Mu-Zip

Alejandro Fernández Suárez

Marc Sunet Pérez

Cómo usar mu-zip

Para **comprimir** una imagen PPM:

```
muzip <nombre-imagen.ppm>
```

Es importante que el nombre del archivo acabe en .ppm porque el programa usa la extensión para saber si tiene que comprimir o descomprimir. El archivo resultante será el equivalente pero con extensión .mz.

Lo mismo pasa al **descomprimir**, le damos un archivo acabado en .mz y nos generará el correspondiente .ppm. El comando seria:

```
muzip <nombre-archivo-muzip.mz>
```

No obstante, se puede **dar un nombre al archivo de salida** a continuación del de entrada para utilizarlo en lugar del que se asigna por defecto.

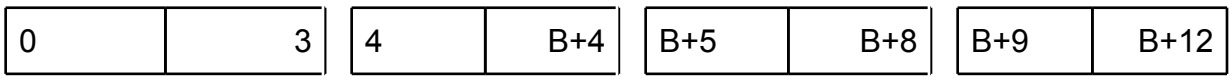
```
muzip <.ppm/.mz> [output]
```

Al comprimir una imagen PPM, se pueden configurar los parámetros alfa (número real que idenftifica la permisividad de compresión, a mayor valor, más se comprime, pero se pierde más detalle de la imagen original, por defecto es 255.5), p y q (tamaño de los bloques de compresión, donde p es el numero de filas por bloque y q las columnas, por defecto son N/64 y M/64, respectivamente), basta con llamar al compresor de la siguiente manera:

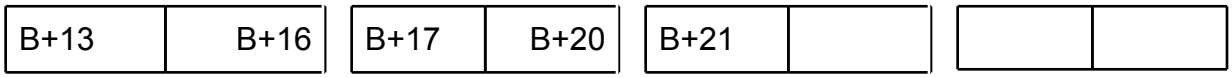
```
muzip <*.ppm> [output] [p] [q] [alfa]
```

Formato del archivo muzip (.mz)

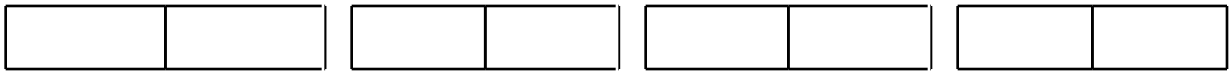
El archivo mz tiene el siguiente aspecto:



B = Tamaño blob Huffman | Blob Huffman | p | q



M | N | pixeles bloque 1 | pixeles bloque 2



pixeles bloque ... | pixeles bloque ... | pixeles bloque ... | pixeles bloque J

Codificación de Huffman

Para la realización de la prática se ha implementado una codificación de Huffman genérica. La función de codificación trata una secuencia de valores de un tipo dado y produce un blob binario para su posterior decodificación.

Algoritmo de Codificación

En general, el algoritmo de codificación no presenta ninguna peculiaridad salvo su genericidad.

Dada una secuencia de valores de un cierto alfabeto:

1. Anotamos la frecuencia de aparición de cada símbolo en la secuencia.
2. Creamos una hoja por cada símbolo y las almacenamos en una PQ.
3. Mientras haya más de un elemento en la PQ:
4. Sacar los dos nodos de menor frecuencia.
5. Crear un nuevo nodo con los dos anteriores como hijos y con frecuencia igual a la suma de esos dos.
6. Insertar el nuevo nodo en la PQ.
7. Saltar al paso 3.

Finalmente, una vez construído el árbol de Huffman procedemos a codificar la secuencia dada como una secuencia de bits.

La única característica de la implementación es que en la construcción del árbol las frecuencias de aparición no se almacenan en los nodos, ya que una vez construído el árbol nos es irrelevante qué frecuencia tenga cada símbolo. Para acomodar el cambio, en vez de mantener una cola de nodos mantenemos una cola de pares (nodo, frecuencia).

Serialización de la tabla y secuencia codificada

La serialización de la tabla y la secuencia puede resultar nefasta si se hace ingenuamente.

Empezamos comentando la serialización de la secuencia ya que es la más sencilla y nos será últil para explicar la serialización de la tabla.

Serialización de la Secuencia

Dada nuestra implementación, disponemos de una secuencia de bools que representan los bits de la codificación en Huffman de la secuencia original.

Para aprovechar al máximo el espacio, debemos agrupar estos bools en grupos de ocho y almacenar estos grupos en un byte. Por ejemplo, dada la secuencia

0 1 1 0 0 1 0 0

el byte resultante sería 0x64.

Es posible que la secuencia de bools no sea un múltiple de ocho. En tal caso, al último byte le sobrarán algunos bits. Por ejemplo, dada la secuencia

0 1 1 0 0 1 0 0 **1 1**

su serialización serían los bytes 64 C0.

Debemos pensar ahora qué sucede al deserializar los bytes anteriores. Dados los bytes 64 C0, qué secuencia de bits representan? Es la secuencia

0 1 1 0 0 1 0 0 **1 1**

o es la secuencia

0 1 1 0 0 1 0 0 **1 1 0 0** 0 0 0

No podemos saberlo ya que no sabemos qué bits de los 16 que hay en 64 C0 son relevantes.

Para resolver la ambigüedad, al serializar la secuencia de bits precedemos los bytes resultantes por un número que representa el número relevante de bits en esos bytes. Por ejemplo, en el caso anterior la serializacón sería:

0B 64 C0
El valor 0xB (12) nos indica cuantos bits de los bytes 64 C0 son relevantes, o visto de otra forma, la cantidad de bits que había en la secuencia original. La deserialización de 0B 64 C0 es ahora inambigua.

El convenio acordado hasta ahora de utilizar el primer byte como el número de bits

relevantes en la secuencia tiene una seria limitación, y es que con un solo byte sólo podremos almacenar secuencias de hasta 255 bits.

A continuación introducimos otra mejora: en función de cuántos bits almacenemos utilizamos un tamaño u otro para representar ese valor. Por ejemplo, si no almacenamos más de 255 bits utilizamos un byte, si no almacenamos más de 65535 utilizamos un word, y en caso contrario utilizamos un dword.

Este nuevo convenio nos permite almacenar secuencias de bits más largas y a la vez nos asegura el no desperdiciar espacio al almacenar el valor en si. Por ejemplo, podríamos utilizar siempre un dword para representar el número de bits en la secuencia, pero en el caso de que hubiese menos de 255 estaríamos desperdiciando 3 bytes, y en el caso de que hubiere menos de 65535 desperdiciaríamos 2 bytes.

Siguiendo con el ejemplo anterior, la serialización de

0 1 1 0 0 1 0 0 **1 1**

seguiría siendo

0B 64 C0

ya que con un solo byte podemos codificar el 12, el número de bits en la secuencia en este caso.

No obstante, todavía nos falta por cubrir un último detalle. Cuál es ahora la deserialización de 0B 64 C0? Cómo sabemos si se ha utilizado un byte, un word, o un dword para representar el número de bits en la secuencia?

Para resolver la ambigüedad, precedemos al valor númerico por un byte que nos indica qué tamaño tiene en bytes ese valor númerico. Tomaremos:

0 = byte
1 = word
2 = dword
La serialización de la secuencia ahora sería:

00 0B 64 C0

El 00 nos indica que se ha utilizado un byte para representar el número de bits en la

secuencia. A continuación leemos el 0B, indicándonos que sólo 12 bits son relevantes en 64 C0, y con eso obtendríamos la secuencia original.

Finalmente, cabe destacar que en ningún momento hemos hablado de char, short o int, ya que los tamaños de estos tipos varían entre plataformas.

Además, en el caso de que se utilice un word o un dword almacenamos el valor en little endian, así que una máquina big endian debe reordenar los bytes para poder interpretarlos correctamente.

Serialización de la tabla

La tabla de Huffman está compuesta de:

- Los símbolos del alfabeto.
- La codificación binaria de cada símbolo.

Debemos serializar la tabla en una secuencia de bytes que podamos utilizar para la posterior reconstrucción de la misma.

Para aprovechar al máximo el espacio, desentrelazamos los símbolos y su codificación. De esta manera podemos agrupar la codificación en grupos de 8 bits y almacenar cada grupo en un byte. Por ejemplo, dada la siguiente tabla:

Símbolo	Codificación
a	00
b	10
c	01
d	11

Al desentrelazarla en dos arrays de símbolos y codificación, nos queda:

array de símbolos	a b c d
array de codificación	0 0 1 0 0 1 1 1

En el array de codificación simplemente metemos los bits de todos los símbolos, empezando por el primer símbolo de la tabla y avanzando hasta el último.

Este ejemplo es un tanto engañoso porque todos los símbolos se representan con dos bits, pero en general los símbolos tendrán un número variable de bits (de eso trata la codificación de Huffman). Debemos entonces añadir una última pieza de información para saber qué bits del array de codificación codifican qué símbolo del alfabeto.

Como en el array de codificación aparecen los bits de los símbolos en el mismo orden en que aparecen los símbolos en el array de símbolos (primero los bits de a, luego los de b, etc.), lo que hacemos es anotar, en un tercer array, en qué bit empieza la codificación de cada símbolo.

En el ejemplo anterior, nos quedarían los siguientes arrays:

array de símbolos	a b c d
array de codificación	0 0 1 0 0 1 1 1
array de índices	0 2 4 6 8

Además, añadimos un último índice (en este caso el 8) para indicar el final de la secuencia de bits de la codificación.

Para deserializar la tabla, leemos el primer símbolo (a) y el primer índice (0). A continuación vamos leyendo bits del array de codificación hasta que llegamos al siguiente índice (2). En tal caso, leemos el siguiente símbolo del array de símbolos y repetimos el proceso hasta que se han leído todos los símbolos.

Almacenamiento de datos

Para mantener aislada la representación específica del problema de los algoritmos de compresión y descompresión, así como de la estructura de datos que permite acceder por bloques a una tabla de datos.

Las clases *Bloque* y *Pixel*

Como la implementación de los algoritmos de compresión son completamente genéricos, se crearon estas clases que encapsulan los datos de un pixel de imagen PPM completa y un bloque de una matriz específica.

- La clase *Pixel* permite acceder al pixel de una imagen de forma individual y es la encargada de calcular la “distancia” entre dos píxeles dados.

- La clase *Bloque* encapsula el tipo de datos con el que trabaja la estructura de datos métrica (el GHT), y también implementa la operación de distancia entre bloques. De esta manera, el GHT genérico solo requiere que el tipo de datos con el que se instancia la clase implemente el operador de “resta”.

La clase *Matriz*

Es la estructura de datos encargada de indexar una tabla de elementos como si fuera una matriz compuesta por bloques.

Funciones de compresión y descompresión

La función de compresión almacena en una tabla los indices de los bloques que componen la imagen comprimida y guarda la dirección de inicio de cada bloque en la imagen original. Es por esto que es necesario compactar los datos después de comprimir, para hacer que cada bloque requerido para la imagen comprimida esté almacenado de forma contigua en memoria y poder almacenarlo en un archivo.

La descompresión es mucho más sencilla, una vez se ha recuperado del archivo el “cubo” que contiene los bloques y la tabla de índices que compone la imagen, simplemente se carga todo en una matriz que indexa sobre la nueva imagen que se está descomprimiendo, produciendo así el la imagen resultado.