Universidad Nacional de San Luis
Facultad de Ciencias Físico-Matemáticas y Naturales
Departamento de Informática

Índices Dinámicos para Espacios Métricos de Alta Dimensionalidad

Nora Susana Reyes

Asesor Científico: ***Dr. Gonzalo Navarro***

San Luis - Argentina

Diciembre, 2002

Universidad Nacional de San Luis
Facultad de Ciencias Físico-Matemáticas y Naturales
Departamento de Informática

Índices Dinámicos para Espacios Métricos de Alta Dimensionalidad

Nora Susana Reyes

Asesor Científico	:	**Dr. Gonzalo Navarro**
Profesores del Tribunal	:	Dr. Edgar Chávez
	:	Dr. Raúl Gallard

Tesis para optar al Grado de
Magister en Ciencias de la Computación

Este trabajo ha sido parcialmente financiado por el proyecto CYTED VII.19 RIBIDI y por el Proyecto FOMEC del Departamento de Informática de la U.N.S.L.

San Luis - Argentina

Diciembre, 2002

Agradecimientos

Terminar esta tesis ha sido, hasta el momento, para mí el trabajo más importante y de mayor envergadura que haya encarado, pero no es el final de una etapa sino más bien el comienzo de la que considero será la etapa más importante de mi formación académica.

Durante este último tiempo conseguí finalmente encontrar un área en la que me gusta investigar y aprender. Ha sido también una época difícil, pero en la que he conseguido varios logros personales y además algunas recompensas importantes.

Por lo tanto, estoy muy agradecida con toda la gente que lo hizo posible, por su desinteresada ayuda, amistad y paciencia. De todos modos es posible que me olvide de algunos nombres y espero que si es así me sepan perdonar por ello.

Antes que a nadie quiero agradecer a mi esposo Guillermo y a mis hijos, Nora, Rocío y Sebastián no sólo por permitirme realizar esta tesis, aún a costa de quitarnos tiempo juntos, sino también porque sin ellos mi vida estaría vacía y mis logros académicos carecerían de sentido.

Quiero agradecer especialmente a mi director de tesis Gonzalo Navarro, por creer en mí aún sin conocerme, por haberme brindado todo lo que estuvo a su alcance y que el creyó que yo aprovecharía, y además por su excelente trabajo como mi director, en algunos casos como mi psicólogo, en otros como mi profesor y hasta en otros como mi amigo. Hago extensivo en este caso mi agradecimiento también a Betina (su esposa) por brindarme cálidamente un lugarcito junto a su familia, cuando la mía estaba lejos.

No puedo dejar de agradecer, por supuesto, a mi familia que siempre me apoyó y creyó en mí, aún sin entender demasiado mi trabajo.

Mucha gente, entre las que me incluyo, pensaba en que iba a ser difícil realizar una tesis a distancia y más teniendo familia numerosa (con Sebastián en camino); pero realmente descubrí que cuando uno trabaja en lo que realmente le gusta y además con un “buen” director (no sólo en lo académico sino también en lo personal) se consigue lograr lo que antes se creía imposible.

Quiero agradecer muy especialmente a Ricardo Baeza-Yates, porque si no hubiera sido por él, y por su apoyo, no hubiese conocido en la Universidad de Chile a quien es actualmente mi guía, Gonzalo.

También quisiera agradecer a quienes me ayudaron y apoyaron siempre que pudieron (mis amigas y compañeras de trabajo) Patricia, Norma, Oli, Fabiana, Verónica y Teté, y a quienes me formaron en esta disciplina (mis profesores, jefes y alumnos).

Con Norma nos animamos a empezar esta etapa juntas y nos acompañamos mutuamente, escuchándonos y sugiriéndonos ideas una a la otra, cuando parecía que nuestras neuronas se desconectaban o entraban en crisis; por lo cual ella tiene también mi mayor agradecimiento y mi reconocimiento a su capacidad.

A Marcela y Susana porque desde su puesto de Directoras del Departamento, cada una de ellas me facilitó que empezara y culminara esta etapa.

A los miembros del tribunal Raúl Gallard y Edgar Chávez por su paciencia y sugerencias para mejorar este trabajo.

A Hugo quien constantemente me marcó las ventajas de aprender y formarse en aquello en que uno espera poder enseñar.

Durante este tiempo he conocido mucha gente a quien ahora quiero recordar, porque me brindaron

una mano. Benjamín, quien me brindó desinteresadamente su ayuda cuando se lo solicité. A la gente de mi Departamento que me ayudó siempre que se lo solicité (Beto, Javier, Daniel y Daniel). A la gente del Departamento de Ciencias de la Computación de la Universidad de Chile que me recibió muy bien durante mis cortas, pero productivas, estancias.

Finalmente quiero agradecer el apoyo financiero parcial para este trabajo del proyecto CYTED VII.19 RIBIDI y de la Universidad Nacional de San Luis, por medio del proyecto FOMEC del Dpto. de Informática y por medio del proyecto de Base de Datos y Teoría de Modelos.

Quiero agradecer también muy especialmente a Rosa (mi suegra), Ramona y Marisa por cuidar de mi familia cuando he necesitado su ayuda, con mucho cariño y paciencia.

Nora Reyes

San Luis, diciembre de 2002

A mis hijos y a mi marido por su paciencia y amor,
sin los que nada en mi vida tendría sentido.

A mis padres por hacer de mí lo que soy
y siempre estar a mi lado.

Índices Dinámicos para Espacios Métricos de Alta Dimensionalidad

Muchas aplicaciones computacionales necesitan buscar información en una base de datos. Tradicionalmente la operación de búsqueda se ha aplicado a *datos estructurados* y las bases de datos tradicionales se construyen alrededor del concepto de búsqueda exacta. En la actualidad han surgido depósitos no estructurados de información y no sólo se consultan nuevos tipos de datos (texto libre, imágenes, audio y video), sino que además ya no es posible estructurar la información de la manera clásica y, aún cuando sea posible hacerlo, nuevas aplicaciones tales como minería de datos (*data mining*) requieren acceder a la base de datos por cualquier campo, no sólo aquellos marcados como "claves". Por lo tanto, se necesitan nuevos modelos y algoritmos de búsqueda más generales.

Un concepto unificador es el de "búsqueda por similitud" o "búsqueda por proximidad", es decir buscar elementos de la base de datos que sean similares a un elemento de consulta dado. La similaridad es modelizada usando una función de distancia (o métrica) que satisface ciertas propiedades y el conjunto de objetos es llamado un *espacio métrico*. En general, como la distancia es bastante costosa de calcular, el objetivo es reducir el número de evaluaciones de distancia.

Uno de los principales obstáculos para el diseño de técnicas de búsqueda eficientes en espacios métricos es la existencia en aplicaciones reales de los así llamados "espacios de alta dimensionalidad". Las técnicas tradicionales de indexación en su mayoría no son eficientes en espacios de alta dimensión. La búsqueda por proximidad en espacios métricos se torna intrínsecamente más difícil mientras mayor sea la dimensión intrínseca del espacio; este hecho es conocido como la *maldición de la dimensionalidad*.

Existen numerosos métodos para preprocesar un conjunto a fin de reducir el número de evaluaciones de distancia en tiempo de consulta. Todos ellos se basan en dividir la base de datos, lo que se ha heredado de las ideas clásicas de *dividir para conquistar* y de la búsqueda de datos típica. El Árbol de Aproximación Espacial *SA-tree* [Nav99, Nav02] en cambio es una nueva estructura de datos, propuesta recientemente, que es específica de la búsqueda espacial. Más que dividir el conjunto de candidatos durante la búsqueda, se trata de "acercarse espacialmente" a la query q. El *SA-tree* ha demostrado ser muy competitivo en espacios métricos de alta o media dimensionalidad (espacios "difíciles") o para responder a consultas con baja selectividad, pero su principal desventaja es que era una estructura de datos totalmente estática. Por lo tanto, esto la convertía en poco útil para muchas de las aplicaciones reales. Así, nuestro interés fue el de producir una versión *dinámica* del *SA-tree*, con el fin de aprovechar sus bondades y salvar su principal desventaja.

Hemos obtenido un algoritmo de inserción eficiente que nos permite construir incrementalmente el árbol con costos que *mejoran* ampliamente los costos del *SA-tree* original y algoritmos de eliminación o borrado eficientes, aspecto no muy común en las estructuras de datos existentes para búsqueda en espacios métricos. Además logramos mantener la búsqueda eficiente y en algunos casos aún mejoramos su desempeño respecto de la versión original, esto último especialmente en dimensiones bajas. Agregamos un parámetro a la estructura, que es fácil de sintonizar, y que nos permite adaptarla mejor a la dimensión intrínseca del espacio métrico considerado y otro que permite sintonizar costo de búsqueda versus costo de eliminación. El *SA-tree* original no tenía un buen desempeño en espacios métricos de baja dimensión, lo que también hemos logrado superar con nuestro *SA-tree* dinámico, haciéndolo más competitivo en esta área. Nuestro estudio nos ha permitido obtener una visión más profunda de la estructura misma, y así descubrir la posibilidad de relajar algunas condiciones, que se planteaban como necesarias para el *SA-tree* original, manteniendo la correctitud de la estructura.

Creemos que este trabajo constituye un aporte valioso al desarrollo y comprensión del problema de búsqueda en espacios métricos, además de proveer estructura para búsqueda por proximidad muy competitiva y totalmente dinámica, tanto en espacios métricos de alta como en los de baja dimensión.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Muchas aplicaciones computacionales necesitan buscar información en una base de datos. Aplicaciones simples presentan problemas de búsqueda simple, mientras que aplicaciones más complejas requerirán, en general, una forma más sofisticada de búsqueda.

Tradicionalmente la operación de búsqueda se ha aplicado a *datos estructurados*, es decir búsqueda exacta sobre información numérica o alfabética. O sea, dada una consulta o query de búsqueda se recupera el número o cadena de caracteres (*strings*) que es exactamente igual a la consulta. Las bases de datos tradicionales se construyen alrededor del concepto de búsqueda exacta: la base de datos se divide en registros y cada registro posee una clave completamente comparable.

Así, las consultas a la base de datos retornan todos los registros cuyas claves coinciden con la clave de búsqueda. Las búsquedas más sofisticadas tales como consultas por rango sobre claves numéricas o búsqueda de prefijos sobre claves alfabéticas también se basan en el concepto que dos claves son o no iguales, o que existe un orden total sobre las claves. Aunque, en estos últimos años, las bases de datos ya incluyen la capacidad de almacenar nuevos tipos de datos tales como imágenes, la búsqueda todavía se realiza sobre un número predeterminado de claves de tipos numérico o alfabético.

Con la evolución de las tecnologías de información y comunicación, han surgido depósitos no estructurados de información. No sólo se consultan nuevos tipos de datos tales como texto libre, imágenes, audio y video, sino que además ya no es posible estructurar la información en claves y registros. Tal estructuración es muy dificultosa (tanto manual como computacionalmente) y restringe de antemano los tipos de consultas que luego se pueden realizar. Aún cuando sea posible una estructuración clásica, nuevas aplicaciones tales como minería de datos (*data mining*) requieren acceder a la base de datos por cualquier campo, no sólo aquellos marcados como “claves”. Por lo tanto, se necesitan nuevos modelos para buscar en depósitos o almacenamientos de datos no estructurados.

Los escenarios anteriores requieren algoritmos de búsqueda y modelos más generales que aquellos que se usan clásicamente para datos simples. Un concepto unificador es el de “búsqueda por similitud” o “búsqueda por proximidad”, es decir buscar elementos de la base de datos que sean similares o próximos a un elemento de consulta dado. La similaridad es modelizada usando una función de distancia (o métrica) que satisface las propiedades de *desigualdad triangular*, *positividad estricta* y *simetría*, y el conjunto de objetos es llamado un *espacio métrico*.

En algunas aplicaciones los espacios métricos resultan ser de un tipo particular llamado “espacio vectorial”, donde los elementos consisten de D coordenadas de valores reales. Existen muchos trabajos que

aprovechan las propiedades geométricas sobre espacios vectoriales, pero normalmente éstos no se pueden extender a los espacios métricos generales donde la única información disponible es la distancia entre objetos. En este caso general, la distancia es bastante costosa de calcular, así que el objetivo general es reducir el número de evaluaciones de distancia. En contraste, las operaciones en espacios vectoriales tienden a ser simples y por lo tanto, el objetivo principal es reducir la entrada/salida (E/S).

Existen dos tipos de consultas típicas en búsqueda por proximidad, ellos son:

- *Consulta por Rango*: se desea recuperar los elementos de la base de datos que se encuentren a una distancia no mayor que un cierto radio de tolerancia de un elemento de consulta (*query*) dado.
- *k-vecinos más cercanos*: se desea recuperar los k elementos que se encuentren más cerca de una query dada.

Una manera trivial de responder ambos tipos de consulta es realizando una búsqueda exhaustiva en la base de datos, es decir, comparando todos los elementos de la base de datos contra el elemento consultado y retornando aquellos elementos que se encuentren suficientemente cerca de éste; pero, por lo general, esto resulta demasiado costoso para aplicaciones reales.

Se han logrado algunos importantes avances para buscar en espacios métricos generales, en su gran mayoría alrededor de la idea de construir un *índice*, es decir una estructura de datos que reduzca el número de evaluaciones de distancia en tiempo de consulta. Algunos trabajos recientes [CPZ97, PAA98] tratan de obtener al mismo tiempo los objetivos de reducir el número de evaluaciones de distancia y la cantidad de entrada/salida realizada.

En [CNBYM01] se presenta un marco de trabajo unificador para describir y analizar todas las soluciones existentes a este problema. Allí se muestra que todos los algoritmos de indexación existentes para búsqueda por proximidad consisten en construir clases de equivalencia, luego descartar algunas clases y buscar exhaustivamente en el resto. Los algoritmos de búsqueda por proximidad se pueden dividir en dos grandes áreas: *algoritmos basados en pivotes* y *algoritmos basados en particiones compactas*, las cuales abarcan todos los métodos existentes. También se presentan en [CNBYM01] métodos cuantitativos para estimar la dificultad intrínseca de buscar sobre un espacio métrico dado y se proveen cotas sobre el problema de búsqueda. Esto incluye una definición cuantitativa de la noción conceptual de “dimensionalidad intrínseca”, la cual resulta ser muy apropiada.

1.1. Motivación

A continuación presentamos una muestra de la amplia gama de aplicaciones en donde aparece el concepto de búsqueda por proximidad.

Consultas por contenido en bases de datos estructuradas

En general la consulta que se realiza a la base de datos aporta una *parte* de un registro de información, y necesita recuperar un registro *completo*. En la aproximación clásica, la parte aportada es fija (la *clave*). Mas aún, no se permite buscar con una clave incompleta o errónea. Por otro lado, en la aproximación más general usada en la actualidad el concepto de búsqueda con una clave es generalizado al de búsqueda con un subconjunto arbitrario del registro, permitiendo o no errores.

Los tipos de búsqueda posibles son: búsqueda de *claves* o *puntos* (se da toda la información de la clave), búsqueda de *rango* (sólo se dan algunos campos o sólo se especifica un rango de valores para ellos) y búsqueda de *proximidad* (los registros "cercanos" al consultado se consideran interesantes). Estos tipos de búsqueda son de utilidad en minería de datos (donde las partes de interés del registro no se conocen de antemano), cuando la información no es precisa, cuando buscamos un rango de valores, cuando la clave de búsqueda puede contener errores (por ejemplo, palabras mal deletreadas), etc.

Una solución general al problema de las consultas por rango por cualquier campo de registro es el *grid file* [NH84]. El dominio de la base de datos se ve como un hiperrectángulo de D dimensiones (una por campo de registro), donde cada dimensión tiene un orden de acuerdo al dominio del campo (numérico o alfabético). Cada registro presente en la base de datos se considera como un punto dentro del hiperrectángulo. Una consulta especifica un subrectángulo (es decir, un rango para cada dimensión), y se recuperan todos los puntos dentro de la consulta especificada. Esto no resuelve el problema de búsqueda sobre tipos de datos no tradicionales, ni permite errores que no pueden ser expresados por una consulta por rango. Sin embargo, convierte el problema de búsqueda original en el problema de obtener, en un espacio dado, todos los puntos "cercanos" a un punto de consulta dado.

Consulta por contenido en objetos multimedia

Nuevos tipos de datos tales como imágenes, huellas dactilares, audio y video (llamados tipos de datos "multimedia") no se pueden consultar significativamente en el sentido clásico, no sólo porque no pueden ser ordenados, sino porque ninguna aplicación estaría interesada en buscar un segmento de audio exactamente igual a uno dado. La probabilidad de que dos imágenes diferentes sean iguales pixel a pixel es insignificante a menos que sean copias digitales de la misma fuente. En aplicaciones multimedia, todas las consultas preguntan por objetos *similares* a uno dado. Algunos ejemplos de aplicaciones son reconocimiento de caras, correspondencia de huellas dactilares, reconocimiento de voz y en bases de datos multimedia generales.

Muchas aproximaciones se basan en la definición de una función de similaridad entre objetos, que debe ser provista por un experto, pero que no presuponen el tipo de consultas que se pueden responder. En muchos casos, la distancia es obtenida via un conjunto de D "características" que se extraen del objeto (por ejemplo: en una imagen una característica útil es el color promedio). Entonces, cada objeto se representa por sus D características, es decir un punto en un espacio vectorial de dimensión finita D (los llamaremos en adelante espacios vectoriales o espacios D-dimensionales), y nuevamente volvemos a un caso de consultas por rango sobre espacios vectoriales.

Recuperación de texto

La recuperación de texto no estructurado, aunque éste no sea considerado un tipo de datos multimedia, presenta problemas similares a la recuperación multimedia.

Esto se debe a que los documentos de texto en general no están estructurados para proveer fácilmente la información deseada. Se puede buscar en los documentos de texto por cadenas (strings) que están presentes o no, pero en muchos casos se busca en ellos por conceptos semánticos de interés. Por ejemplo, en un escenario ideal se permitiría buscar en un texto por un concepto tal como *"impedir el funcionamiento normal de algo"*, recuperando la palabra *"bloquear"*. Este problema de búsqueda no se puede solucionar apropiadamente usando las herramientas clásicas.

Existe actualmente un grupo de medidas de similaridad orientadas a resolver este problema [SM83, FBY92, BYRN99]. El problema se resuelve básicamente recuperando documentos similares a una consulta dada. El usuario puede aportar un documento como una consulta, para que el sistema encuentre documentos similares. Algunas aproximaciones se basan en transformar un documento en un

vector de valores reales, así cada dimensión es una palabra del vocabulario y la relevancia de la palabra en el documento (computada utilizando alguna fórmula) es la coordenada del documento para esa dimensión. Se definen funciones de similaridad en ese espacio. Sin embargo, la dimensionalidad del espacio es muy alta (miles de dimensiones).

Otro problema relacionado a la recuperación de texto es cómo se escriben las palabras. Como han surgido enormes bases de datos con bajo control de calidad (por ej. la Web) donde son comunes, en el texto y en la consulta, errores de tipeo, escritura o reconocimiento óptico de caracteres, no se pueden recuperar documentos que contienen palabras mal escritas realizando una consulta correctamente escrita. Existen modelos de similaridad entre palabras que capturan muy bien esa clase de errores. En este caso, se da una palabra y se quieren recuperar todas las palabras cercanas a ella. Otro ejemplo de aplicación son los revisores ortográficos, donde se buscan variantes cercanas a la palabra mal escrita.

Biología computacional

Los objetos básicos de estudio en biología molecular son el ADN y las secuencias de proteínas. Si éstos se pueden modelizar como textos, tenemos el problema de encontrar una secuencia dada de caracteres dentro de una secuencia más larga. Sin embargo, es improbable que ocurra una coincidencia exacta, y los biólogos están más interesados en encontrar partes de una secuencia más larga que son similares a una secuencia corta dada. El hecho que la búsqueda no sea exacta es debido a que diferencias menores en cadenas genéticas describen seres de la misma especie o de especies cercanamente relacionadas. La medida de similaridad utilizada está relacionada con la probabilidad de mutaciones tales como inversión de pedazos en la secuencia u otros reacomodamientos.

Reconocimiento de patrones y aproximación de funciones

Una definición simplificada del reconocimiento de patrones es la construcción de un aproximador de función. En este problema hay una muestra finita de los datos, y cada una está rotulada con la clase a la que pertenece. Cuando aparece una nueva muestra, se desea rotularla con uno de los rótulos de los datos conocidos. En otras palabras el clasificador se puede pensar como una función definida desde el espacio de objetos (datos) en el conjunto de rótulos.

Si los objetos son vectores $D-$dimensionales de números reales entonces una elección natural son las redes neuronales y los aproximadores de función difusos. Otro aproximador de función popular es el clasificador de los $k-$vecinos más cercanos, que consiste en encontrar los k objetos más cercanos a la muestra no rotulada, y asignarle a esta muestra el rótulo que posee la mayoría entre estos k objetos.

En general, en cualquier problema donde se quiera inferir una función basado en un conjunto finito de muestras es una aplicación potencial.

Compresión y transmisión de audio y video

La transmisión de audio y video es un problema importante, por ejemplo en conferencias de audio y video por Internet o en una comunicación inalámbrica. Un cuadro (una figura estática en un video o un fragmento del audio) se puede pensar como formado por un número de “subcuadros” (posiblemente superpuestos). El problema se puede resolver enviando el primer cuadro completo y para los próximos cuadros enviar sólo los subcuadros que tienen una diferencia significativa con respecto a los previamente enviados. Esta descripción abarca al MPEG estándar.

Los algoritmos de hecho usan un buffer de subcuadros. Cada vez que se va a enviar un cuadro se lo busca en el buffer de subcuadros (con una tolerancia) y si no se lo encuentra se agrega el subcuadro completo al buffer. Si el subcuadro se encuentra, entonces sólo se envía el índice del cuadro similar encontrado. Esto implica, naturalmente, que hay que incorporar al servidor un algoritmo de

búsqueda por similaridad rápido para mantener un mínimo de velocidad de transferencia de cuadros por segundo.

1.2. Relevancia y Aportes de la Tesis

Todas las aplicaciones anteriores se pueden modelizar suponiendo que los objetos son cajas negras, y que la única forma de compararlos es calculando la distancia entre ellos.

Uno de los principales obstáculos para el diseño de técnicas de búsqueda eficientes en espacios métricos es la existencia en aplicaciones reales de los así llamados espacios de alta dimensionalidad. Las técnicas tradicionales de indexación en su mayoría no son eficientes en espacios de alta dimensión.

Esto no es casual. La búsqueda por proximidad en espacios métricos se torna intrínsecamente más difícil mientras mayor sea la dimensión intrínseca del espacio [Yia93, Bri95, CM97, CNBYM01]. Este hecho es conocido como la *maldición de la dimensionalidad*.

El concepto de "dimensionalidad" está relacionado a la "facilidad" o "dificultad" de buscar en el espacio: espacios de alta dimensión poseen una distribución de probabilidad de las distancias entre elementos cuyo histograma es más concentrado y tiene una media mayor. Esto provoca que el trabajo a realizar por cualquier algoritmo de búsqueda por similaridad sea dificultoso. En el caso extremo podemos tener un espacio donde $d(x,x) = 0$ y $\forall y \neq x,\ d(x,y) = 1$, donde cualquier consulta se deba resolver comparando exhaustivamente la query contra todos los objetos de la base de datos.

La dimensión intrínseca de un espacio métrico se define en [CNBYM01, CN00b, CN01] como $\mu^2/2\sigma^2$, donde μ y σ^2 son la media y la varianza del histograma de distancias del espacio métrico. Esta definición es coherente con la noción de dimensión en un espacio vectorial con coordenadas uniformemente distribuidas. La idea es que, a medida que crece la dimensionalidad intrínseca del espacio, la media del histograma μ crece y su varianza σ^2 se reduce.

Existen numerosos métodos para preprocesar un conjunto a fin de reducir el número de evaluaciones de distancia. Todos ellos se basan en dividir la base de datos, lo que se ha heredado de las ideas clásicas de *dividir para conquistar* y de la búsqueda de datos típicos (v.g. árboles de búsqueda binaria).

El Árbol de Aproximación Espacial *SA-tree* [Nav99, Nav02] en cambio es una nueva estructura de datos, propuesta recientemente, que es específica de la búsqueda espacial. Más que dividir el conjunto de candidatos durante la búsqueda, se trata de comenzar la búsqueda en algún punto del espacio y acercarse espacialmente a la query q, encontrando elementos cada vez más cercanos a ella. El *SA-tree* ha demostrado ser muy competitivo en espacios métricos de alta o media dimensionalidad (espacios "difíciles") o para responder a consultas con baja selectividad.

La principal desventaja del *SA-tree* es que era una estructura de datos totalmente estática, es decir para su construcción se debían conocer de antemano todos los elementos de la base de datos. Por lo tanto, esto la convertía en poco útil para muchas de las aplicaciones reales mencionadas previamente.

Así, nuestro interés fue el de producir una versión *dinámica* del *SA-tree*, con el fin de aprovechar sus bondades y salvar su principal desventaja.

El problema con el que nos encontramos en un comienzo es que la concepción original del *SA-tree* es intrínsecamente estática. La aproximación espacial se realiza vía los "vecinos". El árbol mantiene para cada elemento a de la base de datos un conjunto $N(a)$ de "vecinos". Los vecinos de a son aquellos elementos que

se encuentran más cerca de él que de ningún otro elemento de la base de datos. Por lo tanto, al incorporar un nuevo elemento x se lo debe agregar como vecino de algún elemento existente a y se debe reconstruir todo el subárbol de a. Esto se debe a que las vecindades pueden cambiar al estar ahora presente x en el espacio, y entonces se deben recalcular para mantener que cada elemento pertenezca a la vecindad de su elemento más cercano. El problema dual de eliminar un elemento x presenta similares desafíos.

Por todo lo expresado anteriormente consideramos que los aportes más relevantes de esta tesis son

- Hemos obtenido distintos algoritmos de inserción eficientes para el *SA-tree*. En particular, el algoritmo propuesto finalmente nos permite construir incrementalmente el árbol con costos que *mejoran* ampliamente los costos del *SA-tree* original.
- Hemos logrado algoritmos de eliminación o borrado eficientes en un *SA-tree*, aspecto no muy común en las estructuras de datos para búsqueda en espacios métricos existentes.
- Logramos mantener la búsqueda eficiente y en algunos casos aún mejoramos su desempeño respecto de la versión original de *SA-tree*, esto último especialmente en dimensiones bajas.
- Conseguimos una visión más profunda de la estructura misma, lo que nos permitió descubrir la posibilidad de relajar algunas condiciones, que se planteaban como necesarias para el *SA-tree* original, manteniendo la correctitud de la estructura.
- Agregamos un parámetro a la estructura, que es fácil de sintonizar, y que nos permite adaptarla mejor a la dimensión intrínseca del espacio métrico considerado. Otro permite sintonizar costo de búsqueda versus costo de eliminación.
- El *SA-tree* original no tenía un buen desempeño en espacios métricos de baja dimensión, lo que también hemos logrado superar con nuestro *SA-tree* dinámico, haciéndolo más competitivo en esta área.
- Hemos conseguido publicar los principales resultados de nuestro trabajo en [Nav02, NR01, RHN01, HRBY$^+$02, NR02a, RN02, NR02b].

Aunque nuestros resultados son aplicables principalmente al *SA-tree*, algunos de ellos podrían ser aplicables a otras estructuras.

Creemos que este trabajo constituye un aporte valioso al desarrollo y comprensión del problema de búsqueda en espacios métricos, además de proveer estructura para búsqueda por proximidad muy competitiva y totalmente dinámica, tanto en espacios métricos de alta dimensión como en los de baja.

1.3. Organización de la Tesis

Hemos dividido esta presentación en dos partes, una primera parte que introduce todos los conceptos necesarios para entender la tesis y la segunda dedicada exclusivamente a describir nuestros aportes.

Primera Parte
En esta parte se introducen los conceptos necesarios para entender la tesis:

- En el Capítulo 2 se explican todos los conceptos básicos necesarios para leer esta tesis, como así también se presentan otros trabajos relacionados, que permiten poner en contexto nuestro aporte.
- En el Capítulo 3 se describe detalladamente el Árbol de Aproximación Espacial (*SA-tree*) en su versión original [Nav99, Nav02].

Segunda Parte
En esta parte se presenta la nueva estructura, junto con todo el análisis previo realizado hasta llegar a la versión definitiva de la misma.

- En el Capítulo 4 se explican las distintas alternativas experimentadas para realizar la construcción incremental del Árbol de Aproximación Espacial y la alternativa finalmente obtenida.
- En el Capítulo 5 analizamos distintas opciones para realizar eliminaciones y la posible combinación de métodos que permite cierta versatilidad respecto del balance deseado entre costo de eliminación y costo de búsqueda.
- En el Capítulo 6 se dan las conclusiones a las que hemos llegado, como así también hacia dónde planeamos continuar este trabajo en el futuro.
- En el Apendice A se muestran más resultados experimentales sobre el comportamiento de nuestro *SA-tree* dinámico para distintos espacios métricos, junto con un breve análisis de esos resultados. Las comparaciones se realizan contra la versión estática del *SA-tree* porque ella ya se había comparado contra otras estructuras en [Nav02] y había demostrado ser muy competitiva, sobre todo en espacios de alta dimensión o consultas de baja selectividad.

Capítulo 2

Conceptos y Trabajos Previos

Todas las aplicaciones presentadas en el Capítulo 1 comparten un marco de trabajo común, el cual es en esencia encontrar objetos cercanos, bajo alguna función de similaridad adecuada, entre un conjunto finito de elementos. En esta sección se presenta el modelo formal que comprende todos los casos anteriores.

2.1. Espacios Métricos

Se introduce ahora la notación básica para el problema de satisfacer consultas por proximidad. El conjunto U denotará el universo de objetos *válidos*. Un subconjunto finito de él, S, de tamaño $n = |S|$, es el conjunto de objetos donde se busca. S será llamado el *diccionario*, *base de datos* o simplemente nuestro conjunto de *objetos* o *elementos*.

La función

$$d : U \times U \to \mathbb{R}$$

denotará una medida de "distancia" entre objetos (es decir, mientras más pequeña es la distancia, más cercanos o similares son los objetos). Las funciones de distancia tienen las siguientes propiedades:

($p1$) $\forall x, y \in U, d(x, y) \geq 0$ positividad,

($p2$) $\forall x, y \in U, d(x, y) = d(y, x)$ simetría,

($p3$) $\forall x \in U, d(x, x) = 0$ reflexividad,

y en la mayoría de los casos

($p4$) $\forall x, y \in U, x \neq y \Rightarrow d(x, y) > 0$ positividad estricta.

Las propiedades enumeradas de la función de similaridad sólo aseguran su definición consistente y no pueden ser usadas para ahorrarse comparaciones en una consulta por proximidad. Si d es en verdad una *métrica*, es decir si satisface

($p5$) $\forall x, y, z \in U, d(x, y) \leq d(x, z) + d(z, y)$ desigualdad triangular,

entonces el par (U, d) se denomina un *espacio métrico*.

De aquí en más se utilizará el término *distancia* entendiendo que se hace referencia a una métrica.

2.2. Consultas por Proximidad

Existen dos tipos básicos de consultas de interés en espacios métricos:

Consulta por rango $(q, r)_d$**.** Recuperar todos los elementos que están a distancia r de q. Esto es $\{x \in S\,, d(q, x) \leq r\}$.

Consulta de $k-$**vecino más cercano** $k - NN(q)$**.** Recuperar los k elementos más cercanos a q en S. Esto es, recuperar un conjunto $A \subseteq S$ tal que $|A| = k$ y $\forall x \in A, y \in S - A, d(q, x) \leq d(q, y)$. Notar que nos satisface cualquier conjunto de k elementos que cumpla la condición.

El tipo más básico de consulta es la consulta por rango. La Figura 2.1 ilustra un ejemplo de ambos tipos de consulta. A la izquierda se muestra una consulta por rango con radio r y a la derecha una consulta por los 5-vecinos más cercanos a q. En este último caso, para hacer más evidente el tipo de consulta, se ha graficado también el rango necesario para encerrar al menos a 5 puntos (aunque éste realmente encierra a más de 5 objetos). Así es posible observar que existirían, para esta consulta (dado q y con $k = 5$), otras posibles respuestas. Las consultas se realizan sobre un conjunto de puntos de $\mathbb{R}^2$, como el espacio métrico, para mayor claridad.

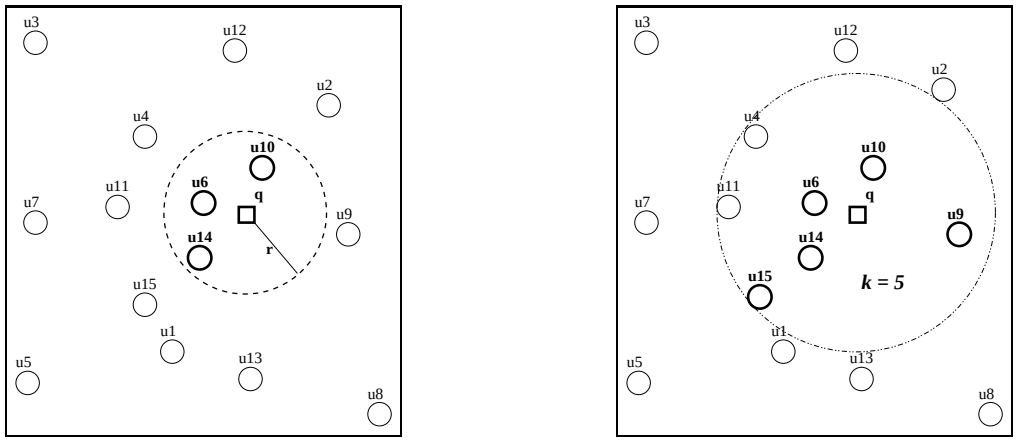


Figura 2.1: Un ejemplo de consulta por rango (izquierda) y por k-vecinos más cercanos (derecha) sobre un conjunto de puntos de $\mathbb{R}^2$.

Una *consulta por rango* será por lo tanto un par $(q, r)_d$ siendo q un elemento de U y r un número real indicando el *radio* (o tolerancia) de la consulta. El conjunto $\{u \in S, d(q, u) \leq r\}$ será llamado la *salida* o *respuesta* de la consulta por rango.

Se usa "NN" como una abreviatura de "Nearest Neighbor" (vecino más cercano), y se le da el nombre genérico de "consulta-NN" al último tipo de consulta y "búsquedas-NN" a las técnicas utilizadas para resolverlas. Como se verá más adelante, las consultas-NN se pueden construir sistemáticamente sobre consultas por rango.

El tiempo total para evaluar una consulta se puede dividir en

$$T = \#\text{ evaluaciones de distancia } \times \text{ complejidad de } d() + \text{ tiempo extra de CPU } + \text{ tiempo de E/S}$$

y desearíamos minimizar T. En muchas aplicaciones, sin embargo, evaluar $d()$ es tan costoso que todas las demás componentes del costo se pueden ignorar. Éste es el modelo utilizado en el presente trabajo, y por lo tanto la medida de complejidad de los algoritmos será el *número* de evaluaciones de distancias realizadas.

Un *algoritmo de indexación* es un procedimiento para construir de antemano una estructura de datos (llamada *índice*) diseñada para ahorrar computaciones de distancia cuando luego se responden consultas por proximidad. Lo importante es, por lo tanto, diseñar algoritmos de indexación eficientes para reducir las evaluaciones de distancia. Todos los algoritmos de indexación particionan el conjunto S en subconjuntos. Se construye el índice para permitir determinar un conjunto de subconjuntos candidatos dónde se pueden encontrar los elementos candidatos a la consulta. Durante la consulta, se busca en el índice para encontrar los subconjuntos relevantes y luego se inspeccionan exhaustivamente todos estos conjuntos. Todas estas estructuras trabajan sobre la base de descartar elementos usando la desigualdad triangular (la única propiedad que nos permite ahorrar evaluaciones de distancia). La Figura 2.2 resume el modelo general utilizado para indexación y consultas en espacios métricos.

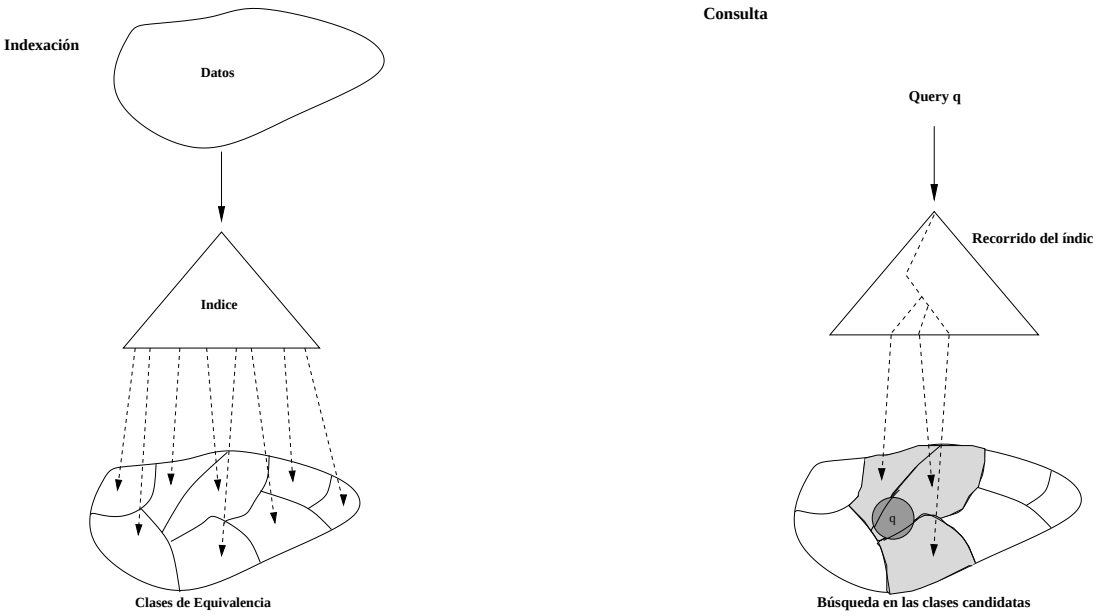


Figura 2.2: Modelo general que se utiliza para indexación y consulta en espacios métricos

2.3. Maldición de la Dimensionalidad

Uno de los principales obstáculos para el diseño de técnicas de búsqueda eficiente en espacios métricos es la existencia y ubicuidad en aplicaciones reales de los así llamados espacios de alta dimensionalidad. Las técnicas tradicionales de indexación como los *Kd-trees* [Ben75, Ben79] poseen una dependencia exponencial sobre la dimensión representacional del espacio.

Existen métodos efectivos para buscar sobre espacios de dimensión finita D (en adelante los llamaremos espacios vectoriales o espacios D-dimensionales). Sin embargo, para 20 dimensiones o más dichas

estructuras dejan de desempeñarse bien. Nos dedicamos aquí, como ya mencionamos, a espacios métricos generales, aunque las soluciones planteadas son también adecuadas para espacios D-dimensionales.

En [CNBYM01, CN00b, CN01] se muestra que el concepto de dimensionalidad intrínseca se puede entender aún en espacios métricos generales, se da una definición cuantitativa de ella y se muestra analíticamente la razón para la llamada "maldición de la dimensionalidad".

Es interesante notar que el concepto de "dimensionalidad" está relacionado con la "facilidad" o "dificultad" para buscar en un espacio D-dimensional: los espacios dimensionales más altos tienen una distribución de probabilidad de distancias entre elementos cuyo histograma es más concentrado y con una media grande. Esto hace que el trabajo de cualquier algoritmo de búsqueda por similaridad sea más dificultoso (esto se discute en [Yia93, Bri95, CM97, CNBYM01]). En el caso extremo tenemos un espacio donde $d(x, x) = 0$ y $\forall y \neq x,\ d(x, y) = 1$, donde se debe comparar exhaustivamente la query contra cada elemento en el conjunto. Se extiende esta idea diciendo que un espacio métrico general es más "difícil" (dimensión intrínseca más alta) que otro cuando su histograma de distancia es más concentrado que el del otro.

La idea es que, a medida que crece la dimensionalidad intrínseca del espacio, la media del histograma μ crece y su varianza σ^2 se reduce.

La Figura 2.3 nos muestra, de manera intuitiva, el por qué los histogramas más concentrados producen espacios métricos más difíciles (alta dimensión intrínseca). Sea p un elemento de la base de datos y q una query. La desigualdad triangular implica que cada elemento x tal que $|d(q, p) - d(p, x)| > r$ no puede estar a distancia menor o igual que r de q, así podríamos descartar a x. Sin embargo, en un histograma más concentrado las diferencias entre dos distancias aleatorias son más cercanas a cero y por lo tanto la probabilidad de descartar un elemento x es más baja. Las áreas sombreadas en la figura muestran los puntos que no se pueden descartar. A medida que el histograma es más y más concentrado alrededor de esa media, menor cantidad de puntos se pueden descartar usando la información que nos brinda $d(p, q)$.

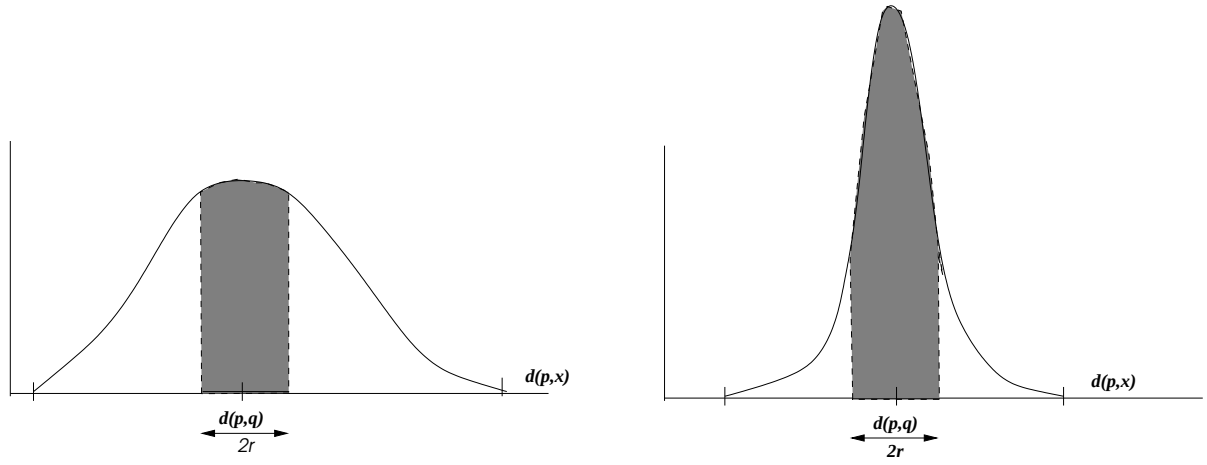


Figura 2.3: Un histograma de distancias para un espacio métrico de dimensión baja (izquierda) y de dimensión alta (derecha)

Este fenómeno es independiente de la naturaleza del espacio métrico (en particular si es vectorial o no) y nos da una manera de determinar cuán difícil es buscar sobre un espacio métrico arbitrario.

La *dimensionalidad intrínseca* de un espacio métrico se define como $\rho = \frac{\mu^2}{2\sigma^2}$, donde μ y σ^2 son la media y la varianza del histograma de distancias. Lo importante de la fórmula es que la dimensionalidad intrínseca

crece con la media y se reduce con la varianza del histograma. Esta definición además es coherente con la noción de dimensión en un espacio vectorial con coordenadas uniformemente distribuidas.

2.4. Ejemplos de Espacios Métricos

Presentamos aquí algunos ejemplos de espacios métricos particulares: espacios vectoriales, espacio de documentos representados como vectores y diccionarios (espacios de cadenas de caracteres o strings sobre un alfabeto).

2.4.1. Espacios Vectoriales

Si los elementos del espacio métrico (U, d) son realmente tuplas de números reales, entonces el par se denomina *espacio vectorial*. Un espacio vectorial de dimensión finita D es un espacio métrico particular donde los objetos se identifican por D números reales $(x_1, x_2, \ldots, x_D)$ y cada x_i es llamada "coordenada" del objeto (en adelante los llamaremos espacios vectoriales o espacios D-dimensionales). Existen distintas funciones de distancia que se pueden usar en un espacio métrico, pero las más usadas son las de la familia de L_s o *familia de distancias de Minkowski*, que se define como:

$$L_s\left((x_1,\ldots,x_D),(y_1,\ldots,y_D)\right) = \left(\sum_{1\le i\le D} |x_i - y_i|^s\right)^{1/s}$$

Algunos ejemplos de métricas pertenecientes a esta familia de distancias son la distancia L_1 conocida como *distancia Manhattan*, la L_2 que es más conocida como *distancia Euclídea*, y se corresponde a nuestra noción habitual de distancia, y la distancia L_∞, conocida también como *distancia del máximo*.

En nuestros ejemplos sobre espacios vectoriales usaremos la distancia L_2, la que se define como:

$$L_2\left((x_1,\ldots,x_D),(y_1,\ldots,y_D)\right) = \sqrt{\sum_{i=1}^{D} |x_i - y_i|^2} \tag{2.1}$$

En muchas aplicaciones los espacios métricos son en realidad espacios vectoriales, es decir que los objetos son puntos D-dimensionales y la similaridad se puede interpretar geométricamente. Un espacio vectorial permite más libertad al diseñar algoritmos de búsqueda, porque es posible usar la información de coordenadas y geométrica que no está disponible en los espacios métricos generales.

Las estructuras de búsqueda para espacios vectoriales más populares son *Kd-trees* [Ben75, Ben79], los *R-trees* [Gut84], los *Quad-trees* [Sam84] y los *X-trees* [BKK96] que son los más recientes. Todas estas técnicas usan ampliamente la información de coordenadas para agrupar y clasificar puntos en el espacio. Desafortunadamente estas técnicas son muy sensibles a la dimensión del espacio. Los algoritmos de búsqueda de punto más cercano y por rango dependen exponencialmente de la dimensión del espacio [Cha94].

Los espacios vectoriales pueden tener grandes diferencias entre su *dimensión representacional* (D) y su *dimensión intrínseca* (es decir, el número real de dimensiones en las cuales se pueden embeber los puntos manteniendo la distancia entre ellos).

Por ejemplo, uno de los espacios vectoriales considerados en nuestros experimentos corresponde a un espacio con dimensión representacional $D = 101$, pero en donde realmente existen sólo 200 clusters distintos, por lo cual su dimensión intrínseca es mucho menor (el histograma es menos concentrado que el de un espacio uniforme).

Si en un espacio vectorial sólo utilizamos la función de distancia para realizar las búsquedas, es decir, no utilizamos la información de las coordenadas de los objetos del espacio; es sencillo simular las búsquedas en un espacio métrico general. Una ventaja adicional es que la dimensión intrínseca del espacio se muestra independiente de cualquier dimensión representacional.

Para los experimentos hemos considerado distintos espacios vectoriales. Generamos espacios de vectores con coordenadas reales en el cubo unitario con distribución uniforme, en dimensiones 5, 10, 15 y 20. También utilizamos espacios de vectores generados con una distribución de Gauss, con media 1 y varianza 0.1, en dimensiones 5, 10, 15 y 20, para lo cual utilizamos los programas que se encuentran disponibles en la página web de DIMACS (*http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html*). Generamos del mismo modo espacios de vectores con distribución de Gauss, con media 1 y varianza 0.1, en dimensión 101 y agrupados en 200 clusters, como un ejemplo de espacio de baja dimensionalidad intrínseca, aunque alta dimensión representacional. Finalmente consideramos un espacio vectorial de dimensión 20, de 40700 vectores Euclidianos, de vectores de características de imágenes descargadas de la NASA (obtenido también desde la página web de DIMACS). En todos estos espacios métricos hemos utilizado la distancia L_2, que se definió en (2.1).

2.4.2. Modelo Vectorial para Documentos

En recuperación de la información [BYRN99] se define un *documento* como una "unidad de recuperación", la cual puede ser un párrafo, una sección, un capítulo, una página Web, un artículo o un libro completo. Los modelos clásicos en recuperación de la información consideran que cada documento está descripto por un conjunto representativo de palabras claves llamadas *términos*, que son palabras cuya semántica ayuda a definir los temas principales del documento.

Uno de estos modelos, el *modelo vectorial*, considera un documento como un vector t-dimensional, donde t es el número total de términos de la colección. Cada coordenada i del vector está asociada a un término del documento, cuyo valor corresponde a un "peso" positivo w_{ij} si es que dicho término (el término i) pertenece al documento j ó 0 en caso contrario. Si $\mathbb{D}$ es el conjunto de documentos y d_j es el j-ésimo documento perteneciente a $\mathbb{D}$, entonces $d_j = (w_{1j}, w_{2j}, \ldots, w_{tj})$.

En el modelo vectorial se calcula el *grado de similitud* entre un documento d y una consulta q, la cual se puede ver como un conjunto de términos o como un documento completo, como el grado de similitud entre los vectores $\vec{d_j}$ y $\vec{q}$. Esta correlación puede ser cuantificada, por ejemplo, como el coseno del ángulo formado entre ambos vectores:

$$sim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \times |\vec{q}|} = \frac{\sum_{i=1}^{t} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2 \times \sum_{i=1}^{t} w_{iq}^2}}$$

donde w_{iq} es el peso del i-ésimo término en la consulta q.

Los pesos de los términos pueden calcularse de varias formas. Una de las más importantes es utilizando *esquemas tf-idf*, en donde los pesos están dados por:

$$w_{ij} = f_{i,j} \times \log\left(\frac{N}{n_i}\right)$$

donde N es el número total de documentos, n_i es el número de documentos en donde el i-ésimo término aparece, y $f_{i,j}$ es la *frecuencia normalizada* del i-ésimo término, dada por:

$$f_{i,j} = \frac{freq_{i,j}}{\text{máx}_{\ell=1\ldots t}(freq_{\ell,j})}$$

donde $freq_{i,j}$ es la frecuencia del i-ésimo término en el documento d_j, y $\text{máx}_{\ell=1\ldots t}(freq_{\ell,j})$ es el máximo valor de frecuencia sobre todos los términos contenidos en d_j.

Si se considera que los documentos son puntos en un espacio métrico, el problema de la búsqueda de documentos similares a una consulta dada se reduce a una búsqueda en proximidad en el espacio métrico. Dado que $sim(d_j, q)$ sólo es una medida de similaridad, que en particular no cumple con la desigualdad triangular, se utiliza el ángulo formado entre los vectores $\vec{d_j}$ y $\vec{q}$, $d(d_j, q) = \arccos(sim(d_j, q))$, como función de distancia y por lo tanto se la denomina "distancia coseno" [BYRN99]. Por lo tanto, $(\mathbb{D}, d)$ es un espacio métrico.

Como ejemplo de esta clase de espacios métricos, hemos utilizado para los experimentos un espacio métrico de documentos con la función de distancia coseno. Tomamos 25896 documentos de la colección FR (Registro Federal) de TREC-3 [Har95].

2.4.3. Diccionarios

Otro ejemplo posible de espacio métrico, que también se ha utilizado en los experimentos de este trabajo, corresponde a un conjunto de palabras o diccionario. En este caso los objetos son palabras (o strings) en un determinado lenguaje (por ejemplo, Inglés, Español, Francés, Italiano, etc.).

Para medir la similitud entre palabras utilizamos la función de distancia conocida como *distancia de edición* (o *distancia de Levenshtein*). Se define como la cantidad de inserciones, eliminaciones o cambios de caracteres que hay que realizar para convertir una palabra en la otra. Claramente ésta es una función de distancia *discreta* y tiene muchas aplicaciones en recuperación de texto, procesamiento de señales y biología computacional [Nav01].

Si se consideran las palabras como puntos de un espacio métrico, el problema de buscar palabras similares a una dada se reduce a una búsqueda por proximidad en el espacio métrico. El caso particular de un diccionario es de interés en aplicaciones de correctores ortográficos.

Hemos usado para nuestros experimentos diccionarios de palabras en distintos idiomas. Los diccionarios considerados son uno de Inglés de 69069 palabras, otro de Español de 51789 palabras, uno de Francés de 138257 palabras y otro de Italiano de 116879 palabras, que sirven como representantes de los que conseguimos y son idiomas con los cuales estamos familiarizados. En los experimentos sobre estos espacios utilizamos la distancia de edición.

2.5. Trabajos Relacionados

Los algoritmos de búsqueda en espacios métricos generales se dividen en dos grandes áreas: *algoritmos basados en pivotes* y *algoritmos basados en particiones compactas* (ver [CNBYM01] para un análisis detallado).

El objetivo de estos algoritmos es reducir al mínimo el número de evaluaciones de distancia necesarias para responder a búsquedas por rango y a búsquedas de k-vecinos más cercanos.

La Figura 2.4 ilustra la clasificación de los índices de acuerdo a sus características más importantes.

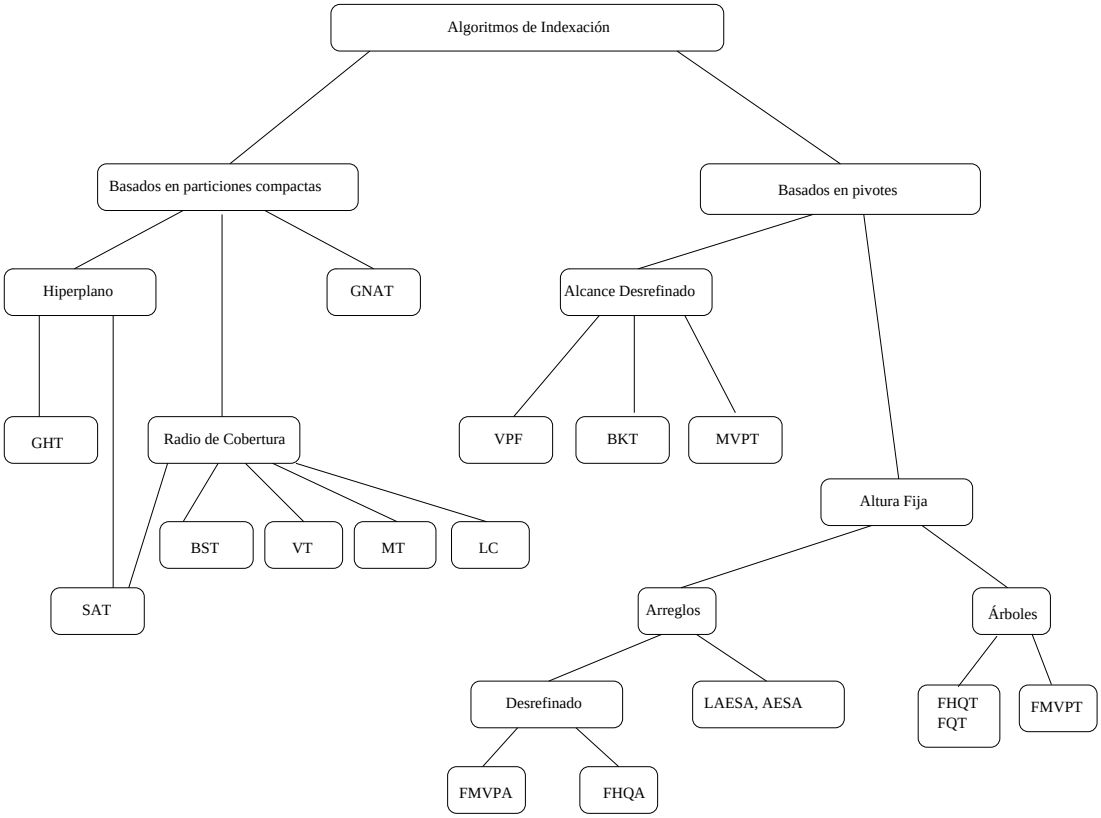


Figura 2.4: Taxonomía de los algoritmos existentes para búsqueda por proximidad en espacios métricos.

Por claridad en lo que sigue, en la Tabla 2.1 se muestra el nombre completo de cada una de las estructuras de datos para búsqueda por proximidad en espacios métricos (que aparecen clasificadas en la Figura 2.4), junto a la abreviatura que usaremos a lo largo de este trabajo y la referencia del artículo en donde se propuso cada una de ellas.

Analizamos brevemente aquí en qué consisten los algoritmos de cada una de estas grandes áreas, para dar el contexto de nuestro trabajo.

Algoritmos basados en pivotes. La idea es usar un conjunto de k elementos distinguidos ("pivotes") $p_1, p_2, \ldots, p_k \in S$ y almacenar, para cada elemento x de la base de datos, su distancia a los k pivotes $(d(x, p_1), \ldots, d(x, p_k))$. Dada una query q, se calcula su distancia a los k pivotes $(d(q, p_1), \ldots, d(q, p_k))$. Ahora, si para algún pivote p_i se cumple que $|d(q, p_i) - d(x, p_i)| > r$, enton-

Nombre Abreviado	*Estructura*	*Referencia*
BK-trees	Burkhard-Keller trees	[BK73]
FQ-trees	Fixed Queries trees	[BYCMW94]
FHQ-trees	Fixed Height FQ-trees	[BYCMW94, BY97, BYN98]
FQ-arrays	Fixed Queries arrays	[CMBY99, CMN01]
Metric Trees	Metric Trees	[Uhl91b]
VP-trees	Vantage Point trees	[Yia93, Chi94]
MVP-trees	Multi-Vantage Point trees	[BO97]
VP-forest	Excluded Middle Vantage Point Forest	[Yia99]
BS-trees	Bisector trees	[KM83]
GH-trees	Generalized-Hyperplane trees	[Uhl91b]
GNA-trees	Geometric Near-neighbor Access trees	[Bri95]
V-trees	Voronoi trees	[DN87]
M-trees	M-trees	[CPZ97]
LC	List of Clusters	[CN00a]
SA-trees	Spatial Aproximation trees	[Nav99]
AESA	Approximating Eliminating Search Algorithm	[Vid86]
LAESA	Linear AESA	[MOV94]

Tabla 2.1: Nombres de las estructuras de datos para búsquedas por proximidad en espacios métricos, con las referencias a los artículos en donde se propone cada una de ellas.

ces por la desigualdad triangular conocemos que $d(q,x) > r$ y, por lo tanto, no es necesario evaluar explícitamente $d(x,p)$. Los elementos que no se puedan descartar por medio de esta regla se deben comparar directamente contra la query.

Algoritmos tales como *AESA* [Vid86], *LAESA* [MOV94], *Spaghettis* y sus variantes [CMBY99, NN97], *FQ-trees* y sus variantes [BYCMW94] y *FQ-arrays* [CMN01], son casi las implementaciones más directas de esta idea, y difieren básicamente en su estructura extra para reducir el costo de CPU de encontrar los puntos candidatos, pero no en el número de evaluaciones de distancia que realizan.

Existen algunas estructuras de datos como árboles que utilizan esta idea de una manera indirecta: seleccionan un pivote como la raíz del árbol y dividen el espacio de acuerdo a las distancias a la raíz. A cada subárbol le corresponde una porción (el número y amplitud de las porciones difiere de acuerdo a las estrategias). En cada subárbol, se selecciona un nuevo pivote y así sucesivamente. La búsqueda realiza un backtrack sobre el árbol y se utiliza la desigualdad triangular para podar subárboles, es decir, si a es la raíz del árbol y b la raíz de un hijo tal que $d(a,b) \in [x_1, x_2]$, entonces podemos evitar entrar en el subárbol b siempre que se cumpla que $[d(q,a) - r, d(q,a) + r]$ no intersecte a $[x_1, x_2]$. Las estructuras de datos que usan esta idea son los *BK-trees* y sus variantes [BK73, Sha77], los *Metric trees* [Uhl91b], *TLAESA* [MOC96], y los *VP-trees* y variantes [Yia93, BO97, Yia00].

Algoritmos basados en particiones compactas. La segunda tendencia consiste en dividir el espacio en zonas o regiones tan compactas como sea posible. Normalmente se realiza recursivamente, y se almacena un punto representativo ("centro") para cada zona más algunos pocos datos extras que permitan descartar rápidamente la zona durante una búsqueda. Se pueden usar dos criterios para delimitar una zona.

El primero es el área de *Voronoi*, donde seleccionamos un conjunto de centros y colocamos cada

punto dentro de la región de su centro más cercano. Las áreas se delimitan con hiperplanos y las zonas son análogas a las regiones de Voronoi en espacios vectoriales. Sea $\{c_1, \ldots, c_m\}$ el conjunto de centros. Durante la búsqueda evaluamos $d(q, c_1), \ldots, d(q, c_m)$, elegimos el centro c_j más cercano y descartamos cada zona cuyo centro c_i satisfaga que $d(q, c_i) > d(q, c_j) + 2r$, porque su área de Voronoi no puede intersectar la "bola" de la query (con centro q y radio r)[1].

El segundo criterio es el de *radio de cobertura* $rc(c_i)$, el cual es la máxima distancia entre el centro c_i y un elemento de su zona. Si $d(q, c_i) - r > rc(c_i)$, entonces no es necesario considerar la zona i.

Las técnicas se pueden combinar. Algunas que usan sólo hiperplanos son los *GH-trees* y sus variantes [Uhl91b, NVZ92], y los *Voronoi trees* [DN87, Nol89]. Algunas que usan sólo radios de cobertura son los *M-trees* [CPZ97] y *Lista de Clusters* [CN00a]. Uno que usa ambos criterios es el *GNA-tree* [Bri95].

Para responder a las consultas 1-NN, simulamos una búsqueda por rango con un radio inicialmente $r = \infty$, y reducimos r a medida que encontremos elementos más y más cercanos a q. Al final, tenemos en r la distancia al elemento más cercano y hemos visto a todos los más cercanos. A diferencia de la consulta por rango, ahora estamos interesados en encontrar rápidamente elementos cercanos a q para reducir r lo antes posible, por ello existen varias heurísticas para lograrlo. Una de las más interesantes se propone en [Uhl91a] para los *Metric Trees*, donde se almacenan los subárboles en una cola de prioridad en un orden heurísticamente prometedor. El recorrido es más general que un backtracking. Cada vez que procesamos el subárbol más prometedor, debemos agregar sus hijos a la cola de prioridad. En algún momento podemos detener la búsqueda usando criterios de poda basados en la desigualdad triangular. El orden, para procesar los subárboles, es la cota inferior que entrega el índice.

Las consultas k-NN son realizadas como una generalización de las consultas 1-NN. En lugar de un elemento más cercano, se mantiene una cola de prioridad de los k elementos más cercanos que se conocen. El valor r ahora es la distancia a la que se encuentra el elemento más alejado de q, considerando los k candidatos actuales. Cada nuevo candidato se inserta en un heap y puede desalojar a uno más lejano de la cola (por lo tanto se reduce r para lo que sigue del algoritmo).

Notar que todos los trabajos descriptos se basan en dividir la base de datos, lo que se hereda de las ideas clásicas de *dividir para conquistar* que se aplican en estructuras de datos típicas (v.g. árboles binarios de búsqueda). En este trabajo se utiliza otra aproximación que es específica de la búsqueda espacial. Más que dividir el conjunto de candidatos durante la búsqueda, tratamos de acercarnos espacialmente cada vez más a la query.

2.5.1. Capacidades Dinámicas.

La mayoría de las estructuras de datos para espacios métricos están diseñadas para construirse sobre un conjunto de datos estático. En muchas aplicaciones esto no es razonable porque se deben insertar y eliminar elementos dinámicamente. Algunas estructuras de datos soportan inserciones, pero casi ninguna soporta eliminaciones.

En [CNBYM01] se analizan las estructuras y sus capacidades dinámicas y se llega a las siguientes conclusiones:

[1] Llamamos "bola" de la query q al conjunto de puntos que se encuentran a distancia a lo más r de q.

Inserciones. Entre las estructuras que allí se analizan, el *SA-tree* hasta ese momento *era* la menos dinámica [Nav99], porque necesitaba conocer completamente todo el conjunto para construir el índice. La familia *VP* (*VP-tree, MVP-tree y VP-forest*) tiene el problema de basarse en estadísticas globales (tal como la mediana) para construir el árbol, por lo tanto más tarde se pueden hacer inserciones pero el desempeño de la estructura se puede degradar. Finalmente, el *FQ-array* necesita en principio insertar en la mitad de un arreglo, pero esto se puede realizar usando técnicas estándar. Todas las otras estructuras de datos pueden realizar inserciones de una manera razonable. Hay algunos parámetros de las estructuras que pueden depender de n y por ello requieren una reorganización estructural periódica.

Eliminaciones. Las eliminaciones son un poco más complicadas. Además de las estructuras anteriores, que presentan problemas para la inserción, *BK-trees*, *GH-trees*, *BS-trees*, *V-trees*, *GNA-trees* y la familia *VP* no admiten eliminaciones de un nodo interno del árbol, porque ellos juegan un rol esencial en la organización del subárbol. Desde luego esto se puede solucionar sólo marcando el nodo como eliminado y manteniéndolo para el propósito de elegir los caminos a seguir, pero la calidad de la estructura de datos se degrada a lo largo del tiempo. Más aún, en algunas aplicaciones no es aceptable dicha solución por cuestiones de espacio, esto puede ocurrir cuando los objetos son muy voluminosos y mantenerlos se hace imposible.

Por lo tanto, las únicas estructuras que soportan completamente inserciones y eliminaciones son las de la familia *FQ* (*FQ-trees*, *FQH-trees*, *FQ-arrays*, porque no poseen nodos internos), *AESA* y *LAESA* (porque son sólo vectores de coordenadas), el *M-tree* (que se diseñó con capacidades dinámicas en mente y cuyos algoritmos de inserción y eliminación hacen recordar a los del *B-tree*), y una variante de los *GH-trees* diseñados para soportar operaciones dinámicas [Ver95]. El análisis de esta última estructura muestra que las inserciones dinámicas se pueden realizar en tiempo de peor caso amortizado de $O(\log^2 n)$, y las eliminaciones se pueden realizar a un costo similar bajo algunas restricciones.

Capítulo 3

Árbol de Aproximación Espacial

Todos los trabajos previos se basan en dividir la base de datos, lo que se ha heredado desde las ideas clásicas de *dividir para conquistar* y de la búsqueda de datos típicos (v.g. árboles de búsqueda binaria). Se describe aquí una nueva técnica que es específica de la búsqueda espacial, propuesta en [Nav02]. Más que dividir el conjunto de candidatos durante la búsqueda, se trata de comenzar la búsqueda en algún punto del espacio y acercarse espacialmente a la query q, encontrando elementos cada vez más cercanos a ella. Para introducir el Árbol de Aproximación Espacial (*SA-tree*) necesitamos entender en qué consiste la aproximación espacial.

3.1. Acercamiento a la Aproximación Espacial

Para entender qué es la aproximación espacial analizamos las consultas de *1-vecino más cercano* o *1-NN* (después veremos cómo resolver todos los tipos de consultas). En lugar de los algoritmos conocidos para resolver las consultas por proximidad dividiendo el conjunto de candidatos, trabajamos con una aproximación diferente. En este modelo, nos ubicamos en un elemento de S dado y tratamos de acercarnos "espacialmente" a la query. Cuando no se puede hacer más este movimiento, estamos posicionados en el elemento más cercano a la query en todo el conjunto.

Las aproximaciones son realizadas sólo vía "vecinos". Cada elemento del conjunto $a \in S$ tiene un conjunto de vecinos $N(a)$, y está permitido moverse sólo a los vecinos. La estructura natural para representar esta restricción es un grafo dirigido. Los nodos son los elementos del conjunto S y están conectados a sus vecinos por arcos. Más específicamente, existe un arco desde a hasta b si es posible moverse de a a b en un único paso.

Cuando tal grafo está definido adecuadamente, el proceso de búsqueda para una query q es simple: comenzamos la búsqueda posicionados en un nodo aleatorio a y consideramos todos sus vecinos. Si ningún vecino está más cerca de q que de a, entonces, entregamos a como el vecino más cercano a q. En otro caso, seleccionamos algún vecino b que esté más cerca de q que a y nos movemos a b. Se puede elegir a b como el vecino que esté más cerca de q (*best-fit*) o como el primero que se encuentre más cerca de q que a (*first-fit*).

Para que el algoritmo funcione, el grafo debe contener suficientes arcos. El grafo más simple que se puede utilizar es el grafo completo, es decir todos los pares de nodos son vecinos. Sin embargo, esto implica hacer n evaluaciones de distancia sólo para verificar los vecinos del primer nodo. Por lo tanto, es preferible

aquel grafo que tenga el *menor* número posible de arcos y que aún así permita responder correctamente a todas las consultas. Este grafo $G = (S, \{(a, b), a \in S, b \in N(a)\})$ debe cumplir la siguiente propiedad:

Condición 1: $\forall a \in S$, $\forall q \in U$, si $\forall b \in N(a),\ d(q, a) \leq d(q, b)$, entonces $\forall b \in S,\ d(q, a) \leq d(q, b)$.

Esto significa que, dado *cualquier* elemento q, si no se puede acercar más a q desde a yendo a través de sus vecinos, entonces es porque a ya es el elemento más cercano a q en todo el conjunto S. Es claro que si el grafo G satisface la Condición 1 podemos buscar por aproximación espacial. Buscamos un grafo de esta clase que sea minimal (en cantidad de arcos).

Se puede ver esto de otra manera: cada elemento $a \in S$ tiene un subconjunto de U donde él es la respuesta apropiada (es decir, el conjunto de objetos más cercanos a a que a cualquier otro elemento de S). Esto es el análogo exacto de una "región de Voronoi" para espacios Euclídeos en geometría computacional [Aur91][1]. La respuesta a una query q es el elemento $a \in S$ propietario de la región de Voronoi en la que cae q. Si a no es la respuesta, necesitamos ser capaces de movernos a otro elemento más cercano a q. Es suficiente para ello conectar a con todos sus "vecinos de Voronoi" (es decir, los elementos del conjunto S cuyas áreas de Voronoi comparten una frontera con la de a), ya que si a no es la respuesta, entonces un vecino de Voronoi estará más cerca de q (esto es exactamente lo que establece la Condición 1).

Consideremos el hiperplano entre a y b (es decir, aquel que divide el área de puntos x más cercanos a a o más cercanos a b). Cada elemento b que agreguemos como un vecino de a hará que la búsqueda se mueva desde a hacia b, si q está del lado del hiperplano de b. Por lo tanto, si (y sólo si) se le agregan todos los vecinos de Voronoi a a, la única zona en donde la consulta no se moverá más allá de a será exactamente el área donde a es el vecino más cercano.

Por lo tanto, en un espacio vectorial, el grafo minimal que buscamos corresponde a la triangulación clásica de Delaunay (un grafo donde los elementos que son vecinos de Voronoi están conectados). Así, la respuesta ideal en términos de complejidad de espacio es el grafo de Delaunay (generalizado para espacios arbitrarios), y permitiría también búsquedas rápidas. La Figura 3.1 muestra un ejemplo. Comenzamos la búsqueda desde p_{11} y alcanzamos a p_9, el nodo más cercano a q, moviéndonos siempre a vecinos más y más cercanos a q.

Desafortunadamente, no es posible computar el grafo de Delaunay de un espacio métrico general dado sólo el conjunto de distancias entre los elementos de S sin ninguna indicación adicional de la estructura del espacio. Esto es porque, dado el conjunto de $|S|^2$ distancias, diferentes espacios tendrán diferentes grafos. Más aún, no es posible probar que un único arco desde algún nodo a a un nodo b no está en el grafo de Delaunay. Por lo tanto, el único superconjunto del grafo de Delaunay que funciona para un espacio métrico arbitrario es el grafo completo, y como ya se explicó es inútil. Esto elimina la estructura de datos para aplicaciones generales. Se formaliza esta noción como un teorema.

Teorema:

Dadas las distancias entre cada par de elementos de un subconjunto finito S de un espacio métrico desconocido U, entonces para cada a, $b \in S$ existe una elección de U donde a y b están conectados en el grafo de Delaunay de S.

[1] El nombre correcto en un espacio métrico general es "Dominio de Dirichlet" [Bri95].

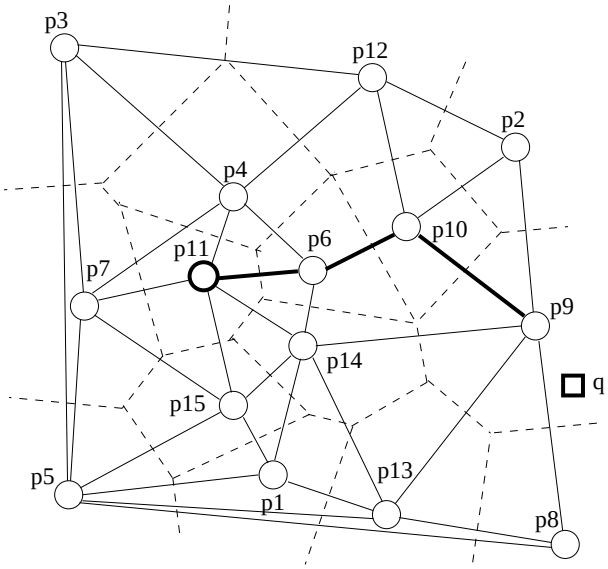


Figura 3.1: Un ejemplo del proceso de búsqueda con un grafo de Delaunay (arcos sólidos) correspondiente a una partición de Voronoi (áreas delimitadas por líneas punteadas).

Demostración:

Dado el conjunto de distancias, creamos un nuevo elemento $x \in U$ tal que $d(a, x) = M + \epsilon$, $d(b, x) = M$, y $d(y, x) = M + 2\epsilon$ para todos los otros y de S. Esto satisface todas las desigualdades triangulares si $\epsilon \leq 1/2\ min_{y,z \in S}\{d(y, z)\}$ y $M \geq 1/2\ max_{y,z \in S}\{d(y, z)\}$. Por lo tanto, tal x puede existir en U. Ahora, dada la consulta $q = x$ y dado que estamos actualmente en el elemento a, b es el elemento más cercano a x y la única manera de moverse hacia b sin irse más lejos de q es un arco directo desde a a b (ver Figura 3.1). Este argumento se puede repetir para cada par de elementos a, $b \in S$.

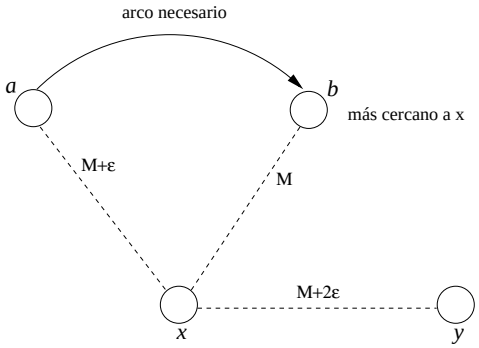


Figura 3.2: Ilustración del teorema.

Como la demostración del teorema supone que se pueden construir determinados elementos a distancia arbitraria entre dos dados, y como no todos los espacios métricos tienen esta propiedad, en realidad está afirmando que existen espacios donde no se puede construir un grafo de aproximación; aunque en ciertos espacios podría ser posible construirlo.

3.2. El Árbol de Aproximación Espacial

Para lograr una solución factible se hacen dos simplificaciones cruciales a la idea general. La simplificación resultante responde sólo a un conjunto reducido de consultas, es decir las consultas por el 1-vecino más cercano (*1-NN*) de $q \in S$, que no es más que una búsqueda exacta. Sin embargo, luego mostraremos (Sección 3.2.2) cómo combinar la aproximación espacial con backtracking para responder a cualquier query $q \in U$ (no sólo $q \in S$), para consultas por rango y consultas de k-vecinos más cercanos (*k-NN*).

(1) No comenzamos atravesando el grafo desde un nodo aleatorio sino desde uno fijo, y por lo tanto no se necesitan todos los arcos de Voronoi.

(2) El grafo sólo podrá responder correctamente consultas por elementos $q \in S$, es decir sólo elementos presentes en la base de datos.

Haciendo estas simplificaciones, se construye un grafo análogo al de Delaunay para buscar por aproximación espacial consultas 1-NN. Realmente, el resultado no es un grafo sino un árbol, el cual se ha llamado *SA-tree* (*"Árbol de Aproximación Espacial"*). Luego mostraremos cómo usar este árbol para buscar cualquier query $q \in U$ (no sólo para $q \in S$) y cualquier tipo de consulta.

3.2.1. Proceso de Construcción

Seleccionamos aleatoriamente un elemento $a \in S$ como la raíz del árbol. A continuación se selecciona un conjunto adecuado de vecinos $N(a)$ que satisface la siguiente propiedad:

Condición 2: (dados a, S) $\forall x \in X, x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\},\ d(x,y) > d(x,a)$.

Es decir, los vecinos de a forman un conjunto tal que cualquier vecino está más cerca de a que de cualquier otro vecino. La parte "sólo si" ($\Leftarrow$) de la definición garantiza que si podemos acercarnos a algún $b \in S$ entonces un elemento en $N(a)$ está más cerca de b que de a, porque pusimos como vecinos directos a todos aquellos elementos que no están más cerca de cualquier otro vecino. La parte "si" ($\Rightarrow$) apunta a obtener además sólo los vecinos necesarios.

Notar que el conjunto $N(a)$ está definido no trivialmente en términos de sí mismo y que múltiples soluciones cumplen la definición. Por ejemplo, si a está alejado de b y c y éstos a su vez están cerca entre sí, entonces ambos $N(a) = \{b\}$ y $N(a) = \{c\}$ satisfacen la condición.

Encontrar el conjunto $N(a)$ más pequeño posible parece ser un problema de optimización combinatoria no trivial, ya que si incluimos un elemento necesitamos sacar otros (esto ocurre entre b y c en el ejemplo del párrafo anterior). Sin embargo, heurísticas simples que agregan más vecinos que los necesarios trabajan bien. Comenzamos con el nodo inicial a y su "bolsa" manteniendo todo el resto de S. Como se espera que los nodos más cercanos sean más probablemente los vecinos, primero se ordena el conjunto de nodos por distancia a a. Entonces, comenzamos agregando nodos a $N(a)$ (el cual es inicialmente vacío). Cada vez que consideramos un nuevo nodo b , analizamos si es más cercano a algún elemento de $N(a)$ que a a mismo. Si ese no es el caso, se agrega b a $N(a)$.

Al finalizar tenemos un conjunto adecuado de vecinos. Notar que la Condición 2 se satisface gracias al hecho de haber considerado los elementos en orden creciente de distancia a a. La parte "$\Leftarrow$" de la condición

claramente se cumple porque se inserta en $N(a)$ cualquier elemento que satisface la parte derecha de la cláusula. La parte "$\Rightarrow$" es más delicada. Sea $x \neq y \in N(a)$. Si y está más cerca de a que x entonces y se consideró primero. El algoritmo de construcción garantiza que si insertamos x en $N(a)$ entonces $d(x,a) < d(x,y)$. Si, por otro lado, x está más cerca de a que y, entonces $d(x,y) > d(y,a) \geq d(x,a)$ (es decir, un vecino no puede ser descartado por un nuevo vecino que se insertó después).

Ahora debemos decidir en cuál de las bolsas de los vecinos poner el resto de los nodos. Se puede poner cada nodo que no está en $\{a\} \cup N(a)$, como ya se estableció, en la bolsa de su elemento más cercano en $N(a)$ (*best-fit*). Se observa que esto requiere una segunda pasada cuando esté determinado completamente $N(a)$. Como se muestra luego, es también posible poner cada nodo en la bolsa del primer vecino más cercano que a (*first-fit*), y no es necesaria una segunda pasada, pero la búsqueda se degrada.

Como ya se ha hecho con a, se procesan recursivamente todos sus vecinos, cada uno con los elementos de su bolsa. Notar que la estructura resultante no es un grafo sino un árbol, en el cual se puede buscar por cualquier $q \in S$ por aproximación espacial para consultas del vecino más cercano. El mecanismo consiste en comparar q contra $\{a\} \cup N(a)$. Si a es el más cercano a q, entonces a es la respuesta, en otro caso continuamos la búsqueda por el subárbol del elemento más cercano a q en $N(a)$.

La razón por la que esto funciona es que, en tiempo de búsqueda, se puede replicar lo que pasó con q durante el proceso de construcción (v.g. entrar en el subárbol del vecino más cercano a q), hasta que alcancemos a q. Esto es porque q ya está en el árbol, o sea, estamos realizando una búsqueda exacta.

Finalmente, ahorramos algunas comparaciones durante la búsqueda almacenando en cada nodo a su radio de cobertura, es decir la máxima distancia $R(a)$ entre a y cualquier elemento en el subárbol con raíz a. La manera en que se usa esta información se hará evidente en la Sección 3.2.2.

La Figura 3.3 muestra el proceso de construcción. Se invoca por primera vez como Construir(a, $S - \{a\}$) dónde a es un elemento aleatorio del conjunto S. Notar que, excepto para el primer nivel de recursión, conocemos todas las distancias $d(v,a)$ para cada $v \in S$ y por lo tanto no necesitamos recalcularlas. De manera similar, algunas de las $d(v,c)$ en la línea 10 ya se conocen desde la línea 6. La información que se almacena en la estructura de datos es la raíz a y los valores de $N()$ y $R()$ para todos los nodos.

```
Construir (Nodo a, Conjunto de Elementos S)

1.   N(a) ← ∅        /* vecinos de a */
2.   R(a) ← 0        /* radio de cobertura */
3.   Ordenar S por distancia a a (más cercano primero)
4.   For v ∈ S
5.      R(a) ← máx(R(a), d(v,a))
6.      If ∀b ∈ N(a), d(v,a) < d(v,b) Then
7.         N(a) ← N(a) ∪ {v}
8.   For b ∈ N(a) S(b) ← ∅       /* subárboles */
9.   For v ∈ S − N(a)
10.     Sea c ∈ N(a) el que minimiza d(v,c)
11.     S(c) ← S(c) ∪ {v}
12.  For b ∈ N(a) Construir (b,S(b))      /* construir subárboles */
```

Figura 3.3: Algoritmo para construir un *SA-tree*.

Siendo un árbol, el espacio necesario para esta estructura es $O(n)$.

3.2.2. Búsqueda

Desde luego es de poco interés buscar sólo elementos $q \in S$. El árbol descripto puede, sin embargo, ser utilizado como un dispositivo para resolver consultas de cualquier tipo para cualquier $q \in U$. Comencemos analizando las consultas por rango con radio r.

La observación clave es que, aún si $q \notin S$, las respuestas a la consulta son elementos $q' \in S$. Así usamos el árbol para simular que se está buscando un elemento $q' \in S$. No conocemos a q', pero como $d(q, q') \leq r$, podemos obtener a partir de q alguna información de distancia con respecto a q': por la desigualdad triangular se cumple que para cualquier $x \in U$, $d(x, q) - r \leq d(x, q') \leq d(x, q) + r$.

Cuando conocemos el q que estamos buscando, vamos directamente al vecino de a más cercano a q. Ahora, estamos buscando a un q' desconocido y no sabemos cuál es el vecino de a más cercano a q'. Por lo tanto tenemos que explorar varios vecinos posibles. A algunos vecinos, afortunadamente, se los puede deducir como irrelevantes, en la medida que q' no los pudo haber elegido en la construcción si se cumple que $d(q, q') \leq r$.

En lugar de ir simplemente al vecino más cercano, determinamos primero el vecino más cercano c de q entre $a \cup N(a)$. Así, para cada b en $\{a\} \cup N(a)$ sabemos que $d(c, q) \leq d(b, q)$. Sin embargo, como se explicó, es posible que $d(c, q') > d(b, q')$ y por lo tanto no encontraremos a q' sólo entrando en el subárbol de c. En su lugar, debemos entrar en *todos* los vecinos $b \in N(a)$ tal que $d(q, b) \leq d(q, c) + 2r$. Esto es porque el elemento virtual q' que estamos buscando puede diferir de q en a lo más r en cualquier evaluación de distancia, por ello se podría haber insertado dentro de tales nodos b. En otras palabras, un vecino b tal que $d(q, b) > d(q, c) + 2r$ satisface que $d(q', b) \geq d(q, b) - r > d(q, c) + r \geq d(q', c)$, así q' no se podría haber insertado en el subárbol de b. En cualquier otro caso, no estamos seguros de ello y debemos entrar al subárbol de b.

Una manera diferente de ver este proceso es limitar inferiormente la distancia entre q y cualquier nodo x en el subárbol de b. Por la desigualdad triangular tenemos que $d(x, q) \geq d(x, c) - d(q, c)$ y $d(x, q) \geq d(q, b) - d(x, b)$. Resumiendo ambas desigualdades y manteniendo en mente que $d(x, b) \leq d(x, c)$ y $d(q, c) \leq d(q, b)$, obtenemos que $2d(x, q) \geq (d(q, b) - d(q, c)) + (d(x, c) - d(x, b)) \geq d(q, b) - d(q, c)$. Por lo tanto $d(x, q) \geq (d(q, b) - d(q, c))/2$. Si el último término es mayor que r, seguramente podemos descartar cada x en el subárbol de b. La condición es por ello $(d(q, b) - d(q, c))/2 > r$, o $d(q, b) > d(q, c) + 2r$.

El proceso garantiza que compararemos q contra cada nodo al que no se le puede probar que esté suficientemente lejos de q. De aquí, informando cada nodo q' que comparamos contra q y para el cual se cumple que $d(q, q') \leq r$, estamos seguros de informar cada elemento relevante.

Como se puede observar, lo que originalmente se concibió como una búsqueda por aproximación siguiendo un único camino se combina ahora con *backtracking*, por ello buscamos en varios caminos. Éste es el precio por no ser capaces de construir un verdadero grafo de aproximación espacial. La Figura 3.4 ilustra el proceso de búsqueda, comenzando desde p_{11} (la raíz). Sólo p_9 se encuentra en el resultado, pero se atraviesan todos los arcos resaltados.

El algoritmo de búsqueda se puede mejorar un poco más. Cuando buscamos un elemento $q \in S$ (es decir, una búsqueda exacta por un nodo del árbol), seguimos un único camino desde la raíz a q. En cada nodo a' en este camino, elegimos el más cercano a q entre $\{a'\} \cup N(a')$. Por lo tanto, si la búsqueda se encuentra actualmente en el nodo a del árbol, sabemos que q está más cerca de a que de cualquier ancestro a' de a y también que de cualquier vecino de a'. De aquí, si llamamos $A(a)$ al conjunto de ancestros de a (incluyendo a a) y $N(A(a)) = \bigcup_{a' \in A(a)} N(a')$ tenemos que, durante la búsqueda, podemos evitar entrar en

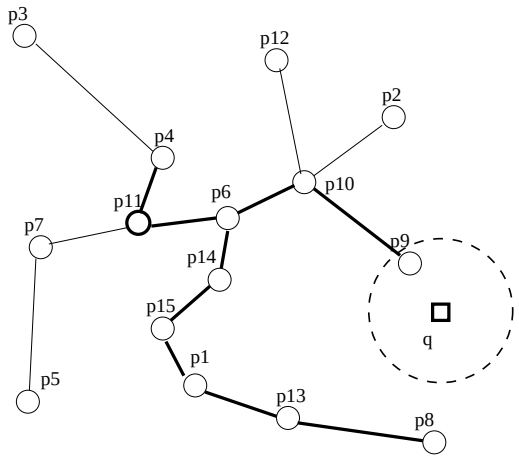


Figura 3.4: Un ejemplo del proceso de búsqueda.

cualquier elemento $x \in N(a)$ tal que

$$d(q,x) > 2r + \text{mín}\{d(q,c),\ c \in N(A(a))\}$$

debido a que podemos mostrar usando la desigualdad triangular que ningún q' con $d(q,q') \leq r$ se pudo almacenar dentro de x. Esta condición es una versión más estricta de la condición original $d(q,x) > 2r + \text{mín}\{d(q,c),\ c \in \{a\} \cup N(a)\}$.

Usamos esta observación como sigue. En cualquier nodo b de la búsqueda mantenemos registro de la mínima distancia d_{min} a q vista hasta el momento a lo largo del camino, incluyendo vecinos. Entramos sólo en los vecinos que no estén más lejos que $d_{min} + 2r$ de q.

Finalmente, el radio de cobertura $R(a)$ se usa para reducir más el costo de búsqueda. Nunca entramos en un subárbol con raíz a en donde $d(q,a) > R(a) + r$, ya que esto implica que $d(q',a) > R(a)$ para cualquier q' tal que $d(q,q') \leq r$. La definición de $R(a)$ implica que q' no puede pertenecer al subárbol de a. La Figura 3.5 muestra el algoritmo. Se invoca por primera vez como `BúsquedaRango(`a`,`q`,`r`,`$d(a,q)$`)` dónde a es la raíz del árbol. Notar que, en invocaciones recursivas, la distancia $d(a,q)$ ya está calculada.

```
BúsquedaRango (Nodo a, Query q, Radio r, Distancia d_min)

1. If d(a,q) ≤ R(a) + r Then
2.     If d(a,q) ≤ r Then Informar a
3.     d_min ← mín {d_min} ∪ {d(q,c), c ∈ N(a)}
4.     For b ∈ N(a)
5.         If d(b,q) ≤ d_min + 2r Then BúsquedaRango (b,q,r,d_min)
```

Figura 3.5: Algoritmo para buscar q con radio r en un *SA-tree*.

3.2.3. Búsqueda de vecinos más cercanos

También podemos realizar búsquedas del vecino más cercano simulando una búsqueda por rango, donde vamos reduciendo el radio de búsqueda a medida que obtenemos más información. Para resolver las con-

sultas del vecino más cercano (*1-NN*), comenzamos buscando con $r = \infty$, y reducimos r cada vez que realizamos una nueva comparación que da una distancia más chica que r. Finalmente, informamos el elemento más cercano que hayamos visto durante toda la búsqueda. Para las consultas sobre los k vecinos más cercanos (*k-NN*) almacenamos todo el tiempo una cola de prioridad con los k elementos más cercanos a q vistos hasta el momento. El radio r es la distancia entre q y el elemento más lejano en la cola (∞ cuando todavía tenemos menos que k candidatos). Cada vez que aparece un nuevo candidato lo insertamos en la cola, lo cual puede desplazar a otro elemento y por lo tanto reducir r. Al final, la cola contiene los k elementos más cercanos a q (recordar la Sección 2.5).

En una búsqueda por rango normal con radio fijo r, no es importante el orden en el que hacemos backtrack en el árbol. Ahora éste no es el caso, porque podríamos rápidamente encontrar elementos cercanos a q para reducir pronto r. Una idea general propuesta en [Uhl91a] se puede adaptar a esta estructura de datos. Tenemos una cola de prioridad, donde los elementos más prometedores están primero. Inicialmente, insertamos la raíz del *SA-tree* en la estructura de datos. Iterativamente, extraemos la raíz del subárbol más prometedor, la procesamos e insertamos todas las raíces de sus subárboles en la cola. Esto se repite hasta que la cola se vacíe o se pueda descartar su subárbol más prometedor (es decir, su "valor prometido" es suficientemente malo).

La medida más elegante de cuán prometedor es un subárbol es un límite inferior de la distancia entre q y cualquier elemento del subárbol. Cuando este límite inferior excede r podemos detener completamente el proceso. Tenemos en realidad dos posibles límites inferiores:

1. Como encontramos el vecino más cercano c y entonces entramos en cualquier otro vecino b tal que $d(q,b) - d(q,c) \leq 2r$, no tendríamos que entrar en el subárbol con raíz b si no se cumple que $(d(q,b) - d(q,c))/2 \leq r$. De hecho, este c se toma sobre los vecinos de cualquier ancestro ($N(A(b))$).
2. Por el límite inferior de la distancia entre q y un elemento en el subárbol tenemos que $d(q,b) - R(b) \leq r$.

Como r se reduce durante la búsqueda, un nodo b puede parecer útil cuando se lo inserta en la cola de prioridad y ser inútil después, cuando se lo extrae de la cola para procesarlo. Así almacenamos junto con b el máximo de los dos límites inferiores, y usamos este máximo para ordenar los subárboles en la cola de prioridad, con el más pequeño primero. A medida que extraemos subárboles de la cola, verificamos si su valor excede a r, en cuyo caso detenemos el proceso completo y ya se sabe que todos los subárboles restantes contendrán elementos irrelevantes. Notar que necesitamos mantener el rastro de d_{min} separadamente y que los nodos hijos heredan el límite inferior de sus padres.

La Figura 3.6 presenta el algoritmo, donde A es una cola de prioridad de pares *(nodo,distancia)* ordenados por *distancia* creciente y Q es una cola de prioridad de ternas *(nodo,lbound,*d_{min}*)* ordenados por *lbound* creciente.

3.3. Análisis

Daremos ahora un breve análisis de la estructura de *SA-tree*. Un análisis completo y mayores detalles respecto de lo que aquí se menciona se encuentra en [Nav02](Apéndices A y B).

Éste es un análisis que se ha simplificado en varios aspectos, por ejemplo se supone que la distribución de distancia de los nodos que están en el árbol es la misma que en el espacio global. Tampoco se tiene

```
BúsquedaNN (Árbol a, Query q, Vecinos requeridos k)

1.   Q ← {(a, máx(0, d(q,a) − R(a)), d(q,a))}   /* subárboles prometedores */
2.   A ← ∅   /* mejor respuesta hasta ahora */
3.   r ← ∞
4.   While Q no sea vacía
5.      (b, t, d_min) ← elemento en Q con el t más pequeño
6.      Q ← Q − {(b, t, d_min)}
7.      If t > r Break   /* criterio global de parada */
8.      A ← A ∪ {(b, d(q,b))}
9.      If |A| = k + 1
10.       (c, maxd) ← elemento en A con maxd más grande
11.       A ← A − {(c, maxd)}
12.     If |A| = k
13.       (c, maxd) ← elemento en A con maxd más grande
14.       r ← maxd
15.     d_min ← mín {d_min} ∪ {d(c,q), c ∈ N(b)}
16.     For v ∈ N(b)
17.        Q ← Q ∪ {(v, máx(t, (d(q,v)−)/2, d(q,v) − R(v)), d_min)}
18.  Informar la respuesta A
```

Figura 3.6: Algoritmo para buscar los k vecinos más cercanos a q en un *SA-tree*.

en cuenta que se ordenan las bolsas antes de seleccionar los vecinos (los resultados son pesimistas en este sentido, dado que parece como si tuvieran más vecinos). Sin embargo, como se puede ver en los resultados experimentales [Nav02], la adecuación con la realidad es muy buena. Este análisis se hace para una función de distancia continua, aunque adaptarlo al caso discreto es inmediato.

Los resultados se pueden resumir como sigue. El *SA-tree* necesita espacio lineal $O(n)$, un tiempo de construcción $O(n \log^2 n / \log\log n)$ y tiempo de búsqueda sublineal $O(n^{1-\Theta(1/\log\log n)})$ en espacios difíciles (alta dimensionalidad intrínseca) y $O(n^\alpha)$ $(0 < \alpha < 1)$ en espacios sencillos (baja dimensionalidad intrínseca).

Se demuestra (en el Apéndice A de [Nav02]) que el número promedio de vecinos es

$$N(n) \;=\; \Theta(\log n)$$

bajo las simplificaciones ya mencionadas. La constante está entre 1.00 y 1.35. Existe también una parte constante que sería especialmente relevante en espacios sencillos. Sin embargo, para dicho análisis basta con $\Theta(\log n)$.

Esto permite determinar algunos parámetros de *SA-tree*. Por ejemplo, como en promedio $\Theta(n/\log n)$ elementos van en cada subárbol, la profundidad promedio de una hoja en el árbol es

$$H(n) \;=\; 1 + H\left(\frac{n}{\log n}\right) \;=\; \Theta\left(\frac{\log n}{\log\log n}\right)$$

lo que se obtiene en detalle en el Apéndice B de [Nav02].

El costo de construcción es como sigue (en términos de evaluaciones de distancia). La bolsa con los n elementos es comparada contra el nodo raíz. Se seleccionan $\Theta(\log n)$ como vecinos y luego todos los demás elementos se comparan contra los vecinos y se incorporan en una bolsa. Luego, todos los vecinos se

construyen recursivamente. Por lo tanto, llamando $B(n)$ al costo de construir un *SA-tree* de n elementos, se obtiene la recurrencia

$$B(n) \;=\; n\log n + \log(n) B\left(\frac{n}{\log n}\right)$$

la que se resuelve como

$$B(n) \;=\; \Theta\left(\frac{n\log^2 n}{\log\log n}\right)$$

El espacio necesario por el índice (número de vínculos) es $O(n)$ porque es un árbol.

Como existen $\Theta(\log n)$ vecinos, y se entra en cada uno con la misma probabilidad, el tamaño del conjunto dentro de un vecino es $\Theta(n/\log n)$. De aquí, si llamamos $T(n)$ al costo de buscar con n elementos, se tiene que (ver Apéndice B de [Nav02])

$$T(n) \;=\; \Theta\left(n^{1-\Theta(1/\log\log n)}\right)$$

Esto muestra sublinealidad con respecto a n. Por otro lado, a medida que se incrementa el radio de búsqueda o se incrementa la dimensión intrínseca del espacio, el costo se aproxima al lineal.

Por otra parte, se hace notar que cuando el espacio es más sencillo (v.g. espacios de vectores con dimensión menor que $O(\log n)$), el número de vecinos $N(n)$ es más cercano a una constante porque no pueden existir tantos vecinos. En ese caso el análisis produce $T(n) = O(n^\alpha)$ para una constante $0 < \alpha < 1$. Aunque se prefiere, sin embargo, ser fiel a la complejidad más conservadora.

3.4. Resultados Experimentales

En [Nav02] se muestran resultados experimentales sobre el comportamiento del *SA-tree* sobre un conjunto sintético de puntos aleatorios en un espacio D-dimensional: cada coordenada fue elegida uniforme e independientemente en $[0, 1)$. Sin embargo, no se usa el hecho que el espacio tiene coordenadas, y se tratan los puntos como objetos abstractos en un espacio métrico desconocido. Esta elección permite controlar la dimensionalidad exacta (dificultad) con la que se trabaja, lo cual no se puede hacer fácilmente si el espacio es un espacio métrico general o los puntos provienen de una situación real (donde, a pesar de que están inmersos en un espacio D-dimensional, su dimensión real puede ser menor). Las pruebas se realizan utilizando distancia Euclídea (L_2) para cuatro dimensiones: 5, 10, 15 y 20. Luego, cuando se compara *SA-tree* contra otras estructuras, se muestran también espacios métricos reales.

Costo de Construcción y Forma del Árbol Los resultados muestran que el análisis realizado es bastante preciso, a pesar de las simplificaciones efectuadas. Por lo tanto, se puede predecir cómo se comportan los árboles como una función del tamaño de la base de datos n. Sin embargo, los experimentos brindan información adicional sobre un aspecto que no se captura analíticamente, como es el comportamiento de los árboles a medida que crece la dimensión del conjunto. Los árboles se vuelven más anchos y cortos en dimensiones más altas, y consecuentemente más difíciles de construir. Esto muestra cómo el *SA-tree* se adapta automáticamente a la dimensión de los datos sin necesidad de sintonización externa, una característica que aparece en muy pocas estructuras de datos. Otros artículos, tales como el de los *GNA-trees* [Bri95], sugieren usar una aridad grande para altas dimensiones y reducir la aridad en niveles más bajos del árbol, cosa que ocurre naturalmente en los *SA-trees*.

Costos de Consultas Se analizan ambos tipos de búsqueda, por rango y de vecino más cercano. Para búsqueda por rango se han seleccionado manualmente los radios que recuperan aproximadamente el 0.01 %, el 0.1 % y el 1 % del conjunto. Para las búsquedas de vecino más cercano, directamente se requiere el número de elementos. Queda claro en los experimentos la sublinealidad de las consultas, que el análisis realizado es bastante preciso, que los resultados empeoran rápidamente a medida que crece la dimensión o el radio de búsqueda y que el algoritmo de búsqueda de vecino más cercano está bastante cerca del correspondiente de búsqueda por rango, lo cual da una idea de que se utiliza una buena heurística en la búsqueda del vecino más cercano. Esto no es sorprendente porque ya se ha demostrado en [HS00] que este algoritmo de búsqueda de los k-vecinos más cercanos es de *rango-óptimo*[2].

3.4.1. Comparación contra otras Estructuras

Se compara el *SA-tree* contra otras estructuras de datos. Existen demasiadas propuestas para compararla contra todas, así que se selecciona un pequeño conjunto de buenas representantes. Algunas estructuras se comportan mejor que el *SA-tree*, pero a costa de cantidades imprácticas de memoria (v.g. *AESA* [Vid86] necesita espacio $O(n^2)$) o tiempo de construcción (v.g. *AESA* [Vid86] y *Lista de Clusters* [CN00a] necesitan tiempo de construcción $O(n^2)$). Las estructuras elegidas son:

- Pivotes(s): es un algoritmo genérico, donde se limita la cantidad de espacio permitido a s veces el del *SA-tree*.
- Clusters(t): es el esquema propuesto en [CN00a]. Esta estructura toma espacio lineal y se muestra que tiene mejor comportamiento en espacios de alta dimensión. Sin embargo, para que esto ocurra es necesario pagar un costo de construcción cuadrático, lo cual no es realista comparado aún contra el (ya costoso) costo de construcción del *SA-tree*. El parámetro t indica cuántas veces el tiempo de construcción de la *Lista de clusters* fue superior al del *SA-tree*, y los tamaños de los clusters se definen en función de t (de acuerdo al tiempo de construcción del *SA-tree*).
- GNA-tree: es una simplificación de la estructura propuesta en [Bri95]. Se selecciona aleatoriamente un conjunto de m centros y el resto de los elementos son enviados al subárbol de su centro más cercano. Se construyen los subárboles recursivamente. Durante la búsqueda la query se compara contra los m centros y se ingresa en el más cercano, c, y en aquellos cuya región de Voronoi intersecte con la bola de la query (es decir, $d(q, c_i) \leq d(q, c) + 2r$). Se usan los radios de cobertura también para incrementar la poda. Esta estructura utiliza espacio lineal y tiempo de construcción cercano al del *SA-tree*, por lo tanto no se le agrega ningún parámetro de sintonía. Más aún, se eligen manualmente los mejores m para cada caso. Los experimentos con la estructura propuesta en [Bri95] muestran que esta simplificación es indistinguible en desempeño.

Se concluye de los experimentos realizados en [Nav02] que *SA-tree*, respecto de los algoritmos de pivotes idealizados, tolera mejor los espacios más difíciles (alta dimensión) o los rangos más amplios (baja selectividad). Un índice de pivotes usando cuatro veces la cantidad de memoria que el *SA-tree* es más rápido sólo para dimensión 5 y un radio que recupera menos del 0.1 % de la base de datos. A medida que crecen la dimensionalidad o el radio de búsqueda, los algoritmos de pivotes necesitan más y más memoria para poder competir. En espacios de alta dimensión o radios de búsqueda amplios, los algoritmos de pivotes no pueden competir aún utilizando 64 veces la cantidad de memoria requerida por los *SA-trees*.

[2] Un algoritmo es de *rango-óptimo* si encuentra el k-ésimo vecino más cercano o_k de q después de visitar los mismos nodos del *SA-tree* que al realizar una búsqueda por rango con radio $d(q, o_k)$.

Los algoritmos de clustering toleran mejor los espacios de mayor dimensión y los radios de búsqueda más amplios, con una velocidad de crecimiento similar a la de los *SA-trees*. El *SA-tree* es mejor que el *GNA-tree* para dimensiones mayores que 10. *Lista de clusters* en cambio, necesita más y más tiempo de construcción que los *SA-trees* para superarlos a medida que crecen la dimensionalidad o el radio de búsqueda: 2 veces en dimensión 5, 4 veces en dimensión 10, 4 a 8 veces en dimensión 15, y 8 veces en dimensión 20.

Finalmente se comparan las estructuras usando un diccionario palabras en Español con distancia de edición o distancia de Levenshtein (ver Sección 2.4.3).

En este espacio bajo distancia de edición, el *SA-tree* supera al *GNA-tree*. Un algoritmo de pivotes necesita 8 veces más espacio que el *SA-tree* para superarlo cuando el radio de búsqueda es grande (3 o 4). La *Lista de clusters* necesita 4 veces el costo de construcción de los *SA-trees* para alcanzar una mejor eficiencia.

El segundo espacio métrico es el de documentos bajo distancia coseno (ver Sección 2.4.2).

En este espacio, la estructura con pivotes (aún usando 64 pivotes) y el *GNA-tree* tienen un pobre desempeño. El único competidor para el *SA-tree* es la *Lista de clusters*. Cuando se recuperan pocos elementos ésta necesita 8 veces más tiempo de construcción para superar a los *SA-trees*.

Como se puede ver en [Nav02], los *SA-trees* obtienen un buen balance entre eficiencia en las búsquedas y espacio/costo de construcción. Las otras estructuras necesitan utilizar mucho más espacio o tiempo de construcción para superarlos, cuando el espacio es de alta dimensión o los radios de búsqueda son amplios.

Capítulo 4

Construcción Incremental

Presentamos y analizamos en este capítulo diferentes opciones consideradas para construir incrementalmente un *SA-tree*. Algunas de ellas derivan de soluciones clásicas y bien conocidas en la comunidad de las estructuras de datos; aunque, las que resultaron más prometedoras y aquella elegida finalmente se han desarrollado específicamente basándose en las propiedades particulares del árbol e involucran una nueva visión algorítmica sobre el comportamiento de esta estructura de datos. Como resultado final mostramos la opción definitivamente elegida, con la que logramos inserciones rápidas, mantenemos la buena eficiencia de búsqueda del *SA-tree* en espacios de alta dimensión o consultas de baja selectividad y mejoramos su comportamiento en espacios de baja dimensión.

Como punto de comparación en lo que sigue, la construcción estática para el diccionario de 69069 palabras en Inglés cuesta cerca de 5 millones de comparaciones y 12,5 millones para el espacio de vectores de dimensión 15 uniformemente distribuidos en el cubo unitario.

4.1. Primeras Opciones Analizadas

El *SA-tree* es una estructura cuya construcción necesita conocer todos los elementos de S. En particular, es muy dificultoso agregarle nuevos elementos bajo la estrategia *best-fit* cuando el árbol ya está construido. Cada vez que se agrega un nuevo elemento, debemos bajar en el árbol siguiendo en cada paso el vecino más cercano hasta que el nuevo elemento deba volverse un vecino más del nodo corriente a de acuerdo a la Condición 1 (Sección 3.1), es decir que el nuevo elemento esté más cerca de a que de los vecinos de $N(a)$. Todo el subárbol cuya raíz es a se debe reconstruir completamente, dado que algunos nodos que entraron por otro vecino podrían ahora preferir entrar en el vecino nuevo.

En esta sección discutimos y evaluamos empíricamente las primeras alternativas consideradas para permitir inserciones de nuevos elementos en un árbol ya construido.

4.1.1. Reconstruyendo el Subárbol

Esta aproximación ingenua mantiene el *SA-tree* tal como si se hubiera generado completamente con el método estático original. Cuando se inserta un nuevo elemento x, de acuerdo a esta opción, se busca el lugar apropiado de inserción a en el árbol; es decir, el punto en el que x está más cerca de a que de sus vecinos en

$N(a)$ y por lo tanto x se debe convertir en vecino de a. Luego, se recuperan todos los elementos del subárbol con raíz a (como si se volviera a llenar la "bolsa" de a usada en la construcción estática) y se reconstruye el subárbol con raíz a con el mismo método estático descripto en Sección 3.2.1.

Por lo tanto, cuando se inserta un elemento x como un nuevo vecino de a, reconstruye el subárbol completo con raíz a. Esto se debe a que, si $N(a) = \{v_1, v_2, \ldots v_k\}$ antes de insertar a x, los vecinos de a y todos los elementos que se encuentran en los subárboles cuyas raíces son $v_1, v_2, \ldots v_k$, cuando buscaron su punto apropiado de inserción, no consideraron la nueva opción de elegir a x como vecino más cercano.

En la Figura 4.1 se muestra un ejemplo de una posible situación de un árbol con raíz a antes (arriba) y después (abajo) de la incorporación de un nuevo elemento x. A la izquierda de cada gráfica se ilustra la situación espacial y a la derecha el *SA-tree* correspondiente. Se puede apreciar en este ejemplo cómo, luego de la reconstrucción, se reubican los elementos para respetar que cada elemento sea vecino de aquel que está más próximo; pero ello implica volver a calcular las distancias de todos los elementos a a y reconstruir nuevamente el subárbol (tal como lo haría la versión estática).

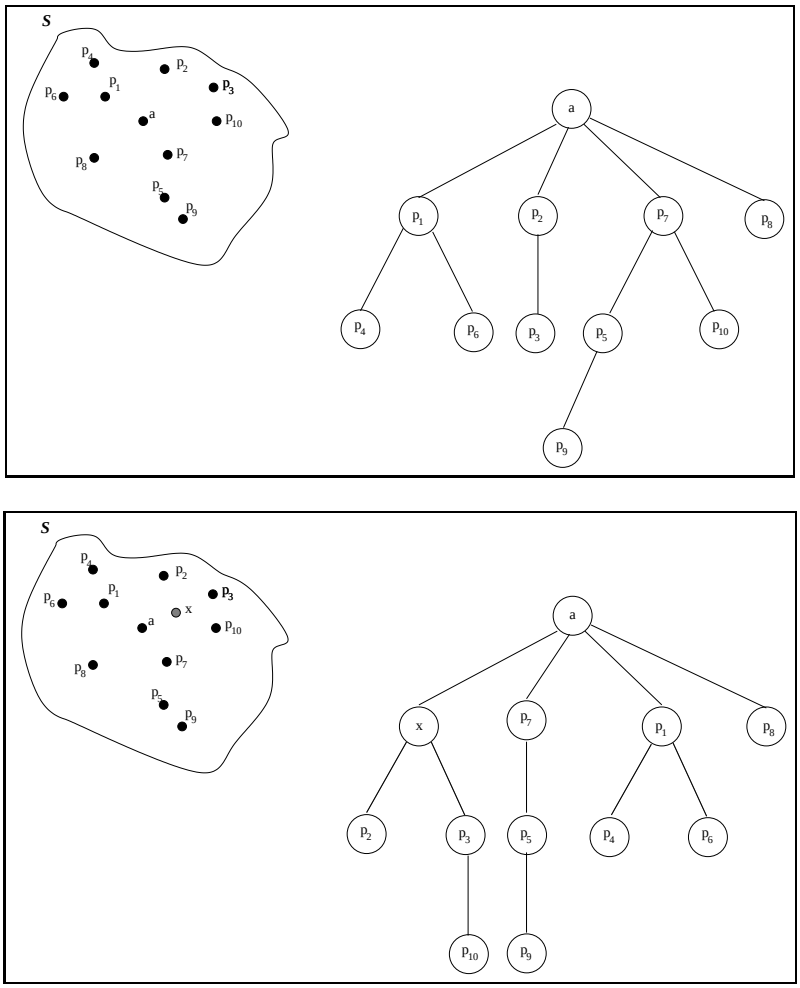


Figura 4.1: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la inserción del nuevo elemento x en un *SA-tree* con *reconstrucción de subárbol*.

Este método tiene la ventaja de preservar el mismo árbol que se hubiera construido estáticamente, pero

la construcción es demasiado costosa en relación a la construcción estática.

La Figura 4.2 muestra para el diccionario de palabras en Inglés y para el espacio de vectores de dimensión 15 que la construcción dinámica es demasiado costosa comparada con la estática (140 veces más costosa en el diccionario y casi 230 veces más en el espacio de vectores).

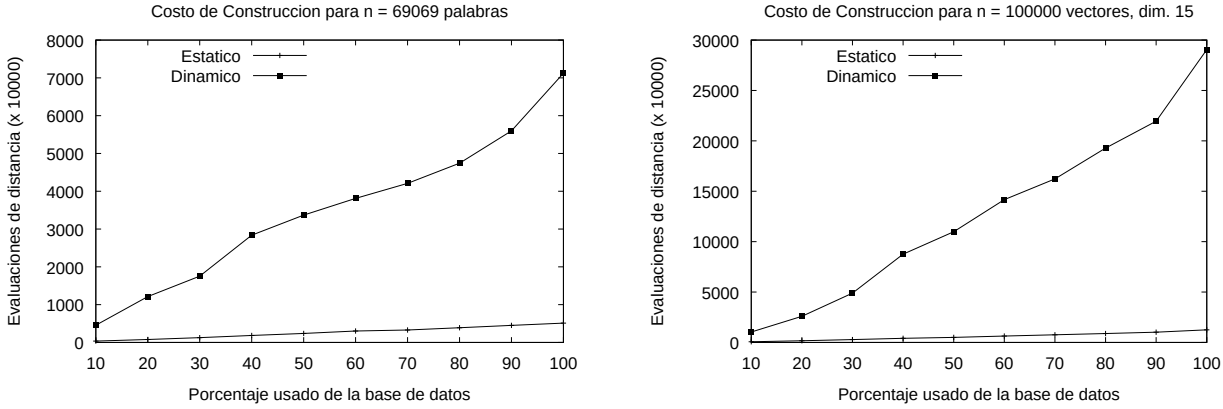


Figura 4.2: Costo de construcción reconstruyendo subárboles.

La Figura 4.3 muestra el algoritmo de inserción para esta opción de construcción incremental. Seguimos sólo un camino desde la raíz del árbol al padre del elemento insertado y luego debemos reconstruir el subárbol completo. El algoritmo se invoca como `InsertarRB(`a`,`x`)`, donde a es la raíz del árbol y x es el elemento a insertar. El *SA-tree* se puede construir ahora comenzando con un único primer nodo a donde $N(a) = \emptyset$ y $R(a) = 0$, y luego realizando sucesivas inserciones. El algoritmo `Construir(`a`,`S`)` que se invoca en la línea 8, es el utilizado para la construcción estática del *SA-tree* (ver Figura 3.3).

```
InsertarRB (Nodo a, Elemento x)

1.  c ← argmin_{b∈N(a)} d(b,x)
2.  If d(a,x) < d(c,x) Then
3.      N(a) ← N(a) ∪ {x}
4.      N(x) ← ∅,  R(x) ← 0
5.      Recuperar en S los elementos del subárbol con raíz a
6.      Construir (a,S) /* reconstruye el subárbol */
7.  Else
8.      R(a) ← máx{R(a), d(a,x)}
9.      InsertarRB (c,x)
```

Figura 4.3: Algoritmo de inserción de un nuevo elemento x en un *SA-tree* dinámico con raíz a, con la opción de reconstruir subárboles.

Claramente, como el árbol que se obtiene de acuerdo a esta alternativa es el mismo que se hubiera obtenido al construir el *SA-tree* estáticamente, para realizar las búsquedas se pueden utilizar los mismos algoritmos descriptos en la Figura 3.5 y en la Figura 3.6.

La Figura 4.4 ilustra los costos de consulta por rango y de k-vecinos más cercanos que se obtienen usando esta alternativa, en el espacio de palabras (arriba) y en el espacio de vectores en dimensión 15 (abajo). Como estos costos coinciden con los costos de consulta del *SA-tree* estático, sirven como referencia para lo que sigue.

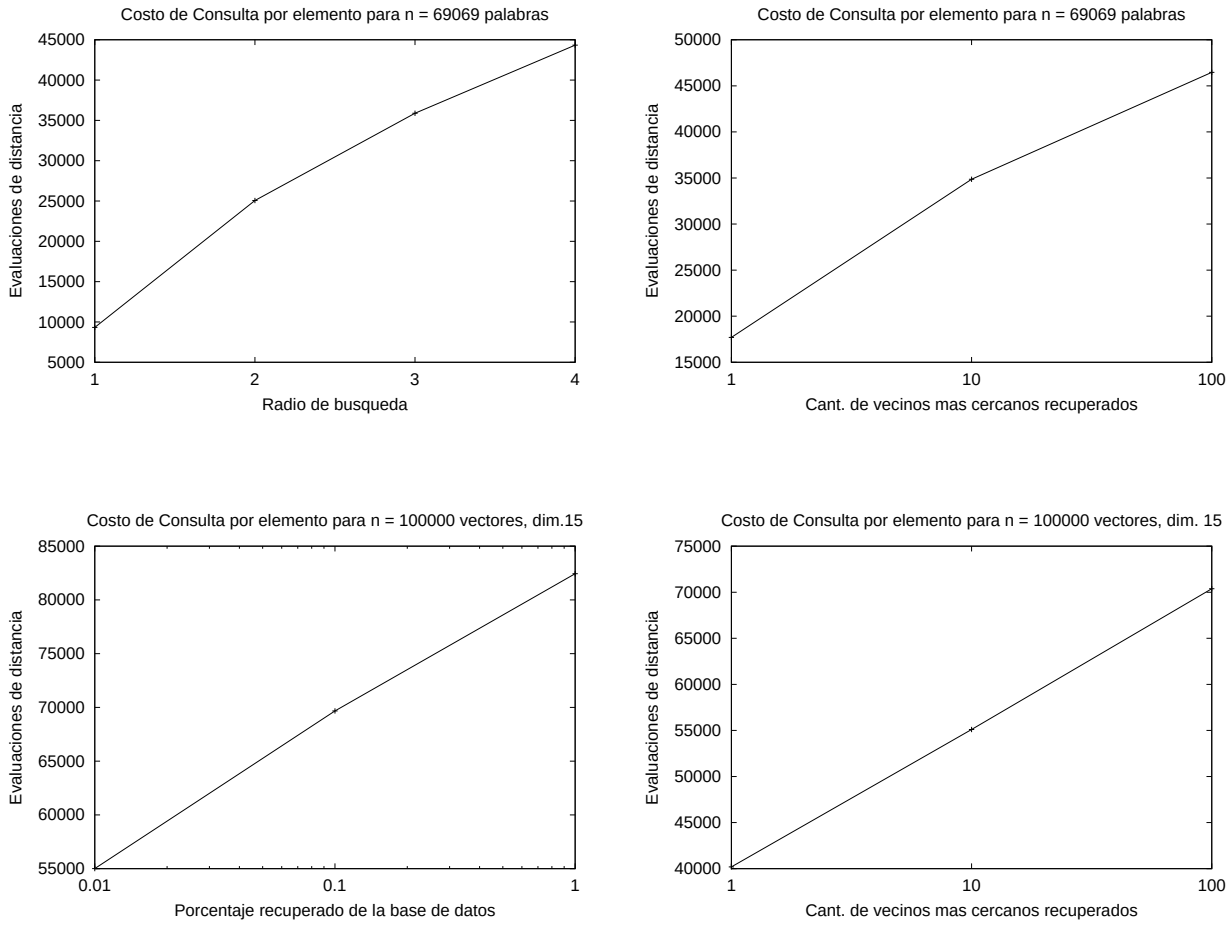


Figura 4.4: Costos de búsqueda usando *reconstrucción de subárboles*.

4.1.2. Área de Rebalse

Podemos tener en cada nodo un área de rebalse con vecinos "extra" que deberían estar en el subárbol, pero que no se han clasificado aún. Para insertar un nuevo elemento x se busca el punto apropiado a entre los elementos ya clasificados en el árbol. Cuando el nuevo elemento x se debe volver un vecino de un nodo a, lo ubicamos en el área de rebalse de a. Cada vez que alcanzamos a a durante una búsqueda, también comparamos al elemento buscado q (la query) contra cada elemento en su área de rebalse e informamos sobre cualquier elemento que esté suficientemente cerca.

Debemos limitar el tamaño de este área de rebalse para mantener razonable la eficiencia de la búsqueda y reconstruimos un subárbol cuando su área de rebalse excede un determinado tamaño. La principal pregunta es cuál es el balance en la práctica entre costo de construcción y costo de búsqueda. A medida que el área de rebalse sea más chica tendremos que reconstruir más a menudo el árbol y mejoramos los costos de búsqueda, pero el tiempo de construcción crecerá.

La Figura 4.5 ilustra el proceso de inserción de un elemento x en un *SA-tree* con raíz a, de acuerdo a la técnica que usa áreas de rebalse. El algoritmo se invoca como `InsertarAR(`a`,`x`)`. $MaxAR$ es el tamaño máximo permitido para el área de rebalse $AR()$ de un nodo. Seguimos sólo un camino desde la raíz del árbol al padre del elemento insertado. El *SA-tree* se puede construir ahora comenzando con un único primer nodo a donde $N(a) = \emptyset$, $AR(a) = \emptyset$ y $R(a) = 0$, y luego realizando sucesivas inserciones. El algoritmo `Construir(`a`,`S`)` que se invoca en la línea 7, es el utilizado para la construcción estática del *SA-tree* (ver Figura 3.3). En la línea 5 se recuperan todos los elementos del subárbol de a, incluyendo los que se encuentran pendientes de clasificación en las áreas de rebalse).

```
InsertarAR (Nodo a, Elemento x)

 1.  c ← argmin_{b∈N(a)} d(b,x)
 2.  If d(a,x) < d(c,x) Then
 3.     If |AR(a)| < MaxAR Then AR(a) ← AR(a) ∪ {x} /* se deja pendiente */
 4.     Else
 5.        Recuperar en S los elementos del subárbol con raíz a
 6.        S ← S ∪ {x}
 7.        Construir (a,S) /* reconstruye el subárbol */
 8.  Else
 9.     R(a) ← máx{R(a), d(a,x)}
10.     InsertarAR (c,x)
```

Figura 4.5: Algoritmo de inserción de un nuevo elemento x en un *SA-tree* dinámico con raíz a, usando áreas de rebalse.

En la Figura 4.6 se muestra el mismo espacio del ejemplo de Figura 4.1. Arriba se muestra una posible situación de un árbol con raíz a antes (arriba) y después (abajo) de la incorporación de un nuevo elemento x. Se ha considerado que cada nodo posee un área de rebalse de tamaño 2 y que el árbol se generó de acuerdo a la siguiente secuencia de inserciones: $a, p_1, p_2, \ldots, p_{10}$. A la izquierda de cada gráfica se ilustra la situación espacial y a la derecha el *SA-tree* correspondiente. Se puede apreciar en este ejemplo cómo en las áreas de rebalse quedan elementos sin clasificar.

Para ilustrar mejor cómo se construye el árbol, mostramos en Figura 4.7 la situación intermedia que se obtiene antes (arriba) y después (abajo) de incorporar al elemento p_8. El elemento p_8 debe ser vecino de a, pero a tiene su área de rebalse completa. Por lo tanto, se debe reconstruir el subárbol completo (se vacían

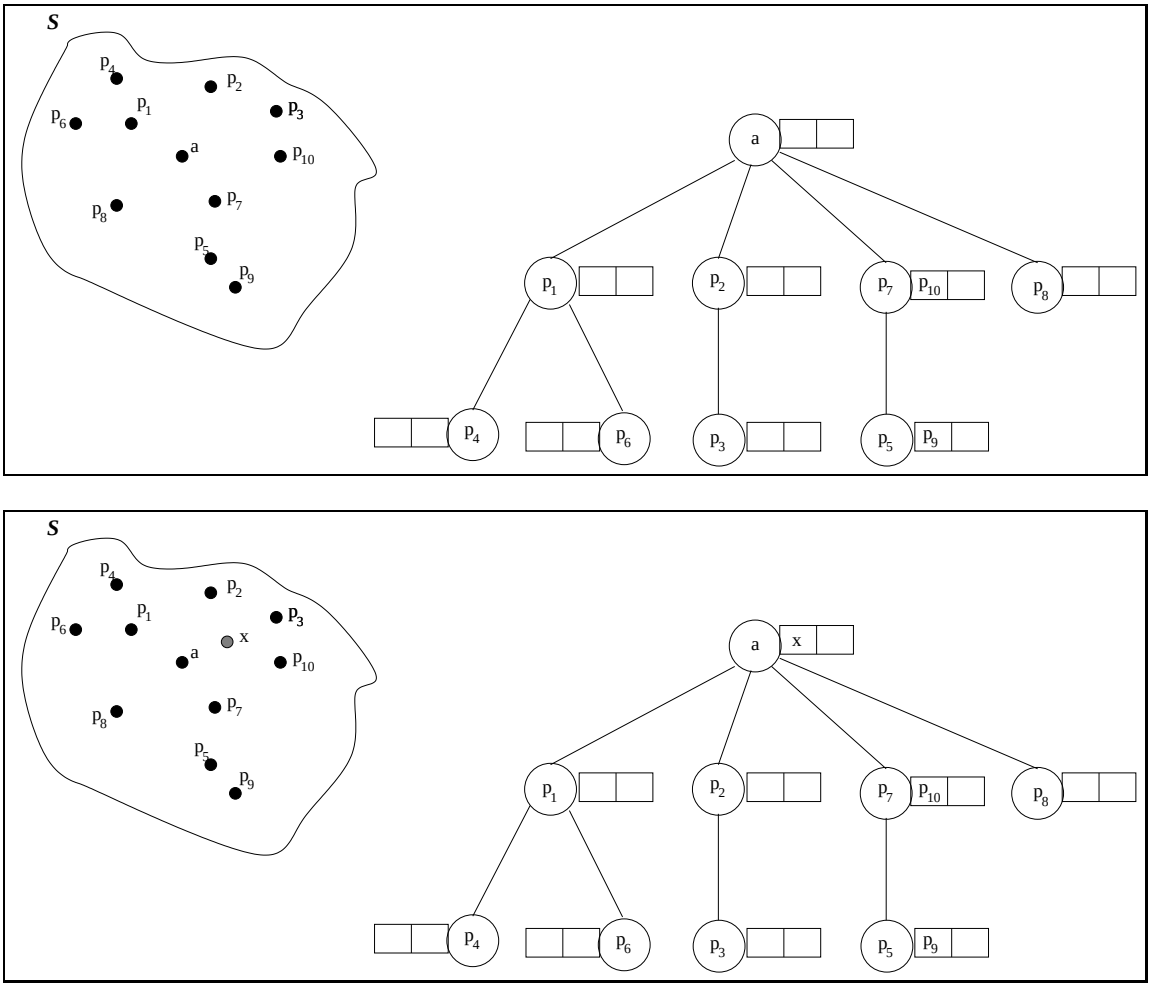


Figura 4.6: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la inserción del nuevo elemento x en un *SA-tree* con áreas de rebalse.

todas las áreas de rebalse y se clasifican todos los elementos pendientes). La reconstrucción aquí también implica volver a calcular las distancias de todos los elementos a a y reconstruir nuevamente el subárbol (tal como lo haría la versión estática).

En la Figura 4.8 se muestra el algoritmo de búsqueda para un *SA-tree* construido con la estrategia que utiliza *áreas de rebalse*. Se invoca por primera vez como `BúsquedaRangoAR(`a`,`q`,`r`,`$d(a,q)$`)` donde a es la raíz del árbol. Notar que, en invocaciones recursivas la distancia $d(a, q)$ ya está calculada. La única diferencia respecto del algoritmo presentado en la Figura 3.5, se encuentra en las líneas 3 y 4 en donde se debe comparar la query q contra todos los elementos en el área de rebalse ($AR()$) y se informan en la respuesta todos aquellos elementos que están suficientemente próximos a q.

La Figura 4.9 muestra el costo de construcción con diferentes tamaños para el área de rebalse, los cuales exhiben grandes fluctuaciones y en algunos casos aún cuestan menos que una construcción estática. Esto es posible porque muchos elementos se dejan en las áreas de rebalse sin clasificar. Por ejemplo, considerando el área de rebalse de tamaño 1000, casi todos los elementos están en el área de rebalse en el caso del diccionario y casi el 60 % en el caso del espacio de vectores de dimensión 15. Estas fluctuaciones aparecen debido a

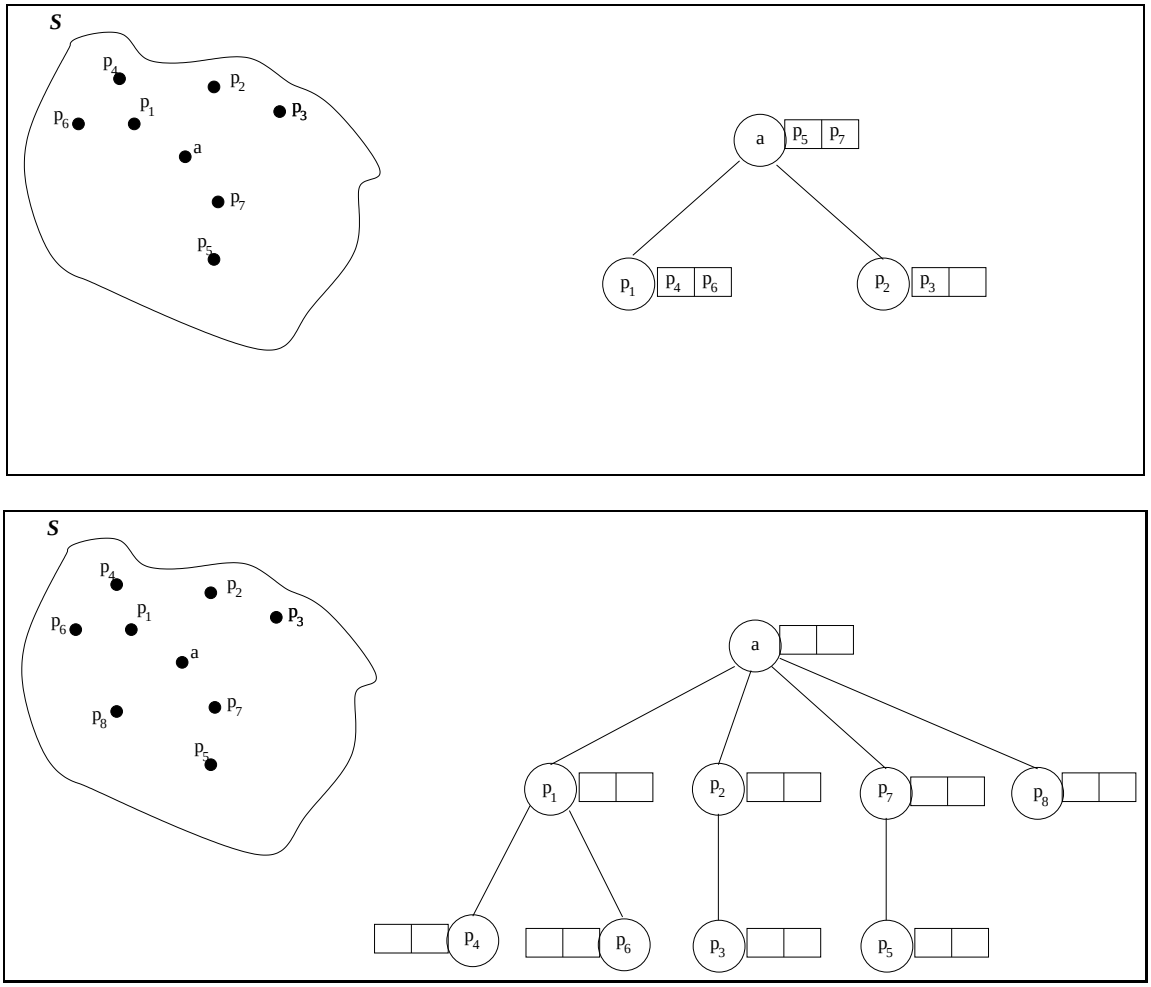


Figura 4.7: Ejemplo de la situación previa (arriba) y posterior (abajo) a la inserción de p_8 en el *SA-tree* de Figura 4.6.

que, para un tamaño dado de conjunto n, tamaños más grandes para el área de rebalse pueden producir más reconstrucciones que tamaños pequeños. El efecto es conocido, por ejemplo aparece cuando se estudia el número de divisiones (*split*) como una función del tamaño de página de un *B-tree* [BYL89].

La Figura 4.10 muestra, para cada uno de los tamaños considerados para el área de rebalse, cuántos elementos quedan aún sin clasificar. Esto permite apreciar cómo esto se relaciona tanto con los costos de búsqueda como con los de construcción.

La Figura 4.11 muestra los costos de búsqueda usando áreas de rebalse.

Como se puede observar comparando los resultados de la Figura 4.9 con aquellos de Figura 4.17, esta técnica es competitiva contra la construcción estática si se elige el tamaño correcto para el área de rebalse. Además, por ejemplo, con tamaño de 500 obtuvimos casi el mismo costo de búsqueda que con la versión estática, al modesto precio de 10 % de costo de construcción extra para el caso del diccionario y 30 % para el espacio de vectores de dimensión 15. El principal problema de este método es su alta sensibilidad a las fluctuaciones, las cuales hacen muy dificultoso seleccionar un buen tamaño para dicha área. El tamaño

```
BúsquedaRangoAR (Nodo a, Query q, Radio r, Distancia d_min)

1. If d(a,q) ≤ R(a) + r Then
2.     If d(a,q) ≤ r Then Informar a
3.     For b ∈ AR(a)
4.         If d(b,q) ≤ r Then Informar b
5.     d_min ← mín {d_min} ∪ {d(q,c), c ∈ N(a)}
6.     For b ∈ N(a)
7.         If d(b,q) ≤ d_min + 2r Then BúsquedaRangoAR (b,q,r,d_min)
```

Figura 4.8: Algoritmo para buscar q con radio r en un *SA-tree* construido usando *áreas de rebalse*.

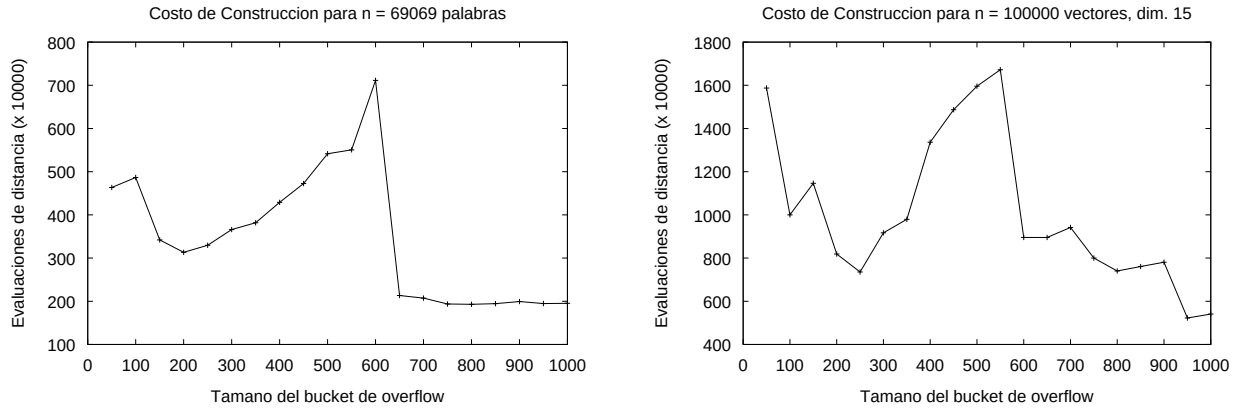


Figura 4.9: Costo de construcción usando área de rebalse.

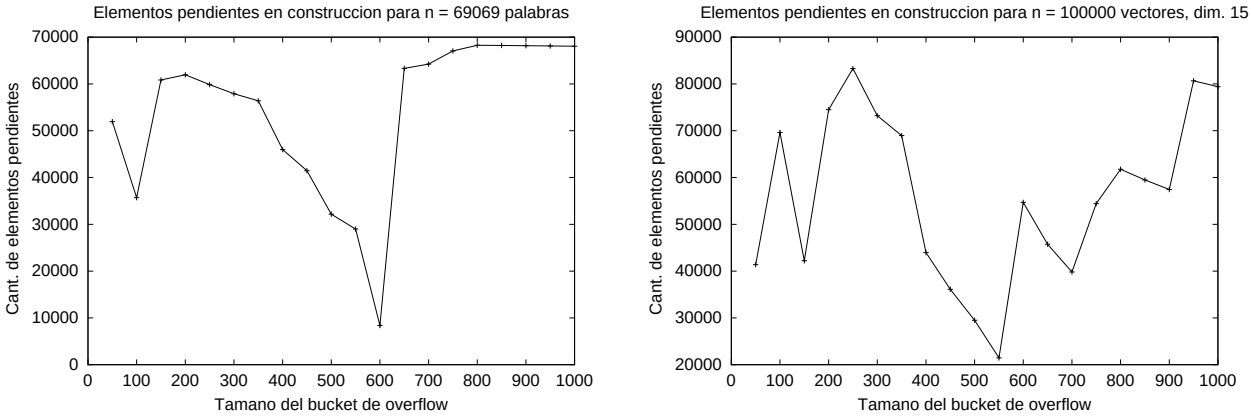


Figura 4.10: Cantidad de elementos pendientes de clasificar usando área de rebalse.

intermedio de 500 funciona bien, debido a que en ese caso en el área de rebalse está el 30 % de los elementos en el diccionario y el 15 % en el espacio de vectores de dimensión 15.

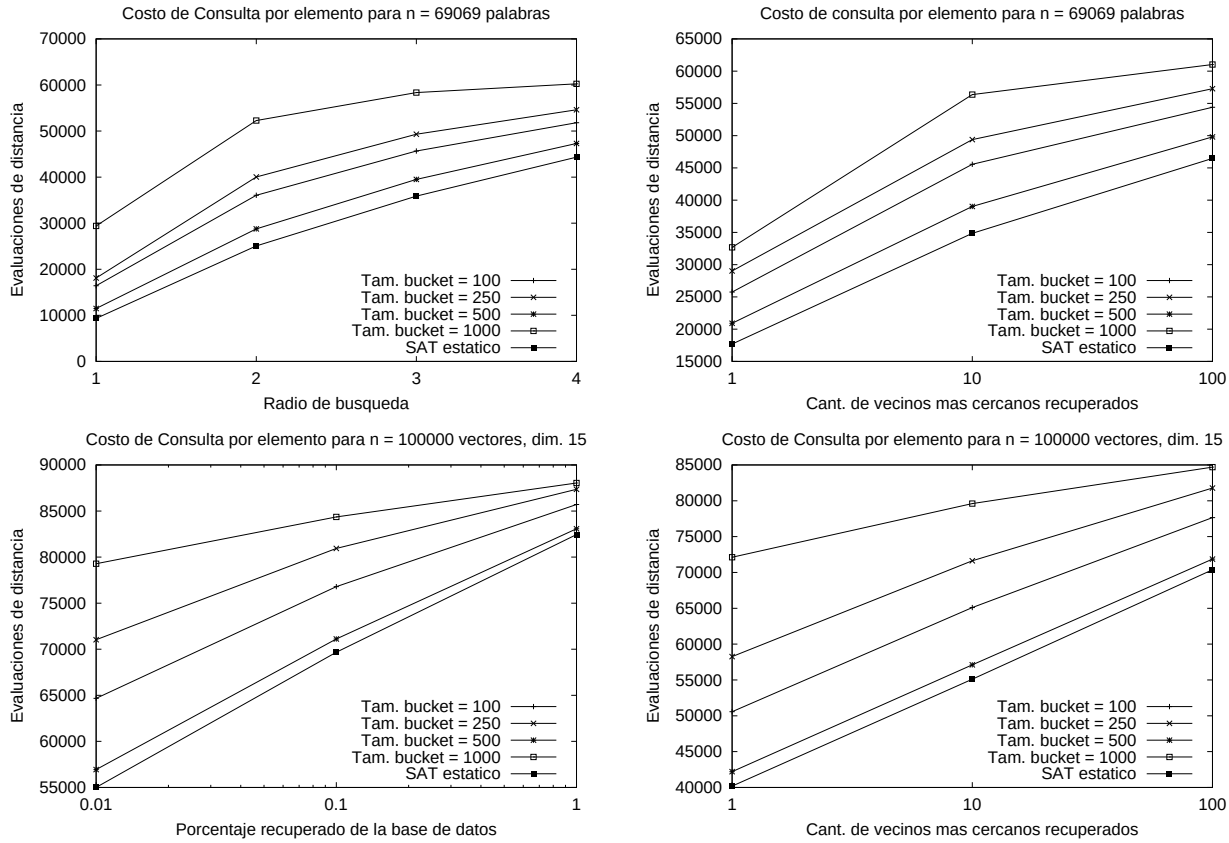


Figura 4.11: Costos de búsqueda usando áreas de rebalse.

4.1.3. Una Estrategia First-Fit

Otra solución posible es cambiar nuestra estrategia *best-fit* para poner elementos dentro de las "bolsas" de los vecinos de a durante la construcción. Una alternativa en la construcción estática, *first-fit*, es poner cada nodo en la bolsa del primer vecino más cercano que a a x. Ahora determinar $N(a)$ y la bolsa para cada uno de los otros elementos puede lograrse en una única pasada sobre los elementos del espacio.

Con la estrategia *first-fit*, sin embargo, podemos agregar fácilmente más elementos, tomando al nuevo elemento x como si hubiera sido el último en la bolsa, lo cual significa que cuando se vuelva un vecino de a se lo puede agregar simplemente como el último vecino de a; y no han habido elementos posteriores que tuvieran la posibilidad de entrar en x. Esto permite que la estructura se construya por sucesivas inserciones.

La Figura 4.12 ilustra el mismo espacio del ejemplo de Figura 4.1. Arriba se muestra una posible situación de un árbol con raíz a antes (arriba) y después (abajo) de la incorporación de un nuevo elemento x. Se ha considerado que el árbol se generó con la siguiente secuencia de inserciones: $a, p_1, p_2, \ldots, p_{10}$. A la izquierda de cada gráfica se ilustra la situación espacial y a la derecha el *SA-tree* correspondiente.

En la Figura 4.13 se encuentra el algoritmo que permite realizar la inserción de un elemento x en un *SA-tree* con raíz a, construido de acuerdo a la estrategia *first-fit*.

La Figura 4.17 muestra que la construcción (estática o dinámica) usando *first-fit* es mucho más económica que usando *best-fit*. Más aún, *first-fit* cuesta exactamente lo mismo y produce el mismo árbol en el caso

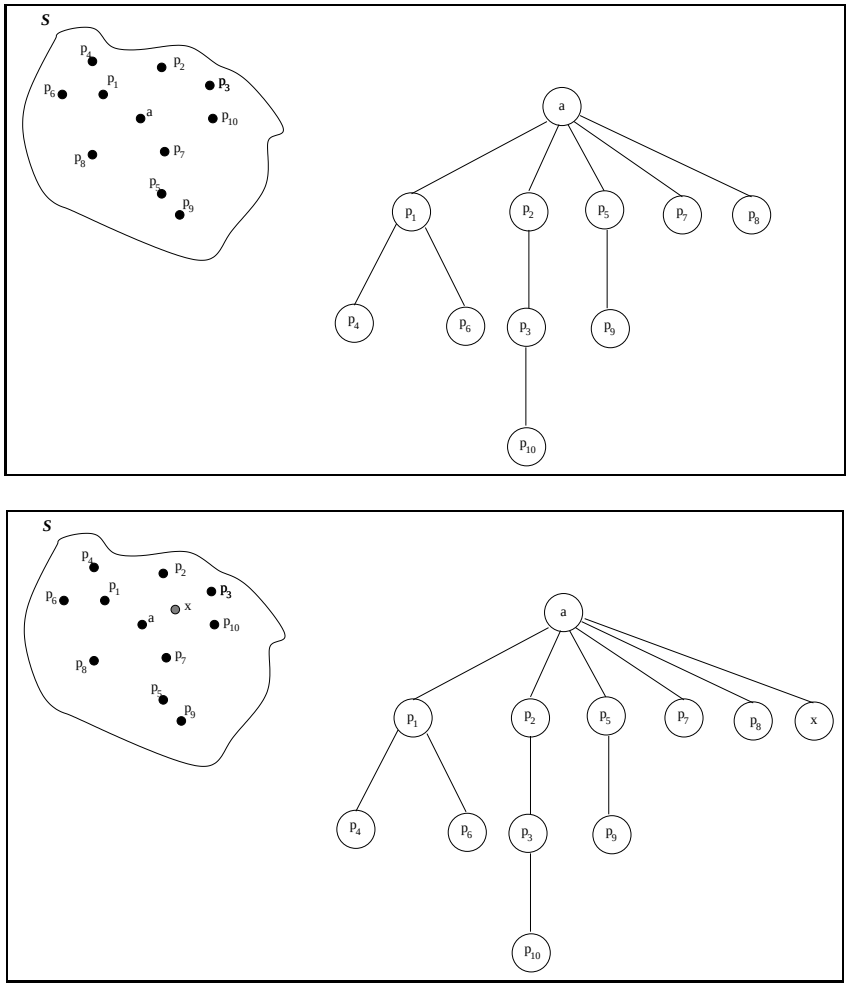


Figura 4.12: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la inserción del nuevo elemento x en un *SA-tree* con estrategia *first-fit*.

estático o dinámico.

La búsqueda por rango bajo la estrategia *first-fit* es un poco diferente. Consideramos los vecinos $\{v_1, \ldots, v_k\}$ de a en orden. Realizamos la minimización a medida que atravesamos los vecinos. Es decir, entramos en el subárbol de v_1 si $d(q, v_1) \leq d(q, a) + 2r$; en el subárbol de v_2 si $d(q, v_2) \leq \text{mín}\{d(q, a), d(q, v_1)\} + 2r$; y en general entramos en el subárbol de v_i si se cumple que $d(q, v_i) \leq \text{mín}\{d(q, a), d(q, v_1), \ldots, d(q, v_{i-1})\} + 2r$. Esto porque v_{i+j} nunca podrá contener un elemento que se insertaría en v_i.

La Figura 4.14 muestra el algoritmo de búsqueda considerando la creación (estática o dinámica) del *SA-tree* de acuerdo a la estrategia *first-fit*. Se invoca como `BúsquedaRangoFF(`$a, q, r, d(a, q)$`)`, donde a es la raíz del árbol. Notar que, en invocaciones recursivas, la distancia $d(a, q)$ ya está calculada. La línea 5 realiza la minimización a medida que atraviesa, en orden, los vecinos.

La Figura 4.19 muestra los tiempos de búsqueda. Como se puede observar, el costo adicional de la estrategia *first-fit* es demasiado alto, a tal punto que hace que la estructura no sea competitiva comparada

```
InsertarFF (Nodo a, Elemento x)

 1.  R(a) ← máx{R(a), d(a,x)}
 2.  c ← a
 3.  For v_i ∈ N(a) /* tomando los vecinos v_i en orden */
 4.     If d(v_i,x) ≤ d(a,x) Then
 5.        c ← v_i /* el primer vecino más cerca */
 6.        Break
 7.  If c = a Then
 8.     N(a) ← N(a) ∪ {x}
 9.     N(x) ← ∅, R(x) ← 0
10.  Else
11.     InsertarFF (c,x)
```

Figura 4.13: Algoritmo de inserción de un nuevo elemento x en un *SA-tree* dinámico con raíz a, usando la estrategia *first-fit*.

```
BúsquedaRangoFF (Nodo a, Query q, Radio r, Distancia d_min)

1. If d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Informar a
3.    For v_i ∈ N(a)   /* considerando en orden los vecinos v_1,...,v_k */
4.       If d(v_i,q) ≤ d_min + 2r Then BúsquedaRangoFF (v_i,q,r,d_min)
5.       d_min ← mín{d_min, d(q,v_i)}
```

Figura 4.14: Algoritmo para buscar q con radio r en un *SA-tree* construido con estrategia *first-fit*.

contra otras existentes.

4.1.4. Timestamp

Una alternativa que tiene cierto parecido con la previa, pero que es más sofisticada, consiste en mantener un *timestamp* del tiempo de inserción de cada elemento. Cuando insertamos un nuevo elemento, lo agregamos como vecino en el punto apropiado usando la estrategia *best-fit* y *no* reconstruimos el árbol. Consideramos que los vecinos se agregan al final, así leyéndolos de izquierda a derecha tenemos tiempos crecientes de inserción. También se mantiene que el padre es siempre más viejo que sus hijos.

La Figura 4.15 muestra el mismo espacio del ejemplo de Figura 4.1. Arriba se muestra una posible situación de un árbol con raíz a antes (arriba) y después (abajo) de la incorporación de un nuevo elemento x. Se ha considerado que la secuencia de inserciones: $a, p_1, p_2, \ldots, p_{10}$ para construir el árbol. En cada nodo se indica el timestamp del momento de inserción.

La Figura 4.16 ilustra el proceso de inserción. Seguimos sólo un camino desde la raíz del árbol al padre del elemento insertado. La función se invoca como `InsertarTS(`a`,`x`)`, donde a es la raíz del árbol y x es el elemento a insertar. El *SA-tree* se puede construir ahora comenzando con un único primer nodo a donde $N(a) = \emptyset$, $timestamp(a) = 1$ y $R(a) = 0$, y luego realizando sucesivas inserciones. $CurrentTime$ es el tiempo actual, el cual se incrementa luego de cada nueva inserción.

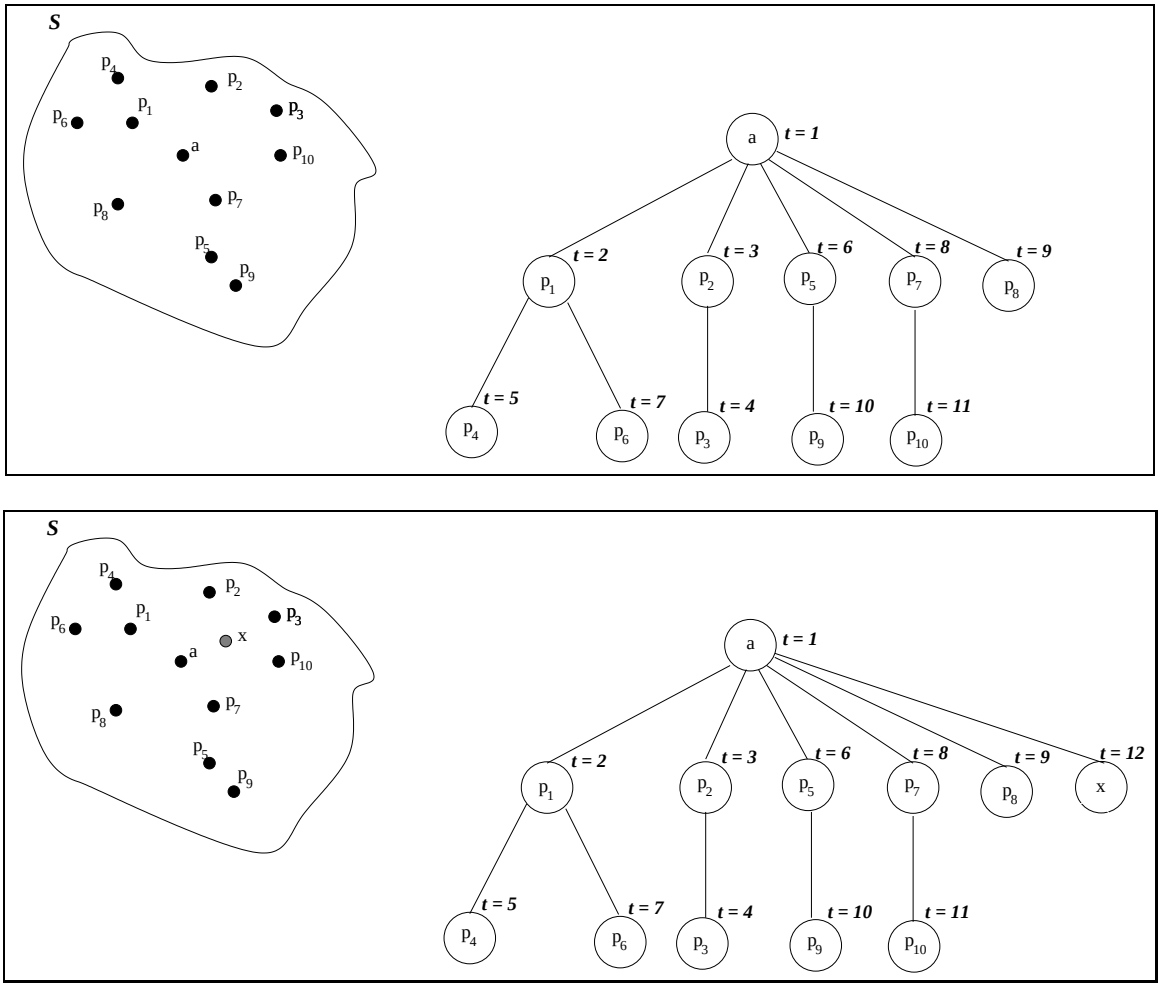


Figura 4.15: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la inserción del nuevo elemento x en un *SA-tree* con la técnica de *timestamp*.

Como se ve en la Figura 4.17, la construcción con esta alternativa puede costar un poco más o un poco menos que la versión estática dependiendo del caso. Las dos versiones de este método consideradas, rotuladas como “arriba” y “abajo” en la gráfica, corresponden a qué hacemos cuando insertamos un nuevo elemento si las distancias a la raíz y al vecino más cercano son iguales. La primera inserta el elemento como un nuevo vecino y la segunda lo inserta en el subárbol del vecino más cercano. Esto hace una diferencia sólo cuando se trabaja con funciones de distancia discretas. En todo caso optamos por la versión “abajo” de aquí en adelante, salvo que indiquemos lo contrario.

Durante la búsqueda, consideramos los vecinos $\{v_1, \ldots, v_k\}$ de a de más viejo a más nuevo. Realizamos la minimización a medida que atravesamos los vecinos, exactamente como en la Sección 4.1.3. Esto es porque entre la inserción de v_i y la de v_{i+j} pueden haber aparecido nuevos elementos que prefirieron a v_i sólo porque v_{i+j} no era aún un vecino, así podemos perder un elemento si no entramos al subárbol de v_i debido a la existencia de v_{i+j}.

Notar que, aunque el proceso de búsqueda es el mismo que para *first-fit*, la inserción coloca los elementos en su subárbol más cercano, por ello la estructura es más balanceada.

```
InsertarTS (Nodo a, Elemento x)

1.  R(a) ← máx{R(a), d(a,x)}
2.  c ← argmin_{b∈N(a)} d(b,x)
3.  If d(a,x) < d(c,x) Then
4.      N(a) ← N(a) ∪ {x}
5.      N(x) ← ∅,  R(x) ← 0
6.      timestamp(x) ← CurrentTime
7.  Else InsertarTS (c,x)
```

Figura 4.16: Inserción de un nuevo elemento x en un *SA-tree* dinámico con raíz a, usando la técnica de *timestamp*.

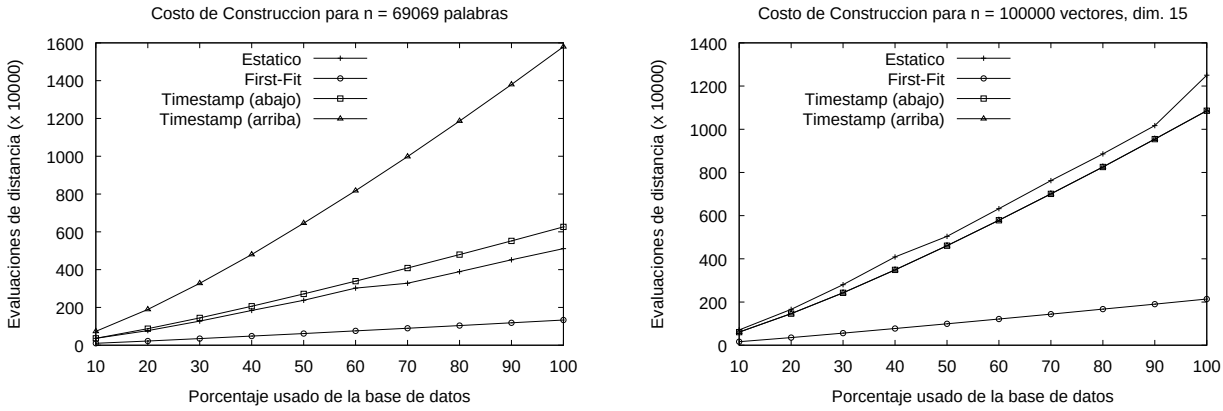


Figura 4.17: Costos de construcción usando la versión estática, *first-fit* y *timestamps*.

Hasta ahora necesitábamos realmente los timestamps sólo para mantener los vecinos ordenados. Un esquema aún más sofisticado es usar los timestamps para reducir el trabajo a realizar dentro de los vecinos más viejos. Digamos que $d(q, v_i) > d(q, v_{i+j}) + 2r$. Tenemos que entrar en v_i debido a que es más viejo. Sin embargo, sólo los elementos con timestamp más pequeño que el de v_{i+j} se deberían considerar cuando buscamos dentro de v_i; los elementos más nuevos ya han visto a v_{i+j} y no pueden estar dentro de v_i. Como los nodos padres son más viejos que sus descendientes, tan pronto como encontremos un nodo dentro del subárbol de v_i con timestamp más grande que el de v_{i+j} podemos detener la búsqueda en esa rama, debido a que su subárbol es aún más joven. Observar que ese timestamp máximo se hereda de padres a hijos.

La Figura 4.18 ilustra el algoritmo de búsqueda por rango considerando la creación del *SA-tree* usando la estrategia de *timestamp*. La minimización que realiza la línea 7 se realiza a medida que atraviesa en orden creciente de timestamp los vecinos del nodo considerado. Se invoca por primera vez como `BúsquedaRangoTS(`a`,`q`,`r`,`$d(a,q)$`,`$CurrentTime$`)` dónde a es la raíz del árbol. Notar que, en invocaciones recursivas la distancia $d(a, q)$ ya está calculada.

La Figura 4.19 compara esta técnica contra la estática. Como se puede observar, ésta es una excelente alternativa a la construcción estática en el caso del espacio de vectores de dimensión 15, dando básicamente los mismos costos de construcción y búsqueda con el valor agregado del dinamismo. En el caso del diccionario, la técnica que usa timestamp es significativamente peor que la estática (aunque la versión "arriba" se comporta levemente mejor para la búsqueda de los vecinos más cercanos). El problema es que la versión "arriba" es mucho más costosa de construir, ya que necesita más de 3 veces el costo de construcción estática.

```
BúsquedaRangoTS (Nodo a, Query q, Radio r, Distancia d_min, Timestamp t)

1. If timestamp(a) < t  ∧  d(a,q) ≤ R(a) + r  Then
2.     If d(a,q) ≤ r  Then Informar a
3.     For v_i ∈ N(a)    /* en orden creciente de timestamp */
4.         If d(v_i,q) ≤ d_min + 2r  Then
5.         k ← mín{timestamp(v_j), j > i ∧ d(v_i,q) > d(v_j,q) + 2r} ∪ {CurrentTime}
6.         BúsquedaRangoTS (v_i, q, r, d_min, mín{t,k})
7.         d_min ← mín{d_min, d(q,v_i)}
```

Figura 4.18: Algoritmo para buscar q con radio r en un *SA-tree* construido con la técnica de *timestamp*.

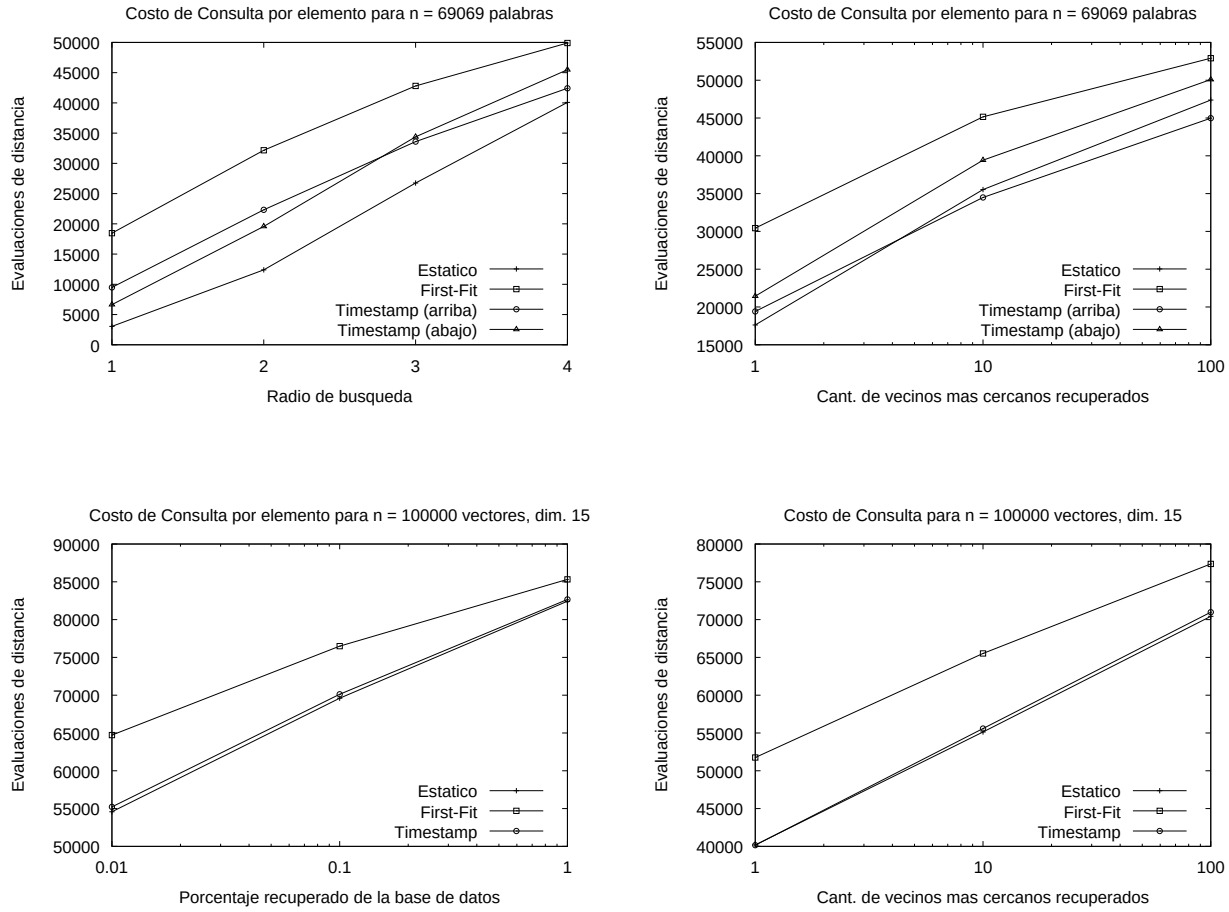


Figura 4.19: Costos de búsqueda usando la versión estática, *first-fit* y las dos versiones de la técnica de *timestamp*.

4.1.5. Insertando cerca de las Hojas

Otra alternativa es como sigue. Podemos relajar la Condición 1 (Sección 3.2.1), cuyo principal objetivo es garantizar que si q está más cerca de a que de cualquier vecino en $N(a)$ entonces podamos detener la

búsqueda en ese punto. La idea es que, durante la búsqueda, en lugar de encontrar el c más cercano entre $\{a\} \cup N(a)$ y entrar en cada $b \in N(a)$ tal que $d(q,b) \leq d(q,c) + 2r$, excluimos la raíz del subárbol $\{a\}$ de la minimización. Por lo tanto, continuamos *siempre* hasta las hojas por el vecino más cercano y por otros suficientemente cerca. Esto parece empeorar levemente el tiempo de la búsqueda, pero el costo es mínimo.

El beneficio es que ya no estamos forzados a insertar un nuevo elemento x como vecino de a. Aunque x esté más cerca de a que de cualquier elemento en $N(a)$, tenemos la opción de no ponerlo como vecino de a sino insertarlo en su vecino más cercano de $N(a)$. Durante la búsqueda alcanzaremos a x porque los procesos de búsqueda e inserción son similares.

Esta libertad abre nuevas posibilidades en la estructuración del árbol que estudiamos más adelante, pero la consecuencia inmediata es que podemos insertar siempre cerca de las hojas del árbol. Por lo tanto, el árbol es sólo de lectura en su parte superior y cambia sólo cerca de las hojas.

Sin embargo, permitimos la reconstrucción de pequeños subárboles para evitar que el árbol se transforme en una lista. Así, podemos insertar x como vecino cuando el tamaño del subárbol a reconstruir es suficientemente pequeño, lo cual nos lleva a mantener un balance entre costo de inserción y calidad del árbol para la búsqueda.

En la Figura 4.20 se muestra, en el espacio del ejemplo de Figura 4.1, una posible situación de un árbol con raíz a antes (arriba) y después (abajo) de la incorporación de un nuevo elemento x. Se ha considerado que el tamaño máximo de subárbol que se permite reconstruir es de 3 elementos y que el árbol se generó de acuerdo a la siguiente secuencia de inserciones: $a, p_1, p_2, \ldots, p_{10}$. Se puede apreciar en este ejemplo cómo el árbol crece cerca de las hojas y la parte superior permanece inalterada.

La Figura 4.21 muestra el algoritmo para insertar un elemento x en un *SA-tree* dinámico con raíz a, construido con la técnica de *inserción cerca de las hojas*. Se invoca como `InsertarH(`a`,`x`)`. $MaxTree$ es el máximo tamaño de subárbol que se permite reconstruir y $size(a)$ es el tamaño del subárbol con raíz a. Seguimos sólo un camino desde la raíz del árbol al padre del elemento insertado. El *SA-tree* se puede construir ahora comenzando con un único primer nodo a donde $N(a) = \emptyset$ y $R(a) = 0$, y luego realizando sucesivas inserciones. En línea 6 se invoca el algorimo `Construir(`a`,`S`)` que es el utilizado para la construcción estática del *SA-tree* (ver Figura 3.3) y realiza la reconstrucción del subárbol.

La Figura 4.22 muestra el costo de construcción para diferentes tamaños máximos de árbol que se puedan reconstruir ($MaxTree$). Como se puede observar, permitir un tamaño de árbol de 50 produce el mismo costo de construcción de la versión estática.

En la Figura 4.23 se presenta el algoritmo de búsqueda por rango considerando que el *SA-tree* se ha construido con la estrategia de *inserción cerca de las hojas*. Se invoca por primera vez como `BúsquedaRangoH(`a`,`q`,`r`)` dónde a es la raíz del árbol. El código es similar al del algoritmo de búsqueda por rango del *SA-tree* original (ver Figura 3.5), pero aquí en la línea 3 se hace la minimización sólo sobre los vecinos del nodo. Notar que d_min no se hereda porque ahora no estamos seguros de que un nuevo elemento x es vecino del primer nodo a que satisface la Condición 1 en su camino. Es posible que el tamaño del subárbol de a fuera mayor que el tamaño máximo que se permite reconstruir ($MaxTree$) y x fuera forzado a elegir un vecino de a.

Finalmente, la Figura 4.24 muestra los tiempos de búsqueda utilizando esta técnica. Se puede ver que, usando un tamaño de árbol de 50 se obtiene el mismo tiempo de búsqueda o aún mejor comparado con la versión estática, lo cual muestra que puede ser beneficioso mover elementos en el árbol hacia abajo. Este hecho hace que esta alternativa sea muy interesante y que se estudie más a fondo en la Sección 4.2.

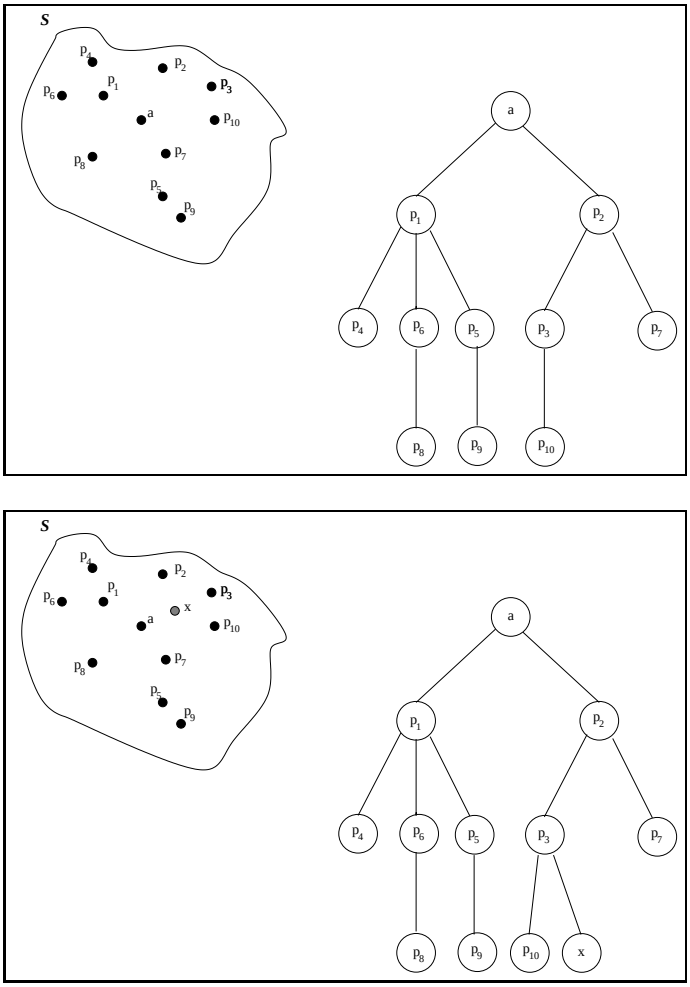


Figura 4.20: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la inserción del nuevo elemento x en un *SA-tree* con la técnica de *inserción cerca de las hojas*.

4.1.6. Análisis de las Alternativas

Hasta aquí hemos presentado varias técnicas que hacían posible modificar un *SA-tree* para que fuera una estructura de datos dinámica soportando inserciones y sin degradar su desempeño. Además hemos mostrado que los invariantes del *SA-tree* se pueden relajar de maneras no previstas antes de este estudio, por ejemplo tenemos la opción de agregar o no vecinos.

A partir de las opciones analizadas habíamos considerado que el uso de *áreas de rebalse* mostraba que era posible obtener tiempos de construcción y búsqueda similares a los de la versión estática, aunque había que estudiar más la elección del tamaño de dicha área. *Timestamp* también había mostrado ser una opción competitiva en algunos espacios métricos y poco atractiva en otros, otro hecho que requería estudio. Finalmente *insertar cerca de las hojas* había mostrado el potencial de incluso mejorar el desempeño de la versión estática, aunque también requería estudiar el efecto del tamaño del subárbol que se permitía reconstruir.

Claramente otras alternativas, tales como *reconstrucción* y *first-fit* no han demostrado ser competitivas,

```
InsertarH (Nodo a, Elemento x)

1.  c ← argmin_{b∈N(a)} d(b,x)
2.  If d(a,x) < d(c,x)  ∧  size(a) < MaxTree  Then
3.      N(a) ← N(a) ∪ {x}
4.      N(x) ← ∅,  R(x) ← 0
5.      Recuperar en S los elementos del subárbol con raíz a
6.      Construir (a,S) /* reconstruye el subárbol */
7.  Else
8.      R(a) ← máx{R(a), d(a,x)}
9.      InsertarH (c,x)
```

Figura 4.21: Inserción de un nuevo elemento x en un *SA-tree* dinámico con raíz a, usando la técnica de *inserción cerca de las hojas*.

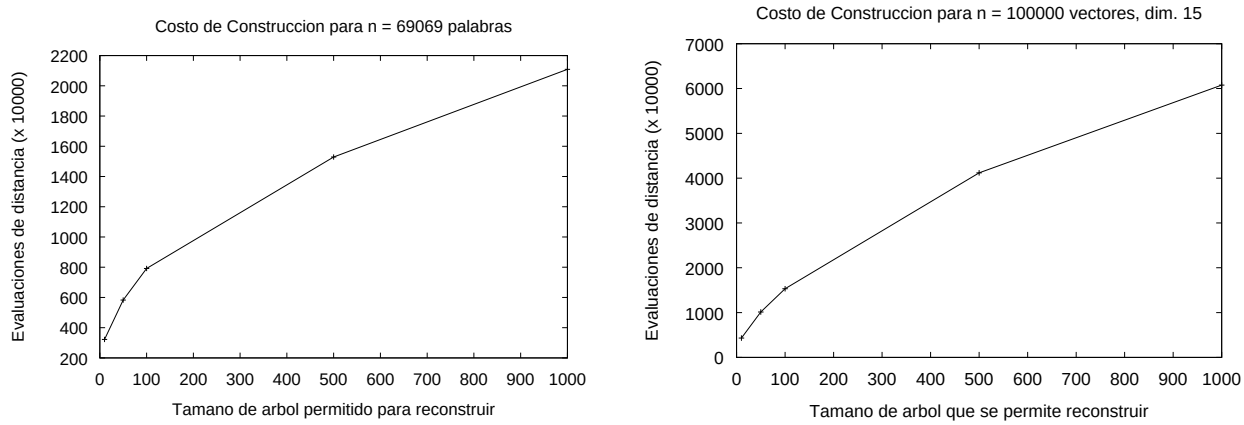


Figura 4.22: Costos de construcción insertando cerca de las hojas.

```
BúsquedaRangoH (Nodo a, Query q, Radio r)

1.  If d(a,q) ≤ R(a) + r  Then
2.      If d(a,q) ≤ r  Then Informar a
3.      d_min  ←  mín  {d(q,v_i), v_i  ∈ N(a)}
4.      For  v_i ∈ N(a)
5.          If  d(v_i,q) ≤ d_min + 2r  Then BúsquedaRangoH (v_i,q,r)
```

Figura 4.23: Algoritmo para buscar q con radio r en un *SA-tree* construido con la estrategia de *inserción cerca de las hojas*.

aunque la última da muy bajos costos de construcción, lo cual podría ser de interés en espacios de baja dimensión, donde su poca efectividad en la búsqueda puede ser tan crítica.

El estudio de estas primeras opciones puso en evidencia que era posible convertir al *SA-tree* en dinámico y que la estructura aún podría mejorar en un esquema dinámico, contrariamente a lo que suponíamos previamente de que deberíamos pagar un costo adicional por el dinamismo. Por otro lado, necesitábamos analizar más las alternativas más prometedoras para entenderlas mejor. De este análisis surgió una alternativa que combina las estudiadas pero las supera a todas, que se detalla en la siguiente sección.

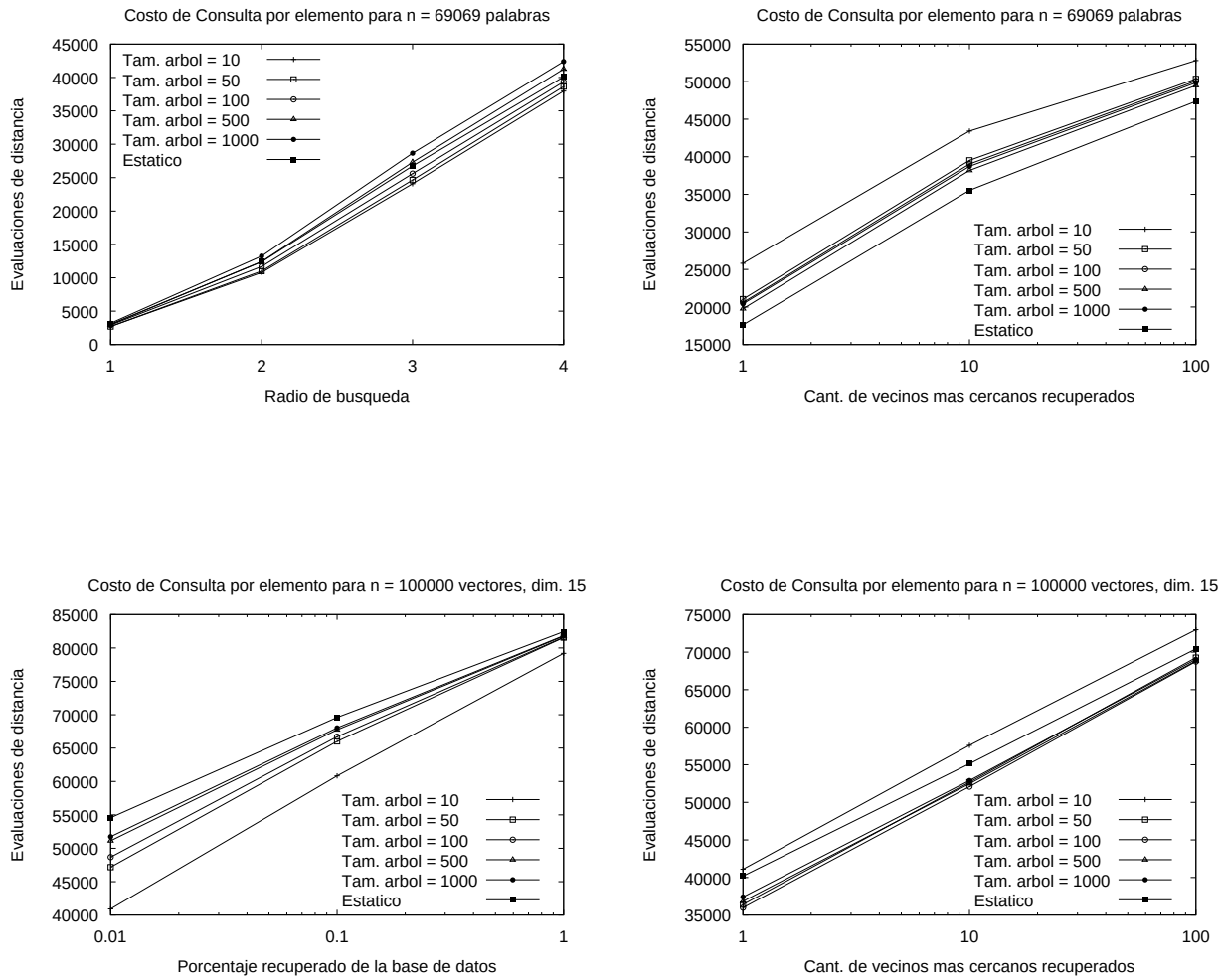


Figura 4.24: Costos de búsqueda *insertando cerca de las hojas*.

4.2. Una Nueva Técnica de Construcción Incremental

Timestamp permitía insertar un elemento con una técnica bastante similar a la usada para la construcción estática, registrando el tiempo en que se insertó cada elemento. Cabe destacar que esta técnica obtuvo un desempeño muy similar a la versión estática, sin realizar ninguna reconstrucción. *Insertar cerca de las hojas*, por otro lado, limitaba el tamaño máximo de árbol a reconstruir (es decir, donde se podía insertar un nuevo elemento), con el objetivo de reconstruir sólo pequeños subárboles. La técnica permitía evitar que las inserciones se realizaran en el punto que requeriría la Condición 1, difiriéndolas a un punto más bajo en el árbol. Sorpresivamente esta técnica aún *mejoraba* el desempeño en bajas dimensiones, lo que fue un factor que compensaba ampliamente el costo de las reconstrucciones.

Seguimos esta línea y determinamos que el hecho clave era que estos árboles tenían una aridad reducida. Más aún, la principal razón del pobre desempeño del *SA-tree* en espacios de baja dimensión es su excesiva aridad (el árbol automáticamente adapta su aridad a la dimensión, pero no óptimamente). Por lo tanto, decidimos enfocarnos directamente sobre la aridad máxima permitida y convertirla en un parámetro a sintonizar. Usamos la misma técnica para limitar el tamaño del árbol a reconstruir que para limitar ahora la aridad del árbol. Más aún, uniendo esta técnica con *timestamp* no tenemos que compensar costos de reconstrucción y así obtenemos lo mejor de ambas técnicas.

Observamos que uno de los aspectos más agradables del *SA-tree* original era no tener un parámetro que sintonizar, así cualquier inexperto podría usarlo. Nuestro nuevo parámetro no es perjudicial en este sentido, debido a que se puede tomar como ∞ para obtener el mismo desempeño que el *SA-tree* original [1]. Además, hemos obtenido grandes mejoras en bajas dimensiones eligiendo adecuadamente la máxima aridad del árbol. A continuación detallamos esta nueva estructura, llamada *timestamp con aridad limitada*.

4.2.1. Inserción

Para construir incrementalmente el *SA-tree* fijamos una aridad máxima para el árbol, y también mantenemos un *timestamp* del tiempo de inserción de cada elemento. Cuando insertamos un nuevo elemento x, lo agregamos como un vecino en el punto apropiado a (Condición 1) sólo si la aridad del nodo a no es ya la máxima. En otro caso, aún cuando x esté más cerca de a que de cualquier $b \in N(a)$, forzamos a x a que elija el vecino más cercano en $N(a)$ y vamos hacia abajo en el árbol, hasta alcanzar un nodo a donde se satisfaga la Condición 1 (x esté más cerca de a que cualquier $b \in N(a)$) y la aridad del nodo a no sea la máxima (esto eventualmente ocurre en una hoja). En este punto agregamos x al final de la lista de $N(a)$, le colocamos el timestamp actual a x e incrementamos el timestamp actual.

La Figura 4.25 muestra, en el mismo ejemplo de Figura 4.1, la situación antes (arriba) y después (abajo) de insertar un nuevo elemento x en un *SA-tree* dinámico con raíz a, construido con la nueva alternativa de *timestamp con aridad limitada*. Para este ejemplo hemos considerado una aridad máxima de 3 y que la secuencia de inserciones es nuevamente $a, p_1, p_2, \ldots, p_{10}$. En cada nodo se indica el timestamp del tiempo de inserción.

Recordar que leyendo los vecinos de izquierda a derecha tenemos timestamps crecientes y que se cumple que el padre es siempre más antiguo que sus hijos. Notar también que ahora, al igual que con *inserción cerca de las hojas*, no estamos seguros de que un nuevo elemento x es vecino del primer nodo a que satisface la Condición 1 en su camino. Es posible que la aridad de a fuera ya máxima y x fuera forzado a elegir un vecino de a. Consideraremos pronto las implicaciones de esto en el proceso de búsqueda.

[1] En ese caso se convertiría en *timestamp* puro.

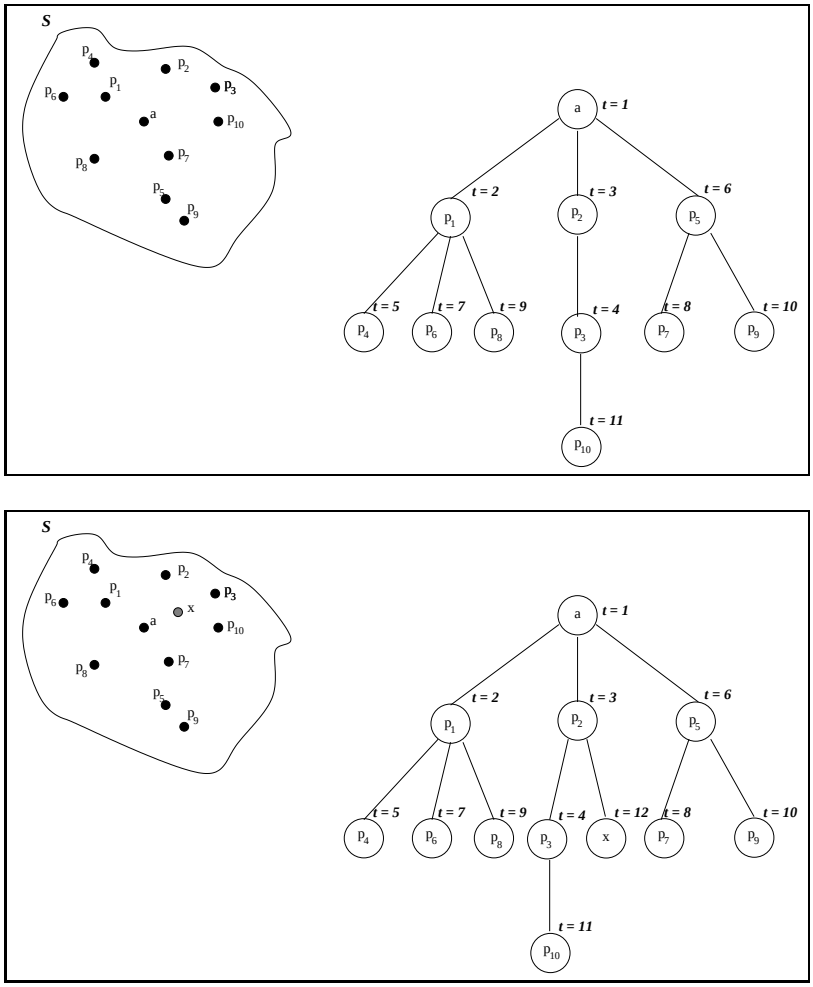


Figura 4.25: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la inserción del nuevo elemento x en un *SA-tree* dinámico, usando la técnica de *timestamp con aridad limitada*.

La Figura 4.26 ilustra el proceso de inserción. Seguimos sólo un camino desde la raíz del árbol al padre del elemento insertado. La función se invoca como `Insertar(`a`,`x`)`, donde a es la raíz del árbol y x es el elemento a insertar. El *SA-tree* se puede construir ahora comenzando con un único primer nodo a donde $N(a) = \emptyset$ y $R(a) = 0$, y luego realizando sucesivas inserciones.

```
Insertar (Nodo a, Elemento x)

1.  R(a) ← máx{R(a), d(a,x)}
2.  c ← argmin_{b∈N(a)} d(b,x)
3.  If d(a,x) < d(c,x) ∧ |N(a)| < MaxArity Then
4.      N(a) ← N(a) ∪ {x}
5.      N(x) ← ∅,  R(x) ← 0
6.      timestamp(x) ← CurrentTime
7.  Else Insertar (c,x)
```

Figura 4.26: Inserción de un nuevo elemento x en un *SA-tree* dinámico con raíz a. $MaxArity$ es la aridad máxima del árbol y $CurrentTime$ es el tiempo actual, el cual se incrementa luego de cada nueva inserción.

Para analizar el costo de una inserción supongamos que el árbol tiene siempre aridad A y que está perfectamente balanceado. En ese caso podemos deducir la altura máxima del árbol h, sabiendo que:

$$\begin{aligned}
\sum_{i=0}^{h} A^i &= n \\
\frac{A^{h+1}-1}{A-1} &= n \\
A^{h+1}-1 &= n(A-1) \\
A^{h+1} &= n(A-1)+1 \\
h+1 &= \log_A(n(A-1)+1) \\
h &= (\log_A(n(A-1)+1))-1 \\
h &= \log_A n + O(1/n)
\end{aligned}$$

Así, en el peor caso el elemento a insertar se debe comparar contra los A vecinos de cada nodo en el camino hasta una hoja. Por lo tanto, cada inserción en un árbol cuesta $A(\log_A n + O(1/n))$.

La Figura 4.27 compara el costo de la construcción incremental usando esta nueva técnica contra la construcción estática para conjuntos crecientes de la base de datos. Mostramos aridades 4, 8, 16 y 32. En todos los casos, el desempeño de la construcción mejora a medida que reducimos la aridad del árbol, siendo mucho mejor que la construcción estática (por ejemplo, 2 veces más rápida sobre las palabras y 4 veces más rápida sobre los vectores de dimensión 15). Mostramos también la comparación en un espacio de vectores en dimensión 5, para que se pueda observar que la construcción incremental logra mejores costos de construcción que el método estático también en espacios de baja dimensión. Notar que si permitimos una aridad suficientemente grande (v.g., 32 sobre palabras) la versión incremental es algo peor que la versión estática (cuya aridad es ilimitada). Esto muestra que la aridad reducida es un factor clave para bajar los costos de construcción. Si recordamos que el costo de inserción con aridad A es $A\,O(\log_A n)$ y sobre aridad ilimitada se mostró en [Nav02] que la aridad promedio es $A = O(\log n)$, entonces el costo de construcción por elemento es $O(\log^2 n/\log\log n)$ con aridad ilimitada.

La pregunta es cómo afecta una aridad reducida a los tiempos de búsqueda. Consideramos esto a continuación.

4.2.2. Búsquedas

Durante la búsqueda debemos considerar dos hechos importantes. El primero es que, cuando se insertó un elemento x, un nodo a en su camino puede no haberse elegido como su padre porque su aridad ya era máxima. Así en lugar de elegir el elemento más cercano a x entre $\{a\} \cup N(a)$, podemos haber tenido que elegir sólo entre $N(a)$. Esto significa que tenemos que excluir a $\{a\}$ de la minimización de la línea 3 en la Figura 3.5. El segundo hecho que tenemos que considerar es que, cuando se insertó x los elementos con timestamp más alto no estaban presentes en el árbol; así, x pudo elegir su vecino más cercano sólo entre los elementos más viejos que él.

Por lo tanto, consideramos los vecinos $\{v_1, \ldots, v_k\}$ de a de más viejo a más nuevo, sin tener en cuenta a a, y realizamos la minimización a medida que atravesamos la lista de vecinos. Esto significa que entramos

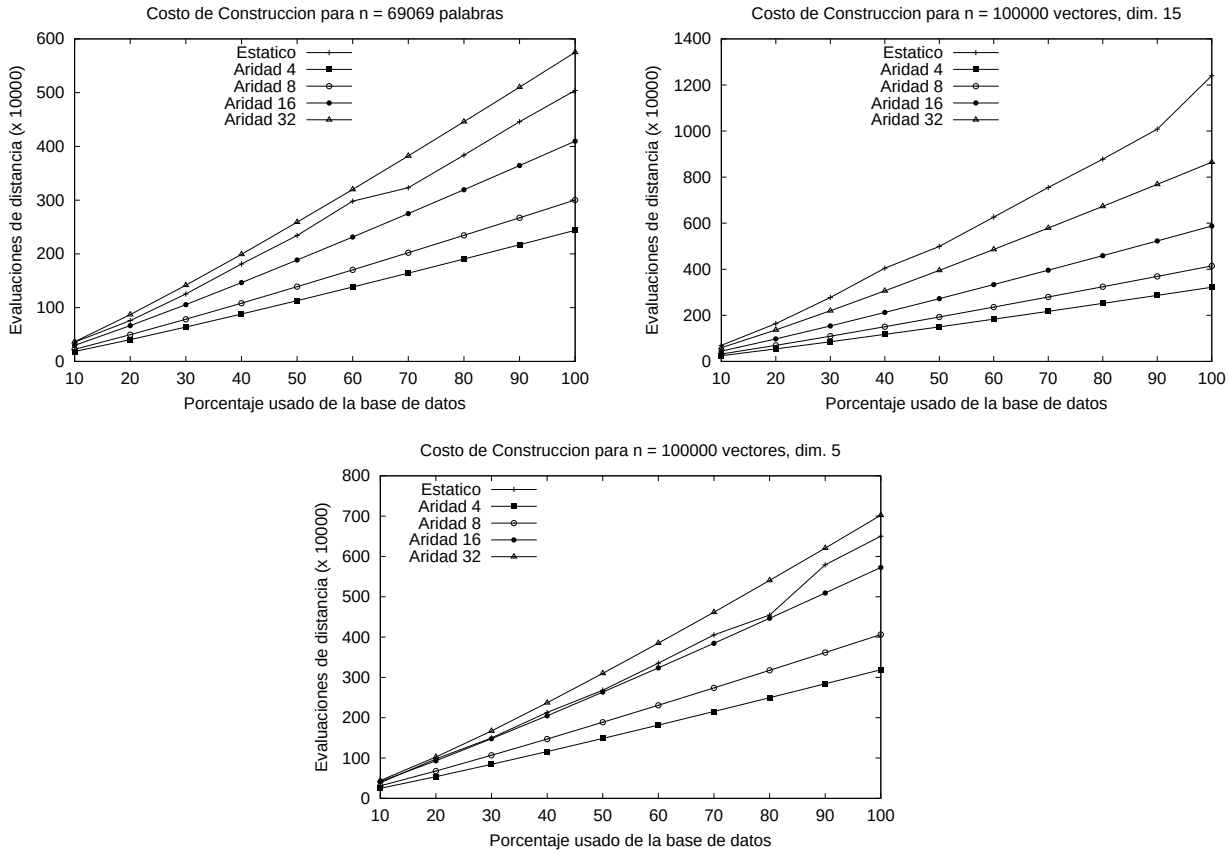


Figura 4.27: Costos de construcción estática versus dinámica.

en el subárbol de v_i si $d(q, v_i) \leq \text{mín}\{d(q, v_1), \ldots, d(q, v_{i-1})\} + 2r$. Es decir, siempre entramos en v_1; entramos en v_2 si $d(q, v_2) \leq d(q, v_1) + 2r$; y así siguiendo. Recordemos nuevamente la razón: entre la inserción de v_i y v_{i+j} pudieron aparecer nuevos elementos que eligieron a v_i sólo porque v_{i+j} no estaba aún presente, así podríamos perder un elemento si no entramos en v_i por la existencia de v_{i+j}. Debido a la aridad limitada y al igual que en la Sección 4.1.5, d_{min} no se hereda.

Hasta ahora no hemos necesitado realmente los timestamps exactos salvo sólo para mantener los vecinos ordenados por timestamp. Podemos usar mejor la información del timestamp para reducir el trabajo a realizar dentro de los vecinos más antiguos, tal como lo explicamos en la Sección 4.1.4.

La Figura 4.28 muestra el algoritmo de búsqueda, que se invoca inicialmente como `BúsquedaporRango(`a`,`q`,`r`,`$CurrentTime$`)`, donde a es la raíz del árbol. Notar que $d(a, q)$ siempre se conoce excepto en la primera invocación. A pesar de la naturaleza cuadrática de la iteración implícita en las líneas 4 y 6 la query, desde luego, se compara sólo una vez contra cada vecino.

La Figura 4.29 compara esta técnica contra la estática. En el caso de las palabras, el método estático da un tiempo de búsqueda levemente mejor que la técnica dinámica. En el espacio de vectores de dimensión 15, las aridades 16 y 32 mejoran (por un pequeño margen) el desempeño estático. También incluimos un ejemplo de vectores de dimensión 5, mostrando que en bajas dimensiones pequeñas aridades mejoran ampliamente el tiempo de búsqueda del método estático. La mejor aridad para la búsqueda depende del espacio métrico, pero la regla, a grosso modo, es que aridades bajas son buenas para dimensiones bajas o radios de búsqueda

```
BúsquedaporRango (Nodo a, Query q, Radio r, Timestamp t)

1. If timestamp(a) < t  ∧  d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Informar a
3.    d_min  ←  ∞
4.    For v_i ∈ N(a) en orden creciente de timestamp Do
5.       If d(v_i,q) ≤ d_min + 2r Then
6.          k ← mín{timestamp(v_j), j > i ∧ d(v_i,q) > d(v_j,q) + 2r} ∪ {CurrentTime}
7.          BúsquedaporRango (v_i, q, r, mín{t,k})
8.       d_min ← mín{d_min, d(v_i,q)}
```

Figura 4.28: Buscando q con radio r en un *SA-tree* dinámico.

pequeños.

Es importante notar que hemos logrado dinamismo y también mejorado el desempeño de la construcción. En algunos casos también hemos (ampliamente) mejorado el desempeño de la búsqueda, mientras que en otros casos pagamos un pequeño precio por tener dinamismo. En general, esta estructura se convierte en una elección muy conveniente.

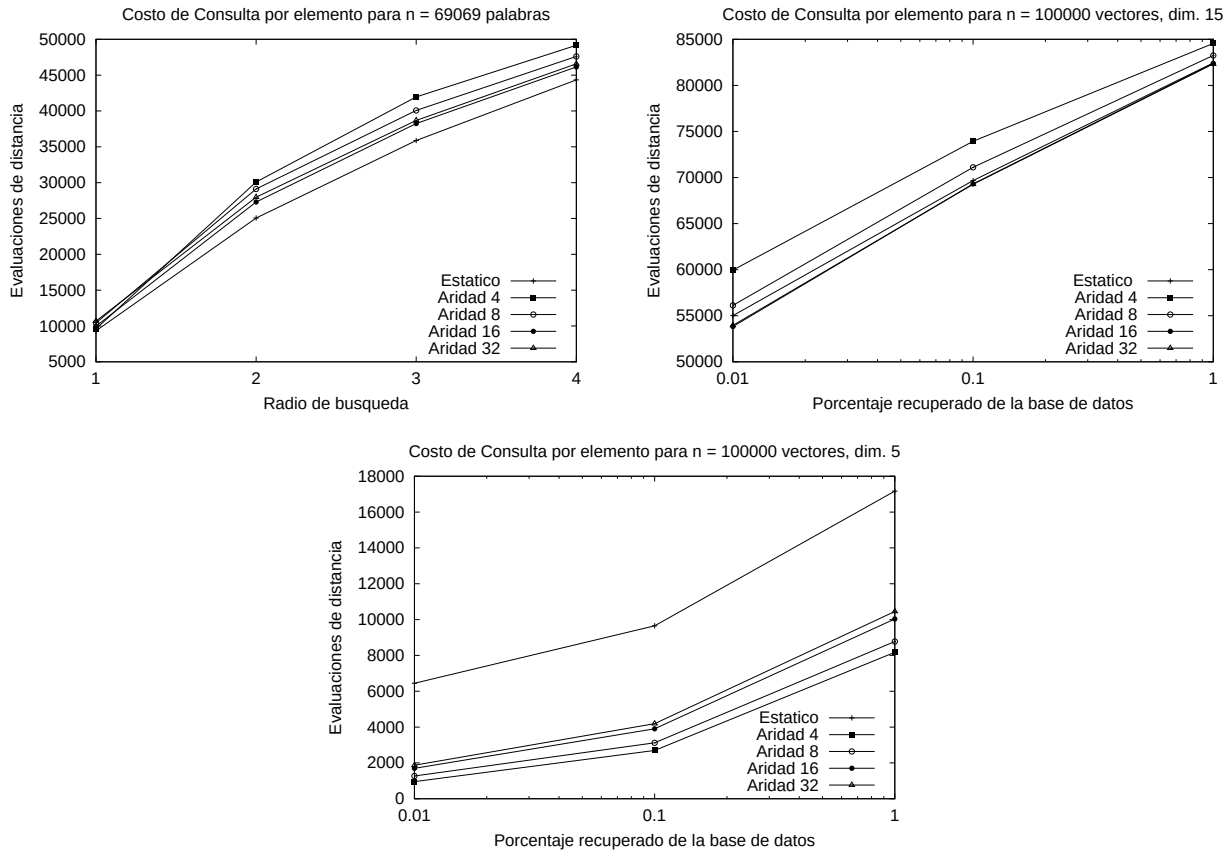


Figura 4.29: Costos de búsqueda estática versus dinámica.

Mostramos ahora una comparación de los costos de construcción y búsqueda entre los métodos in-

crementales previos y el nuevo método propuesto. La Figura 4.30 muestra los costos de construcción y búsqueda en el espacio de las palabras, del *SA-tree* estático, *timestamp*, *inserción cerca de las hojas* y nuestro nuevo *timestamp con aridad limitada*. Hemos utilizado los mejores parámetros que encontramos en los experimentos para todos los métodos. La Figura 4.31 muestra la misma comparación para el espacio de vectores de dimensión 15. Observar que eligiendo la aridad adecuada (29) alcanzamos el mismo desempeño que el método estático para la búsqueda.

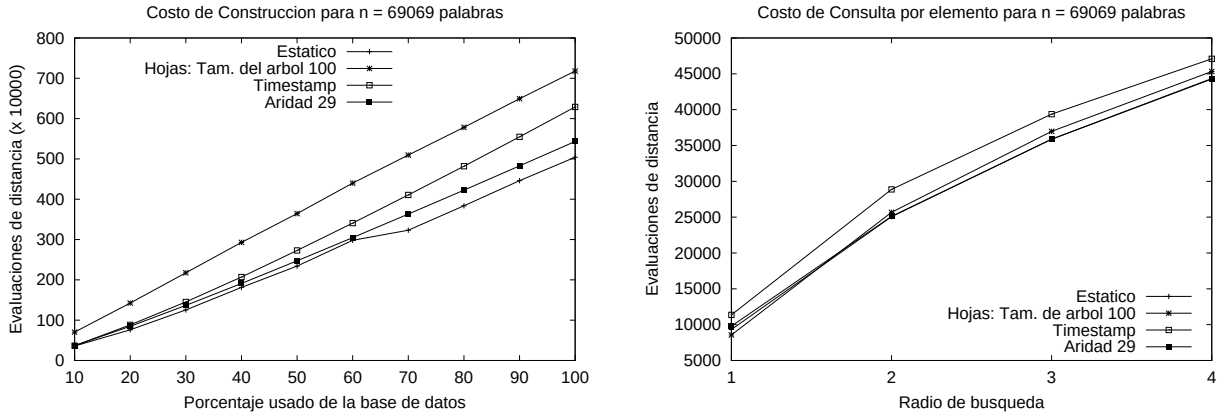


Figura 4.30: Costos de construcción y búsqueda para diferentes métodos en el espacio de palabras.

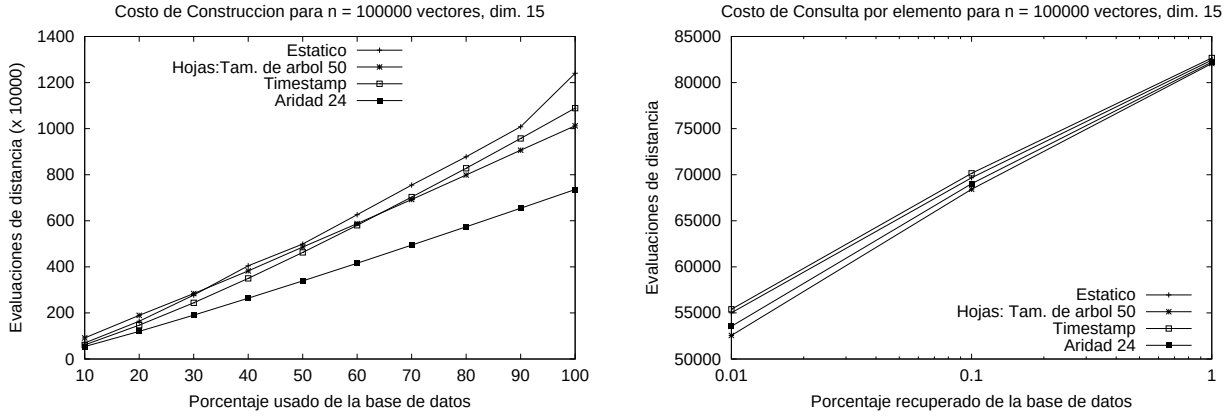


Figura 4.31: Costos de construcción y búsqueda para diferentes métodos en el espacio de vectores de dimensión 15.

Como se puede observar, nuestro nuevo método supera ampliamente a los métodos incrementales previos en tiempos de construcción, y al mismo tiempo es competitivo (cuando no mejor) en tiempo de búsqueda. En el caso de vectores de coordenadas supera por mucho a la versión estática, mientras que en el caso de las palabras es casi igual, con el valor agregado del dinamismo.

Hemos introducido la aridad del árbol como un nuevo parámetro a sintonizar. Falta analizar cuán dificultoso es encontrar la mejor aridad y cuán costoso es un pequeño error en esta sintonía.

Mostramos algunos resultados sobre las aridades óptimas para los espacios considerados. La Figura 4.32 muestra aridades alrededor del valor óptimo. Encontramos que la mejor aridad para el espacio de palabras es 29 y 24 para el espacio de vectores de dimensión 15. Es importante notar que, alrededor de estos óptimos,

los costos difieren en muy pocas evaluaciones de distancia. La Tabla 4.1 y la Tabla 4.2 dan los costos para los espacios de palabras y de vectores de dimensión 15, respectivamente.

Esto muestra que, aunque fallemos al elegir la mejor aridad por un margen razonable para construir el *SA-tree*, el precio que se pagará en tiempo de búsqueda no será muy significativo. Esto no ocurría con la técnica que utiliza *áreas de rebalse* (Sección 4.1.2), donde un pequeño error era fatal.

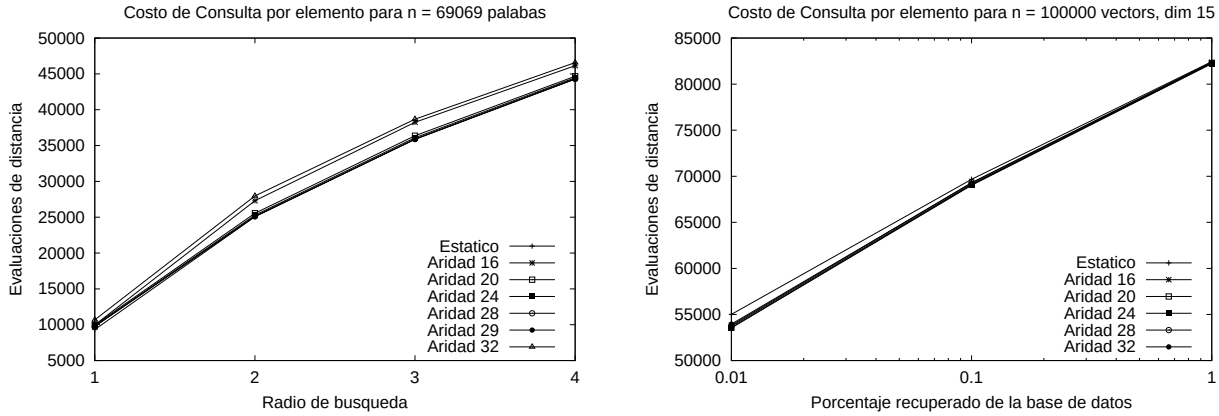


Figura 4.32: Costos de búsqueda estática vs. dinámica para diferentes aridades.

Método	Radio de Búsqueda			
	1	2	3	4
Estático	9324.89	25065.42	35883.64	44335.43
Aridad 16	9972.58	27298.47	38249.70	46146.64
Aridad 20	9958.61	25554.92	36367.81	44665.95
Aridad 24	9816.64	25286.59	36088.06	44471.21
Aridad 28	9762.68	25123.40	35898.67	44296.86
Aridad 29	9795.26	25110.16	35862.23	44268.24
Aridad 32	10679.11	27971.59	38685.50	46575.76

Tabla 4.1: Valores numéricos de la Figura 4.32 (izquierda).

Método	Porcentaje recuperado		
	0.01 %	0.1 %	1 %
Estático	55008.77	69673.37	82440.31
Aridad 16	53831.91	69308.48	82383.96
Aridad 20	53586.18	69076.20	82265.03
Aridad 24	53545.81	69021.79	82223.39
Aridad 28	53721.04	69153.21	82283.62
Aridad 32	53951.40	69264.22	82329.44

Tabla 4.2: Valores numéricos de la Figura 4.32 (derecha).

4.2.3. Búsqueda de Vecinos más Cercanos

También podemos realizar la búsqueda del vecino más cercano (1-NN) y de los k vecinos más cercanos, como se describió en la Sección 3.2.3 para la versión original del *SA-tree*, simulando una búsqueda por rango donde se reduzca el radio de búsqueda a medida que obtenemos más y más información.

En una búsqueda por rango fijo r, el orden en el cual se hace el backtracking en el árbol carece de importancia. En este caso no sucede lo mismo, porque si queremos encontrar rápidamente los elementos cercanos a q lo más pronto posible tendremos que reducir r. Nuevamente aquí usamos la idea general, propuesta en [Uhl91a] porque se puede adaptar también a nuestra nueva estructura de datos. Mantenemos una cola de prioridad de subárboles, donde el más prometedor está primero. Inicialmente, insertamos la raíz del *SA-tree* en la estructura de datos. Iterativamente, extraemos la raíz del subárbol más prometedor, lo procesamos e insertamos todas las raíces de sus subárboles en la cola. Esto se repite hasta que la cola se vacíe o se pueda descartar su subárbol más prometedor (es decir, su "valor de promesa" es suficientemente malo) o se pueda podar dicho subárbol por información adicional proveniente de los timestamps.

Durante la búsqueda debemos recordar dos hechos importantes, que tuvimos en cuenta para realizar la búsqueda por rango. El primero es que, cuando se insertó un elemento x, un nodo a en su camino puede no haberse elegido como su padre porque su aridad ya era máxima. Así en lugar de elegir el elemento más cercano a x entre $\{a\} \cup N(a)$, podemos haber tenido que elegir sólo entre $N(a)$. Esto significa que tenemos que excluir $\{a\}$ de la minimización que se realiza entre las líneas 15 y 19 en la Figura 4.33 y por consiguiente también excluimos a $N(A(a))$. El segundo hecho que tenemos que considerar es que, cuando se insertó x, los elementos con timestamp más alto no estaban presentes en el árbol, así x pudo elegir su vecino más cercano sólo entre los elementos más viejos que él. Esto significa que la minimización se debe realizar a medida que atraviesa los vecinos de $N(a)$.

Ahora, de manera similar a lo que ocurría en la versión original del *SA-tree*, la medida más elegante de cuán prometedor es un subárbol es un límite inferior de la distancia entre q y cualquier elemento en el subárbol. Cuando este límite inferior excede r podemos detener el proceso completo. Tenemos en realidad tres posibles límites inferiores:

1. Como encontramos el vecino más cercano c considerado hasta el momento y entonces entramos en cualquier otro vecino b tal que $d(q,b) - d(q,c) \leq 2r$, no tendríamos que entrar en el subárbol con raíz b si no se cumple que $(d(q,b) - d(q,c))/2 \leq r$. Pero, de hecho, este c se toma ahora sólo sobre los vecinos atravesados hasta el momento y no considera los ancestros.

2. Por el límite inferior de la distancia entre q y un elemento en el subárbol tenemos que $d(q,b) - R(b) \leq r$.

3. El timestamp también da otro criterio de corte. Una visión equivalente a la usada para este criterio es la siguiente. Cada vez que entramos a un hijo y de v_i, buscamos los hermanos v_{i+j} que sean más viejos que y. Sobre este conjunto calculamos el radio máximo que evitaría procesar a y, $r_y = \max\{d(q,v_i) - d(q,v_{i+j}), j > 0\}/2$. Si resulta que $r > r_y$, entonces no necesitamos procesar a y. De modo que r_y es una tercera cota para r.

Como r se reduce a lo largo de la búsqueda, un nodo b puede parecer útil cuando se lo inserta en la cola de prioridad y ser inútil después, cuando se lo extrae de la cola para procesarlo. Así almacenamos junto con b el máximo de los límites inferiores, t, y usamos este máximo para ordenar los subárboles en la cola de prioridad, con el más pequeño primero. Como antes, este límite inferior t se hereda de padres a hijos. A

medida que extraemos subárboles de la cola, verificamos si su valor t excede a r, en cuyo caso detenemos el proceso completo y ya se sabe que todos los subárboles restantes contienen elementos irrelevantes.

La Figura 4.33 presenta el algoritmo.

```
BúsquedaNN (Árbol a, Query q, Vecinos requeridos k)

1.   Q ← {(a, máx(0, d(q,a) − R(a)))}  /* subárboles prometedores */
2.   A ← ∅  /* mejor respuesta hasta ahora */
3.   r ← ∞
4.   While Q no sea vacía
5.      (b,t) ← elemento en Q con el t más pequeño
6.      Q ← Q − {(b,t)}
7.      If t > r Break  /* criterio global de parada */
8.      A ← A ∪ {(b, d(q,b))}
9.      If |A| = k + 1
10.        (c, maxd) ← elemento en A con maxd más grande
11.        A ← A − {(c, maxd)}
12.     If |A| = k
13.       (c, maxd) ← elemento en A con maxd más grande
14.       r ← maxd
15.     d_min ← ∞
16.     For v_i ∈ N(b)  /* en orden creciente de timestamp *
17.       maxr ← máx {(d(q,v_i) − d(q,v_j))/2, j > i}
18.       Q ← Q ∪ {(v_i, máx{maxr, (d(q,v_i) − d_min)/2, d(q,v_i) − R(v_i)})}
19.       d_min ← mín {d_min, d(q,v_i)}
20.  Informar la respuesta A
```

Figura 4.33: Algoritmo para buscar los k vecinos más cercanos a q en un *SA-tree* dinámico. A es una cola de prioridad de pares *(nodo,distancia)* ordenados por *distancia* creciente. Q es una cola de prioridad de pares *(nodo,lbound)* ordenados por *lbound* creciente.

Capítulo 5

Eliminación

Ya hemos propuesto un buen método para manejar inserciones en el *SA-tree*. En este capítulo proponemos y analizamos experimentalmente distintos métodos para realizar eliminaciones. Mostramos que es posible eliminar elementos en el *SA-tree*, pagando un bajo costo por permitir total dinamismo y manteniendo aún una buena eficiencia de búsqueda.

El dinamismo completo no es común en estructuras de datos métricas [CNBYM01] (ver Sección 2.5.1). Aunque es bastante usual permitir inserciones eficientes, no ocurre lo mismo con las eliminaciones. En varios índices uno puede eliminar algunos elementos, pero existen determinados elementos que nunca pueden ser eliminados. Esto es particularmente problemático en el escenario de los espacios métricos, donde los objetos podrían ser muy grandes (v.g. imágenes) y sería indispensable eliminarlos físicamente. Nuestros algoritmos permiten eliminar cualquier elemento de un *SA-tree*, lo cual es destacable para una estructura de datos cuya concepción original fue marcadamente estática [Nav02].

5.1. Acerca de Encontrar el Elemento a Eliminar

Para eliminar un elemento x, el primer paso es localizarlo en el árbol. A diferencia de la mayoría de las estructuras de datos clásicas, en nuestro *SA-tree* hacer esto no es equivalente a simular la inserción de x y ver a dónde nos guía en el árbol. La razón es que el árbol era diferente cuando se insertó x. Si x se insertara de nuevo, podría elegir ir por un camino diferente en el árbol, el cual no existía al momento de la primera inserción.

Una solución elegante a este problema es realizar una búsqueda por rango con radio cero, es decir, una consulta de la forma $(x, 0)$. Esto es razonablemente barato y nos llevará por todos los lugares en el árbol donde se podría haber insertado a x.

Por otro lado, es dependiente de la aplicación si esta búsqueda es o no necesaria. La aplicación podría retornar información cuando se inserta un objeto en la base de datos. Esta información podría contener un puntero al correspondiente nodo del árbol, y agregando en el árbol punteros al padre permitiríamos ubicar el camino sin costo adicional (en términos de evaluaciones de distancia). Otra opción posible es que la aplicación conozca el timestamp de inserción del elemento a eliminar, porque en ese caso se lo puede buscar como si se lo fuera a insertar (obviando los elementos más nuevos cuando se busca el camino por donde meterse).Así, en lo que sigue, no consideramos la localización del objeto como parte del problema de eliminación, aunque hemos mostrado cómo hacerlo, de varias maneras, si es necesario.

5.2. Primeras Opciones Analizadas

Hemos analizado varias alternativas para eliminar elementos de un *SA-tree* dinámico. Desde el comienzo hemos descartado la opción trivial de marcar al elemento como eliminado sin eliminarlo realmente. Como se explicó, esta solución probablemente sea inaceptable en la mayoría de las aplicaciones. Entendemos que el elemento tiene que ser eliminado físicamente. Podemos, si lo deseamos, mantener su nodo en el árbol, pero no el objeto en sí mismo.

Claramente una hoja del árbol siempre se puede remover sin ninguna complicación, así que nos ocuparemos de cómo remover nodos internos del árbol.

5.2.1. Nodos Ficticios

Nuestra primera alternativa para eliminar un elemento x es dejar su nodo en el árbol (sin contenido) y marcarlo como eliminado. Llamamos a estos nodos *ficticios*. Aunque es poco costoso y simple al momento de la eliminación, debemos ver ahora cómo llevar a cabo una búsqueda consistente cuando algunos nodos no contienen objetos.

La Figura 5.1 muestra la situación antes (arriba) y después (abajo) de eliminar el elemento p_2 del árbol, usando la técnica de marcarlo como *nodo ficticio*.

Básicamente, si el nodo $b \in N(a)$ es ficticio, no tenemos suficiente información para poder evitar entrar en el subárbol de b cuando hemos alcanzado a a. Así no podemos incluir a b en la minimización y debemos entrar siempre a su subárbol (excepto si podemos usar el timestamp de b para podar la búsqueda).

Si analizamos la misma situación sobre la alternativa de *timestamp puro* (ver Sección 4.1.4), podríamos considerar la mínima distancia desde la query q a los elementos en $N(A(a))$ que son más viejos que b (es decir, los elementos contra los que se comparó b cuando se insertó en el árbol) como una cota estimada de la distancia entre la query q y b. O sea, $d_{est} = \min\{d(q,c), c \in N(A(a)) \wedge timestamp(c) < timestamp(b)\}$. Así entraríamos al subárbol de b sólo si se cumple que $d_{est} \leq d_{min} + 2r$. En nuestra versión de *timestamp con aridad limitada* no podemos incluir a a en la minimización y por consiguiente tampoco a $N(A(a))$.

La búsqueda realizada al momento de la inserción, por otro lado, tiene que seguir sólo un camino en el árbol. En este caso, se es libre de elegir insertar el nuevo elemento en cualquier vecino ficticio del nodo actual, o en el vecino más cercano no ficticio. Sin embargo, una buena política es tratar de no incrementar el tamaño de los subárboles cuya raíz es un nodo ficticio, porque eventualmente estos subárboles se deberán reconstruir (ver Sección 5.4).

Así, aunque la eliminación es simple, se degrada el proceso de búsqueda.

5.2.2. Reinsertando Subárboles

Una idea muy difundida en el área de búsqueda en espacios Euclídeos es que eliminar y reinsertar los elementos de una página de disco es beneficioso porque, con más elementos en el árbol, el espacio puede agruparse mejor. Seguimos este principio ahora para obtener un método con eliminaciones más costosas, pero con buen desempeño de búsqueda.

Cuando se elimina un nodo x, desconectamos el subárbol con raíz x del árbol principal. Esta operación no afecta la correctitud del resto del árbol, pero ahora tenemos que reinsertar los subárboles cuyas raíces son

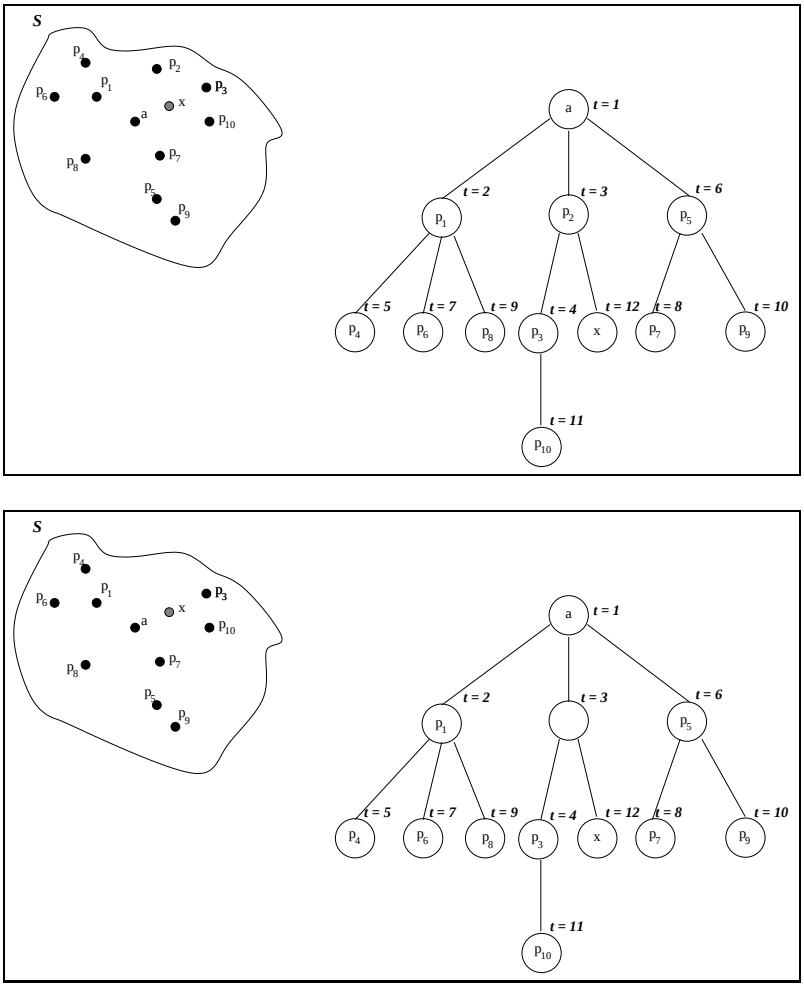


Figura 5.1: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la eliminación del elemento p_2 en un *SA-tree* dinámico, usando la técnica de *nodos ficticios*.

los nodos de $N(x)$. Para hacerlo eficientemente tratamos de reinsertar subárboles completos, siempre que sea posible.

Para reinsertar un subárbol con raíz y, seguimos los mismos pasos que si insertáramos un nuevo objeto y y encontramos el punto de inserción a. La diferencia es que suponemos que y es un objeto "gordo" con radio $R(y)$. Podemos elegir poner el subárbol completo con raíz y como un nuevo vecino de a sólo si $d(y,a) + R(y)$ es menor que $d(y,b)$ para cualquier $b \in N(a)$. Similarmente, podemos elegir bajar por el vecino $c \in N(a)$ sólo si $d(y,c) + R(y)$ es menor que $d(y,b)$ para cualquier $b \in N(a)$. Cuando ninguna de estas condiciones se cumple, estamos forzados a dividir el subárbol con raíz y en sus elementos: uno es el elemento y y los otros son los subárboles con raíces en $N(y)$. Una vez dividido el subárbol, continuamos el proceso de inserción con cada uno de los componentes por separado.

Cada vez que insertamos un nodo o un subárbol, obtenemos un timestamp nuevo para el nodo o la raíz del subárbol. Los elementos dentro del subárbol obtendrán sus nuevos timestamps manteniendo el ordenamiento relativo dentro de los elementos del subárbol. La manera más sencilla de hacerlo es suponer que los timestamps se almacenan relativos a los de su padre. De este modo, no se tiene que hacer nada. Necesitamos,

sin embargo, almacenar en cada nodo el diferencial máximo de tiempo almacenado en el subárbol, para actualizar adecuadamente a $CurrentTime$ cuando se reinserta un subárbol completo. Durante el proceso de inserción esto se hace fácilmente y por simplicidad se omite en el pseudocódigo.

Durante la reinserción, también modificamos los radios de cobertura de los nodos a del árbol atravesados. Cuando insertamos un subárbol completo tenemos que sumar $d(y, a) + R(y)$, lo que es mayor que lo necesario. Más aún, esto ocurre aunque finalmente debamos descomponer el subárbol y terminar insertándolo elemento por elemento. Esto involucra un costo, en tiempo de búsqueda, por haber reinsertado un subárbol completo de una sola vez.

La Figura 5.2 muestra la situación antes (arriba) y después (abajo) de eliminar el elemento p_2 del árbol, usando la técnica de *reinserción de subárboles*. Notar que a los elementos p_3, p_{10} y x se les asigna un nuevo valor de timestamp. En este ejemplo se desconecta el subárbol con raíz p_2 y se reinsertan completos los subárboles con raíces p_3 y x. Cabe aclarar que antes de eliminar a p_2 el radio de cobertura de a era $R(a) = d(a, p_9)$ (p_9 es el elemento más distante desde a) y luego de la reinserción de los subárboles $R(a) = d(a, p_3) + d(p_3, p_{10})$ que es mayor que lo necesario. Mostramos en la Figura 5.3 la situación en el espacio, con el radio de cobertura de a antes (izquierda) y después (derecha) de la reinserción del subárbol de p_3.

Notar que puede parecer que, cuando buscamos el lugar para reinsertar los subárboles de un nodo x eliminado, uno podría ahorrar tiempo comenzando la búsqueda en el padre de x; sin embargo, el árbol ha cambiado desde el momento en que se creó el subárbol de x, y ahora pueden existir nuevas opciones.

La Figura 5.4 muestra el algoritmo para reinsertar un árbol con raíz y en un *SA-tree* dinámico con raíz a. La eliminación de un nodo x se hace primero localizándolo en el árbol (digamos, $x \in N(b)$), luego removiéndolo de $N(b)$, y finalmente reinsertando cada subárbol $y \in N(x)$ usando `ReinsertarT(a,y)`.

5.2.3. Reinsertando Subárboles Elemento por Elemento

La reinserción de subárboles completos acarrea, en algunos espacios, un inconveniente adicional que puede degradar el desempeño de la búsqueda. Si se debe reinsertar un subárbol con raíz y, el proceso atraviesa un camino desde la raíz del árbol hasta llegar a un punto en el que o reinserta el subárbol completo o debe dividirlo, y reinsertar y y cada uno de los subárboles con raíces en $N(y)$. En cada uno de los nodos a que atraviesa en ese camino actualiza, inevitablemente, $R(a)$ a un valor posiblemente mayor que el necesario ($d(a, y) + R(y)$). Esto provoca que dichos radios de cobertura puedan quedar sobredimensionados (aún si no se logra reinsertar completo al subárbol).

Así una alternativa, similar a la anterior, que no provoca sobredimensión en los radios de cobertura es reinsertar de a uno los elementos del subárbol con raíz y. En este caso la búsqueda tiene un mejor desempeño, aunque no así la eliminación.

La situación antes y después de eliminar al elemento p_2 en el árbol que se mostró en Figura 4.25, usando esta técnica, produce exactamente el mismo árbol que se muestra en la Figura 5.2 (arriba). En este caso se destaca que $R(a)$ no varía antes y después de la eliminación.

El algoritmo para realizar la reinserción elemento por elemento se consigue variando el algoritmo de Figura 5.4. El nuevo algoritmo se muestra en Figura 5.5. El proceso de eliminación en este caso es el mismo que ya se describió anteriormente.

Para considerar cómo afectan las eliminaciones al costo de búsqueda, buscamos sobre un índice que

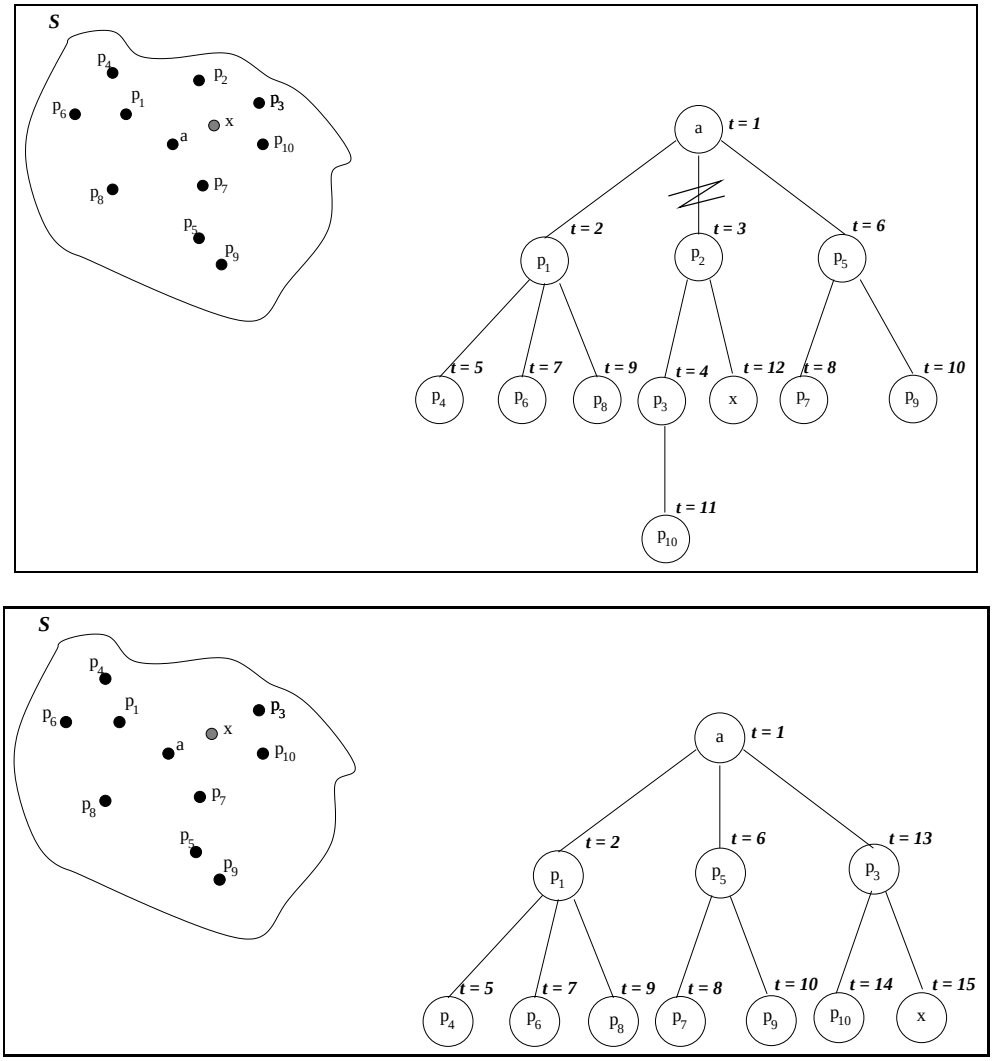


Figura 5.2: Ejemplo de una situación posible previa (arriba) y posterior (abajo) a la eliminación del elemento p_2 en un *SA-tree* dinámico, usando *reinserción de subárboles*.

contiene la mitad de los elementos de la base de datos. Esta mitad se construye insertando más elementos y luego eliminando suficientes elementos para dejar el 50 % del conjunto en el índice. Así, comparamos la búsqueda sobre conjuntos del mismo tamaño donde un porcentaje de los elementos se ha eliminado para dejar el conjunto de ese tamaño. Por ejemplo, 30 % de eliminaciones significa que se insertaron 80000 elementos y luego se eliminaron 30000, de manera tal de dejar 50000 elementos (la mitad del conjunto).

La Figura 5.6 muestra los costos de las búsquedas sobre el espacio de vectores de dimensión 5 con distribución uniforme descripto en Sección 2.4.1, cuando se eliminan los porcentajes de la base de datos que se indican (izquierda). Claramente las búsquedas se degradan a medida que aumenta el número de elementos eliminados. Mostramos también el resultado de un experimento en el que corregimos, luego de las eliminaciones, los radios de cobertura de todos los elementos del árbol (derecha). Como se puede observar el comportamiento de las búsquedas mejora un poco, pero no del todo, por las razones que consideraremos en la Sección 5.3.

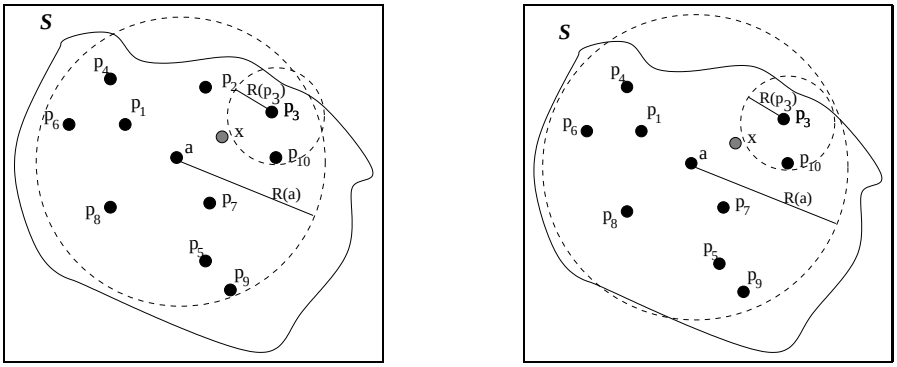


Figura 5.3: Radio de cobertura $R(a)$ antes (izquierda) y después (derecha) de la reinserción del subárbol con raíz p_3 el *SA-tree* dinámico de Figura 5.2, usando *reinserción de subárboles*.

```
ReinsertarT (Nodo a, Nodo y)
1.  If |N(a)| < MaxArity Then M ← {a} ∪ N(a)  Else M ← N(a)
2.  c1 ← argmin_{b∈M} d(b, y)
3.  c2 ← argmin_{b∈M−{c1}} d(b, y)
4.  If d(c1, y) + R(y) ≤ d(c2, y) Then /* mantener completo el subárbol */
5.     R(a) ← máx{R(a), d(a, y) + R(y)}
6.     If c1 = a Then /* reinsertarlo aquí */
7.        N(a) ← N(a) ∪ {y}
8.        time(y) ← CurrentTime /* el subárbol se corre automáticamente */
9.     Else ReinsertarT (c1, y) /* bajar */
10. Else /* dividir el subárbol */
11.    For z ∈ N(y) Do ReinsertarT (a, z)
12.    N(y) ← ∅, R(y) ← 0
13.    ReinsertarT (a, y)
```

Figura 5.4: Algoritmo para reinsertar un subárbol con raíz y en un *SA-tree* dinámico con raíz a.

Optimización

Una optimización al proceso de reinserción de subárboles utiliza más inteligentemente los timestamps. Supongamos que x será eliminado, y sea $A(x)$ el conjunto de los ancestros de x, es decir, todos los nodos en el camino desde la raíz a x. Para cada nodo c que pertenece al subárbol con raíz x tenemos que $A(x) \subset A(c)$. Así, cuando se insertó el nodo c, se comparó contra todos los vecinos de cada nodo en $A(x)$ cuyo timestamp fuera menor que el de c. Usando esta información podemos evitar evaluar las distancias a esos nodos cuando los revisitamos al momento de reinsertar c. Es decir, cuando buscamos el vecino más cercano a c, sabemos que el que está en $A(x)$ está más cerca de c que cualquier otro vecino más viejo, así tenemos que considerar sólo los más nuevos. Notar que esto es válido mientras que reentremos por el mismo camino donde x fue insertado previamente. Esta optimización también es aplicable al método de reinserción de a elementos.

Otra optimización posible

El problema que persiste en la eliminación de elementos, aún en la opción que reinserta elemento por elemento, es que los radios de cobertura pueden quedar sobredimensionados, pues nunca se reduce para

```
ReinsertarE (Nodo a, Nodo y)
 1.  If |N(a)| < MaxArity Then M ← {a} ∪ N(a)  Else M ← N(a)
 2.  N ← N(y) /* mantener la vecindad de y */
 3.  N(y) ← ∅,  R(y) ← 0
 4.  c1 ← argmin_{b∈M} d(b, y)
 5.  R(a) ← máx{R(a), d(a, y)}
 6.  If c1 = a Then /* reinsertarlo aquí */
 7.       N(a) ← N(a) ∪ {y}
 8.       time(y) ← CurrentTime
 9.  Else ReinsertarE (c1, y) /* bajar */
10.  for z ∈ N do ReinsertarE (a, z)
```

Figura 5.5: Algoritmo para reinsertar elemento por elemento un subárbol con raíz y en un *SA-tree* dinámico con raíz a.

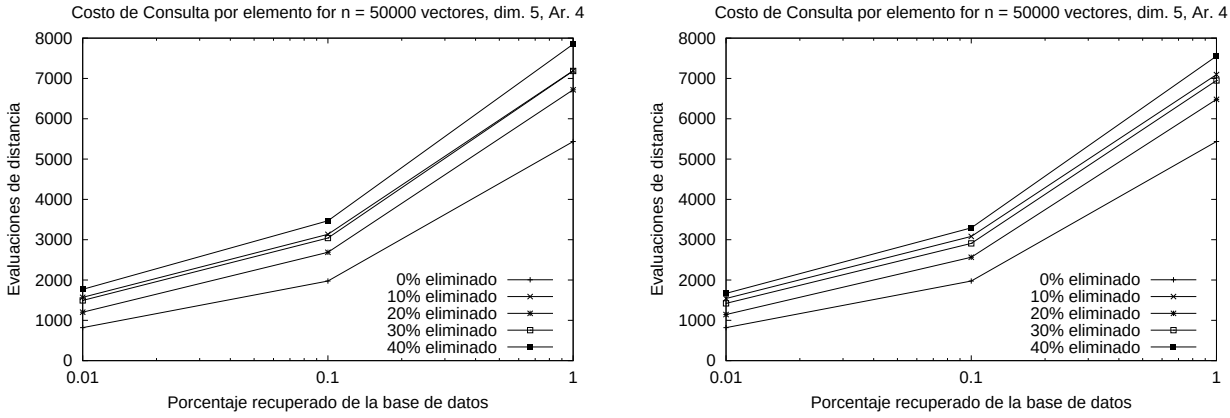


Figura 5.6: Costos de búsqueda para distintos porcentajes eliminados de la base de datos, en el espacio vectorial de dimensión 5, sin corregir los radios de cobertura (izquierda) y corrigiéndolos (derecha).

reajustarse por el subárbol eliminado.

Si eliminamos un elemento x, cada ancestro $a \in A(x)$ para el cual x era el elemento más distante en su subárbol tendrá posiblemente su $R(a)$ sobredimensionado.

Durante la eliminación de x podemos llevar a cabo el siguiente proceso. Sea b el padre de x, recalculamos $R(b)$ como $R(b) = \text{mín}\{d(b,v)+R(v), v \in N(b)-\{x\}\}$ y para cada ancestro $a \in A(x)-\{b\}$ recalculamos $R(a) = \text{mín}\{d(a,v) + R(v), v \in N(a)\}$. Esto no recupera los radios óptimos, pero los puede mejorar.

Cabe aclarar que para llevar a cabo esta optimización no son necesarios nuevos cálculos de distancia, porque todas las distancias involucradas se han debido calcular durante la búsqueda por rango $(x, 0)$ realizada para localizar a x en el árbol.

Esta optimización no se ha tenido en cuenta en los experimentos que se muestran en el capítulo, porque no esperamos que mejoren significativamente los tiempos de búsqueda (ver por ejemplo los resultados que muestra la Figura 5.6).

5.2.4. Análisis

El costo promedio de la reinserción de un subárbol es como sigue. Suponemos por simplicidad que reinsertamos los elementos de a uno, que el árbol tiene siempre aridad A y que está perfectamente balanceado. En ese caso en el nivel 0 hay un subárbol con los n elementos, en el nivel 1 hay A subárboles con $(n-1)/A$ elementos cada uno, y en general en el nivel i del árbol hay A^i subárboles con $(n-((A^i-1)/(A-1)))/A^i$ elementos cada uno. Como hay n subárboles posibles y el máximo nivel que se alcanza es $h = \log_A(n(A-1)+1)-1$, se puede deducir que el tamaño promedio de un árbol aleatoriamente elegido $S(n)$ es:

$$
\begin{aligned}
S(n) &= \sum_{i=0}^{h} \frac{A^i \cdot (n-((A^i-1)/(A-1)))/A^i}{n} \\
&= \frac{1}{n} \cdot \sum_{i=0}^{h} (n-((A^i-1)/(A-1))) \\
&= \frac{(h+1)n}{n} - \frac{1}{n} \cdot \sum_{i=0}^{h} \frac{(A^i-1)}{(A-1)} \\
&= (h+1) - \frac{1}{n(A-1)} \cdot \sum_{i=0}^{h} (A^i-1) \\
&= (h+1) - \frac{(n-(h+1))}{n(A-1)} \\
&= (h+1) - \frac{1}{(A-1)} + \frac{(h+1)}{n(A-1)} \\
&= \log_A(n(A-1)+1) - 1 + 1 + \frac{\log_A(n(A-1)+1)-1+1}{n(A-1)} - \frac{1}{(A-1)} \\
&= \log_A(n(A-1)+1) + \frac{\log_A(n(A-1)+1)}{n(A-1)} - \frac{1}{(A-1)} \\
&= \log_A n + O(1)
\end{aligned}
$$

Entonces, como el tamaño promedio de un subárbol aleatoriamente elegido es $\log_A n\ (1+o(1))$ y cada una de las (re)inserciones cuesta $A\ \log_A n\ (1+o(1))$ (ver Sección 4.2.1), el costo promedio de eliminación es $(A\ \log_A^2 n)\ (1+o(1))$. Esto es mucho más costoso que una inserción. En el análisis no hemos considerado las posibles optimizaciones, que nos permiten ahorrar algunas evaluaciones de distancia, porque el costo cambiaría sólo en un factor constante. La mejora podría llegar en promedio a $(A/2)\log_A^2 n\ (1+o(1))$.

5.3. Reconstrucción de Subárboles

El estudio de las opciones de reinserción muestran que a medida que crece el número de eliminaciones la búsqueda se degrada, lo cual se supone que se mantendrá con el tiempo. En parte se debe a la inevitable sobredimensión de los radios, pero no es la única razón tal como lo muestra la Figura 5.6. Es bastante grave que las búsquedas se degraden a medida que se insertan y borran elementos, porque no es sólo un problema de cantidad de elementos eliminados sino también de estabilidad en el tiempo.

Otra opción analizada, que persigue no degradar las búsquedas, es la siguiente. Supongamos que debemos borrar un elemento x que es hijo de a, al desaparecer x de $N(a)$ sabemos que los elementos más jóvenes que x, en el subárbol de a, se compararon contra x para decidir su punto de inserción. Por lo tanto, estos elementos ante la falta de x podrían elegir otro camino si se reinsertaran nuevamente en el árbol. Entonces, podemos recuperar del subárbol de a todos los elementos más jóvenes que x (es decir, aquellos con timestamp mayor) y reinsertarlos en el árbol, dejando el árbol tal como hubiese quedado si x no hubiera existido nunca.

Si los elementos más jóvenes que x se reinsertan como elementos totalmente nuevos, o sea si se les actualiza su timestamp, no quedaría más remedio que buscarles el punto apropiado de reinserción comenzando desde la raíz del árbol (este elemento se consideraría más nuevo que todos los elementos presentes en el árbol). Sin embargo, si les mantenemos su timestamp el punto apropiado de reinserción se puede buscar a partir de a y así nos ahorraríamos todas las comparaciones que se deben realizar para llegar hasta a. Pero, para que las reinserciones se realicen correctamente, debemos reinsertar los elementos en el mismo orden en que lo hicieron originalmente; es decir, que la secuencia de reinserciones debe ser por orden creciente de timestamp.

La Figura 5.7 muestra la situación de los elementos del subárbol de a respecto del timestamp de x

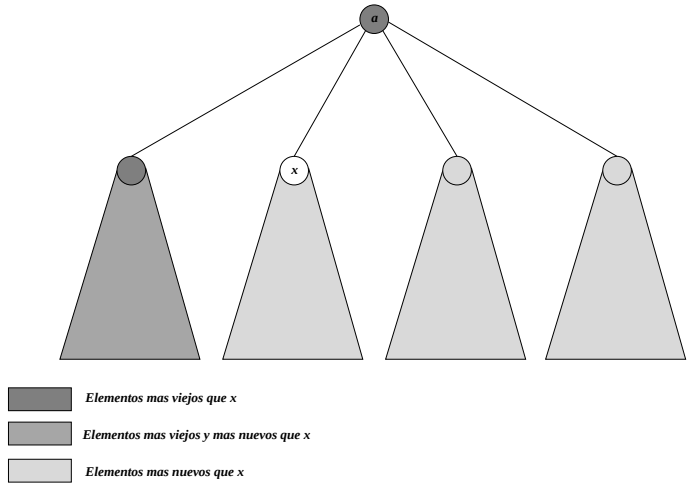


Figura 5.7: Ejemplo de dónde se encuentran los elementos del subárbol con raíz a, de acuerdo a su timestamp, respecto del timestamp de x

En esta opción entonces debemos recuperar todos los elementos más jóvenes que x desde el subárbol de a, desconectarlos de donde están ubicados, ordenarlos en orden creciente de timestamp y reinsertarlos de a uno, buscando su punto de reinserción a partir de a.

La Figura 5.8 muestra los costos de eliminación (izquierda) y de búsqueda (derecha) al eliminar la cantidad de elementos que se indica, sobre el espacio de vectores de dimensión 5. Se puede observar que las búsquedas no se degradan a medida que aumenta el número de elementos eliminados, a diferencia de lo que ocurría, para este espacio, al usar el método de reinserción de a elementos (ver Figura 5.6). Por otro lado, las eliminaciones son mucho más costosas.

La Figura 5.9 muestra el algoritmo que permite recuperar del subárbol de a todos los elementos más jóvenes que x. Por simplicidad en el código denotamos con $T(b)$ al conjunto de elementos en el subárbol con

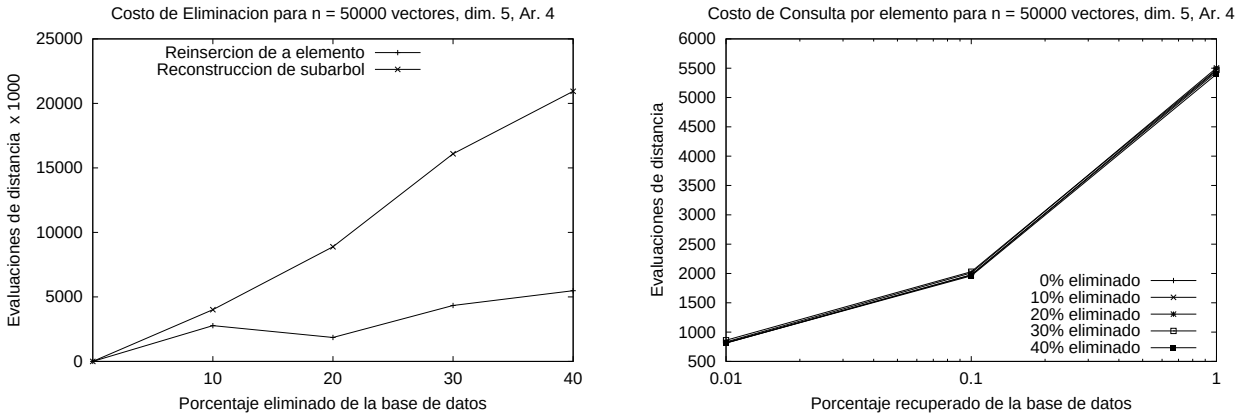


Figura 5.8: Costos de eliminación (derecha) y búsqueda (izquierda) para el método de reconstrucción de subárboles para distintos porcentajes de elementos eliminados.

raíz b. La Figura 5.10 ilustra el proceso de reconstrucción del subárbol, el cual invoca a `RecuperarTS(`a`,`x`)`, de acuerdo a esta técnica.

```
RecuperarTS (Nodo a, Nodo x)
 1.  Q ← {a} , T ← ∅
 2.  While Q no sea vacía
 3.     b ← primer elemento de Q
 4.     Q ← Q − {b}
 5.     For v ∈ N(b)
 6.        If timestamp(v) > timestamp(x) Then
 7.           N(b) ← N(b) − {v}
 8.           T ← T ∪ T(b)
 9.        Else
10.           Q ← Q ∪ {v}
11.  Return T
```

Figura 5.9: Algoritmo para recuperar del subárbol con raíz a todos los elementos más jóvenes que $x \in N(a)$.

El problema que persiste también en este método es que los radios de cobertura pueden quedar sobredimensionados, pues nunca se reduce para reajustarse por el elemento eliminado. Es decir, si eliminamos un elemento x, cada ancestro $a \in A(x)$ para el cual x era el elemento más distante en su subárbol tendrá posiblemente su $R(a)$ sobredimensionado. A pesar de ello, este problema no parece afectar demasiado a las búsquedas, ya que no se degradan significativamente a medida que se eliminan más elementos.

5.3.1. Análisis

Supongamos que tenemos un árbol de aridad A completamente balanceado con n elementos y ya sabemos que en ese caso la altura máxima del árbol es $h = \log_A(n(A-1)+1) - 1$. Existen n posibles raíces de subárboles a eliminar, o sea hay n posibles posiciones para x. Si el elemento a eliminar es la raíz del árbol éste es un caso extremo que ocurre con probabilidad $1/n$ y en ese caso la reconstrucción realiza casi el mismo trabajo que se realizó para construir el árbol, o sea $nA\log_A n$ (ver Sección 4.2.1). Multipli-

```
ReconstruirTS (Nodo a, Nodo x)
1.  T ← RecuperarTS (a,x)
2.  Ordenar T, en orden creciente de timestamp
3.  N(a) ← N(a) − {x}
4.  For v ∈ T
5.     Insertar (a, v) /* sin modificar su timestamp */
```

Figura 5.10: Algoritmo para reconstruir el subárbol con raíz a en un *SA-tree* dinámico, luego de la eliminación de $x \in N(a)$.

cando esto por la probabilidad $1/n$ de que ocurra, nos da un aporte de $O(\log_A n)$ al costo promedio total, que como veremos es de orden inferior. En otro caso, supongamos que el nodo a eliminar se encuentra en el primer nivel del árbol: hay A nodos posibles y al eliminar cualquiera de ellos se debería reconstruir el árbol completo, reinsertando en promedio $(n-1)/2$ elementos (considerando que la mitad de los elementos serán más jóvenes) y cada reinserción costaría a lo sumo $A\ h$ comparaciones. Si x está en el nivel 2, hay A^2 posibilidades para x, y en ese caso se deberán reinsertar $(n-(A+1))/2A$ elementos y cada reinserción costaría $A(h-1)$. En el caso general, si x está en el nivel $i+1$ hay A^{i+1} posibilidades para x, se deberán reinsertar $(n-(A^{i+1}-1)/(A-1))/2A^i$ elementos y cada reinserción costaría $A(h-i)$ comparaciones.

Planteamos ahora la siguiente fórmula para obtener el costo promedio de una reconstrucción $R(n)$. En el siguiente desarrollo se excluye, por simplicidad, el caso particular de eliminar la raíz del árbol (como dijimos aportaba un término de orden inferior a la suma).

$$
\begin{aligned}
R(n) &= \sum_{i=0}^{h-1} \frac{A^{i+1}}{n} \cdot \frac{n-((A^{i+1}-1)/(A-1))}{2A^i} \cdot A\,(h-i) \\
&= \frac{A^2}{2n} \cdot \sum_{i=0}^{h-1} \Big(n - \frac{(A^{i+1}-1)}{(A-1)}\Big) \cdot (h-i) \\
&= \frac{A^2}{2} \cdot \sum_{i=0}^{h-1} (h-i) - \frac{A^2}{2n(A-1)} \cdot \sum_{i=0}^{h-1} (A^{i+1}-1)(h-i) \\
&= \frac{A^2}{2} \cdot \frac{h(h-1)}{2} - \frac{A^2}{2n(A-1)} \cdot \sum_{i=0}^{h-1} (A^{i+1}h - h - iA^{i+1} + i) \\
&= \frac{A^2}{4}h(h-1) - \frac{A^2}{2n(A-1)} \cdot \Big((n-1)h - h^2 - \sum_{i=0}^{h-1} iA^{i+1} + \frac{h(h-1)}{2}\Big) \\
&= \frac{A^2}{4}h(h-1) - \frac{A^2}{2n(A-1)} \cdot \Big(nh - h - h^2 - A \cdot \sum_{i=0}^{h-1} iA^i + \frac{h^2-h}{2}\Big) \\
&= \frac{A^2}{4}h(h-1) - \frac{A^2}{2n(A-1)} \cdot \Big(nh - \frac{3h}{2} - \frac{h^2}{2} - A \cdot \Big((h-1)(n-1) + \frac{h+1-n}{A-1}\Big)\Big) \\
&= \frac{A^2}{4}h(h-1) - \frac{A^2h}{2(A-1)} + \frac{3A^2h + A^2h^2}{4n(A-1)} + \frac{A^3(h-1)}{2n(A-1)} + \frac{A^3(n-1)}{2n(A-1)} + \frac{A^3(h+1-n)}{2n(A-1)^2} \\
&= \frac{A}{4} \cdot \log_A^2 n + O(\log_A n)
\end{aligned}
$$

Por lo tanto, el costo medio de reconstrucción de subárboles es $(A^2/4)\ \log_A^2 n\ (1 + o(1))$ que es efectivamente más costoso que la reinserción de a elementos (o de subárboles), diferencia que aumenta a medida que aumentamos la aridad del árbol.

5.4. Combinando Métodos

Tenemos cuatro métodos. El uso de nodos ficticios elimina elementos sin costo pero degrada el desempeño de la búsqueda del árbol. La reinserción de subárboles y de a elementos hacen una reinserción de subárbol costosa, pero tratan de mantener la calidad de la búsqueda del árbol. La reconstrucción de subárboles hace una reconstrucción aún más costosa, pero logra que la búsqueda no se degrade. Observar que el costo de reinsertar un subárbol, o de reconstruirlo, no sería muy diferente si contuviera nodos ficticios, con lo cual eliminaríamos varios nodos ficticios (recordar que se crea uno por cada eliminación) al precio de una sola reinserción o reconstrucción. La idea general es entonces dejar que aparezcan varios nodos ficticios y luego eliminarlos todos con una única reinserción o reconstrucción, balanceando entre los extremos de dejar los nodos ficticios para siempre y de nunca permitir que existan, y amortizando así el alto costo de la reinserción o de la reconstrucción sobre muchas eliminaciones.

Nuestra idea es asegurar que cada subárbol tenga a lo sumo una fracción α de nodos ficticios[1]. Decimos que tales subárboles son "compensados". Cuando marcamos un nuevo nodo como ficticio, verificamos si no tenemos una descompensación. En ese caso, se borra x y se reinsertan sus subárboles. La única diferencia es que nunca reinsertamos un subárbol cuya raíz sea un nodo ficticio, sino que descomponemos el subárbol y descartamos la raíz ficticia.

Surge una complicación al remover el subárbol con raíz x porque se pueden descompensar varios ancestros de x, aún si x fuera sólo una hoja que puede removerse directamente, y aún si el ancestro no tiene como raíz un nodo ficticio. La Figura 5.11 muestra dos ejemplos de situaciones posibles al eliminar un elemento x del árbol. A la izquierda se ilustra la situación en la que si usamos un factor $\alpha = 1/3$ se descompensan todos los ancestros de x ($A(x)$). A la derecha aparece un árbol en el que si usamos un factor $\alpha = 1/6$ se descompensan el nodo x (ficticio) y sus ancestros no ficticios. Los nodos resaltados son los nodos ficticios del árbol.

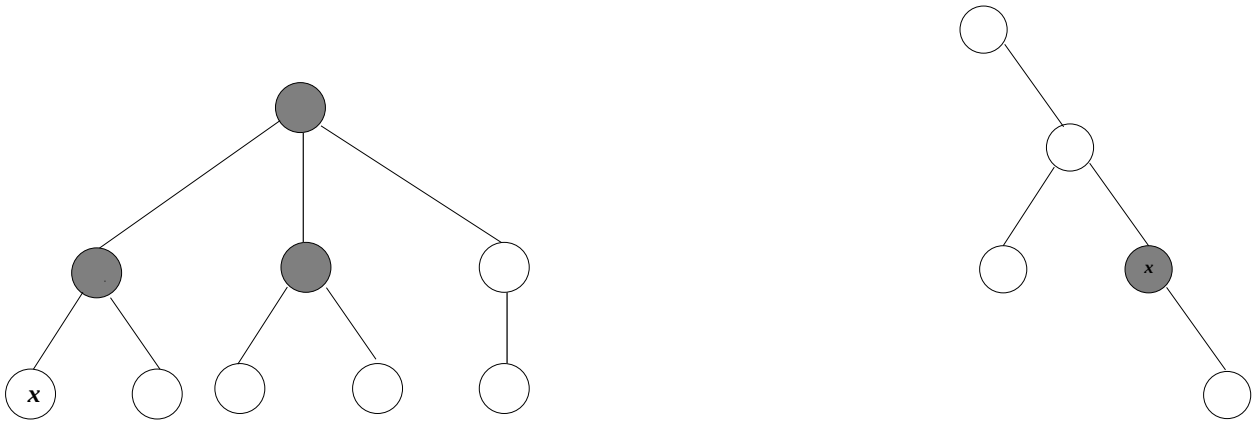


Figura 5.11: Un ejemplo de árbol en el que si se elimina el nodo x se descompensan todos sus ancestros.

[1]En algunos casos hablamos del porcentaje α de nodos ficticios permitidos, respecto del total de nodos del árbol, en lugar de expresarlo como la fracción.

Entonces, optamos por una solución simple. Buscamos el ancestro más bajo b de x que se haya descompensado y reinsertamos o reconstruimos todo el subárbol con raíz b. Reinsertamos subárboles completos sólo cuando ellos no contengan nodos ficticios para que esta reinserción mejore la compensación del árbol en general. En el caso de la reconstrucción de subárboles o de la reinserción elemento por elemento sólo consideramos los nodos no ficticios en el proceso. A pesar de que parecería más apropiado buscar el ancestro más alto de x que se haya descompensado y reinsertar o reconstruir su subárbol, esta opción aumenta considerablemente el costo de la reinserción o reconstrucción (multiplicamos aproximadamente por A la cantidad de elementos a reinsertar cada vez que subimos un nivel). Además es posible que al reinsertar los elementos del subárbol con raíz b algunos elementos vuelvan a elegir el mismo camino y así compensen a los ancestros descompensados de b.

El resultado es que no garantizamos realmente que la fracción de ficticios sea inferior a α, aunque en general se cumple. Por ejemplo, observamos que en el espacio de vectores de dimensión 15, usando un $\alpha = 30\,\%$, eliminando el 10 % de los elementos el α "real" es 2.2 %, eliminando el 20 % es 3.9 %, eliminado el 30 % es 5.5 % y eliminando el 40 % alcanza el 6.6 %; lo que claramente es menor que el α elegido como parámetro.

La técnica combinando con reinserción tiene un buen desempeño, aún si reinsertamos los elementos uno por uno (en lugar de subárboles completos). Si tuviéramos la garantía de reinsertar un subárbol sólo cuando una fracción α de sus elementos son ficticios, esto significaría que si m fuera el tamaño del subárbol a reconstruir, pagaríamos $m(1-\alpha)$ reinserciones por cada $m\alpha$ eliminaciones hechas en el subárbol. De aquí, el costo amortizado de una eliminación sería a lo sumo $(1-\alpha)/\alpha$ veces el costo de una inserción, es decir, $((1-\alpha)/\alpha)\ A \log_A n$. Hacemos un razonamiento similar para la combinación de reconstrucción de subárboles con nodos ficticios: reconstruimos un subárbol sólo cuando una fracción α de sus elementos son ficticios, esto significa que si m fuera el tamaño del árbol a reconstruir, pagaríamos en promedio $Am(1-\alpha)/2$ reinserciones por cada $m\alpha$ eliminaciones realizadas en ese subárbol. Así llegamos a que el costo amortizado de una eliminación sería $((1-\alpha)/\alpha)\ A^2 \log_A n$.

Asintóticamente, el árbol trabajaría como si permanentemente tuviera una fracción α de nodos ficticios. Por lo tanto, podemos controlar el balance entre costos de búsqueda y de eliminación. Notar que nodos ficticios puros se corresponde a $\alpha = 100\,\%$ y reinserción de subárboles pura (o de a elementos pura o reconstrucción de subárboles pura) corresponde a $\alpha = 0\,\%$.

5.4.1. Comparación Experimental

Comparemos ahora los cuatro métodos para realizar eliminaciones sobre el espacio del diccionario de palabras en Inglés (descripto en Sección 2.4.3), sobre el espacio de vectores de dimensión 15 y sobre el espacio de vectores de dimensión 5 con distribución uniforme (la descripción de estos espacios se encuentra en Sección 2.4.1).

Mostramos los resultados de dos tipos de experimentos.

(a) Creamos el árbol con el 90 % de los elementos de la base de datos (reservando el 10 % restante aleatoriamente elegido para las consultas), eliminamos un 10 % de los elementos (también elegidos aleatoriamente) y luego consultamos con el 10 % de los elementos que habíamos reservado.

(b) Creamos el árbol con un 60 %,. . .,90 % de los elementos de la base de datos (reservando el 10 % restante aleatoriamente elegido para las consultas), eliminamos suficientes elementos para dejar el 50 % del conjunto en el índice y luego realizamos las consultas.

El experimento tipo **(b)** lo realizamos, tal como se describió en Sección 5.2.3, para considerar cómo afectan las eliminaciones al costo de búsqueda, buscando sobre un índice que contiene la mitad de los elementos de la base de datos. Así, comparamos la búsqueda sobre conjuntos del mismo tamaño donde un porcentaje de los elementos se ha eliminado para dejar el conjunto de ese tamaño. En este caso, por ejemplo, 30 % de eliminaciones significa que se insertaron 49335 elementos y luego se eliminaron 14801, de manera tal de dejar 34534 elementos (la mitad del conjunto), para el diccionario de 69069 palabras en Inglés.

La Figura 5.12 muestra los costos de eliminación para el primer tipo de experimento en el que eliminamos sólo un 10 % de los elementos de la base de datos, para el espacio de palabras y para el espacio de vectores de dimensión 15. La Figura 5.13 muestra los costos de las eliminaciones para el otro tipo de experimento. En ambos casos mostramos los costos de reinserción completa (es decir, reinsertando los subárboles después de cada eliminación), con y sin la optimización final propuesta. Como se puede observar, ahorramos cerca del 50 % de los costos de eliminación con la optimización. También mostramos que raramente se pueden reinsertar subárboles completos, porque reinsertar los elementos de a uno tiene el mismo costo (las gráficas de reinserción de subárbol y reinserción de a elementos optimizada coinciden). Así se podría usar el algoritmo simplificado sin sacrificar mucho. También mostramos el método combinado con $\alpha = 1\,\%$, $3\,\%$ y $5\,\%$ (Figura 5.12). En la Figura 5.13 mostramos valores mayores de α, desde $0\,\%$ (reinserción completa) hasta $100\,\%$ (nodos ficticios puros), como así también porcentajes más grandes de eliminaciones (sólo usaremos de ahora en más la versión optimizada de reinserciones).

Se puede observar que, aún con reinserción completa, el costo de eliminación individual no es tan alto. Por ejemplo, en el diccionario de Inglés, el costo de inserción promedio es cercano a 58 evaluaciones de distancia por elemento. Con el método optimizado cada eliminación cuesta cerca de 173 evaluaciones de distancia, es decir, 3 veces el costo de una inserción. El método combinado mejora sobre éste: usando α tan pequeño como 1 % tenemos un costo de eliminación de 65 evaluaciones de distancia, que se acerca al costo de las inserciones, y con $\alpha{=}3\,\%$ éste se reduce a 35.

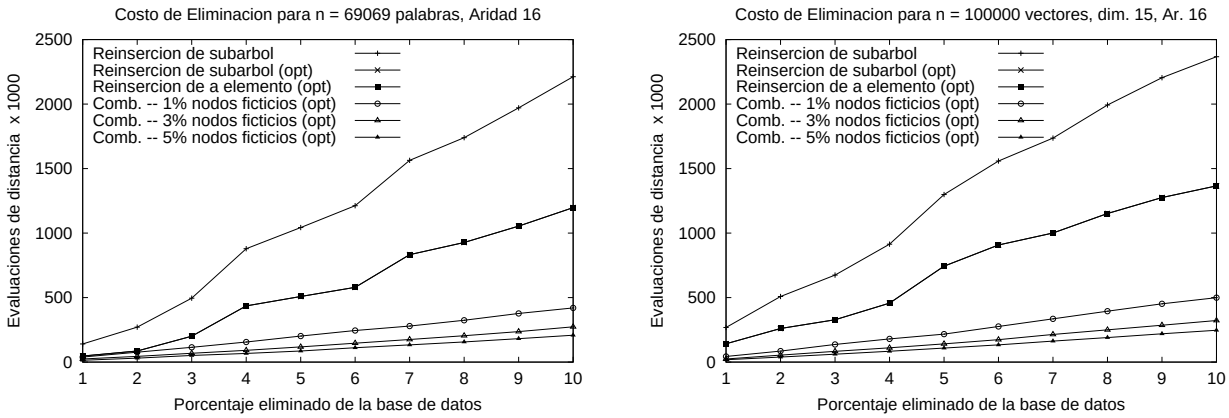


Figura 5.12: Costos de eliminación usando diferentes métodos, al eliminar un 10 % de los elementos.

La Figura 5.14 muestra la comparación de los costos de eliminación entre el método de reconstrucción de subárboles y el de reinserción de a elementos (derecha). Se puede observar que los costos de reconstruir son muy elevados aún en relación con reinsertar de a uno los elementos del subárbol. Se muestra también que a medida que crece la aridad del árbol el costo de la reconstrucción aumenta (izquierda). En ambos casos se ha considerado el espacio de vectores de dimensión 15.

Comparamos los métodos eliminando diferentes porcentajes de la base de datos para que se aprecie no

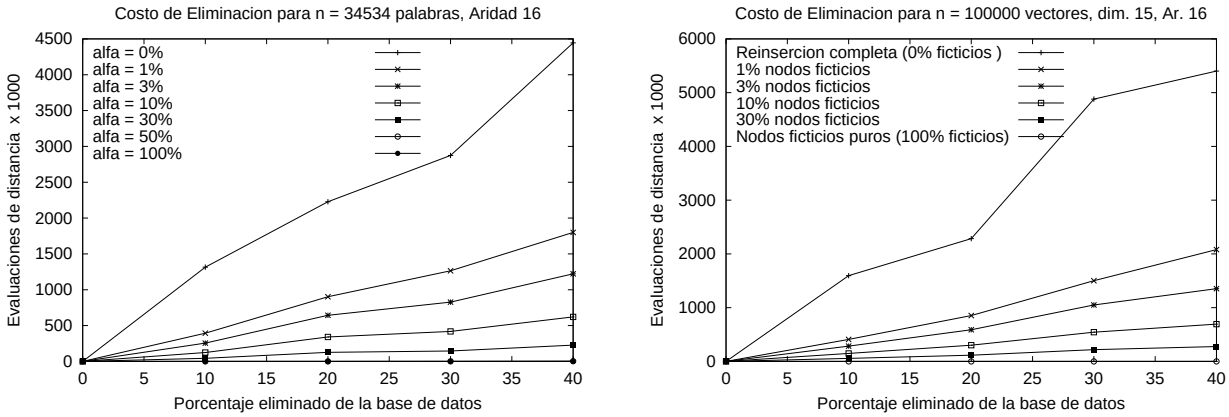


Figura 5.13: Costos de eliminación usando diferentes métodos al eliminar hasta un 40 % de los elementos.

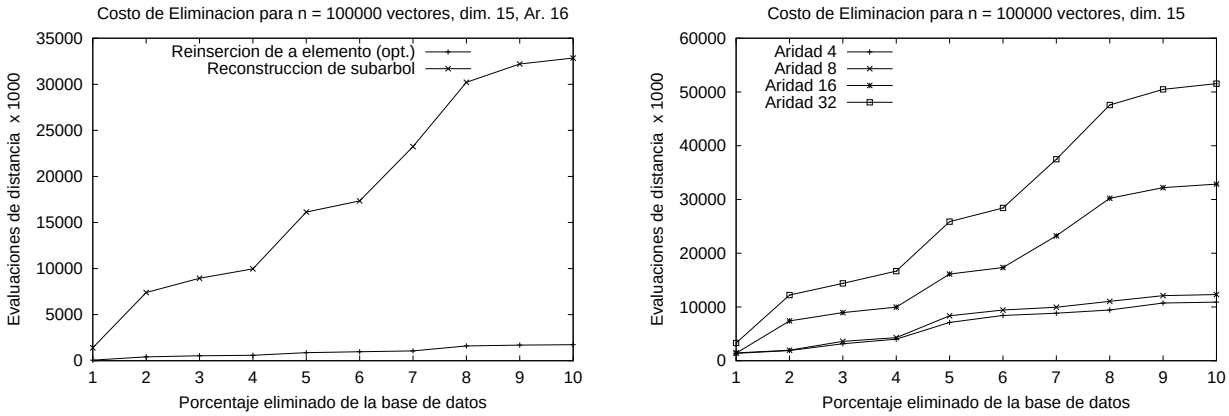


Figura 5.14: Costos de eliminación para el método de reconstrucción de subárboles.

sólo el costo de eliminación por elemento sino también el efecto acumulativo de las eliminaciones sobre la estructura (por ej. por la sobredimensión de los radios de cobertura).

La Figura 5.15 muestra los costos de eliminación para los distintos valores de α para aridades 16 y 32 en el espacio de vectores de dimensión 15. Se puede observar en este caso que el costo de reconstrucción pura es muy elevado, pero en cuanto permitimos un valor de $\alpha = 1\,\%$ los costos de eliminación bajan considerablemente.

Consideraremos ahora el comportamiento de las búsquedas luego de eliminarle elementos al árbol. La Figura 5.16 muestra algunos resultados. Como se puede ver aún con reinserción pura ($\alpha = 0\,\%$) la calidad de la búsqueda se degrada, aunque apenas perceptiblemente y no monótonamente con el número de eliminaciones realizadas. A medida que α crece, los costos de búsqueda se incrementan debido a la necesidad de entrar en cada hijo de los nodos ficticios. La diferencia en costos de búsqueda deja de ser razonable tan pronto como $\alpha = 10\,\%$, y de hecho es significativo aún para $\alpha = 1\,\%$. Así uno tiene que elegir el correcto balance entre costos de eliminación y búsqueda dependiendo de la aplicación. Un buen balance para palabras es $\alpha = 1\,\%$.

La Figura 5.17 muestra los datos para permitir comparar el cambio en los costos de búsqueda a medida

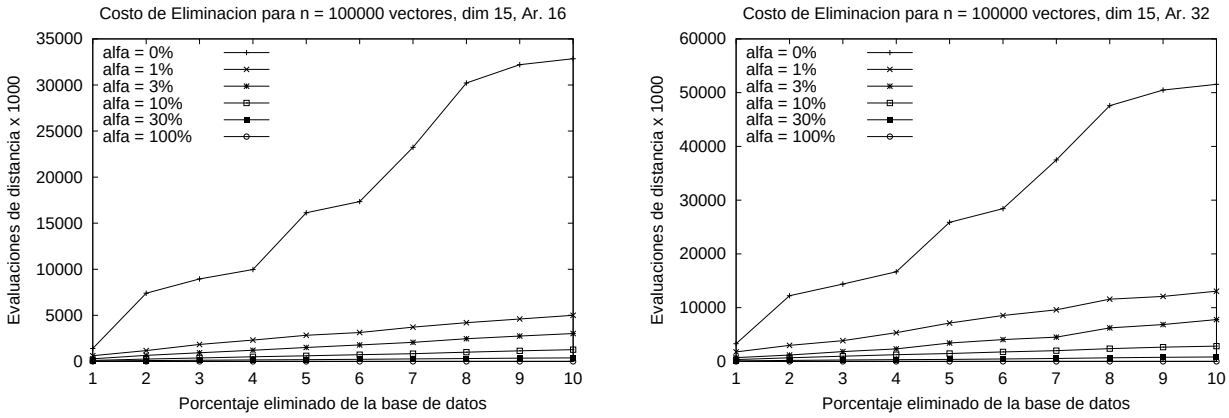


Figura 5.15: Costos de eliminación combinando reconstrucción de subárboles con nodos ficticios.

que crece el α para el espacio de las palabras (arriba) y para el espacio de vectores con distribución uniforme de dimensión 15 (abajo).

La Figura 5.18 muestra algunos resultados para el método combinado con reconstrucción de subárbol para el espacio de vectores de dimensión 15. Como se puede ver en este caso aún considerando un valor de $\alpha = 10\,\%$ la calidad de la búsqueda no se degrada significativamente con el número de eliminaciones realizadas. A medida que α crece es de esperar que los costos de búsqueda se incrementen debido a la necesidad de entrar en cada hijo de los nodos ficticios, pero es evidente que la reconstrucción de subárboles logra un mejor desempeño en estos casos que las combinaciones con reinserción de subárbol o de a elementos. La diferencia en costos de búsqueda deja de ser razonable tan pronto como $\alpha = 30\,\%$, pero de hecho no es significativo para los α menores. Así uno tiene que elegir el correcto balance entre costos de eliminación y búsqueda dependiendo de la aplicación. Un buen balance para vectores de dimensión 15 es $\alpha = 10\,\%$. La Figura 5.19 permite comparar el cambio en los costos de búsqueda a medida que crece el α (para el mismo espacio y método anterior).

La Figura 5.20 muestra los costos de búsqueda al eliminar usando el método combinado de reconstrucción de subárboles con nodos ficticios para el espacio de vectores de dimensión 5. Arriba mostramos los costos de búsqueda al eliminar el 10 % (izquierda) y 40 % (derecha) de la base de datos como se comportan los distintos valores de α. Abajo comparamos los costos de búsqueda para los diferentes porcentajes eliminados para $\alpha = 1\,\%$ (izquierda) y $\alpha = 10\,\%$ (derecha).

5.5. Análisis de las Alternativas

Como se ha podido observar, existen distintas opciones para realizar eliminaciones en un *SA-tree dinámico*. La opción presentada en Sección 5.2.1 (*nodos ficticios*) tiene la ventaja de lograr eliminar un elemento sin costo, pero al momento de las búsquedas su desempeño es pobre. Esto la descarta para la mayoría de las aplicaciones.

Por otra parte se analizaron dos opciones distintas de reinserción de los subárboles del elemento eliminado (Sección 5.2.2 y Sección 5.2.3) y una opción de reconstrucción de subárboles (Sección 5.3).

La primera trata de aprovechar el trabajo ya realizado cuando se insertaron los elementos, tratando de

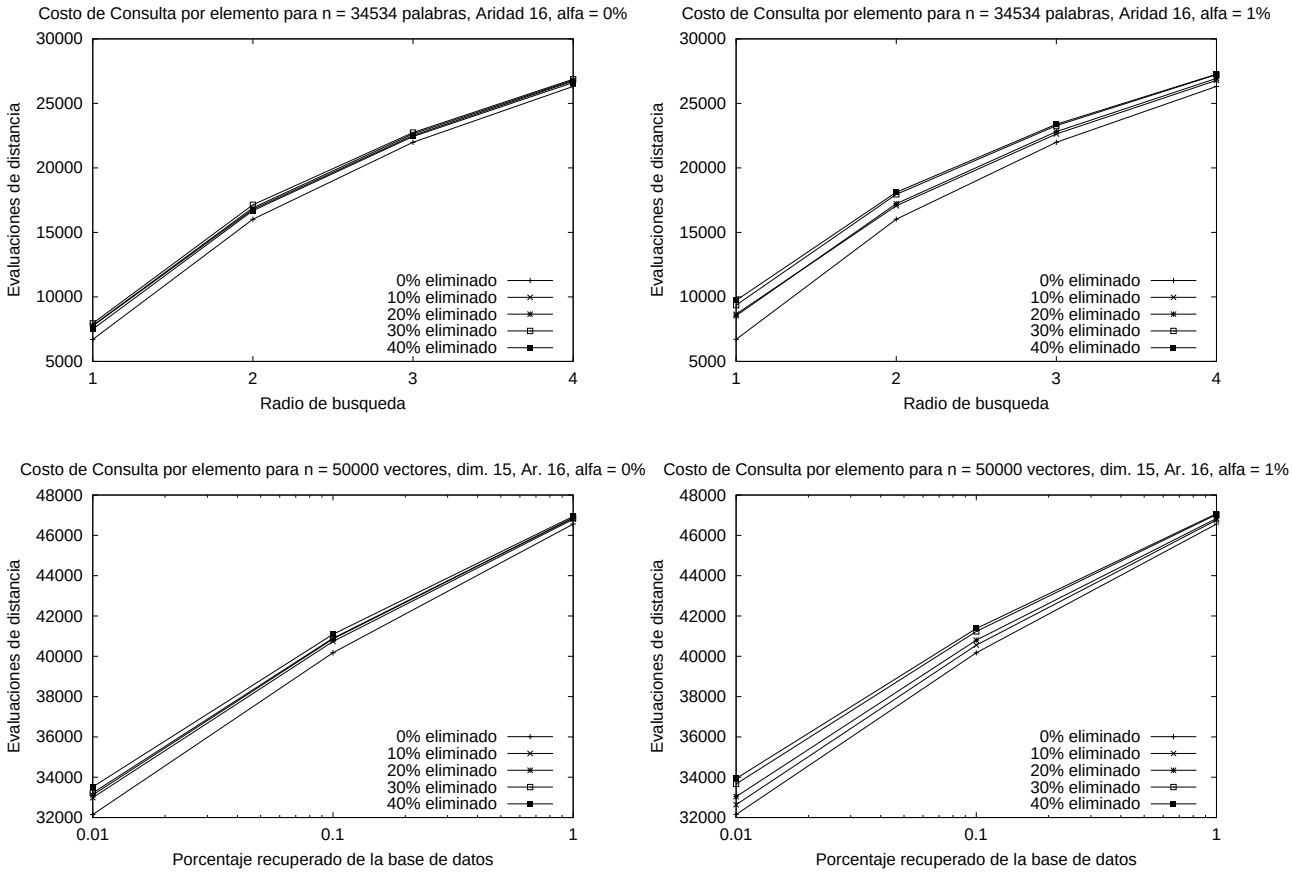


Figura 5.16: Costos de búsqueda usando el método de eliminación que combina reinserción con nodos ficticios. A la izquierda el caso de $\alpha = 0\,\%$, y a la derecha de $1\,\%$.

reinsertar subárboles completos. Esta técnica trata de minimizar los costos de la reinserción, aunque no siempre lo logra, respecto de reinsertar uno por uno los elementos; pero degrada la búsqueda en parte por la posible sobredimensión de los radios.

La segunda opción de reinserción en cambio reincorpora uno por uno los elementos considerándolos como si fueran totalmente nuevos. Esta técnica paga un precio más alto por la reinserción, aunque su búsqueda se comporta mejor por no sobredimensionar los radios de cobertura debido a que reinserta uno por uno los elementos.

La tercera opción reconstruye el subárbol del padre del elemento eliminado, tratando de dejar el árbol tal como hubiese sido si nunca hubiese existido dicho elemento. Esta técnica logra mantener el buen desempeño de la búsqueda, pero a costa de un muy alto costo de reconstrucción.

Ambas opciones de reinserción mejoran aplicando la optimización propuesta (basada en la información que se tiene por conocer los timestamps de los elementos).

Además hay que tener en cuenta que en los experimentos realizados, sobre todos los espacios métricos analizados, sólo se logran reinsertar completos subárboles muy pequeños. Por lo tanto, el método que reinserta subárboles no logra mejorar significativamente el costo de eliminación respecto del que reinsertaba elemento por elemento y el desempeño de su búsqueda, en general, es peor.

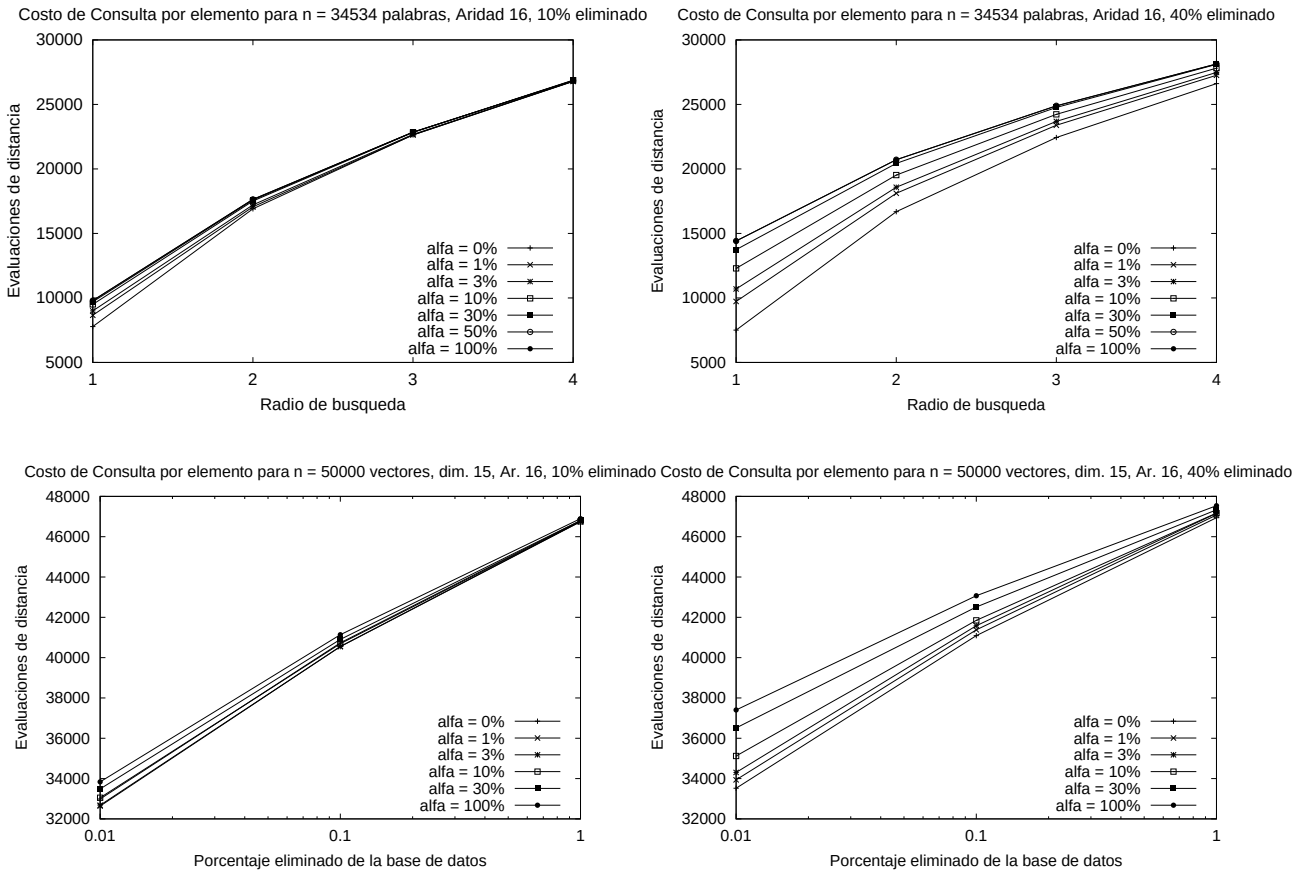


Figura 5.17: Costos de búsqueda usando el método de eliminación que combina reinserción con nodos ficticios, comparando α. A la izquierda eliminamos el 10 % de la base de datos, a la derecha el 40 %.

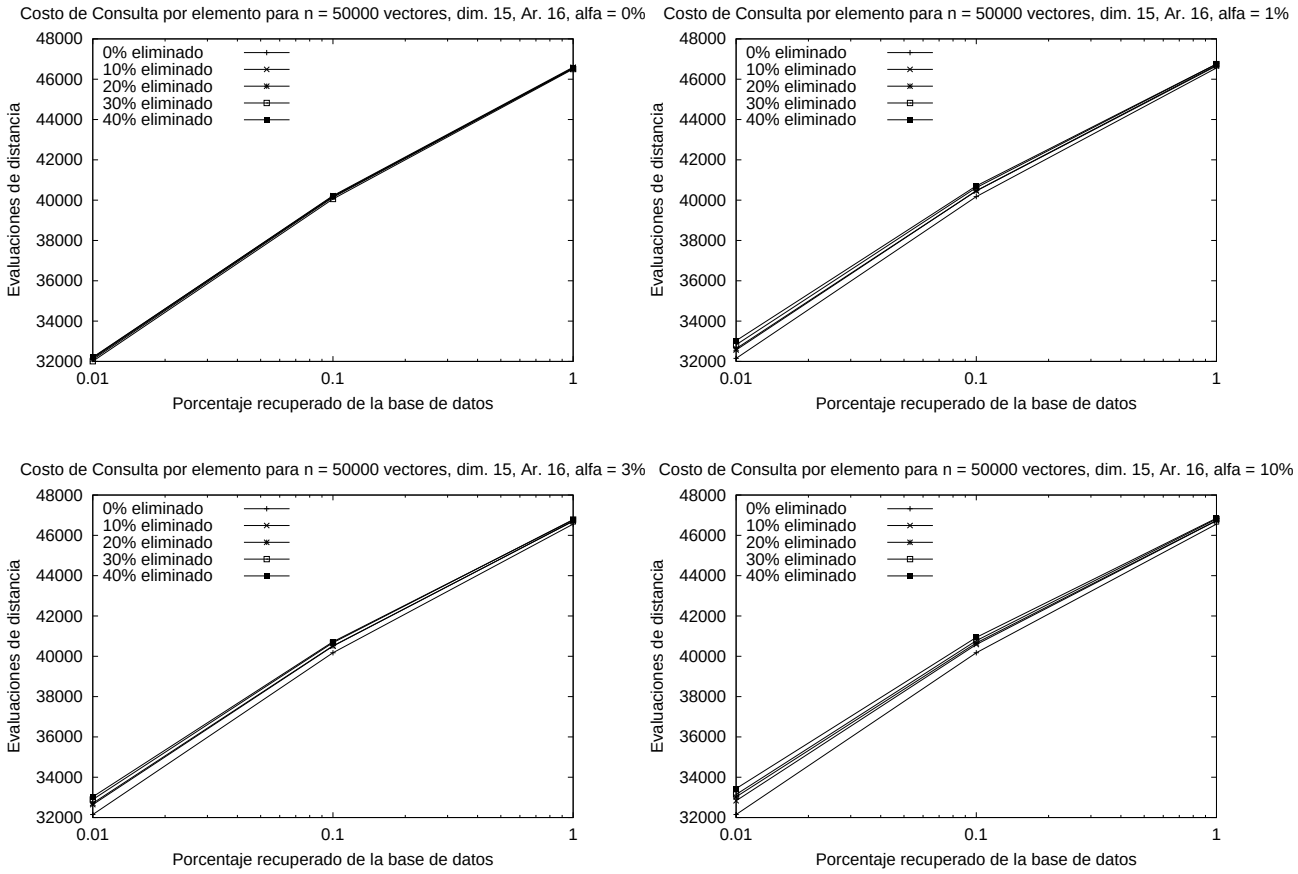


Figura 5.18: Costos de búsqueda usando el método de eliminación que combina reconstrucción con nodos ficticios. A la izquierda los casos de $\alpha = 0\,\%$ (arriba) y $3\,\%$ (abajo), y a la derecha de $1\,\%$ (arriba) y $10\,\%$ (abajo).

Por otra parte, se presenta una alternativa en la Sección 5.4 que intenta reunir en un parámetro (α) qué es más significativo, si el costo de eliminación o el de consulta. Cuando tomamos $\alpha = 100\,\%$ no tenemos costo de eliminación, pero pagamos un mayor precio en las búsquedas. Por el contrario, si tenemos $\alpha = 0\,\%$ pagamos un mayor precio por la eliminación, pero intentamos beneficiar las búsquedas. Además, esta combinación de tener nodos ficticios y reinserción se puede realizar con cualquiera de las dos formas de reinserción, y también con la técnica de reconstrucción de subárboles.

Finalmente consideramos que se obtienen mejores métodos combinando con nodos ficticios reinserción elemento por elemento (usando la optimización), y reconstrucción de subárboles. Aunque en cualquiera de las dos combinaciones agregamos un parámetro a sintonizar, no consideramos perjudicial su uso, porque podemos tomarlo como $\alpha = 0\,\%$ obteniendo el método de reinserción o reconstrucción pura (según sea el caso); pero, si en alguna aplicación particular es de interés bajar los costos de eliminación aún a costa de tener posiblemente un mayor costo de búsqueda, se lo puede usar para adecuarse más al comportamiento deseado.

Para decidir entre usar reinserción de a elemento o reconstrucción de subárboles hay que analizar si se desea beneficiar las eliminaciones o las búsquedas, ya que la reinserción logra un mejor costo de eliminación a costa de degradar la búsqueda, y la reconstrucción mejora el desempeño de las búsquedas a costa de

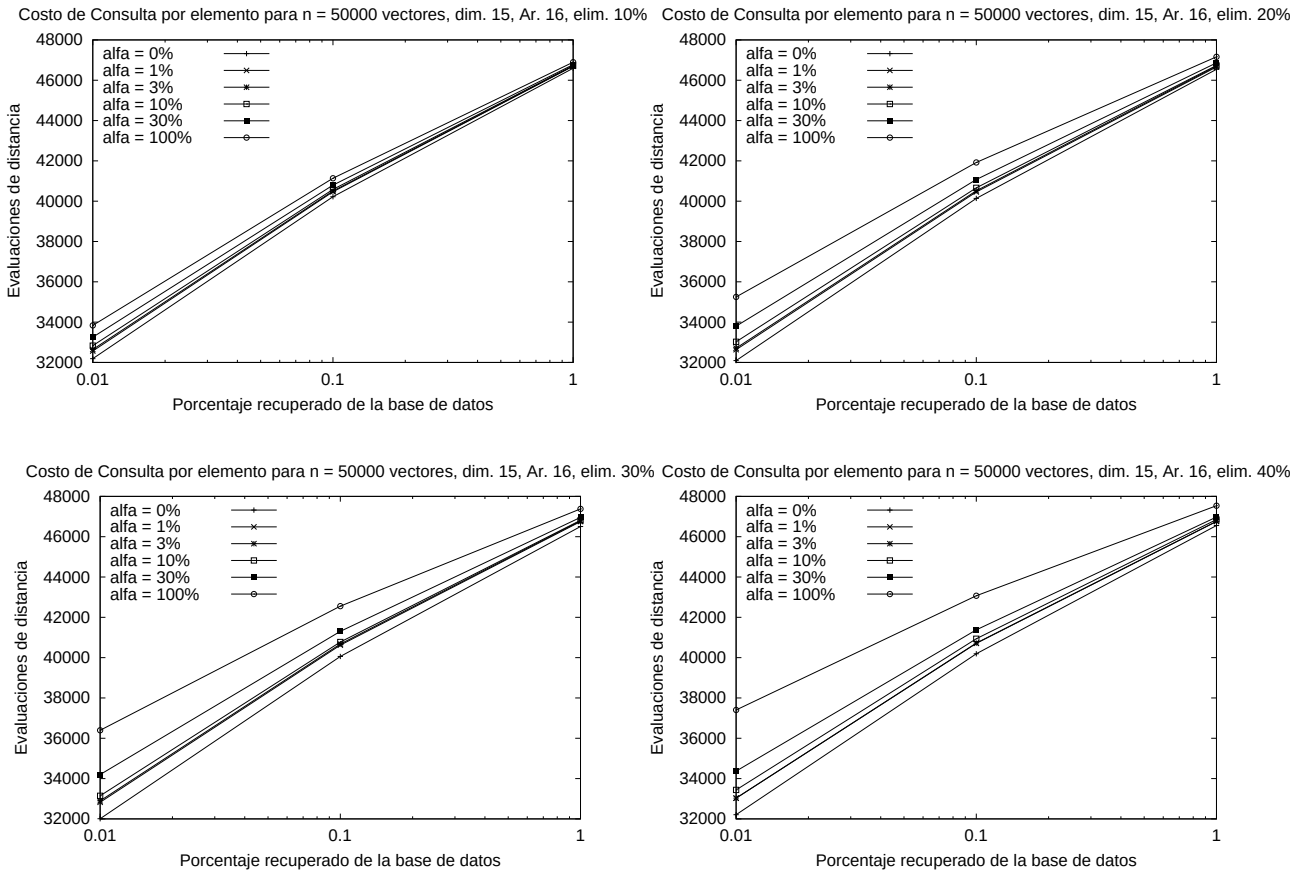


Figura 5.19: Costos de búsqueda usando el método de eliminación que combina reconstrucción con nodos ficticios, comparando α y eliminando 10 %, 20 %, 30 % y 40 % de los elementos de la base de datos.

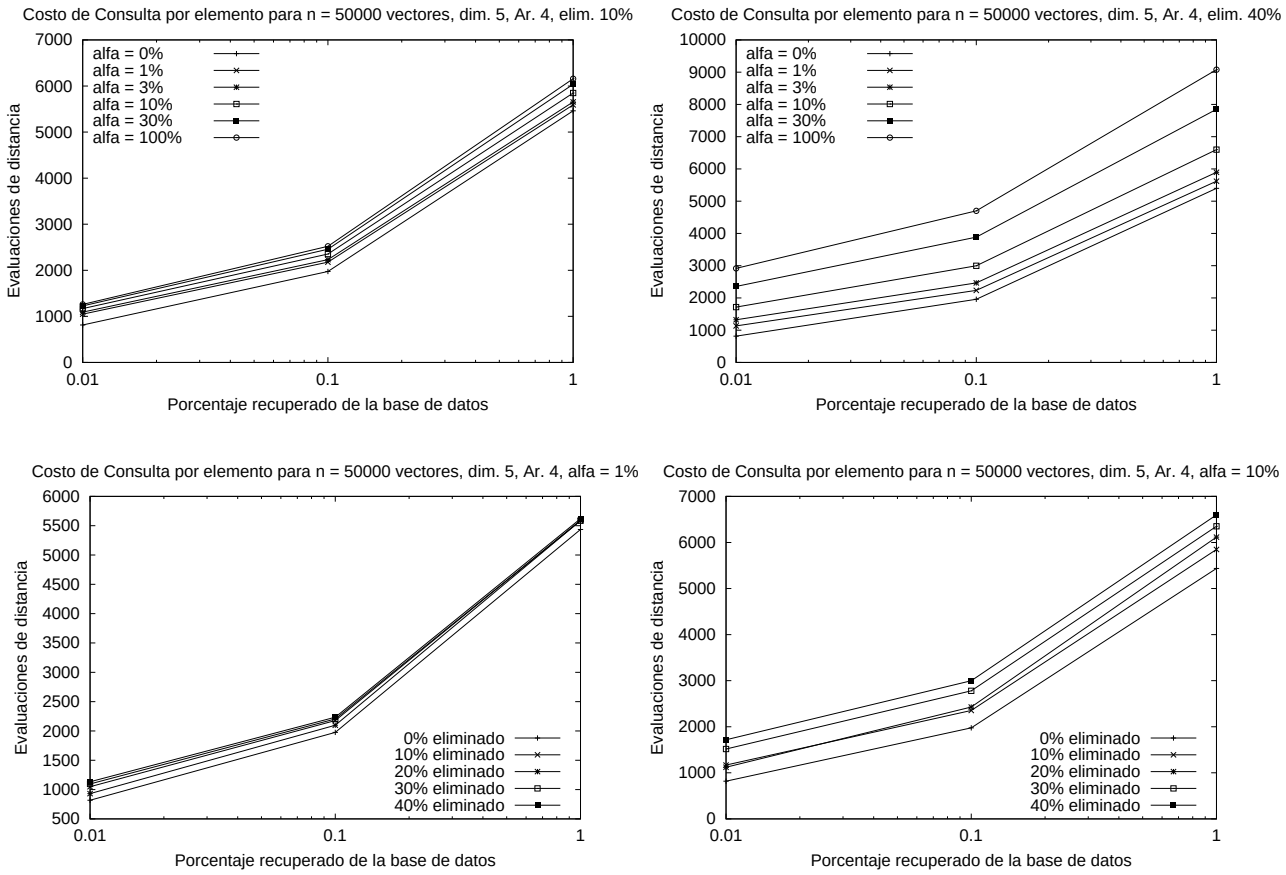


Figura 5.20: Costos de búsqueda usando el método de eliminación que combina reconstrucción con nodos ficticios, comparando α y porcentajes eliminados de la base de datos.

aumentar los costos de eliminación. Pero como es de esperar que, en general, se realicen más búsquedas en la estructura que eliminaciones resulta más adecuado el método combinado con reconstrucción de subárboles. Además es bastante grave que las búsquedas se degraden a medida que se insertan y borran elementos, porque no es sólo un problema de cantidad de elementos eliminados sino también de estabilidad en el tiempo.

La degradación que sufren en la búsqueda los métodos que reinsertan subárboles o elementos, en parte se debe a la sobredimensión de los radios, pero tal como se evidencia en la Figura 5.6 ésta no es la única razón. Creemos que la degradación se debe también a que al eliminar un elemento x de los vecinos de a se pierde la capacidad de usar la distancia a x durante la minimización, y que aunque al reinsertar su subárbol algunos de sus elementos se ubiquen en $N(a)$, ellos tendrán menor posibilidad para disminuir más rápidamente el radio (por reinsertarse como vecinos más jóvenes en $N(a)$). Esto nos daría la razón por la que en el método de reconstrucción de subárboles no se degrada la búsqueda a medida que crece el número de elementos eliminados.

La Figura 5.21 muestra la comparación entre los métodos de reinserción de a elemento y reconstrucción de subárboles para el espacio de vectores de dimensión 15. Se han graficado en cada caso los costos de eliminación contra los de búsqueda para cada valor de α, para cuando se elimina el 10 % o el 40 % de los elementos, y considerando las búsquedas por rango que recuperan el 0.01 % y 1 % de los elementos. Se puede observar, por ejemplo, que cuando se elimina el 10 % de los elementos de la base de datos el método de reconstrucción de subárbol tiene mayores costos de eliminación para lograr el mismo costo de búsqueda que el método de reinserción; pero, sin embargo, permite obtener costos de consulta menores (que son imposibles de obtener con el método de reinserción) si se pagan mayores costos de eliminación. Por otro lado, cuando el porcentaje de elementos eliminados es del 40 %, ya el método de reconstrucción ofrece un mejor compromiso que el de reinserción: el método de reconstrucción obtiene un mismo costo de búsqueda a un menor costo de eliminación.

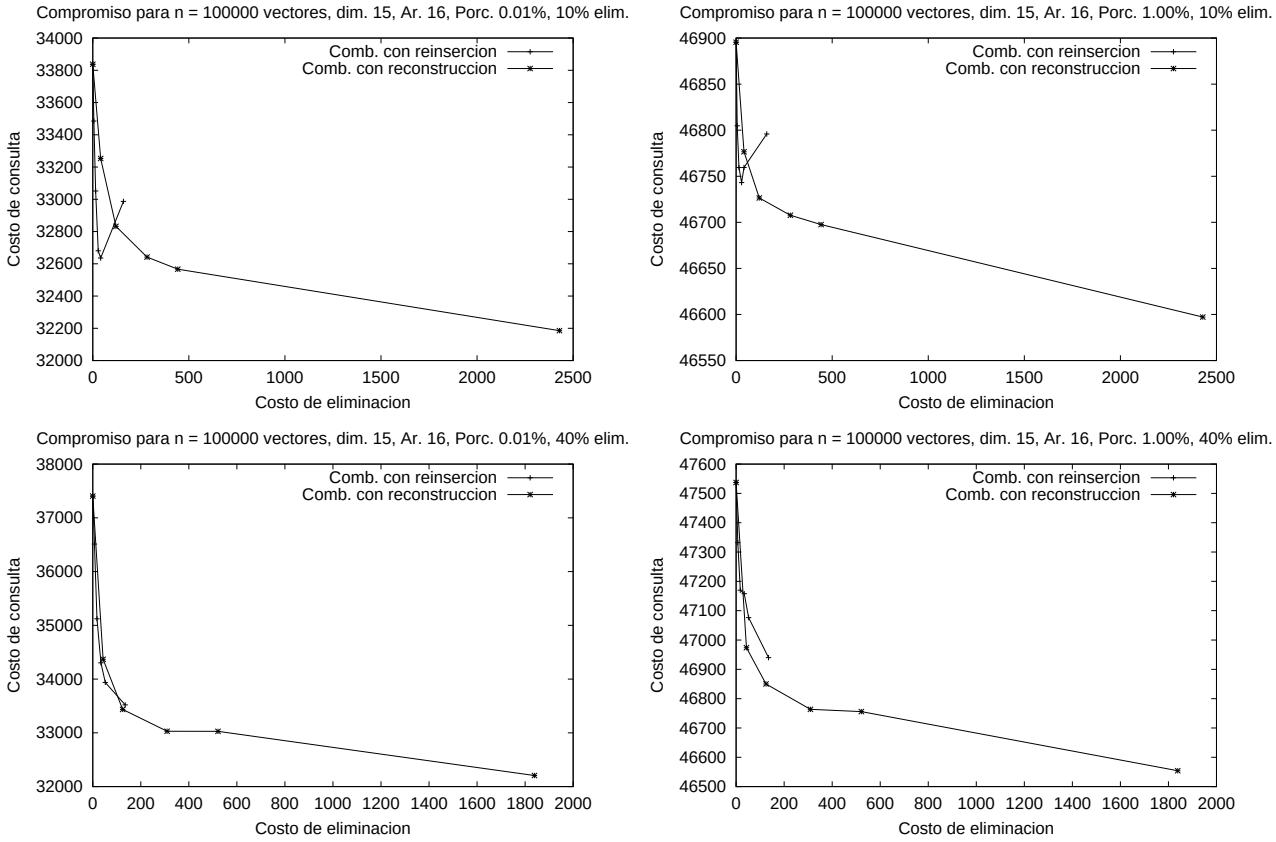


Figura 5.21: Compromiso entre el método de reinserción de a elemento y el método de reconstrucción de subárboles, considerando el 10 % (arriba) y para el 40 % de elementos eliminados (abajo) y para las búsqueda por rango que recuperan el 0.01 % (izquierda) y 1 % (derecha).

Capítulo 6

Conclusiones

El propósito de esta tesis fue conseguir una estructura de datos dinámica y competitiva para búsqueda en espacios métricos de alta dimensionalidad, las cuales no son muy comunes. Existen numerosas aplicaciones reales en donde es necesaria una estructura de datos con dichas características. En este capítulo presentaremos, de manera general, lo que hemos logrado y lo que aún nos queda por hacer en el futuro.

6.1. Aportes

La búsqueda en espacios métricos de alta dimensionalidad requiere de buenas estructuras de datos que permitan realizar dichas búsquedas de manera eficiente, que sean razonables de construir y que admitan dinamismo, es decir que se puedan construir incrementalmente y que sea posible eliminarles físicamente elementos.

El *SA-tree* ya ha demostrado ser muy competitivo en espacios métricos de alta o media dimensionalidad (espacios "difíciles") o para responder a consultas con baja selectividad, pero su principal desventaja es que era una estructura de datos totalmente estática, es decir para su construcción se debían conocer de antemano todos los elementos de la base de datos y luego no se lo podía modificar. Por lo tanto, esto la convertía en poco útil para muchas de las aplicaciones reales.

Así, nuestro interés fue el de producir una versión *dinámica* del *SA-tree*, con el fin de aprovechar sus bondades y salvar su principal desventaja.

Por todo lo expresado anteriormente consideramos que los aportes más relevantes de esta tesis son

- Hemos obtenido distintos algoritmos de inserción eficientes para el *SA-tree* y hemos mostrado que existen más alternativas que aquellas que aparecen en primera instancia. En particular, el algoritmo propuesto finalmente nos permite construir incrementalmente el árbol con costos que *mejoran* ampliamente los costos del *SA-tree* original.
- Hemos logrado algoritmos de eliminación o borrado eficientes en un *SA-tree*, aspecto no muy común en las estructuras de datos existentes para búsqueda en espacios métricos.
- Logramos mantener la búsqueda eficiente y en algunos casos aún mejoramos su desempeño respecto de la versión original de *SA-tree*, esto último especialmente en dimensiones bajas.

- Conseguimos una visión más profunda de la estructura misma, lo que nos permitió descubrir la posibilidad de relajar algunas condiciones, que se planteaban como necesarias para el *SA-tree* original, manteniendo la correctitud de la estructura.

- Agregamos como parámetro de la estructura la aridad, que es fácil de sintonizar y que nos permite adaptarla mejor a la dimensión intrínseca del espacio métrico considerado. Otro parámetro, la fracción máxima de nodos ficticios en el árbol, nos permite sintonizar costo de búsqueda versus costo de eliminación.

- El *SA-tree* original no tenía un buen desempeño en espacios métricos de baja dimensión, ni en la construcción ni en la búsqueda, lo que también hemos logrado superar con nuestro *SA-tree* dinámico, haciéndolo más competitivo en esta área.

- Hemos salvado todas las debilidades del *SA-tree* original, y así, nuestro nuevo *SA-tree* dinámico se presenta como una estructura de datos práctica y eficiente que se puede utilizar en un espectro más amplio de aplicaciones, mientras mantiene las buenas características de la estructura de datos original.

- Otro aspecto interesante en nuestra nueva estructura es que, además de haber demostrado un comportamiento competitivo, da la posibilidad de que se puedan realizar búsquedas por proximidad (de cualquier tipo) considerando un estado *anterior* de la base de datos (si no consideramos los borrados). Para esto basta con controlar cada vez que se alcanza un subárbol si su timestamp es menor que el deseado. Si es así se procede de acuerdo a la búsqueda correspondiente, en otro caso (si el timestamp es mayor que el tiempo límite) se descarta todo el subárbol ya que, como ya se vió todos sus descendientes tendrán un mayor timestamp y no tenemos interés en ellos.

- Aunque nuestros resultados son aplicables principalmente al *SA-tree*, algunos de ellos podrían ser aplicables a otras estructuras.

Creemos que esta tesis constituye un aporte valioso al desarrollo y comprensión del problema de búsqueda en espacios métricos, además de proveer estructura para búsqueda por proximidad muy competitiva y totalmente dinámica, tanto en espacios métricos de alta dimensión como en los de baja.

Cabe mencionar aquí también que los principales resultados de nuestro trabajo han sido publicados en [Nav02, NR01, RHN01, HRBY$^+$02, NR02a, RN02, NR02b].

6.2. Trabajos Futuros

A continuación enunciamos algunas de las posibles extensiones a nuestra estructura y algunas líneas de aplicación, a algunas de ellas pensamos dedicarnos dentro del plan de doctorado, otras ya se están desarrollando:

- Dedicarnos a hacer que el *SA-tree* trabaje eficientemente en memoria secundaria. En ese caso serán relevantes tanto el número de evaluaciones de distancia como el de accesos a disco. Una solución simple es tratar de almacenar subárboles completos en páginas de disco para minimizar el número de páginas leídas durante la búsqueda. Nuestro *SA-tree* dinámico tiene una relación interesante con esto, no sólo porque la parte alta del árbol es sólo de lectura, sino también porque el parámetro de la aridad del árbol nos permite hacer que los vecinos entren en una página de disco.

- Estudiar más a fondo la alternativa de eliminación que usa reconstrucción de subárboles, que es muy prometedora y se podría mejorar. Por ejemplo, se podría recordar qué subárboles están iguales antes y después de la reconstrucción, de modo que al reentrar a ellos, si no han cambiado, no sea necesario recalcular dónde insertar el elemento, sino que se pueda ponerlo donde estaba antes.

- Desarrollar una versión paralela del *SA-tree dinámico*. Este trabajo se ha comenzado a desarrollar por un alumno de Mauricio Marín, de la Universidad de Magallanes (Chile), como su trabajo de fin de carrera.

- Desarrollar una estructura de datos dinámica e híbrida para búsqueda en espacios métricos, que combine la aproximación espacial con técnicas de particiones basadas en pivotes. Este trabajo lo está desarrollando actualmente Diego Arroyuelo como su trabajo final de la Licenciatura en Cs. de la Computación en esta Universidad, bajo la dirección de Gonzalo Navarro y la codirección de Nora Reyes.

- Continuar con algunos de los aspectos que requieren más estudio. Por ejemplo: estudiar si es posible elegir el mejor punto de inserción para un elemento, desde el punto de vista de las búsquedas, o determinar en una eliminación si se reconstruye o no como un compromiso entre el costo de realizar la reconstrucción del subárbol y los beneficios o perjuicios que se producirían en las búsquedas posteriores; sólo por mencionar algunos de ellos.

- Analizar si es posible resolver eficientemente un conjunto de queries, estructurándolo y aprovechando la aproximación espacial, para obtener un mejor desempeño que haber realizado las queries de a una.

- Estudiar si es posible aplicar nuestro *SA-tree* dinámico para resolver eficientemente queries sobre bases de datos métricas como por ejemplo los “join espaciales” o los “join multiway”.

- Analizar si es posible usar el *SA-tree* como el índice que soporte un tipo de datos métrico en una base de datos relacional, lo que implica resolver problemas de control de concurrencia, recuperación frente a fallas, planes de acceso, solución de consultas (en especial el “join” ya mencionado) y memoria secundaria (ya mencionado).

Bibliografía

[Aur91] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

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

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

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

[BKK96] S. Berchtold, D. Keim, and H. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. 22nd Conference on Very Large Databases (VLDB'96)*, pages 28–39, 1996.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM Conference on Management of Data (ACM SIGMOD'97)*, 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.

[BY97] R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

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

[BYL89] R. Baeza-Yates and P. Larson. Performance of B^+-trees with partial expansions. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):248–257, 1989.

[BYN98] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE CS Press, 1998.

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

[Cha94] B. Chazelle. Computational geometry: a retrospective. In *Proc. of the 26th ACM Symposium on the Theory of Computing (STOC'94)*, pages 75–94, 1994.

[Chi94] T. Chiueh. Content-based image indexing. In *Proc. of the 20th Conference on Very Large Databases (VLDB'94)*, pages 582–593, 1994.

[CM97] E. Chávez and J. Marroquín. Proximity queries in metric spaces. In R. Baeza-Yates, editor, *Proc. 4th South American Workshop on String Processing (WSP'97)*, pages 21–36. Carleton University Press, 1997.

[CMBY99] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 38–46. IEEE CS Press, 1999.

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

[CN00a] E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 75–86. IEEE CS Press, 2000.

[CN00b] E. Chávez and G. Navarro. Measuring the dimensionality of general metric spaces. Technical Report TR/DCC-00-1, Dept. of Computer Science, University of Chile, 2000.

[CN01] E. Chávez and G. Navarro. Towards measuring the searching complexity of metric spaces. In *Proc. Mexican Computing Meeting*, volume II, pages 969–978, Aguascalientes, México, 2001. Sociedad Mexicana de Ciencias de la Computación.

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

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

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

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

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

[HRBY+02] N. Herrera, N. Reyes, R. Baeza-Yates, G. Navarro, and E. Chávez. Búsquedas en espacios métricos. In *Proc. IV Workshop de Investigadores en Ciencias de la Computación (WICC'02)*, pages 290–293, Bahía Blanca, Argentina, 2002.

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

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

[MOC96] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996.

[MOV94] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

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

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[Nav02] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[NH84] J. Nievergelt and H. Hinterberger. The grid file: an adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[NN97] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[Nol89] H. Noltemeier. Voronoi trees and applications. In *Proc. International Workshop on Discrete Algorithms and Complexity*, pages 69–74, Fukuoka Recent Hotel, Fukuoka, Japan, 1989.

[NR01] G. Navarro and N. Reyes. Dynamic spatial approximation trees. In *Proc. XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 213–222. IEEE CS Press, 2001.

[NR02a] G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 254–270. Springer, 2002.

[NR02b] G. Navarro and N. Reyes. Improved dynamic spatial approximation trees. In *Proceedings of the XXVIII Latin American Conference on Informatics (CLEI'02)*, page 74, Montevideo, Uruguay, 2002. Abstract in print, complete papers in CD-ROM.

[NVZ92] H. Noltemeier, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – a tool for efficient partitioning of complex schenes of geometric objects. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203. Springer-Verlag, 1992.

[PAA98] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *Proc. ACM SPAA'98*, Puerto Vallarta, Mexico, 1998.

[RHN01] N. Reyes, N. Herrera, and G. Navarro. Búsqueda en espacios métricos: Árbol de aproximación espacial dinámico. In *Proc. III Workshop de Investigadores en Ciencias de la Computación (WICC'01)*, pages 36–38, San Luis, Argentina, 2001.

[RN02] N. Reyes and G. Navarro. Eliminación en arboles de aproximación espacial dinámicos. In *Actas del VIII Congreso Argentino de Ciencias de la Computación (CACIC'02)*, pages 821–833, Buenos Aires, Argentina, 2002. Publicación en CD-rom.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[Sha77] M. Shapiro. The choice of reference points in best-match file searching. *Comm. of the ACM*, 20(5):339–343, 1977.

[SM83] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[Uhl91a] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.

[Uhl91b] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[Ver95] K. Verbarg. The C-Tree: a dynamically balanced spatial index. *Computing Suppl.*, 10:323–340, 1995.

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

[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.

[Yia99] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.

[Yia00] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000. To appear.

Apéndice A

Resultados Experimentales

Mostramos aquí algunos resultados experimentales que permiten apreciar más completamente el comportamiento de nuestro *SA-tree dinámico*. Para ello, analizamos más en detalle algunos resultados para los distintos tipos de espacios métricos considerados en los experimentos (descriptos en la Sección 2.4.1, Sección 2.4.2 y Sección 2.4.3).

A.1. Construcción Incremental

En los experimentos de construcción se construye el árbol incrementalmente, con el 10 %, 20 %, ..., 100 % de los elementos de la base de datos.

Vamos a analizar los resultados obtenidos para los espacios métricos utilizados. En todos los casos al menos hemos realizado la generación del *SA-tree dinámico* con las aridades 4, 8, 16 y 32.

A.1.1. Espacios Vectoriales

Mostramos ahora los costos de construcción para los distintos espacios vectoriales considerados en la Sección 2.4.1. En todos estos espacios hemos utilizado la distancia L_2.

La Figura A.1 muestra la comparación de los costos de construcción para los espacios vectoriales con distribución uniforme considerando dimensiones 5, 10, 15 y 20. Cada espacio consta de 100000 vectores.

Como se puede observar, en todos los casos las aridades bajas son las que obtienen menor costo de construcción. Más aún, para casi todas las aridades consideradas se obtiene un menor costo de construcción que el del *SA-tree* original. La única excepción ocurre en el espacio de dimensión 5, que sólo la construcción con aridad 32 es más costosa que la versión estática.

A modo ilustrativo, se muestran en la Tabla A.1.1 las aridades máximas del *SA-tree* original para cada espacio vectorial considerado:

La Figura A.2 ilustra la comparación entre los costos de construcción para los espacios de vectores generados usando una distribución de Gauss (de media 1 y varianza $0{,}1$), considerando dimensiones 5, 10, 15 y 20. Cada espacio posee 100000 vectores.

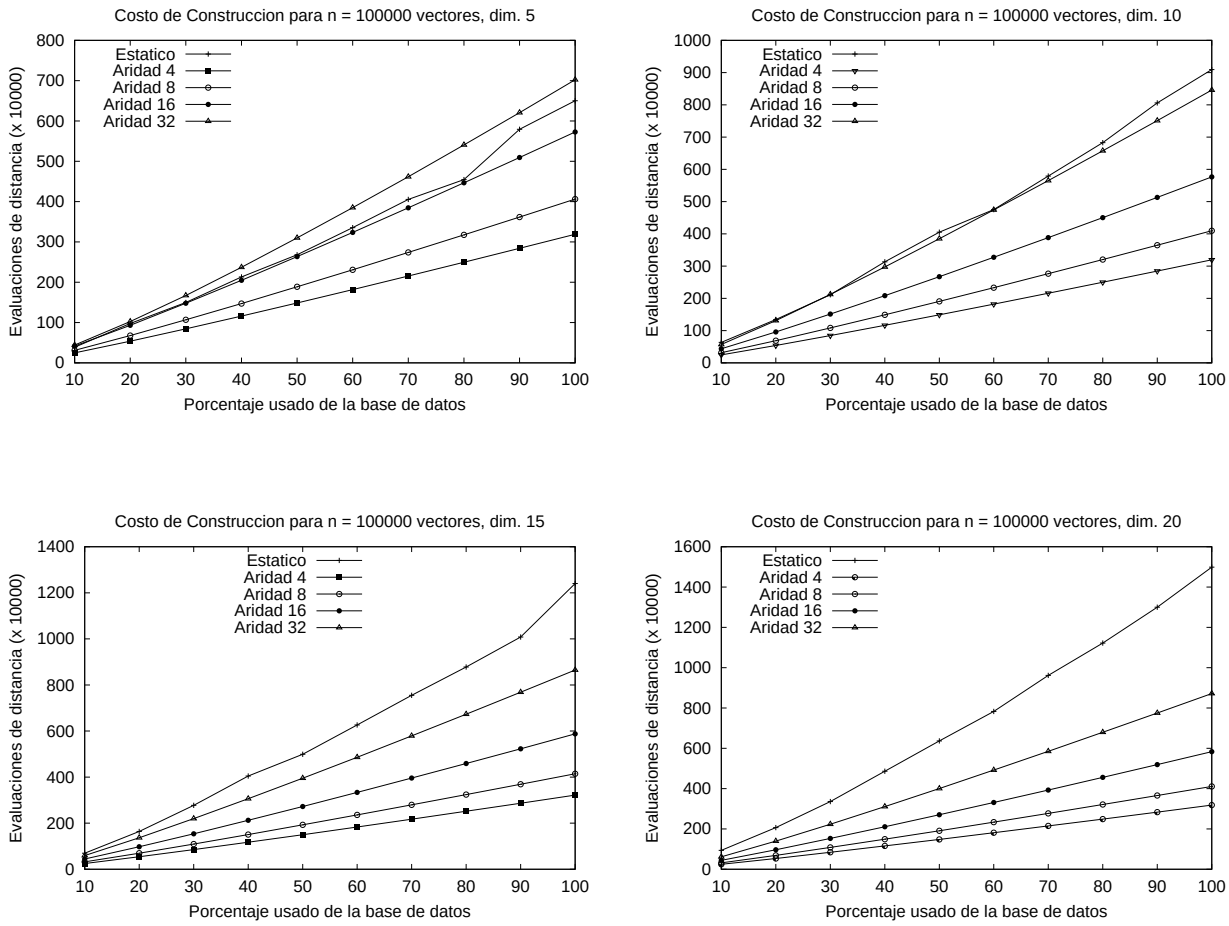


Figura A.1: Comparación entre costos de construcción incremental y estática para espacios vectoriales con distribución uniforme en el cubo unitario.

Espacio métrico	Aridad máxima
Vectores distr. uniforme, dim. 5	10
Vectores distr. uniforme, dim. 10	22
Vectores distr. uniforme, dim. 15	44
Vectores distr. uniforme, dim. 20	60
Vectores distr. Gauss, dim. 5	10
Vectores distr. Gauss, dim. 10	24
Vectores distr. Gauss, dim. 15	44
Vectores distr. Gauss, dim. 20	80
Vectores distr. Gauss, dim. 101 (200 clusters)	79
Vectores de imágenes, dim. 20	34

Tabla A.1: Aridades máximas del *SA-tree* original para los distintos espacios vectoriales considerados.

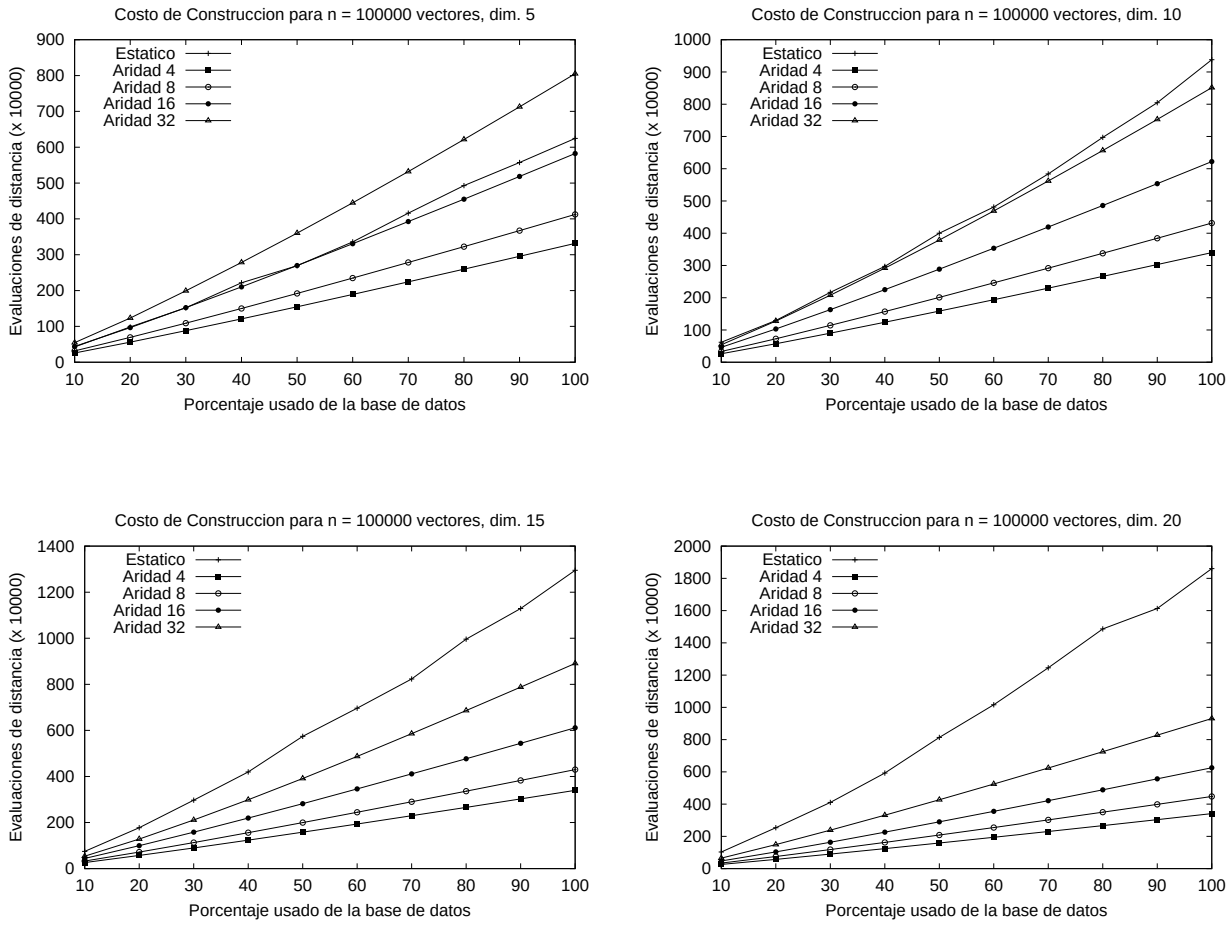


Figura A.2: Comparación entre costos de construcción incremental y estática para espacios vectoriales con distribución de Gauss.

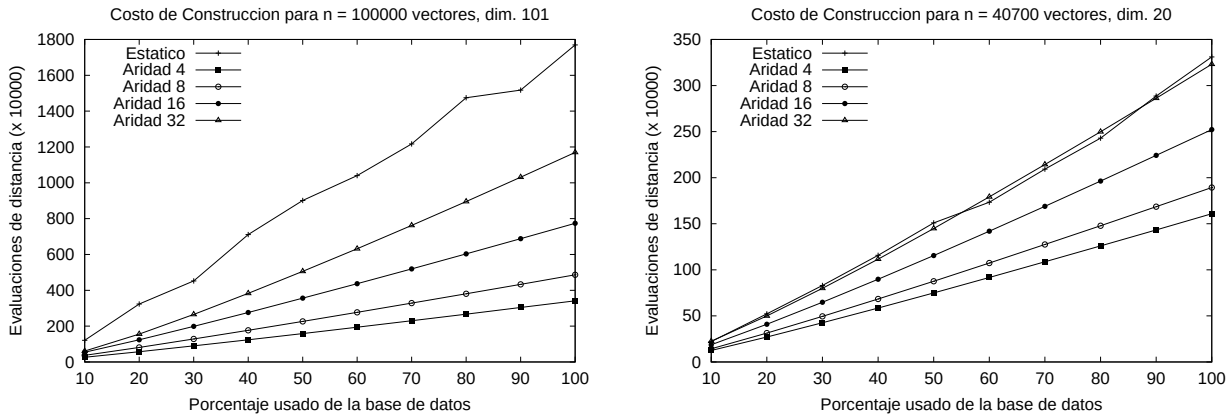


Figura A.3: Comparación entre costos de construcción incremental y estática para espacios vectoriales con distribución de Gauss en dimensión 101 (derecha) y para vectores de características de imágenes de la NASA en dimensión 20 (izquierda).

En el espacio vectorial con distribución de Gauss vemos que nuevamente aparece la misma situación, se obtienen costos de construcción menores a medida que bajamos la aridad máxima; y en todos los casos, salvo en dimensión 5 y con aridad 32, se supera a la versión original.

A la derecha de la Figura A.3 se muestran los costos de construcción para el espacio de vectores de 100000 con distribución de Gauss de dimensión 101 y considerando 200 clusters. Este espacio es de muy baja dimensión intrínseca (el histograma es menos concentrado que el de un espacio con distribución uniforme). En este espacio la versión estática no se adapta razonablemente a la dimensión porque alcanza una aridad máxima de 79; por lo tanto, la versión dinámica aún para aridad 32 la supera ampliamente.

A la izquierda de la Figura A.3 se muestran los costos de construcción para el espacio de 40700 vectores de características, en dimensión 20, de imágenes de la NASA.

A.1.2. Diccionarios

Para estos espacios la función de distancia utilizada es la distancia de edición que es discreta. La Figura A.4 muestra la comparación para los diccionarios de Inglés (69069 palabras), de Español (51789 palabras), de Francés (138257 palabras) y de Italiano (116879 palabras).

Se puede observar que en todos los casos mejoramos los costos de construcción usando aridades 4, 8 y 16. Con aridad 32 son mayores los costos que los del *SA-tree* estático, salvo para el diccionario en Español.

Las aridades máximas del *SA-tree* estático son: 25 para el diccionario en Inglés, 22 para Español, 25 para Francés y 20 para Italiano.

A.1.3. Espacio de documentos

La Figura A.5 muestra el resultado de los experimentos para el espacio de documentos descripto en Sección 2.4.2.

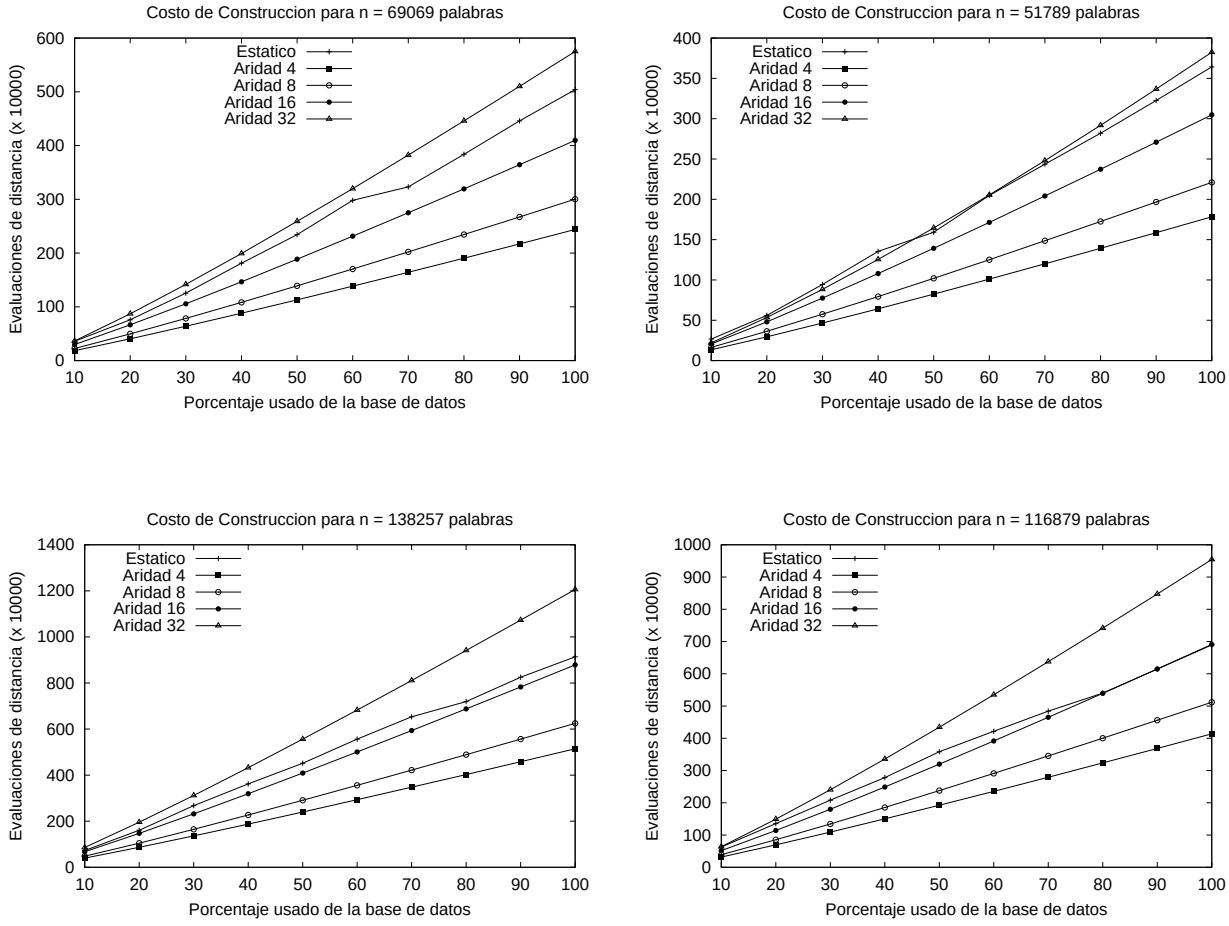


Figura A.4: Comparación entre costos de construcción incremental y estática para diccionarios de palabras en diferentes idiomas.

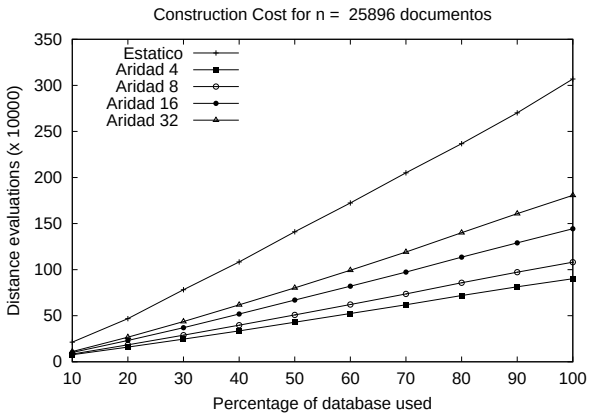


Figura A.5: Comparación entre costos de construcción incremental y estática para el espacio de documentos.

A.1.4. Análisis de los resultados

Podemos observar que en todos los experimentos las aridades más bajas logran los mejores costos de construcción y que a medida que crece la aridad crece también el costo de construcción del árbol.

Claramente la aridad se vuelve un parámetro importante a la hora de controlar los costos de construcción.

Hemos logrado, como se puede observar, un buen método de construcción incremental del *SA-tree*, porque bajamos los costos de construcción considerablemente (tanto en espacios de alta dimensionalidad como en los de baja).

A.2. Búsquedas

Realizamos numerosos experimentos para comprobar el desempeño de las búsquedas por rango y de k-vecinos más cercanos de nuestra versión dinámica del *SA-tree*, comparadas contra la versión estática.

En los experimentos de búsqueda, ya sea por rango o de vecinos más cercanos construimos el árbol con el 90 % de los elementos y luego consultamos con el 10 % restante (elegidos aleatoriamente); salvo para el caso del espacio de documentos, en el que creamos la estructura con 25860 documentos y luego consultamos con los 100 elementos restantes (elegidos aleatoriamente).

Las consultas por rango se hacen con radios que permiten recuperar el 0.01 %, el 0.1 % y el 1 % de la base de datos para los espacios que utilizan funciones de distancia continuas y radios 1, 2, 3 y 4 para cuando se utilizan funciones de distancia discretas.

Para las consultas por los k-vecinos más cercanos tomamos para k los valores: 1, 10 y 100. Cuando k es 1 estamos respondiendo las consultas 1-NN.

Para algunos espacios no pondremos las comparaciones para las consultas de vecinos más cercanos, porque basta con observar los resultados de la búsqueda por rango para ver si el método es bueno o no para competir contra la versión estática en ese espacio.

A.2.1. Espacios vectoriales

La Figura A.6 muestra la comparación para los espacios de vectores con distribución uniforme, considerando las dimensiones de 5, 10, 15 y 20.

Claramente en el espacio de dimensión 5 (baja dimensión intrínseca) mejoramos significativamente no sólo los costos de construcción sino también los de búsqueda, aún para aridad 32. En dimensión 10 con cualquiera de las aridades obtenemos mejor desempeño en las búsquedas. Por otro lado en dimensiones 15 y 20 (dimensiones altas) con aridades grandes (16 y 32 para dimensión 15, y 32 para dimensión 20) superamos o igualamos los costos de búsqueda de la versión estática.

La Figura A.7 muestra los costos de búsqueda de los k-vecinos más cercanos para los espacios con distribución uniforme en dimensión 15 y 20.

En la Figura A.8 se muestran los costos de búsqueda por rango y en la Figura A.9 los costos de las búsquedas de los k-vecinos más cercanos, para los espacios de vectores generados con distribución de Gauss, en dimensiones 5, 10, 15 y 20.

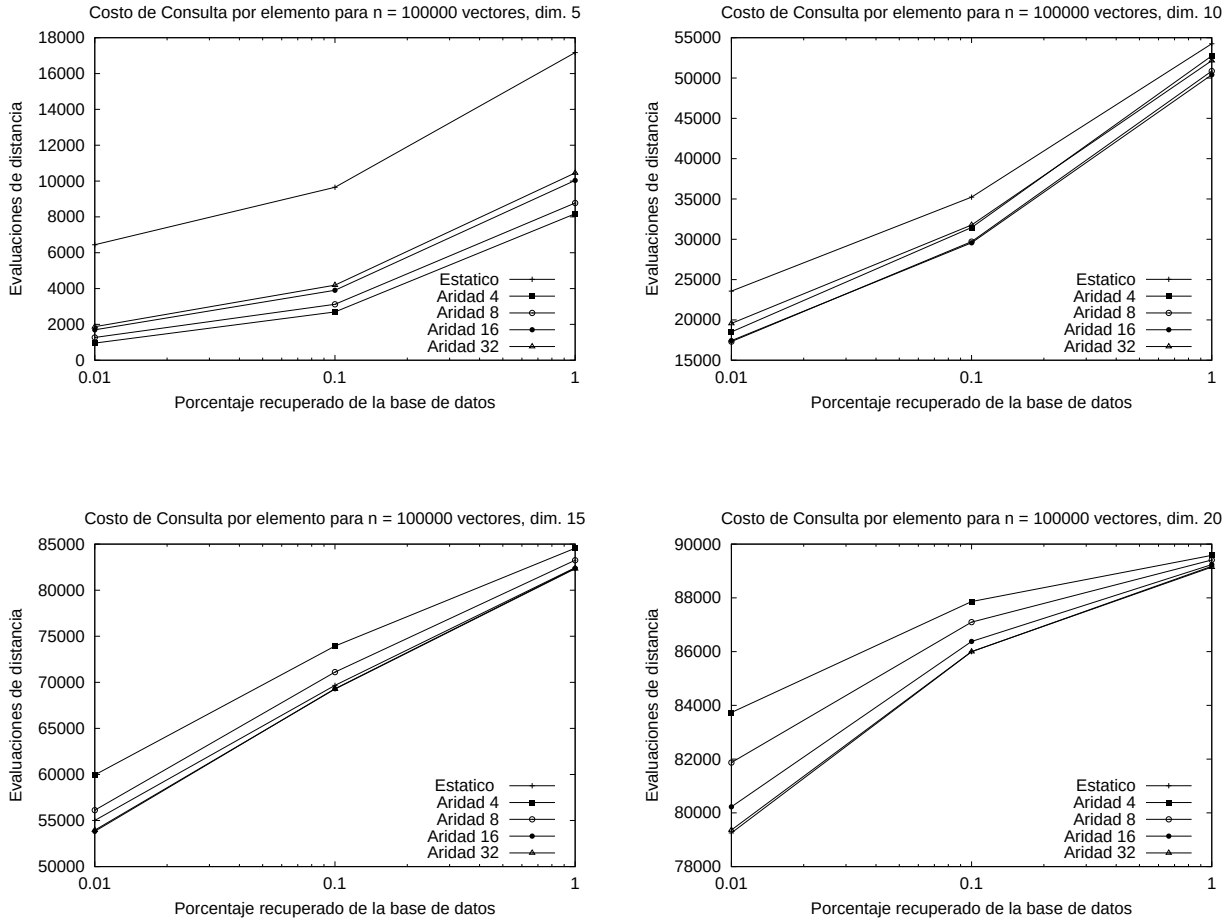


Figura A.6: Comparación de costos de búsqueda por rango en espacios vectoriales con distribución uniforme.

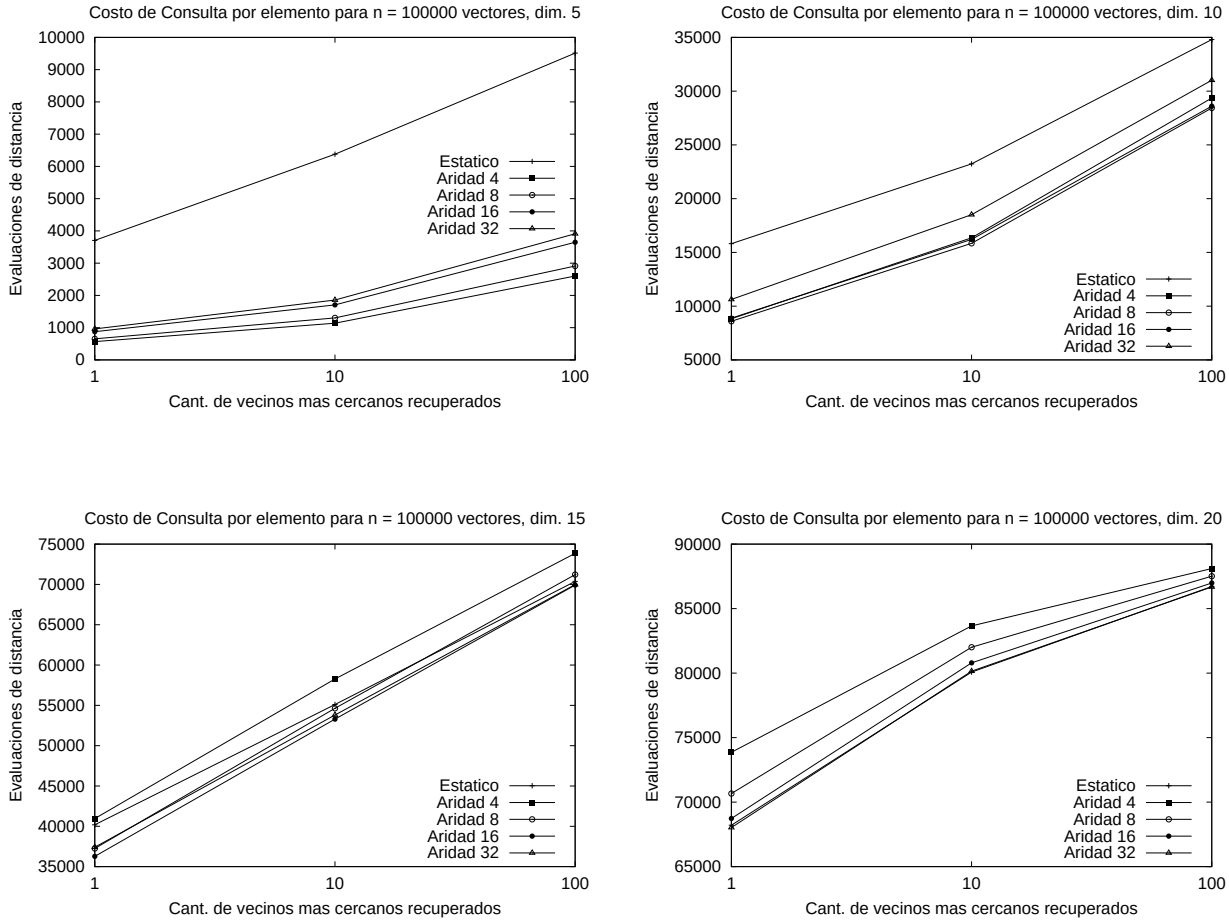


Figura A.7: Comparación de costos de búsqueda de k-vecinos más cercanos en espacios vectoriales con distribución uniforme.

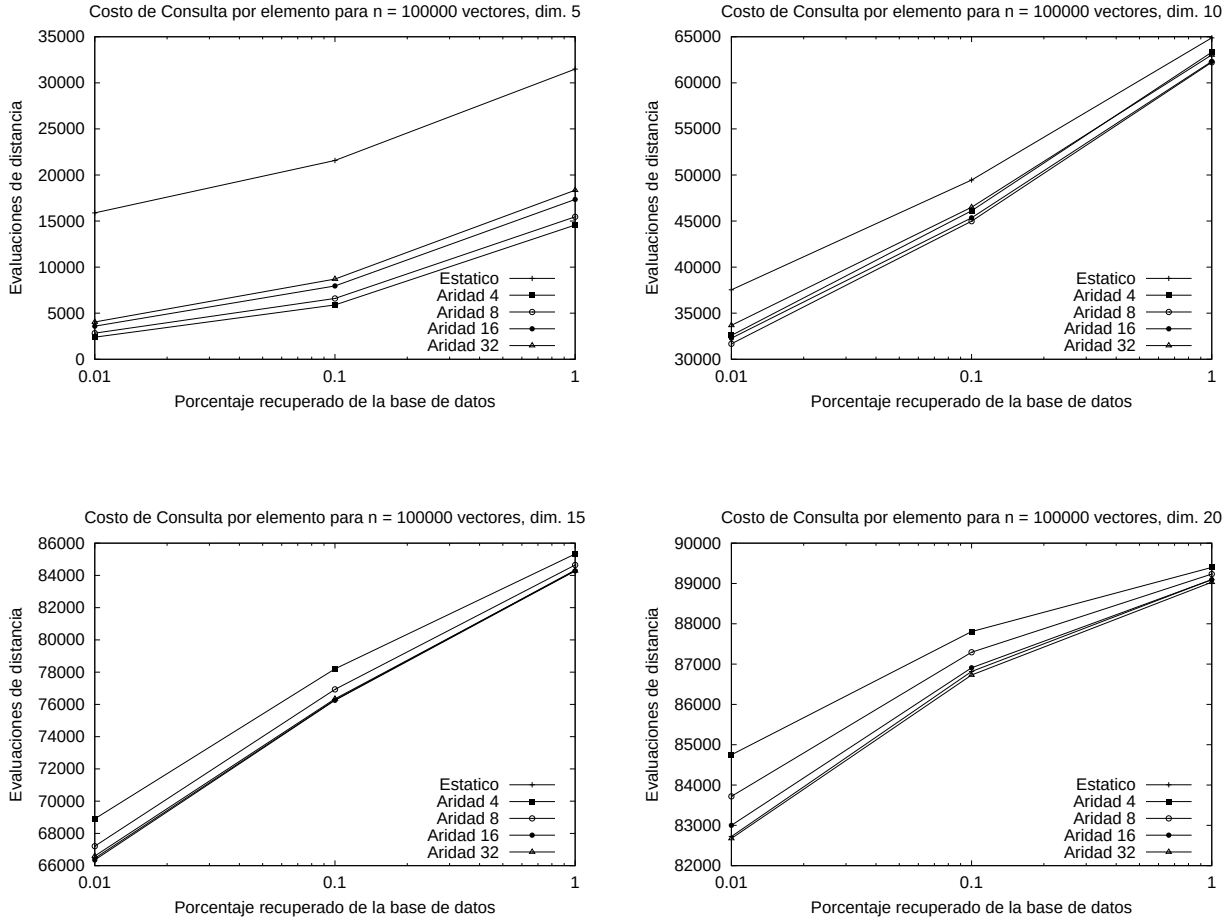


Figura A.8: Comparación de costos de búsqueda por rango en espacios vectoriales con distribución de Gauss.

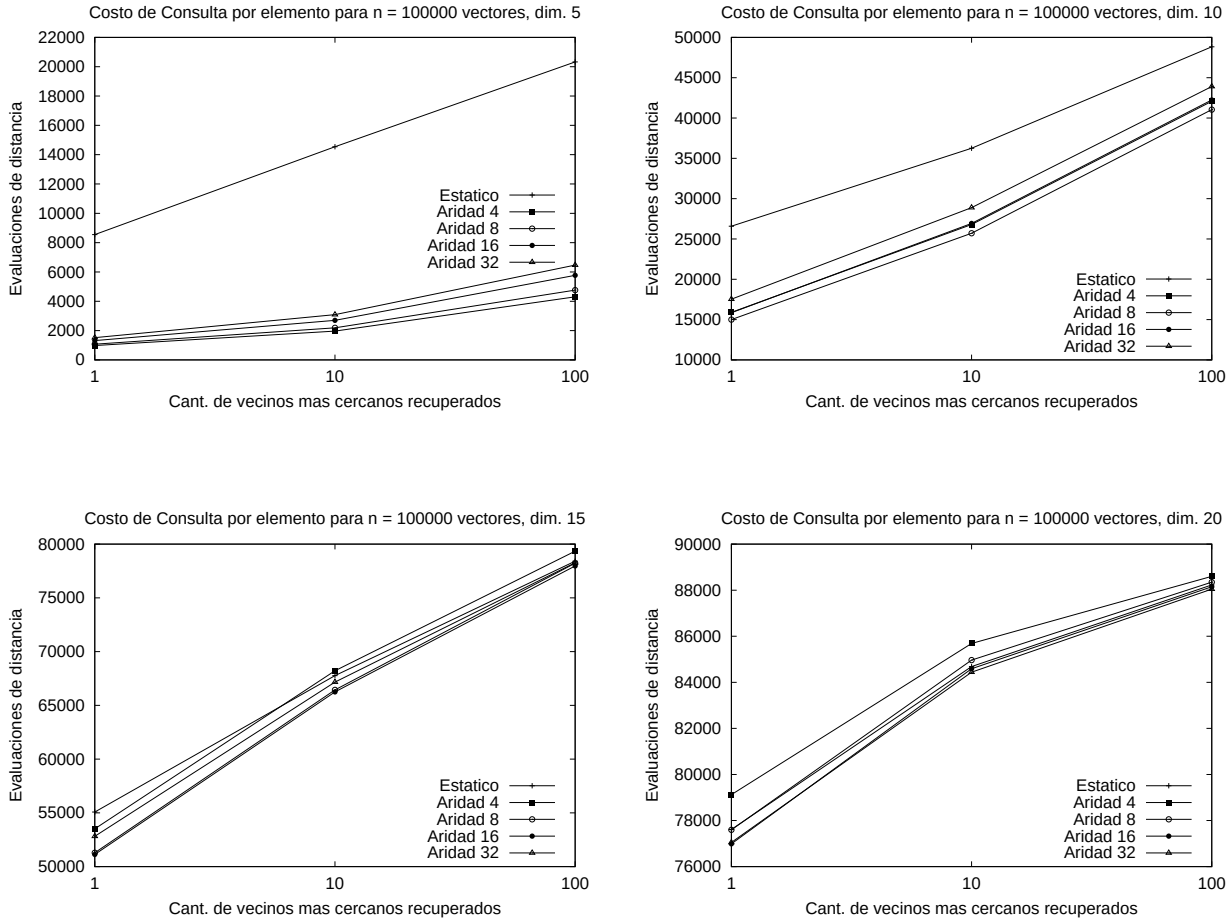


Figura A.9: Comparación de costos de búsqueda de los k-vecinos más cercanos en espacios vectoriales con distribución de Gauss.

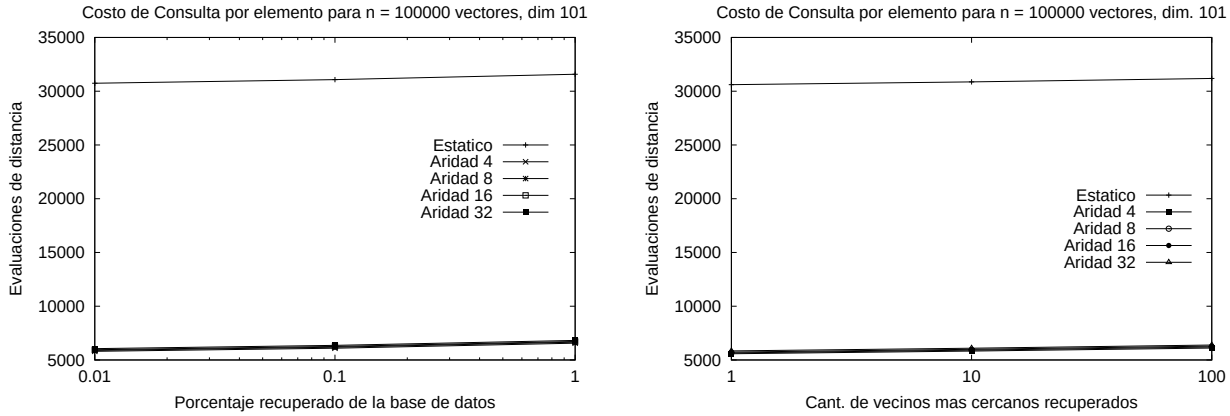


Figura A.10: Comparación de costos de búsquedas por rango (izquierda) y de k-vecinos más cercanos (derecha), en espacios vectoriales con distribución de Gauss en dimensión 101 y agrupados en 200 clusters.

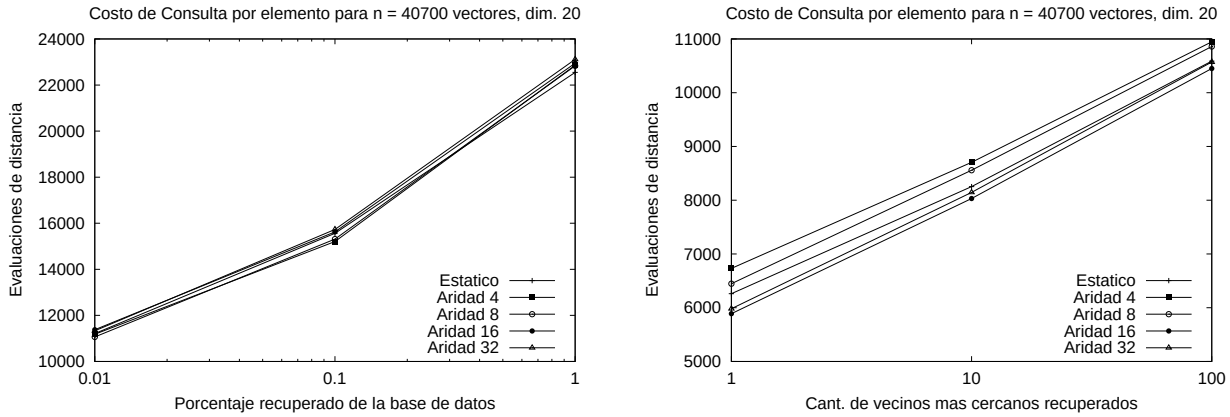


Figura A.11: Comparación de costos de búsquedas por rango (izquierda) y de k-vecinos más cercanos (derecha),en el espacio de vectores de características de imágenes de la NASA en dimensión 20.

En la Figura A.2.1 se muestra la comparación de los dos tipos de consulta para el espacio vectorial con distribución de Gauss, en dimensión 101 y constando de 200 clusters. Como se puede observar en este espacio la mejora del *SA-tree* dinámico es considerable, las búsquedas en la versión estática cuestan aproximadamente 5 veces más. Esta diferencia hace más evidente que al elegir una aridad pequeña mejoramos mucho las búsquedas en espacios de baja dimensión. Por lo tanto, en espacios de baja dimensión convienen aridades pequeñas porque cuestan menos la construcción y las búsquedas.

La Figura A.11 muestra los costos de ambos tipos de consulta para el espacio de los vectores de características de las imágenes de la NASA (dimensión 20).

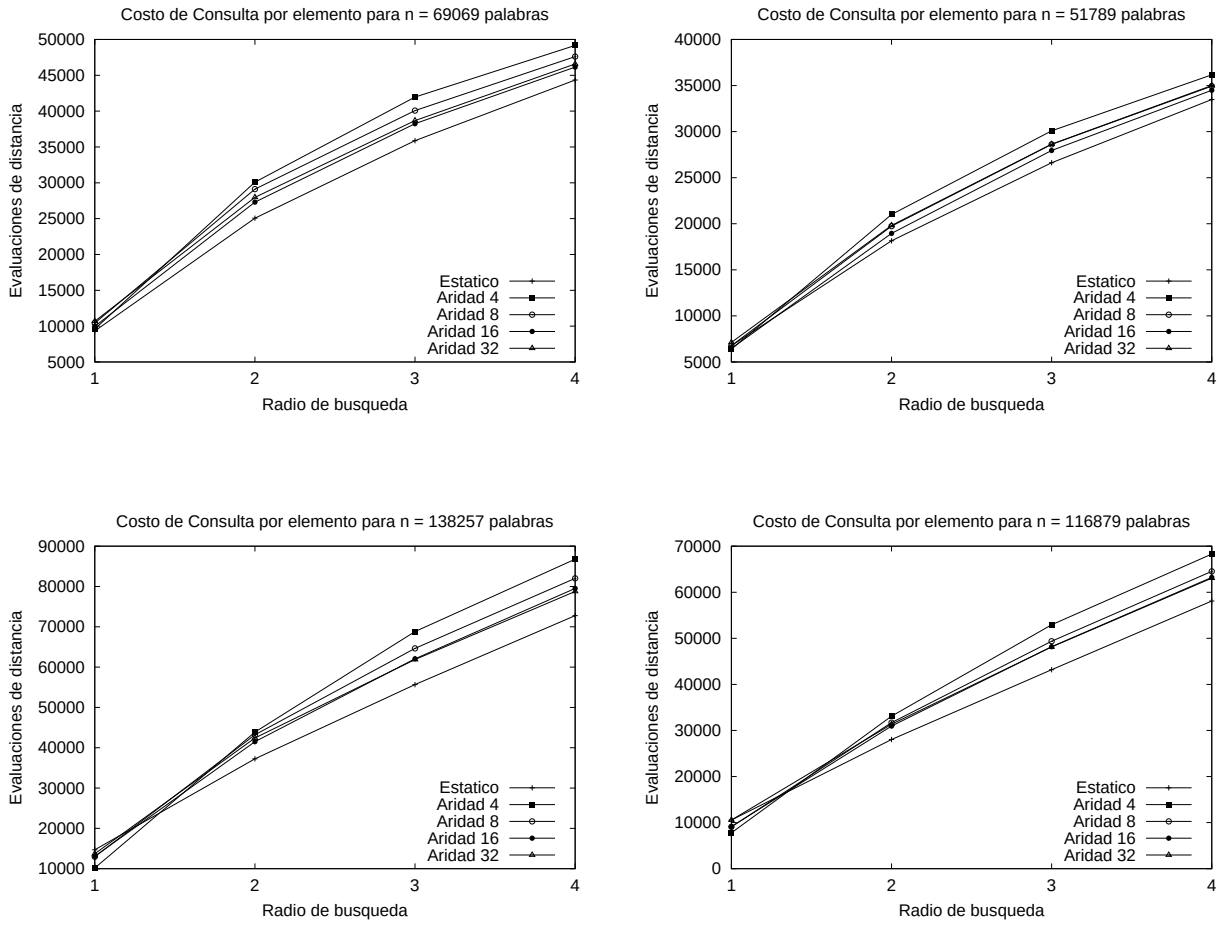


Figura A.12: Comparación de costos de búsqueda por rango en diccionarios de palabras en distintos idiomas.

A.2.2. Diccionarios

La Figura A.2.2 muestra los resultados para las búsquedas por rango, mientras que la Figura A.2.2 muestra los de las búsquedas de vecinos mas cercanos, para los distintos diccionarios considerados.

A.2.3. Espacio de Documentos

Para evaluar los experimentos de búsqueda para este espacio, disminuimos el número de consultas a realizar debido a los tiempos reales de procesamiento en los equipos disponibles. Por lo tanto, aquí se realizaron 100 búsquedas de documentos aleatoriamente elegidos, sobre un árbol construido con el resto de la base de datos (25796 documentos).

La Figura A.14 muestra a la derecha los costos de búsquedas por rango para este espacio.

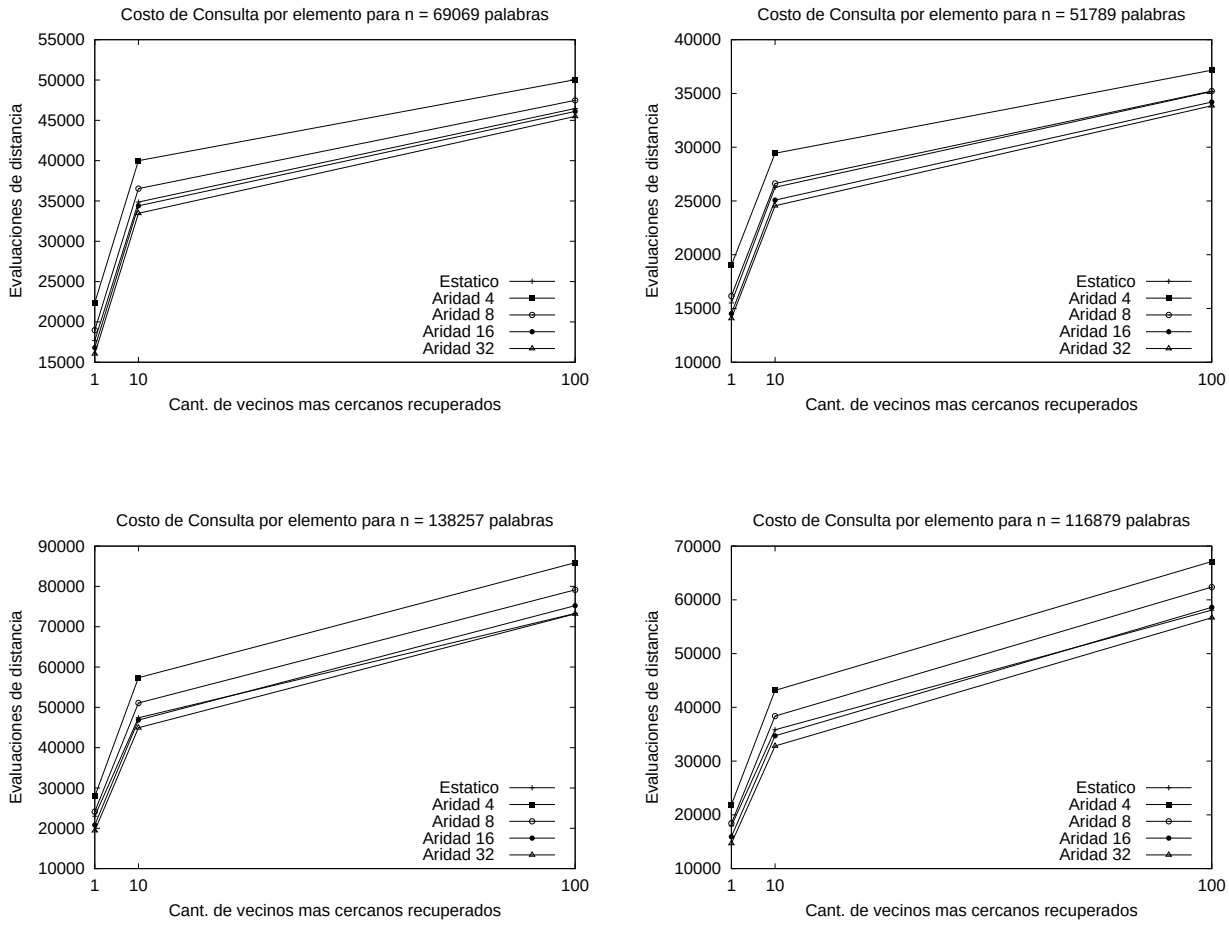


Figura A.13: Comparación de costos de búsqueda de k-vecinos más cercanos en diccionarios en distintos idiomas.

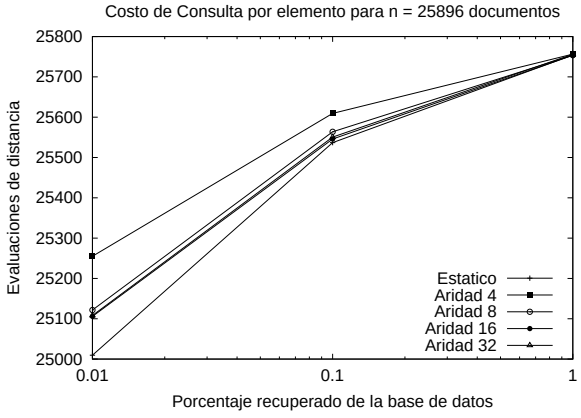


Figura A.14: Comparación de costos de búsqueda por rango para el espacio de documentos y usando distancia coseno.

A.2.4. Análisis de los resultados

Como se puede observar en todos los casos hemos conseguido mantener eficiente la búsqueda y en algunos casos superar al *SA-tree* original, eligiendo adecuadamente la aridad máxima del árbol.

Ya hemos analizado que a menor aridad menor costo de construcción, pero la mejor aridad para las búsquedas depende principalmente de la dimensión del espacio. Espacios de alta dimensionalidad obtienen mejores resultados con aridades altas y los de baja dimensionalidad con aridades pequeñas.

Es importante observar que aunque no elijamos la mejor aridad para un *SA-tree* dinámico el precio a pagar posteriormente en las búsquedas no es demasiado alto (ver por ejemplo la Tabla 4.1 y la Tabla 4.2).

A.3. Eliminaciones

Mostraremos aquí los experimentos realizados para algunos de los espacios métricos considerados, para analizar cómo se comportan los distintos métodos de eliminación.

Para los primeros experimentos construimos el árbol con el 90 % de los elementos de la base de datos, reservamos el 10 % restante (elegido aleatoriamente) para las búsquedas y luego eliminamos un 10 % de los elementos.

La Figura A.15 muestra los costos de eliminación para el espacio de vectores de dimensión 15 y para el diccionario de palabras en Inglés, para aridades 16 y 32, comparando las técnicas de reinserción de subárbol, de a elemento, nodos ficticios y las técnicas combinadas con los α que se indican en cada caso.

La Figura A.16 muestra los costos de las búsquedas para el espacio de vectores de dimensión 15 y para el diccionario de palabras en Inglés, para aridades 16 y 32, comparando las técnicas de reinserción de subárbol, de a elemento, nodos ficticios y las técnicas combinadas con los α que se indican en cada caso.

La Figura A.17 muestra la comparación de los costos de eliminación entre el método de reinserción de a elemento (con la optimización) y el de reconstrucción de subárbol para el espacio de vectores de dimensión 15 (izquierda) y la comparación del método de reconstrucción para las distintas aridades.

La Figura A.18 muestra la comparación de los costos de eliminación entre el método de reinserción de a elementos (con la optimización) y el de reconstrucción de subárbol para el espacio de vectores de dimensión 101 y 200 clusters (izquierda) y la comparación del método de reconstrucción para las distintas aridades.

La Figura A.19 muestra los costos de las búsquedas para el espacio de vectores de dimensión 15 (izquierda) y para el de dimensión 101 y 200 clusters (derecha) comparando las técnicas de reconstrucción de subárbol, reinserción de a elemento.

La Figura A.20 ilustra la comparación de los costos de eliminación del método combinado de reconstrucción de subárbol y nodos ficticios para los distintos valores de α en el espacio de vectores de dimensión 15 (izquierda) y el de dimensión 101 y 200 clusters (derecha).

La Figura A.21 muestra los costos de las búsquedas para el espacio de vectores de dimensión 15 (izquierda) y para el de dimensión 101 y 200 clusters (derecha) comparando el método combinado con reconstrucción de subárbol para los distintos valores de α.

Para los siguientes experimentos construimos el árbol con distintos porcentajes de la base de datos (50 %, 60 %,. . .,90 %) y eliminamos la cantidad de elementos necesarios para dejar el árbol con el 50 % de

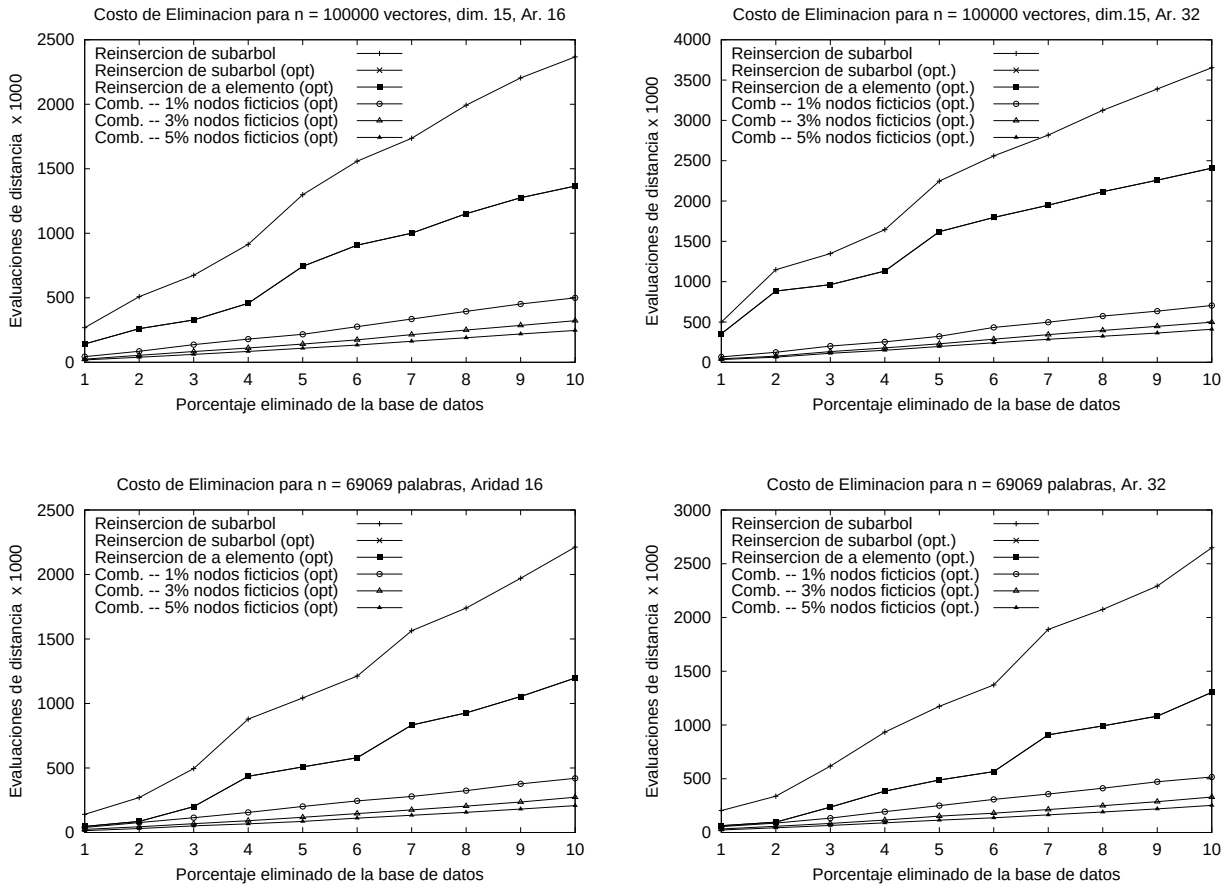


Figura A.15: Comparación de costos de eliminación de diferentes métodos para el espacio de vectores de dimensión 15 (arriba) y para el diccionario de Inglés (abajo), con aridades 16 y 32.

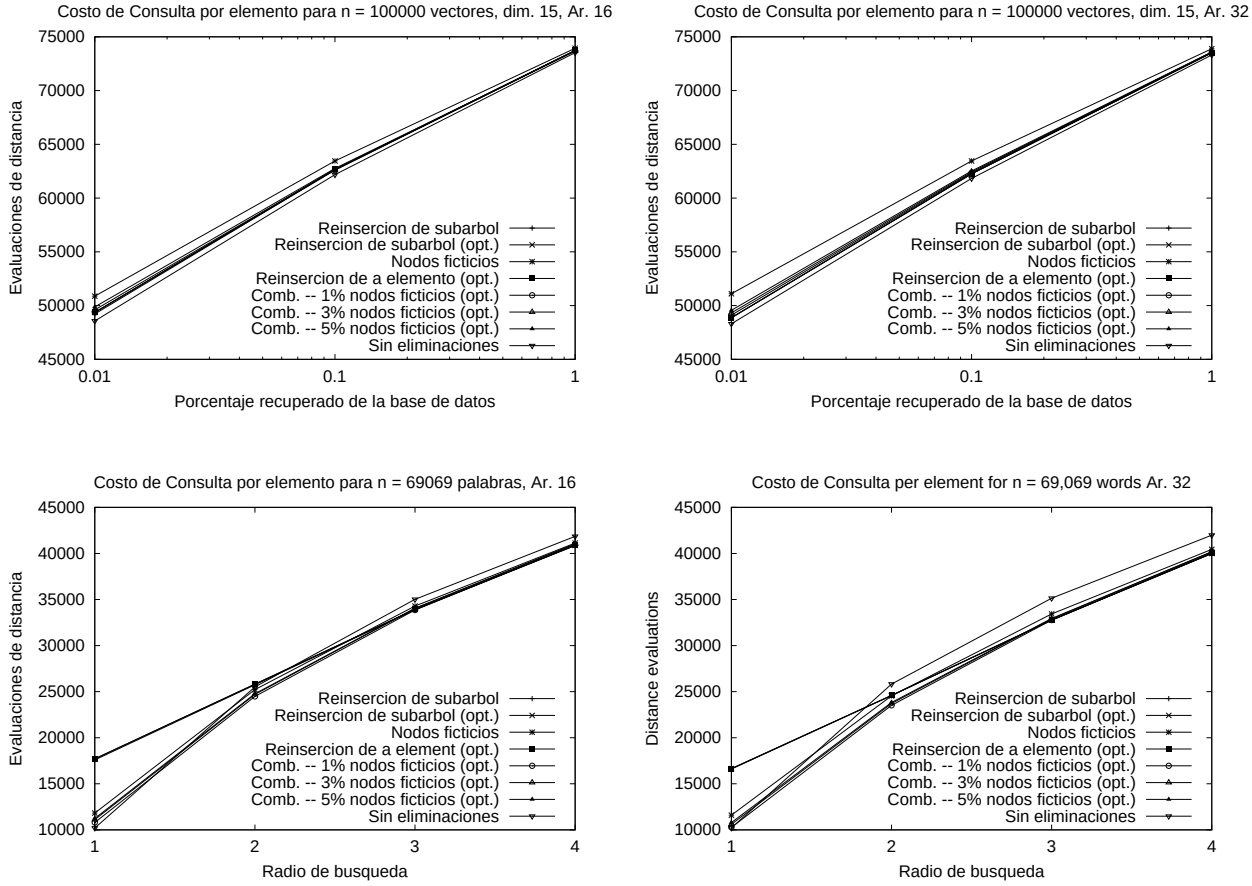


Figura A.16: Comparación de costos de búsqueda de diferentes métodos para el espacio de vectores de dimensión 15 (arriba) y para el diccionario de Inglés (abajo), con aridades 16 y 32.

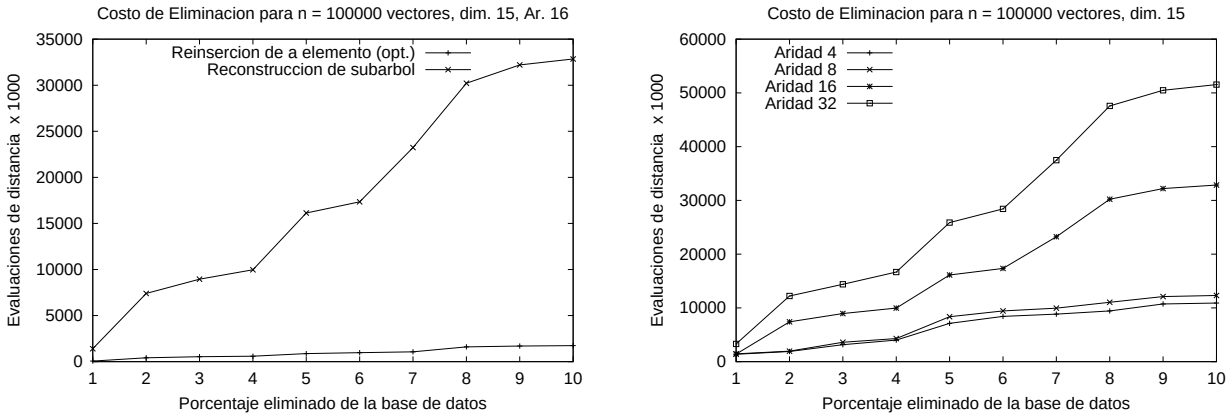
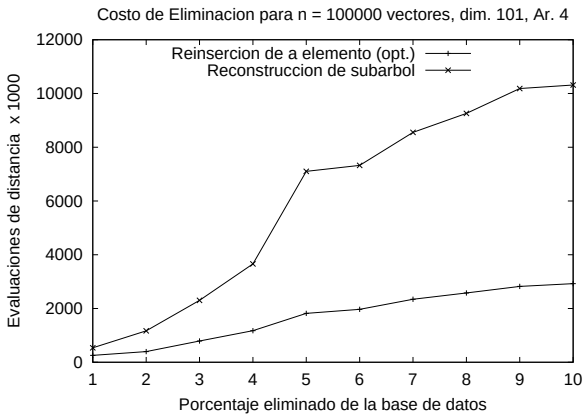


Figura A.17: Comparación de los costos de eliminación para el espacio de vectores de dimensión 15 entre el método de reinserción de a elemento y reconstrucción de subárbol (izquierda) y comparación para diferentes aridades del método de reconstrucción de subárbol (derecha).



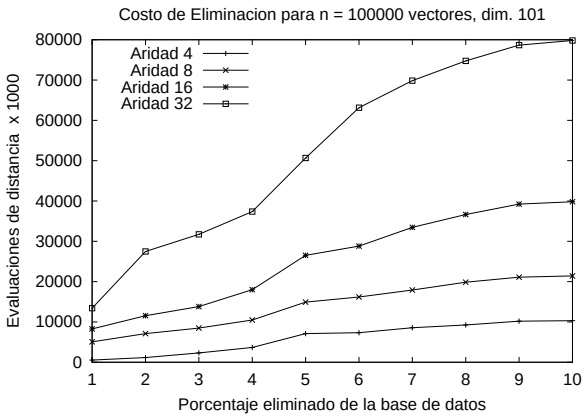
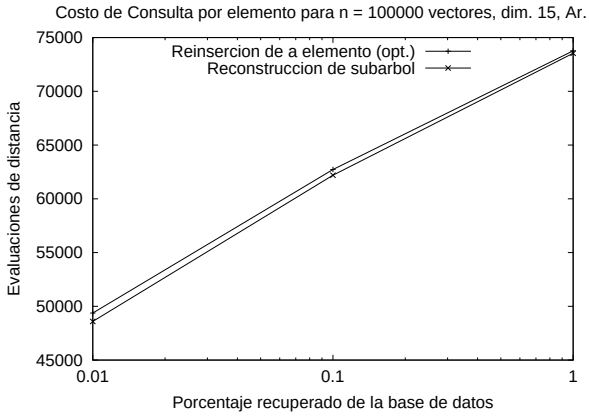


Figura A.18: Comparación de los costos de eliminación para el espacio de vectores de dimensión 101 entre el método de reinserción de a elemento y reconstrucción de subárbol (izquierda) y comparación para diferentes aridades del método de reconstrucción de subárbol (derecha).



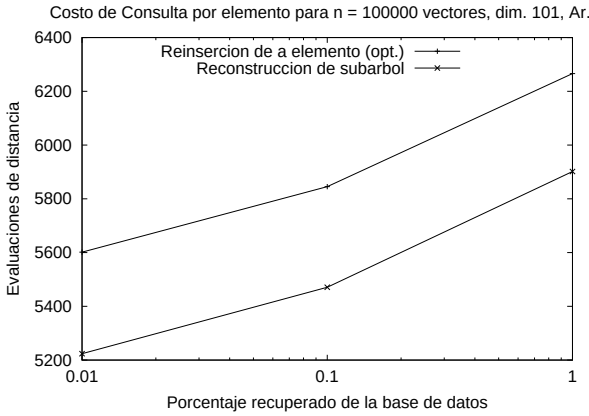
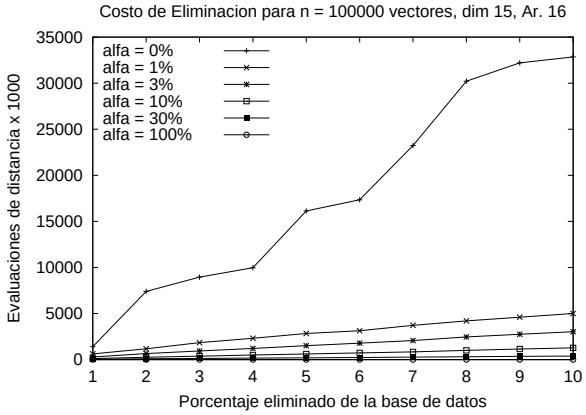


Figura A.19: Comparación de costos de búsqueda de reinserción de a elemento y reconstrucción de subárbol espacio de vectores de dimensión 15 (izquierda) y de dimensión 101 (derecha).



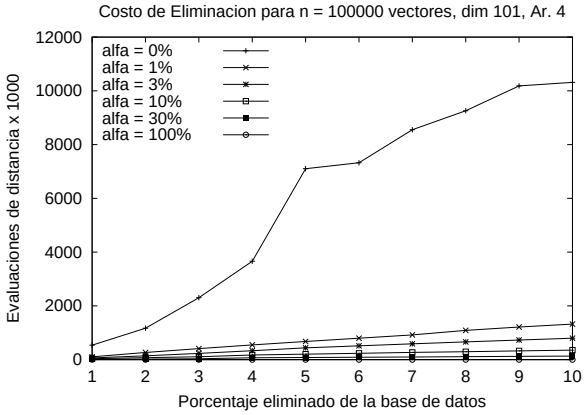
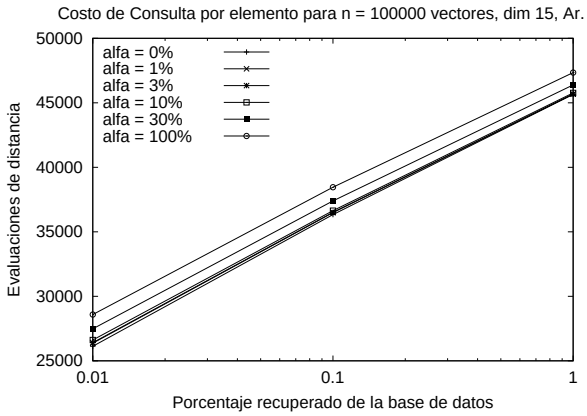


Figura A.20: Comparación de costos de eliminación para el método combinado con reconstrucción de subárbol usando diferentes valores de α, para el espacio de vectores de dimensión 15 (izquierda) y de dimensión 101 (derecha).



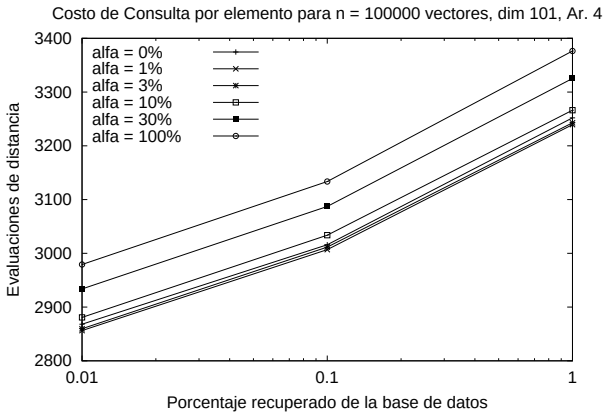


Figura A.21: Comparación de costos de búsqueda del método combinado con reconstrucción de subárbol para los diferentes valores de α, en el espacio de vectores de dimensión 15 (izquierda) y de dimensión 101 (derecha).

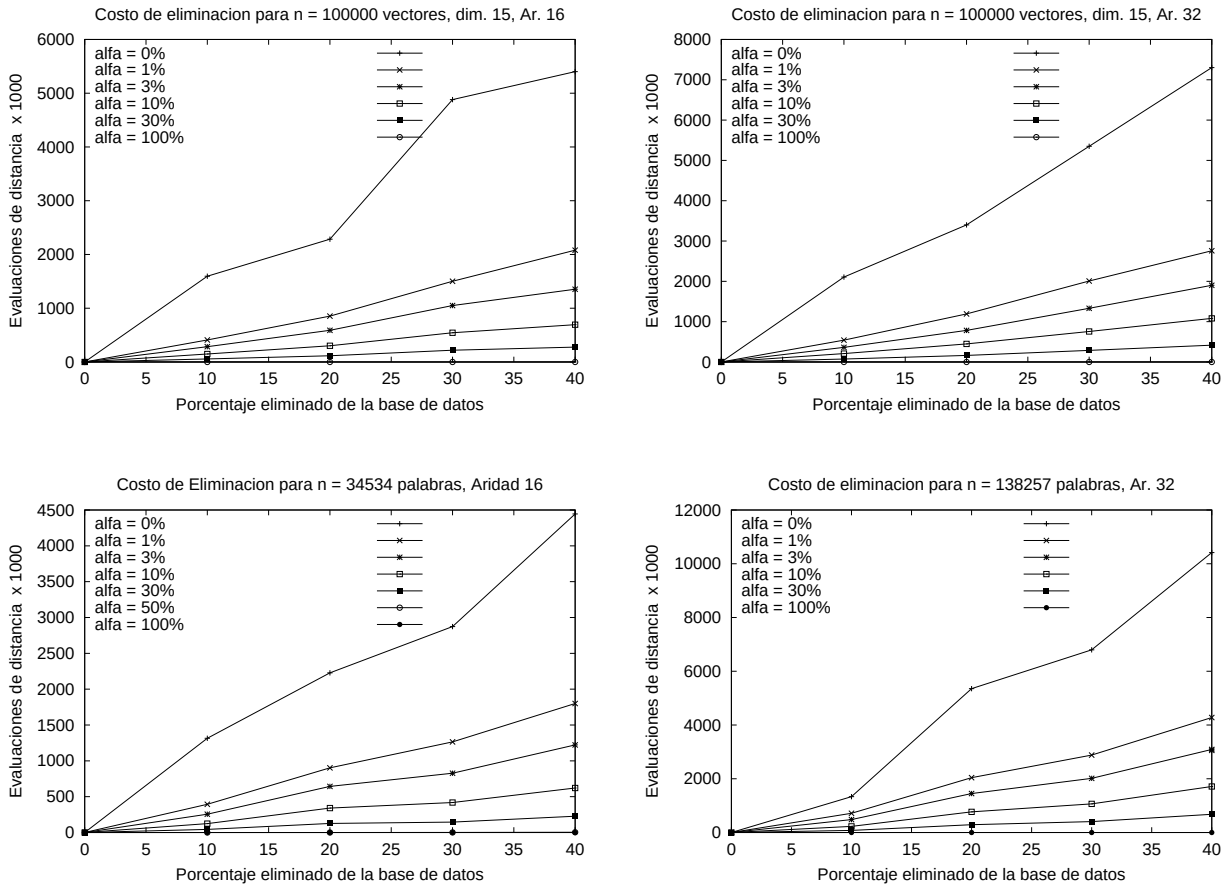


Figura A.22: Comparación de costos de eliminación de diferentes métodos para el espacio de vectores de dimensión 15 con aridades 16 y 32 (arriba) y para los diccionarios (abajo) de Inglés (izquierda) y de Francés (derecha).

elementos de la base de datos. El 10 % de los elementos, elegidos aleatoriamente, se reservan para realizar las búsquedas que permiten evaluar el efecto acumulativo de las eliminaciones sobre la estructura. Es muy importante verificar que las eliminaciones no degraden significativamente el proceso de búsqueda.

La Figura A.22 muestra los costos de eliminación para el espacio de vectores de dimensión 15 (arriba) y para los diccionarios (abajo) de Inglés usando aridad 16 (izquierda) y de Francés usando aridad 32 (derecha), comparando las técnica combinada con reinserción de a elemento para los distintos valores de α que se indican.

La Figura A.23 muestra los costos de las búsquedas para el espacio de vectores de dimensión 15 usando aridad 16 y considerando los distintos porcentajes de elementos eliminados, para la técnica combinada con reinserción de a elementos, comparando los diferentes valores de α y la versión estática. La Figura A.24 muestra los mismos resultados pero comparando el comportamiento para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$, para los diferentes porcentajes eliminados.

La Figura A.25 muestra los costos de búsqueda luego de eliminar un 10 % y un 40 % de los elementos para los diccionarios de Inglés (arriba) y de Francés (abajo), comparando las técnica combinada con reinserción de a elemento para los diferentes valores de α y la versión estática. La Figura A.26 muestra

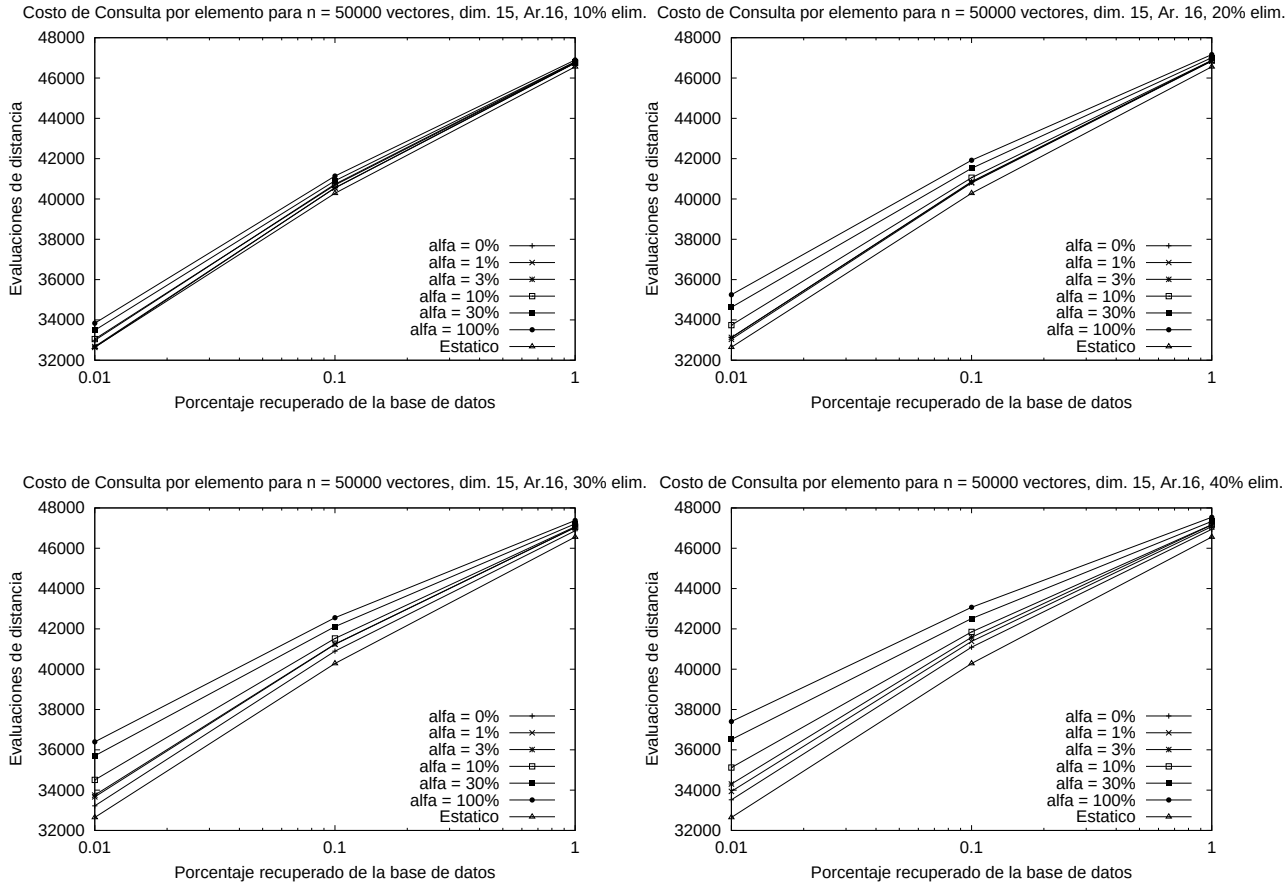


Figura A.23: Comparación de costos de búsqueda para los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el espacio de vectores de dimensión 15 con aridad 16.

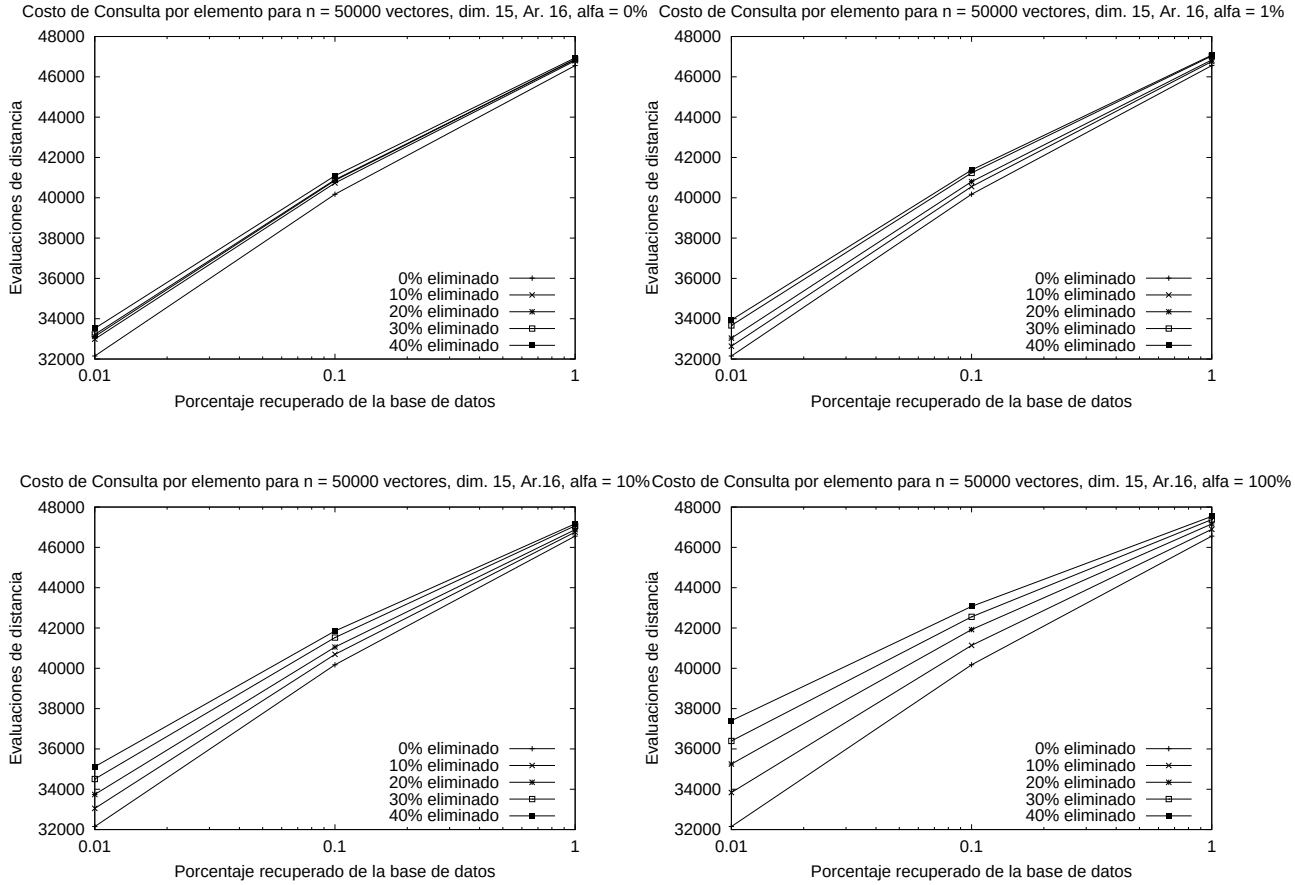


Figura A.24: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el espacio de vectores de dimensión 15 con aridad 16.

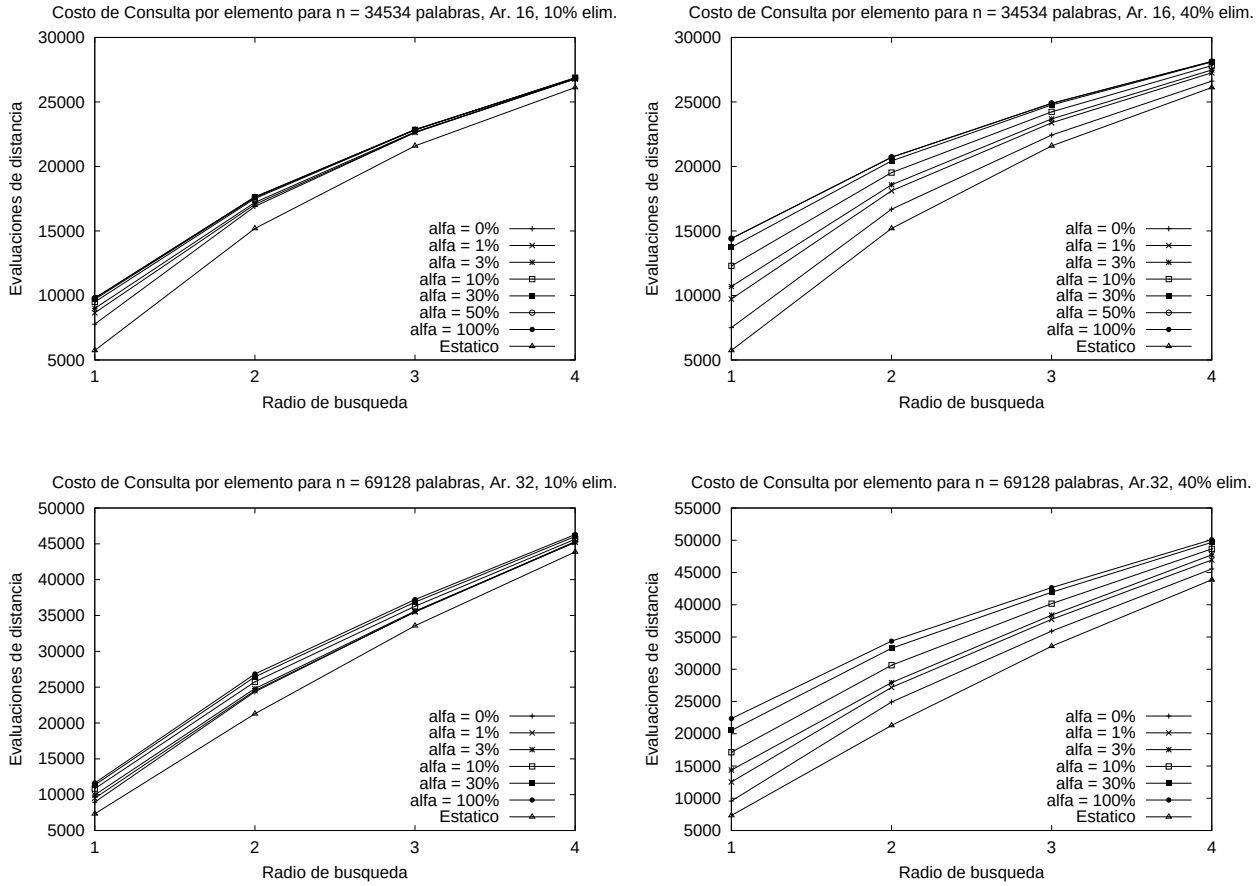


Figura A.25: Comparación de costos de búsqueda para 10 % y 40 % de elementos eliminados usando el método combinado de reinserción de a elemento para el diccionario de Inglés (arriba) y de Francés (abajo).

los mismos resultados pero comparando el comportamiento para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$, para los diferentes porcentajes eliminados para el diccionario de Inglés y la Figura A.27 para el de Francés.

Consideramos ahora dos espacios de baja dimensión, el de vectores de dimensión 5 con distribución uniforme y el de dimensión 101 y 200 clusters con distribución de Gauss. La Figura A.28 muestra los resultados obtenidos para 10 % y 40 % de elementos eliminados comparando los distintos valores de α, en el método combinado de reinserción de a elemento, en el espacio de vectores de dimensión 5 y la Figura A.29 muestra el mismo análisis en el espacio de vectores de dimensión 101.

La Figura A.30 compara los costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$, al eliminar los distintos porcentajes de elementos para el espacio de vectores de dimensión 5. La Figura A.31 realiza la misma comparación para el espacio de vectores de dimensión 101.

Como se puede observar, en este tipo de espacio el comportamiento para los distintos valores de α no es el que se espera y además se puede observar que las búsquedas se degradan a medida que aumenta el número de elementos eliminados.

La Figura A.32 muestra la comparación de los costos de eliminación entre el método de reinserción de a elemento (con la optimización) y el de reconstrucción de subárbol para el espacio de vectores de dimensión

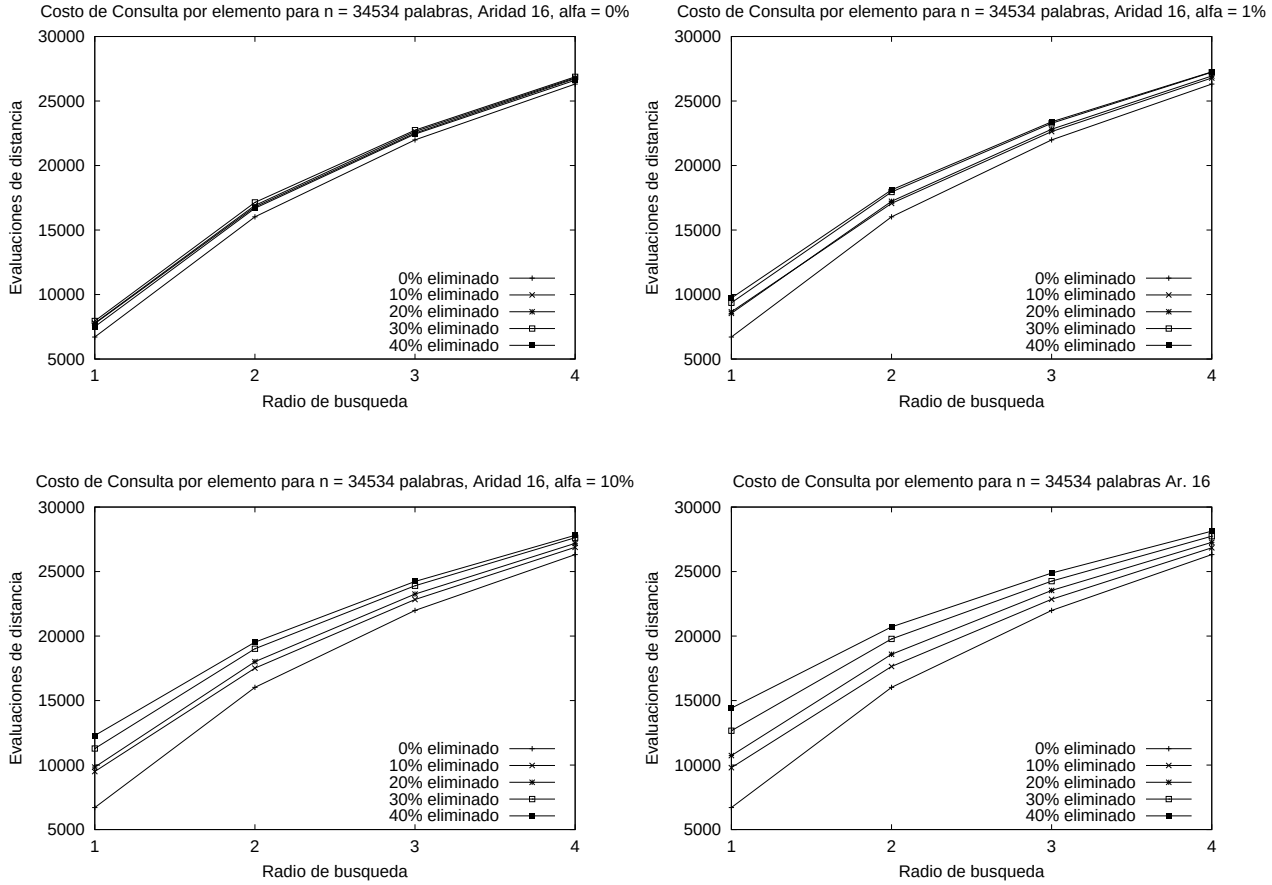


Figura A.26: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el diccionario de Inglés con aridad 16.

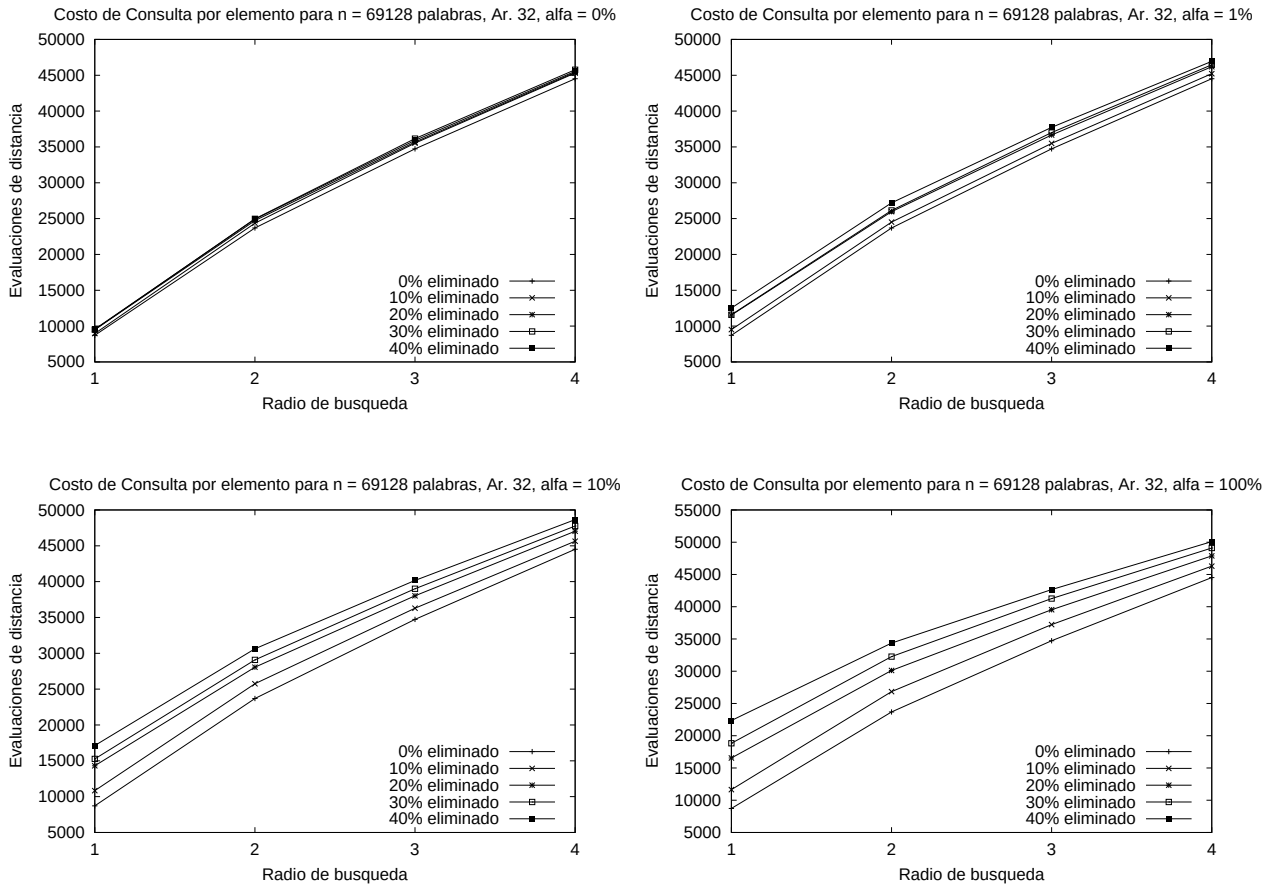


Figura A.27: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el diccionario de Francés con aridad 32.

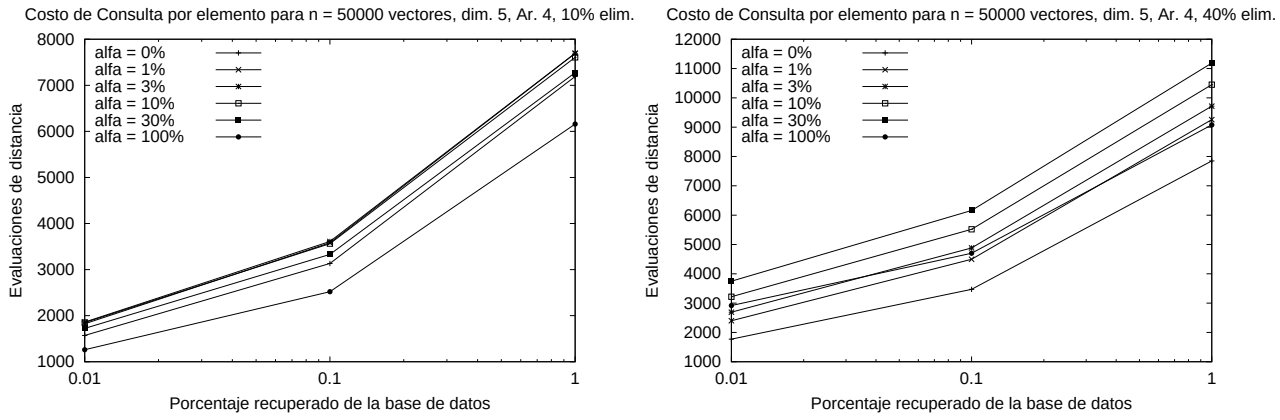


Figura A.28: Comparación de costos de búsqueda para los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el espacio de vectores de dimensión 5 con aridad 4.

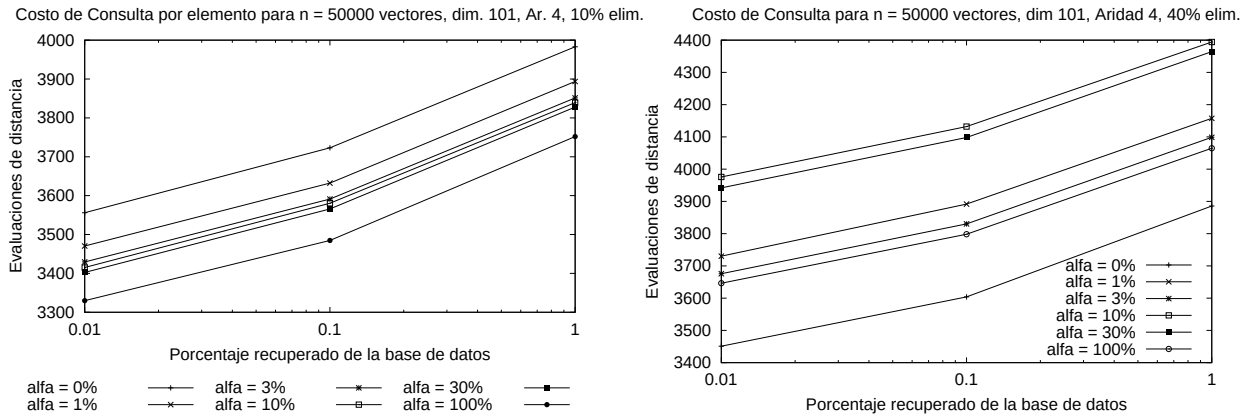


Figura A.29: Comparación de costos de búsqueda para los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el espacio de vectores de dimensión 101 con aridad 4.

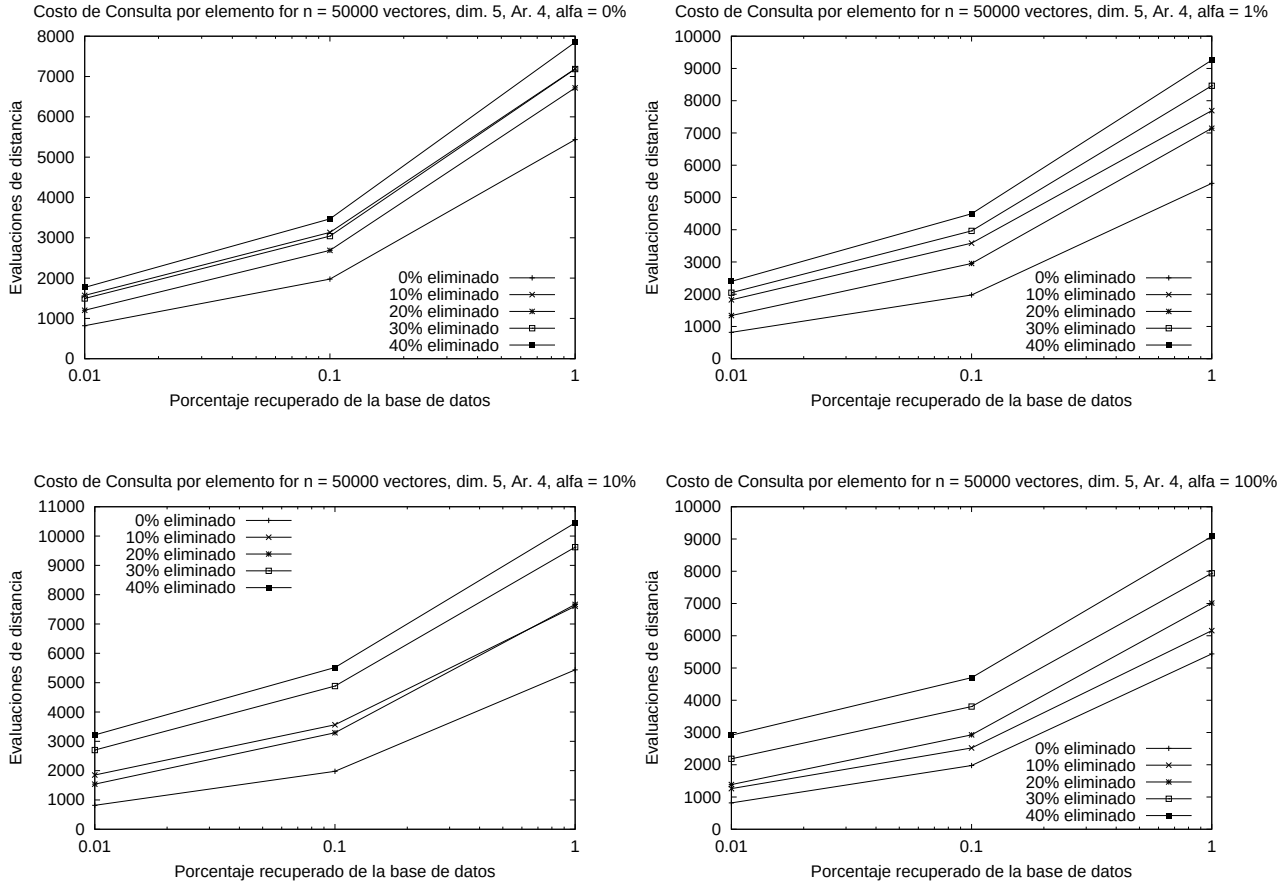


Figura A.30: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el espacio de vectores de dimensión 5 con aridad 4.

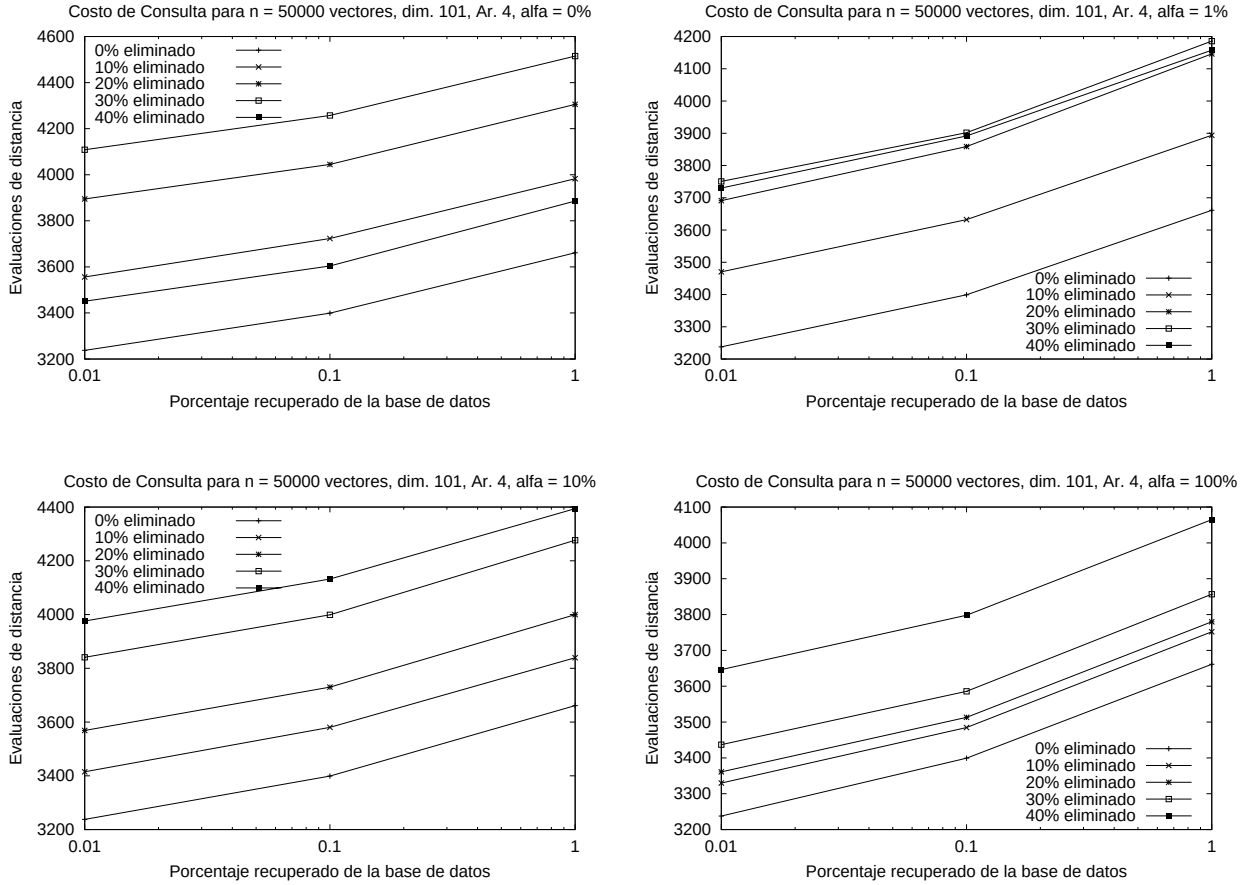


Figura A.31: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reinserción de a elemento para el espacio de vectores de dimensión 101 con aridad 4.

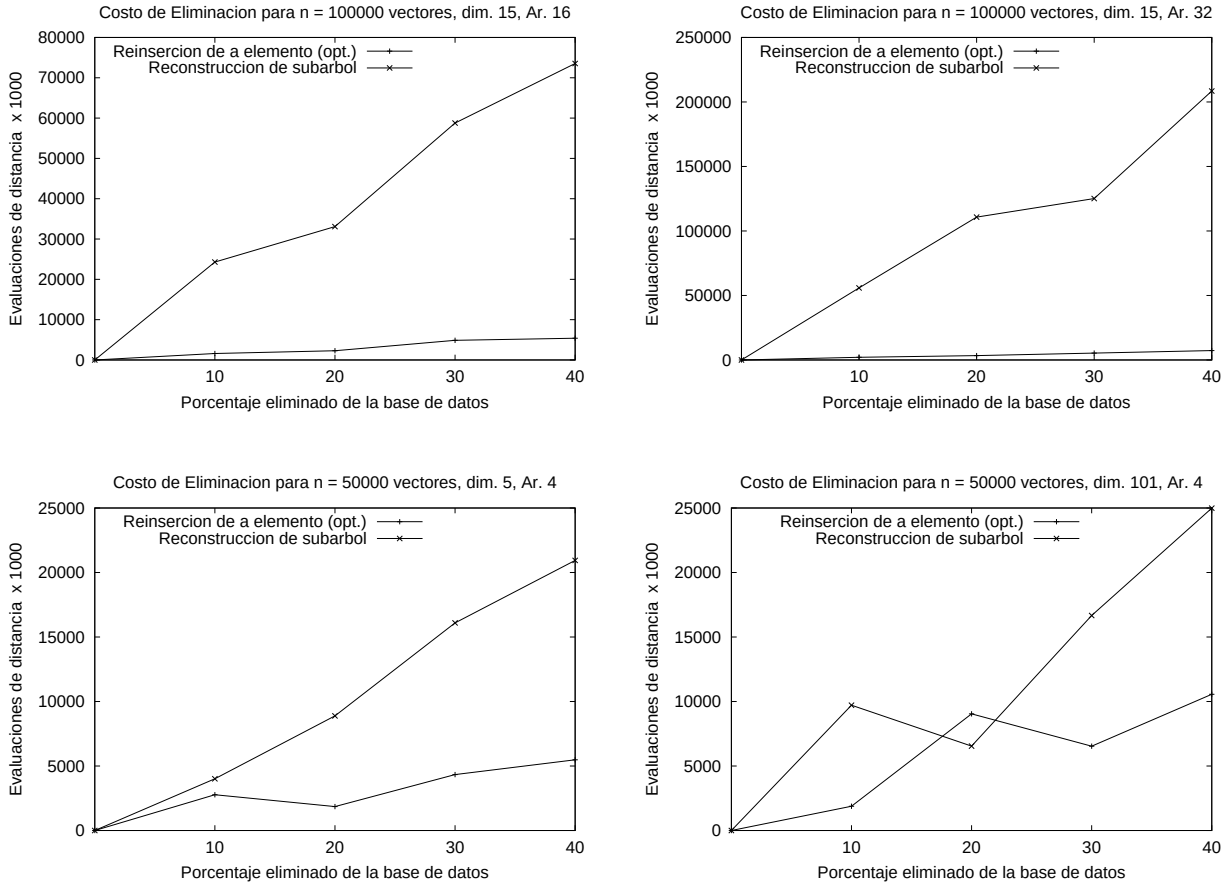


Figura A.32: Comparación de los costos de eliminación entre el método de reinserción de a elemento y reconstrucción de subárbol para el espacio de vectores de dimensión 15 (arriba) y para los espacios vectoriales de baja dimensión (abajo), para el de dimensión 5 (izquierda) y para el de dimensión 101 (derecha).

15 (arriba) usando aridades 16 y 32, y para los espacios de vectores de baja dimensión (abajo), para el de dimensión 5 (izquierda) y para el de dimensión 101 (derecha).

La Figura A.33 muestra los costos de las búsquedas para el espacio de vectores de dimensión 15 usando aridad 16 y considerando los distintos porcentajes de elementos eliminados, para la técnica combinada con reconstrucción de subárbol, comparando los diferentes valores de α. La Figura A.34 muestra los mismos resultados pero comparando el comportamiento para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$, para los diferentes porcentajes eliminados.

La Figura A.35 muestra los resultados obtenidos para $10\,\%$,$20\,\%$, $30\,\%$ y $40\,\%$ de elementos eliminados comparando los distintos valores de α, en el método combinado de reconstrucción de subárbol, en el espacio de vectores de dimensión 5 y la Figura A.36 muestra el mismo análisis en el espacio de vectores de dimensión 101.

La Figura A.37 compara los costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$, al eliminar los distintos porcentajes de elementos para el espacio de vectores de dimensión 5. La Figura A.38 realiza la misma comparación para el espacio de vectores de dimensión 101.

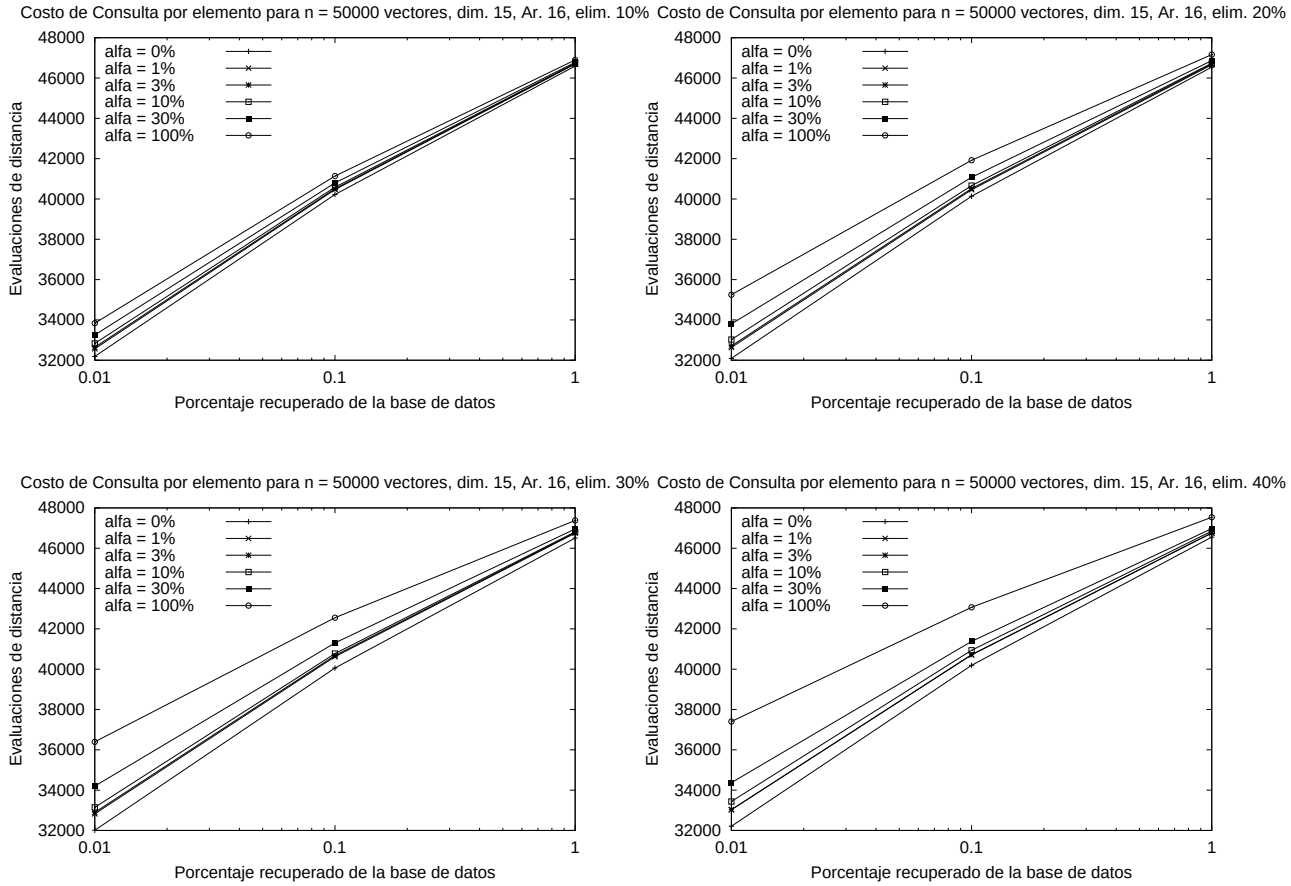


Figura A.33: Comparación de costos de búsqueda para los distintos porcentajes de elementos eliminados usando el método combinado de reconstrucción de subárbol para el espacio de vectores de dimensión 15 con aridad 16.

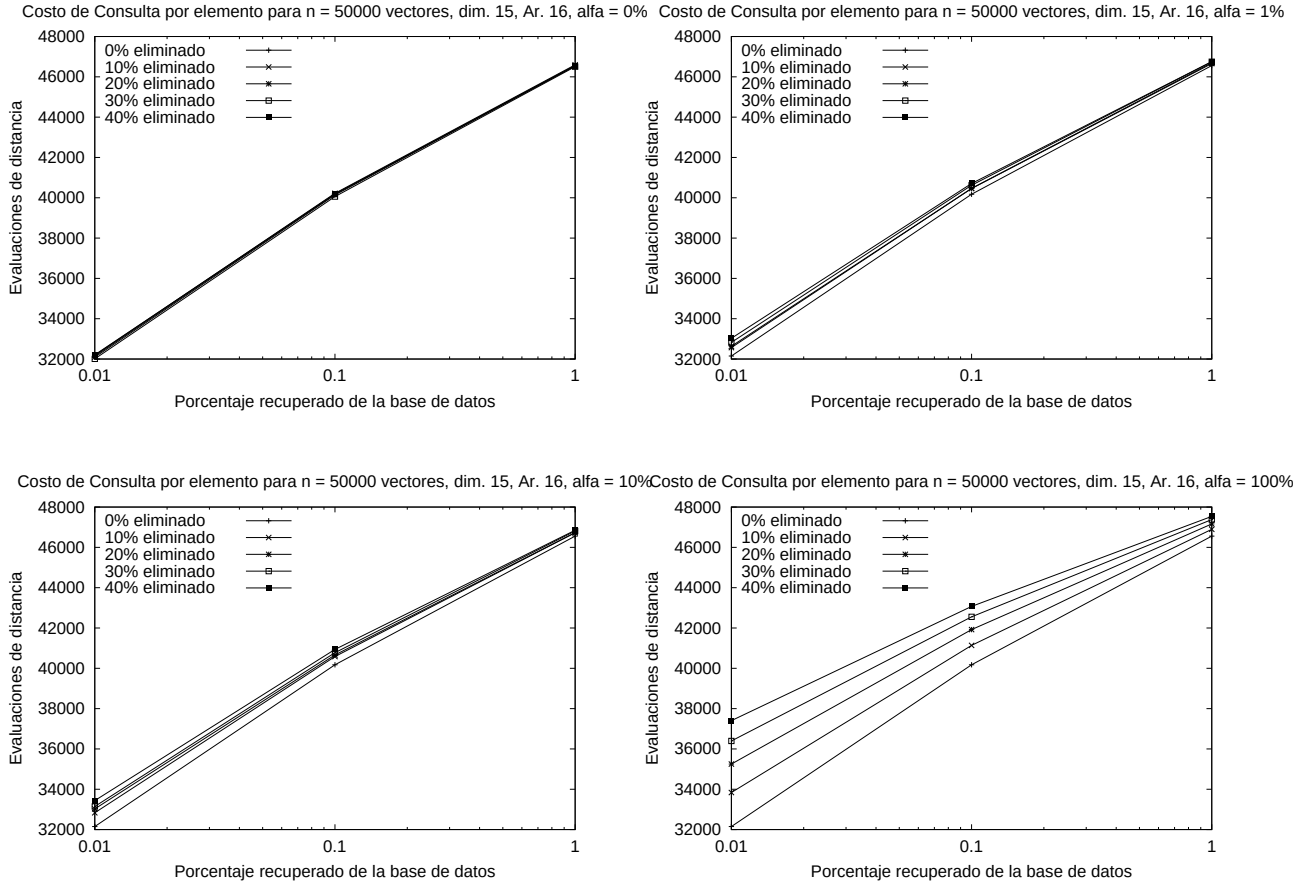


Figura A.34: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reconstrucción de subárbol para el espacio de vectores de dimensión 15 con aridad 16.

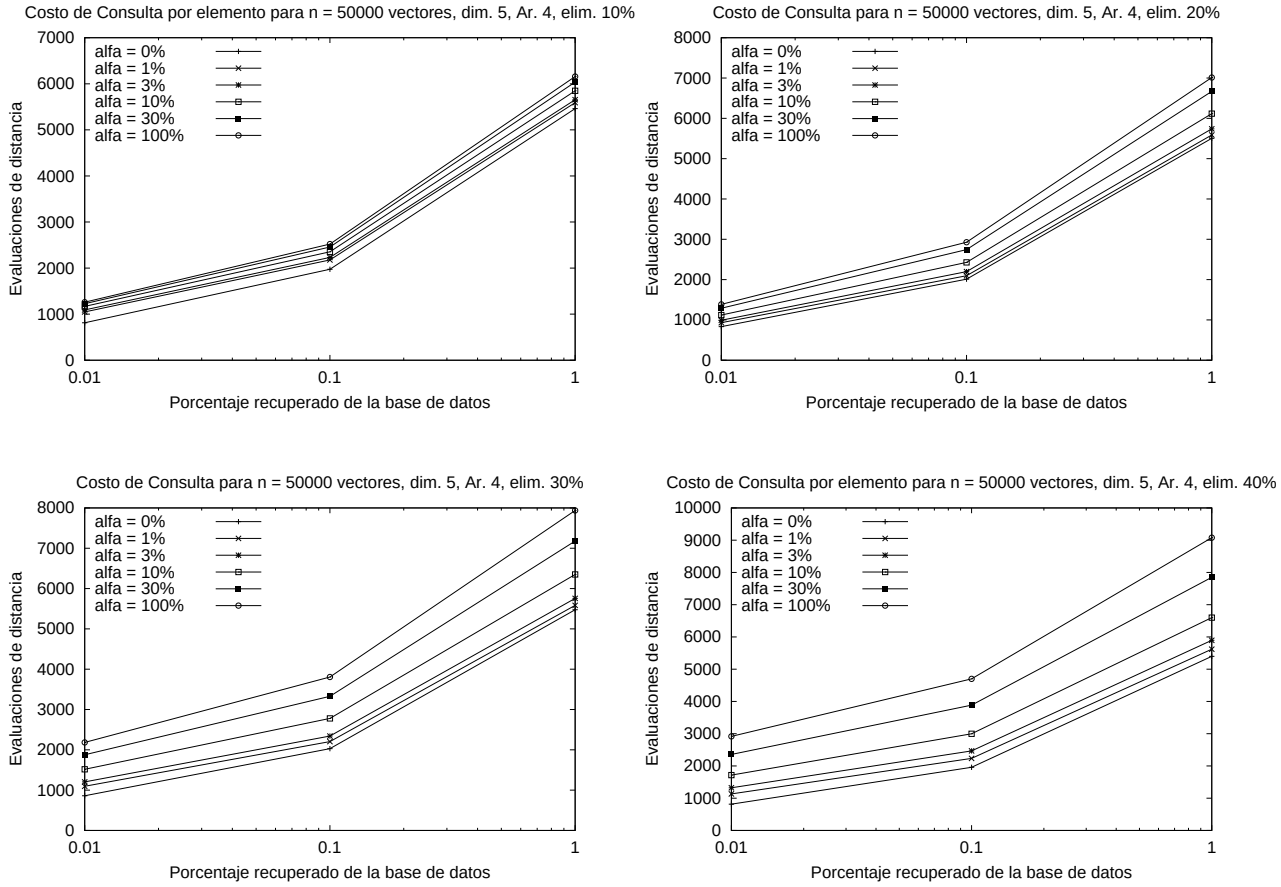


Figura A.35: Comparación de costos de búsqueda para los distintos porcentajes de elementos eliminados usando el método combinado de reconstrucción de subárbol para el espacio de vectores de dimensión 5 con aridad 4.

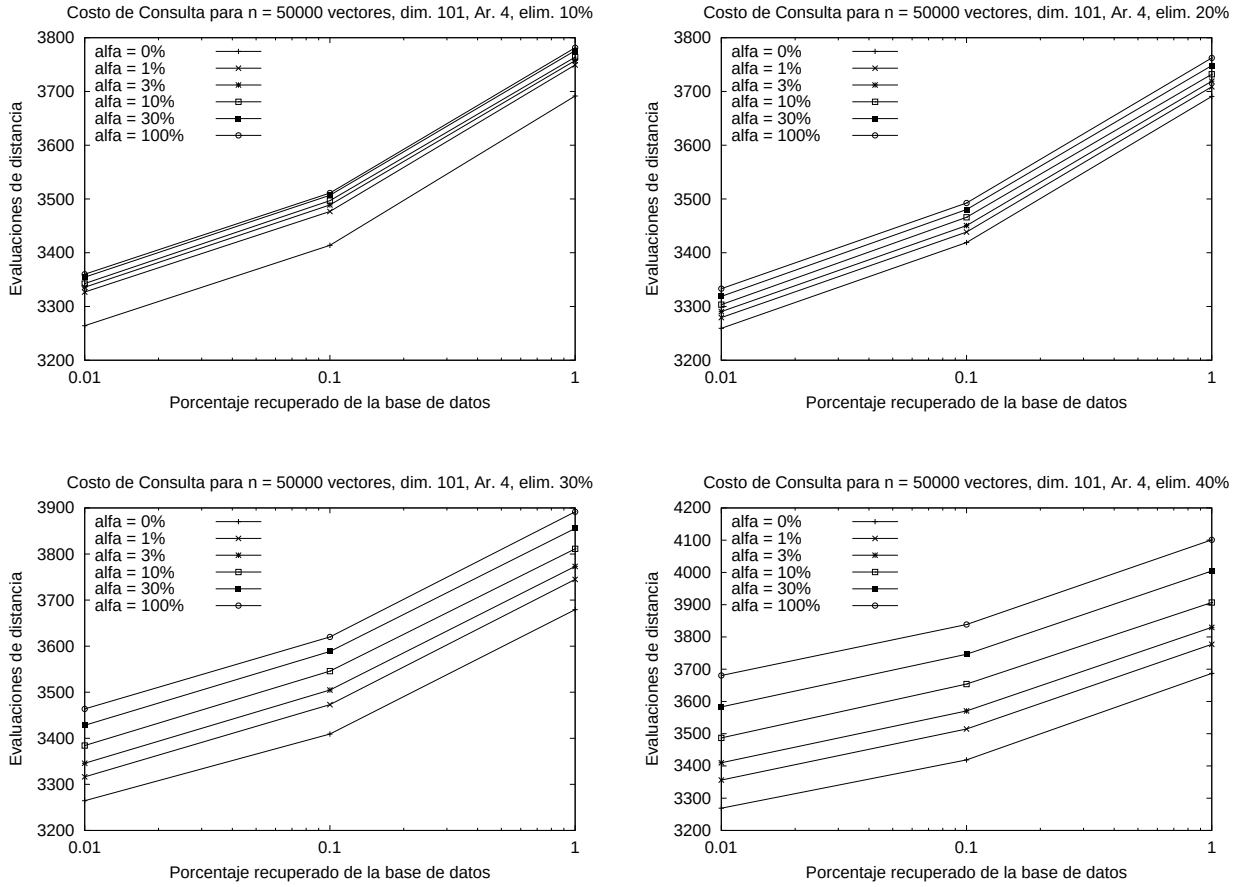


Figura A.36: Comparación de costos de búsqueda para los distintos porcentajes de elementos eliminados usando el método combinado de reconstrucción de subárbol para el espacio de vectores de dimensión 101 con aridad 4.

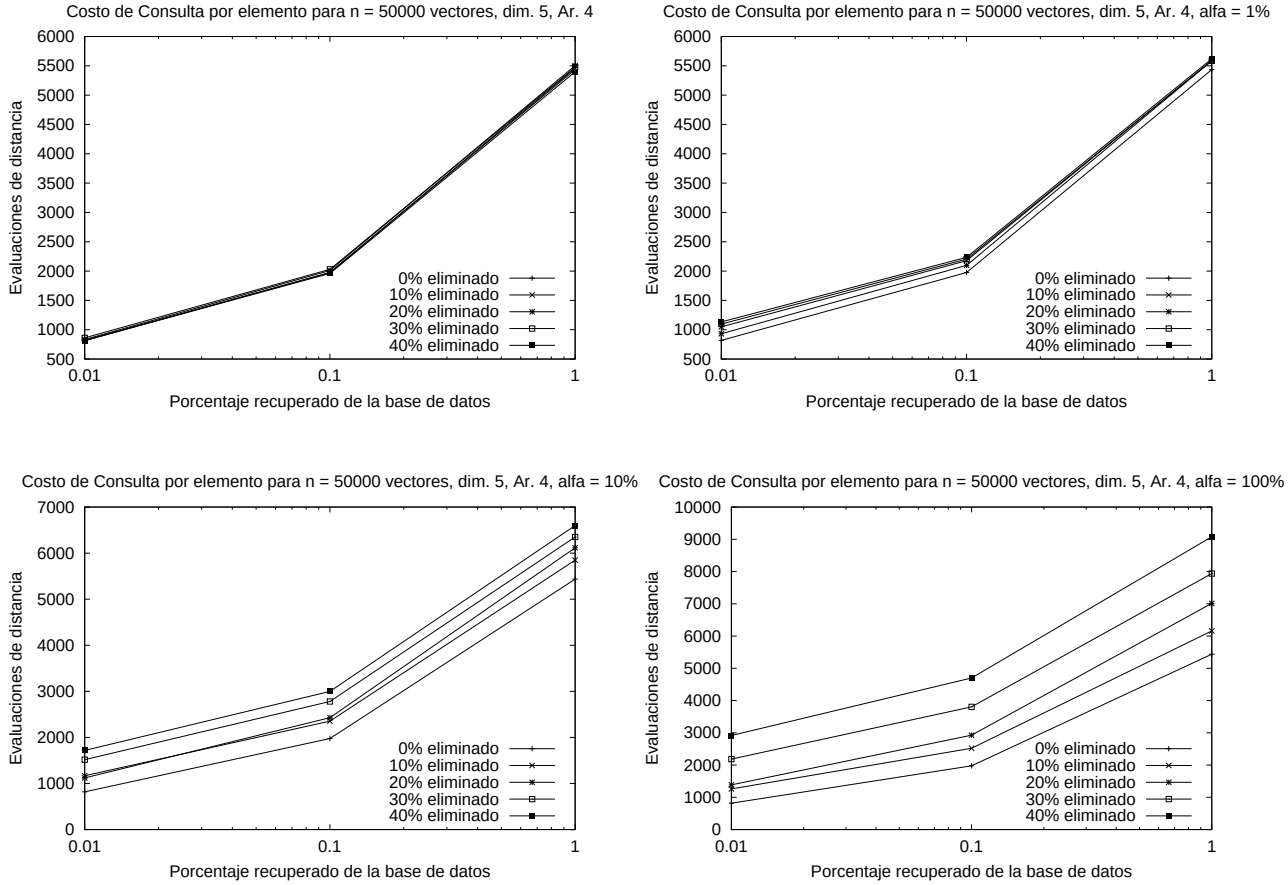


Figura A.37: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reconstrucción de subárbol para el espacio de vectores de dimensión 5 con aridad 4.

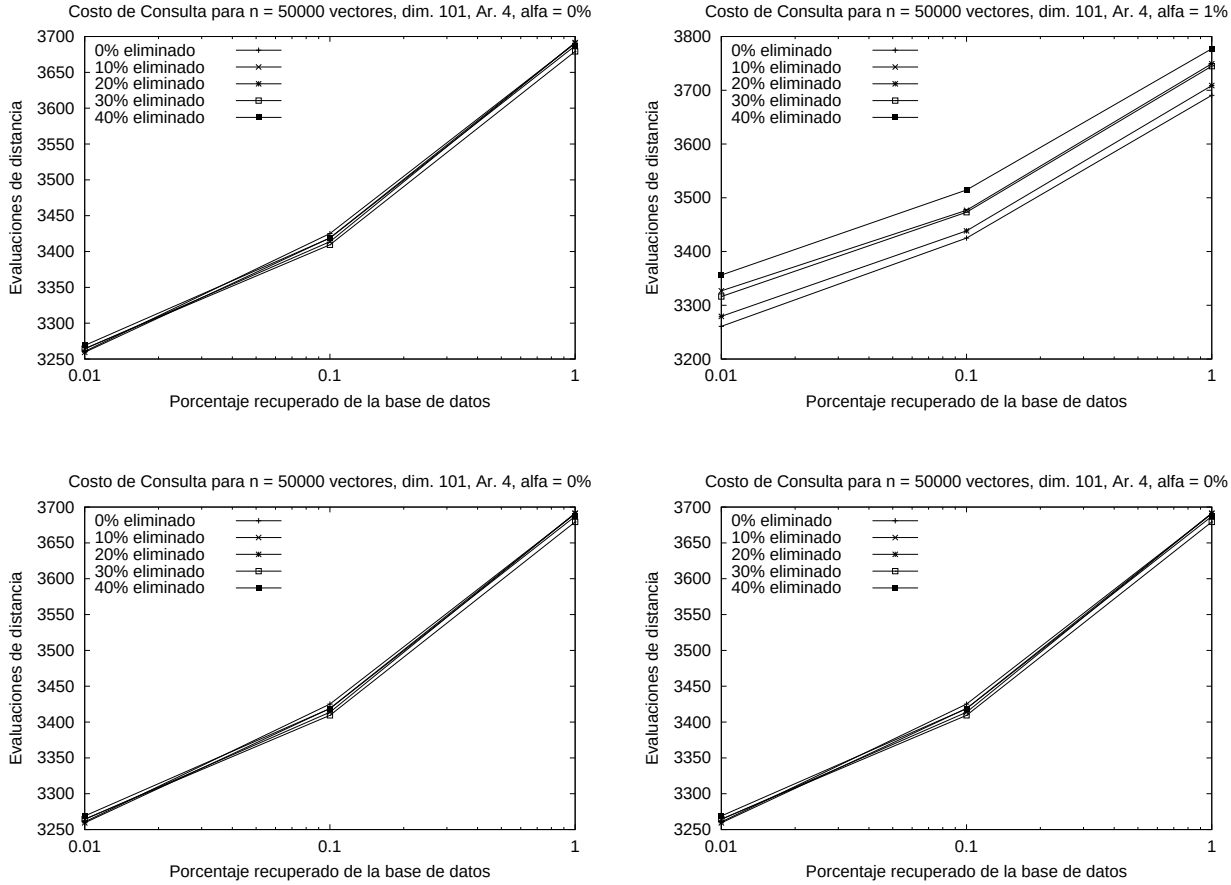


Figura A.38: Comparación de costos de búsqueda para $\alpha = 0\,\%$, $1\,\%$, $10\,\%$ y $100\,\%$ considerando los distintos porcentajes de elementos eliminados usando el método combinado de reconstrucción de subárbol para el espacio de vectores de dimensión 101 con aridad 4.

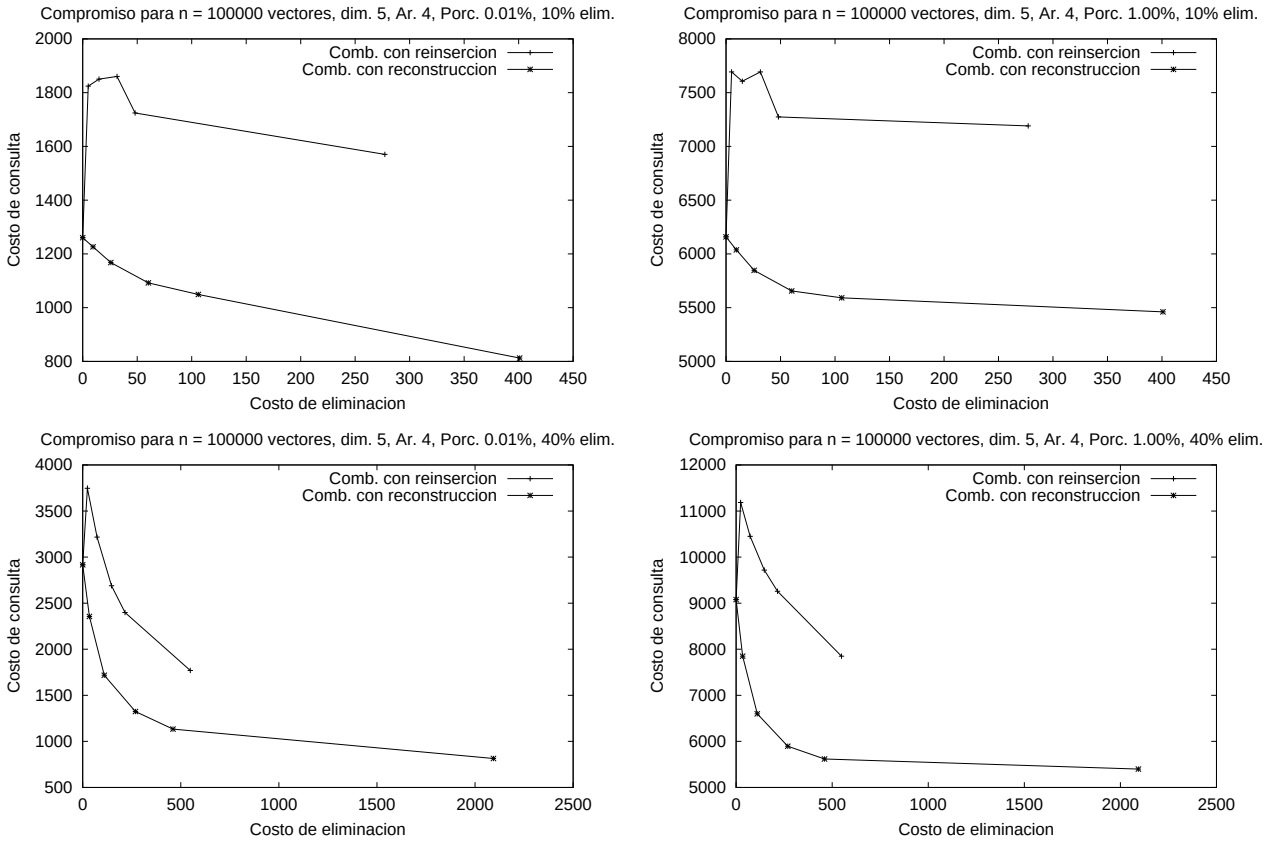


Figura A.39: Compromiso entre el método de reinserción de a elemento y el método de reconstrucción de subárboles, considerando el 10 % (arriba) y para el 40 % de elementos eliminados (abajo) y para las búsqueda por rango que recuperan el 0.01 % (izquierda) y 1 % (derecha) para el espacio de vectores de dimensión 5.

La Figura A.39 muestra la comparación entre los métodos de reinserción de a elementos y de reconstrucción de subárboles para el espacio de vectores de dimensión 5. Se han graficado en cada caso los costos de eliminación contra los de búsqueda para cada uno de los valores de α, para cuando se elimina el 10 % o el 40 % de los elementos y considerando las búsquedas por rango que recuperan el 0.01 % y el 1 % de los elementos.

Para verificar si la degradación que evidencia la búsqueda a medida que aumenta el número de elementos eliminados se debe a la sobredimensión de los radios de cobertura, realizamos el siguiente experimento: tomando el caso de $\alpha = 0\,\%$, luego de las eliminaciones corregimos los radios de cobertura de todos los nodos. Como se puede observar en la Figura A.40, para el espacio de vectores de dimensión 101, que éste no es el único motivo porque la mejora obtenida no es muy significativa.

La Figura A.41 muestra la comparación de los costos de las búsquedas para $\alpha = 0\,\%$ con los radios de cobertura corregidos para el espacio de vectores de dimensión 5.

Se puede observar que en el espacio de vectores con distribución de Gauss si se elimina el 10 % de los elementos se desempeña mejor el $\alpha = 100\,\%$ (*nodos ficticios* puros) que el $\alpha = 0\,\%$ (*reinserción* pura). En cambio al eliminar el 40 % de los elementos se invierte completamente cuál es el mejor α, y se

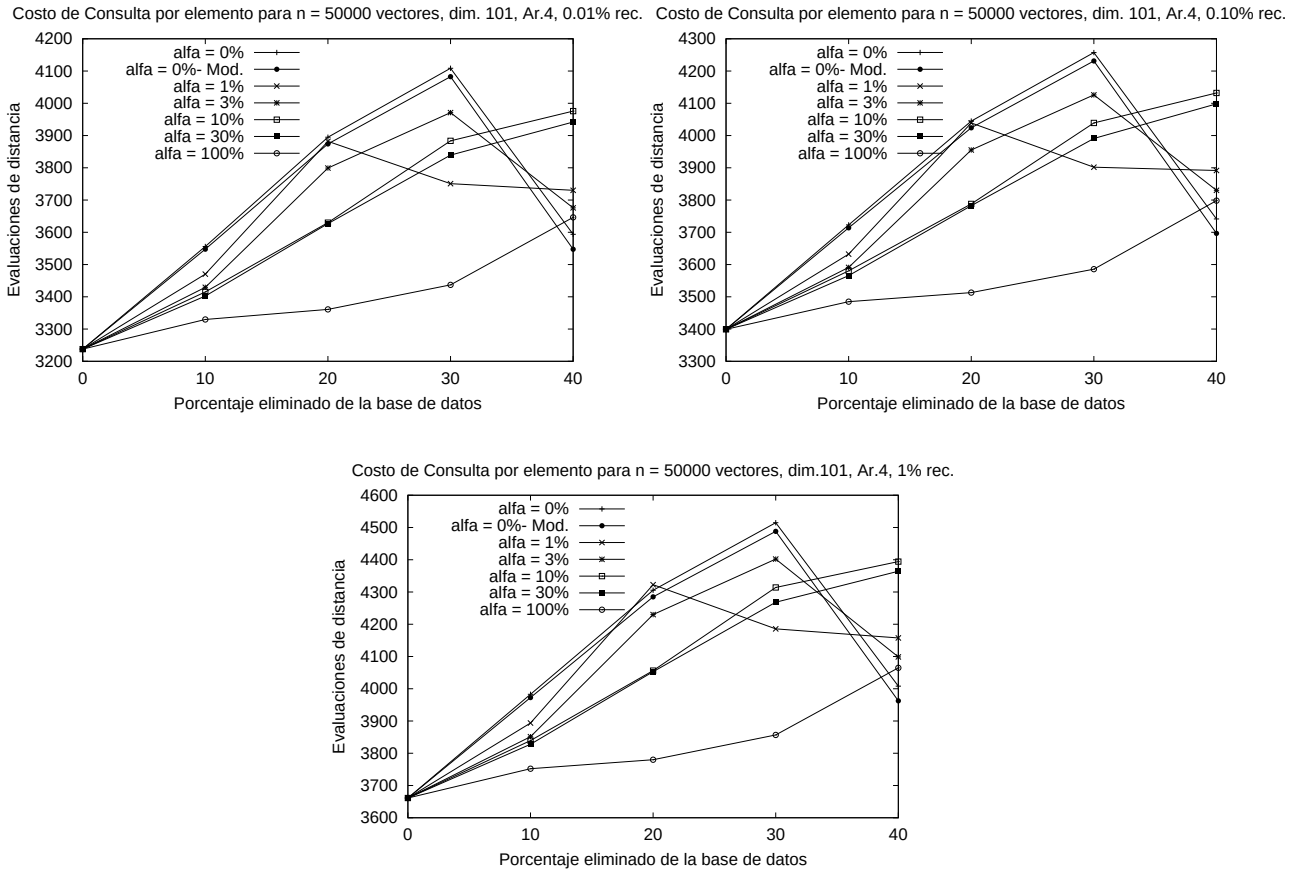


Figura A.40: Costos de búsqueda usando diferentes porcentajes de nodos eliminados, comparando α y comparando para $\alpha = 0\,\%$ con los radios de cobertura corregidos para el espacio de vectores de dimensión 101. Arriba a la izquierda recuperamos el 0.01 % de la base de datos y a la derecha el 0.10 %. Abajo recuperamos el 1 % de la base de datos.

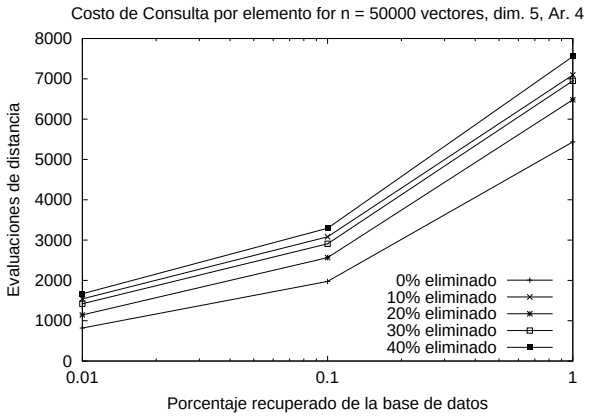


Figura A.41: Costos de búsqueda usando diferentes porcentajes de nodos eliminados, comparando para $\alpha = 0\,\%$ con los radios de cobertura corregidos, para el espacio de vectores de dimensión 5.

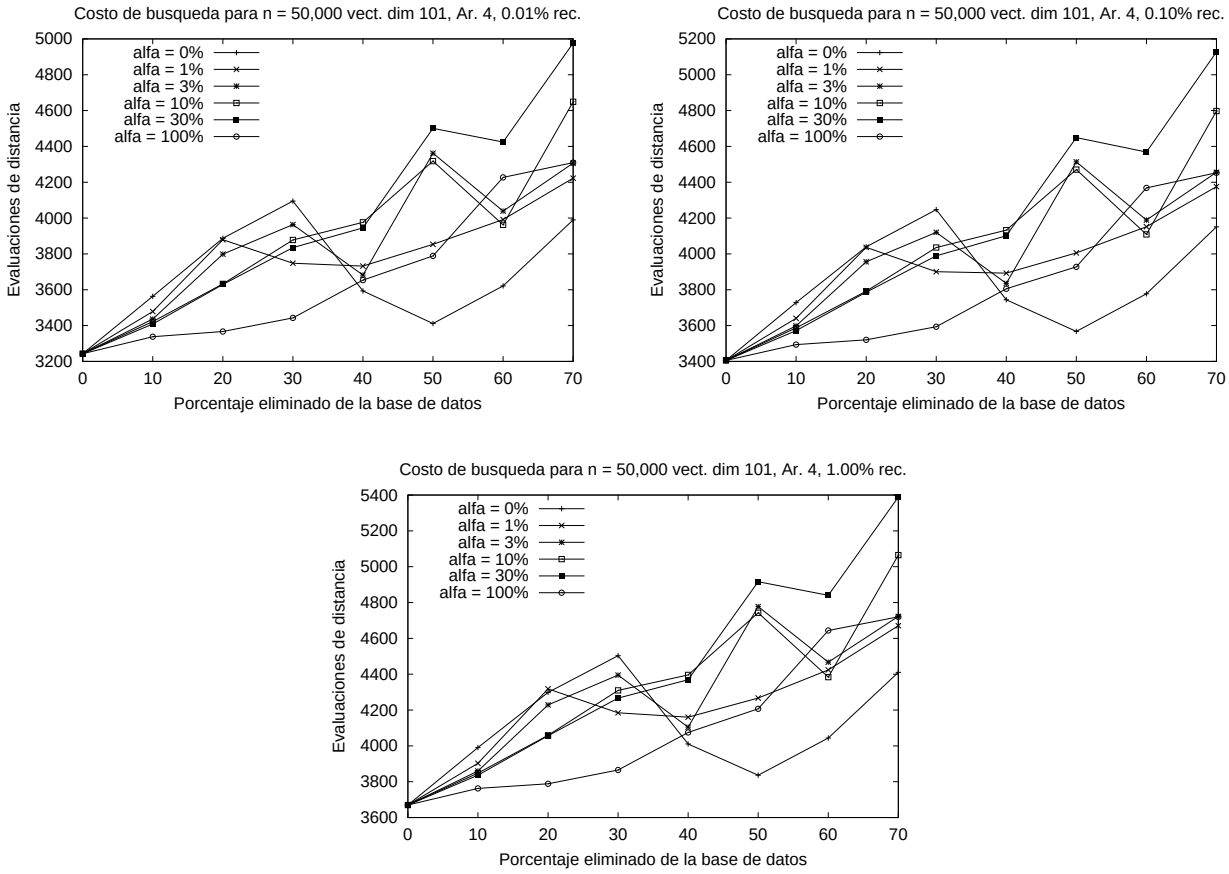


Figura A.42: Costos de búsqueda usando diferentes porcentajes de nodos eliminados, comparando α. Arriba a la izquierda recuperamos el 0.01 % de la base de datos y a la derecha el 0.10 %. Abajo recuperamos el 1 % de la base de datos.

aumenta la diferencia entre estas opciones extremas. Para tratar de entender mejor este comportamiento tan arbitrario realizamos el siguiente experimento: generamos un nuevo espacio de vectores de dimensión 101, con las mismas características del anterior, pero con 130000 elementos, generamos el árbol con 50000, 60000,. . .,120000 elementos y eliminamos la cantidad necesaria en cada caso para dejar sólo 50000 elementos en el árbol, luego realizamos las búsquedas con los 10000 elementos restantes (elegidos aleatoriamente). Por ahora sólo comprobamos que si eliminamos porcentajes aún mayores el comportamiento se estabiliza. La Figura A.42 muestra para cada porcentaje de elementos que recuperan las búsquedas el comportamiento para los distintos α respecto del número de elementos eliminados, agrupamos en este caso por el porcentaje recuperado en las búsquedas.

En otros experimentos, realizamos varias veces inserciones, eliminaciones y búsquedas, para observar cómo se comportaría el árbol en situaciones más próximas a las que pueden ocurrir en aplicaciones reales.

En el primero de estos experimentos armamos el árbol con el 90 % de los elementos, reservando el 10 % restante (elegido aleatoriamente) para las consultas y eliminamos y reinsertamos nuevamente varias veces un 70 % de los elementos. La Figura A.43 muestra los resultados obtenidos sobre el espacio de vectores de dimensión 15 (arriba) y para los diccionarios (abajo) de Inglés (izquierda) y de Francés (derecha) usando aridad 32 y usando el método de reinserción de a elementos.

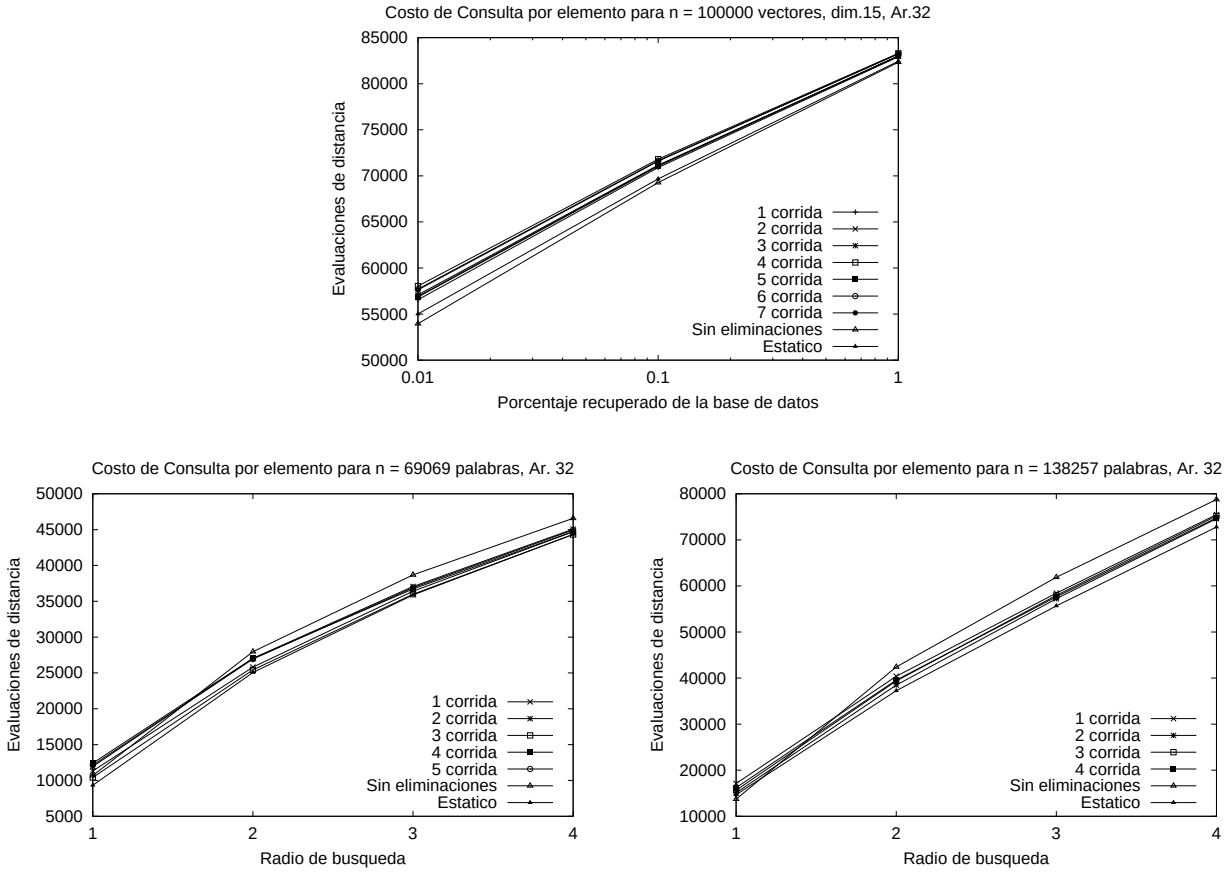


Figura A.43: Costos de búsqueda eliminando y reinsertando nuevamente un 70 % de los elementos para el espacio de vectores de dimensión 15 (arriba) y para los diccionarios (abajo) de Inglés (izquierda) y de Francés (derecha) usando aridad 32.

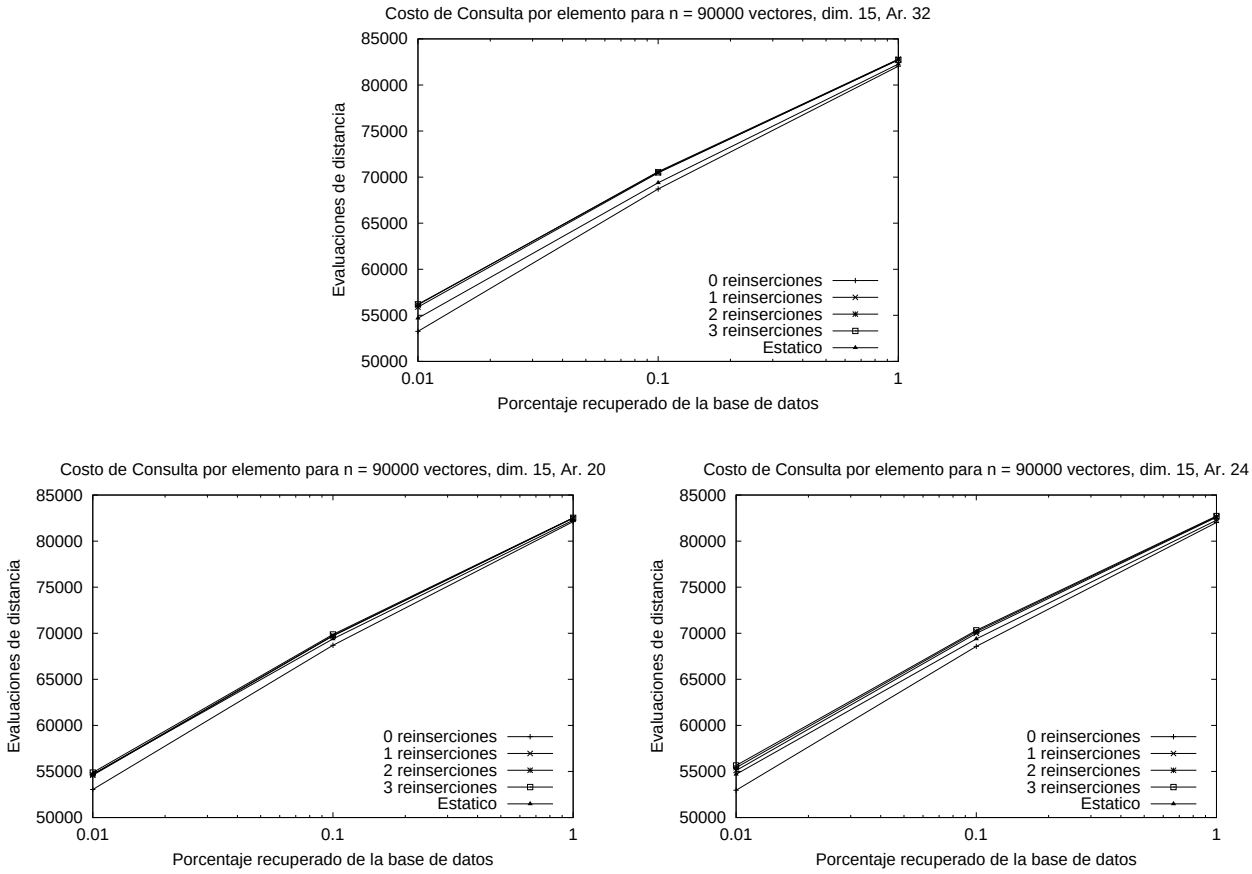


Figura A.44: Costos de búsqueda eliminando 70000 elementos e insertando 70000 nuevos varias veces, para el espacio de vectores de dimensión 15 usando aridades 32 (arriba), 20 (abajo, izquierda) y 24 (abajo, derecha).

Para el segundo de estos experimentos creamos el árbol con 90000 elementos, y varias veces eliminamos 70000, luego insertamos 70000 nuevos elementos, como siempre reservamos 10000 elementos (aleatoriamente elegidos) para realizar las búsquedas. La Figura A.44 muestra los resultados de este experimento usando el método de reinserción de a elementos para el espacio de vectores de dimensión 15 y usando distintas aridades.