Course Notes for

CS 1501
Algorithm Implementation

By

John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- What is Huffman missing?
  - Although OPTIMAL for single character (word) compression, Huffman does not take into account patterns / repeated sequences in a file
  - Ex: A file with 1000 As followed by 1000 Bs, etc. for every ASCII character will not compress AT ALL with Huffman
    - Yet it seems like this file should be compressable
    - We can use run-length encoding in this case (see text)
      - However run-length encoding is very specific and not generally effective for most files (since they do not typically have long runs of each character)

Huffman Shortcomings

- Huffman is also limited in the theoretical amount that it can compress
  - If data is taken 1 byte (8 bits) at a time the best possible compression ratio it can achieve is 8/1 (if we use a single bit for all produced codewords)
    - Ex: We have a file containing only one character
  - We could instead take 2 bytes at a time to increase the maximum compression ratio to 16/1
    - However, this would also increase the "alphabet" size to 64K and would greatly increase the size of the tree
      - > At the bottom we would have 64K initial nodes
      - > Tree info in the file would also be larger

- Idea of LZW:
  - Instead of using variable length codewords to encode single characters …
  - Use block codewords to to encode groups of characters

Huffman Compression:
1 char per codeword
variable length codewords

D	A	E	B	F
1001	00	101	11	1000

LZW Compression:
Multiple chars per codeword
Fixed length codewords

2 char	4 char	8 char	5 char
k bits	k bits	k bits	k bits

LZW Compression

- The more characters that can be represented by a single codeword, the better the compression
  - Ex: Consider the word "the" – 24 bits using ASCII
    - If we can encode the entire word in one 12 bit codeword, we have cut its size in half
    - If we can encode a longer string in a single codeword we can get even more compression
    - So our goal would be to encode as many characters as possible with a single codeword
- But we are not psychic and we don’t know that “the” or other longer strings are actually present in the original file
- So how do we assign a single codeword to a long string of characters?

LZW Compression

- We build patterns up gradually
  - Start with a single character for each codeword
  - Add new codewords for pairs of characters as they are found
  - Once we have pairs, look to increase to triples
    - Continue building longer strings in this way
- Following this procedure and if the patterns repeat we can eventually build up very long strings, encoded in single codewords
- Let's see how this will work in more detail

- LZW Compression Algorithm:
  - Initialize a dictionary to single character strings
    - Using their ASCII codes as codewords
      - This will allow us to:
        - Look up a string and return its codeword
        - Ex: Look up 'A' it will return 65
  - while not at end of input file
    - match longest prefix from file in dictionary
    - output codeword returned for that prefix
    - add longest prefix + next character to dictionary using next (new) codeword
  - Let's look at a simple (partial) trace on next slide
    - See also lzw.txt

LZW Compression

SEE SPOT RUN. RUN SPOT RUN. RUN RUN SPOT. RUN RUN RUN.

STEP	MATCH	OUTPUT	ADD to Dict
1	'S'	83	'SE' (256)
2	'E'	69	'EE' (257)
3	'E'	69	'E ' (258)
4	' '	32	' S' (259)
5	'S'	83	'SP' (260)
6	'P'	80	'PO' (261)
7	'O'	79	'OT' (262)
8	'T'	84	'T ' (263)
9	' '	32	' R' (264)
10	'R'	82	'RU' (265)
11	'U'	85	'UN' (266)
12	'N'	78	'N.' (267)
13	'.'	46	'. ' (268)
14	' R'	264	' RU' (269)

now we are starting to compress…

output to file

str	code
…	
'A'	65
…	
'ÿ'	255
'SE'	256
'EE'	257
'E '	258
' S'	259
'SP'	260
'PO'	261
'OT'	262
'T '	263
' R'	264
'RU'	265
'UN'	266
'N.'	267
'. '	268
' RU'	269

- LZW Decompression Algorithm
  - Initialize a dictionary to same data as before
    - But now we switch the keys and values
      - We look up a codeword and return its string
      - Ex: Look up 65 it will return 'A'
  - while not at end of input file
    - read next codeword from file
    - look up codeword in dictionary and output corresponding string
    - update dictionary with new codeword, string pair
      - See more detail about this in future slides
  - See lzw2.txt and next slide

output from compression

LZW Decompression

```
83 69 69 32 83 80 79 84 32 82 85 78 46 264
```

STEP	CODE	OUTPUT	ADD to Dict
1	83	'S'	---
2	69	'E'	'SE' (256)
3	69	'E'	'EE' (257)
4	32	' '	'E ' (258)
5	83	'S'	' S' (259)
6	80	'P'	'SP' (260)
7	79	'O'	'PO' (261)
8	84	'T'	'OT' (262)
9	32	' '	'T ' (263)
10	82	'R'	' R' (264)
11	85	'U'	'RU' (265)
12	78	'N'	'UN' (266)
13	46	'.'	'N.' (267)
14	264	' R'	'. ' (269)

output to file

Cannot add to dictionary yet…why?

str	code
…	
'A'	65
…	
'ÿ'	255
'SE'	256
'EE'	257
'E '	258
' S'	259
'SP'	260
'PO'	261
'OT'	262
'T '	263
' R'	264
'RU'	265
'UN'	266
'N.'	267
'. '	2[illegible]8

- Why / how does it work?
  - The compression and decompression algorithms are both building the EXACT SAME (codeword, string) dictionary as they proceed
    - Compression stores them as (string, codeword)
      - During compression, strings are looked up and codewords are returned
    - Decompression stores them as (codeword, string)
      - During decompression, codewords are looked up and strings are returned
    - As long as both follow the same steps, the compressed file does not need to store any extra information about the code

- This is an adaptive algorithm
  - The compression adapts to the patterns as they are seen, and the decompression does the same
  - The same characters in different places in the original file may map to different codewords
- However, as we discussed, the decompression algorithm is one step "behind" the compression algorithm in building the dictionary
  - In most situations this is not a problem
  - However, if, during compression, the (pattern, codeword) that was just added to the dictionary is immediately used in the next step, the decompression algorithm will not yet know the codeword

- What exactly is this case and how can it occur?
  - Consider the following text file:

AAAAAAAAAAAAAAA…

Step	Match	Output	Add
1	'A'	65	('AA',256)
2	'AA'	256	('AAA',257)
3	'AAA'	257	('AAAA',258)
4	'AAAA'	258	('AAAAA',259)

  - So why is this a problem?
  - Let's look at decompress to see…

- Output file from compress:

65 256 257 258

Step	Code	Output	Add
1	65	'A'	--
2	256	????	

- Codeword 256 is not yet in the dictionary!
  - How can we decode it?
- Luckily this special case can be recognized and handled relatively easily

IF (codeword is not found in dictionary)

THEN output:

  - Match from previous step + first char of match from previous step

LZW Special Case

- Idea: The only way a codeword would NOT be found is if it was the codeword added at the step just before this one
  - Thus the match would be the match from that step plus one more character, which must be the first char from that previous match

65 256 257 258

Step	Code	Output	Add
1	65	'A'	--
2	256	'A'+'A'	('AA',256)
3	257	'AA'+'A'	('AAA',257)
4	258	'AAA'+'A'	('AAAA',258)

  - See lzw3.txt for another example

1) How to represent / manage the dictionary
2) How many bits to use for the codewords
3) What to do when / if we run out of codewords
4) How to do I/O with fractions of bytes
   - This issue applies to Huffman and other compression algorithms as well

LZW Implementation Issues

1) How to represent / manage the dictionary

- For compress we need a symbol table with strings for keys and ints (Integer) for values
- What operations do we need?
  - Insert and Lookup
- For file with M characters, we must do M Lookups
  - Num. of Inserts depends on how long our patterns get
  - Thus we want these to be VERY FAST
    - – Sorted Array takes much too long for Inserts
    - – BST would be ok, but even lgM per Lookup is probably more time than we want to spend
      - > Would yield total of Theta(MlgM) total time
    - – Do we have any better options?

- Two most likely candidates:
  - Trie or Hash Table
  - Both allow lookups in time proportional to string length, independent of the number of strings
  - Since the new keys are all prefixes already in the tree, and plus one additional character, and since we are searching for prefixes
    - > The Trie would be a great choice
    - > However, memory issues would make us prefer the DLB
  - Text LZW.java uses a ternary search tree (TST). This is not as fast as a trie but uses less memory

LZW Implementation Issues

- What about decompress?
  - Here the idea is easier
    - Now codewords are the key values, and the strings are returned
    - We can simply use the codeword values to index an array of strings
      - Gives us constant time lookup of the codewords
    - See LZW.java

2) How many bits to use for the codewords?

- Fewer bits:
  - Smaller codewords, giving compression EARLIER in the process
  - Fewer available codewords, limiting the compression LATER in the process
- More bits:
  - Larger codewords, delaying actual compression until longer patterns are found
  - More available codewords, allowing for greater compression LATER in the process
- Ex: Consider 10 bits vs. 16 bits

LZW Implementation Issues

- 10 bits
  - Before any patterns are found we will increase by only 2 bits over ASCII
  - Patterns of even 2 characters will give decent compression
  - Only $2^{10} = 1024$ total codewords available
- 16 bits
  - Before any patterns are found will double size of ASCII (16 vs. 8)
  - We have $2^{16} = 64K$ total codewords – much more potential to find longer patterns

LZW Implementation Issues

- Can we get the "best of both worlds"?
  - We'd like to use fewer bits earlier in the process, to get compression sooner
  - We'd like to use more bits later in the process, to get greater compression later in the file
  - In fact this is exactly what the Unix compress algorithm does
    - It starts out using 9 bits for codewords, adding an extra bit when all codewords for previous size are used. Ex:
      - 9 bits for codewords 0-511
      - 10 bits for codewords 512-1023
      - 11 bits for codewords 1024-2047
      - etc
    - Decompress does the same so it works!

3) What to do when / if we run out of codewords

- If we use a block code of a specific size, we have a finite number of codewords that we can represent
  - Even the "compress" version would eventually stop adding bits due to I/O issues (we will discuss next)
- When all codewords have been used, what do we do?

LZW Implementation Issues

- Two primary options, each with advantages and disadvantages:
  - Keep compressing as before, but simply stop adding new entries to the dictionary
    - > Adv: Maintains long patterns that have already been built up
    - > Disadv: If file content changes (with new patterns) those will not be compressed effectively
  - Throw out entire dictionary, then start again with the single characters
    - > Adv: Allows new patterns to be compressed
    - > Disadv: Until new patterns are built and added to the dictionary, compression is minimal
  - We could try to get the "best of both worlds"
    - > Discuss

4) How to do I/O with fractions of bytes

- Unless we pick an exact multiple of a byte for our codeword size (8, 16, 24, 32 bits) we will need to input and output fractions of bytes
- We will not actually input / output fractions of bytes
  - Rather we will keep a buffer and read / write exact numbers of bytes, processing the necessary bits from the buffer
  - This involves some bit operations to be done
    - Shifting, bitwise OR, etc.
- See BinaryStdIn.java and BinaryStdOut.java