Utilización de un índice métrico para búsqueda aproximada de patrones *

Verónica Ludueña
Departamento de Informática, Universidad Nacional de San Luis.
Ejército de los Andes 950, San Luis, Argentina.
Tel.: 02652-420822-257 – Fax: 02652-430224.
vlud@unsl.edu.ar

y

Gonzalo Navarro
Departamento de Ciencias de la Computación, Universidad de Chile.
Blanco Encalada 2120, Santiago, Chile.
Tel.: +56-2-6892736 – Fax: +56-2-6895531
gnavarro@dcc.uchile.cl

Resumen

Uno de los problemas abiertos en la búsqueda de patrones combinatoria es la indexación de texto para permitir búsqueda aproximada sobre él. Presentamos aquí una implementación de un método nuevo y simple de indexación para el problema de búsqueda aproximada de patrones. El esquema aprovecha las propiedades métricas que posee la distancia de edición y puede ser aplicado a cualquier otra métrica existente entre strings. Consideramos un espacio métrico donde los elementos son los sufijos del texto, construimos un índice métrico, y las búsquedas aproximadas se ven como consultas por proximidad sobre ese espacio métrico.

Palabras claves: búsqueda en texto, algoritmos, espacios métricos.

---

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

1. Introducción y motivación

Uno de los problemas abiertos en búsqueda de patrones combinatoria es la indexación de texto para permitir búsqueda aproximada sobre dicho texto. El problema de búsqueda aproximada de patrones es: Dado un texto largo T de longitud n, un patrón (string) P de longitud m (comparativamente corto) y un umbral r, recuperar todas las ocurrencias del patrón, es decir, los substrings del texto cuya *distancia de edición* al patrón es a lo más r. La *distancia de edición* entre dos strings se define como la mínima cantidad de inserciones, supresiones o substituciones de caracteres necesaria para que los strings sean iguales. Esta distancia es usada en muchas aplicaciones, pero cualquier otra distancia que sea de interés puede ser aplicada.

En la versión on-line de este problema, el patrón puede ser preprocesado pero el texto no. Existen numerosas soluciones a este problema [18], pero ninguna de ellas es aceptable cuando el texto es demasiado largo porque el tiempo de búsqueda es proporcional a la longitud del texto. Sólo recientemente se le ha prestado más atención al problema de indexación de texto para búsqueda aproximada de strings y por ello los esquemas de indexación para este problema están aún bastante inmaduros.

Existen algunos esquemas de indexación especializados para buscar de a palabras sobre textos de lenguaje natural [16, 4]. Estos índices se desempeñan bastante bien en ese caso pero no se pueden extender para trabajar en el caso general. Existen aplicaciones extremadamente importantes que caen fuera de este caso tales como ADN, proteínas, música o lenguajes orientales.

Los índices que resuelven el problema general se pueden dividir en tres clases.*Backtracking* [13, 21, 9, 12] que usa el árbol de sufijo [1], arreglo de sufijo [15] o grafos dirigidos acíclicos de palabras (DAWG) [10] del texto con el fin de factorizar sus repeticiones. Se latsimula un algoritmo secuencial sobre el texto haciendo backtracking sobre la estructura. Estos algoritmos son atractivos cuando se buscan patrones muy cortos.

Particionado [20, 19, 3] particiona el patrón en piezas para asegurar que alguna de las piezas deba aparecer sin alteraciones dentro de cada ocurrencia. Se usa un índice capaz de realizar búsquedas exactas para detectar las piezas y se verifican con un algoritmo secuencial las áreas del texto que tengan suficiente evidencia de contener una ocurrencia. Estos algoritmos funcionan bien sólo cuando r/m es pequeño.

La tercera clase [17, 5] es un híbrido entre las otras dos. Se divide el patrón en piezas largas que pueden aún contener (menos) errores, se las busca usando backtracking y se verifican las potenciales ocurrencias de texto como en el método de particionado. Los algoritmos híbridos son más efectivos porque pueden encontrar el punto adecuado entre longitud de las piezas a buscar para un nivel de error permitido. Estos métodos toleran factores de error r/m moderados.

En [8] se propuso una nueva manera de aproximarse a este problema, la cual considera que la distancia de edición satisface la desigualdad triangular y por lo tanto define un espacio métrico sobre el conjunto de los substrings del texto. Así se reexpresa el problema de la búsqueda aproximada como un problema de búsqueda por rango sobre este espacio métrico. Este enfoque se ha intentado antes en [6, 2], pero en esos casos las particularidades del problema hacían posible indexar $O(n)$ elementos. En el caso general se tienen $O(n^2)$ substrings del texto. Su principal contribución es que brinda un método (basado sobre el árbol de sufijos del texto) que colapsa los $O(n^2)$ substrings del texto en $O(n)$ conjuntos y encuentra una manera de construir un espacio métrico sobre esos conjuntos. El resultado es un método de indexación que, a costa de requerir en espacio promedio $O(n \log n)$ y tiempo de construcción promedio $O(n \log^2 n)$, permite encontrar las R ocurrencias aproximadas del patrón en tiempo promedio $O(m \log^2 n + m^2 + R)$, lo cual produce una brecha de complejidad sobre trabajos previos, y se puede extender más fácilmente la idea a otras funciones de distancia (tales como inversiones).

Más aún, este método representa un enfoque original al problema y abre muchas posibilidades para mejoras. Presentamos aquí una versión más simple del índice propuesto en [8] que necesita espacio $O(n)$ y que, a pesar de no producir una brecha en la complejidad, puede ser mejor en la práctica.

Utilizaremos la siguiente notación en este artículo. Dado un string $s \in \Sigma^*$ denotamos su longitud con $|s|$. Denotamos también con s_i al i-ésimo símbolo de s, para un entero $i \in 1, \ldots, |s|$ y con $s_{i\ldots j} = s_i s_{i+1} \ldots s_j$ (que sería el string vacío si $i > j$) y $s_{i\ldots} = s_{i\ldots|s|}$. El string vacío se denota como ϵ. Se dice que un string x es un *prefijo* de xy, un *sufijo* de yx y un substring de yxz.

Este artículo está organizado de la siguiente manera: la Sección 2 describe algunos conceptos relacionados a la búsqueda en espacio métricos, la Sección 3 describe en más detalle el enfoque planteado en [8], la Sección 4 explica en mayor detalle el método utilizado en esta implementación, la Sección 5 muestra algunos resultados experimentales y la Sección 6 da las conclusiones y las propuestas para trabajo futuro.

2. Espacios Métricos

Nos concentramos en esta sección en dar sólo los conceptos relacionados a la búsqueda en espacios métricos que son relevantes para este trabajo, más detalles se pueden ver en [7].

Informalmente, un espacio métrico es un conjunto de objetos y una función de distancia definida entre ellos, la cual satisface la desigualdad triangular. El problema de *búsqueda por proximidad* en espacios métricos consiste de indexar el conjunto de manera tal que luego dada una consulta se puedan encontrar rápidamente todos los elementos del conjunto que estén suficientemente cerca del elemento consultado. Esto tiene aplicaciones en un gran número de campos, tales como bases de datos no tradicionales (donde no se usa la búsqueda exacta); aprendizaje de máquina y clasificación; cuantización y compresión de imágenes; recuperación de texto; biología computacional; predicción de funciones; etc.

Formalmente, un *espacio métrico* es un par $(\mathbb{X}, d)$, donde $\mathbb{X}$ es un "universo" de objetos y $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ es una función de distancia definida sobre $\mathbb{X}$ que devuelve valores no negativos. La distancia satisface las propiedades de reflexividad ($d(x, x) = 0$), positividad estricta ($x \neq y \Rightarrow d(x, y) > 0$), simetría ($d(x, y) = d(y, x)$) y desigualdad triangular ($d(x, y) \leq d(x, z) + d(z, y)$).

Un subconjunto finito $\mathbb{U}$ de $\mathbb{X}$, de tamaño $n = |\mathbb{U}|$, es el conjunto de objetos donde se busca. Entre todas las clases de consultas de interés en espacios métricos, estamos interesados en las llamadas *consultas por rango*: Dada una consulta $q \in \mathbb{X}$ y un radio de tolerancia r, encontrar el conjunto de todos los elementos de $\mathbb{U}$ que están a lo sumo a distancia r de q. Formalmente, la salida de la consulta $(q, r)_d = \{x \in \mathbb{U}\ , d(q, x) \leq r\}$. El objetivo es preprocesar el conjunto a fin de minimizar el costo computacional de producir la respuesta $(q, r)_d$.

Sobre la amplia gama de algoritmos existentes para indexar espacios métricos, nos concentramos en los llamados *basados en pivotes*, los cuales se construyen sobre una sola idea general: Seleccionar k elementos $\{p_1, p_2, \ldots, p_k\}$ de $\mathbb{U}$ (llamados *pivotes*), e identificar cada elemento $u \in \mathbb{U}$ con un punto k–dimensional $((d(u, p_1), \ldots, d(u, p_k))$. Con esta información, podemos filtrar, usando la desigualdad triangular, cualquier elemento u tal que $|d(q, p_i) - d(u, p_i)| > r$ para algún pivote p_i, dado que en ese caso sabemos que $d(q, u) > r$ sin necesidad de evaluar realmente $d(q, u)$. Aquellos elementos que no se puedan filtrar usando esta regla se comparan directamente contra q.

Una característica interesante de los algoritmos basados en pivotes es que reducen el número final de evaluaciones de distancia aumentando el número de pivotes. Se define $D_k(x, y) = \text{máx}_{1 \leq j \leq k} |d(x, p_j) - d(y, p_j)|$. Hacer uso de los pivotes equivale a descartar los elementos u tales que $D_k(q, u) > r$. A medida que se agreguen más pivotes se realizan más evaluaciones

de distancia (exactamente k) para computar $D_k(q,*)$ (a éstas se las llama evaluaciones *internas*), pero por otra parte $D_k(q,*)$ incrementa su valor y así tiene una mayor probabilidad de filtrar o descartar más elementos (a aquellas comparaciones contra elementos que no pueden ser descartados por D_k se las llama *externas*). Por lo tanto existe un número de pivotes óptimo.

Si estamos interesados no sólo en el número de evaluaciones de distancia realizadas sino también en el tiempo total de CPU requerido, entonces recorrer todos los n elementos para descartar algunos de ellos podría ser inaceptable. En ese caso, se necesitan métodos de *búsqueda por rango multidimensional*, los cuales incluyen estructuras tales como kd–tree, R–tree, X–tree, etc. [22, 11]. Esas estructuras permiten indexar un conjunto de objetos en un espacio k–dimensional con el fin de procesar consultas por rango.

En este trabajo nos interesamos en un espacio métrico donde el universo es el conjunto de strings sobre algún alfabeto, es decir $\mathbb{X} = \Sigma^*$, y la función de distancia es la llamada *distancia de edición* o *distancia de Levenshtein*. Ésta se define como el mínimo número de cantidad de inserciones, supresiones o substituciones de caracteres necesaria para hacer que los strings sean iguales [14, 18]. La distancia de edición, y de hecho cualquier otra distancia definida como la mejor manera de convertir un elemento en otro, es reflexiva, estrictamente positiva (en la medida que no existan operaciones de costo cero), simétrica (siempre y cuando las operaciones permitidas sean simétricas) y satisface la desigualdad triangular.

El algoritmo para computar la *distancia de edición* $ed()$ se basa en programación dinámica. Supongamos que se necesita computar $ed(x,y)$. Se debe llenar una matriz $C_{0..|x|,0..|y|}$, donde $C_{i,j} = ed(x_{1..i}, y_{1...j})$, entonces $C_{|x|,|y|} = ed(x,y)$. Esto se computa de la siguiente manera:

$$C_{i,0} = i\ ,\ \ C_{0,j} = j,$$
$$C_{i,j} = \textbf{if}(x_i = y_j)\ \textbf{then}\ C_{i-1,j-1}\ \textbf{else}\ 1 + \text{mín}(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$$

El algoritmo toma $O(|x|,|y|)$ tiempo. La matriz puede ser llenada por fila o por columna. El espacio requerido será solamente $O(|y|)$, dado que solamente se deben almacenar las filas previas para computar una nueva, además se debe mantener una fila y almacenarla.

3. Índice Métrico para Búsqueda Aproximada de Patrones

Una aproximación directa de la indexación de un texto para búsqueda aproximada usando técnicas de espacios métricos tiene como principal problema la existencia de $O(n^2)$ substrings diferentes en un texto y por lo tanto se deben indexar $O(n^2)$ objetos, lo cual es inaceptable.

El árbol de sufijos provee una representación concisa de todos los substrings del texto en $O(n)$ espacio. Así, en lugar de indexar todos los substrings del texto, solamente se indexan los nodos explícitos del árbol de sufijos. Por lo tanto hay $O(n)$ objetos ha ser indexados como un espacio métrico bajo la distancia de edición.

Ahora, cada nodo interno explícito del árbol de sufijos se representa a sí mismo y a los nodos que descienden de él por un paso unario. Así, cada nodo explícito que corresponde a un string xy y su padre corresponde al string x representa el siguiente conjunto de strings

$$x[y] = \{xy_1, xy_1y_2, \ldots, xy\}$$

Por ejemplo si el string "$a[bra]$" es un nodo interno de árbol, éste representa

$$\text{“}a[bra]\text{”} = \{\text{“}ab\text{”}, \text{“}abr\text{”}, \text{“}abra\text{”}\}$$

Las hojas del árbol de sufijos representan un único substring del texto y todas sus extensiones hasta llegar al sufijo de texto completo.

Entonces en lugar de indexar $O(n^2)$ substrings del texto, se indexarán $O(n)$ conjuntos de strings, que son los conjuntos representados por los nodos explícitos internos y externos del árbol de sufijos.

Al momento de decidir cómo indexar este espacio métrico formado por $O(n)$ conjuntos de strings surgen varias opciones posibles, pero la propuesta en este artículo es la basada en *pivotes*. Se seleccionan aleatoriamente k pivotes de longitudes $0, 1, 2, \ldots, k-1$. Para cada nodo explícito del árbol de sufijos $x[y]$ y cada pivote p_i se calcula la distancia entre p_i y todos los strings representados por $x[y]$. Del conjunto de distancias desde un nodo $x[y]$ a p_i se almacenarán la mínima y la máxima. Como todos esos strings son de la forma $\{xy_1, \ldots, y_j, 1 \leq j \leq |y|\}$ las distancias de edición pueden ser computadas en $O(|p_i||xy|)$ tiempo.

Dado que en los nodos externos del árbol de sufijos el y tiende a ser bastante largo ($O(n)$ en promedio) se usa una gran cantidad de tiempo computacional calculando las distancias de edición y se obtienen distancias máximas muy altas. La solución dada por los autores a esta situación fue la pesimista: cuando el nodo del árbol sea externo se asumirá que su distancia máxima es n. La distancia mínima puede encontrarse en $O(|p_i| \text{máx}(|p_i|, |x|))$ porque no es necesario considerar strings arbitrariamente largos $xy_1, \ldots, y_j$.

Una vez que se ha hecho esto, para todos los nodos del árbol de sufijos y todos los pivotes, se tiene un conjunto de k valores máximos y mínimos para cada nodo explícito del árbol de sufijos. Esto puede ser visto como un hiper-rectángulo en k dimensiones:

$$\begin{aligned} x[y] \quad \rightarrow \langle \quad & (\text{mín}(ed(x[y], p_0)), \ldots, \text{mín}(ed(x[y], p_{k-1}))), \\ & (\text{máx}(ed(x[y], p_0)), \ldots, \text{máx}(ed(x[y], p_{k-1}))) \ \rangle \end{aligned}$$

y es seguro que todos los strings en $x[y]$ caen dentro del rectángulo.

Sea P el patrón a buscar con, a lo sumo, r errores. Esto es un query por rango con radio r en el espacio métrico de los nodos del árbol de sufijos. Como en los algoritmos basados en pivotes, se compara el patrón P contra los k pivotes y se obtiene una coordenada k-dimensional $(ed(P, p_1), \ldots, ed(P, p_k))$.

Sea p_i un pivote dado y $x[y]$ un nodo dado. Si se cumple que:

$$ed(P, p_i) + r < \text{mín}(ed(x[y], p_i)) \ \vee \ ed(P, p_i) - r > \text{máx}(ed(x[y], p_i)$$

entonces, por desigualdad triangular, se sabe que $ed(P, xy') > r$ para cualquier $xy' \in x[y]$. La eliminación de un string, como posible respuesta, puede ser hecha usando cualquier pivote p_i. De hecho los nodos que no pueden ser eliminados son aquellos cuyo hiper-rectángulo tiene una intersección no vacía con el hiper-rectángulo $\langle (ed(P, p_1) - r, \ldots, (ed(P, p_k) - r), (ed(P, p_1) + r, \ldots, (ed(P, p_k) + r) \rangle$.

En la Figura 1 puede verse el nodo que contiene un conjunto de puntos y se almacena su mínima y su máxima distancia a dos pivotes. Esto determina un rectángulo bidimensional donde caen todas las distancias desde cualquier substring del nodo a los pivotes. El query es un patrón P y una tolerancia r, la cual define un círculo alrededor de P.

Después, teniendo la distancia de P a los pivotes, se puede crear un hiper-cubo (en este caso un cuadrado) de lado $2r + 1$. Si el cuadrado no interseca con el rectángulo ningún substring del nodo está lo suficientemente cerca de P. El problema de encontrar todos los rectángulos k-dimensionales que intersecan con el rectángulo de un query dado es el clásico problema de *búsqueda por rango multimensional*. Aquellos nodos $x[y]$ que no puedan ser eliminados por ningún pivote deberán ser comparados directamente contra el patrón P. Para aquellos cuya distancia mínima a P es a lo más r se reportarán todas sus ocurrencias. Los puntos de comienzos de las mismas se encuentran en las hojas de los subárboles con raíz en el nodo que coincidió.

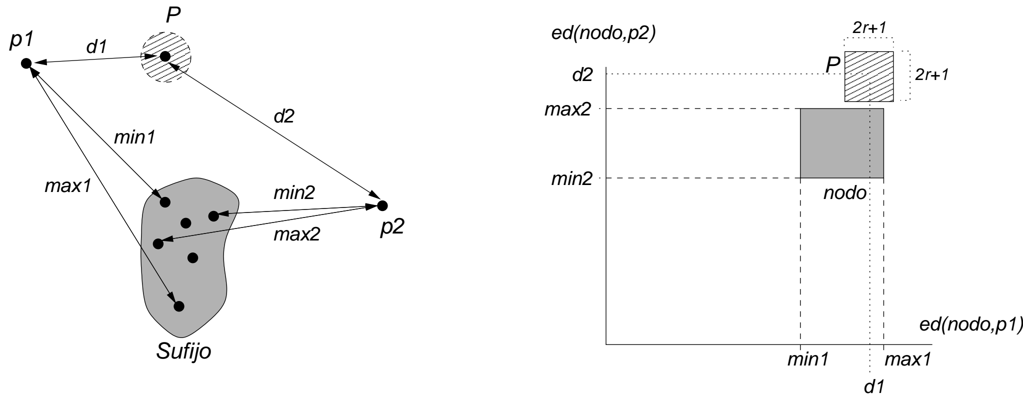

Figura 1: Regla de eliminación usando dos pivotes

4. Nuestra implementación

El presente trabajo describe una implementación de una propuesta alternativa presentada en [8] que es más simple que la descripta anteriormente pero que, aunque no reduce drásticamente la complejidad, puede ser mejor en la práctica.

Un índice más simple que deriva de las mismas ideas del citado artículo, considera solamente los n sufijos del texto y no nodos internos. Cada sufijo $[T_{j...}]$ representa todos los substrings del texto que comienzan en la posición j, esos sufijos son de la forma $\{T_{j...1}, \ldots, T_{j...n}\}, con\ 1 \leq j \leq n$, y cada sufijo es indexado de acuerdo a la mínima distancia entre esos substrings y cada pivote p_i.

Una de las ventajas de esta propuesta es la reducción de espacio. No solamente el conjunto $\mathbb{U}$ puede ser reducido hasta la mitad de los elementos presentes en la aproximación original, sino que también solamente k valores (distancias mínimas), y no $2k$, son almacenados para cada elemento. Esto permite utilizar cuatro veces más pivotes que en la propuesta previa con el mismo requerimiento de memoria. Notar que no necesitamos construir ni guardar en ninguna estructura auxiliar los sufijos, ya que ellos se pueden obtener e indexar directamente desde el texto. Así, la única estructura necesaria sería el *índice métrico*.

Una desventaja es que la selectividad de los pivotes se reduce debido a que solamente los valores mínimos son almacenados y esto impacta en el descarte de sufijos.

La construcción del *índice métrico* comienza con la selección aleatoria de k substrings de texto diferentes que serán nuestros pivotes. Estos substrings son obtenidos del mismo texto sobre el cual se buscará el patrón.

Para cada sufijo $[T_{j...}]$ y cada pivote p_i se calcula la distancia mínima. Para ello se computará la distancia entre p_i y todos los substrings representados por $[T_{j...}]$ y del conjunto de distancias se almacenará la mínima. Una vez que se ha realizado esto para todos los sufijos $[T_{j...}]$ y todos los pivotes p_i se tiene un conjunto de k valores de distancias mínimas para cada sufijo. Esto constituye lo que llamamos *firma* del sufijo, lo que puede ser visto como un punto en k dimensiones:

$$[T_{j..}] \quad \rightarrow \quad (\text{mín}(ed([T_{j..}], p_0)), \ldots, \text{mín}(ed([T_{j..}], p_{k-1})))$$

siendo este punto la *coordenada mínima en todas las dimensiones* del conjunto de strings representa-

dos por $[T_{j...}]$. El algoritmo de construcción del *índice métrico* puede verse en la Figura 2.

Generación (Texto T, cantidad de pivotes k)
1. Sean $p_1, \ldots, p_k$ pivotes
2. Para todo j (con $1 \leq j \leq n$)
3. $\quad S_j \leftarrow [T_{j...}]$
4. $\quad$ Para todo i (con $1 \leq i \leq k$)
5. $\quad\quad pivote \leftarrow p_i$
6. $\quad\quad Indice[S_j, pivote] \leftarrow \min(ed(T_{j..j+1}, p_i), \ldots, ed(T_{j..n}, p_i))$

Figura 2: Generación del Indice de búsqueda para un texto T de tamaño n con k pivotes

Como se dijo, la distancia de edición mínima puede ser encontrada en $O(|p_i| \max(|p_i|, |x|))$ tiempo porque no es necesario considerar strings arbitrariamente largos $xy_1, \ldots, y_j$. Dado que se computa la matriz de distancias fila por fila, luego de haber procesado x se tendrá el valor mínimo v visto hasta el momento. Luego no tiene sentido considerar las filas j tales que $|x| + j - |p_i| > v$, dado que de aquí en más la distancia entre p_i y x sólo crecerá o se mantendrá. Por lo tanto se trabaja sólo hasta la fila $j = v + |p_i| - |x| \leq |p_i|$. Por otro lado, cuando se está calculando la distancia entre p_i y un string de la forma $T_{j...j+f}$ siempre se tiene $ed(T_{j...j+f-1}, p_i)$ ya calculada; por lo tanto, no hace falta recalcularla porque con sólo guardar la última fila computada puedo obtener $ed(T_{j...j+f}, p_i)$, calculando solamente la fila correspondiente al caracter $j + f$. Todo esto permite calcular la mínima distancia en $O(|p_i|^2)$.

En la Figura 3 se muestra un ejemplo de índice métrico obtenido a partir del texto "abracadabra". Los pivotes utilizados para calcular la firma de cada sufijo fueron p_1 = "*cad*" y p_2 = "*br*". La estructura $Indice$ almacena la mínima distancia desde cada sufijo al pivote correspondiente. La j-ésima fila corresponde al sufijo $[T_{j...}]$ y la i-ésima columna al pivote p_i, por lo tanto $Indice[j, i]$ contiene la mínima distancia desde $[T_{j...}]$ a p_i.

texto

1	2	3	4	5	6	7	8	9	10	11
a	b	r	a	c	a	d	a	b	r	a

pivotes *p1* = "cad" *p2* = "br"

	p1	*p2*
[T1...]	2	1
[T2...]	3	0
[T3...]	2	1
[T4...]	1	2
[T5...]	0	2
[T6...]	1	2
[T7...]	2	2
[T8...]	2	1
[T9...]	3	0
[T10...]	2	1
[T11...]	2	2

Indice

Figura 3: Indexación del texto 'abracadabra'

4.1. Buscar un patrón

Si se considera la búsqueda de un patrón P dado con a lo sumo r errores se está ante un query por rango de radio r en el espacio métrico de sufijos. Como sabemos, al usar un algoritmo basado en pivotes, debemos comparar el patrón P contra los k pivotes obteniendo así una coordenada k-dimensional $(ed(P, p_1), \ldots, ed(P, p_k))$ (firma).

Sea p_i un pivote dado y $[T_{j\ldots}]$ un sufijo de T, si se cumple que:

$$ed(P, p_i) + r < \text{mín}(ed([T_{j\ldots}], p_i))$$

entonces por la desigualdad triangular se sabe que $ed(P, [T_{j\ldots}]) > r$ para todo substring que esté representado por $[T_{j\ldots}]$. La eliminación de un sufijo como posible respuesta a la consulta realizada puede hacerse usando cualquier pivote p_i. Al buscar qué sufijos podrán ser eliminados se cae, como ya se dijo anteriormente, en un clásico problema de búsqueda por rango en un espacio multidimensional. De hecho los nodos que no pueden ser eliminados son aquellos cuyo semi-espacio tiene una intersección no vacía con el rectángulo $\langle(ed(P, p_1) - r, \ldots, (ed(P, p_k) - r), (ed(P, p_1) + r, \ldots, (ed(P, p_k) + r)\rangle$.

En la Figura 4 se muestra la regla de eliminación usando dos pivotes. En ella puede verse el sufijo $[T_{j..}]$ que contiene un conjunto de puntos que son los substrings representados por él. Para ese sufijo se almacena su mínima distancia a los dos pivotes p_1 y p_2. Esto determina un semi-espacio (zona sombreada en **(b)**) donde caen todas las distancias desde cualquier substring del sufijo a los dos pivotes. El query es un patrón P y una tolerancia r, la cual define un región alrededor de P. Luego de calcular la distancia de P a los pivotes p_1 y p_2 se puede considerar un hiper-cubo (en este caso un cuadrado) de lado $2r + 1$. Si el cuadrado no interseca con el semi-espacio entonces aseguramos que ningún substring del sufijo $[T_{j\ldots}]$ está lo suficientemente cerca de P.

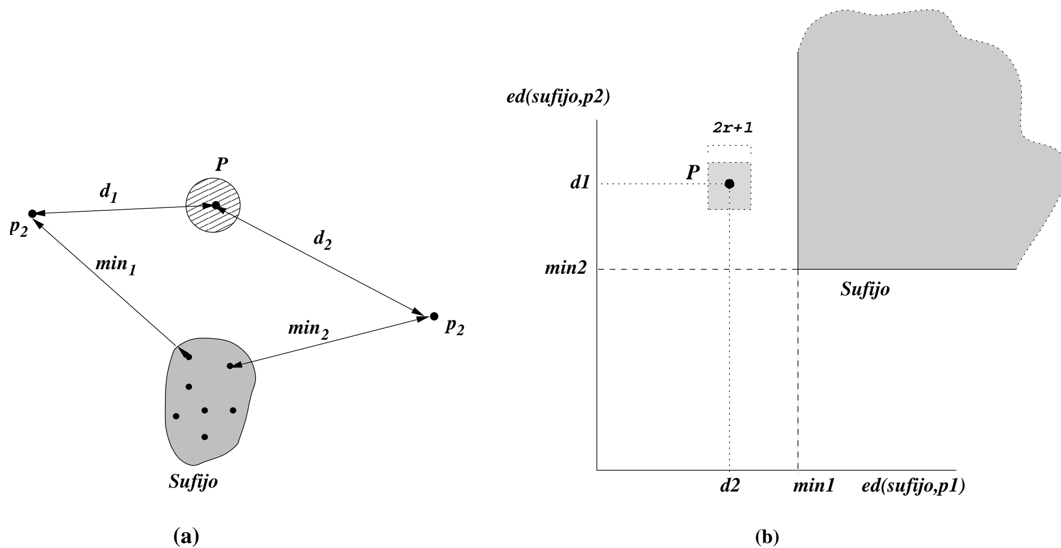

Figura 4: Regla de eliminación usando dos pivotes

Los sufijos que no puedan ser eliminados usando algún pivote p_i se deberán comparar directamente contra el patrón P. Esto implica computar la distancia de edición entre cada substring representado por el sufijo $[T_{j\ldots}]$ y p_i obteniendo luego la mínima. Para aquellos sufijos cuya distancia mínima a P sea a lo sumo r, se reportará la posición de comienzo de esos sufijos en el texto.

En la Figura 5 se muestra el código que permite buscar un patrón P con a lo sumo r errores en un texto T indexado usando el índice propuesto, el que hace uso de una estructura auxiliar P_f para almacenar la firma del patrón.

```
Búsqueda (Indice I, Patrón P, Error r,)
1. Para todo i   (con 1 ≤ i ≤ k)
2.    P_f[i]  ←  ed(P, p_i)
3. Para todo j  (con 1 ≤ j ≤ n)
4.    S_j  ←  [T_{j...}]
5.    Para todo i   (con 1 ≤ i ≤ k)
6.        If P_f[i] + r < I[Sj, p_i] Then
7.                     break      /*descarto [T_{i...}]
8.          else
9.            If ed([T_{i...}], P) ≤ r Then
10.                     reportar j
```

Figura 5: Búsqueda del patrón P con error r.

5. Resultados experimentales

Para evaluar el comportamiento de nuestro índice hemos realizado experimentos sobre distintos textos. Los textos utilizados son un diccionario de palabras en inglés de 584338 caracteres y un diccionario de palabras en francés de 1442394 caracteres. Para los experimentos realizamos búsquedas con distintos patrones de largo 9. En todos los casos los pivotes utilizados para construir el índice también eran de largo 9. En el caso del diccionario de inglés utilizamos 10, 15, 18, 20, 22, 25, 28 y 30 pivotes, y para el diccionario de francés 10, 15, 18, 20 y 22 pivotes.

La Figura 6 muestra la cantidad de strings descartados por el índice, considerando distinta cantidad de pivotes. Los valores mostrados corresponden al promedio sobre 10 búsquedas de distintos patrones. Como puede observarse, a medida que aumentamos la cantidad de pivotes logramos aumentar significativamente la eficiencia del índice.

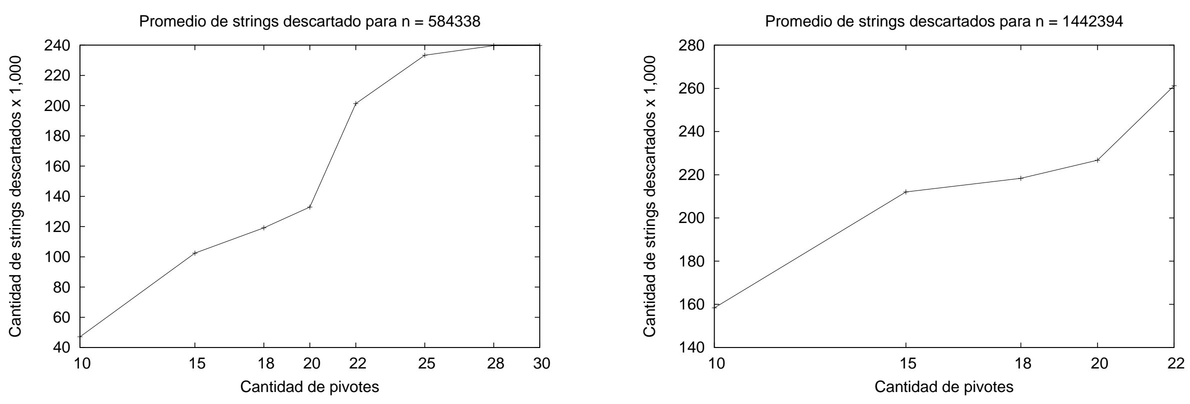

Figura 6: Promedio de strings descartados por el índice considerando distinta cantidad de pivotes.

La Figura 7 muestra la cantidad promedio de caracteres comparados contra el patrón para responder a la búsqueda, considerando distinta cantidad de pivotes. Los valores mostrados corresponden al promedio sobre 10 búsquedas de distintos patrones. Como puede observarse, a medida que aumentamos la cantidad de pivotes gracias al índice no necesitamos compararnos contra todo el texto.

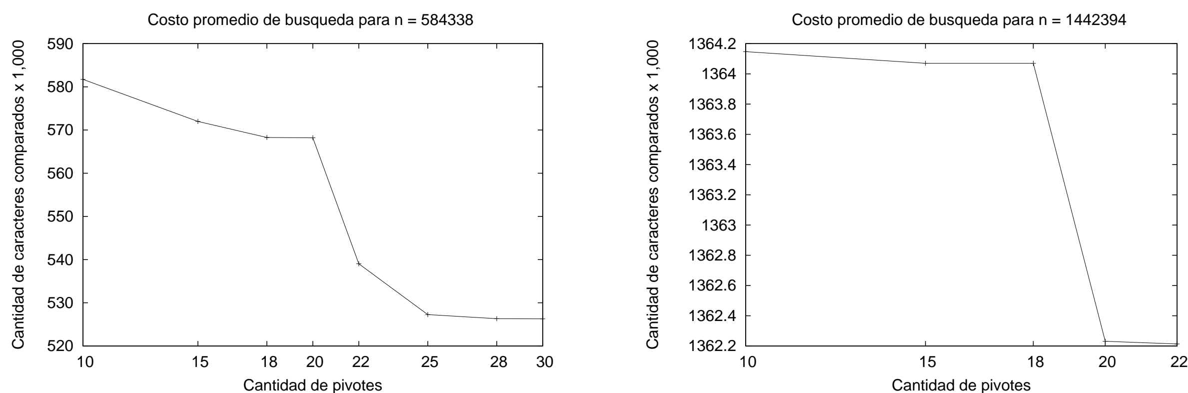

Figura 7: Promedio de caracteres comparados por el índice considerando distinta cantidad de pivotes.

6. Conclusiones y Trabajo Futuro

Hemos mostrado que es posible construir un índice para búsqueda aproximada de patrones utilizando un enfoque distinto basado en búsquedas por rangos en espacios métricos.

Los experimentos hasta ahora realizados han mostrado que gracias al descarte significativo logrado por el índice, no es necesario mirar todo el texto. Para una versión final del artículo trataremos de completar com más resultados experimentales a fin de confirmar si el buen comportamiento del índice se mantiene.

Como se ha observado en los experimentos que algunos pivotes tienen mejor comportamiento que otros, planeamos también estudiar cuáles son sus características con el fin de volcar este conocimiento en la etapa de selección de pivotes.

Estudiaremos a futuro una forma de mejorar esta propuesta basándonos en el conocimiento de que los pivotes con distancias mínimas grandes permiten la eliminación de una mayor cantidad de sufijos. Entonces se podrían almacenar para cada pivote p_i sólo los s valores más grandes de las distancias $\min(ed(p_i, [T_{j...}]))$; siendo s fijado de antemano.

Otra posible mejora a analizar se basa en tomar un primer pivote, determinar sus s sufijos más lejanos, almacenar esos sufijos y sus distancias mínimas en una lista en forma ordenada y luego descartar esos sufijos para próximas consideraciones. Este proceso se repetiría para los otros pivotes hasta que todos los sufijos hayan sido incluidos en alguna lista. Esta idea podría ser eficiente y competitiva en una implementación en memoria secundaria.

Referencias

[1] Alberto Apostolico and Zvi Galil, editors. *Combinatorial Algorithms on Words*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[2] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. SPIRE'98*, pages 14–22. IEEE Computer Press, 1998.

[3] R. Baeza-Yates and G. Navarro. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.

[4] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *J. of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.

[5] R. Baeza-Yates and G. Navarro. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms (JDA)*, 1(1):205–239, 2000. Special issue on Matching Patterns.

[6] E. Bugnion, T. Roos, F. Shi, P. Widmayer, and F. Widmer. Approximate multiple string matching using spatial indexes. In *Proc. 1st South American Workshop on String Processing (WSP'93)*, pages 43–54, 1993.

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

[8] E. Chávez and G. Navarro. A metric index for approximate string matching. In *Proc. of the 5th Latin American Symposium on Theoretical Informatics (LATIN'02)*, LNCS 2286, pages 181–195, 2002.

[9] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.

[10] M. Crochemore. Transducers and repetitions. *Theor. Comp. Sci.*, 45:63–86, 1986.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comp. Surv.*, 30(2):170–231, 1998.

[12] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[13] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248, 1991.

[14] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[15] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, pages 935–948, 1993.

[16] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.

[17] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.

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

[19] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.

[20] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.

[21] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

[22] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Comp. Lab., Univ. of California, July 1996.