CS3302 Compression Practical

Sizhe Yuen

2016-10-07

1 Introduction

In this practical we were asked to implement the Adaptive Huffman algorithm to encode and decode files. My implementation was able to construct the code tree incrementally, swapping nodes when necessary to maintain the sibling property and use the code tree to both encode and decode text and other file formats.

2 Design

2.1 Lookup table

In my `AHEncoder` class, I have the field:

```
private HashMap<Integer, Node> lookup;
```

This field is use to store a direct index to every Node in the Huffman tree. The integer is the integer representation of the symbol that was read from the file.

2.2 Encode

In my encode function, I read each character from the input stream and check the lookup table. If the lookup table already contains that character, then from the lookup table I can get the node in the tree and build the code up traversing up through the parents. Otherwise I send the NYT code and the code for the uncompressed character, then insert the new node into the tree. The basic algorithm was as follows:

```
while $c \neq EOF$ do
    if $c$ in lookup table then
        getCode($c$)
    else
        getCode($NYT$)
        getUncompressed($c$)
        insert($c$)
    end if
    updateTree($c$)
end while
```

I found the encoding process to be quite simple to understand and most of the problems I encountered when trying to implement the algorithm was to do with the code tree and how to swap two nodes.

2.3 Swapping nodes

My condition to check whether nodes should be swapped is simple. Whenever a node's weight is about to be incremented, I traverse the tree to find the node with the highest index and the same weight. If the two nodes are not the same and are not parent/child of each other, then they are swapped. To actually swap the nodes, instead of swapping just the subtrees and the labels of the nodes, I chose to swap the nodes themselves and their parents. This was because I found it confusing if the subtrees and labels were swapped, the pointers to the nodes still stayed the same so I would update the wrong nodes.

2.4 Decode

The algorithm for decoding was essentially the same for encoding, except for building the code. What I do is keep reading the next bit until I reach a leaf node, going either left or right depending on whether I get a 0 or 1 respectively. Once I do we check if the node we are on is the NYT node or not. If it is, read the next 8 bits as uncompressed, otherwise we can output the data of the current leaf node.

```
while $b \neq EOF$ do
    if $currentNode$ is a leaf then
        if $currentNode == NYT$ then
            $c$ = getUncompressed(next 8 bits)
            insert($c$)
            read next 8 bits
        else
            getCode($currentNode$)
        end if
        $currentNode = root$
        updateTree($c$)
    else
        if $b == 0$ then
            $currentNode = currentNode.left$;
        else
            $currentNode = currentNode.right$;
        end if
        read next 1 bit
    end if
end while
```

This algorithm is good in that we are always traversing down the tree as we read each bit, so we don't have to go back or do any extra traversal except when updating the tree.

3 Outcome

3.1 Text files

In general, text files compress easily

3.2 Image files

For image files, I have found that the size of the output from the encoding depends on the image file type. For example, when I try to encode `.jpg` and `.png` files, the file size for the output from the encoding is larger than the original image itself. However, when encoding a `.bmp` image, the encoding works very well.

Original size	Encoded size	Compression	File format
23538	23884	101%	`.png`
1523	1632	107%	`.jpg`
32886	5741	17.5%	`.bmp`

This would be because image formats like `.jpg` already do their own compression and so my Adaptive Huffman algorithm is not able to compress it further, while `.bmp` files are uncompressed.

3.3 Other file formats

Similar to image files, the compression of other files depends on their file format.

Original size	Encoded size	Compression	File format
119520	119939	100.4%	`.pdf`
759542	750048	98.7%	`.pptx`
787306	657929	83.6%	`.wav`

4 Evaluation

4.1 Getting highest index node in block

To get the highest index node, I currently have to traverse the entire code tree, adding all nodes with the same weight to a list, then checking through that list to find the node with the highest index. This can be very slow if the tree becomes very big.
A solution would be to keep a `HasMap` similar to the lookup table I already have for the Node lookups for every weight block. This would involve book keeping on every update on the tree to both add a node when the

4.2 Last byte of encoding

Because this implementation is written in Java, when writing out the bits to the compressed file, the minimum amount that can be written is a byte (8 bits). This was a problem for writing and reading because we want to work in bits, but I got around this problem using a String as a buffer. The problem was when we get to the final few bits, if they don't make up 8 bits, I still need a full byte before writing out to the file, so I fill the rest of the the missing bits with 0s.

```
//fill end with 0s
while (writeBuffer.length() % 8 != 0) {
   writeBuffer += "0";
}
```

Now when it comes to the decoder, it may interpret the last 8 bits differently and this is an issue.