.31 y se observa que el comportamiento de α es el que se presenta en la tabla 7.1 y la gráfica de estos valores es la 7.1.

De la complejidad del algoritmo presentado pueden sacarse algunas conclusiones, entre ellas están, para dimensiones bajas *alpha* tiende a ser muy pequeña y por lo tanto tiene una complejidad lineal.

7.3 Trabajos futuros

Entre los trabajos futuros podemos mencionar:

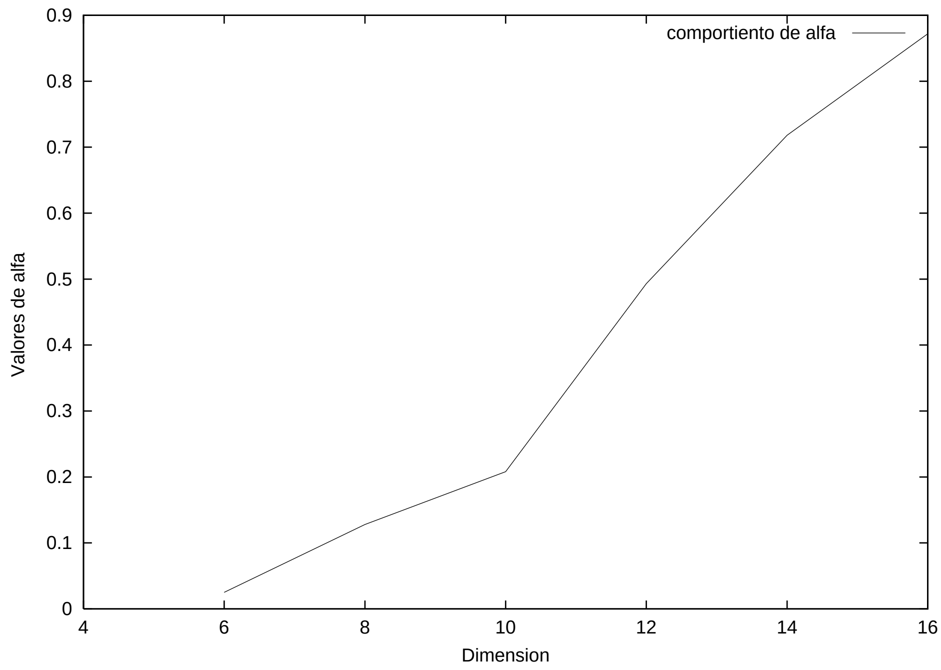


Figura 7.1: Comportamiento del valor de α para diferentes dimensiones

- El resultado del algoritmo para encontrar los K vecinos más cercanos para todos los elementos de una base de datos, puede ser aprovechado para realizar búsquedas de rango, solo con la información de los K vecinos más cercanos. La idea es utilizar la información de cada punto para aproximarse a los puntos más cercanos, por ejemplo, si se realiza una consulta de rango q con radio r y al hacer un cálculo de distancia del punto q al punto i se observa que si este punto (i) tiene un radio mayor que r (es decir, r esta contenido completamente por el radio de i entonces se saben cuales son los elementos resultantes a la consulta sin hacer más cálculos de distancia.

- En ocasiones existen bases de datos que están en constante crecimiento, de acuerdo a la aplicación. El algoritmo presentado utiliza una base de datos con elementos fijos y cada vez que se desea cambiar esta base de datos es necesario rehacer el FQTrie, ahora bien, si se tuviera una base de datos, en la que esta incrementándose la cantidad de elementos con alguna frecuencia, se podría reemplazar el algoritmo de indexamiento propuesto (FQTrie), por un algoritmo de indexamiento dinámico [yEC00], con un algoritmo de indexamiento dinámico, es posible estar agregando nuevos elementos a la base de datos y después de agregados solo se realizaría una búsqueda al igual que se realizó para los otros elementos, y este nuevo elemento conocería sus vecinos más cercanos y las colas de prioridad de los elementos implicados, se modificarían con el algoritmo, en caso de ser necesario. Es importante mencionar que eliminar un elemento (x) de la base de datos, implica que se modifiquen los radios de cada uno de los K vecinos más cercanos de x.

- Otro trabajo propuesto, y que además suena muy alentador, es mejorar la heurística para reducir el radio seguro (radio inicial), si cada búsqueda de los vecinos más cercanos, empezara con un radio menor, sería menos costoso, puesto que se realizarían menos cálculos de distancia, para resolver cada búsqueda. Esta otra heurística consiste en, inicialmente para todos los puntos, llenar su cola de prioridad, con los elementos que están cerca del pivote más cercano a cada punto, es decir, para el punto i, su cola de prioridad inicialmente tendría los vecinos más cercanos al pivote más cercano a este punto, esto reduciría aún más el radio inicial y por lo tanto el número de cálculos de distancia necesarios para conocer los K vecinos sería mucho menor y el tiempo total sería también beneficiado. Este algoritmo, parte de que cada pivote, inicialmente, tienen sus K vecinos más cercanos en su cola de prioridad.

- Otra heurística planteada para mejorar el algoritmo, es no usar una cola de prioridad para guardar los elementos más cercanos sino que por medio de la desigualdad del triángulo, saber cual es el radio mínimo en el que se encuentran los K vecinos más cercanos, esta heurística consiste en ir reduciendo el radio de cada punto a medida que se realizan cálculos de distancia, es decir, si se encuentra un punto que mejora la distancia del radio de un elemento, ahora el elemento de consulta tendrá el radio del nuevo punto, asegurando con esto que al menos existen K posibles vecinos más cercanos. Se pretende con este heurística, almacenar solo la información del radio seguro, con esto se pretende mejorar la memoria utilizada por el algoritmo, y conservar la rapidez del algoritmo. Es importante mencionar que cada vez que se quiera conocer los vecinos más cercanos de un punto, es necesario hacer una búsqueda de rango con el radio seguro de ese elemento.

- Un algoritmo aproximado, podría ser aplicado a este trabajo, y el desarrollo de este algoritmo sería, seleccionar dos puntos aleatorios y calcular la distancia entre ellos, después seleccionar otros dos puntos aleatorios, calcular la distancia y así. Este algoritmo podría repetirse varias veces y los puntos finalmente conocerían sus vecinos más cercanos. La ventaja de este algoritmo, es que no es necesario un algoritmo de indexamiento previo.

Apéndice A

Distancia de edición

Una función de distancia es diferente de acuerdo al tipo de objeto que se este manejando, en esta sección se presenta un ejemplo de una función de distancia empleada entre objetos de tipo palabra [NBY00].

A.1 Distancia de edición

El cálculo de la distancia entre palabras es un algoritmo basado en programación dinámica. Imagine que necesitamos calcular $d(x, y)$, donde x y y son palabras. Se llena una matriz $C_{0\ldots|x|,0\ldots|y|}$, donde $C_{i,j}$ representan el mínimo número de operaciones necesarias para comparar $x_{1\ldots j}$ a $y_{1\ldots i}$. Este cálculo es como sigue.

$$
\begin{aligned}
&C_{j,0} = j, C_{0,i} = i \\
&C_{j,i} = \texttt{ if } (x_j = y_i) \texttt{ then } C_{j-1,i-1} \\
&\qquad\qquad \texttt{else } 1 + min(C_{j-1,i}, C_{j,i-1}, C_{j-1,i-1})
\end{aligned}
$$

donde al final $C_{|x|,|y|} = d(x, y)$. La relación de las fórmulas mencionadas están como sigue. Primero, $C_{j,0}$ y $C_{0,i}$, representa la distancia entre una cadena de longitud j o i y la cadena vacía. Las cancelaciones de j (respectivamente i) son claramente necesarias en cadenas no vacías. Para dos cadenas no vacías de longitud j y i, asumimos inductivamente que todas las distancias entre las cadenas más cortas han sido calculadas, y tratamos convertir $x_{1\ldots j}$ a $y_{1\ldots i}$.

Considerando los últimos caracteres x_j y y_i. Si son iguales, entonces no necesitamos considerarlos y procedemos de la mejor manera posible a convertir $x_{1\ldots j-1}$ en $y_{1\ldots i-1}$. Por otro lado, si no son iguales, debemos tratarlos de alguna manera. Siguiendo las tres operaciones permitidas, podemos borrar x_j y convertir en la mejor

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	4	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

Tabla A.1: Algoritmo de programación dinámica, para realizar el cálculo de distancia entre dos palabras, "survey" y "surgery"

manera $x_{1...j-1}$ en $y_{1...i}$, insertamos y_i al final e $x_{1...j}$ y convertimos de la mejor manera $x_{1...j}$ en $y_{1...i-1}$ o remplazamos x_j por y_i y convertimos de la mejor manera $x_{1...j-1}$ en $y_{1...i-1}$. En todos los casos, el costo es 1 más el costo del resto de los procesos. Note que la inserción en una cadena son equivalentes a borrar en la otra. Note también que las implicaciones racionales $C_{j-1,i-1} \leq 1 + min(C_{j-1,i-1}, C_{j-1,i}, C_{j,i-1})$, la cual es la base para una formulación alternativa de la recurrencia:

$$C_{j-1,i-1} = min(C_{j-1,i-1} + \delta(x_j, y_i), C_{j-1,i}, C_{j,i-1})$$

donde $\delta(a, b)$ es cero para $a = b$ y 1 en otro caso.

Los algoritmos de programación dinámica deben llenar la matriz de tal manera que el vecino superior, izquierdo, y el superior izquierdo de una celda son calculados antes de calcular una celda. Esto es fácilmente alcanzado por cualquiera de los dos, recorriendo las columnas de izquierda a derecha o recorriendo las columnas transversal de arriba hacia abajo. La tabla A.1 muestra este algoritmo para calcular d("survey","surgery").

Por lo tanto, el algoritmo es $O(\mid x \mid\mid y \mid)$ tiempo en el peor de los casos. Al menos el espacio requerido es solo $O(min(\mid x \mid, \mid y \mid))$. Esto es porque, en el caso del procesamiento de una columna, solo las columnas previas deben ser almacenadas en orden para calcular la nueva, y por lo tanto solo guardamos una columna y la actualizamos. Podemos procesar la matriz de renglones o columnas, así el espacio requerido es minimizado.

Por otro lado, la secuencia de operaciones realizadas para transformar x en y pueden ser fácilmente recuperada de la matriz, simplemente desarrollando la celda $C_{|x|,|y|}$ a la celda $C_{0,0}$ siguiendo el camino, esto asocia la fórmula actualizada. En otro caso, al menos, necesitamos guardar la matriz completa o por lo menos una área alrededor de la diagonal principal.

Apéndice B

Cola de prioridad

Los heap's son una estructura de datos [Sed92] que se usa para soportar operaciones con colas de prioridad guardando los registros en un arreglo de tal manera que cada llave esta garantizada por ser mayor que las llaves que están en otras dos posiciones específicas. Este ordenamiento es muy fácil ver si dibujamos el arreglo en un árbol de dos dimensiones con líneas abajo de cada llave a las dos llaves correspondientes. Obsérvese la figura B.1. La representación del *heap* o cola de prioridad como arreglo puede verse en la siguiente tabla B.1.

Esta representación es útil porque es muy fácil tomar el nodo padre y sus hijos. Los padres de un nodo en una posición j están en posición $j/2$ y al contrario. Los dos hijos del nodo en la posición j están en la posición $2j$ y $2j+1$. Una cola de prioridad es un árbol binario completo, representado como un arreglo, en el cual cada nodo satisface la condición de la cola de prioridad. En particular, la llave más grande esta siempre en la primera posición del arreglo. La ventaja de utilizar esta estructura de datos, es que todas las operaciones con esta cola de prioridad pueden ser terminadas

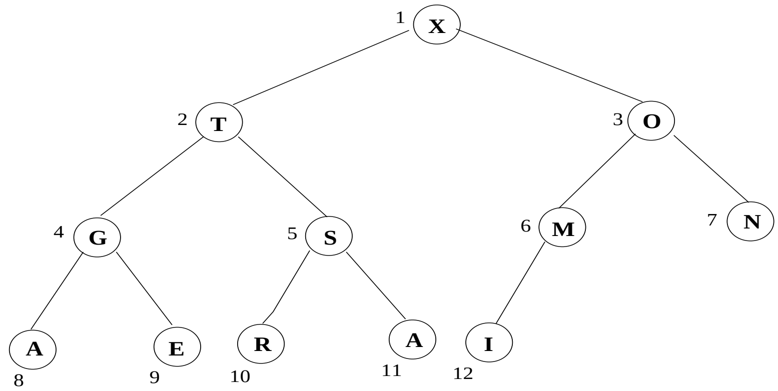


Figura B.1: Representación completa de la cola de prioridad como árbol

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	**X**	**T**	**O**	**G**	**S**	**M**	**N**	**A**	**E**	**R**	**A**	**I**

Tabla B.1: Representación de una cola de prioridad como un arreglo

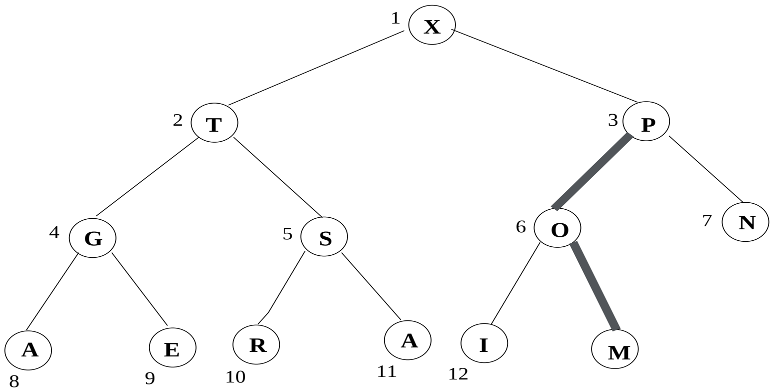


Figura B.2: Insertando un nuevo elemento (P) en la cola de prioridad

en tiempo logarítmico.

B.1 Implementación de una cola de prioridad

En esta sección se presenta el código para las operaciones básicas de la cola de prioridad las cuales son insertar, borrar.

La operación de la cola de prioridad *insertar* agrega elementos a la cola de prioridad, manteniendo la condición de la cola de prioridad en todos los lugares. Esta operación incrementan en uno el tamaño de la cola de prioridad (N debe ser incrementado). El registro a ser insertado es puesto en $a[N]$, pero esto podría violar la propiedad de la cola de prioridad. Si la propiedad de la cola de prioridad es violada (el nuevo nodo es mayor que sus padres), entonces la violación puede ser terminada intercambiando el nuevo nodo con su padre. Esto puede, en su momento, causar una violación, y esto puede ser solucionado de la misma manera. Por ejemplo, si P es insertado en la cola de prioridad de la figura B.1, esto es primero guardado en $a[N]$ como el hijo derecho de M. Entonces, como P es mayor que M, se intercambia con este, y como P es mayor que O, también se intercambia con este, y el proceso termina porque P es menor que X. La cola de prioridad resultante se muestra en la figura B.2.

El código para el método *insertar* es hacia arriba. En la siguiente implementación, *insertar* agregamos un nuevo elemento a $a[N]$, entonces llamamos a $upheap(N)$ para

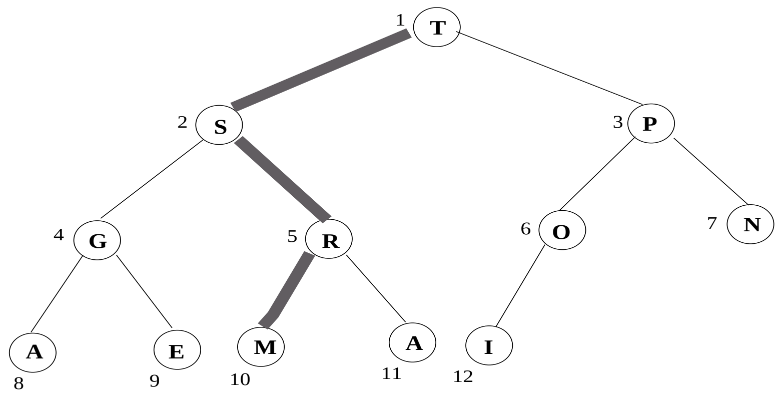


Figura B.3: Removiendo el elemento más grande en la cola de prioridad

romper las violaciones a la condición de la cola de prioridad.

```
upheap(int k)
{
        int v;
        v = a[k]; a[0] = INTMAX;
        while (a[k/2] ≤ v)
        {
               a[k] = a[k/2]; k = k/2;
        }
        a[k] = v;
}
insertar(int v)
{
        a[++N] = v;
        upheap(N);
}
```

Una llave centinela debe ser puesta en $a[0]$ para detener el ciclo para el caso que V es mayor que todas las llaves en la cola de prioridad.

La operación *remover* el más grande envuelve casi el mismo proceso, la figura B.3 muestra el resultado de remover X de la cola de prioridad, como puede observarse, el último elemento de la cola de prioridad se coloca en el lugar del primer elemento, es decir M se coloca en lugar de X, y ahora se hace un recorrido hacia abajo hasta que todos los elementos de la cola de prioridad cumplan con la propiedad de este.

El código para realizar esta operación se muestra a continuación, como puede observarse, ahora el proceso debe ser hacia abajo, y N debe disminuirse en 1.

```
downheap(int k)
{
        int j,v;
        v = a[k];
        while ( k ≤ N/2)
        {
                j = k + k;
                if(j<N && a[j] < a[j+1]) j++;
                if(v ≥ a[j]) break;
                a[k] = a[j]; k = j;
        }
        a[k] = v;
}
remover()
{
        int v = a[1];
        a[1] = a[N--];
        downheap(N);
        return v;
}
```

B.2 Implementación de dos colas de prioridad

En esta sección se presenta una implementación de una doble cola de prioridad [MDAS98]. El propósito de esta estructura, llamada Min-Max heap, es que puede ser construida en tiempo lineal, en contraste con las colas de prioridad convencionales, este permite encontrar el elemento mínimo y el elemento máximo, las operaciones de *insertar, remover mínimo, remover el máximo, encontrar mínimo, encontrar máximo* pueden ser desarrolladas en tiempo logarítmico.

Esta cola de prioridad se define como sigue. Dado un conjunto S de valores, un cola de prioridad min-max sobre S es un árbol binario T con las siguientes propiedades.

1. T tiene la forma de una cola de prioridad

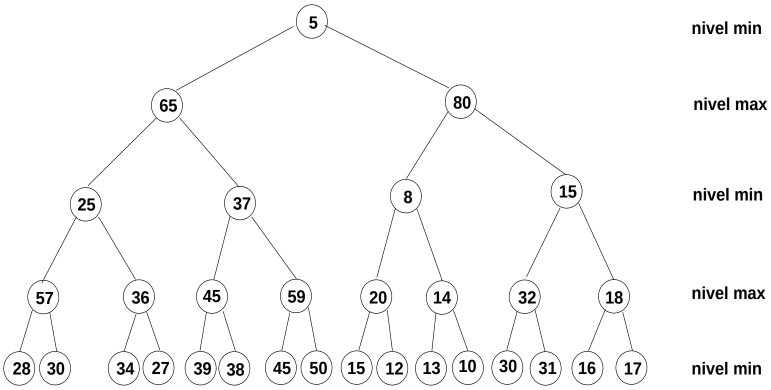


Figura B.4: Ejemplo de una cola de prioridad min-max. La condición de la cola de prioridad alterna entre el mínimo y máximo de nivel a nivel

2. T esta ordenado min-max : los valores guardados en los nodos en los niveles pares (pueden ser los nones) son menores que los valores guardados en sus descendientes, donde la raíz es el nivel cero. Entonces el valor menor es guardado en uno de los hijos raíz, un ejemplo de una cola de prioridad min-max se muestra en la figura B.4.

Una cola de prioridad min-max de N elementos pueden ser ordenados en un arreglo $A[1...N]$. La localidad ith en el arreglo puede corresponder a un nodo localizado en el nivel $\lfloor(log_2\ i)\rfloor$ en la cola de prioridad. Un heap min-max es definido análogamente; como una cola de prioridad, el valor máximo esta guardado en la raíz, y el valor mínimo esta guardado en uno de los hijos de la raíz.

Es interesante observar el diagrama de Hasse para una cola de prioridad min-max el cual mucho más complejo en contraste con una cola de prioridad tradicional. La figura B.5 muestra el diagrama de Hasse para la figura B.4.

El procedimiento para el algoritmo de la cola de prioridad min-max es muy similar a su correspondiente cola de prioridad convencional. La cola de prioridad tradicional es construido hacia arriba y hacia abajo. Cuando el algoritmo examina el subárbol de $A[i]$ entonces ambos subárboles de $A[i]$ son max-ordenados, mientras que el subárbol por sí mismo no puede no ser necesariamente ordenado. El paso Trickle-Down de este algoritmo intercambia los valores de $A[i]$ con el máximo de sus hijos. Este paso es entonces aplicado recursivamente a su hijo máximo para mantener el max-orden. En la cola de prioridad min-max, el orden requerido debe ser establecido entre un elemento sus hijos y su nietos. El procedimiento diferencia entre los niveles min y max. La descripción de este procedimiento es el siguiente.

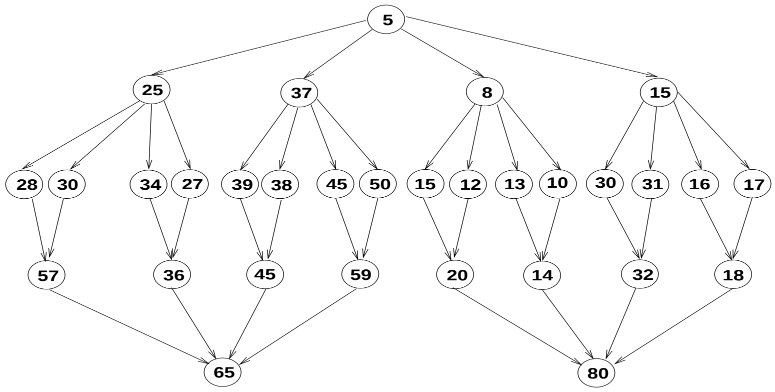


Figura B.5: Diagrama de Hasse para la cola de prioridad min-max de la figura B.4

```
procedure TrickleDown(i)
    i es la posicion en el arreglo
    if i esta en un nivel min then
        TrickleDownMin(i)
    else
        TrickleDownMax(i)
    endif

procedure TrickleDownMin(i)
    if A[i] tiene hijos then
        m:= indice del mas pequeño de los hijos y los nietos (si hay) de A[i]
        if A[m] es un nieto de A[i] then
            if A[m] < A[i] then
                swap A[i] y A[m]
                if A[m] > A[parent(m)] then
                    swap A[m] y A[parent(m)]
                endif
                TrickleDownMin(m)
            endif
        else { A[m] es un hijo de A[i] }
            if A[m] < A[i] then
                swap A[i] y A[m]
            endif
    endif
```

El procedimiento TrickleDownMax es el mismo excepto que los operadores relacionales están invertidos. Las operaciones de DeleteMin y DeleteMax son análogas a borrar en una cola de prioridad convencional. Específicamente, los elementos requeridos son extraídos y la posición vacante es llenada con el ultimo elemento de la cola de prioridad. El orden min-max es mantenido después de aplicar el procedimiento TrickleDown.

Apéndice C

Un modelo de complejidad

Los algoritmos de indexamiento basados en pivotes son construídos para minimizar el número de cálculos de distancia. Este problema tiene un gran número de aplicaciones, y un caso particular bien conocido es el de los espacios vectoriales, donde los objetos son tuplas l-arias de números reales (vector l-dimensional). El problema de búsqueda es conocido por ser difícil a medida que la dimensión crece. Es sabido que el comportamiento de todos los algoritmos basados en pivotes están relacionados directamente con la dimensión intrínseca [CN99].

C.1 Algoritmos de pivotes

Una explicación intuitiva del problema del porque el problema se más difícil a medida que la dimensión crece esta dada en la sección 6.3.1, en esta sección se dará una medida para cuantificar la dimensión intrínseca [CN99].

Definición: La dimensión intrínseca de un espacio métrico esta definida como $\rho = \frac{\mu^2}{2\sigma^2}$, donde μ y σ^2 son la media y la varianza del histograma de distancias correspondiente.

Obsérvese que la dimensión intrínseca crece con la media y con la inversa de la varianza de un histograma de distancias.

Si se pretende conocer el límite inferior de los algoritmos que usan pivotes, para conocer el desempeño del algoritmo, necesitamos relacionar la dimensión intrínseca con la dificultad de la búsqueda con un radio r dado.

Sea $(q, r)_d$ una consulta de rango sobre un espacio métrico indexado por medio de n_p pivotes aleatorios y sea u un elemento de U. La probabilidad de que u no pueda ser excluída desde una verificación directa después de considerar los n_p pivotes es exactamente.

$$Pr(\mid d(q,p_1) - d(u,p_1 \mid \leq r, ..., \mid d(q,p_{n_p}) - d(u,p_{n_p}) \mid \leq r) \tag{C.1}$$

Puesto que todos los pivotes es asumido que son aleatorios y sus distribuciones de distancia variables aleatorias, esta expresión es el producto de probabilidades.

$$Pr(\mid d(q,p_1) - d(u,p_1 \mid \leq r) \times ... \times Pr(\mid d(q,p_{n_p}) - d(u,p_{n_p}) \mid \leq r) \tag{C.2}$$

las cuales, por la misma razón puede ser simplificado a

$Pr(u \text{ no descartado}) = Pr(\mid d(q,p) - d(u,p) \mid \leq r)^{n_p}$

para un pivote aleatorio p.

Si X y Y son dos variables aleatorias con media μ y varianza σ^2, entonces la media de $X - Y$ es 0, y su varianza es $2\sigma^2$. Usando la desigualdad de Chebyschev's tenemos que $Pr(\mid X - Y \mid > \varepsilon) < 2\sigma^2\varepsilon^2$. por lo tanto,

$Pr(u \text{ no descartados }) \leq \left(1 - \frac{2\sigma^2}{r^2}\right)^{n_p}$

Ahora tenemos que el costo total de la búsqueda es le número de cálculos de distancia internas (n_p) más los cálculos externos (aquellos a checar como candidatos), aquellos números es un promedio de $n \times Pr(unodescartados)$. Por lo tanto,

$$costo \geq n_p + n\left(1 - \frac{2\sigma^2}{r^2}\right)^2 \tag{C.3}$$

es un límite inferior del costo promedio de las búsquedas usando pivotes. Optimizando podemos obtener que el mejor número de pivotes es:

$$n_p = \frac{\ln n + \ln\ln(1/t)}{\ln(1/t)} \tag{C.4}$$

donde $t = 1 - 2\sigma^2/r^2$. Usando este número de pivotes n_p, obtenemos un límite inferior absoluto para el costo promedio de cualquier algoritmo basado en pivotes aleatorios.

$$costo \geq \frac{\ln n + \ln\ln(1/t)}{\ln(1/t)} \geq \frac{\ln n}{\ln(1/t)} \geq \frac{r^2}{2\sigma^2}\ln n \tag{C.5}$$

esto muestra que el costo depende fuertemente de σ/r. Si r incrementa t tiende a 1 y el esquema requiere más y más pivotes y por cualquier lado es mas y mas costoso.

Índice de Figuras

Índice de Tablas

Bibliografía

[AGG] G. Nuñez A. García, G. L. Martínez and A. Guzmán. Página www. `http://148.204.20.3/artmin/kd/REPOR_TE.htm`.

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

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

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. 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 *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[BYCMW94] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[CFG99] E. Chávez, K. Figueroa, and M. García. Una estructura dinámica para búsqueda de similaridad. *Encuentro nacional de computación*, 1999.

[Chá97] E. Chávez. Algoritmos para detección de cúmulos en nubes de datos. *4a. Jornadas Iberoamericanas de Informatica.*, 1997.

[Che] P. Chew. Página www. `http://www.cs.corNell.edu/Info/People/-chew/Delaunay.html`.

[Chi95] L. Childs. *A Concrete Introduction to Higher Algebra*. Springer-Verlag, 1995.

[Cla83] K.L. Clarkson. Fast algorithm for the all neares neighbors problem, 1983.

[CMN99] E. Chávez, J. Marroquín, and G. Navarro. Overcoming the curse of dimensionality. Technical report, Universidad Michoacana, México, 1999. Submitted to the *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*. `ftp://garota.fismat.umich.mx/pub/users/-elchavez/fqa.ps.gz`.

[CN99] E. Chávez and G. Navarro. Measuring the dimensionality of general metric spaces. Technical report, Dept. of Computer Science, Univ. of Chile, 1999.

[CNBYM99] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. Technical Report TR/DCC-99-3, Dept. of Computer Science, Univ. of Chile, 1999. `ftp://ftp.dcc.uchile.cl/pub/-users/gnavarro/survmetric.ps.gz`.

[MB96] A.J. Quiroz M.R. Brito, E. Chávez. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & probability y letters*, 1996.

[MDAS98] N. Santoro M. D. Atkinson, J-R. Sack and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10), 1998.

[Nav99a] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, Cancun, Mexico, September 1999. To appear. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/metric.ps.gz`.

[Nav99b] G. Navarro. Searching in metric spaces by spatial approximation. *String processing and Information Retiveral*, 1999.

[NBY00] G. Navarro. and R. Baeza-Yates. A guided tour to approximates string matching. *ACM Computing Surveys*, 2000. To appear.

[RI] Dimacs Research and Education Institute. Página www. `http://-dimacs.rutgers.edu/drei/96/classroom/voronoi/intro.html`.

[Sed92] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1992.

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

[USC] USC. Página www. `http://neuroscholar.usc.edu/-documentation/nmds_analyses.htm`.

[VAnl89] P. Vaidya, O. An, and n log. algorithm for the all-nearest-neighbor problem, 1989.

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

[Yaz] M. Yazdani. Página www. `http://www.media.uwe.ac.uk/masoud/-course/hci/papers/ojeda/reading.htm`.

[yEC00] H. Tejeda y E. Chávez. Una gráfica de proximidad para búsqueda de similaridad. Technical report, Facultad de Ing. Eléctrica, 2000.

[Yia93] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 311–321, 1993.