UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

COMPRESIÓN DE DATOS MEDIANTE ÁRBOLES SUFIJO

TESIS DE LICENCIATURA
QUE PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN
P R E S E N T A:
RICARDO GARCÍA HERRERA

DIRECTOR DE TESIS:
Dr. Guillermo Fernández Anaya

México, D.F. 2002

COMPRESIÓN DE DATOS MEDIANTE ÁRBOLES SUFIJO

RESUMEN

Los algoritmos de compresión de texto se pueden dividir en dos categorías: Algoritmos de diccionario y algoritmos Markov. Los algoritmos de diccionario, desarrollados por Ziv y Lempel en 1977, reemplazan cadenas repetidas por apuntadores a un diccionario en memoria. Son los algoritmos más populares debido a su velocidad de procesamiento.

Los algoritmos Markov, *escannean* el texto caracter por caracter, formando modelos estadísticos que almacenan la probabilidad de ocurrencia de cada caracter en base a su ***contexto*** (es decir, los caracteres vistos anteriormente). Estos algoritmos requieren de más memoria y tiempo de procesamiento que los algoritmos de diccionario, pero son mucho más poderosos.

A los distintos caracteres que han ocurrido después de un contexto determinado se les denomina *predicciones* del contexto. Los contextos que sólo tienen una predicción se denominan *contextos deterministicos*, y los contextos con más de una predicción se denominan *contextos no-deterministicos*.

En esta Tesis analizamos el comportamiento de los contextos deterministicos en los algoritmos Markov, y se propone una técnica para modelar eficientemente el comportamiento de este tipo de contextos. Se proponen también diversas heuristicas de baja complejidad para el modelado de los contextos no-deterministicos.

Para el manejo de memoria se utiliza una estructura de árbol, llamada ***árbol sufijo***, donde los contextos toman forma de nodos, y las cadenas de caracteres toman forma de ramas o vértices que unen a dichos nodos. Esta estructura puede construirse en tiempo lineal sobre una ventana deslizante, con un algoritmo desarrollado en 1996 por N. Jesper Larsson, y aplicado a la compresión Markov, por Matt Timmermans en el 2000.

El resultado es un compresor comparable a PPMZ-2 (considerado estado-del-arte en la compresión Markov), con un requerimiento de memoria mucho más bajo, y un menor tiempo de ejecución.

DEDICATORIA

A Dios, que me dio la vida.

A mi papá y a mi mamá, que desde pequeño me han dado su amor.

A mi hermana, que siempre me ha apoyado en mis proyectos.

A toda mi familia.

A los integrantes del **Laboratorio de Investigación para el Desarrollo Académico** (grupo **LINDA**), sin cuya colaboración esta tesis no habría sido posible.

A mi director de tesis, Guillermo Fernández Anaya.

INDICE

1. INTRODUCCIÓN A LA COMPRESIÓN DE DATOS

1.1. Conceptos Básicos

Los algoritmos de compresión de datos se pueden dividir en dos ramas: ***Lossy*** (con pérdidas) y ***lossless*** (sin pérdidas).

Los algoritmos *lossy* conceden una cierta pérdida de precisión en los datos a comprimir, para ganar una mayor compresión y son usadas en datos de voz y de imágenes.

Los algoritmos *lossless* garantizan un duplicado exacto de los datos originales después de un ciclo de compresión/expansión y son utilizados en bases de datos, texto y programas ejecutables. Si los algoritmos interpretan la entrada como una cadena unidimensional de caracteres, se les denomina algoritmos de *compresión de texto* o de *compresión universal de datos*.

1.2. Contenido de información

Existe un límite para la compresión de cualquier archivo, que es igual a su contenido de información. La información de un caracter o **símbolo** en un archivo, depende del número de ocurrencias en el archivo de dicho símbolo, con respecto al tamaño del archivo.
Si un símbolo es muy frecuente, la información que contiene es muy poca. Si un símbolo es muy escaso, la información que contiene es mucha.

Definición:
Si tenemos M diferentes símbolos m1,m2,... con probabilidades de ocurrencia p1,p2,... supongamos que durante un largo periodo de transmisión (o en un mensaje largo), N símbolos han sido generados. Si N es muy grande, se espera que p1 N símbolos de m1, p2 N de m2, etc. hayan ocurrido en la secuencia.

La **información total** en dicha secuencia será

$$Itotal = p1N\log2(1/p1) + p2N\log2(1/p2) + \ldots$$

La información promedio por intervalo de símbolos, representada por H, será:

$$H = Itotal/M = p1\ \log2(1/p1)+p2\ \log2(1/p2) + \ldots = \Sigma\ p_k\ \log2\ (1/p_k)$$

A ésta se le denomina **entropía.** [7]

1.3. Modelado y Codificado

Según Neal, Cleary y Witten [1], para una compresión eficiente los algoritmos deben seguir el ***paradigma de modelado/codificado***.

De una entrada que consiste en una ***cadena de símbolos***, y un ***modelo estadístico***, el programa codificador produce una cadena de salida, que suele ser una versión comprimida de la entrada.

El decodificador, que debe tener acceso al mismo modelo, regenera la cadena de entrada exactamente a partir de la cadena codificada.

Los símbolos de entrada pueden pertenecer al conjunto de caracteres ASCII o algún alfabeto binario; la cadena codificada es una secuencia de bits.

El modelo es un modo de calcular, en cualquier contexto dado, la distribución de probabilidades del siguiente símbolo de entrada. Tanto el codificador como el decodificador deben tener acceso al mismo modelo, para que el decodificador produzca una salida idéntica a la cadena original. La compresión se logra al transmitir los símbolos más probables en menos bits que los símbolos menos probables.

El modelo estadístico puede ser actualizarse conforme se va leyendo el archivo. Si el modelo no se actualiza, se llamará ***estático***, y si se actualiza se llamará ***adaptivo***.

1.4. Codificadores de Entropía

Una vez obtenido el modelo, se requiere de un codificador que produzca una cadena de bits, basándose en la tabla de probabilidades. Los algoritmos diseñados para desempeñar esta función se denominan ***codificadores de entropía*** *(entropy coders)*. Los codificadores de entropía más utilizados son el codificador de Huffman y el aritmético.

1.4.1. Codificador Huffman

El codificador Huffman crea códigos cuya longitud en bits depende de las probabilidades. Los códigos con mayor probabilidad tienen una longitud menor. Los códigos Huffman poseen la característica de que pueden ser decodificados correctamente a pesar de ser de longitud variable. Esta característica se conoce como ***atributo de prefijo único***.

Los códigos se construyen mediante un árbol binario, donde los símbolos son las hojas de éste árbol.

El árbol se va construyendo de la siguiente forma:

Los dos nodos (símbolos) con las frecuencias (pesos) más bajas son localizados.

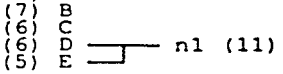
1. Un nodo padre se crea para estos dos nodos. Se le asigna un peso igual a la suma de los pesos de ambos hijos.
2. A la ruta del padre al hijo con la mayor frecuencia se le asigna un 1. Al otro se le asigna un 0.
3. Se repiten los pasos anteriores hasta que sólo quedan dos nodos sin padre, que se unen con el que se llamará nodo raíz.

Los códigos para cada símbolo corresponden a la ruta que va desde la raíz hasta el nodo que corresponde a dicho símbolo.

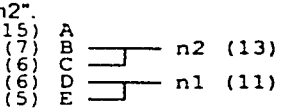
Ejemplo 1: Obtener los códigos Huffman para los sig. Símbolos.

Frecuencia	15	6	6	6	5
Símbolo	A	B	C	D	E

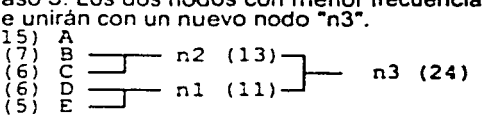
Paso 1. Los dos nodos con menor frecuencia son los nodos D y E.
Se crea un nodo "n1" que unirá a D y E.



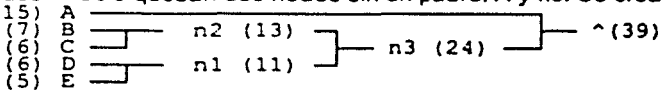
Paso 2. Los dos nodos con menor frecuencia son B y C. Se unirán con un nuevo nodo "n2".



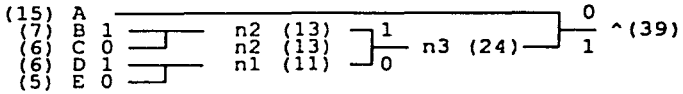
Paso 3. Los dos nodos con menor frecuencia son n1 y n2.
Se unirán con un nuevo nodo "n3".



Paso 4. Sólo quedan dos nodos sin un padre: A y n3. Se crea el nodo raíz que los unirá.



Paso 5. Cada nodo tiene sólo dos hijos.
Al hijo con mayor frecuencia se le asignará un 1, y al otro un 0.



Paso 6. Los códigos Huffman para cada símbolo se obtienen concatenando los dígitos binarios que van de la raíz hasta el símbolo.

A: 0 B: 111 C: 110 D: 101 E: 100

De este modo, el símbolo más frecuente (A) usa menos bits que los símbolos menos frecuentes (B,C,D y E). Por ejemplo, la cadena "AAABACAADABEAA", será codificada como:

A A A B A C A A D A B E A A
0 0 0 111 0 110 0 0 101 0 111 100 0 0

Esta cadena de salida requiere un total de 23 bits para codificar 14 símbolos: en promedio 1.64 bits por símbolo.

La desventaja de utilizar los códigos Huffman es que requieren como mínimo un bit por símbolo (a menos que se codifiquen cadenas de símbolos). El codificador aritmético, en contraste, Puede requerir apenas fracciones de bit por cada símbolo, sin necesidad de agrupar los símbolos.

1.4.2. Codificador Aritmético

El codificador aritmético reemplaza una cadena de símbolos consecutivos por un número sencillo de punto flotante. La salida de un proceso de código aritmético es un número menor que uno y mayor o igual a 0.

La idea es tener una línea de probabilidad de 0 a 1, y asignamos a cada símbolo un rango en esta línea basado en su probabilidad. Entre más probable sea un símbolo, mayor será el rango asignado a él.

Ejemplo 2. Si tenemos 3 símbolos, "a", "b", y "c", con número de ocurrencias 2, 1 y 1 respectivamente, tenemos que:

Símbolo	Probabilidad	Rango
a	0.5	[0.0.5)
b	0.25	[0.5.0.75)
c	0.25	[0.75.1.0)

Una vez codificado un símbolo, reducimos el rango o intervalo de todos los posibles símbolos siguientes, al rango del símbolo codificado. Si empezamos con un rango de [0,1) y codificamos una "a", nuestro rango se reducirá a [0,0.5) y se recalculará la tabla:

Símbolo	Probabilidad	Rango
a	0.25	[0,0.25)
b	0.125	[0.25,0.375)
c	0.125	[0.375,0.5)

El algoritmo para realizar el codificado aritmético es, *a grosso modo*, el siguiente:

```
Lim_inf = 0, Lim_sup = 1
Loop. Para todos los símbolos:
  sym = símbolo a codificar
  rango = lim_sup-lim_inf
  lim_sup = lim_inf + rango* rango_superior(sym)
  lim_inf = lim_inf + rango* rango_inferior(sym)
```

Donde lim_sup y lim_inf especifican el número de salida.

En la implementación, se usa aritmética de punto fijo, con determinados bits de precisión, usualmente 14 bits, donde el número hexadecimal 3FFF representa el 0.9999, y el 0000 representa el cero.

Los bits más significativos de los límites superior e inferior dejan de cambiar si son iguales. El valor único de estos bits se envía a la salida del decodificador para luego ser recorrido de los límites. Al recorrerse cada bit, el bit menos significativo asume un nuevo valor: 1 si el límite es superior, y 0 si el límite es inferior. De este modo se puede manejar una cantidad infinita de símbolos sin perder precisión.

```
Salida <--- Lim_sup [1001010010111] <-- 111111...
            Lim_inf [1001001100000] <-- 000000...
```

Conforme se van codificando más símbolos, la diferencia entre lim_sup y lim_inf es cada vez más pequeña, y puede ser insuficiente para codificar el siguiente símbolo si los bits más significativos de ambos límites difieren. A esta situación se le denomina ***underflow***, y se deben recalcular los límites mediante un proceso llamado ***renormalización***, que puede variar según la implementación del codificador. Cabe notar que la renormalización no modifica el intervalo, sino únicamente su representación.

Existen diversas variantes del codificador aritmético, algunas con patente, y otras en el dominio público.

2. MODELADO DE CONTEXTO FINITO

2.1. Introducción.

Uno de los modos más efectivos de predecir símbolos, y por ende, de obtener compresión en un texto, es basar las predicciones en las estadísticas de los símbolos más recientes. Por ejemplo, en el idioma español es muy probable que el texto "compresió" preceda a la letra "n", y que "ejem" preceda a la letra "p".

Definición.

En el modelado de contexto finito, el ***contexto*** de un símbolo dado es la secuencia de longitud finita de símbolos que le preceden a dicho símbolo. A la longitud de dicho contexto se le llama ***orden del contexto***.

Un contexto de orden **n**, tiene un ***sufijo propio***, o "***contexto sufijo***", de orden **n-1**, que es igual a dicho contexto sin el primer caracter.

Un "**contexto de orden 0**" contiene las ocurrencias de todos los símbolos vistos, independientemente de los símbolos que les preceden.

Un "**contexto de orden -1**" contiene una lista de todos los símbolos posibles, con conteo igual a 1 para cada símbolo.

Si queremos comprimir una cadena de texto, podemos ir fabricando una base de datos de todos los contextos vistos, con un determinado límite de longitud (que denominaremos ***orden del modelo***), para conocer las probabilidades de los símbolos que suceden a dichos contextos.

Como una regla general, si se tienen suficientes muestras, entre mayor sea el orden del modelo, más precisa será la predicción de un símbolo.

Un problema que surge cuando se tienen varios contextos para predecir un símbolo, es saber escoger qué contexto usar, o cómo combinar la información de los contextos de diversos órdenes. La solución más popular se presenta a continuación.

2.2. El Algoritmo PPM: "Prediction by Partial Match"

El algoritmo PPM fue desarrollado por Cleary y Witten en 1984 [2], y su primera implementación eficiente fue desarrollada por Alistair Moffat en 1990 [3]. El algoritmo PPM funciona del siguiente modo para cada símbolo a codificar:

Buscar el contexto de mayor orden correspondiente a la posición actual del archivo (o cadena) a comprimir.

1. Se verifica si el símbolo a codificar se encuentra en la lista de símbolos vistos en el contexto.
2. Si el símbolo se encuentra en la lista, se codifica utilizando el codificador aritmético.
3. Si no se encuentra, se codifica un escape, para luego buscar el contexto sufijo y regresar al paso 1.
4. Actualizar el modelo.

Para simplificar el algoritmo, se reserva un espacio para el escape, como un símbolo adicional. El algoritmo para decodificar es similar:

Buscar el contexto de mayor orden, según la posición del archivo decodificado.

1. Elaborar una tabla de probabilidades acumulada para el decodificador aritmético.
2. Decodificar el siguiente símbolo.
3. ¿Es un escape? Buscar el contexto sufijo y regresar al paso 2.
4. Añadir el símbolo decodificado a la salida y actualizar el modelo.

Ejemplo 3. Supongamos que nuestro compresor tiene la siguiente tabla estadística:

Orden	Contexto	Conteos individuales							Total
		n	t	u	d	b	c	a	
3	sio	9	2						11
2	io	10	2	3					15
1	o	15	6	18	25	1			65
0	(vacio)	30	15	41	68	24	72	32	282

Con esta tabla deseamos codificar la letra "d".

Tenemos un modelo de orden 3, donde el contexto más largo en la posición actual es "sio".

Según el algoritmo, se debe tener en cuenta la posibilidad de que ocurra un caracter no contenido en el contexto actual. Esto se hace reservando espacio para un símbolo adicional, que llamaremos "escape" (esc).

La probabilidad que se le debe asignar al escape es un factor crítico para obtener una buena compresión, lo cual se analizará más adelante. Para este ejemplo, seguiremos la implementación de Moffat, que asigna al escape un conteo igual al número de símbolos distintos de cada contexto.

De este modo, nuestra tabla de probabilidades queda como sigue:

Orden	Contexto	Conteos individuales							Subtotal	esc	TOTAL
		n	t	u	d	b	c	a			
3	sio	9	2						11	2	13
2	io	10	2	3					15	3	18
1	o	15	6	18	25	1			65	5	70
0	(vacio)	30	15	41	68	24	72	32	282	7	289

Como el codificador aritmético requiere una línea de probabilidad, debemos obtener una tabla de probabilidades acumuladas, donde el valor acumulado de un símbolo es igual a la suma de los conteos individuales de los símbolos que le preceden en la tabla. Por lo tanto el valor acumulado para el primer símbolo (en este caso, la letra "n"), es cero.

La tabla cumulativa será la siguiente:

Orden	Contexto	Conteos acumulados								TOTAL
		n	t	u	d	b	c	a	esc	
3	sio	0	9						11	13
2	io	0	10	12					15	18
1	o	0	15	21	39	64			65	70
0	(vacio)	0	30	45	86	154	178	250	282	289

Con la tabla cumulativa, cada número dentro del rango se traduce en un simbolo. Por ejemplo, para el contexto "sio", los números del 0 al 8 se traducen en una "n"; del 9 al 10 en una "t", y del 11 al 12 en un escape.

Ahora aplicaremos el algoritmo PPM para codificar la letra "d":

Comenzamos con el contexto de orden 3.
Se codifica un escape. Número codificado: 11/13
Se busca el contexto de orden 2.
Se codifica un escape. Número codificado: 15/18
Se busca el contexto de orden 1.
Se codifica la letra "d". Número codificado: 39/70
Entonces la cadena codificada sería: "11/13, 15/18, 39/70".

Para obtener la cantidad de bits utilizados, se suman los logaritmos de la probabilidad de cada caracter:

```
H = -log2(2/13) - log2(3/18) - log2(25/70)
H =   2.7       +   2.585    +   1.485      = 6.77 bits
```

Para decodificar:
Se busca el contexto de orden 3. Se decodifica el símbolo 11/13 (escape).
Se busca el contexto de orden 2. Se decodifica el símbolo 15/18 (escape).
Se busca el contexto de orden 1. Se decodifica el símbolo 25/65 (letra "d").

Para lograr una compresión más eficiente, los compresores PPM utilizan un proceso llamado ***exclusión***: Una vez que se codifica un escape, se debe eliminar el espacio reservado para los caracteres del contexto actual, debido a que no volverán a ocurrir.

Repetiremos el mismo ejemplo, utilizando exclusiones. Al usar exclusiones, se deben eliminar del nuevo contexto todos los símbolos que ya han sido "visitados". Para no alterar el modelo, las tablas cumulativas se suelen calcular cada vez que se visita un contexto.

Las tabla de probabilidad individual será:

Orden	Contexto	Conteos individuales							Subtotal	esc	TOTAL
		n	t	u	d	b	c	a			
3	sio	9	2						11	2	13
2	io	-	-	3					3	3	6
1	o	-	-	-	25	1			26	5	31
0	(vacio)	-	-	-	-	-	72	32	104	7	111

La tabla de probabilidad cumulativa será:

Orden	Contexto	Conteos acumulados								TOTAL
		n	t	u	d	b	c	a	esc	
3	sio	0	9						11	13
2	io	-	-	0					3	6
1	o	-	-	-	0	25			26	31
0	(vacío)	-	-	-	-	-	0	72	104	111

Los números a codificar serán:

11/13, 3/6, 0/31.

Y los bits utilizados serán:

```
H = -log2(2/13)   - log2(3/6) - log2(25/31)
  =     2.7       +   1.0     + 0.31
H = 4.01 bits (contra 6.77 bits sin exclusiones)
```

De este modo las exclusiones mejoran el desempeño del algoritmo PPM.

2.3. Exclusiones de Actualización (*Update Exclusions*)

Una característica importante del algoritmo PPM es que hay dos modos distintos de actualizar los contextos. Si se actualizan los conteos para todos los contextos, sin importar cuál fue el primer contexto en reconocer el caracter, se dice que el algoritmo utiliza actualizaciones completas (***full updates***). Si sólo se actualizan los contextos de orden mayor o igual al orden donde se encontró por primera vez el caracter a codificar, se dice que se utiliza exclusión de actualizaciones (***update exclusion***).

En el modelado de contexto finito, el usar *exclusiones de actualización* mejora la compresión significativamente. Sin embargo, tener conteos para ambos casos permite una mayor versatilidad al algoritmo de modelado, como se verá más adelante.

2.4. El problema de la frecuencia-cero. Los métodos de escape

Un problema fundamental en la compresión PPM, es la probabilidad que se le debe asignar al escape en un determinado contexto. Si la probabilidad asignada al escape es demasiado alta, se utilizarán demasiados bits para codificar los caracteres de cada contexto. Y si es demasiado baja, se utilizarán demasiados bits para codificar cada escape. Por lo que predecir

adecuadamente la probabilidad real de escape en un determinado contexto, es un problema fundamental para la compresión PPM.

Este problema es denominado "problema de la frecuencia cero", y puede expresarse de la siguiente forma:

Si en un contexto determinado se han visto n símbolos, de los cuales hay únicamente q símbolos distintos, ¿qué probabilidad hay de recibir un símbolo completamente nuevo?

El problema de la frecuencia cero fue analizado por primera vez por Laplace. Varias personas han dado distintas aproximaciones teóricas [3], pero para su aplicación en la compresión PPM, sólo se han visto soluciones empíricas, denominadas "métodos de escape".

Los dos métodos de escape más populares son los métodos denominados "C" y "D". El método C, propuesto en [2] y utilizado en el ejemplo anterior, establece que P(esc) = q / (q+n).

El método D, propuesto por Howard, establece que P(esc) = q / 2n.

Desafortunadamente, ambos métodos de escape sufren de una falla fundamental: La probabilidad de escape más alta en ambos casos es de 1/2.

Veamos el caso en el que q = n. Es decir, no ha habido un sólo símbolo repetido en un contexto.

Para el método C, P(esc) = q / (q + q) = q / 2q = 0.5
Para el método D, P(esc) = q / 2n = q / 2q = 0.5

Es decir, aunque el contexto tuviera una distribución uniforme (que resultaría en tener veinte o treinta caracteres distintos sin una sola repetición), la probabilidad de escape según los métodos C y D será igual a 0.5 como máximo. En otras palabras, en este tipo de contextos, la probabilidad de escape calculada será muy baja en comparación con la probabilidad de escape real.

Para resolver este problema, Charles Bloom en 1996 propuso la llamada estimación secundaria de escape, o ***SEE*** por sus siglas en inglés (Secondary Escape Estimation). Fue implementada como parte de su compresor PPMZ [13]. Ésta, entre otras técnicas, hicieron del compresor de Bloom el estándar sobre el cual basarse para el desarrollo de posteriores algoritmos de compresión PPM.

La ***SEE*** consiste en agrupar diversos contextos en base a sus últimos caracteres, su orden, y sus valores de q y n; y mantener estadísticas de "hit" (aciertos) y "miss" (escapes) para cada grupo distinto. De este modo, la probabilidad de escape asignada es determinada no arbitrariamente, sino por el mismo archivo a comprimir.

2.5. Modelado sin límite de orden: PPM* y PPMZ

En el modelo PPM, los contextos de mayor orden son más específicos. El problema es que los contextos cuyo orden sobrepasa cierto límite, son demasiado específicos para proveer estadísticas útiles: Sólo tendrán unas cuantas predicciones. Una primera aproximación para resolver el problema es la siguiente.

Definición.

Se dice que un contexto es ***determinístico*** cuando únicamente posee una predicción.

Ciertos estudios [4] han mostrado que los contextos deterministicos escapan con menos frecuencia que los contextos con dos o más caracteres.

Aprovechando esto, Bill Teahan [5] propone una variante del algoritmo PPM, pero sin límite de orden, para aprovechar la baja frecuencia de escape de los contextos deterministicos. Esta variante es denominada PPM*.

El método propuesto por Teahan, es escoger el contexto determinístico más corto, o en su defecto, el contexto más largo.

2.6. El problema del orden local

Un problema que surge con PPM*, es que si el contexto más largo no es determinístico, o si ocurre un escape en el contexto determinístico, es más probable que se requiera codificar de más escapes: Sigue existiendo un límite de orden, y distintos archivos pueden tener distintos valores de orden máximo óptimo. Incluso puede darse el caso donde el orden máximo óptimo cambie en un mismo archivo, por ejemplo, cuando el archivo es la concatenación de varios archivos de distintos tipos.

Para resolver este problema, Bloom propone un método de Estimación del Orden Local (LOE por sus siglas en inglés).

El método ***LOE*** toma el contexto no-determinístico más largo y obtiene una tasa de confianza basada en la probabilidad del símbolo más probable ó

MPS. Esta probabilidad está ponderada con la probabilidad de escape del contexto. Si la tasa de confianza del contexto sufijo es mayor a la del contexto actual, se disminuye el orden por 1, y se repite el proceso.

Además, Bloom incluye la estimación secundaria de escape en el cálculo de la tasa de confianza. La fórmula completa utilizada por Bloom es:

$$\text{Confianza} = 1.1 * [\ P(\text{MPS}) * (1-P(\text{esc}))\ ]$$

Esta fórmula fue obtenida en base a prueba y error, se desconoce por qué el factor de 1.1 mejora el desempeño del compresor. Según Bloom, los contextos más largos tienen un sesgo determinístico respecto al símbolo más probable.

En resumen, el algoritmo utilizado por Bloom es el siguiente:

Algoritmo PPMZ

1. Buscar el contexto más largo.
   Si no es determinístico, saltar al paso 3.
2. Aplicar predicción determinística por separado y (de)codificar el bit de escape.
   Si la predicción fue acertada, saltar al paso 5.
3. Obtener el contexto más conveniente según el algoritmo LOE
4. Aplicar algoritmo PPM estándar, utilizando SEE.
5. Actualizar modelo.

Cabe notar que el modelado para contextos determinísticos utilizado por Bloom, se basa en el contexto más largo (a diferencia de PPM* que usa el contexto determinístico más corto), y en la versión PPMZ2, utiliza una Estimación Secundaria. De esto se hablará en un capitulo posterior.

2.7. Comparación entre diversos compresores

Para comparar los diversos compresores de la familia PPM, se utiliza un conjunto estándar de archivos, denominado ***Calgary Corpus*** [6].

El Calgary Corpus consiste de 18 archivos, 4 del tipo binario y 14 de texto.

La versión 'compacta' del corpus, que es la utilizada para casi todas las comparaciones, excluye 4 archivos de texto (paper3,paper4,paper5 y paper6).

La medida de compresión utilizada son los ***bpc*** o "bits por caracter". Es decir, bits de salida por caracter de entrada. 8 bits por caracter significa que no hay compresión.

Se incluye el popular compresor ZIP (con compresión máxima) para comparación.

Los resultados para los diversos algoritmos son los siguientes:

ARCHIVO	tamaño original	ZIP	PPMC	PPM*	PPMZ v 9.1
bib (texto)	111,261	2.53	2.11	1.91	1.74
book1 (libro)	768,771	3.25	2.48	2.40	2.21
book2 (libro)	610,856	2.71	2.26	2.02	1.87
geo (binario)	102,400	5.37	4.78	4.83	4.64
news (texto)	377,109	3.07	2.65	2.42	2.24
obj1 (binario)	21,504	3.83	3.76	4.00	3.67
obj2 (binario)	246,814	2.63	2.69	2.43	2.23
paper1 (texto)	53,161	2.79	2.48	2.37	2.22
paper2 (texto)	82,199	2.88	2.45	2.36	2.21
pic (imagen)	513,216	0.82	1.09	0.85	0.79
progc (código)	39,611	2.69	2.49	2.40	2.25
progl (código)	71,646	1.80	1.90	1.67	1.47
progp (código)	49,379	1.82	1.84	1.62	1.48
trans (texto)	93,695	1.66	1.77	1.45	2.24
PROMEDIO		2.70	2.48	2.34	2.12

Tabla 1

Como podemos ver, el algoritmo de compresión PPM más simple (PPM, método de escape C) supera en todos los archivos (excepto uno) al compresor ZIP, lo cual prueba la superioridad de los algoritmos de compresión Markov con respecto a los algoritmos de diccionario.
PPM* mejora el promedio de PPMC por 0.14 bpc, y PPMZ es el mejor de los compresores comparados.

3. EL ÁRBOL SUFIJO

Uno de los problemas fundamentales en los algoritmos PPM es su alto requerimiento de memoria y su largo tiempo de procesamiento.

Si deseamos tener un byte en memoria para cada posible caracter de cada contexto, sólo necesitamos 256 bytes para un modelo de orden 0. Para un modelo de orden 1, necesitaremos 64 Kbytes. Para orden 2 necesitaremos 16.7 Mbytes. Hablar de orden 5 ó 6 es impensable sin una adecuada estructura de datos.

3.1. El árbol de contexto

La implementación de Moffat utiliza un árbol para manejar los contextos, de la siguiente forma. Cada contexto es representado como un nodo, donde sus hijos son las predicciones del contexto. Cada nodo tiene un contador, que representa el número de veces que se ha visto cada contexto. A ésta estructura se le denomina "***árbol de contexto***" o "***tria de contexto***" (del inglés ***context trie*** - no existe traducción exacta al español). También se le conoce como tria ***de sufijos***.

Ejemplo 4. El modelo PPM de orden 4 para la palabra "abracadabra" tiene la siguiente representación (sólo se muestran conteos mayor a uno):

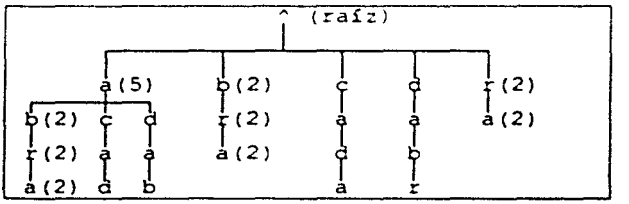


Fig. 1

Para orden 0, tenemos los siguientes símbolos con sus respectivos conteos:
a: 5 b: 2 c: 1 d: 1 r: 2

3.1.1. Otras aplicaciones.

Como se puede ver en la figura anterior, la tira de contexto contiene todos los sufijos de la cadena. Esto la hace una estructura útil para la búsqueda de patrones (por ejemplo palabras) en un archivo.

La búsqueda de patrones está relacionada con la compresión de datos, ya que la compresión de datos se basa principalmente en buscar y eliminar redundancia en un archivo. La familia de algoritmos Lempel-Ziv, utilizada en los compresores comerciales (ZIP, ARJ) se basa en la búsqueda de patrones para eliminar redundancia.

Supongamos que necesitamos buscar varias frases (unas 50, de 80 caracteres cada una) en una colección de 200 archivos de texto de 2 MBytes cada uno, para hacer una base de datos que relacione los textos con las distintas frases.

Para buscar una sola frase de M caracteres en un archivo de N caracteres necesitaremos realizar M*N comparaciones en el peor caso. Para este ejemplo serían 80*2,000,000 = 160 millones de comparaciones por búsqueda. Algoritmos avanzados de búsqueda (como el Boyer-Moore) necesitan únicamente M+N operaciones por búsqueda, que serían 2 millones y 80 comparaciones.

Si preprocesamos nuestra base de datos, formando un árbol de contexto para cada archivo, podemos simplificar el trabajo y necesitaríamos máximo 80 comparaciones por archivo, lo que nos permitiría obtener una respuesta en tiempo real. Es decir, necesitaríamos 200 * 80 * 50 = 400,000 comparaciones para realizar todas las búsquedas.

3.1.2. Desventajas de los árboles de contexto

Desafortunadamente, la complejidad tanto en tiempo como en memoria de los árboles de contexto hacen imposible la implementación práctica de un sistema de búsqueda.

En otras palabras, al utilizar este tipo de árbol sólo podemos mantener un número pequeño de contextos.

Conforme se va leyendo el archivo, la memoria utilizada crece debido a la continua adición de nodos al árbol. Para mantener bajos los requerimientos de memoria, Moffat reconstruye el árbol con los últimos 2048 caracteres del archivo cada vez que la memoria utilizada sobrepasa cierto límite.

La consecuencia de esto en la compresión PPM, es que se perderá una gran cantidad de información útil en la compresión, resultando en una compresión subóptima. En caso de archivos con un alfabeto grande (por ejemplo archivos binarios), se tendrá que reconstruir el árbol con demasiada frecuencia, lo que hará el algoritmo mucho más lento.

Otro problema es acceder a contextos de mayor orden (que debido a su longitud, no pueden ser almacenados en el árbol), se requeriría buscar el contexto en la cadena, pero esto tiene complejidad de $O(N^2)$. Es decir, si el archivo a comprimir tiene N bytes, se requerirán N^2 iteraciones para procesar todo el archivo.

Para resolver esto, Cleary y Teahan proponen incluir una liga a la cadena en los nodos "hoja" del árbol. Así, se puede acceder a los contextos de mayor orden, manteniendo a la vez un acceso rápido a los contextos de menor orden. Para obtener el contexto sufijo, se recurre a un apuntador en cada nodo (éste apuntador se denomina "***liga sufijo***").

Esta implementación, sin embargo, no resuelve el problema de la utilización de memoria. Al analizar el árbol de "abracadabra", notaremos que requiere de 23 nodos. De éstos, 20 tienen únicamente un hijo o son nodos hoja. Si podemos fusionar este tipo de nodos con sus nodos hijos, necesitaremos únicamente 8 nodos para todo el árbol, de los cuales 7 nodos serán nodos "hoja", y uno será nodo nodo "interno").

Así, todos los nodos que no sean "nodos hoja" tendrán como mínimo dos hijos. Si un árbol posee esta característica, se dice que tiene ***rutas comprimidas*** (***path compression***).

3.2. El Árbol Sufijo.

Definición.

Un árbol de contexto con rutas comprimidas, recibe el nombre de ***árbol sufijo.*** El árbol sufijo de una cadena ***T*** se caracteriza porque todos sus nodos representan un sufijo de dicha cadena. Es decir, que se puede encontrar cualquier subcadena dentro del árbol sufijo.

Ejemplo 5. El modelo PPM para la palabra "abracadabra" puede representarse mediante el siguiente árbol sufijo:

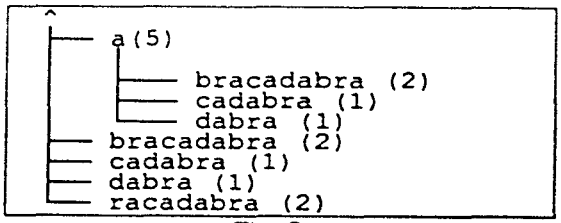
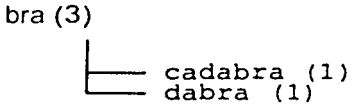


Fig. 2

Aquí necesitamos aclarar que aunque la cadena "bracadabra" sólo ocurre una vez, el nodo que la representa tiene conteo de 2. Esto sucede porque hasta ahora la palabra "bra" es a su vez prefijo de "bracadabra", y no ha aparecido en la cadena un caso donde no suceda esto. Por lo tanto no se pierde información al almacenar los sufijos "bra" y "bracadabra" en el mismo nodo.

Si por ejemplo después de "abracadabra" apareciera el texto "dabra", el nodo correspondiente a "bracadabra" se dividiría del siguiente modo:



Debido a la compresión de rutas, la memoria requerida por un árbol sufijo es lineal con respecto a la cadena que representa.

Teorema.
Un árbol sufijo de una cadena *T* de longitud N, tendrá como máximo 2N nodos.

Demostración:

Si ***T*** es de longitud N, el árbol sufijo correspondiente tendrá como máximo N nodos hoja, correspondientes a todos sus sufijos (incluyendo la cadena completa).

Debido a que cada nodo interno tiene como mínimo dos hijos, los nodos hoja tendrán en total un máximo de N/2 padres. Y éstos tendrán como máximo N/4 padres, y éstos N/8 padres, y así sucesivamente.

Por lo cual, el número de nodos internos máximo (incluida la raíz), será:

$$N/2 + N/4 + N/8 + \ldots = N\,(1/2 + 1/4 + 1/8 + \ldots) = N.$$

Y el número total de nodos presentes en un árbol sufijo será como máximo N nodos *hoja* + N nodos *internos* = 2N nodos.

$\therefore$ Q.E.D.

En otras palabras, el árbol sufijo de una cadena ***T*** requerirá un espacio lineal respecto a ***T***, siempre y cuando cada nodo requiera de un espacio constante en memoria. Esto se logra cuando representamos las cadenas que unen a los nodos como dos apuntadores a la entrada (que indican la posición inicial y final de la cadena). Así, cada nodo requerirá únicamente de dos enteros para representar una cadena de cualquier longitud.

Entonces podremos manejar un modelo de compresión PPM* requiriendo únicamente un espacio de memoria proporcional al tamaño de la entrada.

El árbol sufijo fue propuesto en 1976 por Edward McCreight [8]. Su algoritmo, desafortunadamente, tenía un problema. El árbol sufijo de una cadena se debía construir de derecha a izquierda, es decir, empezando por el final de la cadena.

Esto hacía imposible su implementación en la compresión de datos, ya que el algoritmo requiere conocer la cadena completa para construir el árbol.

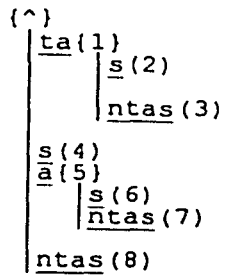
No fue sino hasta 1995 cuando Esko Ukkonen, de la universidad de Helsinki, diseñó un algoritmo para construir árboles sufijo en-línea [9]. Es la característica en-línea la que le permite al algoritmo de Ukkonen ser aplicado en la compresión de datos.

Para entender cómo se comporta el algoritmo de Ukkonen, es necesario explorar más a fondo la estructura de un árbol sufijo.

3.2.1. La estructura del árbol sufijo

Analicemos el árbol para la palabra "tantas".

Representaré los nodos con paréntesis ( ), y estarán numerados.
Los nodos internos los representaré con corchetes { }.
Los vértices que unen a los nodos se componen de las letras de la palabra.



Este árbol está compuesto por 8 nodos. Los nodos 1 y 5 son nodos internos. Los nodos 2,3,4,6,7 y 8 son nodos hoja.

Cada nodo corresponde a un contexto:

{1} ta	{5} a
(2) tas	(6) as
(3) tantas	(7) antas
(4) s	(8) ntas

Si tuviéramos un árbol sin comprimir rutas, cada sufijo (o contexto) estaría representado por un nodo. En cambio, en el árbol sufijo, sólo ciertos sufijos están representados por nodos.

3.2.2. Nomenclatura

Si el sufijo o contexto puede ser representado mediante un nodo, es decir, que no termina en medio de un vértice, se dirá que es un ***"nodo explícito"***. En un árbol sufijo, los nodos explícitos corresponden a los contextos no-deterministicos del modelo PPM.

La cadena representada por un nodo *u*, será denotada como *<u>*.

Si el nodo no contiene todos los caracteres de un vértice, se dirá que es un ***"nodo implícito"***. Los nodos implícitos corresponden a los contextos deterministicos en el modelo PPM.
Por ejemplo, el contexto "t" no está representado por ningún nodo, sino que se encuentra en el vértice que une la raiz con el nodo {1}. El contexto "tan", está contenido en el vértice que une la raiz con (3).

Por razones prácticas, no analizaremos el algoritmo original de Ukkonen, sino la variante expuesta por N. Jesper Larsson [10], que usaremos en nuestro modelo PPM.

Sea el árbol sufijo de una cadena ***T***:

Cada nodo *u* en el árbol guarda dos valores ***pos(u)*** y ***depth(u)***; donde *pos(u)* denota una posición en ***T*** donde es deletreada la *etiqueta* del vértice que va a *u*. Por lo tanto, la etiqueta de *(u,v)* es la cadena de longitud *depth(v)-depth(u)*, que comienza en la posición pos(v) de ***T***.

Por ***child(u,c) = v*** (donde *u* y *v* son nodos, y *c* es un caracter), denotamos que hay un vértice ***(u,v)*** cuya etiqueta comienza con *c*. A ***c***, lo llamamos ***caracter distintivo*** *de (u,v)*.

Para representar un contexto o subcadena α de *T*, definiremos ***point(α)*** como un vector ***(u,k,c)*** donde:

u es el nodo de máxima profundidad para el cual *<u>* es un prefijo de α.
k = $|\alpha|$ - |<u>|. Es decir, la longitud de la subcadena α menos la longitud de la cadena representada por el nodo *u*. Al valor de k lo llamaremos ***proyección***.
c es el (|<u>|+1)ésimo caracter de α, a menos que k=0, en cuyo caso el valor de c es irrelevante. (En estos casos el contexto correspondiente a α es no-deterministico).

Como un ejemplo de contexto **no deterministico** en el árbol sufijo mostrado anteriormente, tenemos al contexto "ta", cuya representación será (1,0,*).

Nota: * significa que el caracter c podrá tomar cualquier valor

OBSERVACIÓN.

En la representación de árbol sufijo de un modelo PPM, las predicciones de un contexto no-determinístico $(u,0,*)$, son los ***caracteres distintivos*** de los vértices que se originan en u.

Para un contexto determinístico (u,k,c), su única predicción corresponde al último caracter del contexto $(u,k+1,c)$.

En este caso, las predicciones del contexto "ta" en el modelo PPM corresponderán a "s" y "n", que son los ***caracteres distintivos*** de los nodos 2 y 3 respectivamente.

Como ejemplo de contexto determinístico, el contexto "tant", será representado por (1,2,"n"). Y su predicción corresponde al último caracter del contexto (1,3,"n"). Es decir, la letra "a".

El algoritmo de Ukkonen se basa en que, para una cadena T, de n caracteres de longitud, el árbol sufijo correspondiente se puede construir en n iteraciones: A partir del el árbol **Tree**[i] (correspondiente a la iteración i) y del caracter t[i+1], se construye el árbol **Tree**[i+1].

La posición correspondiente al sufijo α que se ha de actualizar, se denomina ***punto de inserción***. A la componente u del punto de inserción la llamaremos ***nodo de inserción***. Este nodo puede ser el nodo raiz.

El valor inicial del punto de inserción corresponde al sufijo más largo del árbol, que a su vez está presente en otra posición de la cadena T. A éste valor le llamaremos "***punto activo***" (active point).

3.3. Construcción del árbol sufijo

El algoritmo comienza añadiendo el nuevo símbolo a los sufijos del árbol, empezando por el de mayor longitud. En cada iteración, el punto de inserción se puede encontrar o en un nodo o en un vértice.

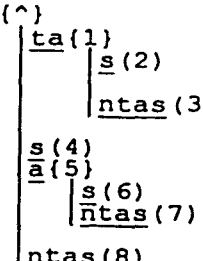
a. Si el punto de inserción se encuentra en un nodo.

En este caso, el punto de inserción tendrá varios hijos, y se busca el símbolo por añadir entre los símbolos iniciales de los vértices correspondientes. Si no se encuentra, se agrega un nuevo nodo hijo al nodo de inserción.

b. Si el punto de inserción se encuentra en un vértice.

En este otro caso el punto de inserción se localizará en un nodo implícito, y sólo tendrá un hijo. La predicción del punto de inserción (u,k,c) es el último caracter del contexto (u,k+1,c). Si el caracter por añadir difiere de la predicción, se dividirá el vértice en dos.

Ejemplo 6. Revisemos el árbol para la palabra "tantas".



Supongamos que el punto activo se localiza en (1,2,"n"). El contexto correspondiente será "tant", y la ruta correspondiente, recorre los nodos {^}, {1} y (3).

{^} ta {1} ntas (3)
_punto de inserción entre "t" y "a".

Si el símbolo correspondiente (en este caso "a") difiere del nuevo símbolo, convertiremos el punto de inserción en un nodo explícito. A este nodo añadiremos un hijo con el nuevo símbolo.

Supongamos que el símbolo a añadir fuera "o", tendremos que dividir el vértice que une a [1] y (3) en dos:

{^} ta {1} nt {9} as (3).

En este instante, el punto de inserción (correspondiente al contexto "tant") se convirtió en un nodo explícito. Ahora podremos añadir un nuevo nodo hijo.

{^} ta {1} nt {9} as (3)
| o (10).

Después de que se actualiza un sufijo del árbol, el punto de inserción apuntará a su sufijo inmediato - que en este caso sería (5,2,"n"), correspondiente al contexto "ant".

Una vez actualizado el punto de inserción, se repetirá el proceso hasta que no se requiera añadir ningún nodo. Esto significa que en el árbol de contexto equivalente (sin compresión de rutas), el símbolo nuevo se encontrará entre los hijos del punto de inserción. Y por lo tanto también se encontrará en los hijos de **todos sus sufijos.** Con lo cual, el árbol quedará actualizado.

A este punto se le denomina ***punto de término***, o ***endpoint.***

A *grosso modo*, el algoritmo para actualizar el árbol sufijo de una cadena *T*, a partir de un nuevo caracter *c*, es el siguiente:

```
Procedimiento Actualizar(c).

1.PuntoActivo = sufijo más largo que se puede encontrar en T.
  Punto de inserción (ins,proj,sym) = punto activo.

2.¿se encuentra c entre las predicciones del PuntoActivo?
3.Sí: punto de término encontrado. Ir al paso 7.
4.No: Si proj <> 0 (el punto activo está en un vértice),
  dividir el vértice creando un nodo u.
  Si proj == 0, u = ins.

5.Crear un nodo hoja s, hijo de u.
6.Buscar el sufijo del Punto de inserción. Ir al paso 2.
7.Actualizar T.
FIN.
```

Notemos que en el algoritmo expuesto no hay un procedimiento para actualizar los nodos hoja. Esto es porque si representamos las hojas como números, y el nodo hoja *leaf(j)* representará el sufijo que comienza en la posición *j* de *T*. De éste modo, al añadir un caracter a la cadena, los nodos hoja serán actualizados automáticamente. Esto implica que si s = *leaf(j)*, después de la iteración i, tendremos *depth(s) = i - j + 1*, y *pos(s) = j - depth(parent(s))*. Por lo tanto, los nodos hoja no requieren almacenar su posición en la cadena, ni su profundidad.

Al comenzar el algoritmo, antes de cualquier iteración, el punto activo estará localizado en el nodo raíz *(^, 0, *)*. El punto activo para la siguiente iteración se obtiene avanzando un caracter hacia abajo del punto de término (considerando que la raíz está en la parte superior del árbol).

En el algoritmo expuesto, vimos que el punto de inserción se representaba como *(ins, proj, sym)*. El caracter *sym* dependerá de la posición extrema derecha de la cadena incluida en el árbol. A esta posición la llamaremos ***front***. El punto de inserción será *(ins,proj,t[front-proj])*.

En la iteración *i*, tenemos que *front = i*, y al terminar la iteración i, *front = i+1*. La variable *proj* también se incrementará. Pero esto pone en riesgo la validez del punto de inserción, ya que podria pasar de ser un nodo implícito a un nodo explícito. Esto implica que también el nodo de inserción deberá ser actualizado.

Para este fin, recurriremos a la función *Canonize*.

```
Función Canonize (ins, proj, t[front-proj])

1.Mientras que proj > 0:
2.   r = child(ins,t[front-proj])
3.   d = depth(r) - depth(ins)
4.   Si r es una hoja, o proj < d, regresar r.
5.   De otro modo: proj = proj - d, ins = r
6.Regresar nil.
```

La función Canonize también se utiliza para buscar el sufijo del punto de inserción, como veremos a continuación.

3.3.1. La liga sufijo

Cada nodo interno *u* del árbol, contiene un apuntador a un nodo v (que representa su contexto sufijo). Este apuntador es llamado ***liga sufijo***. La liga sufijo de un nodo es actualizada en el proceso de creación del árbol sufijo.

La liga sufijo del nodo u se denota como *suf(u) = v*. La cadena representada por *v*, es el sufijo inmediato de la cadena representada por el nodo *u*. Es decir, <u> = c<v>, donde *c* es el primer caracter de <u>.

Para los casos donde u sea el nodo raíz, añadimos un nodo especial *nil*, y definimos *suf(raíz) = nil*, *parent(raíz) = nil*, *depth(nil) = -1*, y *child(nil,c) = raíz*, para cualquier caracter c.

Ahora supongamos que tenemos un contexto *(u,k,c)* que representa una cadena α, y que el vértice cuyo caracter distintivo es c, termina en un nodo v = *child(u,c)*. La cadena que une u con v, la representaremos como <v-u>.

Entonces podremos representar la cadena α como la concatenación de las cadenas <u> y <v-u>.

$$\alpha = <u><v\text{-}u>$$

A la cadena sufijo de α, la denotaremos como *suf*(α), tal que:

$\alpha = c\ suf(\alpha)$, donde c es el primer caracter de α.

Por lo tanto,

$$\alpha = c\ suf(<u>)\ <v\text{-}u>.$$
$$suf(\alpha) = suf(<u>)\ <v\text{-}u>$$
$$suf(\alpha) = <suf(u)>\ <v\text{-}u>$$

En otras palabras, para obtener el sufijo de α sólo necesitamos remover *c* de <u>, lo cual logramos al reemplazar *u* por *suf(u)*.

Luego buscamos la cadena <v-u> que parta del nodo u, por medio de la función *Canonize*. De este modo, el contexto sufijo de *(u,k,c)* puede ser obtenido con sólo dos pasos:

```
1. u = suf(u)
2. Canonize(u,k,c)
```

El algoritmo completo para la construcción del árbol sufijo de una cadena T se muestra en pseudocódigo C al final de este capítulo.

3.3.2. Construcción del árbol sufijo en una ventana deslizante

Desafortunadamente, las aplicaciones de un árbol sufijo en compresión de datos son limitadas si el tamaño del árbol depende del tamaño del archivo a comprimir. Considerando que el tamaño, en memoria de un árbol sufijo es de 10 bytes por caracter de entrada en la implementación más eficiente en memoria [11], un archivo de 5 MBytes de longitud requerirá 50 megabytes para ser comprimido (la implementación presentada en esta tesis requiere aproximadamente 29 bytes por caracter de entrada).

Por esta razón, Larsson modificó el algoritmo de Ukkonen para operar en una ventana deslizante.

Cada nodo hoja del árbol sufijo corresponde a una determinada posición en un buffer circular de tamaño *M*. Entonces el apuntador *front* será una variable módulo *M*. Además de este apuntador, se requerirá de un apuntador adicional *tail*, que apunte a la hoja más antigua del buffer. Es decir, que hay una hoja *v* tal que *<v>=T*, y que *v=leaf(tail)*.

Para mantener la ventana deslizante, se requerirá de una función llamada *AdvanceTail*, que eliminará el nodo hoja más viejo y hará las modificaciones necesarias para asegurar que el árbol sufijo permanezca válido.

Para mantener válido el árbol sufijo en una ventana, son necesarios 4 pasos:

1. Eliminar la hoja más antigua, manteniendo la compresión de rutas.
2. Mantener válidas las variables *ins* y *proj*.
3. Asegurar que el sufijo *T* sea el único en ser removido del árbol.
4. Mantener las etiquetas de todos los vértices válidas.

1) Eliminar la hoja más antigua, manteniendo la compresión de rutas.

Para eliminar la hoja más antigua, tenemos que $v = leaf(tail)$, $u = parent(v)$. Una vez removido el vértice (u,v). Si u tiene al menos dos hijos después de eliminar v, la operación ha sido completada. De no ser así, tenemos que s es el hijo restante de u; $w = parent(u)$. Reemplazamos los vértices (w,u) y (u,s) por un único vértice (w,s). El nodo u se desecha. Finalmente, si s no es hoja, se actualiza $pos(s)$ substrayéndole $depth(u)-depth(w)$.

2) Mantener válidas las variables *ins* y *proj*.

Esto se viola cuando el nodo a borrar u sea igual a ins. En este caso, asignamos a ins el valor de w, e incrementamos *proj* por $depth(u)-depth(v)$.

3) Asegurar que el sufijo *T* sea el único en ser removido del árbol.

Esto no se cumple cuando T tiene un sufijo α que también es un prefijo de T, y si point(α) se localiza en el eje de la hoja que será removida. Pero este caso, sólo se cumple cuando point(α) es el punto de inserción. Por lo que antes de borrar v, debemos llamar a Canonize para comprobar si el resultado es v. Si es así, reemplazamos v por leaf(front-$|\alpha|$).

4) Mantener las etiquetas de todos los vértices válidas.

Es decir, que $pos(u_i)>=$ tail para todos los nodos internos u_i.

Para lograr esto en tiempo constante por cada iteración de AdvanceTail(), Larsson recurre a un esquema de manejo de créditos.

Cuando se añade un nodo hoja al árbol, este nodo le manda un *crédito* a su nodo padre. Cada nodo interno u tiene un contador binario $cred(u)$ inicialmente 0. Si un nodo recibe un crédito cuando el contador es 0, cambia el conteo a 1. Cuando un nodo con contador 1 recibe un crédito, cambia el contador a 0 y le pasa a su padre el crédito que se originó en su hoja más reciente.

Si un nodo por remover tiene un crédito, dicho crédito se pasa a su padre. Al recibir un crédito, un nodo interno u actualiza $pos(u)$. Así todos los nodos internos u_i del árbol actualizan su posición pos(u_i), manteniendo la complejidad lineal del algoritmo.

Larsson explica a fondo en [10] cómo funciona el mecanismo de manejo de créditos.

Pseudocódigo del algoritmo de Ukkonen

```
Procedimiento arbol(T)
 1. i = 0; front = proj = 0; ins = raíz;
 2. While i < |T|
 3.    AdvanceFront(T[i]);

Función AdvanceFront(c)

 1. v = nil; endpoint = false;
 2. do
    {
 3.    r = Canonize(ins, proj, T[front-proj]);
 4.    if(r <> nil)
       {
 5.       if(T[pos(r)+proj] == T[front]) endpoint = true;
 6.       else
          {
 7.          u = nuevo nodo;
 8.          depth(u) = depth(ins) + proj;
 9.          pos(u) = front - proj;
10.          Crea vértices (ins,u) y (u,r);
             Updatecredits(ins,front-depth(u)); cred(u)=1;
11.          Remueve vértice (ins, r);
12.          If (r es nodo interno) pos(r) = pos(r) + proj;
          }
14.    }else
       {
15.       if(child(ins, T[front]) == nil) u = ins;
16.       else endpoint = true;
       }
17.    if(!endpoint)
       {
18.       s = leaf(front - depth(u));
19.       Crea vértice (u, s);
          Updatecredits(u, front-depth(u) );
20.       suf(v) = u; v = u; ins = suf(ins);
       }
21. }while(!endpoint);
22. suf(u) = ins;
23. proj++; front++;
```

NOTA: La función **Updatecredits** no aparece originalmente en el algoritmo de Ukkonen. Forma parte de la modificación de Larsson para operar en una ventana deslizante (ver sig. página)

Este algoritmo opera en tiempo constante, siempre y cuando la operación child(u,c) tome un tiempo constante. La prueba está demostrada en el artículo de

Ukkonen [9].

Modificación de Larsson al algoritmo de Ukkonen (pseudocódigo)

```
Función AdvanceTail()

 1.  r = Canonize(ins,proj,T[front-proj]);
 2.  u = parent(v); remover vértice (u,v);
 3.  if(v==r)
     {
 4.    k=front-(depth(ins)+proj);
 5.    crear vértice(ins,leaf(k));
 6.    updatecredits(ins,k);ins=suf(ins);
 7.  }else
 8.  if(u sólo tiene un hijo restante)
     {
 9.      w = parent(u);
10.      d = depth(u) - depth(w);
11.      if(u==ins)
12.      {  ins = w; proj = proj + d; }
13.      s=hijo restante de u
14.      if(cred(u))
15.         updatecredits(w,pos(s)-depth(u));
16.      Remover vértices (w,u) y (u,s);
17.      crear vértice (w,s);
18.      if(s es nodo interno)
19.        pos(s) = pos(s) - d;
20.      liberar nodo u de la memoria;
     }
21.  tail++;

Función updatecredits(u,k)

1. While(u!=raiz)
   {
2.   v = parent(u);
3.   k = max(k, pos(u) - depth(v) );
4.   pos(u) = k + depth(v);
5.   cred(u) = 1 - cred(u);
6.   if(cred(u)) return;
7.   u = v;
   }
```

El algoritmo de Larsson [10] mantiene un árbol sufijo en una ventana deslizante, en tiempo lineal con respecto al tamaño de la entrada.

4. IMPLEMENTACIÓN DE UN ÁRBOL SUFIJO EN GNU C++

A continuación se describen los detalles de un compresor PPM con árbol sufijo, en el lenguaje C++. La versión utilizada es "DJGPP", un puerto para MSDOS de GNU C++.

Para no empezar desde cero, la implementación está basada en el compresor BICOM de Matt Timmermans. El compresor BICOM consta de un modelo PPM de árbol sufijo (con el algoritmo de Larsson) combinado con un codificador aritmético biyectivo (de ahí el nombre BICOM, BIjective COMpressor) de rango (precisión) de 21 bits. El código de Timmermans está disponible en la red [12].

Para simplificar las rutinas, crearemos 4 tipos de entero sin signo: BYTE, WORD, U32 y U64, de 8,16, 32 y 64 bits respectivamente. El tipo BYTE será el utilizado para designar los símbolos o caracteres de entrada / salida.

```
typedef unsigned long long U64;
typedef unsigned long      U32;
typedef unsigned short     WORD;
typedef unsigned char      BYTE;
```

4.1. Diseño Básico del compresor

Recordando el paradigma modelado / codificado visto en la sección 1.3, BICOM desarrolla tres clases: Para el modelado, **`ArithmeticModel`**; para el codificado, **`ArithmeticEncoder`** y **`ArithmeticDecoder`**.

La clase ArithmeticModel (y sus derivadas) constan de dos funciones Encode y Decode, según se muestra a continuación:

```
class ArithmeticModel
{ public:
  virtual void Encode(BYTE symbol,ArithmeticEncoder *dest)=0;
  virtual BYTE Decode(ArithmeticDecoder *src)=0;
  x
};
```

La clase ArithmeticModel depende a su vez de ArithmeticEncoder y ArithmeticDecoder. Estas dos clases manejan la interfaz de entrada / salida requerida para comprimir un archivo. Las tres funciones fundamentales del codificador aritmético son Encode, GetP y Narrow.

```
  void ArithmeticEncoder::Encode(U32 p1, U32 psymlow, U32
psymhigh);
U32 ArithmeticDecoder::GetP(U32 p1);
void ArithmeticDecoder::Narrow(U32 p1, U32 psymlow,
     U32 psymhigh);
```

Para codificar un caracter de entrada, ArithmeticModel::Encode obtiene el rango correspondiente a dicho caracter, y lo codifica llamando a la función ArithmeticEncoder::Encode.

Para decodificar, `ArithmeticModel::Decode` llama a la función `ArithmeticDecoder::GetP` para obtener el rango del siguiente caracter (el cual se desconoce por el momento), y mediante el modelo estadístico regresa el caracter correspondiente. Una vez decodificado el caracter, se actualiza el rango del codificador aritmético por medio de la función `ArithmeticDecoder::Narrow`.

Las funciones `ArithmeticEncoder::Encode`, y `ArithmeticDecoder::Narrow` desempeñan la misma función: Actualizar el rango del codificador aritmético y leer o escribir un caracter del archivo sin comprimir, según sea necesario.

La función `ArithmeticDecoder::GetP` regresa el rango del codificador aritmético, normalizado según el parámetro P1.
En este caso, 0 <= GetP(P1) < P1.

Lo único necesario para implementar un compresor de árbol sufijo, es crear una clase derivada de *ArithmeticModel*. Timmermans crea la clase *SuffixTreeModel*, que en nuestra implementación dividiremos en dos.

La clase *SuffixTreeBasicModel* se encarga del manejo de memoria y la construcción del árbol sufijo.

El modelo PPM propiamente dicho es programado en la clase *SuffixTreeModel*, derivada a su vez de *SuffixTreeBasicModel*.

4.2. Estructuras básicas en el Árbol Sufijo

Un árbol sufijo está compuesto de vértices y nodos, y éstos se dividirán a su vez en nodos internos y nodos hoja.

Para diseñar las estructuras de datos más adecuadas, comenzaremos por analizar la estructura de un nodo hoja, por ser el más sencillo, ya que no es necesario almacenar información correspondiente a sus hijos.

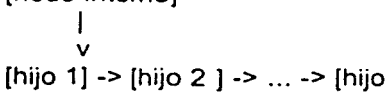
La ventana deslizante de Larsson está compuesta por M nodos hoja, cada uno con su símbolo correspondiente.

La información adicional que requerirá un nodo hoja será un apuntador al nodo padre (parent).

Además debemos considerar cómo un nodo dado su caracter distintivo. La función child() encontrada en la función Canonize() se encarga de esto, pero la implementación detallada depende del programador.

Larsson utilizó una tabla *hashing* para poder un nodo hijo en tiempo constante. Sin embargo, Timmermans encontró este método contraproducente debido a que en la fase de codificación, se debían todos los caracteres presentes en un contexto, es decir, todos los hijos de un nodo. El buscar todos los hijos con una tabla hashing requiere demasiado tiempo. Para resolver el problema se requería una lista ligada, por lo cual la tabla hashing resulta ser redundante.

Entonces para los hijos de un nodo, sólo necesitaremos un apuntador a su primer hijo. Llamaremos a este campo *firstchild*.



Adicionalmente, cada nodo necesita guardar el número de ocurrencias de un símbolo con respecto al nodo padre.

Por lo tanto, un nodo hoja requerirá de los siguientes campos:

1) El caracter correspondiente al buffer
2) El caracter distintivo del nodo
3) El conteo con *full update*
4) El conteo con *update exclusion*
5) El apuntador al nodo padre (requerido para el manejo de créditos)

En la segunda versión de BICOM (actualmente en desarrollo, no disponible al público), Timmermans utiliza arreglos llamados "vectores" que contienen ligas a todos los hijos de un nodo.

El tamaño de dichos vectores podrá variar, siendo siempre potencia de dos. Cuando a un nodo se le debe agregar un hijo, se verifica que el vector tenga espacio suficiente para alojar otra liga. De no ser así, se copia la información a un vector del doble de tamaño, liberando el vector original. Los detalles de implementación los veremos más adelante.

Si analizamos la estructura, vemos que estos vectores se componen en realidad de los *vértices* que surgen de un determinado nodo. Con lo cual reemplazaremos el apuntador firstchild, con un apuntador al vector de vértices.

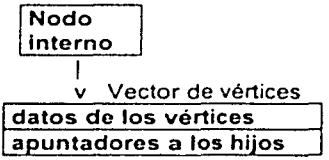


Fig. 3

El manejo de vértices posee varias ventajas sobre la lista ligada:

1) Tiempo de búsqueda de un nodo al construir el árbol sufijo.
En una lista ligada, para el nodo correspondiente a un símbolo, se requiere navegar por todos los nodos hasta encontrar el nodo con el símbolo correspondiente. Con los vectores, sólo se requerirá buscar en un **bloque contiguo** de memoria el símbolo correspondiente. En los procesadores con caché, se requerirá un ***caché miss*** por cada hijo con la lista ligada, a diferencia de un solo caché miss al utilizar los vectores.

2) Tiempo de acceso a los conteos en la fase de predicción
Para la codificación de nuestro algoritmo, necesitamos acceder a todos los nodos. En un contexto de N caracteres, una lista ligada requerirá N *caché misses* para armar la tabla cumulativa. Con los vectores, la información está disponible de antemano en el caché.

3) Aprovechamiento eficiente de la memoria.
Al usar vectores para los vértices, se coloca únicamente la información indispensable en los nodos, con lo cual es más factible el uso de *bit fields* para ahorrar memoria. También, al requerir menos memoria para los nodos hoja, se gana flexibilidad en el tamaño de la ventana deslizante.

4) Facilidad de programación y depuración.
El manejo de listas ligadas debe ser elaborado con sumo cuidado debido al uso de apuntadores y el requerimiento de un apuntador a NIL.

Por lo tanto se necesitarán 3 tipos de estructuras básicas:

El nodo interno (que llamaremos *INode*), el nodo hoja (*LNode*), y el vértice (*nodeinfo*). Debido a que los nodos (sean internos o no) deberán acceder a las mismas funciones, la estructura LNode la redefiniremos como la estructura base de un nodo. INode será una estructura basada en LNode.

Para asegurarnos que no se desperdicie espacio en memoria, agregamos `__attribute__((packed))` al definir cada estructura.

```
#define PACKED __attribute__((packed))
```

4.2.1. Nodo hoja

```
class LNode
{
  public:
  INode *parent;
  BYTE ipos;
  union {
    BYTE childcount;
    BYTE bufsym;
  };
} PACKED;
```

El campo *ipos* indica la posición interna del nodo en el vector de vértices, y la utilizamos para acelerar los accesos.

El campo bufsym se refiere al símbolo / caracter correspondiente al buffer circular. Como este campo sólo se aplica a los nodos hoja, aprovechamos este espacio en los nodos internos para definir el número de hijos de un nodo (*childcount*). Este valor será módulo 256. Como los nodos internos tienen al menos 2 hijos, el valor de *childcount* será igual al número de hijos menos 1. Así, un nodo con 256 hijos tendrá valor de 255, y un nodo con un hijo tendrá valor de 0.

La memoria requerida para cada nodo hoja será de 6 bytes.

4.2.2. Vértice

Cada vértice constará de 4 campos:
El apuntador al nodo en cuestión, `*p`
El símbolo distintivo del nodo, `sym`
Los contadores del modelo PPM: `cnt` para *full update* y `xcnt` para *update exclusion*. Cada contador estará limitado a 12 bits, con lo cual se requerirán 24 bits (3 bytes) para ambos.

```
class nodeinfo
{
  public:
  union
  {
    LNode *p;
    nodeinfo *nextfree;
  };
  BYTE sym;
  unsigned long int cnt:12;
  unsigned long int xcnt:12;
} PACKED;
```

El campo *nextfree que comparte memoria con *p, se utiliza para una lista ligada de los vectores no utilizados.
La memoria requerida para cada vértice será de 8 bytes.

4.2.3. Nodo interno

Es la más compleja de nuestras estructuras. Además de `*parent`, `*childcount` e `ipos`, el nodo interno requerirá de otros *campos esenciales*:
`*info`, un apuntador al vector de vértices.
`*suffix` es la liga sufijo del nodo.
`pos` y `depth`, los valores de posición y profundidad explicados en el algoritmo de Larsson.
`cred`, el bit de los créditos para AdvanceTail.
`tabsize` (table size) de 3 bits, indicará el tamaño en memoria del vector de vértices, según la siguiente tabla:

Valor	0	1	2	3	4	5	6	7
Nodos (max)	2	4	8	16	32	64	128	256

TABLA 2. Tamaño en memoria según *tabsize*

Para aminorar el tiempo de ejecución, añadiremos los siguientes *campos adicionales:*

- `cpl` y `xpl`, que indicarán el total de conteo para los hijos. Cada vez que se incremente el campo *cnt* de un nodo, se incrementará el campo cp1 de su padre. Al incrementar xcnt, se incrementará xp1 al nodo padre.
- `mps`, el símbolo más frecuente del nodo.
- `lastsym` es el símbolo correspondiente al último nodo en ser accedido.
- `localRLE` es el número de accesos consecutivos del último nodo. Sobre este campo hablaremos al explicar el algoritmo de predicción.

```
class INode:public Lnode
{
  public:
  union { INode *nextfree; nodeinfo *info; };
  INode *suffix;
  U32 pos:20; U32 depth:20;
  U32 cpl:24; U32 xpl:24;
  BYTE localRLE; BYTE mps; BYTE lastsym;
  unsigned cred:1; unsigned tabsize:3;
} PACKED;
```

La memoria requerida para cada INode será de 6 bytes (LNode) más 23 bytes: 29 bytes en total.

Notemos que para los nodos internos también se manejará una lista ligada de nodos libres. A continuación explicaremos el manejo y liberación de memoria en nuestro árbol sufijo.

4.3. Manejo de memoria en el árbol sufijo

El manejo de memoria en el árbol sufijo depende de las estructuras básicas: el nodo interno, el nodo hoja y los vectores. Para cada una de éstas estructuras existe una infraestructura de manejo de memoria. La más sencilla es la que corresponde a los nodos hoja, ya que son alojados de manera contigua en la ventana deslizante.

Como parámetro, establecemos de antemano que el máximo tamaño de la ventana deslizante será de 1 MByte. De este modo, tendremos un máximo de 2^{20} nodos hoja y 2^{20} nodos internos.

4.3.1. Nodos hoja

Los nodos hoja se localizan en una ventana de tamaño variable llamada `*m_leafbuf`, del tipo `LNode`. El tamaño de la ventana en nodos es igual al tamaño del archivo a comprimir. Para el descompresor, esta información está disponible en un encabezado de 24 bits al inicio del archivo comprimido. Una vez leído el tamaño de la ventana, éste se guarda en la estructura `SuffixTreeBasicModel`.

4.3.2. Nodos internos

Para alojar y desalojar nodos internos en memoria, utilizamos dos funciones: `NewI(U32 pos,U32 depth)` y `delI`, que son la interfaz entre el manejo del árbol sufijo y el manejo de memoria. El primer nodo en ser alojado es el nodo raíz, que llamaremos `m_rootnode`.

La memoria que contendrá todos los nodos internos está dividida en 256 bloques de memoria contigua, cada uno con 4096 nodos. Estos bloques son alojados y desalojados dinámicamente, según las necesidades del árbol. Para este fin, creamos un arreglo de 256 apuntadores del tipo *INode. Los apuntadores se inicializan con 0.

Cada elemento del arreglo apunta a uno de los bloques de 4096 nodos:

```
INode     *m_iblocks[256];
```

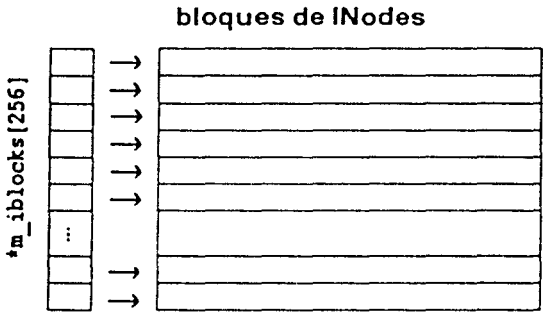


Fig. 4. Manejo de memoria para los nodos internos

Adicionalmente, para manejar los nodos que hayan sido borrados del árbol, utilizaremos una "lista libre", que es una lista ligada de los nodos libres. La lista libre se accesa con un apuntador del tipo *INode, llamado m_ifreelist. Larsson ya utilizaba la lista libre en su algoritmo de árbol sufijo.

```
INode     *m_ifreelist;
```

Cada vez que se borra un nodo `n`, el campo `*nextfree` de dicho nodo toma el valor de `m_ifreelist`, y `m_ifreelist` toma el valor de `n`. De este modo, `*m_ifreelist` siempre apuntará al último nodo borrado.

Al ser invocada, la función `NewI()` verifica que `m_ifreelist` no apunte a NULL (es decir, no tenga un valor de 0). Si apunta a NULL, significa que todos los nodos alojados están en uso y se requerirá alojar un nuevo nodo de la memoria.

Si `m_ifreelist` no es NULL, el apuntador del nuevo nodo tomará el valor de `m_ifreelist`, y `m_ifreelist` apuntará a `m_ifreelist->nextfree`.

Al usar la lista libre para reciclar los nodos borrados garantizamos que no habrá fragmentación en la memoria utilizada para los nodos, por lo cual el espacio de los bloques será utilizado con el 100% de eficiencia.

Si la lista libre está vacía, se requerirá alojar un nuevo nodo.

Para esto, usamos una variable entera llamada m_nexti, que apunta al número del nodo por ser alojado. Los 8 bits más significativos de m_nexti indican indican la posición en *y* del nodo siguiente, en otras palabras, el índice del apuntador en m_iblocks. Los 12 bits menos significativos indican la posición en x con respecto al bloque correspondiente.

La dirección absoluta del nuevo nodo *n* será:

```
n = &m_iblocks[y][x]
```

Si x == 0, entonces se deberá alojar en memoria un nuevo bloque:

```
m_iblocks[y]=(INode*)malloc(4096*sizeof(INode));
```

Si el resultado es 0, entonces no hay suficiente memoria para completar la operación y se reporta un error.

Una vez terminado el proceso de compresión, no hay necesidad de utilizar listas ligadas para liberar la memoria al sistema operativo; simplemente se liberan todos los elementos de m_iblocks que sean distintos de 0.

Este esquema de bloques permite el utilizar la memoria de los nodos en forma de una matriz bidimensional, con incrementos graduales en la utilización de la memoria.

4.3.3. Vértices

El manejo de vectores para los vértices es similar al del manejo de los nodos. La diferencia es que hay 8 diferentes listas ligadas según el tamaño de los vectores (2,4,8,16,32,64,128 y 256 elementos).

Las variables y funciones a usar para el manejo de los vectores son las siguientes:

```
U32      m_nextinf;
/* m_nextinf es análoga a m_nexti. Indica el número correspondiente
a la línea de vectores. El valor máximo válido es 255.
/*

nodeinfo *vecfree[8]; // análoga a *m_ifreelist
nodeinfo *infblocks[256]; // análoga a m_iblocks

nodeinfo **newtable(BYTE numchildren); // análoga a newI
void freetable(nodeinfo *table,BYTE tabsize); // a delI

nodeinfo *newvecrow();

/* newvecrow() aloja una línea completa de vectores a partir de la
memoria libre. Después de alojar los vectores, los organiza
formando una lista ligada para poder ser usados en newtable(). El
campo *nextfree del último vector de la tabla apuntará a NULL.
*/

nodeinfo *splitvec(BYTE idx);
```

Hay que notar que empezamos teniendo únicamente vectores de tamaño 256, y cuando se requiere, se van dividiendo estos vectores conforme sea necesario. Para dividir un vector, recurriremos a `splitvec()`. La función `splitvec` se encarga de dividir en dos el primer vector libre (cuyo tamaño especificamos en el parámetro idx), manejando automáticamente las listas libres correspondientes.

Las funciones splitvec y newtable se muestran a continuación:

```
nodeinfo *splitvec(BYTE idx)
{
  vecfree[idx-1]=vecfree[idx];
  vecfree[idx]=vecfree[idx]->nextfree;
  idx--;
  (vecfree[idx]+(2 << idx))->nextfree=0;
  vecfree[idx]->nextfree=vecfree[idx]+(2 << idx);
  return vecfree[idx];
}
nodeinfo **newtable(BYTE numchildren)
{
  int curidx,nextidx;
  curidx=sizeidx[numchildren];
```

```
// sizeidx es una referencia al tamaño de la tabla según
// el número de hijos.
  for(nextidx=curidx;nextidx<maxtab &&
  !vecfree[nextidx];nextidx++);
// si no hay una tabla de tamaño igual a curidx, el loop busca la
// tabla de tamaño inmediato superior, hasta encontrar
// la primera tabla disponible para irla dividiendo.

// debemos comprobar que haya un vector libre. Si no lo hay,
tratamos de obtenerlo de la memoria.
  if(nextidx==maxtab && !vecfree[maxtab])
  { if(!(vecfree[maxtab]=newvecrow())) return 0; }
  while(nextidx>curidx)
  {
    splitvec(nextidx);// dividimos los vectores conforme
    nextidx--;        // sea necesario
  }
  return &vecfree[curidx];
}
```

Hay que notar que no hay una función inversa a `splitvec()`. La razón es que perdería demasiado tiempo de procesamiento al fusionar dos vectores pequeños, para posteriormente tener que dividir el vector resultante. Así, la función `freetable()` sólo se encarga de manejar la lista libre correspondiente.

```
void freetable(nodeinfo *table,BYTE tabsize)
{
  nodeinfo **idx;
  if(tabsize>maxtab) tabsize=maxtab;
  idx=&vecfree[tabsize];
  table->nextfree=(*idx);
  (*idx)=table;
}
```

La función `delI` para liberar un nodo, es similar:

```
void delI(INode *n)
{
  if(!n) return;
  if(n->info)
    freetable(n->info,n->tabsize);
  n->nextfree=m_ifreelist;
  m_ifreelist=n;
}
```

Notemos que al crear un nodo interno también creamos su vector de vértices correspondiente. Por esta razón vemos a la función `freetable()` aparecer en `delI()`.

Una vez explicado esto, podemos mostrar la función NewI() que aloja un nodo en memoria.

```
INode NewI(U32 depth,U32 pos)
{
  INode *ret; nodeinfo **tab; U32 x,y;
  ret=m_ifreelist; // el nuevo nodo
  unsigned int tabsize;

  if(depth>=2) { tab=newtable(2); tabsize=0; }
  else { tab=newtable(255); tabsize=maxtab; }
// para los nodos de profundidad 0 (raíz) y 1, utilizamos
// tablas de 256 elementos. Para los demás, tablas de 2
// elementos.
  if(!tab || !(*tab)) return 0; // error - no hay vector
  if (ret) m_ifreelist=ret->nextfree; // usar lista libre
  else // se busca el nodo en la linea de nodos
  {
    x=m_nexti & 4095;
    y=m_nexti >> 12;
    if(!x) // línea llena? alojamos una nueva línea
    {
      if(y==256) return 0; // error - más de 2^20 nodos
      m_iblocks[y]=(INode*)malloc(4096*sizeof(INode));
      if(!m_iblocks[y]) return 0; // insuficiente memoria
      m_memused+=4096*sizeof(INode);
      bzero(m_iblocks[y],4096*sizeof(INode));
    }
    ret=&m_iblocks[y][x];
    m_nexti++;
  }
  // *** no hay errores? Inicializamos nodo interno ***
  ret->mps=0; ret->lastsym=0; ret->localRLE=1;
  ret->cpl=0; ret->xpl=0; ret->cred=1;
  ret->parent=0; ret->childcount=(BYTE)-1;
  ret->depth=depth; ret->pos=pos;
  ret->info=(*tab); (*tab)=(*tab)->nextfree;
  ret->info->nextfree=0; ret->tabsize=tabsize;
  ret->suffix=0; // liga sufijo
  if(tabsize==maxtab)
     bzero(ret->info,256*sizeof(nodeinfo));
  return ret;
}
```

4.4. Implementando el algoritmo de Larsson

Todas las variables y funciones que hemos definido forman parte de la clase `SuffixTreeBasicModel`. Adicionalmente a las estructuras básicas, necesitaremos otras clases y variables que nos ayuden a la construcción del árbol sufijo.

Para empezar, necesitaremos una función que nos indique si un nodo es nodo-interno o nodo-hoja. En su implementación, Timmermans añadía una bandera a la estructura LNode. Sin embargo, podemos distinguir si un nodo es hoja simplemente al examinar su dirección. Todos los nodos hoja se encuentran dentro de la ventana deslizante `m_leafbuf`. Y todos los nodos internos se encuentran fuera de esta ventana. Por lo tanto, sólo tenemos que verificar si la dirección de un nodo está dentro de los límites de `m_leafbuf`.

```
inline bool isleaf(LNode *p)
{ return(p>=m_leafbuf && p-m_leafbuf <=m_maxsize); }
// m_maxsize es una variable de SuffixTreeBasicModel que indica
// el tamaño de la ventana deslizante.
```

También definiremos una función getsuffix que regresa el sufijo de determinado nodo interno. No es estrictamente necesario definir esta función, pero puede ser útil para futuras expansiones del algoritmo.

```
inline INode *getsuffix(INode *p) { return (p->suffix); }
```

Para construir el árbol sufijo, es indispensable mantener una variable para el punto activo. Para encapsular los valores de ***(u,k,c)*** del punto activo (o de cualquier contexto en particular), Timmermans define la clase Point.

El nodo de inserción u se representa con ins. La variable de proyección k, se representa con *proj*. Y el caracter correspondiente al vértice para contextos deterministicos, lo denominamos sym.
Adicionalmente, para los contextos deterministicos añadimos el nodo r, que será el nodo en el cual termina el vértice en cuestión.

Para simplificar la programación, definimos una unión. Si nos referimos a "r", hablaremos de un nodo hoja. Si nos referimos a "rr", hablaremos de un nodo interno. Hacemos esto para evitar el *typecast* ((INode*)r) cada vez que necesitemos la posición o profundidad de r, cuando éste sea interno.

```
class Point
{
 public:
  inline bool InEdge() { return(proj>0);}
  U32 proj; // proyección
  BYTE sym; // el símbolo distintivo del hijo único si proj>0
  INode *ins; // nodo de inserción
  union { LNode *r; INode *rr; };
};
```

Una vez definido esto, podemos mostrar las funciones necesarias para manejar el árbol sufijo.
La actualización de nuestro árbol se realiza mediante la función UpdateTree:

```
inline void SuffixTreeBasicModel::UpdateTree(BYTE c)
{
  // Cuando el número de caracteres leídos es menor al tamaño
  // de la ventana deslizante, no hay necesidad de llamar a
  // AdvanceTail(). En el momento en que el tamaño de la cadena
  // es igual al de la ventana, es necesario deslizar la
  // ventana un caracter.

  if(m_cursize==m_maxsize)
    AdvanceTail();
// Si lo que deseamos no es utilizar una ventana deslizante, sino
// borrar el árbol completo cuando alcanza su tamaño máximo,
// reemplazamos AdvanceTail() por Reset(). En la práctica, este
// método es mucho más rápido que la ventana deslizante.
  else
    ++m_cursize; // m_cursize es el tamaño de la cadena.

  m_leafbuf[front].bufsym=c;
  m_leafbuf[front].parent=0;
  AdvanceFront();
};
```

Adicionalmente, usaremos otra función Update(), que a su vez llamará a UpdateTree(). La función Update() se encargará de actualizar variables auxiliares de los contextos, y para manejar casos especiales.

4.4.1. El modo RLE

Uno de estos casos es cuando empezamos a recibir una secuencia repetitiva del mismo caracter. Por ejemplo, en un archivo de texto podemos recibir una multitud de espacios. Como caso concreto, en el archivo PIC recibimos aproximadamente 60,000 bytes consecutivos con el valor de cero.
Para estos casos, definimos un estado o contexto especial, al cual

denominamos "RLE" (por las siglas del método de compresión Run Length Encoding, que comprime valores consecutivos de un mismo caracter).

El "modo RLE" estará definido con una variable llamada `m_rlecount`, que es igual al número de veces consecutivas que hemos recibido el mismo caracter. Definiremos un umbral, `MIN_RLE = 8`. Cuando `m_rlecount` es igual a `MIN_RLE`, diremos que hemos entrado al modo RLE.

La razón por la cual debemos manejar el modo RLE como un caso especial, es que en este modo hay una probabilidad muy grande de recibir el mismo caracter. Esto nos lleva a tomar el modo RLE como un contexto determinístico, cuya única predicción es el caracter repetido.

Otra razón por la cual éste debe ser un caso especial, es que al recibir el primer caracter distinto del repetido, se pierde la información de la repetición de caracteres en nuestro modelo PPM. Veamos por qué:

Supongamos que el punto activo corresponde al contexto "00000000" (8 ceros), cuya única predicción hasta el momento es "0". En ese punto recibimos un caracter "a". El contexto "00000000" tiene ahora 2 predicciones: "0" y "a". El sufijo del contexto es "0000000" (7 ceros), que también tiene como única predicción "0". Este contexto también tiene que ser actualizado.

Si el caracter que sucede a los 0's no ha sido visto previamente (por ejemplo si hemos recibido únicamente una sola cadena de 0's repetidos), entonces tendremos que actualizar tantos sufijos como 0's haya habido. Con esto, el modelo PPM no es capaz de reconocer por sí mismo una secuencia de caracteres repetidos, ya que el contexto correspondiente a la cadena completa de ceros tendrá un conteo (con *update exclusion*) de 1 para el caracter repetido, y un conteo de 1 para el primer caracter que suceda a la repetición. Entonces, para encontrar el contexto más adecuado, tendremos que usar una estimación de orden local, lo que requerirá buscar entre todos los sufijos y comparar cuál de ellos será más eficiente.

Por esta razón es que el compresor PPMZ usa un preprocesamiento que elimine las secuencias repetitivas del archivo, para posteriormente utilizar el algoritmo PPM. Pero esto dificulta la programación de los compresores, ya que debe haber dos modelos por separado: El modelo RLE - que debe estar mezclado con las rutinas de Entrada/Salida, y el modelo PPM.

En este trabajo, se propone una solución alternativa: manejar el modo RLE dentro del mismo modelo PPM, utilizando m_rlecount como predictor determinístico, con lo cual la compresión será equivalente al manejo RLE por

separado de PPMZ - sin tomar en cuenta la Estimación Secundaria de Escape (capítulo 6), con lo cual mejora aún más la compresión.

A continuación mostramos la función `Update()` que maneja el caso RLE:

```
void SuffixTreeBasicModel::Update(BYTE c)
{
  if(!alphabet[c]) {alphabet[c]=1;alphasize++;}
  // alphasize indica el tamaño del alfabeto visto hasta ahora.
  // Esto es útil para el manejo de archivos binarios.
  if(lastescaped) m_okcount=0;else
  if(m_okcount<1 << 20) m_okcount++;
  // lastescaped y m_okcount se usan en contextos determinísticos
  // (no nos conciernen por ahora).
  if(m_curpos && c==m_lastchar) // caracter repetido?
  {
    m_rlecount++;
    if(m_rlecount>=MIN_RLE)
       m_normcount=0; // entramos al modo RLE
  }
  else // los últimos dos caracteres difieren.
  {
    if(m_rlecount>=MIN_RLE)
      m_lastrlecount=m_rlecount; // salvamos m_rlecount
    m_rlecount=0; // salimos del modo RLE
    m_normcount++;
  }

  UpdateTree(c); // Actualizamos el árbol sufijo
  context4=(context4<<8) | c; // 4 últimos caracteres
};
```

4.4.2. Funciones Auxiliares

Para actualizar nuestro árbol sufijo, requeriremos de funciones que ayuden en el manejo de los nodos y vértices. Algunas funciones son fáciles de implementar, por lo que se muestra sólo su declaración.

```
inline BYTE getsym(LNode *p) // obtiene el caracter distintivo
{                            // de un nodo
  if(!p->parent) return 0;
  return (p->parent->info+p->ipos)->sym;
}

inline U32 getcount(LNode *p)
{ return (p->parent->info+p->ipos)->cnt; }
inline U32 getxcount(LNode *p)
{ return (p->parent->info+p->ipos)->xcnt; }
```

```
// getcount y getxcount regresan los conteos de un nodo en
// particular, accesando al vértice correspondiente.

U32  incCnt(INode *parent,nodeinfo *inf);
U32  incxCnt(INode *parent,nodeinfo *inf);
// incrementan los conteos PPM de un vértice determinado

void newEdge(INode *parent,LNode *child,BYTE c,
             U32 inicounts,U32 inixcounts);
// añade el nodo child al nodo parent, con c como caracter
// distintivo. También inicializa los conteos del vértice.

U32  delEdge(INode *parent,LNode *child);
// borra el vértice que une a parent y child.

U32  chgEdge(INode *parent,LNode *child,LNode *newchild);
// reemplaza el nodo hijo localizado en un vértice por otro.
// Los datos del vértice como el caracter distintivo y los
// conteos no cambian.

nodeinfo *getchild(INode *parent,BYTE c,LNode *forced);
// getchild busca el vértice con caracter distintivo c que se
// origina en el nodo parent. También se utiliza para añadir
// un nuevo nodo forced a parent. Si sólo se desea buscar un
// vértice, el parámetro forced deberá ser NULL.

LNode *getfirstchild(INode *p);
// utilizado para la fase de predicción y para la ventana
// deslizante. Busca el primer hijo de un nodo interno. Esta
// función es necesaria porque en nuestra implementación, los
// nodos con más de 127 hijos tienen sus vértices ordenados
// alfabéticamente para un rápido acceso.
inline U32 M0(U32 a,U32 b)
{ return ((a>=b) ? a-b : (a+m_maxsize)-b ); }
inline U32 MM(U32 i)
{ return ((i<m_maxsize) ? i : i-m_maxsize); }

// Las funciones M0 y MM son utilizadas para el manejo de la
// ventana deslizante en el buffer circular. M0 es utilizada
// para la resta de dos posiciones "a" y "b" en la ventana.
// MM se utiliza para las sumas.
```

4.4.3. Funciones específicas para la actualización del árbol sufijo

Las funciones dedicadas al manejo del árbol sufijo son las siguientes:

```
void getcsuffix(Point &p,BYTE c);
// busca el sufijo de un contexto en particular,
// sea o no un contexto determinístico.

void Canonize(Point *p); // La función Canonize de Ukkonen.

void UpdateCredits(INode *v,int i);
// manejo de créditos de la ventana deslizante

void UpdateTreeTop(BYTE c);
// En nuestra implementación, se actualizan los conteos
// full-update de los contextos cuya profundidad sea menor
// a un umbral establecido, para el manejo de los contextos
// no-deterministicos en la fase de predicción.

void Encode(BYTE symbol,ArithmeticEncoder *dest)
{
  Walk(symbol,dest,0);  // Encode y Decode se encargan de la
  Update(symbol);       // (de)codificación de un caracter para
  m_curpos++;           // posteriormente actualizar el árbol
}                       // sufijo

BYTE Decode(ArithmeticDecoder *src)
{
  BYTE ret=Walk(0,0,src);
  Update(ret);
  m_curpos++;
  return ret;
}

virtual BYTE Walk(BYTE c,ArithmeticEncoder *dest,
                  ArithmeticDecoder *src);
// La función SuffixTreeBasicModel::Walk simplemente utiliza
// una codificación de orden 0. Su descendiente,
// SuffixTreeModel::Walk se encarga del algoritmo PPM
// propiamente dicho.
```

4.4.4. Función AdvanceFront

Ahora procedemos a implementar el algoritmo de Larsson.

```
inline void SuffixTreeBasicModel::AdvanceFront()
{
  static INode *u, *v; // los nodos los creamos estáticos para
  static LNode *s;     // optimizar velocidad
  static nodeinfo *inf;

  int j; BYTE b, c;

  u=0;v=0;
// los nodos "u" y "v" son utilizados para actualizar
// las ligas sufijo

  while(ap.ins->depth+ap.proj>=MAXDEPTH) getcsuffix(ap,0);

// Si manejamos una ventana fija (no-deslizante), podemos utilizar
// la línea anterior para imponer un límite de profundidad al
// árbol. Esto nos ahorra tiempo y memoria en la creación de nodos.
  c=m_leafbuf[front].bufsym; // nuevo caracter
  Canonize(&ap);
  while (1) // Entramos al loop para actualizar el árbol
  {
    if (!ap.proj)
    { //El punto de inserción se localiza en un nodo.
      m_extracountnode=0;
      if (!ap.ins)
      {
        ap.r=m_rootnode; // Si el nodo de inserción es NIL,
        break;           // el punto de ins. se encuentra en ROOT.
      }
      s=&m_leafbuf[MO(front,ap.ins->depth)];
      ap.sym=c;

      inf=getchild(ap.ins, c,s);

      if(!inf)    // Buscamos si el contexto actual tiene un
      {           // hijo con caracter distintivo c. Si no es así,
        u=ap.ins;// se añade el nodo hoja correspondiente a la
        ap.r=0;  // posición actual.
      }
      else // si hay un hijo con caracter distintivo c,
      {    // entonces el endpoint ha sido encontrado.

        ap.r=inf->p; // procedemos a actualizar los conteos
                     // de full-update, y de update-exclusion
```

```
      inccnt(ap.ins,inf);
      // inccnt puede modificar la posición del vértice dentro
      // del vector, así que necesitamos recalcularla.
      inf=ap.r->parent->info+ap.r->ipos;

      incxcnt(ap.ins,inf);


      m_extracountnode=ap.r;

    // m_extracountnode es utilizada por Matt Timmermans para
      // remover el conteo que acabamos de añadir, en caso de
      // que el vértice sea dividido posteriormente.

      if(m_normcount) UpdateTreeTop(c);
      // Para que las repeticiones largas de caracteres no
      // afecten a los contextos de profundidad menor a
      // ap.ins, sólo actualizamos éstos contextos (full update)
      // cuando NO estemos en el modo RLE.
      break; // Al terminar la actualización, salimos del loop.
    }
  }
  else
  {
    // Si ap.proj es distinto de cero, el punto de inserción
    // se encuentra en un vértice.

    j=isleaf(ap.r) ?
      MM((ap.r-m_leafbuf)+ap.ins->depth) : ap.rr->pos;
    b=m_leafbuf[MM(j+ap.proj)].bufsym;

    // b es el siguiente símbolo en la etiqueta
    // del vértice (ap.ins,ap.r)

    if (c==b) // b es igual que m_leafbuf[front].sym ?
       break; // de ser así, el endpoint ha sido encontrado.


    // b no es el mismo símbolo que m_leafbuf[front].sym.
    // Por lo tanto habrá que dividir el vértice.

    u=NewI(ap.ins->depth+ap.proj,MO(front,ap.proj));

    // creamos un nuevo nodo u, en el punto de inserción.
    // Es en este punto que el vértice se dividirá en dos.

    if(!u)
    {
```

```
      fprintf(stderr,"\nMemoria agotada!\n Pos: %lu\n",m_curpos);
      exit(1);
    }
     U32 rcounts,rxcounts;
     rcounts=getcount(ap.r);rxcounts=getxcount(ap.r);
     chgEdge(ap.ins,ap.r,u);

     if (m_extracountnode==ap.r)
     { // como vamos a dividir el vértice, debemos eliminar
       // el conteo que acabamos de añadir
       if(rcounts>1) --rcounts;
       if(rxcounts>1) --rxcounts;
     }

   // El vértice que dividimos era (ap.ins,ap.r)
   // Ahora lo dividimos en dos: (ap.ins, u) y (u, ap.r)

   newEdge(u, ap.r, b,rcounts,rxcounts);

   // Una vez dividido el vértice podemos añadir el nodo-hoja s
   // y crear el vértice (u,s)

   s=&m_leafbuf[M0(front,u->depth)];
   newEdge(u, s, c, 1,1);

   if (!isleaf(ap.r))
     ap.rr->pos=MM(j+ap.proj);
   // actualizamos ap.r->pos. Esta parte es necesaria debido
   // a que tenemos que actualizar las posiciones de todos los
   // nodos internos para que queden dentro de los límites de
   // la ventana deslizante.
}

// Una vez añadido el nuevo nodo, procedemos a actualizar los
// créditos y las ligas sufijo

m_extracountnode=0;

if (u==ap.ins)
  UpdateCredits(u, M0(front,u->depth));
// Si es nodo nuevo, no actualizamos los créditos

if (v)
  { v->suffix=u; }
}
v=u;

// Una vez actualizada la liga sufijo, avanzamos el punto
// de inserción a su sufijo correspondiente, para continuar
```

```
      // la actualización del árbol mientras sea necesario.
    getcsuffix(ap,c);
  }

finish: // El árbol ha sido completado.
    // Ahora actualizamos el punto activo,
    // que es igual al endpoint.

  if (v && v->depth)
    v->suffix=ap.ins;
  ++ap.proj;  // incrementamos proj y front
  front=MM(front+1);
}
```

Las funciones AdvanceTail() y Canonize() casi no difieren de la implementación de Timmermans, excepto en el manejo de los conteos y vértices.

4.4.5. Función AdvanceTail

```
void SuffixTreeBasicModel::AdvanceTail()
{
  INode *u, *w;
  LNode *v,*s;
  BYTE c;
  int i, d;
  U32 oldcounts,oldxcounts;

  Canonize(&ap);      // ap es el punto activo
  v=&m_leafbuf[tail];// v es la hoja por borrar
  tail=MM(tail+1);    // avanzamos tail
  u=v->parent;        // obtenemos el nodo padre de v

   c=getsym(v);
   oldcounts=getcount(v); // salvamos los conteos del vértice (u,v)
   oldxcounts=getxcount(v);
   delEdge(u, v);          // y borramos el vértice (u,v)

  if (v==ap.r)         // si el punto activo está en (u,v)
  {                    // reemplazamos v por el nodo hoja equivalente
    i=MO(front,(ap.ins->depth+ap.proj));
    if (m_extracountnode==v)
    {
      m_extracountnode=m_leafbuf+i;
      ++oldcounts;
      ++oldxcounts;
    }
    newEdge(ap.ins, m_leafbuf+i,c,oldcounts,oldxcounts);
    UpdateCredits(ap.ins, i);
    ap.ins=getsuffix(ap.ins);
    ap.r=0; // inicializamos el punto activo:
```

```
            // Si r = 0, la función Canonize() busca el vértice
            // correspondiente en el árbol
    }
    else if (u && u!=m_rootnode && !u->childcount)
    {
      // si u sólo tiene un hijo restante, procedemos a borrar a u,
      // reemplazándolo con su único hijo
      w=u->parent;
      d=u->depth - w->depth;
      s=getfirstchild(u); // el hijo restante de u.


      if (u==ap.ins) // si el punto activo estaba entre (u,v)
        {            // será necesario actualizarlo
        ap.ins=w;
        ap.proj+=d;
        ap.sym=c;
        m_extracountnode=0;
        }
      else if (u==ap.r)
        {
          if (m_extracountnode==u) m_extracountnode=s;
          ap.r=s;
        }
      if (u->cred) // actualizamos los créditos del nodo
        UpdateCredits(w, M0(u->pos,w->depth));

      delEdge(u, s);
      chgEdge(w,u,s);

      if (!isleaf(s)) // si el nodo s es interno, actualizamos pos
      { ((INode*)s)->pos=M0(((INode*)s)->pos,d); }
      delI(u); // por último borramos el nodo u.
    }
}
```

4.4.6. Función Canonize

```
/* Si r no es cero, se asume que r==child(ins,a), y (ins,r) es el
vértice del punto de inserción. */

void SuffixTreeBasicModel::Canonize(SuffixTreeBasicModel::Point *p)
{
  unsigned int depthdiff;
 1:  if (p->proj && !(p->ins))
 2:  { p->ins=m_rootnode; --(p->proj); p->r=0; }
 3:  while (p->proj)
     {
 4:    if (!p->r)
       {
 5:      p->sym=m_leafbuf[MO(front,p->proj)].bufsym;
 6:      p->r=getchild(p->ins, p->sym,0)->p;
       }
 7:    if (isleaf(p->r)) break;
 8:    depthdiff=p->rr->depth - p->ins->depth;
 9:    if (p->proj<depthdiff)
10:   // si el punto de inserción no se extiende hasta r
11:     break;
12:   p->proj-=depthdiff; p->ins=p->rr; p->r=0;
     }
}
```

4.4.7. Obteniendo el sufijo de un contexto

Recordando lo visto en la sección 3.3.1, la liga sufijo provee el modo para encontrar el sufijo de un contexto, una operación básica en la construcción de los árboles sufijo.

Si sólo nos preocupamos de los contextos no-deterministicos, el obtener el sufijo de un contexto (localizado en un nodo) es una operación trivial.

```
ins = ins->suffix
```

Para obtener el sufijo de un contexto determinístico, habrá que llamar a la función `Canonize()`. Pero hay un modo más eficiente de obtener el sufijo de un contexto determinístico. Comparemos primero las operaciones que necesitamos realizar mediante Canonize(), una vez obtenido el sufijo del punto de inserción `ins`.

Ilustraremos la operación con un ejemplo.

Ejemplo 7. Aplicar el algoritmo de Ukkonen para encontrar el sufijo del contexto "xyzabc".

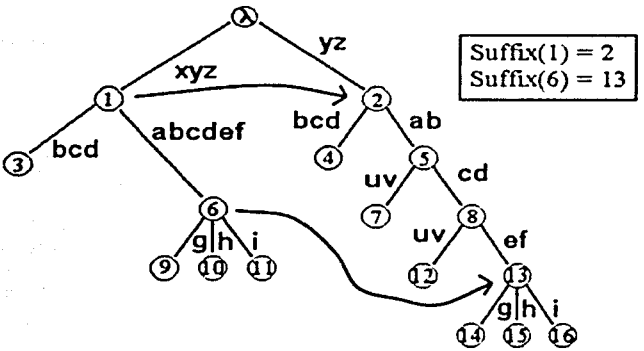


Fig. 5. Buscando el sufijo de un contexto determinístico

Observemos el árbol de la fig. 5. Supongamos que nuestro punto de inserción está localizado en el contexto "xyzabc", en el vértice (1,6). El vector correspondiente será (1,3,"a"). El sufijo de nuestro contexto estará en el vértice (5,8) y su vector será (5,1,"c").

Comparemos las operaciones requeridas por el algoritmo de Ukkonen para pasar del punto (1,3,"a") al punto (5,1,"c").

1. Buscamos el sufijo del punto de inserción.
   Ins = 2; proj = 3; r = NIL
2. Llamamos a Canonize.

Canonize()

Iteración #1.

Linea 3. proj != 0 ? Si.
Linea 4. r = 0 (NIL) ? Si.
Linea 5. sym = "a"
Linea 6. r = nodo 5.
Linea 7. r es nodo interno? Si. Continuamos.
Linea 8. depthdiff = depth(5) - depth(2) = 2
Linea 9. proj > depthdiff? Si. Recalculamos ins, r y proj.
Linea 12. ins = 5; r = 0; proj = 3 - 2 = 1. Se repite el loop.

Iteración #2.

Linea 3. proj != 0 ? Sí.
Linea 4. r = 0? Sí.
Linea 5. sym = "c"
Linea 6. r = nodo 8.
Linea 7. r es nodo interno? Sí. Continuamos.
Linea 8. depthdiff = depth(8) - depth(5) = 6 - 4 = 2.
Linea 9. proj > depthdiff? No. Terminamos.

Valores finales: ins = 5. r = 8; proj = 1; sym = "c".

Nuestro nuevo punto de inserción está localizado en (5,1,"c").

Para llegar a este punto requerimos de 1 salto de sufijo, 4 búsquedas de hijos (dos hijos por cada nodo), y un salto de padre-hijo. En total, 6 operaciones. Ahora vamos a analizar el árbol más de cerca.

OBSERVACIÓN.

Si en un árbol sufijo tenemos dos nodos *a* y *b*,
siempre se cumple que:
Si a es ancestro de b, entonces suf(a) es ancestro de suf(b).
(donde suf(a) es la liga sufijo del nodo a, y b puede ser un nodo hoja)

Si observamos con detenimiento el nodo 6 (en el cual termina el vértice correspondiente al contexto "xyzabc"), notaremos que su sufijo (nodo 13) tiene como ancestro el nodo 5.

Por lo tanto será más fácil buscar el sufijo de un *contexto determinístico* (o *nodo implícito* como lo llama Ukkonen) si en lugar de comenzar con el sufijo del nodo de inserción "ins", buscamos el sufijo del nodo hijo correspondiente "r" y navegamos por sus ancestros.

Como en un árbol sufijo sólo hay un padre por nodo, nuestra búsqueda no requerirá buscar en los vértices, sino saltar a través del apuntador *parent*.

En la implementación de Ukkonen, esto no era posible ya que sólo había apuntadores padre->hijo. Pero el algoritmo de Larsson requiere forzosamente la existencia de apuntadores hijo->padre (debido al manejo de créditos en la ventana deslizante), por lo cual podemos formular un algoritmo que navegue por los ancestros de un nodo.

Algoritmo "buscar sufijo". Encuentra el sufijo de un contexto determinístico.

1. Obtener la longitud del contexto
```
clen = ins.depth + proj
```
2. Disminuir la longitud del contexto, por uno
(la longitud del sufijo es la longitud de la cadena menos uno)
```
clen = clen - 1
```
3. Obtener el sufijo del nodo hijo. Notemos que el sufijo de un nodo hoja es el nodo hoja que le sucede en la ventana deslizante, por lo que sólo habrá que incrementar el apuntador por uno en estos casos.
```
r = getsuffix(r)
```
4. Obtener el nuevo nodo de inserción.
```
ins = r->parent
```
5. Sustituir ins por ins->parent mientras la profundidad sea mayor a la deseada.
```
while(ins->depth > clen)
{ r = ins; ins = ins->parent; }
```
6. Si el nodo hijo es interno y su profundidad es igual a la deseada, el nodo de inserción es el nodo hijo.
```
If(isleaf(r) && ((INode*)r)->depth == clen)
{ ins = r; r = 0; }
```
7. Ajustar proj según la profundidad.
```
proj = clen - ins->depth;
```
8. Actualizar el caracter distintivo de ser necesario
```
if(proj) sym = getsym(r);
```

Ahora veamos el número de operaciones requeridas con el nuevo algoritmo.

Ejemplo 8. Aplicar el nuevo algoritmo para encontrar el sufijo del contexto "xyzabc" del ejemplo anterior.

El punto de inserción es (1,3,"c").

ins = nodo #1; proj = 3; sym = "c"; r = nodo #6.

1. Obtenemos la longitud del contexto.
   clen = depth(nodo #1) + proj = 3 + 3 = 6
2. clen = clen - 1 = 5
3. r = suf(nodo #6) = (nodo #13)
4. ins = r->parent = nodo #8
5. ins->depth > 5 ? (6 > 5?): si.
   r = ins; ins = ins -> parent;
   r = nodo #8; ins = nodo #5;
5. ins->depth > clen? (4 > 5?) no.
6. r-> depth == clen? (6 == 5?) no.
7. proj = clen - ins->depth = 5 - 4 = 1.
8. sym = "c".

El nuevo contexto se localiza en (5,1,"c").

Ahora contemos las operaciones.
1 Salto de sufijo, 1 salto hijo-padre y 0 búsquedas de hijos.
Total: 2 operaciones contra 6 del algoritmo de Ukkonen.

Dicho esto, podemos exponer la función getcsuffix.

```
void SuffixTreeBasicModel::getcsuffix(Point &p,BYTE c)
{
  U32 clen;
  LNode *suf;
  INode *ins;

  if(!p.r || !p.proj) goto DEFAULT;
  // cuando el contexto es no-determinístico, es un caso trivial

  if(isleaf(p.r)) // el nodo hijo del contexto es un nodo-hoja?
  {
    // comparamos si el nodo hoja no es la hoja más reciente
    // en este caso, no es posible obtener su sufijo
    if(MM(p.r-m_leafbuf+1)>=front)
      goto DEFAULT;

    clen=p.ins->depth+p.proj-1;
    suf=m_leafbuf+MM(p.r-m_leafbuf+1);
    p.sym=m_leafbuf[M0(front,p.proj)].bufsym;
    ins=suf->parent; // si el nuevo nodo de inserción es NIL
    if(!ins)         // usamos el método de Ukkonen, al no poder
      goto DEFAULT;  // obtener el padre del nuevo sufijo

    while(ins->depth>clen)
    { suf=ins;ins=ins->parent; }

    if(!isleaf(suf) && ((INode*)suf)->depth==clen)
    { ins=(INode*)suf; suf=0; }
    p.ins=ins;
    p.r=suf;
    p.proj=clen-p.ins->depth;

    if(p.proj)
      p.sym=getsym(p.r); // actualizamos el caracter distintivo sym
    else
    {
      p.sym=getsym(p.ins);
      if(p.r && getsym(p.r)!=c) p.r=0;
    }
  return;
  }
```

```
    // si el nodo hijo NO es un nodo hoja:

    suf=getsuffix(p.rr);
    if(!suf || !suf->parent) // si la profundidad es 0 o 1,
       goto DEFAULT;         // usamos el método de Ukkonen
                             // (es más rápido en estos casos)
    clen=p.ins->depth+p.proj;
    clen--;
    p.r=suf;p.ins=p.r->parent;
    while(p.ins->depth>clen)
      { p.r=p.ins;p.ins=p.ins->parent; }
    if(p.rr->depth==clen)
      { p.ins=p.rr; p.r=0; }
    p.proj=clen-p.ins->depth;

    // actualizamos sym

    if(p.proj) p.sym=getsym(p.r);else
    {
      p.sym=getsym(p.ins);
      if(p.r && getsym(p.r)!=c) p.r=0;
    }

    return;

    /**/

  DEFAULT: // método tradicional de Ukkonen
    p.ins=getsuffix(p.ins);
    p.r=0;
    Canonize(&p);
}
```

Ahora que hemos desarrollado el algoritmo para actualizar nuestro árbol sufijo en línea, podemos utilizar nuestra clase **SuffixTreeBasicModel** para desarrollar un compresor PPM rápido y eficiente, cuyo algoritmo expondremos en el siguiente capítulo.

5. COMPRESOR PPM CON UN ÁRBOL SUFIJO

5.1. Introducción

A continuación mostramos un diagrama de flujo para nuestro compresor de árbol sufijo, conforme a lo visto en el capitulo anterior.

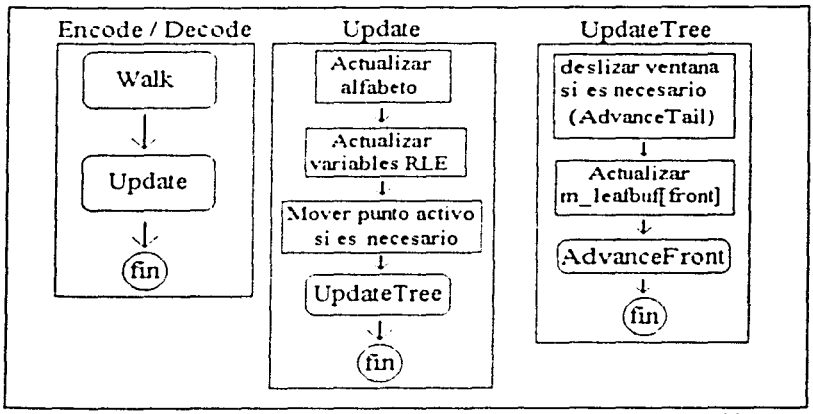


Fig. 6. Diagrama de bloques de las rutinas de compresión

Como hemos visto, la rutina `Walk()` se encarga de (de)codificar el nuevo caracter según el modelo PPM. Una vez (de)codificado el nuevo caracter, `encode()` o `decode()` se encargan de llamar a la función `update()`.

Hasta ahora sólo nos hemos ocupado de la función `Update()`, que se encarga de actualizar las variables auxiliares (como el punto activo), y el árbol sufijo según el algoritmo de Larsson.

La función `Walk()` es la encargada de (de)codificar los caracteres del archivo a (des)comprimir. Es esta rutina en la cual desarrollamos el algoritmo de compresión basado en PPMZ, incluyendo la predicción de contextos deterministicos, la Estimación de Orden Local (LOE), y la Estimación de Escapes Secundaria (SEE).

En este capitulo expondremos la función `Walk()`, especificamente en lo que concierne al manejo de contextos no deterministicos. La predicción de contextos deterministicos será expuesta en el capitulo 6.

5.2. Clases de C++ utilizadas en el modelo

Todo compresor PPM necesita un modelo estadístico independiente, ya sea de orden 0, o de orden -1 (ver sección 2.1). La mayoría de los compresores PPM utilizan un modelo de orden -1 (en el que todos los símbolos tienen conteo de 1) cuando ninguno de los contextos en un momento determinado ha sido capaz de predecir un caracter. Nosotros usaremos un modelo de orden 0, inicializado con conteos de 1 para todos los caracteres posibles.

El modelo estadístico de orden 0 será el modelo adaptivo de Timmermans, que asigna más peso a los caracteres más recientes. El modelo está encapsulado en la clase `SimpleAdaptiveModel`, al que haremos algunas modificaciones para mejorar su desempeño en la compresión.

El modelo `SimpleAdaptiveModel` consta de una ventana deslizante de 4 KBytes de longitud, dividido en zonas de diversos tamaños. La zona más reciente es la que mayor peso tiene en el modelo. Timmermans define la clase `ProbHeapModelBase` como una clase de modelo estadístico básico de orden 0 para su uso con un codificador aritmético.

La jerarquía de clases para los modelos estadísticos en nuestro programa de compresión es la siguiente:

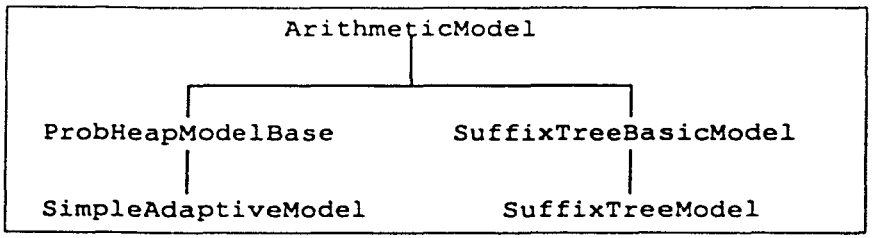


Fig. 7. Jerarquía de clases para la etapa de modelado

De este modo, cualquier clase descendiente de `ArithmeticModel` puede ser usada con el codificador aritmético de Timmermans.

Nuestro modelo de árbol sufijo está contenido en la clase `SuffixTreeBasicModel`, cuyo algoritmo de compresión consiste en un modelo de orden 0, que puede ser substituido con alguna clase descendiente. Esto se logra haciendo de la función `Walk()` - que es la que realiza la compresión - una función virtual que puede ser reemplazada.

En este caso, la función `SuffixTreeModel::Walk` es la que contiene el modelo PPM, como veremos más adelante. La clase `ProbHeapModelBase` está definida a continuación.

```
class ProbHeapModelBase : public ArithmeticModel
{
  public:
  ProbHeapModelBase();
  ~ProbHeapModelBase();

  public: // estas dos funciones son heredadas de ArithmeticModel
  virtual void Encode(BYTE symbol,ArithmeticEncoder *dest);
  virtual BYTE Decode(ArithmeticDecoder *src);

  public:
  double entropyused; // esta variable es la primera modificación
// que hacemos al modelo de Timmermans, debido a que en nuestro
// modelo de árbol sufijo tendremos varios modelos de orden 0
// independientes, donde escogeremos aquél que tenga la menor
// entropía, es decir, aquel que produzca la mayor compresión.

// las funciones Encode, Decode y Update son las funciones
// encargadas de codificar, decodificar un caracter, y actualizar
// el modelo de orden 0.
  void Encode(BYTE symbol,ArithmeticEncoder *dest,ExclusionSet
&excl);
  BYTE Decode(ArithmeticDecoder *src,ExclusionSet &excl);
  virtual void Update(int symbol) = 0;
};
```

En esta clase aparece una nueva variable creada por Timmermans para el manejo de exclusiones: `excl`, del tipo `ExclusionSet`. Cuando un contexto en particular falla en predecir un caracter, todos los caracteres que contiene son añadidos a una lista de ***caracteres excluidos.*** Al mantener esta lista, nuestro modelo de orden 0 libera el espacio reservado para estos caracteres, como lo vimos al final del ejemplo 3 (capitulo 2). La clase SuffixTreeModel incluye una variable de este tipo, llamada `m_excl`.

```
class ExclusionSet // archivos exclude.cpp y exclude.h
{ public:
    ExclusionSet();
    bool Exclude(BYTE sym); // Excluimos un caracter
    bool IsExcluded(BYTE sym);
    // IsExcluded verifica si un caracter está excluido
    void Clear(); // borra todos los caracteres de la lista
    unsigned NumExcls(); // cantidad de caracteres excluidos
};
```

La clase SimpleAdaptiveModel la definimos como sigue:

```
class SimpleAdaptiveModel : public ProbHeapModelBase
{
  public:
  SimpleAdaptiveModel();
  void Reset(); // reset() asigna todos los conteos a 1
  void Update(int symbol);
  private:
  int window[4096],*w0,*w1,*w2,*w3,*w4,*w5,*w6,*w7;
  // manejo de la ventana deslizante (ver más adelante)
};
```

5.3. Algoritmo PPM

Una vez expuestas las clases necesarias para implementar nuestro algoritmo, delinearemos un pseudocódigo para la función `Walk()`.

I. Fase de inicialización

1. Inicialización; nueva variable **contexto** = punto activo

II. Predicción determinística

2. ¿Estamos en un contexto determinístico?
   En tal caso, efectuar la Estimación Secundaria de Escape para contextos determinísticos. Si no hubo escape, **terminar.**

3. Acabamos de salir de una secuencia muy larga de **modo RLE?**
   (m_rlecount >= MAX_RLE ?) De ser así saltamos a **orden 0**

III. Predicción no determinística

4. Heurística especial para archivos aleatorios (p.e. GEO)
   Si es un archivo binario y hemos codificado con un promedio mayor a 5 bits por caracter, saltar al **orden 0**

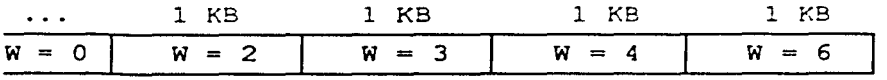
5. Si el contexto tiene profundidad 0, usar modelo de **orden 0**
6. Construir lista de contextos utilizando **LOE**
7. Para cada contexto en la lista, ejecutar **rutina de predicción**
   (incluyendo exclusiones, estimación secundaria de escapes...)
   Si hubo una predicción exitosa, **terminar.**

IV. Predicción de orden 0

8. (de)codificar el caracter con el modelo más adecuado
9. Actualiza los modelos de orden 0
10. **Terminar.**

5.4. Modelado de orden 0

El modelo básico de orden 0 se encuentra en la clase SimpleAdaptiveModel creada por Timmermans. Originalmente, esta clase tenía una ventana deslizante dividida en 4 zonas de 1 Kbyte cada una, asignando mayores pesos a los caracteres más recientes. Dichos pesos fueron asignados en base a prueba y error, fueron 6,4,3 y 2. Los caracteres que sobrepasen 4 KBytes de antigüedad son eliminados del modelo.



Al recibir un caracter, éste se añade al final de la ventana, y se suma al modelo de orden 0 con un peso de 6. Conforme se van recibiendo nuevos caracteres, los caracteres más viejos van perdiendo su peso.

Por ejemplo, si el caracter "a" está en la posición 1023 y pasa a la 1024, del modelo de orden 0 se substraen 2 puntos correspondientes al caracter "a".

De éste modo, los caracteres más recientes influyen más sobre el modelo que los caracteres más antiguos.

Experimentalmente se encontró que dividir la ventana logarítmicamente producía mejores resultados en la compresión de archivos. La ventana se dividió en 7 zonas, de 32, 64, 128, 256, 512, 1024 y 2048 bytes respectivamente. En total tenemos una ventana de 4064 bytes.

Los pesos también se asignaron en base a prueba y error. Dichos pesos fueron de 24, 12, 5, 4, 3, 2 y 1.

Los últimos 32 caracteres recibidos tendrán un peso de 24; los siguientes 64 tendrán un peso de 12; los siguientes 128 tendrán un peso de 6; y asi sucesivamente. El modelo resultante es mucho más eficiente para la compresión que un modelo de orden 0 sin memoria.

Hay que notar que este modelo es utilizado con exclusiones de actualización, es decir, los caracteres que se añaden al modelo son aquellos que no se han podido codificar con un modelo PPM.

El código de la clase SimpleAdaptiveModel es el siguiente (repetimos la definición para mayor claridad):

```
class SimpleAdaptiveModel : public ProbHeapModelBase
{ public:
   SimpleAdaptiveModel();
   void Reset();
   void Update(int symbol); // heredado de ProbHeapModelBase
 private:
   int window[4096],*w0,*w1,*w2,*w3,*w4,*w5,*w6,*w7;
   // *w0 .. *w7 NO son los pesos, sino los apuntadores que indican
   // las divisiones entre las zonas de la ventana
}

SimpleAdaptiveModel::SimpleAdaptiveModel() { Reset(); }

void SimpleAdaptiveModel::Reset()
{
   clear();
   int i;
   // Inicializamos la ventana
   for (i=4096;i--;) window[i]=-1; // -1 = no hay símbolo

   w0 = window; // cada zona es del doble de tamaño que la anterior
   w1 = w0 +   32; w2 = w1 +   64; w3 = w2 +  128; w4 = w3 +  256;
   w5 = w4 +  512; w6 = w5 + 1024; w7 = w6 + 2048;
}

void SimpleAdaptiveModel::Update(int symbol)
{ // w1..w7 se modifican conforme a un buffer circular
   w1=((w1==window)?w1+4095:w1-1);
   if (*w1>=0) SubP(*w1,12); // 24 - 12 = 12
   w2=((w2==window)?w2+4095:w2-1);
   if (*w2>=0) SubP(*w2,7); // 12 - 7 = 5
   w3=((w3==window)?w3+4095:w3-1);
   if (*w3>=0) SubP(*w3,1); // 5 - 1 = 4
   w4=((w4==window)?w4+4095:w4-1);
   if (*w4>=0) SubP(*w4,1); // 4 - 1 = 3
   w5=((w5==window)?w5+4095:w5-1);
   if (*w5>=0) SubP(*w5,1); // 3 - 1 = 2
   w6=((w6==window)?w6+4095:w6-1);
   if (*w6>=0) SubP(*w6,1); // 2 - 1 = 1
   w7=((w7==window)?w7+4095:w7-1);
   if (*w7>=0) SubP(*w7,1); // 1 - 1 = 0

   w0=((w0==window)?w0+4095:w0-1);
   *w0=symbol;
   AddP(symbol,20);
}
```

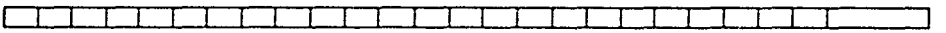
5.4.1. Expandiendo el modelado de orden 0 para archivos binarios

Una vez que tenemos un modelo funcional de orden 0, podemos extenderlo para trabajar con archivos binarios. Estos archivos suelen utilizar palabras de 16 o 32 bits, sobre todo los programas ejecutables para los procesadores 80x86 (486, Pentium™).

Para manejar este tipo de archivos, necesitaremos de 5 modelos de orden 0: Un modelo global, y 4 modelos independientes para el *offset módulo 4* de la posición actual del archivo.

```
SimpleAdaptiveModel order0,order0a,order0b,order0c,order0d;
SimpleAdaptiveModel *zptr4[4];

// *** Inicialización

zptr4[0]=&order0a;zptr4[1]=&order0b;
zptr4[2]=&order0c;zptr4[3]=&order0d;

// *** Codificación de orden 0 en la rutina walk()

SimpleAdaptiveModel *model4,*coder1,*coder2;

  model4=zptr4[m_curpos & 3];
```

El apuntador model4 escoge uno de los 4 modelos utilizados para archivos binarios, dependiendo de la posición del archivo, como aquí se muestra:



Para la codificación de nuestro caracter en modelo de orden 0, disponemos de 2 modelos: `model4` y `order0`. El modelo `order0` es nuestro modelo básico, y `model4` será el modelo a utilizar para archivos binarios.

Para distinguir cuál modelo debemos utilizar para la codificación, simplemente escogemos aquel modelo cuya entropia sea la menor. Disponemos de dos variables de punto flotante que guardarán las entropías: `zent1` para `order0`, y `zent4` para `model4`.

```
  coder1=&order0; coder2=model4;
  if(zent4>0 && zent4<zent1) { coder1=model4; coder2=&order0; }
```

El modelo con el cual codificaremos el caracter, lo guardaremos en el apuntador `coder1`. El otro modelo lo guardaremos en `coder2`.

Una vez que disponemos de estos apuntadores, podremos codificar con un modelo, y actualizar ambos modelos, para distinguir en la siguiente iteración de `Walk()`, qué modelo conviene utilizar.

```
if(src)
    c=coder1->Decode(src,m_excl);
  else if(dest)
    coder1->Encode(c,dest,m_excl);

  coder2->Encode(c,0,m_excl);
```

`src` y `dest` son el codificador y decodificador aritméticos. Notemos que el parámetro utilizado para coder2 es 0 (NULL), de tal manera que coder2 no codifica su salida, únicamente actualiza sus datos estadísticos.
Una vez (de)codificado el caracter, actualizamos las variables de entropía.

```
  zentropy+=coder1->entropyused;
  bitsused[m_curpos & 3]+=coder1->entropyused;
  zent1+=order0.entropyused;
  zent4+=model4->entropyused;

  order0.Update(c);
  model4->Update(c);

  return(c);
```

La variable bitsused se actualiza tanto para codificación PPM como para codificación de orden 0. En archivos de tipo aleatorio (como *geo* del Calgary Corpus), algunas veces es más eficiente utilizar un modelo de orden 0 directamente, sin tener que pasar por la codificación PPM. Estos casos los podemos detectar cuando la codificación PPM se vuelve ineficiente, mediante un simple umbral al inicio de la codificación no-determinística (paso 4).

```
// heurística especial para archivos aleatorios
 if(alphasize>=128 && m_curpos>=50 &&
    bitsused[m_curpos & 3]>5*(m_curpos/4))
   goto ORDER0;
```

Esta decisión sencilla pero efectiva, simplemente compara si hasta el momento los caracteres que se han codificado sobrepasan los 5 bits por caracter (5 bpc). Esto se hace para los 4 modelos de orden 0, de tal manera que la codificación de orden 0 sólo se utiliza de ser necesaria.

5.5. Estimación de Orden Local (LOE)

La estimación de orden local fue introducida por Charles Bloom en su compresor PPMZ, debido a que establecer *a priori* un límite de orden limita el desempeño de la compresión. La pregunta que se debe plantear es, ¿cómo conocer cuál es el orden óptimo para codificar un caracter?

Si se escoge un orden demasiado alto, es muy probable que el contexto elegido contenga muy poca información, por ejemplo, 2 ó 3 caracteres con un conteo de 1 ó 2 para cada uno de éstos caracteres. Esto resultará en una codificación ineficiente: Si el símbolo a codificar está presente en el contexto, la baja probabilidad de dicho símbolo será aproximadamente de 1/3, lo que implica una codificación de al menos -log2(1/3) = 1.5 bits. Si el símbolo no está presente en el contexto (lo que es muy probable), se requerirá codificar un escape.

Si el orden a escoger es demasiado bajo, caemos en el otro extremo: Un contexto con demasiados caracteres, y una probabilidad muy baja para cada uno de ellos.

La solución presentada por Bloom se basa en establecer una tasa de confianza, según la probabilidad del símbolo más probable ó ***MPS***:

$$\text{Confianza} = 1.1 * [\ P(\text{MPS})\ ]$$

Al construir una lista de todos los contextos, el contexto más indicado será aquél con mayor indice de confianza.

La solución de Bloom presenta un problema en los contextos de orden muy alto: Podemos tener un contexto cuyo símbolo más probable tenga una alta probabilidad de ocurrencia, y sin embargo el contexto tenga muy poca información, lo que indica una muy alta probabilidad de escape. Para solucionar este problema, Bloom incluye la probabilidad de escape en la ecuación:

$$\text{Confianza} = 1.1 * [\ P(\text{MPS}) * (1-P(\text{esc}))\ ]$$

Sin embargo, para un buen desempeño del LOE, es necesario tener una Estimación Secundaria de Escape ó SEE (basada en varias tablas estadisticas independientes), lo que aumenta la complejidad de su algoritmo, por lo que es necesario buscar una alternativa rápida y eficiente. En esta Tesis proponemos una solución paramétrica al problema del orden local.

Conociendo el hecho de que los contextos de orden demasiado alto poseen muy poca información, se pueden descartar aquellos contextos cuyo conteo total para el símbolo más probable (MPS) sea demasiado bajo. En base a esto, establecemos el primer parámetro, que llamaremos MINLOE, con valor de 5 (encontrado experimentalmente).

Para evitar saltar contextos jóvenes de bajo orden, también estableceremos un parámetro de orden mínimo, para evitar saltar contextos de bajo orden. Dicho parámetro será MINORDER, con valor de 4 (también encontrado experimentalmente).

El algoritmo resultante es una **LOE de 2 etapas**: En la primera etapa, se descartan los contextos con poca información estadística. En la segunda etapa, se aplica el algoritmo de Bloom.

La primera etapa consta sólo de 2 líneas de código:

```
while(ctx.ins->depth>MINORDER && ctx.ins->info->cnt<MINLOE)
    ctx.ins=getsuffix(ctx.ins);
```

donde:

`ctx.ins` es el nodo de inserción de nuestro árbol sufijo, es decir, el contexto correspondiente al punto activo.
`ctx.ins->info->cnt` es el conteo total del símbolo más frecuente (la tabla de frecuencias está ordenada de mayor a menor).
`ctx.ins->depth` es la profundidad (orden) del contexto.

Esta sencilla modificación, además de disminuir la complejidad del algoritmo, mejora su desempeño en la compresión.

Se ha notado que al modificar los parámetros, la compresión en algunos archivos del Calgary Corpus mejora, pero empeora en otros; es decir, la aproximación es perfectible. Sin embargo en promedio, es más efectiva que la aproximación exhaustiva de Bloom en su compresor PPMZ2 (que prueba todo el algoritmo de (de)codificación con diversos órdenes, escogiendo el mejor).

Para la segunda etapa, por cuestiones de tiempo no se incluyó la Estimación Secundaria de Escape (SEE) en el cálculo de la confianza de un contexto, por lo que el desempeño de nuestro compresor no iguala a PPMZ2 en el caso del archivo más largo del Corpus (book1). A pesar de esto, nuestro

compresor fue mejor en todos los demás archivos, como podremos ver más adelante.

La Estimación de Orden Local queda implementada como sigue:

```
void SuffixTreeModel::makecontextlist(BYTE c,Point &ctx,unsigned
int &numctx)
{
  int i;
  U64 mpsp,bestmpsp;
  nodeinfo *curinfo,*rleinfo;
  INode *lastsuf;
  bestmpsp=0; // bestmpsp es el mejor valor de confianza obtenido

  // MAXNONDET es el límite superior de orden para contextos
  // no-determinísticos, en nuestro caso es 20
  if(ctx.ins->depth>MAXNONDET)
  {
    if(ctx.ins->depth-MAXNONDET<=MAXNONDET)
    {
      while(ctx.ins->depth>MAXNONDET) ctx.ins=getsuffix(ctx.ins);
    }else
    {
      ctx.ins=m_rootnode;
      ctx.proj=MAXNONDET;
      ctx.r=0;
      Canonize(&ctx);
      while(ctx.proj)
        getcsuffix(ctx,0);
    }
  }
  ctx.r=0;
  // LOE: Primera etapa
  while(ctx.ins->depth>MINORDER && ctx.ins->info->cnt<MINLOE)
     ctx.ins=getsuffix(ctx.ins);

  // Segunda Etapa
  for (i=0;i<MAXWALK && ctx.ins;++i)
  {
     if(ctx.ins==m_rootnode)
       break;
     curinfo=ctx.ins->info+ctx.ins->mps;
     mpsp=curinfo->cnt;

     if(ctx.ins->localRLE>=2 &&
        !m_excl.IsExcluded(ctx.ins->lastsym))
     {
       rleinfo=getchild(ctx.ins,ctx.ins->lastsym,0);
       if(rleinfo==curinfo)
         mpsp+=(rleinfo->xcnt*(ctx.ins->localRLE-1));
```

```
        else
          mpsp+=(rleinfo->xcnt*(ctx.ins->localRLE));
      }
      // En caso de contextos cuyo caracter más probable haya sido
      // repetido 2 o más veces, multiplicamos por localRLE
      // Las multiplicaciones y divisiones las efectuamos con
      // aritmética de punto fijo, con 16 bits para los decimales

      mpsp<<=16;
      mpsp/=(ctx.ins->cpl+ctx.ins->childcount+1);
      mpsp+=(1<<16)/(ctx.ins->childcount+1);
      if(!m_excl.IsExcluded(ctx.ins->lastsym))
         mpsp+=(1<<16)/(ctx.ins->childcount+1);

      if (!numctx)
        bestmpsp=mpsp;
          else
      if(mpsp>bestmpsp)
      {
        numctx=0;
        bestmpsp=mpsp;
      }else
      if(numctx>MAXNONDET)
        break;

    cnode[numctx]=ctx.ins;
    ++numctx;
    ctx.ins=getsuffix(ctx.ins);
  }
}
```

Una vez que la lista de contextos está completa, podemos efectuar el algoritmo de predicción PPM visto en el capítulo 2 (con algunas modificaciones adicionales).

5.6. Estimación de Escape Secundaria (SEE)

Recordando el problema de la frecuencia-cero visto en el algoritmo PPM, notamos que hay diversas heuristicas para aproximar la frecuencia de escape en un determinado contexto, como los métodos C y D.

El problema con estos métodos de escape es que sólo dependen de 2 variables: El número de caracteres distintos vistos en el contexto (q), y el número de caracteres vistos en total (n). No se toma en cuenta el contexto mismo - por ejemplo, en un archivo de texto, después de un cambio de linea (CR/LF) la distribución de probabilidad de los caracteres tiende a ser uniforme.

Los denominados "métodos de escape" tampoco toman en cuenta que la probabilidad de escape de un contexto también depende del orden del contexto, o de si ha habido escapes anteriores a dicho contexto.

Bloom toma varios de estos factores en cuenta para su Estimación de Escape Secundaria: En base a ciertos valores, como q, n, el orden del contexto, etc., construye una matriz cuyos elementos es el número de hits (no escapes) y misses (escapes) para los contextos correspondientes.

Los valores que Bloom utiliza para su compresor PPMZ1 son los siguientes:

1. El orden del contexto (0-8) cuantizado a 2 bits
2. El número de escapes (q) cuantizado a 2 bits
3. El número de caracteres vistos (n) cuantizado a 3 bits
4. 7 bits del último caracter del contexto (orden 1) y 3 bits del caracter de orden 2

El resultado es un arreglo de 16 bits. Adicionalmente, Bloom también usa otros arreglos de 15 y 7 bits, teniendo en total 3 matrices de probabilidades de escape.

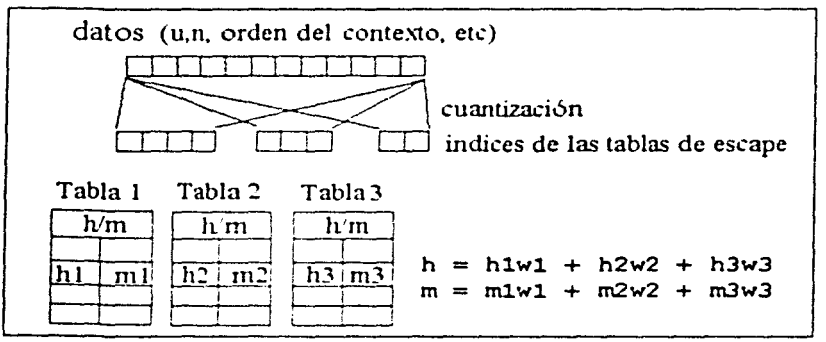


Fig. 8. Estimación de Escape Secundaria (SEE)

Los pesos w1, w2 y w3 con que se calculan *h* (prob. de hit) y *m* (prob. de escape) son calculados en base a la entropía o la probabilidad de escape (m / h+m), también mediante el método de prueba y error.

Una vez obtenidas las variables *h* y *m*, podemos calcular P(hit) y P(esc) para el uso de nuestro codificador aritmético. Después de haber (de)codificado el caracter o escape del contexto, se actualizan las tablas secundarias.

dependiendo si hubo un escape en nuestro contexto PPM. Son estas técnicas, LOE y SEE, las que le dan una gran eficiencia al compresor PPMZ de Bloom. Al mejorar ambas técnicas, nuestro compresor será aún más eficiente.

5.7. Modificaciones a la Estimación Secundaria de Escape

El algoritmo SEE de nuestro compresor es muy similar al de Bloom. Estas son las diferencias:

5.7.1. Preprocesamiento de *q* y *n*

En lugar de utilizar los conteos de escape y de caracteres directamente en nuestras matrices secundarias, precalculamos la probabilidad de escape, añadiendo factores como las repeticiones consecutivas de un caracter en un contexto (a este número lo llamaremos **RLE local**), utilizando esta probabilidad aproximada como índice para nuestras matrices.

Comenzamos con dos variables auxiliares, *probhit*, y *probmiss*. Dichas variables serán utilizadas para el cálculo de la probabilidad de escape aproximada, que se utilizará en el cálculo de los índices de las matrices secundarias. Una vez fijados estos índices, se obtiene la probabilidad final de escape mediante el algoritmo *SEE*, guardándola en *probhit* y *probmiss*.

```
    phit=totalcnt;
    phit-=xtot;
    probhit=phit; // probhit es igual al conteo total de caracteres
                  // no-excluidos vistos en el contexto actual
    rle_count=0;
    if(curnode->localRLE>=2)
    {
// si no ha habido escapes hasta ahora, utilizamos el "RLE local",
// que es similar al conteo RLE visto en el capítulo 4. Si el
// último caracter visto en un contexto es igual al anterior, se
// incrementa el conteo de RLE local para el contexto. En el
// momento en el que el contexto recibe un caracter distinto, el
// RLE local adquiere el valor de 1.

// El valor del RLE local se multiplica por el conteo con exclusión
de actualización del caracter repetido, y se divide entre 4. Este
valor es añadido a la variable probhit.
```

```
      edge=getchild(curnode,curnode->lastsym,0);
    if(!cnum2 && !m_excl.IsExcluded(edge->sym))
    {
      rle_count=edge->xcnt*(curnode->localRLE);
      probhit+=rle_count/4;
      rle_count=rle_count/3+1;
    }
  }

  if(!detesc && !cnum2)
  // si el último caracter (de)codificado no requirió escape,
  // entonces la probabilidad de escape disminuye. Por esto
  // multiplicamos probhit por 1.1
  {
    if(curnode->cpl>=4)
      probhit=(probhit*11)/10;
  }

  // La variable probmiss es el conteo de escapes para el
  // contexto actual. De este modo, probhit es un valor
  // modificado de n, y probmiss es el valor de q.
  probmiss=curnode->childcount+1;
```

Ya que han sido calculadas las variables probhit y probmiss, obtenemos nuestra principal variable, *escidx*, para la Estimación Secundaria de Escape.

```
escidx=((probhit/probmiss)-1); //2
if(escidx>=5)
{
  if(escidx>=80) escidx=7;else
  if(escidx>=30) escidx=6;else escidx=5;
}
if(escidx) escidx+=3;else
{ escidx=(3*(curnode->xpl-jmax))/jmax;if(escidx>2) escidx=2; }
// jmax es igual al conteo de escape q
```

La variable *escidx* nos da una aproximación logarítmica de la probabilidad de escape, según la siguiente tabla:

(probhit/probmiss)-1	escidx
< 1	de 0 a 3
1	4
2	5
3	6
4	7
5 a 29	8
30 a 79	9
> 80	10

TABLA 3. Indices para la variable *escidx*

Cuando escidx = 0, es decir, cuando n / q < 2, se cambia la variable "*(n / q) -1*" por "*(3*(n-q)) / q*". Así añadimos dos muy importantes bits de precisión a nuestra aproximación de la probabilidad de escape.

5.7.2. Indices de las matrices secundarias

Adicionalmente, tenemos otros índices para nuestras matrices.

La variable *escidx1* es igual al caracter de orden 1 de nuestro contexto (es decir, el último caracter visto en la cadena).

La variable *escidx2* es igual a los 4 últimos caracteres vistos, cuantizados cada uno a 2 bits según la tabla de reemplazo "charidx".

escidx2=((charidx[m_lastchar]>>1)<<2) | (charidx[m_last4[1]]>>1);

```
escidx2=(escidx2<<2) | (charidx[m_last4[2]]>>1);
escidx2=(escidx2<<2) | (charidx[m_last4[3]]>>1);
```

La tabla de reemplazo charidx arroja los valores del 0 al 7, dependiendo del caracter:

```
0: Del caracter 1 al 31 (excepto los abajo mencionados)
1: separadores (0, tab, LF, fin de hoja, CR, EOF, ESC,
    coma, punto, punto y coma, dos puntos.
2: Símbolos no aritméticos: !, comillas, #, $, apóstrofe, ? , @,
    [, \, ], ^, _, apóstrofe inversa, {, |, }, ~ y el caracter 127.
3: Símbolos aritméticos y números (caracteres 32 al 63 excepto los
arriba mencionados)
4: A-Z
5: a-z
6: espacio, caracteres 128 al 191
7: caracteres del 192 al 255
```

La tabla de reemplazo, discrimina información para los archivos de texto, resultando en 3 bits por caracter (que usaremos en la Estimación Secundaria para los contextos determinísticos). Para reducir a 2 bits por caracter, simplemente eliminamos el bit menos significativo.

Los otros dos índices utilizados son *escidx3* y *escidxq*. El índice *escidxq* se basa en el conteo de escapes q.

```
escidxq=curnode->childcount-1;
if(escidxq>=70) escidxq=8;else if(escidxq>=30) escidxq=7;else
if(escidxq>=17) escidxq=6;else if(escidxq>=10) escidxq=5;else
if(escidxq>=6)  escidxq=4;else if(escidxq>=3)  escidxq=3;else
if(escidxq>=2)  escidxq=2;
```

El índice ordidx, nos indica si ha habido algún escape anterior a este contexto. Finalmente, *escidx3* se basa en la diferencia de escapes entre el contexto actual y el contexto sufijo.

```
if(curnode->suffix) escidx3=curnode->suffix->childcount-1;else
escidx3=255;
escidx3-=curnode->childcount-1;

if(escidx3>=100) escidx3=8;else if(escidx3>=40) escidx3=7;else
if(escidx3>=17)  escidx3=6;else if(escidx3>=10) escidx3=5;else
if(escidx3>=6)   escidx3=4;else if(escidx3>=3)   escidx3=3;else
if(escidx3>=2)   escidx3=2;

      pred1=&seepred1[escidx1][ordidx][escidxq][escidx];
      pred2=&seepred2[escidx2][ordidx][escidxq][escidx];
      pred3=&seepred3[escidx2][ordidx][escidx3][escidx];
```

En base a estos índices manejamos 3 tablas, de 17 bits cada una. Los valores finales de probhit y probmiss se modifican conforme al siguiente código:

```
U32 h1,h2,h3,m1,m2,m3,phit1,pmiss1;

h1=pred1->h;m1=pred1->m;
h2=pred2->h;m2=pred2->m;
if(alphasize>128) { h1*=3;m1*=3; }
// para archivos binarios, la primera tabla adquiere mayor peso,
// ya que depende de los 8 bits del último caracter visto

// la tabla 3 adquiere menor peso y sólo es utilizada cuando el
// número de escapes es menor al de las otras 2 tablas combinadas

h3=pred3->h>>1;m3=pred3->m>>1;
phit1=h1+h2;pmiss1=m1+m2;
if(m3<pmiss1) { phit1+=h3; pmiss1+=m3; }

probhit=probhit<<1;probmiss=probmiss<<1;

// los valores de probhit se multiplican por 2, de tal
// modo que las tablas de SEE sólo adquieren un 33% del
// peso total

probhit+=phit1;probmiss+=pmiss1;

if(!probhit) probhit++;
if(!probmiss) probmiss++;
```

```
// como última modificación, para los casos en sólo haya habido
// un conteo por caracter distinto, se aumenta la probabilidad
// de escape, ya que el contexto tiene una distribución plana.

if(curnode->xp1==jmax) { probmiss+=4; } //4
```

Las tablas de Estimación de Escape Secundaria, se basan en un objeto, denominado *bitpred*, que se encarga del manejo de probabilidades de escape.

```
class bitpred
{
  public:
  unsigned int h:12;
  unsigned int m:12;
  inline void clear() {h=0;m=0;}
  inline void hit() { h++;if(!h) { h=2048;if(m) m=(m >> 1)+1; } }
  inline void miss(){ m++;if(!m) { m=2048;if(h) h=(h >> 1)+1; } }
  inline void put(bool esc){ if(esc) { miss(); } else { hit();} }
  inline U32 total() { return (h+m); }
  inline U32 safetotal()
  { U32 tmp; tmp=h+m; if(!h) tmp++; if(!m) tmp++; return(tmp); }
  inline double phit()
  { return (h ? ((double)h)/safetotal() : 1.0 / safetotal()); }
  inline double pesc()
  { return (m ? ((double)m)/safetotal() : 1.0 / safetotal()); }
  inline U32 localentropy()
  {
    U32 n,hh,mm;double ph,pm;
    hh=h;if(!h) hh=1;mm=m;if(!m) mm=1;
    return inentropy(hh,mm);
  }
  inline U32 totalentropy()
  {
    if(!(h+m)) return 0;
    return ientropy(h,m);
  }
} PACKED;
```

5.8. Otras modificaciones al algoritmo PPMZ

Como última mejora, se añadió una heurística para saltar automáticamente los contextos jóvenes una vez que se ha escapado de algún contexto. Esta toma de decisión se ejecuta antes de la Estimación Secundaria de Escape. Los parámetros fueron estimados en base a prueba y error.

```
if(cnum2){
  if(alphasize>=128)
    { if(13*(curnode->xp1-jmax)<=10*jmax) continue; }
  else
    { if(14*(curnode->xp1-jmax)<=8*jmax) continue;  }
}
```

6. MODELADO DETERMINÍSTICO EN LA COMPRESIÓN PPM

6.1. Introducción

El problema de la frecuencia-cero ha sido investigado desde el surgimiento del algoritmo PPM. En algunos experimentos con cadenas de bits [4], John Cleary y Bill Teahan se percataron de que en un modelo PPM, la frecuencia de escape de los contextos deterministicos (aquellos contextos que sólo han tenido una predicción), es menor a la de los contextos no-deterministicos.

Para codificar un contexto deterministico en un modelo PPM, sólo debemos codificar un bit: Si el caracter predicho es diferente del caracter a codificar, entonces se envía un escape al codificador aritmético. Si el caracter predicho es el mismo caracter a codificar, se codifica un no-escape. La probabilidad de escape determina la compresión de cada bit codificado.

En 1995, Teahan logró implementar un modelo PPM sin límite de orden. Su algoritmo, PPM* [5], obtuvo una mayor compresión comparado con la implementación PPMC de Moffat. Sin embargo, no se logró un aumento significativo en la compresión, debido a la falta de una estimación de orden local: Al procesar todos los contextos no-deterministicos, se desperdiciaban demasiados bits en la codificación de los escapes. Por otro lado, PPM* procesaba todos los contextos deterministicos, incluso aquellos con alta probabilidad de escape.

Para resolver el problema del modelado de contextos largos, Charles Bloom propuso en su compresor PPMZ un límite inferior para el orden de los contextos deterministicos. Si el orden (longitud) de un contexto deterministico no sobrepasa dicho límite, se procede a buscar el contexto más apropiado para la codificación, mediante la Estimación de Orden Local. A pesar de su buen desempeño, el compresor de Bloom no aprovechó al máximo los contextos deterministicos para la compresión, como veremos a continuación.

6.2. Codificación de contextos deterministicos: PPM* y PPMZ

El programa de Teahan obtenía como *predictor* (es decir, un estimador de la probabilidad de escape) de los contextos deterministicos, el conteo con *full update* del contexto deterministico más corto. Por otro lado, Bloom sólo toma en cuenta aquellos contextos con longitud mayor a 12, utilizando una Estimación de Escape Secundaria.

Para obtener el desempeño de ambos programas, modificamos el código de PPMZ (9.1) y PPM* para reportar el número de contextos deterministicos procesados y su codificación para cada archivo. Las pruebas se realizaron sobre el conjunto de archivos ***Calgary Corpus***.

TABLA 4.
Codificación de contextos deterministicos en PPMZ y PPM*

	PPMZ			PPM*	
Archivo	**Contextos procesados**	**Codificación:**		**Contextos procesados**	**Codificación:**	
		bytes	**bits/sim**		**Bytes**	**Bits/sim**
bib	32883	1515.4	0.3687	81192	5369.5	0.5291
book1	56897	5372.8	0.7554	468069	44565.9	0.7617
book2	123752	8682.5	0.5613	418559	33121.5	0.6331
geo	2261	43.2	0.1529	36119	3840.5	0.8506
news	86205	3767.7	0.3496	249531	20445.6	0.6555
obj1	4554	95.3	0.1674	10677	815.4	0.6110
obj2	74382	3007.7	0.3235	166017	11966.2	0.5766
paper1	7903	483.9	0.4898	35332	3152.3	0.7138
paper2	8410	642.2	0.5937	52683	4842.9	0.7326
progc	7079	410.2	0.4636	27061	2270.7	0.6713
progl	30624	1206.2	0.3151	55555	3903.9	0.5622
progp	21046	853.1	0.3243	39555	2614.7	0.5288
trans	51882	1429.3	0.2204	77250	4736.8	0.4905

Después de comparar el desempeño para la codificación de contextos deterministicos, notamos que PPMZ logra una mayor compresión por cada bit de escape codificado; sin embargo, notamos que esto se debe a que procesa una **pequeña fracción** de los contextos deterministicos presentes en cada archivo.

Esto quiere decir que la gran mayoría de contextos deterministicos de un archivo tienen una longitud corta, por lo que es necesario investigar su comportamiento con respecto a diversas variables, como longitud del contexto, la localización del contexto en nuestro árbol sufijo, etc. El contexto determinístico que analizamos siempre será el más largo presente en nuestro árbol sufijo.

6.3. Análisis del comportamiento de los contextos deterministicos

El archivo de prueba que usaremos será paper1, que representa a los archivos de texto pequeños, en los cuales los contextos deterministicos tienen un gran peso sobre la compresión.

Analizaremos 3 variables:
1) La longitud del contexto determinístico más largo
2) El número de sus sufijos determinísticos, incluyendo al mismo contexto
3) La suma de conteos con exclusión de actualizaciones para todos los sufijos determinísticos. Para esta última variable, sólo analizamos a partir del conteo 2 (todos los nodos del árbol tienen conteo mayor o igual a 2)

TABLA 5.
Comportamiento de los contextos determinísticos para PAPER1:

	Según longitud del contexto determinístico			Según # de sufijos determinísticos			Según suma de conteos (con update exclusion)		
	Hits	Misses	P(hit)	Hits	Misses	P(hit)	Hits	Misses	P(hit)
1:	27	68	0.2842	4802	4717	0.5045	--	--	----
2:	489	583	0.4562	3682	1506	0.7097	2950	4167	0.4145
3:	1623	1184	0.5782	3228	1027	0.7586	2333	1563	0.5988
4:	2826	1389	0.6705	2363	652	0.7837	2243	1088	0.6734
5:	2998	1297	0.6980	1852	418	0.8159	2038	716	0.7400
6:	2719	1061	0.7193	1595	330	0.8286	1868	514	0.7842
7:	2348	934	0.7154	1179	208	0.8500	1620	378	0.8108
8:	1991	649	0.7542	936	153	0.8595	1447	257	0.8492
9:	1617	541	0.7493	785	101	0.8860	1271	194	0.8676
10:	1258	393	0.7620	615	82	0.8824	1042	128	0.8906
11:	1051	276	0.7920	561	61	0.9019	907	107	0.8945
12:	832	222	0.7894	441	48	0.9018	857	71	0.9235
13:	694	179	0.7950	302	44	0.8728	727	61	0.9226
14:	599	134	0.8172	290	16	0.9477	623	65	0.9055
15:	487	104	0.8240	269	35	0.8849	516	26	0.9520
16:	398	92	0.8122	217	25	0.8967	455	38	0.9229
17:	351	58	0.8582	189	17	0.9175	413	29	0.9344
18:	294	52	0.8497	159	13	0.9244	372	21	0.9466
19:	268	47	0.8508	150	6	0.9615	322	17	0.9499
20:	221	43	0.8371	135	5	0.9643	312	6	0.9811
21:	199	27	0.8805	110	8	0.9322	261	8	0.9703
22:	177	26	0.8719	102	5	0.9533	238	7	0.9714
23:	153	23	0.8693	105	4	0.9633	237	9	0.9634
24:	130	26	0.8333	91	1	0.9891	197	7	0.9657
25:	111	20	0.8473	80	4	0.9524	173	3	0.9830
26:	101	10	0.9099	85	5	0.9444	178	2	0.9889
27:	93	12	0.8857	62	4	0.9394	164	6	0.9647
28:	87	5	0.9457	82	4	0.9535	131	6	0.9562
29:	86	4	0.9556	76	2	0.9744	115	4	0.9664
30:	77	9	0.8953	62	3	0.9538	91	2	0.9785
31:	74	3	0.9610	48	0	1.0000	101	2	0.9806
32+:	1398	81	0.9452	1124	48	0.9590	1574	50	0.9692

Al analizar los contextos deterministicos de longitud (orden) menor a 12, vemos que su probabilidad de éxito va de 0.28 a 0.79. La eficiencia (relativa) en la compresión por parte del compresor PPMZ se debe a que escoge contextos con probabilidad de éxito mayor a 0.8. Sin embargo, Bloom deja de codificar aproximadamente 28 mil 35 mil contextos deterministicos en total, es decir, más del 80%.

Utilizando otras variables para predecir la probabilidad de escape, podríamos aprovechar una mayor cantidad de contextos deterministicos de nuestro archivo. Al utilizar la fórmula de la entropia vista en el capítulo 1, podemos obtener la cantidad de información (en bytes) para nuestras 3 variables *predictoras*.

6.3.1. Cantidad de información

Por ejemplo, los contextos deterministicos de orden 1 tuvieron 27 *hits* (éxitos) y 68 *misses* (escapes). Es decir, 27 bits con probabilidad de ocurrencia de 0.2842, y 68 bits con probabilidad de ocurrencia de (1 - 0.2842).

La cantidad de información, en bits para estos escapes, será: 27* - log2(0.2842) + 68 * log2(1 - 0.2842) = 49.0055 + 32.8012 = 81.85 bits.

Repitiendo la fórmula para todos los contextos deterministicos, tenemos que:

Para la **longitud**, tenemos 28437.78 bits de información = 3554 bytes.

Para el **número de sufijos deterministicos:** 26683.43 bits = 3336 bytes.

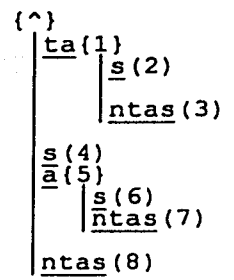
Para la suma de **conteos deterministicos**: 26212.76 bits = 3277 bytes.

Una vez analizada la información para nuestras 3 variables, vemos que la variable más apropiada para codificar contextos deterministicos será la suma de los conteos con *update exclusion*. Sin embargo, comparando con el conteo del contexto deterministico más corto, vemos que PPM* utiliza sólo 3152.3 bytes para 35,331 contextos deterministicos. Para mejorar esta marca, necesitamos utilizar otras variables para predecir los escapes deterministicos. Afortunadamente, el árbol sufijo nos provee con información adicional que no es accesible a los algoritmos PPMZ y PPM*.

Debemos aclarar que se estableció un límite a la cantidad de sufijos deterministicos, para ahorrar tiempo de procesamiento en determinados archivos (p.e. pic, que tiene largas repeticiones del mismo caracter). El límite establecido fue de 64 contextos.

6.4. Nodos hoja y nodos internos

En un árbol sufijo, los contextos deterministicos son representados por vértices entre dos nodos. Y estos vértices, pueden terminar en nodos internos o en nodos hoja.



Por ejemplo, analicemos en el árbol correspondiente a la palabra "tantas", el vértice que une a la raíz {^} con el nodo {1}. El contexto "t" es un contexto determinístico, cuyo vértice termina en un nodo interno.

Los demás contextos determinísticos terminan en los nodos hoja (3), (7) y (8).

Hemos encontrado que la probabilidad de escape de los contextos determinísticos, varía dependiendo si éstos terminan en un nodo hoja o en un nodo interno. Por convención, llamaremos a los primeros "***contextos determinísticos - hoja***" y a los otros, "***contextos determinísticos internos***".

Se hicieron pruebas sobre varios archivos del *Calgary Corpus*, agrupando los contextos determinísticos en 14 renglones, según su suma de conteos, desde 2 hasta 15 ó mayor a 15, y en dos columnas, dependiendo si el contexto determinístico más corto es *hoja* o *interno*. Los contextos correspondientes a repeticiones del mismo caracter se toman en cuenta como *internos*, siempre y cuando el número de repeticiones sobrepase el umbral *MIN_RLE* (esto indica que la probabilidad de escape es muy baja).

TABLA 6. Probabilidades de Éxito p(hit) según el tipo de contexto

	PAPER1		PAPER2		BOOK1		BOOK2		TRANS	
cnt	hoja	int.	hoja	int	hoja	int	hoja	int	hoja	int
2 :	0.385	0.552	0.385	0.552	0.370	0.611	0.403	0.581	0.404	0.502
3 :	0.526	0.692	0.526	0.692	0.408	0.679	0.454	0.679	0.535	0.699
4 :	0.555	0.786	0.555	0.786	0.427	0.739	0.492	0.751	0.564	0.801
5 :	0.576	0.850	0.576	0.850	0.451	0.786	0.507	0.810	0.670	0.836
6 :	0.616	0.869	0.616	0.869	0.474	0.821	0.545	0.844	0.632	0.859
7 :	0.609	0.899	0.609	0.899	0.504	0.857	0.559	0.874	0.684	0.877
8 :	0.655	0.916	0.655	0.916	0.501	0.872	0.569	0.899	0.703	0.923
9 :	0.652	0.934	0.652	0.934	0.525	0.909	0.578	0.910	0.734	0.941
10 :	0.686	0.944	0.686	0.944	0.523	0.919	0.606	0.921	0.797	0.954
11 :	0.674	0.943	0.674	0.943	0.531	0.935	0.622	0.937	0.789	0.957
12 :	0.710	0.958	0.720	0.965	0.575	0.942	0.663	0.947	0.844	0.944
13 :	0.720	0.965	0.710	0.958	0.580	0.955	0.673	0.961	0.788	0.971
14 :	0.652	0.943	0.652	0.943	0.567	0.965	0.675	0.961	0.756	0.970
15+:	0.857	0.981	0.857	0.981	0.717	0.990	0.784	0.987	0.955	0.990

ARCHIVOS BINARIOS:

	GEO		OBJ1		OBJ2		PIC	
cnt	hoja	int	hoja	int	hoja	int	hoja	int
2 :	0.303	0.660	0.322	0.379	0.347	0.473	0.461	0.818
3 :	0.413	0.848	0.416	0.683	0.503	0.657	0.505	0.816
4 :	0.483	0.726	0.592	0.798	0.585	0.761	0.611	0.873
5 :	0.409	0.901	0.646	0.868	0.640	0.816	0.640	0.909
6 :	0.500	0.944	0.662	0.909	0.691	0.872	0.633	0.926
7 :	0.500	0.984	0.658	0.922	0.681	0.908	0.582	0.938
8 :	0.625	0.982	0.672	0.935	0.742	0.924	0.687	0.943
9 :	0.875	0.988	0.721	0.986	0.732	0.939	0.695	0.953
10 :	0.800	0.967	0.710	0.962	0.774	0.945	0.623	0.955
11 :	0.333	0.969	0.794	0.961	0.758	0.955	0.724	0.954
12 :	0.933	0.983	0.826	0.969	0.762	0.960	0.663	0.954
13 :	0.571	0.984	0.565	0.966	0.770	0.963	0.729	0.958
14 :	0.833	0.983	0.783	0.969	0.859	0.965	0.669	0.959
15+:	0.811	0.995	0.813	0.995	0.909	0.990	0.903	0.999

Sin importar el tipo de archivo: texto, ejecutable, de tipo aleatorio (GEO) o gráfico (PIC); o su tamaño (pequeño, grande), el comportamiento de los contextos deterministicos es similar: La probabilidad de éxito de un contexto determinístico aumenta con la suma de conteos, y siempre será mayor si el contexto deterministico más corto termina en un nodo interno.

Comparemos los datos de PAPER1 con los de la tabla 5 (columna 3):

TABLA 6A.
DATOS DESGLOSADOS
PARA PAPER1

	total	hoja	int.
2 :	0.4145	0.3849	0.5524
3 :	0.5988	0.5265	0.6917
4 :	0.6734	0.5554	0.7859
5 :	**0.7400**	**0.5756**	**0.8502**
6 :	0.7842	0.6165	0.8687
7 :	0.8108	0.6092	0.8992
8 :	0.8492	0.6553	0.9163
9 :	0.8676	0.6522	0.9339
10 :	0.8906	0.6860	0.9440
11 :	0.8945	0.6739	0.9434
12 :	0.9235	0.7197	0.9650
13 :	0.9226	0.7105	0.9585
14 :	0.9055	0.6517	0.9432

Analicemos los datos para el conteo 5:

Si no tomamos en cuenta el tipo de contexto determinístico, tendremos una probabilidad de éxito general de 74%. En cambio, al diferenciar los contextos deterministicos internos, podemos garantizar una probabilidad de éxito del 85%, y para los contextos hoja tendremos un 57%.

Y para el conteo 14, podemos distinguir los contextos con probabilidad de éxito de 65% de aquellos con 94%.

Esto quiere decir que, además de detectar contextos deterministicos con una gran probabilidad de éxito, podemos ahorrar bits en la codificación de los

escapes más probables (contextos deterministicos hoja), o incluso descartarlos si la probabilidad de éxito es demasiado baja (esto quiere decir que muy probablemente se codificará un escape, por lo cual será mejor utilizar directamente la Estimación de Orden local).

6.5. Contextos deterministicos muy jóvenes.

Veamos más de cerca los datos de los contextos deterministicos hoja con conteo 2 (tabla 6). Estos contextos poseen 3 caracteristicas:

1) El conteo 2 nos dice que para un contexto deterministico no existen sufijos deterministicos.

2) Por otro lado, al tener el conteo mínimo y terminar en un nodo hoja, podemos confirmar que es un contexto joven, y muy probablemente tengamos que escapar al sufijo no-deterministico.

3) Este tipo de contextos son de los más frecuentes en el modelado PPM (en la tabla 5, columna 3, los contextos con conteo 2 presentan más de 4000 escapes - suponiendo que el 50% son contextos hoja, tendriamos aproximadamente 2000 escapes, un número relativamente alto).

Si queremos modelar mejor este tipo de contextos, necesitaremos de más información. Una buena aproximación es utilizar la longitud del contexto.

Por otro lado, podemos aprovechar la información del sufijo no-deterministico si utilizamos un poco de lógica. Al conocer el símbolo más probable (MPS) del contexto sufijo, podemos darnos una idea de la probabilidad de escape. ¿Este símbolo es el mismo símbolo predicho por nuestro contexto deterministico? ¿Cuál es su probabilidad de ocurrencia? Con esto podremos evitar codificar escapes que serán muy probables.

Por ejemplo, si nuestro contexto deterministico predice el caracter "s", y el contexto sufijo tiene el caracter más probable "t" con una probabilidad de ocurrencia del 50% o 60%, no tiene caso seguir el análisis de nuestro contexto deterministico y procederemos a la Estimación de Orden Local (LOE).

Experimentalmente, se ha encontrado efectiva la siguiente heurística para los contextos determinísticos jóvenes:

Si el MPS del sufijo no-determinístico difiere, y si:

```
a) Su probabilidad de ocurrencia es mayor a 1/2, o bien:
b) si es mayor a 1/8, y
además el orden del contexto determinístico >= 4,
entonces saltar el modelado determinístico y
proceder al algoritmo LOE.
```

6.6. Estimación de Escape Secundaria para los contextos deterministicos.

Además de la suma de conteos deterministicos y la distinción de contextos deterministicos internos, podemos mejorar nuestra Estimación de Escape utilizando los últimos caracteres del contexto.

Por ejemplo, si el último caracter del contexto (es decir, el caracter correspondiente a un contexto de orden-1) es un caracter separador (espacio, punto, coma, escape), la probabilidad de escape de nuestro contexto determinístico será más alta, que si el último caracter es por ejemplo, una letra o número.

Originalmente se utilizaron los 4 últimos caracteres del contexto para construir una matriz de probabilidades, pero se encontró que si reemplazamos el caracter de orden-4 (es decir, el anterior al antepenúltimo del contexto) por el caracter predicho, la compresión aumentó significativamente.

Al igual que con la SEE para contextos no-deterministicos, tenemos varias matrices de predicción (4). El elemento básico de cada matriz será una tabla de 3 columnas x 8 renglones.

```
  | A B C
=#=======
0| * b c
1| a b c
2| a b c
3| a b -
4| a b -
5| a b -
6| a b -
7| a b -
```

Columna A: contextos deterministicos hoja.
Columna B: contextos deterministicos internos.
Los renglones para A y B dependen de la suma de conteos deterministicos:

indice	0	1	2	3	4	5	6	7
conteo	2	3	4	5	6	7-14	15-71	72+

Los *contextos determinísticos jóvenes* (columna A, renglón 0) son desglosados en la columna C según el orden del contexto (0, 1, 2+)

Las matrices secundarias son las siguientes:

- **detpred** - 8 bits: Los 6 bits más significativos del caracter de orden 1; y dos bits para el caracter predicho (los 2 bits más significativos de la tabla de reemplazo ***charidx*** (ver *capítulo 5, "indices de las matrices secundarias"*).
- **detpred2** - 1 elemento (matriz global)
- **detpred3** - 6 bits: 3 bits para el caracter de orden 1, y 3 bits para el caracter predicho (tabla *charidx*)
- **detpred4** - 12 bits: 3 bits para cada caracter de orden 1,2,3; y 3 bits para el caracter predicho. (tabla *charidx*)

Cada tabla arrojará, según los datos del contexto, un valor de probabilidad de éxito (h) y otro valor para la probabilidad de escape (m). Así tendremos 4 parejas de valores: h1/m1, h2/m2, h3/m3, y h4/m4 que tendrán distinto peso.

Para aprovechar eficazmente la información de la tabla, se sumarán los valores de las distintas columnas y renglones cuando el conteo h+m esté por debajo de cierto límite (los límites fueron encontrados experimentalmente). Los valores finales de h,m son calculados mediante la siguiente función:

```
void gethm(bitpred *pred,unsigned ix,unsigned iy,
           U32 &desthit,U32 &destmiss,U32 lim)
{ U32 h,m;
  bitpred *tmppred;

  tmppred=pred+ix*PREDROWS+iy;
  h=tmppred->h;
  m=tmppred->m;

  if(iy && h+m<lim){
    tmppred=pred+ix*PREDROWS+iy-1;
    h+=tmppred->h;
    m+=tmppred->m;
  }
  if(iy+1<PREDROWS && h+m<lim){
    tmppred=pred+ix*PREDROWS+iy+1;
    h+=tmppred->h;
    m+=tmppred->m;
  }
```

```
    if(ix==1 && h+m<lim){
    tmppred=pred+(ix-1)*PREDROWS+iy;
    h+=tmppred->h;
    m+=tmppred->m;
  }
  while(h+m>=16382){
    h>>=1;
    m>>=1;
    if(!h) h++;
    if(!m) m++;
  }

  desthit=h;
  destmiss=m;
}
```

Finalmente, las 4 variables son sumadas con distintos pesos (también encontrados experimentalmente), para dar como resultado la probabilidad final de escape (phit,pmiss).

El código para el cálculo de las variables es el siguiente:
(La variable ***escidx*** corresponde a las columnas, y ***escidx2*** a los renglones)

```
U32 h1,m1,h2,m2,h3,m3,h4,m4,tmpidx,tmpidx1,tmpidx3;


tmpidx=charidx[curchar];
tmpidx3=tmpidx=(tmpidx << 3) | (charidx[m_lastchar]);
tmpidx=(tmpidx << 3) | (charidx[m_last4[1]]);
tmpidx=(tmpidx << 3) | (charidx[m_last4[2]]);

tmpidx1=((m_lastchar) & 252) | (charidx[curchar]>>1);

mypred=&detpred[tmpidx1][escidx][escidx2];
mypred2=&detpred2[escidx][escidx2];
mypred3=&detpred3[tmpidx3][escidx][escidx2];
mypred4=&detpred4[tmpidx][escidx][escidx2];

gethm(&detpred [tmpidx1][0][0],escidx,escidx2,h1,m1,16);
gethm(&detpred2          [0][0],escidx,escidx2,h2,m2,20);
gethm(&detpred3[tmpidx3][0][0],escidx,escidx2,h3,m3,16);
gethm(&detpred4[tmpidx ][0][0],escidx,escidx2,h4,m4,12);

phit=0;pmiss=0;

h2>>=2;m2>>=2;          // h2,m2 tendrá un peso de 0.25
h3>>=1;m3>>=1;          // h3,m3 tendrá un peso de 0.5
h4+=h4<<1;m4+=m4<<1;    // h4,m4 tendrá un peso de 2.0
```

```
// Las distintas variables sólo serán añadidas dependiendo de su
// conteo de escape y de la suma de phit, pmiss.

phit= h4;pmiss= m4;
if(m4>m1 || phit+pmiss<20) { phit+=h1;pmiss+=m1; }
if(m1>m3 || phit+pmiss<20) { phit+=h3;pmiss+=m3; }
if(phit+pmiss<=10)         { phit+=h2;pmiss+=m2; }
```

6.7. Postprocesamiento de la probabilidad de escape

Una vez obtenidas phit,pmiss verificamos que si la probabilidad de escape es lo suficientemente alta como para saltar el contexto determinístico.

```
if((pmiss>=20 && phit<=20 && pmiss>=10*phit)) goto DETMISS;

if(escidx!=2 && pmiss>=6)
  // Aproximación de Ventana deslizante basada en pmiss
  // Al escalar los conteos después de un número determinado de
  // escapes, damos un mayor peso a los datos más recientes
 {
    if(pmiss>=40 && phit<=1) goto DETMISS;
    // Si sólo hemos recibido un hit y muchos escapes, entonces
    // asumimos que P(esc)=1
    mypred2->h-=mypred2->h>>4;mypred2->m-=mypred2->m>>4;
    mypred3->h-=mypred3->h>>4;mypred3->m-=mypred3->m>>4;
    if(pmiss>=20)
    {
      // después de un número determinado de escapes, disminuimos
      // el conteo total dividiéndolo entre 4.
      mypred2->h-=mypred2->h>>2;mypred2->m-=mypred2->m>>2;
      mypred3->h-=mypred3->h>>2;mypred3->m-=mypred3->m>>2;
    }
 }
```

Posteriormente hacemos una segunda estimación de escape, basada **exclusivamente** en el conteo determinístico y el número de hojas del nodo hijo (dependiendo de si nuestro contexto determinístico es interno).

```
 phit+=longcnt*numleaves;
  // numleaves es el número de hijos del nodo final de nuestro
  // contexto determinístico - igual a 1 si es nodo hoja.
  // Se encontró que la probabilidad de escape disminuye conforme
  // aumenta el número de hijos del nodo.
 pmiss++;
```

Finalmente, tomamos en cuenta si en la predicción del último caracter hubo un escape. Si el caracter anterior fue predicho sin necesidad de escapes, es

probable que no se requiera un escape para predecir el siguiente caracter.

```
if(lastescaped) { pmiss+=3; phit-=phit>>3; }
```

Una vez obtenidos los valores final de phit y pmiss, se utilizan para la (de)codificación aritmética del bit de escape, actualizando las matrices de Estimación de Escape Secundaria.

6.8. Resultados preliminares

A continuación publicamos el número de bits requeridos para codificar los bits de escape de contextos deterministicos en los archivos del *Calgary Corpus*. Para una justa comparación con PPM* y PPMZ (tabla 4), se procesaron todos los contextos deterministicos, incluso aquellos con alta probabilidad de escape.

	PPMZ			PPM*		
Archivo	**Contextos procesados**	**Codificación:**		**Contextos procesados**	**Codificación:**	
		bytes	**bits/sim**		**Bytes**	**Bits/sim**
bib	32883	1515.4	0.3687	81192	5369.5	0.5291
book1	56897	5372.8	0.7554	468069	44565.9	0.7617
book2	123752	8682.5	0.5613	418559	33121.5	0.6331
geo	2261	43.2	0.1529	36119	3840.5	0.8506
news	86205	3767.7	0.3496	249531	20445.6	0.6555
obj1	4554	95.3	0.1674	10677	815.4	0.6110
obj2	74382	3007.7	0.3235	166017	11966.2	0.5766
paper1	7903	483.9	0.4898	35332	3152.3	0.7138
paper2	8410	642.2	0.5937	52883	4842.9	0.7326
pic*	N/A	N/A	N/A	381648	13970.5	0.2928
progc	7079	410.2	0.4636	27061	2270.7	0.6713
progl	30624	1206.2	0.3151	55555	3903.9	0.5622
progp	21046	853.1	0.3243	39555	2614.7	0.5288
trans	51882	1429.3	0.2204	77250	4736.8	0.4905

* El archivo pic no se procesó con PPMZ debido a la cantidad de tiempo requerida para procesarlo (En más de 6 horas el programa sólo habia procesado menos del 1%).

TABLA 7. Codificación de contextos determinísticos con el Compresor experimental

archivo	contextos procesados	codificación (bytes)	(bits / sim)
bib	81189	4196.1	0.4135
book1	468066	38521.0	0.6584
book2	418557	28535.7	0.5454
geo	36117	1698.3	0.3762
news	249528	15738.3	0.5046
obj1	11724	656.9	0.4482
obj2	166014	8832.7	0.4256
paper1	35329	2601.1	0.5890
paper2	52881	4066.1	0.6151
pic	423967	7574.2	0.1429
progc	27059	1886.4	0.5577
progl	55552	2839.5	0.4089
progp	39553	1889.0	0.3821
trans	77247	2833.4	0.2934

Como podemos ver, la codificación de contextos determinísticos es superior en todos los casos a PPM*. Los resultados de PPMZ aparentemente son superiores debido a la pequeña cantidad de contextos procesados.

7. RESULTADOS GLOBALES

A continuación comparamos nuestro compresor experimental con el compresor PPMZ-2 (versión mejorada de PPMZ), y otros compresores comerciales, como ZIP y BZIP.

Antes de dar los resultados de la comparación, daremos una introducción al conjunto de archivos a ser comparados.

7.1. Introducción al Calgary Corpus

El Calgary Corpus es un conjunto de archivos de prueba diseñado para comparar diversos compresores de datos. Consiste de archivos de diversos tamaños y tipos.

BIB - Archivo de texto de tamaño mediano, consiste en una colección de fichas bibliográficas - tamaño: 111K

BOOK1 - Archivo de texto de tamaño grande - Es la digitalización de un libro de Thomas Hardy, *"Far from the Madding Croud"*. Tamaño: 768K.

BOOK2 - Archivo de texto de tamaño grande - digitalización de un tomo científico. "*Why Speech Output? Principles of computer speech*". Tamaño: 610K

GEO - Archivo binario de tamaño mediano - lista de datos con alta aleatoriedad, de 32 bits. La mayoría de estos datos tienen el byte más significativo igual a cero.

NEWS - Archivo de texto de tamaño mediano - compilación de artículos encontrados en usenet. Tamaño: 377K

OBJ1 - Archivo binario de tamaño pequeño - archivo ejecutable para VAX. Tamaño: 21K

OBJ2 - Archivo binario de tamaño mediano - archivo ejecutable para VAX. Tamaño: 246K

PAPER1 - Archivo de texto de tamaño pequeño - Este archivo, curiosamente, es el artículo original de Neal, Cleary y Witten sobre la codificación aritmética para la compresión de datos. Tamaño: 53K

PAPER2 - Archivo de texto de tamaño mediano - Artículo de Witten sobre la inseguridad en los sistemas de computadoras, publicado en la Universidad de Calgary. Tamaño: 82K

PIC - Archivo binario de tamaño grande - Una imagen de fax en blanco y negro. Una característica de este archivo, es que si se rota la imagen 90 grados, el archivo se puede comprimir hasta aproximadamente 30K - algunos compresores toman en cuenta esta característica para obtener mejores resultados, pero esto es aprovechar conocimiento previo para la compresión.

El archivo PIC contiene largas secuencias de 0's, lo cual pone a prueba la capacidad de los compresores para comprimir archivos con baja entropía.

PROGC - Archivo de texto de tamaño pequeño - código fuente en C de un compresor del tipo LZW para VAX. Tamaño: 39K

PROGL - Archivo de texto de tamaño mediano - código fuente en LISP. Tamaño: 71K

PROGP - Archivo de texto de tamaño pequeño - código fuente en pascal. Tamaño: 49K

TRANS - Archivo de texto de tamaño mediano - Es la copia de una transmisión vía modem de una sesión en UNIX. Este archivo se caracteriza por contener una gran cantidad de cadenas repetidas, con lo cual se obtiene una gran compresión al aprovechar el modelado de contextos determinísticos. Tamaño: 93K

El Calgary Corpus tiene además 4 archivos, PAPER3, PAPER4, PAPER5 y PAPER6 - pero no son reportados en artículos de compresión debido a que son muy similares en tamaño y estructura a PAPER1 o PAPER2.

Cabe notar que existe un conjunto más actualizado de archivos de prueba para compresión de datos, denominado Canterbury Corpus. Sin embargo, el Calgary Corpus sigue siendo el conjunto más popular, debido en parte a que fue el conjunto de prueba para la mayoría de compresores PPM (PPMC, PPM*, PPMD). Por lo tanto éste será el conjunto que usaremos para nuestras pruebas.

Los archivos del Calgary Corpus y Canterbury corpus pueden ser encontrados en la dirección web de la Universidad de Canterbury:

http://corpus.canterbury.ac.nz

7.2. Resultados de compresión: Calgary Corpus

7.2.1. Comparación con diversos compresores, detallada

Compresión (tamaño en bytes del archivo comprimido)

Archivo	Original	ZIP -9	BZIP2 -9	PPMZ2	nuestro compresor
bib	111,261	34,878	27,467	**23,873**	23,874
book1	768,771	312,257	232,598	**210,952**	212,101
book2	610,856	206,134	157,443	140,932	**140,701**
geo	102,400	68,392	56,921	58,578	**54,566**
news	377,109	144,377	118,600	103,951	**103,808**
obj1	21,504	10,297	10,787	9,841	**9,559**
obj2	246,814	81,064	76,441	69,137	**67,014**
paper1	53,161	18,518	16,558	14,711	**14,531**
paper2	82,199	29,642	25,041	22,449	**22,440**
pic	513,216	52,359	49,759	48,193	**46,983**
progc	39,611	13,237	12,544	11,178	**10,927**
progl	71,646	16,140	15,579	12,938	**12,752**
progp	49,379	11,162	10,710	8,948	**8,773**
trans	93,695	18,838	17,899	14,224	**13,964**
TOTAL	3,141,622	1,017,295	828,347	749,905	**741,993**

Compresión en porcentaje (porcentaje del archivo comprimido sobre el original)

Archivo	ZIP -9	BZIP2 -9	PPMZ2	nuestro compresor
bib	31.35%	24.69%	21.46%	21.46%
book1	40.62%	30.26%	27.44%	27.59%
book2	33.75%	25.77%	23.07%	23.03%
geo	66.79%	55.59%	57.21%	53.29%
news	38.29%	31.45%	27.57%	27.53%
obj1	47.88%	50.16%	45.76%	44.45%
obj2	32.84%	30.97%	28.01%	27.15%
paper1	34.83%	31.15%	27.67%	27.33%
paper2	36.06%	30.46%	27.31%	27.30%
pic	10.20%	9.70%	9.39%	9.15%
progc	33.42%	31.67%	28.22%	27.59%
progl	22.53%	21.74%	18.06%	17.80%
progp	22.60%	21.69%	18.12%	17.77%
trans	20.11%	19.10%	15.18%	14.90%
TOTAL	32.38%	26.37%	23.87%	23.62%
PROMEDIO	33.66%	29.60%	26.75%	26.17%

Comparación: Compresión en bits por caracter
(bits utilizados por caracter original)

Archivo	ZIP -9	BZIP2 -9	PPMZ2	nuestro compresor
bib	2.5078	1.9750	**1.7165**	1.7166
book1	3.2494	2.4205	**2.1952**	2.2072
book2	2.6996	2.0619	1.8457	**1.8427**
geo	5.3431	4.4470	4.5764	**4.2630**
news	3.0628	2.5160	2.2052	**2.2022**
obj1	3.8307	4.0130	3.6611	**3.5562**
obj2	2.6275	2.4777	2.2409	**2.1721**
paper1	2.7867	2.4918	2.2138	**2.1867**
paper2	2.8849	2.4371	2.1848	**2.1840**
pic	0.8162	0.7756	0.7512	**0.7324**
progc	2.6734	2.5334	2.2576	**2.2069**
progl	1.8022	1.7396	1.4447	**1.4239**
progp	1.8084	1.7352	1.4497	**1.4213**
trans	1.6085	1.5283	1.2145	**1.1923**
TOTAL	2.5905	2.1093	1.9096	**1.8895**
PROMEDIO	2.6929	2.3680	2.1398	**2.0934**

7.2.2. Comparación con diversos compresores, compresión y tiempo total (sin detallar)

Las pruebas se efectuaron en una Pentium a 166 MHz, con Windows 95, y 64 MB en RAM.

ZIP (opción -9, máxima compresión)

El compresor más popular, utiliza una variante del algoritmo LZ77.

Tiempo de compresión: 14.67 s. Compresión: 33.66%

BZIP (opción -9, máxima compresión)
Este compresor utiliza la transformación BWT. Esta transformación se basa también en el uso de árboles sufijos.

Tiempo de compresión: 14.51 s. Compresión: 29.60%

PPMZ
El compresor de investigación de Charles Bloom fue el estado del arte en compresión PPM al ser liberado en 1996. La versión probada es la 9.1, con preprocesamiento RLE activado.

Tiempo de compresión: 6 minutos, 6.19 segundos. Compresión: 24.19%

PPMZ-2
Última versión de PPMZ, liberada en 1999. Incluye una versión exhaustiva de la Estimación de Orden Local, más una Estimación de Escape Secundaria para los contextos deterministicos.

Tiempo de compresión: El programa no se pudo correr debido a los requerimientos de memoria.
Compresión: 23.87%

Compresor Experimental, máxima compresión
Memoria máxima utilizada: 29 MB
Tiempo de compresión: 3 minutos 9.08 segundos.
Compresión: 23.62%

Compresor Experimental. Límite de orden: 255.
Memoria máxima utilizada: 26 MB.
Tiempo de compresión: 2 minutos 30.27 segundos.
Compresión: 23.63%

Compresor Experimental. Límite de orden: 8.
Memoria máxima utilizada: 16.3 MB
Tiempo de compresión: 2 minutos 8.20 segundos
Compresión: 24.31%

Compresor Experimental. Límite de orden: 255. Ventana: 128 KB
Memoria máxima utilizada: 5.71 MB
Tiempo de compresión: 2 minutos 23.85 segundos
Compresión: 25.64%

Compresor Experimental. Límite de orden: 8. Ventana: 128 KB
Memoria máxima utilizada: 4.47 MB
Tiempo de compresión: 2 minutos 5.83 segundos
Compresión: 26.19%

8. CONCLUSIONES

1. El uso del árbol sufijo con una ventana finita hace posible la implementación de un compresor del tipo PPM con bajos requerimientos de memoria RAM, que pueden ir de los 3 MB hasta los 32 MB. La mayoría de las computadoras personales poseen esta capacidad.

2. El modelado de los contextos determinísticos es esencial para el buen desempeño de un compresor PPM.

3. Nuestro compresor no fue diseñado para su aplicación comercial, por lo que es perfectible en términos de uso de memoria y velocidad de procesamiento.

4. Mediante métodos sencillos de toma de decisiones se puede obtener una compresión 4:1, en contraste con la aproximación exhaustiva de compresores como PPMZ.

9. TRABAJO A FUTURO

1. La compresión universal de datos es un problema abierto. Recientemente se han descubierto estructuras de datos con requerimientos de memoria lineales y capaces de manejar la misma información que un árbol sufijo.

Una de estas estructuras, la DAWG (Direct Acyclic Word Graph - Gráfica de palabras acíclica directa), y su versión compacta, la CDAWG, puede ser también utilizada en los algoritmos PPM. Sin embargo, el comportamiento de los contextos determinísticos en este tipo de estructuras no se ha analizado.

2. En esta tesis hemos aplicado y mejorado el algoritmo PPMZ para la compresión de datos. Una de las bases de este algoritmo es la elección adecuada de un contexto para la codificación (Estimación de Orden Local). Sin embargo, queda la posibilidad de combinar las estadísticas de más de un contexto para obtener una mejor compresión. El problema es desarrollar un algoritmo capaz de hacer esto y al mismo tiempo mantener una baja complejidad en el procesamiento.

REFERENCIAS

La mayoría de los siguientes artículos se pueden encontrar en http://www.researchindex.org

[1] Ian H. Witten, Radford M. Neal, John G. Cleary (1987)
"Arithmetic Coding for Data Compression"
CACM 30(6):520-541, Jun 1987.
Disponible en el archivo "PAPER1" del Calgary Corpus.

[2] Cleary, J.G. y Witten, I.H. (1984)
"Data compression using adaptive coding and partial string matching"
IEEE Transactions on Communications, 32(4), 396-402

[3] Moffat, A. (1990)
"Implementing the PPM data compression scheme"
IEEE Transactions on Communications, 38(11), 1917-1921

[4] Cleary, J.G. y Teahan, W.J. (1995)
"Experiments on the zero frequency problem"
Proceedings DCC '95.
IEEE Computer Society Press, 1995.

[5] Cleary, J.G. y Teahan, W.J. (1995).
"Unbounded length contexts for PPM"
Proceedings DCC '95.
IEEE Computer Society Press, 1995.

[6] **Calgary corpus**
http://corpus.canterbury.ac.nz

[7] Lynch, Thomas J. (1985)
"Data Compression: Techniques and Applications"
Lifetime Learning Publications, Belmont, CA, 1985

[8] E. M. McCreight (1976)
"A Space-Economical Suffix Tree Construction Algorithm"
Journal of the ACM, 23(2):262-272, 1976

[9] E. Ukkonen (1995)
"Online Construction of Suffix Trees"
Algorithmica, 14(3):249-260, 1995

[10] N. J. Larsson (1999)
"Structures of String Matching and Data Compression"
PhD thesis, Dept. of Comp. Science, Lund Univ., 1999
Disponible en http://www.researchindex.org

[11] Stefan Kurtz (1999)
"Reducing the Space Requirement of Suffix Trees"
Software - Practice and Experience 29(13):1149-1171, 1999

[12] Matt Timmermans (2000)
"BICOM: Bijective Compressor"
http://www3.sympatico.ca/mtimmerm/bicom/bicom.html

[13] Charles Bloom
"PPMZ: High Compression Markov Predictive Coder"
http://www.cbloom.com/src/ppmz.html