**Міністерство освіти і науки України**  
**Національний технічний університет України**  
**“Київський політехнічний інститут”**

Інститут Прикладного системного аналізу  
Кафедра Системного проектування

Лабораторна робота з дисципліни  
«Теорія інформації та кодування»

на тему  
**«ДОСЛІДЖЕННЯ ВЛАСТИВОСТЕЙ ТА ОСНОВНИХ ХАРАКТЕРИСТИК КОДІВ ХЕММІНГА»**

Виконав:

студент 2 курсу

групи ДА-92

Насікан Д. Ю.

Київ 2021

**Мета роботи -** дослідження алгоритмів побудови кодів Хеммінга та їх можливостей виявлення та корекції помилок при використанні в інформаційних технологіях.

**ЗАВДАННЯ**

1. Ознайомитись з принципами побудови та основними характеристиками кодів Хемммінга.
2. Розробити алгоритми кодування текстових повідомлень з використанням коду Хемммінга з мінімальною кодовою відстанню dmin =3 і dmin =4.
3. Розробити алгоритми декодування текстових повідомлень з використанням коду Хемммінга з мінімальною кодовою відстанню dmin =3 і dmin =4.
4. Реалізувати розроблені алгоритми на мові програмування високого рівня. В програмах передбачити можливість вводу текстових повідомлень вручну з клавіатури, а також з підготовлених заздалегідь файлів.
5. Підготувати текстовий файл з наступним змістом: Своє прізвище, ім'я, по батькові, рік та місце народження, група, який навчальний заклад закінчив та ін. Зберегти файл у форматі з підримкою кодової таблиці Windows 1251.
6. Перевірити коректність роботи розроблених кодерів і декодерів для кодів Хемммінга (п.4 та п.5) без внесення помилок.
7. Модифікувати програми кодерів і декодерів для забезпечення можливості внесення вручну однократних та багатократних помилок в текстові повідомлення. Провести тестування декодерів при внесенні однократних помилок в текстові повідомлення.
8. Провести тестування кодерів і декодерів при внесенні вручну багатократних помилок (2, 3, 4, 5 ....) в текстові повідомлення .Результати тестування задокументувати.
9. Модифікувати програми кодерів і декодерів для забезпечення можливості автоматичного внесення однократних та багатократних помилок в файли з текстовими повідомленнями. Кількість помилок повинна вводитись в діалоговому режимі. Провести тестування на файлах довільної довжини і кількості помилок.
10. **ХІД РОБОТИ**
11. Розробимо алгоритми кодування текстових повідомлень з використанням коду Хемммінга з мінімальною кодовою відстанню dmin =3 і dmin =4:

Кодування для dmin=3:

Спочатку усі символи з вхідної строки переводимо у бінарний код (бінарне представлення). Всі бінарні коди для кожного символу займають по 8 біт. Після цього потрібно вставити контрольні біти (коди Хемінга) на місця з номерами 2n, і в залежності від вхідного розміру інформаційного «слова», будемо мати (n+1) контрольні біти, тобто довжина вхідного інформаційного слова в нас збільшиться на (n+1) біт. Спочатку усі контрольні біти мають значення — 0. Тепер розрахуємо значення кожного контрольного біта за таким алгоритмом: значення кожного контрольного біта залежить від значень інформаційних біт, проте, не від всіх, а тільки від тих, які цей контрольний біт контролює. Для того, щоб зрозміти, за які біти відповідає кожен контрольний біт необхідно зрозуміти просту закономірність: контрольний біт з номером N контролює усі наступні N біт через кожні N біт, починаючи з позиції N. Після цього розуміння можна вже легко визначити значення кожного контрольного біта: беремо кожний контрольний біт і дивимося, скільки серед контролюємих їм біт одиниць, рахуємо кількість цих одиниць, і якщо ця кількість відповідає парному числу — контрольний біт приймає значення 0, а якщо непарному — контрольний біт приймає зачення 1. Проте, можна й навпаки брати: для парних — 1, а для непарних — 1. Головне, щоб в частинах кодування і декодування був однаковий. (В даній програмі вважається, що для парних — контрольний біт примає значення 0, а для непарних — значення 1).

Кодування для dmin=4:

Спочатку усі символи з вхідної строки переводимо у бінарний код (бінарне представлення). Всі бінарні коди для кожного символу займають по 8 біт. Після цього потрібно вставити контрольні біти (коди Хемінга) на місця з номерами 2n, і в залежності від вхідного розміру інформаційного «слова», будемо мати (n+1) контрольні біти, тобто довжина вхідного інформаційного слова в нас збільшиться на (n+1) біт. Спочатку усі контрольні біти мають значення — 0. Тепер розрахуємо значення кожного контрольного біта за таким алгоритмом: значення кожного контрольного біта залежить від значень інформаційних біт, проте, не від всіх, а тільки від тих, які цей контрольний біт контролює. Для того, щоб зрозміти, за які біти відповідає кожен контрольний біт необхідно зрозуміти просту закономірність: контрольний біт з номером N контролює усі наступні N біт через кожні N біт, починаючи з позиції N. Після цього розуміння можна вже легко визначити значення кожного контрольного біта: беремо кожний контрольний біт і дивимося, скільки серед контролюємих їм біт одиниць, рахуємо кількість цих одиниць, і якщо ця кількість відповідає парному числу — контрольний біт приймає значення 0, а якщо непарному — контрольний біт приймає зачення 1. Проте, можна й навпаки брати: для парних — 1, а для непарних — 1. Головне, щоб в частинах кодування і декодування був однаковий. (В даній програмі вважається, що для парних — контрольний біт примає значення 0, а для непарних — значення 1).

Після цього нам потрібно порахувати значення елемента КР0, який надає можливість контролювати всю кодову комбінацію на парність. Для того, щоб визначити парність, потрібно скласти усі одиниці нашого коду, який ми отримали вже після розрахунку контрольних бітів. Якщо ця сума парна, то КР0 приймає значення 1, якщо непарна — значення 0.

1. Розробимо алгоритми декодування текстових повідомлень з використанням коду Хемммінга з мінімальною кодовою відстанню dmin =3 і dmin =4:

Для dmin=3 можна знайти лише одну помилку. Отже, для цього перевіряються контрольні біти: беремо отриману закодовану строку, і знову рахуємо усі контрольні біти, якщо вони всі мають 0, отже, це так званий «синдром нуля», що означає, що помилок немає. В нашій програмі можна обрати позицію для помилки, знайти та виправити її. Отже, якщо при перевірці контрольних біт, хоча б один має ненульове значення, то це означає, що помилка є. Для того, щоб знайти позицію цієї помилки достатньо додати номери позицій наших отриманих «помилкових контрольних бітів» (які в останньом розрахунку мали ненульові значення), і це значення їх суми буде номером позиції «пошкодженого біта», отже, місце помилки знайдено. А оскільки, це двійковий код, то виправлення помилкового біта зводиться до інверсії значення цього біта.

1. Реалізуємо розроблені алгоритми на мові програмування python. В програмах передбачити можливість вводу текстових повідомлень вручну з клавіатури, а також з підготовлених заздалегідь файлів.

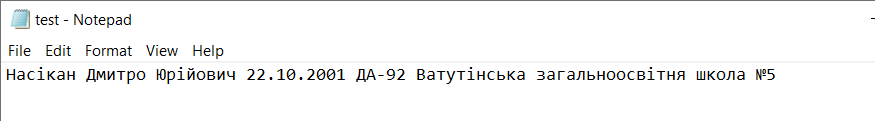
File main.py:

from bitarray import bitarray  
from functools import reduce  
from random import randint  
  
  
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////]def encode\_file(file\_name, extension, errors\_1, errors\_2):  
 with open(f'{file\_name}.{extension}', 'rb') as input\_file:  
 with open(f'{file\_name}-{extension}.lab4', 'wb') as output\_file:  
  
 while True:  
 bytes\_from\_file = bitarray()  
 raw\_bytes = input\_file.read(15)  
  
 if len(raw\_bytes) == 0:  
 break  
  
 bytes\_from\_file.frombytes(raw\_bytes)  
  
 bytes\_to\_file = extended\_hamming\_encode(bytes\_from\_file)  
 bytes\_to\_file\_with\_errors = generate\_errors(bytes\_to\_file, errors\_1, errors\_2)  
 bytes\_to\_file\_with\_errors.tofile(output\_file)  
  
  
def generate\_errors(bit\_arr, errors\_1, errors\_2):  
 if errors\_1:  
 error\_i = randint(0, len(bit\_arr) - 1)  
 bit\_arr[error\_i] = not bit\_arr[error\_i]  
  
 if errors\_2:  
 error\_i = randint(0, len(bit\_arr) - 1)  
 bit\_arr[error\_i] = not bit\_arr[error\_i]  
  
 return bit\_arr  
  
  
def decode\_file(file\_name, extension):  
 with open(f'{file\_name}.lab4', 'rb') as input\_file:  
 with open(f'{file\_name}-dec.{extension}', 'wb') as output\_file:  
  
 while True:  
 bytes\_from\_file = bitarray()  
 raw\_bytes = input\_file.read(16)  
  
 if len(raw\_bytes) == 0:  
 break  
  
 bytes\_from\_file.frombytes(raw\_bytes)  
  
 bytes\_to\_file = extended\_hamming\_decode(bytes\_from\_file)  
 bytes\_to\_file.tofile(output\_file)  
  
  
# використовуємо блоки довжиною 16 біт. 11 бітів з даними, 5 допоміжних  
def extended\_hamming\_encode(bit\_arr):  
 if len(bit\_arr) < 120:  
 print('EOF')  
 return bit\_arr  
  
 encoded\_arr = hamming\_encode(bit\_arr, 7)  
 # записуємо в перший біт парність закодованого повідомлення  
 encoded\_arr.insert(0, reduce(lambda x, y: x ^ y, encoded\_arr))  
  
 return encoded\_arr  
  
  
def extended\_hamming\_decode(bit\_arr):  
 if len(bit\_arr) < 128:  
 print('EOF')  
 return bit\_arr  
  
 hamming\_coded\_arr = bit\_arr[1:]  
 parity\_bit = bit\_arr[0]  
 decoded\_arr, error = hamming\_decode(hamming\_coded\_arr[:], 7)  
 parity = reduce(lambda x, y: x ^ y, hamming\_coded\_arr)  
  
 if error is not None and parity == parity\_bit:  
 raise Exception(f'There is 2 errors in the sequence {bit\_arr}, so it cannot be decoded.')  
  
 return decoded\_arr  
  
  
def hamming\_encode(bit\_arr, max\_pow):  
 for i in range(0, max\_pow):  
 bit\_arr.insert(2 \*\* i - 1, 0)  
  
 print(bit\_arr)  
  
 step = 2  
 for i in [2 \*\* j - 1 for j in range(0, max\_pow)]:  
 parity\_check\_bit\_list = []  
 parity\_bit\_index = i  
 while i < len(bit\_arr):  
 # print(f'Taken {list(val for val in bit\_arr[i:i + step // 2])} which is bit\_arr[{i}:{i + step // 2}]')  
  
 parity\_check\_bit\_list.extend(val for val in bit\_arr[i:i + step // 2])  
  
 i += step  
  
 parity = reduce(lambda x, y: x ^ y, parity\_check\_bit\_list)  
 bit\_arr[parity\_bit\_index] = parity  
 step \*= 2  
  
 print(bit\_arr)  
 return bit\_arr  
  
  
def hamming\_decode(bit\_arr, max\_pow):  
 step = 2  
  
 error\_bit\_index = None  
 for i in [2 \*\* j - 1 for j in range(0, max\_pow)]:  
 parity\_check\_bit\_list = []  
 parity\_bit\_index = i  
 while i < len(bit\_arr):  
 # print(f'Taken {list(val for val in bit\_arr[i:i + step // 2])} which is bit\_arr[{i}:{i + step // 2}]')  
 parity\_check\_bit\_list.extend(val for val in bit\_arr[i:i + step // 2])  
 i += step  
  
 parity\_check\_bit\_list[0] = 0  
 parity = reduce(lambda x, y: x ^ y, parity\_check\_bit\_list)  
 if parity != bit\_arr[parity\_bit\_index]:  
 if error\_bit\_index is None:  
 error\_bit\_index = parity\_bit\_index  
 else:  
 error\_bit\_index += parity\_bit\_index + 1  
  
 step \*= 2  
  
 if error\_bit\_index is None:  
 print('There is no errors in the sequence.')  
 else:  
 print(f'The error was found on the position: {error\_bit\_index}')  
 bit\_arr[error\_bit\_index] = not bit\_arr[error\_bit\_index]  
 print(f'Corrected array of bits: {bit\_arr}')  
  
 for i in reversed(range(0, max\_pow)):  
 del bit\_arr[2 \*\* i - 1]  
  
 return bit\_arr, error\_bit\_index  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 ha = '0100010000111101'  
 br = '0011111001001000'  
 t1 = '10011010'  
  
 a = bitarray(t1)  
 print(f'Given array of bits: {a}')  
  
 enc\_bit\_arr = hamming\_encode(a[:], 3)  
 print(f'Encoded array of bits: {enc\_bit\_arr}')  
  
 print(f'Generating random error...')  
 error\_index = randint(0, len(enc\_bit\_arr) - 1)  
 enc\_bit\_arr[error\_index] = not enc\_bit\_arr[error\_index]  
 print(f'Random error generated on position {error\_index}.')  
 print(f'Bit array with error: {enc\_bit\_arr}')  
  
 decoded\_bit\_arr, err\_index = hamming\_decode(enc\_bit\_arr, 3)  
 print(f'Initial array of bits: {decoded\_bit\_arr}')  
  
 print(a)  
 print(decoded\_bit\_arr)

File test.py:

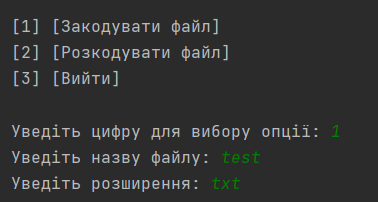
from main import encode\_file, decode\_file  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 while True:  
 print('\n[1] [Закодувати файл]\n[2] [Розкодувати файл]\n[3] [Вийти]\n')  
 option = input('Уведіть цифру для вибору опції: ')  
  
 if option == '1':  
 f\_n = input('Уведіть назву файлу: ')  
 ext = input('Уведіть розширення: ')  
 generate\_1\_errors = bool(input('Згенерувати помилки: ')) # введіть шось, щоб згенерувати, або нічого  
  
 if generate\_1\_errors:  
 generate\_2\_errors = bool(input('Згенерувати 2 помилки(код буде неможливо декодувати): '))  
 else:  
 generate\_2\_errors = False  
  
 encode\_file(f\_n, ext, generate\_1\_errors, generate\_2\_errors)  
  
 elif option == '2':  
 f\_n = input('Уведіть назву файлу: ')  
 decode\_file(f\_n, f\_n.split('-')[1])  
  
 elif option == '3':  
 break

1. Підготуємо текстовий файл з наступним змістом: Своє прізвище, ім'я, по батькові, рік та місце народження, група, який навчальний заклад закінчив та ін. Зберегти файл у форматі з підримкою кодової таблиці Windows 1251:



1. Перевіримо коректність роботи розроблених кодерів і декодерів для кодів Хемммінга (п.4 та п.5) без внесення помилок:

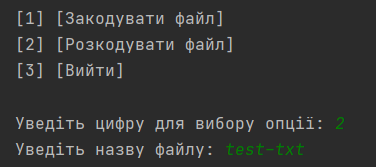
Закодуємо даний файл користуючись кодами Хемінга:



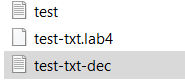
Як бачимо, з’явився новий закодований файл:

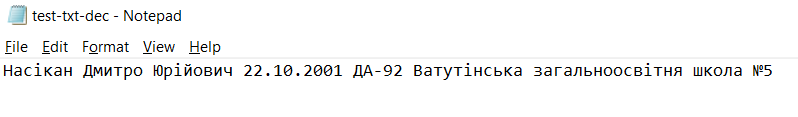


Тепер розкодуємо файл:

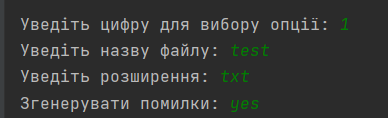


Як бачимо, файл розкодовано успішно, помилок немає:



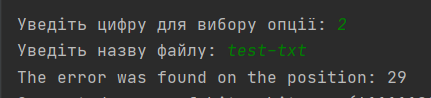


1. Проведемо тестування декодерів при внесенні однократних та багатократних помилок:

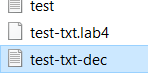


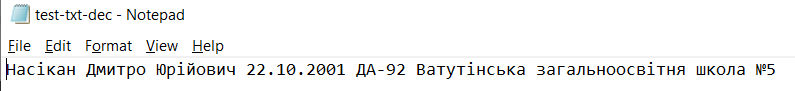
Згенерований файл:



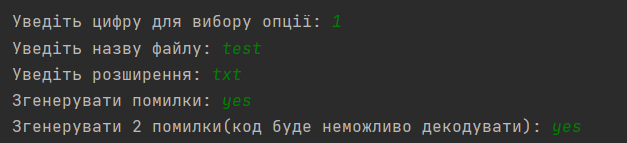


Як бачимо, корекція помилок працює успішно, розкодований файл має такий самий вигляд, як і вхідний:





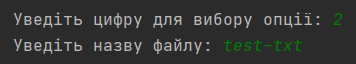
Тепер проведемо тестування внісши 2 помилки:

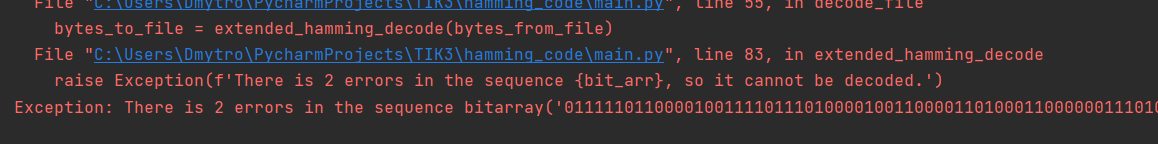


Отримуємо файл:



Спробуємо розкодувати цей файл:





Як бачимо, програма виявила наявність двох помилок і зазначила, що файл не може бути розкодованим.

**ВИСНОВКИ**

У ході даної лабораторної роботи мною були вивчені особливості кодів Хемінга, розроблений та реалізований алгоритм кодування послідовності бітів кодами Хемінга. Програма була протестована на файлах різної довжини, та спроможна коректувати код, що містить одноразову помилку та виявляти дворазові помилки.