Compression via Huffman Coding

CMPUT 275 - Winter 2018

University of Alberta

Compression

Basic Goal

Represent a file using fewer bits, even if we have to store it in an unconventional format.

e.g. Suppose you have a text file whose characters are only `a` and `b`.

You could associate each `a` with the **bit** `0` and each `b` with the **bit** `1`.

This way, you could store the text using about $1/8$ the space of storing each ASCII character. Of course, you would have to reconstruct the original text file to view it as a plain-text file.

Benefits of Compression

- Use less memory when storing the file.
- Faster transmission of the file.

Another Example

What if the text file only has characters `a`, `b`, and `n`.

`abananaban`

What is a good compression technique?

Can use:

- `a` $\to$ 00
- `b` $\to$ 01
- `c` $\to$ 10

So

`abananaban` $\to$ 00010010001000010010

This gives a compression rate of 1/4 over a plaintext file.

Is there a better scheme to compress a text file with only three bytes?

Yes: Associate the most frequent character with `0` and the remaining two with `10` and `11`.

Compression Rate
At least $1/3$ of the characters go from 8 bits to 1 bit, the remaining characters go from 8 bits to 2 bits. This will use fewer bytes!

Compression Rate

Let n_a, n_b, n_n denote the number of occurrences of each letter and $n = n_a + n_b + n_c$.

Suppose, for example, a is the most frequent. Then $n_a \geq n/3$.

The number of **bits** used in the compressed string is

$$n_a + 2 \cdot n_b + 2 \cdot n_c = 2 \cdot n - n_a \leq 2 \cdot n - \frac{n}{3} = \frac{5}{3}n.$$

So the number of **bytes** is roughly

$$\frac{1}{8} \cdot \frac{5}{3}n \approx 0.2083 \cdot n.$$

This is better than the file size of $0.25 \cdot n$ we would get by assigning each character a 2-byte string.

Decoding

Can we decode such a file? Some characters were encoded with one bit and some with two bits. How can we decode:

```
10010011
```

where we have used the encoding

$$\mathtt{a} \to 0, \quad \mathtt{b} \to 10, \quad \mathtt{n} \to 11.$$

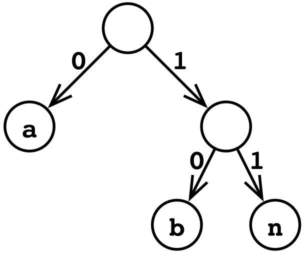
Note no bit-sequence is the prefix of another bit sequence. This will uniquely determine how to decode.

Do It!

`baban`

Decoding Tree

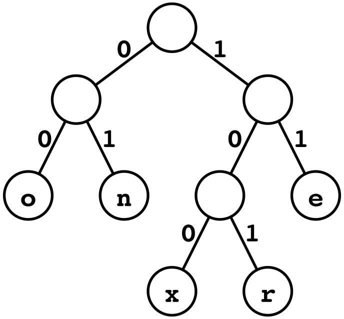
If no bit sequence is the prefix of another in our encoding, we can build a **decoding tree**.



- Each character is a leaf.
- 0/1 labels of the edges on the root-to-leaf path is the encoding of the character at the leaf.

Decode a bit sequence by using the bits to traverse the tree.

- start from the root and follow the 0/1 edge corresponding to the next bit
- output the byte at a leaf whenever you reach one, and return to the root



```
001001101 --> 00 (o) 100 (x) 11 (e) 01 (n) --> oxen
```

Building a Decoding Tree

The real art is in picking the encoding for each byte.

Requirement
No bit sequence for any character is the prefix of another bit sequence.
Note: Such an encoding scheme is called a **prefix code**.

Desire
More frequent characters have shorter bit sequences.

Concrete Optimization Problem
Construct a decoding tree to minimize the total number of bits used to compress the file.

This can be done very easily using **Huffman Trees**: trees constructed according to a simple greedy procedure.

For technical reasons, we include a special **EOF** (end of file) sentinel in our compression, even though it is not a byte of the original file.

To build a Huffman Tree, first do a frequency count of all bytes that appear in the file.

Example

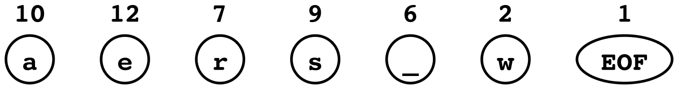
```
freq[‘a’] = 10
freq[‘e’] = 12
freq[‘r’] = 7
freq[‘s’] = 9
freq[‘ ’] = 6
freq[‘w’] = 2
freq[EOF] = 1
```

Comment

Ultimately keys will be bytes, not characters: see the code we develop.

Forming a Huffman Tree

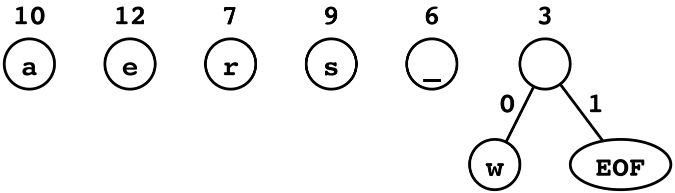
Initially each character is a (trivial) Huffman tree by itself.



Numbers: Frequency count.

Forming a Huffman Tree

Pick the two with lowest frequency count and “merge” their trees (set as children of a new node).

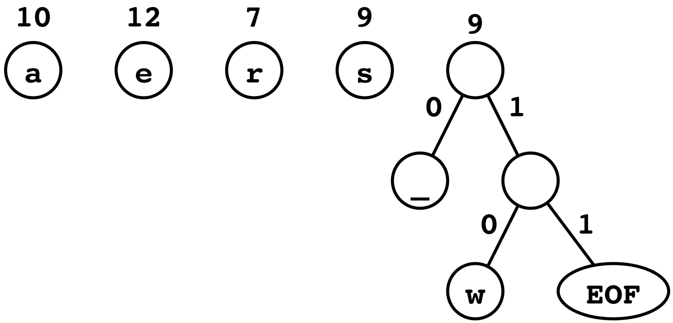


It doesn’t matter which one is the left child and which one is the right.

The number on this new tree is the total frequency count of all leaves.

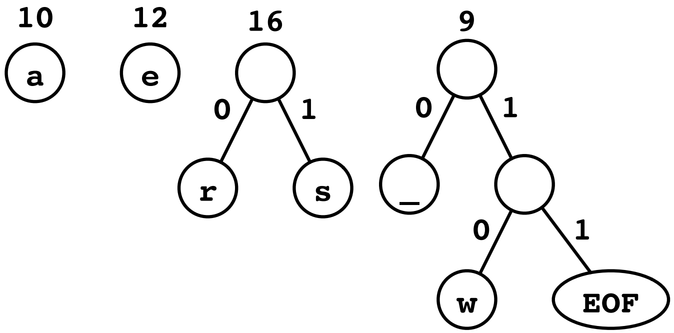
Forming a Huffman Tree

Repeat: Pick two trees with lowest total frequency count and merge.



Forming a Huffman Tree

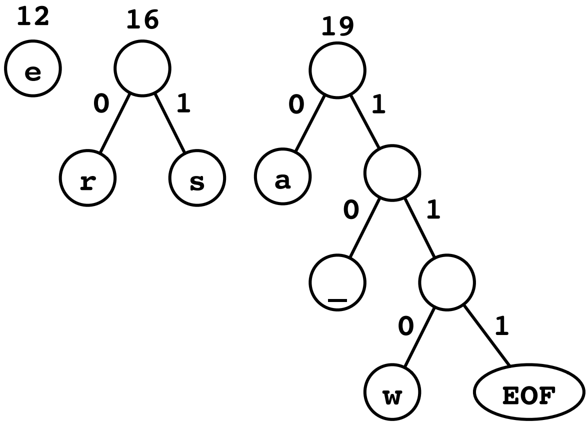
Repeat: Pick two trees with lowest total frequency count and merge.



In the case of a tie, it doesn't matter which one you pick.

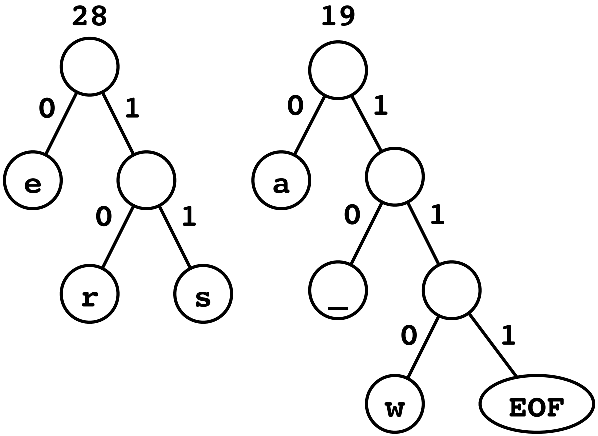
Forming a Huffman Tree

Repeat: Pick two trees with lowest total frequency count and merge.



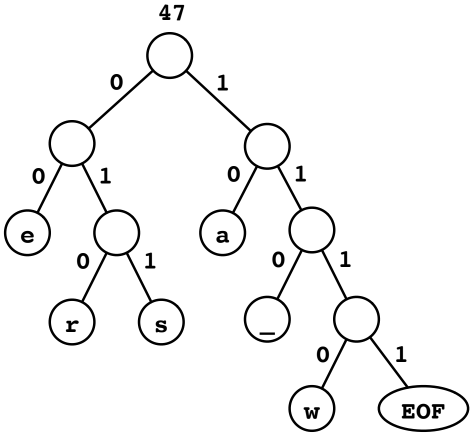
Forming a Huffman Tree

Repeat: Pick two trees with lowest total frequency count and merge.



Forming a Huffman Tree

Repeat: Done, here is our Huffman Tree!



Algorithm Summary

Initially, each character is the single node in a trivial Huffman tree.

Say the **total frequency count** of a tree is the sum of the frequencies of its leaves.

While there is more than one tree, **merge** the two whose total frequency count is smallest.

Merging trees T_1 and T_2 means creating a new node and setting T_1 and T_2 as its two children.

Running Time:

- Each merge can be implemented to run in $O(1)$ time.
- Use heaps to find the two trees with lowest frequency count!
- n iterations where n is the number of distinct bytes seen in the file: we start with $n + 1$ trivial leaf nodes (the $+1$ to account for the new EOF sentinel) and repeat until there is 1.
- So $O(n \log n)$ running time in total.

That's it!

But Why Is This Optimal?

That is, why is the total encoding length of the file using the tree constructed by this algorithm have the minimum length over all possible encoding trees?

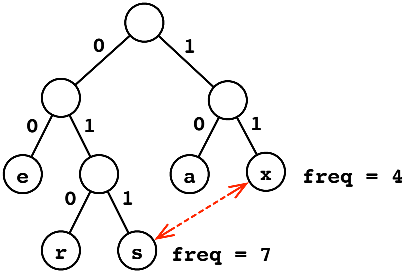
The algorithm is a **greedy** algorithm. Intuitively we want low-frequency bytes to have longer compressed bit sequences. So it seems to make sense to merge them earlier.

But that is not good enough for a proper proof. Often simple intuition leads to an incorrect greedy algorithm. So why does it work here?

Precise Idea

We need to see why the greedy algorithm does not make a mistake.

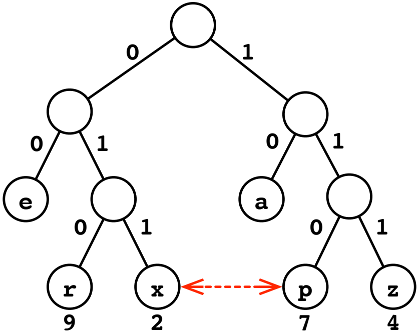
Claim: Some optimum encoding tree has the two least-frequency bytes as siblings.



To see this, we may assume they are at deepest leaves. Otherwise, swapping one with a deeper leaf will improve the encoding length.

Precise Idea

Claim: There is an optimum encoding tree where the two byte with least frequency are siblings and are deepest among all leaves.



If the two least frequent bytes are still not siblings we can move one to be a sibling of the other: the encoding length is not changed.

To Complete The Proof

The algorithm we just saw will merge the two least-frequent bytes into a Huffman tree and have them siblings of each other.

We just argued there is an optimum encoding tree that has these nodes as siblings.

Treat these siblings as a single “byte” with the new combined frequency. Arguing the same way with this smaller instance (i.e. using induction) completes the argument.

Summary

Building The Huffman Tree

- Do a frequency count of all bytes in the file (include a count of 1 for the EOF sentinel).
- Build the Huffman tree using the greedy algorithm we just described.

Summary

Compressing the File

- For each character, determine its `0/1` encoding in the compression by looking at the root-to-leaf path.
- Output the sequence of $0/1$ bits obtained by replacing each original byte by its compressed bits.
- Don't forget the final sequence for the EOF sentinel.

Summary

Decoding a File

- Starting from the root, traverse the Huffman tree. Each bit from the input sequence tells you when to go left or right.
- Once you reach the EOF leaf, quit.
- If you ever reach another leaf, output the byte, return to the root, and continue reading bits.

Considerations

We use an EOF sentinel mainly because the last byte of the compressed file might not be “complete”. Perhaps only 3 bits of it were used in the compressed file.

So decoding EOF tells us when to stop.

Obviously, to decode we have to know the Huffman tree. So when we send the compressed file we better send some representation of the Huffman tree as well.

The assignment will have you put some encoding of the Huffman tree at the start of the compressed file, followed by compressed file.

Afterthoughts

This compression is exploiting frequencies of characters. Of course, this is a fairly simple form of compression but we do see reductions in files that focus only on a subset of the 2^8 different bytes.

It tends to work best on plain text files and bitmap images with a small range of colours.

Real compression exploits other patterns and often targets specific file types (pictures, text, etc.). Some compression even allows for some data loss, such as `.jpeg` files. This is a fairly advanced topic.

Also, no compression scheme can make **every** file smaller. There are 2^n files of length n yet only $1 + 2 + 4 + 8 + 2^{n-1} = 2^n - 1$ possible files of length $< n$.