Computer Networks - Exp 3

Kartik Jolapara

60004200107 - B1

Aim

To implement CRC and Hamming Code as error detection and correction codes.

Theory

Hamming Code

Hamming code is a set of error-correction codes that can be used to detect and correct the errors that can occur when the data is moved or stored from the sender to the receiver.

Redundant bits are extra binary bits that are generated and added to the information-carrying bits of data transfer to ensure that no bits were lost during the data transfer. These redundancy bits are placed at the positions which correspond to the power of 2.

Parity Bits

A parity bit is a bit appended to a data of binary bits to ensure that the total number of 1’s in the data is even or odd. Parity bits are used for error detection. There are two types of parity bits:

Even Parity Bit

In the case of even parity, for a given set of bits, the number of 1’s are counted. If that count is odd, the parity bit value is set to 1, making the total count of occurrences of 1’s an even number. If the total number of 1’s in a given set of bits is already even, the parity bit’s value is 0.

Odd Parity Bit

In the case of odd parity, for a given set of bits, the number of 1's are counted. If that count is even, the parity bit value is set to 1, making the total count of occurrences of 1's an odd number. If the total number of 1's in a given set of bits is already odd, the parity bit's value is 0.

Code

```
#include <stdio.h>

#include <math.h>

int hamming_calculate(int position, int length, int code[])

{

   int count = 0;

   int i, j, k;

   i = position - 1;

   while (i < length)

   {

      for (j = i; j < i + position; j++)

      {

         if (code[j] == 1)

         {

            count++;

         }

      }

      i = i + 2 * position;
```

```
    }
    if (count % 2 == 0)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
void print(int n, int p, int code[])
{
    for (int i = 0; i < n + p; i++)
    {
        printf(" %d \t", code[i]);
    }
    printf("\n");
    int j = 0;
    for (int i = 0; i < n + p; i++)
    {
        if (i == (pow(2, j) - 1))
```

```
        {
            printf(" P%d \t", i + 1);
            j++;
        }
        else
        {
            printf(" D%d \t", i + 1);
        }
    }
}
int main()
{
    int code[50], input[50];
    int n, p;
    int counter;
    printf("Enter the size of hamming code : ");
    scanf("%d", &n);
    for (int i = 1; i < n; i++)
    {
        if (pow(2, i) >= n + i + 1)
        {
```

```
        p = i;
        break;
    }
}
printf("Enter the data of hamming code \n");
for (int i = 0; i < n; i++)
{
    printf("Enter the value of bit %d : ", i + 1);
    scanf("%d", &input[i]);
    printf("\n");
}
int j = 0, k = 0;
for (int i = 0; i < n + p; i++)
{
    if (i == (pow(2, j) - 1))
    {
        code[i] = 0;
        j++;
    }
    else
    {
```

```
            code[i] = input[k];
            k++;
        }
    }
    printf("The Intial Hamming code is :\n");
    print(n, p, code);
    for (int i = 0; i < p; i++)
    {
        int position = pow(2, i);
        int value = hamming_calculate(position, n + p, code);
        code[position - 1] = value;
    }
    printf("\n\n");
    printf("The Final Hamminag Code is : \n");
    print(n, p, code);
    return 0;
}
```

Output

```
Enter the size of hamming code : 10
Enter the data of hamming code
Enter the value of bit 1 : 1

Enter the value of bit 2 : 0

Enter the value of bit 3 : 1

Enter the value of bit 4 : 0

Enter the value of bit 5 : 1

Enter the value of bit 6 : 0

Enter the value of bit 7 : 1

Enter the value of bit 8 : 0

Enter the value of bit 9 : 1

Enter the value of bit 10 : 0

The Intial Hamming code is :
 0       0       1       0       0       1       0       0       1       0       1       0       1       0
 P1      P2      D3      P4      D5      D6      D7      P8      D9      D10     D11     D12     D13     D14

The Final Hamminag Code is :
 0       1       1       0       0       1       0       1       1       0       1       0       1       0
 P1      P2      D3      P4      D5      D6      D7      P8      D9      D10     D11     D12     D13     D14
```

Cyclic Redundancy Check (CRC)

CRC or Cyclic Redundancy Check is a method of detecting accidental changes/errors in the communication channel. CRC uses **Generator Polynomial** which is available on both sender and receiver side. An example generator polynomial is of the form like $x^3 + x + 1$. This generator polynomial represents key 1011.

CRC is based on binary division.

In CRC, a sequence of redundant bits, called cyclic redundancy check bits, are appended to the end of the data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number. At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted. A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.

Code

```
#include <stdio.h>

void modulo2Div(int n, int l, int g[], int d[], int code[])

{

   int key[l + n], m, j, k;

   for (int i = 0; i < l; i++)

   {

      m = 0;

      k = key[i];

      for (j = i; j < i + 4; j++)

      {

         if (i == 0)
```

```
{
    if (g[m] == d[m])
    {
        key[j] = 0;
    }
    else
    {
        key[j] = 1;
    }
}
else if (k == 0)
{
    if (key[j] == 0)
    {
        key[j] = 0;
    }
    else
    {
        key[j] = 1;
    }
}
```

```
        else
        {
            if (key[j] == g[m])
            {
                key[j] = 0;
            }
            else
            {
                key[j] = 1;
            }
        }
        m++;
    }
    key[j] = d[i + 4];
}
for (int i = 0; i < l; i++)
{
    code[i] = d[i];
}
for (int i = l; i < n + l; i++)
{
```

```
        code[i] = key[i];
    }
}
int main()
{
    int n, error = 0;
    printf("Enter the highest degree of G(x) : ");
    scanf("%d", &n);
    int g[n];
    for (int i = 0; i <= n; i++)
    {
        printf("\nEnter the coefficient of x^%d : ", n - i);
        scanf("%d", &g[i]);
    }
    int l;
    printf("Enter the length of original data : ");
    scanf("%d", &l);
    int d[l + n];
    printf("\nEnter the original data : ");
    for (int i = 0; i < l; i++)
    {
```

```
    scanf("%d", &d[i]);
}
for (int i = 0; i < n; i++)
{
    d[l + i] = 0;
}
int code[l + n];
modulo2Div(n, l, g, d, code);
printf("\nGenerator : ");
for (int i = 0; i <= n; i++)
{
    printf("%d", g[i]);
}
printf("\n");
printf("\nData : ");
for (int i = 0; i < l; i++)
{
    printf("%d", d[i]);
}
printf("\n");
printf("\nRemainder : ");
```

```
for (int i = l; i < l + n; i++)
{
    printf("%d", code[i]);
}
printf("\n");
printf("\nCodeword : ");
for (int i = 0; i < l + n; i++)
{
    printf("%d", code[i]);
}
printf("\n");
modulo2Div(n, l, g, code, code);
for (int i = l; i < l + n; i++)
{
    if (!code[i] == 0)
    {
        error = 1;
    }
}
if (error)
{
```

```
        printf("\nError!");
    }
    else
    {
        printf("\nRemainder at receiving side : ");
        for (int i = l; i < l + n; i++)
        {
            printf("%d", code[i]);
        }
        printf("\n");
        printf("\nReceived successfully");
    }
    return 0;
}
```

Output

```
Kartik:CN Exp 3 -  CRC & Hamming/ (master) $ ./Hamming.exe
Enter the highest degree of G(x) : 3

Enter the coefficient of x^3 : 1

Enter the coefficient of x^2 : 1

Enter the coefficient of x^1 : 0

Enter the coefficient of x^0 : 1
Enter the length of original data : 6

Enter the original data : 1
0
0
1
0
0

Generator : 1101

Data : 100100

Remainder : 001

Codeword : 100100001

Remainder at receiving side : 000

Received successfullyKartik:CN Exp 3 -  CRC & Hamming/ (master) $
```

Conclusion

Thus we successfully implemented Error detection and correction techniques.