Applications: Compression algorithms

CS240: Data Structures and Data Management
Slide Set 15

Jérémy Barbay

March 14-16 2006

Outline

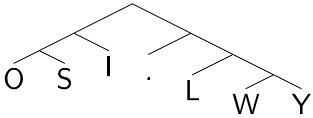
Idea

- Why should a character like ‘`Z`’ use the same amount of space as an ‘`e`’
- Use a **variable-length** encoding
- Popular characters are encoded with shorter bit patterns
- Danger: Consider the encoding
  - ‘`A`’ – ‘`101`’
  - ‘`M`’ – ‘**110**’
  - ‘`I`’ – ‘`101`**110**’

  Decode “110101110101”

Encoding Tree

- We need a **prefix code**
- The prefix of one code cannot represent another symbol
- An encoding tree ensures our code has this property
  - Proper binary tree, with each character at a leaf



Coding

- **Encoding**
  - Using the tree, build a dictionary of letters–codes
  - For each letter of input, output corresponding code
  - Encode: `LOSSY`
    110000010011111
- **Decoding**
  - Traverse the tree using the input bits
  - If you encounter a letter, output it and return to root
  - Decode: `001011101101111`
    SILLY

Building Tree

- Not all possible trees encode as well
- How do we build the "best" tree
- First determine the frequency of each character
- Build the tree bottom up using a priority queue
- Use tree weights as the priority
  - Weight of a leaf is frequency of the character, $f(c)$
  - Weight of other trees is the sum of its leaves

Huffman Coding

BuildHuffTree(*input*)

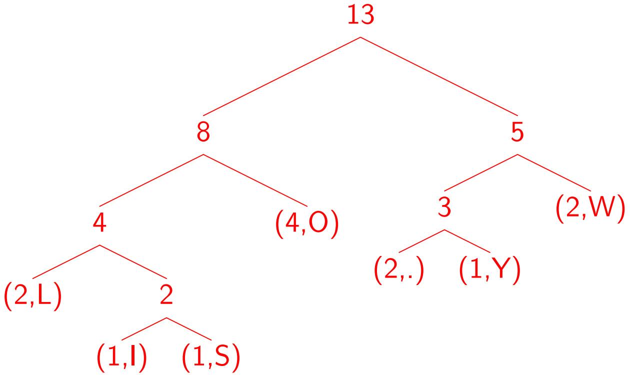
for each unique character c **do**
  Create a tree, T, with c as the only node
  $\mathrm{PQ}.\textsc{insert}(\ f(c),\ T\ )$ /* use $f(c)$ as priority */
end for
while $\mathrm{PQ}.\textsc{size}() > 1$ **do**
  $T_1 \leftarrow \mathrm{PQ}.\textsc{extractMin}()$ — (with priority w_1)
  $T_2 \leftarrow \mathrm{PQ}.\textsc{extractMin}()$ — (with priority w_2)
  Create a tree, T, with empty root and T_1, T_2 as children
  $\mathrm{PQ}.\textsc{insert}(\ w_1 + w_2,\ T\ )$
end while
return $\mathrm{PQ}.\textsc{extractMin}()$

Example

- **Input**: `WOOL·IS·WOOLY`

c	·	I	L	O	S	W	Y
$f(c)$	2	1	2	4	1	2	1

- Huffman Tree:



Comparing Encoding Trees

- The tree presented earlier could also encode the input string. Which one is better?
- Consider the length of the encoding generated by each tree:

$$\begin{aligned} & \sum_c (\text{number of occurrences of } c) \cdot \text{length of code for } c \\ = & \sum_c f(c) \cdot \textsc{Depth}(c) \end{aligned}$$

- Call this value the weighted path length (WPL) for the tree

Huffman Optimality

Theorem

- *Let S be a set of n characters and their frequencies*
- *Let H be a Huffman Tree constructed over S*
- *Let T be any encoding tree for S*

$$WPL(H) \leq WPL(T)$$

Claim

If x, y are nodes of minimal frequencies,
then there is an optimal tree in which they are sibling

Proof of the Claim

Proof.

In an optimal tree, moving x and y to be sibling leaves of max depth does not increase cost.

1. If x does not have max depth, take the node u of max depth and swap x and u: as x is of minimal frequency, there is no loss in the encoding.
2. if y is of maximal depth, it can be switched iwth x's sibling, and else nothing is lost by the switch.

□

Proof of Theorem

Using previous claim, suppose that x and y of minimal frequency are siblings in both H and T.
Note that, for $f(z) = f(x) + f(y)$:

$$\begin{aligned} WPL(H(S)) &= WPL(H(S)/\{x,y\} \cup \{z\}) + f(x) + f(y) \\ WPL(T(S) &= WPL(T(S)/\{x,y\} \cup \{z\}) + f(x) + f(y) \end{aligned}$$

1. We prove the result by induction on n
2. let H_n="for each set S of n chars and frequencies, $WPL(H(S)) \leq WPL(T(S)) \forall T$"
3. Base case: H_2, there is only one possible tree, $H(S) = T(S)$
4. Suppose H_{n-1} for $n \geq 3$
   - by H_{n-1}, $WPL(H(S)) \leq WPL(T(S))$;
   - by previous note, $WPL(H(S)/\{x,y\} \cup \{z\} \leq WPL(T(S)/\{x,y\} \cup \{z\})$.
5. Hence, $\forall n \geq 2$, H_n is true, i.e. the Hufman tree is optimal.

Details

- How is the decoder going to know the encoding?
  The frequencies must be transmitted, or the tree.
- How many passes of the data are required for the encoder?
  Two passes: one for computing the frequencies, one for the encoding.
- **Improvement** – Adaptive Huffman
  Read the text by blocks, use the frequencies of the previous block to encode the current block.

Outline

Compression

- Why?
  To take advantage of the non-uniform distribution of the input.
- Exact or Approximate?
  - Lossless, as in text compression
  - Lossy, as in images, music, video

Ideas

- Cannot compress random data
- Must exist some underlying patterns in the data
- Patterns in text files:
  - *e* is more frequent than *z*.
  - *th* is more frequent than *qv*.
- Patterns in Media:
  - uniform sky, area, stripes in pictures.
  - Fixed Background in movies, object in translation.
  - Repetitive (rythmic or not) patterns in music.

ASCII Files

- American Standard Code for Information Interchange
- Fixed-width encoding
- Encode 128 characters using 8 bits
  Extended ASCII has additional 128 non-standard characters
- Encoding and decoding done by lookup tables
  - k bits indexes 2^k items
  - $\lceil \lg n \rceil$ bits for n items.

Lempel-Ziv

- Fixed-width encoding using k bits
  - Store a dictionary of 2^k entries
  - $k = 12$ is typical
- First 128 (or 256) entries are single ASCII characters
  - Example: $\ldots$, $(A, 65)$, $(B, 66)$,$\ldots$, $(a, 97)$,$\ldots$
- Remaining entries involve multiple characters
- Must ensure both encoder and decoder can build an identical dictionary

Encoding Algorithm

LZ-Encode(*input*)

Initialize dictionary D with all single characters

$s \leftarrow$ *first char of input*

$n \leftarrow \textsc{Code}(\ s\ )$

Output n

while *input has more chars* **do**

$t \leftarrow s$

$s \leftarrow$ *longest prefix from input in D*

$n \leftarrow \textsc{Code}(\ s\ )$

Output n

$c \leftarrow$ *first character of s*

Insert tc into D with next code number

end while

Example

- **Input**: `YO!·YOU!·YOUR·YOYO!`
- We will initialize with a 128 character dictionary:
- '·' visually represents the space character

Code	String
32	·
33	!
79	O
82	R
85	U
89	Y

Code	String
128	
129	
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	

Trace

Input: `YO!·YOU!·YOUR·YOYO!`

t	s	n	c	tc
	Y	89		

Trace (Filled in)

t	*s*	*n*	*c*	*tc*
	Y	89		
Y	O	79	O	YO
O	!	33	!	o!
!	.	32	.	!.
.	YO	128	Y	.Y
YO	U	85	U	YOU
U	!.	130	!	U!
!.	YOU	132	Y	!.Y
YOU	R	82	R	YOUR
R	.Y	131	.	R.
.Y	O	79	O	.YO
O	YO	128	Y	OY
YO	!	33	!	YO!

Y	O	!	.	YO	U	!.	YOU	R	.Y	O	YO	!
89	79	33	32	128	85	130	132	82	131	79	128	33

Dictionary

The dictionary then looks like this:

Code	String
32	·
33	!
79	O
82	R
85	U
89	Y

Code	String
128	YO
129	O!
130	!.
131	.Y
132	YOU
133	U!
134	YOUR
135	R.
136	.YO
137	OY
138	YO!
139	

Question: what do you do when the table is full?

Duplicates

- **Input**: `aaaaaaaaa`

Code	String
97	a
128	
129	
130	
131	

t	*s*	*n*	*c*	*tc*

- Convention: We do not prevent duplicate strings from being inserted into dictionary
- Convention: Encoder uses the highest numbered code if duplicates inserted

Duplicates (filled in)

- **Input**: `aaaaaaaaa`

Code	**String**
97	a
128	aa
129	aa
130	aaa
131	aaa

t	s	n	c	tc
	a	97		
a	a	97	a	aa
a	aa	128	a	aa
aa	aa	129	a	aaa
aa	aaa	130	a	aaa
aaa				

- **Output**:

a	a	aa	aa	aaa
97	97	128	129	130

Decoding Algorithm

LZ-Decode input An input stream of k bit codes An output stream of ASCII characters

Initialize dictionary D with all single characters
$n \leftarrow$ *first k bits of input*
$s \leftarrow$ DECODE(n)
Output s
while *input has more codes* **do**
  $t \leftarrow s$
  $n \leftarrow$ *next k bits of input*
  $s \leftarrow$ DECODE(n)
  Output s
  $c \leftarrow$ *first character of s*
  Insert tc into D with next code number
end while

Example

- **Input**: `97 97 128 129 130`

Code	**String**
97	`a`
128	
129	
130	
131	

t	n	s	c	tc

- **Output**:

Example (filled in)

- **Input**: `97 97 128 129 130`

Code	String
97	a
128	aa
129	aa
130	aaa
131	aaa

t	n	s	c	tc
	97	a		
a	97	a	a	aa
a	128	aa	a	aa
aa	129	aa	a	aaa
aa	130	aaa	a	aaa
aaa				

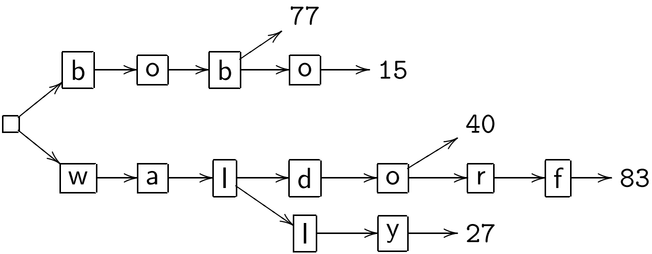
- **Output**:

  a a aa aa aaa

Outline

Tries

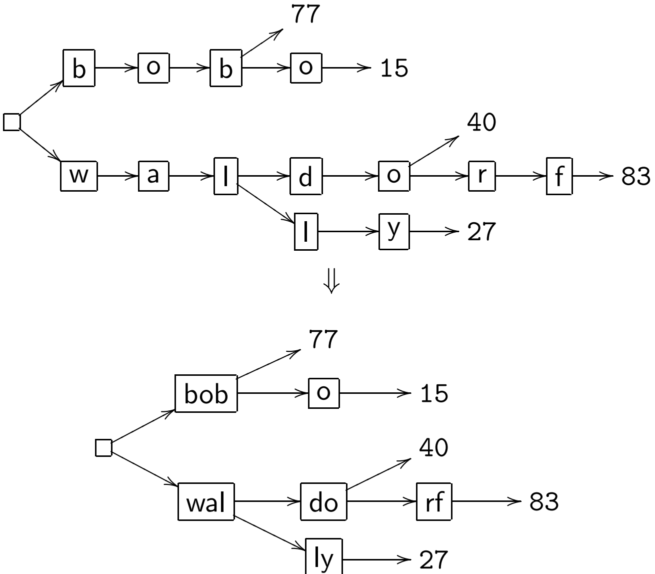
- From re-**trie**-val
- A multi-dimensional, digital search tree
- Use individual letters of key to organize and search
- Data stored in leaves
- { (bob,77), (bobo,15), (waldo,40), (waldorf,83), (wally,27) }



- Runtime independent of number of strings in dictionary!

Patricia Tries

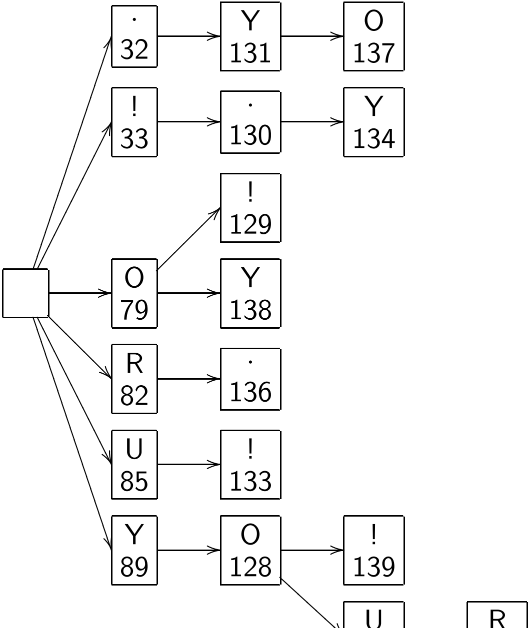
- Practical Algorithm To Retrieve Info Coded In Alphanumeric
- Compress long common subsequences into one node



Data Structures

- Encoding
  - Need to match the longest prefix
  - Build a trie, and advance as far as possible
- Decoding
  - Need to find a string associated to a code
  - An array indexed from $0..2^k - 1$ is enough
  - Decoder does not need a trie

LZ-Encode Trie



Details

- What if dictionary fills?
  - Simplest:
    restart from scratch.
  - Most complicated:
    count which codes are usefull and keep those.
  - Heuristics:
    - change code size k dynamically (complicated but done).
    - keep only the Least Recently Used codes.
- Can we eliminate duplicates in D?
  Yes, but it necessitates more work (especially in the decoding), which might not be worth the effort.

Outline

Summary of Compression

- Huffman – typically reduce up to 50% of original
- Lempel-Ziv – typically reduce up to 40% of original
- `pack` – Huffman
- `compact` – Adaptive Huffman
- `gzip`, `compress` – Lempel-Ziv
- We saw several trie data-structures, among which PATRICIA tries, which can be used for other things.

Corollary

Huffman is the best possible encoding algorithm when probabilities are known and independent.

Reading Materials

	Topic	GT	CLRS
	Lempel-Ziv	429–432	
	Huffman	440–442	385–390

- GT = Algorithm Design, by Goodrich & Tamassia
- CLRS = Introduction to Algorithms, by Cormen, Leisersen, Rivest & Stein