El algoritmo de compresión de datos LZ78

Elvira Mayordomo

junio 2003

Contenido

- **Algoritmos de compresión sin pérdidas**
- **LZ78: idea principal**
- **LZ78 en detalle**
- **¿Cuánto comprime?**
- **Propuesta del primer trabajo de curso**

Hoy

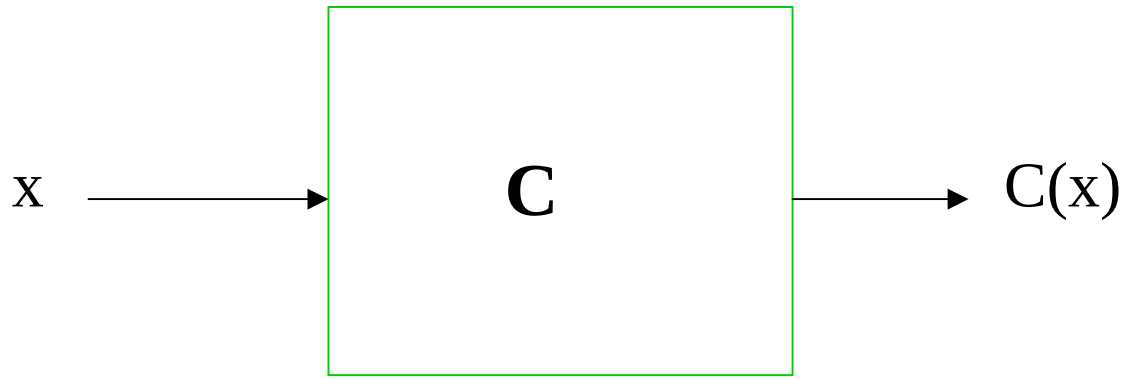
- **Vamos a empezar con un algoritmo concreto, viendo cómo funciona y porqué**
- **A partir de él veremos otros algoritmos de compresión que se derivan de él**

Hoy

- **Ziv, Lempel: “Compression of individual sequences via variable rate coding” IEEE Trans. Inf. Th., 24 (1978), 530-536**
- **Sheinwald: “On the Ziv-Lempel proof and related topics”. Procs. IEEE 82 (1994), 866-871**

Algoritmos de compresión

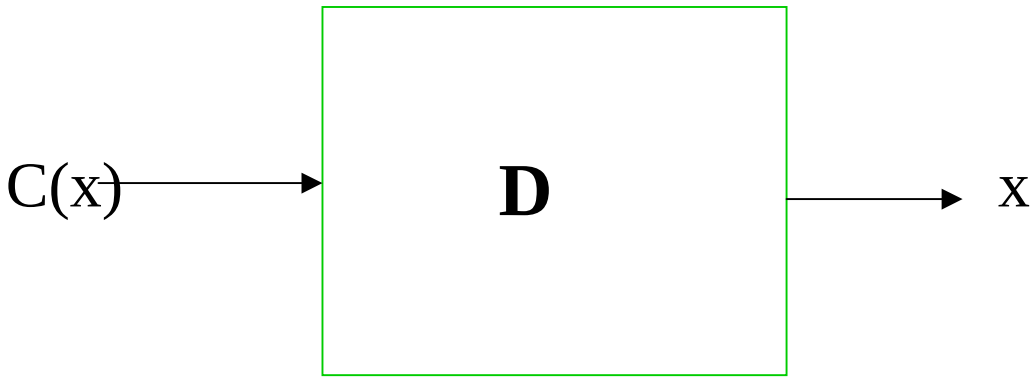
- **La entrada y la salida son strings (cadenas, secuencias finitas)**



compresor

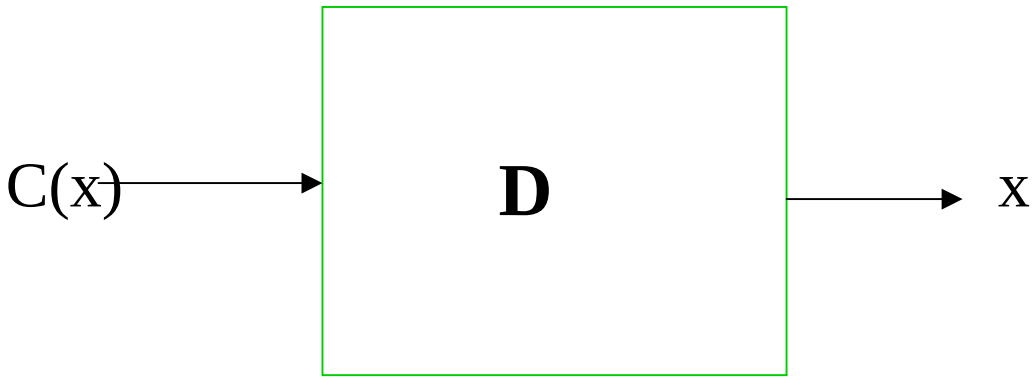
Algoritmos de compresión

- **A partir de la salida se puede reconstruir la entrada**



Algoritmos de compresión

- **A partir de la salida se puede reconstruir la entrada** **siempre: lossless compressor**



Ejemplo

- **Suponemos que queremos comprimir strings binarios**
- **El compresor sustituye 00000 por 0 y al resto de los bits les pone 1 delante**

01001000000011010000000000000010

Objetivo

- **Comprimir lo máximo posible todos los strings ???**

- **Eso es imposible, ¿por qué?**

Objetivo

- **Comprimir lo máximo posible todos los strings ???**
- **Eso es imposible, ¿por qué?**
- **Nos conformamos con comprimir “lo fácil de comprimir” (lo formalizamos más adelante)**

El LZ78

- **Es el algoritmo de compresión más utilizado (y estudiado)**
- **De él se derivan el compress de Unix, el formato GIF y los algoritmos LZW, LZMW**

LZ78: idea principal

- **Partimos el string en trozos (frases) de forma que cada trozo es uno de los anteriores más un símbolo**

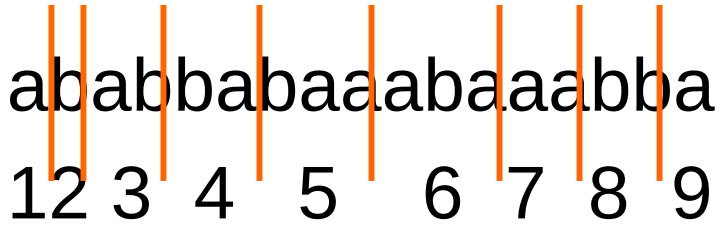
ababbabaaabaaabba

LZ78: idea principal

- **Partimos el string en trozos (frases) de forma que cada trozo es uno de los anteriores más un símbolo**

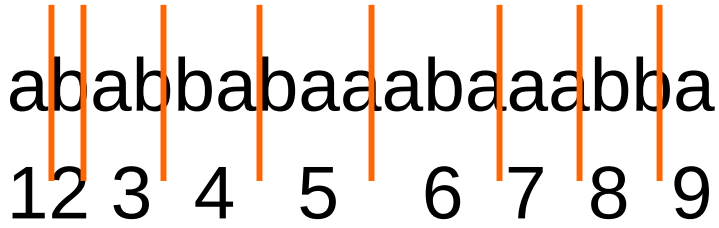
LZ78: idea principal

- **Numeramos las frases**



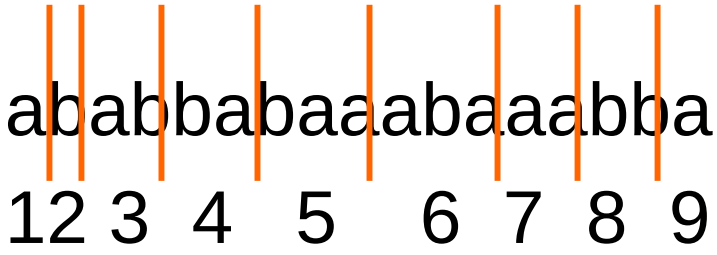
LZ78: idea principal

- **Numeramos las frases**



- **Cada frase es una de las anteriores más un símbolo**

LZ78: idea principal



(0,a) (0,b) (1,b) (2,a) (4,a) (3,a) (1,a) (2,b) (1,)

1 2 3 4 5 6 7 8 9

0 es la frase vacía

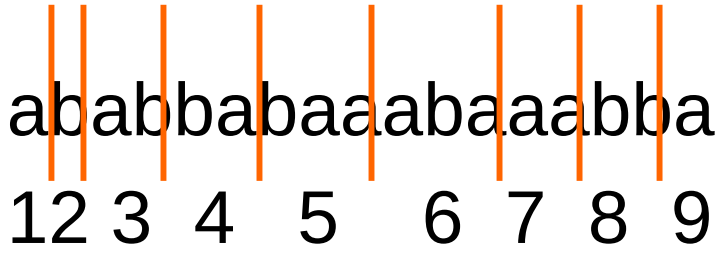
Más ejemplos

ababbbabaabab

LZ78: idea principal

- **Se trata de “parsear” la entrada en frases diferentes**
- **¿Cuál es la salida exactamente?**

LZ78



(0,a) (0,b) (1,b) (2,a) (4,a) (3,a) (1,a) (2,b) (1,)

1 2 3 4 5 6 7 8 9

- **Codificamos** (0,a)(0,b)(1,b)(2,a)...

LZ78: la salida

- **Codificamos la parte correspondiente a la frase número** n **que es**

  $$(i,s)$$

  utilizando $\lceil \log_2(n) \rceil$ **bits para** i

  y para el símbolo s**, depende de cuantos símbolos diferentes haya**

LZ78: la salida

(0,a)	(0,b)	(1,b)	(2,a)	(4,a)	(3,a)
0	01	011	100	1000	0110

LZ78: la salida

- **Si hay α símbolos, $\log_2(\alpha)$ bits**

Por ejemplo, para α=26, 5 bits

(0,f)	(0,h)	(1,r)
00101	001000	01.....

LZ78: resumen

- **Comprimir(x)**
  -- En cada momento hemos encontrado las
  -- frases w(1) ... w(n)
  Mientras quede por leer
  Buscar la frase más larga de entre
  w(1) ... w(n) desde el cursor
  Guardar el número de frase i y el siguiente
  símbolo s
  Añadir la nueva frase= w(i)s
  Fin

LZ78: resumen

- **Descomprimir(y)**

  -- En cada momento hemos descomprimido el trozo

  -- z= w(1) ... w(n) y conocemos w(1), ..., w(n)

  Mientras quede por leer de y

  Mirar los siguientes $\lceil \log_2(n+1) \rceil$ bits para sacar el número de frase i

  Mirar los siguientes $\log_2 \alpha$ bits para sacar el símbolo s

  Concatenar z:= z w(i) s

  Añadir la nueva frase w(i)s

  Fin

LZ78: resumen

- **Comprimir(x)**
  -- En cada momento hemos encontrado las
  -- frases w(1) ... w(n)
- **Descomprimir(y)**
  -- En cada momento hemos descomprimido el trozo
  -- z= w(1) ... w(n) y conocemos w(1), ..., w(n)

Al conjunto de frases en cada momento del proceso se le llama diccionario

Y esto es compresión con diccionario adaptivo

LZ78 ???

- **Ya sabemos cómo funciona pero ...**
  - **¿Cuánto comprime?**
  - **¿Es mejor que otros métodos?**

LZ78: ¿cuánto comprime?

- **El tamaño de C(x) depende sólo del número de frases en que dividimos x**

a ab aba abaa abaab 15 símbolos

b a ba bb aa 8 símbolos

LZ78: ¿cuánto comprime?

- **Si t(x) es el número de frases en que LZ78 divide a x:**

$$|C(x)| = \sum_1^{t(x)} \left(\lceil \log_2(n+1) \rceil + \log_2 \alpha \right)$$

$$\approx \quad t(x) \left(\log_2(t(x)) + \log_2 \alpha \right)$$

LZ78: ¿cuánto comprime?

- **Strings que LZ78 comprime mucho:**

|x|	|C(x)|
55	35
210	89
20100	1545

LZ78: ¿cuánto comprime?

- **Hay strings que LZ78 comprime mucho.**

  Ya hemos razonado que no pueden ser tantos
- **¿Es LZ78 mejor que explotar alguna regularidad sencilla?**

  Para entradas largas sí

De momento

- **Sabemos cómo funciona el LZ78**
- **Falta justificar cuánto comprime**
- **También veremos otras variantes (LZ77)**

De momento

- **No hemos hablado de cómo lo implementamos**
  - **→ ¿Cómo guardamos el diccionario?**
  - **→ ¿Cuánto ocupa?**
- **Al fijar esos detalles de implementación cambiamos el algoritmo en sí**

Propuesto

1) **Buscar todas las variantes conocidas de LZ78**

   (al menos “compress” de Unix, GIF, LZW, LZC, LZT, LZMW, LZJ)

2) **Explicar en detalle cómo funcionan y qué prestaciones tienen**

Los compresores de estados finitos

Elvira Mayordomo

junio 2003

Contenido

- **¿Qué son los compresores de estados finitos?**
- **Ejemplos**
- **¿Por qué LZ78 es mejor que cualquiera de ellos?**
- **¿Por qué LZ78 es “lo mejor posible”?**

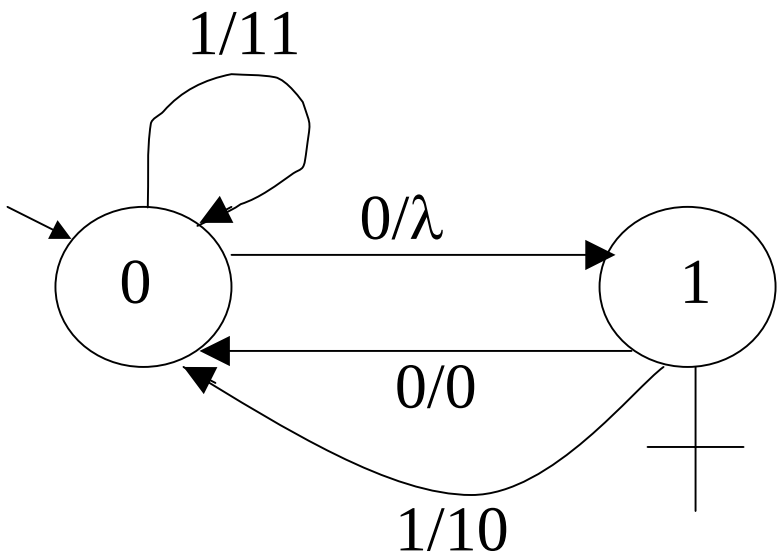
Hoy

- **Vamos a ver los mecanismos de compresión que se usaban antes del Lempel-Ziv**
- **Se trata de compresores sencillos que explotan regularidades de las entradas**

Seguimos con ...

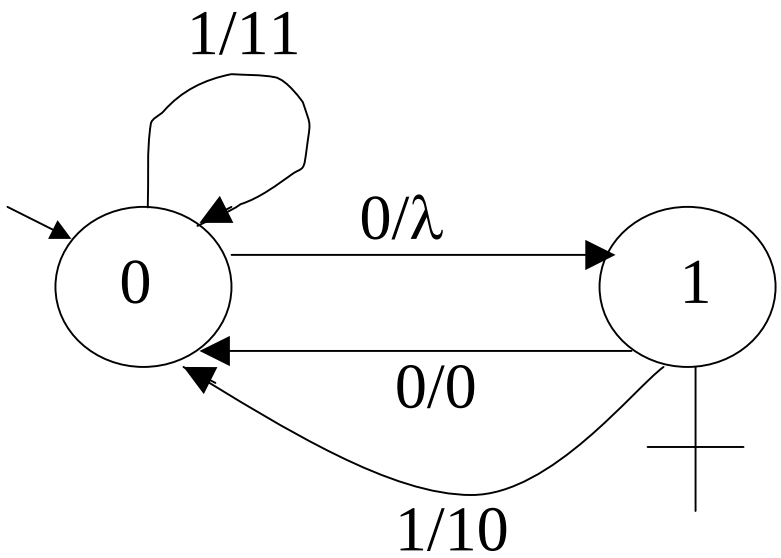
- **Ziv, Lempel: “Compression of individual sequences via variable rate coding” IEEE Trans. Inf. Th., 24 (1978), 530-536**
- **Sheinwald: “On the Ziv-Lempel proof and related topics”. Procs. IEEE 82 (1994), 866-871**

Ejemplo



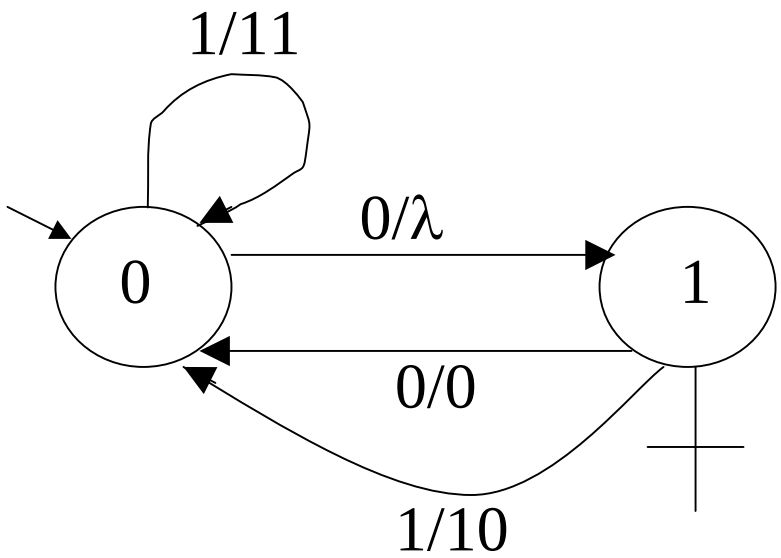
b/w **quiere decir que si leo el bit b la salida es w**

Ejemplo



A partir de la salida y del estado al que llego puedo recuperar la entrada

Ejemplo



Con este los bloques de ceros se comprimen ...

Definición

- **Un ILFSC (information-lossless finite state compressor) es un “autómata con salida”**

$$A=(Q, \delta, q_0, c_A)$$

...

Definición

- **Un ILFSC es**

$$A=(Q, \delta, q_0, c_A)$$

Q es el conjunto de estados,

q_0 es el estado inicial

Definición

- **Un ILFSC es**

$$A=(Q, \delta, q_0, c_A)$$

δ es la función de transición

δ(q,b) me dice a qué estado llego si estoy en el estado q y leo b

Definición

- **Un ILFSC es**

$$A=(Q, \delta, q_0, c_A)$$

c_A es la función de salida

$c_A(q,b)$ me dice la salida si estoy en el estado q y leo b

Definición de ILFSC

- **Importante: A partir de la salida**

$$c_A(x) = c_A(q_0,x)$$

y del estado al que llego

$$\delta(x) = \delta(q_0,x)$$

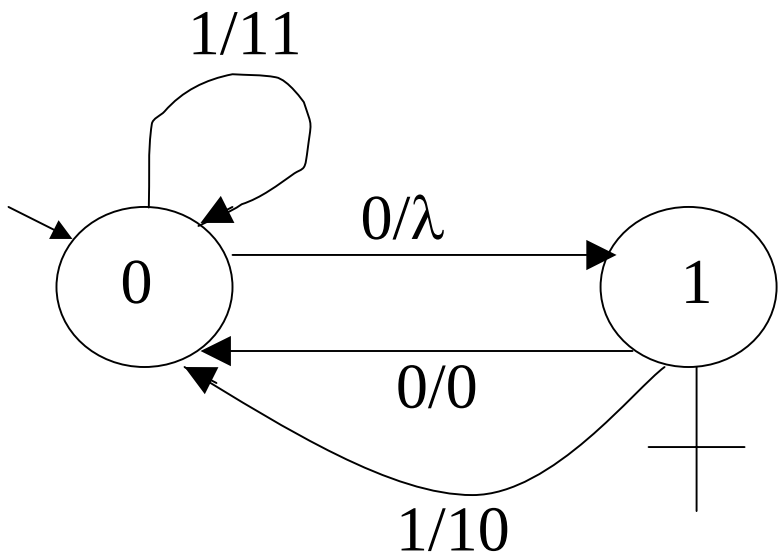
tengo que poder reconstruir x

Definición de ILFSC

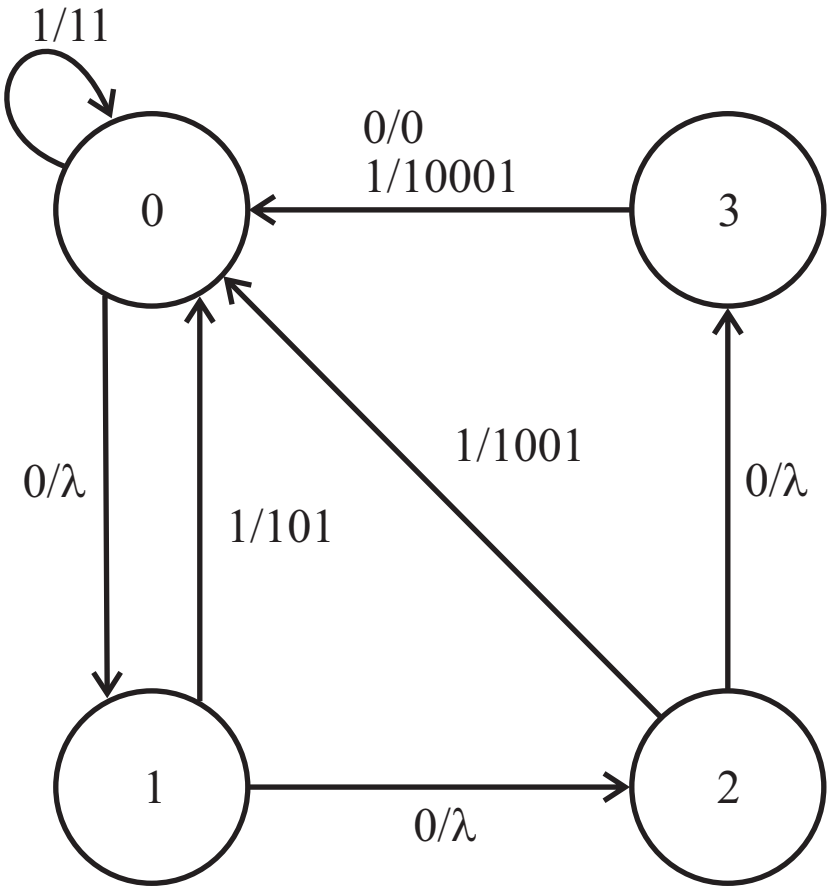
- **Lo que exijo es que para cualquier estado q y para dos strings distintos x, x'**

$$(c_A(q,x), \delta(q,x)) \neq (c_A(q,x'), \delta(q,x'))$$

Ejemplo



Ejemplo



ILFSC

- **Cada ILFSC me da un algoritmo de compresión con salida**

  $C(x) = (c_A(x), \delta(x))$

- **¿Cuánto comprimen?**

Compresión con ILFSC

- **Normalmente se mira**

$$\rho_A(x) = \frac{|c_A(x)|}{|x|}$$

|x| es la longitud de x

Compresión con ILFSC

- **Y para ver el comportamiento con entradas grandes se estudia qué pasa con secuencias infinitas z**

$$\rho_A(z) = \text{limsup}_n \ \rho_A(z[1..n])$$

Compresión con LZ78

$$\rho_{LZ}(z) = \limsup_n \frac{|LZ(z[1..n])|}{n}$$

LZ98: universal para ILFSC

- **¿Por qué?**

Lo que vamos a ver es que para cualquier ILFSC A, y para cualquier secuencia infinita z:

$$\rho_{LZ}(z) \leq \rho_{A}(z)$$

LZ98: universal para ILFSC

- **Para ello vamos a ver que si tenemos una partición cualquiera en frases distintas de z[1..n]= w(1) ... w(f(n)) entonces**

$$|c_A(z[1..n])| \geq f(n) \log_2(f(n)) - \varepsilon(n)$$

Razonamiento:

$|c_A(z[1..n])| \geq f(n) \log_2(f(n)) - \varepsilon(n)$

Como $z[1..n] = w(1) \ldots w(f(n))$ con frases distintas entonces también son distintos

$$(q_i, c_A(q_i, w(i)), q_{i+1})$$

$q_i = \delta(q_0, w(1) \ldots w(i-1))$

Razonamiento:

Si A tiene s estados entonces hay como máximo $s^2 2^k$ $(q_i, c_A(q_i,w(i)), q_{i+1})$ con $|c_A(q_i,w(i))| = k$

Lo más corto que puede ser $c_A(z[1..n])$ es cuando hay s^2 de longitud 1, $s^2 2^2$ de longitud 2, etc

Razonamiento:

... cuando hay s^2 de longitud 1, $s^2 2^2$ de longitud 2, etc

Si f(n) está entre $s^2 2^{L+1}$ y $s^2 2^{L+2}$ eso sería

$|c_A(z[1..n])| \geq$

$$\Sigma^L \, j \, s^2 2^j + (L+1)\,(f(n) - s^2 2^{L+1} + 1)$$

Razonamiento:

$|c_A(z[1..n])| \geq$

$\Sigma^L\ j\ s^2 2^j + (L+1)\ (f(n) - s^2 2^{L+1} + 1) =$

$s^2((L-1)2^{L+1}+2) + (L+1)\ (f(n) - s^2 2^{L+1} + 1)$

...

$|c_A(z[1..n])| \geq f(n)\ \log_2(f(n)) - \varepsilon(n)$

$$\rho_{LZ}(z) \leq \rho_A(z)$$

Tenemos, para cualquier partición de z[1..n] en f(n) frases:

$$|c_A(z[1..n])| \geq f(n) \log_2(f(n)) - \varepsilon(n)$$

$$\rho_{LZ}(z) \leq \rho_A(z)$$

Y para la partición de z[1..n] en t(n) frases según LZ78:

$$|LZ(z[1..n]))| \approx t(n)\,(\log_2(t(n)) + 1)$$

(además t(n)/n tiende a cero)

(he tomado α=2)

$$\rho_{LZ}(z) \leq \rho_A(z)$$

Luego

$$|c_A(z[1..n])| / n \geq$$
$$(t(n) \log_2(t(n)) - \varepsilon(n)) / n \approx$$
$$|LZ(z[1..n]))| / n - \varepsilon'(n)$$

ILFSC

- **Dado A ILFSC y una partición cualquiera en frases distintas de z[1..n]= w(1) ... w(f(n)) entonces**

$$|c_A(z[1..n])| \geq f(n) \log_2(f(n)) - \varepsilon(n)$$

LZ

Si z[1..n] se parte en t(n) frases según LZ78:

$$|LZ(z[1..n]))| \approx t(n) \left(\log_2(t(n)) + 1 \right)$$

Conclusión

LZ78 es mejor que cualquier compresor de estados finitos para longitudes suficientemente grandes

Preguntas

- **La catástrofe del bit de más**

  001001001100110 ...

  1001001001100110 ...

- **LZ78 comprime secuencias que los ILFSC no comprimen**

Comparación con la entropía

Si tenemos una secuencia infinita z podemos considerar la entropía experimental o a posteriori

Es decir ¿cómo de variada es z?

Definición formal de entropía

Frecuencia relativa de v en u

P(u,v)=

“número de veces de v en u” / (|u|-|v|+1)

Ej: u=01101101 v=1101 P=2/5

Definición formal de entropía

P(u,v)= "número de veces de v en u" / (|u|-|v|)

k-entropía $H_k(z[1.n])=$

$$-1/k \sum_v P((z[1.n],v) \log P(z[1.n],v)$$

La suma es sobre las v de longitud k

Definición formal de entropía

La k-entropía de z

$$H_k(z) = \text{limsup}_n\ H_k(z[1.n])$$

representa lo "variada" que es z a nivel de trozos de k bits.

La entropía a posteriori de z es

$$H(z) = \lim_k H_k(z)$$

LZ comprime

Para cualquier z

$$\rho_{LZ}(z) \leq H(z)$$

LZ comprime

Si se trata de entropía a priori (tenemos una distribución de probabilidad para las secuencias infinitas) entonces

con probabilidad 1 sobre z

$$\rho_{LZ}(z) \leq H(z)$$

El algoritmo de compresión de datos LZ77

Elvira Mayordomo

junio 2003

Contenido

- **LZ77: idea principal**
- **LZ77 en detalle**
- **LZ77 y LZ78**
- **Propuesta del segundo trabajo de curso**

Hoy

- **El LZ77 es anterior al LZ78**
- **Las ideas principales son similares pero no utiliza diccionario, con lo que las implementaciones concretas son muy diferentes**

Hoy

- **Ziv, Lempel: “A universal algorithm for sequential data compression” IEEE Trans. Inf. Th., 23 (1977), 337-343**
- **Shields: “Performance of the LZ algorithms on individual sequences” IEEE Trans. Inf. Th., 45 (1999), 1283-1288**

El LZ77

- **De él se derivan ZIP, GZIP, WINZIP, PKZIP, LZSS, LZB, LZH, ARJ, RFC**

LZ77: idea principal

- **En cada momento buscamos el trozo más largo que empieza en el cursor y que ya ha ocurrido antes, más un símbolo**

ababbabaaabaaabba

LZ77: idea principal

- **En cada momento buscamos el trozo más largo que empieza en el cursor y que ya ha ocurrido antes, más un símbolo**

Más ejemplos

a | b | abb | bab | aabab |

0 | 01 | 0102 | 10210212 | 021021200 |

a | ac | aacab | caba | aac |

LZ77: el output

- **Para cada trozo damos (p,l,c)**
  - **p es la posición de la anterior ocurrencia (hacia atrás)**
  - **l es la longitud de la ocurrencia**
  - **c es el siguiente carácter**

LZ77: el output

0 01 0102 10210212 021021200

(0,0,0) (1,1,1) (2,3,2) (3,7,2) (7,8,0)

LZ77: algoritmo

- **Comprimir(x)**

  -- Hemos comprimido x[1..n-1]

  Mientras quede por leer

  Buscar i<n y L lo mayor posible tal que

  x [n..n+L-1] = x [i..i+L-1]

  Output n-i, L, x[n+L]

  Fin

LZ77: algoritmo

- **Descomprimir(y)**

  -- En cada momento hemos descomprimido el trozo

  -- x[1..n-1]

  Mientras quede por leer de y

  Sacar el siguiente (i,L,s)

  Para k:=0 .. L-1

  x[n+k]:=x[n-i+k]

  -- No hay problema si n-i+k > n-1

  Fin

LZ77: resumen

- **No utilizamos diccionario**
- **El hecho de que mire desde el principio lo puede hacer excesivamente lento**

LZ77 y LZ78

- El LZ77 no tiene patentes prácticamente
- El LZ78 y sus variantes sí

- El LZ78 es más fácil de implementar ??
- El LZ77 comprime tanto como el LZ78 ??

De momento

- **No hemos hablado de cómo lo implementamos**

  → ¿Es razonable buscar desde el principio (sliding window)?

- **Al fijar esos detalles de implementación cambiamos el algoritmo en sí**

Propuesto

1) **Buscar todas las variantes conocidas de LZ77**

   (al menos ZIP, GZIP, WINZIP, PKZIP, LZSS, LZB, LZH, ARJ, RFC)

2) **Explicar en detalle cómo funcionan y qué prestaciones tienen**

Las técnicas de compresión de datos sin pérdida de información (lossless)

→ Un resumen personal

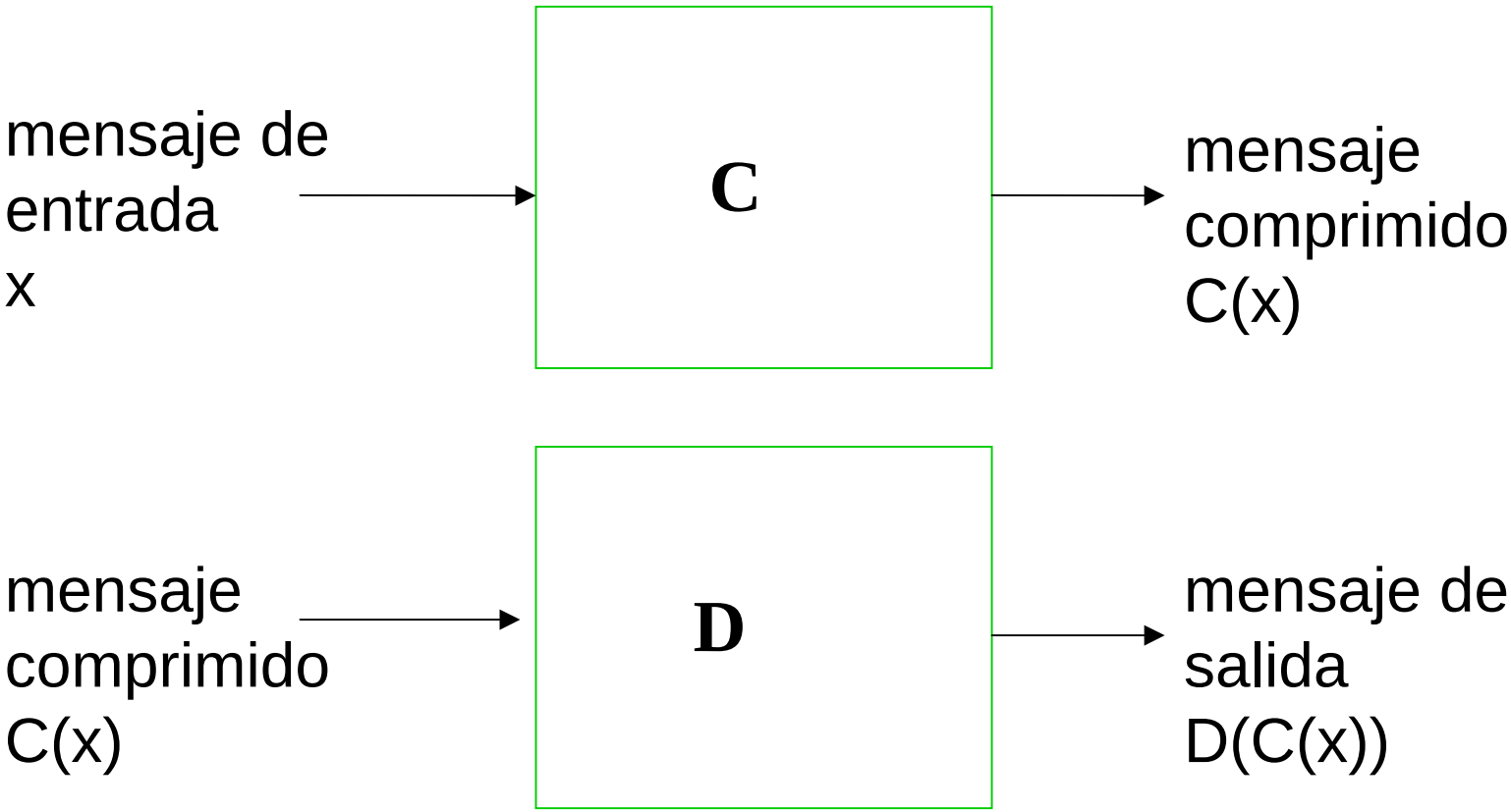
Elvira Mayordomo

junio 2003

Contenido

- Lossless versus lossy
- Codificación por probabilidades:
  - Huffman y código aritmético
- Aplicaciones de cod. por probabilidades
- Algoritmos Lempel-Ziv (vistos)
- Otros: Burrows-Wheeler

Lossless versus lossy



Lossless versus lossy

- **Lossless:**

  Mensaje de entrada = Mensaje de salida

  $$D(C(x)) = x$$

- **Lossy:**

  Mensaje de entrada ≈ Mensaje de salida

  $$D(C(x)) \approx x$$

Lossless versus lossy

- **Lossy no quiere decir necesariamente pérdida de calidad**
- **Depende de los datos**
- **Los algoritmos de propósito general son fundamentalmente lossless**

Hoy

- **Veremos técnicas concretas que se utilizan fundamentalmente combinadas**
- **Por ejemplo los mejores compresores de imágenes las utilizan prácticamente todas (más alguna lossy)**

Hoy

- **Guy E. Belloch: “Introduction to Data Compression”**

  http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/compression.pdf

Codificación por probabilidades

- **Supongamos que conocemos a priori la frecuencia con que aparece cada símbolo**
- **Utilizamos esta información para comprimir mucho los mensajes que aparecen más a menudo**

Codificación por probabilidades

- **¿Y si no conocemos a priori la frecuencia con que aparece cada símbolo?**
- **La vamos adivinando sobre la marcha**

- **También podemos utilizar frecuencias que dependen del contexto**

Codificación por probabilidades

- **Empezamos suponiendo que sí sabemos la frecuencia con que aparece cada símbolo**

Codificación por probabilidades

- **Ejemplo:**

  De cada 1000 caracteres en inglés:

A	B	C	D	E	F	G	H	I	J	K	L	M
73	9	30	44	130	28	16	35	74	2	3	35	25

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
78	74	27	3	77	63	93	27	13	16	5	19	1

Código Huffman

- **Usa directamente las frecuencias de cada símbolo para hacer una compresión símbolo a símbolo**
  - **Los caracteres más frecuentes tienen una codificación más corta.**
- **Lo hace por medio de códigos “libres de prefijo” (prefix codes)**

Código Huffman: códigos prefijos

- **Asignamos a cada símbolo s un código c(s)**

- **Para cada dos símbolos distintos s, s’**

  c(s) no es prefijo de c(s’)

Código Huffman: códigos prefijos

- **Ejemplo:**

  c(a)= 0 c(b)= 110 c(c)= 111 c(d)= 10

  01100 ??

  1110

  0101100

Código Huffman: códigos prefijos

- **Propiedad:**

  Todo código prefijo se puede decodificar de forma única

  Es decir, a partir de C(x) se puede recuperar x (para cualquier cadena x)

Código Huffman: frecuencias

- **Con un sencillo algoritmo se construye un prefix code que asigna codificación más corta a los símbolos más frecuentes**
- **Es un código óptimo dentro de los códigos prefijos**

Código Huffman: frecuencias

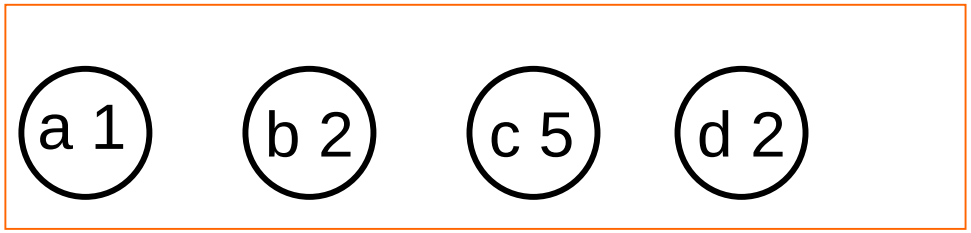
- **Con un sencillo algoritmo se construye un prefix code que asigna codificación más corta a los símbolos más frecuentes**

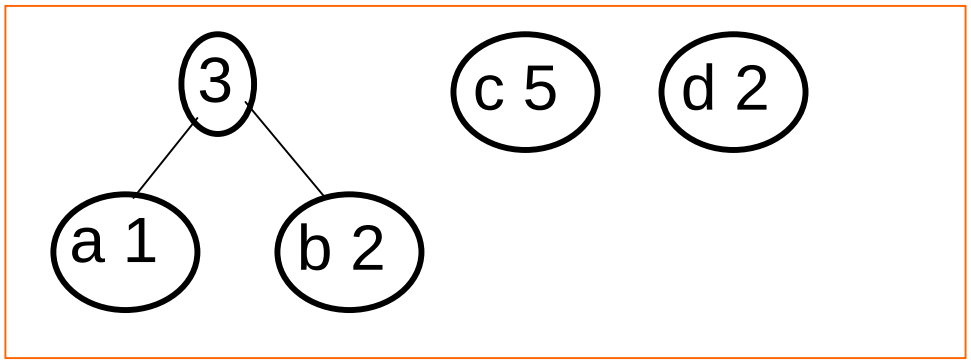
Código Huffman

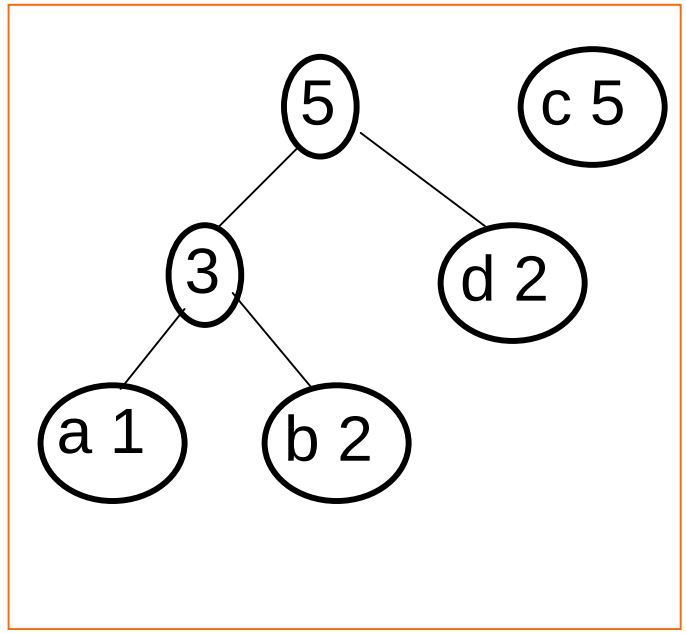
- **Empieza con un vértice por cada símbolo, con peso la frecuencia de ese símbolo**

- **Repetir hasta que haya un único árbol:**
  - **Seleccionar los dos árboles que tengan menores pesos en la raíz: p1 y p2**
  - **Unirlos con una raíz de peso p1+p2**

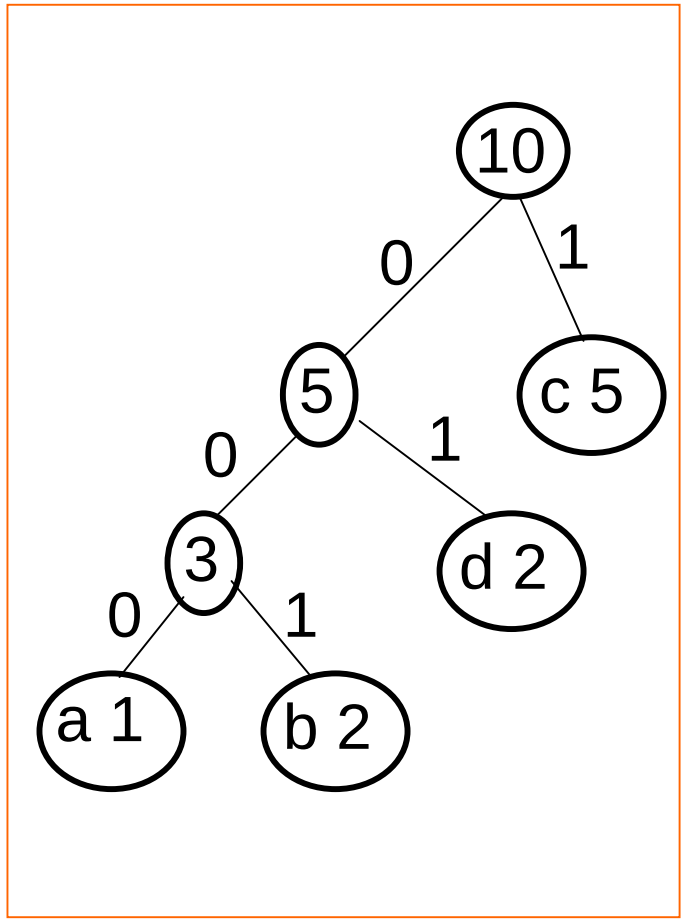
Código Huffman



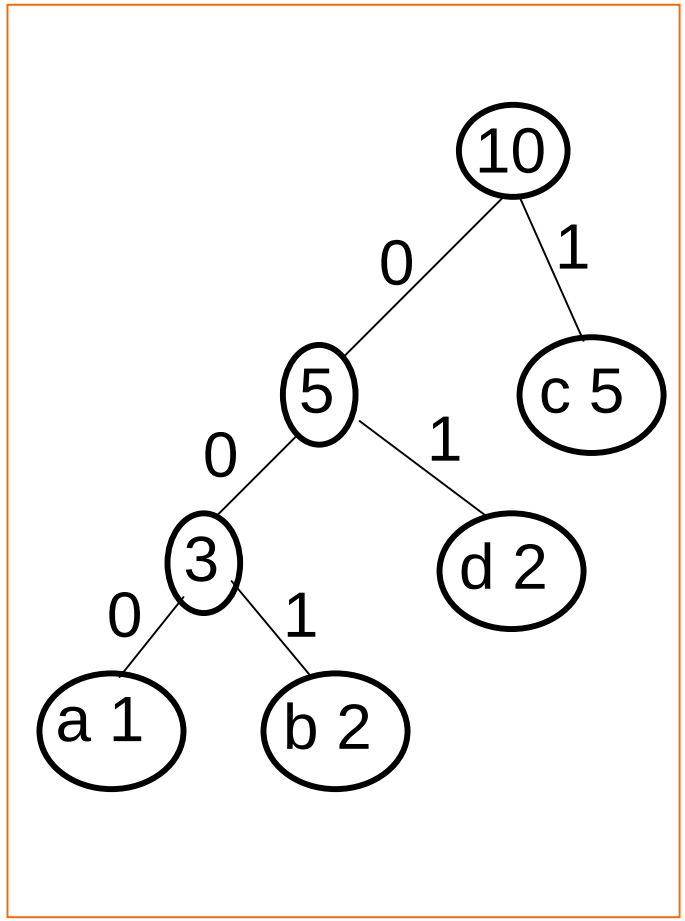




Código Huffman



Código Huffman



a = 000

b = 001

d = 01

c = 1

Código Huffman: frecuencias

- **Es un código óptimo dentro de los códigos prefijos**
- **Es rápido tanto en compresión como en descompresión**
- **Forma parte de la gran mayoría de los algoritmos de compresión**

  gzip, bzip, jpeg, para fax, ...

Código Huffman: frecuencias

- **Es un código óptimo dentro de los códigos prefijos**
- **Pero esto es si consideramos los códigos que actúan carácter a carácter**

Código Huffman: problemas

- **Utiliza al menos un bit por símbolo**
- **Aunque un mensaje de 1000 símbolos esté formado sólo por símbolos muy frecuentes tendremos que utilizar 1000 bits para comprimirlo**

Solución: código aritmético

- **Permite “mezclar” los símbolos del mensaje de entrada**
- **Se usa en jpeg, mpeg, PPM ...**
- **Más costoso que Huffman**

Código aritmético

- **Para cada símbolo s, conocemos**

  p(s) = frecuencia(s)/ número total

	A	B	C	D	E
	73	9	30	44	130
p(s)	.073	.009	.03	.044	.13

Código aritmético

- **A cada símbolo le asociamos un intervalo de números en [0,1]**

	A	B	C
p(s)	.2	.5	. 3
i(s)	[0,.2)	[.2, .7)	[.7,1)

Código aritmético

- **A cada dos símbolos s_1 s_2 les asociamos un intervalo de números en $i(s_1)$**

$$i(B)=[.2, .7)$$

	BA	BB	BC	
$p(s_2)$	.2	.5	.3	
$i(s_1\ s_2)$	[.2,.3)	[.3, .55)	[.55,.7)	...

Código aritmético

- **Y así sucesivamente**
- **Por ejemplo en el caso anterior:**

$$i(BAC)=[.27, .3)$$

Código aritmético: importante

- A mensajes distintos corresponden intervalos disjuntos
- Así que podemos identificar el mensaje a partir de cualquier número de su intervalo (y de la longitud del mensaje)

Código aritmético

- **Ejemplo:**

  si tenemos un mensaje de longitud 2 identificado por el número .6

Código aritmético

- **Tiene que ser BC**

	BA	BB	BC	
$p(s_2)$	.2	.5	.3	
$i(s_1\ s_2)$	[.2,.3)	[.3, .55)	[.55,.7)	...

Código aritmético

- **Falta ver cómo asociamos un número a cada intervalo ...**

Código aritmético

- **Falta ver cómo asociamos un número a cada intervalo ...**
- **Para ello elegimos un número en binario de forma que todos los números que empiezan como él están en el intervalo**

Código aritmético

- **Para ello elegimos un número en binario de forma que todos los números que empiezan como él están en el intervalo**

[0, .33)	**→**	**.01**
[.33, .66)	**→**	**.1**
[.66,1)	**→**	**.11**

Código aritmético: resumen

- **A cada cadena (mensaje de entrada) le asociamos un intervalo en [0,1]**
- **A cada intervalo le asociamos un número corto en binario (mensaje comprimido)**
- **A partir de la longitud de entrada y el número binario podemos recuperar el mensaje**

Código aritmético: resumen

- **El proceso puede ser lento y requiere aritmética de alta precisión (esto se puede arreglar)**
- **Pero el tamaño de los mensajes comprimidos es muy cercano al óptimo (considerando el mensaje completo)**

más de códigos por probabilidades

- **No es necesario que las frecuencias sean siempre las mismas**
- **Por ejemplo pueden depender de la parte ya vista del mensaje (el contexto)**

  p.ej. la frecuencia de “e” después de “th” puede ser mucho mayor

Aplicaciones de los códigos por probabilidades

- **¿Cómo generamos las probabilidades?**
- **Usar directamente las frecuencias no funciona muy bien (por ejemplo 4,5 bits por carácter en texto en inglés)**

Aplicaciones de los códigos por probabilidades

¿Cómo generamos las probabilidades?

- **Códigos con transformación:**
  run-length, move to front, residual
- **Probabilidades condicionadas**
  PPM

Run-length

- **Se codifica primero el mensaje contando el número de símbolos iguales seguidos**

  abbbaacccca → (a,1)(b,3)(a,2)(c,4)(a,1)

- **Después se utiliza código Huffman ...**

Run-length:Facsimile ITU T4

- **ITU= International Telecommunications Standard**
- **Usado por todos los Faxes domésticos**
- **Los símbolos son blanco y negro**
- **Para el código Huffman, hay tablas de frecuencias predeterminadas para (s,n)**

Move to front

- Transforma cada mensaje en una secuencia de enteros, a los que luego se les aplica Huffman o aritmético
- Empezamos con los símbolos en orden [a,b ...]
- Para cada símbolo del mensaje
  - Devuelve la posición del símbolo (b → 2)
  - Pon el símbolo al principio [b,a,c,d...]
- Se supone que los números serán pequeños

Move to front

dfac

d → 4 [d a b c e f g

f → 6 [f d a b c e g

a → 3 [a f d b c e g

c → 5 [c a f d b e g

→ 4635

Residual coding

- **Típicamente usado para símbolos que representan algún tipo de amplitud**
  - **p.ej. Nivel de gris, amplitud en audio**
- **Idea básica:**
  - **adivinar el valor basándose en el contexto**
  - **devolver la diferencia entre el valor verdadero y el adivinado**
  - **usar después códigos por probabilidades**

Residual coding: JPEG LS

- No confundir con el viejo lossless JPEG
- Usa 4 pixels como contexto

NW	N	NE
W	*	

- Intenta adivinar el valor de * basado en W, NW, N y NE

JPEG LS: paso 1

- **adivina:**

  min(N,W) si NW ≥ max(N,W)

  max(N,W) si NW< min(N,W)

  N+W-NW en otro caso

NW	N	NE
W	*	

JPEG LS: paso 2

- corrige la predicción anterior usando 3 cantidades: W-NW, NW-N, N-NE
- clasifica cada una en 9 categorías
- después de la corrección, residual coding

NW	N	NE
W	*	

Prob. condicionadas: PPM

- Usa los k símbolos anteriores como contexto
- Por ejemplo si de cada 12 veces que aparece “th” 7 veces está seguido de “e”

  p(e | th) = 7/12
- Utilizamos código aritmético para estas probabilidades
- Hay que mantener k pequeño para que el diccionario sea manejable

PPM ...

- **Pero los diccionarios pueden ser enormes**
- **Y hay muchos 0**
- **La solución es construir el diccionario sobre la marcha y si un contexto de tamaño k no aparece mirar el de tamaño k-1, k-2, ...**

  → el diccionario guarda las frecuencias para todos los contextos de tamaño ≤ k (incluido 0)

Burrows-Wheeler

- **Reciente, se usa para bzip**
- **Muy rápido**
- **Compresión razonable**

Burrows-Wheeler

- **Primero se ordena el contexto**

$d_1e_2c_3o_4d_5e_6$

$ecode_6$	d_1
$coded_1$	e_2
$odede_2$	c_3
$dedec_3$	o_4
$edeco_4$	d_5
$decod_5$	e_6

Burrows-Wheeler

- **Primero se ordena el contexto**

$d_1e_2c_3o_4d_5e_6$

$ecode_6$	d_1	→	$dedec_3$	o_4
$coded_1$	e_2		$coded_1$	e_2
$odede_2$	c_3		$decod_5$	e_6
$dedec_3$	o_4		$odede_2$	c_3
$edeco_4$	d_5		$ecode_6$	d_1 ←
$decod_5$	e_6		$edeco_4$	d_5

Burrows-Wheeler

- **Segundo: me quedo con la última columna del contexto**

dede c_3	o_4	c_3	o_4
code d_1	e_2	d_1	e_2
deco d_5	e_6	d_5	e_6
oded e_2	c_3	e_2	c_3
ecod e_6	d_1 ←	e_6	d_1 ←
edec o_4	d_5	o_4	d_5

Burrows-Wheeler

- **Propiedad: el orden de las letras iguales es el mismo para las dos columnas**

dede c_3	o_4	c_3	o_4
code d_1	e_2	d_1	e_2
deco d_5	e_6	d_5	e_6
oded e_2	c_3	e_2	c_3
ecod e_6	d_1 ←	e_6	d_1 ←
edec o_4	d_5	o_4	d_5

Burrows-Wheeler

- **A partir de esas dos columnas puedo recuperar la palabra**

c	o
d	e
d	e
e	c
e	d ←
o	d

Burrows-Wheeler

- **A partir de la salida (oeecdd) saco la columna de contexto → ordenando**

o	c
e	d
e	d
c	e
d ←	e
d	o

Burrows-Wheeler

- **Lo que devuelve el algoritmo es una codificación de la salida (oeecdd)**

- **Usa el move to front**

Burrows-Wheeler

- **Es más rápido que PPM y más lento que los LZ**
- **Comprime más que LZ y menos que PPM**

Resumen de lossless

- **Por probabilidades:**
  - **Huffman y código aritmético**
- **Aplicaciones de cod. por probabilidades:**
  - **Códigos con transformación:**
    run-length, move to front, residual
  - **Probabilidades condicionadas: PPM**
- **Lempel-Ziv**
- **Burrows-Wheeler**