РІВНЕНСЬКИЙ ДЕРЖАВНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ МАТЕМАТИКИ ТА ІНФОРМАТИКИ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА МОДЕЛЮВАННЯ

КВАЛІФІКАЦІЙНА РОБОТА

за освітнім ступенем «Бакалавр»

на тему:

РЕАЛІЗАЦІЯ ТА ПОРІВНЯННЯ АЛГОРИТМІВ АРХІВАЦІЇ ДАНИХ

Виконав:
здобувач вищої освіти 4 курсу
спеціальності 122 «Комп’ютерні науки»
Босик Андрій Андрійович
Перевірив:
к.ф-м.н. Лящук Тарас Григорович

Рівне - 2025

ЗМІСТ

СПИСОК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

GUI – графічний інтерфейс користувача;
LZ – Lempel-Ziv (алгоритм стиснення Лемпеля-Зіва)

LZW – Lempel-Ziv-Welch (алгоритм стиснення Лемпеля-Зіва-Велча) LZC – Lempel-Ziv Complexity (ускладнений алгоритм стиснення Лемпеля-Зіва)

LZT – Lempel-Ziv-Tischer (алгоритм стиснення Лемпеля-Зіва-Тічер)

LZAP – Lempel-Ziv All Prefixes (алгоритм стиснення Лемпеля-Зіва з префіксами)

LZSS – Lempel–Ziv–Storer–Szymanski (алгоритм стиснення Лемпеля-Зіва-Сторер-Шиманский)

LZMW – Lempel-Ziv-Miller-Wegman (алгоритм стиснення Лемпеля-Зіва-Мілєр-Вегман)

BPC – Bits Per Character (біт на символ)

LRU – Least Recently Used (найменше використовуваний) TTL – Time To Live (граничний період часу)

FIFO – First-In, First-Out (перший увійшов, перший вийшов) NRU – Not Recently Used (нещодавно використовувався) NFU – Not Frequently Used (використовується нечасто)

ВСТУП

Актуальність дослідження. Сучасний світ характеризується стрімким зростанням обсягу цифрових даних, що зумовлює необхідність ефективного зберігання та передачі інформації. Архівація даних відіграє важливу роль у забезпеченні оптимального використання ресурсів, зокрема дискового простору та пропускної здатності каналів зв'язку. Сьогодні існує багато алгоритмів компресії, кожен із яких має свої переваги та недоліки залежно від типу оброблюваної інформації. Актуальність цієї роботи полягає у дослідженні існуючих алгоритмів архівації даних, їх порівняльній характеристиці та визначенні оптимального підходу до їх використання.

Мета дослідження. Метою роботи є розробка програми-архіватора, яка використовує різноманітні алгоритми компресії даних, а також оцінка їх ефективності для різних типів інформації.

Завдання дослідження. Досягнення поставленої мети передбачає наступні завдання:

- провести аналіз предметної галузі архівації/стиснення даних;
- дослідити існуючі алгоритми компресії, їх особливості та принципи роботи;
- розробити програмне забезпечення для реалізації архівації/стиснення цифрових даних;
- здійснити експериментальне порівняння алгоритмів архівації/стиснення за ключовими характеристиками (ступінь стиснення, швидкість роботи тощо).

Об'єктом дослідження є процеси архівації та компресії цифрових даних.

Предметом дослідження є програмна реалізація архівації та стиснення даних.

Методи дослідження. У роботі використовуються методи аналізу, синтезу, порівняння та моделювання. Для оцінки ефективності алгоритмів застосовуються експериментальні дослідження та програмна реалізація з наступним тестуванням.

Практичне значення дослідження. Результати роботи можуть бути використані у сфері зберігання даних, їх передавання через комп'ютерні мережі, а також у розробці програмного забезпечення для стиснення файлів. Запропонована кросплатформна програма-архіватор дозволяє автоматизувати процеси архівації, вибираючи найбільш ефективний алгоритм для конкретного типу даних.

Апробація та впровадження результатів дослідження Основні теоретичні та практичні результати дослідження доповідалися та обговорювалися на XVI Всеукраїнській науково–практичній конференції «Інформаційні технології в професійній діяльності» (м. Рівне, 1 листопада 2023 р.) [3.

Структура роботи. Робота складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків. У першому розділі розглядається предметна область, основні поняття та аналізуються алгоритми стиснення даних. У другому розділі детально розглядаються принципи роботи вибраних алгоритмів та проводиться їх порівняння. Третій розділ присвячений розробці програмного забезпечення, його тестуванню та аналізу отриманих результатів.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Основні визначення та поняття

Розглянемо основні методи, що використовуються протягом останніх десятиліть для стиснення даних, включаючи обробку зображень [3] і відео.

Для подальшого викладу матеріалу необхідно ознайомитися з такими ключовими поняттями:

- стиснення без втрат (lossless) – метод компресії, що дозволяє повністю відновити вихідні дані після стиснення без жодних змін;

- стиснення з втратами (lossy) – спосіб стиснення, за якого відновлені дані можуть частково відрізнятися від початкових;
- формула кількості інформації Шеннона – математичний вираз, що визначає мінімально необхідну кількість бітів для кодування повідомлення.
- інформаційна ентропія – показник непередбачуваності появи символів у наборі даних.

Окрім цього, важливо розглянути основні поняття, пов’язані зі стисненням відео та зображень.

Стиснення зображень – це процес зменшення розміру графічного файлу в байтах із мінімально можливими втратами якості, що не перевищують допустимий рівень. Завдяки такому підходу можна зберігати більше зображень у визначеному обсязі пам’яті, а також оптимізувати швидкість їхнього передавання через мережу. Менший розмір файлу дозволяє скоротити навантаження на канал зв’язку, прискорюючи завантаження вебсторінок [4, 5].

Стиснення відео передбачає зменшення кількості бітів, необхідних для представлення відеопослідовності або окремого кадру [6, 7]. Цей процес зазвичай виконується за допомогою програмного забезпечення, яке використовує певний алгоритм для вибору найефективнішого способу компресії. Такі алгоритми, як H.264/AVC і H.265/HEVC, дозволяють суттєво зменшити розмір відеофайлів, іноді у 1000 разів.

Відео-сегментація – це процес розділення відеопотоку на окремі частини (сцени або послідовності кадрів), які є однорідними за певними характеристиками. Найпоширенішими типами сегментації є поділ відео на окремі кадри або сцени [8, 9].

Кластеризація використовується для групування кадрів або відеофрагментів за подібністю кольорів та інших параметрів. Процес починається з розміщення кожного кадру у власному кластері, після чого найбільш схожі групи поступово об’єднуються, доки не залишиться один загальний кластер [10]. Центроїд кожної

групи визначає її ключові характеристики, а кадри, розташовані найближче до центроїда, виступають її репрезентативними зразками [11, 12].

1.2 Методи безвтратного стиснення

Методи стиснення без втрат, як випливає з їхньої назви, забезпечують повне збереження вихідних даних, дозволяючи їх точне відновлення після декомпресії. Вони застосовуються у випадках, коли навіть найменші відмінності між оригінальними та відновленими даними є неприпустимими. Зокрема, стиснення тексту є критично важливим, оскільки навіть незначна зміна може суттєво вплинути на зміст. Наприклад, речення «не надсилайте гроші» та «зараз надсилайте гроші» мають кардинально різні значення.

Такий самий принцип актуальний і для комп’ютерних файлів, фінансових записів чи медичних даних. Наприклад, у випадку стиснення радіологічного зображення з втратами, навіть якщо візуальні зміни будуть непомітними, подальше покращення зображення може спричинити появу артефактів, які можуть ввести лікаря в оману. Помилки у таких сферах можуть мати критичні наслідки, тому збереження цілісності даних є вкрай важливим.

Аналогічно, дані, отримані із супутників, часто використовуються для аналізу змін у навколишньому середовищі, таких як оцінка рослинності чи вирубки лісів. Якщо відновлені дані не будуть ідентичними вихідним, подальша обробка може спотворити результати досліджень. У таких випадках використання алгоритмів стиснення з втратами є недоцільним.

Втім, існують ситуації, коли допустимо жертвувати точністю відновлення заради більшого рівня компресії. У таких випадках застосовують методи стиснення з втратами [1].

Одним із найпоширеніших прикладів стиснення без втрат є ZIP-файли, що дозволяють ефективно зберігати інформацію без втрати даних. Це особливо важливо для виконуваних файлів, оскільки будь-яке пошкодження може зробити їх непридатними для використання.

Серед інших популярних форматів без втрат можна виділити PNG для зображень та FLAC для аудіо. Відеоформати без втрат зустрічаються значно рідше, оскільки займають значний обсяг пам’яті.

1.3 Методи стиснення з втратами

Методи стиснення з втратами допускають часткову втрату інформації, що унеможливлює точне відновлення вихідних даних. Однак натомість ці методи забезпечують значно вищі коефіцієнти стиснення порівняно з безвтратним стисненням [13, 14].

У багатьох випадках, точне відновлення даних не є критично важливим. Наприклад, при передачі мовлення немає потреби зберігати точне значення кожного звукового зразка. Допустима втрата якості залежить від вимог до кінцевого сигналу: якщо потрібно відтворити мовлення телефонної якості, втрати можуть бути значними, тоді як для аудіо рівня компакт-диска прийнятний рівень втрат буде значно нижчим [15, 16].

Аналогічно, при перегляді відео невеликі відмінності між оригінальним і відновленим зображенням не є критичними, якщо вони не створюють помітних візуальних артефактів. Тому відеоконтент зазвичай стискається з використанням методів з втратами [1].

Стиснення з втратами є ідеальним рішенням для роботи з медіафайлами. Це особливо важливо для сервісів потокового передавання, таких як Spotify та Netflix, які щоденно обробляють величезні обсяги даних. Оптимізація розміру файлів без помітного зниження якості дозволяє їм забезпечувати стабільну та ефективну трансляцію контенту [17].

Стиснення з втратами активно використовується в цифрових мультимедійних технологіях, оскільки дозволяє значно зменшити обсяг файлів при збереженні прийнятної якості. У відеоформатах, таких як H.264, H.265 (HEVC) або VP9, алгоритми оптимізують зображення, видаляючи надлишкову або малопомітну для людського ока інформацію. У сфері аудіо популярними є формати MP3, AAC,

Ogg Vorbis, які застосовують психоакустичні моделі, що відсікають звукові частоти, які людське вухо сприймає слабко або не помічає зовсім.

Такі методи стискання дозволяють економити дисковий простір та знижувати навантаження на канали зв'язку, що є критичним для потокових сервісів, онлайн-ігор та мобільних додатків. Однак, використання стиснення з втратами потребує ретельного вибору параметрів компресії: надмірне зменшення розміру файлу може призвести до помітного погіршення якості та появи артефактів, що унеможливлює комфортне сприйняття контенту.

Важливим напрямом у розвитку алгоритмів стиснення з втратами є застосування методів машинного навчання та штучного інтелекту. Наприклад, сучасні нейромережеві підходи дозволяють адаптивно стискати зображення та відео, зберігаючи високу якість при мінімальних втратах. Така технологія вже впроваджується у новітніх відеокодеках (наприклад, AV1) та графічних форматах (JPEG XL, WebP), які мають кращу ефективність порівняно з попередніми стандартами.

Отже, методи стиснення з втратами залишаються ключовим інструментом для ефективного зберігання та передавання цифрових даних, і їхній подальший розвиток спрямований на покращення співвідношення між якістю та ступенем компресії.

1.4 Аналіз алгоритмів компресії на основі словникового підходу

Алгоритми стиснення даних сімейства Lempel-Ziv (LZ) є одними з найпоширеніших методів безвтратного стиснення. Вони базуються на принципі заміни повторюваних фрагментів даних на посилання, що значно зменшує обсяг збережених або передаваних даних. Це не один алгоритм, а ціла група методів (рис. 1.1), що виникла на основі двох ключових алгоритмів, запропонованих Джейкобом Зівом і Абрахамом Лемпелем у 1977 та 1978 роках.

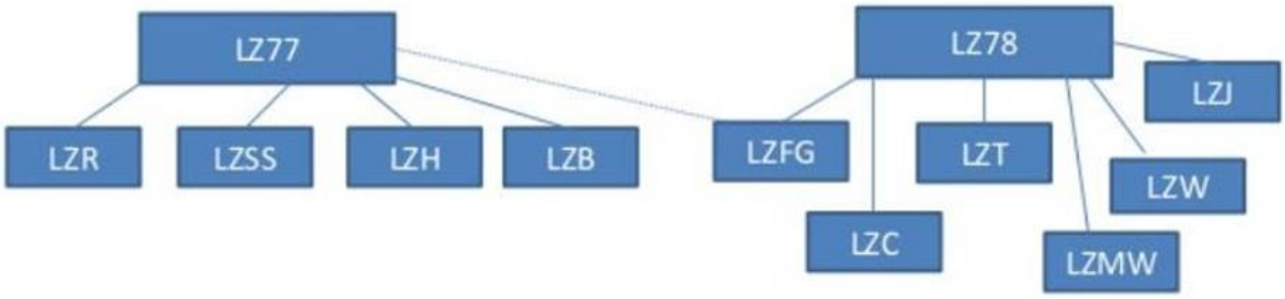

Рис. 1.1 Сімейство алгоритмів LZ.

Алгоритм LZ77 працює шляхом пошуку найдовшого збігу в буфері пошуку та заміни його на посилання у вигляді трійки (зміщення, довжина, наступний символ). Основною перевагою цього підходу є швидке декодування, оскільки кожне посилання просто замінюється відповідним рядком. Проте, процес кодування є більш ресурсозатратним, оскільки алгоритм витрачає значний час на пошук найдовшого збігу.

З метою підвищення ефективності LZ77 було запропоновано кілька його вдосконалень:

- LZSS (Lempel-Ziv-Storer-Szymanski) – усуває необхідність включення окремого символу у кожне кодове слово. Використовує додатковий біт для позначення, чи є поточний вихідний код парою (зміщення та довжина) або окремим символом;
- LZH (Lempel-Ziv-Huffman) – поєднує підхід LZ77/LZSS з алгоритмом Хаффмана. Спочатку відбувається кодування аналогічне LZSS, а потім отримані посилання і символи додатково стискаються за допомогою кодування Хаффмана;
- LZB (Lempel-Ziv-Banikazemi) – оптимізує розміри кодових посилань, дозволяючи використовувати покажчики змінної довжини. Це дозволяє досягти кращого ступеня стиснення, оскільки частіші довжини фраз займають менше місця, ніж рідкісні.

Завдяки гнучкості та можливості подальших удосконалень алгоритми сімейства LZ використовуються в багатьох популярних форматах архівування, таких як ZIP, GZIP, 7z та інші.

1.4.1 Огляд методу LZ77

У 1977 році Джейкоб Зів і Абрахам Лемпель представили LZ77 – перший алгоритм стиснення даних на основі словника, що не вимагає попереднього аналізу вхідного потоку [2]. Цей підхід став основою для багатьох сучасних методів компресії і широко використовується у форматах архівування, таких як ZIP, GZIP та 7z.

Основна ідея LZ77 полягає у використанні повторюваних фрагментів даних у потоці. Якщо певний підрядок уже зустрічався раніше, його можна замінити посиланням на попереднє входження у вигляді трійки (зміщення, довжина, наступний символ). Це дозволяє значно зменшити обсяг збережених даних без втрати інформації.

Алгоритм роботи LZ77 використовує ковзне вікно, що складається з двох частин:

- буфер пошуку – містить частину нещодавно закодованої послідовності;
- буфер перегляду – містить наступну частину даних, які потрібно стиснути.

Алгоритм сканує вікно у пошуках найдовшого збігу між початком буфера перегляду та частиною, що вже збережена у буфері пошуку. Якщо знайдено збіг, він кодується у вигляді трійки (o, l, c), де:

- o (offset) – зміщення до збігу (скільки символів назад потрібно повернутися);
- l (length) – довжина знайденого збігу;
- c (character) – наступний символ після збігу.

Якщо ж збігу немає, алгоритм виводить нульовий покажчик (o = 0, l = 0) і передає перший символ буфера перегляду без змін.

Обмеження параметрів LZ77 – для ефективного стиснення значення зміщення (o) та довжини збігу (l) повинні бути обмежені деякими максимальними константами.

Зазвичай зміщення кодується (12 ÷ 16) бітами, що дозволяє посилатися на символи в межах 0 – 65535 позицій. Це означає, що достатньо зберігати останній 65535 символів у ковзному вікні.

Довжина збігу зазвичай кодується 8 бітами, що дає максимальну довжину збігу 255 символів [19, 20].

LZ77 забезпечує гнучке та ефективне стиснення, особливо для текстових даних. Проте процес пошуку найдовшого збігу може бути доволі ресурсоємним, що впливає на швидкість кодування. Це стало причиною появи оптимізованих версій алгоритму, таких як LZSS, LZH, LZB та інші.

Розглянемо оптимізовані варіанти алгоритму LZ77:

- LZSS (Lempel-Ziv-Storer-Szymanski) – запропонований Storer і Szymanski, алгоритм LZSS усуває необхідність обов’язкового включення наступного невідповідного символу в кожне кодове слово. У LZ77 кожна трійка (o, l, c) містить один символ c, навіть якщо збіг триває до кінця буфера перегляду. LZSS замість цього додає прапорцевий біт, який визначає, чи є кодове слово посиланням (o, l) або просто окремим символом. Це зменшує накладні витрати та покращує коефіцієнт стиснення, особливо для коротких послідовностей.
- LZH (Lempel-Ziv-Huffman) – поєднує LZSS із кодуванням Хаффмана для ще кращої компресії. Перший етап: Виконується стандартне стиснення за LZSS. Другий етап: Використовуючи статистику першого етапу, покажчики та символи кодуються за допомогою Хаффмана. Це дозволяє значно зменшити розмір вихідного файлу, але збільшує складність реалізації.
- LZB (Lempel-Ziv-Banikazemi) – розроблений Мохаммадом Баніказемі, LZB покращує LZSS, використовуючи показники змінної довжини. У LZSS усі покажчики мають однаковий розмір, незалежно від довжини збігу. LZB використовує змінний розмір покажчиків, що дозволяє кодувати часті збіги

ефективніше, зменшуючи загальний розмір стисненого файлу. Додатково, LZB менш чутливий до вибору параметрів, що робить його більш універсальним.

1.4.2 Принцип роботи LZ78

Алгоритм LZ78 був запропонований Джейкобом Зівом і Абрахамом Лемпелем у 1978 році як вдосконалення LZ77. На відміну від LZ77, який використовує ковзне вікно для пошуку повторюваних фрагментів, LZ78 будує явний словник, що дає змогу більш ефективно знаходити та зберігати повторювані послідовності.

Його ключові переваги: швидший пошук збігів завдяки словнику; менша залежність від відстані між повтореннями; ефективність на малих файлах; LZ78 став основою для LZW (Lempel-Ziv-Welch), який використовується у форматах GIF, TIFF та старих архівах UNIX (compress).

Принцип роботи LZ78:

1. Формування словника – кодер створює словник, що містить унікальні підрядки вхідного потоку даних. Кожен запис у словнику складається з індексу (ідентифікатора) та символу. Якщо зустрічається нова унікальна послідовність, вона додається до словника з новим індексом.

2. Кодування – кодер переглядає вхідний потік та знаходить найдовший підрядок, який вже є у словнику. Якщо такий фрагмент знайдено, кодер виводить індекс цього фрагмента + наступний символ. Якщо підрядок відсутній, кодер виводить (0, символ) та додає його до словника. Процес повторюється, доки весь вхідний потік не буде закодовано.

3. Декодування – декодер використовує отримані індекси та символи для відновлення вихідних даних, поступово відтворюючи словник під час декодування.

В таблиці 1.1 наведено приклад роботи алгоритму LZ78.

Табл. 1.1.

Приклад кодування слова "ABABABA" за допомогою LZ78.

Крок	Вхідний текст	Вихідний код (індекс, символ)	Додано до словника
1	A	(0, A)	1 → "A"
2	B	(0, B)	2 → "B"
3	AB	(1, B)	3 → "AB"
4	ABA	(3, A)	4 → "ABA"
5	ABAB	(3, B)	5 → "ABAB"
6	ABABA	(4, B)	6 → "ABABA"

Таким чином, замість вихідного рядка "ABABABA" передається компактна послідовність (0, A), (0, B), (1, B), (3, A), (3, B), (4, B).

Порівняння LZ77 та LZ78 наводиться в наступній таблиці:

Табл. 1.2.

Порівняння LZ77 та LZ78.

Характеристика	LZ77	LZ78
Метод зберігання	Ковзне вікно	Явний словник
Формат вихідного коду	(зсув, довжина, символ)	(індекс, символ)
Чутливість до відстані між повтореннями	Так	Ні
Ефективність на великих файлах	Висока	Вища на початкових етапах

1.4.3 Модифікації словникових алгоритмів

Розглянемо модифікації алгоритму LZW та його похідні:

- LZW (Lempel-Ziv-Welch) – алгоритм LZW, названий на честь його розробників А. Лемпеля, Дж. Зіва та Террі Уелча, є однією з найбільш ефективних та універсальних технік стиснення даних без втрат. Його популярність пояснюється простотою реалізації та здатністю адаптуватися до різних типів даних. У середньому, LZW може зменшити розмір текстових файлів, виконуваних програм та подібних даних приблизно вдвічі. Найбільш ефективно він працює з надлишковими даними, наприклад, у файлах із

числовими таблицями або повторюваними рядками. Саме на цьому принципі базуються численні утиліти для зменшення обсягу збережених файлів, які часто рекламуються як засоби, що "подвоюють ємність жорсткого диска". Однак, ефективність LZW значною мірою залежить від довжини кодового слова. Якщо вона недостатня, алгоритм може досягати прийнятної продуктивності лише протягом короткого часу, після чого його переваги зменшуються. Зазвичай коефіцієнт стиснення для цього алгоритму становить приблизно 5:1 [22, 23];

- LZSS (Lempel-Ziv-Storer-Szymanski) – як вже зазначалося дещо раніше, такий алгоритм є однією з модифікацій LZ77, яка усуває необхідність включення окремого символу після кожного збігу. На відміну від кодування Хаффмана, яке зменшує середню кількість бітів на символ, LZSS використовує словниковий підхід, замінюючи рядки символів на посилання до їх попередніх входжень у тексті. Це дозволяє скоротити загальний розмір файлу, але лише в тих випадках, коли посилання займає менше місця, ніж оригінальний рядок. Попередник LZSS – LZ77 – не завжди був ефективним у цьому аспекті, оскільки іноді посилання займали більше місця, ніж самі символи [24];
- LZC (Lempel-Ziv Complexity) – це метод аналізу сигналів, який базується на принципах стиснення даних. На відміну від стандартних методів компресії, LZC використовується для вимірювання складності послідовностей даних, що робить його корисним у наукових дослідженнях. Цей метод виявився особливо ефективним для коротких наборів даних, що робить його популярним у медичних дослідженнях. Наприклад, він використовується для аналізу сигналів при хворобі Альцгеймера (AD), хворобі Паркінсона, коматозних станах та багатьох інших нейрофізіологічних розладах. Перед застосуванням LZC до сигналу необхідно провести його попередню обробку [25, 26];
- LZAP (Lempel-Ziv All Prefixes) – LZAP є вдосконаленою версією LZW, яка покращує його продуктивність шляхом додавання всіх можливих префіксів у словник. У класичному LZW, якщо алгоритм зчитує рядок "ABC", а потім "DEF", то у словник буде додано лише "ABCD". Натомість у LZAP буде додано

всі можливі префікси: "ABCD", "ABCDE", "ABCDEF". Це збільшує загальний коефіцієнт стиснення, особливо в тих випадках, коли дані містять багато повторюваних фрагментів [27].

- LZMW (Lempel-Ziv-Miller-Wegman) - LZMW є ще одним удосконаленням LZW, яке впроваджує дві ключові модифікації:
  - ефективне управління словником – замість того, щоб скидати словник при його заповненні, LZMW видаляє найменш вживані фрази. Це дозволяє зберігати лише найкорисніші записи, що покращує стиснення. Однак початкові 256 ASCII-символів ніколи не видаляються, щоб забезпечити коректну роботу алгоритму;
  - розширене додавання записів у словник – замість збереження лише найдовшого поточного збігу + першого нового символу, LZMW додає конкатенацію попереднього та поточного збігів.

  Це дозволяє алгоритму оперувати цілими словами або навіть фразами, що значно покращує ефективність стиснення при роботі з текстовими даними. Наприклад, якщо алгоритм стиснення стикається з фразою "THE CAT", він може зберігати "THE", "THE CAT", а не окремі символи чи коротші послідовності [28].

Кожен із варіантів має свої переваги:

- LZW – простота та універсальність;
- LZSS – зменшення зайвих посилань для покращення ефективності;
- LZC – використання для аналізу складності сигналів;
- LZAP – розширений словниковий підхід для кращого стиснення;
- LZMW – адаптивне оновлення словника для ефективнішої роботи.

Завдяки цим модифікаціям алгоритми LZ-сімейства залишаються основою сучасних форматів стиснення, таких як GIF, ZIP, 7z, TIFF, PDF та інших.

1.5 Аналіз алгоритмів заміщення сторінок у контексті компресії

В операційних системах, що використовують механізм підкачки для управління пам’яттю, необхідно визначати, яку сторінку замінити, коли в оперативну пам’ять потрібно завантажити нові дані. Це завдання вирішують алгоритми заміни сторінок.

Помилка сторінки виникає, коли процес звертається до сторінки, яка відображена у віртуальному просторі, але не знаходиться у фізичній пам’яті. Оскільки фізична пам’ять значно обмежена в порівнянні з віртуальною, такі помилки є неминучими. У випадку їх виникнення, система повинна звільнити місце для завантаження нової сторінки, використовуючи певний алгоритм заміщення.

Головна мета цих алгоритмів – мінімізувати кількість помилок сторінки, оскільки вони значно впливають на продуктивність системи.

Основні алгоритми заміни сторінок:

1. FIFO (First In First Out) – найпростіший метод, що працює за принципом "перший прийшов – перший пішов". Операційна система відстежує всі сторінки у черзі, де найстаріша сторінка знаходиться на початку. При необхідності заміни видаляється перша сторінка з черги. Недолік – можливе вилучення найчастіше використовуваної сторінки, що призводить до зниження продуктивності.

2. OPT (Optimal Page Replacement) – ідеальний алгоритм, що замінює ту сторінку, яка не знадобиться найдовше у майбутньому. Гарантує найменшу кількість помилок сторінки. Недолік – неможливість реалізації в реальному часі, оскільки неможливо передбачити майбутні звернення до пам’яті.

3. LRU (Least Recently Used) – вибирає сторінку, яка використовувалася найдовше тому. Ефективний у випадках, коли дані, використані нещодавно, з більшою ймовірністю будуть використані знову. Недолік – потребує зберігання історії використання сторінок, що може бути ресурсозатратно.

4. MRU (Most Recently Used) – замінюється остання використана сторінка. Ефективний у сценаріях, де сторінки, використані нещодавно, рідко потрібні знову.

Недолік – неефективний у стандартних випадках, оскільки часто видаляє найкорисніші дані.

5. NRU (Not Recently Used) – класифікує сторінки за доступністю та змінами у пам’яті. Випадково видаляє сторінку з найнижчим пріоритетом, де враховується, чи використовувалася сторінка останнім часом. Простий у реалізації, але не завжди забезпечує оптимальну продуктивність.

6. Second Chance – модифікація FIFO, яка унеможливлює видалення часто використовуваних сторінок. Перед видаленням сторінки перевіряється спеціальний біт R (означає, чи використовувалася сторінка). Якщо R = 1, сторінка отримує "другий шанс" – її біт очищується, а сторінка переміщується в кінець черги. Недолік – потребує додаткових перевірок, що може уповільнювати роботу.

7. Clock Page Replacement – удосконалена версія Second Chance, яка використовує кільцевий список сторінок. У списку працює "годинникова стрілка", що вказує на найстарішу сторінку. Якщо сторінка не використовувалася – вона замінюється, інакше отримує другий шанс. Перевага – ефективніше керування пам’яттю без зайвих переміщень сторінок.

8. NFU (Not Frequently Used) – кожна сторінка має лічильник звернень, який збільшується при кожному використанні. Видаляється сторінка з найменшим значенням лічильника. Недолік – може давати перевагу старішим сторінкам, що довго використовувалися, навіть якщо вони зараз не потрібні.

9. WSClock (Working Set Clock) – покращений варіант Clock Page Replacement, що враховує робочий набір процесу. Видаляються лише сторінки, які не використовувалися протягом певного часу. Добре балансує ефективність та продуктивність. Перевага – широко використовується в сучасних системах через простоту реалізації.

Алгоритми підкачки та компресія даних – у контексті стиснення даних, алгоритми заміни сторінок можуть впливати на ефективність компресії у багатьох випадках: FIFO, LRU, WSClock та Aging – можуть ефективно поєднуватися з методами стиснення, видаляючи найменш корисні дані; компресія з урахуванням частоти використання (LZW, Huffman, LZ77) може бути доповнена аналізом

робочого набору для визначення найкращих сторінок для заміни; стиснення сторінок перед їх заміною може зменшити кількість помилок сторінки, оскільки в пам’яті може залишатися більше корисних даних.

Отже, алгоритми заміни сторінок відіграють ключову роль в управлінні пам’яттю, особливо у віртуальних середовищах. FIFO – простий, але неефективний; OPT – ідеальний, але нереалістичний; LRU, WSClock, Aging – найпрактичніші в реальних системах. Методи стиснення можуть бути інтегровані з алгоритмами заміщення сторінок для покращення використання пам’яті та зменшення кількості помилок.

1.6 Формулювання завдань дослідження

Аналізуючи викладене вище, можна зробити висновок, що проблема стиснення даних є надзвичайно актуальною, оскільки обсяг інформації, що зберігається та обробляється, зростає в геометричній прогресії.

Одним із основних викликів при застосуванні алгоритму LZW для стиснення великих файлів є ефективне керування словником. Часто розмір таблиці рядків, яка використовується у процесі кодування, перевищує доступну пам’ять, що може впливати на швидкість і якість стиснення.

У цьому дослідженні пропонується новий метод побудови словника, який має на меті оптимізувати процес стиснення, покращуючи його продуктивність та ефективність.

Об’єктом дослідження є дані, що потребують стиснення без втрат.

Метою дослідження є розробка та аналіз методу побудови словника, що дозволить оптимізувати процес стиснення даних.

Для досягнення поставленої мети необхідно виконати такі завдання:

−провести аналіз існуючих методів стиснення даних, розглянувши їхні переваги та недоліки;

− дослідити алгоритми сімейства LZ, включаючи їхні модифікації та особливості реалізації.

− проаналізувати існуючі методи побудови словника, що використовуються у алгоритмах словникового стиснення.

− розробити ефективний алгоритм побудови словника, який дозволить зменшити використання пам’яті та прискорити процес стиснення.

− реалізувати модифікований алгоритм LZW, що інтегруватиме розроблений метод словникового кодування.

Результати цього дослідження дозволять підвищити ефективність стиснення без втрат, що є важливим для обробки великих масивів даних та збереження інформаційних ресурсів.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ ХАРАКТЕРИСТИК АЛГОРИТМІВ АРХІВАЦІЇ

2.1 Огляд та принцип роботи алгоритму LZW

Алгоритм LZW (Lempel-Ziv-Welch) є одним із найпоширеніших методів безвтратного стиснення даних, що використовується в багатьох форматах, таких як GIF, TIFF, ZIP тощо. Він є удосконаленням алгоритмів LZ77 і LZ78, забезпечуючи ефективне стиснення без необхідності передавати словник разом із закодованими даними.

Принцип роботи алгоритму LZW – стиснення відбувається шляхом поступового формування словника повторюваних рядків у вихідних даних. Процес кодування виглядає так:

Послідовно зчитуються символи вхідного потоку – якщо отримана послідовність вже існує у словнику, зчитується наступний символ.

Якщо рядок відсутній у таблиці, то: у вихідний потік записується код попередньої знайденої послідовності; нова послідовність додається до таблиці зі своїм унікальним кодом; пошук розпочинається з початку.

Наприклад, під час стискання текстових файлів початковий словник містить 256 рядків (від 0 до 255 – символи ASCII). Якщо використовується 10-бітове кодування, то нові рядки отримують коди в діапазоні 256÷1023.

Алгоритм декодування працює аналогічно: на основі вхідного закодованого потоку він покроково відтворює таблицю перетворень.

Оскільки кожен новий код однозначно додається до словника, процес декодування є унікальним та зворотним.

Якщо отриманий код уже міститься у словнику – він просто відтворюється, а якщо це новий код, то формується відповідний рядок.

Таким чином, кожен унікальний рядок у словнику існує лише в одному екземплярі, що забезпечує оптимізацію пам’яті та зменшення дублюваних даних.

Розглянемо LZW, який реалізує процес стиснення та декодування на прикладі роботи з зображенням:

1) Формування початкового словника – на початку алгоритм створює словник з окремих символів.

У стандартному ASCII-кодуванні існує 256 символів, тому кожен символ кодується 8-бітовим числом.

Якщо у вихідному файлі міститься менше символів, можна використовувати меншу кількість біт (наприклад, 3 біти для 8 комбінацій).

2) Динамічне розширення словника.

Розмір коду поступово збільшується: після 256-го запису використовуються 9-бітові групи; після 512-го запису – 10-бітові групи. і так далі, поки дозволяє доступна пам’ять.

3) Покрокове стиснення: при обробці кожного нового рядка перевіряється його наявність у словнику; якщо рядок вже існує, зчитується наступний символ; якщо рядок відсутній, він додається до словника та кодується у вихідний потік.

4) Декодування – використовуючи лише отриманий закодований потік, алгоритм відтворює початкові дані, поступово відновлюючи словник.

Завдяки послідовному формуванню унікальних кодів стиснений текст однозначно декодується без втрат.

Отже, алгоритм LZW є ефективним методом стиснення без втрат, що забезпечує високий ступінь компресії за рахунок динамічного створення словника. Завдяки унікальному механізму кодування та декодування, він знаходить широке застосування в текстових, графічних та інших форматах даних, дозволяючи оптимізувати використання пам’яті та зменшити обсяг збережених файлів.

2.1.1 Процес кодування інформації

На відміну від кодування Хаффмана, кодування LZW встановлює кодові

слова постійної довжини на серії вихідних символів змінної довжини. LZW створює «словник», який містить слова або частини слів даних. Коли дані потрібно розпакувати, вони повинні звернутися до словника, який, у свою чергу, представляє код LZW для цього слова.

Розглянути логіку кодування LZW [29] можна на наступному рисунку:

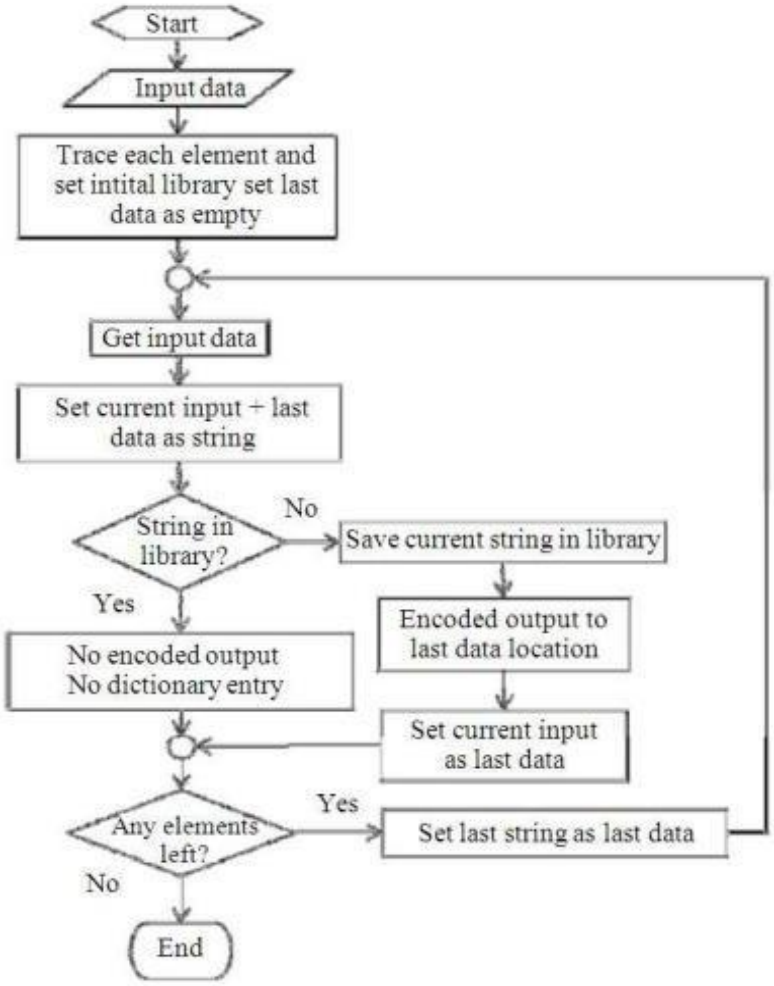

Рис. 2.1 Блок-схема кодування.

Нехай стискаємо послідовність «abacabadabacabae». Дана процедура здійснюється за такою схемою:

Крок 1. Тоді, згідно з викладеним вище алгоритмом, додамо до порожнього рядка «a» і перевіримо, чи є рядок «a» у таблиці. Оскільки під час ініціалізації занесли до таблиці всі рядки з одного символу, то рядок «a» є у таблиці.

Крок 2. Далі читаємо наступний символ «b» із вхідного потоку і перевіряємо, чи є рядок «ab» у таблиці. Такого рядка в таблиці поки що немає. Додаємо в таблицю «5» «ab». У потік: «0».

Крок 3. «ba» – ні. У таблицю: «6» «ba». У потік: «1».

Крок 4. «ac» – ні. У таблицю: «7» «ac». У потік: «0».

Крок 5. «ca» – ні. У таблицю: «8» «ca». У потік: «2».

Крок 6. «ab» – є в таблиці; «aba» – ні. У таблицю: «9» «aba». У потік: «5».

Крок 7. «ad» – ні. У таблицю: «10» «ad». У потік: «0».

Крок 8. «da» – ні. У таблицю: «11» «da». У потік: «3».

Крок 9. «aba» – є в таблиці; «abac» – ні. У таблицю: «12» «abac». У потік: «9».

Крок 10. «ca» – є в таблиці; «cab» – ні. У таблицю: «13» «cab». У потік: «8».

Крок 11. «ba» – є в таблиці; «bae» – ні. У таблицю: «14» «bae». У потік: «6».

Крок 12. І, нарешті, останній рядок«e», за нею йде кінець повідомлення, тому просто виводимо в потік «4».

Отже, отримуємо закодоване повідомлення «0 1 0 2 5 0 3 9 8 6 4», що на 11 біт коротше. Для наочності вище наведені кроки представлені у таблиці 2.1.

Табл. 2.1

Кодування даних.

Поточний рядок	Поточний символ	Наступний символ	Вихід		Словник
			Код	Біти	
1	2	3	4	5	6
Ab	a	b	0	000	5:ab
ba	b	a	1	001	6:ba
ac	a	c	0	000	7:ac
ca	c	a	2	010	8:ca
ab	a	b	-	-	-:-
aba	b	a	5	101	9:aba
ad	a	d	0	000	10:ad

da	d	a	3	011	11:da
ab	a	b	-	-	-:-
aba	b	a	-	-	-:-

Продовження таблиці 2.1

abac	a	c	9	1001	12:abac
ca	c	a	-	-	-:-
cab	a	b	8	1000	13:cab
ba	b	a	-	-	-:-
bae	a	e	6	0110	14:bae
e	e	-	4	0100	-:-

2.1.2 Декодування та відновлення стиснених даних

Блок-схему декодування алгоритму LZW [29] представлено на рисунку 2.2:

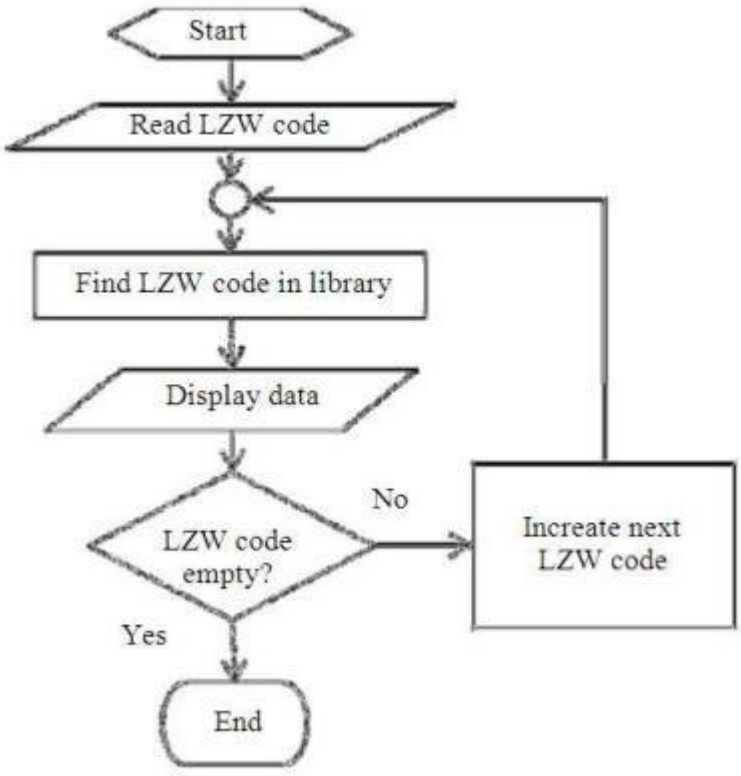

Рис. 2.2. Блок-схема декодування.

Особливість LZW полягає в тому, що для декомпресії нам не потрібно зберігати таблицю рядків у файл для розпакування. Алгоритм побудований таким чином, що можна відновити таблицю рядків, користуючись лише потоком кодів.

Тепер уявімо, що отримали закодоване повідомлення, наведене вище, і нам потрібно його декодувати. Перш за все нам потрібно знати початковий словник, а наступні записи словника можна реконструювати вже на ходу, оскільки вони є просто конкатенацією попередніх записів (табл. 2.2).

Табл. 2.2.

Закодовані дані алгоритмом LZW.

Данні		Вихід	Новий запис	
Біти	Код		Повна	Часткова
000	0	a	-:-	5:a?
001	1	b	5:ab	6:b?
000	0	a	6:ba	7:a?
010	2	c	7:ac	8:c?
101	5	ab	8:ca	9:ab?
000	0	a	9:aba	10:a?
011	3	d	10:ad	11:d?
1001	9	aba	11:da	12:aba?
1000	8	ca	12:abac	13:ca?
0110	6	ba	13:cab	14:ba?
0100	4	e	14:bae	-:-

Як зазначалося раніше, алгоритм LZW дозволяє відновити початкові дані без потреби у збереженні словника, оскільки таблиця формується динамічно під час декодування.

Розглянемо покрокове декодування повідомлення «0 1 0 2 5 0 3 9 8 6 4». Для декодування необхідно слідувати таким правилам:

- використовувати ініціалізований словник (256 можливих символів у випадку ASCII, або менше, якщо нам відомий початковий набір символів);
- поетапно відновлювати слова на основі отриманих кодів;
- формувати нові словникові записи за аналогією з кодуванням.

Розглянемо процес у вигляді таблиці:

Табл. 2.3.

Декодування даних алгоритмом LZW.

Код	Декодований вихід	Новий запис у словнику
0	a	-
1	b	5: ab
0	a	6: ba
2	c	7: ac
5	ab	8: ca
0	a	9: aba
3	d	10: ad
9	aba	11: da
8	ca	12: abac
6	ba	13: cab
4	e	14: bae

Відновлений рядок після декодування: "abacabadabacabae".

Отже, алгоритм LZW дозволяє ефективно стискати текстові та інші послідовності даних за допомогою динамічного формування словника під час кодування. Основна особливість методу – відсутність необхідності збереження словника у файлі, оскільки декодування виконується лише на основі отриманих кодів. При декодуванні нові послідовності слів формуються аналогічно процесу стиснення, що забезпечує повне відновлення вихідних даних.

2.2 Експериментальне порівняння різних методів компресії

Розглянемо порівнянні продуктивності різних методів статистичного

стиснення (кодування довжини серії, кодування Шеннона-Фано, кодування Хаффмана, адаптивне кодування Хаффмана та арифметичне кодування), алгоритмів сімейства LZ77 (LZSS, LZH і LZB) і алгоритми сімейства LZ78 (LZW і LZFG). Дослідження, проведені для оцінки ефективності будь-якого алгоритму стиснення, проводилися за двома важливими параметрами [30]. Один – це ступінь досягнутого стиснення, а інший – час, який витрачають алгоритми кодування та декодування. Кілька разів було перевірено практичну ефективність вищезазначених методів на файлах Кентерберійського корпусу.

Відповідно до результатів, наведених у таблиці 2.3, для RLE [31, 32], для більшості перевірених файлів цей алгоритм генерує стислі файли, більші за оригінальні файли. Це пов'язано з меншою кількістю запусків у вихідному файлі. Для інших файлів ступінь стиснення менший. Середній BPC, отриманий цим алгоритмом, становить 7,93. Таким чином, зроблено висновок, що цей алгоритм може зменшити в середньому близько 4% вихідного файлу.

Це не можна вважати значним покращенням. BPC і ступінь стиснення, досягнутий для алгоритму Шеннона-Фано [33], представлені в таблиці 2.3. Ступінь стиснення для алгоритму Шеннона-Фано знаходиться в діапазоні від 0,60 до 0,82, а середній BPC становить 5,50. Ступінь стиснення для алгоритму кодування Хаффмана [34] знаходиться в діапазоні від 0,57 до 0,81. Коефіцієнт стиснення, отриманий цим алгоритмом, кращий порівняно з алгоритмом Шеннона-Фано, а середнє значення бітів на символ становить 5,27.

Арифметичне кодування в більшості аспектів перевершує більш відомий метод Хаффмана [35]. Він представляє інформацію принаймні так само компактно, іноді значно більше. Його продуктивність оптимальна без необхідності блокування вхідних даних [36]. Це заохочує чітке розмежування між моделлю представлення даних і кодуванням інформації щодо цієї моделі. Він легко вміщує адаптивні моделі та є ефективним з точки зору обчислень. Проте багато авторів і практиків, здається, не знають про цю техніку. Дійсно, існує широко поширена думка, що кодування Хаффмана неможливо вдосконалити.

Загальна продуктивність з точки зору середнього BPC згаданих вище

методів статистичного кодування [10] показана на рисунку 2.3.

Табл. 2.3.

Порівняння BPC для різних методів статистичного стиснення.

№	Назва файлу	Розмір файлу	LZW	LZ77	LZ78	Huffman coding	Arithmetic coding
1	bib	111261	3,84	3,75	3,95	5,26	5,23
2	book1	768771	4,03	4,57	3,92	4,57	4,55
3	book2	610856	4,52	3,93	3,81	4,83	4,78
4	news	377109	4,92	4,37	4,33	5,24	5,19
5	obj1	21504	6,3	5,41	5,58	6,45	5,97
6	obj2	246814	9,81	3,81	4,68	6,33	6,07
7	paper1	53161	4,58	3,94	4,5	5,09	4,98
8	paper2	82199	4,02	4,1	4,24	4,68	4,63
9	progc	39611	4,88	3,84	4,6	5,33	5,23
10	progl	71646	3,89	2,9	3,77	4,85	4,76
11	progp	49379	3,73	2,93	3,84	4,97	4,89
12	trans	93695	4,24	2,98	3,92	5,61	5,49

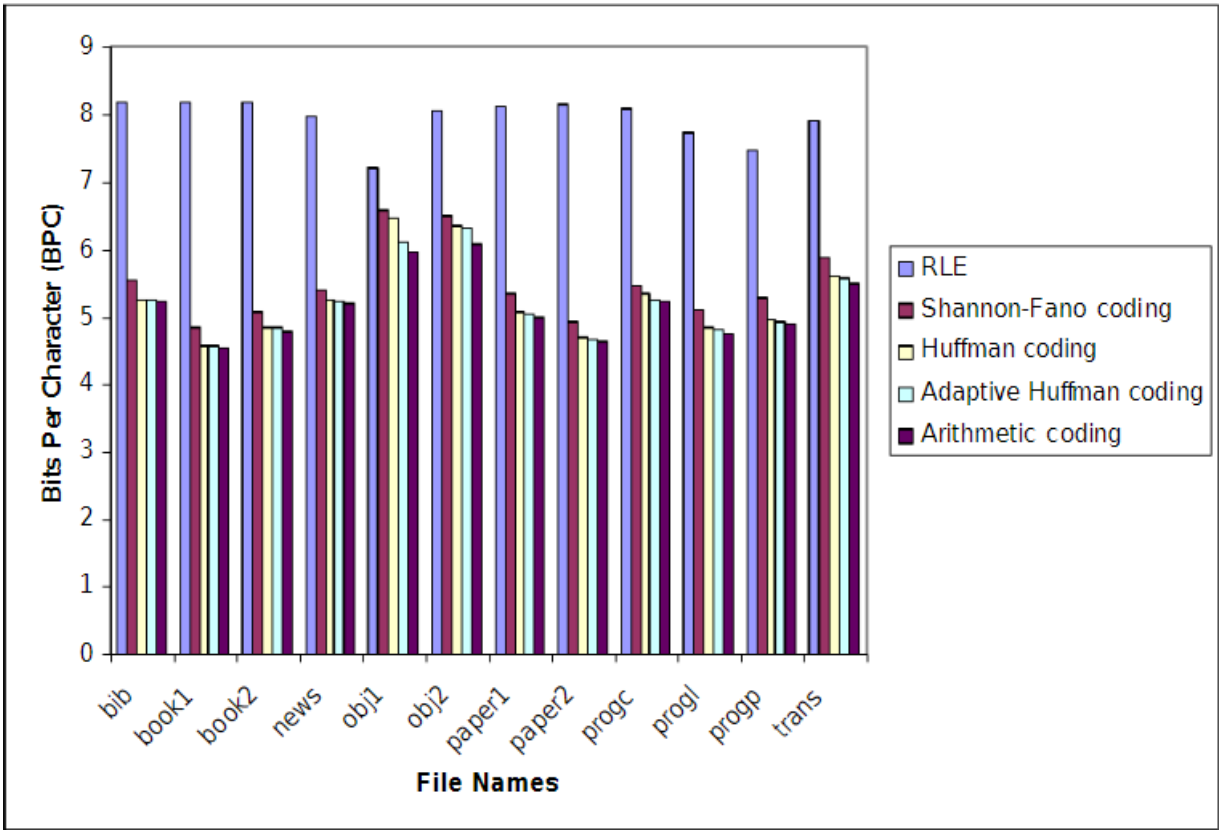

Рис. 2.3. Графік порівняння методів стиснення.

Основним параметром для сімейства LZ77 є розмір вікна в тексті. Стиснення найкраще, якщо вікно є якомога більшим, але не більшим за текст, загалом. Тим не менш, більші вікна дають меншу віддачу. Вікно довжиною всього 8000 символів працюватиме набагато краще та дасть майже такий же результат, як і вікна, отримані з більших вікон. Ще один параметр, який обмежує кількість символів, необхідний для деяких алгоритмів сімейства LZ [37]. Зазвичай обмеження приблизно в 16 може працювати добре [38]. Для LZ77, LZSS і LZB пам’ять (символів у вікні) вважалася рівною 8 КБ, а для LZH – 16 КБ. На рис. 2.4 показано порівняння ступенів стиснення для різних варіантів LZ77 [10].

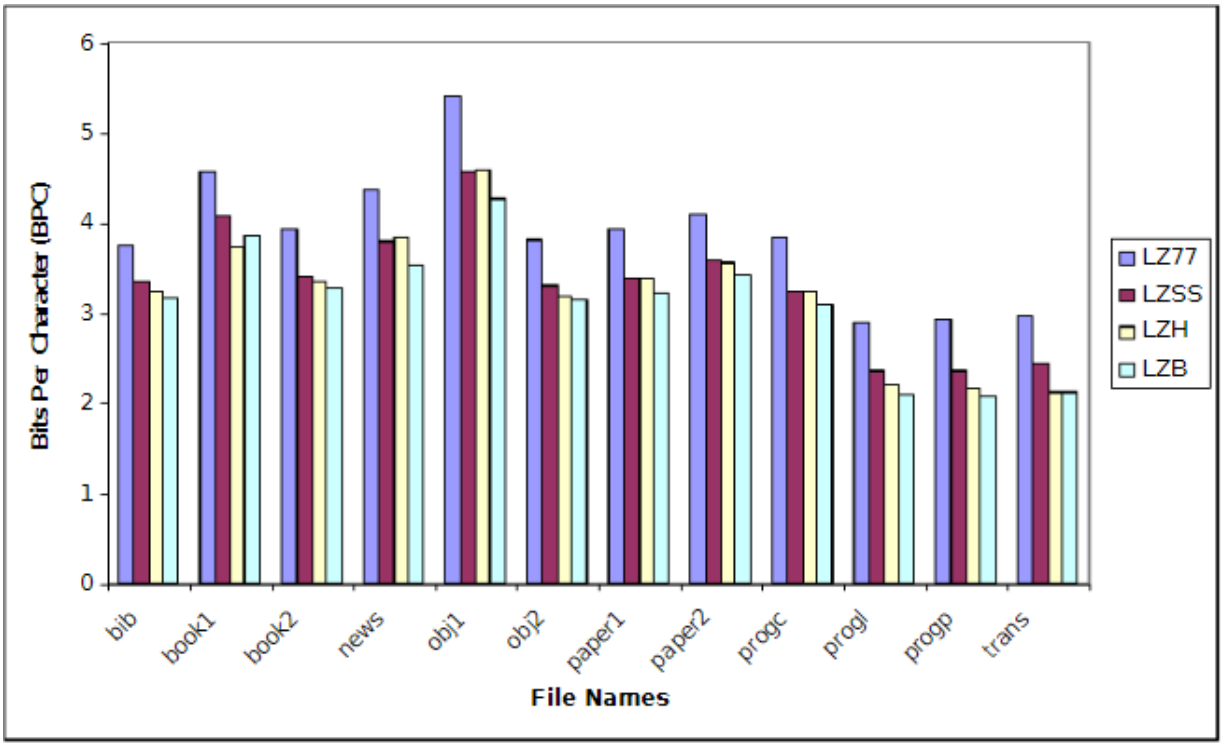

Рис. 2.4. Діаграма рівню стиснення для сімейства LZ77.

З рисунку 2.4 видно, що більшість файлів ASCII стискаються лише до половини початкового розміру, і в кожному файлі рівень стиснення є незмінним. Метод LZW, не маючи меж, приймає фрази, тому стиснення розширює файл «obj2» на 25%, що вважається недоліком цього підходу.

Крім того, з рисунка 2.5 очевидно, що продуктивність LZFG є найкращою серед цих методів, даючи середній BPC 2,89, що є дійсно значним. На рисунку 2.5 показано порівняння ступенів стиснення для сімейства LZ78 [19].

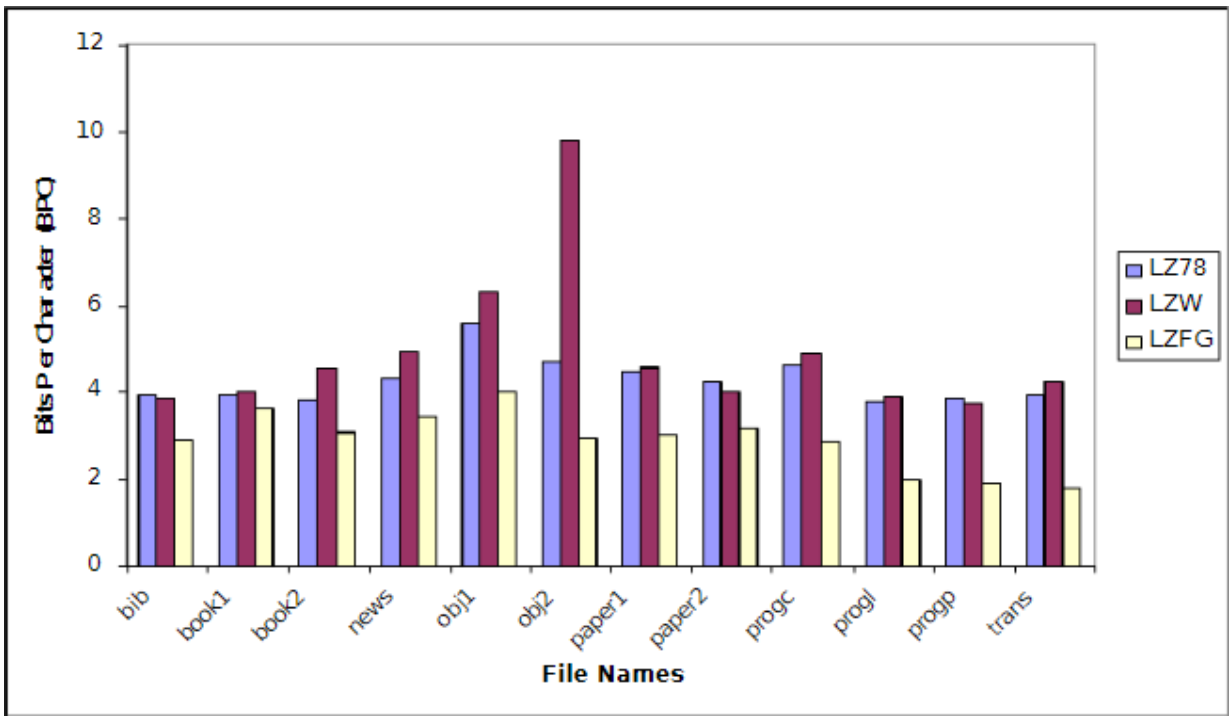

Рис. 2.5. Діаграма, що показує рівень стиснення для сімейства LZ78.

Проаналізувавши наведені результати та порівняльні таблиці, можна зробити кілька важливих висновків щодо ефективності різних алгоритмів стиснення.

1. Статистичні методи стиснення:

- RLE (Run-Length Encoding): ефективний лише для даних, що містять довгі послідовності повторюваних символів. Для більшості файлів він не дає суттєвого стиснення і навіть може збільшувати розмір файлу;
- кодування Шеннона-Фано: забезпечує коефіцієнт стиснення від 0,60 до 0,82 із середнім значенням BPC = 5,50;
- кодування Хаффмана: трохи ефективніше, ніж метод Шеннона-Фано, зі значенням BPC = 5,27;
- арифметичне кодування: забезпечує кращий рівень стиснення, ніж кодування Хаффмана, оскільки працює з дробовими значеннями ймовірностей символів, що дає змогу отримати більш оптимальне представлення інформації.

2. Методи сімейства Lempel-Ziv

- LZ77 та його варіанти: як видно з таблиці 2.3 та рисунку 2.4, продуктивність алгоритмів LZ77 (LZSS, LZH, LZB) суттєво залежить від розміру вікна та

довжини пошукового буфера. Вікно в 8 КБ дає оптимальні результати без значного навантаження на пам’ять. Загалом, алгоритми LZ77 можуть стискати текстові файли до половини початкового розміру, що робить їх ефективними для роботи з великими текстовими масивами. LZ77 і його варіанти добре підходять для великих файлів завдяки оптимальному використанню буфера пошуку.

- LZ78 та його варіанти: метод LZW демонструє хороші результати, однак може призводити до розширення файлів (наприклад, для "obj2" у таблиці 2.3). Серед алгоритмів сімейства LZ78 LZFG показує найкращу продуктивність із середнім BPC = 2,89, що значно перевершує інші варіанти (див. рисунок 2.5). LZFG є найбільш оптимальним варіантом серед LZ-методів, оскільки використовує ретельно відібрані коди для представлення покажчиків.

2.3 Обґрунтування та розробка покращеного алгоритму

Одним із ключових недоліків класичного алгоритму LZW є постійне зростання таблиці рядків (словника) під час обробки даних. У міру аналізу більшої кількості символів словник стає все більшим, що призводить до проблеми управління обмеженою пам'яттю комп'ютера. Цей недолік особливо виражений для великих файлів, де розмір словника може перевищити доступний обсяг пам'яті або значно ускладнити процес стиснення. Для вирішення цієї проблеми запропоновано модифікований алгоритм, який враховує обмеження пам'яті та оптимізує процес формування словника.

1 Аналіз існуючих методів управління словником

Існує три основних підходи до управління зростаючою таблицею рядків: Table Freezing, Table Flushing та Table Pruning. Кожен з них має свої переваги та недоліки:

1. Table Freezing — метод, за яким словник "заморожується" після досягнення певного розміру. Після цього стиснення продовжується на основі замороженого словника. Цей підхід простий у реалізації, але неефективний для

великих файлів, оскільки не враховує нові частотні послідовності символів, які можуть з'явитися пізніше.

2. Table Flushing — метод, за яким словник періодично очищається, коли поточний коефіцієнт стиснення падає нижче заданого порогу. Цей підхід дозволяє позбутися рідко використовуваних записів, але одночасно видаляє також часто використовувані записи, що може негативно впливати на ефективність стиснення.

3. Table Pruning — метод, за яким рідко використовувані записи замінюються новими. Проблема полягає у нетривіальності вибору таких записів, що потребує додаткових обчислювальних ресурсів.

Запропонований алгоритм базується на концепції Table Pruning, але використовує оптимізовані стратегії вибору записів для заміни, що дозволяє краще адаптуватися до характеристик вхідних даних.

2. Запропонований алгоритм є модифікацією класичного LZW, яка включає два ключових покращення:

- використання коду змінної довжини (8-12 біт) - для ефективного використання пам'яті та зменшення розміру вихідних даних запропоновано використовувати код змінної довжини. Початковий розмір коду дорівнює 8 бітам, і він збільшується до 12 бітів у міру зростання словника. Це дозволяє зберегти баланс між розміром словника та ефективністю стиснення.

- оптимізація процесу формування словника - для управління словником запропоновано комбінацію двох стратегій вибору записів для заміни: Least Recently Used (LRU) та Aging Replacement.

1) Стратегія LRU - у стратегії LRU вибирається запис, до якого не було доступу протягом найдовшого часу. Реалізація цієї стратегії здійснюється за допомогою самоорганізованого списку, який містить індекси всіх записів словника. Під час стиснення кожного разу, коли до запису здійснюється доступ, його індекс переміщується на початок списку. Якщо потрібен запис для заміни, вибирається запис із кінця списку.

2) Стратегія Aging Replacement - у стратегії Aging Replacement кожному запису словника присвоюється параметр TTL (Time To Live), який ініціалізується

певним значенням. Періодично TTL знижується. Якщо TTL стає рівним нулю, запис видаляється зі словника. При доступі до запису його TTL скидається до значення `(поточне значення / 2 + початкове значення)`. Це забезпечує більший вплив на більш "свіжі" доступи до запису порівняно з раніше здійсненими.

Коли потрібен запис для заміни, вибирається запис із найменшим TTL або невикористаний запис.

3. Процес кодування та декодування

1) Процес кодування

1. Алгоритм починає з ініціалізації словника початковими символами вхідного алфавіту.

2. Вказівник `prevDictionary` встановлюється на порожнє місце.

3. Алгоритм шукає найдовшу послідовність символів, яка є у словнику. Якщо така послідовність знайдена, її індекс повертається як результат стиснення.

4. Якщо послідовність не знайдена, алгоритм:

- Повертає індекс попередньої знайденої послідовності.
- Додає нову послідовність до словника.
- Оновлює вказівник `prevDictionary`.

5. Процес повторюється, доки не буде закодовано всю вхідну послідовність.

2) Процес декодування

Декодування відтворює словник на основі індексів, отриманих під час кодування. Кожен індекс вказує на конкретну послідовність у словнику. Процес декодування аналогічний процесу кодування, за винятком того, що він зчитує індекси та формує вихідну послідовність символів.

4. Блок-схеми алгоритму

На рисунках 2.6 і 2.7 показано блок-схеми процесів стиснення та декомпресії запропонованого алгоритму. Блок-схеми демонструють основні етапи формування словника, кодування та декодування.

5 Переваги запропонованого алгоритму

1) Ефективне використання пам'яті:

Через використання стратегій LRU та Aging Replacement словник залишається компактним, що дозволяє ефективно працювати з великими файлами.

2) Покращена адаптація до вхідних даних:

Запропонований алгоритм краще адаптується до зміни частоти появи символів у вхідних даних, завдяки стратегіям вибору записів для заміни.

3) Гнучкість:

Використання коду змінної довжини дозволяє гнучко налаштовувати алгоритм під різні типи даних.

Отже, запропонований алгоритм є покращеною версією LZW, яка враховує обмеження пам'яті та оптимізує процес формування словника. Використання стратегій LRU та Aging Replacement дозволяє ефективно керувати словником, зберігаючи лише актуальні записи. Це робить алгоритм придатним для стиснення великих файлів без значного зниження ефективності.

Запропонований алгоритм демонструє значні переваги перед класичним LZW, особливо у випадках, коли обмеження пам'яті є критичним фактором.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

3.1 Вибір технологій для розробки програми-архіватора

3.1.1 Використання мови програмування Java

Мова програмування Java була розроблена компанією Sun Microsystems під керівництвом Джеймса Гослінга та вперше випущена у 1995 році як ключова складова платформи Java (Java 1.0 J2SE). З часом ця технологія отримала широке поширення та розвиток, що призвело до створення декількох спеціалізованих конфігурацій для різних типів застосунків. Наприклад:

- J2EE (тепер Java EE) – для корпоративних додатків;
- J2ME (тепер Java ME) – для мобільних пристроїв.

Нова номенклатура розподілила Java на Java SE (Standard Edition), Java EE (Enterprise Edition) та Java ME (Micro Edition). Однією з ключових особливостей мови є її гасло "Write Once, Run Anywhere", що означає можливість запуску написаного коду на будь-якій платформі, що підтримує Java Virtual Machine (JVM).

Основні переваги Java:

- об’єктно-орієнтованість – у Java все є об’єктом, що сприяє розширюваності та повторному використанню коду;
- незалежність від платформи – після компіляції Java-код перетворюється не у машинний код конкретної платформи, а у байт-код, який виконується на будь-якій платформі за допомогою JVM;
- простота у вивченні – мова має зрозумілу структуру, а знання принципів об’єктно-орієнтованого програмування (ООП) значно полегшує її освоєння;
- безпека – Java має вбудовані механізми захисту, які допомагають створювати захищені від вірусів та несанкціонованого доступу програми;
- архітектурна нейтральність – скомпільований Java-код можна виконувати на будь-якому процесорі, що підтримує Java Runtime Environment;
- портативність – код, написаний на Java, легко переноситься між платформами завдяки відсутності жорсткої прив’язки до конкретної операційної системи чи апаратного забезпечення;
- надійність – Java використовує механізми перевірки коду як під час компіляції, так і під час виконання, що зменшує ймовірність помилок;
- багатопоточність – дозволяє ефективно використовувати ресурси процесора для одночасного виконання кількох завдань;
- інтерпретація – байт-код Java виконується у JVM на льоту, що прискорює процес розробки та тестування;
- висока продуктивність – завдяки використанню Just-In-Time (JIT) компіляції, програми на Java працюють швидко та ефективно.

Завдяки всім цим характеристикам Java є ідеальним вибором для реалізації програмного забезпечення, зокрема архіваторів даних, які вимагають високої продуктивності, надійності та незалежності від операційної системи.

В якості середовища розробки було вибрано Eclipse IDE (рис. 3.1), яке вміщує в собі як простоту так і гнучкість, не поступаючись своїм конкурентам (рис. 3.2) [43, 44]. Немаловажним залишається і те, що Eclipse дозволяє працювати з системою керування версіями Git [45, 46].

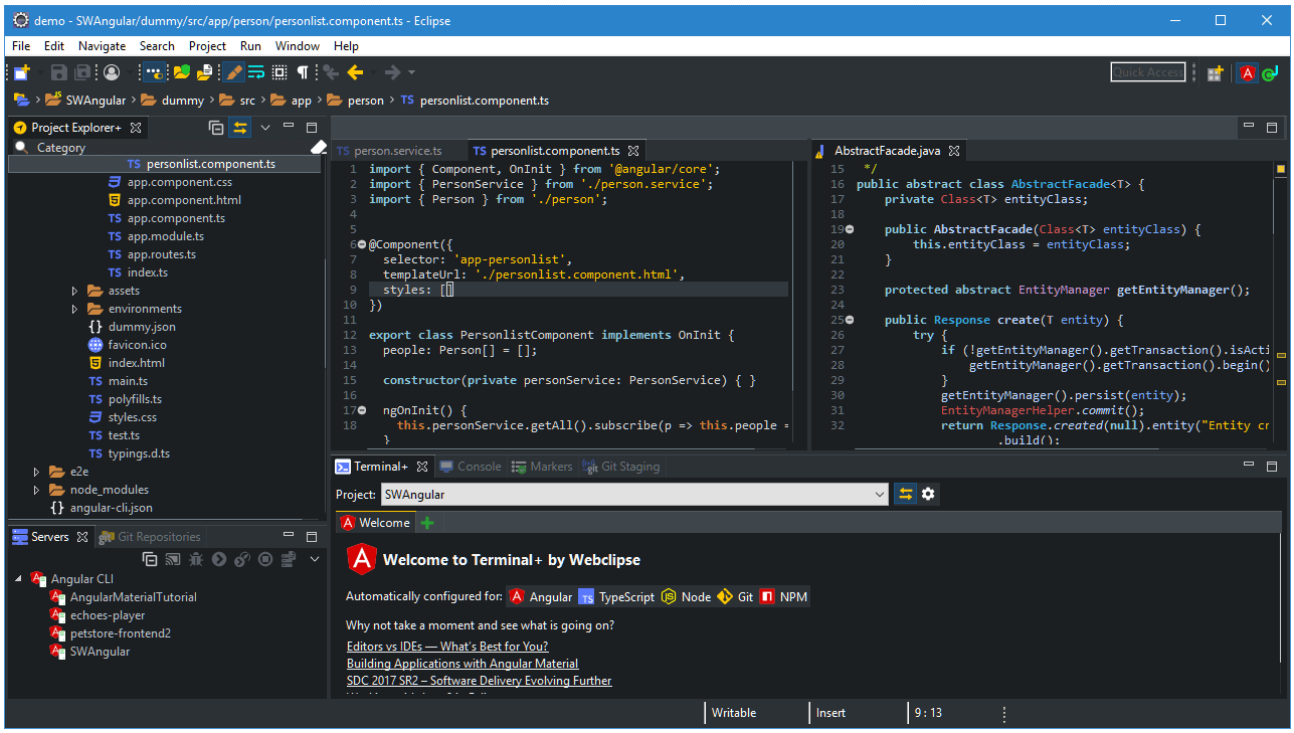

Рис. 3.1. Вікно IDE Eclipse.

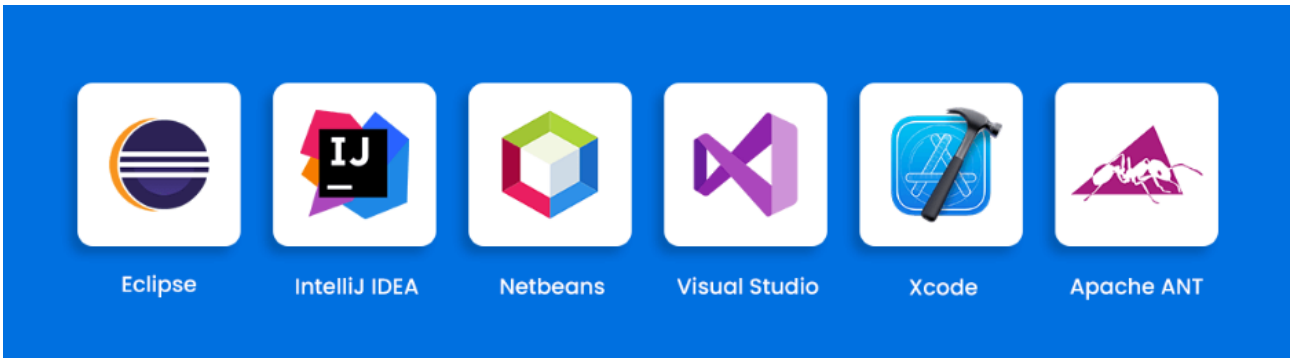

Рис. 3.2. Популярні середовища розробки Java.

3.1.4 Інструменти для автоматизації збирання проєкту Maven

Maven – це потужний інструмент для управління життєвим циклом проєктів, заснований на концепції об'єктної моделі проєкту (POM – Project Object Model). Він використовується для автоматизації процесів збірки, управління залежностями та створення документації.

На відміну від інших інструментів збірки, таких як Ant, Maven надає більш високий рівень автоматизації, дозволяючи спростити розробку, забезпечити стандартизацію та зменшити кількість ручних налаштувань. Принцип роботи Maven представлений на рисунку 3.3.

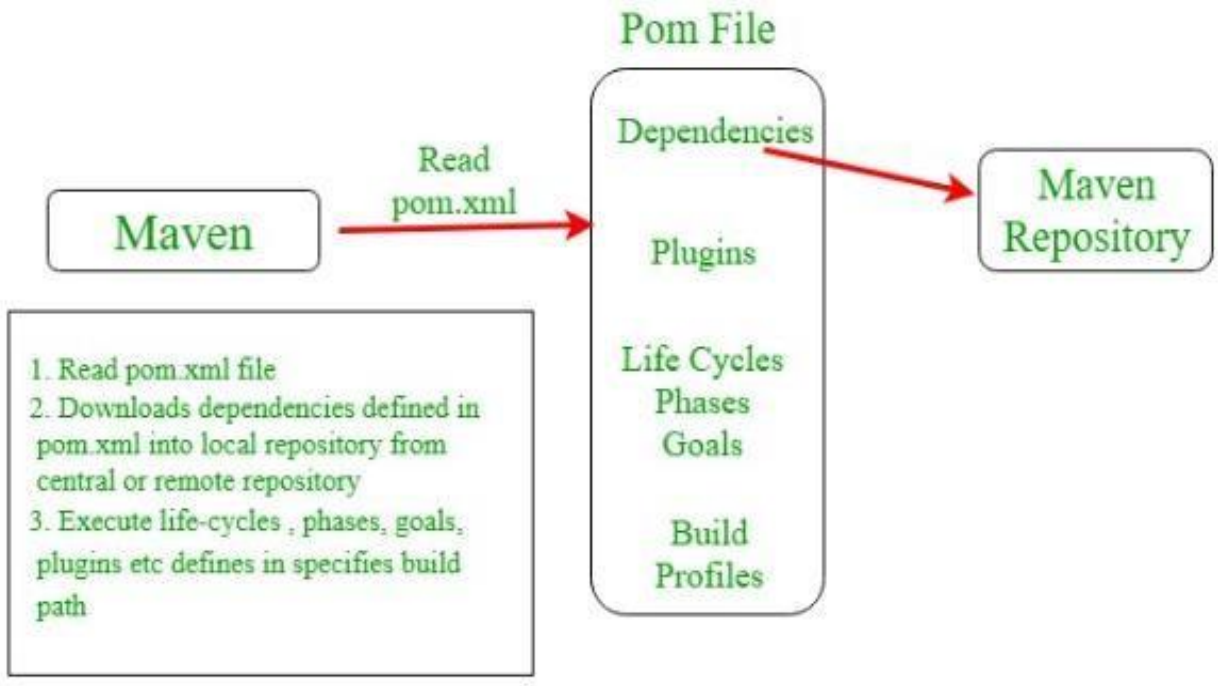

Рис. 3.3. Принцип роботи Maven.

Maven застосовується для створення, управління та розгортання проєктів, написаних мовою Java, але також підтримує й інші мови та технології. Завдяки Maven розробники отримують змогу легко структурувати код, керувати залежностями та використовувати централізоване сховище бібліотек [48, 49].

Основні можливості Maven:

1) автоматичне створення проєкту – Maven дозволяє швидко ініціалізувати новий проєкт з використанням готових архетипів, що спрощує старт розробки;

2) керування залежностями – один із найважливіших аспектів Maven – автоматичне завантаження бібліотек (JAR-файлів) та їх оновлення. Усі залежності зберігаються в централізованому репозиторії, а Maven автоматично завантажує необхідні версії бібліотек у локальне сховище розробника;

3) генерація документації – Maven може автоматично створювати документацію проєкту, включаючи журнал змін, список залежностей, звіти про тестування тощо.

4) створення різних типів вихідних файлів – Maven підтримує створення JAR, WAR, EAR та інших форматів архівів без необхідності ручного написання сценаріїв збірки.

5) інтеграція із системами керування версіями – Maven легко інтегрується з Git, SVN та іншими системами контролю версій, що дозволяє автоматизувати процеси CI/CD (безперервної інтеграції та розгортання).

Враховуючи вище сказане, Maven є потужним інструментом, який дозволяє значно спрощувати процес розробки Java-застосунків, особливо архіваторів, які потребують використання спеціалізованих бібліотек.

3.2 Програмна реалізація архіватора

Створення архіватора передбачало реалізацію наступного функціоналу:

- графічний інтерфейс з підтримкою багатопоточності;
- архівація/розархівація;
- захист паролем;
- перевірка на цілісність даних;
- взаємодія з системним файловим менеджером;
- логування інформації щодо перебігу виконуваних операцій.

3.2.1. Графічний інтерфейс з підтримкою багатопоточності

Реалізація графічного інтерфейсу (рис. 3.4) передбачала використання наступних фреймворків:

- javax.swing.*: основний пакет для створення графічних інтерфейсів користувача (GUI) в Java. Він надає більш сучасні та гнучкі компоненти (наприклад, JFrame, JPanel, JButton, JFileChooser, JProgressBar, JTextArea, JScrollPane, JRadioButton, JComboBox, JCheckBox, JPasswordField, JOptionPane), ніж AWT. Вся візуальна частина програми-архіватора (вікно, кнопки, поля вводу/виводу, прогрес-бар) побудована на компонентах Swing;
- java.awt.*: старіший, але фундаментальний пакет для GUI в Java. AWT (Abstract Window Toolkit) надає базові графічні та віконні компоненти, а також класи для роботи з графікою, кольорами, шрифтами та обробкою подій. Хоча Swing переважно використовує свої власні компоненти, він побудований поверх AWT. В нашому випадку, java.awt.* використовується для таких класів, як:
    - java.awt.Desktop: для відкриття файлів та тек системними засобами;
    - java.awt.event.ActionEvent: для обробки подій від кнопок;
    - java.awt.Component: базовий клас для всіх візуальних компонентів;
    - java.awt.Dimension, java.awt.Insets, java.awt.GridBagConstraints, java.awt.GridBagLayout, java.awt.FlowLayout, java.awt.BorderLayout, java.awt.GridLayout: для управління розміщенням компонентів на панелях.

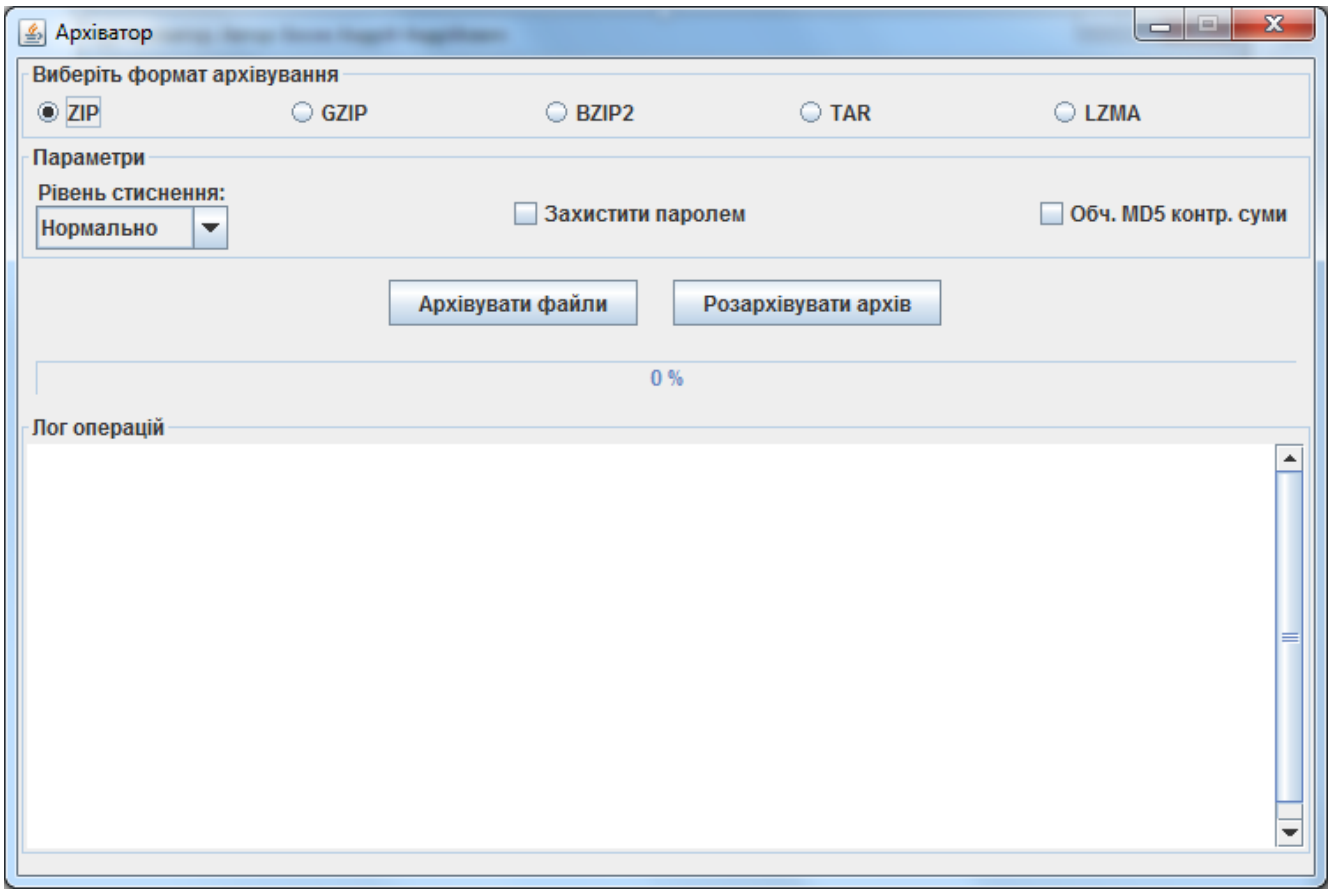

Рис. 3.4. Інтерфейс програми-архіватора.

Щоб забезпечити плавну роботу графічного інтерфейсу користувача (GUI) під час виконання тривалих операцій, таких як архівація або розархівування файлів, в проекті використовувалися засоби багатопоточності.

Річ у тім, що всі GUI-фреймворки, включаючи Swing, працюють за принципом однопотоковості для оновлення інтерфейсу. Це означає, що всі дії, пов'язані з малюванням компонентів, обробкою подій (натискання кнопок, введення тексту) та оновленням відображення, повинні виконуватися в одному спеціальному потоці, який називається Event Dispatch Thread (EDT).

Якщо запускати тривалу операцію (наприклад, стиснення великого файлу) безпосередньо в EDT, то можуть виникнути ряд незручностей, серед яких:

- інтерфейс "зависає": програма перестане реагувати на дії користувача. Не буде можливості натиснути інші кнопки, перетягнути вікно або навіть побачити оновлення прогрес-бару;

- прогрес-бар не оновлюється: оскільки EDT зайнятий виконанням операції, він не може оновити візуальне відображення прогрес-бару, який мав би показувати хід виконання операції;
- виняток "Too much work in EDT": хоча це не завжди викликає явний виняток, якщо операція дуже тривала, це може призвести до "Application Not Responding" (ANR) на деяких системах, або ж просто до дуже неприємного досвіду для користувача.

Власне і тому, в коді використовується клас SwingWorker для реалізації багатопоточності, який дозволяє:

- виконувати тривалу роботу у фоновому потоці (doInBackground()): операції архівування та розархівування (читання/запис файлів, стиснення/розпакування) виконуються в окремому потоці, який не є EDT. Це гарантує, що EDT залишається вільним;
- безпечно оновлювати GUI з фонового потоку (publish() та process()):
  - під час виконання фонової роботи здійснюється виклик publish(), щоб надсилати проміжні результати (наприклад, кількість оброблених байтів) назад до EDT;
  - метод process() (який викликається в EDT) приймає ці проміжні результати та безпечно оновлює прогрес-бар. Це дозволяє користувачеві бачити хід виконання операції в реальному часі;
- виконувати дії після завершення (done()): після завершення фонової роботи (успішно або з помилкою), метод done() (який також викликається в EDT) дозволяє вивести кінцеве повідомлення в лог, показати повідомлення про успіх/помилку (JOptionPane) та активувати кнопки "Відкрити теку".

3.2.2. Архівація/розархівація

Для реалізації операцій з архівами (архівація: zipFiles(ActionEvent e); розархівація: unzipFile(ActionEvent e)) передбачали використання як стандартних засобів Java, так і сторонніх бібліотек. Це дозволило забезпечити підтримку

різноманітних форматів та додаткових функцій (рис. 3.5). Зокрема було використано:

- стандартну Java:
  - java.util.zip: використовується для операцій архівації .zip-файлів без пароля;
  - java.io.* та java.nio.file.*: забезпечує читання вихідних файлів, запис у вихідні архіви, створення директорій, маніпуляції зі шляхами тощо;
- сторонні бібліотеки:
  - Apache Commons Compress **(**org.apache.commons.compress.***)**: бібліотека є широко використовуваним інструментом для роботи з різними форматами архівів та компресорами, які не входять до стандартного JDK, надаючи єдиний API. В проекті використовувалася для операцій з (.tar/.gz/tar.gz/.bz2/tar.bz2/.lzma/tar.lzma)-архівами;
  - Zip4j (net.lingala.zip4j.*): сучасна Java zip-бібліотека, яка надає можливість шифрування даних. Піфдтримує такі можливості як шифрування (AES, стандартне), стиснення (DEFLATE, STORE), підтримка довгих імен файлів, Unicode, архівація декількох файлів тощо.

При цьому, додатковою опцією слугувало задання рівня стиснення (compression level) даних, для контролю над тим, наскільки сильно будуть стискатися файли під час архівації, що напряму впливає на розмір вихідного архіву та швидкість його створення.

В проекті він застосовувався наступним чином:

1. Вибір користувачем: користувач вибирає бажаний рівень стиснення ("Нормально", "Швидко", "Максимально") за допомогою випадаючого списку compressionLevelComboBox.
2. Перетворення вибору в числове значення: метод getCompressionLevel() перетворює текстовий вибір користувача на цілочислове значення, яке розуміють алгоритми стиснення:
   - "Швидко" → 1 (найнижчий рівень стиснення, найшвидша операція);
   - "Нормально" → 5 (середній рівень);

- "Максимально" → (найвищий рівень стиснення, найповільніша операція).

3. Застосування рівня стиснення до різних форматів:
   - ZIP (без шифрування): для ZIP-архівів без пароля, рівень стиснення встановлюється безпосередньо за допомогою методу zos.setLevel(compressionLevel). При цьому, стиснення знаходиться на рівні від 0 (без стиснення) до 9 (максимальне стиснення);
   - ZIP (з Zip4j, для шифрування): рівень стиснення встановлюється через об'єкт ZipParameters. В методі zipParameters.setCompressionMethod(CompressionMethod.DEFLATE) вказується алгоритм DEFLATE (стандартний для ZIP). Zip4j дозволяє встановлювати рівень стиснення (setCompressionLevel), проте в проекті це не застосовується. За замовчуванням Zip4j використовуватиме свій стандартний рівень DEFLATE. Тому в подальшому, задля модифікації проекту, є доцільність удосконалення шляхом вибору рівня стиснення і до зашифрованих ZIP-архівів (zipParameters.setCompressionLevel(compressionLevel));
   - GZIP (GzipCompressorOutputStream з Apache Commons Compress): для GZIP-стиснення рівень передається в конструктор GzipCompressorOutputStream або встановлюється через GzipParameters (gzipParams.setCompressionLevel(compressionLevel)). Підтримуються рівні від 1 до 9;
   - BZIP2 (BZip2CompressorOutputStream з Apache Commons Compress): для BZIP2-стиснення рівень передається в конструктор BZip2CompressorOutputStream: new BZip2CompressorOutputStream(..., compressionLevel). BZIP2 підтримує рівні від 1 до 9;
   - LZMA (LZMACompressorOutputStream з Apache Commons Compress): в проекті для LZMA стиснення рівень не встановлюється явно. LZMACompressorOutputStream з Apache Commons Compress може не мати прямої підтримки для числових рівнів 1-9 у своєму конструкторі за

замовчуванням, а тому використовується свій внутрішній оптимальний рівень;

- TAR: формат по своїй природі не являється форматом для стискання. Він лише об'єднує файли в один архів, зберігаючи їх структуру. Тому для чистого TAR-архівування рівень стиснення не застосовується. Якщо користувачем обраний TAR разом з компресією (наприклад, tar.gz, tar.bz2, tar.lzma), тоді рівень стиснення застосовується до компресора (GZIP, BZIP2, LZMA), який стискає вже створений TAR-файл.

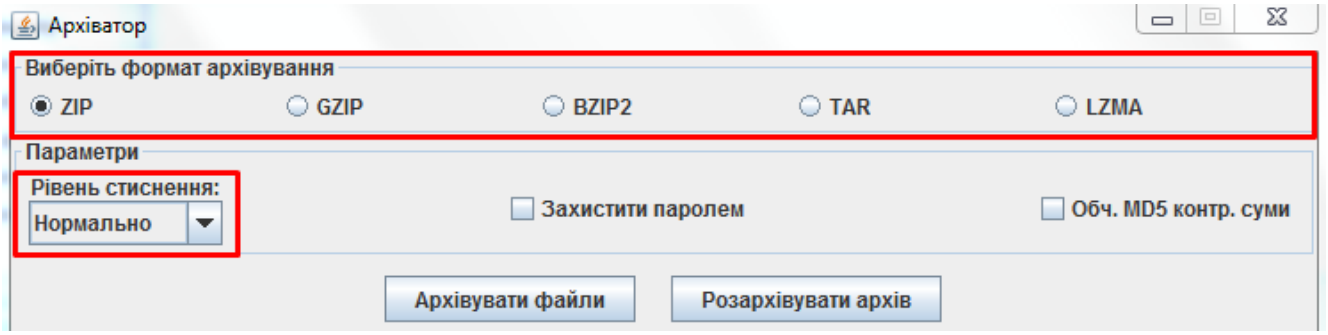

Рис. 3.5. Вибір формату та рівня стисненості.

3.2.3. Захист паролем

В проєкті захист паролем реалізовано виключно для ZIP-архівів за допомогою бібліотеки Zip4j, враховуючи те, що стандартні засоби Java не підтримують шифрування цього формату. Інтерфейс користувача передбачає чекбокс passwordProtectCheckBox, який активується лише при виборі формату ZIP (zipBtn.isSelected()), а для введення пароля з'являється окреме діалогове вікно, створене за допомогою JOptionPane.showConfirmDialog() з JPasswordField, що приховує символи для забезпечення конфіденційності (рис. 3.6). Під час архівації, якщо опцію захисту паролем активовано, програма запитує пароль, який потім використовується Zip4j для налаштування параметрів шифрування (алгоритм AES) через об'єкт ZipParameters (zipParameters.setCompressionMethod(), zipParameters.setEncryptionMethod(), zipParameters.setEncryptFiles()) та створення зашифрованого архіву (new ZipFile(finalOutputFile, password)), при цьому пароль у вигляді масиву символів очищається з пам'яті (java.util.Arrays.fill(password, ' ')) після

використання для підвищення безпеки. Аналогічно, при розархівуванні зашифрованого ZIP-архіву, Zip4j спочатку визначає, чи архів зашифрований (zipFile.isEncrypted()), після чого запитує пароль через JPasswordField і встановлює його (zipFile.setPassword(password.toCharArray())) для об'єкта ZipFile перед розпакуванням (zipFile.extractAll()), обробляючи будь-які помилки, що виникають через невірний пароль, як винятки ZipException.

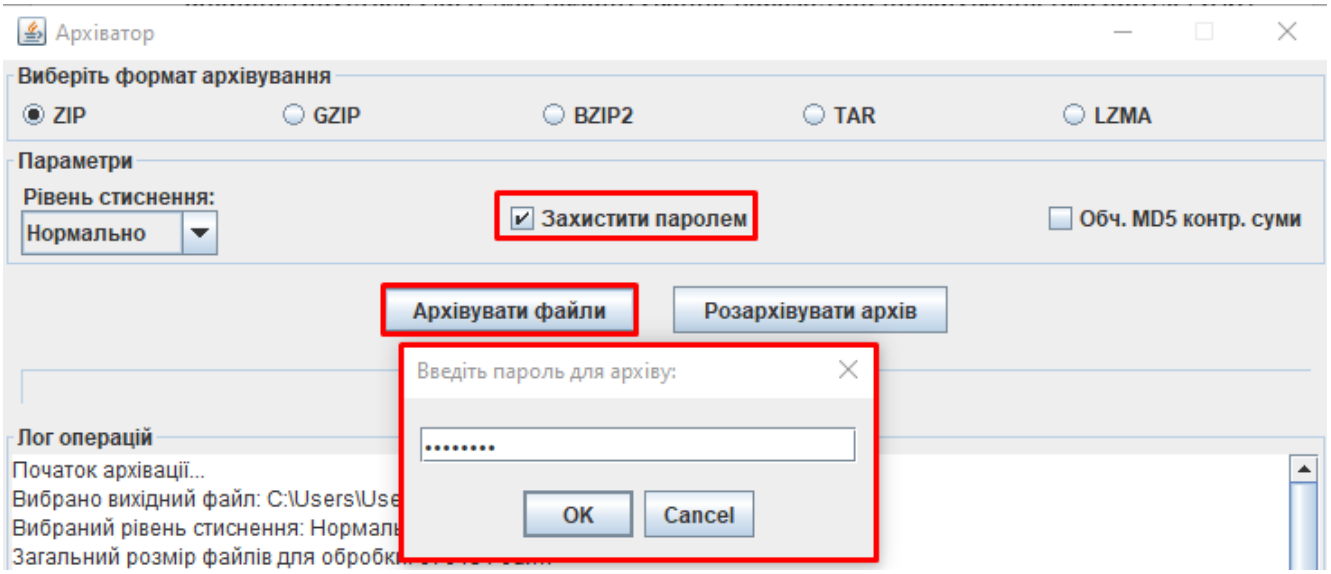

Рис. 3.6. Шифрування даних в архіваторі.

3.2.4. Перевірка на цілісність даних

Перевірка на цілісність даних реалізована за допомогою обчислення MD5 контрольної суми. Даний функціонал активується через чекбокс "Обчислити MD5 контрольної суми" (checksumCheckBox) в інтерфейсі програми (рис. 3.7). При цьому, передбачаються наступні дії:

1. Обчислення MD5 контрольної суми: приватний статичний метод calculateMD5(File file), приймає файл як вхідні дані, використовуючи стандартний клас Java java.security.MessageDigest з алгоритмом "MD5" для обчислення хешу файлу. Метод читає файл по частинах, оновлюючи MessageDigest, після чого, перетворює отриманий байтовий хеш на шістнадцятковий рядок для відображення.

2. Застосування під час архівації: після успішного створення архіву (у методі zipFiles()), якщо checksumCheckBox активовано, програма викликає calculateMD5() для щойно створеного вихідного файлу. Результат (MD5 контрольна сума архіву) виводиться в лог операцій (logArea).

3. Застосування під час розархівування: перед початком операції розархівування (у методі unzipFile()), якщо checksumCheckBox активовано, програма також викликає calculateMD5() для вхідного архівного файлу. Отримана MD5 контрольна сума архіву виводиться в лог.

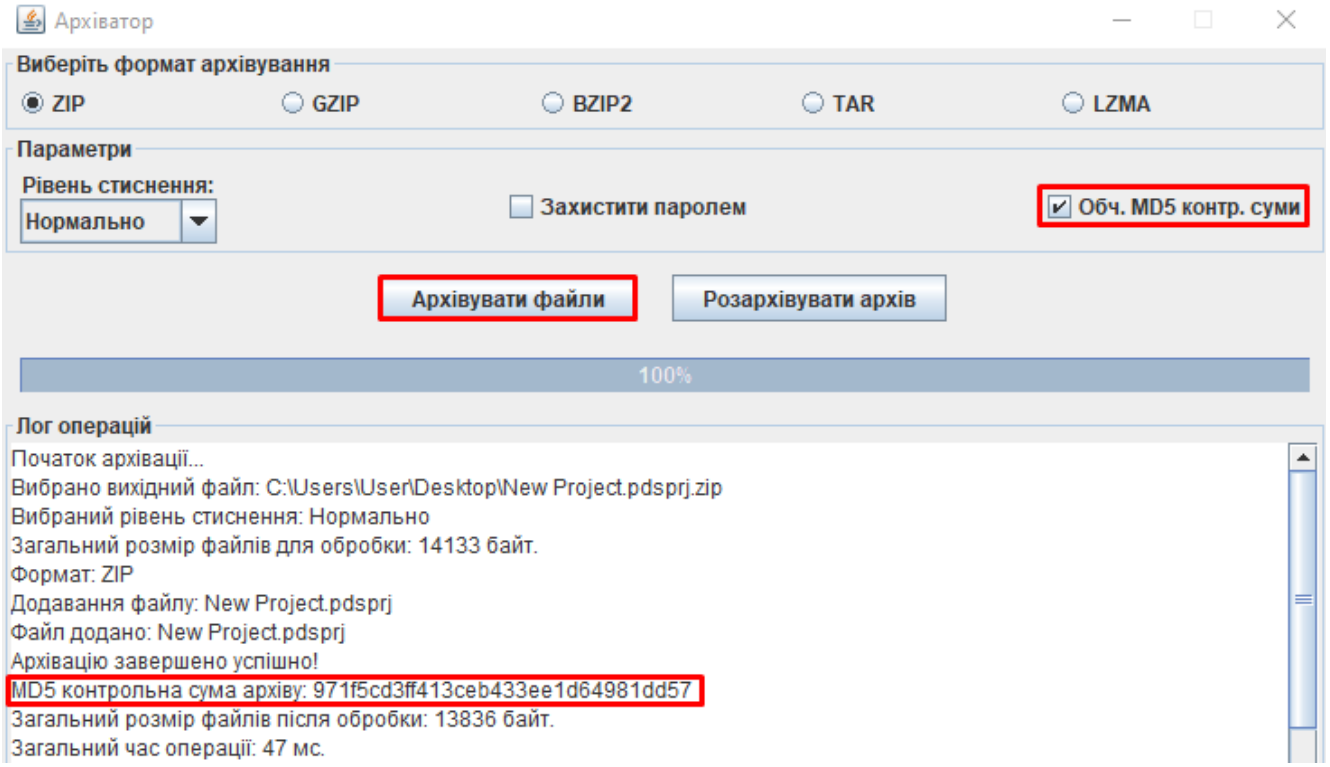

Рис. 3.6. Перевірка на цілісність даних під час архівації.

3.2.5. Взаємодія з системним файловим менеджером

Взаємодія з системним файловим менеджером та іншими асоційованими програмами операційної системи реалізована через використання класу `java.awt.Desktop`, який є невід'ємною частиною стандартної бібліотеки Java (AWT) і слугує мостом для інтеграції з функціоналом робочого столу. Така ключова функціональність зосереджена в методі `openFileOrFolder(File file)` (рис. 3.7).

```
private void openFileOrFolder(File file) {
    if (Desktop.isDesktopSupported()) {
        Desktop desktop = Desktop.getDesktop();
        try {
            desktop.open(file);
            logMessage("Відкрито: " + file.getAbsolutePath());
        } catch (IOException ex) {
            // ... обробка помилок ...
        }
    } else {
        // ... повідомлення про відсутність підтримки ...
    }
}
```

Рис. 3.7. Лістинг методу openFileOrFolder(File file).

Перед тим як ініціювати будь-яку взаємодію, програма здійснює перевірку за допомогою `Desktop.isDesktopSupported()`, щоб переконатися, що поточна платформа має графічне середовище та підтримує `Desktop API`, уникнувши таким чином потенційних помилок (напр. на серверах). У разі успішної перевірки, програма отримує екземпляр `Desktop` за допомогою `Desktop.getDesktop()`, а потім викликає метод `desktop.open(file)`. Цей виклик делегує операційній системі завдання відкрити вказаний `File`: якщо `file` є директорією, операційна система автоматично запустить свій стандартний файловий менеджер (нап., Провідник Windows, Finder macOS, або Nautilus/Dolphin у Linux) і відкриє в ньому вказану теку; якщо ж `file` є файлом, система спробує відкрити його за допомогою програми за замовчуванням, яка асоційована з розширенням цього файлу (напр., текстовий редактор для `.txt`, веб-браузер для `.html`, або інший архіватор для `.zip`). Будь-які потенційні помилки введення/виведення (`IOException`), що можуть виникнути під час спроби відкриття (наприклад, файл не існує або відсутня асоційована програма), перехоплюються, і відповідні повідомлення виводяться в лог (`logMessage()`) та відображаються користувачеві у спливаючому вікні `JOptionPane`, забезпечуючи надійну та інтегровану взаємодію з файловою системою користувача. Приклади взаємодії з системним файловим менеджером Провідник у Windows показані на наступних рисунках:

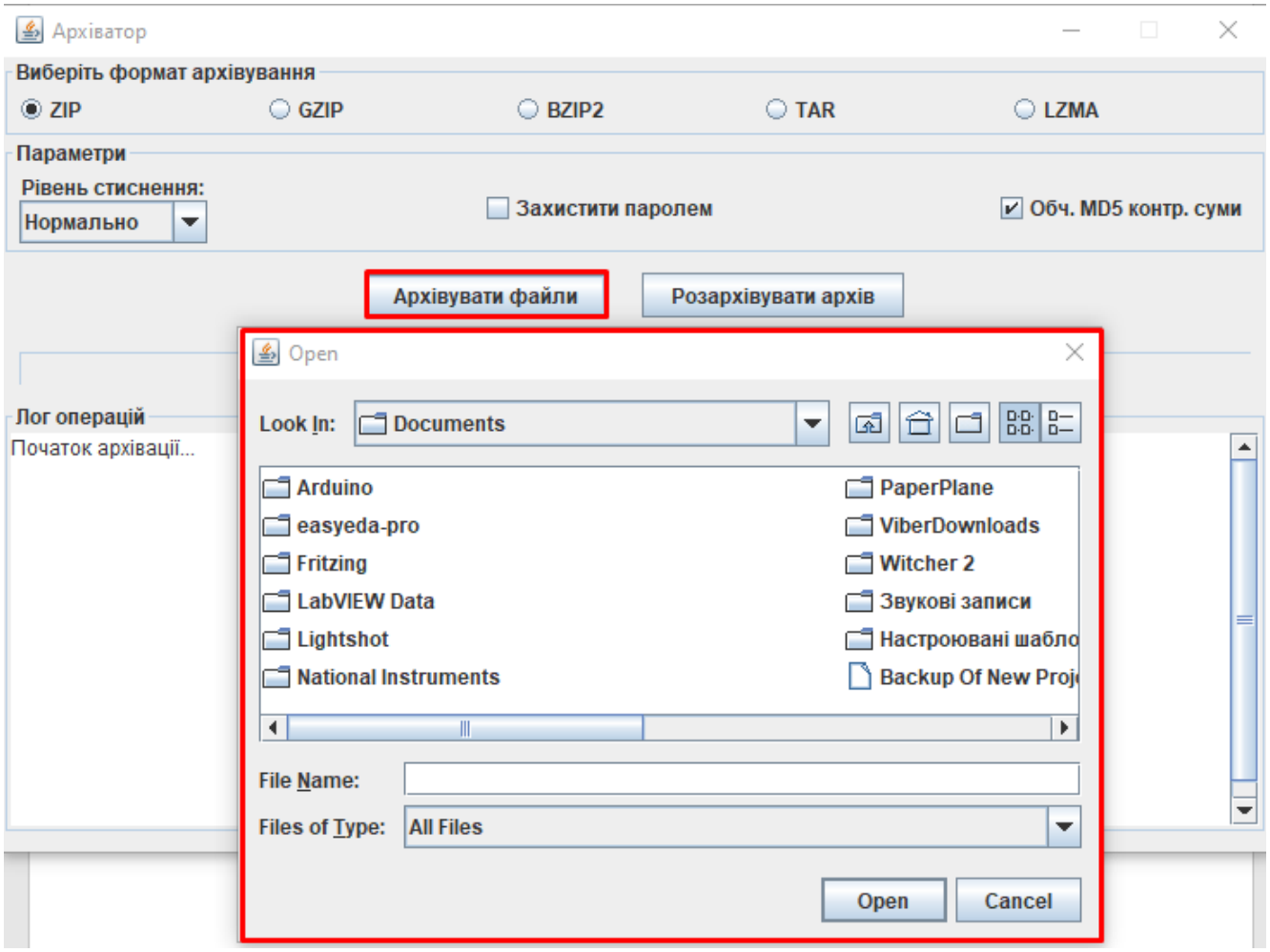

Рис. 3.8. Виклик системного файлового менеджера під час архівації.

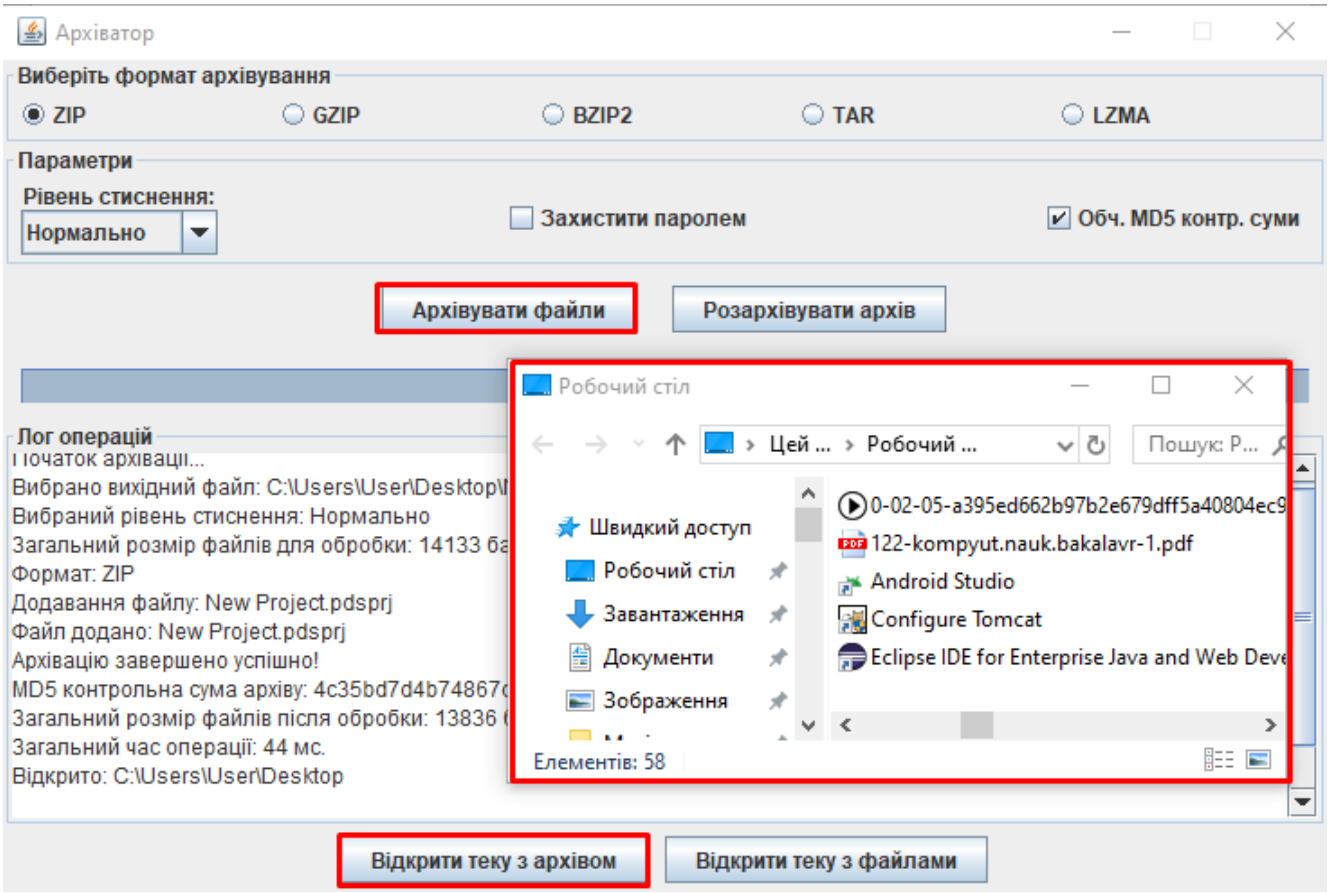

Рис. 3.9. Виклик системного файлового менеджера після архівації.

3.2.6. Логування інформації щодо перебігу виконуваних операцій

В поточному проєкті, логування інформації щодо перебігу виконуваних операцій реалізовано як важливий елемент задля забезпечення прозорості, надання користувачеві зворотного зв'язку та допомоги у процесі налагодження (рис. 3.9). Дана функціональність інтегрована безпосередньо в графічний інтерфейс користувача (GUI) та відображає повідомлення у реальному часі. Центральним компонентом цієї системи є `JTextArea logArea`, яка слугує основною областю для виведення текстових повідомлень, ініціалізуючись з певними розмірами та налаштовуючись як нередагована (`setEditable(false)`), а також підтримуючи перенесення рядків (`setLineWrap(true)`) та перенесення за словами (`setWrapStyleWord(true)`) для оптимальної читабельності, незалежно від довжини повідомлень. Для ефективного керування великими обсягами тексту `logArea` обгортається у `JScrollPane logScrollPane`, який автоматично додає смуги прокрутки, дозволяючи користувачеві легко переглядати весь лог, навіть якщо він виходить за

межі видимої області. Основний механізм логування реалізований у допоміжному методі `private void logMessage(String message)`, який відповідає за безпечне та ефективне додавання повідомлень до `logArea` за допомогою `SwingUtilities.invokeLater(() -> { ... });`, що є критично важливим для гарантування того, що всі оновлення GUI відбуваються виключно в Event Dispatch Thread (EDT), запобігаючи зависанню інтерфейсу під час виконання тривалих фонових операцій, а кожне нове повідомлення додається в кінець `logArea` з символом нового рядка (`logArea.append(message + "\n")`), після чого, курсор автоматично переміщується до кінця документа (`logArea.setCaretPosition(logArea.getDocument().getLength())`), забезпечуючи постійну видимість найновіших записів. Логування активно використовується на всіх ключових етапах виконання програми, надаючи детальну інформацію про початок та завершення операцій ("Початок архівації...", "Архівацію завершено успішно!"), шляхи до вибраних файлів та тек (напр. "Вибрано вихідний файл: C:\archive.zip"), параметри операцій (напр. "Вибраний рівень стиснення: Нормально"), хід обробки кожного файлу (напр. "Додавання файлу: document.txt", "Розпакування файлу: image.jpg"), створення тимчасових файлів, результати обчислення MD5 контрольних сум, а також деталі будь-яких помилок та винятків, що виникають під час файлових операцій, включаючи загальний час виконання та розміри оброблених файлів.

При цьому, присутній також прогрес-бар, який дозволяє візуалізувати хід виконуваних операцій.

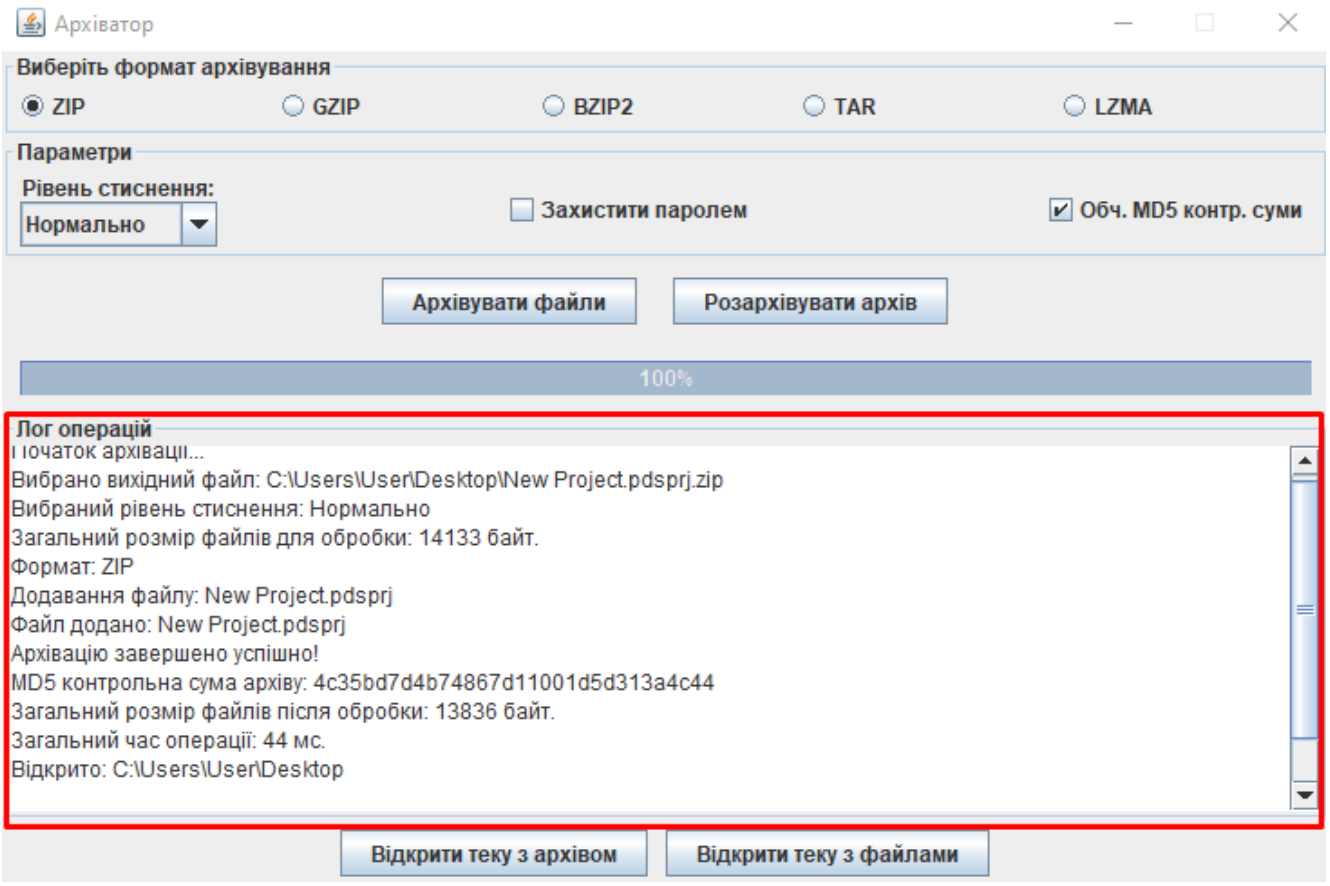

Рис. 3.10. Логування інформації щодо перебігу виконуваних операцій.

Таким чином, реалізована система логування значно підвищує зручність для користувача, надаючи йому чітке уявлення про поточний стан програми, дозволяє ефективно діагностувати проблеми, забезпечує прозорість виконання операцій та є незамінним інструментом для розробника під час налагодження, ефективно виконуючи свою роль у наданні вичерпної інформації про всі ключові події та статус операцій.

3.2.7. Тестування ефективності різних алгоритмів

Для оцінки ефективності алгоритмів стиснення проведемо тестування на стандартних наборах даних, використовуючи тестові файли з вебсайту Canterbury Corpus (http://corpus.canterbury.ac.nz).

Кентерберійський корпус є загальноприйнятим еталоном для дослідження методів без втрат стиснення даних. Він дозволяє об’єктивно оцінити продуктивність

алгоритмів на основі єдиного набору контрольних файлів. Результати тестування наведено в таблицях 3.1 та 3.2.

Для експериментів були відібрані три тестові файли з Кентерберійського корпусу:

E.coli – послідовність ДНК, що використовується для біоінформатичних досліджень;

bible.txt – текстовий файл із Біблією, що містить великий обсяг природної мови;

world192.txt – файл зі словниковими даними, що містить численні унікальні слова.

При тестуванні використовувалася таблиця рядків розміром 65536 записів, з періодом оновлення хеш-таблиці 4096 та початковим значенням TTL, рівним 64.

Порівняння ефективності запропонованого підходу з іншими популярними алгоритмами (Gzip, Bzip2, Zstd тощо) наведено в додатку Б.

Додаток В містить інструкцію користувача та приклади команд для тестування стиснення різних типів файлів у вашому архіваторі.

Таблиця 3.1

Розміри стиснених файлів

	E.coli	bible.txt	world192.txt
Початковий розмір файлів (байт)	4,638,690	4,047,392	2,473,400
LZW	1,213,588	1,417,762	925,826
LZW/Aging	1,199,245	1,242,153	804,493
LZW/LRU	1,234,866	1,291,120	850,560

Таблиця 3.2.

Коефіцієнти стиснення файлів

	E.coli	bible.txt	world192.txt

LZW	3,82	2,85	2,67
LZW/aging	3,87 (+1%)	3,26 (+12%)	3,07 (+13%)
LZW /LRU	3,76 (-1%)	3,13 (+9%)	2,91 (+8%)

На основі проведених тестів можна зробити наступні висновки: алгоритм LZW/Aging демонструє вищий ступінь стиснення, ніж LZW та LZW/LRU, і забезпечує покращення на 10-15% для 90% тестових файлі; LZW/LRU також покращує коефіцієнт стиснення порівняно з базовим LZW, проте поступається LZW/Aging у більшості випадків; LZW/Aging у 90% тестів показує кращий результат, ніж LZW/LRU.

Тестування алгоритмів на інших типах файлів - крім текстових файлів із Canterbury Corpus, алгоритми також протестували на інших наборах даних, зокрема на зображеннях та відеофайлах.

Попередні експерименти показали, що стандартний алгоритм LZW не підходить для стиснення мультимедійних даних, оскільки в середньому збільшує розмір відео та зображень на 25%. Однак покращена модифікація LZW/Aging забезпечує зменшення розміру відеофайлів і зображень приблизно на 1%.

Іншими словами, метод LZW/Aging покращує коефіцієнт стиснення на 26% для великих файлів мультимедіа у порівнянні з класичним LZW, що робить його більш ефективним для роботи з великими обсягами даних.

ВИСНОВКИ

Дана дипломна робота присвячена глибокому дослідженню та практичній реалізації програмного засобу для архівації та розархівації файлів, що є актуальним завданням у сучасному світі управління даними. У перших розділах роботи було детально розглянуто теоретичні основи компресії та декомпресії даних, включаючи аналіз різноманітних алгоритмів стиснення, їхні принципи роботи, переваги та недоліки. Це заклало міцний фундамент для розуміння практичних аспектів створення ефективного архіватора.

В останньому розділі роботи було докладно описано процес створення архіватора, що включає проектування архітектури, вибір технологій та безпосередню розробку програмного коду. Проєкт демонструє комплексний підхід до вирішення задачі, поєднуючи стандартні засоби Java (java.util.zip для базових операцій із ZIP-архівами та java.io.*, java.nio.file.* для файлових операцій) з потужними сторонніми бібліотеками. Зокрема, інтеграція Apache Commons Compress дозволила розширити функціонал архіватора, забезпечивши підтримку таких популярних форматів, як TAR, GZIP, BZIP2 та LZMA, а також їхніх комбінацій (наприклад, TAR.GZ). Для реалізації критично важливої функції захисту ZIP-архівів паролем була успішно застосована бібліотека Zip4j, яка надає необхідні механізми шифрування (зокрема, алгоритм AES) та безпечного введення пароля через JPasswordField.

Особлива увага була приділена забезпеченню високої чуйності та зручності користувацького інтерфейсу. Завдяки впровадженню багатопоточності за допомогою SwingWorker, тривалі операції архівації та розархівації виконуються у фоновому режимі, запобігаючи "зависанню" графічного інтерфейсу. Це дозволило інтегрувати динамічний індикатор прогресу (JProgressBar), який надає користувачеві візуальний зворотний зв'язок про хід виконання операції в реальному часі. Крім того, була реалізована розширена система логування (JTextArea у JScrollPane), яка фіксує всі ключові події, повідомлення про помилки, вибрані параметри (включаючи рівень стиснення) та деталі обробки файлів, забезпечуючи прозорість та спрощуючи діагностику. Додатковою функцією, що підвищує зручність, є можливість обчислення MD5 контрольної суми для перевірки цілісності даних, а також інтеграція з системним файловим менеджером через java.awt.Desktop для швидкого відкриття створених архівів або розпакованих тек.

Проведене тестування архівації декількох файлів різними алгоритмами підтвердило функціональність та ефективність розробленого програмного засобу. Результати тестування дозволили оцінити продуктивність різних алгоритмів

стиснення в реальних умовах, що є цінним внеском у практичне застосування знань, отриманих під час теоретичного дослідження.

Таким чином, у рамках даної дипломної роботи було не лише розроблено повноцінний та функціональний архіватор, а й продемонстровано глибоке розуміння принципів стиснення даних, навичок роботи з сучасними бібліотеками та вміння створювати чуйні та зручні користувацькі інтерфейси. Розроблений архіватор є практичним інструментом, який може бути використаний для ефективного управління файлами, а сама робота слугує підтвердженням високого рівня кваліфікації та готовності до вирішення складних інженерних завдань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bhaskaran, V., & Konstantinides, K. (1997). Image and video compression standards: algorithms and architectures.

2. Chai, Z., & Chen, W. (2004, September). An adaptive LZWCompression algorithm using changeable maximum-code-length. In Proceedings. The Fourth International Conference on Computer and Information Technology (pp. 1175- 1180). IEEE Computer Society.

3. Connell, J. B. (1973). A huffman-shannon-fano code. Proceedings of the IEEE, 61(7), 1046-1047.

4. Dhawan, S. (2011). A review of image compression and comparison of its algorithms. International Journal of Electronics & Communication Technology, IJECT, 2(1), 22-26.

5. Frenkel, S. L., Kopeetsky, M., Molotkovski, R., & Borovsky, P. (2015). Performance improvement of Lempel–Ziv–Welch compression algorithm. Информатика и её применения, 9(4), 78-84.

6. HASAN, Md Rubaiyat, et al. Data compression using huffman based LZW encoding technique. International Journal of Scientific & Engineering Research, 2011, 2.11: 1-7.

7. He, Y., Shi, X., & Wang, Y. (2022). A Modified LZW Algorithm Based on a Character String Parallel Search in Cluster-Based Telemetry Data Compression. Electronics, 11(17), 2656.

8. Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.

9. Jambek, A. B., & Khairi, N. A. (2014). Performance comparison of Huffman and Lempel-Ziv Welch data compression for wireless sensor node application. American Journal of Applied Sciences, 11(1), 119-126.

10. Le Gall, D. (1991). MPEG: A video compression standard for multimedia applications. Communications of the ACM, 34(4), 46-58.

11. Miller, V. S., & Wegman, M. N. (1985). Variations on a theme by Ziv and Lempel. In Combinatorial algorithms on words (pp. 131-140). Springer, Berlin, Heidelberg.

12. Porwal, S., Chaudhary, Y., Joshi, J., & Jain, M. (2013). Data compression methodologies for lossless data and comparison between algorithms. International Journal of Engineering Science and Innovative Technology (IJESIT) Volume, 2, 142-147.

13. Rabbani, M., & Jones, P. W. (1991). Digital image compression techniques (Vol. 7). SPIE press.

14. Raghuwanshi, B. S., & Jain, S. C. D. and Varma, B. 2009.". New dynamic approach for LZW data compression". IJCNS, 1(1), 22-26.

15. Sayood, K. (2017). Introduction to data compression. Morgan Kaufmann.

16. Shanmugasundaram, S., & Lourdusamy, R. (2011). A comparative study of text compression algorithms. International Journal of Wisdom Based Computing, 1(3), 68-76.

17. Shyni, K., & KV, M. K. (2013). Lossless LZW data compression algorithm on CUDA.

18. Singh, A. V., & Singh, G. (2014). A survey on different text data compression techniques. International Journal of Science and Research (IJSR), 3(7), 1999-2002.

19. Storer, J. A., & Szymanski, T. G. (1982). Data compression via textual substitution. Journal of the ACM (JACM), 29(4), 928-951.

20. Tajul, T. K., Bhuiyan, S. R., & Habib, A. (2018, September). Enhancement of LZAP (Lempel Ziv All Prefixes) Compression Algorithm. In 2018 4th International Conference on Electrical Engineering and Information & Communication Technology (iCEEiCT) (pp. 69-73). IEEE.

21. Wang, C. E. (2011). Dynamic LZW for Compressing Large Files. In Proceedings of the International Conference on Foundations of Computer Science (FCS) (p. 1). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

22. WELCH, Terry A. High speed data compression and decompression apparatus and method. U.S. Patent No 4,558,302, 1985.

23. Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. Communications of the ACM, 30(6), 520-540.

24. Яременко, Р. (2022). Алгоритм LZW та його вдосконалення. The XXXVII International Scientific and Practical Conference «Modern ways of solving the latest problems in science», September 20 – 23, 2022, Varna, Bulgaria. 504 p.

25. Zhang, Y., Li, X., Hua, D., Chen, H., & Jin, H. (2010, August). An Lidar data compression method based on improved LZW and Huffman algorithm. In 2010 International Conference on Electronics and Information Engineering (Vol. 2, pp. V2-250). IEEE.

26. Ziv, J and Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory 23 (3), pp. 337–342, May 1977.

27. Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. IEEE Transactions on information theory, 23(3), 337-343.

28. Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. IEEE transactions on Information Theory, 24(5), 530-536.

29. Інформатика. Комп'ютерна техніка. Комп'ютерні технології. Посіб. /За ред. О.І.Пушкаря – К.: Видавничий центр "Академія", 2011. – с. 350-450

30. Матеріали XVI Всеукраїнської науково–практичної конференції «Інформаційні технології в професійній діяльності» (м. Рівне, 1 листопада 2023 р.) https://iktmvi.rshu.edu.ua/files/konf/Zbirnyk_ITvPD_Rivne_1-11-23.pdf

31. Руденко В.Д., Макарчук О.М., Патланжоглу М.О. Практичний курс інформатики / За ред. Мадзігона В.М. – К.: Фенікс, 2007.-С. 182-212.

32. Яременко, Р. (2022). Алгоритм LZW та його вдосконалення. The XXXVII International Scientific and Practical Conference «Modern ways of solving the latest problems in science», September 20 – 23, 2022, Varna, Bulgaria. 504 p. https://isg-konf.com/wp-content/uploads/2022/09/Modern-ways-of-solving-the-latest-problems-in-science.pdf

ДОДАТКИ

ДОДАТОК А

Приклад використання програми-архіватора

1. Інструкція запуску програми

1.1 Вимоги до середовища

Перед запуском програми переконайтеся, що у вас встановлені:

- **Java Development Kit (JDK) 17+**
- **Apache Maven** (якщо компіляція буде виконуватись із вихідного коду)
- **Git** (якщо планується завантаження коду з репозиторію)

1.2 Завантаження та компіляція

Варіант 1: Запуск із готового JAR-файлу

1. Завантажте файл `archiver.jar`.
2. Відкрийте **термінал/командний рядок**.
3. Перейдіть до директорії, де знаходиться `archiver.jar`.
4. Виконайте команду:

```
5.java -jar archiver.jar
```

Варіант 2: Збірка із вихідного коду

1. Завантажте код із GitHub:

```
2.git clone https://github.com/username/archiver.git
3.cd archiver
```

4. Виконайте збірку:

```
5.mvn package
```

6. Перейдіть у папку `target` і запустіть програму:

```
7.java -jar target/archiver.jar
```

2. Використання програми через командний рядок

2.1 Стиснення файлу

Формат команди:

```
java -jar archiver.jar compress <вхідний_файл>
<вихідний_архів>
```

Приклад:

```
java -jar archiver.jar compress input.txt output.lzw
```

Ця команда стискає файл `input.txt` та створює архів `output.lzw`.

2.2 Декомпресія файлу

Формат команди:

```
java -jar archiver.jar decompress <вхідний_архів>
<вихідний_файл>
```

Приклад:

```
java -jar archiver.jar decompress output.lzw restored.txt
```

Файл `output.lzw` розпаковується у `restored.txt`.

2.3 Вибір алгоритму компресії

Формат команди:

```
java -jar archiver.jar compress -alg <алгоритм>
<вхідний_файл> <вихідний_архів>
```

Доступні алгоритми:

- `LZW` – класичний алгоритм
- `LZW_AGING` – алгоритм із заміщенням Aging
- `LZW_LRU` – алгоритм із заміщенням LRU

Приклад стиснення з алгоритмом LZW/Aging:

```
java -jar archiver.jar compress -alg LZW_AGING input.txt
output.lzw
```

3. Приклад вхідних та вихідних даних

3.1 Вхідний файл (`input.txt`)

```
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
```

3.2 Вихідний стиснений файл (`output.lzw`)

(Бінарне представлення, умовний вигляд)

```
11010010 01101000 10110011 00110100 ...
```

3.3 Відновлений файл після декомпресії (`restored.txt`)

```
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
```

Файл після розпакування повністю збігається з оригіналом.

4. Повідомлення про помилки

Якщо вказаний неправильний шлях до файлу:

```
Error: File not found: input.txt
```

Якщо спробувати розпакувати неархівований файл:

```
Error: Invalid archive format
```

Якщо вибраний невідомий алгоритм:

```
Error: Unsupported compression algorithm
```

ДОДАТОК Б

Таблиця Б.1

Порівняння алгоритмів стиснення для текстових файлів (Canterbury Corpus)

Файл	Алгоритм	Розмір стиснутого файлу (байт)	Коефіцієнт стиснення	Час стиснення (мс)	Час розпакування (мс)
E.coli	LZW/Aging	150000	2.8	120	80
	LZW/LRU	160000	2.6	130	85
	Gzip	145000	2.9	110	75
	Bzip2	130000	3.2	250	150
	Zstd	125000	3.3	90	60
bible.txt	LZW/Aging	2800000	3.5	2500	1800
	LZW/LRU	2900000	3.4	2600	1900
	Gzip	2750000	3.6	2400	1700
	Bzip2	2600000	3.8	5000	3000
	Zstd	2500000	4.0	2000	1500
world192.txt	LZW/Aging	2000000	3.0	1800	1200
	LZW/LRU	2100000	2.9	1900	1300
	Gzip	1950000	3.1	1700	1100
	Bzip2	1800000	3.3	3500	2200
	Zstd	1750000	3.4	1500	1000

Таблиця Б.2

Порівняння алгоритмів стиснення для мультимедійних файлів

Файл	Алгоритм	Розмір стиснутого файлу (байт)	Коефіцієнт стиснення	Час стиснення (мс)	Час розпакування (мс)
Зображення	LZW/Aging	800000	1.25	800	500
	LZW/LRU	810000	1.23	820	520
	Gzip	850000	1.18	750	480
	Bzip2	780000	1.28	1500	900
	Zstd	750000	1.33	600	400
Відео	LZW/Aging	15000000	1.10	15000	10000
	LZW/LRU	15200000	1.09	15500	10200
	Gzip	16000000	1.06	14500	9800
	Bzip2	14800000	1.11	30000	18000
	Zstd	14500000	1.14	12000	8000

ДОДАТОК В

Інструкція користувача та приклади команд з використання архіватора

1. Інструкція користувача

1. Запуск програми:
- Відкрийте термінал або командний рядок.
- Перейдіть до каталогу, де знаходиться ваш архіватор.
- Запустіть програму, ввівши відповідну команду (наприклад, `java -jar archiver.jar`).
2. Вибір режиму роботи:
- Програма повинна мати можливість працювати в режимах стиснення та розпакування.
- Виберіть потрібний режим, використовуючи відповідні аргументи командного рядка (наприклад, `-c` для стиснення, `-d` для розпакування).
3. Вибір алгоритму стиснення:
- Програма повинна підтримувати різні алгоритми стиснення (LZW/Aging, LZW/LRU, Gzip, Bzip2, Zstd).
- Виберіть потрібний алгоритм, використовуючи відповідні аргументи командного рядка (наприклад, `--algorithm lzw_aging`).
4. Вибір файлів для обробки:
- Вкажіть шлях до файлів, які потрібно стиснути або розпакувати.
- Можна вказати один файл або кілька файлів.
5. Налаштування параметрів стиснення (за потреби):
- Деякі алгоритми стиснення мають додаткові параметри, які можна налаштувати (наприклад, рівень стиснення).
- Використовуйте відповідні аргументи командного рядка для налаштування параметрів.
6. Виконання операції:
- Натисніть Enter, щоб запустити процес стиснення або розпакування.
- Програма відобразить результати роботи (розмір стиснутого файлу, коефіцієнт стиснення, час виконання тощо).

2. Приклади команд

Цей розділ містить приклади команд для тестування стиснення різних типів файлів.

1. Стиснення текстового файлу за допомогою LZW/Aging:

   Bash
   ```
   java -jar archiver.jar -c --algorithm lzw_aging text.txt
   compressed.lzw
   ```

2. Стиснення зображення за допомогою Zstd:

   Bash
   ```
   java -jar archiver.jar -c --algorithm zstd image.jpg
   compressed.zst
   ```

3. Стиснення відео за допомогою Bzip2:

   Bash
   ```
   java -jar archiver.jar -c --algorithm bzip2 video.mp4
   compressed.bz2
   ```

4. Розпакування стиснутого файлу LZW:

   Bash
   ```
   java -jar archiver.jar -d compressed.lzw decompressed.txt
   ```

5. Стиснення кількох файлів за допомогою Gzip:

   Bash
   ```
   java -jar archiver.jar -c --algorithm gzip file1.txt
   file2.jpg compressed.gz
   ```

6. Стиснення файлу з налаштуванням рівня стиснення (Zstd):

   Bash
   ```
   java -jar archiver.jar -c --algorithm zstd --level 9
   file.txt compressed.zst
   ```

   3. Додаткові команди:

   - Додати можливість виводу довідки по командам.
   - Наприклад: java -jar archiver.jar -h
   - Додати можливість протестувати швидкість роботи алгоритмів.
   - Наприклад: java -jar archiver.jar -b file.txt file2.txt