Universidad Católica Nuestra Señora de la Asunción

Teoría y Aplicación de la Informática 2

- Trabajo Práctico -

Algoritmos de Compresión Sin Pérdida de Datos

Alumno: Roberto Giménez

Año: 2004

Algoritmos de Compresión Sin Pérdida de Datos

Introducción

Los algoritmos de compresión han jugado un papel importante en el mundo de la informática. Gracias a ellos, podemos ahorrar espacio y tiempo, lo cual es beneficioso para cualquier usuario. Prácticamente cualquier flujo de datos a través de Internet está comprimido de alguna u otra manera. Instaladores, música, videos, todos son transmitidos de manera comprimida o ya poseen un formato comprimido. A continuación presentaré un conjunto de algoritmos y técnicas de compresión donde a partir del resultado final se puede restaurar el origen de manera exacta, esto es, sin pérdida de datos.

La codificación Run-Length

La codificación Run-Length es la más simple de todas las técnicas de compresión. Permite la compresión de cadenas de caracteres existen grandes tramos de caracteres iguales. Por ejemplo, la cadena "abcddddddcbbbbabcdef" puede ser codificada como "abc6dc4babcdef", donde los números indican la cantidad de veces que las letras posteriores a las mismas deben ser repetidas.

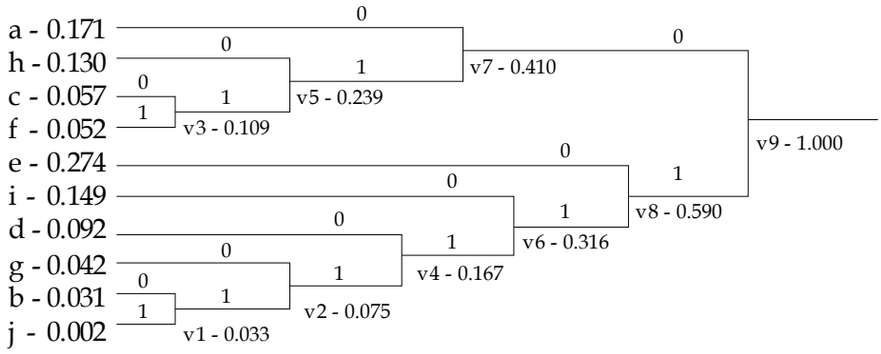
En la realidad, esta forma de codificación no es realizable de manera directa ya que no hay manera de diferenciar un número de un código de repetición. Un enfoque común es utilizar bytes con signo para indicar la presencia o ausencia de repeticiones. Si el byte encontrado es negativo, significa que los siguientes n bytes (indicado por el byte) deben ser escritos de manera normal. Si el byte es positivo, significa que el siguiente byte debe ser repetido n veces (indicado por el byte). El ejemplo anterior sería entonces codificado como "-3 a b c 6 d -1 c 4 b 6 a b c d e f ".

La codificación Huffman

La codificación Huffman es una codificación que pasa un contenido de tamaño fijo a uno de tamaño variable (menor por lo general). Está basada en asignar códigos cortos a caracteres muy repetidos y códigos largos a caracteres menos repetidos. La asignación se realiza de la siguiente manera:

- Se hallan las probabilidades de aparición cada carácter.
- Se toman los dos caracteres menos probables y a uno de ellos se le asigna un 0 y al otro un 1 como inicio de su código.
- Se suman las probabilidades de estos dos caracteres y esta suma es asignada a un *carácter virtual*, que no posee representación pero participa en las elecciones.
- Los caracteres anteriormente tomados ya no participarán de las elecciones. Sin embargo, si el carácter virtual es eventualmente elegido, el código asignado al mismo irá directamente seguido a los códigos del par de caracteres que lo crearon.
- Se repite el proceso hasta que se cree el carácter virtual de probabilidad 1.

El siguiente ejemplo ilustra el procedimiento:



a - 0 0
h - 0 1 0
c - 0 1 1 0
f - 0 1 1 1
e - 1 0
i - 0 1 1 1
d - 1 1 1 0
g - 1 1 1 1 0
b - 1 1 1 1 1 0
j - 1 1 1 1 1 1

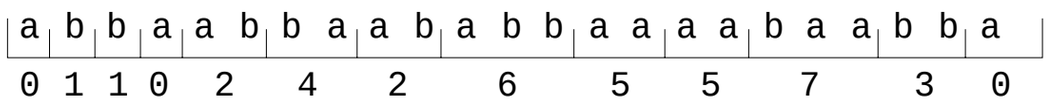
Hay situaciones en donde se generan ataduras entre códigos. Para evitar esto, debemos aplicar un procedimiento de rotura de ataduras.

Codificación Lempel-Ziv

La codificación Lempel-Zip es una codificación de tamaño variable a fijo basado en un diccionario. La idea básica es dividir la entrada en bloques no superpuestos de diferentes longitudes mientras se construye un diccionario de *bloques ya vistos*. Los pasos a seguir son:

- Inicializar el diccionario para que contenga todos los bloques de longitud 1, esto es, todos los posibles bloques de una letra.
- Buscar en la entrada el bloque más largo que aparece en el diccionario.
- Codificar el bloque por su índice en el diccionario.
- Agregar el bloque encontrado seguido del siguiente símbolo del siguiente bloque en el diccionario. Si se ha establecido un límite en la cantidad de entradas, no agregar nuevos bloques si se ha llegado al mismo.
- Seguir con este procedimiento hasta terminar la entrada.

Por ejemplo, la entrada "abbaabbaababbaaaabaabba" se divide en los siguientes bloques:



Índice	Entrada	Índice	Entrada	Índice	Entrada
0	a	5	a a	10	a a a
1	b	6	a b b	11	a a b
2	a b	7	b a a	12	b a a b
3	b b	8	a b a	13	b b a
4	b a	9	a b b a		

Codificación Aritmética

La codificación Huffman asigna un código de salida cada símbolo, con los códigos de salida siendo tan cortos como 1 bit, a considerablemente más largos que los símbolos de entrada, estrictamente dependiendo de sus probabilidades. La cantidad óptima de bits a ser usados por cada símbolo es:

$$\log_2 \frac{1}{p}$$

donde p es la probabilidad de aparición de cada carácter. Por lo tanto, si la probabilidad de un carácter es 1/256, tal como se lo encontraría en un flujo aleatorio de bytes, la cantidad óptima de bits por carácter es:

$$\log_2 256 = 8$$

si la probabilidad fuese ½, la cantidad de bits necesarios sería 1. El problema con este esquema radica en el hecho de que los códigos Huffman deben poseer un número entero de bits. Por ejemplo, si la probabilidad de un carácter es 1/3, la cantidad óptima de bits para codificar ese carácter es alrededor de 1,6. El código Huffman debe sin embargo asignar ya sea 1 o 2 bits al código, y cualquiera de las dos opciones deriva en un mensaje comprimido más largo que el teóricamente posible.

Ésta codificación no óptima se vuelve un problema notable cuando la probabilidad de un carácter se vuelve muy alta. Si el método estadístico asigna a un carácter el 90% de aparición, el tamaño de código óptimo debería ser 0,15 bits. El código Huffman probablemente asignaría un bit al símbolo, el cual es 6,7 veces más largo de lo que se necesita.

Cómo funciona

La codificación aritmética sobrepasa completamente la idea de remplazar un símbolo de entrada con un código específico. En lugar de eso, toma un flujo de símbolos de entrada y lo remplaza por un único número de punto flotante. Cuanto más complejo el mensaje, más bits son necesarios en la salida. No fue hasta hace poco que métodos prácticos fueron encontrados para implementar el método en computadoras con registros de tamaño fijo.

La salida de la codificación aritmética es un único número menor que 1 y mayor o igual que 0. Este único número puede ser decodificado de manera única para crear el flujo exacto de símbolos de entrada. Para poder codificar el flujo, los símbolos deben tener asignados sus probabilidades. Luego, se asigna un rango en [0, 1) a cada letra, donde los rangos no se solapan y cada uno tiene un tamaño igual a la probabilidad de aparición de cada letra (los rangos incluyen al inicio pero no al final). Por ejemplo, en la frase "BILL GATES", las probabilidades son las siguientes:

Símbolo	Probabilidad	Rango
espacio	1/10 = 0,1	0 – 0,1
A	1/10 = 0,1	0,1 – 0,2
B	1/10 = 0,1	0,2 - 0,3
E	1/10 = 0,1	0,3 - 0,4
G	1/10 = 0,1	0,4 - 0,5
I	1/10 = 0,1	0,5 - 0,6
L	2/10 = 0,2	0,6 - 0,8
S	1/10 = 0,1	0,8 - 0,9
T	1/10 = 0,1	0,9 - 1,0

Una vez obtenido los rangos, podemos comenzar la codificación. Para proceder a la misma, definimos la función rango*(x)* de la siguiente manera:

1- Si |x| = 1, entonces rango(x) = [r_i, r_j), donde r_i y r_j representan al rango de x establecido en la tabla de rangos.

2- Para dos cadenas *a* y *b*, si rango(*a*) = [a_i, a_f) y rango(*b*) = [b_i, b_f) entonces $\text{rango}(ab) = [a_i + (\Delta a)b_i, a_i + (\Delta a)b_f), \Delta a = a_f - a_i$.

Para poder hallar el número de una cierta frase, primero debemos hallar su rango. Definamos la siguiente operación entre rangos:

rango(a) * rango(b) = rango(ab)

Entonces, rango($x_1x_2…x_n$) = rango(x_1) * rango(x_2) * … * rango(x_n). Haciendo que cada x_i sea un carácter de la entrada, podemos hallar rango(x) realizando la operación de manera repetida en cada carácter de x partiendo del primer carácter, luego el segundo, y así sucesivamente hasta el último carácter. Para el ejemplo anterior, los resultados son los siguientes:

```
Carácter     Inicio          Fin
-------------   ---------        ----------
                0,0              1,0
    B           0,2              0,3
    I           0,25             0,26
    L           0,256            0,258
    L           0,2572           0,2576
  SPACE         0,25720          0,25724
    G           0,257216         0,257220
    A           0,2572164        0,2572168
    T           0,25721676       0,2572168
    E           0,257216772      0,257216776
    S           0,2572167752     0,2572167756
```

El valor para “BILL GATES” es finalmente un número comprendido en [0,2572167752; 0.2572167756). Tomamos al límite inferior 0.2572167752. El mismo codificará a la cadena de manera única utilizando el esquema de codificación presente.

Para poder decodificar un número *n*, debemos identificar dentro del rango de cuál letra cae el número a decodificar. En el ejemplo anterior, cae en el rango de B, ya que el número 0.2572167752 se encuentra entre 0,2 y 0,3, por lo tanto, la primera letra de la frase es B. Para poder continuar, debemos hallar un número *n'* que identifique al resto de la palabra. Esto lo logramos de la siguiente manera.

$$
\begin{aligned}
& n \in \text{rango}(x) \\
& \equiv n \in \text{rango}(ab), a = \text{inicio}(x), b = \text{resto}(x) \\
& \equiv n \in \text{rango}(a) + (\Delta a)\,\text{rango}(b) \\
& \equiv n \in [a_i + (\Delta a)b_i, a_i + (\Delta a)b_f) \\
& \equiv a_i + (\Delta a)b_i \le n < a_i + (\Delta a)b_f \\
& \equiv (\Delta a)b_i \le n - a_i < (\Delta a)b_f \\
& \equiv b_i \le \frac{n - a_i}{\Delta a} < b_f \\
& \equiv \frac{n - a_i}{\Delta a} \in \text{rango}(b) \\
& \equiv n' = \frac{n - a_i}{\Delta a}
\end{aligned}
$$

Procedemos de esta manera hasta decodificar la frase completa. Para el ejemplo anterior, los resultados son los siguientes:

Número	Símbolo	Ini	Fin	Rango
0,2572167752	B	0,2	0,3	0,1
0,572167752	I	0,5	0,6	0,1
0,72167752	L	0,6	0,8	0,2
0,6083876	L	0,6	0,8	0,2
0,041938	SPACE	0,0	0,1	0,1
0,41938	G	0,4	0,5	0,1
0,1938	A	0,2	0,3	0,1
0,938	T	0,9	1,0	0,1
0,38	E	0,3	0,4	0,1
0,8	S	0,8	0,9	0,1
0,0				

Compresión

Veamos ahora el resultado de la codificación en términos de ahorro de espacio. Si quisiéramos codificar el número 0.2572167752, necesitaríamos por lo menos espacio para poder codificar la parte decimal del número, esto es 2572167752. A primera impresión, sin embargo, podemos notar que el ahorro será significativo, ya que de una cadena que normalmente necesita un byte por carácter para poder ser codificada (10 bytes en total), se ha reducido a un número de 10 dígitos (no caracteres). Sabemos que cada byte necesita 8 bits para poder ser codificado, en contraste con los $\log_2(10) \approx 3{,}32$ bits necesarios para codificar cada dígito decimal. En conclusión, de necesitar 80 bits para codificar la palabra "BILL GATES", hemos pasado a necesitar $\log2(2572167752) \approx 32$ bits. Debemos además saber la cantidad de bits que ocupa el número (en éste caso, 32). Si asignamos 8 bits más para esto, tenemos un total de 40 bits, con un ahorro total del 50,0%.

El caso anterior es, sin embargo, el peor de los mismos, ya que hemos tomado a un número que tiene una cantidad igual de dígitos que el límite inferior y superior. Hay situaciones en las cuales un número de menor cantidad de dígitos puede ser elegido. Por ejemplo, para la cadena "AAAAAAA" podemos elegir un número en (0,43046721; 0,4782969), pudiendo ser alternativas óptimas 0,44, 0,45, 0,46 o 0,47, todas con sólo 2 dígitos frente a 7 bytes (7 bits vs 56 bits, 87,5 % de ahorro).

Implementación

En la práctica, utilizar números en punto flotante se vuelve insuficiente incluso para cadenas cortas debido al límite en precisión. Una alternativa a esto es definir un tipo de dato de punto flotante que pueda almacenar una cantidad arbitraria de dígitos limitada únicamente por la cantidad de memoria disponible. Ésta alternativa es sin embargo ineficiente debido al número de cómputos requeridos para "mantener" al número y poder permitir que el mismo pueda ser operando en las expresiones aritméticas.

La alternativa válida utilizada actualmente es utilizar enteros de 16 o 32 bits para representar a los números de punto flotante, y trabajar con éstos enteros de manera directa aplicando ligeras modificaciones a los algoritmos.

La transformación Burrows-Wheeler

Michael Burrows y David Wheeler lanzaron al público un reporte de investigación en 1994 discutiendo trabajo que estuvieron realizando en el Centro de Investigación de Sistemas Digitales en Palo Alto, California. Su paper "Un Algoritmo de Compresión de Datos Sin Pérdida Basado en Ordenación de Bloques" presentaba un algoritmo de compresión basado en una previa transformación no publicada descubierta por Wheeler en 1983.

Mientras que el paper discute un conjunto completo de algoritmos para la compresión y descompresión, el corazón verdadero del paper consiste en el descubrimiento del algoritmo BWT.

La transformación de Burrows-Wheeler (BWT) transforma un bloque de datos en un formato que es extremadamente adecuado para compresión.

Ordenamiento

El algoritmo BWT toma un bloque de datos y lo ordena usando un algoritmo de ordenación. El bloque de salida resultante contiene exactamente los mismos elementos de datos con los que empezó, difiriendo sólo en el ordenamiento. La transformación es reversible, o sea, la ordenación original puede ser restablecida si pérdida de fidelidad.

Tomemos el siguiente ejemplo:

w	h	e	e	l	e	r

Primero, creamos una matriz con la palabra rotada en un carácter en cada fila.

C0	w	h	e	e	l	e	r
C1	h	e	e	l	e	r	w
C2	e	e	l	e	r	w	h
C3	e	l	e	r	w	h	e
C4	l	e	r	w	h	e	e
C5	e	r	w	h	e	e	l
C6	r	w	h	e	e	l	e

Para poder representar las cadenas de manera eficiente, no debemos crear cada cadena en memoria, sino que simplemente debemos almacenar la cadena inicial en un búfer y luego el índice de inicio de cada cadena. Por ejemplo, la cadena 3 tiene un índice de inicio 3, por lo que para poder representarla se debe tomar los caracteres 3 a 6 del búfer ("eler"), y luego seguir con 0, 1 y 2 ("whe"). El índice de cadena es el mismo que el índice de inicio, por lo que la i-ésima cadena inicia en el i-ésimo carácter del búfer. Luego, ordenamos las cadenas utilizando una comparación lexicográfica, utilizando funciones similares a strcmp o memcmp de C. Luego de ordenar las cadenas, las mismas quedan de la siguiente manera:

	I						F
C2	e	e	l	e	r	w	h
C3	e	l	e	r	w	h	e
C5	e	r	w	h	e	e	l
C1	h	e	e	l	e	r	w
C4	l	e	r	w	h	e	e
C6	r	w	h	e	e	l	e
C0	w	h	e	e	l	e	r

Hemos marcado las columnas Inicio (I) y Final (F) ya que son importantes por el siguiente motivo: los caracteres en F son los caracteres prefijos de los caracteres en I en las mismas filas. O sea, en C1, 'w' es el prefijo de 'h'.

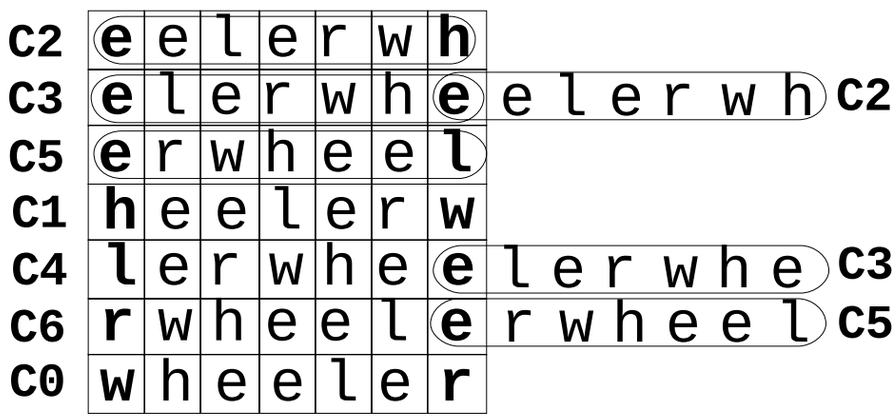
La salida de BWT consiste en dos cadenas: una copia de la columna L y un *índice primario*, un entero que indica la fila en la cual se encuentra la primera letra del búfer original, o sea, la posición de la cadena 1. En este caso, la primera cadena es "helwerr" y la segunda el índice 3.

Recomposición

Para poder recomponer C0, debemos notar que las letras en F son inmediatamente seguidas de las letras en I. Si podemos obtener el vector I, entonces podemos recomponer C0 a partir de estos pares de letras. Para poder obtener el vector I notamos que el vector I y el vector F contienen las mismas letras. Esto es así ya que el texto es rotado completamente, y cada letra pasa por cada columna exactamente una vez. Sin embargo, el vector I contiene estas letras de manera ordenada, debido al ordenamiento lexicográfico realizado inicialmente. Por lo tanto, para obtener I, simplemente debemos ordenar lexicográficamente a F.

Como ya hemos mencionado, al tener el vector I y F, podemos saber todos los pares de letras encontrados en C0. Para poder recomponer C0, primero nos situamos en la fila indicada por el índice primario. El vector F en esta fila contiene la primera letra de C0, ya que apunta a C1, el cual es C0 rotado una letra a la izquierda. La siguiente letra está en I en la misma fila. Para saber la siguiente letra, buscamos en F la fila en la cual la letra en I aparece. Una vez encontrada la fila, la letra en I en ella es el la siguiente letra. Repetimos el proceso hasta finalizar la construcción de C0.

Aunque el proceso es sencillo, ¿qué ocurre cuando existe más de una ocurrencia de una letra en F?, ¿cómo sabemos cual fila tomar?. Para responder a esta pregunta debemos primero saber que las palabras que inician en F están ordenadas lexicográficamente. Esto se debe a que el resto de las mismas inician en I, el cual fue previamente ordenado. Por lo tanto, como el orden lexicográfico ordena las palabras que empiezan con igual letra por la segunda, las palabras ordenadas en I están en el mismo orden que las que empiezan en F, esto es, lexicográficamente. Por lo tanto, la primera letra repetida en I corresponde a la primera letra en F, la segunda en I a la segunda en F, y así sucesivamente. El siguiente gráfico ilustra esto para el ejemplo anterior.



Los pares tomados, iniciado por el índice primario 3, son los siguientes: wh, he, ee, el, le, er, formando la palabra "wheeler", o sea C0.

La Transformación es apropiada para la Compresión

Ahora que ya sabemos cómo transformar una cadena, surge la siguiente pregunta: ¿Porqué la transformación Burrows-Wheeler es buena para la compresión?. Consideremos la siguiente cadena (subrayamos los textos para poder visualizar los espacios en blanco):

```
Un padre para cien hijos, antes que cien hijos para un padre
(Miguel de Cervantes).
```

Genera la siguiente salida:

```
ee,ealnnnnessa ss) (.rrpp vpp   aarduruiiCtti  ccMhhiieeeuUaajj
aaddeoeeonnqg r
```

Podemos ver que la cadena de salida contiene muchos caracteres repetidos seguidos unos de otros. Esto hace que la salida sea apropiada para la compresión run-lenght, la cual agrupa varios caracteres en un código que indica el carácter presente y la cantidad de repeticiones.

Compresión

Podemos tomar la salida de BWT y simplemente aplicar un compresor convencional al mismo, pero Burrows y Wheeler sugieren un mejor acercamiento. Ellos recomiendan utilizar un esquema de *Mover al Frente*, seguido por un codificador entrópico.

Un codificador MAF es una pieza de trabajo relativamente trivial. Simplemente mantiene en memoria todos los 256 códigos posibles en una lista. Cada vez que un carácter debe ser emitido, se envía su posición en la lista y luego se lo mueve al frente de la misma. Por ejemplo, la cadena "tttWtwttt", tiene una salida de enteros { 116, 0, 0, 88, 1, 119, 1, 0, 0 }.

Si la salida de la operación BWT contiene una gran porción de caracteres repetidos, podemos esperar que aplicando el codificador MTF nos dará un archivo lleno de muchos ceros, y muchos enteros pequeños. En este punto, el archivo puede finalmente ser comprimido por un codificador entrópico, típicamente un codificador Huffman o un codificador aritmético.

A continuación presentamos una tabla con datos comparativos:

Nombre	Tamaño	Tam. ZIP	Tam. BTW	Ahorro
bib	111,261	35,821	29,567	17,46 %
book1	768,771	315,999	275,831	12,71 %
book2	610,856	209,061	186,592	10,75 %
geo	102,4	68,917	62,12	9,86 %
news	377,109	146,01	134,174	8,11 %
obj1	21,504	10,311	10,857	-5,30 %
obj2	246,814	81,846	81,948	-0,12 %
paper1	53,161	18,624	17,724	4,83 %
paper2	82,199	29,795	26,956	9,53 %
paper3	46,526	18,106	16,995	6,14 %
paper4	13,286	5,509	5,529	-0,36 %
paper5	11,954	4,962	5,136	-3,51 %
paper6	38,105	13,331	13,159	1,29 %
pic	513,216	54,188	50,829	6,20 %
progc	39,611	13,34	13,312	0,21 %
progl	71,646	16,227	16,688	-2,84 %
progp	49,379	11,248	11,404	-1,39 %
trans	93,695	19,691	19,301	1,98 %
Totales		1072,986	978,122	8,84 %

Podemos concluir que BWT es un buen competidor del clásico ZIP, ya que un 8,84 % no es despreciable. Para que la compresión sea buena, el tamaño de bloque debe ser mayor a unos cuantos kilobytes.

Conclusión

He presentado en este trabajo las técnicas más populares de compresión sin pérdida de datos. Aunque existen sinnúmero de técnicas de compresión, creo que las presentadas pueden ilustrar el "estado del arte" de las técnicas actuales.

Bibliografía

www.dogma.net/markn/articles/bwt/bwt.htm

en.wikipedia.org/wiki/Burrows-Wheeler_transform

www.data-compression.info/Algorithms/BWT/

www.faqs.org/faqs/compression-faq/part2/section-9.html

www.data-compression.com/lossless.shtml

www.dogma.net/markn/articles/lzw/lzw.htm

www.dogma.net/markn/articles/arith/part1.htm