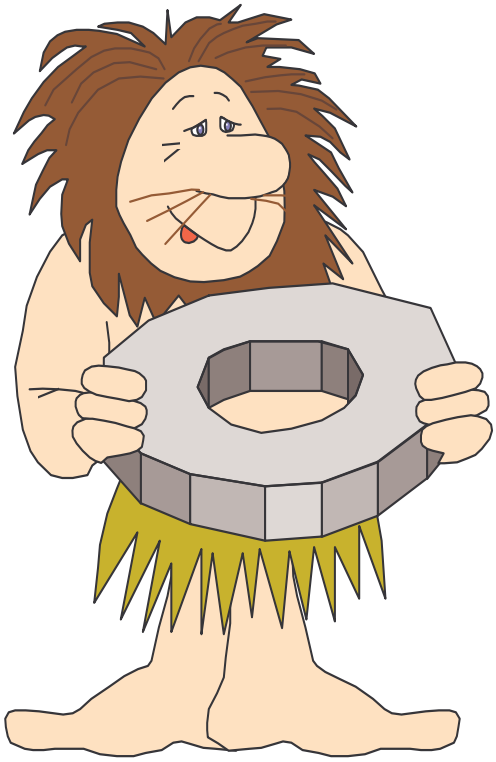
75.06 Organización de Datos.

Introducción a la Teoría de la Información.

Cuarta Edición.



Teoría de la Información.
Compresión de Datos.
Códigos Autocorrectores.
Criptografía.

Sobre la cuarta edición.

La cuarta edición corrige errores de todo tipo que existían en la tercera edición y se han agregado algunos ejemplos de los métodos. Esto no implica que existan más errores o que nuevos errores no se hayan agregado. Si encuentra algún error, por favor avísenos así lo corregimos para la próxima edición.

Sobre la tercera edición.

Cambios en Compresión de Datos.

Nuevos algoritmos de Compresión:
- DMC
- Huffword.
- Gzip.
- LZRW1.
- PPM*.

Nuevos temas sobre compresión de datos.
- Principio de localidad.
- Sincronización, algoritmos auto-sincronizantes.
- Predicción de textos en castellano, limites teóricos. Experiencia de predicción.
- Algoritmos de compresión estadística óptimos para el método de Block-Sorting.
- El modelo de Shannon.
- El modelo Estructurado.

Se corrigieron errores y se revisaron todos los algoritmos de compresión estudiados, la descripción de muchos de los algoritmos fue mejorada. Se incluye un texto comparativo con datos sobre todos los algoritmos estudiados. Se corrigieron numerosos errores, se rehizo la descripción del mecanismo de implosión. A pedido de muchas personas se agrega un seguimiento detallado del algoritmo PPMC.

Cambios en la Edición.

Comentario sobre bibliografía recomendada en cada uno de los temas.
Edición nueva en formato WORD7.
Nuevas tablas y gráficos.

ÍNDICE.

El propósito de este texto es tratar tres temas relacionados con el almacenamiento de información en computadoras digitales: la compresión de datos, los códigos autocorrectores y la encriptación de datos. Estos temas serán desarrollados tomando como marco la 'Teoría de la información" elaborada por Claude Shannon en el año 1948, esta teoría proporciona nomenclatura, conceptos y definiciones que serán utilizados a lo largo del texto. Por esto la primera parte de este apunte desarrolla una introducción a la teoría de la información.

La segunda parte del apunte trata en detalle la compresión de datos. Se proporcionan todos los fundamentos teóricos de la compresión de datos y se estudian distintos algoritmos de compresión basados en diversas técnicas. El apunte reúne textos, ejemplos y gráficos de diversas fuentes. Dada la notoria escasez de material que existe sobre compresión de datos consideramos que este apunte brinda tanto en profundidad como en extensión una cobertura muy amplia del tema.

En la tercera parte del apunte se desarrolla la teoría, la construcción y las propiedades de los códigos autocorrectores y el análisis de canales ruidosos de transmisión de datos. El tema se desarrolla también en el marco de la teoría de la información de Shannon. Esta parte del texto incluye numerosas formulaciones matemáticas que si bien son de escasa complejidad requieren el manejo de algunas fórmulas y teoremas particulares. Todas las fórmulas y teoremas se explican a modo de repaso pero puede que aquellos que no tengan estas formulaciones muy presentes necesiten leer algún material complementario, principalmente en cuanto al cálculo de probabilidades.

La última parte del apunte trata sobre encriptación y desencriptación de datos y está dividida en dos grandes partes: una para algoritmos de clave pública y otra para algoritmos de clave privada. En ambas partes se desarrollan diversas técnicas de encriptación y se mencionan cuales son las debilidades particulares de cada algoritmo si se quiere descifrar el código. Los algoritmos de clave pública han cobrado últimamente un sorprendente interés debido al advenimiento de la 'era de internet' por lo que en forma inesperada esta parte del apunte puede resultar muy interesante en la actualidad.

De acuerdo a la forma en la cual fue escrito el apunte se presupone que el lector tiene conocimientos de programación, estructuras de datos, manejo de distintas bases, sobre todo información representada en binario y en hexadecimal y que se tienen conocimientos, básicos, de probabilidad y estadística.

El 90% de este apunte es de carácter netamente técnico, la cantidad de información que se recibe es muy alta en comparación con la longitud del apunte por lo que puede que sea necesario leer varias veces alguno de los temas para poder comprenderlo en su totalidad, la tercera edición incorporó varios ejemplos adicionales a la primera edición con los cuales esperamos que los temas mas complejos puedan estudiarse con mayor facilidad. La segunda edición incorporó, también, algunos temas nuevos que son importantes y fueron omitidos en la primera edición. Numerosos errores fueron corregidos y otros nuevos se agregaron.

Este apunte no podría haber sido escrito sin la colaboración de varios alumnos del curso de organización de datos del primer cuatrimestre de 1997 quienes nos acercaron sus opiniones, sus ideas y fundamentalmente aliento. Esperamos que disfruten del apunte tanto como nosotros disfrutamos en escribirlo.

La cátedra.

Primera Parte:

Teoría de la Información

Hace ya mucho tiempo que la información es almacenada en forma digital, sin embargo hubo un periodo durante el cual las computadoras eran una novedad, y donde los únicos medios de almacenamiento conocidos eran los libros. Es en ese marco que un matemático francés: Claude Shannon se dedicó a elaborar una teoría sobre como tratar a la información almacenada en forma digital. Sobre eso trata nuestro apunte y esa es la razón por la cual la primera parte del apunte esta destinada a entender los fundamentos básicos de la Teoría de la Información.

Existen muchas definiciones de 'información'. Dependiendo del autor o del contexto en el cual se discuta el tema podemos encontrar aproximaciones de lo más variadas. En este apunte trabajaremos con información que se almacena en computadoras. Una computadora en su nivel mas bajo conoce solamente dos símbolos (por ahora): el cero y el uno, pues funciona en base a circuitos electrónicos que utilizan elementos bi-estables; en este texto trataremos sobre problemas relacionados con el almacenamiento de información en computadoras digitales, estudiaremos técnicas de compresión de datos para lograr almacenar mensajes utilizando la menor cantidad de bits posibles, observaremos técnicas de encriptación para ocultar la información almacenada de miradas indiscretas y desarrollaremos técnicas de autocorrección para proteger a la información almacenada frente a posibles fallas (ninguna computadora está garantizada contra fallas, la suya tampoco).

Nos basaremos en este apunte en la denominada 'teoría de la información' elaborada por Shannon en 1948 y que ha cobrado mucha vigencia con el advenimiento de la era de las computadoras digitales. Para comenzar debemos partir de algunas definiciones sobre la nomenclatura a utilizar, nomenclatura basada en la teoría de la información de Shannon.

FUENTE: Una fuente es todo aquello que emite mensajes. Por ejemplo, una fuente puede ser una computadora y mensajes sus archivos, una fuente puede ser un dispositivo de transmisión de datos y mensajes los datos enviados, etc. Una fuente es en sí misma un conjunto finito de mensajes: todos los posibles mensajes que puede emitir dicha fuente. En compresión de datos tomaremos como fuente al archivo a comprimir y como mensajes a los caracteres que conforman dicho archivo.

MENSAJE: Un mensaje es un conjunto de ceros y unos. Un archivo, un paquete de datos que viaja por una red y cualquier cosa que tenga una representación binaria puede considerarse un mensaje. El concepto de mensaje se aplica también a alfabetos de mas de dos símbolos, pero debido a que tratamos con información digital nos referiremos casi siempre a mensajes binarios.

CÓDIGO: Un código es un conjunto de unos y ceros que se usan para representar a un cierto mensaje de acuerdo a reglas o convenciones preestablecidas. Por ejemplo al mensaje 0010 lo podemos representar con el código 1101 usando para codificar la función (NOT). La forma en la cual codificamos es arbitraria.

Un mensaje puede, en algunos casos representarse con un código de menor longitud que el mensaje original. Supongamos que a cualquier mensaje S lo codificamos usando un cierto algoritmo de forma tal que cada S es codificado en L(S) bits, definimos entonces a la información contenida en el mensaje S como la cantidad mínima de bits necesarios para codificar un mensaje.

INFORMACIÓN: La información contenida en un mensaje es proporcional a la cantidad de bits que se requieren como mínimo para representar al mensaje.

El concepto de información puede entenderse mas fácilmente si consideramos un ejemplo.
Supongamos que estamos leyendo un mensaje y hemos leído "string of ch", la probabilidad de que el mensaje continúe con "aracters" es muy alta por lo tanto cuando realmente leemos "aracters" del archivo la cantidad de información que recibimos es muy baja pues estabamos en condiciones de predecir que era lo que iba a ocurrir. La ocurrencia de mensajes de alta probabilidad de aparición aporta menos información que la ocurrencia de mensajes menos probables. Si luego de "string of ch" leemos "imichurri" la cantidad de información que recibimos es mucho mayor.

•Clasificación de Fuentes:

Por la naturaleza generativa de sus mensajes una fuente puede ser aleatoria o determinística.

Por la relación entre los mensajes emitidos una fuente puede ser estructurada o no estructurada (o caótica).

Existen varios tipos de fuentes. Para la teoría de la información interesan las fuentes aleatorias y estructuradas. Una fuente es aleatoria cuando no es posible predecir cual es el próximo mensaje a emitir por la misma. Una fuente es estructurada cuando posee un cierto nivel de redundancia, una fuente no estructurada o de información pura es aquella en que todos los mensajes son absolutamente aleatorios sin relación alguna ni sentido aparente, este tipo de fuente emite mensajes que no se pueden comprimir, un mensaje para poder ser comprimido debe poseer un cierto nivel de redundancia, la información pura no puede ser comprimida pues perderíamos un grado de conocimiento sobre el mensaje.

Ejemplos:

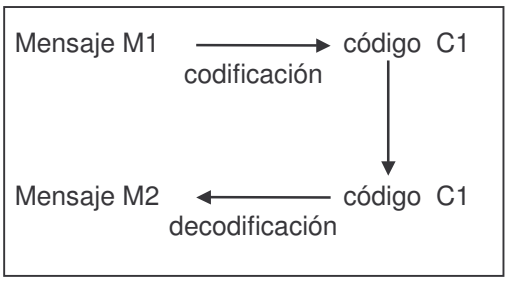
Fuente Estructurada.

```
ABABEEEENHABIAUN
AVEZIIIILOLOPEPE
0123456789AAAAAA
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
AATRSSSTRSSSTRSS
S000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000HABIAUNAV
EZUNAVACAENLAQUE
BRADADELHUMAHUAC
AHHHHHHHHJJJJJHH
HCCHCNC,.CNNCXMH
DSJHDSKHDKSEWIOW
```

Fuente no-estructurada.

```
KJHFSDDBFMNXBVCY
SGHFDSBFABFAGS7D
TFSGDF4K7RFGSGLB
CVMSA98VSC,VMASD
FVYWW4RN43.N.,NS
VVYSD.8FYGG43///
RNGGGSFD;Y8VV834
4,G.N,NSSG8HQ34T
JKHFSSSPUGGGQ932
222;QQQQQL/DFFV9
34UTT34M,.GF/ERG
77Y3448;OTFYS0D9
YGH/324[GOFV90Y3
222/;4TGG3KJ4-
7GYFS.VVV/34G888
DFU/G340T888REW.
.
```

Distintos tipo de códigos, códigos decodificables.



Como vimos, un código es una representación binaria de un cierto mensaje, el proceso de codificación y decodificación es simple e intuitivo, el codificador recibe un mensaje "m1" y emite un código, el decodificador recibe un código y emite un mensaje "m2", considerando que nuestros codificadores pretenderán no perder información debemos pedir que m1 sea igual a m2, es decir que el código es decodificable.

Un código es decodificable sí y solo sí un código solo puede corresponder a un único mensaje.

Ejemplo:
Sea el siguiente esquema de codificación:

a=0
b=01
c=10

Si el decodificador recibe el código: "0010" no puede distinguir si el mensaje original fue "aba" o "aac", ya que puede interpretarlo como 0 01 0 o como 0 0 10

Este tipo de código no-decodificable se denomina "libre de prefijo", para construir códigos libres de prefijo para un alfabeto puede usarse casi cualquier función, aunque todo el proceso de codificación en sí deja de tener sentido pues no es posible decodificar correctamente los mensajes.

Ejemplo 2:
Sea el siguiente esquema:

a=0
b=01
c=11

El código "00111" solo puede corresponder a "abc" (intente decodificarlo de otra forma!) sin embargo el decodificador es incapaz de deducir esto a medida que va leyendo los códigos, necesita saber cuales van a ser los próximos bits para poder interpretar los anteriores, este tipo de código no se considera decodificable (aunque lo sea) y también tendremos que evitarlo.

•Códigos prefijos.

Los códigos prefijos son códigos decodificables, cada código se puede reconocer por poseer un prefijo que es único al código, el prefijo de un código no puede ser igual a un código ya existente.

Ejemplo:

a=0
b=10
c=11

Es un código prefijo, notar que si agregamos mas códigos estos no pueden comenzar con "0" ni "10" ni "11" pues el decodificador los confundiría con "a" "b" o "c", en consecuencia a este código no podríamos agregarle nuevos códigos.

•Entropía de una fuente.

Ya hablamos del concepto de información, veremos ahora como se mide la misma . De acuerdo a la teoría de la información, el nivel de información de una fuente se puede medir según la entropía de la misma. Los estudios sobre la entropía son de suma importancia en la teoría de la información y se deben principalmente a Shannon, existen a su vez un gran numero de propiedades respecto de la entropía de variables aleatorias debidas a Kolmogorov.

Dada una fuente "F" que emite mensajes, resulta frecuente observar que los mensajes emitidos no resulten equiprobables sino que tienen una cierta probabilidad de ocurrencia dependiendo del mensaje. Para codificar los mensajes de una fuente intentaremos pues utilizar menor cantidad de bits para los mensajes más probables y mayor cantidad de bits para los mensajes menos probables de forma tal que el promedio de bits utilizados para codificar los mensajes sea menor a la cantidad de bits promedio de los mensajes originales. Esta es la base de la compresión de datos.

A este tipo de fuente se la denomina fuente de orden-0 pues la probabilidad de ocurrencia de un mensaje no depende de los mensajes anteriores, a las fuentes de orden superior se las puede representar mediante una fuente de orden-0 utilizando técnicas de modelización apropiadas.

Definimos a la probabilidad de ocurrencia de un mensaje en una fuente como la cantidad de apariciones de dicho mensaje dividido el total de mensajes.

Supongamos que Pi (P sub i) es la probabilidad de ocurrencia del mensaje-i de una fuente, y supongamos que Li es la longitud del código utilizado para representar a dicho mensaje, la longitud promedio de todos los mensajes codificados de la fuente se puede obtener como:

$$H = \sum_{i=0}^{n} Pi * Li$$

Promedio ponderado de las longitudes de los códigos de acuerdo a sus probabilidades de ocurrencia, al número "H" se lo denomina "Entropía de la fuente" y tiene gran importancia. La entropía de la fuente determina el nivel de compresión que podemos obtener como máximo para un conjunto de datos, si consideramos como fuente a un archivo y obtenemos las probabilidades de ocurrencia de cada caracter en el archivo podremos calcular la longitud promedio del archivo comprimido, se demuestra que **no es posible comprimir estadísticamente un mensaje/archivo mas allá de su entropía.** Lo cual implica que considerando únicamente la frecuencia de aparición de cada caracter la entropía de la fuente nos da el límite teórico de compresión, mediante otras técnicas no-estadísticas puede, tal vez, superarse este límite.

El objetivo de la compresión de datos es encontrar los Li que minimizan a "H", además los Li se deben determinar en función de los Pi, pues la longitud de los códigos debe depender de la probabilidad de ocurrencia de los mismos (los mas ocurrentes queremos codificarlos en menos bits).

Se plantea pues:

$$H = \sum_{i=0}^{n} Pi * f(Pi) \quad \text{(Minimizar)}$$

A partir de aquí y tras intrincados procedimientos matemáticos que fueron demostrados por Shannon oportunamente se llega a que H es mínimo cuando f(Pi) = log2 (1/Pi).

Entonces:

La longitud mínima con la cual puede codificarse un mensaje puede calcularse como Li=log2(1/Pi) = -log2(Pi). Esto da una idea de la longitud a emplear en los códigos a usar para los caracteres de un archivo en función de su probabilidad de ocurrencia.

Reemplazando Li podemos escribir a H como:

$$H = \sum_{i=0}^{n} Pi * (-\log_2 Pi)$$

$$H = \sum_{i=0}^{n} Pi * (-\log_2 Pi) \quad \text{(Repetido por Conveniencia)}$$

De aquí se deduce que la entropía de la fuente depende únicamente de la probabilidad de ocurrencia de cada mensaje de la misma, por ello la importancia de los compresores estadísticos (aquellos que se basan en la probabilidad de ocurrencia de cada caracter). Shannon demostró, oportunamente que no es posible comprimir una fuente estadísticamente mas allá del nivel indicado por su entropía.

•Un mundo de 8 bits.

Como sabemos un archivo en una computadora es una secuencia de BITS, sin embargo en nuestras definiciones de entropía y longitud ideal de los caracteres estamos considerando la probabilidad de ocurrencia de caracteres es decir bloques de 8 bits, podríamos pensar si los resultados cambiarían considerando bytes de 9 o 7 bits, por ejemplo. La respuesta es afirmativa, pero la entropía tendería a aumentar a medida que nos alejamos del valor de 8 bits por byte. El motivo por el cual ocurre esto reside en que en cualquier archivo almacenado en una computadora los elementos que pueden ser dependientes unos de otros son los bytes y no los bits, esto hace a la estructura de la fuente, concepto del cual hemos hablado. A medida que nos alejamos del valor de 8 bits para nuestro concepto de byte perdemos estructura en nuestra fuente y la misma cada vez se vuelve mas aleatoria por lo que podremos comprimirla en menor medida. De aquí surge que tomemos siempre bytes de 8 bits, tarea que puede sonar obvia pero que merece mas de una reflexión.

Ejemplo:

Sea el siguiente string/archivo/fuente:

"Holasaludosatodos" (17 bytes)

Tenemos la siguiente tabla:

Carácter	Frecuencia	Probabilidad	Longitud Ideal
H	1	1/17=0.0588	-log2(0.0588)=4.0874 bits
o	4	4/17=0.2353	-log2(0.2353)=2.0874 bits
l	2	2/17=0.1176	-log2(0.1176)=3.0874 bits.
a	3	3/17=0.1765	-log2(0.1765)=2.5025 bits
s	3	3/17=0.1765	-log2(0.1765)=2.5025 bits
u	1	1/17=0.0588	-log2(0.0588)=4.0874 bits
d	2	2/17=0.1176	-log2(0.1176)=3.0874 bits
t	1	1/17=0.0588	-log2(0.0588)=4.0874 bits

H = 3 * 0.0588 * 4.0874 + 0.2353 * 2.0874 + 2 * 0.1176 * 3.0874 + 2 * 0.1765 * 2.5025

H = 2.82176233222 bits x byte.

H * 17 = 47.96 bits.

El string en cuestión no puede ser comprimido en menos de 47.96 bits, es decir unos 6 bytes. Este es el límite teórico e ideal al cual puede comprimirse nuestra fuente.

Ejercicios:

1) Calcular la entropía de los siguientes strings:

a) "Mimamamemima"
b) "Aconcagua003"
c) "Abcdefghijkl"
d) "AAAAAAAAAAAAAAAAAAAAA"

2) Conociendo H, ¿puede deducirse si un string puede comprimirse? ¿cómo?

•La desigualdad de Kraft.

Sea el alfabeto A={a0,a1,...,an-1} la cantidad de símbolos de "A" es |A|=N.

Supongamos que Cn es el código libre de prefijo correspondiente al caracter An y sea Ln la longitud de Cn. (Notar que Cn puede ser cualquier código)

Definimos al número de Kraft "K" como.

$$K = \sum_{i=0}^{n} 2^{(-Li)}$$

Se demuestra que **para un código prefijo K debe ser menor o igual a uno**, esta es la desigualdad de Kraft que no demostraremos aquí.

$$\sum_{i=0}^{n} 2^{(-Li)} \leq 1 \quad \text{desigualdad de Kraft.}$$

Esta desigualdad es fundamental pues si queremos un código decodificable sabemos que tenemos que satisfacer la desigualdad de Kraft.

Sabemos que la longitud Li de un código debe ser al menos -log2(Pi) para satisfacer la entropía, si además queremos que el código sea decodificable debemos pedirle que cumpla la desigualdad de Kraft, supongamos que Ei es la cantidad de bits que debemos agregar a la longitud determinada por la entropía para que el código sea decodificable.

Sea:

$$Li = -\log 2(Pi) + Ei \quad \text{(Ei es la longitud extra que debemos agregar)}$$

$$2^{(-Li)} = 2^{(-\log 2(Pi)} * 2^{-Ei} \quad \text{con} \quad 2^{\log 2(Pi)} = Pi \text{ (daddah!)}$$

$$= Pi * 2^{(-Ei)}$$

Planteando el número de Kraft y reemplazando Li por el valor calculado.

$$K = \sum_{i=0}^{n} Pi * 2^{(-Ei)}$$

Sea Ei = E para todo i (caso mas simple en el cual a todos los códigos le agregamos la misma longitud)

$$K = \sum_{i=0}^{n} \frac{Pi}{2^{E}} = \frac{1}{2^{E}}$$

Por la desigualdad de Kraft.

$$\frac{1}{2^{E}} <= 1$$

$$2^{(-E)} <= 2^{0} \quad (2^{0} = 1 \text{ dadah!})$$

$$-E <= 0$$

$$E >= 0$$

Por lo tanto un código solo es decodificable si su longitud es mayor o igual a la determinada por la entropía.

Segunda Parte:

Compresión de Datos.

La compresión de datos es una verdadera ciencia dentro de otra. La idea fundamental es muy simple, guardar la mayor cantidad de datos posibles en la menor cantidad de espacio posible. Los algoritmos de compresión a lo largo de la historia fueron progresando comprimiendo cada vez mas. Hoy en día coexisten algoritmos muy simples, otros muy complejos, algunos muy extraños y otros muy ingeniosos. Este capítulo revisa casi todos los algoritmos de compresión existentes y como veremos el mundo esta lleno de sorpresas.

•Compresión de datos, conceptos.

La idea fundamental de la compresión de datos es reducir el tamaño de los archivos de forma tal que estos ocupen menos espacio y que dado el archivo comprimido pueda recuperarse el archivo original (simple!)

La compresión de datos esta sólidamente fundamentada en toda la teoría que antes desarrolláramos: Para la compresión de datos la fuente es el archivo a comprimir y los mensajes son los caracteres que componen el archivo. La idea de la compresión de datos es, dado un archivo, representar a los caracteres mas probables con menos bits que los caracteres menos probables de forma tal que la longitud promedio del archivo comprimido sea menor a la del archivo original.

Debemos distinguir compresión de compactación, la compactación se refiere a técnicas destructivas, es decir que una vez compactado un archivo se pierde información y no puede recuperarse el original, la compactación se puede aplicar a imágenes o sonido para las cuales la perdida de fidelidad puede llegar a justificarse por el nivel de compresión.

•Teorema fundamental de la compresión de datos.

"No existe un algoritmo que sea capaz de comprimir cualquier conjunto de datos"

Técnicamente:

"Si un compresor comprime un conjunto de datos en un factor F entonces necesariamente expande a algún otro conjunto de datos en un factor MAYOR que F"

Demostración:

Supongamos un conjunto de N caracteres y supongamos también que a todos los caracteres los representamos por códigos de igual longitud entonces:

Ln=log2(N)

Veamos si se cumple la desigualdad de Kraft.

$$K = \sum_{0}^{n} 2^{(-Li)} = \sum_{0}^{n} 2^{(-\log 2(N))} = \sum_{0}^{n} \frac{1}{2^{\log 2(N)}} = \frac{N}{N} = 1 \quad \text{(vale)}$$

Fijemos las longitudes log2(N) para 254 de nuestros 256 caracteres: L2..L255 y supongamos que L0 se codifica con una longitud menor dejando libre L1 para ver que pasa.

Supongamos ahora que nuestro compresor comprime al caracter L0 en "D" bits.

L0=log2(N)-D

Suponiendo que los caracteres L2..L255 se codifican con la misma longitud de antes, queremos ver cual será la longitud de L1.

Li = log2(N) para 2<=i<N

Buscamos el L1 mínimo:

$$K = \sum_{0}^{255} 2^{(-Li)} \quad \text{número de Kraft}$$

$$= \sum_{0}^{255} 2^{(-\log 2(N))} + 2^{(-L0)} + 2^{(-L1)} - 2^{(-\log 2(N))} - 2^{(\log 2(N))}$$

$$= 1 + 2^{(-L0)} + 2^{(-L1)} - 2^{(-\log 2(N))} - 2^{(-\log 2(N))}$$

Por la desigualdad de Kraft sabemos que K<=1.

$$2^{(-L0)} + 2^{(-L1)} - 2^{(-\log 2(N))} - 2^{(-\log 2(N))} <= 0$$

$$2^{(-L1)} <= 2 * 2^{(-\log 2(N))} - 2^{(-L0)}$$

Desarrollamos $2 * 2^{(-\log 2(N))}$

$$= \frac{2}{\log 2(N)} = \frac{2}{N}$$

y $2^{(-L0)} = 2^{(D - \log 2(N))} = \frac{2^{D}}{2^{(\log 2(N))}} = \frac{2^{D}}{N}$

Entonces:

$$2^{(-L1)} <= \frac{2}{N} - \frac{2^D}{N}$$

$$\frac{N}{2^{L1}} <= 2 - 2^D$$

Sea $L1 = \log 2(N) + E$ (y queremos calcular E)

$$2^{L1} = N * 2^E$$

$$\frac{N}{2^{L1}} = \frac{1}{2^E}$$

$$2^{(-E)} <= 2^1 - 2^D$$

$$E >= -\log 2(2 * (1 - 2^{(D-1)}))$$

$$E >= -1 - \log 2(1 - 2^{(D-1)})$$

De aquí se deduce:

1) D<=1 con lo cual observamos que si solo se cambia la longitud de dos códigos entonces las longitudes cambian a lo sumo un bit.

2) Si D=1 entonces E>=∞ !!!

3) Si D=0 entonces E>0 en general E=c(D)*D con c(D)>1
por ejemplo si D=0.5
E>=0.77155

De aquí se demuestra que si achicamos L0 en 0.5 bits debemos agrandar L1 en 0.77155 bits, lo cual demuestra el teorema.

Si todos los caracteres son equiprobables comprimir uno de ellos implica expandir el archivo.

El teorema puede 'mostrarse' en una forma menos técnica razonando de la siguiente manera: Si un algoritmo comprime 'cualquier' archivo entonces comprime a todos los archivos de 'n' bits. Supongamos que los comprime solo un bit (lo mínimo que le podemos pedir). Hay 2^N archivos de N bits y 2^{N-1} archivos de N-1 bits, por lo tanto habrá por lo menos 2 archivos de N bits que se comprimieron generando el MISMO archivo de N-1 bits, por que resulta imposible recuperar el archivo original a partir del comprimido.

El razonamiento es claro pero sin embargo ...

•El episodio del Datafile/16.

En abril de 1992 WEB technologies de Smyrna anuncia el desarrollo de un programa que comprimiría a cualquier archivo de mas de 64Kb a un dieciseisavo de su longitud original (en promedio), WEB no solo proclamaba eso sino también que el algoritmo podía ser aplicado en forma recursiva a archivos ya comprimidos por él mismo con idéntico resultado. El revuelo causado entre los neófitos por tal anuncio fue de tal magnitud que el newsgroup sobre compresión de datos se vio sobresaturado de mensajes sobre el famoso programa, infructuosos fueron los intentos de los entendidos en explicar que tal cosa no era posible, la gente quería tener ese programa!

En Junio de 1992 WEB vuelve al ataque anunciando que prácticamente cualquier cantidad de bytes podían ser comprimidos en 1Kb aplicando el algoritmo recursivamente, el ingenio incluso ya tiene un nombre: Datafile/16.

Esto esta tomado textualmente de un texto de WEB technologies:

"DataFiles/16 will compress all types of binary files to approximately one-sixteenth of their original size ... regardless of the type of file (word processing document, spreadsheet file, image file, executable file, etc.), NO DATA WILL BE LOST by DataFiles/16. Performed on a 386/25 machine, the program can complete a compression/decompression cycle on one megabyte of data in less than thirty seconds"

"The compressed output file created by DataFiles/16 can be used as the input file to subsequent executions of the program. This feature of the utility is known as recursive or iterative compression, and will enable you to compress your data files to a tiny fraction of the original size. In fact, virtually any amount of computer data can be compressed to under 1024 bytes using DataFiles/16 to compress its own output files muliple times. Then, by repeating in reverse the steps taken to perform the recusive compression, all original data can be decompressed to its original form without the loss of a single bit. Convenient, single floppy DATA TRANSPORTATION"

El 28 de Junio Earl Bradley de WEB technologies anuncia un importante acuerdo con una empresa de Sillicon Valley a la cual se le vendería el algoritmo.

WEB continua tirando pistas durante Junio, anuncian que el código ocupa 4K y que solo usa aritmética entera y un buffer de 65Kb.

Mientras el compresor no aparecía las sospechas se hacían mayores, pero la gran mayoría de los usuarios realmente creía que el producto existía,
ante reiterados pedidos WEB insistía en que el acuerdo con Sillicon Valley le impedía lanzar inmediatamente el producto.

El 10 de Julio de 1992 WEB realiza uno de los anuncios mas cómicos en la historia de la computación, leer para creer:

“El equipo de testeo ha descubierto un nuevo problema en el cual cuatro números en una matriz implicaban un mismo valor. Los programadores se encuentran trabajando en un preprocesador para eliminar este raro caso."

El 3 de agosto de 1992 WEB anuncia que sus programadores fueron incapaces de resolver el susodicho problema y que el proyecto se cancelaba. La conclusión que debemos obtener de esto es que no existen los milagros, no se puede ir en contra de las propiedades demostradas matemáticamente y no es posible comprimir cualquier conjunto de datos. El texto es anecdótico y resulta cómico pero millones de personas fueron engañadas durante mucho tiempo. (Y alguna empresa de Sillicon Valley también).

•Compresión estadística.

Los compresores estadísticos conforman la columna vertebral de la compresión de datos. La idea básica es: El compresor "adivina" cual va a ser el input y escribe menos bits al output si adivinó correctamente. Notemos que el descompresor debe "adivinar" de la misma forma que el compresor para poder decodificar el archivo.

Un compresor estadístico es aquel que se basa en la probabilidad de ocurrencia de los caracteres (mensajes) en un archivo (fuente) para comprimirlo.

Los compresores estadísticos buscan representar a cada caracter con un número de bits proporcional a -log2(Pi) (longitud ideal de acuerdo a la entropía).

Por ejemplo si solo tenemos cuatro caracteres a,b,c y d y el compresor sabe que la probabilidad de "a" es 0.5, "b" es 0.25, "c" es 0.25 y "d" es 0. Entonces intentaremos codificar de la forma:

1 bit para "a" (-log2(.5)=1)
2 bits para b y c (-log2(.25)=2)

Por ejemplo:

a : 0
b : 10
c : 11

Como "d" tiene probabilidad de ocurrencia cero ni siquiera lo codificamos.

Notar que existen varias otras posibilidades Ej {1,01,00} todas las combinaciones valen mientras se respeten las longitudes de los códigos, y estos sean decodificables (prefijos).

Si todos los caracteres ASCII son equiprobables para un archivo entonces usaremos -log2(1/256) = 8 bits para codificar cada uno, es decir que no podemos comprimir el archivo.

En un texto como este el caracter espacio es muy ocurrente por lo que podríamos representarlo con 3 bits mientras que el ";" es muy raro por lo que podríamos usar 12 bits o incluso mas para representarlo.

Las longitudes Li de los caracteres se obtienen de -log2(Pi), notemos que Li puede ser un número no-entero, la mayoría de los compresores estadísticos deben usar códigos de longitud entera por lo que desperdiciarán algunos bits en promedio debido al redondeo, existen compresores estadísticos que pueden usar códigos de longitudes no enteras, se denominan compresores aritméticos y resultan los mas eficientes.

Una forma de evitar el redondeo es ablocando símbolos, por ejemplo:

AA : 0
AB : 10
BA : 110
BB : 111

Con lo cual A es representado en alrededor de 0.7 bits y B en 1.4 bits (valores no enteros), el ablocar códigos para lograr valores enteros para los bloques que correspondan a valores no-enteros de los caracteres es sin embargo una tarea que puede llegar a resultar mucho mas costosa que el proceso de compresión en si por lo que su utilidad es únicamente teórica.

El objetivo de la compresión estadística es en definitiva determinar cual deberá ser la longitud de los códigos correspondientes a los caracteres de un archivo, el valor de los códigos puede ser cualquiera siempre y cuando se respeten las longitudes, veremos a continuación dos técnicas que se utilizan frecuentemente para obtener las longitudes de estos códigos.

La compresión estadística se vale para poder comprimir una determinada fuente en el caracter estructurado de la misma, aquellas fuentes que son totalmente aleatorias, es decir de información pura no pueden comprimirse, es por esto que la mayoría de los algoritmos de compresión y programas compresores que los utilizan tienden a expandir a los archivos que fueron generados en forma aleatoria, estos archivos no presentan ninguna relación evidente entre sus bytes (si el algoritmo de generación de random es bueno!) por lo que son un claro ejemplo de fuente no estructurada. La compresión estadística no se ve afectada por la incapacidad de comprimir este tipo de archivos ya que el 90% de la información que se almacena en una computadora es de caracter netamente estructurado por lo que podremos comprimirla con facilidad. Cuanto mejor sea para archivos no aleatorios y peor para archivos random mejor será un algoritmo de compresión estadístico.

Códigos de Huffman.

Versiones:
D. Huffman. (1952)

•Huffman estático.

Este algoritmo desarrollado por Huffman en 1952 es el mas popular y utilizado de los compresores estadísticos existiendo numerosas variantes, su popularidad se basa en que se demuestra que un árbol de Huffman es óptimo para la compresión de una cierta cantidad de caracteres ('cualquier' otro árbol que se use para comprimir genera un archivo igual o mayor al generado por Huffman).

El sentido de la palabra 'óptimo' aquí es el siguiente: Si las longitudes de los códigos deben ser enteras entonces las longitudes que genera un árbol de Huffman son óptimas. Cualquier otro árbol que genere longitudes enteras es igual o inferior al de Huffman.

Huffman inventó el método como respuesta a un desafío lanzado por un profesor de la universidad que ofrecía la aprobación completa del curso a aquellos alumnos que desarrollaran alguna técnica o algoritmo novedoso. Como podrán imaginarse el ingenioso Huffman se salió con la suya y durante mas de 20 años el “método de Huffman” fue reconocido como el mejor algoritmo de compresión de datos.

La técnica para construir el árbol es la siguiente:

Se le asigna a cada caracter que aparece en el archivo su frecuencia de aparición, caracteres con frecuencia cero se descartan pues no van a aparecer.

Se tienen pues n conjuntos de un caracter cada uno, con su respectiva frecuencia, a continuación mientras la cantidad de conjuntos sea mayor que uno se toman los dos conjuntos cuya sumatoria de frecuencias sea menor y se unen. Cuando queda un único conjunto queda formado un árbol binario, veamos:

Sea el archivo:

AEBEGCEAEDBEAFG

Los siguientes son los caracteres y su frecuencia:

A	3
B	2
C	1
D	1
E	5
F	1
G	2

Tomamos los dos conjuntos de menor frecuencia "C" y "D" que se juntan en un nuevo conjunto de frecuencia=2

Los dos conjuntos de menor frecuencia son ahora "F" y "CD" de frecuencias 2 y 1 (los juntamos en un conjunto de frecuencia 3)

Para el próximo paso juntamos "B" y "G" de frecuencias 2 cada uno.

Ahora hay que juntar "A" con "CDF" de frecuencia 3 cada uno.

La próxima unión es entre "BG" (4) y "E" (5)

Por ultimo se unen los dos únicos conjuntos que quedan.

Solo queda un conjunto por lo que queda formado el árbol.

Como se puede observar queda formado un árbol binario en el cual cada una de las hojas del árbol corresponde a cada uno de los caracteres del archivo.

El siguiente paso consiste en asignar a cada caracter un código binario, para ello lo único que hay que hacer es fijar como convención que rama del árbol es la cero y cual es la 1, supongamos que izquierda es cero y derecha es uno.

Es importante mencionar en este punto que es fundamental que compresor y descompresor funcionen coherentemente. Si así no lo hicieren el mensaje obtenido no sería igual al emitido. Por ejemplo si el descompresor utilizara un criterio opuesto al compresor, el resultado sería completamente erróneo. **Este punto debe ser tomado en cuenta en todos los métodos que se desarrollen**.

El número que se le asigna a cada caracter depende del camino que hay que seguir para llegar a su hoja desde la raíz, para la "A" tenemos que doblar a la izquierda dos veces por lo que su código será 00 (en binario).

Los códigos que resultan son:

Char	Freq	Código
A	3	00
B	2	100
C	1	0100
D	1	0101
E	5	11

El archivo comprimido ocuparía:
3x2 + 3x2 + 4x1 + 4x1 + 5x2 + 1x3 + 2x3 = 33 bits.

Como se puede observar necesitamos menos bits para los caracteres que aparecen mas frecuentemente en el archivo y que los códigos generados son prefijos.

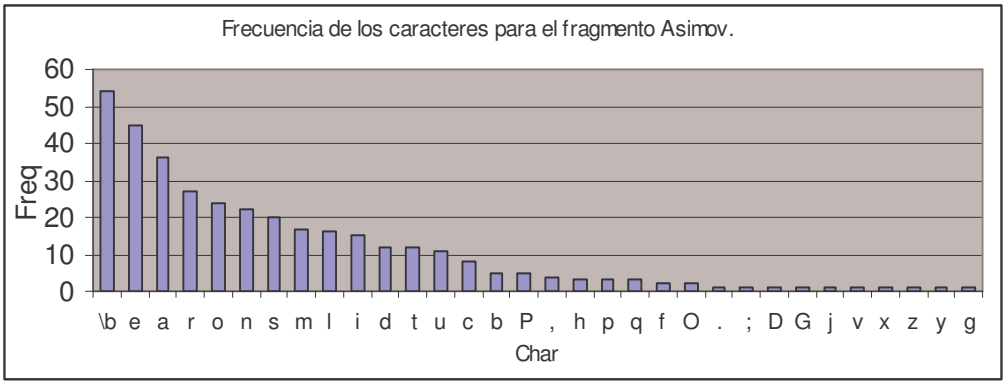
Para comprimir el archivo, lo único que hay que hacer es ir leyendo el archivo original y reemplazar a cada caracter leído por su código correspondiente, además el compresor deberá, también, almacenar el árbol en el archivo comprimido pues el descompresor necesita saber cual fue el árbol utilizado para poder descomprimir.

El descompresor deberá primero recuperar el árbol y luego va descomprimiendo procesando bit por bit, comienza en la raíz del árbol y va doblando según encuentre ceros o unos, cuando llega a una hoja emite el caracter encontrado y vuelve a comenzar desde la raíz. El fin del archivo se indica al descompresor pre-concatenando al archivo comprimido la longitud del archivo original o bien inventando un caracter nuevo (código 256) que indique el fin de archivo (ver sincronización).

Ejemplo para el fragmento “Asimov”.

> Diez meses antes, el Primer Orador habia contempleado aquellas mismas estrellas, que en ninguna otra parte eran numerosas como en el centro de ese enorma nucleo de materia que el hombre llama la Galaxia, con un sentimiento de duda; pero ahora se reflejaba una sombria satisfaccion en el rostro redondo y rubicundo de Preem Palver, Primer Orador.

Por ejemplo para el fragmento Asimov la frecuencia de aparición de los caracteres fue la siguiente:



Al espacio en blanco y a la letra ‘e’, les corresponderían códigos de tres bits, mientras que a la ‘y’, ‘g’ y otros que solo aparecen una vez les corresponden códigos de 9 bits. En total el archivo se comprime en 197 bytes promediando 4.52 bits por byte.

Uno de los problemas de la compresión por Huffman es la necesidad de guardar y recuperar el árbol utilizado, o la tabla de frecuencias, en el archivo comprimido, lo cual ocupa lugar y además la tarea de construir, guardar y recuperar el árbol insume tiempo, por ello habitualmente se plantean dos variantes: la utilización de árboles canónicos y el empleo de algoritmos dinámicos (o adaptativos). La segunda alternativa es la elegida en casi todos los casos.

•Arboles canónicos.

Esta variante consiste en la utilización de un mismo árbol para comprimir cualquier archivo, a este árbol se lo suele denominar 'árbol canónico'. El árbol canónico puede ser único o variar en función del tipo de archivo a comprimir, se pueden construir por ejemplo árboles canónicos muy eficientes para compresión de textos en castellano o en inglés. Aunque no resultan muy eficientes se pueden usar también árboles canónicos para archivos binarios basándose en estadísticas que por ejemplo dicen que el ASCII 00 aparece muy frecuentemente en archivos binarios.

La utilización de árboles canónicos permite al compresor ahorrarse la tarea de construir el árbol y obtener estadísticas sobre el archivo, utilizando árboles canónicos se puede comprimir por Huffman 'on the fly', al no ser necesario almacenar el árbol, el archivo comprimido necesita menos espacio y el descompresor no necesita recuperarlo.

La desventaja fundamental reside en que los árboles canónicos son eficientes para el tipo de archivo para el cual fueron construidos y es imposible construir árboles canónicos que sirvan para cualquier archivo, el empleo de árboles canónicos solo puede usarse para un tipo de archivo muy determinado ya que de lo contrario, no solo suelen ser ineficientes, sino que incluso suele ser normal que el archivo comprimido sea mayor que el original.

•Huffman adaptativo o dinámico:

La compresión dinámica o adaptativa por Huffman es otra de las técnicas utilizadas para no tener que construir el árbol previamente y además tener que guardarlo en el archivo. Con esta técnica el árbol se va construyendo a medida que se lee el archivo suponiendo en un principio que todos los caracteres son equiprobables, se parte del árbol canónico básico en el cual todos los caracteres tienen frecuencia uno, cuando se lee un caracter se incrementa en uno su frecuencia y se reconstruye el árbol y así sucesivamente.

De esta forma tanto el compresor como el descompresor se evitan recabar las estadísticas del archivo con anterioridad, ya que lo harán a medida que comprimen, obviamente esta técnica es menos eficiente que la estática pues las estadísticas comienzan con una muestra muy pequeña, el nivel de compresión aumenta a medida que se leen mas caracteres del archivo, pero la estadística completa solo se obtiene cuando el archivo ya ha sido procesado y no queda nada por comprimir!.

Básicamente cualquier algoritmo de compresión puede clasificarse en tres categorías según como se procesa el archivo de entrada.

***Algoritmos estáticos**: Leen una vez el archivo para recolectar estadísticas y luego otra vez para comprimirlo.

***Algoritmos semi-estáticos**: Levantan en memoria un bloque del archivo, lo procesan en forma estática y luego graban el bloque comprimido. Solo hacen una lectura del archivo original pero cada bloque es procesado dos veces (en memoria).

***Algoritmos dinámicos**: Leen una vez el archivo de entrada y lo van comprimiendo a medida que lo leen.

Aunque resulta obviamente mas ineficiente que el algoritmo estático la compresión por Huffman dinámico es sumamente utilizada, fundamentalmente por sus ventajas al ir

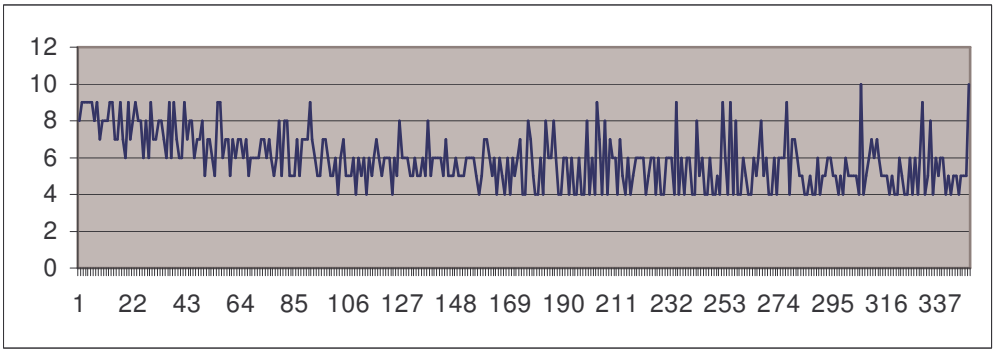
generando el árbol a medida que comprime, la compresión por Huffman dinámico casi siempre es mejor que la utilización de árboles canónicos.

La gran mayoría de los compresores comerciales utilizan la compresión por Huffman en sus algoritmos combinándola generalmente con algún algoritmo tipo LZSS. Muchos de ellos utilizan Huffman dinámico debido a que comprimen por bloques, esto se explicará mas adelante cuando se trate la compresión por bloques.

En el siguiente gráfico se muestra la cantidad de bits emitidos de un compresor Huffman adaptativo por símbolo para el fragmento “Asimov” .

> Diez meses antes, el Primer Orador habia contempleado aquellas mismas estrellas, que en ninguna otra parte eran numerosas como en el centro de ese enorma nucleo de materia que el hombre llama la Galaxia, con un sentimiento de duda; pero ahora se reflejaba una sombria satisfaccion en el rostro redondo y rubicundo de Preem Palver, Primer Orador.

Huffman Adaptativo.



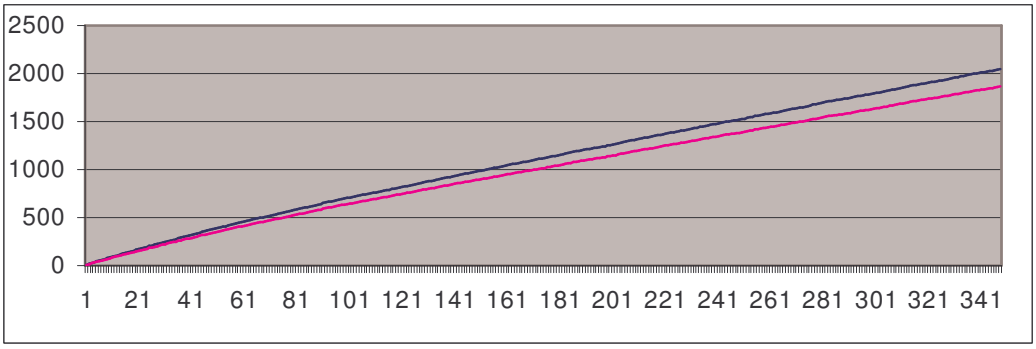
Bits emitidos en función de la cantidad de caracteres procesados para “Asimov”

Como puede verse hay una tendencia descendente en cuanto a la cantidad de bits emitidos a medida que el algoritmo aprende. Durante el proceso los caracteres nuevos causan picos de emisión de hasta 10 bits.

En total el método de Huffman adaptativo emite 2050 bits para el fragmento “Asimov” promediando 5,89 bits por byte contra los 4.52 bpb del método estático podría afirmarse que la compresión adaptativa es ineficiente, pero si consideramos el ahorro insumido al no guardar el árbol y el tiempo ahorrado al procesar una única vez el archivo podemos entender que la compresión adaptativa a veces es necesaria y hasta algunas veces obligatoria.

A continuación mostramos un gráfico que muestra el tamaño total del archivo comprimido para el fragmento “Asimov” a medida que se procesaba el archivo.

Tamaño del archivo comprimido para el fragmento Asimov.
Comprimido usando: Huffman adaptativo y compresión aritmética adaptativa de orden cero.



Longitud del archivo comprimido en función de la cantidad de caracteres procesados.

La línea que corre por debajo del algoritmo de Huffman corresponde a un algoritmo aritmético adaptativo de orden cero y sirve como punto de comparación. Como puede verse a medida que el algoritmo ‘aprende’ la curva crece menos y tiende a hacerse logarítmica, lo cual es lógico si pensamos que el tamaño final de todo el archivo es –log2(P1*P2*P3*...*Pn).

Para descomprimir el descompresor hace exactamente lo mismo que el algoritmo que comprime, comienza suponiendo que todos los caracteres son equiprobables, descomprime el primer caracter, incrementa la frecuencia del caracter leído y reconstruye el árbol, el siguiente caracter lo descomprime con el árbol nuevo y así sucesivamente. Si se quiere hacer que el algoritmo sea mas rápido se puede reconstruir el árbol cada n bytes (n>1). De esta forma por cada caracter leído solo se incrementa su frecuencia y se descomprime usando el árbol anterior, cuando se leyeron “n” caracteres se reconstruye el árbol en función de las frecuencias y se sigue comprimiendo, evidentemente el descompresor tiene que saber cual es la forma en la que se comprimió y luego solo le resta imitar el comportamiento del compresor.

Alcance.

“El método de Huffman mas que un algoritmo de compresión es una herramienta para construir otros algoritmos de compresión.”

El algoritmo de Huffman es un verdadero clásico de la compresión de datos. En cuanto a nivel de compresión un algoritmo que utilice un Huffman simple, hoy en día es obsoleto ya que el nivel de compresión del algoritmo es en promedio de 5 bits/byte y existen algoritmos que actualmente alcanzan alrededor de 2 bits por byte. El método, sin embargo, sigue usándose con asiduidad en algoritmos que utilizan, como parte de un algoritmo de compresión mas complejo, a un compresor estadístico. Dentro de este campo Huffman compite con los compresores aritméticos de orden cero, frente a éstos Huffman tiene un nivel de compresión un tanto menor pero es mucho mas rápido, consume menos recursos y es mas fácil de implementar. Siempre que necesiten un compresor estadístico de orden cero no duden en usar Huffman.

Hemos presentado un algoritmo clásico, hoy ya obsoleto como método de compresión independiente pero integrante casi-ireemplazable de muchos de los algoritmos de compresión mas eficientes de la actualidad.

Versiones:

C.Shannon, T.Fano. (1954).

Se denomina de esta forma a una técnica desarrollada por Shannon y Fano para la construcción del árbol binario a partir del cual se comprimirá un archivo partiendo de la estadística sobre la frecuencia de aparición de cada caracter en el archivo.

Cabe destacar que los árboles de Shannon-Fano no constituyen por si mismos un algoritmo nuevo de compresión sino que simplemente indican otra forma para la construcción del árbol con el cual se comprimirá el archivo, la única diferencia entre la compresión por árboles de Huffman o Shannon-Fano es la forma en la cual se construye el árbol.

A diferencia del árbol generado por Huffman el de Shannon-Fano puede no ser óptimo, la ventaja de los árboles de Shannon-Fano por sobre los de Huffman reside en que son mas fáciles de implementar algorítmicamente.

El esquema es el siguiente:

Se obtiene la frecuencia de aparición de cada caracter en el archivo, todos los caracteres se agrupan en un único conjunto de n caracteres, a continuación y mientras queden conjuntos de mas de un caracter se realiza una partición de cada conjunto de forma tal que cada partición tenga una frecuencia acumulada semejante.

Ejemplo:

Para el mismo archivo que Huffman las frecuencias eran:

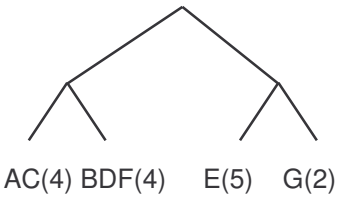
A	3
B	2
C	1
D	1
E	5
F	1
G	2

El conjunto que se forma es:

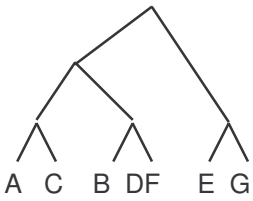
ABCDEFG(15)

Como el total es 15 particionamos en 8 y 7.

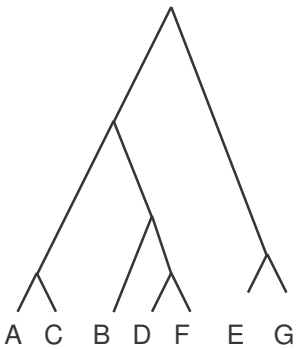
Partimos el primer conjunto en 4-4 y el segundo en 5-2 (única partición posible)



AC lo partimos en 3-1 y BDF en 2-2



Finalmente particionamos DF que es el único conjunto de mas de 1 caracter.



Con lo cual queda formado el árbol.

Los códigos son:

Char	Freq	Código
A	3	000
B	2	010
C	1	001
D	1	0110
E	5	10
F	1	0111
G	2	11

El archivo comprimido ocuparía:
3x2 + 2x3 + 1x3 + 1x4 + 5x2 + 1x4 + 2x2 = 37 bits.

Como se puede observar ocupa 4 bits mas que el archivo comprimido utilizando el árbol generado por Huffman, la ventaja de la técnica de Shannon-Fano reside en que es sumamente sencillo implementar el algoritmo que construye el árbol, el algoritmo para construir el árbol de Huffman es mas complejo.

Al igual que en Huffman los árboles de Shannon-Fano pueden construirse en forma adaptativa, no tiene sentido hablar de árboles canónicos de Shannon-Fano pues Shannon-Fano es una técnica para construir el árbol (!).

Otra variante:

La técnica para hacer el split de un nodo en Shannon-Fano puede diferir.

Originalmente se pretende subdividir el nodo en dos nodos de frecuencias acumuladas similares, sin embargo as frecuente utilizar otro esquema mucho mas 'algorítmico' que consiste en:

Si se quiere hacer el split de S0 se divide a S0 en S01 y S02, originalmente S02=S0

Luego mientras el peso de S01 sea menor que el de S02 se toma el caracter de mayor frecuencia de S02 y se lo pasa a S01.

Este suele ser el algoritmo empleado en todas las implementaciones de árboles de Shannon-Fano, como puede verse la implementación de este algoritmo es casi trivial.

Los árboles de Shannon-Fano son utilizados en la mayoría de los algoritmos del PkZip de PkWare (en combinación con otros métodos) para compresión estadística.

La utilización del método de Shannon-Fano en reemplazo del de Huffman es uno de los motivos por los cuales el Pkzip es uno de los algoritmos de compresión mas rápidos.

•Almacenamiento de códigos binarios: Implosión.

El objetivo de la compresión estadística es, como antes mencionábamos, conseguir un código binario que represente a cada caracter de forma tal que a los caracteres mas frecuentes en un archivo se le asignan códigos con menor cantidad de bits que a los caracteres menos probables. Vimos también dos técnicas para construir estos códigos, los árboles de Huffman y los árboles de Shannon-Fano, el objetivo de ambos métodos era el mismo variando únicamente la forma en la cual se construía el árbol, se hizo hincapié, también, en que los códigos generados por Huffman siempre eran óptimos mientras que los generados por Shannon-Fano podían no serlo.

En los compresores estadísticos estáticos dijimos que era necesario almacenar el árbol usado para comprimir en el archivo comprimido de forma tal que el descompresor pudiera recuperar el árbol antes de empezar a descomprimir, pero esto no es del todo cierto.

En realidad no hay que almacenar el árbol sino los códigos que corresponden a cada caracter y que se generaron a partir del árbol, conociendo los códigos es sencillo reconstruir el árbol y ni siquiera hace falta saber si cero es izquierda o derecha para reconstruir el árbol.

Profundizando aun mas podemos llegar a notar que tampoco es necesario guardar el código correspondiente a cada caracter pues lo que importa no es el código en sí sino, la longitud en bits de dicho código, el valor del código es irrelevante pues puede alcanzarse el mismo nivel de compresión utilizando códigos distintos pero respetando la longitud de los mismos, de esta forma concluimos en que en el archivo comprimido hay que guardar la longitud en bits del código correspondiente a cada caracter, a continuación describimos uno de los métodos usados para ello.

Se almacenan las longitudes de los códigos ordenadas por caracter, la primera longitud corresponde al ASCII(00), la segunda al ASCII(01) y así sucesivamente. Existe en códigos Huffman una forma de obtener la longitud máxima en bits que puede necesitar un código en función de la cantidad de caracteres a comprimir, para 256 caracteres la longitud máxima es 254 y nunca en un archivo tendremos que comprimir mas de 256 caracteres distintos, por lo que sabemos que la longitud máxima de un código es 254, esto puede representarse con dos nibbles, además resulta bastante habitual que varios caracteres compartan la misma longitud para sus códigos, por ejemplo es bastante probable que los caracteres numéricos tengan asignados códigos de igual longitud. De esto se deduce que además es bueno contar con un método que permita indicar cuando una longitud se repite "n" veces en lugar de usar n nibbles uno por cada longitud, la codificación es pues:

Las longitudes de los códigos se almacenan de a bytes de la forma:

El primer byte indica la cantidad de bytes utilizados para almacenar la tabla de las longitudes de los códigos.

Los restantes bytes se codifican de la forma:

Los primeros 4 bits indican la cantidad de veces que se repite la longitud menos uno (no existe cantidad de veces=0).

Los segundos 8 bits indican la longitud del código menos uno (no hay códigos de cero bits)

De forma tal que 40A indicaría 5 longitudes de 11 bits cada una, para 8 caracteres podríamos tener por ejemplo:

07202001002001103 (hexa)

que se traduce así:

```
07  = 8 bytes ocupados para las longitudes de los códigos.
202 = 3 códigos de tres bits de longitud cada uno.
001 = 1 código de dos bits de longitud cada uno.
002 = 1 código de tres bits de longitud cada uno.
001 = 1 código de dos bits de longitud cada uno.
103 = 2 códigos de 4 bits de longitud cada uno.
```

Notar que es ineficiente guardar códigos chiquitos (para pocos caracteres) de esta forma. Se demuestra que este método empieza a ser eficiente para conjuntos de al menos 48 caracteres.

Volviendo al ejemplo anterior con nuestros 8 caracteres sabemos la longitud de los códigos de cada uno.

"0" = 3 bits
"1" = 3 bits
"2" = 3 bits
"3" = 2 bits
"4" = 3 bits
"5" = 2 bits
"6" = 4 bits
"7" = 4 bits.

A continuación hay que obtener los códigos sabiendo únicamente las longitudes y la cantidad de códigos de cada longitud, esto no puede hacerse en forma trivial pues los códigos deben quedar de forma tal que respondan a la forma de un árbol de Huffman que corresponde a los caracteres del archivo. Si se asignaran trivialmente podría quedar un árbol del cual se obtuvieran en realidad otros códigos.

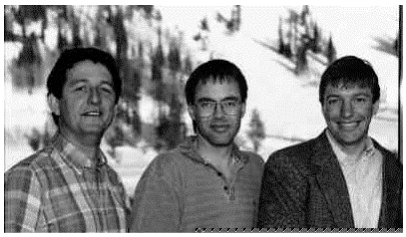
De las longitudes deben obtenerse los códigos, de los códigos el árbol y del árbol se obtienen los mismos códigos con las mismas longitudes, debe ser un camino cerrado.

Este método es en general mas eficiente que el guardar el árbol o bien las frecuencias de los caracteres en el archivo comprimido, la mayoría de los algoritmos estáticos almacena en el archivo comprimido la longitud de los códigos y no las frecuencias de los caracteres o bien los códigos mismos.

Compresión Aritmética.

Versiones:

I.E.Witten. (1987)
R.Neal, J.Cleary. (1987)
Patentes varias de IBM.



Witten, Moffat y Bell. Gurúes de la compresión aritmética.

Como vimos anteriormente la teoría de Shannon establece que la longitud ideal del código correspondiente a un cierto caracter es -log2(P(C)) siendo P(C) la probabilidad de ocurrencia de este caracter, los métodos de Huffman y Shannon-Fano mostraban formas de construir estos códigos de forma tal que la longitud de los mismos se aproximara a la ideal, sin embargo, la longitud de los códigos de Huffman y Shannon-Fano debía ser un número entero de bits por lo que cuando -log2(P(C)) no daba un número entero debíamos redondearlo y esto implicaba muchas veces usar mas bits de los necesarios. La compresión aritmética de reciente aparición en el mundo de la computación (1987 aproximadamente) establece la forma de codificar a los caracteres de un archivo con códigos cuya longitud puede ser cualquier número real, de esta forma podemos codificar a cada caracter con su longitud ideal y comprimir un archivo de acuerdo a su entropía, se demuestra que la compresión aritmética es el algoritmo óptimo de compresión estadística.

La compresión aritmética puede encuadrarse dentro de los algoritmos de compresión estadística ya que se basa al igual que estos en un estudio previo de la frecuencia de aparición de cada caracter en el archivo, la diferencia reside en que la compresión aritmética no genera un árbol como la mayoría de los algoritmos de compresión estadística sino que utiliza una técnica completamente distinta.

En la compresión aritmética un archivo es representado por un intervalo de los números reales perteneciente al [0,1). A medida que el archivo se hace mas largo el intervalo necesario para representarlo se hace mas chico y el número de bits necesarios para especificar dicho intervalo se hace mayor. Los caracteres que se van leyendo del archivo reducen el tamaño del intervalo en proporción a la probabilidad de aparición del caracter en el archivo, los caracteres mas frecuentes reducen el intervalo en menor medida que los caracteres menos frecuentes de forma tal que agregan menos bits al archivo comprimido.

Antes de procesar el archivo a comprimir el rango para el archivo es todo el intervalo [0,1) 0<=x<1. A medida que se van procesando los caracteres el rango es reducido a la porción de intervalo que corresponde al caracter leído.

Supongamos que queremos comprimir el mensaje: "abcda" (5 bytes)

Las frecuencias son: Las probabilidades son:

a=2 a=0.4
b=1 b=0.2
c=1 c=0.2
d=1 d=0.2

Int1	Int2	Int3	Int4	Int5
0.0	0.0	0.16	0.2080	0.22080
a	a	a	a	a
0.4	0.16	0.192	0.2144	0.22208
b	b	b	b	b
0.6	0.24	0.208	0.2176	0.22272
c	c	c	c	c
0.8	0.32	0.224	0.2208	0.22336
d	d	d	d	d
1.0	0.40	0.240	0.2240	0.2240

Nuestro primer intervalo es el [0,1) dividido según la figura I1 de acuerdo a la probabilidad de cada caracter.

El primer caracter a comprimir es una "a" por lo que nuestro intervalo se reduce ahora al [0.0,0.4] (intervalo que corresponde a la "a") el cual volvemos a subdividir de la misma forma en que lo hicimos antes(columna Int2).

Subdividimos el [0.0,0.4] usando la misma tabla de frecuencias que usamos para subdividir al [0,1), nos queda el intervalo Int2.

El segundo caracter es una "b" por lo que el nuevo intervalo es el [0.16,0.24] repetimos la subdivisión.

Seguimos achicando el intervalo hasta llegar a la división final, el ultimo caracter es una "a" por lo que para codificar el archivo elegimos un número cualquiera perteneciente al [0.22080,0.22208]

Elegimos el 0.2208 (nota: aquí tomamos siempre la cota inferior del intervalo aunque podemos convenir en cualquier otro número que pertenezca al intervalo, por ejemplo (LI+LS)/2)

El resultado es pues 0.2208 que es el archivo comprimido !!

(Además hay que marcar el end of file de alguna forma o bien indicar cual es el largo del archivo original para que el descompresor sepa cuando detenerse)

Para descomprimir basta con conocer el número y la tabla, con el número 0,22208 y la tabla original sabemos que el primer caracter es una "a" pues 0,22208 pertenece al [0.0,0.4] ahora subdividimos el [0.0,0.4] tal y como lo hicimos al comprimir, volvemos a ver en que intervalo cae el número y conocemos el siguiente caracter, y así sucesivamente hasta encontrar el caracter de end of file.

Los algoritmos de compresión aritmética pueden implementarse también en forma adaptativa, comenzando con una tabla en la cual todos los caracteres se consideran equiprobables, a medida que se leen caracteres del archivo se re-calculan las probabilidades de aparición de cada caracter y el próximo caracter se codifica particionando el intervalo actual de acuerdo a la nueva tabla de probabilidades, el algoritmo descompresor adaptativo tendrá que ir realizando el mismo trabajo a medida que descomprime, de esta forma se logra evitar el tener que incluir la tabla de frecuencias originales en el archivo comprimido ya que esta se puede ir generando a medida que se descomprime, a medida que se va procesando un archivo el algoritmo comprime cada vez mejor ya que adquiere 'entrenamiento'. La mayoría de los compresores aritméticos están implementados en forma adaptativa.

El proceso de comprimir y descomprimir una secuencia de símbolos usando compresión aritmética no es muy complicado, pero a primera vista parece completamente impráctico. La mayoría de las computadoras soportan números de punto flotante de hasta 80 bits aproximadamente, lo cual implicaría empezar todo de nuevo cada 10 o 15 caracteres, además cada máquina implementa los números flotantes de forma distinta.

En conclusión la compresión aritmética es mas eficiente si logra ser implementada usando aritmética estándar de enteros de 16 o 32 bits, la gran mayoría de los compresores aritméticos no utilizan números de punto flotante ni serían beneficiados por usarlos. Mas adelante veremos como implementar la compresión aritmética usando números enteros mediante una técnica denominada renormalización.

A primera vista se podría suponer que la probabilidad de aparición de un caracter en el archivo es un único número fijo entre 0 y 1, sin embargo esto no es completamente cierto. Dependiendo del modelo que esta siendo utilizado la probabilidad de cada símbolo puede cambiar levemente. Por ejemplo al comprimir un programa fuente escrito en leguaje "C" la probabilidad de encontrar un caracter de fin de línea en el archivo podría ser aproximadamente 1/40 (esto se obtiene dividiendo la cantidad de caracteres de fin de línea del archivo por la cantidad total de caracteres del archivo) pero si usáramos un modelo probabilístico que considerara también cual fue el caracter previo las probabilidades cambiarían, para nuestro caso si el caracter previo fue un ";" la probabilidad de encontrar un fin de línea asciende a 1/2. De esta forma se observa que un modelo probabilístico que solo considera las probabilidades de aparición de cada caracter en el archivo es inferior a un modelo que considera la probabilidad de aparición de cada caracter en función del caracter anterior.

•Modelos probabilísticos de contexto finito.

El tipo de modelo utilizado por los compresores aritméticos se denomina de contexto finito. Esta basado en una idea muy simple: las probabilidades de aparición de cada símbolo a procesar se calculan basándose en el contexto en el cual aparece el símbolo.

El contexto esta dado por los n caracteres anteriormente leídos del archivo.

El modelo de contexto finito mas simple se conoce como modelo de orden-0, en el cual la probabilidad de aparición de cada símbolo es independiente de los símbolos previos, se consideran solo las frecuencias de aparición de cada caracter en el archivo, para este modelo necesitamos solamente una tabla de 256 posiciones, una para cada caracter que pueda ser encontrado en el archivo.

En un modelo de orden-1 tomamos la probabilidad de aparición de cada símbolo en función de cual fue el símbolo anterior. (¿cual es la probabilidad de ocurrencia de una "u" si antes leímos una "z"?) Para este modelo necesitamos 256 tablas de 256 posiciones cada una, es decir que para cada caracter debemos considerar su probabilidad de aparición en función de todos los demás caracteres.

Para un modelo de orden 2 en el cual consideramos la probabilidad de aparición de un símbolo en función de los dos símbolos previos necesitamos 256x256 tablas de 256 caracteres cada una, a medida que el orden del modelo aumenta el nivel de compresión se hace cada vez mayor. Por ejemplo la probabilidad de la letra "u" en este texto puede que sea solo de un 5% pero, si la letra previa fue una "q", la probabilidad podría aumentar a un 99% (!!) La capacidad de predecir caracteres con probabilidades altas disminuye la cantidad de bits necesarios para representarlos y los modelos de mayor contexto nos permiten hacer mejores predicciones. Recordemos que la compresión aritmética comprime caracteres en función de sus probabilidades de ocurrencia usando la longitud optima en bits de acuerdo a la entropía, de esta forma lo único que debemos hacer es mejorar el modelo probabilístico para poder comprimir mejor.

Desafortunadamente a medida que el orden del modelo aumenta en forma lineal la memoria necesaria para las tablas aumenta exponencialmente. Para un modelo de orden-0 necesitamos 256 bytes, pero una vez que el orden del modelo aumenta a 2 o 3 hasta los diseños mas brillantes necesitaran cientos y cientos de kilobytes.

Si un modelo de orden superior es además adaptativo tenemos que enfrentar el costo de ir actualizando el modelo probabilístico, al incrementar la probabilidad de un determinado caracter tenemos que enfrentarnos al costo potencial de tener que actualizar las probabilidades de todos los demás caracteres además del leído, en promedio necesitaremos unas 128 operaciones aritméticas para cada símbolo codificado o decodificado además de las necesarias para la subdivisión del intervalo.

Debido al alto costo en memoria y CPU los modelos adaptativos de orden superior se han hecho prácticos solo en los últimos diez años. Resulta curioso pero a medida que el costo de los discos y la memoria disminuye también disminuye el costo de comprimir los datos que allí se almacenan, a medida que estos costos disminuyan aun mas podremos implementar algoritmos aun mas efectivos que los que son prácticos hoy en día.

Frecuentemente los compresores aritméticos 'switchean' el orden del modelo a medida que comprimen de forma de aumentar la velocidad de compresión, el algoritmo va comprobando la velocidad de compresión en función del nivel de compresión y trata de mantenerse dentro de límites predeterminados, para ello puede que sea necesario disminuir o aumentar el orden del modelo que esta siendo utilizado, de esta forma los algoritmos mas avanzados utilizan modelos adaptativos y de orden variable, el rendimiento del algoritmo puede parametrizarse en función de la velocidad o nivel de compresión que se desee, un algoritmo que quiera ser veloz solo sube de orden cuando el nivel de compresión decae mucho mientras que los algoritmos 'trituradores' solo bajan de orden cuando la velocidad se hace intolerable. Notemos que si bien estos algoritmos son de orden variable comprimen a cada caracter usando un orden fijo, si estamos en orden-3 el próximo caracter se comprime usando orden-3 sea cual fuera dicho caracter, luego se analiza el rendimiento y se elige un nuevo orden, usando códigos especiales se informa al descompresor de los cambios de orden.

Entre los esquemas de compresión estadística Huffman es óptimo si cada caracter debe representarse con un número fijo de bits, sin esta restricción la compresión aritmética es superior. Los algoritmos de compresión aritmética mas avanzados superan ampliamente en nivel de compresión a todos los algoritmos de compresión utilizados habitualmente. La única desventaja de los compresores aritméticos reside en la necesidad de mucha memoria y un CPU veloz para funcionar correctamente, a medida que las máquinas tienen cada vez mas memoria y son mas rápidas los compresores aritméticos se hacen cada vez mas eficientes. Hay que tener en cuenta que un compresor por Huffman nunca podrá comprimir mejor si el hardware mejora mientras que un compresor aritmético aumenta mas y mas su rendimiento a medida que dispone de mas recursos.

Los compresores aritméticos actuales son en general productos de investigación surgidos de universidades, y funcionan bastante bien, de los comerciales solo el HAP utiliza compresión aritmética (modelo de orden variable hasta 3), HAP obviamente es un tanto mas lento que la competencia pero comprime mucho mas, el algoritmo de compresión mas eficiente hasta la fecha es el PPMC, que usa compresión aritmética, mas adelante desarrollaremos la técnica PPM.

•Compresión aritmética usando aritmética de enteros.

•Renormalización.

El esquema de compresión aritmética descripto funciona perfectamente bien resultando óptimo para compresión estadística, sin embargo no aparece a simple vista una forma de implementar esta técnica en una computadora sin usar números de punto flotante, aún usando números de punto flotante la precisión de los mismos nunca podrá ser suficiente como para poder comprimir cualquier archivo. Para evitar el uso de números de punto flotante se han desarrollado técnicas que implementan la compresión aritmética utilizando únicamente números enteros. Ilustraremos el método usando números en base 10 pero debemos mencionar que en las implementaciones utilizadas se emplean como es obvio números binarios, la técnica es la misma independientemente de la base utilizada.

El algoritmo de compresión debe mantener dos valores enteros que se corresponden al limite inferior y limite superior del intervalo actual, en el comienzo el intervalo es el [0,1) por lo que el limite inferior será 00000 y el superior 99999 (suponemos que trabajamos con decimales de cinco dígitos y que XXXXX representa 0.XXXXX) de forma tal que 99999 es en realidad 0.99999... que es igual a 1 (0,9 periódico es 1)) debemos recordar entonces que el valor del limite superior continua con nueves hacia la derecha infinitas veces. A los limites inferior y superior los llamaremos Piso y techo.

Piso =00000
Techo=99999

A continuación debemos leer un caracter del archivo y actualizar el intervalo de la forma:

Rango = Techo - Piso
Nuevo Techo = Piso + rango * techo(símbolo)
Nuevo Piso = Piso + rango * piso(símbolo)

La forma en la cual se calculan el rango y los valores piso y techo depende de la implementación, aquí mostramos la forma mas sencilla, las fórmulas mas eficientes por su velocidad de cálculo son las de Rissanen-Mohiuddin descubiertas recientemente (en 1995).

Supongamos que leímos una letra "B" y que en la tabla de probabilidades vemos que "B" pertenece al [0.2,0.3) - cabe aclarar que el techo y piso de los distintos símbolos se calcula siempre sobre el intervalo [0,1) -

Rango = 99999 - 00000 = 100000 (sumamos 1 pues el 99999 era periódico)
Nuevo Techo =00000 +100000*0.3 =30000 =29999 (restamos 1 pues es 29999...)
Nuevo Piso = 00000 + 100000 * 0.2 = 20000

Piso = 20000
Techo = 29999

A continuación deberíamos repetir el proceso pero antes observemos un detalle: cuando los primeros dígitos de los valores Piso y Techo son iguales podemos demostrar que los caracteres que se lean a continuación no van a cambiar estos dígitos, es decir que el primer dígito decimal del número que representara al archivo comprimido será un "2" de esta forma podemos escribir el "2" y correr Piso y Techo hacia la izquierda (shifts)

Piso = 00000 (completamos con ceros el piso)
Techo = 99999 (completamos con nueves el techo)
número = 0,2... (ya conocemos el primer dígito)

Supongamos que ahora leemos ahora una “C”, cuyo intervalo –siempre en base al intervalo [0,1)- es el [0.35 , 0.48). Aplicando las fórmulas obtenemos :

Rango = 99999 - 00000 = 100000 (sumamos 1 pues el 999999 era periódico)
Nuevo Techo =00000 +100000*0.48 = 48000 = 47999 (restamos 1 pues es 29999...)
Nuevo Piso = 00000 + 100000 * 0.35 = 35000

Piso = 35000
Techo = 47999
número = 0,2... (seguimos conociendo solo el primer dígito, ya que los primeros no coinciden aún)

De esta forma continuamos leyendo otro caracter, actualizando piso y techo, y emitiendo otro dígito cuando coincidan los primeros dígitos de Piso y Techo, así hasta terminar de comprimir el archivo.

Problemas:

Este esquema funciona bien pero puede llegar a tener un problema: el underflow. Supongamos que en algún momento de nuestra compresión nos quedara techo=70004 piso=69995, cuando leamos el próximo caracter puede que piso y techo no se modifiquen, es decir que empezamos a perder precisión y no podemos emitir ningún dígito pues los primeros dígitos de piso y techo no coinciden, este es un problema numérico denominado underflow.

Para solucionar los posibles casos de underflow se recurre a la renormalización, la renormalización implica realizar un chequeo sobre los valores de piso y techo antes de continuar comprimiendo de forma tal que evitemos el underflow, el chequeo consiste en verificar si los primeros dígitos de piso y techo difieren en una unidad. Si así fuera se verifica que los segundos dígitos de piso y techo sean 9 y 0 respectivamente entonces podría producirse underflow, para evitarlo lo que se hace es eliminar los segundos dígitos de piso y techo, correr los demás dígitos a la derecha e incrementar el contador de underflow.

Ejemplo:

Techo = 40344 -> 43449
Piso = 39810 -> 38100
Under = 0 -> 1

Y continuamos con el esquema anterior, cuando los primeros dígitos de piso y techo coincidan debemos ver si a que valor convergieron, si han convergido al valor del techo ("4" en el ejemplo) entonces debemos emitir el valor ("4") y tantos ceros como nos indique el contador de underflow. Si el valor al cual convergen fue el de piso ("3" en el ejemplo) emitimos el valor del piso("3") y tantos nueves como nos indique el contador de underflow.

Este esquema de compresión aritmética usando aritmética de enteros mas renormalización es el utilizado por todas las implementaciones de compresores aritméticos.

Versiones:

Witten, Moffat, Bell. (1988)

El algoritmo PPM es la versión mas avanzada de los algoritmos de compresión aritmética, la idea básica es usar un modelo probabilístico que no utilice un único contexto para predecir cada caracter sino que utilice varios contextos al mismo tiempo de forma tal que la predicción sea lo mas acertada posible.

PPM utiliza contextos de varios ordenes hasta un cierto límite, por ejemplo orden-0 hasta orden-6, cada caracter intenta ser comprimido utilizando el máximo orden posible.

El algoritmo PPM es un algoritmo de compresión adaptativa, para empezar analicemos algunas desventajas de los modelos adaptativos utilizados hasta el momento.

Los modelos de orden superior como hemos mencionado deben mantener la probabilidad de ocurrencia de cada caracter en un determinado contexto, al comenzar seria lógico suponer que la probabilidad de los caracteres se supone equiprobable para todos los contextos y que estas probabilidades se incrementan a medida que se procesa el archivo, pero resulta que este esquema es poco eficiente. Supongamos que en un modelo de orden-2 los dos últimos caracteres leídos fueron "SQ" y que leemos del archivo el caracter "U" (estamos en el contexto "SQ") la probabilidad del caracter "U" en este contexto debería ser alta, sin embargo si comenzamos suponiendo que todos los caracteres son equiprobables tenemos que al principio la probabilidad de la "U" será de 1/257 (256 caracteres mas el caracter especial de EOF) con lo cual cuando leamos 10 veces la letra "U" en el contexto "SQ" su probabilidad será de 11/267, cuando sin embargo debería tener probabilidad "1", como vemos inicializar a todos los caracteres en forma equiprobable es muy ineficiente, debemos, entonces inicializar a todos los caracteres en probabilidad cero.

Una vez dentro del contexto "SQ" al leer una "U" nos encontraremos con la desagradable sorpresa de que todos los caracteres tienen probabilidad cero, con lo cual no podemos seguir comprimiendo el archivo (!), este tipo de problema se denomina problema de frecuencia cero (Zero frequency problem) y es muy común en el campo de la compresión de datos.

La solución utilizada consiste en inventar un caracter especial de "Escape", tenemos pues 258 caracteres posibles (los 256 ASCII, el EOF y el de Escape) el caracter de escape se inicializa con probabilidad 1 en todos los contextos mientras que el resto de los caracteres tienen probabilidad cero, entonces:

Supongamos que hemos leído "SQ" y que ahora leemos una "U" en el contexto "SQ" la "U" tiene probabilidad cero por lo que incrementamos su frecuencia (para cuando volvamos a encontarla) y emitimos un código de escape. (El contexto "SQ" queda ahora con probabilidad 1/2 para la "U" y el "Escape" y probabilidad cero para todos los otros caracteres). El emitir un escape implica descender el orden del modelo en una magnitud, estamos ahora pues en el contexto "Q", analizamos la probabilidad de la "U" en el contexto "Q", si es distinta de cero comprimimos, si es cero emitimos otro escape y así sucesivamente.

Notemos que la próxima vez que en el contexto "SQ" encontremos una "U" observaremos que tiene probabilidad 1/2 con lo cual leyendo solo una vez la "U" en "SQ" la comprimimos mucho mas que leyéndola 10 veces si hubiésemos inicializado todo en forma equiprobable (1/2 > 11/267). Como en el comienzo todos los contextos se inicializan en probabilidad cero al leer el primer caracter del archivo deberíamos emitir un escape ,con lo cual pasaríamos de orden-0 a orden menos 1 (!!), esto es correcto, la tabla para orden -1 supone a todos los caracteres equiprobables, de forma tal que con orden -1 tenemos una probabilidad de 1/257 para cada caracter. (no existe escape en el orden -1).

- **PPM**

La idea básica del PPM es usar los últimos caracteres leídos en el archivo de input para predecir cual será el próximo caracter a leer.

Los modelos que calculan la probabilidad de cada caracter en base a los n símbolos anteriores se denominan modelos de contexto finito de orden-n. PPM utiliza un conjunto de modelos de orden finito de distintos ordenes desde cero hasta un cierto máximo para predecir el próximo caracter.

Para cada modelo se debe tomar nota de todos los caracteres que siguieron a cada string de "m" caracteres leído anteriormente y del número de veces que ocurrió cada aparición, las probabilidades de ocurrencia de los caracteres se calculan a partir de estas tablas, de esta forma para cada modelo tenemos probabilidades de ocurrencia distintas para cada caracter, estas probabilidades se combinan en una sola y la compresión aritmética es usada para codificar el caracter en base a esta probabilidad.

- **PPMC (PPM método 'C')**

Cuando el compresor encuentra un caracter que tiene probabilidad cero emite un código de escape que le indica al descompresor que tiene que descender un orden de magnitud para dicho caracter, la forma en la cual se tratan los códigos de escape dentro del modelo probabilístico depende del método de PPM a utilizar, el método “A” por ejemplo asigna frecuencia 1 inicialmente a los escapes y luego aumenta dicha frecuencia a medida que ocurren nuevos escapes. el método mas eficiente es el denominado método "C" que asigna al código de escape una frecuencia igual a la cantidad de caracteres con probabilidad distinta de cero del modelo. Al algoritmo de PPM que utiliza el método "C" se lo denomina PPMC. El algoritmo PPMC es considerado el algoritmo de compresión mas eficiente hasta la fecha.

El funcionamiento del PPMC se puede entender fácilmente con un ejemplo:

Supongamos que queremos comprimir el string "abracadabra" con PPMC de orden máximo=2

- Inicializamos el modelo de orden -1 con probabilidad 1/257 para todos los caracteres. Este modelo no se actualiza nunca.
- Inicializamos los modelos de orden-0 hasta orden-2 con probabilidad cero para todos los caracteres, excepto el escape (ESC) que se inicializa con probabilidad 1.

Estado inicial.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	0						
	b	1/257		b	0						
	c	1/257		c	0						
	d	1/257		d	0						
	r	1/257		r	0						
				Esc	1						

- Leemos "a"

- Empezamos con contexto = "" es decir orden-0
- Leemos el caracter "a" que tiene probabilidad cero en el modelo de orden-0 (si no existe el caracter entonces tiene probabilidad cero), emitimos un escape, actualizamos la frecuencia de la "A" en el modelo de orden-0 a 1 y actualizamos la frecuencia del escape a la cantidad de símbolos con probabilidad distinta de cero, es decir sigue siendo 1, y pasamos al orden -1.
- La probabilidad de "a" en el modelo de orden -1 es 1/257, comprimimos "a" con probabilidades 1*1/257= 1/257 lo cual implica emitir –log2(1/257)=8,005 bits. El modelo –1 no se actualiza.

Las tablas quedan de la forma:

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	½						
	b	1/257		b	0						
	c	1/257		c	0						
	d	1/257		d	0						
	r	1/257		r	0						
				Esc	½						

Contexto “a”

- Leemos “b”

- Nuestro contexto es ahora "a", es decir que comenzamos por el orden 1.
- Leemos la "b" que tiene probabilidad cero en el modelo de orden 1 con contexto “a”, emitimos un escape, actualizamos las frecuencias de la "b" y el "escape" en el modelo de orden "1" y pasamos al modelo de orden-0.
- La "b" tiene probabilidad cero en el modelo de orden-0, emitimos otro escape, actualizamos la probabilidad de la "b" y el escape en el modelo de orden-0 y pasamos al modelo de orden -1.
- Finalmente en el modelo de orden -1 comprimimos "b" con probabilidad 1/257.

Las probabilidades usadas fueron 1*1/2*1/257

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	1/4	a	b	1/2			
	b	1/257		b	1/4	a	Esc	1/2			
	c	1/257		c	0						
	d	1/257		d	0						
	r	1/257		r	0						
				Esc	2/4						

Contexto "**ab**"

- Leemos "r"

- Estamos en el modelo de orden 2 con contexto "ab", allí la r no existe por lo que emitimos un escape de probabilidad 1 y bajamos al modelo de orden 1, luego de actualizar la frecuencia de la "r" y el Esc en el modelo de orden 2.
- Con orden 1 el contexto es "b", allí la "r" no existe por lo que emitimos otro escape de probabilidad 1, actualizamos las frecuencias y pasamos al modelo de orden 0.
- En el modelo de orden 0 la "r" no existe, emitimos un escape de probabilidad 2/4, actualizamos las tablas correspondientes en el modelo y pasamos al modelo de orden –1.
- En el modelo de orden –1 comprimimos la "r" con probabilidad 1/257.

En total la probabilidad de la "r" fue 1*1*2/4*1/257.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	1/6	a	b	1/2	ab	r	1/2
	b	1/257		b	1/6	a	Esc	1/2	ab	Esc	1/2
	c	1/257		c	0	b	r	1/2			
	d	1/257		d	0	b	Esc	1/2			
	r	1/257		r	1/6						
				Esc	3/6						

Contexto: "a**br**"

Leemos "a".

- En el modelo de orden 2 con contexto "br" la "a" no existe por lo que se emite un escape de probabilidad 1, se actualizan las frecuencias y se pasa al modelo de orden 1 con contexto "r".
- En el modelo de orden 1 para el contexto "r" la "a" no existe por lo que emitimos otro escape de probabilidad 1, actualizamos las frecuencias y pasamos al modelo de orden 0, en donde si se encuentra a la "a" con probabilidad 1/6, por lo que comprimimos la "a" y actualizamos las frecuencias

La probabilidad de la "a" fue 1*1*1/6.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	2/7	a	b	1/2	ab	r	1/2
	b	1/257		b	1/7	a	Esc	1/2	ab	Esc	1/2
	c	1/257		c	0	b	r	1/2	br	a	1/2
	d	1/257		d	0	b	Esc	1/2	br	Esc	1/2
	r	1/257		r	1/7	r	a	1/2			
				Esc	3/7	r	Esc	1/2			

Contexto: "ab**ra**"

Leemos "c".

- En el modelo de orden 2 con contexto "ra", la c no existe, tampoco en el modelo de orden 1 con contexto "a", ni tampoco en el modelo de orden 0 por lo tanto emitimos 3 escapes de probabilidad 1,1/2 y 3/7 respectivamente, actualizamos todas las frecuencias y comprimimos la "c" con probabilidad 1/257 en el modelo -1.

Probabilidad de la "c": 1*1/2*3/7*1/257.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	2/9	a	b	1/4	ab	r	1/2
	b	1/257		b	1/9	a	c	1/4	ab	Esc	1/2
	c	1/257		c	1/9	a	Esc	2/4	br	a	1/2
	d	1/257		d	0	b	r	1/2	br	Esc	1/2
	r	1/257		r	1/9	b	Esc	1/2	ra	c	1/2
				Esc	4/9	r	a	1/2	ra	Esc	1/2
						r	Esc	1/2			

Contexto: "abr**ac**"

Leemos "a".

- En el modelo de orden 2 con contexto "ac" la "a" no existe por lo que se emite un escape de probabilidad 1, actualizamos y pasamos al modelo de orden 1.
- Con contexto "c", la "a" no existe en el modelo de orden 1 por lo que emitimos un escape de probabilidad 1, actualizamos y pasamos al modelo de orden 0.
- En el modelo de orden cero encontramos la "a" con probabilidad 2/9, la emitimos y actualizamos.

Probabilidad de la "a": 1*1*2/9.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	3/10	a	b	1/4	ab	r	1/2
	b	1/257		b	1/10	a	c	1/4	ab	Esc	1/2
	c	1/257		c	1/10	a	Esc	2/4	br	a	1/2
	d	1/257		r	1/10	b	r	1/2	br	Esc	1/2
	r	1/257		d	0	b	Esc	1/2	ra	c	1/2
				Esc	4/10	r	a	1/2	ra	Esc	1/2
						r	Esc	1/2	ac	a	1/2
						c	a	1/2	ac	Esc	1/2
						c	Esc	1/2			

Contexto: ‘abra**ca**”

Leemos “d”.

- Con contexto “ca” en el modelo de orden 2 la “d” no existe por lo que pasamos al modelo de orden 1 emitiendo un escape de probabilidad 1.
- En el modelo de orden 1 con contexto “a” la “d” no existe por lo que emitimos un escape, en este caso de probabilidad 2/4. En el modelo de orden 0 la “d” no existe por lo que se emite un escape de probabilidad 4/10 y pasamos al modelo de orden –1 donde comprimimos la “d” con probabilidad 1/257.

Probabilidad de la “d”: 1*2/4*4/10*1/257

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	3/12	a	b	1/6	ab	r	1/2
	b	1/257		b	1/12	a	c	1/6	ab	E	1/2
	c	1/257		c	1/12	a	d	1/6	br	a	1/2
	d	1/257		d	1/12	a	E	3/6	br	E	1/2
	r	1/257		r	1/12	b	r	1/2	ra	c	1/2
				Esc	5/12	b	Esc	1/2	ra	Esc	1/2
						r	a	1/2	ac	a	1/2
						r	Esc	1/2	ac	Esc	1/2
						c	a	1/2	ca	d	1/2
						c	Esc	1/2	ca	Esc	1/2

Contexto: ‘abrac**ad**”

Leemos “a”

- Con contexto “ad” la “a” no existe por lo que emitimos un escape de probabilidad 1.
- En el modelo de orden 1 con contexto “d” la “a” no existe por lo que emitimos otro escape de probabilidad 1.
- En el modelo de orden 0 encontramos la “a” con probabilidad 3/12. Emitimos el código correspondiente y actualizamos los modelos utilizados.

Probabilidad de la “a” 1*1*3/12

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	4/13	a	b	1/6	ab	r	1/2
	b	1/257		b	1/13	a	c	1/6	ab	Esc	1/2
	c	1/257		c	1/13	a	d	1/6	br	a	1/2
	d	1/257		d	1/13	a	Esc	3/6	br	Esc	1/2
	r	1/257		r	1/13	b	r	1/2	ra	c	1/2
				Esc	5/13	b	Esc	1/2	ra	Esc	1/2
						r	a	1/2	ac	a	1/2
						r	Esc	1/2	ac	Esc	1/2
						c	a	1/2	ca	d	1/2
						c	Esc	1/2	ca	Esc	1/2
						d	a	1/2	ad	a	1/2
						d	Esc	1/2	ad	Esc	1/2

Contexto: ‘abraca**da**”

Leemos “b”

- En el modelo de orden 2 la “b” no existe en el contexto “da” por lo que emitimos un escape de probabilidad 1 y pasamos al modelo de orden 1, actualizando previamente las frecuencias..
- En el modelo de orden 1 con contexto “a” la “b” tiene probabilidad 1/6 por lo que comprimimos la “b” y actualizamos las frecuencias. Los modelos no utilizados no se modifican.

Probabilidad de la “b”: 1*1/6

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	4/13	a	b	2/7	ab	r	1/2
	b	1/257		b	1/13	a	c	1/7	ab	Esc	1/2
	c	1/257		c	1/13	a	d	1/7	br	a	1/2
	d	1/257		d	1/13	a	Esc	3/7	br	Esc	1/2
	r	1/257		r	1/13	b	r	1/2	ra	c	1/2
				Esc	5/13	b	Esc	1/2	ra	Esc	1/2
						r	a	1/2	ac	a	1/2
						r	Esc	1/2	ac	Esc	1/2
						c	a	1/2	ca	d	1/2
						c	Esc	1/2	ca	Esc	1/2
						d	a	1/2	ad	a	1/2
						d	Esc	1/2	ad	Esc	1/2
									da	b	1/2
									da	Esc	1/2

Contexto: ‘abracad**ab**”

Leemos “r”.

- En el modelo de orden 2 encontramos la “r” con contexto “ab” por lo que comprimimos la “r” con probabilidad 1/2 (un solo bit!) y actualizamos las tablas de frecuencias.

Probabilidad de la ‘”r” 1/2.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	4/13	a	b	2/7	ab	r	2/3
	b	1/257		b	1/13	a	c	1/7	ab	Esc	1/3
	c	1/257		c	1/13	a	d	1/7	br	a	1/2
	d	1/257		d	1/13	a	Esc	3/7	br	Esc	1/2
	r	1/257		r	1/13	b	r	1/2	ra	c	1/2
				Esc	5/13	b	Esc	1/2	ra	Esc	1/2
						r	a	1/2	ac	a	1/2
						r	Esc	1/2	ac	Esc	1/2
						c	a	1/2	ca	d	1/2
						c	Esc	1/2	ca	Esc	1/2
						d	a	1/2	ad	a	1/2
						d	Esc	1/2	ad	Esc	1/2
									da	b	1/2
									da	Esc	1/2

Contexto: ‘abracada**br**”

Leemos “a”

- La “a” también es encontrada en el modelo de orden 2, con contexto “br”. Por lo que se comprime con probabilidad 1/2 y se actualiza solo este modelo.

Probabilidad de la “a”: 1/2.

Modelo –1			Modelo 0			Modelo 1			Modelo 2		
Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P	Ctx	Ch	P
	a	1/257		a	4/13	a	b	2/7	ab	r	2/3
	b	1/257		b	1/13	a	c	1/7	ab	Esc	1/3
	c	1/257		c	1/13	a	d	1/7	br	a	2/3
	d	1/257		d	1/13	a	Esc	3/7	br	Esc	1/3
	r	1/257		r	1/13	b	r	1/2	ra	c	1/2
				Esc	5/13	b	Esc	1/2	ra	Esc	1/2
						r	a	1/2	ac	a	1/2
						r	Esc	1/2	ac	Esc	1/2
						c	a	1/2	ca	d	1/2
						c	Esc	1/2	ca	Esc	1/2
						d	a	1/2	ad	a	1/2
						d	Esc	1/2	ad	Esc	1/2
									da	b	1/2
									da	Esc	1/2

Contexto: ‘abracadab**ra**”

Al terminar de comprimir "abracadabra" los modelos quedan de la forma:

MODELO DE ORDEN 2

contexto	char	freq	prob
ab	r	2	2/3
ab	ESC	1	1/3
ac	a	1	1/2
ac	ESC	1	1/2
ad	a	1	1/2
ad	ESC	1	1/2
br	a	2	2/3
br	ESC	1	1/3
ca	d	1	1/2
ca	ESC	1	1/2
da	b	1	1/2
da	ESC	1	1/2
ra	c	1	1/2
ra	ESC	1	1/2

MODELO DE ORDEN 1

contexto	char	freq	prob
a	b	2	2/7
a	c	1	1/7
a	d	1	1/7
a	ESC	3	3/7
b	r	1	1/2
b	ESC	1	1/
c	a	1	1/2
c	ESC	1	1/2
d	a	1	1/2
d	ESC	1	1/2
r	a	1	1/2
r	ESC	1	1/2

MODELO DE ORDEN 0

char	freq	prob
a	4	4/13
b	1	1/13
c	1	1/13
d	1	1/13
r	1	1/13
ESC	5	5/13

MODELO DE ORDEN -1

char	prob
a	1/257
b	1/257
c	1/257
d	1/257
r	1/257

Supongamos ahora que el string tiene un caracter mas que queremos comprimir, veamos que ocurriría si el caracter fuese una "c", una "d" o una "t"

Nuestro contexto final es "ra", y leemos una "c" observamos que con el contexto "ra" la "c" tiene probabilidad 1/2 por lo que comprimimos "c" con p=1/2.

Si leemos una "d" vemos que "d" tiene probabilidad 0 en el contexto "ra", actualizamos el contexto "ra" y emitimos un escape de probabilidad 1/2, pasamos al modelo de orden-1 por lo que nuestro contexto se reduce a "a". En el contexto "a" la "d" tiene probabilidad 1/7 por lo que comprimimos la "d" con probabilidad = 1/2 * 1/7

Si leemos una "t" vemos que la "t" tiene probabilidad cero en el contexto "ra" por lo que emitimos un escape de probabilidad 1/2 y pasamos al modelo de orden-1, la "t" también tiene probabilidad cero en el contexto "a" por lo que emitimos un escape de probabilidad 3/7 y pasamos al modelo de orden-0, aquí también la "t" tiene probabilidad 0, emitimos un escape de probabilidad 5/16 y pasamos al contexto de orden -1, en el cual codificamos la "t" con probabilidad 1/257.

•**Mecanismo de exclusión**.

Analicemos nuevamente el caso en el cual comprimíamos a la "d", al no estar la "d" en el contexto "ra" pasamos al modelo de orden-1 con contexto "a", veamos el modelo de orden1 con contexto "a"

cont	char	freq	prob
a	b	2	2/7
a	c	1	1/7
a	d	1	1/7
a	ESC	3	3/7

Según este modelo deberíamos codificar la "d" con probabilidad 1/7 pero veamos que esto puede mejorarse, si observamos el contexto "a" podemos ver que los caracteres que tienen probabilidad distinta de cero son (obviando el ESC) {b,c,d}. Sabemos también que venimos del contexto "ra". Observemos qué caracteres tienen probabilidad distinta de cero en el contexto "ra" : {c} luego, en el contexto "a" el caracter "c" no puede ser el que estoy comprimiendo pues si así fuera, hubiese sido codificado en el contexto "ra". En base a esto podemos modificar el modelo de orden-1 disminuyendo la frecuencia de la "c" a cero (solo para este caso ,la tabla del modelo volverá quedar como estaba antes de actualizarla en base a lo emitido).

Si sabemos que "c" no puede ocurrir la probabilidad de la "d" es ahora de 1/6 y comprimimos usando esta nueva probabilidad.

A este análisis que se hace en cada modelo conociendo el contexto del cual se llega se lo denomina principio de exclusión.

Apliquemos el principio de exclusión para el caso en el cual comprimíamos la "t", como "t" no aparece en el contexto "ra" emitimos un escape de probabilidad 1/2 y pasamos al modelo de orden "1" con contexto "a", en este modelo tenemos los caracteres {b,c,d} como "c" aparecía en el contexto "ra" sabemos que no puede ocurrir, como "t" no aparece emitimos un escape de probabilidad 3/6 **(Notar que la probabilidad del Esc no se modifica)**. En el modelo de orden-0 tenemos los caracteres {a,b,c,d,r} "c" aparecía en el contexto "ra" así que lo eliminamos y "b,c y d" aparecían en el contexto "a" así que también los eliminamos, como la "t" no aparece emitimos un escape de probabilidad 5/12. Pasamos al modelo de orden -1, aquí aparecen todos los caracteres pero podemos eliminar a los que "vimos" en otros contextos: {a,b,c,d,r} por lo que a la "t" la comprimimos con probabilidad 1/252.

Sabiendo que la compresión aritmética comprime cada caracter con longitud -log2(P(c)) analizamos cuanto nos ocupa comprimir la "c" la "d" o la "t" en cada caso.

Caracter	Probabilidad sin exclusión	Bits
c	1/2	-log2(1/2) = 1 bit
d	1/2,1/7	-log2(1/14) = 3.81 bits
t	1/2,3/7,5/16,1/257	-log2(15/28784) = 11.90 bits.

Caracter	Probabilidad con exclusión	Bits
c	1/2	-log2(1/2) = 1 bit
d	1/2,1/6	-log2(1/12) = 3.6 bits
t	1/2,3/6,5/12,1/252	-log2(15/36288) = 11.24 bits.

Recordar que para actualizar las frecuencias se utilizan las tablas tal cual estaban, las modificaciones a las probabilidades realizadas por el mecanismo de exclusión se utilizan exclusivamente para la compresión del caracter que se está comprimiendo.

Dado que una mejora en el cálculo de las probabilidades causa una mejora en el nivel de compresión todas las versiones de PPMC utilizan el mecanismo de exclusión.

Podría suponerse que cuanto mayor sea el orden máximo del algoritmo mejor será la compresión, sin embargo esto no es cierto, a medida que el orden aumenta el modelo de predicciones se hace mayor pero a su vez aumenta el overhead en bits utilizados por los "escapes", es fácil imaginar que en algún momento el overhead que se gasta en bits para los escapes empieza a ser mas costoso que los bits que se ahorran al predecir mejor los caracteres. Experimentalmente se ha comprobado que el algoritmo PPMC, comprime cada vez mejor mientras el orden máximo aumenta hasta cinco, con orden máximo seis el nivel de compresión comienza a decrecer.

Nivel de compresión del PPMC de acuerdo al orden máximo del modelo.

Orden máximo	*Nivel de compresión (Bits/Byte)*
0	8.00
1	4.60
2	3.70
3	2.90
4	2.40
5	2.34
6	2.38
7	2.40
8	2.50
9	2.60
10	2.62
11	2.63

PPMC es hoy en día uno de los algoritmos con mayor nivel de compresión, comprime alrededor de 2.3 bits por byte. El algoritmo ha sido estudiado en gran detalle y su principal logro es armar un modelo probabilístico de excelente nivel. Un ejemplo interesante que puede hacerse para chequear un modelo probabilístico para la compresión de textos es comprimir un texto determinado y luego concatenarle al archivo comprimido una cierta cantidad de bits random, si el modelo es muy bueno el archivo descomprimido será el original seguido de algo que tiene un formato e incluso estilo bastante parecido al original.

Ejemplo:

Se comprimió el texto completo de La Biblia con un algoritmo PPMC de orden 5, luego al archivo comprimido se le agregaron bits aleatorios y se descomprimió, a continuación del texto original el algoritmo de descompresión originó lo siguiente:

> hebrews 9
> Answere the to thath good pierceived; an holdnest by in of best to thered givery prinetra, ye and body ough und, evanance the spitestated brews.
>
> 10
> This yiel
> 3 17
> This trusalonicle: for with Lord been of prom the him, the resend a reginess.
>
> hebrews 10 19
> I hat I made withousnessmay be? whethround and humber, and quarrow, the we shall themfulefuge of joy Because the we he will thathe made and holy shall bein fless we law, ear is immed to doctrinessaloniansome, and office, as quit not by from amongue, which werein by train of the shalli.
>
> hebrews 1 5
> Nebuke 24
> And eve alonians 4 But wer! maritted word.

Como puede observarse el modelo alimentado con bits aleatorios genera un texto que mantiene la estructura del texto original y si bien es incoherente es visualmente legible, es decir que a simple vista el texto parece claro. Incluso algunas palabras no existen (trusalonicle, etc) pero sin dudas parecen palabras bíblicas. (esto no pretende ofender a nadie, simplemente debe tomarse como un análisis lingüístico.)

Al repetir el mismo experimento con un Huffman estático de orden cero se obtuvo lo siguiente:

> odeum
> 2y
> . oB
> ?eeeio,y
> ueeem7ssK noo s,,?mm2wsmH
> ee
>
> s
> euuv,oe?t?t emaet e
> edyueve8oWesHne
> oHos'tNeeoe,e wweGyLedeuuCCLGuN9HuwuHtG2Nw uGuRGLdT
> GeNe CoW
> G
> dTwwu ew:Neww8
> e.CeRHeTGewwCwfeCNf

Como puede observarse para los mismos bits aleatorios que antes un modelo estático de orden cero genera un galimatías ininteligible.

En cuanto al futuro, muchos algoritmos de compresión basados en un modelo probabilístico sofisticado y luego compresión aritmética se basan en lo realizado por el método PPM. PPM* desarrollado a continuación es un buen ejemplo del legado de la técnica en el mundillo de la compresión de datos.

Versiones:
Witten, Clearly, Teahan. (1995)

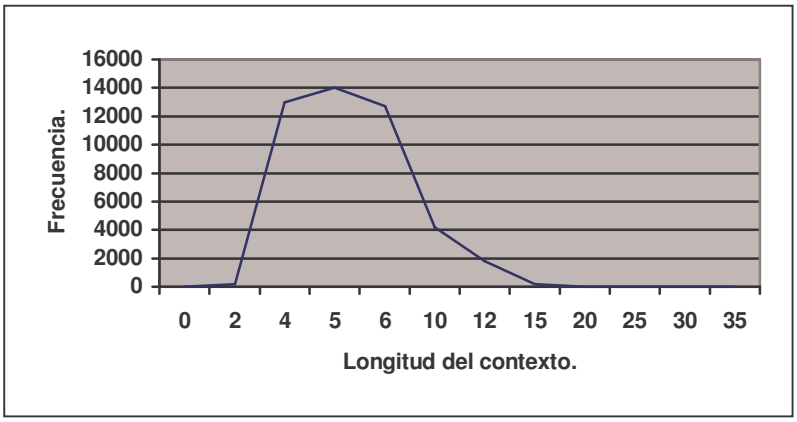
El algoritmo PPMC como hemos visto, calcula la probabilidad de ocurrencia de cada caracter en base a varios contextos en forma conjunta hasta un orden máximo especificado. Si bien el modelo probabilístico construido por PPMC es muy eficiente está limitado por la restricción de no poder usar contextos de longitud ilimitada. El nivel de compresión podría, tal vez, mejorarse si esta restricción no existiera. El algoritmo PPM*, basado en PPMC utiliza modelos de contextos ilimitados. Para poder llevar registro de todos los contextos vistos hasta el momento y no limitar la longitud máxima de los mismos PPM* utiliza estructuras de datos especiales con intención de minimizar el consumo de memoria necesario para registrar todos los contextos.

En PPM* antes de calcular las probabilidades de ocurrencia de cada caracter hay que elegir cual va a ser el contexto en el cual se van a calcular dichas probabilidades. Mientras que en PPMC siempre se elegía el contexto mayor en PPM* esta no es una buena idea pues al estar ilimitada la longitud de los contextos la utilización repetida del contexto de mayor longitud para cada predicción ocasionaría una catarata de escapes que degradarían el nivel de compresión del algoritmo. Otra posibilidad es llevar un registro del nivel de compresión alcanzado por cada contexto, cuando varios contextos son candidatos se elige aquel que tiene mejor nivel de compresión. Curiosamente esta política ocasiona resultados insatisfactorios ya que no permite a contextos relativamente nuevos aprender con rapidez pues estos se ven desplazados por contextos antiguos con mejor nivel de compresión.

Contextos determinísticos.

En PPM* se dice que un contexto es determinístico si hace exactamente una predicción (es decir que, además del escape, un solo caracter tiene frecuencia distinta de cero). La política para elección de contextos de PPM* consiste en elegir de entre todos los contextos determinísticos posibles a aquel de menor longitud. Si no hay ningún contexto determinístico se elige de entre los contextos posibles no determinísticos a aquel de mayor longitud.

A continuación mostramos un histograma sobre la longitud de los contextos determinísticos elegidos por PPM* con la técnica descripta.



No es de extrañar que la longitud de contexto determinístico mas usado sea 5, que es el orden máximo óptimo para PPMC. Pero a su vez en PPM* otros contextos mayores que 5 se utilizan con éxito, como se ve en el gráfico contextos entre 6 y 11 son muy utilizados y otros de longitudes aun mayores también se utilizan en algunas predicciones.

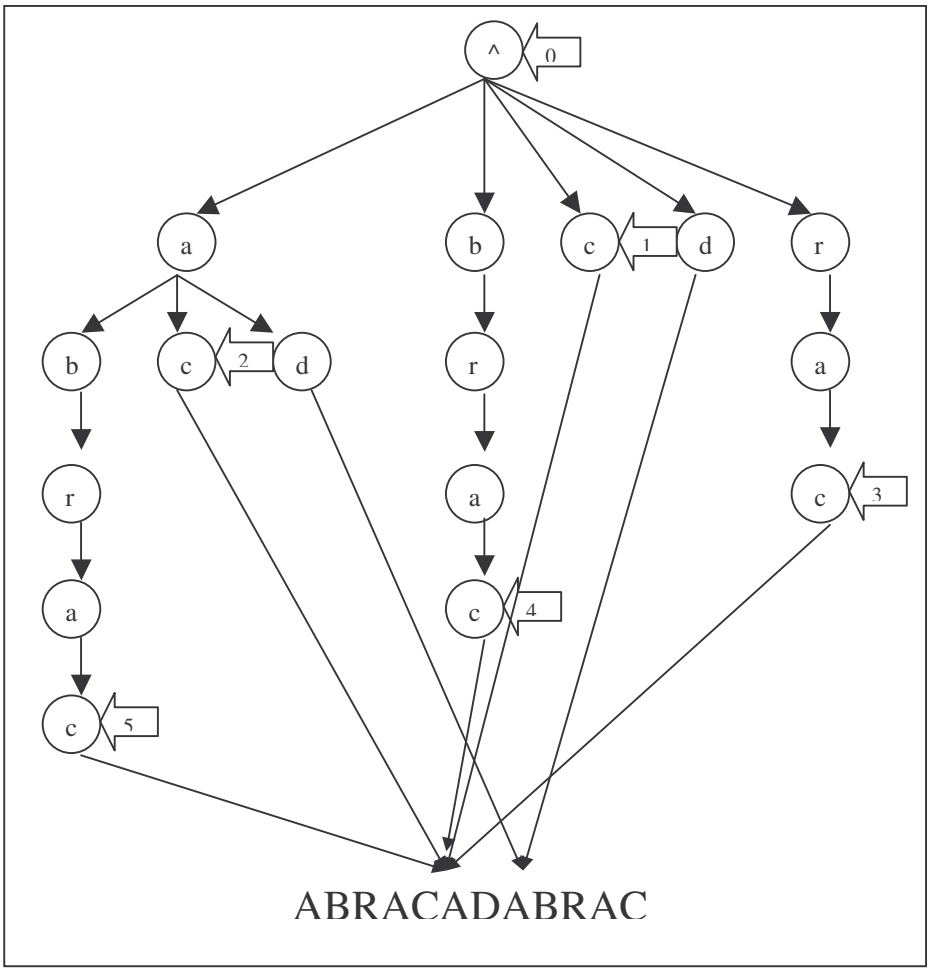
Una vez elegido el contexto con el cual se va a predecir un caracter PPM* trabaja igual que PPMC, si el caracter no aparece en dicho contexto se emite un escape y el algoritmo pasa al contexto de longitud menor hasta llegar al contexto de orden –1 en el cual todos los caracteres son equiprobables.

Conociendo la forma en la cual se elige el contexto y como se comprime a partir del contexto elegido queda descripto el funcionamiento del algoritmo, lo único que queda por estudiar es la forma en la cual PPM* lleva registro de todos los contextos observados.

PPM* utiliza para llevar registro de todos los contextos posibles un árbol tipo Patricia. Dicha estructura es muy similar a un Trie pero con algunos cambios, en primer lugar veamos como trabaja PPM* usando un Trie. (La transformación del Trie al árbol tipo Patricia es sencilla).

PPM* utiliza un trie cuya raíz es el caracter nulo "^" del cual van a desprenderse todos los contextos observados por el algoritmo, antes de explicar la construcción del trie desde cero veamos un ejemplo.

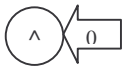
Luego de procesar el string "abracadabrac" el trie del PPM* es el siguiente:



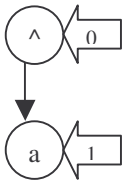
En el trie de PPM* las hojas son siempre para contextos únicos, por ejemplo ‘abrac’ es un contexto único mientras que ‘ab’,’abr’ y ‘abra’ ya aparecieron dos veces en el texto, los contextos únicos apuntan a la posición del string que los seguía. Junto con el trie se guardan 5 punteros a la posición del trie que corresponden al contexto actual, en este caso ‘c’ (1),’ac’ (2), ‘rac’ (3), ‘brac’ (4) y ‘abrac’ (5) además del string nulo que es siempre el contexto cero (0). De todos los contextos posibles se elige uno de acuerdo a la política mencionada y luego en base a dicho contexto se comprime el próximo caracter del string usando compresión aritmética para la probabilidad calculada. (Aclaración : Si bien “ABRAC” aparece 2 veces en el texto, solo una vez se utiliza como contexto)

El proceso de actualización del trie se realiza cada vez que se lee un nuevo caracter del string, para ilustrar como se realizan las actualizaciones vamos a construir el trie anterior desde cero. El string a comprimir es ‘abracadabrac’.

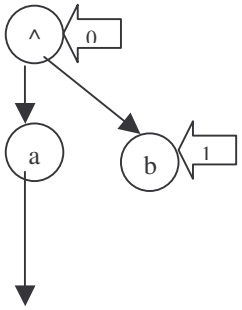
1) En un principio solo existe el contexto nulo.



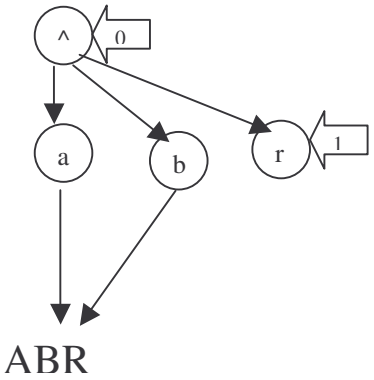
2) Al leer una ‘a’ en este contexto solo existe el escape (con frecuencia 1) por lo que se emite un escape y se comprime a la ‘a’ en el modelo de orden –1 con probabilidad 1/257. Se actualiza el trie quedando:



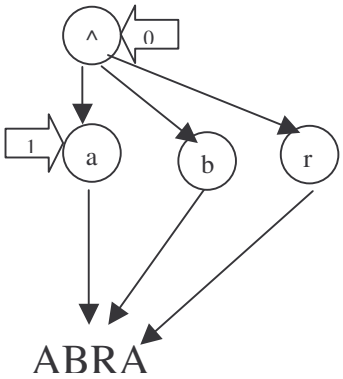
3) Existen ahora dos contextos (0) y (1), siendo el único determinístico el contexto (0), por lo que se elige dicho contexto. Se lee una ‘b’ que al tener frecuencia cero en contexto (0) genera un escape de probabilidad 1/2 y finalmente se comprime con probabilidad 1/257 en el modelo -1. Se actualiza el trie, haciendo que el contexto ‘a’ apunte a la ‘b’ en el archivo y dando de alta el contexto ‘b’.



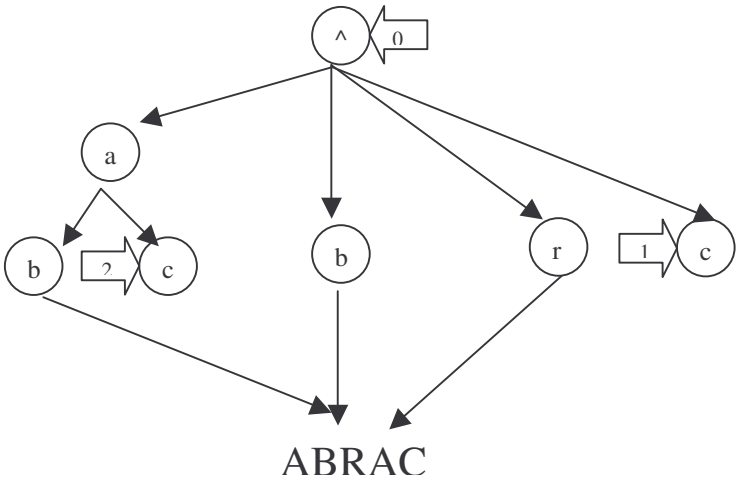
4) Nuevamente hay dos contextos, y no hay contextos determinísticos por lo que se elige el contexto no determinístico mas largo (1), allí la ‘r’ que se lee del archivo genera un escape de probabilidad 1, se pasa al contexto (0) en donde la ‘r’ genera un nuevo escape de probabilidad 2/4, se pasa al modelo de orden –1 y se comprime la ‘r’ con probabilidad 1/257, se actualizan las frecuencias y se actualiza el trie, quedando:



5) Aquí el contexto elegido es el (1), el caracter leído ‘a’ genera en el contexto (1) un escape de probabilidad 1 y en el contexto (0) se comprime con probabilidad 1/6. Se actualiza el trie. Aquí se produce el caso en el cual en el contexto de orden (0) la a existía por lo que se debe crear un nuevo puntero a dicho contexto (el contexto ‘a’ que ya existía).

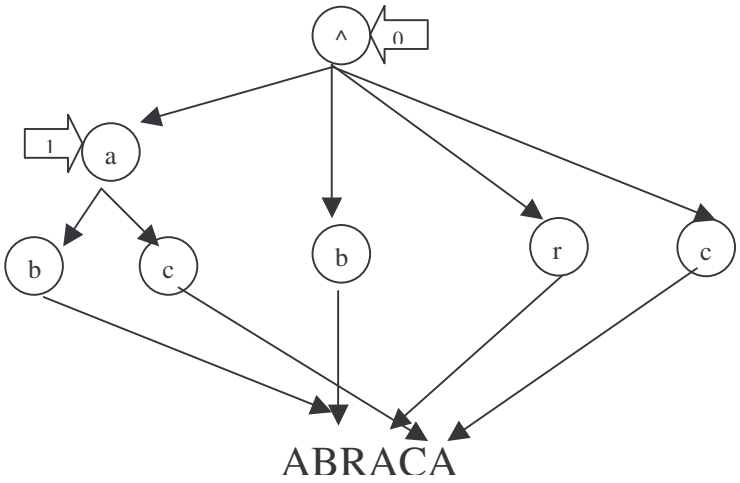


5) Aquí el contexto elegido es el (1) en donde la ‘c’ leída del archivo genera un escape de probabilidad 1/2, se pasa al contexto de orden 0 donde la ‘c’ genera un escape de probabilidad 3/7. (La a tiene frecuencia 2). Finalmente se comprime la ‘c’ en el modelo de orden –1 y se actualiza el trie. En esta actualización como el contexto (1) predecía un caracter distinto hay que hacer un split del nodo creando un nodo para la predicción anterior y un nodo para el caracter nuevo ‘c’. El trie queda:



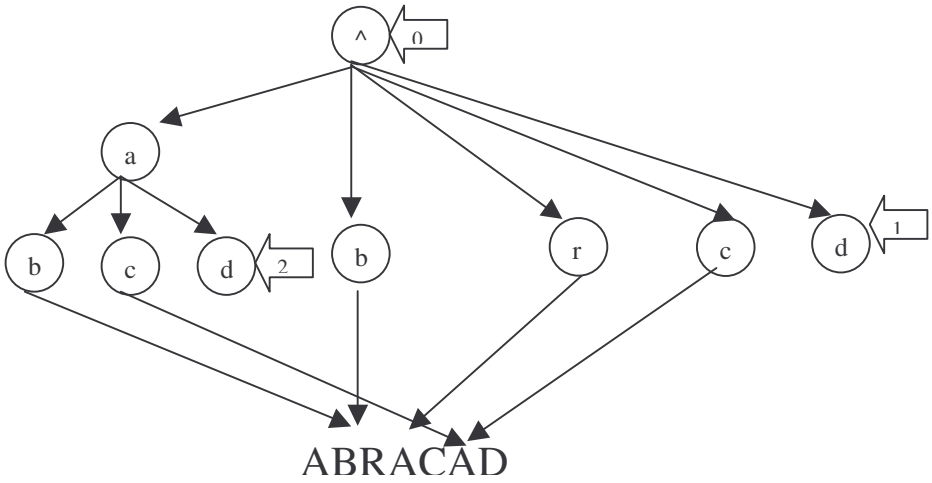
ABRAC

6) Ahora se elige el contexto (2) , en donde la ‘a’ leída genera un escape de probabilidad 1. Se pasa al contexto (1) en el cual se emite otro escape de probabilidad 1, finalmente en el contexto de orden cero (0) se comprime la ‘a’ con probabilidad 2/9. Se actualiza el trie.

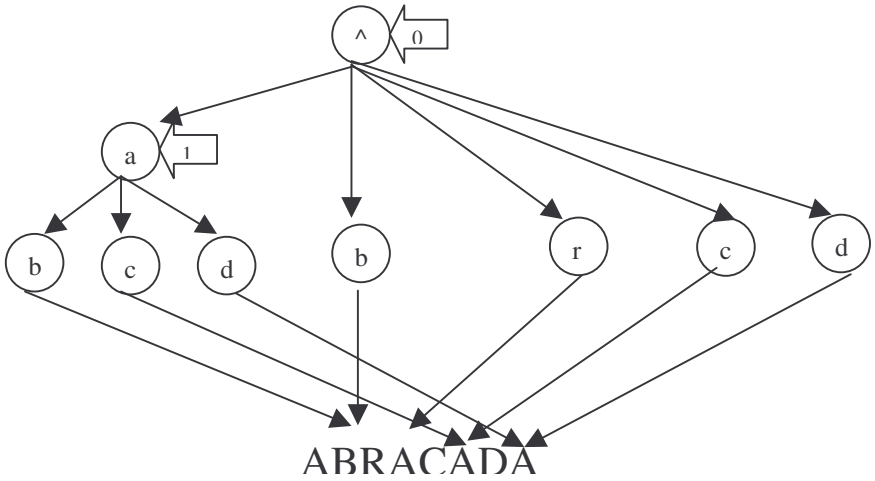


ABRACA

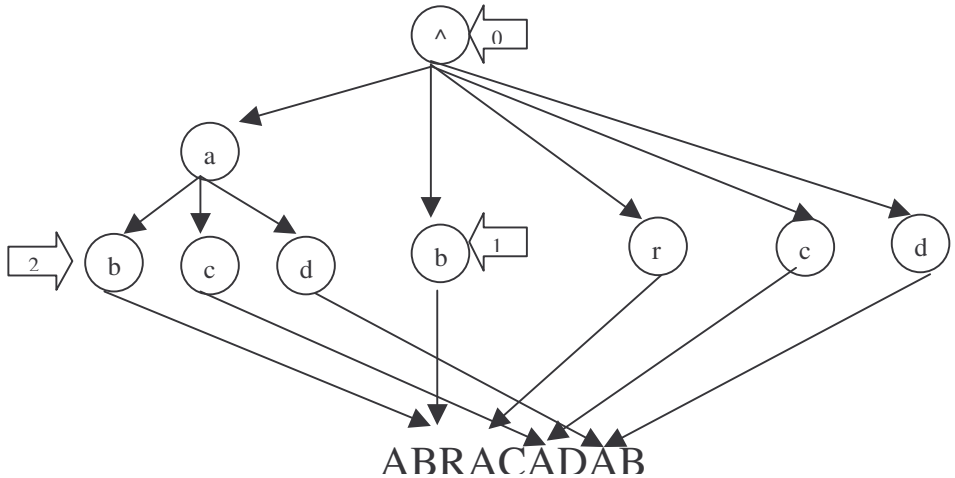
7) Se lee a continuación una ‘d’ que genera un escape en el contexto (1) de probabilidad 2/4, luego genera un escape en el contexto (0) de probabilidad 4/10 y finalmente se comprime en el modelo de orden –1 con probabilidad 1/257. El trie se actualiza de la siguiente manera.



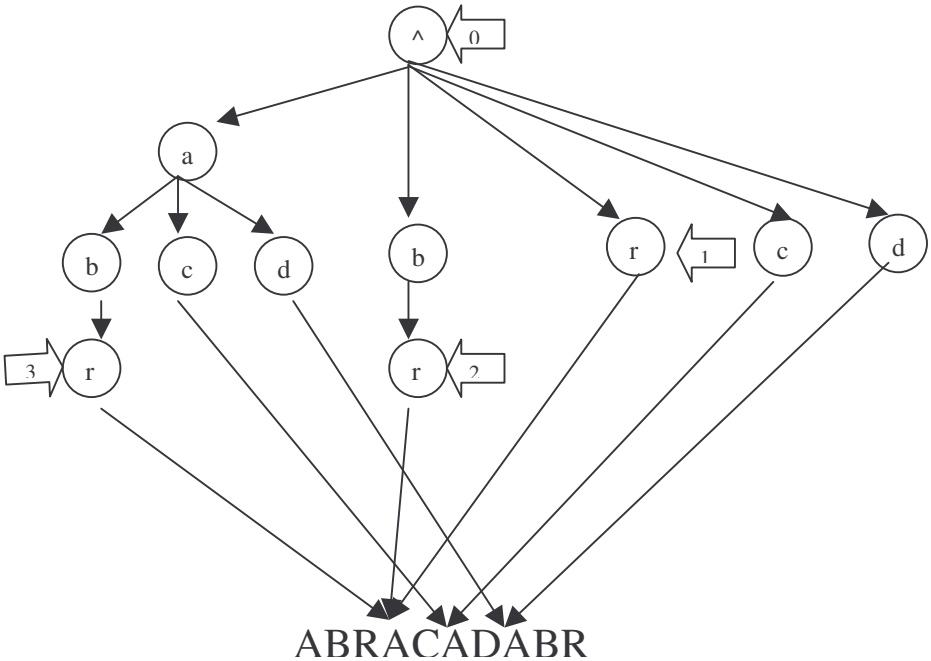
8) El próximo caracter del archivo es una ‘a’, el contexto utilizado es el (2) en el cual se genera un escape de probabilidad 1 y se pasa al contexto 1 en el cual se emite otro escape de probabilidad 1 y se pasa al contexto (0) en donde la ‘a’ se comprime con probabilidad 3/12. Se actualiza el trie.



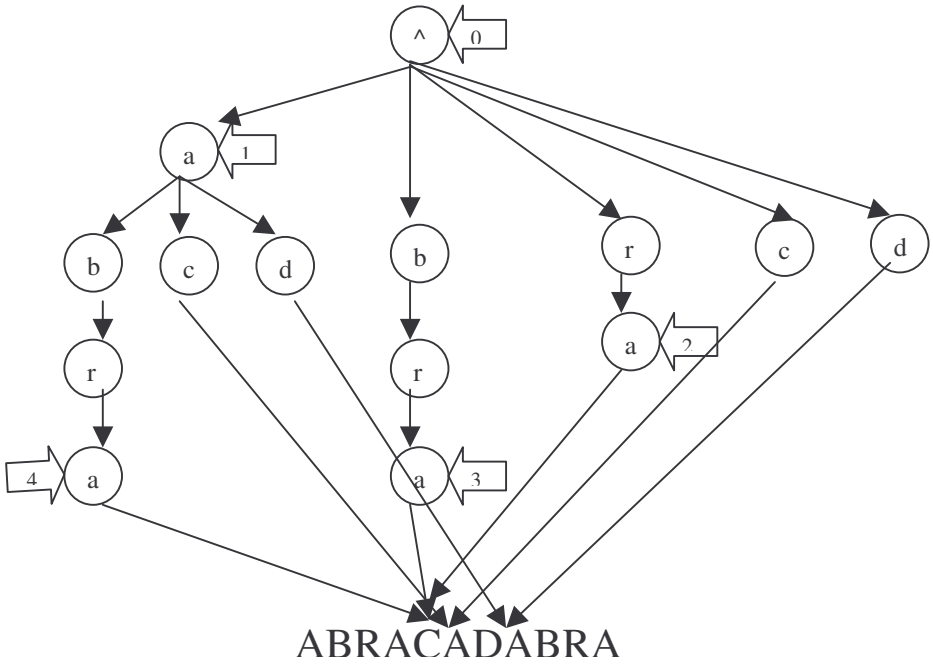
9) El próximo caracter es una ‘b’ que es encontrada en el contexto (1) y se comprime con probabilidad 1/6. Se actualiza el trie simplemente desplazando el puntero del contexto (1) hacia la b y además creando un puntero hacia la b que se descolgaba del contexto (0). Recordemos que PPM* siempre tiene listos todos los contextos anteriores.



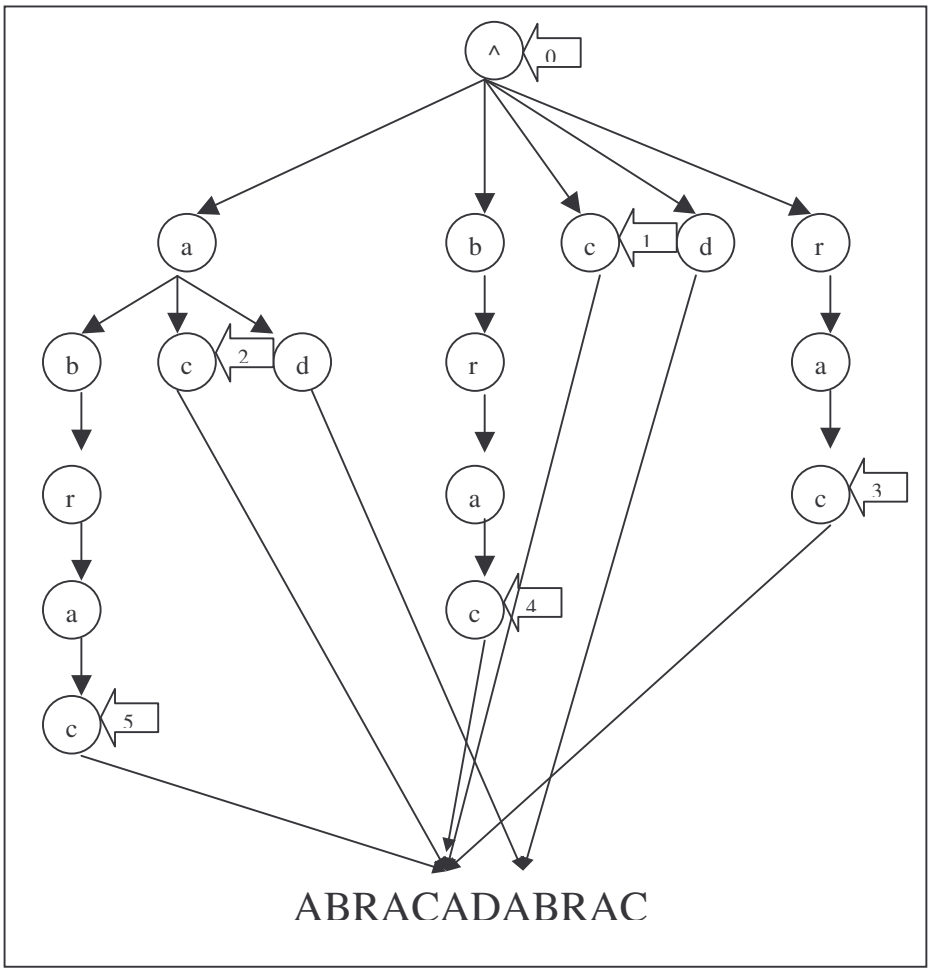
10) Se lee ahora del archivo una ‘r’ que se comprime usando el contexto (2), la r allí se comprime con probabilidad 1/2 y como el nodo apunta a un caracter del string que además fue el caracter observado se agrega un nuevo nodo descendiente de b con la letra ‘r’, lo mismo hay que hacer con el nodo correspondiente al contexto (1) ya que si bien al comprimir solo se usa uno de los múltiples contextos activos luego de comprimir hay que actualizar todos los contextos. Además como el último caracter visto fue una ‘a’ hay que crear un puntero al nodo ‘r’ descendiente de ‘^’.



11) Por ultimo se lee una ‘a’ que en (3) se comprime con probabilidad 1/2, se actualiza el trie de la siguiente forma:



12) Si ahora se lee en ultimo lugar la ‘c’, se comienza por el contexto (4) que es el más largo. Allí se emite un escape con probabilidad 1, al igual que en los contextos (3) y (2). Por último en el contexto (1) se encuentra la c que se comprime con probabilidad 1/7. Se actualiza el trie llegando al trie que presentamos originalmente:



Generalizando se pueden presentar 4 casos al actualizar un contexto en el trie.

1) El caracter existe en el contexto y además hay un nodo descendiente del nodo que representa al contexto en el trie. Entonces se desplaza el puntero hacia dicho nodo. Esto se ve por ejemplo al leer la ‘b’ en ‘ABRACADAB’.

2) El caracter existe en el contexto pero el nodo al cual apunta el contexto apunta al string. En cuyo caso se crea un nuevo nodo descendiente del nodo del contexto con el caracter encontrado. Este caso se observa al leer “A” en “ABRA”

3) El caracter no existe en el contexto y el nodo que representa al contexto apunta a otro nodo inferior se crea un nuevo nodo descendiente del contexto actual que contiene al caracter observado. Se produce una bifurcación en el trie como al leer la “D” de “ABRACAD” (La ‘a’ apuntaba a ‘b’ y ‘c’ y ahora se crea ‘d’)

4) El caracter no existe en el contexto y el nodo que representa al contexto apunta al string (obviamente apunta a un caracter distinto del leído) Entonces hay que crear DOS nuevos nodos descendientes del nodo del contexto actual. Uno para el caracter observado y otro para el caracter al que se apuntaba anteriormente en el string. Esto se ve al leer la “c” en ABRAC. En donde la “a” apuntaba a la “b” en el string y ahora se crean dos nodos descendientes de la “a”, uno para la “b” y otro para la “c”.

Los arboles tipo Patricia.

El árbol tipo Patricia utilizado por PPM* para llevar registro de los contextos visitados es muy similar al Trie presentado con la salvedad de que las cadenas de nodos repetidos se guardan una única vez. La secuencia 'brac', por ejemplo que desciende de 'a' y del raíz en nuestro ejemplo se guardaría una sola vez, con un doble juego de punteros según cual es el lugar desde el cual se accede al trie. Un árbol tipo Patricia es un trie en el cual cada nodo simple tiene un único puntero hacia su descendiente pero en el descendiente, si es un nodo compuesto hay una lista de cadenas dependientes de cual es el nodo padre. Cada cadena funciona como una rama del árbol independiente pero guardando los datos solo una vez. Este tipo de estructura se utiliza siempre que se deben almacenar árboles muy grandes y se necesita consumir la menor cantidad de memoria posible. El manejo de la estructura es complejo pero permite a PPM* llevar registro de todos los contextos observados a un costo relativamente bajo en términos de uso de memoria. Hay que destacar que el consumo de memoria de PPM* es de todas formas mucho mayor al de PPMC.

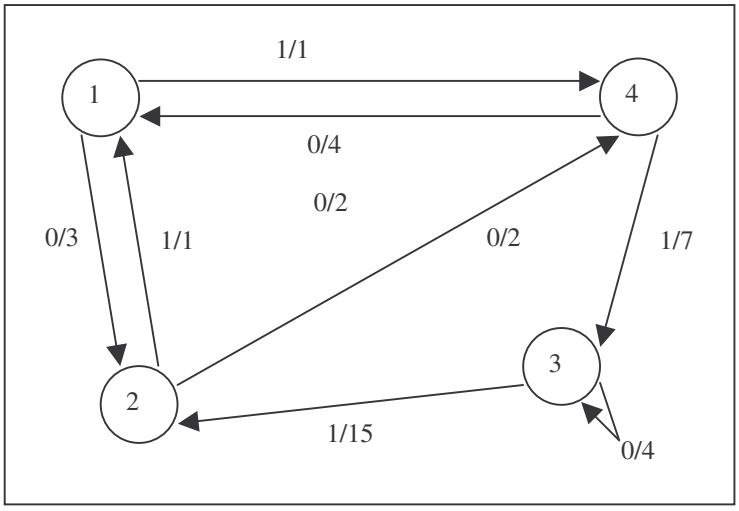
En cuanto al nivel de compresión del algoritmo PPM* alcanza 2.32 bitsxbyte con 32 Mb de memoria y llega a un limite de 2.28 bitsxbyte en hardware con mas de 64 Mb de RAM. Desde el punto de vista teórico PPM* es muy importante ya que se trata del primer algoritmo que calcula probabilidades utilizando para ello contextos de longitud ilimitada.

DMC: Dynamic Markov Compression.

Versiones:
Horspool. (1987)

El algoritmo de compresión DMC (1995) utiliza una máquina de estados finitos como modelo probabilístico del algoritmo para comprimir. A los modelos de este tipo se los suele denominar modelos de Markov, en honor al matemático ruso A.A.Markov. El algoritmo procesa el archivo de entrada tomando de a un bit, (en lugar de un byte como la mayoría de los compresores). Antes de procesar cada bit el algoritmo se encuentra en un determinado ”estado”, para cada estado hay una probabilidad asignada al uno y otra al cero que es la probabilidad usada para comprimir el próximo bit. Dicha probabilidad se comprime utilizando compresión aritmética.

En el siguiente ejemplo se muestra una posible situación del algoritmo DMC en la cual la máquina de estados finitos presenta 4 estados.



En la ilustración se observa el estado de la máquina de 4 estados, para cada estado existen dos posibles salidas (1 y 0), para cada posible salida se indica la frecuencia con que dicha salida ha sido observada, de esta forma si el algoritmo se encuentra en el estado 4, un bit en 0 se codifica con probabilidad 4/11 y además se pasa al estado 1.

Estado actual del modelo del gráfico.

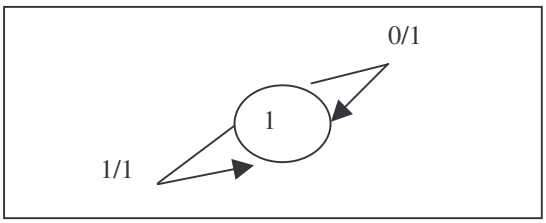
Estado Actual	Bit	Probabilidad	Estado Final
1	1	1/4 = 2 bits.	4
1	0	3/4 = 0.41 bits.	2
2	1	1/3 = 1.58 bits.	1
2	0	2/3 = 0.58 bits.	4
3	1	4/19 = 2.24 bits.	2
3	0	15/19 = 0.34 bits.	3
4	1	7/11 = 0.65 bits.	3
4	0	4/11 = 1.46 bits.	1

Como puede verse dependiendo de la probabilidad de ocurrencia de cada bit el espacio que ocupa dicho bit comprimido varía, en el estado 3 por ejemplo el bit 0 es muy probable y se comprime en 0.34 bits, el 1 en cambio se expande a 2.24 bits debido a que tiene una probabilidad de ocurrencia muy baja.

Hay que destacar que sin utilizar compresión aritmética el algoritmo sería imposible de construir ya que jamás podría comprimirse un bit en menos de un bit.

Estado Inicial del Algoritmo.

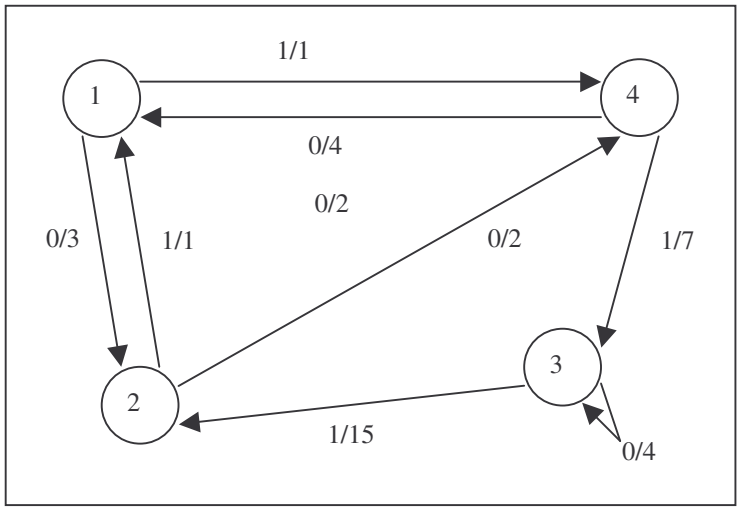
La máquina de estados finitos del DMC comienza siempre en un mismo estado denominado estado inicial cuyo modelo es el siguiente:



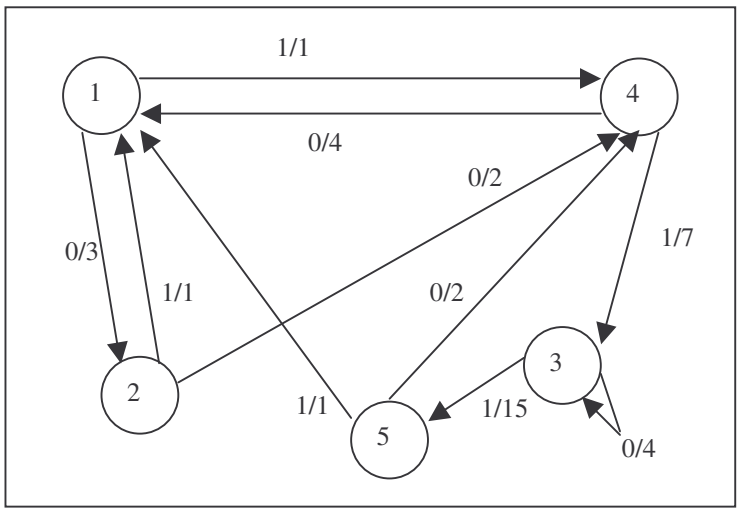
A partir del estado inicial del modelo, cada vez que se encuentra un 1 o un cero en dicho estado se incrementa la frecuencia de dicho bit, cuando uno de los bits alcanza una frecuencia demasiado alta en relación al otro bit se realiza un proceso de clonación del estado al cual se llega por el camino de salida más pesado, creándose un nuevo estado. La decisión sobre cuando se debe realizar una clonación es heurística y es uno de los parámetros a considerar a la hora de implementar el algoritmo, muchas clonaciones reducen la velocidad del algoritmo y además consumen mucha memoria ya que el numero de estados se incrementa dramáticamente, por otro lado el demorar el proceso de clonación reduce el nivel de compresión (esto puede intuirse pensando en que el bit más probable no puede reducirse en más de un bit (lo que ocupaba sin comprimir) mientras que el menos probable puede expandirse en bastante más de un bit, haciendo que el ahorro logrado con varias ocurrencias del primero se pierda con la aparición de uno solo del segundo). En general es necesario buscar un equilibrio aceptable entre ambas opciones según el uso que se le vaya a dar al algoritmo.

Ejemplo de clonación.

Si consideramos el siguiente modelo de 4 estados :



Podemos ver que desde el estado 3 la transición al estado 2 (bit 1) tiene una frecuencia mucho mayor en relación al bit 0 (que vuelve al estado 3), por lo que podría decicidirse (depende de la heurística utilizada) realizar una clonación, la clonación originaría el siguiente modelo:



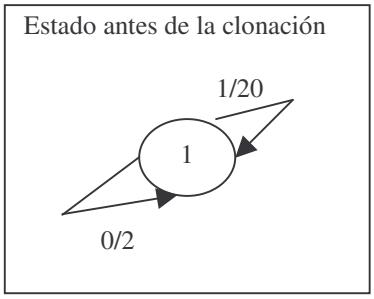
Los cambios producidos por la clonación fueron los siguientes:

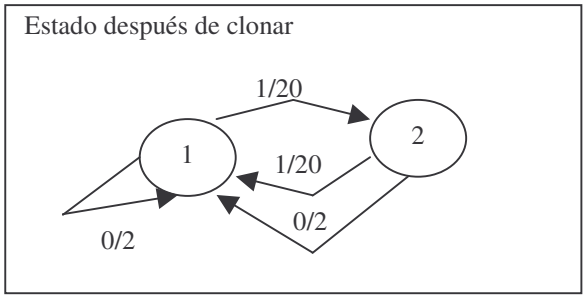
Como la transición de 3 a 2 era muy usada se crea un nuevo estado 5 (una suerte de estado 2') y la transición desde 3 se dirige ahora al estado nuevo. Las salidas desde este nuevo estado se copian iguales a las salidas desde el estado 2.

Hay que analizar ahora cual es el efecto producido por la clonación en el nivel de compresión del algoritmo.

Si se analiza se observa que el modelo nuevo comprime exactamente igual que el modelo anterior, la diferencia es que antes se llegaba al estado dos desde el estado 1 o desde el 3, ahora es como si existieran dos estados 2 diferentes, uno al cual se llega desde el estado 1 y otro al cual se llega desde el estado 3. Si eventualmente una vez que se sale del estado 3 por un bit en 1 se observara que luego un bit en cero es muy probable esta probabilidad no se vería afectada por lo que ocurre al salir del estado 1 con un bit en cero. De esta forma el nivel de compresión del algoritmo se mejora.

Veamos otro ejemplo en el cual mejora la compresión gracias a la clonación:





Veamos qué sucedería al comprimir la tira 1010 en cada caso

Sin Clonar			
Estado Actual	**Bit**	**Probabilidad**	**Estado Final**
1	1	20/22 = 0.041 bits.	1
1	0	2/23 = 1.060 bits.	1
1	1	21/24 = 0.058 bits.	1
1	0	3/25 = 0.921 bits.	1
Total		2.080 bits	

Con Clonación			
Estado Actual	**Bit**	**Probabilidad**	**Estado Final**
1	1	20/22 = 0.041 bits.	2
2	0	2/22 = 1.041 bits.	1
1	1	21/23 = 0.039 bits.	2
2	0	3/23 = 0.885 bits.	1
Total		2.006 bits	

El algoritmo DMC tiene bastante relación con el algoritmo PPMC. En DMC cada estado establece un contexto, el contexto esta dado por los bits que se han visto con anterioridad. El proceso de clonación permite la creación de nuevos contextos de forma tal que las ocurrencias de 1 y 0 en los nuevos contextos no dependa de varios estados anteriores sino únicamente de aquel desde el cual se ha llegado con mayor frecuencia. PPMC comprime en base a un modelo de 'contextos' finitos, mientras que DMC se basa en un modelo de 'estados' finitos con lo cual es capaz de realizar mejores predicciones. Esta ventaja del DMC se ve compensada con la necesidad de realizar la clonación en forma heurística y en definitiva ambos algoritmos tienen un nivel de compresión muy similar. Se supone que en esquemas de memoria cuasi-infinita DMC debería ser por lejos el mejor algoritmo de compresión pero no hay forma de probarlo (por ahora).

El nivel de compresión de DMC es similar al del PPMC, sin embargo la máquina de estados finitos de DMC es mucho mas fácil de implementar, solo hace falta implementar la estructura y el proceso de clonación. En cuanto al consumo de memoria ambos algoritmos consumen una cantidad equivalente, en ambos casos el consumo de memoria es elevado. La principal desventaja de DMC con respecto al PPMC y a todos los demás algoritmos de compresión es que DMC es un algoritmo extremadamente lento ya que procesa el archivo de entrada de a un bit. A menos que se realice una implementación extremadamente eficiente DMC tiende a ser bastante mas lento que la mayoría de los algoritmos. Si se tiene cuidado en optimizar la velocidad de procesamiento del algoritmo DMC es muy recomendable ya que tiene un nivel de compresión excelente y es muy simple en cuanto a su implementación.

- **Compresión por Run-length**

La compresión de archivos por run-length es uno de los métodos mas simples de comprender, la idea básica consiste en reemplazar secuencias de caracteres repetidos en un archivo por un par ordenado del tipo (caracter, veces) indicando la cantidad de veces que el caracter se repite.

Por ejemplo:

AAAABBCCCCDEEE

Se codificaría como:

4A2B4C1D3E

Esto resulta ineficiente para los caracteres que no se repiten pues hay que guardar longitud 1 para cada uno. Una solución es codificar de una cierta forma los pares ordenados y de otra los caracteres simples. Este esquema sencillo da origen a algunos de los algoritmos de compresión mas utilizados como por ejemplo el LZ77, que desarrollaremos a continuación.

Versiones:

LZ77 J.Ziv - A.Lempel. (1977) algoritmo original.
LZSS J.A.Storer - T.G.Szymanski. (1982) leves cambios.
LZ77b T.C.Bell. (1986) usando árboles binarios.

Este algoritmo desarrollado por Lempel y Ziv se basa en un esquema de run-length encoding, para encontrar los caracteres que se repiten utiliza una ventana (buffer) que con la cual recorre el archivo, por este motivo al LZ77 se lo suele conocer como "sliding windows" o "ventanas deslizantes".

El LZ77 deberá fijar primero el tamaño de la ventana o buffer, la ventana se dividirá en dos partes:

- La memoria. (memory window)
- La ventana de inspección. (lookahead window)

El tamaño de cada ventana deberá ser determinado con anterioridad.

El esquema de compresión es el siguiente: El algoritmo posiciona la ventana de inspección al comienzo del archivo y comienza a deslizarla sobre el mismo moviéndose de a un byte a la derecha (¿cuál es la derecha en un archivo?). Por cada byte procesado el algoritmo verifica si los n(*), (n debe ser mayor o igual a una longitud mínima de la repetición, que es uno de los parámetros del algoritmo, por ejemplo n>=2) primeros caracteres de la ventana de inspección pueden encontrarse en la memoria, (busca la aparición mas larga, ya que un string puede aparecer en varias partes de la memoria). Si no encuentra repeticiones simplemente emite el código del caracter, si las encuentra emite un par ordenado (posición, longitud) indicando la posición de la ventana de memoria donde se encontró el string y cual es su longitud, desplazando a la ventana n bytes en lugar de 1.

A los fines de ahorrar bits (cosa que obviamente es muy importante aquí) las posiciones se cuentan desde 0 y las longitudes se generan restándole el mínimo, así por ejemplo si la longitud fuera 3 y la longitud mínima fuera 2, se emitiría, como segundo valor del par ordenado, un 1.

Ejemplo:

Sea el archivo:

AADAAAABCDABCBCDAA (18 caracteres = 18 bytes)

Supongamos que usamos una ventana de 12 bytes dividida en 6 bytes para la memoria y 6 bytes para la ventana de inspección.(los "*" indican posiciones vacías en la ventana de inspección. Antes de empezar se encuentra totalmente vacía)

|******|**A**ADAAA|ABCDABCBCDAA Emite: A

|*****A|**A**DAAAA|BCDABCBCDAA Emite: A

|****AA|**D**AAAAB|CDABCBCDAA Emite: D

|***__AA__D|__AA__AABC|DABCBCDAA Emite: (3,2)

|*__AA__DAA|__AA__BCDA|BCBCDAA Emite: (1,2)

A|ADAAAA|**B**CDABC|BCDAA Emite: B

AA|DAAAAB|**C**DABCB|CDAA Emite: C

AAD|AAAABC|**D**ABCBC|DAA Emite: D

AADA|AA**ABC**D|**ABC**BCD|AA Emite: (2,3)

AADAAAA|**BCDA**BC|**BCDA**A Emite: (0,4)

AADAAAABCDA|BCBCDA|**A** Emite: A

EOF

El archivo comprimido quedaría:

A A D (3,2) (1,2) B C D (2,3) (0,4) A

El problema que se presenta es como distinguir en el archivo comprimido a la codificación de un par ordenado de la codificación de un caracter común, para ello pueden utilizarse códigos de 9 bits de forma tal que el primer bit indica si es un caracter simple o un par ordenado. Si es un caracter simple sabe que los próximos 8 bits representan al caracter en ASCII, si es un par ordenado los próximos n bits representan al par (posición, longitud), n depende de los tamaños máximos y mínimo de las repeticiones y de las posiciones posibles, las cuales están dadas por el tamaño de las ventanas.

Para nuestro archivo la posición máxima es 4, ya que en la posición 5 (última de la ventana de inspección) no es posible tener una repetición de al menos 2 de longitud, y por lo tanto necesitamos 3 bits para almacenarla.

La longitud máxima es 6, pero a su vez la mínima es 2 por lo que necesitaremos otros 3 bits (de 2 a 6 hay 5 números=3 bits, utilizaremos los números de 0 a 4). Notar que siempre es mejor usar múltiplos de 2 para las posiciones y longitudes máximas pues sino desperdiciamos bits. En definitiva se necesitan 7 bits (el bit extra es para distinguir a un caracter de un par ordenado).

En nuestro ejemplo quedaría:

```
 A     = 9 bits.
 A     = 9 bits.
 D     = 9 bits.
(3,2)  = 7 bits.
(1,2)  = 7 bits.
 B     = 9 bits.
 C     = 9 bits.
 D     = 9 bits.
(2,3)  = 7 bits.
(0,4)  = 7 bits.
 A     = 9 bits.
```

total = 84 bits = 11 bytes.

Con lo cual comprimimos el archivo de 18 a 11 bytes.

Para descomprimir, el descompresor trabaja en forma similar y es muy simple, va moviendo su propia ventana sobre el archivo que va generando y cuando detecta un par ordenado efectúa el reemplazo correspondiente según la ventana memoria. El descompresor debe leer de a 9 bits del archivo pues el primer bit le indica si es un caracter o un par ordenado. (si es un par ordenado sabe que debe tomar solo los siguientes 6 bits y guardarse los otros 2 para seguir leyendo).

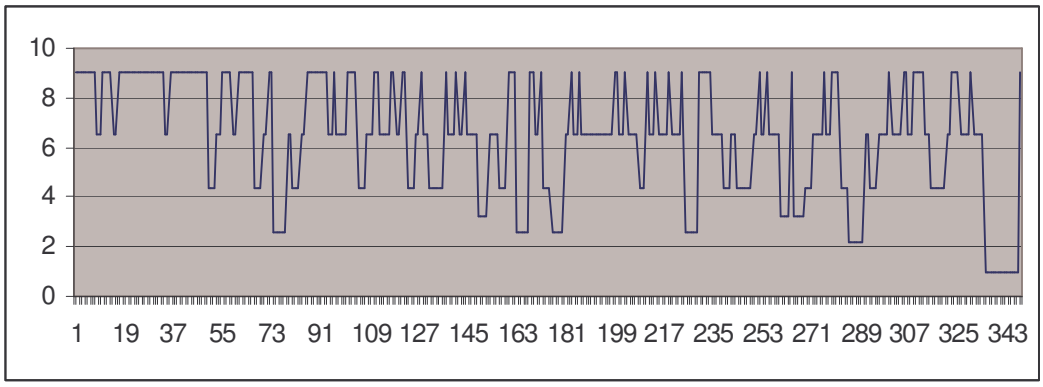
Ejemplo para nuestro archivo:

Memoria	*Ventana-inspección*	*Emite*
-	AAD(3,2)(1,2)B	A
A	AD(3,2)(1,2)BC	A
AA	D(3,2)(1,2)BCD	D
AAD	(3,2)(1,2)BCD(2,3)	AA
AADAA	(1,2)BCD(2,3)(0,4)	AA
ADAAAA	BCD(1,3)(0,4)A	B
DAAAAB	CD(2,3)(0,4)A	C
AAAABC	D(2,3)(0,4)A	D
AAABCD	(2,3)(0,4)A	ABC
BCDABC	(0,4)A	BCDA
BC	A	A
BCA	EOF	EOF

Con lo cual recuperamos el archivo original: AADAAAABCDABCBCDAA

En el LZSS Storer y Szimanski proponen algunas mejoras como por ejemplo la utilización de árboles para buscar mas rápidamente las repeticiones en la memoria.

Bits x byte emitidos por LZ77 para el fragmento Asimov.



La distribución es típica de un LZ77 con muchos códigos de 9 bits y puntos bajos cuando se encuentran repeticiones.

•Principio de Localidad.

Sean los siguientes strings:

S=”cbbcddabdbadacca”

L=”abaabbabcdcdcdcd”

Las probabilidades de los caracteres es igual en ambos strings, es más, en ambos strings los caracteres son equiprobables, pero a simple vista parecería ser que el segundo archivo es mucho mas fácil de comprimir que el segundo. Sin embargo no hay un método de los vistos hasta el momento que pueda comprimir eficientemente el string, ya que para un algoritmo de Run-Length no habría grandes secuencias a comprimir y para un algoritmo estadístico todos los caracteres son equiprobables, Esta es la desventaja de los compresores estadísticos, cuentan cuantas veces aparecen los caracteres sin considerar donde es que aparecen dichos caracteres. En el string “L” podemos observar que se definen dos zonas, una en donde están ubicadas todas las “a” y “b” y otra donde aparecen las “c” y “d”.

Cuando pocos caracteres se concentran en una porción relativamente pequeña de un archivo se dice que el archivo esta localizado. La relación cantidad de bytes/ cantidad de caracteres distintos se puede utilizar para medir la localidad de una determinada porción de archivo.

Un archivo fuertemente localizado puede dividirse en zonas en las cuales es fácil reconocer la sobrepoblación de un determinado conjunto de caracteres. La aparición de algunos caracteres aislados en una zona donde abundan otros caracteres no afecta demasiado la localidad de la zona. Ejemplo:

“acffcaffactafcefaczffacafac”

En el string de arriba la aparición asilada de una “t” o una “z” no afecta demasiado la fuerte localidad del string en donde aparecen una enorme proporción de “a”, ”c” y “f”.

Los archivos que se comprimen fácil por run-length por presentar una secuencia larga de caracteres repetidos son archivos con un alto índice de localidad pero hay otros que también tienen alto grado de localidad y no se comprimen claramente por run-length. Por ejemplo nuestro string “L”.

Existe un algoritmo de compresión basado en el principio de localidad: el algoritmo MTF.

Versiones:

MTF (Move-To-Front) Peter Fenwick 1996.

El algoritmo de compresión por run-length mas moderno es el denominado 'Move to Front' (MTF). Este algoritmo esta diseñado para comprimir eficientemente secuencias de caracteres en las cuales aparecen pocos caracteres distintos, independientemente del orden en que estos aparezcan. En consecuencia MTF comprimirá también en gran medida largas tiras en las cuales se repite un solo caracter tal y como lo hacen los compresores por run-length tradicionales.

El algoritmo fue desarrollado por el Neocelandés Peter Fenwick y presentado por primera en un paper de Fenwick para la ACSC en 1996. Hoy en día los algoritmos mas modernos que usan en su totalidad o en parte una técnica de compresión por run-length utilizan MTF.

La idea del MTF es convertir al archivo original en un nuevo archivo que pueda ser comprimido con gran facilidad por un método estadístico, para ello se basa en las secuencias de caracteres repetidos dentro del archivo. El MTF va a aprovechar estas secuencias de forma tal que al resultado sea muy provechoso aplicarle un método estadístico. El algoritmo, además, tiene otra ventaja: no solamente aprovecha las secuencias repetidas de caracteres sino que también aprovecha aquellas secuencias en las cuales aparecen pocos caracteres sin importar el orden en que estos aparezcan.

El algoritmo es sorprendentemente sencillo de comprender aunque resulta un tanto complejo de explicar en forma teórica, sin embargo mediante un ejemplo es muy sencillo entender el funcionamiento del mismo.

Supongamos que tenemos el mismo archivo que quisimos comprimir con el Lz77:

"AADAAAABCDABCBCDAA" (18 bytes)

Lo primero que debe hacer el MTF es disponer de un arreglo con todos los caracteres posibles, ordenado por su valor (esto es importante, ya que si se ordenara por el orden de aparición en el archivo, es imposible descomprimirlo). Para el ejemplo utilizaremos solo los caracteres que tiene el archivo:

Y='ABCD'

A continuación se va procesando cada caracter del string original y se construye una lista "R" de la siguiente forma:

- A la lista "R" se agrega el numero de caracteres que preceden al caracter leído en el vector "Y".

- Se modifica el vector “Y” colocando el caracter al comienzo del mismo y el resto de los caracteres se corren un lugar hacia la derecha.

El seguimiento en nuestro caso es:

Carácter leído	Lista "R"	Nuevo Vector "Y"
A	0	ABCD
A	00	ABCD
D	003	DABC
A	0031	ADBC
A	00310	ADBC
A	003100	ADBC
A	0031000	ADBC
B	00310002	BADC
C	003100023	CBAD
D	0031000233	DCBA
A	00310002333	ADCB
B	003100023333	BADC
C	0031000233333	CBAD
B	00310002333331	BCAD
C	003100023333311	CBAD
D	0031000233333113	DCBA
A	00310002333331133	ADCB
A	003100023333311330	ADCB

La lista R que se obtuvo del proceso anterior es:

"003100023333311330"

Esta lista indica los códigos ASCIIs de los caracteres en los cuales se transforma nuestro archivo original, al resultado de "R" se le aplica ahora cualquier método estadístico adaptativo, como por ejemplo Huffman o compresión aritmética obteniéndose el archivo comprimido.

Si aplicamos Huffman adaptativo el árbol que se formaría nos daría códigos de longitud 1 para el caracter "3", de longitud 2 para el caracter "0" y de longitud 3 para los caracteres "1" y "2", la longitud total del archivo comprimido sería de 37 bits, es decir alrededor de 5 bytes. Como puede verse el resultado es mucho mejor que los 11 bytes que obteníamos usando el LZ77.

El funcionamiento del algoritmo MTF puede parecer un tanto 'mágico', sin embargo es fácil comprender su idea fundamental, una larga tira de caracteres repetido originará en la lista "R" una larga tira de caracteres cero por lo que luego al aplicar un método estadístico el caracter cero será muy probable y comprimiremos el archivo en gran forma.

Notemos que el algoritmo MTF necesita como complemento de algún método de compresión estadístico para poder funcionar en forma completa.

Notemos también que el MTF como todo algoritmo por run-length aprovecha las secuencias de caracteres repetidos mientras que los métodos estadísticos tradicionales no pueden hacerlo. Por ejemplo:

Si tenemos el archivo: "AAAABBBBCCCCDDDD"

Por cualquier método estadístico observaremos que los 4 caracteres que forman al archivo son equiprobables, el árbol de Huffman que se genera asignara códigos de longitud 2 bits a cada caracter por lo que el archivo final ocupará:

2x4+2x4+2x4+2x4 = 32 bits.

Si aplicamos 'Move to Front' obtenemos en la lista "R":

0000100020003000

Aplicando Huffman adaptativo a la lista "R" obtenemos que al caracter '0' se le asigna un código de un bit, al 1 un código de 2 bits mientras que a los caracteres 2 y tres les corresponden códigos de 3 bits. El resultado final es un archivo de 21 bits (hay un bit extra debido a que el algoritmo adaptativo al comenzar comprime el primer cero en 2 bits pues considera a todos equiprobables). Mucho mejor que aplicando solo Huffman pues estamos aprovechando las secuencias repetidas del archivo.

La principal ventaja de MTF no es sin embargo que su rendimiento sea superior al del LZ77 sino que es capaz de comprimir secuencias que LZ77 no puede, supongamos que tenemos un bloque en el cual aparecen pocos caracteres pero en forma desordenada (sin largas tiras de caracteres repetidos), si aplicamos LZ77 no podremos comprimir casi nada, en cambio usando MTF se generará una lista "R" en la cual predominan los números bajos (0,1,2,3) esta lista como sabemos en la cual los caracteres 0,1,2 y 3 serán mucho mas probables que los demás puede ser comprimida con gran facilidad utilizando cualquier método estadístico.

En cuanto a su rendimiento en velocidad MTF es un algoritmo que puede implementarse en forma extremadamente sencilla y suele resultar muy veloz, la implementación de MTF implica convertir el caracter leído en un número de la lista "R" y a dicho número procesarlo inmediatamente por un método estadístico adaptativo. En comparación con cualquier método estadístico el overhead por convertir cada caracter del archivo en un elemento de la lista "R" es muy chico.

El algoritmo MTF, de muy reciente aparición en el mundo de los métodos de compresión, es de gran importancia pues mejora el campo de los métodos por run-length, casi abandonados desde el LZSS y proporciona una forma sumamente sencilla y elegante de comprimir un archivo en combinación con un método estadístico.

Versiones:

Block Sorting Transformation. M.Burrows, D.J.Wheeler. (1994)
No tenemos foto de Burrows y Wheeler, agradeceremos donaciones.

A continuación presentamos una técnica de compresión que no es en si misma una técnica de compresión sino que es un método por el cual se puede transformar a un bloque de datos en otro bloque de forma tal que el segundo sea muy fácil de comprimir.

Por ejemplo: Si tenemos el string 'abacadae' sería mucho mas conveniente tener en lugar del anterior al string 'aaaabcde' o en todo caso 'aaabcade' en los cuales podríamos aprovechar la secuencia repetida de letras 'a'. El algoritmo de Burrows y Wheeler se encarga precisamente de esta tarea: convertir un string en otro de forma tal que el segundo sea fácil de comprimir por run-length y de forma tal que la transformación que se aplica al string sea reversible para poder descomprimir al archivo.

Objetivo: "Dado un string "S" aplicarle al mismo una transformación reversible de forma tal que el string resultado pueda ser comprimido con mayor facilidad que "S" utilizando un método tipo run-length"

Para lograr el objetivo antes mencionado Burrows y Wheeler presentan dos algoritmos: El algoritmo 'C' que sirve para transformar a un string "S" y un algoritmo 'D' que recupera al string "S" del transformado.

El procedimiento es el siguiente:

Algoritmo 'C'

1) Se forma una matriz "M" que contiene al string "S" y a todas sus rotaciones.
2) Se ordena la matriz lexicográficamente.

Como resultado se obtienen:
- Un string "L" que es la ultima columna de la matriz ordenada.
- Un numero entero "I" que indica el numero de fila del string original "S" en la matriz "M", contando desde 0.

Ejemplo: Sea el bloque S= "brodoro"

1) Formamos la matriz "M" con todas las rotaciones de "S":

M = brodoro
rodorob
odorobr
dorobro
orobrod
robrodo
obrodor

2) Ordenamos "M"

M = brodoro
dorobro
obrodor
odorobr
orobrod
robrodo
rodorob

El resultado es L="oorrdob" (última columna)
I = 0 (S=M[0]).

Algoritmo 'D'

El objetivo es dados "L" e "I" recuperar el string "S", observemos que no conocemos a la matriz "M" pues el resultado del algoritmo 'C' es únicamente "L" e "I".

1) Se ordena lexicográficamente el string "L", al string resultante se lo llama "F". cada fila de "M" es una permutación de "S" luego cada columna es también una permutación de "S". "L" es una permutación de "S", al ordenar "L" obtenemos la primer columna de "M" pues sabemos que "M" estaba ordenada.

2) Se forma un vector "T" que indica para cada elemento de "L" cual es la posición del mismo en el string "F".

3) Para el caracter inicial. S[0] = L[T[I]]
Para el segundo S[1]=L[T[T[I]]]
Para el tercero S[2]=L[T[T[T[i]]]]
etc. (se aplica en forma recurrente)
(En el ejemplo puede verse que es sencillísimo)

Ejemplo:

1) L='oorrdob' => F='bdooorr'

2) T=(6,4,0,1,5,2,3)

3) I=0=T0, T[T0]=T[0]=6 , L[6] = 'b' caracter inicial.
T[6]=3, L[3]='r'
T[3]=1, L[1]='o'
T[1]=4, L[4]='d'
T[4]=5, L[5]='o'
T[5]=2, L[2]='r'
T[2]=0, L[0]='o'

Con lo cual S='brodoro'

Como puede verse a partir de "L" e "I" se reconstruye en forma muy simple el string original "S". Burrows y Wheeler, como corresponde, demostraron que la transformación es reversible siempre.

Hasta aquí tenemos un método interesante que dado un string simplemente construye otro, la pregunta es ¿Para que sirve?

Supongamos que queremos comprimir una porción de texto en castellano, si la palabra 'el' aparece muchas veces en el texto obtendremos al ordenar las rotaciones un bloque en el cual la última columna de la matriz contiene muchas 'e' , junto con otras letras que suelen aparecer antes de una "l", lo que el algoritmo de block-sorting consigue al formar el string "L" es juntar en una misma porción del string a aquellos caracteres que habitualmente preceden a otros. Dado un caracter cualquiera 'ch' los caracteres que mas frecuentemente preceden a 'ch' se van agrupar en un mismo sector dentro del string "L". Para las palabras con 'q' por ejemplo obtendremos que las rotaciones que comienzan con "u" van a agrupar en "L" a todas las letras "q" del texto en un mismo sector. El string "L" por consiguiente tiende a presentar secuencias en las cuales pocos caracteres aparecen muchas veces, por lo que es ideal para comprimirlo usando un algoritmo tipo MTF.

Debemos destacar que las implementaciones del algoritmo de Block-Sorting construyen el string "L" y el entero "I" sin la necesidad de hacer todas las rotaciones y ordenarlas lo cual sería computacionalmente descabellado, la explicación aquí dada sirve perfectamente para entender el algoritmo, pero al implementarlo se puede hacer de forma mucho mas sencilla, el paper de Burrows es un buen lugar donde consultar una implementación eficiente de Block-Sorting.

Ejemplo:
S = 'que lindo que queda el queso'

```
M=      que lindo que queda el queso
        ue lindo que queda el queso q
        e lindo que queda el queso qu
         lindo que queda el queso que
        lindo que queda el queso que
        indo que queda el queso que l
        ndo que queda el queso que li
        do que queda el queso que lin
        o que queda el queso que lind
         que queda el queso que lindo
        que queda el queso que lindo
        ue queda el queso que lindo q
        e queda el queso que lindo qu
         queda el queso que lindo que
        queda el queso que lindo que
        ueda el queso que lindo que q
        eda el queso que lindo que qu
        da el queso que lindo que que
        a el queso que lindo que qued
         el queso que lindo que queda
        el queso que lindo que queda
        l queso que lindo que queda e
         queso que lindo que queda el
        ueso que lindo que queda el q
        eso que lindo que queda el qu
        so que lindo que queda el que
        o que lindo que queda el ques
         que lindo que queda el queso
```

Ordenando M...

```
M=       que lindo el queso que queda
         lindo que queda el queso que
         que lindo que queda el queso
         que queda el queso que lindo
         queda el queso que lindo que
        a el queso que lindo que qued
        da el queso que lindo que que
        do que queda el queso que lin
        e lindo que queda el queso qu
        e queda el queso que lindo qu
        eda el queso que lindo que qu
        el queso que lindo que queda
        eso que lindo que queda el qu
        indo que queda el queso que l
        l queso que lindo que queda e
        lindo que queda el queso que
        ndo que queda el queso que li
        o que lindo que queda el ques
        o que queda el queso que lind
        que lindo que queda el queso
        que queda el queso que lindo
        queda el queso que lindo que
        queso que lindo que queda el
        so que lindo que queda el que
        ue lindo que queda el queso q
        ue queda el queso que lindo q
        ueda el queso que lindo que q
        ueso que lindo que queda el q
```

las cinco primeras líneas empiezan con espacios en blanco, sin este detalle es difícil apreciar que supimos ordenar la matriz :)

S='que lindo que queda el queso'
L='aeooedenuuu ule isd eqqqq'

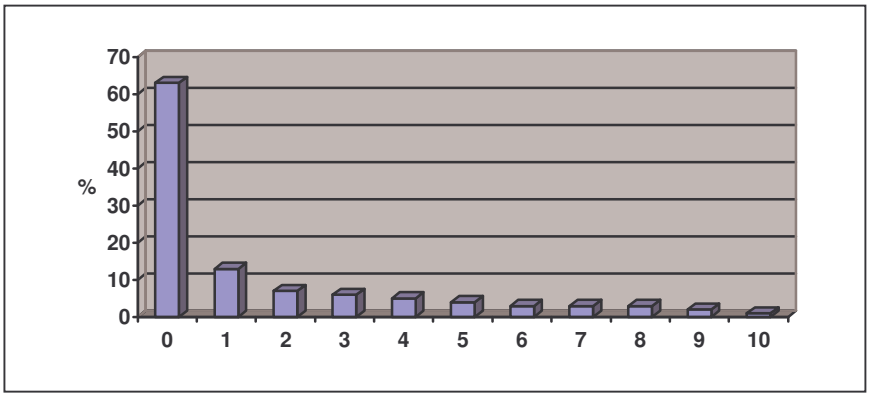
Observemos que en "L" hay una concentración de 'aeo' al comienzo, luego hay cuatro 'u' únicamente mezcladas con un blanco, y hacia el final observamos cuatro blancos y cuatro 'q' mezcladas solamente con una 'e'.

Se puede apreciar con facilidad que el string "L" se puede comprimir con gran facilidad utilizando MTF, el resultado de aplicar Block-Sorting + MTF(c/ Huffman) a esta string es dramáticamente superior al resultado de aplicar cualquier método estadístico en forma aislada.

Hacia un mejor nivel de compresión:

El esquema presentado hasta ahora puede implementarse de dos formas: Block-Sorting + MTF + Huffman adaptativo o bien Block-Sorting + MTF +Compresión Aritmética adaptativa, ambas formas son sumamente similares en cuanto al nivel de compresión siendo el mismo de alrededor de 2.52 bitsxbyte. Este nivel de compresión es excelente pero puede ser mejorado aun más. Las principales optimizaciones que se pueden hacer sobre el nivel de compresión del algoritmo residen en como comprimir la salida del algoritmo MTF. Hasta ahora se usaba un esquema estadístico adaptativo simple pero como veremos esto no es eficiente.

Realizando un estudio estadístico sobre los caracteres emitidos por el algoritmo MTF luego del block-sorting se observó que el caracter cero era el mas emitido con un porcentaje de emisión muy alto (alrededor del 65%) y que luego los demás símbolos ocurrían en porcentajes paulatinamente descendentes pero en niveles sensiblemente inferiores a los del caracter 0.



En el gráfico pueden observarse los porcentajes de emisión de los caracteres por parte del algoritmo MTF al comprimir los archivos del “Calgary Corpus”. Este patrón se observa en la gran mayoría de archivos a los cuales se aplica Block-Sorting + MTF.

En un esquema adaptativo simple, se parte de la presunción de que todos los caracteres son equiprobables, sin embargo es evidente que ese no es el caso. El esquema de compresión estadístico luego de aplicar Block-Sorting + MTF tiene, pues, que ser revisado para mejorar el nivel de compresión.

Compresión Estadística Optima luego de Block-Sorting + MTF.

En primer lugar es necesario que el algoritmo sea adaptativo ya que el algoritmo de block-sorting procesa el archivo de entradas de a bloques, la necesidad de guardar la información estadística sobre el bloque y el tiempo que insumiría realizar una doble pasada (triple o cuádruple considerando el Bs+MTF) hacen que cualquier esquema que utilice un compresor estadístico estático sea ineficiente. En segundo lugar el compresor estadístico debe usar un modelo que dé una marcada preferencia a los caracteres bajos, en especial al cero y descuide los caracteres altos que se observan en menor medida como salida del MTF.

La gran mayoría de los trabajos sobre la elección de un buen compresor estadístico para el algoritmo de Block Sorting son del australiano Peter Fenwick. Una de las curiosidades que observo Fenwick al analizar cual podría ser el mejor algoritmo de compresión estadística para la salida del MTF fue que cuanto mejor era el algoritmo de compresión utilizado peor era el resultado final! Esto se debe a que los mejores algoritmos de compresión estadística se basan en la utilización de contextos, y los contextos son destruidos por el proceso de block sorting, por lo tanto se necesita un algoritmo estadístico de orden 0, pero no cualquier algoritmo sino uno que comience con cierta ventaja respecto de aquellos que suponen todos los caracteres equiprobables. Fenwick propone dos modelos: El modelo de Shannon y El modelo estructurado.

El modelo de Shannon.

Este modelo de compresión estadística adaptativa fue denominado así por Fenwick ya que tiene algunas similitudes (aunque no muchas) a un esquema de compresión propuesto por Shannon en la década del 50. El modelo de Shannon combina 4 modelos binarios (cada uno de ellos emite cero o uno) y un modelo tradicional (para los 253 caracteres restantes) de la siguiente forma.

Si la ultima probabilidad emitida correspondía a un 0, se comienza por el modelo cero. Sino se utiliza el modelo 1.

El modelo cero o uno emite un cero si el caracter leído del archivo es un cero, en caso contrario emite un uno y pasa al modelo 2. Si el caracter leído es un 1 emite un cero, sino emite un uno y pasa al modelo 3. Así sucesivamente hasta el modelo 4, si el caracter era 4 o superior el modelo 4 emite un uno y se pasa al modelo 5 en el cual existen todos los caracteres (excepto el 0, 1, 2 y 3).

Ejemplo: Si los caracteres a comprimir fueran 0,4,2,19,3,1,1,0,6,4,2,3,1,0,0,0 y comenzando por el modelo cero.

Char	M0	M1	M2	M3	M4	M5
0	0					
4	1		1	1	1	4
2		1	1	0		
19		1	1	1	1	19
3		1	1	1	0	
1		1	0			
1		1	0			
0		0				
6	1		1	1	1	6
4		1	1	1	1	4
2		1	1	0		
3		1	1	1	0	
1		1	0			
0		0				
0	0					
0	0					

Espacio ocupado : 69,106 bits, casi 9 bytes.
Nivel de compresión : 4,375 bitsxbyte.

Lo que se emite al archivo comprimido es la probabilidad de cada código en cada modelo. Así, por ejemplo, el modelo 0 comienza con probabilidad 1/2 para el 1 y probabilidad 1/2 para el cero. El primer cero se comprime con probabilidad 1/2. Luego el 4 se comprime con probabilidad 1/3*1/2*1/2*1/2*1/252. El dos se comprime con probabilidad 1/2*2/3*1/3 etc. Las probabilidades se comprimen utilizando compresión aritmética. Con este esquema el nivel de compresión de Block-sorting + MTF + modelo-shannon alcanza 2.36 bitsxbyte. Una mejora sustancial con respecto al modelo con huffman adaptativo o aritmético adaptativo.

El modelo estructurado.

Este modelo plantea la utilización de una jerarquía de modelos de compresión aritmética de orden cero en la cual los caracteres favorecidos tienen mayor jerarquía que los caracteres menos esperados. Se comienza siempre en el primer modelo, si el caracter no figura en dicho modelo se emite un escape y se pasa al modelo inferior. La configuración del modelo consiste en decidir cuantas “capas” utilizar y que caracteres incluir en cada capa. Las pruebas de Fenwick sugieren el siguiente modelo estructurado.

Nivel	Chars
0	0
1	1
2	2-3
3	4-7
4	8-15
5	16-31
6	32-63
7	64-127
8	128-255

De esta forma el caracter 0 tiene un modelo propio en el cual solo coexiste con el escape, en el modelo siguiente se encuentra el 1 y el escape, luego se encuentran el 2, 3 y el escape y así sucesivamente. Así, si observamos el caracter 8, tenemos que emitir 3 escapes con su probabilidad respectiva en cada nivel y luego la probabilidad del 8 en el nivel 4.

Ejemplo: Si los caracteres a comprimir fueran 0,4,2,19,3,1,1,0,6,4,2,3,1,0,0,0 (el mismo conjunto de datos utilizado en el ejemplo anterior)

Caracter	Nivel 0 (0)	Nivel 1 (1)	Nivel 2 (2-3)	Nivel 3 (4 al 7)	Nivel 4 (8 al 15)	Nivel 5 (16 al 31)	Nivel 6 (32 al 63)	Nivel 7 (64 al 127)	Nivel 8 (128 al 255)
0	0								
4	Esc	Esc	Esc	4					
2	Esc	Esc	2						
19	Esc	Esc	Esc	Esc	Esc	19			
3	Esc	Esc	3						
1	Esc	1							
1	Esc	1							
0	0								
6	Esc	Esc	Esc	6					
4	Esc	Esc	Esc	4					
2	Esc	Esc	2						
3	Esc	Esc	3						
1	Esc	1							
0	0								
0	0								
0	0								

Espacio ocupado : 57,807 bits, algo más de 7 bytes.
Nivel de compresión : 3,613 bitsxbyte.

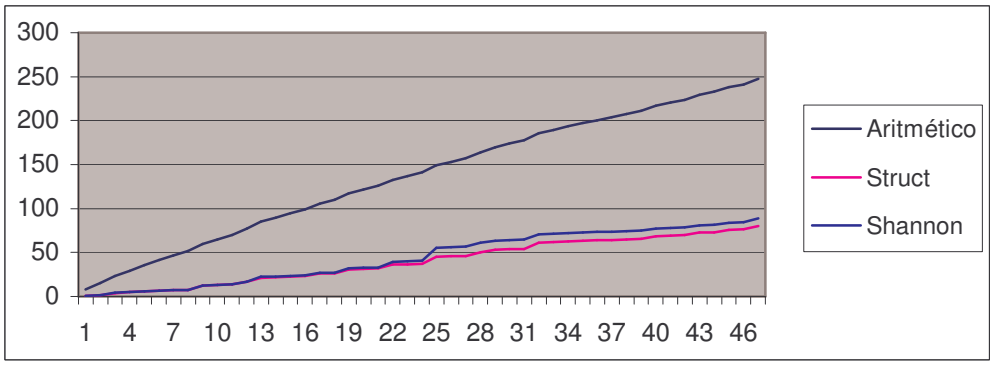
Utilizando este modelo se llega a un nivel de compresión de 2.32 bitsxbyte, el mismo nivel que presenta el algoritmo PPMC. La gran mayoría de las implementaciones de Block-sorting utilizan este tipo de modelo.

Tests:

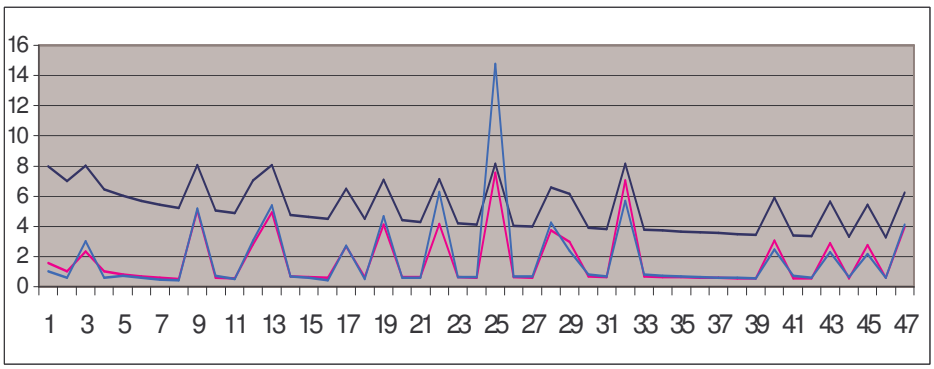
Para ilustrar se testearon tres esquemas contra un juego de datos típico de salida de un esquema de Block-Sorting + MTF. Los esquemas testeados fueron:

a) Block-Sorting + MTF + Aritmético adaptativo.
b) Block-Sorting + MTF + Modelo de Shannon.
c) Block-Sorting + MTF + Modelo Estructurado.

El juego de datos fue:
"00100000200130001020030050021004000000010010102"



Este primer gráfico ilustra la longitud emitida en bits a medida que se comprime el archivo, como puede observarse los tres modelos presentan una curva que crece cada vez menos (aunque se note poco), esto es propio de la naturaleza adaptativa de los algoritmos. Como puede observarse los modelos de Shannon y el estructurado son muy superiores al aritmético tradicional. El modelo estructurado es al final levemente superior al de Shannon.



Este gráfico muestra la cantidad de bits emitidos por cada caracter para los tres esquemas probados. Como se puede apreciar las curvas son muy similares pero los modelos "con ventaja", comienzan mucho mejor que el método de aritmético, esta tendencia se mantendrá hasta el final del bloque. El pico corresponde al 5, caracter difícil en los tres esquemas.

Por último esta es la tabla del lote testeado, la primer columna indica el caracter a comprimir, luego por cada esquema (Aritmético, Estructurado, Shannon) se indican la probabilidad con que se comprimió el caracter, la cantidad de bits que le corresponden y el total de bits acumulados hasta el momento.

C	Aritmético Adaptativo			Estructurado			Shannon		
	Probab.	Bits	Acum	Probab.	Bits	Acum	Probab.	Bits	Acum
0	0,003906	8,000000	8,000000	0,500000	1,000000	1,000000	0,500000	1,000000	1,000000
0	0,007782	7,005625	15,005625	0,666667	0,584963	1,584963	0,666667	0,584963	1,584963
1	0,003876	8,011227	23,016852	0,125000	3,000000	4,584963	0,125000	3,000000	4,584963
0	0,011583	6,431846	29,448698	0,600000	0,736966	5,321928	0,500000	1,000000	5,584963
0	0,015385	6,022368	35,471065	0,666667	0,584963	5,906891	0,600000	0,736966	6,321928
0	0,019157	5,705978	41,177043	0,714286	0,485427	6,392317	0,666667	0,584963	6,906891
0	0,022901	5,448461	46,625504	0,750000	0,415037	6,807355	0,714286	0,485427	7,392317
0	0,026616	5,231564	51,857068	0,777778	0,362570	7,169925	0,750000	0,415037	7,807355
2	0,003788	8,044394	59,901462	0,033333	4,906891	12,076816	0,037037	4,754888	12,562242
0	0,030189	5,049849	64,951311	0,727273	0,459432	12,536247	0,666667	0,584963	13,147205
0	0,033835	4,885357	69,836668	0,750000	0,415037	12,951285	0,700000	0,514573	13,661778
1	0,007491	7,060696	76,897364	0,115385	3,115477	16,066762	0,136364	2,874469	16,536247
3	0,003731	8,066089	84,963453	0,038095	4,714246	20,781007	0,003704	8,076816	24,613063
0	0,037175	4,749534	89,712987	0,666667	0,584963	21,365970	0,600000	0,736966	25,350028
0	0,040741	4,617384	94,330371	0,687500	0,540568	21,906538	0,666667	0,584963	25,934991
0	0,044280	4,497187	98,827558	0,705882	0,502500	22,409039	0,692308	0,530515	26,465506
1	0,011029	6,502500	105,330058	0,138889	2,847997	25,257036	0,142857	2,807355	29,272861
0	0,047619	4,392317	109,722376	0,684211	0,547488	25,804523	0,666667	0,584963	29,857823
2	0,007299	7,098032	116,820408	0,064286	3,959358	29,763881	0,071429	3,807355	33,665178
0	0,050909	4,295933	121,116341	0,666667	0,584963	30,348844	0,714286	0,485427	34,150605
0	0,054348	4,201634	125,317974	0,681818	0,552541	30,901385	0,625000	0,678072	34,828677
3	0,007220	7,113742	132,431717	0,060870	4,038135	34,939520	0,014118	6,146357	40,975033
0	0,057554	4,118941	136,550658	0,666667	0,584963	35,524483	0,750000	0,415037	41,390071
0	0,060932	4,036658	140,587316	0,680000	0,556393	36,080876	0,611111	0,710493	42,100564
5	0,003571	8,129283	148,716599	0,005698	7,455327	43,536203	0,000291	11,748001	53,848565
0	0,064057	3,964501	152,681101	0,666667	0,584963	44,121166	0,777778	0,362570	54,211135
0	0,067376	3,891624	156,572724	0,678571	0,559427	44,680593	0,600000	0,736966	54,948101
2	0,010601	6,559696	163,132420	0,079803	3,647414	48,328007	0,097959	3,351675	58,299776
1	0,014085	6,149747	169,282167	0,121212	3,044394	51,372401	0,072727	3,781360	62,081136
0	0,070175	3,832890	173,115057	0,645161	0,632268	52,004669	0,727273	0,459432	62,540568
0	0,073427	3,767554	176,882611	0,656250	0,607683	52,612352	0,590909	0,758992	63,299560
4	0,003484	8,164907	185,047518	0,016204	5,947533	58,559885	0,000576	10,760601	74,060161
0	0,076389	3,710493	188,758011	0,647059	0,628031	59,187916	0,750000	0,415037	74,475198
0	0,079585	3,651364	192,409375	0,657143	0,605721	59,793637	0,583333	0,777608	75,252806
0	0,082759	3,594947	196,004322	0,666667	0,584963	60,378599	0,600000	0,736966	75,989771
0	0,085911	3,541019	199,545341	0,675676	0,565597	60,944196	0,615385	0,700440	76,690211
0	0,089041	3,489385	203,034726	0,684211	0,547488	61,491684	0,629630	0,667425	77,357636
0	0,092150	3,439869	206,474595	0,692308	0,530515	62,022199	0,642857	0,637430	77,995066
0	0,095238	3,392317	209,866913	0,700000	0,514573	62,536772	0,655172	0,610053	78,605119
1	0,016949	5,882643	215,749556	0,112570	3,151101	65,687873	0,128205	2,963474	81,568593
0	0,097973	3,351472	219,101028	0,690476	0,534336	66,222210	0,769231	0,378512	81,947105
0	0,101010	3,307429	222,408456	0,697674	0,519374	66,741584	0,645161	0,632268	82,579373
1	0,020134	5,634206	228,042663	0,126623	2,981384	69,722968	0,147321	2,762961	85,342334
0	0,103679	3,269805	231,312468	0,688889	0,537657	70,260625	0,785714	0,347923	85,690257
1	0,023333	5,421464	236,733932	0,142029	2,815743	73,076368	0,093333	3,421464	89,111721
0	0,106312	3,233620	239,967551	0,680851	0,554589	73,630957	0,750000	0,415037	89,526759
2	0,013245	6,238405	246,205956	0,069444	3,847997	77,478953	0,052288	4,257388	93,784146

Como puede verse el aritmético necesitó 247 bits, para los 47 bytes de datos, un promedio de 5,23 bitsxbyte. El modelo de Shannon emitió 94 bits, a 1,99 bits x byte y por último el modelo estructurado emitió 78 bits a un promedio de 1,64 bits x byte.

Este tipo de compresores estadísticos adaptativos “con ventaja” y principalmente el modelo estructurado son sumamente útiles cuando se tiene que comprimir en forma adaptativa un conjunto de datos del cual se tiene un cierto conocimiento sobre que caracteres suelen ser más probables que otros. La elaboración de un modelo estructurado de compresión para el algoritmo de Block-Sorting permite que el nivel de compresión del algoritmo pase de 2.51 bpb a 2.34 bpb.

El algoritmo de Block-Sorting de Burrows y Wheeler es un notable aporte al campo de la compresión de datos pues permite transformar los datos de entrada de forma tal que los mismos se puedan comprimir con mayor facilidad. El paper en el cual Burrows y Wheeler describen el algoritmo por primera vez en 1993 fue publicado por la ACSC meses antes de que Witten y cía presentaran el algoritmo aritmético PPM* (el sucesor del PPMC). En su trabajo sobre el PPM* Witten y cía describen al algoritmo de Block-Sorting de Burrows y Wheeler como 'Fascinante y Sorprendente'.

•Compresión por sustitución.

La técnica de compresión por sustitución consiste en reemplazar a cada caracter o grupo de caracteres leídos desde un archivo por un grupo de bits que se obtienen de una tabla. El principio básico de la compresión por sustitución reside en que la tabla de sustitución se va construyendo a medida que se comprime el archivo, no es necesario predeterminar la tabla ni tampoco realizar estadísticas sobre la frecuencia de aparición de los caracteres en el archivo, esta es la diferencia básica con las técnicas de compresión denominadas estadísticas. Los algoritmos de sustitución resultan adaptativos por naturaleza.

Los algoritmos de compresión por sustitución son en general un tanto menos eficientes, en nivel de compresión, que la combinación de algoritmos tipo LZ77 y compresión estadística pero tienen una ventaja: solo necesitan una única pasada para comprimir un archivo, por este motivo a este tipo de algoritmos se los suele llamar también 'on the fly' porque comprimen al archivo al mismo momento que este se va leyendo, este tipo de algoritmo resulta ideal para compresión de datos en comunicaciones, por ejemplo por módem pues se pueden comprimir los datos a medida que se emiten y descomprimirlos a medida que se reciben sin necesidad de construir previamente ningún tipo de árbol o tabla, los algoritmos de compresión de las normas MNP5,V32,V42, etc utilizan este tipo de algoritmo.

El algoritmo de sustitución mas conocido es el LZ78 creado por Lempel y Ziv y modificado luego por Welch motivo por el cual también se lo conoce también como LZW (Lempel-Ziv-Welch).

Versiones.

LZ78. Lempel-ziv. (1978)
LZW. T.A.Welch. (1978)

Compresión:

LZW lee bytes de 8 bits y escribe bytes de N-bits, al comienzo N=9 y su valor puede aumentar hasta un limite fijado por la implementación, en general el limite de N oscila entre 12 y 14 bits. Los mejores resultados se logran con N=14. La primera observación que surge es bastante elemental, si el algoritmo lee bytes de 8 bits y escribe bytes de 9 bits (o mas) ¿como es que comprime los datos?

LZW utiliza una tabla que va llenando a medida que comprime los datos, la tabla tiene 2^NMAX entradas en nuestro caso 2^14 entradas, cada entrada en la tabla aloja un string de 1 o más bytes. Las primeras 256 entradas en la tabla se inicializan con los caracteres ASCII normales (0..255), las siguientes entradas (256..511) contendrán cadenas de bytes (strings) encontrados previamente en el archivo. Cuando las 512 primeras entradas de la tabla están llenas empiezan a escribirse bytes de 10 bits hasta que las entradas de la tabla llenen las primeras 1024 posiciones (0..1023) allí se comenzaran a escribir bytes de 11 bits y así sucesivamente hasta que la tabla este llena y se haya alcanzado el N máximo.

El algoritmo de compresión en pseudocódigo es de la forma:

```
STRING = Get first byte
while there is more input data
{BYTE = get next input byte
if STRING+BYTE is in code table  STRING=STRING+BYTE
   else  {output code for STRING
          if code table is not full add STRING+BYTE to code table
          STRING = BYTE
          }
}
output the code for STRING
```

Ejemplo.

Consideremos el siguiente archivo de entrada, (se toma un ejemplo altamente redundante pues de lo contrario el algoritmo LZW requeriría un archivo mas grande para poder alcanzar un nivel de compresión aceptable)

"PEP/PERSON/PEPERS/PERT"

El proceso de compresión seria de la forma:

código emitido (output)	Modificaciones a la tabla.
P	256=PE
E	257=EP
P	258=P/
/	259=/P
256	260=PER
R	261=RS
S	262=SO
O	263=ON
N	264=N/
259	265=/PE
257	266=EPE
E	267=ER
261	268=RS/
265	269=/PER
R	270=RT
T	-

El total emitido son 16 códigos de 9 bits o sea: 144 bits.
Y el archivo original tenia 22 códigos de 8 bits o sea: 176 bits.

El ejemplo es muy breve pero muestra la técnica mediante la cual comprime el algoritmo, en este caso solo se emitieron códigos de 9 bits pues no fue necesario ocupar mas de 512 posiciones en la tabla.

La tabla del LZ78 además puede llevar estadísticas sobre cuantas veces se encontró cada string (hits), o hace cuantos caracteres que no se produce cada repetición (tiempo), estos datos pueden ser útiles si se va a utilizar un mecanismo de clearing (ver mas adelante).

Descompresión.

Para descomprimir se utiliza un proceso muy similar, se van leyendo códigos de 9 bits, se los busca en la tabla, se escribe el valor hallado en la tabla y se lee otro byte, se busca el par en la tabla, si no esta se lo agrega, etc... A medida que la tabla se va llenando se cambia el número de bits que se leen del archivo, por ejemplo cuando se llena la posición 511 de la tabla el próximo código a leer será de 10 bits y así sucesivamente. Nótese que el algoritmo no necesita que la tabla creada para comprimir el archivo se guarde dentro del archivo comprimido pues reconstruye la tabla a medida que descomprime, tampoco son necesarias señales de ningún tipo para aumentar el número de bits a leer pues este número se incrementa automágicamente a medida que se llena la tabla.

El algoritmo de descompresión en pseudocódigo es (*) :

```
OLDCODE = input first code
output OLDCODE
while there is more input data
  {NEWCODE = get next input code
   STRING = translation of NEWCODE
   output STRING
   BYTE = 1st byte of STRING
   add OLDCODE+BYTE to the code table
  OLDCODE = NEWCODE
}
```

(*) Falta agregar el manejo de un caso particular que se detalla a continuación, por claridad esto no fue agregado en el algoritmo.

Caso particular de descompresión. (¡!)

Al algoritmo de descompresión ilustrado hay que agregarle un caso muy particular en el cual el descompresor tiene que ser bastante inteligente para poder descomprimir. Supongamos que estamos en una posición del archivo a comprimir de forma tal que los próximos caracteres son:

“abaabaac”

Y supongamos, también, que “aba” es el último string agregado en la tabla en posición 269, al leer “a” –“ab”-“aba” encuentra todos los strings en la tabla por lo que toma “abaa”. Como no esta en la tabla lo agrega (270=abaa) y emite el código 269. Ahora lee “a”-“ab”-“aba”-“abaa” strings que están en la tabla, “abaac” no esta por lo que lo agrega (271=abaac) emite 270. En total el compresor emite 269-270-c.

Veamos que hace con todo esto el descompresor. Al leer 269 sabe que se trata de “aba”, pero ahora, lee 270 y el código 270 aun no lo tiene porque no sabe cual era el próximo caracter. En este caso el descompresor puede deducir el código 270 y agregarlo solito en la tabla: en primer lugar 270 comienza con aba, porque el código anterior es aba y este caso solo se da cuando el compresor emite un código que acababa de generar , en segundo lugar el próximo caracter del archivo tiene que ser una “a”, ya que para que se de este caso el último caracter del código 270 tiene que ser igual al primero del 269. Por lo tanto el descompresor puede guardar 270=abaa y emitir abaa. Si este caso no se tiene en cuenta todo el proceso de compresión y descompresión deja de tener sentido!

Características principales:

- El código es sencillo y fácil de implementar.
- El nivel de compresión no es muy elevado.
- Es muy veloz si se optimiza el manejo de la tabla.
- Permite comprimir 'on the fly' con una única pasada.
- Los archivos chicos en general los expande.
- La necesidad de memoria y nivel de compresión se ajustan variando el N máximo, para N=14 (resultados óptimos) necesita unos 100Kb para la tabla.
- No se hace mas lento al comprimir archivos grandes, la velocidad de compresión es constante lo cual lo hace apto para compresión en tiempo real.

LZW no es utilizado como algoritmo principal en los compresores comerciales debido a que no comprime tanto como otros algoritmos, su uso se centra fundamentalmente en compresión de datos en comunicaciones y productos como stacker o doublespace que comprimen 'on the fly', el único compresor conocido que utiliza LZW en forma 'pura' es el conocido *compress* de Unix.

Mecanismo de Clearing.

Cuando el algoritmo LZ78 encuentra que la tabla en la cual guarda los strings observados anteriormente se llena la opción original del algoritmo consiste en expandir la longitud de los códigos emitidos en un bit duplicando de esta forma el espacio en la tabla, esta política no siempre es eficiente ya que puede ocurrir que la tabla se haya llenado con strings de escasa probabilidad que ya no se observan por lo que se desperdicia todo el espacio en la tabla, si la tabla se llena rápidamente con strings poco probables el LZ78 emitirá códigos grandes sin encontrar suficientes 'matches' en la tabla como para alcanzar un nivel de compresión aceptable.

El algoritmo es, no obstante, lo suficientemente flexible como para evitar este problema en un gran número de casos utilizando un mecanismo denominado clearing.

El mecanismo de clearing consiste en decidir, cuando se llena la tabla, si se va a continuar con códigos un bit mas grandes o bien si se va a eliminar todo el contenido de la tabla volviendo el código al tamaño de 9 bits.

El mecanismo de clearing puede ser:

Total: Si se borra toda la tabla.
Parcial: Si solo se eliminarán de la tabla algunos strings.

Un algoritmo LZ78 que utilice clearing debe destinar uno de los códigos de la tabla del 256 en adelante para que el descompresor sepa cuando ocurrió un clearing de la tabla. El compresor cada vez que se llena la tabla debe optar entre seguir comprimiendo con códigos un bit mayores o bien emitir un código de clearing. Los factores a tener en cuenta para hacer el clearing son:

- Tamaño de los códigos. (Tal vez solo conviene considerar el clearing cuando la longitud de los códigos excede los 'n'bits)
- Nivel de compresión. (Si se está comprimiendo mucho no conviene eliminar de la tabla los strings que se tienen).
- etc...

Además el compresor puede hacer un clearing total o parcial, algunos compresores solo hacen un tipo de clearing (total o parcial), otros, en cambio, definen varios códigos de clearing que le indican al descompresor el tipo de clearing efectuado en la tabla. Ejemplos de clearing parcial:

- Eliminar de la tabla los 'n' strings menos encontrados.
- Mantener en la tabla los 'n' strings mas encontrados.
- Eliminar de la tabla los strings de mas de 'n' bytes con menos de 'm' hits.
- Mantener todos los strings de menos de 'n' bytes, de los restantes eliminar aquellos con menos de 'm' hits.
- Mantener los strings con mas de 'n' hits.
- Eliminar los strings con menos de 'm' hits.
- Mantener en la tabla los 'm' strings mas recientemente encontrados. (algoritmo tipo LRU que es muy utilizado, en el LZ77 está implícito).
- Etcétera.

Al efectuar un clearing parcial puede continuarse con códigos de la misma longitud (con entradas libres en la tabla causadas por el clearing) o bien empaquetar la tabla de forma tal de bajar el tamaño de los códigos emitidos. (Se calcula en función del número de strings que sobreviven al clearing).

El tipo de clearing a efectuar puede ser decidido en función de la longitud actual de los códigos, el nivel de compresión etc.

La cantidad de opciones posibles al utilizar este mecanismo es muy amplia, el rendimiento de un LZ78 puede variar notablemente con el mecanismo utilizado.

Es muy importante, como siempre, que compresor y descompresor actúen en forma coherente.

• Algoritmos Predictores.

Versiones:

PPP Predictor. (Kasman Thomas, 1993)
Ooz-1. (KRT 1996)

Los algoritmos predictores son una rama nueva dentro de la compresión de datos. En realidad no son muy novedosos en cuanto a su técnica de compresión ya que tienen notables similitudes con los métodos tipo PPM, y con los algoritmos de sustitución tipo LZW. En común todos los algoritmos predictores se caracterizan por intentar 'adivinar' cual será el próximo caracter a leer en un archivo, si el algoritmo hace una predicción acertada (adivina) emite un solo bit indicando TRUE (acerté!), si no acierta debe emitir un FALSE seguido por el caracter leído. El descompresor debe realizar predicciones de la misma forma que el compresor para poder descomprimir, si el descompresor lee un TRUE simplemente emite el caracter predicho, de lo contrario debe leer cual es el caracter a emitir.

La predicción es realizada en función de los 'n' caracteres anteriores los cuales definen el contexto del algoritmo predictor.

El esquema básico de funcionamiento es:

> En base a un contexto de 'n' caracteres realizar una predicción sobre cual será el próximo caracter del archivo.
>
> Leer un caracter del archivo.
>
> Si el caracter leido coincide con la predicción emitir un TRUE.
>
> Si el caracter leido no coincide con la predicción emitir un FALSE seguido del caracter leido del archivo y actualizar el modelo utilizado para realizar las predicciones.

Curiosidad: Con este algoritmo puede hacerse un excelente compresor para un archivo en particular, el algoritmo debe tener orden = tamaño del archivo, si el archivo coincide con el esperado emite un solo bit (cero), sino emite un uno seguido del archivo leído. (algoritmo muy particular, comprime un archivo a un bit y al resto los expande un bit).

Versiones:

PPP Predictor: Kasman Thomas. (1993)

El algoritmo PPP predictor es una versión de Kasman Thomas sobre un algoritmo predictor básico de orden 'N', Thomas pretende que se le otorguen derechos sobre cualquier algoritmo predictor independientemente de su orden, la oficina de patentes de EE.UU. le ha negado la petición a Thomas en varias oportunidades. El texto original de Thomas es el siguiente. (Luego explicamos el algoritmo, el texto original es solo por placer...)

'A system for data compression and decompression is disclosed. A series of fixed length overlapping segments, called hashed strings, are formed from an input data sequence. a retrieved character is the next character in the input sequence after a particular hash string. A hash string relates a particular hash string to a unique address in a look-up table (LUT). An associated character for the particular hash string is stored in the LUT at the address. When a particular hash string is considered, the content of the LUT address asociated with the hash string is checked to determine wether the associated character matches the retrieved character following the hash string. If there is a match, a Boolean TRUE is output; if there is no match, a Boolean FALSE along with the retrieved character is output. Furthermore, if there is no match, then the LUT us updated by replacing the associated character in the LUT with the retrieved character'

PPP utiliza una tabla denominada Look-up-table (LUT) para guardar los caracteres que corresponden a la predicción de un cierto contexto, para acceder a la LUT utiliza una función de hashing en base al contexto.

fhash(contexto) = posición.

LUT[posición] = predicción.

Luego procede como un algoritmo predictor clásico emitiendo un TRUE o bien un FALSE seguido del caracter encontrado.

Ejemplo:

PPP de orden 1.
Sea el archivo 'ABCDABCABCDABC' = 14 x 8 = 112 bits.

El algoritmo procede de la siguiente manera:

La primera predicción la hace al azar, por ejemplo predecimos una 'J', leemos una 'A' por lo que la predicción falló, entonces emitimos un FALSE-A. Ahora predecimos con contexto 'A', como en la tabla no hay nada predecimos un Ascii(00), leemos una 'B' y sabemos que fallamos nuevamente, emitimos FALSE-B y en la tabla con contexto 'A' ponemos una 'B', así sucesivamente.

Emitimos:
FA,FB,FC,FD,FA,T,T,FA,T,T,FD,T,T,T = 9x7 (FALSE) + 7 (TRUE) = 70 bits.

La tabla del algoritmo queda de la forma:

Contexto	Predicción	Hits
A	B	3
B	C	3
C	D,A,D	0
D	A	1

Observemos que el contexto 'C' nunca hizo una predicción exitosa ya que la letra 'C' vino acompañada la primera vez por una 'D', luego por una 'A' y finalmente por una 'D' nuevamente.

Ejemplo para el fragmento “Asimov”.

> Diez meses antes, el Primer Orador habia contemplado aquellas mismas estrellas, que en ninguna otra parte eran numerosas como en el centro de ese enorme nucleo de materia que el hombre llama la Galaxia, con un sentimiento de duda; pero ahora se reflejaba una sombria satisfaccion en el rostro redondo y rubicundo de Preem Palver, Primer Orador.

Contexto	Predicción
(blanco)	m/a/e/P/O/h/c/a/m/e/q/e/n/o/p/e/t/n/c/e/c/d/e/n/d/m /q/e/h/l/G/c/u/s/d/p/a/s/r/u/s/e/r/y/r/d/P/O/.
,	(blanco)
;	(blanco)
a	n/d/b/(blanco)/d/q/s/(blanco)/r/n/s/t/(blanco)/m/(blanco) l/x/,/;/h/(blanco)/b/(blanco)/t/c/l/d
b	i/r/a/r/i
c	o/e/l/o/c/i/u
D	i
d	o/e/u/a/o/e/o
e	z/s/l/r/m/l/s/l/(blanco)/n/(blanco)/r/n/l/n/(blanco) /s/(blanco)/n/(blanco)/o/(blanco)/r/(blanco)/l /(blanco)/n/(blanco)/r/(blanco)/f/j/n/l/d/(blanco)/e/m /r
f	l/a
g	u
G	a
h	a/o
i	e/m/a/s/n/a/m/e/a/s/o/c/m
j	a
l	(blanco)/a/l/a/l/a/(blanco)/e/a/(blanco)/l/a/e /(blanco)/v
m	e/p/i/a/e/o/e/a/b/a/i/b/(blanco)/e
n	t/(blanco)/i/g/a/(blanco)/u/(blanco)/t/o/u/(blanco) /t/a/(blanco)/d
O	r
o	r/n/(blanco)/t/s/m/(blanco)/r/(blanco)/m/n/(blanco)/r m/n/s/(blanco)/n/(blanco)/r
P	r/l/a/e
p	r/a/r
q	u
r	i/(blanco)/a/(blanco)/e/a/t/a/o/m/i/e/o/a/e/i/o/e/u/e (blanco)/i/(blanco)/a/(blanco)
s	e/(blanco)/,/(blanco)/m/(blanco)/t/,/a/(blanco)/e o/a/f/t
t	e/r/e/a/r/e/i/o/i/r
u	e/n/m/c/e/n/d/n/b/n
v	e
x	i
z	(blanco)
y	(blanco)

El algoritmo emite 301 falsos y 48 trues para un total de

301x9 + 48 = 2757 bits = 345 bytes.

El nivel de compresión del PPP para este texto es pobre, en general el algoritmo PPP es mucho mas eficiente para información binaria que para información textual. Con los

textos cualquier modelo probabilístico basado en contextos lo supera ampliamente. Esto se debe a que el modelo que PPP establece para predecir textos es muy pobre, con un predictor de orden 2 el archivo se hubiese podido comprimir bastante aunque aun con los mejores modelos de predicción posibles PPP no alcanza el nivel de compresión de otros algoritmos, con un modelo super-desarrollado de predicción un PPP logro comprimir "Asimov" en 147 bytes, apenas parejo con un simple compresor basado en el método de Huffman...

Mecanismo de Tracking.

Una de las desventajas visibles de cualquier algoritmo predictor consiste en que cuando un caracter difiere de su predicción el caracter de la tabla es reemplazado por el caracter encontrado. Supongamos que en un texto determinado luego de la letra 'q' siempre se lee una 'u' salvo en una ocasión donde se lee una 's'. El algoritmo va a 'gastar' 9 bits en la 's' y luego otros 9 bits en la próxima 'u', ya que va a predecir una 's'. La solución a este problema consiste en mantener una lista ordenada de caracteres a predecir por contexto y predecir siempre el caracter que mas veces apareció luego de un determinado contexto, luego de leer cada caracter se incrementa la frecuencia del mismo dentro de su contexto y se re-ordena la lista. La desventaja de esta técnica reside en que en los casos en los cuales a un caracter lo seguía siempre uno determinado hasta que luego siempre lo sigue otro el algoritmo tarda en 'aprender' el cambio ocurrido en el archivo de entrada. En general el método con 'tracking' suele ser un tanto mejor al algoritmo predictor standard. El algoritmo Ooz-1 es un predictor de orden uno con tracking usando una matriz de 256x256 cuyas filas actualiza y ordena por cada caracter leído. Para el texto "Asimov "el algoritmo Ooz-1 hubiese comprimido algunos bytes mas aunque no demasiados, para información binaria Ooz-1 y PPP son parejos.

Hay que considerar además qué predicción hace el algoritmo cuando se produce un empate entre dos o mas símbolos de un determinado contexto al intentar una predicción, la política mas usual consiste en mantener como predicción al caracter que fue visto mas recientemente en dicho contexto.

En general el mecanismo de tracking empeora varios casos y mejora otros tantos por lo que su utilización puede o no ser beneficiosa dependiendo del archivo a comprimir. El mecanismo si debe ser considerado para predictores mas complejos o de orden superior.

Además los códigos de 9 bits emitidos cuando se produce una predicción errónea pueden ser tratados subsecuentemente para disminuir el tamaño del archivo comprimido, por ejemplo puede usarse un Huffman adaptativo para los caracteres que siguen a un FALSE de forma tal de intentar emitir menos bits. La combinación de predictor+compresor estadístico es considerada muy interesante en el mundo de la compresión de datos, existen pocos algoritmos predictores implementados hasta la fecha por lo que no se dispone de resultados sobre el rendimiento de los mismos, aunque, se supone que podrían llegar a comprimir muy bien si se realiza una buena implementación.

Los algoritmos predictores de orden superior en cuanto al manejo de la tabla pueden, al igual que el LZ78, implementar varias políticas de clearing ya que aquí es imposible guardar en memoria todas las posibles combinaciones de contextos para ordenes avanzados (para orden 3 ya se hace molesto pues serían 256x256x256 entradas en la tabla), por ello se utilizan en conjunto funciones de hashing para acceder a la tabla en base a un contexto y mecanismos de clearing para cuando la tabla se llena o bien el nivel de compresión no es aceptable. Los algoritmos predictores son muy apropiados para experimentar variantes, las decisiones a tomar son muchas y en general la implementación de este tipo de algoritmos es sencilla.

Factores a considerar al implementar predictores:

Orden de predicción (cantidad de caracteres del contexto).
Método de acceso a la tabla, función de hashing, tamaño de la tabla.
Mecanismos de clearing si la tabla se llena o si el nivel de compresión es malo,
clearing total o parcial, tipo de clearing utilizado, etc.
Utilización del mecanismo de tracking.
Compresión estadística de los códigos de 9 bits emitidos, utilización de un predictor en combinación con un algoritmo estadístico adaptativo.
etc, etc, etc....

Versiones:

LZARI H.Okimura (1989) LZSS + compresión aritmética.
LZHUF H.Yoshizaki (1988) LZSS c/hashing + Huffman.

Los compresores comerciales mas conocidos (ARJ,LHA,PKZIP, etc) no utilizan ninguno de los algoritmos descriptos anteriormente en forma individual, sino que suelen emplear una combinación de ambos, las combinaciones mas populares son las que surgen a partir del algoritmo LZ77 o LZSS.

Originalmente el LZ77 proponía utilizar un buffer de 4096 bytes, con lo cual la posición podía ser codificada en 12 bits, usando 4 bits para la longitud con lo cual se necesitaban 2 bytes para cada par ordenado posición-longitud, de esta forma solamente se codificaran a un par ordenado las repeticiones cuya longitud sea 3 o mas, ya que las repeticiones de longitud 2 ocuparían lo mismo comprimidas que descomprimidas.

En cada paso el LZSS emite o bien un caracter o bien un par ordenado (posición,longitud). Entre estos puede que por ejemplo la letra "e" sea mas frecuente que la letra "x" y que un par ordenado con longitud 3 sea mas frecuente que uno de longitud 18.

Por lo que si codificamos los caracteres y pares ordenados mas frecuentes en menos bits y los menos frecuentes en mas bits la longitud total del archivo comprimido será menor, de esto se deduce que es conveniente utilizar árboles de Huffman o compresión aritmética junto con el algoritmo LZSS, esto es lo que hacen la mayoría de los compresores actuales. Usualmente se utilizan árboles de Huffman adaptativos (ver árboles de Huffman), el problema es que la cantidad de combinaciones (posición-longitud) pueden ser muchísimas y cuando el conjunto de elementos se vuelve muy grande la compresión por Huffman deja de ser eficiente, la solución mas utilizada se basa en lo siguiente:

Se extiende el set de caracteres de 256 a aproximadamente 300 en tamaño, los primeros 256 caracteres (0 a 255) son los habituales, mientras que los caracteres (253 + n) representan a las longitudes n de un par ordenado (n>=3), por lo que el caracter 256 representa a la longitud 3. Al encontrar una longitud codificada se sabe que lo que sigue será una posición.

Este set de caracteres extendido se codifica utilizando compresión aritmética o bien código Huffman.

A la versión que utiliza compresión aritmética se la conoce como LZARI mientras que la versión que utiliza árboles de Huffman es conocida como LZHUF.

A su vez, se observa que las repeticiones se dan mas frecuentemente en las posiciones altas de la ventana (Ej: repeticiones tipo AAAAAA) Por lo que es más probable que la posición sea un número cercano a 4096 que a 1. Tomando en cuenta esto se utiliza un esquema para codificar los 6 bits mas significativos de los 12 necesarios para cada posición, para ello se utiliza una tabla de hashing con la cual se codifican estos 6 bits, los restantes 6 bits se emiten sin codificar, de esta forma se logra que los pares ordenados con posición alta , que son los mas probables, ocupen menos bits.

Un ejemplo de juguete:

A continuación mostraremos el funcionamiento de una versión de juguete de un algoritmo tipo LZHUF, la limitación esta dada por usar ventanas chicas y suponer un set de caracteres mas chico que el ASCII pero usaremos todas las variantes que suelen emplear los compresores actuales.

Esto se hace simplemente para ilustrar el funcionamiento y que las tablas y demás entren en papel.

Nuestro algoritmo se va a basar en:
- Usamos una memoria de 8 bytes.
- La ventana de inspección es de 8 bytes.

El archivo a comprimir será:

522222016716746471 6745 (22 bytes)

Primero debemos armar nuestra tabla para el Huffman adaptativo, para el set de caracteres observamos el archivo y vemos que aparecen 8 caracteres distintos (0..7) por lo que no vamos a codificar el resto (en el caso de un archivo normal no lo recorreríamos y codificaríamos los 256 caracteres y las tablas no nos entrarían en este papel) Como vimos antes vamos a extender el set de caracteres de forma tal que tengamos un caracter extra para cada posible longitud de nuestros pares ordenados (longitud, posición)

La longitud puede variar entre 2 y 8 por lo que necesitaremos 7 caracteres extra. (Tomamos solo repeticiones de por lo menos 2 caracteres).

Nuestra tabla de caracteres es de la forma:

número	Representa:
0	"0"
1	"1"
2	"2"
3	"3"
4	"4"
5	"5"
6	"6"
7	"7"
8	L2
9	L3
10	L4
11	L5
12	L6
13	L7
14	L8

Estos caracteres comunes + especiales serán codificados por Huffman adaptativo (comenzaremos suponiéndolos equiprobables e iremos cambiando el árbol a medida que leemos caracteres).

De esta forma leyendo del archivo comprimido y conociendo la tabla sabemos si vino un caracter o un par ordenado, si vino un par ordenado sabemos su longitud (depende del caracter) por lo que debemos leer del archivo su posición.

La posición puede variar desde 1 a 7 (o 0 a 6 según el caso) dado que la ultima posición de la ventana nunca puede iniciar una repetición de dos o mas caracteres, por lo que necesitaríamos 3 bits para representarla, como vimos las posiciones mas cercanas a 8 son mas probables por lo que usaremos una tabla para codificar los 3 bits de la posición de la forma:

Posición:	Codificación:
1	0000
2	0001
3	001
4	100
5	01
6	11
7	101

De esta forma cuando leemos una longitud podemos, inspeccionando la tabla, saber cual es la posición en la ventana.

Nota: La tabla se construyo usando probabilidades para cada posición calculadas a ojo de buen cubero.

Procederemos entonces a la monumental tarea de comprimir y descomprimir el siguiente archivo:

5222220167167464716745 (22 bytes)

Partimos de la situación en la cual no leímos ningún caracter por lo que todos tienen frecuencia cero y el árbol Huffman el básico.

Los códigos son: (considerando cero izquierda y uno derecha)

0= 0000 L2= 1000
1= 0001 L3= 1001
2= 0010 L4= 1010
3= 0011 L5= 1011
4= 0100 L6= 1100
5= 0101 L7= 1101
6= 0110 L8= 111
7= 0111

El primer caracter que leemos es un "5" y en la memoria no hay nada, por lo que debemos:

- Buscar el código Huffman para el "5" y emitirlo.
- Incrementar la frecuencia del "5" y rearmar el árbol Huffman.

Emitimos pues: 0101

Para cada paso mostraremos:

- El contenido de la ventana-memoria.
- El contenido de la ventana de inspección.
- El código que se emite en bits.

Además mostraremos una tabla con la frecuencia de aparición de cada caracter y su código Huffman en cada paso.

52222201671674646167 45 (22 bytes)

Paso	Memoria	Inspección	Emite:	Corresponde	Bits
1	–	52222201	0101	"5"	4
2	5	22222016	00010	"2"	5
3	52	22220167	01	"2"	2
4	522	22201671	001000101	(7,2)	9
5	52222	20167167	1	"2"	1
6	522222	01671674	0000000	"0"	7
7	5222220	16716746	00000000	"1"	8
8	52222201	67167464	0000010	"6"	7
9	22222016	71674646	0000100	"7"	7
10	22220167	16746461	000001011	(6,3)	9
11	20167167	46461674	0000001	"4"	7
12	01671674	64616745	1000	"6"	4
13	16716746	4616745x	011011	(6,2)	6
14	71674646	16745x	000000010001	(2,4)	12
15	46461674	5x	0100	"5"	4
16	64616745	x	–	EOF	

Total de Bits comprimidos: 92 = 12 bytes.

Tablas Huffman

Chr	Paso1	Paso2	Paso3	Paso4	Pasos5-6	Paso7	Paso8
0	0/0000	0/00000	0/00000	0/00000	0/0000000	1/0001	1/0001
1	0/0001	0/00001	0/00001	0/00001	0/0000001	0/00000000	1/001
2	0/0010	0/00010	1/01	2/1	2/1 /3/1	3/1	3/1
3	0/0011	0/00011	0/000100	0/000100	0/0000010	0/00000001	0/0000000
4	0/0100	0/00100	0/000101	0/000101	0/0000011	0/00000010	0/0000001
5	0/0101	1/1	1/1	1/01	1/001	1/001	1/010
6	0/0110	0/00101	0/000110	0/000110	0/0000100	0/00000011	0/0000010
7	0/0111	0/00110	0/000111	0/000111	0/0000101	0/00000100	0/0000011
L2	0/1000	0/00111	0/001000	0/001000	1/01	1/01	1/011
L3	0/1001	0/01000	0/001001	0/001001	0/0000110	0/00000101	0/00001000
L4	0/1010	0/01001	0/001010	0/001010	0/0000111	0/00000110	0/00001001
L5	0/1011	0/01010	0/001011	0/001011	0/000100	0/00000111	0/00001010
L6	0/1100	0/01011	0/001100	0/001100	0/000101	0/0000100	0/00001011
L7	0/1101	0/0110	0/001101	0/001101	0/000110	0/0000101	0/0000110
L8	0/111	0/0111	0/00111	0/00111	0/000111	0/000011	0/0000111

Chr	Paso8	Paso9	Paso10	Paso11	Paso12	Paso13
0	1/0001	1/001	1/0001	1/0001	1/0001	1/0001
1	1/001	1/01	1/001	1/001	1/001	1/0010
2	3/1	3/11	3/11	3/11	3/11	3/11
3	0/0000000	0/000000	0/0000000	0/0000000	0/0000000	0/0000000
4	0/0000001	0/000001	0/0000001	0/0000001	1/010	1/0011
5	1/010	1/1000	1/010	1/010	1/011	1/0100
6	0/0000010	1/1001	1/011	1/011	1/1000	2/10
7	0/0000011	0/0000100	1/100	1/1000	1/1001	1/0101
L2	1/011	1/101	1/101	1/1001	1/1010	1/0110
L3	0/00001000	0/0000101	0/0000010	1/101	1/1011	1/0111
L4	0/00001001	0/0000110	0/0000011	0/0000010	0/0000001	0/0000001
L5	0/00001010	0/0000111	0/0000100	0/0000011	0/0000010	0/0000010
L6	0/00001011	0/000100	0/0000101	0/0000100	0/0000011	0/0000011
L7	0/0000110	0/000101	0/0000110	0/0000101	0/000010	0/000010
L8	0/0000111	0/00011	0/0000111	0/000011	0/000011	0/000011

Chr	Paso14	Paso15	FINAL
0	1/00001	1/00001	1/00001
1	1/0001	1/00010	1/0001
2	3/11	3/001	3/11
3	0/00000000	0/0000000	0/0000000
4	1/0010	1/00011	1/0010
5	1/0011	1/0100	2/011
6	2/011	2/10	2/100
7	1/0100	1/0101	1/0011
L2	2/10	2/11	2/101
L3	1/0101	1/0110	1/0100
L4	0/00000001	1/0111	1/0101
L5	0/00000010	0/0000001	0/0000001
L6	0/00000011	0/00000100	0/00000100
L7	0/0000010	0/00000101	0/00000101
L8	0/0000011	0/0000011	0/0000011

Como se puede observar el archivo se comprimió de 22 a 12 bytes.

Siguiendo paso a paso el algoritmo se puede observar como mejora al incluir las longitudes como caracteres especiales pues si aparecen muchas repeticiones de igual longitud cada vez necesitamos menos bits para el par ordenado, en el ejemplo tenemos 2 repeticiones de longitud=2, para la primera usamos 9 bits pero para la segunda solo 6, análogamente al tener codificadas las posiciones necesitamos menos bits para las posiciones mas probables. La codificación de las posiciones se hace por tabla fija y no adaptativa para seguir el esquema del algoritmo original en el cual es incomodo hacer un Huffman adaptativo para las 4096 posibles posiciones (directamente se hashean los 6 bits de orden superior de acuerdo a criterios preestablecidos)

El algoritmo nuestro solo comprimió 22 caracteres, pero se puede notar que la eficiencia del algoritmo mejora para archivos mas grandes ya que el Huffman adaptativo necesita una cierta cantidad de bytes de "entrenamiento" antes de empezar a rendir en forma optima. Este mismo esquema LZSS + Hashing + Huffman es utilizado por un gran número de compresores comerciales y de distribución libre: ARJ, LHA, PkZIP, etc.

Versiones:

Julian Seward. (1996).

Durante 1996, un señor llamado Julian Seward, dedicó parte de su tiempo a realizar un algoritmo de compresión de datos que utilizara las técnicas mas modernas descubiertas hasta el momento. Seward decidió implementar el algoritmo de Block-Sorting de Burrows y Wheeler, aplicarle al resultado el algoritmo MTF de Fenwick y al resultado (la famosa lista "R") comprimirlo por compresión aritmética. El compresor desarrollado se denominó Bzip y superó en nivel de compresión ampliamente a todos los compresores comerciales existentes hasta el momento en un tiempo de ejecución comparable al del Arj (el pkzip es mas veloz que el Bzip pero comprime realmente mucho menos). Bzip, de distribución libre y gratuita (se distribuyen los fuentes) se convirtió rápidamente en un compresor muy utilizado ya que en tiempos de ejecución muy buenos lograba un nivel de compresión sorprendente. Actualmente se estudiaba la posibilidad de que Bzip reemplace al Gzip de GNU como standard de compresión de datos para Linux.

Las versiones mas avanzadas de los compresores aritméticos, fundamentalmente algoritmos muy nuevos como el PPMZ, PPMQ, PPMD*, PPMDet y otros (que siguen la línea del PPMc) superan levemente en nivel de compresión al Bzip aunque tardan centurias en comprimir. Considerando el nivel de compresión puro los monstruos trituradores aritméticos siguen ganando pero si consideramos la relación nivel de compresión - tiempo de ejecución Bzip es sin dudas el mejor algoritmo de compresión surgido hasta la fecha. (*)

(*) Nota del autor: Para Julio de 1997 Bzip ya había sido superado por otro compresor utilizando block-sorting usando en el compresor estadístico un modelo estructurado, es imposible mantener este apunte al día sobre todo si por cada compresor nuevo hay que fotocopiar mas de 100 páginas :), les ruego que se mantengan informados, el mundo no para de girar.

•Compresión por palabras.

Hasta ahora todos los algoritmos revisados con excepción del DMC procesan el archivo de entrada tomando de a un byte, existen otros algoritmos originalmente diseñados para compresión de textos que procesan al archivo de entradas en palabras asignando una probabilidad de ocurrencia a cada palabra en base a su frecuencia y comprimiendo en base a dicha probabilidad. Sorprendentemente para volúmenes de información grandes este tipo de algoritmos no solo comprime bien los textos sino también información binaria de cualquier tipo.

Versiones:

Moffat, Sharman, Zobel. (1994)

El algoritmo Huffword es un compresor de archivos por palabras que para comprimir utiliza el método de Huffman. El algoritmo utiliza dos pasadas sobre el archivo original, en la primera crea la tabla de palabras y la tabla de no-palabras mientras que en la segunda pasada comprime.

En la primera pasada Huffword construye dos tablas, una tabla de palabras y una tabla de no-palabras, las no palabras son los signos de puntuación, uno o mas espacios en blanco etc. Para comprimir y descomprimir el algoritmo va a suponer que se alternan en el texto palabras y no-palabras de forma tal que se construirán dos árboles de huffman independientes y que se alternaran en su uso al comprimir o descomprimir un archivo.

Ejemplo:

“la mejor forma de lograr la felicidad. Esa es la cuestión.”

El algoritmo genera una tabla con las palabras:

la/mejor/forma/de/lograr/felicidad/Esa/es/cuestión.

Y una tabla de no-palabras:

/ /./. / (blanco, punto y punto-blanco)

Luego se generan los árboles de huffman correspondientes a ambas tablas y finalmente el texto se comprime. Ambos arboles deben guardarse en el archivo comprimido.

Una desventaja que presenta este método es el tratamiento de los números, supongamos un texto en el cual las páginas están numeradas, en la tabla de palabras vamos a tener una enorme cantidad de palabras que solo ocurren una vez, una por cada número de página. Para evitar este problema se recurre a la idea de limitar el tamaño de las palabras numéricas a una cierta cantidad de dígitos (por ejemplo 4) para lograr esto es necesario inventar una palabra ficticia: la palabra vacía que permite switchear al compresor y al descompresor entre las tablas de palabras y no-palabras libremente. La palabra vacía se trata como una palabra mas con la diferencia de no generar nada al descomprimirse.

El número 1234671 se representaría como 1234/palabra_vacia/671 al descomprimir el 1234 se busca en la tabla de palabras, la palabra vacía se busca en la tabla de no-palabras y el 671 nuevamente en la tabla de palabras. (recordar que siempre se supone que se alternan palabras y no-palabras).

El algoritmo Huffword tiene como desventaja la necesidad de almacenar la tabla de palabras y no-palabras en el archivo comprimido por lo que su rendimiento comienza a ser satisfactorio para archivos relativamente grandes. Si el archivo es lo suficientemente grande Huffword comprime bien tanto textos como archivos binarios lo cual resulta llamativo.

Ejemplo para el fragmento (Asimov) :

> Diez meses antes, el Primer Orador habia contemplado aquellas mismas estrellas, que en ninguna otra parte eran numerosas como en el centro de ese enorme nucleo de materia que el hombre llama la Galaxia, con un sentimiento de duda; pero ahora se reflejaba una sombria satisfaccion en el rostro redondo y rubicundo de Preem Palver, Primer Orador.

Tabla de Palabras:

Palabra	Frecuencia	Código
Diez	1	000000
meses	1	000001
antes	1	000010
el	4	1011
Primer	2	1101
Orador	2	1100
había	1	000011
contemplado	1	000100
aquellas	1	000101
mismas	1	000110
estrellas	1	000111
que	2	1100
en	3	1111
ninguna	1	001000
otra	1	001001
parte	1	001010
eran	1	001011
tan	1	001100
numerosas	1	001101
como	1	001110
centro	1	001111
de	3	1010
ese	1	010000
enorme	1	010001
núcleo	1	010010
materia	1	010011
hombre	1	010100
llama	1	010101
la	1	010110
Galaxia	1	010111
con	1	011000
un	1	011001
sentimiento	1	011010
duda	1	011011
pero	1	011100
ahora	1	011101
se	1	011110
reflejaba	1	011111
una	1	100000
sombría	1	100001
satisfacción	1	100010
rostro	1	100011
redondo	1	100100
rubicundo	1	100101
Preem	1	100110
Palver	1	100111

Tabla de no-palabras:

Palabra	Frecuencia	Código

(blanco)	51	0
(coma) + (blanco)	4	10
(punto y coma blanco)	1	110
(punto)	1	111

En primer lugar es evidente que el fragmento es demasiado pequeño como para observar el nivel de compresión del algoritmo por lo que se incluye únicamente como forma de entender el funcionamiento del algoritmo.

El archivo original ocupaba:

349 bytes.

El archivo comprimido ocuparía:

40x6 + 6x4 + 6x4 + 4x4 + 51x1 + 4x2 + 2x3 = 369 bits = 46 bytes.

Pero, lamentablemente hay que guardar la tabla de palabras y la tabla de no-palabras con la frecuencia de cada una lo cual ocupa:

Tabla de Palabras ocupa:

306 bytes.

Tabla de no-palabras ocupa:

11 bytes.

Total del archivo: 306 + 11 + 46 = 363 bytes.

Como puede verse un párrafo pequeño en general se expande al usar Huffword, el párrafo que ilustramos sin embargo sirve para notar a partir de que tamaño de párrafo el algoritmo comienza a ser efectivo, para el párrafo “Asimov” el archivo comprimido ocupa apenas un poco mas que el archivo original, para párrafos de mayor extensión la versión comprimida empezará a ocupar cada vez menos espacio en relación al original. Esto se debe a que las palabras comenzarán a repetirse y luego de haber recolectado un buen número de palabras la probabilidad de que en un párrafo aparezcan nuevas palabras se reduce y cada palabra que ya fue incorporada al léxico se comprime en unos pocos bits incrementando notablemente el nivel de compresión.

El algoritmo presenta algunas otras ventajas. Es autosincronizante y sumamente veloz ya que comprime y descomprime los archivos de a palabras y no de a bytes. Y además es ideal para indexar archivos comprimidos ya que se puede tener un acceso relativamente rápido a cualquier palabra del texto pues todas las palabras del texto comprimido pueden encontrarse en la tabla junto con el código binario que la representa.

La utilización principal de Huffword es la compresión de volúmenes de texto grandes a los cuales se quiera tener acceso random aun estando el texto original comprimido. Numerosos sistemas de almacenamiento de información textual usan Huffword, como por ejemplo el sistema Mg (Witten,Bell,Moffat) que se consigue gratuitamente en internet.

Versiones:

GNU. 1993-1994.

Gzip es una versión de LzHuf de GNU, el algoritmo reemplazo rápidamente al compress como standard de compresión de datos para Linux, últimamente se evaluaba la posibilidad de reemplazar el standard Gzip por Bzip pero Bzip es todavía un tanto lento como para que dicho cambio sea claramente ventajoso. Gzip es una versión de Lzhuf modificada para mejor compresión y también mejor velocidad.

Gzip procesa el archivo de entrada en bloques de 64Kb por lo que se dice que el algoritmo de compresión de Gzip es de tipo semi-estático, esto quiere decir que realiza una sola pasada del archivo de entrada (como si fuera dinámico) pero cada bloque leído del archivo es procesado a su vez en forma estática. La ventana memoria es de 64Kb, con lo cual todo el bloque es tenido en cuenta para encontrar repeticiones.

Gzip utiliza sobre cada bloque de 64Kb una combinación de LZ77 y Huffman. Por cada byte leído Gzip hashea los tres primeros caracteres con lo cual obtiene un puntero a una lista enlazada (todas las colisiones se enlazan en una lista aunque si se llega a un tamaño limite se eliminan las primeras n colisiones), la lista contiene todas los offsets dentro del bloque donde se habían encontrado previamente los tres caracteres. La lista es recorrida secuencialmente en búsqueda de la repetición de mayor longitud. Si se encuentra una repetición se la codifica como un par ordenado (longitud + offset en el bloque) . Se encuentre o no una repetición el offset es agregado a la lista para que sea encontrado por una futura ocurrencia de dichos caracteres.

Gzip en la primera pasada sobre el bloque crea un árbol de Huffman para los caracteres y las longitudes y otro árbol, independiente para los offsets. Como las longitudes se codifican en el mismo árbol que los caracteres al igual que en Lzhuf es sencillo para el descompresor distinguir a un caracter de un par ordenado, el descompresor sabe que siempre luego de una longitud viene un offset. En la segunda pasada Gzip almacena una representación comprimida de ambos árboles y luego comprime el bloque usando dichos árboles. Gzip es un algoritmo goloso (greedy), ya que siempre que encuentre una repetición para una tripla de caracteres (de longitud 3 o mayor) va a comprimirla como un par ordenado longitud-offset. Esto no siempre es recomendable ya que a veces es conveniente saltear una repetición para luego encontrar otra de mayor longitud. El algoritmo puede configurarse para optimizar el nivel de compresión en cuyo caso Gzip tiene en cuenta la posibilidad de no emitir un par ordenado para una repetición si con ello logra una mejora en el nivel de compresión. La versión no-golosa de Gzip alcanza un nivel de compresión promedio de 2.70 bits x byte.

Ejemplo (El mismo que se uso en el algoritmo Lzhuf):

“522222016716746 4716745”

El bloque es de la forma (indicando los offsets)

5	**2**	**2**	**2**	**2**	**2**	**0**	**1**	**6**	**7**	**1**	**6**	**7**	**4**	**6**	**4**	**7**	**1**	**6**	**7**	**4**	**5**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

En la primer pasada el algoritmo genera la siguiente tabla:

Chars	Pointers
522	0
222	1,2,3
220	4
201	5
016	6
167	7,10,17
671	8
716	9,16
674	11,18
746	12
464	13
647	14
471	15
745	19

Además se generan los árboles Huffman para las siguientes tablas.

Tabla de longitudes, caracteres y códigos.

Char	Freq
7	1/101
6	2/010
5	2/011
4	2/100
2	5/00
1	1/1100
0	1/1101
l3	1/1110
l5	1/1111

Tabla de Offsets

Offset	Freq
7	1/0
9	1/1

Una vez construidos los dos arboles Huffman el string comprimido es:

“5222220167167464716745”

011-00-00-00-00-00-1101-1100-010-101-**11100**-100-010-100-**11111**-011

Total: 49 bits. Alrededor de 7 bytes más lo que ocupe la representación binaria de los dos árboles (que no es mucho). Como puede verse en este caso Gzip es en general superior a la implementación dinámica tradicional de Lzhuf (92 bits) aunque un ejemplo muy chico no es representativo las pruebas realizadas verifican la superioridad de Gzip en la mayoría de los casos.

El algoritmo Gzip es en definitiva una implementación muy buena de Lzhuf semi-estático por bloques usando Lz77 con hashing y Huffman. Muchos compresores comerciales usan técnicas realmente muy similares, ya que el balance entre nivel de compresión y velocidad es muy útil para el uso tradicional que se le da a un compresor (Gzip es un compresor usable, no es un algoritmo de competición). Es de destacar que Gzip es un algoritmo de distribución libre y gratuita, los fuentes pueden conseguirse sin problemas en internet o en la distribución misma de Gzip.

El algoritmo LZRW1 es otra versión de Lzhuf aunque optimizado para lograr mayor velocidad de compresión, esto va, como es de suponer, en detrimento del nivel de compresión del algoritmo.

Básicamente LZRW1 es muy similar al algoritmo utilizado en Gzip y puede describirse simplemente mencionando las diferencias entre ambos.

LZRW1 también hashea los tres caracteres próximos del texto pero no mantiene una lista para las colisiones, la tabla usada para guardar los offsets contiene solo una posición por tripla de caracteres, con esto se evita el tener que actualizar la lista y además recorrer la lista cada vez que se quiere buscar un string ya observado.

LZRW1 solo actualiza la tabla cuando encuentra un string que machea, las triplas de caracteres que hashean a una misma posición pero no corresponden a un string ya encontrado se descartan. Estos dos cambios fundamentales se acompañan de un manejo optimizado de la tabla, la función de hashing y el procesamiento del archivo con lo cual se logra un pequeño incremento adicional en la velocidad de compresión del algoritmo.

En general el algoritmo LZRW1 es considerado el mas rápido de los algoritmos de compresión que alcanzan un nivel de compresión aceptable. LZRW1 comprime en promedio a unos 4.46 bits x byte pero en velocidad es muy superior a casi todos los demás algoritmos.

Cuadro Resumen de Algoritmos.

Nombre	Año	Autor(es)	Nivel de Compresión (bits/byte)	Velocidad Compres. (Mbitsxsec)	Velocidad de Descomp. (Mbitsxsec)	Obs.
Huffman	1952	D.A.Huffman	4.99	30	58	
Huffman adaptativo.	1952	D.A.Huffman	5.3	48	48	
Shannon Fano.	1952	C.Shannon Fano	5.25	40	77	Versión estática.
Compresión Aritmética.	1985	Witten Neal Clearly	4.90	6	5	Modelo de Orden-0.
PPMC	1988	Witten Moffat Bell	2.34	6	6	Orden 5.
PPM*	1995	Witten Clearly Teahan.	2.28	5.2	5.2	
MTF + Huffman.	1995	P.Fenwick.	3.75	31	31	
LZRW1	1977	Ziv Lempel	4.46	90	200	Optimizado para velocidad
LZ78 compress	1978	Ziv Lempel Welch	3.64	30	65	LZW clásico.
LZHuff	1984	Varios.	3.05	26	23	ARJ.
LZAri	1984	Varios.	3.02	12	14	
Huffword	1994	Moffat. Sharman. Zobel.	3.56	10	55	
PPP	1993	K. Thomas	6.86	305	305	
Ooz-1	1996	KWR.	6.75	150	150	
DMC	1987	Cormack. Horspool.	2.58	2	2	
Gzip	1995	GNU.	2.70	9	110	
Bzip	1996	Burrows Wheeler. Fenwick. J.Seward.	2.32			Block-Sorting + MTF + Compresión Aritmética. Modelo Estructurado.

•El mejor algoritmo.

Como hemos visto la compresión aritmética logra, a partir de una cierta probabilidad representar a dicha probabilidad con la longitud mínima de acuerdo a la entropía mas un overhead mínimo. El reto consiste, en lograr mejores modelos para conseguir mejores predicciones, el mejor modelo construido hasta la fecha se basa en predicciones realizadas por seres humanos. El conocimiento del lenguaje y la capacidad deductiva de las personas no puede, de momento, ser alcanzado por ninguna máquina por lo que un modelo probabilístico generado por personas comunes es un buen parámetro indicador sobre el limite máximo al cual se puede comprimir un determinado texto.

Ejemplo:

Tomemos de nuevo el fragmento “Asimov” y veamos que ocurre cuando el modelo probabilístico se basa en las predicciones generadas por un grupo de personas.

> Diez meses antes, el Primer Orador habia contempleado aquellas mismas estrellas, que en ninguna otra parte eran numerosas como en el centro de ese enorma nucleo de materia que el hombre llama la Galaxia, con un sentimiento de duda; pero ahora se reflejaba una sombria satisfaccion en el rostro redondo y rubicundo de Preem Palver, Primer Orador.

El test fue realizado con un grupo de 20 personas, cada una de ellas debía anotar en un formulario especial cual pensaba que iba a ser el próximo caracter del texto, luego dicho caracter se daba a conocer y el proceso se repetía. El texto se iba anotando en una pizarra de forma tal que todas las personas del grupo conocían todos los caracteres anteriores en el momento de hacer la predicción. Curiosamente si la cantidad de caracteres “publicados ”se limitara a los 5 o 6 caracteres anteriores los resultados serían muy parecidos ya que es imposible “borrar” la memoria de los seres humanos (podría hacerse pero no se conseguirían voluntarios...). Luego para comprimir el texto se tomo la probabilidad de ocurrencia del caracter de acuerdo a las predicciones del grupo (por ejemplo si 12 de las 20 personas predijeron dicho caracter la probabilidad del mismo es de 12/21). El caracter extra es un escape necesario por si se diera la aparición de algún caracter que nadie predijo, en cuyo caso se emite un escape y luego se comprime el caracter con probabilidad 1/256. Las probabilidades se comprimían, obviamente con compresión aritmética.

Los resultados obtenidos por el test son muy interesantes ya que no existen experiencias anteriores sobre el calculo del limite máximo de compresión para textos en castellano, para textos en Ingles Shannon estimo en la década del 50 que el limite teórico de compresión era de entre 0,6 y 1,5 bits por byte.

Para el fragmento Asimov se registro un nivel de compresión de 1,35 bits x byte.

La última parte del texto resulto particularmente difícil, principalmente el nombre Preem Palver, compuesto por dos palabras que son absolutamente infrecuentes e inhallables en el idioma castellano. Por lo que deberíamos estimar que el promedio real debería oscilar entre 0,85 y 1,23 bits por byte.

Nivel teóico maximo de compresión. Para el fragmento Asimov.

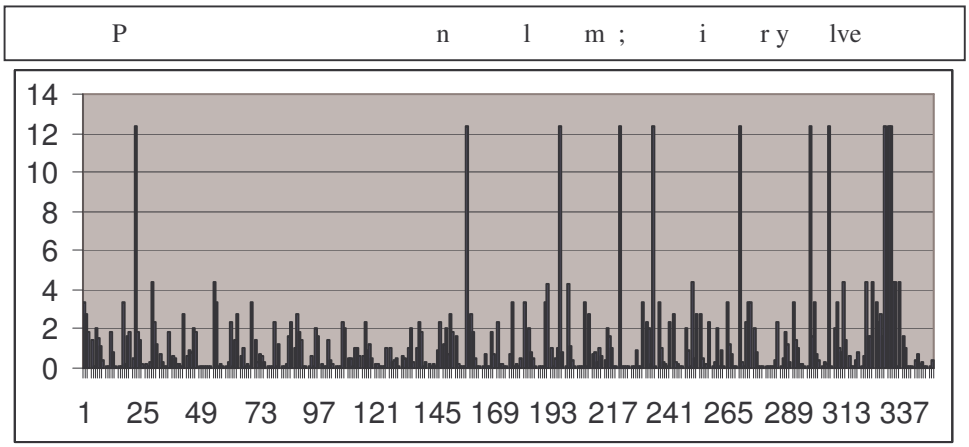
1,35 bits x byte.

Tabla de Resultados para el fragmento “Asimov”

C	P	Bits	C	P	Bits	C	P	Bits	C	P	Bits	C	P	Bits	C	P	Bits	C	P	Bits	C	P	Bits
D	0,1	3,39	l	0,95	0,07	t	0,33	1,58	e	0,9	0,14	m	0,95	0,07	a	0,19	2,39	l	0,43	1,222	i	0,48	1,07
i	0,14	2,81	a	0,95	0,07	r	0,86	0,22		0,52	0,93	a	0,7	0,51	h	0,14	2,81		0,81	0,305	m	0,9	0,14
e	0,29	1,81	d	0,9	0,14	a	0,9	0,14	e	0,19	2,39		0,5	1	o	0,81	0,3	r	0,1	3,392	e	0,95	0,07
z	0,38	1,39	o	0,95	0,07		0,95	0,07	s	0,43	1,22	l	0	12,3	r	0,86	0,22	o	0,38	1,392	r	0,95	0,07
	0,9	0,14		0,9	0,14	p	0,38	1,39	e	0,24	2,07	a	0,55	0,86	a	0,95	0,07	s	0,48	1,07		0,76	0,39
m	0,25	2	a	0,05	4,39	a	0,76	0,39		0,62	0,69		0,95	0,07		0,95	0,07	t	0,86	0,222	C	0,62	0,69
e	0,35	1,51	q	0,1	3,39	r	0,86	0,22	e	0,14	2,81	G	0,05	4,32	s	0,24	2,07	r	0,86	0,222	r	0,86	0,22
s	0,45	1,15	u	0,9	0,14	t	0,9	0,14	n	0,29	1,81	a	0,45	1,15	e	0,52	0,93	o	0,9	0,144	a	0,81	0,3
e	0,75	0,42	e	0,86	0,22	e	0,9	0,14	o	0,33	1,58	l	0,75	0,42		0,9	0,14		0,9	0,144	d	0,95	0,07
s	0,95	0,07	l	0,95	0,07		0,95	0,07	r	0,86	0,22	a	0,9	0,15	r	0,05	4,39	r	0	12,39	o	0,95	0,07
	0,9	0,15	l	0,9	0,14	e	0,19	2,39	m	0,9	0,14	x	0,95	0,07	e	0,71	0,49	e	0,33	1,585	r	0,95	0,07
a	0,29	1,81	a	0,81	0,3	r	0,24	2,07	e	0,95	0,07	i	0,95	0,07	f	0,14	2,81	d	0,1	3,392	.	0,76	0,39
n	0,57	0,81	s	0,19	2,39	a	0,71	0,49		0,95	0,07	a	0,95	0,07	l	0,14	2,81	o	0,62	0,692			
t	0,95	0,07		0,95	0,07	n	0,71	0,49	n	0	12,4	,	0,1	3,32	e	0,71	0,49	n	0,76	0,392			
e	0,95	0,07	m	0,38	1,39		0,95	0,07	u	0,14	2,81		0,95	0,07	j	0,86	0,22	d	0,95	0,07			
s	0,95	0,07	i	0,14	2,81	t	0,48	1,07	c	0,29	1,81	c	0,15	2,74	a	0,86	0,22	o	0,81	0,305			
,	0,1	3,39	s	0,67	0,58	a	0,48	1,07	l	0,71	0,49	o	0,6	0,74	b	0,19	2,39		0,86	0,222			
	0,86	0,22	m	0,48	1,07	n	0,67	0,58	e	0,95	0,07	n	0,55	0,86	a	0,9	0,14	y	0	12,39			
e	0,33	1,58	a	0,9	0,14		0,67	0,58	o	0,95	0,07		0,81	0,3		0,81	0,3		0,9	0,144			
l	0,29	1,81	s	0,86	0,22	n	0,19	2,39		0,9	0,14	u	0,48	1,07	u	0,24	2,07	r	0,24	2,07			
	0,71	0,49		0,95	0,07	u	0,52	0,93	d	0,62	0,69	n	0,67	0,58	n	0,9	0,14	u	0,1	3,392			
P	0	12,4	e	0,1	3,39	m	0,43	1,22	e	0,9	0,14		0,76	0,39	a	0,52	0,93	b	0,48	1,07			
r	0,29	1,81	s	0,38	1,39	e	0,71	0,49		0,95	0,07	s	0,24	2,07		0,9	0,14	i	0,57	0,807			
i	0,38	1,39	t	0,81	0,3	r	0,86	0,22	m	0,29	1,81	e	0,33	1,58	s	0,1	3,39	c	0,05	4,392			
m	0,86	0,22	r	0,62	0,69	o	0,86	0,22	a	0,62	0,69	n	0,48	1,07	o	0,43	1,22	u	0,38	1,392			
e	0,86	0,22	e	0,67	0,58	s	0,95	0,07	t	0,19	2,39	t	0,95	0,07	m	0,62	0,69	n	0,67	0,585			
r	0,9	0,14	l	0,81	0,3	a	0,95	0,07	e	0,86	0,22	l	0,95	0,07	b	0,95	0,07	d	0,9	0,144			
	0,81	0,3	l	0,95	0,07	s	0,95	0,07	r	0,86	0,22	m	0	12,4	r	0,95	0,07	o	0,95	0,07			
O	0,05	4,39	a	0,95	0,07		0,48	1,07	i	0,95	0,07	i	0,95	0,07	i	0	12,4		0,76	0,392			
r	0,19	2,39	s	0,95	0,07	c	0,48	1,07	a	0,95	0,07	e	0,95	0,07	a	0,81	0,3	d	0,57	0,807			
a	0,43	1,22	,	0,19	2,39	o	0,81	0,3		0,62	0,69	n	0,95	0,07		0,9	0,14	e	0,9	0,144			
d	0,62	0,69		0,9	0,14	m	0,76	0,39	q	0,1	3,39	t	0,95	0,07	s	0,19	2,39		0,67	0,585			
o	0,81	0,3	q	0,43	1,22	o	0,71	0,49	u	0,95	0,07	o	0,95	0,07	a	0,1	3,39	P	0,05	4,392			
r	0,9	0,14	u	0,95	0,07		0,95	0,07	e	0,86	0,22		0,9	0,14	t	0,1	3,39	r	0,33	1,585			
	0,95	0,07	e	0,95	0,07	e	0,67	0,58		0,95	0,07	d	0,52	0,93	i	0,24	2,07	e	0,33	1,585			
h	0,29	1,81		0,86	0,22	n	0,71	0,49	e	0,71	0,49	e	0,9	0,14	s	0,57	0,81	e	0,05	4,392			
a	0,67	0,58	e	0,33	1,58		0,95	0,07	l	0,1	3,32		0,9	0,14	f	0,95	0,07	m	0,1	3,392			
b	0,71	0,49	n	0,19	2,39	e	0,48	1,07		0,8	0,32	d	0,1	3,39	a	0,95	0,07		0,19	2,392			
i	0,86	0,22		0,48	1,07	l	0,24	2,07	h	0,25	2	u	0,19	2,39	c	0,95	0,07	P	0,14	2,807			
a	0,86	0,22	n	0,14	2,81		0,81	0,3	o	0,55	0,86	d	0,24	2,07	c	0,95	0,07	a	0,33	1,585			
	0,95	0,07	i	0,29	1,81	c	0,48	1,07	m	0,7	0,51	a	0,86	0,22	i	0,95	0,07	l	0	12,39			
c	0,14	2,81	n	0,38	1,39	e	0,19	2,39	b	0,95	0,07	;	0	12,4	o	0,95	0,07	v	0	12,39			
o	0,67	0,58	g	0,9	0,14	n	0,29	1,81	r	0,95	0,07		0,95	0,07	n	0,95	0,07	e	0	12,39			
n	0,52	0,93	u	0,95	0,07	t	0,9	0,14	e	0,95	0,07	p	0,1	3,39		0,76	0,39	r	0,05	4,392			
t	0,57	0,81	n	0,95	0,07	r	0,81	0,3		0,9	0,15	e	0,48	1,07	e	0,19	2,39	,	0,05	4,392			
e	0,24	2,07	a	0,67	0,58	o	0,86	0,22	l	0,1	3,32	r	0,81	0,3	n	0,9	0,14		0,95	0,07			
m	0,29	1,81		0,95	0,07		0,95	0,07	l	0,05	4,32	o	0,86	0,22		0,71	0,49	P	0,05	4,392			
p	0,9	0,14	o	0,24	2,07	d	0,86	0,22	a	0,5	1		0,95	0,07	e	0,29	1,81	r	0,33	1,585			

La tabla muestra para cada caracter del texto la probabilidad de acuerdo a las predicciones del grupo y la cantidad de bits emitidos para dicho caracter. El total de bits ocupados por los 348 caracteres del archivo fue de 470 bits. Lo que da el promedio de 1,35 bits por byte.

El siguiente gráfico muestra la cantidad de bits emitidos por cada caracter del texto.

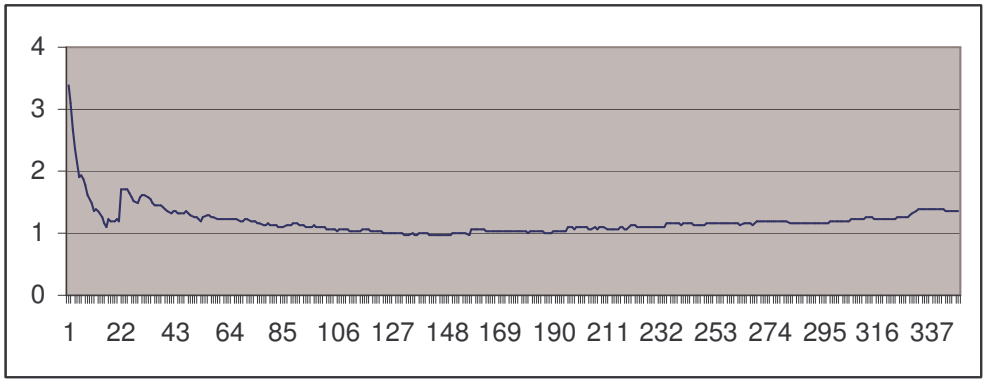


Los picos corresponden a aquellos caracteres que no fueron adivinados por ninguna persona del grupo y para los cuales se tuvo que emitir un escape. Los caracteres en cuestión son los que figuran en la parte superior del gráfico. Muchos de los escapes son justificables, la “P” mayúscula es muy poco predecible, la “m” de sentimiento nadie la acertó pues la mayoría esperaba una “d” para “sentido”, el ‘;’ es también justificable al igual que tres de las letras del apellido “Palver”. Mas raros son los fallos en la “n” de núcleo o la “r” de redondo e inexplicablemente a nadie se le ocurrió la “l” de “la”, caracter absolutamente normal, puede ser que luego de “llama”, la mayoría esperaba un nombre o algo así.

Caracteres que no fueron adivinados por ninguna de las personas del test.

> Diez meses antes, el **P**rimer Orador habia contemplado aquellas mismas estrellas, que en ninguna otra parte eran numerosas como en el centro de ese enorme **n**ucleo de materia que el hombre llama **l**a Galaxia, con un senti**m**iento de duda**;** pero ahora se reflejaba una sombr**i**a satisfaccion en el rostro **r**edondo **y** rubicundo de Preem Pa**lve**r, Primer Orador.

Gráfico del nivel de compresión en bits x byte a medida que se procesaba el párrafo.



Como puede observarse en el gráfico rápidamente el nivel de compresión bajo de mas de tres bpb a alrededor de 1,5 y a medida que se conocían mas caracteres del texto el nivel fue bajando aun mas. El mejor nivel de compresión fue de 0,963027 bpb (Luego del caracter “e” en el fragmento “tan numerosas como en el centro de”. El peor nivel de compresión, como es lógico de cualquier proceso de compresión adaptativo se registro para el primer caracter en 3,39 bpb.

El estudio anterior es importante para tener un conocimiento relativamente bueno de cuanto mas puede avanzarse en el campo de la compresión de datos, al menos para textos. Tanto en castellano como en inglés el nivel teórico límite de compresión es de alrededor de 1 bit por byte. Teniendo en cuenta que los mejores algoritmos de compresión hasta la fecha alcanzan alrededor de 2 bits por byte podemos suponer que aun queda mucho para investigar (estamos comprimiendo al doble del tamaño mínimo ideal). Por otra parte es dudoso que alguna máquina o algoritmo pueda algún día alcanzar el grado de conocimiento necesario como para equiparar sus predicciones a las predicciones de un ser humano. Y si alguna vez se construye una máquina capaz de hacer tal cosa entonces tendremos cosas mucho mas interesantes para investigar que la compresión de datos.

•Sincronización.

La sincronización es un tema importante de la compresión de datos que tiene relación con temas de transmisión de datos y códigos autocorrectores. La sincronización se refiere básicamente a que problemas pueden surgir entre el compresor y el descompresor y como se solucionan los mismos. El primer problema básico de sincronización es como hacer que el descompresor sepa cuando debe detenerse, es decir, la longitud del archivo original.

Longitud del archivo a descomprimir.

Este problema se origina en la necesidad de redondear la cantidad de bits emitidos por el compresor a un múltiplo de 8 ya que el archivo comprimido debe almacenarse en una cantidad entera de bytes. Hay dos soluciones básicas que sirven para cualquier algoritmo: la codificación de un símbolo especial de EOF o pre-concatenar al archivo comprimido la longitud del archivo original en bytes.

El símbolo de EOF.

En este caso a los 256 símbolos ASCII se les agrega un símbolo especial de End of File, este símbolo solo será emitido una vez y cuando el descompresor lo encuentre sabrá que se detiene el proceso de descompresión. Con este método y debido a que el caracter de EOF es muy poco probable (de hecho solo ocurrirá una única vez en todo el archivo) se agregan al archivo en promedio unos 15 bits, pueden ser mas o menos dependiendo de la longitud del archivo.

Pre-concatenar la longitud original.

Con este método la longitud del archivo original se guarda al comienzo del archivo comprimido, el descompresor lee la longitud y luego descomprime dicha cantidad de bytes ignorando el resto. Esta técnica implica en general agregar unos 32 bits al archivo comprimido para almacenar la longitud.

Una solución aun mejor.

Existe un tercer método que resulta mas eficiente que los dos anteriores pero que, lamentablemente, no es aplicable a todos los algoritmos. El método consiste en lo siguiente: luego del último bit del archivo comprimido se concatena un bit en 1 y luego tantos bits en cero como hagan falta para llegar a una cantidad de bytes entera. El descompresor antes de procesar el archivo comprimido lo empieza a leer de atrás para adelante y elimina todos los ceros y el primer uno. De esta forma sabe cuales son todos los bits que corresponden al archivo original comprimido. Este método como mínimo agrega 2 bits (10) y como máximo un byte entero. (Si el archivo ocupaba justo una cantidad entera de bytes o un bit menos). En promedio se agregan unos 3 bits por archivo, mucho menos que en los métodos anteriores.

Este método no sirve para archivos comprimidos con compresión aritmética, para un compresor aritmético conocer la longitud en bits del archivo comprimido no es de utilidad ya que de todas formas no sabría como detenerse ya que siempre estaría en el mismo intervalo de números y continuaría emitiendo el último símbolo en forma infinita. Para archivos comprimidos con compresión aritmética se recurre casi siempre a la utilización del símbolo especial de EOF.

El segundo problema básico de sincronización es que ocurre con el descompresor si el archivo comprimido sufre algún tipo de daño, en los casos en los cuales se quiera tener esto en cuenta habrá que intentar recuperar la mayor cantidad de bytes posibles del archivo original. Por ejemplo en un texto de 5 o 6 Mb, un bit alterado que implique una o dos líneas ilegibles son aceptables por lo que no es bueno que si el archivo comprimido se dañe no pueda recuperarse ningún dato. Esto es fundamental al utilizar archivos comprimidos en bases de datos. En los sistemas de recuperación total de textos (Full retrieval text systems) se almacenan gran cantidad de documentos ocupando un gran volumen por lo que se suele guardar la información comprimida pero esto no debe ir en contra de la integridad física de la base de datos ya que de lo contrario el sistema seria poco confiable, en estas situaciones la sincronización entre el compresor y el descompresor es fundamental.

Un método de sincronización consiste en establecer un algoritmo, método o protocolo de compresión que permita, en caso de dañarse el archivo original recuperar la mayor cantidad de datos posibles.

Hay algoritmos de compresión que son auto-sincronizantes, es decir, que si se daña el archivo comprimido el descompresor, luego de trastabillar en algunos bytes, vuelve a emitir los caracteres que correspondían al archivo original. La gran mayoría de los compresores estadísticos que emiten códigos de longitudes (en bits) enteras, como por ejemplo Huffman, Shannon Fano o Huffword son auto-sincronizantes. De hecho es muy difícil construir un compresor de este tipo que no sea autosincronizante!

Tomemos como ejemplo el código Huffman generado para el fragmento Asimov.

> Diez meses antes, el Primer Orador habia contemplado aquellas mismas estrellas, que en ninguna otra parte eran tan numerosas como en el centro de ese enorme nucleo de materia que el hombre llama la Galaxia, con un sentimiento de duda; pero ahora se reflejaba una sombria satisfaccion en el rostro redondo y rubicundo de Preem Palver, Primer Orador.

Char	Freq	Code
	54	000
e	45	010
a	36	111
r	27	0010
o	24	0011
n	22	0110
s	20	1000
m	17	1010
l	16	1101
i	15	01110
d	12	01111
t	12	10010
u	11	10011
c	8	110010
b	5	101100
P	5	101101
,	4	1100111
h	3	1011100
p	3	1011101
q	3	1011110
f	2	1011111
O	2	1100000
.	1	11000010
;	1	11000100
D	1	11000101
G	1	11000110
j	1	11000111
v	1	11000011
x	1	110011001
z	1	110011010
y	1	110011011
g	1	110011000

Veamos que ocurre si cambia un bit del archivo comprimido, por ejemplo uno de los bits correspondientes a la m de meses.

```
 m  e  s  e  s  -  a  n  t  e  s  ,  -     e  l
1110 010 1000 010 1000 000 111 0110 10010 010 1000 1100111 000 010 1101
 a  r  s  e  s  -  a  n  t  e  s  ,  -     e  l
```

Como puede observarse un bit erróneo en el archivo comprimido causo que dos de los caracteres descomprimidos sean erróneos, pero luego de dichos caracteres el archivo original fue recuperado sin problemas. Este es un claro ejemplo de un algoritmo auto-sincronizante.

Hay muchos algoritmos que no pueden ser auto-sincronizantes, en principio ningún compresor aritmético es autosincronizante, tampoco son auto-sincronizantes los algoritmos adaptativos ya que, una vez que se produce un error el descompresor empieza a aprender cosas equivocadas y nunca mas vuelve al camino original.

En los algoritmos que no son auto-sincronizantes los problemas de sincronización deben resolverse agregando 'puntos de sincronización' es decir comprimiendo el archivo original en bloques y no como una única tira de bytes, de esta forma si se produce algún error solo se pierde un bloque de datos y no todo el archivo. En general la compresión por bloques es muy utilizada no solo por esto sino porque hasta puede presentar alguna ligera mejora en cuanto al nivel de compresión de algunos algoritmos (ver compresión por bloques más adelante). En particular, en los algoritmos aritméticos adaptativos la sincronización plantea algunos problemas ya que si por cada punto de sincronización (bloque) el modelo se destruye y se reinicia de cero el nivel de compresión se ve sensiblemente afectado. Para evitar este problema hay dos posibles enfoques: uno es guardar el modelo luego de cada bloque procesado para poder retomarlo en el bloque siguiente. Esta técnica implica utilizar mucho espacio (sobre todo en modelos de orden grande) para guardar información que, a menos que ocurra un error, será redundante. La segunda posibilidad consiste en entrenar al modelo antes de la compresión de cada bloque, para ello se usan porciones del bloque a comprimir. Esta segunda técnica ahorra espacio pero desperdicia tiempo, necesario para el precalentamiento del modelo y además implica que el algoritmo deja de ser dinámico y pasa a ser semi-estático. Como ambos enfoques son problemáticos, en general no se utilizan compresores aritméticos en esquemas en los cuales la sincronización sea un requisito importante.

•El estado del arte en compresión de datos.

Desde que Shannon enunciara su teoría de la información en 1948 la compresión de datos adquirió un importante auge y muchas personalidades de renombre se dedicaron a estudiar la forma de comprimir un mensaje en forma optima. En 1952 D.A.Huffman presento su método para determinar la longitud en bits que debía tener cada caracter de un archivo dada su probabilidad de ocurrencia de forma tal que la longitud del archivo comprimido sea mínima, se demostró que el método de Huffman era óptimo para códigos cuya longitud en bits debía ser entera, fue demostrado por el propio Huffman que otros métodos como los de Shannon-Fano eran sub-óptimos en algunos casos.

Sin embargo siempre se supo que el método de Huffman no era el ideal, según las teoría de Shannon la longitud que debía tener el código correspondiente a un caracter de probabilidad Pi debía ser -log2(Pi) de forma tal que para una probabilidad de 1/3 la longitud del código debía ser de 1.6 bits, como los códigos de Huffman ocupan una cantidad entera de bits el 1.6 debía redondearse a 1 o bien a 2 bits deformando el esquema de compresión y alejándolo del óptimo (si el óptimo es 1.6 entonces usar longitud 1 o 2 afecta no solo al código en cuestión sino a todos los demás en forma negativa)

Existían casos aun peores, si un caracter tenia una probabilidad de 0.98 la formula -log2(0.98) nos indica que debemos representar a este caracter con un código de 0.15 bits! como el mínimo es 1 bit usando Huffman debemos si o si usar 6 veces mas bits que en el caso óptimo!

La forma de comprimir en forma optima de acuerdo a la formula de Shannon no fue descubierta hasta el surgimiento de los compresores aritméticos, la compresión aritmética revolucionó el estado de los algoritmos de compresión al permitir que un caracter fuera representado con una cantidad no entera de bits, permitiendo que cada caracter fuera representado con la cantidad óptima de bits necesaria!.

Un caso extremo que demuestra el poder de la compresión aritmética consiste en comprimir archivos en los cuales un caracter tiene una probabilidad muy alta, por ejemplo supongamos que el caracter "0" tiene una probabilidad de 16382/16383 y que el caracter de EOF tiene una probabilidad de 1/16383, si comprimimos un archivo con 100.000 ceros por Huffman obtenemos un archivo de 12501 bytes mientras que un compresor aritmético es capaz de comprimir el mismo archivo en 3 bytes!!!.

Una vez demostrado que la compresión aritmética lograba representar a cada caracter en la cantidad de bits óptima dada su probabilidad los algoritmos de compresión se dedicaron a mejorar la forma en la cual se calculaba la probabilidad de cada caracter, comprimir mejor paso a significar predecir mejor, de esta forma surgieron los modelos de orden superior en los cuales la probabilidad de cada caracter ya no se considera en forma aislada sino inmersa en un contexto determinado.

En la actualidad los mejores algoritmos de compresión están totalmente basados en compresión aritmética, algunos combinan compresión por run-length pero el común denominador es el uso de compresión aritmética para representar los códigos, los algoritmos de compresión suelen testearse usando un lote de archivos especiales de distintas características de forma tal que todos puedan cotejarse en función del mismo lote de datos, en la actualidad el lote mas usado para probar algoritmos de compresión es el "Calgary corpus" que se puede conseguir en internet, obviamente existen otros lotes, el nivel de compresión de un algoritmo se mide en bits por byte en función de un lote determinado, el algoritmo LZFG por ejemplo promedia "2.95 bpb CC" (2.95 bits por byte Calgary-Corpus).

Uno de los últimos Tests realizados sobre los archivos de Calgary-Corpus de los mejores algoritmos de la actualidad arrojo los siguientes resultados.

t	*PPM**	*PkZip*	*LZFG*	*LZS*	*Bzip*	*PPMZ*
Velocidad Kb/s :	3	10	9	85	9	2
Memoria usada :	2500K	206K	180K	16K	512K	4500K
Bits x Byte :	2.283	2.710	2.950	4.601	2.253	2.057

Los mejores algoritmos de compresión hasta el día de la fecha son el Bzip de Julian Seward. Y el PPMz, evolución avanzada y ya casi complicadísima de los compresores aritméticos, con la saludable sorpresa de que Bzip es sumamente aplicable mientras que los aritméticos mas complejos siguen presentando problemas insustanciables, hasta que el hardware avance, con respecto a los tiempos insumidos. El nivel de compresión del PPMz es realmente formidable, sin embargo es dudosa su utilización con fines prácticos. Versiones ya compiladas para distintos sistemas operativos o los fuentes mismos de Bzip pueden conseguirse sin problemas y mucha gente ha estado usando Bzip hasta el día de hoy sin quejarse en absoluto.

Que puede pasar en el futuro es un verdadero misterio, los compresores aritméticos y las técnicas de modelización para un calculo de probabilidades cada vez mejor siguen evolucionando, a su vez, en la medida en que el hardware avance los compresores aritméticos serán cada vez mas prácticos y un día, sin que nadie se dé cuenta, puede que todos estemos usando algún compresor aritmético de orden super-avanzado en nuestras casas. Por otra parte con PPMZ se han quedado sin letras para la próxima versión lo cual nos llega de interrogantes: ¿Que sigue? PPMAA , PPMA1, PPMα ?. Quién sabe...

Los compresores basados en algoritmos combinados por otra parte han avanzado mucho con el descubrimiento del MTF y el algoritmo de Block-Sorting, estos dos métodos son muy recientes y no fueron utilizados hasta hace muy poco. Además dada la extrema sencillez de ambos algoritmos no resulta comprensible como no fueron descubiertos con anterioridad. Esto hace suponer que nuevos métodos, sorprendentes y reveladores puedan surgir en los próximos años. En la medida en que la imaginación no se agote el mundo de la compresión de datos seguirá avanzando día a día.

• Compresión por bloques. (Block-Basis).

La compresión por bloques no implica un nuevo algoritmo de compresión sino que es una técnica de compresión que puede aplicarse a cualquier algoritmo de compresión, incluyendo todos los vistos hasta el momento.

El concepto básico es muy sencillo, en lugar de comprimir todo el archivo como un gran conjunto de bytes se particiona al archivo en bloques de un cierto tamaño fijo y se comprime cada bloque en forma independiente.

La pregunta que surge es: ¿cuál es la ventaja de comprimir varios bloques en lugar de uno solo?. La principal diferencia reside en que comprimiendo a un archivo como un único bloque se produce un efecto de globalización en las probabilidades de ocurrencia de los caracteres que tiende a ser perjudicial, bloques de compresión mas pequeños permiten comprimir con mayor precisión cada uno de ellos. Como ejemplo notorio pensemos que podemos encontrar en un archivo bloques compuestos por un único caracter o por un grupo reducido de los mismos con lo cual una compresión estadística cualquiera podría ser sumamente beneficiosa. La desventaja reside en que la globalización a veces es beneficiosa, para archivos que son casi-random puede que no observemos que algunos caracteres son levemente mas probables que otros hasta haber procesado el archivo entero, si procesamos cada bloque en forma independiente no podremos notar esta pequeña diferencia que nos permitiría comprimir mejor. En general es sumamente raro encontrar casos en los cuales la globalización sea beneficiosa y no convenga comprimir por bloques, en primer lugar porque esto solo ocurre en archivos casi totalmente random que como sabemos pueden comprimirse muy poco por lo que no nos interesa esforzarnos demasiado en comprimirlos (dada una técnica t1 y una técnica t2 supongamos que t1 comprime mejor a los archivos random que t2 mientras que t2 comprime mejor a los archivos estructurados que t1 entonces nos quedaremos con t2 pues en conjunto obtendremos mucho mejores resultados que con t1). Además la ventaja en nivel de compresión que se obtiene cuando la globalización es efectiva es muchísimo menor a la ventaja que se obtiene cuando la compresión por bloques es conveniente.

Observemos un ejemplo sumamente ilustrativo.

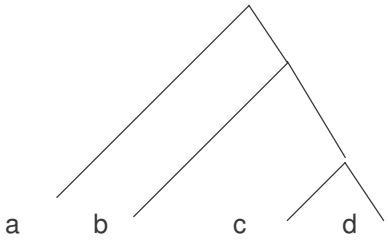
Sea el archivo:

"accaaddacabaacaabbbddbcbabdbbdbc"

Supongamos que comprimimos este archivo usando el método de Huffman estático común antes desarrollado.

char	freq
a	**10**
b	**10**
c	**6**
d	**6**

El árbol de Huffman quedaría de la forma:



Conviniendo 0=izquierda y 1=derecha los códigos y las longitudes generadas son:

Char	Cod	Long(bits)
a	**0**	**1**
b	**10**	**2**
c	**110**	**3**
d	**111**	**3**

De acuerdo a la cantidad de veces que cada caracter aparece en el archivo la longitud final del archivo comprimido es igual a:

10x1+10x2+3x6+3x6=10+20+18+18=66 bits. (*)

Veamos que pasa ahora si dividimos al archivo en dos bloques de 16 bytes cada uno y comprimimos ambos bloques en forma independiente.

Bloque 1: accaaddacabaacaa

Las frecuencias, códigos y longitudes de cada caracter aplicando el método de Huffman estático son:

BLOQUE1:

Char	Freq	Cod	Long(bits)
a	9	0	1
b	1	110	3
c	4	10	2
d	2	111	3

Bloque 2: bbbddbcbabdbbdbc

Las frecuencias, códigos y longitudes de cada caracter aplicando el método de Huffman estático son:

BLOQUE2:

Char	Freq	Cod	Long(bits)
a	1	110	3
b	9	0	1
c	2	111	3
d	4	10	2

La longitud del archivo comprimido será la suma de las longitudes de ambos bloques.

Long(Bloque1) = 9x1 + 4x2 + 2x3 + 1x3 = 26 bits.
Long(Bloque2) = 9x1 + 4x2 + 2x3 + 1x3 = 26 bits.

Long(archivo) = 26 + 26 = 52 bits. (*)

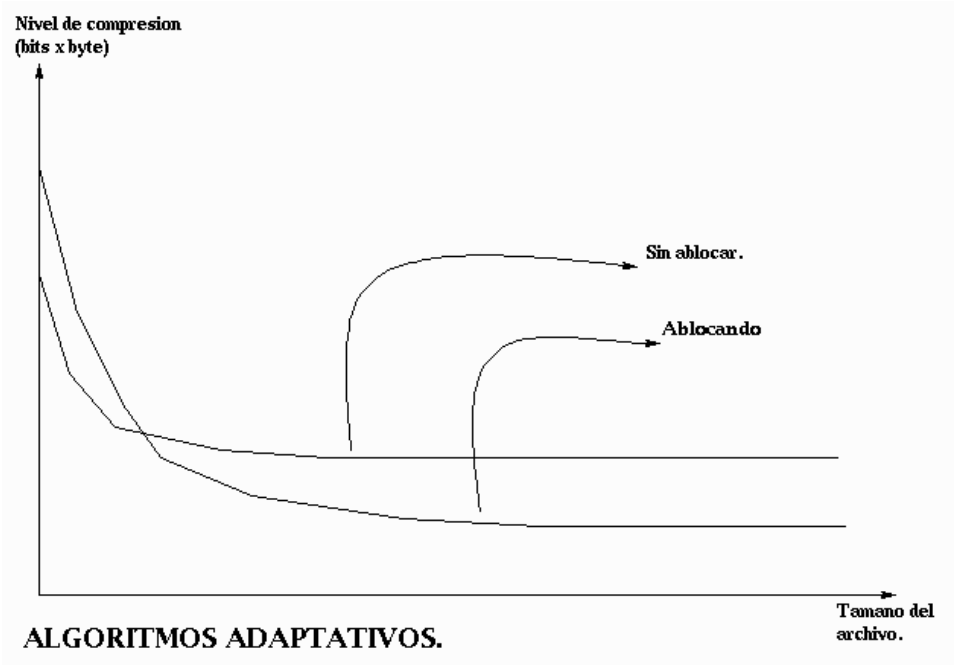
Como podemos apreciar comprimimos al archivo en 52 bits, longitud bastante menor a la de 66 bits que hubiésemos obtenido si no ablocabamos.

(*) En realidad al tamaño del archivo comprimido hay que sumarle el tamaño de los dos árboles de Huffman, en el segundo ejemplo debemos almacenar dos árboles (uno por cada bloque) mientras que en nuestro primer ejemplo (el clásico) solo debemos almacenar un árbol.

En general si utilizamos un método estadístico estático los métodos que comprimen por bloques a la larga tienden a comprimir cada vez peor pues la longitud extra impuesta a cada bloque por tener que guardar el árbol tiende a ser perjudicial. Es por esto que todos (99%) los métodos que comprimen por bloques utilizan algoritmos adaptativos. De aquí que los métodos adaptativos sean tan utilizados en los productos comerciales, no por sus ventajas intrínsecas sino porque son el complemento obligado de cualquier técnica de compresión por bloques.

En nuestro ejemplo la versión adaptativa de Huffman comprimiría al bloque 1 en 28 bits y al bloque 2 en 27 bits. La diferencia radica en que el bloque 2 permite al algoritmo aprender con mayor rapidez cuales son los caracteres mas frecuentes en el bloque mientras que en bloque 1 el comienzo "acca" hace que la segunda "a" la comprima considerando que la probabilidad de la "c" es mayor, lo cual se revierte al seguir leyendo caracteres del bloque, por eso el bit extra. La longitud total sería pues de 55 bits. Esta longitud sigue siendo menor que los 66 bits del método tradicional y todavía tenemos otra ventaja: no hay que guardar ningún árbol!.

A continuación se muestra un gráfico indicativo sobre el nivel de compresión comparativo entre los algoritmos adaptativos que comprimen por bloques y aquellos que no lo hacen.



ALGORITMOS ADAPTATIVOS.

En el gráfico podemos observar como los algoritmos que ablocan comienzan siendo inferiores a los que no ablocan debido a que la velocidad de entrenamiento de los algoritmos adaptativos que no ablocan es necesariamente superior, en los algoritmos que ablocan es necesario que todos los bloques 'aprendan' para que el algoritmo tienda a alcanzar su máximo nivel de compresión. Una vez que se alcanza dicho nivel de entrenamiento las ventajas de la compresión por bloques causan que el nivel de compresión alcanzado sea superior.

El gráfico como puede suponerse es válido para un cierto tamaño de bloque fijo, si el tamaño del bloque varía la curva cambia, el objetivo es obviamente encontrar el tamaño de bloque óptimo para el cual se alcance el mejor nivel de compresión promedio. El tamaño del bloque es el parámetro mas importante de los esquemas de compresión por bloque, el tamaño ideal depende del método a utilizar, los tamaños mas utilizados comercialmente oscilan entre los 2Kb y los 64Kb. Cuánto mas chico sea el bloque mas bloques tendremos y perderemos nivel de compresión debido a la necesidad de entrenamiento en la compresión adaptativa de cada bloque en contraparte cuanto mas grande sea el bloque perderemos nivel de compresión debido a la globalización. El secreto reside en encontrar un balance que nos indique el tamaño óptimo del bloque.

•Archivadores.

Se llama archivador (archiver) a un programa que suele ser utilizado junto con casi cualquier compresor que permite la manipulación de varios archivos dentro de un único archivo comprimido. La función básica de un archivador es comprimir varios archivos generando un único archivo comprimido, para esto existen dos técnicas básicas:

1. Juntar todos los archivos a comprimir y luego comprimir el archivo resultante.
2. Comprimir todos los archivos individualmente y luego juntarlos en un único archivo.

El esquema mas utilizado suele ser el segundo pues permite aprovechar en mayor medida las características propias de archivos de distinto tipo, si estos se juntaran en un único archivo grande muchos detalles se perderían al englobarse toda la información. Además la segunda técnica impide el manejo corrector de los archivos individuales dentro del archivo comprimido lo cual es muy importante.

Un archivador puede proveer además las siguientes funciones:

- Agregar archivos al archivo comprimido. **(agregación)**
- Eliminar archivos de adentro de un archivo comprimido. **(eliminación)**
- Descomprimir un determinado archivo o bien un conjunto de archivos de adentro de un archivo comprimido. **(extracción)**
- Descomprimir un determinado archivo eliminándolo del conjunto de archivos de adentro de un archivo comprimido **(extracción y eliminación)**
- Comprimir los subdirectorios y sus archivos en forma comprimida de forma tal que al descomprimir se vuelva a generar el árbol comprimido **(compresión recursiva)**
- Comprimir varios archivos generándose un archivo ejecutable que se descomprime automáticamente sin necesidad de un descompresor (el ejecutable incluirá al descompresor y a los datos comprimidos) (**autoextractables).**
- Comprimir generándose varios archivos de un cierto tamaño, útil si la información se debe guardar en disquetes, por ejemplo. (**Multivolumenes)**
- Comprimir generándose varios volúmenes de un cierto tamaño siendo el primero de ellos ejecutable de forma tal que no se necesite al descompresor. **(Multivolumenes autoextractables)**
- Agregar un chequeo de integridad de forma tal que si se modifica alguno de los archivos comprimidos no pueda descomprimirse el archivo. (**Autenticación)**
- Agregar códigos autocorrectores al archivo comprimido. **(Autocorrección).**

La gran mayoría de los productos comerciales proveen de varias de estas opciones, la gama de opciones que ofrece un compresor debe ser tenida en cuanta como un factor importante a la hora de utilizarlo, sin embargo estos factores no son tan importantes como el algoritmo utilizado ya que sobre cualquier algoritmo de compresión puede construirse un archivador tan bueno como se desee.

•Productos Comerciales:

Existen en el mercado una enorme cantidad de productos destinados a la compresión y descompresión de datos, a su vez existen numerosos tests realizados en los cuales se puede ver a grandes rasgos cuales son los mejores compresores del momento, no reproducimos en este texto ninguno de estos tests pues no es nuestra intención el comparar los distintos compresores del mercado sino simplemente mencionar algunos que se destacan y sus cualidades principales.

•ARJ 2.42 (R.Young) [LZ77+Huffman dinámico por bloques+hashing]
Este compresor es muy utilizado y combina muy bien velocidad y nivel de compresión, genera archivos autoextractables y multivolúmenes.

•PKZIP 2.04g (PkWare inc) [Varios métodos]
Sin dudas el mas utilizado de todos los compresores, a punto tal que se ha convertido casi en un standard, su punto mas destacado es la velocidad, es uno de los mas rápidos y comprime bastante bien, sin embargo se queda atrás en cuanto a opciones, no hace multivolúmenes ni genera autoextractables. (salvo usando ejecutables adicionales como ZIP2EXE).

•HAP 3.0 (Hammarsoft) [Aritmético: PPMC]
Este es el único compresor comercial que utiliza compresión aritmética y como es lógico es uno de los que mas comprime, solamente superado por el Bzip, es notable que sea bastante rápido.

•RAR 2.52 [Algoritmos propios]
Compresor de origen ruso, se destaca por su excelente interfase, es el único que además de manejarse por línea de comandos tiene una interfase estilo Norton-Commander muy cómoda y práctica. Tiene todas las opciones que se puedan pedir, autoextractables, volúmenes sólidos, multivolúmenes, verificación de integridad y es el único que hace autoextractables multivolúmenes.

•LHA 2.55 (Yoshi) [Lz77+Huffman dinámico por bloques c/hashing]
Este compresor fue muy utilizado hace un tiempo pero últimamente se ha dejado de utilizar siendo desplazado por el ARJ y el RAR, además del tradicional PkZip. Tiene un rendimiento bueno, comparable al del ARJ pero viene con menos opciones.

•Ultracompressor 2. [Varios métodos]
Este es el único compresor que aplica técnicas diferenciadas según la extensión del archivo a comprimir, posee varios algoritmos especializados en distintos formatos de archivos, comprime realmente bien tardando un poco mas que los compresores mas utilizados. Muy configurable.

•X1 0.94c (Stig Valentini) [Método propio]
Este es uno de los últimos compresores surgidos en el mercado de dominio público, tiene un nivel de compresión excelente y es bastante rápido, viene con todas las opciones de ultima moda y además agrega la posibilidad de agregar a los archivos códigos autocorrectores, este es el único compresor que hace esto.

•RDC 95+ (RDC tech) [Algoritmo RDC]
Este es el compresor mas rápido del oeste (y del mercado), el nivel de compresión no es excelente pero la velocidad es realmente notable, indispensable cuando se quiere comprimir un gran volumen de información rápido y no es necesario comprimir mucho.

Bzip (Julian Seward) [Block sorting + MTF + Compresión aritmética]
El mejor algoritmo en relación tiempo - nivel de compresión, comprime muchísimo en un tiempo comparable al del arj y levemente superior al del Pkzip, al mejor estilo de los compresores de distribución libre y gratuita lo único que hace es comprimir y descomprimir, y lo hace solamente para archivos en forma individual.

•Referencias:

[1989] Bell, T.C, Cleary, J.G. and Witten, I.H, "Text Compression",
Prentice-Hall 1989. ISBN: 0-13-911991-4.
Este libro es la referencia obligatoria sobre algoritmos de compresión de datos. Los autores revisan en forma muy clara y con ejemplos los algoritmos de Huffman, de compresión aritmética, PPM, LZ77, LZ78 y otros.

[1991] Mark Nelson, "The Data Compression Book"
M&T Books, Redwood City, CA, 1991. ISBN 1-55851-216-0.
Otro libro que detalla una larga tira de algoritmos de compresión, no tiene tanto detalle como “Text Compression”

[1990] Williams, R. "Adaptive Data Compression", Kluwer Books, 1990.
ISBN: 0-7923-9085-7. Price: US$75.

[1988] Storer, J.A. "Data Compression: Methods and Theory", Computer Science Press, Rockville, MD. ISBN: 0-88175-161-8.
Este libro se basa fundamentalmente en el estudio de los algoritmos LZSS y LZ78. Abunda en detalles de implementación y trae los fuentes de varios algoritmos basándose siempre en LZSS.

[1991] Elements of Information Theory, by T.M.Cover and J.A.Thomas John Wiley & Sons, 1991. ISBN 0-471-06259-6.

[1989] Bell, T.C, Witten, I.H, and Cleary, J.G. "Modeling for Text Compression", ACM Computing Surveys, Vol.21, No.4 (December 1989), p.557

[1987] Lelewer, D.A, and Hirschberg, D.S. "Data Compression", ACM
Computing Surveys, Vol.19, No.3 (September 1987), p.261.

[1994] Burrows, M. and Wheeler, D.J. "A Block-sorting lossless Data compression Algorithm." SRC Research Report. Digital.
Este es el Paper original donde se detalla el algoritmo de Block-Sorting. Los autores describen el algoritmo, explican porque comprime bien y explican como puede lograrse una implementación eficiente del método. Clarísimo.

[1994] Witten, Moffat, Bell. “Managing Gigabytes”
Van Nostrand Reinhold. ISBN 0-442-01863-0.
Este libro se dedica al manejo de Bases de Datos para sistemas de recuperación total de textos manejando volúmenes de información muy grandes. El capitulo dos detalla varios algoritmos de compresión, incluyendo Huffword. El capitulo 9 describe como implementar el algoritmo Huffword para este tipo de sistemas. El resto del libro describe la creación, compresión y utilización de índices sobre el texto comprimido.

[1995] J.Clearly, W. Teahan, Ian Witten. “Unbounded length contexts for PPM”. Department of Computer Science, University of Waikato, New Zealand.
Paper original del algoritmo PPM*, se describe como antecesor al algoritmo PPMC.

[1996] Peter Fenwick. “Block Sorting Text Compression.” Department of Computer Science, the University of Auckland. Private Bag 92019. New Zealand.
Optimizaciones sobre implementación del algoritmo de Block-sorting. Aquí se describen el modelo de Shannon y el modelo Estructurado.