Revista de Educación Matemática
Volumen 32, N°1 (2017), páginas 25 – 36
Unión Matemática Argentina - Famaf (UNC)

# COMPRESIÓN DE IMÁGENES CODIFICACIÓN DE HUFFMAN

Javier Lezama

## §1. Introdución

La compresión de datos digitales, imágenes digitales, es el proceso de reducción del volumen de datos para representar una determinada cantidad de información. Es decir, un conjunto de datos puede contener datos redundantes que son de poca relevancia o son datos que se repiten en el conjunto, los cuales si se identifican pueden ser eliminados.

El ojo humano responde con diferente sensibilidad a la información visual que recibe. La información a la que es menos sensible se puede descartar sin afectar a la percepción de la imagen, suprimiéndose lo que se conoce como redundancia visual, y produciéndose a la vez la pérdida de ciertas características de la imagen.

La compresión de las imágenes digitales se ha hecho imprescindible debido a que el tamaño de los archivos de imagen es cada vez mayor. La compresión de una fotografía es la reducción de los datos digitales que no resultan necesarios e importantes. Esta compresión permite almacenar mayor número de imágenes al conseguir que los archivos resultantes no ocupen mucho espacio.

El gran tamaño de los archivos de imagen ha hecho que se desarrollen muchos métodos para comprimir datos. La compresión es una parte muy importante en el flujo de trabajo con imágenes digitales ya que sin esta los archivos ocuparían la mayor parte de nuestros sistemas de almacenamiento y su velocidad de transferencia se vería disminuida.

El objetivo de la compresión de imágenes es reducir los datos redundantes e irrelevantes de una imagen con la menor pérdida de calidad posible para permitir su almacenamiento o transmisión de forma eficiente.

En este artículo describiremos el proceso de compresión de imágenes dedicando la parte I del mismo a la codificación de Huffman y la parte II al estándar de compresión JPEG, y los pasos que éste lleva.

Empezaremos dando la definición de una imagen en escala de grises.

**Definición 1.** Una imagen en escala de grises se representa, en un ordenador, con una matriz $F$, de tamaño $M \times N$, en donde cada entrada corresponde a un píxel (que proviene de las primeras letras de **pic**ture **el**ement) y el valor de la entrada es un número entre 0 y 255, que corresponde a una intensidad de gris. Donde el valor 0 corresponde al color negro y el 255 al blanco.

Los 256 posibles valores de intensidad de grises se muestran a continuación.

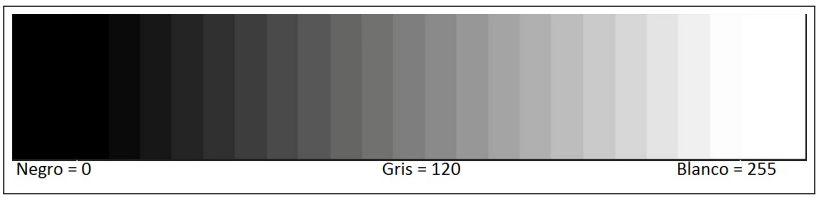


FIGURA 1. Rango de valores de intensidad de grises desde 0 (negro) hasta 255 (blanco).

Aún si uno puede ver la imagen completa, la intensidad individual de los píxeles es muy difícil de ver, o distinguir para el ojo humano, debido a la continuidad de la imagen. Por lo tanto hay información redundante, es decir que se repite o indetectable para la visión humana. Esto es una ventaja de las imágenes de alta resolución, más puntos, o píxeles, por pulgada (dpi - dots per inch) produce una imagen más fina.

**Ejemplo 2.** La Figura 2 muestra un ejemplo de una imagen en escala de grises de tamaño $768 \times 512$ píxeles. ✦


FIGURA 2. Imagen en escala de grises.

Con el fin de obtener una mejor idea de los valores de intensidad de píxel, en el siguiente ejemplo tomamos el bloque de tamaño $15 \times 15$ píxeles que representa la esquina inferior izquierda de la Figura 2. La imagen a continuación muestra el bloque en cuestión, así como sus valores de intensidad en su matriz correspondiente.

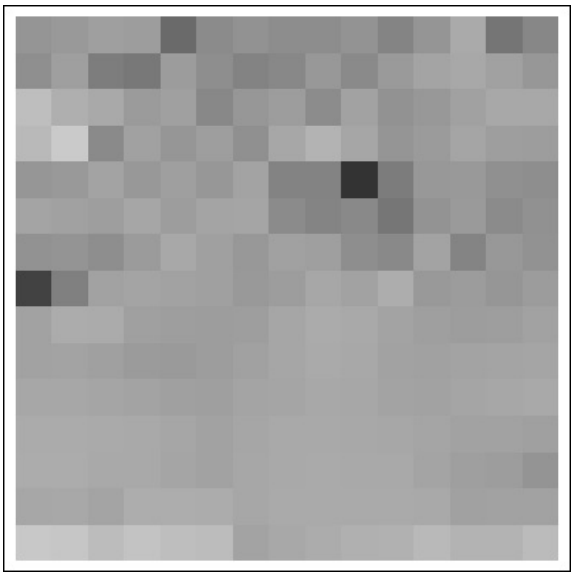

Figura 3. Bloque 15 x 15.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 150 | 154 | 160 | 157 | 106 | 140 | 147 | 142 | 141 | 147 | 132 | 150 | 171 | 117 | 136 |
| 144 | 159 | 125 | 121 | 157 | 143 | 132 | 136 | 153 | 138 | 155 | 164 | 169 | 162 | 152 |
| 190 | 175 | 169 | 155 | 161 | 136 | 152 | 158 | 141 | 162 | 147 | 153 | 161 | 168 | 169 |
| 185 | 203 | 139 | 161 | 151 | 159 | 145 | 167 | 179 | 167 | 150 | 155 | 165 | 159 | 158 |
| 151 | 153 | 163 | 152 | 160 | 152 | 164 | 131 | 131 | 51 | 124 | 152 | 154 | 145 | 143 |
| 164 | 162 | 158 | 167 | 157 | 164 | 166 | 139 | 132 | 138 | 119 | 148 | 154 | 139 | 146 |
| 147 | 148 | 143 | 155 | 169 | 160 | 152 | 161 | 159 | 143 | 138 | 163 | 132 | 152 | 146 |
| 66 | 129 | 163 | 165 | 163 | 161 | 154 | 157 | 167 | 162 | 174 | 153 | 156 | 151 | 156 |
| 162 | 173 | 172 | 161 | 158 | 158 | 159 | 167 | 171 | 169 | 164 | 159 | 158 | 159 | 162 |
| 163 | 164 | 161 | 155 | 155 | 158 | 161 | 167 | 171 | 168 | 162 | 162 | 163 | 164 | 166 |
| 167 | 167 | 165 | 163 | 160 | 160 | 164 | 166 | 169 | 168 | 164 | 163 | 165 | 167 | 170 |
| 172 | 171 | 170 | 170 | 166 | 163 | 166 | 170 | 169 | 168 | 167 | 165 | 163 | 163 | 160 |
| 173 | 172 | 170 | 169 | 166 | 163 | 167 | 169 | 170 | 170 | 170 | 165 | 160 | 157 | 148 |
| 167 | 168 | 165 | 173 | 173 | 172 | 167 | 170 | 170 | 171 | 171 | 169 | 162 | 163 | 162 |
| 200 | 198 | 189 | 196 | 191 | 188 | 163 | 168 | 172 | 177 | 177 | 186 | 180 | 180 | 188 |

Cuadro 1. Valores de intensidad de los píxeles de la Figura 3.

¿Cuál es el espacio necesario de almacenamiento de una imagen en una computadora?

**Definición 3.** La unidad fundamental de una computadora es un bit (**b**inary un**it**). Un bit (unidad binaria) puede valer 0 o 1. Un byte (la unidad fundamental de almacenamiento en una PC) está compuesta por 8 bits. Como cada bit vale 0 o 1 y 8 bits son un byte, hay $2^8 = 256$ bytes diferentes.

Cada byte tiene su propia representación en base 2 (sistema binario). Por ejemplo, la intensidad correspondiente al valor 118 se escribe en sistema binario de la siguiente forma:

$$118 = \mathbf{0}\cdot 2^7 + \mathbf{1}\cdot 2^6 + \mathbf{1}\cdot 2^5 + \mathbf{1}\cdot 2^4 + \mathbf{0}\cdot 2^3 + \mathbf{1}\cdot 2^2 + \mathbf{1}\cdot 2^1 + \mathbf{0}\cdot 2^0 = (01110110)_2$$

El Código Estándar Americano para el Intercambio de Información, American Standard Code for Information Interchange (ASCII), asigna a cada uno de los 256 bytes posibles un carácter del teclado. Ver Figura 4.

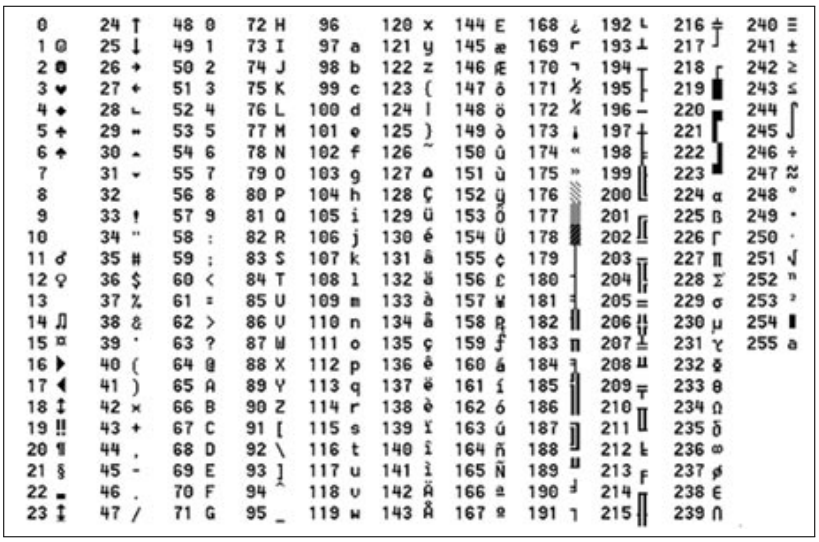

FIGURA 4. Código ASCII

**Observación 4.** Este código nació a partir de reordenar y expandir el conjunto de símbolos y caracteres ya utilizados en aquel momento en telegrafía por la compañía Bell. En un primer momento solo incluía letras mayúsculas y números, pero en 1967 se agregaron las letras minúsculas y algunos caracteres de control, formando así lo que se conoce como US-ASCII, es decir los caracteres del 0 al 127. Así con este conjunto de solo 128 caracteres fue publicado en 1967 como estándar, conteniendo todo lo necesario para escribir en idioma Inglés.

En 1981, la empresa IBM desarrolló una extensión de 8 bits del código ASCII, llamada "página de código 437", en esta versión se reemplazaron algunos caracteres de control obsoletos, por caracteres gráficos. Además se incorporaron 128 caracteres nuevos, con símbolos, signos, gráficos adicionales y letras latinas, necesarias para la escritura de textos en otros idiomas, como por ejemplo el Español. Así fue como se sumaron los caracteres ASCII que van del 128 al 255. Dicha extensión varia según el idioma en algunos simbolos, aunque son muy pocos. ✧

Cuando guardamos una imagen digital como un archivo en una cámara o un servidor web, ésta se guarda esencialmente como una larga cadena de bits (ceros y unos), donde las filas de la imagen están concatenadas formando una larga cadena de bits. Cada píxel de la imagen se compone de un byte y cada byte se construye a partir de 8 bits. Si la imagen tiene dimensiones $M \times N$ píxeles, entonces necesitamos $MN$ bytes o $8MN$ bits para almacenar la imagen. Hay 256 bytes diferentes y cada uno tiene su propia representación en base 2 (bits). Cada una de estas representaciones tiene 8 bits de longitud. Así que para un navegador web o cámara para visualizar una imagen almacenada, debe tener la cadena de bits además del diccionario que describe la representación de bits de cada byte.

**Ejemplo 5.** La imagen de la Figura 2 es de tamaño $768 \times 512$ por lo tanto al almacenar dicha imagen en un ordenador, el archivo que contiene la cadena de bits será de una longitud de ¡$768 \times 512 \times 8 = 3.145.728$ bits! ✦

Gracias a la continuidad de las imágenes y al hecho anteriormente mencionado con respecto a que la intensidad individual de los píxeles es muy difícil de ver, o distinguir para el ojo humano. Hay información sobreabundante, que puede comprimirse o eliminarse, reduciendo así de manera considerable la cantidad de espacio necesario de almacenamiento.

En otras palabras, la compresión de imágenes trata de minimizar el número de bits necesarios para representar una imagen.

La idea de la compresión es muy sencilla - simplemente cambiamos la representación de bits de cada byte con la esperanza de que el nuevo diccionario produce una cadena más corta de bits necesarios para almacenar la imagen. A diferencia de la representación ASCII de cada byte, nuestro nuevo diccionario podría utilizar representaciones de longitud de bits variables. De hecho, en 1952, David Huffman [2] hizo la siguiente observación:

> *En lugar de utilizar el mismo número de bits para representar cada carácter,*
> *¿Por qué no utilizar una cadena de bits corta para los caracteres*
> *que aparecen con mayor frecuencia en una imagen y una cadena de bits más larga*
> *para los caracteres que aparecen con menor frecuencia en la imagen?*

## §2. Código de Huffman

Con este estudio, Huffman superó a su profesor, quien había trabajado con el inventor de la teoría de la información Claude Shannon con el fin de desarrollar un código similar. Huffman solucionó la mayor parte de los errores en el algoritmo de codificación Shannon-Fano. La solución se basaba en el proceso de construir el árbol de abajo hacia arriba en vez de arriba hacia abajo.

La idea de Huffman fue identificar los píxeles que aparecen con mayor frecuencia en una imagen y asignarles representaciones cortas. Los píxeles que ocurren con menor frecuencia en una imagen se les asigna representaciones largas. Podemos entender mejor esta idea si nos fijamos en un ejemplo de codificación de Huffman:

**Ejemplo 6.** Supongamos que queremos almacenar la palabra **abracadabra** en el disco. Recordemos que a cada letra de la palabra **abracadabra** se le asigna un número en el código ASCII por lo que la palabra **abracadabra** puede ser la primera fila de una imagen digital. De hecho, la tabla a continuación da la representación en código ASCII y en bits para cada una de las letras de la palabra **abracadabra**.

| Letra | ASCII | Binario | Frecuencia | Frecuencia relativa |
|---|---|---|---|---|
| a | 97 | 01100001 | 5 | $5/11 \approx 0{,}45$ |
| b | 98 | 01100010 | 2 | $2/11 \approx 0{,}18$ |
| r | 114 | 01110010 | 2 | $2/11 \approx 0{,}18$ |
| c | 99 | 01100011 | 1 | $1/11 \approx 0{,}09$ |
| d | 100 | 01100100 | 1 | $1/11 \approx 0{,}09$ |

La palabra **abracadabra** se construye a partir de 11 caracteres, cada uno requiere 8 bits. Por lo tanto necesitamos $11 \times 8 = 88$ bits para almacenarla. Aquí está la palabra **abracadabra** expresada en bits:

| 01100001 | 01100010 | 01110010 | 01100001 | 01100011 | 01100001 | 01100100 | 01100001 | 01100010 | 01110010 | 01100001 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | r | a | c | a | d | a | b | r | a |

Para obtener el código de Huffman de cada carácter hay que construir un árbol binario de nodos, a partir de una lista de nodos, cuyo tamaño depende del número de caracteres, $n$. Los nodos contienen dos campos, el símbolo y el peso (frecuencia de aparición).

La técnica utilizada por el algoritmo de Huffman, consiste en la creación de un árbol binario en el que se etiquetan los nodos hoja con los caracteres, junto a sus frecuencias, y de forma consecutiva se van uniendo cada pareja de nodos que menos frecuencia sumen, pasando a crear un nuevo nodo intermedio etiquetado con dicha suma. Se procede a realizar esta acción hasta que no quedan nodos hoja por unir a ningún nodo superior, y se ha formado el árbol binario.

Cada nodo del árbol puede ser o bien un nodo hoja o bien un nodo interno. Inicialmente se considera que todos los nodos de la lista inicial son nodos hoja del árbol. Al ir construyendo el árbol, los nodos internos tendrán un peso y dos nodos hijos, y opcionalmente un enlace al nodo padre que puede servir para recorrer el árbol en ambas direcciones. Por convención el bit 0 se asocia a la rama izquierda y el bit 1 a la derecha. Una vez finalizado el árbol contendrá $n$ nodos hijos y $n - 1$ nodos internos.

Posteriormente de que se etiquetan las aristas que unen cada uno de los nodos con ceros y unos (hijo derecho e izquierdo, respectivamente, por ejemplo), el código resultante para cada carácter es la lectura, siguiendo la rama, desde la raíz hacia cada carácter (o viceversa) de cada una de las etiquetas de las aristas. ✦

A continuación describiremos, paso por paso, como el algoritmo de Hufmann anteriormente explicado reduce la cantidad de bits necesarios para almacenar la palabra **abracadabra**.

## §3. Algoritmo de Huffman

El proceso de construcción del árbol consiste en ordenar los caracteres únicos por frecuencia relativa, de menor a mayor y de izquierda a derecha. Ponemos cada uno de estos y sus frecuencias relativas en nodos conectados por ramas. Posteriormente se forma un nodo intermedio que agrupa a los dos nodos hoja que tienen menor peso (frecuencia de aparición). El nuevo nodo intermedio tendrá como nodos hijos a éstos dos nodos hoja y su campo peso será igual a la suma de los pesos de los nodos hijos. Los dos nodos hijos se eliminan de la lista de nodos, sustituyéndolos por el nuevo nodo intermedio. El proceso se repite hasta que sólo quede un nodo en la lista. Éste último nodo se convierte en el nodo raíz del árbol de Huffman.

**Ejemplo 7.** Codificación de Huffman para almacenar la palabra **abracadabra** paso por paso.

*Primer paso.* El primer paso consiste en ordenar los caracteres únicos por frecuencia relativa, de menor a mayor y de izquierda a derecha. Ponemos cada uno de estos y sus frecuencias relativas en nodos conectados por ramas como se ilustra a continuación, también conocido como árbol del código de Huffman:

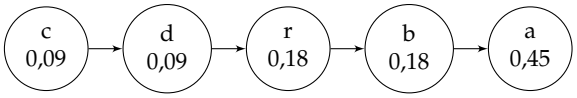


Figura 5. Las cinco letras, caracteres de la palabra **abracadabra**, ordenados por sus frecuencias relativas.

*Segundo paso.* Ahora tomamos los dos nodos más a la izquierda y sumamos sus frecuencias relativas para obtener el nodo suma $0{,}18$. Creamos un nuevo nodo con frecuencia relativa $0{,}18$ que tiene dos hijos, los nodos **d** y **c**. Completamos el arbol ubicando los nodos **r**, **b**, **a** a la derecha del nodo $0{,}18$. El proceso se ilustra en el siguiente gráfico:

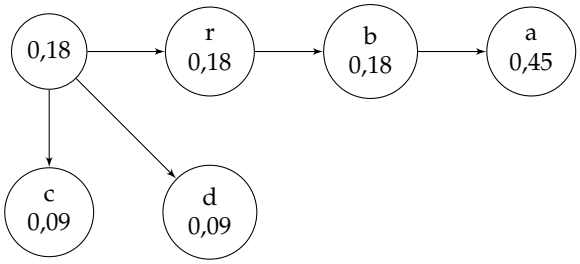


Figura 6. Paso 2 en el proceso de Huffman.

*Tercer paso.* El tercer paso es idéntico al segundo, sumamos las frecuencias relativas de los dos nodos que están más a la izquierda y creamos un nuevo nodo que tiene dos hijos. Ordenamos los nodos del nivel superior, según su frecuencia relativa, de menor a mayor y de izquierda a derecha. El proceso se ilustra en el siguiente gráfico:

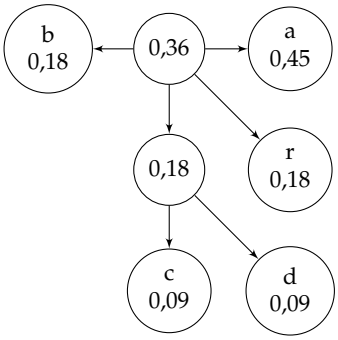


Figura 7. Paso 3 en el proceso de Huffman.

*Cuarto paso.* Seguimos el algoritmo repitiendo esencialmente los dos últimos pasos anteriormente detallados, hasta obtener un nodo raíz, de frecuencia relativa $1$.

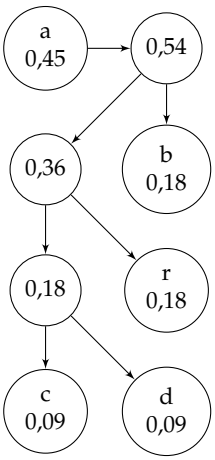


Figura 8. Paso 4 en el proceso de Huffman.

*Quinto paso.* El paso final en la codificación de Huffman es etiquetar todas las ramas que conectan los nodos, en el arbol del código de Huffman. Ramas con pendientes positivas están etiquetadas con 0 y las ramas con pendientes negativas están etiquetadas con 1. El resultado es un árbol de Huffman completo.

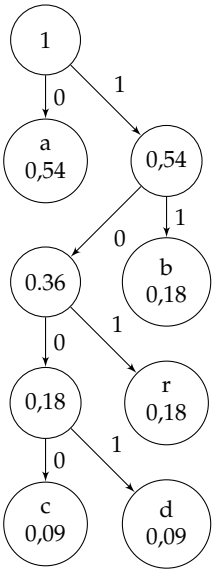


Figura 9. Paso 5 en el proceso de Huffman.

Ahora podemos leer la nueva representación de bits para cada carácter del árbol de Huffman de la palabra **a**bracadabra. Empezamos en el nodo superior y de ahí creamos la nueva representación mediante la recopilación de 0 y 1 a medida que avanzamos hasta el carácter deseado. Por ejemplo, en lugar de la representación en código ASCII del carácter **a**, 01100001, utilizamos simplemente el 0. La tabla a continuación muestra todas las nuevas representaciones de bits para las letras de la palabra **abracadabra**.

| Letra | binario | Huffman |
|---|---|---|
| a | 01100001 | 0 |
| b | 01100010 | 11 |
| r | 01110010 | 101 |
| c | 01100011 | 1000 |
| d | 01100100 | 1001 |

Con el código de Huffman podemos escribir la palabra **abracadabra** mediante la siguiente cadena de 0 y 1:

| 0 | 11 | 101 | 0 | 1000 | 0 | 1001 | 0 | 11 | 101 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | r | a | c | a | d | a | b | r | a |

La nueva representación de bits para la palabra **abracadabra** requiere sólo de 23 bits en lugar de los 88 bits originales. Esto representa un ahorro de ¡más del 75 %! Normalmente representamos la tasa de compresión en bits por píxel (bpp). Puesto que necesitamos 23 bits para representar la palabra **abracadabra** y hay 11 letras de la palabra, la compresión es del $23/11 = 2{,}09$ bpp. Esto significa que, en promedio, cada píxel requiere 2,09 bits de espacio de almacenamiento. ✦

El algoritmo de construcción del árbol de Huffman puede resumirse así:

- Crear un nodo hoja para cada símbolo, asociando un peso según su frecuencia de aparición e insertarlo en la lista ordenada ascendentemente.
- Mientras haya más de un nodo en la lista:
  - ⋆ Eliminar los dos nodos con menor probabilidad de la lista.
  - ⋆ Crear un nuevo nodo interno que enlace a los nodos anteriores, asignándole como peso la suma de los pesos de los nodos hijos.
  - ⋆ Insertar el nuevo nodo en la lista, (en el lugar que le corresponda según el peso).
- El nodo que quede es el nodo raíz del árbol.

*Invertir el procedimiento* Para invertir un flujo de bits construido a partir del código de Huffman y así poder determinar la entrada original el procedimiento se basa en la siguiente observación:

**Observación 8.** Cada nodo de un carácter en el árbol de Huffman completado es una rama terminal. Dado que no puede haber ramas posteriores de los nodos de cada caracter, tenemos la garantía de que no habrá redundancia en las nuevas representaciones de bits para cada carácter, es decir la codificación es única para cada carácter. Por lo tanto, si tenemos el diccionario y el flujo de bits codificados con el método de Huffman, podemos determinar los caracteres originales. ✧

**Ejemplo 9.** Como ejemplo, veamos como es la reconstrucción de la palabra **abracadabra**. Recordemos el diccionario de Huffman y el flujo de bits de la palabra **abracadabra**

| Letra | Huffman |
|---|---|
| a | 0 |
| b | 11 |
| r | 101 |
| c | 1000 |
| d | 1001 |

Cadena de bits: 01110101000010010111010

Empezamos con el primer carácter, en este caso el 0, según la tabla anterior el 0 representa a letra **a**. Seguimos con el 11 que representa a la letra **b**, pues el 1 no representa nada en el diccionario dado. Después viene el 101 que representa a la letra **r**, 0 a la letra **a**, 1000 a la letra **c**, 0 a la letra **a**, 1001 a la letra **d**, 0 a la letra **a**, 11 a la letra **b**, 101 a la letra **r**, 0 a la letra **a** por lo tanto la cadena de bits representa la palabra **abracadabra**. ✦

Veamos otro ejemplo.

**Ejemplo 10.** Consideremos el diccionario de Huffman y flujo de bits a continuación:

| Letra | Huffman |
|---|---|
| n | 01 |
| e | 10 |
| u | 11 |
| q | 00 |

Cadena de bits: 01101100111001

Empezamos con el primer carácter, en este caso el 0, según la tabla anterior el 0 no representa a ninguna letra. Tomamos el 01 que representa a la letra **n**. Después viene el 10 que representa a la letra **e**, 11 a la letra **u**, 00 a la letra **q**, 11 a la letra **u**, 10 a la letra **e**, 01 a la letra **n**, por lo tanto la cadena de bits representa la palabra **neuquén**. ✦

## §4. La codificación de Huffman y las imágenes digitales

Es sencillo considerar solamente palabras cortas como **abracadabra** o **Neuquén**. En cambio el rango de valores de bytes de una imagen digital sin duda puede ir de 0 a 255 y en este caso el dibujo del árbol es impracticable. Igualmente podemos calcular el código de Huffman para imágenes digitales y lo explicamos a continuación.

La imagen de la Figura 2 como vimos anteriormente es de tamaño $512 \times 768$ píxeles para un total de $512 \times 768 = 393{,}216$ píxeles o $393{,}216 \times 8 = 3,145,728$ bits. Resulta que todas las 256 intensidades en la escala de grises aparecen en la imagen. Si calculamos el código de Huffman para esta imagen nos encontramos con que podemos comprimir el

tamaño de almacenamiento y utilizar $3,033,236$ bits o aproximadamente $7{,}714$bpp. Lo que representa un pequeño ahorro en la cantidad de bits a guardar.

La imagen de la Figura 10, muestra la imagen resultante de aplicarle el procedimiento de compresión de imágenes a la imagen original 2.


Figura 10. Imagen resultante.

¿Qué salió mal? El hecho de que las 256 intensidades de los píxeles aparecieran en la imagen forzó el proceso de Huffman para formar el número máximo de niveles (sólo había cinco niveles en nuestro árbol de Huffman para el ejemplo **abracadabra**) y eso hizo nuevas representaciones de bits sólo modestamente más cortos que los originales.

El código de Huffman produce un ahorro cuando se aplica directamente a una imagen, que funcionará mucho mejor si la imagen se compone de un número menor de intensidades distintas. Lo que los investigadores han aprendido es que la transformación de una imagen para una configuración diferente, donde la salida resultante se compone de un número significativamente menor de intensidades o píxeles, está aumenta dramáticamente el potencial de compresión de la codificación de Huffman (u otros métodos de codificación).

Si bien la codificación de Huffman esbozada anteriormente es ingeniosa no se utiliza en los algoritmos de compresión, aunque es una herramienta útil para comprender la idea básica de la compresión de imágenes. El lector interesado puede consultar el libro [1].

## §5. Inconvenientes

La codificación de Huffman, llamada así por su inventor D. A. Huffman, alcanza la cantidad mínima de redundancia posible en un conjunto fijo de códigos de longitud variable. Esto no significa que la codificación de Huffman es un método de codificación óptimo. Significa que proporciona la mejor aproximación para los símbolos de codificación cuando se utiliza códigos de ancho fijo.

El problema con el código de codificación de Huffman o Shannon-Fano es que utilizan un número entero de bits. Si la cantidad media de un carácter dado es de 2,5 bits, el código

de Huffman para ese carácter debe ser 2 ó 3 bits, no 2,5. Debido a esto, la codificación de Huffman no puede considerarse un método de codificación óptima, aunque es la mejor aproximación que utilizan los códigos fijos con un número entero de bits.

Aunque la codificación de Huffman es ineficiente debido al uso de un número entero de bits por código, es relativamente fácil de implementar y muy económico, tanto para la codificación y decodificación. Huffman publicó su trabajo en 1952, y al instante se convirtió en el documento más citado en teoría de la información. La obra original de Huffman dio lugar a numerosas variaciones menores, y dominó el mundo de la codificación hasta la década de 1980.

A medida que el costo de los ciclos de la CPU se redujo, han surgido nuevas técnicas de codificación más eficientes. Uno en particular, la codificación aritmética, es un sucesor para la codificación de Huffman. La codificación aritmética es algo más complicada en concepto y aplicación que el estándar de códigos de longuitud variable. No produce un código único para cada símbolo. En lugar de ello, produce un código para un mensaje completo. Cada símbolo añadido al mensaje incrementalmente modifica el código de salida. Esto es una mejora porque el efecto neto de cada símbolo de entrada en el código de salida puede ser un número fraccionario de bits en lugar de un número entero. Así que si la entropía para el carácter 'e' es de 2,5 bits es posible añadir exactamente 2,5 bits para el código de salida.

En una segunda parte del trabajo abarcaremos el estándar de compresión de imágenes JPEG, que utiliza una versión mas sostificada del código de Huffman, y la matemática que interviene en el mismo.

## Referencias

[1] Gailly J., Nelson M. *The Data Compression Book*. Second edition. M & T Books. 1996.

[2] Huffman D. A. *A Method for the construction of minimum-redundancy codes*. Proceedings of the IRE. Volume: 40, Issue: 9 pag 1098 - 1101. IEEE. Septiembre 1952.

[3] Lezama J. *Image compression by Johnson graphs*. Proceeding of 2015 XVI Workshop on Information Processing and Control (RPIC). Ieee-explore. ISBN 978-1-4673-8466-7. Octubre de 2015.

[4] Levstein F., Lezama J., Maldonado C., Penazzi, D., Schilman M. *Hamming Graph based Image Compression with variable Threshold*, GVIP Journal, ISSN: 1687-398X, Volume 15, Issue 1, ICGST, Delaware, USA, June 2015.

[5] Image Compression. http://www.whydomath.org/node/wavlets/index.html. SIAM.

Javier Lezama
*Facultad de Matemática, Astronomía, Física y Computación* (Famaf), *Universidad Nacional de Córdoba* (UNC).
Av. Medina Allende s/n , Ciudad Universitaria (X5000HUA) Córdoba, Argentina.
(✉) javitolez@gmail.com

Recibido: *9 de enero de 2016*.
Aceptado: *15 de abril de 2016*.
Publicado en línea: *12 de junio de 2017*.

# Ahorrando bits

*Nota editorial por Ricardo Podestá*

La *Teoría de la Información* nace en 1948 con un trabajo seminal de Claude Shannon y trata sobre el estudio de la transmisión, procesamiento, utilización y almacenamiento de información. Con ella nacen la *Teoría de Códigos* y la *Compresión de Datos* y acoge en su seno a la *Criptografía*, disciplina que existía desde la antigüedad, aunque de forma más sencilla.

Supongamos que tenemos un conjunto de datos que queremos enviar (texto, imagen, sonido, etc.), lo que constituye nuestro *mensaje*, a través de un canal adecuado para el mismo. Queremos transmitir nuestro mensaje de forma segura y de la mejor manera posible. Pero ¿qué significan *segura* y *mejor* aquí? Esto depende claramente de cuál sea nuestro propósito, a saber:

- *Que sea secreto*. Si lo que queremos es transmitirlo de forma secreta entre partes amigas (en el sentido que aunque seamos espiados o el mensaje sea interceptado, éste permanezca inescrutable a los curiosos), estamos en presencia de la *criptografía*. Dejando de lado el obvio uso militar y en inteligencia/espionaje, situaciones típicas de la vida cotidiana en que se usa la criptografía son cuando se hacen compras o pagos por intenet, claves de cuentas bancarias o cuentas de internet, aplicaciones de teléfonos móviles, transmisiones de TV codificadas, etc, etc.
- *Que sea fiable*. Si lo que queremos en cambio es transmitir el mensaje de forma fiable y segura (en el sentido que si al enviar el mensaje se cometen errores, éstos sean detectados y corregidos automáticamente), sin preocuparnos por la secrecía, estamos hablando de la *teoría de códigos autocorrectores*. Ejemplos de esto son transmisiones de fotos tomadas por sondas espaciales, lectura de discos compactos como CD, DVD y Blu-ray, almacenamiento de datos en discos rígidos de computadoras, códigos de barras de productos como el EAN-13, los códigos editoriales ISBN e ISSN, transmisiones de TV por cable o fibra óptica, etc.
- *Que sea eficiente*. Si deseamos transmitir el mensaje de forma *rápida y poco costosa* (en el sentido de usar la menor cantidad de datos para mantener la información), pero manteniendo la esencia del mensaje original, estamos en presencia de la *compresión de datos*. Casos paradigmáticos de archivos comprimidos son los de tipo ZIP (datos), los JPG y JPGE (imágenes) y los MP3, MP4 y MP5 (música, fotos y videos).

Para enviar el mensaje por el canal es preciso transformar la información original para que se adapte al canal de transmisión, para luego de recibido volver a transformar éste a su forma original. Este proceso de hacer y deshacer se llama respectivamente *encriptar/desencriptar*, *codificar/decodificar*, *comprimir/descomprimir*. Lo más común es usar canales binarios, como en las computadoras, donde todas las palabras del mensaje son cadenas formadas por 0's y 1's. A estos dígitos binarios se los conoce como *bits* (por *bi*nary digi*ts*).

Mostraremos a continuación, con ejemplos concretos, la diferencia práctica entre codificar un mensaje (agregando bits) o comprimirlo (quitando bits), en ambos casos sin modificar la información original. Es decir, comparamos codificación vs compresión de datos (ítems 2 y 3 anteriores) dejando de lado la encriptación.

## *Agregando bits (cuando la redundancia no está de más)*

Imaginemos que queremos codificar las palabras 'sur', 'este', 'norte' y 'oeste'. Podemos elegir simplemente las letras S, E, N, O. Si nuestro canal es binario debemos hacerlo sólo con 0's y 1's. Está claro que necesitamos sólo 2 bits, ya que por ejemplo podemos tomar la siguiente codificanción

$$\mathrm{S} \to 10, \quad \mathrm{E} \to 00, \quad \mathrm{N} \to 01, \quad \mathrm{O} \to 11$$

correspondiente a numerar en forma antihoraria comenzando desde el sur.

Supongamos ahora que al enviar el mensaje $00$ se recibe $01$. ¿Es posible darse cuenta de este error? ¡No! Y el problema es que nuestro código

$$C_1 = \{00, 01, 10, 11\}$$

no detecta errores. Es decir, si hay un error en la transmisión, la palabra recibida sigue siendo una palabra del código. Por ejemplo, si cometemos un error al enviar $00$ recibimos $01$ ó $10$, y como ambas son palabras de $C_1$, no detectamos ningún error. Análogamente para $01$, $10$ y $11$.

Nos preguntamos ¿cómo podemos remediar esto? Lo usual es agregar *redundancia*. La forma más fácil de hacerlo es mediante un dígito de control de paridad. O sea, agregamos un dígito extra a cada palabra del código de modo que la suma de los dígitos de la nueva palabra sea par. En nuestro caso, el nuevo código obtenido así es

$$C_2 = \{000, 011, 101, 110\}.$$

Si ahora transmitimos $000$ y un error es cometido en la transmisión, entonces recibimos $100$, $010$ ó $001$. Como ninguna de estas palabras pertenece al código, detectamos un error. Lo mismo sucede con las otras palabras de $C_2$. Es decir, si se comete un único error al enviar *cualquier* palabra del código, la palabra recibida no pertenecerá al código. Luego, el código $C_2$ *detecta* cualquier error (que asumimos de un sólo bit). Notar, sin embargo, que $C_2$ no detecta 2 errores. Es decir, si se cometen 2 errores, la palabra recibida estará en el código cualquiera sea la palabra del código enviada.

Lamentablemente, el código $C_2$ no corrige errores. Esto es, una vez detectado el error, no podemos decidir cuál fue la palabra del código enviada. En nuestro ejemplo, supongamos que enviamos $000$ y recibimos $010$. Si bien detectamos el error, el código no puede corregirlo por sí mismo. En efecto, suponiendo que hay un sólo error, la palabra $010$ puede ser decodificada como $110$, como $000$ ó como $011$, todas palabras del código.

Pero, ¿será posible modificar el código original para que detecte y *corrija* errores? Afortunadamente, la respuesta es ¡sí! Una solución fácil es agregar mayor redundancia, a costa de tener que transmitir más. Formamos el código

$$C_3 = \{000000, 000111, 111000, 111111\}$$

repitiendo 3 veces cada bit del código original $C_1$. Ahora, si enviamos una palabra cualquiera y se comete un único error, cualquier que nos llegue puede ser corregida. Por ejemplo, si al enviar $000000$ recibimos $000100$, no sólo podemos detectar el error, sino que también podemos corregirlo. Intuitivamente, $000100$ está "más cerca" de $000000$ que de $000111$, $111000$ ó $111111$, ya que difiere de la palabra original en un sólo bit (y no en 3 o 6 como las otras). Luego, corregimos $000100$ como $000000$ y no como $000111$ ya que es más probable cometer un error que cometer tres. Este nuevo código $C_3$ detecta hasta dos errores y corrige uno (¿por qué?). Luego, hemos mejorado las propiedades detectoras y correctoras del código original $C_1$ y de $C_2$.

La moraleja es que, para que nuestro código sea capaz de detectar y corregir errores automáticamente, es necesario y fundamental agregar cierta *redundancia* a la información original. En la práctica, se agrega mucha más redundancia (de formas matemáticamente mucho mejores) para que se puedan detectar y corregir muchos más errores.

## *Ahorrando bits (cuando menos es más)*

Ahora estamos en la situación inversa a la anterior. Supongamos que nuestros mensajes son las letras del alfabeto romano A, B, C,..., Z, que son 27 en el caso del idioma español (por la Ñ). Si queremos usar 0's y 1's necesitaremos 5 bits, ya que $2^4 < 27 < 2^5$. Por ejemplo, podríamos tomar:

| | | |
|---|---|---|
| A=00001 | B=00010 | C=00011 |
| D=00100 | E=00101 | F=00110 |
| G=00111 | H=01000 | I=01001 |
| J=01010 | K=01011 | L=01100 |
| M=01101 | N=01110 | Ñ=01111 |
| O=10000 | P=10001 | Q=10010 |
| R=10011 | S=10100 | T=10101 |
| U=10110 | V=10111 | W=11000 |
| X=11001 | Y=11010 | Z=11011 |

La palabra 'hola' se codificaría entonces así

$$\text{HOLA} = 01000100000110000001.$$

¿Cómo podemos usar menos símbolos? Pues de una forma muy simple e ingeniosa. Codificamos las letras con distintas longitudes, donde a las letras más usadas le asignamos las cadenas de bits más cortas y a las letras escasamente usadas las cadenas más largas. En español, la frecuencia de aparición de las letras, ordenadas de mayor a menor, es:

E, A, O, S, R, N, I, D, L, C, T, U, M, P, B, G, V, Y, Q, H, F, Z, J, Ñ, X, W, K.

Por lo que tomaríamos E=1, A=10, O=11, S=100, R=101, N=110, etc. Ahora, tendríamos

$$\text{HOLA} = 1010011100110.$$

donde H$=10100$ y L$=1000$. Notar que hemos pasado de 20 bits a 13 bits, lo que representa una ganancia del 35 %.

Este es el principio que rige al famoso *código Morse* ideado por el pintor e inventor norteamericano Samuel F. B. Morse (1791-1872) para su uso en telegrafía con un solo cable. Dicho código consta de *puntos* y *rayas* para codificar las letras del alfabeto, los dígitos, algunos símbolos tipográficos (comas, puntos, etc) y espacios de distinta longitud para separar letras y palabras. En su versión internacional, las letras se codifican con longitudes que varian de 1 a 5 'bits' (la única letra con 5 símbolos es la Ñ pues es un añadido a la versión original) de la siguiente manera:

| | | | | | |
|---|---|---|---|---|---|
| A | = ·− | J | = ·−−− | R | = ·−· |
| B | = −−−· | K | = −·− | S | = ··· |
| C | = −·−· | L | = ·−·· | T | = − |
| D | = −· | M | = −− | U | = ··− |
| E | = · | N | = −· | V | = ···− |
| F | = ··−· | Ñ | = −−·−− | W | = ·−− |
| G | = −−· | O | = −−− | X | = −··− |
| H | = ···· | P | = ·−−· | Y | = −·−− |
| I | = ·· | Q | = −−·− | Z | = −−·· |

Notemos que ahora, al no tener todas las letras la misma longitud, el uso de los espacios es necesario, ya que podría haber ambigüedades en la decodificación. Por ejemplo, si mi amiga dice "I am his date" (yo soy su cita) se codifica así:

I AM HIS DATE = ·· ·− −− ···· ·· ··· −·· ·− − ·

Prescindiendo de los espacios tenemos

···−−−·····−···−−·

que sin embargo podría ser interpretado erroneamente como "sos help" (socorro, ayuda)

SOS HELP = ··· −−− ··· ···· · ·−·· ·−−·

Es muy divertido, aunque no tan fácil, buscar ejemplos similares en español y ¡estimulamos al lector curioso a hacerlo!

Está claro que en una transmisión larga, los símbolos usados para los espacios que separan letras y palabras representan una gran pérdida de tiempo y recursos en la transmisión. ¿Será posible solucionar este problema? Es decir, ¿es posible definir códigos donde las letras tengan distinta longitud y que, sin usar espacios entre letras, todo mensaje puede ser decodificado sin ninguna ambigüedad? Sorprendentemente la respuesta es que sí y fue dada por David Huffman en 1951, siendo aun estudiante universitario. Hoy se lo conoce como *codificación de Huffman* y se lo utiliza en compresión de datos. De esta manera, es posible quitar redundancia de modo de mantener la información original pero con un menor costo de transmisión.