Ex 4

July 25, 2023

Implementation of the Suboptimum Decoding Algorithm, ZJ-Stack

Necessary Libraries and Functions

[9]:

```
import numpy as np
import pandas as pd
from termcolor import colored
```

[10]:

```
LOOKUP_TABLE_Conv = {(2, 1, 3): {'g0': np.array([1, 0, 0, 0], dtype=np.int64),␣
 ↪'g1': np.array([1, 1, 0, 1], dtype=np.int64)}, \
                     (3, 1, 2): {'g0': np.array([1, 1, 0]), 'g1': np.array([1,␣
 ↪0, 1]), 'g2': np.array([1, 1, 1])}, \
                     (2, 1, 2): {'g0': np.array([1, 1, 1]), 'g1': np.array([1,␣
 ↪0, 1])}}
```

[11]:

```
def First_Row_Generator(g_dict: dict) -> np.ndarray:
    num_memory_bits = len(g_dict[list(g_dict.keys())[0]]) - 1
    g_list = []
    for i in range(num_memory_bits + 1):
        for key in g_dict.keys():
            g = g_dict[key]
            g_list.append(g[i])
    g_ndarray = np.array(g_list, dtype=np.int64)
    return g_ndarray
```

[12]:

```
def G_Generator(conv_tuple: tuple, u_length: int) -> np.ndarray:
    h, num_output_bits, m= u_length, conv_tuple[0], conv_tuple[2]
    G = np.zeros((h, num_output_bits*(h + m)), dtype=np.int64)
    g_dict = LOOKUP_TABLE_Conv[conv_tuple]
    g = First_Row_Generator(g_dict)
    count = 0
    for i in range(len(G)):
        G[i][count: len(g) + count] = g
        count += num_output_bits
    return G
```

[13]:

```
def Coder(conv_tuple: tuple, u_seq) -> np.ndarray:
    h = u_seq.shape[1]
    G = G_Generator(conv_tuple=conv_tuple, u_length=h)
    v_seq = (u_seq @ G) % 2
    return v_seq
```

Implementation of the ZJ-Stack as a Sequential Decoding Algorithm:

[1]:

```
LOOKUP_TABLE_METRIC_STACK = {(3, 1, 2): {'Bit Equality': +1, 'Bit Inequality':␣
 ↪-5}}
```

[2]:

```
def Bit_Metric_Calculator(conv_tuple: tuple, partial_r: str, partial_v: str) ->␣
 ↪int:

    metric_dict = LOOKUP_TABLE_METRIC_STACK[conv_tuple]
    b, a = metric_dict['Bit Equality'], metric_dict['Bit Inequality']
    partial_r_list, partial_v_list = list(partial_r), list(partial_v)
    partial_r_ndarray, partial_v_ndarray = np.array(partial_r_list), np.
 ↪array(partial_v_list)
    result = (partial_r_ndarray == partial_v_ndarray)
    new_result = (b - a)*(result) + a
    output = int(new_result.sum())

    return output
```

[3]:
```
def u_str_to_v_str_Generator(conv_tuple, u: str) -> str:

    u_list = []
    for element in u:
        u_list.append(int(element))

    u_seq = np.array([u_list], dtype=np.int64)
    v_seq = Coder(conv_tuple=conv_tuple, u_seq=u_seq)

    v_seq_str = ''
    for element in v_seq[0]:
        v_seq_str += str(element)

    return v_seq_str
```

[4]:
```
def Partial_Generator(u: str, output: str, n: int) -> str:

    u_len = len(u)
    partial_output = output[: u_len*n]

    return partial_output
```

[5]:
```
def Dict_Sorter(input_dict: dict) -> dict:

        sorted_items = sorted(input_dict.items(), key=lambda item: (-item[1],␣
 ↪-len(item[0])))
        sorted_dict = dict(sorted_items)

        return sorted_dict
```

[6]:
```
def u_Dict_Updater(u_metric: dict, conv_tuple: tuple, r: str, h: int) -> dict:

    n = conv_tuple[0]
    u_metric_sorted = Dict_Sorter(u_metric)
    selected_key = list(u_metric_sorted.keys())[0]
    u_len = len(selected_key)

    if u_len < h:

        key_plus_0 = selected_key + '0'
        key_plus_1 = selected_key + '1'
        v_0 = u_str_to_v_str_Generator(conv_tuple=conv_tuple, u=key_plus_0)
        partial_v_0 = Partial_Generator(u=key_plus_0, output=v_0, n=n)
        v_1 = u_str_to_v_str_Generator(conv_tuple=conv_tuple, u=key_plus_1)
        partial_v_1 = Partial_Generator(u=key_plus_1, output=v_1, n=n)
        partial_r = Partial_Generator(u=key_plus_0, output=r, n=n)
        key_plus_0_metric = Bit_Metric_Calculator(conv_tuple=conv_tuple,␣
↪partial_r=partial_r, partial_v=partial_v_0)
        key_plus_1_metric = Bit_Metric_Calculator(conv_tuple=conv_tuple,␣
↪partial_r=partial_r, partial_v=partial_v_1)
        u_metric_sorted[key_plus_0] = key_plus_0_metric
        u_metric_sorted[key_plus_1] = key_plus_1_metric

    elif u_len >= h:

        key_plus_0 = selected_key + '0'
        v_0 = u_str_to_v_str_Generator(conv_tuple=conv_tuple, u=key_plus_0)
        partial_v_0 = Partial_Generator(u=key_plus_0, output=v_0, n=n)
        partial_r = Partial_Generator(u=key_plus_0, output=r, n=n)
        key_plus_0_metric = Bit_Metric_Calculator(conv_tuple=conv_tuple,␣
↪partial_r=partial_r, partial_v=partial_v_0)
        u_metric_sorted[key_plus_0] = key_plus_0_metric

    output_dict = Dict_Sorter(u_metric_sorted)
    del output_dict[selected_key]

    return output_dict
```

```
[7]: def Stack_Decoder(conv_tuple: tuple, r: str) -> tuple:

    # Initilization Part:
    stack_dict = {}
    u_initial_list = ['0', '1']
    n = conv_tuple[0]
    m = conv_tuple[2]
    h = int((len(r) - (n*m)) / n)
    for u in u_initial_list:

        u_len = len(u)
        partial_r = Partial_Generator(u=u, output=r, n=n)
        v_path = u_str_to_v_str_Generator(conv_tuple=conv_tuple, u=u)
        partial_v = Partial_Generator(u=u, output=v_path, n=n)
        value_initial = Bit_Metric_Calculator(conv_tuple=conv_tuple,␣
↪partial_r=partial_r, partial_v=partial_v)
        stack_dict[u] = value_initial

    stack_dict = Dict_Sorter(input_dict=stack_dict)

    step = 1
    print(f'\n{colored("Step ", "blue", attrs=["bold"])}{colored(f"{step}:",␣
↪"blue", attrs=["bold"])}\n\n{colored(f"Stack = ", "green",␣
↪attrs=["bold"])}{stack_dict}\n')
    while True:
        step += 1
        stack_dict = u_Dict_Updater(u_metric=stack_dict, conv_tuple=conv_tuple, r=r, h=h)
        u_first = list(stack_dict.keys())[0]
        print(f'\n{colored("Step ", "blue", attrs=["bold"])}{colored(f"{step}:", "blue", attrs=["bold"])}\n\n{colored(f"Stack = ", "green", attrs=["bold"])}{stack_dict}\n')
        if len(u_first) == h + m:
            break

    decoded_u = list(stack_dict.keys())[0]
    decoded_u = decoded_u[: -m]
    v_hat = u_str_to_v_str_Generator(conv_tuple=conv_tuple, u=decoded_u)
    return decoded_u, v_hat
```

- Test 1:

[14]:

```
r = '010010001110100101011'
u_decoded, v_hat = Stack_Decoder((3, 1, 2), r=r)
print(f'\n{colored(f"Final Result of the ZJ-Stack Algorithm: ", "red", attrs=["bold"])}\n\n{colored(f"r = ", "black", attrs=["bold"])}\
{colored(f"{r}", "black", attrs=["bold"])}\n\n{colored(f"v-hat = ", "black", attrs=["bold"])}\
{colored(f"{v_hat}", "black", attrs=["bold"])}\n\n{colored(f"u-hat = ", "black", attrs=["bold"])}{colored(f"{u_decoded}", "black", attrs=["bold"])}\n')
```

```
Step 1:

Stack = {'0': -3, '1': -9}


Step 2:

Stack = {'00': -6, '1': -9, '01': -12}


Step 3:

Stack = {'000': -9, '1': -9, '01': -12, '001': -15}
```

```
Step 4:

Stack = {'1': -9, '0001': -12, '01': -12, '001': -15, '0000': -18}


Step 5:

Stack = {'11': -6, '0001': -12, '01': -12, '001': -15, '0000': -18,
'10': -24}


Step 6:

Stack = {'111': -3, '0001': -12, '01': -12, '001': -15, '0000':
-18, '110': -21, '10': -24}


Step 7:

Stack = {'1110': 0, '0001': -12, '01': -12, '001': -15, '0000':
-18, '1111': -18, '110': -21, '10': -24}


Step 8:

Stack = {'11101': 3, '0001': -12, '01': -12, '11100': -15, '001':
-15, '0000': -18, '1111': -18, '110': -21, '10': -24}


Step 9:

Stack = {'111010': 6, '0001': -12, '01': -12, '11100': -15, '001':
-15, '0000': -18, '1111': -18, '110': -21, '10': -24}


Step 10:

Stack = {'1110100': 9, '0001': -12, '01': -12, '11100': -15, '001':
-15, '0000': -18, '1111': -18, '110': -21, '10': -24}


Final Result of the ZJ-Stack Algorithm:

r = 010010001110100101011

v-hat = 111010001110100101011

u-hat = 11101
```

- Test 2:

```
[15]: r = '110110110111010101101'
      u_decoded, v_hat = Stack_Decoder((3, 1, 2), r=r)
      print(f'\n{colored(f"Final Result of the ZJ-Stack Algorithm: ", "red",␣
        ↪attrs=["bold"])}\n\n{colored(f"r = ", "black", attrs=["bold"])}\
      {colored(f"{r}", "black", attrs=["bold"])}\n\n{colored(f"v-hat = ", "black",␣
        ↪attrs=["bold"])}\
      {colored(f"{v_hat}", "black", attrs=["bold"])}\n\n{colored(f"u-hat = ",␣
        ↪"black", attrs=["bold"])}{colored(f"{u_decoded}", "black",␣
        ↪attrs=["bold"])}\n')
```

```
Step 1:

Stack = {'1': -3, '0': -9}


Step 2:

Stack = {'11': -6, '0': -9, '10': -12}


Step 3:

Stack = {'110': -3, '0': -9, '10': -12, '111': -21}


Step 4:

Stack = {'1100': -6, '0': -9, '1101': -12, '10': -12, '111': -21}


Step 5:

Stack = {'11000': -9, '0': -9, '1101': -12, '10': -12, '11001':
-15, '111': -21}


Step 6:

Stack = {'0': -9, '1101': -12, '10': -12, '11001': -15, '110000':
-18, '111': -21}
```

```
Step 7:

Stack = {'1101': -12, '10': -12, '01': -12, '11001': -15, '110000':
-18, '00': -18, '111': -21}


Step 8:

Stack = {'11011': -9, '10': -12, '01': -12, '11001': -15, '110000':
-18, '00': -18, '111': -21, '11010': -27}


Step 9:

Stack = {'10': -12, '01': -12, '11001': -15, '110000': -18,
'110110': -18, '00': -18, '111': -21, '11010': -27}


Step 10:

Stack = {'01': -12, '11001': -15, '101': -15, '110000': -18,
'110110': -18, '00': -18, '111': -21, '100': -21, '11010': -27}


Step 11:

Stack = {'11001': -15, '101': -15, '011': -15, '110000': -18,
'110110': -18, '00': -18, '111': -21, '100': -21, '010': -21, '11010': -27}


Step 12:

Stack = {'110010': -12, '101': -15, '011': -15, '110000': -18,
'110110': -18, '00': -18, '111': -21, '100': -21, '010': -21, '11010': -27}


Step 13:

Stack = {'101': -15, '011': -15, '110000': -18, '110110': -18,
'00': -18, '1100100': -21, '111': -21, '100': -21, '010': -21, '11010': -27}


Step 14:

Stack = {'011': -15, '110000': -18, '110110': -18, '1010': -18,
'00': -18, '1100100': -21, '111': -21, '100': -21, '010': -21, '1011': -24,
'11010': -27}


Step 15:

Stack = {'110000': -18, '110110': -18, '1010': -18, '0110': -18,
'00': -18, '1100100': -21, '111': -21, '100': -21, '010': -21, '1011': -24,
'0111': -24, '11010': -27}


Step 16:

Stack = {'110110': -18, '1010': -18, '0110': -18, '00': -18,
'1100100': -21, '111': -21, '100': -21, '010': -21, '1011': -24, '0111': -24,
'1100000': -27, '11010': -27}


Step 17:

Stack = {'1010': -18, '0110': -18, '00': -18, '1100100': -21,
'111': -21, '100': -21, '010': -21, '1011': -24, '0111': -24, '1100000': -27,
'1101100': -27, '11010': -27}


Step 18:

Stack = {'0110': -18, '00': -18, '1100100': -21, '10100': -21,
'111': -21, '100': -21, '010': -21, '1011': -24, '0111': -24, '1100000': -27,
'1101100': -27, '11010': -27, '10101': -27}


Step 19:

Stack = {'00': -18, '1100100': -21, '10100': -21, '01100': -21,
'111': -21, '100': -21, '010': -21, '1011': -24, '0111': -24, '1100000': -27,
'1101100': -27, '11010': -27, '10101': -27, '01101': -27}


Step 20:

Stack = {'1100100': -21, '10100': -21, '01100': -21, '111': -21,
'100': -21, '010': -21, '001': -21, '1011': -24, '0111': -24, '1100000': -27,
'1101100': -27, '11010': -27, '10101': -27, '01101': -27, '000': -27}


Final Result of the ZJ-Stack Algorithm:

r = 110110110111010101101
```

```
v-hat = 111010110011111101011

u-hat = 11001
```

Conclusion:

- As shown in the book in Example 13.5 and Example 13.6 results of the implementation are correct.

Miscellaneous References:

- Stack Algorithm