Data Communication Lab
IT510 P: Programming Assignment 1

Areeb Beigh | 2018BITE029 | 22
Al Asad Khan | 2018BITE062 | 43

November 25, 2020

Contents

1 Objectives **1**

2 Project specification **1**

2.1 Meta . . . 1

2.2 Code structure . . . 1

2.2.1 Line encoder . . . 1

2.2.2 Random bitsream generator . . . 2

2.2.3 Longest palindromic sub-sequence . . . 2

2.3 CLI . . . 4

1 Objectives

1. Digital data generator: generates completely random data sequence and a random sequence with some fixed sub-sequences like eight consecutive zeros. It should also return the longest palindromic sequence in the generated data.

2. Line coding schemes to be implemented: NRZ-L, NRZ-I, Manchester, Differential Manchester, AMI.

3. Scrambling schemes: B8ZS, HDB3.

2 Project specification

2.1 Meta

- Language: Python
- Format: Jupyter Notebook + CLI
- Plotting: matplotlib

2.2 Code structure

2.2.1 Line encoder

The line encoder is implemented as a configurable class: *LineEncoder*.

Sample usage:

```
encoder = LineEncoder(volts=5, interval=2)
encoder.encode_b8zs('110000000011')
```

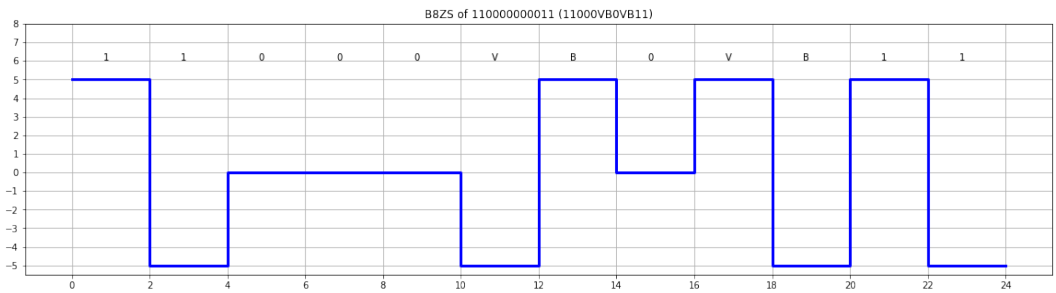


Figure 1: Sample output

2.2.2 Random bitsream generator

The random bitsream generator leverages Python's *random.getrandbits*(n) function that returns a random number with n bits. These bits are then shuffled with *random.shuffle*().

For inserting a repeating subsequence of zeroes, we choose a random starting point in the bitstream previously generated and replace a slice of the stream from this point with 0's.

```python
import random


def generate_bit_stream(length=30, repeating_zeros=None):
    '''
    Generates a random sequence of 1's and 0's.
    '''
    bitstr = list(bin(random.getrandbits(length))[2:])
    random.shuffle(bitstr)

    if repeating_zeros:
        assert repeating_zeros < length, "Repeating zeroes more than bit string"
        z_start = random.randrange(0, length - repeating_zeros + 1)
        z_end = z_start + repeating_zeros
        bitstr[z_start:z_end] = '0' * repeating_zeros

    return ''.join(bitstr)
```

2.2.3 Longest palindromic sub-sequence

The palindromic checks uses the following algorithm:

- Iterate over the given string
- For every character index i: Odd length palindrome check:
  - Set two pointers (ptr1, ptr2) at i-1 and i+1
  - Move pointers outwards until the string isn't a palindrome anymore.

  - Update max_length and start/end indices.

  Even length palindrome check:

  - Set two pointers (ptr1, ptr2) at i and i+1
  - Move pointers outwards until the string isn't a palindrome anymore.
  - Update max_length and start/end indices.

Code:

```python
def check_palindrome(string, ptr1, ptr2, length=0):
    while 0 <= ptr1 and ptr2 <= len(string)-1:
        if string[ptr1] == string[ptr2]:
            ptr1 -= 1
            ptr2 += 1
            length += 2
        else:
            break
    return ptr1, ptr2, length


def longest_palindromic_subseq(string):
    '''
    Find and return start index, end index, and length of the longest palindromic substring
    '''
    if len(string) == 1:
        return 0, 1, 1
    if not string:
        return -1, -1, -1

    idx = 0
    max_length = -1
    start = -1
    end = -1
    while idx < len(string):
        # Check even length
        ptr1, ptr2, length = check_palindrome(string, idx, idx+1)
        if length and length > max_length:
            start = ptr1
```

```
                end = ptr2
                max_length = length
        # Check odd length
        ptr1, ptr2, length = check_palindrome(string, idx-1, idx+1, 1)
        if length != 1 and length > max_length:
            start = ptr1
            end = ptr2
            max_length = length
        idx += 1

    if max_length:
        return start+1, end, max_length
```

Sample run:

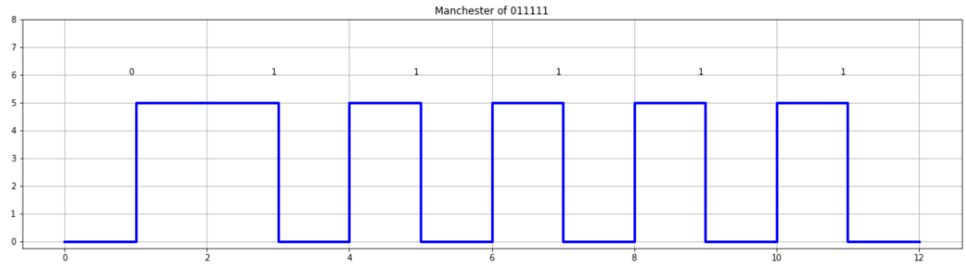
```
s = '111010000101000'
start, end, length = longest_palindromic_subseq(s)
print(start, end, length, s[start:end])

# Output: 2 12 10 1010000101
```

Since the first loop iterates over the string and the palindromic check is $O(N)$, we get a complexity of $O(N^2)$.

2.3 CLI

The CLI is straight forward with 3 stages:

- The first stage is for bitstream generation with 3 options - custom bitstream, random bitsream, random bitsream with repeating 0's.
- Once the bitstream is set, the second stage displays all the available encoding options. Choosing one plots the encoding of the bitstream and restarts this stage with the same bitstream.
- The third stage is a special stage available for choosing the type of AMI encoding - B8ZS, HDB3 or None.




```
Bitstream: 011111
Select the encoding to perform
1. NRZ-L
2. NRZ-I
3. Manchester
4. Differential Manchester
5. AMI
6. All
7. Back

Choice:
```

Figure 2: Sample CLI output