Combinando Clustering con Aproximación Espacial para Búsquedas en Espacios Métricos *

Marcelo Barroso [1] **Gonzalo Navarro** [2] **Nora Reyes** [1]

Departamento de Informática, Universidad Nacional de San Luis.[1]
Departamento de Ciencias de la Computación, Universidad de Chile. [2]

mabarros@unsl.edu.ar gnavarro@dcc.uchile.cl nreyes@unsl.edu.ar

Resumen

El *modelo de espacios métricos* permite abstraer muchos de los problemas de búsqueda por proximidad. La búsqueda por proximidad tiene múltiples aplicaciones especialmente en el área de bases de datos multimedia. La idea es construir un índice para la base de datos de manera tal de acelerar las consultas por proximidad o similitud. Aunque existen varios índices prometedores, pocos de ellos son dinámicos, es decir, una vez creados muy pocos permiten realizar inserciones y eliminaciones de elementos a un costo razonable.

El *Árbol de Aproximación Espacial* (*dsa–tree*) es un índice recientemente propuesto, que ha demostrado tener buen desempeño en las búsquedas y que además es totalmente dinámico. En este trabajo nos proponemos obtener una nueva estructura de datos para búsqueda en espacios métricos, basada en el *dsa–tree*, que mantenga sus virtudes y que aproveche que en muchos espacios existen clusters de elementos y que además pueda hacer un mejor uso de la memoria disponible para mejorar las búsquedas.

Palabras Claves: *búsqueda por similitud, espacios métricos, bases de datos y algoritmos.*

1. Introducción y Motivación

Con la evolución de las tecnologías de información y comunicación, han surgido almacenamientos no estructurados de información. No sólo se consultan nuevos tipos de datos tales como texto libre, imágenes, audio y vídeo; sino que además, en algunos casos, ya no se puede estructurar más la información en claves y registros. Aún cuando sea posible una estructuración clásica, nuevas aplicaciones tales como la minería de datos requieren acceder a la base de datos por cualquier campo y no sólo por aquellos marcados como "claves". Estos tipos de datos son difíciles de estructurar para adecuarlos al concepto tradicional de búsqueda. Así, han surgido aplicaciones en grandes bases de datos en las que se desea buscar objetos similares. Este tipo de búsqueda se conoce con el nombre de *búsqueda aproximada* o *búsqueda por similitud* y tiene aplicaciones en un amplio número de campos. Algunos ejemplos son bases de datos no tradicionales, búsqueda de texto, recuperación de información, aprendizaje de máquina y clasificación; sólo para nombrar unos pocos.

Como en toda aplicación que realiza búsquedas, surge la necesidad de tener una respuesta rápida y adecuada, y un uso eficiente de memoria, lo que hace necesaria la existencia de estructuras de datos especializadas que incluyan estos aspectos. El planteo general del problema es: existe un universo $\mathbb{U}$ de *objetos* y una función de distancia positiva d: $\mathbb{U} \times \mathbb{U} \rightarrow \mathbb{R}^+$ definida entre ellos. Esta función de distancia satisface los tres axiomas que hacen que el conjunto sea un *espacio métrico*: positividad estricta ($d(x, y) = 0 \Leftrightarrow x = y$), simetría ($d(x, y) = d(y, x)$) y desigualdad triangular

*Este trabajo ha sido financiado parcialmente por el Proyecto RIBIDI CYTED VII.19 (todos los autores) y por el Centro del Núcleo Milenio para Investigación de la Web, Grant P04-067-F, Mideplan, Chile (último autor).

($d(x,z) \leq d(x,y) + d(y,z)$). Mientras más "similares" sean dos objetos menor será la distancia entre ellos. Tenemos una *base de datos* finita $S \subseteq \mathbb{U}$ que puede ser preprocesada (v.g. para construir un índice). Luego, dado un nuevo objeto del universo (un *query* q), debemos recuperar todos los elementos similares de que se encuentran en la base de datos. Existen dos consultas básicas de este tipo:

Búsqueda por rango: recuperar todos los elementos de S a distancia r de un elemento q dado.

Búsqueda de k vecinos más cercanos: dado q, recuperar los k elementos más cercanos a q en S.

La distancia se considera costosa de evaluar (por ejemplo, comparar dos huellas dactilares). Así, es usual definir la complejidad de la búsqueda como el número de evaluaciones de distancia realizadas, dejando de lado otras componentes tales como tiempo de CPU para computaciones colaterales, y aún tiempo de E/S. Dada una base de datos de $|S| = n$ objetos el objetivo es estructurar la base de datos de forma tal de realizar menos de n evaluaciones de distancia (trivialmente n bastarían).

Un caso particular de este problema surge cuando el espacio es un conjunto D-dimensional de puntos y la función de distancia pertenece a la familia L_p de Minkowski: $L_p = (\sum_{1 \leq i \leq d} |x_i - y_i|^p)^{1/p}$. Existen métodos efectivos para buscar sobre espacios D-dimensionales, tales como kd-trees [1] o R-trees [5]. Sin embargo, para 20 dimensiones o más esas estructuras dejan de trabajar bien.

Nos dedicamos en este trabajo a espacios métricos generales, aunque las soluciones son también adecuadas para espacios D-dimensionales. Es interesante notar que el concepto de "dimensionalidad" se puede también traducir a espacios métricos: la característica típica en espacios de alta dimensión con distancias L_p es que la distribución de probabilidad de las distancias tiene un histograma concentrado, haciendo así que el trabajo realizado por cualquier algoritmo de búsqueda por similaridad sea más dificultoso [2, 4]. Para espacios métricos generales existen numerosos métodos para preprocesar la base de datos con el fin de reducir el número de evaluaciones de distancia [4]. Todas aquellas estructuras trabajan básicamente descartando elementos mediante la desigualdad triangular, y la mayoría usa la técnica dividir para conquistar.

El Árbol de Aproximación Espacial Dinámico (*dsa–tree*) es una estructura de esta clase propuesta recientemente [7], basado sobre un nuevo concepto: más que dividir el espacio de búsqueda, aproximarse al query espacialmente, y además es completamente dinámica. El dinamismo completo no es común en estructuras de datos métricas [4]. Además de ser desde el punto de vista algorítmico interesante por sí mismo, se ha mostrado que el *dsa–tree* da un buen balance espacio-tiempo respecto de las otras estructuras existentes sobre espacios métricos de alta dimensión o consultas con baja selectividad, lo cual ocurre en muchas aplicaciones.

A diferencia de algunas otras estructuras de datos métricas [3], el *dsa–tree* no saca mayor provecho si el espacio métrico posee clusters, ni puede mejorar las búsquedas a costa de usar más memoria.

Nos proponemos obtener un nuevo índice para búsqueda en espacios métricos, basado en el *dsa–tree*, que mantenga sus virtudes, pero que aproveche que en muchos espacios existen clusters de elementos y que además haga un mejor uso de la memoria disponible para mejorar las búsquedas.

2. Árbol de Aproximación Espacial Dinámico

Describiremos brevemente aquí la aproximación espacial y el *dsa–tree*, más detalles en [6, 7].

Se puede mostrar la idea general de la aproximación espacial utilizando las *búsquedas del vecino más cercano*. En este modelo, dado $q \in \mathcal{U}$ y estando en algún elemento $a \in \mathcal{S}$ el objetivo es moverse a otro elemento de $\mathcal{S}$ más cercano "espacialmente" de q que a. Cuando no es posible realizar más este movimiento, estamos posicionados en el elemento más cercano a q de S. Las aproximaciones son efectuadas sólo vía los "vecinos". Cada $a \in S$ tiene un conjunto de vecinos $N(a)$.

Para construir incrementalmente al *dsa–tree* se fija una aridad máxima para el árbol y se mantiene información sobre el tiempo de inserción de cada elemento. Cada nodo a en el árbol está conectado

con sus hijos, los cuales forman el conjunto $N(a)$, los *vecinos* de a. Cuando se inserta un nuevo elemento x, se ubica su punto de inserción comenzando desde la raíz del árbol a y realizando el siguiente proceso. Se agrega x a $N(a)$ (como una nueva hoja) si (1) x está más cerca de a que de cualquier elemento $b \in N(a)$, y (2) la aridad del nodo a, $|N(a)|$, no es ya la máxima permitida. En otro caso, se fuerza a x a elegir el vecino más cercano en $N(a)$ y se continúa bajando en el árbol recursivamente, hasta que se alcance un nodo a tal que x esté más cerca de a que de cualquier $b \in N(a)$ y la aridad de a no haya alcanzado la máxima permitida, lo que eventualmente ocurrirá en una hoja del árbol. En ese punto se agrega a x como el vecino más nuevo en $N(a)$, se le coloca a x la marca del tiempo corriente y éste se incrementa. En cada nodo a del árbol se mantiene la siguiente información: el conjunto de vecinos $N(a)$, el tiempo de inserción del nodo $tiempo(a)$ y el radio de cobertura $R(a)$ que es la distancia entre a y el elemento de su subárbol que está más lejos de a.

La Figura 1 ilustra el proceso de inserción. Se sigue 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 *dsa–tree* se puede construir comenzando con un único nodo a donde $N(a) = \emptyset$ y $R(a) = 0$, y luego realizando sucesivas inserciones.

Insertar (Nodo a, Elemento x)
1. $R(a) \leftarrow \max(R(a), d(a,x))$
2. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
3. If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
4. $N(a) \leftarrow N(a) \cup \{x\}$
5. $N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
6. $tiempo(x) \leftarrow CurrentTime$
7. Else Insertar (c, x)

Figura 1: Inserción de un nuevo elemento x dentro del *dsa–tree* con raíz a. $MaxArity$ es la máxima aridad del árbol y $CurrentTime$ es el tiempo actual, que luego se incrementa en cada inserción.

Notar que no podemos asegurar que un nuevo elemento x sea vecino del primer nodo a que satisfaga estar más cerca de a que de cualquier elemento $b \in N(a)$. Es posible que la aridad de a fuera ya máxima y x fuera forzado a elegir un vecino de a, lo cual tiene consecuencias en la búsqueda.

Se ha conseguido demostrar experimentalmente en [7], que el desempeño de la construcción mejora a medida que se reduce la aridad máxima del árbol, siendo mucho mejor que la construcción estática del árbol de aproximación espacial (*sa–tree*). Por lo que la aridad reducida es un factor clave para bajar los costos de construcción.

La idea de la búsqueda por rango es replicar el proceso de inserción de los elementos relevantes para la consulta. Es decir, se procede como si se quisiera insertar q pero considerando que los elementos relevantes pueden estar a distancia hasta r de q. Así, en cada decisión, al simular la inserción de q se permite una tolerancia de $\pm r$; por lo tanto, puede ser que los elementos relevantes fueran insertados en diferentes hijos del nodo corriente, y sea necesario hacer backtracking.

Durante las búsquedas se debe considerar que cuando se insertó un elemento x, un nodo a en su camino pudo no haberse elegido como padre porque su aridad ya era máxima, como se indicó previamente. Entonces, en la búsqueda debemos seleccionar la mínima distancia sólo entre $N(a)$.

Otro hecho importante a 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. En otras palabras, consideremos los vecinos $\{v_1 \ldots v_k\}$ de a de más viejo a más nuevo, sin tener en cuenta a, entre la inserción de v_i y la de v_{i+j} pudieron aparecer nuevos elementos que eligieron a v_i porque v_{i+j} no estaba aún presente; así, estos elementos se deben tener en cuenta en la búsqueda y no se deben descartar por la existencia de v_{i+j}.

Las búsquedas se optimizan usando la información almacenada sobre el tiempo de inserción y el radio de cobertura, dado que nos permiten podar cierta parte del árbol.

La Figura 2 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 la primer invocación. En las líneas 1 y 6 se puede observar la optimización de la búsqueda por medio del tiempo de inserción y el radio de cobertura, como se destacó en el párrafo anterior. 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.

BúsquedaporRango (Nodo a, Query q, Radio r, Timestamp t)
1. If $tiempo(a) < t \ \wedge \ d(a,q) \leq R(a) + r$ Then
2. If $d(a,q) \leq r$ Then informar a
3. $d_{min} \leftarrow \infty$
4. For $b_i \in N(a)$ en orden creciente de timestamp Do
5. If $d(b_i,q) \leq d_{min} + 2r$ Then
6. $k \leftarrow$ mín $\{j > i, \ d(b_i,q) > d(b_j,q) + 2r\}$
7. BúsquedaporRango (b_i, q, r, $tiempo(b_k)$)
8. $d_{min} \leftarrow$ mín$\{d_{min}, d(b_i,q)\}$

Figura 2: Consulta de q con radio r en un *dsa–tree*.

Experimentalmente [7], se puede concluir que la mejor aridad para la búsqueda depende del espacio métrico, pero la regla, grosso modo, es que aridades bajas son buenas para dimensiones bajas o radios de búsqueda pequeños.

Las eliminaciones son más complicadas porque los cambios a realizar en la estructura son más costosos, y no son localizados. Sin embargo, en [7, 8] se muestran distintas posibilidades de eliminación.

Podemos concluir que el *dsa–tree* es completamente dinámico, como ya hemos destacado, y además respecto al *sa–tree* logra mejorar el costo de construcción. En algunos casos mejora el desempeño de las búsquedas, mientras que en otras paga un pequeño precio por su dinamismo. Por lo tanto, el *dsa–tree* se convierte en una estructura muy conveniente.

3. Nuestra Propuesta

El *dsa–tree* es una estructura que realiza la partición del espacio considerando la proximidad espacial, pero si el árbol lograra agrupar los elementos que se encuentran muy cercanos entre sí, lograría mejorar las búsquedas al evitarse el recorrido del árbol para alcanzarlos. Así, nos hemos planteado el estudio de una nueva estructura de datos que realice la aproximación espacial sobre clusters o grupos de elementos, en lugar de elementos individuales.

Podemos pensar entonces que construimos un *dsa–tree*, con la diferencia que cada nodo representa un grupo de elementos muy cercanos (“clusters”); y de este modo, logramos relacionar los clusters por su proximidad en el espacio. Por lo tanto, cada nodo de la estructura sería capaz de almacenar varios elementos de la base de datos. La idea sería que en cada nodo se mantenga un elemento, al que se toma como el centro del cluster correspondiente, y se almacenen los k elementos más cercanos a él, cualquier elemento a mayor distancia del centro que los k elementos, pasaría a formar parte de otro nodo en el árbol, que podría ser un nuevo vecino en algunos casos.

Al igual que en el *dsa–tree* para cada nodo n, se mantiene el radio de cobertura $R(n)$, el conjunto de vecinos del nodo $N(n)$, y el tiempo de inserción del nodo $tiempo\_nodo(n)$. Como cada nodo representará un cluster, se mantendrán el elemento que será el centro del cluster $centro(n)$,

los k elementos más cercanos al centro $cluster(n)$, junto con las distancias entre los elementos del cluster y su centro. Dado que como se mantienen la distancias entre el centro del cluster $centro(n)$ y sus elementos, conocemos el radio del cluster $rc(n)$, es decir la distancia al elemento más alejado del centro. Durante las búsquedas, se pueden utilizar ambos radios, $R(n)$ y $rc(n)$, para permitirnos descartar zonas completas del espacio. A continuación daremos detalles de esta nueva estructura.

3.1. Inserción

Para construir incrementalmente nuestra estructura de datos, mantenemos las consideraciones del *dsa–tree*, es decir que fijamos una aridad máxima para el árbol y almacenamos información del tiempo de inserción de cada nodo; pero además, mantenemos el tiempo de inserción $tiempo(a)$ para cada elemento a que se encuentra en el árbol.

Al intentar insertar un nuevo elemento en un nodo cuyo cluster ya tiene sus k elementos presentes, debemos decidir cuáles son los k elementos más cercanos al centro que deberían quedar en el cluster para mantener el mínimo volumen del mismo, esto nos permitirá mejorar el desempeño de las búsquedas. Entonces, para cada elemento en el cluster se podrían almacenar las distancias al centro y mantener los elementos del cluster ordenados por esta distancia; evitando así recalcular distancias para decidir quién queda y, por consiguiente, quién sale del cluster. Además estas distancias juegan un rol importante en el proceso de búsqueda.

Debido a la aproximación espacial, al insertar un nuevo elemento x, deberíamos bajar por el árbol hasta encontrar el nodo n tal que x esté más cerca de su centro a que de los centros de los nodos vecinos en $N(n)$. Si en el cluster de ese nodo hay lugar para un elemento más, se lo insertaría junto con su distancia $d(x, a)$. Si no hay lugar, elegiríamos el elemento más distante b entre los k elementos del cluster y x, es decir el $k + 1$ en el orden de distancias con respecto al centro a. A continuación deberíamos analizar dos casos posibles:

Si b es x: x se debería agregar como centro de un nuevo nodo vecino de n, si la aridad de n lo permite; en otro caso, debería elegir el nodo cuyo centro, entre todos los nodos vecinos en $N(n)$, es el más cercano y continuar el proceso de inserción desde allí.

Si b es distinto de x: b debería elegir al centro más cercano c entre a y los centros de los nodos vecinos en $N(n)$ que sean más nuevos que b, debido a que cuando b se insertó no se tuvo que comparar con ellos. Luego, si c es a, el proceso que sigue es idéntico a lo realizado cuando b es x; en otro caso, si c no es a, se continúa con la inserción de b desde el nodo cuyo centro es c.

La Figura 3 ilustra el proceso de inserción. La función es invocada como InsertarCluster(a,x), donde a es la raíz del árbol y x es el elemento a insertar. Como se puede observar, al igual que en el *dsa–tree*, sólo seguimos un camino desde la raíz del árbol hasta el cluster o el nodo padre, en caso de que sea un nuevo centro vecino, donde va a ser insertado el elemento. En la línea 3 se decide si el elemento x cae en el cluster del correspondiente nodo a, en primer lugar x debe estar más cerca del centro de a que de los centros de sus vecinos, y posteriormente determinamos si hay lugar dentro del cluster o si su distancia determina que debe ir dentro de él. Aquí se pueden observar los dos casos posibles que analizamos previamente, siempre y cuando el cluster ya contenga sus k elementos. El primer caso corresponde con la rama del Else, línea 11, de la condición que determina si el elemento debe pertenecer al cluster (línea 3). Luego, se determina si al elemento se lo puede ubicar como el centro de un nuevo nodo vecino b, siempre y cuando la máxima aridad lo permita; en caso contrario, la inserción va a continuar por el centro vecino más cercano. El segundo caso que analizamos, se corresponde la rama del Then (línea 4) de la condición, donde el elemento se inserta en el cluster del nodo, pero el elemento más distante se reinsertará en el árbol; líneas 8, 9 y 10. Podemos observar

que cuando se inserta un nuevo elemento x en el cluster se almacena la distancia al centro $d'(x)$ y su tiempo de inserción $tiempo(x)$, lineas 5 y 6 respectivamente.

Cabe destacar que cuando se reinserta un elemento y, por quedar afuera del cluster (línea 10), éste ya tendrá su tiempo de inserción. Por lo tanto, este valor será reutilizado en vez de asignarle uno nuevo. Además el elemento y sólo se compara con los centros vecinos más nuevo que él en la línea 2.

```
InsertarCluster (Nodo a, Elemento x)
1.  R(a) ← máx(R(a), d(centro(a), x))
    // Sea rc(a) la distancia desde centro(a) al elemento más alejado en cluster(a)
2.  c ← argmin_{b∈N(a)} d(centro(b), x)
3.  If (d(centro(a), x) < d(centro(c), x)) ∧ ((|cluster(a)| < k) ∨ (d(centro(a), x) < rc(a))) Then
4.      cluster(a) ← cluster(a) ∪ {x}
5.      d'(x) ← d(centro(a), x)
6.      tiempo(x) ← CurrentTime
7.      If (|cluster(a)| = k + 1) Then
8.          y ← elemento en cl(a) con mayor d'
9.          cluster(a) ← cluster(a) − {y}
10.         InsertarCluster(a, y)
11.  Else
12.     If d(centro(a), x) < d(centro(c), x) ∧ |N(a)| < MaxArity Then
13.         N(a) ← N(a) ∪ {b}  //b es el nuevo nodo vecino de a, con x como centro
14.         centro(b) ← x
15.         N(b) ← ∅,  R(b) ← 0
16.         cluster(b) ← ∅
17.         tiempo(x) ← CurrentTime
18.         tiempo_nodo(b) ← CurrentTime
19.     Else
20.         InsertarCluster (c, x)
```

Figura 3: Inserción de un nuevo elemento x en el árbol con raíz a. $MaxArity$ es la máxima aridad del árbol, k la capacidad del cluster y $CurrentTime$ el tiempo actual.

El árbol se puede construir comenzando con un único primer nodo a, donde contiene el elemento centro $centro(a)$, $N(a) = \emptyset$, $R(a) = 0$ y $cluster(a) = \emptyset$; y luego realizando sucesivas inserciones. Los siguientes k inserciones, desde el primer único nodo se tomarán como los k objetos del cluster del nodo raíz y luego, al insertar el $k+2$, recién se creará un nuevo nodo cuyo centro será el elemento más distante del cluster del nodo raíz.

Teniendo en cuenta la manera en que se realizan las inserciones, es posible observar que ninguna inserción modificaría el centro de un cluster y además que es posible que durante una inserción se cree a lo sumo un nuevo nodo y que ésta afecte posiblemente a varios nodos.

3.2. Búsquedas

En la búsqueda de un elemento q con radio r procedemos similarmente al *dsa–tree*, es decir realizando aproximación espacial entre los centros de los nodos. También consideramos los dos hechos importantes que influyen en el proceso de búsqueda del *dsa–tree*. El hecho de que cuando insertamos (o reinsertamos) un elemento x, un nodo a pudo no haberse elegido como padre por que su aridad ya era máxima; y que x pudo elegir su centro vecino más cercano sólo entre los nodos más viejos que x.

Además de las consideraciones previamente mencionadas, al tener clusters en los nodos debemos verificar si hay o no intersección entre la zona consultada y el cluster. Más aún, si no la hay

se pueden descartar todos los elementos del cluster sin necesidad de compararlos contra q. A través del radio del cluster de un nodo a, $rc(a)$, verificamos si existe dicha intersección. Si el cluster no se pudo descartar para la consulta, usamos al centro de cada nodo como un pivote para los elementos x_i que se encuentran en el cluster, ya que mantenemos las distancias $d'(x_i)$ respecto del centro del nodo a. De esta manera es posible que, evitemos algunos cálculos de distancia entre q y los x_i, si $|d(q, centro(a)) - d'(x_i)| > r$. Es importante destacar que si la zona consultada cae completamente dentro de un cluster, podemos estar seguros que en ninguna otra parte del árbol encontraremos elementos relevantes para esa consulta.

La Figura 4 muestra el algoritmo de búsqueda, que se invoca inicialmente como BúsquedaRangoCluster(a,q,r,$CurrentTime$), donde a es la raíz del árbol. En la línea 3 del algoritmo, se determina si existe intersección entre el cluster y la zona consultada, posteriormente en caso de que haya intersección la línea 7 determina si la búsqueda debe continuar o no.

```
BusquedaRangoCluster (Nodo a, Query q, Radio r, Timestamp t)
1.  If tiempo_nodo(a) < t  ∧  d(centro(a), q) ≤ R(a) + r  Then
2.     If d(centro(a), q) ≤ r  Then Informar a
3.     If (d(centro(a), q) − r ≤ rc(a)) ∨ (d(centro(a), q) + r ≤ rc(a))  Then
4.        For c_i ∈ cluster(a)  Do
5.           If |(d(centro(a), q) − d'(c_i)| ≤ r  Then
6.              If (d(c_i, q) ≤ r  Then Informar c_i
7.        If d(centro(a), q) + r < rc(a)  Then Terminar búsqueda
8.     d_min  ←  ∞
9.     For b_i ∈ N(a) en orden creciente de timestamp Do
10.       If d(centro(b_i), q) ≤ d_min + 2r  Then
11.          k ← mín  {j > i,  d(centro(b_i), q) > d(centro(b_j), q) + 2r}
12.          BúsquedaRangoCluster (b_i, q, r, time(b_k))
13.       d_min ← mín{d_min, d(centro(b_i), q)}
```

Figura 4: Consulta de q con radio r en nuestra nueva estructura.

4. Resultados Experimentales

Para los experimentos hemos seleccionado dos espacios métricos diferentes.

Imágenes de la NASA: 40700 vectores de características, en dimensión 20, generado de imágenes descargadas de la NASA. Usamos la distancia Euclídea.
`http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`.
Éste es un espacio fácil (el histograma de distancia no es tan concentrado).

Palabras: un diccionario con 69069 palabras en inglés. La distancia es la *distancia de edición*, esto es, el mínimo número de inserciones, eliminaciones o cambios de caracteres necesarios para que dos palabras sean iguales. Se usa en recuperación de textos para manejar errores de ortografía, de escritura y reconocimiento óptico de caracteres (OCR). Es de dificultad media baja.

En ambos casos, construimos los índices con el 90 % de los elementos y usamos el 10 % restante (seleccionado aleatoriamente) como consultas. Hemos considerado en las consulta por rango recuperar el 0.01 %, el 0.1 % y el 1 % de la base de datos, cuando la función de distancia es continua. Esto quiere decir radios $0{,}605740$, $0{,}780000$ y $1{,}009000$ para las imágenes de la NASA. Como las palabras tienen una distancia discreta, entonces usamos radios de 1 a 4, el cual recupera el 0.00003 %, 0.00037 %, 0.00326 % y 0.01757 % de la base de datos, respectivamente. Las mismas consultas fueron usadas para todos los experimentos sobre la misma base de datos. Realizamos 10 ejecuciones de cada

experimento sobre permutaciones distintas de la base de datos; por lo tanto, los resultados exhibidos corresponden a los valores medios obtenidos. Como existen algoritmos de rango óptimos para las búsquedas de los k-vecinos más cercanos, no hemos realizado experimentos para esta clase de búsquedas separadamente.

Algunos de los experimentos que hemos realizado comparan el costo de construcción incremental de esta estructura contra el *dsa–tree* y el *sa–tree* para conjuntos crecientes de la base de datos. Experimentamos con los tamaños de cluster 10, 50, 100, 150 y 200, y con las aridades 2, 4, 8, 16, 32 y sin límite de aridad. De los resultados obtenidos podemos deducir principalmente que la aridad reducida es un factor clave para bajar los costos de construcción como ocurría con el *dsa–tree*, pero además el tamaño del cluster es también un factor importante para el costo de construcción, ya que a medida que el cluster crece los costos decrecen significativamente.

En la Figura 5 se muestran los costos de construcción para el espacio de vectores de imágenes de la NASA, comparando el comportamiento de los distintos tamaños de cluster para cada aridad. En la Figura 6 se muestran los costos de construcción para el espacio de palabras, comparando también el comportamiento de los distintos tamaños de cluster para cada aridad.

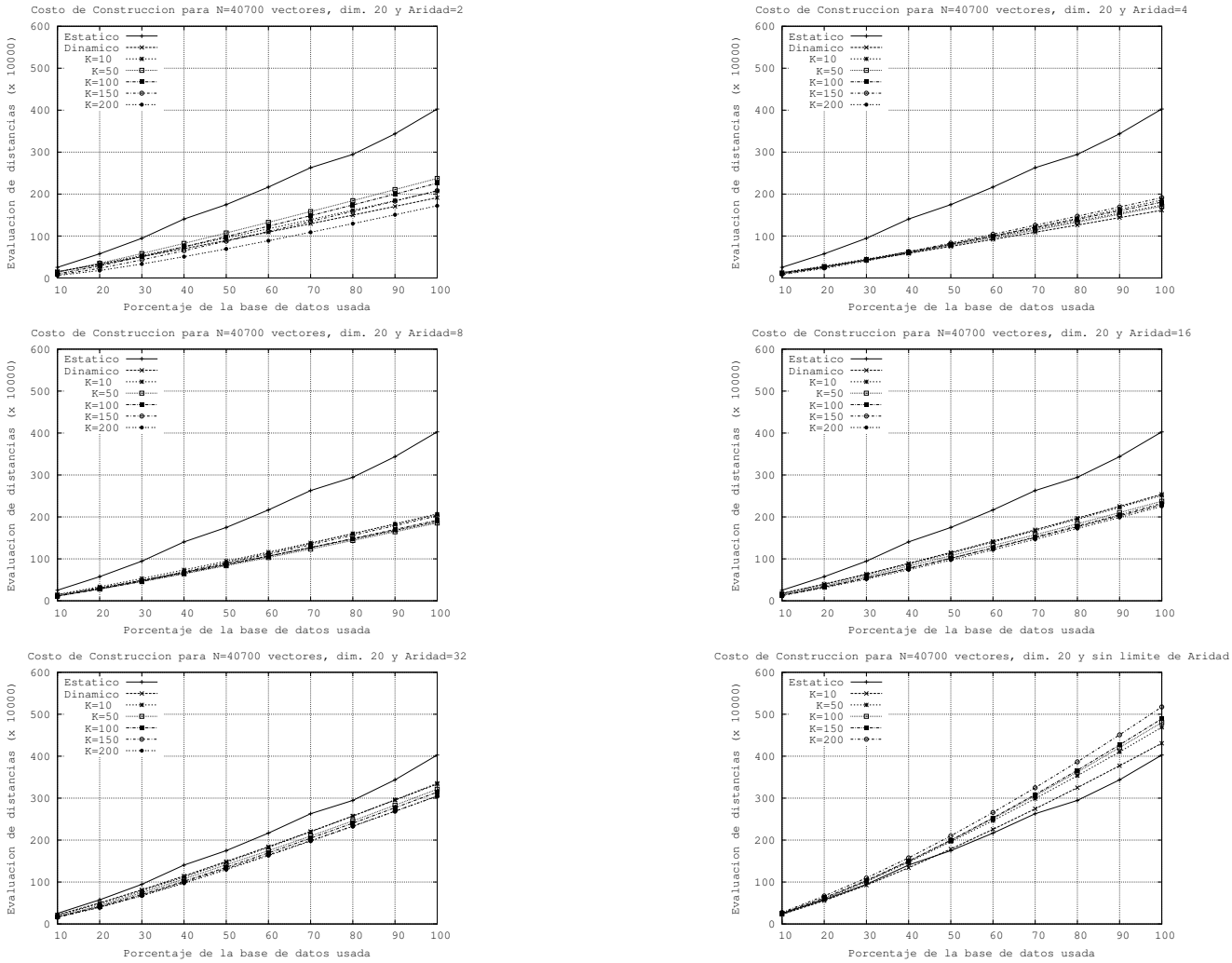

Figura 5: Costos de construcción para el espacio de vectores de imágenes de la NASA, utilizando distintas aridades y comparando los distintos tamaños de cluster.

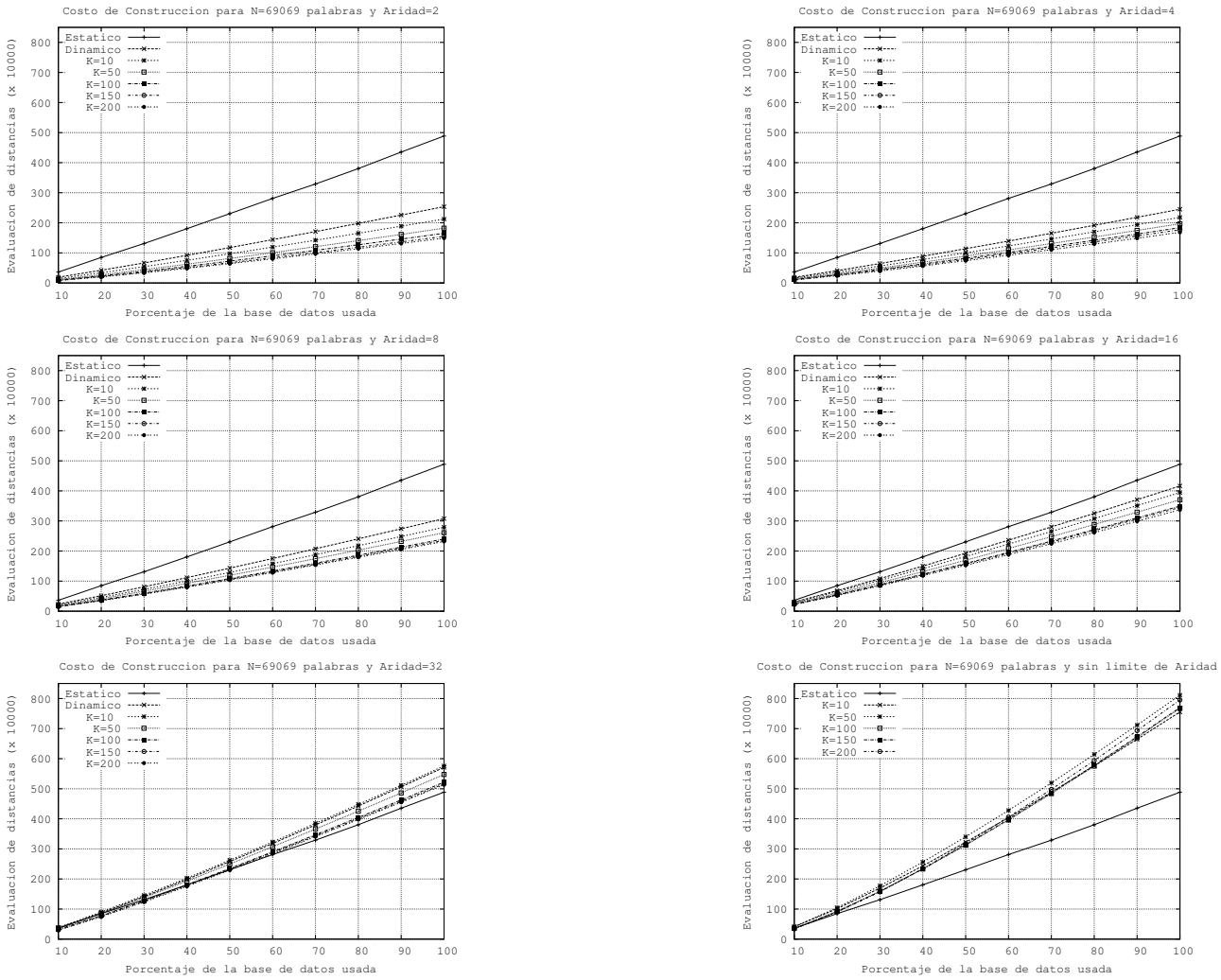

Figura 6: Costos de construcción para el espacio de palabras, utilizando distintas aridades y comparando los distintos tamaños de cluster.

En las Figuras 7 y 8 se muestran los costos promedios de búsqueda de un elemento para el espacio de vectores de imágenes de la NASA. La Figura 7 compara el comportamiento de los distintos tamaños de cluster para cada aridad. Como puede observarse en este espacio para todas las aridades e incluso no limitando la aridad, el mejor tamaño de cluster para las búsquedas es $k = 10$. Así, en Figura 8 mostramos la comparación usando k = 10, para las distintas aridades (a la izquierda). Claramente la mejor aridad para todos los radios de búsqueda considerados es 8. A la derecha de Figura 8 comparamos nuestra estructura con sus mejores parámetros contra el *dsa–tree* para todas las aridades. Es destacable que hemos obtenido un mejor desempeño en las búsquedas que el *dsa–tree* y el *sa–tree*.

En las Figuras 9 y 10 se muestran los costos promedios de búsqueda de un elemento para el diccionario. La Figura 9 compara los distintos tamaños de cluster para cada aridad y la Figura 10 muestra los mismos resultados con los mejores tamaños de cluster observados, es decir para $k = 10$ y $k = 50$. En aridades altas mejora el comportamiento usando $k = 50$ para radios pequeños. Si usamos $k = 10$ mejoran las búsquedas en aridades bajas para todos los radios y en aridades altas para radios grandes. En este espacio también los mejores parámetros para las búsquedas parecen ser $k = 10$ con aridad ilimitada, o con aridad 32, porque se comportan mejor que $k = 50$ para los radios 1, 2 y 3. Podemos observar que hemos logrado mejorar el desempeño de la búsqueda del *sa–tree* en el diccionario, donde el *dsa–tree* no lo lograba [7]. La aridad 32 e ilimitada son las que obtienen mejor

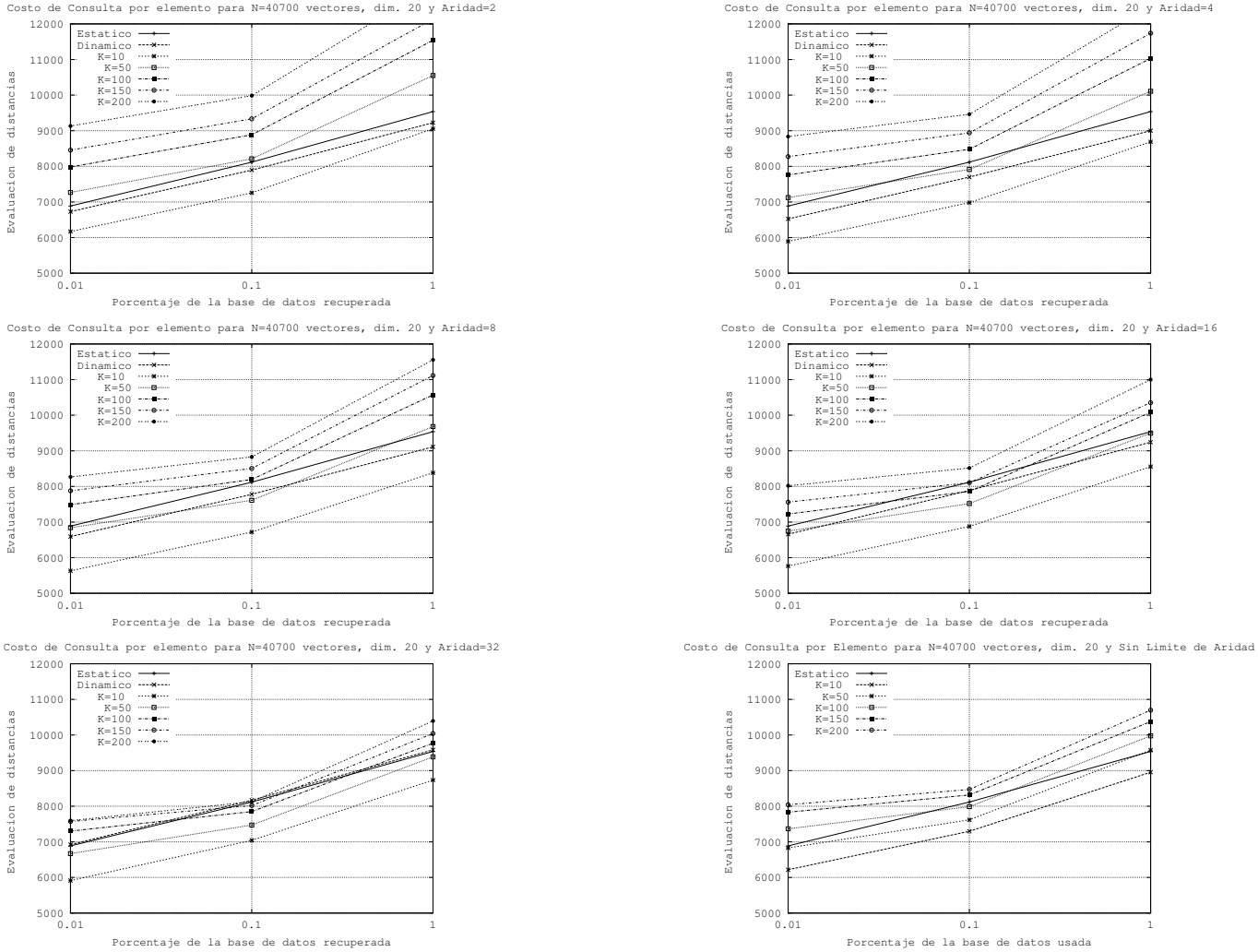

Figura 7: Costos de búsqueda para el espacio de vectores de imágenes de la NASA, utilizando distintas aridades y comparando los distintos tamaños de cluster.

desempeño superando ampliamente al *sa–tree* y al *dsa–tree*.

5. Conclusiones y Trabajo Futuro

Hemos presentado una nueva estructura para búsqueda en espacios métricos que permitiendo agrupar los elementos de la base de datos y manteniendo el dinamismo del *dsa–tree*, logra reducir significativamente los costos de búsqueda. Además hemos mejorado también el costo de construcción.

Para los costos de construcción el tamaño del cluster juega un rol importante, mientras mayor sea el cluster menor será el costo. La aridad reducida también es un factor clave para reducir el costo de construcción. La situación para las búsquedas es distinta, el tamaño de cluster y la aridad dependen del espacio métrico en particular.

Aunque el tamaño de cluster para obtener un mejor desempeño en las búsquedas dependa del espacio, si comparamos con el *sa–tree* y *dsa–tree*, podemos asegurar que independientemente del tamaño del cluster elegido lograremos reducir o mantener el costo de construcción.
Entre los temas que pretendemos analizar a futuro, podemos citar la siguientes:

- Experimentar con otros espacios métricos y analizar los resultados obtenidos.
- Analizar y experimentar con las distintas técnicas de eliminación existentes para el árbol de aproximación espacial.

Figura 8: Costos de búsqueda para el espacio de vectores de imágenes de la NASA, utilizando tamaño de cluster $k = 10$.

- Dado que los nodos tienen tamaño fijo, esta estructura parecería adecuada para memoria secundaria. Pero, ¿sería realmente eficiente en memoria secundaria?
- ¿Es posible mantener el cluster separado del nodo, y permitir que en cada nodo se almacene el centro, con su información asociada y los centros vecinos? ¿Sería una mejor opción para memoria secundaria?
- ¿Sería más adecuado mantener los clusters con cantidad fija de elementos o con radio fijo?
- ¿Existen otras maneras de combinar técnicas de clustering con aproximación espacial para búsquedas en espacios métricos?

Referencias

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

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

[3] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*. To appear.

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

[5] 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.

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

[7] 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.

[8] G. Navarro and N. Reyes. Improved deletions in dynamic spatial approximation trees. In *Proc. of the XXIII International Conference of the Chilean Computer Science Society (SCCC'03)*, pages 13–22. IEEE CS Press, 2003.

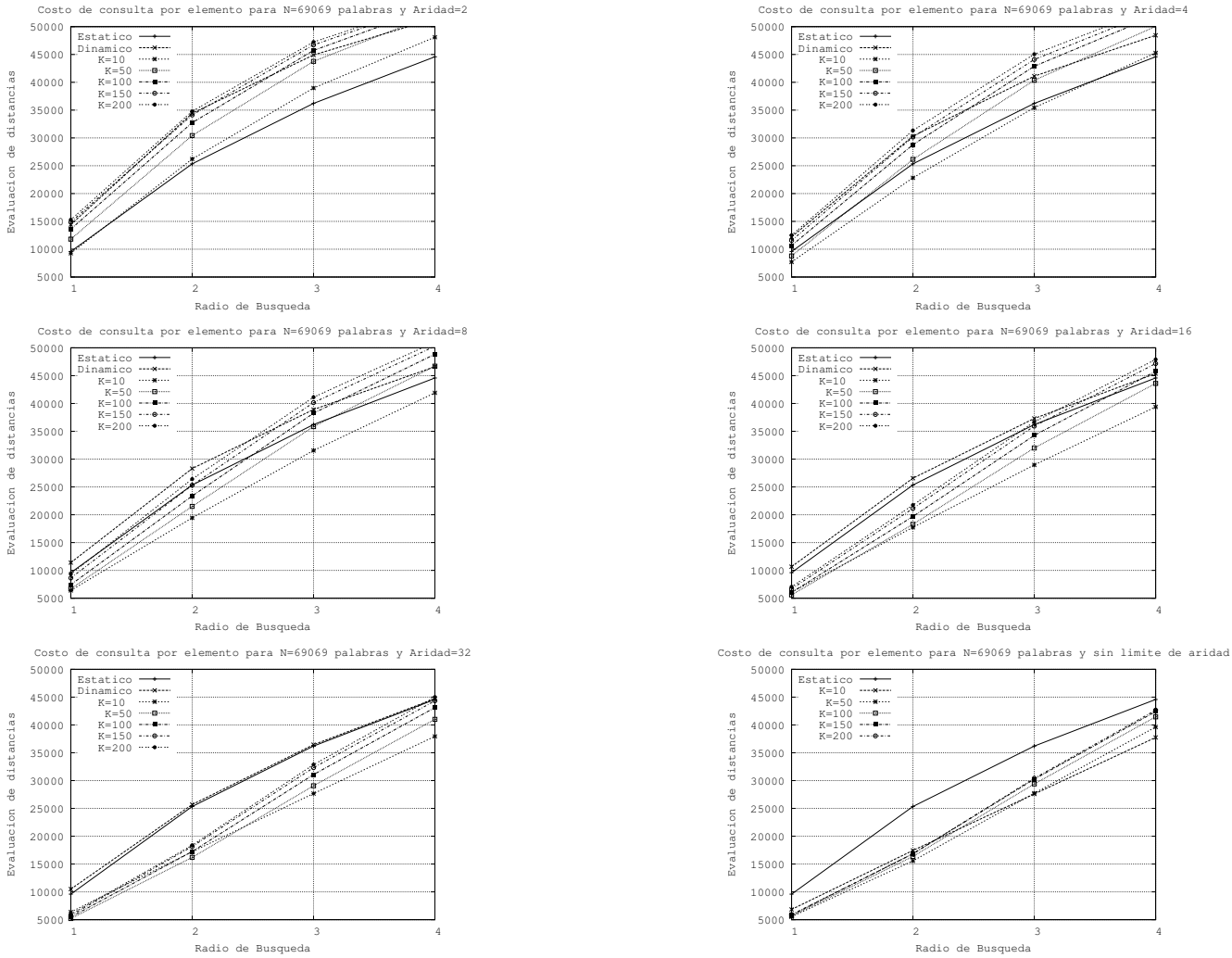

Figura 9: Costos de búsqueda para el espacio de palabras, utilizando distintas aridades y comparando los distintos tamaños de cluster.

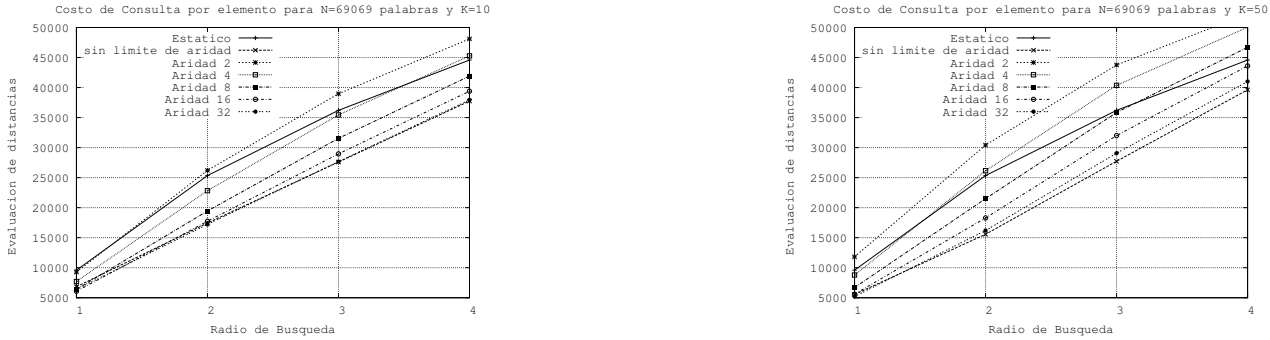

Figura 10: Costos de búsqueda para el espacio de palabras, utilizando tamaños de cluster de $k = 10$ y $k = 50$, comparando las distintas aridades.