Static Vs Adaptive Huffman coding - A Pragmatic Comparison

CSCI-B 503 Algorithms Design and Analysis
Prashanth Kumar Murali

Abstract

The aim of this project is to demonstrate the differences between Static and Adaptive Huffman coding. We have built a robust and fast static huffman coding system which performs much better than any adaptive encoding stradegies in C++. We intend to test with different inputs of varying formats and sizes to compare the compression size and time taken to compress and decompress. We also intend to explore the different applications and possibilities to use either of the techniques. In the end we intend to perform a complexity analysis in terms of time and space and prove that the fast static version performs much better than the adaptive approaches.

Huffman coding is a compression technique used to reduce the number of bits needed to send or store a message. It's based on the idea that frequently-appearing letters or sequence of letters should have shorter bit representations and less common letters should have longer representations.The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) [1] for each possible value of the source symbol.

Huffman coding is a form of compression classified under the lossless data compression technique. It implies that there is no data loss on data decompression. Most of the variable length coding systems have the problem of delimiters. In order to identify if a sequence or a character has ended, one has to introduce a stop element. For example the Morse code system uses pauses after a set of dots and dashes to indicate the end of a letter. Huffman coding is a much smarter variable length coding system as it eliminates the use of delimiters. The code is formed from a tree in such a way that each code ends with the characters being identified after which there is no possible traversal. The characters are placed in the leaf nodes and the path to that is unique. Hence, it should be faster than other systems as it does not need to make any comparisons for termination.

Model and Algorithm

We have Implemented both the static and adaptive Huffman Coding in C++. Each of these accepts a input file of any format and generates a compressed binary file. These binary files can be decoded to get the original inputs back without any loss. The whole code base for each of these methods do no exceed 200 lines of code. The initial plan as per the proposal was just to implement Huffman coding for text files thereby comparing the encoding standards of Static and Adaptive variants only on plain text. But the end product is a tool which can compress data of any format using both the variants. The basic concept of reducing file size in Huffman coding is that we can reduce the representation of the characters in a file based on its frequencies.

Static Huffman Coding

This methods starts off by constructing a list of all characters that are present in a filestream. For text files this is the most straightforward as we would just be constructing a list of all unique letters. But for all the other formats we just clump data into 8 bit characters and form a list of those. Then we make this a hashmap of characters to the frequency of occurance in the current file. When we encounter the same character again we increment the count in the hash. The tree node structure for this variant is as follows:

```
class HuffNode {
  public:
    char data;
    int freq;
    HuffNode * left, * right;

    HuffNode(char data, int freq=0,
        HuffNode *left = NULL,
        HuffNode *right = NULL) {
      this->data = data;
      this->left = left;
      this->right = right;
      this->freq = freq;
    }

    ~HuffNode() {
      delete left, delete right;
    }
};
```

Note that each node has 2 values - a character and its frequency. The steps for encoding [3] a raw file is as follows:

- Get input stream of data from a file.
- Find the characters in the file and compute the frequencies of each. Store this in a hashmap (character, frequency). We do this in one parse through the file.
- We build the encoding tree from the frequency map. We have employed a unique way of implementing this. Leaf nodes are created for each character. These are pushed into a priority queue which is sorted by the frequencies of each character. The first 2 elements are popped off to form the first 2 leaves and the sum of their frequencies is pushed into the queue with blank character. After this, in each step the sum of frequencies and the next character is popped, the character is made a leaf and the incremental sum will become its parent.
- The tree thus formed is used to construct a lookup table of characters and associated code. We use a queue to append the bit values on traversing the tree. By our logic of forming the tree, the left leaves are always the characters, so at these points we create a new entry in the lookup table for the code formed so far and the character in the node.
- This lookup table is written into a key file which will be used for decoding.
- We encode the file based on the lookup table by replacing the characters with the text code.
- We write the data by setting the bits to 1 when the code is 1 and 0 otherwise. There may be codes lesser than 8 bits which is the character size required to write in a file. So we pack with the bits of the next character.

The resulting file is very concise in size because of the bit packing we do in the final step. Consider the following example:

```
peter piper picked a peck of pickled
peppercorns.
```

The Code lookup table for the above text by this method is:

```
  101
a 110010
c 1101
d 10011
e 111
f 01100
i 1000
k 0111
n 110011
o 11000
p 00
r 010
s 01101
t 10010
```

Notice that the code of the letter P is the smallest as it is the most frequently appearing letter. Hence in the compressed file the letter "P" which tookup 8 bits of space originally would only take 2 bits. Accordingly the size of the file which was 47 bytes got reduced to 21 bytes. Hence, this method is very effective for text encoding and compression.

Decoding the files is simple too. The following are the steps to do that.

- Read the key file and construct the tree out of that file. Every 0 would correspond to a new left subtree node and every 1 to a right node. The end would have the character.
- Read the binary stream from the encoded file and traverse through the tree till a leaf node. Add the character in the leaf node to the output stream and write to the file.

As you can see the decoding process is much simpler and hence takes much lesser time than encoding.

Adaptive Huffman Coding

Adaptive or Dynamic Huffman coding is a bit more complex and ambitious when compared to static huffman coding. The idea is to not parse the file twice and finish encoding in one parse. There are 2 effective algorithms to achieve this. We have studied and implemented one of them and derived results from a readily available package for the other.

Vitter Algorithm

In Vitter Algorithm the code is represented as a tree structure in which each node has a corresponding weight and a unique number [4]. Weights must satisfy the sibling property, which states that the nodes must be listed in order of decreasing weight. Hence the tree node for this algorithm is as follows:

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

Notice that the node has extra weight and number attributes. In terms of connections, it has the parent node connection which was not there in Static. The steps to encode using this algorithm is as follows:

- Read in the symbol or character from the file.
- Both Compression and Decompression starts with only root node which has the maximum number and a Not Yet Transmitted (NYT) node.
- For every symbol in the file, we have to transmit code to its leaf node.

- If the current symbol is new and Not Yet Transmitted, 2 nodes are added as children to the NYT node. One will become the next NYT node and the other leaf will be the symbol's node. Increase weight for the newly created leaf node and the old NYT node. Then restructure this tree.
- For restructuring the tree, we move the parent node, check if the node has the highest number in the block, else swap the node with the highest number except if the node is the root.
- Finally increase weight for the current node.
- If the item is already in the tree and not a NYT, go to the symbol's leaf node check if the number is the highest number in the block, swap with the highest number and increase the weight for the current node.

The decompressor follows almost the same steps as above. It reads a symbol and adds it to the tree. On reading the compressed huffman code, it scans the tree to find the symbol and replaces it in the text. It rearranges the tree as the compressor does. The decompressor too is more complex than Static Huffman Coding. The symbol, number, weight for the characters in the example described above is as follows. Note that this is printed in order of position in the tree.

code	character	number	weight
0x72	r	4079	4
0x64	d	4072	2
0x6f	o	4071	2
0x63	c	4076	4
0x6b	k	4075	3
0x70	p	4091	9
0x2e	.	4066	1
0x73	s	4067	1
0x61	a	4074	1
0x6c	l	4069	1
0x69	i	4082	3
0x74	t	4073	1
0x66	f	4063	1
0x6e	n	4065	1
0x20	\s	4085	7
0x65	e	4084	8

FGK Algorithm

In FGK algortihm both the encoder and the decoder maintain dynamically changing huffman code trees. For each symbol encountered in a file a codeword is sent through the current tree which updates the tree. The problem is to efficiently update the tree from n symbols at a point to n+1 symbols when a new symbol is encountered. The Solution to this is a 2 step process [4].

- The original tree is transformed to a valid Huffman Tree with the current n symbols.
- The current symbol value is incremented and the correspoding parent are changed. Then the tree is reshuffled based on the changed values.

This generates a valid huffman tree with n+1 symbols. The sibling property is also satisfied. There remains no node with the previous weight that is higher numbered. We haven't implemented this algorithm from scratch. We referred a library YaAHC (Yet Another Adaptive Huffman Coding) [11] which provided a c++ implementation of this algorithm. Decompression here takes place the same as vitter algorithm. The tree is updated on every symbol update.

Complexity Analysis

We proceed to determine the time and space complexity of each of these algorithms. We will be seeing the time taken in each step for the three algorithms and conclude the complexity

Time

The time taken for each step of static encoding is as follows.

Table 1: Static Huffman Encoding

Step	Time Complexity
Build Frequency	O(n)
Build Encoding Tree	O(k)
Build Lookup Table	O(klogk)
Encode	O(n)

In Build Frrequency, we read the input file character by character and compute frequencies and store it in a list. Hence the time taken for this is O(n) where n is the number of characters. In Build Encoding tree we take this sorted frequency table and create nodes for each of them and put it in a priority queue. This takes k time considering we have k unique alphabets. Then we insert these into the tree by popping from the priority queue which takes another k time and hence the time complexity for this step is O(k). To build the lookup table to encounter a character we parse the tree to get to the character's node and then we use the path to find the code (left subtree is '0' and right subtree '1'). Hence, we put in k entries with a maximum time of log(k) to parse through the tree for each entry. The max running time for this is O(klogk).

The Encoding step itself takes only O(n) time as we parse through the characters in the input file again, lookup the code for each character in the lookup table and put those bits in the output stream. The time taken for this is O(n). We can say that the maximum time complexity of the whole process is O(n). We have reduced the time for encoding here by using the lookup table.

Table 2: Static Huffman Decoding

Step	Time Complexity
Build Decoding Tree	O(klogk)
Decode	O(nlogk)

The above table shows the decoding part of static huffman algorithm. We take in the key file and build the Decoding Tree from that. This takes O(klog(k)) time as there are k unique symbols and insertion of each into the tree can take a maximum of log(k) time. For decoding step we take the bitstream from the encoded file and traverse through the tree to find the characters. Assuming the input had n characters, the decoding taked O(nlog(k)) time. This stays the same with all the forms of adaptive decoding.

For both the Adaptive Huffman Coding algorithms, the encoding time is the same. For every character encountered in the text they go by either placing them in the tree or tracing their position in the tree. When there are n characters and k unique characters in the text, this comes to O(nlog(k)).

This is comparitively much higher than the O(n) in Static Huffman Coding. This will produce significant difference in time, on large file sizes with varied and dense unique character space. This will be evident in the next section where we test the algorithms on different types of data.

Space

The static algorithm takes much more space when compared to the adaptive algorithms. The static algorithm takes k space for storing the tree and another k space for storing the lookup table. Moreover during compression the static variant also stores the lookup table as a key file with which the file can be decoded. This key file can occupy space in the order of klog(k). While this may seem to decrease the efficiency of compression, for large files, this key size become negligible.

The Adaptive algortihms take only k space for storing the tree and no other extra space. They also don't produce any key file and the tree is constructed dynamically. It is definite that the Adaptive Huffman Coding algorithms are much more space efficient than the static ones. The compression ratio is fairly the same for all the three.

Example Data

We have considered different types of data to analyse the performance of each of these algorithms. Most of the data used is text as it is easier to verify the validity of compression. We have extensively tested with data of varying sizes ranging from 4kb - 200mb. The text data is obtained from gutenberg.org [12] which are books in plain text format, open-sourced. Images, Videos, PDF documents and a few other file formats of comparable sizes have also been tested.

Results and Analysis

We did a pragmatic analysis of the performance of all 3 algorithms and generated statistical data to prove that Static Huffman Coding performs much better than the rest.

Running Time

Static Huffman Coding performed supremely well on high size data, contrary to normal beliefs. Eventhough it had to do 2n passes, it was still much better than nlog(k) of the other 2 algorithms. Specifically the time on files more than 1 MB was about 46% lesser than the commercial FGK algorithm and about 220 times fater than the naive Vitter algorithm.

Figure 1: Performance of Static Huffman Coding

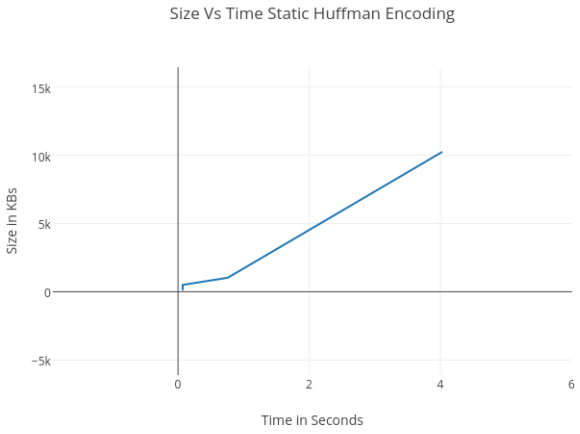
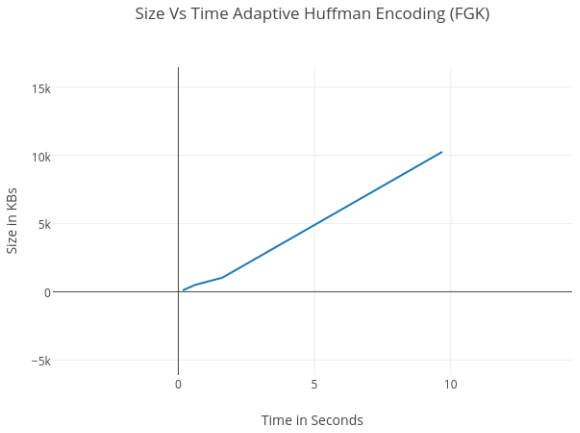


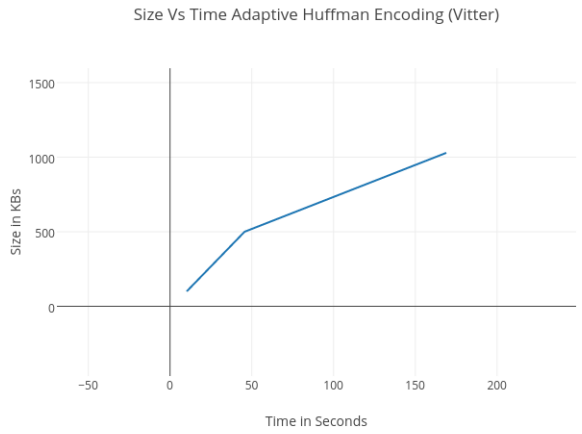
Figure 2: Performance of FGK Huffman Coding



Encoding Vs Decoding

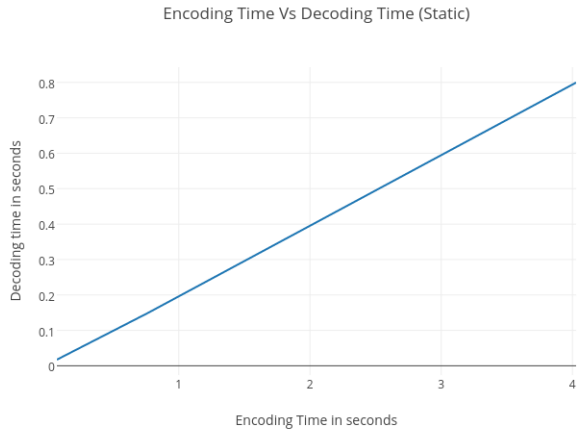
We analyzed the relationship between the encoding time and decoding to get some more staggering suggestions that static was performing better. The following graphs show the relation between compression and decompression time on files ranging from 100 kb to 100 mb in size. You can find that even here static performs much better with decoding taking place 5 times faster than encoding compared to a

Figure 3: Performance of Vitter Huffman Coding



mere 0.8 acceleration in vitter and 2 times faster in FGK. If the application involves exponentially faster decoding time, Static Huffman coding is still the way to go. Figure 4, 5 and 6 demonstrate this.

Figure 4: Time ratio between compression and decompression - Static Huffman Coding



Filetypes

We tested to compress different filetypes and not only text in order to verify the performance in terms on time and compression on each. The table gives a view of this on all 3 algorithms. As we can see, there is not much of a compression in JPEG images, videos and PDFs. These are already low space formats and even then a few kbs of size has been reduced. In order to make a clear case regarding the compression we synthesized new files with high repetition of symbols. Compressing them gave great results upto the range of 1:8 compression ratio. The results

Figure 5: Time ratio between compression and decompression - Vitter Huffman Coding

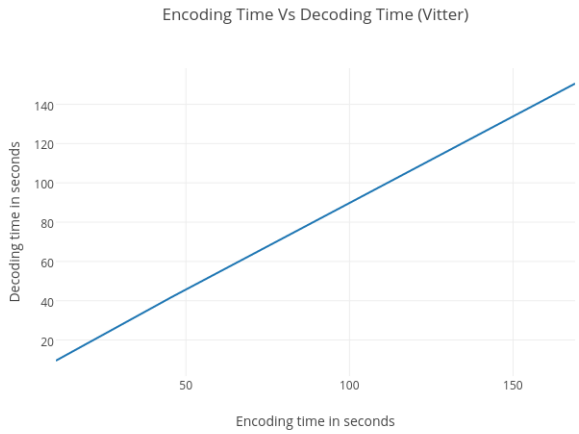
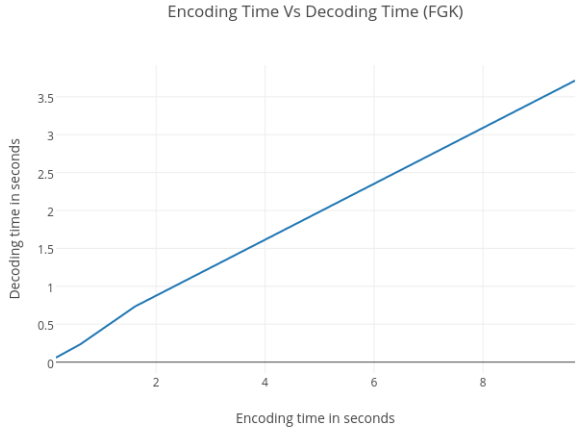


Figure 6: Time ratio between compression and decompression - FGK Huffman Coding



of this can also be seen in table. Even here static performed significantly better than the other 2 algorithms.

File Types	Compressed Sizes			Actual Size
	Static	Vitter	FGK	
JPEG Image	2.1 MB	2.1 MB	2.1 MB	2.1 MB
MP4 Video	1.9 MB	1.8 MB	1.8 MB	2.1 MB
PDF data	1.8 MB	1.9 MB	1.9 MB	2.1 MB
High repetition data	12 kb	16 kb	16 kb	90 kb

Table 3: Compression on different formats of data

Conclusions and Improvements

We would like to conclude that the Static Huffman Coding algorithm has many positives when compared to the adaptive ones. It is much faster than its counterparts and easier to implement. It involves very less work to obtain the codes and there is a convinient way to view them. Since there is no way to decode it without the key file, it can also be used as a means to transmit data with privacy. It is a variable length code algorithm and hence it would be extremely difficult to decrypt it without the actual file or the key file. It does have its drawback of using up more space to do this, but if there is a window of accomodating that space, this method can provide amazing results.

As an improvement we can try to store the key file as binary and read it too. It would reduce the file size of the keyfile. In network communication the key can be exchanged via handshake and we can implement streams of fast data transfer by an amalgamation of adaptive and static huffman coding.

You can find the codebase at https://github.iu.edu/prmurali/algorithms_503_project

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/