Title: Implementation of hamming codes for error detection and correction.

Done by: Thota Surya Vara Prasad Rao, AP22110011059,CSE-G.

Submitted on: 31-08-2024

Objective:
Hamming codes are a method used to detect and correct errors during data transmission. They are particularly effective for correcting single-bit errors.

How it Works:
Hamming codes enable error detection and correction by adding redundant (parity) bits to the original data. The positions of these parity bits are powers of 2 (1, 2, 4, 8, etc.). Each parity bit is responsible for checking specific positions within the encoded data. If an error occurs during transmission, the parity bits can identify the exact position of the erroneous bit, allowing for correction.

Limitation:

1. **Single-Bit Error Correction Only**: Hamming codes are designed to detect and correct single-bit errors. They are not equipped to correct errors that affect multiple bits simultaneously.
2. **No Detection of Double-Bit Errors**: Although Hamming codes can detect double-bit errors, they cannot correct them. In some cases, a double-bit error might be misinterpreted as a single-bit error, leading to incorrect corrections and potential data corruption.
3. **Overhead in Redundant Bits**: The addition of redundant bits increases the data size, resulting in overhead. This can be inefficient for systems transmitting large volumes of data.
4. **Complexity in Implementation**: As the length of the data increases, calculating parity bits and performing error correction becomes more complex, making Hamming codes less efficient for larger datasets.

Problem Statement:
Write a program to implement an Error Detection and Correction Technique using the Hamming codes algorithm.

CODE:

```
#include <iostream>

#include <cmath>

using namespace std;
```

```
// Function to calculate parity bits for the encoded data
void CalculateParityBits(int encodedData[], int dataLength, int redundantBits) {
    for (int i = 0; i < redundantBits; ++i) {
        int parityBitPosition = pow(2, i);
        for (int j = 1; j <= dataLength + redundantBits; ++j) {
            if (((j >> i) & 1) == 1) {
                encodedData[parityBitPosition] ^= encodedData[j];
            }
        }
    }
}

// Function to find the position of the erroneous bit in the received data
int FindErrorPosition(int receivedData[], int dataLength, int redundantBits) {
    int errorPosition = 0;
    for (int i = 0; i < redundantBits; ++i) {
        int parityBitPosition = pow(2, i);
        int parity = 0;
        for (int j = 1; j <= dataLength + redundantBits; ++j) {
            if (((j >> i) & 1) == 1) {
                parity ^= receivedData[j];
            }
        }
        if (parity != 0) {
            errorPosition += parityBitPosition;
        }
    }
    return errorPosition;
```

```
}

int main() {
    int dataLength;
    cout << "Enter the length of data bits: ";
    cin >> dataLength;

    // Calculate the number of redundant bits required
    int redundantBits = 0;
    while (pow(2, redundantBits) < dataLength + redundantBits + 1) {
        ++redundantBits;
    }

    // Arrays to store the original data bits and the encoded data
    int dataBits[dataLength + 1]; // Original data bits
    int encodedData[dataLength + redundantBits + 1] = {0}; // Encoded data with
parity bits

    // Input the data bits from the user
    cout << "Enter the data bits: ";
    for (int i = 1; i <= dataLength; ++i) {
        cin >> dataBits[i];
    }

    // Fill the encoded data array with the data bits, skipping parity bit positions
    int dataIndex = 1;
    for (int i = 1; i <= dataLength + redundantBits; ++i) {
        if (((i) & (i - 1)) == 0) {
            continue; // Skip the parity bit positions
```

```
        }
        encodedData[i] = dataBits[dataIndex++];
    }

    // Calculate and set the parity bits in the encoded data
    CalculateParityBits(encodedData, dataLength, redundantBits);

    // Output the encoded data
    cout << "Encoded data: ";
    for (int i = 1; i <= dataLength + redundantBits; ++i) {
        cout << encodedData[i] << " ";
    }
    cout << endl;

    // Simulate a transmission by flipping one bit
    int receivedData[dataLength + redundantBits + 1];
    for (int i = 1; i <= dataLength + redundantBits; ++i) {
        receivedData[i] = encodedData[i];
    }
    int errorPosition;
    cout << "Enter the position to introduce error (1 to " << dataLength +
redundantBits << "): ";
    cin >> errorPosition;
    receivedData[errorPosition] ^= 1;

    // Find the position of the error in the received data
    errorPosition = FindErrorPosition(receivedData, dataLength, redundantBits);

    // Check if there is an error in the received data
```

```
    if (errorPosition == 0) {
        cout << "No error in the data bits." << endl;
    } else {
        cout << "Error found in the data bit. The position of the error bit is: " <<
errorPosition << endl;
        // Correct the error by flipping the bit at the error position
        receivedData[errorPosition] ^= 1;

        // Output the corrected data
        cout << "The data after error detection and correction is: ";
        for (int i = 1; i <= dataLength; ++i) {
            cout << receivedData[i] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

OUTPUTS:

```
PS C:\Users\surya\OneDrive\Desktop\cn lab> cd "c:\Users\surya\OneDrive\Desktop\cn lab\" ; if ($?) { g++ hamming.cpp -o hamming } ; if ($?) { .\hamming
}
Enter the length of data bits: 4
Enter the data bits: 1 1 0 0
Encoded data: 0 1 1 1 1 0 0
Enter the received data bits: 0 0 1 1 1 0 0
Error found in the data bit. The position of the error bit is: 2
The data after error detection and correction is: 0 1 1 1 1 0 0
```

In above output screen, one bit is flipped so it detected the error position and flipped the bit in the error position and gave us the data after error correction. We can conclude that the data after correction and the encoded data are same

```
PS C:\Users\surya\OneDrive\Desktop\cn lab> cd "c:\Users\surya\OneDrive\Desktop\cn lab\" ; if ($?) { g++ hamming.cpp -o hamming } ; if ($?) { .\hamming
}
Enter the length of data bits: 4
Enter the data bits: 1 1 1 1
Encoded data: 1 1 1 1 1 1 1
Enter the received data bits: 1 1 1 1 1 1 1
No error in the data bits.
```

In above output screen, no bit is flipped so it gave the output as no error in the data bits.

```
PS C:\Users\surya\OneDrive\Desktop\cn lab> cd "c:\Users\surya\OneDrive\Desktop\cn lab\" ; if ($?) { g++ hamming.cpp -o hamming } ; if ($?) { .\hamming
}
Enter the length of data bits: 4
Enter the data bits: 1 1 1 1
Encoded data: 1 1 1 1 1 1 1
Enter the received data bits: 0 1 0 1 1 1 1
Error found in the data bit. The position of the error bit is: 2
The data after error detection and correction is: 0 0 0 1 1 1 1
PS C:\Users\surya\OneDrive\Desktop\cn lab>
```

In above output screen, two bits are flipped in the positions 1 and 3. The algorithm detects the error but it cannot find the position of the bits where the data is flipped and it gives wrong output. Therefore, we can conclude that hamming codes algorithm can detect 2-bit error but it cannot correct it.

Problems Faced:

1. **Managing the Positions of Data and Parity Bits**: It was challenging to correctly position the data and parity bits since parity bits occupy positions that are powers of 2. Misplacing these bits could easily lead to incorrect results and hinder the error detection and correction process.

Conclusion:

- **Knowledge**: I gained a solid understanding of how Hamming codes work for error detection and correction, and learned how to implement them effectively in data transmission.
- **Skill**: I refined my C++ programming skills by applying concepts such as bit manipulation, array handling, and logical operations. This allowed me to successfully implement the Hamming code algorithm and enhance my ability to handle complex coding challenges.