File Compression (CS214 // Assignment 2)

Contributors: Seth Karten & Yash Shah

Overview

In this assignment, we were asked to design a program which uses Huffman Codes to compress and decompress files. The assignment's primary objective served to help us learn and use the file system interface to read files and directories. Using the `readdir()` and `opendir()` functions, we were able to recursively explore the entire directory tree and eventually use `open()` and `read()` functions to read and analyze the contents of each file.

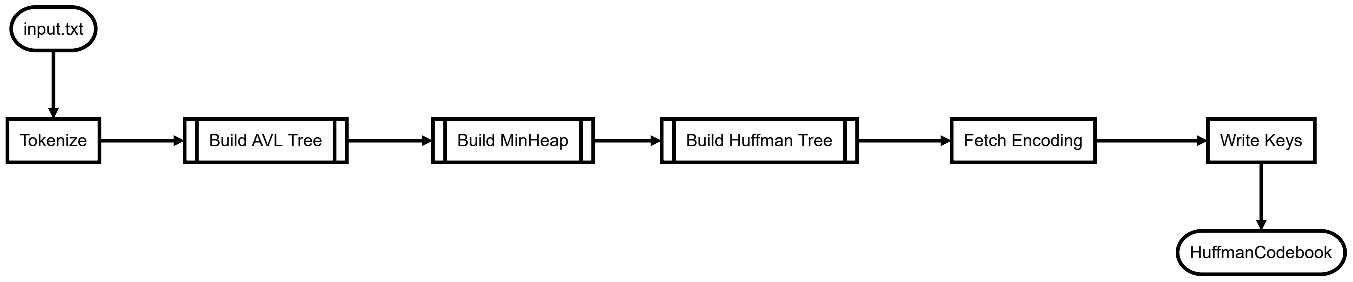
This assignment consists of three major parts: (1) building the codebook using Huffman codes on tokens read from files, (2) compressing the files, and (3) decompressing the compressed files.

Building the Codebook

When building the codebook, we used three primary data structures in our implementation to optimize the process:

- AVL Tree
- MinHeap
- Huffman Tree

The `build_codebook` routine performs the following operations:



We start the build proces by using the AVL tree to store tokens. We use the AVL tree for the `O(log(n))` time complexity it provides for insertion and lookup operations. This reduces our overhead as we can quickly append items from a file buffer without having repeated elements, while being able to count the frequency of each token occuring.

Next, we build a MinHeap by traversing through this AVL tree of elements with the priority factor as the frequency of occurence of each token. The MinHeap's `O(log(n))` time complexity allows us to quickly sort the tokens to prepare them for encoding using Huffman Codes as it serves as an ordered list of tokens.

Finally, we build the Huffman Tree as it's structure allows us to quickly determine each token's encoding string. As we traverse through the entire tree, we keep track of left/right moves and build up the encoding string until we reach a leaf node. This operation gives us an `O(n)` time complexity for encoding keys as we only ever visit each token once.

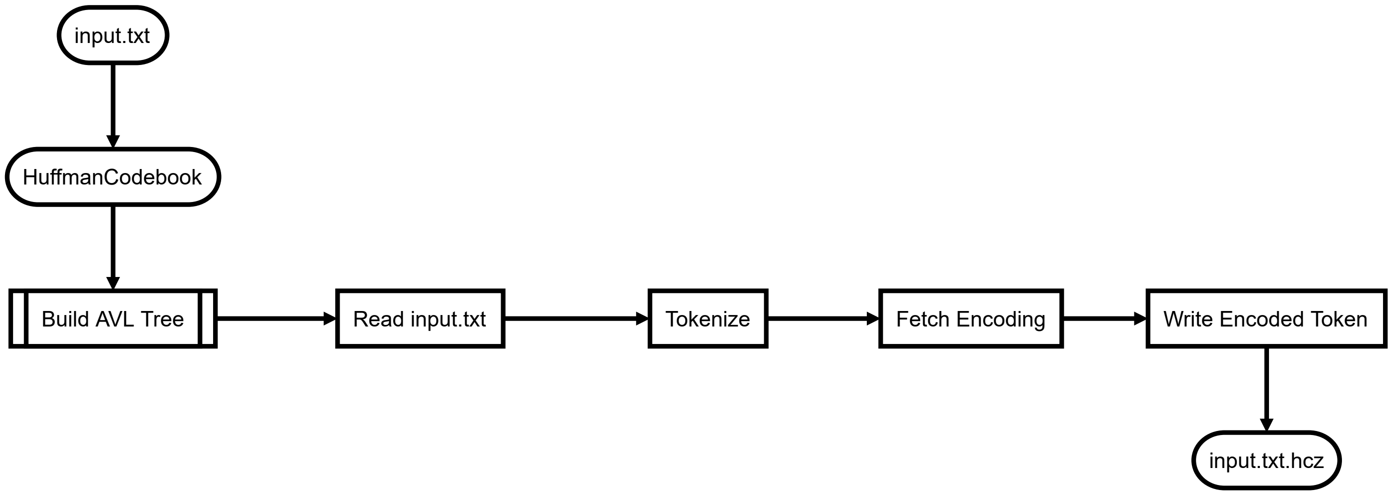
For the overall time complexity, we understand that we need `O(n)` time to insert all `n` tokens into the AVL tree. Insertion of each token into either the AVL tree or MinHeap takes `O(logn)` time. Therefore, without loss of generality, we get `O(n*log(n))` time complexity when running this process.

Regarding space complexity for this process, we have to account for the worst case on each of the three data structures we use. In our worst case, each of the `n` characters in our file could be a separate token, and thus, we could potentially have `n` nodes in our structures. Due to the nature of all three data structures, we can deduce that the space complexity for building the codebook is `O(n)`.

Compressing Files

When compressing files, we used only the AVL Tree to optimze the process. We leveraged the AVL tree's `O(log(n))` time complexity for lookups when searching the encoding string for each token found in the file.

The flowchart outlines the compression process:



We start by building an AVL tree from the `HuffmanCodebook` file which stores all available tokens and their respective encodings. After that, we iterate through the input file to fetch all tokens to compress, and lookup their respective encodings for each token, and write them, in order, in the compressed file.

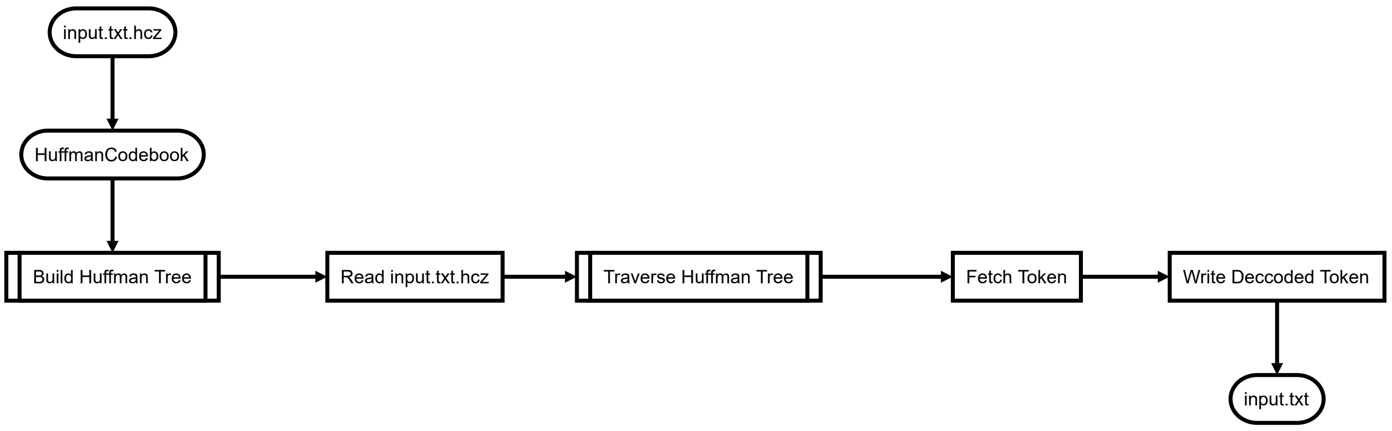
We found the overall time complexity for this process to be dependent on the number of tokens in the file, which could be up to `n` characters. We use the AVL tree, once again, to store these tokens, and therefore, our bottlenecks reside in the lookup time, as well as insertion time for the AVL tree, giving us a big O time complexity for this process of `O(n*log(n))`.

Regarding space complexity, we will be storing all tokens in memory to iterate through them for encoding purposes. Therefore, our big O space complexity for this process will be `O(n)`.

Decompressing Files

When decompressing fies, we employed the Huffman Tree to optimize the process. We leveraged the fact each encoding string reflected left/right moves for the binary tree when we searched for the token that was originally stored.

The flowchart outlines the decompression process:



We begin the decompressing process by reading the `HuffmanCodebook` file and loading its contents into a Huffman Tree. As previously stated, this data structure allows us to treat the encoding string as instructions for traversing the binary tree until we reach a token. This allows us to achieve a time complexity of `O(n)`, where `n` is the number of bits in the encoded string. Once we decode a token, can write it to the original file.

For decompression, we will be loading the tokens stored in our Huffman Codebook file, and simply building the Huffman Tree structure from it to decode all of our tokens. Because the HuffmanCodebook file has keys stored in pre-sorted order, our most time intensive operations include loading the tokens, and searching through them. Loading them will take `O(n)` time, and decoding takes `O(log(n))` time. Therefore, the big O time complexity for this process is `O(n*log(n))`.

Regarding space complexity, we will be storing all tokens in memory so that we can decode all possible tokens which were encoded in the compressed file. Therefore, our big O space complexity for this process will be `O(n)`

Recursive Operations

Another significant component of this assignment required traversing through directories and being able to manipulate files and subdirectories that exist within them. Hence, a recursive approach was applied to fetch all eligible files for the operation and appended to a LinkedList. We used a Linked List because of it's simplicity in storage, as well as it's ability to easily traverse the list at the end.

We defined the following struct to store each item in the Linked List:

```
typedef struct _FileNode {
        char * file_path;
        struct _FileNode * next;
} FileNode;
```

We ensured that we modularized our program for each operation, and this allowed us to iterate through this linked list of files and call compress/decompress on each of these files. As a result, whenever our program receives a recursive file, it fetches files, and iterates through the files and calls the operation requested on that file.

However, during the build process, rather than invoking the entire process of building the codebook, we split up the process into two:

- Building the AVL Tree of tokens
- Building the Huffman Tree to fetch encoding & writing the keys to a file

By doing this, we optimized our program by ensuring that we accounted for all tokens that exist in all files within that directory, and only performing the encoding process once.

For safety, we ensured that the input received by our program is a valid directory path using `stat`. If it is not found to be a directory, we simply inform the user that the flag requires a directory input and cleanly exits.

Sanitizing Control Characters

In order to ensure reliable compression and decompression of files, we had to ensure that our program would not be confused by control characters present in the input files. We realized that we couldn't simply use any obscure ASCII character for this as that in itself would serve as an obvious point of failure for our program, and we would essentially be playing the odds to see if it shows up in a test case.

As a result, we employed a method where we would convert all control characters to their numerical ASCII equivalent and encode that using Huffman Codes as a token.

For example, since the newline character (`\n`) holds the ASCII value of `010`, our sanitize function would convert the single character to the token string `!010`.

We use the exclamation point in the beginning of our encoded string so that we know to convert the numbers back to the character on decompress. As stated before, we're confident that this doesn't present a point of failure as we would eventually convert the exclamation point in the following way:

```
ASCII value of ! = 033
```

Therefore, we would store the encoded version as:

```
'!' => "!033"
```

Using this method enables us to ensure that all three of the required functions of this program (build, compress, decompress) will behave as expected on all types of inputs.