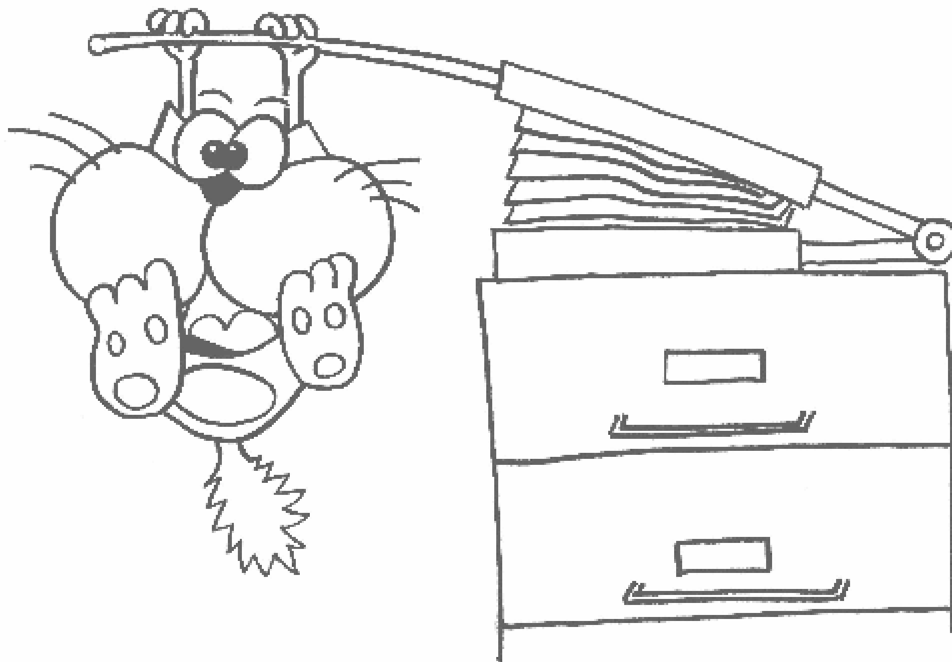
Compresión de datos



INDICE:

INTRODUCCIÓN 3

DATOS E INFORMACIÓN 3
CÓDIGOS PREFIJOS 4
ENTROPÍA 4

COMPRESORES ESTADÍSTICOS 7

HUFFMAN 7
- Representación de bits en bytes 9
- Huffman dinámico 10
- Códigos de Shannon Fano 11
- Manejo eficiente del árbol 12
- Half coding (1/2 coding) 15

COMPRESIÓN ARITMÉTICA 17
- Aritmética de Enteros: 20
- Descompresión en Aritmético: 23
- Implementación con números binarios: 23

UTILIZACIÓN DE CONTEXTOS 23
PPMC 24
- Inicio de la compresión: 25
- Descompresión: 30

COMPRESIÓN NO ESTADÍSTICA 32

LZ77 32
LZ78 - LZW 36
- Caso particular 39
- Clearing 40
- Implementación eficiente de la tabla 40

LZHUFF 42
LZP 43

LOCALIDAD EN ARCHIVOS 47

LOCALIDAD 47
MOVE TO FRONT 47
BLOCK SORTING 49
- Descompresión 50
- Implementación 52

MODELOS QUE APROVECHAN LA TRANSFORMACIÓN BS + MTF 53
- Modelo de Shannon 53
- Modelo Aritmético 54
- Half Coding 55

ANEXO 56

DESIGUALDAD DE KRAFT 56
ESTADO DEL ARTE 57

Introducción

Este apunte todavia se encuentra en pleno proceso de creación. Es por eso que, pese a haber sido revisado varias veces, puede contener errores. Cualquier problema que encuentren les agraderecemos que nos lo hagan saber para que al salir la version definitiva pueda cumplir con el objetivo de forma completa

Datos e Información

Antes de ver los distintos métodos de compresión, es necesario discutir acerca de qué es lo que logran hacer los compresores. Las computadoras hoy por hoy permiten almacenar datos en ella gracias a distintos medios, ya sean magnéticos, eléctricos, etc, brindando una cantidad dada de bits, que pueden tomar 2 valores distintos, denominados "1" y "0" por convención. Los usuarios utilizan un grupo de dichos bits para guardar algunos datos que consideren importantes conservar, ya sea un documento con un trabajo práctico, una imagen de un paisaje o una canción

Todo lo anterior es sabido por cualquiera que se encuentre leyendo este apunte, o de lo contrario tendrá algún inconveniente para comprender el resto del mismo. Sin embargo, hay que notar algo. En el párrafo anterior nunca se habló de información. Y es que no es lo mismo guardar datos y guardar información, cosa importante para comprender la compresión de archivos.

Una primer diferencia entre los datos y la información la vemos en el pasaje del mundo real al mundo de las computadoras. En este pasaje, se debe sacrificar un poco de información para poder representar la realidad con un dato. Suponiendo que queremos almacenar un color en la computadora para después poder recordarlo, nos encontraremos frente al problema de cuantos bits utilizar para que éste color después no sea confundido con otro. Utilizando por ejemplo 3 bytes (uno para el rojo, otro para el verde y otro para el azul) llegamos a algo aceptable, sin embargo no hay un solo color en el mundo que se pueda representar con el mismo valor de rojo, verde y azul, sino que existen infinitos colores con una tonalidad un poco distinta entre sí. Por mayor cantidad de bits que utilicemos en la representación, no llegaremos a poder definir exactamente el color, sino una gama de colores similares entre sí

Otra diferencia entre dato e información la vemos cuando una palabra, por ejemplo "Maradona", es relacionada inmediatamente por nosotros con un jugador de fútbol. Sin embargo para un afroamericano obeso que viva en Chicago y cuya vida consista en ver Football Americano y comer nachos, la cadena de caracteres "Maradona" no significa nada, de hecho quizás no pueda decir si se está hablando de una persona, de una ciudad o de un tipo de comida. En base a quien lea la cadena de caracteres puede obtener mucho de ella o directamente nada.

Ahora bien, aun cuando no sean lo mismo, uno intenta que haya una relación entre los datos y la información. Por más que no sea el color exacto, uno pretende que con leer su dato llegue a un color tal que visualmente no sea distinto. También uno no puede pretender que todo el mundo comprenda su dato, pero sí que al menos lo comprenda aquél que lo deba leer. Así es como se relaciona una información con un dato, utilizando un método de codificación, como ser que el código es de 3 bytes, el primer byte es para rojo, de un valor en que 0 significa ausencia de rojo, etc etc

Quitémonos el primer problema de encima, asumiendo que no podremos representar en un 100% la información, ya que esta es de caracter infinito y los datos son de caracter finito. Igualmente, el valor al que llegamos (o sea, el dato en si), posee una información, no tan completa como la inicial, pero que igual nos servirá. Ahora lo que nos interesa es cuanto ocupa la información. Si todas las palabras tuvieran unos 20 caracteres, el gasto de hojas sería muy grande. Lo mismo pasa con los datos, ya que estos ocupan un espacio en un medio que nos cuesta dinero, o cuando los transmitimos tardamos un tiempo proporcional a la longitud de los mismos, con lo que

queremos que la codificación que elijamos ocupe lo menos posible. Dado a que ya sabemos cómo se almacenan los datos, podemos decir que lo que queremos es almacenar los datos utilizando la menor cantidad de bits posible.

La compresión de archivos es un proceso que tiene como entrada un archivo con una determinada cantidad de bits y como salida otro archivo con otra cantidad de bits, que puede ser igual, mayor o menor a la anterior. Un buen compresor obviamente será aquel que para la gran mayoría de los casos el tamaño sea menor. El proceso inverso es la descompresión y es necesario que el archivo original pueda recuperarse en su totalidad.

Existen otras técnicas conocidas como compactación que no cumplen con lo anterior, es decir, en base a un archivo original generan otro con distinta longitud pero de él no puede volverse al original con ningún proceso. Estos métodos, también conocidos como métodos con perdida, no serán vistos en este apunte ya que tienen perdida de información; pero no por ello debe pensarse que no sirve ya que combinados con mecanismos de compresión sin perdida generan archivos mucho mas pequeños y en general se basan en que la perdida que efectúan no se note en el archivo final.

En este apunte estudiaremos técnicas de compresión sin pérdida de datos. Pese a que siempre hablaremos de archivos, no quiere decir que la compresión necesariamente deba ser hecha sobre ellos, tranquilamente se puede hacer por ejemplo previa al envío de datos, para enviar menos bits, sin que en ningún momento se trabaje con un medio físico de almacenamiento

Códigos prefijos

Lo que nos interesa en un archivo no son los distintos 1 y 0 que lo componen sino la información que puedo obtener de ellos. Es necesario que al comprimir un archivo no pierda la información o de lo contrario se habrá pagado un precio caro por ocupar menos espacio. Si queremos almacenar algo que puede tomar 500 valores distintos, es evidente que no podremos utilizar solo 8 bits. Sin embargo podemos aprovecharnos mucho de otra información que tenemos en base a la totalidad del archivo y que hasta ahora no estamos utilizando. Por ejemplo, si almacenamos algo que puede tomar solo 4 valores y tuviéramos el archivo

1 1 1 1 2 1 1 1 1 3 1 1 1 1 4 1 1 1 1 1

Utilizando 2 bits para almacenar cada valor tendríamos un total de 2*20 = 40 bits. Pero a simple vista se ve que el valor 1 es muy frecuente. Sería bueno contar con una forma alternativa de almacenar el archivo en el cual se utilizaran menos bits para el 1. Sin embargo necesitamos que el archivo pueda ser correctamente interpretado, ya que si elegimos una representación del estilo :

1= “0” 2 = “01” 3 = “10” 4 = “11”

el archivo “010110”, se podría interpretar tanto como los valores 1, 3, 4, 1 o como los valores 2, 2, 3.

Para que esto no se de debemos trabajar con códigos prefijos, en los cuales la codificación de un valor posible no debe nunca comenzar con la codificación de otro valor posible. El código utilizado antes no es prefijo porque la codificación del valor 2 comienza con un 0, que es la codificación del valor 1. Si utilizamos códigos prefijos, estaremos seguros que se podrán decodificar siempre. Por ejemplo, la codificación

1 = “0” 2 = “10” 3 = “110” 4 = “111”

es prefija. Si codificamos el archivo anterior tendremos lo siguiente:

0 0 0 0 10 0 0 0 0 110 0 0 0 0 111 0 0 0 0 0

Que son 25 bits, mucho menor que los 40 del archivo original

Entropía

Al formular su teoría de la información a mediados de 1948, Claude Shannon (foto) estudió como lograr enviar la menor cantidad de bits posibles logrando igual que

el mensaje original pudiera ser reconstruido. En base a esto buscó dimensionar la información que nos da un mensaje.

Suponiendo que se hubiera recibido el texto "Diego Armando Marad", el hecho de que a continuación recibamos los últimos 3 caracteres "ona" no nos da mucha mas información ya que era exactamente lo que esperábamos. En cambio, de recibir otra tira de caracteres distinta, tendremos una cantidad de información mayor, ya que ahora veremos que estábamos recibiendo el nombre de otra persona distinta del que pensábamos. Shannon justamente determino que un símbolo contiene más información mientras menos probable sea, y planteó la siguiente fórmula:

$I(s) = Log_2 (1 / P(s)) = - Log_2 (P(s))$

O sea, la información de un símbolo (I(s)) es igual al logaritmo en base dos de 1 sobre la probabilidad de ocurrencia de dicho símbolo (P(s)). La unidad de medida de la información que utiliza esta fórmula es los bits. En el ejemplo anterior, el símbolo 1 ocurría 17 veces en un mensaje de 20 símbolos. La información de dicho símbolo entonces es de $-\log_2(17/20) = 0.2345$. En cambio, la de los otros símbolos es de $-\log_2(1/20) = 4.32$. Dado que utilizaremos muchas veces el logaritmo en base 2, de ahora en mas log(x) significará logaritmo en base 2 de x y se especificará cuando se esta hablando de logaritmo en base 10

Shannon fue más allá y analizó el problema de reducir el tamaño total de los datos a utilizar en un archivo. Independizándose de la cantidad, calculó el tamaño promedio de un símbolo cualquiera del archivo f como:

$$H(f) = \sum_{\forall s} P(s) * L(s)$$

Donde P(s) es la probabilidad del símbolo, L(s) es la longitud en bits del símbolo y H(f) el tamaño promedio de un símbolo. Siguiendo con el ejemplo, la primer representación (2 bits para cada símbolo) tiene un valor H(s) = 17/20 * 2 + 1/20 * 2 + 1/20 * 2 + 1/20 * 2 = 2 bits. En el segundo ejemplo (1 bit para 1, 2 para el 2 y 3 bits para el 3 y el 4) H(s) = 17/20 * 1 + 1/20 * 2 + 1/20 * 3 + 1/20 * 3 = 1.25 bits por símbolo.

Para lograr reducir el tamaño necesito una codificación que me de el menor valor posible de H(f). Dado que no se puede cambiar las probabilidades, ya que se estaría cambiando el mensaje, se buscará una función L(s) tal que minimice el H(f) asociado. Matemáticamente (no es intención de este apunte demostrarlo por lo complejo del razonamiento), se llega a que H(f) es mínimo cuando L(s) = I(s), es decir cuando $H(f) = \sum_{\forall s} - P(s) * Log[P(s)]$.

Este valor de H(f) es conocido como entropía del archivo f y es muy importante ya que fija un límite al tamaño del archivo: sea cual sea la codificación que se utilice, no se podrá llegar a un archivo que ocupe menos que su entropía multiplicada por su longitud. Es importante hacer notar que la entropía es calculada para una función L(s) que permita códigos prefijos, con lo que no es posible decir que con L(s) = 0 el valor de H(f) es menor a la entropía, ya que de utilizar 0 bits para cada código no se podría recuperar el archivo inicial. Esto se asegura viéndose que se cumpla la desigualdad de Kraft, que se estudia en el anexo del apunte

Este valor de la entropía es muy importante ya que la gran mayoría de métodos están orientados a buscar una codificación que use longitudes de símbolos lo mas parecido posible al tamaño de su información, para lograr aproximarse al limite de tamaño mínimo del archivo. Sin embargo es importante hacer notar que hablar de símbolo no es lo mismo que hablar de byte, sino que un símbolo es cualquier entrada que pueda ser comprimirá, y algunos compresores que veremos no utilizaran únicamente como símbolo al caracter a comprimir sino también otros aspectos como por ejemplo, los caracteres que preceden a dicho caracter a comprimir.

Analizando la fórmula de la entropía, vemos que el archivo ocupará lo mínimo cuando para cada símbolo se guarde una cantidad de bits igual a la información que contenga el mismo. De hecho justamente esa era la idea original de la compresión, poder hallar una codificación alternativa de los valores sin perder la información que estos nos daban. Todo lo que no es información es redundancia, se están utilizando más bits que los necesarios. Por ende será tarea de los compresores

quitar la mayor redundancia posible para obtener un archivo de longitud menor pero con la misma información. Resta entonces encontrar un método que permita utilizar los conceptos de la entropía para comprimir un archivo, y esto es exactamente lo que veremos en la próxima sección

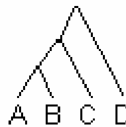
Compresores Estadísticos

Los compresores estadísticos reciben su nombre por basarse en las probabilidades de los símbolos a comprimir, emitiendo códigos de menor longitud para símbolos mas probables y de mayor longitud para los más improbables. Esta longitud se calcula intentando ser lo más fieles a la longitud dada por la información del símbolo, ya que haciendo eso se estará aproximando a la entropía del archivo.

Huffman

En este método se utiliza un árbol binario (árbol de Huffman, nombre que obtuvo de quien lo creara en 1952) en el cual cada hoja es representada por uno de los posibles símbolos a comprimir y su codificación en bits está dada por el recorrido que se debe efectuar para llegar a la hoja desde la raíz, emitiendo un "0" cuando se sigue el arco izquierdo y un "1" cuando se sigue el derecho (esto es únicamente por convención, tranquilamente se puede seguir la inversa con los mismos resultados, salvo que el resultado será el de aplicar un NOT binario al archivo)

Por ejemplo, si se tiene el árbol:



Las codificaciones de los símbolos son:

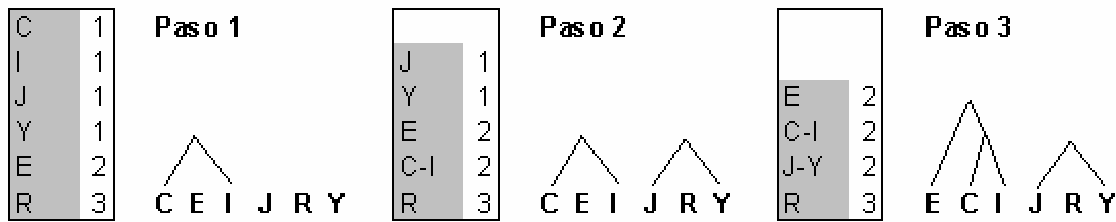
A: 000 B: 001 C: 01 D: 0

Un árbol de Huffman tiene una propiedad muy importante y es que genera códigos prefijos, puesto que para que un código de un símbolo comenzara con un código de un segundo símbolo, este tendría que tener un arco hacia la hoja del primero o hacia un padre de este, con lo que no estaría en un nodo hoja y por lo tanto no cumpliría la definición de árbol de huffman. Sin embargo, nos interesa también que las longitudes de los códigos estén relacionadas con su probabilidad (de acuerdo con la información del símbolo), por lo que necesitamos un método para construir, partiendo de las mismas, el árbol de Huffman correspondiente.

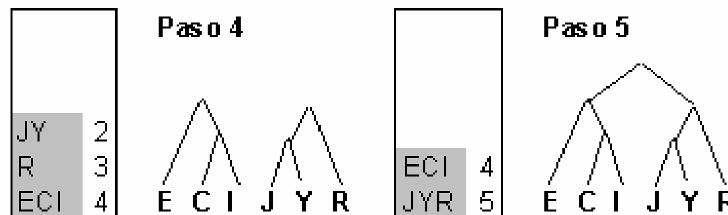
Lo primero que necesitamos es conocer la frecuencia de cada símbolo (utilizaremos frecuencias en vez de probabilidades ya que son proporcionales y más fáciles de manejar), para lo cual tendremos que hacer una primera pasada por el archivo leyendo caracter por caracter y guardando su frecuencia. Luego con estas se procede a armar el árbol, tomando los dos caracteres con menor frecuencia y generando un nodo padre. Luego se repite el proceso, pero en vez de considerar a estos dos símbolos se considerará solamente al padre, que tendrá asociada una frecuencia igual a la suma de las frecuencias de los nodos que une. El proceso finaliza cuando quedan 2 símbolos y se los une, generando la raíz del árbol

Para hacerlo algorítmicamente utilizaremos una lista símbolo – frecuencia ordenada por este último campo. El nodo padre que se cree en cada paso tendrá como hijo izquierdo al primero de la lista y como hijo derecho al segundo de ella. Cuando este nodo se agregue a la lista, en caso de existir otros nodos con frecuencia igual, se agregara a continuación de ellos para no ser elegido primero y generar códigos demasiado largos. Igualmente es necesario aclarar que tanto en este punto como en todos los siguientes, se da una visión teórica del método, sin querer decir en ningún momento que la única opción para implementar un Huffman es utilizando un árbol binario con una lista; de hecho suelen ser mucho mas eficientes otras opciones, pero que de usarse dificultarían la comprensión del método.

Como ejemplo de construcción de Huffman comprimiremos el archivo “JERRYRICE”, en homenaje a este famoso receptor:



En el primer paso, luego de obtener las frecuencias de cada símbolo, se genera un nodo con C e I como hijo. Este nodo tiene frecuencia 2 y se agrega a la lista a continuación de la E que tiene la misma frecuencia. Luego en el segundo paso se efectúa lo mismo para los nodos J e Y. En el tercer paso, se elige como hijo izquierdo al E y como hijo derecho al C-I. Es bueno notar que si al generarse C-I y J-Y se hubieran agregado en la lista antes que el E, estos dos habrían sido elegidos y tendríamos 4 símbolos a una distancia 2 del mismo padre, en cambio ahora tenemos al E a una distancia de 1 arco y al C e I a una distancia de 2, lo que generará finalmente códigos menores. Para una mejor apreciación del árbol, se cambio de lugar a la E, de modo que los arcos no se crucen entre sí



Luego del tercer paso, los dos nodos de menor frecuencia son el J-Y y el R, con lo que se unen, llegando al paso final en el que solo quedan 2 nodos que al unirse forman la raíz del árbol. Luego de las operaciones se llega a :

Car.	representación en Huffman	Longitud	Probabilidad	información	Diferencias en bit (Info – Huff)
C	010	3	1/9	3.17	+0.17
E	00	2	2/9	2.16	+0.16
I	011	3	1/9	3.17	+0.17
J	100	3	1/9	3.17	+0.17
R	11	2	3/9	1.58	-0.42
Y	101	3	1/9	3.17	+0.17

La representación final del archivo comprimido es:

100-00-11-11-101-11-011-010-00

Comparando con la entropía, en 4 de los 9 casos hemos ahorrado 0.17 bits, en 2 (las 2 apariciones de la E) 0.16 bits pero en 3 casos (con la R) hemos perdido 0.42 bits, con lo que al final el archivo quedará 0.26 bits peor que la entropía (21.74 bits que es el mínimo dado por la entropía contra 22 bits que ocupó finalmente el archivo). Sin embargo el resultado que obtuvimos es muy bueno, especialmente si consideramos que siempre tendremos que redondear a una cantidad entera de bits (y redondear para arriba, o estaremos perdiendo información)

La descompresión es sencilla, simplemente se debe formar el mismo árbol que se formo en la compresión y se lee bit a bit, recorriendo el árbol de Huffman. Cada vez que se llega a un nodo, se emite el caracter correspondiente y en el próximo paso se comienza desde la raíz. Sin embargo, el problema es como construir inicialmente el árbol de Huffman. Para comprimir, con una leída del

archivo bastaba pero en la descompresión no se cuenta con el archivo original y los bits no pueden ser interpretados sin el.

La única solución que queda es que el compresor agregue a la tira de bits la tabla final de frecuencias para que el descompresor pueda generar el mismo árbol de huffman. Si bien esto hace que el archivo final sea más grande, puede considerarse despreciable en archivos de tamaño normal (si se guardan las frecuencias como enteros tenemos 256 caracteres * 4 bytes = 1024 bytes, que no influyen mucho si al comprimir un archivo de 10 megas se llega a uno de 2 megas o de 2,001 megas)

El método de huffman que hemos hecho hasta ahora es conocido como huffman estático por utilizar siempre la misma tabla. El gran problema que tiene es el hecho de necesitar recorrer el archivo inicialmente para generar el árbol. Esto implica una pérdida de tiempo notable cuando se lo utilice para archivos grandes y una imposibilidad de utilizar en métodos que no permiten doble lectura de los datos (por ejemplo, si se quiere anexar un modulo compresor para una tira de bytes que saldrán por el socket de una máquina a la otra). Existe una alternativa, el huffman dinámico, muy similar al estático, que veremos mas adelante ya que primero es necesario analizar un último problema que se encuentra al generar el archivo comprimido

Representación de bits en bytes

Cualquier filesystem que se precie de serlo hoy por hoy nos permite guardar un archivo de la cantidad de bytes que deseemos (con un límite obviamente), sin hacernos que sea necesario generar archivos múltiplos del tamaño del sector del disco. Sin embargo, ninguno (al menos conocido por el autor al momento de confeccionar el apunte) permite guardar una cantidad dada de bits. Esto es más que nada porque en la gran mayoría de los casos no es algo necesario e implicaría agregado de información de control innecesaria

Sin embargo, cuando guardamos los bits del archivo comprimido tendremos que de alguna forma poder distinguir entre los bits que nos interesan y el "relleno" que haya que insertar para llegar a una cantidad entera de bytes. En el ejemplo visto, el archivo comprimido (sin la tabla de frecuencias, que debe anexarse) ocupaba 22 bits, con lo que necesitaremos 3 bytes y nos sobraran 2 bits. Si esos 2 bits se guardan como "00" podrían interpretarse como una "E" por el descompresor y generar entonces un archivo descomprimido incorrecto

Es necesario indicar que esos dos bits finales no deben ser considerados por el descompresor, las alternativas que tenemos son 3. La primera es agregar a la tira de bits un bit 1 y luego tantos bits 0 como sea necesario para llenar el archivo. A continuación se muestra como quedaría para el archivo de ejemplo y para otro que justo ocupara una cantidad entera de 1 byte:

1 0 0 0 0 1 1 1	1 1 0 1 1 1 0 1	1 0 1 0 0 0 1 0

1 0 0 0 0 1 1 1	1 0 0 0 0 0 0 0

El descompresor deberá primero que nada leer el ultimo byte del archivo y desde el último bit ir hacia atrás hasta encontrar un bit 1. La tira de bits a considerar será desde el inicio hasta el bit anterior de ese 1. Salvo en el caso de que los bits sean múltiplos de 8, no se estará desperdiciando ningún byte de mas (en el caso de 7 bits, solo se agrega un uno al final y ningún cero), por lo que el desperdicio es de 1 bit en promedio (1 byte en 1/8 de los casos, 0 en el resto), lo que es más que aceptable. Sin embargo, como veremos mas adelante, esta técnica no es aplicable en algunos compresores (como ser el aritmético)

La segunda alternativa es mucho mas simple de programar, y pese a que tiene un desperdicio mayor (4 bytes generalmente), este es tan pequeño que se considera despreciable. La alternativa es la de guardar la cantidad de caracteres comprimidos, que como mucho será del tamaño de un entero. El descompresor entonces parará cuando descomprima N caracteres y los bits que queden no serán interpretados, por lo que pueden valer cualquier cosa. En el caso del huffman estático contamos con una gran ventaja y es que si sumamos la frecuencia de cada caracter (que podemos obtener de la

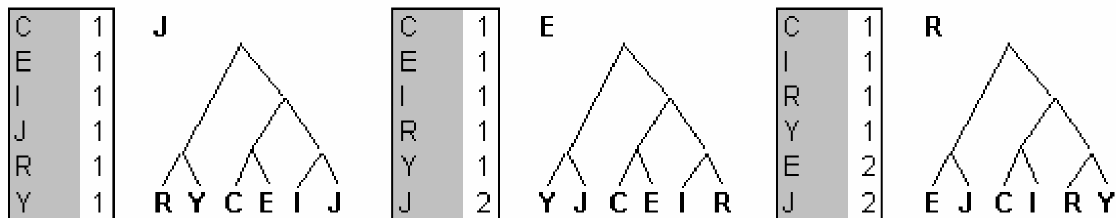
tabla de frecuencias que guardamos junto con la tira de bits) tendremos la longitud del archivo inicial por lo que no es requerido ningún otro tipo de dato. Con otros compresores no será tan simple con lo que necesitaremos usar algunas de estas técnicas

Finalmente la tercer alternativa es inventar un caracter extra que corresponda al fin de archivo (EOF). Este caracter será tratado como si fuera uno mas de los 256 del archivo de entrada y solo se comprimirá cuando se detecte que se llegó al final del archivo a comprimir. Por tener una muy baja probabilidad (1 / (tamaño del archivo + 1)), la compresión de este caracter utilizará una gran cantidad de bits, pero generalmente menos que 4 bytes, con lo que es una opción a tener en cuenta cuando no tengamos el dato de la cantidad de caracteres como en huffman estático

Huffman dinámico

Como vimos antes, el huffman dinámico tiene la ventaja de no necesitar una recorrida inicial para generar el árbol, sino que en cada compresión de caracter generará un árbol de acuerdo a las frecuencias de caracteres que haya visto hasta ese momento. Entonces, se deberá ir acumulando las frecuencias de cada caracter y con el método anteriormente visto generar un árbol de Huffman para las frecuencias encontradas, luego emitir el código correspondiente al caracter a comprimir y actualizar la tabla (y el árbol) sumándole 1 a la frecuencia de dicho caracter. Es importante notar que la actualización de la frecuencia debe ser hecha después de comprimir el caracter ya que si se actualizara antes, el descompresor no sabrá cual era ese caracter y por ende no podría conocer la tabla actualizada hasta haberlo descomprimido

Inicialmente, la tabla debe contener frecuencia 1 para cada uno de los caracteres, esto es así porque de tener frecuencia 0, los padres siempre tendrían frecuencia 0+0= 0 y no se estaría estimando correctamente la cantidad de niveles del subárbol correspondiente. La compresión del mismo archivo con huffman dinámico es:

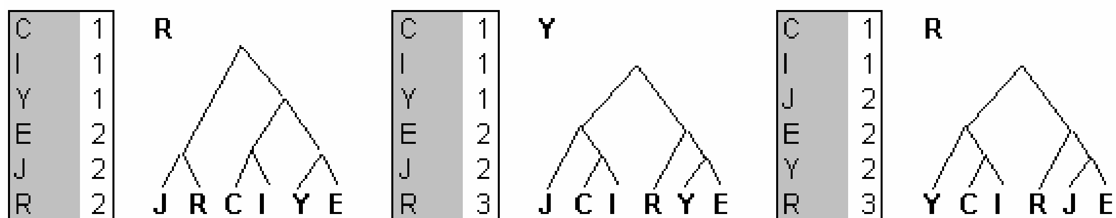


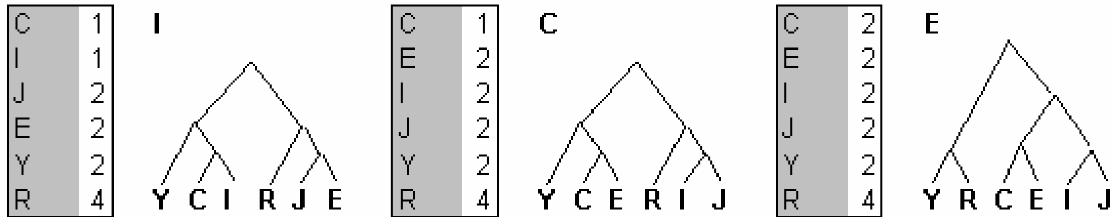
Es importante notar que en la imagen anterior se muestra el árbol de huffman completo para la tabla de frecuencias que se encuentra a su izquierda. No se debe confundir con las imágenes anteriores que mostraban paso a paso la construcción del árbol (aquí se muestra el árbol final únicamente)

Cuando se lee la J se utiliza el árbol formado por considerar que todos los caracteres tienen frecuencia 1. Como nota aparte , en realidad se tendrían que tomar a los 256 caracteres pero esto tornaría en árboles ilegibles, con lo que para la simplicidad del apunte se utilizaran únicamente los 6 caracteres del archivo.

El código de la J en el árbol armado (el primero de la imagen) es 111, con lo que se emiten dichos 3 bits y se actualiza la frecuencia de la J que pasa a ser 2, formándose un nuevo árbol (segundo en la imagen). El siguiente caracter a comprimir es la E, que se codifica como 101. Luego de actualizar el árbol con la frecuencia 2 para la E, se comprime la R como 110

Siguiendo hasta completar todos los caracteres se llega a:





El archivo comprimido entonces es:

111-101-110-01-110-10-011-010-101

Con una longitud de 25 bits, 3 mas que lo que ocupaba con huffman estático. El descompresor partirá también de un árbol formado por todos los caracteres con frecuencia 1, recorrerá el árbol hasta una hoja, emitirá el caracter correspondiente y aumentará su frecuencia en 1, reconstruyendo el árbol para utilizarlo para descomprimir el siguiente caracter

En general el huffman dinámico es peor que el estático en cuanto a nivel de compresión, aunque no en una forma crítica, ya que a medida que se vayan comprimiendo caracteres el árbol formado se asimilará bastante al árbol del huffman estático. Como gran ventaja está el hecho de no necesitar recorrer el archivo inicialmente, y por ello tampoco se necesita guardar ninguna tabla de frecuencias. Sin embargo, pese a que podría pensarse que esta ultima ventaja lo hace mas rápido, esto no es así por un gran problema que tiene, que es el tiempo consumido en generar cada árbol, tiempo que se repite 1 millón de veces en un archivo tan pequeño como de 1 MB. Hay varias opciones para intentar generar de forma más rápida pero solo la ultima que veremos no implica perdida en el nivel de compresión.

La primera opción es tan simple que no merece un apartado. Consta en reconstruir el árbol únicamente cada N caracteres, por ejemplo cada 10. Las frecuencias igual se van guardando pero durante tandas de 10 caracteres se utiliza el mismo árbol. Esto reduce el tiempo exactamente en un factor de N, pero perjudica bastante en el tamaño final del archivo ya que se puede estar utilizando un árbol muy malo por un tiempo muy grande. Las otras dos opciones que veremos son los códigos de shannon fano, una forma alternativa y mas rápida de crear un árbol similar al de huffman, y finalmente un manejo mas eficiente del árbol de huffman para evitar una reconstrucción total en cada paso

Códigos de Shannon Fano

Tal como dijimos anteriormente, los códigos de shannon fano surgen de un árbol similar al de Huffman pero de construcción mucho más rápida. El árbol de shannon fano se construye agrupando primero a todos los caracteres en un conjunto, que luego es dividido en 2 conjuntos de frecuencias similares. Se genera un nodo del árbol cuyo hijo izquierdo es el primer conjunto y su hijo derecho el segundo. Este procedimiento se repite con los 2 conjuntos que quedan recursivamente hasta que todos los conjuntos queden con 1 solo caracter. En ese momento tendremos el árbol de shannon fano

La cuestión cae en como generar algorítmicamente 2 conjuntos de frecuencia similar, con lo que hay que conseguir algún tipo de heurística (forma de hacer las cosas) que sea rápida y a la vez tenga buenos resultados. Una de ellas puede ser la siguiente.

- Para subdividir el conjunto A en dos conjuntos A1 y A2, primero se genera el conjunto vacío A1 y a A2 se le agregan todos los elementos de A (o sea, A2 = A)
- Mientras la frecuencia total del conjunto A1 (la suma de todos sus elementos) sea menor a la de A2, se pasa el elemento de mayor frecuencia de A2 a A1. En caso de haber 2 con la misma frecuencia, se pasa el menor en orden alfabético.

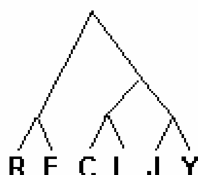
Esta heurística es bastante simple y puede utilizarse con el archivo de ejemplo:

- Inicialmente se pasa del subconjunto completo el caracter R (frecuencia 3), luego el E (frecuencia 2). En este momento la frecuencia de A1 (que es 5 por tener a R y E) es mayor a la de A2 (que es 4 por tener a C, I, J y Y) con lo que se para y se subdividen

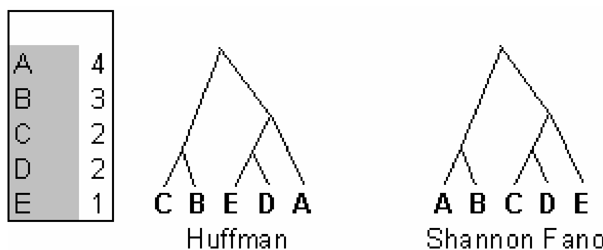
• Para el conjunto A11, se agrega la R con frecuencia 3 y ya es mayor al A12 que tiene solo a la E con frecuencia 2

• Para el conjunto A21 se agregan la C y la I, y ahí la frecuencia deja de ser menor para ser igual entre ambos conjuntos (frecuencia 2)

• A21 y A22 se subdividen en 2 subconjuntos cada uno con un solo caracter. Ha terminado el método con el siguiente árbol:



El archivo comprimido seria 110-01-00-00-111-00-101-100-01 que ocupa 22 bits, igual que el de huffman estático. Sin embargo, esto no fue mas que una coincidencia ya que en la mayoría de los casos el utilizar códigos de shannon fano implica utilizar mas bits que con árboles de Huffman, y además la diferencia en tiempos de construcción es muy grande. Veremos a continuación un ejemplo en el que usando las técnicas dadas, se nota la diferencia entre árbol de huffman y árbol de shannon fano, el archivo será “AAAABBBCCDDE”:

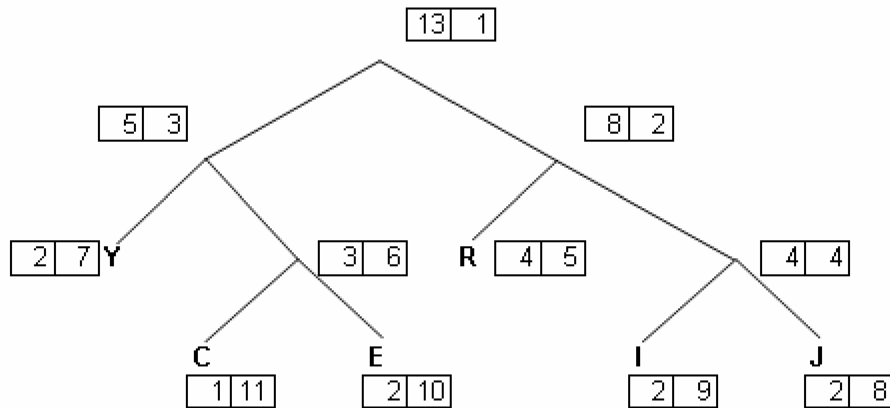


Aquí se puede apreciar la diferencia. Utilizando huffman el archivo comprimido es 11-11-11-11-01-01-01-00-00-101-101-100, ocupando entonces 27 bits; en cambio con shannon fano se tiene 00-00-00-00-01-01-01-100-100-101-101-11, dando un total de 28 bits, 1 bit peor lo que no es poco considerando el tamaño pequeño del archivo

Manejo eficiente del árbol

Con un poco de información de control agregada, se puede lograr una actualización rápida del árbol de huffman sin tener que reestructurarlo en su totalidad, lo que resulta muy costoso debido a que tiene 256 hojas (257 si se agrega el EOF) y por ende 511 o 513 nodos. Si se vuelve al ejemplo de huffman dinámico, se verá que desde el paso de la segunda R se utilizaron 3 árboles distintos, pero sin embargo para todos árboles la R y la J se emiten con 2 bits, y el resto de los caracteres con 3. Esta reestructuración de árboles fue absolutamente inútil ya que de no hacerse el archivo final hubiera ocupado lo mismo. Con un manejo eficiente del árbol nos ahorraremos entonces estas pérdidas de tiempo sin perder nivel de compresión del archivo. Para ello querremos que si dos códigos tienen la misma frecuencia, su longitud en bits no varíe en más de 1, diferencia aceptable ya que en huffman se suele beneficiar al ultimo código elegido de los que tienen igual frecuencia (para ver un ejemplo, hacer un árbol de 3 símbolos con frecuencia 1 y se vera que el mayor de ellos tendrá longitud 1 y el resto longitud 2). Lo que si no se querrá de ninguna manera es que un código con frecuencia mayor a otro tenga una longitud en bits mayor!

Veamos algunas propiedades del árbol de huffman. Lo primero que se ve es que los nodos o bien tienen 2 hijos, o no tienen ninguno. Una segunda propiedad requiere inicialmente que numeremos todos los nodos del árbol en forma ascendiente partiendo de la raíz y numerando los nodos de nivel a nivel, de derecha a izquierda. Como ejemplo tomaremos el árbol utilizado para comprimir la C en huffman dinámico, que repetimos a continuación, pero estructurado por niveles y agregando a cada nodo 2 números, el primero su frecuencia y el segundo la numeración:



Las frecuencias de cada nodo se repiten en la siguiente tabla:

Nodo	Freq	Nodo	Freq	Nodo	Freq
1	13	5	4	9	2
2	8	6	3	10	2
3	5	7	2	11	1
4	4	8	2		

Ahora si podemos enunciar la segunda propiedad de los árboles de huffman: Para 2 nodos cualesquiera X e Y, si el numero de nodo de X es menor al numero de nodo de Y, entonces la frecuencia de X es mayor o igual a la frecuencia de Y. O, dicha de otra forma, al recorrer los nodos por numero de nodo ascendiente, se los estará recorriendo también por frecuencia descendiente. Si se cumplen las dos propiedades vistas, entonces estaremos ante un árbol de Huffman.

Aprovechándonos de esta última propiedad, analizaremos que es lo que pasa cuando se agrega la frecuencia a un caracter, y por ende a su nodo asociado con numero N. Para que se siga cumpliendo la segunda propiedad (la primera se cumple porque la estructura no fue cambiada), se debe comparar la nueva frecuencia del nodo cambiado N con la del nodo N-1. Si se sigue cumpliendo que freq(N-1) es mayor o igual a freq(N), también se cumplirá para los nodos anteriores a N-1 ya que su frecuencia también es mayor o igual a la del mismo, que no cambió. también se cumple para los nodos siguientes a N, puesto que freq(N) ya era mayor o igual antes de aumentarse en uno. Entonces con este simple checkeo con la frecuencia del nodo anterior se puede saber si se siguen cumpliendo ambas reglas y por ende si el árbol sigue siendo un árbol de huffman. De darse este caso, entonces no se debe reestructurar

Veamos un ejemplo, aumentando la frecuencia de Y y por ende la del nodo 7 que pasa a ser 3. también se deben actualizar las frecuencias de los nodos que hay en el camino del 7 a la raíz, con lo que la frecuencia del nodo 3 pasa a ser 6 y la frecuencia de la raíz, 14. Las frecuencias de cada nodo se muestran en la siguiente tabla:

Nodo	Freq	Nodo	Freq	Nodo	Freq
1	14	5	4	9	2
2	8	6	3	10	2
3	6	7	3	11	1
4	4	8	2		

Para el nodo 6 se cumple que su frecuencia (3) es mayor o igual a la del 7 (también 3). Se puede ver que ocurre lo mismo para todos los nodos anteriores al 7 (es necesario verificar ya que el cambio puede afectar a los padres del 7). Para los nodos mayores a 7 no es necesario revisar que se

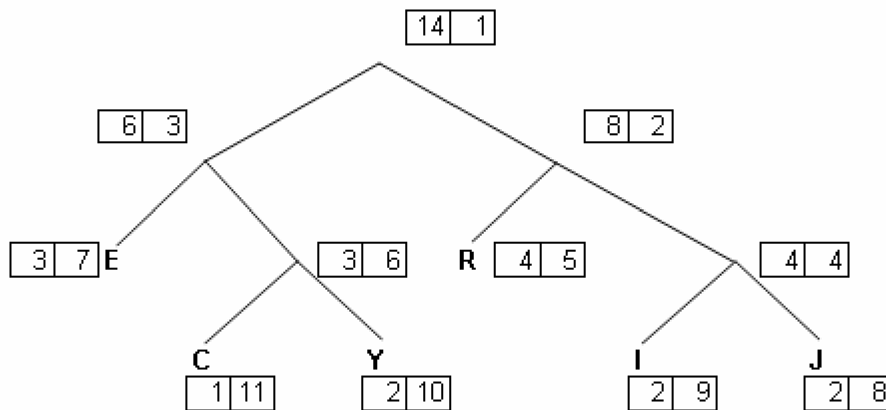
cumpla la propiedad porque es trivial, si se aumenta la frecuencia del 7, seguirá teniendo mayor frecuencia que los nodos de numero mayor. El árbol entonces sigue siendo un árbol de huffman y no se debe reestructurar. Queda como ejercicio para el lector generar el árbol con la nueva frecuencia de Y y confirmarlo. Recordemos que a veces pueden darse cambios en el árbol pero lo que nos interesa es que para códigos de misma frecuencia la variación en longitud de bits no sea mayor a 1 entre distintos árboles

Ahora veamos como actuar cuando al modificar una frecuencia se deja de cumplir que es menor o igual que la del nodo anterior. Para ello volveremos al árbol inicial con frecuencia 2 para Y y aumentaremos en 1 la frecuencia de E, y por ende la del nodo 10. también se deberán aumentar las frecuencias de los nodos que están en el camino a la raíz, que son el 3, el 6 y el 1. La tabla de frecuencias por nodo queda de la siguiente forma:

Nodo	Freq	Nodo	Freq	Nodo	Freq
1	14	5	4	9	2
2	8	6	4	10	3
3	6	7	2	11	1
4	4	8	2		

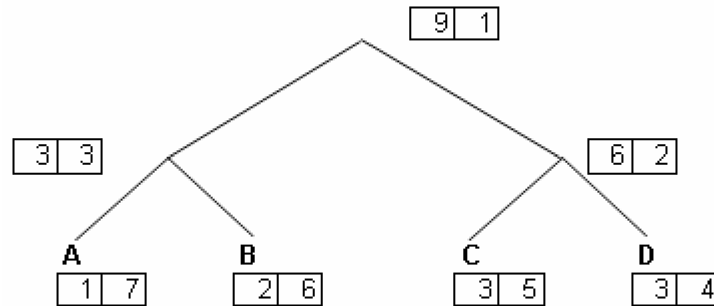
En este caso la frecuencia del nodo 9 (2) no es mayor o igual a la del nodo 10 (3) con lo que se necesita modificar el árbol para que se siga cumpliendo. Lo que se hace, en vez de una reestructuración completa del árbol, es efectuar un intercambio de nodos entre el nodo que no cumple (en este caso el 10) y el nodo con menor numeración que tenga frecuencia menor a dicho nodo (en este caso, el nodo 7). Al intercambiar los nodos se debe recalcular la frecuencia de nodos no-hoja que habrán cambiado (en este caso, la del 6 vuelve a su valor inicial de 3) y se debe verificar que se siga cumpliendo la segunda propiedad de los árboles de huffman o haciendo un nuevo intercambio hasta que eso ocurra

Luego del intercambio de nodos, queda el siguiente árbol:

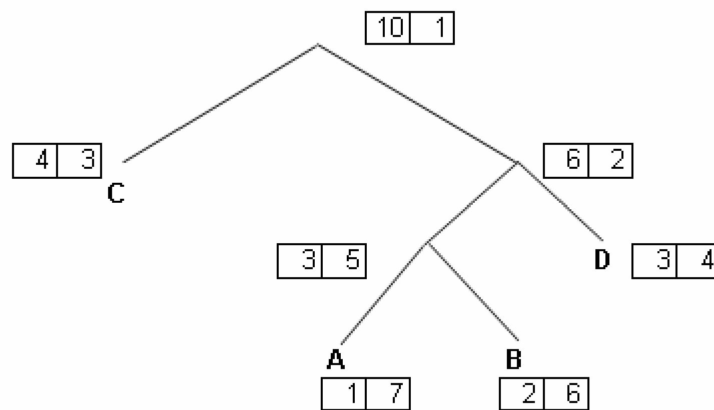


Se puede verificar que cumple las 2 propiedades con lo que es un árbol de Huffman. En vez de hacer una reestructuración completa, con 1 simple intercambio logramos el mismo objetivo

Finalmente, veremos un caso extra que muestra el intercambio de nodos hoja con nodos no-hoja. Sea el siguiente árbol, con su numeración de nodos y sus frecuencias:



Ahora si le aumentamos en 1 la frecuencia a C, se deja de cumplir la propiedad ya que la frecuencia del nodo 4 será menor a la del nodo 5. Al intentar intercambiarlo, el nodo con menor numero con menor frecuencia que 4 (la nueva frecuencia del nodo 5) es el 3. El intercambio de nodos se hace tal que quede el siguiente árbol:



Este nuevo árbol cumple con las dos propiedades, por lo que es un árbol de huffman.

Half coding (1/2 coding)

Este método, desarrollado por David Wheeler, es únicamente eficiente cuando un caracter es mucho mas probable que el resto. Con un huffman normal, cuando tenga probabilidad mayor a ½ se lo comprimirá siempre con un bit, redondeando hacia arriba valores que pueden ser mucho menores a 1. Para estos casos existe la técnica de half coding que generalmente permite un ahorro de bits.

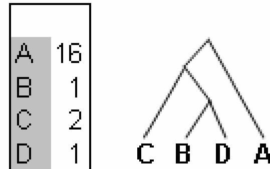
En principio analicemos la situación: cuando un caracter tiene probabilidad mayor a ½ significa que hay una gran posibilidad de que se den secuencias seguidas del mismo. De hecho, ya sabiendo que mas del 50% de los caracteres de un archivo son el mismo, si o si se deberá dar al menos una ocurrencia de 2 caracteres seguidos. El half coding lo que hace es en vez de codificar el caracter mas probable con una tira de bits, representa de una cierta forma la cantidad de veces que se repite dicho caracter. Para ello, en la construcción del árbol no utiliza el caracter mas probable sino que en su reemplazo utiliza otros dos símbolos extra (no deben ser valores posibles del archivo) que denominaremos genéricamente α y β. El reemplazo del caracter mas frecuente por dichos símbolos extras se hace de la siguiente forma:

- Para una tira de dicho caracter, se obtiene su longitud, que llamaremos L
- Se representa L+1 en binario
- Se quita el primer bit de esta representación, que sí o sí es un 1 (el menor valor de L es 1 y al representar 2 en binario se tiene “10”), con lo que no es necesario guardarlo
- Se reemplazan los 0 de la cadena por el símbolo α y los 1 por el β

Veámoslo con un ejemplo. Si tenemos el archivo:

AAAABAACAAAAAAACDAAA

Con un huffman estático se lo comprime de la siguiente forma:



1-1-1-1-010-1-1-00-1-1-1-1-1-1-1-00-011-1-1-1

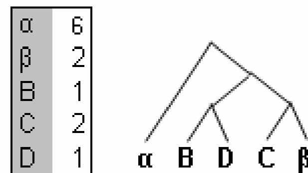
El archivo final ocupa 26 bits. Ahora veamos como queda luego de efectuar la conversión del half coding. En el archivo tenemos 4 tiras de caracteres A:

- Primero hay una tira de 4 A, con lo que se representa 5 en binario. A dicha tira (101) se le quita el primer 1, quedando 01, que se intercambia por α β
- Para la segunda tira, de 2 A, se representa 3 en binario (11) y se quita el primer 1, con lo que se intercambiará por β
- Para la tercer tira, de 7 A, se representa 8 en binario (1000), que quitando el primer 1 y transformándolo queda representado como α α α
- Finalmente, la ultima tira es de 3 A, con lo que al 4 en binario (100) se le quita el primer 1 y se lo intercambia como α α

El archivo luego de los intercambios queda de la siguiente forma:

α β B β C α α α C D α α

Comprimiendo el archivo con huffman llegamos a lo siguiente:



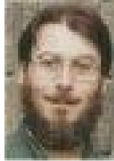
0-111-100-111-110-0-0-0-110-101-0-0

El archivo comprimido ahora ocupa 24 bits, 2 menos que antes, lo que no es poco (8% menos). En general el proceso de half coding se hace dinámicamente, para lo que el compresor cuando reconoce un caracter con alta frecuencia comienza a utilizar este método hasta que detecte que su probabilidad bajo de ½. Si se hiciera estático hay que tener en cuenta que se debe guardar cual fue el caracter más frecuente, ya que de otro modo nunca se sabría si fue realmente una A o cualquier otro caracter como la G. En cambio en forma dinámica, como el método solo se empieza a hacer al detectar un caracter con alta probabilidad, el descompresor tendrá el árbol del paso anterior y podrá ver cual es el caracter con probabilidad alta y que por ende fue reemplazado por los símbolos α o β

La descompresión de esta clase de archivos no es complicada, el único cambio es que cuando descomprimen cualquiera de los dos caracteres especiales en vez de emitir algo siguen descomprimiendo hasta encontrar un caracter no especial. En memoria van almacenando la tira de caracteres especiales descomprimida, a la que luego cambian las α por 0, las β por 1, agregan un 1 al inicio y transforman de binario a decimal, obteniendo L+1. Luego emiten L veces el caracter más probable.

Compresión Aritmética

La compresión Aritmética tiene su nacimiento para la informática en IBM. Por el año 1987 esta empresa, patenta los métodos I.E. Wittem (primera foto), R.Neal (segunda foto) y J. Cleary, que son diferentes formas de aplicar la compresión aritmética, con nombres elegidos en honor a los investigadores que los desarrollaron.



En definitiva el aritmético es otro método de compresión estadística, dado que se basa en las frecuencias de aparición de los mensajes. Pero, a diferencia de los árboles de compresión, esta manera de generar los códigos logra los objetivos de la teoría de Shannon, llegando a comprimir un mensaje en aquella longitud L_i definida al comienzo del apunte, siendo ésta un valor no entero de bits en la mayoría de los casos

Para entrar en los detalles debemos aclarar que esta técnica es muy diferente a la de los árboles, pero también tiene una forma estática y otra adaptativa:

- **Método estático:**

Se parte de un intervalo de números reales entre el cero y el uno (matemáticamente [0;1)) en el cual se distribuyen las probabilidades de ocurrencia de los simbolos, es decir, se subdivide el intervalo en porciones en las que la relación de tamaño entre la misma y el intervalo completo está dada por su probabilidad. Como ejemplo, si un símbolo tiene una probabilidad de 1/3, la subdivisión correspondiente al mismo ocupará un tercio del largo del intervalo. Llamaremos de ahora en más piso del intervalo al mínimo valor del mismo y techo al máximo. El piso y el techo de este intervalo inicial son entonces 0 y 1:

A continuación se lee el primer símbolo y se observa dentro de qué rango de valores está desplegada su probabilidad. Ése rango pasará a ser el nuevo intervalo general (su piso es ahora el piso general reemplazando al cero y su techo pasa a ser el techo total reemplazando al uno) y sobre el nuevo intervalo se distribuyen las mismas probabilidades de aparición de símbolos del comienzo. Una vez completa esta operación se leerá el siguiente símbolo y se repetirá el ciclo de actualización del intervalo.

Cuando se llega al fin de archivo lo único que resta hacer es elegir un valor (un número) que esté entre el techo y el piso del intervalo general resultante y guardarlo en disco (emitirlo).

Como ejemplo, vamos a comprimir el archivo "DIVIDIDOS".

La probabilidad de los caracteres es la siguiente:

Carácter (mensaje)	Frecuencia de aparición	Probabilidad de aparición
D	3	1/3
I	3	1/3
O	1	1/9
S	1	1/9
V	1	1/9

La compresión del archivo se da de la siguiente forma:

<table>
<tr><th></th><th>I</th><th>II</th><th>III</th><th>IV</th><th>V</th></tr>
<tr><td>-</td><td>1,000</td><td>1,0000</td><td>0,888889</td><td>0,790123</td><td>0,786008</td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>D</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>-</td><td>2/3</td><td>0,888889</td><td>0,851852</td><td>0,786008</td><td>0,784636</td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>I</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>-</td><td>1/3</td><td>0,777778</td><td>0,814815</td><td>0,781893</td><td>0,783265</td></tr>
<tr><td>O</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>-</td><td>2/9</td><td>0,740741</td><td>0,802469</td><td>0,780521</td><td>0,782807</td></tr>
<tr><td>S</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>-</td><td>1/9</td><td>0,703704</td><td>0,790123</td><td>0,779150</td><td>0,782350</td></tr>
<tr><td>V</td><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>-</td><td>0,000</td><td>0,666666</td><td>0,777778</td><td>0,777778</td><td>0,781893</td></tr>
<tr><td></td><td>Leo la “D”</td><td>Leo la “I”</td><td>Leo la “V”</td><td>Leo la “I”</td><td>Leo la “D”</td></tr>
</table>

<table>
<tr><th></th><th>VI</th><th>VII</th><th>VIII</th><th>IX</th><th>X</th></tr>
<tr><td>-</td><td>0,786008</td><td>0,786008</td><td>0,785551</td><td>0,785449</td><td>(0,785436)</td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>D</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>-</td><td>0,785551</td><td>0,784636</td><td>0,785500</td><td>0,785443</td><td></td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>I</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td></td><td>|</td><td>|</td><td>|</td><td>|</td><td>0,785435</td></tr>
<tr><td>-</td><td>0,785094</td><td>0,783265</td><td>0,785432</td><td>0,785438</td><td></td></tr>
<tr><td>O</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>-</td><td>0,784941</td><td>0,782807</td><td>0,785416</td><td>0,785436</td><td></td></tr>
<tr><td>S</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>-</td><td>0,784789</td><td>0,782350</td><td>0,785416</td><td>0,785434</td><td></td></tr>
<tr><td>V</td><td>|</td><td>|</td><td>|</td><td>|</td><td></td></tr>
<tr><td>-</td><td>0,784636</td><td>0,781893</td><td>0,785399</td><td>0,785432</td><td>(0,785434)</td></tr>
<tr><td></td><td>Leo la “I”</td><td>Leo la “D”</td><td>Leo la “O”</td><td>Leo la “S”</td><td>EMITIDO</td></tr>
</table>

Como observación, los intervalos son a cada lectura más chicos. Como se puede apreciar, para probabilidades más chicas se genera un intervalo con piso y techo más parecidos, ya que el intervalo es más chico y hay poca diferencia entre piso y techo. Mientras más parecidos sean el piso y el techo necesitaré más precision para representar el número final ya que debe identificar univocamente al intervalo

Se ve que mientras más grande sea el intervalo menor será la cantidad de bits necesitada para representar el numero final, con lo que hay una relación ente la probabilidad del símbolo que se emite y los bits que implican emitirlos. La relación justamente está dada por la formula de la información, es decir que si emito un simbolo con probabilidad P, estaré utilizando –log2(P) bits. A diferencia del huffman, el compresor aritmético no necesita redondear en cada caso con lo que el archivo final ocupará menos espacio por utilizar la longitud dada por la entropía y no una aproximación como hace Huffman

Como última aclaración, por ser estático este método tambien requiere del agregado de una tabla con las frecuencias leidas para cada caracter o será imposible descomprimir el archivo inicial.

- **Método Dinámico (o adaptativo):**

Se parte de un intervalo igual al estático, pero se distribuyen las probabilidades de todos los símbolos posibles, tomándolas como idénticas. Luego se lee el primer símbolo y se observa dentro de qué rango de valores está desplegada su probabilidad. Ése rango pasará a ser el nuevo intervalo general (como en el estático) y sobre el nuevo intervalo se distribuyen las probabilidades de aparición de símbolos modificadas, sumandole 1 a la frecuencia del comprimido

Una vez completa esta operación se leerá el siguiente símbolo y se repetirá el ciclo de actualización del intervalo. En general se ve que la técnica es similar a huffman en lo que se refiere a mantenimiento de frecuencias de cada caracter (ya sea para forma estática o dinámica) pero lo que cambia es que en vez de generar un arbol para la salida se utiliza un intervalo dividido.

Como ejemplo, comprimamos el mismo archivo suponiendo que sólo existen los símbolos {D, I, O, S, V }

<table>
<tr><th></th><th>I</th><th>II</th><th>III</th><th>IV</th><th>V</th></tr>
<tr><td>-
D

-
I

-
O

-
S

-
V

-</td><td>1,000
|
|
4/5
|
|
3/5
|
|
2/5
|
|
1/5
|
|
0,000</td><td>1,0000
|
|
0,933333
|
|
0,900000
|
|
0,866667
|
|
0,833333
|
|
0,800000</td><td>0,933333
|
|
0,923810
|
|
0,914286
|
|
0,909524
|
|
0,904762
|
|
0,900000</td><td>0,904762
|
|
0,903571
|
|
0,902381
|
|
0,901786
|
|
0,901190
|
|
0,900000</td><td>0,903571
|
|
0,903307
|
|
0,902910
|
|
0,902778
|
|
0,902646
|
|
0,902381</td></tr>
<tr><td></td><td>Leo la “D”</td><td>Leo la “I”</td><td>Leo la “V”</td><td>Leo la “I”</td><td>Leo la “D”</td></tr>
</table>

<table>
<tr><th></th><th>VI</th><th>VII</th><th>VIII</th><th>IX</th><th>X</th></tr>
<tr><td>-

D

-

I

-
O
-
S
-
V

-</td><td>0,903571
|
|
|
0,903492
|
|
|
0,904325
|
0,903386
|
0,903360
|
|
0,903307</td><td>0,903492
|
|
|
0,903470
|
|
|
0,903442
|
0,903435
|
0,903427
|
|
0,903413</td><td>0,903492
|
|
|
0,903485
|
|
|
0,903478
|
0,903476
|
0,903475
|
|
0,903471</td><td>0,903478**2**
|
|
|
0,903477**6**
|
|
|
0,903477**1**
|
0,903476**8**
|
0,903476**7**
|
|
0,903476**4**</td><td>(0,9034768)

0,9034767
otra elección
0,90347675

(0,9034767)</td></tr>
<tr><td></td><td>Leo la “I”</td><td>Leo la “D”</td><td>Leo la “O”</td><td>Leo la “S”</td><td>EMITIDO</td></tr>
</table>

Como observación, el método adaptativo "aprende" más lento que su par, como se nota claramente no llega a las proporciones del estático y necesita de al menos un dígito más para comprimir el mismo texto. Además, como efecto de lo antes mencionado, los intervalos son más chicos que en el estático.

Aritmética de Enteros:

El principal problema que tienen estas aplicaciones es que la cantidad de dígitos decimales de las máquinas es limitada, y eso llevaría al fracaso al modelo de compresión. Pero esto se soluciona utilizando una aritmética de enteros en lugar de los números de punto flotante.

La aritmética de enteros reemplaza a los números de punto flotante que están entre creo y uno por números enteros que están entre el cero y un número entero de varias cifras. Entonces ahora se desplazan todos los límites entre segmentos de probabilidades a valores enteros en una recta natural (en lugar de real).

Como la precisión está dada por los dígitos de los números, y estos son pocos, se emplea una técnica para independizarse de esa cantidad. Ésta consiste en emitir (guardar en disco o un buffer intermedio) los dígitos izquierdos de los valores extremos del intervalo general cuando son iguales entre sí, y recuperar la precisión de trabajo agregando por derecha tantos ceros y nueves como cifras emitimos (se agregan al piso y techo respectivamente).

Por ejemplo, sea un intervalo de comienzo con "n" dígitos:

[00000 ; 99999)
Que serían "0,**00000**000..."
y "0,**99999**999..."

Las fórmulas para el cálculo del nuevo techo y piso al emitir un símbolo son:

Rango = Techo – Piso.
Nuevo Techo = Piso + Rango * Techo(mensaje).
Nuevo Piso = Piso + Rango * Piso(mensaje).

- **Método de Rissanen – Mohiuddin (1995):**

Es un método para el cálculo de los segmentos destinados a cada probabilidad de mensaje.

Rango = 99999 – 00000 + **1** (por el 9999 periódico).
Nuevo Techo = (00000 + 100000 (Rango) * Pt) **– 1** (para recobrar el valor periódico y la proporción del intervalo).
Obs.: "Pt" es el valor del techo del mensaje.
Nuevo Piso = 00000 + 100000 (Rango) * Pp
Obs.: "Pp" es el valor del piso del mensaje.

<table>
<tr><th></th><th>III</th><th>IV</th><th>V</th><th></th><th>IX</th><th>X</th></tr>
<tr><td>-

D

-

I

-
O
-
S
-
V
-</td><td>8887
|
|
|
8516
|
|
|
8146
|
8022
|
7899
|
7776</td><td><u>7</u>899<u>9</u>
|
|
|
8586
|
|
|
8172
|
8035
|
7897
|
<u>7</u>776<u>0</u></td><td><u>8</u>586<u>9</u>
|
|
|
4486
|
|
|
3102
|
2641
|
2180
|
<u>8</u>172<u>0</u></td><td>…</td><td>3057
|
|
|
2999
|
|
|
2942
|
2922
|
2903
|
2885</td><td>(2922)

2910

(2903)</td></tr>
<tr><td></td><td></td><td>Emito el 7 y agrego.</td><td>Emito el 8 y agrego</td><td>...</td><td></td><td>Emito</td></tr>
<tr><td></td><td>Leo la “D”</td><td>Leo la “I”</td><td>Leo la “V”</td><td>...</td><td>Leo la “S”</td><td>EMITIDO</td></tr>
</table>

Emisión total: 7852910. Para emitir primero voy al siguiente paso, veo que los extremos son iguales; emito, recargo y luego distribuyo las probabilidades.

Esta técnica tiene muchas ventajas, permite comprimir cualquier archivo en cualquier máquina que tenga aritmética de enteros (por ej. micros muy chicos de tres dígitos), pero tiene un serio inconveniente a resolver:

<table>
<tr><th></th><th>III</th><th>IV</th></tr>
<tr><td>-
A
-

B

-

C

-</td><td>20005
|
19994
|
|
|
|
|
19505
|
|
|
|
|
19005</td><td>20005
| }?
20005
|
|
|
|
|
20000
|
|
|
|
|
19994</td></tr>
<tr><td></td><td></td><td>Estado de Underflow</td></tr>
<tr><td></td><td>Leo la “A”</td><td></td></tr>
</table>

¿Cuál es la probabilidad de aparecer de “A”? Lo que ocurrió es que la precisión de trabajo no permite diferenciar los extremos de los sub-intervalos. Para esto existe un proceso de “renormalización” de los números que permite seguir comprimiendo. Lo que indica es:

- Quitar los segundos dígitos izquierdos de los extremos y sumar un uno en el “contador de underflow”, que cuenta cuantos dígitos se han cortado por este problema.
- Luego se agregan por derecha tantos ceros y nueves como sean necesarios para conservar la precisión.
- Se pasa al siguiente mensaje y se verifica si los extremos están en condiciones de emitir un dígito, en caso de que así fuera se observa si ese dígito pertenecía al techo del intervalo anterior o a su piso (o sea, si fuimos hacia arriba o hacia abajo del intervalo al comprimir) y a la emisión del

dígito izquierdo común se le agregan las de tantos ceros o nueves como el contador de underflow indique (ceros si fue hacia el techo y nueves si fue hacia el piso). Y el contador regresa a cero.

Una forma práctica de renormalizar es hacerlo cuando sin coincidir los dígitos izquierdos de los extremos (no se puede emitir), los segundos izquierdos son 9 en piso y 0 en techo. Entonces preveo el problema.

Continuando con el ejemplo anterior:

	III	III	III
-	2**0**005	2005**9**	20059
A	|	|	|
-	19994		19950
	|	|	|
	|	|	|
B	|	|	|
	|	|	|
	|	|	|
-	19505		15164
	|	|	|
	|	|	|
C	|	|	|
	|	|	|
	|	|	|
-	1**9**005	1005**0**	10050
	No comprimo aún.	Elimino los segundos dígitos y actualizo.	Renormalizado Underflow = **1**
	(debo leer la “B”)	UNDERFLOW = 1	Leo la “B”

	IV	IV	...
-	**1**9950	9950**9**	
A	|	|	
-		98982	
	|	|	
	|	|	
B	|	|	
	|	|	
	|	|	...
-		75574	
	|	|	
	|	|	
C	|	|	
	|	|	
	|	|	
-	**1**5164	5164**0**	
	Emito el 1 y luego un 9.	Underflow = 0	...
	(debo leer la “B”)	Leo la “B”	…

Descompresión en Aritmético:

Para cualquier aritmético la descompresión consiste en leer el número (o un número de la cantidad de dígitos de trabajo) y fijarse en qué parte del intervalo cae, emitir el simbolo al que le corresponde el sub-intervalo que contiene al valor, actualizar los límites del intervalo, actualizar la tabla de frecuencias si el método es estático, y finalmente repetir el paso para los siguientes simbolos hasta llegar al fin de archivo.

En caso que se esté trabajando con aritmética de enteros tengo que cortar los dígitos de izquierda igual que cuando comprimo (si son iguales) para no perder el intervalo.

Para todos los métodos la descompresión es el algoritmo inverso de la compresión, pero esto no quiere decir hacer todo al revés, sino hacer lo opuesto en cuanto a la emisión, pero hacer lo mismo en lo referido al resto de los procesos del método para no perder la "historia" del código leído.

Queda para el lector descomprimir los valores obtenidos en los distintos métodos para "DIVIDIDOS".

Implementación con números binarios:

Como sabemos las computadoras son máquinas binarias, entonces debemos pasar todo esto a valores en bits.

Para ello se utiliza generalmente el valor entero estándar del lenguaje de programación de trabajo.

Por ejemplo, si tomamos un entero de 16 bits tenemos 65535 valores distintos en nuestro intervalo de trabajo y los límites pasan a ser:

[0000h ; FFFFh)

El Underflow en este caso se produce entre el 0,29xxx y el 0,30xxx que sería en binarios: (tomando los números de a bytes).

Techo = **1000 0000** 0011 0000

Piso = **0111 1111** 1010 0101

Pero también funciona el emitir los bits/bytes siguientes, según cuál estemos trabajando.

Utilización de Contextos

Hasta ahora todos los métodos de compresión que se desarrollaron se basan en la cantidad de apariciones de un mensaje. Pero tenemos al menos una característica de la fuente (el archivo a comprimir) que no hemos aprovechado, a saber la determinística, en la que puedo predecir el siguiente símbolo que vendrá. ¿Cómo hacer para aprovechar esto? Veamos a qué nos referimos con esta denominación:

Fuente 1: "AZJTDOPARTPQMASXX"
Fuente 2: "AAABBBTTTLLLQQQWW"

En la fuente "1" las apariciones de los caracteres es azarosa y de ello no podemos concluir nada, pero en la fuente "2" es claro que la aparición de una letra hace muy posible su repetición. Es a esto lo que se llama grado de determinación. Y en la compresión se lo interpreta como "contexto".

Un contexto en compresión es el conjunto de mensajes (o símbolos) que son contiguos al que se está comprimiendo según alguna relación dada (lectura secuencial del archivo, posición en una imagen, etc.). Si tomamos en cuenta el contexto como la letra anterior a la que se lee y hacemos las cuentas de la probabilidad de aparición sobre este marco nos quedaría lo siguiente:

Sin tomar contextos:

P(A) $_{\text{fuente 1}}$: f(A) / #letras = 3 / 17

P(A) $_{\text{fuente 2}}$: f(A) / #letras = 3 / 17

Con contextos:

P(A) $_{f1}$ dado "comienzo de archivo" = ¿? (no tenemos contexto)

P(A) $_{f1}$ dado "P" = luego de la "P" aparece la "A" y la "Q", entonces podemos decir que es ½ (y de 1/2 para "Q").

P(A) $_{f2}$ dado "comienzo de archivo" = ¿? (no tenemos contexto)

P(A) $_{f2}$ dato "A"= luego de una "A" aparecen una "A", otra "A" y una "B", entonces las probabilidades son "A" 2/3 y "B" 1/3.

Se puede observar que las probabilidades son mayores cuando pensamos en contextos. De hecho, en el castellano cualquier persona luego de una Q y una U puede suponer que existe una gran probabilidad de que sigan una E o una I, probabilidad que no sería tan grande si desconocieran que estas 2 letras anteceden a lo que vendrá. Como en el paradigma de compresión que estamos utilizando, siempre es mejor tener mensajes más probables (para generar archivos más chicos) y por naturaleza de la información a comprimir (en la que es común encontrar ciertas relaciones entre la secuencia de los mensajes), la utilización de contextos es una mejora indudable.

En informática se trabaja con "Contextos finitos" porque tomamos una cantidad limitada de mensajes de contexto del mensaje. En el caso de textos en compresión estadística los contextos pueden ser las palabras o caracteres anteriores al mensaje actual, según sea el tipo de mensaje elegido.

El problema de los contextos es que requieren muchos más recursos de cálculo y almacenamiento, es claro ver cómo crece exponencialmente la cantidad de memoria requerida. Por ejemplo, sin contexto necesito simplemente una tabla con la frecuencia de cada caracter. Al tener contexto de 1 caracter, necesitare 256 tablas, cada una representando las frecuencias de todos los caracteres con un contexto dado (por ejemplo, "A", "B", etc). Al usar contextos de 2 caracteres los posibles contextos son 256^2 = 65536, con lo que necesito una cantidad igual de tablas de frecuencia para trabajar

Llamaremos de ahora en más orden de un contexto es la cantidad de mensajes que se toman en cuenta como parte del contexto.Ej:

- Contexto de orden 0: Es la letra leída. "i"
- Contexto de orden 1: Es la letra leída dada la anterior. "di"
- Contexto de orden 2: Es la letra leída dadas las dos anteriores. "idi"
- Contexto de orden n: Es la letra leída dadas las "n" anteriores.

PPMC

Es uno de los algoritmos más avanzados de compresión aritmética, parte de la familia de los algoritmos PPM (Prediction by Partial Matching) y también se lo vincula con el modelo de Markov (o Markoviano), dado que las matemáticas de contextos porbabilísticos son denominadas "propiedades de Markov". Entre los autores se encuentran A. Moffat (foto 1) y T. Bell (foto 2)

Se basa en la utilización de varios niveles de contextos que se actualizan dinámicamente, los ordenes de estos niveles van desde cero (0) a seis (6), y de la compresión aritmética para codificar sus resultados. Según la experiencia, la mejor implementación es de hasta cuarto o quinto orden.

Este algoritmo tiene el problema de identificar cuándo cambiar de contexto y, como todos, cuándo es el fin de archivo. Para ello se genera un modelo de contexto llamado de "orden –1" en el que se encuentran todos los caracteres más un especial: el EOF, que se emite al llegar al fin de archivo. Otro caracter especial que aparece en

el resto de los modelos es el ESC (escape de contexto) que se emite cuando debo cambiar de contexto.

La distribución de las probabilidades es la siguiente:

• Nivel **–1**:

Aparecen todos los caracteres (ASCII más EOF) excepto el de escape, dado que de este nivel no se puede escapar.

• Nivel **0**:

Aparecen los caracteres que se fueron encontrando en el archivo que no tenían contextos superiores y el caracter de escape (ESC).

• Nivel **1**:

Aparecen los caracteres que al leerse tenían el caracter anterior como contexto máximo hasta ese momento y el escape (ESC).

• Nivel **2**:

Ídem Nivel 3, pero con dos caracteres de contexto.

• Nivel **n**:

Ídem Nivel 3, pero con "n" caracteres de contexto.

Inicio de la compresión:

La compresión en PPMC utiliza el método de exclusión para mejorar aún más las probabilidades de los caracteres en cada contexto. Si bien existen varios métodos para excluir, nosotros utilizaremos el de exclusión total ("full exclusion").

Ahora vamos a tratar de desarrollar unos pasos previos al ejemplo de una compresión real en PPMC para acercar al lector a los conceptos que en ella se vuelcan.

• **Mecanismo de exclusión:**

El principio de exclusión se basa en que un caracter que viene de un ESC de nivel superior no estaba en dicho contexto, entonces es claro que ninguno de los caracteres que están en aquél nivel pueden ser el leído.

Entonces se re-distribuyen las probabilidades de compresión para cada caso en particular, quitando del cálculo los caracteres que seguro no puede ser el actual (los que sí estaban en el contexto anterior).

• **Primeros pasos: Compresión en contexto "0" sin exclusión:**

En este caso vamos a obtener las probabilidades con las que se comprimiría un texto ("DIVIDIDOS") con un PPMC de contexto "0" (esto incluye al contexto "-1").

Notación: Lo que está entre paréntesis son las probabilidades con las que se codifica el mensaje (caracter).

Lectura	Contexto –1	Contexto 0	Emisión
D	TODOS LOS CARACTERES MÁS EL DE EOF.	ESC (1)	ESC(1),D(1/257)
I	"	ESC(1/2) D(1/2)	ESC(1/2),I(1/257)
V	"	ESC(2/4) D(1/4) I(1/4)	ESC(2/4),V(1/257)
I	"	ESC(3/6) D(1/6) I(1/6) V(1/6)	I(1/6)
D	"	ESC(3/7) D(1/7) I(2/7)	D(1/7)

		V(1/7)	
I	“	ESC(3/8) D(2/8) I(2/8) V(1/8)	I(2/8)
D	“	ESC(3/9) D(3/9) I(2/9) V(1/9)	D(3/9)
O	“	ESC(3/10)D(4/10) I(2/10) V(1/10)	ESC(3/10), O(1/257)
S	“	ESC(4/12)D(4/12)I(2/12) V(1/12) O(1/12)	ESC(4/12), S(1/257)
EOF	“	ESC(5/14)D(4/14)I(2/14) V(1/14) O(1/14) S(1/14)	ESC(5/14),EOF(1/257)

- **Compresión con contexto 3 sin exclusión:**

Ahora vamos a comprimir con tres niveles de contexto el mísmo archivo.

Lectura	Contexto –1	Contexto 0	Contexto 1	Contexto 2	Contexto 3	Emisión
D	Todos los caracteres más el EOF	ESC(1)				ESC(1), D(1/257)
I	“	ESC(1/2) D(1/2)	**D|** ESC(1)			ESC(1), ESC(1/2), I(1/257)
V	“	ESC(2/4) D(1/4) I(1/4)	D| ESC(1/2) I(1/2) **I|** ESC(1)	**DI|** ESC(1)		ESC(1), ESC(1), ESC(2/4), V(1/257)
I	“	ESC(3/6) D(1/6) I(1/6) V(1/6)	D| ESC(1/2) I(1/2) I| ESC(1/2) V(1/2) **V|** ESC(1)	DI| ESC(1/2) I(1/2) **IV|** ESC(1)	**DIV|** ESC(1)	ESC(1), ESC(1), ESC(1), I(1/6)
D	“	ESC(3/7) D(1/7) I(1/7) V(1/7)	D| ESC(1/2) I(1/2) **I|** ESC(1/2) V(1/2) V| ESC(1/2) I(1/2)	DI| ESC(1/2) I(1/2) IV| ESC(1/2) I(1/2) **VI|** ESC(1)	DIV| ESC(1/2) I(1/2) **IVI|** ESC(1)	ESC(1), ESC(1), ESC(1/2), D(1/7)
I	“	ESC(3/8) D(2/8) I(1/8) V(1/8)	**D|** ESC(1/2) I(1/2) I| ESC(1/4) V(1/4) D(1/4) V| ESC(1/2) I(1/2)	DI| ESC(1/2) I(1/2) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) **ID|** ESC(1)	DIV| ESC(1/2) I(1/2) IVI| ESC(1/2) I(1/2) **VID|** ESC(1)	ESC(1), ESC(1), I(1/2)
D	“	ESC(3/8)	D| ESC(1/3)	**DI|**	DIV|	ESC(1),

<table>
<tr><td></td><td></td><td>D(2/8)
I(1/8)
V(1/8)</td><td>I(2/3)
I| ESC(1/4)
V(1/4)
D(1/4)
V| ESC(1/2)
I(1/2)</td><td>ESC(1/2)
I(1/2)
IV|
ESC(1/2)
I(1/2)
VI|
ESC(1/2)
D(1/2)
ID|
ESC(1/2)
I(1/2)</td><td>ESC(1/2)
I(1/2)
IVI|
ESC(1/2)
I(1/2)
VID|
ESC(1/2)
I(1/2)
IDI|
ESC(1)</td><td>ESC(1/2),
D(1/4)</td></tr>
<tr><td>O</td><td>“</td><td>ESC(3/8)
D(2/8)
I(1/8)
V(1/8)</td><td>D| ESC(1/3)
I(2/3)
I| ESC(1/5)
V(1/5)
D(2/5)
V| ESC(1/2)
I(1/2)</td><td>DI|
ESC(2/4)
I(1/4)
D(1/4)
IV|
ESC(1/2)
I(1/2)
VI|
ESC(1/2)
D(1/2)
ID|
ESC(1/2)
I(1/2)</td><td>DIV|
ESC(1/2)
I(1/2)
IVI|
ESC(1/2)
I(1/2)
VID|
ESC(1/2)
I(1/2)
IDI|
ESC(1/2)
D(1/2)
DID|
ESC(1)</td><td>ESC(1),
ESC(1/2),
ESC(1/3),
ESC(3/8),
O(1/257)</td></tr>
<tr><td>S</td><td>“</td><td>ESC(4/10)
D(2/10)
I(1/10)
V(1/10)
O(1/10)</td><td>D| ESC(2/5)
I(2/5)
O(1/5)
I| ESC(1/5)
V(1/5)
D(2/5)
V| ESC(1/2)
I(1/2)
O| ESC(1)</td><td>DI|
ESC(2/4)
I(1/4)
D(1/4)
IV|
ESC(1/2)
I(1/2)
VI|
ESC(1/2)
D(1/2)
ID|
ESC(2/4)
I(1/4)
O(1/4)
DO| ESC(1)</td><td>DIV|
ESC(1/2)
I(1/2)
IVI|
ESC(1/2)
I(1/2)
VID|
ESC(1/2)
I(1/2)
IDI|
ESC(1/2)
D(1/2)
DID|
ESC(1/2)
O(1/2)
IDO|ESC(1)</td><td>ESC(1)
ESC(1)
ESC(1)
ESC(4/10)
O(1/257)</td></tr>
</table>

EOF	“	ESC(6/12) D(2/12) I(1/12) V(1/12) O(1/12)	D| ESC(2/5) I(2/5) O(1/5) I| ESC(1/5) V(1/5) D(2/5) V| ESC(1/2) I(1/2) O| ESC(1/2) S(1/2) **S|** ESC(1)	DI| ESC(2/4) I(1/4) D(1/4) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) ID| ESC(2/4) I(1/4) O(1/4) DO| ESC(1/2) S(1/2) **OS|** ESC(1)	DIV| ESC(1/2) I(1/2) IVI| ESC(1/2) I(1/2) VID| ESC(1/2) I(1/2) IDI| ESC(1/2) D(1/2) DID| ESC(1/2) O(1/2) IDO|ESC(1/2) S(1/2) **DOS**| ESC(1)	ESC(1) ESC(1) ESC(1) ESC(6/12) EOF(1/257)

- **PPMC (completo):**

Ahora sí, vamos a comprimir a contexto 3 con exclusión total el archivo.

Lectura	Contexto –1	Contexto 0	Contexto 1	Contexto 2	Contexto 3	Emisión
D	Todos los caracteres más el EOF	ESC(1)				ESC(1), D(1/257)
I	“	ESC(1/2) D(1/2)	**D|** ESC(1)			ESC(1), ESC(1/2), I(1/257)
V	“	ESC(2/4) D(1/4) I(1/4)	D| ESC(1/2) I(1/2) **I|** ESC(1)	**DI|** ESC(1)		ESC(1), ESC(1), ESC(2/4), V(1/257)
I	“	ESC(3/6) D(1/6) I(1/6) V(1/6)	D| ESC(1/2) I(1/2) I| ESC(1/2) V(1/2) **V|** ESC(1)	DI| ESC(1/2) I(1/2) **IV|** ESC(1)	**DIV|** ESC(1)	ESC(1), ESC(1), ESC(1), I(1/6)
D	“	ESC(3/6) D(1/6) I(1/6) V(1/7)	D| ESC(1/2) I(1/2) **I|** ESC(1/2) V(1/2) V| ESC(1/2) I(1/2)	DI| ESC(1/2) I(1/2) IV| ESC(1/2) I(1/2) **VI|** ESC(1)	DIV| ESC(1/2) I(1/2) **IVI|** ESC(1)	ESC(1), ESC(1), ESC(1/2), D(1/6)
I	“	ESC(3/8) D(2/8)	**D|** ESC(1/2) I(1/2)	DI| ESC(1/2)	DIV| ESC(1/2)	ESC(1), ESC(1), I(1/2)

		I(1/8) V(1/8)	I| ESC(1/4) V(1/4) D(1/4) V| ESC(1/2) I(1/2)	I(1/2) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) **ID|** <u>ESC(1)</u>	I(1/2) IVI| ESC(1/2) I(1/2) **VID|** <u>ESC(1)</u>	
D	"	ESC(3/8) D(2/8) I(1/8) V(1/8)	D| ESC(1/3) I(2/3) **I**| ESC(1/4) V(1/4) <u>D(1/4)</u> V| ESC(1/2) I(1/2)	**DI|** <u>ESC(1/2)</u> I(1/2) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) ID| ESC(1/2) I(1/2)	DIV| ESC(1/2) I(1/2) IVI| ESC(1/2) I(1/2) VID| ESC(1/2) I(1/2) **IDI|** <u>ESC(1)</u>	ESC(1), ESC(1/2), D(1/4)
O	"	<u>ESC(3/7)</u> D(2/7) I(1/8) V(1/7)	**D|** <u>ESC(1)</u> I(2/3) I| ESC(1/5) V(1/5) D(2/5) V| ESC(1/2) I(1/2)	DI| ESC(2/4) I(1/4) D(1/4) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) **ID|** <u>ESC(1/2)</u> I(1/2)	DIV| ESC(1/2) I(1/2) IVI| ESC(1/2) I(1/2) VID| ESC(1/2) I(1/2) IDI| ESC(1/2) D(1/2) **DID|** <u>ESC(1)</u>	ESC(1), ESC(1/2), ESC(1), ESC(3/7), O(1/257)
S	"	<u>ESC(4/10)</u> D(2/10) I(1/10) V(1/10) O(1/10)	D| ESC(2/5) I(2/5) O(1/5) I| ESC(1/5) V(1/5) D(2/5) V| ESC(1/2) I(1/2) **O|** <u>ESC(1)</u>	DI| ESC(2/4) I(1/4) D(1/4) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) ID| ESC(2/4) I(1/4) O(1/4) **DO|** <u>ESC(1)</u>	DIV| ESC(1/2) I(1/2) IVI| ESC(1/2) I(1/2) VID| ESC(1/2) I(1/2) IDI| ESC(1/2) D(1/2) DID| ESC(1/2) O(1/2) **IDO|**<u>ESC(1)</u>	ESC(1) ESC(1) ESC(1) ESC(4/10) O(1/257)
EOF	"	<u>ESC(6/12)</u> D(2/12) I(1/12)	D| ESC(2/5) I(2/5) O(1/5)	DI| ESC(2/4) I(1/4)	DIV| ESC(1/2) I(1/2)	ESC(1) ESC(1) ESC(1)

		V(1/12) O(1/12)	I| ESC(1/5) V(1/5) D(2/5) V| ESC(1/2) I(1/2) O| ESC(1/2) S(1/2) **S|** ESC(1)	D(1/4) IV| ESC(1/2) I(1/2) VI| ESC(1/2) D(1/2) ID| ESC(2/4) I(1/4) O(1/4) DO| ESC(1/2) S(1/2) **OS|** ESC(1)	IVI| ESC(1/2) I(1/2) VID| ESC(1/2) I(1/2) IDI| ESC(1/2) D(1/2) DID| ESC(1/2) O(1/2) IDO|ESC(1/2) S(1/2) **DOS**| ESC(1)	ESC(6/12) EOF(1/257)

Descompresión:

Para descomprimir se comienza por un intervalo del compresor aritmético correspondiente al contexto “-1” con los 257 caracteres dónde obtendremos el primer carácter comprimido (recuerda que en el ejemplo se emitió “D(1/257)”, los ESC(1) no se escriben de disco en compresión aritmética). Luego pasamos a un intervalo de contexto “0” que tiene al carácter que salió más el escape, ambos con probabilidad un medio (1/2), se leerá el carácter “ESC(1/2)” que significa cambiar el intervalo por otro que represente al contexto “-1” pero con valores extremos iguales a los del carácter del contexto anterior (el “0”, en el ejemplo el ESC(1/2)).

-1	0	-1	1 / **I**	0	2 / **IV**	...
1 | 254 / 257 D 253 / 257 | | | | | | | | | 0	254 / 257 | | ESC | | | 507 / 514 | | D | | | 253 / 257	508 / 514 | | | 249 / 132098 I 248 / 132098 | | | | | | | 507 / 514	249 / 13098 | | | | | | ESC | | | | | | 248 / 13098	249 / 13098 | | ESC | | | 497 / 26196 | D 331 / 17464 | I | 248 / 13098	249 / 13098 | | | | | | ESC | | | | | | 497 / 26196	…
Primer Paso	Segundo Paso	Terser Paso	Cuarto Paso	Quinto Paso	Sexto Paso	...
Leo “D”	Leo “ESC”	Leo “I”	Leo “ESC”	Leo “ESC”	Leo la “S”	...

Como se observa, por cada caracter que se identifica (distinto de ESC) se pasa a un nivel superior, dónde comenzará la búsqueda del siguiente; siendo éste nuevo contexto mayor en uno al

primero del caracter anterior. Esto se repite hasta que lleguemos al máximo contexto de la compresión, y luego sólo regresaremos hasta él para retomar la secuencia de descompresión.

Las distintas formas de implementación son por Hash tables con listas para las colisiones, por Tries o por listas múltiples. En general lo que buscan es reducir el consumo masivo de memoria que tiene el compresor para lograr tiempos de compresion y descompresión aceptables

Compresión no estadística

Hasta ahora hemos visto compresores que para un caracter en base a su probabilidad definen su codificación y emiten dichos bits. Sin embargo hay distintas formas de compresión que no se basan estrictamente en todas las probabilidades de los caracteres. A grandes rasgos se suele encontrar 3 grupos en los que se dividen a los compresores no estadísticos:

- **Compresores por run length**: la base de estos compresores está en detectar repeticiones en el archivo (por ejemplo, repeticiones de 1 caracter) y reemplazarlas por otros símbolos que indiquen la repetición y el largo de ella (por ejemplo el texto "aaaaa" se reemplazará de alguna forma por "(a:5)" indicando 5 repeticiones de la letra a). La dificultad de estos métodos consiste en que para el caso de archivos con pocas repeticiones no se tenga una gran perdida de espacio generando archivos demasiado grandes (como ejemplo, el archivo "abc" se reemplazaría por "(a:1)(b:1)(c:1)", que seguramente ocupará mucho mas innecesariamente)
- **Compresores predictores**: estos hacen una predicción (por ejemplo, "el próximo caracter será una A") y en caso de acertarla emitirán un código de longitud muy pequeña (como ser un bit). El descompresor deberá hacer la misma predicción y gracias a ello podrá regenerar el archivo principal. Para los compresores predictores es fundamental que la predicción sea buena o en caso contrario se estarán agregando muchos datos innecesarios para indicar que fue incorrecta
- **Compresores por sustitución**: son los que consisten en sustituir una cadena de varios caracteres por un símbolo de menor longitud en bits. El problema aquí es definir que cadenas se sustituirán, ya que de intentar representar todas las cadenas posibles no se podrá elegir símbolos de menor longitud que las mismas y no se estaría comprimiendo absolutamente nada (si quiero reemplazar todas las cadenas de 2 caracteres, necesito representar 256*256 = 65536 cadenas diferentes, cosa que no puedo hacer con menos de 16 bits)

Generalmente un compresor que pertenezca únicamente a uno de los grupos no suele ser tan bueno, pero en cambio al generar compresores híbridos de varias categorías los resultados mejoran notablemente. De hecho, el PPMC visto anteriormente no se considera un compresor totalmente estadístico sino, como su nombre lo indica, un híbrido entre estadístico y predictor, ya que predice la probabilidad de cada caracter en base a las ocurrencias en el mismo contexto de un orden dado.

A continuación veremos 4 métodos que comparten el inicio del nombre pero sin embargo tienen características particulares bastante importantes y que los distinguen entre si. Estos 4 compresores son bastante simples de implementar, sin embargo son hoy por hoy muy utilizados (con algunas modificaciones) en el ambiente informática. El LZ77 es la base del formato grafico sin perdidas PNG, muy utilizado en imágenes blanco y negro. El LZ78 se encuentra detrás del formato GIF, formato mas usado para imágenes en color sin perdidas (el JPG es un formato de compactación). Finalmente el LZHUFF con variaciones es utilizado por los compresores de archivos ARJ y GZIP, de cuya popularidad ni siquiera es necesario hablar

LZ77

El primer algoritmo que veremos es del tipo "compresor por run length". Recibe su nombre de sus dos creadores, Lempel y Ziv, y se lo conoce también como "sliding windows" por utilizar para la compresión dos ventanas o buffers:

- El primer buffer se denomina **ventana de inspección** y contiene los siguientes N caracteres a comprimir del archivo.

• El segundo buffer es la **ventana de memoria** y contiene los últimos M caracteres comprimidos.

Lo que intentará hacer el compresor es encontrar repeticiones del string que comienza en el primer caracter de la ventana de inspección junto con cualquiera de los strings que comienzan en cualquier posición de la ventana de memoria. Cuando haya al menos una repetición (puede haber varias) diremos que hay un match de longitud K, donde K indica cuantos caracteres son iguales entre ambos strings. Por simplicidad, dado que si en una posición hay un match de longitud K, también lo hay de longitud K-1, K-2, etc, solo hablaremos del match de longitud mas grande para la misma posición. también ignoraremos los casos de match de longitud 1 porque no son repeticiones de substrings sino simplemente el mismo caracter

Veámoslo con ejemplos, suponiendo los contenidos de ambas ventanas de longitud máxima 6 (M=6, N=6):

Ventana de memoria	Ventana de inspección	
PASCUA	CUANDO	Hay un match que se da en la posición 3 de la ventana de memoria (tomando que la primer posición es la 0). El match es de longitud 3 (“CUA”)
SALSAS	SALADO	Hay 2 matchs: el primero en la posición 0 de la ventana de memoria y de longitud 3 (“SAL”), el segundo en la posición 3 de la ventana de memoria y de longitud 2 (“SA”)
SASASA	ESASAS	No hay matchs, ya que no hay substrings en la ventana de memoria que comiencen con la letra E

Luego de todas estas definiciones, podemos comenzar a explicar el algoritmo LZ77 hablando de los 3 parámetros que tiene. Los primeros 2 ya los hemos visto y son los tamaños de ambas ventanas (N y M), su definición estará dada no solo por el gusto del programador sino primero por el factor de la memoria disponible de la maquina (que no se puede superar) y luego por un tema de performance, ya que mientras mas grande sea la ventana de memoria mas substrings se deberán examinar para ver si hay matchs. El tercer y ultimo parámetro es una longitud mínima que implica que en caso de encontrar un match de longitud menor a ella, lo ignoraremos por completo. Mas adelante cuando veamos la codificación del LZ77 veremos el por que de este parámetro.

Para comprimir un archivo:

• Inicialmente se toma la ventana de memoria como vacía (con algún caracter especial para indicar ello por ejemplo) y se carga la ventana de inspección con los primeros N caracteres del archivo

• Mientras sigan quedando caracteres en la ventana de inspección (mientras no se haya llegado al fin del archivo) se compara el string que comienza en la posición 0 de dicha ventana con todos los strings que comienzan en cada posición de la ventana de memoria.

- o Si no hay ningún match de longitud mayor a la mínima, entonces se emite el primer caracter de la ventana de inspección y ambas ventanas se desplazan 1 caracter (la ventana de memoria pierde su primer caracter y el ultimo pasa a ser el caracter recientemente comprimido; la ventana de inspección también pierde su primer caracter y lee del archivo a comprimir el siguiente caracter, que ubica al final)
- o Si hay al menos un match de longitud mayor a la mínima, se toma el match de mayor longitud (puede haber varios matchs) y se emite primero la posición en la que se dio el match en la ventana de memoria y luego la longitud del mismo. Las dos ventanas se desplazan tantos caracteres como sea la longitud de dicho match (la de memoria pierde sus primeros k caracteres y los k últimos pasan a ser el substring

que matcheo, la de inspección también pierde sus primeros k caracteres y lee otros k del archivo que los ubica al final). Nota: en caso de empate entre 2 matchs con la longitud máxima se debe tomar alguna convención, por ejemplo emitir la de menor posición

El descompresor, que veremos en detalle mas adelante, podrá ir armando la misma ventana de memoria en cada paso, por lo que al recibir un par ordenado (posición p, longitud l), podrá formar el match de longitud l. Entonces al emitir un caracter se habrá comprimido 1 solo byte, pero al emitir un par ordenado (posición p, longitud l) se habrá comprimido l bytes en un paso.

Ya teniendo la explicación del algoritmo podemos seguir un ejemplo. Para ello comprimamos el archivo “RAPATAPATAPARAPAPA” con ventana de memoria de 6 caracteres, ventana de inspección de 5 y longitud mínima de match de 2 caracteres:

Inicialmente la ventana de memoria esta vacía y la de inspección se llena con los caracteres “RAPAT” que son los primeros 5 del archivo:

Memoria	inspección
______	RAPAT

No hay match alguno, con lo que se emite una R. Las ventanas se desplazan 1 caracter con lo que la ventana de memoria será de 5 caracteres blancos y una R (la que se acaba de comprimir) y la ventana de inspección comenzará con “APAT” y el ultimo caracter será la “A”, que debe ser leída del archivo

Memoria	inspección
_____R	APATA

En este caso ocurre algo similar al anterior, se emite una A y se desplaza 1 caracter

Memoria	inspección
____RA	PATAP

Se emite una P y se desplaza un caracter

Memoria	inspección
___RAP	ATAPA

Sigue sin haber matchs, se emite una A y se desplaza 1 caracter

Memoria	inspección
__RAPA	TAPAT

Se emite T, se desplaza un caracter

Memoria	inspección
_RAPAT	APATA

Por fin encontramos un match. Este se da en la posición 2 y es de longitud 4 (“APAT). Se emite entonces el par ordenado (posición, longitud), que en este caso es (P2, L4). Recordar que todavía en la memoria nos queda un caracter sin uso (el primero) que se arrastro desde el inicio de la compresión cuando la ventana de memoria estaba en blanco, por eso la posición de la R es 1 y entonces la de la A es 2. Ahora las ventanas se deben desplazar 4 caracteres, con lo que la ventana

de memoria pierde a los primeros 4 (_RAP) y pasa a ser ATAPAT (el AT que ya estaba, el APAT que se comprimió con (P2, L4)). La ventana de inspección comenzara con la A que no se pudo comprimir en este paso y luego leerá 4 caracteres: “PARA”

Memoria	inspección
ATAPAT	APARA

En este caso, hay 1 match de longitud 3 en la posición 2, con lo que se emite (P2,L3) y se desplaza cada ventana 3 posiciones

Memoria	inspección
PATAPA	RAPAP

No hay matchs, se emite una R y se desplaza 1 posición

Memoria	inspección
ATAPAR	APAPA

Se emite (P2, L3). Al desplazar se vera que no hay mas caracteres a leer por lo que cuando se vacíe la ventana de inspección se habrá terminado la compresión del archivo

Memoria	inspección
PARAPA	PA

Hay 2 matchs de longitud 2, uno en la posición 0 y otro en la posición 4. Se elige por convención emitir la de menor posición, con lo que se emite (P0,L2). Al desplazar las ventanas 2 caracteres se ve que la ventana de inspección esta vacía, con lo que se termina el método. El resultado de la compresión fue:

R – A – P – A – T – (P2,L4) – (P2,L3) – R – (P2,L3) – (P0,L2)

En la siguiente tabla se resume la compresión en cada paso mostrándose además en la primer columna la ubicación de cada ventana en el archivo completo (Los caracteres entre paréntesis forman parte de la ventana de memoria, los caracteres entre corchetes forman parte de la ventana de inspección):

Ubicación de las ventanas	Ventana de Memoria	Ventana de inspección	Salida del compresor
()[RAPAT]APATAPARAPAPA	______	RAPAT	R
(R)[APATA]PATAPARAPAPA	_____R	APATA	A
(RA)[PATAP]ATAPARAPAPA	____RA	PATAP	P
(RAP)[ATAPA]TAPARAPAPA	___RAP	ATAPA	A
(RAPA)[TAPAT]APARAPAPA	__RAPA	TAPAT	T
(RAPAT)[APATA]PARAPAPA	_RAPAT	APATA	(P2-L4)
RAP(ATAPAT)[APARA]PAPA	ATAPAT	APARA	(P2-L3)
RAPATA(PATAPA)[RAPAP]A	PATAPA	RAPAP	R
RAPATAP(ATAPAR)[APAPA]	ATAPAR	APAPA	(P2-L3)
RAPATAPATA(PARAPA)[PA]	PARAPA	PA___	(P0-L2)

Para generar el archivo de salida, necesitaremos acordar una forma de representar lo que se emite. En primer lugar hay que distinguir en si se emite un caracter o se emite un par posición-

longitud. Para ello usaremos un bit que por convención será 1 cuando sea un caracter, 0 cuando sea un par (puede tranquilamente tomarse el opuesto). Al emitir un caracter entonces se emite primero un bit 1 y luego los 8 bits del caracter, utilizando 9 bits en total para representarlo. Para un par posición longitud se emite un 0, luego la posición, que dado que hay 5 valores posibles (P6 nunca se emitirá ya que es imposible tener un match de longitud 2 desde esa posición), se emitirá con 3 bits y la longitud como tiene 4 valores posibles (L2, L3, L4 y L5) con 2 bits. En total entonces para un par posición longitud se emite 1+3+2 = 6 bits

En el ejemplo emitimos 6 caracteres (6*9 = 54 bits) y 4 pares ordenados (4*6 = 24 bits) con lo que en total tendremos un archivo comprimido de 78 bits. El archivo inicial tenia 18 caracteres, lo que da un total de 18*8 = 144 bits, casi el doble que el archivo final comprimido.

Volviendo a un tema que se dejo de lado inicialmente, analizaremos el por qué contar con una longitud mínima de match. Supongamos que trabajamos con una ventana de memoria de 4096 bytes y una de inspección del mismo tamaño. Para representar el par longitud posición, necesitamos 1 bit de diferenciación (para reconocer que no es un caracter), luego 12 bits para indicar la posición y 12 bits para indicar la longitud, dando un total de 25 bits. Para un match de longitud 2, emitirlo como par posición longitud será mucho mas costoso que emitir los 2 caracteres por separado, que siguen ocupando 9+9=18 bits. Entonces con estos tamaños, convendrá elegir como longitud mínima de match a 3, ya que ahí si el emitirla en 25 bits de par posición longitud es mas conveniente que emitir 3 caracteres utilizando 27

Nos queda ver como funciona el descompresor. Como dijimos antes, este podrá formar la misma ventana de memoria en cada paso con los últimos caracteres descomprimidos, pero no la ventana de inspección ya que no conoce los caracteres que vendrán. Sin embargo esto no es ningún impedimento para que funcione (si lo fuera, no estaríamos analizando el método)

El descompresor primero leerá 1 bit para distinguir si es un caracter o un par posición longitud. En el primer caso, leerá 8 bits y descomprimirá el caracter. En el segundo leerá 3 bits que guardará como posición p, 2 bits que guardará como longitud l, y luego descomprimirá desde la posición p de su ventana de memoria los l caracteres siguientes. Veámoslo con el ejemplo:

Ventana de memoria	Lectura del archivo comprimido	Emisión al archivo descomprimido
______	R	R
_____R	A	A
____RA	P	P
___RAP	A	A
__RAPA	T	T
_RAPAT	(P2-L4)	APAT
ATAPAT	(P2-L3)	APA
PATAPA	R	R
ATAPAR	(P2-L3)	APA
PARAPA	(P0-L2)	PA

Como puede verse, en la ultima columna se encuentra el archivo inicial. Para cerrar el tema, resta comentar la problemática del LZ77 que consiste en que mientras mas grande sea la ventana de memoria, mejor se comprimirá porque habrá mas chances de encontrar repeticiones, sin embargo esto juega en contra de la velocidad de compresión ya que es bastante costoso comparar con los strings de cada posición en busca del mejor match. Distintas implementaciones buscan solucionar el problema almacenando la ventana en forma de tries, en hash tables, y otras opciones.

LZ78 - LZW

El LZ78 es muy similar al compresor LZW (difieren en algunos factores de su implementación pero la idea es la misma), con lo que de ahora en más analizaremos únicamente al

primero de ellos. Este compresor es del grupo “compresión por sustitución”, ya que va leyendo el archivo buscando en una tabla de strings la entrada que tenga un match mas largo con la posición actual del archivo. Luego emitirá un numero que corresponde con la posición en la tabla de dicha entrada.

La salida del LZ78 es entonces dependiente de la cantidad de entradas de la tabla, cuando tenga entre 257 y 512 entradas emitirá números de 9 bits, luego cuando haya entre 513 y 1024, de 10 bits, etc. Esta tabla inicialmente contiene a los 256 caracteres (para que siempre haya una entrada con que comprimir el caracter si no hay matchs de longitud de al menos 2) y se irá llenando a medida que se vaya comprimiendo el archivo. El descompresor podrá saber como se fue llenando e ir armando la tabla de la misma forma que el compresor, con lo que sabrá cuando tiene 513 entradas y leerá 10 bits y no 9 como antes (lo mismo para cuando se complete otra potencia de 2, deberá leer 1 bit mas)

Veamos el compresor en mas detalle:

- Inicialmente, se forma una tabla con 256 posiciones, cada una correspondiendo a los 256 distintos valores de un byte
- Se lee el primer caracter del archivo a comprimir. Se busca en la tabla si existe una entrada con el mismo y en caso de que ocurra se lee un caracter mas, buscando ahora en la tabla si se encuentra alguna entrada con el string de 2 caracteres. Este proceso se repite (se sigue leyendo un caracter y formando un string mas grande) hasta no encontrar en la tabla una entrada para dicho string
- Se emite el numero de entrada para el string mas largo que se encontró en la tabla, utilizando una cantidad de bits dada por el tamaño de la tabla (se calcula como log2(máxima entrada de la tabla+1) redondeado hacia arriba, teniendo que sumar 1 porque la primer entrada es la numero 0)
- Se agrega a la tabla una entrada formada por el string que no se encontró en ella, para que si se repite en el archivo en un momento posterior se pueda comprimir mejor.
- Dado que en el paso anterior no se comprimió todo lo leído (se salió del ciclo cuando no se encontró una entrada en la tabla) sino que se comprimió todo el string menos el ultimo caracter, el nuevo string pasa a ser únicamente el ultimo de los caracteres de la cadena anterior. Se repite el paso dos (leer mas caracteres hasta no encontrar entradas en la tabla) hasta que se llegue al fin del archivo, momento en que se emite la entrada de la tabla actual y se finaliza la compresión

Para ver en mas detalle el funcionamiento, se comprimirá a modo de ejemplo el archivo “TERRYTERRELLTERRENCE” , formado por nombres de receptores en la NFL (como por ejemplo Terrell Owens)

- Inicialmente la tabla tiene los 256 caracteres (entradas 0 a 255). Se lee una T, que se encuentra en la tabla con lo que se lee el siguiente caracter que es la E. “TE” no esta en la tabla por lo que se emite el mayor string encontrado (“T”) con 8 bits (en general como solo en este caso se va a dar que se emitan 8 bits, suele adaptarse para mayor simplicidad que el primer símbolo ya se emita con 9 bits)
- Se agrega a la tabla una nueva entrada (la 256) formada por el string “TE”. El nuevo string pasa a ser “E”
- “E” se encuentra en la tabla con lo que se lee, otro caracter. “ER” no se encuentra en la tabla con lo que se emite la entrada correspondiente a “E” con 9 bits (la tabla ya tiene 257 entradas), luego se agrega “ER” a la tabla en la entrada 257 y el nuevo string pasa a ser “R”
- El proceso no sufre cambios significativos para los siguientes 3 caracteres. Se emitirá “R” agregando “RR” en la entrada 258, luego se emite “R” y se agrega “RY” en la entrada 259 y

finalmente se emite “Y” y se agrega “YT” en la entrada 260. Luego de esta emisión, el nuevo string es “T”

- Como “T” se encuentra en la tabla, se lee un caracter que es la E. El nuevo string “TE” también se encuentra en la tabla (es la entrada 256) con lo que se lee otro caracter mas, formándose el string “TER”. Este no se encuentra en la tabla con lo que se emite 256 en 9 bits y se agrega “TER” en la entrada 261 de la tabla. El nuevo string pasa a ser el ultimo caracter del string anterior, o sea “R”. En este paso el compresor por primera vez tuvo una ventaja ya que represento 2 caracteres (el string TE) con 9 bits en vez de los 16 que ocupaban en el archivo original
- “R” se encuentra en la tabla con lo que se lee otro caracter. “RR” también se encuentra en la tabla (entrada 258) con lo que se lee otro caracter. “RRE” no se encuentra en la tabla, con lo que se emite 258 en 9 bits y se agrega “RRE” en la entrada 262 de la tabla. El nuevo string pasa a ser “E”

A continuación se muestra en una tabla, con menos detalle, la compresión total del archivo indicando para cada paso en que lugar del archivo se encuentra comprimiendo, que entrada emite y que entrada se agrega a la tabla:

Ubicación del compresor	Código emitido	Nuevas entradas
TERRYTERRELLTERRENCE	T	256 - TE
ERRYTERRELLTERRENCE	E	257 - ER
RRYTERRELLTERRENCE	R	258 - RR
RYTERRELLTERRENCE	R	259 - RY
YTERRELLTERRENCE	Y	260 - YT
TERRELLTERRENCE	256	261 - TER
RRELLTERRENCE	258	262 - RRE
ELLTERRENCE	E	263 - EL
LLTERRENCE	L	264 - LL
LTERRENCE	L	265 - LT
TERRENCE	261	266 - TERR
RENCE	R	267 - RE
ENCE	E	268 - EN
NCE	N	269 - NC
CE	C	270 - CE
E	E	--

La compresión del archivo dio como resultado 1 código de 8 bits y luego 15 códigos de 9 bits, totalizando 143 bits. El archivo inicial contaba con 20 caracteres de 8 bits, o sea 160 bits. Si bien la ganancia es muy chica para apreciarla, hay que ver que se agregaron muy pocas entradas a la tabla (solo 15) y en general para archivos mas largos se suelen dar mejores compresiones. Igualmente es de notar que el LZ77 suele tener mejores resultados que el LZ78.

La descompresión del archivo es mas simple aun. Lo único que hace es leer un numero de una cantidad de bits dada por el tamaño de la tabla (utilizando la misma formula que al comprimir) y emitir el string correspondiente a la entrada del numero leído. La única gran diferencia con el compresor es que este generaba la tabla siempre con el string comprimido mas un caracter, pero el descompresor lo desconoce a este ultimo (todavía no lo descomprimió!) y no puede formar la misma entrada hasta el paso siguiente. Por ello en todo paso salvo el primero, se agrega a la tabla el string descomprimido en el paso anterior mas el primer caracter descomprimido en el paso actual

Veamos como será la descompresión del mismo archivo, mostrando el símbolo leído, el string descomprimido y la entrada que se agrega a la tabla:

Código leído	String descomprimido	Nuevas entradas
T	T	--
E	E	256 - TE
R	R	257 - ER
R	R	258 - RR
Y	Y	259 - RY
256	TE	260 - YT
258	RR	261 - TER
E	E	262 - RRE
L	L	263 - EL
L	L	264 - LL
261	TER	265 - LT
R	R	266 - TERR
E	E	267 - RE
N	N	268 - EN
C	C	269 - NC
E	E	270 – CE

Como se ve en la segunda columna queda el archivo original

Caso particular

El hecho de que la entrada de la tabla se forme en un paso posterior, da lugar a lo que se conoce como “caso particular”, que es cuando el descompresor debe emitir el string asociado a una entrada en la tabla que todavía no pudo formar (por necesitar del primer caracter del siguiente código). El caso particular ocurre si al comprimir un archivo se emite una entrada que se agrego a la tabla en el paso anterior. Por ejemplo, si se comprime el archivo “ABABABA” :

Ubicación del compresor	Código emitido	Nuevas entradas
ABABABA	A	256 - AB
BABABA	B	257 - BA
ABABA	256	258 - ABA
ABA	258	--

En el cuarto paso se emite la entrada 258, que justamente es la entrada que se genero en el paso anterior, con lo que estamos ante un caso particular. Veamos como lo maneja el descompresor:

Código leído	String descomprimido	Nuevas entradas
A	A	--
B	B	256 – AB
256	AB	257 – BA
258	????	

Como se ve, en el cuarto paso el descompresor ve que tiene que emitir el string asociado a la entrada 258 de la tabla, pero todavía no cuenta con la misma por lo que a primera vista no tiene forma de saber que emitir. Sin embargo, esta problema solo se da cuando hay un caso particular, con lo que analizando un poco se verá que es posible generar la entrada faltante

Primero que nada analicemos todo desde el punto de vista del descompresor, suponiendo que no sabemos el contenido de la entrada 258. Lo que si podemos decir es que el compresor en el tercer paso encontró el string AB en la tabla y luego leyó un caracter que todavía desconocemos. Representando al mismo con el símbolo “*”, también sabemos que el compresor busco “AB*” en la tabla y no lo encontró (en caso de haberlo encontrado, habría emitido otra entrada y no la 256 correspondiente a AB). Al no encontrarlo, emitió 256 y agrego la entrada 258, que es “AB*”

En el siguiente paso, el string inicial fue “*”. Luego de encontrar strings en la tabla y leer los siguientes caracteres del archivo, en un momento se emitió la entrada 258. Esto quiere decir que el string formado por el * y otros caracteres desconocidos es el asociado a la entrada 258. Por ende, la entrada 258 comienza con el caracter * desconocido

Inicialmente supimos que la entrada 258 la formamos como “AB*”. Luego supimos que el primer caracter de la entrada es el caracter desconocido *. Pero, como ya sabemos que el primer caracter de la entrada es una A (lo que nos faltaba saber es el ultimo caracter), podemos decir que *=A. Entonces, “AB*” = “ABA” y finalmente formamos la entrada 258 completa. Con esta información el descompresor podrá terminar de emitir el archivo inicial

La regla de oro para el caso particular es entonces: cuando al descomprimir se encuentra una entrada que todavía no existe en la tabla, dicha entrada se puede formar como la concatenación de lo descomprimido en el paso anterior y el primer caracter de lo descomprimido en el paso anterior

Clearing

Un tema importante a tener en cuenta al implementar un LZ78 es el crecimiento de la tabla. Si bien nos interesa que esta vaya agregando entradas porque mejorarán el nivel de compresión, llega un momento en el que se cuenta con una cantidad tan grande que la emisión es de muchos bits (por ejemplo, 14) y el encontrar un match de longitud pequeña no implica un ahorro tan importante como la perdida que se tiene al no encontrarlo (con 14 bits, emitir un string de 2 caracteres ahorra 2 bits, pero emitir un caracter es una perdida de 6)

Es necesario entonces definir un limite máximo de crecimiento de la tabla que al ser alcanzado implique una reestructuración de la misma, lo que se conoce como clearing. En esta reestructuración se quitarán algunas o todas las entradas de la tabla para volver a una cantidad baja de bits por emisión. La decisión de que entradas quitar es una decisión de diseño, puede llevarse un contador de frecuencia para cada entrada y quitar aquellas que se hayan utilizado pocas veces, pueden quitarse las entradas asociadas con strings muy largos o hasta pueden quitarse todas y volver a la tabla inicial con 256 entradas

Lo importante al elegir un método de clearing es que el descompresor efectúe el mismo proceso en el mismo momento y de la misma forma, o de lo contrario el archivo descomprimido no guardará ninguna similitud con el inicial desde dicho punto. Muchos compresores deciden utilizar un caracter extra para indicar al descompresor que debe efectuar el clearing (por ejemplo, la entrada 256 de la tabla) con lo que pueden efectuar cualquier análisis para elegir el momento de reestructuración sin que el descompresor necesite saberlo. Tranquilamente en este caso 2 archivos comprimidos de distinto tamaño pueden descomprimirse en un mismo archivo único si se utilizaron con 2 compresores distintos que no elegían el momento de clearing de acuerdo a los mismos factores

Implementación eficiente de la tabla

Se puede ver rápidamente que si se utiliza únicamente una tabla para las entradas, la compresión y descompresión tardaran mucho ya que para encontrar el string buscado se debe recorrer una a una cada entrada de la misma. En archivos de gran tamaño, cosa que implica además una tabla de muchas entradas, este proceso será muy costoso y es necesario buscar una alternativa

La opción mas utilizada es modificar la entrada para que en vez de contener el string completo contenga dos campos, el primero es un entero que indica la entrada de la tabla que

contiene al string que es 1 caracter mas corto, y el segundo es el ultimo caracter del string de la entrada. Por ejemplo, veremos la tabla del ultimo paso de la compresión del archivo de ejemplo:

Tabla sin modificar	
Nro	String
256	TE
257	ER
258	RR
259	RY
260	YT
261	TER
262	RRE
263	EL
264	LL
265	LT
266	TERR
267	RE
268	EN
269	NC
270	CE

Tabla modificada		
Nro	Ent. Ant.	Caracter
256	T	E
257	E	R
258	R	R
259	R	Y
260	Y	T
261	256	R
262	258	E
263	E	L
264	L	L
265	L	T
266	261	R
267	R	E
268	E	N
269	N	C
270	C	E

Las entradas de la tabla modificada no se almacenan en un vector o una lista, sino que se utiliza la estructura conocida como Hash List, estructura que vendría a ser similar a un archivo directo pero en memoria. En esta estructura se cuenta con N listas y una función de hash f(x). Un elemento e solamente puede estar en la lista i si hash(e) mod N = i. Este tipo de estructura es muy rápido para búsquedas sin necesitar mucha información administrativa

Supongamos que lo que sigue en el archivo es "TERRA". Con la primer tabla, se debería buscar linealmente primero una ocurrencia de TE (es obvio que "T" estará en la tabla con lo que en la practica ni siquiera se busca), luego TER, luego de TERR y finalmente TERRA; que no se encontrara. Todas estas búsquedas son lineales, leyendo entrada a entrada, y aumentando su tiempo de resolución en un gran factor si la tabla tuviera muchas entradas, ya que no se sabe si existe TERRA hasta llegar al final.

En cambio con la tabla modificada el proceso es el siguiente:

- Se busca en la hash list una entrada que tenga Ent. Ant = T, Caracter = E. Esta búsqueda equivale a buscar "TE" en la tabla, pero solo se buscará en una de las listas. La búsqueda devolverá que la entrada existe y es la 256
- Se lee el siguiente caracter (R) y ahora se busca Ent. Ant = 256, Caracter =R. Esta búsqueda devuelve que la entrada existe y es la 261
- Se lee el siguiente caracter (R) y ahora se busca Ent. Ant = 261, Caracter =R. Esta búsqueda devuelve que la entrada existe y es la 266
- Se lee el siguiente caracter (A) y ahora se busca Ent. Ant = 266, Caracter =A. Esta búsqueda devuelve que la entrada no existe, por lo que se emite 266 y se agrega "TERRA" a la tabla de la forma Nro = 271, Ent Ant = 266, Caracter = A.

En este ejemplo se busco en 4 listas distintas de la hash list (1 lista recorrida en cada paso). Si se tuvieran 16 listas distintas (N=16) y suponiendo una buena distribución de las entradas en ellas, solamente se habrá recorrido en promedio 1/4 de las entradas, en vez que todas como se hacía con un simple vector de strings

LZHUFF

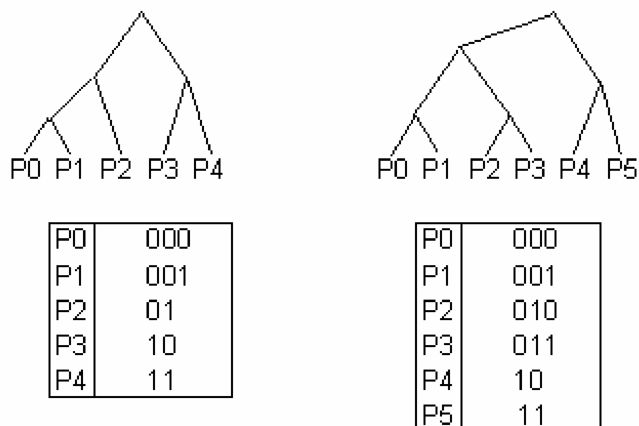
En el LZ77 utilizábamos 9 bits para emitir un caracter y otra cantidad (6 en el ejemplo) para representar un par posición-longitud. Esto no es muy bueno ya que para un determinado tipo de archivos podría haber muchas repeticiones de longitud 4 y ninguna de otra longitud, con lo que seria deseable que se utilicen menos bits para representar a la primera. Lo mismo ocurre con los caracteres, puede darse el caso de que algunos en particular sean mucho mas probables que otros y no hay justificación para utilizar 9 bits siempre. La solución a estos problemas es adaptar el LZ77 con un modelo estadístico, convirtiéndose en LZHUFF por utilizar los árboles de Huffman

En este compresor se cuenta con 2 árboles, el primero contiene no solo a los 256 caracteres posibles sino también a todas las longitudes que se pueden dar; el segundo contiene a las distintas posiciones. El primer árbol se manejará de la misma forma que un árbol de huffman dinámico, inicializando la frecuencia de cada caracter y cada longitud en 1 cuando se comprima o descomprima, en cambio el segundo es un árbol fijo que no cambiará durante la compresión (esto simplemente se toma así por convención).

El método de compresión será el mismo que en el LZ77, pero lo que cambiará es la forma de emitir los resultados. Ahora cuando se comprima un caracter se emitirán los bits correspondientes al mismo según el árbol de caracteres y longitudes, luego se aumentará la frecuencia del caracter y se reconstruirá el árbol. En cambio cuando se quiera comprimir un par posición-longitud primero se emitirá el código de la longitud utilizando el mismo árbol de huffman y a continuación el código en bits correspondiente con la posición. Finalmente se aumentará únicamente la frecuencia de la longitud emitida, reconstruyendo nuevamente el árbol

El descompresor ira leyendo bit a bit y deberá saber entonces que cuando descomprime un código de longitud, los siguientes bits debe interpretarlos utilizando el árbol de posiciones para así formar el par completo y poder regenerar el archivo inicial. Cuando descomprima un código de caracter, simplemente emitirá el mismo y seguirá utilizando el árbol de caracteres y longitudes

La explicación de por que en un árbol no se juntan las posiciones con los caracteres es que es mas probable que para un archivo dado haya una repetición de un conjunto de longitudes; por ejemplo en archivos de una factura telefónica habrá muchos matchs de la longitud de un numero de teléfono cuando se hayan efectuado varias llamadas a un mismo lugar. En cambio para las posiciones lo que se observa es que en la gran mayoría de los casos las posiciones más altas tienen mayor probabilidad, ya que las repeticiones se dan en lo mas recientemente comprimido. Entonces se arma un árbol fijo en el que se dan códigos más cortos para las posiciones más altas y códigos más largos para las bajas. Mostraremos dos árboles posibles para 5 posiciones (como en el ejemplo de LZ77) y para 6:



P0	000
P1	001
P2	01
P3	10
P4	11

P0	000
P1	001
P2	010
P3	011
P4	10
P5	11

LZP

El LZP es un compresor del tipo predictor (de ahí la P de su nombre) pero que a su vez tiene partes estadísticas y de run length. Volviendo unas paginas hacia arriba, al examinar el LZ77 veíamos que uno de sus problemas era el tiempo que se perdía en buscar el mejor match dentro de la ventana de memoria. Luego cuando vimos el LZHUFF notamos que se obtienen mejoras en el nivel de compresión si se agrega una visión estadística al método

En si el LZP toma un poco de todo lo visto y en su conjunto genera un compresor que aunque no es muy conocido tiene unos resultados muy buenos para lo simple que es programarlo. Su creador, Charles Bloom (foto), es uno de los personajes mas activos en los últimos tiempos del mundo de la compresión de datos.

La idea del LZP surge del análisis del LZ77. Uno de los problemas es que la gran cantidad de bits que se necesitan para representar la posición del match, cantidad que aumenta mientras más grande sea la ventana de memoria, con lo que la idea fue la mas simple: no emitir la posición

Esto que a primera vista parece una locura, es la base del éxito del LZP. En vez de gastar costosos bits en emitir la posición y la longitud, lo que hace el LZP es predice una posición donde cree que estará el mejor match. En cualquier caso posible, haya o no un match, emitirá la longitud del mismo (se admite longitud 0 entonces) y luego emitirá el primer caracter que no matchee. Este proceso se repetirá hasta el fin del archivo a comprimir

Queda entonces el problema de que posición predecir. Para definir esto, la idea de Bloom fue utilizar el contexto del caracter a comprimir, prediciendo que habrá un match la ultima vez que se haya registrado el mismo contexto de N caracteres. Entonces el compresor deberá tener registrado para cada posible contexto de longitud N en qué posición se vio por ultima vez, y cuando haya que predecir la posición simplemente tomará dicho valor

Luego de varias pruebas con distintos valores de N, Bloom llegó a la conclusión de que el valor optimo es N = 4, sin embargo esto trae un pequeño inconveniente ya que para guardar todos las posiciones donde se dio cada contexto se necesitan 2^32 enteros, con lo que se utilizan funciones de hash para disminuir el tamaño de la tabla. En este apunte para no complicar la comprensión del método tomaremos contexto de 2 caracteres, con lo que tranquilamente pueden representarse en una tabla de 65536 entradas.

Veamos ahora si el método en detalle

- Inicialmente se setea un valor nulo para cada contexto de 2 caracteres en un vector de posiciones. Cualquier búsqueda de match contra ese valor emitirá longitud 0
- Mientras no se termine el archivo
  - Obtener el contexto c de 2 caracteres de la posición actual. En base a este, obtener del vector de posiciones la posición p en que se predice habrá un buen match
  - Comparar el string que comienza en p con el string que comienza en el siguiente caracter a comprimir. Emitir la longitud de matcheo (en cuantos caracteres son iguales ambos strings)
  - Emitir el primer caracter que no matchea
  - Actualizar el vector de posiciones, poniendo en la entrada del contexto c a la posición en la que se comenzó el loop

Como se puede ver, el método alterna 2 tipos de emisiones, primero un numero que representa una longitud y luego un caracter. Para emitir las longitudes se utiliza un aritmético dinámico con las longitudes de 0 a 255. Si la longitud l a emitir es mayor o igual a 255 lo que se hace es emitir 255 y luego emitir l -255 de la misma forma. Ejemplos de emisión:

- 260 se emite como 255 y luego 5
- 1000 se emite como 255, 255, 255 y 235
- 255 se emite como 255 y luego 0

Para los caracteres se utiliza un modelo aritmético de orden 1, es decir se cuenta 255 tablas de frecuencia para cada contexto de 1 caracter y la elección de cual utilizar esta dada por el caracter anterior al que se debe emitir. Para representar estas emisiones utilizaremos LX cuando se emita una longitud X y (A,B) para decir que se emitió una B con contexto A. En este ultimo caso debe quedar bien claro que solo se está comprimiendo la B, pero se especifica además utilizar el contexto “A” para utilizar un modelo más pertinente

Queda analizar el caso de los primeros caracteres a comprimir, ya que no tienen ningún contexto. Para seguir con el mecanismo se emitirá también longitud 0, pero debe quedar en claro que hasta no formar un contexto de 2 caracteres tranquilamente podría no emitirse longitud alguna, siempre que el descompresor este al tanto que primero debe descomprimir caracteres y recién cuando tiene un contexto de orden 2 debe descomprimir longitudes y luego caracteres. Además, el primer caracter no tiene un contexto de orden 1 con lo que se fija que lo antecede un espacio (representado como \b). Otra cosa que no se puede efectuar en estos casos particulares es actualizar el vector de posiciones

Ahora si veamos la compresión del archivo “ASASARASSASARAS”. A modo de ayuda, mostraremos en la siguiente tabla las posiciones de cada caracter:

A	S	A	S	A	R	A	S	S	A	S	A	R	A	S
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Para el primer caracter no hay contexto con lo que se emite longitud 0. El primer caracter que no matchea entonces es el caracter de la posición 0 (A), que no tiene contexto de orden 1 con lo que se toma que es \b. Se emite entonces (\b, A)
- Para el segundo caracter, tampoco se cuenta con contexto con lo que se vuelve a emitir longitud 0. Luego se emite la S con su contexto de orden 1, es decir (A, S)
- Para el tercer caracter se tiene el contexto “AS”. Al no haber posición con la que comparar, se emite L0. El caracter que no matchea entonces es el de la posición 2 , que se emite con su contexto: (S,A). Luego se actualiza el vector de posiciones diciendo que el contexto “AS” se dio en la posición 2
- En el cuarto caracter ocurre lo mismo que con el tercero; se emite L0, luego (A,S) y se actualiza el vector de posiciones diciendo que el contexto “SA” se dio en la posición 3
- Al comprimir el quinto caracter se ve que su contexto (AS) se dio anteriormente en la posición 2. Se comparan los strings que comienzan en la posición 4 y en la 2 y se ve que hay un match de longitud 1, por lo que se emite L1. Luego se emite el primer caracter que no matchea con su contexto de orden 1, es decir (A,R). En este paso entonces se comprimieron dos caracteres, primero la A de la posición 4 que se comprimió con L1, luego la R que se comprimió con su contexto
- Nos encontramos actualmente comprimiendo la A de la posición 6. El contexto “AR” no tiene posición asociada con lo que emitimos L0, luego (R,A) y actualizamos el vector de contextos con AR=6
- En el siguiente paso ocurre lo mismo, emito L0, luego (A,S) y actualizo el vector con RA=7
- Al comprimir el caracter S de la posición 8 se ve que su contexto AS se dio en la posición 4 por ultima vez. Se comparan los strings que comienzan en la posición 8 con la posición 4 y se ve que no hay match, con lo que se emite L0, luego (S,S) y se actualiza el vector de contextos AS=8. En este caso, la predicción fallo ya que nos dio una posición en que no había match
- Para el caracter A de la posición 9, su contexto es “SS”, que no tiene posición asociada en el vector, con lo que se emite L0, luego (SA) y se actualiza el vector con SS=9
- Finalmente para la S de la posición 10, su contexto “SA” se dio por ultima vez en la posición 3. Se comparan los strings que comienzan en la posición 10 y en la posición 3 y se ve que

hay un match de longitud 5, con lo que se emite L5. Dado que se llego al final del archivo, no hay mas que emitir

A continuación se verá en una tabla el seguimiento del método resumido:

posición	Contexto	Comparar con pos	Salida	actualización
0	--	--	Long 0 - (\b,A)	--
1	--	--	Long 0 - (A,S)	--
2	AS	--	Long 0 - (S,A)	AS=2
3	SA	--	Long 0 - (A,S)	SA=3
4	AS	2	Long 1 - (A,R)	AS=4
6	AR	--	Long 0 - (R,A)	AR=6
7	RA	--	Long 0 - (A,S)	RA=7
8	AS	4	Long 0 - (S,S)	AS=8
9	SS	--	Long 0 - (S,A)	SS=9
10	SA	3	Long 5	SA=10

Analizando los resultados de el archivo de prueba, vemos que hay un gran numero de longitudes 0, con lo que estas se representarán con pocos bits a medida que avance la compresión. también vemos que el utilizar modelos de orden 1 para los caracteres fue una decisión acertada, como ejemplo se ve que cuando el contexto es “A” la S tiene una gran probabilidad, con lo que se emitirá también utilizando poco espacio

El descompresor sabe que primero debe descomprimir una longitud y luego un caracter. Cuando lea una longitud distinta de 0, deberá tomar el contexto de orden 2 (que son los dos últimos caracteres que descomprimió) y buscar en su vector de posiciones donde se registro por ultima vez dicho contexto. Desde esa posición, deberá copiar al archivo descomprimido tantos caracteres como la longitud le diga. Si lee una longitud 0, directamente no hace nada

Luego de leer la longitud, deberá tomar el ultimo caracter descomprimido como contexto, y utilizando el modelo correspondiente, descomprimir 1 caracter, que añadirá al archivo de salida

Finalmente, para guardar coherencia, deberá actualizar su vector de posiciones para el contexto de orden 2 inicial. A continuación se muestra en una tabla la descompresión del archivo de ejemplo, alternando descompresión de longitudes y de caracteres. En el caso de longitudes se tomara como contexto los últimos 2 caracteres descomprimidos para obtener la posición, en el caso de caracteres se asume que el compresor sabe que fue lo ultimo que descomprimió y por ello puede formar su contexto de orden 1

Lectura	Contexto	Copiar desde posición:	Salida	actualización
L0	--	--		
(\b, A)			A	--
L0	--	--		
(A, S)			S	--
L0	AS	--		
(S, A)			A	AS = 2
L0	SA	--		
(A, S)			S	SA = 3
L1	AS	2	A	
(A,R)			R	AS = 4

L0	AR	--		
(R,A)			A	AR = 6
L0	RA	--		
(A,S)			S	RA = 7
L0	AS	4		
(S,S)			S	AS = 8
L0	SS	--		
(S,A)			A	SS = 9
L5	SA	3	SARAS	

Como era de esperarse, en la columna Salida se ha formado todo el archivo original

Un caso interesante que puede darse con el LZP (siempre que se tome en cuenta en la implementación) es que al verificar la longitud de match se pase más allá de lo comprimido actualmente. Por ejemplo, sea el archivo “AAAAAAAAAA” (10 A seguidas), la compresión es:

posición	Contexto	Comparar con pos	Salida	actualización
0	--	--	Long 0 - (\b,A)	--
1	--	--	Long 0 - (A,A)	--
2	AA	--	Long 0 - (S,A)	AA=2
3	SA	2	Long 7	AA=3

En el ultimo paso de la compresión se comparó la posición 3 con la posición 2 y se vio que matchean en 7 caracteres. Al hacer la comparación, el string que comienza en la posición 2 no termino allí como hubiera hecho en el LZ77 sino que siguió hasta casi el fin del archivo

Para que este archivo pueda descomprimirse, se debe implementar una copia caracter a caracter cuando la longitud no sea 0. Supongamos que estamos descomprimiendo un archivo y nos dicen que desde la posición X hay un match de longitud 7 con la posición Y. Si tomamos al archivo como un vector donde F [i] es el caracter i del mismo, la descompresión de lo anterior implicaría que F[X] = F[Y], F [X+1] = F[Y+1], F [X+2] = F[Y+2], ..., F[X+6] = F[Y+6]. Con esta misma filosofía veremos que se puede descomprimir el archivo de 10 A. Supongamos que ya estamos en el ultimo paso, habiendo descomprimido 3 A y teniendo en el vector de posiciones que el contexto AA se dio en la posición 2 por ultima vez. Luego nos viene una longitud 7, con lo que sabremos que para el archivo descomprimido F[3] = F[2] = A. Luego F[4] = F[3], y como ya sabemos que F[3] = A, entonces F[4] = A también. Finalmente llegaremos a F[9] = F[8] = A con lo que habremos generado nuestro archivo de 10 A consecutivas

Localidad en archivos

Localidad

Supongamos que tenemos 2 archivos:
Archivo 1: AAEAAABBBBEEBCCCCCDEEDDDDFFFFF
Archivo 2: EFACFDBAEDFEBBCDCDADBCFACEABEF

Ambos archivos tienen la misma probabilidad para cada caracter (hay 5 A, 5 B, etc), sin embargo, deberían comprimirse de la misma forma? Si se utilizara un método estadístico sin contextos como el Huffman, el archivo final seria del mismo tamaño (aunque obviamente no el mismo) ya que cada caracter tiene la misma probabilidad en cada archivo, sin embargo en el primero se ve que al comienzo es muy probable encontrarse con una A, en la mitad es muy probable que haya una B, una C o una E y al final una D o una F. Esto se conoce como que el archivo tiene una localidad, ya que ciertos caracteres son mas probables en una zona del mismo. Y generalmente esto se da con mucha frecuencia en distintos tipos de archivo, por lo que es de pensar que sería bueno contar con algún método para aprovecharla

Move to Front

El move to front no es en si un compresor, ya que nunca va a comprimir ni expandir un archivo. Sin embargo, el archivo de salida será distinto al primero ya que tendrá una distribución de probabilidades mucho mejor, gracias a la cual la entropía será menor y utilizando métodos estadísticos se obtendrán archivos mas chicos

El Move to Front funciona muy bien para archivos con localidad ya que codifica cada caracter con un numero que representa hace cuantos caracteres distintos no aparecía el caracter a comprimir. Para ello comienza con una lista de caracteres inicialmente ordenados por número. Cuando viene el caracter X, lo busca en esa lista y emite la posición del caracter en la misma, luego modifica la lista llevando el caracter al inicio. En caso de que vengan muchos caracteres X, ahora emitirá siempre un 0 (ya que el caracter X está al principio de la misma). Si esto se repite con otro caracter Y igualmente se seguirá emitiendo un 0 (salvo la primer ocurrencia) con lo que la probabilidad del caracter 0 en el archivo de salida será mucho mayor

Utilizaremos como ejemplo al archivo 1 de la sección de Localidad, y para simplificar en el vector solo ingresaremos los caracteres de la A a la F en vez que los 255:

Entrada	Vector a examinar	Salida	Vector actualizado
A	ABCDEF	0	No cambia
A	ABCDEF	0	No cambia
E	ABCDEF	4	EABCDF
A	EABCDF	1	AEBCDF
A	AEBCDF	0	No cambia
A	AEBCDF	0	No cambia
B	AEBCDF	2	BAECDF
B	BAECDF	0	No cambia
B	BAECDF	0	No cambia
B	BAECDF	0	No cambia
E	BAECDF	2	EBACDF
E	EBACDF	0	No cambia

B	EBACDF	1	BEACDF
C	BEACDF	3	CBEADF
C	CBEADF	0	No cambia
C	CBEADF	0	No cambia
C	CBEADF	0	No cambia
C	CBEADF	0	No cambia
D	CBEADF	4	DCBEAF
E	DCBEAF	3	EDCBAF
E	EDCBAF	0	No cambia
D	EDCBAF	1	DECBAF
D	DECBAF	0	No cambia
D	DECBAF	0	No cambia
D	DECBAF	0	No cambia
F	DECBAF	5	FDECBA
F	FDECBA	0	No cambia
F	FDECBA	0	No cambia
F	FDECBA	0	No cambia
F	FDECBA	0	No cambia

La salida entonces es:

004100200020130000430100050000

Antes de la transformación los símbolos tenían todos una probabilidad de 1/5, con lo que la entropía era de $H(f) = -5 * 1/5 * \log(1/5) = 2.32$ bits por símbolo

Ahora tenemos al 0 con probabilidad 20/30, al 1 con 3/30, al 2, 3 y 4 con probabilidad 2/30 y finalmente al 5 con probabilidad 1/30. La entropía de el nuevo archivo es:

$H(f) = - [2/3 * \log(2/3) + 1/10 * \log(1/10)+ 3 * 1/15 * \log(1/15) + 1/30 * \log(1/30)]$

$H(f) = 1.67$ bits por símbolo

Como se ve la entropía es mucho menor y por ende el archivo luego de ser procesado por move to front será comprimido mejor por algún método estadístico. Cabe aclarar sin embargo que no siempre el move to front mejorará la entropía, sino que únicamente para los casos con localidad (si no, con simplemente hacer el move to front de la salida del move to front la cantidad de veces que uno quisiera podría comprimir cualquier archivo a cualquier tamaño)

La descompresión es bastante simple, lo que se hace es armar primero el mismo vector inicial (ABCDEF en el ejemplo) y luego ir leyendo los caracteres del archivo comprimido, emitiendo como salida el caracter que esté en la posición leída. Por ejemplo, como lo primero que viene en el archivo es un 0 se emite una A. Luego de cada emisión, se pasa al inicio del vector el caracter descomprimido para ser coherente con la compresión. Descomprimamos el archivo anteriormente generado:

Entrada	Vector a examinar	Salida	Vector actualizado
0	ABCDEF	A	No cambia
0	ABCDEF	A	No cambia
4	ABCDEF	E	EABCDF
1	EABCDF	A	AEBCDF
0	AEBCDF	A	No cambia
0	AEBCDF	A	No cambia
2	AEBCDF	B	BAECDF
0	BAECDF	B	No cambia
0	BAECDF	B	No cambia
0	BAECDF	B	No cambia

2	BAECDF	E	EBACDF
0	EBACDF	E	No cambia
1	EBACDF	B	BEACDF
3	BEACDF	C	CBEADF
0	CBEADF	C	No cambia
0	CBEADF	C	No cambia
0	CBEADF	C	No cambia
0	CBEADF	C	No cambia
4	CBEADF	D	DCBEAF
3	DCBEAF	E	EDCBAF
0	EDCBAF	E	No cambia
1	EDCBAF	D	DECBAF
0	DECBAF	D	No cambia
0	DECBAF	D	No cambia
0	DECBAF	D	No cambia
5	DECBAF	F	FDECBA
0	FDECBA	F	No cambia
0	FDECBA	F	No cambia
0	FDECBA	F	No cambia
0	FDECBA	F	No cambia

Una aclaración importante para la implementación es que pese a que se habla de vector de caracteres o lista de caracteres, en general es importante encontrar una estructura que permita efectuar el pase de cualquier caracter al inicio de forma rápida. Un vector estático generara tiempos grandes de compresión por la reestructuración del mismo en memoria, en cambio si se utiliza una lista doblemente enlazada se ahorrara mucho tiempo

Block Sorting

Viendo las grandes ventajas que tiene el Move to Front, seria agradable poder hacer que funcione para una mayor cantidad de archivos y no solo para los que tienen una gran localidad. Contamos con una técnica, llamada Block Sorting, creada por Burrows y Wheeler en 1994, que permite para un archivo de entrada generar uno de salida con grandes probabilidades de tener más localidad que el inicial. Esta técnica no solo no comprime sino que siempre expande el archivo en el tamaño de un entero, pero esta expansión será contrarrestada con una mejor salida de un Move to front aplicado luego del Block Sorting. Veamos como es el método:

- Para el archivo a comprimir F, se forma una matriz con todas las rotaciones del mismo
- Se ordena la matriz por filas
- La salida del block sorting es la ultima columna más un entero que es el numero de fila en que quedó el archivo inicial

Utilizaremos como ejemplo al archivo “SALABADANZA”, la matriz con rotaciones es:

SALABADANZA
ALABADANZAS
LABADANZASA
ABADANZASAL
BADANZASALA
ADANZASALAB
DANZASALABA
ANZASALABAD
NZASALABADA
ZASALABADAN
ASALABADANZ

Y ordenada por filas:

ABADANZASAL
ADANZASALAB
ALABADANZAS
ANZASALABAD
ASALABADANZ
BADANZASALA
DANZASALABA
LABADANZASA
NZASALABADA
SALABADANZA
ZASALABADAN

La salida del block sorting es entonces la ultima columna (LBSDZAAAAAN) mas el numero de fila en que se encuentra el archivo original (marcado con negrita), que en este caso es 9 (contando que la primer fila es la 0). Se ve que las A se han juntado todas, lo que dará una tira larga de ceros si se le aplica el move to front al archivo

Descompresión

La descompresión no es nada trivial en este método, de hecho a primera vista se puede llegar a pensar que no hay forma de recomponer el archivo original, pero obviamente si esto fuera así no estaríamos viendo el método. El proceso de descompresión es el siguiente:

- Del archivo comprimido se separa lo que es la ultima columna (que la pondremos en un vector llamado C) y el numero de fila, que lo llamaremos I
- El vector C se ordena en otro vector que llamaremos O. Por tener los caracteres del archivo inicial, al ordenarse el vector O que se generará será igual a la primer columna de la matriz ordenada cuando se comprimió el archivo. Tendríamos para el ejemplo algo del estilo:

AXXXXXXXXXL
AXXXXXXXXXB
AXXXXXXXXXS
AXXXXXXXXXD
AXXXXXXXXXZ
BXXXXXXXXXA
DXXXXXXXXXA
LXXXXXXXXXA
NXXXXXXXXXA
SXXXXXXXXX**A**
ZXXXXXXXXXN

Donde X es un caracter por ahora desconocido. La fila 9 se puede marcar en negrita porque sabemos que es la que contiene al archivo original por el valor I que será de 9

- Como cada fila es una rotación del archivo, sabemos que para cada fila, la cadena formada por el ultimo caracter y luego el primero de la misma es una cadena que está incluida en el archivo final. Como la matriz contiene a todas las rotaciones, entonces tendremos todas las subcadenas de 2 caracteres del archivo. Para el ejemplo, sabemos entonces que todas las subcadenas de 2 caracteres son: LA, BA, SA,DA,ZA, AB, AD, AL, AN, AS y NZ. Con estas subcadenas, se puede formar la segunda columna, tomando el caracter de la primera columna como el primero de los caracteres y deduciendo entonces cual es el segundo. Para un mismo valor de primer caracter, el valor del segundo está dado por el orden de las subcadenas. Entonces para las primeras 5 filas, el valor de la segunda columna está dada por las subcadenas AB, AD, AL, AN y AS, y además en ese orden (recordemos que las filas están ordenadas). Formamos entonces la segunda columna de la matriz con el conocimiento reciente

ABXXXXXXXXL
ADXXXXXXXXB
ALXXXXXXXXS
ANXXXXXXXXD
ASXXXXXXXXZ
BAXXXXXXXXA
DAXXXXXXXXA
LAXXXXXXXXA
NZXXXXXXXXA
SAXXXXXXXX**A**
ZAXXXXXXXXN

- Repetimos el paso anterior pero ahora en vez de conocer las subcadenas de longitud 2 podemos formar las de longitud 3: LAB, BAD, SAL, DAN, ZAS, ABA, ADA, ALA, ANZ, ASA y NZA. Podemos formar con ellos la tercera columna, utilizando los valores de las primeras 2 columnas y en base a estos, tomando los valores posibles de las subcadenas de longitud 3. Luego de formar la tercer columna se contaran con las subcadenas de longitud 4 y se podrá formar la cuarta columna, etc
- Una vez que se tenga la matriz completa, se accede a la fila dada por el numero I del archivo (en este caso, la novena) y dicha fila será el archivo inicial

Existe una alternativa más sencilla a este método de reformación de la matriz, que si bien en el trasfondo sigue la misma lógica nos permite resolver la descompresión de una forma más rápida. Esta se explica a continuación:

- Del archivo comprimido se separa lo que es la ultima columna (que la pondremos en un vector llamado C) y el numero de fila, que lo llamaremos I
- El vector C se ordena en otro vector que llamaremos O. Por tener los caracteres del archivo inicial, al ordenarse el vector O que se generará será igual a la primer columna de la matriz. Para el ejemplo, el vector O seria "AAAAABDLNSZ"
- Se forma un vector de enteros que llamaremos P, donde P[i] es la posición del caracter O[i] en el vector C, o sea la posición del iésimo caracter del vector ordenado en la última columna. Cuando haya repetición de caracteres (como en el ejemplo, que hay 5 A), se pondrá sin repetir el menor valor posible hasta el momento. En el ejemplo, tomando nuevamente que la primer posición es la 0 (es totalmente necesario que se tome la misma posición que se tomo cuando se eligió el valor I al comprimir, o la primer posición es la 0 o es la 1) y recordando que C = "LBSDZAAAAAN" y O = "AAAAABDLNSZ" el vector P es "5-6-7-8-9-1-3-0-10-2-4"
- Finalmente se forma el archivo descomprimido D donde :
  - D[0] = C[P[I]]
  - D[1] = C[P[P[I]]]
  - D[2] = C[P[P[P[I]]]]
  - Etc…..
- Para el ejemplo que estamos viendo, I = 9 =>
  - D[0] = C[P[9]] = C[2] = S
  - D[1] = C[P[P[I]]] = C[P[2]] = C[7] = A
  - D[2] = C[P[P[P[I]]]] = C[P[7]] = C[0] = L
  - D[3] = C[P[0]] = C[5] = A
  - D[4] = C[P[5]] = C[1] = B
  - D[5] = C[P[1]] = C[6] = A
  - D[6] = C[P[6]] = C[3] = D
  - D[7] = C[P[3]] = C[8] = A
  - D[8] = C[P[8]] = C[10] = N
  - D[9] = C[P[10]] = C[4] = Z
  - D[10] = C[P[4]] = C[9] = A
  - Aquí se para ya que se tiene el vector completo, se puede dar cuenta fácil porque se llego a la longitud total (sabemos la longitud porque es la misma que la del vector S) o porque otra vez estaríamos inspeccionando el caracter inicial, es decir C[P[I]]. Como se ve, hemos llegado al archivo original que es SALABADANZA

Implementación

Lo primero que salta a la vista es que si bien el utilizar una matriz para las rotaciones sirve para comprender el algoritmo, llevado a la practica es absolutamente imposible. Para comprimir un bloque de 1KB se necesita una matriz de 1KB x 1KB = 1MB! El hecho de ser rotaciones nos facilita un poco las cosas ya que por ejemplo cada fila podría representarse con un offset en el archivo indicando el caracter de inicio de la rotación, y luego para formar la fila se sigue desde el offset hasta el final del archivo y luego desde el inicio hasta el caracter del offset

El segundo problema que se percibe (y mucho) al generar un block sorting es la extremada lentitud que puede llegar a tener en algunos casos. En general hay que pensar que para hacer el ordenamiento de la matriz se deben comparar una cantidad muy grande de strings, y además si el archivo tiene varios substrings iguales, una comparación entre ambos requerirá llegar hasta el primer caracter distinto, para lo cual se irán comparando una cantidad muy grande de caracteres iguales. Esto en total hace que el método sea muy lento y se deba buscar alternativas para optimizar el tiempo de ordenamiento de la matriz

Como ejemplo veremos una que consiste en repetir luego del archivo los primeros N caracteres y luego agregar un valor especial de fin de archivo (o fin de bloque si se procesa el

archivo en varios bloques por no poder entrar todo en memoria). Este valor debe ser mayor al resto (por ejemplo, 257) con lo que las comparaciones entre 2 substrings darán como menor al que no sigue hasta el fin del archivo. La comparación entonces no se hará entre rotaciones completas sino hasta llegar al valor especial, ya que siempre que se encuentre en el mismo este será mayor a otro caracter (que no puede ser también el valor especial en esa posición porque entonces se estaría comparando una rotación consigo misma)

Si tomamos como ejemplo el archivo “AAABAAA”, la comparación entre la rotación “AAAAAAB” (desde la primer A luego de la B) y “AAAABAA” (desde la última A) necesita de comparar 5 caracteres hasta llegar a la B en la segunda rotación. Si se agregan los primeros 2 caracteres luego del archivo y el caracter de fin de archivo que representaremos como *, se tiene “AAABAAAAA*” (se subrayaron los caracteres agregados). La comparación de las mismas rotaciones llevará a comparar “AAAAA*” con “AAA*”. Esta comparación falla luego de 4 caracteres (uno menos que antes) dando como menor a la primer rotación. En textos con grandes repeticiones (y por ende, mas complicados de usar en el apunte porque se prestan a confusiones) se verán muchas ganancias de tiempos de comparación, aunque variará el archivo de salida ya que se está agregando un caracter extra que modifica el orden de las rotaciones. Es importante remarcar que bajo ningún concepto se deben considerar rotaciones que comiencen en los caracteres agregados, ya que no forman parte del archivo

Modelos que aprovechan la transformación BS + MTF

Dado que el block sorting aumenta notablemente la localidad, y el move to front emite gran cantidad de caracteres bajos para ese tipo de archivos, es de pensar que un modelo dinámico de compresión en el que inicialmente todos los caracteres tengan frecuencia 1 no es muy aceptable. De hecho, según estadísticas realizadas luego de aplicar BS + MTF, la probabilidad del 0 suele ser mayor al 50%, y la de los siguientes caracteres (1, 2, etc) muy alta cercana al 10% en cada caso. Sería bueno contar con modelos que le den preferencia a estos caracteres aumentando su probabilidad inicial, con lo que contaremos con mejores estimaciones de las probabilidades y por ende archivos comprimidos mas chicos. A continuación veremos 2 de ellos, ambos creados por Peter Fenwick (foto)

Modelo de Shannon

El primer modelo que veremos es el modelo de shannon, que tiene 5 modelos binarios (solo dos valores son posibles) para los caracteres del 0 al 3 y luego un modelo normal para los caracteres del 4 al 255 y el EOF. Todos los modelos son aritméticos, este requerimiento surge de que en caso de no serlo, para los modelos binarios independientemente de las probabilidades siempre se emitiría 1 bit para cualquiera de los valores.

En los modelos binarios se cuentan con 2 valores posibles de emisión: match y escape. Se emitirá el primero si el caracter a comprimir es el asociado con el modelo y el segundo si no lo es. Los modelos son:

- Modelo 0: Emite “Match” si el caracter es 0, “ESC” si es otro
- Modelo 1: Lo mismo que el modelo 0
- Modelo 2: “Match” si el caracter es 1, “ESC” si es otro
- Modelo 3: “Match” si el caracter es 2, “ESC” si es otro
- Modelo 4: “Match” si el caracter es 3, “ESC” si es otro

Por tener solo 2 valores, en general al match se lo denomina “1” y al escape se lo denomina “0”. A primera vista parece ser que no hay diferencia entre los modelos 0 y 1 ya que ambos reaccionan para el caracter 0. La distinción se hace en que el compresor para cada paso o bien comienza desde el modelo 0 o bien comienza desde el modelo uno, eligiendo el primero de ellos cuando el caracter comprimido anterior fue el 0 y el modelo 1 cuando fue otro

Para ambos casos, si el caracter a comprimir es el 0, se emitirá “Match” en el modelo que esté, y luego se comprimirá el siguiente caracter (comenzando por el modelo 0). En cambio, si no lo es se emitirá un escape y se pasará al modelo 2. En este si el caracter a comprimir es un 1 se emite match y luego se empieza del modelo 1, en cambio si no lo es se emite un escape y se pasa al modelo 2. Esto se repite pasando al 3 y al 4 y en caso de que el caracter fuera mayor al 3 se llegará al modelo 5 donde sí o sí el caracter estará y se lo emitirá con la probabilidad asociada. Inicialmente en cada modelo la probabilidad de cada símbolo es 1, pero el hecho de tener que pasar por distintos modelos hasta llegar a uno que contenga el caracter a comprimir hace que la probabilidad del 0 inicial sea ½, la del 1 ¼, la del 2 1/8, etc

Veamos como funciona con un archivo ejemplo que esta formado por los caracteres
0, 0, 2, 0, 0, 3, 5, 0, 8

Car	M0	M1	M2	M3	M4	M5	Observaciones
0	M(1/2)						Al ser el primer caracter, se comienza desde el modelo 0. Este emite un match y se pasa al siguiente caracter. Se actualizan las probabilidades en el modelo 0
0	M(2/3)						Ahora en el modelo 0 el match se emite con probabilidad de 2/3 por haber aumentado su frecuencia en el paso anterior
2	E(1/4)		E(1/2)	M(1/2)			Se emite un escape en el modelo 0 y en el modelo 2 (no se pasa por el 1). Luego en el modelo 3 se emite un match. Se actualizan frecuencias en modelos 0, 2 y 3 (solo por los modelos en los que paso)
0		M(1/2)					Como el ultimo caracter no fue un 0, se comienza del modelo 1, que tiene al match y al escape con frecuencias iniciales 1
0	M(3/5)						Se vuelve al modelo 0 por ser un caracter 0 el ultimo comprimido
3	E(2/6)		E(2/3)	E(1/3)	M(1/2)		El caracter 3 se encuentra recién en el modelo 4
5		E(1/3)	E(3/4)	E(2/4)	E(1/3)	5(1/253)	Se llega al modelo 5, donde hay 253 símbolos con frecuencia 1.
0		M(2/4)					
8	E(3/7)		E(3/4)	E(2/4)	E(2/3)	8(1/254)	La frecuencia acumulada del modelo 5 es 254 porque el caracter 5 tiene frecuencia 2
EOF		E(2/5)	E(4/5)	E(3/5)	E(3/4)	EOF(1/255)	Fin del archivo

Se ve que en los modelos 2 y 4, al no haber matchs, la probabilidad del escape irá creciendo con lo que al pasar por esos modelos se utilizaran muy pocos bits. también se ve que los matchs del 0, tanto en el modelo 0 como en el 1, se emiten siempre con probabilidad mayor o igual a ½, con lo que al 0 se lo estará codificando siempre con 1 bit o menos. Para caracteres mayores al 3, la probabilidad es pequeña ya que se va acumulando la multiplicación de varios escapes y por eso se terminan necesitando muchos bits para representarlos

Modelo Aritmético

El segundo modelo que veremos es el aritmético, que cuenta con 9 modelos que recorre siempre en el mismo orden, comenzando por el modelo 0 y terminando de ser necesario en el 8. A medida que se avanza en modelos cada vez contienen mas caracteres, de hecho los únicos modelos

binarios que hay son el 0 y el 1 que contienen exactamente a los caracteres 0 y 1 (mas sus escapes correspondiente), luego el modelo 2 ya contiene a dos caracteres (el 2 y el 3) y a su escape. La distribución de caracteres en el modelo es la siguiente:

Modelo	Caracteres
0	0
1	1
2	2, 3
3	4, 5, 6, 7
4	8 al 15
5	16 al 31
6	32 al 63
7	64 al 127
8	128 al 255 y EOF

La forma de comprimir un caracter es similar al modelo de Shannon. Inicialmente todos los caracteres de cada modelo tienen frecuencia 1. Se comienza en el modelo 0 y se emite o bien el 0 si ese era el caracter a comprimir o bien el escape y se pasa al modelo siguiente. Así se continua emitiendo escapes hasta llegar al modelo que contenga al caracter a comprimir, que se emite con su probabilidad respectiva. Por ejemplo, para el mismo archivo analizado con el modelo de shannon la compresión es:

Car	M0	M1	M2	M3	M4	M5	M6	M7	M8
0	0 (1/2)								
0	0 (2/3)								
2	E (1/4)	E (1/2)	2 (1/3)						
0	0 (3/5)								
0	0 (4/6)								
3	E (2/7)	E (2/3)	3 (1/4)						
5	E (3/8)	E (3/4)	E (1/5)	5 (1/5)					
0	0 (5/9)								
8	E (4/10)	E (4/5)	E (2/6)	E (1/6)	8 (1/8)				
EOF	E (5/11)	E (5/6)	E (3/7)	E (2/7)	E (1/9)	E (1/16)	E (1/32)	E (1/64)	EOF (129)

La gran diferencia con el modelo de shannon (aparte de que comienza siempre en el modelo 0) es que en otros modelos que no sean el ultimo hay mas de 1 caracter posible, con lo que las probabilidades inicialmente están distribuidas de forma distinta, por ejemplo la primera vez que se llega al modelo 3 se emite un 5 con 1/5 de probabilidades, esto es porque hay otros 4 símbolos (el 4, el 6, el 7, y el ESC) que tienen frecuencia 1 como el 5. Sin embargo, esta probabilidad es mucho menor que la equivalente del modelo de shannon que seria de 1/253.

En general como para archivos cotidianos el move to front suele emitir un porcentaje importante de caracteres entre 4 y el 20, el modelo estructurado, que les da una mejor distribución de probabilidades que el de shannon, funciona mucho mejor, sin tener grandes dificultades de implementación

Half Coding

El método de half coding es extremadamente útil en salidas de BS+MTF ya que el caracter 0 cumple con el requisito de probabilidades muy altas. Ya se presupone el 0 como caracter mas probable y se utiliza el método de forma dinámica, regenerando los árboles en cada paso (emisión de caracteres o de símbolos especiales). Este método es el que mejor resultados obtiene de los tres.

Anexo

Desigualdad de Kraft

Para saber si un codigo es posible de ser decodificado se debe utilizar la desigualdad de Kraft. Si suponemos que se pueden emitir N simbolos, que representaremos como S0, S1 …. SN-1 y cada uno con su longitud asociada Li, independiente de la del resto, la desigualdad de kraft nos dice que si se cumple que

$$\sum_{i=0}^{N-1} 2^{-Li} \leq 1$$

Entonces el codigo es decodificable. Como ejemplo trivial supongamos que queremos ver cuantos simbolos distintos podemos representar si todos tienen una longitud igual de L:

$$\sum_{i=0}^{N-1} 2^{-Li} = \sum_{i=0}^{N-1} 2^{-L} = 2^{-L} \sum_{i=0}^{N-1} 1 = 2^{-L} * N \leq 1$$

$$N \leq 1/2^{-L}$$

$$N \leq 2^{L}$$

Se ve que como mucho podremos representar a 2^L símbolos distintos y de ahí en mas necesiaríamos mas bits para representarlos o se dejaría de cumplir la desigualdad de Kraft

Ahora veamos para el caso de la entropia, donde la longitud asociada a cada símbolo está dada por su probabilidad de la forma Li = -Log2(Pi):

$$\sum_{i=0}^{N-1} 2^{-Li} = \sum_{i=0}^{N-1} 2^{Log2(Pi)} = \sum_{i=0}^{N-1} Pi \leq 1$$

Como los N simbolos son los unicos posibles, la sumatoria de 0 a N de Pi da 1. Entonces

$$\sum_{i=0}^{N-1} Pi = 1 \leq 1$$

Al ser 1 menor o igual a 1 (de hecho, igual), se cumple la desigualdad de Kraft y podemos decir que las longitudes dadas por la entropía generan codigos decodificables. Lo que nos falta demostrar es que son óptimos. Supongamos que al código Cj lo quisieramos representar con un valor K menor a su longitud. Lj entonces sería igual a –Log2(Pj) – K

Reemplazando en la desigualdad

$$\sum_{i=0}^{N-1} 2^{-Li} = \sum_{i=0, i\neq j}^{N-1} 2^{Log2(Pi)} + 2^{Log2(Pi)+k} = (1-Pj) + Pj * 2^{k} \leq 1$$

$$\sum_{i=0}^{N-1} 2^{-Li} = (1-Pj) + Pj(1-1+2^{k}) = 1 - Pj + Pj + Pj(-1+2^{k}) \leq 1$$

$$\sum_{i=0}^{N-1} 2^{-Li} = 1 + Pj(-1+2^{k}) \leq 1$$

$$1 + Pj(-1+2^{k}) \leq 1$$

$$Pj(-1+2^{k}) \leq 0$$

$$-1 + 2^{k} \leq 0$$

$$2^{k} \leq 1$$

$2^k \leq 2^0$

$k \leq 0$

Hemos llegado a la conclusion que para que se siga cumpliendo la desigualdad de kraft la cantidad K disminuida a la longitud de un simbolo cualquiera tiene que ser menor igual a 0. Esto significa que en realidad se estaría agregando longitud (quitando una cantidad negativa) o se lo estaría dejando igual (si k = 0). Al haber tomado un simbolo cj cualquiera, podemos asegurar que esto ocurre para cualquiera de los símbolos, entonces podemos decir que no solo las longitudes dadas por la entropía generan codigos decodificables sino que además estos son óptimos y no pueden reducirse en su longitud

Estado del arte

En el año 1948 con la teoría de la información de Shannon comienza la larga vida de los compresores. El gran problema inicial fue el límite teórico de la entropía y todos los esfuerzos se centraron en generar códigos que se aproximaran a él lo más posible. Con los árboles de Huffman primero y el compresor aritmético luego (considerado actualmente como el estado del arte de los compresores estadísticos) se alcanzo rápidamente el límite impuesto y muchos predijeron una muerte prematura de la investigación de la compresión

Por suerte estas personas estaban equivocadas, ya que si bien el límite de la entropía continúa hoy siendo una muralla impenetrable, la investigación se centro en como estimar mejor las probabilidades de los símbolos para lograr así seguir disminuyendo el tamaño ocupado. Paralelamente se desarrollaron métodos no estadísticos que al combinarse con el resto generaron niveles de compresión muy buenos. Así nacieron compresores que abusaban de los contextos como la gama completa de PPM, y técnicas para redefinir las probabilidades como el Block Sorting, que genero un avance importante para la compresión y hoy por hoy sigue siendo estudiado buscando su optimización

Sin embargo surgiría un factor no tan considerado hasta el momento: el tiempo. Si bien compresores como el PPMC y el Block Sorting utilizaban muy pocos bits, se necesitaba demasiado tiempo para comprimir y descomprimir archivos. Además este tiempo estaba ligado directamente con la cantidad de recursos disponibles, lo que dificultaba la estandarización, por no servirle a todos los usuarios finales por igual. Así es como hoy en día el ZIP, que no cuenta con un nivel de compresión asombroso, continua liderando indiscutiblemente el área de formatos de compresión, gracias a su excelente desenvolvimiento en todo tipo de computadora que se lo use. Las ganancias en espacio que tienen métodos más avanzados no son lo suficientemente importantes en comparación con las necesidades que tienen, lo que dificulta cualquier tipo de cambio de costumbres

Así es como se produjo el primer cambio de rumbo en la historia de los compresores, pasando de buscar el menor tamaño posible a intentar lograr un balance entre el mismo y el tiempo de compresión y descompresión. En la actualidad son pocas las personas que trabajan directamente en la creación y/o optimización de métodos de compresión y si bien los resultados son muy buenos, es poco probable que logren una aplicación práctica inmediata (de hecho, en ningun caso es ese el objetivo de los investigadores)

En los últimos años, se ha dado un nuevo cambio, quizás mas brusco que el anterior, y que consiste en dejar de lado la idea de poder recuperar el archivo completamente y buscar tamaños excesivamente mas chicos aceptando la pérdida de información que implica superar los límites de la entropía. Estamos hablando de los métodos de compactación, que se basan en la imperfección humana para generar una pérdida de información poco perceptible a nivel de sentidos pero muy notoria a nivel de tamaño ocupado en disco. Combinados con métodos de compresión sin perdida, se han creado formatos de archivos especiales como el JPEG, el MP3 y el DIVX, que aplastan en cuanto a tamaño a sus antecesores BMP, WAV y AVI sin compresión. El auge de Internet con el

envío de audio y video en tiempo real, el crecimiento de la industria de los videojuegos y la multimedización del software en los últimos tiempos hacen que la compactación de datos juegue un papel decisivo en la competencia entre distintos productos, ganando el que logre la mejor calidad con el menor costo posible.

Volviendo así a sus raíces de intentar utilizar la menor cantidad de bits posibles, la historia de los compresores seguirá generando nuevos formatos de archivos con pérdidas de mayor calidad hasta que en algún momento sufra un nuevo giro y se reoriente hacia otro objetivo por ahora desconocido