Huffman Coding, LZW Lab—Monday, February 27 / Tuesday, February 28

Introduction

This week we will be reviewing Huffman Coding and looking at a new topic that was not covered in class, which is LZW Compression. The primary goal of this lab is to introduce LZW compression which will be used in the next programming assignment. Reviewing Huffman would be a secondary objective.

Huffman Encoding

Huffman coding is a common technique used for lossless compression of text. It uses relative character frequency to encode text such that the least possible space is taken on average. In this lab we will be focusing on encoding text into binary, though later you will learn other methods.

Encoding Implementation

Characters are encoded by runs of bits of non-constant length. The encoding is prefix free, which means that there doesn’t exist a character with a binary representation that is the prefix of another character’s representation. We represent an encoding as a trie, with each leaf representing a character, and the path from the root to the leaf describing its encoding. Each edge in the tree is assigned either 1 or 0. By convention, each left edge is 0. We can construct such a tree as follows:

- As input, assume that we have access to the relative probabilities with which each character appears. For a given text to compress, this can be generated in a preprocessing step before the compression. Treat each character as a tree of one node, with weight equal to its frequency, and construct a min-heap of the trees. While the tree is not empty, we preform the following steps.
  1. Remove the lowest frequency item from the heap (tree \( A \)), then remove the new lowest frequency item (tree \( B \)).
  2. Construct a new tree by creating a root node with two children: \( A \) as the left child, and \( B \) as the right child.
  3. Add this tree back into the heap, with weight equal to the combined weights of \( A \) and \( B \).

This process is complete when only one tree is left in the heap, our final Huffman coding tree.

Encoding Length

When transmitting an Huffman coded file, both the trie-backed encoding and the compressed contents have to be transmitted.

Relative to single character encoding schemes, Huffman’s algorithm produces an optimal encoding, that is, the compressed contents of the text will be the shortest length possible. Using the frequency with which each character appears, and the number of bits needed for each compressed character, we can find the length of the compressed text.

**Definition 1.** The number of bits to represent a character \( c \) is equal to depth of the leaf representing it, \( d(c) \) in the trie. The length, or number of bits to represent the entire text is therefore equal to the following expression.

\[
L = \sum_{c \in R} d(c) \times freq(c)
\]

\[1\]This direction of this step is convention; the reverse is just as valid, but for this course we will expect you to abide by this standard.
Loosely speaking, Huffman Coding makes the term \( d(c) \times freq(c) \) vary less between characters than it otherwise would, and therefore minimizes this summed over all characters. Each character is represented with brevity consistent with its relative importance, or frequency, in the text.

**Decoding Implementation**

Because the representation of the encoding is included, decoding is simple. While processing the text, follow bit by bit through the trie until a leaf node is reached, then output that character.

**LZW Compression**

LZW is a compression scheme where both single characters and sequences of characters can be given a unique encoding. Longer reoccurring sequences can be represented concisely as a single number instead of being represented with multiple characters each time. Through this, LZW utilizes patterns in the text to its advantage by representing them with less bits than it would otherwise take.

**Compression Implementation**

We start by encoding all characters using more bits than it would otherwise take, such as 12 bits instead of 8 bits for ASCII code. For characters, the higher order bits are padded with 0s. All are put on a symbol table to begin. As we scan through the file to compress, we do the following

1. Find the longest string \( s \) in the symbol table that is a prefix of the input. This will just be the initial characters when we begin.
2. Write the corresponding 12-bit encoding out.
3. Scan one character further than \( s \) in the input text, to include the next character, \( c \).
4. Add this longer string \( s + c \), to the symbol table, associated with the next unused encoding.

The step of looking ahead allows LZW to adapt to patterns in the text and find progressively longer and longer reoccurring portions in the text.

**Expansion Implementation**

Unlike in Huffman Coding, the symbol table doesn’t need to be transmitted with the compressed text. Instead, it can be generated while the compressed text is decoded. At the start, the padded single characters are put on the symbol table, keyed by their corresponding encodings.

1. Bob looks up the encoding in his symbol table, and outputs the corresponding string \( s \).
2. Bob looks up the following encoding in the symbol table.
3. If the encoding exists in the table, take the first character, \( c \) of the corresponding string. Add \( s + c \) to the symbol table, mapped to the next unused to encoding.
4. If the encoding doesn’t exist in the table, we know the unknown corresponding string must be the encoding that we are trying to add to the table right now! Therefore it has \( s \) as a prefix, and ends with the first character of \( s, c \). Add \( s + c \) to the symbol table mapped to the respective encoding.
Discussion Questions

What type of data structure makes sense to represent the symbol table during LZW compression? What about expansion?

*Solution.* For compression we are searching for the longest prefix in the remaining input text that has been seen before, and therefore has an encoding. Search by prefix makes a trie the perfect representation.

For expansion, we just need to map ordered integers, the encodings, to strings. This makes a standard string array appropriate.

What are pros and cons of Huffman Coding?

*Solution.* Pros: Huffman codings represents only the characters that are contained in the next, without wasting space in the encoding length for characters that are not present. Additionally, characters that occur most take less bits to encode each time.

Cons: It does not help with longer patterns in the text. Storing the symbol table takes up additional space. Therefore it isn’t appropriate for very small files where the table size would be significant. Encoding a text requires a preprocessing step to generate the table, does not work with a data stream.

What are the pros and cons of LZW.

*Solution.* Pros: It identifies and optimizes for longer reoccurring patterns in the text. The encoding doesn’t need to be transmitted.

Cons: It takes additional bits to represent single characters at the start or when it doesn’t match a longer pattern. Strings that reoccur still have to be represented more verbosely at first. Eventually the symbol table will fill up and present an issue. (Variable Width Huffman Coding allows the encodings to increase in length when this happens).

How can we get around some of these cons, or combine the pros. Like in sorting, many real compression implementations combine the benefits of multiple algorithms.

*Solution.* 1. Adaptive Huffman Coding works online, like LZW, by generating the symbol table as the text is processed. The trie is modified as the text is processed and the character frequencies change.

2. LZ77 and LZ78 are two compression implementations similar to LZW. Previously seen strings are represented by two integers that point backwards in the text to the start point and length of the re-occurrence.

3. DEFLATE separates a file into blocks, each is first compressed by LZ77 and then Huffman Encoded.

Practice Problems

**Problem 1.** Construct an optimal Huffman coding for the following alphabet and frequency table:

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.4</td>
<td>0.3</td>
<td>0.15</td>
<td>0.1</td>
<td>0.05</td>
</tr>
</tbody>
</table>

What is the average encoded character length for the above encoding?

*Solution.* The following encoded character length for the above encoding?

```
/ \                           
A / \                         
/ \ B                         
/ \ C                         
E   D
```
Problem 2. Gwen Stefani is rewriting her 2004 hit "Hollaback Girl" to be an educational children’s song with the purpose of educating the youth on the proper spelling of bananas. The infinite length song simply repeats "bananas", in code, "b-a-n-a-n-a-s". Without enough hard drive space, she tries to compress the text file of lyrics she wrote. Propose an effective compression method for a trillion lines of this song.

(a) Help Gwen compress the first word of her song using LZW compression, with a 3 bit output and only the alphabet \( R = \{ b, a, n, s \} \).

(b) Is the compression technique used in (a) satisfying? Compress the song using \( k \geq 5 \) bits for each output, using LZW. Which number \( k \) would return the most efficient compression of the song?

(c) Use Huffman coding to encode the lyrics of this song. How does the encoding length compare with the encoding length of LZW in part (b)?

Solution. (a) The dictionary and encoding for LZW with 4 bit output will be like this.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>b</td>
<td>a</td>
<td>n</td>
<td>s</td>
<td>ba</td>
<td>an</td>
<td>na</td>
<td>ana</td>
</tr>
</tbody>
</table>

For decompression, the receiver would know the initial alphabet, and output the letters that match the single characters. When 5, "an", is reached, they would similarly find it in their symbol table.

(b) Using only 3 bits will not be able to compress 'bananas' very much. The first 'bananas' will be 18 bits. From the second will be 12 bits. This is only a slight improvement from just encoding each character in \( R = \{ b, a, n, s \} \) with 2 bits each (This encodes 'bananas' in 14 bits). This can be improved by encoding using more bits. In fact, the whole string 'bananas' can be encoded in 6 bits using the LZW algorithm. This will encode the remaining 'bananas' with 6 bits each, greatly reducing the encoding length.

(c) Using Huffman, the frequency table will look like this.

<table>
<thead>
<tr>
<th>Character</th>
<th>b</th>
<th>a</th>
<th>n</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding</td>
<td>001</td>
<td>01</td>
<td>000</td>
<td></td>
</tr>
</tbody>
</table>

This will only reduce 'bananas' into 13 bits, which is very inefficient compared to our encoding in part (b). Remark! A compression strategy that could simply store "'bananas, B-A-N-A-N-A-S'∗10^{12}" would be a greatly reduced file size.

Problem 3. Construct an alphabet \( A \) with frequencies such that in an optimal Huffman coding there exist at least two encodings of length exactly \((n − 2)\), where \( n \) is the size of the alphabet. \( n \) must be at least 5.

Solution. Multiple solutions exist. One possible is:

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.6</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Represented as a tree as:

```
/ \
A / \ 
/ \ / \ 
B C D E
```
Problem 4. LZW Decoding: Decode the following LZW encoded stream

0 1 4 2 3 7 6 3 10 0 8 11

using the initial dictionary

<table>
<thead>
<tr>
<th>Encoding</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Solution. Using the LZW decompression algorithm starting from the initial dictionary, the answer is

a b a b c d c a b c d a b c d a d c d a