M of N Code

Joaquim Espada

Data transmission

Login : con0004

15 November 2016

Summary

Introduction

This assignment is carried out within the scope of subject data transmission. In this assignment I will explain the theoretical part of the algorithm M of n Code and also known as Constant Weight Code.

M of n Code is an error detection code algorithm. This kind of algorithm allows the detection of errors in data transmission over unreliable or noisy communication channels. The main idea is that the sender encodes the message in a redundant way by adding a sequence of bits to the original message and the receiver checks the redundant data and determines if has happened an transmission error or not.

In the last part of this assignment will be an implementation of this algorithm using octave programming language.

M of n Code

The m of n code is a separable error detection code with a code word length of n bits, where each code word contains exactly m instances of a "ones" and also all codewords share the same Hamming weight. This algorithm can't correct errors.

Hamming Weight

The Hamming weight of a string is the number of symbols that are different from the zero-symbol of the alphabet used.

Example :

Alphabet used	Zero-symbol	String	Hamming Weight
Binary - “1” or “0”	0	11101	4
Binary - “1” or “0”	1	11101000	4
Decimal – “0” until “9”	0	789012340567	10

Encoding Message

The simplest implementation is to append a string of ones to the original data until it contains m ones, then append zeros to create a code of length n.

Example :

3 of 6 Code

3 → number of “ones” instances

6 → length of the codeword

Original 3 data bits	Appended bits	Encoded Message
000	111	000 111
001	110	001 110
010	110	010 110
100	110	100 110
101	100	101 100

110	100	110 100
111	000	111 000

Decoding

The easiest way to decode a received message is only to keep the original data bits and discard the appended bits. The size of the original message may be calculated by the following formula:

$$Original\,data\,lenght = n - m$$

So we should only read the first x bits provided formula and discard the rest.

Note: This algorithm is only valid if the bits are appended in the final of the original message.

Example :

3 of 6 Code

$$Original\,data\,lenght = 6 - 3 = 3$$

Encoded Message	Discarded Bits	Original 3 data bits
000 111	111	000
001 110	110	001
010 110	110	010
100 110	110	100
101 100	100	101
110 100	100	110
111 000	000	111

Detecting Error

Regarding the error detection a single bit error will cause the code word to have either m + 1 or m – 1 "ones".

m + 1 case :
Original message : 000 111
Received message : 100 111
Error found!

m - 1 case :
Original message : 000 111
Received message : 000 110
Error found!

Undetected Errors

Original message : 101 100

Received message : 001 101

Observation: If two bits with opposite value are affected by noise and consequently flip their values, this algorithm will not find a the error.

Theoretical Error Correction Algorithm

Has previously stated this algorithm can't correct errors only spot them. So the way that I found to correct the transmission errors is the following the receiver has a list where are stored all possible messages to received. Then is the hamming distance is calculated with the received message and the messages stored in the list. The message stored in the list associated with the computation with the lowest hamming distance value will be the correct message. If more than one computation of hamming distance has the same the message must be retransmitted.

Theoretical Algorithm Restrain

In this algorithm all the codewords generated by the M of N code must share an minimal Hamming Distance of three to correct one error.

Hamming Distance

Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different.

	000	001	010	011	100	101	110	111
000	0	1	1	2	1	2	2	3
001	1	0	2	1	2	1	3	2
010	1	2	0	1	2	3	1	2
011	2	1	1	0	3	2	2	1
100	1	2	2	3	0	1	1	2
101	2	1	3	2	1	0	2	1
110	2	3	1	2	1	2	0	1
111	3	2	2	1	2	1	1	0

Codewords for error correction

	000 111	001 110	010 101	011 100	100 011	101 010	110 001	111 000
000 111	0	2	3	4	2	2	4	6
001 110	2	0	4	2	4	2	6	4
010 101	2	4	0	2	4	6	2	3
011 100	4	2	2	0	6	4	3	2
100 011	2	4	4	6	0	2	2	4
101 010	4	2	6	4	2	0	4	2
110 001	4	6	2	4	2	4	0	2
111 000	6	4	4	2	4	2	2	0

Example:

Received Message with error 1: 001 111

Received Message with error 2: 100 001

Codewords	Hamming Distance message 1	Hamming Distance message 2
000 111	1	2
010 101	3	4
011 100	3	6
110 001	5	2
111 000	5	2

After computing the Hamming Distance between the received message with error 1 the codewords stored in the list of possible messages we can see that the corrected message is “001 111”.

After doing the same to the second message the algorithm wasn't able to correct the error so the message must the retransmitted.

Algorithm Implementation

The algorithm implementation was made using Octave. Octave is a high-level interpreted language, primarily intended for numerical computations. It provides capabilities for the numerical solution of linear and nonlinear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation. The Octave language is quite similar to Matlab so that most programs are easily portable.

File run.m

The purpose of the run file is to demonstrate how the algorithm works with several examples.

```
function run()
 m = 3
 n = 6
 printf("Enconding Messages\n")
 original_data = [0 0 0; 0 0 1; 0 1 0; 1 0 0; 1 0 1; 1 1 0; 1 1 1]
  for i = 1: size(original_data)
   temp = original_data(i,:)
   encode_message(temp,m,n)
  endfor

 printf("Decode Messages\n")
 original_data = [0 0 0 1 1 1; 0 0 1 1 1 0; 0 1 0 1 1 0;
 1 0 0 1 1 0; 1 0 1 1 0 0; 1 1 0 1 0 0; ]
  for i = 1: size(original_data)
   temp = original_data(i,:)
   decode_message(temp,m,n)
  endfor

   printf("\n Sending Messages with errors \n")
   messages_with_error = [1 0 0 1 1 1; 0 0 1 1 0 0;
```

```
                0 0 0 1 1 0; 0 0 0 1 0 0;
                0 0 1 1 1 1; 1 1 1 1 0 1; 1 1 1 0 0 1;
                0 0 0 1 1 1; 1 1 1 0 0 0; 0 0 1 1 1 0]
  for i = 1: size(messages_with_error)
   temp = messages_with_error(i,:)
   error_dectetion(temp,m,n)
  endfor
    dict = [0 0 0 1 1 1; 0 1 0 1 0 1; 0 1 1 1 0 0;1 1 0 0 0 1; 1 1 1 0 0 0]
     message_to_correct = [0 0 1 1 1 1; 1 0 0 0 0 1]

  for i = 1: size(message_to_correct)
   temp = message_to_correct(i,:)
 remove_error(temp,dict)
  endfor
endfunction
```

File encode_message.m

This file encodes messages accordingly to the Encode Messages section. Dependency on file ones.m.

```
# message - array with the message
# n - numbers of bits supposed to exist
# m - number of 1 bits supposed to exist
function new_message = encode_message(message,m,n)
 n_ones = ones(message)
 new_message = []
 #copy original message
 for i = 1: length(message)
```

```
  new_message(i) = message(i)
 endfor
 # end of copy
 counter = length(message)+1
 while(n_ones < m || counter <= n)
  if(n_ones < m)
   new_message(counter) = 1
  else
   new_message(counter) = 0
  endif
  counter++
  n_ones++
 endwhile
 printf("\nDecoded Message : \n")
 decode_message(new_message,m,n)
endfunction
```

File decode_message.m

The function on this file decodes messages.

```
# message - array with the message
# n - numbers of bits supposed to exist
# m - number of 1 bits supposed to exist
function decode_message(message,m,n)
 new_message = []
 x = n - m #x = original data size
 for i = 1: x
  new_message(i) = message(i)
 endfor
```

```
endfunction
```

File ones.m

The function on this file counts the number of ones present in one message. This function is used in the files encode_message.m , error_dectetion.m as an auxiliary function.

```
function retval = ones(message)
  retval = 0
  for i = 1 : length(message)
    if(message(i) == 1)
      retval++
    endif
  endfor
endfunction
```

File error_dectetion.m

The function in this file finds errors based in the number of “1” instances regarding the specifications of n and m parameters. Dependency on file ones.m.

```
# message - array containing the message in binary
# n - numbers of bits suposed to exist
# m - number of 1 bits supposed to exist
function error_dectetion(message,m,n)
  n_ones = ones(message)
  if(n_ones == m)
     printf("\n No error \n")
  elseif(n_ones < m)
        printf("Case: m - %d \n",(m-n_ones))
  elseif(n_ones > m)
        printf("Case: m + %d \n",(n_ones-m))
   endif
```

```
  printf("\n")
endfunction
```

File remove_error.m

Function responsible to remove errors. This function takes as parameters the message to be analyzed for error and the list with code words.

```
function message = remove_error(m,dict)
 c = 20000000000000
 counter = 1
 retransmission_sensor = 0
 message = []
 for i= 1:rows(dict)
  dict_temp = dict(i,:)
  printf("dict_temp -> %d \n", length(dict_temp))
  temp = hamming_distance(m,dict_temp)
  if(temp == c)
    printf("\nRestransmission\n")
    retransmission_sensor = 1
   break
 endif
  if(temp < c)
   c = temp
   message = dict_temp
   counter = counter + 1
  endif
 endfor
 if(retransmission_sensor ==1)
   printf("\nRestransmission\n")
```

```
  else
    printf("input message")
    m
    message
  endif
endfunction
```

File hamming_distance.m

The function in this file computes the hamming distance between to messages.

```
# m1 message 1 in binary
# m2 message 2 in binary
# this functions returns the hamming distance value between two
#messages
function d = hamming_distance(m1,m2)
  d = 0
  printf("m1 -> %d \n", length(m1))
  printf("m2 -> %d \n", length(m2))
  if(length(m1)==length(m2))
    for i=1:length(m1)
      if(m1(i) != m2(i))
        d++
      endif
    endfor
  else
    printf("The messages must have the same lenght \n")
  endif
   d
endfunction
```

Conclusion

After concluding this assignment I concluded that the original M of N Code is can only be used for error detection.

This algorithm can find error by counting the number of ones instances in the message. There are two main cases of error defection, m + 1 case and m – 1. In the first the message has one more instance of one that was supposed to have. In the second case there is a one missing. This error detection algorithm isn't perfect because if two bits are flipped in the message the error will not the found.

Regarding the fact that this algorithm couldn't correct errors I tried develop one. The main idea is that the receiver holds a list where are stored all possible messages to received. Then is computed the hamming distance between the received message and the messages within the list, the message associated with the lowest computation is the correction of the message if there are more than one equal value the message must be retransmitted. The drawbacks of this algorithm is that all codeword must share a minimal hamming distance of three.

About the implementation of the algorithm, the code was written in high-level interpreted language Octave. While coding I had some difficulties because I never had used a mathematical language before, I also used few Octave files with function that allows functions to be reused. In the end I think that code is fairly good and works well.

References

https://en.wikipedia.org/wiki/Hamming_distance

https://en.wikipedia.org/wiki/Hamming_weight

https://en.wikipedia.org/wiki/Constant-weight_code

https://books.google.de/books?id=8kSjBQAAQBAJ&pg=PA107&dq=m+of+n+code&hl=pt-PT&sa=X&redir_esc=y#v=onepage&q=m%20of%20n%20code&f=false

http://wiki.octave.org/GNU_Octave_Wiki