Huffman Encoding Essay

Abstract

This is an essay outlining how a lossless compression and decompression algorithm can work using Huffman encoding. I outline the processes of: Making an array of all characters from plain text file, count frequencies of each letter. How to make a Huffman tree. Going down each branch and record the depth of each leaf, record all the characters at each depth in a list to make canonical codes. Encoding the codes and Huffman codes of all characters into one binary string, in the process compressing. Reading the binary file, decrypting and recreating the original text file.

1 Introduction - How to run:

- Log on to a Durham computer (in windows mode).
- Place the unzipped directory 'digital_communications', in a directory of your choice, say Desktop.
- Navigate to the App Hub from the start menu, launch a Python 3 IDE (there should be a Python 3.6.3 IDE).
- Go to the command prompt and type 'pip install chardet'.
- In command prompt, navigate to the directory in the 'digital_communications' folder.
- In that directory add any plain text files you would like to encode.
- To encode: In command prompt type 'python encode_decode.py encode file_to_encode output_filename' where 'file_to_encode' is replaced by the name of the plain text file you want to encode, and 'output_filename' is the name of the compressed file the program will write.
- To decode: In command prompt type 'python encode_decode.py decode file_to_decode output_filename' where 'file_to_decode' is replaced by the name of the compressed '.hc' file you want to decode, and 'output_filename' is the name of the plain text file the program will write.

2 Making a Huffman tree

2.1 Reading a text file

Open the text file in read binary form 'rb', use the module 'chardet' to detect the encoding type from the opened file. assign a variable 'charenc' the encoding type as a string. Next create a variable 'text_ file' and open the text file in the correct encoding type. Put each character in a list from the text file. Make a variable called 'counter' from the collections library, by calling 'collections.Counter(character_ list)', for the list we just made. 'counter' is a set of tuples with the first element in each tuple being the character, and the second element being the frequency of that character.

2.2 Making tree from frequency table

- For easier readability of my code I split counter into 2 lists, 'values' and 'counts' using the zip function, on counter organised by the frequency of each character, and using '[::-1]' to reverse it, making 'values' and 'counts' in ascending order.
- Next I made an array called 'branches' to hold the structure for my Huffman tree. 'branches = [branch]' eg. Where 'branch' can be: A tuple containing 2 tuples: (TUPLE_ 0, TUPLE_ 1) which has 2 ways you can go down. A tuple (in position 0) and a value (in position 1) eg. (TUPLE, 'a'). Or a tuple and a total sum of all the frequencies of characters inside that tuple: (TUPLE, sum), where sum is an int.
- I start by adding the first element to branches, which looks like so, 'branches = [((values[0], values[1]) , counts[0] + counts[1])]' all this does is put the two least frequent characters into a tuple, then that and the sum of their frequencies into another tuple. (I then remove the first 2 elements from the lists 'values' and 'counts'). To create the tree, I use a while loop that runs while the size of the list 'values' is greater than 0 (could have equally been 'counts').

2.3 Inside the while loop

- I find the 'branch' in 'branches' that is least frequent, by using 'min(ARRAY)' where ARRAY is 'branches' but only the second item in each tuple in the list (these are where the sum of the branch is stored) by using a lambda function in place of ARRAY, 'branches, key = lambda t: t[1]'.
'if (len(values) == 1 or counts[1] >= branch_ count) :' This means if the size of the array 'values' is 1, the condition is satisfied, otherwise 'len(values) > 1' and therefore so is 'counts'.

- This then checks if the second least frequent letter is greater or equal to the lowest branch. This essentially compares wether $1^{st} + 2^{nd}$ least frequent letter $\leq 1^{st}$ least frequent letter + lowest branch. If it is, we must add the new letter to the branch. 'branches.append(((lowest_ branch, values[0]), counts[0] + branch_ count))' then remove the branch 'lowest_ branch' (as it's within the branch we just added), then remove the letter and its frequency from 'values' and 'counts'. If this is not true then the $1^{st} + 2^{nd}$ least frequent letters is lower than the branch, so we need a new branch. 'branches.append(((values[0], values[1]), counts[0] + counts[1]))' and then removing the first 2 values from the lists 'values' and 'counts'.

2.4 Grouping all the branches

- Now, this has almost made our tree, although it will usually contain more than 2 branches at the first place in the list 'branches'. To group these multiple branches into pairs (by least frequent sums) I made another while loop. The loop just takes the two lowest branches and pairs them together, adds them in one tuple with their sum, and then removes the original ones. This completes the Huffman tree. The tree now looks like so: (((BRANCH), (BRANCH)), total_ sum), where each BRANCH has the same format (if there are more branches in it), but a leaf if it is the end of a branch.

3 Finding depth in tree of each letter

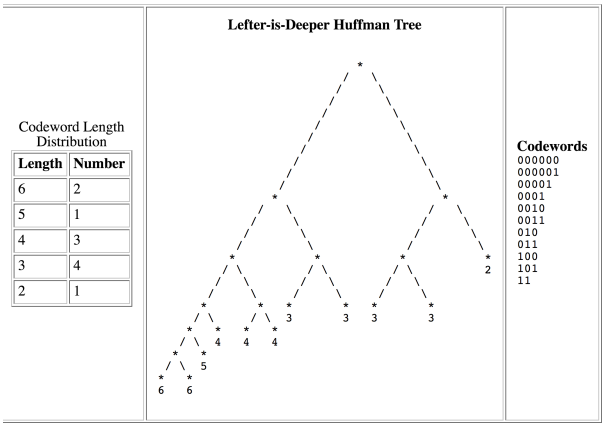
Forming the binary codes for each character from the Huffman tree, 'branches' just made:
Firstly I made an empty array called 'huffman' which will be filled with tuples with first element the character and the second the binary code associated with it. My looping system keeps track of the routes through the 'branches ' array by recording the binary codes of completed routes, this is done in a set called 'completed_routes'. By default the loop searches the '0'th element in each tuple first unless there is already a route that contains '00' or '01' after the current route.
Outer while loop:
I make a boolean variable 'continu' initialised to 'True' and make a while that runs while 'continu'. In the loop I assign a variable I called 'go_down' to the first element in 'branches' (the whole tree of tuples) and initialise the string for the binary code for that route. Then I made another while loop, which I will come back to, but it basically finds the next route down the tree. I make a copy of all the completed routes and run through each route checking if the routes with a '0' as the last element. If it does, I make a variable, 'first' that is the route, not including the last character, and then check if completed routes contains 'first' concatenated with '1' or '11'. If it does we've explored all routes that start with 'first' + '0', so I add 'first' to completed routes so that I can use it later to not go down that route again. Finally in the outside loop I make a for-loop to check whether all the codes have been completed: First I set 'continu' to 'False' then check all the characters in the current route 'string' if they are '0', if they are, I set 'continu' to 'True'. The outer while loop will therefore terminate as soon as the route is all '1's, which is when the whole tree is done.
Inner while loop:
This loop is broken when a route is found. It's job is to construct the next distinct route to a letter through the 'branches' array. Each loop the variable 'go_down' is updated, by reassigning it to 'go_down = go_down[0]' or 'go_down = go_down[1]', until the final loop- when the route is found. There are only a few scenarios that we can test for that determine how we update/find our route (using type checking): We check if we are already at a letter (a leaf), then update the list 'huffman' with that letter, and the route to it in a tuple, add the route to 'completed_routes', then break the loop. Otherwise, check if the length of 'go_down' is 0, or if 'go_down' is a tuple with a int in the 2nd slot of the tuple and go string down (not updating the route). Now we essentially just have a tree that is made of tuples (branches) and letters (leaves). This means we need appropriate conditions for a route change depending on which routes we've been down. I make a variable called 'route' and assign it to 'string' (the current binary string for the route down the tree) just to use as a copy. Case of two tuples within the tuple 'go_down': This is a junction of two different branches: (Use 'elif' so that this only checks when the previous ones aren't satisfied.) Then check if 'route' concatenated with '00' or '01' are in 'completed_routes', what this means, if they are in 'completed_routes' is that all the routes continuing from 'route' + '0' have been tried, and we must go down 'route' + '1'. So the next line is 'go_down = go_down[1]' then 'string += '1" to update the route. Else if this is not true, we must check if the route is the empty string or 'route' + '0' is not in the completed routes, if they are not, we can go down that way ('go_down = go_down[0]' and 'string += '0"). Then we use 'elif' to similarly check for 'route + '1". Finally in the case where the second element in the tuple is a leaf. (Again use 'elif' so that this only checks when the previous ones aren't satisfied.) Similar to the last case, if the two routes ('route' concatenated with '00' or '01') are in 'completed_routes' we must go down 'route' + '1'. Else: Then check if the first element in 'go_down' is a tuple and if so, if 'route' is the empty string or 'route' + '0' is not in completed routes as before, if not, go_down[0]. Else if the first element in 'go_down' is also a string? if 'route' + '0' is not in completed routes again ('go_down = go_down[0]' and 'string += '0") else if 'route' + '1' is not in completed_routes: 'go_down = go_down[1]' and 'string += '1".

4 Making Canonical codes for each letter

4.1 Function 'make_ tree(sets_ freqs):'

[1] This function works out binary routes for each character, that have the same length as the depth of that character in the Huffman tree.

[1] http://www.cs.uofs.edu/~mccloske/courses/cmps340/huff_canonical_dec2015.html




```
def make_tree(sets_freqs):
    huffman = []
    codess = []
    next_route = ''
    for set_ in sets_freqs:
        next_route, codess = pass_set_return_closed_route(set_[0], set_[1], next_route)
        huffman += codess
    return huffman

def pass_set_return_closed_route(chars, length, next_route) :
    codes = []
    num = 0
    if next_route != '':
        if len(next_route) > length:
            next_route = all_0s(length, remove_ends(length, next_route))
        num = int(end_concat0(length, next_route), 2)

    for char in chars:
        codes.append((char, front_concat0(length, to_binary(num))))
        num += 1
    next_route = remove_end0s(front_concat0(length, to_binary(num)))
    return next_route, codes
```

Figure 1: Lefter-is-Deeper Huffman Tree.

- I pass a list 'sets_ freqs' which is all the characters at each depth in a list (in alphabetical order for each depth). This initialisers a list Huffman, and codes and a string next_ route.
- Then goes through each set of letters (for each successive level/depth- starting from the longest depth).
- I then update the next_ route string and codes array using another function called 'pass_ set_ return_ closed_ route(set_ [0], set_ [1], next_ route)', which passes the list of characters at that depth, the depth and the next route. Then I concatenate the list codes with Huffman.

4.2 Function 'pass_ set_ return_ closed_ route(set_ [0], set_ [1], next_ route):'

'pass_ set_ return_ closed_ route' also makes a list 'codes' and sets a variable 'num' to 0. Then checks if the 'next_ route' is not the empty string. If not, then if the the length of the next route is larger than the depth then:

- assign next route to 'next_ route' with the excess bits cut off. Since 'next_ route' is a binary number its Huffman code must not be longer than the depth in the Huffman tree. If 'next_ route' is all zeros it reassigns 'next_ route' to '00...01' the length of the current depth, if 'next_ route' was all zeros, then checks if the current 'next_ route' ends in a '0' then it replaces the last character in the route from '0' to '1'. Finally add zeros to the end of the number until it is the correct length, and record the actual integer this number correspond to in the variable 'num'.
- This is all that needs to be done to increment the 'next_ route' variable to the route of the first character in the set we passed this function (at that depth).

Now we can just do a for-loop through each character in the set we were passed. In the loop we:

- Make a code for the current character from the binary form of num, with zeros concatenated to the start to make it the correct length. Append this code to 'codes' list in a tuple with the character it corresponds to. Increment the 'num' variable by 1.
- Finally what we need to return for this function is: the next route, which is the current 'num' in binary form- again with the appropriate number of zeros at the front of the number, and all the end zeros trimmed off. As well as the codes array.

In summary the function 'make_ tree(sets_ freqs):', takes a list of all the characters at each depth in a list (in alphabetical order for each depth), and returns a canonical tree. This mean we can always create the same Huffman tree just from a list of this form. This is useful in encoding files as only information in the list 'sets_ freqs' will have to be encoded, instead of the whole tree.

5 Encoding

All of the information about the encoding of the file is put into a string of '1' 's and '0' 's. Call this string 'binary_string'. This is the format I chose to encode my string:

- binary_string = remainder (3 bits long) + encoding_type (2 bits long) + '000'
- binary_string += noise_byte (only there if remainder != 0, will contain 8 - remainder number of '0' 's)
- binary_string += [max_level] (8 bits long)
- binary_string += [freq_on_level1][freq_on_level2]etc.
- binary_string += [unicode binary value] (for all characters)

- binary_string += main_string

The remainder is a integer from 0 to 7, and hence can be represented by 3 bits. The number of encodings my program can analyse is 3, so 2 two bits of information are required. This could be extended to represent more encoding types, I've left 3 bits after this just to fill up the rest of the byte which could be used for representing more different encodings. Next if the remainder is 0, then the completed binary_string can immediately be written in a byte array. If the remainder is non 0, then I add (8 - remainder) '0' 's just to make the completed binary_string a multiple of 8. Then I add a byte that represent the integer for the maximum depth reached in the Huffman tree. Then I use 1 bit to represent the number of different characters of the first level since there can only be a maximum of 1 characters on the first level (only 1 possible leaf, so 1 bit can represent this), then I use 2 bits to represent the number of different characters of level 2 as 2 is a maximum number of leaves there can be. Then this repeated with 3 for level 3, 4 for level 4 etc. up to the maximum depth.

5.1 Writing the unicode binary values for all different characters

```
def text_to_bits(text, encoding='utf-8', errors='surrogatepass'):
    bits = bin(int.from_bytes(text.encode(encoding, errors), 'big'))[2:]
    return bits.zfill(8 * ((len(bits) + 7) // 8))
```

```
def bits_to_text(bits, encoding='utf-8', errors='surrogatepass'):
    n = int(bits, 2)
    return n.to_bytes((n.bit_length() + 7) // 8, 'big').decode(encoding, errors) or '\0'
```

Figure 2: text_to_bits function and bits_to_text function.

The characters are ordered in a list, 'chars_set', such that for each successive level, they are in alphabetical order. eg. level 1 sorted alphabetically, concatenated with level 2 sorted alphabetically etc. This order is essential so that we can recreated our canonical codes upon decoding. For each character in 'chars_set' I call the function 'text_to_bits' with arguments: character and encoding type. Then I concatenate the output binary to the binary string.

5.2 Writing the main string

This is just a loop through all the characters in the document that concatenates the binary string with the canonical code for each character. The whole binary string can now immediately be written into a byte array as binary_string is a multiple of 8. Then written to a file, in binary form 'wb', choose extension '.hc'.

6 Decoding

The structure I specified for the encoding can be deconstructed to collect all the information we need. Start by reading the first 3 bits for the remainder, the next 2 bits for encoding type, remove the first byte. Then remove (8 - remainder) from the next byte (unless remainder = 0). The next byte we read, we assign to the max level (or depth) of the Huffman tree. Now we can read off the number of characters at each depth, up to the max depth. First read the first bit for a depth of 1, next the following 2 bits, etc. now we know how many of each character at each depth. Now we sum the total number of characters (the sum from each level), make a character list by looping the number of times there are characters, appending each character to the list after decoding the appropriate length of binary depending on the encoding. (ascii always takes 8 bits, utf-8 can use 8,16,24 or 32, UTF-16 can take 16 or 32). So for encodings utf-8 and UTF-16 we must check the start of each binary string where it indicates how much bytes it takes before passing it to the function bits_to_text Figure(2).
Now we have all we need to make our canonical codes for each character, we can recreate the 'set_freqs' list, by making a list of the characters on each successive level, adding it to a tuple with the level as the second element in the tuple, repeating for all the levels/remaining characters. This is the only requirement for making canonical codes, so we can pass it back to our original function 'make_tree' and it returns a list of tuples, with first elements being the character, and the second being the Huffman code.
Finally we can loop through the remainder of the binary string adding each character to a list as soon as it matches a Huffman code. This makes a list that is identical to that from reading the original text file, so we can write the list to a file and we are done. This looping process for this however must loop for all bits in the string, and check each time if that code is in the Huffman list, which is a very computationally time consuming for large Huffman lists.

7 Analysis

Name:	Size: (bytes)	Compr. ratio	binary read:	counter time	Tree time:	Depths time:	Canonical:	file write:	huffm. decoc	main decode	write time
variety of utf8 c	86,743	0.622	0.438796	0.010736	0.000502	0.006076	0.000516	0.256717	0.06259	0.804654	0.0233
Japanese Art bo	204,704	0.591	0.8842	0.02574	0.000452	0.003444	0.0003531	0.5111	0.1695	1.257	0.05282
Einstein relativi	264,667	0.652	1.185	0.05019	0.0003922	0.006668	0.0007389	0.7584	0.233	1.981	0.0807
russian	440,485	0.584	0.0008681	0.06301	0.0003119	0.004049	0.0003681	1.139	0.3375	2.801	0.1251
latin	841,378	0.5933	0.003419	0.1232	0.0004287	0.006195	0.0004752	2.455	0.7112	7.073	0.286
chinese(1)	1,238,846	0.4318	0.007134	0.1206	0.3025	5.741	0.01346	40.61	3.04	212.8	0.1338
chinese(2)	1,977,045	0.4146	0.005074	0.2007	0.4321	7.558	0.0171	76.59	5.293	367.6	0.2044
war and peace	3,293,491	0.5956	15.41	0.4608	0.0003462	0.005132	0.0005679	9.222	2.789	23.79	0.8429

Figure 3: Timings data.

My algorithm seems to work well over a variety of utf-8 files, however parts of the algorithm performed badly for Chinese characters as there was a much larger set of distinct characters to map to the tree and make the Huffman codes. The counter time performed close to linearly with file size as we'd expect. Once the counter computation is done, we can see that the time to make a tree or codes are around the same as we have a similar number of distinct characters to map in each case. Although for Chinese there is far more characters/codes to map and we can see this in Figure(4). The time for my algorithm to find the depths of each character in the tree was the main limitation in the algorithm, it has to search through a list of routes it can go down before choosing the next route, this does not take very long for small character sets, but very long for large character sets. The file write time should correspond to to file size needed to produce but also limited because the file is written character by character, which each characters index must be found and then the code at that index written (so again very large character sets will take a long time to write). Similarly for the decoding of the Huffman small headers correspond to a small character set, larger ones will take longer as they have deeper branches in the tree. The decode of the main string will behave similarly to the file write time, however have to do more check against all the Huffman codes for each added bit as they are read, this takes takes a long time will very long strings to read and large sets of Huffman codes. The compression ratios were around 0.6, but notably lower for Chinese documents as most Chinese characters will have unicode binary values that are around 4 bytes long which tend to be significantly longer than the Huffman code we generate.

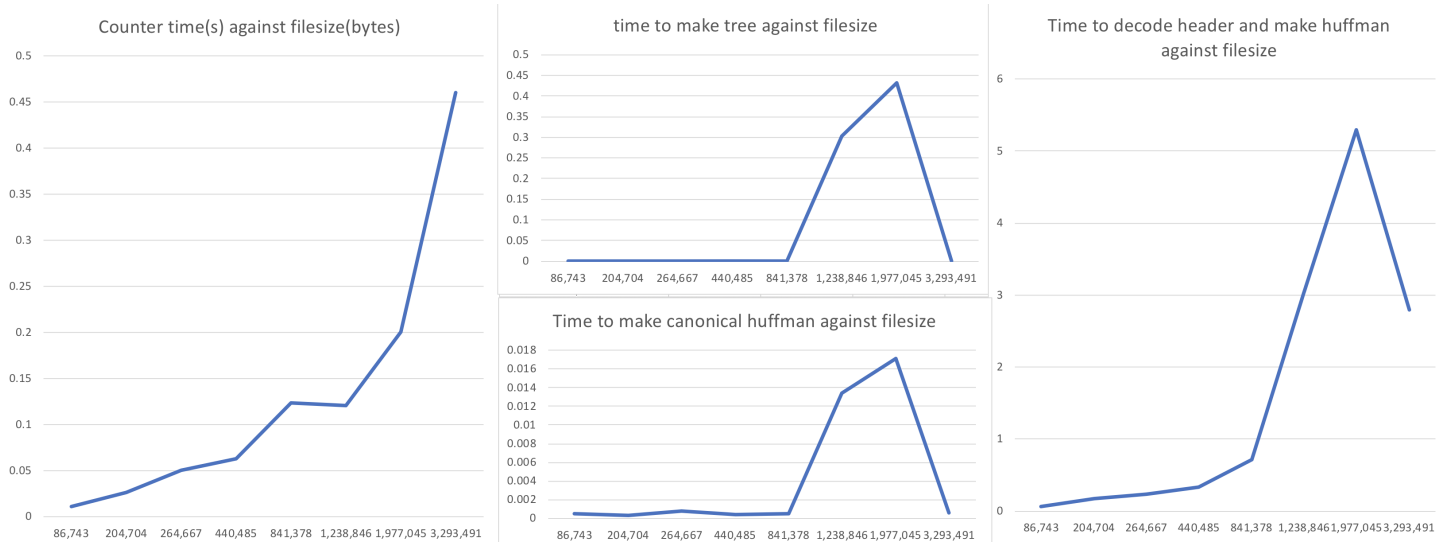


Figure 4: Timings plots.

8 Conclusion

In conclusion my encoding and decoding algorithm can compress plain text files in ascii, utf-8 and UTF-16 without any loss of information. The degree to which the order of all the characters is compressed is at a minimum as I use Huffman encoding. The degree to which the table of Huffman codes to character unicode values is compressed, requires very close to the minimum amount of information. I store the max depth as one byte, then the number of chars on level 1 in one bit, on level 2 in 2 bits, etc. Then I store all the unicode binary values of the characters in alphabetical order on the lowest level, the next lowest level etc. From this I can rebuild the canonical Huffman codes. I also store the remainder of bits of the string as the first 3 bits of the first byte, and then the encoding type as the next 2 bits, and if there is a remainder the next byte contains 8-remainder '0' 's. The main areas for my algorithm to be improved is when computing the depth of each character in the Huffman tree, ideally this would be computed when the tree is made as this is all the information we need for each character (as the actual code will be assigned from the canonical function). In my algorithm I have to calculate each route down the tree and record an compare each route before making a new one, this is very expensive computational for very large sets of different characters. Another improvement would be compatibility for more encoding types.

References

[1] http://www.cs.uofs.edu/~mccloske/courses/cmps340/huff_canonical_dec2015.html.