CITS 3002
Computer Networks

Lab 1: Error Correction

Worth: 2% of the unit

1. Tasks

TASK 1. HAMMING DISTANCE

Hamming Distance (HDist) can be used to measure the distance between two codewords (see Lecture 2 Physical Layer). By doing so, if we are working with a set of known codewords, then we can decide the closest codeword the received data may be, if there is an error. In this task, we will implement a function named `hamming_distance(codeword1, codeword2)` which takes 2 input parameters `codeword1` and `codeword2` in the form of binary values (e.g., `codeword1= '0100101'`), and return the HDist between the two input codewords.

Now that our HDist is working correctly, we will use it to identify the actual codeword from the given list of received data. Write another function `checking_codewords(codewords, received_data)` to determine the correct codeword IFF there are any errors in received data (i.e., the output is the corrected codeword string). Here, `codewords` is a list of codewords (e.g., ['1111', '0000']), and `received_data` is a string of received codeword. You can assume the length of the codeword will always match. For example we have

```
    codewords = ['0000000000', '1111100000', '0000011111', '1111111111']
    received_data = '0000000010'
```

Then it will correct it to '0000000000''

Note that if the received_data cannot be corrected, then the function returns a string `'error detected'`.

TASK 2. HAMMING CODE

Another approach is to use Hamming Code (HCode), which is useful when we don't have an agreed set of codewords. In this task, we will write HCode (although a good idea to write a general HCode for any length of codeword, you are welcome to specifically write for (11, 7) HCode, just like the example we used in the lectures). Write two functions: `hamming_code_generator(codeword)` for generating the HCode with the given input `codeword`, and `hamming_code_error_detection(received_data)` for checking if there are any errors in the received codeword `received_data`.

Note, this task can be a bit challenging if solving for general purpose HCode. Do utilise various online resources if you are implementing that.

TASK 3. CYCLIC REDUNDANCY CHECK

Cyclic redundancy check (CRC) is a common technique used to provide error detection. An example implementation (in Python) for CRC16 is provided to you in `task3.py`, as well as the data corruption function `corrupt_data(data)`. In this task, we will be designing a test to examine robust the CRC algorithm is. Here are the steps taken to do so.

1. Write a code `random_message(n)` to generate a random byte message of length `n`.
2. Generate the checksum value using CRC code provided.
3. Run the corruption on the generated message.
4. Check whether the checksum is different or not.
5. Repeat steps 1 to 4 for a large number of times (e.g., 10000) and see how often an error (i.e., the corruption) goes unnoticed.

Note: see how the performance (speed and accuracy) changes with respect to different variables (e.g., message length, corruption rate, the number of runs in step 5 etc.).

TASK 4. OUR OWN CHECKSUM

Now we will implement a trivial checksum function, which simply adds the value of each character (in bytes, so ASCII values) in a given message.
Then, carry out the same experiment you did in Task 3 to check the robustness of this checksum. This about why this checksum is more robust (or not).

4. Quiz on LMS

To gain 2% allocated mark for this lab, please ensure that you solve lab questions on LMS by the deadline.