Lab 5: Huffman Codes and File Compression

March 31, 2016

Binary trees can be used in various encoding and decoding problems. For example, Morse code represents each character in a message by a sequence of dots and dashes. This encoding can be represented using binary trees. In this lab we will focus on another coding scheme, Huffman codes.

Variable-Length Codes

The idea of a variable-length code scheme is to use shorter codes for those characters that occur more frequently and longer codes for those used less frequently. For example, 'E' in Morse code is a single dot, whereas 'Z' is represented as $--\cdot\cdot$. The objective is to minimize the expected length of the code for a character. This reduces the number of bits that must be sent when transmitting encoded messages. These variable-length coding schemes are also useful when compressing data because they reduce the number of bits that must be stored.

Suppose that some character set $\{C_1,\ C_2,\ \ldots,\ C_n\}$ is given, and certain weights $w_1,\ w_2, \ldots,\ w_n$ are associated with these characters; w_i is the weight attached to C_i and is a measure of how frequently this character occurs in messages. If $l_1,\ l_2,\ \ldots,\ l_n$ are the lengths of the codes for characters $C_1,\ C_2,\ \ldots,\ C_n$, respectively, then the *expected length* of the code for any one of these characters is given by

$$w_1l_1 + w_2l_2 + \ \ldots\ + w_nl_n = \Sigma_{i=1}^{n} w_il_i$$

For example, consider the five characters A, B, C, D, and E, and suppose they occur with the following weights

Char	A	B	C	D	E
Weight	.2	.1	.1	.15	.45

In Morse Code with a dot replaced by 0 and a dash by 1, these characters are encoded as follows:

Char	A	B	C	D	E
Weight	.2	.1	.1	.15	.45
Code	01	1000	1010	100	0

Immediate Decodability

Another useful property of some coding schemes is that they are *immediately decodable*. This means that no sequence of bits that represents a character is a prefix of a longer sequence for some other character. Note that the preceding Morse code scheme is not immediately decodable because, for example, the code for E (0) is a prefix of the code for A (01), and the code for D (100) is a prefix of the code for B (1000).

Huffman Codes

The following algorithm, given by D. A. Huffman in 1952, can be used to construct coding schemes that are immediately decodable and for which each character has a minimal expected code length.

Algorithm 1 Huffman's Algorithm

▷ Constructs a binary code for a given set of characters for which the
▷ expected length of the bit string for a given character is minimal
Initialize a list of one-node binary trees containing the weights $w_1,\ w_2,\ \ldots,\ w_n$, one for each of the characters $C_1,\ C_2,\ \ldots,\ C_n$.
repeat
    Find two trees T' and T'' in the list with roots of minimal weights w' and w''.
    Replace these two trees with a binary tree whose root is $w' + w''$, and whose subtrees are T' and T'', and label the left pointer to the left subtree as 0 and the right as 1.
until $n - 1$ times
The code for character C_i is the bit string labeling of the path in the final binary tree from the root to the leaf for C_i.

In Class Example

We will do an example in-class.

The immediate decodability property of Huffman codes is easy to see. Each character is associated with a leaf node in the tree, and there is a unique path from the root of the tree to each leaf. Based on this observation we can construct an easy decoding algorithm:

Algorithm 2 Huffman Decoding Algorithm

```
1:  ▷ Decode a message using a Huffman tree
2:  Input: Huffamn binary tree and a bit string encoding of some message.
3:  Output: The decoded message.
4:  Initialize a pointer p to the root of the Huffman tree.
5:  while The end of the message string has not been reached do
6:      Let x be the next bit in the string.
7:      if x == 0 then
8:          set p equal to its left child pointer.
9:      else
10:         set p equal to its right child pointer.
11:     end if
12:     if p points to a leaf then
13:         display the character associated with that leaf.
14:         reset p to the root of the Huffman Tree.
15:     end if
16: end while
```

Problem 1

You should construct a program that does the following:

1. Take a weights file as input. The file should include a list of character and the weight for each character. Each line of the file will have two parts, a character and a weight separated by a space.

2. The program should use the input to construct the binary Huffman tree. The binary tree should be your own code and not from the STL.

3. The program should then be able to take one of two types of files.

   (a) A text file consisting of text using only the character from the weights file. If the user enters this type of file your program should do the following.

   - Use the above Huffman tree to construct the encoding Huffman encoded version of the text file.

4. A huffman encoded file based on the same weights from above. Your program should do the following:

    - Use the above Huffman tree to decode the file back to the original text version.

Problem 2

Using your code from problem 1 above, write a program to compress a file using a Huffman code and to decompress a file generated using this code. The program should do the following:

1. The program should first read through the file and determine the number of occurrences of each character in the file. The weight of each character will be the frequency count for that character.

2. The program should then use these weights to construct the Huffman codes for the characters in the file.

3. It should then read the file again and encode it using these Huffman codes and generate a file containing this encoded data. The output file should be called filename.hzip, where filename is the name of the original non-encoded file.

4. Produce a second output file, called filename.hcodes, which included a list of the characters and their binary Huffman codes. Each line should contain one character and its corresponding code separated by a space.

5. The program should also compute, and print to the terminal, the compression ratio, which is the number of bits in the compressed file divided by the total number of bits in the original file (eight times the number of character is the file).

6. The program should also include an option to take as input a compressed file (a .hzip file), and a encoding file (a .hcodes file), and produce a decompressed version of the file using the codes from the encoding file.

Note The compressed output file needs to be a true binary file. That is, if you just write the character '0' to a file in-place of the character 'A', you are not actually saving space. They both use the same number of bits in their ASCII encoding. Thus you need to write the compressed file as just a string of binary bits.

What to turn in

You should just submit your code and a makefile for the above two problems. The makefile should just produce an executable for problem 2 above but you should use your code from problem 1 in your solution for problem 2.