Los algoritmos de compresión de datos sin pérdida de información (lossless)

Algoritmia para problemas difíciles
2-12-22
Elvira Mayordomo

Contenido

- **Lossless versus lossy**
- Algoritmos Lempel-Ziv
  - Mejoran los algoritmos con estados finitos (demostración en apéndice)
- Codificación por probabilidades:
  - Huffman y código aritmético
- Aplicaciones de cod. por probabilidades (apéndice)
- Otros: Burrows-Wheeler (apéndice)

Aquí

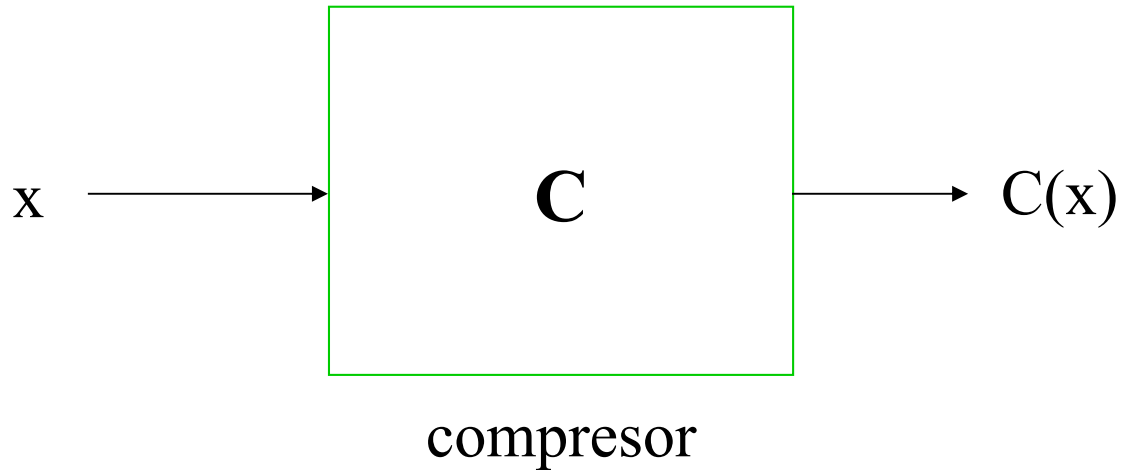
- Veremos técnicas concretas que se utilizan fundamentalmente combinadas
- Por ejemplo los mejores compresores de imágenes las utilizan prácticamente todas (más alguna lossy)

Referencias

- Guy E. Belloch: “Introduction to Data Compression” https://www.cs.cmu.edu/~guyb/realworld/compression.pdf
- Khalid Sayood: “Introduction to Data Compression”, Fifth Edition. Elsevier 2018

Algoritmos de compresión

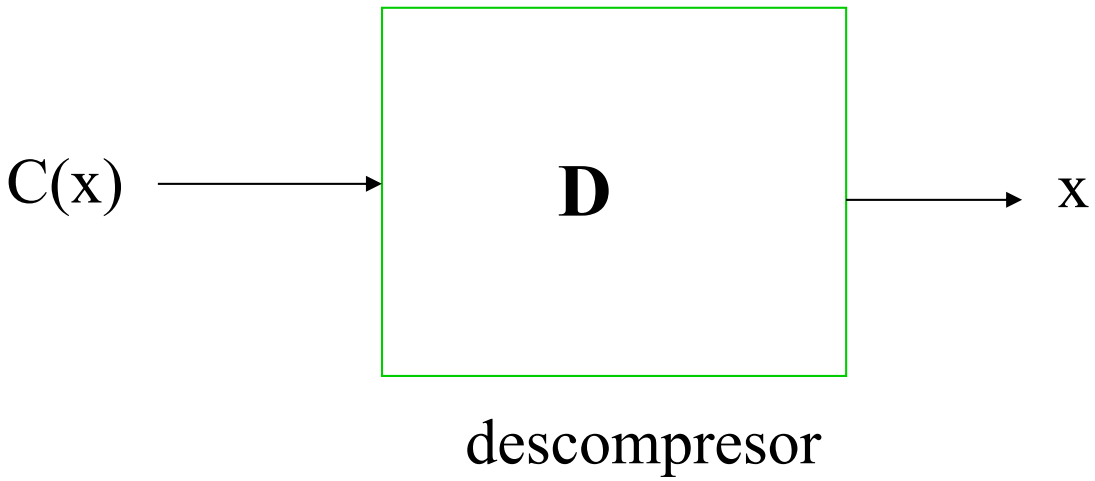
- La entrada y la salida son strings (cadenas, secuencias finitas)



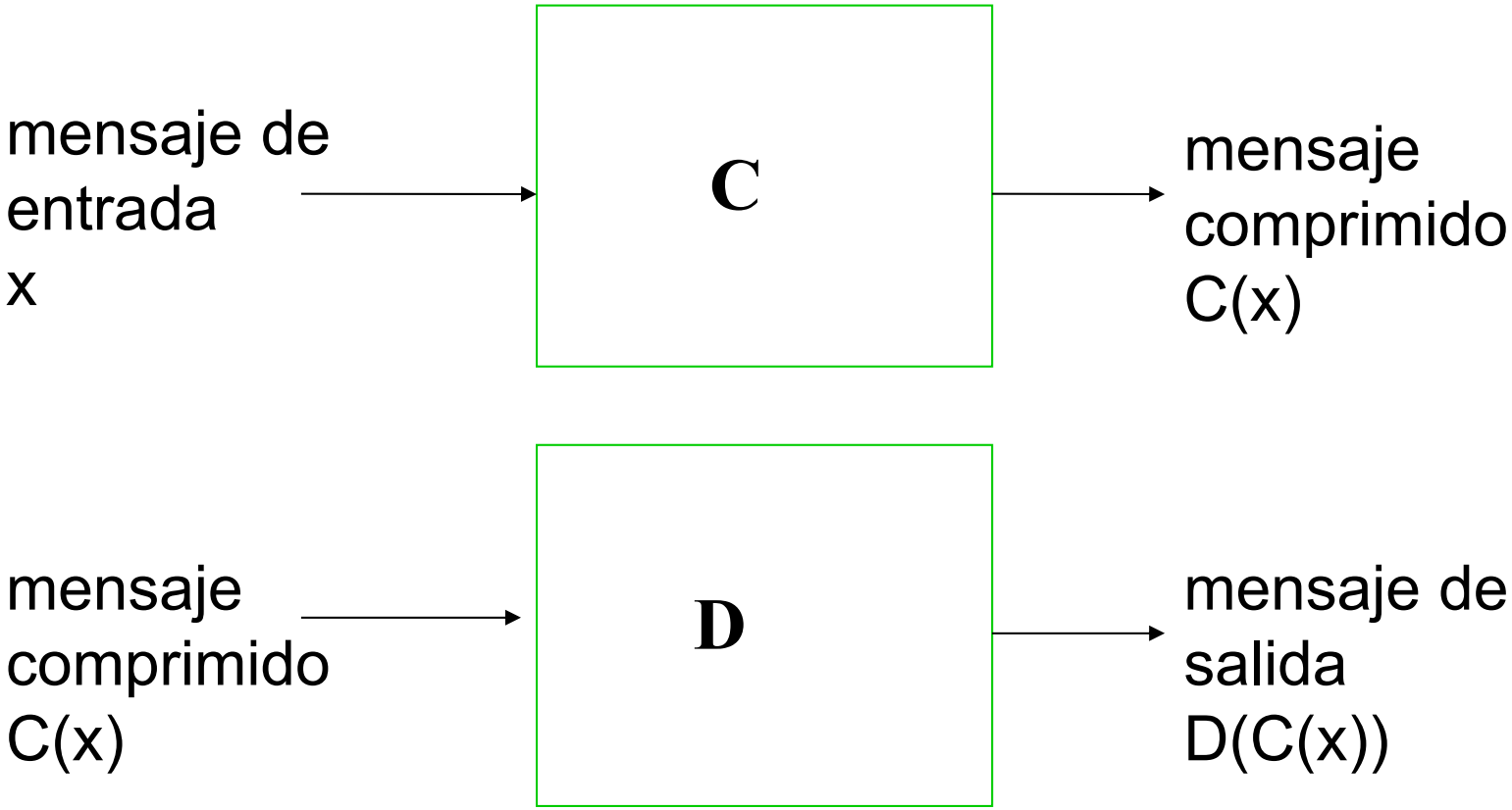
Nota: alfabetos de entrada y salida fijos, strings muy largas

Algoritmos de compresión

- A partir de la salida se puede reconstruir la entrada



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

Ejemplo

- Suponemos que queremos comprimir strings binarios
- El compresor sustituye 00000 por 00 y al resto de los grupos de 5 bits les pone 1 delante

010010000000110100000000000000010

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)

Contenido

- Lossless versus lossy
- **Algoritmos Lempel-Ziv**
  - Mejoran los algoritmos con estados finitos (demostración en apéndice)
- Codificación por probabilidades:
  - Huffman y código aritmético
- Aplicaciones de cod. por probabilidades (apéndice)
- Otros: Burrows-Wheeler (apéndice)

Algoritmos de Lempel-Ziv

- Principalmente son dos: LZ77 y LZ78
- Los más utilizados, con muchos litigios por patentes
- Se demostró que LZ78 era mejor que cualquier compresor de estados finitos

Hoy

- Vamos a empezar con un algoritmo concreto, LZ78, 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

El LZ78

- Es el algoritmo de compresión más 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

a|b|ab|ba|baa|aba|aa|bba

LZ78: idea principal

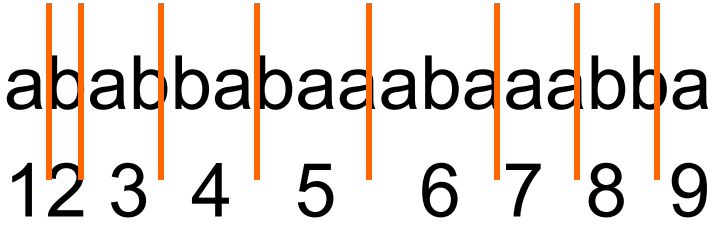
- Numeramos las frases

abababbabaaabaaabba

1 2 3 4 5 6 7 8 9

LZ78: idea principal

- Numeramos las frases



- Cada frase es una de las anteriores más un símbolo

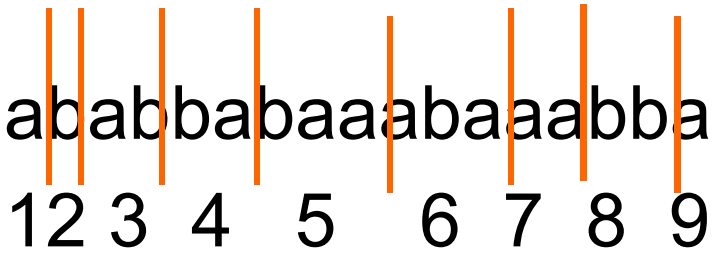
LZ78: idea principal

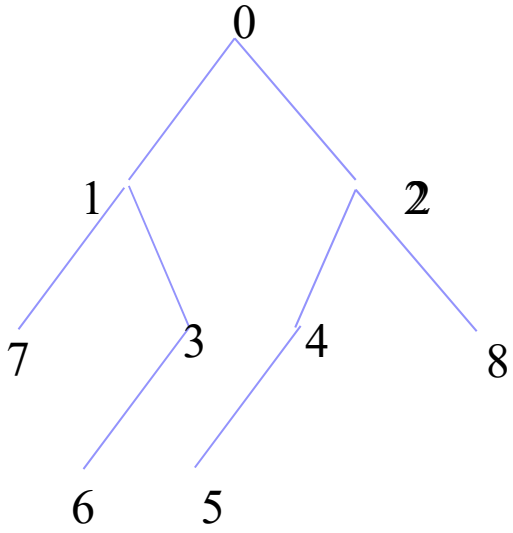
a	b	a	b	b	a	b	a	a	a	b	a	a	a	b	b	a
1	2	3	4	5	6	7	8	9

(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

LZ78: conceptos





- Alfabeto de entrada {a,b}
- Frase: baa es la frase 5
- Diccionario, conjunto de frases

Más ejemplos

- ababbbabaabab

LZ78: idea principal

- Se trata de “parsear” la entrada en frases diferentes

- ¿Cuál es la salida exactamente?

LZ78

abababaaabaaabba

12 3 4 5 6 7 8 9

(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 en binario 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 cuántos 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) y comprimido en z

  Mientras quede por leer

  Buscar la frase más larga w(i) de entre

  w(1) ... w(n) desde el cursor

  Si se acaba el input: z := zi

  si no (s es el siguiente símbolo)

  z := z(i,s)

(i en binario, usando $\lceil \log_2(n+1) \rceil$ bits, s con $\log_2 \alpha$ bits)

  Añadir al diccionario la nueva frase= w(i)s

  Output z

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
  Si se acaba el input z:= z w(i)
  si no
  Los siguientes $\log_2\alpha$ bits para sacar símbolo s
  Concatenar z:= z w(i) s
  Añadir al diccionario la nueva frase w(i)s
  Output z

LZ78: resumen

- Comprimir(x)
  -- En cada momento hemos encontrado las
  -- frases w(1) ... w(n) y comprimido en z
- 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:

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

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

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

Ratio de compresión

- Normalmente se mira

$$\rho_{LZ}(x) = \frac{|C(x)|}{|x|}$$

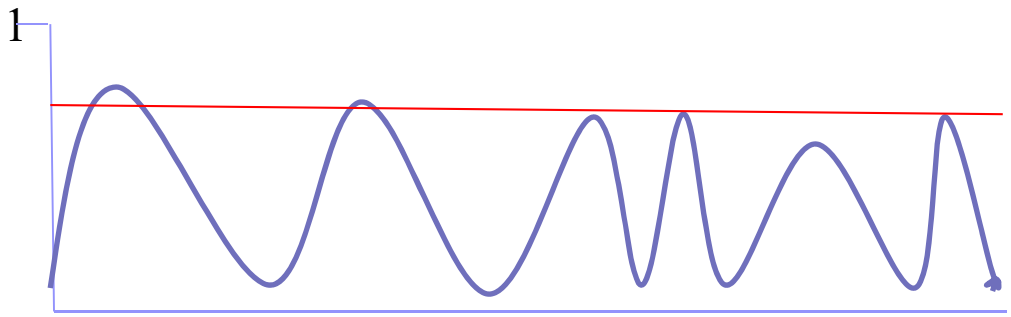
|x| es la longitud de x

Importante: nos interesa un alfabeto fijo y que la entrada sea muy larga.

Ratio de compresión

- Y para ver el comportamiento con entradas grandes se estudian asintóticamente **secuencias infinitas z**

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



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?

Los compresores FS

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

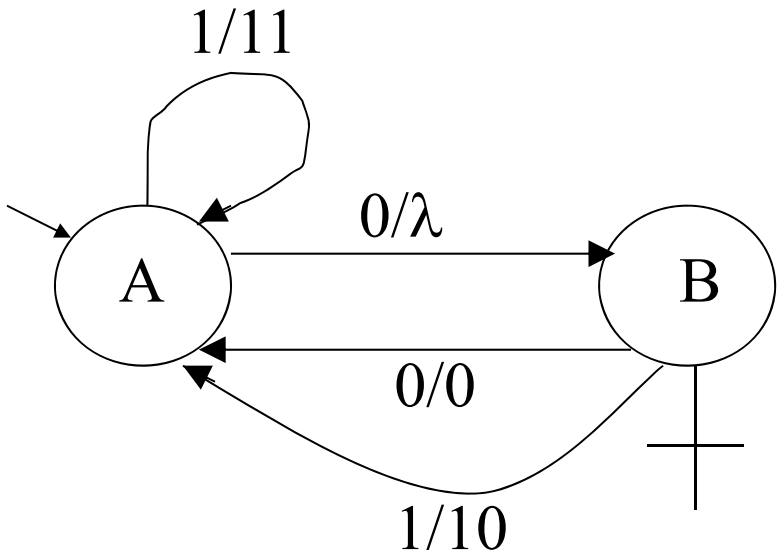
A continuación

- 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 ...

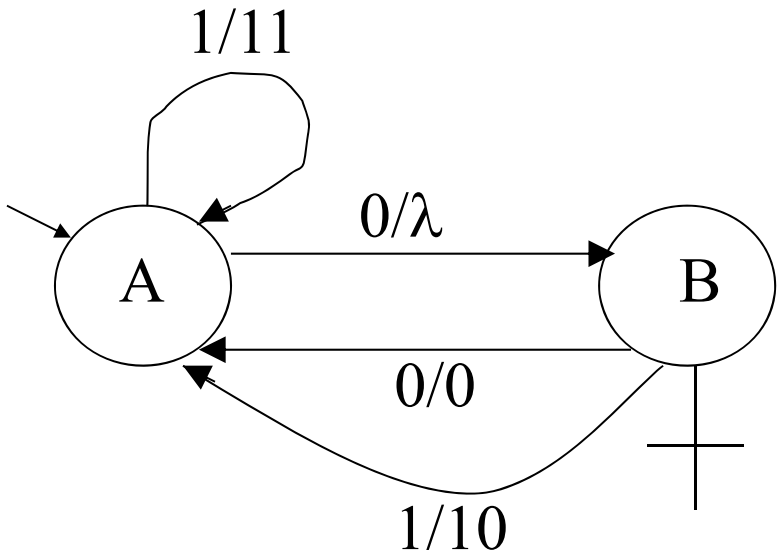
- Ziv, Lempel: “Compression of individual sequences via variable rate coding” IEEE Trans. Inf. Th., 24 (1978), 530-536
- Sheinwald: “On the Ziv-Lempel proof and related topics”. Procs. IEEE 82 (1994), 866-871

Ejemplo



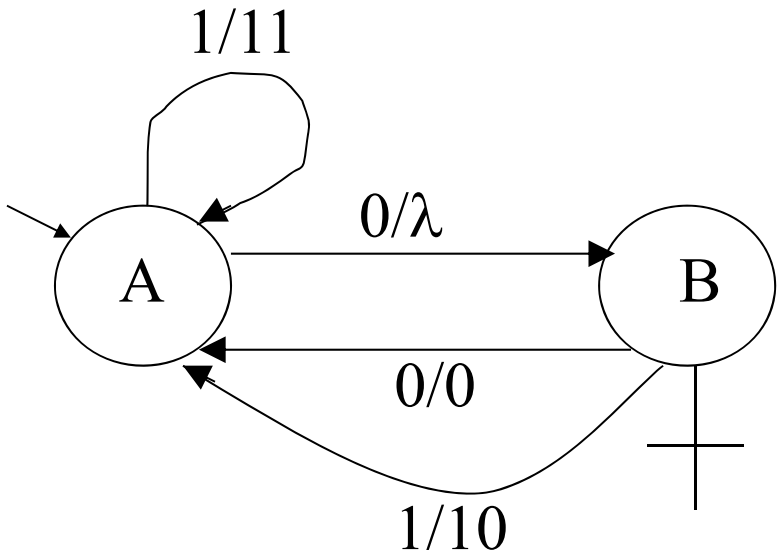
b/w **quiere decir que si leo el bit b la salida es w**

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

$\delta(q,b)$ 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

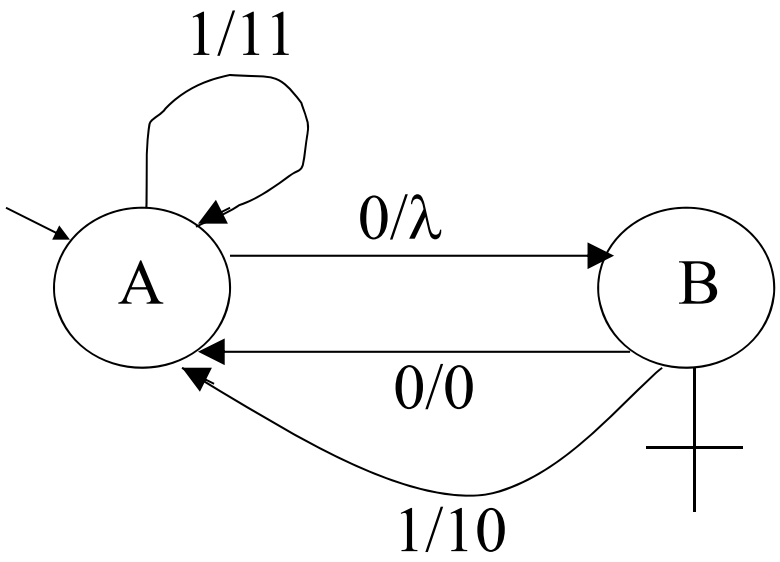
- Esto es IL (information lossless)

Definición de ILFSC

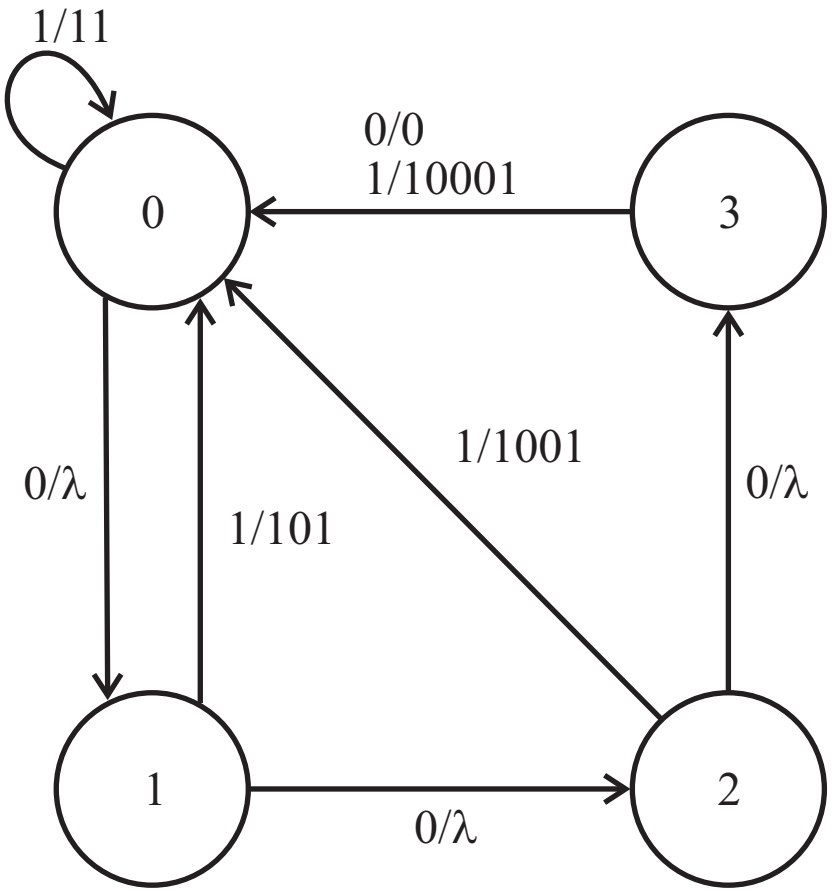
- Lo que exijo es que para cualquier estado q y para dos strings distintos x, x'

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

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

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

$|x|$ es la longitud de x

Importante: nos interesa un alfabeto fijo y que la entrada sea muy larga.

Compresión con ILFSC

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

$$\rho_A(z) = \limsup_n \ \rho_A(z[1..n])$$

Compresión con LZ78

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

LZ78: universal para ILFSC

- ¿Por qué?

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

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

LZ78: universal para ILFSC

- Para ello usamos que si tenemos una partición cualquiera en frases distintas de $z[1..n]= w(1) \ldots w(f(n))$ y A es un ILFSC* entonces

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

- El alfabeto de salida son 2 símbolos

>>>> Demostración: Ver apéndice

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

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

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

$$\rho_A(z[1..n]) \geq f(n) \log_2(f(n))/n - O(f(n))/n$$

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

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

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

$$\rho_{LZ}(z[1..n]) \approx t(n) \log_2(t(n)) / n + t(n)/n$$

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

$\rho_A(z[1..n]) \geq t(n) \log_2(t(n))/n - O(t(n))/n$

$\rho_{LZ}(z[1..n]) \approx t(n) \log_2(t(n)) /n + t(n)/n$

Como $t(n) \leq n/\sqrt{(\log n)}$ tenemos que $t(n)/n \rightarrow 0$

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

Hemos aprendido:

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

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

Sabíamos de LZ

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

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

Conclusión

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

Preguntas

- La catástrofe del bit de más

  001001001100110 ...

  100100100110 0 ...

- LZ78 comprime secuencias que los ILFSC no comprimen

LZ77

- LZ77: idea principal
- LZ77 en detalle
- LZ77 y LZ78

A continuación

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

Usamos

- 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

- Es el algoritmo de compresión más utilizado
- De él se derivan ZIP, GZIP, WINZIP, PKZIP, LZSS, LZB, LZH, ARJ, RFC
- El Deflate (librería zlib) es LZ77 + Huffman

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

ababbabaaabaaabba

Más ejemplos

- ababbbabaabab
- 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 ($\leq$ n)
  - c es el siguiente carácter

Nota: Si la estamos comprimiendo el fragmento a partir de la posición n, usamos $\log_2 n$ bits para escribir cada uno de los números p y l

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 (≤ n) tal que

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

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

  (n-i (en $\log_2 n$ bits), L (en $\log_2 n$ bits), x[n+L] (en $\log_2 \alpha$ bits))

  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) (es decir, 2 $\log_2 n + \log_2 \alpha$ bits)

  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

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

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 ??

Contenido

- Lossless versus lossy
- Algoritmos Lempel-Ziv
  - Mejoran los algoritmos con estados finitos (demostración en apéndice)
- **Codificación por probabilidades:**
  - Huffman y código aritmético
- Aplicaciones de cod. por probabilidades (apéndice)
- Otros: Burrows-Wheeler (apéndice)

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
  - Comprimimos en 2 pasadas
  - 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

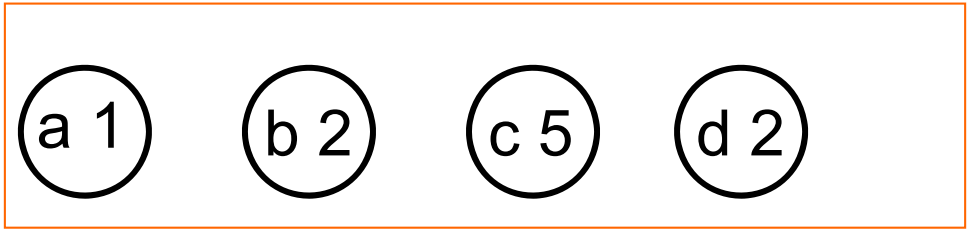
- Con un sencillo algoritmo se construye un prefix code que asigna codificación más corta a los símbolos más frecuentes

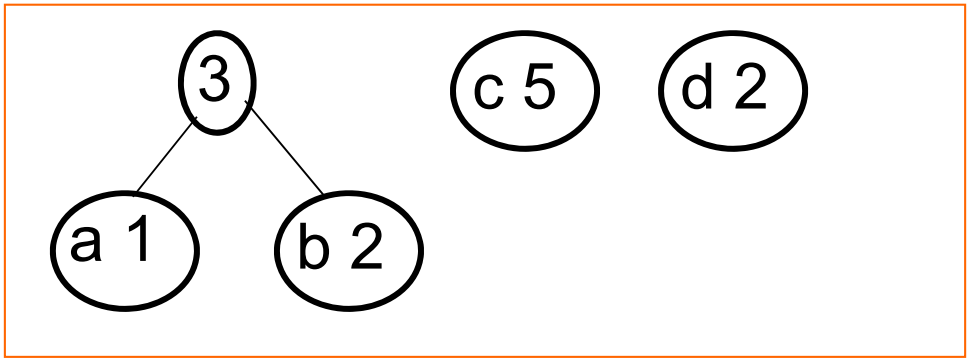
Código Huffman

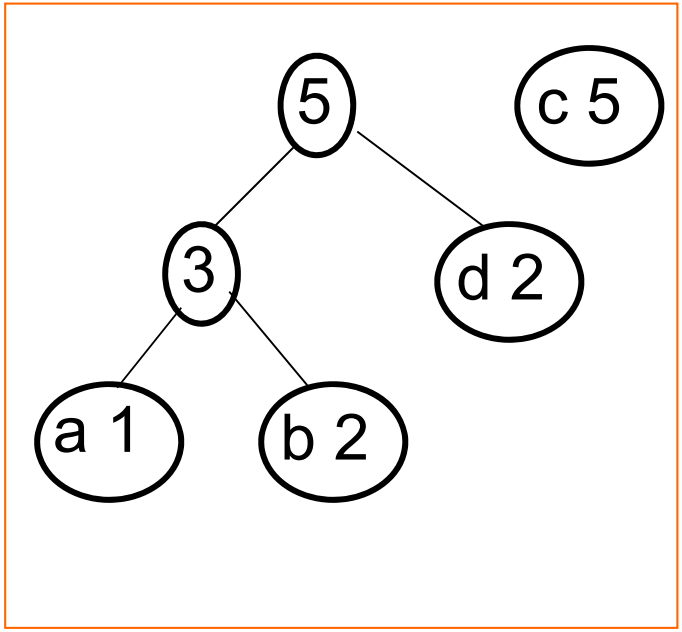
- Empieza con un vértice por cada símbolo, con peso la frecuencia de ese símbolo

- 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

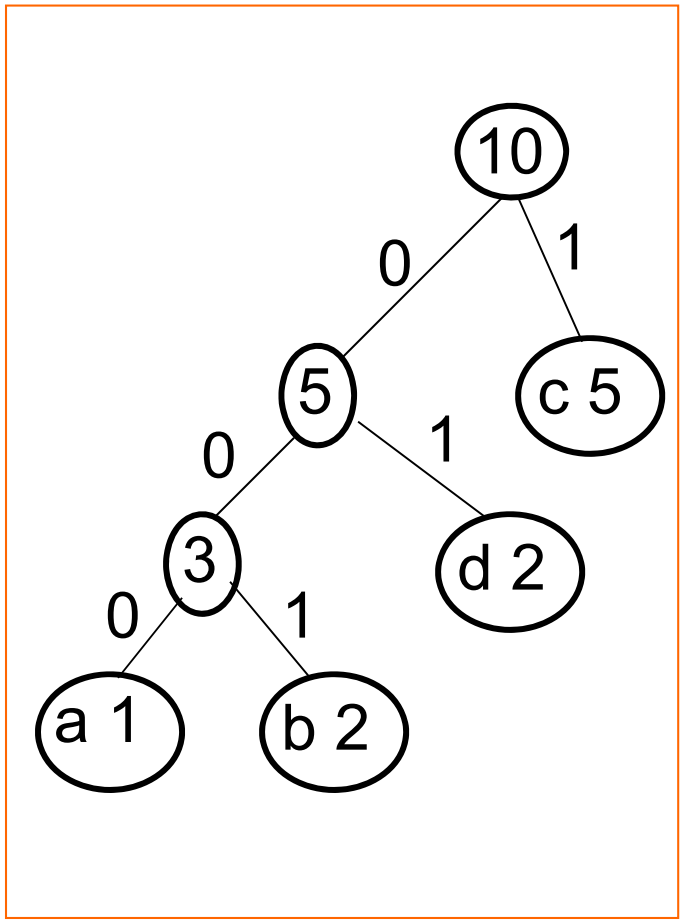
Código Huffman



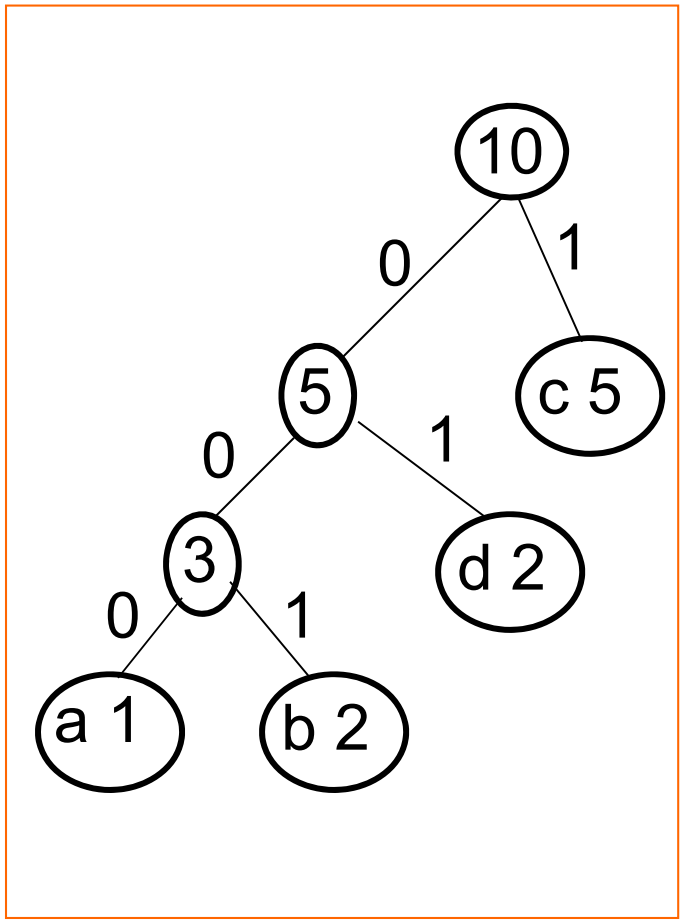




Código Huffman



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
- Hay que conocer las frecuencias de cada símbolo o bien calcularlas

Solución: código aritmético

- Permite “mezclar” los símbolos del mensaje de entrada (bloques de varios símbolos)
- 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:

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

Cuidado: cuanto menos se parezca esto a la frecuencia real peor funcionará
- la frecuencia de los símbolos por separado puede ser muy diferente de los bloques de símbolos (por ejemplo BB no aparece nunca en español)

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)	→	.001
[.33, .66)	→	.011
[.66,1)	→	.11

- De esta forma conseguimos un código libre de prefijos (o código prefijo)

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)
- Tiene que ser una longitud pequeña para conocer las frecuencias
- Pero el tamaño de los mensajes comprimidos es muy cercano al óptimo

Resumen de lossless

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

APÉNDICE

Contenido

- Lossless versus lossy
- Algoritmos Lempel-Ziv
  - Mejoran los algoritmos con estados finitos **(demostración en apéndice)**
- Codificación por probabilidades:
  - Huffman y código aritmético
- **Aplicaciones de cod. por probabilidades (apéndice)**
- **Otros: Burrows-Wheeler (apéndice)**

LZ78: universal para ILFSC

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

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

- El alfabeto de salida son 2 símbolos

LZ78: universal para ILFSC

- Intuitivamente, si el autómata es IL no puede comprimir demasiado frases distintas (salvo lo que pueda guardar en los estados)

- si tenemos una partición cualquiera en frases distintas de $z[1..n]= w(1) \ldots w(f(n))$ y A es un ILFSC entonces

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

Razonamiento:

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

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 0, $s^2 2$ de longitud 1, $s^2 2^2$ de longitud 2, etc

Razonamiento:

... cuando hay hay s^2 de longitud 0, $s^2 2$ de longitud 1, $s^2 2^2$ de longitud 2, etc

Si f(n) está entre

$$\Sigma_{i=0}^{L} s^2 2^i \quad y \quad \Sigma_{i=0}^{L+1} s^2 2^i$$

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

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

$$\Sigma_{i=0}^{L} i\, s^2 2^i + (L+1)\,(f(n) - s^2 2^{L+1} + s^2)$$

Razonamiento:

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

$\Sigma_{i=0}^{L} i\, s^2 2^i + (L+1)\,(f(n) - s^2 2^{L+1} + s^2) \geq$

$\geq f(n)\,(L-1)$

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

y $\Sigma_{i=0}^{L} i\, s^2 2^i$

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

Contenido

- Lossless versus lossy
- Algoritmos Lempel-Ziv
  - Mejoran los algoritmos con estados finitos
- Codificación por probabilidades:
  - Huffman y código aritmético
- **Aplicaciones de cod. por probabilidades**
- Otros: Burrows-Wheeler

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

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”

  $p(e \mid th) = 7/12$
- 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 $\leq$ k (incluido 0)

Contenido

- Lossless versus lossy
- Algoritmos Lempel-Ziv
  - Mejoran los algoritmos con estados finitos
- Codificación por probabilidades:
  - Huffman y código aritmético
- Aplicaciones de cod. por probabilidades
- **Otros: Burrows-Wheeler**

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)

- Luego 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

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