

Lab 9: Huffman Encoding

Overview

A Huffman Tree is a specific implementation of a Binary Tree, where the left branch represents a 0 in some binary string and the right branch represents a 1 in the same binary string. You will be constructing a Huffman Tree, based on the input from a text file, storing that text file as a compressed binary file, and decoding the binary file back into text.

Huffman Code

In a basic implementation of a Huffman Code, ASCII characters are encoded, using a technique based on the frequency of the character's usage in a given text, as binary strings, the more common characters being assigned fewer bits than the less common characters. When using this encoding while saving the text as a file, we can thus reduce the amount of space required by the text, as the more common characters will often use less than the minimum 8 bits required to store an ASCII character.

Assignment Detail

You will write a program that will ask the user for a text file, load that text file into memory, and build a Huffman Tree by generating a **frequency map** for all the characters in the text. A frequency map is a table that stores the number of times a character is used in a given text. This frequency map will be an integer array. Every character will use the index that matches that character's ASCII number to store its frequency. Note that white space characters, such as space, newlines, tabs, etc should be considered valid characters and will have their own frequency count.

Once a Huffman Tree has been created, the user will be prompted for a filename in which to save the text in a compressed binary file, using the Huffman encoding described by the Huffman Tree. You will then write the encoded text to a binary file a single bit at a time using a `BitOutputStream` object. This binary file will use the `.bin` extension

Once the file has been compressed, you will report the new size of the compressed file as well as the size savings in terms of percent savings.

Finally, you will decode the binary file you created to verify the encoding and binary file creation was successful and display the text in the console.

Compressing the text

To compress the text into binary using the generated Huffman encoding, you will first build a lookup table using a `HashMap` that will have as its key an ASCII character and the value will be a `String` representation of the Huffman binary string. This is done to allow for quick retrieval of the encoding.

Once the table is built, you will iterate through the characters of the text, find the Huffman code in the `HashMap`, then write the bits with a `BitOutputStream`. Do this until all of the characters of the text have been written to the binary file.

Decompressing the text

When decompressing the binary file back to text, you will use a `BitInputStream` to read from the binary file one bit at a time. Using each bit, you will traverse through the `HuffmanTree`, starting at the root, until you reach a leaf node. This node will contain the encoded character. Once a character is found, add that character to the output `String` and start traversing again from the root. Repeat until all bits have been read.

Creating a HuffmanTree

When building a `HuffmanTree`, the goal is to build from the bottom up, creating all the leaf nodes, then build the paths up to the root. To do this, in the `buildTree()` method we will first construct the leaf nodes for each character using the frequency map and the character's ASCII values. These nodes will all be stored in a `PriorityQueue`, a data structure that sorts its elements according to their `compareTo()` method and returns them in ascending order (from low to high). Only store leaf nodes that contain characters with at least one occurrence in the text.

Once all the leaf nodes have been created, and the nodes with a non-zero frequency value have been added to the `PriorityQueue`, we will start building the tree upwards. To do this, we will take the first two nodes from the queue (which will be the two with the lowest frequencies) and combine them, in order, as the children of a new inner node. This node will be added back to the queue. Continue in this fashion, combining nodes into new inner nodes until there is only one node remaining. This will be the root of the tree.

The BitStream library

You will need to import the `BitStream.jar` file into your project in order to access the `BitOutputStream` and `BitInputStream` classes that will allow you to write individual bits to a file. Without these classes, you would only be able to write a minimum of a byte (8 bits) at a time. These classes use integer and `String` representations of bits and read/write the corresponding 0 or 1 values.

Additional information

- Make use of the UML class diagrams and the java docs. There is a wealth of information contained in both of those documents
- When creating the HuffmanTree, the PriorityQueue will at first contain all leaf nodes. To build the tree, we will remove the first two nodes in the queue and assign them as the left and right nodes of a new node, which we will then add back to the queue
 - The PriorityQueue sorts the nodes by the *total frequency* of the tree rooted at that node, so a node that you create using two leaf nodes will have a frequency that is the sum of the frequency of those two leaf nodes. A node with several descendants will have a frequent that is the sum of the frequencies of all of those nodes.
 - We will continue combining nodes in this manner until there is only one node remaining, which will end up being the root of the HuffmanTree
- Once the HuffmanTree has been completed, you will then traverse through the tree and add the entries to your HashMap, using the character value of the node as the key.
 - The traversal itself will build the bit string that will be the value that will be stored in the HashMap.
- After the HashMap is filled, you will use it to get the new binary encodings for each character of the input file and write the bits to your output file using a BitOutputStream.
- When decompressing the .bin file, you will use a BitInputStream to read the bits one at a time, which are then used to look up characters in the HuffmanTree.
 - Each bit string will eventually find a leaf node, which is the character that has been encoded. Once a leaf node is discovered, store that character in an output String. The next bit will be the first bit of a new character.
- When reporting the file size and compression savings, the File class's `length()` method may prove useful.
- To determine if a node is a leaf node, the `instanceOf` keyword will prove useful.
- Remember to add the `BitStream.jar` library to your project
- Sample output of a successful program run is given below, Your output should match this as closely as possible.

Sample Output

Enter file to be compressed (without .txt extension): NaturalScience

SYMBOL	WEIGHT	HUFFMAN CODE
o	72	0000
r	74	0001
e	151	001
s	77	0100
n	78	0101
y	18	011000
S	5	01100100
O	2	011001010
'	3	011001011
T	11	0110011
d	41	01101
a	90	0111
f	22	100000
A	11	1000010
W	12	1000011
h	47	10001
i	95	1001
w	24	101000
g	24	101001
p	25	101010
x	5	10101100
b	7	10101101
v	15	1010111
t	104	1011
	216	110
m	28	111000
u	28	111001
	61	11101
l	62	11110
z	2	1111100000
-	1	11111000010
E	1	11111000011
q	1	11111000100
M	1	11111000101
C	2	1111100011
L	2	1111100100
N	2	1111100101
F	2	1111100110
G	2	1111100111
,	4	111110100
k	4	111110101
I	8	11111011
c	33	111111

Enter the destination file name: huff

Compression complete.

Old file size: 1472

Compressed size: 819

Compression savings: 55.64%
Press enter to decompress file.

Decompression complete.

When the ebbing tide retreats
Along the rocky shoreline
It leaves a trail of tidal pools
In a short-lived galaxy
Each microcosmic planet
A complete society
A simple kind mirror
To reflect upon our own
All the busy little creatures
Chasing out their destinies
Living in their pools
They soon forget about the sea
Wheels within wheels
In a spiral array
A pattern so grand and complex
Time after time
We lose sight of the way
Our causes can't see their effects
A quantum leap forward
In time and space
The universe learned to expand
The mess and the magic
Triumphant and tragic
A mechanized world, out of hand
Computerized clinic
For superior cynics
Who dance to a synthetic band
In their own image
Their world is fashioned
No wonder they don't understand
Wheels within wheels
In a spiral array
A pattern so grand and complex
Time after time
We lose sight of the way
Our causes can't see their effects
Science, like nature
Must also be tamed
With a view towards its preservation
Given the same
State of integrity
It will surely serve us well
Art as expression,
Not as market campaigns
Will still capture our imaginations
Given the same
State of integrity
It will surely help us along

The most endangered species
The honest man
Will still survive annihilation
Forming a world
State of integrity
Sensitive, open and strong
Wave after wave
Will flow with the tide
And bury the world as it does
Tide after tide
Will flow and recede
Leaving life to go on
As it was

Process finished with exit code 0