Huffman Coding

Miles President

May 14, 2020

Motivation

Huffman coding is a greedy algorithm used for lossless data compression, meaning that the size of the data is reduced without losing any of the information it contains. Among the many uses of data compression include fitting more information into disk storage and faster transmission of data over networks [8].

Background

The algorithm was developed in 1951 by David A. Huffman, who discovered it as a doctoral student at the Massachusetts Institute of Technology. His professor, Robert M. Fano (of Shannon-Fano coding), had assigned the problem of coming up with the most efficient way to represent bytes of data using binary codes. Little did Huffman know that this was an open problem at the time. Professor Fano was actually working on the problem himself and had yet to come up with an optimal solution [6]. Today, Huffman coding remains a backbone of many compression algorithms, including PKZIP, which produces the popular ZIP file format [3].

Description

Huffman coding works by representing each byte as sequence of bits. The number of bits used to encode the byte differs based on the frequency with which the byte appears in the data. The codes are such that no code is a prefix of any other, which makes decoding the compressed data simple. In general, this technique is known as *variable-length prefix encoding*.

The algorithm is as follows: Given a sequence of bytes, we iterate through them and keep track of the number of appearances of each byte, storing the resulting counts in a table. The set of different bytes that appear in the input is called the *alphabet* and the counts are also referred to as *frequencies*.

Next, we build the code binary tree. To begin, we construct a separate binary tree for each byte in the alphabet. Each tree consists of a single node that stores its corresponding byte, as well the byte's frequency as its value. Then, while there are two or more trees

remaining, we find the two trees with the smallest root values and create a new node whose value is the sum of these root values. We make the two trees we selected the left and right children of the new node. A visualization for the tree construction is provided in Figure 1.

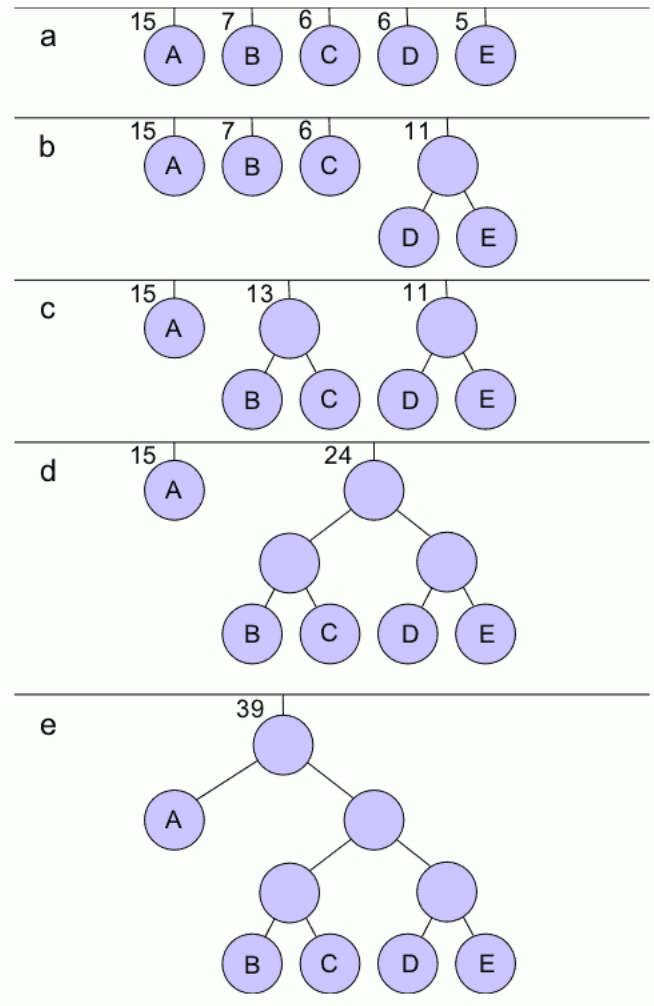

Figure 1: Construction of Huffman code tree [4]

Now, we traverse the code tree to its leaves, keeping track of each root-to-leaf path in the following way: Each time we choose a left child, we append a "0" to the path. Each time we choose a right child, we append a "1" to the path. When we reach a leaf, we store the path as a value in a table keyed by the byte that the leaf represents. The path represents the bits that will be used to encode the byte.

Finally, to compress the data, we first write our table (or the code tree itself) as bits into an output stream. Then, we iterate through the input bytes. For each byte, we consult our table to write the correct bits into the output stream and then return the output stream. Decompressing the data simply requires reading the code tree from the beginning of the byte stream. Then, we traverse the tree according to the rest of the bits. Each time we reach a leaf in the tree, we add the corresponding byte to our stream of decompressed data and restart from the root of the tree.

Next, we analyze the runtime of the algorithm. Suppose we are compressing n bytes, among which there are k different types of bytes. Calculating the frequencies takes $O(n)$ time. Building the code tree requires k iterations. In each iteration, using a priority queue allows us to extract the minimum in constant time and insert in $O(\log k)$ time. Therefore, building the tree requires $O(k \log k)$ time. The tree is binary with k leaves, so it has at most $2k - 1$ total nodes. Therefore, traversing the tree to build the table that maps bytes to bit patterns takes $O(k)$ time. The table has k entries, so writing it will take $O(k)$ time. With this table, compressing each of n bytes takes constant time, so compressing all the data takes $O(n)$ time. Altogether, the algorithm takes $O(n + k \log k)$ time. It is worth it to note that, in practice, $n \gg k$, so the complexity with respect to k (actually building the optimal code tree) is usually insignificant with regards to the total runtime.

Properties

First, we define an *optimal code tree*. An optimal code tree is one that minimizes the total number of bits used to represent the data. Let A be the alphabet of bytes. Let $d(x)$ be the distance from the root of the tree to the leaf corresponding to byte x, i.e., the number of bits it will take to represent x. Let $f(x)$ be the frequency of byte x, i.e., the number of times x appears in the input. Then an optimal code tree T minimizes the value $B(T) = \sum_{x \in A} d(x)f(x)$. We now show that Huffman coding is guaranteed to build an optimal code tree, following the proof outline given by Mordecai Golin [1].

Lemma: In any optimal code tree, every nonleaf node must have two children.

Proof. By way of contradiction, suppose we have an optimal code tree T in which an internal node p has one child c. Then, we could remove p and replace it with c to get tree T'. In T', the root-to-leaf distance of all leaves stemming from c have been descreased by one, so $B(T') < B(T)$. Since T was optimal, this is a contradiction. □

Lemma: Let x and y be the leaves corresponding to the bytes with the two smallest frequencies in the alphabet. Then there exists an optimal code tree such that x and y are sibling leaf nodes at the maximum depth of the tree.

Proof. Let T be an optimal tree. By the previous lemma, we know every leaf has a sibling, so let a and b the sibling leaf nodes at the maximum depth of the tree. Let x and y be the leaves with the two smallest frequencies in T. Then we know $d(x) \leq d(a)$ and $d(y) \leq d(b)$. We also know $f(x) \leq f(a)$ and $f(y) \leq f(b)$. Now, let T' be the tree with a and x swapped and b and y swapped. Note that in T', x and y are siblings at the maximum depth of the tree. Then, we have

$$\begin{aligned} B(T') &= B(T) - f(a)d(a) - f(x)d(x) + f(a)d(x) + f(x)d(a) \\ &\quad - f(b)d(b) - f(y)d(y) + f(b)d(y) + f(y)d(b) \\ &= B(T) - \underbrace{(d(a) - d(x))}_{\geq 0}\underbrace{(f(a) - f(x))}_{\geq 0} - \underbrace{(d(b) - d(y))}_{\geq 0}\underbrace{(f(b) - f(y))}_{\geq 0} \qquad (1) \\ &\leq B(T) \end{aligned}$$

Therefore, T' is also optimal.

□

Theorem: The Huffman code algorithm produces an optimal code tree.

Proof. The proof is by induction on n, the size of the alphabet.

Base case: Suppose $n = 1$. Huffman coding produces a tree that is just a root node. This is clearly optimal.

Inductive hypothesis: Assume that the algorithm produces an optimal code tree for any alphabet of size n.

Inductive step: Consider an alphabet A with $n + 1$ bytes. Let x and y be the bytes with the two smallest frequencies in A. By our lemma, there exists an optimal tree where the leaves corresponding to x and y are siblings at the maximum depth of the tree. Let T be such an optimal tree.

Now, let A' be the alphabet with n bytes constructed by removing x and y from A and adding z, where $f(z) = f(x) + f(y)$. Let T' be the code tree for A' created by removing x and y from T and replacing their parent with z. Noting that $d(x) = d(y) = d(z) + 1$, we have

$$\begin{aligned}
B(T) &= B(T') + f(x)d(x) + f(y)d(y) - f(z)d(z) \\
&= B(T') + f(x)d(x) + f(y)d(y) - (f(x) + f(y))(d(x) - 1) \\
&= B(T') + f(x)d(x) + f(y)d(y) - f(x)d(x) - f(y)d(x) + f(x) + f(y) \\
&= B(T') + f(x)d(x) + f(y)d(y) - f(x)d(x) - f(y)d(y) + f(x) + f(y) \\
&= B(T') + f(x) + f(y)
\end{aligned}$$

Next, consider the tree $H_{A'}$ produced by Huffman coding for A'. By our inductive hypothesis, $H_{A'}$ is optimal, so $B(H_{A'}) \leq B(T')$. Adding x and y as children of z in $H_{A'}$ gives us exactly the Huffman code tree produced for A, H_A, since $f(x) + f(y) = f(z)$. By the same reasoning as above, we have $B(H_A) = B(H_{A'}) + f(x) + f(y)$. Then, this gives us

$$\begin{aligned}
B(H_A) &= B(H_{A'}) + f(x) + f(y) \\
&\leq B(T') + f(x) + f(y) \\
&= B(T)
\end{aligned}$$

Since T was optimal, that means H_A is also optimal.

□

Limitations

The main limitation of the algorithm we have described, which is known more specifically as *static* Huffman coding, is that it requires having all the data up front in order to build

the code tree according to the frequencies. This makes it a poor technique for compressing real-time data such as that from sensors [7]. For real-time data, a variation called *adaptive* or *dynamic* Huffman coding can be used. This algorithm adjusts the code tree as it consumes data to account for changing byte frequencies. Another drawback of Huffman coding is that its compression ratio often falls short of some other compression algorithms, such as arithmetic coding [5] [2].

References

[1] Mordecai Golin. Huffman encoding and data compression, 2003. `http://home.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL17.pdf`.

[2] Anmol Jyot Maan. Analysis and comparison of algorithms for lossless data compression. *International Journal of Information and Computation Technology*, 3(3):139–146, 2013.

[3] PKWARE. *PKZIP 6.0 Command Line Users Manual*, 2002.

[4] Andreas Roever. HuffmanCodeAlg, 2004. `https://commons.wikimedia.org/w/index.php?curid=3000007`.

[5] Asadollah Shahbahrami, Ramin Bahrampour, and Mobin Sabbaghi Rostami. Evaluation of huffman and arithmetic algorithms for multimedia compression standards. *International Journal of Information and Computation Technology*, 1(4), 2011.

[6] Gary Stix. Profile: David A. Huffman. *Scientific American*, 265(3):54–58, September 1991.

[7] C. Tharini and P. Vanaja Ranjan. Design of modified adaptive huffman data compression algorithm for wireless sensor network. *Journal of Computer Science*, 5(6):466–470, 2009.

[8] Julie Zelenski and Keith Schwarz. Huffman encoding and data compression, May 2012. `https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/handouts/220%20Huffman%20Encoding.pdf`.