Dante D'Amico and Alex Wurst

May 17th, 2021

CPE 593: Applied Data Structures and Algorithms

Term Project: Bzip2 Re-implementation

Abstract:

Bzip2 is a widely used compression algorithm originally created by Julian Seward. This algorithm uses many different techniques and transforms to reduce file size, such as run-length encoding, Burrows-Wheeler transform, move-to-front transform, and Huffman coding. The goal of our project is to document the algorithm along with all of the steps involved, and to re-implement the algorithm, within the scope of text compression and decompression, in C++ according to those steps.

The full steps involved in the Bzip2 algorithm are as follows.

1. Run-length encoding
2. Burrows-Wheeler transform
3. Move-to-front transform
4. Run-length encoding (on transformed data)
5. Huffman coding
6. Huffman table selection
7. Huffman table unary encoding
8. Delta encoding
9. Sparse bit array

For the scope of our project, and to achieve compression and decompression results that can be clearly demonstrated on text files, we will be implementing the first 5 of the algorithms in the Bzip2 stack. This paper will discuss the theoretical aspects of each of the steps involved in our recreation of the Bzip2 algorithm, while also pointing out the goal of each step and how they lend themselves to compressing data. This

paper will also cover how we implemented these individual steps and how we formed them into a cohesive program. The reader will also discover some of the challenges that we faced in the creation of this project.

I. Initial Run-Length Encoding

The first step of the Bzip2 algorithm is run-length encoding. The function of this step is to replace runs of four or more repeated characters with the first four characters, followed by the subsequent repeat length. The repeat length is a byte value between 0 and 251. The example below demonstrates this:

```
AAAAAAAAAABBCCCCCCCCCCCC → AAAA6BBCCCC8
```

In run-length encoding, all runs of four or more characters are converted to this format, even if the run length after the first four characters is 0, for the sake of reversibility. For example:

```
AAAA → AAAA0
```

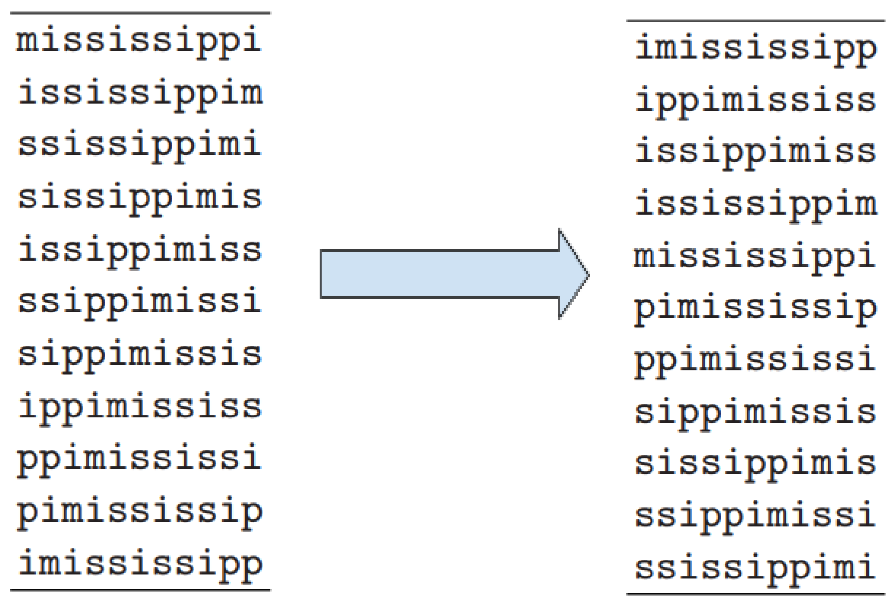
The goal of this stage is compression. In the worst-case scenario, the data can be expanded by 25% of its original size, which would occur if the original data consisted of only runs of 4 consecutive characters. But the best case scenario of this step is that it can reduce the data to 2% of its original size.

To perform run-length encoding, we read the stream character by character until a new, non-repeating character is found. At this point, we determine how to write the compressed data; if there are four or more repeating characters, we write the character four times followed by the number of subsequent repeats. If there are fewer than four, we simply write the characters directly. The reversal of this process is intuitive, as we can read encoded text, and determine and output the original text from the same simple protocol that defines run-length encoding.

II. Burrows-Wheeler Transform

The Burrows-Wheeler Transform is a core step of the Bzip2 compression stack that is a reversible block-sort, with the intention to reorder data to make it easier to compress with the algorithms that follow this step. As the name suggests, this is a transform, and does not actually compress the data.

The transform takes place on a string of *N* characters, upon which this string is rotated, or cyclically shifted, *N* times. These rotations are then sorted lexicographically. The following figure demonstrates these shifts on an original input string “mississippi” and the lexicographical sorting of these rotations :



mississippi	imississipp
ississippim	ippimississ
ssissippimi	issippimiss
sissippimis	ississippim
issippimiss	mississippi
ssippimissi	pimississip
sippimissis	ppimississi
ippimississ	sippimissis
ppimississi	sissippimis
pimississip	ssippimissi
imississipp	ssissippimi

Note that of all the rotations of the original string, at least one rotation is the same as the original string. The index of this occurrence in the array of rotations is kept as an "origin pointer," which will be needed to reverse this transform later.

The output of the Burrows-Wheeler transform is a string, which is comprised of the last character of each rotation in the lexicographically sorted array. Looking at the above figure again for reference, it can be seen that the BWT of the string "mississippi" is "pssmipissii."

In our implementation of the transform, an input stream is read into a string, while the length (*count)* of the string is also accounted for. From this string, a vector of *count* number of cyclic rotations is created. The elements of the vector are then lexicographically sorted. The "origin pointer" is found to keep track of the index of which the original string appears in the sorted vector of rotations. Finally, the last character of each rotation in the sorted vector is added to the output stream as the final result.

To decode the BWT, we use the result of the forward BWT to reconstruct the original string. Recall that the result of the forward BWT is the last column of all the cyclic rotations of the original string. Sorting these characters lexicographically results in the first column of the rotations of the original string. With this, we can reconstruct the entire original list of rotations, and use the origin pointer to retrieve the original string.

The way we implemented this decode was to first construct a string from the stream (string *S*). A vector of strings of length *N* is initialized. We then insert the individual characters of *S* into the first position of each string. The strings are sorted. We again insert the characters of S into the first position of each string, sort the vector again, and repeat this process N times. The result of this is the lexicographically sorted vector of cyclic rotations of the original string. From this, we pick the original string using our origin pointer, and output that string.

III. Move-to-Front Transform

The goal of this next transform is to reduce the entropy of a stream. Each character is replaced by its index in a stack of recently used symbols. Long runs of repeated characters result in a chain of zeros and a character that has not appeared recently would be replaced by a larger integer. To better understand this, we can think of the english alphabet as a stack, indexed from 0 to 25, starting with ‘a’. We will use the string “mississippi” once again.

1. **m** is at index 12, so 12 is added to the sequence. **m** is now moved to the front of the stack (to index 0).
2. **i** is at index 9, 9 is added to the sequence. **i** is moved to the front of the stack
3. **s** is at index 20, added to sequence, moved to front
4. **s** is at index 0, added to sequence, still at front
5. **i** is at index 1, added to sequence, moved to front
6. **s** is at index 1, added to sequence, moved to front
7. **s** is at index 0, added to sequence, still at front
8. **i** is at index 1, added to sequence, moved to front
9. **p** is at index 18, added to sequence, moved to front
10. **p** is at index 0, added to sequence, still at front
11. **i** is at index 1, added to sequence, moved to front

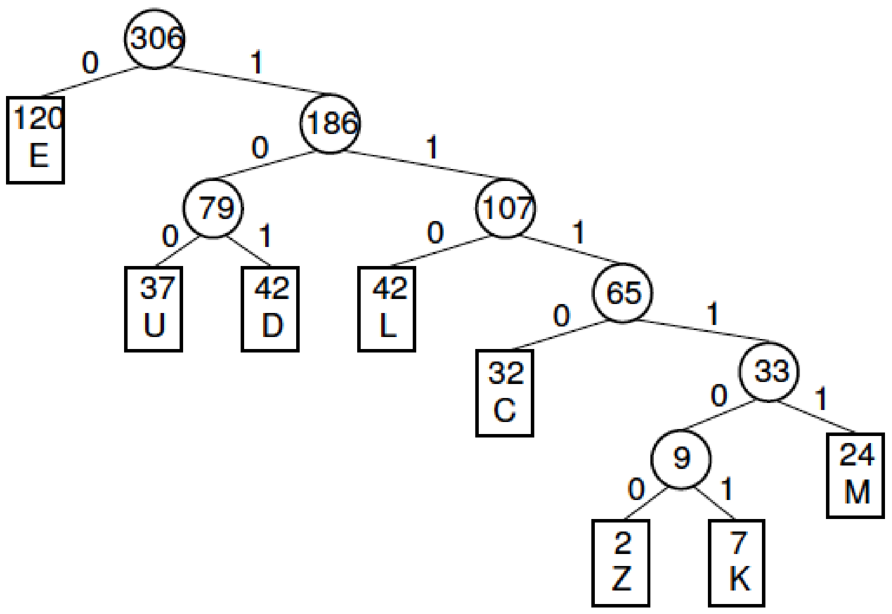
These steps result in an output of integers, [12, 9, 20, 0, 1, 1, 0, 1, 18, 0, 1]. To decode this, we would again start with the original english alphabet stack, and use the index values to form the text. Index 12 would be **m,** which then gets moved to the front of the stack, and so on.

In our implementation, we simply perform these steps in code. We first initialize a symbol table. For each character in the stream, we then look for it in the symbol table, output the symbol's position in the table, and move that symbol to the front.

To decode, we read the list of integers and output the symbol located at the index position in the symbol table, and move that symbol to the front of the table. We repeat this until the original string is output.

IV. Huffman Coding

Huffman coding is the next critical step in the Bzip2 compression stack, and is used to compress data. The main goal of this algorithm is to map symbols of high frequency to codewords with smaller bit-lengths. This method involves a Huffman Tree data structure, which is a binary tree generated from the frequencies of symbols in the text. The figure below demonstrates a Huffman Tree with characters and their frequency values:



For example: In this tree, the symbol **E** has a frequency of 120, and a code of 0. The symbol **Z** has a frequency of 2, and a code of 111100. Once a Huffman Tree has been generated, the tree is then traversed to create a dictionary which maps the symbols to their corresponding codes. The codes are generated by traversing the tree and labelling each edge to the left of a given node as “0” and labelling the right edge “1.” By recursively visiting every node and keeping track of the path taken, each symbol-code mapping can be added to a hashmap.

For the reversal of Huffman coding, “the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream” (Wikipedia).

Our implementation of Huffman coding also aligns closely with this description. First, a hashmap is created for the symbol-code mappings to be stored. Then a histogram of the characters in the text and their frequencies is generated to be used in the creation of the Huffman tree. A priority queue is used to generate the tree structure, where priorities are determined such that the symbols with the highest frequencies have the highest priority. Initially, a node is created for each symbol and added to the priority queue. Then, pairs of nodes are dequeued and an internal node is created. This internal node has an arbitrary symbol and a frequency equal to the sum of the dequeued nodes' frequencies. These dequeued nodes are then set as children to this internal node, and the internal node is added back to the queue. This process repeats until there is only one node remaining in the queue, which is the root. Once the tree is created, then the symbols in the text stream are mapped to the Huffman encoded values. The encoded symbols are finally written to output.

To decode a Huffman-encoded stream, it is necessary to have the symbol-code mappings, as well as the bit lengths of the codes. This is because the encoded symbols have variable widths, with the higher-frequency symbols being mapped to smaller codes. As such, the size of each symbol is saved during the decoding process using a delta encoding method. This means that the change in bit length is saved after writing each bit to the output file. When decoding, these delta values can be used to determine how many bits must be read for the next symbol to decode. The decoding process uses the same mappings as the encoding process and outputs the matching symbol for each received code to determine the decoded output.

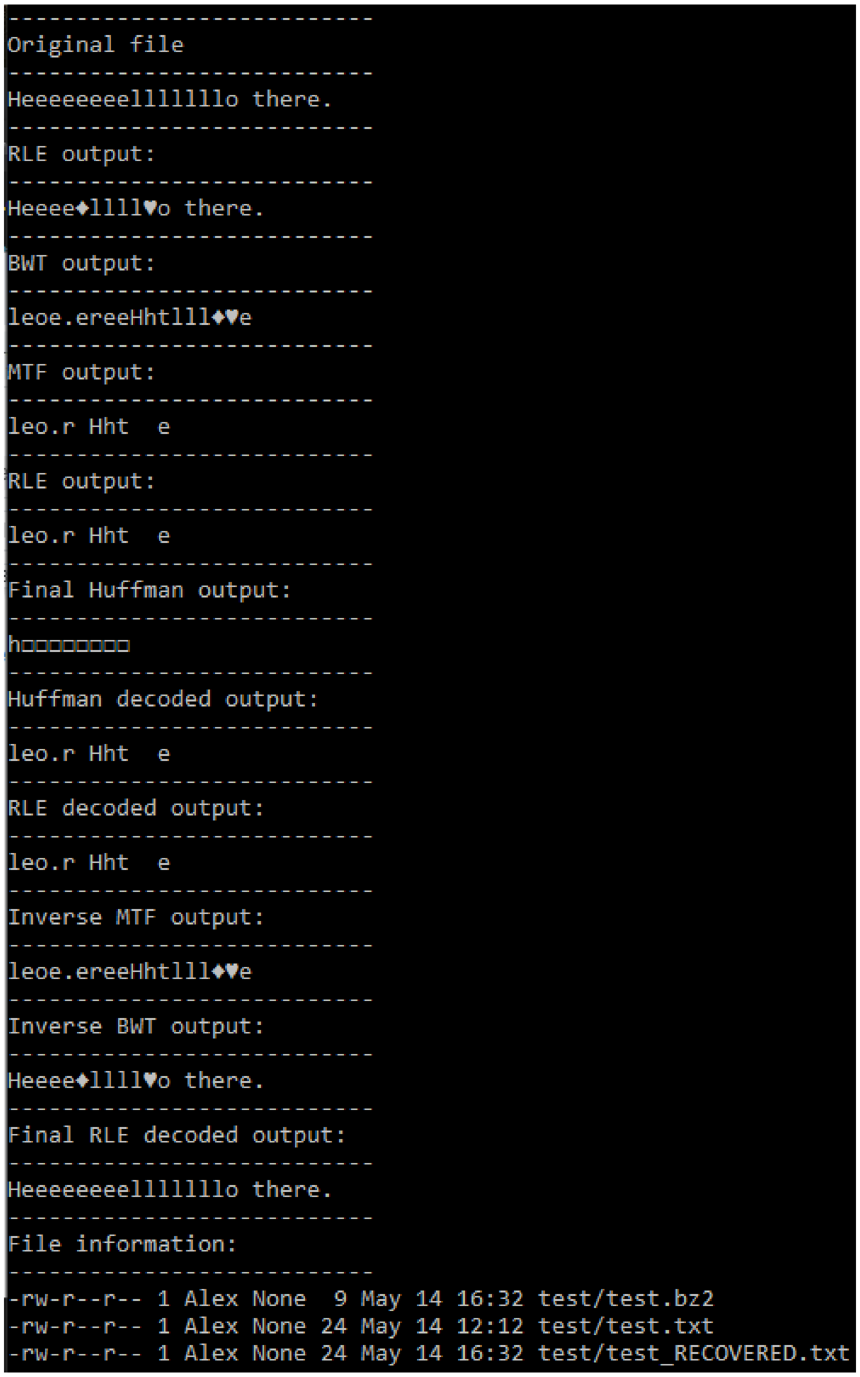
V. Challenges

Initially, our hope and intention was to create a program capable of not only compressing and decompressing text files, but also having the functionality of being compatible with the actual Bzip2 program. In other words, we aimed to have our program be able to decompress Bzip2 files, and have our program compress files that are able to then be decompressed using Bzip2. As we worked our way through learning about the various algorithms in the Bzip2 stack and worked on implementing them, we met obstacles that prevented us from reaching this initial goal. To communicate the nature of these obstacles, it may be best for us to directly describe the issues we found facing us.

We first faced this challenge when implementing the Burrows-Wheeler transform. In order for the transform to be decoded/reversed, information from the encoding stage is needed. Within the confines of our own program, this is not a problem, as we are in complete control of how this necessary information could be passed along to the functions handling the decoding of the transformation. But in perspective of trying to pass this information to the Bzip2 program, we were not able to determine where or how to do so. The same situation also faces us in the opposite scenario, where we were unable to determine how to receive the information aside from the stream of characters of an encoded Bzip2 file necessary to decode the file and reverse the transformation.

VI. Demo

The following terminal output shows a simple demonstration of compressing and decompressing a small text file. Output files are written after each step of the encoding and decoding processes, and are shown in the terminal.As the figure shows, each step is performed and the file is successfully recovered after being compressed without any errors.



```
---------------------------
Original file
---------------------------
Heeeeeeeelllllllo there.
---------------------------
RLE output:
---------------------------
Heeee♦llll♥o there.
---------------------------
BWT output:
---------------------------
leoe.ereeHhtlll♦♥e
---------------------------
MTF output:
---------------------------
leo.r Hht  e
---------------------------
RLE output:
---------------------------
leo.r Hht  e
---------------------------
Final Huffman output:
---------------------------
h□□□□□□□□
---------------------------
Huffman decoded output:
---------------------------
leo.r Hht  e
---------------------------
RLE decoded output:
---------------------------
leo.r Hht  e
---------------------------
Inverse MTF output:
---------------------------
leoe.ereeHhtlll♦♥e
---------------------------
Inverse BWT output:
---------------------------
Heeee♦llll♥o there.
---------------------------
Final RLE decoded output:
---------------------------
Heeeeeeeelllllllo there.
---------------------------
File information:
---------------------------
-rw-r--r-- 1 Alex None  9 May 14 16:32 test/test.bz2
-rw-r--r-- 1 Alex None 24 May 14 12:12 test/test.txt
-rw-r--r-- 1 Alex None 24 May 14 16:32 test/test_RECOVERED.txt
```

VII. Results and Conclusions

This project has shown the successful re-implementation of the Bzip2 file compression and decompression algorithm using the C++ programming language.

After running tests on some sample files, files are compressed to at least 50% of their original size, or often even smaller. For the most part, files are also able to be decompressed successfully without any errors or out-of-place characters that were not in the original file. However, this applies mostly to smaller files (kilobytes). One problem that arose during testing was that very large files were not recovered perfectly when decompressing. As such, there is still room to improve upon this implementation. For example, using smaller blocks when compressing and decompressing files would likely lead to better and more reversible results, with the potential tradeoff of a longer execution time.

Works Cited

(2008) How the Burrows-Wheeler Transform works. In: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-78909-5_2

Burrows, M, and D.J. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Systems Research Center.

“Bzip2.” *Wikipedia*, Wikimedia Foundation, 4 Mar. 2021, en.wikipedia.org/wiki/Bzip2.

“Huffman Coding.” *Wikipedia*, Wikimedia Foundation, 22 Apr. 2021, en.wikipedia.org/wiki/Huffman_coding.

“Move-to-Front Transform.” *Wikipedia*, Wikimedia Foundation, 13 Apr. 2021, en.wikipedia.org/wiki/Move-to-front_transform.