HED (Huffman Encoder - Decoder) - An Application for Text Encoding and Decoding

Students: Miroslav Z. Manić[1] and Ivan S. Nikolić[1]

Mentors: Radomir Stanković[1] and Dušan Gajić[1]

Abstract – **Huffman encoding is a lossless data compression method often used in practical applications like MP3 audio encoding, JPEG image encoding or ZIP's DEFLATE algorithm. In this paper we present the application we named HED (Huffman Encoder - Decoder) which we developed using C#. This software tool can be used for both text compression and decompression and includes the compression statistics (compression ratio, compression time etc.). Results of the practical experiments conducted with the HED application are also presented and analyzed.**

Keywords - **Huffman encoding and decoding, algorithm implementation in C#.**

I. INTRODUCTION

Lossless data compression is a class of data compression algorithms. Those algorithms allows the exact original data to be reconstructed from the compressed data.

In computer science, data compression is the process of encoding information using fewer bits than a more obvious representation would use, through use of specific encoding schemes. For example, this article could be encoded with fewer bits if we accept the convention that the word "Huffman" be encoded as "Huf". One popular instance of compression that many computer users are familiar with is the ZIP file format, which storing many files in a single output file.

Compressed data communication only works when both the sender and receiver of the information understand the encoding scheme. It means that compressed data can only be understood if we know which decoding method is used by the sender. This is possible because most data that we use have statistical redundancy. Detailed explanation of this is given in [2]. For example, the letter 'e' is much more common in English text than the letter 'z' [3]. Lossless compression algorithms exploit statistical redundancy in such a way as to represented the sender's data more concisely, but nevertheless perfectly. Compression is important because it helps reduce the consumption of expensive resources, such as disk space. On the other side, compression requires information processing power, which can also be expensive. Some schemes are reversible so that the original data can be reconstructed (lossless data compression), while others accept some loss of data in order to achieve higher compression (lossy data compression).

Huffman coding is a statistical technique which attempts to reduce the amount of bits required to represent a string of symbols. The algorithm allows symbols to vary in length. Shorter codes are assigned to the most frequently used symbols, and longer codes to the symbols which appear less frequently in the string. Applications which uses Huffman code are very frequent in computer science. This code is not only for text coding, it is used for picture compression, audio and video compression etc. Huffman coding is useful for compression data where there are bits which are most frequently used.

In this paper we present the application we named HED (Huffman Encoder - Decoder) which is developed using C#. This software tool can be used for both text compression and decompression and under the hood there is Huffman algorithm for data compression.

The paper is organized as follows. In section 2 we will see Huffman coding procedure. The architecture and implementation of Huffman algorithm in our application is given in Section 3. Section 4 is reserved for practical experiments and results which are presented and analyzed. Section 5 summarizes the results.

II. HUFFMAN ENCODING AND DECODING PROCEDURE

Arithmetic coding can be viewed as a generalization of Huffman coding. Although arithmetic coding offers better compression performance than Huffman coding, Huffman coding is still in wide use because of its simplicity.

Encoding

Huffman encoding today is often used as a "back-end" to some other compression method. DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by Huffman coding [1].

Huffman coding uses a specific method for choosing the representation for each symbol. It results in a prefix code that expresses the most common source characters using shorter strings of bits.

Students: [1]Miroslav Z. Manić and Ivan S. Nikolić are with the Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Nis, Serbia, E-mails: mmanic@elfak.ni.ac.rs, slashdance007@gmail.com.

Mentors: [1]Radomir Stanković and Dušan Gajić are with the University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia, E-mails: dusan.gajic@elfak.ni.ac.rs, radomir.stankovic@gmail.com.

First that we need to resolve before we start is "What is a character?". For our implementation a character is any 8-bit combination. In general, a Huffman code for an N characters alphabet, may yield symbols with a maximum code length of N - 1.

The Huffman algorithm works by creating a binary tree of nodes. These can be stored in a regular array. The size of which depends on the number of symbols. A node can be either a leaf node or an internal node. All nodes are leaf nodes, which contain the symbol itself, the link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree. Detailed explanation of this is given in [1]. Here is one example which shows how to create Huffman tree. Given a 6 symbol alphabet with the following symbol probabilities: A = 1, B = 2, C = 4, D = 8, E = 16, F = 32. Step 1. Combine A and B into AB with a probability of 3. Step 2. Combine AB and C into ABC with a probability of 7. Step 3. Combine ABC and D into ABCD with a probability of 15. Step 4. Combine ABCD and E into ABCDE with a probability of 31. Step 5. Combine ABCDE and F into ABCDEF with a probability of 63. Result is shown on Fugure 1.

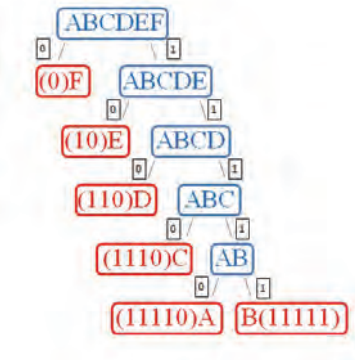

Fig.1 Tree results

Decoding

Generally, the process of decoding is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream. The Huffman tree must be somehow reconstructed if we want to decode aor file. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed. A naive approach might be to prepend the frequency count of each character to the compression stream. Another method is to simply prepend the Huffman tree, bit by bit, to the output stream. For example, assuming that the value of 0 represents a parent node and 1 a leaf node, whenever the latter is encountered the tree building routine simply reads the next 8 bits to determine the character value of that particular leaf. The process continues recursively until the last leaf node is reached; at that point, the Huffman tree will thus be faithfully reconstructed. The overhead using such a method ranges from roughly 2 to 320 bytes. For details see article given in [1]. On the end, the decoding must be able to determine when to stop producing output. In section 3 we presented technic that we used for decompression in our application [5]. Before we see the experimental results for our application, we should be first introduced to the architecture of application and how it works under the hood.

III. Architecture and Implementation of HED

HED application is developed in C#. We used Visual Studio 2010 and .NET Framework 3.5. This is powerfull tool for making applications which have really good design and which works in real time with low lag.

The main task of Huffman Encoder – Decoder (that we named HED application) is compression of text files. Our version of application has decoder, so you can decode file which is encoded with HED and which has no losses.

Design for application was made using Adobe Photoshop. Specifically, we used Microsoft Visual Studio 2010 professional, with .net framevork3.5. Adobe Photoshop was used to design the program and the algorithm is implemented in C#. Application is designed to simulate the Windows Aero interface. Following the example of Microsoft Office wizard to help, we created a robot that appears during the operation of the application and asks the user to simply click it toget help for HED application.

Fig.2 Help Wizard

HED applications can be divided into several units. Based on the Huffman algorithm and its modes, the first thing the program has to do is to count the characters which appear in the entered text. This counter is the first important unit in the HED application. In addition, it makes a Huffman tree. It also

makes code for all the different characters that appear in the entered text. When the application has the code it may start coding.

HED application is made to do compression, called encoding, and also the decompression or decoding of already compressed text. Decoding is implemented with two functions. The first function reads file header which contains data required for decompression and then makes Huffman tree. It's Based on the tree. Compressed data is stored in the rest of the file. The second group of functions perform decompression.

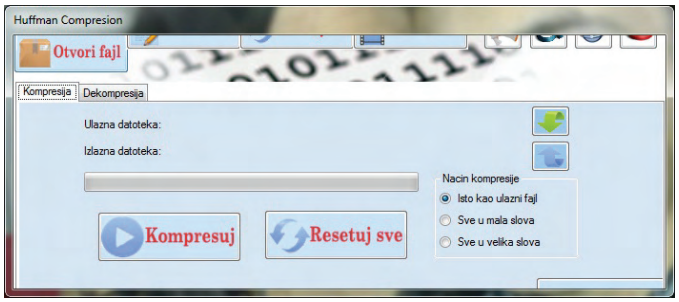

Fig.3 Application interface

First function of the HED application performs counting of different characters which appear in the input text. It consists of "for loops" which do counting in just one pass.This is one of reasons why this program is so fast. The number of character occurrences is remembered in a row, and the characters that appear in the input text of this function are remembered in the second row. Application uses this rows during labor. Building of the Huffman tree is performed by several functions related to the entity. This tree is used for creating the character codes. Each tree node represents a data structure and contains the character and number of its occurrences. One of these functions is used to insert the node to the right place, the second is used to create a new node based on two adjacent nodes.Making accurate Haffman tree is very important because function read character codes used for encoding from it.

Creating codes is perform by special functions. She moves through Haffman tree recursively calls itself and remember in a series of codes based on Haffman algorithm. Codes are stored as a string, but only temporarily. During encoding, these codes are converted to a sequence of bits. This conversion is done with a special function.

This works on the principle of logical shifting buffer to the left. If the input string (a sequence of 0 or 1) logic 1, the functions other than shift to the left adds a 1 at the endof the buffer, if the input string 0 then just shift the contents of the buffer left. Enteringinto the buffer is complete when the buffer is full rapport, when write 8 bits. This function is used for coding the input text. Coding performs a function that reads character by character from text input, read itscode, converts it into a series of bits and stored in the output file.Switch code that is remembered as a string into a sequence of bits perform functions which we write because C # programming language does not support working with bits. This function returns one byte witch is through the buffer stored in the output file.Compression is also performed one-pass through to enter text.

Decompression performed reverse work than the work which is performed by compression. Since some data that are used in the compression is also required for decompression, we must remember them in the outputfile.This part of the file is called a header. It remembers all the characters that appear in the coded text and the number of occurrences (frequency) of these characters. So that all characters were included they must be remembered in UNICODE code system, so each character in the header occupies 2 bytes.The number of occurrences of character is an integer and takes a byte in header file.These data are important because we need to have the decoding Huffman tree from which we read the character of the input code.Creating a Huffman tree performs a function that is very similar to that as incompression, because their input data practically are the same.The very process of decoding performs a function on the basis of already made huffnan trees and coded data, reading the characters and remember them in the output text file.Here we have a function which for easy work turning sequence of bits in the string.

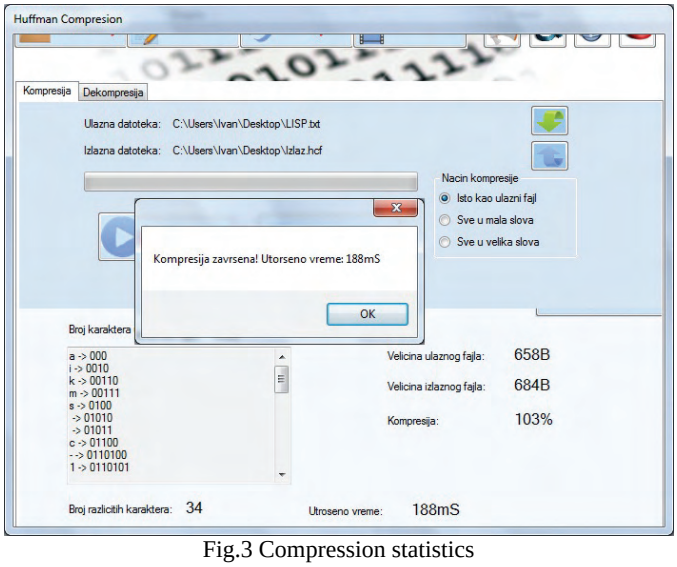

Fig.3 Compression statistics

IV. PRACTICAL EXPERIMENTS AND RESULTS

For testing our application we have used different types of textual files. We used txt, .h, .cpp, HTML, XML, etc. Results shows that there is no matter which format of text document is, it works with all of file formats we mention above, and size of output file does not depend on type of text file. Size of output file directly depends on size of input file. On the other side, characteristics if letters in text have higher frequency is one of the most important reasons why output file is smaller or bigger. So, we have made some txt files and tested our application using them. Some of them have few characters, other have different characters. In the following table we can see some results how output file depends on number of characters and frequency of its occurence.

TABLE I
VALUES IN BYTES AND NUMBER OF CHARACTERS FOR INPUT AND OUTPUT FILES AND COMPRESSION RATE AND TIME ELAPSED FOR OUTPUT FILES

File No.	Input file/Output file [B]	Input No.diferent char./All characters	Compression ratio [%]	Time
1.	5/14	2/2	280	11 ms
2.	10/28	4/10	280	12 ms
3.	10/74	10/10	740	21 ms
4.	21/124	15/21	590	27 ms
5.	39/44	5/39	112	12 ms
6.	1.926/1.617	49/1.926	83	140 ms
7.	11.529/9.062	86/11.529	78	334 ms
8.	103.555/52.709	116/103.408	50	839 ms
9.	3.362.038/963.113	90/1.681.018	28	5.061 ms
10.	4.963.584/2.453.939	116/4.963.584	49	17.713 ms
11.	65.078.895/37.621.271	116/64.851.952	57	3 min 48 s
12.	193.547.880/96.253.836	22/193.547.880	49	3 min 11 s

Now we will discuss results that we got. This is tasted on Intel Dual core processor and it's important for following discussion about duration of this operations. First off all, we can notice that size of output file depends of number of different characters. In file 1 we have only 2 different characters and whole file have only 2 characters. So we can see that result for this is not so good, because output file is bigger than input file. Compression ratio here is 280% (Table 1).

When we have same number of different characters and all characters like in file 3, results are worst and output file is much bigger then input. In file in which number of all characters is more bigger then number of different characters, compression ratio is better (112 %). This is because codes for characters are shorter then ASCII code, and they occurs frequently. We must mention again that output files should be smaller if we didn't used header. Better results are in file 6 (112% compression ratio) because we have larger number of characters then number of different characters. You can see this in other files (7 - 12). Compression ratio is greater and when number of all characters increases. We tasted application with maximum 116 different characters, and best result is in file 10 (49% compression ratio) which has 4.963.584 different characters. In table 2 we can see duration of each compression process. We can notice that the best time is for file 1, and the worst for file 12.

V. CONCLUSION

In this paper An Application for Text Encoding and Decoding is proposed. It uses Huffman algorithm. Experimental results shows that compression ratio depends on ratio of number of different characters and number of all characters in file; when number of different characters is lower, and number of all characters is higher, results are better. Time for compression is lower when we have less number of all characters.

This application can not be compared with other commercial programs that doing encoding and decoding data. It can be used in educational purposes to show students how Huffman algorithm works.

REFERENCES

[1] The Data Compression Book 2nd edition by Mark Nelson and Jean-loup Gailly, M&T Books, New York, NY 1995

[2] Text Compression by Timothy C. Bell, John G. Clearly and Ian H. Witten 1990

[3] Data Compression: The Complete Reference by David Salomon 2nd edition, 2000

[4] http://www.cs.cf.ac.uk/Dave/Multimedia/node210.html, website last visited on 13/04/2011.

[5] http://www.cs.auckland.ac.nz/~jmor159/PLDS210/huffman.html, website last visited on 14/04/2011.

[6] http://www.huffmancoding.com/my-family/my-uncle/huffman-algorithm, website last visited on 12/04/2011.