Wayne State University

Department of Computer Science

Huffman Coding

Mark Nuppnau

Professor: Dr. Xu

A report submitted for completion of CSC 6580: *Design and Analysis of Algorithms*

May 3, 2022

Contents

Chapter 1

Introduction

1.1 Background

The main objective of this project is to compress and decompress a file using the Huffman coding algorithm with a secondary objective of compressing a file as much as possible. Huffman coding is one technique used to compress a file, and is usually used as a one part of a larger compression algorithm. Huffman coding works by first getting a frequency of input values (i.e., bytes, ASCII characters, etc). Suppose there is a text file that contains the statement, *three free referee fee he*. The frequencies for the previous statement are tablulated and then sorted, as shown in 1.1.

Once we have the frequency table created we can create our Huffman coding tree that will give the new encoding values for each character. To create a Huffman coding tree, we take the two smallest frequencies and create a shared parent node, connecting the two characters with the smallest frequencies. A more formal definition of the procedure is shown in 1. (Cormen et al., 2009)

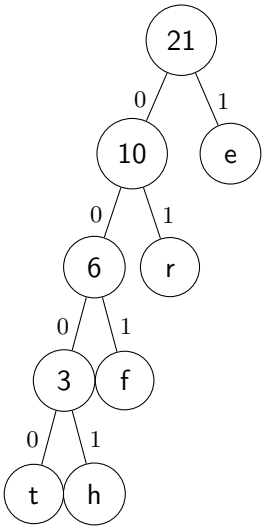
Algorithm 1 Huffman coding

```
1: n = |C|
2: Q = C
3: for i ← 1 to N-1 do
4:     allocate a new node z
5:     x = EXTRACT-MIN(Q)
6:     y = EXTRACT-MIN(Q)
7:     z.left = x
8:     z.right = y
9:     z.freq = x.freq + y.freq
10:    INSERT(Q,z)
11: end for
12: return EXTRACT-MIN(Q)
```

When the above algorithm is applied to the frequency table in 1.1, a parent node will be

1	2	3	4	11
t	h	f	r	e

Table 1.1: Frequency table of string *three free referee fee he*



char	code
t	0000
h	0001
f	001
r	01
e	1

Figure 1.1: The Huffman tree (left tree) and the resulting encoding for each character (right table)

created for *t*, and *h* with a value of 3 (2 + 1). The algorithm will then look at the next two smallest node values out of the updated node list *3,3,4,11*, which will be *3,3* to create a new parent node of value 6. This process will continue until we have our final tree which will contain the Huffman codes associated with each character. To get the associated Huffman codes from the tree, the tree needs to be traversed from the root to the desired character/leaf node. Traversing the tree left is a code 0, and traversing the tree to the right is a code 1. The final tree and resulting Huffman coding for each character can be seen in 1.1.

Now the character *e* can be represented by a single bit, *1*, instead of the ASCII representation of the letter *e*, *01100101*. Since *e* is the most frequently used character, we can save 7 bits of storage space for every occurrence of the letter. Based on the encoding from the Huffman tree, the most bits we would have to store for a letter would be 4 bits. The original file input is not the only data that needs to be stored in the compressed file though. There also needs to be a way to decode the input encodings. There are several techniques used to store the information needed to recreate the encoding table/Huffman tree.

Even though the example above counts the frequency of characters in a string, possibly from a text file, compression needs to apply to any type of data, not just text files. ASCII characters (i.e., 2, I, =) are represented by eight bits (1 byte). Most characters used in a text file are ASCII characters and could be represented with a number between 32 and 127. Since there are 256 possible representations (2^n) of eight bits, it's possible to represent ASCII characters with eight bits and use the extra bit for other purposes. Because of the limitations of ASCII code to represent many different variations of characters, Unicode was created as a single character set. Unicode characters are represented by code points (i.e., *H* is represented as U+0048). The *U+* stands for Unicode, and the *0048* stands for the hexidecimal value of the character. The encoding system UTF-8 was created to store these code points. Hexidecimal values up to 127 are stored in eight bits, and values greater are stored in at least 16 bits. So if there is a text file that needs to be compressed and it contains gothic characters, those characters could be represented by four bytes of data. If a text file contains many different Unicode characters, the number of character encodings created from a Huffman tree could grow significantly.

In order to avoid a very large huffman tree and to be able to read in different file types,

it's possible to read a file byte by byte or even bit by bit. This way, even though there may be groupings of bytes to represent different characters, there are still only 256 possible binary representations of eight bits. This will restrict the Huffman tree from growing too large, and allow the program to read different file types other than text (i.e., binary). If there are a few distinct byte representations with high frequencies from the input file, storage space savings could be significant.

1.2 Problem statement

Given an input file, any file type, implement a set of tools to perform data compression using Huffman coding. The set of tools should include:

- A stand-alone tool to get (byte) symbol frequency stats from a file
- A stand-alone tool to construct a Huffman encoding table and its matching decoding table from a given symbol frequency stats file
- A stand-alone tool to encode a file for a given Huffman encoding table
- A stand-alone tool to decode a Huffman encoded file for a given Huffman decoded table
- An integrated tool to compress and decompress a file using Huffman code with the above components

1.3 Aims and objectives

Aims: My intentions of this project are to; 1. Build a program that will read in any file type (byte by byte), and successfully compress and decompress in a timely manner. 2. Store the Huffman tree/encoding table in the compressed file in an efficient manner. My non-measureable aims are to learn not just the Huffman coding algorithm, but also character sets, and the other various aspects of this project in c++.

Objectives: My objectives are as follows:

- Read in a file and output to a file the frequency stats of the bytes read in
- Read in the frequency stats file and construct a Huffman tree
- With the Huffman tree, construct an encoding/decoding table and output to a file
- Read in the encoding table file and input file to construct and output an encoded string to a file
- Read in a file and get the frequency stats of the bytes read in, construct a Huffman tree and encoding/decoding table, construct an encoded string of the Huffman tree/encoding table and input bytes, and save the encoded string as a compressed file
- Read in the compressed file, rebuild Huffman tree/encoding table from header, decode input encoding string to recreate and output original input

Chapter 2

Design

There is a brief mention in Chapter 1 of the design of this project as it relates to the individual and integrated tools. This chapter will go into more detail of that design.

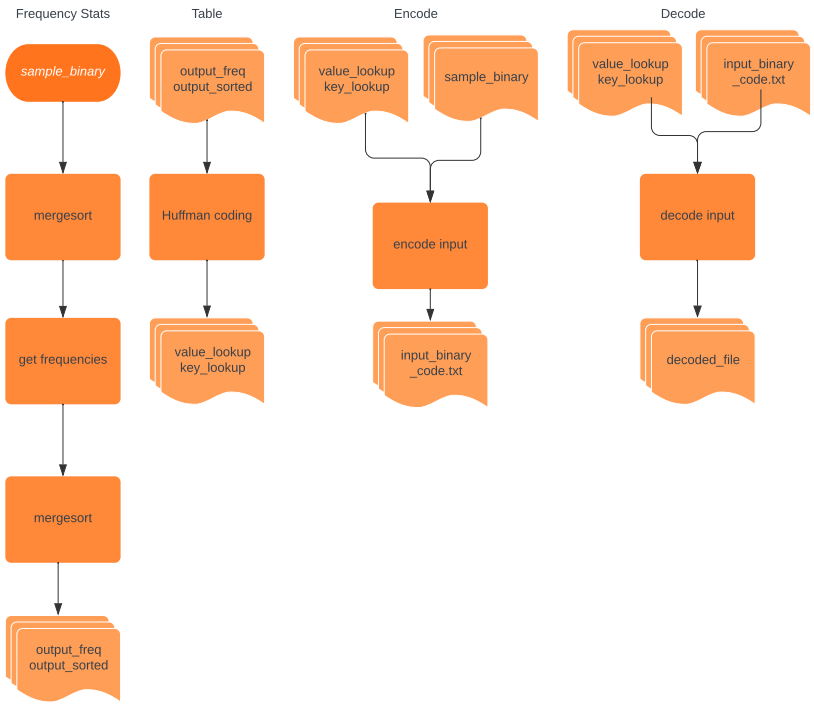


Figure 2.1: High-level flowchart of each individual tool

2.1 Individual tools

A high-level overview of the processes for each individual tools can be viewed in 2.1. The first individual tool is to generate the frequency stats from the input file. The input file being *sample_binary*. The input is fed into a mergesort function that sorts the input by bytes.

The sorted bytes are then counted to get the frequencies of each byte. Those frequencies and bytes are then sorted using mergesort again. The output from the mergesort operation are two vectors that get saved as *output_freq* and *output_sorted*. The file *output_freq* will contain the frequencies of each byte read in from *sample_binary* in ascending order. The file *output_sorted* will contain the bytes in order of ascending frequencies.

To get the Huffman tree/encoding table created, the two files, *output_freq* and *output_sorted*, are read into two separate vectors and fed into the Huffman coding function. The Huffman coding function creates the Huffman tree/encoding table and outputs two vectors that represent the encoding table and saved to file as *value_lookup* and *key_lookup*. The file *value_lookup* will contain the Huffman codes generated from the creation of the Huffman tree. The file *key_lookup* will contain the bytes associated with each Huffman code.

The encode the input data from *sample_binary*, we read in the input data and the two files generated above, *value_lookup* and *key_lookup*. The input data is looped through one byte at a time and is matched against *key_lookup* and the matching encoding from *value_lookup* is returned. This creates the binary encoding string for the input data and is saved to file *input_binary_code.txt*. The final individual tool is the decoding tool. This tool utilizes the text file created in the encoding step and the two files generated in the table step. The input binary encoding text file is read in as a string and for one byte at a time is matched against the *value_lookup* to find the matching byte in *key_lookup*.

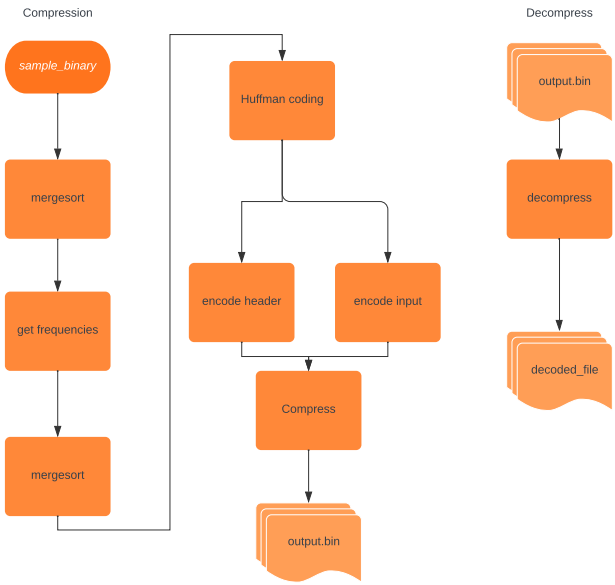


Figure 2.2: High-level flowchart of integrated tools

2.2 Integrated tools

A high-level overview of the processes for the integrated tools, compression and decompression, can be viewed in 2.2. The beginning of the flowchart follows the same paths as the individual

tools with the exception of outputing data to files. The process starts with reading in the *sample_binary* file, sorting the data with mergesort, getting the frequency counts of the bytes, sorting the frequency counts and feeding that data into the Huffman coding function to generate the Huffman encodings for each byte which are then used to create the binary encoding for the input data. The process that differs from the individual tools is the creation of the header binary encoding. This header will store the information needed to regenerate the Huffman tree/encoding table. The header binary encoding and the input binary encoding strings are then concatenated. Any zero or one values in this string will be stored as a single bit, and the remaining values (bytes in the header) will be stored as bytes. The compressed file output from this process is *output.bin*.

The final part of the integrated tool is the decompression. The process starts by reading in the file *output.bin*. It starts by reader information about the header to recreate the Huffman tree/encoding table— there will be more detailed information on this in the next Chapter. Once all of the header information is read in, the program can recreate the Huffman tree/encoding table and start to read in the input binary encoding bit by bit. Each bit read in will be checked against the Huffman tree/encoding table for a match and then will return the associated byte value. When all byte values have been returned, they will be saved to *decoded_file*.

Chapter 3

Implementation

We described in Chapter 2 the design of our project but did not go into much detail regarding that design. In this chapter we will explore that design in more detail. Instead of covering both the individual tools and integrated tools, we will focus just on the integrated tools since the processes within are the same for the individual tools. Reference the flowcharts in Chapter 2 when needed while reading through this Chapter.

3.1 Details of the Compression Algorithm

The input file for this program needs to be named *sample_binary*. Any file can be copied to this file name using *cp example_file.txt sample_binary* in the terminal at the project directory. The *sample_binary* file is read in byte by byte and is stored in a vector. Specifically, each byte is stored in a vector of type *unsigned char*. There is another vector that is created, *input_arr*, to preserve the input data. The standard library class *ifstream* is used as the input stream to operate on files and class *istream_iterator* is used as the input operator. This process can be viewed in 3.1

```
1   std::vector<unsigned char> arr;
2
3   std::ifstream file("sample_binary", std::ios::binary);
4   file.unsetf(std::ios::skipws);
5   std::streampos fileSize;
6
7   file.seekg(0, std::ios::end);
8   fileSize = file.tellg();
9   file.seekg(0, std::ios::beg);
10
11  arr.reserve(fileSize);
12  std::copy(std::istream_iterator<unsigned char>(file), std::istream_iterator
     <unsigned char>(), std::back_inserter(arr));
```

Listing 3.1: Reading in a file byte by byte into a vector of type unsigned char

The vector *arr* is then passed to the mergesort function which sorts the vector of bytes and returns the sorted vector. The sorted array and a newly create array of type *int* are then passed to the *get_freq* function. This function will return a vector of distinct bytes and a vector of frequencies. These two vectors are then passed to another mergersort function that will sort both vectors based on the frequencies and are returned. The returned vectors are then passed into the HuffmanCodes function where the Huffman tree will be created based on the sorted frequency and byte vectors. The HuffmanCodes function builds a Huffman tree by building a min-heap based on the two vectors being passed through. Two new vectors,

char_map and *code_map*, are also passed through the HuffmanCodes function to keep track of the nodes being created in the Huffman tree and will help build the encoding table.

The encoding table is then built from *char_map* and *code_map*. The vector read in earlier, *input_arr*, will now be used to create our input binary encoding. Each byte from the input vector is checked against a mapping of *char_map* and *code_map* and the matching encoding is stored in a new vector. Once all bytes of the vector have been read in and matched, the new vector of encodings creates a single string of binary encodings for the input data. Now that the input data is now encoded, in order to decode that information, the Huffman tree/encoding table needs to be stored along with the input binary encoding.

There are several different ways to encode a Huffman tree/encoding table. The way I was able to encode this information was to calculate the lengths of each binary encoding from the Huffman tree and get the frequency of those lengths. We will use the encoding in Figure 1.1 in Chapter 1 as a simplified example. The characters *e, r, f, h, t* have the following code lengths *1, 2, 3, 4, 4* and those code lengths have the following frequencies *1:1, 1:2, 1:3, 2:4*. A vector is created for the code frequencies , and another for their respective code lengths—these vectors need to be sorted in ascending order by the code lengths.

The header binary encoding will contain a preheader that will start with the number of input binary encodings. In our example, there are 5 binary encodings, one for each unique character. Then we append to the preheader the first value of the code frequency vector, then the first value of code lengths vector and repeat this process for each value in the vectors—a space is entered between each of these values to help determine how many digits to read in when decompressing. Doing this with our example, we would currently have a preheader of *5 1 1 1 2 1 3 2 4*. Then the Huffman encodings and characters are appended one at a time such that the final header string will look like *5 1 1 1 2 1 3 2 4 1e01r001f0001h0000t*. The preheader values, will allow us to loop through the remaining part of the string and help recreate the Huffman tree/encoding table. The values in the preheader *5 1 1 1 2 1 3 2 4* will be stored each as bytes. The remaining values in the header, however, may be stored as a bit if the value is a zero or one. This will help compress more information when dealing with larger and more binary encoding values. The input binary encoding string is then appended to this header string after the header string is converted to a string of zeroes and ones.

When the binary header string or input binary encoding string is created the number of binary digits may not be divisible by eight. Since the program stores data as bytes, the program pads the strings with zeroes when the number of binary digits is not divisible by eight. As an example, suppose the input binary encoding is *10101000101*. Since there are eight binary digits with a remainder of three, the string needs to be padded with five digits (8-3) in order to store this string with two bytes. I chose to pad the strings with zeroes when needs. Also, an extra byte needed to be created to tell if padding was indeed performed. Suppose the padded input binary string is now *1010100000000101*, with a padding of five zeroes. An extra byte can be appended to this string which can give information on the number of zeroes are padded, if any. In the recent padded input binary string there are five zeroes padded, and to represent five a byte like this *00000111* can be create. The last byte of the string can be read when decompressing, checked for consecutive zeroes, then ones, and if that is determined, check the next eight bits from the end to check for a padding equal to the number of zeroes in the previous byte read in. Then, if there is padding, remove the last byte and the padded zeroes from the string. The file that the final binary string is written to is *output.bin*.

3.2 Details of the Decompression Algorithm

The previous section on compression covered most of the detail as it relates to the encoded binary string. The file, *output.bin*, is read in by the program. There are three parts of the encoded binary string that need to be worked on separately; 1. preheader, 2. header, and 3. original input. For the preheader, since there can be multiple digits or single digits, one byte is read at a time until the binary representation of the space key is encountered. Until the space key is encountered, each byte read in is converted to its related ASCII value and concatenated with the next. In the example of a preheader give in the previous section, *5 1 1 1 2 1 3 2 4*, only the first byte of the preheader is read in to determine the number of input binary encodings created from the the original Huffman tree. Then the remaining values of the preheader are stored in two separate vectors, *number_of_codes_v* and *code_size_v*. With the mentioned preheader, *number_of_codes_v* will contain *1,1,1,2* and *code_size_v* will contain *1,2,3,4*.

There will be three loops created to recreate the encoding table originally created by the Huffman tree. Within the third loop, the remaining header binary encoded string, *1e01r001f0001h0000t* in binary encoded form , will be worked on. The first loop will work through the number of unique binary encodings created by the Huffman tree, in this example, 5. The second string will loop through each of the frequencies stored in *number_of_codes_v* and the third loop will loop through each of the code sizes stored in *code_size_v*. Looking at the binary encoded string above, the program will know there is one one-digit to read in, read in the character, then there is one two-digit character to read in then the character, and this will continue until the header is read through fully and our encoding table is recreated. Once this encoding table is recreated the binary encoded input can be read in one bit at a time and matched against the encoding table to pull the related byte value. This is how the original file contents can be recreated.

Aside from the logic of the decompression, there were a couple changes to code that significantly improved the performance of the decompression process. The first was storing the entire binary encoded input string in the standard c++ container of bytes *std::string*. The second had to do with the process of appending binary digits to each other when there was no matching binary encoding in the encoding table. The original code appended two strings like this, *orig_string* = *orig_string* + *new_string*, but the much more performant way to append strings is with the string operator like this, *orig_string* += *new_string*. These two code changes significantly improved the perfomance of the decompression process.

Chapter 4

Results

As mentioned previously, the goals of this project are to have a successfully running program, and to compress the data as much as possible. The best available compression algorithms do not use Huffman coding as a single compression technique, but as a part of a larger algorithm. So the performance of Huffman coding alone will not do well compared to modern compression algorithms. To test the performance of this program, a compression of all files in the Silesia corpus will be performed and compared to the compression size of files using the Bzipped algorithm (Deorowicz, n.d.). The Silesia corpus consists of 12 different files that are intended to cover typical data types.

To give an idea of how Huffman coding is used in modern compression algorithms, below is the algorithm for Bzipped (Wikipedia contributors, 2022):

- Run-Length encoding (RLE) of initial data.
- Burrows-Wheeler transform (BWT), or block sorting.
- Move-to-front (MTF) transform.
- Run-length encoding (RLE) of MTF result.
- Huffman coding.
- Selection between mutliple Huffman tables.
- Unary base-1 encoding of Huffman table selection.
- Delta encoding of Huffman-code bit lengths.
- Sparse bit array showing which symbols are used.

As shown in the algorithm above, Huffman coding is just one part of the algorithm and there are multiple Huffman tables generated. Whereas, this program uses a single Huffman encoding table to compress and decompress a file. The performance results are shown in 4.1.

As read in the results, the single Huffman tree gives decent compression performance on most files. The worst performing compression is on the file *sao*. *Sao* is a binary database made up of records of complex structure for astronomical star catalogues. The best performing compression is with the file *nci*. *Nci* is a chemical database of structures. Despite the decent performance of the single Huffman tree encoding, the results of the Bzipped algorithm are much more impressive. The improvement upon the compression of file *osdb* goes from a compression ratio of 0.827 to 0.277–compressed to a third of the original compression size.

Table 4.1: Performance of Huffman coding on Silesia corpus (size in bytes)

File Name	File Type	Original Size	Compressed Size	Compression Ratio	Bzipped size	Bzipped compression ratio
dickens	English Text	10192446	5826277	0.571	2799528	0.274
mozilla	exe	51220480	39977454	0.780	17914392	0.349
mr	picture	9970564	4623504	0.463	2441280	0.244
nci	database	33553445	10224138	0.304	1812734	0.054
ooffice	exe	6152192	5124814	0.833	2862526	0.465
osdb	database	10085684	8342485	0.827	2802792	0.277
reymont	Polish pdf	6627202	4032048	0.608	1246230	0.188
samba	src	21606400	16547188	0.765	4549790	0.210
sao	bin	7251944	6843822	0.943	4940524	0.681
webster	html	41458703	25929192	0.625	8644714	0.208
xml	html	5345280	3711377	0.694	441186	0.082
x-ray	medical image	8474240	7021908	0.828	4051112	0.478

Then looking at the compression ratio of 0.054 for *nci* with Bzipped, the file is compressed to nearly a sixth of the original compression size. In general, it seems that Huffman coding can be used successfully as part of a larger compression algorithm.

Chapter 5

Reflection

This project came with many learning experiences. I learned much more by implementing the Huffman coding algorithm than just reading about it. I enjoyed implementing different aspects of the project to solve different problems, thinking through high-level and low-level ideas for design and implementation, debugging, thinking through logic to rebuild the encoding table, and other aspects of the project. The one thing that I spent a good amount of time on, and I was not expecting to, was character sets. Before this project, I had a general idea of character sets, but I now understand character sets much more thoroughly. This helped me work through my understanding of how the project should work from an input/output design perspective—with the first version of the program only being able to read in text files. Ultimately, understanding character sets helped me understand how to work with bits and bytes of files as opposed to characters that can be seen on the screen. This was probably my biggest setback with this project. Because of this setback, I did not have time to test additional encoding schemes. If I had more time, I would have tested canonical encoding. I was able to generate the canonical codes from the input successfully, but didn't make it past that point. I understand how other encoding schemes could provide improvements upon compression. If I were to start the project from scratch again, I would likely look at other ways to read in bytes/bits from files. Since I already had most of the code written, I continued to use vectors of the type *unsigned char* to store the bytes read in from the given file. I believe this project has improved my experience and understanding of c++, the Huffman coding algorithm, and character sets. I look forward to taking what I've learned with this project and using that knowledge in practice.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009), *Introduction to Algorithms, Third Edition*, 3rd edn, The MIT Press.

Deorowicz, S. (n.d.), 'Silesia compression corpus'.
URL: *https://sun.aei.polsl.pl// sdeor/index.php?page=silesia*

Wikipedia contributors (2022), 'Bzip2 — Wikipedia, the free encyclopedia'. [Online; accessed 4-May-2022].
URL: *https://en.wikipedia.org/w/index.php?title=Bzip2oldid=1085852497*