Lecture Notes for CSCI 311: Algorithms

Set 12-Greedy Algorithm Example: Huffman Codes

Professor Talmage

October 18, 2024

1 Problem

Consider the *Binary Encoding* problem:

Input: A sequence of characters S (also called a file)

Output: Minimal-length binary sequence replacing each character in S with a fixed binary string

This problem has clear applications in designing file formats for saving human-readable files on a computer, which can only read binary. There are *many* possible solutions. For example: ASCII uses 1 Byte/character, but can only represent 256 distinct characters. Unicode uses 1-4 Bytes/character, so can represent 2^{32} possible symbols, but many are unused. In general, we distinguish two types of solutions: *fixed-length representations*, allocating the same number of bits to each character and *variable-length representations*. In a fixed-length representation, if there are c distinct characters, requires $\lceil \log_2 c \rceil$ bits/character, giving a total encoding length of $|S| \log_2 c$ bits. Variable-length representations may be more efficient (we will explore them more shortly), but require slightly more work to compute the number of characters in an encoding.

Example: Suppose we have a file with 100,000 characters, each one of 6 distinct characters, with frequencies given by the following table:

Character:	a	b	c	d	e	f
Frequency (1000s):	45	13	12	16	9	5
Fixed-length:	000	001	010	011	100	101
Variable-length:	0	101	100	111	1101	1100

The fixed length encoding uses 300,000 bits, while the variable-length encoding uses $45(1) + 13(3) + 12(3) + 16(3) + 9(4) + 5(4) = 224,000$, approximately 25% fewer.

Exercise: Suppose you are given the input string "abcffedaab". Encode this file using each of the above encoding schemes.

Exercise: Decode the following strings:

- "101000010100" (fixed-length encoding)
- "10111010111" (variable-length encoding).

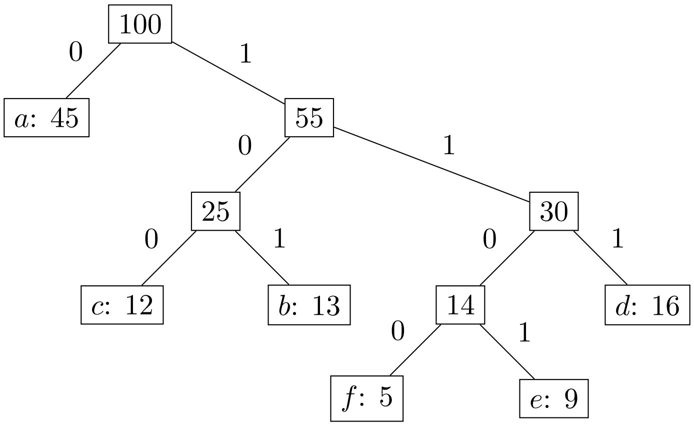
Prefix Codes:

- You may notice that the variable-length encoding above used no encodings of length 2. This makes it more feasible to decode the binary version.
- Suppose that we had used “10” to encode “b”. How can we distinguish a “b” from part of a “c”? This is why fixed-length encodings are so attractive–there is never any ambiguity about where one character ends and the next begins.
- The example above is a *prefix code*: No character’s encoding is a prefix of any other’s.
- We know (though we omit the proof) that there is always an optimal encoding which is a prefix code.

I claimed, at least implicitly, that prefix codes are easy to decode, so let us consider how to do so. Given an encoded file, we can merely read one bit at a time, since we do not know in advance where one character ends and the next begins. We want to check after each bit whether we have an entire character, and if so which one, but we do not want to do a full lookup after each bit, as that is inefficient (something like size of encoded file times size of alphabet). Instead, we can use a data structure to keep track of what characters we might be reading, and simultaneously to determine when we reach the end of a character.

1. Construct a binary tree.
2. Left branches are labeled 0, right branches are labeled 1.
3. Each character in the alphabet is a leaf, and its encoding is the binary string read on the edges from the root to that leaf.
   - Since no character is an ancestor of any other in the tree, we have a prefix code.
4. To decode, read bits from the encoded file, following the matching branch left on 0 or right on 1. When we reach a leaf, we know that we have read an entire character, and what that character is. Write that character to output and return to the root of the tree to read the next bit.
   - Complexity is the number of bits in the encoded file, since there is constant work for each. This is optimal, as we must read the entire file.
   - This type of tree structure, encoding data on paths, is known as a “trie”. (Pronounced as either “tree” or “try”.)

Example (continued): We can draw the encoding tree from our previous example:

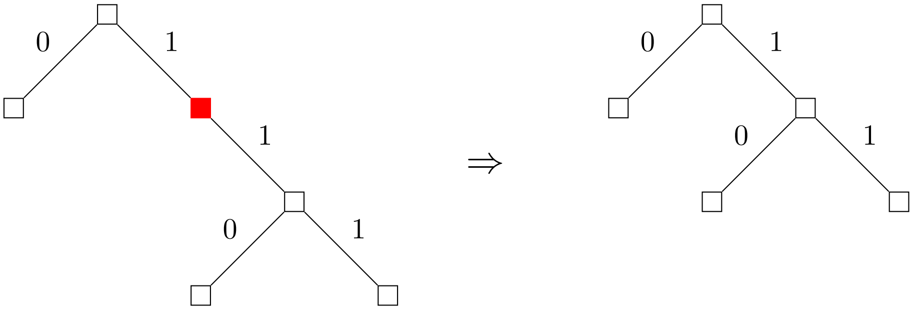


The labels of the non-leaf nodes are the frequencies of all characters with the prefix specified by the node's position. Leaves contain a character, with its frequency. We will see shortly why these frequencies are relevant.

Exercise: Use this tree to decode the string "111110110001111101".

Notes:

- Any optimal tree must be full (every internal node has exactly two children)
  - Otherwise, we could shorten the encoding for some leaf or leaves, without conflict, by cutting out a node with one child and promoting its child into its position.



- Total cost $B(S) = \sum_{c \in C} c.freq * depth(c)$, where C is the alphabet and $depth(c)$ is the number of bits in c's encoding.

Exercise: Draw the prefix tree for the following encoding: $p : 010, i : 0111, c : 00, k : 0110, l : 10, e : 11$.

2 Huffman Codes

Huffman Codes are one particular type of optimal binary encoding, created by someone named Huffman[1]. For input, we take a list of characters C, each of which has a frequency attribute indicating how many times that character appears in the document to encode. This algorithm generates the encoding tree.

```
function HUFFMAN($C$)
    $n = |C|$
    $Q = C$                                  ▷ $Q$ is a min-priority queue, keyed on frequency
    for $i = 1$ to $n - 1$ do
        allocate new tree node $newNode$
        $newNode.left = left = Q.ExtractMin()$
        $newNode.right = right = Q.ExtractMin()$
        $newNode.freq = left.freq + right.freq$
        $Q.insert(newNode)$
    end for
    return $Q.ExtractMin()$
end function
```

Runtime:

- Loop runs $n - 1$ times because it reduces the number of nodes by 1 each time (remove 2, add 1) and stops when there is a single node left, the root.
- Priority queue operations take $O(\log n)$ (heap-based), so total time is $O(n \log n)$.

[1] David A. Huffman, who created these codes for a class' final project while a doctoral student at MIT in the early 1950's. https://en.wikipedia.org/wiki/Huffman_coding

Example: Before we get into the correctness proof, let us work through the code on a couple of example inputs:

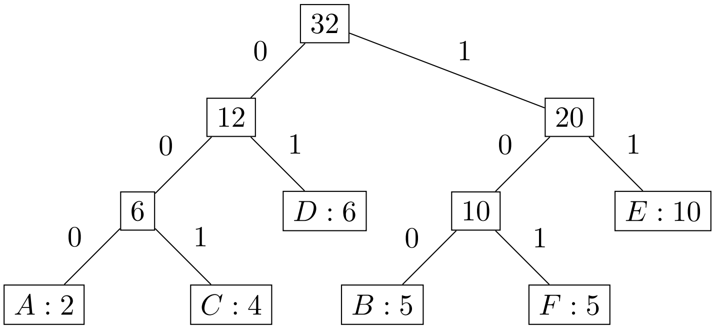
Exercise: What greedy choice does the algorithm make?

The algorithm chooses the least-frequent characters to be at the bottom of the tree, thus having the longest encodings.

Exercise: Determine Huffman codes for each of these characters by building the encoding tree.

Character:	A	B	C	D	E	F
Frequency:	2	5	4	6	10	5

1. A and C are least-frequent, replace with P, $P.freq = 6$, $Q = (P : 6, B : 5, D : 6, E : 10, F : 5)$
2. B and F are least-frequent, replace with R, $R.freq = 10$, $Q = (P : 6, D : 6, E : 10, R : 10)$
3. P and D are least-frequent, replace with S, $S.freq = 12$, $Q = (E : 10, R : 10, S : 12)$
4. E and R are least-frequent, replace with T, $T.freq = 20$, $Q = (S : 12, T : 20)$
5. S and T are least-frequent, replace with U, $U.freq = 32$, $Q = (U : 32)$
6. Return U



This gives the following encodings:

Character:	A	B	C	D	E	F
Frequency:	2	5	4	6	10	5
Encoding:	000	100	001	01	11	101

Exercise: Give Huffman encodings for the following character set:

Character:	A	B	C	D	E	F	G
Frequency:	150	270	45	46	10	300	100

Encodings: $[00, 10, 01111, 0110, 01110, 11, 010]$

Correctness: We need to argue that this greedy algorithm, choosing the least-frequent characters or groups of characters at each step and adding a bit to their encoding, yields optimal encodings. We will take advantage of the fact that some optimal encoding is a prefix code to merely prove that the algorithm gives an optimal prefix encoding, which must then be optimal overall.

Proof Outline:

- Let C be an alphabet with frequencies.
- Lemma 1 (GCP): There is an optimal prefix code in which the two least-frequent characters' encodings differ only in the last bit.
- Lemma 2 (OS): In an optimal encoding, if we expand a leaf to have two leaves as children, respecting frequencies, we have an optimal encoding for the new problem.
- Theorem: `Huffman`(C) generates an optimal prefix encoding.

Lemma 1 (Greedy Choice Property)**.** *If x and y are the characters in C with the smallest frequency values, there exists an optimal prefix code for C in which the codes for x and y are the same length and differ only in the last bit.*

Proof. Assume tree T represents some optimal solution. We will modify T to have x and y as sibling leaves. We assume WLOG that $x.freq \leq y.freq$.

Let a and b be any pair of sibling leaves of maximum depth in T. These must exist, since T is full. WLOG, assume $a.freq \leq b.freq$. Then $x.freq \leq a.freq$ and $y.freq \leq b.freq$ (convince yourself this is true). If ($x = a$ and $y = b$) or ($x = b$), which implies that $y = a$, x and y are already in the desired location, so we are done.

If not, we will convert the tree to place x and y in a and b's places, giving us a tree containing the greedy choice:

1. Exchange the nodes (and thus encodings) for a and x, moving x to a's place at max depth. Call this tree T'.
2. Ensure the total encoding cost has not increased:

$$\begin{aligned} B(T) = B(T') &= \sum_{c\in C} c.freq * d_T(c) - \sum_{c\in C} c.freq * d_{T'}(c) \\ &= \sum_{c\in C} c.freq(d_T(c) - d_{T'}(c)) \\ &= x.freq(d_T(x) - d_{T'}(x)) + a.freq(d_T(a) - d_{T'}(a)) && \text{Only } x, a \text{ change, others cancel} \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) && a, x \text{ swapped so depths equal} \\ &\geq 0 && x.freq \leq a.freq,\ d_T(x) \leq d_T(a) \end{aligned}$$

   Thus, T' is an encoding of equal or lesser total cost. Since T was optimal, T' must have equal cost.
3. Similarly, exchange y and b in the tree.

Now x and y are siblings at max depth, so they have encodings of the same length, differing only in the last bit. □

Lemma 2. *Let x and y be the characters with lowest frequencies (WLOG, $x.freq \leq y.freq$) and $C' = C \setminus \{x, y\} \cup \{z\}$ where $z.freq = x.freq + y.freq$. Let T' be* **any** *optimal prefix code tree for C'. Then replacing z's node by an internal node with x and y as children gives an optimal prefix code tree T for C.*

Proof. We can express the total encoding cost for T in terms of that for T'. For any character c not equal to x or y, $depth(c)$ has not changed. We no longer have z, but we have the same total frequency at one greater depth, so we have $B(T) = B(T') + x.freq + y.freq$, or $B(T') = B(T) - x.freq - y.freq$.

We proceed by contradiction: Assume T is not an optimal tree for C. Then there is some T^0 that is a better encoding. By Lemma 1, we can assume that x and y are max-depth siblings in T^0, WLOG. Replace x and y with a single character z where $z.freq = x.freq + y.freq$. This yields an encoding tree $T^{0'}$ for C' with total cost $B(T^{0'}) = B(T^0) - x.freq - y.freq < B(T) - x.freq - y.freq = B(T')$. But this contradicts the assumption that T' is optimal for C', so T^0 cannot exist and T is optimal. □

Theorem 1. *`Huffman`(C) generates an optimal prefix code.*

Proof. Lemma 2 shows that the problem has optimal substructure. Lemma 1 implies that the greedy choice of giving the longest encodings to the lowest-frequency characters yields part of an optimal solution. Together, this shows that making the greedy choice at each step builds an optimal solution, because there is one containing that choice and any optimal solution to the remaining subproblem can be extended in this way to an optimal solution of the whole problem. □