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 Thesisframework. 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

Our proposed work is intended to be a re-implementation of the BCCBT algorithm proposed by Sjostrand for data compression. To achieve this goal, we have proposed a three-part effort that involves implementing the BCCBT algorithm, testing its performance on various sized text files, and evaluating its effectiveness compared to state-of-the-art compression algorithms. By undertaking this project, we aim to contribute to the field of data compression and provide insights into the effectiveness of the BCCBT algorithm.

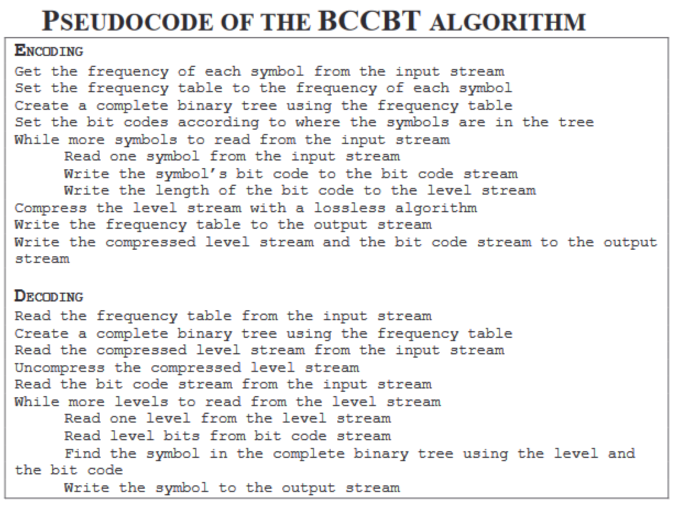
To begin our work, we first configured the necessary software to emulate the BCCBT algorithm. This involved developing a collection of Python programs and a main C program to enable us to accurately simulate the algorithm's behavior. Next, we evaluated the performance of the BCCBT algorithm by conducting tests to measure its compression effectiveness against two other popular compression

algorithms, Huffman and DEFLATE. To perform these tests, we used a diverse set of six .txt files of varying sizes, ranging from 1MB to 80MB. By comparing the compression ratios and processing speeds of each algorithm, we were able to determine the effectiveness of the BCCBT algorithm. Finally, we assessed the effectiveness of our implementation by replicating the evaluation process described in the original paper [1]. Using the same metrics as the original work, we measured various compression thresholds to compare our results with his.
Overall, this framework allowed for us to accurately develop and test our results seamlessly while still being precise with our scientific approach.

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



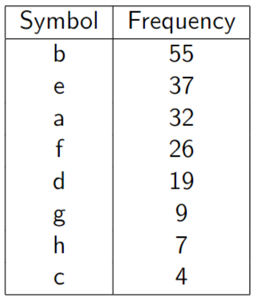
PSEUDOCODE OF THE BCCBT ALGORITHM

```
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: Pseudocode from [1]

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.



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

Fig. 2: Adapted frequency table from [1]

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

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

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 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.

2) Encoding and Decoding: In this section we will walk the reader through an example run-through of the algorithm. Below we will demonstrate the three main steps in encoding and decoding

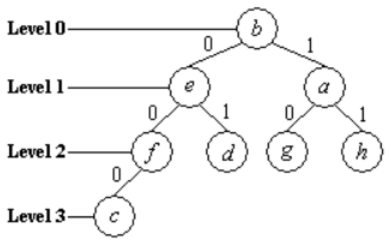


Fig. 3: Complete binary tree used in [1]

using the BCCBT algorithm, it is important to note that this is merely a toy example for the purpose of understating.
Firstly, we need to produce a frequency table, this table is ordered from the highest occurrences to lowest.

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

Fig. 4: Frequency table for symbols

Once we have created the frequency table for our alphabet Σ, we can use it to construct a complete binary tree. The highest occurring symbol in Σ is placed at the root of the tree, and the remaining symbols are added to the tree from left to right in order of their appearance in the frequency table. Note, If we were to traverse the binary tree that we constructed and record the symbols that we encounter as we move from left to right, we will obtain the same sequence of symbols as in the original frequency table, provided that the tree has been constructed correctly. Below is the corresponding complete binary tree that we will use for this demonstration.

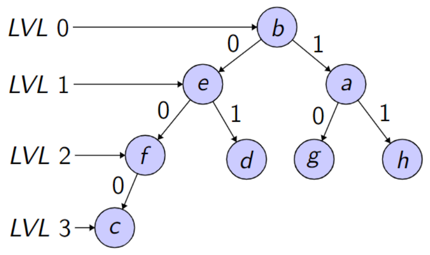


Fig. 5: Complete binary tree

We have now set the bedrock for everything we need to do with our algorithm, whether it be encoding a string, or decoding an encoded one. Before we get to encoding we need to establish the last piece of the puzzle, that is, the corresponding bit codes for the symbols in Σ. In order to generate the bitcodes for each node in the tree, we must follow some simple steps to retrieve them. The process is simple,

1) Position yourself at the root node of the tree
2) Traverse the tree, keeping note of the weights on every edge the was needed to get to your desired symbol
3) After reaching your symbol, you should have a string of 1's and 0's that act like coordinates for finding the specific node.

So, for example, if we wanted to find the bitcode for the symbol 'h', we start at the root of our tree and traverse two right edges, b to a and then a to h, concatenating the edge weights we saw in our walk we get the bitcode 11 for node h. Lastly, before we move on it is important to note that the root of the binary tree i.e. the most occurring symbol in Σ take no bitcode value, it is set as null. Below is the complete table of bitcodes corresponding to all of our symbols in Σ.

BIT CODES							
a	b	c	d	e	f	g	h
1	NULL	000	01	0	00	10	11

Table of bitcodes

This is good and all, but unfortunately, there is a problem with the current bit codes we have constructed from the frequency table. They are not sufficient to encode a string because they are not uniquely decodable. This means that a stranger looking at the complete binary tree alone would not be able to determine what the following string 111' corresponds to - is it three a's or one h' followed by an a'? This lack of uniqueness means that we cannot decode the encoded string without losing information, and if we cannot decode it, we cannot expect a computer to be able to do so either. Therefore, we need to modify our bit codes to ensure that they are uniquely decodable. The way that the BCCBT algorithm handles this is quite clever, a simple yet elegant tactic that will allow for us to uniquely decode, what we do is simply take the corresponding level where the symbol node lays in the tree, and prepend that level value to the bitcode, so for example, ‘h’ had a bitcode ‘11’ but after making this change the bitcode is now [2]11, So with this now, there can’t be any confusion, you note that the symbol is on the second level of our tree, and then take the walk given, 1, then 1 again, and we arrive at h! This is everything we need, we can now generate uniquely decodable strings that will allow lossless decoding.

Lets encode the word ‘edge’. Using our binary tree we take note of the following 3 bitcodes that are generated by iterating the simple bitcode generation steps.

1) e $\rightarrow$ [1]0
2) d $\rightarrow$ [2]01
3) g $\rightarrow$ [2]10

we now have our encoded string

$$\text{‘edge’} \rightarrow [1]0[2]01[2]10[1]0$$

we can now proceed to simulate the decoding of this string.

To decode the encoded string [1]0[2]01[2]10[1]0’, we first create a complete binary tree, which we’ll need for decoding. The decoding process is almost as simple as encoding. We begin by examining the first level token, [1]’, and extracting all the bitcodes between this level token and the next one. In this case, we find the code 0’ and stop grouping since the next character is a level bracket. This gives us our first encoding block, [1]0’. We know with certainty that the desired symbol is in the first level, and we can find it by traversing one 0 edge down. In this case, we land on the symbol ‘e’. We repeat this process for the rest of the string, resulting in the following decoding mappings:

1 0 $\rightarrow$ e
2 01 $\rightarrow$ d
2 10 $\rightarrow$ g
1 0 $\rightarrow$ e

which tells us that our decoded string is

$$[1]0[2]01[2]10[1]0 \rightarrow \text{‘ edge’}$$

and with that we conclude this thorough example of how the BCCBT algorithm goes about encoding and decoding strings of text.

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.

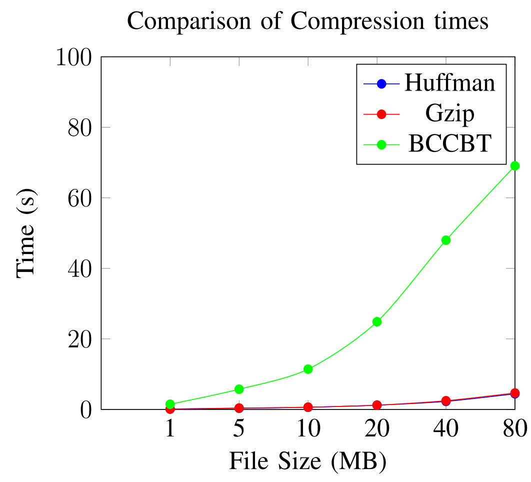
Our software can simply be evaluated through the Makefile which contains pre-built tests for various different file sizes, where each of these tests can be edited at the discretion of the user.

B. Experiments

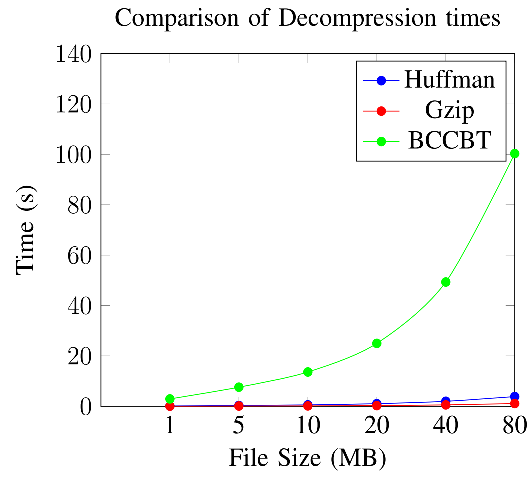
In order to effectively be able to analyze the effectiveness of the BCCBT algorithm we had to find two other comparable open source compression algorithms to test against. As a refresher for the factors that we had used to compare the 3 algorithms against eachother we used the 4 following factors

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

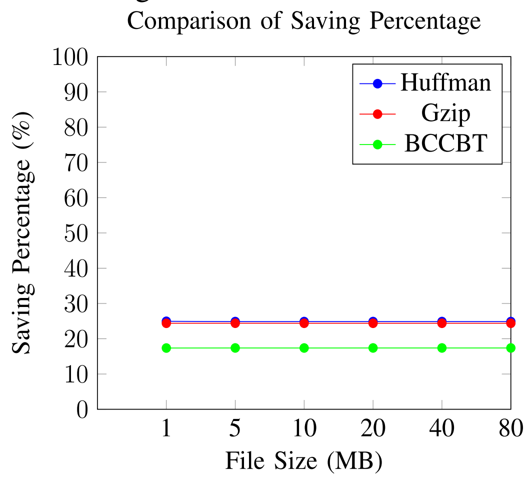
Using these factors will enable us to effectively test whether the BCCBT algorithm is both effective for compressing files of differing sizes along with showing whether or not it is practical and where the use cases for this compression algorithm.



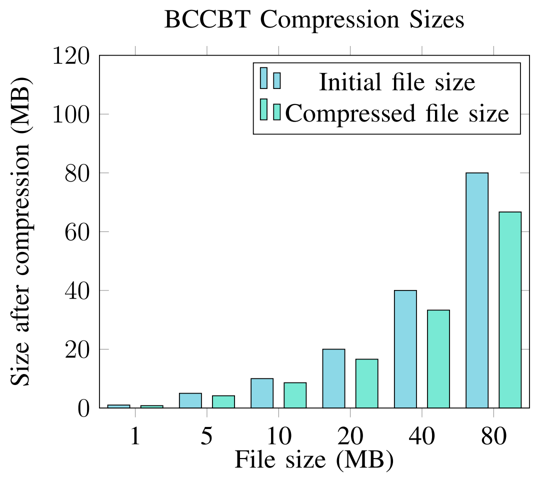
These results indicate that although for low file sizes the compression time is quite similar as the file size gets bigger the compression time exponentially gets bigger and the gap between both Huffman and GZip gets bigger and bigger this result indicates that for the implementation that we made it 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.

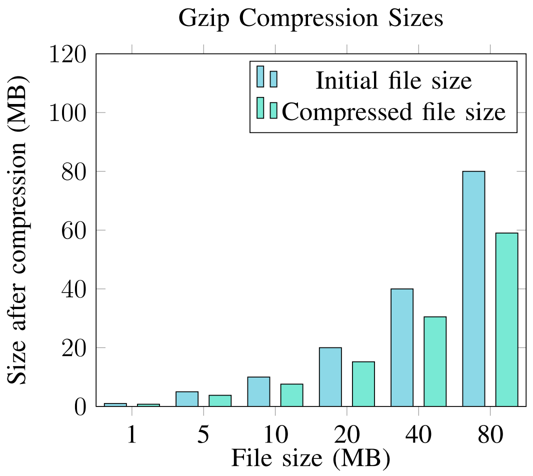


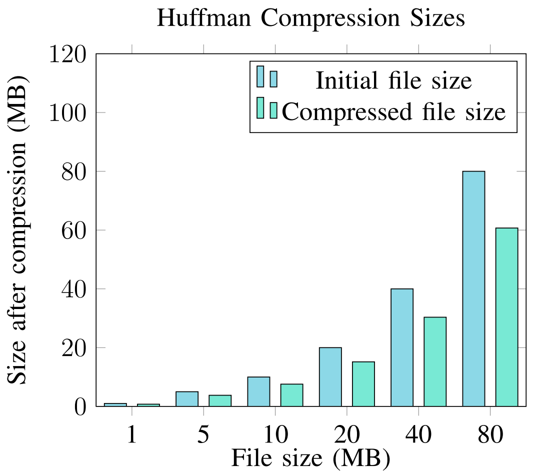
The results for the comparision of the decompression times are quite similar to the compression times mainly because while decompressing we use the similar functions to the compression for the files this graph also represents the problem with the optimization of the compression and decompression of the 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