Static Vs Adaptive Huffman coding - A Pragmatic Comparison

CSCI-B 503 Algorithms Design and Analysis
Madrina Thapa

Abstract

This project aims to elucidate the difference between Static and Adaptive Huffman coding techniques for data compression and decompression of various input format files of different sizes. We have introduced an intuitive version of the Static Huffman coding algorithm which performs better than Adaptive in most of the cases. We have implemented both the Static and Adaptive Huffman coding in C++ and observed a difference between the execution time and size of the compressed file. By the end of the project, we arrived at the conclusion of when to use the Static or Adaptive technique depending on the input file structure.

Introduction

Huffman coding is one of the most popular technique for lossless data compression, it generates variable length codes which is used for encoding and decoding the input file. It is a very effective technique for compressing the data, characteristics of the data being compressed is the major factor in deciding the percentage of saving, which is typically between 20% to 90%. There have been discussions about Huffman coding and it has been proved that there are no encoding techniques which generates shorter average length code. Huffman coding takes advantage of redundancy in data, and for the most frequent symbols or characters occurring in the data, a shorter length code is generated. This in turns helps in data compression thereby reducing the space required for storing the file and time for transmitting the file. The main idea is to build a list of all the symbols occurring in the input file in decreasing order of their frequencies and construct a binary code tree using a bottom-up approach. The minimum length of each symbol is 1 bit as it encodes each symbol separately. At each point, two symbols with the least frequencies are picked from the list and are added to the tree and deleted from the list. A new entry is made in the list which symbolizes the original symbol, whose frequency is equal to the sum of the frequency of both the original symbols. Every leaf in the binary tree denotes a symbol present in the input file. The tree is supposed to be considered complete when there is just one auxiliary symbol existing in the list. The code table is then built by traversing the tree binary tree. Huffman code compress data effectively

```
HUFFMAN(C)
1 n = |C|
2 Q = C
3 for i = 1 to n−1
4    allocate a new node z
5    z.left = x = EXTRACT–MIN(Q)
6    z.right = =y = EXTRACT–MIN(Q)
7    z.freq = x.freq + y.freq
8    INSERT (Q, z)
9 Return EXTRACT–MIN(Q) //return the root of the
```

Static Huffman Coding

Static Huffman encoding involves nothing but just replacing a symbol in the input file by the Huffman code. It is also known as a prefix code which is binary in nature and symbolizes the links that connect the root node to the leaf node which represents the symbol in the input data. Decoding data compressed by the Static Huffman coding makes use of the code table built for constructing the code tree . The original data is recovered from the encoded file by parsing the code tree starting from its root and taking the left link if a 0 is encountered or right otherwise.

```
FREQUENCY(input)
1 for each character a ∈ ∑
2    do freq (a) ← 0
3 while not end of input and
  a is the next symbol
4  do freq (a) ← freq(a) + 1
5 freq(END) ← 1
```

The following example has a data which contains various symbols. Their frequencies are as follows:

Constructing the Huffman code tree for the above input data would give us the following tree and variable length code.

Table 1: Frequencies of the symbols

Symbol	Frequency
A	22
E	15
I	9
O	5
U	3

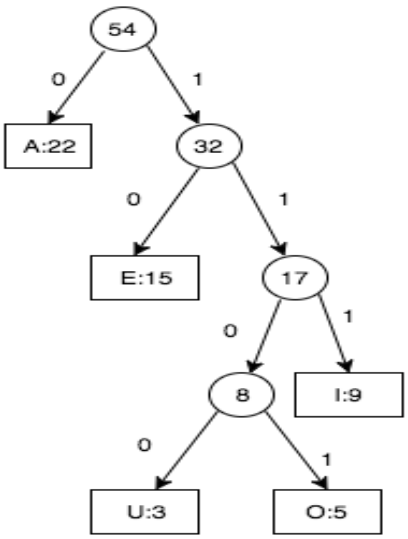


Table 2: Symbols and their codes

Symbol	Code	Code Length
A	22	0
E	15	10
I	9	111
O	5	1101
U	3	1100

Adaptive Huffman Coding

Adaptive Huffman coding technique is a slight variation of normal Huffman Coding. The vital difference between static and adaptive is that in the later input file is encoded dynamically that is the input can be encoded in the real time. The code tree constructed using adaptive has to be constantly updated as the input data is parsed along with the already processed symbols.

Adaptive Huffman principle states that in an optimal tree for n symbols there are nodes numbered $[y_1] < [y_2] < ... < [y_2n - 1]$ such that their corresponding weights $[w_1], [w_2], ..., [w_2n - 1]$ are always $[w_1] \leq [w_2] \leq ... \leq [w_2n - 1]$.

There are various implementations for adaptive but the significant ones are - **FGK algorithm** and **Vitter algorithm**.

FGK Algorithm

Siblings property forms the basis of the FGK algorithm. In this algorithm, both compressor and decompressor make use of dynamic Huffman code trees whose leaves represent the characters or symbols found in the input data and weight is the number of occurrence of each character. At any point during the compression-decompression cycle, exactly k (where k $\leq$ n) unique characters out of n would be present in the tree. In this algorithm, weight update of a node operation is not followed by checking the entire tree but instead it involves checking it only with its right siblings or the right siblings of its parents. Main steps involved in FGK Algorithm are as follows.

- Push an element in the code tree.
- Increment weight of node if the encountered symbol already exists in the code tree. Record its weight and path to the root.
- Check the weight of the recently altered node against all its right siblings to check if Sibling property is maintained and swap if otherwise.
- Move to parent, check weight against parent's right siblings, swap if necessary.
- continuously update the code tree while keeping tracking of the path to root, weight of the nodes, and swapping weights with right siblings if lower.

FGK Algorithm had a few drawbacks and to improve those Vitter proposed another algorithm. *We used an existing implementation of FGK Algorithm[11] in our project, to do a comprehensive analysis of all the Huffman coding algorithms with respect to the algorithm we have implemented that is Static Huffman Coding and Vitter.*

Vitter Algorithm

Vitter algorithm is based on the following conception.

- Nodes in the dynamic code tree are numbered using *implicit numbering*. It numbers the nodes in increasing fashion from left to right and bottom up. Therefore the nodes at the lower level will have a low implicit number in compared to the higher level ones.
- In the code tree, leaves of every node with weight w will precede all internal nodes having weight w.

The basic principle involved in this algorithm is also same as FGK Algorithm, both the compressor and decompressor make use of dynamic Huffman tree which has already encountered characters as its leaves. Except the **Not Yet Transmitted(NYT) node** where all the newly encountered symbols are stored. In this method, every node's weight and its order is kept track. Weights associated with the nodes is nothing but the frequency of the symbol represented by the given node and it increases as we go from left to right and top to bottom. The weight of the auxiliary nodes like the root node and intermediate nodes is the sum of their two child nodes. Orders are nothing but the numbers used for representing the nodes and the nodes of higher order will

have higher weights.

Initially, as the compression starts and no symbols have been encoded yet the very first symbol is added to the output in its original form and then is pushed to the tree and a code is being assigned to it. When the same symbol is encountered next the current code is written to the output and the frequency is incremented by 1. After the frequency has been increased it is made sure that the Huffman Tree constraint is still maintained. Hence every time an input symbol is parsed it should be followed by a tree update operation to maintain Huffman tree constraint which in turn results in modified code.

While decompressing the same steps as compressing are followed but the decompressor needs to know if the input data is the original symbol or a variable length code. Therefore an original symbol is always preceded by an escape character. As every original symbol is preceded by an escape character it should be made sure that it is not a variable length code for any of the symbols in the input data.

Below is the struct used by us for implementing the Vitter Algorithm.

```
typedef struct node{
    unsigned char symbol;
    int weight,
        number;
    node *parent,
         *left,
         *right;
} node;
```

Implementation and Analysis

Main steps involved in Static Huffman Coding are - building a frequency table, using a frequency table to build a code tree, traversing the code tree constructed to build a lookup table and finally encoding the input using the lookup table. The time complexity of the algorithm used by us to implement Static Huffman Coding is O(n) where n is the number of symbols in the input data provided.

We have chosen Vitter Algorithm over FGK Algorithm to implement Adaptive Huffman Coding. Though both FGK and Vitter algorithm have same underlying principles but the tree generated by both the algorithms are different. Vitter Algorithm produces a tree with a minimum height and it also minimizes the sum of distances from the root to the leaves[14]. Main steps involved in Vitter Algorithm are as follows:

Compression in Vitter Algorithm-

- If the input symbol is encountered for the first time then code for NYT is written on the output followed by the fixed code for the symbol.
- Newly encountered symbol is added to the tree.
- If a symbol that already exists in the tree is encountered then its code is written on the output.
- Code tree is then updated to preserve the invariant.

Updating code tree in Vitter Algorithm-

- If y is a node with current weight w, and y is also a root then update its current weight by adding 1 but never change the weight of NYT node.
- Swap y with the biggest order numbered node with the same weight or in the same block if y is not a parent and if the invariant is maintained.

Decompression in Vitter Algorithm-

- Decompressing is done using the dynamic code tree same as the compression.
- When NYT node is encountered, use the fixed code for decoding the symbol and a the new symbol to the tree.
- Code tree is then updated to preserve the invariant.

The time complexity of the Vitter Algorithm is O(n log k) where n is the number of symbols and k is the number of unique symbols in the input data.

Table 3: Running time analysis of Vitter Algorithm(Adaptive Huffman Coding)

Steps involved	Time Complexity
Build dynamic code tree for compression	O(n log k)
Build dynamic code tree for decompression	O(n log k)
Update the dynamic code tree	O(log k)

Results

We tested the algorithms we implemented using various input file of different sizes and format.The input data we used are mostly the ones which a normal person would encounter in day to day life. For example, e-books, news article, log files, emails, spreadsheets involved in varied business, high resolution images and video clips. Table 4 represents various input file formats and sizes used for testing in our project.

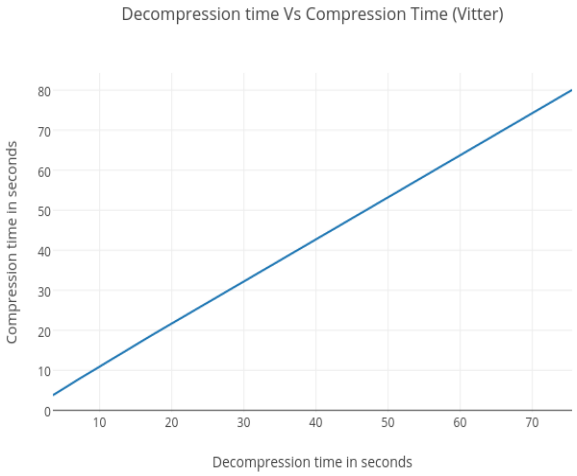
Table 4: Input file type and size range

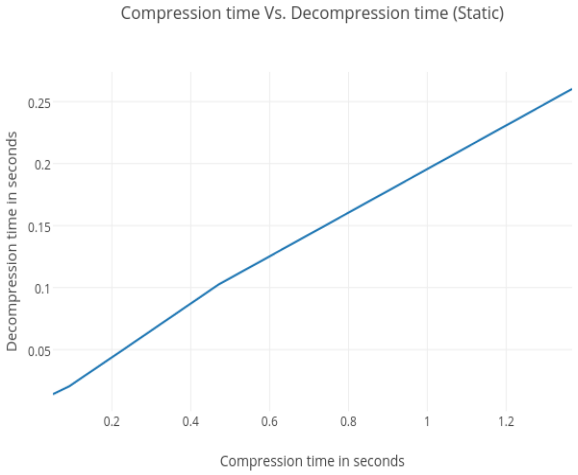
File format	Size range
.txt	100KB - 3MB
.pdf	100KB - 3MB
.png	100KB - 5MB
.mp4	500KB - 5MB
.xls	200KB - 7MB

Table below shows that the compression for the same size file is about 40% faster in Static Huffman coding in compare to the FGK Algorithm.

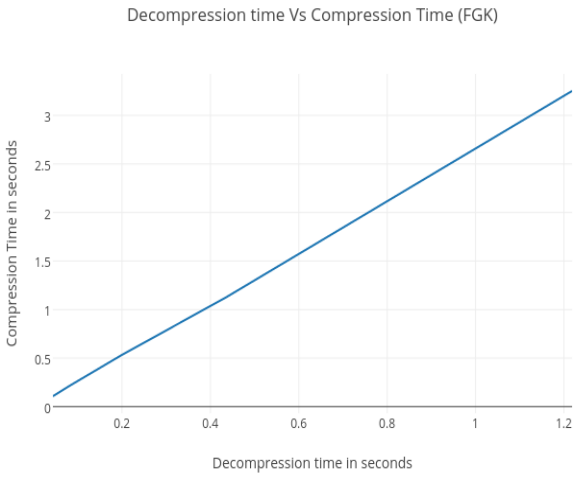
Table 5: Compression and decompression time for multiple file formats

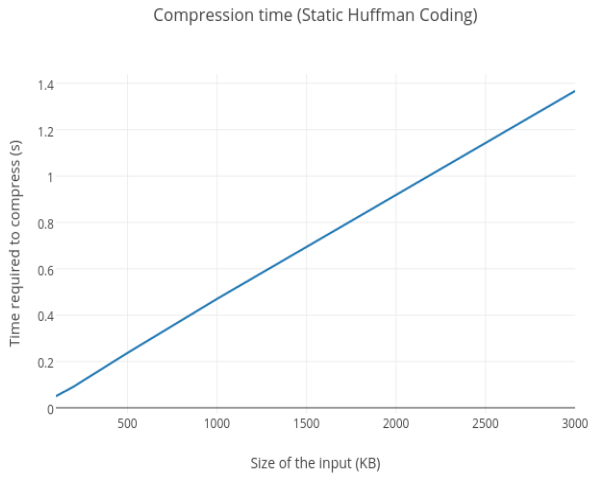
File format	Algorithm	Compression time (s)	Original file siz
.txt	Static	0.470	1000
.txt	FGK	1.129	1000
.png	Static	0.723	1000
.png	FGK	1.481	1000
.mp4	Static	0.721	1200
.mp4	FGK	1.528	1200
.xls	Static	0.473	700
.xls	FGK	1.017	700

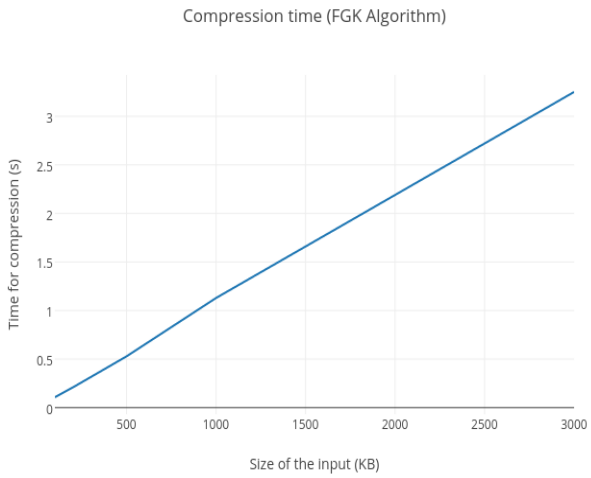


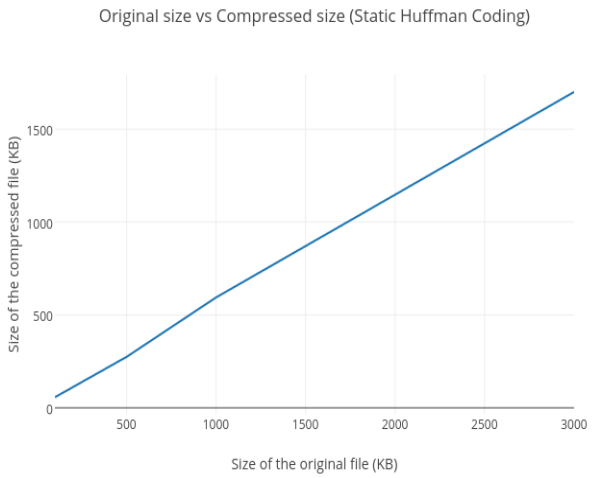


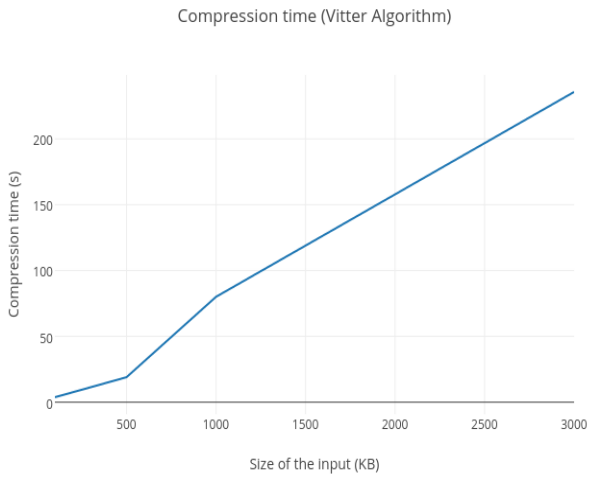
In above three graphs compression time is plotted against the decompression time for all the three algorithms used. Decompression time is normally smaller compared to the compression time for all but in Static Huffman Coding both compression and decompression time is much smaller with respect to both FGK and Vitter algorithm.

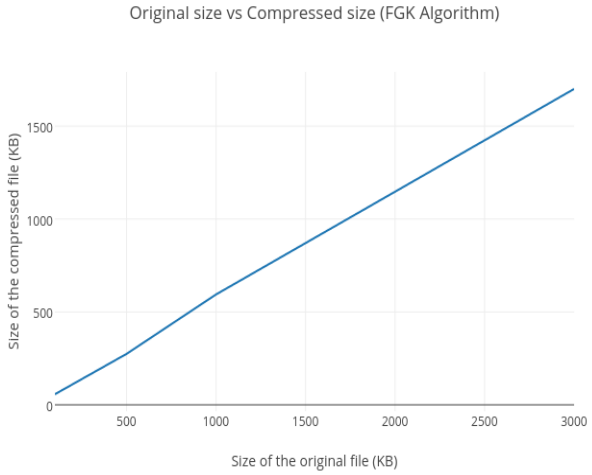


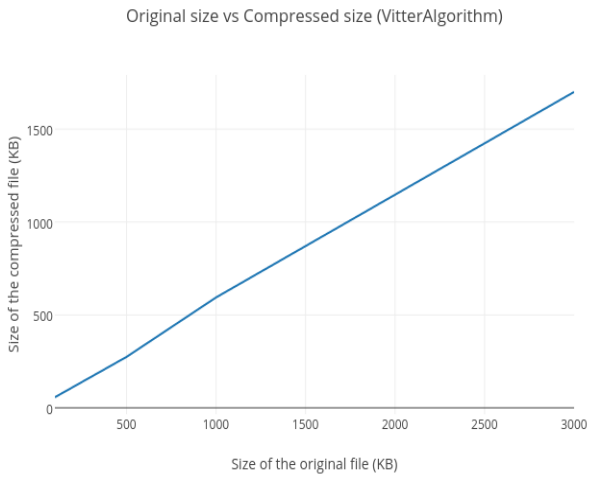












The above three graphs shows the time required in compressing the different sized files by all the three algorithms. It very clear from the above graphs that compression is undoubtedly very fast in Static Huffman Coding followed by FGK Algorithm in which compression time is 2.5 times higher than of the Static Huffman coding and Vitter Algorithm's compression time increases exponentially as the size of the input file increases. So, the compression is fastest in Static Huffman Coding followed by FGK Algorithm and then the Vitter Algorithm.

The three graphs above shows the size of the original file vs size of the compressed file. It is very clear that all the

three algorithms compress the files equally efficiently. The compression ratio is almost 40%.

Conclusions

Close scrutinizing of the results has helped us arrive at a conclusion that, the compression in Static Huffman Coding Algorithm is very fast when compared with both FGK and Vitter algorithm. It holds true even for bigger input data size of all the formats. The lookup table built using the code tree helps in faster encoding and decoding of the input data unlike in both FGK and Vitter Algorithm where the code tree has to be adapted for each symbol in the input data that is encountered.

Another important thing that we observed is that all the three algorithms compress the file equally and the size is reduced by almost 40% for all the inputs.

Static Huffman Coding should be used in all the cases except when the input contains repeated symbols Adaptive Huffman Coding performs well.

References

[1] *Journal Article*
Knuth, Donald E. "Dynamic huffman coding." Journal of algorithms 6, no. 2 (1985): 163-180.

[2] *Book*
Salomon, David. A concise introduction to data compression.
Springer Science & Business Media, 2007.

[3] Huffman Codes, https://cs.calvin.edu/activities/books/c++/ds/2e/WebItems/Chapter15/Huffman.pdf

[4] Data Compression using Huffman Coding, http://www.slideshare.net/RahulKhanwani/data-compression-huffman-coding-algoritham

[5] Java implementation of Huffman, https://github.com/auselen/huffman

[6] Wikipedia Article, https://en.wikipedia.org/wiki/Huffman_coding

[7] https://en.wikipedia.org/wiki/Adaptive_Huffman_coding

[8] MIT press Huffman Coding, https://mitpress.mit.edu/sicp/full-text/sicp/book/node41.html

[9] Sisgraph.com, https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html

[10] prezi.com, https://prezi.com/ovssuaqodxjh/adaptive-huffmancoding/

[11] Yet Another Adaptive Huffman Code, https://github.com/ikalnytskyi/yaahc

[12] Gutenberg, http://gutenberg.org/

[13] Adaptive Huffman Coding, https://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html

[14] Adaptive Huffman Coding, https://courses.cs.washington.edu/courses/csep590a/07au/lectures/lecture