Notas sobre Compresión de Datos

Miguel Morales Sandoval

INAOE, 2003.
mmorales@inaoep.mx

Estas notas presentan algunos de los métodos de compresión de datos más utilizados. Se hace énfasis en los métodos basados en diccionario para compresión de texto. Se proporciona bibliografía para que el lector profundice sobre temas particulares.

METODOS DE COMPRESIÓN

Información y entropía

La *Teoría de la Información* [Sha48] es la disciplina que se encarga del estudio y cuantificación de los procesos que se realizan sobre la *información*. La cantidad de información que nos proporciona cierto dato es menor cuanto más esperamos ese dato. La idea de la *probabilidad de ocurrencia* de cierto evento que nos da información puede modelarse por una variable aleatoria y su conjunto de mensajes y probabilidades asociadas. Al recibir un mensaje de esa variable aleatoria (fuente de información), se obtiene una cantidad de información, que depende sólo de la probabilidad de emisión de ese mensaje. La cantidad de información es una función creciente e inversa a la probabilidad, es decir, un menaje con menor probabilidad proporciona mayor cantidad de información. En 1948, Claude Shannon propuso una medida para la información [Sha48]. Si se supone una fuente F de información de n mensajes F_i, cada uno con probabilidad p_i, la información al recibir un mensaje esta dado por la ecuación:

$$I(F = F_i) = -\log(p_i)$$

La medida media de información o *entropía* de **F** esta dada por la esperanza matemática de la información de cada símbolo

$$H(F) = \sum_{i=1}^{n} p_i * I(F = F_i) = -\sum_{i=1}^{n} p_i * \log(p_i)$$

Al utilizar logaritmo en base 2, la medida estará en bits. Esta medida da una cota superior teórica de la cota de compresión que puede obtenerse de la información, no se puede tener ninguna codificación que consiga una longitud en bits media por símbolo emitido menor que la entropía de la fuente sobre la que se realiza la codificación. Los compresores actuales no llegan a esta cota pero quedan muy cerca.

Compresión de datos

En los últimos años se ha dado un aumento tanto de la capacidad de almacenamiento de datos como en la velocidad de procesamiento en las computadoras. Junto con esto, la tendencia es la disminución de costos en memoria principal y secundaria así como también un aumento de velocidad de estos dispositivos de almacenamiento. Estos acontecimientos ponen en cuestionamiento la necesidad de compresión de datos. Sin embargo, el auge que últimamente han tenido las redes de computadoras, demanda más prestaciones que están por encima de las posibilidades reales. El principal problema al que se enfrentan las redes de comunicación es la velocidad de transferencia de datos. El cambio a mayores velocidades no es tarea fácil, básicamente por razones como la no factibilidad de realizar cambios de infraestructura en las grandes

compañías de redes WAN (cableado, tecnologías, etc.) así como la falta de tecnología que acepte unas velocidades muy elevadas de transmisión.

En este entorno, para conseguir mayores prestaciones de velocidad, se debe recurrir a técnicas que les permitan superar de alguna manera las deficiencias físicas de la red. La técnica más importante en este sentido es la compresión de datos. La compresión de datos es beneficiosa en el sentido de que el proceso de compresión-transmisión-descompresión es más rápido que el proceso de transmisión sin compresión. La compresión de datos no sólo es para la transmisión de datos, si no también para el almacenamiento masivo. La necesidad de almacenamiento también crece por encima de las posibilidades del crecimiento de los discos duros o memoria. Por ejemplo, aplicaciones actuales como el proyecto del Genoma Humano o los servidores de vídeo en demanda, requieren de varios Gigabytes de almacenamiento. La motivación para aplicar compresión a los datos es la reducción de costos tanto en almacenamiento (se requiere menos espacio) como en la transmisión de los datos (se transmiten más rápidamente empleando el mismo ancho de banda). El precio que debe pagarse es cierto tiempo de cómputo para comprimir y descomprimir los datos [WMB99], tradicionalmente, ha existido un compromiso entre los beneficios de compresión y costo computacional [SH93] requerido.

La *compresión* de datos es la codificación de un cuerpo de datos D en un cuerpo de datos más pequeño D' [FY94]. Para comprimir los datos, los métodos de compresión examinan los datos, buscan redundancia en ellos, e intentan removerla. Una parte central en la compresión es la redundancia en los datos. Solo los datos con redundancia pueden comprimirse aplicando un método o algoritmo de compresión que elimine o remueva de alguna forma dicha redundancia. La redundancia depende del tipo de datos (texto, imágenes, sonido, etc), por tanto, no existe un método de compresión universal que pueda ser óptimo para todos los tipos de datos [S02].

El desempeño de los métodos de compresión se mide en base a dos criterios: la razón de compresión y el factor de compresión, siendo el segundo el inverso del primero. Las relaciones para determinar estas medidas están dadas por las igualdades de la figura 2.1. Entre mayor redundancia exista en los datos, mejor razón (factor) de compresión será obtenido.

$$\text{Razón de compresión} = \frac{\text{No Bytes archivo comprimido}}{\text{No Bytes archivo original}}$$

$$\text{Factor de compresión} = \frac{\text{No Bytes archivo original}}{\text{No Bytes archivo comprimido}}$$

Figura 2.1 Evaluación de los métodos de compresión

Métodos de compresión

La compresión de datos puede dividirse en dos tipos principales: compresión *con pérdida* y *compresión sin pérdida*. En compresión sin pérdida, es posible reconstruir exactamente los datos originales D dado D'. Al proceso de reconstrucción se le denomina *descompresión*. Por otra parte, en compresión con pérdida, la descompresión produce solamente una aproximación D* a los datos originales D.

La compresión de datos sin pérdida es comúnmente usada en aplicaciones como compresión de texto, donde la pérdida de un solo bit de información es inaceptable. La compresión con pérdida es usada a menudo para aplicaciones de compresión de imágenes y audio (video) destinadas al entretenimiento. No es el caso en aplicaciones de compresión de imágenes médicas (identificación de tumores o anomalías) [FY94]. Los métodos de compresión con pérdida logran mejores razones de compresión. El reto que siempre se persigue es

conseguir un método de compresión que emplee el tiempo más corto, que sea posible comprimir la mayor cantidad de información, que el espacio de memoria sea lo más mínimo posible y que no exista pérdida de datos.

Un método de compresión *no adaptable* es rígido y no modifica su operación o sus parámetros en respuesta de los datos particulares que se van a comprimir. Un método *adaptable* examina los datos y modifica su operación o parámetros de acuerdo a ellos. Un método *semi-adaptable* o de dos pasadas, lee los datos que van a comprimirse para determinar sus parámetros de operación internos (primera pasada) y a continuación realiza la compresión (segunda pasada) de acuerdo a los parámetros fijados en la primera pasada [S02].

Los métodos de compresión sin pérdida pueden clasificarse en dos tipos: substitucionales (o basados en diccionario) y estadísticos. La principal distinción entre ambos es que en el caso de los métodos estadísticos, la codificación de un símbolo se basa en el contexto en el que este ocurre, mientras que los métodos substitucionales agrupan símbolos creando un tipo de contexto implícito [WMC99]. En general, un compresor estadístico logra mejores razones de compresión que un compresor substitucional pero la complejidad computacional y requerimientos de memoria para los compresores estadísticos son mucho mayores que en los compresores substitucionales [JB95].

Métodos estadísticos.

Los métodos de compresión estadísticos usan las propiedades estadísticas de los datos que van a comprimirse para asignar códigos de longitud variable a los símbolos individuales en los datos. Existen varios métodos estadísticos propuestos, la principal diferencia entre ellos es la forma en la que cada uno obtiene las probabilidades de los símbolos [WMC99]. Los métodos estadísticos emplean un modelo para obtener las probabilidades de los símbolos, la calidad de la compresión que se logra depende de que tan bueno sea ese modelo. Para obtener buena compresión, la estimación de la probabilidad se basa en el contexto en el que ocurren los símbolos. Dadas las probabilidades de los símbolos, la codificación se encarga de convertir dichas probabilidades en una cadena de bits para obtener los datos en forma comprimida. Los métodos estadísticos más comúnmente usados son Codificación Huffman, Codificación Aritmética y PPM.

Codificación de Huffman

La codificación de Huffman [H52] es un método propuesto en 1952 por David Huffman. La idea detrás de la codificación de Huffman es asignar códigos binarios lo más cortos posibles a aquellos símbolos que ocurren con mayor frecuencia en los datos. Los símbolos con poca frecuencia tendrán asignado códigos binarios de longitud más grande. El óptimo desempeño del algoritmo se consigue cuando el número de bits asignado a cada carácter es proporcional al logaritmo de la probabilidad de mismo.

El algoritmo de Huffman construye un árbol binario mediante el cual, asigna los códigos a los símbolos de entrada. Se realiza una primera pasada sobre los datos a comprimirse para obtener las estadísticas de los símbolos. Los símbolos son ordenados en una lista de acuerdo a su probabilidad. Esta lista ordenada de símbolos serán los nodos iniciales para la construcción del árbol. La construcción del árbol se realiza de forma iterativa repitiendo los pasos 1-4 que se muestran en la figura 2.2

1) Si la lista contiene un solo nodo, terminar

2) Los dos nodos (L y R) de la lista con las probabilidades más pequeñas se seleccionan.
3) Se crea un nodo intermedio (F) y se construye un subárbol siendo F el padre, L y R los hijos izquierdo y derecho respectivamente. El arco entre los nodos F y L se etiquetan como 0 y el arco entre los nodos F y R se etiqueta como 1.
4) F es agregado a la lista ordenada con valor de probabilidad igual a la suma de las probabilidades de L y R.

Figura 2.2 Algoritmo para generar el árbol de Huffman

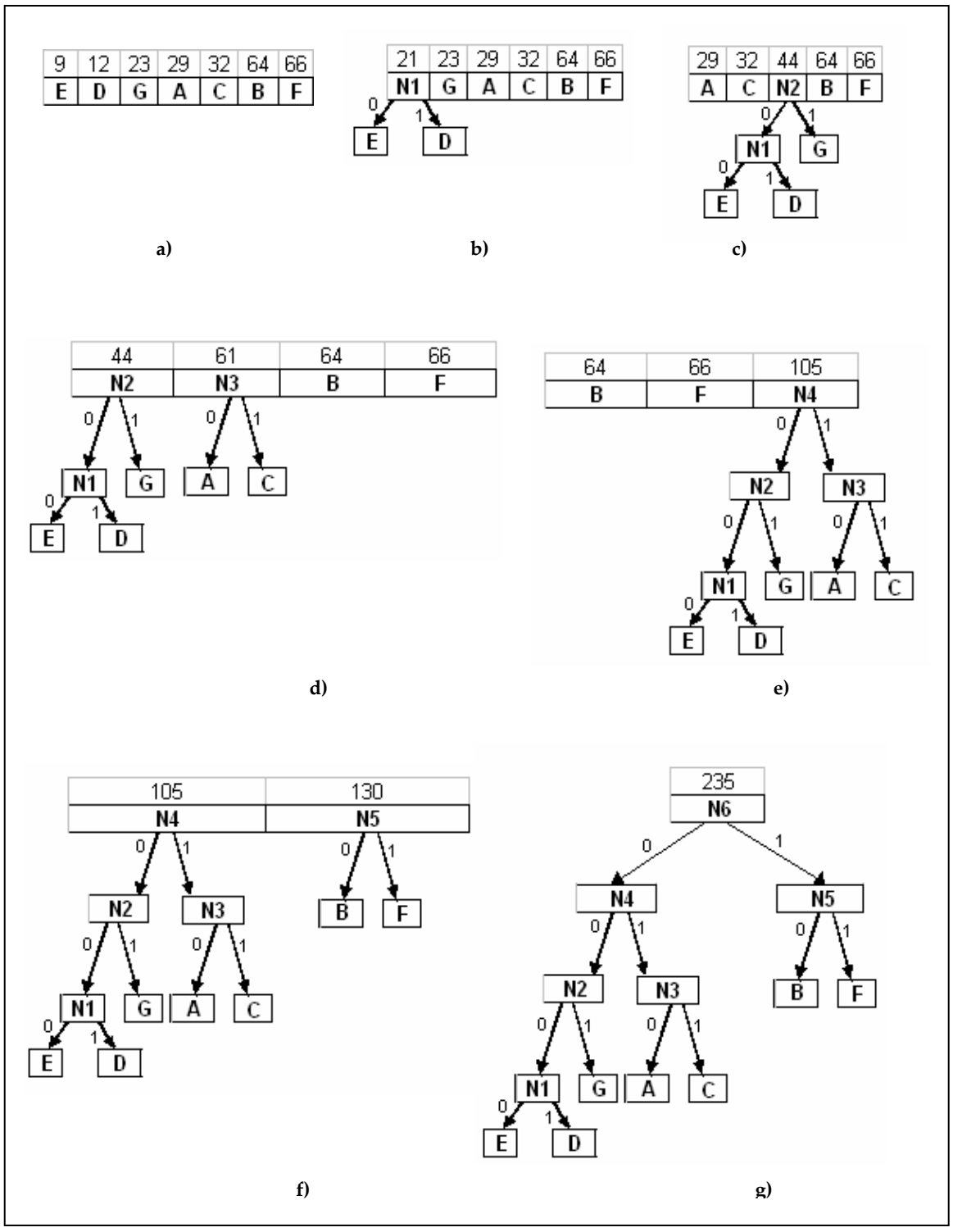

Figura Creación del árbol binario dadas las estadísticas de los símbolos en el mensaje a comprimir.

El árbol resultante contendrá *n* hojas para *n* símbolos en la entrada. Como ejemplo, supongamos que un mensaje va a ser codificado y que una vez obtenido las estadísticas de los símbolos de entrada se tienen las siguientes frecuencias: 66 Fs, 64 Bs, 32 Cs, 29 As, 23 Gs, 12 Ds y 9 Es (figura 2.3a). Los símbolos con mínima

frecuencia, D y E, se seleccionan, se crea un nuevo nodo N1 y se forma el subárbol según el paso 3), N1 es agregado a la lista con valor de frecuencia 21 (figura 2.3b). Ahora, G y N1 son seleccionados, se crea el nodo N2 y se forma el subárbol, N2 se agrega a la lista con valor de frecuencia 44 (figura 2.3c). Los nodos ahora seleccionados son C y N2, se crea N3, se forma el subárbol…, etc.

En la figura 2.3 se muestra el flujo de construcción del árbol de Huffman para el ejemplo anterior. Una vez que se tiene el árbol de Huffman (figura 2.3g), se realiza la asignación de códigos a los símbolos de entrada recorriendo el árbol desde la raíz hasta las hojas. En la tabla 2.1 se muestra los códigos correspondientes al ejemplo de la figura 2.3.

El árbol generado no es único, es decir, pueden existir diferentes árboles de Huffman para los mismos símbolos de entrada. El caso de tener más de un árbol ocurre cuando existen más de dos nodos con la misma mínima probabilidad, en este caso, la selección de tales nodos para la construcción del subárbol conllevara a árboles finales diferentes. Con diferentes árboles posibles, se tienen diferentes codificaciones para los símbolos de entrada pero el número promedio de bits por símbolo es siempre el mismo [S02]. En el ejemplo anterior, el número de bits promedio por símbolo es (2+2+3+3+3+4+4)/7 = 3.

Cuando más de una posible codificación es posible, se selecciona aquella representación donde la varianza sea mínima [S02]. La varianza de un código mide que tanto se desvía el tamaño de los códigos individuales de su tamaño promedio. Por otra parte, auque el tamaño del código de un símbolo depende de la probabilidad de dicho símbolo, el tamaño del alfabeto también influye en la longitud de los códigos. Para un alfabeto grande, las probabilidades de los símbolos se reducen y por tanto los códigos de Huffman crecen.

Tabla 2.1 Códigos de Huffman

Símbolo	Código
B	10
F	11
G	001
A	010
C	011
E	0000
D	0001

Proceso de descompresión

El compresor determina los códigos de los símbolos basándose en su probabilidad o frecuencia de ocurrencia en los datos. Estos códigos se almacenan junto con los datos comprimidos para que el descompresor pueda reconstruir los datos originales. El descompresor construye el árbol de Huffman y comienza la descompresión posicionándose en la raíz del árbol. El descompresor lee el primer bit de los datos comprimidos, si es cero, se mueve al hijo derecho, si es uno, se mueve hacia el hijo izquierdo. El proceso se repite, se lee el siguiente bit de los datos comprimidos y se avanza hacia el hijo derecho si el bit leído es un cero o al izquierdo si el bit leído es un uno. Cuando se llega a una hoja en el árbol, el descompresor encuentra el código original y regresa el valor del símbolo como dato descomprimido. El descompresor se coloca nuevamente en la raíz del árbol y se repite el mismo proceso.

Huffman Adaptable

El método original propuesto por Huffman, requiere dos pasadas sobre los datos para realizar la compresión, en la primera, se obtienen las estadísticas de los símbolos, en la segunda se realiza la compresión. Entre ambos pasos se crea el árbol binario. El método suele ser lento debido a estas dos pasadas y por tanto, poco atractivo en aplicaciones de compresión de datos en ¿tiempo real?. El método usado en la práctica es Codificación de Huffman Adaptable o Dinámica [CH83]. En este método, el compresor y el descompresor

inician con un árbol vacío y conforme se leen y procesan (comprimir/descomprimir) los símbolos, ambos modifica el árbol de la misma forma.

Inicialmente, el compresor lee cada símbolo de entrada y si es la primera vez que aparece, lo escribe tal cual al archivo de salida, lo agrega al árbol y le asigna un código de salida. Si el símbolo vuelve a ocurrir, el actual código asignado a él se escribe en el archivo de salida, su frecuencia se incrementa en uno y se actualiza el árbol para mantener la mejor codificación de acuerdo a las frecuencias de los símbolos. El compresor refleja los mismos pasos que el compresor, cuando lee un símbolo no-comprimido, lo agrega al árbol y le asigna un código. Cuando lee un código comprimido, utiliza el árbol que ha formado hasta el momento para obtener el símbolo asociado. El descompresor también reorganiza el árbol de la misma forma que el compresor. El descompresor necesita saber distinguir entre códigos comprimidos y códigos no-comprimidos. Para ello, utiliza un símbolo especial de escape que puede variar a lo largo del proceso de descompresión. Generalmente, el símbolo de escape corresponde al valor codificado de una hoja en el árbol con ocurrencia cero.

El árbol es verificado por cada símbolo de entrada. El símbolo con mayor frecuencia de ocurrencia debe aparecer en los primeros niveles del árbol mientras que los símbolos menos frecuentes se encuentran el los últimos niveles. Sea *X* el nodo del símbolo actual de entrada, *F* su frecuencia de ocurrencia. Las operaciones para modificar el árbol se realizan en un ciclo comenzando en el nodo correspondiente al símbolo de entrada actual. Cada iteración se compone de tres operaciones:

1) Comparar *X* con sus sucesores, de izquierda a derecha y de abajo hacia arriba. Si el sucesor inmediato tiene frecuencia *F+1* o mayor, los nodos están aun en orden y no se realiza ningún cambio. De lo contrario, algunos sucesores tienen frecuencia igual o más pequeña que *X*. En este caso, *X* debe intercambiarse con el último de sus sucesores que probabilidad igual o menor excepto con su padre.
2) La frecuencia de *X* se incrementa en 1y también la frecuencia de todos sus padres.
3) Si *X* es la raíz del árbol, el ciclo termina, de lo contrario, los pasos 1-3 se repiten ahora con el padre del nodo X.

Codificación Aritmética

Codificación aritmética [WNC87] es una técnica que obtiene excelente razón de compresión, aunque es un método lentota que requiere por lo menos de una multiplicación por cada símbolo de entrada. En teoría, la codificación aritmética asigna un único *codeword* a cada posible conjunto de datos. La compresión aritmética se basan también en las probabilidades de ocurrencia de los mensajes emitidos por la fuente de información. Se basa en la representación de un valor del intervalo [0,1] con más decimales (más precisión) cuanto más información contengan los datos a comprimir. La codificación aritmética se realiza siguiendo los pasos siguientes:

1. Inicialmente, el intervalo actual es [L, H) = [0, 1)
2. Para cada símbolo *s* en la entrada se realizan dos acciones:
    a. El intervalo actual se subdivide en subintervalos. El tamaño del subintervalo es proporcional a la probabilidad estimada de los símbolos.
    b. Se selecciona el subintervalo que corresponde a s, este intervalo es ahora el intervalo actual.
3. Cuando el archivo se ha procesado, la salida final como resultado de la compresión es un número dentro del intervalo actual, generalmente, aquel que ocupe menos bits.

Para cada símbolo procesado, el intervalo se hace más pequeño y requiere de más bits para representarlo. Supóngase que se desea codificar la secuencia de caracteres ABCBAA. Las frecuencias de cada uno de los símbolos son: 3 As, 2 Bs y 1 C. Las probabilidades y valores de las variables para cada uno de los símbolos se muestran en la tabla 2.2

Tabla 2.2 Valores de las variables en la codificación aritmética

Símbolo	Frecuencia	Probabilidad	Rango [L,H]	LR	HR
A	3	3/6 = (0.5)	[0.00, 0.50)	0.00	0.50
B	2	2/6 = 0.33	[0.50, 0.83)	0.50	0.83
C	1	1/6 = 0.16	[0.83, 1.00)	0.83	1.00

Los símbolos y frecuencias de la tabla 2.2 se escriben en el archivo de salida antes de que inicie la compresión del archivo. La codificación inicia definiendo dos variables L = 0 y H = 1 formando el intervalo inicial [0, 1). Conforme los caracteres a codificar incrementan, el cálculo se vuelve más complicado. La naturaleza de realizar cálculos sobre números flotantes implica mayor tiempo de procesamiento haciendo al método lento respecto a otros. Otra de las ventajas de la codificación aritmética frente a la codificación de Huffman es que la representación se calcula al vuelo, es decir, se requiere de menos memoria [WMB99].

Predicción mediante "Matching" Parcial (PPM)

En 1984, Cleary y Witten introdujeron el algoritmo PPM [CW84]. Este algoritmo calcula las estadísticas de los símbolos en contextos que han aparecido anteriormente. En este algoritmo se emplea un codificador aritmético para asignar códigos a los símbolos. La desventaja de PPM es que son lentos en la ejecución y requieren mayor memoria para almacenar las estadísticas de los símbolos. El algoritmo PPM obtiene las mejores razones de compresión dentro del grupo de algoritmos de compresión sin pérdida. Su baja velocidad de ejecución y los requerimientos de memoria limitan su uso en la práctica.

Métodos substitucionales o basados en diccionario.

Los métodos basados en diccionario usan el principio de remplazar cadenas de datos con *codewords* que identifican a esa cadena dentro de un diccionario [WMB99]. El diccionario contiene una lista de subcadenas y codewords asociados a cada una de ellas. El diccionario que contiene las cadenas de símbolos puede ser estático o adaptable (dinámico). Los métodos basados en diccionario, a diferencia de los métodos estadísticos, usan codewords de longitud fija y no requieren de las estadísticas de los símbolos para realizar la compresión. Prácticamente, todos los métodos de compresión sustitucionales están basados en los métodos de compresión desarrollado por Jacob Ziv y Abraham Lempel en los 70s, los métodos LZ77 y LZ78. En ambos métodos una subcadena es remplazada por un apuntador a donde dicha cadena haya ocurrido previamente en el diccionario, el cual, se crea dinámicamente. Las principales diferencias entre los métodos sustitucionales son en como los apuntadores son representados y en las limitaciones que imponen sobre lo que los apuntadores pueden representar.

LZ77

El método LZ77 [ZL77] usa como diccionario parte de la información previamente leída. El codificador mantiene una ventana al archivo de entrada y recorre la entrada en esa ventana de derecha a izquierda tantas veces como cadenas de símbolos son codificadas. La ventana se divide en dos partes, la parte de la izquierda se llama buffer de búsqueda y es el diccionario actual; la parte de la derecha se llama buffer hacia el frente y contiene símbolos aun no codificados (figura 2.4). El algoritmo para la codificación se muestra en la figura 2.5. Inicialmente, el buffer de búsqueda puede rellenarse con algún símbolo inicial mientras que el buffer hacia el frente se rellena con los primeros símbolos del archivo de entrada. En la práctica, el buffer de búsqueda es de algunos cientos de bytes mientras que el buffer hacia el frente es de decenas de bytes.

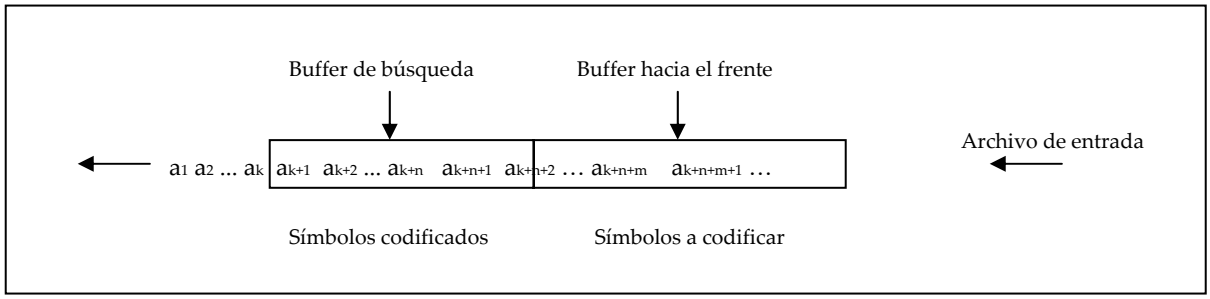

Figura 2.4 Diccionario en el método LZ77

El decodificador es más simple que el codificador. El decodificador debe mantener una ventana de igual tamaño que la ventana de búsqueda empleada en el codifcador. Inicialmente, este buffer es rellenado con el mismo símbolo empleado para rellenar el buffer inicialmente en el codificador. Los pasos para realizar la descompresión se listan en la figura 2.6.

---

1. El codificador busca la subcadena más grande en el buffer de búsqueda (de derecha a izquierda) que sea el prefijo del buffer hacia el frente.
    a. Si tal subcadena existe, la codificación consiste de una tripleta $T = (O, L, C)$, donde O es la distancia en símbolos desde inicio de la subcadena encontrada hasta el final del buffer de búsqueda, L es la longitud de la subcadena encontrada y C es el símbolo siguiente a la subcadena encontrada en el buffer hacia el frente.
    b. Si la cadena no existe, O y L son puestos a cero y C es el primer símbolo del buffer hacia el frente (en la figura 2. 4, el símbolo a_{k+n+1}).
2. Ambos buffers se recorren hacia la izquierda $L + 1$ posiciones.

Figura 2.5 Algoritmo de codificación en LZ77

El método LZ77 no requiere de las frecuencias de ocurrencia de los símbolos presentes en los datos que se van a comprimir. El método solo busca patrones de subcadena en el buffer de búsqueda que ocurran en el buffer hacia el frente, suponiendo que dichos patrones ocurren continuamente, al menos dentro de la subcadena que engloban ambos buffers. Si esta suposición se cumple en el conjunto de datos que se va a comprimir, el método logra buenas razones de compresión. El método LZ77 ha sido mejorado desde su invención, entre las mejoras destacan la utilización de campos O y L de longitud variable y el uso de diferentes tamaños de diccionario (longitud de la ventana). La utilización de una ventana más grande implica mayor tiempo de búsqueda de los patrones y obliga a la utilización de mejores estructuras de datos para realizar la implementación. Por otra parte, el utilizar una ventana más grande implica el uso de campos O y L más grandes con lo que la razón de compresión se ve afectada. Otro factor a considerarse es la posible expansión del archivo en vez de la reducción deseada. Debido a que se requiere un codeword de tres campos, esto puede producir la expansión del archivo de salida para los casos donde la subcadena encontrada sea de longitud 1 o 2. Una posible solución es escribir al archivo de salida codewords cuando la cadena encontrada sea de longitud mayor a 3 y escribir los símbolos sin comprimir cuando la cadena encontrada sea de longitud menor o igual a 3. Esto implica el uso de un símbolo especial que permita diferenciar al decodificador entre codewords y datos no comprimidos.

1. El decodificador lee cada tripleta T = (O, L, C). Se dan dos casos para L:
    a. L es igual a cero:
        i. C se escribe al archivo de salida
        ii. El buffer se recorre a la izquierda una posición y se rellena con C
    b. L es diferente de cero:
        i. La subcadena S en el buffer iniciando en la posición O (contando de derecha a izquierda) y de longitud L se copia al archivo de salida.
        ii. C se escribe al archivo de salida
        iii. El buffer se recorre L posiciones a la izquierda, se rellena con la subcadena L.
        iv. El buffer se recorre 1 posición a la izquierda y se rellena con el símbolo C.

Figura 2.6 Algoritmo de decodificación en LZ77

LZ78

El método LZ78 [ZL78] no utiliza una ventana sobre los datos a comprimirse como en el caso del LZ77. Se tiene un diccionario que contiene las cadenas que han ocurrido previamente. El diccionario esta vacío inicialmente y su tamaño esta limitado por la memoria disponible. Para ilustrar la forma en la que el método funciona, considérese un diccionario (arreglo lineal) de N localidades con la capacidad de almacenar una cadena se símbolos en cada una de ellas. . El diccionario se inicializa guardando en la posición cero del diccionario la cadena vacía. El algoritmo de codificación se muestra en la figura 2.7.

1. Mientras existan símbolos a la entrada
    a. $S = Null$, $Pos = 0$
    b. X = siguiente símbolo de entrada, $S = S \cdot X$
    c. Mientras S exista en el diccionario
        i. $Pos = Pos(S)$
        ii. X = siguiente símbolo de entrada, $S = S \cdot X$
    d. Salida: (Pos, X)
    e. Guardar S en el diccionario

Figura 2.6 Algoritmo de codificación en LZ78

El proceso es iterativo y termina cuando ya no existen más símbolos a la entrada para codificar. En cada iteración S se inicializa a $Null$ ($S = Null$ indica una cadena vacía que siempre se encuentra en la posición cero del diccionario). El símbolo X del archivo de entrada se lee y se busca la cadena $S \cdot X$ (concatenación de S y X) en el diccionario, si la cadena $S \cdot X$ se encuentra en el diccionario, S es ahora $S \cdot X$ y se lee un nuevo símbolo X. Nuevamente, se busca $S \cdot X$ en el diccionario y si la cadena se encuentra, se vuelve a leer otro símbolo de entrada y el proceso se repite buscando nuevamente $S \cdot X$ en el diccionario. Si la cadena $S \cdot X$ no se encuentra en el diccionario, se guarda la cadena $S \cdot X$ en una posición disponible en el diccionario y se escribe al archivo de salida la posición de S dentro del diccionario y el símbolo X.

A diferencia del método LZ77, ahora los codewords se componen solo de dos campos, un campo apuntador o índice que identifica la posición de una cadena dentro del diccionario y un campo que contiene el último símbolo que se ha leído de la entrada. Entre más grande sea el diccionario, mas cadenas son almacenadas y se tienen concordancias de cadenas más grandes que mejoran la razón de compresión, pero se requieren apuntadores más grandes y el tiempo de búsqueda se incrementa. La representación del diccionario puede ser una estructura de árbol llamada *trie*. En este caso, inicialmente, el árbol se compone únicamente de una nodo raíz que representa la cadena vacía. Todos las cadenas que comienzan con la cadena vacía se agregan al árbol como hijos de la raíz. Cada uno de los símbolos que son hijos de la raíz se convierten ahora en la raíz de un subárbol para todos las cadenas que comienzan con ese símbolo. En la tabla 2.3 y figura 2.7 se

muestran las representaciones del diccionario en forma lineal y de árbol respectivamente de la codificación de la cadena *abacabacaba* mediante el método LZ78.

Tabla 2.3 Diccionario lineal en LZ78

Diccionario		Codeword
0	Null	
1	a	(0, a)
2	b	(0, b)
3	ac	(1, c)
4	ab	(1, b)
5	aca	(3, a)
6	ba	(2, a)

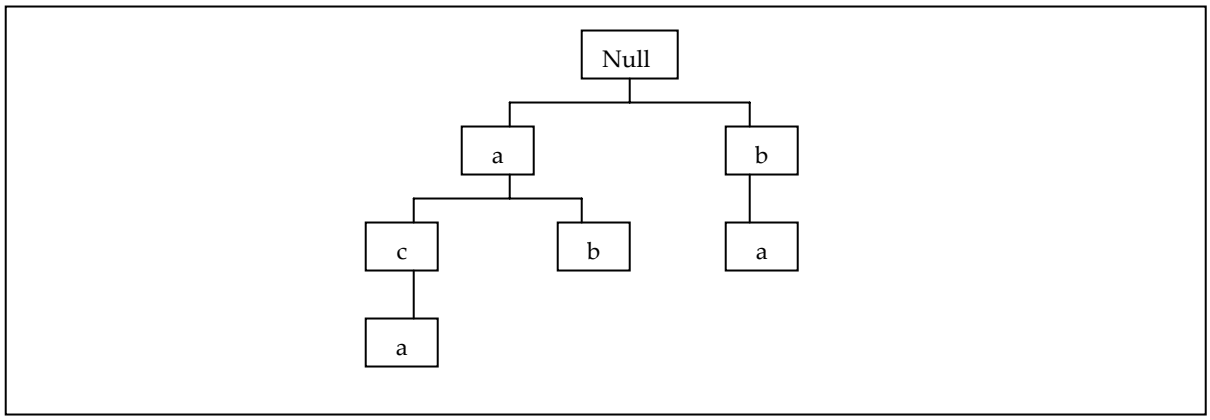

Figura 2.7 Diccionario LZ78 como un árbol

La representación de árbol tiene dos ventajas principales: la asignación de memoria se realiza conforme nuevos símbolos se agreguen al árbol y la búsqueda de cadenas se realiza más rápido. Cuando el espacio de memoria asignado se agota, las nuevas cadenas ya no se agregan pero el árbol aún sirve para continuar con la codificación. El árbol puede vaciarse por completo y reiniciar ella construcción con los nuevos símbolos de entrada o bien, se pueden eliminar del árbol solo aquellas cadenas que sean menos utilizadas. En este último caso, no existe un buen algoritmo que decida que nodos deben eliminarse y de que forma debe realizarse tal eliminación [S02]. La codificación en LZ78 es más rápida que la realizada en LZ77 pero el decodificador en LZ78 es más complejo que en LZ77. El decodificador debe construir y mantener el diccionario de la misma forma que lo realizó el codificador.

LZW

El método LZW [W84] es una variante del método LZ78. El codeword consiste ahora de únicamente una referencia a la posición en el diccionario de la cadena encontrada. En el caso de codificar símbolos de 8 bits, las primeras 256 posiciones del diccionario se inicializan con los 256 símbolos antes de iniciar la codificación. Este paso inicial asegura que cualquier símbolo leído de la entrada siempre se localizará en el diccionario con lo que es posible manejar únicamente un apuntador a la posición del diccionario como código de salida. El codificador lee cada símbolo de la entrada y los acumula en una cadena S. Cada vez que un símbolo X es leído de la entrada, el símbolo se concatena con S y la cadena resultante se busca en el diccionario. Siempre que la concatenación es encontrada en el diccionario, un nuevo símbolo X es leído de la entrada, se concatena a S y se realiza la búsqueda de la nueva cadena $S{\cdot}X$. Cuando la concatenación $S{\cdot}X$ no se encuentra en el diccionario (S se encuentra en el diccionario pero $S{\cdot}X$ no se encuentra), el codificador escribe a la salida la posición en el diccionario de la cadena S, guarda $S{\cdot}X$ en una localidad disponible en el diccionario e inicializa S con el símbolo X. El algoritmo del codificador *LZW* se muestra en la figura 2.8.

1. Para $i = 0$ hasta 255
   a. $Pos(i) \leftarrow S_i$
2. S = Cadena vacía
3. Mientras existan símbolos en la entrada
   a. X = siguiente símbolo de entrada
   b. Mientras $S \cdot X$ se encuentre en el diccionario
      i. $S \leftarrow S \cdot X$
      ii. X = siguiente símbolo de entrada
   c. Escribir a la salida Posición de S en el diccionario
   d. Guardar $S \cdot X$ en el diccionario
   e. $S \leftarrow X$

Figura 2.8 Algoritmo de codificación en LZW

El tamaño del diccionario en LZW debe ser mayor de 256 localidades. Las implementaciones de LZW a menudo usan apuntadores de 9 a 16 bits (diccionario de 512 a 64Kb). Cuando el diccionario se llena, se deben tomar en cuenta las medidas necesarias, de la misma forma que en LZ78. Al igual que e LZ78, un diccionario grande implica mejor razón de compresión a costa de mayor tiempo para realizar la compresión. LZW se adapta lentamente a la entrada de los datos ya que la construcción de las cadenas o frases se realiza de carácter en carácter.

La decodificación en LZW se realiza de la siguiente forma: El diccionario se construye con las 256 primeras posiciones con los 256 símbolos. El decodificador lee el archivo comprimido, el cual consiste de apuntadores al diccionario. Cada apuntador indica la posición en el diccionario de donde se recuperaran el o los símbolos no comprimidos para escribirlos al archivo de salida. El decodificador también construye el diccionario de la misma forma que el codificador conforme nuevos símbolos son recuperados del diccionario. Cuando se lee el primer apuntador del archivo comprimido, se recupera el símbolo X almacenado en esa posición en el diccionario y se asigna a S ($S \leftarrow X$). S se escribe al archivo de salida y se lee el siguiente apuntador. El contenido del apuntador se almacena en una variable T y también se escribe en el archivo de salida, el primer símbolo S1 de T se concatena con S para formar la cadena SS1, si SS1 no existe en el diccionario entonces se agrega. S es ahora T, se lee el siguiente apuntador y se repiten los mismos pasos. El algoritmo de decodificación se muestra en la figura 2.9. En la tabla 2.4 se muestra el diccionario después de aplicar el algoritmo de codificación LZW a la cadena de entrada abacabaca. En la figura 2.10 se muestra los pasos de decodificación según el algoritmo de la figura 2.8 para el ejemplo de la tabla 2.4.

1. P ← Primer apuntador en archivo comprimido
2. S ← Pos(P). Escribir al archivo de salida S.
3. P ← Siguiente apuntador en el archivo comprimido
4. T ← Pos(P). Escribir al archivo de salida T.
5. S_1 ← primer símbolo de T
6. Agregar SS_1 en el diccionario, si dicha cadena no se encuentra.
7. S ← T. Regresar al paso 3.

Figura 2.9 Algoritmo de decodificación en LZW

Tabla 2.4 Diccionario después de aplicar el algoritmo de codificación LZW

Pos	Frase	Código de Salida
0	a	
1	b	
2	c	
3	ab	0
4	ba	1
5	ac	0
6	ca	2
7	aba	3
8	aca	5
9	a(eof)	0

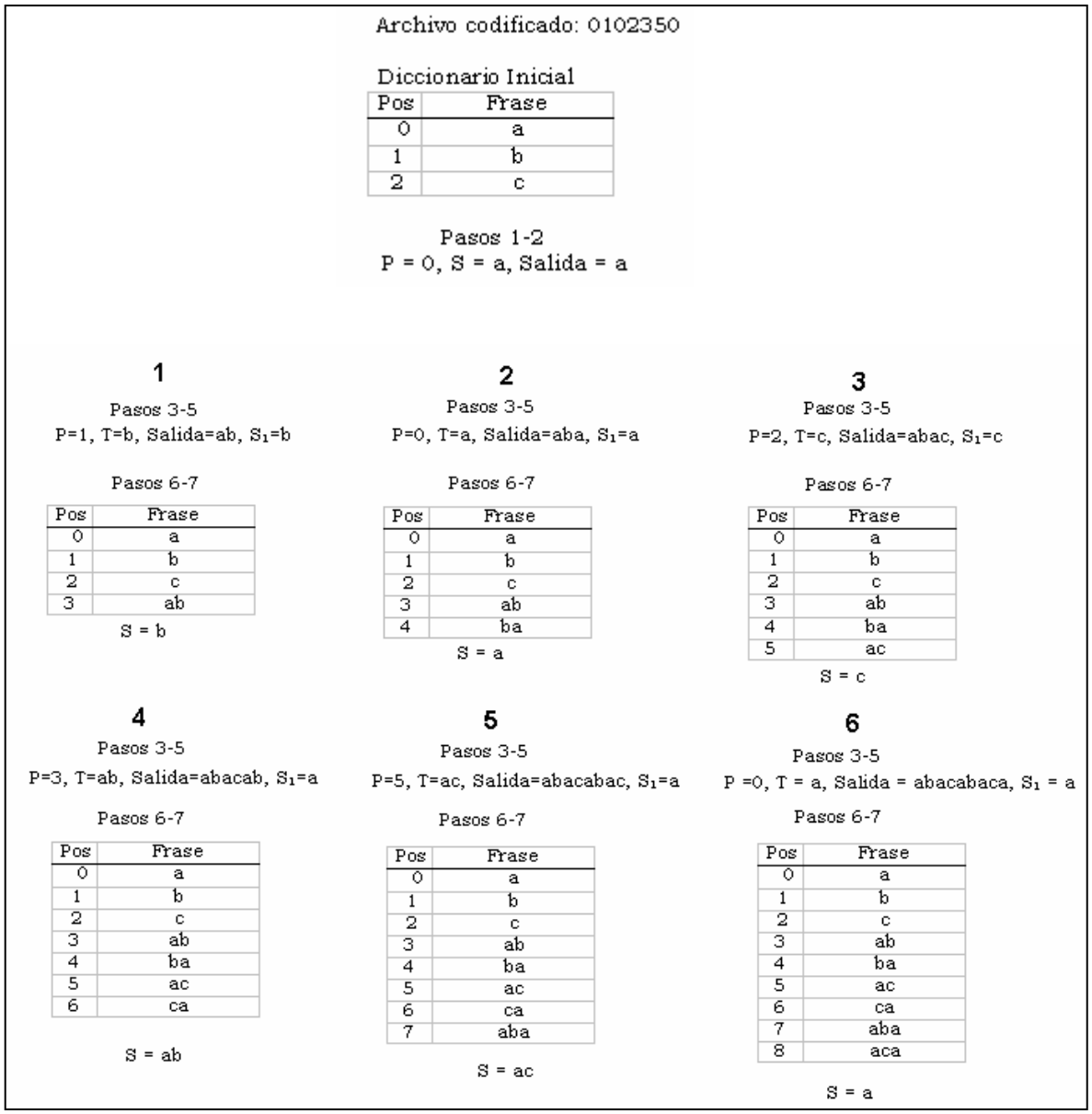

Figura 2.10 Decodificación en LZW para el ejemplo de la tabla 2.4

El diccionario en LZW puede implementarse, al igual que en LZ78, por medio de un *trie*. Cada nueva cadena agregada al diccionario es diferente únicamente en un símbolo respecto a otra cadena ya existente (lo que en el *trie* sería una cadena padre). El problema con esta representación son las múltiples ramas que un nodo puede contener.
Otros métodos de compresión sin pérdida basados en diccionarios son LZS, LZSS, Run Length y X-Match.

Comparación entre los métodos de compresión

La selección de un método de compresión dentro de los que existen no es sencilla. Un método puede ser más rápido que otro, otros pueden lograr mejores razones de compresión, otros pueden requerir mayor memoria de trabajo. Las implementaciones de los métodos de compresión están siendo mejoradas continuamente por lo que resulta difícil dar resultados definitivos. El desempeño de los métodos de compresión depende de tipo de dato que se va a comprimir. Para medir el desempeño de los programas, en este trabajo, se utiliza el corpus de Canterbury [CB] como benchmark. El corpus se compone de 11 archivos que fueron seleccionados con el propósito de probar sistemas de compresión. Las tres métricas a considerarse son: razón de compresión, tiempo de ejecución y recursos de memoria. La velocidad de ejecución es difícil de establecer ya que depende de varios factores como que tan bien fue implementado el método, la arquitectura de la máquina sobre la que se ejecutó el programa y que tan bueno fue el compilador empleado.

Comentarios finales

En la práctica, en cuanto a compresión estadística se refiere, es preferible codificación de Huffman ya que obtiene razón de compresión cercana a la óptima (entropía de la fuente) y es más rápido que codificación aritmética y requiere menos recursos que PPM. La codificación en LZ78 es más rápida que la realizada en LZ77, ya que no se realizan las comparaciones buscando match de subcadenas pero el decodificador en LZ78 es más complejo que en LZ77 ya que el diccionario debe construirse de la misma forma que en el codificador. LZW, que es una mejora al método LZ78, se adapta lentamente a la entrada de los datos ya que la construcción de las cadenas o frases se realiza de carácter en carácter (es ineficiente para casos donde existe un mismo símbolo repetido varias veces). Los compresores estadísticos logran mejor razón de compresión que los compresores sustitucionales pero la complejidad computacional y requerimientos de memoria para los compresores estadísticos son mucho mayores que en los compresores sustitucionales (por ejemplo, LZ requiere decenas de kilobytes mientras que PPM por ejemplo requiere de cientos o miles de kilobytes). Además, la decodificación es más simple y más rápida en los métodos sustitucionales que en los métodos estadísticos. Todos los métodos de compresión adaptables, que son los métodos que ofrecen mejores razones de compresión, requieren una cantidad significativa de memoria durante la compresión y descompresión.

Referencias

[S02] Salomon, D, *A Gude to Data Compresión Methods*, New York, Springer-Verlag, 2002.

[WMB99] Witten. I.H., Moffat, A. and Bell, T.C., *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second Edition, San Francisco, CA, Academic Press, 1999.

[WS93] Weiss, J. and Schremp, D. "*Putting data on diet"*. IEEE Spectrum, August 1993.

[SH93] Stauffer, L.M., and Hirschberg, D.S. *"Parallel Text Compression"*. Technical Report 91-44, Unuversity of California, Irvine. January 1993.

[CW84] Cleary, J.G. and Witten, I.H. *"Data Compression using adaptive coding and partial sting matching*", IEEE Transactions on Communications, 32(4), 396-402. 1984.

[JB95] Jung, B. and Burleson. W.P. *"Real-Time VLSI compression for high-speed wireless local area networks"*, in Data Compression Conference, March 1995.

[FY94] Fowler, J. and Yagel, R. (1994) *"Lossless Compression of Volume Data"*, Symposium on Volume Visualization, pp. 43--50, Oct. 1994.

[H52] Huffman, D., *"A Method for the Construction of Minimum Redundancy Codes"*, Proceedings of de IRE 40(9): 1098-1101.

[WNC87] Witten, I.H., Neal, R.M., Cleary, J.G., *"Arithmetic Coding for Data Compression"*, Communication of the ACM, Vol 30, No. 6, pp. 520-540, June 1987

[CH83] Cormack, G.V., and Horspool, R.N. "Algorithms for Adaptive Huffman Codes." Inf. Processing Letters 18, 3 (March 1983), pp. 159-166.

[Sha48] Shannon, C.E., "A mathematical theory of communication", in Key Papers in the Development of Information Theory (D. Slepian, ed.), pp. 5-18, New York: IEEE Press, 1948.

[ZL77] Ziv, J. and Lempel, A., "A Universal Algorithm for Secuential Data Compression", IEEE Trans. Information Theory, vol. 23, pp. 337-343, May 1977.

[ZL78] Ziv, J. and Lempel, A., "Compression of Individual Secuences via Variable-Rate Coding", IEEE Transactions on Information Theory, vol. 24, pp. 530-536, 1978.

[W84] Welch, T., "A Technique for High-Performance Data Compression", IEEE Computer, vol. 17, pp. 8-19, June 1984.

[CB] Canterbury Corpus. Disponible en http://corpus.canterbury.ac.nz