

LZP

Charles Bloom

Este método tiene como objetivo mejorar tanto la velocidad como el nivel de compresión del LZ77. Se trata de un híbrido entre un LZ y un compresor Predictor (de ahí el nombre). El compresor LZ77 define la posición del string matcheado como equiprobable es por eso que si tenemos una ventana de 64Kb debemos emitir una posición de 16 bits. Si de alguna manera pudiéramos saber de antemano la posición en donde se encuentra el mejor match (es decir, predecir esa posición) no tendríamos que guardarla en la salida y ahorraríamos espacio y velocidad (espacio al no guardar la posición y velocidad al no tener que buscar el mejor match). Lamentablemente la única forma de encontrar el mejor match es comparando con todos los strings de la ventana. Sin embargo podemos aproximarnos al mejor match utilizando un método predictor. LZP utiliza los últimos n caracteres (es decir un contexto de orden n) para predecir el mejor match. Por ejemplo, supongamos que estamos comprimiendo el mensaje “...los alumnos...” y estamos parados en la letra “a” entonces si utilizamos un contexto de orden 4, se utilizará el contexto “los\b” para predecir la posición del string que mejor matchea a “alumnos...”. El autor propone cuatro versiones que se diferencian en la velocidad y el nivel de compresión. Nosotros explicaremos la tercera (LZP3) que alcanza los niveles de compresión del Pkzip y es tres veces más rápida. Básicamente realiza los siguientes pasos: obtiene el contexto C de 4 bytes (los 4 caracteres anteriores a la posición actual). Luego calcula una dirección de 16 bits en base al contexto utilizando la siguiente fórmula: $H = ((C \gg 15) \wedge C) \& 0xFFFF$ (los operadores utilizados tienen el mismo significado que en el lenguaje C). Con H como índice, accede a una tabla que guarda la posición P del último string con el mismo H (candidato a mejor match) y el C (contexto) de ese string. Ahora si el C actual es distinto al guardado o la posición P es nula o no válida, se emite una longitud 0 en caso contrario se emite la longitud del match entre el string de la posición previa y el string de la posición actual. Luego (independientemente de lo que se haya emitido) se guarda en la posición H de la tabla la posición y el contexto actual. Por último emitimos el primer carácter no matcheado con un modelo de orden 1. Solo nos resta saber como codificamos las longitudes y el último carácter. Para las longitudes, el autor propone utilizar un compresor estadístico adaptativo (un huffman o uno aritmético) de orden 0 con 256 longitudes (0-255). El símbolo 255 significa “255 o más caracteres coincidentes”. Por ejemplo si tenemos un match de 511 emitimos 255, 255, 1. El último carácter no matcheado se comprime de la misma forma (es decir, con un compresor aritmético o huffman) pero utilizando un modelo de orden 1, en este caso el carácter anterior al no matcheado (el último carácter matcheado) se utiliza como contexto de orden 1. El siguiente cuadro es un pseudo-código del compresor:

```
C = Buffer[P-4] << 24 + Buffer[P-3] << 16 + Buffer[P-2] << 8 + Buffer[P-1]
H = ((C>>15)^C)&0xFFFF
if( Index[H].P != NULL && Index[H].C == C )
{
    L = I = ObtenerLongMatch( Index[H].P, P )
    while( I >= 255 )
    { EmitirLongitud( 255 ); I -= 255; }
    EmitirLongitud(I)
}
EmitirCaracter(Buffer[P+L-1], Buffer[P+L] )
Index[H].C = C; Index[H].P = P;
P += L+1.
```

Como dijimos anteriormente, existen muchas versiones del algoritmo básico LZP que se diferencian en la velocidad y el nivel de compresión. Pero la idea básica es predecir el mejor match de forma tal de no tener que emitir la posición en donde se encuentra y a su vez ahorrar tiempo en su búsqueda. Por eso vamos a ver un ejemplo sencillo de compresión que utiliza la idea básica de este método. Para ello tomaremos un contexto de orden 2, al ser un contexto pequeño no necesitamos hacer hashing puesto que todos los contextos posibles son 64K (un espacio de direcciones relativamente pequeño). También al no haber colisiones no hace falta guardar C. Al igual que el método anterior emitiremos un literal con un contexto de orden 1 luego de hacer cada match. Supongamos que queremos comprimir el siguiente mensaje:

“ABCCBABCCBCCBCC”

Procesamos el mensaje en un solo bloque (si el mensaje es grande deberíamos comprimir de a muchos bloques):

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

Para los dos primeros literales no tenemos contexto definido por lo que en general los emitiremos con un contexto de orden 1. Una solución posible podría ser (\b, A) y (A,B) (el primer elemento es el contexto y el segundo el carácter a emitir).

A partir del literal N° 2 resumimos los pasos como sigue:

POSICION	CONTEXTO	INDICE	LONG. MATCH	SALIDA
2	AB		0	0,(B,C)
3	BC	AB=2	0	0,(C,C)
4	CC	AB=2,BC=3	0	0,(C,B)
5	CB	AB=2,BC=3,CC=4	0	0,(B,A)
6	BA	AB=2,BC=3,CC=4,CB=5	0	0,(A,B)
7	AB	AB=2,BC=3,CC=4,CB=5,BA=6	3	3,(B,C)
11	BC	AB=7,BC=3,CC=4,CB=5,BA=6	2	2,(B,C)
14	BC	AB=7,BC=11,CC=4,CB=5,BA=6	1	1

Por lo tanto emitimos:

(\b,A),(A,B),0,(B,C),0,(C,C),0,(C,B),0,(B,A),0,(A,B),3,(B,C),2,(B,C),1.

Luego las longitudes se comprimen con un modelo estadístico de orden 0 y los literales con un modelo estadístico de orden 1. Ambos modelos deben ser adaptativos.