Umeå University
Institution för Datavetenskap

Datavetenskapens byggstenar 7.5 p
DV160HT15

OU3 Huffman Coding

Submitted 2016-02-18
Author: Simon Andersson (`dv15san@cs.umu.se`)
Lorenz Gerber (`dv15lgr@cs.umu.se`)
Instructor: Lena Kallin Westin / Erik Moström / Jonathan Westin

Contents

1 Introduction

The aim of this laboration was to plan and implement a command line program written in C that accomplishes encoding and decoding of text files according to the *Huffman* algorithm [1].

The *Huffman* algoritm is used for data compression, in our case for text files. The basic idea is that instead of using 8 bits for every character, find unique variable length binary representations for every character where the most common used characters get the shortest binary sequences. In practice, this includes several steps: First a frequency count table of all characters has to be compiled. Then a binary tree is constructed where characters and their count frequency as leafs. Characters with high frequency count will be placed the closest to the tree root. A more detailed description of this process will be given in the method description.

2 Material and Methods

2.1 Datatypes

The provided datatypes were all implemented according to descriptions in Janlert and Wiberg [2]. We used *prioqueue* (which is built on *list_2cell*), *tree_3cell*, and *bitset*. Further, we defined a *struct* called *freqChar* to store a character and it's associated frequency. This datatype was used as label in the Huffman tree.

2.2 Work Organization

On an initial kick-off meeting, we discussed the problem and possible solution strategies. Then we created repositories for the *code* and for the *report* on *github* and setup a team in a workgroup messaging app (*slack*). All further work and communication was done remote using the afore-mentioned tools.

3 Results

3.1 Test runs

A test run using the provided file *löremipsum.txt* for the frequency analysis to encode and then decode *balenPaEkebyGostaBerlingsSagaSelmaLagerlof.txt* resulted in a decoded text which matched the original in every position. The program executed without errors. Compression values are shown in table 1. Testruns with *valgrind* confirmed that there were also no memory leaks.

Several other test runs have been performed using different, smaller .txt files as well. These include tests on files with every character 'a-ö', files with every ASCII sign and various shorter texts. All of these test runs produced correctly encoded/decoded files without problems.

The program has also been tested with too few/incorrect parameters as well as files which could not be opened. These tests resulted in the program exiting with the correct error messages as was to be expected.

Table 1 File compression values. The *Löremipsum* text was in both cases used to determine the frequency table.

(all values in bytes)	before	encoded	decoded
loremipsum	7’815	4’464	7’815
lagerlof	33’611	22’847	33’612

3.2 Program structure

Essentially this program consists of six different *central* functions. These are complemented by a few simple functions to print error messages et cetera. The functions are called from main depending on command line parameters input by user and apart from those parameters there is no user interaction. Following is a brief description of the six functions as well as a few words about the command line parameters.

Parameters

This program takes four parameters from the command line. The first one determines whether the user wants to *-encode* or *-decode* the input file. The remaining three parameters represent three files where the first one is the file upon which the frequency analysis will be done, the second one is said input file and the third file is where the result will be saved.

If user enters option *-encode* the functions *getFrequency*, *buildHuffmanTree*, *traverse-Tree*, and, *encodeFile* will be called to encode the input file and save the result in the output file. If *-decode* is sent as a parameter *getFrequency*, *buildHuffmanTree*, and, *decodeFile* will be called to decode text from the input file and save the decoded text in the output file. And lastly if the user were to give an incorrect option or if at least one file cannot be opened the program will exit with an instruction on how to properly use the program.

getFrequency

In *getFrequency* frequencies for characters read from the frequency file are calculated. All frequencies are stored in an array of size 256, enough to store every character in ASCII, where indexes 0 through 255 represent the different symbols. Even though it is probably not read from the file the symbol with ASCII code 4 is given frequency 1. This is because this character is the *end of transmission* sign which will be written during encoding no matter what text is encoded.

After the frequency analysis we implemented a short code snippet to handle the case of zero frequencies. The code first multiplies each frequency value by 1000 and adds then 1 to every zero frequency. The implications of this code will be treated in the discussion.

compareTrees

This function is the function required by the *prioqueue* to correctly enqueue and dequeue trees from the priority queue. The labels of two root nodes in two different binary trees is inspected. The frequency value in these two *freqChars* are compared and the tree with the lower value is given higher priority.

buildHuffmanTree

Now we have all the tools necessary to construct the Huffman tree. This function uses the provided data types *prioqueue* and *tree_3cell*. From the frequency array 256 trees are created each with a single node labeled with a *freqChar* struct storing one of 256 ASCII characters and the frequency value for that character. These trees are then enqueued into a priority queue using the *compareTrees* function so that the trees with the lowest frequencies have the highest priority.

Then two trees are taken from the front of the queue and merged into a new tree with a parent root node storing the sum of frequencies of these two smaller trees. After this the new tree is enqueued again and the process is repeated until only one tree remains in the queue. When this occurs every character has its place in a single Huffman tree thus completing the tree which is returned.

traverseTree

Once the tree is built we need to extract binary sequences for every character from it. This is done by using the *traverseTree* function.

traverseTree is a recursive function performing a preorder traversal of a binary tree. In addition to the binary tree the function takes a binary tree position, an array to store bit sequences, and, an array containing pointers to 256 bitsets as parameters. The function will check if the current position in the tree has left and/or right children. If a left children exists a zero will be written in the array and the function will make a recursive call with the position of this child. The procedure is the same if a right child exists except that a one is written in the array instead of a zero. Once a childless position is found the bit sequence is inserted into a bitset whose pointer is stored in the pointer array and once every leaf has been visited the function is complete.

encodeFile

encodeFile is the function responsible for encoding the input file. This function will read one character at a time from the input file . From the bit-set-pointer array the correct bitset is chosen. The bit sequence from this bitset is inserted into a new bitset which will store all bit sequences for the entire input file. Last in the sequence an *end of transmission* sign is placed. Once every character has been read and stored as a bit sequence in the compound bitset the function will save the whole compound bitset in the output file.

decodeFile

In *decodeFile* every symbol in the input file is read and converted back to binary sequence. This sequence is stored in a boolean-array as ones and zeros. The sequence is then used to wander through the Huffman tree. For every ‘1’ read we move right and for every ‘0’ we move left until a leaf is reached. Once we reach a leaf the character stored in that leaf’s *freqChar* is read and written in the output file. When *end of transmission* is reached the decoding ends.

4 Discussion

4.1 Zero Frequency Chars

As mentioned in the course homepage, zero frequency characters have special effect on the huffman tree building procedure. Usually, adding two trees with the lowest value would raise the new tree to a higher value. However, this doesn't happen with zero value trees until they get combined with such that have a value larger than zero. Practically, this lead to trees where all inital zero value chars end up in the same branch as a long tail. Our initial idea to avoid this scenario was to raise all zero frequncies to value one. However, this would slightly modify the distribution of characters with low frequency. Whether this is a problem or not, would have to be determined in more depth, however we came up with a solution that circumvents this situation. After determining the frequencies, all values are multiplied by 1000, which does not affect the zero values but keeps the order among the chars with a frquency value of 1 and higher. Then all zero values are set to value 1. This will basically result in two well shaped subtrees.

4.2 Bitset operations

The implemented datatype *bitset* has an operation to *export* the bitset into a *char* array. We used this operation to prepare the data for writing to file. However, there is no matching operation implemented to *import* a char vector into a *bitset*, which would have been our prefered solution. For simplicity, we did not modify the datatype *bitset* but used a double *for* loop with a *bitshift* operator to convert the encoded data back.

4.3 File operations

Currently we use the functions *fgetc()*, *fputc* and *fprintf* for file I/O. In *getFrequency()* and *encodeFile()* we open the file stream with the argument 'rt' while in *decodeFile()* we open it with 'rb'. The latter was necessary to use the functions *fseek()* and *ftell* for determining file size. File size was neccessary to set the upper limit of the read loop and to allocate dynamic memory for the bit sequence to be converted. This is probably not the sleekest possible solution, however we tested several others that did not work as reliable as the implemented one.

5 Conclusions

We are satisfied with our program. The encoding and decoding work as intended and all of our test runs went well. Perhaps we should have implemented more checks for errors in parameters and files but those in the code have been sufficient for every test we have performed. Another area to improve upon might be the use of *freqChar* which turned out to be a bit awkward, it is possible another data type would have been easier to use although *freqChar* is simple and easy to understand. Overall the work progressed without any major problems.

6 Contributions

Both authors were involved in every function with either writing or debugging/checking it. The initials in table 2 stand for the person who initially wrote the respective function.

Table 2 work contributionsn

planning and defining the strategy	SAN, LGR
setting up and managing git repos	LGR
initial code structure	SAN, LGR
main program and argument handling	LGR
char frequency count	SAN
compare tree function	SAN
build Huffman tree	SAN
tree traversal function	SAN
huffman code from tree traversal	LGR
encode function	LGR
decode function	SAN, LGR
file read/write	LGR
commenting and styling code	SAN, LGR
memory leak check	SAN
setting up LaTex document	LGR
writing report	SAN LGR

References

[1] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E*, 40(9):1098–1101, 1952.

[2] L.E. Janlert and T. Wiberg. *Datatyper och algoritmer*. Studentlitteratur, 2000.