COMP 102: Excursions in Computer Science

Lecture 16: Data compression

Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

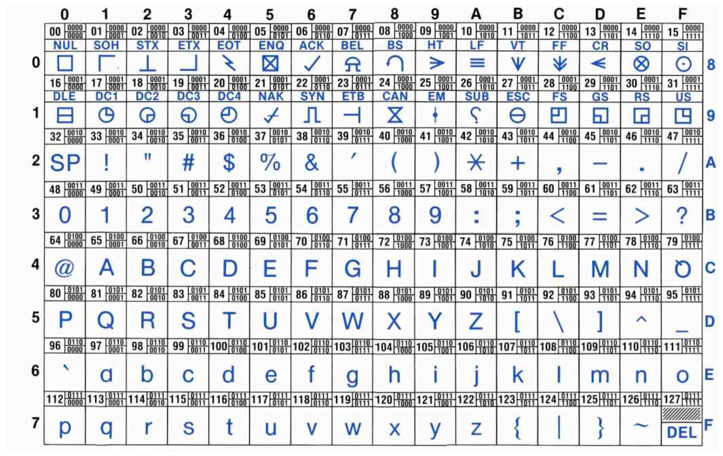
Class web page: www.cs.mcgill.ca/~jpineau/comp102

Why compress data?

- Up until now we've assumed that anything we want to encode can be encoded
  - variables,
  - states in finite state machine
  - etc.
- Today we discuss how to select an efficient encoding for our problem.

ASCII encoding

- Each character is encoded using 1 byte.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL	08 BS	09 HT	10 LF	11 VT	12 FF	13 CR	14 SO	15 SI	8
1	16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB	24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US	9
2	32 SP	33 !	34 "	35 #	36 $	37 %	38 &	39 '	40 (	41)	42 *	43 +	44 ,	45 –	46 .	47 /	A
3	48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?	B
4	64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O	C
5	80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _	D
6	96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o	E
7	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 DEL	F

Data Compression

- **Data compression** means encoding a data file using fewer bits than the original file.
  - Possible when the file has *redundancies*

Example: In English, the letter 'e' occurs more often than the letter 'z'. If we could come up with an encoding scheme where 'e' would require less bits than 'z' or any other letter, then the encoded data file would now be slightly shorter.

Types of compression

1. **Lossless** - encoding data and then decoding it will give back exactly the original data
   - we don't lose any information by compressing it
   - the encoded file will be longer as a result

2. **Lossy** - the decoded data will not be exactly the same as the original, but will be close enough
   - we lose some information by encoding it, but the encoded file is much shorter
   - this is ok for image/audio compression but not for text files

Definitions

- An **alphabet** is a set of symbols that we wish to encode.

- Examples:
  1. the digits from 0 to 9 : A = {0,1,...,9}
  2. lowercase letters: A = {a,b,c,...,z}
  3. pixels in an image

Codewords

- A **codeword** (or code) is a mapping from an alphabet to a set of binary strings. The code of a symbol A_i is $C(A_i)$
- The **length** of a codeword is the number of bits in the codeword, and is denoted by λ

Example: $A= \{A_1, A_2, A_3\}$

  - $C(A_1) = 0, C(A_2) = 0011, C(A_3) = 0$
  - $\lambda(A_1) = \lambda(A_3) = 1, \quad \lambda(A_2) = 4$

- Is this a good code?
  - No! A_1 and A_3 are mapped to the same binary string. This will make decoding them impossible.
- Better code: $C(A_1) = 0, C(A_2) = 0011, C(A_3) = 1$

Types of codewords

1. **Fixed length** - the codewords for all symbols have the same length λ
   - good: easy to decode (read the same number of bits at a time from the encoded file)
   - bad: the encoded file might be longer

2. **Variable length**
   - pro: encoded file will be shorter
   - con: need a smart algorithm to read bits from the encoded file

Types of code words

Example: A= {A_1, A_2, A_3}. Encode s = $A_2\,A_1\,A_1\,A_3\,A_1$

- Fixed length: $C(A_1) = 00$, $C(A_2) = 10$, $C(A_3) = 11$
  - $C(s) = 1000001100$
  - Can we decode it? Yes! Just read 2 bits at a time, and look up which symbol has that codeword
- Variable length: $C(A_1) = 0$, $C(A_2) = 10$, $C(A_3) = 1$
  - $C(s) = 100010$
  - Can we decode it? No! When we get to the last 2 bits, we don't know whether it should be A_2 or $A_3\,A_1$.

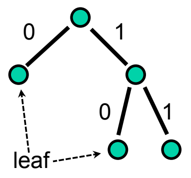
How can we fix this?

Prefix codes

- A **prefix code** is a code such that no codeword is a prefix of any other codeword.
- Is $C(A_1) = 00$, $C(A_2) = 10$, $C(A_3) = 11$ a prefix code?
  - Yes!
- Is $C(A_1) = 0$, $C(A_2) = 10$, $C(A_3) = 1$ a prefix code?
  - No! $C(A_3)$ is a prefix for $C(A_2)$.
- Better: $C(A_1) = 0$, $C(A_2) = 10$, $C(A_3) = 11$.
  - Now we can decode the string $C(s) = 1000110$

Side note: binary trees

- A **binary tree** is a data structure where each node has at most two children.
- Sometimes, it is useful to label left branches with '0' and right branches with '1' .
- A node with no children is called a **leaf**.

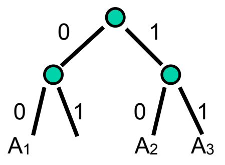


Prefix codes as binary trees

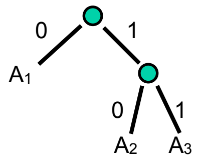
- We can represent any prefix code as a binary tree, such that each codeword follows a different branch from the root note to a leaf.
- This works for variable or fixed length codes.

Prefix codes as binary trees

- Example 1: $C(A_1) = 00$, $C(A_2) = 10$, $C(A_3) = 11$



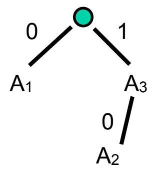
- Example 2: $C(A_1) = 0$, $C(A_2) = 10$, $C(A_3) = 11$



Can we build a binary tree for: $C(A_1) = 0$, $C(A_2) = 10$, $C(A_3) = 1$?

Prefix codes as binary trees

- What about codes that are **not** prefix trees?
- $C(A_1) = 0$, $C(A_2) = 10$, $C(A_3) = 1$



- This doesn't work because A_3 is no longer a leaf! This is why in the example we weren't able to decode any strings containing "10".

Frequencies of symbols

- We can talk about how likely a symbol is to appear in a string in terms of probabilities
- When we say $p(A_1)= 0.5$, that means that 1 out of 2 symbols in a string is likely to be A_1
- The probabilities for all symbols should sum to 1
  - $p(A_1)= 0.2$, $p(A_2)= 0.5$, $p(A_3)= 0.3$

Huffman Coding

- **Input**: any alphabet $A= \{A_1, A_2, ...,A_N\}$ and the frequencies $p(A_1),..., p(A_N)$ of the symbols in the alphabet
- **Output**: an optimal prefix code such that the symbol with the highest frequency has the shortest codeword and the symbol with the lowest frequency has the longest codeword.

Huffman Coding Algorithm

1. **Re-order** the symbols in order of decreasing frequencies (i.e. the number with the highest frequency comes first)

2. **Merge** the last two symbols, A_N and A_{N-1} into a new symbol $A_{N,N-1}$ such that $p(A_{N,N-1}) = p(A_N) + p(A_{N-1})$. Remove A_N and A_{N-1} from the list, and add $A_{N,N-1}$ instead

3. **Add** A_N and A_{N-1} to the Huffman tree (if not already there)

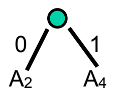
4. **Repeat** steps 1-2 until there is only one symbol left

17 *Joelle Pineau*

Huffman Coding example

- Input: $A = \{A_1, A_2, A_3, A_4\}$
- $p(A_1) = 0.25$, $p(A_2) = 0.2$, $p(A_3) = 0.4$, $p(A_4) = 0.15$

1. Order the symbols:
   - » $p(A_3) = 0.4$
   - » $p(A_1) = 0.25$
   - » $p(A_2) = 0.2$
   - » $p(A_4) = 0.15$
2. Merge A_2 and A_4, creating $A_{2,4}$ with $p(A_{2,4}) = 0.35$
3. Update the list:
   - » $p(A_3) = 0.4$
   - » $p(A_{2,4}) = 0.35$
   - » $p(A_1) = 0.25$
4. Add A_2 and A_4 to the tree:

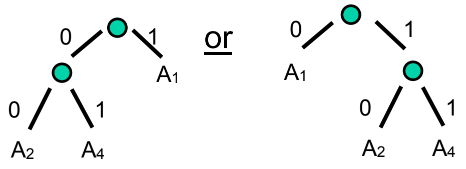


18 *Joelle Pineau*

Huffman Coding example(2)

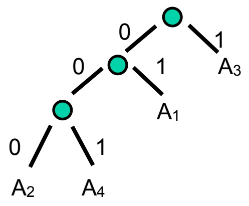
5. Merge $A_{2,4}$ and A_1, creating $A_{1,2,4}$ with $p(A_{1,2,4}) = 0.6$
6. Update the list:
   » $p(A_{1,2,4}) = 0.6$
   » $p(A_3) = 0.4$

7. Since $A_{2,4}$ is already in the tree (i.e the node above A_2 and A_4), add A_1 to the tree. Note that this can be added either to the left or to the right:



Huffman Coding example(2)

8. Merge $A_{1,2,4}$ and A_3, creating $A_{1,2,3,4}$ with $p(A_{1,2,3,4}) = 1$
9. Add A_3 to the tree



- Now we can read off the codewords from the tree:
- $C(A_1) = 01$, $C(A_2) = 000$, $C(A_3) = 1$, $C(A_4) = 001$

Huffman coding remarks

- We could have constructed more than one tree (i.e. adding the new nodes to the left or to the right)
  - This decision does not affect the final codeword lengths.
- If there are more than two symbols with the same probability, we can choose any of them to merge (order doesn't matter).
  - This will affect the codewords in the sense that one of those symbols will have a longer codeword than the other ones.
  - But because the symbols have equal frequencies, it does not make a difference which one has a longer codeword.

Summary

- Understand the purpose of compressions.
- Know how to define: alphabet, codeword, prefix codes.
- Understand the trade-off between symbol frequency and length of codeword.
- Be able to encode and decode text using Huffman coding.