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

Інститут Прикладного системного аналізу  
Кафедра Системного проектування

**ЗВІТ**

**Про виконання самостійної роботи**

**з дисципліни  
«Теорія інформації та кодування»**

**на тему:**«АНАЛІЗ ФОРМАТІВ ДАНИХ З ВИКОРИСТАННЯМ АЛГОРИТМІВ СТИСНЕННЯ ДАНИХ БЕЗ ВТРАТ»

Студента 2 курсу

групи ДА-92

Насікана Д. Ю.

Керівник:

доц. к. т. н.

Капшук О. О.

Київ 2021

**ЗМІСТ**

1. **ВСТУП..........................................................................................................3**
2. **ЗАГАЛЬНИЙ ОГЛЯД СТИСНЕННЯ БЕЗ ВТРАТ.............................5**
3. **АГЛОРИТМ ХАФФМАНА......................................................................7**
   1. ***Алгоритми кодування та декодування повідомлення................*7**
   2. ***Побудова дерева Хаффмана.............................................................*8**
   3. ***Недоліки алгоритму Хаффмана.....................................................*8**
4. **АЛГОРИТМ LZW.....................................................................................10**
   1. ***Процес кодування.............................................................................*10**
   2. ***Процес декодування.........................................................................*11**
5. **ФОРМАТИ ДАНИХ, ЩО ВИКОРИСТОВУЮТЬ АЛГОРИТМИ СТИСНЕННЯ БЕЗ ВТРАТ.....................................................................13**

***5.1. Універсальний формат PAQ..........................................................*14**

***5.2. Формат стиснення звуку FLAC....................................................*15**

***5.3. Стиснення графіки у GIF...............................................................*16**

***5.4. Стиснення графіки у JPEG-LS......................................................*17**

***5.5. Універсальні формати даних.........................................................*18**

***5.6. Формати кодування аудіо..............................................................*19**

***5.7. Формати кодування графічних файлів........................................*20**

***5.8. Кодування відеофайлів....................................................................*21**

1. **ТЕСТУВАННЯ ТА ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ФОРМАТІВ ДАНИХ СТИСНЕННЯ БЕЗ ВТРАТ.....................................................22**

***6.1. Тестування форматів ZIP, 7-ZIP, RAR та PAQ........................*22**

***6.2. Тестування аудіоформатів ALAC, DST, FLAC, WavPack.........*25**

***6.3. Тестування растрових форматів PNG, JPEG-LS та FLIF.....*26**

1. **ВИСНОВКИ...............................................................................................28**
2. **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..............................................31**
3. **ДОДАТКИ..................................................................................................32**
4. **ВСТУП**

Стиснення даних - це алгоритмічне перетворення даних задля зменшення їх обсягу. Стиснення використовується для більш раціонального використання пристроїв зберігання і передачі даних й засноване на усуненні надмірності, що міститься у вихідних даних. Найпростішим прикладом надмірності є повторення фрагментів в тексті. Інший тип надмірності пов'язаний з тим, що деякі значення в даних зустрічаються частіше, ніж інші. Скорочення обсягу даних досягається шляхом заміни часто зустрічаються даних короткими кодовими словами, а рідкісні дані - довгими (ентропійне кодування)[3].

У наш час, коли з кожним днем людство продукує все більше й більше цифрової інформації, постає питання про те, як зберігати та передавати інформацію таким чином, щоб вона займала найменше місця на цифрових носіях або використовувала найменше часу та ресурсів для передачі. Цілком зрозуміло, що стиснення даних відіграє дуже важливу роль у цьому питанні, так як воно призводить до зменшення розміру файлу, а відповідно, до оптимізації зберігання та передачі.

Загалом, розрізняють два види стиснення: стиснення без втрат й із втратами. При використанні першого виду інформація стискається таким чином, що при розтисненні (декомпресії) вона відновлюється повністю та є точною копією вхідної інформації. Стиснення з утратами, навпаки, дозволяє лише відновлення даних, які є тільки наближенням до початкових даних. Стиснення без втрат використовується, коли важливо, щоб відновленні дані були ідентичні оригіналу. Типовий приклад — виконуваний файл або джерельний код, текстовий файл.

**Мета роботи** полягає у дослідженні основних алгоритмів стиснення без втрат, таких як алгоритм Хаффмана та LZW, аналізі та дослідженні форматів даних, які використовують алгоритми стиснення без втрат, таких як ZIP, RAR, FLAC та PNG.

**Завдання:** вивчити та описати принцип роботи основних алгоритмів стиснення без втрат, таких як алгоритм Хаффмана та LZW, розробити програми, що реалізують стиснення файлів довільних розширень цими алгоритмами, визначити найбільш широковикористовувані формати стиснення даних без втрат, дослідити та проаналізувати роботу таких форматів даних стиснення без втрат, як ZIP, RAR, FLAC та PNG.

1. **ЗАГАЛЬНИЙ ОГЛЯД АЛГОРИТМІВ СТИСНЕННЯ БЕЗ ВТРАТ**

У загальних рисах значення стиснення без втрат полягає в пошуку закономірності в початкових даних і з її урахуванням генерації іншої послідовності, яка повністю описує початкову. Наприклад, для кодування бінарних послідовностей, в яких багато нулів та мало одиниць, ми можемо використати таку заміну:

У такому випадку при кодуванні послідовності у 16 бітів:

Отримаємо послідовність з 13 бітів:

Більшість алгоритмів стиснення без втрат працюють у дві стадії: на першій генерується статистична модель для вхідних даних, що надає інформацію про те, як саме розподілені імовірності появи тих чи інших символів, і які символи з’являються найчастіше, друга - представляє вхідні дані в бітовому вигляді, використовуючи побудовану модель для того, щоб мінімізувати довжину кодових послідовностей, які трапляються найчастіше.[10]

Алгоритми стиснення без втрат не можуть гарантувати стиснення для усіх видів вхідних даних. Іншими словами, для будь-якого алгоритму стиснення без втрат, існує такий набір вхідних даних, які не зменшуються після обробки алгоритмом, а навпаки — збільшуються. «Трюк», що дозволяє алгоритмам стиснення без втрат (при використанні на даних для яких вони були спроектовані) послідовно стискати файли до меншого розміру, є те, що файли, для яких алгоритми спроектовані діяти, мають деяку форму легко змодельованої надмірності, яку алгоритм повинен усувати, таким чином зменшуючи їх розмір внаслідок цієї надмірності. Алгоритми стиснення без втрат проектуються під один тип файлів: наприклад, програми для стиснення аудіо не ефективні для стиснення текстових файлів і навпаки. [10]

Список найбільш популярних алгоритмів стиснення без втрат [10]:

* Алгоритм Хаффмана
* ANS (модифікація алгоритму Хаффмана)
* Deflate (суміш алгоритму LZ та алгоритму Хаффмана)
* Z-standard (суміш Хаффмана, ANS та LZ)
* PPM

LZ сімейство:

* LZ77 та LZ78
* LZW
* LZMA
* LZO

Як бачимо, більшість цих алгоритмів для загального кодування без втрат є модифікаціями алгоритмів Хаффмана та LZW, які є фундаментальними алгоритмами кодування без втрат. Для розуміння роботи кодування без втрат та подальшого аналізу форматів, у яких використовуються алгоритми кодування без втрат, розглянемо цих 2 алгоритми детальніше.

1. **АЛГОРИТМ ХАФФМАНА**

Алгоритм Хаффмана - алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю, що був розроблений в 1952 році аспірантом Массачусетського технологічного інституту Девідом Хаффманом при написанні ним курсової роботи. Ідея алгоритму полягає в тому, що знаючи ймовірності появи символів в повідомленні, можна описати процедуру побудови кодів змінної довжини, що складаються з цілої кількості бітів. Символам з більшою ймовірністю ставляться у відповідність коротші коди. Коди Хаффмана мають властивість префіксних, що дозволяє однозначно їх декодувати. [1]

* 1. ***Алгоритми кодування та декодування повідомлення***

Для кодування повідомлення алгоритмом Хаффмана потрібно:

1. Побудувати таблицю частот появи символів у повідомленні
2. Побудувати Дерево Хаффмана
3. Побудувати таблицю Хаффмана
4. Закодувати повідомлення таблицею Хаффмана

Варто зазначити, що разом із закодованим повідомленням потрібно збегігати й дерево для декодування повідомлення у майбутньому.

Для декодування повідомлення, що було закодовано алгоритмом Хаффмана потрібно:

1. За допомогою збереженого дерева Хаффмана розкодувати повідомлення, лінійно проходячись по закодованому повідомленні

Так як основну складність у виконанні процесів кодування та декодування представляє побудова дерева Хаффмана, розглянемо її окремо.

* 1. ***Побудова дерева Хаффмана***

Класичний алгоритм Хаффмана на вході отримує таблицю частот появи символів в повідомленні. Далі на підставі цієї таблиці будується дерево кодування Хаффмана (Н-дерево).

1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожен лист має вагу, який може дорівнювати або ймовірності, або кількості входжень символу у вихідне повідомлення.
2. Вибираються два вільних вузли дерева з найменшими вагами.
3. Створюється вузол предок з вагою, рівною їх сумарній вазі, що об’єднує ці два вузли, як його нащадків.
4. Предок додається в список вільних вузлів, а два його нащадка видаляються з цього списку.
5. Одній з гілок, що виходить з вузла предка, ставиться у відповідність біт 1, іншій - біт 0.
6. Кроки, починаючи з другого, повторюються до тих пір, поки в списку вільних вузлів не залишиться тільки один вільний вузол. Він і буде вважатися коренем дерева.

Зазвичай на практиці, замість списку використовують структуру даних, що зветься черга з пріоритетами, яка реалізує додавання у чергу елементу з числовим пріоритетом, та отримання з черги елемент з найбільшим (або найменшим) пріоритетом. [1]

* 1. ***Недоліки алгоритму Хаффмана***

Класичний алгоритм Хаффмана має ряд істотних недоліків.

По-перше, для відновлення стиснутого повідомлення розкодовувач повинен мати дерево Хаффмана або таблицю частот вихідного повідомлення, якою користувався закодовувач. Отже, довжина стиснутого повідомлення збільшується на довжину таблиці частот, яка повинна надсилатися разом з данимим, що може звести нанівець всі зусилля зі стиснення повідомлення.

По-друге, необхідність наявності повної частотної статистики перед початком власне кодування вимагає двох проходів повідомлення: одного для побудови моделі повідомлення (таблиці частот і Н-дерева), іншого для, власне, кодування, що значно впливає на швидкість кодування.

По-третє, надмірність кодування є нульовою лише в тих випадках, коли ймовірності кодованих символів є зворотними ступенями числа 2.

Лістинг програми, що реалізує архівацію файлів, використовуючи кодування Хаффмана, наведений у додатках.

1. **АЛГОРИТМ LZW**

LZW, Алгори́тм Ле́мпеля — Зі́ва — Ве́лча — універсальний алгоритм стиснення данних без втрат, створений Авраамом, Яковом Зівом і Террі Вейчем. Опублікований Велчем 1984 року як покращена реалізація алгоритму LZ78, опублікованого Лемпелем і Зівом 1978 року. [5]

Даний алгоритм при стисненні даних динамічно створює таблицю перетворення рядків: певним послідовностям символів ставляться у відповідність групи бітів фіксованої довжини (зазвичай 12-бітні). Таблиця ініціюється всіма 1-символьними рядками. По мірі кодування алгоритм переглядає текст символ за символом і зберігає кожен новий унікальний 2-символьний рядок в таблицю у вигляді пари код/символ, де код посилається на відповідний перший символ. Після того, як новий 2-символьний рядок збережений в таблиці, на вихід передається код першого символу. Коли на вході читається черговий символ, для нього по таблиці знаходиться рядок, що вже зустрічався, максимальної довжини, після чого в таблиці зберігається код цього рядка з наступним символом на вході; на вихід видається код цього рядка, а наступний символ використовується як початок наступного рядка.

Алгоритму декодування на вході потрібен лише закодований текст, оскільки він може відтворювати відповідну таблицю перетворень безпосередньо по закодованому тексту.[4, 5]

* 1. ***Процес кодування***

Процес кодування виглядає наступним чином. Послідовно потрібно зчитати символи вхідного потоку вхідного повідомлення і зробити перевірку, чи існує в створеній таблиці рядків такий рядок (тобто перевірити наявність в адаптивному словнику цього рядка). Якщо такий рядок існує, зчитується наступний символ, який додається до попереднього рядка, який виявився для нашого словника відомим, і якщо рядок, який утворився такою конкатинацією не існує в таблиці рядків (отже, немає в словнику), то його потрібно занести в код для попередньо знайденого рядка, новоутворений рядок додається до таблиці (до словника), а пошук починається знову з останнього символу нашого потоку. Слід наголосити на тому, що код рядка – індекс рядка в таблиці.

Даний алгоритм при стисненні (кодуванні) динамічно створює таблицю перетворення рядків: певним послідовностям символів (словам) ставляться у відповідність групи бітів фіксованої довжини.

Спочатку таблиця ініціюється всіма 1-символьними рядками (у випадку 8-бітних символів – це 256 записів).

По мірі кодування алгоритм переглядає текст символ за символом і зберігає кожен новий унікальний 2-символьний рядок в таблицю у вигляді пари код/символ, де код посилається на відповідний перший символ. Після того, як новий 2-символьний рядок збережений в таблиці, на вихід передається код першого символу. Коли на вході читається черговий символ, для нього по таблиці знаходиться рядок, що вже зустрічався, максимальної довжини, після чого в таблиці зберігається код цього рядка з наступним символом на вході; на вихід видається код цього рядка, а наступний символ використовується у якості початку наступного рядка.

* 1. ***Процес декодування***

Особливість LZW полягає в тому, що для декомпрессії нам не потрібно зберігати таблицю рядків у файлі. Алгоритм побудований таким чином, що може відновити таблицю рядків, використовуючи при цьому лише потік кодів вихідного закодованого повідомлення.

Для декодування повідомлення потрібно знати початковий словник, тобто вхідний алфавіт, за початковий словник обираємо таблицю ASCII з 256 різноманітними символами, які могли використовуватись у нашому вхідному файлі. Наступні записи словника будемо реконструювати вже під час декодування, оскільки вони є просто конкатенацією, тобто звичайним склеюванням символів.

Алгоритму декодування на вході потрібно лише закодоване повідомлення, оскільки він зможе відновити відповідну таблицю перетворення безпосередньо по закодованому повідомленню. Алгоритм генерує однозначний декодований код за рахунок того, що кожного разу, коли генерується новий код, новий рядок додається в таблицю рядків. LZW постійно перевіряє, чи є цей рядок вже відомим (тобто ми маємо конкретний код для цього рядку в нашому словнику), і якщо так, то він виводить відповідний код без генерації нового. Таким чином, кожний рядок буде зберігатися в єдиному екземплярі і мати свій унікальний номер. Отже, при декодуванні для отримання нового коду генерується рядок, а при отриманні вже відомих кодів, рядок витягується зі словника.[2]

1. **ФОРМАТИ ДАНИХ, ЩО ВИКОРИСТОВУЮТЬ АЛГОРИТМИ СТИСНЕННЯ БЕЗ ВТРАТ**

Після розгляду двох основних алгоритмів без втрат, які лежать в основі майже всіх інших алгоритмів кодування без втрат, що використовуються у популярних форматах даних, можемо перейти до розгляду самих форматів та їх функціонування.

Для початку, наведемо список форматів, що використовують алгоритми кодування без втрат: [10]

Універсальні формати стиснення без втрат:

* ZIP
* 7-ZIP
* RAR
* PAQ

Аудіо-формати:

* ALAC – Apple Lossless
* DST - Direct Stream Transfer
* DTS-HD
* FLAC - Free Lossless Audio Codec
* MLP - Meridian Lossless Packing
* TTA - True Audio Lossless
* WavPack - WavPack lossless

Графічні формати:

* GIF - без втрат лише для зображень, що містять не більше 256 кольорів
* JPEG-LS
* PNG — Portable Network Graphics
* TIFF
* FLIF — Free Lossless Image Format

Відео формати:

* CorePNG
* Huffyuv
* Ffvhuff
* Lagarith

Розглянемо більш детально та опишемо функціонування таких відомих форматів, як PAQ, FLAC, GIF та PNG.

* 1. ***Універсальний формат PAQ***

Серія вільних архіваторів стиснення без втрат із текстовим інтерфейсом, які спільними зусиллями розробників піднялись на перші місця рейтингів багатьох тестів стиснення даних. PAQ використовує алгоритм context mixing, який пов'язаний із алгоритмом стиснення PPM. При цьому, компресія поділяється на дві фази: моделювання та кодування. PAQ використовує алгоритм змішування контекстів, який, хоч і пов'язаний із PPM, відрізняється тим, що ймовірність виникнення наступного символу розраховується на основі великої кількості моделей, залежних від різних контекстів, але не обов'язково послідовних. Деякі версії PAQ обробляють текст, продивляючись його й заміняючи слова з тексту, які містяться в зовнішньому словнику 1- і 3-байтними кодами. Додатково слова у верхньому регістрі кодуються спеціальним символом і перекладом у нижній регістр.

* 1. ***Формат стиснення звуку FLAC***

FLAC — аудіокодек для стиснення, що на відміну від таких кодеків, як MP3, WMA чи Ogg Vorbis, FLAC забезпечує стиснення без втрат, тобто при розпакуванні звукові дані залишаються повністю ідентичним до початкового файлу перед стисненням.

**Кодування**:

Як і більшість кодеків, FLAC ділить вхідний потік на блоки і кодує їх незалежно один від одного. Блок запаковується у фрейм і додається до потоку. Базовий кодер використовує блоки постійного розміру для всього потоку, однак формат передбачає наявність блоків різної довжини в потоці. Розмір блоку - дуже важливий параметр для кодування. Якщо він занадто малий, то в потоці буде надто багато заголовків фреймів, що зменшить рівень стиснення. Якщо ж розмір великий, то кодер не зможе підібрати ефективну модель стиснення. Розуміння процесу моделювання допоможе вам збільшити рівень стиснення для деяких типів вхідних даних.

На наступному етапі кодер намагається апроксимувати сигнал такою функцією, щоб отриманий після її вирахування з оригіналу результат (званий різницею, залишком, помилкою) можна було закодувати мінімальною кількістю бітів. Параметри функцій теж повинні записуватися, тому вони не повинні займати багато місця. FLAC використовує два методи формування апроксимацій:

* підгонка простого полінома до сигналу
* загальне кодування з лінійними предикторами (LPC).

По-перше, постійне поліноміальне пророкування працює значно швидше, але менш точно, ніж LPC. Чим вище порядок LPC, тим повільніше, але краще буде модель. Однак зі збільшенням порядку виграш буде все менш значним. В деякій точці (зазвичай близько 9) процедура кодера, що визначає найкращий порядок, починає помилятися і розмір одержуваних фреймів зростає. Щоб подолати це, можна використовувати повний перебір, що призведе до значного збільшення часу кодування. По-друге, параметри для постійних предикторів можуть бути описані трьома бітами, а параметри для моделі LPC залежать від кількості біт на семпл і порядку LPC. Це означає, що розмір заголовка фрейму залежить від обраного методу і порядку і може вплинути на оптимальний розмір блоку.

Коли модель підібрана, кодер віднімає наближення з оригіналу, щоб отримати залишковий (помилковий) сигнал, який потім кодується без втрат. Для цього використовується та обставина, що різницевий сигнал зазвичай має розподіл Лапласа і є набір ентропійних кодів, що має назву кодування Райса, що дозволяє ефективно і швидко кодувати ці сигнали без використання словника.

Кодування Райса складається з знаходження одного параметра, що відповідає розподілу сигналу, а потім використання його для складання кодів. При зміні розподілу змінюється і оптимальний параметр, тому є метод, що дозволяє перераховувати його в разі потреби. Залишок може бути розбитий на контексти або розділи, у кожного з яких буде свій параметр Райса. FLAC дозволяє вказати, як потрібно проводити розбиття. Залишок може бути розбитий на 2 n розділів.

* 1. ***Стиснення графіки у GIF***

8-бітний растровий графічний формат, що використовує до 256 чітких кольорів із 24-бітного діапазону RGB. Формат було розроблено компанією CompuServe у 1987 році, і з того часу набув широкої популярності у всесвітній павутині завдяки своїй відносній простоті та мобільності. Одними з головних особливостей формату є підтримка анімації та прозорості.

**Кодування:**

Дані пікселів зображення, відскановані горизонтально зверху ліворуч, перетворюються за допомогою кодування LZW у коди, які потім відображаються у байти для зберігання у файлі. Таблиця кодів спочатку містить коди, які на один біт довші за розмір символу, щоб вмістити два спеціальні коди clr та end та коди для рядків, які додаються під час процесу. Коли таблиця заповнена, довжина коду збільшується, щоб дати простір для більшої кількості рядків, максимум до коду 4095 = FFF (шістнадцятковий). Коли декодер будує свою таблицю, він відстежує збільшення довжини коду і може відповідно розпаковувати вхідні байти. Коди пікселів, як правило, не відповідають 8-бітовому розміру байтів, тому коди упаковуються в байти за схемою «little-Endian». Цей байтовий потік зберігається у файлі у вигляді серії "підблоків". Кожен підблок має максимальну довжину 255 байт і має префікс байта, що вказує кількість байтів даних у підблоці. Серія підблоків завершується порожнім підблоком (одиничним 0 байтом, що вказує на підблок з 0 байтами даних).

* 1. ***Стиснення графіки у JPEG-LS***

JPEG-LS — стандарт стиснення зображень без втрат, що є доповненням до відомих форматів стиснення зображень JPEG і JPEG 2000, орієнтованих, передусім, на стиснення з втратами.

**Кодування:**

Формат JPEG-LS був заснований на форматі LOCO-I (Low Complexity Lossless Compression for Images). Алгоритм стиснення без втрат LOCO-I, прийнятий за основу при розробці стандарту JPEG-LS, вперше передбачав не тільки lossless, але й near lossless режим (стиснення з обмеженими, що задаються користувачем, втратами). Декодер JPEG-LS майже не відрізняється від кодера, тому цей алгоритм стиснення є симетричним.

Алгоритм стиснення, що лежить в основі JPEG-LS, використовує адаптивне передбачення значення поточного пікселя за оточенням, що включає вже закодовані пікселі (метод Median Edge Detection), класифікацію контексту, контекстне моделювання помилки передбачення та її корекцію, а також ентропійне кодування скоригованої помилки передбачення (використовується кодування Голомба-Райса). Для підвищення ефективності кодування низькоентропійних зображень (або фрагментів зображень) алгоритм передбачає автоматичний перехід в режим кодування довжин серій, що дозволяє використовувати його для стиснення без втрат (або з обмеженими втратами) не тільки фотореалістичних зображень, але й комп'ютерної графіки.

Коротко розглянемо та опишемо інші формати даних з використанням алгоритмів стиснення без втрат, що були зазначені на початку цього розділу.

* 1. ***Універсальні формати даних***
     1. ***ZIP та 7-ZIP***

ZIP - формат архівації файлів і стиснення даних без втрат, що був створений в 1989 році Філом Кацом та може містити один або кілька файлів і каталогів, які можуть бути стиснуті різними алгоритмами. Найбільш часто в ZIP використовується алгоритм стиснення Deflate, що є суммішю алгоритму Хаффмана та LZW.

7-zip – аналог zip, що також використовує алгоритм стиснення Deflate, та є трохи швидшим.

* + 1. ***RAR***

RAR — поширений формат стиснення даних і програма-архіватор. Формат розроблений російським програмістом Євгеном Рошалом Він написав програму-архіватор для пакування/розпаковування RAR, спочатку під DOS, потім і для інших платформ. Метод стиснення так і залишається закритим. Операції компресії із форматом RAR зазвичай повільніші, ніж за використання ранніх алгоритмів стиснення, таких як ZIP і gzip, але зазвичай забезпечують вищу якість компресії.

***5.6. Формати кодування аудіо***

***5.6.1. ALAC***

Apple Lossless — відкритий аудіокодек для стиснення цифрової музики без втрат якості, розроблений Apple Inc., цей формат повсюдно використовується в продуктах Apple. В Apple Lossless дані зберігаються в контейнері MP4 з розширенням .m4a. Хоча Apple Lossless має таке ж розширення файлу, як AAC, це не AAC, цей кодек схожий з іншими lossless-кодеками, такими, як FLAC тощо, які використовують моделі лінійного передбачення.

***5.6.2. DTS***

DTS, Digital Theater System - сімейство систем цифрового багатоканального звукозапису, створене компанією «Digital Theater System» для демонстрації цифрових фонограм в кінотеатрах синхронно з прокатними фільмокопіями. Кодер і декодер підтримують численні комбінації каналів: стерео, чотири канали, чотири канали та LFE - такі саундтреки випускаються на комерційній основі на DVD, CD і Laserdisc. Різновиди формату DTS, призначені для кінотеатрів і для побутових пристроїв, сильно відрізняються по алгоритмам кодування і за ступенем стиснення.

***5.6.3. TTA***

The True Audio (TTA) кодек - аудіокодек, що здійснює стиснення аудіофайлів без втрат, здатний працювати в режимі реального часу. Кодек заснований на адаптивних фільтрах і має такі ж або кращі характеристики, як і більшість сучасних кодер-декодерів без втрат. Основними вимогами до розробки компресора були прийнятна ступінь стиснення і висока швидкість роботи. Показники стиснення TTA-кодека залежать від змісту стискуваного музичного файлу, але стислий розмір в основному буде в межах від 30 до 70% від оригінального.

***5.6.4. WavPack***

WavPack - вільний lossless -аудіокодек, заснований на LZW алгоритмі стиснення даних без втрат. Працює з файлами розрядністю біт: 8, 16, 24, 32-float. Підтримує многоканальность (до 256) і високі частоти дискретизації. Є гібридний режим з втратами, що працює схоже з кодеками сімейства ADPCM. Ефективність стиснення залежить від вихідних даних, але зазвичай лежить в діапазоні 30-70% для звичайної популярної музики, вище - для класичної музики.

***5.7. Формати кодування графічних файлів***

***5.7.1. PNG***

PNG (Portable Network Graphics) — растровий формат збереження графічної інформації, що використовує стиснення без втрат. PNG був створений для заміни формату GIF графічним форматом, який не потребує ліцензії для використання. PNG використовує алгоритм стистення повідомлень Deflate.

***5.7.2. FLIF***

Free Lossless Image Format (FLIF) - вільний формат стиснення зображень без втрати якості, який, по завіреннях розробників, за ступенем стиснення перевершує PNG, lossless WebP, lossless BPG і lossless JPEG 2000. В даний час стандартизований і документований. FLIF використовує один з варіантів арифметичного кодування MANIAC - Meta-Adaptive Near-zero Integer Arithmetic Coding, в якості алгоритму ентропійного стиснення, що є варіантом CABAC (контекстно-адаптивне двійкове арифметичне кодування), де контексти є вузлами дерев рішень, які динамічно засвоюються під час кодування.

***5.8. Кодування відеофайлів***

***5.8.1. CorePNG***

CorePNG - це стискаючий без втрат RGB-відеокодек, заснований на використанні методу стиснення зображень PNG. Кожен кадр відео стискається з використанням PNG-стиснення, дозволяючи використовувати всі можливості формату PNG, але також наслідуючи всі його недоліки.

***5.8.2. HuffYUV та Ffvhuff***

HuffYUV — кодек, призначений для стиснення відео без втрат у цілях заміни нестисненого YCbCr як формату відео захоплення. «Без втрат» означає, що результат декомпресії біт-в-біт ідентичний потоку перед компресією (за умови, що не проводилося перетворення колірного простору). Алгоритм HuffYUV передбачає кожен піксель кадру і потім кодує похибку за алгоритмом Хаффмана. Ffvhuff – покращена версія HuffYUV, що демонструє поліпшену ефективність стиснення з використанням доданої в FF адаптивної таблиці Хаффмана.

***5.8.3. Lagarith***

Відеокодек без втрат із відкритим кодом, написаний Беном Грінвудом. Це розгалуження коду HuffYUV, що пропонує кращу компресію за рахунок значно зниженої швидкості на однопроцесорних системах.

1. **ТЕСТУВАННЯ ТА ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ФОРМАТІВ ДАНИХ СТИСНЕННЯ БЕЗ ВТРАТ**

У цьому розділі будуть наведені результати тестування різних форматів даних, що послуговуються алгоритмами стиснення без втрат із категорій, що були розглянуті у попередньому розділі.

* 1. ***Тестування універсальних форматів ZIP, 7-ZIP, RAR та PAQ***

Усі режими було протестовано на 10 файлах розміром 300 МБ кожен. Був знайдений середній час стиснення та розтиснення, а також, розмір архіву.

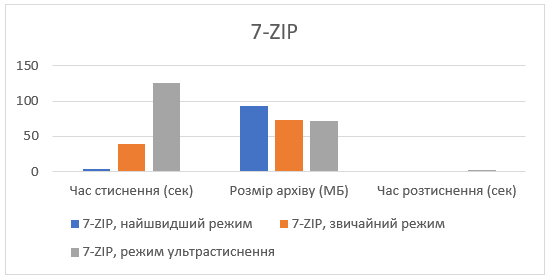


Рис. 1. Результати роботи архіватора з форматом 7-ZIP

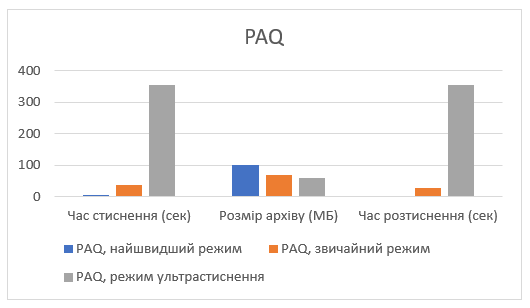


Рис. 2. Результати роботи архіватора з форматом PAQ

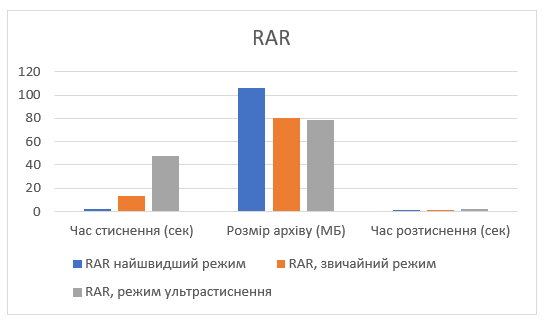


Рис. 3. Результати роботи архіватора з форматом RAR

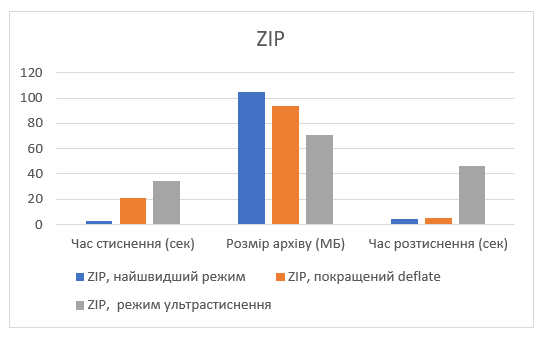


Рис. 4. Результати роботи архіватора з форматом ZIP

Як бачимо, явним фаворитом по якості стискання є формат PAQ, але він має й найвищий час стиснення та розтиснення. На другому місця йде формат 7-zip потім RAR та ZIP. Найкращий же час стиснення та розтиснення демонструє формат RAR. Докладніше ефективність форматів можна порівняти за наступними графіками.

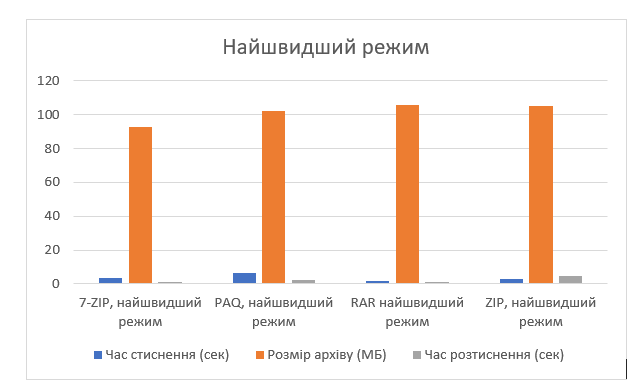


Рис. 5. Порівняння роботи форматів у найшвидшому режимі

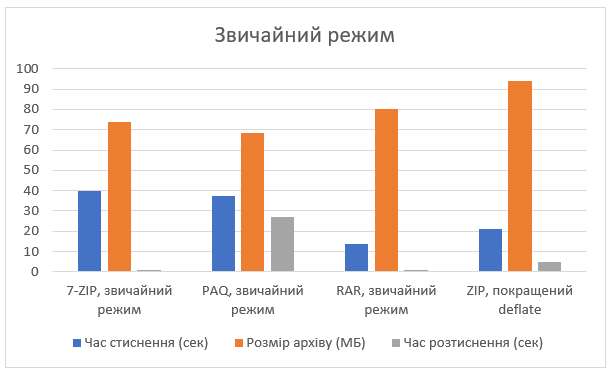


Рис. 6. Порівняння роботи форматів у звичайному режимі

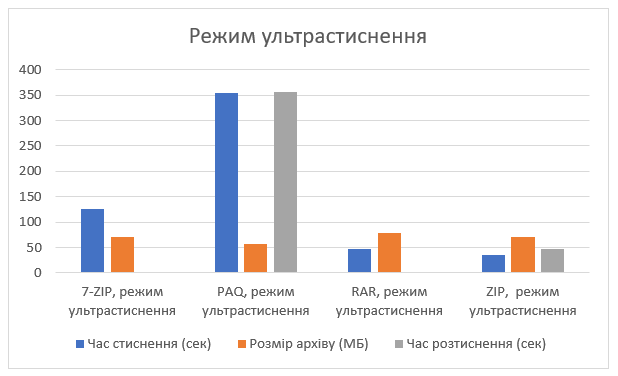


Рис. 7. Порівняння роботи форматів у режимі найкращого стиснення

Як бачимо, кожен формат має хороші показники при різних ступенях стиснення, також, одні виграють по часу, інші – по коефіцієнту стиснення, тому однозначного лідера виділити неможливо. [9]

* 1. ***Тестування аудіоформатів ALAC, DST, FLAC, WavPack***

Під час цього тесту було взято зразок аудіо розміром у 10 МБ у форматі без стиснення (WAV), та перекодовано в розглянуті формати. Замірявся розмір вихідного файлу та коефіцієнт стиснення. Швидкість кодування та декодування не вимірювалася, так як вона дуже мала й приблизно однакова для всіх форматів. Більше того, більшість аудіоплеєрів програють такий формат декодуючи його на ходу, тому швидкість кодування та декодування особливої ролі не грає. [8]

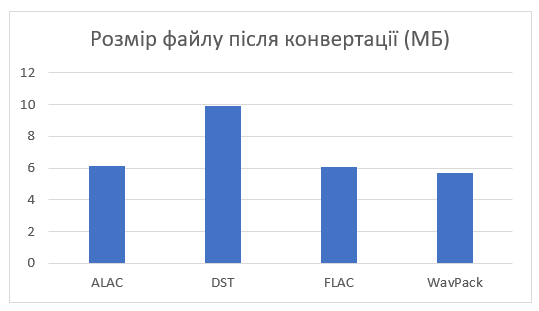


Рис. 8. Розмір файлу після стиснення

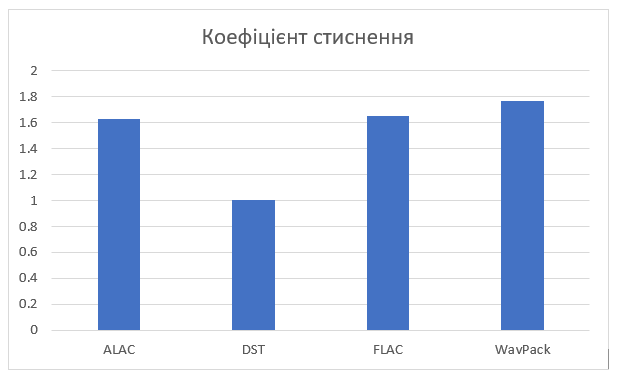


Рис. 9. Коефіцієнт стиснення.

Як бачимо, найкраще стиснення показують формати WavPack та FLAC, а найгірше – DST.

* 1. ***Тестування растрових форматів PNG, JPEG-LS та FLIF***

Під час цього тесту було взяте зображення розміром 7.2 МБ. Воно було перекодовано у всі наведені формати. Замірявся розмір вихідного файлу та коефіцієнт стиснення. [6]

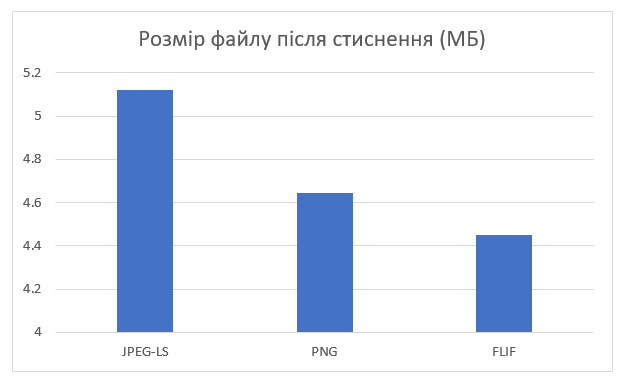


Рис.10. Розмір файлу після стиснення

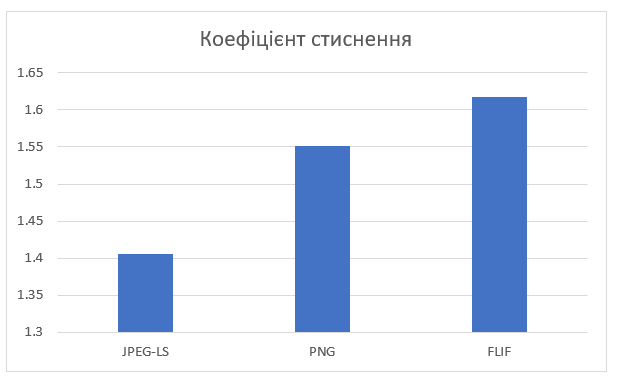


Рис.11. Коефіцієнт стиснення.

Як бачимо, фаворитом по якості стиснення є формат FLIF. Найгірше ж стискає формат JPEG-LS.

1. **ВИСНОВКИ**

У ході цієї роботи було досліджено основні алгоритми стиснення без втрат та формати, що послуговуються цими алгоритмами.

Стиснення без втрат - це клас алгоритмів стиснення даних, що дозволяє ідеально (тобто без втрат) реконструювати вихідні дані із стиснених даних.

Як відомо, жоден алгоритм стиснення без втрат не може ефективно стиснути всі можливі типи дані. З цієї причини існує багато різних алгоритмів, які розроблені або з урахуванням певного типу вхідних даних, або з конкретними припущеннями про те, який тип надмірності можуть містити не стиснуті дані. Однак, як видно з класифікації, більшість алгоритмів стиснення без втрат основані на двох основних – алгоритмі Хаффмана та сімействі алгоритмів LZ.

Так як ці два алгоритми є основою інших, їх роботу було детально розглянуто пунктах 3 та 4 даної роботи.

Також було розглянуто найпопулярніші формати даних, що використовують алгоритми стиснення без втрат, описано їх роботу, проведено тестування та порівняння.

У результаті тестування універсальних форматів було отримано наступну статистичну інформацію.

|  |  |  |  |
| --- | --- | --- | --- |
| Назва архіватора | Час стиснення (сек) | Розмір архіву (МБ) | Час розтиснення (сек) |
| 7-ZIP, найшвидший режим | 3.5 | 92.9 | 0.9 |
| 7-ZIP, звичайний режим | 39.6 | 73.6 | 1 |
| 7-ZIP, режим ультрастиснення | 125 | 71.4 | 2.7 |
| PAQ, найшвидший режим | 6.3 | 102 | 2.1 |
| PAQ, звичайний режим | 37.2 | 68.5 | 26.8 |
| PAQ, режим ультрастиснення | 354 | 57.6 | 356 |
| RAR найшвидший режим | 2 | 106 | 1 |
| RAR, звичайний режим | 13.8 | 80.4 | 1 |
| RAR, режим ультрастиснення | 48 | 78.9 | 1.8 |
| ZIP, найшвидший режим | 3 | 105 | 4.5 |
| ZIP, покращений deflate | 21 | 94 | 5 |
| ZIP, режим ультрастиснення | 34.7 | 70.7 | 46.2 |

Табл. 1. Результати тестування універсальних алгоритмів стиснення

З отриманих даних можемо судити, що найкраще стиснення демонструє формат PAQ, але час стиснення та розтиснення у нього найгірший. Найшвидшим же форматом є ZIP, але коефіцієнт стиснення у нього відносно низький.

Графіки тестування інших форматів даних наведено у пункті 6. З них можна зробити висновок, що найкраще стиснення серед аудіоформатів показують формати WavPack та FLAC, а серед графічних – FLIF.

Якщо говорити про області використання, то стиснення без втрат застосовується у випадках, коли важливо, щоб оригінал та стиснені дані були однаковими. Типовими прикладами є файли формату exe, текстові документи та сирцевий код. Зрозуміло, що невеликі втрати при стисненні цих файлів приведуть до дуже великих збоїв у роботі, або помилок у тексті, тому тут використовуються тільки алгоритми стиснення без втрат. Деякі формати файлів зображень, такі як PNG або GIF, використовують лише стиснення без втрат, тоді як інші, як TIFF та MNG, можуть використовувати як методи без втрат, так і з втратами. Формати аудіо без втрат найчастіше використовуються для архівування або виробництва, тоді як менші аудіофайли із втратами зазвичай використовуються на портативних програвачах, а в інших випадках, коли місце для зберігання обмежене або точна копія аудіо не потрібна.

1. **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Ватолин Д., Ратушняк А., Юкин В. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео. М.: ДИАЛОГ-МИФИ, 2003. — 384с.

2. Д.Сэломон. Сжатие данных, изображений и звука Москва: Техносфера, 2004. - 368с.

3. Хеммінг Р.У. «Теория кодирования и теория информации» М.: Радио и связь, 1983 . — 176 с.

4. Вікіпедія. Вільна енциклопедія [Електроний ресурс]. Режим доступу: https://ru.wikipedia.org/wiki/LZ77(10.04.2020)

5. Вікіпедія. Вільна енциклопедія [Електроний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch(26.03.2021)

6. Вікіпедія. Вільна енциклопедія [Електроний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Comparison\_of\_graphics\_file\_formats (17.05.2021)

8. Вікіпедія. Вільна енциклопедія [Електроний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Comparison\_of\_audio\_coding\_formats (13.01.2021)

9. Порівняння роботи архіваторів[Електроний ресурс]. Режим доступу: https://peazip.github.io/maximum-compression-benchmark.html (12.05.2019)

10. Вікіпедія. Вільна енциклопедія [Електроний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Lossless\_compression (04.05.2021)

1. **ДОДАТКИ**
   1. ***Виконуваний код алгоритму стиснення Хаффмана***

Лістинг представлений мовою програмування python.

Файл huffman\_tree\_functions.py:

from queue import PriorityQueue

from dataclasses import dataclass, field

from typing import Any

from bitarray import bitarray

@dataclass(order=True)

class PrioritizedItem:

priority: int

item: Any = field(compare=False)

def unpack(self):

return self.priority, self.item

@staticmethod

def pack(unpacked\_item):

return PrioritizedItem(unpacked\_item[0], unpacked\_item[1])

class TreeNode:

def \_\_init\_\_(self, value, left\_child=None, right\_child=None):

self.value = value

self.left\_child = left\_child

self.right\_child = right\_child

class HuffmanTree:

def \_\_init\_\_(self, root):

self.root = root

self.current\_node = self.root

self.serialized\_tree\_path = bitarray()

def get\_huffman\_dict(self):

codes\_dict = dict()

self.\_\_inorder\_dfs('', codes\_dict, self.root)

return codes\_dict

# обходимо дерево для отримання таблиці елемент: код

def \_\_inorder\_dfs(self, current\_code, codes\_dict, node):

if node is None:

return

self.\_\_inorder\_dfs(current\_code + '0', codes\_dict, node.left\_child)

if node.value is not None:

codes\_dict[node.value] = current\_code

self.\_\_inorder\_dfs(current\_code + '1', codes\_dict, node.right\_child)

# побітово рухаємося по дереву (0 - вліво, 1 - вправо), поки не буде знайдений листок

def feed\_bit(self, bit):

if bit == 0:

self.current\_node = self.current\_node.left\_child

if bit == 1:

self.current\_node = self.current\_node.right\_child

if self.current\_node.value is not None:

val = self.current\_node.value

self.current\_node = self.root

return val

# старий метод, використовувати тільки для текстових кодів

def get\_symbol(self, code):

return self.\_\_get\_symbol(code, self.root)

# старий метод, використовувати тільки для текстових кодів

def \_\_get\_symbol(self, current\_code, node):

if node.value is not None:

return node.value, current\_code

if len(current\_code) == 0:

raise Exception('Given code does not match any symbol.')

if current\_code[0] == '0':

return self.\_\_get\_symbol(current\_code[1:], node.left\_child)

elif current\_code[0] == '1':

return self.\_\_get\_symbol(current\_code[1:], node.right\_child)

# отримуємо бітове представлення дерева у форматі 1 - звичайний вузол, 0 - листок + значення листка у бітах

def get\_path(self):

self.\_\_get\_path(self.root)

return self.serialized\_tree\_path

def \_\_get\_path(self, node):

if node.value is not None:

value\_bits = bitarray()

value\_bits.frombytes(node.value.to\_bytes(1, 'little'))

value\_bits.insert(0, 0)

self.serialized\_tree\_path += value\_bits

else:

self.serialized\_tree\_path.append(1)

self.\_\_get\_path(node.left\_child)

self.\_\_get\_path(node.right\_child)

# отримуємо список унікальних символів з файлу та кількість входжень кожного з них

def \_\_construct\_frequency\_list(message):

result = {}

list\_result = []

for symbol in message:

try:

info = result[symbol]

except KeyError:

info = 0

info += 1

result[symbol] = info

for key, value in result.items():

list\_result.append([key, value])

return sorted(list\_result, key=lambda x: x[1], reverse=True)

def \_\_construct\_priority\_queue(frequency\_list):

queue = PriorityQueue()

for symbol in frequency\_list:

queue.put(PrioritizedItem(symbol[1], TreeNode(symbol[0])))

return queue

def build\_huffman\_tree(message):

p\_queue = \_\_construct\_priority\_queue(\_\_construct\_frequency\_list(message))

while p\_queue.qsize() > 1:

item1 = p\_queue.get().unpack()

item2 = p\_queue.get().unpack()

result = PrioritizedItem.pack((item1[0] + item2[0], TreeNode(None, left\_child=item1[1], right\_child=item2[1])))

p\_queue.put(result)

return HuffmanTree(p\_queue.get().unpack()[1])

class HuffmanBuilder:

def \_\_init\_\_(self, path):

self.path = path

def get\_next\_path\_item(self):

next\_item = self.path[0]

self.path = self.path[1:]

return next\_item

def get\_leaf\_value(self):

value = self.path[0:8]

self.path = self.path[8:]

return value

def build\_tree(self):

return HuffmanTree(self.\_\_build\_tree\_from\_path())

def \_\_build\_tree\_from\_path(self):

current\_path\_item = self.get\_next\_path\_item()

if current\_path\_item == 1:

current\_node = TreeNode(None)

current\_node.left\_child = self.\_\_build\_tree\_from\_path()

current\_node.right\_child = self.\_\_build\_tree\_from\_path()

else:

current\_node = TreeNode(int.from\_bytes(self.get\_leaf\_value().tobytes(), 'little'))

return current\_node

Файл main.py:

import huffman\_tree\_functions

import math

from bitarray import bitarray

def huffman\_encode(huffman\_dict, message):

result = bitarray()

for symbol in message:

result += bitarray(huffman\_dict[symbol])

return result

def huffman\_decode(huffman\_tree, encoded\_message, file\_name):

with open(f'{file\_name.split("-")[0]}-decoded.{file\_name.split("-")[1]}', 'wb') as f:

for index, bit in enumerate(encoded\_message):

try:

f.write(huffman\_tree.feed\_bit(bit).to\_bytes(1, 'little'))

except AttributeError:

continue

def calculate\_entropy(frequency\_list, size):

i = 0

entropy = 0

while i < len(frequency\_list):

probability = frequency\_list[i][1] / size

entropy += probability \* math.log2(probability)

i += 1

return -entropy

def encode\_file(file\_name, extension):

with open(f'{file\_name}.{extension}', 'rb') as file:

bytes\_from\_file = file.read()

# будуємо дерево Хаффмана

huff\_tree = huffman\_tree\_functions.build\_huffman\_tree(bytes\_from\_file)

# отримуємо таблицю Хаффмана

huff\_dict = huff\_tree.get\_huffman\_dict()

# кодуємо послідовність байтів з файлу

enc\_bytes = huffman\_encode(huff\_dict, bytes\_from\_file)

# перетворюємо дерево у bitarray

encoded\_tree = huff\_tree.get\_path()

# доповнюємо bitarray нулями до тих пір, поки його довжина не ділиться на 8

encoded\_tree.fill()

# отримуємо довжину дерева у байтах

tree\_len\_bytes = len(encoded\_tree).to\_bytes(2, 'little')

# записуємо дані у файл

with open(f'{file\_name}-{extension}-encoded.lab3', 'wb') as file:

file.write(tree\_len\_bytes)

encoded\_tree.tofile(file)

enc\_bytes.tofile(file)

def decode\_file(file\_name):

tree\_path = bitarray()

encoded\_bytes\_from\_file = bitarray()

with open(f'{file\_name}.lab3', 'rb') as file:

# читаємо довжину дерева у бітах

tree\_len = file.read(2)

tree\_len = int.from\_bytes(tree\_len, 'little')

# читаємо байти дерева з файлу

raw\_tree\_bytes = file.read(int(tree\_len / 8))

tree\_path.frombytes(raw\_tree\_bytes)

# будуємо дерево Хаффмана з його бітового представлення

tree\_builder = huffman\_tree\_functions.HuffmanBuilder(tree\_path)

tree\_from\_file = tree\_builder.build\_tree()

# читаємо закодований вміст файлу

encoded\_bytes\_from\_file.frombytes(file.read())

# розкодовуємо вміст файлу та записуємо результат у файл

huffman\_decode(tree\_from\_file, encoded\_bytes\_from\_file, file\_name)

if \_\_name\_\_ == '\_\_main\_\_':

while True:

print('\n[1] [Закодувати файл]\n[2] [Розкодувати файл]\n[3] [Знайти ентопію файлу]\n[4] [Вийти]\n')

option = input('Уведіть цифру для вибору опції: ')

if option == '1':

f\_n = input('Уведіть назву файлу: ')

ext = input('Уведіть розширення: ')

encode\_file(f\_n, ext)

elif option == '2':

f\_n = input('Уведіть назву файлу: ')

decode\_file(f\_n)

elif option == '3':

f\_n = input('Уведіть назву файлу: ')

ext = input('Уведіть розширення: ')

with open(f'{f\_n}.{ext}', 'rb') as file:

contents = file.read()

f\_list = huffman\_tree\_functions.\_\_construct\_frequency\_list(contents)

print(calculate\_entropy(f\_list, len(contents)))

elif option == '4':

break