Politechnika Śląska

SILESIAN UNIVERSITY OF TECHNOLOGY

Faculty of Automatic control, Electronisc and Informatiscs

Fundamentals of computer programming

«Huffman»

Author: **Andrii Bobchuk**

Instructor: **dr inż. Agnieszka Danek**

Year: 2020

Deadline: 08.11.2020

1 Project topic

Implement a program that compresses files with the Huffmann's method.

The program is run in command line with switches:

-i to specify the input file

-o to specify the output file

-m to specify the mode: **c – compress, d – decompress**

-d to specify the dictionary file (created in compression, used in decompression)

---

2 Analysis of the task - 2.1 Data structures

The main rule in Huffman file compression:

THE MORE A CHARACTER IS MENTIONED, THE SHORTER ITS SIZE SHOULD BE.

A normal char takes 1 byte or 8 bit to encode it according to ASCII table, but according to Huffman we need to decrease the size of the most frequent characters.

For instance knowing that "Space" is the most frequent character in most text files we should assingn it the shortest Hufman code. Thereby instead of long 00100000 (Which is a binary code of "Space") we get a short 101 (just an example).

Nevertheless it causes one HUGE problem - reading and recognizing symbols. When computer wants to translate binary code into a normal text it separates the whole code by 8 digits to convert those small groups into letters, spaces, special sybols etc. However according to the Huffman, all characters have variable bit code size, so how to read it the right way then?

Answer: You cannot read the Huffman code the wrong way! It is a **prefix code** which means none of the codeword is the beginning of another codeword. And it's all thanks to the tree system which assigns codes to characters (By the way - the shortest possible codes).

To implement this task in C++ we will need:

Containers:

Maps – to hold character and its frequency (int) or binary code (vector<bool>);

Vectors – to process encoded text we've read from the text file, to store nodes, to keep Huffman codes assigned to each character;

Classes:

Compare – includes parameters for comparing nodes before sorting

Node – which will include the character and its frequency and be a part of the tree of nodes

Filestreams:

Ifstream/ofstream – to read and write files

---

2 Analysis of the task - *2.2 Algorithms*

Program has 2 main tasks: compress and decompress files. For both of these reasons there are different sets of functions.

To compress file:

1) Open the input file, count the frequencies of each char in input file using associative array "MAP" with two fields of types char and int, write it to a dictionary file (it is much easier (and way more efficient) to use the dictionary file of type "(int)character:frequency" than "int(character):code" because we are going to recreate the tree before decompressing and the fastest way to do this is with the first type of dictionary). By the way, we write character as integers to file (to avoid issues on writing/reading)
2) Based on the frequencies of chars generate nodes "Character-Frequency"
3) Write nodes to the list
4) Sort the list
5) Take two nodes from left
6) Create a parent node which includes sum of their nodes
7) Put a new node to the list
8) Reapeat step 4-7 untill we have just 1 element – it will be the root of tree
9) Generate a dictionary – if we go to the left child, we assign 1, else 0. When we reached the leaf node we should write the code and appropriate char to the map.
10) Take an input file and substitute each character with the value from the dictionary

To decompress file:

1) Open the dictionary file and convert it to a map <int, int> (character:frequency)
2) Again - Based on the frequencies of chars generate nodes "Character-Frequency"
3) Write nodes to the list
4) Sort the list
5) Take two nodes from left
6) Create a parent node which includes sum of their nodes
7) Put a new node to the list
8) Reapeat step 4-7 untill we have just 1 element – it will be the root of tree
9) Now we have tree, let's go through it while reading file and turn to the left or right if we see 0 or 1.

10) When the current node of the tree doesn’t have children then we should print its value - char (The approach with walking through the tree is MUCH faster than continuous lookups in biary code for similar codewords as in dictionary)

---

3 External specification

Program is launched in command line. It requires to type the name of an input file after the –i switch; name of an output file after the –o switch; name of the dictionary file after the –d switch; and mode (c for compress, d for decompress) after switch –m.

Here is an imput structure:

Huffman.exe -i *[name_of_file_to_be_encoded.txt]* **-o** *[name_of_the_decoded_file.bin]* **-m** *[mode: c\d]* **-d** *[name_of_the_dictionary_file.bin]*

If the user wants to compress file A into file B and generate a dictionary C he has to type:
Huffman.exe -i A.txt -o B.bin -m c -d C.bin
To decompress file B into file A using an existing dictionary C user has to type for example:
Huffman.exe -i B.bin -o A.txt -m d -d C.bin

Dictionary and compressed files are binary. Input can be any file. The switches may be provided in ANY order.

The program called with **no parameters or with parameter ‘-h**’ will print a short manual:

```
| Tutorial:                                            |
| type '-i' to specify the input file;                 |
| type '-o' to specify the output file;                |
| type '-d' to specify the dictionary file;            |
| type '-m' to specify the mode:                       |
|                                'c' to compress;      |
|                                'd' to decompress;    |
```

Program called with mistakes in arguments will print a **general error message:**

```
|====================== ERROR!! ======================|
| You made a mistake typing command line arguments!    |
| Tutorial:                                            |
| type '-i' to specify the input file;                 |
| type '-o' to specify the output file;                |
| type '-d' to specify the dictionary file;            |
| type '-m' to specify the mode:                       |
|                                  'c' to compress;         |
|                                  'd' to decompress;       |
| NOTE: You may pass arguments in ANY order                 |
|===========================================================|
```

Program called with **non-existing input filename** will print the first table AND:

```
Error: Cannot read input file
```

Program called with **non-existing input filename** will print the first table AND:

```
Error: File is empty
```

Program called with **wrong mode specified** will print the first table AND:

```
Error: Mode is not specified
```

---

4 Internal specification - *4.1 Program overview*

Program starts with the main function. Depending on the user input program can print a manual, a general error messege or/and point to some specific input mistakes. If everything if correct the program call either compress function or decompress. Those functions are located in functions.cpp file.

When we call **"compress(InFile, OutFile, Dictionary);"**:

(we pass the input, output and a dictionary filename.)

At first function calls another function **countFrequencies(In, freqTable);** which counts the frequencies of characters in opened input file, put the data to a map which was passed as a parameter.

Then function "compress" calls the function **fillNodeList(Dict, freqTable, nodePointersList);** which fills the list of nodes

After that function "compress" calls the function **createHuffmanTree(nodePointersList);** which makes parents nodes, sort the list and obtain a root in the end.

After that function "compress" calls the function **DictionaryGenerator(root, dict, code);** which makes pairs "character:code" required for encoding.

Then we do the encoding substituting the chars with codes and write it to a binary output file.

When we call **“`decompress(InFile, OutFile, Dictionary);`”**:

(we pass the input, output and a dictionary filename.)

At first function converts the dictionary file to map of ints, then calls the function **`fillNodeList(Dict, freqTable, nodePointersList);`** which fills the list of nodes

After that function “decompress” calls the function **`createHuffmanTree(nodePointersList);`** which makes parents nodes, sort the list and obtain a root in the end.

Then we do the decoding just by walking through the tree picking either 1 or 0 untill we get the current symbol.

After that function writes the decoded file.

4 Internal specification *- 4.2 Description of types and functions*

Description of types and functions is mived to the appendix

4 Internal specification *- 4.3 Testing*

The program has been tested on .txt, .png and .jpg files. Obviouslyt he compress ratio is the best while working with .txt files (approximately 50%)(Because Huffman algorithm was designed for text files mainly), but when it comes to photos the compressed bin files are only about 20% lighter than the original files.

Handling errors. Program will notify if you:

- Entered too many or too little arguments;
- Entered the incorrect mode;
- Entered a non-existing input file;
- Want to compress an empty file
- Did anything else the wrong way

4 Internal specification **-** *4.4 Conclusion*

Command line interface encoder which helps to save space on the hard drive using Huffman algorithm (Best compression for txt files)