UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTUDIO COMPARATIVO DE ALGORITMOS INDEXADOS PARA BÚSQUEDA APROXIMADA EN TEXTO

JAVIER BUSTOS JIMÉNEZ

COMISIÓN EXAMINADORA		CALIFICACIONES: NOTA (n^o)	(Letras)	FIRMA
PROFESOR GUÍA SR. GONZALO NAVARRO	:	.....	.....	.....
PROFESOR DE COMISIÓN SR. RICARDO BAEZA-YATES	:	.....	.....	.....
PROFESOR DE COMISIÓN SR. CLAUDIO GUTIERREZ	:	.....	.....	.....
PROFESOR INVITADO SR. JOAO PAULO KITAJIMA	:	.....	.....	.....
NOTA FINAL EXAMEN DE GRADO	:	.....	.....	.....

TESIS PARA OPTAR AL GRADO DE
MAGISTER EN CIENCIAS MENCIÓN COMPUTACIÓN

SANTIAGO DE CHILE
JULIO DE 2002

Resumen

Resumen.

Gracias

Índice General

Capítulo 1

Introducción

El problema de la búsqueda aproximada en texto aparece en una gran cantidad de ramas dentro de las Ciencias de la Computación, aplicándose a recuperación de texto, biología computacional, reconocimiento de patrones, minería de datos, bases de datos multimediales, etc.

El problema se describe de la siguiente manera: dado un texto de largo *n* y un patrón de largo *m* (con $m << n$) encontrar todas las ocurrencias del texto cuya *distancia de edición* con el patrón sea a lo más *k* (el "error permitido"). La distancia de edición entre dos *strings* está definida como el mínimo número de inserciones, reemplazos y/o borrados necesarios para que ambos strings sean iguales.

Una alternativa de resolución de este problema es el preprocesamiento de texto, para construir estructuras de datos que permitan agilizar las búsquedas. A este tipo de preprocesamiento de la información se le conoce como *indexación*. Para este tipo de problema existen tres corrientes de estudio de algoritmos que utilizan distintas estructuras de índice (ver Tabla 1).

El estudio comparativo de estos algoritmos se ha abordado anteriormente [1] pero estas comparaciones han fallado por los siguientes motivos:

- La implementación de dichos algoritmos se ha realizado por distintas personas, por lo cual poseen distintas optimizaciones, distorsionando el resultado de los estudios.
- En muchos algoritmos no se han estudiado debidamente los parámetros que optimizan su funcionamiento.

	Backtracking	**Intermedio**	**Reducción a búsqueda exacta**
Suffix Tree	Simple[4] Sofisticado[15, 2]	Partición en j pedazos[10]	
Suffix Array	Simple[4]	Partición en j pedazos[10]	Partición en $k+s$ pedazos[13, 9]
q-gramas		Partición en j pedazos[8]	Partición en $k+s$ pedazos[13, 9]
q-samples		*q-gramas* aproximados[12]	*q-sampling*[14]

Tabla 1.1: Método según estructura de dato utilizada

Esta tesis se concentra en entender los diversos algoritmos (reimplementándolos de ser necesario), encontrar el rendimiento óptimo para cada uno según sus parámetros y finalmente compararlos para distintos tipos y tamaños de texto y patrón, tolerancia a errores, etc. Esto significa:

- Entender el funcionamiento de los esquemas existentes.
- Obtener implementación eficiente para todos ellos.
- Encontrar los parámetros óptimos de cada índice para cada caso.
- Comparar los distintos índices entre sí.
- Obtener recomendaciones sobre qué índice utilizar según el problema presentado.

1.1 Metodología

La siguiente es la metodología usada para desarrollar la tesis:

- Entender los distintos algoritmos existentes.
- Evaluar la calidad de las implementaciones existentes.
- Implementar (o reimplementar según el caso) los algoritmos que sea necesario para garantizar una calidad uniforme entre ellos.
- Encontrar los valores óptimos de cada parámetro en cada estructura separadamente.
- Comparar *Suffix Tree* y *Suffix Array* como estructura de dato para indexación:

  - Comparar los algoritmos de backtracking (Sobre *Suffix Tree*).
  - Comparar el algoritmo de backtracking simple [4] sobre *Suffix Tree* y sobre *Suffix Array*.

- Buscar el óptimo del algoritmo de partición en j pedazos[10] sobre *Suffix Array*. Este algoritmo incluye backtracking simple ($j = 1$) y reducción a búsqueda exacta ($j = k + 1$) como casos particulares.
- Comparar los algoritmos que funcionan sobre *q-gramas.*
- Comparar los algoritmos que funcionan sobre *q-samples*.
- Realizar un análisis de los resultados obtenidos y a partir de ellos señalar las recomendaciones generales de este estudio.

Los parámetros para las comparaciones entre índices serán los siguientes:

- Texto: Inglés y ADN; de tamaño 1 Mb para comparar los algoritmos de Ukkonen[15] y Gonnet[4]; 10 Mb para comparar los algoritmos sobre *q-gramas* y de 30 Mb para el resto de las comparaciones.
- Patrones: tamaño entre 10 y 200 caracteres.
- Nivel de error: Entre un 10-40$\%$ del tamaño del patrón (medido en caracteres).

Al terminar este estudio se espera contar con un análisis completo de los algoritmos mencionados, obteniendo los óptimos de aplicación para cada uno de ellos, un mapa *espacio utilizado por el índice* v/s *tiempo de búsqueda* según los distintos casos de tamaño del texto (*n*) y patrón (*m*), y nivel de error permitido (*k*); que permita conocer la mejor alternativa en cada caso y recomendar la utilización de algunos de ellos para problemas determinados.

Capítulo 2

Índices

A continuación se presentan una serie de conceptos básicos sobre índices:

2.1 Conceptos Básicos

Trie

Consideremos un alfabeto finito $\Sigma = \{\sigma_1, ..., \sigma_t\}$ de cardinalidad $t \geq 2$. Mediante Σ^* se denotará el conjunto de secuencias o cadenas formadas por símbolos de Σ. [7]

Definición 1 *Dado un conjunto finito de secuencias $X \subset \Sigma^*$, el trie E correspondiente a X es un árbol t-ario definido recursivamente como:*

1. *Si X contiene un sólo elemento (o ninguno) entonces E es un árbol consistente en un único nodo que contiene al único elemento de X (o está vacío).*
2. *Si $\|X\| \geq 2$, sea E_i el trie correspondiente a $X_i = \{y/x = \sigma_i y \in X \wedge \sigma_i \in \Sigma\}$. Entonces E es un árbol t-ario construído por una raíz y los t subárboles $E_1, ..., E_t$; cada uno de los E_i subárboles tiene rotulada su raíz con σ_i (Figura 2.1).*

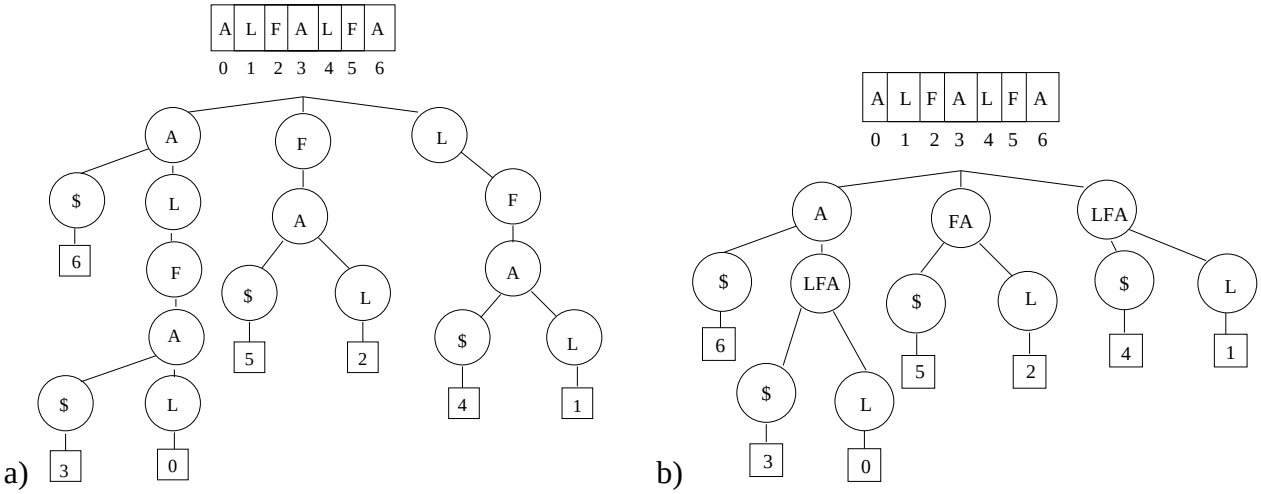


Figura 2.1: a) *Suffix Trie* para la palabra **alfalfa** b) *Suffix Tree* para la palabra **alfalfa**. En ambas el caracter $ indica el fin de palabra.

Suffix Trie

Un *Suffix Trie*[11] es un *Trie* que almacena todos los sufijos de un texto T y cuyas hojas almacenan punteros a la ubicación de ese sufijo en el texto. Cada hoja representa un sufijo y cada nodo interno respresenta un único substring repetido de T. Cada substring de T puede ser encontrando realizando una búsqueda en el *Trie* a partir de la raíz (Figura 2.1).

Suffix Tree

Un *Suffix Tree*[11] corresponde a un *Suffix Trie* cuyos caminos unarios han sido compactados. Los nodos internos almacenan el número de caracteres que han sido compactados y punteros al texto para conocerlos; y las hojas representan la posición en el texto en la que se encuentra el sufijo. Para buscar un string en el *Suffix Tree* se utiliza el mismo esquema que para un *Suffix Trie* (Figura 2.1).

Suffix Array

Si las hojas de un *Suffix Tree* son recorridas de izquierda a derecha, se obtienen todos los sufijos del texto ordenados lexicográficamente. Un *Suffix Array*[11] es un arreglo que contiene todos los punteros a los sufijos del texto ordenados lexicográficamente. El arreglo almacena un puntero por posición en el texto. Para el ejemplo de la Figura 2.1 su *Suffix Array* sería:

6	3	0	5	2	4	1

q-grama

Se conoce como *q-grama*[11] a un string de largo *q*.

Índice de q-gramas

Un índice de *q-gramas*[11] almacena a todos los *q-gramas* de un texto y posee referencias a sus ubicaciones en el texto. Su implementación se puede realizar mediante una tabla de hashing o un trie de altura q donde sus hojas representan las posiciones en el texto (Figura 2.2.a). Para almacenar en el índice los últimos $q-1$ carácteres del texto se completa cada *q-grama* con tantos $ (símbolo que indica el fin de palabra) como sean necesarios.

Índice de q-samples

Un índice de q-samples[11] almacena a todos los *q-gramas* de un texto cuya posición es múltiplo de h ($h \geq q$ definida por cada algoritmo) y posee además referencias a sus ubicaciones en el texto. Su implementación, al igual que el *índice de q-gramas*, se puede realizar mediante una tabla de hashing o un trie de altura q donde sus hojas representan las posiciones en el texto (Figura 2.2.b).

a)

alf	0,3
lfa	1,4
fal	2
fa$	5
a$$	6

b)

al	0
lf	4

Figura 2.2: a) índice de q-gramas para la palabra **alfalfa** con q=3. b) índice de q-samples para la palabra **alfalfa** con q=2 y h=4. Para ambos casos el símbolo $ es un caracter nulo, que indica el fin de palabra.

2.2 Algoritmos de Indexación

A continuación se presentan los algoritmos de indexación estudiados en esta tesis:

2.2.1 Suffix Tree de Stefan Kurtz

Se eligió este *Suffix Tree (ST)*[6] dado que experimentalmente se comprobó que es el que menor espacio necesita para indexar un texto: 12 veces el tamaño para DNA y 9.6 veces para texto en inglés. Otras implementaciones crean un *Suffix Tree* de 20 veces el tamaño del texto, lo que las hace generalmente imprácticas.

Kurtz [6] plantea que cada nodo no terminal del árbol de sufijos consiste de 5 componentes (w es el substring, $\overline{w}$ el camino que lo representa en el *ST*):

1. *firstchild*: se refiere al primer hijo del nodo que representa a la cadena w (el camino desde la raíz hasta $\overline{w}$).
2. *branchbrother*: se refiere al hermano de la derecha de $\overline{w}$ (el siguiente lexicográficamente), si no hay hermano entonces es un nodo nulo.
3. *depth*: se refiere a la profundidad del camino que representa a w.
4. *headposition*: se refiere a la posición de inicio de w en el texto, es decir: la posición de inicio de w como prefijo más corto de algún otro sufijo (el menor lexicográficamente).
5. *suffixlink*: sea un nodo intermedio $\overline{v}$ tal que, si w es de la forma av para algún $a \in \Sigma$ y $v \in \Sigma^*$ entonces el *suffixlink* es una arista que conecta $\overline{w}$ a $\overline{v}$.

Por lo tanto, cada búsqueda se realiza vía *firstchild*, *branchbrother*, *suffixlink*, y referencias a las hojas. Las posiciones en el texto para chequear las ocurrencias se obtienen mediante *depth* y *headposition*.

Además, Kurtz observó que para los caracteres de una cadena representada por el nodo $\overline{wu}$, los caracteres de u se pueden obtener eliminando los primeros *depth(w)* caracteres desde la posición inicial de wu; en otras palabras si $\overline{wu}$ es un nodo intermedio, y $u = x_i...x_{i+l-1}$ entonces $i = headposition(\overline{wu}) + depth(\overline{w})$ y $l = depth(\overline{wu}) - depth(\overline{w})$.

En [6], Kurtz nota que la información que se almacena en los nodos intermedios posee redundancia, para eliminar esta redundancia se dividen los nodos en *large* y *small*. Los nodos son numerados desde la raíz hacia las hojas, de izquierda a derecha (Breadth First Search); los nodos *small* cumplen con la siguiente propiedad:

Si $\overline{aw}$ es $small$, entonces $nodenum(\overline{aw}) + 1 = nodenum(\overline{w})$.

Y los nodos $large$ cumplen la siguiente propiedad:

Si $\overline{aw}$ no es $small$ y $nodenum(\overline{aw}) > 1$, entonces $\overline{aw}$ es $large$.

La raíz no es ni $small$ ni $large$.

Se define como cadena de nodos a la secuencia contigua (en numeración) de nodos $b_l, ..., b_r$ con $l \leq r$ tal que:

- b_{l-1} no es un nodo $small$.
- $b_l, ..., b_{r-1}$ son nodos $small$.
- b_r es un nodo $large$.

Entonces se cumplen las siguientes propiedades:

1. $depth(b_i) = depth(b_r) + (r - i)$.
2. $headposition(b_i) = headposition(b_r) - (r - i)$.
3. $suffixlink(b_i) = b_{i+1}$.

Las tres propiedades anteriores señalan que no es necesario almacenar $depth(b_i)$, $headposition(b_i)$ ni $suffixlink(b_i)$ para todo $i \in [l, r-1]$; puesto que si se conoce la distancia entre b_i y b_r (en rigor $r - i$), pueden ser calculados en tiempo constante.

Por lo tanto, se utilizan 2 enteros (de 4 bytes cada uno) para almacenar los nodos $small$ y 4 enteros (de 4 bytes cada uno) para almacenar los nodos $large$. Experimentalmente en [6] se muestra que cada nodo intermedio es seguido por un nodo $small$ y que sólo el nodo anterior a una hoja es $large$. Además, Kurtz plantea no utilizar una estructura arbórea para almacenar los nodos, sino más bien una estructura de datos con dos arreglos para almacenar los nodos internos y las hojas y con los campos necesarios para almacenar la información del último nodo visitado. Los caminos entre nodos son simulados por operaciones algebraicas no atómicas, lo cual influye de mala forma en el rendimiento de los algoritmos de búsqueda aproximada: existe un costo extra elevado por pasar de un nodo padre a un nodo hijo y uno mucho mayor por seguir un *suffix link*.

Las siguientes son las fórmulas que plantea Kurtz para simular el árbol:

```
#define GETCHILD(B)          ((((*(B)) << 2) & MAXINDEX) | \
                             ((*((B)+1)) >> 30))

#define GETBROTHER(B)        ((*((B)+1)) & (MAXINDEX | NILBIT))
#define LEAFBROTHERVAL(V)    ((V) & (MAXINDEX | NILBIT))

#define GETHEADPOS(B)        (*((B)+3) & MAXTLEN)

#define GETLEAFINDEX(V)      ((V) >> 1)
#define GETBRANCHINDEX(V)    (V)

#define NODEADDRESS(N)       ((Uint) ((N) - state->branchtab))
```

La siguiente es la función para obtener la profundidad de un nodo:

```
static Uint getdepth(Uint *btptr)
{
Uint thirdval = *(btptr+2);
if(thirdval & SMALLDEPTHMARK)
  {
  return thirdval & SMALLDEPTH;
  }
else
  {
  return thirdval & MAXTLEN;
  }
}
```

Las siguientes son las funciones para obtener el *suffix link* de un nodo en tiempo de búsqueda:

```
Uint getlargelinkafterconstruction(struct MccState *state,
                                   Uint *btptr,Uint depth)
{
Uint succ;
if(depth == 1) return 0;
if(ISSMALLDEPTH(depth))
  {
  return (((*(btptr+2) & LOWERLINKPATT) >> SMALLDEPTHBITS) |
           ((*(btptr+3) & MIDDLELINKPATT) >> SHIFTMIDDLE)    |
         ((state->leafbrother[GETHEADPOS(btptr)] & EXTRAPATT)
         >> SHIFTHIGHER)) << 1;
  }
succ = GETCHILD(btptr);
```

```
14 while(!NILPTR(succ))
15   {
16   if(ISLEAF(succ))
17       {
18       succ=LEAFBROTHERVAL(state->leafbrother[GETLEAFINDEX(succ)]);
19       }
20   else
21       {
22       succ=GETBROTHER(state->branchtab + succ);
23       }
24   }
25 return succ & MAXINDEX;
26 }
27
28 Uint f(struct MccState *state, Uint *btptr, Uint depth)
29 {
30 Uint suffixlink;
31 if(ISLARGE(*btptr))
32     {
33     suffixlink=getlargelinkafterconstruction(state,btptr,depth);
34     }
35 else
36     {
37     suffixlink=NODEADDRESS(btptr) + SMALLINTS;
38     }
39 return suffixlink;
40 }
```

2.2.2 Suffix Array

Para el desarrollo de esta tesis se estudiaron dos algoritmos de construcción de *Suffix Array*; el primero utiliza el algoritmo $quicksort$ para ordenar las posiciones de inicio de los sufijos; este algoritmo tiene un sobrecosto de 4 veces el tamaño del texto: cada letra del texto se convierte ahora en una posición y el conjunto de posiciones es ordenado lexicográficamente.

El segundo, proporcionado por Larsson y Sadakane, utiliza el algoritmo *quicksort con mediana de 3* para el ordenamiento de trozos grandes del arreglo y $bucketsort$ para trozos pequeños. Además, el algoritmo almacena el texto en un arreglo de tamaño $4n$ (si se toma en cuenta que cada caracter es 1 byte, y cada entero 4 bytes) y construye el arreglo de sufijos en otro arreglo de tamaño $4n$. En

el arreglo donde se tenía el texto se almacena después la inversa del arreglo de sufijos (es decir: si en la casilla 0 del arreglo de sufijos se almacenaba el número 28, quiere decir que en la casilla número 28 de la inversa se almacena 0; en otras palabras, en la inversa se almacena la posición en el *Suffix Array* de cada sufijo).

La información extra almacenada por el algoritmo de Larsson y Sadakane en ningún modo mejora el rendimiento de los algoritmos de búsqueda, sólo mejora su tiempo de construcción. Una vez terminada la construcción el arreglo auxiliar se puede descartar.

2.2.3 Índices de q-gramas y q-samples

Para implementar estos dos índices se utilizó un *Trie* de altura q, en cuyas hojas se tiene una lista con todas las apariciones del *q-grama* en el texto. La diferencia entre uno y otro recae en que en el primero la distancia entre *q-gramas* es 1 y en el segundo es un parámetro h entregado al momento de construir el índice.

Durante la experimentación se notó que aunque el computador donde se desarrollaba la construcción del índice mostraba la cantidad de memoria suficiente para trabajar, el índice no se podía construir por falta de memoria. Realizando un seguimiento de la construcción se notó que el problema lo producía el comando `realloc`, que dado un puntero le re-asigna la memoria utilizable a ese puntero y además copia el contenido; en otras palabras, si el puntero `p` tiene asignado 1024 bytes y se realiza el llamado `realloc(p,2048)` entonces el computador busca un bloque de 2048 bytes y una vez que lo encuentra copia el contenido de los 1024 bytes anteriores a la nueva dirección y recién ahí libera el bloque anterior; por lo tanto la cantidad real de memoria utilizada en el proceso es de 3072 bytes.

Además, dado que los $q-gramas$ siguen la Ley de Zipf [11], se producía un sobrecosto en tiempo de construcción al realizar la copia de las posiciones.

Se decidió por lo tanto construir el *Trie* en dos pasos: el primero determinaba cuántas veces se repetía cada q-grama y el segundo pedía la cantidad exacta de memoria para almacenar en un arreglo la lista de esas posiciones. Eso llevó no sólo a una mejora en la cantidad real de memoria utilizada por el algoritmo sino además, como efecto colateral de no utilizar `realloc`, a una mejora en el tiempo de construcción, como se puede apreciar en la Tabla 2.1.

Texto (MB)	Tiempo c/realloc (seg)	Tiempo s/realloc (seg)
1,00	9,63	9,74
2,00	19,13	19,25
3,00	30,34	28,53
4,00	42,73	38,88
5,00	63,04	47,40
6,00	77,83	57,72
7,00	95,65	66,31
8,00	107,59	78,08
9,00	126,37	86,94
10,00	147,49	96,39

Tabla 2.1: Tiempo de construcción de un índice de qramas con q=8. El texto está medido en MegaBytes y los tiempos en segundos.

2.3 Comparación entre índices

Para comparar los índices se tomó en consideración sólo 2 parámetros: MegaBytes (MB) de memoria utilizado por el índice y tiempo de construcción; para la experimentación se utilizó dos tipos de textos: DNA del homo sapiens[3] y *texto en inglés* obtenido de la base de datos del *Wall Street Journal*[5]. En ambos casos, se utilizaron textos de prueba que variaban entre 1 y 30 MB, en diferencias de 1 MB uno de otro.

Capítulo 3

Algoritmos de Búsqueda

En la primera parte de este capítulo se presentan los conceptos básicos para comprender los algoritmos de búsqueda y la definición y cálculo de la distancia de edición. En los siguientes capítulos se presentan los algoritmos divididos por tipo de enfoque.

3.1 Conceptos Básicos

3.1.1 Lemas y Definiciones

Algunos de los índices presentados en esta tesis se basan en los siguientes lemas:

Lema 1 *[1] Si un patrón calza con un string con k errores, entonces se puede dividir el patrón en j partes, alguna de las cuales aparecerá en el string con a lo más $\lfloor k/j \rfloor$ errores.*

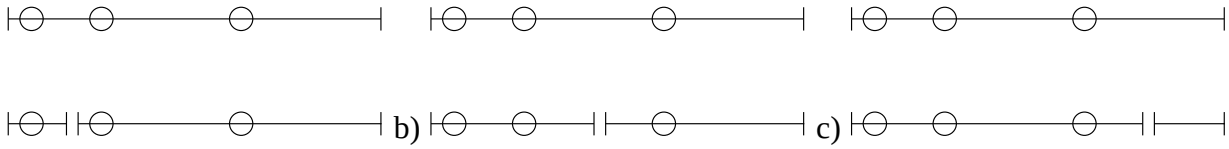


Figura 3.1: La figura muestra la división en dos partes de un patrón que calza en el texto con $k = 3$. a), b) y c) muestran que cualquiera sea la división, siempre aparece a lo menos un trozo con error a lo más $\lfloor k/j \rfloor$.

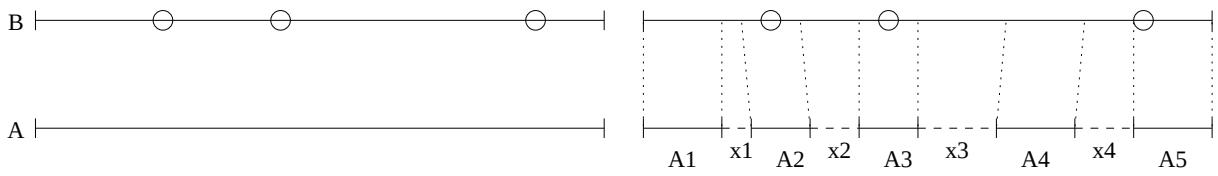


Figura 3.2: La figura muestra un ejemplo del lema 2: la distancia de edición entre A y B es de 3 y se aplica el lema con $s = 2$. Esas s piezas aparecen en B sin error.

Lema 2 *[11] Sean A y B dos strings cuya distancia de edición es d(A,B)*$\leq k$.
Sea $A = A_1 x_1 A_2 x_2 ... x_{k+s-1} A_{k+s}$ *con* k *y* $s \geq 1$. *Entonces, a lo menos s strings* $A_{i_1} ... A_{i_s}$ *aparecen en B. Más aún, su distancias relativas dentro de B no difieren de aquellas dentro de A en más de k.*

Lema 3 *[11] Si P ocurre con k errores y terminando en* $T_{...i}$, *entonces al menos* $m + 1 - (k + 1)q$ *q-gramas de P ocurren en* $T_{i-m-k+1...i}$.

Figura 3.3: La figura muestra ejemplos del lema 3 con $q = 3$: a) Si hay una búsqueda exacta ($k = 0$) exitosa en el texto, entonces todos los *q-gramas* de P aparecen en el el texto. b) Si hay una búsqueda aproximada con $k = 1$ exitosa, entonces se eliminan en el peor caso, qk *q-gramas*.

Lema 4 *[11] Sean A y B dos strings tal que* $d(A, B) \leq k$, *sea* $A = A_1 x_1 A_2 x_2 ... x_{j-1} A_j$, $\forall A_i, x_i$ *strings y* $j \geq 1$. *Entonces, a lo menos un string* A_i *aparece en B con a lo más* $\lfloor k/j \rfloor$ *errores.*

Definición 2 *Backtracking: Técnica para encontrar soluciones basada en la búsqueda a partir de una de varias selecciones. Si ésta arroja un resultado negativo la búsqueda "retrocede" hasta el punto de la elección y se realiza a partir de ese punto una nueva búsqueda utilizando otra preferencia.*

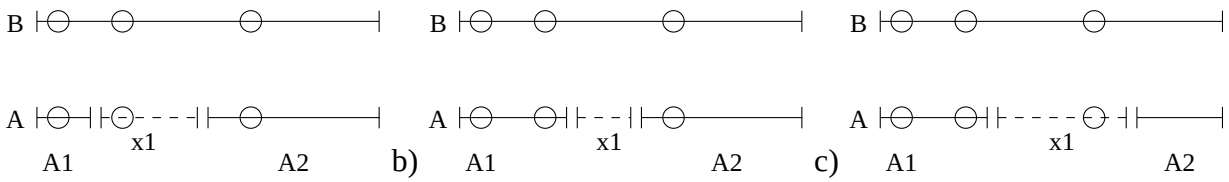


Figura 3.4: La figura muestra un ejemplo del Lema 4 con $k = 3$ y $j = 2$. a), b) y c) muestran que cualquiera sea la división, siempre aparece a lo menos un trozo A_i con error a lo más $\lfloor k/j \rfloor$.

Definición 3 *Reducción a búsqueda exacta: Técnica para encontrar soluciones basado en el Lema 1; se divide el patrón en j pedazos tal que $\lfloor k/j \rfloor = 0$, y luego se aplican condiciones de filtro y chequeo sobre todas las posiciones que reportan las diferentes búsquedas exactas de los j pedazos.*

3.1.2 Distancia de Edición

Se define la distancia de edición entre dos strings x e y (en adelante $d(x, y)$) como la cantidad inserciones, reemplazos y/o borrados de caracteres que se le deben aplicar a x para que sea idéntico a y.

Para el cálculo de la distancia de edición se utilizó en esta tesis la técnica basada en programación dinámica[11]: Se llena una matriz $C_{0...|x|,0...|y|}$, donde donde $C_{j,i}$ representa el mínimo número de operaciones que se necesitan para que $x_{1...j}$ calce en $y_{1...i}$ en forma exacta. El cálculo se realiza de la siguiente manera:

- $C_{j,0} = j, C_{0,i} = i$
- $C_{j,i} =$ si $(x_j = y_i)$ entonces $C_{j-1,i-1}$ sino $1 + min(C_{j-1,i}, C_{j,i-1}, C_{j-1,i-1})$

Dado que la inserción en una cadena es equivalente al borrado en la otra $C_{j-1,i-1} \leq 1+min(C_{j-1,i}, C_{j,i-1}, C_{j-1,}$ lo que lleva a modificar la recurrencia (para reducir el número de operaciones y ganar en eficiencia) a la siguiente fórmula (Figura 3.5.a):

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

Donde $\delta(a, b)$ es 0 si $a = b$ y 1 si son distintos.

Los algoritmos estudiados en esta tesis que utilizan este tipo de cálculo son *backtracking simple*Gonnet[4] y *partición en j pedazos (Suffix Tree y Suffix Array)*[10] para la búsqueda; y *partición en j pedazos*

a)

		t	e	c	i	t	o	s
	0	1	2	3	4	5	6	7
t	1	0	1	2	3	4	5	6
e	2	1	0	1	2	3	4	5
s	3	2	1	1	2	4	4	4
i	4	3	2	2	1	2	3	4
s	5	4	3	3	2	2	3	3

b)

		t	e	c	i	t	o	s
	0	0	0	0	0	0	0	0
t	1	0	1	1	1	0	1	1
e	2	1	0	1	2	1	1	2
s	3	2	1	1	2	2	2	1
i	4	3	2	2	1	2	3	2
s	5	4	3	3	2	2	3	3

Figura 3.5: a) Matriz de programación dinámica para calcular la distancia de edición entre $tesis$ y $tecitos$. b) La variación del cálculo, para buscar el patrón $tesis$ en el texto $tecitos$.

(q-gramas)[8] en la construcción de los patrones de búsqueda.

El cálculo anterior se utiliza para la distancia de edición entre dos cadenas de caracteres, sin embargo al buscar interesa la distancia contra cualquier substring de (T), por lo tanto se debe permitir que cualquier posición del texto sea un potencial inicio de calce. Esto se logra haciendo $C_{0,i} = 0$ $\forall i \in 0, .., n$ (n es el tamaño del texto). En otras palabras, el patrón "vacío" calza sin errores en cualquier posición del texto.

Para realizar el cálculo se procede de la siguiente manera (Figura 3.5.b):

- $C_{j,0} = j, C_{0,i} = 0$
- $C_{j,i} =$ si $(P_j = T_i)$ entonces $C_{j-1,i-1}$ sino $1 + min(C_{j-1,i}, C_{j,i-1}, C_{j-1,i-1})$

Los algoritmos estudiados en esta tesis que utilizan este tipo de cálculo son los de backtracking modificado[15] para la búsqueda; *q-gramas* aproximados[12] en la construcción de los patrones de búsqueda y chequeo; y Partición en j pedazos[10, 8], Partición en $k + s$ pedazos[13, 9] y *q-sampling*[14] en el chequeo de las posiciones reportada en cada búsqueda.

La programación dinámica debe llenar la matriz de tal forma que antes de calcular el valor de una celda se debe ya tener los valores de las celdas ubicadas inmediatamente arriba, a la izquierda y arriba-izquierda (Figura 3.5). Se puede deducir de ahí que la matriz se puede recorrer por filas de izquierda a derecha o por columnas de arriba hacia abajo.

Esta técnica utiliza un espacio extra $O(min(|x|, |y|)$. Esto debido a que si se recorre por columnas, sólo la anterior columna es necesaria para calcular la nueva (análogo por filas).

Para determinar un calce en una búsqueda aproximada con un k dado, la última fila de la matriz debe contener algún valor menor que k. Dado que los valores almacenados en las filas son no decrecientes, para determinar que un patrón no calza con un texto sólo se debe verificar que todos los valores de una fila sean mayores que k; por lo tanto, no es necesario calcular los valores de toda la matriz para eliminar una búsqueda. Lo mismo se aplica a las columnas si se calcula la distancia de edición entre dos cadenas.

3.2 Backtracking

Los algoritmos estudiados en esta tesis y que utilizan esta técnica son los algoritmos de Backtracking Simple[4] y Backtracking Sofisticado[15, 2].

3.2.1 Backtracking Simple

El algoritmo de búsqueda consiste en hacer $backtracking$ sobre el *Suffix Tree* o el *Suffix Array* construído a partir del texto. Se debe determinar cuándo hay calce (y reportar todo el subárbol o rango) o cuándo no es necesario seguir considerando una búsqueda.

Backtracking Simple sobre Suffix Tree

Se recorre el *Suffix Tree* desde la raíz hacia las hojas, bajando por las ramas en orden lexicográfico. Se calcula la distancia de edición entre el patrón y todas las cadenas representadas por un camino en el

Suffix Tree, utilizando la matriz de distancia de edición de la Figura 3.5.a.

Mientras se chequea el texto perteneciente a un nodo, se puede verificar que toda la columna (o fila, dependiendo de como se llene la matriz) sea menor o igual al error buscado, de no ser así se abandona la búsqueda por ese nodo y se regresa al nodo anterior (backtracking).

Si el último valor de la columna (o fila) es menor o igual al error buscado, quiere decir que hay un calce y por lo tanto se reporta todo el subárbol a partir de ese nodo. Luego se abandona la búsqueda por ese nodo y se continúa desde el nodo anterior (backtracking).

Puede suceder que mientras se baja por el árbol chequeando nodo por nodo, se llegue a una hoja. En tal caso, se debe ir al texto y continuar el chequeo desde la posición que indica la hoja. En caso de calce se reporta esa posición.

Un ejemplo práctico se puede apreciar en la Figura 3.6. Se busca el patrón FAL en el *Suffix Tree* del texto ALFALFA con un error ($k = 1$). En la figura las columnas representan el patrón y las filas el texto que se forma al decender por el árbol.

La matriz de distancia de edición se va llenando por columnas, em Figura 3.6.a se aprecia que la primera letra (A) no elimina ese nodo (existe en esa columna un valor menor o igual a uno) y por lo tanto se chequean sus hijos. Al momento de chequear el patrón con el texto formado por el primer nodo (A$) sí se descarta la búsqueda por ese nodo y se continúa por el siguiente (ALFA); aunque en el dibujo la matriz está calculada para el patrón completo se debe notar que el texto AL calza con el patrón FAL con un error (segunda columna, cuarta fila), por lo tanto se reporta ese sub-árbol.

En Figura 3.6.b se puede ver como el patrón FAL calza con un error en el texto FA (la última fila de la última columna chequeada es menor o igual que k) y por lo tanto se reporta todo ese subárbol.

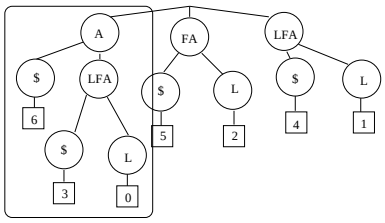
En Figura 3.6.c se puede ver como el comparar FAL con LFA no arroja calce pero tampoco descarta ese nodo para seguir la búsqueda. Al bajar al nodo que representa el texto LFA$ sí es descartada la búsqueda por ese camino. Al bajar entonces al nodo que representa el texto LFAL sí hay calce y se reporta ese sub-árbol (que en este caso es una hoja con la posición del sufijo).

Backtracking Simple sobre Suffix Array

La idea es simular una búsqueda sobre un *Suffix Tree* utilizando el *Suffix Array*. Para ello se debe ingresar en todos los intervalos del arreglo de sufijos (utilizando programación dinámica para determinar la validez de ese intervalo y/o reportar un calce), delimitando los intervalos de acuerdo a la siguiente letra del texto.

Una forma sencilla es tomar el primer elemento del intervalo, ir al texto y ver qué letra le corresponde y luego hacer búsqueda binaria en el intervalo para determinar dónde terminan los sufijos que empiezan con esa letra. Eso entrega el primer intervalo. Luego se mira la casilla que le sigue a ese primer intervalo en el arreglo y se realiza la misma operación hasta determinar todos los intervalos. Finalmente se ingresa a cada intervalo, chequeando en cada uno de ellos que se pueda continuar la búsqueda (o que se deba reportar el intervalo). Sólo en los intervalos en que sea válido continuar

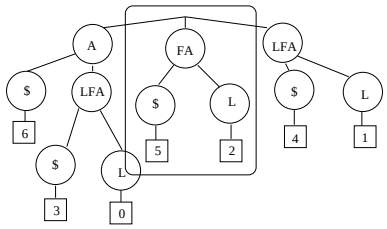
a)



		A	$
	0	1	2
F	1	1	2
A	2	1	2
L	3	2	2

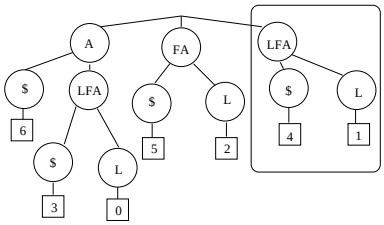
		A	L	F	A
	0	1	2	3	4
F	1	1	2	2	3
A	2	1	2	3	2
L	3	2	1	2	3

b)



		F	A
	0	1	2
F	1	0	1
A	2	1	0
L	3	2	1

c)



		L	F	A	$
	0	1	2	3	4
F	1	1	1	2	3
A	2	2	2	1	2
L	3	2	3	2	2

		L	F	A	L
	0	1	2	3	4
F	1	1	1	2	3
A	2	2	2	1	2
L	3	2	3	2	1

Figura 3.6: a), b) y c) son tres pasos sucesivos de una búsqueda con Backtracking Simple sobre un *Suffix Trie*.

con la búsqueda se repite el proceso.

El algoritmo estudiado presentaba dos mejoras en su implementación:

1. *No comenzar en la primera casilla sino la del medio*. Así las búsquedas binarias se reducen más rapidamente: se determina el intervalo que contiene la casilla del medio y se sigue con los dos pedazos de la izquierda y la derecha.
2. *Si el intervalo es muy corto no se hace búsqueda binaria sino secuencial*.

Por ejemplo, en el capítulo 2 se determinó que para la palabra `alfalfa` su arreglo de sufijos es:

6	3	0	5	2	4	1

Si realizamos la búsqueda del patrón `alf` sin errores (k=0) utilizando el algoritmo de Gonnet, entonces los pasos son los siguientes:

Primero: Se determinan los intervalos para una letra (se parte por la casilla del medio, en este caso: `5`).

6	3	0	5	2	4	1

Segundo: Se chequea la primera letra del patrón (`a`) para cada intervalo, lo cual elimina el segundo y tercer intervalo.

Tercero: Se repite el procedimiento para el primer intervalo, subdividiéndolo en intervalos para 2 letras (comenzando por la casilla del medio, en este caso: `3`).

6	3	0	5	2	4	1

Cuarto: Se chequea la segunda letra del patrón (`l`) en cada intervalo, lo cual elimina al primer subintervalo.

Quinto: Se repite el procedimiento para el primer subintervalo, pero en este caso aquél queda intacto.

Sexto: Se chequea la tercera letra del patrón (`f`) en este subintervalo, como hubo calce se reportan las posiciones 0 y 3.

3.2.2 Backtracking Sofisticado

Ukkonen observó en [15] que al realizar bactracking simple sobre un *Suffix Tree* muchas veces se reportaban posiciones que correspondían a la inserción al principio de otro calce. Por ejemplo en la Figura 3.6 se aprecia que backtacking simple reporta para el patrón `FAL` la posición del texto `LFAL` y del texto `FAL`; obviamente la interesante para una búsqueda es la segunda, porque es más exacta. En índices sobre grandes textos eso influye de mala forma en el desempeño del algoritmo, puesto que se referencian muchas posiciones que en la práctica no son útiles.

Ukkonen plantea en [15] que el estado de una búsqueda en cierto punto del texto depende de los caracteres leídos anteriormente, y además que las posiciones *interesantes* son las que evitan la inserción de carácteres extra al principio. Él llama *prefijos viables* a los substrings que pueden ser prefijos *interesantes* de una ocurrencia aproximada ($k > 0$) del patrón.

Para encontrar estos *prefijos viables*, Ukkonen propone eliminar la primera letra del prefijo representado por el camino recorrido en el *Suffix Tree* utilizando el *suffix link*, pasando transversalmente entre los nodos del árbol y eliminando así la primera letra. Además, se debe verificar que el nuevo prefijo pueda ser catalogado como *prefijo viable* y de ser así continuar desde ese punto la búsqueda; para ello se utiliza una nueva matriz L definida como:

- $L_{0,i} = j$
- $L_{j,i} =$ si $C_{j,i} = C_{j-1,i} + 1$ entonces $L_{j-1,i}$
  sino, si $C_{j-1,i-1} = C_{j-1,i-1} + d(p_j, t_i$ entonces $L_{j-1,i-1} + 1$
  sino $L_{j,i-1} + 1$

$L_{j,i}$ representa al tamaño de la cadena más corta de un texto T que termina en t_i cuya distancia de edición con P_j (las primeras j letras del patrón P) es igual a $C_{j,i}$ (Figura 3.7). El tamaño del prefijo viable para una columna i es $L_{h,i}$ donde h es el mayor índice donde $C_{h,i} \leq k$.

a)

		a	a	b	b	b	b	b
	0	0	0	0	0	0	0	0
a	1	0	0	1	1	1	1	1
b	2	1	1	0	1	1	1	1
b	3	2	2	1	0	1	1	1
b	4	3	2	1	0	1	1	1

b)

		a	a	b	b	b	b	b
	0	0	0	0	0	0	0	0
a	1	1	1	0	0	0	0	0
b	1	1	1	2	1	1	1	1
b	1	1	1	2	3	2	2	2
b	1	1	1	2	3	4	3	3

Figura 3.7: Si $T = aabbbbb$ y $P = abbb$ entonces en el algoritmo de Ukkonen a) representa a la matriz C y b) representa a la matriz L.

El algoritmo que plantea Ukkonen en [15] se basa en el cálculo de columnas de las matrices `C` y `L` independiente de cómo se llegó a un nodo y del marcar los nodos ya visitados para no repetir una búsqueda. Para efectos de mejor entendimiento se explicará el algoritmo sobre un *Suffix Trie* y luego se portará a un *Suffix Tree*.

Lo primero que se hace es eliminar la raíz, para que cualquier búsqueda vía *suffix link* termine al llegar a ella, e inicializar las primeras columnas de `C` y `L`. Luego desde la raíz se procede recursivamente:

1. Para cada nodo hijo no eliminado se calculan las columnas CC y LL (los nuevos valores de las columnas, basándose en C y L) para la letra almacenada en ese nodo.

2. Se calcula el tamaño del *prefijo viable*, esto es el valor de LL(h) donde h es el mayor índice tal que CC(h)$\leq$k.

3. Mientras la profundidad de ese nodo sea mayor que el tamaño del prefijo viable y ese nodo no esté eliminado, se elimina el nodo y se pasa al nodo apuntado por su *suffix link*.

4. Si la profundidad del nodo actual es igual al tamaño del *prefijo viable* y el nodo actual no está eliminado se verifica un calce (CC(m)$\leq$k) y de ser así se agrega el nodo actual a la lista de nodos a reportar. Además, como continúa la búsqueda a partir de este nodo (porque representa un prefijo de un texto *interesante*), se eliminan todos los nodos que van desde el actual a la raíz siguiendo el camino vía *suffix link*.

5. Se repite el algoritmo para el nodo actual pasando como parámetro las columnas CC y LL.

6. Finalmente se reportan todos los sub-árboles de los nodos almacenados en la lista de nodos a reportar.

Para portarlo a un *Suffix Tree* es necesario tener en consideración que:

- Cada nodo puede almacenar más de un carácter, por lo tanto se pueden realizar más de un cálculo de CC y LL por nodo.

- Luego de pasar de un nodo a otro via *suffix link* buscando el *prefijo viable* puede suceder que la profundidad (p) sea menor que el tamaño del *prefijo viable* (len) en ese nodo, por lo tanto el siguiente cálculo de CC y LL se debe realizar desde la $len - p$-ava letra de ese nodo.

- Se deben realizar $p - len$ saltos via *suffix link* para buscar el *prefijo viable* (paso 2), antes de eliminar se debe verificar que el *suffix link* no apunte al mismo nodo.

- Antes de comenzar la eliminación de los nodos alcanzados via *suffix link* (paso 4) se deben verificar primero todos los carácteres almacenados en el nodo actual. De no ser así se corre el riesgo de eliminar un nodo que representa un *prefijo viable* antes que el algoritmo lo chequee, perdiendo una posición *interesante*.

Un ejemplo del algoritmo aplicado sobre el *Suffix Tree* del texto ALFALFA, buscando el texto FAL con un error se presenta en la Figura 3.8:

En la Figura 3.8.a se muestra el inicio del algoritmo, bajando por la rama de menor valor lexicográfico y calculando las columnas de C y L en cada nodo, las únicas diferencias con *backtracking*

simple hasta este punto son el marcar los nodos como visitados (círculos grises) y calcular el tamaño del *prefijo viable* (celdas destacadas) para cada nodo.

Se puede apreciar en la Figura 3.8.b cómo el algoritmo abandonó una búsqueda por la rama A$ porque el tamaño del *prefijo viable* es cero y por lo tanto debe ir eliminando nodos hasta la raíz siguiendo los *suffix link*. La siguiente búsqueda es por el nodo que representa ALFA y se aprecia que este nodo reporta un calce con el texto AL y por lo tanto se agrega a la lista de los nodos a reportar.

En la Figura 3.8.c se continúa la búsqueda ahora con el texto ALF, donde el tamaño del *prefijo viable* (1) es mayor que la profundidad del árbol en ese punto (3), por lo tanto hay que abandonar ese nodo. La Figura 3.8.d muestra la eliminación de los nodos vía *suffix link* hasta encontrar la profundidad deseada.

La Figura 3.8.e muestra que a pesar de no continuar la búsqueda en el nodo que representa el texto ALFA sí se utilizan sus columnas para calcular los nuevos valores de C y L. En este caso el nodo reporta un calce y se agrega a la lista de nodos a reportar.

Dos situaciones interesantes se muestra en la Figura 3.8.f: la primera es al marcar el nodo actual como visitado y sabiendo que se continuará la búsqueda desde ese punto, se eliminan los nodos entre él y la raíz alcanzados vía *suffix link*; como ese nodo ya está eliminado no se continúa la iteración. La segunda es la continuación de la búsqueda a partir de los hijos del nodo actual, calculando los nuevos valores de las columnas utilizando sólo las columnas de su padre sin importar los anteriores cálculos.

La Figura 3.8.g muestra que el nodo que representa al texto FA$ calza con un error en el patrón, por lo tanto se agrega a la lista de nodos a reportar y se eliminan los nodos entre él y la raíz via *suffix link*; como el primero (A$) ya está eliminado no se continúa la iteración.

Finalmente, la Figura 3.8.h muestra que el nodo que representa al texto FAL calza sin error en el patrón, por lo tanto se agrega a la lista de nodos a reportar y se eliminan los nodos entre él y la raíz via *suffix link*; como el primero (ALFA) ya está eliminado no se continúa la iteración.

Luego se reportan los nodos perteneciente a la lista, teniendo cuidado de no reportar dos veces el mismo sub-árbol.

Archie Cobbs propone en [2] una modificación al algoritmo de Ukkonen: **generar** los prefijos via-

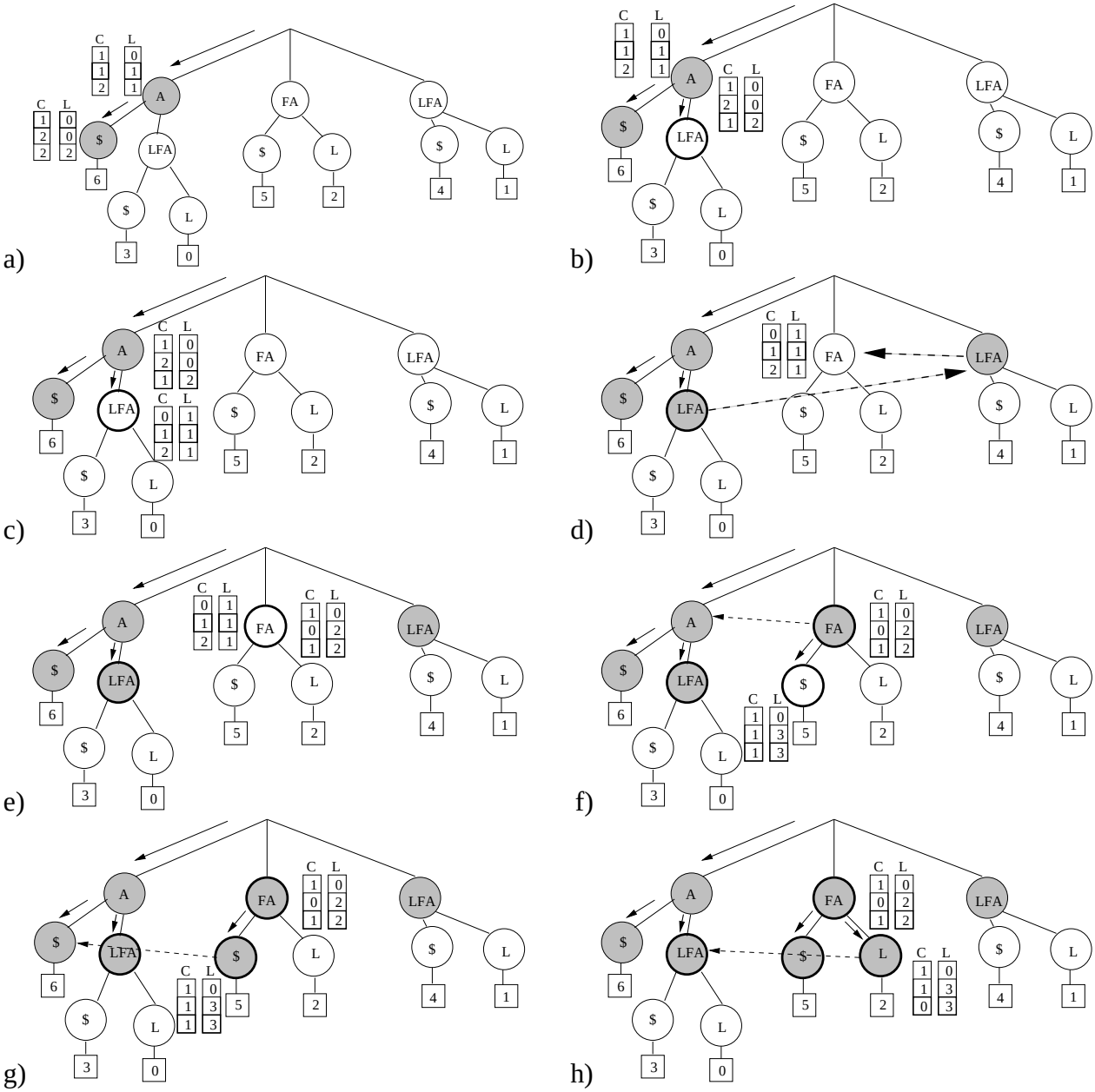


Figura 3.8: Las figuras a),..., h) muestran estados sucesivos del algoritmo de Ukkonen sobre el *Suffix Tree* para el texto `alfalfa` con $k = 1$.

bles **usando** el *Suffix Tree* sólo como lectura y utilizando una tabla de hashing para almacenar la información adicional con los nodos del *Suffix Tree* y luego utilizar búsqueda exacta para encontrar las posiciones de ellos.

Se utiliza la matriz de programación dinámica definida en la Figura 3.5.b y el algoritmo trabaja fila a fila en m rondas. En cada ronda se generan los candidatos extendiendo vertical (borrado) y diagonalmente (reemplazo) a los candidatos de la ronda anterior; y horizontalmente (inserción) a los candidatos pertenecientes a la ronda actual, cada nuevo candidato para una ronda i significa el chequeo de la matriz dinámica hasta la *i-ésima* fila, si todos los valores en esa fila son mayores a k el candidato se desecha. Inicialmente (ronda 0) el único candidato es el string vacío.

Para filtrar los candidatos se utilizan dos reglas: *suffix preemption* y *cosuffix preemption*; la primera se refiere a que se elijen los candidatos que representen a los sufijos más largos, para ello se sigue el camino desde el nodo a la raíz vía *suffix link* y si se encuentra otro sufijo con la misma distancia de edición se elimina el más corto. La segunda se refiere a que si un candidato es sufijo de otro y además posee una distancia de edición mayor, debe ser eliminado del conjunto de candidatos.

Este algoritmo no se estudió en la presente tesis porque dado que Ukkonen planteó el inspeccionar menor cantidad de nodos en [15] y experimentalmente su algoritmo no logró una real mejora del backtracking simple, el algoritmo planteado por Cobbs en [2] seguirá el mismo camino, sino peor. Además, en [11] se demuestra que la implementación de este algoritmo es por lo menos 5 veces peor en tiempo de búsqueda que los algoritmos de *backtracking simple*, sin contar además el espacio extra que utiliza en cada una de ellas.

3.3 Reducción a Búsqueda Exacta

Los algoritmos estudiados en esta tesis que utilizan esta técnica de búsqueda son los algoritmos de partición em $k + s$ pedazos [13, 9], para *Suffix Array* e *índices de q-gramas*; y el algoritmo *q-sampling*[14], para *índices de q-samples*.

3.3.1 Partición en k+s pedazos

Basándose en el Lema 2, Shi[13] divide el patrón en $k + s$ bloques disjuntos (con k la distancia de edición entre el patrón y el texto buscado y $s > 0$). Luego realiza una búsqueda exacta de cada uno de los $k + s$ trozos, aplica un filtro a las posiciones reportadas por ellos y quienes superan esa etapa

son chequeados directamente en el texto.

El filtro aplicado a las posiciones reportadas corresponde a 4 restricciones que se deben cumplir:

1. Al menos s bloques deben aparecer en el texto (Figura 3.9).
2. La ocurrencia de los bloques debe mantener el orden en el texto (Figura 3.10).
3. No debe haber traslape de bloques (Figura 3.11).
4. La distancia entre el segmento de texto al que corresponde el primer bloque y el que corresponde al último no debe ser mayor a *m+k* (con m el largo del patrón) (Figura 3.12).

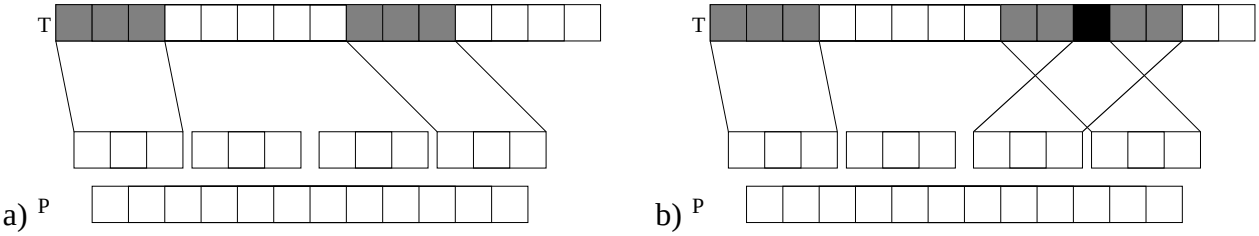


Figura 3.9: Condición 1 del algoritmo de Shi: a) no la cumple, b) sí la cumple

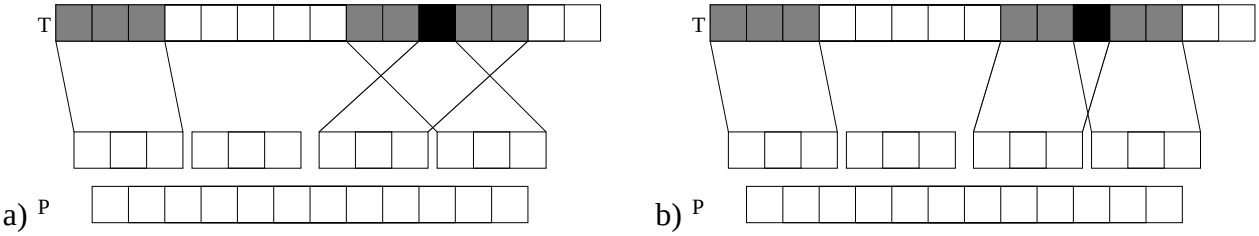


Figura 3.10: Condición 2 del algoritmo de Shi: a) no la cumple, b) sí la cumple

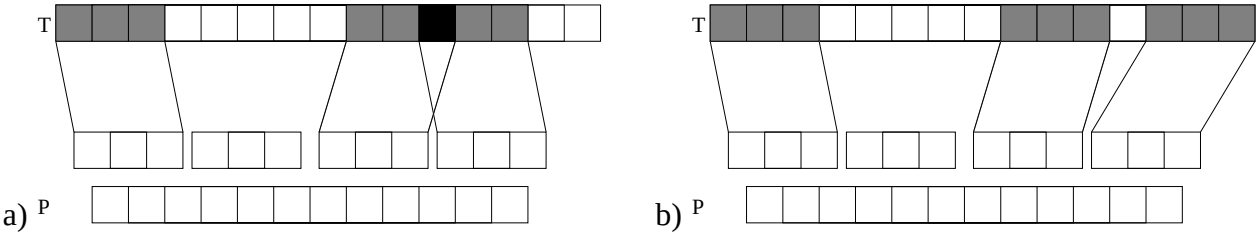


Figura 3.11: Condición 3 del algoritmo de Shi: a) no la cumple, b) sí la cumple

Se busca en el índice la secuencia de s bloques que cumplan las cuatro restricciones y en ese caso se chequea que el patrón calce en el texto con a lo más k errores; para ello se utiliza la matriz

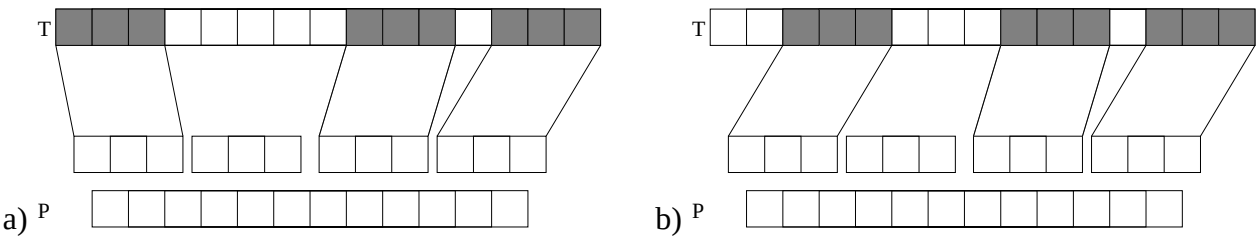


Figura 3.12: Condición 4 del algoritmo de Shi: a) no la cumple, b) sí la cumple

de programación dinámica descrita descita en la Figura 3.5.b, y se chequea desde `pos-k` hasta `pos+m+k`, donde `pos` es la posición de inicio del patrón en el texto, `m` el tamaño del patrón y `k` el máximo error permitido.

Baeza-Yates y Navarro proponen en [9] una simplificación, basándose en Lema 2: dividen el patrón en $k + 1$ partes, por lo que no es necesario verificar las tres últimas condiciones anteriores.

3.3.2 q-sampling

Erkki Sutinen y Jorma Tarhio proponen en [14] un nuevo esquema de búsqueda utilizando un índice de *q-samples* como estructura de datos para almacenar el texto.

La idea principal es utilizar el Lema 2 pensando que los errores ocurren en el texto y A serán las ocurrencias de P en T. En este caso los *q-samples* ($T_{1,\ldots,q}, T_{h+1,\ldots,h+q}, T_{2h+1,\ldots,2h+q}$) del índice corresponderán a los bloques A_i y el espacio entre *q-samples* a x_i y queremos que al menos s de estos *q-samples* aparezcan en P.

Sabemos además que un patrón posee $m - q + 1$ q-gramas, y que el mínimo tamaño de una ocurrencia de P en T es de $m - k$; por lo tanto la mayor distancia que nos asegura que se encontrarán s de los *q-gramas* de P en T es:

$$h = \lfloor \frac{m-k-q+1}{k+s} \rfloor$$

Este h es la distancia que separa los $q - samples$ y además por definición de índice de $q - samples$ $h \geq q$. Pero la mayoría de las veces h es un parámetro que no es posible modificar en tiempo de búsqueda, por lo tanto las condiciones de ésta se deben realizar sobre el patrón y no sobre el índice. Por lo tanto la condición utilizada es:

$$s = \lfloor \frac{m-k-q+1}{h} - k \rfloor \text{ con } s > 0$$

Por otro lado, basandose en el Lema 3 en que A es el texto y B el patrón, se divide el patrón en zonas de búsqueda $Q_i = P[ih+1, ..., (i+1)h+k+q-1]; i \geq 0$ (Figura 3.13) y se aplica el siguiente algoritmo:

- Se buscan todos los *q-gramas* del bloque Q_i en el índice y sin error.
- Por cada posición p que reporte calce del $q-grama$ en ese bloque, se actualiza un contador $M\{p-ih\} = M\{p-ih\} + 1$.

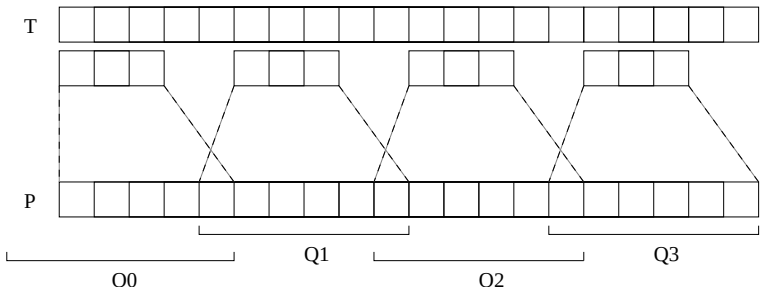


Figura 3.13: Bloques de búsqueda de algoritmos sobre q-samples

Luego, para todos los valores en que $M\{i\} \geq s$, se chequea el texto con la matriz de programación dinámica definida en la Figura 3.5.b desde $i - 2k + 2$ hasta $i + h + m + k$.

3.4 Intermedio

Los siguientes algoritmos dividen el problema de buscar un patrón P en el texto T con k errores a buscar j pedazos del patrón en T con $\lfloor k/j \rfloor$ errores, basándose en el Lema 1.

Los algoritmos que utilizan esta técnica son los algoritmos de partición en j pedazos[10, 8] y el algoritmo de *q-gramas aproximados*[12].

3.4.1 Partición en j pedazos

Ricardo Baeza-Yates y Gonzalo Navarro presntan en [10] un algoritmo basado en la combinación de un *Suffix Tree* o un *Suffix Array* con una partición del patrón:

- Se parte el patrón en *j* piezas, basándose en el Lema 1.
- Se busca cada una de ellas en el índice usando el backtracking simple[4].
- Para cada posición reportada, se chequea el calce usando la matriz de programación dinámica descrita en la Figura 3.5.b.

Myers plantea en [8] el concepto de **vecindad** de un patrón; esto es: dado un patrón P, la lista de todos las cadenas P' tal que la distancia de edición entre P y P' es menor que k representa la vecindad de P.

La matriz de programación dinámica descrita en la Figura 3.5.a es utilizada para construir, a partir del alfabeto del texto, todas las posibles cadenas de la vecindad. Una heurística es chequear cada vez que se le agregue una letra a la cadena si todavia pertenece a la vecindad; de no ser así se corta la construcción por esa rama y se realiza $backtracking$.

Luego, todas las cadenas de la vecindad son buscadas sin error en el índice y todas las posiciones reportadas son chequeadas utilizando la matriz de distancia de edición descrita en la Figura 3.5.b.

Se plantea en [8] la utilización de un índice de *q-gramas* para la búsqueda; por lo tanto sólo se pueden buscar patrones de tamaño a lo más $q - k$. Para patrones más largos, el patrón es dividido (para dar con el tamaño necesario) binariamente en 2^i partes iguales (con *i* el nivel de partición), cada parte se busca con $\lfloor k/2^i \rfloor$ (Lema 1) errores y en caso de ocurrencia se chequea el patrón en el texto utilizando verificación jerárquica.

La idea de la verificación jerárquica es dividir el patrón en j pedazos (Myers plantea en [8] que j debe ser potencia de dos), para buscar un patrón entonces se deberán buscar sus mitades, para buscar cada mitad sus cuartos y así sucesivamente. Por cada pedazo encontrado se verifica si el subpatrón de la jerarquía superior está presente en una vecindad del texto encontrado, de no ser así no se continúa la búsqueda. La figura Figura 3.14 muestra un ejemplo del chequeo usando verificación jerárquica. La aparición de la cadena `bbb` con $\lfloor k/4 \rfloor$ errores genera el chequeo de la cadena `aaabbb` en el texto con $\lfloor k/2^i \rfloor$ errores; la aparición de esa cadena genera el chequeo del patrón con k errores.

Aunque se plantea en [8] la utilización de un índice de q-gramas como estructura de indexación, el algoritmo también se puede portar a *Suffix Array* y *Suffix Tree* buscando la partición óptima del patrón.

aaabbbcccddd
aaabbb cccddd
aaa **bbb** ccc ccc

Figura 3.14: Partición jerárquica de un patrón en 4 piezas. Sólo son chequeados los pedazos en negrita, de abajo hacia arriba.

3.4.2 q-gramas aproximados

Este algoritmo utiliza, al igual que el algoritmo de *q-sampling*, un índice de *q-samples* para almacenar el texto y sus posiciones; y, al igual que el algoritmo presentado por Myers[8], genera una vecindad para el patrón de búsqueda.

Está basado en el Lema 4, donde A es el texto (los $A_i, i = 1, ..., j$ son los *q-samples*) y B es el patrón. Para garantizar que al menos j *q-samples* aparezcan en toda ocurrencia del patrón P se debe cumplir que:

$$h \leq \lfloor \frac{m-k-q+1}{j} \rfloor$$

El algoritmo descrito en [12] es el siguiente:

- Se divide el patrón en bloques Q_i (Figura 3.13) al igual que el algoritmo ST.
- Se crea un contador M_r, inicialmente $M_r = j(e+1)$ que corresponde al número de errores que posee el *q-sample* r ($r = pos/h$) y e un parámetro ajustable en tiempo de búsqueda con $q \geq e \geq \lfloor k/j \rfloor$. Notar que inicialmente cada *q-sample* posee tantos errores que descarta su calce.
- Se recorre todo el $trie$ para buscar los *q-samples* y se comparan con el bloque Q_i usando la matriz de programación dinamica definida en la Figura 3.5.b, simulando que el bloque Q_i es el texto y los caminos del $trie$ el patrón de búsqueda (por lo tanto se llena la matriz por filas y no por columnas).
- Al llegar a una hoja se chequea que el error sea a lo más e, en tal caso se reporta el *q-sample* y su error; es decir, se actualizan todos los valores de $M_r = M_r - (e+1) + bed(d_r, Q_i)$, donde $bed(d, Q)$ entrega la menor distancia de edición (alineamiento) que posee el *q-sample* d en el bloque Q y d_r es el *q-sample* que comienza en la posición $pos = rh$. Notar que al llegar a una hoja el número de posiciones a las cuales se les actualiza el contador M_r puede ser mayor que uno.

- Dado que todos los valores de una fila a otra no decrecen, si éstos son mayores que e, entonces se abandona la búsqueda por esa rama.

Finalmente, al igual que en el algoritmo de *q-sampling*[14], se debe chequear completamente el patrón en todas las áreas donde se encuentre un calce potencial. Para ellos se utiliza la matriz dinámica definida en la Figura 3.5.b. A diferencia de ST, en este caso las áreas "interesantes" son donde $M_r \leq k$.

Se debe notar que en [12] no se hace mayor referencia a los valores que deben tomar j y e; es parte del desarrollo de esta tesis encontrar los valores óptimos para ambos parámetros puesto que existe un *tradeoff* para ellos:

- Si se elije un valor de e pequeño, la búsqueda en las vecindades (dependientes de e) será menos costosa que para grandes valores; pero generará mayor cantidad de verificaciones al desestimar el real número de errores del patrón en el texto.
- Para una búsqueda dada, con e fijo, mientras mayor es el valor de j, mayor es el tamaño de las secuencias buscadas y por lo tanto menor el error por pedazo. Por lo tanto el costo de encontrar un *q-sample* relevante disminuye, pero las verificaciones en el texto aumentan.

Bibliografía

[1] R. BAEZA-YATES, G. NAVARRO, E. SUTINEN Y J.TARHIO. Indexing methods for approximate text retrieval. Informe técnico TR/DCC-97-2, Dept. of Computer Science, Univ. of Chile (1999). `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequidx.ps.gz`.

[2] A. COBBS. Fast approximate matching using suffix trees. En "Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)", páginas 41–54 (1995).

[3] GENBANK. Dna homosapiens. `http://www.ncbi.nlm.nih.gov`. Aproximadamente 50 MB de texto plano. (2000).

[4] G. GONNET, R. BAEZA-YATES Y T. SNIDER. "Information Retrieval: Data Structures and Algorithms", capítulo 3: New indices for text: Pat trees and Pat arrays, páginas 66–82. Prentice-Hall, Englewood Cliffs, NJ (1992).

[5] D. HARMAN. Overview of the Third Text REtrieval Conference. En "Proc. Third Text REtrieval Conference (TREC-3)", páginas 1–19 (1995). NIST Special Publication 500-207.

[6] STEFAN KURTZ. Reducing the space requeriment of suffix trees. Informe técnico 98-03, Technische Fakultät, Universität Bielefeld (diciembre 1998).

[7] CONRADO MARTÍNEZ. Tries. `http://www-assig.fib.upc.es/%7Eeda/`. Material Docente (2000).

[8] E. MYERS. A sublinear algorithm for approximate keyword searching. *Algorithmica* **Vol. 12**(Nº 4/5), 345–374 (1994). Preliminary version as Tech. Report TR-90-25, Dept. of Comp. Sci., Univ. of Arizona, 1990.

[9] G. NAVARRO Y R. BAEZA-YATES. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal* **Vol. 1**(Nº 2) (1998). `http://www.clei.cl`.

[10] G. NAVARRO Y R. BAEZA-YATES. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)* **Vol. 1**(Nº 1), 205–239 (2000). Special issue on Matching Patterns.

[11] G. NAVARRO, R. BAEZA-YATES, E. SUTINEN Y J. TARHIO. Indexing methods for approximate string matching. Manuscript.

[12] G. NAVARRO, E. SUTINEN, J. TANNINEN Y J. TARHIO. Indexing text with approximate q-grams. En "Proceedings of the 11st Annual Symposium on Combinatorial Pattern Matching (CPM'2000)", LNCS 1848, páginas 350–363 (2000).

[13] F. SHI. Fast approximate string matching with q-blocks sequences. En "Proc. 3rd South American Workshop on String Processing (WSP'96)", páginas 257–271. Carleton University Press (1996).

[14] E. SUTINEN Y J. TARHIO. Filtration with q-samples in approximate string matching. En "Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)", LNCS 1075, páginas 50–61 (1996).

[15] E. UKKONEN. Approximate string matching over suffix trees. En "Proc. 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93)", páginas 228–242 (1993).