The Implementation and Comparison of the BCCBT Data Compression Algorithm

Aiden Taylor - B.Sc. in Computer Science
Noah Pinel - B.Sc. in Computer Science
Ty Irving - B.Sc. in Computer Science

***Abstract*—When storing, transferring, or receiving files it is important to have data compression algorithms to make these processes easier and more efficient. In our project we dive into the actual implementation of a proposed data compression algorithm, and use certain factors of comparison to guage our implementation against other open source data compression algorithms.**

I. Introduction

The purpose of this paper is to take a deeper dive into the implementation and comparison of data compression algorithms. In the modern age, data compression algorithms have become an integral part of our daily lives. These algorithms help us save space on our hard drives, use up less bandwidth, speed up communications, and so much more.

Unfortunately, all these great benefits do not come without costs. In reality, data compression algorithms can be incredibly difficult to devise, and even more difficult to implement. Devising a data compression algorithm requires rigorous mathematical justifications for all the processes taken, and then implementing a data compression algorithm requires intimate knowledge of optimization and memory management techniques.

Once a data compression algorithm is devised and implemented, then there will be a need to see how it stacks up against the other readily available options. No one algorithm excels at compressing every type of data [1], so comparing a data compression algorithm against other available options can highlight the areas where an algorithm exceeds, or where it falls behind. The most important factors of comparisons to note are the compression and decompression times, and how much of the original file the algorithm is able to save without losing any data from the source file.

In this study, we decided to focus on the already devised Bit Code Complete Binary Tree (BCCBT) data compression algorithm which was proposed by Mattias Håkansson Sjöstrand in his 2005 Master's Thesis [1]. Using the pdseudocode provided in Sjöstrand's thesis, we first implemented the algorithm, which was written mainly in C with some accompanying Python scripts, then we used four factors of comparison to guage our implementation against two other open source data compression algorithms. From these comparisons we make conclusions about where our implementation excels, or where it falls behind.

II. Proposed Work

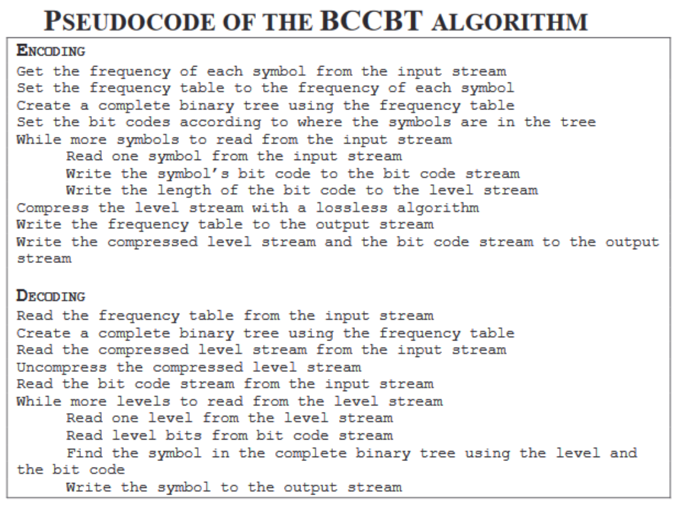
III. Evaluation

A. Implementation

Our implementation of the BCCBT data compression algorithm closely follows the pseudocode provided in Sjöstrand's thesis, which can observed in Figure 1. At a high level, our implementation of the BCCBT data compression algorithm can essentially be split up into two different sections, those being the implementation of the Complete Binary Tree, and the implementation of the Encoding and Decoding of files. To help understand both of these sections we will use example 3-11 from the thesis that as this example does an excellent job of explaining how the BCCBT algorithm works at a high level [1]. Finally, we will briefly discuss the developed software and how it can be evaluated.

1) Complete Binary Tree: To start example 3-11, suppose that we have an alphabet

$$\Sigma = \{a, b, c, d, e, f, g, h\}$$

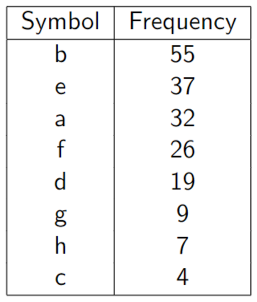


PSEUDOCODE OF THE BCCBT ALGORITHM

```
ENCODING
Get the frequency of each symbol from the input stream
Set the frequency table to the frequency of each symbol
Create a complete binary tree using the frequency table
Set the bit codes according to where the symbols are in the tree
While more symbols to read from the input stream
     Read one symbol from the input stream
     Write the symbol's bit code to the bit code stream
     Write the length of the bit code to the level stream
Compress the level stream with a lossless algorithm
Write the frequency table to the output stream
Write the compressed level stream and the bit code stream to the output
stream

DECODING
Read the frequency table from the input stream
Create a complete binary tree using the frequency table
Read the compressed level stream from the input stream
Uncompress the compressed level stream
Read the bit code stream from the input stream
While more levels to read from the level stream
     Read one level from the level stream
     Read level bits from bit code stream
     Find the symbol in the complete binary tree using the level and
the bit code
     Write the symbol to the output stream
```

Fig. 1. [1]



Symbol	Frequency
b	55
e	37
a	32
f	26
d	19
g	9
h	7
c	4

Fig. 2. [1]

taken from a source file where each symbol has a frequency given from the table in Figure 2. Looking at the pseudocode, we now want to construct a complete binary tree that will allow us to set the bit codes of each symbol based on its location in the binary tree. Before we construct the complete binary tree in this example, lets first note some of important properties of a complete binary tree:

- All levels of a complete binary tree are completely full except possibly the lowest level.
- The complete binary tree is filled in from top down and left to right at each level.
- The number of nodes in a complete binary tree at level n is 2^n ($n \in \mathbb{Z}_9$).

The last of these properties is really only important for Theorem 3-1 in the thesis [1], and we do not actually use this Theorem's optimization technique in our implementation. Now back to the example, the psuedocode says to construct the complete binary tree using the frequency table. This means that we insert the highest frequency symbol b as the root node, then at the next level of the binary tree starting

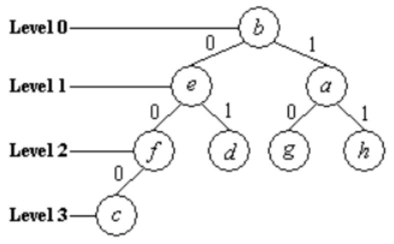


Fig. 3. [1]

from the left we fill in the next highest frequency symbol e, then moving one node to the right at the same level we fill in the next highest frequency symbol a, and so on. Continue this process until there are no more unique symbols to add into the binary tree. This will result in the binary tree having the required properties of a complete binary tree. The complete binary tree generated in this example can be seen in Figure 3. Now with this complete binary tree we will move on to the next section where we can see how the bit codes are generated, how the unique symbols are encoded, and how an encoded string can be uniquely decoded.

2) Encoding and Decoding:

3) Developed Software:

- **bccbt.c:** This file contains the implementation for both constructing and searching the Complete Binary Tree, which in turn contains the implementation for encoding symbols to character arrays of 1's and 0's, and decoding character arrays of 1's and 0's into their corresponding symbols.
- **bitarray.py:** This file contains the implementation for converting a character array of 1's and 0's to actual bits that can be written to a binary file.
- **bitpull.py:** This file contains the implementation for converting a binary file to a character array of 1's and 0's.
- **Makefile:** This file contains tests with the required sequence of commands needed in order to properly execute our implementation.

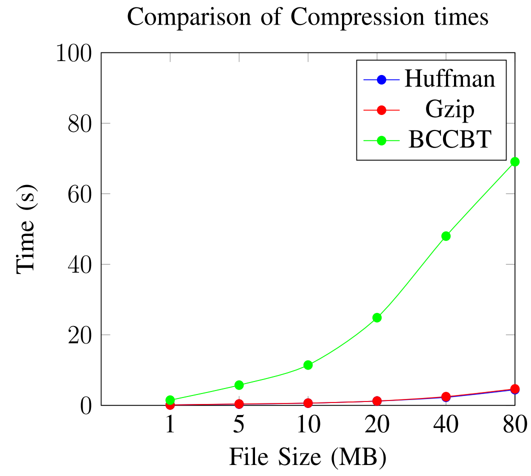
Our software can simply be evaluated through the Makefile which contains pre-built tests for various different file sizes, where each of these tests can be edited at the discretion of the user.

B. Experiments

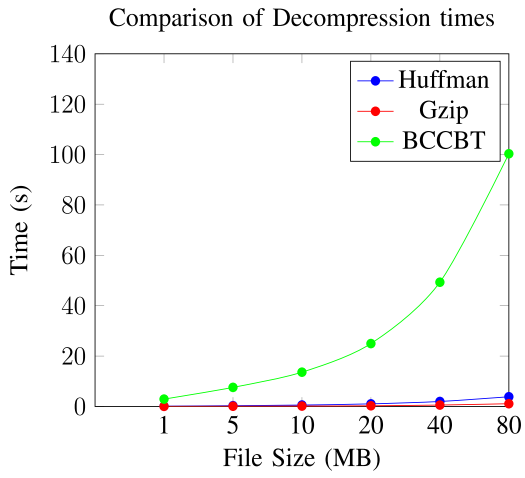
In order to effectively be able to analyze the effectiveness of the BCCBT algorithm we had to find two other comparable open source compression algorithms to test against. As a refresher for the factors that we had used to compare the 3 algorithms against eachother we used the 4 following factors

1) Compression Time
2) Decompression Time
3) Saving % = $\frac{Orig\ File\ Size - Compressed\ File\ Size}{Orig\ File\ Size}$
4) Compression Ratio = $\frac{Compressed\ File\ Size}{Original\ File\ Size}$

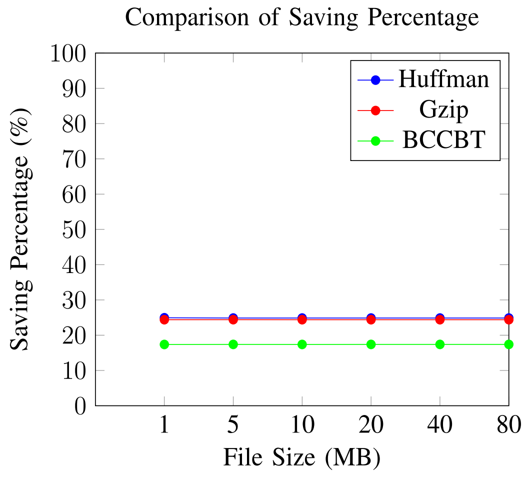
Using these factors will enable us to effectively test whether the BCCBT algorithm is both effective for compressing files of differing sizes along with showing whether or not it is practical and where the use cases for this compression algorithm.



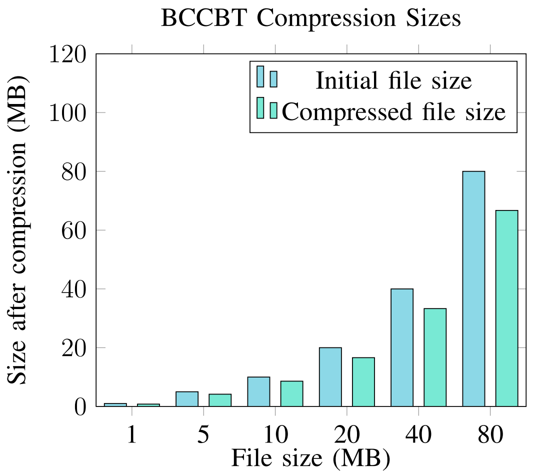
These results indicate that although for low file sizes the compression time is quite similar as the file size gets bigger the compression time exponentially gets bigger and the gap between both Huffman and GZip gets bigger and bigger this result indicates that for the implementation that we made it would not be practical or in your interest to use this on file sizes above 1MB however the compression time of this algorithm can be cut down in further implementations of the algorithm. Through our tests we were able to find out why the compression times of our implementation of the BCCBT algorithm were slower than GZip and Huffman and it was due to the population of the complete binary tree and traversing through the tree recursively rather than iteratively which had a heavy impact on the compression time making the implementation much slower when compared to the Huffman and Gzip compression algorithms.

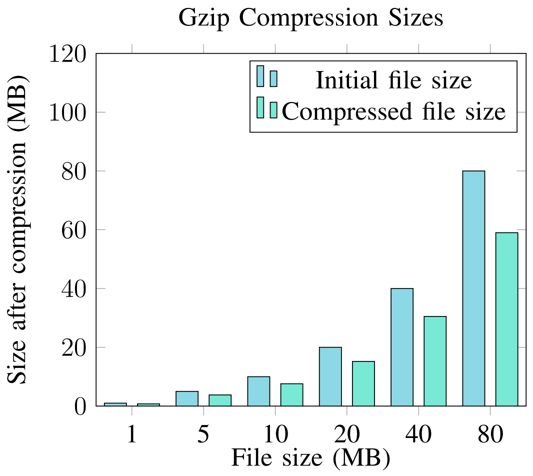


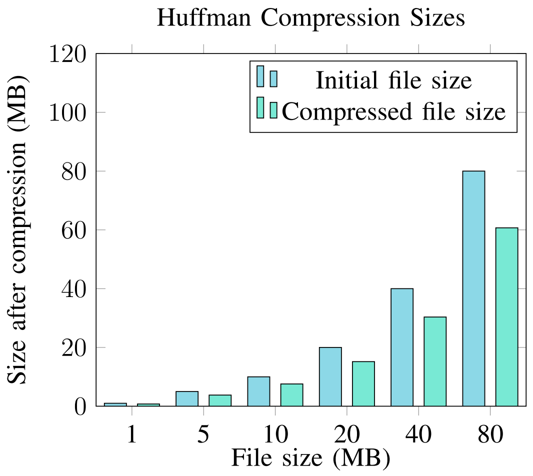
The results for the comparision of the decompression times are quite similar to the compression times mainly because while decompressing we use the similar functions to the compression for the files this graph also represents the problem with the optimization of the compression and decompression of the implementation of the BCCBT algorithm while at higher file sizes.



The graph above demonstrates the saving percentage in % which shows the comparision of the different algorithms for compression and the overall effectiveness of each one depending on file size, this demonstrates that the saving percentage is linear and the gap between the algorithms do not end up becoming bigger unlike the compression/decompression times.







The compression size graphs for the Huffman, Gzip, and BCCBT all show that no matter what the file size is the amount the file size is being reduced by stays linear. Another important part to note about the graph is that the BCCBT does not make the compressed file size as small as the other compression algorithms further indicating how the algorithm is not as efficient as the opensource algorithms that we had compared our implementation against. In conclusion based on the experiment results that have been presented the BCCBT algorithm is a efficient compression algorithm however the implementation that compresses and decompresses the bitcodes, lvls, and frequency files make the distinction of usability clear as the implementation of the BCCBT algorithm performs much slower making it less practical for high file sizes.

IV. Findings and Conclusion

Overall the results of our experiments have shown us that the algorithm performs slightly worse when compared to the open sourced compression algorithms that were used as comparision. Although the results for the time of compression and decompression were heavily influenced by the time available in order to implement the compression algorithm from scratch the compression sizes were quite similar to the algorithms used to compare. The main takeaway from the results is that no matter the file size it stays constant on the size that the source file is going to be compressed into and it does not deviate unlike the time used to compress and decompress. We can say with confidence that the compression algorithm proposed is an effective algorithm to used to compress any file size given time does not play a factor.

V. Planning and Execution

When comparing our final work with the project proposal, it is clear to see that we were effectively able to achieve our main goal which was to implement the BCCBT algorithm in order to compare it against other compression algorithms. Having been able to successfully implement the BCCBT compression algorithm we also had to compare it against some other compression techniques and we were successfully able to achieve that as well and the results that we achieved were as expected. The largest challenge within our project was successfully implementing the BCCBT algorithm since we had to write the compression and decompression from scratch, this majorily impacted us because of the time constraint we had for this project the optimization of the code implemented was not as optimal as it could have been. Given more time or any future work on this project it would be able to yield better results due to optimization.

The work throughout this assignment was mainly split across the group members for every segment that we split the project up into. All members of the group worked on the implementation splitting that into encoding and decoding along with the setup of the binary trees used within those functions. The presentation and papers were evenly split, with each group member working on their assigned seciton mainly focusing on what they did during the implementation and research.

References

[1] M. H. Sjostrand, "A study in compression algorithms," 2005. [Online]. Available: http://bth.diva-portal.org/smash/record.jsf?pid=diva2:830266