Lossless Compression via Burrows Wheeler Transform and Arithmetic Coding

Sinem Unal

We implement a lossless data compressor and decompressor using Burrows Wheeler transform and arithmetic coding. It was originally a graduate level course project in Cornell University. For the course project, we were originally required to design an encoder and a decoder to accomplish lossless compression of an English text file, specifically, a part of the novel *A Tale of Two Cities* by Charles Dickens. In this updated version, the implementation is modified in order to handle different files.[1] The algorithm is also tested on Canterbury dataset and some large text files.

Introduction

In literature, there are several methods such as arithmetic coding, *prediction by partial matching* (PPM) and *Lempel Ziv* (LZ) coding to compress a text. Among these schemes, there is no single scheme which is the best in terms of compression rate, the amount of memory usage and time. Each of them has advantages and disadvantages over the others in terms of criteria mentioned above. For example, LZ based coding schemes are faster and requires less memory when compared to PPM but PPM achieves higher compression rates. Also, arithmetic coding gives a slightly better compression ratio when compared to LZ coding but it is worse than PPM, which requires larger memory and a harder implementation. Besides these coding methods, there are some transformations such as *Burrows Wheeler transformation* (BWT) and *move to front* (MTF) coding and they can be implemented on the file before implementing the compression algorithms in order to increase the compression rate [1, 2]. Therefore, we decided to use arithmetic coding as suggested by [3, 4, 5, 6]. Moreover, we use BWT and MTF coding before applying arithmetic coding. General overview of the coding scheme will be explained in the next section.

Overview of the Algorithm

As mentioned in previous section, we use BWT, MTF coding and arithmetic coding and we review the approach in [3, 4, 5, 6]. For the encoding part, first we apply BWT to text file. BWT is a reversible transform such that it basically changes the ordering of the symbols in a block of data and it does not change the size of data. The reason why we use BWT is that it puts the characters of words which occur frequently in data together

[1] Specifically, the algorithm can compress a given file with size up to 2 GB.

Table 1: Burrows Wheeler Transform Steps on "WE-WERE-ALL"

Step 1	Step 2	Step 3
S0: WE-WERE-ALL	S7: -ALLWE-WERE	E
S1: E-WERE-ALLW	S2: -WERE-ALLWE	E
S2: -WERE-ALLWE	S8: ALLWE-WERE-	-
S3: WERE-ALLWE-	S6: E-ALLWE-WER	R
S4: ERE-ALLWE-W	S1: E-WERE-ALLW	W
S5: RE-ALLWE-WE	S4: ERE-ALLWE-W	W
S6: E-ALLWE-WER	S9: LLWE-WERE-A	A
S7: -ALLWE-WERE	S10: LWE-WERE-AL	L
S8: ALLWE-WERE-	S5: RE-ALLWE-WE	E
S9: LLWE-WERE-A	S0: WE-WERE-ALL	L
S10: LWE-WERE-AL	S3: WERE-ALLWE-	-

so transformed data consists of regions each of which contains large repetition of a small number of characters. After applying BWT, we apply MTF encoding to data. Since BWT output contains lots of repetitions, output of the MTF contains small numbers followed by lots of zeros which enables us to obtain larger compression rate when we apply encoding algorithm using statistical properties of data. Therefore, in the final step of encoding process we apply zero order adaptive arithmetic encoding. For the decoding part, we follow the reverse order in the encoding part. In other words, first we apply zero order adaptive arithmetic decoding. Then we feed the data to the MTF decoder and finally, we apply inverse BWT to get the original file. In the following three sections we will explain each method implemented to compress and decompress the data.

Burrows Wheeler Transform

We implement BWT and inverse BWT as suggested in [3], while also benefiting from a very useful dicusssion on the topic in [5]. First we will explain BWT and reverse operation on an example text string "WE-WERE-ALL". Then we will give a pseudo code for the algorithm. For the encoding, first we make all the rotations of a given string "WE-WERE-ALL". Now we have $N = 11$ (the length of the string or block length) rotated forms of the string as shown in Step 1 of Table 1. Secondly, we sort those strings in a lexicographical order. As a final step, we output the string by taking the last character of each sorted string and keeping the number of the sorted row which is the original string. For this example the output is S="EE-RWWALEL-", I = 9, output string and the index of original string respectively. Table 1 shows each step.

For the decoding part, we perform the inverse BWT. Given the transformed string and the index of the original string we can obtain the following additional information. First information is that for each symbol in a given string, we can determine the number of preceding symbols which are the same. Secondly, we can determine the number of symbols which are less than in lexicographical order for each symbol. Table 2 and 3

Table 2: Step 1 for inverse BWT

Position	Symbol 2	Matching #	Position	Symbol 2	Matching #
0	E	0	6	E	0
1	E	1	7	L	0
2	-	0	8	E	2
3	R	0	9	L	1
4	W	0	10	-	1
5	W	1			

Table 3: Step 2 for inverse BWT

Symbol 2	No. less than
-	0
A	2
E	3
L	6
R	8
W	9

illustrate this information. By using this information, we can reconstruct the original string starting with the last character. Note that when the length of the data to be transformed increases, BWT becomes more effective since we can capture more repetitions. Note that we know the last character of original string (it is the I^{th} index in the transformed string). Next, we find the second to last symbol. This symbol comes from the string S10 (i.e. SN-1) which starts with the last symbol that we found at previous step. From Table 2, we know the number of preceding symbols which are the same as the last symbol that we found, meaning that we know the number of strings which begin with the same last symbol whose index is smaller than the symbol coming from the string SN-1. Also, from Table 3 we know the number of strings beginning with lexicographically smaller symbols that came from before the symbol coming from the string SN-1. Since there is no other string which can contribute to transformed string S before SN-1, the index of the symbol coming from SN-1 in S is equal to the sum of the numbers coming from Table 2 and Table 3. The remaining symbols can be obtained by following the same procedure. Pseudo codes for BWT and inverse BWT are given in Algorithm 1 and 2 respectively.

Move To Front Coding

Similar to BWT, we implement MTF encoding and MTF decoding as suggested in [3] and utilizing the discussion in [5]. We begin with explaining the MTF encoding and decoding procedure and then we provide the pseudo code for them. For the MTF

Algorithm 1 BWT

input : *input_arr*
output : *output_arr*
rotate_arr ← concatenate (*input_arr*)
ptr_rotate ← addresses of *rotate_arr*
sort(*ptr_rotate*)
for $i = 1$: length of *input_arr* **do**
    output_arr[i] ← last index of each rotation in *ptr_rotate*
    if *ptr_rotate*[i] points *input_arr* **then**
        last element of *output_arr* ← i
end for

Algorithm 2 Reverse BWT

input : *input_arr*
output : *output_arr*
$Table2 \leftarrow 0$
for $i = 1$: length of *output_arr* **do** ▷ constructs $Table2$
    $Table2[i] \leftarrow Table2[input\_arr[i]]$
    $Table3[input\_arr[i]] + +$ ▷ frequency of i^{th} symbol in the input so far
$sum \leftarrow 0$
for $i = 1 : no\_of\_symbols$ **do** ▷ constructs $Table3$
    $sum \leftarrow sum + Table3[i]$
    $Table3[i] \leftarrow sum - Table3[i]$
end for
$idx \leftarrow$ the index of the original array in the sorted rotations
for $i = no\_of\_symbols : 1$ **do**
    $output\_arr[i] \leftarrow input\_arr[idx]$
    $idx = Table2[idx] + Table3[input\_arr[idx]]$
end for

encoding, we have a symbol list consisting of the characters in a given string. We will encode our source string based on the symbol list which is changing dynamically during the encoding. First, our symbol list is sorted in lexicographical order.[2] We encode the first symbol in source string such that we write the index of the symbol in the symbol list. Then this symbol is moved to the front of the symbol list. Then for the rest of the symbols, we apply same procedure. Note that if we have lots of repetitions in the given string, output of the MTF contains many zeros which enables us to obtain better compression ratio while using arithmetic encoding. Also, note that MTF does not change the length of the input string. For the MTF decoding, we mostly apply the reverse procedure in the MTF encoding. First our symbol list is again in a lexicographical order. We decode the first symbol such that we write the symbol whose index in the symbol list is the first symbol in encoded string. Then, we update our symbol list as explained in encoding part. For the remaining symbols, we apply the same procedure. Pseudo codes for MTF encoding and MTF decoding are given below.

Algorithm 3 MTF

```
input :  input_arr
output :  output_arr
char_list                                   ▷ Sorted list of characters
for i = 1 : length of input_arr do
    idx ← current pos of input_arr[i] in char_list
    output_arr[i] ← idx
    move idx^th element of char_list to front
end for
```

Algorithm 4 Reverse MTF

```
input :  input_arr
output :  output_arr
char_list                                   ▷ Sorted list of characters
for i = 1 : length of input_arr do
    idx ← input_arr[i]
    output_arr[i] ← char_list[idx]
    move idx^th element of char_list to front
end for
```

Arithmetic Coding

To implement zero order adaptive arithmetic coding, we use the source code introduced in [4]. We arrange the code in order to make it suitable to our overall implementation. For the zero order adaptive arithmetic coding we use adaptive model for calculating the probabilities of the symbols. Since it is zero order, we do not need to calculate transition probabilities. In principle, arithmetic coding part is the same as what we

[2]One can use other types of ordering at the encoding as long as it is consistent with the decoder.

covered in class. In this section, first we provide pseudo code of the both decoding and encoding algorithms and then explain those following the discussion in [4].
Pseudo code for the encoding and decoding procedures [4] are given in Algorithms 5 and 6 respectively.

Algorithm 5 Arithmetic Encoding

encode_symbol($symbol$, cum_freq):
    $range \leftarrow high - low$
    $high \leftarrow low + range * cum\_freq[symbol - 1]$
    $low \leftarrow low + range * cum\_freq[symbol]$
return
for each *symbol* in input **do**
    encode_symbol($symbol$, cum_freq)
    transmit any value in the range [low, $high$)
end for
encode a distinguished "terminator" *symbol*

Algorithm 6 Arithmetic Decoding

decode_symbol(cum_freq):
    $value \leftarrow$ received number
    find *symbol* such that
    $cum\_freq[symbol] <= (value - low)/(high - low) < cum\_freq[symbol - 1]$
    ▷ guarantees $value$ lies within new (low, high) range that will be calculated as :
    $range \leftarrow high - low$
    $high \leftarrow low + range * cum\_freq[symbol - 1]$
    $low \leftarrow low + range * cum\_freq[symbol]$
return *symbol*
while *symbol* ! = "terminator" *symbol* **do**
    $symbol \leftarrow$ decode_symbol(cum_freq)
end while

In the pseudo code symbols are numbered as $1, 2, \ldots, number\_of\_symbols$[3]. Also, frequency range for the i^{th} symbol is between $cum\_freq[i]$ and $cum\_freq[i-1]$. Note that $cum\_freq[0]$ gives the total frequency range which is used for normalization. Although the pseudo code gives the main parts of encoding and decoding algorithms, there are important implementation details such as the use of integer arithmetic when calculating the probabilities, preventing underflow and overflow, and updating the model.

For the encoding part, we start with the adaptive model. Since we numbered symbols as $1, 2, \ldots, number\_of\_symbols$, we create an index to char and char to index translation. Also, as mentioned above, we use cum_freq to represent frequencies of symbols. After setting the model up, we initialize a buffer which is used for writing the encoded data to output. Lastly we initialize $low = 0$ and $high = Top\_value$ which is the largest

[3]In our implementation we take $number\_of\_symbols$ 257 (number of characters plus the terminating character).

possible code value. Also, a data type *code_value* is defined for *low* and *high* variables. Constants $First\_qtr$ (point after first quarter = $Top\_value/4 + 1$), $Half$ (point after $half$), and $Third\_qtr$ (point after third quarter) are defined to be used for preventing overflow and underflow integer arithmetic. After finishing the initialization, we encode each character using *encode_symbol* method and then update the adaptive model. This process continues until we reach the end of the file. In *encode_symbol*, we calculate the range as $high - low$ and narrow the code region for this symbol by calculating high and low value using cum_freq of the symbol. As the code region gets narrow, we have to be careful and make the narrowing such that $low < Half \leq high$. Also, if there are any bits which are the same, they are transmitted without waiting the completion of encoding since those bits do not affect the calculation of code range. Moreover, the variables *high* and *low* are changed and so are encoded bits by using the constants $First\_qtr$, $Half$ and $Third\_qtr$ such that we prevent underflow and overflow. We can prevent underflow by keeping the range greater than $Max\_frequency$, which is the maximum allowed cumulative frequency count. The value of $Max\_frequency$ depends on the Top_value as expected. We can keep the range greater than $Max\_frequency$ by expanding the range using $First\_qtr$, $Half$ and $Third\_qtr$ when it is necessary. Moreover, overflow can be prevented by keeping $range * Max\_frequency$ in the integer word length range. After encoding the symbol, we update our model by changing the cum_freq values. After encoding whole data, we put special "terminator" symbol and encode it also to distinguish the encoded blocks. After encoding the "terminator" symbol we terminate our encoding part. Also, note that each encoding process we successively write the encoded data to the file.

For the decoding part, we first start the adaptive model as in the encoding part and make the initialization. Then, we start decoding by *decode_symbol* and updating the model successively until we get the special terminating symbol which shows us that decoding should be terminated. In *decode_symbol*, we use the value that we got from the encoded data to calculate the freq value. Then by narrowing the code region (updating the *high* and *low* values and so the *range*) we find the input symbol and update the model. Also, as in the encoder case, a care is taken for overflow and underflow. Furthermore, so as to guarantee the decoding process is correct, we make the decoding such that $low + [(high - low + 1) * cum\_freq[symbol]]/cum\_freq[0] <$ input value $< low + [(high - low + 1) * cum\_freq[symbol - 1]]/cum\_freq[0] - 1$. Also, note that each decoding process we successively write the output data.

Performance

In order to evaluate the performance of our algorithm we used the Canterbury dataset [7], which is one of the datasets used as a benchmark for comparing lossless compression algorithms. Compression ratio is defined as average number of bits used to represent a character, which takes 8 bits when it is uncompressed. Table 4 shows the performance of our scheme over those files. Compression ratio ranges from 0.9 to 3.46 bits/char at maximum. If we consider the text files only, the average compression ratio is 2.58 bits/char. Compression performances of various compressors on the Canterbury dataset can be found in [8].

In order to see the performance on the larger size files, we utilize the data in [9].

Table 4: Compression Results - Canterbury Dataset

File name	Initial size (in B)	Compressed size (in B)	Compression ratio (in bits/char)
alice29.txt	152,089	48,359	2.54
asyoulik.txt	125,179	44,170	2.82
lcet10.txt	426,754	121,396	2.28
plrabn12.txt	481,861	161,076	2.67
cp.html	24,603	8,602	2.80
fields.c	11,150	3,284	2.36
grammar.lsp	3,721	1,367	2.94
kennedy.xls	1,029,744	177,055	1.38
ptt5	513,216	58,499	0.91
sum	38,240	13,945	2.92
xargs.1	4,227	1,829	3.46

Table 5: Compression Results - Large text files

File name	Initial size (in B)	Compressed size (in B)	Compression ratio (bits)
enwik8	100,000,000	27,999,100	2.24
text8.txt	100,000,000	26,042,510	2.08

The file "enwik8" consists of the first 100 MB of the XML dump of English Wikipedia on March 3, 2006. "text8.txt", on the other hand, contains the cleaned text file [4] of the XML text dump. Table 5 shows the comparison results. Here we achieve lower compression ratio on the text files than those in the Canterbury dataset.

Conclusion

To sum up, we implemented Burrows Wheeler Transform and Move to Front Coding in order to put the original text file into a form such that it is more compressible when we apply zero order adaptive arithmetic coding which uses statistical information of data. In this project, we originally compressed a part of the novel A Tale of Two Cities whose size is approximately 755 kB. After applying encoding, the size of the compressed file is approximately 232 kB. Hence we achieved roughly 2.5 bits/char compression rate. We also tested the algorithm on different files and get 2.58 bits/char for text files in Canterbury dataset and lower compression ratios for relatively large text files.

References

[1] A. Moffat and A. Turpin, *Compression and Coding Algorithms*, 2002.

[4] For details on how to create clean text file from the XML text dump, please see [9].

[2] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*.

[3] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Systems Research Center Research Report, Tech. Rep., 1994.

[4] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987. [Online]. Available: http://doi.acm.org/10.1145/214762.214771

[5] M. Dipperstein, "Burrows-wheeler transform discussion and implementation." [Online]. Available: http://michael.dipperstein.com/bwt/index.html

[6] M. Nelson, "Data compression with the burrows-wheeler transform." [Online]. Available: http://marknelson.us/1996/09/01/bwt/

[7] "The canterbury corpus." [Online]. Available: http://corpus.canterbury.ac.nz/index.html

[8] "The canterbury corpus, table of compression ratio results." [Online]. Available: http://corpus.canterbury.ac.nz/details/cantrbry/RatioByRatio.html

[9] "Large text compression benchmark, about the test data." [Online]. Available: https://cs.fit.edu/ mmahoney/compression/textdata.html