Lossless Compression tutorial for rahtab

- Structure
- Tables
- Bits
- Data methods

This document follows ASCII, and 8-bit per byte implementation.

Section 1: Structure

The main part of compression, is getting in the mindset of bits, rather than bytes. Most(if not all) compression algorithms, implement tables based on how much data is sent to the algo. Commonly the size of an encrypted/letter piece is achieved by logarithms, a common implementation is:

$d = \lfloor \lg n_{unique} \rfloor$ ($\lfloor \rfloor$ represents the floor() function, or truncation of the decimal, also lg is $\log_2$ simplified).

Where d is the size of data, and n_{unique} are the number of unique bytes.

Ex: "cheese"

First of all, get the 'unique' letters: 'cehs', now just run it through.

$2 = \lfloor \lg 4 \rfloor$

With this, you know that you only need two bits to represent four different pieces of data.

00
01
10
11

} the four pieces of data.

Now to assign values to our example, 00=c, 01=h, 10=e, 11=s,

Now "cheese" can be turned into 000110101110, which is two bytes, which is a 66.6% reduction.

The next key part to compression is storing the data in a table, in layman's terms "how do I know what the data stands for?"

As the table is only repeated once, it is a very minute part of a file size in large files, but in small files, it could be almost half the files composition. A common form of a table is to simply state the letter, then the value of what describes it. This is a poor way of making a table, since every unique byte is pumped into the table.

For my compression algo, I used offsets from a value.

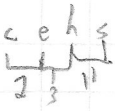
Lets take the example, and store the data in my table format.

"ceh"

First, you need to state the initial byte fully literally,

c

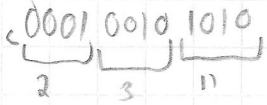
Then from there, map the offsets of each letter.



Then you take the largest offset(11), and get how many bits you need to 'describe' it.

$y = \lfloor \lg 11 \rfloor + 1$

Now you map the offsets:



Which is 3 bytes, compared to 5 for putting down each byte. As you may have noticed the offsets are written with a decrementation of one, Since an offset of zero will not be encountered, 0 is an offset of one, and not zero.

Step 3: The header

The header is a key part of any file, which contains data on how to read the data. As the table does too, this just repeats once too, meaning that it may be a small or large part of the file. My header is 6 bits for my algo. It's contents are the filler bits, and the size of each table piece.

Lets fully implement my algo.

Step 1: Sort the data

"cheese" → "cehs"

Step 2: Figure out the offsets

c e h s
2 3 11

Step 3: find the largest offset, and store it in a header

$\lceil \lg 11 + 1 \rceil$

011 cc

011 cccc cccccc00 01 00 10 10 00 10 01 011 10 10 00 2 fill

Step 4: Find the filler amount.

$8 - (6 + \lfloor \lg 11 \rfloor + 1 \cdot (4-1) + \lfloor \lg 4 \rfloor \cdot 6) \% 8$

$8 - (6 + \lfloor \lg b_{offset} + 1 \rfloor \cdot (b_{unique} - 1) + \lfloor \lg b_{unique} \rfloor \cdot b_{string}) \% 8$

which simplifies to

$8 - 30 \% 8 = 2$

011 010

you can have zero bytes of filler here, so store the value exactly.

Step 5: Throw in the table

011 010 01 [4 00 1] 00 01 00 10 10

c in ASCII

Step 5: Map the data

011 010 01 100 01 100 01 10 00 10 01 011 10 10 00

c h e e s e your 2 bits of fill

Step 6: Add the highest byte at the start

s (011 010 01 100 01 100 01 10 00 10 01 011 10 10 00) (the s is needed so the reader knows when the table hits the end)

The result yields 6 byte, so there is no compression present, but since it is such a small word, the table and header add up.

To find out the total amount of bytes this will yield!

$$\lceil (22 + \lfloor \lg b_{offset} + 1 \rfloor \cdot (b_{unique} - 1) + \lfloor \lg b_{unique} \rfloor \cdot b_{string}) / 8 \rceil$$

↑ largest offset byte (b_{offset})

↑ length of string, 6 = "cheese" (b_{string})

cab
↓
abc

c

c(000xxx01)(10000100)(0100001d

Section 2: Types of data compression

1. Run-length encoding (RLE)

It is comprised of numbering duplicate bytes.

RLE is a very common, easy, and quick data compression method. This is common in image based files when there are repeats of bytes.

Consider the string:

"aaaabbbccccdd"

Run-length encoded:

'4a 3b 4cdd' Challenge Question: Why is it 'dd' and not '2d'?

Tip: Using ascii allows to 'number' bytes up to 258 with a single byte.
↑
Because 00000000 should be considered 3 since 0, 1, and 2 are not used.

Conflicts:

"4444aaabbbcc" → "44 3a 3bcc" ← uh oh

A fix to this conflict is to always have only one 'number' followed by one letter, but this too can be a conflict.

"aaabaaaa" → "3ab4a"

↑ 3 a's
↑ b amount of 4's?
↑ a amount of what?

This is easily read by humans, but computers not so much.

There are 2 different flawless ways to implement RLE.

Version I: Terminators

"aaabbaccc" → "|3a|2b|a|3c"

Version II: ~~Label~~ Number all:

"aaabbaccc" → "3a2b1a3c"

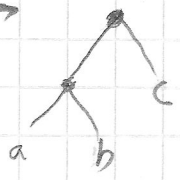
These 2 implementations can ~~change~~ either be take turns being the best compression, depending on the initial string

RLE is commonly implemented, then layered over with a compression algorithm, or vice versa. It is also used in most(if not all) image Filetypes, since colors like white & black are easily compressed.

2. ~~Table based Implementations~~. Trees

A common way of creating new character codes is by using trees. Trees are split into 2 nodes, per node. The 'first' node is 0, and the second is '1'

Basic tree:

"abccc" → 

→ 0001111

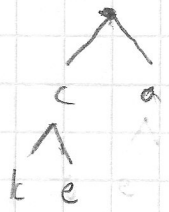
Now a has become: 00
b: 01
c: 1

Most algorithms also use a priority type tree, where the commonly used letters will have the shortest character codes.

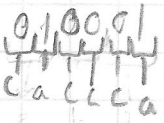
Hint: A letter ALWAYS ends a node, otherwise you could not know where the 'data' stops.

Heres an example:

"Cake" →



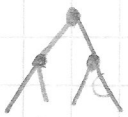
c : 0
a : 1
k : 0 0
e : 0 1



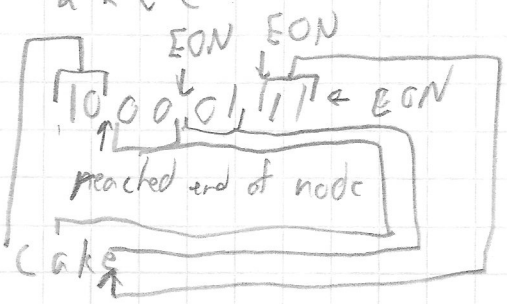
Failure!

Correct example:

"cake" →



a : 0 0
k : 0 1
c : 1 0
e : 1 1



New tk nodes are all unique, and there is no 'half-node.'

Priority Trees:

An easy way of a priority tree, is to sort, then RLE a string. Then put it in a table in order.

"cheese"

↓

"eeehs"

↓

"e3ehs"

Now sort it by frequency

"3echs"

↓

Then remove RLE

↓

"echs"

Then put it in a tree in left to right, top down order.

Now:

e : 0
c : 1
h : 00
s : 01

→ 10000010

1 byte.

Since the types of tables vary, so does the way you 'map/describe' the tree in the compressed file.

3: Tableless compression

Some methods require absolutly no table in the file, the file is read and the table is built simultaneously. A common method of this is LZW (ZIP files)

D: I haven't found any documentation to find this.

Lossless Data Compression

Created by: Falkman (aka. Gamozo) for rohitab.com

This document uses ASCII, and 8-bit per byte implementation.
All binary numbers are split into 8-bit sectors.
Last Revised: April 27, 2009

Section 1: Structure

The main part of compression, is getting in the mindset of bits, rather than bytes. Most (if not all) compression algorithms create new character code tables based on how much data is sent to the algorithm. Commonly the size of a new table letter is achieved by logarithms. Here is a common example of this:

$d = \underline{\log_2 n_{unique}}$ (The underline represents the C floor() function, which truncates the decimal)

Where d is the size of the data (in bits), and n_{unique} is the number of 'unique' (number of bytes total, not including repeated bytes) bytes.

Ex: "cheese"

First of all, remove duplicates (getting just the unique letters).

"cehs"

Then use the equation to find out the size necessary for each letter.

$2 = \underline{\log_2 4}$

This means that it takes two bits to represent each letter.

00 = c
01 = e
10 = h
11 = s

Now you can use these codes to rebuild the string.

01100101 1101

Which is only two bytes, which is a 66.6% reduction of "cheese".

The next key part of compression is storing the new character set, so when it is decrypted, the decompresser knows what the bit chunks represent. As the new table is only repeated once in the output, it is a very minute part of a file size in large files, but in small files, it could take up almost the entire file. A common form of a table is to simply state the letter, then the value of what 'describes' it. This is a poor way of making a table, since every unique byte is added into the table, normally killing

the reduction size of the compression.

For my own compression algorithm, I use offsets from an initial value, which results in a much smaller table size.

Lets take the first example, and store the data in my table format.

"cehs"

First, you need to state the initial byte, literally (as in just plunk down the letter).

c

Then, from there, map out all the offset values.

$c_2 e_3 h_{11} s$

Then you take the largest offset (in this case 11), and figure out how many bits are needed to 'describe' it.

$4 = \underline{\log_2(11+1)}$

Now you know that each letter definition must be 4 bits long, and you now place the offsets into the output string.

c00010010 1010

Which is 3 bytes, compared to 5 or more if you were to place each letter, followed by the definition. As you may have noticed, the offsets are written decremented by one, since an offset of zero will not be encountered, zero is considered to be an offset of one, and not zero.

The next part of the compression, is the header. The header is a key part of any file, it contains the information about how to read the data that is following. As the table does too, this also just repeats once, thus meaning again, that it could make a big difference in a small file. My header is only 6 bits total, it contains the amount of filler bits, and the size of each piece in the table.

Lets fully implement my compression method.

Step 1: Sort the data

"cheese" → "cehs"

Step 2: Figure out the offsets

$c_2 e_3 h_{11} s$

11

Step 3: Find the largest offset, and store it in the header

$$4 = \underline{\log_2(11+1)}$$

Output: 011
Step 4: Find the filler amount

$$b_{filler} = 8 - (6 + \underline{\log_2(b_{offset}+1)} \cdot (b_{unique} - 1) + \underline{\log_2 b_{unique}} \cdot b_{string}) \, mod \, 8$$

Where b_{filler} is the amount of filler, b_{offset} is the largest offset size, b_{unique} is the number of unique bytes, and b_{string} is the length of the entire string.

$$b_{filler} = 8 - (6 + \underline{\log_2(11+1)} \cdot (4-1) + \underline{\log_2 4} \cdot 6) \, mod \, 8$$

$$b_{filler} = 8 - 30 \, mod \, 8$$

$b_{filler} = 2$ (This time, the value is not decremented, because it is possible to have 0 bits of filler, but not possible to have 8 bits.)

Output: 011010

Step 5: Throw in the table.

← Since we already created the table, there is no reason to go through the steps again.

Output: 01101001 10001100 01001010 10

Note: 01100011 was substituted for 'c'.

Step 6: Map the data

01101001 10001100 01001010 10001001 01110100

Step 7: Add the 'highest' byte at the start

s01101001 10001100 01001010 10001001 01110100 (This step is needed because the decompresser would not know when to stop creating the table, this way once the offsets add up to 's', it will know that the data follows)

The result yields 6 bytes, so there was no compression in this case, but in a larger file, containing {c, e, h, or s's}, the compression ratio would be large. With such a small word, as mentioned before, the table and header make up much of the output, so the output isn't much more compressed (if at all), then the input.

To find out the total amount of bits that this compression method will yield:

$$b_{output} = 22 + \underline{\log_2(b_{offset} + 1)} \cdot (b_{unique} - 1) + \underline{\log_2 b_{unique}} \cdot b_{string}$$

Section 2: Types of data compression

1. Run-length encoding (RLE)

   RLE is a very common, easy, and quick data compress method. It is comprised of numbering the duplicate bytes. This is very common in image based files where there are many repeats of bytes.

   Consider the string:

   "aaaabbbccccdd"

   Run-length encoded:

   "4a3b4cdd" Challenge question: Why is it 'dd' and not '2d'?

   Tip: Using ASCII values allows to 'number' bytes up to 258 with a single byte. Since 00000000 should be considered 3, since 0, 1, and 2, are not used in RLE.

   Conflicts:

   "4444aaabbbcc" → "443a3bcc" (Uh oh, 443?)

   A fix to this conflict is to always have only one 'number' followed by one letter, but this too can be a conflict.

   "aaabaaaa" → "3ab4a"

   3 a's - OK
   b amount of 4's – ERROR

   This is easily read by humans, but computers... not so much.

   There are two different flawless ways to implement RLE.

   Version I: Identifiers

   "aaabbaccc" → "!3abba!3c"

   This will always have some sort of character token, to tell the decoder that this following number, is showing how many of the next letter follows.

   Version II: Number all

   "aaabbaccc" → "3a2b1a3c"

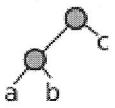
These two implementations can take turns being the 'best' compression, depending on the initial string to be compressed.

RLE is commonly implemented, then layered over with a compression algorithm, or vice versa. It is also used in most (if not all) image file types, since colors like black & white are easily compressed.

2. Trees

Another very common way of creating new character codes, is by using trees. Trees are split into two nodes, per node. The 'first' node is 0, and the second is 1.

Basic tree:

"abccc" → 

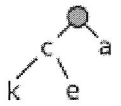
→ 0001111

Now 'a' has become: 00
'b': 01
'c': 1

Most algorithms also use a priority type tree, where the most commonly used letters will have the shortest character codes.

A letter ALWAYS ends a node, there will never be a node that passes though a different letter, otherwise you would not know when to stop reading your current byte, and move on to a new one.

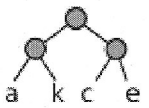
Heres an example of a fail tree:

"cake" → 

c: 0
a: 1
k: 00
e: 01

Turns into: 010001
Which is decoded to: caccca

Correct Example:

"cake" → 

a: 00
k: 01
c: 10
e: 11

Turns into: 10000111
Which is decoded to: cake

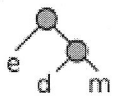
Now the nodes all end with a letter, and there are no letters 'on the way' to the node.

Priority Trees (Same as Huffman, and weighted trees):

An easy way of creating a priority tree is to sort the string, RLE the string, then sort according to frequency, remove RLE, and then store directly into the tree in left to right order.

"deemed" (Thanks to Limitz for this word)
⇩ Sort the letters.
"ddeeem"
⇩ RLE.
"2d3e1m"
⇩ Sort by frequency.
"3e2d1m"
⇩ Remove RLE.
"edm"

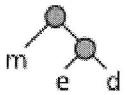
Now put it in a tree left to right, top to bottom order.



e: 0
d: 10
m: 11

Turns into: 10001101 0 (9 bits, from 48 bits)

Worst case scenario(not using priority):



m: 0
e: 10
d: 11

Turns into: 11101001 011 (11 bits, compared to 9, this gets much worse is 'larger' situations)

The last thing about tree based compression is that you need to store the tree in some sector of the output. Since all trees vary, and some trees can have more efficient ways of storing the tree than others, there is no 'generic' storing method (except for the 'letter-code' method).

3. Lempel-Ziv-Welch (LZW)

Another very common method of compression, is LZW. LZW creates its own table according to previously used bytes. Such as the word "papa". LZW would take the string, then output the 'p', followed by 'a', then it would make a new definition for "pa", since it is not already in the list of two letter combinations.

Heres how it would look compressing "ohlololol"

Lets first assign each letter with a value, since there are only 3 letters, we use 2-bit strings.

h: 00
l: 01
o: 10

Lets make a table of the encryption process:

This	*Next*	*List*
X	o	X
o	h	oh
h	l	hl
l	o	lo
o	l	ol
lo	l	lol
lo	l	(lol already in DB)
l	X	X

Now lets put substitute binary in for this.

This	*Next*	*List*
X	10	X
10	00	11
00	01	100
01	010	101
010	001	110
101	101	111
101	001	(lol already in DB)
001	X	X

Now put it in 1 line, by reading directly down the left column.

10000101 01011010 01

Now to decode it.

This	*Next*	*List*
X	o	X
o	h	oh(11)
h	l	hl(100)
l	o	lo(101)
o	lo	ol(110)
lo	lo	lol(111)
lo	l	(lol already in DB)
l	X	X

(The highlighted letter shows that you only take the first letter from the multi-wide dictionary string)

Now write it out in full, reading down the left column again.

"ohlololol"

Success!

Lets look back at what happened, after compression, we were left with the binary string:

10000101 01011010 01

which is 3 bytes, or 2.25, but you can't a fourth of a byte. Now, if we were to code each letter directly, without the dictionary, we would end up with:

01000110 01100110 01

which is the same, but if we were to add more and more lol's at the end of the string, the LZW compressed string would be much shorter. Just like any other compression method, it will normally be right around the same size of the normal string if the string is short, but the larger it gets, the larger the compression ratio becomes.

Also, notice that an newly added piece to the dictionary, cannot be used 'on the next line' right after it was added, because on the same line, that would be in the 'next' column, but it would be undefined, causing the decompresser to not be able to recover the data.