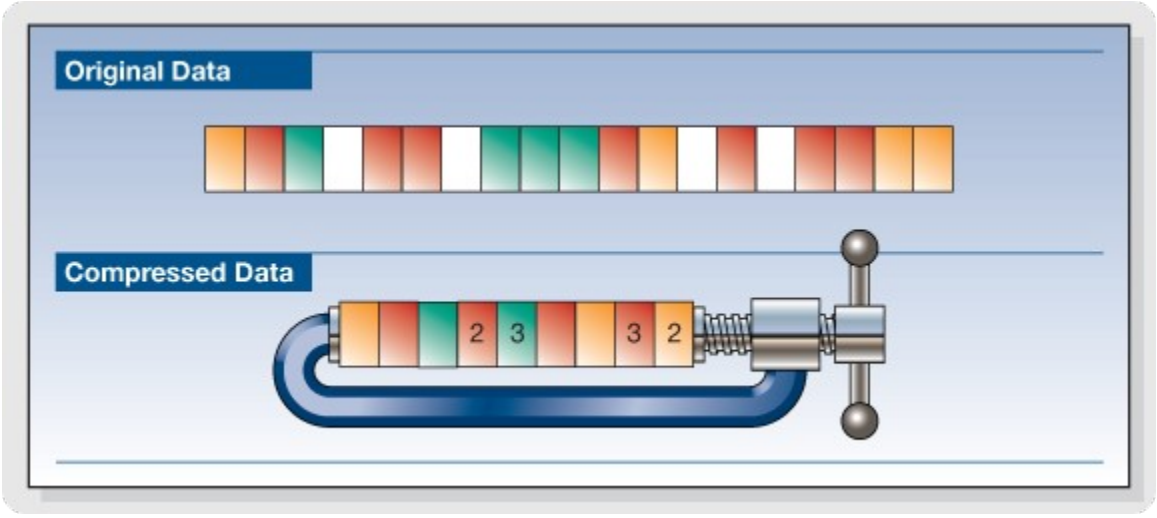
Name: Joaquim Miguel Conceição Espada

Login: con0004

Date: 31st October of 2016

Theoretical report

Title: Data Compression



Introduction to problematics:

The main objective of this class is to study the topic Data Compression and a few algorithms that allow to compress data. The problematic in this lecture is following: how can we send a file faster over a networking without relying in networking technologies (xDSL/power-lines/analogue modems).
Data compression is the answer because it makes possible for data to be transmitted quicker because it has become smaller.

Elaboration:

Data Compression

Data compression is the process of modifying, encoding or converting the bits structure of data in such a way that the file becomes smaller. Data compression enables sending a data object or file quickly over a network or the Internet and in optimizing physical storage resources.
A common data compression technique removes and replaces repetitive data elements and symbols to

reduce the data size. Data Compression can be divided in two groups: lossy and lossless.

Lossy Data Compression

Lossy file compression results in lost data and quality from the original version. Lossy compression is typically associated with image files but can also be used for audio files. Lossy compression removes data from the original file, the resulting file often takes up much less disk space than the original.
For example, a JPEG image may reduce an image's file size by more than 80%, with little noticeable effect.

Lossless Data Compression

Lossless compression reduces a file's size with no loss of quality. Lossless compression basically rewrites the data of the original file in a more efficient way. However, because no quality is lost, the resulting files are typically much larger than image and audio files compressed with lossy compression.
Lossless data compression algorithms usually exploit statistical redundancy to represent data without losing any information, so that the process is reversible. Lossless compression is possible because most real-world data exhibits statistical redundancy.

Run-length encoding (RLE)

Run-length encoding (RLE) is a very simple form of lossless data compression in which runs the data sequence in which the same data value occurs in many consecutive data elements are stored as a single data value and count, rather than as the original run.

Example 1:
AAAAABBAAA → Raw Data
5A 2B 3A → Encoded data

$$Compression_{Ratio} = \frac{10}{6} \simeq 1{,}67$$

Example 2:
ABCD → Raw data
1A1B1C1D → Encoded data

$$Compression_{Ratio} = \frac{4}{8} = 0{,}5$$

Observation : This algorithm isn't good for compression when the data has lot of different data values.

Huffman coding

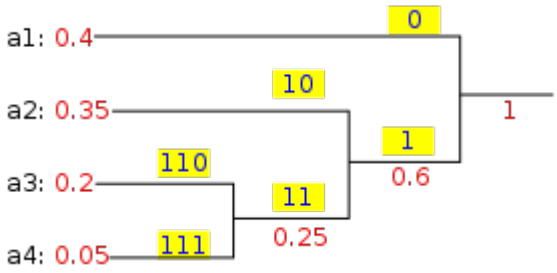
Huffman coding is a particular type of optimal prefix code that is commonly used for lossless data compression. This algorithm posses a better compression ratio than RLE and it is based in a binary three. It is often used in off-line compression.

Example:

1 – Calculate the probability of each symbol to appear

Symbol	Probability
a1	0.4
a2	0.35
a3	0.2
a4	0.05

2- Construct this binary tree. The binary tree is generated from left to right taking the two least probable symbols and putting them together to form another equivalent symbol having a probability that equals the sum of the two symbols. The process is repeated until there is just one symbol. The tree can then be read backwards, from right to left, assigning different bits to different branches.



3- Construct the encoding/decoding table

Symbol	Code
a1	0
a2	10
a3	110
a4	111

Adaptive Huffman coding

Adaptive Huffman coding (also called Dynamic Huffman coding) is an adaptive coding technique based on Huffman coding. It permits building the code as the symbols are being transmitted, having no

initial knowledge of source distribution, that allows one-pass encoding and adaptation to changing conditions in data. The benefit of one-pass procedure is that the source can be encoded in real time, though it becomes more sensitive to transmission errors, since just a single loss ruins the whole code.

Adaptive Huffman coding

Some important notations and constraints:

- **Implicit Numbering**: Nodes are numbered in increasing order by level and from left to right. i.e. nodes at bottom level will have low implicit number as compared to upper level nodes and nodes on same level are numbered in increasing order from left to right.
- **Invariant**: For each weight w, all leaves of weight w precedes all internal nodes having weight w.
- **Blocks**: Nodes of same weight and same type (i.e. either leaf node or internal node) form a Block.
- **Leader**: Highest numbered node in a block.
- **Parent Node weight**: sum of the weigths of if its two child nodes
- **Blocks**: are interlinked by increasing order of their weights.
- **A leaf block**: always precedes internal block of same weight, thus maintaining the invariant.
- ***NYT(Not Yet Transferred)****: is special node and used to represents symbols which are* not yet transferred.

Example:

Message to encode : "abb"
Binary Encoded data → 01100001 001100010 11
Binary Raw data → 01100001 01100010 01100010

$$Compression_{Ratio} = \frac{26}{21} \approx 1{,}24$$

Step 1:
Start with an empty tree.

For "a" transmit its binary code.

Step 2:

NYT spawns two child nodes: 254 and 255, both with weight 0. Increase weight for root and 255. Code for "a", associated with node 255, is 1.

For "b" transmit 0 (for NYT node) then its binary code.

Step 3:

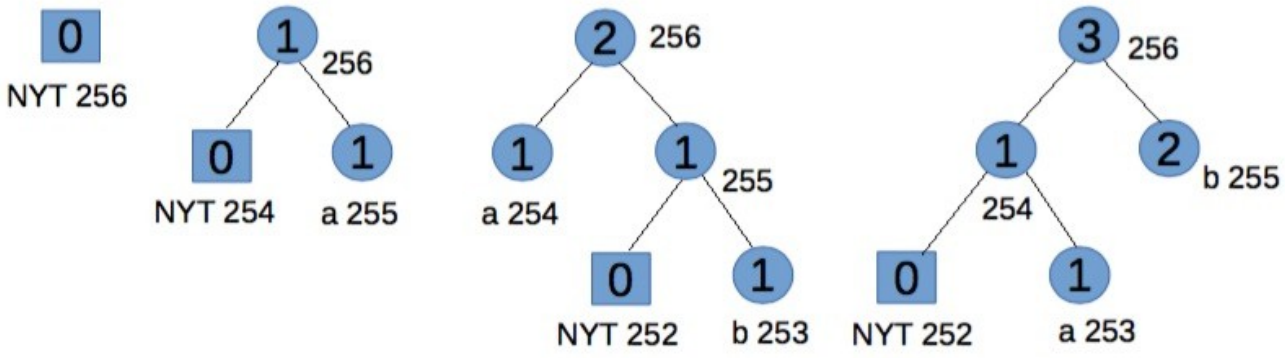
NYT spawns two child nodes: 252 for NYT and 253 for leaf node, both with weight 0. Increase weights for 253, 254, and root. To maintain Vitter's invariant that all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w, the branch starting with node 254 should be swap (in terms of symbols and weights, but not number ordering) with node 255. Code for "b" is 11.

For the second "b" transmit 11.

Step 4:

Go to leaf node 253. Notice we have two blocks with weight 1. Node 253 and 254 is one block (consisting of leaves), node 255 is another block (consisting of internal nodes). For node 253, the biggest number in its block is 254, so swap the weights and symbols of nodes 253 and 254. Now node 254 and the branch starting from node 255 satisfy the Slide And Increment condition and hence must be swap. At last increase node 255 and 256's weight.

Future code for "b" is 1, and for "a" is now 01, which reflects their frequency.



LZ77

LZ77 is a lossless data compression algorithm. LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a length-distance pair, which is equivalent to the statement "each of the next length characters is equal to the characters exactly distance characters behind it in the uncompressed stream".To spot matches, the encoder must

keep track of some amount of the most recent data. The structure in which this data is held is called a sliding window, which is why LZ77 is sometimes called sliding window compression. The encoder needs to keep this data to look for matches, and the decoder needs to keep this data to interpret the matches the encoder refers to. The larger the sliding window is, the longer back the encoder may search for creating references.

Example

Message to encode:

Message	A	B	C	B	A	B	C	A	B	C
Letter position	1	2	3	4	5	6	7	8	9	10

Step	Position	Agreement	Next char	Output
1	1	--------	A	00A
2	2	--------	B	00B
3	3	--------	C	00C
4	4	B	A	21A
5	6	BC	A	42A
6	9	BC	EOT	4EOT

LZ78

LZ78 is a lossless data compression algorithm. This algorithm is a refined version of the LZ77. LZ78 algorithms achieve compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input data stream. Each dictionary entry is of the form dictionary[...] = {index, character}, where index is the index to a previous dictionary entry, and character is appended to the string represented by dictionary[index]

Example:

Message	A	B	B	C	B	C	A	B	A
Letter position	1	2	3	4	5	6	7	8	9

Step	Position	Dictionary	Output
1	1	A	(0,A)
2	2	B	(0,B)
3	3	BC	(2,C)

4	5	BCA	(3,A)
5	8	BA	(2,A)

LZW

LZW is a lossless data compression algorithm. It is an improved implementation of the LZ78 algorithm.

LZW starts out with a dictionary of 256 characters (in the case of 8 bits) and uses those as the "standard" character set. It then reads data 8 bits at a time (e.g., 't', 'r', etc.) and encodes the data as the number that represents its index in the dictionary. Every time it comes across a new substring (say, "tr"), it adds it to the dictionary; every time it comes across a substring it has already seen, it just reads in a new character and concatenates it with the current string to get a new substring. The next time LZW revisits a substring, it will be encoded using a single number. Usually a maximum number of entries (say, 4096) is defined for the dictionary, so that the process doesn't run away with memory. Thus, the codes which are taking place of the substrings in this example are 12 bits long (2^12 = 4096). It is necessary for the codes to be longer in bits than the characters (12 vs. 8 bits), but since many frequently occurring substrings will be replaced by a single code, in the long haul, compression is achieved.

Example:
Message to encode : banana

Dictionary	
Index	Entry
0	a
1	b
2	d
3	n

Input	Current String	Seen Before	Encoded Output	New Dictionary Entry/Index
b	b	yes	nothing	none
ba	ba	no	1	ba/5
ban	an	no	1,0	an/6
bana	na	no	1,0,3	na/7
banan	an	yes	no change	none

banana	ana	no	1,0,3,6	ana/8

Conclusion:

In this class we studied several compress algorithms. There are two main types of algorithms, the lossy algorithms and lossless algorithms. In lossless algorithms there isn't any discarded data, unlike the lossy algorithms. Usually, a compressed file by a lossy algorithm will occupy less space on disk and it's transmission is faster, but the file will lose quality.

About the studied algorithms, the simplest and easiest to understand was Run-length encoding but had a major downside when the message to compress has a lot of different values the files compress size can be bigger than the not compressed one. The main difference that I spotted in Huffman codes and LZ famliy is that the Huffman code is a statistical method that often uses data structures like binary trees and the LZ family is based on a sliding windows functions and dictionary data structures,

References:

- Ing. Pavel Nevlud lectures
- http://techterms.com/definition/lossless
- https://en.wikipedia.org/wiki/Data_compression
- https://en.wikipedia.org/wiki/Run-length_encoding
- https://en.wikipedia.org/wiki/Huffman_coding
- https://en.wikipedia.org/wiki/Adaptive_Huffman_coding
- http://image.slidesharecdn.com/compressionprojectpresentation-121214155433-phpapp02/95/compression-project-presentation-4-638.jpg?cb=1355500610
- https://en.wikipedia.org/wiki/LZ77_and_LZ78#LZ77
- https://www.cs.duke.edu/csed/curious/compression/lzw.html
- https://en.wikipedia.org/wiki/Lempel–Ziv–Welch