Huffman Coding Assignment Report

Jazer Barclay

STUDENT ID: 20837308

CI583 – DATA STRUCTURES AND OPERATING SYSTEMS

6TH JANUARY 2022

Table of Contents

1. Introduction

This report describes and explains the time complexity of the encode and decode methods used in my Huffman coding assignment and how data structures serve a purpose in modern-day operating systems.

Huffman coding is a lossless compression algorithm that uses the frequency of characters or bytes to create the most compact bit representation of a data. It does this by constructing a frequency sorted binary tree known as a Huffman tree, substituting the characters with bit patterns thus compressing the file with zero data loss. Huffman is used in zip and image file compression such as GZIP, JPEG and PNG. It is also used in the compression of HTTP headers.

Computational complexity is how an algorithm's requirements grow as its data workload increases in the time and space domains. Big O notation represents this magnitude of growth as a function of its input length. It is the simplified analysis of an algorithm's efficiency. In this assignment, the worst-case scenario is considered for the time complexity domain.

2. Encode method complexity

2.1. Generating the frequency table

First, a table of all character frequencies is created. Since we must loop over each of the characters in the input string using a for loop, the worst-case scenario for this method would be on the order of O(n). As the length of the string 'n' grows, so does the time taken linearly.

2.2. Creating the Huffman tree

A Huffman tree is built next using the frequency table. Starting with the two lowest frequency characters of and working towards the largest until a single element remains in the queue, which holds the complete Huffman tree. To retrieve the characters in frequencies ordered smallest to largest, a priority queue is used. The priority queue sorts nodes by frequency labels, with functions to enqueue new elements and dequeue the smallest.

This queue initially implemented a sorted array where insertion would traverse the values until a greater value was encountered and further values shifted forward. Unfortunately, to enqueue and dequeue from a sorted array is an O(n) complexity since potentially all values would need shifting. This compounds with each character needing insertion and removal from the queue, resulting in $O(n^2)$ complexity.

A min-heap data structure now replaces the previous implementation to improve the performance of the enqueue and dequeue methods. A min-heap uses a complete binary search tree that raises the smallest values to the top. Insertion and removal require simply traversing the tree's depth, which is log-linear, or O(log(n)) compared its size. As a result, use of this new data structure has reduced complexity of generating the Huffman tree from $O(n^2)$ to O(n log(n)). The min-heap implementation was added to the new class 'MinHeap' and extends 'PQueue' allowing replacement in the 'treeFromFreqTable' method while maintaining JUnit test functionality without alteration.

2.3. Building the codes

We can recursively call the traverse method for left and right nodes on the Huffman tree and receive a map containing an array list of all characters and their corresponding list of boolean values which represent their codes. Like the min-heap, we only need to travel the tree's depth to get all characters and their matching codes. However the complexity remains O(n), where 'n' is the number of nodes, as we must traverse all nodes to obtain all codes in the tree.

2.4. Encoding the data

Encoding utilises all previously mentioned methods to generate the required codes based on the given input string. Then, using the encoded values of each character, encode loops through each character in the string and stores its corresponding code in a list of binary values as booleans. The method finally returns the map of character codes with the encoded data, which completes the Huffman coding.

Although the encode method only loops over each character, we consider the overall worst-case of methods used too. For the encode method, the 'treeFromFreqTable' has a complexity of O(n log(n)), making it the worst-performing based on its input size.

Thus, the encode method overall has an order of complexity of O(n log(n)), where 'n' is the length of the input string.

3. Decode method complexity

3.1. Building a Huffman tree from a code map

To rebuild the Huffman tree, we must loop through each code to construct the branch and leaf nodes. This is multiplied by the number of characters that need inserting into the tree. This can be seen in the code as the nested for loops.

In the average case where we have a somewhat more balanced Huffman tree, we would expect code lengths to be towards O(log(n)) and when in conjunction with each character, we have an average case of O(n log(n)).

However, in the worst-case, the longest code in a heavily unbalanced tree would be the total number of characters minus 1; or O(n-1) complexity. Multiply this by looping the total number of characters in the map and we reach a complexity of O(n*n-1), which simplifies to an order of $O(n^2)$ in the worst-case.

3.2. Decoding the data

Decoding takes a map of all characters with their codes and the encoded data as a list of booleans. The earlier method is used to construct a Huffman tree using the provided codes, which we can use to decode the parsed data. We can traverse the tree and resolve each character by looping through the boolean values, finally adding it to a string. Even though we loop over each Boolean in the encoded data giving us an O(n), our complexity is determined by the worst performing method or structure which is the 'treeFromCode' resulting in $O(n^2)$.

4. Functional compression and decompression implementation

To produce a functional compression and decompression tool that allows files to be compressed or decompressed, a main method would need implementing. This main method would take the file name and flags to compress or decompress the given file in the 'args' array.

It would also require storing the Huffman tree as binary within a file alongside the encoded data. Included within the Huffman class is a compress and decompress function created that emulates this. They can store the Huffman codes with the encoded data and decompress from the generated 'hf' file. A new JUnit test named 'testCompressDecompress' was created to run and test against a set of different text files which can be set.

5. Structures and algorithms in operating systems

Operating systems have finite resources limited by the hardware they use. Data structures allow the operating system to employ the best methods of managing the available resources at any given time. However, the system must also remain interactive and alert to user input. This priority for interactivity is where processor scheduling is essential.

5.1. Time-sharing scheduling and priority queues

Time-sharing systems uses a priority queue to organise when to prioritise first requests and, after their quantum, reduce their priority to become a lower priority background process.

In older traditional systems, a simple batch system (SBS) was implemented where each process was run to completion however would not allow for dynamic interaction. Instead, the scheduler's job was to pick the following process to be run. This SBS would be unfeasible for a modern system, and new scheduling strategies such as time-sharing are implemented.

Where multiple processes require access to processor time, their priority can determine how soon and how much time they have to use the processor. To determine priority and reduce longer tasks causing more critical and time-sensitive tasks, such as user input and interaction, a priority queue is implemented.

New tasks are enqueued with the highest priority at the beginning of the queue with a set amount of processor time referred to as quantum. Then, the task is dequeued and, as the task consumes its quantum, it is enqueued with a lower priority and its new quantum reduced. This way, important and short processes such as mouse input are completed immediately, whereas long computation tasks such as batch file transfers move further to the back of the queue. When there are few important tasks to complete, these lower priority processes complete.

5.2. The FAT file system and linked lists

The file allocation table, or FAT, file system is used by many devices such as cameras due to its simple design and resulting implementation. To track its contents, it splits its available space into sectors. These sectors are referenced and linked together using a linked list which indicates where the next sector in a sequence can be found.

5.3. Directory listings and hash functions

Modern file systems have no limit on how many files can exist within a directory. However, with a large number of files, searching would take O(n) time when iterating through each file in a linear list.

A hash function maps data of any size to a fixed-size value. An example of this would be the 'hashCode' in java which converts a string into a 32 bit integer using a simple algorithm. If all files were converted using a hash function and their locations stored, we could easily search for a file using the same method and resolve its location is memory.

The complexity of search for a file regardless of its location reduces O(1) where the searched location can be converted to its hash value and looked up directly in the hash table. Using a hash table does come with the potential for collisions where indexes can be used via inodes.