COMP261 Lecture 20

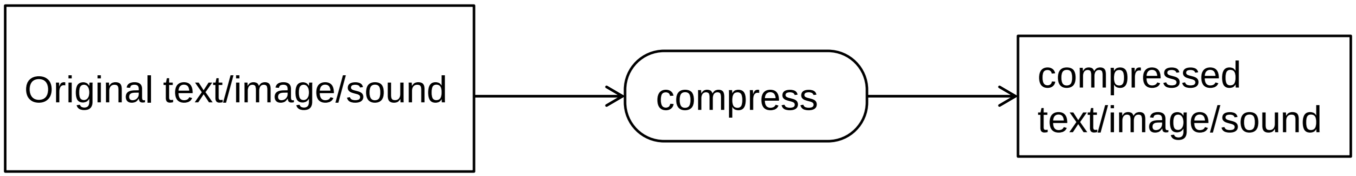
Lindsay Groves

Data Compression 2

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Data/Text Compression

- Reducing the memory required to store some information.



- Huffman coding minimises the number of bits for each symbol.

- Can we do better by looking at repeated sequences of symbols, rather than at individual symbols?

Run Length Encoding

- If data contains lots of runs of repeated symbols, it may be efficient to store as (count, symbol) pairs.

- E.g. could use two bytes for each character, one giving the count (up to 256).
  aaabbaaaaaacaa $\rightarrow$ 3a2b6a1c2a

- Or, use 6 bits to store black and white image data, one bit for the repeated bit and 5 bits for the count.
  11111111000000111111111111
  $\rightarrow$ 010001 001100 011001

Run Length Encoding

- Clearly a trade-off in the number of bit used for counts v. typical run length.
- Good for some forms of data – no good for others.
- Can we reduce redundancy when few but long runs?
- Better still …

Lempel-Ziv

- Lossless compression.
- LZ77 = simple compression, using repeated patterns
  - basis for many later, more sophisticated compression schemes.

- Key idea:
  - If you find a repeated pattern, replace the later occurrence by a link to the first:

```
a contrived text containing riveting contrasting
```

```
a contrived text
```

(Note: This ignores patterns of length 1 – they are included later.)

Lempel-Ziv

- How can we distinguish pointers from ordinary characters?
- Store text as triples, either:
  - [offset,length,c] if there is a repeated pattern with given length and offset, and c is the next input symbol, or
  - [0,0,c] if no repeated pattern here, c as before.
- To limit size of offset and length, we:
  - limit the size of the window to left of current position in which we look for a match, and
  - limit the distance ahead we look in the input for a match.

Lempel-Ziv Example

`a contrived text containing riveting contrasting …`

➔

[0,0,a] [0,0,] [0,0,c] [0,0,o] [0,0,n] [0,0,t] [0,0,r] [0,0,i] [0,0,v] [0,0,e] [0,0,d]
a ‘ ’ c o n t r i v e d

[10,1,t] [4,1,x] [3,1,] [15,4,a] [15,1,n] [2,2,g] [11,1,r] [22,3,t] [9,4,c] [35,4,a]
‘ ’ t e x t ‘ ’ cont a i n in g ‘ ’ r ive t ‘ing ’ c ontr a

[0,0,s] [12,5,t]
s ting ‘ ’

(Note how including length 1 matches changes the coding.)

This takes 69 bytes to store 48 characters – assuming offset, length and character each take one byte. Would improve with longer text.

Lempel-Ziv 77

- **skljsadf lkjhwep oury d dmsmesjkh fjdhfjdfjdpppdjkhf sdjkh fjdhfjds fjksdh kjjjfiuiwe dsd fdsf sdsa**

- Outputs a string of tuples:
  - [offset, length, nextCharacter]   or [0,0,character]
- Moves a cursor through the text one character at a time
  - cursor points at the next character to be encoded.
- Drags a "sliding window" behind the cursor.
  - searches for matches only in this sliding window
- Expands a lookahead buffer from the cursor
  - this is the string it tries to match in the sliding window.
- Searches for a match for the longest possible lookahead
  - stops expanding when there isn't a match
- Insert triple of match point, length, and next character

Lempel-Ziv 77 – high level

Algorithm

cursor ← 0; windowSize ← 100 // some suitable size

while cursor < text.length-1:

look for longest prefix of text[cursor .. text.length-1]

occurring in text[max(cursor-windowSize,0) .. cursor-1]

if found, add [offset,length,text[cursor+length]] to output

else add [0,0, text[cursor]] to output

advance cursor by length+1

We can use various approaches to find the longest matching substring:

- Brute force: Look for longest match at each position in window
- KMP, Boyer Moore
- Some form of trie?

See *Longest-match String Searching for Ziv-Lempel Compression*, Bell and Kulp, Software-Pratice and Experience, 1993.

Lempel-Ziv 77 – coding, a first attempt

```
cursor ← 0
windowSize ← 100 // some suitable size
while cursor < text.size
    length ← 0
    prevMatch ← 0
    loop
        match ← stringMatch( text[cursor.. cursor+length],
                    text[(cursor<windowSize)?0:cursor-windowSize .. cursor-1] )
        if match succeeded then
            prevMatch ← match
            length ← length + 1
        else
            output( [suitable value for prevMatch, length, text[cursor+length ]])
            cursor ← cursor + length + 1
            break
```

Cursor – WindowSize should never point before 0,

cursor+length mustn't go past end of text

- This looks for an occurrence of text[cursor..cursor+length] in text[start..cursor-1], for increasing values of length, until none is found, then outputs a triple.
- This is wasteful – we know there is no match before prevMatch, so there's no point looking there again! Could we improve by starting from prevMatch?
- Or find longest match starting at each position in window and record longest?

Decompression

`a contrived text containing riveting contrasting t`

➔

[0,0,a][0,0,][0,0,c][0,0,o][0,0,n][0,0,t][0,0,r][0,0,i][0,0,v][0,0,e][0,0,d][10,1,t]
[4,1,x][3,1,][15,4,a][15,1,n][2,2,g][11,1,r][22,3,t][9,4,c][35,4,a][0,0,s][12,5,t]

- Decode each tuple in turn:

cursor $\leftarrow$ 0

for each tuple

[0, 0, *ch*] : output[cursor++] $\leftarrow$ *ch*

[*offset*, *length, ch*] :

for j = 0 to length-1

output [cursor++] $\leftarrow$ output[cursor-*offset*]

output[cursor++] $\leftarrow$ *ch*

Lempel Ziv

- Encoding is expensive, decoding is cheap
- Many improvements/variants have been proposed
  - See Wikipedia and other online summaries
- E.g.: Use two types out output value:
  - (offset, length) pair for repeated sequence,
  - character for non-repeat
  - How can we distinguish them?
- Can be used in conjunction with Huffman coding.