Data Compression Coursework I

Student: Gabriel Pires e Albuquerque de Mello - 3331638501
Professor: Ida Mengyi Pu

1. Introduction

The purpose of this coursework is to practice the Huffman Coding algorithm and Extended Huffman Coding, as well as compare the efficiency between these two approaches of compression.

Basically, what Huffman Coding does is decrease the amount of bits needed to represent each specific data symbol making use of a complete binary tree - or Huffman tree. The extended version of this algorithm tries to improve the coding efficiency by combining two or more symbols to encode.

As result, Huffman Coding generates a code unique decodable, as consequence of complete binary tree.

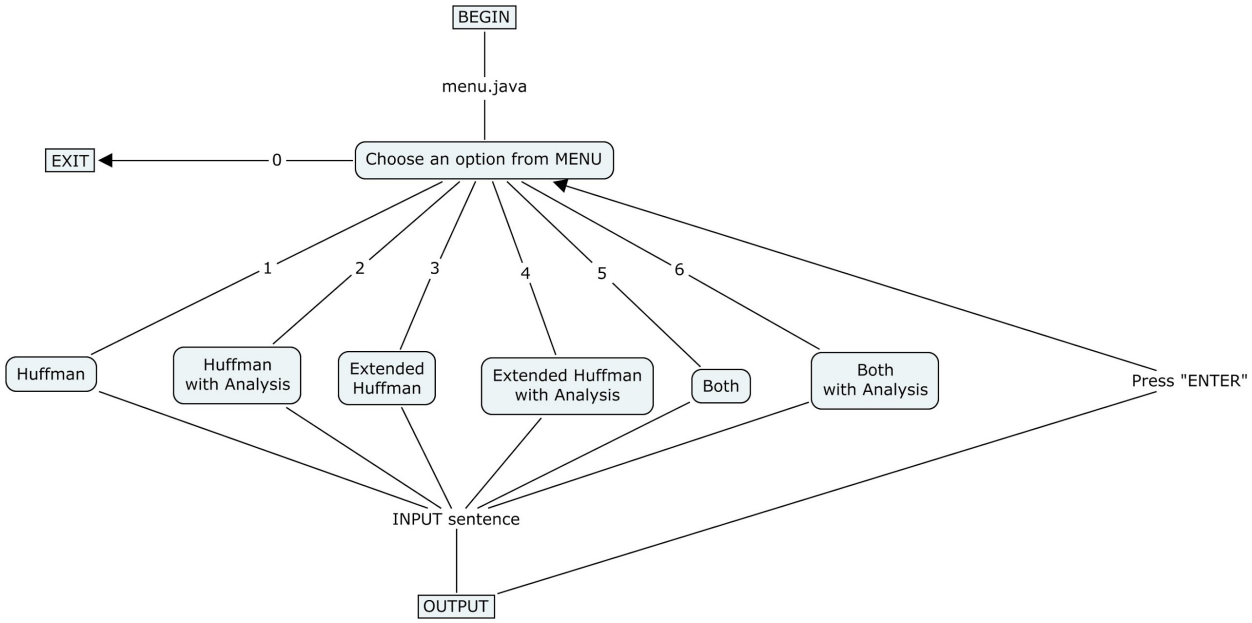
2. Approaches

The program contains 4 java files and 5 classes, being “menu.java” the main Class. Only Standard Java Libraries[1] were used in this implementation, which are:

- *java.util.ArrayList;*
- *java.util.Collections;*
- *java.util.List;*
- *java.io.BufferedReader;*
- *java.io.InputStreamReader;*

[1] All other implementations were made by the author of this coursework itself, which guarantee the no presence of plagiarism.

Running the *menu* file, the following flux will occur:



Summing up the options, it is possible to run each algorithm separately or together with or without analysis.

Symbol class

The *Symbol* class is the most important class to be understood. This class is declared in the file *HuffmanCompression* and a *Symbol* contains the following informations:

- *symbol* - The data symbol itself stored as a *String*;
- *frequency* - The amount of times this data symbol appears in the data;
- *code* - Stores the encoded string (0's and/or 1's) for this symbol;
- *probability* - Based on the frequency of this symbol.

Input

To run Huffman or Extended Huffman Coding, we need a sentence to be encoded. The sentence must contain only ASCII characters and no line break, once that a line break will trigger the end of input process.

After a sentence is readed from the keyboard, a pre-process begins. This pre-process consists in create a *List* of *Symbols* adding non-existents symbols to the list, or increasing the frequency of the existing ones. After allocating the whole input sentence, the probability of each symbol is calculated following the equation:

$$probability = \frac{frequency}{numberOfSymbols}$$

Note that the *code* variable still unused for each symbol, once this will be allocated while running the Huffman and/or Extended Huffman.

Huffman Coding

The implemented Huffman Coding algorithm can be find in the file "*HuffmanCompression.java"*. As input, it receives a *list* of *Symbols* and as output it will generate and store the binary code of each *Symbol*.

To reach this aim, the following steps are made:

1. ***Sort*** the *symbol list* - using *Merge sort* from *Collections* library;
2. **S*tacks*** the whole *list* following the sequence:
    a. The last two *symbols* from the *list*;
    b. The combination of the last two *symbols* from the *list* with the updated frequency and probability;
    c. Remove the last two *symbols* from the *list*;
3. ***Unstack*** in order to create an Array as abstraction of a ***Binary Tree***;
4. ***Encode*** each *symbol* of the *list* storing it in the *code* variable;

Extended Huffman Coding

The Extended version of Huffman Coding follow the same steps of Huffman Coding, with one divergence: A new input list is generated with the combination of two *symbols* from the original *list*, as well as its frequency and probabilities;

Analysis

The analysis consists in:

- For original INPUT:
    - Calculate size in bits;
    - Calculate entropy;
- For Huffman Coding and Extended Huffman Coding:
    - Calcula size in bits;
    - Calculate saving bits and its percentage related to the original input size;
    - Calculate Average Length;
    - Calculate Coding Efficiency;

3. Demonstration

The program begins in a menu section, as described already:

```
|
2910325 Data Compression coursework
 by Gabriel Pires e Albuquerque de Mello

        ********************
        Declaration: Sorry but part of the program was copied from the Internet!
        1. Huffman Code
        2. Huffman Code with Analysis
        3. Extended Huffman Code
        4. Extended Huffman Code with Analysis
        5. Both Algorithms without Analysis
        6. Both Algorithms with Analysis

        0. Exit
        ********************
Please input a single digit (0-6):
```

As demonstration, the following examples will be executed:

- Huffman Code.
    - Input: “BILLBEATSBEN”;
- Extended Huffman Code
    - Input: “BILLBEATSBEN”;
- Both Algorithms with Analysis
    - Optimal case for Huffman. Input: “aabbccdd”
    - Worse case for Huffman. Input: “aaaaabaaaabcaaabcd”;
    - Slightly bigger input - 10000 letters randomly generated with alphabet [a, b, c, d];

Huffman Code

As can be seen in the image below, when the option “Huffman Code” is chosen from the menu, the user must write a sentence and press enter.

```
1
Write a sentence to be compressed:
BILLBEATSBEN

Generated code by Huffman:
B: 10 (0.25)
L: 001 (0.16666667)
E: 010 (0.16666667)
I: 011 (0.083333336)
A: 110 (0.083333336)
T: 111 (0.083333336)
S: 0000 (0.083333336)
N: 0001 (0.083333336)

Input sentence: BILLBEATSBEN
Huffman encoded sentence: 10011001001100101101110000100100001

Press ENTER to go back to menu!
```

As output, each data symbol is printed in the screen followed by its code and probability in brackets. After, the Input sentence is printed followed by its encoded form.

Extended Huffman Code

The image below shows a cutted part of each combined data symbol printed in the screen followed by its code and probability in brackets. After, the Input sentence is printed followed by its encoded form.

```
AI: 111111 (0.006944445)
AA: 0100110 (0.006944445)
AT: 0100111 (0.006944445)
AS: 1011000 (0.006944445)
AN: 1011001 (0.006944445)
TB: 010100 (0.020833334)
TL: 101111 (0.01388889)
TE: 0000000 (0.01388889)
TI: 0000001 (0.006944445)
TA: 0101010 (0.006944445)
TT: 0101011 (0.006944445)
TS: 1011100 (0.006944445)
TN: 1011101 (0.006944445)
SB: 010110 (0.020833334)
SL: 110001 (0.01388889)
SE: 0000010 (0.01388889)
SI: 0000011 (0.006944445)
SA: 0101110 (0.006944445)
ST: 0101111 (0.006944445)
SS: 1100000 (0.006944445)
SN: 1100001 (0.006944445)
NB: 011100 (0.020833334)
NL: 110011 (0.01388889)
NE: 0010100 (0.01388889)
NI: 0010101 (0.006944445)
NA: 0111010 (0.006944445)
NT: 0111011 (0.006944445)
NS: 1100100 (0.006944445)
NN: 1100101 (0.006944445)

Input sentence: BILLBEATSBEN
Extended Huffman encoded sentence: 011011001101100010011101011010100 1
```

Both Algorithms with Analysis

- Optimal case for Huffman:

Huffman Coding is optimal when its probabilities are powers of 2. In this case, its code Average Length will meet the Entropy. To illustrate this situation, the output generated from "*aabbccdd*" is showed below.

```
                ::ANALYSIS::

        :INPUT:
Sentence: aabbccdd
SIZE: 64 bits
Entropy: 2.0

        :HUFFMAN:
Coded sentence: 0000010110101111
SIZE: 16 bits
Saving bits: 48.0 (75.0%)
Average length: 2.0
Coding efficiency: 100.0%

        :EXTENDED HUFFMAN:
Coded sentence: 0000010110101111
SIZE: 16 bits
Saving bits: 48.0 (75.0%)
Average length: 2.0
Coding efficiency: 100.0%
```

As can be seen, the Coding Efficiency for both Huffman and Extended Huffman are 100%, once the Average Length equals the input entropy.

- Worse case for Huffman

Logically, when the input probabilities are not power of 2, Huffman Coding begins to present less Coding Efficiency. To illustrate this situation, the following output was generated from the sentence "*aaaaabaaaabcaaabcd*":

```
                ::ANALYSIS::

        :INPUT:
Sentence: aaaaabaaaabcaaabcd
SIZE: 144 bits
Entropy: 1.4046784642543921

        :HUFFMAN:
Coded sentence: 000001100001110000011100101
SIZE: 27 bits
Saving bits: 117.0 (81.25%)
Average length: 1.5
Coding efficiency: 93.64523%

        :EXTENDED HUFFMAN:
Coded sentence: 11010110011110100001101
SIZE: 23 bits
Saving bits: 121.0 (84.02778%)
Average length: 1.4583334
Coding efficiency: 96.32081%
```

As can be seen, in this case the Coding Efficiency for Extended Huffman is slightly better, once the Average Length is close to the input entropy. For this input, Extended Huffman saves 4 bits more than Huffman Coding.

- Slightly bigger input

As last demonstration, a slightly bigger input created in an online generator[2] was used as input.

```
                ::ANALYSIS::

        :INPUT:
Sentence:
SIZE: 80016 bits
Entropy: 2.0021817123586043

        :HUFFMAN:
Coded sentence:
SIZE: 22475 bits
Saving bits: 57541.0 (71.911865%)
Average length: 2.2470505
Coding efficiency: 89.10266%

        :EXTENDED HUFFMAN:
Coded sentence:
SIZE: 21211 bits
Saving bits: 58805.0 (73.49155%)
Average length: 2.123924
Coding efficiency: 94.26805%
```

[2] http://www.dave-reed.com/Nifty/randSeq.html

Note that both input sentence and coded sentence could not be printed in the screen, even though the algorithms runned normally. The output is bit more impressive, reaching more than 4% in Coding Efficiency.

4. Discussion

This was a great experience as my first Java development, and as one of the most famous programming language[3], it is not at all a waste of time to learn it. This language showed itself very practical, since it has many useful standard libraries, and a simple syntax based on classes. For sure my development structure could be better, but I realized this as I was experiencing.

About the coursework, the time available for development was perfect. I could conclude the coding even with more 4 others courseworks ongoing. The assignment made me feel some lack of information, but I am used to have specific input and output to work with, when it is no gave as an instruction it become a bit hard to perceive if I am achieving the assignment expectations or not.

[3] http://langpop.com/