## Una Implementación del Algoritmo de Compresión Lempel-Ziv Welch.

V. G. Ruiz vi@iron.uaml.es
I. García inma@iron.ualm.es

Depto de Arquitectura de Computadores y Electrónica. Universidad de Almería. 04120. Cañada de San Urbano (Almería).

Resumen. En este trabajo, tras una introducción a los métodos de compresión más usados, se propone una implementación software del algoritmo de compresión de Lempel-Ziv Welch (LZW). LZW es un algoritmo de compresión libre de error que puede ser usado para comprimir datos de cualquier procedencia. Originalmente fue diseñado para ser implementado sobre un dispositivo hardware [1], pero debido a su rapidez y sencillez, su implementación software es también factible tal como lo demuestra su uso en el estándar de compresión de imagenes GIF [14]. En una sola pasada detecta y elimina la redundancia debida a la codificación y a la repetición de caracteres. En nuestro trabajo proponemos una implementación software de LZW, mostramos la respuesta de LZW en función de un conjunto de parámetros y hacemos un análisis comparativo con otros algoritmos de compresión.

### 1. Introducción a la compresión de datos.

La compresión de datos libre de error es el proceso a través del cual un conjunto de datos se transforma en otro más pequeño, pero ambos conteniendo la misma información. Este procedimiento se lleva a cabo mediante algún tipo de codificación que elimine las redundancias en los datos. Su principal utilidad se encuentra en el almacenamiento de información y en la transmisión de mensajes. En el primer caso necesitaremos una menor cantidad de espacio de almacenamiento y en el segundo, consumimos un ancho de banda menor del canal de comunicación. La figura 1 describe esquemáticamente los procesos de compresión y descompresión. Aunque el coste de los dispositivos de almacenamiento y de los medios de transmisión son cada vez menores y su capacidad y velocidad de transferencia aumenta vertiginosamente, el uso masivo de computadoras interconectadas, formando lo que se conoce con el nombre de autopistas de la información, hace que trabajar con datos no comprimidos sea costoso y lento. Frente a esta necesidad aparecen los algoritmos de compresión de datos, que tratan de eliminar cuanta redundancia sea posible. El tipo y cantidad de redundancia a eliminar depende de la naturaleza de los datos que estemos comprimiendo y caracteriza fuertemente al compresor y al descompresor. Existen tres tipos de redundancia, conocidos como redundancia en la codifi-

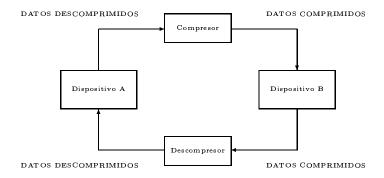


Figura 1. El dispositivo A posee un mayor ancho de banda o más capacidad de almacenamiento que el B.

cación, redundancia en la secuencia de aparición y redundancia psico-visual o psico-auditiva:

- ♦ Los compresores basados en la eliminación de la redundancia en la codificación aprovechan que la frecuencia de uso de los símbolos del alfabeto, que estamos utilizando para representar la información, no siga una distribución uniforme. Esto se debe a que usamos más frecuentemente unos caracteres que otros. En 1952 Huffman propuso el llamado algoritmo de compresión de longitud variable (variable-length coding) o algoritmo de Huffman [7], [8], que elimina este tipo de redundancia, que puede ser cuantizada por el primer estimador de la entropía. En otras palabras, el número de bits por símbolo necesarios para codificar una cadena de caracteres mediante el algoritmo de Huffman es igual a la entropía de la cadena. Los algoritmos que eliminan la redundancia en la codificación no provocan pérdida de información.
- ♦ Los algoritmos que eliminan la redundancia en la secuencia de aparición tienen en cuenta que normalmente existe algún tipo de correlación entre uno o varios de los símbolos ya aparecidos y él o los que van a aparecer. Por ejemplo, en los textos escritos en castellano, la probabilidad de aparición del carácter m delante del p es mucho mayor que la probabilidad de aparición del carácter n previo al p. En el ámbito del procesamiento de imágenes, este tipo de redundancia se denomina redundancia interpixel e indica la tendencia de que pixels próximos tengan niveles parecidos entre si, y formando patrones que se repiten en la imagen. Este tipo de redundancia puede ser calculado por los estimadores superiores de la entropía [7], [11], [1]. Existen gran cantidad de algoritmos que codifican la información eliminando este tipo de redundancia tales como RLC (Run Length Coding) [7], CAC (Constant Area Coding) [7] o el propio LZW [13] y que no provocan ninguna pérdida de información.
- ♦ La eliminación de la redundancia psico-visual y psico-auditiva tiene en cuenta que el ojo y el oído humano no responde con la misma sensibilidad a toda la información visual y auditiva. Existe cierta información que puede ser eliminada sin que se produzca un deterioro significativo de la calidad de la imagen o del sonido. La percepción humana de la información en una imagen no realiza un análisis cuantitativo de cada pixel o del valor de la luminancia en la imagen [7]. En general, un observador intenta fijarse en características tales como las fronteras, aristas o las texturas de las regiones, y las combina mentalmente para realizar asociaciones con otros grupos de patrones conocidos. El cerebro correlaciona esta información con el conocimiento anterior y así realiza la interpretación de la imagen [7]. La eliminación de este tipo de redundancia produce elevadas relaciones de compresión a costa de la pérdida irreversible de parte de la información. Todos los algoritmos de compresión que usan transformadas, tales como la transformada coseno [7], [6], [15], la rápida transformada de Hadamard [11], [7] o la computacionalmente costosa transformada Karhunen-Loève [7], [5], están basados en la eliminación de la redundancia psico-visual.

LZW es un algoritmo que elimina la redundancia en la codificación y la redundancia en la secuencia de aparición, y cuya principal característica, como mostraremos más adelante, es su sencillez de implementación y velocidad de ejecución. Es un algoritmo de compresión lossless, es decir, que los datos recuperados tras la compresión y descompresión son exactamente los datos de partida. Estas características son las que han motivado el inicio de este trabajo. Nuestro interés por los algoritmos de compresión se centra en su posible utilización en la transmisión de mensajes en un multiprocesador con memoria distribuida, en aquellas aplicaciones en las que el volumen de datos transmitidos es excesivamente grande, y por lo tanto incompatible con la obtención de una eficiencia adecuada. Se trata de aplicaciones paralelas que ven como su speed up no es el deseable debido a los tiempos de latencia de la red de interconexión

y que podrían aprovechar el funcionamiento concurrente que poseen la CPU y el hardware dedicado al enrutamiento, recepción y transmisión de mensajes. Por ejemplo, Vinod et al. [2] utilizan RLC para comprimir las imágenes raster que envían a través de la red de estaciones de trabajo interconectadas mediante red Ethernet que alcanza un ancho de banda máximo de 10M bits/seg. Nuestro trabajo puede considerarse como un primer acercamiento al campo de la compresión, es decir, una primera fase de análisis de los **algoritmos de compresión lossless**. El objetivo es obtener un algoritmo con una buena relación de compresión y cuyos tiempos de ejecución sean suficientemente cortos para que puedan ser utilizados como etapa previa al intercambio de mensajes en un sistema multiprocesador. El resto de este trabajo se ha organizado de la siguiente forma: en la sección 2 describimos los algoritmos LZW de compresión y descompresión. En la sección 3 presentamos las características de nuestra implementación de LZW. La sección 4 esta dedicada a la evaluación de los resultados y la comparación con otras soluciones estándar. Terminamos con una sección en la que exponemos nuestras conclusiones.

## 2. El algoritmo LZW.

#### 2.1. El codificador.

LZW es un algoritmo greedy adaptativo usado en la compresión de datos binarios, que en una sola pasada construye una tabla de cadenas (ST) en la que anota cada una de las diferentes secuencias de símbolos de entrada (normalmente bytes). LZW es adaptativo debido a que el contenido de la tabla de cadenas se genera a partir de los datos de entrada a comprimir. Su razón de compresión y tiempo de ejecución dependen de este proceso adaptativo. cuando la cadena de entrada no se encuentra en la tabla, el compresor emite un código cuya longitud depende del tamaño de la tabla construida hasta ese instante. El conjunto de todos los códigos generados forman la información comprimida (normalmente el fichero comprimido). El algoritmo inicial de Lempel-Ziv basa su compresión en convertir cadenas de longitud variable en símbolos de longitud fija (los códigos de compresión). Igual que el algoritmo de Huffman, que asigna a los caracteres de uso más frecuente una codificación mínima, LZW explota la redundancia en la codificación debido a que símbolos más frecuentes forman cadenas más largas en la tabla con lo que el número de bits por carácter disminuye, al ser el tamaño del código de salida independiente del tamaño de la cadena que codifica[1]. Por otra parte cualquier patrón repetido en la secuencia de caracteres de entrada es fácilmente detectable por LZW, por lo que también elimina la redundancia en la secuencia de aparición [1]. La eficacia de LZW en la eliminación de estos tipos de redundancia depende del tamaño de la tabla de cadenas. Es claro que una tabla de cadenas grande y llena emite menos códigos debido a que la probabilidad de encontrar la cadena de entrada es mayor, por lo que es interesante trabajar con tablas de cadenas grandes. El codificador trabaja entorno a dos variables llamadas raíz y prefijo, representadas por las letras k y w. La raíz contiene el símbolo que estamos comprimiendo y el prefijo contiene la cadena de símbolos que, previamente, ha sido localizada en la tabla de cadenas (ST). El compresor tiene en cuenta que si la cadena wk (la concatenación del prefijo y la raíz) existe en la tabla de cadenas, entonces la cadena w también existe en la tabla de cadenas. De esta forma, cadenas muy largas pueden ser almacenadas en la tabla, sin que esta sea excesivamente grande. Esto es debido a que las cadenas están repartidas de forma recursiva a lo largo ella. En cada entrada existe un carácter y una dirección donde se encuentra el siguiente carácter de la cadena. Quizás la característica más importante es que, a diferencia de otros algoritmos de compresión, por ejemplo el de Huffman [7], LZW no necesita enviar al decodificador la tabla de cadenas. LZW, en el proceso de descompresión, crea una tabla con las mismas cadenas que

la que el codificador creó en el transcurso de la compresión. El algoritmo básico del codificador es el siguiente:

```
1. Inicializar(ST)
      w \leftarrow Primer\_Caracter(IN)
      mientras No\_Fin(IN)
 3.
                 k \leftarrow Siguiente\_Caracter(IN)
 4.
                 si Existe(wk, ST)
 5.
                            w \leftarrow Direction(wk, ST)
 6.
 7.
                 sino
 8.
                            OUT \leftarrow w
 9.
                            ST \leftarrow wk
                            w \leftarrow k
10.
```

En el que se han usado los siguientes símbolos y variables:

IN: la fuente de símbolos que vamos a comprimir. OUT: donde almacenamos los códigos comprimidos.

ST: String Table (la tabla de cadenas).

w : la cadena prefijo. Contiene una cadena de símbolos o caracteres.

k : la raíz. Contiene un único carácter.

wk : la concatenación del prefijo con la raíz, formando una nueva cadena.

Iter	IN	W	k	OUT	tabla de cadenas					
0	'a'	97	-	Inic	direcc	w	k	izq(<=)	der(>)	
1	'nb'n	98	'nb'	97						
2	'a'	97	'a'	98	#0	-	0	-1	-1	
3	'nb'n	256	'nb'	_	:	:	:	:	:	
4	'c'	99	'c'	256	#97	-	'a'	262	256	
5	'n,	98	'nb'	99	#98	-	'n,	257	-1	
6	'a'	257	'a'	_	#99	-	'с'	259	-1	
7	'n,	98	'b'	257	:	:	:	:		
8	'a'	257	'a'	_	#255	-	255	-1	-1	
9	'n,	260	'b'	_		.=====				
10	'a'	97	'a'	260	#256	97	'n,	-1	258	'ab'
11	'a'	97	'a'	97	#257	98	'a'	-1	260	'ba'
12	'a'	262	'a'	_	#258	256	'c'	-1	-1	'abc'
13	'a'	97	'a'	262	#259	99	'n,	-1	-1	'cb'
14	'a'	262	'a'	_	#260	257	'n,	261	-1	'bab'
15	'a'	263	'a'	_	#261	260	'a'	-1	-1	'baba'
16	'a'	97	'a'	263	#262	97	'a'	263	-1	'aa'
17	'a'	262	'a'	_	#263	262	'a'	264	-1	'aaa'
18	'a'	263	'a'	_	#264	263	'a'	265	-1	'aaaa'
19	'a'	264	'a'	_	#265	264	'a'	-1	-1	'aaaaa'
20	'a'	97	'a'	264	:	:	:	:	:	
:	:	:	:	:						cadena codificac

En la tabla de la izquierda se describen los valores que van tomando las variables w y k, y las entradas y las salidas para cada iteración del codificador. En la tabla de la derecha se describe el contenido de la tabla de cadenas generada para este ejemplo. Es fácil ver que k (la raiz) es la variable que contiene el nuevo carácter a tratar (que proviene directamente de la entrada) y

que w (el prefijo) es el índice, en la tabla de cadenas, de la secuencia de símbolos reconocidos hasta ese instante y que ya están almacenados en la tabla. En cada entrada de la tabla de cadenas existen cuatro campos:

w: el prefijo. k: la raíz. izq: siguiente entrada en la tabla con raíz menor o igual. der: siguiente entrada en la tabla con raíz mayor.

Los campos izq y der forman un árbol binario cuyo fin es el de buscar eficientemente la cadena wk. Si el árbol se construye de forma equilibrada, en  $log_2(n)$  accesos se localiza wk, donde n es el número de entradas almacenadas en la tabla de cadenas. De una eficiente búsqueda de las cadenas wk en la ST depende el tiempo de ejecución del compresor.

#### 2.2. El decodificador.

La característica más importante del algoritmo LZW consiste en que no es necesario transmitir de la tabla de cadenas desde el compresor al descompresor, sino sólo los datos comprimidos. Como veremos, en cada iteración únicamente sabiendo cuáles son las raíces de la ST y los códigos de compresión, el descompresor genera una ST con idénticos contenidos al que el compresor usó para construir los códigos. Por otra parte, las necesidades computacionales del descompresor son muy diferentes. Empíricamente hemos encontrado que el decodificador es al menos, el doble de rápido que su respectivo codificador debido a que no se realiza búsquedas en la tabla de cadenas. Para saber si un código generado por el codificador existe en la tabla sencillamente vemos si es mayor que la entrada más grande almacenada en ella (la última que hemos almacenado siempre por el final). No es necesario el uso de técnicas de búsqueda y por tanto la estructura de la tabla de cadenas es más sencilla. El decodificador, en cada iteración incrementa la tabla con un nuevo código de entrada y emite uno o más caracteres que son exactamente los mismos que los que el compresor codificó. El principal problema a resolver en el descompresor consiste en encontrar la cadena asociada a un código de entrada. En cada entrada física de la tabla almacenamos un carácter (k) y sobre que entrada de índice inferior (w)se encuentra el siguiente carácter de la cadena (ver el ejemplo asociado y la estructura de la tabla de cadenas). En una dirección de la tabla de cadenas sólo almacenamos un carácter de cada una de las cadenas reconocidas por el compresor. La cadena descomprimida es formada visitando en sentido descendente cada una de las entradas en la tabla. En cada visita encontramos un carácter de la cadena que es concatenado a los que ya tenemos. Este proceso se acaba cuando extraemos el carácter almacenado en una de las 256 primeras direcciones de la tabla de cadenas que almacenan únicamente una raíz. La creación de esta pequeña tabla inicial se realiza en el paso 1 (inicialización de la ST). Una vez hallada la cadena, no puede ser directamente emitida, pues al recorrerse la ST en sentido descendente, se genera de forma invertida. La principal tarea que debe de realizar eficientemente el decodificador es la inversión de cadenas. Esto se puede realizar mediante soluciones recursivas o iterativas, siendo estas últimas las más eficientes. A continuación se muestra la estructura algorítmica del descompresor.

```
1. Inicializar(ST)

2. code \leftarrow Primer\_Caracter(IN)

3. OUT \leftarrow code

4. old \leftarrow code

5. mientras No\_Fin(IN)

6. code \leftarrow Siguiente\_Caracter(IN)

7. w \leftarrow old
```

```
8.
                  si Existe(code, ST)
                             OUT \leftarrow String(code)
 9.
10.
                  sino
11.
                             OUT \leftarrow String(w), k
12.
                  k \leftarrow Primer\_Caracter(OUT)
                  ST \leftarrow wk
13.
                  old \leftarrow code
14.
```

Donde la terminología utilizada es la siguiente:

IN: entrada de la que provienen los códigos comprimidos.

OUT: salida sobre la que almacenamos los caracteres originales descomprimidos.

ST: tabla de cadenas. code: código de entrada comprimido.

w: el prefijo, normalmente una cadena con más de un símbolo.

k: una raíz.

wk: la concatenación de la cadena w y el carácter k.

old: un estado anterior para code.

String(code): la cadena asociada a la entrada code.

String(code), k: la concatenación de la cadena String(code) y el carácter k.

Primer\_Caracter (OUT) : el primer carácter de la última cadena escrita sobre OUT.

Volviendo a nuestro anterior ejemplo:

97,98,256,99,257,260,97,262,263,264,265,266,267,97,...

		old	w	k		tabla de cadenas			
IN	code				OUT	direcc	W	k	
97	97	97	-	-	97=a	#0	-	0	
98	98	98	97	98	98=b	:	:	:	
256	256	256	98	97	256=ab	#97	-	97=a	
99	99	99	256	99	99=c	#98	-	98=b	
257	257	257	99	98	257=ba	#99	-	99=c	
260	260	260	257	98	257,98=bab	:	:	:	
97	97	97	260	97	97=a	#255	-	255	
262	262	262	97	97	97,97=aa			======	
263	263	263	262	97	262,97=aaa	#256	97	98=ab	
264	264	264	263	97	263,97=aaaa	#257	98	97=ba	
265	265	265	264	97	264,97=aaaaa	#258	256	99=abc	
:	:	:	:	:	:	#259	99	98=cb	
						#260	257	98=bab	
						#261	260	97=baba	
						#262	97	97=aa	
						#263	262	97=aaa	
						#264	263	97=aaaa	
						#265	264	97=aaaaa	
						:	:	:	

En la tabla de la izquierda mostramos los valores que toman las variables usadas en el descompresor y en la tabla de la derecha se muestra cómo quedaría la ST. Podemos observar que posee las mismas cadenas que la tabla construida por el compresor, aunque ahora la estructura es más sencilla al no existir proceso de búsqueda. En cada entrada tenemos un prefijo (w) y una raíz (k). Por ejemplo, para construir la cadena asociada a la entrada 261, anotamos en primer lugar su raíz (una a) y saltamos a la entrada 260. Allí encontramos que la raíz es b, con lo que ya tenemos ab. Como aún no hemos alcanzado una entrada que corresponda a una raíz, saltamos a la entrada 257, con lo que obtenemos aba. Por último saltamos a la entrada 98 que corresponde a una raíz, obteniendo la cadena abab. Falta invertirla y estará lista para ser emitida como cadena descomprimida: baba.

#### 2.3. La complejidad del compresor y del descompresor.

En la figura 2 mostramos el comportamiento de los algoritmos de compresión y descompresión en función del tamaño del problema, que en nuestro caso es la longitud del fichero que compactamos. Los tiempos fueron tomados sobre una SUN SparcStation IPC, que comprimió un

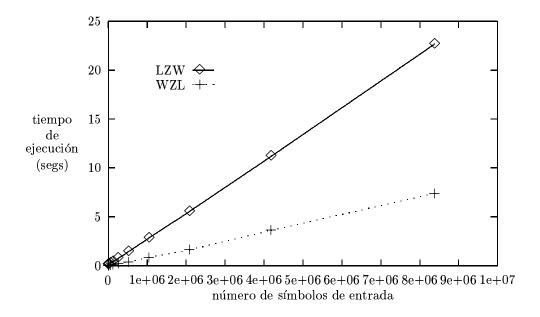


Figura 2. Tiempos de compresión y descompresión en función del tamaño del problema.

fichero creado con la utilidad mkfile del sistema operativo UNIX que permite generar ficheros de cualquier longitud. Puede observarse que tanto el tiempo de compresión como el de descompresión son lineales, lo que indica que los dos procesos presentan una complejidad de O(n) donde n es el tamaño del fichero de datos utilizado en la compresión y descompresión.

## 3. Implementación del algoritmo LZW.

En este apartado analizamos opciones de diseño del algoritmo LZW, tales como el algoritmo de búsqueda en la tabla de cadenas utilizado por el compresor, la longitud de los códigos de salida, la utilización de buffers y la longitud de la tabla de cadenas. Este análisis justifica las decisiones de diseño que hemos adoptado para nuestra implementación con el objetivo de mejorar el rendimiento del algoritmo.

#### 3.1. Algoritmos de búsqueda del compresor.

Como se aprecia en el paso 5 del algoritmo del compresor, es necesario saber si la cadena wk está almacenada en la tabla de cadenas. Esta tarea puede ser llevada a cabo mediante los siguientes algoritmos de búsqueda:

 $\diamond$  **Búsqueda secuencial.** Es conocido que, aunque la programación de esta búsqueda es cómoda y sencilla, este algoritmo arroja una complejidad de O(n) donde n es el tamaño del problema. Por tanto, es computacionalmente demasiado costosa debido a la gran cantidad

de comparaciones que son necesarias. Únicamente sería utilizable en tablas de cadenas muy pequeñas.

- Mediante técnicas de hashing. Probablemente, este es el mejor método para localizar la cadena wk dentro de la tabla, va que si la función de transformación se ha escogido convenientemente, y no se producen colisiones, necesitaremos realizar un sólo acceso a la tabla de cadenas para comprobar si la cadena wk buscada está almacenada en ella. Sin embargo, en LZW se plantea un problema adicional relacionado con la longitud de los códigos de compresión emitidos. Es conocido que el almacenamiento en una tabla hash no se realiza de forma incremental, sino que la tabla se llena aleatoriamente dependiendo de la entrada que estamos almacenando y de la función de transformación. Si consideramos un tamaño máximo de 65536 entradas en la tabla de cadenas, el primer almacenamiento puede producirse en la entrada 60000 (por ejemplo). Esto quiere decir que desde el comienzo del proceso de compresión se necesitaran 16 bits para codificar el código de salida, lo que no es deseable cuando la tabla de cadenas está casi vacía. Si como en el ejemplo, inicializamos la tabla con 256 raíces, y llenamos la tabla por el final, esto es, a partir de la entrada 256, 9 bits serían suficientes para almacenar los códigos de salida, mientras que el tamaño de la tabla sea inferior a 512. En ese instante se pasará a emitir 10 bits por código. Este comportamiento es difícil de conseguir mediante técnicas de hashing.
- Mediante búsqueda binaria. Una búsqueda de este tipo únicamente se puede realizar cuando el conjunto de elementos sobre el que lanzamos la búsqueda está ordenado. Una forma cómoda de construir la tabla de cadenas, ordenada en función de la siguiente raíz a buscar, es el árbol binario que se ha utilizado en nuestra implementación de LZW. Si el árbol construido en la creación dinámica de la tabla de cadenas es un árbol equilibrado, entonces el número de accesos a la tabla de cadenas, para encontrar la cadena wk, sería  $log_2(n)$ , donde n es el tamaño de la tabla de cadenas. Sin embargo, aunque esta situación no suele ser la mas frecuente, podemos asegurar que en el peor de los casos, con una búsqueda binaria se realizarán muchos menos accesos a la tabla de cadenas que con un algoritmo de búsqueda secuencial. Aunque el tiempo de búsqueda en la tabla de cadenas es mayor que el que se necesita con la técnica hashing, en este caso tenemos la ventaja de poder ir llenando la tabla por su final, debido a que primero insertamos una nueva cadena wk y a continuación enlazamos esta nueva entrada con la última entrada visitada en el proceso de búsqueda v que contiene todos los símbolos de la cadena w. Ya que insertamos por el final la cadena wk, en función de si la nueva raíz k es mayor o menor que el último carácter de la cadena w, el campo izg o der de la entrada w está apuntando a la dirección de wk. De esta forma se construye el árbol binario de búsqueda. En nuestra implementación el compresor realiza una búsqueda binaria  $(log_2(n))$  accesos donde n es el tamaño máximo de la ST), mientras que el descompresor únicamente comprueba si un código está en la tabla (un acceso), comparándolo con el número de cadenas almacenadas (el tamaño de la ST). Esta es la razón por la cuál, mientras el compresor gasta más tiempo en mejorar la razón de compresión (trabajando con ST más largas), el descompresor realiza su tarea de forma independiente del tamaño de la ST, tal y como se demuestra en la figura 3. Podemos apreciar que el descompresor incluso mejora su tiempo de ejecución debido a que necesita leer menos códigos de entrada comprimidos y realizar menos vaciados de su tabla de cadenas.

#### 3.2. La generación de códigos de salida comprimidos de longitud variable.

Igual que ocurre en el estándar de compresión GIF, nuestra implementación de LZW emite códigos comprimidos de longitud mínima en función del tamaño de la tabla de cadenas generada

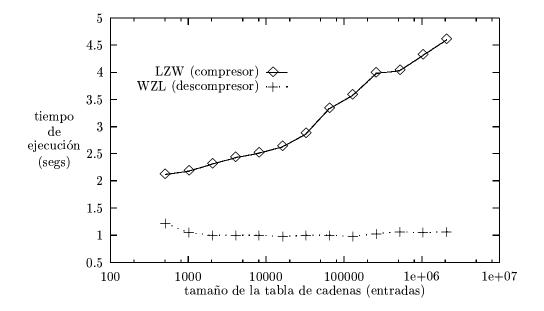


Figura 3. El tiempo de compresión frente al de descompresión.

en cada instante. Esto es importante pues emitiendo sólo el número de bits necesarios para codificar los códigos aumentamos la razón de compresión. Sin embargo, el tiempo de ejecución del algoritmo crece al tener que trabajar con cadenas de bits cuya longitud ya no es múltiplo de 8.

#### 3.3. Minimización el número de accesos a los dispositivos de entrada y salida.

Analizando el algoritmo del codificador vemos que accedemos al dispositivo de entrada cada vez que necesitamos un carácter y cuando generamos un código. El decodificador se comporta de forma similar. Aunque el sistema operativo UNIX utiliza buffers de memoria para minimizar el número de accesos a los dispositivos de I/O, el uso de buffers de memoria auxiliares aumenta el ancho de banda de entrada y salida desde los dispositivos. Con este fin se han implementado buffers de entrada y salida y la justificación de su utilización aparece representada en la figura 4, donde se compara el rendimiento del mismo algoritmo con y sin buffers para la entrada y salida de datos. Los tiempos han sido tomados sobre una SUN SparcStation 10, comprimiendo un fichero postscript de 1121203 caracteres. Comparamos los 3.567 segundos, que es el tiempo de compresión sin el uso de buffers, con el tiempo de compresión en función del tamaño de los buffers de entrada y salida.

#### 3.4. Tamaño de la tabla de cadenas.

El tamaño de la tabla de cadenas en el algoritmo LZW es un parámetro crítico, que afecta tanto a la razón de compresión, como al tiempo de ejecución. A diferencia del algoritmo usado en el estándar GIF que funciona a partir de una tabla de cadenas de 4096 entradas, nuestra implementación de LZW se ha realizado para trabajar con tablas de cualquier tamaño. Esto nos ha permitido evaluar la eficiencia del compresor en función del tamaño de la tabla de cadenas, y cuyos resultados se presentan en la sección 4.

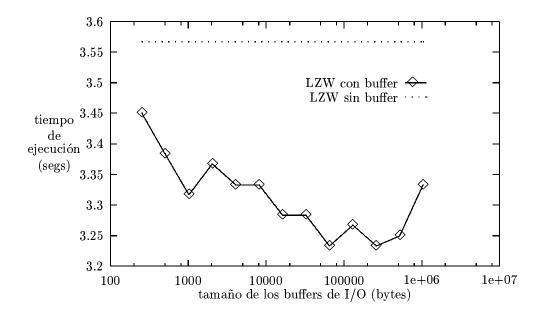


Figura 4. Influencia del tamaño de los buffers de I/O sobre el tiempo de ejecución de LZW.

# 4. Evaluación de la implementación de LZW: tiempos y razones de compresión.

En esta sección presentamos los resultados obtenidos en la ejecución de LZW sobre diversos computadores, y la comparación de los tiempos y razones de compresión de LZW con otros algoritmos. En la figura 5 hemos representado los tiempos de compresión de LZW en función del tamaño de la tabla de cadenas, usando un fichero de 1121203 bytes. Los tiempos fueron medidos sobre 5 computadores diferentes, obteniéndose los mejores valores para la estación de trabajo de Digital con un DECchip 21064A a 233 MHz, y resaltando la falta de utilidad de la capacidad de procesamiento vectorial del Convex 240 sobre esta aplicación. Las razones de compresión obtenidas para LZW varían entre el 42%, para un tamaño de la tabla de cadenas de 1024, y el 30%, para un tamaño de 262144. Hemos observado que a partir de un determinado tamaño de la tabla de cadenas la razón de compresión no mejora. Esto se debe a que la ST no se llena completamente en ningún instante de la compresión, pero evidentemente esto dependerá del tamaño y características de los datos con los que se esté trabajando. En la figura 6 mostramos los tiempos y razones de compresión obtenidos por la utilidad compress del sistema operativo UNIX y por los programas gzip y zip (con la opción -9). La flecha vertical indica el punto en el que LZW comienza a comprimir más que compress. Todas las muestras fueron tomadas sobre una SUN SparcStation 10. Podemos observar que LZW es unas 10 veces mas rápido que gzip o zip, aunque los valores de la razón de compresión de LZW son aproximadamente la mitad que los obtenidos para gzip o zip. Los resultados obtenidos con nuestra implementación son similares a los que se obtienen con el algoritmo compress, aunque tenemos la ventaja de poder modificar el tamaño de la tabla de cadenas, de forma que podemos siempre utilizar de la forma más adecuada los recursos disponibles para obtener el mejor rendimiento. En relación con compress podemos decir que LZW obtiene la misma razón de compresión consumiendo el 75%del tiempo que necesita compress. Para valores mas grandes de la tabla de cadenas obtenemos una razón de compresión ligeramente mejor, aunque en este caso el tiempo de compresión es muy similar al de compress.

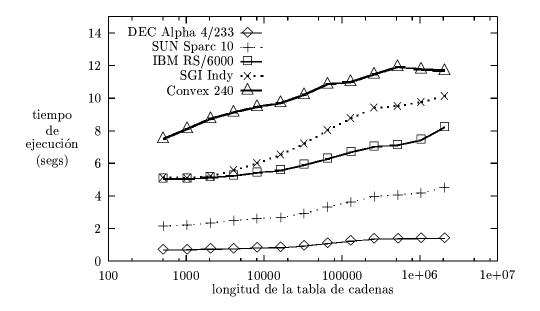


Figura 5. Tiempo de compresión de LZW sobre diferentes arquitecturas.

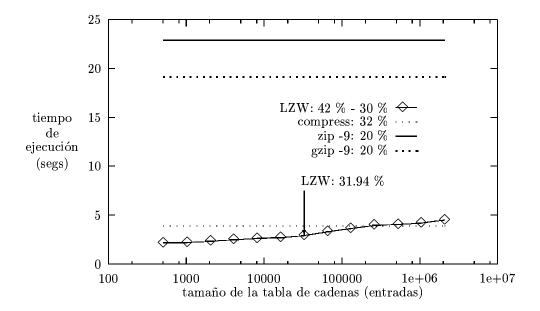


Figura 6. Comparación de los tiempos y razones de compresión de varios algoritmos.

#### 5. Conclusiones.

LZW es un algoritmo cuya principal ventaja es la de ser rápido, por lo que es muy usado en la compresión lossless de ficheros de datos antes de su almacenamiento o transmisión. El rendimiento del compresor se ve influenciado por el tamaño de la tabla de cadenas con la que trabaja, pues al incrementarla, aumentamos el tiempo de ejecución junto con la razón de compresión. Hemos implementado una versión del algoritmos de Lempel-Ziv Welch, con un parámetro libre (tamaño de la tabla de cadenas) que llega a ser competitivo en la razón

de compresión y el tiempo de ejecución con el algoritmo compress de UNIX, aunque como hemos visto la relación de compresión es inferior a la de los algoritmos zip o gzip, con los que puede competir en el tiempo de ejecución. Estamos pues en situación de hacer un análisis de la utilidad de LZW en el paso de mensajes en un sistema multiprocesador. No obstante, pensamos que algunas modificaciones serán todavía necesarias para adaptar el algoritmo a este tipo de aplicaciones. Por otro lado, también estamos interesados en las técnicas de compresión lossy y pensamos que quizás LZW pueda adaptarse a este tipo de compresión si realizamos búsquedas aproximadas (no exhaustivas) en la tabla de cadenas, con lo que trabajaremos con tablas más pequeñas (al aumentar el número de búsquedas con éxito) y emitiremos códigos de menor longitud y por tanto, las razones de compresión y la velocidad de ejecución aumentarán. El precio que se paga es la pérdida de información de parte de los datos originales, cuya aplicabilidad aparece en el campo del procesamiento de imagenes.

## Bibliografía

- 1. T. A. Welch, A Technique for High-Performance Data Compression, IEEE Computer, Jun 1984, pp: 8-19.
- V. Anupam, C. Bajaj, D. Schikore, and M. Schikore, Distributed and Collaborative Visualization, IEEE Computer, July 1994, pags: 32-43.
- 3. E. Shusterman and M. Feder, Image Compression via Improved Quadtree Decomposition Algorithms, IEEE Transactions on Image Processing, vol 3, no 2, March 1994, pags: 207-215.
- C. M. Huang and R. W. Harris, A Comparison of Several Vector Quantization Codebook Generation Approaches, IEEE Transactions on Image Processing, vol 2, no 1, January 1993, pags: 108-112.
- A. K. Jain, A Fast Karhunen-Loéve Transform for a Class of Random Process, IEEE Transactions on Communications, September 1976, pags: 290-296.
- B. C. Smith, L. A. Rowe, Algorithms for Manipulating Compressed Images, IEEE Computer Graphics and Applications, September 1993, pags: 34-42.
- 7. R. C. Gonzalez, P. Wintz, Digital Image Processing, Addison-Wesley, 1988.
- 8. W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes in C, Cambridge University Press, 1988.
- 9. N. Dale, S. C. Lilly, Pascal y Estructuras de Datos, McGraw-Hill, 1989.
- 10. Robert Sedgewick, Algorithms, Adison-Wesley, 1988.
- 11. V. G. Ruiz, I. G. Fernández, *Una Solución Paralela a la Compresión de Imágenes*, Nuevas Tendencias en la Informática: Arquitecturas Paralelas y Programación Declarativa, 1994, pags: 185-199.
- 12. A. Rosenfeld, A. C. Kak, Digital Picture Processing, Academic Press, 1982.
- 13. S. Blackstock, LZW and GIF explained, Internet news, 21 April 1989.
- 14. Compuserve Incorporated, Graphics Interchange Format (GIF) Specification, (c) Compuserve Incorporated, June 15, 1987.
- 15. A. C. Hung, PVRG-MPEG CODEC 1.1, September 15, 1994.