Huffman Compression Analysis

Mitchell Pomery (21130887)

Algorithm

Huffman compression, or Huffman coding, is a loss-less form of compression. It works by replacing more common bit patterns, with smaller bit patterns, and less common bit patterns with longer bit patterns.

Due to the fact that each compressed files ideal huffman tree is different, the input stream has to be parsed twice. The first time over creates the huffman tree, which can then be converted to a map, and the second time to use the map to compress the output. Decompressing similarly takes three steps, reading the huffman tree from the file, reading the mappings, and then reading the compressed data and converting it into it's uncompressed form.

Reading and writing from disk is also slow if done one byte at a time, so the helper class writer buffer was written, allowing for large amounts of data to be stored in the RAM before being written sequentially to disk. Before implementing the buffer, writing 25MB to disk would take 2 minutes, whereas with the buffer in place, writing the same 25MB takes only 3 seconds. The WriteBuffer class also has the advantage of being used to make sure that bits are converted to bytes for writing.

Compressing

Creating the Huffman Tree:

```
for each character in the file
  frequency[character]++
for each character possible
  if frequency[character] > 0
      create huffman tree with character and frequency
      add huffman tree to a priority queue

while priority queue size > 1
   a =  priority queue dequeue
   b =  priority queue dequeue
   c = a frequency +  b frequency
   create huffman tree with child trees a and b, and frequency c

final huffman tree =  priority queue dequeue
```

Converting the Huffman Tree to a Map

```
add top huffman tree to a tree list
values list = list containing one empty item
map = empty array
position = 0

while position < tree list size
   if tree list[position] is a leaf
      key = tree list[position]'s character
      value = values list[position]
      map add (key, value)
   else
      initial value = values list[position]
      left =  initial value + “0”
```

```
    right =  initial value + “1”
    add left tree to tree list
    add right tree to tree list
    position++

return map
```

Converting Input into a Compressed Output

```
for each character in the input
  find it's new value using the map
  write the value to the output
```

Decompressing

Decompressing the Data

```
root tree
current tree = root tree
while reading the file bit by bit
  if bit == 0
    current tree = current tree left tree
  else
    currentTree = current tree right tree
  if current tree is a leaf
    write current tree byte to output stream
    current tree = root tree
```

Description of Implementation

Several classes were used to implement huffman compression and decompression. Because huffman compression is bit pattern based, an easy way to use bits was required, so the BitByteConverter class and the WriteBuffer class were written. The BitByteConverter class purely changes bytes into boolean arrays and back again, while the WriteBuffer class lets us easily write bits and bytes. The HuffmanTree class is used to make sure we get the maximum compression possible on the data and the Huffman class does the compression and decompressing. The Compress class is a nice wrapper that lets you compress and decompress files from the command line and also lets you know how to use it.

BitByteConverter

Java has no nicely inbuilt ways of dealing with bits, so a BitByteConverter class was written. It takes a byte and turns it into a boolean array, and it can take a boolean array of length 8 and turn it into a byte. This made dealing with bits easier, as they could be stored as boolean arrays, and easily be manipulated. As such, the WriteBuffer class was also written so that it could deal with the boolean arrays as inputs when writing bits.

WriteBuffer

The WriteBuffer class drastically improved compression times, as it kept the data to write to disk in RAM until there was a significant amount to justify writing it to disk, instead of waiting for each byte to be written to disk. The flush method also made sure that any bits left over at the end of the compressed data were converted to bits, and the padding was marked so that the decompressor would not try to convert them to stray data at the end of the file. This was achieved by having the last byte contain 1's for the number of bits to be ignored from the previous byte.

HuffmanTree

The Huffman Tree is an integral part of Huffman Compression, as it is used to make sure that the outputted stream is as small as possible. To build a good huffman tree, you take every byte in the file, and count it's frequency, putting it into a priority queue. You then take out the two items with the highest priority, and put them as the left and right tree in a new huffman tree with the frequency set to the sum of two child frequencies. You put this back into the huffman tree and repeat until you are left with only one item in the priority queue.

public boolean[] toBooleanArray()

Converts the huffman tree to a boolean array that can then be written to an output stream so that it can also be used to recreate the huffman tree for decompression.

public boolean[][] toArrayList()

Convert the huffman tree to a map so that bytes can easily be converted into their compressed form.

Huffman

The huffman class is the main class used for compressing and decompressing data. It only has two methods, one for compressing and one for decompressing data. This class relies heavily on the other classes for support to do the compression.

Theoretical Analysis

Compression

Using Huffman Compression, the time taken to compress a file is affected by the file size, and the number of different bytes in the file.

Firstly the file is traversed and the frequency of each byte is recorded. This is done byte by byte, so as the number of bytes in the file doubles, so does the time to collect the frequencies. Therefore the initial reading of the file and collecting of frequencies takes linear time, dependent on the file size. These frequencies are then put into a priority queue to create a huffman tree,

When creating the huffman tree, a priority queue is used to sort out where each character should appear on the tree. The first two items are removed from the priority queue and made children to a new huffman tree with the frequency set to the sum of it's children frequencies. As the number of different bytes in the file grows, the time taken to create the huffman tree also grows.

For a file containing two different bytes, the priority queue is only accessed once. Meanwhile a file containing 4 different bytes has its priority queue accessed 3 times, and a file containing 8 different bytes has it's priority queue accessed 7 times. This pattern continues right up to a file containing all 256 possible different bytes. This shows that the time taken to create the huffman tree is linearly dependent on the number of different bytes in the file.

Using the mapping made, each bit in the file is converted to a string of bytes. This is achieved by performing a breadth first search on the huffman tree, while keeping track of several lists. The time taken to complete the breadth first search depends on the number of nodes in the huffman tree. The number of nodes in the huffman tree is approximately equal to twice the number of leaves in the tree, and so a tree with twice the number of leaves will only take twice as long to search. Creating a map takes linear time, depending on the number of different bytes in the file.

Once the huffman tree is created, a mapping is then made to quickly convert bytes to strings of bits. As each byte is read, it is converted to an integer and the bit string it maps to is written to a file. It takes the same amount of time to find each bit string in the map using the integer, so the thing that affects the time is the number of bytes in the input file that need to be mapped. This means that the final step runs in linear time.

Compressing a file takes linear time and is dependent on the file size, and the number of different bytes in the file. However the number of different bytes in the file maxes at 256, and the file size has no maximum, so as the file gets larger, the time becomes more dependent on the file size, and the number of different bytes becomes insignificant.

```
Time to run huffman compression over a file:
          = O(n + b)
n = file size
b = number of different bytes in the file
```

Decompression

Recreating the huffman tree from the compressed data is done by reading bits until a complete tree is made. It is done similarly to when creating the tree for compression and runs in linear time, dependent on the number of different bytes in the decompressed file.

Once the tree has been created, the rest of the data is explored bit by bit, and the huffman tree is traversed until a leaf node is hit. The byte that the leaf holds is then written out, and the tree traversal is restarted until the file is finished. Due the the fact that the tree is traversed, instead of looping through a map to find the byte, the decompression runs in linear time, depending on the amount of data to decompress.

Overall the decompression takes linear time dependent on the number of bytes in the original file, and the length of the compressed data. As more data is compressed, the limit of 256 different bits becomes insignificant, and so the main change in decompression time is the size of the compressed data.

```
Time to run huffman decompression over a file:
          = O(b + n)
n = compressed data size
b = number of different bytes in the decompressed file
```

Experimental Analysis

Experimental analysis was undertaken by compressing several different files ranging in size and file type. Already compressed file formats such as PDF's, MP4's and JPG's saw virtually no saved space when compressed, whereas uncompressed file types such as TXT files and AVI's saw massive space saved.

File Type	Count	Uncompressed (b)	Compressed (b)	Compress (s)	Decompress (s)	Ratio
txt	6	9903270b	5668471b	0.6	1	57%
avi	1	5015852064b	3416221394b	325	430	68%
pdf	4	33476141b	32395551b	2.5	3.7	97%
jpg	3	4061109b	4059673b	0.3	0.6	100%
mp4	1	17069668b	17069989b	1.2	1.8	100%

Before implementing the WriteBuffer class, the time to write a 25MB compressed file was about 2 minutes. After it was implemented, this time dropped drastically to around 2 seconds. This is due to the fact that instead of writing each byte to the disk one at a time, the class would hold an array of bytes to be written to disk in bulk, causing one big sequential write instead of several small randomly timed writes. This class also made it realistic to compress and decompress large files, such as the 4GB file that was tested.