Assignment 1

Karan Amin
Saavi Dhingra

Name:

fileCompressor

Description:

FileCompressor is a basic compression program that implements the popular Huffman Coding algorithm to build a codebook or compress and decompress files in a lossless manner. The algorithm reads a file or a path as input from the command line, tokenizes the document's and creates a Huffman Table based on the rate of occurrence of each token. Using Huffman Code will us a lossless dynamic-length prefix code with zero ambiguity. Given a certain flag as input the program will either create a ./HuffmanCodebook in the current working directory, compress a file given and output the compressed file with the same name with a ".hcz" extension, or decompress a ".hcz" file given as produce the original decompressed file without the ".hcz" extension. It must be noted that building a codebook MUST precede compressing and decompressing files.

Command Line Interface

```
./fileCompressor <flag> <file or path> <codebook path>
```

Flags:

-b: This flag will build a ONE codebook with the name "HuffmanCodebook" in the current working directory. The codebook will be based on the file given or all the files if a directory path is given. If paired with a -R it will be based on every file in the directory given and every subdirectory.

-c: This flag will compress the file given or all the files if given a directory as the path. If paired with -R it will recursively descend down the directory given and even subdirectory and compress every file in them. The compressed file will be given the same name with a ".hcz" extension and will be placed in the same directory as its original.

-d: This flag will decompress the file given or all the files if given a directory as the path. If paired with -R it will recursively descend down the directory given and even subdirectory and decompress every file in them. It will only decompress files with a ".hcz" file as its extension.

-R: This flag is to indicate that the program should recursively descend down the path given and apply one of the flags above to every file in the given directory and every subdirectory. In order to use this flag the path given MUST be a directory path and not a file path.

Examples:

```
./fileCompressor -R -c ./testdirectory/ ./HuffmanCodebook
```

The command above will recursive descend down the directory tree starting in the testdirectory directory and compresses every file producing a ".hcz" compressed file with the same name as the original.

```
./fileCompressor -b ./firstdirectory/random.txt
```

The above command will find the random.txt and build a codebook based on the tokens in the file. Note that when using the -b flag, the HuffmanCodebook file path is not needed because the codebook is what we are creating. The codebook will be named "HuffmanCodebook" and will be in the current working directory.

Design and Implementation:

The overall flow of the program depends on the flag given. The overall time complexity will be drastically reduced by using smart data structures like Hash Tables. When building the codebook the time complexity depends on if we are recursively descending down each directory and opening every file or if it is only for one file. For this we will assume we are dealing with one file. The program starts by running a loop through each command line argument and sets appropriate boolean flags based on which flags were given as input. Based on which flags were set either buildCodebook(), compressFile(), or decompressFile() is called. The filepath, a bool for if -R was set, and the Huffman Codebook filepath is given as parameters. In each of the three functions there are two choices, recursive or non recursive.

If the recursion flag is set, then in each of the tree functions a call to recursiveBehavior is made where a function pointer to recursiveBuildCodebook(), recursiveCompress(), recursiveDecompress(0 is passed depending on which function was called in the beginning. Each of these functions implements the corresponding algorithm described below.

In buildCodebook is it is non recursive then another function is called which opens the filepath given. The function reads from the file byte by byte storing it in a string, and when it hits a control character or a space, it makes the string a token and inserts it into the Hash Table in O(1) time. For the hashing function we implemented a polynomial rolling hash function for strings which is known to be optimal. We used open addressing with linear probing and an initial size of

500 with a load factor threshold of 0.75. Once past the threshold the size of multiplied by itself 500 x 500 = 250 000. This type of hash table helps lower collisions while not wasting space so that we can keep the O(1) time complexity for inserts of tokens. If a token is already in the hash table it increments its occurrence counter by 1. At worse case all tokens are unique and inserting all n tokens will take O(n) time. Once all the tokens are in the hash table, another function is called to traverse through the hash table and insert each string-occurrence pair into the min heap. The occurrence value is what we will base the min heap ordering with. Traversing the entire hash table will be linear to the size of the hash table. If we assume the hash table size is close to the number of tokens then the time complexity of traveling the hash table is O(n). We have to note that inserting a node into the min heap uses the siftUp process which is done in O(logn) time if there are n nodes in the min heap. Since we do that n times for each token in the hash table the overall complexity is O(nlogn). The Space complexity up until now has stayed at O(n) since we assumed we will be using most of the space in the hash table. These assumptions are based on fact that our load factors 0.75 which means if there are n tokens in the table the overall size of the hash table is n / 0.75, which reduces to O(n) space complexity. Once we move all the nodes into the min heap we have no use for the hash table so we can dealloc the space for it. The Huffman Algorithm takes the minimum off the min heap in O(log n) time worst case because it deletes the root in constant time, but then it replaces it with the last node and then does the sift down process to keep the heap property alive. This sift down at worse is done in O(logn). In the algorithm we delete two nodes and replace it with one node. If there are n nodes in the heap, after the first iteration there will be n-1 nodes, then n-2, until there is 1 node. Each iteration is done in worse case O(logn) time for a total complexity of O(nlogn). Once there is only one node left the algorithm is finished. The only job left is to display the result to the ./HuffmanCodebook. For this we have a binary tree left over from the algorithm. Using a post-order traversal going left right root, everything we reach a leaf, we can track the path writing 0 whenever we go left and 1 whoever we go right and then writing the actual token stored in the leaf node. Any traversal of a binary tree is done in O(k) if there are k nodes in the tree, since we have to visit each node to reach all n leaves. Note that n < k. The most time consuming part of the algorithm is moving all the nodes from the hash table into the min heap in O(nlogn) time. The reason we don't just insert into the min heap directory bypassing the hash table is about a document will have multiple repeat tokens. Searching and updating the occurrence of a token in a hash table is done in constant time vs linear time traversing a min heap to find and update a token's occurrence. The many constant lookups on the hash table makes up for the transfer of data between the hash table and min heap. Since this is the complexity with the highest growth in our algorithm we can say our algorithm runs in O(nlogn) for building a codebook.

If we are compressing a file, we first open the file and tokenize it. Using the same method as above we read byte by byte and use control characters and spaces as delimiters. Instead of inserting it into a hash table, we instead use a linked list. The reason for this is that a linked list preserves token ordering on each token comes before and after each other. In the linked list node we have a field to store the bitcode corresponding to the token. Adding to a linked list is simple and easy and is done in O(1) adding to the end of the list. We can add to the end of the list by have a pointer that stays at the end every iteration. Adding all n tokens is similar to above and is done in O(n) time. The next thing we do is read the codebook. Since the codebook is formatted perfectly reading from it should be a breeze. We store the escape character on the first line in a character, and move to line 2. Until we see a \t we add the character a bit string, once we see a \t we add all subsequent characters to a token strings until we see a newline. Once we have the token-bitcode pair we insert it into the hash table in O(1) time. If there are n tokens the complexity will be O(n). Once all the tokens from the codebook are in the hash table we go back to the linked list we made easier and start from the head. Each node corresponds to a token-bit string pair in the hast table and we can use the power of hash tables to look up in constant time. This is why we used a hash table here instead of storing the tokens from the codebook in a list, we save run time using a hash table. Since there are n nodes in the linked list, with each node doing a constant lookup to get its corresponding bit string, the overall complexity is O(n) again. Once the entire linked list is traversed and updated with its corresponding bit string, all we have to do is open a .hcz with the same name and start at the head of the list again and write the corresponding bit string to the file in one long stream. The most time consuming part is going through the linked list and doing constant lookups in the hash table. Since this is the most consuming part we can say running the compressing part of fileCompressor is done in O(n) time.

Finally we have decompress. It will very similar to compression but opposite. We start by reading in the codebook with the same exact method as above for compression. Next we have to create a Huffman tree in order to decipher the .hcz files. We can do a traversal of the hash table in O(n) time with the explanation given above. For each entry in the hash table we take its bit string can update the Huffman tree structure so that it reflects the bitstring. In short if the bit string for the token “random” is “011” then we make sure the tree has a left then right then right path which leads to a leaf where we can store the word “random”. Calculating the time complexity of this operation is a little tricky since bit strings vary greatly between each token. If on average each bit string is k characters long, ten it takes k operations to create a tree with the same structure, so we can say it is done in O(k). Since bit strings are suppose to relatively small in size compared to n we can say its insignificant and that creating the tree structure is done in constant time since k << n (Much much smaller than n, which is the number of tokens). Once the tree is built, we can start reading the .hcz file character by character. We have a pointer at the root of the Huffman tree, every time we see a 0 we go left, every time we see a 1 we go right. When we hit a left there is nowhere to go so we can print out the word stored in that node to a result string. After we concatenate the word to a result token we reset the pointer to point back at the root node of the Huffman tree and read the next byte. If there are n leaf nodes in the Huffman tree, and each leaf node is relatively on the same level for simplicity, getting to each leaf node will take relatively O(log n). Since there are supposedly n tokens, worse case all of them are unique and we have to to traverse the tree n times to get to all n leaf nodes for a time complexity of O(nlongn). Once the entire bit string is read, the result string would be populated with the original document, all we have to do is use the write command to write it to a file without the .hcz extension. For the space complexity we populated the entire hash table with all n tokens which we then transferred over to the Huffman tree structure. We realloced the hash table immediately afterwards and the Huffman tree structure has n tokens which gives a space complexity of O(n) since all the non leaf nodes are empty and basically placeholders. The most time consuming part of this algorithm is reading the bit string and traversing the Huffman tree to find the leaf, so we can say that our decompress algorithm behaves most like it and has a time complexity of O(nlogn).