**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.

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

From the provided datatypes we used *prioqueue* (which is built on *list_2cell*), *tree_3cell*, and *bitset*. Aside from the provided datatypes another composite datatype, *freqChar*, storing a character and associated frequency was created. This datatype was the one stored as data 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

blablabla

# 4 Program structure

## 4.1 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*, *traverseTree*, 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 in the input file and save the decoded text in output file. And lastly if the user were to give an incorrect option or if the files cannot be opened or in some way return errors the program will exit with an instruction on how to properly use the program.

### 4.2 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. Once the analysis is done every position with a frequency of 0 is set to 1 to avoid problems while building the tree later on. To account for this *real* frequencies are multiplied by 1000 to keep their priority over these artificial values.

### 4.3 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 a lower value is given higher priority.

### 4.4 buildHuffmanTree

Now we have all the tools necessary to construct the Huffman tree. 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 dequeued and merged into a new tree with a root node storing the sum of frequencies of these two smaller trees. After this the new tree is enqueued again and the process repeats until only one tree remains in the queue. Every character now has its place in a single Huffman tree. The tree is complete and gets returned.

### 4.5 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 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 0 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 1 is written in the array instead of a 0. Once a childless position is found the bit sequence is inserted into a bitset whose pointer is stored in the pointer array.

### 4.6 encodeFile

*encodeFile* is the function responsible for encoding the input file. This function will read a character 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 a *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.

**Table 1** 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 |

### 4.7 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 tree. For every 1 read we move right and for every 0 we move left until a leaf is reached. Once we reached 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.

## 5 Discussion

bla bla bla...

## 6 Contributions

Both authors were involved in every function with either writing or debugging/checking it. The initials in table 1 stand for the person who initially wrote the respective function.

## References