File Compressor

AUTHORS

Rakshay Kanthadai

Krishna Prajapati

NAME

Huffman.c - C file with main()

fileCompressor - compiled binary executable

DESCRIPTION

The fileCompressor program is a binary executable that takes in an operation flag and possibly a recursive flag. If the recursive flag is used, then the path argument must be a path to a directory, and fileCompressor will perform an operation on all files and subdirectories within the path given. Similarly, if it is not used, the path argument must be a file, and the operation specified by the other flag. FileCompressor will perform the operation on the file whose path is specified. The following are the possible operations of fileCompressor.

<BUILD> - FileCompressor will build the Huffman codebook for the given argument. It will count the frequencies of every token that can be read, and then it will output a file called "HuffmanCodebook". This file is read-only by default, and it contains the Huffman code for the tokens read. This file is created in the same directory as the binary executable.

<COMPRESS> - FileCompressor will use a Huffman codebook and attempt to compress the given argument into its respective Huffman code. On success, if the input path was <path>.<extension>, then the output of compress will be <path>.<extension>.hcz.

<DECOMPRESS>-FileCompressor will use a Huffman codebook to attempt to turn a .hcz file into the original file. On success, if the input path was <path>.hcz, then the output of compress will be <path>.

<RECURSIVE> - FileCompressor will take in a directory path instead of a file path argument. This operation must be combined with one of the 3 above. FileCompressor will then perform that operation on every file and every file within the subdirectories. If building, there will be only 1 output, which is "HuffmanCodebook" in the binary's directory. If compressing or decompressing, the files create will be placed in the same directory as it is read from.

USE

./fileCompressor <flag(s)> <path> <HuffmanCodebook>

<flag> - a flag that indicates which operation to do. There must be one and only one of the following flags: -b, -c, -d. Optionally, you can include a -R flag for recursive mode. The flags must be the first argument

<path> - an input path to a file or directory that the operation should be performed on. If recursive mode is used, this path must be to a directory path. If recursive mode is not used, this path must be a file path. Note that if the operation is build or compress, then .hcz files are ignored in recursive mode, and give an error in non-recursive mode. Similarly, decompress will also only look at .hcz files in recursive mode, and give an error in non-recursive mode.

<HuffmanCodebook> - this is a path to HuffmanCodebook. If the build flag is used, then do not include this last argument. If the compress or decompress flag is used, then giving a file path to a codebook is required. Note that the codebook must be named “HuffmanCodebook” exactly, though it can be in any directory.

ERRORS

Fatal Errors: These are errors that will kill the program.

1. Not enough arguments- program will terminate and print a Fatal Error statement if the number of arguments is incorrect. This includes the case where not enough arguments are used for a certain flag

2. Invalid flag use - program will terminate and print a Fatal Error statement if the instructions for inputting a flag above is not followed.

3. Cannot open file/directory - program will terminate and print a Fatal Error statement if the file/directory cannot be opened/found. Additionally, if attempting to build or compress a .hcz file, or decompress a file that is not .hcz, then there will be a fatal error in non-recursive mode.

4. Not enough space - program will terminate and print a Fatal Error statement if there is not enough memory space on the heap to allocate the space needed to store data.

5. Read error - program will terminate and print a Fatal Error statement if there is a problem reading the file.

6. Quicksort memory error - program will terminate and print a Fatal Error statement if there is not enough memory to allocate in the quicksort method.

7. Invalid Huffman code Bit String- program will terminate if it reads a HuffmanCodebook and reads a bitstring with a character other than 0 or 1. It will print this Fatal Error

8. Missing Huffman Code- program will print a Fatal Error and terminate if the Huffman code for a token is missing in “HuffmanCodebook”, but is present in the file that will be compressed.

9. Invalid .hcz Tokens- program will print a Fatal Error and terminate if the compressed .hcz contains characters other than 0 and 1.

10. Invalid Encryption- program will print a Fatal Error if a HuffmanCodebook cannot be used to decompress a file. It is detected if the remaining bit string at the end of the .hcz cannot be encoded into anything in the HuffmanCodebook.

Warnings: These are clarifications for the user in cases that seem incorrect, but the program can still run.

1. Empty file/directory - program will print out a warning if it recognizes that a file is empty or if a directory contains empty files, empty directories, or is empty itself. An empty codebook will be written with the default escape sequence.

2. No (de)compressible files – this warning only appears in recursive mode. If in compression mode and there are no files in the directories that can be compressed (no openable files), or if in decompression mode and there are no files in the directories that can be decompressed (no .hcz), then this warning will be printed.

RETURN VALUE

Program will output the proper files based on the operation. If performing the build operation, the program will output a "HuffmanCodebook" file in the binary file's directory regardless of whether recursion mode was used.

If performing the compress operation non-recursively, the program will output a .hcz in the same directory as the given file. If it is done recursively, then each compressible file will get its own .hcz file in their respective directories.

If performing the decompress operation non-recursively, the program will output the original file in the same directory as the compressed file. If it is done recursively, then each decompressible file will have the original file outputted in each of the decompressible file's own directories.

The program will not return anything on error.

TIME & SPACE COMPLEXITY:

Build uses an AVL tree to read and add all of the tokens. Reading will take O(N) time where N is the number of bytes. Adding each token into the AVL takes O(n * log(n)) time where n is the number of tokens. We then turn this into a min heap, which takes O(n log(n)) time as well. We then perform the Huffman algorithm in O(n log(n)) time due to the min heap. We then traverse the Huffman tree in order and write it to the output file. Traversing takes O(n) time since we are using recursion, and writing takes O(M) time where M is number of inputted bytes per token plus the length of the bit strings. Totaling all of these operations results in an O(n* log(n) + N + M) running time.

As for the space complexity, there are n Nodes that total will contain N bytes of information in the program after reading the input file. When building the min heap, an array of size n is created, with n new tree nodes, as well as a repeated N bytes of information. Thus, the min heap takes up the same amount of space as the AVL tree. Since performing the Huffman algorithm and writing the output used already stored memory, the total space complexity of build is O(N+n), where N is bytes and n is the number of tokens.

Compressing reads the input file in O(N) time, while reading the HuffmanCodebook takes O(M) time. Creating an AVL tree that contains the m tokens will take O(m * log(m)) time. Checking each read token against the AVL tree will take O(n * log(m)) time. Writing the output, which is the compressed file, will take O(x) time, where x is the sum of number of bytes of the encoded tokens. Therefore, compression will take a total of O(N + M + x + (n+m) * log(m)) time.

As for the space complexity, the AVL tree will contain n tokens that contain N bytes of token information total and at worst case m*c bytes of bit string information, where c is some constant. This AVL tree is traversed as the compressible file is read in 200 byte buffers.

Therefore, reading the compressible file and writing the compressed file take up O(1) space. Therefore, the total space complexity is O(N+n+m).

Decompressing is the exact same algorithm but backwards, so decompression will take O(N + M + x + (n+m) * log(m)) time as well. Additionally, the space complexity will also be O(N + n+ m).

Recursion does all of these operations in the same running time and space complexity, but it will be multiplied by F where F is the number of files.