МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
“БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”
ИНТЕЛЛЕКТУАЛЬНЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

Отчёт
по дисциплине
Современные методы защиты компьютерных систем
по лабораторной работе №2
«Матричные итеративные коды»

Выполнил:
Студент группы ИИ-22
Борейша О.С.
Проверил:
Хацкевич А. С.

Брест 2024

Цель: приобретение практических навыков кодирования/декодирования двоичных данных при использовании итеративных кодов.

Задачи:

1. Закрепить теоретические знания по использованию итеративных кодов для повышения надежности передачи и хранения в памяти компьютера двоичных данных.
2. Разработать приложение для кодирования/декодирования двоичной информации итеративным кодом с различной относительной избыточностью кодовых слов.
3. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.
4. Ответить на контрольные вопросы

Код программы:

```
import numpy as np
import random


def generate_random_binary_string(length: int) -> str:
    return ''.join(random.choice(['0', '1']) for _ in range(length))


def flip_bit(binary_string: str, position: int) -> str:
    if position < 0 or position >= len(binary_string):
        raise ValueError("Position out of range")

    # Изменяем символ на противоположный
    flipped_bit = '1' if binary_string[position] == '0' else '0'

    # Собираем новую строку с заменой символа
    return binary_string[:position] + flipped_bit + binary_string[position + 1:]


def find_difference_position(str1, str2):
    if len(str1) != len(str2):
        return "Strings must be same length"
    for i in range(len(str1)):
        if str1[i] != str2[i]:
            return i
    return -1


def binary_string_to_array(binary_string, k1, k2, z=1):
    if z == 1:
        total_elements = k1 * k2
    else:
        total_elements = k1 * k2 * z
    if len(binary_string) < total_elements:
        raise ValueError(f"Недостаточно данных в строке для заполнения матриц:
требуется {total_elements} элементов.")
    binary_list = [int(bit) for bit in binary_string[:total_elements]]
    if z == 1:
        array = np.array(binary_list).reshape(k1, k2)
    else:
        array = np.array(binary_list).reshape(z, k1, k2)
    return array
```

```
def get_control_bits_2d(matrix, g):
    if g == 2:
        row_sums_mod2 = np.sum(matrix, axis=1) % 2
        col_sums_mod2 = np.sum(matrix, axis=0) % 2
        row_str = ''.join(map(str, row_sums_mod2))
        col_str = ''.join(map(str, col_sums_mod2))
        # print(f'row{row_sums_mod2}\ncol{col_sums_mod2}')
        return row_str, col_str
    elif g == 3:
        row_sums_mod2 = np.sum(matrix, axis=1) % 2
        col_sums_mod2 = np.sum(matrix, axis=0) % 2

        m, n = matrix.shape
        diagonal_sums = []

        for d in range(m + n - 1):
            diagonal_sum = 0
            for i in range(m):
                j = d - i
                if 0 <= j < n:
                    diagonal_sum += matrix[i, j]
            diagonal_sums.append(diagonal_sum % 2)

        row_str = ''.join(map(str, row_sums_mod2))
        col_str = ''.join(map(str, col_sums_mod2))
        diag_str = ''.join((map(str,diagonal_sums)))
        # print(f'row{row_sums_mod2}\ncol{col_sums_mod2}\ndiag{diagonal_sums}')
        return row_str, col_str, diag_str


def get_control_bits_3d(matrix):
    z_size = matrix.shape[0]
    z_sums = []
    sums_with_layer_info = []
    for i in range(z_size):
        matrix_sum = np.sum(matrix[i])
        mod2_sum = matrix_sum % 2
        row_sums = np.sum(matrix[i], axis=1) % 2
        col_sums = np.sum(matrix[i], axis=0) % 2
        sums_with_layer_info.append((i, ''.join(map(str, row_sums)),
''.join(map(str, col_sums))))
        z_sums.append(mod2_sum)
    z_paritet = ''.join(map(str, z_sums))
    # print(f'z-paritet: {z_paritet}\nsums: {sums_with_layer_info}')
    return z_paritet, sums_with_layer_info


def encode(data, k1, k2, z, g):
    arr = binary_string_to_array(data, k1, k2, z)
    print(f'Matrix:\n{arr}')
    if z == 1:
        if g == 2:
            row, col = get_control_bits_2d(arr, g)
            encoded_data = data + row + col
            return encoded_data, row + col
        elif g == 3:
            row, col, diag = get_control_bits_2d(arr, g)
            encoded_data = data + row + col + diag
            return encoded_data, row+col+diag
    else:
        if g == 3:
            z_paritet, sums = get_control_bits_3d(arr)
            sums_str = ''
            for layer_info in sums:
```

```
                layer_index, row_sums, col_sums = layer_info
                sums_str += row_sums
                sums_str += col_sums
            encoded_data = data + z_paritet + sums_str
            return encoded_data, z_paritet + sums_str


def decode(data, k1, k2, z, g):
    if z == 1:
        if g == 2:
            inf_bits_len = k1 * k1 * z
            inf_bits = data[:inf_bits_len]
            row_control_bits = data[inf_bits_len:inf_bits_len + k1]
            col_control_bits = data[inf_bits_len + k1:inf_bits_len + k1 + k2]
            arr = binary_string_to_array(data, k1, k2, z)
            print(f'Accepted matrix:\n{arr}')
            new_row, new_col = get_control_bits_2d(arr, g)
            print(f'inf_bits: {inf_bits}\nrow: {row_control_bits}\ncol: {col_con-
trol_bits}')
            print(f'new_row: {new_row}\nnew_col: {new_col}')
            diff_row = find_difference_position(new_row, row_control_bits)
            diff_col = find_difference_position(new_col, col_control_bits)
            # print(f'diff_row - {diff_row}, diff_col - {diff_col}')
            if diff_col+1 and diff_row+1:
                print(f'Error on position [{diff_row + 1},{diff_col + 1}]')
            elif diff_col == -1 and diff_row == -1:
                print('Errors not found.')
        elif g == 3:
            inf_bits_len = k1*k1*z
            inf_bits = data[:inf_bits_len]
            row_control_bits = data[inf_bits_len:inf_bits_len + k1]
            col_control_bits = data[inf_bits_len + k1:inf_bits_len + k1 + k2]
            diag_control_bits = data[inf_bits_len + k1 + k2:]
            arr = binary_string_to_array(data, k1, k2, z)
            print(f'Accepted matrix:\n{arr}')
            new_row, new_col, new_diag = get_control_bits_2d(arr, g)
            print(f'inf_bits: {inf_bits}\nrow: {row_control_bits}\ncol: {col_con-
trol_bits}\ndiag: {diag_control_bits}')
            print(f'new_row: {new_row}\nnew_col: {new_col}\nnew_diag: {new_diag}')
            diff_row = find_difference_position(new_row, row_control_bits)
            diff_col = find_difference_position(new_col, col_control_bits)
            if diff_col+1 and diff_row+1:
                print(f'Error on position [{diff_row+1},{diff_col+1}]')
            elif diff_col == -1 and diff_row == -1:
                print('Errors not found.')
    else:
        if g == 3:
            inf_bits_len = k1 * k1 * z
            inf_bits = data[:inf_bits_len]
            z_paritet = data[inf_bits_len:inf_bits_len + z]
            remaining_data = data[inf_bits_len + z:]
            sums = [remaining_data[i * (k1 + k2):(i + 1) * (k1 + k2)] for i in
range(z)]
            arr = binary_string_to_array(inf_bits, k1, k2, z)
            print(f'Accepted matrix:\n{arr}')
            new_z_paritet, new_sums = get_control_bits_3d(arr)
            print(f'inf_bits: {inf_bits}\nz_paritet: {z_paritet}')
            print(f'new_z_paritet: {new_z_paritet}')
            diff_z = find_difference_position(z_paritet, new_z_paritet)
            if diff_z+1:
                print(f'Error found on {diff_z+1} layer ', end='')
                row_diff = find_difference_position(sums[diff_z][:k1],
new_sums[diff_z][1])
                col_diff = find_difference_position(sums[diff_z][k1:],
new_sums[diff_z][2])
```

```
                print(f'on position [{row_diff+1}, {col_diff+1}]')
            elif diff_z == -1:
                print('Errors not found.')


def main():
    k1, k2, z, g = 3, 3, 5, 3
    input_data_example = '0110101111010001'
    input_data = generate_random_binary_string(k1*k2*z)
    print(f'Input data: {input_data}')
    encoded_data, control_bits = encode(input_data, k1, k2, z, g)
    print(f'Encoded data: {encoded_data}\nControl bits: {control_bits}')
    corruted_data = flip_bit(encoded_data, 20)
    print(f'Corrupted encoded data: {corruted_data}')
    decode(encoded_data, k1, k2, z, g)


if __name__ == '__main__':
    main()
```

Результат работы:

```
Input data: 000000100110100100011111000000001110000001011
Matrix:
[[[0 0 0]
  [0 0 0]
  [1 0 0]]

 [[1 1 0]
  [1 0 0]
  [1 0 0]]

 [[0 1 1]
  [1 1 1]
  [0 0 0]]

 [[0 0 0]
  [0 0 1]
  [1 1 0]]

 [[0 0 0]
  [0 0 1]
  [0 1 1]]]
Encoded data: 000000100110100100011111000000001110000001011101110011000111100101000101110100 10
Control bits: 10111001100011110010100010111010010
```

```
Corrupted encoded data: 0000001001101001000101110000000011100000010111011100110001111001010001011101001 0
Accepted matrix:
[[[0 0 0]
  [0 0 0]
  [1 0 0]]

 [[1 1 0]
  [1 0 0]
  [1 0 0]]

 [[0 1 1]
  [1 1 1]
  [0 0 0]]

 [[0 0 0]
  [0 0 1]
  [1 1 0]]

 [[0 0 0]
  [0 0 1]
  [0 1 1]]]
inf_bits: 000000100110100100011111000000001110000001011
z_paritet: 10111
new_z_paritet: 10111
Errors not found.
```

Вывод: приобрёл практические навыки кодирования/декодирования двоичных данных при использовании итеративных кодов.