Subjects for Laboratory Test

Information Theory 2018-2019

Lab 2

1. Write a C program to compute the entropy of a file.

- The program shall receive the name of the file as a command-line argument: `entropy.exe myfile.txt`
- The program should follow the following steps:
  - Open the file for reading (in binary format)
  - Count the number of apparitions of every byte value:
    * hold an array of 256 counters, one for every possibly byte value
    * repeatedly read one byte from the file
    * increment the counter corresponding to the byte read
    * also store and increment a counter for the total number of bytes
  - Compute the probability of every byte value: divide each byte counter to the global counter
  - Compute the entropy, based on the probabilities
  - Show the result

Lab 5

1. Write a C program to perform decoding of the files encoded from the previous laboratory (given separately in a `.zip` file). The program shall be called as follows:

   `Decode.exe code.dat input.txt output.txt`

- The arguments are:
  - `code.dat`: a file containing the code to be used (known as the "codebook" file)
  - `input.txt`: the encoded file
  - `output.txt`: the output file (decoded)

The codebook file contains a vector of 256 elements of the following structure type:

```
typedef struct
{
    int len;                /* length of code, in bits */
    unsigned long code;     /* the first "len" bits are the codeword */
} CODE32BIT;
```

- The program will follow the following steps:
  - Read the full vector from the codebook file;
  - Read the full input encoded file into an array of type `unsigned char`, of max size 1MB (i.e. 1.000.000 bytes);
  - Store the number of bytes actually read (return value of `fread`), so that you know how much of the array is actually used;
  - Decode the characters from the `data` array, as follows:
    - While we haven't processed all bytes actually read, do the following:
      - For all characters, try and see which codeword matches the next bits in `data`;
      - When a codeword matches, write that character in the output file;
      - Advance in the `data` array with the size of the matched codeword;
- **Note**: every time there is a **single codeword** that fully matches the next bits in the `data` array.

Lab 6

1. Write a C program that creates a Shannon code from an input data file. The program shall be called as follows:

   ```
   ShannonCode.exe input.txt code.dat
   ```

   - The arguments are:
     - `input.txt`: the input file, from which the code is created
     - `code.dat`: the output file containing the Shannon code created (known as the "codebook" file). It shall contain a vector of 256 elements of the `CODE32BIT` structure type also used in the previous laboratories.
   - The program will follow the following steps:
     - Declare a vector with 256 elements of the `CODE32BIT` structure type
     - Read the input file and compute the probabilities of every character, just like it was done in lab L02 (copy that code)
     - Do Shannon coding:
       * Sort the probabilities vector in descending order
       * Create the cumulative probabilities vector
       * Compute the length of each codeword, `len`
       * For every cumulative value, find the first `len` bits of its binary value and store them in the codeword
     - Display the codewords for all characters
     - Save the codeword vector to the output file

Lab 7

1. Write a C program that computes and appends the parity bit for every byte in a data file. The program shall be called as follows:

   `Parity.exe input.txe output.txt`

   - The arguments are:
     - `input.txt`: the input file
     - `output.txt`: the output file produced by the program (12.5% larger than the input)
   - The program should consist of the following steps:
     - declare two large vectors of `unsigned char`, for input and output bits
     - open the input file and read everything into the input vector
     - for every group of 8 bits from the input vector:
       * copy them to the output vector
       * compute the parity bit
       * append it to the output vector
     - write the output vector to the output data file

Lab 8

1. Write a C program that checks the parity bit for every byte in a data file.

   The input file is the file produced by the previous program, which has a parity bit inserted after every 8 bits of data.

   The output file should contain only the data, with the parity bits removed.

   The program checks the parity bits and, whenever it detects a mismatch, it prints a message in the console: "Error detected ad byte number X", where X is the byte position in the file.

   The program shall be called as follows:

   `‘ParityCheck.exe input.dat output.dat‘`

   - The arguments are:
     - `input.dat`: the input file produced by the program in the previous lab
     - `output.dat`: the output file, containing only the data (parity bits removed)
   - The program should consist of the following steps:
     - declare two large vectors of `unsigned char`, for input and output bits
     - open the input file and read everything into the input vector
     - for every group of 9 bits from the input vector:
       * copy the first 8 bits to the output vector
       * compute the parity bit of these 8 bits
       * check if the parity bit is the same as the 9th bit (parity bit) from the input vector
       * if not, print a message
     - write the output vector to the output data file

Lab 9

1. Write a C program that performs encoding of a data file with the Hamming (8,4) SECDED code.

   The program shall be called as follows:

   `HammingEncode.exe original.dat encoded.dat`

   - The arguments are:
     - `original.dat`: the original input file
     - `encoded.dat`: the encoded output file
   - The program should consist of the following steps:
     - declare two large vectors of `unsigned char`, for input and output bits
     - open the input file and read everything into the input vector
     - for every group of 4 bits from the input vector:
       * compute the control bits c_0, c_1, c_2, c_4
       * write all the 8 bits in the correct order in the output vector
       * advance by 4 and repeat
     - write the output vector to the output data file

   Hamming (8,4) SECDED encoding procedure operates on a block of 4 information bits (denoted as i_3, i_5, i_6, i_7) and produces a block of 8 output bits:

   $$\mathbf{c} = c_0 c_1 c_2 i_3 c_4 i_5 i_6 i_7$$

   The bits denoted with c are parity bits (or *control* bits), and are computed as follows:

   $$\begin{cases} c_0 = i_3 \oplus i_5 \oplus i_6 = c_1 \oplus c_2 \oplus i_3 \oplus c_4 \oplus i_5 \oplus i_6 \oplus i_7 \\ c_1 = i_3 \oplus i_5 \oplus i_7 \\ c_2 = i_3 \oplus i_6 \oplus i_7 \\ c_4 = i_5 \oplus i_6 \oplus i_7 \end{cases}$$

Lab 10

1. Write a C program that performs decoding of a data file previously with the encoded Hamming (8,4) SECDED code.

   The program shall be called as follows:

   `HammingDecode.exe encoded.dat decoded.dat`

   - The arguments are:
     - `encoded.dat`: the input file, previously encoded
     - `decoded.dat`: the output decoded file
   - The program should consist of the following steps:
     - declare two large vectors of `unsigned char`, for input and output bits
     - open the input file and read everything into the input vector
     - for every group of 8 bits from the input vector:
       * compute the syndrome bits z_0, z_1, z_2, z_3
       * if one error is detected, fix the error (by toggling the erroneous bit) and display a message "Fixed 1 error\n"
       * if two errors are detected, display a message "Detected 2 errors, can't fix\n"
       * write the 4 information bits (i_3, i_5, i_6, i_7) in the output vector
       * advance and repeat
     - write the output vector to the output data file

   Considering the received 8-bit block

   $$\mathbf{r} = r_0 r_1 r_2 r_3 r_4 r_5 r_6 r_7,$$

   decoding is done by computing the **syndrome**:

   $$\begin{cases} z_0 &= r_0 \oplus r_1 \oplus r_2 \oplus r_3 \oplus r_4 \oplus r_5 \oplus r_6 \oplus r_7 \\ z_1 &= r_4 \oplus r_5 \oplus r_6 \oplus r_7 \\ z_2 &= r_2 \oplus r_3 \oplus r_6 \oplus r_7 \\ z_3 &= r_1 \oplus r_3 \oplus r_5 \oplus r_7 \end{cases}$$

   The following cases may take place:

   - $z_0 = z_1 = z_2 = z_3 = 0$: no error
   - $z_0 = 1$: 1 error on position given by $z_1 z_2 z_{3(10)}$
   - $z_0 = 0$, other $z_i \neq 0$: 2 errors on unknown positions

Lab 11

1. Write a C program that performs CRC-16 computation and checking.

   The program shall be called in two possible ways:

   a. with two arguments. In this case the program takes the first as input and produces the encoded file as output.

   ```
   CRC16.exe original.dat encoded.dat
   ```

   b. with one argument. In this case the program takes the encoded file as input and checks if the CRC is OK or not.

   ```
   CRC16.exe encoded.dat
   ```

   - The arguments are:
     - `original.dat`: the input file (original / encoded)
     - `encoded.dat` [optional]: the encoded file, with the CRC value appended
   - The program should consist of the following steps:
     - define an array g with the values 10100000000000011
     - declare one large vector of `unsigned char` for input bits
     - open the input file and read everything into the input vector
     - for every bit in the input vector
       - * if the bit is 1, do XOR starting from this bit with the 17 bits in g
     - there will be 16 bits remaining at the end of the original input vector (the CRC-16 value)
     - then:
       a. If the program is called with two arguments:
          - * write the vector to the output data file, including the CRC-16 value at the end

       b. If the program is called with one argument:
          - * if the CRC-16 value is 0, display "File OK\n", otherwise display "Data corrupted\n"

Lab 12

1. Write a C program to simulate a BSC for a given file. The program shall be called as follows:

```
BSC.exe 0.01 input.txt output.txt
```

- The arguments are:
  - `0.01`: the error probability p of the channel
  - `input.txt`: the input file
  - `output.txt`: the output file
- The program will follow the following steps:
  - declare one large vector of `unsigned char` for input bits
  - open the input file and read everything into the input vector
  - for every bit in the input vector
    - generate a random number `x`, and based on `x` do the following:
    - toggle the bit, with probability p
    - leave the bit unchanged, with probability $1 - p$
  - write the vector to the output data file