Implementing Various Codes

Justin Koo

December 10, 2018

Introduction

This paper is intended to provide a guide for implementing a small set of codes using pseudo-code. It is important to distinguish between *code* and *pseudo-code*, for they have no relation to each other in the context of this paper. When we use the term *code*, we refer to a system of rules and algorithms for transforming information from one representation to another. When we use the term *pseudo-code*, we refer to the simplified notation of describing an algorithm. Pseudo-code is used primarily to present an algorithm to a general audience by stripping away the syntactical nuances of a particular programming language. **Algorithm 1** showcases an example of the style of pseudo-code we will use throughout this paper. It describes an algorithm to find the minimum value among a set of numbers.

Algorithm 1 Obtain the minimum value from a set of values V

```
1: procedure GET-MINIMUM(V)
2:     m ← ∞
3:     for each value v ∈ V do
4:         if v < m then
5:             m ← v
6:     return m
```

Within each of the algorithms presented throughout, boldface words represent keywords or flow-of-control structures that are present in most general purpose programming languages; it is expected that the reader has a good understanding of them.

Huffman Codes

A way to store data efficiently is to use Huffman Codes. To give a brief overview, Huffman Codes assign each source word to a unique binary encoding by constructing the Huffman tree. As a result no encoding is a prefix of another encoding, thereby minimizing the length of the encoding. The two major steps of the encoding algorithm include 1.) constructing the Huffman tree and 2.) obtaining the encoding of each source word using the constructed tree. **Algorithm 2** [1] performs step 1.) by constructing the Huffman tree given a list of source characters C and their relative frequencies. The procedure returns the root of the tree.

In line 2 we create a priority queue using the characters in C. The priority queue is simply a data structure that allows us to extract the Node or character with the lowest frequency efficiently. The **while**-loop of lines 3-10, performs the tree construction. First we allocate a new Node and assign its left and right children to be the two nodes with the lowest frequency obtained in lines 5-6. We then set the frequency of this new Node to be the sum of the frequencies of its children. Finally we insert the newly constructed Node back into the priority queue. Notice that we iterate until the priority queue has just one Node within. This node is the root of the desired Huffman tree. Iteration is guaranteed to terminate since the number of elements in the priority queue decreases by one each iteration. This is because during each iteration, we remove two elements and add a new one.

Algorithm 2 Constructing a Huffman Tree from a set of characters C

```
1:  procedure CONSTRUCT-TREE(C)
2:      Q ← a new minimum priority queue with characters in C
3:      while |Q| > 1 do
4:          z ← a new Node
5:          x ← EXTRACT-MIN(Q)
6:          y ← EXTRACT-MIN(Q)
7:          z.left ← x
8:          z.right ← y
9:          z.freq ← x.freq + y.freq
10:         INSERT(Q, z)
11:     return Extract-Min(Q)
```

Now that we have constructed the Huffman tree, it remains to obtain the encoding of each character. We do this by performing a recursive depth-first traversal of the tree. Along the traversal, we store the traversal path so that when we reach a leaf-node, which will be a character in C, we may obtain its encoding by looking at the traversal path taken up to that point. **Algorithm 3** performs a depth-first tree traversal and stores each character encoding in an auxiliary Map data structure.

Algorithm 3 Obtaining the encoding using a Huffman Tree. r is the root of the tree and p is the path taken to get to r.

```
1:  procedure GET-ENCODING(r, p)
2:      if r = NIL then
3:          return
4:      if r.left = NIL AND r.right = NIL then
5:          Add the key-value pair (r.value, p) to an auxiliary Map.
6:      if r.left ≠ NIL then
7:          GET-ENCODING(r.left, p + "0")
8:      if r.right ≠ NIL then
9:          GET-ENCODING(r.right, p + "1")
10:     return
```

In line 2 we check if we have traversed to a leaf node. If that is the case, we add the node's value (a character in this case) and the binary string representing the path to reach the node as a key-value pair in the Map. Line 5 traverses the left sub-tree of the root recursively as indicated to the recursive call to GET-ENCODING. We append the character "0" to the path to indicate that our traversal is going left. Likewise, line 7 traverses the right sub-tree recursively and appends the character "1" to the path to indicate our traversal is going right. In the initial call to the GET-ENCODING procedure, r should be the root of the tree returned by the CONSTRUCT-TREE procedure, and p should be an empty string. Once the GET-ENCODING procedure terminates, the auxiliary Map will contain key-value pairs containing the characters and their encoding respectively.

In order to decode Huffman Codes, we simply traverse the constructed tree according to the binary "path" the encoding specifies. When we reach a leaf node in our traversal, we append the character to the decoding and repeat the process. **Algorithm 4** outlines such a procedure.

The **for**-loop of lines 5-12 perform the decoding of each of the characters in d. Within each iteration, we go either left or right in our traversal depending on the digit in the encoding. Only when we reach a leaf node do we append the decoded character and reset the traversal back to the root.

Algorithm 4 Decoding a binary string d representing a sequence of encoded characters using a Huffman tree with root r.

```
1:  procedure DECODE-HUFFMAN(r, d)
2:      s ← an empty string
3:      t ← r
4:      n ← d.length
5:      for i = 1 to n do
6:          if d[i] = '0' then
7:              t ← t.left
8:          else if d[i] = '1' then
9:              t ← t.right
10:         if t.left = NIL AND t.right = NIL then
11:             s ← s + t.value
12:             t ← r
13:     return s
```

Repetition Codes

One of the simplest and most intuitive ways to guard against errors during transmission of information is to repeat the transmission and add redundancy. In the binary case, repetition codes repeat each binary digit sent a specified number of times in hopes of being able to correct any errors in the received message. Algorithm 4 describes a procedure to transform a string of binary data d, repeating each digit r times, all while having an error probability e in any of the transmitted digits.

Algorithm 5 Transforming data d using a repetition code with repetition parameter r and error probability e

```
1:  procedure REPETITION-CODE(d, r, e)
2:      n ← d.length
3:      td ← an empty string to hold the transformed data
4:      for i = 1 to n do
5:          for j = 1 to r do
6:              k ← a random number in [0, 1]
7:              if k ≤ e then
8:                  td ← td + FLIP(d[i])
9:              else
10:                 td ← td + d[i]
11:     return td
```

The FLIP() procedure simply returns the opposite digit of its argument. We can decode the output of the REPETITION-CODE() procedure by decoding each "block" of supposedly repeated digits to be the digit that has the highest occurrence in the "block". **Algorithm 5** describes such a procedure.

Algorithm 6 Decoding transformed data td by examining each "block" of size r

```
1:  procedure DECODE-REPETITION-CODE(td, r)
2:      d ← an empty string to hold the decoded data
3:      for each block b of r digits in td do
4:          c ← the most frequent digit in b
5:          d ← d + c
6:      return d
```

The Hamming [7,4] Code

Recall that the Hamming [7,4] code is a specific type of linear code with an alphabet over $\mathbf{F}_2$ and the generating 4x7 matrix G (not in standard form) with entries in $\mathbf{F}_2$ is:

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

[2]. Like repetition codes, the Hamming [7,4] code adds redundancy to the transmission in the form of 3 parity bits to guard against possible errors. To encode a 4-bit binary string s into its 7-bit encoding, we simply left multiply G by s over $Z/2$. **Algorithm 7** describes this short procedure.

Algorithm 7 Encoding a 4-bit binary string s using the Hamming [7,4] code

```
1: procedure ENCODE-HAMMING-7-4(s)
2:     G ← the generating matrix for the Hamming [7,4] code
3:     return s × G over Z/2
```

For the special case of the Hamming [7,4] code, we can decode a received 7-bit encoding y that has at most 1 error. This requires the use of the parity-check matrix H (not in standard form) of the Hamming [7,4] code:

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

[2]. The procedure is as follows: when $c = y \times H^T$ yields the zero-vector, then no error has occurred. When $c = y \times H^T$ yields a non-zero vector, then c^T is the binary representation of the location in which the error occurred. **Algorithm 8** describes this decoding procedure.

Algorithm 8 Decoding a 7-bit binary string y with at most 1 error.

```
1: procedure DECODE-HAMMING-7-4(y)
2:     H ← the parity-check matrix of the Hamming [7,4] code
3:     c ← y × H^T
4:     if c is not the zero vector then
5:         l ← the base-10 representation of c^T
6:         y[l] ← FLIP(y[l])
7:     return y
```

Linear Codes

In this section, we describe the method of *syndrome decoding* that can be applied to any linear [*n,k*] code (including the Hamming [7,4] code). Our motivation for using a different linear code other than the Hamming [7,4] code is to be able to correct more than 1 error. In general, a linear [*n,k*] code has associated with it a $k \times n$ generating matrix G (in standard form) with entries over some finite field $\mathbf{F}_q$ and whose rows are linearly independent [2]. If these conditions are satisfied, then encoding a length k input vector s to a length n output vector c is simply the task of left multiplying G by s over $\mathbf{F}_q$. One can easily modify **Algorithm 7** to achieve this result.

The task of decoding a general linear [*n,k*] code is different than decoding the Hamming [7,4] code since we expect that more than 1 error can occur. As a result, **Algorithm 8** fails. To find a different decoding

procedure that can correct more than 1 error, we turn to the method of syndrome decoding. Recall that a coset of a code C is a subset of $\mathbf{F}_q^n$ of the form (with fixed $v_o \in \mathbf{F}_q^n$):

$$\{v_o + v \mid v \in C\} = v_o + C$$

and the coset leader is the element in the coset with the smallest Hamming weight [2]. Take for example the linear [5,3] code over $\mathbf{F}_2$ with generating matrix G:

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The codewords C generated by G are all linear combinations of the rows of G over $\mathbf{F}_2$:

$$C = \{00000, 10010, 01011, 00111, 11001, 10101, 01100, 11110\}$$

The cosets and their respective coset leaders are shown in the following table:

v_o	$v_o + C$	Coset Leader
00000	$\{00000, 10010, 01011, 00111, 11001, 10101, 01100, 11110\}$	00000
00001	$\{00001, 10011, 01010, 00110, 11000, 10100, 01100, 11111\}$	00001
00010	$\{00010, 10000, 01001, 00101, 11011, 10111, 01110, 11100\}$	00010
00100	$\{00100, 10110, 01111, 00011, 11101, 10001, 01000, 11010\}$	00100

Suppose we receive the word $y = 11101 \notin C$. To decode we first find the coset x that y belongs to. In this case x is the last coset. Then we subtract the coset leader of x from y to obtain the decoding. For this case we get $11101 - 00100 = 11001$. It should be noted that this example of a linear code can never correct more than 1 error, but other codes can be constructed that can. It should also be noted that whenever there are multiple candidates for a coset leader, we are left in an ambiguous decoding situation.

The decoding procedure for the example above is generalized in **Algorithm 9**. The **for**-loop of lines 3-5 searches for the coset x that y belongs to. We then decode y as the difference between y and the coset leader of x. While this procedure is simple enough, the major preparation of it comes in generating the set of cosets S and computing each of their coset leaders. Ideally, this should be only done once and the results should be stored in some sort of look-up table to avoid repeated computation.

Algorithm 9 Decoding a length n string y to some code word $c \in C$ using the set of cosets S

```
1: procedure DECODE-LINEAR-CODE(y, S)
2:     x ← and empty set
3:     for each coset s ∈ S do
4:         if y ∈ s then
5:             x ← s
6:     return y − x.coset-leader
```

Conclusion

This paper was intended to outline how might we implement several coding schemes programmatically. We have given instructions, in the form of pseudo-code, on how to encode and decode Huffman and linear codes (including repetition codes and the Hamming [7,4] code). These codes represent only but a small fraction of possible coding schemes; however, taken as a whole, they address the primary issues of transmitting data both *efficiently* and *accurately*. Huffman codes seek to maximize efficiency by minimizing the length of transmitted data. Linear codes address accuracy by adding redundancy in the transmission. It is the hope that this paper demonstrates how performing encoding or decoding procedures, menial tasks to do by hand, are the perfect task for a computer to perform.

Bibliography

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[2] P.B. Garrett. *The Mathematics of Coding Theory: Information, Compression, Error Correction, and Finite Fields*. Prentice Hall, 2004. ISBN: 9780131019676. URL: https://books.google.com/books?id=sE6dQAAACAAJ.