PSG COLLEGE OF TECHNOLOGY

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES

COMPUTER NETWORKS

PROBLEM SHEET 6– HAMMING CODE

The purpose of this lab is for you to gain familiarity with Hamming codes. Write a python program that will implement a Hamming code to accomplish error detection and correction. Give your program true input and input that needs to be corrected, with the bits in error being anywhere throughout the data (including in those bits added by the Hamming code algorithm). Show that your program functions correctly for the given task.

Task 1. Use the Hamming code to calculate the code word for 10011010.

```
#generating hamming code for signal
def hammingcode():
        d=input("Enter the signal: ")
        data=list(d)
        data.reverse()
        r=0
        c=0
        h=[]
        j=0
        ch=0
        while(len(d)+r+1)>2**r:
        r=r+1
        for i in range(0,r+len(d)):
                p=2**c
                if(p==i-1):
                        h.append(0)
                        c=c+1
                else:
                        h.append(int(data[j]))
                        j=j+1
        for parity in range(0, len(h)):
                ph = (2**ch)
                if ph==parity+1:
                        startIndex = ph-1
                        i=startIndex
                        toXOR = []
                while i<len(h):
                        block=h[i:i+ph]
                        toXOR.extend(block)
                        i=i+2*ph
                for z in range(1,len(toXOR)):
                        h[startIndex]=h[startIndex]^toXOR[z]
                ch+=1
        h.reverse()
        print("Hamming code =", h)
#main
hammingcode()
```

Task 2. Use the Hamming code to calculate the code word for 10011010. Assume that instead of 011100101010 the receiver got 011100101110. Show how the receiver can calculate which bit was wrong and correct it.

```
#finding the error bit
def errorCorrection():
        print('Enter the hamming code received')
        d=input()
        data=list(d)
        data.reverse()
        c,ch,error,h,parity_list,h_copy=0,0,0,[],[],[]

        for k in range(0,len(data)):
        p=(2**c)
        h.append(int(data[k]))
        h_copy.append(data[k])
        if(p==(k+1)):
        c=c+1

        for parity in range(0,(len(h))):
        ph=(2**ch)
        if(ph==(parity+1)):

        startIndex=ph-1
        i=startIndex
        toXor=[]

        while(i<len(h)):
        block=h[i:i+ph]
        toXor.extend(block)
        i+=2*ph

        for z in range(1,len(toXor)):
        h[startIndex]=h[startIndex]^toXor[z]
        parity_list.append(h[parity])
        ch+=1
        parity_list.reverse()
        error=sum(int(parity_list) * (2 ** i) for i, parity_list in enumerate(parity_list[::-1]))

        if((error)==0):
        print('There is no error in the hamming code received')

        else:
        print('Error is in',error,'bit')

        #correcting the error
        if(h_copy[error-1]=='0'):
        h_copy[error-1]='1'

        elif(h_copy[error-1]=='1'):
        h_copy[error-1]='0'
        print('After correction hamming code is:- ')
        print(h_copy)
```

#main
errorCorrection()

Task 3. Test if code word 010101100011 is correct, assuming it was created using an even parity Hamming code. If it is incorrect, indicate what the correct code word should have been. Finally, indicate what the original data was.

The working of Hamming code is given below for your reference.

Hamming Code

In 1950, Richard Hamming developed an innovative way of adding bits to a number in such a way that transmission errors involving no more than a single bit could be detected and corrected.

The number of parity bits depends on the number of data bits:

Data Bits :	**4**	**8**	**16**	**32**	**64**	**128**
Parity Bits:	**3**	**4**	**5**	**6**	**7**	**8**
Codeword :	**7**	**12**	**21**	**38**	**71**	**136**

For a data of size 2^n bits, n+1 parity bits are embedded to form the codeword. It's interesting to note that doubling the number of data bits results in the addition of only 1 more data bit. Of course, the longer the codeword, the greater the chance that more than error might occur.

Placing the Parity Bits

We will number the bits from left to right, beginning with 1. In other words, bit 1 is the most significant bit.

The parity bit positions are powers of 2: {1,2,4,8,16,32...}. All remaining positions hold data bits. Here is a table representing a 21-bit code word:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
P	P		P				P								P					

The 16-bit data value 1000111100110101 would be stored as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
P	P	1	P	0	0	0	P	1	1	1	1	0	0	1	P	1	0	1	0	1

Calculating Parity

The following table shows exactly which data bits are checked by each parity bit in a 21-bit code word:

Parity Bit	**Checks the following Data Bits**	**Hint***
1	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21	use 1, skip 1, use 1, skip 1, ...
2	2, 3, 6, 7, 10, 11, 14, 15, 18, 19	use 2, skip 2, use 2, skip 2, ...
4	4, 5, 6, 7, 12, 13, 14, 15, 20, 21	use 4, skip 4, use 4, ...
8	8, 9, 10, 11, 12, 13, 14, 15	use 8, skip 8, use 8, ...
16	16, 17, 18, 19, 20, 21	use 16, skip 16, ...

Encoding a Data Value

For the first example, let's use the 8-bit data value 1 1 0 0 1 1 1 1, which will produce a 12-bit code word. Let's start by filling in the data bits:

1	2	3	4	5	6	7	8	9	10	11	12
P	P	**1**	P	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

Next, we begin calculating and inserting each of the parity bits.

P1: To calculate the parity bit in position 1, we sum the bits in positions 3, 5, 7, 9, and 11: (1+1+0+1+1 = 4). This sum is even (indicating *even parity*), so parity bit 1 should be assigned a value of 0. By doing this, we allow the parity to remain even:

1	2	3	4	5	6	7	8	9	10	11	12
0	P	**1**	P	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

P2: To generate the parity bit in position 2, we sum the bits in positions 3, 6, 7, 10, and 11: (1+0+0+1+1 = 3). The sum is odd, so we assign a value of 1 to parity bit 2. This produces even parity for the combined group of bits 2, 3, 6, 7, 10, and 11:

1	2	3	4	5	6	7	8	9	10	11	12
0	**1**	**1**	P	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

P4: To generate the parity bit in position 4, we sum the bits in positions 5, 6, 7, and 12: (1+0+0+1 = 2). This results in **even** parity, so we set parity bit 4 to zero, leaving the parity even:

1	2	3	4	5	6	7	8	9	10	11	12

0	1	**1**	**0**	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

P8: To generate the parity bit in position 8, we sum the bits in positions 9, 10, 11 and 12: (1+1+1+1 = 4). This results in **even** parity, so we set parity bit 8 to zero, leaving the parity even:

1	2	3	4	5	6	7	8	9	10	11	12
0	1	**1**	0	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

All parity bits have been created, and the resulting code word is: 011010001111.

Detecting a Single Error

When a code word is received, the receiver must verify the correctness of the data. This is accomplished by counting the 1 bits in each bit and verifying that each has even parity. Recall that we arbitrarily decided to use even parity when creating code words. Here are the bit groups for a 12-bit code value:

Parity Bit	**Bit Group**
1	1, 3, 5, 7, 9, 11
2	2, 3, 6, 7, 10, 11
4	4, 5, 6, 7, 12
8	8, 9, 10, 11, 12

If one of these groups produces an odd number of bits, the receiver knows that a transmission error occurred. As long as only a single bit was altered, it can be corrected. The method can be best shown using concrete examples.

Example 1: Suppose that the bit in position 4 was reversed, producing 011110001111. The receiver would detect an odd parity in the bit group associated with parity bit 4. After eliminating all bits from this group that also appear in other groups, the only remaining bit is bit 4. The receiver would toggle this bit, thus correcting the transmission error.

Example 2: Suppose that bit 7 was reversed, producing 011010101111. The bit groups based on parity bits 1, 2, and 4 would have odd parity. The only bit that is shared by all three groups (the *intersection* of the three sets of bits) is bit 7, so again the error bit is identified:

Parity Bit	**Bit Group**
1	1, 3, 5, 7, 9, 11
2	2, 3, 6, 7, 10, 11
4	4, 5, 6, 7, 12
8	8, 9, 10, 11, 12

Example 3: Suppose that bit 6 was reversed, producing 011011001111. The groups based on parity bits 2 and 4 would have odd parity. Notice that two bits are shared by these two groups (their intersection): 6 and 7:

Parity Bit	Bit Group
1	1, 3, 5, 7, 9, 11
2	2, 3, 6, 7, 10, 11
4	4, 5, 6, 7, 12
8	8, 9, 10, 11, 12

But then, but 7 occurs in group 1, which has even parity. This leaves bit 6 as the only choice as the incorrect bit.

Multiple Errors

If two errors were to occur, we could detect the presence of an error, but it would not be possible to correct the error. Consider, for example, that both bits 5 and 7 were incorrect. The bit groups based on parity bit 2 would have odd parity. Groups 1 and 4, on the other hand, would have even parity because bits 5 and 7 would counteract each other:

Parity Bit	Bit Group
1	1, 3, 5, 7
2	2, 3, 6, 7
4	4, 5, 6, 7

This would incorrectly lead us to the conclusion that bit 2 is the culprit, as it is the only bit that does not occur in groups 1 and 4. But toggling bit 2 would not fix the error--it would simply make it worse.

URL:

1. Behrouz A Forouzan, “Data Communications and Networking”, Tata McGraw Hill, 2013
2. https://codegolf.stackexchange.com/questions/45684/correct-errors-using-hamming7-4
3. https://gist.github.com/vatsal-sodha/f8f16b1999a0b5228143e637d617c797
4. http://code.activestate.com/recipes/580691-hamming-error-correction-code/
5. https://www.tutorialspoint.com/hamming-code-for-single-error-correction-double-error-detection
6. http://programacionits.blogspot.com/2013/05/hamming-code.html
7. https://www.guru99.com/hamming-code-error-correction-example.html