bachelorproject

Ahmad Othman

April 2022

Abstract

Test abstract

1 Introduction

[**einstein**] [**burrows1994block**]

We are storing and transmitting more and more data every year and as our demand for storing and transmitting data increases so does our demand for better data compressors. There are many advantages in being able to compress a message efficiently. Among these advantages is that we are able to better utilize the storage capacity for both short and long term data storage. This also has benefits in data transmission. By reducing the length of the message we intend to send, we reduce the amount of time needed to send it. For these reasons there are very clear economic and technological incentives to research subjects relating to data compression and to continuously develop better techniques that serve this purpose.

Compressors work by inputting a file, processing it, and then outputting an encoded file. This encoded file is generally shorter than the input file, but contains enough information so that a decompressor is able regenerate the original file from the encoded file. Compressors are able to do this by removing redundant information in the data. Some compressors are developed to only compress certain specific types of data, for example being developed specifically to compress only image files or only DNA files. Other compressors are developed to be more general-purpose and are in many cases able to to compress a greater variety of different types of data. When deciding which method to use for compression there are two major factors to consider: compression ratio and (de)compression speed. For some purposes we might want to compress fast but still with relatively good ratio, while other times compression ratio may be the main focus. This speed vs ratio is a common trade-off, not only in compression. One program that is one of the champions of compression, and which has an excellent compression ratio in terms of its speed is *bzip*2.

*bzip*2 is a general-purpose and lossless compression program originally developed by Julian Seward and now maintained by Mark Wielaard and Micah Snyder. It combines several self-contained techniques and algorithms that are used in data

compression, the core of which being the Burrows–Wheeler transform (BWT).

Where and how BZIP2 is known to perform very well

In this project we wish to examine and justify the used techniques in *bzip*2.....

2 Theory? (Faglig fremstilling)

2.1 Entropy?

Shannon coding theorem
Pidgeonhole principle on compressing

2.2 *bzip*2 Resume

*bzip*2 is an open-source file compression program. It is considered a powerful compressor, generally having a better compression ratio than the more commonly-used *gzip* and *zip*, but having slower compression and decompression times.[1] It is a general-purpose compressor that combines 5 self-contained layers of compression techniques, these layers are

1. Run-length encoding (RLE 1)
2. Burrows-Wheeler transform (BWT)
3. Move-to-front encoding (MTF)
4. Run-length encoding (RLE 2)
5. Huffman encoding (using multiple Huffman trees).

The core of *bzip*2 is the Burrows-Wheeler transform. uses several methods, the core of which being the Burrows-Wheeler transform. The techniques used in *bzip*2 are loosely based on the research article by burrows and wheeler

2.3

2.4 Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler transform (BWT) is the second technique used in the *bzip*2 compressor. It is a vital step and lays the groundwork for the next a It is a reversible data transform that was invented by Michael Burrows and David Wheeler, and released in a research report in 1994. The transform is itself not a compressor but preprocesses data, preparing it for other compression algorithms. It takes a block of data S and outputs another block of data L, and a number I. Burrows and Wheeler showed how given L and I, we can restore the original data S. The output data L is a permutation of the input S but where identical

[1] https://tukaani.org/lzma/benchmarks.html

characters are grouped more together. This grouping together of characters as we will show works very well with other algorithms for compressing the data.

2.4.1 Transform

Given a string $S =$ "*abracadabra*", we wish to perform the BWT on it. We start by introducing a new character \$ to S and appending it to the end of the string. \$ is a character that does not previously appear in S and is lexicographically the smallest character. Thus we have $S' =$ "*abracadabra*\$" We now generate all cyclic shifts of S' and sort them lexicographically.

0	abracadabra$
1	bracadabra$a
2	racadabra$ab
3	acadabra$abr
4	cadabra$abra
5	adabra$abrac
6	dabra$abraca
7	abra$abracad
8	bra$abracada
9	ra$abracadab
10	a$abracadabr
11	$abracadabra

$\longrightarrow$

11	$abracadabra
10	a$abracadabr
7	abra$abracad
0	abracadabra$
3	acadabra$abr
5	adabra$abrac
8	bra$abracada
1	bracadabra$a
4	cadabra$abra
6	dabra$abraca
9	ra$abracadab
2	racadabra$ab

After lexicographically sorting the cyclic shifts (the matrix on the right), our output will be the last column. We will have our output $L =$ "*ard*\$*rcaaaabb*" and $I = 3$. L will be the last column of the sorted cyclic shifts and I is the row number of the original string in the sorted cyclic shifts matrix. Since we have defined each row in the matrix to be a cyclic shift of our original string, then obviously, in the unsorted matrix each row and column is a cyclic shift of S'. After sorting the rows of the matrix, the columns will no longer necessarily be the cyclic shifts of S' but they will still be permutations of it. This algorithm provides a good conceptual way to imagine and explain the transform but is not used in practice since it requires constructing the entire matrix in memory, requiring $O(n^2)$ space.

Burrows and Wheeler observed that when we lexicographically sort the rows of the matrix, the order they end up in is the same order the suffixes are in, in the suffix array. This is the case since like the suffix array of S, when we sort the cyclic shifts of S' we only need to sort up to and including the \$ symbol. This

means that the transform can be computed by constructing the suffix array

i	
0	abracadabra$
1	bracadabra$
2	racadabra$
3	acadabra$
4	cadabra$
5	adabra$
6	dabra$
7	abra$
8	bra$
9	ra$
10	a$
11	$

$\longrightarrow$

i	SA[i]	
0	11	$
1	10	a$
2	7	abra$
3	0	abracadabra$
4	3	acadabra$
5	5	adabra$
6	8	bra$
7	1	bracadabra$
8	4	cadabra$
9	6	dabra$
10	9	ra$
11	2	racadabra$

The BWT can then be constructed directly from the suffix array SA and our original string S. We build the string L for each index $i = 0 \dots |S| - 1$:

$$L[i] = \begin{cases} S[SA[i] - 1] & SA[i] > 0 \\ \$ & SA[i] = 0 \end{cases}$$

The character in $L[i]$ will be the character one step left of the character at index $SA[i]$, which is $S[SA[i] - 1]$. When $SA[i] = 0$ the character will be the special \$ character. In the same example we will have the same string $L =$ "$ard\$rcaaaabb$" and $I = 3$. This reduces the problem of finding the BWT of S to the problem of constructing the suffix array of S. After transforming using the BWT, we can omit \$ in the output since we know the value of I.

2.4.2 Reverse Transform

From L we can construct the sorted containing the sorted cyclic shifts. We start by adding the column L and sorting it. Then we continuously add L as a column at the front and sorting the rows. We do this until the rows have the same length as L.

a	$	a$	$a	a$a	$ab	a$ab
r	a	ra	a$	ra$	a$a	ra$a
d	a	da	ab	dab	abr	dabr
$	a	$a	ab	$ab	abr	$abr
r	a	ra	ac	rac	aca	raca
c	a	ca	ad	cad	ada	cada
a	b	ab	br	abr	bra	abra
a	b	ab	br	abr	bra	abra
a	c	ac	ca	aca	cad	acad
a	d	ad	da	ada	dab	adab
b	r	br	ra	bra	ra$	bra$
b	r	br	ra	bra	rac	brac

The original string S'
This transformation is the core of *bzip*2

2.5 Move-to-front Encoding

The Move-to-Front Encoding (MTF) is the third algorithm used in the *bzip*2 compressor. It is a reversible lossless encoder that, like the BWT, does not change the size of input. It takes advantage of the expected output of the BWT to further processes the data.

2.5.1 Encoding

The encoding process works by replacing each character by its index in a list containing the alphabet. Each time a character is read the list is updated, where that character is pushed to the front of the list. To encode using the MTF encoder we first initialize a list containing the alphabet. In this case we use our previous output from the BWT, the string $L =$ "*ardrcaaaabb*". This is the string we wish to encode. We initialize a list containing the symbols appearing in L in alphabetical order.

0	1	2	3	4
a	b	c	d	r

Now we iterate through L. For each character in L, we replace it by the index in which it appears in the list. Then we pop that character from its position in the list and preprend it to the front of the list.

[a]rdrcaaaabb	a	b	c	d	r
0[r]drcaaaabb	a	b	c	d	r
04[d]rcaaaabb	r	a	b	c	d
044[r]caaaabb	d	r	a	b	c
0441[c]aaaabb	r	d	a	b	c
04414[a]aaabb	c	r	d	a	b
044143[a]aabb	a	c	r	d	b
0441430[a]abb	a	c	r	d	b
04414300[a]bb	a	c	r	d	b
044143000[b]b	a	c	r	d	b
0441430004[b]	b	a	c	r	d
04414300040	b	a	c	r	d

The output of the MTF with input $L =$ "*ardrcaaaabb*" is then "04414300040". The usage of MTF encoding after the BWT step was proposed by Burrows and

Wheeler. The assumption is that after the BWT step, the data will contain runs of identical characters. In this case the MTF will replace these runs of identical characters by a specific number followed by many zeros. This means that after the MTF step, our data will predominantly contain zeros with occasional breaks of different numbers.

2.5.2 Decoding

2.6 Run-length Encoding

The Run-length Encoding (RLE) is an important step in the compression performance of the *bzip*2 compressor. For any type data, if there are sequences of identically repeating characters, there is no need to store all the characters in those sequences of repeating characters. The RLE replaces those sequences with one character identifying what character that sequence is comprised of, and then amount of characters appearing in that sequence.
We show an example of RLE. Assume that we wish to encode "aaaaabbbbcaaaa" using RLE. We know that the string can be represented as being a string consisting of 5 a's, 4 b's, 1 c and 4 a's. We can write this as the following

$$aaaaabbbbcaaaa \rightarrow 5a4bc4a.$$

This does not remove any information of the data but, only sequences data in order We can reduce this data and simply write the amounts.

The RLE step tries to take advantage of the output of the MTF step. Since as previously mentioned, from the output of the MTF step we can expect our data to contain many sequences of consecutive zeros. The RLE will remove those consecutive zeros and instead write the amount of zeros that will appear in that sequence of zeros.

2.7 Huffman Encoding

The last step of *bzip*2 is the Huffman encoder.

3 Implementation

3.1 Multiple Huffman Trees

The *bzip*2 compressor uses multiple Huffman trees in order to efficiently compress the data. The amount of trees it uses is entirely dependent on the size of the data but can be between 2 and 6, although it will usually be 6. Huffman encoding guarantees symbol-code optimality for the entire data, but it is not necessarily optimal for local segments of the data. For this reason, *bzip*2 splits the entire text into segments each of size 50 bytes. It then assigns each It starts by partitioning the characters the frequency of characters

4 Experiments?

4.1 Run-length encoding

4.2 Multiple Huffman

5 Conclusion

In this thesis, we did....
Natural future work could include..

6 Conclusion