Звіт до комп'ютерного проєкту на тему:

"Дослідження методів компресії даних та створення власної програми-кодека"

Підготувала команда 12:
Андрій Кульбаба
Оксана Москв'як
Олег Гуцуляк
Олександр Стаднік

8 травня 2025 р.

Зміст

1 Вступ

Компресія даних є важливою галуззю інформатики, що лежить на перетині теорії інформації, кодування та алгоритмів. Цей проєкт присвячений практичному дослідженню та реалізації класичних методів стиснення. Теоретичною основою були алгоритми LZ77, LZW, LZ78 та кодування Гаффмана, вивчені в курсі "Дискретна математика".

2 Мета та завдання проєкту

Метою проєкту є реалізація вибраних алгоритмів стиснення (LZ77, LZW, LZ78, Deflate, RLE), розробка програмного кодека з графічним інтерфейсом для їх застосування, та експериментальне дослідження ефективності цих алгоритмів на різних типах даних.

Основні завдання включали:

- Програмна реалізація зазначених алгоритмів стиснення.
- Створення застосунку-кодека для операцій стиснення/розпакування файлів.
- Тестування розробленого кодека та порівняльний аналіз продуктивності алгоритмів.

3 Реалізація алгоритмів

3.1 Huffman coding (Кодування Гаффмана)

Huffman coding – алгоритм оптимального префіксного кодування з мінімальною надмірністю, реалізований на основі побудови дерева Хаффмана. Алгоритм закодовує певні послідовності, надаючи найбільш використовуваним послідовностям коди найменшої довжини, і навпаки.

Основні компоненти, використані при реалізації:

- `class Node` – прототип об'єкта вершини дерева Гаффмана, використовується для зберігання інформації про символи, які будуть закодовані.
- `class HuffmanTree` – об'єкт, який містить реалізацію алгоритму.

Методи:

- Функція `char_frequency` – будує словник частоти значень, що зберігає символ та кількість його використання.

- Функція `codes_generation` – рекурсивно присвоює бінарні коди кожному символу, безпосередньо використовуючи структуру дерева та роботу із вершинами, що містять основну інформацію про кількість символів. В основі цього лежить pre-order (прямий) обхід дерева, який це забезпечує.

- Функція `tree` – будує дерево Гаффмана, об'єднуючи дві вершини із найменшим значенням (частотою), через використання структури min-heap, тобто повного бінарного дерева, де кожна вершина менша або дорівнює своїм синам. Процес триває доки не залишиться одна вершина, тобто корінь дерева.

- Функція `compress_file` – виконує основну мету алгоритму – закодування файлу та збереження необхідних для декомпресії даних. Спочатку читаємо, отриманий від користувача файл за допомогою `mmap`. Отримуємо масив байтів, що відповідають нашому файлу. Наступним кроком відбувається перевірка чи не побудовано вже дерева, тоді створюється словник, створюються вершини, безпосередньо беручи участь в реалізації дерева Гаффмана, далі створюється дерево і генеруються бінарні коди. Інформація перетворюється у біти за допомогою `bitarray` через додавання бітового рядка для кожного символу до кінцевого потоку бітів, задля збереження довжини справжнього закодованого файлу та об'єднання значень із словника в послідовність, яка була першочергово. Як результат отримуємо файл з розширенням `.bin`, що містить закодовану інформацію про файл та файл `.json`, який містить словник кодів та значень, розширення початкового файлу та довжину справжньої інформації.

- Функція `decompress_file` – виконує декомпресування стиснутого раніше файлу. Дані зчитуються у масив бітів `bitarray` та згодом обрізаються згідно з довжиною справжньої інформації, отриманої із словника. Запускаємо цикл, який ітерується по наших даних, шукаючи в словнику відповідну послідовність. Коли цю послідовність знайдено, то її додано до загального результату. Відновленні дані записуються у файл із початковим розширенням, яке ми дістали із словника.

Використані структури / модулі:

- `mmap` – для зменшення використання фізичної пам'яті, універсальності читання різноманітних файлів.
- `deque` – для видалення переглянутого "декодованого"знаку.
- `defaultdict` – у створенні словника, який зберігає кількість використання певного знаку.
- `bitarray` – для збереження результату кодування із байтів у біти та навпаки.
- `heapq` – використовується як структура min-heap – повне бінарне дерево, корінь якого має найменше значення серед усіх його нащадків, це має виконуватись для лівого і правого піддерев, потрібна щоб об'єднати дві вершини мінімального значення в 1 при побудові.

Переваги, які було визначено під час дослідження:

- Алгоритм не потребує попереднього аналізу всього файлу (працює потоково).
- Стискає багато різних типів файлів, від json до wav.
- Дані, пропоновані для стискання, швидко стискаються, у порівнянні з іншими алгоритмами.

Недоліки, які було визначено під час дослідження:

- Алгоритм не завжди працює ефективно для даних без повторів.
- Розтискає дані значно повільніше ніж інші алгоритми.
- Для великих за розміром файлів стискає не надто ефективно.

3.2 LZ77

Алгоритм LZ77 є одним із класичних методів словникового стиснення даних без втрат. Його основна ідея полягає у пошуку повторюваних послідовностей байтів у вже оброблених даних та заміні цих послідовностей спеціальними посиланнями. Цей підхід базується на концепції "ковзного вікна"(sliding window).

Принцип роботи "ковзного вікна": Основні компоненти, використані при реалізації:

- Клас LZ77: Цей клас інкапсулює всю логіку алгоритму, включаючи процеси стиснення та декомпресії. Він оперує параметрами `window_size` (розмір буфера пошуку, що налаштовується) та `lookahead_buffer_size` (фіксований розмір буфера попереднього перегляду).

- Функція `find_match`: Це ключова функція, відповідальна за пошук найдовшого можливого збігу для поточної послідовності байтів (що починається в буфері попереднього перегляду) всередині буфера пошуку.

- Оптимізація пошуку: Для прискорення пошуку потенційних збігів у цій реалізації використовується хеш-таблиця (стандартний словник Python `dict`). Коли потрібно знайти збіг для поточної 3-байтової послідовності, алгоритм швидко отримує з хеш-таблиці список позицій, де така ж послідовність зустрічалася раніше.

- Очищення хеш-таблиці: Щоб хеш-таблиця не розросталася безмежно і містила лише актуальні позиції (ті, що все ще знаходяться в межах поточного ковзного вікна), реалізовано механізм її очищення, який очищає дані, які вийшли за межі `window_size`.

Функція `compress_file`: Ця функція керує повним процесом стиснення вхідного файлу.

- Читання файлу: Вхідний файл читається з використанням `mmap`.

- Ітерація та пошук збігів: Алгоритм проходить по даних, на кожній позиції викликаючи функцію `find_match` для пошуку найдовшого збігу.

- Формування вихідного потоку: Стиснені дані записуються у вигляді бітового потоку за допомогою бібліотеки `bitarray`.

- Схема кодування:

  – Збіг: Починається з 1 біта-прапорця, що вказує на збіг. Далі йдуть біти, що кодують відстань та довжину. У цій реалізації це 16 бітів: 12 бітів для відстані (дозволяє відстані до 4095) та 4 біти для довжини (дозволяє довжини до 15, плюс мінімальна довжина збігу).

  – Літерал: Починається з 1 біта-прапорця, що вказує на літерал (наприклад, False або 0). Далі йдуть 8 бітів, що представляють сам байт літералу.

- Збереження метаданих: На початку стисненого файлу зберігається службова інформація, необхідна для правильної декомпресії. У цій реалізації це інформація про розширення оригінального файлу: спочатку записується довжина рядка розширення, а потім самі байти цього рядка. Для пакування цих метаданих у бінарний формат використовується модуль `struct`.

Функція `decompress_file`: Відповідає за процес відновлення оригінальних даних зі стисненого файлу.

- Читання метаданих: Спочатку з файлу зчитується інформація про розширення оригінального файлу.
- Читання стиснених даних: Основний потік стиснених даних зчитується в об'єкт `bitarray`.
- Декодування бітового потоку: Алгоритм послідовно читає біти з потоку. Перший біт кожного коду (прапорець) визначає, чи є наступний запис літералом чи посиланням (відстань, довжина).
- Відтворення даних:
  - Якщо декодовано літерал, він просто додається до вихідного буфера (`bytearray`).
  - Якщо декодовано пару (відстань, довжина), то довжина байтів копіюється з уже декодованої частини вихідного буфера, починаючи з позиції, що на відстань байтів раніше від поточного кінця буфера.
- Запис результату: Відновлені дані записуються у вихідний файл, якому надається збережене оригінальне розширення.

Використані структури / модулі:

- `mmap / memoryview`: Для ефективної роботи з файлами та доступу до даних в пам'яті.
- `dict` (хеш-таблиця): Для індексації позицій та прискорення пошуку збігів.
- `bitarray`: Для гнучкої роботи з бітовими потоками змінної довжини.
- `struct`: Для пакування/розпакування бінарних метаданих (інформації про розширення).

3.3 LZ78

Алгоритм LZ78 є одним із методів LZ стиснення даних без втрат, який будує словник під час обробки вхідного потоку даних та зберігає побачені підрядки і замінює повторення посиланнями на словник.

`compress(data: bytes) → list[tuple[int, bytes]]`

- Ініціалізує порожній словник, порожній результат і пустий буфер.
- Для кожного байта з вхідних даних додає його до буфера й перевіряє:
  - якщо новий рядок уже є в словнику — оновлює буфер
  - інакше записує в результат пару (код префіксу або 0, поточний байт), додає новий рядок до словника, очищує буфер
- Після обходу, якщо в буфері щось залишилося, дописує пару (код буфера, пустий байт).
- Повертає список усіх пар.

`decompress(pairs: list[tuple[int, bytes]]) → bytes`

- Ініціалізує словник із записом 0 → порожній рядок і порожній вихідний буфер.
- Для кожної пари (код, байт) зчитує з словника рядок за цим кодом, додає до нього байт, дописує результат у вихідний буфер і заносить у словник новий рядок.
- Повертає конкатенацію всіх відновлених фрагментів.

`compress_file(input_path, output_path)`

- Зчитує весь файл у байтовий масив.
- Викликає `compress(data)`.
- Записує в бінарний файл для кожної пари: 4-байтовий big-endian індекс, 1-байтовий розмір фрагмента (0 або 1), сам байт (якщо є).

`decompress_file(input_path, output_path)`

- Читає бінарний файл в пам'ять.
- По черзі витягує з нього 4 байти → індекс, 1 байт → довжину, потрібну кількість байт → дані; будує список пар.

- Викликає `decompress(pairs)`.
- Записує відновлені байти у файл.

Методи:

- Метод `compress(data: bytes)` приймає байтові дані та виконує стиснення за допомогою словника, який будується динамічно. Алгоритм зчитує послідовності байтів, які вже зустрічалися, і замінює їх індексами в словнику разом з наступним байтом. Повертається список пар (індекс, байт).
- Метод `decompress(pairs: list[tuple[int, bytes]])` приймає список пар (індекс, байт) та динамічно будує словник для відновлення оригінальних даних. Починаючи з пустого словника, він реконструює кожен фрагмент і об'єднує їх у повні дані. Повертає розпаковані байти.
- Метод `compress_file(input_path: str, output_path: str)` зчитує файл у байтовому форматі, стискає вміст за допомогою `compress`, і зберігає стиснені пари у бінарному файлі. Індекси зберігаються як 4 байти, після чого йде сам байт.
- Метод `decompress_file(input_path: str, output_path: str)` читає бінарний файл, інтерпретує його як послідовність стиснених пар (індекс, байт) і відновлює початкові байти за допомогою `decompress`. Відновлені дані записуються у вивідний файл.

Використані структури / модулі:

- `dict` — використовується як словник для збереження повторних послідовностей під час стиснення та розпакування.
- `bytearray` — застосовується для ефективного зберігання результату під час розпакування.
- `int.to_bytes` і `int.from_bytes` — вбудовані методи Python для перетворення цілих чисел у байти та навпаки, що дозволяє працювати з файлами без потреби в зовнішніх бібліотеках.

3.4 LZW

Алгоритм LZW (Lempel–Ziv–Welch) — це класичний метод стиснення без втрат, який ґрунтується на динамічному побудуванні словника послідовностей байтів. На початку словник містить лише окремі байти (0–255), а

в процесі стиснення до нього поступово додаються нові комбінації символів, які повторюються у вхідних даних. Замість збереження самих фрагментів, алгоритм записує коди відповідних словникових записів. Аналогічно, при декодуванні словник відновлюється на льоту за однаковими правилами.

Основні компоненти:

- `Клас LZWCompressor`: Клас `LZWCompressor` реалізує як стиснення, так і відновлення даних за алгоритмом LZW. Він також підтримує збереження/зчитування стиснених файлів у бінарному форматі.

1. `compress(data: bytes)` Це основна функція стиснення байтових даних. Кроки роботи:
   1.1. Ініціалізується словник, де кожен байт (від 0 до 255) відповідає собі.
   1.2. Змінна `w` містить поточну оброблювану послідовність байтів.
   1.3. Для кожного символу `c` (типу `int`, але приводиться до байта):
      - Формується `wc = w + c`.
      - Якщо `wc` вже в словнику, продовжується накопичення.
      - Якщо `wc` ще немає в словнику:
        – Записується код `w` до результату.
        – `wc` додається до словника з новим кодом.
        – Починається нова послідовність `w = c`.
   1.4. Після завершення циклу, якщо залишилась непорожня `w`, її код також додається.

   Структура: словник `bytes → int`.
   Результат: список цілих чисел-кодів.

2. `decompress(compressed_data: list[int])` Функція декодування — відновлює байти з масиву чисел. Кроки роботи:
   2.1. Ініціалізується словник, де кожен код (0–255) відповідає одному байту.
   2.2. `w` — останній розпізнаний фрагмент (починається з першого символу).
   2.3. Для кожного наступного коду `k`:
      - Якщо `k` є в словнику — це відома послідовність.

- Якщо `k == dict_size`, формується нова послідовність як `w + w[0]` (особливий випадок).
- Інакше — дані зіпсовані (`raise ValueError`).

2.4. Додається новий запис у словник: `w +` перший символ поточної послідовності.

2.5. `w` оновлюється.

Структура: словник `int → bytes`.
Результат: об'єкт `bytes`.

3. `compress_file(input_path, output_path="compressed_lzw.bin")`

   - Зчитує файл у байтах.
   - Викликає `compress`.
   - Результат зберігає у двійковий файл (4 байти на одне число, big-endian).
   - Запам'ятовує розширення оригінального файлу у `file_extension`.

   Важливо: не записує словник, оскільки він однаково відтворюється під час декодування.

4. `decompress_file(input_path="compressed_lzw.bin")`

   - Зчитує список цілих кодів по 4 байти з бінарного файлу.
   - Викликає `decompress`.
   - Відновлені байти записує в новий файл з оригінальним розширенням.

   Примітка: Якщо не вказано розширення (тобто не викликано `compress_file`), викидається помилка.

Використані структури в алгоритмі:

- Словник у вигляді `dict[bytes, int]` (при стисненні):
  - Ключами є послідовності байтів, значеннями — цілі числа (коди).
- Словник `dict[int, bytes]` (при декодуванні):
  - Зворотне відображення: коди → послідовності байтів.
- Кодування/декодування чисел у 4 байти:

  - Забезпечує однозначне зчитування при зберіганні у файл.

- Поле `file_extension`:
  - Дає змогу зберегти стиснений файл без втрати типу (наприклад, .txt, .png тощо).

3.5 RLE

RLE – алгоритм безвтратного стиснення даних, який замінює повторювані символи на пару: (символ, кількість повторів).

`compress(data: bytes) → list[tuple[int, bytes]]`

- Якщо вхід порожній — повертає порожній список.
- Ініціалізує поточний байт і лічильник = 1.
- Для кожного наступного байта перевіряє:
  - якщо він збігається з поточним і лічильник < 255 — збільшує лічильник
  - інакше додає до результату (лічильник, поточний байт), скидає лічильник = 1 і оновлює поточний байт
- Після циклу дописує останню пару (лічильник, поточний байт).
- Повертає список.

`decompress(runs: list[tuple[int, bytes]]) → bytes`

- Ініціалізує порожній буфер.
- Для кожної пари (лічильник, байт) повторює байт задану кількість разів і дописує в буфер.
- Повертає зібрані байти.

`compress_file(input_path, output_path)`

- Зчитує байти файлу.
- Викликає `compress`.
- Записує в бінарний файл для кожної пари: 4-байтовий big-endian лічильник і 1 байт значення.

`decompress_file(input_path, output_path)`

- Зчитує весь бінарний файл.
- По 4 байти → лічильник, по 1 байту → значення; збирає список.
- Викликає `decompress`.
- Записує результат у файл.

Методи:

- Метод `compress(data: bytes)` приймає послідовність байтів і стискає повторювані символи. Алгоритм знаходить серії однакових байтів та замінює їх парою (count, byte). Наприклад, AAAA → (4, A). Повертає список пар (count: int, byte: bytes).
- Метод `decompress(pairs: list[tuple[int, bytes]])` приймає список пар (count, byte) та відновлює оригінальну послідовність байтів, повторюючи кожен байт count разів. Повертає розпаковані байти.
- Метод `compress_file(input_path: str, output_path: str)` читає вхідний файл у байтовому режимі, стискає вміст за допомогою `compress`, і записує у вихідний `.bin` файл. Кожна пара (count, byte) записується як: 1 байт для кількості (max 255) + 1 байт значення.
- Метод `decompress_file(input_path: str, output_path: str)` читає стиснений файл у вигляді послідовності пар (count, byte) і відновлює повні байти. Результат записується у файл в оригінальному форматі.

Використані структури / модулі:

- `bytearray` — використовується для поступового зберігання результату у пам'яті при декомпресії.
- `int.to_bytes` і `int.from_bytes` — використовуються для кодування числових значень у байти (і навпаки), що забезпечує сумісність із будь-яким типом файлу.

Примітка: Через свою простоту, RLE ефективно стискає лише дані з великою кількістю повторюваних байтів (наприклад, монохромні зображення або послідовності з нулями). У випадках із складними, неповторюваними даними, алгоритм може навіть збільшити розмір файлу.

3.6 DEFLATE

Алгоритм DEFLATE є потужним методом стиснення даних без втрат, який широко використовується у багатьох популярних форматах (наприклад, ZIP, PNG). Його ефективність полягає в комбінації двох перевірених підходів: пошуку повторюваних послідовностей (подібно до LZ77) та статистичного кодування (кодування Хаффмана).

Загальний принцип роботи DEFLATE: Процес стиснення за алгоритмом DEFLATE можна розділити на два основні етапи:

1. **Пошук та заміна повторів (Етап LZ77):** На цьому етапі програма аналізує вхідні дані, шукаючи послідовності байтів, які вже зустрічалися раніше. Замість того, щоб зберігати ці повтори знову, вони замінюються спеціальними посиланнями. Кожне таке посилання складається з двох частин:

    - Відстань: Вказує, наскільки далеко назад у вже оброблених даних знаходиться початок повторюваної послідовності.
    - Довжина: Вказує, скільки байтів становить ця повторювана послідовність. Байти, для яких не знайдено відповідних повторів (або якщо повтори занадто короткі), залишаються як є і називаються літералами. Результатом цього етапу є послідовність, що складається з літералів та посилань (довжина, відстань).

2. **Кодування Хаффманом (Статистичне кодування):** Отримана на попередньому етапі послідовність літералів та посилань далі стискається за допомогою кодів Хаффмана. Цей метод призначає коротші бінарні коди тим елементам (літералам або посиланням), які зустрічаються частіше, і довші коди тим, які зустрічаються рідше. У даній реалізації алгоритму DEFLATE використовуються фіксовані (статичні) коди Хаффмана. Це означає, що таблиці кодів Хаффмана для літералів/довжин та для відстаней є стандартними та заздалегідь визначеними, а не генеруються на основі частот конкретного вхідного файлу. Це спрощує процес кодування та декодування.

Ключові аспекти реалізації:

- `Клас Deflate`: Виступає як основний координатор, що об'єднує логіку LZ77 для пошуку повторів та механізм кодування Хаффманом для подальшого стиснення.

- Внутрішнє використання `LZ77`: Модуль LZ77 відповідає за перший етап – знаходження повторів та генерацію потоку літералів і посилань (довжина, відстань).
- Фіксовані коди Хаффмана: Реалізація використовує заздалегідь визначені таблиці кодів Хаффмана (згідно зі стандартом DEFLATE) для кодування літералів, довжин збігів та відстаней. Ці таблиці генеруються функціями `_get_fixed_lit_len_encoding_map` та `_get_fixed_dist_encodi`

Процес стиснення (`compress_file`):

- Вхідний файл обробляється логікою LZ77 для отримання послідовності літералів та пар (довжина, відстань).
- Формується вихідний бітовий потік:
  - Записується заголовок блоку, що вказує на використання фіксованих кодів Хаффмана.
  - Кожен літерал або довжина збігу кодується відповідним фіксованим кодом Хаффмана.
  - Якщо це збіг (довжина), то після коду довжини записується відповідна кількість екстра-бітів (якщо потрібно для уточнення довжини), а потім записується код відстані (також з фіксованого дерева Хаффмана) та його екстра-біти.
  - В кінці потоку даних блоку додається спеціальний маркер кінця блоку (End-of-Block).
- На початку стисненого файлу зберігається інформація про розширення оригінального файлу.

Процес розпакування (`decompress_file`):

- Зчитується інформація про розширення файлу.
- Стиснені дані обробляються блок за блоком. Для блоків, стиснених фіксованими кодами Хаффмана:
  - Символи (літерали/довжини та відстані) декодуються за допомогою відповідних фіксованих таблиць Хаффмана.
  - Якщо декодовано літерал, він додається до вихідних даних.
  - Якщо декодовано довжину збігу, зчитуються екстра-біти довжини, потім декодується відстань та її екстра-біти. Після цього відповідна кількість байт копіюється з уже розпакованої частини даних.

  - Обробка блоку завершується при зустрічі маркера кінця блоку.

- Відновлені дані записуються у вихідний файл.

Використані структури / модулі:

- `mmap / memoryview`: Для ефективної роботи з файлами та доступу до даних в пам'яті.
- `dict` (хеш-таблиця): Для індексації позицій та прискорення пошуку збігів.
- `bitarray`: Для гнучкої роботи з бітовими потоками змінної довжини.
- `os`: Для пакування/розпакування бінарних метаданих (інформації про розширення).

3.7 DCMP для wav

4 Створення програми-кодека

Програму-кодек було створено за допомогою бібліотеки PyQt6, яка працює на основі вбудованих віджетів, які було модифіковано відповідно до вимог.

Реалізовано клас `MainWindow`, який успадковується із класу передбаченого бібліотекою.

Методи класу `MainWindow`:

- `__init__` – містить увесь візуальний функціонал та є основою всього застосунку.
- `pick_file` – функція, яка обробляє запит користувача на кнопку "pick a file тобто обрання єдиного файлу для компресії, перевіряє чи вибраний користувачем файл є правильного розширення та повідомляє йому, якщо це не так. Назва вибраного файлу відображається на екрані.
- `compress_file` – функція, яка обробляє процес компресування файлів, а саме першопочатково – вибір користувача, щодо алгоритму для стиснення. Відповідно до вибраного користувачем алгоритму, викликається функція `compress_file` цього алгоритму, куди передається шлях до файлу. Після успішного стиснення на екрані

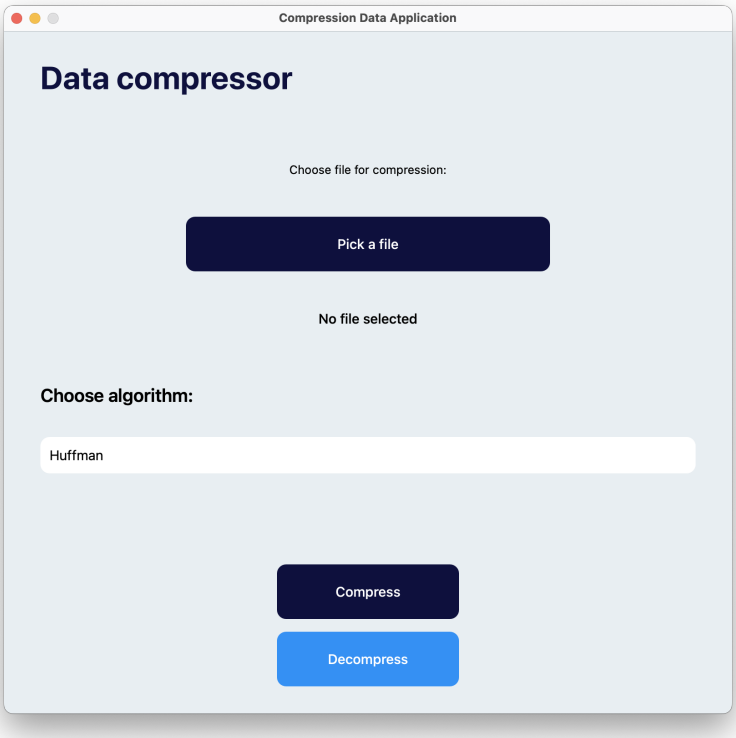


Рис. 1: Інтерфейс програми-кодека під час початку роботи

з'являється інформація про початковий розмір файлу у КВ та розмір цього ж файлу після стиснення, що реалізовано за допомогою модуля `os`. Функція також обробляє випадки, коли користувач натискає кнопку "Compress", не вибравши файлу, повідомляючи про помилку.

- `decompress_file` – функція, яка обробляє процес декомпресування файлів, відповідно до вибраного швидше користувачем алгоритму для стиснення. Для цього алгоритму викликається функція `decompress_file` цього алгоритму. Після успішного розстиснення файлу на екрані з'являється інформація про успішність виконання. Функція також обробляє випадки, коли користувач натискає кнопку "Decompress", не вибравши файлу або не компресувавши його попередньо, повідомляючи про помилку.

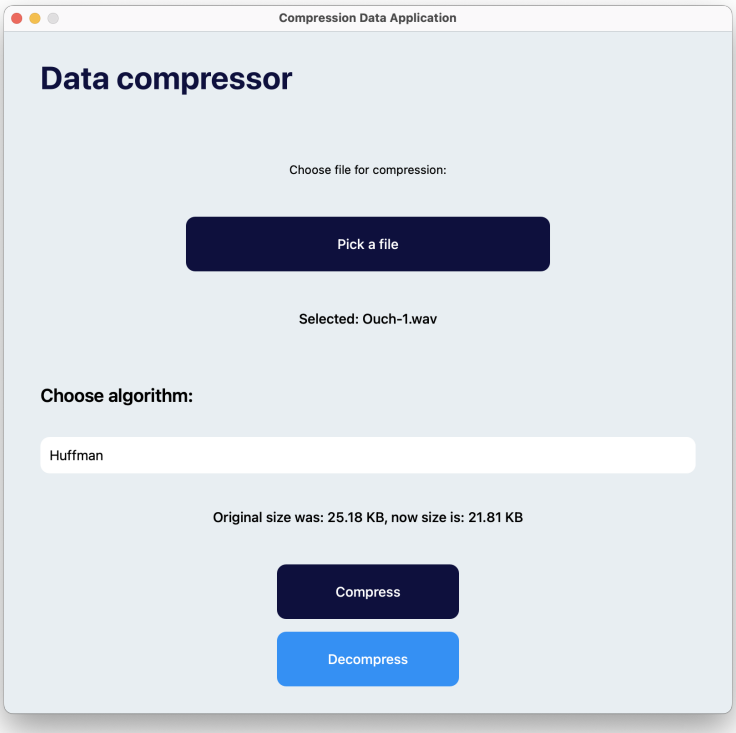


Рис. 2: Інтерфейс програми-кодека після стиснення файлу

5 Тестування

Окрім імплементацій алгоритмів та створення програми-кодека було також проведено 3 етап – тестування та дослідження ефективності алгоритмів, зроблено порівняння алгоритмів для певних форматів та надано візуалізацію у формі графіків. Уся інформація цього етапу знаходиться у файлі `testing_performance.ipynb` на Github.

Результати тестування:

- **Тестування формату .txt**
  - Використані алгоритми: Кодування Гаффмана, LZ77, LZ78, LZW, DEFLATE.
  - Приклад “Місто” В. Підмогильний:
    - Найбільш ефективним алгоритмом у цьому тестуванні стало Кодування Гаффмана, водночас він став найбільш ефективним й по часу стиснення, але найменш ефективним по

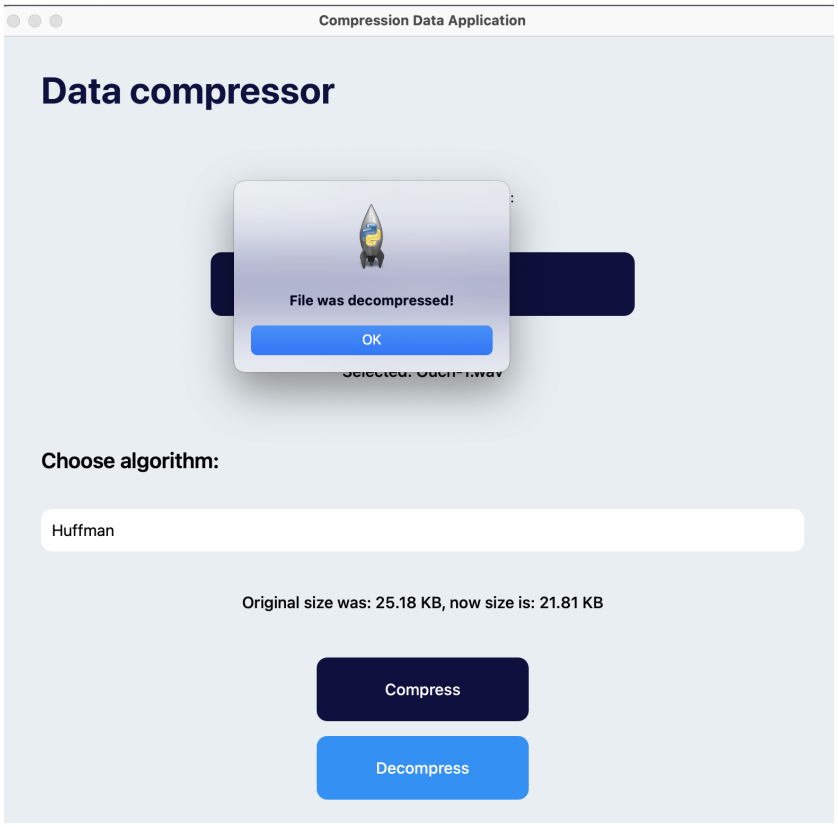
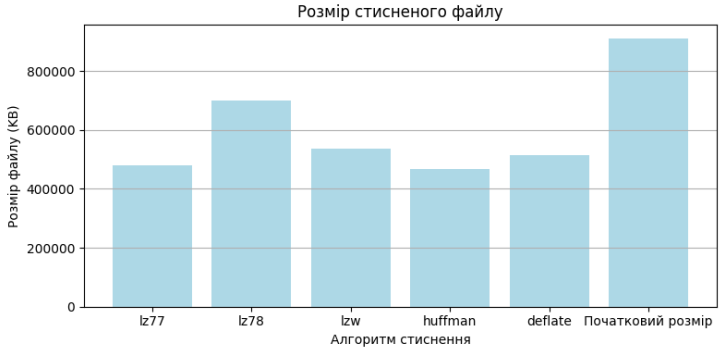


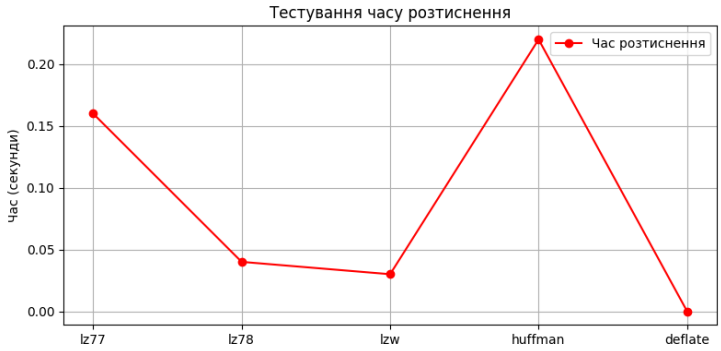
Рис. 3: Інтерфейс програми-кодека під час виведення повідомлення для користувача

часу розтиснення. Найшвидше декомпресія файлу відбувається в алгоритмі DEFLATE.



(а) Результат 1 для "Місто"

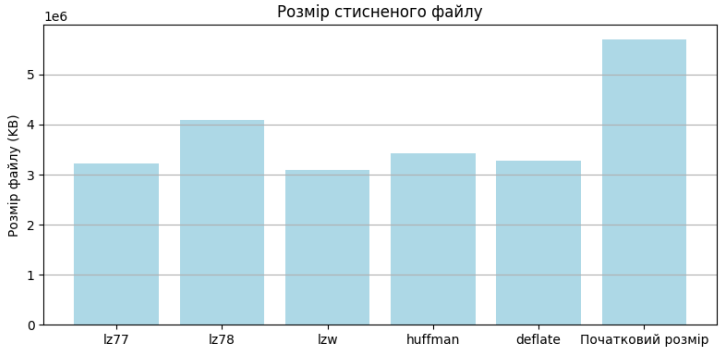
(б) Результат 2 для "Місто"



(в) Результат 3 для "Місто"

Рис. 4: Візуалізація результатів тестування для "Місто"В. Підмогильний

– Приклад “Holy Bible” - великий файл:
  * Найбільш ефективним алгоритмом у цьому тестуванні став LZW. Найбільш ефективним по часу стиснення стало Кодування Гаффмана. Найшвидше декомпресія файлу відбувається в алгоритмі DEFLATE. Найменш ефективним по часу розтиснення досі залишається Гаффман.



(а) Результат 1 для "Holy Bible"

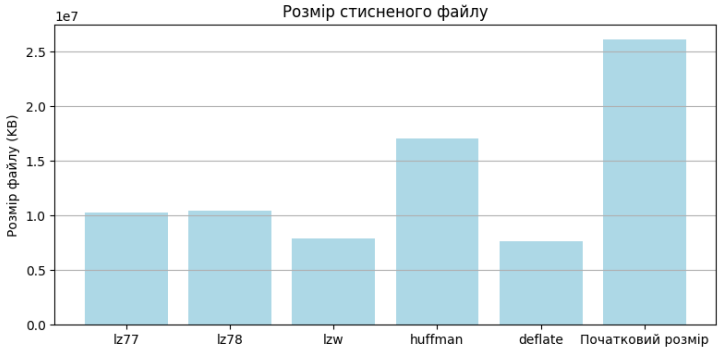
(б) Результат 2 для "Holy Bible"

(в) Результат 3 для "Holy Bible"

Рис. 5: Візуалізація результатів тестування для "Holy Bible"

- **Тестування формату .json**
  - Використані алгоритми: Кодування Гаффмана, LZ77, LZ78, LZW, DEFLATE.
  - Приклад .json файлу довжиною в 11352 рядків:

* Найбільш ефективним алгоритмом у цьому тестуванні став DEFLATE. Найбільш ефективним по часу стиснення стало Кодування Гаффмана. Найшвидше декомпресія файлу відбувається в алгоритмі DEFLATE.



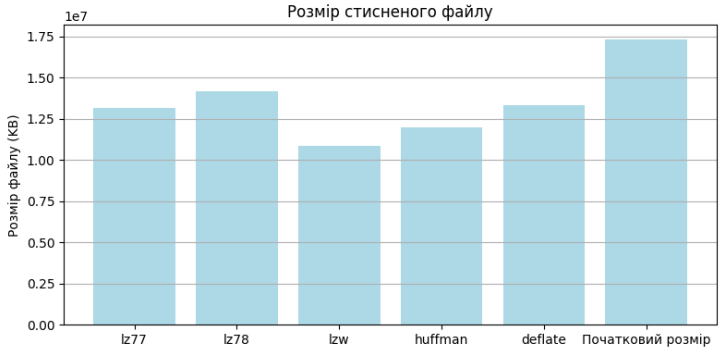
(а) Результат 1 для JSON (11352)

(б) Результат 2 для JSON (11352)

(в) Результат 3 для JSON (11352)

Рис. 6: Візуалізація результатів тестування для .json файлу (11352 рядків)

- **Тестування формату .csv**
  - Використані алгоритми: Кодування Гаффмана, LZ77, LZ78, LZW, DEFLATE.
  - Приклад .csv файлу довжиною в 100000 рядків:
    * Найбільш ефективним алгоритмом у цьому тестуванні став LZW. Найбільш ефективним по часу стиснення стало Кодування Гаффмана. Найшвидше декомпресія файлу відбувається в алгоритмі DEFLATE, як і в попередніх тестуваннях.



(а) Результат 1 для CSV (100000)

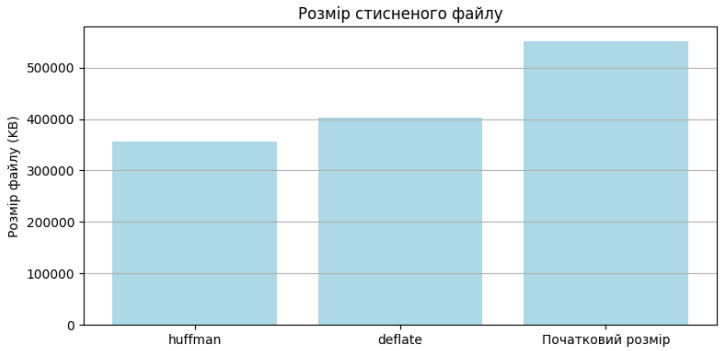
(б) Результат 2 для CSV (100000)

(в) Результат 3 для CSV (100000)

Рис. 7: Візуалізація результатів тестування для .csv файлу (100000 рядків)

- **Тестування формату .bmp**
  - Використані алгоритми: Кодування Гаффмана, DEFLATE.
  - Найефективнішим алгоритмом у цьому тестуванні є DEFLATE. Найбільш ефективним по часу стиснення стало Кодування Гаф-

фмана. Найшвидше декомпресія файлу відбувається в алгоритмі DEFLATE, як і в попередніх тестуваннях.



(а) Результат 1 для BMP

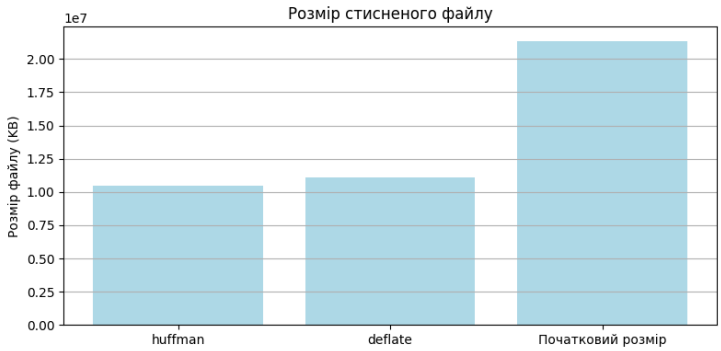
(б) Результат 2 для BMP

(в) Результат 3 для BMP

Рис. 8: Візуалізація результатів тестування для формату .bmp

- **Тестування формату .jpg**
  - Використані алгоритми: Кодування Гаффмана, DEFLATE.

– Найефективнішим алгоритмом у цьому тестуванні є Кодування Гаффмана.



(а) Результат 1 для JPG

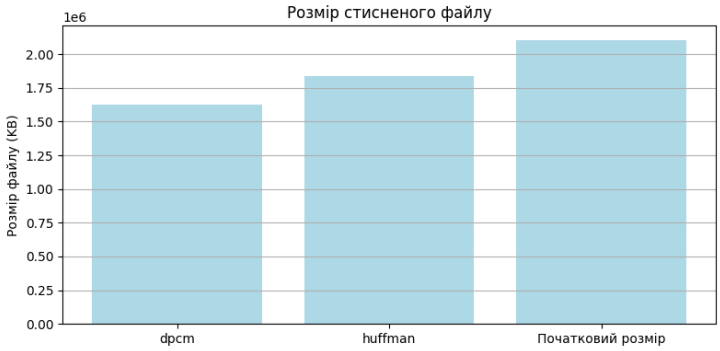
(б) Результат 2 для JPG

(в) Результат 3 для JPG

Рис. 9: Візуалізація результатів тестування для формату .jpg

- **Тестування .wav**
  - Використані алгоритми: Кодування Гаффмана, DPCM.

– Найефективніше у цьому алгоритмі у процесі стиснення та декомпресії показав себе DPCM.



(а) Результат 1 для WAV

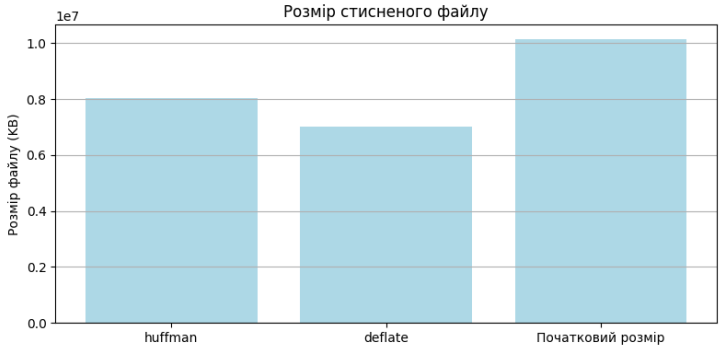
(б) Результат 2 для WAV

(в) Результат 3 для WAV

Рис. 10: Візуалізація результатів тестування для формату .wav

- **Тестування .tiff**
  - Використані алгоритми: Кодування Гаффмана, Deflate.

– Deflate стискає краще файли, але програє у часі декомпресії, що є очікуваним для цього алгоритму.



(а) Результат 1 для TIFF

(б) Результат 2 для TIFF

(в) Результат 3 для TIFF

Рис. 11: Візуалізація результатів тестування для формату .tiff

6 Висновки

У ході виконання проєкту було розроблено та реалізовано низку класичних алгоритмів стиснення, здатних обробляти більшість поширених типів файлів. Важливим етапом стала розробка програмного застосунку з графічним інтерфейсом, який надає користувачам можливість обирати потрібний алгоритм для стиснення та водночас допомагає уникнути застосування невідповідних методів до певних типів даних. Експериментальне тестування продемонструвало, що різні типи файлів реагують на стиснення по-різному: алгоритми, ефективні для текстових даних, можуть виявитися менш дієвими для зображень чи аудіо. Зокрема, при роботі з методом DPCM (дельта-кодування) для звукових файлів було відзначено, що видалення або модифікація компонентів сигналу, несуттєвих для людського сприйняття, може позитивно впливати на ступінь стиснення. Це наштовхує на думку про важливість врахування психоакустичних та візуальних особливостей при розробці спеціалізованих кодеків. Ключовим висновком дослідження стало підтвердження того, що універсального алгоритму стиснення, який би однаково ефективно працював для всіх типів даних, не існує. Вибір оптимального методу завжди залежить від характеристик самого файлу. Загалом, проєкт надав цінну можливість застосувати теоретичні знання, отримані в курсі "Дискретна математика для вирішення практичних завдань у галузі обробки та стиснення даних.