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.

In this report I detail how I implemented the algorithm and the outcome of running my code against different file formats.

2 Design

2.1 Lookup table

In my `AHCoder` 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. This `HashMap` makes it very easy to output the code of a symbol already seen before.

We can directly find Node for that symbol and move up the tree, appending a 0 or 1 depending on if we were our parent's left or right node until we reach the root.

2.2 Encoding

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($lookup[c]$)
  else
    getCode(NYT)
    getUncompressed(c)
    insert(c)
  end if
  updateTree($lookup[c]$)
end while

`getCode()` is given a node in the tree to traverse up. When reading the input file, I am reading using `ints` and storing that as the data in the Node. Only when having to output the uncompressed symbol out, I parse that `int` into binary before writing it out. Otherwise the binary that is written comes from the code tree. I found the encoding process to be quite simple to understand. The difficulty came in dealing with bytes instead of bits because of Java and how to update the tree properly.

2.3 Writing out in bytes

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 cannot just send a few bits and not a full byte as Java will fill in the rest of the byte. I got around this problem by using a `StringBuilder` as a buffer and only writing out to the file once the String has length of at least 8.

```
if (buffer.length() > MAX_BUFFER_SIZE) {
   writeOut(buffer)
}

...

private void writeOut(StringBuilder code) {
   while(code.length() >= 8) {
      out.write(Integer.parseInt(code.substring(0, 8));
      code = code.delete(0, 8);
   }
}
```

I use a constant for maximum buffer size here because it would be inefficient to write every time the buffer gets a full byte as file IO is slow. I also don't want to write only after the whole input file is read because then too much could be stored in memory when processing a large file.

Another 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. My solution for this came from Martynas Noreika, who suggested to fill the rest of those 8 bits with as much of the NYT symbol as possible. So after reading the last symbol when decoding, when we read the remaining bits, we are only going to go down the path to the NYT node. If we end up actually reaching the NYT leaf node, we will try to read the next 8 bits for the uncompressed symbol and fail, because we are on the last 8 bits, there will not be more than 8 bits after the NYT code. Then we can fail to get those next 8 bits and not write any wrong symbols to the decoded file.

2.4 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. The subtrees will then follow the nodes as they are swapped and the indexes also have to be swapped to keep the sibling property. 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.

2.5 Decoding

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 we reach a leaf node, we check if the node we are on is the NYT node or not. If it is, read the next 8 bits as uncompressed, as we have just received a NYT node. Otherwise we can output the data of the current leaf node and reset the current node pointer back to the root.

---

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

Another buffer is used in a similar way to the encode step. We always want to

read more into the buffer so we can keep decoding, but not so much as to have to store a very large String in memory. So again I use the same `MAX_BUFFER_SIZE` to determine when to read more. 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

The compression of text files depends on the content of the file. If for example the text file contained no duplicate symbols, then the encoded file would be larger than the original, as we always have to output the codeword for NYT and also the uncompressed code. In the extreme opposite case where the text file is only the same symbol for the entirety of its length, the best compression we can achieve is 1/8% or 12.5% of the original file, going from 8 bits per symbol to a single bit per symbol.

Original size	Encoded size	Compression	File
95	172	181%	All unique symbols
526501	65815	12.5%	All same symbol
5458196	3157434	57.8%	Sample text (Shakespeare)

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. In my example `.bmp` file, the picture also only consists of black and white, which means there would be less symbols to encode and therefore could achieve close to 12.5% compression.

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`

I think part of the reason for the compression of different file formats being bad is they use a lot more symbols than a text file would. Because of this we are sending more and longer NYT symbols and uncompressed symbols instead of sending the shorter codewords from our tree, causing the compression to be equal or worse than the original.

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 looping through that list to find the node with the highest index. This can be very slow if the tree becomes very big as we traverse the entire tree each time to get all the nodes.

A solution would be to keep a `HashMap` similar to the lookup table I already have for the Node lookups for every weight block. This would involve overhead on every update of the tree to update the `HashMap` but this update should cost less than traversing the whole tree on every swap. If the `HashMap` only stores the highest node of every block, the time needed to fetch that node is constant, instead of being linear.