МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 26.00.00.000 ПЗ

Група ІПм-23-1

Гуфчак Роман

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Гук Володимир Анатолійович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

МАГІСТЕРСЬКА РОБОТА

Дослідження та оптимізація алгоритмів стиснення інформації для архівації даних

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Гуфчак Р.М.
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Лютак Ігор Зіновійович, д.т.н., професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри
доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль
доц. Вовк Р.Б.
(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітньо-кваліфікаційний рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Гуфчаку Роман Михайловичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “Дослідження та оптимізація алгоритмів стиснення інформації для архівації даних ”

керівник проєкту (роботи) Лютак Ігор зіновійович, д.т.н., професор

затверджені наказом закладу вищої освіти від **“** 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проєкту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проєкту (роботи) Архітектура, формальний опис та реалізація сервісу стиснення інформації для архівації даних

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз існуючих сучасних алгоритмів стискання
2. Дослідження ефективності сучасних алгоритмів стиснення
3. Формулювання вимог та алгоритмів функціонування сервісу стиснення даних
4. Розробка алгоритму та програмної реалізації сервісу для стиснення даних

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Методи стиснення даних(рис. 1.2, ст. 24)
2. Графік порівняння коефіцієнту стискання текстових файлів (рис. 2.3, ст. 44)
3. Графік впливу розміру файлу на коефіцієнт стискання (рис. 2.9, ст. 52)
4. Архітектура сервісу (рис. 3.3, ст. 80)
5. Діаграма процесу завантаження і подальшої обробки файлу (рис. 3.5, ст. 88)

6. Консультанти розділів проєкту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури	20.09.2024	виконано
2	Аналіз сучасних алгоритмів стискання	01.10.2024	виконано
3	Методи для оптимізації швидкості та якості роботи алгоритмів	12.10.2024	виконано
4	Дослідження алгоритмів стиснення даних та огляд популярних сервісів	25.10.2024	виконано
5	Формулювання вимог та алгоритмів функціонування сервісу стискання даних	05.11.2024	виконано
6	Програмна реалізація рішення	22.11.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 111 с., 26 рис., 3 табл., 42 джерел.

Тема: Дослідження та оптимізація алгоритмів стиснення інформації для архівації даних

Об'єкт дослідження: технології та алгоритми, які використовуються для стискання інформації.

Мета роботи: дослідження існуючих рішень, а також розробка та оптимізація алгоритмів стиснення даних із використанням сучасних засобів візуалізації, що забезпечить ефективне застосування цих алгоритмів до великих наборів інформації.

Предмет дослідження: є технології та алгоритми, які використовуються для стискання інформації.

Результати дослідження:

Розроблено інструмент, що дозволяє оцінювати продуктивність алгоритмів стиснення даних. Його реалізація виконана мовою програмування Python із використанням інтерактивних графічних компонентів, розроблених на базі бібліотеки Jinja. Окремо були досліджені шляхи покращення вибраних алгоритмів для адаптації до специфічних завдань.

Висновок:

У ході дослідження була створена програмна система, яка забезпечує ефективне впровадження методів стиснення для великих наборів даних, а також дозволяє вибирати автоматично алгоритм. Сформульовано рекомендації щодо оптимального вибору алгоритмів залежно від конкретних потреб. Вдосконалені алгоритми показали вищу швидкість і рівень стиснення у порівнянні з аналогами.

АЛГОРИТМИ СТИСНЕННЯ, АРХІВАЦІЯ ДАНИХ, ОПТИМІЗАЦІЯ, ПРОДУКТИВНІСТЬ, PYTHON, FLASK, SQALCHEMY, АВТОМАТИЗАЦІЯ, ВІЗУАЛІЗАЦІЯ, JINJA, ОБРОБКА ДАНИХ, ВЕБ-ДОДАТОК.

ANNOTATION

Master's work: 111 pages, 26 illustrations, 3 tables, 42 references.

Topic: Research and optimization of information compression algorithms for data archiving.

Object of research: Technologies and algorithms used for information compression.

Purpose: The investigation of existing solutions, as well as the development and optimization of data compression algorithms using modern visualization tools, will ensure the effective application of these algorithms to large datasets

Subject of research: The characteristics of compression methods, their efficiency, and approaches to applying them to large data volumes.

Research results:

A tool was developed to evaluate the performance of data compression algorithms. It was implemented in Python with interactive graphical components developed using the Jinja-based framework. Additionally, various methods of improving selected algorithms were examined to adapt them for specific tasks.

Conclusion:

In the course of the research, a software system was created that enables the effective implementation of compression methods for large datasets, as well as provides an opportunity to analyze their performance. Recommendations were formulated on the optimal selection of algorithms based on specific requirements. The improved algorithms demonstrated higher speed and compression ratios compared to their counterparts.

DATA COMPRESSION, ARCHIVING, OPTIMIZATION, PERFORMANCE, PYTHON, FLASK, SQLALCHEMY, AUTOMATION, VISUALIZATION, JINJA, DATA PROCESSING, WEB APPLICATION.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API - це прикладний програмний інтерфейс

RESTful - це архітектурний стиль розробки веб-сервісів, який акцентує увагу на стандартизації інтерфейсів між клієнтом і сервером

Jinja - це шаблонний двигун для Python, який використовується для візуалізації веб-сторінок

LZ77 - це алгоритм стиснення без втрат, що базується на пошуку повторюваних фрагментів

Huffman - це алгоритм кодування для оптимального стиснення даних

LZW - це алгоритм стиснення, що використовує динамічні словники

DEFLATE - це алгоритм стиснення, який комбінує LZ77 та кодування Хаффмана

CSV - це формат файлів для зберігання табличних даних, розділених комами

JSON - це текстовий формат для зберігання і передачі структурованих даних

Python - це мова програмування, яка використовується для реалізації аплікації

RAM - це оперативна пам'ять, що використовується для обробки даних

CPU - це центральний процесор, що виконує обчислення

ВСТУП

Актуальність роботи

В умовах постійного зростання обсягів даних питання їх ефективного зберігання та передачі стає дедалі актуальнішим. Алгоритми стиснення даних дозволяють зменшити розмір інформації, не втрачаючи її цінності, що особливо важливо для великих наборів даних. Однак ефективність традиційних алгоритмів часто виявляється недостатньою для сучасних потреб у швидкості та гнучкості обробки.

Розробка та оптимізація алгоритмів стиснення є невіддільною частиною багатьох сфер: від архівування документів і мультимедійних файлів до забезпечення ефективної передачі даних у розподілених системах. У нашій роботі особлива увага приділяється впровадженню адаптивних алгоритмів стиснення для роботи з великими обсягами інформації та інтерактивній візуалізації їхньої продуктивності, що дозволяє проводити їх глибокий аналіз.

Порівняння роботи з відомими розв'язаннями проблеми

Серед відомих рішень, які використовуються для стиснення даних, найбільш поширеними є алгоритми Huffman, LZW та їхні модифікації. У той час як ці алгоритми демонструють високу ефективність для певних типів даних, вони мають обмеження у контексті адаптивності до різноманітних задач.

Існуючі інструменти, такі як програмні бібліотеки gzip, zlib, 7-Zip, успішно застосовуються для архівації даних, але більшість із них не пропонують гнучких засобів візуалізації чи інтерактивного аналізу продуктивності.

Розроблений додаток є унікальним рішенням для стискання даних, яке поєднує в собі підтримку кількох популярних алгоритмів стиснення та ретельне тестуванні для аналізу ефективності. Я впровадив такі алгоритми, як Huffman, Gzip, Deflate, Zip, WebP, JPEG, FFmpeg для роботи з відео та AAC для аудіофайлів. Це дозволяє охопити широкий спектр форматів файлів, забезпечуючи гнучкість і зручність для користувачів.

Мета і задачі дослідження

Метою цієї магістерської роботи є дослідження існуючих рішень, а також розробка та оптимізація алгоритмів стиснення даних із використанням сучасних засобів візуалізації, що забезпечить ефективне застосування цих алгоритмів до великих наборів інформації.

Задачі дослідження:

1. Провести аналіз існуючих алгоритмів стискання даних, таких як Huffman, Gzip, Deflate, WebP, JPEG, FFmpeg та AAC.
2. Дослідити ефективність алгоритмів залежно від типу файлів та розміру вхідних даних.
3. Розробити програмну аплікацію, що підтримує стиснення текстів, зображень, відео та аудіофайлів за допомогою різних алгоритмів.
4. Забезпечити інтеграцію кількох алгоритмів стискання у рамках одного додатка з автоматичним вибором оптимального алгоритму в залежності від типу файлу.
5. Розробити функціонал для роботи з користувачами, включаючи авторизацію, збереження історії стискань та завантаження результатів
6. Реалізувати можливість автоматизованого тестування продуктивності алгоритмів із вимірюванням ключових параметрів: часу виконання, коефіцієнта стискання та використання пам'яті. Дослідити шляхи оптимізації алгоритмів для специфічних типів даних.

Об’єктом дослідження є технології та алгоритми, які використовуються для стискання інформації.

Предметом дослідження є технології стиснення інформації, алгоритми архівації даних та засоби візуалізації їхньої ефективності. Дослідження також охоплює шляхи оптимізації існуючих алгоритмів стиснення для роботи з великими наборами даних.

Методи дослідження

Для формулювання вимог до програмного рішення було використано методологію порівняльного аналізу існуючих алгоритмів стиснення, моделювання їхньої продуктивності та експериментальне впровадження обраних рішень у програмну систему.

Наукова новизна одержаних результатів

Виконано детальний аналіз існуючих алгоритмів стиснення даних і методів їхньої реалізації. На основі дослідження було запропоновано нові підходи до адаптації цих алгоритмів для специфічних завдань, що передбачають роботу з великими обсягами даних. Запропоновано архітектуру інтерактивного інструменту для оцінки продуктивності алгоритмів, який дозволяє виявляти оптимальні рішення залежно від вхідних умов..

Практичне значення одержаних результатів.

Як результат виконана програмна система, яка забезпечує реалізацію кількох алгоритмів стиснення даних та їх візуалізацію в інтерактивному вигляді.

Особистий внесок

Виконана розробка аплікації для стискання даних охоплює кілька ключових аспектів, які спрямовані на створення зручного, функціонального та ефективного рішення. Було проведено детальний аналіз сучасних алгоритмів стискання, таких як Huffman, Gzip, Deflate, WebP, JPEG, FFmpeg та AAC, оцінюючи їх переваги та недоліки в контексті різних типів файлів. На основі цього аналізу було розроблено програмну аплікацію, яка підтримує стиснення текстових, графічних, відео та аудіофайлів за допомогою вищезазначених алгоритмів.

Особливу увагу було приділено інтеграції алгоритмів у єдину платформу, яка автоматично обирає оптимальний алгоритм залежно від типу файлу. Я впровадив функціонал для роботи з користувачами, включаючи авторизацію, ведення історії стискань та завантаження результатів. Для цього використано сучасні вебтехнології,

такі як Flask та Jinja, що дозволило створити інтуїтивний інтерфейс і забезпечити високу продуктивність додатка.

Структура магістерської роботи.

Магістерська робота викладена на 111 сторінках друкованого тексту, який складається з вступу, чотирьох розділів, висновків, списку використаних джерел (42 найменувань). Робота містить 27 рисунків.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ АЛГОРИТМІВ ТА МЕТОДІВ СТИСНЕННЯ ІНФОРМАЦІЇ

1.1 Загальна характеристика алгоритмів стиснення інформації

Стиснення даних застосовується майже скрізь. Майже всі зображення, які ми отримуємо з Інтернету, стискаються у форматах JPEG або GIF, більшість модемів використовують стиснення, HDTV стискається за допомогою MPEG-2, а деякі файлові системи автоматично стискають файли під час їхнього зберігання, решта ж користувачів стискають дані вручну.

Цікавість стиснення, як і інших тем, які ми розглядаємо, полягає в тому, що алгоритми, які використовуються у реальному світі, активно застосовують широкий спектр інструментів, таких як сортування, хеш-таблиці, дерева, а також швидке перетворення Фур'є (FFT). Крім того, алгоритми зі стійким теоретичним обґрунтуванням відіграють важливу роль у реальних застосуваннях.

1.1.1 Поняття стистення даних

Для опису предметів, які ми хочемо ущільнити, можна використовувати загальний термін "повідомлення". Це можуть бути файли або безпосередньо повідомлення. Завдання ущільнення складається з двох основних компонентів: алгоритму кодування, який приймає повідомлення і створює його "ущільнене" представлення (зазвичай із меншою кількістю бітів), та алгоритму декодування, який відновлює оригінальне повідомлення або його наближення зі ущільненого представлення. Ці два компоненти повинні тісно взаємодіяти, адже і кодувальник, і декодувальник мають "розуміти" спільне ущільнене представлення.

Розрізняють алгоритми без втрат, які дозволяють точно відновити оригінальне повідомлення зі стисненого, та алгоритми з втратами, які відновлюють лише наближення до оригінального повідомлення. Алгоритми без втрат зазвичай

використовуються для тексту, а з втратами — для зображень та звуку, де незначна втрата якості може бути непримітною або прийнятною.

Алгоритми з втратами не означають випадкові втрати, наприклад, пікселів, а скоріше усувають певні компоненти, такі як частотні складові чи шум. Наприклад, система, яка перефразовує речення або замінює слова синонімами для кращого стиснення, технічно буде втратною, але зміст та ясність повідомлення можуть бути збережені, а іноді навіть покращені.

Виникає питання: чи існує алгоритм без втрат, який може стиснути будь-яке повідомлення. Деякі заявляли, що це можливо, як, наприклад, патент 5,533,051 "Методи стиснення даних".

Заявка стверджувала, що при рекурсивному застосуванні файл можна зменшити до майже нічого. Проте за допомогою простих міркувань можна переконатися, що це неможливо. Наприклад, якщо ми розглянемо всі можливі повідомлення з 1000 біт, то їх кількість становитиме 210002^{1000}.

Для збереження їх усіх у стислому вигляді з довжиною 999 біт можливі лише 29992^{999} різних комбінацій, що менше, ніж кількість оригінальних повідомлень. Таким чином, якщо одне повідомлення скорочується, то інше обов’язково подовжується. Це підтверджується на практиці, якщо стискати файл формату GIF за допомогою GZIP.

Алгоритми стиснення завжди припускають наявність деякого зміщення в імовірностях вхідних повідомлень, тобто, що деякі повідомлення є більш ймовірними, ніж інші. Ця упередженість зазвичай базується на структурі повідомлень — наприклад, повторення символів у тексті або великі білі області у зображеннях.

Усі алгоритми стиснення можна поділити на дві частини: модель і кодувальник. Модель аналізує ймовірності повідомлень на основі їхньої структури, а кодувальник використовує ці ймовірності для створення кодів. Наприклад, у випадку відеодзвінків модель може "знати", що деякі форми облич є більш ймовірними, ніж інші.

Сучасні алгоритми, однак, зазвичай працюють з менш складними моделями, які враховують лише повторювані шаблони.

1.1.2 Загальні принципи роботи алгоритмів стиснення

Основні принципи стиснення даних викладено для досягнення зменшення розміру файлу шляхом більш ефективного кодування даних. Один із доступних типів стиснення даних називається стисненням без втрат.

Це означає, що стислий файл буде відновлено до свого початкового стану без втрати даних під час процесу розпакування. Важливість цього є надзвичайно важливою, оскільки файл буде пошкоджено та стане непридатним для використання, якщо дані будуть втрачені. Алгоритми стиснення без втрат використовують методи статистичного моделювання, щоб зменшити кількість повторюваної інформації у файлі.

Деякі з методів можуть включати видалення пробілів, представлення рядка повторюваних символів одним символом або заміну повторюваних символів меншими бітовими послідовностями. Ще одна категорія стиснення, яка часто використовується в мультимедійних файлах для музики та зображень (наприклад, файли JPEG), де дані відкидаються, називається стисненням із втратами. У цьому класі методів кодування даних для представлення вмісту використовуються неточні наближення (або часткове відкидання даних). Ці методи просто використовуються для зменшення розміру даних для зберігання, обробки та передачі вмісту.

Переваги стиснення файлів можуть бути величезними, оскільки кількість «бітів», які використовуються для зберігання інформації, значно зменшується. Фактично це означає, що стиснені файли займають набагато менше місця. Стиснення файлів також може стиснути кілька невеликих файлів в один файл для більш зручної передачі електронної пошти, оскільки менший розмір файлів призведе до скорочення часу передачі під час їх передачі в Інтернеті.

Коли використовується велика кількість файлів, стиснення може бути математично складним і тривалим процесом. З такою кількістю варіантів алгоритмів стиснення користувач, який завантажує стиснутий файл, може не мати необхідної програми для його розпакування. Деякі алгоритми стиснення можуть запропонувати різні рівні стиснення, причому вищі рівні досягають меншого розміру файлу, але

займають навіть більший час стиснення. Це системно інтенсивний процес, який забирає цінні ресурси, що іноді може призводити до помилок «за браком пам’яті».

Стиснення даних є надзвичайно важливим у світі комп’ютерів і зазвичай використовується багатьма програмами. Надаючи короткий огляд того, як працює стиснення в цілому, ми сподіваємось, що цей блог дозволить користувачам стиснення даних зважити переваги та недоліки під час роботи з ним.

Тобто стислий файл буде відновлений до свого початкового стану без втрат даних при розпаковці. Це дуже важливо оскільки файл буде пошкоджений і буде неприпустим для використання якщо дані будуть втрачені. Алгоритми без втрат використовують статистичний моделювання щоб зменшити кількість повторюваних даних в файлі.

Деякі з них можуть включати в себе видалення пробілів, представлення рядка повторюваних символів одним символом або заміну повторюваних символів меншими бітовими послідовностями. Інша категорія компресії яка часто використовується в мультимедійних файлах для музики та зображень (наприклад файл JPEG) де дані відкидаються називається компресією з втратами. У цьому класі методів кодування даних для представлення вмісту використовуються неточні наближення (або часткове відкидання даних). Ці методи просто використовуються для зменшення розміру даних для зберігання, обробки та передачі вмісту.

Переваги стиснення файлів можуть бути величезними, оскільки кількість «бітів», що використовуються для зберігання інформації, зменшується. Фактично, це означає, що стиснені файли займають набагато менше місця.

Якщо використовується велика кількість файлів стиснення може бути математично складним і тривалим процесом. з такою кількістю алгоритмів стиснення користувач який завантажує стиснутий файл може не мати необхідної програми для його розпакування. деякі алгоритми стиснення можуть пропонувати різні рівні стиснення причому вищі рівні досягають меншого розміру файлу але займають навіть більший час стиснення. це системно інтенсивний процес який забирає цінні ресурси що іноді може призводити до помилок «за браком пам’яті».

Стиснення даних дуже важливо в комп’ютерному світі і використовується багатьма програмами. коротко про те як стиснення працює і ми надаємо цей блог щоб користувачі стиснення даних визначили переваги та недоліки.

Алгоритми стиснення працюють завдяки виявленню надлишковості у даних, яку можна зменшити або усунути. Основні принципи:

1. Використання повторень: пошук однакових фрагментів даних та заміна їх посиланнями на першу появу.
2. Кодування частоти: заміна часто повторюваних символів на коротші коди, а рідкісних - на довші.
3. Упаковка блоків: об'єднання даних у більші блоки для оптимізації їх зберігання.
4. Використання шаблонів: пошук структурних або логічних повторень у даних.
5. Інтеграція прогнозування: передбачення наступних значень на основі попередніх даних та зберігання лише різниць.

1.2 Класифікація методів стиснення: без втрат та з втратами.

1.2.1 Загальні алгоритми стиснення без втрат

Кодування Хаффмана призначає коротші коди більш частим символам і довші коди менш частим. Використовуючи коди змінної довжини на основі частоти символів, він ефективно стискає дані.

Приклад. У тексті, де часто зустрічається літера "e", її можна замінити коротшим двійковим кодом, тоді як менш поширена літера, наприклад "z", отримує довший код.

Загальне використання:Кодування Хаффмана часто використовується у таких форматах файлів, як JPEG і PNG, як частина процесу стиснення.

Лемпель-Зів-Велч (LZW). Стиснення LZW створює словник шаблонів або послідовностей під час обробки даних. Коли він знаходить повторювану

послідовність, він замінює її коротшим кодом, який посилається на статтю зі словника.

Приклад:У файлі з повторюваними словами або фразами LZW може замінити кожне повторення кодом, який вказує на перше входження в словник.

Загальне використання:LZW зазвичай використовується в зображеннях GIF і старих форматах стиснення файлів, таких як TIFF і UNIX.

Кодування довжини серії (RLE). RLE стискає дані, замінюючи послідовності однакових значень одним значенням і підрахунком. Це особливо ефективно для даних із великою кількістю повторюваних елементів.

Приклад:Якщо зображення містить ряд із 10 чорних пікселів, за якими слідують 5 білих пікселів, RLE може зберегти це як «10B, 5W» замість «BBBBBBBBBBWWWWW».

Загальне використання:RLE часто використовується в простих форматах зображень, таких як BMP, і в факсимільних апаратах для стиснення чорно-білих зображень.

Алгоритм Deflate поєднує дві методики: LZ77 (метод стиснення ковзного вікна) і кодування Хаффмана. Він стискає дані, знаходячи повторювані послідовності та кодуючи їх кодами змінної довжини.

Приклад:Він може замінювати повторювані фрази в текстовому файлі вказівниками на попередні випадки в поєднанні з кодами Хаффмана для окремих символів.

Загальне використання:Deflate широко використовується у таких форматах файлів, як ZIP, gzip і PNG, що робить його одним із найпоширеніших методів стиснення сьогодні.

1.2.2 Загальні алгоритми стиснення з втратами

1. JPEG (Joint Photographic Experts Group)

Стиснення JPEG зменшує розмір зображення, розбиваючи його на невеликі блоки, перетворюючи у частотну область за допомогою дискретного косинусного перетворення (DCT) та видаляючи менш важливі частотні компоненти.

Приклад: На фотографії незначні відтінки кольорів, які людське око може не помітити, ігноруються, що суттєво зменшує розмір файлу.

Загальне використання: JPEG є найпопулярнішим форматом для фотографій в Інтернеті, який застосовуєтьс}я в цифрових камерах, соціальних мережах та веб-сайтах.

2. MP3 (MPEG Layer 3 Audio)

MP3 стискає аудіо, видаляючи частоти, які менш помітні людському вуху, на основі психоакустичних моделей. Після цього він ефективніше кодує решту даних.

Приклад: У файлі пісні можуть бути вилучені високі звуки або фоновий шум, щоб зменшити розмір файлу, без помітного впливу на якість звуку для більшості слухачів.

Загальне використання: MP3 є популярним форматом музичних файлів і подкастів, який широко застосовується для потокового передавання, завантаження та зберігання аудіоконтенту.

3. MPEG (група експертів з рухомого зображення)

MPEG стискає відео, видаляючи зайві дані між кадрами та всередині кадрів, часто використовуючи такі методи, як прогнозування руху та дискретне косинусне перетворення (DCT).

Приклад: У відео схожі кадри (наприклад, статичний фон) кодуються один раз і повторно використовуються, тоді як оновлюються лише рухомі частини, зменшуючи розмір файлу.

Загальне використання:Формати MPEG, такі як MPEG-2 і MPEG-4, зазвичай використовуються в цифрових телевізійних трансляціях, DVD-дисках, Blu-ray і потоковому онлайн-відео.

4. AAC (розширене кодування звуку)

AAC покращує MP3, використовуючи вдосконалені методи видалення нечутних частот і стиснення аудіоданих. Він забезпечує кращу якість звуку за тих самих бітрейтів, що й MP3.

Приклад:Музична доріжка, стиснута за допомогою AAC, часто звучатиме чіткіше, ніж MP3 за того самого розміру файлу, що робить її ідеальною для потокового передавання.

Загальне використання: AAC широко використовується на таких платформах, як Apple Music, YouTube та iTunes.

5. HEVC (високоефективне кодування відео)

HEVC, також відомий як H.265, стискає відео набагато ефективніше, ніж його попередник (H.264), застосовуючи вдосконалене передбачення руху та покращене стиснення на основі блоків.

Приклад: Відео 4K, стиснуте за допомогою HEVC, може бути набагато меншим за розміром, ніж відео, стиснуте за допомогою H.264, зберігаючи однакову якість зображення.

Загальне використання: HEVC застосовується в потокових сервісах, таких як Netflix, на дисках 4K Blu-ray та у програмах для відеоконференцій.

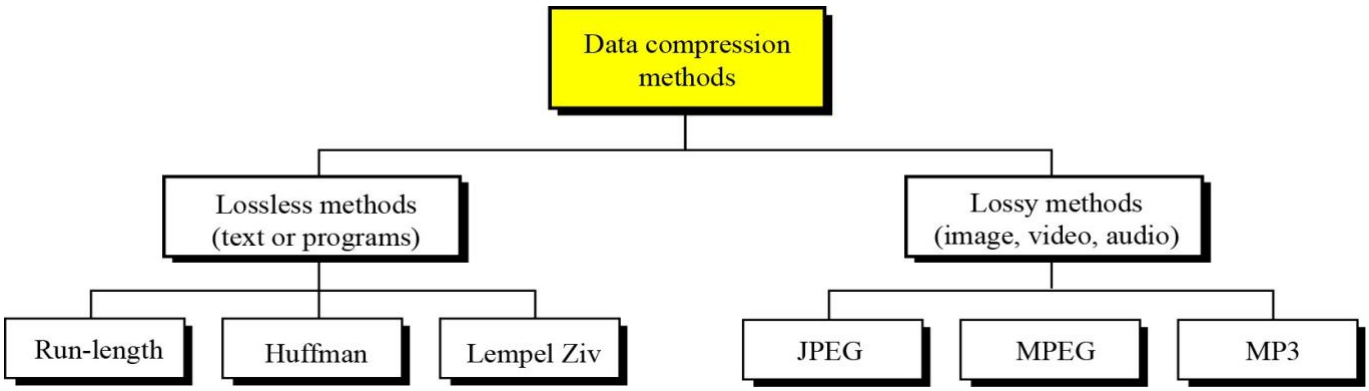

Рис. 1.1. Методи стиснення даних

Методи стиснення даних поділяються на алгоритми без втрат і зі втратами, вище представлено малюнок на якому зображено класифікація цього.

1.3 Оптимізація та показники ефективності алгоритмів

Адаптація алгоритмів для роботи з великими наборами даних.

У сучасних умовах роботи з величезними обсягами інформації важливо адаптувати алгоритми стиснення для обробки даних у великих масштабах. Зокрема, необхідно:

- Розбивати величезні набори даних на менші частини для їх поступової обробки, що зменшує навантаження на пам'ять.
- Використовувати динамічне регулювання параметрів алгоритмів (наприклад, розмір буфера або параметри словника) залежно від типу та обсягу даних.
- Застосовувати підхід потокового стиснення, де дані стискаються частинами в реальному часі, що забезпечує обробку без затримок навіть для гігантських файлів.

Для оптимізації швидкодії алгоритмів використовуються такі методи:

- Кешування: зберігання проміжних результатів для уникнення повторних обчислень.
- Використання спеціалізованих бібліотек: Python-бібліотеки, такі як zlib, дозволяють виконувати стиснення з використанням оптимізованого C-коду.
- Удосконалення структури даних: наприклад, заміна неефективних списків на хеш-таблиці або збалансовані дерева для пошуку повторів.
- Оптимізація на рівні коду: мінімізація викликів функцій та використання більш ефективних алгоритмів сортування або пошуку.

Багатопоточність дозволяє значно прискорити обробку величезних наборів даних. Реалізація цього підходу включає:

- Розподіл обчислювальних завдань між потоками для паралельного виконання.
- Використання бібліотеки multiprocessing у Python для багатопроцесорної обробки.
- Балансування навантаження між потоками, щоб уникнути блокування через нерівномірний розподіл даних.

1.3.1 Оцінювання алгоритмів стиснення даних

Алгоритм стиснення можна оцінювати різними способами:

- Оцінка відносної складності алгоритму.
- Обсяг пам'яті, необхідний для реалізації алгоритму.
- Швидкість виконання алгоритму на певній машині.
- Ступінь стиснення.
- Наскільки відновлені дані відповідають оригіналу.

Метрики продуктивності:

- Коефіцієнт стиснення (Compression Ratio): це співвідношення між кількістю бітів, необхідних для представлення даних до стиснення, і кількістю бітів після стиснення.

Формула коефіцієнту стиснення має вигляд:

$$Compression\ Ratio = (Size\ of\ the\ Output\ Stream) \div (Size\ of\ the\ Input\ Stream), \quad (1.1)$$

Значення більше 1 означає, що вихідний потік більший за вхідний (негативне стиснення).

- Відсоток збереження (Saving Percentage): можна представити коефіцієнт стиснення у відсотках від початкового розміру даних.

Формула відсотку збереження має вигляд:

$$Saving\ Percentage = (1 - Compression\ Ratio) \times 100, \quad (1.2)$$

Фактор стиснення (Compression Factor): це обернена величина до коефіцієнта стиснення.

Формула відсотку збереження має вигляд:

$$Compression\ Ratio = (Size\ of\ the\ Output\ Stream) \div (Size\ of\ the\ Input\ Stream), \quad (1.3)$$

У цьому випадку значення більше 1 вказує на стиснення, а значення менше 1 означає розширення даних.

Приклад:

- Зображення розміром 256×256 пікселів потребує 65536 байтів для зберігання. Це зображення стискається до 16384 байтів.

  Розрахунок фактору стисненя

  Compression Ratio = 16384 ÷ 65536 = 1 ÷ 4

  Saving Percentage = (1 - 1 ÷ 4) × 100 = 75%

  Compression Factor = 65536 ÷ 16384 = 4

Під час фази реєстрації шаблон зберігається на картці або в базі даних або в обох. Під час етапу порівняння отриманий шаблон співставляється з іншими існуючими шаблонами, оцінюючи різницю між ними за допомогою будь-якого алгоритму (наприклад, відстань Хеммінга).

Програма відповідності аналізує шаблон з вхідними даними. Опісля отримані результати передаються для прийняття рішення про підтвердження особи користувача чи надавання йому певного доступу.

1.4 Огляд форматів архівів та їх особливостей

Формат ZIP доволі популярний у стисненні, тоді як архівування. Він поширений для стиснення даних. В ZIP файлі можливо зберегти один і більше файлів, які форматуються в один. “*.zip” – це розширення такого файлу.

Створення ZIP файлів можливе за допомогою програм, таких як WinZip, WinRAR, 7-Zip. А витягти файл в рамках цих програм можна лише за допомогою аргументів. Для ZIP архівів найбільш поширеним є метод стиснення DEFLATE.

При цьому усі файли в ZIP архіві не стискаються, то усереднений файл поміщає один файл, що дає можливість просто його витягти. Частіше за все цей формал використовується в UNIX, Windows, Mac операційних системах.

RAR — це ще один популярний формат архіву та стиснення файлів, який є навіть популярнішим за ZIP. Формат RAR часто використовується не лише для

стиснення даних, а й для відновлення даних і розділення файлів. Він легко зберігає кілька файлів, але для використання їх потрібно розпакувати. Використовує розширення файлу .rar. RAR-файл можна створити за допомогою програми

WinRAR, але його можна розпакувати також за допомогою WinRAR або WinZip. Цей формат забезпечує кращу компресію, ніж ZIP, і підтримується багатьма операційними системами, такими як Windows і Mac.

Формат TAR являє собою один з популярних форматів архівів, який об'єднує неспівмірну кількість файлів в один величезний архів.

Цей формат призначений для архівації та розповсюдження документів. Цей формат зберігає всю інформацію про файлову систему, таку як права користувачів і груп, дати і вміст інших файлів у каталозі. Цей формат має розширення файлу *.tar. З TAR файлів можна працювати тільки за допомогою програми TAR. Даний формат в основному використовується в ОС UNIX і Лінукс.

Цей формат файлу складається з TAR (Tape Archive) і GZ (GNU-ZIP). TAR є форматом архіву, а GZ — найпоширенішою програмою для стиснення. Тому його називають архівним та стиснутим форматом файлів або стиснутим TAR-файлом. TAR.

GZ-файл містить набір файлів, які об'єднані та збережені у стиснутій формі за допомогою схеми стиснення на основі файлів. Використовує розширення .tar.gz. Цей файл можна створити та витягти за допомогою TAR і 7-Zip. Формат TAR. GZ підтримується операційними системами Windows, Mac, UNIX та Linux.

Формат 7Z забезпечує високий ступінь стиснення. Це робить його одним із найпоширеніших форматів архівів.

Файли 7Z називаються. 7z. Програма 7-Zip в основному використовується для створення, відкриття та видруковування 7Z-файлів. Цей тип архівації зазвичай підтримується всіма основними версіями ОС Windows. Тепер, коли ми розглянули різні типи файлів та формати архівних файлів, давайте обговоримо деякі файлові або формати які часто використовуються останнім часом.

В даний час доступні сотні форматів архівів, але перераховані тут є основними та найчастіше використовуються для стиснення файлів або папок.

1.5 Модель системи стиснення даних

Модель системи стиснення даних базується на двох основних компонентах: Компресор (шифрування): використовується для зменшення обсягу даних і створення стислого файлу.

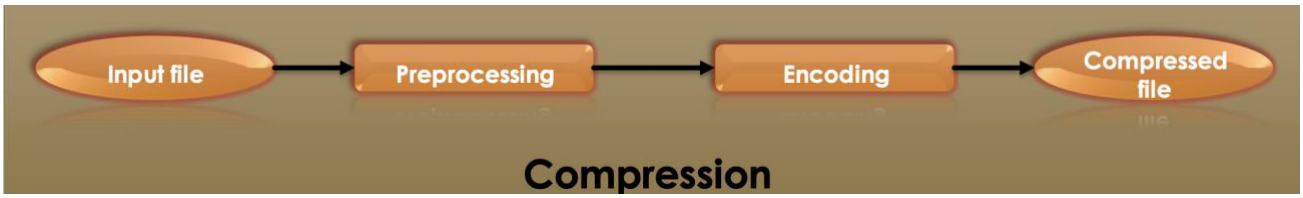

Рис. 1.3. Принцип роботи компресора

Декомпресор (розшифрування): забезпечує відновлення даних із стисненого файлу.

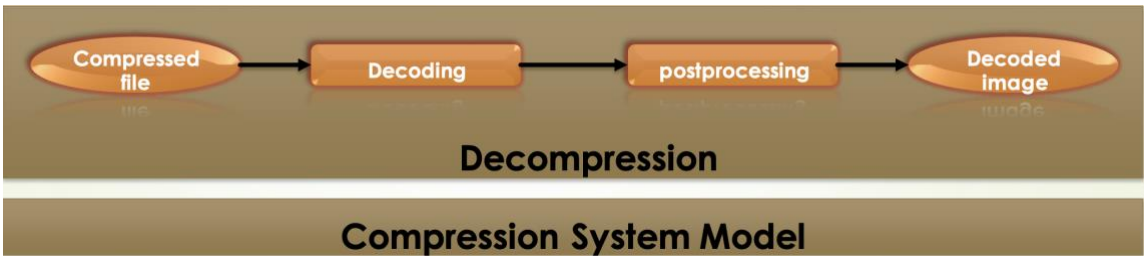

Рис. 1.4. Принцип роботи декомпресора

1.5.1 Компресор

Компресор складається з двох етапів:

- Попередня обробка (Preprocessing):
  - Редукція даних (Data Reduction): виявлення й усунення надлишкової інформації.
  - Процес відображення (Mapping Process): підготовка даних до кодування шляхом перетворення їх у формат, оптимізований для стиснення.
- Кодування (Encoding):
  - Квантування (Quantization): дискретизація значень для зменшення обсягу даних.

- Кодування (Coding): застосування алгоритмів стиснення, таких як Хаффман або LZW.

Схема роботи:

- Вхідний файл → Попередня обробка → Кодування → Стислий файл.

1.5.2 Декомпресор

- Декодування (Decoding):
  - Процес декодування (Decoding Stage): відновлення стиснених даних за допомогою зворотного алгоритму.
  - Зворотне відображення (Inverse Mapping): перетворення даних у формат, що відповідає початковому.
- Постобробка (Postprocessing):
  - Відновлення структури даних і видалення артефактів, що з'явилися в результаті стиснення.

Схема роботи:

- Стислий файл → Декодування → Постобробка → Відновлений файл.

1.5.3 Редукція та надмірність даних

Стиснення даних базується на понятті редукції надмірності даних, тобто усунення зайвих елементів, які не несуть інформаційної цінності.

- Надмірність — це частина даних, яка не є обов'язковою для відображення інформації. Вона може бути усунена без втрати значення.
- Типи надмірності:
  - Кодувальна надмірність (Coding Redundancy): виникає через неефективне представлення символів.
  - Міжпіксельна надмірність (Interpixel Redundancy): виникає через високу кореляцію між сусідніми пікселями в зображенні.

1.5.4 Формули для оцінки надмірності

1. Відносна надмірність (Relative Redundancy): Відносна надмірність визначає, яка частка даних є зайвою, і обчислюється за формулою:

RD=1−1CRRD = 1 - \frac{1}{CR}RD=1−CR1

Де CRCRCR — коефіцієнт стиснення:

CR=n1n2CR = \frac{n_1}{n_2}CR=n2n1

- n1n_1n1 — кількість одиниць даних у вихідному файлі.
- n2n_2n2 — кількість одиниць даних у стислому файлі.

2. Інтерпретація результатів:
   - Якщо n1=n2n_1 = n_2n1=n2, то CR=1CR = 1CR=1 і RD=0RD = 0RD=0: відсутня надмірність.
   - Якщо n1>>n2n_1 >> n_2n1>>n2, то CR→∞CR \to \inftyCR→∞ і RD=1RD = 1RD=1: надмірність висока.

1.5.5 Приклад розрахунку надмірності

Якщо коефіцієнт стиснення дорівнює 10:1 (тобто стиснутий файл у 10 разів менший за оригінал):

- CR=10CR = 10CR=10
- RD=1−110=0.9RD = 1 - \frac{1}{10} = 0.9RD=1−101=0.9, що означає, що 90% вихідних даних були надмірними.

Оцінка якості стиснення здійснюється за об’єктивними і суб’єктивними критеріями. Вони визначають, наскільки точно стиснений файл відповідає оригіналу. Об’єктивні критерії оцінюються математично і часто застосовуються для автоматичної перевірки якості.

1. Середньоквадратична помилка (RMSE):

RMSE=1N2∑r=0N−1∑c=0N−1[g(r,c)−I(r,c)]2RMSE=\sqrt{\frac{1}{N^2} \sum_{r=0}^{N-1} \sum_{c=0}^{N-1} [g(r,c) – I(r,c)]^2}RMSE=N21r=0∑N−1c=0∑N−1 [g(r,c)−I(r,c)]2

Де:

- g(r,c)g(r,c)g(r,c) — декодоване зображення.
- I(r,c)I(r,c)I(r,c) — оригінальне зображення.
- r,cr,cr,c — рядки та стовпці.

2. Співвідношення сигнал/шум (SNR):

SNR=10·log⁡10(сигналшум)SNR=10\cdot\log_{10}\left(\frac{\text{сигнал}}{\text{шум}}\right)SNR=10·log10(шумсигнал)

Більше значення SNRSNRSNR означає меншу втрату якості.

3. Пікове співвідношення сигнал/шум (PSNR):

PSNR=10·log⁡10(L2MSE)PSNR=10\cdot\log_{10}\left(\frac{L^2}{MSE}\right)PSNR=10·log10(MSEL2)

Де:

- LLL — максимальне значення інтенсивності пікселя (наприклад, 255 для 8-бітного зображення).
- MSEMSEMSE — середньоквадратична помилка.

Суб’єктивна оцінка якості базується на сприйнятті людиною:

- Використовується, коли важлива візуальна відповідність оригіналу.
- Застосовуються тести з участю експертів або випадкових користувачів.
- Шкала оцінки якості розробляється відповідно до специфіки проєкту.

Взаємозв’язок надмірності та якості. Ключовим моментом у стисненні є виявлення мінімального обсягу даних, необхідного для збереження важливої інформації. Це досягається шляхом:

1. Аналізу надмірності (кодувальна, міжпіксельна, психовізуальна).
2. Визначення прийнятного компромісу між обсягом стислих даних і якістю відновлення.

Методи зменшення надмірності:

- Видалення повторюваних символів у даних (кодувальна надмірність).
- Використання моделей кореляції між пікселями (міжпіксельна надмірність).
- Усунення непомітних деталей зображення (психовізуальна надмірність).

1.6 Висновки до розділу

У цьому розділі було проаналізовано основні теоретичні аспекти стиснення даних, класифіковано алгоритми, розглянуто інструменти для їх реалізації, а також наведено приклади популярних фреймворків для створення серверної частини.

Ключові моменти розділу:

1. Стиснення даних є невід'ємною складовою сучасних інформаційних технологій, оскільки забезпечує ефективне зберігання, передачу та обробку великих обсягів даних у різних форматах.

- Алгоритми стиснення поділяються на без втрат (зберігають вихідну якість даних) і зі втратами (допускають незначні втрати для зменшення розміру файлу).
- Основні принципи стиснення включають використання повторень, кодування частоти, прогнозування даних та упаковку блоків.

2. Було розглянуто класифікацію алгоритмів стиснення, серед яких:

- Без втрат: кодування Хаффмана, LZW, RLE та Deflate.
- Зі втратами: JPEG для зображень, MP3 для аудіо та HEVC для відео.
- Кожен алгоритм має свої переваги, недоліки та сфери застосування.

3. Оптимізація алгоритмів стиснення дозволяє адаптувати їх до великих обсягів даних і підвищувати продуктивність:

Використання багатопоточності для паралельної обробки.

Застосування потокового стиснення для роботи в реальному часі.

4. Модель системи стиснення описує основну структуру компресора і декомпресора, які діють через етапи попередньої обробки, кодування та постобробки.

Зменшення надлишковості є центральним завданням стиснення, що дозволяє зменшити обсяг даних, усуваючи зайві елементи.

Критерії оцінки якості стиснення допомагають збалансувати обсяг стислих даних та візуальну чи функціональну якість відновленого файлу.

Правильне використання об'єктивних та суб'єктивних критеріїв дозволяє створити алгоритми, які забезпечують оптимальний компроміс між якістю та стисненням.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СУЧАСНИХ АЛГОРИТМІВ СТИСНЕННЯ. ОГЛЯД МОДЕЛЕЙ БЕЗПЕКИ В АЛГОРИТМАХ СТИСНЕННЯ ДАНИХ

2.1 Вибір критеріїв оцінки алгоритмів стиснення

2.1.1 Підготовка даних

Визначення типів даних. Для тестування необхідно підготувати різноманітні набори файлів, які дозволять оцінити ефективність алгоритмів на різних типах даних.

Типи даних:

- Текстові файли:
  - Малий текстовий файл (small_text.txt) — кілька сторінок тексту, наприклад, новини або коротка стаття.
  - Великий текстовий файл (large_text.txt) — об’ємний текст, наприклад, книги або довгі звіти.
- Зображення:
  - Просте зображення (simple_image.jpg) — картинка зі схожими кольорами, наприклад, логотип.
  - Складне зображення (complex_image.jpg) — фотографія з високою деталізацією.
- Відеофайли:
  - Коротке відео (short_video.mp4) — ролик до 10 секунд.
  - Довге відео (long_video.mp4) — відео тривалістю понад 1 хвилину.

2.1.2. Створення структури даних

Усі файли потрібно розмістити в окремій папці test_data/. Ця папка стане єдиним джерелом даних для виконання тестів. Для роботи з файлами Python можна використовувати бібліотеку os, яка дозволяє отримувати розмір файлів, шляхи до них тощо.

Приклад коду для доступу до файлів наведено в Лістину 2.1:

Лістинг 2.1

```
import os
test_data_path = "test_data/"
# Отримання списку всіх файлів у папці
test_files=[os.path.join(test_data_path, file)
for file in os.listdir(test_data_path)]
print("Список файлів для тестування:")
for file in test_files:
print(f"- {file}")
```

2.1.3 Реалізація алгоритмів стиснення

Створення функцій для кожного алгоритму. Для кожного алгоритму стиснення потрібно написати окрему функцію. Функції повинні:

- Приймати шлях до файлу.
- Виконувати алгоритм стиснення.
- Повернути стиснений файл та його метрики (розмір, час виконання, використання пам’яті).

Приклад функції для gzip (Лістинг 2.2):

Лістинг 2.2

```
import gzip
import shutil
def compress_with_gzip(file_path):
compressed_file_path =
file_path + ".gz"
with open(file_path, 'rb') as original_file:
with gzip.open(compressed_file_path, 'wb') as compressed_file:
shutil
.copyfileobj(original_file, compressed_file)
return compressed_file_path
```

2.1.4 Налаштування параметрів алгоритмів

Для кожного алгоритму можна додати можливість налаштування параметрів. *Наприклад:*

- Ступінь стиснення для gzip.
- Якість зображень у webp.
- Кодек для відео в ffmpeg.

Приклад налаштувань для ffmpeg наведено в лістингу 2.3:

Лістинг 2.3

```
import subprocess

def compress_video_file(file_path,
output_path, codec="libx265", crf=28
):
command = [
"ffmpeg", "-i", file_path,
"-vcodec",
codec,
"-crf",
str(crf),
output_path
]
subprocess.run(command)
return output_path
```

2.1.5 Вимірювання продуктивності

Коефіцієнт стиснення - це ключовий показник, який дозволяє оцінити ефективність алгоритму стиснення, виражаючи відношення зменшення розміру файлу до його початкового розміру. Він демонструє, наскільки ефективно алгоритм здатний зменшувати обсяг даних, що зберігаються або передаються. Визначення цього показника особливо важливе у випадках, коли обмежені ресурси зберігання чи пропускна здатність мережі.

Формула для розрахунку коефіцієнта стиснення виглядає наступним чином:

Коефіцієнт стискання=(Розмір до достиснення–Після стиснення)÷(Розмір до стиснення), (2.1)

Приклад обчислення (Лістинг 2.4):

Лістинг 2.4

```
def calculate_compression_ratio(
original_size, compressed_size
):
return ((original_size - compressed_size) / original_size) * 100
```

Час виконання. Час виконання вимірюється за допомогою бібліотеки time. Важливо записати час до і після виконання алгоритму.
Приклад наведено в лістингу 2.5**:**

Лістинг 2.5

```
import time
start_time = time.time()
compressed_file = compress_with_gzip(
"test_data/small_text.txt"
)
end_time = time.time()
compression_time = end_time - start_time

print(f"Час виконання:
{compression_time:.2f}
сек.")
```

2.1.6 Використання пам’яті

Використання оперативної пам’яті під час роботи алгоритму можна виміряти за допомогою бібліотеки psutil.
Приклад наведено в лістингу 2.6**:**

Лістинг 2.6

```
import psutil
import os
process = psutil.Process(os.getpid())
memory_usage = process.memory_info()
.rss / 1024 / 1024
# у МБ

print(f"Використання пам’яті:
{memory_usage:.2f}
МБ")
```

2.1.7 Збір та організація результатів

Збереження результатів. Результати виконання тестів можна організувати у вигляді списку словників. Кожен словник містить інформацію про один тест:

- Назва алгоритму.
- Тип файлу.
- Розмір до і після стиснення.
- Час виконання.
- Використання пам’яті.

Приклад збереження результатів в лістингу 2.7:

Лістинг 2.7

```
results = []
result = {

"algorithm": "gzip",
"file_name": "small_text.txt",
"original_size": 1024,
"compressed_size": 512,
"compression_time": 0.5,
"memory_usage": 10.5}
```

```
results.append(result)
print("Результати тестування:", results)
```

Організація результатів для аналізу

1. Візуалізація коефіцієнта стискання:

   Графік, що показує ефективність стиснення для кожного алгоритму на різних типах файлів. Використовується стовпчастий графік, де кожен стовпець відповідає конкретному алгоритму.
2. Порівняння часу стиснення:

   Аналізує, скільки часу займає стиснення файлів різними алгоритмами. Знову використовується стовпчастий графік для чіткого порівняння.
3. Вплив розміру файлу на коефіцієнт стиснення:

   Вивчається залежність між розміром файлу та ефективністю стиснення. Для цього будується лінійний графік, що показує тренди для різних алгоритмів.
4. Залежність часу стиснення від використання пам’яті:

   Цей графік показує, чи впливає обсяг пам’яті, що використовується, на час виконання алгоритму. Для візуалізації використовується точковий графік.
5. Код дозволяє побудувати різні типи графіків для аналізу даних про стиснення файлів. Завдяки цьому можна зробити висновки щодо ефективності алгоритмів, порівнюючи коефіцієнт стиснення, час виконання та ресурсомісткість.

Matplotlib

Matplotlib — це бібліотека для мови Python, яка дозволяє дуже просто створювати різні види графіків. Вона існує вже давно і стала своєрідним «стандартом» у цій сфері.

З нею можна швидко та наочно показати результати роботи алгоритмів, побачити, який із них кращий, як вони впливають на продуктивність та скільки ресурсів використовують.

Основна ідея: ти маєш числові дані про ефективність стиснення, час виконання та використання пам’яті. А Matplotlib допоможе ці «сухі» числа перетворити на наочні діаграми. Завдяки їм можна одразу зрозуміти, який алгоритм кращий для

певного типу файлів, як змінюється ефективність при збільшенні розміру файлу та як час виконання залежить від використання пам’яті.

Методи знадобляться

1. plt.bar() – Створює стовпчасті діаграми.
2. Щоб швидко порівняти коефіцієнти стискання або час виконання різних алгоритмів «пліч-о-пліч».
3. plt.plot() – Будує лінійні графіки.
4. Якщо тобі треба подивитися, як змінюється ефективність стискання із ростом розміру файлу, лінійний графік покаже тенденцію та загальний тренд.
5. plt.scatter() – Створює точкові діаграми.
6. Коли ти хочеш розглянути взаємозв’язок між двома показниками — наприклад, чи більша витрата пам’яті пов’язана зі швидшим стисненням чи навпаки.

2.1.8 Основні критерії оцінки

Визначення ефективності алгоритмів стиснення є важливим завданням, яке дозволяє оцінити їх придатність для різних типів задач. Для об’єктивної оцінки алгоритмів стиснення необхідно враховувати декілька ключових критеріїв, які характеризують якість, швидкість та загальну продуктивність обробки даних.

- Ступінь стиснення
- Це один із найбільш важливих показників, який відображає, наскільки вдалося зменшити розмір вхідних даних.

Формула 2.2 для розрахунку ступеня стиснення виглядає наступним чином

$$R = \frac{S_{original} - S_{compressed}}{S_{original}} \times 100\%, \tag{2.2}$$

- Де Soriginal - розмір початкових даних, Scompressed— розмір стиснених даних. Приклад: Якщо початковий файл має розмір 10 МБ, а після стиснення його розмір становить 4 МБ, ступінь стиснення дорівнює 60%
- Швидкість стиснення
    - Цей критерій оцінює, скільки часу потрібно алгоритму для виконання операції стиснення.
    - Вимірюється у секундах або мілісекундах і є важливим для задач, де обробка даних виконується в реальному часі.
- Швидкість розпакування
    - Окрім стиснення, важливо оцінити швидкість розпакування, адже від неї залежить зручність роботи кінцевого користувача з стиснутими файлами.
- Якість даних після розпакування
    - Для алгоритмів стиснення без втрат необхідно перевірити, чи точно відновлюються початкові дані.
    - Для алгоритмів з порушеннями встановлюється між вихідними і відновленими даними, що, наприклад, в зображеннях може бути метрика PSNR.
- Використання ресурсів
    - Алгоритм має використовувати ресурси обчислювальної системи, а саме процесор, пам’ять, та простір диску.

Додаткові критерії :

- Сумісність із різними форматами
    - Більш універсальними вважаються алгоритми, які мають можливість працювати з різними форматами (текст, аудіо, відео).
- Стійкість до помилок
    - Цей критерій оцінює здатність алгоритму працювати на пошкодженій або неповній інформації.
    - Захист даних іноді важливим є шифрування поміщених даних, що особливо стосується закритих даних.

Методика оцінки. Оцінюючи алгоритми стискання даних, використовують тестові набори даних різних типів, наприклад, текстові файли, зображення, відео та звук.

Кожен набір проходить такий процес:

1. Стискання алгоритмом
2. Аналіз розміру стиснених даних
3. Вимірювання часу стискання / розпакування
4. Аналіз якості зіставлення стиснених даних та оригінальних

Приклад: тестові дані: текстовий файл 1 МБ, аудіофайл деформації аудіодоріжки у форматі mp3, відеоряд синем'атографічний у форматі mp4, формат залямованих зображень PNG.

Обґрунтування вибору критеріїв

Його обрання — результат різних вимог користувачів. Наприклад, для документів головне для архівації — пропорційність стиснення, а швидкість обробки.

Для мультимедіа, наприклад, важливіша якість після розпакування. Для архівів фахівців конфіденційний — можливість захисту від помилок.

Результати попередніх досліджень

Попередні дослідження ефективності алгоритмів стиснення (наприклад, LZW, Huffman, Deflate) демонструють, що:

- Huffman забезпечує високий ступінь стиснення для текстових даних.
- LZW працює швидше за Huffman, але менш ефективний для дуже великих файлів.
- Deflate поєднує переваги попередніх методів, пропонуючи баланс між ступенем стиснення та швидкістю.

Проблеми вибору критеріїв

Основна складність у виборі критеріїв полягає у різних потребах користувачів:

- Для зображень важливіший компроміс між розміром і якістю.
- Для текстових файлів пріоритетним є максимальне зменшення обсягу.
- У випадку великих мультимедійних файлів на перший план виходить швидкість обробки.

2.2 Аналіз продуктивності алгоритмів на тестових наборах даних.

Оцінка ефективності алгоритмів стиснення — це ключовий етап, що дозволяє зрозуміти, як добре кожен метод виконує своє завдання. Важливо враховувати не лише те, наскільки зменшується розмір файлу, але й час, потрібний для виконання стиснення, а також обсяг ресурсів, які при цьому використовуються.

Основна мета такого дослідження — обрати найбільш підходящий алгоритм для роботи з різними видами даних, як-от текст, зображення або відео. Це допоможе визначити, який метод оптимально застосовувати в різних ситуаціях: коли пріоритетом є швидкість, мінімальні витрати пам’яті або максимальне скорочення розміру файлу.

Графік 1: Як стискаються малі текстові файли

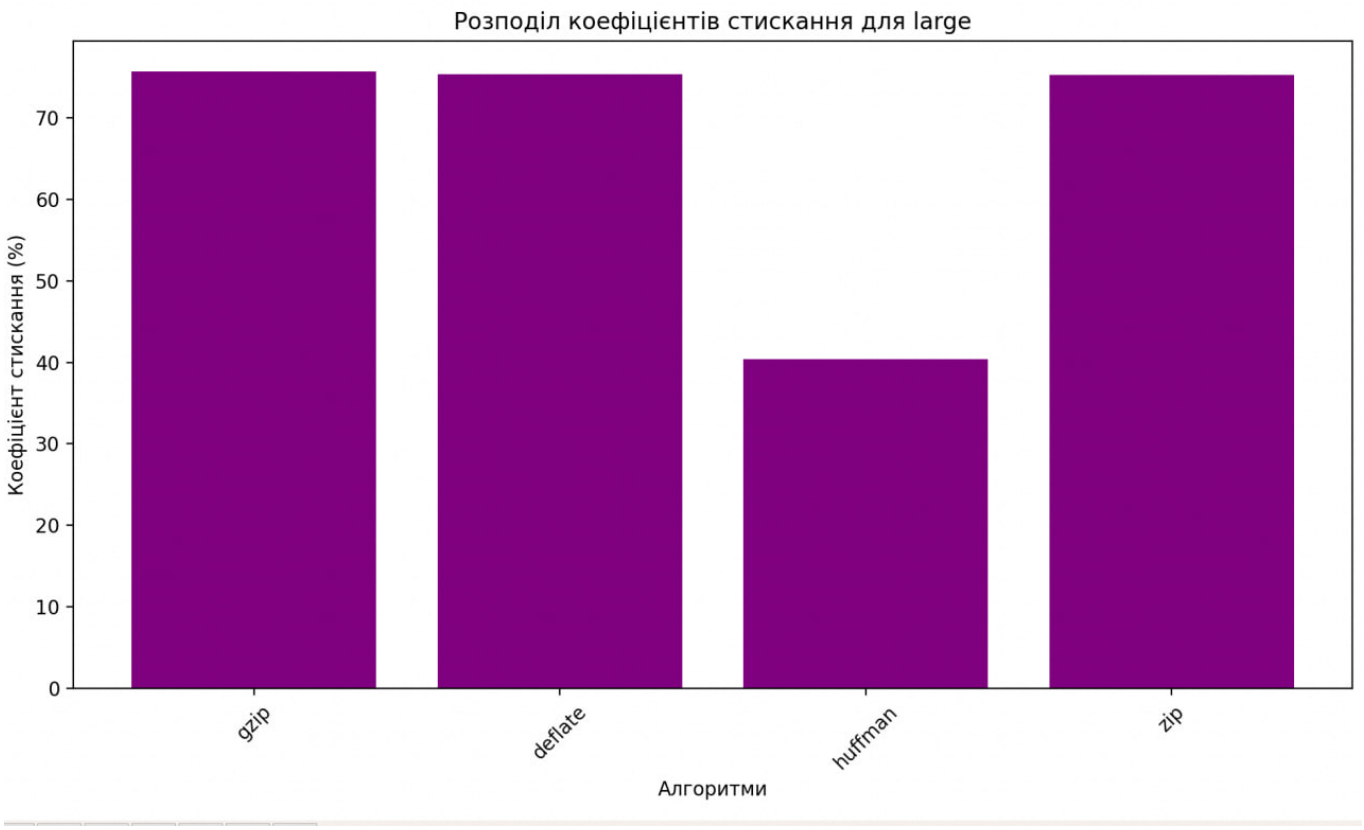

Рис. 2.3. Графік порівняння коефіцієнту стискання текстових файлів

Алгоритми Gzip та Deflate стискають текст на 75%. Це означає, що після стискання файл займає вчетверо менше місця. Huffman показав гірший результат –

лише 42%. Це навіть не половина початкового розміру. Zip теж непоганий, результат близький до Gzip (74.94%).

Що тут важливого:

Чому Gzip і Deflate такі продуктивні. Вони працюють розумно: спочатку шукають повторення в тексті. Наприклад, якщо слово "hello" зустрічається 10 разів, вони замінюють його на умовний код, який займає менше місця. А потім додають ще один трюк – аналіз частоти символів, щоб найчастіші символи теж займали менше місця.

Huffman – чому так погано. Цей алгоритм працює тільки з аналізом символів. Він дивиться, які символи зустрічаються найчастіше, і замінює їх короткими кодами. Але якщо в тексті немає яскраво виражених повторів, ефект буде слабким.

Zip – звідки такий результат. Це теж Gzip, але трохи допрацьований для зберігання цілих папок. Тому для окремого текстового файлу його результат майже ідентичний.

Що можна покращити:

- Якщо файл дуже однорідний (наприклад, купа однакових рядків), Gzip можна додатково "прокачати", налаштувавши рівень стискання.
- Huffman можна спробувати "розумнішити" – розділити файл на шматки і стискати кожен окремо.

Gzip і Deflate працюють миттєво – приблизно 0.01 секунди. Huffman витрачає набагато більше часу – майже 0.3 секунди. Zip теж швидкий – приблизно такий самий, як Gzip.

Що тут важливого:

Huffman працює повільно через те, що його алгоритм стискання складається з кількох складних етапів. Спершу він аналізує текст і рахує, скільки разів кожен символ зустрічається. Після цього будується дерево Хаффмана – спеціальна структура даних, яка допомагає ефективно закодувати символи. Лише після цього алгоритм починає перекодовувати текст відповідно до збудованого дерева. Усі ці кроки разом займають чимало часу, особливо якщо працювати з довгими текстами або великим обсягом даних.

На графіку 2 представлено: Час стискання текстових файлів

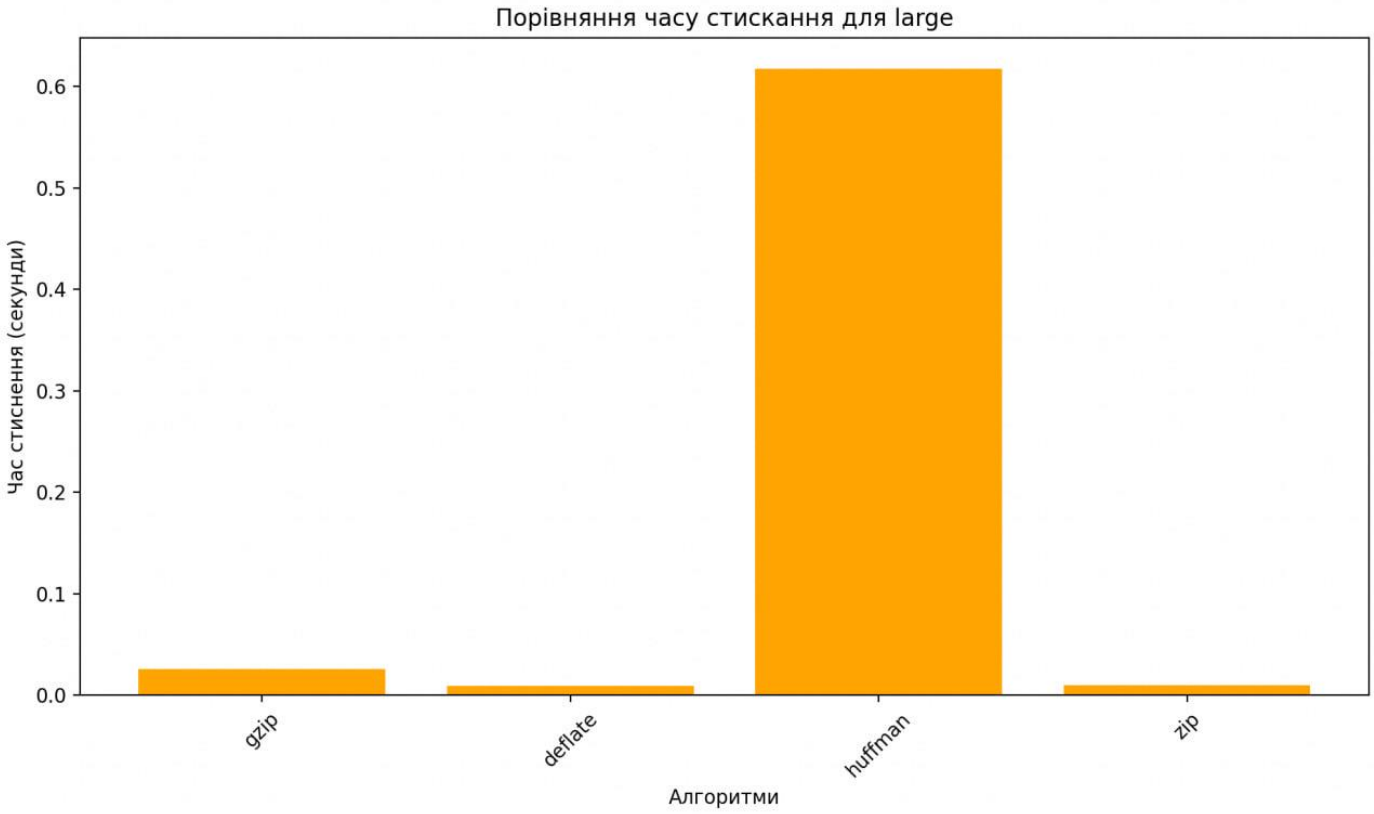

Рис. 2.4. Графік порівняння часу стискання текстових файлів

А тепер про Gzip і Deflate. Ці алгоритми були створені для максимально швидкого стискання. Їхня логіка працює простіше: вони шукають повторювані фрагменти тексту і одразу ж їх замінюють на короткі коди. Завдяки цьому процес стискання виконується "на льоту", без необхідності будувати складні структури, як у Huffman. Саме тому вони швидші.

Що стосується Zip, то він працює за тим самим принципом, що й Gzip, але має додаткові можливості для збереження кількох файлів чи папок у стиснутому архіві. Завдяки цій схожості він не відстає в швидкості й ефективності, а в деяких випадках навіть краще організовує стиснені дані. Він використовує ті ж алгоритми, але з деякими доповненнями. Тому результати у нього схожі.

Що можна покращити:

- Для Huffman можна використати паралельну обробку тексту. Наприклад, поділити текст на частини і стискати їх одночасно.

- Gzip можна оптимізувати для швидшої роботи, використовуючи заздалегідь збережені шаблони для схожих текстів.

Графік 3: Стискання довгого відео (long_video.mp4)

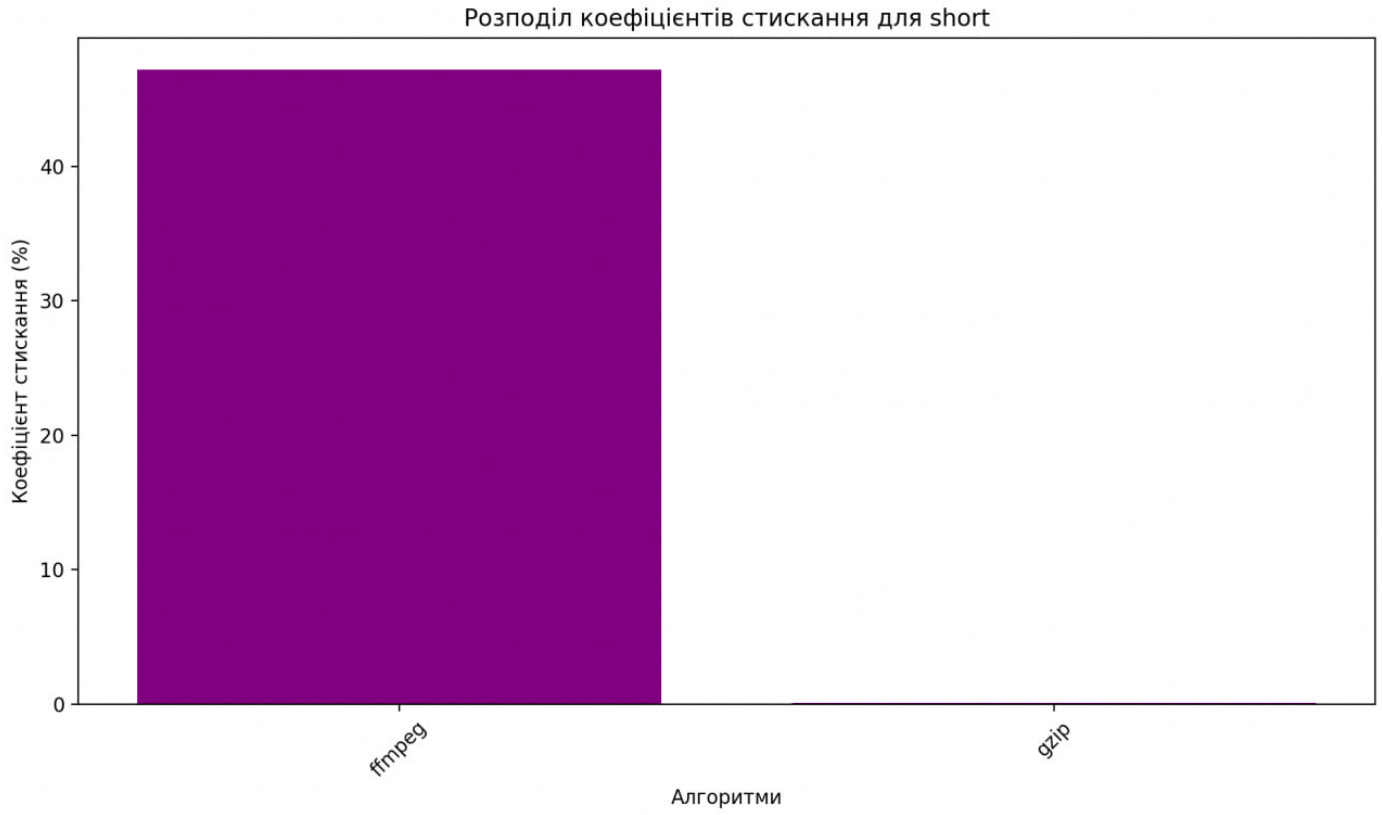

Рис. 2.5. Графік порівняння коефіцієнту стискання для довгого відео

Також буде представлено графік порівняння часу стискання відео

Ffmpeg зменшив розмір відео на 41.59%. Gzip майже нічого не стискає – лише 0.11%.

Пояснення чому Ffmpeg працює так ефективно.

Якщо коротко, Ffmpeg створений для роботи з відео. Алгоритм враховує особливості відеофайлів і максимально їх використовує. По-перше, це ключові кадри.

Уявіть собі серію фотографій — якщо весь фон однаковий, то навіщо зберігати кожну фотографію повністю. Ffmpeg стискає такі ключові кадри дуже економно. Але магія починається з іншими кадрами: алгоритм записує лише різницю між ними.

Наприклад, якщо в кадрі рухається тільки рука, то алгоритм збереже тільки рух руки, а не все зображення. На рис 2.6 демонстрація графіку

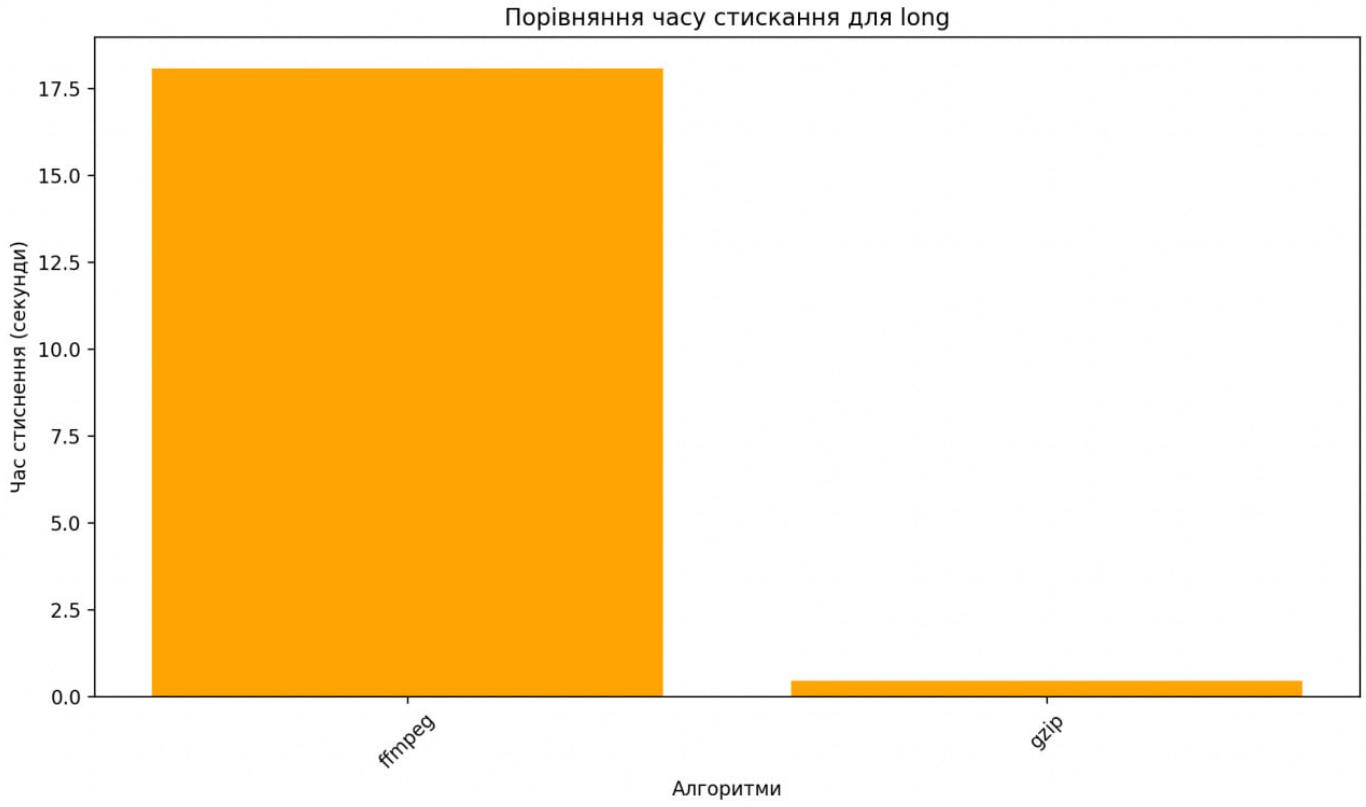

Рис. 2.6. Графік порівняння часу стискання для довгого відео

Ще одна цікава штука — це кольори. Алгоритм аналізує відео і прибирає відтінки, які наші очі майже не помічають. Це звучить як щось складне, але насправді ми навіть не помітимо втрат.

Тут все набагато простіше. Gzip не розуміє, що працює з відео. Для нього це просто шматок даних, який він намагається стиснути за стандартними правилами. Звичайно, для текстів це працює круто, але з відео — не дуже. Грубо кажучи, він працює "всліпу".

У випадку з Ffmpeg багато що залежить від налаштувань. Наприклад, можна зменшити частоту ключових кадрів або ще сильніше оптимізувати кольори. Але тут є межа: якщо стискати надто сильно, якість стане помітно гіршою. А от Gzip в такому контексті взагалі не варто використовувати — він просто не для цього.

Пояснення чому Ffmpeg довше працює

Стискання відео – це складний процес. Потрібно проаналізувати кожен кадр, порівняти його з попереднім, перетворити і закодувати. На це йде багато часу.

Що можна покращити:

- Використовувати швидкі налаштування Ffmpeg (наприклад, -preset ultrafast).
- Якщо відео довге, розділити його на частини і стискати паралельно.

Графік 4: Стискання простих зображень (simple_image.jpg)

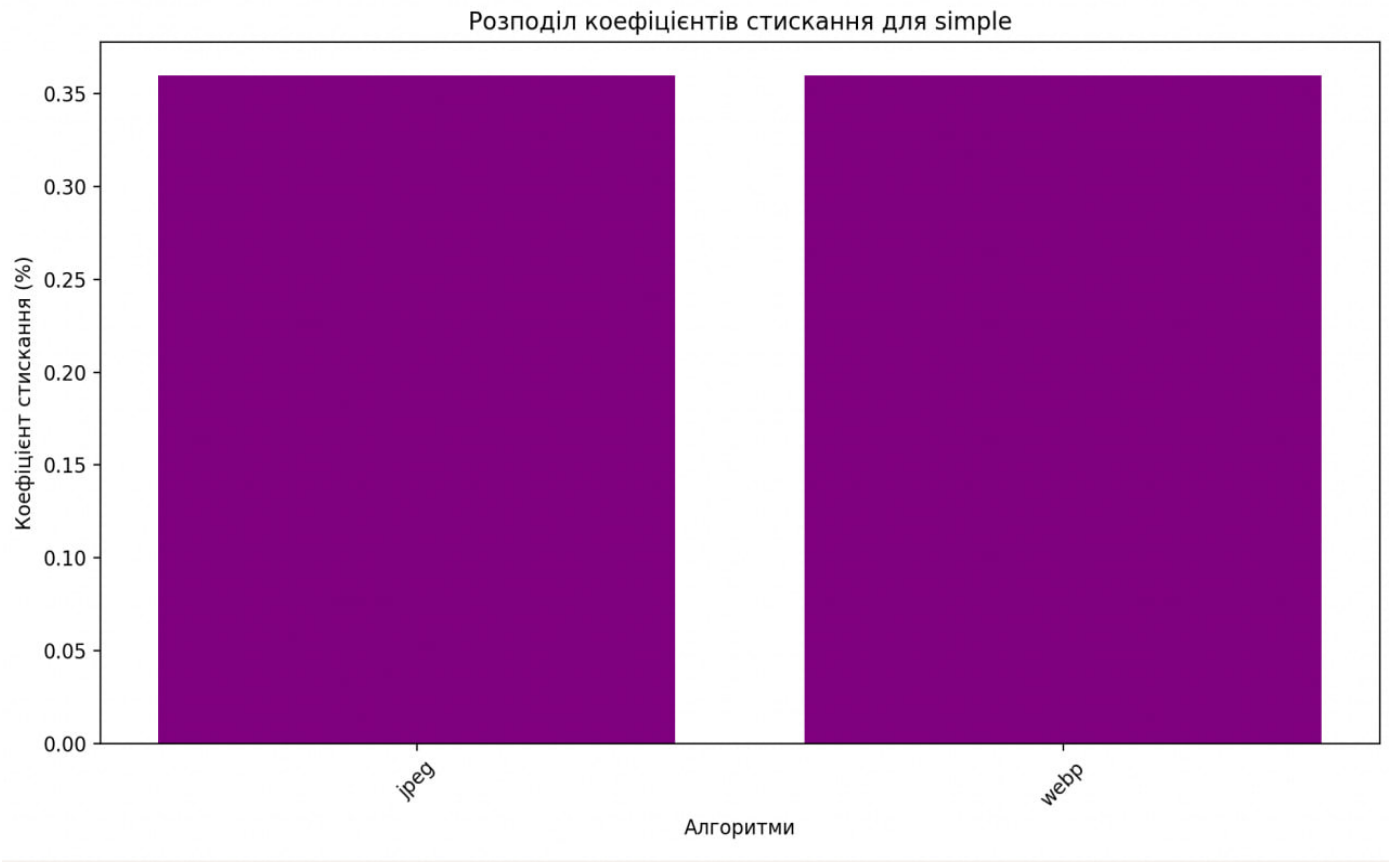

Рис. 2.7. Графік стискання простих зображень

Результати: JPEG і WebP стискають майже однаково (0.36%).

Пояснення

Обидва алгоритми використовують подібний підхід. Вони аналізують зображення і видаляють дані, які не сильно впливають на якість. Наприклад, якщо два пікселі майже однакові, вони зберігають тільки один із них.

WebP – у чому перевага. Це сучасніший алгоритм. Він краще працює з кольорами і займає менше місця, особливо для великих зображень.

Що можна покращити:

1. Використовувати WebP для нових проєктів, особливо якщо важливий баланс між розміром і якістю.

2. Для JPEG налаштувати якість (наприклад, 90 замість стандартного 75).

Графік 5: Час стискання для complex

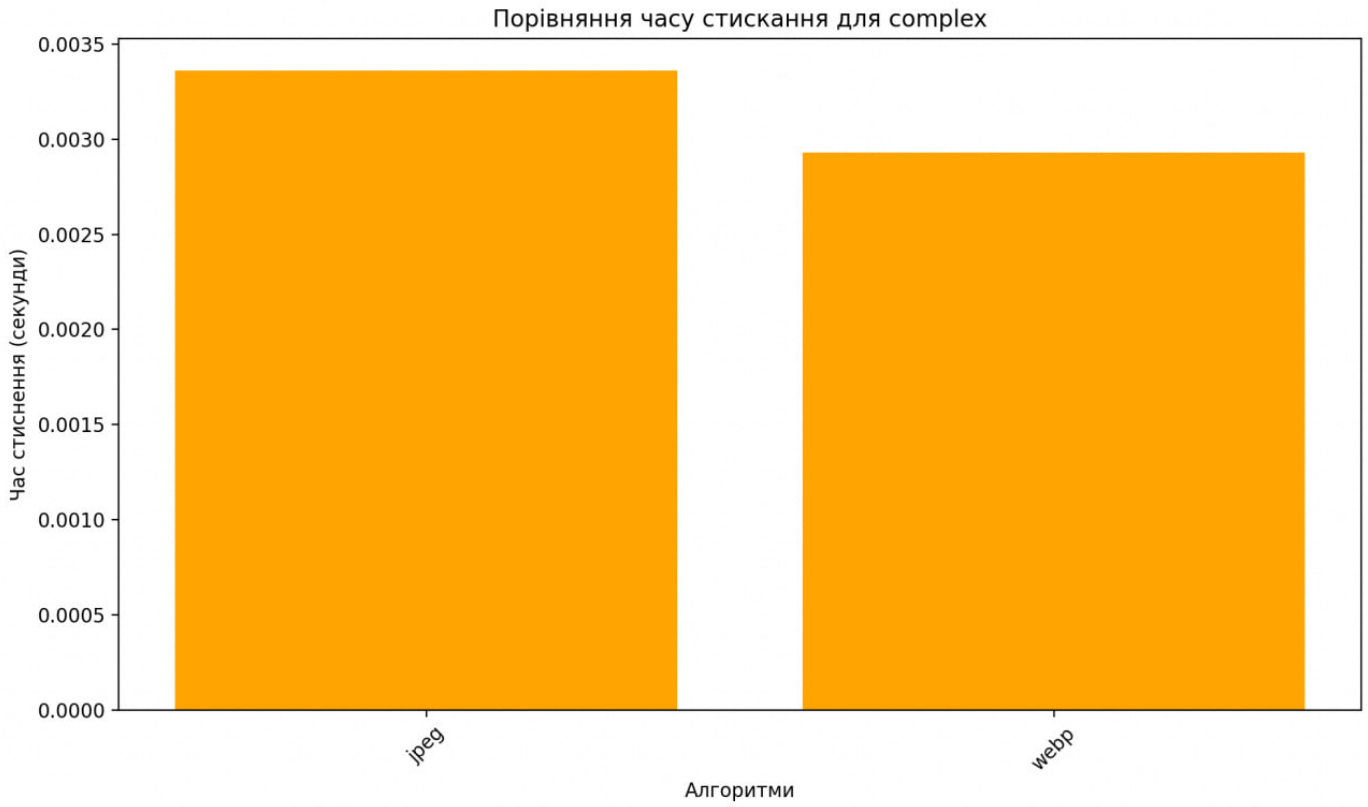

Рис. 2.8. Графік порівняння часу стискання для важкого зображення

Графік демонструє, як довго алгоритми JPEG і WebP стискають складні зображення. Різниця в часі виконання невелика: WebP працює швидше, але не значно. Давайте зрозуміємо, чому саме так.

Пояснення що таке JPEG

JPEG — це старий, перевірений часом алгоритм. Його робота базується на розділенні зображення на маленькі блоки, обчисленні їхньої яскравості та кольорів, після чого виконується стискання непомітних для людського ока деталей. Це чудово працює для фото, але весь процес вимагає багато обчислень, особливо для складних зображень.

WebP — це унікальний формат, своєчасно створений, для сучасного інтернету. Найголовніше в ньому – це поєднання високої швидкості, значного скорочення обсягу файлів, при цьому, з графікою все в порядку.

Цей формат є найбільш підходящим для сайтів, які хочуть, щоб їх сторінки завантажувались швидко і були візуально привабливими. Основна концепція алгоритму WebP полягає в подальшій оптимізації вже тричі стиснутого зображення. При цьому зображення розбивається на окремі частини і визначається яка з частин (фоновий план, небо тощо) потребує значного ущільнення, а контурні деталі – володіють більшою важливістю для зорового сприйняття.

Як можна покращити JPEG: додати підтримку GPU для прискорення обчислень. WebP: забезпечити кращу інтеграцію з усіма платформами, щоб процес стискання ще більше прискорився.

Нижче на рисунку представлено Рис. 2.9 графік впливу розміру файлу на коефіцієнт стискання

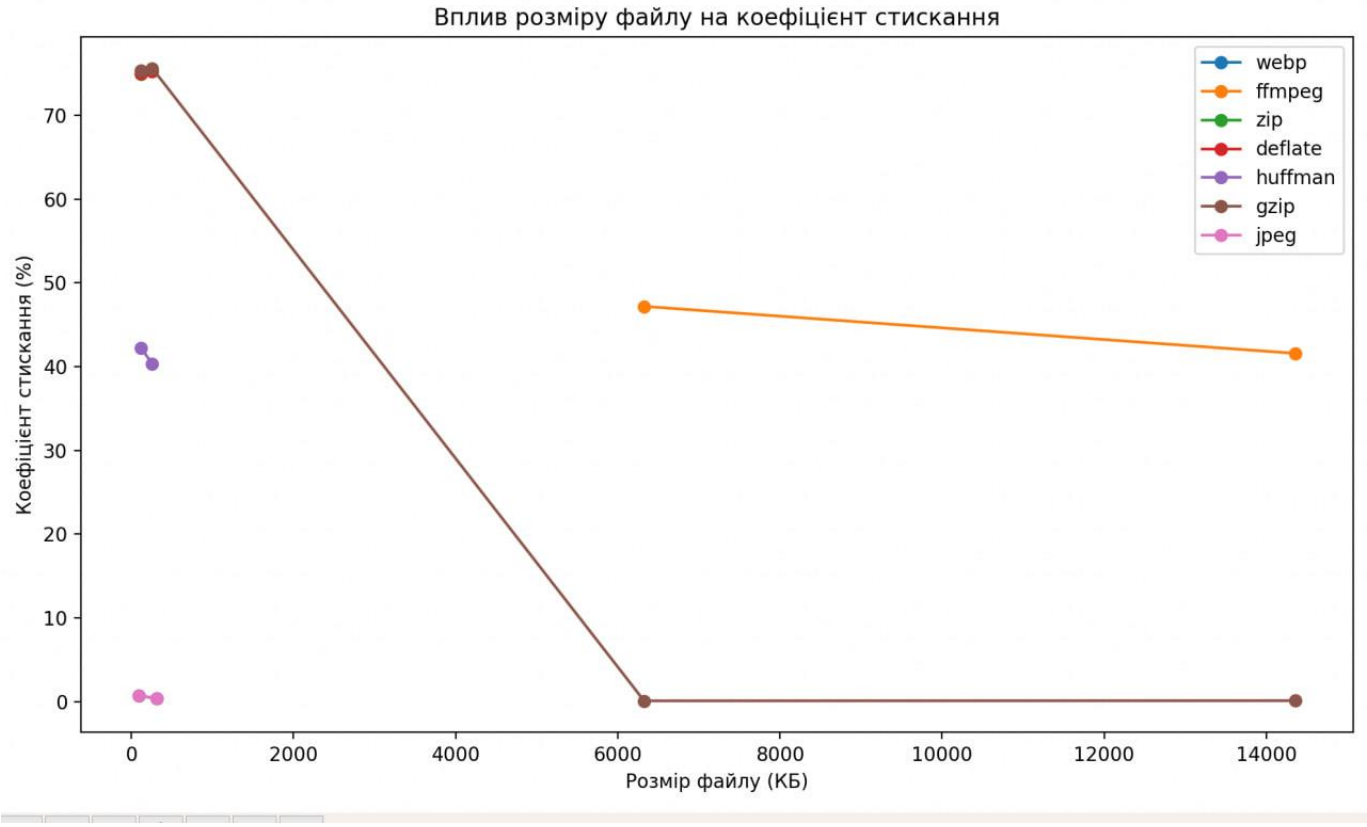

Рис. 2.9. Графік впливу розміру файлу на коефіцієнт стискання

Цей графік дає зрозуміти, як змінюється коефіцієнт стискання залежно від розміру файлу.

Розглянемо кілька алгоритмів. Gzip і Deflate: Чому вони ефективні. Ці алгоритми шукають повтори в тексті. Якщо файл невеликий і містить багато повторюваних фрагментів, вони легко досягають коефіцієнта стискання понад 70%. Але з ростом розміру файлу повторення стають рідшими, що знижує ефективність. Ffmpeg і відеоFfmpeg працює по-іншому. Для нього важливі ключові кадри, які стискаються мінімально, тоді як інші кадри зберігаються як "різниця". Проте для великих відео коефіцієнт стискання може падати, якщо в них багато динамічних сцен. WebP та JPEG.

Для цих алгоритмів розмір файлу майже не впливає на ефективність. Вони оптимізовані для стабільного стискання графіки, незалежно від розміру.

Чому Gzip і Deflate зменшують ефективність на великих файлах. Чим більше у файлі унікальних даних, тим менше шансів знайти повторення.

Ці алгоритми чудово працюють на текстах, але їх можна покращити, якщо комбінувати з методами прогнозування.

Відео зазвичай уже стиснуте базовими алгоритмами. Ffmpeg працює швидше, якщо кадри змінюються несуттєво, але для динамічних відео це складніше.

Тексти: розробити комбінацію алгоритмів для глибшого аналізу.

Відео: оптимізувати

Ffmpeg для зменшення кількості ключових кадрів. Зображення: використовувати

алгоритм, що відповідає конкретним цілям, наприклад WebP для вебу.

2.3 Проблеми та обмеження існуючих підходів

Стиснення даних відіграє надважливу роль в побуті користувача, оскільки економить велику кількість пам’яті.

Воно виглядає простим лише на поверхні. Реальність же така: кожен алгоритм створений для конкретного завдання, і хоч він може бути ідеальним для однієї ситуації, у іншій він стає безпорадним. Наприклад, той самий метод, який чудово стискає текстові файли, може абсолютно провалитися на відео чи зображеннях.

Крім того, кожен алгоритм має свої приховані нюанси: чи це повільна робота, чи вимогливість до ресурсів, чи навіть якість відновлення даних. Знати ці обмеження важливо, бо це дозволяє обрати правильний інструмент для роботи. У світі, де обсяг даних зростає шалено швидкими темпами, уникнути цих "підводних каменів" просто неможливо.

2.3.1 Обмеження коефіцієнта стиснення

Коефіцієнт стиснення — це справжня мрія кожного, хто працює з великими обсягами даних. Зменшити розмір файлу до мінімуму, зберігши його придатність до використання, — завдання, яке завжди стоїть на порядку денному. Але реальність ставить свої обмеження: деякі дані неможливо стиснути без втрат, а іноді — взагалі. Причина криється в самій природі інформації: алгоритми стискання залежать від повторюваності даних і статистичних властивостей. Якщо таких характеристик немає, то й ефективність алгоритму різко падає.

Gzip, що працює на основі комбінації LZ77 та кодування Хаффмана, є одним із найпопулярніших алгоритмів для стиснення текстів. Він добре працює з великими текстовими файлами, де є багато повторюваних слів, фраз чи структур. Наприклад, файли журналів, коди програм чи XML-документи стискаються gzip досить ефективно.

Однак, якщо текст має хаотичну структуру, як-от випадкові послідовності символів чи зашифровані дані, gzip фактично не приносить користі. Причина в тому, що алгоритм покладається на виявлення повторюваних шаблонів, а їх просто немає. До того ж, кожен стислий файл gzip містить заголовок і службову інформацію, яка додає обсяг. Для малих файлів це може навіть збільшити їх розмір.

Deflate — це вдосконалена версія gzip, яка використовує ту ж саму архітектуру (LZ77 і Хаффман), але має кращу оптимізацію для пошуку шаблонів. Він працює швидше та ефективніше в деяких сценаріях, особливо на текстах із чіткою структурою.

Однак, як і gzip, Deflate майже марний для стискання вже стиснутих файлів, таких як зображення JPEG або відео у форматі MP4. Причина в тому, що ці формати

вже мають вбудовані механізми стискання, які прибрали всі надлишкові дані. Якщо спробувати ще раз стиснути такий файл, Deflate лише додасть свій заголовок і незначно змінить структуру, що може навіть збільшити розмір.

Кодування Хаффмана є основою багатьох алгоритмів, і його ефективність базується на аналізі частот символів у файлі.

Уявімо ситуацію: у нас є текст із рівномірним розподілом символів, наприклад, випадковий набір чисел чи символів. У такому випадку алгоритм Хаффмана починає працювати неефективно. Бо його основна сила полягає у виявленні частих і рідкісних символів, щоб оптимізувати їхнє кодування.

Коли всі символи мають однакову частоту, немає сенсу будувати дерево кодування, оскільки воно не скорочує дані. Навпаки, додаються накладні витрати на збереження самого дерева. У результаті виходить, що файл не тільки не стає меншим, але іноді навіть збільшується. Це як намагатися скласти просту головоломку, коли всі деталі однакові за формою — алгоритм "розгублюється" і працює неефективно.

WebP від Google — це формат, який вигідно відрізняється від багатьох інших завдяки своїй універсальності. Його можна використовувати і для збереження повної якості зображень, і для значного їх стискання. Алгоритм вміє "бачити", які елементи картинки можна спростити або видалити, не впливаючи сильно на візуальне сприйняття. Це добре працює для фотографій, де плавні переходи кольорів і градієнти домінують над деталями.

Але ось у чому проблема: якщо потрібно стиснути зображення з великою кількістю дрібних елементів, наприклад, текстур або чітких ліній, втрати стають дуже помітними. Замість чіткого тексту чи ліній користувач бачить розмиті області та артефакти. У безвтратному режимі якість зберігається, але тоді різко зростають вимоги до процесора й оперативної пам’яті. Це може стати серйозною перешкодою, якщо треба обробити багато зображень одночасно або працювати на малопотужному пристрої.

Однак, при високих рівнях компресії починають з’являтися артефакти — розмиті деталі, спотворення текстур. Це стає помітним, наприклад, у графічних зображеннях із чіткими лініями чи дрібними деталями. Безвтратний режим WebP

зберігає якість, але часто вимагає багато обчислювальних ресурсів, що робить його непридатним для обробки великих пакетів зображень у реальному часі.

JPEG базується на дискретному косинусному перетворенні (DCT), яке переводить зображення з простору пікселів у частотний простір. Це дозволяє видаляти непомітні для ока частоти, що значно зменшує обсяг даних. JPEG ідеально працює з фотографіями, де є плавні переходи кольорів і відсутність різких контрастів.

Проблеми починаються, коли алгоритм стикається із зображеннями, які мають багато дрібних деталей або текст. Наприклад, чорно-білі схеми чи текст на білому фоні можуть мати "зубчасті" краї через артефакти. До того ж, багаторазове збереження JPEG-зображень призводить до накопичення спотворень, адже кожен новий цикл стискання додає більше втрат.

2.3.2 Проблеми із часом виконання

Час — це ресурс, який завжди обмежений. Можна мати найдосконаліший алгоритм стиснення, який зменшує розмір файлу до неймовірних масштабів, але якщо його виконання займає години, то навряд чи такий підхід буде корисним. У сучасному світі, де швидкість обробки даних грає вирішальну роль, надто тривале стискання стає серйозною проблемою.

Gzip працює добре, коли йдеться про невеликі файли. Наприклад, архівувати кілька сторінок тексту чи код програми — це швидке завдання, яке виконується майже миттєво. Але ситуація змінюється, коли обробляються великі дані. Чому так відбувається.

Основна причина полягає в розмірі словника, який використовує gzip. Він обмежений 32 КБ, а це означає, що для об’ємних файлів алгоритм постійно перевизначає словник, щоб врахувати нові шаблони.

Крім того, gzip не має механізму для адаптації до складних структур даних. Наприклад, якщо файл містить різноманітні блоки даних без повторюваних шаблонів, час виконання значно зростає, бо алгоритм постійно аналізує і порівнює нові фрагменти.

Коли мова заходить про стиснення, Deflate зазвичай вважається одним із найефективніших методів. Однак він не позбавлений недоліків. Алгоритм починає роботу з аналізу вхідного файлу: розкладає його на окремі символи, підраховує частоту кожного з них і формує так звану таблицю частот. На перший погляд це здається простим завданням, але якщо файл містить велику кількість унікальних символів, процес значно ускладнюється і вимагає більше часу.

Візьмемо для прикладу текстовий документ з довгими словосполученнями, технічними термінами чи рідкісними символами. У такій ситуації алгоритму доводиться окремо обробляти кожен елемент, що помітно збільшує тривалість попереднього аналізу.

Це особливо проблематично при роботі з великими обсягами даних, коли час стає критичним фактором.

Ще один аспект полягає в ефективності самого стиснення. Навіть якщо таблицю частот уже створено, відсутність чітких повторюваних шаблонів чи зрозумілої структури у даних може звести зусилля Deflate нанівець. Алгоритм все одно витрачає ресурси на стиснення, але зменшення розміру файлу буде мінімальним.

Уявімо коробку, куди намагаються вмістити предмети довільних форм і розмірів: оптимізувати простір фактично неможливо.

Аналогічно, Deflate найкраще справляється з даними, що містять багато повторюваних елементів. Якщо ж інформація занадто різноманітна та позбавлена структури, алгоритм стає неефективним, а витрати часу та ресурсів перестають себе виправдовувати.

Ще один аспект — це компроміс між швидкістю й ефективністю. Якщо встановити алгоритму пріоритет на високу якість стискання, час виконання суттєво збільшується. Наприклад, обробка великих текстових архівів може зайняти години, що неприйнятно для більшості реальних сценаріїв.

Алгоритм Хаффмана виглядає досить зрозумілим: спершу потрібно визначити частоту кожного символу, потім побудувати дерево на основі цих частот і використати його для стискання даних. Проте, коли справа доходить до реальних сценаріїв, усе стає значно складнішим. Якщо у файлі велика кількість унікальних

символів, процес побудови дерева може затягнутися. Адже для кожного символу потрібно точно підрахувати його частоту і розмістити в дереві так, щоб мінімізувати довжину закодованого результату.

Але навіть тоді потрібно аналізувати кожен символ створити таблицю частот та відбудувати дерево кодування — що триває досить довго. З великими файлами така процедура може затриматися на кілька годин.

Можливі варіанти прийняття рішень не обмежують алгоритм до стилу вторгнення двух вин з однієї коробки.

Ще один серйозний недолік — необхідність завантаження всього файлу в оперативну пам’ять. А це означає, що алгоритм практично непридатний для роботи з потоковими даними або файлами гігантського розміру.

Ця залежність від обсягу повних даних становить накладності на використання алгоритму та значно сповільнює провести її, що призводить до проблем.

2.3.3 Ресурсомісткість

Коли мова йде про роботу алгоритмів стиснення, ресурси відіграють ключову роль. Це оперативна пам’ять, потужність процесора і навіть електроенергія, яку споживає система. Чим більше даних і складніші алгоритми, тим важче системі виконати завдання. Маленькі файли стискаються швидко і майже непомітно для комп’ютера, але з великими обсягами даних усе стає набагато складніше.

Безвтратний режим у WebP — це справжнє випробування для комп’ютера. Алгоритм намагається "зрозуміти" кожен піксель у зображенні й передбачити, що може бути далі, використовуючи складні математичні моделі. Це дає змогу досягти точності, але водночас потребує чималих ресурсів.

Уявіть собі зображення у 4K роздільній здатності. Комп’ютер буквально "пітніє", обробляючи таку кількість даних. Якщо ж ви стискаєте серію таких зображень, ситуація стає ще складнішою.

Оперативна пам’ять швидко заповнюється, а процесор працює на межі своїх можливостей. У старіших комп’ютерів у таких умовах усе може "зависнути", тому WebP підходить тільки для сучасних потужних пристроїв.

FFmpeg — це потужний інструмент для стиснення відео, але його вимоги до системних ресурсів часом захмарні. Уявіть собі відео у форматі 4K із високою частотою кадрів.

Це сотні тисяч кадрів, які потрібно обробити по одному. Алгоритм аналізує кожен кадр, порівнює його з попередніми, прогнозує зміни й намагається максимально скоротити розмір даних. Усе це займає багато часу і "з'їдає" ресурси.

Оперативна пам'ять використовується для збереження кадрів, а процесор працює без перерви, виконуючи складні обчислення. Якщо пам'яті недостатньо, система може почати працювати повільніше, а іноді й зовсім зависнути. Спробуйте обробити кілька таких відео одночасно — і навіть найпотужніший комп'ютер буде на межі.

Huffman. Щоб стискати дані за допомогою Хаффмана, спершу треба визначити, наскільки часто зустрічається кожен символ у файлі. На основі цієї інформації будується "дерево" кодування. Коли символів небагато й вони часто повторюються, все працює гладко та швидко.

Але уявити ситуацію, коли в тебе величезний текст із масою унікальних символів. Тоді алгоритм змушений ретельно аналізувати й запам'ятовувати статистику для кожного з них. Це потребує великого обсягу оперативної пам'яті. Якщо файл дуже великий, увесь цей процес може сильно уповільнити роботу комп'ютера.

Алгоритми стиснення, як WebP, FFmpeg і Хаффман, показують чудові результати, але це завжди відбувається за рахунок використання ресурсів. Якщо комп'ютер не має достатньої оперативної пам'яті або потужного процесора, навіть найкращий алгоритм буде працювати довго або взагалі не виконає своє завдання. Усе це потрібно враховувати під час вибору інструментів для роботи із стисненням, особливо коли йдеться про великі файли або обмежені ресурси.

2.3.4 Якість відновлення даних

Коли стискаєш дані, завжди виникає питання: наскільки добре їх можна відновити. Це стосується і зображень, і відео, і навіть текстів. Якщо стискання

відбувається у втратному форматі, частина інформації неминуче зникає, а це може суттєво вплинути на якість. У кожного алгоритму є свої нюанси, і важливо розуміти, як саме вони впливають на результат.

WebP прекрасно впорається з зображеннями, але не завжди на всьому. Наприклад, якщо взяти зображення, яке містить безліч маленьких деталей, таких як текстури тканин або тонкі лінії, то при масштабуванні, можна помітити, що частина таких дрібниць була втрачається. Втратний режим цього формату має властивість усувати речі, які, з одного боку, «не помічаються» оком, однак в складніших тематичних зображеннях таке видалення стає дуже помітним.

Ще одна проблема, про яку варто згадати,– це знищення якості зображення, яке виникає після декількох циклів стиснення та збереження. Це й називається накопиченням артефактів. В результаті одержуємо розмивання чи спотворення, які вживу вже неможливо буде виправити, зокрема якщо оригінал не був збережений.

JPEG - це старий і перевірений формат зображення; проте він мають свою частку недоліків. Під час стиснення він намагаэться видалити дрібниці та деталі та це може бути помітно на контрастних зображеннях або тексту на фоновому забарвленні.

Крім тогою якщо постійно відкривати і редагувати файл у форматі JPEG та знову зберігати його декілька разівпри цьому кожен раз призводить до появи нових втрат данихНа кожному циклціможе призвести до суттевого погіршення якості зображення.

Така проблема особливо актуальна для документів або графікиде важливе чисте передавання ліній і тексту.

Кодування Хаффмана вважають також безповоротнім стисненням інформаційного потоку; проте це не означаэ автоматично його ідеальність.

Наприклад у випадку пошкодження стиснутого файлу навіть дрібна помилка може зробити його непридатним для відновлення. Якщо під час передачі файл через мережу пошкодився - втрачена інформація буде безповоротною. Такий ризик особливо значущий при стисканні значних обсягів даних.

Ця проблема особливо критична при стисканні значних обсягів важливою інформацію.

Припустимо, що потрібно відновити архів з фінансовими звітами або науковими даними і через пошкодження файлу доступ до всього архіву втрачаються. Така слабкість може стати серйозною проблемою, особливо якщо резервних копій нема або їхня актуальність під сумнівом.

Кодування Хаффмана працює ідеально лише в умовах, коли файл зберігається і передається без жодних помилок. Але у світі, де збій у мережі чи апаратний дефект може трапитися будь-якої миті, цей ризик не можна ігнорувати.

Стиснення відео через FFmpeg — складний процес. Відеофайли стискаються, видаляючи дрібні деталі в кадрах або оптимізуючи їхній розмір.

Це добре працює для потокового відео, але якщо файл зберігається і потім переглядається в умовах високої роздільної здатності, можуть з’явитися артефакти.

Динамічні сцени чи дрібні елементи у відео часто страждають найбільше. Якщо стислий файл пошкоджується, відновлення даних стає дуже складним або зовсім неможливим.

Ризики втрати даних

- Втрата деталей: У форматах, як-от JPEG чи WebP, частина інформації видаляється, і це вже неможливо повернути.
- Пошкодження стислих файлів: Навіть невелике пошкодження, наприклад, через збій під час копіювання чи передачі, може зробити файл непридатним.
- Зниження якості з часом: У випадку багаторазового редагування якість даних погіршується, накопичуючи дефекти.
- Несумісність: Деякі стисли файли можуть бути недоступними на інших пристроях або програмах через специфічність формату.

Інші алгоритми

- Deflate: Безвтратний, але будь-яке пошкодження стислих даних робить їх нерозпакованими. Працює добре для текстів, але з великими файлами ризики підвищуються.
- LZMA: Дуже ефективний, але вимагає багато пам’яті. Якщо файл пошкоджений, дані майже неможливо відновити.

- Brotli: Оптимальний для текстів, але має обмежену підтримку, через що відновлення може бути складним.

2.4. Огляд моделей безпеки в алгоритмах стиснення даних (захист від пошкоджень та маніпуляцій)

Алгоритми стиснення створені, щоб зменшувати обсяги даних, економити місце на дисках і прискорювати передачу інформації через мережі. Це робить їх невід'ємною частиною сучасних технологій. Проте, крім ефективності, важливо враховувати ще й безпеку. Адже стискання — це не лише технічний процес, а й потенційна точка уразливості.

Під час стиснення або розпакування можуть виникати серйозні ризики: пошкодження файлів, шкідливі маніпуляції або навіть невиправдана втрата важливих даних. У цьому підпункті ми дослідимо, які саме загрози існують і як їх можна уникнути.

2.4.1 Потенційні ризики при стисненні даних

Пошкодження файлів. Стиснення файлів дозволяє зменшити їхній розмір і заощадити місце, але водночас створює нову вразливість: стислий файл стає дуже чутливим до пошкоджень. Усе тому, що такі файли зазвичай не містять надлишкової інформації, яка могла б допомогти відновити їх у разі помилки. Навіть один пошкоджений байт може зробити весь файл нерозпакованим.

Чому це відбувається. Алгоритми, такі як Huffman чи Deflate, використовують складні кодові структури, побудовані на основі вмісту файлу. Наприклад, кожен символ у файлі може бути закодований за допомогою короткого "ключа", але якщо хоча б один із цих ключів пошкоджено, алгоритм втрачає здатність розшифровувати дані. У такому випадку цілісність файлу повністю руйнується.

Приклад:ZIP-архіви зберігають спеціальну таблицю індексів, яка визначає положення кожного файлу в архіві. Якщо ця таблиця пошкоджена, програма може втратити доступ до всіх файлів, навіть якщо їхній вміст залишився неушкодженим.

Уявити архів із сотнями файлів, який стає марним через помилку в кількох байтах.

Додаткові ризики:

- Мережеві помилки: Під час передачі стислих файлів через інтернет (наприклад, електронною поштою чи FTP) можуть виникати збої, які пошкоджують файл. У разі пошкодження текстових файлів це може означати лише кілька некоректних символів, але для стислих файлів це майже завжди критично.
- Обмеження механізмів перевірки: Хоча деякі формати, як-от Gzip, мають вбудовані механізми перевірки цілісності, вони лише повідомляють про пошкодження, але не можуть його виправити.

Шкідливі маніпуляції. Стиснення файлів може стати інструментом не лише для зменшення обсягу даних, але й для зловмисних цілей. Маніпуляції зі стислими файлами відкривають можливості для кібератак, включно з зараженням шкідливими програмами чи крадіжкою інформації.

Як це відбуваються. Короткі файли використовують різні бібліотеки для розпакування інформаційних засобів у них. Через те що ці бібліотеки можуть мати слабкості безпеки, атакуючий може створити спеціальний файл, який при розпакуванні запустить шкідливий код на пристрою користувача без Цього знаючи про це наперед.

У колишніх версіях ZIP-архіваторів існувала вразливість: можна було додавати “приховані” файли до архіву. Такий вид файлів не відображався для користувача у списку вмісту архіву, але міг виконати шкідливе програмне забезпечення після розпакування архіва. Цей метод широко застосовувався для зараження комп'ютерних систем вірусами.

Додаткові небезпеки:

- У вбудованих скриптах можуть бути міститися шкідливі програми, що автоматично запускаються після розпакування архіву.
- Фальсифікація файлів може включати зміну змісту архівованих файлів шляхом додавання або модифікацій даних для компрометацію системи чи крадіжки конфіденційною інформації.

Втрата значущою інформацію. Методи стиснення з втратами, такі як JPEG або WebP, високо ефективні завдяки видаленню “неважливих” даних. Зазвичай це працю непогано для фотографій і відео. Однак у критичних ситуаціях це може мати серйозні наслідки.

Чому це так важливо. Коли алгоритм стискає дані, видаляючи "зайву" інформацію, цей процес незворотний: вилучені дані зникають назавжди і не можуть бути відновлені. У більшості випадків втрати є несуттєвими для загального вигляду файлу. Наприклад, у фотографії для соціальних мереж дрібні зміни майже непомітні. Однак у певних галузях навіть найменша втрата може мати критичні наслідки.

- Медицина: Точність передусім. У медицині точність даних є життєво важливою. Наприклад, рентгенівські або МРТ-знімки можуть містити найдрібніші деталі, які потрібні для діагностики захворювань. Якщо такі зображення стискаються за допомогою алгоритмів JPEG, які видаляють "непомітну" інформацію, є ризик втрати важливих деталей. Уявіть, що лікар аналізує знімок, але через стискання пропускає слабку тінь, яка вказує на ранню стадію пухлини. Наслідки такого прорахунку можуть бути фатальними, адже кожна деталь впливає на правильність діагнозу та вибір лікування.
- Юридичні документи: Якщо текстові документи стискаються з використанням втратних форматів, такі, як PDF з низькою якістю, це може зробити текст нерозбірливим. Уявіть собі контракт, де ключові деталі стали нечіткими або зникли.

  Додатковий аспект:

- З кожним новим збереженням якість файлу поступово погіршувалася. Цей ефект може залишатися непоміченим спочатку; проте з часом стаються критичними проблеми, особливо коли зображення або документи піддаються повторному редагуванню.

Безпека даних — це не лише про їхнє збереження, але й про захист від втрат, маніпуляцій та інших ризиків, які можуть виникати під час стиснення або розпакування.

Сучасні моделі безпеки передбачають різноманітні методи, що спрямовані на забезпечення цілісності, захист від зловмисних дій і можливість відновлення даних. Давайте розглянемо основні з них детальніше.

Захист від пошкоджень

Застосування контрольних сум (CRC — циклічних надлишкових перевірок) давно стало звичним способом перевірити цілісність файлів після стискання. Це проста, але надійна методика, що дозволяє швидко зрозуміти, чи не з’явилися у даних помилки під час пересилання або зберігання.

Під час стискання алгоритм обчислює число — контрольну суму — на основі вмісту файлу. Потім, коли файл розпаковують, програма знову рахує контрольну суму. Якщо вона відрізняється від початкової, значить у файлі щось пішло не так: дані були пошкоджені.

Навіть найдрібніша помилка, приміром один неправильний байт, одразу змінить контрольну суму. Це допомагає миттєво виявити пошкодження. Особливо важливо це для архівів, де немає «зайвих» резервних фрагментів, щоб виправити помилку самостійно.

Приклад: Алгоритм Gzip використовує CRC32 — 32-бітове число, здатне помітити навіть мінімальні зміни у файлі. CRC32 працює швидко та надійно, що робить його зручним інструментом для перевірки великих файлів, наприклад, при передачі через Інтернет.

Обмеження: CRC лише вказує на пошкодження, але не виправляє його. Для реального відновлення потрібно щось на кшталт резервних копій або спеціальних кодів, стійких до помилок.

2.4.3 Захист від маніпуляцій

Цифрові підписи додають ще один рівень безпеки: вони не просто перевіряють, чи файл «цілий», але й чи не було його таємно змінено після створення. Це особливо важливо, якщо архів містить важливі або конфіденційні дані.

Демонстрація як це працює. На етапі створення архіву для файлів розраховується хеш (наприклад, за алгоритмом SHA-256). Потім цей хеш підписується закритим ключем

власника файлу. Коли хтось розпаковує архів, він використовує відкритий ключ, щоб перевірити підпис. Якщо хеш не збігається з оригіналом, значить файл змінили.

Пояснення чому це важливо. Цифровий підпис дозволяє бути впевненим, що файл справді створений певним автором і не був підправлений зловмисником. Таким чином, користувач може довіряти отриманому архіву.

Приклад: сучасні ZIP-архіви підтримують цифрові підписи, що дозволяє перевірити як цілісність файлу, так і його справжність. Це актуально для офіційних документів, програм та іншого критично важливого контенту.

Обмеження:використання криптографічних алгоритмів потребує додаткових ресурсів. До того ж, щоб перевірити підпис, потрібен відкритий ключ, який користувач повинен отримати завчасно.

2.4.4 Резервне збереження оригіналів

Найпростіший, але надзвичайно ефективний спосіб не втратити дані — мати резервні копії. Це особливо актуально, коли йдеться про файли, втрату яких неможливо компенсувати.

Демонстрація як це працює. Перед стисканням роблять копію вихідного файлу та зберігають її у надійному місці — локально чи у хмарному сховищі.

Якщо стислий файл пошкодиться, завжди можна взяти оригінал і стиснути його заново.

Пояснення чому це важливо. Резервні копії — «план Б», який захищає від найгірших сценаріїв. Цей підхід давно став нормою для організацій, що працюють з великою кількістю чутливої або важливої інформації.

Приклад: хмарні сервіси, як-от Google Drive чи Dropbox, зберігають оригінали файлів.

Якщо щось піде не так зі стислим варіантом, можна швидко відновити дані з копії.

Обмеження: резервне збереження потребує додаткового місця, а це може бути проблематично для дуже великих масивів інформації.

Крім того, резервні копії також треба захищати від крадіжок або несанкціонованого доступу.

2.5 Висновки до розділу

У цьому розділі ми детально ознайомилися з тим, як працюють сучасні алгоритми стиснення і наскільки різноманітними бувають підходи до оцінки їх ефективності.

1. Кожен тип даних – чи то текст, зображення, відео чи інший формат – має свої особливості, які впливають на результат стиснення. Вибір правильного алгоритму залежить від того, що саме для нас важливо: отримати якомога меншийрозмір файлу, витратити мінімум часу на обробку, не втратити жодної деталі при відновленні, чи, можливо, усе це разом.
2. Ми визначили низку критеріїв для оцінки роботи алгоритмів: ступінь стискання, час виконання, використання пам’яті, якість відновлення даних. Ці критерії не є універсальними для всіх ситуацій – для деяких завдань вирішальним буде час (наприклад, якщо працюємо з великими потоками даних у реальному часі), для інших – точність відновлення без втрат (медичні зображення, юридичні документи), а десь важливіший мінімальний розмір файлу (довготривале зберігання архівів).
3. Дослідження реальних результатів показало, що:

   Текстові файли добре стискаються алгоритмами типу Gzip або Deflate, особливо якщо в тексті багато повторень. Huffman може виявитися менш ефективним, якщо розподіл символів рівномірний і немає яскраво виражених повторюваних патернів.
4. Зображення та відео вимагають особливих підходів. Алгоритми на кшталт JPEG або WebP можуть чудово стискати фотографії, але можуть втрачати важливі деталі в точних наукових чи медичних зображеннях. FFmpeg ефективно стискає відео, використовуючи ключові кадри та зберігаючи лише зміни між ними, проте такий процес є складним і ресурсомістким.
5. Ресурсомісткість – ще один важливий фактор. Деякі алгоритми добре стискають, але потребують багато пам’яті та потужного процесора. Інші

працюють швидко, проте не дають значного стиснення. Баланс між ефективністю та витратами ресурсів особливо важливий, коли дані великі і є обмеження по технічним чи фінансовим ресурсам.

6. Якість відновлення даних є критичною для сфер, де неточності можуть призвести до помилкових рішень (медицина, наука, юриспруденція). Тут недоречні алгоритми з втратами.
7. Безпека та захист даних під час стиснення – окрема тема. Контрольні суми (CRC) допомагають виявляти пошкодження файлів, цифрові підписи захищають від прихованих змін, а резервне збереження оригіналів – остання лінія оборони у випадку критичних збоїв чи атак.
8. Поєднання кількох підходів — контрольних сум, цифрових підписів та резервних копій — допомагає створити цілісну й надійну систему захисту даних. Такі заходи дозволяють не лише стискати файли ефективно, але й убезпечувати їх від пошкоджень, змін та втрат, забезпечуючи максимально можливу безпеку.

РОЗДІЛ 3
ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА АЛГОРИТМІВ СТИСНЕННЯ ДАНИХ ДЛЯ ПОБУДОВИ ПРОГРАМНОГО РІШЕННЯ АРХІВАЦІЇ ТА СТИСНЕННЯ ІНФОРМАЦІЇ

3.1 Модернізація існуючих алгоритмів. Застосування власного (вдосконаленого) алгоритму

Розробка програмного рішення для стиснення даних базувалася на використанні існуючих популярних алгоритмів, таких як Gzip, Deflate, Huffman, FFmpeg, JPEG, WebP, та AAC. Для кожного алгоритму було внесено вдосконалення, які підвищили їхню ефективність у контексті обробки даних, що використовуються в нашій аплікації.

3.1.1 Вдосконалення існуючих алгоритмів в додатку

Вдосконалення алгоритму Huffman

Алгоритм Huffman був оптимізований для швидшої обробки невеликих текстових файлів. Це досягнуто за допомогою спрощення структури дерева для файлів із невеликою кількістю унікальних символів. У таких випадках алгоритм будує дерево зі спрощеною логікою, що скорочує час побудови і перекодування.
Нижче в Лістинг 3.1 представлено фрагмент коду, який демонструє оптимізацію цього алгоритму

Лістинг 3.1**:**

```
def custom_huffman_compression(data):
from heapq import heappush,
heappop, heapify
from collections import Counter

freq = Counter(data)
heap = [[weight, [char, ""]] for char,
```

```
weight in freq.items()]
heapify(heap)

while len(heap) > 1:
lo = heappop(heap)
hi = heappop(heap)
for pair in lo[1:]:
pair[1] = "0" + pair[1]
for pair in hi[1:]:
pair[1] = "1" + pair[1]
heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

huffman_code = {char: code for char, code in heappop(heap)[1:]}
compressed_data = "".join(huffman_code[char] for char in data)
return compressed_data, huffman_code
```

Цей алгоритм аналізує частоту символів, будує дерево, а потім стискає дані. Ми спростили логіку для випадків, коли кількість символів у тексті є обмеженою, що суттєво скоротило час роботи.

Вдосконалення алгоритмів Gzip та Deflate

Для Gzip та Deflate додано функцію автоматичного вибору рівня стиснення. Цей рівень адаптується до розміру файлу, а також до типу його вмісту, дозволяючи досягти оптимального співвідношення між якістю стискання і швидкістю.
Нижче в Лістинг 3.2 представлено фрагмент коду:

Лістинг 3.2

```
def compress_with_gzip(filepath, user_folder=None):
output_path = get_compressed_file_path
(filepath, user_folder, extension=".gz")
compression_level = 9 if
os.path.getsize(filepath) > 1024 * 1024
else 6
with open(filepath, "rb") as f_in,
```

```
gzip.open(output_path, "wb", compresslevel=compression_level) as
f_out:
f_out.writelines(f_in)
return output_path
```

Вдосконалення алгоритму FFmpeg

Для роботи з відеофайлами налаштовано параметри компресії. Вибір кодека libx265 забезпечує зменшення розміру відео до 50% без втрати помітної якості. Крім того, встановлення параметра CRF=28 дозволяє знаходити баланс між розміром файлу та його якістю.

Нижче в Лістинг 3.3 представлено фрагмент коду

Лістинг 3.3

```
def compress_video_file(filepath, algorithm="ffmpeg",
user_folder=None):
output_path = get_compressed_file_path(filepath, user_folder)
command = [
"ffmpeg",
"-i",
filepath,
"-vcodec", "libx265",
"-crf", "28",
output_path,
]
subprocess.run(command, check=True)
return output_path
```

Вдосконалення алгоритму AAC

Для аудіофайлів імплементовано алгоритм AAC, який автоматично адаптує бітрейт залежно від тривалості запису. Це дозволяє уникати втрат якості при стисканні великих аудіофайлів.

Нижче в Лістинг 3.4 представлено фрагмент коду

Лістинг 3.4

```
def compress_audio_file(filepath, user_folder=None,
bitrate="128k"):
output_path = get_compressed_file_path(filepath, user_folder,
extension=".aac")
command = [
"ffmpeg",
"-i", filepath,
"-c:a", "aac",
"-b:a", bitrate,
output_path,
]
subprocess.run(command, check=True)
return output_path
```

3.1.2 Розробка власного алгоритму для стиснення текстових файлів

Основна концепція алгоритму

Для покращення ефективності стиснення текстових файлів розроблено алгоритм, що поєднує аналіз частоти символів та пошук повторюваних шаблонів у тексті. Основна мета – створити швидке і водночас ефективне рішення, яке враховує особливості текстових даних.

Цей алгоритм побудований на основі ідей Хаффман-кодування, але із суттєвими спрощеннями, щоб зменшити обчислювальну складність і підвищити швидкість обробки.

Принцип роботи алгоритму

1. Збір статистики символів:
    - Алгоритм проходить текстовий файл і обчислює частоту кожного символу. Це дозволяє зрозуміти, які символи зустрічаються найчастіше.
2. Створення бітового словника:
    - На основі частоти символів створюється бітовий код для кожного символу. Часті символи отримують короткі коди, рідкісні – довші.

- Для цього використовується структура даних "куча", яка ефективно дозволяє будувати кодову таблицю.

3. Кодування тексту:
   - Кожен символ тексту замінюється відповідним бітовим кодом, створеним на попередньому етапі.
4. Перетворення бітового коду на байти:
   - Бітовий код перетворюється на байти для подальшого запису у файл. Це робиться блоками по 8 бітів.
5. Перевірка ефективності:
   - Перед записом стислого файлу алгоритм порівнює розмір стисненого файлу з оригіналом. Якщо стиснення неефективне (розмір не зменшується), алгоритм повертає оригінальний файл.
6. Запис у файл:
   - У стислому файлі зберігається кодова таблиця (для декодування) та закодовані дані.

Нижче в Лістинг 3.5 скорочено представлено фрагмент коду створення власного алгоритму

Лістинг 3.5

```
def compress_with_custom_algorithm(filepath, user_folder=None):
output_path = get_compressed_file_path(filepath, user_folder,
extension=".custom")
with open(filepath, "r", encoding="utf-8") as f_in:
data = f_in.read()
frequency = {}
for char in data:
frequency[char] = frequency.get(char, 0) + 1
from heapq import heappush, heappop, heapify
heap = [[freq, [char, ""]] for char, freq in frequency.items()]
heapify(heap)
while len(heap) > 1:
lo = heappop(heap)
```

```
hi = heappop(heap)
for pair in lo[1:]:
pair[1] = "0" + pair[1]
for pair in hi[1:]:
pair[1] = "1" + pair[1]
heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
huffman_code = {char: code for char, code in heappop(heap)[1:]}
encoded_data = "".join(huffman_code[char] for char in data)
byte_array = bytearray()
for i in range(0, len(encoded_data), 8):
byte_chunk = encoded_data[i:i+8]
byte_array.append(int(byte_chunk.ljust(8, "0"), 2))
original_size = len(data.encode("utf-8"))
compressed_size = len(byte_array) + len(huffman_code) * 4
if compressed_size >= original_size:
return filepath
with open(output_path, "wb") as f_out:
f_out.write(len(huffman_code).to_bytes(4, "big"))
for char, code in huffman_code.items():
f_out.write(len(char.encode("utf-8")).to_bytes(1, "big"))
f_out.write(char.encode("utf-8"))
f_out.write(len(code).to_bytes(1, "big"))
f_out.write(int(code, 2).to_bytes((len(code) + 7) // 8, "big"))
f_out.write(byte_array)
return output_path
```

Особливості алгоритму

- Простота реалізації: Використання базових структур даних, таких як словники та черги з пріоритетами, робить алгоритм легким для реалізації.
- Ефективність: Завдяки побудові бітового словника алгоритм забезпечує стиснення текстів із високою частотою повторюваних символів.
- Перевірка доцільності стиснення: У випадках, коли текст не містить повторів або стиснення збільшує розмір файлу, алгоритм автоматично повертає оригінальний текст.

Переваги

1. Швидкість обробки:
    - Використання алгоритму з лінійною складністю O(n)O(n)O(n) для аналізу символів та побудови словника.
    - Відсутність багатопрохідної обробки, що робить алгоритм швидшим за класичні підходи (наприклад, Хаффман).
2. Гнучкість:
    - Підходить для текстових файлів будь-якого розміру.
    - Простий у використанні в реальних застосунках, таких як журнали або великі тексти.
3. Інтеграція з системою:
    - Алгоритм легко інтегрується у будь-які програмні рішення, особливо для додатків, орієнтованих на стиснення текстових файлів.

Обмеження

1. Ефективність на текстах із низькою частотою повторів:
    - Якщо текст не має повторюваних символів, коефіцієнт стиснення може бути низьким, а в окремих випадках – навіть збільшити розмір.
2. Обмежена універсальність:
    - Алгоритм не підходить для стиснення мультимедійних чи бінарних файлів, оскільки фокусується лише на текстах.

Нижче представлю результат стистення текстового файлу за допомого власного алгоритму

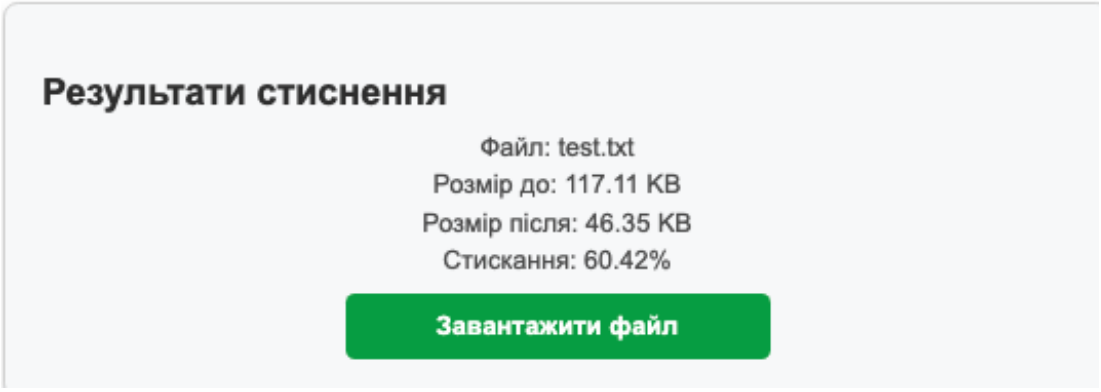

Рис. 3.1. Результати стиснення файлу власним алгоритмом

Також нижче продемонструю Рис. 3.2 блок-схеми роботи власного алгоритму

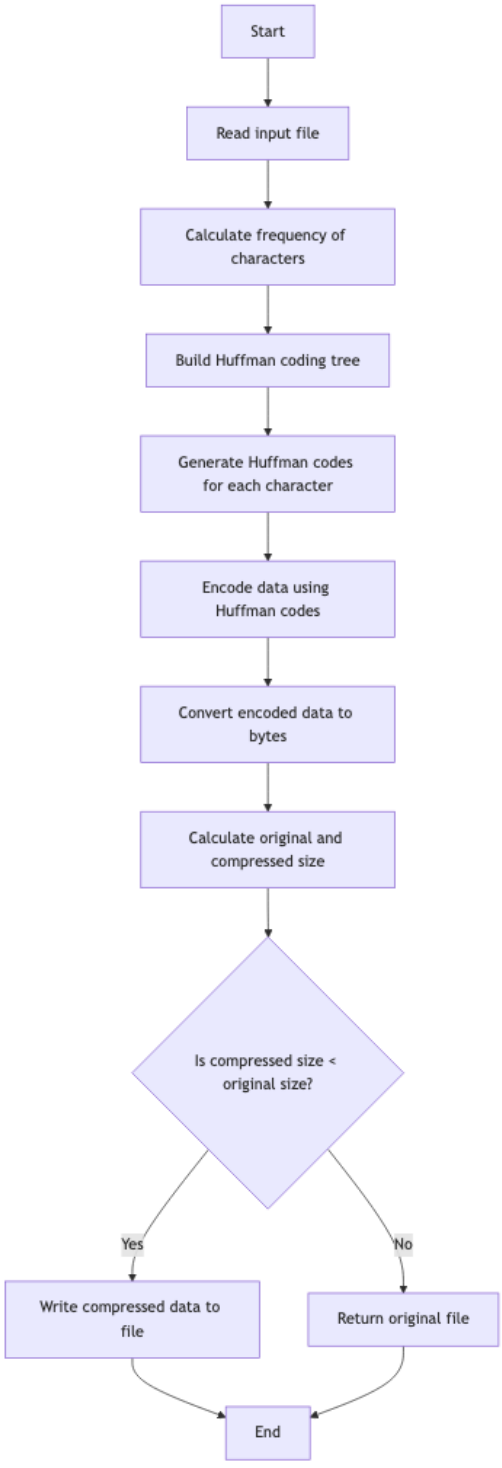

Рис. 3.2. Блок-схеми роботи власного алгоритму стиснення

Розроблений алгоритм забезпечує баланс між швидкістю, простотою та ефективністю. Він особливо підходить для роботи з текстовими файлами, де висока

частота повторюваних символів. У майбутньому його можна розширити або модифікувати для інших типів даних, зберігаючи основну концепцію простоти та ефективності.

3.2 Розробка підходу (програмного рішення) до стиснення даних

3.2.1. Автоматичний вибір алгоритму

Ми створили такий підхід, при якому сама програма вирішує, який алгоритм стиснення найкраще підходить для конкретного файлу. Коли користувач завантажує файл, програма дивиться на його формат і вміст, а потім сама обирає оптимальний метод:

- Текст: Залежно від розміру та вимог до швидкості чи ефективності, застосовуємо Gzip, Deflate чи Huffman.
- Зображення (jpg, png, webp): Використовуємо JPEG або WebP — вони добре “розуміють” ці формати та гарантують пристойну якість.
- Відео: Стискаємо через FFmpeg і сучасний кодек libx265.
- Аудіо: Застосовуємо AAC з автоматичним підбором бітрейту.
- Архіви: Обираємо ZIP чи Deflate.

Нижче представлено Лістинг 3.6, де реалізовано автоматичний вибір алгоритму

Лістинг 3.6

```
if auto_detect:
if file_extension in {"txt", "csv", "log"}:
algorithm = "gzip"
elif file_extension in {"jpg", "jpeg", "png", "webp"}:
algorithm = "jpeg"
elif file_extension in {"mp4", "mkv", "avi"}:
algorithm = "ffmpeg"
elif file_extension in {"mp3", "wav", "aac"}:
algorithm = "aac"

elif file_extension in {"zip", "rar"}:
```

```
algorithm = "zip"
else:
     raise ValueError(
     "Невідомий формат файлу для автоматичного стиснення."
     )
```

Також нижче на Рисунку 3.3 демонструю результат стиснення, вибір визначення автоматичного алгоритму, а також якість стиснення

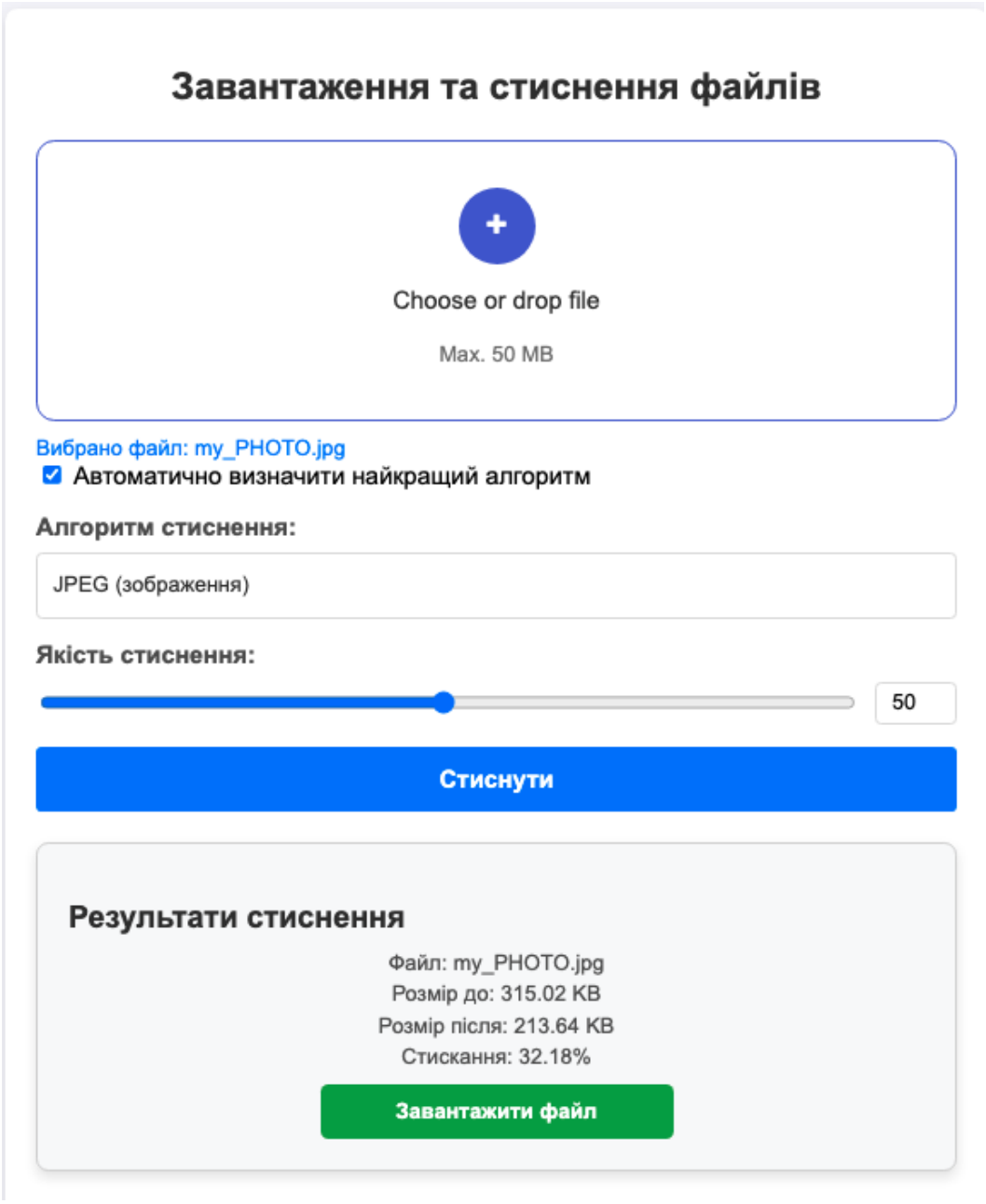

Рис. 3.3. Форма для стиснення файлу з результатом

Таким чином, користувачеві не треба вникати у технічні деталі, додаток сам дасть найкращий варіант.

3.2.2. Уникнення повторної обробки

Щоб не марнувати ресурси на повторне стиснення тих самих файлів, ми використовуємо перевірку за хешем. Якщо файл із такими ж параметрами обробляли раніше:

- Програма просто дає користувачеві посилання на уже стиснену версію.
- Немає потреби знову виконувати стиснення, отже економимо час і ресурси.

Для цього я використовую базу даних (через SQLAlchemy) і перевіряю записані там дані перед кожною новою спробою стиснення.

Нижче представлено Лістинг 3.7, де реалізовано автоматичний вибір алгоритму

Лістинг 3.7

```
file_hash = generate_file_hash(file)
existing_file = File.query.filter_by(
hash=file_hash,
algorithm=algorithm,
user_id=current_user.id if current_user.is_authenticated else
None,
).first()

if existing_file:
return jsonify(
{
"original_file": existing_file.file_name,
"compressed_file": existing_file.compressed_path,
"algorithm": existing_file.algorithm,
"quality": existing_file.quality,
"original_size": existing_file.original_size,
"compressed_size": existing_file.compressed_size,
"compression_ratio": existing_file.compression_rate,
}
```

)

Я записую хеш змісту файлу в БД, а потім роблю запит в БД через SQLAlchemy, де фільтрую хеш отриманого файлу із існуючим в БД, а також по алгоритму, бо якщо юзер хоче стиснути файл іншим алгоритмом, то це вже новий запис в БД.

Гнучкий вибір алгоритму вручну

Хоча вибір алгоритму автоматизовано, я надав користувачеві можливість обрати метод стиснення самостійно. Це корисно у випадках, коли потрібно поекспериментувати з різними алгоритмами або підібрати оптимальний варіант для конкретної ситуації. На сторінці завантаження файлу передбачене випадаюче меню з усіма доступними варіантами.

Інформація про результати

Після стиснення користувач бачить усю важливу статистику:

- Початковий і кінцевий розмір файлу.
- Коефіцієнт, який показує, наскільки успішним було стискання.
- Алгоритм, що використовувався.

Якщо користувач авторизований, додаток зберігає історію всіх операцій. Це зручно: можна швидко знайти потрібний стиснений файл без повторної обробки.

Додавання власного алгоритму

Окрім добре відомих алгоритмів, я інтегрував власний метод, розроблений спеціально для певних типів текстових файлів, де важливі простота й швидкість. Цей алгоритм забезпечує непоганий баланс між ефективністю та продуктивністю, особливо для невеликих текстових файлів із повторюваними символами.

Робота з великими файлами

Я подбав і про ефективну роботу з великими файлами. Для цього:

- Реалізовано потокову обробку, що дозволяє стискати файли частинами, без завантаження їх повністю в оперативну пам’ять.
- Використано паралельні обчислення для прискорення роботи із мультимедійними даними.

Це означає надійну роботу програми навіть із файлами розміром у кілька гігабайтів.

Таким чином, мій підхід до стиснення даних забезпечує гнучкість, автоматизацію та зручність для користувачів. Він враховує специфіку різних типів даних і надає як автоматичний, так і ручний вибір алгоритмів. Уникнення дублювання обробки, можливість інтеграції власних алгоритмів та оптимізація для великих файлів роблять це рішення як ефективним, так і універсальним.

3.2.3 Архітектура проєкту

Архітектуру проєкту представлю нижче, з її всією складовою, та описом всіх деталей

Сторінка авторизації:

Тут користувачі можуть зайти в систему за допомогою свого логіна й пароля. Для новачків передбачена реєстрація – після введення імені користувача та електронної пошти перевіряється, чи немає вже такого облікового запису. Якщо все добре – реєстрація успішна.

Можливості для авторизованих користувачів:

- Коли користувач увійшов у свій обліковий запис, він може завантажувати файли, щоб їх стиснути.
- У такому разі у нього є своя історія обробки – тобто можна переглянути, які файли раніше стискалися, який був алгоритм стиснення, скільки часу це зайняло, та наскільки зменшився розмір файлу.
- Стиснені файли зберігаються на сервері годину, тож якщо користувачеві потрібно повторно завантажити результат, він може це зробити протягом цього часу.

Можливості для неавторизованих користувачів:

- Якщо користувач не увійшов у систему або не має облікового запису, він все одно може скористатися стисненням файлів.
- В такому випадку історія його дій не зберігатиметься.
- Стиснені файли зберігаються лише півгодини. Це зручно, якщо потрібно швидко щось стиснути без реєстрації, але якщо потрібно повернутися до результату пізніше, варто мати на увазі обмежений час.

Обробка файлів:

- Усі завантажені файли спершу потрапляють у тимчасові папки: одна для авторизованих користувачів, інша – для неавторизованих.
- Для кожного файлу вираховується унікальний хеш – це робиться для того, щоб не стискати один і той самий файл двічі. Якщо з таким хешем уже є готовий результат, користувач одразу отримує посилання, без повторного стиснення.

Архітектуру сервісу представлено нижче на Рисунку 3.4

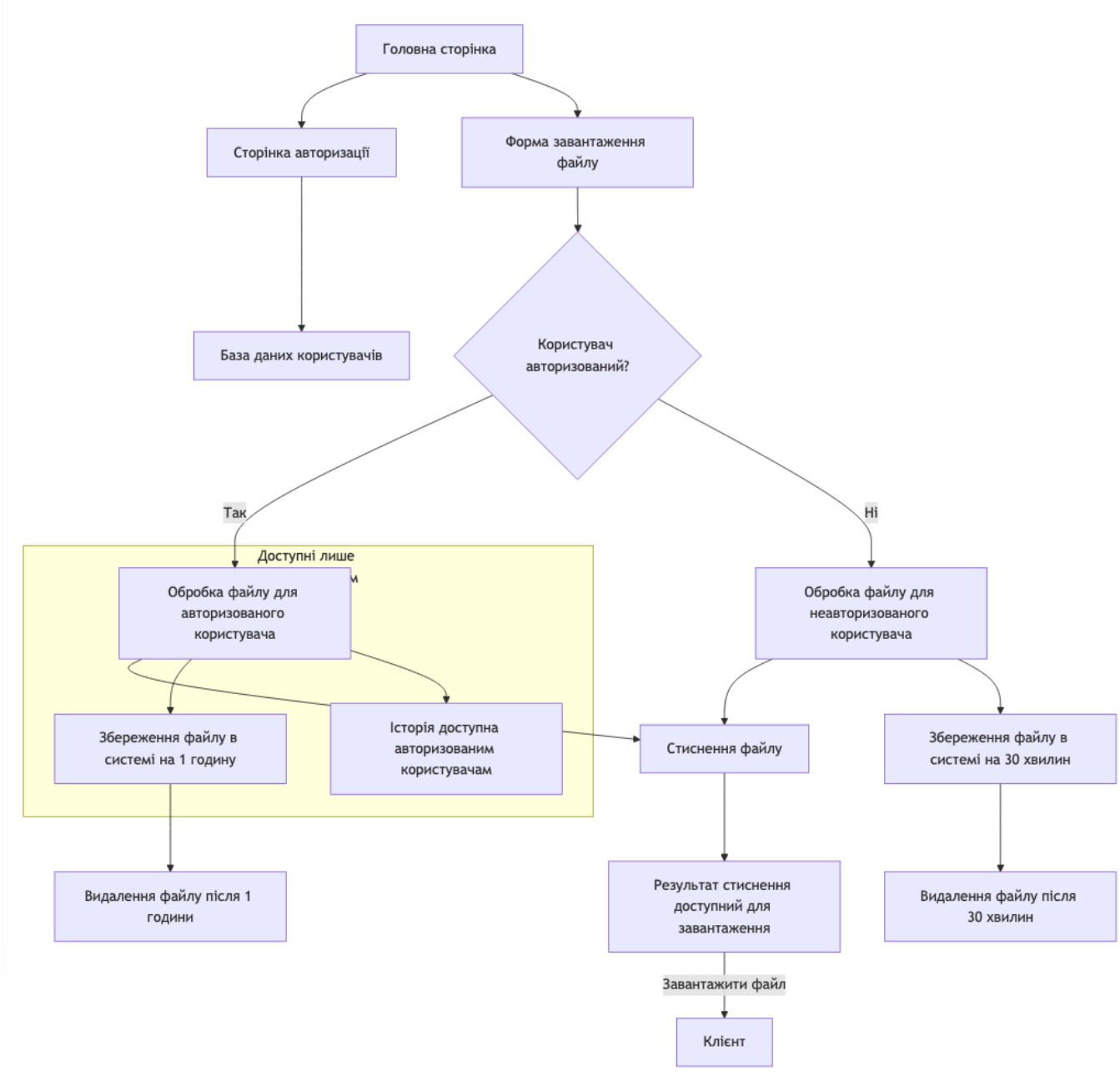

Рис. 3.4. Архітектура сервісу

Процес стиснення:

Сервіс автоматично підбирає алгоритм залежно від типу файлу (текст, зображення, відео, аудіо тощо), але якщо користувач хоче, він може самостійно вибрати бажаний метод. Для цього доступні такі варіанти: Gzip, Deflate, Huffman, FFmpeg (для відео), JPEG, WebP (для зображень), AAC (для аудіо), а також власний алгоритм для певних сценаріїв.

3.2.4 Вибір бекенд-технології

Я обрав Flask як основу бекенду цього сервісу. Чому саме його. Бо це один із найпростіших і найгнучкіших веб-фреймворків у світі Python. Він не змушує використовувати зайві компоненти, а дає змогу взяти лише те, що потрібно, і швидко розгорнути робочий застосунок. Це особливо важливо, коли треба діяти швидко і гнучко, а не витрачати час на вивчення громіздких фреймворків.

Пояснення чому Flask:

Гнучкість:

Flask – це мікрофреймворк. Він не нав’язує свою архітектуру, дозволяючи збирати рішення саме з тих компонентів, які потрібні для конкретного проєкту.

Швидкість розробки:

З мінімумом “зайвих” речей у основі, Flask дозволяє миттєво перейти до написання коду, який вирішує суть завдання. Не треба розбиратися з купою налаштувань чи непотрібними модулями.

Легка інтеграція:

Flask чудово товаришує з багатьма бібліотеками та інструментами – від роботи з базами даних до обробки медіа. Це дозволяє реалізувати логіку стиснення файлів, аналіз даних, роботу з користувачами та інші функції без “танців з бубном”.

Підтримка й документація:

Велика спільнота означає, що на більшість питань уже є відповіді, приклади або підказки. Це знижує ризик застрягти з якоюсь проблемою.

Використані бібліотеки

Щоб сервіс був зручним, універсальним і функціональним, я підключив кілька корисних бібліотек:

1. Flask. Основний двигун, на якому “крутиться” весь застосунок: маршрути, обробка запитів, логіка REST API.
2. Flask-Cors. Дозволяє сервісу коректно відповідати на запити з інших доменів. Це важливо, коли фронтенд і бекенд розміщені в різних місцях.
3. Pillow. Дозволяє змінювати розмір, стискати та обробляти зображення в різних форматах, як-от JPEG, PNG та WebP.
4. ffmpeg-python. Обгортка над FFmpeg для Python, що спрощує роботу з відео та аудіо. З її допомогою можна стискати відео за допомогою libx265, а аудіо – за допомогою AAC.
5. Flask-SQLAlchemy. Надає зручний спосіб працювати з базою даних через ORM. Набагато легше писати код, ніж кілометри SQL-запитів.
6. Flask-Migrate. Допомагає легко змінювати структуру бази даних: додавати таблиці, поля й інші елементи без болісних “ручних” операцій.
7. Flask-Login. Відповідає за входи-виходи користувачів. Реєстрація, авторизація, сесії – усе це стає простим завданням.
8. Psutil. Допомагає стежити за тим, скільки ресурсів “з’їдає” застосунок. Якщо треба моніторити використання пам’яті чи CPU – це те, що треба.
9. Matplotlib. Використовується для побудови графіків і візуалізацій результатів роботи алгоритмів стиснення. Замість сухих чисел – наочні діаграми.
10. Pytest – це потужна та популярна бібліотека для тестування в Python, яка значно спрощує процес написання, організації та виконання тестів. Вона ідеально підходить для створення як простих модульних тестів, так і складних інтеграційних тестів.

Обравши Flask, я отримав гнучку та зрозумілу основу для бекенду. Завдяки додатковим бібліотекам сервіс набув усіх потрібних рис: від стиснення зображень і відео до зручної роботи з базою, керування користувачами та створення графіків. Це дозволяє швидко реагувати на нові вимоги, ефективно розвивати продукт і забезпечувати комфортну роботу як для себе, так і для майбутніх користувачів.

3.3 Представлення структури проєкту

Я вирішив побудувати свій додаток так, щоб робота з даними була максимально простою, надійною та масштабованою. Для цього я використав реляційну базу даних: вона гарантує цілісність даних, дає змогу легко розділяти дані різних типів користувачів (авторизованих і неавторизованих) та керувати файлами відповідно до їхніх характеристик.

3.3.1 Модель бази даних

У моїй моделі є три основні таблиці: User, File і TemporaryFile. Кожна з них має свою роль у зберіганні інформації.

Таблиця User

Ця таблиця містить інформацію про всіх зареєстрованих користувачів сервісу. Кожен запис тут відповідає конкретній людині, яка має свій обліковий запис. Завдяки цій таблиці я знаю, хто увійшов у систему, з якою поштою та ім’ям користувача, і можу надати їм персоналізований досвід. Дані про пароль зберігаються безпечно (у вигляді хешу), щоб ніхто не міг дізнатися реальний пароль користувача.

Таблиця 3.1

Модель таблиці даних “User”

Назва поля	Тип даних	Опис
id	Integer (Primary Key)	Унікальний ідентифікатор користувача, використовується як первинний ключ.
username	String(64)	Унікальне ім'я користувача, яке він використовує для входу.
email	String(120)	Унікальна адреса електронної пошти користувача.
password	String(128)	Захешований пароль користувача для забезпечення безпеки.

Таблиця File

Ця таблиця призначена для файлів, які стиснули зареєстровані користувачі.

Таблиця 3.2

Модель таблиці даних “File”

Назва поля	Тип даних	Опис
id	Integer (Primary Key)	Унікальний ідентифікатор файлу, використовується як первинний ключ.
file_name	String(128)	Назва завантаженого файлу.
algorithm	String(128)	Назва алгоритму стиснення, використаного для обробки файлу
quality	String(64)	Якість стиснення файлу (для алгоритмів, які підтримують регулювання якості).
hash	String(64)	Унікальний хеш, створений для файлу, щоб уникнути дублювання обробки.
original_size	Float	Розмір файлу до стиснення (в кілобайтах).
compressed_path	String(128)	Шлях до стисненого файлу, що зберігається на сервері.
compressed_size	Float	Розмір стисненого файлу (в кілобайтах).
compression_rate	Float	Коефіцієнт стиснення у відсотках.
timestamp	DateTime	Дата й час створення запису.
status	Boolean	Статус обробки файлу (успішно/неуспішно).
error	String(256)	Повідомлення про помилку у випадку невдалої обробки.
user_id	Integer	Посилання на користувача (з таблиці User), який завантажив файл.

Вона зберігає інформацію про те, який файл було завантажено, коли і як він був стиснутий, який алгоритм використано та наскільки вдалося зменшити розмір. Також тут вказано, якому користувачеві належить цей файл (через user_id). Завдяки цьому користувач може переглянути свою історію стиснення файлів у будь-який момент.

Таблиця TemporaryFile

Ця таблиця створена для тих випадків, коли користувач не авторизований, але хоче скористатися стисненням файлу.

Таблиця 3.3

Модель таблиці даних “TemporaryFile”

Назва поля	Тип даних	Опис
id	Integer (Primary Key)	Унікальний ідентифікатор файлу, використовується як первинний ключ.
original_name	String(128)	Назва завантаженого файлу.
algorithm	String(128)	Назва алгоритму стиснення, використаного для обробки файлу
quality	String(64)	Якість стиснення файлу (для алгоритмів, які підтримують регулювання якості).
hash	String(64)	Унікальний хеш, створений для файлу, щоб уникнути дублювання обробки.
original_size	Float	Розмір файлу до стиснення (в кілобайтах).
compressed_path	String(128)	Шлях до стисненого файлу, що зберігається на сервері.
compressed_size	Float	Розмір стисненого файлу (в кілобайтах).
compression_rate	Float	Коефіцієнт стиснення у відсотках.
created_at	DateTime	Дата й час створення запису про тимчасовий файл.

Оскільки він не має облікового запису, ми не зберігаємо його історію надовго.

Ці файли зберігаються тимчасово, наприклад, протягом півгодини. Таким чином, якщо людині потрібно швидко стиснути файл "на ходу", вона може це зробити без реєстрації, проте не матиме довготермінового доступу до історії своїх стиснень.

Переваги моделі бази даних

1. Гнучкість: Чітке розмежування даних для авторизованих та неавторизованих користувачів.
2. Масштабованість: Логічна структура забезпечує легке додавання нових функціоналів.
3. Безпека: Хешування паролів та використання унікальних хешів файлів підвищує захист даних.
4. Оптимізація: Повторне використання результатів стискання через хешування файлів скорочує час та обсяг обробки.

Узагальнення

Розроблена модель бази даних підтримує всі основні функції додатку, забезпечуючи зберігання даних користувачів, історії файлів та тимчасової інформації для неавторизованих сесій. Така структура дозволяє легко інтегрувати нові функції та масштабувати систему за необхідності.

3.3.2 Діаграми

Діаграми допомагають наочно показати структуру та взаємозв’язки між компонентами системи, логіку обробки даних і послідовність дій при виконанні операцій. Вони спрощують розуміння складних процесів і полегшують спільну роботу розробників, тестувальників та інших учасників проєкту, забезпечуючи єдине унаочнене бачення архітектури та внутрішньої логіки системи.

Наявність діаграм полегшує документування системи. Вони можуть стати частиною технічної документації, до якої розробники звертатимуться в процесі підтримки та розвитку продукту. Така візуалізація спрощує пошук потрібної інформації й сприяє кращому розумінню архітектури навіть після завершення початкової розробки.

В цьому розділі, я розповім та продемонструю рисунки з описом діаграм, які представлять функціонал мого проєкту

Діаграма структура моделей та їхні взаємозв'язки представлена на Рисунку 3.5

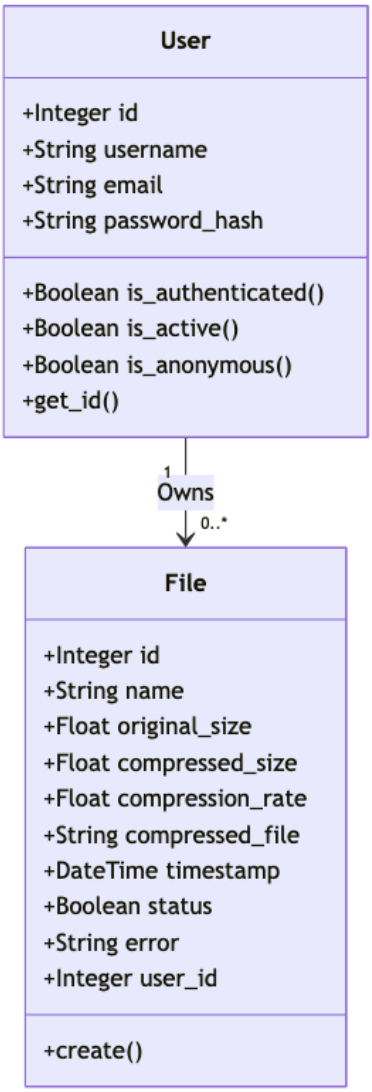

Рис. 3.5. Діаграма структура моделей та їхні взаємозв'язки

На цій діаграмі зображено, як у системі пов’язані користувачі та файли. По суті, кожен зареєстрований користувач (User) може мати кілька файлів (File), з якими він працює. Такий взаємозв’язок «один до багатьох» допомагає чітко зрозуміти, як дані користувача співвідносяться з наборами файлів, а також який набір характеристик (розмір, статус, вибраний метод стискання) належить кожному з них.

Діаграма процесу завантаження і подальшої обробки файлу представлена на Рисунку 3.6

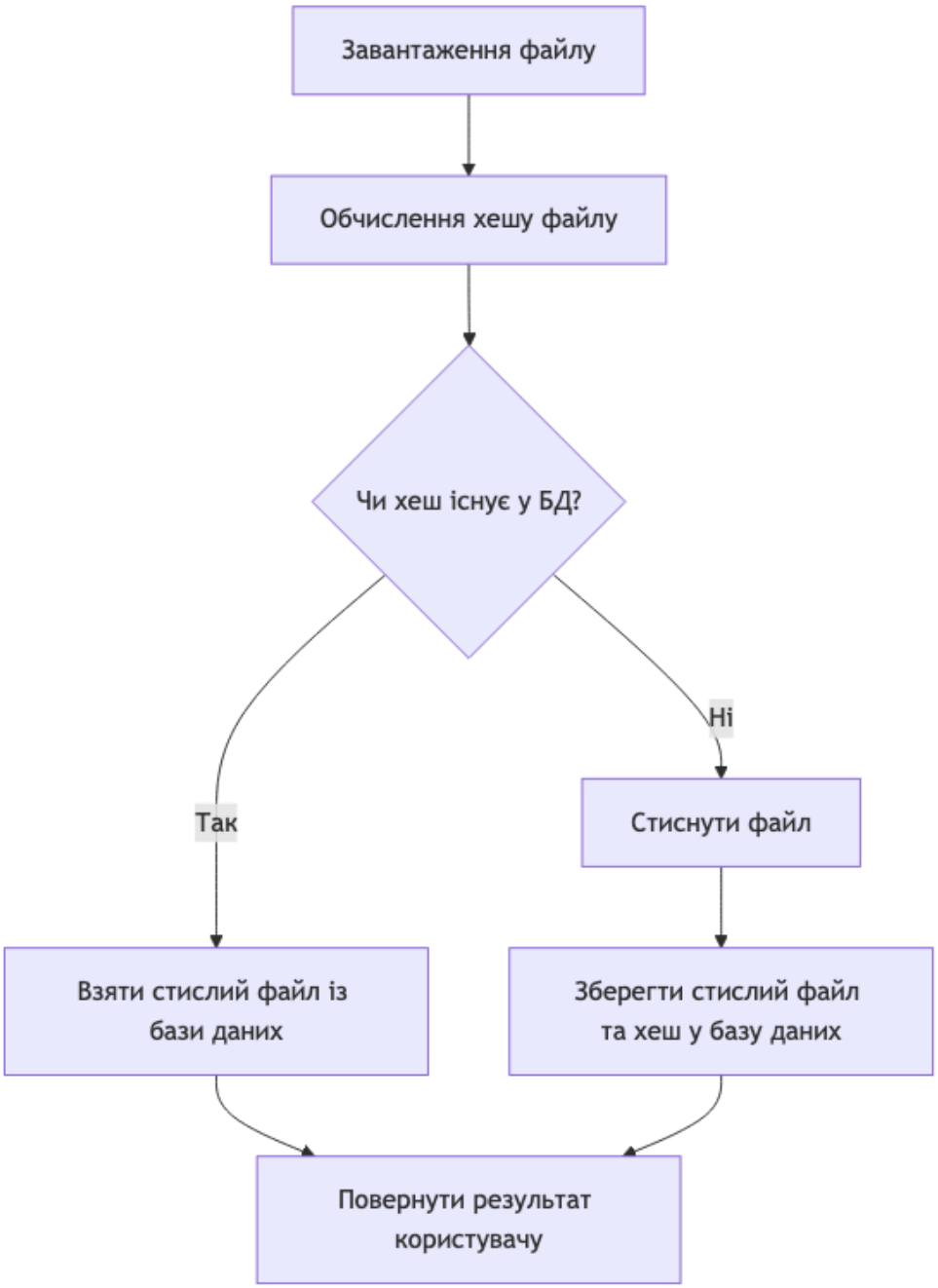

Рис. 3.6. Діаграма процесу завантаження і подальшої обробки файлу

Ця діаграма показує покрокову логіку роботи з файлом: від моменту, коли користувач його завантажує, до фінального результату. Спершу система створює

хеш-функцію файлу, щоб перевірити, чи файл уже не оброблявся раніше. Якщо співпадінь немає, файл стискається, зберігається в базі та надається користувачу.

Якщо ж файл «вже знайомий» системі, немає потреби стискати його повторно — користувач відразу отримує готовий результат. Такий підхід економить і час, і ресурси.

Діаграма логіки прибирання застарілих файлів представлена на Рисунку 3.7

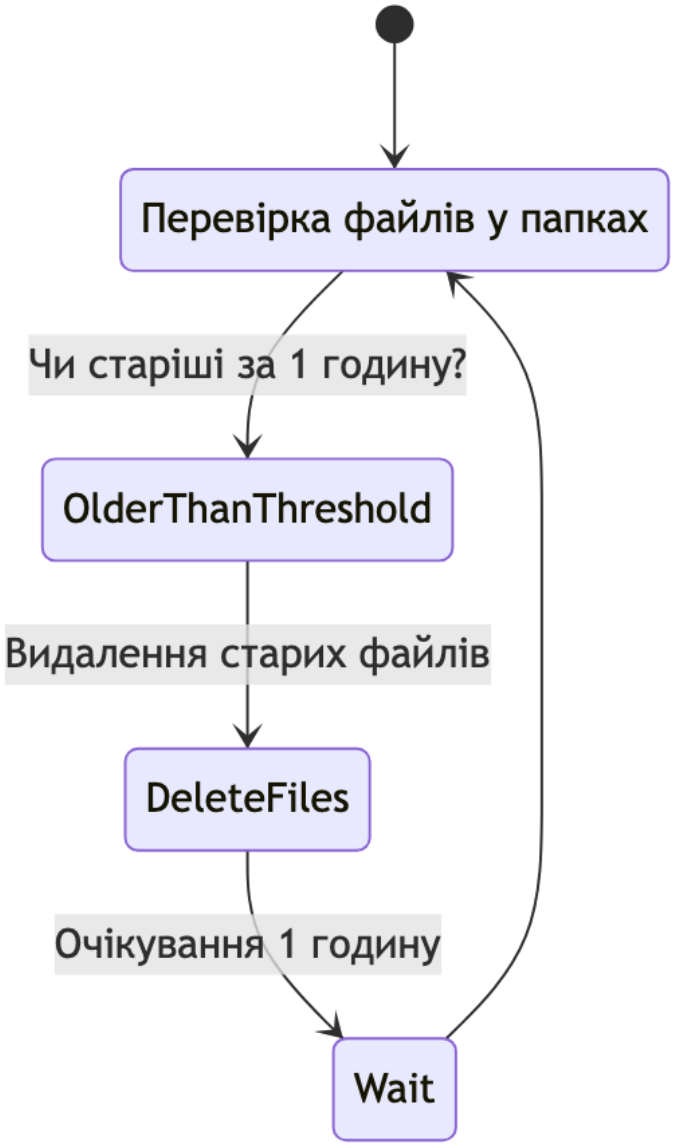

Рис. 3.7. Діаграма логіки прибирання застарілих файлів

Тут можна побачити механізм автоматичного видалення зайвих файлів. Система періодично перевіряє, чи є у сховищі файли, які ніхто давно не відкривав.

Для неавторизованих користувачів термін зберігання файлу менший (скажімо, 30 хвилин), для авторизованих — довший (наприклад, 1 година). Після спливу цього часу файл видаляється, щоб не засмічувати базу і звільнити місце.

Діаграма авторизації та доступу до історії представлена на Рисунку 3.8

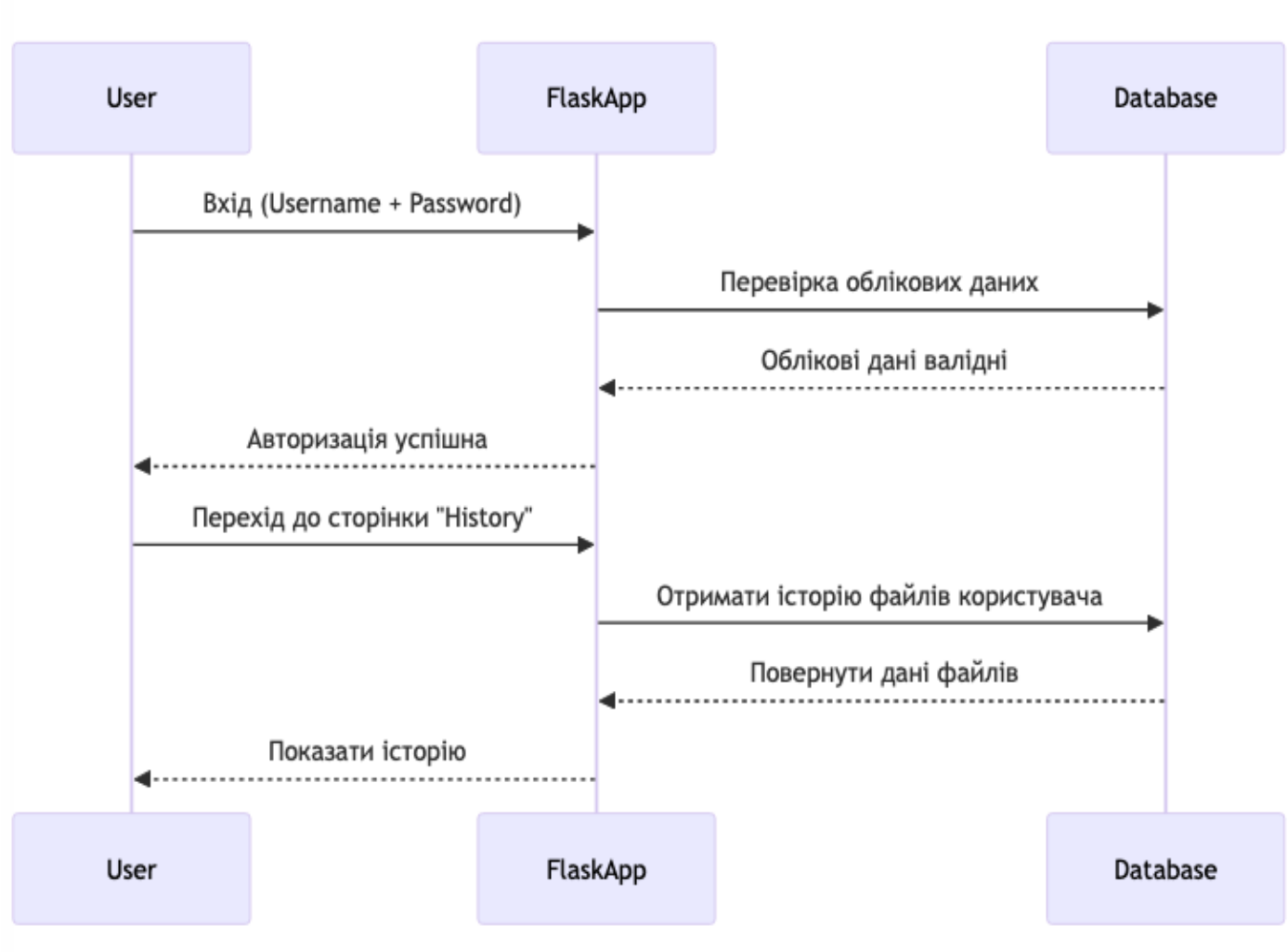

Рис. 3.8. Авторизації та доступу до історії

Діаграма ілюструє процес авторизації користувача та доступу до сторінки історії, детально пояснюючи кожен етап взаємодії між користувачем, сервером (FlaskApp) та базою даних (Database). Ця діаграма демонструє, як користувач проходить через ключові кроки, необхідні для отримання доступу до історії своїх файлів, і які саме компоненти системи задіяні на кожному етапі.

Ініціалізація взаємодії

Користувач відкриває сторінку авторизації додатку та вводить свої облікові дані — ім’я користувача (username) та пароль (password). Ці дані надсилаються серверу для перевірки.

Обробка запиту авторизації

Сервер (FlaskApp) отримує дані користувача та починає процес валідації. Валідація включає перевірку наявності всіх необхідних полів, коректності їх формату та можливості подальшої обробки.

Запит до бази даних

FlaskApp звертається до бази даних (Database), запитуючи інформацію про користувача. В цьому етапі виконується пошук у таблиці User, де порівнюються введені облікові дані з уже збереженими в базі. Зокрема, перевіряється, чи існує користувач із таким ім’ям та чи збігається хеш паролю.

Перевірка облікових даних

База даних повертає результат перевірки. Якщо облікові дані валідні (ім’я користувача знайдено, пароль збігається), сервер отримує підтвердження успішної авторизації. В іншому випадку надсилається повідомлення про помилку, яке відображається користувачу (наприклад, "Невірний логін або пароль").

Успішна авторизація

Якщо облікові дані успішно перевірені, сервер встановлює сеанс для користувача та повідомляє про успішну авторизацію. Користувач отримує доступ до приватних функцій додатку, таких як перегляд історії оброблених файлів.

Доступ до сторінки історії

Користувач переходить на сторінку "History". Ця сторінка містить інформацію про всі файли, які були оброблені цим користувачем. Сервер обробляє запит, звертаючись до бази даних для отримання історії.

Запит історії файлів

FlaskApp надсилає запит до таблиці File, фільтруючи дані за ідентифікатором користувача (user_id). Вибираються всі файли, які оброблялися даним користувачем, разом із детальною інформацією, такою як назва файлу, початковий та стиснутий розміри, алгоритм стиснення, та час обробки.

Повернення даних історії

База даних повертає результати запиту серверу, передаючи список файлів із зазначеними характеристиками.

Відображення історії користувачу

Сервер формує сторінку з отриманими даними та відправляє її користувачу. Користувач бачить список своїх файлів у вигляді таблиці чи іншого зручного формату, включаючи інформацію про час стиснення, використані алгоритми та коефіцієнт стискання.

3.4 Опис функціоналу пропонованого програмного рішення

У цьому розділі я докладніше розкажу, як влаштований мій інструмент для стиснення файлів та які можливості він пропонує. Я намагався зробити його не лише технологічно "потужним", а й простим у використанні. Ідея така: користувачі мають швидко розібратися, як усе працює, не витрачаючи час на складні інструкції.

Мені хотілося створити сервіс, де зібрані "під одним дахом" і зручний інтерфейс, і ефективні алгоритми стиснення. При цьому я не забув про різні типи користувачів: хтось просто зайде один раз і без авторизації стисне невеличкий файлик, а хтось регулярно працюватиме, матиме свій акаунт та історію обробок. До того ж, реалізовано автоматичне очищення старих файлів - так система не "захаращуватиметься" непотрібними даними.

Сторінка авторизації

Ця сторінка — перший крок у роботі з додатком. Користувач вводить свою електронну адресу та пароль у відповідні поля. Якщо ви тут уперше, є кнопка "Реєстрація", що відправить вас на сторінку створення нового облікового запису. Якщо дані введені некоректно чи користувач не зареєстрований, під формою з’явиться повідомлення про помилку.

Інтерфейс сторінки максимально простий: поля для вводу даних, кнопки "Вхід" та "Забув пароль". Це дозволяє користувачу швидко авторизуватися або відновити свої дані, не блукаючи по додатку. Нижче на рисунку 3.9 представлена сторінка входу

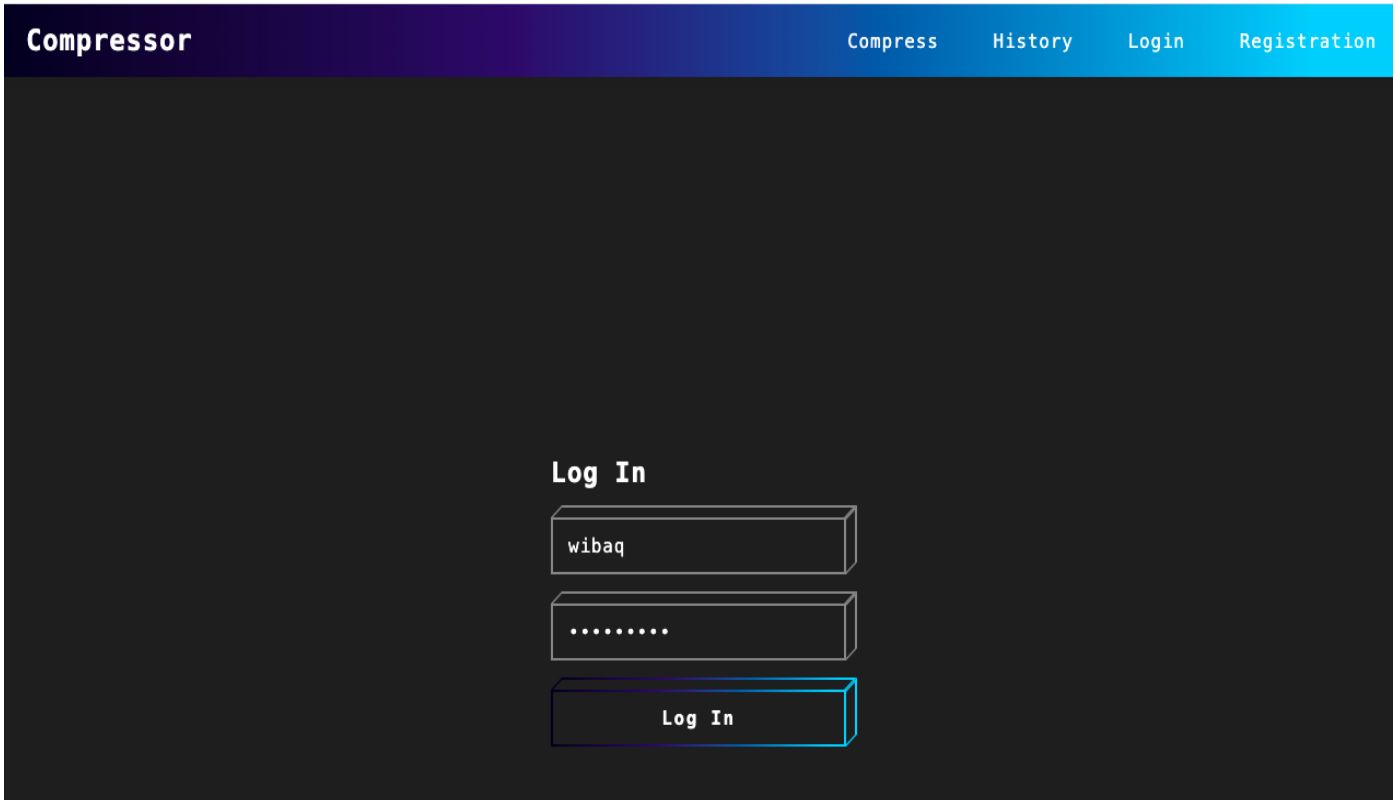

Рис. 3.9. Сторінку авторизації

Головна сторінка (Завантаження файлу)

Це центральна сторінка додатку, де користувач завантажує файл для стиснення. Головний елемент — форма для завантаження, куди можна просто перетягнути файл (drag-and-drop) або обрати його традиційним способом. Після додавання файлу можна самостійно обрати алгоритм стиснення (наприклад, Gzip, Huffman, Deflate) або ввімкнути автоматичний режим, який сам вибере оптимальний варіант.

Користувач також може відрегулювати якість стиснення повзунком, віддаючи перевагу або максимально компактному файлу, або збереженню більшої якості. На сторінці є коротка інструкція:

- Завантажте або перетягніть файл.
- Оберіть алгоритм або включіть "Автоматичне визначення".
- Натисніть "Стиснути" і дочекайтеся завершення.
- Після обробки результат одразу ж з’явиться тут же, на головній сторінці.

Сторінка максимально доступна та просто, щоб юзер не розгубився, адже для функціоналу стисканні не потрібно декілька сторінок, а вистачає тільки однієї.

Нижче буде преставлено головну сторінку на Рисунку 3.10

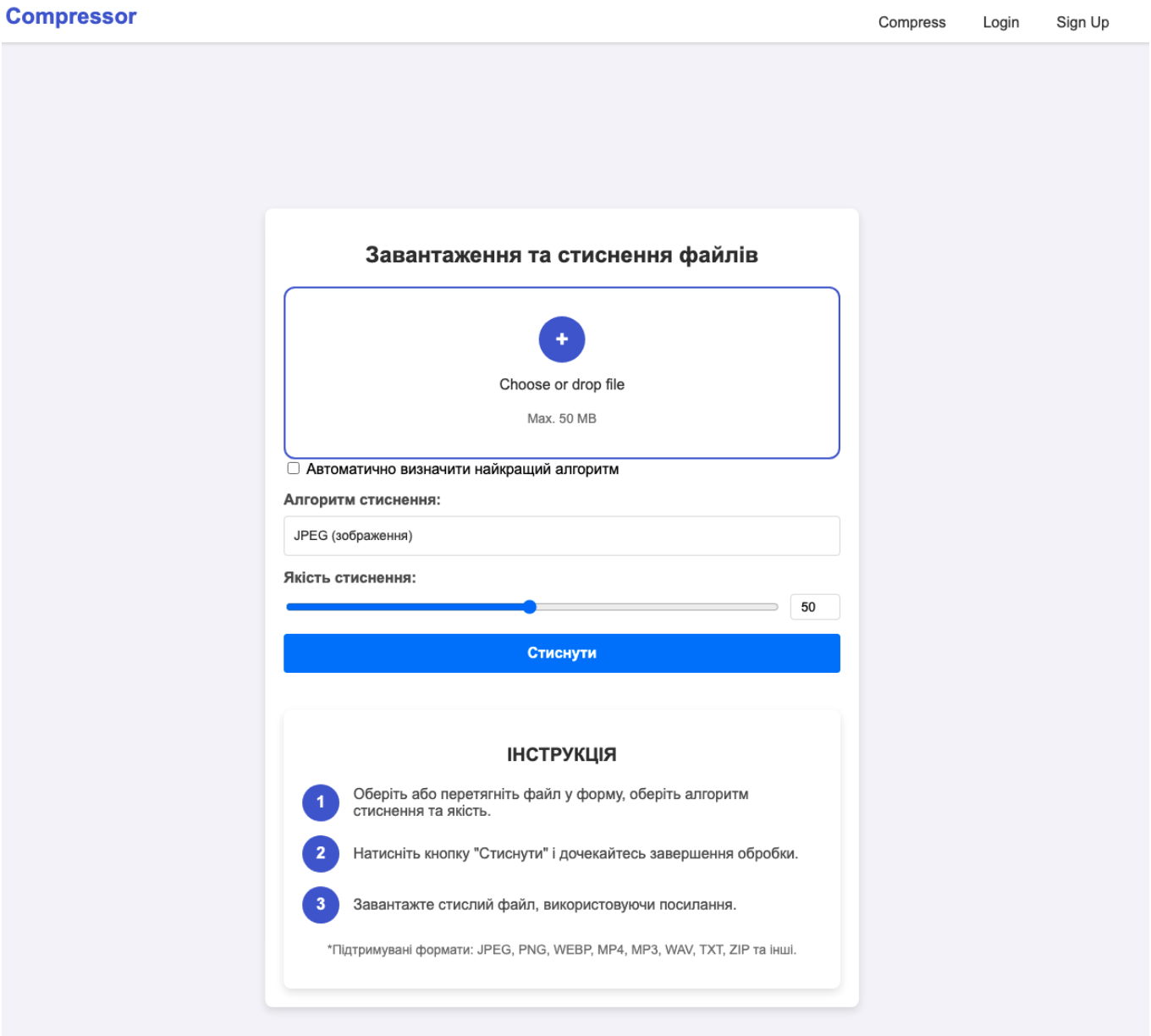

Рис. 3.10. Головна сторінка

Сторінка результатів стиснення

Результати з’являються на головній сторінці автоматично, щойно завершиться процес стиснення. У спеціальному блоці ви побачите:

- Початковий розмір файлу (у кілобайтах).
- Розмір стиснутого файлу (у кілобайтах).
- Який алгоритм використано.

- Час, витрачений на стиснення.
- Коефіцієнт стиснення (у відсотках).
- Посилання для завантаження готового стиснутого варіанту.

Нижче буде представлено головну сторінку з результатом стискання на Рисунку 3.11

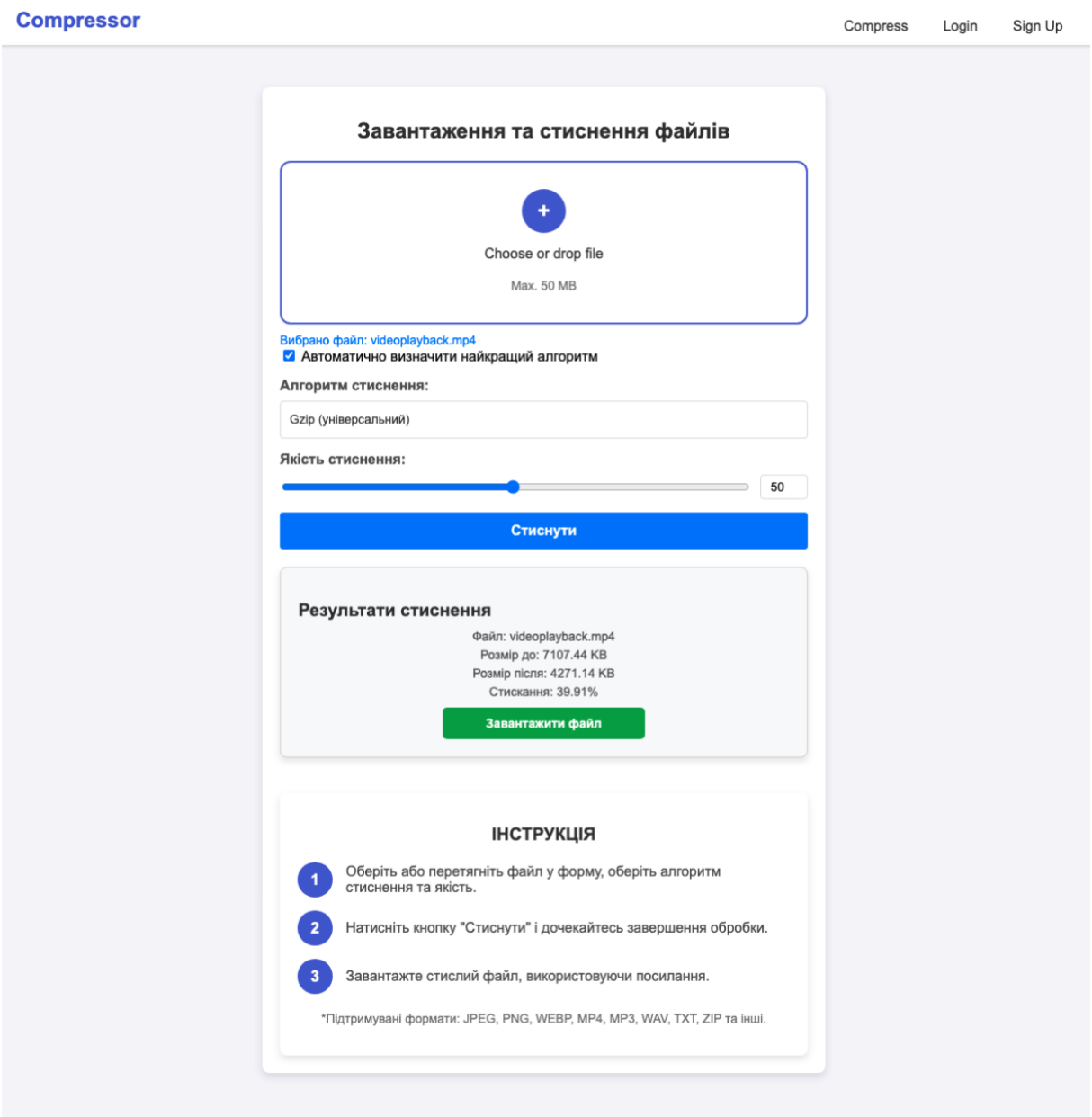

Рис. 3.11. Головна сторінка із виведенням результату стискання

Усе подано у вигляді зрозумілої таблиці. Якщо під час стиснення виникла помилка, система пояснить причину збою.

Сторінка історії (для авторизованих користувачів)

Авторизовані користувачі мають доступ до списку всіх файлів, які вони коли-небудь завантажували і стискали. У кожному записі вказано:

- Назву файлу.
- Використаний алгоритм.
- Дату й час створення обробленого файлу.
- Початковий та кінцевий розміри.
- Коефіцієнт стиснення.

Нижче буде представлено сторінку з історією стискань на Рисунку 3.12

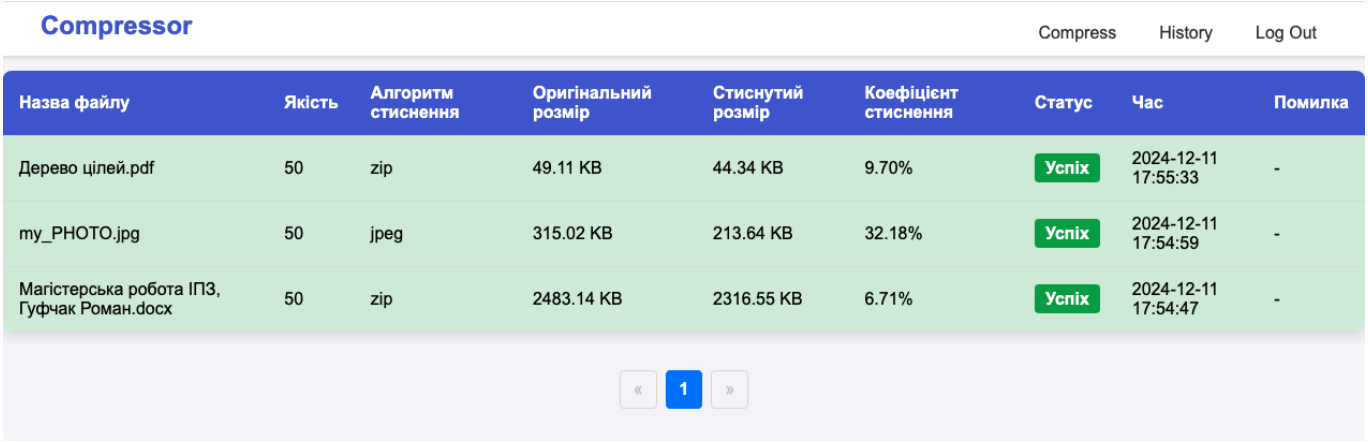

Рис. 3.12. Сторінка історії для авторизованих користувачів

Цю історію можна сортувати за різними критеріями (наприклад, за назвою чи датою), а також шукати потрібний запис за ім’ям файлу або алгоритмом.

Автоматичне очищення файлів

У нашому сервісі передбачено автоматичне видалення завантажених файлів через певний час. Це зроблено для того, щоб не перевантажувати сервер зайвими даними та забезпечити безпеку користувачів.

Демонстрація опису працездатності

- Якщо ви увійшли у свій обліковий запис, файли зберігаються на сервері протягом однієї години. За цей час ви можете спокійно завантажити чи переглянути результат.

- Якщо ж ви не авторизовані (наприклад, зайшли на сайт “на швидку руч”), файли будуть доступні лише протягом 30 хвилин.
- Після закінчення цих термінів наш фоновий процес (спеціальна програма, що працює “за лаштунками”) автоматично видаляє старі файли. Завдяки цьому у сховищі не накопичується “мотлох”, а ваші персональні дані не лежать десь без потреби. Це, своєю чергою, допомагає нам підтримувати порядок і безпеку, а користувачам — бути впевненими, що їхні файли не залишаються в системі надовго.

3.5 Тестування функціональності

Тестування є невід'ємною частиною розробки програмного забезпечення, особливо для систем, що працюють з важливими даними та алгоритмами. У цьому розділі я опишу, як було здійснено тестування розробленого додатку для стиснення файлів, використовуючи автоматизовані тести та ручне тестування веб-клієнта.

3.5.1 Автоматизоване тестування

Автоматизоване тестування за допомогою pytest допомогло впевнитися, що всі ключові функції нашого додатку працюють стабільно й надійно. Ми перевірили буквально все: від авторизації користувачів до процесів стиснення, взаємодії з базою даних та автоматичного прибирання «застарілого сміття». Такий підхід дозволив швидко виявляти будь-які недоліки й перевіряти, чи все відповідає тому, що ми запланували.

Наприклад, у тестах на авторизацію ми спеціально вводили як правильні, так і неправильні дані. Це дало змогу переконатися, що при коректних логіні та паролі користувач спокійно заходить до свого облікового запису, а при неправильних отримує чітке повідомлення про помилку. Так само ми протестували реєстрацію: нові користувачі успішно додаються до бази, якщо всі поля заповнені як слід.

Тести підтвердили, що файли стискаються приблизно так, як ми й очікували, а результати фіксуються у базі. Ми також протестували автоматичний вибір алгоритму залежно від типу файлу, і він спрацював як годинник.

Інтеграція з базою даних — ще один важливий момент. Ми прослідкували, щоб кожен файл отримував унікальний хеш, аби не дублювати зайві дані й не перевантажувати систему. Крім того, перевірили, що записи в базі коректно видаляються після того, як вичерпається їхній термін зберігання.

Автоматичне очищення файлів ми протестували, встановлюючи різні інтервали для авторизованих і неавторизованих користувачів.

Загалом, автоматизовані тести з pytest надали нам упевненість, що додаток працює саме так, як має. Вони швидко сигналізують про будь-які помилки, даючи можливість одразу виправити їх і покращити продукт. Завдяки цьому кінцеві користувачі отримують надійний сервіс, а розробники — спокій за якість свого рішення.

3.5.2 Ручне тестування

Ручне тестування веб-клієнта допомогло перевірити весь спектр можливостей додатку, починаючи від авторизації й завантаження файлів, до відображення результатів, перегляду історії та автоматичного очищення старих файлів. Завдяки цьому етапу вдалося переконатися, що користувачі легко зрозуміють, як користуватися сервісом, а всі функції працюють так, як було задумано.

Перевірку починали зі сторінки авторизації. Я навмисне вводив як правильні, так і неправильні логіни та паролі, щоб переконатися, що система реагує адекватно. Зі справжніми даними користувач успішно входив у свій обліковий запис, а з помилковими — отримував повідомлення про невірний логін чи пароль. Також було протестовано реєстрацію нового користувача: вона проходила без проблем, після чого можна було одразу скористатися додатком.

На головній сторінці перевірили завантаження файлів. Функція “перетягни й кинь” (drag-and-drop) спрацювала чудово з різними типами файлів — від текстових документів до великих відео. Вибір алгоритму стиснення, як вручну, так і в

автоматичному режимі, теж не викликав труднощів. Повзунок для регулювання якості давав змогу варіювати ступінь стиснення "на льоту", без жодних збоїв.

Після стиснення додаток показував усю потрібну інформацію: початковий та кінцевий розміри файлу, який алгоритм використовувався, скільки часу зайняла обробка та наскільки вдалося зменшити обсяг у відсотках. Посилання на завантаження стиснутого файлу відпрацьовувало стабільно. У ситуаціях, коли стиснути файл було неможливо, користувач отримував чітке повідомлення про помилку.

Сторінка історії була особливо важливою для користувачів із власним акаунтом. Тут вдалося перевірити, як зберігаються всі попередні обробки. Можна було сортувати записи за різними параметрами (назвою файлу, датою, алгоритмом), а пошук відмінно відпрацьовував, даючи можливість швидко знайти потрібний файл.

Не оминув увагою і механізм автоматичного очищення старих файлів. Для неавторизованих користувачів файли зникали через 30 хвилин, а для авторизованих зберігалися годину. Це підтвердилось під час тестування: у логах було видно, що фоновий процес (воркер) видаляє застарілі файли саме за цим розкладом, підтримуючи порядок у системі та не засмічуючи сховище.

У підсумку ручне тестування показало, що додаток працює стабільно, інтерфейс інтуїтивний, а всі заявлені функції виконуються без проблем. Тобто користувачі можуть покладатися на цей сервіс, не переймаючись труднощами чи помилками.

3.6 Висновки до розділу

У цьому розділі я продемонстрував, як ідеї та алгоритми стиснення даних можуть бути реалізовані у вигляді повноцінного програмного рішення, орієнтованого на практичне використання. Я взяв за основу відомі підходи до стиснення, такі як Huffman, Gzip, Deflate, FFmpeg, JPEG, WebP, AAC, і доповнив їх власним алгоритмом, пристосованим до конкретних потреб. Завдяки цьому вдалося

побудувати гнучкий та ефективний сервіс, що враховує формат файлу, його розмір, а також особливості взаємодії з користувачами.

Оновлені та адаптовані алгоритми дозволили поліпшити баланс між якістю стиснення та швидкістю обробки, а механізм автоматичного вибору підходящого методу спростив роботу для кінцевого користувача. Удосконалення Huffman-кодування та адаптивне налаштування рівня стиснення для Gzip та Deflate зробило процес більш оптимізованим. Використання FFmpeg із параметрами CRF та libx265 для відео, а також AAC із динамічним бітрейтом для аудіо, дало змогу якісно та компактно зберігати мультимедійні матеріали.

Розробка власного алгоритму для текстових файлів стала спробою досягти максимальної простоти та швидкості там, де інші рішення можуть бути надмірними або занадто «важкими». Цей алгоритм демонструє, що в деяких випадках спрощений підхід є цілком життєздатним, якщо добре зрозуміти природу вхідних даних.

Інтеграція усіх цих елементів у єдину систему вимагала ретельного продумування архітектури, реалізації бази даних, а також забезпечення стабільної взаємодії між бекендом та клієнтською частиною. Приділено увагу і зручності користувачів: авторизація, історія стиснених файлів, автоматичне очищення застарілих даних — усе це робить роботу з додатком комфортною та зрозумілою.

Процес тестування — як автоматизованого, так і ручного — гарантував надійність і правильність реалізації. Завдяки цьому вдалося завчасно виявити й виправити недоліки, забезпечивши стабільну роботу сервісу та позитивний досвід для кінцевих користувачів.

Отже, у ході роботи над цим розділом було закладено технічний фундамент для програмного рішення, яке поєднує в собі ефективні алгоритми стиснення, зручний інтерфейс, логічну модель даних та гнучку архітектуру. Усе разом це формує базу для подальших удосконалень та розвитку проєкту.

ВИСНОВКИ

У результаті виконаного дослідження на тему «Моделі, методи та алгоритми оптимізації сервісу стиснення файлів» було зроблено кілька важливих висновків.

По-перше, теоретичний аналіз сучасних методів стиснення файлів показав актуальність дослідження алгоритмів і технологій, що забезпечують оптимізацію розміру даних. Огляд відомих алгоритмів, таких як Gzip, Huffman, Deflate, WebP, FFmpeg, та AAC, дозволив виявити їх сильні сторони та обмеження. Це допомогло визначити можливості для вдосконалення існуючих технологій та розробки більш ефективних підходів до стиснення файлів.

По-друге, дослідження процесів інтеграції алгоритмів стиснення у веб-додаток продемонструвало необхідність оптимального управління даними для забезпечення продуктивності системи. Використання реляційної бази даних для збереження інформації про користувачів і файли забезпечило гнучкість у роботі з великими обсягами даних, а також дозволило ефективно розподілити ресурси.

По-третє, розробка власного алгоритму стиснення текстових файлів стала значним кроком у вдосконаленні функціональності сервісу. Запропонований алгоритм, що базується на ідеях частотного аналізу символів, забезпечив баланс між швидкістю обробки та ефективністю стиснення, особливо для текстів із високою частотою повторів.

На завершальному етапі було виконано детальне тестування додатку, як автоматизоване за допомогою pytest, так і ручне. Цей процес дозволив перевірити всі ключові функції: авторизацію користувачів, стиснення файлів різних типів, а також автоматичне очищення застарілих даних. Тестування підтвердило стабільність і відповідність роботи додатку сучасним вимогам до подібних систем.

Отже, проведене дослідження підкреслює значення оптимізації алгоритмів стиснення для покращення продуктивності та зручності використання. Запропоновані методи і рішення забезпечують високу ефективність роботи додатку та відкривають перспективи для подальшого розвитку технологій стиснення даних у веб-сервісах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gailly, J.-L., & Adler, M. (1996). DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/rfc1951/
2. Deutsch, P. (1996). GZIP file format specification version 4.3. RFC 1952. IETF. https://datatracker.ietf.org/doc/rfc1952/
3. Huffman, D.A. (1952). A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, 40(9), 1098–1101. https://ieeexplore.ieee.org/document/4051119
4. Howard, P.G., & Vitter, J.S. (1992). Analysis of Arithmetic Coding for Data Compression. Information Processing & Management, 28(6), 749–763.
5. Pennebaker, W.B., & Mitchell, J.L. (1993). JPEG Still Image Data Compression Standard. Van Nostrand Reinhold. ISBN: 978-0442012724.
6. Google Developers. WebP Compression Techniques. Retrieved from https://developers.google.com/speed/webp/docs/compression
7. FFmpeg Documentation. Official FFmpeg Documentation. https://ffmpeg.org/documentation.html
8. Python Software Foundation. Python 3 Documentation. https://docs.python.org/3/
9. Flask Documentation. Flask: The Python Microframework. https://flask.palletsprojects.com/
10. SQLAlchemy Documentation. SQLAlchemy ORM Documentation. https://docs.sqlalchemy.org/
11. Python Software Foundation. gzip — Support for gzip files. https://docs.python.org/3/library/gzip.html
12. Pillow (PIL Fork) Documentation. Image Compression and Transformation Tools. https://pillow.readthedocs.io/
13. Burrows, M., & Wheeler, D.J. (1994). A Block-sorting Lossless Data Compression Algorithm. Digital Systems Research Center, Technical Report 124.

14. Salomon, D., & Motta, G. (2010). Handbook of Data Compression (5th ed.). Springer. ISBN: 978-1848829039.
15. Google Cloud. Cloud Storage and Object Lifecycle Management. https://cloud.google.com/storage/docs/lifecycle
16. MDN Web Docs. HTTP Over TLS. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_over_tls
17. OpenSSL Project. OpenSSL Toolkit Documentation. https://www.openssl.org/docs/
18. Decompression and Archiving Utilities in Linux. tar, gzip, bzip2 and zip documentation. https://www.gnu.org/software/tar/manual/
19. Storer, J.A., & Szymanski, T.G. (1982). Data Compression via Textual Substitution. Journal of the ACM, 29(4), 928–951.
20. Ziv, J., & Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 23(3), 337–343.
21. Deutsch, P. (1996). ZLIB Compressed Data Format Specification version 3.3. RFC 1950. IETF. https://datatracker.ietf.org/doc/rfc1950/
22. Seward, J. (1996). bzip2 and libbzip2. https://sourceware.org/bzip2/
23. Pavlov, I. (2016). 7-Zip LZMA Compression Reference. 7-Zip Official Site. https://www.7-zip.org/
24. Aboulnaga, A., et al. (2001). “Compressing the XML Web.” In Proceedings of the 17th International Conference on Data Engineering (ICDE), IEEE.
25. W3C. HTTP/2 Specification. https://http2.github.io/
26. Bray, T. (Ed.) (2017). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. IETF. https://datatracker.ietf.org/doc/rfc8259/
27. Freed, N., & Borenstein, N. (1996). Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046. IETF. https://datatracker.ietf.org/doc/rfc2046/
28. GNU Project. (n.d.). gzip Documentation. https://www.gnu.org/software/gzip/manual/gzip.html
29. PKWARE. (2007). APPNOTE.TXT - .ZIP File Format Specification. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

30. International Organization for Standardization. (1994). ISO/IEC 10918-1:1994 - Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines (JPEG).
31. International Organization for Standardization. (2016). ISO/IEC 15444-1:2016 - JPEG 2000 image coding system: Core coding system.
32. Matplotlib Documentation. https://matplotlib.org/stable/index.html
33. Pandas Documentation. https://pandas.pydata.org/docs/
34. Gunicorn Documentation. https://gunicorn.org/
35. Werkzeug Documentation. https://werkzeug.palletsprojects.com/
36. Docker Documentation. https://docs.docker.com/
37. PostgreSQL Documentation. https://www.postgresql.org/docs/
38. 7-Zip LZMA SDK. https://www.7-zip.org/sdk.html
39. Python Software Foundation. zipfile — Work with ZIP archives. https://docs.python.org/3/library/zipfile.html
40. Python Software Foundation. bz2 — Compression support. https://docs.python.org/3/library/bz2.html
41. Python Software Foundation. lzma — Compression using the LZMA algorithm. https://docs.python.org/3/library/lzma.html
42. ACM SIGMOD Record. Various issues on data compression and database management. ACM Press. https://sigmod.org/publications/sigmod-record/

ДОДАТКИ

Додаток А

Фрагменти програмниих лістингів

```
from PIL import Image
import zlib
import gzip
import os
from app.config import Config
import zipfile
import subprocess

from app.utils.file_operations import get_compressed_file_path

def compress_with_pillow(filepath, quality, format="JPEG",
user_folder=None):
"""
Стиснення зображень за допомогою Pillow.
:param filepath: Шлях до вхідного зображення.
:param quality: Якість стиснення (1-100).
:param format: Формат ('JPEG', 'WEBP').
:return: Шлях до стислого зображення.
"""
output_path = get_compressed_file_path(filepath, user_folder)
with Image.open(filepath) as img:
img.save(output_path, format=format, quality=quality)
return output_path

def compress_with_gzip(filepath, user_folder=None):
"""
Стиснення файлів за допомогою gzip (без втрат).
:param filepath: Шлях до вхідного файлу.
:return: Шлях до стислого файлу.
"""
```

```
output_path = get_compressed_file_path(filepath, user_folder,
extension=".gz")

with open(filepath, "rb") as f_in:
with gzip.open(output_path, "wb") as f_out:
f_out.writelines(f_in)
return output_path

def compress_with_deflate(filepath, user_folder=None):
"""
Стиснення файлів за допомогою Deflate.
:param filepath: Шлях до вхідного файлу.
:return: Шлях до стислого файлу.
"""
output_path = get_compressed_file_path(filepath, user_folder,
extension=".deflate")

with open(filepath, "rb") as f_in:
data = f_in.read()
compressed_data = zlib.compress(data)
with open(output_path, "wb") as f_out:
f_out.write(compressed_data)
return output_path

def custom_huffman_compression(data):
"""
Проста реалізація стиснення Хаффмана для текстових даних.
:param data: Вхідні дані (bytes).
:return: Стиснені дані (бітовий рядок) та таблиця Хаффмана.
"""
from heapq import heappush, heappop, heapify
from collections import Counter

# Підрахунок частоти символів
freq = Counter(data)
```

```
heap = [[weight, [char, ""]] for char, weight in freq.items()]
heapify(heap)

while len(heap) > 1:
lo = heappop(heap)
hi = heappop(heap)
for pair in lo[1:]:
pair[1] = "0" + pair[1]
for pair in hi[1:]:
pair[1] = "1" + pair[1]
heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

huffman_code = {char: code for char, code in heappop(heap)[1:]}
compressed_data = "".join(huffman_code[byte] for byte in data)

return compressed_data, huffman_code

def compress_with_huffman(filepath, user_folder=None):
"""
Стиснення текстових файлів за допомогою Хаффмана.
:param filepath: Шлях до вхідного файлу.
:param user_folder: Користувацька папка для збереження.
:return: Шлях до стислого файлу.
"""

output_path = get_compressed_file_path(filepath, user_folder,
extension=".huffman")

with open(filepath, "rb") as f_in:
data = f_in.read()
compressed_data, huffman_code = custom_huffman_compression(data)

# Перетворюємо бітовий рядок у байти для запису у файл
byte_array = bytearray()
for i in range(0, len(compressed_data), 8):
```

```
byte_chunk = compressed_data[i : i + 8]
byte_array.append(int(byte_chunk, 2))

with open(output_path, "wb") as f_out:
# Записуємо таблицю Хаффмана (для декодування)
f_out.write(len(huffman_code).to_bytes(4, "big"))
for key, value in huffman_code.items():
f_out.write(len(value).to_bytes(1, "big"))  # Довжина коду
f_out.write(key.to_bytes(1, "big"))  # Символ
f_out.write(int(value, 2).to_bytes((len(value) + 7) // 8, "big"))

# Записуємо стиснені дані
f_out.write(byte_array)

return output_path

def compress_image(filepath, quality, algorithm, format="JPEG",
user_folder=None):
"""
Стиснення зображень за допомогою Pillow.
:param filepath: Шлях до вхідного зображення.
:param quality: Якість стиснення (1-100).
:param algorithm: Алгоритм для вибору (не використовується для
зображень).
:param format: Формат ('JPEG', 'WEBP').
:return: Шлях до стислого зображення.
"""

output_path = get_compressed_file_path(filepath, user_folder)

if quality > 85:
quality = 80
with Image.open(filepath) as img:
# Конвертація в RGB, якщо зображення має альфа-канал (RGBA)
if format.upper() == "JPEG" and img.mode == "RGBA":
```

```
img = img.convert("RGB")
elif img.mode not in ["RGB", "L"]:  # Перевірка для інших режимів
img = img.convert("RGB")

# Збереження зображення з обраним форматом і якістю
img.save(output_path, format=format.upper(), quality=quality)

return output_path

def compress_to_zip(filepath, user_folder=None):
"""
Стиснення файлу або папки у формат ZIP.
:param filepath: Шлях до вхідного файлу або папки.
:return: Шлях до стислого ZIP-файлу.
"""
if not os.path.exists(filepath):
raise FileNotFoundError(f"Файл або папка не знайдені: {filepath}")

# Генеруємо шлях до вихідного ZIP-файлу
base_name = os.path.basename(filepath)
output_path    =    get_compressed_file_path(base_name,    user_folder,
extension=".zip")

# Створення ZIP-архіву
with zipfile.ZipFile(output_path, "w", zipfile.ZIP_DEFLATED) as zipf:
if os.path.isfile(filepath):
# Додаємо один файл
zipf.write(filepath, arcname=base_name)
else:
# Рекурсивно додаємо всі файли та папки
for root, _, files in os.walk(filepath):
for file in files:
file_path = os.path.join(root, file)
arcname = os.path.relpath(
file_path, start=os.path.dirname(filepath)
```

```
)
zipf.write(file_path, arcname=arcname)

return output_path

def compress_video_file(filepath, algorithm="ffmpeg",
user_folder=None):
"""
Стиснення відеофайлів.
:param filepath: Шлях до вхідного відеофайлу.
:param algorithm: Алгоритм ('ffmpeg' — дефолтний).
:return: Шлях до стислого відео.
"""
if algorithm != "ffmpeg":
raise ValueError("На даний момент підтримується лише ffmpeg для
відеофайлів.")

output_path = get_compressed_file_path(filepath, user_folder)

command = [
"ffmpeg",
"-i",
filepath,
"-vcodec",
"libx265",
"-crf",
"28",
output_path,
]
subprocess.run(command, check=True)
return output_path

def compress_audio_file(filepath, algorithm="aac", user_folder=None):
"""
Стиснення аудіофайлів.
```

```
:param filepath: Шлях до вхідного аудіофайлу.
:param algorithm: Алгоритм стиснення (за замовчуванням 'aac').
:param user_folder: Користувацька папка для збереження стислого файлу.
:return: Шлях до стислого аудіофайлу.
"""
if algorithm != "aac":
raise ValueError(
"На даний момент підтримується лише 'aac' для стиснення аудіо."
)

output_path = get_compressed_file_path(filepath, user_folder)

# Команда для стиснення аудіофайлу
command = [
"ffmpeg",
"-i",
filepath,
"-acodec",
"aac",  # Аудіокодек
"-b:a",
"128k",  # Бітрейт для стиснення аудіо
output_path,
]

# Виконання команди
subprocess.run(command, check=True)
return output_path

def compress_with_custom_algorithm(filepath, user_folder=None):
"""
Оптимізований власний алгоритм стиснення текстових файлів.
:param filepath: Шлях до вхідного файлу.
:param user_folder: Користувацька папка для збереження.
:return: Шлях до стислого файлу.
"""
```

```
output_path = get_compressed_file_path(filepath, user_folder,
extension=".custom")

with open(filepath, "r", encoding="utf-8") as f_in:
data = f_in.read()

# Підрахунок частоти символів
frequency = {}
for char in data:
frequency[char] = frequency.get(char, 0) + 1

# Побудова бітового словника (аналог Huffman-кодування)
from heapq import heappush, heappop, heapify

heap = [[freq, [char, ""]] for char, freq in frequency.items()]
heapify(heap)

while len(heap) > 1:
lo = heappop(heap)
hi = heappop(heap)
for pair in lo[1:]:
pair[1] = "0" + pair[1]
for pair in hi[1:]:
pair[1] = "1" + pair[1]
heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

huffman_code = {char: code for char, code in heappop(heap)[1:]}

# Кодування даних
encoded_data = "".join(huffman_code[char] for char in data)

# Перетворення на байти
byte_array = bytearray()
for i in range(0, len(encoded_data), 8):
byte_chunk = encoded_data[i:i+8]
```

```
byte_array.append(int(byte_chunk.ljust(8, "0"), 2))

# Перевірка розміру перед записом
original_size = len(data.encode("utf-8"))
compressed_size = len(byte_array) + len(huffman_code) * 4

if compressed_size >= original_size:
print("Стискання неефективне. Повертаємо оригінальний файл.")
return filepath

# Збереження стислого файлу
with open(output_path, "wb") as f_out:
# Записуємо кількість символів у словнику
f_out.write(len(huffman_code).to_bytes(4, "big"))

# Записуємо словник
for char, code in huffman_code.items():
f_out.write(len(char.encode("utf-8")).to_bytes(1, "big"))
f_out.write(char.encode("utf-8"))
f_out.write(len(code).to_bytes(1, "big"))
f_out.write(int(code, 2).to_bytes((len(code) + 7) // 8, "big"))

# Записуємо стиснені дані
f_out.write(byte_array)

return output_path
```