CS3302 P1 - Adaptive Huffman Coding

Ben Hope - 150000845

1 Introduction

The objective of this practical was to implement Adaptive Huffman Encoding and Decoding. I spent some time researching Adaptive Huffman coding and my main resources were Section 3.4 of "Introduction to Data Compression 3rd Edition" by Khalid Sayood and also an article on Adaptive Huffman Coding by Jonathan Low.

2 Extension : Description of Algorithms

There are two main implementations of the Adaptive Huffman algorithm which will be described below:

2.1 FGK - Faller, Gallagher and Knuth

- Start with a root node.
- If the character has not been seen before
  - Insert new node into end of tree.
  - Increment node frequency.
- If character has been seen before, then go down the tree to find the node for the character and increment the weight of that node.
- If the weight of this node is higher than the weight of its sibling to the right, then swap the two nodes.
- While the current node is not the root node

- Node = parent node of current node
- If the current node has a higher weight than its sibling, then swap it with its sibling node.

2.2 Vitter

- Start with a root node.
- Create an NYT (not yet transmitted) node and place it at the root of the tree. This should always have 0 weight and the lowest ID number.
- Read in character.
  - If it has not been seen before, then :
    - Make a new parent node at the current position of the NYT node.
    - Set its right child to a new node of weight 1 and value of the new character, set the left child as the NYT node, with frequency 0 and no value.
  - If it has been seen before, then go to the node which represents it.
- If there are any nodes with the same weight as it but have a higher ID, then swap the node with the node with the highest ID but same weight in the tree. Swap ID numbers with it too.
- While the current node is not the root node:
  - Set the current node as the parent of the current node.
  - Increment the weight of the current node.
  - If there are any nodes that have the same weight in the group but have a higher ID number, then swap the node in the tree with the node with the highest ID number but in the same weight groups.

2.3 Encoding and Decoding

When encoding, if it is a character that has not been transmitted yet, first encode the path to the NYT node so that the decoder knows that it is about to see the representation of a new character. After transmitting the NYT node, transmit a pre-agreed representation of the new node - for example, a Unicode representation of the new character.

If it is a character that has been seen before, we transmit the code for the character by creating a String of bits, by taking the route to the node. We add a '1' to the String for every right child that is navigated to, and a '0' for every left child that is navigated to. If it reaches the NYT node, it means that a new character is about to be read in, so the decoder needs to read in the next bits up until enough have been read to determine what the new character is. If it navigates to another leaf node instead then we add the character that the node represents to the decompressed file.

The decoder constructs a tree at the same time as the encoder so that they end up with similar trees. The decoder reads in each bit so that it can navigate the tree. If the bit is a '0', then navigate to the left child of the current node. If it is a '1', then navigate to the left child.

2.4 Comparing the Two Algorithms

The FGK algorithm does not check the entire tree in the same way that Vitter does - all it checks are parent nodes and right siblings to see if they need to swap. This makes the tree traversal faster, since we only take a direct route from the node to the root and make swaps on that route. However, the additional checks mean that the Vitter tree always has a minimum length which is beneficial for worst case time complexity.

3 Design

My implementation is based on the Vitter algorithm. My tree class does not just use the tree. There are two additional hashmaps that it uses. The first is a hashmap that maps a character to the corresponding node that it represents, which means finding a node takes O(1) time, since it directly points to the node. With a pointer to the NYT node, this also means that creating a new node takes O(1) time since all we have to do it go to the NYT node and create a new parent node and new node with the new character from there. The alternative to this would be to search the entire tree for the NYT node or the node which we want to look for, but the only way the tree is sorted is by the weight, not the character, which makes searching difficult.

The other additional structure is another hashmap, which maps an integer representing the weight of a node to a linked list of all the nodes that have that exact weight. This is for when we need to determine when incrementing the frequency of a node if there are any other nodes that have the same weight but a higher ID. Since nodes are added to the beginning of the list as they get their new frequency, this means that the ones of the lowest ID are put at the beginning of the list and the highest ID at the end. Therefore, the list is ordered by ID number. So to check if there are any nodes in the tree that have a higher ID but same weight, we just have to check to see if the node we are checking has the same reference as the node at the end of the list. If it does, then the node at the end of the list takes the position of the node we are checking, both in list position, tree position and ID number, and we move the node that we are checking to the beginning of the list that contains all the nodes with its new frequency. However, if the node that we are checking is at the end of the list, it means that it is the node with the highest ID in that weight group and so therefore we just need to remove it from the end of the list and add it to the beginning of the next list. We have to check however that the node at the end of the weight list is not the parent node before we try and swap with it, since the parent node cannot have a lower ID number than its children and it can't swap with its child node in the tree. Neither the root node nor the NYT node are added to weight groups since they should not be compared or swapped.

3.1 Tree Operations

Both the encoder and the decoder model individual Huffman trees. When the tree reads in a character, it first checks the hashmap of all the nodes to see if it exists or not by seeing if the key exists in the map or not. If it does exist, then it means the node has been seen before, if it does not exist then it means it has not been seen.

If the node has not been seen before, then we go to the NYT node using the reference to the NYT node stored by the tree and create a new parent node and a new leaf node to represent the new string. We give the new parent node the old ID of the NYT node, we give the leaf node the old NYT ID - 1, and we give the NYT node the old ID - 2. We then add the NYT node as the new parent's left child and the new string node as the node's right child. We then append the new parent to the tree at the position where the NYT node was previously. We then add the new parent node and the new string node to the linked list representing ndoes of weight 1. If we have seen the node before, we can just jump to the node from the hashmap and do the weight list checks before incrementing the frequency. We then move up to the node's parent and do the weight group checks on that one before incrementing its frequency. Repeat this until we reach the root node.

3.2 Encoder

The encoder class reads in a file using the FileInputStream class. The file reads in a byte and converts it into a string of the bits it represents. The encoder then checks if the bit string has already appeared in its Adaptive Huffman tree or not using the hashmap. If it has already been seen, then it encodes the path from the root to the node. If it has not already been seen, then it encodes the path from the root to the NYT node and then encodes the ASCII representation of the character, with a 0 bit on the front to make it a whole byte length.

The encoding works by keeping a buffer of results from these operations. This buffer is a string of bits from any previous operations. For example, if we encode the path to the NYT node and then the ASCII representation of a character, we add the encoded results to this buffer. When this buffer gets a length of 8 or greater, the encoder removes the first 8 bits from the

buffer, converts these into its character representation and writes it to the compressed file.

If we reach the end of the file and there are fewer than 8 bits that need to be added to the file, we start a path to the NYT node to make the decoder believe that it is waiting for a new character right at the end and then fill the rest of the space with 0s if there is any space left over. By doing this, the decoder will either :

- Only receive part of the path as that is all there was room for, so will go down a certain length of the path towards the NYT node and finish before it reaches it, finishing the process acceptably.
- Receive the entire path, receive less than a byte of 0s since there must be at least 1 bit in the byte that represents actual file content and a certain amount of bits dedicated to the path to the NYT. Therefore, the decoder will not actually add this character into the tree. It will instead start reading it, find there are no more bytes after this and exit.

The encoder can also give the tree variable amounts of bits to add to the tree that need to be specified. By default it uses 8 (byte) size, but it can also use 4 or 2, which need to be passed as command line arguments. These produce different trees, since we have fewer bit combinations that we can have depending on how few bits we choose to build the tree with, so overall, a tree where the nodes represent 2 bit strings will have fewer nodes than a tree where each node represents 8 bit strings, but each node will have a higher weight.

3.3 Decoder

The decoder carries out the reverse of this process. It opens a FileInputStream for reading the file and reads in byte per byte. It also has a string buffer of the 8 bits of the byte being read in, as these are needed for determining the characters and the path of the tree. To get the current bit for the operation, we just need to remove the bit from the beginning of the buffer. When this buffer runs out of bits, we load in the next byte into the buffer from the file.

It starts with a node that it is currently examining. When it reads in a

bit, it asks the tree to get the left child of the current node if the bit is a 0, or get the right child if the bit is a 1. When it gets this child, it asks the tree if the root is the NYT node or node, and if it is not, then find out if it is a leaf node or not. If it is the NYT node, then it reads in the next 8 bits from the file to get the ASCII representation of the new character to add. If it is a leaf node, then it gets the character that the leaf node represents and writes it to the decompressed file. This process continues until the end of the file has been reached.

4 Testing

For testing, I made a class in collaboration with another student, Martynas Noreika, that takes the root of a tree and then prints it out in the terminal. Here I could manually check what my tree looked like, with weights, ID numbers and the string that the character represents.

5 Implementation

Both my encoder and decoder work with a number of different file types, including images and audio. They can be executed separately using the ant file as demonstrated below. I have constructed an ant file for assembling the project. I have not compiled any code in this handin package, nor have I got the compressed and decompressed files, so I have an ant command to carry those out for you : to automatically make the AdaptiveHuffmanCoding.jar file for the project and produce encodings and decodings of every included test file, execute

```
$ ant
```

and by default it will execute the encoder and decoder on all the test files, along with producing encodings using 4 bit strings in the tree and 2 bit strings in the tree. This took about 4 and a half minutes on lab machines. To encode a file, use

```
$ ant encode filename bitsInStringToUse
```

So for example, if I wanted to encode a file called "test.txt" with 8 bit strings (byte) in the tree, I would use

```
$ ant encode test.txt 8
```

This will produce the file test.compressed8.txt, which can be decompressed. If I wanted to encode "test.txt" with 4 bit strings in the tree, I would use

```
$ ant encode test.txt 4
```

This will produce test.compressed4.txt. This cannot be decompressed, as 4 bit and 2 bit string decoding was not implemented, but we can still use compression sizes for tests for it. To decompress a "test.compressed8.txt", use

```
$ ant decode test.compressed8.txt
```

This will produce test.decompressed.txt. To produce an 8 bit string tree node storage encoding and a decoding for test.txt, use

```
$ ant encode-and-decode text.txt
```

6 Results

I compared a number of different file types for the compression and decompression. These are included in the test items directory. I will explain results below.

6.1 Small Text Files

I used the "lorem ipsum" file for small text compression testing. The compressor worked very well for text files, since they have not already undergone any compression. We can see that when we use fewer bits to store words in the tree, the file is bigger since we have fewer possible nodes we can use and so there is lots of repetition for small characters. The results are below :

- loremipsum.txt : 1.4 kb
- loremipsum.compressed8.txt : 801 bytes - 57.1% of the size of the original.
- loremipsum.compressed4.txt : 1.2kb - 85.7% of the size of the original.
- loremipsum.compressed2.txt : 1.5kb - 107.1% of the size of the original.

6.2 Large Text Files

For this test, I used the complete works of Shakespeare test files.

- complete_shakespeare_works.txt : 5.5 mb
- complete_shakespeare_works.compressed8.txt : 3.2 mb - 58.2% of the size of the original.
- complete_shakespeare_works.compressed8.txt : 4.7mb - 85.5% of the size of the original.
- complete_shakespeare_works.compressed8.txt : 5.8mb - 105.5% of the size of the original.

6.3 Bitmap Files

Bitmap files are not compressed at all - they have a representation for every pixel. Therefore, the compression rate is large. I used the duck.bmp file for this test.

- duck.bmp : 819.1kb
- duck.compressed8.bmp : 543.0kb - 66.2% of the size of the original.
- duck.compressed4.bmp : 639.3kb - 78.04% of the size of the original.
- duck.compressed2.bmp : 726.7kb - 88.7% of the size of the original.

6.4 JPEG

A .jpg image has already undergone lossy compression by removing details in the image that the human won't be able to notice, such as difference in colour depth, so the results show that the compression is not affective on pre-compressed images.

- testpic.jpg : 87.6kb
- testpic.compressed8.jpg : 87.6kb - same size as original.
- testpic.compressed4.jpg : 88.8kb - 101.4% of the size of the original.
- testpic.compressed2.jpg : 97.8kb - 111.6% of the size of the original.

6.5 PDF

A PDF already has a lot of compression applied to it too, so the results are not too different from the original.

- test_document_pdf.pdf : 77.1kb
- test_document_pdf.compressed8.pdf : 76.4kb - 99.1% size of the original.
- test_document_pdf.compressed4.pdf : 78.2kb - 101.4% size of the original.
- test_document_pdf.compressed2.pdf : 86.4kb - 112.2% size of the original.

6.6 MP3 Audio

MP3 Audio is already undergone lossy compression by removing details that a human will not notice, such as certain frequencies that are not able to be perceived, so results are not too different from original material.

- Dave Brubeck Quartet - Blue Rondo A La Turk.mp3 : 6.5mb
- Dave Brubeck Quartet - Blue Rondo A La Turk.compressed8.mp3 : 6.4mb - 98.4% size of the original.
- Dave Brubeck Quartet - Blue Rondo A La Turk.compressed4.mp3 : 6.6mb - 101.5% size of the original.
- Dave Brubeck Quartet - Blue Rondo A La Turk.compressed2.mp3 : 7.2mb - 110.8% size of the original

7 Conclusion

Clearly the Adaptive Huffman Coding is a very useful compression technique. It needs no context for the data that is being streamed and it is a "zero-memory" solution, meaning that the data only needs be passed once and nothing else from the sender is needed to be referred back to or asked again for from the current data - all information can be inferred from the given

data at the given point and the tree that has been constructed. The adaptive solution also means that there needs to be no initial pass over the data to collect frequencies and then encode, which halves the time taken to encode the data as there only needs to be one pass over it. The compression results are very good too as we can see from above with data that has not been compressed.