UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

EL ALGORITMO DE BOYER-MOORE Y SU INFLUENCIA

TESIS
Para obtener el título de
Licenciado en Ciencias de la Computación
P R E S E N T A
Virginia Teodosio Procopio

Directora de tesis: M. en C. Elisa Viso Gurovich

2005

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

El algoritmo de Boyer-Moore y su influencia

Virginia Teodosio Procopio

Enero 2005

A dos personas muy importantes en mi vida,
a quienes debo gran parte de mi ser ...
A ti mamá
A ti papá

Los amo

¿Qué realidad es en verdad más intensa: la del presente,
absorbida al instante por nuestros sentidos y discernible,
o la memoria de lo que hemos experimentado previamente?
¿Es el presente de verdad más real que el pasado?
En el fondo, no me siento capaz de dar una respuesta.

M. C. Escher

Agradecimientos

Agradezco profundamente a todas las personas que, directa o indirectamente tuvieron que ver con este trabajo pues fueron el motor que me impulsó a concluir una tarea que ya se había extendido de más en el tiempo. Sin embargo no puedo omitir algunos casos especiales .....

Elisa, muchísimas gracias por todo, por compartir conmigo tu experiencia, tus anécdotas, tu forma de pensar – en varios aspectos de la vida –, y por supuesto por permitirme trabajar contigo, a tu lado y contagiarme siempre en tu lucha por el trabajo y sobre todo por la paciencia que siempre me tuviste y que no creo merecer.

A *los sinodales* que se dieron un tiempo, en sus múltiples actividades, para leer este trabajo.

A los profesores que me enseñaron mucho más de lo que se enseña en un aula de clases, *José Galaviz Casas*, *Elisa*, *Hortensia Galeana*, *Juan G.* – gracias por creer en mí – y *Ana Margarita Guzmán Gómez.*

A mi *Patricia*; eres la persona que me inspira a luchar por mis ideales, gracias por enseñarme a hacerlo.

Mane gracias por quererme tanto y por soportarme n veces más, sobre todo en los últimos tiempos.

A mi muchachito *Victor*, gracias por tu lucha por entenderme, pero sobre todo por permitirme crecer a tu lado.

A *Yadira* por los "viejos tiempos inolvidables".

A mi hermanito *Sergio* por la felicidad que me brinda el quererte tanto.

Victorino[1] gracias por ese espíritu de alegría, tal vez no sepas lo mucho que aprendo de tí cada día.

A *Galo* por enseñarme a buscar el lado bueno de las cosas, y por tu paciencia durante tanto tiempo en que tuve la fortuna de estar a tu lado, sobre todo en momentos tan importantes.

A mi querida *Lorena Eudave* que siempre tiene un momento para escuchar mis triunfos y mis penas; sin tí no hubiera podido salir de la peor crisis de mi vida.

Fernado y Carlos por que sé que me cobijarán en su seno en cualquier momento y también que algún día concluiremos tantos proyectos inconclusos.

A mis queridos *Gil*, *Tita* y *Crevel*, – grandes compañeros de batalla – que siempre escucharon mis quejas y me dieron ánimos todo el tiempo.

[1] YAMM

A mis amiguitas *Liz* y *Vero* por brindarme su apoyo incondicional y sus infinitos consejos.

A mis *queridos hermanos*. A todos mis sobrinos, en especial a *Andreota, Divia, Nalany y Omar* por tanta inocencia y tantas sonrisas y a *Abiga y Luis Felipe* por escuchar siempre mis comentarios. ¡Los quiero mucho!

A *"los Spice", Winono, Greñas, Fito* – y a Karla por hacerte tan feliz y porque juntas, tal vez, nos hagamos millonarias –, *Renecito y también Pedro, Luis Manuel, Gabelo, Steve* – desde tierras frías – y *Javier*, mis amigos con los que avancé por este sinuoso camino en Ciencias, por tantas aventuras, ¡y las que faltan!.

A la *Banda del Estacionamiento*; Toño "El Puas", Tania, Miguel, Memo, Pablo, Charly, "la Sombra", Julio, Daniel, Alberto, Gabo, Gise, Moni, Alejandro y el buen Richard, viejos amigos con los que he arreglado el mundo múltiples veces.

A *los cuates de DepVis*: Héctor y Mónica – aunque formalmente no eres de Depvis, pero como si lo fueras –, Juan, Hugo, Liz Heras, Alberto Barrios, Guillermo, Multti, Daniel y Rosmery, en especial a Carlos Puebla.

Al matemático *Roli*, aunque tienes una creencia de sólo haberme dejado el gusto por las matemáticas.

Gracias a todos por haberme permitido estar en su mundo por algún tiempo.

I really should thank you for it; after all, it was your life
that taught me the purpose of the whole life ....
Smith en "Matrix Revolutions".

Índice general

Introducción vii

Índice de figuras xi

1. Algoritmo de Boyer-Moore 1
- 1.1. Introducción . . . 1
- 1.2. Definiciones básicas . . . 1
- 1.3. Descripción del algoritmo de Boyer-Moore . . . 2
  - 1.3.1. Exploración de derecha a izquierda *(right-to-left scan)* . . . 4
  - 1.3.2. Regla del carácter erróneo *(Bad character rule)* . . . 4
  - 1.3.3. Regla extendida del carácter erróneo (*extended bad character rule*) . . . 6
  - 1.3.4. Regla (fuerte) del buen sufijo *((Strong) good suffix rule)* . . . 9
  - 1.3.5. El algoritmo Boyer-Moore . . . 20

2. Versión original del *ABM* 23
- 2.1. Introducción . . . 23
- 2.2. La Versión Original del ABM . . . 23
- 2.3. Regla δ_1 . . . 24
  - 2.3.1. Los valores δ_1 . . . 24
- 2.4. Regla δ_2 . . . 26
- 2.5. AOBM . . . 28

3. El algoritmo de Boyer-Moore y δ_2 29
- 3.1. Introducción . . . 29
- 3.2. La idea nueva: salto por carácter *(charJump)* . . . 29
- 3.3. La idea antigua: saltos por apareos *(match Jumps)* . . . 31

4. Árboles Sufijos 41
- 4.1. Introducción . . . 41
- 4.2. Definiciones básicas . . . 41
- 4.3. Uso de árboles sufijos para apareamiento exacto . . . 45
- 4.4. El algoritmo ingenuo que obtiene un árbol sufijo . . . 47
- 4.5. El algoritmo de Esko Ukkonen que construye árboles sufijos . . . 48

4.5.1. Actualizaciones en un árbol sufijo 49
4.5.2. Construcción del árbol sufijo 51
4.5.3. Ligas sufijas 52
4.5.4. Usando *ligas sufijas* para la construcción del árbol sufijo A 53

5. El *ABM* y árboles sufijos **57**
5.1. Introducción 57
5.2. Apareamiento de Patrones en Dominio-Comprimido 57
5.2.1. Transformación de Burrows-Wheeler 58
5.2.2. El ABM en texto de Dominio Comprimido con TBW 60
5.3. El algoritmo de árboles sufijos y su uso en TBW 62
5.4. La construcción del árbol sufijo 63
5.4.1. Podando el árbol sufijo 64

Comentarios Finales **65**

Bibliografía **67**

Índice de figuras

1.1. Proceso de búsqueda del patrón P en el texto T. 3
1.2. Proceso de exploración en el cual se ahorran comparaciones, al conocer información adicional sobre P. 4
1.3. Exploración de derecha a izquierda. 4
1.4. Regla del carácter erróneo. 5
1.5. El arreglo R. 6
1.6. *Regla del carácter erróneo* v.s. *regla extendida del carácter erróneo*. 7
1.7. Preproceso para la *regla extendida del carácter erróneo*. 8
1.8. *Regla (fuerte) del buen sufijo*. 10
1.9. *Regla del buen sufijo* v.s. *regla fuerte del buen sufijo*. 11
1.10. Cálculo de un valor de N. 12
1.11. Relación entre N y L', con respecto a P. 13
1.12. Relación entre N y L, con respecto a P. 13
1.13. N-caja. 14
1.14. Cálculo de un valor N sin ninguna operación extra. 15
1.15. El algoritmo N, paso 2.c). 16
1.16. El algoritmo N, paso 2.c) con $N(k') < |\beta|$. 16
1.17. El algoritmo N, paso 2.c) con $N(k') \geq |\beta|$. 16
1.18. Valores de l'. 19

2.1. Alineación en AOBM. 24
2.2. Movimiento en δ_1 $(apt + \delta_1(X))$. 25
2.3. Movimiento en δ_1 $(apt + q + 1)$. 26
2.4. Movimiento en δ_2. 27

3.1. *Salto por carácter*. 30
3.2. Uso de la *regla salto por carácter*. 30
3.3. Uso de *reubica* para mover a *apt* I. 32
3.4. Uso de *reubica* para mover a *apt* II. 32
3.5. Cálculo de los valores de *salto por Apareo*. 33
3.6. Representación de $sufijo[k] = r$. 34
3.7. Cálculo de los valores de *sufijo* I. 34
3.8. Cálculo de los valores de *sufijo* II. 35

4.1. Árbol sufijo correspondiente a la cadena $aabc$. La cadena a es la etiqueta del vértice v. La etiqueta del camino que termina en 2 es abc. La cadena aab etiqueta un camino que termina en la arista $(v, 1)$. 45
4.2. Árbol sufijo correspondiente a la cadena $abcabdabde$. 46
4.3. Las tres primeras iteraciones del algoritmo de Ukkonen para la construcción del árbol sufijo. 49
4.4. Árbol sufijo correspondiente a la cadena $abbcc$. 49
4.5. Ligas sufijas en un árbol sufijo. 52

5.1. Representación de M y M'. 63

Introducción

La búsqueda de patrones en texto, al que denotaremos como BPT por sus siglas, es un problema que consiste en determinar si un patrón dado aparece en un texto; si esto pasa entonces se debe conocer las posiciones del texto donde el patrón ocurre. Estas ocurrencias se refieren a *apareamiento exacto* (*exact matching*), es decir donde las cadenas coinciden carácter a carácter.

BPT es un problema muy importante e interesante pues existe una gran cantidad de aplicaciones, problemas e investigaciones que dependen de su solución para poder conseguir un determinado fin, obtener un resultado, sugerir ideas de construcción para nuevos algoritmos o simplemente como parte del proceso de investigación y/o desarrollo. Estos ambientes presentan características diversas y se intenta agruparlas para dar algoritmos que garanticen el mismo tiempo de ejecución, manteniendo una cierta independencia en el tratamiento de cada una de ellas. Por tal motivo, el estudio de BPT empezó desde hace mucho tiempo, pues así lo ha exigido el entorno del hombre en problemas que van desde editores de texto hasta investigación en ADN.

El primer algoritmo que se creó como una solución es el conocido como *algoritmo ingenuo*, que es correcto sin importar las características que un problema presente, pero dependiendo de ellas puede resultar computacionalmente ineficiente. Sin embargo, a lo largo de la historia han surgido una gran cantidad de algoritmos que resuelven el mismo problema y que son mejores, en lo que se refiere al tiempo y espacio usado. Los primeros algoritmos presentados formalmente con una complejidad menor son los de Knuth-Morris-Pratt[5] y Boyer-Moore[2]. La mayoría de los algoritmos para BPT los toman como base, los cuáles en teoría tienen el mismo orden de complejidad, pero en la práctica éste último resulta tener mejores resultados experimentales.

En este trabajo estudiaremos algoritmos que resuelven el problema BPT, desde el algoritmo de Boyer-Moore en sus inicios y algunas modificaciones hasta llegar con el algoritmo de Esko Ukkonen, el cuál es un ejemplo claro de la influencia que el primero tuvo sobre él; donde la principal ventaja es que mejora la complejidad del algoritmo original, aunque presenta cambios significativos como la estructura de datos usada, arreglos y árboles respectivamente; y no así la representación de los datos mismos, pues ambos crean *cadenas sufijas* y se basan en la misma idea para su construcción.

Existe en todo algoritmo partes elementales que no corresponden a la descripción propia de él pero que son indispensables para su implementación; éstas son partes vulnerables que podrían incrementar el desempeño general del algoritmo; generalmente se implementan dependiendo de cada problema particular, siempre y cuando garanticen los tiempos

que la teoría asume. En este trabajo se hace énfasis en éstas partes tan importantes como el algoritmo mismo.

Capítulo 1

Algoritmo de Boyer-Moore

1.1. Introducción

El Algoritmo de Boyer-Moore resuelve el problema de apareamiento exacto (*exact matching*) el cual consiste en encontrar todas las ocurrencias del patrón P de n símbolos, en el texto T que consiste de m símbolos o carácteres. A lo largo de este trabajo mantendremos constante el hecho de que $|P| = n$ y $|T| = m$.

Por ejemplo, si $P = aba$ y T está definido como sigue:

	1	2	3	4	5	6	7	8	9	10	11	12
T=	b	b	a	b	a	x	a	b	a	b	a	y

entonces P ocurre en T empezando en las posiciones 3, 7 y 9.

Este capítulo describe el algoritmo de Boyer-Moore, analizando el tiempo de ejecución, que es O(n+m) en el peor caso. Asimismo revisaremos el espacio requerido para almacenar los datos y el preproceso que se debe hacer al patrón a buscar, de tal manera que se cumpla con el tiempo de ejecución antes mencionado.

1.2. Definiciones básicas

- Un **símbolo o carácter** es un objeto indivisible.[1]
- Un **alfabeto** es un conjunto finito de símbolos. En general, se usa la letra griega Σ para denotarlo.
- Una **cadena** S es una sucesión finita de símbolos; cada uno pertenece a un *alfabeto finito* y se encuentran ordenados de izquierda a derecha. $|S|$ denota el número de símbolos en S.
- Para cualquier cadena S, $S(i)$ denota el $i-$ésimo carácter de S. Decimos que S es **una cadena sobre** Σ, si cada $S(i) \in \Sigma$.

[1] Frecuentemente, para fines de manejo de cadenas, el término más común a emplear es carácter, por lo que trataremos de apegarnos a esta nomenclatura.

Ejemplos:
Sea $\Sigma = \{0, 1\}$ y $S = 0011001$. S es una cadena sobre Σ, donde $|S| = 7$.
Sea $\Sigma = \{a, b, c, d\}$ y $S = acb$. S es una cadena sobre Σ, donde $|S| = 3$.

- Para cualquier cadena S, $S[i \dots j]$ es una **subcadena** de S que empieza en la posición i y termina en j. Si $i = 1$, entonces $S[i \dots j] = S[1 \dots j]$ es un **prefijo** de S. Si $j = |S|$, entonces $S[i \dots j] = S[i \dots |S|]$ es un **sufijo** de S.

Ejemplo:
Universidad es una cadena, *ersid* es una subcadena, *Univ* es un prefijo e *idad* es un sufijo.

- Un **prefijo propio, sufijo propio o subcadena propia** de S es, respectivamente un prefijo, sufijo o subcadena de S, que no es toda la cadena S.

- Supongamos que a y b son variables que contienen cada una de ellas a un carácter. Se dice que a y b **se aparean (*match*)**, cuando al compararlos resulta que $a = b$; de lo contrario se dice que **no se aparean (*mismatch*)**.

1.3. Descripción del algoritmo de Boyer-Moore

Antes de describir el algoritmo de Boyer-Moore, que por economía denotaremos como ABM, consideraremos una forma básica de resolver el problema de apareamiento exacto.

La forma más fácil de obtener la solución a este problema, es alinear el lado izquierdo de T con el lado izquierdo de P y comparar carácter con carácter hasta llegar al último de P, el del lado derecho, o encontrar que dos carácteres alineados resultan ser distintos. Si ocurre que los carácteres comparados no se aparean, mover a P una posición a la derecha respecto a T. Si se ha comparado completamente a P, quiere decir que existe una presencia de P en T, por lo que se reporta la posición izquierda donde se alinean el principio de P con T, se mueve a P un lugar a la derecha y se inicia la comparación de nuevo; este proceso se muestra en la Figura 1.1.

En esta figura se observa, en la primera alineación, que la primera comparación que se realiza es $T(1)$ con $P(1)$; al ocurrir un no-apareo inmediato, se prosigue a mover a P una posición, resultando la segunda alineación, y la comparación de $T(2)$ con $P(1)$, $T(3)$ con $P(2)$, y así sucesivamente obteniendo apareos, y por lo tanto continuando la exploración, hasta que la comparación de $T(9)$ y $P(8)$ conduce a un no-apareo. Entonces se recorre P una posición (tercera alineación), se recomienza la exploración y al ocurrir un no-apareo, se repiten estos últimos pasos dos veces más (cuarta y quinta alineación), hasta que en el próximo movimiento de P (la sexta alineación), ocurren ocho apareos consecutivos, que es la longitud de P, lo que conduce a saber que P aparece en T empezando en la posición 6. Con este ejemplo, este algoritmo realiza 20 comparaciones[2] para poder llegar a encontrar la presencia de P en T.

[2]Basta contar el número total de $\surd$'s y de $\star$'s.

		1	2	3	4	5	6	7	8	9	10	11	12	13
	T:	x	a	b	x	y	a	b	x	y	a	b	x	z
1ra alineación	P:	a	b	x	y	a	b	x	z					
		$\star$												
2da alineación			a	b	x	y	a	b	x	z				
			$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\star$				
3ra alineación				a	b	x	y	a	b	x	z			
				$\star$										
4ta alineación					a	b	x	y	a	b	x	z		
					$\star$									
5ta alineación						a	b	x	y	a	b	x	z	
						$\star$								
6ta alineación							a	b	x	y	a	b	x	z
							$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Figura 1.1: Proceso de búsqueda del patrón P en el texto T. Donde $\surd$ indica un apareo y $\star$ indica un no-apareo.

A este algoritmo que hemos descrito, se le conoce como *algoritmo ingenuo* (*naive algorithm*), al que denotaremos AI, y que hace honor a su nombre, pues es muy fácil de entender y de programar, sin embargo, el tiempo de ejecución es de nm para cualquier caso, lo cual para fines prácticos resulta muy costoso. La razón de ello es que AI empieza cada ciclo de exploración como si fuese el primero, sin analizar la información que va conociendo de P y de T, pues si lo hiciera, podría realizar movimientos del patrón por más de un carácter, cuando ocurriese un no-apareo, cuidando al mismo tiempo de que este movimiento sea lo suficientemente largo para evitar comparaciones posteriores inútiles y suficientemente corto para no perder una presencia de P en T. Esta acción ayudaría a eliminar algunas de las comparaciones que conducen a un no-apareo, moviendo a P a través de T más rápidamente.

Para mostrar que se podría tener un mejor tiempo de proceso que el que se obtiene con AI, consideremos las cadenas de T y P usadas en la Figura 1.1, en donde después de la novena comparación, $T(9)$ con P(8), se podrían ahorrar las próximas tres comparaciones moviendo a P para alinearlo con $T(6)$, resultando en lo que corresponde en la figura con la tercera alineación. Esto se hace precisamente aprovechando el conocimiento que se adquiere sobre las cadenas al realizar las primeras comparaciones, pues después de esto, se sabe que $P(1) = a$ ocurre en la posición 5; mientras que en T ocurre primero en la posición 2 y luego en la posición 6. Por lo tanto se puede concluir que no existirán apareos hasta que P sea alineado, al menos, con $T(6)$, logrando con ello el ahorro de comparaciones que sabemos de antemano conducirán a no-apareos. Este proceso se muestra en la Figura 1.2

El ABM, para lograr solucionar el problema de apareamiento exacto, se basa en la idea anterior con algunas modificaciones que consisten en el uso de tres reglas, que explicaremos a la brevedad: "exploración de derecha a izquierda"(*right to left scan*), "movimiento por un carácter erróneo"(*character shift rule*) y "movimiento por un buen sufijo" (*good suffix*

		1	2	3	4	5	6	7	8	9	10	11	12	13
	T:	x	a	b	x	y	a	b	x	y	a	b	x	z
1ra alineación	P:	a	b	x	y	a	b	x	z					
		$\star$												
2da alineación			a	b	x	y	a	b	x	z				
			$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\star$				
3ra alineación							a	b	x	y	a	b	x	z
							$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Figura 1.2: Proceso de exploración en el cual se ahorran comparaciones, al conocer información adicional sobre P.

shift rule). Estas tres reglas se aplican a P en una etapa de preprocesamiento, con las cuales se logra obtener una complejidad de $O(n+m)$ en el peor de los casos. Además, si también se usan sobre P unas extensiones de dichas reglas, que se explicarán más adelante, se logra obtener un tiempo de $O(m)$.

1.3.1. Exploración de derecha a izquierda *(right-to-left scan)*

Esta regla alinea el patrón P con el texto T, de la misma forma en que lo hacía el AI, pero contrario a éste, la comparación de los carácteres la realiza de derecha a izquierda.

Un ejemplo se muestra en la Figura 1.3, donde el patrón está alineado empezando en $T(3)$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T:	x	p	b	c	t	b	x	a	b	p	q	x	c	t	b	p	q
P:			t	p	a	b	x	a	b								
					$\star$	$\surd$	$\surd$	$\surd$	$\surd$								

Figura 1.3: Exploración de derecha a izquierda. Dadas de esta forma la alineación de P y T, la primera comparación fallida sucede con $T(5)$ y $P(3)$.

Con esta alineación, la primera comparación que el ABM realiza es de $T(9)$ con $P(7)$, es decir, el último carácter de P con el respectivo carácter de T, y de ahí sigue comparando hacia la izquierda, hasta que encuentra que $T(5)$ y $P(3)$ no se aparean; en ese momento se recorre a P hacia la derecha[3], la cual en el ABM se determina por alguna de las otras dos reglas antes mencionadas, y se iniciarían de nuevo las comparaciones.

1.3.2. Regla del carácter erróneo *(Bad character rule)*

Para darnos una idea de esta regla supongamos la siguiente situación: dada una alineación de P con T, si sucede un no-apareo de $P(n) = c$ y $T(j) = x$, es decir en la comparación

[3]En la figura no se muestra este corrimiento, pues aún no hemos explicado como se realiza en *ABM*.

inicial, y si conocemos la posición k más a la derecha de P, tal que $P(k) = T(j) = x$, entonces podemos reubicar a P hacia la derecha de tal forma que $P(k)$ quede abajo de $T(j)$, y recomenzar la exploración, con la garantía de tener mayores posibilidades de encontrar una ocurrencia de P en T (ver Figura 1.4(a)). Si x no ocurre en P, entonces este patrón se reubica pasando la posición j en T (ver Figura 1.4(b)).

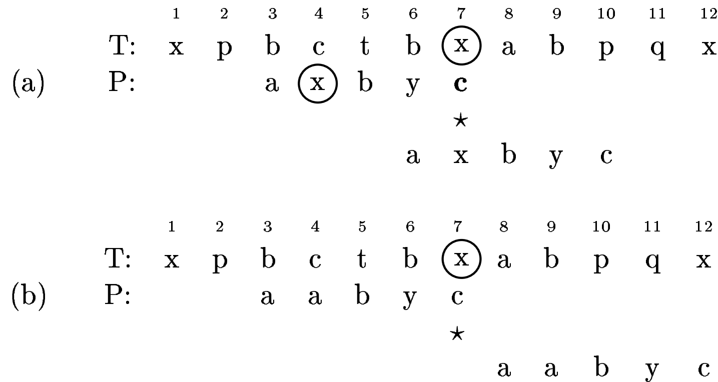


Figura 1.4: Regla del carácter erróneo. **(a)** Al comparar a $T(7)$ con $P(n), n = 5$, ocurre un no apareo, por lo que se busca la posición k más a la derecha de P en donde $P(k) = T(7)$; ésta es $P(2)$, por lo que P es alineada de tal forma que $P(2)$ coincida con $T(7)$. **(b)** En este caso al ocurrir el no apareo de $T(7)$ con $P(5)$, y como $T(7)$ no figura en P, el patrón se reubica pasando la posición 7 de T.

Procedemos a formalizar esta idea con la siguiente definición:

Definición 1.3.1. *Para cada x en el alfabeto Σ, sea $R(x)$ la posición más a la derecha donde x ocurre en P. $R(x) = 0$ si x no ocurre en P.*

Algoritmo para calcular los valores del arreglo R

El algoritmo que calcula los valores $R(x)$, es:

Consideremos un arreglo[4] R del tamaño del alfabeto Σ; al inicio asignamos ceros a todas las posiciones, empezamos a recorrer a P de izquierda a derecha y asignamos la posición i−ésima, en la que estamos situados, al carácter $P(i)$ en R. De esta forma se garantiza que al terminar de recorrer a P, R guarda las posiciones más a la derecha, en donde apareció cada carácter de P y 0 para los carácteres que pertenecen a Σ y no a P.

Un ejemplo de este proceso se muestra en la Figura 1.5, en donde observamos que cuando un carácter x aparece más de una vez, sólo perdura en R la última asignación, es decir la posición más a la derecha en donde x ocurre en P. Tal es el caso de d que obtiene en primer lugar la posición 2, cambiando luego a 4 y por último a 6.

[4] El tipo de almacenamiento depende del alfabeto con el que se esté trabajando, cuidando que esta elección garantice la eficiencia de acceso al valor R, dado un carácter x.

$$\Sigma = \{ \overset{1}{a} \ \overset{2}{b} \ \overset{3}{c} \ \overset{4}{d} \ \overset{5}{e} \} \qquad P = \overset{1}{c} \ \overset{2}{d} \ \overset{3}{c} \ \overset{4}{d} \ \overset{5}{a} \ \overset{6}{d}$$

		a	**b**	**c**	**d**	**e**
pos. 1:	$R = [$	0,	0,	1,	0,	0]
pos. 2:	$R = [$	0,	0,	1,	2,	0]
pos. 3:	$R = [$	0,	0,	3,	2,	0]
pos. 4:	$R = [$	0,	0,	3,	4,	0]
pos. 5:	$R = [$	5,	0,	3,	4,	0]
pos. 6:	$R = [$	5,	0,	3,	6,	0]

Figura 1.5: El arreglo R. Éstos valores corresponden a los carácteres posicionados en el mismo lugar como en Σ. Se ilustran los valores que este arreglo contiene en cada paso del algoritmo.

Es claro que este algoritmo se ejecuta en tiempo lineal con respecto al patrón y que tenemos que reservar memoria suficiente para almacenar la posición de cada uno de los carácteres del alfabeto[5], o sea, $|\Sigma|$.

Una vez que tenemos calculados los valores $R(x)$, los usamos en el algoritmo que corresponde a la *regla del carácter erróneo* (*bad character shift rule*) para determinar el número de posiciones a la derecha que P se recorrerá al ocurrir un no-apareo, el cual trata en lo posible que éste sea mayor que uno.

Algoritmo de la regla del carácter erróneo

Supongamos que, como en la Figura 1.3, los carácteres de $P[(n-i)\dots n]$ se aparean con los respectivos carácteres de T en una alineación de ambas cadenas, pero ocurre un no-apareo de $P(i)$ con $T(k)$. La *regla del carácter erróneo* dice que P será reubicado por $max\{1, i - R(T(k))\}$ posiciones a la derecha. Es decir, si la ocurrencia más a la derecha en P del carácter $T(k)$ es en la posición j, tal que $j < i$, entonces reubicamos a P para que el carácter j de P esté abajo del carácter k de T. En cualquier otro caso, movemos a P una posición y recomenzamos el proceso de exploración para la nueva alineación.

1.3.3. Regla extendida del carácter erróneo (*extended bad character rule*)

Existen ocasiones en que para ciertos patrones, la regla del *carácter erróneo* no encuentra la posición deseada de reubicación del patrón, si es que el carácter $T(j)$, que ocasionó el no-apareo, se encuentra a la derecha del carácter correspondiente a P para esa alineación. En la Figura 1.6(a) se observa que con la *regla del carácter erróneo*, al ocurrir el no-apareo

[5]En ciertos casos se recomienda la representación binaria del carácter para ubicarlo dentro del arreglo, haciendo las conversiones necesarias o el uso de hash; esta elección depende del alfabeto que se esté trabajando.

de $T(6)$ con $P(5)$ y encontrar que 8 es la posición más a la derecha de P donde ocurre $T(6)$, esta regla indicaría el corrimiento de P por una posición, pues $i - R(T(k))$ resulta ser una posición inválida para la cadena; pero como $T(6)$ también ocurre en $P(1)$, lo mejor sería reubicar a P para que coincida $P(1)$ con $T(6)$, como se observa en la Figura 1.6(b). Esta regla implementa esta idea para obtener beneficios en eficiencia.

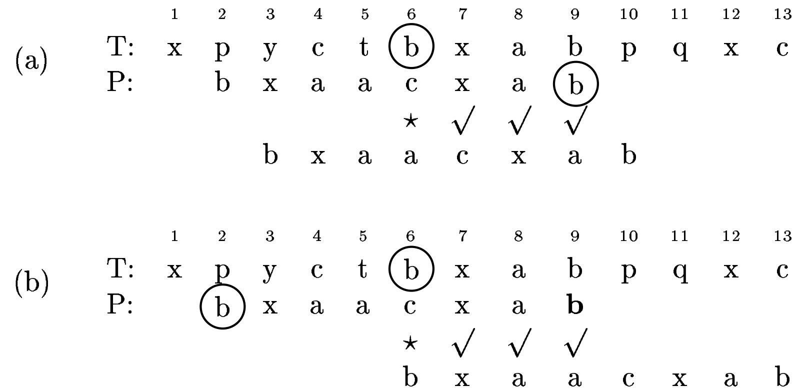


Figura 1.6: *Regla del carácter erróneo* v.s. *regla extendida del carácter erróneo.* **(a)** Con la primera regla P se reubica por una posición a la derecha. **(b)** Con la segunda, P se reubica por 4 posiciones a la derecha.

Esta situación es común cuando el texto contiene muchas repeticiones de carácteres, situación típica del ADN, con alfabeto de tamaño cuatro; en tales casos el uso de la *regla extendida del carácter erróneo* debería trabajar de forma más robusta, y de esta forma reemplazaría a la *regla del carácter erróneo.*

En resumen ésta regla consiste en que cuando un no-apareo ocurre en la posición i de P y el carácter que lo ocasionó en T es x, entonces P se reubica a la derecha hasta que la x más cercana a la izquierda de i en P, esté bajo la x de T.

Si esa x no existe, entonces P se reubica hacia la derecha pasando la posición i.

Preprocesamiento de la *regla extendida del carácter erróneo*

Para la implementación de esta regla, necesitamos preprocesar a P, y de esta forma se garantiza mejorar la eficiencia en tiempo y espacio.

Este preproceso incluye el cálculo de la posición de la ocurrencia más cercana de x en P, a la izquierda de i, para cada posición i en P y para cada carácter x en Σ. La forma más simple de obtener esta información es tener un arreglo de $n \times |\Sigma|$ para guardar estos datos, y entonces cuando ocurre un no-apareo con x de T en la posición i de P, basta obtener la entrada (i, x) de este arreglo. Evidentemente la consulta de dicha tabla, dados i y x, es rápida en el acceso pero muy ineficiente respecto a espacio, además de que el tiempo para construir el arreglo puede ser excesivo. La forma más sencilla consiste en recorrer el

patrón de izquierda a derecha y asignar a cada $(i, P(i-1))$, con i fija, la posición $i-1$ y para el resto de (i, y), con $x \neq y$, se les asigna el valor respectivo $(i-1, y)$, obteniendo con esto $n \times |\Sigma|$ en tiempo de ejecución, $2 \leq i \leq n$; cuando $i = 1$, $(i, x) = 0 \ \forall$ x $\in \Sigma$.

Un ejemplo del proceso de este algoritmo se muestra en la Figura 1.7.

$\Sigma = \{$ a b c d $\}$ P:= a b c c a d (posiciones 1 2 3 4 5 6)

i=1

	a	b	c	d
1	0	0	0	0
2				
3				
4				
5				
6				

i=2

	a	b	c	d
1	0	0	0	0
2	1	0	0	0
3				
4				
5				
6				

i=3

	a	b	c	d
1	0	0	0	0
2	1	0	0	0
3	1	2	0	0
4				
5				
6				

Figura 1.7: Preproceso para la *regla extendida del carácter erróneo*. En este esquema se muestra el arreglo bidimensional, resultado de realizar las tres primeras iteraciones del algoritmo más simple para preprocesar a P.

Para mejorar la eficiencia en espacio respecto a la descrita anteriormente, se procede a recorrer a P de derecha a izquierda, recolectando las posiciones donde x ocurre en P, para todo $x \in \Sigma$. Como la exploración es de derecha a izquierda, obtendremos listas de valores en orden descendente. Por ejemplo, si P está definido como sigue:

	1	2	3	4	5	6	7	8	9
P:=	y	x	x	z	w	x	z	x	y

tenemos que la lista asociada al carácter x es $[8, 6, 3, 2]$, para y es $[9, 1]$, para z es $[7, 4]$ y para w es $[5]$.

La forma de usar esta información consiste en que cuando ocurre un no-apareo en la posición i de P con el carácter x de T, se explora la lista asignada a x hasta encontrar el primer número menor que i, obteniendo la posición deseada de x; si no existe este número menor, quiere decir que no ocurre a la izquierda de i, por lo que P se reubica pasando la posición x de T, que implica mover a P por i posiciones.

Implementando esta última idea se mejora la eficiencia de calcular las listas, donde el tiempo de explorar cada una es de a lo más $n - i$, con lo cual se puede afectar el tiempo de ejecución del ABM, llevándolo al doble. Sin embargo este aumento puede reducirse a menos del doble, dependiendo del problema que se esté resolviendo y también realizando alguna búsqueda binaria sobre las listas para reducir el tiempo de ejecución.

Una garantía de la regla extendida, es que obtenemos reubicaciones de P por cantidades más grandes, pero en general se elije la regla simple para evitar incrementar la complejidad en el momento del apareamiento a costa de espacio.

1.3.4. Regla (fuerte) del buen sufijo *((Strong) good suffix rule)*

Esta regla surge para resolver los problemas cuando existen muchas repeticiones de carácteres en el texto, como los que se mencionaron en la sección anterior.

El método de preprocesamiento original para esta regla es considerado muy difícil y poco entendible [4], pero con algunas ideas, que explicaremos más adelante, este proceso se entiende de mejor manera.

La regla fuerte del buen sufijo consiste en:

Dada una alineación de P y T, supongamos que una subcadena t de T se aparea con un sufijo de P ocurriendo un no-apareo en la siguiente comparación a la izquierda, como se muestra en la Figura 1.8(a). Se debe encontrar, si existe, la copia más a la derecha t' de t en P, tal que t' no es t y el carácter a la izquierda de t' difiere del carácter a la izquierda de t – ver Figura 1.8(b). Si esta subcadena existe, recorre a P hacia la derecha para que t' quede bajo la subcadena t de T – Figura 1.8(c). Si t' no existe, entonces recorre a P pasando el lado izquierdo de t en T por la menor cantidad para que un prefijo del patrón aparee a un sufijo de t en T – Figura 1.8(d). Si tal corrimiento no es posible, entonces recorre a P por n posiciones hacia la derecha, es decir pasando a t en T –Figura 1.8(e). Si no se encuentra un no-apareo y se termina de revisar P, entonces se ha encontrado una ocurrencia de P; se debe mover a P por la menor cantidad para que un prefijo propio del patrón recorrido aparee un sufijo de la ocurrencia de P en T, como se vió en la Figura 1.8(d), sólo que ahora es del patrón completo encontrado en T. Si tal corrimiento no es posible, entonces recorre a P por n lugares, es decir, mover a P pasando t en T.

Teorema 1.3.1. *El uso de la* regla del buen sufijo *nunca mueve a P brincando una presencia de P en T.*

Demostración. Supongamos que el lado derecho de P se encuentra alineado con el carácter k de T, antes de realizar algún movimiento, y supongamos que la *regla del buen sufijo* mueve a P, de tal forma que el último carácter, $P(n)$, queda alineado con k', donde $k' > k$. Cualquier ocurrencia de P que termine en una posición j estrictamente entre k y k' violaría la regla de selección de k', ya que esto implicaría que una copia más cercana de t ocurre en P o que existe un prefijo más grande de P que aparea a un sufijo de t. Por lo tanto esta ocurrencia de P no existe y la regla nunca mueve a P brincando una presencia de P en T. ❏

El algoritmo original de Boyer-Moore usa una regla más débil que la *regla fuerte del buen sufijo*[4], la cual difiere de ésta en que no se pide que el carácter a la izquierda de t' sea distinto del carácter a la izquierda de t.

La desventaja de esta "regla débil" es que las reubicaciones de P se realizan brincando menos posiciones, ocasionando con esto comparaciones vanas –como se muestra en la Figura 1.9– razón por la cual el ABM usa la *regla fuerte del buen sufijo* para garantizar el tiempo de ejecución lineal.

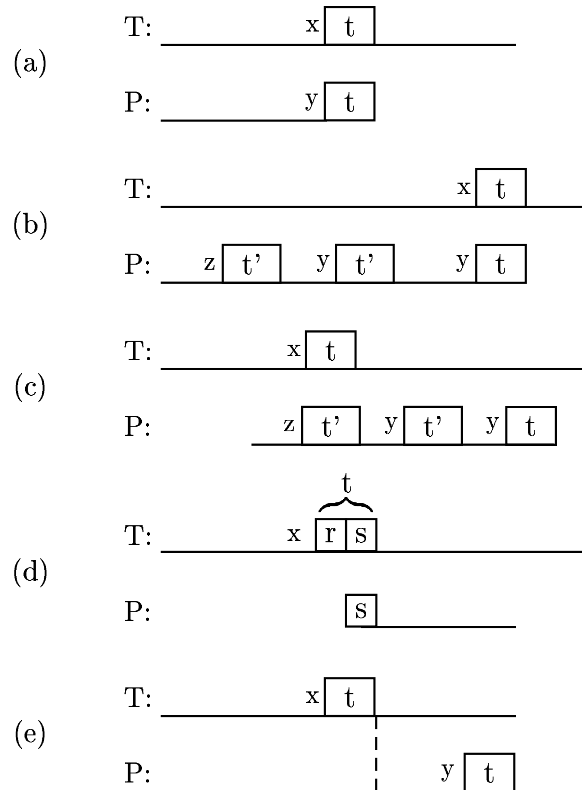


Figura 1.8: *Regla (fuerte) del buen sufijo.* **(a)** Un sufijo de P encuentra un apareo con una subcadena de T, pero en la próxima comparación ocurre un no-apareo. **(b)** La copia t' de t que nos sirve es la que se encuentra más a la izquierda, pues cumple que $z \neq y$. **(c)** Una vez encontrada la copia de t deseada, se reubica a P de tal forma que t' quede abajo de t. **(d)** En este caso se encontró un apareo de un sufijo de t en T con un prefijo de P, aquí $t = rs$. **(e)** El patrón se reubica pasando t en T.

Preprocesamiento para *la regla del buen sufijo*

El preprocesamiento que explicaremos a continuación se aplica en el ABM. Para ello definiremos lo siguiente:

Definición 1.3.2. *Para cada i, con $i > 1$, definimos* $\mathbf{L}(\mathbf{i})$ *como la posición más grande, menor que n, tal que la cadena $P[i \dots n]$ aparea un sufijo de $P[1 \dots L(i)]$. $L(i)$ es 0 si no existe una posición que satisfaga estas condiciones.*

Definición 1.3.3. *Para cada i, sea* $\mathbf{L}'(\mathbf{i})$ *la posición más grande, menor que n, tal que la cadena $P[i \dots n]$ aparea a un sufijo de $P[1 \dots L'(i)]$ y tal que el carácter que precede a*

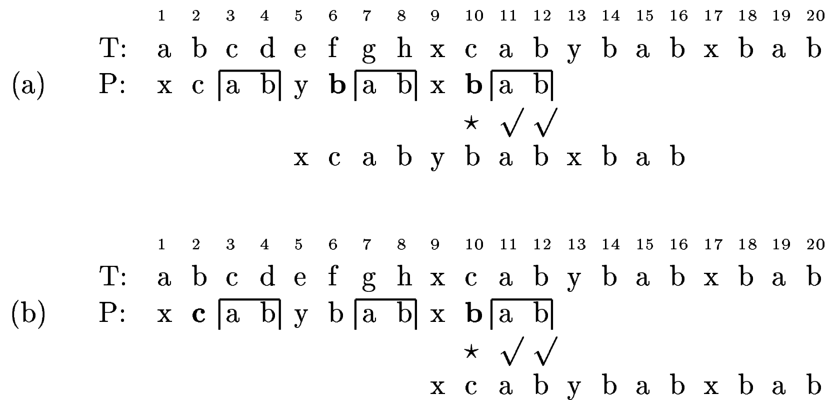
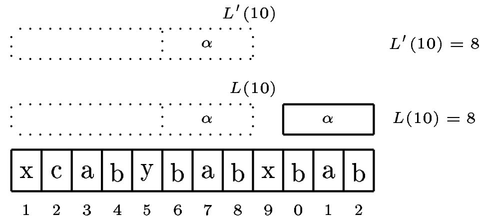


Figura 1.9: *Regla del buen sufijo* v.s. *regla fuerte del buen sufijo.* Al ocurrir el no-apareo de $T(10)$ con $P(10)$ **(a)** Con la primera P se reubica para que $P[7 \ldots 8]$ esté bajo $T[11 \ldots 12]$, **(b)** Con la segunda P se reubica de tal forma que $P[3 \ldots 4]$ quede abajo de $T[11 \ldots 12]$.

dicho sufijo no es igual a $P(i-1)$. $L'(i)$ es 0 si no existe una posición que satisfaga estas condiciones.

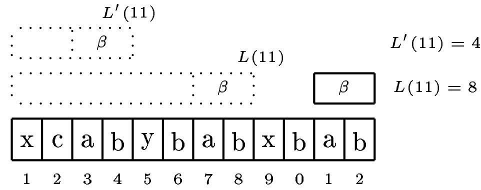
En resumen, $L(i)$ es la posición final de la copia de $P[i \ldots n]$ que no es sufijo de P, y de igual manera para $L'(i)$, sólo que en este último se exige que el carácter que precede a dicha copia sea distinto de $P(i-1)$.

Por ejemplo, si el patrón está definido de la siguiente manera, $P = xcabybabxbab$ donde $n = 12$, entonces para $i = 10$, sea α la subcadena $P[i \ldots 12] = bab$; la posición mayor (distinta de 12) donde α aparece es 8, como se muestra a continuación.



En el diagrama anterior, también se muestra el valor para $L'(10)$, donde en este caso coincide con $L(10)$, ya que $P(9) \neq P(5)$.

Si $i = 11$, sea β la subcadena $P[i \ldots 12] = ab$, la posición final donde β vuelve aparecer, sin ser sufijo de P, es 8, por lo que tenemos que $L(11) = 8$, pero aquí $P(10) = P(6)$, por lo que para asignar valor a $L'(11)$ se vuelve a buscar a β hacia la izquierda; en este caso ocurre terminando en $P(4)$, y como $P(10) \neq P(2)$, entonces $L'(11) = 4$, como se muestra en el siguiente esquema.



Estos valores son de gran ayuda, según la regla que se desee usar, pues la *regla débil del buen sufijo* o la *regla fuerte del buen sufijo*, implicaría el uso de $L(i)$ y $L'(i)$ respectivamente, además de que $L'(i)$, para $1 < i \leq n$, se puede calcular en la etapa de preprocesamiento del ABM (y si se desea, también $L(i)$) en tiempo $O(n)$ usando las definiciones y teoremas que presentaremos a continuación.

Así para la versión del ABM que ocupa a la *regla fuerte del buen sufijo*, si el carácter $i-1$ de P ocasionó un no-apareo y $L'(i) > 0$, entonces P se mueve a la derecha por $n - L'(i)$ posiciones, logrando con esto movimientos de P como los que se muestran en la Figura 1.9(b).

Pasamos ahora a calcular los valores de L y L' justificando la complejidad de $O(n)$ que esto conlleva.

Definición 1.3.4. *Sea P una cadena; definimos $N(j)$ como la longitud del sufijo más grande de la subcadena $P[1 \ldots j]$ que también es sufijo de P.*

Por ejemplo, si $P = cabdabxabdab$, entonces $N(3) = 2$ y $N(6) = 5$. Siguiendo esta definición mostramos en la Figura 1.10 las cadenas que se aparean para $j = 3$ y 6.

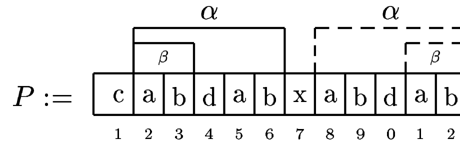


Figura 1.10: Cálculo de un valor de N. Dado P, $N(6)$ es la longitud de α; el sufijo más grande de la subcadena $P[1 \ldots 6]$ que también se aparea con un sufijo de P. También se muestra $N(3)$, con β representando el sufijo correspondiente.

Teorema 1.3.2. *$L(i)$ es el índice j más grande, menor que n, tal que $N(j) \geq |P[i \ldots n]| = n-i+1$. $L'(i)$ es el índice j más grande, menor que n, tal que $N(j) = |P[i \ldots n]| = n-i+1$.*

Demostración. Cada valor de $N(j)$ corresponde al sufijo más grande que puede obtenerse del prefijo que termina en j y que también es sufijo de P.

Al tener un sufijo de longitud $N(j) = |P[i \ldots n]|$, sabemos que el valor N asignado a j corresponde a la longitud más grande, menor que n (pues perdería sentido si no fuera elegido de esta manera), de una subcadena que es $P[(j - N(j) + 1) \ldots j]$, que termina en j y que aparea a un sufijo de P, que es $P[(n - N(j) + 1) \ldots n]$, justo la definición de $L'(i)$, para $i = n - N(j) + 1$ (ver Figura 1.11).

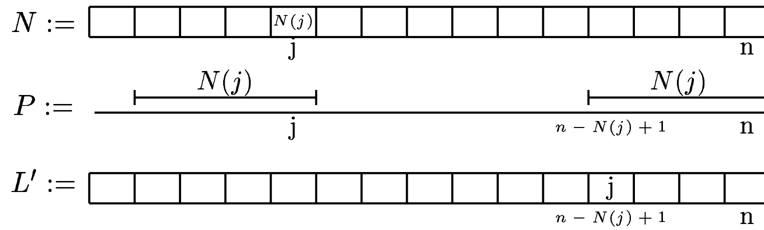


Figura 1.11: Relación entre N y L', con respecto a P.

En el caso que $N(j) > |P[i \dots n]|$, quiere decir que se encontró en la posición j, un sufijo α de longitud $N(j)$, que corresponde al valor para $L(n - N(j) + 1)$, y los sufijos que se van formando al avanzar hacia la derecha también lo son para α, por lo que el valor de $L(i)$ para $n - N(j) + 1 \leq i \leq n$ queda fijo a j (ver Figura 1.12). ❏

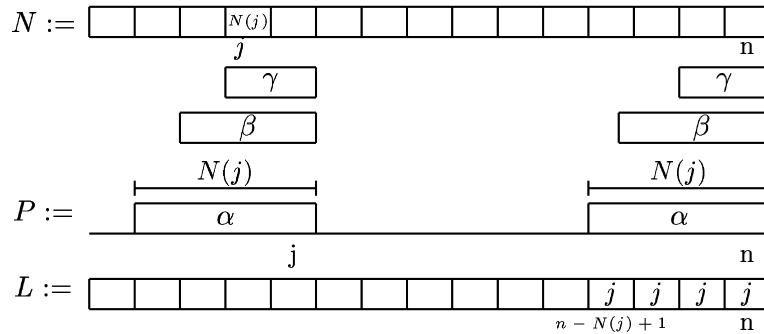


Figura 1.12: Relación entre N y L, con respecto a P.

La relación que existe entre $L(i)$ y $N(i)$ es que los primeros se pueden obtener a partir de los segundos, considerando las relaciones dadas en el teorema anterior.

Antes de continuar con el algoritmo que calcula los valores de $L(i)$, a partir de los valores de $N(j)$, mostraremos la forma de obtener a los segundos en tiempo lineal con respecto a la longitud de P.

El arreglo N

Como veíamos anteriormente, al calcular los valores N obtenemos *cajas* que definen a las subcadenas que se aparean con un sufijo dado, como se muestra en la Figura 1.13. Si $N(i) > 0$, cada *caja* termina en alguna posición $i < |P|$, en donde su longitud es $N(i)$. De esta forma cada una de ellas representa una subcadena[6] de longitud máxima que aparea a un sufijo de P. De esto surgen las siguientes definiciones:

[6]La cual no termina en $|P|$.

Definición 1.3.5. *Para cada $i < n$, tal que $N(i) > 0$, la* N-caja *en i es la subcadena que termina en i y empieza en $i - N(i) + 1$.*

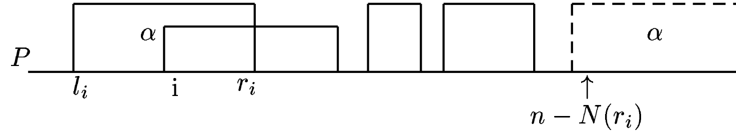


Figura 1.13: N-caja. Cada *caja* representa una subcadena que se aparea con un sufijo; cada una termina entre las posiciones i y $n-1$; l_i representa el inicio de la primer caja, la cuál termina en i ó a la derecha de esa posición, α denota esa *N(i)-caja*. También se muestra la copia de α como sufijo de P.

Definición 1.3.6. *Para cada $i < n$, $l_i = min\{j - N(j) + 1 \mid i \leq j < n, N(j) > 0\}$ y $r_i = j$.*

Es decir, l_i es la posición más a la izquierda de todas las *N-cajas* que terminan en, o después de i, y a su vez es la posición inicial, a la derecha, de la *N-caja* que termina en r_i; l_i puede ser elegido para ser el lado izquierdo de varias de estas *N-cajas*. Por ejemplo, si el patrón, está definido como sigue:

P=	y	c	b	a	a	b	a	a	x	a	c	b	a	a	b	a	a
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Entonces para $j = 15$, $N(8) = 7$, $l_{15} = 2$ y $r_{15} = 8$.

Como mencionamos anteriormente, este algoritmo puede ser implementado en tiempo lineal, respecto a P, aprovechando algunas ideas que explicaremos a continuación.

El patrón se recorre de derecha a izquierda, empezando por $i = n - 1$, en donde para cada iteración i, el algoritmo sólo necesita los valores de r y l anteriores inmediatos, $j = i - 1$, por lo que basta con el uso de dos variables que almacenen a dichos valores y que se actualicen en la iteración sobre i; si se desea pueden ser almacenados.

Los valores de N

- El primer paso del algoritmo es calcular $N(n-1)$, lo cual se realiza comparando, de derecha a izquierda, los carácteres $P[1 \ldots (n-1)]$ con $P[1 \ldots n]$ hasta que se encuentre un no-apareo; el valor a asignar es la longitud de la subcadena que se apareó.

  Si $N(n-1) > 0$, entonces $l = l_{n-1} = n - N(n-1) + 1$ y $r = r_{n-1} = n - 1$ y si $N(n-1) = 0$ entonces $l = r = n$.

- A continuación, se asume inductivamente que el algoritmo ha calculado correctamente $N(i)\ \forall\ i \in \{n \ldots k+1\}$, por lo que conoce también los valores de $r = r_{k+1}$ y $l = l_{k+1}$; el próximo paso es calcular $N(k)$, recordemos que k se va decrementando, ya que la exploración es de derecha a izquierda.

Es importante mencionar el hecho de que al usar los valores de N anteriormente calculados se ahorra tiempo en ejecución, ya que un caso implicaría no realizar más operaciones y obtener de inmediato a $N(k)$. Un ejemplo de esto es: si tenemos a P como en el esquema anterior, con $k = 5$, se han calculado $N(16)$ hasta $N(6)$, $l = 2$ y $r = 8$; entonces sabemos que existe una subcadena, que empieza en $P(2)$ con longitud 7 y que se aparea con un sufijo de P, de la misma longitud (ver Figura 1.14), de lo anterior también sabemos que existe una subcadena β de longitud 4, que aparea a la subcadena de la misma longitud, que termina en $P(14)$, esto pasa porque existe la repetición de α en $P(2)$ y en $P(11)$; como $N(14) = 3$, la cuál ya se había calculado, quiere decir que existe una subcadena γ que termina en $P(14)$ que aparea a un sufijo de P, por lo que este valor es útil para el cálculo de $N(5)$; ya que $N(14) < |\beta| = 4$, entonces se deduce que $N(5) = N(14)$. Por lo que en este caso no fué necesario ninguna operación extra para calcular este valor.

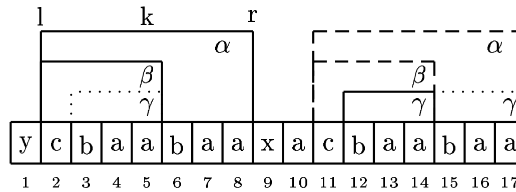


Figura 1.14: Cálculo de un valor N sin ninguna operación extra. Se ilustra la forma en que se calcula el valor de $N(5)$, aprovechando el conocimiento de información previamente obtenida.

El algoritmo N

Dadas $N(i)$ donde $k + 1 \leq i < n$, r y l, entonces $N(k)$ se calcula y r, l se actualizan de la siguiente manera:

1. Si $k < l$, encontrar $N(k)$ al comparar los carácteres, empezando en la posición k y seguir con las sucesivas comparaciones hacia la izquierda hasta que se encuentre un no-apareo. La longitud de la subcadena que se aparea es el valor asignado a $N(k)$. Si $N(k) > 0$, entonces $l = k - N(k) + 1$ y $r = k$.

2. Si $k \geq l$, entonces la posición k está contenida en una *N-caja*, por lo que ocurre lo siguiente:

   a) $P(k)$ está contenida en la subcadena $P[l \dots r]$ (a la que denotamos α), tal que $r < n$.

   b) α aparea un sufijo de P, que es $P[(n - N(r)) \dots n]$; como consecuencia $P(k)$ ocurre en la posición $k' = k + r + 1$.

   c) Existe $P[l \dots k]$, denotada con β, la cuál aparea a la subcadena $P[(n - N(r) + 1) \dots k']$. De lo anterior existe una subcadena que termina en la posición k y que se aparea con un sufijo de P de longitud $= min\{N(k'), |\beta| = k - l + 1\}$ (ver Figura 1.15).

   Cuando se ha elegido ese mínimo, surgen dos casos más:

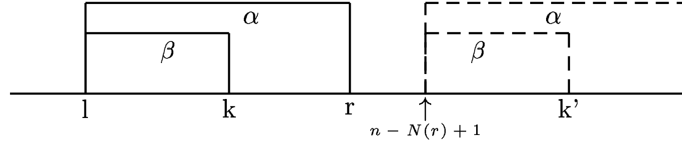


Figura 1.15: El algoritmo N, paso 2.c). La subcadena $P[l \ldots k]$, etiquetada con β también ocurre teniendo como posición final a k'.

1) Si $N(k') < |\beta|$, entonces $N(k) = N(k')$ y los valores de l y r permanecen inalterados (ver Figura 1.16).

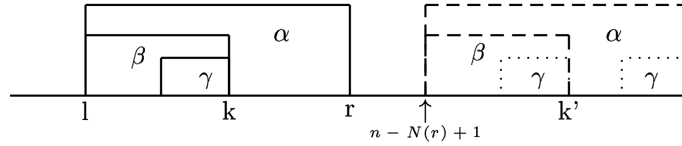


Figura 1.16: El algoritmo N, paso 2.c) con $N(k') < |\beta|$. Caso 2(a). La subcadena $\gamma = P[k' - N(k') + 1 \ldots k']$, cumple que $|\gamma| < |\beta|$ y es la subcadena más grande que termina en esa posición y que aparea un sufijo de S, en este caso $N(k) = N(k')$.

2) Si $N(k') \geq |\beta|$, entonces la subcadena $P[l \ldots k]$ debe ser un sufijo de P y $N(k) \geq |\beta| = k - l$. Como $N(k)$ podría ser estrictamente más grande que $|\beta|$, entonces comparamos los carácteres empezando en la posición $l - 1$ de P con el carácter en $n - |\beta| - 1$, y luego hacia la izquierda, hasta que ocurra un no-apareo. Supongamos que el no-apareo ocurre en el carácter $q \leq l - 1$. Entonces $N(k) = k - q$, $l = q + 1$ y $r = k$ (ver Figura 1.17).

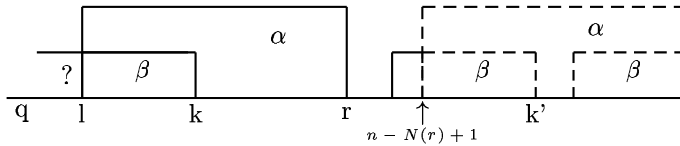


Figura 1.17: El algoritmo N, paso 2.c) con $N(k') \geq |\beta|$. Caso 2(b). La subcadena más grande que termina en k' que aparea a un sufijo de S, tiene longitud de al menos $|\beta|$, por lo que resta revisar los carácteres a la izquierda de l.

Teorema 1.3.3. *Usando el algoritmo N, el valor para $N(k)$ se calcula correctamente y las variables r y l se actualizan correctamente.*

Demostración. Caso 1. Si $N(k) > 0$, el algoritmo encuentra una nueva *N-caja* que empieza en, o bien antes de k; y como antes de que $N(k)$ sea calculada no existe ninguna

N-caja que cumpla con estas características, entonces es correcto asignar $l = k - N(k) + 1$ y $r = k$. Por tanto, en este caso el algoritmo trabaja correctamente.

Caso 2(a). La subcadena que termina en la posición k puede aparearse con un sufijo de P sólo si tiene longitud $N(k')$; supongamos que la subcadena tiene longitud mayor a $N(k')$, entonces $P[k - N(k')]$, deberá aparear a $P[n - N(k')]$; como $N(k') < |\beta|$, entonces $P[k - N(k')]$ deberá aparear a $P[k' - N(k')]$, por lo que este último deberá aparear a $P[n - N(k')]$, lo cuál es una contradicción a la definición de $N(k')$, pues implicaría la existencia de una subcadena más larga a la indicada por $N(k')$ que termina en k' y aparea a un sufijo.

De forma análoga surgiría una contradicción si suponemos que la subcadena que termina en la posición k tiene longitud menor a $N(k')$.

Por los dos casos anteriores, lo que queda es que $N(k) = N(k')$ y como $k - N(k) + 1 > l$, entonces r y l permanecen con sus valores anteriores.

Caso 2(b). Como se argumentó en el algoritmo, β debe ser un sufijo de P, por lo que se puede verificar si existe una prolongación de este apareo al comparar, de derecha a izquierda, desde el carácter $l - 1$ con $n - |\beta| - 1$, por lo que esta extensión al apareo es calculado correctamente y ya que $k - N(k) + 1 \leq l$, el algoritmo actualiza correctamente los valores de r y l. ❏

Teorema 1.3.4. *Todos los valores $N(i)$ son calculados por el algoritmo en tiempo $O(|P|)$.*

Demostración. Cada comparación de dos carácteres da origen a un apareo o a un no-apareo, así que los casos extremos serían que ocurrieran sólo apareos o sólo no-apareos, motivo por el cual nos dedicamos a "contar" el número de ellos que pudieran ocurrir.

Cada iteración que ejecuta algunas comparaciones de carácteres termina cuando se encuentra con un no-apareo; por lo tanto existen a lo más $|P|$ no-apareos durante todo el algoritmo. Para acotar el número de apareos, debemos considerar que $l_k \leq l + 1$, para cada iteración k. Luego, sea k una iteración donde ocurren $q > 0$ apareos. Entonces a l_k se le asigna, al menos, $l_{k+1} - q$. Finalmente, $l_k \leq |P|$, así que el número total de apareos que ocurre durante cualquier ejecución del algoritmo es a lo más $|P|$. Y además es claro que a cualquier otro caso se aplica también la misma cota superior, ya que es el complemento de apareos con no-apareos. ❏

Algoritmo para calcular los valores de $L(i)$

Ahora que ya sabemos como obtener los valores de N en tiempo lineal con respecto a $|P|$, procedemos a obtener los valores de L usando los primeros. Este procedimiento se muestra en el siguiente algoritmo, el cual usa también las relaciones dadas en el Teorema 1.3.4.

```
int[] LP = new int[ n+1 ];

for( int j=1; j <= n; j++ ){
    i = n-N[j]+1;
    LP[i] = j;
}
```

Listado 1.1: Cálculo de los valores de L' basado en N.

Los valores $L(i)$ pueden ser obtenidos, si se desea, al añadir las siguientes líneas al código anterior:

```
L[2] = LP[2]

for( int i = 3; i <= n; i ++ )
    L[i] = max(L[i-1], LP[i]);
```

Listado 1.2: Cálculo de los valores de L.

Teorema 1.3.5. *El Algoritmo Boyer-Moore basado en N, calcula correctamente los valores de L.*

Demostración. $L(i)$ es la posición final de la subcadena más a la derecha (que no es sufijo) que se aparea con $P[i \dots n]$. Por lo tanto, esa subcadena empieza en la posición $L(i) - n + i$, la cual denotamos por j.

Para probar que $L(i) = max\{L(i-1), L'(i)\}$ consideremos el carácter en la posición $j-1$ y supongamos que $j > 1$ (ya que si $j = 1$, entonces el carácter $j-1$ no existe y entonces $L(i-1) = 0$ y $L'(i) = L(i)$). Si $P(j-1) = P(i-1)$ entonces $L(i) = L(i-1)$. Si $P(j-1) \neq P(i-1)$ entonces $L(i) = L'(i-1)$. De esto se concluye que en todos los casos, $L(i)$ debe ser $L'(i)$ o $L(i-1)$. Sin embargo no es todo, ya que $L(i)$ debe ser más grande que o igual a ambos valores($L'(i)$ y $L(i-1)$) para cumplir con su definición, por lo que entonces tomamos el valor máximo entre ellos. ❏

El uso de los valores L y L' en ABM

Como anteriormente mencionamos, cuando ocurre un no-apareo en el carácter $i-1$ de P y $L'(i) > 0$, se reubica al patrón por $n - L'(i)$ posiciones (ver Figura 1.9); esta exigencia de $L'(i) > 0$ es precisamente porque nos garantiza que el sufijo $P[i \dots n]$ se repite en algún lugar del patrón[7]. En caso de que dicho sufijo no reaparezca en P, es decir que $L'(i) = 0$, la

[7] Con las debidas restricciones que se dan en las definiciones de $L(i)$ y $L'(i)$, según la que se esté utilizando.

regla fuerte del buen sufijo nos dice que reubiquemos a P por la menor cantidad posible, para que un prefijo de P se aparee con un sufijo de la subcadena de T con la que se está comparando al patrón: esta última acción se lleva a cabo también si P ha ocurrido en T; el ABM está preparado para resolver tales casos y no perder una ocurrencia de P, para lo cual se usa lo siguiente:

Definición 1.3.7. *$l'(i)$ denota la longitud más grande del sufijo de $P[i \dots n]$ que es también un prefijo de P. Si no existe, entonces $l'(i)$ es cero.*

Por ejemplo, si tenemos que $P = abdabdab$ entonces $l'(2) = 5$ y $l'(5) = 2$, como se muestra en la Figura 1.18.

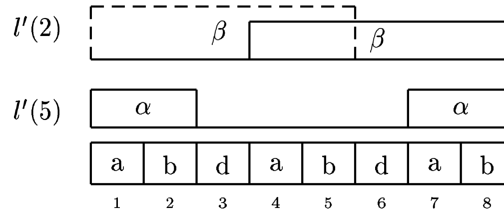


Figura 1.18: Valores de l'. Las subcadenas que forma la definición $l'(i)$, para: $i = \{2, 5\}$.

Teorema 1.3.6. *$l'(i) = \max\{j \mid j \leq |P[i \dots n]|$ y $N(j) = j\}$, donde $|P[i \dots n]| = n-i+1$.*

Demostración. Por definición, si $N(j) = j$ y como $|P[1 \dots j]| = j$, entonces tenemos una subcadena α de longitud j que aparece como prefijo y como sufijo en P. Luego, por definición $l'(i)$ es la longitud más grande, por lo que recorremos el arreglo de N, de derecha a izquierda, para obtener los valores mayores donde $N(j) = j$. Al ir encontrando estas posiciones y si $j = |P[i \dots n]|$ significa que tenemos un prefijo de longitud j que también es sufijo en P, además esa longitud es máxima para $P[i \dots n]$ pues es toda la subcadena.

El otro caso que puede ocurrir es que $j < |P[i \dots n]|$, lo cual significa que tenemos un prefijo de longitud j que es sufijo de P y que está contenido en $P[i \dots n]$, y es sufijo de longitud máxima, por lo que también cumple con la definición de $l(i)$. ❏

El algoritmo que calcula los valores de $l'(i)$

Haciendo uso del Teorema 1.3.6, podemos calcular los valores de l; la idea es encontrarlos empezando desde n y moviéndonos hacia 1, mientras que recorremos a N al revés, lo cual tiene sentido por la relación que existe entre dichos arreglos.

```
int[] lp = new int[n+1];
int max = 0;
int j = 1;

for( int i = n; i >= 1; i-- ){
    if( j == N[j] && max < j )
        max = j;
    lp[i] = max;
    j++;
}
```

Listado 1.3: Cálculo de los valores de l'.

Es claro que este algoritmo se ejecuta en tiempo lineal con respecto a $|P|$, lo cual queda demostrado al recorrer una sola vez cada posición de P en el segundo ciclo.

Ahora que ya sabemos lo que $L'(i)$ y $l'(i)$ significan en P y que además conocemos la forma de obtenerlas en tiempo lineal con respecto a $|P|$, sólo falta indicar su uso en el ABM.

Usando los valores de L y l'

En caso de que ocurra un no-apareo en $i-1$ y $L'(i) > 0$, con la *regla del buen sufijo* se recorre a P por $n - L'(i)$ lugares a la derecha. Si $L'(i) = 0$, entonces P se recorre por $n - l'(i)$ lugares a la derecha.

El otro caso es cuando se encuentra una ocurrencia de P: se recorre a P por $n - l'(2)$ lugares a la derecha.

Si $l'(i) = 0$ y un prefijo de P no puede aparearse con un sufijo de la subcadena T_1(la cadena de T con la que se comparó al patrón) entonces P se recorre pasando T_1 en T, es decir recorre n lugares; de ésta manera obtemos los valores $n - l'(i) = n - 0 = n$, por lo que dicha regla funciona correctamente, también en este caso.

1.3.5. El algoritmo Boyer-Moore

El ABM ocupa las dos reglas que anteriormente estudiamos, la *regla del buen sufijo* y la *regla del carácter erróneo*, para mover a P según el conocimiento que se tenga sobre los carácteres que lo componen y el carácter que se conoce de T en un momento determinado. Un objetivo claro es tratar que las reubicaciones de P sean lo más grandes posibles, por lo que si ya sabemos que un reposicionamiento nos conducirá a no-apareos, lo mejor es entonces tomar el mayor salto que estas dos reglas regrese, y de alguna forma cumplir con nuestro propósito.

```
calculaL(); //
calculalp();//Etapa de preprocesamiento
calculaR(); //
```

```
int k = n;

while(k >= m){
    i = n;
    h = k;

    while( i > 0 && P[i] == T[h]){
        i --; h --;
    }
    if( i == 0 ){
      reportaPOcurreTerminando( k );
      k += n - lp[2];
    }else{
      k = maxEntreCaracterErroneoYBuenSufijo( k );
    }
}
```

Listado 1.4: ABM.

Con esta forma de actuar del ABM, y si existe una ocurrencia de P en T, la posición que se regresa es la final donde P ocurrió en T.

Esta versión del algoritmo se presenta en [4]. Con estas definiciones Gusfield[4] logró que el Algoritmo de Boyer Moore, así como las ideas que en él se usan, logren ser entendibles, pero también tenemos que resaltar que realizó algunos cambios a la versión original para que su objetivo se cumpliera de forma sencilla.

Capítulo 2

Versión original del *ABM*

2.1. Introducción

En 1977 Robert S. Boyer y J. Strother Moore presentan un algoritmo que tiene la misma complejidad, en teoría, que el que anteriormente habían presentado Knuth, Morris y Pratt, sin embargo para ciertos casos, el tiempo de ejecución es sublineal con respecto a $i + |P|$, donde i es la posición en la cuál el patrón ocurre en T.

En el Capítulo 1 estudiamos una versión del ABM que Gusfield presenta en [4], donde los cambios que se realizan buscan obtener un mejor entendimiento de dicho algoritmo.

En este capítulo se estudia la versión original del ABM, haciendo enfásis en las diferencias con respecto a la versión de Gusfield y realizando también los cambios pertinentes para que la versión de dicho algoritmo se transforme en la versión original del ABM.

2.2. La Versión Original del ABM

La idea que usa el algoritmo original de Boyer-Moore, al que denotaremos como AOBM, es muy similar a la que estudiamos en el capítulo anterior en cuanto a la forma de aprovechar el conocimiento previo que se va obteniendo de P y la dirección en la que se comparan los carácteres.

AOBM usa dos reglas δ_1 y δ_2, las cuales determinan, a diferencia del algoritmo del ABM, la posición de T que se alinea con la posición final de P, para iniciar una nueva comparación en la cual lleva implícitamente el movimiento del patrón, como lo hacían la *regla del buen sufijo* y la *regla del carácter erróneo*. Por tal motivo si P ocurre en T, este algoritmo regresa la posición inicial en T donde el patrón ocurrió.

Para ejemplificar las diferencias de ambos algoritmos en este sentido, nos fijamos en la Figura 2.1, en donde dados P y T, podemos observar que la subcadena $P[6 \dots 7] = eb$, se repite en $P[3 \dots 4]$ y los carácteres que preceden a dichas subcadenas son distintos ($P(5) \neq P(2)$). Así que al encontrar un no-apareo de $T(7)$ con $P(5)$, la mejor opción es reubicar a P para que $P(2)$ quede abajo de $T(7)$; con ABM, la *regla del buen sufijo* regresa 3, que es la indicación del número de posiciones que P se recorre para que dicha alineación ocurra; mientras que AOBM inicia con un apuntador, al que llamaremos *apt*, al carácter

en T que se alinea con $P(n)$ ($T(9)$ en la misma Figura) y al empezar la exploración éste va decrementando hasta encontrarse un no-apareo en $T(7)$; entonces los movimientos están dados para que *apt* vuelva a colocarse al carácter en T que queda alineado con $P(n)$, después de haber recorrido a P los lugares suficientes, y se cumpla la alineación de $P(2)$ con $T(7)$. De lo anterior, en AOBM se regresa 5 que indica el movimiento del apuntador, el cual lleva implícitamente el recorrido de P por tres posiciones.

		1	2	3	4	5	6	7	8	9	10	11	12	13
	T:	a	e	d	a	c	d	a	e	b	c	e	b	c
1ª alin.	P:			d	a	e	b	c	e	b				
				1	2	3	4	5	6	7				
								⋆	√	√				
2ª alin.							d	a	e	b	c	e	b	
							√	√	√	√	√	√	√	

Figura 2.1: Alineación en AOBM. Las dos alineaciones que tienen que realizarse de P con T al conocer la repetición de *eb* en $P(3)$ y empezar de nuevo la siguiente comparación.

Estos valores que AOBM maneja son dados en las tablas δ_1 y δ_2, en donde la primera es la equivalente a la *regla del carácter erróneo* y la segunda es equivalente a la *regla del buen sufijo.*

2.3. Regla δ_1

Como anteriormente mencionamos, esta tabla guarda los valores equivalentes a los que genera la *regla del carácter erróneo*, sin su extensión.

Cuando AOBM empieza su proceso, supone la existencia de la tabla δ_1 la cual tiene longitud de tamaño $|\Sigma|$, en donde el valor correspondiente a cada carácter del alfabeto está definido a continuación.

Definición 2.3.1. *Para cada $x \in \Sigma$, si x ocurre en P, $\delta_1(x) = |P| - j$, en donde j es la mayor posición de P, tal que $P(j) = x$. Si x no ocurre en P, entonces $\delta_1(x) = |P|$.*

Con esta definición, $\delta_1(P(n)) = 0$ para todo patrón con longitud n.

Para calcular estos valores, debemos notar que la definición lleva implícitamente el cálculo de la j más grande donde $P(j) = x$, para cada $x \in \Sigma$; estos valores ya sabemos cómo obtenerlos, pues los calculamos en el Capítulo 1 con el nombre de $R(x)$, así es que su cálculo lo damos por hecho.

2.3.1. Los valores δ_1

Para calcular los valores de $\delta_1(x)$ se realiza lo siguiente: $\delta_1(x) = |P| - R(x)\ \forall x \in \Sigma$. Como puede verse esta regla también funciona en el caso en que x no aparece en P, pues

$R(x)$ da 0 como resultado según su definición, por lo que $\delta_1(x) = |P|$.

Para ejemplificar la forma en que AOBM funciona usando sólo esta tabla, calcularemos los valores $\delta_1(x)$ para P, con $\Sigma = \{A, B, C, X, Y, Z\}$ el cuál está formado de la siguiente manera:

	1	2	3	4	5	6	7	8	9
P:	A	B	C	X	X	X	A	B	C

Al calcular los valores de R, obtenemos:

	A	B	C	X	Y	Z
R=	7	8	9	6	0	0

por lo que los valores de δ_1 quedan:

	A	B	C	X	Y	Z
δ_1=	2	1	0	3	9	9

No debemos olvidar que AOBM, a diferencia de ABM, busca con δ_1 encontrar la posición en T que se alinea con $P(n)$, después de haber recorrido al patrón y llegar a un no-apareo.

AOBM funciona de la siguiente forma: si al comparar a $T(i)$ con $P(j)$ ocurre un no-apareo y si q es el total de carácteres que se aparearon antes de surgir éste, entonces:

- Si la ocurrencia más a la derecha de $T(i)$ en P es a la izquierda de $P(j)$, es decir, si $R(T(i)) < j$, entonces *apt* apunta a $apt + \delta_1(T(i))$. Se puede observar esta forma de actuar del AOBM en el ejemplo de la Figura 2.2.

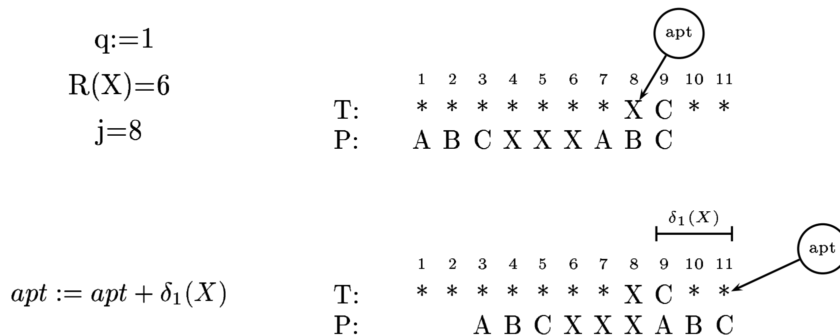


Figura 2.2: Movimiento en δ_1 ($apt + \delta_1(X)$). La primera comparación fallida es de $T(8)$ con $P(8)$, por lo que *apt* se queda apuntando a $T(8)$; como $6 = R(X) < j = 8$, entonces la nueva comparación se realiza con *apt* apuntando a $apt + \delta_1(X)$, que resulta en *apt* apuntando a $T(11)$. Después de esto, el proceso de apareamiento se reinicia.

- Si la ocurrencia más a la derecha de $T(i)$ en P es a la derecha de $P(j)$, es decir, el valor de $R(T(i)) \geq j$, entonces *apt* apunta a $apt + q + 1$, porque si quisiéramos alinear a $R(T(i))$ con $T(i)$, significaría mover a P hacia la izquierda, lo cual sería una reubicación vana pues implicaría volver a comparar carácteres que sabemos conducen a un no-apareo de P con la subcadena respectiva en T. Este proceso se observa en la Figura 2.3.

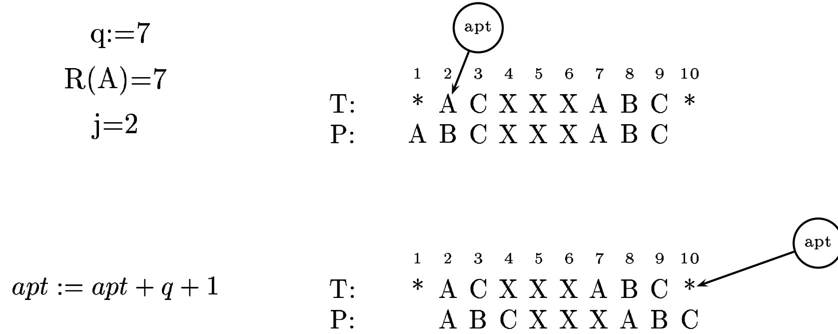


Figura 2.3: Movimiento en δ_1 $(apt+q+1)$. El símbolo $*$ representa cualquier carácter colocado en una posición de alguna cadena. La primera comparación fallida es de $T(2)$ y $P(2)$, por lo que *apt* se queda apuntando a $T(2)$; en este caso, como $7 = R(A) > j = 2$, entonces la nueva comparación se realiza con *apt* apuntando a $apt + q + 1$, que resulta en *apt* apuntando a $T(10)$.

2.4. Regla δ_2

Esta tabla maneja los valores equivalentes a los de la *regla del buen sufijo*, pues su idea en el manejo de subcadenas es la misma que como lo hace dicha regla, sólo que no hay que olvidar que AOBM maneja a *apt* como un elemento primordial, por lo que añade otra característica al significado de estos valores.

El AOBM también trabaja sobre los valores contenidos en la tabla δ_2, la cual tiene longitud $|P|$. El valor que $\delta_2(i)$ almacena lleva implícitamente dos valores:

- la distancia que se debe mover a P hacia la derecha para que la reocurrencia de los últimos $|P| - j$ carácteres de P se alineen con la ocurrencia de los mismos en T.

- más la distancia adicional que corresponde mover a *apt* al carácter de T que se alinea con el último de P después de haber reubicado a P como el punto anterior lo indica.

Para ejemplificar como AOBM funciona al usar esta tabla, tomaremos el alfabeto y el patrón que usamos para definir a δ_1 y el ejemplo que se muestra en la Figura 2.4.

Recordemos de la *regla del buen sufijo*, que para una posición i en P en la que ha ocurrido un no-apareo, se construye el sufijo de $P[i + 1 \ldots n]$ para el cuál se desea que

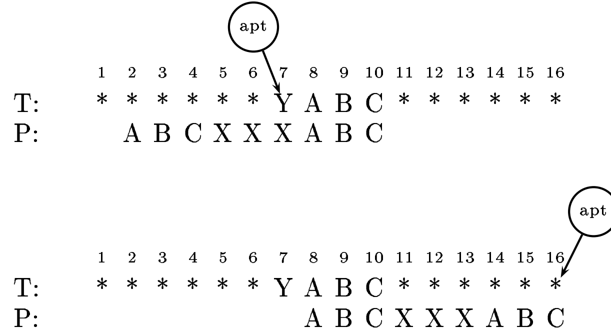


Figura 2.4: Movimiento en δ_2. Al realizar la primera comparación y obtener la primera comparación fallida de $T(7)$ con $P(6)$, *apt* se queda apuntando a $T(7)$ y al realizar el análisis correspondiente, la regla de δ_2 nos indica mover a *apt* 9 posiciones para poder realizar la siguiente comparación de T y P dada la siguiente alineación.

exista la copia α en P, siempre y cuando $P(i)$ sea distinto al carácter que precede a α; así en nuestro ejemplo al encontrar el no-apareo, *apt* se queda apuntando a $T(7)$, pero al saber que el sufijo α constituído por $P[7\dots 9] = ABC$, se repite en la posición 1 y los carácteres que preceden a dichas subcadenas son distintos, entonces P se recorre por 6 posiciones para que α sea alineado con su copia, por lo que *apt* queda apuntando a $apt + 9 = 7 + 9 = 16$. Calculando de esta forma los valores para cada posición de P, obtenemos que la tabla δ_2 queda así:

	1	2	3	4	5	6	7	8	9
P:	A	B	C	X	X	X	A	B	C
δ_2:	14	13	12	11	10	9	11	10	1

Ahora veremos la forma de calcular los valores de δ_2. Para ello primero debemos definir la *reocurrencia plausible más a la derecha* del sufijo de P, para cada $i \leq n$, que es la posición inicial (del lado derecho) de la subcadena que es copia del sufijo $P[j+1\dots n]$, tal que $P[j]$ es distinto al carácter que precede a dicha subcadena de P; para esto enunciaremos algunas definiciones.

Definición 2.4.1. *Sea \$ un carácter que no ocurre en P. Si $i < 1$ entonces $P(i)$ es \$.*

Definición 2.4.2. *Decimos que dos secuencias de carácteres $[c_1 \dots c_n]$ y $[d_1 \dots d_n]$ se* **unifican** *si para todo i con $1 \leq i \leq n$ $c_i = d_i$ o $c_i = \$$ o $d_i = \$$.*

Definición 2.4.3. *La posición de la reocurrencia plausible más a la derecha del sufijo de P que empieza en la posición $j+1$, a la que denotamos por* **rpr(j)** *es la k más grande menor o igual que n tal que $P[(j+1)\dots n]$ y $P[k\dots(k+n-j-1)]$ se unifican, ya sea que $k \leq 1$ o bien $P(k-1) \neq P(j)$.*

Una vez calculados los valores $rpr(i)$, se usan para el cálculo de los valores de δ_2 con la siguiente fórmula:

$$\delta_2 = |P| + 1 - rpr(j)$$

Con la definición de rpr, muchas veces el valor de k resulta negativo. En cierta medida, es esta regla la que le ha otorgado al algoritmo su fama de "obscuro" pues en el artículo original no dice exactamente cómo se realiza la elección del valor para k, cuando no ocurre la copia del sufijo en P. Una forma un tanto distinta de calcular estos valores, pero que sigue la misma idea, la estudiaremos en la siguiente sección con un algoritmo presentado en [1].

2.5. AOBM

Este algoritmo[1] también busca lograr que P sea recorrido por los saltos mayores posibles, de tal modo que se asegure no se salte una ocurrencia de P en T, por lo que toma el máximo entre el valor que obtiene de δ_1 y δ_2.

```
textl := |T|;
i := |P|;

top: if i > textl
        then
        return false

j := |P|

loop: if j = 0
        then
        return i+1

       if T[i] = P[j]
        then
        j:= j - 1
        i:= i - 1
        goto loop
        close

        i:= i + max( δ1(T[i]) , δ2[j] )
        goto top;
```

Listado 2.1: AOBM.

En el siguiente capítulo estudiaremos otra modificación a este algoritmo, el cuál conserva de forma más precisa las ideas para calcular los valores correspondientes a las reglas δ_1 y δ_2 que se presentaron en el algoritmo original.

[1]Hoy en día la organización del algoritmo se ve muy anticuada, por las transferencias de control que pudieran haber sido estructuradas con iteraciones.

Capítulo 3

El algoritmo de Boyer-Moore y δ_2

3.1. Introducción

El algoritmo que plantea Baase en su libro[1] es un tanto diferente al $AOBM$ en el cálculo y en el manejo de los valores, ya que son equivalentes a los que hemos estado trabajando. Sin embargo, el objetivo de su estudio es el entendimiento de la regla respectiva a δ_2, y para lograrlo presentaremos el algoritmo completo, omitiendo las partes correspondientes a cálculos que anteriormente hemos realizado, y remarcando la forma de encontrar valores que no habíamos obtenido. Con esto lograremos tener una noción distinta de cómo calcular los movimientos de *apt*, pues aunque la idea para obtenerlos es la misma que mencionamos en el capítulo anterior, la forma de calcularlos es muy intuitiva, y entonces podríamos sustituir esta parte en el AOBM para entenderlo de mejor manera.

Para conservar la idea del AOBM, Baase[1] también utiliza dos reglas distintas para saber cuál es la cantidad por la cual debe mover a *apt*, después de haber ocurrido un no-apareo y poder volver a recomenzar la exploración del texto. Las reglas que utiliza son: *la idea nueva (the new idea)* y *la idea antigua (the old idea)*, las cuales presentaremos a continuación.

3.2. La idea nueva: salto por carácter *(charJump)*

Estos valores representan a los de δ_1, en el AOBM. Se resume el comportamiento de la siguiente manera: si al realizar la exploración ocurre un no-apareo de $P(i)$ con $T(j)$, y este último no ocurre en P, entonces al reiniciar la exploración *apt* debe ser incrementado por n posiciones sobre T, de tal forma que quede apuntando al carácter en T que coincida con el último de P para reiniciar el proceso de búsqueda, como sucede en el ejemplo que se presenta en la Figura 3.1

En caso de que el carácter $T(j)$ ocurra en P en la posición k, entonces el cálculo de estos valores se realiza dependiendo de la posición de k con respecto a i. Este proceso se resume en dos casos, ya sea que $k < i$ o que $k \geq i$, como habíamos visto en el capítulo anterior; asimismo sabemos cómo usarlos según se requiera.

Una de las diferencias de este algoritmo con otros para apareamiento de texto radica

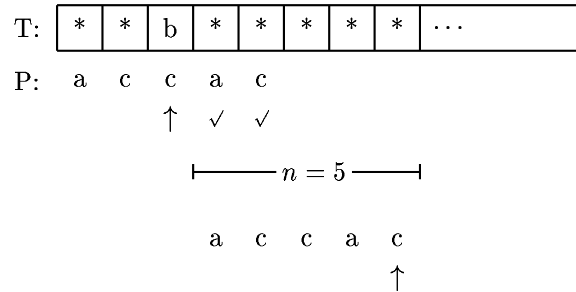


Figura 3.1: *Salto por carácter.* Al ocurrir un no-apareo de $T(3)$ con $P(3)$, es en esa posición en donde *apt* se queda apuntando (↑). Hay que notar que $T(3)$ no figura en P, por lo que *apt* se tiene que reubicar n lugares (es decir queda apuntando a $T(8)$) para volver a quedar apuntando al carácter en T que se alinea con el último de P y poder reiniciar la exploración.

principalmente en el cálculo de éstos valores, ya que se realiza "al vuelo", es decir no es necesario usar arreglos de valores que auxilien al cálculo principal como lo hacíamos con $R[i]$; además que aprovecha algunos comportamientos de P para los carácteres que lo constituyen y que sólo se aplica al caso 1.

Para ver el funcionamiento nos auxiliaremos de la Figura 3.2, en la cual observamos que al ocurrir el no-apareo de $T(3)$ con $P(3)$, y como el carácter en T que lo ocasionó ocurre en $P(1)$, entonces para que *apt* sea reubicado se toman en cuenta dos valores: (a) el número q de carácteres que se lograron aparear antes de que ocurriera el no-apareo y (b) el número de posiciones que bastaría mover al patrón para que $P(1)$ se alinee con $T(3)$; con esto *apt* se reubicaría justo por $q + p = n - 1 = 5$ posiciones, donde $|P| = n$ y 1 es la posición mayor donde a ocurre en el patrón.

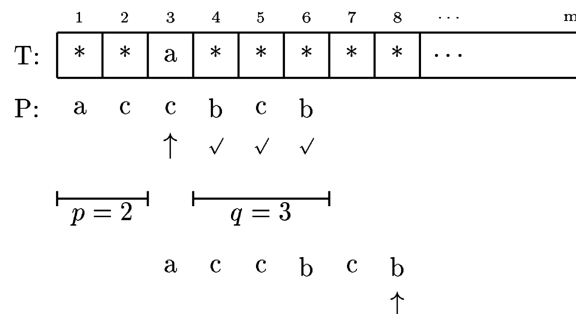


Figura 3.2: Uso de la *regla salto por carácter.* Se muestran los valores que se deben tomar en cuenta para reubicar a *apt* después de haber ocurrido un no-apareo, dado que el carácter en T ocurre en P.

A continuación presentamos el código que calcula estos valores.

```
void calculoSaltosPorCaracter(){
        char ch;
```

```
    int k;
    int n = Pattern.lenght;

    for ( ch = 0; ch < Sigma.length; ch ++ )
          saltoPorCaracter[ ch ] = n;
    for ( k = 1; k <= n; k ++ )
          saltoPorCaracter[ P[k] ] = n - k;
}
```

Listado 3.1: Cálculo de los valores de *salto por carácter.*

3.3. La idea antigua: saltos por apareos *(match Jumps)*

Esta es la regla equivalente a δ_2 que usa la idea del uso de *sufijos*; como al calcular los valores lo hace de una forma distinta a como lo mostramos en capítulos anteriores, introduciremos algunos nuevos conceptos, mientras que omitiremos la explicación de cómo se usan los sufijos, invitando al lector a regresar al Capítulo 1.3.4 para referencias.

Definición 3.3.1. ***reubica[i]****, para* $1 \leq i \leq n$*, es el número de posiciones para* ***reubicar a*** P *después de haber ocurrido un no-apareo en* $P(i)$*.*

Sabemos que si un apareo ocurre en la posición i entonces $n-i$ representa la cantidad de carácteres que fueron apareados antes de ocurrir el no-apareo; la suma de $(n-i)+reubica[i]$ corresponde a qué tan lejos *apt* debe ser reubicado. Estos valores pueden ser almacenados en un arreglo de tamaño $|P|$ de tal forma que al ocurrir un no-apareo en la posición $i-$ésima de P (que es en la que está apuntando *apt*), el valor en este arreglo nos dirá por cuántas posiciones tendrá que ser recorrido *apt* y así estar listo para reiniciar la exploración; formalmente la definición queda de la siguiente manera:

Definición 3.3.2. ***saltoPorApareo[i]*** $= reubica[i]+n-i$*, para* $1 \leq i \leq n$*, es la cantidad por la que se debe* ***incrementar*** *a apt para reiniciar la exploración, después de haber ocurrido un no-apareo en* $P(i)$*.*

Para ayudarnos a entender un poco mejor estas dos definiciones y su dependencia, utilizaremos la Figura 3.3, en la que se representan los casos generales del comportamiento de P y T durante el proceso de exploración.

En la Figura 3.3(b), se observa el significado del valor $reubica[i]$ y la forma en que se usa para saber cómo incrementar a *apt*, ya que a partir de éste podemos obtener el valor para $saltoPorApareo[i]$.

Después de observar esta figura, podemos darnos cuenta que, si r es la posición en P donde comienza la copia del sufijo α y éste es máximo, entonces $reubica[i] = i - r$. Por lo que si reubicamos a P por esta cantidad, origina que ambas copias de α, la de T y la segunda de P, quede alineadas.

Esta regla se usa con las condiciones de *la regla fuerte del buen sufijo*, es decir restringiendo que $P(r) \neq P(i)$.

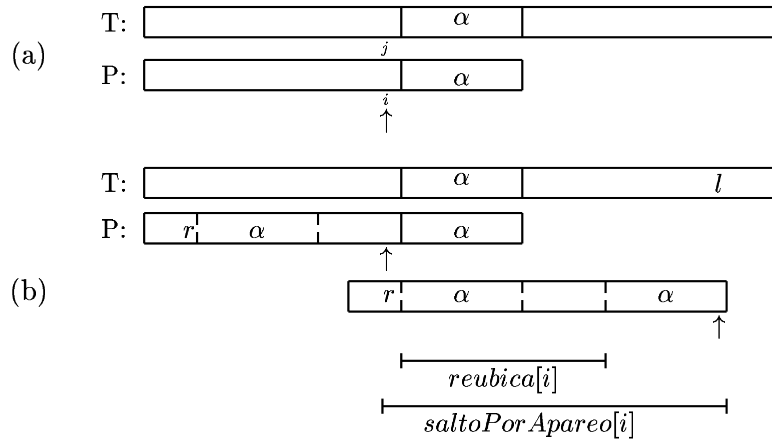


Figura 3.3: Uso de *reubica* para mover a *apt* I. **(a)** Existe un no-apareo de $T(i)$ con $P(j)$. **(b)** $reubica[i]$ se utiliza para saber por cuanto reposicionar a P después de que ha sucedido el no-apareo. Los movimientos de *apt* están representados por $\uparrow$, los cuales apuntan al carácter en T que se alinea en esa posición, es decir a $T(j)$ y a $T(l)$, respectivamente.

Análogamente a la figura anterior, en la Figura 3.4 se muestra la forma en que se recorre a P al no ocurrir la copia de la subcadena α, así como los respectivos valores para $reubica[i]$ y $saltoPorApareo[i]$.

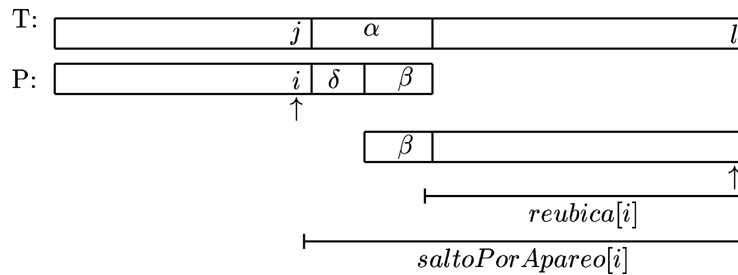


Figura 3.4: Uso de *reubica* para mover a *apt* II. En esta figura $\alpha = \delta\beta$. Al no ocurrir la copia de α en P, el patrón se reubica para encontrar un *prefijo* que se apareó con un sufijo de α en T.

Cuando se presenta el caso que se muestra en la Figura 3.4, y si el *prefijo* tiene q carácteres, entonces $reubica[i] = n - q$.

Ahora calcularemos estos valores dado el patrón $P = abaaba$ con $n = 6$, como en la Figura 3.5. El cálculo de los valores se realiza recorriendo a P de derecha a izquierda. Los valores que van siendo calculados, se colocan bajo el carácter correspondiente, y el símbolo $\uparrow$ señala al carácter con el que se está trabajando en un momento dado. En cada paso el patrón se reubica hacia la derecha para alinearlo con una subcadena que aparea a un sufijo. El carácter que precede a la subcadena debe ser distinto del carácter que precede al sufijo, como se muestra en la Figura 3.5(b). De esta forma, para la posición más a la

derecha $saltoPorApareo[6] = 1$, ya que $P(5) \neq P(6)$.

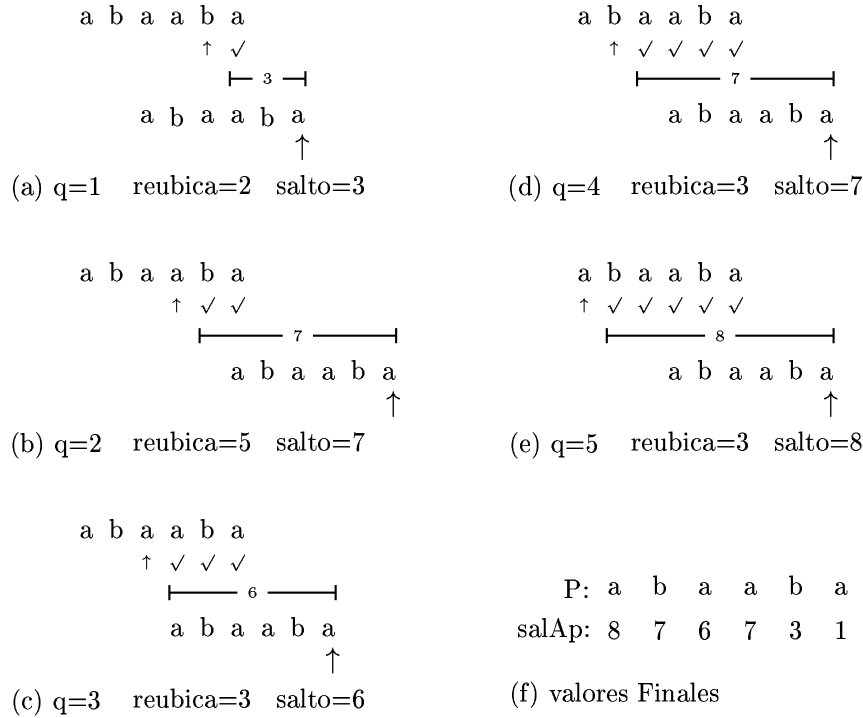


Figura 3.5: Cálculo de los valores de *salto por Apareo*. q es un número supuesto de carácteres apareados antes de ocurrir el no-apareo (es decir el número total de $\surd$); *reubica* es el número de posiciones por las que se tiene que reubicar a P para continuar con el proceso de exploración y *salto* es el número de posiciones por las que se tiene que mover a *apt* para que apunte al carácter alineado con $P(n)$, después de reubicar a P.

Antes de continuar con el desarrollo de este algoritmo, introduciremos algunos otros conceptos que serán auxiliares en el cálculo de los valores para el arreglo *saltoPorApareo*.

Definición 3.3.3. *$sufijo[k] = r$, si existe una subcadena, $P[(k+1)\ldots(k+n-r)]$ que se aparea con el sufijo $P[(r+1)\ldots n]$, donde $r > k$ y esta subcadena es de longitud máxima*[1].

En palabras más simples, en $sufijo[k]$ vamos a registrar la posición del sufijo más grande de P que se aparea con la subcadena que empieza en la posición $k+1$. En la Figura 3.6 se observa la representación gráfica de esta definición.

Para calcular estos valores se usa un algoritmo recursivo, que aunque no es la forma más eficiente de hacerla, sí es muy intuitiva; además, con un buen manejo de los valores

[1] Esta definición se presenta en su forma inversa en [1], así es que hemos realizado los cambios pertinentes para su uso en esta sección; de igual forma también se realizaron los cambios en el algoritmo que calcula estos valores.

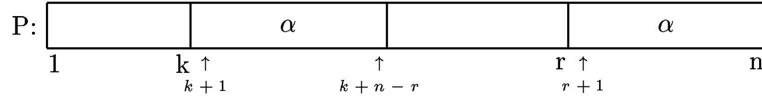


Figura 3.6: Representación de $sufijo[k] = r$. Se representa con α a las dos subcadenas de P que se aparean.

que se han calculado hasta antes del tiempo k, se logra obtener un tiempo de ejecución del *O(n)*. Este consiste en recorrer el patrón de derecha a izquierda, tomando como caso base[2] $sufijo[n] = n + 1$. Luego suponemos que se han calculado los valores para los índices n hasta $k + 1$, donde para ejemplificar supondremos que $P(k + 1) = s$, como se muestra en la Figura 3.7(a); el siguiente paso es calcular el valor para $sufijo[k]$. El caso más fácil es cuando tenemos que $P(k + 1) = P(s)$, y supondremos que el carácter en esas posiciones es a, entonces se formaría la subcadena $a\alpha$ empezando en $P(k + 1)$ y en $P(s)$, como se muestra en la Figura 3.7(b), por lo que $sufijo[k] = s - 1$.

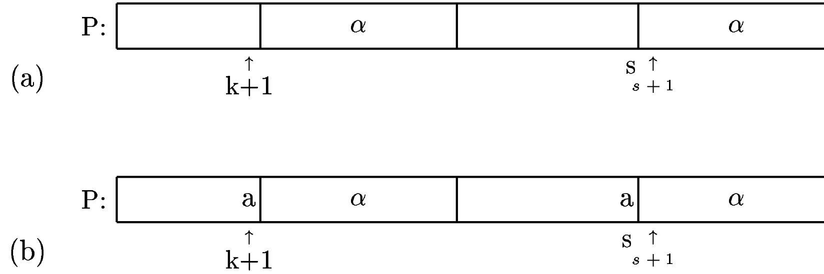


Figura 3.7: Cálculo de los valores de *sufijo* I. **(a)** $sufijo[k + 1] = s$. **(b)** Como $P(k + 1) = P(s)$, entonces $sufijo[k] = s - 1$.

Si $P(k + 1) \neq P(s)$, entonces debemos encontrar un sufijo de P que se aparee con una subcadena empezando en $s + 1$; como el tamaño de la subcadena α que se representa en la Figura 3.7(b) no puede ser extendido, entonces realizamos la búsqueda hacia los valores que ya habíamos calculado, es decir, hacia el final del patrón (recordemos que el cálculo de los valores es de derecha a izquierda). Así pues nos fijamos en el valor de $sufijo[s] = s_2$, lo cuál conlleva a la existencia de un sufijo de P que empieza en $s_2 + 1$ que se aparea con una subcadena empezando en $s + 1$; su representación gráfica se muestra en la Figura 3.8. Repetimos el proceso anterior, es decir, si $P(k+1) = P(s_2)$, entonces $sufijo[k] = s_2 - 1$; si $P(k+1) \neq P(s_2)$ entonces nuestro nuevo punto de partida es $sufijo[s_2]$; y así continuamos hasta que $P(k+1)$ sea igual al carácter en P que se esté explorando en un momento dado, o hasta que $sufijo[i] = n + 1$, para alguna i.

Con la anterior forma de encontrar los valores del arreglo $sufijo$, presentamos el algoritmo que los calcula.

[2]Este valor puede saberse de forma sencilla al aplicar la definición para $k = n$.

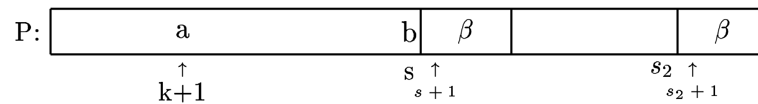


Figura 3.8: Cálculo de los valores de *sufijo* II. Como $P(k+1) \neq P(s)$, nos fijamos en $sufijo[s] = s_2$.

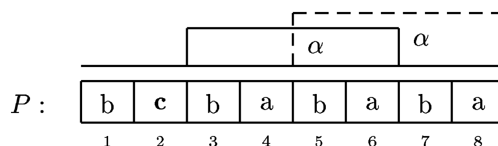
```
calculaSufijos() {

    int s, n = Pattern.length;
    int[] sufijo = new int[ n+1 ];

    sufijo[n] = n+1;
    for( int k = n-1; k >= 0; k-- ){
        s = sufijo[k+1];
        while( s <= n ){
          if( P[k+1] == P[s] )
              break;
          s = sufijo[s];
        }
        sufijo[k] = s - 1;
    }
}
```

Listado 3.2: Cálculo de los valores de *sufijo*.

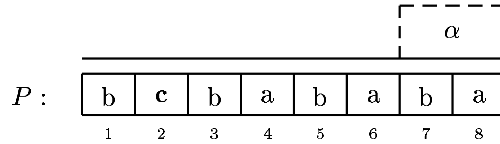
Consideremos un ejemplo particular que nos auxilia en la comprensión del uso recursivo del algoritmo *calculaSufijos* para obtener los valores del arreglo $sufijo$. Sea el patrón $P = bcbababa$ y supongamos que tenemos calculados los siguientes valores de dicho arreglo:

	1	2	3	4	5	6	7	8
P:=	**b**	**c**	**b**	**a**	**b**	**a**	**b**	**a**
sufijo		4	5	6	7	8	8	9

Como el cálculo de estos valores se realiza de derecha a izquierda, nuestro siguiente valor a calcular es $sufijo[1]$; sabemos que $sufijo[2] = 4$, es decir que el *sufijo* que empieza en $P(4 + 1) = P(5)$ y tiene longitud 4 se aparea con la subcadena de P que empieza en la posición $2 + 1 = 3$ y que tiene la misma longitud que la anterior. Estas subcadenas se muestran a continuación:



Si $P(2)$ fuera igual a $P(4)$, entonces a las dos subcadenas anteriores se les concatenaría $P(2)$, por ejemplo, y entonces resultaría que $sufijo[1] = sufijo[2] - 1 = 3$; sin embargo vemos que $P(2) \neq P(4)$, por lo que entonces nos fijamos en $sufijo[4] = 6$, con lo cuál se forma el siguiente sufijo:



donde sabemos que $P(6) \neq P(2)$, por lo que no coinciden las subcadenas correspondientes, así es que una vez más recurrimos a nuestros valores previamente calculados, por lo que ahora nos fijamos en el valor de $sufijo[6] = 8$, siendo con este valor con el cuál se cumple la definición; entonces obtenemos que $sufijo[1] = 8$ quedando nuestros valores de la siguiente forma:

	1	2	3	4	5	6	7	8
sufijo	8	4	5	6	7	8	8	9

El paso a seguir es calcular los valores para el arreglo *reubica*, el cual contiene los valores que se definieron al inicio de esta sección al igual que la idea para calcularlos. El algoritmo que los calcula se muestra a continuación.

```
calculaReubicacionesDeP(){

   int s, n = Pattern.length;
   int[] reubica = new int[ n+1 ];

   for( int k=1; k <= n; k ++ )
      reubica[ k ] = n+1;

   for( int k = n-1; k >= 0 ; k -- ){
      s = sufijo[ k+1 ];
      while( s <= n){
         if( P[k+1] == P[s] )
            break;
         reubica[s] = min( reubica[s], s-(k+1) );
         s = sufijo[s];
      }
   }

   int l = 1, mov = sufijo[ 0 ];
   while( mov <= m ) {
      for( int k = l; k <= mov ; k ++ )
         reubica[k] = min( reubica[k], mov );
```

```
        l = mov+1;
        mov = sufijo[mov];
    }
}
```

Listado 3.3: Cálculo de las reubicaciones para P.

Después de haber presentado los dos algoritmos que calculan los valores de los arreglos *sufijo* y *reubica*, podemos observar que el espacio en memoria requerido para almacenar dichos valores es $2n + 1$ y sin considerar que además nos falta calcular y almacenar los valores para el arreglo *saltoPorApareo*, por lo que el tiempo de ejecución y de almacenamiento para el algoritmo total aumenta; sin embargo, su cálculo lo podemos realizar en un solo recorrido del patrón como a continuación se presenta, disminuyendo el tiempo de ejecución. Más adelante veremos como ahorrar espacio en memoria.

```
calculaSufijosYReubicaciones(){

    int s, n = Pattern.length;
    int[] reubica = new int[ n+1 ];
    int[] sufijo = new int[ n+1 ];

    for( int k = 1; k <= n; k ++ )
        reubica[ k ] = n+1;

    sufijo[ n ] = n+1;

    for( int k = n-1; k >=0 ; k-- ){
        s = sufijo[ k+1 ];

        while( s <= n ){
            if( P[ k+1 ] == P[ s ] )
                break;
            reubica[ s ] = min( reubica[ s ], s-(k+1) );
            s = sufijo[ s ];
        }
        sufijo[ k ] = s-1;
    }

    int l = 1, mov = sufijo[0];

    while( mov <= n ){
        for( int k = l; k <= mov; k++ )
            reubica[k] = min( reubica[k], mov );
        l = mov+1;
```

```
        mov = sufijo[mov];
    }

    for( int k=1; k <= n; k++ )
        saltoPorApareo[k] = reubica[k] + ( n - k );
}
```

Listado 3.4: Cálculo de Sufijos y de Reubicaciones en una sóla iteración.

Algo que hay que notar en este último algoritmo y es que además de haber fusionado los dos anteriores, se ha introducido el cálculo de los valores para el arreglo *saltoPorApareo* dado en la Definición 3.3.1, que para lograrlo usa los valores almacenados en el arreglo *reubica*; este último es del mismo tamaño que el arreglo *saltoPorApareo*, además que su uso sólo es para el cálculo de los valores del anterior y sobre todo que no existe dependencia para el cálculo de los valores de ambos arreglos, por lo que una forma de disminuir la cantidad de almacenamiento del algoritmo completo es que cuando se calculan los valores de *reubica*, éstos sean almacenados en *saltoPorApareo* y entonces en el último ciclo del algoritmo, se irán almacenando los valores para el último arreglo usando el valor guardado anteriormente para la posición $i-$ésima, de tal forma que la última línea sería reemplazada por la siguiente:

```
        saltoPorApareo[k] += ( n - k );
```

Listado 3.5: Cálculo de los valores para "saltoPorApareo".

Reconsiderando el patrón *abaaba* que presentamos en la Figura 3.5, y aplicándole el algoritmo anterior obtenemos los siguientes valores:

P		**a**	**b**	**a**	**a**	**b**	**a**
sufijo	3	4	5	5	6	6	7
reubica		3	3	3	5	2	1
saltoPorApareo		8	7	6	7	3	1

Por último presentamos el algoritmo de Boyer-Moore que utiliza los valores almacenados en los arreglos: *saltoPorCarácter* y *saltoPorApareo*.

```
int exploracionBoyerMoore(){

    int apareo = -1;
    int j,k;//j recorre a T y k a P

    int n = Pattern.length;
    j = k = n;
```

```
    while( finTexto(T,j) == false )
        if( k < 1 ){
            apareo = j+1;
            break;
        }

        //el caso en que se van apareando P y T caracter a caracter
        if( T[j] == P[k] ){
            j--;
            k--;
        }
        //cuando ha surgido un no-apareo, se tiene que
        //reubicar el apuntador de j a T
        else{
             j += max( saltoPorCaracter[ T[j] ], saltoPorApareo[k] );
             k = n;
        }
        return apareo;
}
```

Listado 3.6: El algoritmo de Boyer-Moore utilizando los valores de *saltoPorCarácter* y de *saltoPorApareo*.

El comportamiento del algoritmo de Boyer-Moore depende del tamaño del alfabeto con el que se esté trabajando y la repetición de subcadenas dentro del patrón mismo.

Dos extensiones al problema de *apareamiento de patrones* frecuentemente usados son:

- encontrar todas las ocurrencias de P en T
- encontrar cualquier ocurrencia de P en un conjunto finito de ocurrencias en T

La implementación de los algoritmos que resuelven estos problemas difieren significativamente del algoritmo que hemos presentado, pues para resolver la primera extensión al encontrar una ocurrencia, se volvería aplicar el algoritmo teniendo en cuenta el texto actual como el texto original menos la parte que ya ha sido explorada (en la que se encontró la ocurrencia), y de esta forma continuar hasta que todo el texto sea explorado, mientras que la implementación al segundo problema es la que hemos estado trabajando, donde la ocurrencia del patrón que se encuentra es la primera.

De esta forma queda presentado el algoritmo de Boyer-Moore, considerando que existen aún gran cantidad de versiones que implementan distintas ideas y que incluso logran que el tiempo de ejecución sea mejor para fines prácticos, ya que la complejidad asintótica es la misma en todos los casos, cambiando las constantes involucradas.

Sin embargo, en nuestro interés por realizar búsquedas con gran rapidez, sin descuidar la eficiencia de implementación, seguiremos en la búsqueda de algoritmos que realicen esta tarea de la mejor forma posible para lograr obtener los mejores resultados en las tareas que involucren su aplicación; esta es la razón que origina el desarrollo del siguiente capítulo.

Capítulo 4

Árboles Sufijos

4.1. Introducción

En capítulos anteriores estudiamos algunos algoritmos que realizan búsquedas de patrones en un texto basados en la construcción de sufijos, los cuáles toman como base el algoritmo de Boyer-Moore. Sin embargo la literatura nos señala la existencia de algoritmos diferentes que también son usados en aplicaciones que requieren la solución de *apareamiento exacto* y donde el tiempo de ejecución disminuye; dichos algoritmos se basan en *árboles sufijos.*

En este capítulo estudiaremos árboles sufijos que, como veremos más adelante, también toman la idea de construcción de subcadenas sufijas para lograr una búsqueda más eficiente, con la cuál el tiempo de preprocesamiento, es decir la construcción del árbol, se realiza en $O(m)$, si es que S_1 tiene longitud m, y realiza la búsqueda de cualquier cadena S_2, de longitud n, en $O(n)$. Estas características dan una gran ventaja a la solución de problemas, pues S_1 y S_2 pueden adaptarse para representar al texto o al patrón según convenga y de esta forma lograr el mejor comportamiento de la aplicación. En este trabajo se supone que los problemas a solucionar son de tal forma que tenemos un texto fijo para el cuál se construye el árbol y entonces realizamos búsquedas sobre él de uno o más patrones, los cuáles no son todos conocidos al inicio de la ejecución del programa. Si esto no fuera así, entonces lo que conviene es solucionar el *problema de apareamiento exacto de un conjunto (exact set matching)*[1].

4.2. Definiciones básicas

Un *árbol sufijo* es una estructura de datos en la cual se representan todos los sufijos que pueden formarse de una cadena S, siguiendo ciertas reglas. Aún más, debido a la construcción de estos árboles, también se representan en él todas las subcadenas de la cadena dada. Para cada cadena S, de longitud m, el número de subcadenas diferentes que

[1] Consiste en que si tenemos un conjunto de patrones $\{P\}$ a buscar en un texto T, esto puede resolverse realizando la búsqueda simultánea de cada patrón y de esta manera se logra que T sea recorrido una sola vez.

se pueden formar son: $\sum_{i=1}^{m} i = m(m+1)/2$. Aún así el tiempo de creación del árbol sufijo es $O(m)$.

Más formalmente nuestra definición queda de la siguiente forma:

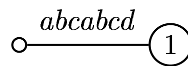
Definición 4.2.1. *Sea A un* árbol sufijo*, para un texto T, tal que $|T| = m$, entonces:*

- *A es un árbol dirigido con exactamente m hojas, las cuáles están numeradas de 1 a m.*
- *Cada vértice interno de A tiene al menos 2 hijos. Esto excluye a la raíz.*
- *Cada arista se encuentra etiquetada con una subcadena de T distinta de vacío. Si v es un vértice del árbol, y a_1, a_2 son dos aristas que salen de v, las etiquetas correspondientes a ellas no tienen prefijos que se apareen, lo cual sucede para cualquier par de aristas de v.*
- *Para cada hoja i, existe un camino C desde la raíz a esa hoja, tal que la concatenación de todas las etiquetas de las aristas pertenecientes a C nos da el sufijo de T que empieza en la posición i.*

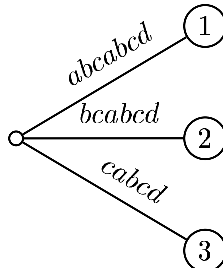
A continuación realizaremos la construcción de un árbol sufijo para el texto T y de esta forma podremos ayudarnos a apreciar las características anteriormente mencionadas.

Dado $T = abcabcd$, la construcción del árbol sufijo A correspondiente, es como sigue: la cadena se recorre de izquierda a derecha y en cada posición i, se construye el sufijo de T correspondiente a esa posición y se añade al árbol, en este proceso se cuida la aplicación correcta de los puntos listados en la definición.

- $i = 1$. Empezamos con la inserción del nodo raíz con un nodo hijo el cual su arista tiene como etiqueta la cadena completa, ya que es el sufijo más grande y se numera con 1.

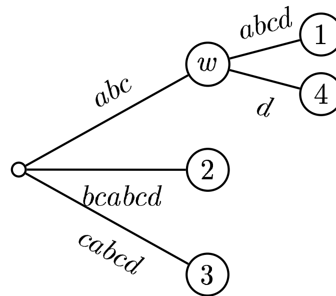


- $i = 2$. El sufijo correspondiente es *bcabcd*; como $T(1) \neq T(2)$ entonces creamos un nodo terminal distinto etiquetado con 2. De igual forma para $i = 3$, quedando como resultado final el árbol mostrado a continuación.

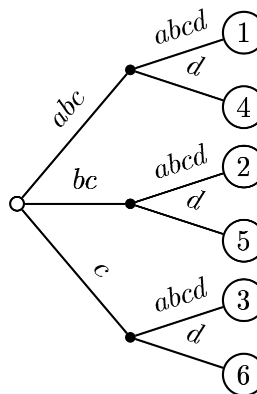


- $i = 4$, cuando llegamos a esta posición el sufijo formado es $abcd$ el cual tiene un prefijo (abc) que coincide con un prefijo de un sufijo anterior de T, el formado en la posición 1, por lo que tenemos que seguir el camino marcado por la etiqueta de esa arista correspondiente y en el momento de encontrar que un carácter no se aparea, entonces se inserta un nuevo nodo w; para marcar la diferencia de sufijos se añade una nueva hoja, en donde las etiquetas se forman así:

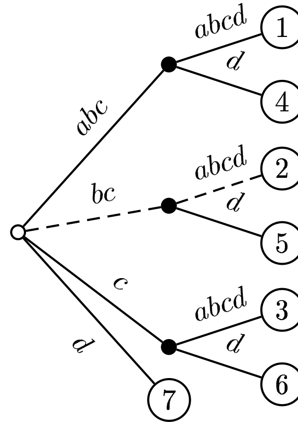
  1. Se agrega una arista del nodo w a la hoja que antes ya teníamos, y su etiqueta queda con la subcadena que no se apareó con el sufijo que actualmente estamos trabajando.
  2. Se agrega una nueva hoja con una arista de w hacia ella, y la etiqueta que le corresponde es la subcadena que no se apareó con el sufijo que coincide con el mismo prefijo y que completa el sufijo correspondiente a esa posición.



- $i = 5$ y 6. Se repite el proceso análogo que se usó para $i = 4$, resultando el siguiente árbol.



- $i = 7$; para esta posición el proceso es análogo que en las 3 primeras posiciones, resultando entonces el siguiente árbol sufijo correpondiente a T.



El último punto listado en la definición se observa a partir del esquema anterior. Por ejemplo, para la posición 2 se marca la ruta desde la raíz hasta la hoja que tiene este mismo número como etiqueta con líneas punteadas que indican el camino, donde al concatenar las etiquetas de las aristas nos da el sufijo a partir de $T(2)$, es decir $T[2..m] = bcabcd$ y ocurre lo mismo para cada una de las posiciones en T.

Si analizamos con cuidado notaremos que la construcción de A para T no causó ningún problema al seguir la última regla listada en la definición, ya que el último carácter d, no aparece en ninguna otra posición, lo que garantiza que ningún sufijo de T es algún prefijo de algún otro sufijo distinto; esto no ocurre para cadenas del tipo *abcabc*, en donde podemos observar que el sufijo *bc*, en la posición 5, es un prefijo del sufijo *bcabc*, en la posición 2; para garantizar que esto no pase en la construcción de nuestros árboles sufijos, se usará un *carácter extra (digamos \$)* que no pertenezca a Σ, para que al concatenarlo con el texto ($T\$$), nos garantice el cumplimiento de esta regla.

Otra observación importante es que en cada paso del proceso se forma un árbol correspondiente a esa posición, pero lo más relevante es que estos árboles no son independientes entre sí pues la construcción de uno depende del inmediato anterior construído, lo cual nos da la pauta para la implementación del algoritmo que desarrolla este proceso. Como un requerimiento, que se mencionó en la introducción a este capítulo, este preproceso tiene un orden de complejidad de $O(m)$.

Estas observaciones se formalizan de la siguiente manera:

Definición 4.2.2. *La* etiqueta de un nodo *w es la etiqueta de la arista que incide en él. La denotamos como etiq(v).*

Definición 4.2.3. *Para cualquier nodo v en un árbol sufijo,* la profundidad de la cadena *de v es el número de carácteres en la etiqueta de la arista de v.*

Definición 4.2.4. *La* etiqueta de un camino *desde la raíz hasta el extremo de un nodo, es la concatenación, en orden, de las etiquetas, que son subcadenas de T, de las aristas que pertenecen a ese camino. Como todo camino termina en un nodo v entonces esta etiqueta será denotada como etiq($C(v)$).*

Definición 4.2.5. *Un camino C que termina en la mitad de una arista (u, v) divide la etiqueta (u, v) en una posición k que depende de la subcadena que contiene. Cuando surge*

esta división, la etiqueta de C es ahora la etiqueta de u concatenada con el sufijo de la etiqueta de (u, v) que se forma a partir de la posición k.

Para ejemplificar estas definiciones observemos la Figura 4.1, donde se muestra el árbol sufijo para la cadena *aabc*.

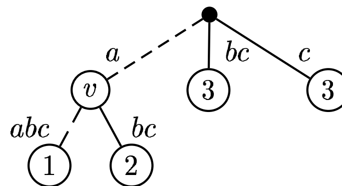


Figura 4.1: Árbol sufijo correspondiente a la cadena *aabc*. La cadena a es la etiqueta del vértice v. La etiqueta del camino que termina en 2 es *abc*. La cadena *aab* etiqueta un camino que termina en la arista $(v, 1)$.

4.3. Uso de árboles sufijos para apareamiento exacto

Ahora que sabemos la información que un árbol sufijo contiene así como la manera en que ésta está representada, lo que sigue es saber cómo se usan estos árboles para el problema de *apareamiento exacto*, que es nuestro objetivo inmediato; después se retomará su implementación. El problema fundamental que nos conduce a estudiar estas estructuras lo podemos expresar de la siguiente manera:

Dado un patrón P y un texto T, encontrar todas las ocurrencias de P en T en $O(n + m)$, con n y m la longitud de P y T respectivamente.

Este problema se resuelve con árboles sufijos de la siguiente manera:

- Construye el árbol sufijo correspondiente a T en O(m)
- Aparea los carácteres de P a lo largo del único camino C de T hasta que:
  a) *Se encuentre un no-apareo de P en C.* Este caso significa que P no ocurre en T.
  b) *No existan más apareos de P en C.* Cuando ocurre este caso, nos fijamos en el subárbol que se forma bajo el vértice que tiene por etiqueta la subcadena donde se quedó el apareo de P, y obtenemos los números de cada hoja perteneciente a ese árbol, los cuales nos indican las posiciones en donde P ocurre en T.

Una de las ideas clave que se aplica para el caso (b) es el hecho de notar que P ocurre en T empezando en la posición j si y sólo si P ocurre como prefijo de $T[j \dots m]$. Esto último sucede si y sólo si la cadena P etiqueta una parte inicial del camino desde la raíz

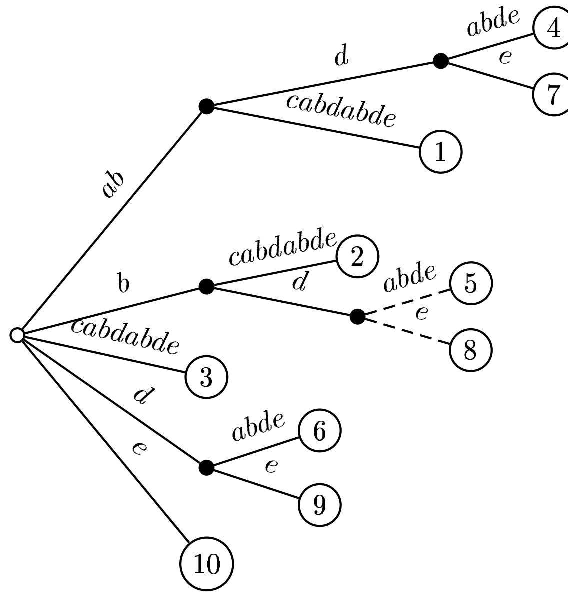


Figura 4.2: Árbol sufijo correspondiente a la cadena *abcabdabde*. Al realizar la búsqueda del patrón $P = bd$ empezamos a aparear los carácteres correspondientes y cuando se ha terminado la longitud de P sabemos que el patrón ocurre en T; para encontrar estas posiciones, se examina el subárbol marcado con líneas punteadas y se toman las etiquetas de las hojas que pertenecen a él, en este caso 5 y 8.

a la hoja j, es decir el camino inicial que sigue el algoritmo de apareamiento. Esto se muestra en la Figura 4.2.

El hecho de que al buscar una cadena sólo sea revisado el subárbol correspondiente a una arista del nodo raíz, tiene que ver con que en la construcción no se permite repeticiones de prefijos en dos aristas que *salen* de un mismo vértice, por lo que el camino de apareamiento es único, logrando que el tiempo para aparear un patrón sea proporcional con su longitud.

Por ejemplo en la Figura 4.2, si buscamos el patrón ab, éste aparece tres veces en el texto, en las posiciones 1, 4 y 7. El algoritmo encargado de encontrar estas posiciones sólo tiene que revisar el subárbol correspondiente al vértice donde concluyó el apareo y localizar las etiquetas de las hojas. Por lo tanto, todas las ocurrencias de P en T pueden ser encontradas en $O(n+m)$; aunque ésta es la misma complejidad que hemos estado obteniendo de algoritmos anteriores, la distribución es diferente, como lo habíamos mencionado en la introducción de este capítulo, ya que el tiempo de preprocesamiento en ellos es de $O(n)$ y de búsqueda es de $O(m)$, mientras que con árboles sufijos el tiempo de preprocesamiento es de $O(m)$ y el de búsqueda es de $O(n+k)$, donde k es el número de veces que P ocurre en T.

4.4. El algoritmo ingenuo que obtiene un árbol sufijo

Presentamos una forma sencilla e intuitiva de construir un árbol sufijo para una cadena dada. Ésta consiste en tomar la idea de construir el árbol A_{i+1} a partir del árbol A_i, y de esta manera obtener el árbol final inductivamente sobre i, donde i representa la posición en la cadena, y por lo tanto tiene un rango desde 1 hasta m, como lo vimos en el ejemplo de la subsección 4.1.

De esta forma el algoritmo, para el caso general, quedaría de la siguiente manera:

1. Caso Base: Construímos un *nodo raíz* con un nodo hoja que tenga como etiqueta a $S\$$, es decir, la cadena completa que es el sufijo correspondiente a la posición 1.

2. Hipótesis: Suponemos que el árbol A_i, para el sufijo $S[i \dots m]\$$ está construído y ahora con base en éste, construiremos el árbol correspondiente a $S[i+1 \dots m]\$$, es decir a A_{i+1}, para $2 \leq i \leq m$, de la siguiente forma:

    - Empezando en la raíz de A_i, encontrar el camino más largo, tal que un prefijo de la etiqueta se aparee con un prefijo de $S[i+1 \dots m]\$$. Este camino es único porque no existen dos aristas que salgan del nodo raíz y que sus etiquetas empiecen con el mismo carácter.
    - En algún punto no serán posibles más apareos, ya que ningún sufijo de $S\$$ es un prefijo de algún otro sufijo de $S\$$, lo cuál está garantizado por el uso de \$. Cuando se alcanza este punto, puede pasar que el siguiente carácter para aparear pertenezca a esa misma arista o sea parte de la etiqueta del nodo siguiente. Si está en la mitad de una arista (u, v), entonces divide esta arista en dos al insertar un nuevo nodo w exactamente entre el último carácter que se apareó y el carácter de la etiqueta que causó el no-apareo.
    - La nueva etiqueta para (u, w) se etiqueta con la subcadena de $S[i+1 \dots m]\$$ que se apareó con la etiqueta de (u, v), mientras que la etiqueta para la arista (w, v) es el resto de la etiqueta de (u, v).
    - A continuación, se crea un nuevo nodo-hoja etiquetado con $i+1$, el cual se conecta con la arista $(w, i+1)$, la cual se etiqueta con la parte del sufijo $S[i+1 \dots m]\$$ que no se apareó.
    - Con esto resulta que el árbol tiene un único camino desde la raíz a la hoja $i+1$ que tiene por etiqueta $S[i+1 \dots m]\$$

Este algoritmo ingenuo construye árboles sufijos en $O(m^2)$ para una cadena de longitud m, lo cual es un costo excesivo, habida cuenta de que existen algoritmos más eficientes para esta misma tarea.

Por ejemplo, algoritmos que realizan esta construcción en tiempo lineal tomando ideas semejantes a la que maneja el *algoritmo ingenuo*, como son el propuesto por Edward McCreight en 1976 el cual presenta ciertas desventajas [9] como es que el árbol tiene que ser construído en orden inverso, lo cual hace más difícil el uso para aplicaciones que tienen que ver con compresión de datos. Otro algoritmo muy conocido es el de Esko Ukkonen[4], que

aunque es menos intuitivo en su construcción, realiza mejoras al algoritmo que McCreight presentó, pues además de que la construcción la realiza recorriendo al texto de izquierda a derecha, por lo cual no es necesario conocer todo el texto al inicio de la ejecución, también sirve para compresión de datos. Ésta es la razón por la cuál en este trabajo se realiza el estudio del algoritmo de Esko Ukkonen.

4.5. El algoritmo de Esko Ukkonen que construye árboles sufijos

Una cadena de texto T con m carácteres, contiene m prefijos, los que se forman con cada iteración i-ésima, con $1 \leq i \leq m$. Cada uno de ellos contiene i sufijos, obteniendo un total de $m\,(m+1)/2$ sufijos. Para la construcción de A_i, el algoritmo comienza con un árbol vacío y durante la actualización correspondiente a cada i, se trabaja con los i sufijos correspondientes a cada i-prefijo; el algoritmo debe garantizar que cada uno de ellos se represente en A_i al terminar la iteración correspondiente. Para formalizar esta contrucción de sufijos, correspondientes a un prefijo dado, consideremos las siguientes definiciones:

Definición 4.5.1. *Sea S una cadena de longitud m. El prefijo $i-$ésimo de S, $p(i)$ es la subcadena $S[1 \ldots i]$; el sufijo $k-$ésimo de S, $s(k)$ es la subcadena $S[k \ldots m]$.*

Definición 4.5.2. *Dada una cadena S, s_{i_j} es el sufijo j-ésimo[2] correspondiente a $p(i)$, donde $1 \leq j \leq i$.*

Como ejemplo, consideremos la cadena abc; si $i = 1$, $p(1) = a$ y su único sufijo es ese carácter mismo; para $i = 2$ tenemos $p(2) = ab$ y los sufijos son $s_{2_1} = ab$ y $s_{2_2} = b$; finalmente cuando $i = 3$, $p(3) = abc$ y los sufijos son: $s_{3_1} = abc$, $s_{3_2} = bc$ y $s_{3_3} = c$.

Esta forma de insertar prefijos en un árbol sufijo hace que algunas veces perdamos un poco de intuición, con respecto a la forma en que se construye en el algoritmo ingenuo, pero en realidad toma sentido al recordar que la construcción del árbol correspondiente a una posición i no es independiente del inmediato anterior, y esto es porque el conjunto $\{s_{i_j}\}$ de un prefijo tampoco es independiente del conjunto correpondiente al prefijo inmediato anterior, pues difieren en un solo carácter.

Por ejemplo, en la cadena anterior vimos que los sufijos para el prefijo con $i = 2$ aparecen como prefijos para el siguiente conjunto de $\{s_{3_j}\}$; de esta forma s_{2_1} es prefijo de s_{3_1} y s_{2_2} es prefijo de s_{3_2}.

Una observación importante es que como estamos comparando los sufijos de dos prefijos contiguos, entonces s_{i_j} difiere de s_{i+1_j} por un solo carácter, de hecho el carácter que lo hace prefijo de este último. De lo anterior resulta entonces que al añadir los sufijos del prefijo actual, en realidad sólo se actualizan estas representaciones por ese carácter, pues los prefijos fueron ingresados cuando se construyó el árbol para $i-1$; lo único que se debe cuidar es la forma como se realizan estas actualizaciones, lo que revisaremos a continuación.

En la Figura 4.3 se muestran las primeras tres iteraciones de este algoritmo para el texto $T = abcabcd$.

[2] El rango de la j nos indica que el número total de sufijos para una cadena es la longitud de la misma.

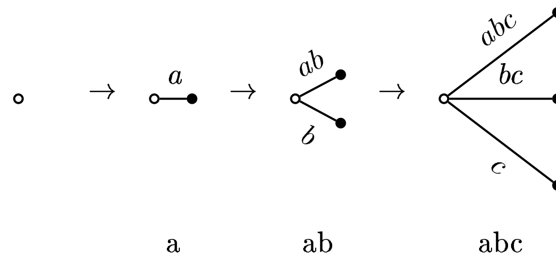


Figura 4.3: Las tres primeras iteraciones del algoritmo de Ukkonen para la construcción del árbol sufijo. Correspondiente a la cadena $T = abcabcd$.

4.5.1. Actualizaciones en un árbol sufijo

En esta sección presentamos la forma en que se realiza la implementación del algoritmo haciendo enfásis en los tipos de actualizaciones que se realizan sobre el árbol, por lo que presentamos algunas convenciones que nos ayudarán a explicar este proceso de mejor manera.

En un árbol sufijo tenemos tres tipos de nodos, los cuáles podemos definir así:

Definición 4.5.3. Nodos hoja. *Son los nodos que están en el último nivel del árbol, y no tienen hijos.*

Definición 4.5.4. Nodos explícitos. *Son nodos que representan un punto en el árbol en donde surgen dos o más caminos a través de aristas distintas.*

Definición 4.5.5. Nodos implícitos. *Son puntos en el árbol donde un prefijo aparece sobre una etiqueta de alguna arista pero no es toda ella. Éstos algunas veces se convierten en nodos explícitos, según la construcción del árbol.*

Ejemplos de las definiciones anteriores pueden apreciarse de mejor manera en la Figura 4.4, en donde 0 y 3 son *nodos explícitos*; *ab*, *abb*, *abbc*, son prefijos que están contenidos en la etiqueta de una arista sin completarla, por lo que representan *nodos implícitos.* Y los nodos 4 y 5 son nodos hojas, o simplemente hojas.

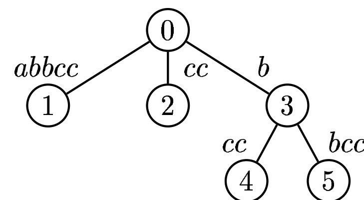


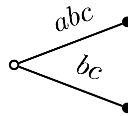
Figura 4.4: Árbol sufijo correspondiente a la cadena *abbcc*.

Tomemos como ejemplo la construcción del árbol para el texto $T = abccd$, en la cuál las primeras tres iteraciones coinciden con las mostradas en la Figura 4.3; y para ejemplificar

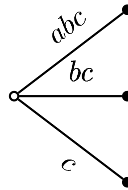
los tipos de actualizaciones, que enumeramos a continuación, observemos el paso de A_2 a A_3.

1. *La extensión de la etiqueta de una arista.* Consiste en la concatenación de un carácter a la etiqueta de una arista ya existente en el árbol.

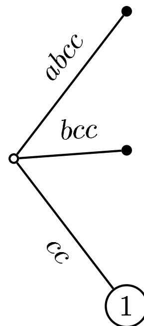
   Cuando $i = 3$, $p(3) = abc$ se diferencia de $p(2) = ab$ por el carácter c, por lo que al actualizar el árbol, lo que tenemos que hacer es añadirlo a las etiquetas de las aristas que ya existen en el árbol; de esta forma se va trabajando desde el sufijo más grande hasta el más pequeño. Este caso corresponde al siguiente tipo de actualización:



2. *Añadir una nueva hoja a un nodo explícito.* Cuando llegamos a la cadena vacía, implica añadir el sufijo c; como no existe una arista de la cual su etiqueta empiece con este carácter, entonces se añade una nueva hoja, la cual tendrá como etiqueta a c. Con esto obtenemos el árbol A_3.



3. *Actualización de un nodo implícito.* Al empezar a construir A_4, se realizan las actualizaciones de las aristas que ya existen, empezando por el sufijo más grande;

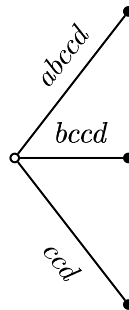


   cuando se llega a la cadena vacía, observamos que ya existe un nodo que tiene como etiqueta a c, por lo que no hay ningún tipo de trabajo que hacer y sin embargo la actualización existe, aunque el árbol A_4 sigue igual en apariencia.

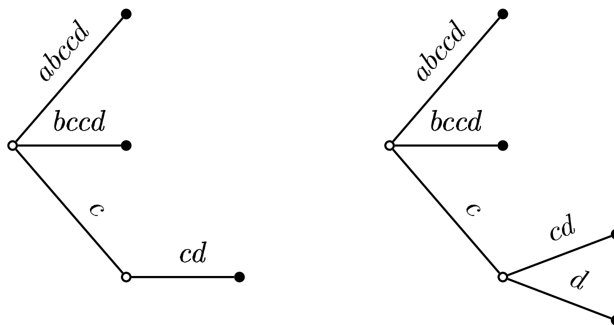
4. *División de una arista correspondiente a un nodo implícito.* Consiste en dividir una arista para agregar un nodo el cual conservará como hijo al nodo en el que incidía

la arista dividida, además de un nuevo nodo que representa al resto de la etiqueta del sufijo s_{i_j}

Este tipo de actualización la observamos al construir a A_5. Como lo hemos hecho, empezamos a actualizar los sufijos que se encuentran en A_4 empezando por el sufijo más grande hasta llegar al s_{4_4}, a los que sólo hemos concatenado el último carácter de $T[5] = d$ a las etiquetas de estas aristas; con esto el árbol se observa así:



Cuando llegamos al sufijo $s_{5_4} = cd$, entonces encontramos que existe una arista que sale del nodo raíz, la cual tiene como prefijo en su etiqueta al carácter c, por lo que el siguiente paso consiste en dividir esta arista e insertar un nuevo nodo. Luego es necesario añadir una nueva hoja para poder representar este último carácter de $s(5)$. El resultado de estos dos pasos se observa en el siguiente esquema.



Usando las reglas de actualización listadas anteriormente, una vez que el final del sufijo s_{i_j} de $T[1 \ldots i]$ ha sido encontrado en el árbol A_i, sólo toma tiempo constante aplicarlas y de esta manera garantizar que $s_{i_j}S(i+1)$ se encuentre en el árbol. Entonces la tarea a la que nos debemos enfocar es saber precisamente cómo localizar estos sufijos en el árbol para después realizar la actualización correspondiente.

4.5.2. Construcción del árbol sufijo

La forma más sencilla de encontrar el final de cualquier sufijo s_{i_j} en el árbol es caminar desde el nodo raíz e ir apareando los carácteres correpondientes; de esta forma el tiempo de ejecución para un sufijo s_{i_j} nos toma tiempo $O(i+1-j)$, por lo que como $p(i)$ tiene i sufijos, entonces construir A_{i+1} nos toma tiempo $O(i^2)$, de esto concluímos que A_m

ocupará $O(m^3)$ en tiempo de ejecución. Pero el Algoritmo de Ukkonen implementa algunas otras características para no sólo lograr el tiempo de construcción lineal con respecto a la longitud del texto, sino también para reducir espacio de almacenamiento.

4.5.3. Ligas sufijas

Definición 4.5.6. *Sea $V(A)$ el conjunto de vértices para el árbol A y sea v un nodo interno en $V(A)$ con $etiq(C(v)) = x\alpha$, en donde x denota un carácter y α es una subcadena, posiblemente vacía. Si existe $v_2 \neq v$ en $V(A)$ con $etiq(C(v_2)) = \alpha$, entonces un apuntador desde v a v_2 es una* liga sufija*, la cual denotaremos como $ls(v, v_2)$. Si α es vacía, entonces la liga sufija apunta al nodo raíz.*

Un ejemplo de esta definición, que incluye el caso de α vacía, se observa en la Figura 4.5, donde las ligas sufijas están con líneas punteadas.

Esta definición no garantiza que cada nodo interno de un árbol sufijo implícito tenga una *liga sufija*; sin embargo podemos afirmar el cumplimiento de algunas características que se presentan en cada fase de la construcción del árbol sufijo, en el siguiente lema y corolario.

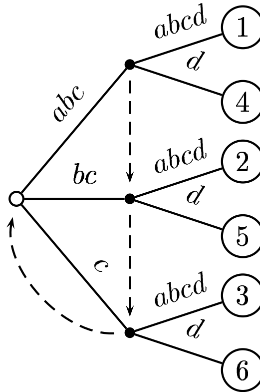


Figura 4.5: Ligas sufijas en un árbol sufijo. Cada nodo interno tiene una *liga sufija* a aquel nodo que tiene como etiqueta de camino a la correspondiente etiqueta de un sufijo como lo marca la definición.

Lema 4.5.1. *Si un nuevo nodo interno v con $etiq(C(v)) = x\alpha$ se agrega al árbol actual durante la fase $i+1$ (o sea en la construcción de A_{i+1} y la construcción del sufijo $s_{(i+1)_j}$), entonces pasa alguno de los siguientes casos:* a) *el camino etiquetado con α ya termina en un nodo interno del árbol actual o* b) *un nodo interno al final de la cadena α será creado, usando las reglas de actualización, al llegar al sufijo $s_{i+1_{j+1}}$ en la misma fase de A_{i+1}.*

Demostración. Un nodo interno se crea al trabajar con el sufijo s_{i+1_j}, sólo cuando se utiliza alguna de las reglas 2 ó 4. Esto significa que en la extensión j, el camino etiquetado $x\alpha$ continúa con algún otro carácter distinto de $S(i+1)$, digamos c. Así, en la extensión $j+$

1, existe un camino etiquetado con α en el árbol y éste ciertamente tiene una continuación con el carácter c, aunque posiblemente también con algunos otros carácteres. Existen dos casos por considerar:

a) el camino etiquetado con α continúa solo con el carácter c; ó

b) continúa con algún carácter adicional.

Cuando el carácter que le sigue a α sólo es c, la regla de actualización 4 crea un nodo v_2 al final del camino α. Cuando α continúa con dos carácteres diferentes, entonces ya debe existir un nodo v_2 al final del camino de α. En ambos casos el Lema se cumple. ❏

Corolario 4.5.1. *En el algoritmo de Ukkonen, cualquier nodo interno que se crea tiene asignada una* liga sufija *al final de la siguiente extensión,* $j+1$.

Demostración. Esta prueba se realiza por inducción sobre los prefijos de S. El caso base, cuando $i = 1$ se cumple, ya que A_1 no contiene nodos internos. Así es que supongamos que se cumple al final de la fase i y consideramos la siguiente fase $i+1$. Por el lema anterior, cuando se crea un nodo v en la extensión j, el nodo v_2 correspondiente a la *liga sufija* desde v será encontrada o creada en la extensión $j+1$. En la última extensión de cada fase, ningún nodo interno se crea, ya que esta extensión trabaja con el último carácter S(i+1), de esta forma, todas las ligas sufijas creadas desde nodos internos creados en la fase $i+1$ son conocidos al final de la fase, y el árbol A_{i+1} tiene todas sus ligas sufijas para cada nodo interno. ❏

4.5.4. Usando *ligas sufijas* para la construcción del árbol sufijo A

Una vez que hemos visto las fases en las que se construirán las ligas sufijas en el algoritmo, lo que sigue es ver la forma en que se utilizan y aprovechan para disminuir el tiempo de contrucción de A.

Recordemos que en la fase $i+1$ el algorimo localiza el sufijo $S[j \dots i]$ de $S[1 \dots i]$ en la extensión j, para $1 \leq j \leq i+1$, a través de algún camino que surge desde la raíz del árbol actual. Las ligas sufijas pueden acortar este camino y cada extensión. Las primeras dos extensiones, $j = 1$ y $j = 2$, para cualquier fase $i+1$ son las más fáciles de describir, por lo que empezaremos con ellas.

El final de la subcadena $S[1 \dots i]$ debe terminar en una hoja de A_i, ya que ésta es la subcadena más grande representada en el árbol. Resulta fácil encontrar el final de este sufijo, ya que así como los árboles son construídos, podemos guardar un apuntador a la hoja correspondiente a la cadena actual $S[1 \dots i]$, y de esta forma la actualización del árbol se realiza con la regla 1; por lo que la primera extensión de cualquier fase es especial y su actualización se realiza en tiempo constante.

Sea la cadena $S[1 \dots i]$ con etiqueta $x\alpha$, donde x es un carácter y α es una subcadena posiblemente vacía; sea v el nodo padre de la hoja 1 y γ la etiqueta de la arista que incide

en ella. El próximo paso del algoritmo debe encontrar el final de la cadena $S[2 \dots i] = \alpha$ en el árbol actual. El nodo v debe ser o el nodo raíz o un nodo interno de A_i. Si es la raíz, entonces para encontrar el final de α se debe bajar por el árbol, a partir de la raíz, siguiendo el camino etiquetado con α, de la misma forma que el "algoritmo ingenuo". Pero si v es un nodo interno, entonces, por el corolario anterior y ya que v fué creado en A_i, tiene una liga sufija al nodo v_2. Como v_2 tiene como etiqueta de su camino un prefijo de α, entonces el final de α debe terminar en el subárbol de v_2. De esta forma, en la búsqueda del final de α en el árbol actual, el algoritmo no necesita bajar el camino completo desde la raíz, sino empezar desde el nodo v_2. Éste es el objetivo principal de incluir ligas sufijas en el árbol. En resumen, el algoritmo busca a γ a partir de v_2.

Para extender cualquier otra cadena $S[j \dots i]$ a $S[j \dots i+1]$ para $j > 2$, se repite la misma idea general; empezar en v_k, el nodo al final de la cadena $S[j-1 \dots i]$; obtener el padre de v_k, que es la raíz o es un nodo interno v que tiene una liga sufija desde él; sea γ' la etiqueta de la arista de v_k; suponiendo que v no es la raíz, atravesamos la liga sufija desde v a v_2; después buscamos a γ' en el subárbol de v_2, pues γ' es un sufijo de $S[j \dots i]$; finalmente extendemos el sufijo de $S[j \dots i+1]$ de acuerdo a las reglas de actualización que antes estudiamos.

De esta manera hemos presentado el algoritmo completo que construye el árbol sufijo; sin embargo, el tiempo de ejecución no es de $O(|T|)$; para obtener esta cota se añaden al algoritmo algunas características que le permite realizar el proceso de forma más rápida. Antes de proceder con éstas características, introducimos la definición de altura de un nodo.

Definición 4.5.7. *Sea $v \in V(A)$. Definimos la altura de v como el número de vértices que existen en un camino C que va de v a la raíz.*

Procedemos a presentar algunas mejoras al método que construye el árbol.

- El truco de *cuenta y salta* con compresión de la etiqueta de las aristas; éste consiste en que la búsqueda de una cadena S, a partir de un vértice v, no dependa de su longitud, sino de la altura de v; la compresión de las etiquetas consiste en que cada una de ellas esté representada por un par de números que representen la posición de T donde el sufijo empieza y termina. Ilustremos de manera más precisa.

  Supongamos que estamos en la fase j de la construcción de A_{i+1}, es decir que cuando ésta termina, la cadena $T[j \dots i]T(i+1)$ estará representada en $A_{i+1}(j)$. Con el uso de "ligas sufijas", el algoritmo busca un sufijo S de $T[j \dots i]$, a partir de $v \in V(A_i)$. Sea $|S| = m$ y $u \in V(A_i)$ tal que $|etiq(u)| = n$ y $etiq(u)(1) == S(1)$.

  1. Caso $|etiq(u)| < |S|$. Entonces: $v = u, S = S[j+n \dots i]$ y sea u' tal que $|etiq(u')| = n'$ y $|etiq(u')| = n$; es decir se repite el proceso con sólo verificar el aparemiento del primer carácter de cada cadena y que se cumpla la condición que define a este caso.

  2. Caso $|etiq(u)| >= |S|$. Entonces: sabemos que S es un prefijo de $etiq(u)$, de longitud n.

En resumen, este proceso consiste en ir brincando entre nodos contiguos, donde cada uno pertenece a un camino. Si suponemos que conocer la longitud de una arista y obtener un carácter de S toma tiempo constante, entonces el proceso antes descrito es proporcional al número de nodos en un camino. De esta manera, al usar esta regla cualquier fase del algoritmo se realiza en $O(n)$; el problema es que tenemos un total de n fases por lo que la ejecución del algoritmo, aún usando este truco, es $O(n^2)$.

- El uso de la regla 3, en la fase j, implica la actualización implícita de $j+1$ hasta $i+1$.
- Cuando se crea un nodo hoja, ésta característica se mantiene durante toda la ejecución del algoritmo.

Este método, a diferencia del ABM, es considerado como dependiente de Σ. El uso de árboles sufijos no está restringido a la solución del problema de *apareamiento exacto*, pues también es útil para la solución de problemas más complicados, como es *apareamiento inexacto* entre otros; en [4] se muestra una lista de aplicaciones y en algunos casos el algoritmo mismo.

A pesar de que existe una gran cantidad de aplicaciones que dependen del uso de estas estructuras que hemos presentado, no debemos olvidar que existen algunas otras que pertenecen a la misma familia, como *suffix trie*[9] y *Patricia trie*[8] las cuáles otorgan cualidades especiales en la forma de construcción del árbol y que pueden ser aprovechadas según el problema que estemos resolviendo; básicamente estas diferencias tienen que ver con la forma en que los datos son representados en el árbol.

Capítulo 5

El ABM y árboles sufijos

5.1. Introducción

Hemos estudiado la forma en que el ABM y AS trabajan. Las ideas que ambos usan para lograr disminuir su tiempo de ejecución y de espacio tienen que ver con las ideas de preprocesar y analizar la cadena que se va a representar en la estructura correspondiente. Sin embargo, éstas tienen que ser modificadas dependiendo del caso particular de una aplicación para lograr su máxima eficiencia. Las características que obligan a modificar al algoritmo son tan diversas que no pueden ser enumeradas de forma precisa; sin embargo, podemos tomar sólo algunos casos para ejemplificar y estudiar estas variantes. Es éste nuestro objetivo en el presente capítulo, buscando enriquecer y darnos algunas ideas intuitivas de modificación a los algoritmos originales.

5.2. Apareamiento de Patrones en Dominio-Comprimido

La compresión consiste en reducir el tamaño de un conjunto de datos al eliminar información redundante de su estructura. Una de las técnicas generales de compresión consiste en asignar un patrón pequeño de bits a aquellos símbolos que ocurren con mayor frecuencia.

Existen dos opciones para realizar búsquedas de patrones en texto comprimido:

- Descompresión-búsqueda (*decompress-then-search*). El texto es descomprimido totalmente y enseguida se realiza la búsqueda del patrón. Este método, como todos los "ingenuos", tiene la ventaja de ser muy claro, pero tiene grandes costos en tiempo de ejecución (al realizar todo el proceso necesario para descomprimirlo) y de requerimientos de almacenamiento (al lograr descomprimirlo, se tiene que almacenar para después comenzar con el proceso de apareamiento de patrones).

- Apareamiento de patrones en dominio comprimido (*compressed-domain pattern matching*). El apareamiento del patrón se realiza sobre el texto sin descomprimirlo o descomprimido parcialmente. Las ventajas evidentemente son que el archivo de datos

es pequeño, por lo que el proceso de apareamiento toma menos tiempo para ejecutarse y sobre todo que se ahorra el tiempo del proceso de descompresión del archivo, además que el espacio requerido es menor. Sin embargo también tiene desventajas, la principal tiene que ver con que el proceso de compresión elimina información acerca de la estructura del archivo.

Existen muchas técnicas que se usan en apareamiento de patrones en dominios comprimidos, las cuáles consisten en comprimir el patrón y después buscarlo en el texto comprimido. Sin embargo, este método podría no funcionar en subcadenas que tienen diferentes representaciones dependiendo del contexto. Esto sucede cuando los límites en el texto comprimido no corresponden a los del texto original, como sucede en el código aritmético, o donde el archivo de entrada se codifica adaptivamente[10]. Algunas otras técnicas están basadas en los sistemas de compresión de la familia LZ(Lempel-Ziv), otros incluyen métodos para texto codificado con Huffman y otros sistemas están usando la transformación de Burrows-Wheeler[10], a la que denotaremos como TBW, como un paso de preprocesamiento para alcanzar un "buen" nivel de compresión. Algunos de los compresores más avanzados que usan esta transformación son la familia de *Bzip*[7, 3] y *Szip*[7], entre otros.

5.2.1. Transformación de Burrows-Wheeler

También se conoce como *ordenamiento por bloque*(*block-sorting*), y se usa como base en algunos algoritmos de *compresión sin pérdida* (*lossless*). Ésta consiste en realizar una permutación de una cadena de entrada S, permitiéndonos posteriormente su recuperación y al mismo tiempo ofrece la ventaja que sobre esta cadena se puede realizar una compresión más efectiva al lograr que los carácteres que aparecen en contextos similares estén agrupados en la salida.

La transformación ordena todos los carácteres del archivo de texto de entrada usando su *contexto* como la clave para realizar este proceso. El *contexto* se refiere al conjunto de carácteres que se encuentra inmediatamente antes o después de un carácter x a partir del cual se realizará el ordenamiento. En este algoritmo se usa el contexto "después" de x pues de esta forma se logran algunas ventajas – que explicaremos más adelante de forma breve –, para ciertas aplicaciones, entre las que se incluye la que es objeto de estudio de esta sección. Una característica importante es que el *contexto* permite determinar cuántos carácteres, a partir de x, son necesarios para contestar cualquier interrogante acerca de su orden.

Consideremos la cadena $S = emblema$ de longitud 7; si realizamos todas las rotaciones, de esta cadena, por un carácter, obtenemos una matriz M que se ve de la siguiente forma:

$$M: \begin{array}{ccccccc} \mathrm{e} & \mathrm{m} & \mathrm{b} & \mathrm{l} & \mathrm{e} & \mathrm{m} & \mathrm{a} \\ \mathrm{m} & \mathrm{b} & \mathrm{l} & \mathrm{e} & \mathrm{m} & \mathrm{a} & \mathrm{e} \\ \mathrm{b} & \mathrm{l} & \mathrm{e} & \mathrm{m} & \mathrm{a} & \mathrm{e} & \mathrm{m} \\ \mathrm{l} & \mathrm{e} & \mathrm{m} & \mathrm{a} & \mathrm{e} & \mathrm{m} & \mathrm{b} \\ \mathrm{e} & \mathrm{m} & \mathrm{a} & \mathrm{e} & \mathrm{m} & \mathrm{b} & \mathrm{l} \\ \mathrm{m} & \mathrm{a} & \mathrm{e} & \mathrm{m} & \mathrm{b} & \mathrm{l} & \mathrm{e} \\ \mathrm{a} & \mathrm{e} & \mathrm{m} & \mathrm{b} & \mathrm{l} & \mathrm{e} & \mathrm{m} \end{array}$$

Ahora nos disponemos a ordenar lexicográficamente los renglones de M obteniendo entonces a M', guardando además la última columna en un arreglo especial al que denotaremos L, como se muestra a continuación:

$$M' : \begin{array}{cccccccc} & & & & & & & \mathbf{L} \\ {\scriptstyle 1} & a & e & m & b & l & e & \mathbf{m} \\ {\scriptstyle 2} & b & l & e & m & a & e & \mathbf{m} \\ {\scriptstyle 3} & e & m & a & e & m & b & \mathbf{l} \\ {\scriptstyle 4} & \mathbf{e} & \mathbf{m} & \mathbf{b} & \mathbf{l} & \mathbf{e} & \mathbf{m} & \mathbf{a} \\ {\scriptstyle 5} & l & e & m & a & e & m & \mathbf{b} \\ {\scriptstyle 6} & m & a & e & m & b & l & \mathbf{e} \\ {\scriptstyle 7} & m & b & l & e & m & a & \mathbf{e} \end{array}$$

Entonces la transformación consiste en obtener la posición de M' en donde aparece la cadena original S, tenemos entonces que $TBW(S) = (L, 4)$. En resumen, al aplicar la transformación a S, obtenemos un par ordenado, donde el primer elemento es el arreglo que guarda las últimas posiciones del conjunto ordenado de rotaciones de S y el segundo es la posición I en la matriz donde ocurre S; este par nos permitirá, al realizar la transformación inversa, obtener la cadena original.

El proceso inverso es sencillo: usando L podemos obtener el primer carácter correspondiente al renglón i en M', guardando estos valores en un arreglo F. La forma de hacerlo es simplemente ordenar a L pues sabemos que los renglones en M' se encuentran ordenados. El resultado se puede constatar en el siguiente esquema:

$$M' : \begin{array}{ccccccc} \mathrm{F} & & & & & & \mathrm{L} \\ \mathbf{a} & e & m & b & l & e & \mathbf{m} \\ \mathbf{b} & l & e & m & a & e & \mathbf{m} \\ \mathbf{e} & m & a & e & m & b & \mathbf{l} \\ \mathbf{e} & m & b & l & e & m & \mathbf{a} \\ \mathbf{l} & e & m & a & e & m & \mathbf{b} \\ \mathbf{m} & a & e & m & b & l & \mathbf{e} \\ \mathbf{m} & b & l & e & m & a & \mathbf{e} \end{array}$$

Al tener a L y F con sus respectivos valores podemos definir una relación uno a uno entre ellos, la cual consiste en que dada la posición $i-$ésima de L existe una posicíon $j-$ésima en F que hace referencia al mismo carácter. El resultado de esta relación puede guardarse en un arreglo denominado R, el cual para nuestro ejemplo queda así:

$$R = [\begin{array}{ccccccc} {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \\ 5 & 6 & 7 & 1 & 3 & 4 & 2 \end{array}]$$

Para poder calcular estos valores nos auxiliaremos de las siguientes definiciones:

Definición 5.2.1. *Sea R una* relación uno a uno *entre L y F, tal que $R[j] = i$ si y sólo si $L[j]$ es la k ocurrencia de x en L y $F[i]$ es la k ocurrencia de x en F, con $x \in \Sigma$.*

De la definición anterior sabemos que $F[R[j]] = L[j]$, por la misma construcción de R, pues el carácter $L[i]$ precede cíclicamente a $F[i]$ en S; por lo tanto, sustituyendo $i = R[j]$ tenemos entonces que $L[R[j]]$ precede cíclicamente a $R[j]$ en S.

Retomando al índice I, sabemos que el último carácter de S es $L[I]$; de esta manera podemos usar al arreglo R para dar los predecesores de cada carácter.

Definición 5.2.2. *Considerando el arreglo R, sea R^i definida de la siguiente forma:*

- $R^0[x] = x$
- $R^{i+1}[x] = R[R^i[x]]$

Con estos valores procedemos a calcular a S dados I y L a partir de la siguiente definición:

Definición 5.2.3. $S[N-i] = L[R^i[I]]$, *para* $i = 0, \ldots, N-1$.

De esta forma se calcula a S de atrás hacia adelante[1]. En [10] lo realizan de forma inversa sólo que almacenan algunos valores extras en arreglos lineales, correspondientes a $|S|$.

Una vez que tenemos la forma de realizar el cálculo de TBW y la manera de recuperar nuestra cadena original, entonces lo natural es ver el porqué esta cadena L transformada nos permite obtener compresiones más eficientes y sencillas. Al tener un carácter x que comúnmente se repite en algún texto T, cuando se realicen las rotaciones de T y el ordenamiento de ellas, una gran cantidad de estos renglones podrían terminar en x, por lo que L contendrá a este carácter en muy distintas posiciones, posiblemente no consecutivas. Este mismo argumento se puede aplicar a todos los carácteres de las palabras, y de esta forma se pueden localizar regiones en L donde la probabilidad de diferencia de carácteres sea mínima. Esto conlleva a que la probabilidad de que un carácter x ocurra en esa región es grande, si es que x ocurre cerca de esa región, y muy baja en otro caso. Esta propiedad es la que tiene la mayor ventaja, ya que comúnmente se usa con el codificador *movimiento al frente*[3] (*move-to-front*) seguido por código de Huffman o código aritmético[3].

5.2.2. El ABM en texto de Dominio Comprimido con TBW

Este algoritmo nos permite realizar búsquedas descomprimiendo sólo parcialmente el texto comprimido. Para tal efecto, durante la etapa en que se aplica la transformación BW al texto de interés se calculan algunos otros valores que a continuación describiremos.

Consideremos H un arreglo que relaciona los carácteres del texto original con su respectiva posición en el arreglo F[2], es decir:

Definición 5.2.4. *Sea $H[i]$ una relación tal que $T[i] = F[H[i]]$, con $1 \leq i \leq n$*

[1] Aunque en [3] dice que pueden calcularlo de adelante hacia atrás, sin mostrar la manera de hacerlo.

[2] El arreglo que se representa en la matriz M' definida al inicio de esta sección y que ya sabemos cómo calcular.

Este arreglo[3] es útil al realizar el apareamiento de patrones con ABM para reportar las posiciones en T de las ocurrencias del patrón. La forma de calcular estos valores se basa en la idea que se refleja en el arreglo R. De esta forma, el arreglo H correspondiente a nuestra cadena de ejemplo queda de la siguiente forma:

$$H = [\; \overset{1}{3} \;\; \overset{2}{6} \;\; \overset{3}{2} \;\; \overset{4}{5} \;\; \overset{5}{4} \;\; \overset{6}{7} \;\; \overset{7}{1} \;]$$

En este algoritmo se usan las dos reglas principales de ABM, a) la *regla extendida del carácter erróneo* y b) la *regla de movimiento por un buen sufijo*. Esta última se usa de la misma manera en que se presenta en el Capítulo 1, pero se realizan algunas modificaciones a la primera que tienen que ver con que esta regla usa listas ligadas para almacenar las ocurrencias de cada carácter en la cadena preprocesada, y aunque esto ahorra espacio en memoria, la desventaja es la búsqueda del índice adecuado; para evitar este tiempo de búsqueda se usa una *técnica de dos pasos*, que consiste en que en el primer paso PP se calcula el número de ocurrencias de cada carácter y se distribuyen de forma ordenada en un arreglo de tamaño n. Retomando nuestro ejemplo, los valores quedan así:

$$\text{PP} = [\; \overset{1}{\text{a}} \;\; \overset{2}{\text{b}} \;\; \overset{3}{\text{e}} \;\; \overset{4}{\text{e}} \;\; \overset{5}{\text{l}} \;\; \overset{6}{\text{m}} \;\; \overset{7}{\text{m}} \;]$$

De esta manera, este arreglo está retomando la definición de F, que ya sabemos la manera de obtener los valores. El siguiente paso SP es obtener los índices en el patrón de cada carácter, quedándonos nuestro arreglo así:

	SP	PP		
1	7	a	→	a
2	3	b	→	b
3	1	e	→	e
4	5	e		
5	4	l	→	l
6	2	m	→	m
7	6	m		

Al calcular estos valores, resulta más eficiente el algoritmo que si se usan listas ligadas, aunque aún así usamos $O(n+|\Sigma|)$ en espacio para almacenar el resultado de estos cálculos.

Entonces, el algoritmo que realiza *búsquedas de patrones en texto de dominio comprimido* usando ABM depende de los arreglos F y H, siendo posible decodificar porciones del texto en una forma semialeatoria sin decodificar el texto completo; el código se presenta a continuación:

[3] H así definido también tiene una inversa explicada en [10], pero para fines de este algoritmo no es necesario mostrarla.

```
void aparTexDomComp(P,F,H){
    int sG = sB = 0;
    calSufijosBuenos( P );
    calCaracterErrExt( P );
    k = 1;
    while( k <= m-n+1 ){
        i = m;
        while( i > 0 && P[i] == F[ H[ k+i-1]] )
          i -= 1;
        if( i == 0 ){
          reportaPOcurreEn( k-n+1 );
          k += G[ 0 ];
        }else{
          sG = valorSufijoBueno();
          sB = valorCaracterErrExt();
          k += max( sG, sB );
        }
    }
}
```

Listado 5.1: Apareamiento de texto en Dominio Comprimido.

El algoritmo queda así usando los cambios para la *regla extendida del carácter erróneo*, la cuál obtiene su mayor eficiencia con alfabetos pequeños; entonces resulta tener la misma rapidez que los algoritmos usados en texto no comprimido[10]. Una observación importante es el hecho de usar valores almacenados en arreglos de igual longitud al texto original.

5.3. El algoritmo de árboles sufijos y su uso en TBW

En esta sección se refleja la importancia que tiene el *árbol de contexto* (*context tree*) que está implícito en TBW, y, sobre todo, la relación que existe entre este árbol y el árbol sufijo correspondiente a la cadena de entrada. La importancia de estudiar los árboles de contexto en esta transformación reside en el hecho de que se pueden explorar las características de la cadena de entrada identificando los puntos principales, los cuáles se usan en los métodos de compresión que finalmente sugieren una posible dirección hacia algoritmos eficientes.

La ventaja de usar TBW estriba en los requerimientos moderados de recursos computacionales, comparados con otros métodos de compresión con funcionamiento similar.

Normalmente la parte computacionalmente crítica de la transformación es el obtener a M' a partir del ordenamiento de los renglones de M, que presentamos en 5.1.1, para obtener a M'; una forma de hacer esto es: en M podemos sustituir los renglones por los sufijos correspondientes a cada posición $i-$ésima, de tal forma que en lugar de M' nos ocupamos de ordenar a todos los sufijos de la cadena de entrada. Para nuestro ejemplo tenemos el esquema de la Figura 5.1, el cual no tiene una representación matricial dado

que estamos representando sólo los sufijos de S; aun cuando la representación ya no es matricial, les ponemos los mismos nombres sólo para mostrar la relación entre éstos y los primeros resultados.

$M:$							
	e	m	b	l	e	m	a
	m	b	l	e	m	a	
	b	l	e	m	a		
	l	e	m	a			
	e	m	a				
	m	a					
	a						

$M':$							
	a						
	b	l	e	m	a		
	e	m	a				
	e	m	b	l	e	m	a
	l	e	m	a			
	m	a					
	m	b	l	e	m	a	

Figura 5.1: Representación de M y M'. A partir de los sufijos de la cadena de entrada.

Teorema 5.3.1. *Un* árbol sufijo lexicográfico[4] *para una cadena de longitud n puede ser construído en $O(n + t(n))$, donde $t(n)$ es el tiempo que toma ordenar n símbolos.*

Demostración. El árbol sufijo se construye de igual forma que un árbol sufijo común excepto por el momento en que se agrega un nodo u a un nodo v; ya que la posición en que un nodo se agregará debe calcularse, ésta puede saberse al realizar un revisión lineal al conjunto de nodos hijos que posee el nodo raíz, con lo cual el tiempo de ejecución se ve limitado por el tiempo de realizar este ordenamiento, pues ya sabemos que la construcción del árbol es lineal con respecto al tamaño de la entrada. ❏

El algoritmo que originalmente se utiliza para el ordenamiento de M tiene una complejidad en el tiempo de $O(n \log n)$, el cual se puede mejorar al usar un árbol sufijo (como se menciona en [3]), que se recorre en orden y se obtiene la secuencia ordenada de los valores que contienen las hojas; este paso puede realizarse de forma lineal con respecto a la longitud de entrada. De esta forma tenemos el siguiente corolario del Teorema 6.1. para TBW, con la modificación que acabamos de indicar.

Corolario 5.3.1. *La complejidad de TBW is $O(n + t(n))$, donde n es la longitud de la cadena de entrada y $t(n)$ el tiempo para ordenar n símbolos[6].*

5.4. La construcción del árbol sufijo

Como anteriormente habíamos mencionado, un árbol sufijo puede ser usado para producir la cadena que TBW origina, a la que denotamos como S'. La forma es la siguiente: el árbol se recorre de izquierda a derecha y por cada hoja encontrada, el símbolo que precede

[4]Es un árbol sufijo como ya lo conocemos, sólo que las etiquetas de las aristas de los nodos hijos de cada nodo interno y la raíz, están ordenadas.

a la posición correspondiente de S es dado como el próximo carácter de S'; sin embargo no sólo termina aquí la importancia de la información que el árbol posee, ya que ésta también representa las similitudes entre *contextos*. Las hojas del árbol corresponden a dichos contextos; el padre (el inmediato superior) de un par de hojas, nos da la similitud entre dos contextos al obtener la concatenación de las etiquetas de las aristas que pertenecen al camino desde la raíz a ese nodo.

5.4.1. Podando el árbol sufijo

Para mantener la cuenta de las frecuencias de cada símbolo para cada nodo interno se debe guardar una estadística acerca de las propiedades del contexto de la cadena. Ésta podemos realizarla, y quitamos todos los nodos internos que no representen algún cambio significativo en la distribución, comparado con su padre. Eventualmente para cadenas de longitud grande, el número de nodos internos converge hacia un número que refleja el número de estados en un modelo de árbol de la fuente.

Para realizar este proceso, se usa un algoritmo recursivo que poda al árbol sufijo empezando de abajo hacia arriba y de izquierda a derecha, el cual tiene la ventaja de ser simple y rápido y necesita poco espacio. En cada momento sólo es necesario guardar la cuenta de la frecuencia de los nodos sobre el camino desde la raíz al nodo que se procesa en un momento determinado. Éste limita los requerimientos de espacio a la altura del árbol por el tamaño del alfabeto; el espacio se reduce aún más si se eliminan todos los nodos que se encuentran más allá de una profundidad constante D. Este último proceso no afecta el resultado final pues es muy raro que los nodos más allá de una profundidad alrededor de 7 se mantengan[6]; sin embargo lo importante de este paso del algorimo es que el espacio requerido se reduce.

Comentarios Finales

Los algoritmos mostrados en este trabajo son de gran importancia, no sólo por el aprendizaje que su construcción conlleva, sino por la gran cantidad de problemas que con ellos se resuelven.

Para obtener soluciones óptimas aveces basta el uso de los algoritmos más básicos, como AI – un caso de esto, son los problemas que manipulan alfabetos pequeños –, sin embargo cuando las características del problema cambian los resultados pueden ser ineficientes; por tal motivo es importante buscar alternativas que brinden mejores resultados, lo que conlleva el estudio de algoritmos más sofisticados cuya implementación contiene un nivel de detalle mayor que los primeros. En el algoritmo de Boyer-Moore y el de construcción de árboles sufijos, el nivel de detalle, además del que conlleva cada algoritmo mismo, básicamente consiste en cuidar que las operaciones realizadas, como accesos a elementos, garanticen los tiempos que la teoría del algoritmo supone; de lo contrario se debe modificar el tiempo de ejecución del mismo.

Un elemento importante en este proceso de transición es el alfabeto que se manipula en un momento dado, pues al hablar de árboles sufijos, por ejemplo, la longitud del alfabeto influye en la elección de la estructura de datos que se elige para la representación de la cadena de interés.

Algo muy importante es que el estudio de cada algoritmo nos conduce al análisis de modificación necesario para adaptarlo a cada problema particular y resolverlo de la mejor manera posible, garantizando el tiempo de ejecución que la teoría determina, pero no sólo eso, sino que el estudio de algoritmos originales nos da la base para la creación de algunos más sofisticados que resuelvan el mismo problema pero que nos permitan obtener mejores resultados, en este sentido tenemos la influencia clara de el ABM en el concepto de *árboles sufijos*.

Por último sólo nos resta enfatizar que muchos problemas tienen soluciones más eficientes al utilizar alguna estructura de datos perteneciente a la misma familia, tal es el caso de aplicaciones en *genómica* y *proteómica* en donde se buscan patrones "similares", es decir, que sólo se aparee una subcadena o subcadenas del patrón de interés. La forma de resolver esto es utilizar un *árbol sufijo generalizado* (*generalized suffix tree*) en el cual se representa un subconjunto de cadenas y todos los sufijos de cada una de ellas [11, 12], en ésta última estructura vemos de nuevo la influencia de los algoritmos base.

Bibliografía

[1] S. Baase and A. Gelder. *Computer Algorithms –Introduction to Design and Analysis.* Reading: Addison-Wesley, 2000.

[2] R. S. Boyer and J. S. Moore. *A Fast String Searching Algorithm.* Communications of the ACM, pp. 762-772, 1977.

[3] M. Burrows and D. Wheeler. *A block sorting lossless data compression algorithm.* Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[4] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press. pp. 5-23, 89-107, 1997.

[5] D. E. Knuth, J. H. Morris and V. R. Pratt. *Fast Pattern Matching in Strings.* SIAM Journal on Computing. pp. 323-350, 1977.

[6] N. J. Larson. *Structures of String Matching and Data Compression.* PhD thesis, Department of Computer Science. Lund University, 1999.

[7] G. Manzini. *An Analysis of the Burrows-Wheeler Transform.* In Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms, pp. 669-677, 1999.

[8] D. R. Morrison. *PATRICIA–Practical Algorithm To Retrieve Information Coded in Alphanumeric.* Journal of the ACM. Volume 15, Issue 4. pp. 514-534. October 1968.

[9] M. Nelson. *Fast String Searching With Suffix Trees.* Dr. Dobb's Journal. pp. 115-119, August 1996.

[10] M. Powell. *Compressed-Domain Pattern Matching with the Burrows-Wheeler Transform.* Honours Project Report, Department Computer Science & Software Engineering, University of Canterbury, 2001.

[11] J. Tsong-Li et. al. *Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results*.* ACM SIGMOD. Volume 23, Issue 2. pp. 115-125. June 1994.

[12] J. Vilo. *Pattern Discovery from Biosequences.* Series of Publications A. Report A-2002-3. Department of Computer Science. University of Helsinki, Finland. November 2002.