COMP3491 Codes and Cryptography Report

gqgw27

1 Static Dictionary of English words

The first step that will be used in this compression program is to shorten the length of common English words that appear in the text, before applying any further compression methods. This was achieved by first downloading a text file of common English words [1] containing around 65,000 lowercase words and generating an extra 65,000 words for their equivalent capitalised counterparts. Then, each word of length three or more was replaced by a 3-byte string and stored in a separate static dictionary for the program to access from. As a static dictionary is used, the encoded file cannot increase in size as there is no need to pass any extra information. Therefore, compression is essentially guaranteed.

The encoder will first find the English words placed between other symbols (for this program 25 common symbols like spaces, commas, brackets etc. have been identified) and then convert the words that occur in the dictionary to their 3-byte counterpart. When decoding, the decoder will again separate words from symbols and decode the words back to English using the same static dictionary used in encoding. The reason 3 bytes was used, is for the decoder to uniquely identify the encoded 3-byte words from other characters in the stream. This is because the input files are known to be encoded in ASCII and so the characters range from 0 to 127 (as evident from the assignment spec and the given testEncoderDecoder.py). As a result, the first byte will take 13 values between 128 and 140, while the second and third take 103 values (128 – 25 symbols) from 0 to 127. This means that over 130,000 words (13x103x103 = 137,917) can be encoded and decoded uniquely.

One further optimisation I made was to group consecutive English words separated by a space together without passing the space. For example, 5 words that appear together would only take 5x3 = 15 bytes, rather than 5x3 + 4 (space) = 19 bytes.
The encoding that will be used for compression will be Latin-1, as it can store 0 to 255 characters using only 1 byte, unlike UTF-8 which uses 2 bytes for 128 to 255.

2 BWT Encoding

The main technique that will be used is the Burrows Wheeler Transform. Although the Burrows Wheeler transform in itself does not compress, it is an extremely useful technique that will help compress the resulting data a lot more. This is because the transform essentially "sorts" the input data such that similar and identical characters are often grouped together [2]. So essentially there is added repetition that is brought into the text causing redundancy, and hence the resulting text can be further compressed better.

For some text of N bytes, the way I initially implemented the transform was by generating N permutations of the text, where each permutation is a cyclic shift of the original text from 0 to N-1 [3], and appending to an array. The resulting array is then sorted and the last character from each permutation is added together, forming the resulting transformation. The index of the original unshifted message in the new sorted array is also noted.
For decoding, I followed the process from [3], having two arrays **F** and **L**, where **L** is the encoded text and **F** is the sorted version of the encoded text. Using the initial index $\boldsymbol{i}$, the text can be decoded using arrays **F** and **L**. The first decoded letter **s** will be $\boldsymbol{i}^{\text{th}}$ letter in **F**. Then, the index $\boldsymbol{i}$ is replaced

by the index in which **s** appears in **L**. The second decoded letter will be the letter in **F** in the new index position. This process is repeated until the whole sequence is decoded.

3 Improving performance of BWT

Although the above implementation is able to perform the Burrows Wheeler transform on a given text, there is a severe hit on memory that just doesn't make it feasible to run as is. Consider a text of N bytes. There is an array of size N, containing the N cyclic shifts, meaning that the size of the text in the array itself is N^2 bytes. For even a text of just 1MB, the array would require 1 terabyte of memory!

As a result of this, two changes needed to be made:

1. Performing the Burrows Wheeler transform only in blocks of fixed size across the text rather than on the whole text itself.
2. Reducing memory usage

The first criteria is simple to implement, with the only change being is needing to store the multiple index values for each of the number of blocks. To reduce the memory usage, the size of the array needs to be drastically reduced. Instead of adding the whole cyclic shift of size N to the array, I add a tuple containing **1.** *The first 500 characters of the cyclic shift* **2.** *The last character in the cyclic shift* and **3.** *The corresponding cyclic shift number.*

This array is then sorted according to the first item in the tuple (i.e. the first 500 characters in the cyclic shift). The sorted array is then scanned to check for portions in the array, whose items are all the same (indicating the initial 500 characters weren't enough for sorting correctly). Using the cyclic shift number that is present in the tuple, the full cyclic shift is performed and this portion of the array is sorted correctly. The last characters are then added to the encoded result in their correct positions. For portions in the array whose items are not the same (indicating correct sorting), the last character (this is stored in the tuple) is added to the encoded result in the correct position.

In this implementation the memory usage is drastically reduced and much larger batch sizes can now be used, resulting in a better "sorting" thereby compression later on. 1MB batches now use only around 500MB to 1000MB of memory.

4 MTF Encoding

After performing the Burrows Wheeler transform, Move to Front encoding is performed on the resultant sequence. This works by having a list of the source alphabet and encoding the index where each character occurs in that alphabet list. However, after each each character is encoded, that character is brought to the front of the alphabet list and the next character is encoded. So ultimately, repeated runs of characters will be encoded as multiple 0's. Although MTF alone doesn't cause compression, this will improve redundancy even more. This is because although there are different repeated runs of characters after BWT, when MTF is performed, all repeated characters, regardless of what the repeated character is, will be encoded as a run of repeated 0's [4].

5 Run Length Encoding

Now that the sequence contains multiple runs of repeated characters, a run length type encoding was performed on the resulting sequence, which will now result in compression. Due to Mtf encoding having been done prior, most of these runs will be runs of 0 and only a few may not be. I will implement this by consider a few different cases.

1. If there is a run of 0's it can be encoded in 2 bytes; 1 byte for the length of the run (no more than 255 runs will be encoded at a time to restrict to 1 byte, and so throughout this section)

and 1 byte representing the current case. In order for compression, only runs of length 3 or more will be considered using this first case.

2. If there is a run of 0's of length exactly 2, it can be encoded by passing one byte representing this case
3. If there is a run that doesn't consist of 0's, this can be encoded in 3 bytes; 1 byte representing the length of the run, 1 byte representing the current case and 1 byte representing the current character. In order for compression, only runs of length 4 or more will be considered in this case.
4. If the length of the run that doesn't consist of 0's is exactly 3, then this can be encoded in 2 bytes; 1 byte for the character and 1 byte representing the current case.
5. For all other cases the character will only be passed just as it is.

The bytes that I will use to represent these four cases will be 141, 142, 143 and 144, as these do not and cannot occur anywhere else in the text. For decoding, the opposite of the above is just performed.

6 Arithmetic Encoding

Finally, after performing the run length encoding, I will perform arithmetic encoding on the resulting sequence. For implementation I have taken reference from [3] and implemented the integer version of the algorithm. After performing the arithmetic encoding I end up with a bit-stream of 0's and 1's. However, in order for me to decode the encoded result successfully, I will also need to pass the cumulative probability table of the characters, along with the encoded result. Since the cumulative probability table can be constructed from the counts table (which is smaller and represents the counts of each character in the sequence), I will pass the counts table instead as a dictionary.

- First, I will use 8 bits to store the number of items in the counts dictionary. This is sufficient as the number of different characters will not exceed 256.
- Then, 3 bits are used to represent the **minimum number of bits (b1)** needed to store the maximum character value (up to 8 bits, hence up to character 255).
- Then, 5 bits are used to represent the **minimum number of bits (b2)** needed to store the maximum count value for a given character (i.e. up to 32 bits to store the count, hence up to a max count value of 2^{32}).
- The counts dictionary is then passed in as a key,value pair where each key,value pair takes exactly **b1+b2** bits.
- Finally the main encoded bit-stream is passed in.

In order to convert the bits into Bytes, I add any extra 0's to the end of the bit-stream to ensure that it is a multiple of 8. From here, I just consider every block of size eight, convert the binary sequence into an integer and encode the resulting character from 0 to 255.

References

[1] http://www.gwicks.net/dictionaries.htm?fbclid=IwAR1sI63YaMJR51tzPM0K1s8CmdKyT5gBdk2GehJCTnra9fJxhmMU5LbKS0

[2] https://sites.google.com/site/datacompressionguide/bwt

[3] Sayood, K., 2012. Introduction To Data Compression. Amsterdam: Morgan Kaufmann.

[4] Abel, Jürgen. (2010). Post BWT stages of the Burrows-Wheeler compression algorithm. Softw., Pract. Exper.. 40. 751-777. 10.1002/spe.982.