Sprawozdanie z przedmiotu „Telekomunikacja i prezentacja sygnałów”

Zadanie nr 1:

Transmisja danych z wykorzystaniem kodowania umożliwiającego automatyczną korekcję błędów

Autor: Jan Karwowski 216793

Cel zadania

Celem zadania jest opracowanie kodów korygujących jeden i dwa błędy w wiadomości ośmiobitowej oraz stworzenie programu wykorzystującego opracowane kody do zakodowania i odkodowania pliku.

Kod korygujący jeden błąd

Aby korygować jeden błąd zostały wykorzystane cztery bity parzystości, których znaczenie jest reprezentowane przez macierz, gdzie każdy wiersz odpowiada jednemu bitowi parzystości. Macierz ta spełnia dwa następujące wymagania:
- nie posiadak kolumny zerowej
- wszystkie kolumny są różne

Spełnienie powyższych wymogów pozwala korygować pojedynczy błąd.

Plik include/kod1.h

```
#include "binary.h"

BinaryMatrix H1 =
{
    {0, 1, 1, 1, 0, 1, 1, 0,     1, 0, 0, 0},
    {1, 0, 1, 1, 0, 0, 1, 1,     0, 1, 0, 0},
    {1, 1, 0, 1, 1, 0, 0, 1,     0, 0, 1, 0},
    {1, 1, 1, 0, 1, 1, 0, 0,     0, 0, 0, 1},
};
```

Kod korygujący dwa błędy

Aby korygować dwa błędy wykorzystane zostało osiem bitów parzystości. Powstała macierz musi spełnić dodatkowy warunek:
- żadna kolumna nie może być sumą dwóch pozostałych

Plik include/kod2.h

```
#include "binary.h"

BinaryMatrix H2 =
{
      {0, 1, 1, 1, 1, 1, 1, 1,      1, 0, 0, 0, 0, 0, 0, 0},
      {1, 0, 1, 1, 1, 1, 1, 1,      0, 1, 0, 0, 0, 0, 0, 0},
      {1, 1, 0, 1, 1, 1, 1, 1,      0, 0, 1, 0, 0, 0, 0, 0},
      {1, 1, 1, 0, 1, 1, 1, 1,      0, 0, 0, 1, 0, 0, 0, 0},
      {1, 1, 1, 1, 0, 1, 1, 1,      0, 0, 0, 0, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 0, 1, 1,      0, 0, 0, 0, 0, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 0, 1,      0, 0, 0, 0, 0, 0, 1, 0},
      {1, 1, 1, 1, 1, 1, 1, 0,      0, 0, 0, 0, 0, 0, 0, 1},
};
```

Program

Plik include/binary.h

```
#include <vector>
#include <iostream>

typedef std::vector<bool> BinaryVector;
typedef std::vector<std::vector<bool>> BinaryMatrix;
typedef unsigned char byte;
typedef std::vector<byte> ByteVector;

BinaryVector multiplyMatrixByVector(const BinaryMatrix& matrix, const
BinaryVector& vector);

BinaryVector codeWord(const BinaryVector& word, const BinaryMatrix& matrix);

BinaryVector decodeWord(const BinaryVector& word, const BinaryMatrix& matrix);

ByteVector codeBytes(const ByteVector& bytes, const BinaryMatrix& matrix);

ByteVector decodeBytes(const ByteVector& bytes, const BinaryMatrix& matrix);

std::ostream& operator<<(std::ostream& os, const BinaryVector& vector);
std::ostream& operator<<(std::ostream& os, const BinaryMatrix& matrix);
```

Powyższy kod zawiera definicję typów i funkcji.

Typy:
- BinaryVector – wektor pojedynczych bitów reprezentowanych przez wartości bool
- BinaryMatrix – zwykły wektor binarnych wektorów, które są wierszami macierzy
- byte – 8-bitowa zmienna
- ByteVector – wektor bajtów

Funkcje**:**
- multiplyMatrixByVector() - mnożenie macierzy, której wiersze są wektorami bitów przez wektor bitów traktowany jako kolumnowy
- codeWord() - funkcja kodująca jedno (np. 8-bitowe) słowo na podstawie macierzy opisującej kod, przyjmuje i zwraca wektor binarny
- decodeWord() - funkcja odwrotna do codeWord()
- codeBytes() - funkcja kodująca cały zbiór bajtów, dla każdego wywołuje codeWord()
- decodeBytes() - funkcja odwrotna do codeBytes()
- operatory << służące do wypisywania macierzy bitowej i wektora bitowego do strumienia (np. na konsolę)

Plik src/binary.cpp

```
#include "binary.h"
#include <iostream>
#include <vector>
using namespace std;

BinaryVector multiplyMatrixByVector(const BinaryMatrix& matrix, const
BinaryVector& vector){
    BinaryVector result;
    for(BinaryVector row : matrix){
        bool rowResult = 0;
```

```
        for(int i = 0; i < row.size(); i++){
            rowResult ^= row[i] & vector[i];
        }
        result.push_back(rowResult);
    }
    return result;
}

BinaryVector codeWord(const BinaryVector& word, const BinaryMatrix& matrix){
    int numberOfParityBits = matrix.size();
    BinaryVector encoded(word);
    encoded.insert(encoded.end(), numberOfParityBits, 0);
    BinaryVector parityBits = multiplyMatrixByVector(matrix, encoded);
    encoded.erase(encoded.end() - numberOfParityBits, encoded.end());
    encoded.insert(encoded.end(), parityBits.begin(), parityBits.end());
    //-----
    //cout << "Kodowanie słowa: " << word << "  --->  " << encoded << endl;
    //-----
    return encoded;
}

BinaryVector decodeWord(const BinaryVector& word, const BinaryMatrix& matrix){
    const int numberOfColumns = 8 + matrix.size();
    const int numberOfRows = matrix.size();
    //create errorVector and check
    BinaryVector errorVector = multiplyMatrixByVector(matrix, word);
    bool correct = true;
    for(bool b : errorVector){
        if(b){
            correct = false;
            break;
        }
    }
    //-----
    //cout << "errorVector: " << errorVector << endl;
    //-----
    int errorBitNumber = -1;
    int errorBitNumber1 = -1;
    int errorBitNumber2 = -1;
    if(!correct){
        //searching matrix column identical with errorVector
        for(int j = 0; j < numberOfColumns; j++){
            bool identical = true;
            for(int i = 0; i < numberOfRows; i++){
                if(matrix[i][j] != errorVector[i]){
                    identical = false;
                    break;
                }
            }
            if(identical){
                errorBitNumber = j;
                break;
            }
        }
        //searching matrix columns which sum equals errorVector
        if(errorBitNumber == -1 && numberOfRows >= 7){
            for(int j1 = 0; j1 < numberOfColumns; j1++){
                bool identical;
                for(int j2 = j1 + 1; j2 < numberOfColumns; j2++){
                    identical = true;
                    for(int i = 0; i < numberOfRows; i++){
                        if(matrix[i][j1] ^ matrix[i][j2] != errorVector[i]){
                            identical = false;
                            break;
```

```
                        }
                    }
                    if(identical){
                        errorBitNumber1 = j1;
                        errorBitNumber2 = j2;
                        break;
                    }
                }
                if(identical){
                    break;
                }
            }
        }
    }
    //decode and repair
    int numberOfParityBits = matrix.size();
    BinaryVector decoded(word);
    if(errorBitNumber != -1){
        //korekcja jednego błędu
        decoded[errorBitNumber] = !decoded[errorBitNumber];
    }else if(errorBitNumber1 != -1 && errorBitNumber2 != -1){
        //korekcja dwóch błędów
        decoded[errorBitNumber1] = !decoded[errorBitNumber1];
        decoded[errorBitNumber2] = !decoded[errorBitNumber2];
    }
    decoded.erase(decoded.end() - numberOfParityBits, decoded.end());
    //-----
    //cout << "Dekodowanie słowa: " << word << "  --->  " << decoded << endl;
    //-----
    return decoded;
}

inline BinaryVector convertByteToBinaryVector(byte b){
    BinaryVector vector;
    for(int i = 0; i < 8; i++){
        vector.push_back(b & (1 << (7 - i)));
    }
    return vector;
}

ByteVector codeBytes(const ByteVector& bytes, const BinaryMatrix& matrix){
    //output vectors
    ByteVector outputInBytes;
    BinaryVector outputInBits;
    //code bytes to outputInBits
    for(byte b : bytes){
        BinaryVector encoded = codeWord(convertByteToBinaryVector(b), matrix);
        outputInBits.insert(outputInBits.end(), encoded.begin(), encoded.end());
    }
    //convert outputInBits to outputInBytes
    int counter = 0;
    byte buffer = 0;
    for(bool b : outputInBits){
        buffer += b;
        if(counter == 7){
            outputInBytes.push_back(buffer);
            buffer = 0;
        }
        counter = (counter + 1) % 8;
        buffer <<= 1;
    }
    if(counter != 0){
        buffer <<= 7 - counter;            //filling with zeros
        outputInBytes.push_back(buffer);
```

```
    }
    return outputInBytes;
}

ByteVector decodeBytes(const ByteVector& bytes, const BinaryMatrix& matrix){
    //output vectors
    ByteVector outputInBytes;
    int wordLength = 8 + matrix.size();
    //convert bytes to BinaryVector;
    BinaryVector inputInBits;
    for(byte b : bytes){
        BinaryVector bits = convertByteToBinaryVector(b);
        inputInBits.insert(inputInBits.end(), bits.begin(), bits.end());
    }
    //decode input and convert to bytes
    for(unsigned long i = 0; i < inputInBits.size() / wordLength; i++){
        BinaryVector decoded =
            decodeWord(BinaryVector(inputInBits.begin() + (i * wordLength),
inputInBits.begin() + ((i + 1) * wordLength)), matrix);
        byte b = 0;
        for(int j = 0; j < 8; j++){
            b += decoded[j];
            if(j != 8 - 1)
                b <<= 1;
        }
        outputInBytes.push_back(b);
    }
    return outputInBytes;
}

ostream& operator<<(ostream& os, const BinaryVector& vector){
    os << "[";
    for(int i = 0; i < vector.size(); i++){
        os << (int)vector[i];
        if(i != vector.size() - 1)
            cout << " ";
    }
    os << "]";
    return os;
}

ostream& operator<<(ostream& os, const BinaryMatrix& matrix){
    for(int i = 0; i < matrix.size(); i++){
        os << matrix[i];
        if(i != matrix.size() - 1)
            cout << endl;
    }
    return os;
}
```

Szczegóły implementacji:

- multiplyMatrixByVector()
Wynikiem mnożenia macierzy przez wektor jest wektor, którego każdy element jest sumą iloczynów (mod 2) kolejnych elementów danego wiersza macierzy i kolejnych elementów wektora, przez który mnożymy.

- codeWord()
Kodowanie słowa polega na znalezieniu bitów parzystości, poprawnych dla danego ciągu kodowanego. Aby to zrobić do słowa kodowanego zostają dostawione bity 0, tyle ile ma być bitów

parzystości. Następnie powstały wektor bitowy jest wymnażany przez macierz w wyniku czego dostajemy wektor, który ma jedynki tam gdzie bity parzystości powinny być ustawione i zera tam gdzie ich być nie powinno. Wystarczy więc zastąpić dostawione wcześniej zera wynikiem mnożenia.

- decodeWord()
Dekodowanie słowa polega najpierw na sprawdzeniu czy w ogóle występują w nim błędy. W tym celu wymnaża się odebrany ciąg bitów przez macierz i jeżeli powstały wektor jest zerowy to słowo jest poprawne (przy założeniu że maksymalna liczba błędów to dwa). Jeżeli wektor nie jest zerowy, to (dalej przy wspomnianym założeniu) jest on jedną z kolumn macierzy lub sumą dwóch kolumn macierzy. Szukamy więc najpierw kolumny, która jest taka sama jak otrzymany wektor, a jeżeli się nie uda, to szukamy dwóch kolumn, których suma jest równa temu wektorowi. Pozostaje tylko poprawić bity, których numery odpowiadając numerom znalezionych kolumn.

- codeBytes() i decodeBytes()
Działanie obu tych funkcji sprowadza się do podzielenia wejściowego wektora bajtów na ciągi po 8 bitów a następnie odpowiednio przekazywaniu ich do funkcji codeWord() lub decodeWord()

Plik src/main.cpp

```
#include "binary.h"
#include "kod1.h"
#include "kod2.h"
#include <fstream>
#include <iostream>
using namespace std;

int main(int n, char** args){
    if(n != 3){
        cout << "using: code/decode filename" << endl;
        return 0;
    }
    string command(args[1]);
    string filename(args[2]);
    ifstream is(filename, ios::binary);
    ByteVector bytes;
    if(!is){
        cout << "Unknown file!" << endl;
        return 0;
    }else{
        while(is){
            byte b = is.get();
            if(is)
                bytes.push_back(b);
        }
    }
    if(command == "code"){
        ByteVector encoded = codeBytes(bytes, H2);
        ofstream os("encoded", ios::binary);
        for(byte b : encoded){
            os.put(b);
        }
        os.close();
    }else if(command == "decode"){
        ByteVector decoded = decodeBytes(bytes, H2);
        ofstream os("decoded", ios::binary);
        for(byte b : decoded){
            os.put(b);
        }
        os.close();
```

```
    }else{
        cout << "Unknown command!" << endl;
    }
    is.close();
}
```

Jedynym zadanie powyższego kodu jest stworzenie prostego interfejsu użytkownika, który umożliwia wybór kodowania bądź dekodowania wybranego pliku.