UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DEPARTAMENTO DE ELECTRÓNICA Y TELECOMUNICACIÓN

TESIS DOCTORAL

ESTRUCTURA Y ESQUEMAS DE BÚSQUEDA POR SIMILITUD DE CADENAS DE CARACTERES. UNA APLICACIÓN PARA PETICIONES COMPLEJAS DE LOCALIZACIÓN DE PALABRAS EN ARCHIVOS DOCUMENTALES

MARGARITA DÍAZ ROCA

Las Palmas de Gran Canaria, 1993

30-1992/93

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
UNIDAD DE TERCER CICLO Y POSTGRADO

Reunido el día de la fecha,el Tribunal nombrado por el Excmo. Sr.Rector Magfco. de esta Universidad, la aspirante expuso esta TESIS DOCTORAL.

Terminada la lectura y contestadas por la Doctoranda las objeciones formuladas por los señores jueces del Tribunal,éste calificó dicho trabajo con la nota de APTO CUM LAUDE

Las Palmas de G. C., a 22 de Junio de 1993. Por unanimidad.

El Presidente: Dr. D. Antonio Núñez Ordoñez,

El Secretario: Dr. D. Manuel Pérez Cota,

El Vocal: Dr. D. Manuel Alvar Ezquerra,

El Vocal: Dr. D. Francisco Sanchís Mareo,

La Vocal: Drª Dª Mª Victoria Rodriguez Uría,

La Doctoranda: Dª Margarita Díaz Roca,

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
E. T. S. I. TELECOMUNICACIÓN

TESIS DOCTORAL

Estructura y esquemas de búsqueda por similitud de cadenas de caracteres.

Una aplicación para peticiones complejas de localización de palabras en archivos documentales

Autora: Dª. Margarita Díaz Roca
Director: Dr. D. Octavio Santana Suárez
Departamento de Electrónica y Telecomunicación
Mayo 1993

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DOCTORADO EN INGENIERÍA DE TELECOMUNICACIÓN

DEPARTAMENTO DE ELECTRÓNICA Y TELECOMUNICACIÓN
PROGRAMA DE INGENIERÍA ELECTRÓNICA

ESTRUCTURA Y ESQUEMAS DE BÚSQUEDA POR SIMILITUD DE CADENAS DE CARACTERES.

UNA APLICACIÓN PARA PETICIONES COMPLEJAS DE LOCALIZACIÓN DE PALABRAS EN ARCHIVOS DOCUMENTALES

Tesis Doctoral presentada por D^{a}. Margarita Díaz Roca
Dirigida por el Dr. D. Octavio Santana Suárez

El Director, *La Doctoranda,*

Las Palmas de Gran Canaria. Mayo, 1993

RESUMEN

Este trabajo trata aspectos teóricos y experimentales en torno al problema de la búsqueda de las cadenas más similares a una dada. El concepto de similitud es en el sentido de la distancia de Levenshtein, *DL*. El objetivo que se persigue es la optimización de los recursos de tiempo y espacio de los esquemas de búsqueda y de la estructura de datos que los soporta.

Se define una nueva distancia que se ha denominado distancia invariante trasposicional, *DIT*, debido al hecho de que su valor no depende de las operaciones de trasposición a que pueda ser sometida una cadena. Si bien *DIT* no puede usarse por si sola para la determinación de las cadenas más similares, su importancia deviene de la circunstancia de que su valor entre dos cadenas es siempre inferior o igual a la *DL* entre estas dos mismas cadenas, siendo su coste computacional sensiblemente inferior; lo cual puede ser aplicado para la construcción de un filtro adaptivo *DIT/DL* que tenga por misión reducir el número de cadenas de la base de datos a las que se les calcula la *DL* con la cadena de búsqueda.

Se diseña una estructura, *S-D*, al objeto de compartir las componentes de *DIT* y no tener que calcular completamente la *DIT* de la cadena de búsqueda a todas y cada una de las cadenas del diccionario. El esquema de búsqueda de las cadenas más similares que se apoya en esta estructura, recorriéndola a través de las componentes de *DIT*, y que usa este valor como criterio de poda se denomina esquema decreciente. Se estudian nuevas estrategias para un esquema de búsqueda creciente, donde el radio de búsqueda, en oposición a la evolución clásica decreciente, sigue una línea de modificación creciente. Asimismo, se propone un esquema decreciente con radio ascendente tal que en función del incremento del radio de búsqueda define una familia de esquemas intermedios que conectan a los esquemas creciente y decreciente.

Prolongando la línea de optimización de las realizaciones de los esquemas de búsqueda decreciente y creciente, se define un nuevo umbral *DS*, cuyo valor se encuentra entre *DIT* y *DL*; y debido a que tiene un menor costo que *DL* es útil para descartar un número de cadenas a las que no es necesario evaluar *DL*. También se

introduce un refinamiento en la poda del índice que acorta su recorrido.

Si el tamaño de la estructura es tal que no es posible ubicarla en memoria interna, se plantea el problema de su paginación con el fin de minimizar el número de accesos a disco. Un primer intento para resolver el problema consiste en llenar las páginas con los nodos al recorrer la estructura en preorden o en postorden. También se propone una nueva forma de paginar la estructura de tal modo que se respete el siguiente principio: durante la búsqueda, una vez que se abandone una página no se vuelve a acceder a ella por otro camino.

Finalmente se concreta en una aplicación a la localización de palabras en texto libre, que constituye una parte fundamental de un problema práctico de gran importancia: la organización y utilización de información procedente de fuentes heterogéneas. Existen dos aproximaciones a este problema, efectuar la búsqueda directamente sobre el documento sin ningún tipo de preparación previa o someter los escritos a un tratamiento anterior y generar un índice que haga viable los accesos posteriores al ejemplar. Siguiendo este último enfoque, se utiliza como índice la estructura *S-D* construida a partir del conjunto de las palabras diferentes que se obtienen del documento al descartar las cadenas consideradas *vacías*. Sobre la estructura así construida se permiten los siguientes tipos de búsquedas: exacta, más similares, con operadores booleanos, máscaras, truncamientos, cercanía, antecedencia, párrafos, sentencias, frases y complejas.

Se construye asimismo un analizador sintáctico que cumple un doble cometido, ya que ha de identificar cada tipo de petición —diferenciando las componentes y los conectores lógicos en el caso de las complejas— y, a la vez, determinar la correctitud sintáctica. Como complemento al analizador se realiza un optimizador para solucionar las búsquedas complejas en el menor tiempo posible; se pretende calcular una petición equivalente a la solicitada por el usuario de forma que se resuelva más rápidamente, minimizando así el tiempo de respuesta del sistema.

A Paco, Aída y Teba

AGRADECIMIENTOS

Quiero expresar mi profundo agradecimiento al Profesor D. Octavio Santana Suárez, Director del Grupo de Investigación en Estructuras de Datos en el cual se ha desarrollado mi trabajo, por haber desempeñado un papel esencial en mi formación científica y profesional. Sin sus reflexiones, consejos y dirección no hubiera sido posible realizar esta Tesis.

A todos mis compañeros del Grupo de Investigación en Estructuras de Datos, mi agradecimiento especial por haberme animado siempre. Gracias por su colaboración, ayuda incondicional y disponibilidad con la que siempre he podido contar.

También quiero agradecer al Departamento de Electrónica y Telecomunicación y a la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad de Las Palmas de Gran Canaria, y en particular a su Director, el Profesor D. Antonio Núñez Ordóñez, el interés y el apoyo mostrado para llevar a buen término este trabajo.

ÍNDICE

Apéndices

Introducción

El almacenamiento de grandes volúmenes de datos textuales en un ordenador es un proceso relativamente sencillo, el problema se plantea a la hora de manejar y recuperar en un tiempo aceptable la información almacenada; la búsqueda de cadenas es la parte fundamental de este problema.

La búsqueda de coincidencias de cadenas consiste en encontrar las ocurrencias de un modelo o patrón en un texto, siendo tanto el patrón como el texto cadenas tomadas de algún alfabeto. Los algoritmos clásicos —válidos para documentos cortos— [KM77], [BM77] y [HO80], realizan la búsqueda de coincidencias directamente sobre el texto en un tiempo que es función lineal de la longitud del mismo; existen posteriores optimizaciones para ciertos casos, [SU90], [CO91] y [HS91].

En la práctica, también se necesita a menudo analizar situaciones donde los datos no son del todo correctos. Considérese la situación en la que la cadena de entrada contiene errores de sustitución o presenta un cierto número de diferencias con el patrón, [BY89], [TU90] y [UW90], y de todas formas es necesario encontrar las apariciones del patrón en el texto.

Se podría suponer que lo que se proporciona como cadena de búsqueda es precisamente algo parecido a lo que previamente ha sido almacenado en algún registro o registros. La cadena de búsqueda no coincide exactamente porque ha sido modificada por algún proceso de distorsión. Sin embargo, puede ocurrir que una cadena que haya sufrido alguna alteración se parezca mucho e incluso coincida con otra previamente

almacenada; el objetivo de la búsqueda es precisamente la recuperación de estas cadenas.

Diversos han sido los autores que han estudiado el concepto de similitud o distancia entre dos cadenas. Alberga, [AL67], usó matrices binarias para evaluar una medida de la disimilitud entre cadenas. Szanser, [SZ73b], desarrolló un proceso matemático de coincidencia elástica que era efectivo en un 95%. Morgan, [MO70], realizaba un o_exclusivo entre dos cadenas para determinar si había ocurrido un único error simple. Wagner y Fischer, [WF74], desarrollaron un algoritmo de programación dinámica que evalúa la distancia entre dos cadenas, medida como el mínimo coste de la secuencia de operaciones de edición (sustitución, extracción e inserción). Ukkonen, [UK83], ideó una forma de cálculo eficiente de la matriz de diferencias de Levenshtein, [LE66], usando sus diagonales. Landau y Vishkin, [LV85a], [LV85b], [LV86a] y [LV86b]; Galil y Giancarlo, [GG86], y el propio Ukkonen, [UK85], utilizan en sus trabajos, con las adaptaciones necesarias, la computación eficiente de la distancia de edición introducida en [UK83]. Tanaka y Kojima, [TK87], continúan utilizando la introducida por Wagner y Fischer, [WF74].

Si se plantea el hecho de corregir los errores introducidos por la distorsión, mediante el proceso de recuperación se obtiene una tentativa de corrección de errores, y con la recuperación de una cadena no relevante se obtiene un error. Este punto de vista detección/corrección se adopta en teoría de comunicaciones y reconocimiento de patrones.

Los métodos de corrección de cadenas se pueden clasificar en dos categorías: los métodos estadísticos que usan N-gramas, probabilidades de confusión y de

ocurrencia; y los métodos de diccionarios que están basados en similitudes o distancias. En general, los métodos estadísticos son más rápidos que los que usan diccionarios, mientras que los métodos de diccionarios pueden obtener mejores tasas de corrección que los estadísticos.

El algoritmo de Viterbi clásico, [FO73], y sus modificaciones, [NE75], [ST79] y [HS82], han sido sugeridos para corregir solamente errores de sustitución. La característica principal del algoritmo de Viterbi es que usa un modelo de Markov del diccionario de entrada, caracterizado por sus unigrama y digrama de frecuencias. La gran desventaja de estos algoritmos radica en un excesivo cálculo, que es independiente del tamaño del diccionario. La bondad de la corrección tan sólo alcanza valores en torno al 50%, [NE75]. Es bien sabido que, para mejorar la realización, es necesario usar el diccionario y no únicamente su modelo estadístico. Kashyap y Oomen, [KO85], proponen una mejora para la corrección de cadenas de longitud pequeña.

La principal dificultad de los métodos que usan la distancia de Levenshtein entre dos cadenas, con el objeto de elegir las cadenas más parecidas a la de búsqueda de entre las que forman el diccionario, radica en el hecho de que se ha de comparar cada cadena de búsqueda con la base de datos entera, o gran parte de ella, [OT76] y [TK87]. Existen algunos métodos de corrección, [RE71], [RH74] y [UL75], que utilizan el digrama o trigrama de frecuencias. Sin embargo, se pone de manifiesto, [OT76], que tales métodos son inferiores a los que utilizan la distancia de Levenshtein.

Recientemente, los investigadores han tomado un creciente interés por los métodos rápidos de corrección o búsqueda para vocabularios grandes. Una línea es encontrar métodos estadísticos con alta tasa de corrección: Kurita y Aizawa, [KA84],

estudian un método rápido para corregir sólo errores de sustitución; Shinghal, [SH83], obtiene una tasa de corrección ligeramente menor pero resulta casi diez veces más rápido que el método de diccionario. Otra línea consiste en aumentar la velocidad de los métodos de diccionario: Itahashi y Yokoyama, [IY84], logran una mejora basada en la selección de algunos fonemas; Tanaka, [TK86], presenta otro método de corrección para errores de sustitución, [TT86], posteriormente extendido a errores de inserción y extracción; Kaneko y Dixon, [KD83], discuten un método de decisión jerárquica para un vocabulario grande en un reconocedor de voz; Hall y Dowling, [HD80], e Ito y Kizawa, [IK82], también discuten técnicas de corrección de errores en ristras para ficheros grandes y organizados jerárquicamente. Tanaka y Kojima, [TK87], proponen un método multietápico de alta velocidad para la corrección de ristras de fonemas, basado en la distancia de Levenshtein, usando ficheros jerarquizados.

Cada cadena puede considerarse como un punto en el espacio multidimensional de secuencias de caracteres, donde no todas las secuencias de caracteres son posibles. Los métodos para la recuperación de datos multidimensionales permiten llevar a cabo búsquedas asociativas, del tipo que interesan a este trabajo, teniendo que acceder tan sólo a una porción reducida de la base de datos en cuestión. Los algoritmos más conocidos se encuentran en los trabajos de Bentley y Finkel, [FB74], [BE75], [FB77], [BE79] y [BE85], quienes proponen los árboles Quad y los K-D que son estructuras basadas en la comparación de claves; y Burkhard y Keller, [BK73], que presentan la estructura BK que se organiza a partir de las distancias.

Las bases de datos documentales requieren métodos de búsqueda más rápidos que los mencionados inicialmente —tiempo lineal respecto de la longitud del texto. Para

ello se someten los escritos a un tratamiento previo, al objeto de construir una estructura auxiliar que aumente la velocidad de las ulteriores consultas. Estos métodos de búsqueda en texto, [FA85] y [FB92], pueden clasificarse en tres categorías: índices lexicográficos, [GO87] y [MM90]; índices basados en dispersiones, [HA71], [RI77], [LA83], [FC84] y [FA88]; y técnicas de agrupamiento, [SS85] y [WI88].

Este trabajo trata aspectos teórico-prácticos en torno al problema de la búsqueda de las cadenas más similares a una dada en grandes volúmenes de datos. El concepto de similitud se entiende en el sentido de la distancia de Levenshtein, **DL**. Dado un diccionario de cadenas y una distancia en el espacio de las mismas, las cadenas más similares a una dada —que puede encontrarse o no en el diccionario— son todas las del diccionario que se encuentran a distancia mínima de la proporcionada. Se construye como índice una estructura de datos que evita el recorrido secuencial del diccionario. El objetivo que se persigue es la optimización de los recursos de tiempo y espacio, de los esquemas de búsqueda y de la estructura de datos que los soporta.

Planteados los antecedentes y el marco en el que se sitúa el presente trabajo se pueden concretar y resumir los capítulos de la siguiente forma:

En el Capítulo 1, se define la distancia de Levenshtein, [LE66], con el cálculo propuesto por Wagner y Fisher, [WF74], y la optimización de Ukkonen, [UK83]; asimismo se introduce una nueva medida que se ha denominado distancia invariante trasposicional, **DIT**, debido al hecho de que su valor no depende de las operaciones de trasposición a que pueda ser sometida una cadena, [SD87], y cumple las propiedades de distancia entre cadenas sin considerar la disposición secuencial de sus caracteres. Si bien **DIT** no puede usarse por sí sola para la determinación de las cadenas más

similares en el sentido de Levenshtein, su importancia deviene de la circunstancia de que su valor entre dos cadenas es siempre inferior o igual a la **DL** entre estas dos mismas cadenas, siendo su coste computacional sensiblemente inferior; tales propiedades aconsejan la construcción de un filtro adaptativo **DIT | DL** que tenga por misión reducir el número de cadenas de la base de datos a las que se les calcula la **DL** con la cadena de búsqueda.

En el Capítulo 2, se diseña una estructura cuyo propósito es llevar a cabo una compartición de las componentes de **DIT**, a fin de no tener que calcular completamente la **DIT** de la cadena de búsqueda a todas y cada una de las cadenas del diccionario. Esta estructura fue diseñada por Santana y Díaz, [SD87] y [SD89], y se denomina estructura **S-D**. Se estudian criterios para la reducción de la longitud del camino promedio, [SD87]. Se comprueba la evolución del comportamiento de esta estructura con los tamaños de los diccionarios. El esquema de búsqueda que se apoya en la estructura **S-D**, recorriéndola a través de las componentes de **DIT** y que usa este valor como criterio de poda, se denomina esquema *decreciente*, [SD87]. Se estudia un nuevo esquema de búsqueda denominado *creciente*, [DS90] y [SP90], donde el radio de búsqueda, en oposición a la evolución clásica decreciente, sigue una línea de modificación creciente. Además, se propone un esquema *decreciente* con radio ascendente tal que en función del incremento del radio de búsqueda define una *familia de esquemas* intermedios que conectan a los esquemas *creciente* y *decreciente*, [DS90]. Se presentan los resultados de los experimentos realizados con los diferentes esquemas.

Prolongando la línea de optimización de las realizaciones de los esquemas de búsqueda *decreciente* y *creciente* de las cadenas más similares a una dada en la estructura **S-D**, en el Capítulo 3, se recomienda una nueva poda en el índice que acorta su recorrido, [SR90], comprobándose experimentalmente su eficacia. Se define un nuevo filtro, **DS**, en función de la subsecuencia común más larga entre cadenas, [SR90], que recoge propiedades ignoradas por **DIT**; se demuestra que es una distancia, [DS93], que cumple $DIT \leq 2*DS \leq 2*DL$ y que tiene un menor costo computacional que **DL**. Se verifica experimentalmente que dicho filtro es útil para descartar un número de cadenas a las que no es necesario evaluar su **DL** con la de búsqueda.

Si el tamaño de la estructura **S-D** es tal que no es posible ubicarla en memoria interna; en el Capítulo 4, se plantea el problema de su paginación con el fin de minimizar el número de accesos a disco. Los criterios a tener en cuenta son: el tiempo de respuesta a las peticiones y la ocupación. No se consideran los problemas relativos al mantenimiento —inserciones, extracciones y reorganizaciones— debido al carácter estático del diccionario. Un primer intento para resolver el problema consiste en llenar las páginas con los nodos al recorrer la estructura en *preorden* o en *postorden*. También se propone una nueva forma de paginar la estructura, *segunbusca*, de tal modo que se respete el siguiente principio: durante la búsqueda, una vez que se abandone una página no se vuelve a acceder a ella por otro camino. Todas las propuestas se ensayan experimentalmente.

En el Capítulo 5, se aplica la estructura **S-D** y los esquemas de búsqueda estudiados aportando una solución a la localización de palabras en texto libre; ello constituye una parte fundamental de un problema práctico de gran importancia: la

organización y utilización de información procedente de fuentes heterogéneas. Los escritos se someten a un tratamiento previo; la intención es generar un índice, **S-D**, que haga viable los accesos posteriores al ejemplar cuando sea necesario. El corpus que se indiza, como ejemplo, para llevar a cabo las localizaciones textuales es un Diccionario de Medicina, [JV87]. Se permiten los siguientes tipos de búsquedas: *exacta*, *más similares*, con *operadores booleanos*, *máscaras*, *truncamientos*, *cercanía*, *antecedencia*, *párrafos*, *sentencias*, *frases* y *compleja*, [SD92]. Además se construye un analizador sintáctico, [SD92], que cumple un doble cometido, ya que ha de distinguir cada tipo de búsqueda; en el caso de las *complejas*, diferenciar las componentes y los *conectores lógicos* y, a la vez, determinar la corrección sintáctica de cualquier petición. Como complemento al analizador, se dispone de un optimizador con el fin de obtener la solución de una búsqueda *compleja* en el menor tiempo posible. Por último, para hacer la aplicación más accesible a un mayor número de usuarios, se implanta en un ordenador personal utilizando la paginación que ha resultado más adecuada para el índice. Se realizan estudios experimentales que ilustran la aplicación.

Las conclusiones y principales aportaciones se recogen en el Capítulo 6.

Capítulo 1.
Distancias entre Cadenas

Sea **X** una cadena de caracteres sobre un alfabeto $\{\alpha_1,\ldots,\alpha_m\}$, y sea μ la cadena nula —sin caracteres. Se denota por $X<i>$ el carácter que está en la **i**-ésima posición de la cadena **X**, $X<i,j>$ la secuencia de caracteres que va de $X<i>$ a $X<j>$ ambos inclusive; si $i>j$ entonces $X<i,j>=\mu$. $|X|$ denota la longitud de la cadena **X**.

Una operación de edición es el par $(\Phi,\Omega)\neq(\mu,\mu)$ donde Φ y Ω son cadenas de longitud menor o igual que uno, es decir, o son μ o son un único carácter. La cadena **Y** resulta de aplicar (Φ,Ω) a la cadena **X** y se denota por $X\rightarrow Y$, si $X=\sigma\Phi\pi$ e $Y=\sigma\Omega\pi$, donde σ y π son cadenas de longitud mayor o igual que cero.

Wagner y Fischer, [WF74], consideran tres tipos de operaciones de edición. (Φ,Ω) es una operación de *sustitución* si $\Phi\neq\mu$, $\Omega\neq\mu$ y $\Phi\neq\Omega$; es una operación de *extracción* si $\Omega=\mu$ y es una operación de *inserción* si $\Phi=\mu$.

Sea **S** una secuencia $S_1, S_2,\ldots, S_n$ de operaciones de edición. Una *derivación_S* de **X** hasta **Y** es una secuencia de cadenas $X_0, X_1,\ldots, X_n$ tal que $X=X_0$, $Y=X_n$ y $X_{j-1}\rightarrow X_j$ vía S_j $\forall$ $j=1,\ldots,n$.

1.1. Distancia de Levenshtein, DL.

Sea Γ una función arbitraria de costo que asigna a cada operación de edición (Φ,Ω) un número real no negativo $\Gamma(\Phi,\Omega)$. Se puede extender Γ a la secuencia **S** definiendo:

$$\Gamma(S)=\begin{cases}\sum_{j=1}^{n}\Gamma(S_j) & si\ n\geq 1\\ 0 & si\ n=0\end{cases}$$

Definición: *Se llama distancia de Levenshtein, DL(X,Y), entre las cadenas X e Y, al mínimo costo de todas las secuencias de edición que transformen X en Y. Formalmente* $DL(X,Y)=min\{\Gamma(S)\}$.

DL, también conocida como distancia de edición, fue introducida por Levenshtein, [LE66], y evaluada por Wagner y Fischer, [WF74], a través de un algoritmo de programación dinámica.

En este trabajo se sigue el criterio de que el costo de cualquier operación de edición se considera *unitario*.

1.2. Cálculo de Wagner y Fischer de la Distancia de Levenshtein.

Si **X** e **Y** son dos cadenas cualesquiera, se define $X(i)=X<1,i>$, $Y(j)=Y<1,j>$ y $WF(i,j)=DL(X(i),Y(j))$.

$$WF(0,0)=0$$

$$WF(i,0)=\sum_{r=1}^{i}\Gamma(X<r>,\mu)$$

$$WF(0,j)=\sum_{r=1}^{j}\Gamma(\mu,Y<r>)$$

Es decir, el costo de convertir μ en sí mismo es cero, el costo de reducir la cadena **X** a μ es la suma de los costos de extraer todos los caracteres de **X** y el costo de obtener **Y** a partir de μ es la suma de los costos de añadir cada uno de los caracteres de **Y**.

Para calcular $WF(i,j)$ $i=1..\ |X|$, $j=1..\ |Y|$ hay que tener en cuenta tres casos.

1.- Convertir $X(i-1)$ en $Y(j-1)$ y $X<i>$ en $Y<j>$.

2.- Convertir $X(i-1)$ en $Y(j)$ y extraer $X<i>$.

3.- Convertir $X(i)$ en $Y(j-1)$ y añadir $Y<j>$.

De estas tres posibilidades se escoge aquella cuyo costo sea mínimo.

Por lo tanto:

$$WF(i,j) = \min\{\ WF(i-1,j-1) + \Gamma(X<i>, Y<j>),\ WF(i-1,j) + \Gamma(X<i>, \mu),\ WF(i,j-1) + \Gamma(\mu, Y<j>)\ \}$$

Y la distancia de Levenshtein entre **X** e **Y** viene dada por $WF(|X|, |Y|)$, es decir, $DL(X,Y) = WF(|X|, |Y|)$.

Algoritmo de Wagner y Fischer para el cálculo de la distancia de Levenshtein:

```
Procedimiento CWF_DL (X,Y)
WF(0,0) = 0
para i = 1 hasta | X | hacer
    WF(i,0) = WF(i-1,0) + Γ(X<i>, μ)
fin para
para j = 1 hasta | Y | hacer
    WF(0,j) = WF(0,j-1) + Γ(μ, Y<j>)
fin para
para i = 1 hasta | X | hacer
    para j = 1 hasta | Y | hacer
        m1 = WF(i-1,j-1) + Γ(X<i>, Y<j>)
        m2 = WF(i-1,j) + Γ(X<i>, μ)
        m3 = WF(i,j-1) + Γ(μ, Y<j>)
        WF(i,j) = min(m1,m2,m3)
    fin para
fin para
```

Ejemplo:

Si *X=trabajo* e *Y=pasajero*, la matriz **WF** es la siguiente:

WF			p	a	s	a	j	e	r	o
		0	1	2	3	4	5	6	7	8
	0	0	1	2	3	4	5	6	7	8
t	1	1	1	2	3	4	5	6	7	8
r	2	2	2	2	3	4	5	6	6	7
a	3	3	3	2	3	3	4	5	6	7
b	4	4	4	3	3	4	4	5	6	7
a	5	5	5	4	4	3	4	5	6	7
j	6	6	6	5	5	4	3	4	5	6
o	7	7	7	6	6	5	4	4	5	5

Por tanto: $DL(X,Y)=5$.

Análisis del algoritmo:

La cantidad total de tiempo usada por este algoritmo es proporcional al número de asignaciones ejecutadas (excluyendo las implícitas de los bucles *para*). Este número es exactamente $1+|X|+|Y|+4*|X|*|Y|$, por tanto el tiempo total es $O(|X|*|Y|)$. ◆

1.3. Cálculo Optimizado de la Distancia de Levenshtein.

El cálculo de **DL** fue optimizado por Ukkonen en [UK83], y es posteriormente utilizado por el propio Ukkonen en [UK85], Galil y Giancarlo en [GG86] y Landau y Vishkin en [LV85a], [LV85b], [LV86a] y [LV86b], así como en el presente trabajo.

Este cálculo se realiza utilizando las diagonales de la matriz **WF** de Wagner y Fischer siguiendo a Landau y Vishkin. Una diagonal **d** de la matriz consiste en todos los *WF(i,j)* tales que *i-j=d*.

Dada una distancia **e** y una diagonal **d**, se define:

$$LV(d,e) = máximo\{i/WF(i,j)=e \text{ con } i-j=d\}$$

Esto implica que:

$$e=WF(LV(d,e),LV(d,e)-d) \text{ y } X<LV(d,e)+1> \neq Y<LV(d,e)-d+1>$$

Además si

$$|X| \leq |Y|, \quad df = |X| - |Y| \quad \text{y} \quad LV(df,e) \geq |X|$$

entonces

$$e=WF(|X|,|Y|).$$

A continuación se muestra como se puede calcular, dados **e** y **d**, el valor de *LV(d,e)*. Supóngase que $\forall v<e$ y $\forall g$ ya está calculado *LV(g,v)*.

Si *LV(d,e)=i*, (o sea, *WF(i,j)=e* y *i-j=d*) es porque ocurre alguna de las siguientes posibilidades:

a) (*WF(i-1,j-1)=e-1* y $X<i> \neq Y<j>$) ó *WF(i,j-1)=e-1* ó *WF(i-1,j)=e-1*.

b) *WF(i-1,j-1)=e* y $X<i>=Y<j>$.

Para obtener el valor de *LV(d,e)* es necesario conocer *LV(d-1,e-1)*, *LV(d,e-1)* y *LV(d+1,e-1)*. El cálculo de la matriz **LV** se hará de forma que la fila **df** sea la que primero se calcule, dentro de lo posible, ya que la condición para obtener **DL** es que un valor de *LV(df,e)* supere la longitud de la cadena más corta $|X|$.

Algoritmo para el cálculo optimizado de la distancia de Levenshtein:

Supóngase que $|X| \leq |Y|$

```
Procedimiento COP_DL (X,Y)
LV(0,-1)=-1
para d=1 hasta |X| hacer
   LV(d,d-2)=-1
   LV(d,d-1)=d-1
fin para
para d=1 hasta |Y| hacer
   LV(-d,d-2)=-1
   LV(-d,d-1)=-1
fin para
df=|X|-|Y|
para k=0 hasta |X|/2 hacer
   para td=-1 hasta 0 hacer
      e=k-df+td
      para d=df-k hasta df-1 hacer
         e=e+1
         fila=max{LV(d-1,e-1)+1,LV(d,e-1)+1,LV(d+1,e-1)}
         mientras (fila+1≤|X|) y (X<fila+1>=Y<fila-d+1>) hacer
            fila=fila+1
         fin mientras
         LV(d,e)=fila
      fin para
      e=k+td
      para d=k hasta df paso -1 hacer
         e=e+1
         fila=max{LV(d-1,e-1)+1,LV(d,e-1)+1,LV(d+1,e-1)}
         mientras (fila+1≤|X|) y (X<fila+1>=Y<fila-d+1>) hacer
            fila=fila+1
         fin mientras
         LV(d,e)=fila
      fin para
      si LV(df,e)≥|X| entonces
         WF(|X|,|Y|)=e
         retornar
      fin si
   fin para
fin para
```

Ejemplo:

Sea X=*trabajo* e Y=*pasajero* entonces se tiene que $|X| = 7$, $|Y| = 8$ y, por tanto, $df = -1$. A continuación se muestra de nuevo la matriz **WF**, correspondiente a estas dos cadenas, a la que se le han añadido los valores, **d**, de las diagonales.

WF		p	a	s	a	j	e	r	o	
	0	1	2	3	4	5	6	7	8	d
0	0	1	2	3	4	5	6	7	8	
t 1	1	1	2	3	4	5	6	7	8	-8
r 2	2	2	2	3	4	5	6	6	7	-7
a 3	3	3	2	3	3	4	5	6	7	-6
b 4	4	4	3	3	4	4	5	6	7	-5
a 5	5	5	4	4	3	4	5	6	7	-4
j 6	6	6	5	5	4	3	4	5	6	-3
o 7	7	7	6	6	5	4	4	5	5	-2
d		7	6	5	4	3	2	1	0	-1

Se construye la matriz **LV**, asociada a **X** e **Y**, siguiendo el algoritmo anterior. Para facilitar su comprensión se presenta una traza del mismo.

LV	-1	0	1	2	3	4	5	6	7
-8								-1	-1
-7							-1	-1	
-6						-1	-1		
-5					-1	-1			
-4				-1	-1				
-3			-1	-1	0				
-2		-1	-1	0	1	3			
-1	-1	-1	0	1	3	4	7		
0	-1	0	1	2	3	6			
1	-1	0	1	3	6				
2		-1	1	2					
3			-1	2					
4				-1	3				
5					-1	4			
6						-1	5		
7							-1	6	

k	td	d	e
0	-1	0	0
		-1	1
	0	0	1
		-1	2
1	-1	-2	2
		1	1
		0	2
		-1	3
	0	-2	3
		1	2
		0	3
		-1	4
2	-1	-3	3
		-2	4
		2	2
		1	3
		0	4
		-1	5

Cuando se obtiene $LV(-1,5)=7 \geq |X|$ se detiene el proceso, siendo $WF(|X|, |Y|)=5$ y por tanto $DL(X,Y)=5$.

Análisis del algoritmo:

De la misma forma que en el algoritmo de Wagner y Fischer, se evalúa la cantidad total de tiempo usada realizando el análisis en el peor caso; ello ocurre cuando las dos cadenas no tienen ningún carácter en común:

El número de asignaciones correspondientes a la fase de inicialización es:

$$1+2*|X|+2*|Y|+1=2*|X|+2*|Y|+2 \qquad (1)$$

El número de iteraciones de los dos bucles *para*, en función de la variable **d**, viene dado por la siguiente expresión:

$$(|Y|-|X|+1)+(|Y|-|X|+1)+(|Y|-|X|+3)+$$
$$(|Y|-|X|+3)+\ldots$$

donde cada sumando entre paréntesis con término independiente del mismo valor, es debido a los dos posibles valores de la variable **td**; además dentro de cada uno de los dos bucles se realizan 3 asignaciones y dentro del bucle controlado por **td**, 2 asignaciones de la variable **e**; el número de términos de la expresión en el peor caso es $2*\lfloor |X|/2+1 \rfloor$; por tanto se obtiene:

$$3*[(|Y|-|X|+1)+(|Y|-|X|+1)+(|Y|-|X|+3)+$$
$$+(|Y|-|X|+3)+\overbrace{\ldots\ldots\ldots}^{2*\lfloor |X|/2+1 \rfloor \text{ términos}}]+2*2*\lfloor |X|/2+1 \rfloor =$$
$$=3*(|Y|-|X|)*(2*\lfloor |X|/2+1 \rfloor)+$$
$$+3*(1+1+3+3+5+5+\overbrace{\ldots\ldots\ldots}^{2*\lfloor |X|/2+1 \rfloor \text{ términos}})+4*\lfloor |X|/2+1 \rfloor =$$
$$=3*2*[(|Y|-|X|)*\lfloor |X|/2+1 \rfloor+(1+3+5+\overbrace{\ldots\ldots\ldots}^{\lfloor |X|/2+1 \rfloor \text{ términos}})]+$$
$$+4*\lfloor |X|/2+1 \rfloor =$$
$$=6[(|Y|-|X|)*\lfloor |X|/2+1 \rfloor+\lfloor |X|/2+1 \rfloor^2]+4*\lfloor |X|/2+1 \rfloor$$

añadiendo a esta expresión la representada en (1):

$$2*|X|+2*|Y|+2+6[(|Y|-|X|)*\lfloor |X|/2+1 \rfloor+\lfloor |X|/2+1 \rfloor^2]+$$
$$+4*\lfloor |X|/2+1 \rfloor =$$
$$=6*|Y|*\lfloor |X|/2 \rfloor-6*|X|*\lfloor |X|/2 \rfloor+6*\lfloor |X|/2 \rfloor^2+$$
$$+8*|Y|-4*|X|+16*\lfloor |X|/2 \rfloor+12$$

si $|X|$ es par se obtiene:

$$3*|Y|*|X|-3/2*|X|^2+8*|Y|+4*|X|+12$$

si $|X|$ es impar se obtiene:

$$6*|Y|*(|X|/2-1/2)-6*|X|*(|X|/2-1/2)+6*(|X|/2-1/2)^2+$$
$$+8*|Y|-4*|X|+16*(|X|/2-1/2)+12=$$
$$=3*|Y|*|X|-3/2*|X|^2+5*|Y|+4*|X|+11/2$$

por tanto, en cualquier caso, el tiempo total es $O(|Y|*|X|-|X|^2/2)$. ◆

1.4. Distancia Invariante Trasposicional, DIT.

El alto costo computacional de la distancia de Levenshtein, en contraste con sus prestaciones para encontrar cadenas similares, aconseja introducir una medida menos costosa que seleccione previamente las cadenas a las que se ha de calcular **DL**. La distancia invariante trasposicional entre dos cadenas **X** e **Y**, *DIT(X,Y)*, se define en función del número de veces que está presente cada carácter en ambas cadenas.

Definición: *Dadas dos cadenas X e Y se define*

$$DIT(X,Y)=\sum_{i=1}^{m} abs(X_{\alpha i}-Y_{\alpha i})+abs(|X|-|Y|)$$

donde $X_{\alpha i}$ *e* $Y_{\alpha i}$ *son las frecuencias de aparición del carácter* α_i *en* **X** *y en* **Y** *respectivamente, y* **m** *el tamaño del alfabeto.*

Como se desprende inmediatamente de esta definición, el valor de **DIT** entre dos cadenas de caracteres no varía si se altera el orden secuencial de los mismos, por esta razón se ha denominado distancia invariante trasposicional.

Ejemplo:

Sea X=*trabajo* e Y=*pasajero*, se tiene que:

$$X_a=2,\ X_b=1,\ X_j=1,\ X_o=1,\ X_r=1,\ X_t=1$$

$$Y_a=2,\ Y_e=1,\ Y_j=1,\ Y_o=1,\ Y_p=1,\ Y_r=1,\ Y_s=1$$

por tanto:

$$DIT(X,Y)=5+1=6.$$

Teorema: *DIT es una semidistancia en el conjunto de cadenas de caracteres sobre un alfabeto dado.*

Demostración:

a) $\forall\ X,\ Y$: *Si* $X=Y \Rightarrow DIT(X,Y)=0$

Por la propia definición.

b) $\forall\ X,\ Y$: $DIT(X,Y)=DIT(Y,X)$

Por la propia definición.

c) $\forall\ X,\ Y,\ Z$: $DIT(X,Y)+DIT(Y,Z) \geq DIT(X,Z)$

Por la desigualdad triangular para el valor absoluto:

$$DIT(X,Y) + DIT(Y,Z) = \sum_{i=1}^{m} abs(X_{ci}-Y_{ci}) + abs(\,|\,X\,|\,-\,|\,Y\,|\,) +$$

$$+ \sum_{i=1}^{m} abs(Y_{ci}-Z_{ci}) + abs(\,|\,Y\,|\,-\,|\,Z\,|\,) =$$

$$= \sum_{i=1}^{m} [abs(X_{ci}-Y_{ci}) + abs(Y_{ci}-Z_{ci})] +$$

$$+ abs(\,|\,X\,|\,-\,|\,Y\,|\,) + abs(\,|\,Y\,|\,-\,|\,Z\,|\,) \geq$$

$$\geq \sum_{i=1}^{m} abs(X_{ci}-Z_{ci}) + abs(\,|\,X\,|\,-\,|\,Z\,|\,) = DIT(X,Z).\ \blacksquare$$

Ya que **DIT** es una semidistancia sobre el conjunto de cadenas, se tiene que es una distancia en el conjunto cociente dado por el conjunto de cadenas y la relación de equivalencia inducida; dicho conjunto cociente es el conjunto de cadenas de caracteres sobre un alfabeto prescindiendo del orden entre los mismos. Cada clase de equivalencia estará formada por todas aquellas cadenas que tengan los mismos caracteres aunque se encuentren en diferente orden; este tipo de cadenas se denomina *sinónimos_DIT*.

Análisis del cálculo de DIT:

Se realizará el análisis suponiendo que las frecuencias de aparición de los caracteres en cada uno de las cadenas se encuentran ya calculadas, porque así sucede en los esquemas de búsqueda que posteriormente se expondrán. La cantidad total de tiempo usada en el cálculo de **DIT** es proporcional al número de valores absolutos que se calculan. Dicho número en el peor caso —ocurre cuando ambas cadenas **X** e **Y** no

tienen caracteres comunes y además en cada cadena no hay caracteres repetidos— es $|X| + |Y| + 1$; debe tenerse en cuenta que para aquellos **i** tales que $X_{\alpha i}=0$ e $Y_{\alpha i}=0$ no es necesario calcular el valor absoluto correspondiente —véanse los apartados del Capítulo 2 correspondientes a los esquemas de búsqueda—; por tanto la complejidad de cálculo es $O(|X| + |Y|)$. ◆

Teorema: *Dadas dos cadenas cualesquiera X e Y, se verifica:*

$$DIT(X,Y) \leq 2*DL(X,Y).$$

Demostración:

Supóngase que dadas dos cadenas **X** e **Y** con $|X| \leq |Y|$, tienen **i** caracteres coincidentes —en cuanto a su frecuencia de aparición— independientemente de su posición, los restantes caracteres de **X**, $j = |X| - i$, son todos diferentes a los que restan en **Y** y sea $k = |Y| - |X|$. Se puede decir que existen **i** caracteres comunes, **j** que han sido sustituidos por otros y **k** que han sido añadidos a **X** (o eliminados de **Y**). Bajo estas condiciones se puede afirmar que:

$$DIT(X,Y) = 2*j+k + k = 2*(j+k) \qquad (1)$$

Nótese que **k** es como mínimo el número de *inserciones* sobre **X** (o de *extracciones* sobre **Y**) y que no todos los **i** caracteres coincidentes han de ser necesariamente considerados como tales en el cálculo de **DL** (por ejemplo una trasposición para **DIT** son caracteres coincidentes y para **DL** son dos sustituciones). Puede suponerse que los **i** caracteres comunes están situados al principio en ambas cadenas, a continuación los **j** caracteres en cada una y por último, en **Y**, los **k** restantes.

Es decir: $X<1,i> = Y<1,i>$, los caracteres de $X<i+1,i+j>$ y de $Y<i+1,i+j>$ son los **j** distintos y los caracteres de $Y<i+j+1,i+j+k>$ son los **k** restantes en **Y**.

Considerando la *derivación_S* de **X** a **Y** utilizada al calcular **DL**, sean: **i'** el número de caracteres coincidentes, $i' \leq i$, **j'** el número de sustituciones y **k'** el número de inserciones más extracciones, $k' \geq k$. Y sea $\mathbf{j_1}$ el número de sustituciones, que **DL** admite como tales, de entre las **j** anteriormente citadas, $j' \geq j_1$.

Se pueden reordenar los caracteres de **X** e **Y**, sin pérdida de generalidad en cuanto a la demostración se refiere, partiendo de la ordenación considerada anteriormente, de tal forma que: $X<1,i'> = Y<1,i'>$ y que los caracteres de $X<i+1,i+j_1>$ y de $Y<i+1,i+j_1>$ forman las $\mathbf{j_1}$ sustituciones.

Las **j'** sustituciones se forman además a partir de:

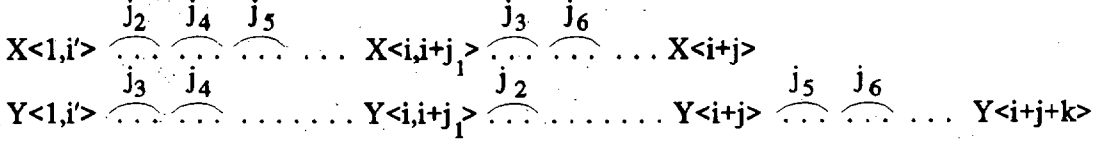
$\mathbf{j_2}$ caracteres de $X<i'+1,i>$ y de $Y<i+j_1+1,i+j>$

$\mathbf{j_3}$ caracteres de $X<i+j_1+1,i+j>$ y de $Y<i'+1,i>$

$\mathbf{j_4}$ caracteres de $X<i'+j_2+1,i>$ y de $Y<i'+j_3+1,i>$

$\mathbf{j_5}$ caracteres de $X<i'+j_2+j_4+1,i>$ y de $Y<i+j+1,i+j+k>$

$\mathbf{j_6}$ caracteres de $X<i+j_1+j_3+1,i+j>$ y de $Y<i+j+j_5+1,i+j+k>$.



entonces:

$$j' = j_1+j_2+j_3+j_4+j_5+j_6$$

y

$$k' = i-(i'+j_2+j_4+j_5)+i-(i'+j_3+j_4)+j-(j_1+j_3+j_6)+j-(j_1+j_2)+k-(j_5+j_6) =$$
$$= 2*i-2*i'+2*j-2*j_1-2*j_2-2*j_3-2*j_4-2*j_5-2*j_6+k = 2*(i-i')+2*(j-j')+k$$

Dado que:

$$DL(X,Y) = j'+k'$$

se tiene que:

$$DL(X,Y) = 2*(i-i')+2*j-j'+k$$

por tanto, teniendo en cuenta (1), se obtiene:

$$DL(X,Y) = DIT(X,Y)/2+2*(i-i')+j-j'$$

como

$$2*(i-i')+j-j' = 2*(i-i')+j-j_1-j_2-j_3-j_4-j_5-j_6$$

y puesto que

$$j-j_1-j_6 \geq 0$$

y

$$2*(i-i') \geq j_2+j_3+j_4+j_5$$

se cumple que

$$2*(i-i')+j-j' \geq 0$$

de ahí que

$$DIT(X,Y) \leq 2*DL(X,Y). \blacksquare$$

Capítulo 2.
Estructura de Santana y Díaz, S-D, y Esquemas de Búsqueda de las Más Similares

Se considera como diccionario, **D**, un determinado conjunto de diferentes cadenas de caracteres sobre cierto alfabeto. Con el fin de reducir el tiempo de respuesta frente a una búsqueda secuencial, el problema de la recuperación en el diccionario de las cadenas más similares a una cadena de búsqueda dada —pudiendo ésta pertenecer o no al mismo— requiere una estructura de almacenamiento del diccionario y un esquema de búsqueda eficiente. Las frecuencias de los caracteres de las palabras de igual longitud se organizan para permitir conjuntamente con el esquema apropiado el abandono de la exhaustividad en la búsqueda; ello conlleva el que sólo se trate una parte del diccionario.

2.1. Estructura de Santana y Díaz, S-D.

El diccionario se organiza como un árbol en el cual se estructuran las componentes que intervienen en el cálculo de **DIT**, como se muestra en la figura 1.

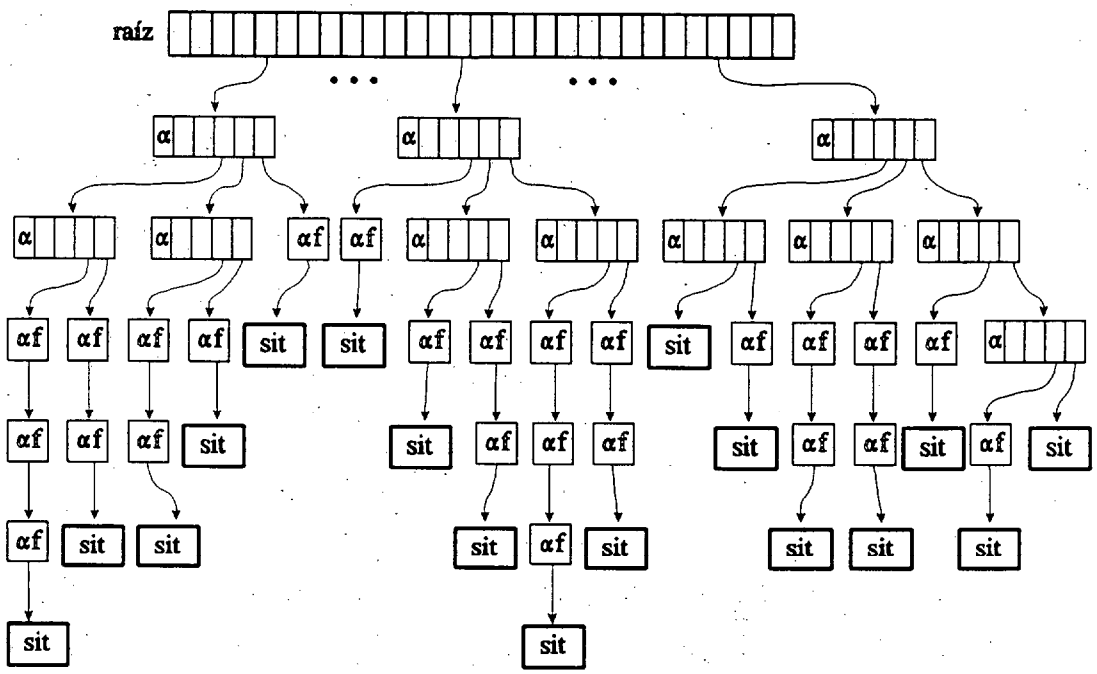


Figura 1

El árbol tiene un nodo *raíz* que discrimina por longitudes de cadenas. El encadenamiento asociado a cada longitud señala a la *parte_árbol* constituida por nodos discriminantes de la forma:

α	p	u	e_p	...	e_i	...	e_u

donde α es el carácter discriminante; e_i es el enlace que apunta al subconjunto de cadenas $D^i \subset D$ tal que la frecuencia del carácter α en cada una de las cadenas de $\mathbf{D^i}$ es **i**; **p** y **u** verifican:

$$0 \leq p < u$$

donde e_p y e_u son el primer y último encadenamiento no-nulo, aunque puede existir algún $i \in \{p+1,\ldots,u-1\}$ tal que $e_i=nulo$.

Cuando una rama de la *parte_árbol* ya no discrimina se pasa a la *parte_cadena* que son listas encadenadas formadas por los restantes pares α**f**, cuyos nodos son:

α	f	e

donde α es un carácter, **f** es la frecuencia con que aparece el carácter α en los *sinónimos_DIT* correspondientes y **e** es un enlace que direcciona a un nodo del mismo tipo o, al acabar la *parte_cadena*, a un *nodo_SIT* de la forma:

ns	C_1	...	C_i	...	C_{ns}

donde se encuentran las **ns** cadenas *sinónimas_DIT*, $\mathbf{C_1},\ldots, \mathbf{C_i},\ldots, \mathbf{C_{ns}}$.

Algoritmo de construcción de la estructura S-D:

```
Procedimiento Construye (raiz)
    raiz(.): contiene los punteros que señalan a la parte_árbol asociada a cada longitud
    SD(.): contiene los nombres de los subdiccionarios correspondientes a cada
    longitud
    longmin: mínima longitud de cadena en el diccionario
    longmax: máxima longitud de cadena en el diccionario
para long = longmin hasta longmax hacer
    raiz(long) = Construye_parte_árbol(SD(long),0)
fin para


Función Construye_parte_árbol (subdiccionario,nct)
    nodo: nodo de la parte_árbol, consta de los siguientes campos:
        cd: carácter discriminante
        pp: primera frecuencia asociada a un encadenamiemto no nulo
        pu: última frecuencia asociada a un encadenamiento no nulo
        e(.): contiene los enlaces a otros nodos
    nct: indica el número de caracteres tratados
si tamaño(subdiccionario) = 0 entonces
    devolver(nulo)
si no
    si tamaño(subdiccionario) = 1 entonces
        devolver(Construye_parte_cadena(subdiccionario,nct))
    si no
        cdisc = Elige_cdisc(subdiccionario,p,u)
        {la función Elige_cdisc selecciona el carácter discriminante en el conjunto de
        caracteres no tratados y devuelve p y u}
        si p = u entonces
            devolver(Construye_parte_cadena(subdiccionario,nct))
        si no
            {se divide el subdiccionario según la frecuencia del carácter discriminante}
            para j = 1 hasta tamaño(subdiccionario) hacer
                f = frecuencia de cdisc en la j-ésima cadena de subdiccionario
                añadir la j-ésima cadena de subdiccionario a subdiccionario_f
            fin para
            crear un nodo de la parte_árbol, nodopa
            nodopa.cd = cdisc
            nodopa.pp = p
            nodopa.pu = u
            para i = p hasta u hacer
                nodopa.e(i) = Construye_parte_árbol(subdiccionario_i,nct + i)
            fin para
            devolver(nodopa)
        fin si
    fin si
fin si
```

```
Función Construye_parte_cadena (subdiccionario,nct)
    nodo: nodo de la parte_cadena, consta de los siguientes campos:
        c: carácter
        f: frecuencia
        e: enlace a otro nodo de la parte_cadena o a un nodo_SIT
si nct < long entonces
    crear un nodo de la parte_cadena, nodopc
    nodopc.c = carácter no tratado de las cadenas de subdiccionario
    nodopc.f = frecuencia de ese carácter
    nct = nct + nodopc.f
    pnodoc = nodopc
    mientras nct < long hacer
        crear un nodo de la parte_cadena, nodo
        nodo.c = carácter no tratado de las cadenas de subdiccionario
        nodo.f = frecuencia de ese carácter
        nct = nct + nodo.f
        nodopc.e = nodo
        nodopc = nodo
    fin mientras
    nodopc.e = Construye_nodo_SIT(subdiccionario)
    devolver(pnodoc)
si no
    devolver(Construye_nodo_SIT(subdiccionario))
fin si


Función Construye_nodo_SIT (subdiccionario)
    nodo: nodo_SIT, consta de los siguientes campos:
        ns: número de sinónimos_DIT
        s(.): cadenas sinónimas_DIT
crear un nodo_SIT, nodo
nodo.ns = tamaño(subdiccionario)
para i = 1 hasta nodo.ns hacer
    nodo.s(i) = i-ésima cadena de subdiccionario
fin para
devolver (nodo)
```

2.1.1. Criterios de Selección del Carácter Discriminante.

Una primera opción para la elección del carácter discriminante es la que establece el siguiente criterio dinámico: se va creando el árbol según se van insertando las cadenas; cuando es necesario crear un nodo de la *parte_árbol* que separe los

caminos de dos palabras se escoge, por ejemplo, el primer carácter por orden alfabético que establezca una diferencia.

Debido a la naturaleza estática del diccionario, se puede pensar en crear la estructura conociendo a priori qué palabras va a contener. De esta forma se puede escoger en todo momento el carácter que más eficazmente realice la tarea discriminatoria. Se elige el carácter que minimice la suma de los cuadrados de los cardinales de los subdiccionarios que genera. Se ha seguido este criterio ya que selecciona caracteres que reparten uniformemente el número de cadenas en cada nodo incidiendo en la reducción de la longitud del camino promedio del árbol. Para una cadena del diccionario, el promedio del número de cadenas entre las que hay que realizar la búsqueda es mínimo desde cada nodo discriminante de la *parte_árbol*. En efecto, dado un nodo discriminante de la *parte_árbol*, sean $\mathbf{c_p},\ldots,\mathbf{c_u}$ el número de cadenas bajo los encadenamiento $\mathbf{e_p},\ldots,\mathbf{e_u}$ respectivamente, $c=c_p+\ldots+c_u$, y sea **X** una de las cadenas del subdiccionario que penden del nodo. El valor esperado del número de cadenas entre las que hay que realizar la búsqueda de **X** es:

$$\frac{c_p}{c} * c_p + \ldots + \frac{c_u}{c} * c_u = \sum_{i \in [p..u]} \frac{c_i^2}{c}$$

Dado que el numerador es mínimo, debido al criterio de selección del carácter discriminante, es mínimo el valor esperado.

Ejemplo de la estructura S-D:

En la figura 2 se muestra un ejemplo de la estructura **S-D** para longitud cuatro, correspondiente al subdiccionario formado por: *amor*, *bebe*, *coro*, *diva*, *loro*, *meta*, *mora*, *ramo*, *raza*, *roma*, *tiro*, *toro* y *vaya*. Los caracteres de los nodos discriminantes se han seleccionado según el criterio de *mínimos cuadrados* descrito anteriormente.

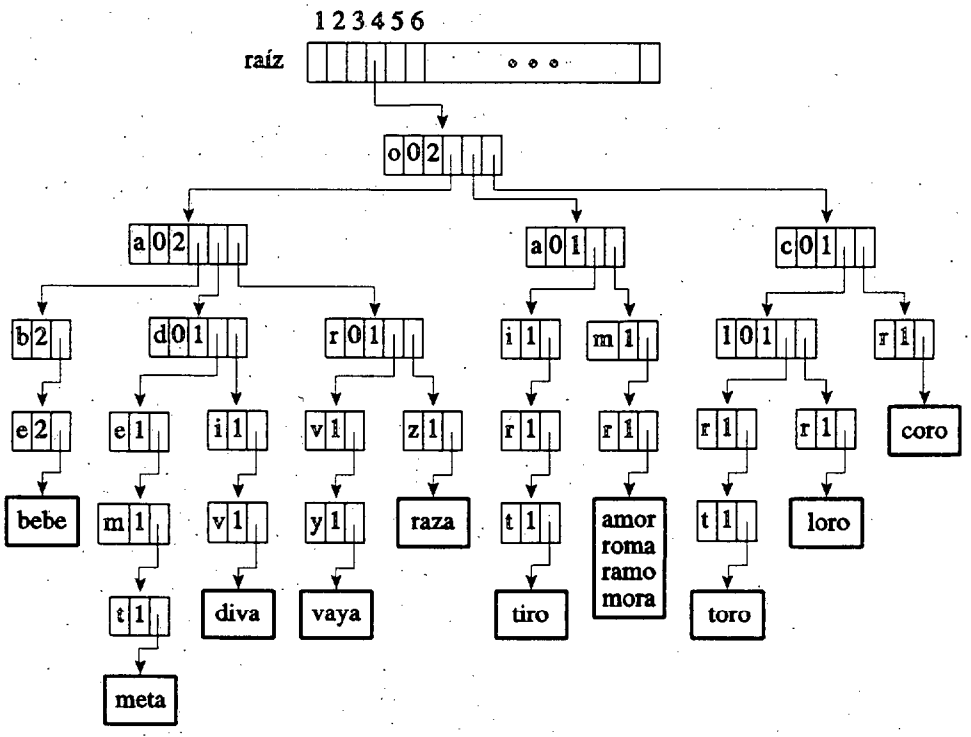


Figura 2

2.1.2. Relación Ocupacional.

Las cadenas situadas en los *nodos_SIT* se pueden considerar como los datos, se entiende que el resto de la estructura **S-D** es el índice.

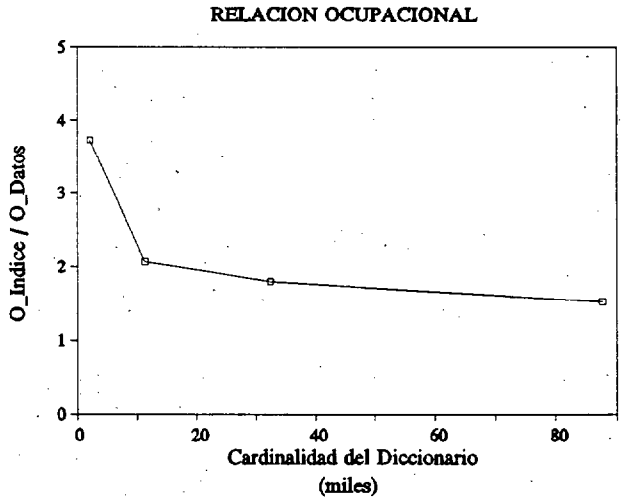


Figura 3

La razón *O_Índice/O_Datos* es el cociente entre la ocupación del índice y la ocupación de los datos expresada en bytes.

Se realizó un experimento, utilizando las palabras de diferentes diccionarios de la lengua española —sin considerar los signos ortográficos— para estudiar la relación entre la razón *O_Índice/O_Datos* y el tamaño del diccionario. Para ello se tiene que α, **p**, **u**, **f** y **ns** ocupan un byte cada uno. $\mathbf{C}_j$ usa $| C_j |$ bytes y cada uno de los enlaces de la estructura rinde cuatro bytes.

Como puede observarse, en la figura 3, la relación ocupacional cae rápidamente para cardinalidades bajas, para aminorar su descenso, manteniéndose a valores razonables, a medida que aumenta la cardinalidad del diccionario.

2.2. Búsqueda de las Cadenas Más Similares.

Definición: *Sea **D** el conjunto de todas las cadenas correctas, es decir, el diccionario, de cardinalidad **NCD**. Sea **B** la cadena de búsqueda, **d** una distancia definida sobre **D** y **SB** un subconjunto de **D** definido como sigue:*

$$SB = \{ X \in D \,/\, d(B,X) = min\{d(B,Y), \forall Y \in D\} \}$$

***SB** se denomina conjunto de cadenas más similares a **B**.*

En este trabajo se estudia el problema de la búsqueda de las cadenas más similares a una dada utilizando la distancia de Levenshtein, **DL**, como medida de similitud. Esta distancia requiere un alto costo computacional, una forma de reducirlo consiste en utilizar **DIT** como filtro de **DL** debido a su menor costo y a la relación existente entre ambas, $DIT(X,Y) \leq 2*DL(X,Y)$.

Las cadenas más similares a una cadena de búsqueda **B** según **DIT**, no son necesariamente las **DL** más similares. Por ejemplo, si **X** difiere de **B** en una trasposición e **Y** difiere de **B** en una inserción se tiene que:

$$DIT(B,X) = 0 < DIT(B,Y) = 1$$

y

$$DL(B,X) = 2 > DL(B,Y) = 1$$

por tanto, **X** es más similar a **B** que **Y** según **DIT** pero no según **DL**.

La estructura **S-D** permite evaluar **DIT** considerando únicamente las frecuencias no nulas. Además, durante el proceso de búsqueda, en el retorno a cada nodo de la

parte_árbol, se dispone del valor acumulado de las componentes de **DIT** de los caracteres situados en el camino desde la raíz hasta dicho nodo.

En un momento del proceso de búsqueda, sea **X** una de las cadenas actuales más similares a **B** e **Y** una cadena del diccionario de la que se quiere estudiar su similitud con **B.** **Y** será considerará como más similar si:

$$DL(B,Y) \leq DL(B,X).$$

Antes de evaluar *DL(B,Y)*, se puede calcular *DIT(B,Y)* para aplicarla como filtro. Si se verifica que

$$DIT(B,Y) > 2*DL(B,X)$$

puesto que

$$DIT(B,Y) \leq 2*DL(B,Y)$$

entonces

$$DL(B,X) < DL(B,Y)$$

y se rechaza **Y** sin necesidad de evaluar **DL**. De aquí que los esquemas de búsqueda se realizan sobre la estructura **S-D** evaluando **DIT** en su recorrido y **DL** sólo en los *nodos_SIT* que permite el filtro.

2.2.1. Esquema de Búsqueda Decreciente.

Se dispone de un umbral, **DTM**, que en todo momento es igual al doble de la distancia de Levenshtein mínima actual, **DLM**, de tal forma que sólo se calcula la **DL** entre la cadena de búsqueda y aquella cadena del diccionario cuya $DIT \leq DTM$, debido a la relación existente entre **DIT** y **DL**. A lo largo del proceso, tanto el radio de

búsqueda, **DLM**, como la respuesta se actualizan cada vez que se obtiene una $DL<DLM$; si $DL=DLM$ se añade a la respuesta la cadena correspondiente. El valor inicial de **DLM** es infinito.

La búsqueda comienza en el nodo *raíz* por la posición correspondiente a la longitud de la cadena de búsqueda, y se van tomando alternativamente las longitudes próximas, situadas a diferencias crecientes de dicha longitud, hasta alcanzar la primera diferencia de longitudes que supere a **DLM**. Basta con explorar estos ramales ya que la diferencia de longitudes indica el número mínimo de inserciones (o de extracciones) y es por tanto una cota inferior de **DL**. En la *parte_árbol* se comienza por el ramal del nodo actual correspondiente a la frecuencia de aparición del carácter en la cadena de búsqueda, y se continúa alternativamente por las frecuencias próximas correspondientes a diferencias crecientes de la frecuencia inicial. El tratamiento de la *parte_cadena* es secuencial. Durante el recorrido por la estructura se evalúa una aproximación progresiva de **DIT**, acumulando los valores de sus componentes. Si, en algún punto de la misma, este valor excede a **DTM** no se prosigue por ese ramal, estableciéndose así un criterio de poda. Para acceder a un *nodo_SIT* se ha de cumplir que $DIT \leq DTM$, en cuyo caso se evalúa **DL** entre la cadena de búsqueda y todas las de ese nodo.

Algoritmo de búsqueda decreciente:

Se invoca desde el programa principal como Decreciente(B, ∞ ,0)

```
Procedimiento Decreciente (B,DLM,nms)
    B: cadena de búsqueda
    DLM: valor de DL mínimo
    nms: número de cadenas más similares
    primero: primer carácter del alfabeto
    ultimo: último carácter del alfabeto
    vf(.): vector de frecuencias de B
    DTM: es el doble de DLM
    cadit: componentes acumuladas de DIT
    lalt: longitud alternativa
    longmax: máxima longitud de cadena en el diccionario
    longmin: mínima longitud de cadena en el diccionario
para c = primero hasta ultimo hacer
    vf(c) = 0
fin para
para i = 1 hasta | B | hacer
    vf(B<i>) = vf(B<i>) + 1
fin para
DTM = ∞
cadit = 0
Busquedadec_pa(raiz( | B | ),cadit, | B | )
i = 1
mientras i ≤ DLM hacer
    cadit = i
    lalt = | B | + i
    si lalt ≤ longmax entonces
        Busquedadec_pa(raiz(lalt),cadit, | B | )
    fin si
    lalt = | B | - i
    si lalt ≥ longmin entonces
        Busquedadec_pa(raiz(lalt),cadit, | B | )
    fin si
    i = i + 1
fin mientras
```

```
Procedimiento Busquedadec_pa (nodo,cadit,cbnt)
{Realiza el recorrido de la parte_árbol en la búsqueda decreciente}
    nodo: nodo actual
    cbnt: número de caracteres de B no tratados
    frec: frecuencia en B del carácter asociado a nodo
    emp: encadenamiento más próximo a frec
    vcadit: valor de las componentes acumuladas de DIT para la alternativa actual
    falt: frecuencia alternativa
si nodo ≠ nulo entonces
    si nodo es un nodo_SIT entonces
        cadit = cadit + cbnt
        si cadit ≤ DTM entonces
            Tratardec_nS(nodo)
        fin si
    si no
        si es un nodo de la parte cadena entonces
            Tratardec_pc(nodo,cadit,cbnt)
        si no  {es un nodo de la parte_árbol}
            frec = vf(nodo.cd)
            cbnt = cbnt-frec
            si (nodo.pp ≤ frec) y (frec ≤ nodo.pu) entonces
                Busquedadec_pa(nodo.e(frec),cadit,cbnt)
            si no
                si frec < nodo.pp entonces
                    emp = nodo.pp
                si no
                    emp = nodo.pu
                fin si
                vcadit = cadit + abs(frec-emp)
                Busquedadec_pa(nodo.e(emp),vcadit,cbnt)
                frec = emp
            fin si
            i = 1
            vcadit = cadit + i
            mientras vcadit ≤ DTM hacer
                falt = frec + i
                si falt ≤ nodo.pu entonces
                    Busquedadec_pa(nodo.e(falt),vcadit,cbnt)
                fin si
                falt = frec-i
                si falt ≥ nodo.pp entonces
                    Busquedadec_pa(nodo.e(falt),vcadit,cbnt)
                fin si
                i = i + 1
                vcadit = cadit + i
            fin mientras
        fin si
    fin si
fin si
```

```
Procedimiento Tratardec_nS (nodo)
{Realiza el tratamiento de los nodos_SIT en la búsqueda decreciente}
   dl: valor de DL entre B y el sinónimo_DIT actual
para j=1 hasta nodo.ns hacer
   dl=COP_DL(B,nodo.s(j))
   si dl < DLM entonces
      DLM=dl
      DTM=2*DLM
      nms=1
      eliminar el conjunto respuesta
      crear un conjunto respuesta formado por nodo.s(j)
      si dl = 0 entonces
         retorno de final de búsqueda
      fin si
   si no
      si dl = DLM entonces
         añadir nodo.s(j) al conjunto respuesta
         nms=nms+1
      fin si
   fin si
fin para
```

```
Procedimiento Tratardec_pc (nodo,cadit,cbnt)
{Realiza el recorrido de la parte_cadena en la búsqueda decreciente}
si nodo es un nodo_SIT entonces
   cadit=cadit+cbnt
   si cadit ≤ DTM entonces
      Tratardec_nS(nodo)
   fin si
si no
   cadit=cadit+abs(nodo.f-vf(nodo.c))
   cbnt=cbnt-vf(nodo.c)
   si cadit ≤ DTM entonces
      Tratardec_pc(nodo.e,cadit,cbnt)
   fin si
fin si
```

2.2.2. Familia de Esquemas.

La diferencia entre el valor inicial y final de **DLM** permite la aceptación de cadenas como similares, muy distantes de la cadena de búsqueda, que no van a estar presentes en la respuesta, prolongando el tiempo de obtención de la misma. ¿Es posible

mejorar la estrategia de búsqueda en función del valor inicial de **DLM**?. El esquema *decreciente* busca en un dominio $\{0, 1, 2,\ldots, +\infty\}$, decreciendo dinámicamente el valor del radio. Una alternativa a esta estrategia sería realizar el mismo esquema de búsqueda pero en un dominio más pequeño. Dado que el valor final de **DLM** estará próximo a cero, se realiza una búsqueda inicial de radio cero y, mientras no se obtenga respuesta, se llevan a cabo sucesivas búsquedas para los consecutivos dominios a partir de cero con un tamaño de dominio previamente fijado. Esto da lugar a una *familia de esquemas* de búsqueda en función del tamaño del dominio, o lo que es lo mismo, del incremento del radio de búsqueda, **IR**.

En cada dominio, se trata de encontrar las cadenas $\mathbf{X}_i$ cuyas $DL(B,X_i)=\tau$, siendo τ el valor más bajo alcanzable que cumple que:

$$\delta < \tau \leq \delta + IR$$

siempre que no existan cadenas $\mathbf{X}_j$ tales que $DL(B,X_j)\leq\delta$; se utiliza un esquema de búsqueda *decreciente* en el que **DLM** se inicializa a $\delta+IR$. Si existe respuesta termina el proceso, en caso contrario se actualiza δ:

$$\delta = \delta + IR$$

y se recomienza la búsqueda desde el nodo *raíz*. Inicialmente $\delta=0$. Evidentemente el incremento del radio de búsqueda permanece constante en cada experiencia.

El esquema *decreciente* se corresponde con el esquema de la *familia* para el cual $IR=\infty$. Si $IR=1$ se obtiene un esquema en el que el radio de búsqueda crece de uno en uno invirtiendo el proceso de actualización del radio del esquema *decreciente*.

Algoritmo de búsqueda de la familia de esquemas:

```
Procedimiento Familia (B,IR)
    B: cadena de búsqueda
    IR: incremento del radio
    nms: cardinalidad de la respuesta
    DLM: valor de DL mínimo
delta = 0
nms = 0
Decreciente(B,0,nms)
si nms = 0 entonces
    repetir
        DLM = delta + IR
        Decreciente(B,DLM,nms)
        delta = delta + IR
    hasta que nms > 0
fin si
```

2.2.3. Esquema de Búsqueda Creciente.

En este esquema se recorre la estructura partiendo del nodo *raíz* y con radio de búsqueda *DLM=0*; si no se encuentra respuesta, se aumenta en una unidad el radio de búsqueda y se comienza la exploración nuevamente desde el nodo *raíz*, repitiendo este proceso hasta que al finalizar un recorrido exista respuesta. En tal caso, se obtiene el conjunto de cadenas más similares y su valor de distancia es **DLM**. Debido a que el radio de búsqueda en cada fase o es menor o es igual que la **DL** mínima, es por lo que el umbral **DTM** permanece constante en dicha fase.

La forma de recorrer el árbol es análoga al esquema de la *familia* con *IR=1*, pero la selección de las alternativas en los nodos discriminantes es diferente. En este último esquema se seleccionaban en vaivén a diferencias crecientes de la posición inicial, para lograr aproximaciones a cadenas más similares que posibiliten un acortamiento de su rango de exploración. En el esquema *creciente* este rango no se

altera en cada fase, porque tampoco lo hace **DTM**, lo que permite una exploración lineal de izquierda a derecha de los subconjuntos que penden del nodo.

Algoritmo de búsqueda creciente:

```
Procedimiento Creciente (B)
    B: cadena de búsqueda
    primero: primer carácter del alfabeto
    ultimo: último carácter del alfabeto
    vf(.): vector de frecuencias de B
    DLM: valor de DL mínimo
    DTM: doble de DLM
    nms: número de cadenas más similares
    linf, lsup: longitud mínima y máxima alcanzadas en la selección de alternativas
    cadit: componentes acumuladas de DIT
    lalt: longitud alternativa
    longmin: mínima longitud de cadena en el diccionario
    longmax: máxima longitud de cadena en el diccionario
para c = primero hasta ultimo hacer
    vf(c) = 0
fin para
para i = 1 hasta | B | hacer
    vf(B<i>) = vf(B<i>) + 1
fin para
linf = | B |
lsup = | B |
DLM = 0
DTM = 0
nms = 0
cadit = 0
Busquedacre_p0(raiz( | B | ))
mientras nms = 0 hacer
    DLM = DLM + 1
    DTM = 2 * DLM
    linf = linf-1
    lsup = lsup + 1
    si linf < longmin entonces linf = longmin fin si
    si lsup > longmax entonces lsup = longmax fin si
    para lalt = linf hasta lsup hacer
        cadit = abs(lalt- | B | )
        Busquedacre_pa(raiz(lalt),cadit, | B | )
    fin para
fin mientras
```

```
Procedimiento Busquedacre_p0 (nodo)
{Comprueba si la cadena de búsqueda se encuentra en el diccionario}
    nodo: nodo actual
    dl: valor de DL entre B y el sinónimo_DIT actual
    nms: número de cadenas más similares
    frec: frecuencia en B del carácter asociado a nodo
si nodo ≠ nulo entonces
    si nodo es un nodo_SIT entonces
        j=1
        repetir
            dl=COP_DL(B,nodo.s(j))
            j=j+1
        hasta que (dl = 0) o (j > nodo.ns)
        si dl = 0 entonces
            crear un conjunto respuesta formado por nodo.s(j)
            nms=1
        fin si
    si no
        si es un nodo de la parte cadena entonces
            si vf(nodo.c) = nodo.f entonces
                Busquedacre_p0(nodo.e)
            fin si
        si no  {es un nodo de la parte_árbol}
            frec=vf(nodo.cd)
            si (nodo.pp ≤ frec) y (frec ≤ nodo.pu) entonces
                Busquedacre_p0(nodo.e(frec))
            fin si
        fin si
    fin si
fin si
```

```
Procedimiento Busquedacre_pa (nodo,cadit,cbnt)
{Realiza el recorrido de la parte_árbol en la búsqueda creciente}
    nodo: nodo actual
    cbnt: número de caracteres de B no tratados
    frec: frecuencia en B del carácter asociado a nodo
    dfm: diferencia de frecuencias máxima
    finf: frecuencia mínima alcanzada en la selección de alternativas
    fsup: frecuencia máxima alcanzada en la selección de alternativas
    falt: frecuencia alternativa
    vcadit: valor de las componentes de DIT para la alternativa actual
si nodo ≠ nulo entonces
    si nodo es un nodo_SIT entonces
        cadit = cadit + cbnt
        si cadit ≤ DTM entonces
            Tratarcre_nS(nodo)
        fin si
    si no
        si es un nodo de la parte cadena entonces
            Tratarcre_pc(nodo,cadit,cbnt)
        si no  {es un nodo de la parte_árbol}
            frec = vf(nodo.cd)
            cbnt = cbnt-frec
            dfm = DTM-cadit
            finf = frec-dfm
            fsup = frec + dfm
            si finf < nodo.pp entonces finf = nodo.pp fin si
            si fsup > nodo.pu entonces fsup = nodo.pu fin si
            para falt = finf hasta fsup hacer
                vcadit = cadit + abs(frec-falt)
                Busquedacre_pa(nodo.e(falt),vcadit,cbnt)
            fin para
        fin si
    fin si
fin si


Procedimiento Tratarcre_nS (nodo)
{Realiza el tratamiento de los nodos_SIT en la búsqueda creciente}
    dl: valor de DL entre B y el sinónimo_DIT actual
para j = 1 hasta nodo.ns hacer
    dl = COP_DL(B,nodo.s(j))
    si dl = DLM entonces
        añadir nodo.s(j) al conjunto respuesta
        nms = nms + 1
    fin si
fin para
```

```
Procedimiento Tratarcre_pc (nodo,cadit,cbnt)
{Realiza el recorrido de la parte_cadena en la búsqueda creciente}
si nodo es un nodo_SIT entonces
    cadit = cadit + cbnt
    si cadit ≤ DTM entonces
        Tratarcre_nS(nodo)
    fin si
si no
    cadit = cadit + abs(nodo.f-vf(nodo.c))
    cbnt = cbnt-vf(nodo.c)
    si cadit ≤ DTM entonces
        Tratarcre_pc(nodo.e,cadit,cbnt)
    fin si
fin si
```

2.2.4. Resultados Experimentales.

Los resultados experimentales se llevaron a cabo en lenguaje C sobre un **HP9000-835**, utilizando un diccionario de 89.655 palabras de la lengua española —sin signos ortográficos, la distribución de frecuencias por longitudes se muestra en la figura 4. La estructura **S-D** construida con este diccionario consta de: 63.994 nodos en la *parte_árbol*, 136.232 en la *parte_cadena* y 77.717 *nodos_SIT*. Se construyen ficheros —de cardinalidad 50— que serán utilizados como argumento de búsqueda de las más similares, para ello se selecciona al azar una palabra del diccionario y se transforma aplicándole diversas operaciones de edición —elegidas aleatoriamente— hasta obtener una palabra

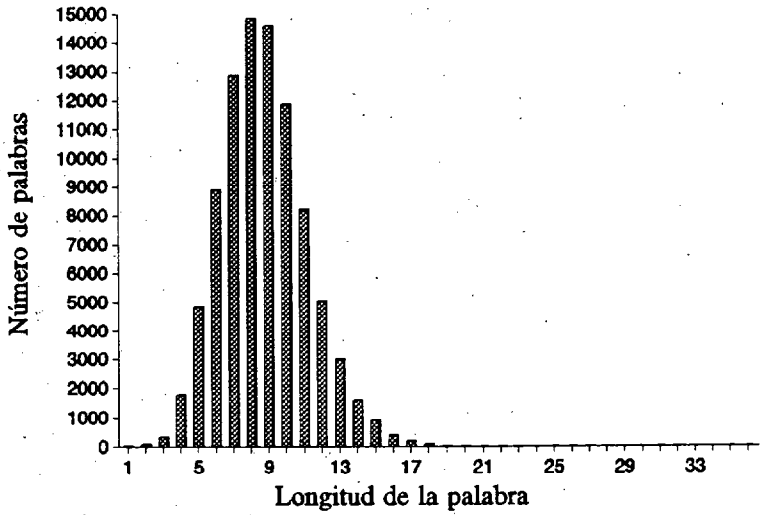


Figura 4

distorsionada, con una **DL** determinada respecto a la correcta. **DL** toma los valores comprendidos entre uno y la mitad de la longitud de la palabra correcta. La longitud de la palabra original, la de la distorsionada y el valor de **DL** indican el fichero al que se va a asignar.

A modo de ejemplo, se muestran en la tabla 1 las palabras más similares a *desmxtadt* que se ha obtenido a partir de *desmayado* aplicándole una *DL=3*.

longitud=8	longitud=9	longitud=10
desmotar	desmolado	desmontado
desatado	desmayado	desmotador
desmatar	desmanado	desmontada
desatada	desmolada	
	desmayada	
	desmanada	

Tabla 1

Se ha realizado el estudio experimental con el fin de comparar los tiempos de búsqueda de la *familia de esquemas* y de los esquemas *decreciente* y *creciente*. Se toma como valor representativo el promedio de los tiempos de búsqueda para cada uno de los ficheros mencionados, teniendo en cuenta que la variabilidad en cada uno de ellos es poco significativa.

Se eligen, como muestra para las representaciones, algunos valores significativos de **DL** ya que, aún siendo éste el parámetro de mayor influencia cuantitativa, el comportamiento relativo de los esquemas es análogo a los que se presentan. Las figuras 5 y 6 muestran la representación tridimensional —tiempo promedio frente a las

longitudes de la palabra original y la distorsionada— de los esquemas *decreciente* y *creciente* para $DL=2$.

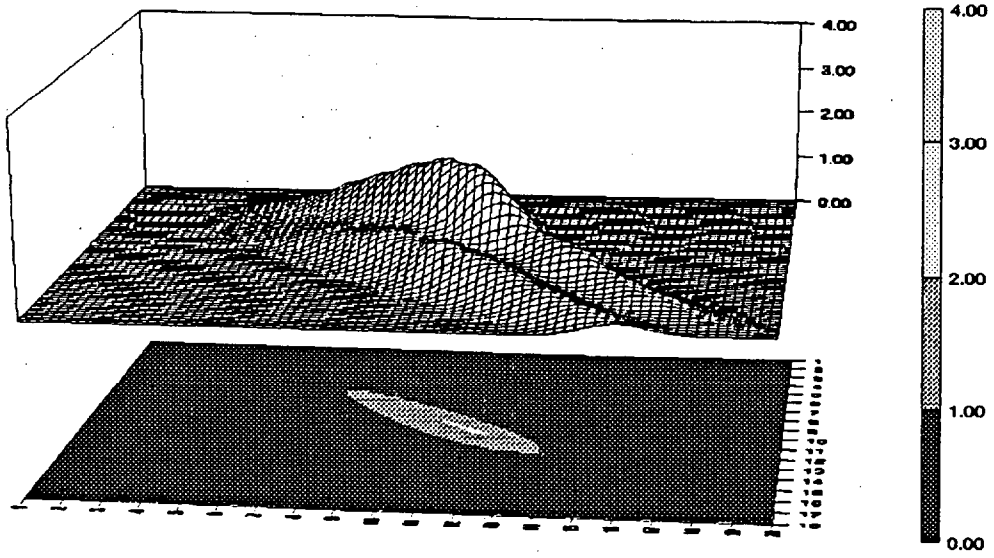


Figura 5

☞ En las figuras 5 y 6, el tiempo de respuesta está fuertemente influenciado por la distribución de longitudes en el diccionario, así los máximos

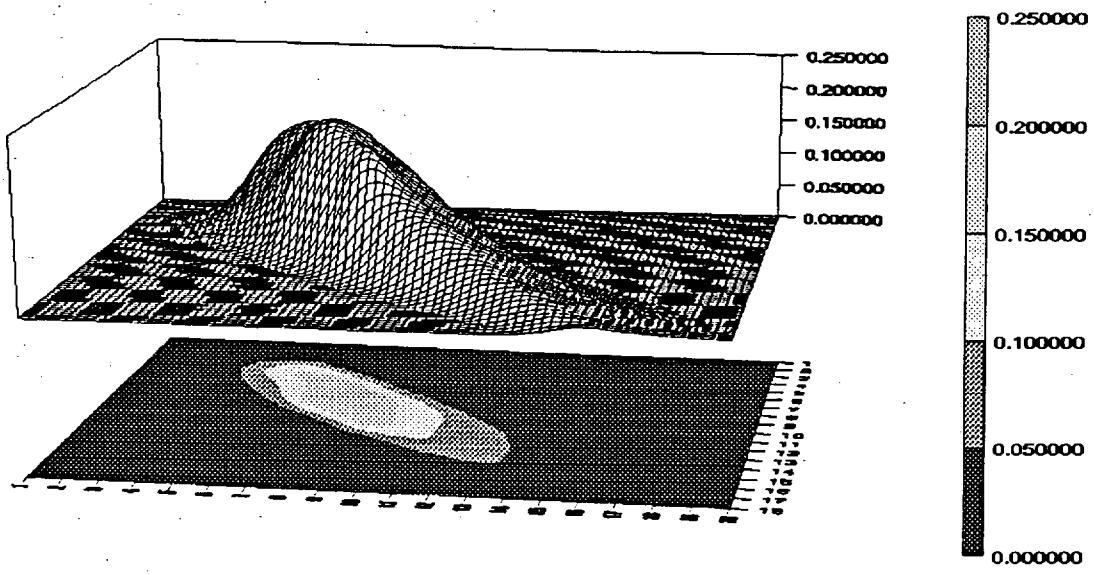


Figura 6

valores se alcanzan para las longitudes intermedias. El esquema *decreciente* presenta un valle central, figura 5, que coincide con la igualdad de longitudes entre la palabra correcta y la distorsionada, debido a que reduce más rapidamente el radio de búsqueda cuando explora en primer lugar el ramal donde se encuentra la palabra correcta.

Debido a las limitaciones que presenta el papel para las representaciones tridimensionales en adelante se harán representaciones bidimensionales. Existen dos razones para seleccionar la longitud de la cadena de búsqueda como variable independiente en las representaciones bidimensionales: en primer lugar, la menor influencia en los tiempos de respuesta de la longitud de la palabra correcta y por otro lado, la conveniencia de referenciar los resultados a partir de la palabra distorsionada por ser ésta la conocida por el usuario.

En las representaciones bidimensionales posteriores, se utiliza la poligonal que une los puntos medios resultantes para cada valor de la longitud de la cadena de búsqueda, para un determinado valor de **DL**.

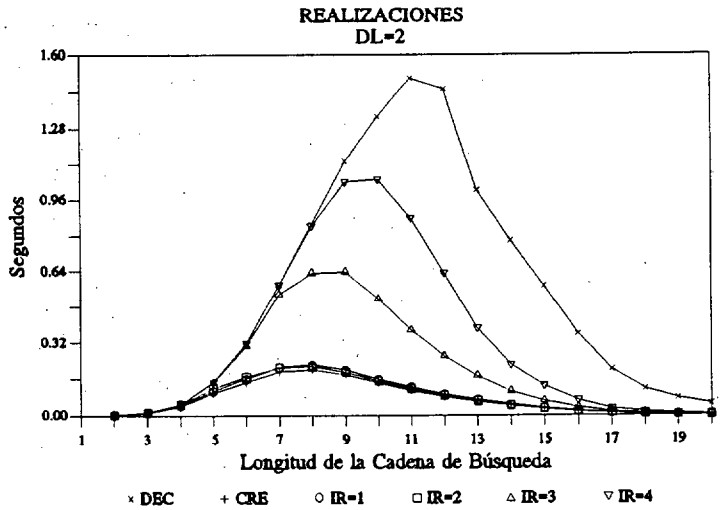


Figura 7

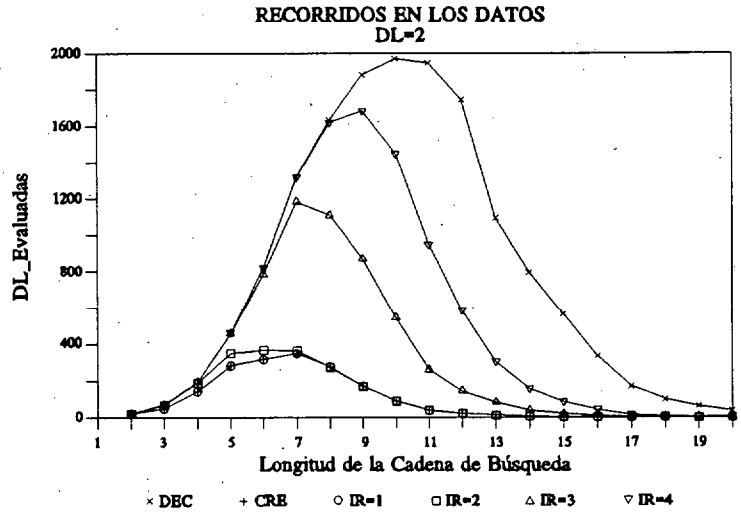


Figura 8

☞ Para *DL=2*, a medida que se incrementa el valor de **IR** crece el tiempo de búsqueda, figura 7, que se emplea en el cálculo de **DL**, figura 8, y en las componentes que intervienen en el cálculo de **DIT**, **C_DIT**, figura 9.

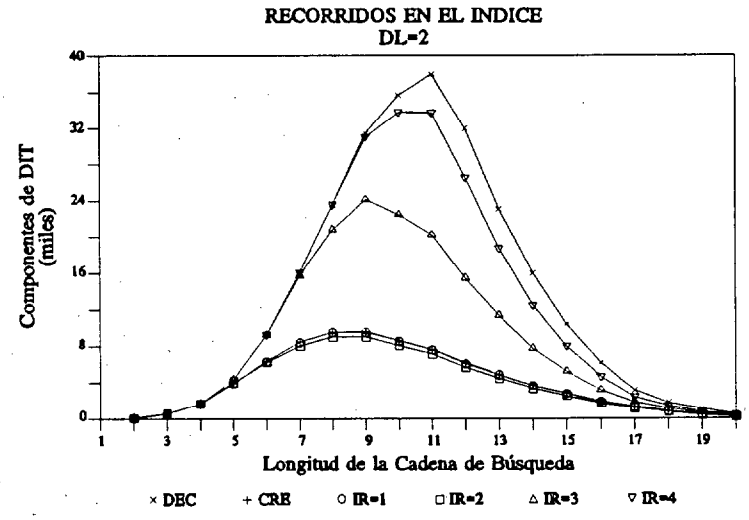


Figura 9

☞ Para *DL=4* se observa una alternancia en los resultados que se muestran en las figuras 10, 11 y 12. Cada valor de **IR** determina una secuencia de valores del radio inicial que se van aproximando a la **DL** mínima. El

tiempo de búsqueda puede ser mejor para valores altos de **IR** si con ellos se alcanza la respuesta en un menor número de fases. Para longitudes de la cadena de búsqueda menores o iguales que ocho, los resultados de tiempos son prácticamente coincidentes debido a que se recuperan palabras con valores de $DL < 4$, lo que da lugar a que no se realice el número esperado de fases para cada valor de **IR**. Para longitudes mayores los resultados aparecen diferenciados siguiendo la alternancia, puesto que la **DL** con las palabras recuperadas es en general igual a cuatro.

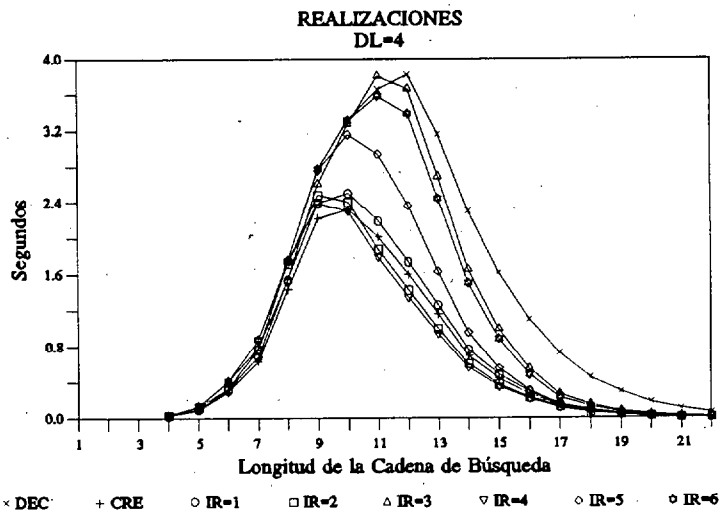


Figura 10

☞ Se muestra, en función del valor de **IR**, toda una gama de resultados que conectan de una manera no abrupta los esquemas *decreciente* y *creciente*, figuras 7-12. Puede observarse que, para los mayores valores de **IR**, las gráficas que representan a las **C_DIT** y al número de **DL** evaluadas, así como las de tiempos se acercan cada vez más a las

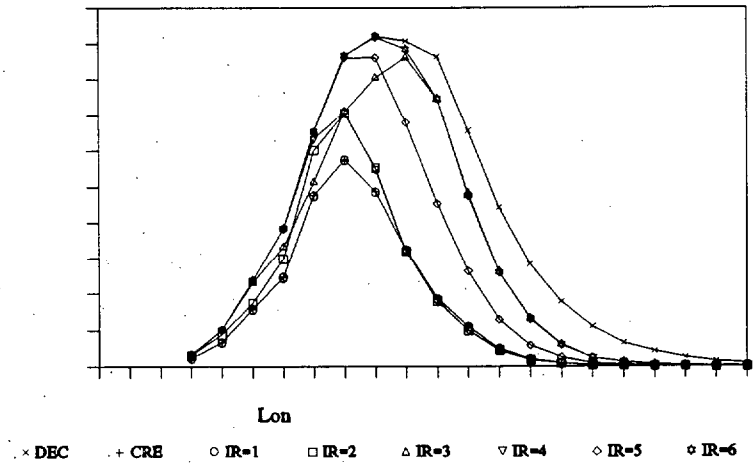


Figura 11

correspondientes del esquema *decreciente*.

☞ El acercamiento máximo de la *familia de esquemas* al *creciente* se obtiene para *IR=1*, en el que se llevan a cabo exactamente el mismo número de **C_DIT** y **DL**, figuras 8, 9 y 11, 12, pero cuya realización,

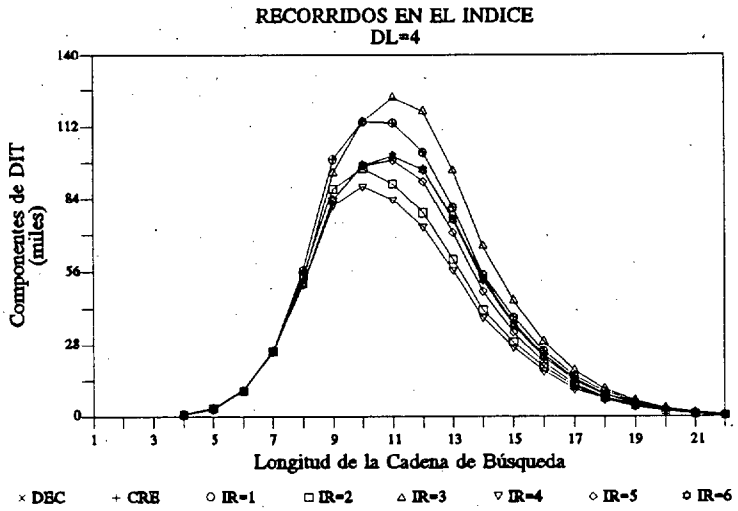


Figura 12

figuras 7 y 10, es ligeramente peor que la del *creciente* debido a la menor eficacia en la toma de alternativas.

☞ La mejor realización del *creciente* respecto al *decreciente* proviene del acentuado descenso en el número de **DL** evaluadas, para todas las longitudes estudiadas, (en las figuras 8 y 11 la gráfica del *creciente* aparece solapada con *IR=1*) incluso a pesar del notable aumento en las **C_DIT** para *DL=4* (en la figura 12 la gráfica del decreciente está solapada con *IR=6*) no le hace perder su atractivo.

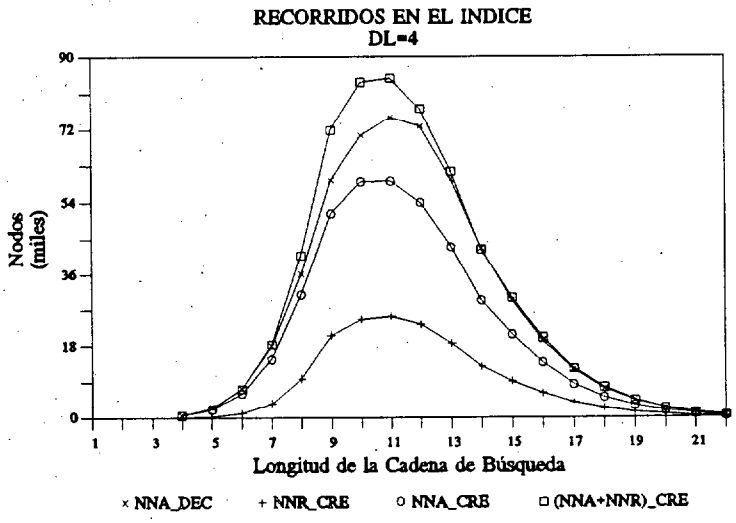


Figura 13

☞ En la figura 13 se ha representado el número de nodos afectados, **NNA**, —nodos del índice que se han explorado alguna vez— para ambos esquemas, observándose que, en todo momento, es inferior para el *creciente*. ¿Cómo es entonces posible el comportamiento expuesto anteriormente acerca de las **C_DIT**? ello se explica por el hecho de que existen nodos que se visitan más de una vez, característica que no posee el esquema *decreciente*; el número de **C_DIT** que se han de calcular

depende, no sólo del **NNA** sino también del número de veces que algunos de éstos se han de revisitar, **NNR**.

☞ Si el esquema *creciente* se lleva a cabo con aprovechamiento de los cálculos previos, el ahorro debido a la recuperación de las **DL** evaluadas para radios de búsqueda inferiores al de la respuesta no es significativo respecto al número de cadenas **DL** exploradas en la fase final; por tanto, el mantenimiento de los valores de **DL** en la estructura no es crítico desde el punto de vista de la realización y su inclusión implicaría un gasto adicional de memoria, figura 14.

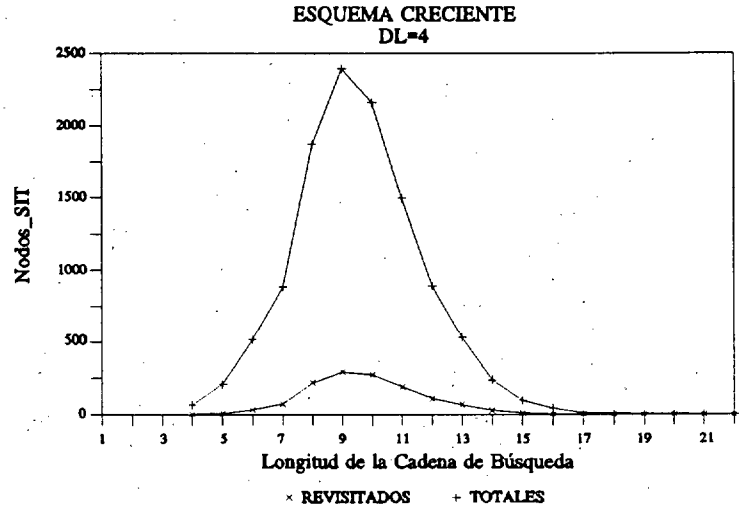


Figura 14

☞ Aproximadamente, en las realizaciones de los esquemas *decreciente* y *creciente*, el recurso temporal se distribuye en el recorrido por el índice evaluando **C_DIT** y en los cálculos de **DL**. En la figura 15, se representa la razón entre el tiempo empleado en el índice y el tiempo total; puede observarse cómo en el *decreciente* es aún importante el

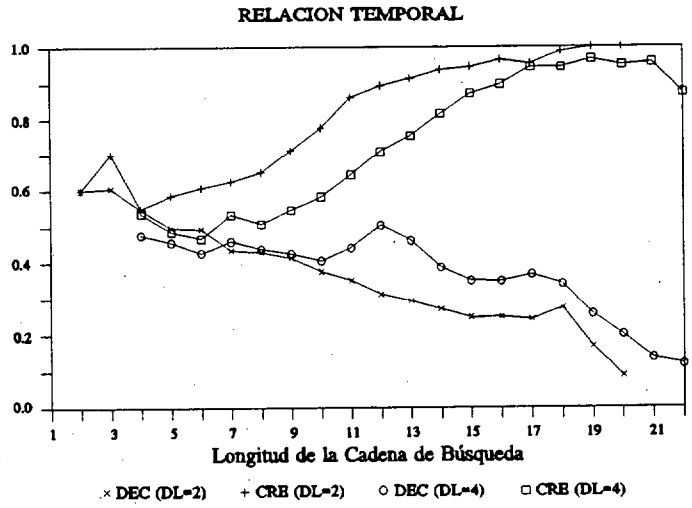


Figura 15

tiempo empleado en cálculos de **DL** —el índice no llega a demandar el 50% de este recurso—; en el *creciente*, la mayor parte del tiempo de cálculo se invierte en el recorrido por el índice, lo cual sugiere que se han de investigar formas de optimizar estos recorridos. Para ambos esquemas, podrán ser empleados nuevos filtros —que aporten propiedades ignoradas por **DIT** con un costo atractivo— con el fin de reducir aún más el número de **DL** que se han de calcular.

Capítulo 3.
Optimización de los Esquemas de Búsqueda Decreciente y Creciente

El tiempo de respuesta de la estructura **S-D** con ambos esquemas está condicionado por la utilización de una poda adecuada en el índice —su eficacia depende de la cantidad de ramales en los que no haya respuesta que se evite explorar— y por la capacidad de reducir el número de veces que se evalúa **DL** —influida por la proximidad existente entre el filtro y **DL**.

3.1. Podas en el Índice: PA y PP.

La porción de estructura recorrida en los esquemas de búsqueda *creciente* y *decreciente* está limitada por la **DIT** entre la cadena de búsqueda y las cadenas del diccionario. La exploración de una alternativa en cada nodo discriminante se realiza siempre que el valor acumulado de las componentes de **DIT**, **CA_DIT**, obtenidas a lo largo del camino desde la raíz, no supere al umbral **DTM**. Este tipo de poda, que se

ha estado utilizando hasta ahora y cuyo valor de corte **CA_DIT** depende exclusivamente de los ancestros, se denomina **PA**.

Se propone una optimización de la poda **PA** que permite reducir la porción de estructura recorrida y, con ello, el número de componentes de **DIT** evaluadas. Se consigue al estimar en cada nodo el valor mínimo atribuible a **DIT** en el camino que va desde la raíz hasta cualquier *nodo_SIT* que penda de ese nodo. Este valor mínimo es el que poda el árbol si supera el umbral **DTM**. Por el carácter predictivo que importa, se ha denominado poda **PP**. Durante el proceso de búsqueda, en cada nodo, para definir el valor que como mínimo tendrá **DIT** es suficiente con suponer que existe una inclusión entre los conjuntos de caracteres no tratados —de la cadena de búsqueda en las cadenas del diccionario que penden de ese nodo o viceversa. Este valor se determina sin más que añadir a **CA_DIT** el valor absoluto de la diferencia entre la suma de las frecuencias de los caracteres no tratados de la cadena de búsqueda, $\sum_{\alpha i \in CNT} B_{\alpha i}$, y los de las cadenas del diccionario correspondientes, $\sum_{\alpha i \in CNT} X_{\alpha i}$; ambas sumas se evalúan sencillamente de la forma:

$$\sum_{\alpha i \in CNT} B_{\alpha i} = |B| - \sum_{\alpha i \in CT} B_{\alpha i}$$

$$\sum_{\alpha i \in CNT} X_{\alpha i} = lr - \sum_{\alpha i \in CT} X_{\alpha i}$$

donde **lr** es la longitud asociada al ramal que se está explorando; tanto $\sum_{\alpha i \in CT} B_{\alpha i}$ como $\sum_{\alpha i \in CT} X_{\alpha i}$ representan, respectivamente, la suma de frecuencias de los caracteres

tratados de la cadena de búsqueda y de las cadenas del diccionario que comparten el nodo.

3.1.1. Resultados Experimentales.

Se ha realizado un estudio experimental para comprobar el efecto de la poda **PP** en las realizaciones de los esquemas de búsqueda *decreciente* y *creciente*.

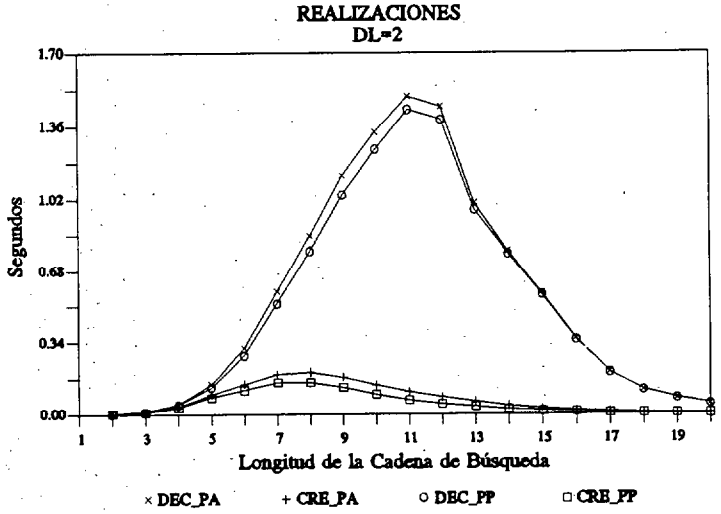


Figura 16

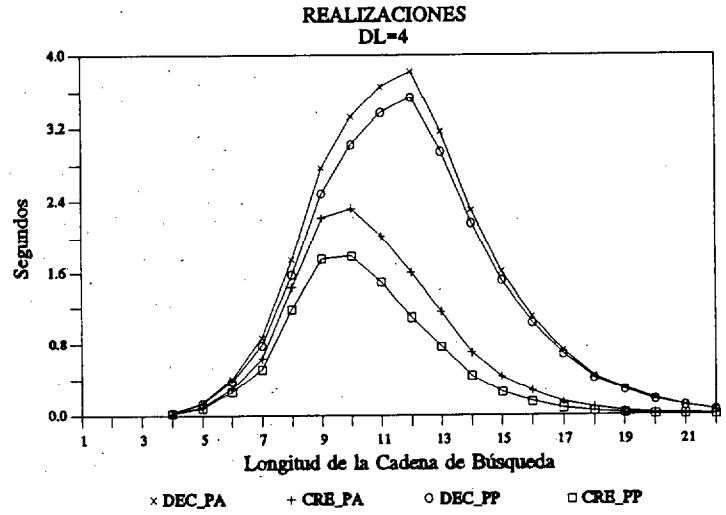


Figura 17

☞ En las figuras 16 y 17, se percibe nítidamente un descenso en los tiempos de búsqueda de los esquemas *decreciente* y *creciente* que utilizan la poda **PP** frente a los correspondientes en los que actúa la poda **PA**; esta diferencia se manifiesta de forma más acusada en el esquema *creciente*. La mejora es más notable a medida que aumenta **DL**.

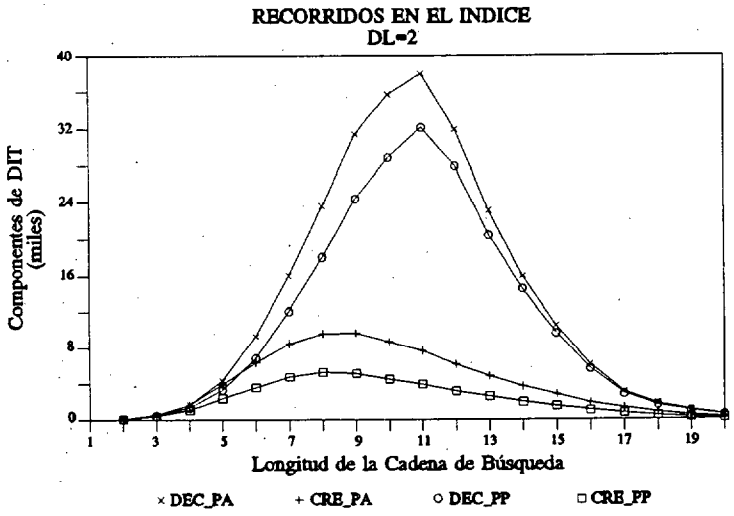


Figura 18

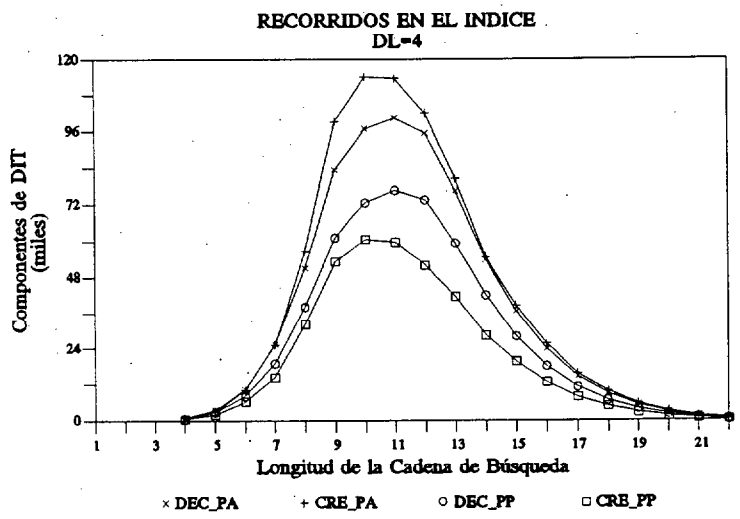


Figura 19

☞ Esta mejor realización se explica exclusivamente en base al ahorro de **C_DIT**, figuras 18 y 19, ya que la aplicación de la nueva poda afecta

sólo al recorrido en el índice y no al número de **DL** que se han de evaluar.

3.2. Distancia de Santana, DS.

Dado que en el cálculo de **DIT** se tienen en cuenta los caracteres comunes aunque no las posiciones que éstos ocupan, siendo esta circunstancia relevante en el cálculo de **DL**, cabe pensar en un filtro intermedio que tenga en cuenta la subsecuencia común más larga entre las dos cadenas.

Diversos artículos presentan algoritmos para encontrar las subsecuencias comunes más largas entre dos cadenas. Wagner y Fischer, [WF74], desarrollan un algoritmo de aplicación del cálculo de **DL** a este fin, con un tiempo de ejecución y una ocupación proporcionales al producto de las longitudes de las cadenas. Hirschberg, [HI75], presenta un algoritmo con un tiempo también cuadrático y un espacio lineal. Posteriormente, Hunt y Szymanski, [HS77], han optimizado el tiempo de ejecución y el espacio; este algoritmo ha sido a su vez mejorado por Kuo y Cross, [KC89], para cadenas de gran longitud.

Definición: *Una cadena **C** es una subsecuencia de una cadena **X** si y sólo si existe una aplicación*

$$F : \{1, 2,\ldots, \mid C \mid\} \longrightarrow \{1, 2,\ldots, \mid X \mid\}$$

*tal que **F** es una función monótona estrictamente creciente y* $F(i)=j \Leftrightarrow C<i> = X<j>$.

Definición: *Una cadena C es una subsecuencia común de dos cadenas X e Y si y sólo si C es una subsecuencia de X y C es subsecuencia de Y.*

Se presenta un algoritmo para el cálculo de la longitud de las subsecuencias comunes más largas, **Lsc**, entre dos cadenas **X** e **Y**, basado en el de Hunt y Szymanski, [HS77]; utiliza un vector, **tr**, que se inicializa a $| X | + 1$ y se actualiza con la posición en **X** de cada nuevo carácter coincidente con alguno de **Y**. Colecciona las posiciones de los caracteres de **X** que forman parte de alguna subsecuencia común y cada vez que se detecta el inicio de una nueva subsecuencia se guarda de nuevo a partir del primer elemento del vector —las posiciones anteriormente almacenadas se pierden. El método consiste en situar en **tr** cada nueva posición en el lugar que le corresponde en orden creciente sustituyendo al valor existente. La longitud de las subsecuencias comunes más largas es igual al número de elementos de **tr** distintos del valor inicial.

Algoritmo para el cálculo de la longitud de las subsecuencias comunes más largas:

```
Función Lsc (X,Y)
    primero: primer carácter del alfabeto
    ultimo: último carácter del alfabeto
    vf(.): vector de frecuencias de X
    lis(.,.): matriz que en cada fila contiene, en orden decreciente, las posiciones en
    X de cada uno de sus caracteres
    lc: longitud de X más uno
    tr(.): almacena las posiciones de los caracteres de X que forman parte de alguna
    subsecuencia común de X e Y
    lista(.): almacena una copia de una fila de la matriz lis correspondiente al carácter
    tratado
para c = primero hasta ultimo hacer
    vf(c) = 0
fin para
```

```
para i = | X | hasta 1 hacer
    vf(X<i>) = vf(X<i>) + 1
    lis(X<i>, vf(X<i>)) = i
fin para
lc = | X | + 1
para i = 1 hasta | Y | hacer
    tr(i) = lc
fin para
para i = 1 hasta | Y | hacer
    si vf(Y<i>) ≠ 0 entonces
        frec = vf(Y<i>)
        lista = lis(Y<i>)
        z = 1
        repetir
            pos = lista(z)
            k = 1
            mientras tr(k) < pos hacer
                k = k + 1
            fin mientras
            tr(k) = pos
            z = z + 1
        hasta que z > frec
    fin si
fin para
k = 1
mientras tr(k) < lc hacer
    k = k + 1
fin mientras
devolver (k-1)
```

Ejemplo:

Sean *X=trabajo* e *Y=pasajero*, el vector de frecuencias de **X**, **vf**, contiene:

$$vf(a)=2,\ vf(b)=1,\ vf(j)=1,\ vf(o)=1,\ vf(r)=1,\ vf(t)=1$$

y

$$vf(\alpha)=0 \quad \forall \alpha \in \{a,b,c,\ldots,z\}-\{a,b,j,o,r,t\}$$

cada fila de la matriz **lis** se corresponde con un carácter y contiene sus posiciones en **X** en orden decreciente, en este caso:

$$lis(a)=(5,3),\ lis(b)=(4),\ lis(j)=(6),\ lis(o)=(7),\ lis(r)=(2),\ lis(t)=(1)$$

En el algoritmo de Hunt y Szymanski el contenido de esta matriz consiste solamente en las listas de posiciones de los caracteres coincidentes de ambas cadenas.

El vector **tr** inicialmente:

(8, 8, 8, 8, 8, 8, 8, 8)

se comprueban los caracteres coincidentes de ambas cadenas recorriendo la cadena *Y=pasajero* de izquierda a derecha. El primer carácter coincidente es **a** por tanto:

lista=(5, 3), pos=5

se actualiza **tr**:

(5, 8, 8, 8, 8, 8, 8, 8)

pos=3

(3, 8, 8, 8, 8, 8, 8, 8)

de nuevo para el carácter **a**:

lista=(5, 3), pos=5

(3, 5, 8, 8, 8, 8, 8, 8)

pos=3

(3, 5, 8, 8, 8, 8, 8, 8)

para el carácter **j**:

lista=(6), pos=6

(3, 5, 6, 8, 8, 8, 8, 8)

para el carácter **r**:

lista=(2), pos=2

(2, 5, 6, 8, 8, 8, 8, 8)

finalmente para el carácter **o**:

$$lista=(7),\ pos=7$$

$$(2, 5, 6, 7, 8, 8, 8, 8)$$

por tanto

$$Lsc(X,Y)=4.$$

Análisis del algoritmo:

La cantidad total de tiempo usada por este algoritmo es proporcional al número de asignaciones ejecutadas. Realiza: **m** —número de caracteres del alfabeto— para inicializar el vector de frecuencias de la cadena **X**, $2^* \mid X \mid$ para actualizar dicho vector y la matriz **lis**, una para inicializar **lc**, e $\mid Y \mid$ para inicializar el vector **tr**; en conjunto

$$m+2^* \mid X \mid +1+ \mid Y \mid$$

El siguiente bucle *para* está controlado por el número de elementos del conjunto $\{(u,v)/\ X<u>=Y<v>\}$, denotado por **r** —7 es el número máximo de asignaciones realizadas en cada coincidencia. Dentro de este mismo bucle se realiza una búsqueda secuencial de la variable **pos** en el vector **tr**; el número esperado de asignaciones es $\mid Y \mid /2$, ya que se espera que **pos** se encuentre en la mitad del vector. Por último se realiza una inicialización de **k** y de nuevo una búsqueda secuencial en el vector **tr** que aporta $\mid Y \mid /2$ asignaciones; por tanto, se tienen:

$$r^*(7+ \mid Y \mid /2)+1+ \mid Y \mid /2$$

en total:

$$m+2^* \mid X \mid +1+ \mid Y \mid +r^*(7+ \mid Y \mid /2)+1+ \mid Y \mid /2$$

Las búsquedas secuenciales se pueden realizar utilizando búsquedas dicotómicas, para cadenas de gran longitud, con lo que se pasa de $|Y|/2$ a $log_2|Y|$. Si además se supone que $|X| \leq |Y|$, se tiene:

$$m+2+2*|X|+|Y|+r*(7+log_2|Y|)+log_2|Y| \leq$$
$$\leq m+2+3*|Y|+r*(7+log_2|Y|)+log_2|Y|$$

por tanto la complejidad de cálculo es $O(r*log_2|Y|+|Y|)$. ◆

Definición: *Dadas dos cadenas X e Y, se define la distancia de Santana, DS(X,Y), como la diferencia entre la longitud mayor de ambas cadenas y la longitud de las subsecuencias comunes más largas, es decir:*

$$DS(X,Y) = máximo\{|X|, |Y|\} - Lsc(X,Y).$$

Teorema: *DS es una distancia en el espacio de las cadenas.*

Demostración:

a) $\forall X, Y: DS(X,Y)=0 \Leftrightarrow X=Y,0$

$$DS(X,Y) = 0 \Leftrightarrow máximo\{|X|, |Y|\} - Lsc(X,Y) = 0 \Leftrightarrow$$

$$\Leftrightarrow máximo\{|X|, |Y|\} = Lsc(X,Y) \Leftrightarrow |X| = |Y| = Lsc(X,Y) \Leftrightarrow X=Y.$$

b) $\forall X, Y: DS(X,Y)=DS(Y,X)$

Por la propia definición.

c) $\forall X, Y, Z: DS(X,Y)+DS(Y,Z) \geq DS(X,Z)$

Sea **LC** la longitud de las subsecuencias comunes más largas de **X**, **Y** y **Z**.

$$DS(X,Y) + DS(Y,Z) =$$
$$= máximo\{ |X|, |Y| \} - Lsc(X,Y) + máximo\{ |Y|, |Z| \} - Lsc(Y,Z) =$$
$$= máximo\{ |X|, |Y| \} + máximo\{ |Y|, |Z| \} - LC - [Lsc(X,Y) + Lsc(Y,Z) - LC]$$

como:

$$Lsc(X,Y) + Lsc(Y,Z) - LC \leq |Y|$$

ya que **LC** es mayor o igual que el número de caracteres de **Y** que forman parte simultánemente de una subsecuencia común más larga de **X** e **Y** y de una subsecuencia común más larga de **Y** y **Z**; se tiene que:

$$DS(X,Y) + DS(Y,Z) \geq$$
$$\geq máximo\{ |X|, |Y| \} + máximo\{ |Y|, |Z| \} - LC - |Y|$$

dado que:

$$máximo\{ |X|, |Y| \} + máximo\{ |Y|, |Z| \} - |Y| \geq máximo\{ |X|, |Z| \}$$

se tiene que:

$$DS(X,Y) + DS(Y,Z) \geq máximo\{ |X|, |Z| \} - LC$$

y como:

$$Lsc(X,Z) \geq LC$$

entonces:

$$DS(X,Y) + DS(Y,Z) \geq máximo\{ |X|, |Z| \} - Lsc(X,Z) = DS(X,Z). \blacksquare$$

Lema: *Sea C una subsecuencia común entre dos cadenas X e Y, se verifica que:*

$$DIT(X,Y) = DIT(X-C, Y-C)$$

donde X-C (Y-C) representa la subcadena de X (Y) que se obtiene al eliminar la subsecuencia C.

Demostración:

Se desprende inmediatamente de la definición de **DIT.** ■

Teorema: *Dadas dos cadenas cualesquiera X e Y, se verifica que:*

$$DIT(X,Y) \leq 2*DS(X,Y) \leq 2*DL(X,Y)$$

Demostración:

Sean:

$$l_{max} = máximo\{ \mid X \mid , \mid Y \mid \}, \quad l_{min} = mínimo\{ \mid X \mid , \mid Y \mid \}$$

y **C** una subsecuencia común más larga entre **X** e **Y**.

Si todos los caracteres de las subcadenas **X-C** e **Y-C** son diferentes, $DIT(X,Y)$ alcanza el máximo valor posible, que será:

$$l_{max} - \mid C \mid + l_{min} - \mid C \mid + (l_{max} - \mid C \mid - (l_{min} - \mid C \mid)) =$$
$$= 2*(l_{max} - \mid C \mid) = 2*DS(X,Y)$$

por tanto, en general:

$$DIT(X,Y) \leq 2*DS(X,Y)$$

Por otro lado, $DL(X,Y)$ alcanza su valor mínimo cuando la transformación de una cadena en otra, se lleva a efecto a través de $l_{min} - \mid C \mid$ sustituciones y $(l_{max} - \mid C \mid - (l_{min} - \mid C \mid))$ inserciones en la cadena de menor longitud (o extracciones en la más larga); este valor es:

$$l_{min} - \mid C \mid + (l_{max} - \mid C \mid - (l_{min} - \mid C \mid)) = l_{max} - \mid C \mid = DS(X,Y)$$

implicando, en cualquier caso, que:

$$DS(X,Y) \leq DL(X,Y).$$ ■

La acotación del teorema anterior, conjuntamente con el hecho de que **DS** tiene un menor costo computacional que **DL**, sugiere la inclusión de **DS** en los esquemas de búsqueda como filtro que tenga por objeto reducir el número de **DL** que se evalúan. El filtro **DS** se utiliza cada vez que se alcanza un *nodo_SIT*. A continuación se muestran las modificaciones de los procedimientos que tratan dichos nodos en ambos esquemas.

*Algoritmos de tratamiento de los nodos_SIT para el esquema decreciente y creciente con el filtro **DS**:*

```
Procedimiento Tratardec_nS (nodo)
    dl: valor de DL entre B y el sinónimo_DIT actual
    l_max: máximo entre las longitudes de B y del ramal del nodo raíz que se está
    explorando
{l_max se calcula una vez conocido el ramal de descenso en el nodo raíz}
para j = 1 hasta nodo.ns hacer
    DS = l_max-Lsc(B,nodo.s(j))
    si DS ≤ DLM entonces
        dl = COP_DL(B,nodo.s(j))
        si dl < DLM entonces
            DLM = dl
            DTM = 2*DLM
            nms = 1
            eliminar el conjunto respuesta
            crear un conjunto respuesta formado por nodo.s(j)
            si dl = 0 entonces retorno de final de búsqueda fin si
        si no
            si dl = DLM entonces
                añadir nodo.s(j) al conjunto respuesta
                nms = nms + 1
            fin si
        fin si
    fin si
fin para
```

```
Procedimiento Tratarcre_nS (nodo)
   dl: valor de DL entre B y el sinónimo_DIT actual
   l_max: máximo entre las longitudes de B y del ramal del nodo raíz que se está
   explorando
{l_max se calcula una vez conocido el ramal de descenso en el nodo raíz}
para j = 1 hasta nodo.ns hacer
   DS = l_max-Lsc(B,nodo.s(j))
   si DS ≤ DLM entonces
      dl = COP_DL(B,nodo.s(j))
      si dl = DLM entonces
         añadir nodo.s(j) al conjunto respuesta
         nms = nms + 1
      fin si
   fin si
fin para
```

La cantidad total de tiempo empleada en la obtención de **DS** depende del tiempo de cálculo de la longitud de las subsecuencias comunes más largas. Se reduce el tiempo de ejecución de **Lsc** debido a que, dada una cadena de búsqueda **B**, las instrucciones correspondientes a la inicialización y actualización de su vector de frecuencias, la actualización de la matriz **lis** y la inicialización de **lc** sólo se ejecutan una vez cuando *Lsc(B,X)* se utiliza en los esquemas de búsqueda.

Las cadenas más similares a **B** según **DS** no son necesariamente las **DL** más similares. En efecto, si **X** difiere de **B** en una extracción y en una inserción, entonces:

$$DL(B,X)=2$$

y

$$DS(B,X)=1$$

se puede encontrar una cadena, **Y**, que difiera de **B** en una sustitución, siendo:

$$DL(B,Y)=1$$

y

$$DS(B,Y)=1$$

por tanto **X** es una cadena más similar a **B** según **DS** que no es **DL** más similar. Por ejemplo: *B=antejo*, *X=tanteo* e *Y=antojo*.

3.2.1. Resultados Experimentales.

Se ha realizado un estudio experimental para comprobar la influencia del filtro **DS** en los esquemas de búsqueda *decreciente* y *creciente* dotados de la poda **PP**.

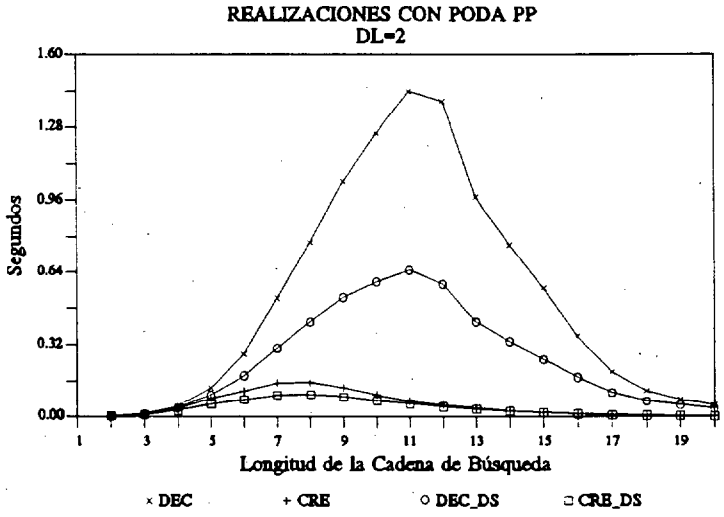


Figura 20

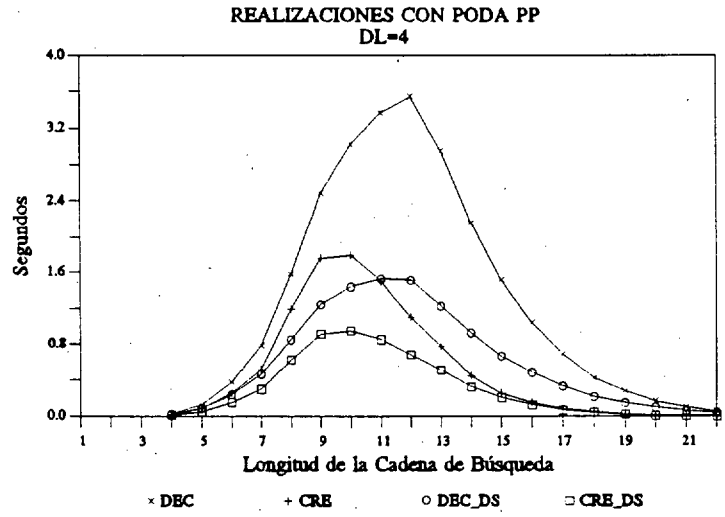


Figura 21

☞ En las figuras 20 y 21 se aprecia un descenso en los tiempos de búsqueda de los esquemas con poda **PP** al incorporarles el filtro **DS**, esta diferencia es más acusada en el esquema *decreciente* y aumenta de forma notoria a medida que crece **DL**.

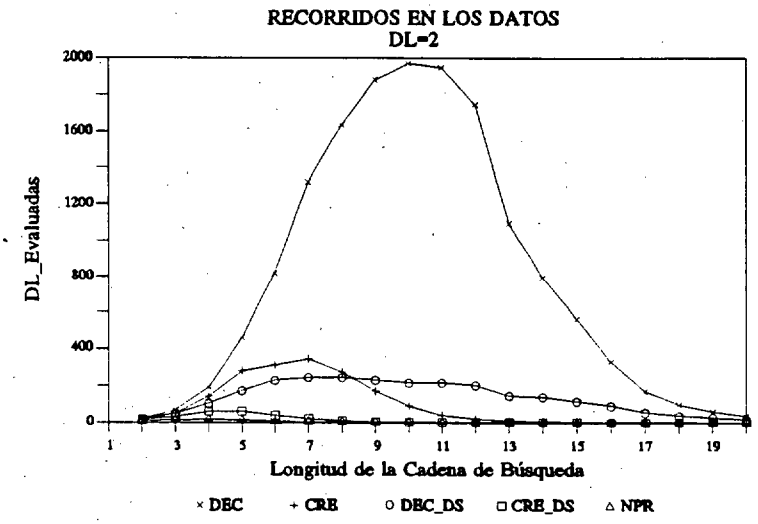


Figura 22

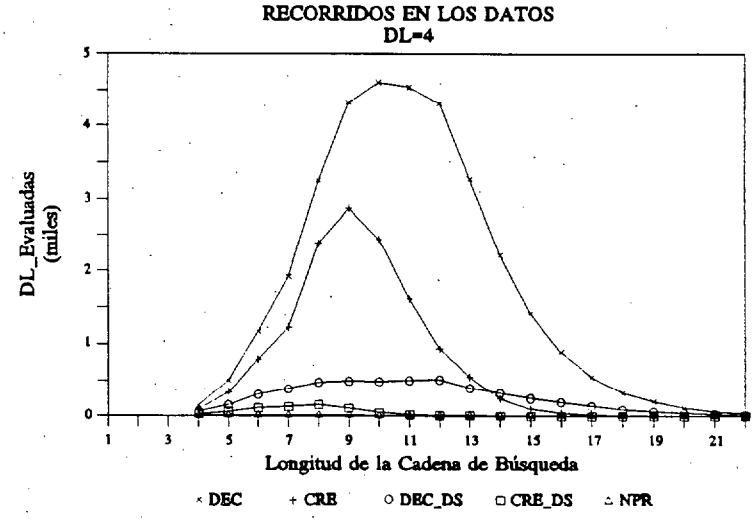


Figura 23

☞ La mejor realización de ambos esquemas al incluir el filtro **DS** se debe exclusivamente al ahorro del número de **DL** evaluadas, figuras 22 y 23, ya que no afecta en absoluto al recorrido en el índice. Obsérvese, en las

figuras 22 y 23, como el número de **DL** evaluadas en ambos esquemas se acerca, bastante más en el *creciente* aunque con mayor ímpetu en el *decreciente*, al tamaño o multiplicidad de la respuesta, **NPR**. **NPR** constituye el nivel mínimo teórico alcanzable para el número de **DL** a calcular.

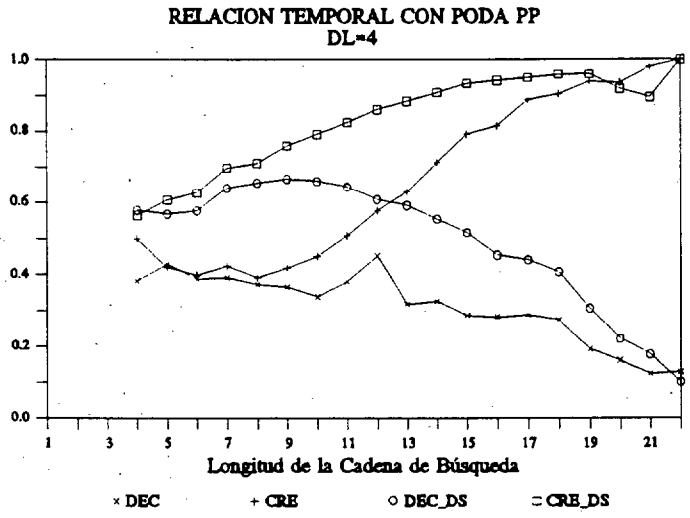


Figura 24

☞ La introducción de la poda **PP** acorta el recorrido en el índice, produciendo en consecuencia una disminución en la relación entre el tiempo empleado en el índice y el total; sin embargo, este efecto se ve ampliamente sobrepasado, figura 24, por la disminución del número de **DL** evaluadas como consecuencia de la introducción del filtro **DS**. La relación temporal del esquema *creciente* con **DS** indica que el tiempo empleado en el índice está por encima del 70% del total. El esquema *decreciente* con **DS** en cambio presenta un reparto más equitativo del

tiempo, salvo para grandes longitudes en las que la fracción de tiempo consumido en el índice disminuye rápidamente.

En posteriores referencias a los esquemas de búsqueda *creciente* y *decreciente* se debe entender que llevan incorporados la poda **PP** y el filtro **DS**.

Capítulo 4.
Paginación de la Estructura S-D

Si el tamaño de la estructura es tal que no es posible ubicarla en memoria interna, se plantea el problema de su paginación con el fin de minimizar el número de accesos a disco. Los criterios a tener en cuenta son: el tiempo de respuesta a las peticiones y la ocupación. No se consideran los problemas relativos al mantenimiento —inserciones, extracciones y reorganizaciones— debido al carácter estático del diccionario.

El nodo *raíz* se considera una página. En las restantes, se sitúan los nodos de la *parte_árbol*, los de la *parte_cadena* y los *nodos_SIT* según el tipo de paginación utilizado. Para ello, se considera la cantidad de memoria necesaria para almacenar los pares αf de cada lista de la *parte_cadena* conjuntamente con la del *nodo_SIT* correspondiente y se organizan de forma secuencial.

4.1. Paginación Según el Recorrido.

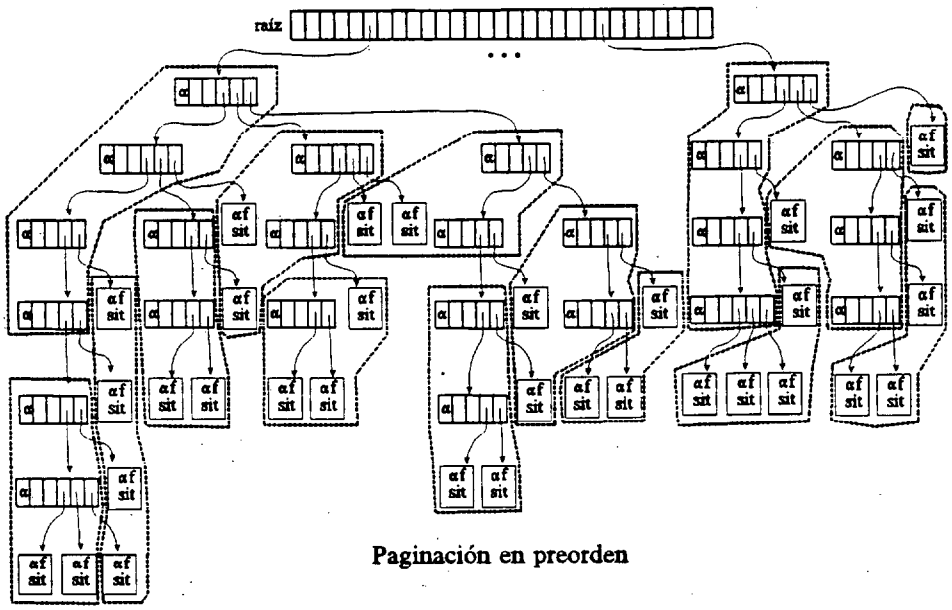


Figura 25

Un primer intento para resolver el problema consiste en llenar las páginas con los nodos al recorrer la estructura en *preorden* o en *postorden* —generalización a un árbol multirrama de los recorridos en un árbol binario. La información inscrita en cada página creada de esta forma se refiere a palabras con igual longitud. Las figuras 25 y 26 muestran un bosquejo de ambas paginaciones, suponiendo que todos los nodos ocupan lo mismo y que la capacidad de cada página es de cuatro nodos.

Obviamente, ambas paginaciones conducen a un alto porcentaje de ocupación. Sin embargo, presentan la posibilidad de favorecer múltiples accesos a una página —en la que existen ramos no conexos, figuras 25 y 26— al seleccionar las alternativas de descenso, desde el mismo o distintos nodos. Se introduce un buffer con el fin de intentar evitar las lecturas repetidas de una página en memoria secundaria.

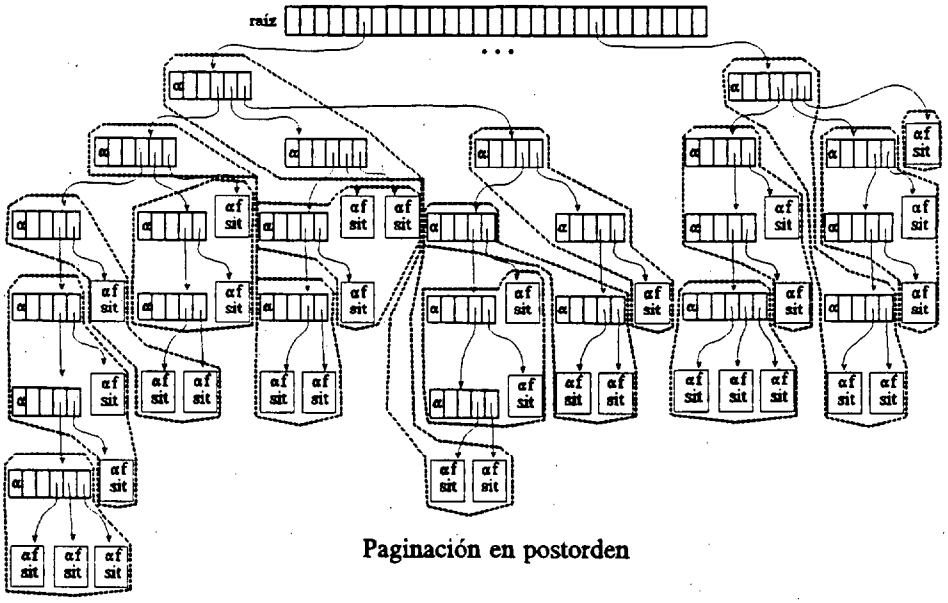


Figura 26

Se estudian distintas políticas de buffer: liberar la página visitada con menor frecuencia, **MFU**, —cada página del buffer tiene asociado un valor que indica el número de veces que se ha examinado—; liberar la página menos recientemente usada, **MRU**, —a cada página del buffer se le asigna un contador de orden de utilización—; la primera en entrar es la primera en salir, **PEPS**, y la última en entrar es la primera en salir, **UEPS**.

La política **MFU** tiene posibilidad de bloquear el buffer con las páginas más reutilizadas y proporciona un buen rendimiento cuando existe un gran número de accesos a la misma página. La política **MRU** se adapta al grado de reutilización de las páginas cuando se explora una determinada longitud de palabras y mantiene el grado de eficacia al cambiar de longitud, en contraposición con la **MFU** que retiene en el buffer páginas de otras longitudes. La política **PEPS** es eficaz para un bajo grado de reutilización y funciona de forma análoga a la **MRU** al cambiar de longitud. La política

UEPS actúa como si trabajara con un buffer que sólo contiene una página, ya que una vez que se completa, únicamente se puede sustituir la última.

4.1.1. Resultados Experimentales.

Se ha realizado una simulación de las dos formas de paginación sobre la estructura **S-D**; se ha medido tanto el número de páginas que se han utilizado como el porcentaje de ocupación de las mismas; los resultados se muestran en la tabla 2. Se observa que el número de páginas y la ocupación de las mismas es prácticamente independiente del tipo de paginación.

	Preorden		Postorden	
Tamaño de la página	Número de páginas	% ocupación	Número de páginas	% ocupación
1K	3.143	98.50%	3.144	98.47%
2K	1.569	98.66%	1.569	98.66%
4K	789	98.10%	789	98.10%
8K	400	96.75%	400	96.75%

Tabla 2

El nodo *raíz* se mantiene en memoria interna durante todo el proceso de búsqueda.

Sobre cada una de las dos formas de paginación de la estructura se ejecutan las búsquedas de las más similares, con el fin de determinar con qué política de buffer —**MFU**, **MRU**, **PEPS** o **UEPS**— y bajo qué esquema de búsqueda —*creciente* o *decreciente*— se alcanza un mejor rendimiento.

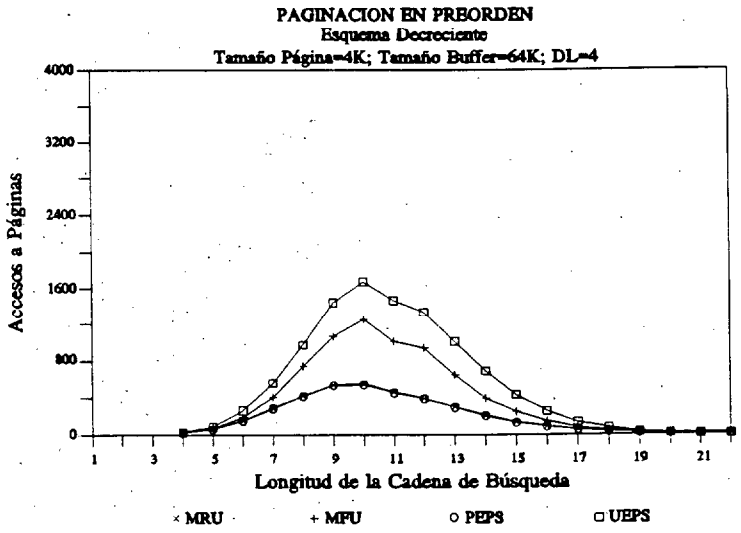


Figura 27

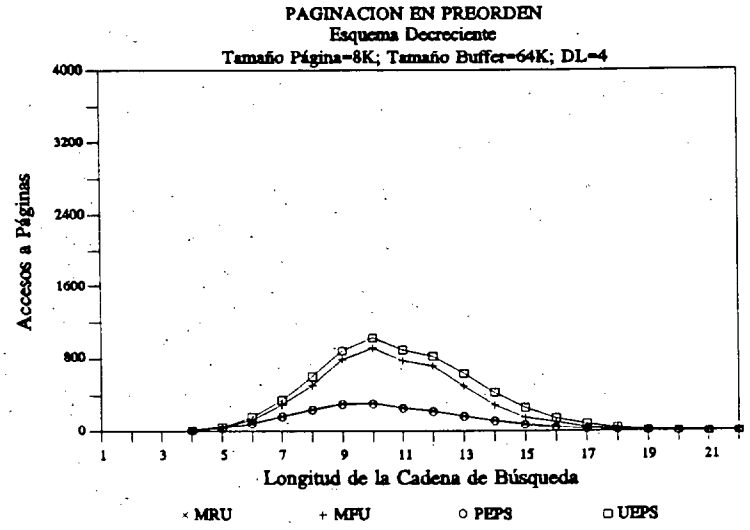


Figura 28

- Claramente, el *tamaño de la página* está en relación inversa con el número de accesos a memoria externa, cuanto más grande es la página menor es el número de transferencias, figuras 27 y 28.
- Asimismo, el *tamaño del buffer* también está en relación inversa con el número de accesos, aunque se ha observado experimentalmente que su influencia es menor, figuras 27 y 29. La política **MFU** presenta una perdida de eficacia al disminuir el tamaño del buffer.

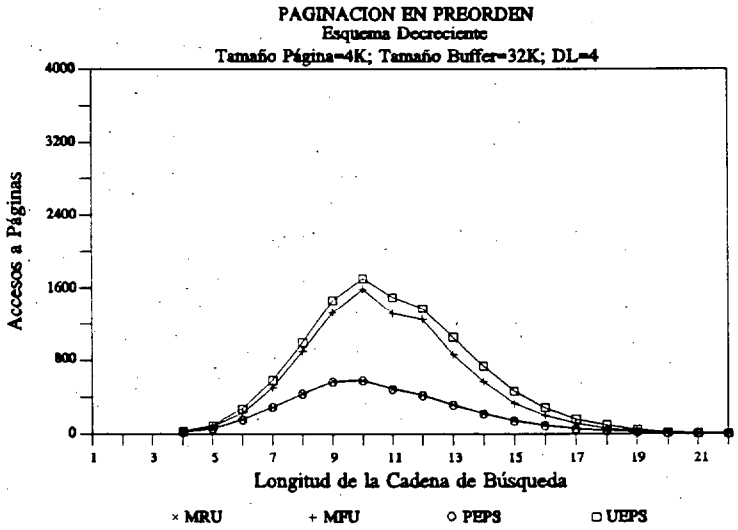


Figura 29

El *tamaño del buffer* se ha elegido de tal forma que como mínimo sea igual a la longitud máxima del camino medida en páginas. Se consideran como valores más adecuados para el *tamaño de la página* a partir de 4K y para el *tamaño del buffer* superior a 32K.

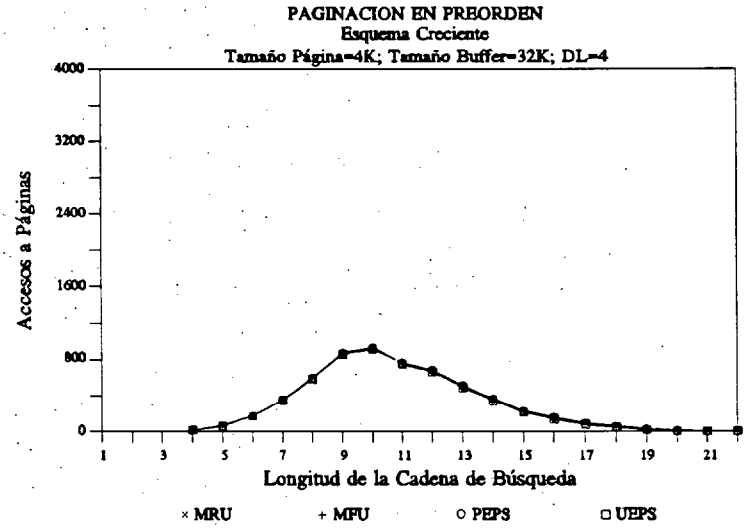


Figura 30

☞ Con el incremento del valor de **DL** aumenta la porción de estructura explorada y con ello el número de accesos, figuras 29, 35 y 30, 36.

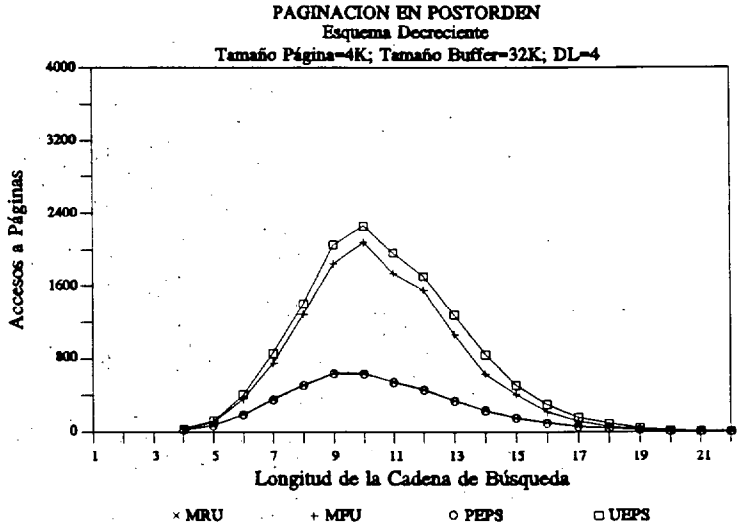


Figura 31

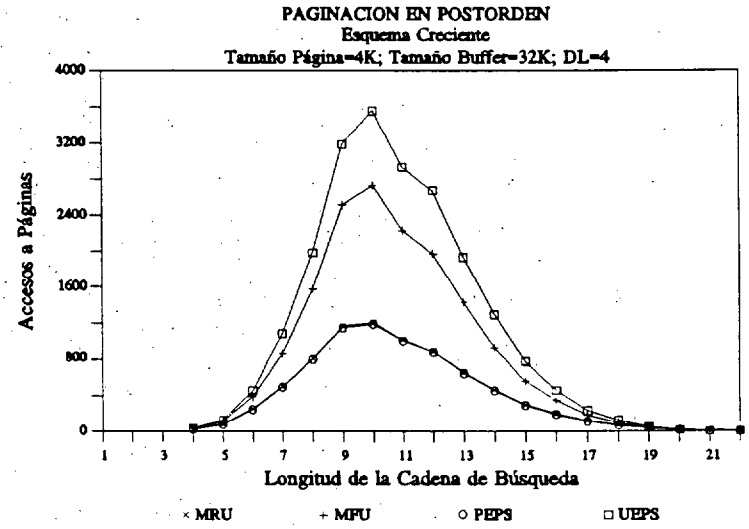


Figura 32

☞ Cualquier política de buffer sobre la estructura paginada en *preorden* siempre da lugar a un mejor rendimiento frente a la obtenida en *postorden*, figuras 29, 31 y 30, 32. Esto es una consecuencia de su mejor adaptación al perfil de la estructura —sesgado hacia la izquierda como se muestra en las figuras 33 y 34 para palabras de longitud 4 y 5 respectivamente— y, por tanto, existe una mayor utilización de la información que guarda cada página.

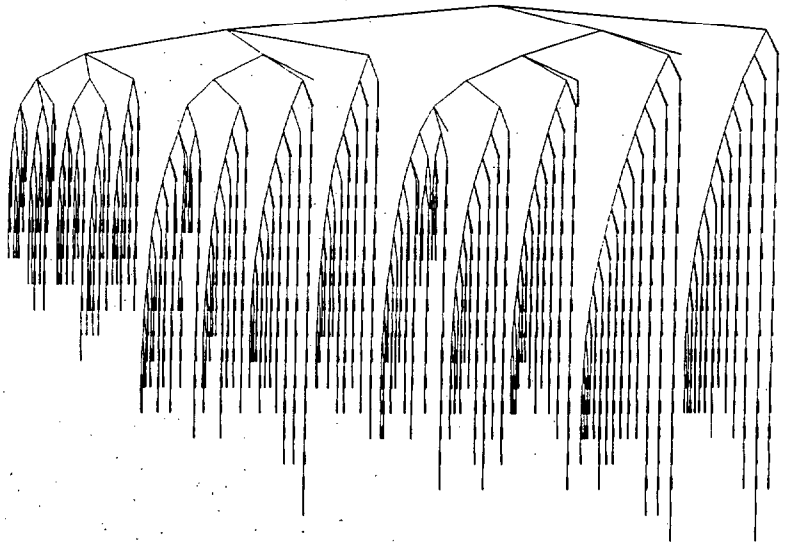


Figura 33

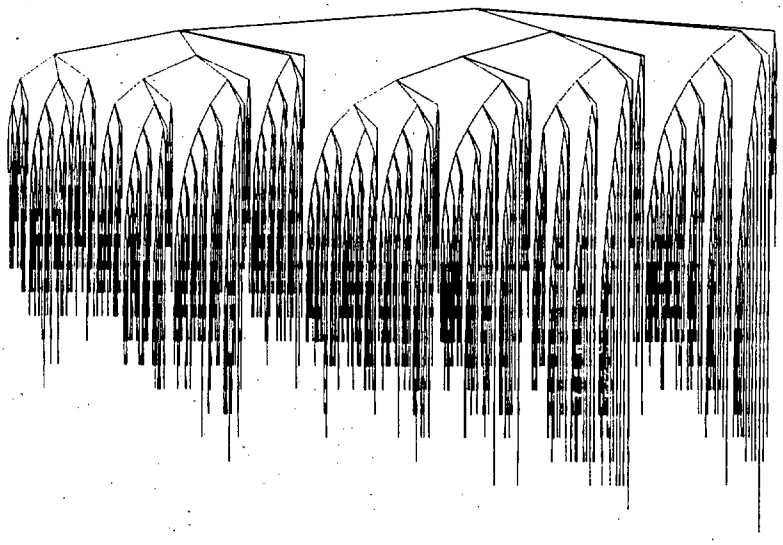


Figura 34

☞ Las políticas del buffer que presentan mejores resultados son la **MRU** y la **PEPS** —esta coincidencia indica que la reutilización es baja—; le siguen **MFU** y **UEPS**. La diferencia de **MRU** y **PEPS** con **MFU** y **UEPS** se debe fundamentalmente al bloqueo del buffer de las dos últimas cuando se explora cada longitud y tal diferencia se acentúa al crecer **DL**. Indudablemente, **MFU** es mejor que **UEPS** porque funciona con un tamaño de buffer mayor que una página. Se ha comprobado

experimentalmente que en el esquema *decreciente* **MFU** y **UEPS** mejoran el rendimiento cuando se libera el buffer al cambiar de longitud, aunque no lo suficiente como para alcanzar a las otras dos políticas. En el esquema *creciente*, el uso de esta opción empeora los resultados.

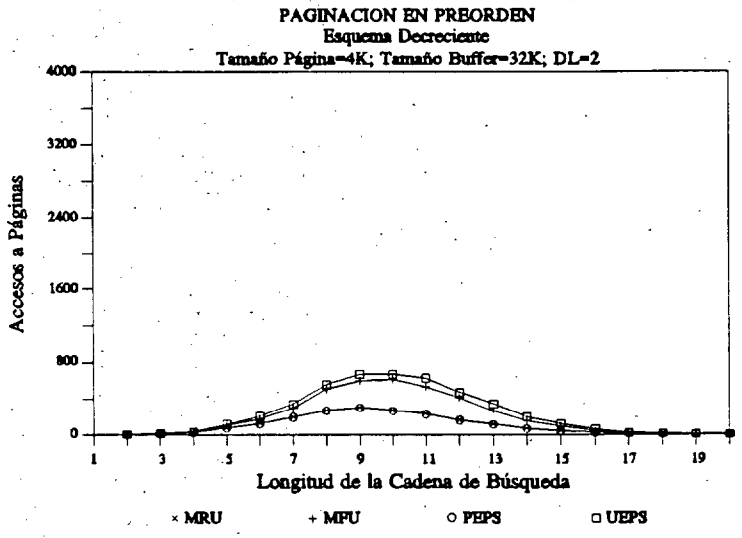


Figura 35

☞ La búsqueda con el esquema *creciente* sobre la estructura paginada en *preorden* presenta una excepción respecto al punto anterior. El número de accesos es independiente de la política del buffer, figuras 30 y 36. La selección de alternativas del esquema *creciente* y la paginación en *preorden* determinan que, en cada fase, sólo se reutiliza la última página accedida, por tanto, no tiene incidencia significativa la política de desalojo de páginas. En cada nueva fase se exploran al menos las mismas páginas que en la anterior, la única política que tiene posibilidad de proporcionar algún beneficio en esta situación es la **UEPS** —mantiene

el camino inicial de una fase a otra—; sin embargo no es significativo tal y como muestran los resultados.

☞ En la búsqueda sobre la estructura paginada en *postorden* se obtienen siempre mejores resultados para el esquema *decreciente* que para el *creciente* cuando $DL>2$, independientemente de la política, figuras 31 y 32. El número de accesos del esquema *creciente* se incrementa con el número de fases que coincide con el valor de la **DL** entre la cadena de búsqueda y sus más similares. Para *tamaño de página* 4K y $DL=2$, el comportamiento de ambos esquemas con **MRU** y **PEPS** es prácticamente el mismo.

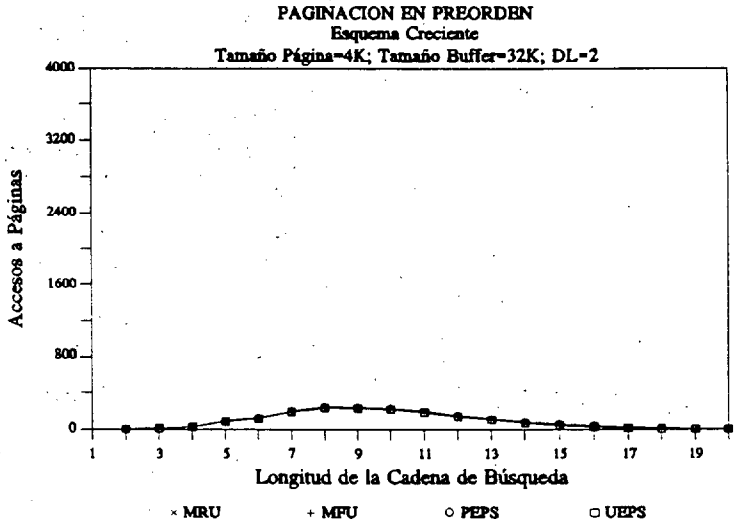


Figura 36

☞ Se ha obtenido que en la búsqueda sobre la estructura paginada en *preorden* el esquema *decreciente* con **MRU** y **PEPS** es mejor que el *creciente* con cualquier política para $DL>2$, figuras 29 y 30. Las políticas **MRU** y **PEPS** mantienen en el buffer páginas utilizadas por el

esquema *decreciente* y que son posteriormente visitadas en función del recorrido en vaivén de este esquema.

- Para $DL \leq 2$ y longitudes intermedias de la palabra de búsqueda, el esquema *creciente*, para todas las políticas del buffer, es ligeramente mejor que el *decreciente* con **MRU** y **PEPS**, figuras 35 y 36, —el *creciente* realiza como mucho dos fases y la porción de estructura implicada es menor. Se ha comprobado que la mejoría disminuye con el aumento del *tamaño de la página* —para un *tamaño de página* 8K y $DL=2$ coinciden.

Como conclusión se obtiene que el menor número de accesos se alcanza con el esquema de búsqueda *decreciente*, excepto para $DL=1$, sobre la estructura paginada en *preorden* y con las políticas **MRU** y **PEPS**. Se selecciona **PEPS** por ser menos costosa en tiempo de ejecución que **MRU**, ya que ésta última necesita controlar el orden de utilización.

4.2. Paginación Atendiendo al Esquema de Búsqueda Decreciente.

Se propone una nueva forma de paginar la estructura, *segunbusca*, de tal modo que se respete el siguiente principio: durante la búsqueda, una vez que se abandone una

página no se vuelve a acceder a ella por otro camino. Esta condición da lugar a que se utilice solamente una pila cuyo tamaño no excederá la longitud en páginas del camino más largo, para mantener en memoria las páginas necesarias.

Es necesario tener construida la estructura **S-D** antes de realizar este proceso de paginación.

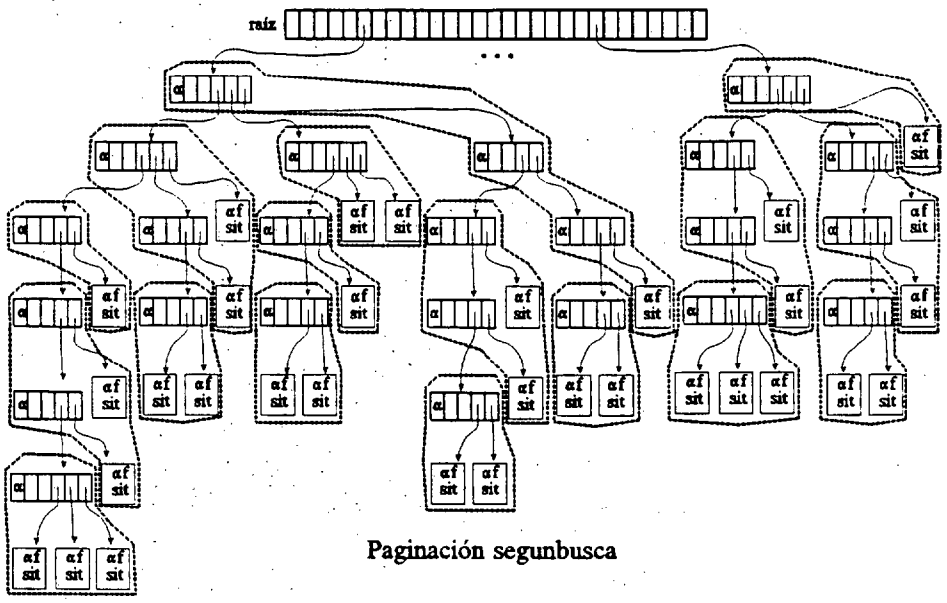


Figura 37

Esta paginación, figura 37, sigue un proceso recursivo tal que en cada *nodo* se comprueba si todos sus hijos y él mismo caben en una página, en cuyo caso, se traslada la decisión de completar la página al *padre de nodo*. En caso contrario, todos los hijos que quepan, empezando por los de menor ocupación, se agrupan en una página junto con *nodo* y se traslada la decisión de completar tal página al *padre de nodo*; el resto de los hijos de *nodo* —los de mayor ocupación— se guardan en páginas separadas. Consecuentemente, se produce un descenso del porcentaje de ocupación. Nótese que los hijos de *nodo* mencionados incluyen a sus descendientes aunque no tienen por qué ser

los subárboles completos, sino que pueden ser restos de subárboles que han quedado sin paginar.

Algoritmo de construcción de la estructura S-D con la paginación segunbusca:

```
Procedimiento Construye_S-Dpag (raiz)
   raiz(.): contiene los punteros que señalan a la parte_árbol asociada a cada longitud
para long = longmin hasta longmax hacer
   si Ocupa_parte_arbol(raiz(long)) > 0 entonces
      situar raiz(long) y sus descendientes no paginados en una nueva página
      almacenar la página
   fin si
fin para
```

La función Ocupa_parte_arbol invoca a las siguientes funciones:

Ocupa(nodo): Devuelve la ocupación en bytes de un nodo de la parte_árbol.

Ocupa_nodo_SIT(nodo): Devuelve la ocupación en bytes de un nodo_SIT.

Ocupa_parte_cadena(nodo): Calcula la ocupación en bytes de los pares σf a partir del primer nodo de la parte cadena, nodo, acumulada con la del correspondiente nodo_SIT.

```
Función Ocupa_parte_arbol (nodo)
{Realiza la paginación de los hijos de nodo y devuelve la ocupación residual}
si nodo = nulo entonces
   devolver (0)
si no
   si nodo es un nodo_SIT entonces
      devolver (Ocupa_nodo_SIT(nodo))
   si no
      si nodo es un nodo de la parte_cadena entonces
         devolver (Ocupa_parte_cadena(nodo))
      si no
         ocupacion = Ocupa(nodo)
         para i = nodo.pp hasta nodo.pu hacer
            ocupahijo(1,i) = i
            ocupahijo(2,i) = Ocupa_parte_arbol(nodo.e(i))
            ocupacion = ocupacion + ocupahijo(2,i)
         fin para
         si ocupacion ≤ tamaño_pagina entonces
            devolver (ocupacion)
         si no
            Ordena(ocupahijo)
            {ordena ocupahijo en valores crecientes de su 2ª fila}
            i = nodo.pu
            repetir
               ocupacion = ocupacion-ocupahijo(2,i)
               situar nodo.e(ocupahijo(1,i)) y sus descendientes no paginados en
               una nueva página
               almacenar la página
            hasta que ocupacion < tamaño_pagina
            devolver (ocupacion)
         fin si
      fin si
   fin si
fin si
```

4.2.1. Resultados Experimentales.

Se ha realizado una simulación sobre la estructura **S-D** de la paginación propuesta, *segunbusca*, los resultados del número de páginas necesarias y su porcentaje de ocupación se muestran en la tabla 3.

	Preorden		Segunbusca	
Tamaño de la página	Número de páginas	% ocupación	Número de páginas	% ocupación
4K	789	98.10%	1.044	74.10%
8K	400	96.75%	535	72.30%

Tabla 3

☞ Como se esperaba, el número de páginas que requiere el almacenamiento de la estructura ha aumentado con respecto a la paginación en *preorden* dando lugar a una pérdida en el porcentaje de ocupación en torno al 24%.

Se ha medido el número de accesos que requiere el esquema de búsqueda *decreciente* sobre la estructura con la nueva paginación, *segunbusca*, y se compara con los resultados obtenidos para el mismo esquema de búsqueda sobre la estructura paginada en *preorden* con la política **PEPS**.

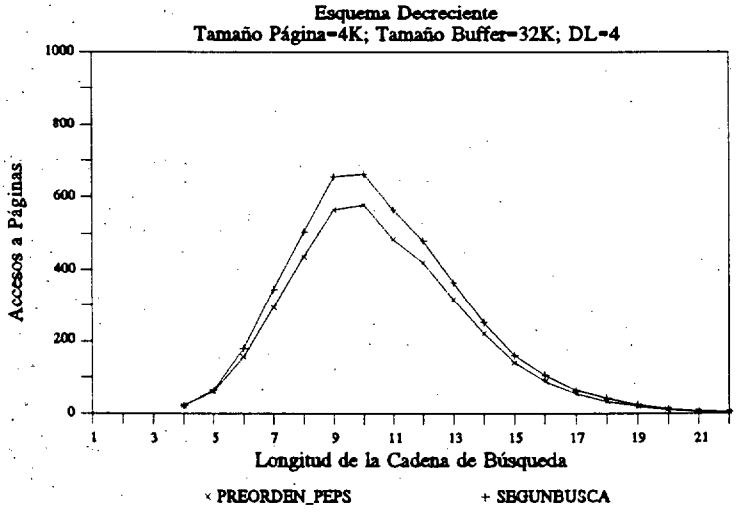


Figura 38

☞ Se ha obtenido que independientemente del valor de **DL** y del *tamaño de la página*, la paginación *segunbusca* no supera el rendimiento

alcanzado por la paginación en *preorden*, figuras 38 y 39. El descenso del porcentaje de ocupación conduce a un mayor número de accesos.

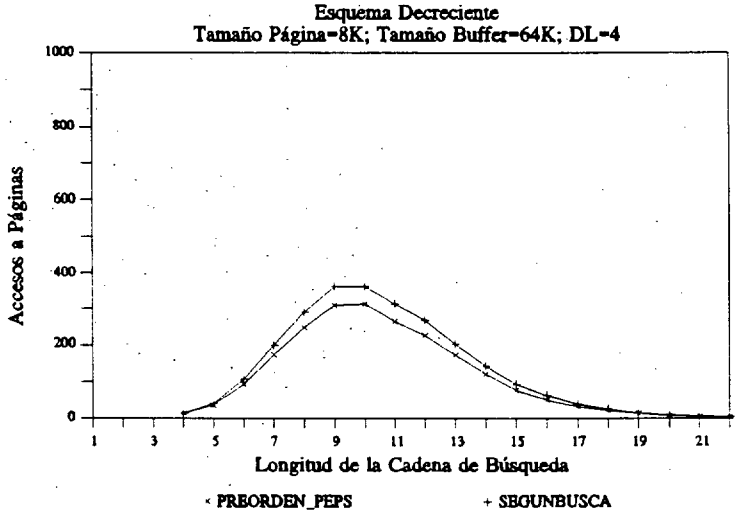


Figura 39

Como conclusión se obtiene que la paginación en *preorden* es la más adecuada.

Capítulo 5.
Aplicación de la Estructura S-D a la Recuperación de Información en Archivos Documentales

La localización de palabras en textos constituye una parte fundamental de un problema práctico de gran importancia: la organización, el almacenamiento y los sistemas de recuperación apropiados para el manejo de la información procedente de fuentes heterogéneas.

Existen dos aproximaciones a la localización de palabras en textos. En la primera, se efectúa la búsqueda directamente sobre el documento sin ningún tipo de preparación preliminar; si se trata con actas de longitud mediana o las consultas son algo complejas, esta manera de proceder puede resultar muy costosa en cuanto a tiempo de respuesta. Desde la segunda perspectiva, se someten previamente los escritos a un tratamiento; la intención es generar un índice que aumente la eficacia de los accesos al ejemplar.

Siguiendo este último enfoque, se utiliza como índice la estructura **S-D**, construida a partir del conjunto de las palabras diferentes que se obtienen del documento al descartar las palabras consideradas *vacías*.

El corpus que se va a indizar para llevar a cabo las localizaciones textuales es un Diccionario de Medicina, [JV87].

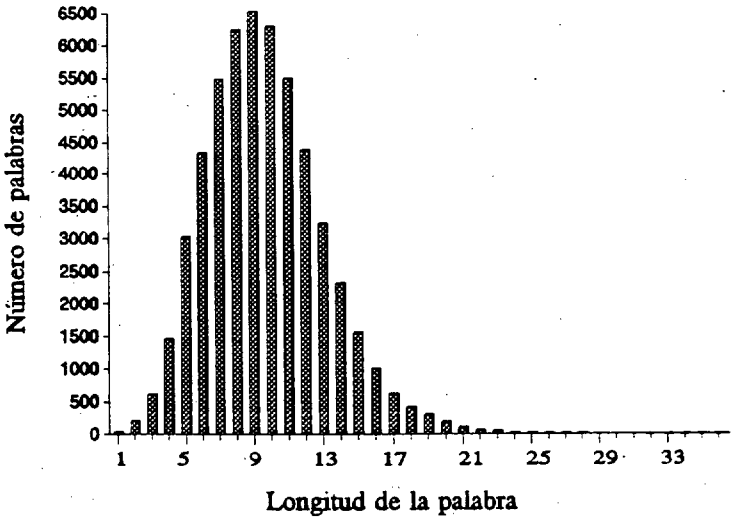


Figura 40

El Diccionario de Medicina constituye un texto formado por un total de 1.054.039 palabras de las cuales 603.152 se consideran *vacías* —584 es la cardinalidad de este conjunto [1]— y de las 450.887 restantes, solamente existen 56.297 palabras diferentes. Se considera que la tilde y la diéresis no suponen, a efectos de evaluación de **DIT** y **DL**, costo de edición alguno por tanto no se han tenido en cuenta los signos ortográficos en la construcción del índice, aunque en la respuesta a una petición se localiza la aparición, en el documento original, de todas las palabras con o sin signos ortográficos. En la figura 40, se muestra la distribución de frecuencias por longitudes de las 54.723 palabras diferentes consideradas para construir **S-D**. Los caracteres numéricos y obviamente los signos de puntuación no son considerados palabras y por ello no se admiten en las peticiones.

Se modifican los *nodos_SIT* añadiendo un puntero a cada *sinónimo_DIT*, de forma que señale a una lista —ordenada de menor a mayor— de posiciones en las que aparece esa palabra en el documento original, como se observa en la figura 41.

[1] En el Apéndice 1 aparece una lista de las mismas.

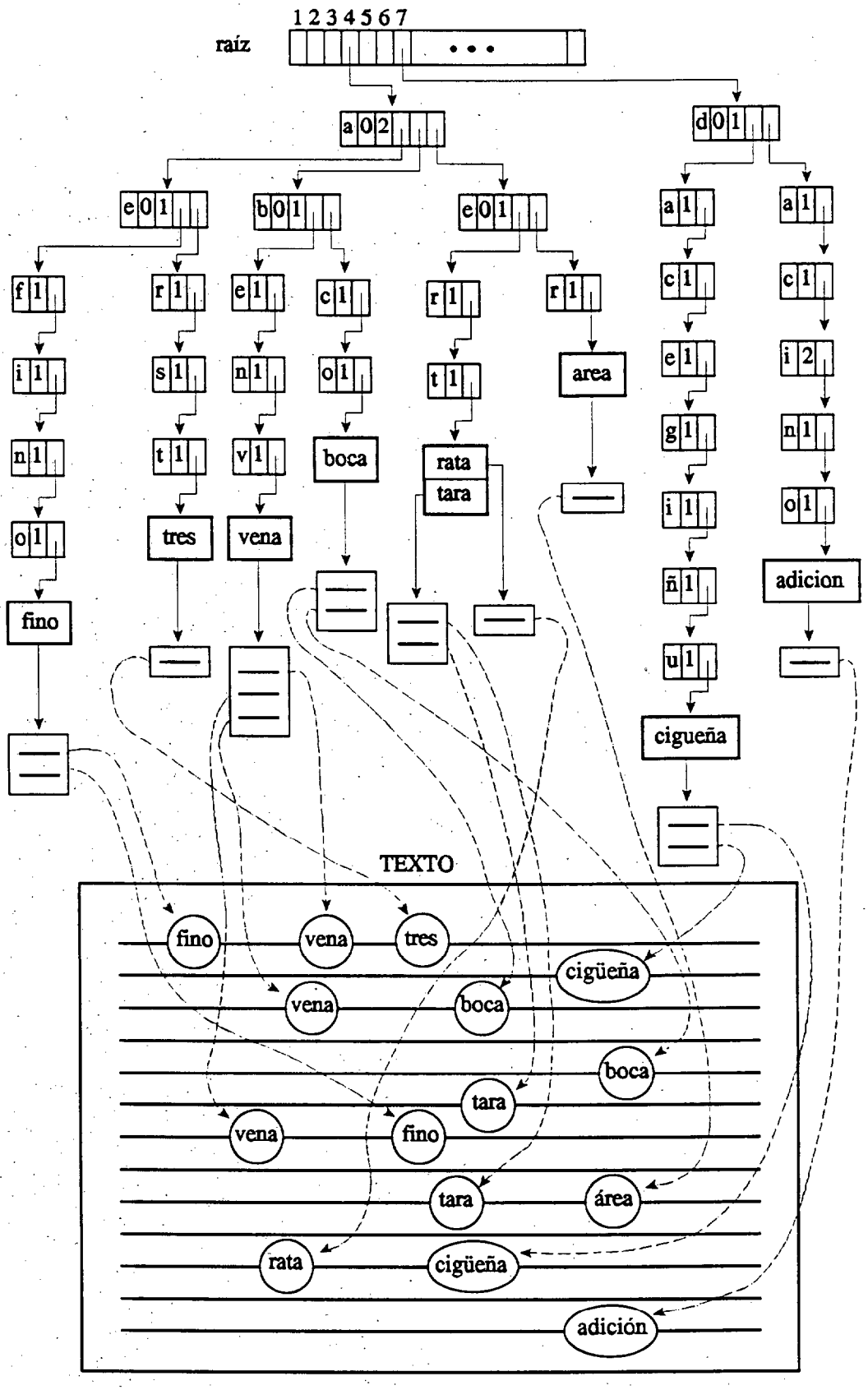


Figura 41

Se estudiarán dos casos: en primer lugar cuando se dispone de todo el índice en memoria interna y a continuación cuando se hace necesaria la paginación de la estructura.

5.1. Tipos de Búsquedas en Archivos Documentales.

Sobre la estructura **S-D** así construida se permiten los siguientes tipos de búsquedas:

— *exacta*

— *más similares*

— *máscaras*

— *truncamientos*: *a la derecha, a la izquierda, a ambos lados*

— *con operadores booleanos*: *o, y, y_no.*

— *cercanía*

— *antecedencia*

— *párrafos*

— *sentencias*

— *frases*

— *compleja*

El Diccionario de Medicina está dividido de forma natural en *artículos* —en general *documentos*— donde cada uno se compone de una entrada y su definición. La lista de posiciones asociada a cada *sinónimo_DIT* indica los distintos *artículos* en los que está presente. Si una palabra se repite dentro de un mismo *artículo*, su lista de posiciones hace referencia una sola vez a este *artículo*.

Las búsquedas de: *cercanía*, *antecedencia*, *párrafos*, *sentencias* y *frases*, así como las *complejas* que incluyan a alguna de las citadas, requieren el tratamiento de cada *artículo* candidato al conjunto respuesta con el fin de verificar las restricciones especificadas en la petición en cuanto a la situación de las palabras.

Cuando se solicita una búsqueda del tipo: *más similares*, *máscaras* o *truncamientos*, en las que el usuario no determina directamente las palabras a localizar, se muestran en la pantalla las palabras que satisfacen la petición permitiéndose la selección de un subconjunto de ellas; se funden entonces las listas de posiciones asociadas a las palabras seleccionadas para posteriormente visualizar los *artículos* correspondientes.

5.2. Búsqueda Exacta.

La búsqueda más elemental consiste en determinar todas las localizaciones de una palabra en el documento. La petición se expresa simplemente indicando cuál es la que se solicita. Por ejemplo: ***sida***, aparece en ocho *artículos*.

Algoritmo de búsqueda exacta:

```
Procedimiento Exacta (nodo)
{Obtiene la lista de posiciones asociada a la palabra de búsqueda}
    B: palabra de búsqueda
    encontrado: es falso cuando se invoca a este procedimiento; si se localiza B, se
    actualiza a verdadero
    frec: frecuencia en B del carácter asociado a nodo
si nodo ≠ nulo entonces
    si nodo es un nodo_SIT entonces
        j=1
        mientras (j≤nodo.ns) y (no encontrado) hacer
            si B = nodo.s(j) entonces
                encontrado=verdadero
                obtener la lista de posiciones asociada a nodo.s(j)
            si no
                j=j+1
            fin si
        fin mientras
    si no
        si es un nodo de la parte cadena entonces
            si vf(nodo.c) = nodo.f entonces
                Exacta(nodo.e)
            fin si
        si no {es un nodo de la parte_árbol}
            frec=vf(nodo.cd)
            si (nodo.pp≤frec) y (frec≤nodo.pu) entonces Exacta(nodo.e(frec)) fin si
        fin si
    fin si
fin si
```

5.3. Búsqueda de las Palabras Más Similares.

La recuperación de las palabras *más similares* a una dada se realiza con el esquema de búsqueda *creciente* cuando se dispone de todo el índice en memoria interna y con el *decreciente* cuando se trabaja con la estructura paginada, ya que, en general, son los que presentan mejor rendimiento para cada una de esas situaciones.

El formato de la petición es: ***+cadena***. Por ejemplo, al solicitar ***+rida*** se obtienen: *ría*, *vida*, *rica*, *risa*, *sida*, *oída*, *brida*.

5.4. Búsqueda con Máscaras.

Si se desea hacer una búsqueda de una palabra en la que se desconocen caracteres en determinadas posiciones, se puede formular una petición en la que se sustituyan los caracteres desconocidos por *comodines*, *.

En una búsqueda con *máscaras*, se recuperan todos los *artículos* en los que se encuentra una palabra de longitud igual a la especificada y que difiera solamente en los caracteres indicados por los asteriscos. Por ejemplo para ***t*m*r*** se recuperan los *artículos* en los que aparecen las palabras *tumor*, *temor*, *tomar*.

Una búsqueda con *máscaras* se lleva a cabo explorando el subárbol correspondiente a la longitud de la petición, teniendo como objetivo inicial la recuperación de todas las palabras cuya **DIT** con la palabra de búsqueda es igual al número de asteriscos (**DTM**=número de asteriscos).

El vector de frecuencias, **vf**, contiene la frecuencia de cada carácter alfabético especificado en la petición y cero para cualquier otro. En cada nodo de la *parte_árbol*, la frecuencia en **vf** del carácter asociado al nodo define el encadenamiento de descenso inicial; las alternativas se toman a partir de este punto sólo hacia la derecha, mientras no se supere el umbral **DTM**. Cada vez que se alcanza un

nodo_SIT, se comprueba si cada sinónimo tiene los caracteres especificados en la petición en las posiciones correspondientes.

Algoritmo de búsqueda con máscaras:

```
Procedimiento Máscara (B)
{Realiza una búsqueda con máscaras}
    B: cadena de búsqueda, formada por caracteres del alfabeto y asteriscos
    vf(.): vector de frecuencias de los caracteres de B, excluyendo los asteriscos
    DTM: número de asteriscos de la petición
    nms: número de palabras que verifican la petición
para c = 'a' hasta 'z' hacer
    vf(c) = 0
fin para
DTM = 0
para i = 1 hasta | B | hacer
    si B<i> = '*' entonces
        DTM = DTM + 1
    si no
        vf(B<i>) = vf(B<i>) + 1
    fin si
fin para
nms = 0
cadit = 0
Masc_pa(raiz( | B | ),cadit, | B | -DTM)
```

```
Procedimiento Masc_pa (nodo,cadit,cbnt)
{Explora la parte_árbol de la estructura S-D}
    nodo: nodo actual
    cadit: componentes acumuladas de DIT
    cbnt: número de caracteres de B no tratados
    frec: frecuencia en B del carácter asociado a nodo
    dfm: diferencia de frecuencias máxima
    finf, fsup: frecuencia mínima y máxima alcanzadas en la selección de alternativas
    falt: frecuencia alternativa
    vcadit: valor de las componentes de DIT para la alternativa actual
    ditm: DIT mínima utilizada en la poda
si nodo ≠ nulo entonces
    si nodo es un nodo_SIT entonces
        cadit = cadit + cbnt
        si cadit ≤ DTM entonces
            Masc_nS(nodo)
        fin si
    si no
        si es un nodo de la parte cadena entonces
            Masc_pc(nodo,cadit,cbnt)
        si no  {es un nodo de la parte_árbol}
            frec = vf(nodo.cd)
            cbnt = cbnt-frec
            dfm = DTM-cadit
            finf = frec
            fsup = frec + dfm
            si finf < nodo.pp entonces
                finf = nodo.pp
            fin si
            si fsup > nodo.pu entonces
                fsup = nodo.pu
            fin si
            para falt = finf hasta fsup hacer
                vcadit = cadit + abs(frec-falt)
                Masc_pa(nodo.e(falt),vcadit,cbnt)
            fin para
        fin si
    fin si
fin si
```

```
Procedimiento Masc_nS (nodo)
{Trata los nodos_SIT}
para j = 1 hasta nodo.ns hacer
    i = 1
    igualcar = verdadero
    mientras (i ≤ | B | ) y igualcar hacer
        si (B<i> ≠ '*') y (B<i> ≠ nodo.s(j)<i>) entonces
            igualcar = falso
        si no
            i = i + 1
        fin si
    fin mientras
    si i > | B | entonces
        añadir nodo.s(j) al conjunto de palabras que verifican la petición
        nms = nms + 1
    fin si
fin para
```

```
Procedimiento Masc_pc (nodo,cadit,cbnt)
{Examina la parte_cadena}
si nodo es un nodo_SIT entonces
    cadit = cadit + cbnt
    si cadit ≤ DTM entonces
        Masc_nS(nodo)
    fin si
si no
    cadit = cadit + abs(nodo.f-vf(nodo.c))
    cbnt = cbnt-vf(nodo.c)
    si cadit ≤ DTM entonces
        Masc_pc(nodo.e,cadit,cbnt)
    fin si
fin si
```

5.5. Búsquedas con Truncamientos.

Las búsquedas con *truncamientos: a la derecha, a la izquierda* y *a ambos lados* permiten recuperar *artículos* que contienen palabras que empiezan,

terminan o presentan alguna coincidencia central con la cadena de caracteres sobre el alfabeto, *cadena*, explicitada en la petición.

Cada tipo de *truncamiento* se expresa de la siguiente forma:

a la derecha: ***cadena!***

a la izquierda: ***!cadena***

a ambos lados: ***!cadena!***

Por ejemplo:

a la derecha: ***tos!***, se obtienen: *tos*, *tose*, *tosa*, *toser*, *tosco*, *tosca*, *toscos*, *toscas*, *tostada*.

a la izquierda: ***!tipo***, se recuperan: *subtipo*, *ojotipo*, *idiotipo*, *optotipo*, *fenotipo*, *genotipo*, *serotipo*, *fagotipo*, *prototipo*, *cariotipo*, *inmunotipo*, *somatotipo*.

a ambos lados: ***!cubo!***, se localizan: *cubo*, *cuboide*, *cuboides*, *cuboideo*, *cuboidal*, *cuboidea*, *cuboideum*, *cuboideos*, *cuboideas*, *calcaneocuboidea*, *calcaneocuboideo*, *calcaneocuboideas*.

Para realizar la búsqueda correspondiente a cualquiera de los tres tipos de *truncamientos* se examinan las palabras con longitud igual o mayor a la cadena especificada en la petición. La exploración de cada longitud se realiza aprovechando el procedimiento de búsqueda con *máscaras* —donde el número de comodines depende de la diferencia de longitudes— y modificando el tratamiento de los *nodos_SIT*.

Algoritmo de búsqueda con truncamiento a la derecha:

```
Procedimiento Truncamiento (B)
    B: cadena de búsqueda especificada en la petición
    vf(.): vector de frecuencias de la cadena de búsqueda
    DTM: diferencia entre la longitud actual y la de la cadena de búsqueda
para c = 'a' hasta 'z' hacer
    vf(c) = 0
fin para
para i = 1 hasta | B | hacer
    vf(B<i>) = vf(B<i>) + 1
fin para
DTM = 0
nms = 0
para lalt = | B | hasta longmax hacer
    cadit = 0
    Masc_pa(raiz(lalt),cadit, | B | )
    DTM = DTM + 1
fin para
```

El procedimiento **Masc_pa**, en este caso, en vez de invocar al procedimiento **Masc_nS** llama al **Truncder_nS** que selecciona los *sinónimos_DIT* que poseen la cadena de búsqueda como prefijo.

```
Procedimiento Truncder_nS (nodo)
{Trata los nodos_SIT en una búsqueda con truncamiento a la derecha}
para j = 1 hasta nodo.ns hacer
    i = 1
    mientras (i ≤ | B | ) y (B<i> = nodo.s(j)<i>) hacer
            i = i + 1
    fin mientras
    si i > | B | entonces
        añadir nodo.s(j) al conjunto de palabras que verifican la petición
        nms = nms + 1
    fin si
fin para
```

Algoritmo de búsqueda con truncamiento a la izquierda:

El algoritmo de *truncamiento a la izquierda* sólo se diferencia del anterior en que invoca al procedimiento **Truncizq_nS** —selecciona los *sinónimos_DIT* que poseen la cadena de búsqueda como subfijo— en vez de al **Truncder_nS**.

```
Procedimiento Truncizq_nS (nodo)
{Trata los nodos_SIT en una búsqueda con truncamiento a la izquierda}
para j = 1 hasta nodo.ns hacer
   i = | B |
   mientras (i > 0) y (B<i> = nodo.s(j) < lalt- | B | +i>) hacer
         i = i-1
   fin mientras
   si i = 0 entonces
      añadir nodo.s(j) al conjunto de palabras que verifican la petición
      nms = nms + 1
   fin si
fin para
```

Algoritmo de búsqueda con truncamiento a ambos lados:

Se utiliza el procedimiento **Truncamiento** con un nuevo tratamiento de los *nodos_SIT*, **Truncamb_ns**; selecciona los *sinónimos_DIT* que poseen la cadena de búsqueda como infijo —busca la primera coincidencia entre el sinónimo y la cadena de búsqueda.

```
Procedimiento Truncamb_nS (nodo)
{Trata los nodos_SIT en una búsqueda con truncamiento a ambos lados}
para j = 1 hasta nodo.ns hacer
   t = lalt- | B |
   r = 0
   valido = falso
   mientras (r ≤ t) y (no valido) hacer
      i = 1
      mientras (i ≤ | B | ) y (B<i> = nodo.s(j) < r + i >) hacer
         i = i + 1
      fin mientras
      si i > | B | entonces
         añadir nodo.s(j) al conjunto de palabras que verifican la petición
         nms = nms + 1
         valido = verdadero
      si no
         r = r + 1
      fin si
   fin mientras
fin para
```


5.6. Búsquedas con Operadores Booleanos.

Se pueden realizar búsquedas que combinen varias cadenas utilizando como conectores los operadores lógicos: *o*, *y*, *y_no*.

5.6.1. Búsqueda Disyuntiva.

Una búsqueda *disyuntiva* proporciona todos los *artículos* en los que aparece al menos una de las palabras solicitadas en la petición. Se solicita de la siguiente forma: ***cadena1 o cadena2***, donde *cadena1* y *cadena2* son peticiones *exactas*, *más similares*, *máscaras* o *truncamientos*. Por ejemplo: ***cáncer o +tos***.

Este proceso se realiza invocando al procedimiento de búsqueda correspondiente a cada petición especificada uniendo posteriormente las listas de posiciones asociadas.

Algoritmo de unión de las listas de posiciones:

```
Procedimiento Unión (list1,list2,long1,long2)
{Une dos listas de posiciones}
    long1, long2: longitudes de las listas de posiciones
    list1(.), list2(.): listas de posiciones que se van a unir
    list3(.): lista en la que se almacena ordenadamente la fusión
    nrd: número de elementos en list3
i=1; j=1; nrd=0
mientras (i≤long1) y (j≤long2) hacer
    si list1(i) < list2(j) entonces
        nrd=nrd+1
        list3(nrd)=list1(i)
        i=i+1
    si no
        nrd=nrd+1
        list3(nrd)=list2(j)
        si list1(i) = list2(j) entonces i=i+1 fin si
        j=j+1
    fin si
fin mientras
mientras i ≤ long1 hacer
    nrd=nrd+1
    list3(nrd)=list1(i)
    i=i+1
fin mientras
mientras j ≤ long2 hacer
    nrd=nrd+1
    list3(nrd)=list2(j)
    j=j+1
fin mientras
```

Análisis del algoritmo en el peor caso:

La cantidad total de tiempo que consume este algoritmo está en relación directa con el número de comparaciones que realiza. El peor caso ocurre cuando para situar cada elemento de la segunda lista en la de fusión se requieren dos comparaciones y para

situar cada elemento de la primera, excepto el último, es necesaria una comparación. Esta situación se verifica cuando las posiciones de las dos listas a fundir son todas distintas y no se agota ninguna de ellas hasta el último instante. Si se desean unir s listas, se invoca al procedimiento para las dos primeras y se mezcla cada una de las *s-2* restantes con el resultado previo. El número máximo de comparaciones en función del número de listas se muestra en la tabla 4 —se denota por n_i la longitud de la i-ésima lista— donde se ha supuesto sin pérdida de generalidad que la primera lista se corresponde con la fusión previa y la segunda con la nueva lista a mezclar.

Número de listas	Número máximo de comparaciones
2	n_1+2*n_2-1
3	$2*n_1+3*n_2+2*n_3-2$
4	$3*n_1+4*n_2+3*n_3+2*n_4-3$
5	$4*n_1+5*n_2+4*n_3+3*n_4+2*n_5-4$
...	...
s	$(s-1)*n_1+s*n_2+(s-1)*n_3+\ldots+2*n_s-(s-1)$

Tabla 4

Dado que:

$$(s-1)*n_1+s*n_2+(s-1)*n_3+\ldots+2*n_s-(s-1) < s*(n_1+n_2+n_3+\ldots+n_s) < s^2*n$$

siendo

$$n = máx\{n_1, n_2, n_3,\ldots, n_s\} \text{ y } s \geq 2$$

se tiene que la complejidad de cálculo en el peor caso es $O(s^2*n)$. ◆

El análisis realizado indica que para obtener un mejor rendimiento es conveniente fundir las listas en orden creciente de longitudes —los coeficientes que afectan a las longitudes son decrecientes a partir de la segunda lista.

Se ha obtenido que las listas de posiciones de la estructura tienen una longitud promedio de 7; la máxima que asciende a 724 y está asociada a la palabra *sangre*.

5.6.2. Búsqueda Conjuntiva.

La respuesta de una petición *conjuntiva* está formada por aquellos *artículos* que contienen una de las palabras solicitadas en la primera petición y otra de la segunda. Esta petición se formula como: ***cadena1 y cadena2***, donde *cadena1* y *cadena2* son peticiones *exactas*, *más similares*, *máscaras* o *truncamientos*. Por ejemplo: ***cáncer y tos!***.

Esta búsqueda se lleva a cabo realizando la intersección de las listas de posiciones que proporciona cada una de las peticiones especificadas.

Algoritmo de intersección de las listas de posiciones:

```
Procedimiento Intersección (list1,list2,long1,long2)
{Realiza la intersección de dos listas de posiciones}
    long1, long2: longitudes de las listas de posiciones
    list1(.), list2(.): listas de posiciones a intersectar. El resultado se almacena en list1
    nrc: número de elementos de la intersección
j=1; nrc=0
mientras (i≤long1) y (j≤long2) hacer
    si list1(i)>list2(j) entonces
        j=j+1
    si no
        si list1(i) = list2(j) entonces
            nrc=nrc+1
            list1(nrc)=list1(i)
            j=j+1
        fin si
        i=i+1
    fin si
fin mientras
long1=nrc
```

Análisis del algoritmo en el peor caso:

De forma análoga al anterior, el tiempo que requiere este algoritmo está en relación directa con el número de comparaciones. Cuando todos los elementos de la primera lista están situados al final de la segunda, se alcanza el peor caso, ya que se realiza una comparación para cada elemento no común de la segunda lista y dos comparaciones para los comunes. La intersección de s listas se lleva a cabo invocando al procedimiento para las dos primeras y cada una de las *s-2* restantes se intersectan con el resultado previo. La tabla 5 muestra el número máximo de comparaciones en función del número de listas a tratar.

Número de listas	Número máximo de comparaciones
2	n_2+n_1
3	$n_2+n_3+2*n_1$
4	$n_2+n_3+n_4+3*n_1$
5	$n_2+n_3+n_4+n_5+4*n_1$
...	...
s	$n_2+n_3+\ldots+n_s+(s-1)*n_1$

Tabla 5

Dado que:

$$n_2+n_3+\ldots+n_s+(s-1)*n_1 < (s-1)*n+(s-1)*n < 2*(s-1)*n < 2*s*n$$

siendo

$$n = máx\{n_1, n_2, n_3,\ldots, n_s\} \text{ y } s \geq 2$$

se tiene que la complejidad de cálculo en el peor caso es $O(s*n)$. ◆

El análisis refleja que si la primera lista es la más corta se obtiene un mejor rendimiento —es la única longitud afectada por un coeficiente mayor que uno.

5.6.3. Búsqueda con Operador *y_no*.

En una búsqueda de una petición de la forma ***cadena1 y_no cadena2*** se obtienen aquellos *artículos* en los que se encuentra alguna palabra solicitada en la petición situada a la izquierda del operador y que a su vez no contienen a ninguna palabra de las solicitadas a la derecha. Por ejemplo: ***tos y_no fiebre***.

La forma de conseguir los *artículos* deseados consiste en eliminar de la lista de posiciones de la izquierda las que se encuentren en la lista de la derecha.

Algoritmo de diferencia de listas de posiciones:

```
Procedimiento Diferencia (list1,list2,long1,long2)
{Elimina de list1 las posiciones que se encuentren en list2}
    long1, long2: longitudes de las listas de posiciones
    list1(.), list2(.): listas de posiciones a intersectar. El resultado se almacena en list1
    nrc: número de elementos de la intersección
i=1; j=1; nrc=1
mientras (i≤long1) y (j≤long2) hacer
    si list1(i)>list2(j) entonces
        j=j+1
    si no
        si list1(i) ≠ list2(j) entonces
            list1(nrc)=list1(i)
            nrc=nrc+1
        si no
            j=j+1
        fin si
        i=i+1
    fin si
fin mientras
mientras i ≤ long1 hacer
    list1(nrc)=list1(i)
    nrc=nrc+1
    i=i+1
fin mientras
long1=nrc
```

Análisis del algoritmo para el peor caso:

De forma análoga a los anteriores, el tiempo que requiere este algoritmo está en relación directa con el número de comparaciones. Sean $\mathbf{n_1}$ y $\mathbf{n_2}$ las longitudes respectivas de las listas. Si $n_1 \geq n_2$, cuando todos los elementos de la segunda lista están situados al final de la primera, se alcanza el peor caso, ya que se realizan dos comparaciones por cada elemento de la primera lista, $2*n_1$; si $n_1 < n_2$, se alcanza el peor caso cuando los elementos de la primera lista se encuentran situados al final de la segunda, ya que se realiza una comparación para los elementos no comunes de la segunda lista y dos para los comunes, $n_2-n_1+2*n_1 < 2*n_2$. En cualquier caso, el número de comparaciones está acotado por $2*n$, siendo $n=máximo\{n_1,n_2\}$, por tanto la complejidad de cálculo, en el peor caso, es $O(n)$. ♦

5.7. Búsquedas de Cercanía y Antecedencia.

Se denominan búsquedas con condiciones topológicas ya que delimitan el *entorno*, medido en número de palabras, en el que se tienen que encontrar las palabras especificadas en la petición.

5.7.1. Cercanía.

La búsqueda de *cercanía* se caracteriza por limitar el número máximo de palabras que pueden existir entre las dos especificadas. Las peticiones se expresan

como: ***cadena1 c/n cadena2,*** donde *cadena1* y *cadena2* deben ser peticiones *exactas* y **n** es un entero positivo que indica la posición más alejada posible desde una cadena a la otra. Por ejemplo: ***fiebre c/5 aguda***, seleccionará aquellos *artículos* en los que se han localizado *fiebre* y *aguda*, verificándose además que entre ellas existen como mucho *n-1* palabras.

Este proceso se realiza eliminando aquellos *artículos* que no cumplen las condiciones de *cercanía* de la lista generada por la búsqueda *conjuntiva* de las dos palabras especificadas.

5.7.2. Antecedencia.

La *antecedencia* presenta una restricción adicional con respecto a la búsqueda anterior ya que establece además el orden de aparición entre las palabras. Las peticiones tienen la forma: ***cadena1 a/n cadena2***. Por ejemplo: ***nervio a/3 facial***, selecciona los *artículos* en los que aparece *nervio* seguida de *facial* y entre ellas no hay más de dos palabras.

5.8. Búsquedas en Párrafos y Sentencias.

En las búsquedas anteriores, el *artículo* es el ámbito de búsqueda por defecto, se puede restringir dicho ámbito a un *párrafo* o a una *sentencia*..

5.8.1. Párrafos.

La búsqueda en ***párrafos*** es una alternativa a la de ***cercanía*** entre palabras en la que se requiere que las palabras especificadas se encuentren en el mismo *párrafo*. Se solicita de la siguiente forma: ***cadena1 p/ cadena2***. Por ejemplo: ***pies p/ manos***, selecciona aquellos *artículos* en los que aparecen, en un mismo *párrafo*, las palabras *pies* y *manos*.

Este proceso se lleva a cabo de forma análoga a la búsqueda de ***cercanía***.

5.8.2. Sentencias.

Es como la búsqueda en ***párrafos*** pero restringida a una *sentencia*. La petición se expresa: ***cadena1 s/ cadena2***. Por ejemplo: ***pies s/ manos***, selecciona aquellos *artículos* en los que aparecen, en la misma *sentencia*, las palabras *pies* y *manos*.

5.9. Búsqueda de Frases.

El objetivo de la búsqueda de ***frases*** es la localización de los *artículos* en los que se encuentra la ***frase*** especificada. La ***frase*** se denota entre comillas. Por ejemplo: ***"de pies y manos"***.

Este proceso se lleva a cabo mediante una búsqueda *conjuntiva* de las palabras no *vacías* de la *frase* y la ulterior comprobación de la existencia de toda la *frase* en los *artículos* preseleccionados.

5.10. Búsqueda Compleja.

Cualquier combinación de los anteriores tipos de búsqueda en una expresión utilizando los conectores lógicos —*y*, *o*, *y_no*— es a lo que se denomina una petición compleja. El orden de prioridad de los conectores es de izquierda a derecha. Si se desea alterar esta prioridad, se deben utilizar paréntesis y en el caso de que existan paréntesis anidados, los más internos son los que se resolverán en primer lugar. Por ejemplo: ***s**a y ((fiebre c/2 aguda) o tos!) y_no sana.***

La obtención de la respuesta de una búsqueda compleja con paréntesis anidados se realiza resolviendo las expresiones desde las más profundas hacia afuera. A partir del paréntesis abierto, se actúa de izquierda a derecha ejecutando cada búsqueda *no_booleana* y resolviendo los *operadores lógicos* una vez que se alcanza el paréntesis cerrado. Por defecto, se considera que la petición *compleja* está encerrada entre paréntesis.

5.10.1. Búsqueda en la que intervienen Peticiones Anteriores.

Iniciada una sesión de consultas y tras haber realizado una serie de peticiones, es posible modificar una petición *previa* para confeccionar otra más *compleja*. El formato ***@n*** indica que la **n**-ésima consulta realizada forma parte de la nueva petición. Supóngase que se desea añadir a la quinta petición, cuya expresión de contenido es: ***cáncer y tumor***, un nuevo término: ***y_no pulmón***, la nueva petición se escribe de la siguiente forma: ***@5 y_no pulmón***.

También se pueden hacer *combinaciones de peticiones previas* sin añadir ningún término nuevo, utilizando los *operadores booleanos*. Por ejemplo: ***@2 y (@4 o @6)***, combina los resultados de las consultas segunda, cuarta y sexta.

5.11. Analizador Sintáctico de la Petición.

El *analizador sintáctico* cumple un doble cometido, ya que ha de identificar cada tipo de petición —diferenciando las componentes y los *conectores lógicos* en el caso de las *complejas*— y, a la vez, determinar la *correctitud sintáctica*.

Se consideran erróneas las siguientes situaciones:

— Si alguna petición *no_booleana* presenta un formato incorrecto. Ya sea porque se especifica una búsqueda no permitida —por ejemplo: ***t*m!***, ***+rida c/9 tos!***—; o bien porque la sintaxis es incompleta —por

ejemplo: si se detecta la presencia de / precedida de **a** o **c** y no le sigue un valor entero o se especifica una *frase* y falta cerrar las comillas, etc.

— Si se utilizan *conectores no permitidos*.

— Si después de un *operador lógico* no existe una búsqueda *no_booleana* o un paréntesis abierto.

— Si se especifica la búsqueda *exacta* de una palabra considerada como *vacía*.

— Si existe un paréntesis abierto para el que no se encuentra el correspondiente paréntesis cerrado.

En caso de que se detecte un error del usuario, se informará al mismo de ello y se señalará en la pantalla la posición en la que se ha encontrado el fallo.

Para *analizar* la petición se recorre de izquierda a derecha identificando las diversas búsquedas *no_booleanas*, *operadores lógicos* y paréntesis, comprobándose que no ocurre ninguna de las situaciones indicadas anteriormente.

Una vez que se ha verificado que la petición es *sintácticamente correcta* e identificado sus componentes, se pasa a la fase de ejecución de la búsqueda.

5.12. Optimizador de la Petición.

Se construye un *optimizador* como complemento al *analizador*. La idea es llegar a la solución de una búsqueda *compleja* en el menor tiempo posible. Se

pretende, siempre que sea factible, calcular una *petición equivalente* a la solicitada por el usuario de forma que, aunque los resultados obtenidos a partir de ambas sean los mismos, esta última *petición* se resuelva de forma más rápida, minimizando así el tiempo de respuesta del sistema.

Por una parte existen los procedimientos comentados con anterioridad que agilizan la ejecución de peticiones del tipo:

cadena1 o cadena2 o cadena3 o cadena4

cadena1 y cadena2 y cadena3 y cadena4

seleccionando la lista de posiciones más corta para comenzar la *unión* o la *intersección* de las mismas.

Adicionalmente se contempla la simplificación de aquellas peticiones en las que es aplicable la propiedad *distributiva* —teniendo en cuenta la *conmutatividad*—, por ejemplo:

(cadena1 y cadena2) o (cadena3 y cadena1)

se puede reducir a:

cadena1 y (cadena2 o cadena3)

y

(cadena1 o cadena2) y (cadena3 o cadena1)

se puede reducir a:

cadena1 o (cadena2 y cadena3)

donde los términos *cadena1*, *cadena2* y *cadena3* denotan cualquier tipo de búsqueda permitida.

5.13. Resultados Experimentales. Ubicación del Índice en Memoria Interna.

La ocupación del Diccionario de Medicina —1.054.039 palabras— es de 7.708.577 Bytes; la estructura **S-D** correspondiente ocupa 2.666.898 Bytes[2]; y las listas de posiciones, asociadas a cada una de las 54.273 palabras del índice, 1.550.740 Bytes.

El objetivo que se persigue es estudiar el comportamiento de la estructura **S-D** frente a las peticiones de respuesta múltiple, es decir, *truncamientos*, *máscaras* y *más similares*. Se mide, en función de la longitud de la cadena especificada, el tiempo requerido para cada tipo de búsqueda y, en cada caso, el número de palabras que verifican la petición —cardinalidad de la respuesta. El tiempo de búsqueda se mide desde el momento en que el usuario realiza la petición hasta que se muestran en pantalla las palabras que la satisfacen.

Inicialmente se comparan los tres tipos de *truncamientos*. Se generan 18 ficheros de 50 cadenas cada uno, desde longitud 2 hasta 19, para cada uno de los tipos de *truncamiento* con el fin de ser utilizados como argumento de búsqueda. Se elige una palabra al azar y se sortea entre uno y la mitad de su longitud para obtener el número de caracteres a eliminar; la longitud restante y la posición a partir de la que se

---

[2] Un carácter ocupa 1 Byte, un entero 4 Bytes y un puntero 4 Bytes.

extraen los caracteres (al principio, al final o a ambos lados) indican el fichero que le corresponde. En el caso del *truncamiento a ambos lados*, se ha de seleccionar el número de caracteres a eliminar por el principio y por el final, de tal forma que no coincida con ninguno de los otros dos tipos de *truncamiento*.

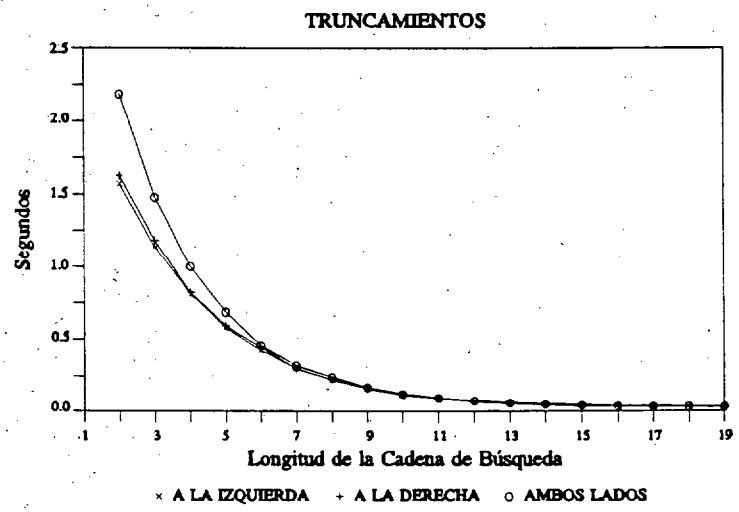


Figura 42

- El tiempo requerido para la búsqueda de los distintos tipos de *truncamientos*, figura 42, decrece al aumentar la longitud de la cadena de búsqueda debido a la disminución correlativa de la estructura que ha de ser explorada.
- El *truncamiento a ambos lados* presenta un tiempo ligeramente superior a los otros dos —prácticamente coincidentes— debido a que requiere un tratamiento algo más costoso de los *sinónimos_DIT*, cuya importancia crece en función del número de *nodos_SIT* visitados; la diferencia se acorta al crecer la longitud de la cadena de búsqueda, figura 42.

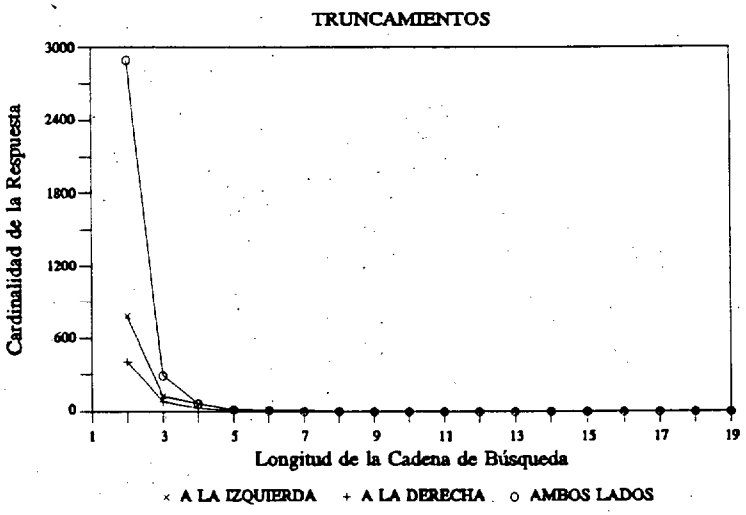


Figura 43

☞ La multiplicidad de la respuesta, figura 43, es significativa para longitudes menores que 10. De forma análoga al tiempo de respuesta, la cardinalidad decrece asintóticamente —hacia una única respuesta.

En el estudio de la búsqueda con *máscaras*, se utilizan 15 ficheros de 50 cadenas cada uno, de longitud 3 a 17. Para crearlos, se elige una palabra al azar —su longitud indica el fichero al que se va a asignar— se sortea el número de asteriscos entre uno y la mitad de su longitud, y por último, se seleccionan aleatoriamente las posiciones de éstos.

☞ La búsqueda con *máscaras* es la más rápida de los tres tipos de búsqueda con respuesta múltiple, puesto que realiza una exploración de la estructura limitada al subárbol correspondiente a la longitud de la cadena de búsqueda.

☞ El tiempo requerido en la búsqueda con *máscaras* está influido por el número de palabras cuya longitud coincide con la de la cadena de búsqueda como se muestra en la figura 44.

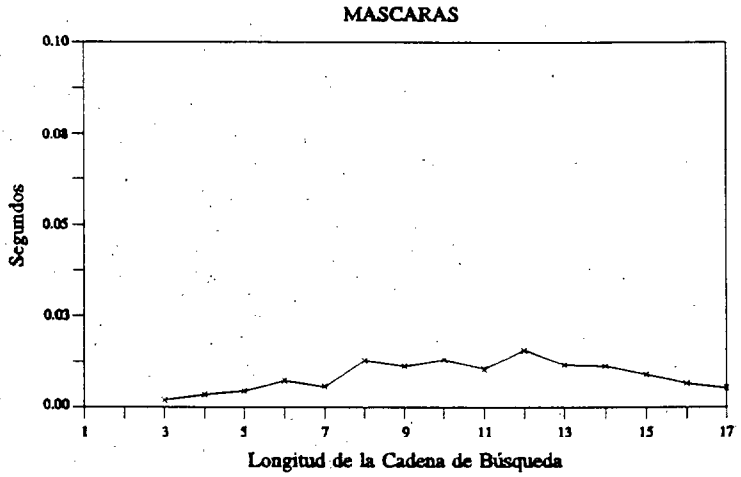


Figura 44

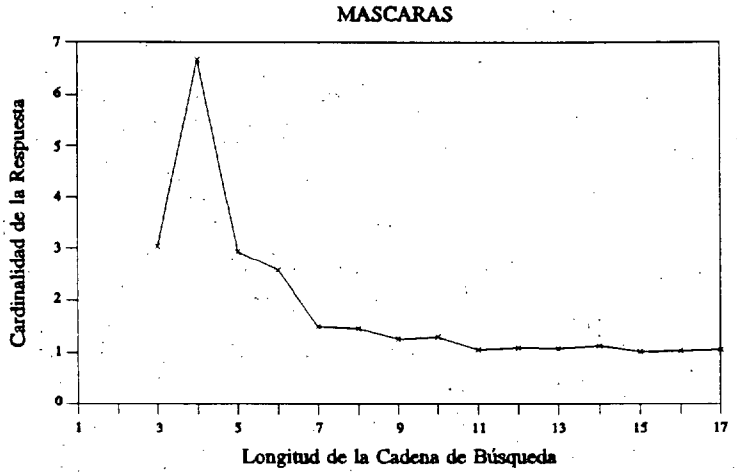


Figura 45

☞ La cardinalidad de la respuesta, figura 45, es mayor para pequeñas longitudes, haciéndose asintótica a uno con el incremento de la longitud.

Para la búsqueda de *más similares*, se selecciona al azar una palabra del Diccionario de Medicina y se transforma aplicándole diversas operaciones de edición, elegidas aleatoriamente, hasta obtener una cadena con una **DL** determinada —$DL=2$, $DL=4$ o $DL=6$— respecto a la palabra original. **DL** es como máximo igual a la mitad

de la longitud de la palabra correcta. La longitud de la palabra distorsionada y el valor de **DL** indican el fichero al que se asigna. Cada fichero contiene 50 cadenas.

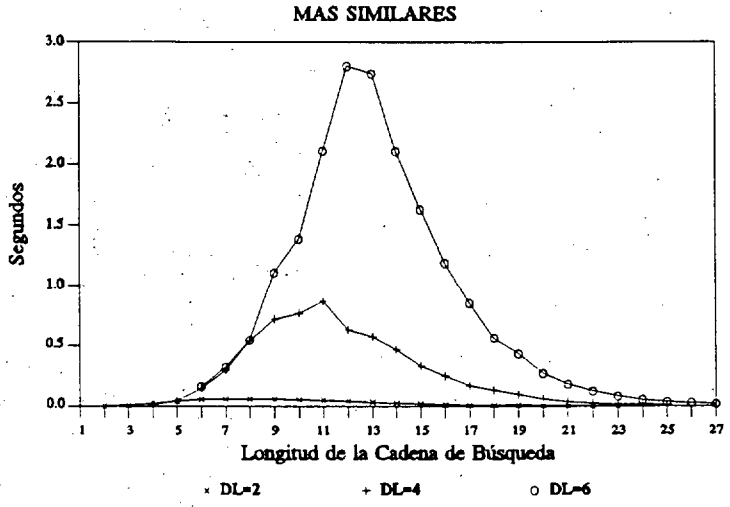


Figura 46

☞ La búsqueda de *más similares* consume un tiempo que está directamente relacionado con la parte de la estructura explorada y con el cálculo de **DL**. Las diferencias para los distintos valores de **DL** se deben tanto a la parte de la estructura implicada como al número de **DL** evaluadas, figura 46.

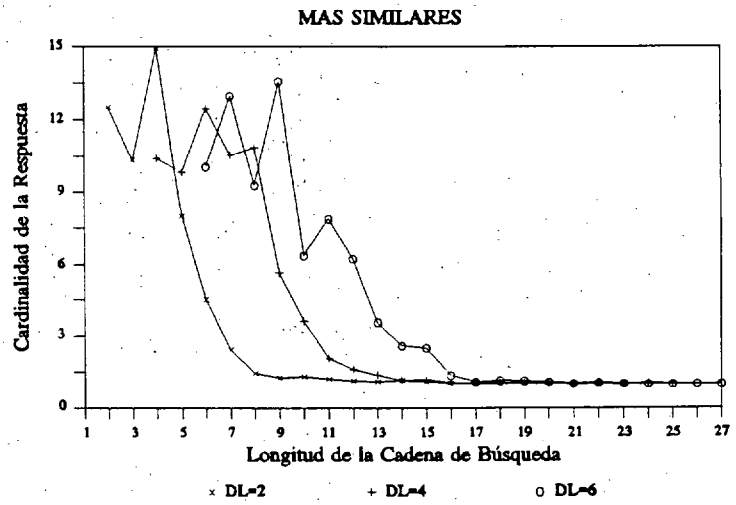


Figura 47

☞ La cardinalidad de la respuesta, figura 47, está determinada por la relación entre el valor de **DL** y la longitud de la cadena de búsqueda —encuentra un mayor número de palabras *más similares* a la cadena buscada para longitudes pequeñas y para mayores valores de **DL**—; sin embargo, el tiempo de búsqueda, figura 46, no depende de la cardinalidad de la respuesta sino del tiempo empleado en recorrer la estructura y en calcular **DL** —que no tiene relación significativa con la cardinalidad de la respuesta.

5.14. Resultados Experimentales. Índice Paginado.

Esta aplicación de la estructura **S-D** a la recuperación de información en archivos documentales se implanta en un **PC** 486 a 33mhz con disco a 19ms para hacerla accesible a un mayor número de usuarios. Para ello se utiliza la paginación en *preorden*, porque tal y como se ha comprobado es la más adecuada para el índice, con un *tamaño de página* de 4K —se obtiene una **S-D** de 538 páginas[3]— y un *tamaño de buffer* de 32K. La búsqueda de las palabras *más similares* se realiza con el esquema *decreciente* ya que en general, sobre la *estructura paginada*, presenta una mejor realización que el *creciente*.

[3] Un carácter ocupa 1 Byte, un entero 2 Bytes y un puntero 4 Bytes.

Inicialmente se repite el experimento realizado anteriormente en la memoria interna de un **HP9000-835**, con el fin de contrastar ésta realización con la obtenida utilizando memoria secundaria en el **PC**, es decir, se mide el tiempo requerido —en función de la longitud de la cadena de búsqueda— para cada tipo de búsqueda con respuesta múltiple, es decir, *truncamientos*, *máscaras* y *más similares*.

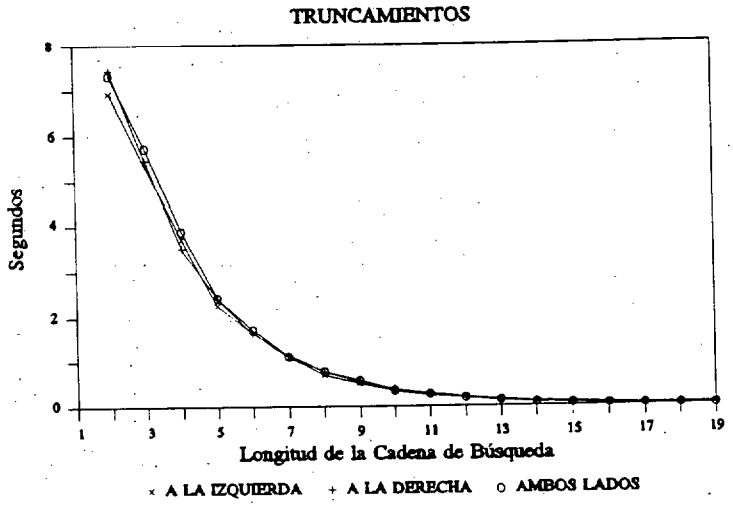


Figura 48

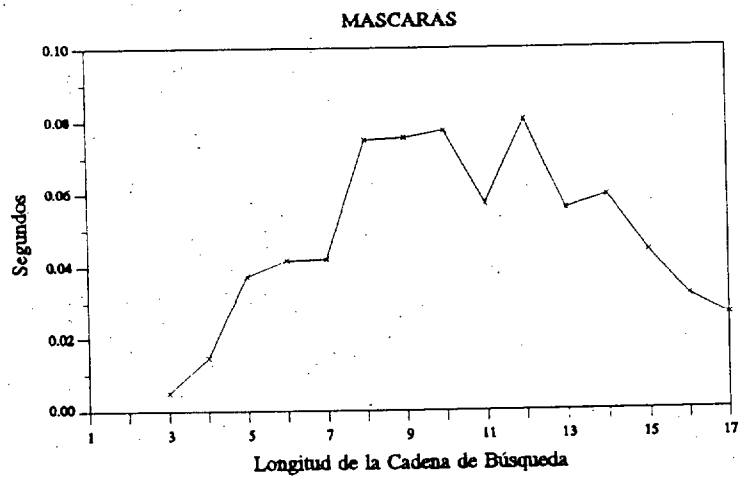


Figura 49

☞ Los resultados obtenidos en el **PC**, figuras 48, 49 y 50, son parecidos, en cuanto a la forma que presentan los tiempos de respuesta, a los

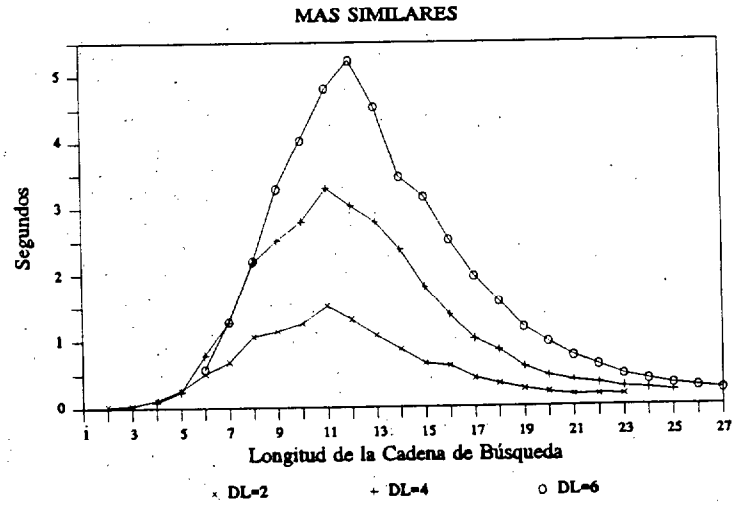


Figura 50

correspondientes obtenidos en el **HP9000**, figuras 42, 44 y 46. Como era de esperar, los valores son cuantitativamente mayores debido fundamentalmente a los accesos a disco.

La segunda parte de este estudio experimental tiene como objetivo estudiar el tiempo necesario para resolver las peticiones permitidas desde que se formulan hasta antes de comenzar a visualizar los *artículos*.

a) Búsqueda *exacta*.

Se selecciona de forma aleatoria una muestra de 100 palabras de entre todas las distintas —sin tener en cuenta las *vacías*— que existen en el Diccionario de Medicina. Se mide el tiempo transcurrido desde que se solicita cada petición de la muestra hasta que se obtiene la confirmación de que hay respuesta. Se calcula el promedio y la desviación típica del tiempo que se invierte en recuperar una palabra que está en el Diccionario de Medicina.

b) Búsqueda de las palabras *más similares* en función del valor de **DL**.

Este cálculo se realiza sobre dos muestras, una para $DL=2$ y otra para $DL=4$, de 100 cadenas cada una. Para obtener una cadena, se selecciona al azar una palabra de entre todas las distintas del Diccionario de Medicina y se transforma aplicándole diversas operaciones de edición, elegidas aleatoriamente, hasta obtener una palabra con una de las **DL** predeterminada respecto a la original —en todo caso se limita **DL** a la mitad de la longitud de la palabra correcta. Se miden, independientemente, el tiempo de recuperación de las palabras *más similares* y el lapso necesario para realizar la unión de las listas de posiciones para cada petición sobre cada una de las dos muestras. Se calculan los promedios y las desviaciones típicas de cada uno de los tiempos medidos para los distintos valores de **DL** bajo las condiciones descritas anteriormente.

c) Búsqueda con *truncamientos*.

Se construyen tres muestras, una para cada tipo de *truncamiento*, de 100 peticiones cada una —que respeten al menos el 50% de los caracteres. Se elige al azar una palabra de entre todas las distintas del Diccionario de Medicina, se sortea entre uno y la mitad de su longitud para obtener el número de caracteres a eliminar, las posiciones de extracción de los caracteres (al principio, al final o a ambos lados) indican la muestra que le corresponde. Para el *truncamiento a ambos*

lados se selecciona aleatoriamente el número de caracteres a eliminar por el principio y por el final, no permitiéndose que estén todos a un mismo lado. Se mide, por una parte el tiempo de recuperación de las palabras que verifican la petición y por otra, el lapso necesario para realizar la unión de las listas de posiciones para cada petición sobre cada una de las tres muestras. Obteniéndose los promedios y las desviaciones típicas de cada uno de los tiempos medidos para los distintos tipos de *truncamiento*.

d) Búsqueda con *máscaras*.

Se seleccionan al azar 100 palabras de entre todas las distintas del Diccionario de Medicina, se construye una muestra de 100 peticiones —que especifiquen al menos el 50% de los caracteres. Para cada palabra se sortea el número de asteriscos entre uno y la mitad de su longitud y se seleccionan al azar las posiciones de éstos. Se miden el tiempo de recuperación de las palabras que verifican la petición y el lapso necesario para realizar la unión de las listas de posiciones para cada petición de la muestra. Se calculan el promedio y la desviación típica de los dos tiempos medidos.

Los resultados experimentales de los apartados **a**, **b**, **c** y **d** se muestran en la tabla 6. La primera y segunda columna corresponden al promedio y la desviación típica, respectivamente, del tiempo —medido en segundos— desde el momento en que el usuario realiza la petición hasta que se muestran en pantalla las palabras que la satisfacen. La tercera y cuarta contienen respectivamente el promedio y la desviación

típica del tiempo en segundos que se consume en la *unión* de las listas de posiciones asociadas a las palabras que forman parte de cada respuesta múltiple, en el caso de que el usuario las seleccionara todas. La quinta columna indica el promedio del número de palabras que verifican la petición, **NPR**.

		Tiempo promedio en el índice	Desviación típica del tiempo del índice	Tiempo promedio de unión de las listas	Desviación típica del tiempo de unión	NPR
Exacta		0.0322	0.04218	0.0000	0.00000	1
Máscaras		0.0663	0.06811	0.1164	0.22555	1.63
Truncamiento	Izquierdo	0.8333	1.51744	0.5875	1.72712	10.68
	Derecho	0.9130	1.68815	0.7162	2.09274	13.3
	Ambos lados	0.9536	1.77283	0.9797	2.34481	73.55
Más similares	DL=2	1.1092	1.05868	0.2738	0.71312	3.14
	DL=4	2.0617	1.74955	0.3182	0.95049	3.14

Tabla 6

- La búsqueda que consume más tiempo en el índice, tabla 6, es la de *más similares*. El tiempo promedio de la *unión* depende del número de palabras que verifican la petición —además de la longitud de las listas de posiciones para cada una de ellas—; las búsquedas para las que se obtiene una mayor multiplicidad son las que tienen *truncamientos*, en particular el *truncamiento a ambos lados*.
- Para todos los tipos reflejados en la tabla 6, se obtienen mayores coeficientes de variación para los tiempos de unión de listas que para los de

recorrido en el índice, debido a la variabilidad en la longitudes de las listas de posiciones.

☞ Se observa, tabla 6, que los valores más altos de las desviaciones típicas tanto en los tiempos de búsqueda como en los de unión de listas se presentan en las peticiones con *truncamientos*; la razón hay que buscarla en la gran variabilidad existente en el número de palabras del Diccionario de Medicina que verifican este tipo de peticiones.

e) Búsquedas con *operadores booleanos*.

A partir de las siete muestras construidas en los apartados **a**, **b**, **c** y **d**, se forman parejas entre los distintos tipos de búsqueda estudiados y se seleccionan 100 pares de peticiones para cada una de las combinaciones posibles, utilizándose éstas para estudiar cada uno de los *operadores booleanos*. Se mide el tiempo transcurrido en segundos desde que se formula la petición *booleana* hasta antes de comenzar a visualizar los *artículos*, se calcula el promedio y la desviación típica de dicho tiempo para cada combinación con cada uno de los *operadores booleanos*.

"Y"		Tiempo promedio de búsqueda						
		Exacta	Máscaras	Truncamiento			Más similares	
				Izquierdo	Derecho	Ambos lados	DL=2	DL=4
Exacta		0.0947	0.2276	1.4626	1.6641	2.0131	1.4710	2.4049
Máscaras		0.2252	0.3439	1.6809	1.8379	2.1437	1.5708	2.5348
Truncamiento	Izquierdo	1.4570	1.5885	2.7490	3.0315	3.3148	2.9930	3.9691
	Derecho	1.7247	1.8466	3.0601	3.1955	3.6493	3.2154	4.1837
	Ambos lados	1.9568	2.1165	3.3131	3.7107	3.8246	3.4667	4.4457
Más similares	DL=2	1.4034	1.5845	3.0368	3.1819	3.4637	2.7220	3.6633
	DL=4	2.3332	2.4893	4.0146	4.1964	4.4630	3.6838	4.5954

Tabla 7

"Y"		Desviación típica del tiempo de búsqueda						
		Exacta	Máscaras	Truncamiento			Más similares	
				Izquierdo	Derecho	Ambos lados	DL=2	DL=4
Exacta		0.0674	0.2377	2.8616	3.4889	4.0614	1.1570	1.8297
Máscaras		0.2386	0.3251	3.0152	3.6551	4.1231	1.1919	1.8083
Truncamiento	Izquierdo	2.8517	2.9582	5.1383	6.1440	6.6046	3.2692	3.4573
	Derecho	3.4922	3.6356	6.1364	6.8282	7.8035	3.7233	3.7015
	Ambos lados	3.8854	4.0301	6.5415	8.0055	7.5716	4.1660	4.1666
Más similares	DL=2	1.1044	1.1714	3.3072	3.8612	4.2116	1.9289	2.6661
	DL=4	1.7911	1.7999	3.5552	3.7501	4.2623	2.6356	3.1100

Tabla 8

"O"		Tiempo promedio de búsqueda						
		Exacta	Máscaras	Truncamiento			Más similares	
				Izquierdo	Derecho	Ambos lados	DL=2	DL=4
Exacta		0.1175	0.3173	1.6144	1.8477	2.1393	1.5500	2.4556
Máscaras		0.2780	0.4244	1.8632	2.0950	2.3525	1.6627	2.6094
Truncamiento	Izquierdo	1.5872	1.7594	2.9348	3.2839	3.5701	3.2397	4.1665
	Derecho	1.8874	2.0173	3.3043	3.5498	3.7843	3.4311	4.4023
	Ambos lados	2.1257	2.2800	3.5156	3.9055	4.1824	3.6900	4.6602
Más similares	DL=2	1.4902	1.6793	3.1524	3.4656	3.7299	2.8801	3.7928
	DL=4	2.4237	2.6105	4.2084	4.4579	4.6694	3.8305	4.7388

Tabla 9

"O"		Desviación típica del tiempo de búsqueda						
		Exacta	Máscaras	Truncamiento			Más similares	
				Izquierdo	Derecho	Ambos lados	DL=2	DL=4
Exacta		0.0779	0.3244	3.1294	3.8754	4.2714	1.2300	1.8743
Máscaras		0.3004	0.4249	3.3140	4.2025	4.5065	1.2457	1.8410
Truncamiento	Izquierdo	3.0871	3.2153	5.5453	6.6077	7.0923	3.5989	3.8256
	Derecho	3.8548	3.9943	6.6105	7.5437	7.8930	4.0832	4.0675
	Ambos lados	4.2015	4.3412	6.9743	8.0797	8.3003	4.5431	4.5758
Más similares	DL=2	1.1956	1.2259	3.5949	4.2944	4.6550	1.9731	2.7134
	DL=4	1.8642	1.8453	3.8910	4.1977	4.6231	2.7585	3.1575

Tabla 10

"Y_NO"		Tiempo promedio de búsqueda						
		Exacta	Máscaras	Truncamiento			Más similares	
				Izquierdo	Derecho	Ambos lados	DL=2	DL=4
Exacta		0.0864	0.2242	1.4642	1.7736	2.0569	1.4599	2.3715
Máscaras		0.2360	0.3504	1.6768	1.8740	2.1496	1.5689	2.5057
Truncamiento	Izquierdo	1.4730	1.6227	2.8232	3.0808	3.4003	2.9727	4.0398
	Derecho	1.7366	1.8554	3.1626	3.3016	3.6597	3.3152	4.2397
	Ambos lados	2.0117	2.1451	3.4047	3.8373	3.8705	3.5320	4.5038
Más similares	DL=2	1.4221	1.5967	3.0450	3.1612	3.4191	2.7574	3.7093
	DL=4	2.3460	2.5256	4.0947	4.2648	4.5022	3.7382	4.6109

Tabla 11

"Y_NO"		Desviación típica del tiempo de búsqueda						
		Exacta	Máscaras	Truncamiento			Más similares	
				Izquierdo	Derecho	Ambos lados	DL=2	DL=4
Exacta		0.0747	0.2343	2.8987	3.6213	4.0876	1.1553	1.7861
Máscaras		0.2702	0.3434	3.0432	3.7177	4.1383	1.1600	1.7835
Truncamiento	Izquierdo	2.9166	3.0493	5.3774	6.3315	6.9510	3.3886	3.6391
	Derecho	3.6150	3.7311	6.5784	7.3318	7.7497	3.9116	3.7971
	Ambos lados	4.0466	4.1056	6.8224	8.0647	7.7743	4.2984	4.2626
Más similares	DL=2	1.1418	1.1745	3.3967	3.7600	4.1721	1.9165	2.7303
	DL=4	1.8005	1.8107	3.6863	3.87099	4.2800	2.7172	3.1210

Tabla 12

☞ De las tres búsquedas *booleanas* la más costosa, en general, es la *disyuntiva*, tabla 9; la *conjuntiva* y la que incluye el *operador y_no* presentan realizaciones parecidas, tablas 7 y 11. Las combinaciones que

presentan un mayor tiempo de respuesta son aquellas en las que intervienen búsquedas de *más similares*.

- ☞ Las tablas 7-12 muestran una tendencia a la simetría teórica respecto a la diagonal principal que en la práctica no se alcanza debido a las fluctuaciones en la medición de los tiempos, aunque cuando se traten listas de igual tamaño y con el *operador y_no* puede existir alguna influencia debida al orden en el que se traten las listas de posiciones.
- ☞ Se puede comprobar el incremento del tiempo promedio de una *combinación* respecto a la suma del consumido por las búsquedas que intervienen en ella utilizando la tabla 6, junto con las tablas 7, 9 y 11.
- ☞ Se mantiene que los mayores coeficientes de variación —tablas 8, 10 y 12 en relación con las 7, 9 y 11— se obtienen, en general, para la resolución de combinaciones en las que intervienen *truncamientos* que son además las que presentan mayor variabilidad.

f) Búsqueda en *sentencias*.

Como muestra se seleccionan al azar parejas de palabras de entre las diferentes del Diccionario de Medicina, sin considerar las que no pertenezcan a un mismo artículo, hasta completar un total de 100.

g) Búsquedas de *cercanía* y *antecedencia*.

Se utilizan las mismas parejas del apartado anterior. Se fija el valor de **n**, siendo 5 para *antecedencia* y 20 para *cercanía*.

h) Búsqueda de *frases*.

Se selecciona aleatoriamente una muestra de 100 *frases* del Diccionario de Medicina formadas por entre cinco y doce palabras, y con no menos de cuatro palabras no *vacías* ni más de seis.

Para los apartados **f**, **g** y **h** se mide el tiempo desde que se formula la petición hasta antes de comenzar a visualizar los artículos —incluyendo la verificación, en cada artículo, de las restricciones especificadas en la petición en cuanto a la situación de las palabras.

Se calcula el promedio y la desviación típica del tiempo medido en segundos para cada uno de los apartados **f**, **g** y **h**, bajo las condiciones expuestas. Los resultados se muestran en la tabla 13.

	Tiempo promedio	Desviación típica del tiempo
Sentencias	0.1949	0.11274
Cercanía	0.2086	0.06851
Antecedencia	0.1895	0.06639
Frases	0.4248	0.16315

Tabla 13

☞ Los cuatro tipos de búsqueda de la tabla 13 requieren el acceso al *artículo* para comprobar si se cumplen las restricciones de la petición, a pesar de ello los resultados obtenidos son bastante satisfactorios. Se observa un tiempo algo mayor para la búsqueda de *frases* debido al mayor número de palabras implicadas. La variabilidad en general es pequeña para este tipo de búsquedas.

Capítulo 6.
Conclusiones y Principales Aportaciones

En este trabajo se propone una solución al problema de la búsqueda de las cadenas *más similares* a una dada —según la distancia de Levenshtein, **DL**— en un determinado diccionario de cadenas.

Como aportación esencial se ha introducido una nueva distancia denominada *distancia invariante trasposicional*, **DIT**, que se define en función de la frecuencia de aparición de cada carácter en ambas cadenas. Se ha analizado su costo computacional obteniéndose que es sensiblemente inferior al de **DL** y además se ha demostrado que $DIT(X,Y) \leq 2*DL(X,Y)$. Estas características permiten su aplicación a la construcción de un filtro adaptivo **DIT | DL** que tiene por misión reducir el número de cadenas del diccionario que han de soportar el cálculo de **DL** con la cadena de búsqueda.

El diccionario se organiza de manera genuina como un árbol en el cual se estructuran las componentes que intervienen en el cálculo de **DIT** —estructura **S-D**— con el fin de optimizar el cálculo **DIT** y a la vez disponer de una forma de uso más eficaz del filtro que seleccione las cadenas con las que se evalúa **DL**. Se propone como criterio de selección del carácter discriminante el de los *mínimos cuadrados*. Se

comprueba la evolución de la relación ocupacional entre el índice y los datos con el incremento de la cardinalidad del diccionario, obteniéndose una tendencia a la igualdad en las respectivas ocupaciones ello indica la bondad de la estructura.

El esquema de búsqueda de las cadenas *más similares*, desarrollado en este trabajo, que comienza con un valor del radio igual a infinito y posteriormente va disminuyendo, con el tratamiento de cadenas cada vez más próximas, ha sido denominado esquema *decreciente*.

Se ha estudiado un nuevo esquema de búsqueda denominado *creciente*, donde el radio de búsqueda, en oposición a la evolución clásica decreciente, sigue una línea de modificación creciente.

Se ha presentado una *familia de esquemas* definida en función del incremento del radio de búsqueda que, de una manera natural, evoluciona conceptualmente desde el *decreciente* al *creciente*.

En el esquema *creciente*, a diferencia del *decreciente*, existen nodos que se visitan más de una vez; en principio esta característica no es deseable, pero queda suficientemente compensada por la aproximación más acertada a las cadenas más próximas, como se ha visto —da lugar a una mejor realización en memoria interna.

La introducción de la poda **PP** ha supuesto, para ambos esquemas, una optimización en el recorrido del índice.

Se ha introducido una nueva distancia, **DS**, definida en función de la longitud de la subsecuencia común más larga; tiene en cuenta características posicionales de las cadenas que el índice no considera y posee un bajo costo computacional; se ha

demostrado además que $DIT(X,Y) \leq 2*DS(X,Y) \leq 2*DL(X,Y)$. Todo ello permite utilizar **DS** como filtro de **DL** con lo que se obtiene una mejor realización.

Se han estudiado diversas formas de paginación de la estructura **S-D** para resolver la situación en la que debido a su tamaño no sea posible ubicarla en memoria interna. Con la paginación en *preorden* se han obtenido los mejores resultados para los dos esquemas de búsqueda. Se selecciona el esquema *decreciente* para la búsqueda de *más similares* en memoria secundaria ya que supera al *creciente* para $DL > 2$ y la mejoría del *creciente* para $DL \leq 2$ se reduce a medida que aumenta el tamaño de la página.

Se ha aplicado la estructura **S-D** a la recuperación de información en texto libre con resultados altamente satisfactorios. La estructura se ha construido con el conjunto de palabras diferentes consideradas no *vacías*, extraídas de un texto de más de un millón de palabras; constituye un índice que permite además de la recuperación de las palabras *más similares* otros tipos de búsqueda habituales en texto libre.

El estudio previo de la paginación de la estructura **S-D** ha permitido implantar esta aplicación en un ordenador personal, suponiendo un incremento del tiempo de respuesta con respecto a la realización en memoria interna que se puede considerar muy aceptable.

La continuidad de este trabajo está en la extensión a la recuperación de información en bases de datos documentales de gran dinamicidad; el objetivo es desarrollar un sistema que basándose en una reestructuración parcial de la estructura permita la recuperación de información en un archivo de documentos que se está

actualizando constantemente, implementándolo en un ambiente multitarea de forma que puedan llevarse a cabo tales reestructuraciones mientras el usuario realiza consultas. Otro aspecto a tratar en futuros trabajos consiste en llevar a cabo una generalidad de los diversos tipos de búsqueda, teniendo en cuenta la organización interna de los diferentes documentos y la adaptación de la estructura a esta nueva situación.

Apéndices

Apéndice 1. Lista de Palabras Vacías

a
acción
ácido
activa
actividad
actúa
actualidad
actualmente
además
afección
afecta
agente
al
algo
alguien
algún
alguna
algunas
algunos
alrededor
ambas
ambos
ángulo
anterior
antes
aparece
aparecen
aparecer
aparición
aplica
aplicación
aplicase
aproximadamente
aquel
aquella
aquellas
aquellos
área
arriba
así
asocia
asociación
asociada
asociado
atrás
aumenta
aumento
aunque
ausencia
años
bajo
base
bastante
bien
borde
cada
cambio
cambios
capacidad
capaz
característica
características
característico
caracteriza
caracterizada
caracterizado
casi
caso
casos
causada
causas
cavidad

célula
células
central
centro
cierta
ciertas
cierto
ciertos
combinación
como
completamente
componente
comprende
compuesto
compuestos
común
con
condiciones
conducto
congénita
conjunto
conoce
consecuencia
considera
consiste
consistente
constituida
constituido
constituye
constituyen
contenido
contiene
contienen
continua
contra
corresponde
cree
crónica
cual
cuales
cualquier
cualquiera
cuando
cuanta
cuantas
cuanto
cuantos
cuatro
cuerpo
cursa
curso
cuya
cuyas
cuyo
cuyos
da
dan
dar
de
debajo
debe
deben
debida
debido
defecto
degeneración
del
delante
demás
denomina
denominada
denominado
dentro
depende
derivado
derivados
desarrolla
desarrollo
descrito
desde
después
determinación
determinada
determinadas
determinado
determinados
determinar
detrás
dícese
diferentes
dirección
disminución
distinguen

distribución
diversas
diversos
dolor
donde
dos
durante
e
efecto
efectos
ej
ejemplo
el
elementos
ella
ellas
ello
ellos
embargo
emplea
empleado
en
encuentra
encuentran
enfermedad
enfermedades
enfermo
enfermos
entre
es
eso
espacio
especial
especialmente
especie
específico
esta
están
estar
estas
este
esto
estos
estructura
estructuras
estudia
estudio
etc
existe
existen
existencia
extensión
externa
externo
extiende
extremo
factor
factores
falta
familiar
fase
fenómeno
fibras
fin
final
finalmente
forma
formación
formada
formado
forman
forman
formando
formas
frecuencia
frecuente
frecuentemente
frecuentes
fue
fuera
función
funciones
fundamental
fundamentalmente
general
generalmente
género
gr
gran
grande
grandes
grupo
ha

haber
habitualmente
hace
hacen
hacer
hacia
halla
hallan
han
hasta
hay
hecho
hueso
i
igual
importancia
importante
importantes
incluso
incluyen
indica
individuos
inferior
inflamación
insuficiencia
intensa
interior
interna
interno
izquierda
junto
la
lado
largo
las
lat.
lateral
laterales
le
les
lesión
lesiones
línea
llama
llamada
llamado
lo
local
localiza
localización
localizada
localizado
los
lugar
mal
manera
manifestaciones
manifiesta
maniobra
mas
material
mayor
mayoría
media
mediante
medida
medio
medios
menor
menos
menudo
método
mía
mías
mientras
mío
míos
misma
mismas
mismo
mismos
mitad
modo
mucha
muchas
mucho
muchos
múltiple
múltiples
músculo
músculos
muy
nada

nadie
ni
nivel
no
nombre
normal
normales
normalmente
nosotros
nuestra
nuestro
nuestros
número
o
observa
observan
observar
observarse
obtenido
ocasionalmente
ocasiones
ocurre
ocurrir
origen
origina
otra
otras
otro
otros
pacientes
para
parece
parte
partes
particularmente
partir
pasan
paso
pequeña
pequeñas
perdida
período
permite
pero
perteneciente
piel
plano
poca
pocas
poco
pocos
por
porción
porque
posee
poseen
posible
posición
posterior
posteriormente
presencia
presenta
presentación
presentan
presentar
presentarse
presente
primaria
primario
primer
primera
primeros
principal
principales
principalmente
probablemente
proceso
procesos
producción
produce
producen
producida
producido
produciendo
producir
producto
productos
progresiva
propia
propiedad
propiedades
propio
provoca

prueba
puede
pueden
pues
punto
que
queda
rápidamente
raramente
reacción
realiza
realizar
recibe
referencia
referirse
refiere
relación
relativamente
relativo
representa
respectivamente
respuesta
resultado
s
se
sea
secundaria
secundario
secundarios
seguida
según
segundo
semejante
sensación
ser
si
sido
siempre
siendo
significa
signo
sigue
simple
sin
síndrome
sino
síntoma
sirve
sistema
situación
situada
situadas
situado
sobre
solo
solución
son
su
suele
suelen
superficial
superficie
superficies
superior
superiores
sus
sustancia
sustancias
suya
suyas
suyo
suyos
tal
tales
tamaño
también
tan
tanta
tantas
tanto
tantos
tejido
tejidos
tendencia
tener
tercer
término
tiempo
tiene
tienen
tipo
tipos
toda

todas
todo
todos
total
totalmente
tras
trata
tratado
tratamiento
través
tres
tu
tuya
tuyas
tuyo
tuyos
u
un
una
unas
únicamente
unión
uno
unos
usualmente
utiliza
utilizada
utilizado
v
va
van
varias
variedad
varios
ve
veces
vez
vía
vosotros
vuestra
vuestro
vuestros
y
ya
yo
zona
zonas

Apéndice 2. Abreviaturas y Siglas

abs	Función valor absoluto
B	Cadena de búsqueda
C_DIT	Componentes de DIT
CA_DIT	Valor acumulado de las componentes de DIT en el recorrido del índice
D	Diccionario: conjunto de cadenas sobre un alfabeto dado
DIT	Distancia invariante trasposicional
DL	Distancia de Levenshtein
DLM	Mínima DL alcanzada
DS	Distancia de Santana
DTM	Radio de búsqueda DIT
IR	Incremento del radio de búsqueda en la familia de esquemas
Lsc	Función que calcula la longitud de la subsecuencia común más larga
LV	Matriz del cálculo optimizado de Landau y Vishkin de DL
MFU	Política para liberar del buffer la página menos frecuentemente usada
MRU	Política para liberar del buffer la página menos recientemente usada
NCD	Cardinalidad del diccionario
NNA	Número de nodos afectados
NNR	Número de veces que se revisitan los nodos
NPR	Multiplicidad de la respuesta: número de palabras que verifican la petición
PA	Poda de la estructura S-D que depende exclusivamente de los ancestros
PC	Ordenador personal

PEPS	Política para liberar del buffer la primera página que entró
PP	Poda predictiva sobre los sucesores en la estructura S-D
S-D	Estructura de Santana y Díaz que organiza las componentes de DIT
SB	Conjunto de cadenas más similares a B
SIT	Sinónimos invariantes trasposicionales
UEPS	Política para liberar del buffer la última página que entró
WF	Matriz del cálculo de Wagner y Fischer de DL

Referencias

[AL67] ALBERGA, C. N.: "String Similarity and Misspellings". Comm. ACM., Vol. 10 (5), 302/313, (1967).

[BE75] BENTLEY, J. L.: "Multidimensional Binary Search Trees used for Associative Searching". Comm. of ACM., Vol. 18, 9, 509/516, (1975).

[BE79] BENTLEY, J. L.: "Multidimensional Binary Search Trees in Database Applications". IEEE Transactions on Software Engineering, Vol. SE-5, 4, 333/340, (1979).

[BE85] BECKLEY, D. A.; EVENS, M. W.; RAMAN, V. K.: "Multikey Retrieval from K-D Trees and Quad-Trees". AT & T Bell Laboratories Naperville, Ill. Go 566, 1/29, (1985).

[BK73] BURKHARD, W. A.; KELLER, R. M.: "Some Approches to Best-Mach File Searching". Comm. ACM., 16, 4, (1973).

[BM77] BOYER, R. S.; MOORE, J. S.: "A Fast String Searching Algorithm". Comm. ACM., Vol. 20, 762/772, (1977).

[BY89] BAEZA-YATES, R.: "Efficient Text Searching". PhD thesis Dept. of Computer Science, Univ. of Waterloo, 1989. Also as Research Report Cs-89-17, (1989).

[CO91] COLE, R.: "Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm". 2nd Symp. on Discrete Algorithms, SIAM, S. Francisco, Cal., 224/233, (1991).

[DS90] DIAZ, M.; SANTANA, O.; RODRIGUEZ, J. C.: "Búsqueda de las Cadenas Más Similares: Esquema Decreciente con Radio de Búsqueda Ascendente, Esquema Creciente". XVI Conferencia Latinoamericana de Informática, Vol I, 90/97, (1990).

[DS93] DIAZ, M; SANTANA, O.; PEREZ, J.: "Distancia dependiente de la Subsecuencia Común Más Larga entre Cadenas de Caracteres". Aceptado para publicar en las Actas de las II Jornadas de Ingeniería de Sistemas Informáticos y Computación, Quito (Ecuador), (1993).

[FA85] FALOUTSOS, C.: "Access Methods for Text". ACM Comm. Surveys, 17, 49/74, (1985).

[FA88] FALOUTSOS, C.: "Signature Files: An Integrated Access Method for Text and Attributes Suitable for Optical Disk Storage". BIT, 28 (4), 736/754, (1988).

[FB74] FINKEL, R. A.; BENTLEY, J. L.: "Quad Trees a Data Structure for Retrieval on Composite Keys". Acta Informatica, 4, 1/9, Springer Verlag, (1974).

[FB77] FRIEDMAN, J. H.; BENTLEY, J. L.; FINKEL, R. A.: "An Algorithm for Finding Best Matches in Logarithmic Expected Time". ACM Transactions on Mathematical Software, Vol. 3, 3, 209/226, (1977).

[FB92] FRAKES, W. B. & BAEZA-YATES, R. editors: "Information Retrieval. Data Structures & Algorithms". Prentice Hall, Englewood Ciffs, New Jersey, (1992).

[FC84] FALOUTSOS, C.; CHRISTODOULAKIS, S.: "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation". ACM TOOIS, 2 (4), 267/288, (1984).

[FO73] FORNEY, G. D.: "The Viterbi Algorithm". Proc. IEEE, Vol. 61, 268/278, (1973).

[GG86] GALIL, Z.; GIANCARLO, R.: "Improved String Matching with k Mismatches", SIGACT News, 17, 4, 52/54, (1986).

[GO87] GONNET, G.: "PAT 1.3: An Efficient Text Searching System, User's Manual", UW Centre for the New OED, University of Waterloo, (1987).

[HA71] HARRISON, M. C.: "Implementation of the Substring Test by Hashing". C. ACM, 14, 777/779, (1971).

[HD80] HALL, P. A. V.; DOWLING, G. R.: "Approximate String Matching". Computing Surveys., Vol. 12, 4, 381/402, (1980).

[HI75] HIRSCHBERG, D. S.: "A Linear Space Algorithm for Computing Maximal Common Subsequences", Communications of the ACM, Vol 18, 341/353, (1975).

[HO80] HORSPOOL, R. N.: "Practical Fast Searching in Strings". Software-Practice and Experience, 10, 501/506, (1980).

[HS77] HUNT, J. W.; SZYMANSKI, T. G.: "A Fast Algorithm for Computing Longest Common Subsequences", Communications of the ACM, Vol 20, 350/353, (1977).

[HS82] HULL, J. J.; SRIHARI, S. N.: "Experiments in Text Recognition with Binary N-Gram and Viterbi Algorithms". IEEE Trans. Pat. Anal. Mach. Intel., Vol. PAMI-4, 5, 520/530, (1982).

[HS91] HUME, A.; SUNDAY, D. M.: "Fast String Searching". AT&T Bell Labs Computing Science Technical Report Nº 156, (1991).

[IK82] ITO, T.; KIZAWA, M.: "Spelling Correction on a Hierarchically Organized File". Trans. Inst. Electron. Commun. Eng. Japan (in Japanese), Vol. J65-D, 1090/1091, (1982).

[IY84] ITAHASHI, S.; YOKOYAMA, S.: "Vocabulary Reduction Effectby Specifying Phoneme Sequences in Words". Trans. Inst. Electron. Commun. Eng. Japan (in Japanese), Vol. J67-D, 869/876, (1984).

[JV87] JOVEN, J.; VILLABONA, C.; JULIA, G.; GONZALEZ-HUIX, F.: "Diccionario de Medicina". Tercera edición, Editorial MARIN, S.A, (1987).

[KA84] KURITA, T.; AIZAWA, T.: "A Method for Correcting Errors on Japanese Words Input and its Application to Spoken Word Recognition with Large Vocabulary". J. Inform. Processing Soc. Jap., Vol. 25, 831/841, (1984).

[KC89] KUO, S.; CROSS, G. R.: "An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings", SIGIR FORUM, Vol 23, Numbers 3, 4, (1989).

[KD83] KANEKO, T.; DIXON, N. R.: "A Hierarchical Decision Approach to Large Vocabulary Discrete Utterance Recognition". IEEE Trans. Acoust., Speech, Signal Processing, Vol. ASSP-31, 1061/1066, (1983).

[KM77] KNUTH, D. E.; MORRIS, J.; PRATT, V.: "Fast Pattern Matching in Strings". SIAM J. on Computing, 6, 323/350, (1977).

[KO85] KASHYAP, R. L.; OOMMEN, B. J.: "Spelling Correction Using Probabilistic Methods". Pattern Recognition Letters, No. 2, 147/154, (1985).

[LA83] LARSON, P.: "A Method for Speeding Up Text Retrieval". Proceedings ACM SIGMOD, San Jose, California, 12, 117/123, (1983).

[LE66] LEVENSHTEIN, V. I.: "Binary Codes Capable of Correcting, Insertions and Reversals". Soviet Phys. Dokl. 10, 707/710, (1966).

[LV85a] LANDAU, G. M.; VISHKIN, U.: "Efficient String Matching in the Presence of Errors". Proc. 26th IEEE FOSC, 126/136, (1985).

[LV85b] LANDAU, G. M.; VISHKIN, U.: "Efficient String Matching with k Differences". TR-36/85, Departament of Computer Science, Tel Aviv Univ., Submitted for Journal Publication, (1985).

[LV86a] LANDAU, G. M.; VISHKIN, U.: "Efficient String Matching with k Mismatches". Theoretical Computer Science, 43, 239/249, (1986).

[LV86b] LANDAU, G. M.; VISHKIN, U.; NUSSINOV, R.: "An Efficient String Matching Algorithm with k Differences for Nucleotide and Amino Acid Sequences". Nucleic Acid Research 14 (1), 31/46, (1986).

[MM90] MANBER, U.; MYERS, G.: "Suffix Arrays: A New Method for On-line String Searches". 1st ACM-SIAM Symposium on Discrete Algorithms, San Francisco, 319/327, (1990).

[MO70] MORGAN, H. L.: "Spelling Correction in System Programs". C. ACM, Vol. 13, 2, 90/94, (1970).

[NE75] NEUHOFF, D. L.: "The Viterbi Algorithm as an Aid in Text Recognition". IEEE Trans. Inform. Theory, Vol. IT-21, 222/226, (1975).

[OT76] OKUDA, T.; TANAKA, E.; KASAI, T.: "A Method for the Correction of Garbled Words Based on the Levenshtein Metric". IEEE Trans. Comput., Vol. C-25, 2, 172/177, (1976).

[RE71] RISEMAN, E. M.; EHRICH, R. W.: "Contextual Word Recognition Using Binary Digrams". IEEE Trans. Comput., Vol. C-20, 397/403, (1971).

[RH74] RISEMAN, E. M.; HANSON, A. R.: "A Contextual Post-Processing System for Error-Correcting Using Binary N-Grams". IEEE Trans. Comput., Vol. C-23, 5, 480/493, (1974).

[RI77] RIVEST, R. L.: "On the Worst-Case Behavior of String-Searching Algorithms". SIAM J. Comput., Vol. 6, 4, 669/674, (1977).

[SD87] SANTANA, O.; DIAZ, M.; MAYOR, O.; REYES, J.: "Esquemas y Estructura para la Búsqueda de las Palabras Más Similares a una Dada". XIII Conferencia Latinoamericana de Informática, Vol. II, 1169/1189, (1987).

[SD89] SANTANA, O.; DIAZ, M.; DUQUE, J. D.; RODRIGUEZ, G.: "Estructuración de las Componentes de la Distancia Invariante Trasposicional, DIT, con Compartición de la Zona No-Discriminante en la Búsqueda de las Cadenas Más Similares". XV Conferencia Latinoamericana de Informática, Vol. II, 335/341, (1989).

[SD92] SANTANA, O.; DIAZ, M.; BALLESTER, A.; SANTANA, F. J.: "Recuperación de Información en Diccionarios". VIII Congreso de la Sociedad Española para el Procesamiento del Lenguaje Natural (SEPLN), (1992).

[SH83] SHINGHAL, R.: "A Hybrid Algorithm for Contextual Text Recognition". Pattern Recognition, Vol. 16, 261/267, (1983).

[SP90] SANTANA. O.; PEREZ, J.; RODRIGUEZ, J. C.: "Increasing Radius Search Schemes for the Most Similar Strings on the Burkhard-Keller Tree". Cybernetics and Systems: An International Journal, (1990).

[SR90] SANTANA, O.; RODRIGUEZ, J. C.; DIAZ, M.: "Búsqueda de las Cadenas Más Similares: Incidencia de la Subsecuencia Común Más Larga en los Esquemas Decreciente y Creciente". XVI Conferencia Latinoamericana de Informática, Vol I, 98/104, (1990).

[SS85] SMALL, H.; SWEENEY, E.: "Clustering the Science Citation Index Using Cocitations. I. A Comparison of Methods". Scientometrics, 7, 391/409, (1985).

[ST79] SHINGHAL, R.; TOUSSAINT, G. T.: "A Bottom-Up and Top-Down Approach to Using Context in Text Recognition". Int. J. of Man-Mach. Studies, Vol. 11, 201/212, (1979).

[SU90] SUNDAY, D.: "A Very Fast Substring Search Algorithm". Communications of the ACM, 33 (8), 132/142, (1990).

[SZ73a] SZANSER, A. J.: "Automatic Error Correction of Natural Text: Part 1". Computer Science, No. 46. National Physics Laboratory, England, (1973).

[SZ73b] SZANSER, A. J.: "Bracketing Technique in Elastic Matching". Comput. J., Vol. 16, 2, 132/134, (1973).

[TK86] TANAKA, E.; KOHASHIGUCHI, T.; SHIMAMURA, K.: "High Speed Error Correcting Method for Substitution Errors". Trans. Inform. Processing Soc. Japan (in Japanese), Vol. 27, 177/182, (1986).

[TK87] TANAKA, E.; KOJIMA, Y.: "A high speed string correction method using a hierarchical file". IEEE Trans. Pattern Anal. Mach. Intell. Vol. PAMI-9, 6, 806/815, (1987).

[TT86] TANAKA, E.; TOYAMA, T.; KAWAI, S.: "High Speed Error Correction of Phoneme Sequences". Pattern Recognition, Vol. 19, 407/412, (1986).

[TU90] TARHIO, J.; UKKONEN, E.: "Boyer-Moore Approach to Approximate String Matching". Proceedings Scandinavian Workshop in Algorithmic Theory, SWAT'90, Lecture Notes in Computer Science, 447, Springer-Verlag, Bergen, Norway, 2, 348/359, (1990).

[UK83] UKKONEN, E.: "On Approximate String Matching". Proc. Int. Conf. Found. Comp. Theor., Lecture Notes in Computer Science 158, Springer-Verlag, 487/495, (1983).

[UK85] UKKONEN, E.: "Finding Approximate Pattern in Strings". J. of Algorithms, 6, 132/137, (1985).

[UL75] ULLMANN, J. R.: "A Binary N-Gram Technique for Automatic Correction of Substitution, Deletion, Insertion and Reversal Errors in Words". The Computer Journal, Vol. 20, 2, 141/147, (1975).

[UW90] UKKONEN, E.; WOOD, D.: "A Simple on-line Algorithm to Approximate String Matching". (Report A-1990-4), Helsinki, Finland, (1990).

[WF74] WAGNER, R. A.; FISCHER, M. J.: "The String-to-String Correction Problem". JACM, 21 (1), 168/173, (1974).

[WI88] WILLETT, P.: "Recent Trends in Hierarchic Document Clustering: A Critical Review". Information Processing & Management, 24 (5), 577/597, (1988).