The Implementation and Comparison of the BCCBT Data Compression Algorithm

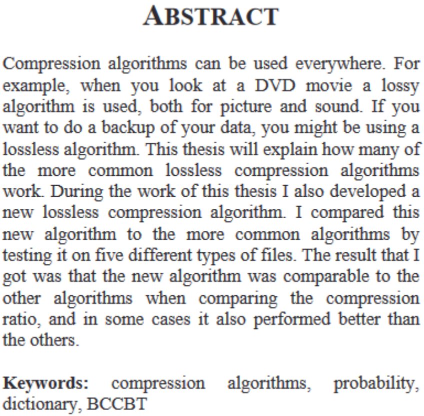
Aiden Taylor - B.Sc. in Computer Science
Noah Pinel - B.Sc. in Computer Science
Ty Irving - B.Sc. in Computer Science

Apr. 11th, 2023

1. Quiz Questions
2. Pseudocode
3. Example
    - Complete Binary Trees and Frequencies
    - Encoding and Decoding
4. Test Results
    - Factors
    - Test Result Graphs
    - Summary
5. Q&A

1. What specific type of Binary Tree is used in the implementation of the BCCBT Data Compression Algorithm? Describe at least one discussed property of this type of Binary Tree.
2. Given the specific binary tree needed for the BCCBT algorithm, where the tree's nodes correspond to symbols in an arbitrary alphabet Σ. Denote a symbol ϕ, such that ϕ is in our arbitrary alphabet Σ. Note, ϕ is **NOT** the root of the tree. How is the bit code generated for the symbol ϕ?
3. Does the BCCBT Data Compression Algorithm make use of the frequency/probability of each unique symbol in the source file? Explain why or why not.

Our project is an implementation of the BCCBT Data Compression Algorithm proposed in Mattias Sjostrand's Master Thesis.

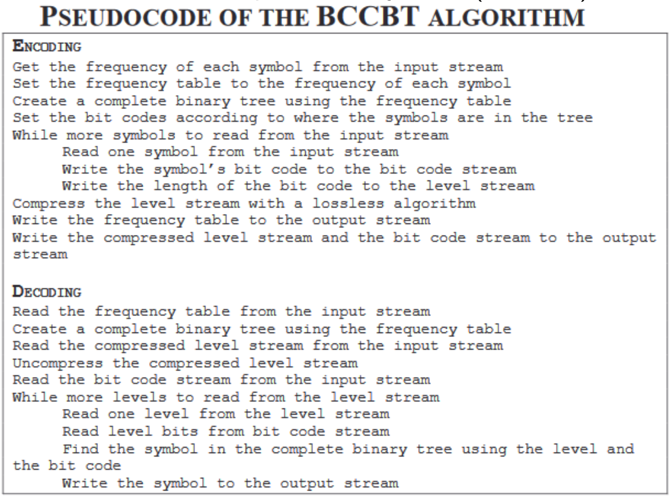


ABSTRACT

Compression algorithms can be used everywhere. For example, when you look at a DVD movie a lossy algorithm is used, both for picture and sound. If you want to do a backup of your data, you might be using a lossless algorithm. This thesis will explain how many of the more common lossless compression algorithms work. During the work of this thesis I also developed a new lossless compression algorithm. I compared this new algorithm to the more common algorithms by testing it on five different types of files. The result that I got was that the new algorithm was comparable to the other algorithms when comparing the compression ratio, and in some cases it also performed better than the others.

Keywords: compression algorithms, probability, dictionary, BCCBT

Bit Code Complete Binary Tree (BCCBT)



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
```

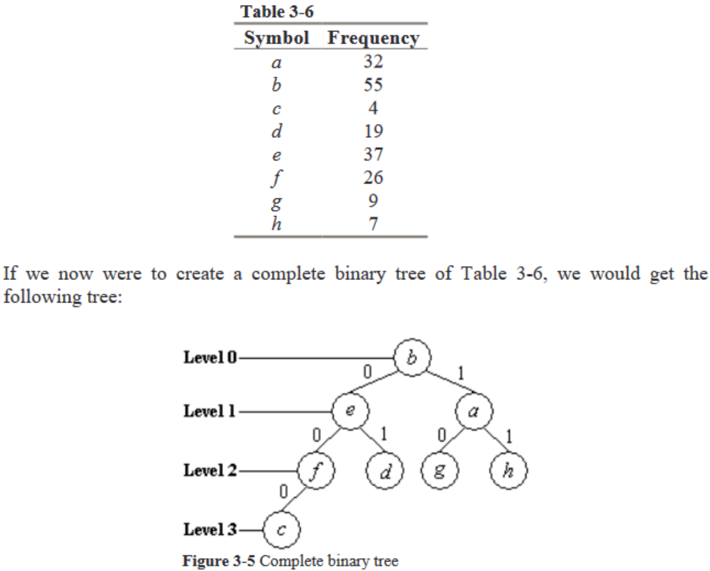


Table 3-6

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

If we now were to create a complete binary tree of Table 3-6, we would get the following tree:

Level 0
Level 1
Level 2
Level 3
b
0
1
e
a
0
1
0
1
f
d
g
h
0
c

Figure 3-5 Complete binary tree

Properties of Complete Binary Trees:

- All levels are completely full except possibly the lowest level.
- Filled from the top down and left to right. i.e. Tree leans left.
- The number of nodes at level n is 2^n $(n \in \mathbb{Z}_9)$.

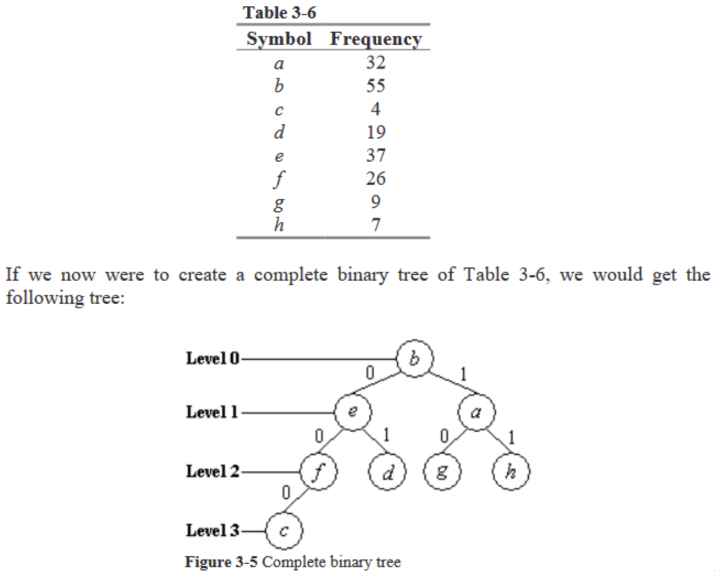


Table 3-6

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

If we now were to create a complete binary tree of Table 3-6, we would get the following tree:

Level 0
b
0
1
Level 1
e
a
0
1
0
1
Level 2
f
d
g
h
0
Level 3
c

Figure 3-5 Complete binary tree

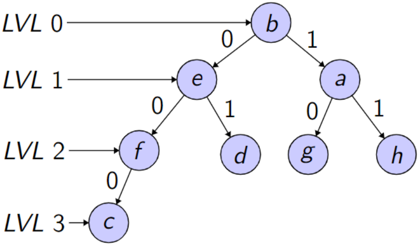
Step 1: Find the Frequency Table

Given an alphabet $\Sigma = \{a, b, c, d, e, f, g, h\}$ where each symbol has the following frequency

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

Step 2: Generate Complete Tree and Get Bit Codes

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



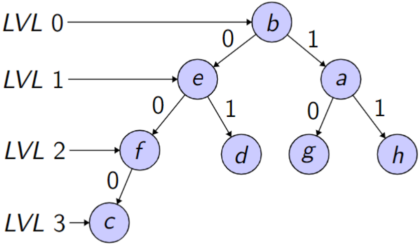
Wait, these bit codes are not uniquely decodable, how do we fix it?

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

Step 3: Making Uniquely Decodable Strings

The solution, append the symbols level to its bit code. The bit strings are now uniquely decodable.

Ex) Say we want to encode the string ‘feed’, encoding would look like this: [2]00[1]0[1]0[2]01, where the number between the [] specifies the level in the tree.



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

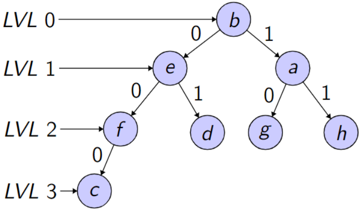
Step 4: Decoding an Encoded String

Encoded String: [1]0[2]01[2]10[1]0

We first look at [1]. This is telling us the symbol is in LVL 1. The next bit, 0, tells us how we should walk the tree, in this case to the left once. We arrive at symbol e. Iterating through this n times:

1. [1]0 → e
2. [2]01 → d
3. [2]10 → g
4. [1]0 → e

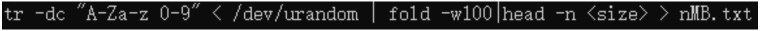
[1]0[2]01[2]10[1]0 ⇒ 'edge'



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

Our test suite is as follows:

- Construct randomly populated text files with the following command:

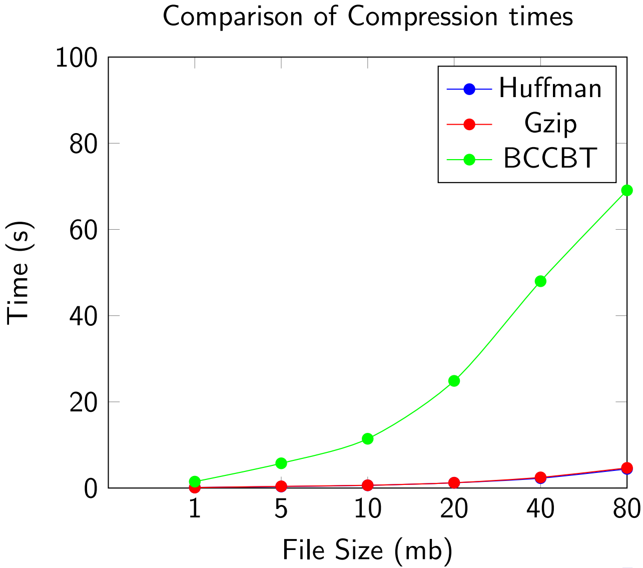


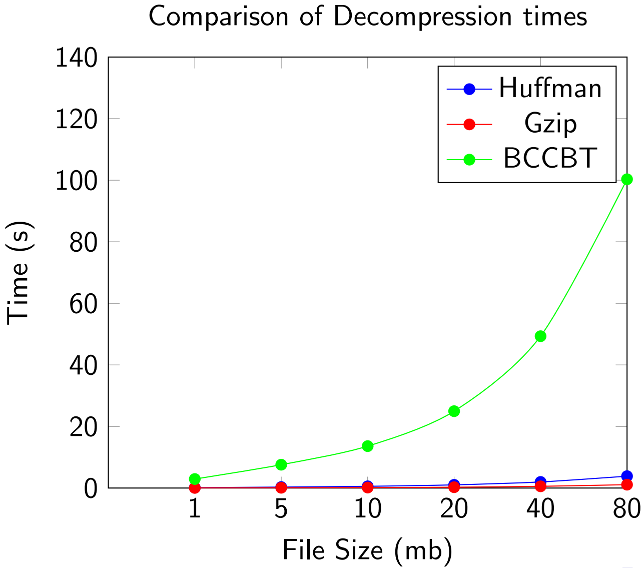
```
tr -dc ″A-Za-z 0-9″ < /dev/urandom | fold -w100 |head -n <size> > nMB.txt
```

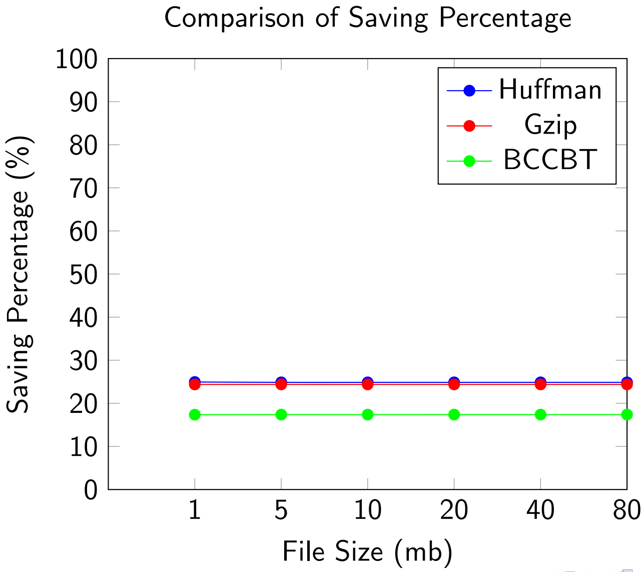
- Compare our implemetation of BCCBT with two open-source compression algorithms using 4 factors of comparison.

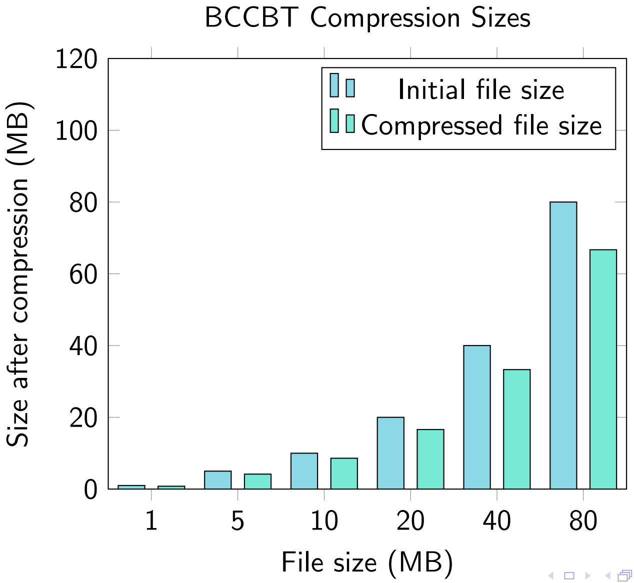
The 4 factors we will be using to analyze and compare these algorithms are as follows:

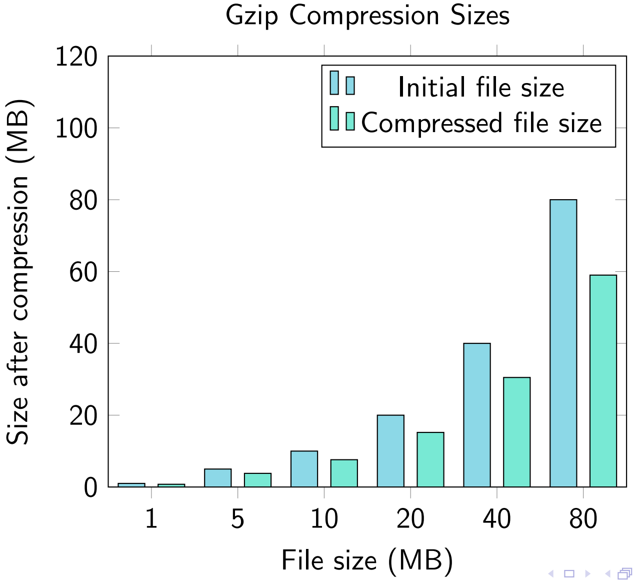
1. Compression Time
2. Decompression Time
3. Saving Percentage $= \frac{Original\ \ File\ \ Size\ -\ Compressed\ \ File\ \ Size}{Original\ \ File\ \ Size}$
4. Compression Ratio $= \frac{Compressed\ \ File\ \ Size}{Original\ \ File\ \ Size}$

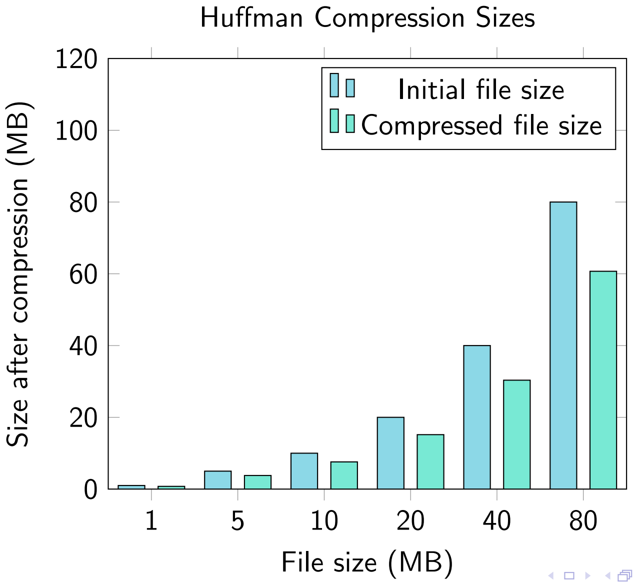

Comparison of Compression times
Time (s)
0
20
40
60
80
100
1
5
10
20
40
80
File Size (mb)
Huffman
Gzip
BCCBT


Comparison of Decompression times
Time (s)
0
20
40
60
80
100
120
140
File Size (mb)
1
5
10
20
40
80
Huffman
Gzip
BCCBT


Comparison of Saving Percentage
Saving Percentage (%)
100
90
80
70
60
50
40
30
20
10
0
1
5
10
20
40
80
File Size (mb)
Huffman
Gzip
BCCBT


BCCBT Compression Sizes
Initial file size
Compressed file size
Size after compression (MB)
0
20
40
60
80
100
120
1
5
10
20
40
80
File size (MB)


Gzip Compression Sizes
Initial file size
Compressed file size
Size after compression (MB)
0
20
40
60
80
100
120
1
5
10
20
40
80
File size (MB)


Huffman Compression Sizes
Initial file size
Compressed file size
Size after compression (MB)
0
20
40
60
80
100
120
1
5
10
20
40
80
File size (MB)

Both Huffman and GZip performed better than our implementation of the BCCBT algorithm in every test. Further summarizing our results, we found that:

- BCCBT fell behind heavily in Compression and Decompression times as files sizes increased, which is due to the lack of optimization done in our implementation of the BCCBT algorithm.
- BCCBT was slightly behind GZip and Huffman in Saving Percentages and Compression Ratio, however, the gap between them did not grow.

That's all, and thank you for listening! Any questions?