AANY: An S3-Like Database

Noah Markowitz - nbmarkow U56221888
Adwait Kulkami - adk1361 U25712111
Austin Jamias - ajamias U08182371
Yuqi Jin - yuqijin8 U55589765

12/14/2023
EC504, Fall 2023

Github.com/aany-crew/AANY

1 Abstract

AANY is a simple file system for compressing and storing plain text files. It operates using Huffman trees that are generated for each unique file in order to require less storage space than the original file size would require. The system comprises client, server, and database nodes designed to facilitate the efficient storage and management of plain text files through a simple database storage system. Clients interact with the server, utilizing various request types such as 'put,' and 'get,' enabling file compression, decompression, and data manipulation within the database. Leveraging Huffman coding, the server efficiently compresses plain text files upon user requests, storing the compressed data and associated metadata in the database nodes. Users can retrieve compressed files, prompting the server to perform decompression and return the original plain text content. This system architecture offers users the ability to self-manage their files, interact with the server for compression and decompression, and execute data operations within the database environment.

2 Overview

This report delves into the implementations of Huffman Coding Compression and Decompression. [4]The Huffman algorithm's strategy revolves around merging the two smallest weights iterative and replacing them with their sum until only one weight remains. The compression process begins by calculating the frequency of each character within a plain text file and constructing a frequency table to illustrate the occurrence of each character. Subsequently, a binary tree is constructed using the previously mentioned merging strategy, assigning '0' and '1' to the left and right branches, respectively. The creation of the Huffman tree entails a inorder traversal to each character detailed in this report. The final step involves replacing each character in the input data with its corresponding Huffman code, resulting in the generation of a compressed output by concatenating these codes.

For the decoding process, the procedure commences from the root of the Huffman tree and follows the path guided by the encoded bits ('0' for the left branch, '1' for the right). Upon reaching a leaf node, the corresponding character is retrieved, and the process restarts from the root. Reconstructing the original data entails following the decoding process to recreate the initial sequence of characters from the encoded binary data.

3 Instruction for running the code

- Step 1: In the project's home directory, type "make simple_server" to build the executable. It will be located in <project home directory>/bin/simple_server.[3]
- Step 2: Run the simple_server executable
- Step 3: Enter Filename: (need to copy paste the path of the example)
- Step 4: Enter Command(PUT/GET): (enter PUT). It Will show something in terminal
- Step 5: Enter Filename: (copy paste the same path before)
- Step 6: Enter Command: (enter GET). It Will show something in terminal

These steps represent that a user chooses a file to compress it and send it to the server. Then user wants to retrieve a file from the server by decompressing the compressed file.

Note: The more detailed instruction for running the code is uploaded to GitHub.

4 Algorithm Pipeline

In order to implement Huffman coding [7] a Huffman tree must first be constructed. A Huffman tree is a tree data where all nodes that represent characters or some meaningful value are leaf nodes while intermediate nodes are simply used to combine them. The resulting tree is such that the leaf nodes with the smallest weight have the greatest depth while the leaf nodes with the greatest weight are situated closer to the root. With this Huffman coding becomes particularly powerful for instances where one or two characters have greater frequency than all other characters combined as opposed to all characters having relatively the same frequency.

Building the Huffman tree is a relatively simple procedure that requires assigning each character a weight (based on its frequency in the instance) and a priority queue. All characters are added to the priority queue and are treated as nodes for a tree not yet built. The two smallest nodes are popped from the tree and then combined to create a parent node whose value is equivalent to the weight of its two children added. This new parent node is then added back into the priority queue. This sequence continues until only one node is left in the priority queue which becomes the root of the tree.

Once the Huffman tree is built then the Huffman codes, sequences of bits representing each character, can be generated. The Huffman code associated with each character (which is a leaf node in the tree) can be obtained by traversing the tree using a tree traversal algorithm and tracking the steps taken to get to each leaf node. If traversing to the left child node then the bit sequence is appended with a "1" and if traversing to the right child then the bit sequence is appended with a "0". The most frequently used characters will have the shortest sequence of bits while the least frequent characters will have the longest sequence of bits. The content of the file can then be replaced by the generated Huffman codes using a hash map while replaces each character with its newly associated bit sequence.

Pseudo Code 1 Huffman Compression

```
procedure HuffmanCoding($S, freq$)
    $n \rightarrow$ length($S$)
    $Q \rightarrow$ priority_queue()
    for $i \rightarrow 1$ to $n$ do
        $node \rightarrow$ new_node()
        $node.char \rightarrow S[i]$
        $node.freq \rightarrow freq[i]$
        $node.left \rightarrow$ NULL
        $node.right \rightarrow$ NULL
        $Q$.insert($node$)
    while $Q$.size() $> 1$ do
        $left \rightarrow Q$.extract_min()
        $right \rightarrow Q$.extract_min()
        $node \rightarrow$ new_node()
        $node.freq \rightarrow left.freq + right.freq$
        $node.left \rightarrow left$
        $node.right \rightarrow right$
        $Q$.insert($node$)
    $root \rightarrow Q$.extract_min()
    BuildCodes($root, ""$)

procedure BuildCodes($node, code$)
    if $node \neq$ NULL then
        if $node.left =$ NULL $\&$ $node.right =$ NULL then
            *Output the character and its code*
        BuildCodes($node.left, code + "0"$)
        BuildCodes($node.right, code + "1"$)
```

To reverse the procedure and convert Huffman codes back to their associated characters, also known as decompression, requires traversing the tree to a leaf node N times with N being the total number of characters present in the original file. Each bit is one at a time and if the bit is a "1" then the left child node is traversed to otherwise (if the bit is a "0") then the right child is traversed to. This continues until a leaf node is reached and the character associated with that leaf node is added back into that file being read or a new file. The algorithm then returns to the root node of the tree and continues reading bits one at a time and traversing the Huffman tree until all bits are read. The result is the decompressed original content.

Pseudo Code 2 Huffman Decompression

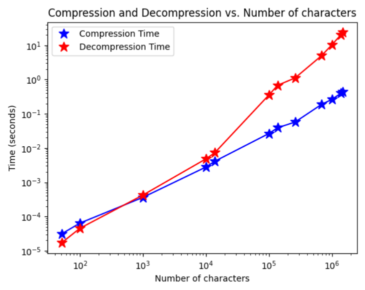
```
procedure HuffmanDecompress(compressedData, huffmanTree)
    $decompressedData \leftarrow$ empty string
    $currentNode \leftarrow$ root of huffmanTree
    for each bit in compressedData do
        if bit is $0$ then
            $currentNode \leftarrow$ left child of $currentNode$
        else
            $currentNode \leftarrow$ right child of $currentNode$
        if $currentNode$ is a leaf then
            Add $currentNode$.symbol to $decompressedData$
            $currentNode \leftarrow$ root of huffmanTree          ▷ Reset for next symbol
    return $decompressedData$
```

5 Results

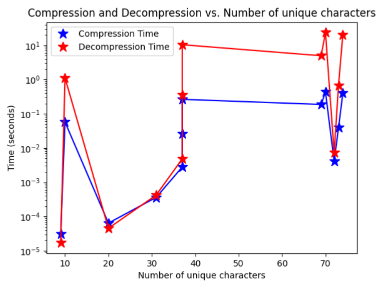
Sample results are generated using two different test scripts[3][2][1][5]. The first is "huffmantreenode_test.cpp" for printing the compiled Huffman tree for a file and checking the bits associated with each unique character. The second is "huffman_timing_tests.cpp" which is used to generate the output used in evaluating the speed of the execution. It generates time information in the console as well as a csv file that can be used for analysis. Analysis is done using the python script "analysis.py" to generate graphs that display the results. The python script generates graphs comparing time versus the parameters of the Huffman Tree.

5.1 Huffman Tree Results

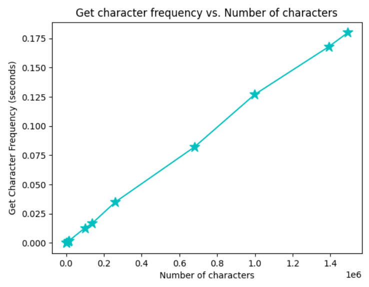
There are a few functions that require directly working with the Huffman tree that take notable execution time: Constructing the tree, traversing the entire tree (in the case of generating a map of Huffman code for each character) and traversing the tree from root node to one leaf node (in the case of matching a bit sequence to a character). Tree traversal is a particularly costly procedure that depends on the number of unique characters (corresponding to number of lead nodes) and the balance of the tree (corresponding to the average frequency of characters). The python script generates graphs comparing time versus the parameters of the Huffman Tree.



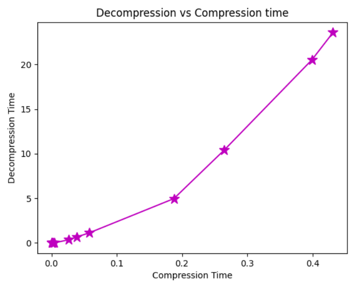
(a)



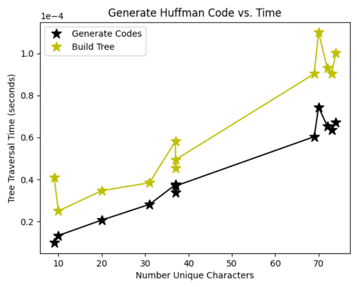
(b)



(c)



(d)



(e)

Figure 1: *(a)Time to compress and decompress text as a function of the number of characters in the file. (b)Time to compress and decompress as a function of the number of unique characters. (c)Time to obtain frequency of each unique character as a function of number of characters in the text file. (d)Decompression time as a function of compression time. (e)Time to generate Huffman codes (perform inorder traversal of tree) as a function of time.*

6 Discussion

From the results as seen in figure [1a], it is evident that compression demonstrates a linear relationship over time, whereas decompression exhibits an exponential trend. It is anticipated that compression operates linearly at O(N), while decompression functions at an expected rate of O(NlogN). Discrepancies between the expected and observed decompression rates may stem from various factors such as data loading into memory, overhead, or other intricacies within the code implementation.

The time to compress and decompress text, as depicted in figure [1b], shows no discernible relationship. This implies that number of unique characters is not the only factor in the time of these operations.

Both figures [1c] and [1d] serve as sanity checks. Figure [1c], displaying time to obtain the frequency of each unique character as function of the number of characters, reveals the anticipated linear relationship. Figure [1d], depicting decompression versus compression time, is also linear as expected.

Figure [1e], depicting the time taken to generate Huffman codes, reveals that the duration for building the tree surpasses the time taken for generating the codes. This difference in processing times may be due to the data size utilized for building the tree, which could be notably larger than the data employed for generating the codes. Such variance in data size often results in varying processing times.

7 Summary

This report explores the implementation and analysis of Huffman coding techniques for compression and decompression of lossless data. The study involved a comprehensive review of Huffman code trees, followed by the practical implementation of compression and decompression algorithms. Additionally, a user-friendly client-server interaction system was developed, facilitating file compression using a stream of '0s and 1s' and subsequent retrieval through decompression at the server end. Notably, our analysis, as depicted in the generated plots, revealed distinct time complexities for compression, observed as linear at O(n), and decompression, observed as exhibiting exponential growth at O(nlogn). An important finding is the substantial influence of the total number of characters, irrespective of file size, on the overall processing time. Furthermore, our study conducted crucial sanity checks to ensure the expected operational functionality, examining the time interval between compression and decompression, as well as the relationship between obtaining frequency and the number of characters. A notable observation was the potential impact of differing data sizes on the tree-building process, where significant variance in data size resulted in notable variations in processing times.

8 Future Work

Potential improvements in compression can be done by using the CRUSH algorithm as used by Ceph [6]. In addition a load balancing algorithm that tracks available storage at each node on the server. As with any software there is always the possibility for more features to be added but AANY focuses primarily on optimizing storage and compression.

Pseudo Code 3 Map Update and Send to Node Logic

```
Class Metadata:
    Attributes:
        map(mp): Map
        num_blocks: Integer
        num_bits: Integer (for the last block)

procedure MapUpdateLogic(RequestType, MetadataObject)
    if RequestType is GET then
        Return $\rightarrow$ Do not Update the map
    else if RequestType is PUT then
        Clear MetadataObject
        Add new MetadataObject
    else if RequestType is CLEAR then
        Clear MetadataObject

procedure SendToNodeLogic(BlockIDs, IPAddresses)
    for each BlockID in BlockIDs do
        for each IP in IPAddresses do
            if IP is available then
                Perform get/put/clear operation to Node

procedure PrintMap(Map)
    for each IP, BlockIDList in Map do
        Print IP address and list of Block IDs
```

References

[1] Datablist sample csv files. datablist. https://www.datablist.com/learn/csv/download-sample-csv-files.

[2] Github-organizations-100.csv. https://media.githubusercontent.com/media/datablist/sample-csv-files/main/files/organizations/organizations-100.csv.

[3] Lorem ipsum. https://www.lipsum.com/.

[4] Mit lecture-19, compression and huffmann coding. https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2012/388115265a456321c4a5d19dc9e05281_MIT6_046JS12_lec19.pdf.

[5] Statistics new zealand. (n.d.). csv files. https://www.stats.govt.nz/large-datasets/csv-files-for-download/.

[6] Crush map — ceph documentation. https://docs.ceph.com/en/latest/rados/operations/crush-map/, 2023. Accessed: 2023-12-14.

[7] Borko Furht, editor. *Huffman Coding*, pages 278–280. Springer US, Boston, MA, 2006.