

Advanced Algorithm

CS403

Fall 2018

Project

Name: Youssef Ali Fayed

ID: 162085

**Index:**

1. **Problem definition**
2. **Applications**
3. **Selected algorithms**
   1. **Huffman Code**
      1. **Compression**
      2. **Decompression**
      3. **Pseudo code**
      4. **Analysis**
         1. **Empirical**
         2. **Analytical**
   2. **LZW**
      1. **Compression**
      2. **Decompression**
      3. **Pseudo code**
      4. **Analysis**
         1. **Empirical**
         2. **Analytical**
   3. **Designed Code**
      1. **Compression**
      2. **Decompression**
      3. **Pseudo code**
      4. **Analysis**
         1. **Empirical**
         2. **Analytical**
4. **Discussion of the results**
5. **Problem definition**

The problem that we are talking about today is the compression of large data to preserve space which is needed in modern days so much as data becomes more and more larger, the official definition of data compression is the process of reducing the size of a data file without losing any of its properties, it is useful for saving resources like space for storing large files but the tradeoff is that it needs computational power to compress and to decompress which in many cases like video compression for example may need expensive hardware, a way to bypass this is to tradeoff the degree of compression and the amount of distortion introduced.

There are two types of compression which are lossless compression and Lossy compression, first we talk about lossless data compression algorithms, it usually uses statistical redundancy to represent the data without losing any information, this type of compression is possible because most real-life data exhibits statistical redundancy, for example if you take an image you may have areas where the color doesn’t change for several pixels so instead for writing the “red color” over and over you can write “5 red pixels” which will save you space.

The other type is Lossy compression works in a different way so instead for coding the redundant port it will delete some of it and the reason why this works is that it depends on the fact that humans perceive the data in a different way so we are not that sensitive like a machine so for example in the example of an image which contain a zone of the same color instead of replacing the zone with the number of pixels it will delete number of these pixels which won’t affect the overall color of the image so it will seem the same for our eye but it will significantly decrease the size.

1. **Applications**

Compression is crucial to a lot of applications, the most obvious use is to store large files like storing backups and for entities that have huge data and need to store them.

Another use for compression is audio and video files, a lossless audio or video file will be so large that it won’t be practical to store it or to use it, so nearly all video and audio files are compressed in one way or another, some of these are lossless compression and some are Lossy.

One more application for compression is coding, nearly all lossless compression algorithms may be considered a coding algorithm as it produces a non-readable version of whatever compressed unless you have the code it used to compress.

1. **Selected Algorithms**

The two selected algorithms Are Huffman Coding And Lempel-Ziv compression (LZ77), where each have a compression and decompression way.

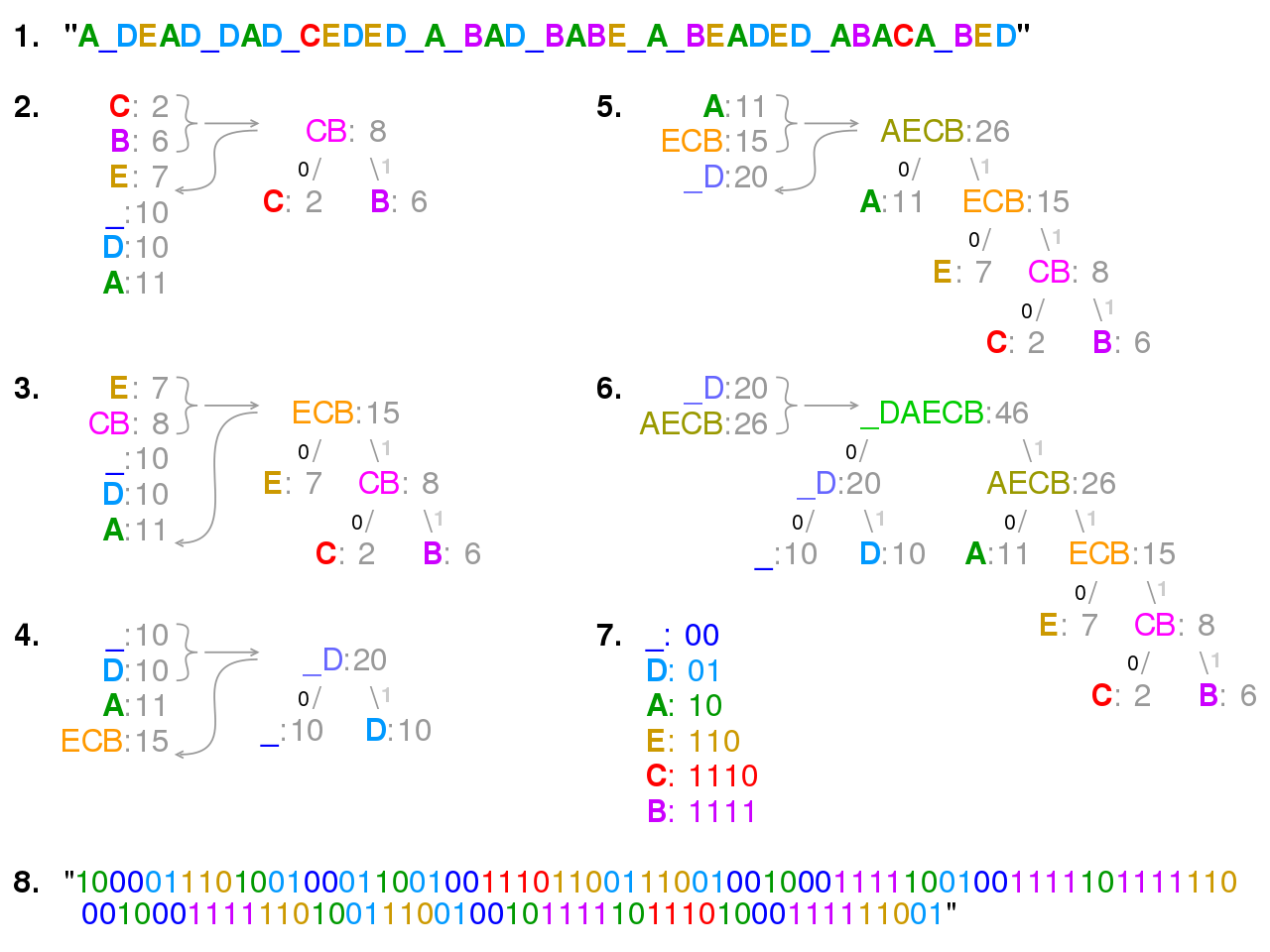
* 1. **Huffman Code**
     1. **Compression**

It compress Data very effectively for saving nearly 20% - 90% of the size depending on the characteristics of the data being compressed. It works by the frequency of the characters in the data, and the way it works is as follow, it creates a binary tree of nodes, these nodes can be stored in a regular array which size depends on the number of symbols, these nodes can be a leaf node or an internal node but at first all the nodes are leaf nodes which contains the symbol and its frequency and optionally the link to a parent node, internal nodes contain weights and link to two children node and an optional link to a parent node, and the ‘0’ represents the left child while ‘1’ represents the right child.

The algorithm starts by putting the frequency of each symbol to the tree then omitting the unused symbols from the tree because by omitting it produces the best optimized tree, then it takes the two nodes with smallest probability and creates a new internal node having these nodes as children with the weight of this new node equal to the sum of the weights of the children and so on until only one node remains

* + 1. **Decompression**

It’s a process of translating the stream of prefix codes to their original symbol by looking at the tree and reading the bits from it according to the previously constructed tree.



* + 1. **Pseudo code**

HUFFMAN.C /

1 n D jCj

2 Q D C

3 for i D 1 to n 1

4 allocate a new node ´

5 ´:left D x D EXTRACT-MIN.Q/

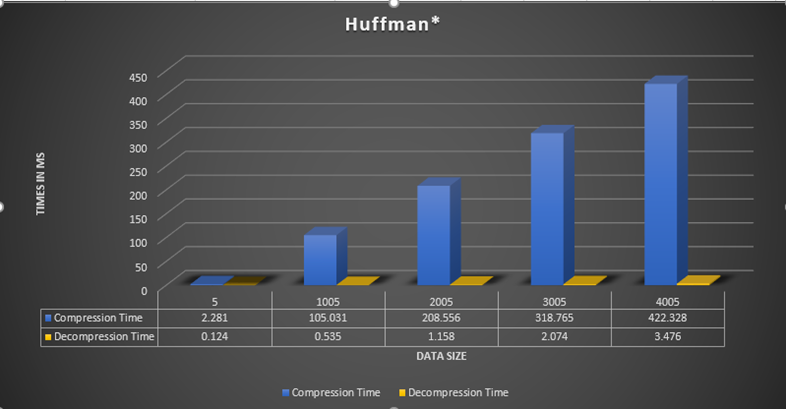
6 ´:right D y D EXTRACT-MIN.Q/

7 ´:freq D x:freq C y:freq

8 INSERT.Q; ´/

9 return EXTRACT-MIN.Q/ // return the root of the tree

* + 1. **Analysis**
       1. **Empirical**

****

* + - 1. **Analytical**

Since Huffman tree can be done by two ways so we have two analysis:

-We can use priority queue to construct the tree so in this case we know that the efficiency of the priority queue is O(log n) per insertion and the tree has n leaves and 2n-1 nodes so the algorithm will be o(nlogn).

-we can also use probability and in this case the time will be linear so it will take o(n) to construct the tree.

-but in both cases the efficiency is not as important as in compression the accuracy is more important.

* 1. **LZW**
     1. **Compression**

It replaces the repeated symbol with a reference to a single copy of data, similar to the way we described before.

* + 1. **Decompression**

By getting the number of before the symbol and putting the symbol this number of times.

* + 1. **Pseudo code**

**while** input is not empty **do**

prefix := longest prefix of input that begins in window

**if** prefix exists **then**

i := distance to start of prefix

l := length of prefix

c := char following prefix in input

**else**

i := 0

l := 0

c := first char of input

**end if**

**output** (i, l, c)

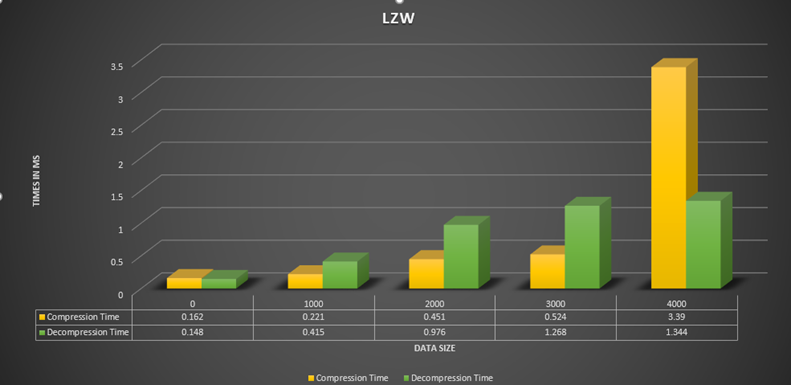
s := pop l+1 chars from front of input

discard l+1 chars from front of window

append s to back of window

**repeat**

* + 1. **Analysis**
       1. **Empirical**

****

* + - 1. **Analytical**

LZW Complexity scale linearly with the data size so the time complexity for both compression and decompression is O(n).

So We Can Safely Say That LZW is Better Than Huffman in every comparison a small or large dataset it doesn’t matter its running time is better but Huffman Produces a More Compact Compression.

* 1. **Designed Code**
     1. **Compression**

It combines both the algorithm above to produce maximum efficiency and it produces a compression that can be controlled.

It match and replace duplicate strings with pointers.(L177)

Replaces symbols with weighted symbols based on the frequency. (Huffman)

The data is broken up in ``blocks,'' and each block uses a single mode of compression. If the compressor wants to switch from non-compressed storage to compression with the trees defined by the specification, or to compression with specified Huffman trees, or to compression with a different pair of Huffman trees, the current block must be ended and a new one begun.

* + 1. **Decompression**

Its Decompression is like reverse engineering both algorithms.

* + 1. **Pseudo code**

HUFFMAN.C /

**while** input is not empty **do**

prefix := longest prefix of input that begins in window

**if** prefix exists **then**

i := distance to start of prefix

l := length of prefix

c := char following prefix in input

**else**

i := 0

l := 0

c := first char of input

**end if**

**output** (i, l, c)

s := pop l+1 chars from front of input

discard l+1 chars from front of window

append s to back of window

**repeat**

* + 1. **Analysis**
       1. **Empirical**
       2. **Analytical**

The time of both compression and decompression is O(n) despite using the Huffman tree because as mentioned before most of the time used in Huffman code was to produce the tree but in this case the tree is already made by LZW which has time of O(n) and all that’s left is for the Huffman code to finalize which add a constant to the complexity so the real time is more than that of LZW of course but the complexity is still O(n) but added the efficiency of the Huffman.

1. **Discussion of the results**

From These Results we can see that there is no best algorithm to compress because you must choose between good compression ratio and time so we have Huffman which has the best compression ratio and LZW Which has the best complexity from these three algorithms and then we have the third algorithm which is a mix of both it has a slight higher time than LZW but it has lot more compression ratio so it is like a middle ground but with that said we can see that it is not ideal for small data compression as the overhead to compress is much high of a price, also in decompression it has the worst time of the three algorithms but our main concern here is compression not decompression.