Scuola universitaria professionale della Svizzera italiana
SUPSI

Università della Svizzera italiana

Istituto Dalle Molle di studi sull'intelligenza artificiale

Algorithms and Data Structures Computer Engineering

Data Compression

Matteo Salani
matteo.salani@idsia.ch

The need of Data Compression

- Big data storage and transmission
- Network bandwidth does not increase at the same pace
- High quality multimedia data require massive data transmission (uncompressed 60fps 4K content requires 17.82 Gb/s)
- Compression algorithms (either general purposes or dedicated) are of extreme relevance

Terminology

- Message: the data to be transfer/compressed (sometimes referred as payload)
- Encoding and Decoding: the operations used to transform the message in to a (possibly) smaller size and encoded message and back
- Compression: operations used to reduce the size of a message
- Lossless: when the reconstructed message after decoding is identical to the message before encoding
- Lossy: when the reconstructed message is not identical (acceptable for some applications, e.g., audio/video compression)

Limits of data compression

- An algorithm that is able to compress everything does not exist
- All possible messages that can be obtained with n bits cannot be expressed conveniently with $n - 1$ bits
- Fundamentally compression works on the idea that not all n bits combinations appear with the same probability, therefore it is wise to represent the most probable messages with lower number of bits

Run Length Encoding - RLE

Basic idea

- Substitute, when convenient, a sequence of equal symbols with *counter* + *symbol*:

  AAABBBBAAABBCCCCCCCAB
  3A4B3A2B7CAB

- Effective on messages with long sequences of repeated symbols
- This basic version works on messages with no numbers (but an algorithm must work for all messages!)

Run Length Encoding - RLE

Commonly:

- Symbols use fixed length encoding (e.g. *byte, word, . . .*)
- Therefore all combinations of bits correspond to legal symbols

How to encode the counter?

- Counter occupies "one cell", that is the same amounts of bits of a symbol
- The counter is "escaped" with a (very low frequency) symbol (& in the example below)

AAABBBBAAABBCCCCCCCAB
AAA&DBAAABB&GCAB

Here, D and G encode the length (4 and 7)
Ideas on how to encode symbol &

RLE of binary sequences

- Substitute the sequence with the counter of equal bits alternating "0" and "1"
- Set a convention on the starting symbol (either 1 or 0)

011110010100011001100111
1 4 2 1 1 1 3 2 2 2 2

Encoding with a fixed length of 3 bits we obtain:

001100010001001001011010010010010

which is longer than the original message!
The RLE may be suitable for a limited number of cases

RLE of binary sequences

Exercise:
Encode the same sequence with an RLE algorithm and counters encoded with 2 bits

011110010100011100110011

Variable Length Encoding - VLE

Encode the message where symbols get an encoding of different length
Idea:

- Less bits (than usual) for most frequent symbols
- More bits (than usual) for least frequent symbols

How to understand when a symbol ends?

VLE - Unique encoding

If any encoding of a symbol IS NOT a prefix of the encoding of other symbols the interpretation of a sequence is unique.

Symbol	Code
A	11
B	00
C	010
D	10
E	011

1100010101011101100011111

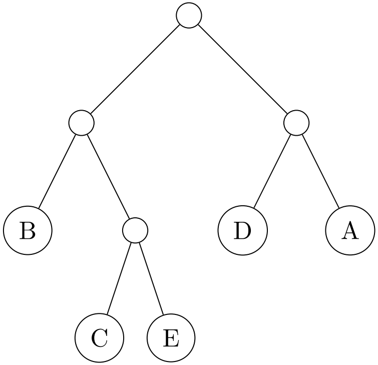
Unique interpretation:
A B C D D A D A B E A

VLE - Unique encoding

Generating unique encoding:

- Consider symbols as leaves of a binary tree
- The encoding is computed with the unique path root - leaf:
  - Assign a “0” code when exploring the left branch
  - Assign a “1” code when exploring the right branch

Symbol	Code
A	11
B	00
C	010
D	10
E	011

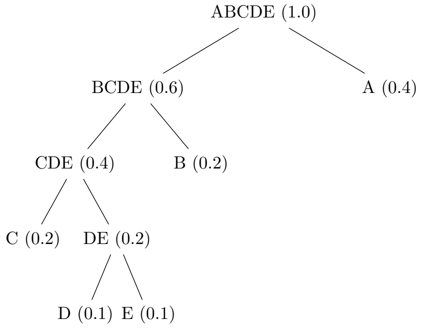


Huffman encoding

It needs the computation of the **frequencies** of the symbols:

Symbol	Frequency
A	0.4
B	0.2
C	0.2
D	0.1
E	0.1

Symbol	Encoding
A	1
B	01
C	000
D	0010
E	0011



Huffman encoding

The algorithm to obtain the Huffman tree is composed of two steps:

- Sort the symbols by decreasing frequencies
- While there is more than one symbol left, merge the two symbols with the lower frequency creating a tree node, reinsert the new symbol in the table of frequencies

It.1	f	It.2	f	It.3	f	It.4	f	It.5	f
A	0.4	A	0.4	A	0.4	BCDE	0.6	ABCDE	1.0
B	0.2	B	0.2	CDE	0.4	A	0.4		
C	0.2	C	0.2	B	0.2				
D	0.1	DE	0.2						
E	0.1								

Huffman encoding

Exercise:

- Produce the Huffman encoding of the following message:

“small example of compression algorithm with huffman’s enconding“

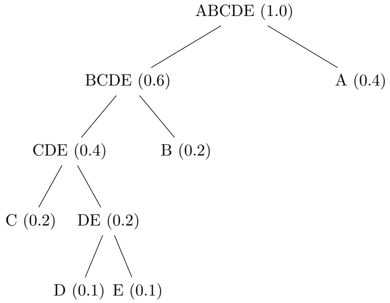
Quantitative measure of an encoding

Definition:

- The *weighted path length* w_p of a leaf is its weight (frequency) times its depth
- The *external path weight* is the sum of the weighted path lengths

The external path weight is a measure of the effectiveness of an encoding.
Theorem:

- The Huffman encoding exhibit the smallest external path weight



Canonical Huffman encoding

A generic Huffman encoding require that the sender (compressor) must provide explicitly the encoding of the symbols as an header of the message.

Canonical Huffman encoding:

- The idea is to send the lengths of the symbols' encoding only by computing a standardized encoding both at the sender and at the receiver:
    - Build an Huffman encoding with the given algorithm
    - Sort the symbols by increasing encoding lengths
    - Assign a sequence of zeros "0" to the first symbol according to the length of the encoding
    - For the consecutive symbols with the same length just add "1" to the previous encoding
    - For the consecutive symbols with higher length add "1" to the previous encoding and left shift the encoding with an appropriate amount of zeros "0"

Canonical Huffman encoding example

Symbol	Encoding	Length	Canonical encoding
A	1	1	0
B	01	2	10
C	000	3	110
D	0010	4	1110
E	0011	4	1111

Shannon Fano's encoding tree

Shannon Fano is another algorithm for producing a prefix code using symbols' frequencies.
Shannon–Fano codes are suboptimal, they do not always achieve the lowest possible expected external path length but they have an expected codeword length within 1 bit of optimal.
The implementation of Shannon-Fano encoding is easier and elegantly designed as a recursive algorithm

1. Compute list of frequency counts
2. Sort the lists of symbols according to frequency
3. Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible.
4. The left part of the list is assigned the binary digit 0, and the right part is assigned the digit 1.
5. Recursively apply the steps 3 and 4 to each of the two halves, until the remaining lists have length one.

Shannon Fano's encoding tree example

Symbol	A	B	C	D	E
Frequencies	0.38	0.18	0.15	0.15	0.14
1st	0		1		
2nd	0	1	0	1	
3rd	0	1	0	0	1
Encoding	00	01	10	110	111

Dictionary based encodings

A dictionary is kept as a reference for repeated sequences of symbols. Instead of the sequence of symbols, its reference in the dictionary is sent. There are two families of dictionary based encodings:

- **Implicit dictionary**, the dictionary is the message itself
- When a sequence is found in the data already sent, a reference is transmitted
    - LZ77 - A.Lempel and J.Ziv, 1977
    - LZSS - Storer and Szymanski, 1982
    - Applications: Gzip, Squeeze, LHA, PKZIP
- **Explicit dictionary**, the dictionary is built during the encoding/decoding process
- When a sequence is found in the dictionary, a reference is transmitted
- The dictionaries at source and destination must be identical!
    - LZ78 - A.Lempel and J.Ziv, 1978
    - LZW - T.Welch, 1984
    - Applications: Compress, Gif, CCITT, ARC, PAK

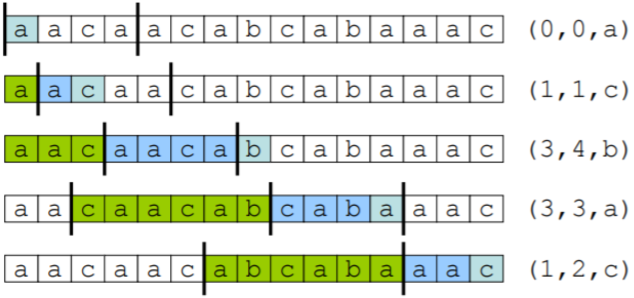
LZ77

Implicit dictionary, the message itself is used as a dictionary.
Only a small portion of the message, called *Sliding Window* is used as a dictionary

- **Sliding Window**: part of the message already encoded (normally 2 . . . 64kb)
- **Lookahead buffer**: a portion of the message to be encoded
- **Encoding:** sequence of codes of the format *(B,L) C*
  - *B*: number of steps to go back in the sliding window to find the start of the sequence to be repeated
  - *L*: length of the sequence to be repeated
  - *C*: next character in the lookahead buffer
- The decoding process keeps track of the sliding window

LZ77 - Example

- Window size = 6
- Lookahead size = 4



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

LZ77 - Another example

Message:

Position	1	2	3	4	5	6	7	8	9
Symbol	A	A	B	C	B	B	A	B	C

Steps:

Step	Cursor	Match	Code
1	1	-	(0,0)A
2	2	A	(1,1)B
3	4	-	(0,0)C
4	5	B	(2,1)B
5	7	AB	(5,2)C

LZ77 - Decoding

- The decoder simply builds the message and uses it as dictionary
- In case $L > B$, the decoder continues using the newly added sequence.
- Example. Dictionary = *abcd*, next code $(2, 9)e$, decoding *abcdcdcdcdcdce*

```
\\ cur = current position in output buffer
\\ (B,L) C = next processed code
for (i=0; i<L; i++){
        output[cur + i] = output[cur - B + i];
}
output [cur + L] = C;
cur += L + 1;
```

Issues of LZ77:

- Compressor is normally slow (finding occurrences of subsequences of symbols)
- Encoding is cumbersome: three values must be sent

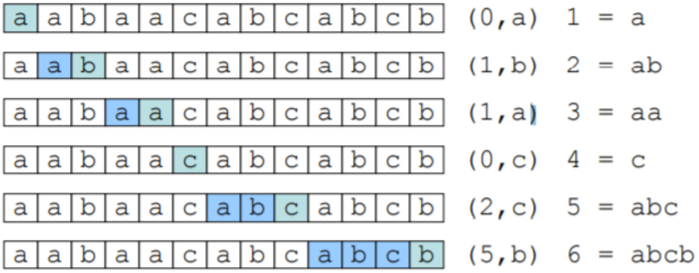
LZ78

The dictionary of symbols is **explicit**, it is built during the coding/decoding dictionary, the dictionary stores an index for each symbol.
The algorithm is simple:

- Find the longest matching string S in the dictionary
- Send the index of S and the next symbol of the message c
- Add Sc to the dictionary (we are sure that Sc does not exist in the dictionary)

Implementation plays an important role in the efficiency and effectiveness of the algorithm (how to store and check the dictionary, how many entries to store)

LZ78 - Example

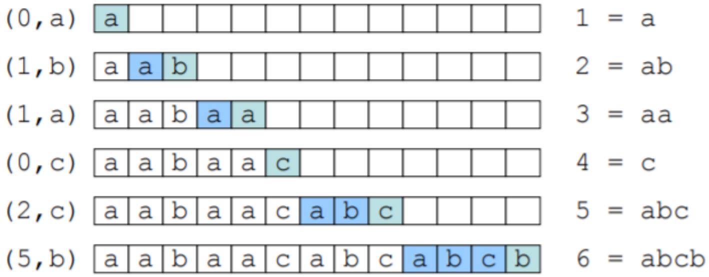


LZ78 - Another example

Message: ABBCBCABA

Step	Values *Sc*	Code	Dictionary
1	A	0 A	1 A
2	B	0 B	2 B
3	BC	2 C	3 BC
4	BCA	3 A	4 BCA
5	BA	2 A	5 BA

LZ78 - Decoding Example


(0,a) a | 1 = a
(1,b) a a b | 2 = ab
(1,a) a a b a a | 3 = aa
(0,c) a a b a a c | 4 = c
(2,c) a a b a a c a b c | 5 = abc
(5,b) a a b a a c a b c a b c b | 6 = abcb

LZ78 - Another decoding example

Step	Code	Reconstructed message	Dictionary
1	0 A	A	1 A
2	0 B	AB	2 B
3	2 C	ABBC	3 BC
4	3 D	ABBCBCD	4 BCD
5	2 A	ABBCBCDBA	5 BA

Messaggio: ABBCBCDBA

LZW

LZW is an improvement of LZ78

- The encoding is improved. It avoids to send the additional character c to the encoded message.
- It requires a *standard* base dictionary for both the encoder and the decoder (e.g. all byte values from 0 to 255).
- The base dictionary is known and it does not require to be sent.

LZW - Encoding pseudo-code

Algorithm 1 LZW - Encoding

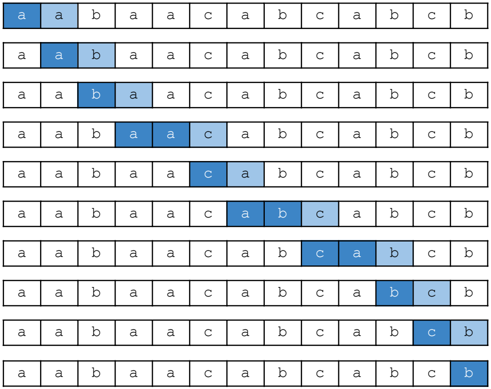
```
1: Initialize $DICTIONARY$
2: $STRING$ = get first input symbol
3: while there are unprocessed symbols do
4:     $SYMBOL$ = get input symbol
5:     if $STRING + SYMBOL \in DICTIONARY$ then
6:         $STRING = STRING + SYMBOL$
7:     else
8:         output the code for $STRING$
9:         add $STRING + SYMBOL$ to $DICTIONARY$
10:        $STRING = SYMBOL$
11:    end if
12: end while
13: output the code for $STRING$
```

LZW - Decoding pseudo-code

Algorithm 2 LZW - Decoding

```
1: Initialize $DICTIONARY$
2: $CODE$ = first input code
3: $STRING = DICTIONARY[CODE]$
4: output $STRING$
5: while there are unprocessed codes do
6:     $CODE$ = next input code
7:     $SYMBOL$ = get input symbol
8:     if $DICTIONARY[CODE]$ is not defined then
9:         $VALUE = STRING + STRING[0]$
10:    else
11:        $VALUE = DICTIONARY[CODE]$
12:    end if
13:    output $VALUE$
14:    add $STRING + VALUE[0]$ to $DICTIONARY$
15:    $STRING = VALUE$
16: end while
```

LZW - Encoding Example



Message	Code	Dictionary
a a b a a c a b c a b c b	112	256=aa
a a b a a c a b c a b c b	112	257=ab
a a b a a c a b c a b c b	113	258=ba
a a b a a c a b c a b c b	256	259=aac
a a b a a c a b c a b c b	114	260=ca
a a b a a c a b c a b c b	257	261=abc
a a b a a c a b c a b c b	260	262=cab
a a b a a c a b c a b c b	113	263=bc
a a b a a c a b c a b c b	114	264=cb
a a b a a c a b c a b c b	113	

LZW - Decoding Example

Code	Message													Dictionary
112	a													
112	a	a												256=aa
113	a	a	b											257=ab
256	a	a	b	a	a									258=ba
114	a	a	b	a	a	c								259=aac
257	a	a	b	a	a	c	a	b						260=ca
260	a	a	b	a	a	c	a	b	c	a				261=abc
113	a	a	b	a	a	c	a	b	c	a	b			262=cab
114	a	a	b	a	a	c	a	b	c	a	b	c		263=bc
113	a	a	b	a	a	c	a	b	c	a	b	c	b	264=cb