Compression Codes

Based on notes by Logan Mayfield

In these notes we look at a useful application of trees: compressing text.

An Application of Trees: Codes

Now we know how to talk about trees, but what do we do with them? Let's say you send a letter to your friend and it's 10,000 characters long. If this letter is encoded in ASCII or UTF-8, then each character requires 1 Byte to encode and the letter clocks in at 10kB[1]. Could we do better in terms of message size?

[1] assuming its plain text of course

One option would be to define another fixed length encoding scheme that uses fewer than 8 bits per character. If your letter uses $N <= 2^n$ characters with $n < 8$ then you could simply devise your own scheme using that number of bits. Another option would be variable length encoding. Basic engineering tells us to optimize the common case. If we could devise a code system where highly frequent characters had shorter codes then we might be able to do even better. Given that natural language character frequencies are far from uniform, we have high hopes that our letter has a really nice variable length encoding scheme.

Variable length encoding schemes have a few problems not the least of which is figuring out where one letter stops and one starts without doing some costly computation.[2]. To avoid this problem we need a *prefix free code*. In prefix free codes, no one code is a prefix for another code. For example, if 010 is the code for *a*, then 010 cannot occur at the start of any other character code. If we guarantee this property then reading encoded messages is unambiguous. If you read 0 then 1 then 0 you must have just read *a* because no other code could possibly start that way. A prefix free code could have variable code lengths for each character and seems to provide a means of solving the where one character ends and another begins problem. So, let's look at the problem of how one generates such a code before we worry about the compression part.

[2] Fixed length codes do not suffer this problem. You simply read in fixed sized increments

I'll begin with what seems like an obvious statement about binary tree paths. Take a moment to convince yourself that it must be true[3].

[3] bonus if you prove it mathematically

The path from a tree's root node to one of its leaf nodes is not a subpath for any other path in the tree.

Given that each path terminates at the leaf, then no other paths follow the leaf and the path to the leaf is obviously not a subpath for something else. Now what if paths somehow represented character codes? Then sub-paths must be prefixes and this statement is equivalent to saying:

The code represented by a path from the root to a leaf node is not the prefix of another other code.

To turn this into a workable coding system we simply place our characters in the leaves of a tree. Codes are then formed by starting at the root of the tree, reading a 0 each time we go left and a 1 when we go right[4]. When a leaf is reached then the binary code we've read is the code for that character. Let's say we had three letters: a, b, and c. One possible variable length, prefix free code for this alphabet would be represented with the tree given in figure 1.

[4] or the other way around

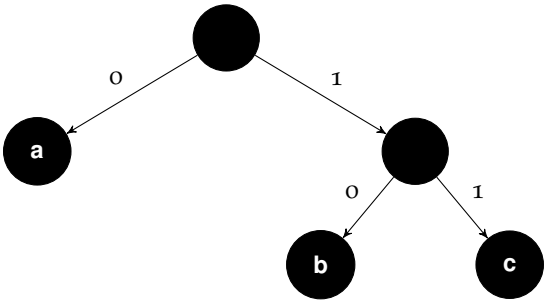

Figure 1: Code as Tree. a = 0, b = 10, c = 11.

We've now reduced prefix-free code production to a simple binary tree construction problem. The trick is to construct the *right* tree. Clearly there are lots of different trees we could construct for any given alphabet. In fact, we can even make a fixed length code from a tree. Figure 2 gives a code-as-tree encoding of a fixed length code for our a, b, and c alphabet.[5].

[5] In fact, figure 1 implies that ASCII can be represented by a tree of height 8. Do you see how?

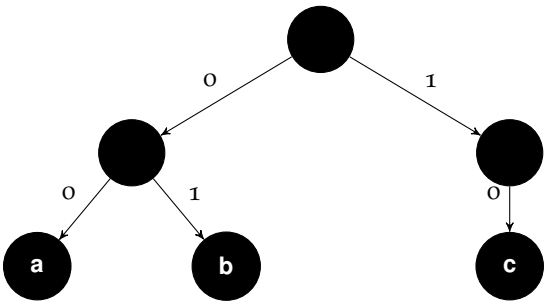

Figure 2: Code as Tree. a = 00, b = 01, c = 10.

Let's step back a second now. Fixed length codes correspond to complete or full binary trees. What about variable length prefix-free codes? It's hard to say as they seem much more flexible. However, if our goal is short code lengths then what we might be looking for is something balanced and shallow or at least just shallow.

Huffman Codes

David Huffman devised a method to construct variable length, prefix free codes that minimized the length of the encoded message and proved that his code was optimal for all such codes [6]. What Huffman did was effectively construct a tree based on the probability of each character from the alphabet occurring. For compressing a single

[6] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952

message we can swap out relative frequencies for probabilities and get maximal compression for that document.

The tree that results from Huffman's algorithm is called a *Huffman Tree*. It's pretty much the exact kind of tree-based representation of a code we looked at before but adjusted to support the algorithm for constructing it.

- Leaf nodes represent letters
- All nodes have a numerical weight associated with them. For leaf nodes, that weight is the probability/relative frequency of the letter it represents. For non-leaf nodes, the weight is the sum of the weights of its subtress.

Huffman's algorithm employs a greedy strategy. We begin with the collection of all the leaf nodes. We then repeat the following process until the collection contains a single tree: remove the two least weighted trees, construct a new tree with these threes as the subtrees, and insert that tree back into the collection. The tree that results is the Huffman tree for your code.

Let's do toy example before we go further. Table 1 lists a simple three letter alphabet and the relative frequency of each letter.

letter	frequency
a	.38
b	.05
c	.57

Table 1: A simple three letter alphabet with frequencies

The formation of a Huffman Tree for our toy alphabet is shown in figure 3. The algorithm requires two iterations of Huffman's process to complete.

The code produced by this tree is given in table 2.

letter	code
a	01
b	00
c	1

Table 2: A Huffman code for the simple three letter alphabet given in table 1

Huffman proved that his code is optimal, that it minimizes the expected length of the message it's encoding compared to any code that existed at the time and that will ever exist. What's clear is that path length is important and that the shorter the path the better. Thus, if we start understand general relationships between the number of nodes or leaves and a tree and potential tree heights we can start to get a better understand of how and why trees are useful in solving problems.

Structure and Its Implications

Looking at coding, Huffman codes, and trees has shown us that trees can help us solve problems by embedding problem logic in

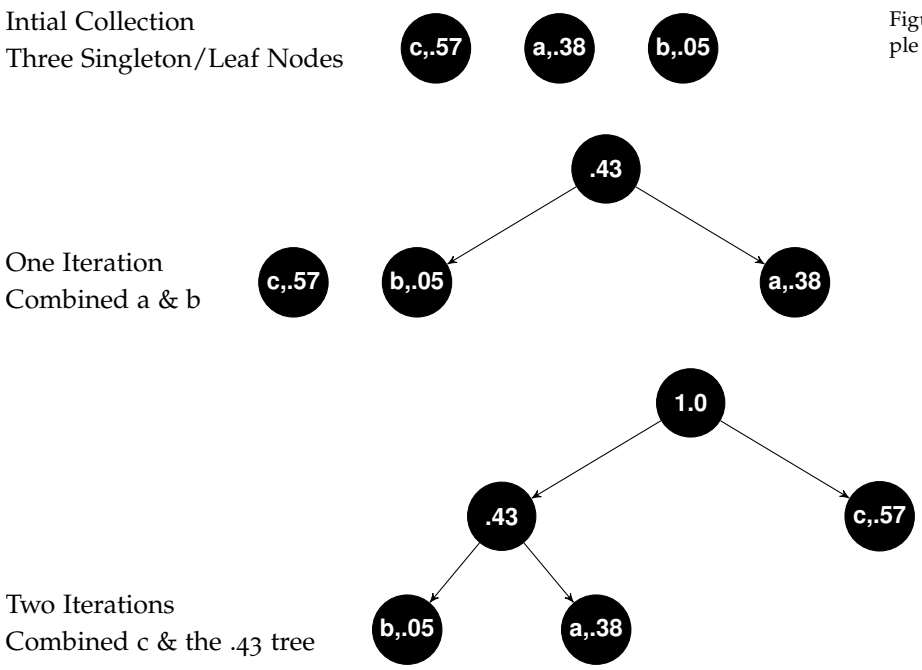

Figure 3: Huffman's Algorithm Example

the structure of a tree. It follows then that understanding certain properties of tree structures lets us understand properties of the process/problem represented by that tree. Much of what we need or want to know boils down to questions relative to size and height. For example, if you want to know the minimum length of a fixed length code for an alphabet with n symbols then you could just figure out the minimum height possible for a tree with n leaves. This leads to a series of questions about the relationships between size, number of leaves, and height.

1. What's the min height for a tree with n leaves? What's the min/-max number of internal nodes for such a tree?
2. How many nodes can a tree of height h contain? How many leaves?
3. What's the min/max path length for a tree containing n nodes? What about n leaves?

Let's begin with a constrained version of one of the more basic questions: *how many leaves are on a full tree of height h?*. This question sets some very useful bounds on the number of leaves as full trees maximize leaves relative to height[7]. To make things easy we'll start with the trivial cases and work up for a bit to find a pattern. The super-trivial case is the empty tree: there are no leaves on an empty tree nor does an empty tree have any height to speak of. A tree of height 0 is just a singleton and it is trivially a leaf. To increase the height to 1 and keep the tree full we add two children to that one node and get 2 leaves. For height 2 we add two children to each of those nodes obtaining 4 leaves. All these trees are shown in figure 4.

[7] can you see why? could you prove it?

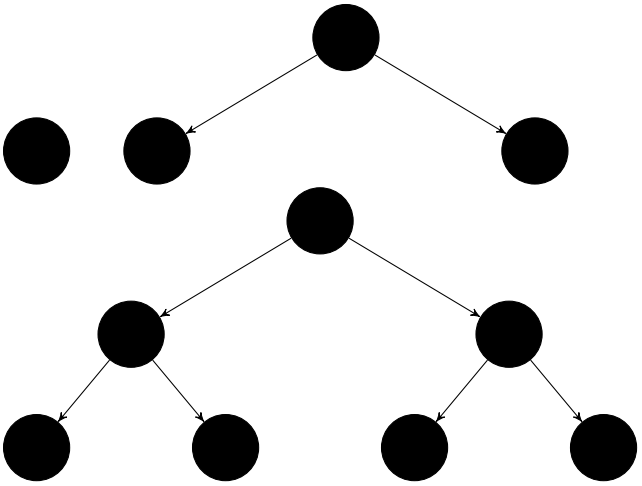

Figure 4: Full Trees of height 0, 1, and 2

The doubling of leaves is clearly going to continue and the underlying pattern is the familiar powers of 2. Now what about size? What's the size of these trees? Again, let's just count up from simple cases and see if we see a pattern. The singleton tree is obviously size 1. A full tree of height 1 is the root and two children, so 3. Next, for height 2 we add to that three 4 children for a size of 7. At height 3 we add 8 children to the 7 nodes of the height 2 full tree for a total of 15 nodes. Do you see the pattern? What if you added 1 to each of these numbers? You have $\{2, 4, 8, 16\}$. The size seems to be one less than 2^{h+1}. What you just solved intuitively is the following well proposition about a series:

Proposition 1.

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Proof. The proof follows by induction on h. For $h = 0$ we see $2^0 = 1$ and $2^1 - 1 = 1$ so the base case holds. We now prove that if $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$ for $k \geq 0$, then it will also be true for $k + 1$.

$$\begin{aligned}
\sum_{i=0}^{k+1} 2^i &= 2^{k+1} + \sum_{i=0}^{k} 2^i \\
&= 2^{k+1} + 2^{k+1} - 1 \\
&= 2(2^{k+1}) - 1 \\
&= 2^{k+2} - 1
\end{aligned}$$

This is equivalent to $2^{(k+1)+1} - 1$ and so by mathematical induction $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ for all $h \geq 0$. □

We now know some important properties about the number of leaves and total nodes for full trees. These are captured in table 3.

height	Num. Leaves	Size
empty	0	0
0	1	1
1	2	3
2	4	7
3	8	15
...	...	...
h	2^h	$2^{h+1} - 1$

Table 3: Properties of Full trees

From here we can relax our thinking a bit and look at complete trees. For a complete tree of height h there is a full tree of height $h - 1$ and one of h that give us lower and upper bounds for leaves and size. Let's just look at all the complete trees of height 2 as shown in figure 5. We exclude from this list the full tree of height 2 as we're interested in trees that are complete but not also full.

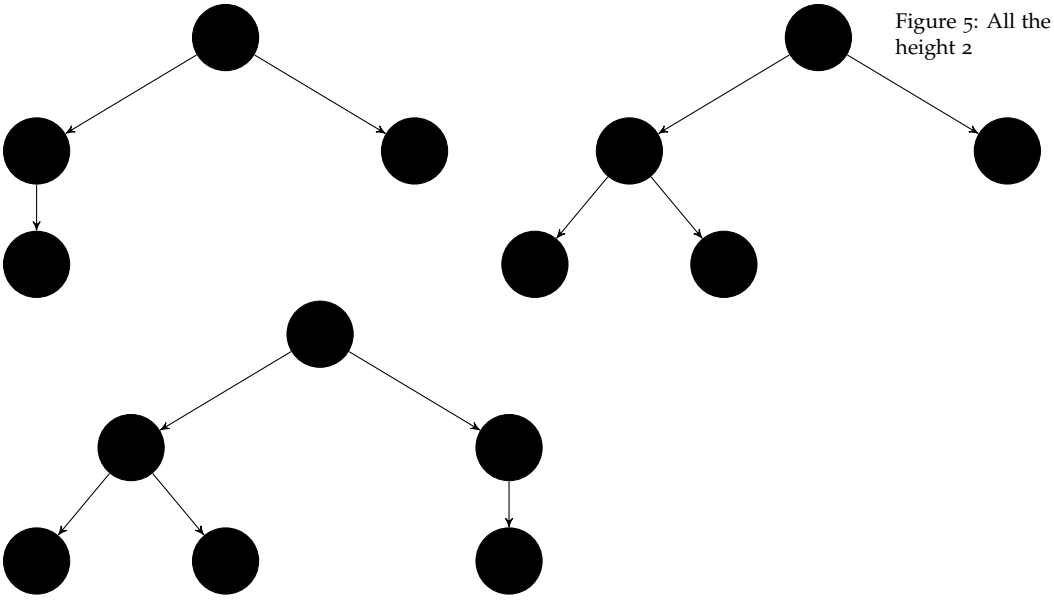

Figure 5: All the Complete Trees of height 2

First lets address the number of leaves question for complete trees. In the minimal case we seem to simply swap one leaf at h for a leaf in the full tree at $h - 1$. So the lower bound for number of leaves is 2^{h-1}. The upper bound has one fewer than the full tree at h so we're looking at $2^h - 1$. Looking at tree sizes is also straight forward. On the low end we have 1 more node than a full tree at height $h - 1$, or 2^h and on the high end one less node than a full tree at height h, or $2^{h+1} - 2$. Finally, it's worth noting that there are no binary trees of height 0 that are full but not complete.

So why all this fuss over full and complete trees? The full binary tree maximizes size and leaves while minimizing tree height. You cannot have any more nodes in a depth h tree than you do in a full tree. Complete trees do something similar but for height balanced trees. You cannot have any more nodes in a height balanced tree than you do in a complete tree that's one node shy of being full. Now

height	min leaves	max leaves	min size	max size
1	1	1	2	2
2	2	3	4	6
3	4	7	8	14
...	...	...	...	...
h	2^{h-1}	$2^h - 1$	2^h	$2^{h+1} - 2$

Table 4: Properties of Complete (but not full) tree

remember that tree height sets the upper bound on path length. It's the path length that we're often most interested in. Huffman wanted a short expected path length to maximize the compression of the message. In lots of cases we just need a short upper bound on paths. Understanding the path length properties of complete and full trees thereby lets us understand some well behaved cases.

Now we flip all of these problems around. Given a full tree of size n, what's the height of the tree. We know that n is exactly $2^{h+1} - 1$ so all we need to do is solve for h

$$\begin{aligned} 2^{h+1} - 1 &= n \\ 2^{h+1} &= n + 1 \\ \log_2(2^{h+1}) &= \log_2(n+1) \\ h + 1 &= \log_2(n+1) \\ h &= \log_2(n+1) - 1 \\ h &= O(\log n) \end{aligned}$$

It's important to note that for a full tree of n nodes the quantity $n + 1$ is an exact power of two so our heights will always come out an exact integer value. Let's just test this with a concrete example. A full tree of height 3 has $2^{3+1} - 1 = 15$ nodes and $\log_2(15+1) - 1 = \log_2(16) - 1 = 4 - 1 = 3$. When n is the number of leaves in a full tree then we know $n = 2^h$ and the height is clearly $\log_2 n$. What have we learned? The height of a full tree is on order the logarithm of its size or the number of its leaves.

What about a complete tree of size n? We know n must between 2^h and $2^{h+1} - 2$. When n is the minimum size, then the height is $\log_2 n$, but what about when it's not the minimum? We know for certain that it will not reach the next power of 2 without actually increasing the height of the tree so the simply solution is to simply take the log and round down to the nearest integer, $\lfloor \log_2 n \rfloor$.

Priority Queues, Heaps, and Huffman's Forest

If my alphabet contains n characters, then I'll need $n - 1$ iterations of Huffman's process to get the final Huffman tree. Each iteration selects the minimum tree from the forest twice, constructs a new tree, and inserts that new tree back to the collection. If we want this process to be efficient, then we need those operations to be efficient. It doesn't take much to see that tree construction is take $O(1)$ operations[8]. The real trick[9] is having efficient operations for our collection.

[8] just assign pointers for left and right along with the new root's value

[9] it often is

References

[1] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.