Huffman coding

Christopher Marshall-Breton

Presentation

The goal of this project is to implement a compression algorithm. This means that we are trying to reduce the space taken up by information, without losing data.
We will work here on text files.
Huffman's coding relies on translating a character into a shortcode, depending on how often it occurs. The more often a character appears in the text to be encoded, the shorter its translation will be.
You may know that a character is encoded on a byte. This means that for each character in a text we need 8 bits in memory. A text of 100 characters will therefore take 800 bits in memory. How then do we reduce this figure?

Introduction

Working on bits directly would be counterproductive here, since we are mainly interested in how Huffman's code works. We will therefore "visually" represent the bits in characters 0 and 1.

The first step of the project will be to translate a classic text into a series of 0s and 1s, coding each letter on one byte.
Example: the letter '**a**' will first be translated by the string "**01100001**". This will let us know, by counting the numbers, how many bits our text originally takes.

We will then translate it using Huffman's code, to compare and estimate the space gained.

Principle

The goal is simple. By default, we code each character on one byte, so 8 bits. This means that for a text of 1000 characters, 8000 bits are necessarily required. How do we reduce this number as much as possible?

We will divide into 3 stages:

The occurrences

If we have to encode all the characters in the ascii table, there is a good chance that we cannot reduce to less than 7 or 8 bits per character. But in the majority of texts, only a few symbols from this table are used. We will therefore be able to redefine a correspondence according to the characters present in the text.
Example: if we only have **A**'s and **L**'s, we can code **A** by **0**, **L** by **1**, and that's it. We have thus gained **7** bits on each letter, a text of **100 A** and **L** will therefore take **100** bits instead of the **800** necessary previously.
For the text "**AAALALALALLALA**", we go from **112** bits (14 * 8) to **14** bits.

Of course, with more letters, it gets more complicated.
If we only have **P**'s of **K**'s and **D**'s, we can code **P**'s as **0**, **K**'s as **11**, and **D**'s as **10**.

Note : note that no letter is coded by a **1**. We will come back to this in the tree section.

With the following dictionary,

P	**0**
K	**10**
D	**11**

We gain **7** bits on the **P**, **6** bits on the **K** and **6** bits on the **D**.
That said, a text with millions of **K**'s and **D**'s, and a single **P** will gain less space with the same dictionary than a text with millions of **P**'s and few **K**'s and **D**.

For the text "**PPKKDPKDPKDPKP**", we go from **112** bits (14 * 8) to **22** bits (6 * 1 + 5 * 2 + 3 * 2).

Indeed, the space that the compressed text will take will be:

$$\Sigma\ \mathbf{(o * b)}$$

with **o** the number of times the character appears, and **b** the space it takes to represent in bits.

In short, the more often a character appears, the less space it needs to be shortcoded to!

We therefore absolutely need a list containing all the **characters** present in the text, as well as their **occurrences**.

Example :

Alice.txt	Characters	Occurrences
`Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do`		20
	n	10
	i	9
	t	9
	e	8
	o	7
	g	6
	r	4
	s	4
	a	4
	h	4
	b	3
	d	3
	f	2
	v	2
	y	2
	k	1
	A	1
	l	1
	w	1
	c	1

Note that space is the character that appears most often, so it should be encoded with the shortest possible binary sequence.

The tree

Now that we have the characters to represent, and their occurrences, we can finally work on the principle of Huffman's code.
Everything depends on how to assign each letter a binary sequence, according to its **frequency** and **without prefix**!

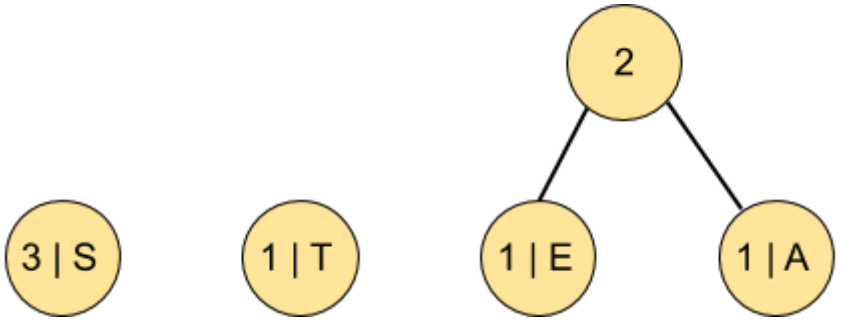
Why without a prefix ?
Let's take the example of a text containing only **P**, **K** and **D**.
We cannot code the **P** by **0**, **K** by **1** and **D** by **10**, because in this case, we cannot distinguish **KP** from **D** (both represented by **10**).
Likewise, if a letter is coded by **XXX**, another letter cannot be coded by **XXX0** or **XXX1**. Otherwise, there is no way to differentiate one letter from a combination of two others.
Therefore, one character cannot be encoded as a prefix of another.

Why by frequency ?
We saw in the previous section that a letter appearing many times must be encoded by the shortest possible binary sequence.

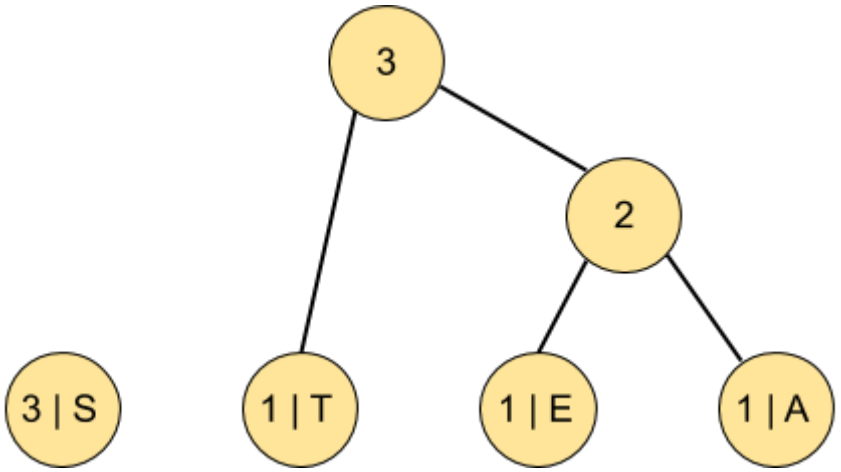
We are therefore going to build a binary tree by putting the characters appearing least often the deepest.
In other words, we will have to "build the tree up" from its end with the weakest occurrences, towards the root with its strongest occurrences.

Example: With the text "**ASSETS**". We have: S (3) T (1) E (1) A (1).

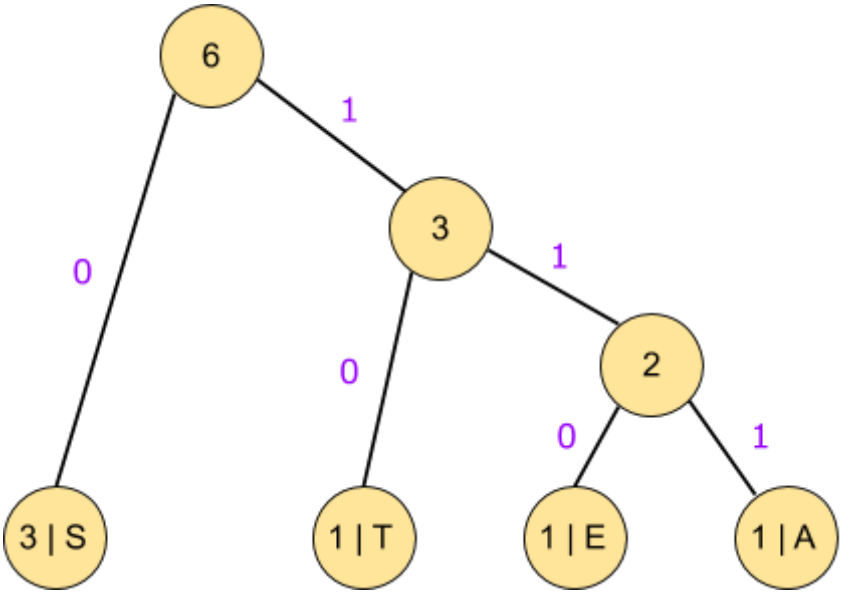
We will regroup **E** and **A** in a node.
This node will then contain two children (**E** and **A**), and its weight will now be 2 (1 for **E** + 1 for **A**)



We continue by grouping the two smallest occurrences together (here 2 and **T**)



We finally end with the two nodes of weight 3 to complete our tree:



This tree allows us to obtain a **unique** sequence **without prefix** for each character, with a shorter code for characters with higher occurrences ! Indeed, at each step to the left, we code with a **0**, for each step to the right, we code with a **1**. The more we advance in the tree (the longer the code) the less occurences the characters have.

We obtain the following dictionary:

Character	bit
S	0
T	10
E	110
A	111

The dictionary

The dictionary is what will allow us to encode / decode the message.
Just convert a character to its binary representation.
The word **ASSETS** gave in binary ASCII `01010100 01000001 01010011 01010011 01000101 01010011` or **48** bits.
Thanks to the Huffman code, we can encode **ASSETS** in `10 111 0 0 110 0` or **11** bits.

Note : to decode, we need the dictionary… which itself takes up space!

Part 1: From letter to bit

We want here to be able to visualize what a text in bits would give. We will therefore explicitly translate each character into a binary string of 0 and 1 corresponding to its ASCII byte.
Here is the correspondence table, between a character and its binary representation.

Example : The word `Alice` will give in binary :
0100000101101100011010010110001101100101

Be careful... do not code each correspondence by hand. There are more ingenious ways ... Also, don't forget the 0's at the start if there are any!

A) ★★: Write a function that reads text from one file, and translates it to its 0 and 1 equivalent in another file.

Example : (the colors are just to show the match, and are not requested):

Alice.txt	Output.txt
Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do	01000001011011000110100101100011011001010 01000000111011101100001011100110010000001 10001001100101011001110110100101101110011 01110011010010110111001100111001000000111 01000110111100100000011001110110010101110 10000100000011101100110010101110010011110 01001000000111010001101001011100100110010 10110010000100000011011110110011000100000 01110011011010010111010001110100011010010 11011100110011100100000011000100111100100 10000001101000011001010111001000100000011 10011011010010111001101110100011001010111 00100010000000110111101101110001000000001110 10001101000011001010010000000110001001100 01011011100110101100101100001000000110000 10110111001100100001000000110111101100110 00100000011010000110000101110110011010010 11011100110011100100000011011100110111101 11010001101000011010010110111001100111001 00000011101000110111100100000011001000110 1111

B) ★: Write a function that displays the number of characters in a txt file.

Example : For the `Alice.txt` file, it will display `103`, for the `Output.txt` file, it will display `824`.

If everything has been done correctly, there should be 8 times more characters in the output file, since each character is coded on one byte (8 bit).

Now we have a quick way of knowing exactly how many bits a message is going to take in memory.

Part 2 : The naive version of the Huffman code

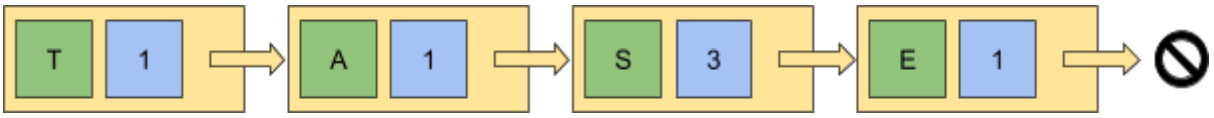
The goal of this project is to minimize the number of binaries required to encode the input text. For the **Alice.txt** file, it will therefore be necessary to find a number smaller than **824** bits. Here, we can aim for a **410**-bit output.

2.1 The occurrences

First step: We want to obtain the correspondence between character and occurrences.

C) ★★: Write a function that returns a list containing each character present in the text, as well as the number of occurrences of that character.

Example : for the text "**ASSETS**", we want the following list :



Tip : We look for the element containing the corresponding letter to add an occurrence. If it is not in the list, we add an element containing this letter. Order does not matter here.

2.2 The tree

Step Two: This is the most complicated part. The algorithm is not very long, if we organize our structures and the steps that we seek to encode.

D) ★★★: Write a function that returns a Huffman tree, from a list of occurrences.

Example : see the tree part. The order in the event of a tie does not matter.

Note : We will have to compare list links with nodes.
It may be wise to group the structures together… List of Nodes maybe?

Tip : A function that isolates and returns the smallest item in a list will be welcome.

2.3 The dictionary

Last step: We now have a Huffman tree. We now need to create the dictionary corresponding to this tree. In other words, match each character in the text to a string of 0's and 1's.

For each leaf, you will have to match the path to get there.

E) ★★: Write a function that stores the dictionary from the Huffman tree in a txt file.

Example : see the tree part, especially the final tree, with the steps in purple.

We want to output a `dico.txt` file containing:

dico.txt

```
S:0
T:10
E:110
A:111
```

The precise format and the separation between character and binary string is at your discretion.

Tip : Start by trying to display the path to a specific letter.

Tip : use a character string (an array) to remember the path traveled to each node.

2.4 Encoding

F) ★★: Write a function that translates a text into a binary sequence based on a Huffman dictionary.

Tip : For each character of the input file, you will have to look for the corresponding code in the dictionary file, and write it in the output file.

Note : The case of the "newline" character can be problematic !!

We just have to put everything together

G) ★: Write a function that compresses a text file. The input file will not be modified, another file, containing the compressed text, will be created.

2.5 Decoding (optional)

H) ★★: Write a function that decompresses a text file from a Huffman tree. The input file will not be modified, another file, containing the decompressed text, will be created.

Note : We will not be able to decompress a file outside of our compression program, since we need the tree in memory.

Ok. We have a way to compress our text file WITHOUT LOSS!

But ... It's soooo long. (approximately 11 seconds for 2000 lines)

It's time to optimize it all.

Part 3 : Optimization

We will here rework the different parts to see what can be optimized.

3.1 The occurrences

The goal is to retrieve the number of times each letter appears in a file. We started with an idea: browse the file, and for each character, look for it in a list. If it isn't in it, we'll add it.

So we're going to do a lot of research, with just a few additions. Especially if the text is long.

It would then be necessary to optimize the search part, even if each addition takes a little more time.

Dichotomous research can be a good idea. So we need a sorted array, and keep it sorted each time we add items.

I) ★★: Write a function which, by dichotomous search, adds an occurrence to an array of nodes when the character has already been found, or which adds a node containing the character otherwise.

I-bis (optional) ★★★: An AVL would be nice too...

3.2 The tree

We now need to sort our array not by character, but by occurrences.

J) ★★: Write a function that sorts an array of nodes based on occurrences.

Tip : Some sorting algorithms are faster than others...

Now that we have a set sorted by occurrences, we can use this order to quickly create our tree. No need to look for the minimum every time.
Except that if we create a new node having the sum of the occurrences of its children, we risk breaking this order. The idea here is to use two queues to build our tree.
The first will contain all our nodes from the sorted array, the second will serve as storage when we create a new node.

We will compare at each stage the first in each queue to know which nodes to take.

K) ★★★: Write a function which, using two queues, creates the Huffman tree from an array of nodes sorted by occurrences.

3.3 The dictionary

For the dictionary, storing it on disk to read it each time is not very efficient. We just want to know which code corresponds to the letter read in the file to compress. Going through the Huffman tree does not help, since we do not know where the letter is.

On the other hand, as for the occurrences, we can retain the letters and their occurrences in memory in an AVL.

This minimizes the number of operations required to obtain the code corresponding to each letter.

L) ★★★★: Write a function that organizes the nodes in an AVL according to the order of the characters.

Tip : For each letter (leaf) of the Huffman tree, we want to add to our AVL a node containing this letter and its code.

Note : The dictionary is therefore now a tree, no longer a file.
If you want to keep your dico.txt file (very useful for tests) you will have to store this tree on disk in a file, in the same format as for question E.

3.4 Encoding

All that remains is to browse this AVL, in an intelligent way, to find the letter to encode, and hence the code associated with it!

M) ★★: Write a function that optimally compresses a text file. The input file will not be modified, another file, containing the compressed text, will be created.

Note : We now have 0.04 seconds for 2 000 lines.

3.5 Decoding

To optimize the decoding, we need to be able to recreate the Huffman tree. We can't keep it in memory, we'll have to find a way to transport it.

We can therefore either go through a separate file, or by storing the dictionary before the text (in the same file).

N) ★★: Write a function that decompresses a text file from a Huffman dictionary file. The input text file will not be modified, another file, containing the decompressed text will be created.

Tip : We will have to recreate the Huffman tree from the dico file.

Note : Each time we compress a file, we will need the dictionary, which will be the decryption key. Which means you have to pass the dictionary on as well. It is possible to store the dictionary in the same output file, making our compression more autonomous.

Next Steps (to go further)

- reformat the dictionary to take up as little space as possible.
- store the dictionary in the compressed file.
- Adapt the entire program to work with a dictionary stored in the compressed file.

For the insanes :

- Refactor everything with REAL bits! => be able to actually compress a file.

Annexes

Si vous voulez tester vos programmes, vous trouverez sur ce drive les fichiers suivants :

- **input.txt** : Un fichier source, contenant le texte à compresser. Il ne contient pas de caractères spéciaux hors table ascii simple.
- **binary.txt** : Une traduction du fichier input.txt en sa représentation binaire.
- **huffman.txt** : Une traduction du fichier input.txt en sa représentation codée par l'arbre de Huffman.
- **dico.txt** : le dictionnaire de Huffman obtenu du fichier input.txt

Note : Le même fichier peut donner des arbres de Huffman différents suivant la priorité donnée en cas d'égalité de fréquence. De même, le dico peut être obtenu en inversant 1 et 0. Vous obtiendrez peut être des choses différentes, qui ne sont pas forcément fausses.