Recovery of √n bytes of data using Backtracking Algorithm

Asha Rani K.P[1], Saravana B[2], Raghu D.R[3], Raghunandan Athreya[4], Sangamesh J C[5]

[1]*Computer Science and Engineering Department,*
Dr. Ambedkar Institute of Technology, Outer Ring Road, Mallathalli, Bangalore-560056, Karnataka, India

[2]*Bachelor of Engineering, computer Science and Engineering*

[3]*Bachelor of Engineering, computer Science and Engineering*

[4]*Bachelor of Engineering, computer Science and Engineering*
Tech Mahindra
Pune, India

[5]*Bachelor of Engineering, computer Science and Engineering*
Tata Consultancy Services
Bangalore, India

Abstract

Assume that a busy server is transferring files across the network, and during the transfer certain data present in the file is replaced with erroneous data. If the client node or receiver node does not have any technique to detect errors, then it would process the erroneous data got, and provide unexpected results. Now assume that the client just has a error detecting technique, it would be able to detect if errors were present in the received file or not, but will not be able to correct it, and in case it wants the correct data it will have to request the busy server to send the file again, this in turn leads to wastage of precious server cycles and bandwidth.

Hence an efficient and effective error detection algorithm is required to overcome the above mentioned problems. Proposed algorithm can recover the erroneous data. It can be applied to all kind of data files.

Here we divide the input data into a 2 D square matrix, calculate xor value of all rows, and columns present in the matrix. This information is transferred along with the actual data, and at receiver end we use backtracking algorithm to recover erroneous data if any. It contains three stages, encoding of the data, testing for errors, decoding the data.

Before sending any file across the network it is encoded by adding header to the actual data, header contains the necessary information required for backtracking algorithm. The output file received is tested for errors and if errors are found it is corrected using backtracking algorithm and the resultant data file is decoded to obtain the original data. If the file size is of n bytes then the backtracking algorithm can correct upto √n bytes of the data.

Keywords : Ex-OR, RAID(Redundant Array of Inexpensive Disks), BackTracking Algorithm, Error Detection, Error Recovery, Repetition schemes, Checksum, Cyclic redundancy checks, Hamming Code, Automatic Repeat-Request.

1. Introduction

Data transfer across network is increasing remarkably. And whenever data is transferred across network, there are always possibilities of erroneous data being received at client end. Hence there is a need for efficient and effective technique for error detection and recovery. There are many techniques available in market today.

Advantages of the existing technique

- CRC is the widely used technique and it can detect errors in large amount of data, by making use of shift registers.
- Many error correcting algorithms are available that corrects few bit of errors like Huffman coding, bch code etc.

Problems with existing techniques

- CRC is widely used technique but this technique can only be used to detect errors, and can correct only 1 bit error that too if the length of the data is small

- Many error detecting algorithms like Huffman coding ,bch code correct only a few bit of errors but the amount of redundancy added is considerably large

Our aim is to come up with a better technique that adds as less redundant data as possible, also detects errors efficiently and also to make the software time efficient.

2. Related Work

2.1 Error Detection

2.1.1 Repetition schemes:

Variations on this theme exist. Given a stream of data that is to be sent, the data is broken up into blocks of bits, and in sending, each block is sent some predetermined number of times. For example, if we want to send "1011", we may repeat this block three times each.

Suppose we send "1011 1011 1011", and this is received as "1010 1011 1011". As one group is not the same as the other two, we can determine that an error has occurred. This scheme is not very efficient, and can be susceptible to problems if the error occurs in exactly the same place for each group (e.g. "1010 1010 1010" in the example above will be detected as correct in this scheme). The scheme however is extremely simple, and is in fact used in some transmissions of numbers stations.

2.1.2 Checksum:

A checksum of a message is an arithmetic sum of message code words of a certain word length, for example byte values, and their carry value. The sum is negated by means of ones-complement, and stored or transferred as an extra code word extending the message.

On the receiver side, a new checksum may be calculated from the extended message. If the new checksum is not 0, an error has been detected.

Checksum schemes include parity bits, check digits and longitudinal redundancy check.

2.1.3 Cyclic redundancy checks:

More complex error detection (and correction) methods make use of the properties of finite fields and polynomials over such fields.

The cyclic redundancy check considers a block of data as the coefficients to a polynomial and then divides by a fixed, predetermined polynomial. The coefficients of the result of division are taken as the redundant data bits, the CRC. On reception, one can recompute the CRC from the payload bits and compare this with the CRC that was received. A mismatch indicates that an error occurred.

2.2 Error Correction

Hamming distance based checks: Since it takes many bit errors to convert one valid Hamming code word to any other valid Hamming code word, the receiver can correct any single-bit error in a word by finding the "closest" valid Hamming code, the one code word that has only one bit different from the received word.

Some codes can correct a certain number of bit errors and only detect further numbers of bit errors. Codes which can correct one error are termed single error correcting (SEC), and those which detect two are termed double error detecting (DED). Hamming codes can correct single-bit errors and detect double-bit errors (SEC-DED)-more sophisticated codes can correct and detect more errors.

An error-correcting code which corrects all errors of up to n bits correctly is also an error-detecting code which can detect at least all errors of up to 2n bits.

2.2.1 Reed–Solomon

In coding theory, Reed–Solomon (RS) codes are non-binary cyclic error-correcting codes invented by Irving S. Reed and Gustave Solomon. They described a s ystematic way of building codes that could detect and correct multiple random symbol errors. By adding t check symbols to the data, an RS code can detect any combination of up to t erroneous symbols, and correct up to $\lfloor t/2 \rfloor$ symbols.

2.2.2 Hamming Code.

In telecommunication, a Hamming code is a linear error-correcting code named after its inventor, Richard Hamming. Hamming codes can detect up to two simultaneous bit errors, and correct single-bit errors; thus, reliable communication is possible when the Hamming distance between the transmitted and received bit patterns is less than or equal to one.

Table 1: Comparisons of various Error Correction Algorithms.

Error Recovery Technique	***Number of Bits Recovered***	***Output file size***
Repetition Scheme	800	2400
Parity Scheme	0	850 - 900
Checksum	0	900
Cyclic Redundancy Check Sum	1	805 -813
Polarity Scheme	800	1600
Hamming code	200	1400
Convolution	800	2400
Reed Solomen	120	2040

Consider the file size in 100 bytes long that is 800 bits the no of bits the various error correcting schemes and the size of the file is given above.

3. Methodology

This project is based on the Even parity property of Ex-OR. We divide the data that needs to be transferred across the network into rows and columns. And while encoding we make sure that data present in each row and column have even parity. We make use of Ex-OR property to accomplish this

Once this is done we transfer data across the network. During transmission if any error occurs we find out the rows in which error has occurred using even parity property of Ex-OR and correct it using Backtracking algorithm.

We have used Ex-OR here because Ex-OR by nature itself is Even Parity. That is if the number of 1(+5V) inputs to the Ex-OR gate is odd then output of Ex-OR gate is 1(+5V) else it is 0.

Table 2: Ex-OR Property

Input 1	*Input 2*	*Output 1*
0	0	0
0	1	1
1	0	1
1	1	0

Here is a simply examples which shows how we detect error using Ex-OR and recover it using backtracking algorithm.

Consider this to be the data we need to transfer across network.

0111 1000 0101 1111 0011 1100 0010 1010 0110 1110 1111 0101 1100 0001 0111 0011 1110 1101 1001 0100 0100 1011 1000 0010 1010

Now these elements are arranged into a matrix.

Fig 1: Arrangement of data

0111	1000	0101	1111	0011
1100	0010	1010	0110	1110
1111	0101	1100	0001	0111
0011	1110	1101	1001	0100
0100	1011	1000	0010	1010

Here during transmission let us assume that error occurred in following locations.

1) 3rd row, 1st column
2) 3rd row, 2nd column
3) 2nd row, 4th column

Elements encircled and hi ghlighted in red below are the positions where error has occurred.

Moving forward we will assume error has occurred at following position (Will be highlighted in blue) and we will apply backtracking algorithm to calculate the values that should have been present there. And thus recover the erroneous data

Step 1: We start of assuming might have occurred on 2nd row 1st column. As underlined.

Fig 2: Recovery as per step 1.

0111	1000	0101	1111	0011
1100	0010	1010	1110	1110
1110	0111	1100	0001	0111
0011	1110	1101	1001	0100
0100	1011	1000	0010	1010

So we try to generate that data that could be present in the cell from rest of the data present in that column.

In this case it is 0111 (1st row 1st column), 1110 (3rd row 1st column), 0011 (4th row 1st column), 0100 (5th row 1st column) and, 0011 (result of Ex-OR of that particular column)

Value got from this is put in 2nd row 1st column and horizontal Ex-OR of 2nd row is calculated to check if our assumption was right. In case our assumption is right then value optioned by calculating the Ex-OR of 2nd row should be equal to Actual Ex-OR value.

Ex-OR of 0111, 1110, 0011, 0100, 0011 is 1101.

Now value 1101 is put in 2nd row 1st column and horizontal Ex-OR of that row is calculated.

Ex-OR value of 1101, 0010,1010,1110,1110 is 0101! = 1100 which is not equal to actual Ex-OR value of that particular Row. So the assumption we made was wrong.

Step 2**:** Now we assume that error was present in 3rd row 1st column.

And we calculate the value that must have been present here using other values in that column and actual Ex-OR value of that column. 0111 (1st row 1st column), 1100 (2rd row 1st column), 0011 (4th row 1st column), 0100 (5th row 1st column) and 0011 (result of Ex-OR of that particular column)

Value that should have been present in 3rd row 1st column is Ex-OR (0111, 1100, 0011, 0100, 0011) = 1111

Now value 1111 is put in 3rd row 1st column and horizontal Ex-OR of that row is calculated.

Ex-OR value of 1111, 0111,1100,0001,0111 is 0010! = 0000

So our assumption of error being present on 3rd row 1st column is wrong.

Step 3: So we move further with the algorithm and assume that error was present in both (1st and 2nd columns of 2nd row). As underlined.

Fig 3: Recovery as per Step 3.

0111	1000	0101	1111	0011
1100	0010	1010	1110	1110
1110	0111	1100	0001	0111
0011	1110	1101	1001	0100
0100	1011	1000	0010	1010

We repeat the process of calculating the values that should have been present in those cells by using the values present in other columns.
Calculating the value present in 2nd row 1st column.
0111 (1st row 1st column), 1110 (3rd row 1st column), 0011 (4th row 1st column), 0100 (5th row 1st column) and 0011 (result of Ex-OR of that particular column)
Ex-OR of 0111, 1110, 0011, 0100, 0011 is 1101.
Calculating the value present in 2nd row 2nd column.
1000 (1st row 2nd column), 0111 (3rd row 2nd column), 1110 (4th row 2nd column), 1011 (5th row 2nd column) and 1010 (Actual Ex-OR value of that column)
Ex-OR of 1000, 0111, 1110, 1011, 1010 is 0000
Now we put 1101(into 2nd row 1st column) and 0000(into 2nd row 2nd column).
And calculate the Ex-OR of that particular row.
Ex-OR (1101, 0000, 1010, 0110, 1110) = 1111! = 1100
So our assumption of error being present in 1st and 2nd column of 2nd row was wrong.

Step 4**:** Now we assume error was present in 1st and 2nd column of 3rd row. As underlined.

Fig 4: Recovery as per step 4.

0111	1000	0101	1111	0011
1100	0010	1010	1110	1110
1110	0111	1100	0001	0111
0011	1110	1101	1001	0100
0100	1011	1000	0010	1010

Calculating the value that was present in 1st column of 3rd row.
0111 (1st row 1st column), 1100 (2rd row 1st column), 0011 (4th row 1st column), 0100 (5th row 1st column) and 0011 (result of Ex-OR of that particular column)

Value that was present in 1st column of 3rd row is
Ex-OR (0111, 1100, 0011, 0100, 0011) = 1111

Calculating the value that was present in 2nd column of 3rd row.
1000 (1st row 2nd column), 0010 (2rd row 2nd column), 1110 (4th row 2nd column), 1011 (5th row 2nd column), 1010 (Actual Ex-OR value of that column)
Value that was present in 2nd column of 3rd row is
Ex-OR (1000, 0010, 1110, 1011, 1010) = 0101
Now we put 1111(into 3nd row 1st column) and 0101(into 2nd row 2nd column)
And calculate the Ex-OR of that particular row.
Ex-OR of 3rd row is Ex-OR (1111, 0101, 1100, 0001, 0111) = 0000
And the value calculated = A ctual Ex-OR value of that particular row so our assumption of error being present in 1st and 2nd column of 3rd row was correct. Moving forward in the algorithm we will be using these values.

Fig 5: Correct data obtained.

0011	1110	1101	1001	0100
0100	1011	1000	0010	1010

Step 5: Now we start of freshly assuming that value present in 3rd column of 2nd row was erroneous.
Repeat the process of finding the value that should have been present there using rest of the values present in that column and Ex-OR value of that column.
0101 (1st row 3rd column), 1100 (3rd row 3rd column), 1101 (4th row 3rd column), 1000 (5th row 3rd column), 0110 (Actual Ex-OR value of 3rd column)
Ex-OR (0101, 1100, 1101, 1000, 0110) = 1010
Now we put 1010 as the value that should have been present in 3rd column of 2nd row and calculate the Ex-OR of that row.
Ex-OR (1100, 0010, 1010, 0110, 1110) = 1100
And this value is not equal to Actual ex-OR value of that row which is 1100. So our assumption was wrong.

Step 6: Now we assume error was present in 3rd column of 3rd row and calculate the value that should have been present there using rest of the values as done previously.
0101 (1st row 3rd column), 1010 (2nd row 3rd column), 1101 (4th row 3rd column), 1000 (5th row 3rd column), 0110 (Actual Ex-OR value of 3rd column)
Ex-OR (0101, 1010, 1101, 1000, 0010) = 1100
Now we put 1000 as the value that should have been present in 3rd column of 3rd row and calculate the Ex-OR of that row.
Ex-OR (1111, 0101, 1100, 0001, 0111) = 0000
And this value is equal to Actual Ex-OR value of that row where that error was present at that location.
Though backtracking algorithm interpreted the error location wrongly it does not harm the actual process of recovery of data. This occurred because there was no error in 3rd column of the matrix. And works in favor of backtracking algorithm to improve its efficiency. Hence our assumption of error being present in 3rd column of 3rd row was correct. But during

transmission there was no error at this particular cell. But still backtracking algorithm told that error was present at that location.

Fig 6: Recovery as per Step 6

0111	1000	0101	1111	0011
1100	0010	1010	1110	1110
1110	0111	1100	0001	0111
0011	1110	1101	1001	0100
0100	1011	1000	0010	1010

Step 7**:** Now we start of freshly assuming that value present in 4th column of 2nd row was erroneous. As underlined.

And we calculate the value that must have been present here using other values in that column and actual Ex-OR value of that column.

1111 (1st row 4th column), 0001 (3rd row 4th column), 1001 (4th row 4th column), 0010 (5th row 4th column), 0011 (Actual Ex-OR value of 4th column)

Ex-OR (1111, 0001, 1001, 0010, 0011) = 0110 Now we put 0110 as the value that should have been present in 4th column of 2rd row and calculate the Ex-OR of that row.

Ex-OR (1100, 0010, 1010, 0110, 1110) = 1100 and this value is equal to Actual Ex-OR value of that row which is 1100. Hence our assumption of error being present in 4th column of 2rd row is correct. And thus we have got the actual data from erroneous one.

Interchange the rows and columns and repeat the process if error is still present.

4. Implementation and result

```
Repeat the process until all columns are covered.
{
  Increment the number of column that needs to be calculated by 1 say j
  {
      Repeat for number of rows for which error has occurred
      {
Assume error had occurred in those numbers of columns and calculate the value that had to be present in all "j" columns by using rest of the values in that column.
Replace all the "j" columns with the value that was calculated in the previous step and calculate the Ex-OR value of that particular row.
Compare the Ex-OR value computed during the previous step with the actual values.
If both are equal then error was detected.
                {
                  Again start from j = 1;
                }
            }
                }
        }
```

The algorithm was implemented and data was recovered using backtracking algorithm.

Here is the file comparison of actual file size versus Encoded data.

Table 3: Recovery of data through BackTracking Algorithm.

Actual File Size(in bytes)	*After Encoding (in bytes)*	*%Increase in size(in bytes)*
25	41	64
100	126	26
1000	1094	9.4
100000	101129	1.12
1000000	1002006	0.2

5. Conclusion

The algorithm discussed above need not be used only for detecting and recovering errors for files transferred across network. It can able be used while storing data too. Currently there are few places where Ex-OR is used for data recovery. This algorithm can be used in all those places, the above algorithm increases the efficiency of Ex-OR is detecting and recovering errors.

One such example is with RAID. Currently RAID (2, 3, 4, 5) uses Ex-OR (parity) to detect and recover data. But has a fault tolerance of 1 disk. But with the help of above algorithm the fault tolerance of RAID can be increased, since data is arrange into a 2D array as compared to Conventional RAID technologies where it is arranged in 1D. That is if the number of disks present is 9 then with the help of RAID 2, 3, 4 or 5 technologies only 1 disk can be recovered. But with the help of above algorithm up to 3 disks can be recovered. As a result fault tolerance is increased to 3 disks. This arrangement of data into 2D array also increase the error detection and recovery property of Ex-OR.

This is one such scenario in which performance is increased with the help of above algorithm. Similarly there are many such scenario in which the above algorithm could be used to increase the amount of data recovered by Ex-OR.

REFERENCES

[1] *Gilles Brassard, Paul Bratley (1995). Fundamentals of Algorithmics. Prentice-Hall*
[2] *HBmeyer.de Interactive animation of a backtracking algorithm.*
[3] *Peterson, W. W. and Brown, D. T. (January 1961). "Cyclic Codes for Error Detection".*
[4] *Shu Lin, Daniel J. Costello, Jr. (1983). Error Control Coding: Fundamentals and Applications*
[5] *http://en.wikipedia.org/wiki/*