МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки**

Група: АК-21М **денна форма навчання**

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«ДОСЛІДЖЕННЯ МЕТОДІВ І АЛГОРИТМІВ ОПТИМАЛЬНОГО КОДУВАННЯ ТА СТИСНЕННЯ ДАНИХ У КОМП'ЮТЕРНИХ СИСТЕМАХ»

ЗА НАКАЗОМ № 4601-5/3262 ВІД 15 ВЕРЕСНЯ 2025 РОКУ

здобувача вищої освіти **Смірнова Олексія Сергійовича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ № 5 від 02.12.2025 р.

В.о. завідувача кафедри ІТММ
PhD
_____**Ковальчук Д.М.**

Науковий керівник
к.т.н.
_____**Соболєв О.В.**

м. Харків 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут “Каразінський банківський інститут”
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп’ютерні науки
Освітня програма Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри

Н. І. Стяглик

“15” вересня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЕКТ)

Смірнова Олексія Сергійовича
(прізвище, ім’я, по батькові студента)

1. Тема роботи «Дослідження методів і алгоритмів оптимального кодування та стиснення даних у комп’ютерних системах»

Керівник роботи к.т.н. Соболєв О. В.

затверджені наказом по університету від “15” вересня 2025 року №4601-5/3262

2. Строк подання студентом роботи 24 листопада 2025 р.

3. Перелік питань, які потрібно розробити:
У розділі 1: Провести аналіз предметної області дослідження методів і алгоритмів оптимального кодування та стиснення даних.
У розділі 2: Провести теоретичне дослідження сучасних методів стиснення даних.
У розділі 3: Розробити програмне забезпечення для практичної реалізації досліджуваних алгоритмів оптимального кодування та стиснення даних у комп’ютерних системах.

4. План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної магістерської роботи
2	Затвердження плану і завдання кваліфікаційної магістерської роботи
3	Здача кваліфікаційної магістерської роботи керівнику
4	Підпис кваліфікаційної магістерської роботи у керівника
5	Підпис кваліфікаційної магістерської роботи у нормо контролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної магістерської роботи
7	Захист кваліфікаційної магістерської роботи

5. Дата видачі завдання 15 вересня 2025 року

Студент _____ О. С. Смірнов
Підпис інніціали, прізвище

Керівник роботи _____ О. В. Соболєв
підпис інніціали, прізвище

РЕФЕРАТ
НА КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ
«ДОСЛІДЖЕННЯ МЕТОДІВ І АЛГОРИТМІВ ОПТИМАЛЬНОГО КОДУВАННЯ ТА СТИСНЕННЯ ДАНИХ У КОМП'ЮТЕРНИХ СИСТЕМАХ»
Смірнова Олексія Сергійовича

Кваліфікаційна магістерська робота містить 73 сторінки, 4 таблиці, 6 рисунків, список літератури з 28 найменувань.

Об'єктом дослідження є процеси кодування та стиснення даних у комп'ютерних системах.

Предметом дослідження є методи й алгоритми оптимального кодування та стиснення даних без втрат, а також їх програмна реалізація.

Мета кваліфікаційної магістерської роботи полягає у дослідженні, аналізі та порівнянні ефективності класичних алгоритмів оптимального кодування та стиснення даних, а також розробленні програмного забезпечення для їх практичної реалізації.

Завданнями кваліфікаційної магістерської роботи є:

– провести аналіз сучасних методів і алгоритмів стиснення даних;

– дослідити принципи роботи алгоритмів Шеннона–Фано, Хаффмана, RLE, LZ77, LZW та LZ78;

– розробити програмне забезпечення для реалізації зазначених алгоритмів;

– провести експериментальне порівняння ефективності алгоритмів на різних типах даних.

Актуальність дослідження: сучасні комп'ютерні системи опрацьовують величезні обсяги інформації, що зростають з розвитком цифрових технологій, мультимедійних сервісів та мереж передачі даних. Оптимальне кодування та стиснення дозволяють підвищити ефективність зберігання та передавання даних, скоротити витрати на ресурси та забезпечити стабільну роботу інформаційних систем.

За результатами дослідження:

– проведено теоретичний аналіз алгоритмів стиснення без втрат, описано їх математичні основи та принципи роботи;

– розроблено програмне забезпечення для автоматизованого тестування алгоритмів стиснення;

– здійснено експериментальне порівняння алгоритмів за показниками коефіцієнта стиснення, часу виконання, швидкості декодування та стабільності результатів;

– визначено практичні рекомендації щодо застосування алгоритмів: RLE доцільний для вузькоспеціалізованих даних із великою кількістю повторюваних символів, Шеннона–Фано менш ефективний порівняно з Хаффманом, LZ78 ефективний у поєднанні з Хаффманом, Хаффман показав стабільні результати для різних типів файлів.

Практична новизна: розроблено програмну систему, що дозволяє досліджувати, аналізувати та порівнювати алгоритми стиснення в єдиному середовищі. Програма придатна для навчальних, наукових та прикладних цілей, забезпечує можливість тестування на текстових, графічних, мультимедійних та виконуваних файлах.

Перспективи подальших досліджень полягають у реалізації комбінованих алгоритмів стиснення, застосуванні адаптивних і нейромережних методів, а також оптимізації процесів стиснення для потокової передачі даних у реальному часі та використанні апаратного прискорення та паралельних обчислень для підвищення швидкодії.

Одержані результати мають теоретичну та практичну цінність, створюють основу для вдосконалення алгоритмів оптимального кодування та сприяють підвищенню ефективності зберігання, передавання та обробки інформації в сучасних комп'ютерних системах.

КЛЮЧОВІ СЛОВА: ОПТИМАЛЬНЕ КОДУВАННЯ, СТИСНЕННЯ ДАНИХ, АЛГОРИТМ ШЕННОНА–ФАНО, АЛГОРИТМ ХАФФМАНА, RLE, LZ77, LZW, LZ78, КОМП'ЮТЕРНІ СИСТЕМИ.

ABSTRACT
AT QUALIFICATION MASTER'S WORK
«RESEARCH OF METHODS AND ALGORITHMS FOR OPTIMAL CODING AND DATA COMPRESSION IN COMPUTER SYSTEMS»
Oleksiy Smirnov

The master’s thesis consists of 73 pages, 4 tables, 6 figures, and a list of 28 references.

The object of research is the processes of coding and data compression in computer systems.

The subject of research is the methods and algorithms of optimal coding and lossless data compression, as well as their software implementation.

The aim of the master’s thesis is to study, analyze, and compare the efficiency of classical algorithms for optimal coding and data compression, as well as to develop software for their practical implementation.

The tasks of the master’s thesis are:

– to analyze modern methods and algorithms for data compression;

– to investigate the principles of Shannon–Fano, Huffman, RLE, LZ77, LZW, and LZ78 algorithms;

– to develop software for implementing the specified algorithms;

– to conduct experimental comparison of the algorithms’ efficiency on different types of data.

Relevance of the research: modern computer systems process huge volumes of information that are continuously increasing with the development of digital technologies, multimedia services, and data transmission networks. Optimal coding and compression allow increasing the efficiency of data storage and transmission, reducing resource costs, and ensuring stable operation of information systems.

Based on the results of the research:

– a theoretical analysis of lossless compression algorithms was conducted, their mathematical foundations and operating principles were described;

– software was developed for automated testing of compression algorithms;

– experimental comparison of the algorithms was performed according to compression ratio, execution time, decoding speed, and result stability;

– practical recommendations for algorithm application were defined: RLE is suitable for highly specialized data with many repeating symbols, Shannon–Fano is less efficient compared to Huffman, LZ78 is effective in combination with Huffman, and Huffman showed stable results for different file types.

Practical novelty: a software system was developed that allows researching, analyzing, and comparing compression algorithms in a single environment. The program is suitable for educational, scientific, and applied purposes and enables testing on text, graphic, multimedia, and executable files.

Prospects for further research include implementing combined compression

algorithms, applying adaptive and neural network methods, as well as optimizing compression processes for real-time data streaming and using hardware acceleration and parallel computing to increase performance.

The obtained results have theoretical and practical value, provide a basis for improving optimal coding algorithms, and contribute to enhancing the efficiency of data storage, transmission, and processing in modern computer systems.

KEYWORDS: OPTIMAL CODING, DATA COMPRESSION, SHANNON–FANO ALGORITHM, HUFFMAN ALGORITHM, RLE, LZ77, LZW, LZ78, COMPUTER SYSTEMS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

ASCII – American Standard Code for Information Interchange (Американський стандартний код обміну інформацією)

RLE – Run-Length Encoding (кодювання методом повторів)

LZ77 – Lempel–Ziv 1977 (алгоритм стиснення даних Лемпеля–Зіва 1977 року)

LZ78 – Lempel–Ziv 1978 (алгоритм стиснення даних Лемпеля–Зіва 1978 року)

LZW – Lempel–Ziv–Welch (алгоритм стиснення даних Лемпеля–Зіва–Велча)

ВСТУП

Сучасні комп’ютерні системи опрацьовують колосальні обсяги інформації, які постійно зростають у зв’язку з розвитком цифрових технологій, мультимедійних сервісів і глобальних мереж передачі даних. Збільшення обсягу даних потребує вдосконалення методів їх зберігання, передавання та обробки. Одним із найефективніших напрямів оптимізації є використання алгоритмів стиснення, які дозволяють зменшити розмір інформації без втрати її змісту або з контрольованою втратою якості.

Завдяки алгоритмам стиснення можливо значно скоротити витрати на зберігання великих обсягів даних, підвищити швидкість їх передавання каналами зв’язку та забезпечити ефективне використання обчислювальних ресурсів. Методи оптимального кодування, такі як алгоритми Шеннона–Фано, Хаффмана, RLE, LZ77, LZW та LZ78, стали базовими елементами сучасних систем архівації, мультимедійних форматів і протоколів обміну інформацією. Проте вибір найдоцільнішого алгоритму залежить від типу даних, структури інформації та вимог до швидкодії.

Актуальність роботи полягає у необхідності підвищення ефективності процесів зберігання та передавання даних у комп’ютерних системах за рахунок використання та порівняння класичних алгоритмів стиснення. Систематизація та експериментальне дослідження цих методів дозволяють оцінити їх практичну придатність, а також створити основу для подальшого вдосконалення алгоритмів і побудови комбінованих підходів до стиснення інформації.

Метою роботи є дослідження, аналіз і порівняння ефективності алгоритмів оптимального кодування та стиснення даних, а також розроблення програмного забезпечення для їх практичної реалізації й експериментальної перевірки.

Завдання дослідження:

– провести аналіз сучасних методів і алгоритмів стиснення даних;

– дослідити принципи роботи алгоритмів Шеннона–Фано, Хаффмана, RLE, LZ77, LZW та LZ78;

– розробити програмне забезпечення для реалізації зазначених алгоритмів;

– провести експериментальне порівняння їх ефективності на різних типах даних.

Об’єктом дослідження є процеси кодування та стиснення інформації у комп’ютерних системах.

Предметом дослідження є методи й алгоритми оптимального кодування та стиснення даних без втрат, а також їх програмна реалізація.

У вступі обґрунтовано актуальність роботи, визначено мету, завдання, об’єкт і предмет дослідження, а також коротко описано зміст основних розділів.

У першому розділі наведено аналіз предметної області, розглянуто історичні аспекти розвитку методів стиснення даних, подано їх класифікацію та сформульовано постановку задачі дослідження.

У другому розділі проведено теоретичне дослідження класичних алгоритмів стиснення без втрат, описано їх математичні основи, принципи роботи та порівняльні характеристики.

У третьому розділі подано опис розробленого програмного забезпечення для реалізації й тестування алгоритмів стиснення, наведено результати експериментів і здійснено аналіз їх ефективності.

Отримані результати мають практичну значущість, оскільки можуть бути використані під час проєктування систем архівації, оптимізації процесів передавання інформації, у навчальних курсах з теорії інформації та програмної інженерії, а також як база для подальших досліджень у сфері вдосконалення алгоритмів стиснення даних.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ОСНОВИ ТА ІСТОРІЯ РОЗВИТКУ МЕТОДІВ ОПТИМАЛЬНОГО КОДУВАННЯ ТА СТИСКАННЯ ДАНИХ

1.1. Значення оптимального кодування та стиснення даних у сучасних комп’ютерних системах

У XXI столітті обсяги цифрової інформації, що оброблюється з використанням комп’ютерних систем, зростають експоненційно. За даними аналітичних досліджень, лише у 2025 році загальний обсяг даних у світі перевищив 175 зеттабайти, а до 2027 року прогнозується понад 200 ZB. Це стрімке зростання створює величезне навантаження на серверні ресурси, канали передачі інформації та системи зберігання даних, особливо в умовах активного розвитку хмарних платформ, мультимедійного контенту та Інтернету речей. Сучасні комп’ютерні системи вже не обмежуються окремими серверами чи робочими станціями – сьогодні це великі розподілені інфраструктури, здатні обробляти терабайти та петабайти даних у реальному часі, що робить питання оптимального кодування та стиснення даних надзвичайно актуальним [1].

Стиснення даних дозволяє зменшити обсяг інформації, що зберігається або передається, без втрати її сенсу або якості. У комп’ютерних системах це забезпечує не лише економію місця на дисках та у пам’яті, а й скорочує час обробки даних, підвищує швидкість обміну інформацією між вузлами системи і знижує витрати на обчислювальні ресурси. Зокрема, у великих хмарних центрах обробка та зберігання даних без застосування ефективних алгоритмів стиснення призводить до значних фінансових та енергетичних витрат. З огляду на сучасні тенденції цифровізації, економія місця та оптимізація передачі даних стає не просто бажаною, а обов’язковою умовою функціонування комп’ютерних систем [1, 2].

Оптимальне кодування, як складова стиснення даних, має на меті

мінімізацію розміру інформації при забезпеченні її повноти та точності. Алгоритми оптимального кодування, такі як кодування Хаффмана, арифметичне кодування, алгоритми Лемпеля-Зіва-Велча (LZW) та інші словникові та адаптивні методи, дозволяють досягти максимальної компресії для конкретного типу даних. Вибір конкретного алгоритму залежить від властивостей інформації, вимог до швидкості обробки та допустимої втрати точності. У реальному світі це означає, що одні алгоритми краще підходять для текстових файлів і логів, інші – для мультимедійних потоків, таких як аудіо, відео та зображення [3].

Важливим аспектом застосування стиснення та оптимального кодування є їхня адаптивність. У великих комп'ютерних системах дані надходять з різних джерел та у різних форматах, тому алгоритми повинні вміти працювати як із потоковими даними, так і з обсягами інформації, що зберігаються у масивах. Адаптивні алгоритми дозволяють змінювати методи кодування залежно від статистичних характеристик даних у реальному часі, що забезпечує високу ефективність стиснення і швидкість обробки. Наприклад, у потокових мультимедійних сервісах використання адаптивних алгоритмів дозволяє зменшити обсяг переданого трафіку без помітної втрати якості для користувача [3, 4].

Ефективність стиснення даних також тісно пов'язана з енергетичною ефективністю комп'ютерних систем. У сучасних дата-центрах, де одночасно працюють тисячі серверів, зменшення обсягу переданої та збереженої інформації призводить до зниження споживання енергії, а отже, і до скорочення вуглецевого сліду цифрової інфраструктури. Це питання набуває особливої актуальності у 2025 році, коли організації та компанії все частіше оцінюють технологічні рішення не лише з точки зору продуктивності, а й з точки зору енергоефективності та екологічного впливу [4].

Крім того, стиснення даних дозволяє підвищити надійність та стійкість комп'ютерних систем. Менший обсяг переданої інформації означає меншу ймовірність втрати даних під час передачі та зменшення навантаження на

канали зв’язку, що особливо важливо для розподілених систем з великою кількістю вузлів. У великих хмарних системах оптимальне кодування даних сприяє більш ефективному використанню мережевих ресурсів і зменшенню затримок при доступі до інформації, що є критично для сервісів реального часу, включаючи відеоконференції, онлайн-ігри та стрімінгові платформи.

Особливо важливо, що сучасні комп’ютерні системи обробляють дані, які постійно генеруються штучним інтелектом, машинним навчанням та аналітичними платформами. Ці дані включають навчальні набори, моделі, лог-файли та результати обчислень, обсяги яких зростають щоденно. Без ефективного стиснення та оптимального кодування робота з такими потоками даних була б фізично неможлива: зберігання всіх результатів обчислень потребувало б величезних ресурсів, а передача таких обсягів через мережу – зайняла б неприйнятно багато часу [2-4].

Таким чином, дослідження та застосування методів оптимального кодування та стиснення даних у комп’ютерних системах є ключовим напрямом сучасної інформатики. Воно дозволяє забезпечити компактне зберігання даних, оптимізувати їх передачу, підвищити продуктивність та енергетичну ефективність систем, а також гарантувати високу якість обробки інформації навіть при роботі з надвеликими масивами даних. У 2025 році ці аспекти стають особливо актуальними у зв’язку зі стрімким зростанням цифрової інформації, розвитком хмарних технологій та необхідністю забезпечення ефективної роботи розподілених комп’ютерних систем у реальному часі.

1.2. Історичний розвиток алгоритмів кодування та стиснення

Історія розвитку алгоритмів стиснення даних тісно пов'язана з еволюцією теорії інформації та зростанням потреби в ефективному зберіганні та передачі даних. Перші кроки в цьому напрямку були зроблені в середині XX століття, коли Клод Шеннон у своїй праці "A Mathematical Theory of

Communication" (1948) сформулював основи теорії інформації [5, 6]. Він увів поняття ентропії як міри інформаційної насиченості повідомлення, що визначає мінімальну кількість бітів, необхідних для його кодування. Ці ідеї стали підґрунтям для майбутніх практичних алгоритмів стиснення.

У 1952 році Девід Хаффман запропонував алгоритм, що забезпечував мінімальну довжину коду за заданими ймовірностями символів. Цей метод став основою для статистичних методів стиснення, де частотний аналіз символів дозволяє створювати оптимальні коди для представлення даних [7].

У 1970–1980-х роках, із поширенням персональних комп’ютерів, виникла потреба у методах, які не потребують попереднього аналізу частот символів. Були розроблені словникові алгоритми стиснення, такі як LZ77, LZ78 та їхня модифікація LZW. Ці алгоритми дозволяють ефективно стискати дані за допомогою автоматичного створення словника повторюваних фрагментів, що значно спрощує процес стиснення та зменшує обсяг необхідної пам'яті [8].

На початку 1990-х, із розвитком цифрових технологій і мультимедіа, особливої популярності набули алгоритми стиснення з втратами. Для зображень найпоширенішим стандартом став JPEG, для аудіо – формат MP3, а для відео – алгоритми родини MPEG. Ці стандарти дозволяють значно зменшити обсяг даних, що передаються або зберігаються, при мінімальних втратам якості, що сприяло масовому розповсюдженню мультимедійного контенту [9].

Останніми роками зростає інтерес до застосування машинного навчання у задачах стиснення. Дослідники експериментують з автоенкодерами, згортковими нейронними мережами та трансформерами для генерації латентних представлень, які можна стискати ще глибше. Ці підходи дозволяють досягати високої ефективності стиснення, особливо в контексті великих даних та складних мультимедійних форматів [10].

1.3. Класифікація сучасних методів кодування та стиснення

Методи стиснення даних класифікуються за різними ознаками, але найосновнішим поділом залишається поділ на алгоритми без втрат (lossless) та алгоритми з втратами (lossy). Цей поділ відрізняє способи, які дозволяють повністю відновити первинні дані після декодування, від тих, що допускають часткову втрату інформації задля отримання більш високої ступені стиснення. Алгоритми без втрат застосовуються у тих випадках, коли навіть найменша втрата даних неприпустима – наприклад, у текстових файлах, програмному коді та ін. Серед сучасних рішень без втрат виділяються RLE (Run-Length Encoding), дуже простий спосіб, який працює добре, якщо в даних багато повторень; словникові методи – LZ77, LZ78, LZW – які автоматично формують словник повторюваних фрагментів; ентропійне кодування, включно з алгоритмами Хаффмана та арифметичного кодування; метод Golomb/Rice та трансформації на кшталт Burrows-Wheeler Transform, які переупорядковують або структурують дані перед ентропійним кодуванням, щоб зробити їх більш стискуваними. Сьогодні до числа популярних lossless-алгоритмів додаються такі, як Zstandard, LZ4 чи Snappy, які оптимізовані під високу швидкість стиснення і розпаковки при прийнятній компресії [1-4, 8, 9].

Алгоритми з втратами використовуються в тих випадках, коли можна пожертвувати частиною інформації без значної шкоди для споживача – найчастіше це мультимедійні дані: зображення, відео, аудіо. Класичним прикладом зображень є JPEG, який використовує дискретне косинусне перетворення (DCT), квантування та подальше ентропійне кодування. Для відео використовуються стандарти MPEG-сімейства, H.264/AVC, H.265/HEVC, новіше AV1, JPEG XL, JPEG XR тощо, які застосовують трансформаційні методи, міжкадрове прогнозування, компресію руху, субсемплінг хроми і адаптивне кодування залишків. Для аудіо MP3 усе ще широко використовується, але набирають популярності нові стандарти, як-от

AAC, Opus, а також датабази нейромережевих підходів, які навчаються стисненню аудіо з урахуванням людського сприйняття. У 2025 році багато таких стандартів вдосконалюються, зокрема в бік підвищення ефективності при низьких бітових ставках, зниження затримки та адаптації до мережевих умов передачі.

Крім базового поділу, виділяють гібридні та адаптивні методи, які поєднують елементи обох підходів: lossless і lossy. Наприклад, в одному з форматів або систем кодування ті частини даних, які є менш чутливими до втрат якості (фон, деталі, текстури), можуть стискатися з втратами, а важливі частини – без втрат. Прикладами таких сучасних підходів є HEIF / HEVC з адаптивним квантуванням, формати зображень чи відео, які дозволяють змішувати рівні якості для різних сцен чи регіонів зображення. Також росте застосування near-lossless методів, тобто способів, які допускають дуже малі і контрольовані втрати (скажімо, максимально допустима похибка або спотворення), що практично не сприймаються користувачем [8, 9].

Ще одна важлива класифікація стосується архітектури та використаних технологій: традиційні алгоритми та алгоритми на основі машинного навчання. У традиційних методах основою є словникові, трансформаційні, ентропійні методи, які вже десятки років довели свою ефективність. Але в останні роки стають популярними підходи з використанням автоенкодерів, згорткових нейронних мереж, трансформерів, генеративних моделей та інших глибинних архітектур, особливо для зображень і відео. Такі методи дозволяють будувати латентні представлення даних, які можна стискати значно глибше, іноді комбінуючи з класичним кодуванням залишків та ентропійної частини. Вони часто використовуються для контенту, де важлива висока візуальна якість при дуже низьких бітрейтах [8, 9].

Також існують спеціалізовані класифікації, залежно від типу даних (текст, зображення, відео, аудіо, наукові дані тощо), швидкості обробки, обмежень на ресурси (обчислювальних, пам’яті, енергії), а також вимог до затримок (реальний час або пакетна обробка). Для прикладу в наукових

обчисленнях, метеорології чи будь-яких високопродуктивних обчисленнях важливі так звані error-bounded lossy компресори та методи, які дозволяють контролювати похибку відновлення, ще зберігаючи значне стиснення. Для тексту та баз даних – пріоритет на абсолютну точність, тому використовують тільки lossless або near-lossless методи.

Сучасні формати та бібліотеки реалізують комбіновані підходи: передобробка даних (наприклад, переструктурування байтів чи бітів, бит-шейфлінг, shuffle / bitshuffle), потім застосування швидкого lossless алгоритму, або спочатку застосування lossy компресії з врахуванням допустимості похибки, потім залишки стискання lossless способом. Приклади: формати зображення як JPEG XR, HEIF, нові кодеки для відео, а в наукових даних спеціалізовані методи як ZFP, SZ для чисел з плаваючою крапкою, які дозволяють гнучко обирати компроміс між точністю та стисненням [8-10].

Отже, класифікація сучасних методів кодування і стиснення даних охоплює такі основні групи: lossless методи, lossy методи, гібридні / адаптивні підходи, методи на основі ML / DL, і спеціалізовані методи для різних типів даних.

1.4. Постановка задачі дослідження

З огляду на результати проведеного аналізу сучасних методів стиснення даних, зокрема алгоритмів Шеннона–Фано, Хаффмана, RLE та LZ78, сформульовано задачу дослідження, що полягає у визначенні ефективності їх застосування для різних типів даних та розробленні інструменту для практичного порівняння. Зростання обсягів цифрової інформації у сучасних комп’ютерних системах зумовлює необхідність оптимізації процесів зберігання і передачі даних, що потребує вибору алгоритмів, здатних забезпечити найкраще співвідношення між ступенем стиснення, швидкодією обробки та збереженням якості інформації. Об’єктом

дослідження є процеси кодування та стиснення інформації у комп’ютерних системах, а предметом – методи оптимального стиснення без втрат і їх програмна реалізація. Метою дослідження є розроблення програмного забезпечення для реалізації та експериментального порівняння класичних алгоритмів стиснення даних і формування рекомендацій щодо вибору оптимального методу залежно від типу вхідної інформації. Для досягнення цієї мети необхідно було здійснити порівняльний аналіз принципів роботи зазначених алгоритмів, розробити програмні модулі для їхньої реалізації, провести експериментальне тестування на різних типах файлів – текстових, графічних і двійкових, – а також визначити коефіцієнт стиснення. Таким чином, постановка задачі передбачає розв’язання як теоретичних, так і прикладних аспектів проблеми оптимального кодування, спрямованих на підвищення ефективності обробки інформації та раціональне використання ресурсів сучасних обчислювальних систем.

1.5. Висновки до розділу 1

У першому розділі, в ході аналізу предметної області було встановлено, що методи оптимального кодування та стиснення даних відіграють ключову роль у забезпеченні ефективності функціонування сучасних комп’ютерних систем. Зростання обсягів інформації та потреба в її швидкій передачі й зберіганні обумовлюють необхідність використання алгоритмів, які дають змогу мінімізувати надлишковість даних без втрати їх змістової цілісності. Оптимальне кодування є основою більшості сучасних технологій обміну, архівації та обробки даних.

Історичний огляд розвитку алгоритмів кодування та стиснення показав, що еволюція цих методів відбувалася поступово – від статистичних принципів Шеннона–Фано та Хаффмана до словникових методів LZ-сімейства. З часом з’явилися трансформаційні та гібридні методи, які поєднують високу швидкодію та адаптивність до різних типів інформації.

Такі підходи стали базою для створення сучасних форматів стиснення зображень, аудіо- та відеоданих.

У результаті класифікації сучасних методів стиснення визначено основні групи алгоритмів: без втрат (lossless) та з втратами (lossy). До першої групи належать алгоритми Шеннона–Фано, Хаффмана, RLE, LZ77, LZ78 та їх модифікації, які забезпечують повне відновлення вихідних даних. Алгоритми з втратами застосовуються переважно у сфері мультимедійної інформації, де допустиме часткове спрощення структури даних задля підвищення ступеня стиснення.

На підставі проведеного аналізу сформульовано постановку задачі дослідження, яка полягає у теоретичному та експериментальному вивченні ефективності класичних алгоритмів стиснення даних, визначенні їх переваг, недоліків і сфер практичного застосування. Обґрунтовано доцільність створення програмного забезпечення для порівняння ефективності алгоритмів Шеннона–Фано, Хаффмана, RLE та LZ78 на різних типах даних. Це дозволить визначити залежність ефективності методів від структури вхідної інформації та обрати оптимальні підходи до її обробки.

Таким чином, у першому розділі сформовано теоретичну та методологічну основу подальших досліджень. Отримані результати стали підґрунтям для розроблення експериментальної частини роботи, спрямованої на практичну реалізацію алгоритмів стиснення та оцінку їх продуктивності у реальних умовах функціонування комп’ютерних систем.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ МЕТОДІВ СТИСКАННЯ ДАНИХ

2.1. Метод кодування Шенона–Фано

Метод Шенона–Фано належить до класичних алгоритмів статистичного кодування без втрат, який застосовується для зменшення середньої довжини кодових слів шляхом призначення коротших кодів більш імовірним символам. Його розробили незалежно один від одного два науковці – Клод Шеннон і Роберт Фано у 1948 році [5, 6]. Ідея методу полягає в упорядкуванні символів за ймовірністю появи та послідовному поділі множини символів на дві частини з приблизно рівними сумами ймовірностей. Кожній підмножині присвоюється двійковий символ – «0» або «1», після чого процес поділу повторюється доти, доки кожен символ не отримає свій унікальний бітовий код. У результаті формується префіксний код, у якому жоден код не є початком іншого, що забезпечує однозначність декодування [2, 6].

Нехай задано множину символів $S = \{s_1, s_2, ..., s_n\}$ з відповідними ймовірностями появи $P = \{p_1, p_2, ..., p_n\}$, де сума всіх імовірностей дорівнює одиниці:

$$\sum_{i=1}^{n} p_i = 1.$$

Метою алгоритму є побудова такого двійкового коду, який мінімізує середню довжину коду

$$L = \sum_{i=1}^{n} p_i \cdot l_i,$$

де l_i – довжина коду для символу s_i. Ідеальною теоретичною межею середньої довжини є ентропія повідомлення, що визначається як

$$H = \sum_{i=1}^{n} p_i \cdot \log_2 p_i.$$

Чим ближче відношення H/L до одиниці, тим ефективніше відбувається кодування.

Робота алгоритму починається з визначення частот або ймовірностей появи кожного символу у вхідному повідомленні. Потім усі символи впорядковуються за спаданням ймовірностей. Далі впорядкований список символів рекурсивно ділиться на дві частини з максимально близькими сумами ймовірностей. Першій групі приписується біт «0», другій – «1». Цей поділ повторюється доти, доки кожен символ не отримає свій власний код. У результаті виходить компактна таблиця кодів, у якій символи з вищою частотою появи мають коротші бітові послідовності [2, 3, 6].

Для ілюстрації розглянемо приклад. Маємо символи з такими ймовірностями: A – 0.4, B – 0.2, C – 0.2, D – 0.1, E – 0.1. Після сортування за спаданням отримаємо порядок A, B, C, D, E. Далі ділимо множину на дві групи: у першій залишаємо символ A з імовірністю 0.4, у другій – решту символів з сумарною ймовірністю 0.6. Першій групі призначаємо код «0», другій – «1». Потім друга група ділиться на дві підмножини: (B, C) і (D, E), для яких відповідно додаємо біти «0» і «1». Отже, коди будуть такі: A – 0, B – 100, C – 101, D – 110, E – 111 (див. таблицю 2.1). Середня довжина коду дорівнює

$$L = 0.4 \times 1 + 0.2 \times 3 + 0.2 \times 3 + 0.1 \times 3 + 0.1 \times 3 = 2.2.$$

Таблиця 2.1.

Приклад кодування методом Шенона–Фано

Символ	Ймовірність символа	1 етап	2 етап	2 етап	Код символа
A	0.4	0			0
B	0.2	1	0	0	100
C	0.2			1	101
D	0.1		1	0	110
E	0.1			1	111

Ентропія повідомлення становить приблизно 2.12 біта, а ефективність кодування $\eta = H/L = 0{,}96.$ Це свідчить про те, що алгоритм працює з високою ефективністю, хоча і не завжди забезпечує мінімальну середню довжину, на відміну від алгоритму Хаффмана.

Формально алгоритм можна описати так. Для впорядкованої множини символів $S = \{s_1, s_2, ..., s_n\}$ з ймовірностями $P = \{p_1, p_2, ..., p_n\}$, де $p_1 \geq p_2 \geq ... \geq p_n$, виконуємо такі дії: якщо множина містить лише один символ, кодування завершено; інакше ділимо її на дві підмножини S_1 і S_2 так, щоб різниця між сумами ймовірностей $\left|\sum_{s_i \in S_1} p_i - \sum_{s_j \in S_2} p_j\right|$ була мінімальною. Усі символи з S_1 отримують біт «0», а з S_2 – «1». Потім алгоритм виконується рекурсивно для кожної підмножини.

Лістинг коду для реалізації алгоритму Шенона–Фано мовою програмування C#, наведено нижче. В програмі обчислюється ймовірності появи кожного байта, рекурсивно формує бітові коди для символів та кодується весь файл у послідовність бітів, яку потім упаковує у байти. Перший байт нового файлу зберігає кількість доданих нулів для вирівнювання до повного байта, що забезпечує коректне декодування. Після завершення функція виводить розмір оригінального та стисненого файлу для порівняння ефективності стиснення.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

class ShannonFanoCompressor
{
    class Symbol
    {
        public byte Value;
        public double Probability;
        public string Code = "";
    }
    public static string CompressFileToNew(string filePath)
```

```
    {
        byte[] data = File.ReadAllBytes(filePath);
        var symbols = data.GroupBy(b => b)
                          .Select(g => new Symbol { Value =
g.Key, Probability = (double)g.Count() / data.Length })
                          .OrderByDescending(s => s.Probability)
                          .ToList();
        ShannonFanoEncode(symbols, 0, symbols.Count - 1);
        var codeTable = symbols.ToDictionary(s => s.Value, s =>
s.Code);
        StringBuilder encodedBits = new StringBuilder();
        foreach (var b in data)
            encodedBits.Append(codeTable[b]);
        int padding = 8 - encodedBits.Length % 8;
        if (padding != 8)
            encodedBits.Append('0', padding);
        byte[] compressedData = new byte[encodedBits.Length /
8];
        for (int i = 0; i < compressedData.Length; i++)
        {
            string byteString = encodedBits.ToString(i * 8, 8);
            compressedData[i] = Convert.ToByte(byteString, 2);
        }
        string directory = Path.GetDirectoryName(filePath);
        string                    fileName                    =
Path.GetFileNameWithoutExtension(filePath);
        string extension = Path.GetExtension(filePath);
        string     newFilePath     =     Path.Combine(directory,
$"{fileName}_compressed{extension}");
        using    (var    fs    =    new    FileStream(newFilePath,
FileMode.Create))
        {
            fs.WriteByte((byte)padding);   //   перший   байт   =
кількість додаткових бітів
            fs.Write(compressedData, 0, compressedData.Length);
        }
        Console.WriteLine($"Оригінальний     файл:     {filePath}
({data.Length} байт)");
        Console.WriteLine($"Стиснений     файл:     {newFilePath}
({compressedData.Length + 1} байт)");
        return newFilePath;
    }
    private static void ShannonFanoEncode(List<Symbol> symbols,
int start, int end)
    {
        if (start >= end) return;
        double total = symbols.Skip(start).Take(end - start +
1).Sum(s => s.Probability);
        double acc = 0;
        int split = start;
        for (int i = start; i <= end; i++)
        {
            acc += symbols[i].Probability;
```

```
            if (acc >= total / 2)
            {
                split = i;
                break;
            }
        }
        for (int i = start; i <= split; i++) symbols[i].Code += 
"0";
        for (int i = split + 1; i <= end; i++) symbols[i].Code 
+= "1";
        ShannonFanoEncode(symbols, start, split);
        ShannonFanoEncode(symbols, split + 1, end);
    }
    static void Main()
    {
        string inputFile = "example.txt"; // вхідний файл
        CompressFileToNew(inputFile);
    }
}
```

Метод Шенона–Фано має низку переваг: просту реалізацію, можливість створення префіксного коду, який забезпечує однозначне декодування, і високу ефективність при правильному визначенні ймовірностей. Водночас він має і певні недоліки – не завжди гарантує мінімальну середню довжину коду, чутливий до похибок у визначенні частот і не є адаптивним, тобто потребує попереднього аналізу всього повідомлення [2]. Незважаючи на це, метод має важливе навчальне та теоретичне значення. Він ілюструє принципи побудови ефективного статистичного кодування та заклав основу для подальших вдосконалень, зокрема алгоритму Хаффмана, який є оптимізованою формою побудови двійкових дерев кодування. Таким чином, алгоритм Шенона–Фано посідає важливе місце в розвитку методів стиснення інформації без втрат і залишається класичним прикладом застосування ймовірнісного підходу до оптимізації довжини кодів.

2.2. Метод кодування Хаффмана

Метод кодування Хаффмана, запропонований Девідом Хаффманом у 1952 році, є класичним алгоритмом ентропійного кодування, який забезпечує

оптимальне стиснення без втрат для статичних джерел інформації. Головна ідея алгоритму полягає в тому, що символи, які зустрічаються частіше, повинні отримувати коротші кодові слова, а рідкісні – довші, при цьому середня довжина кодового слова досягає мінімуму. На відміну від алгоритму Шеннона–Фано, який формує коди рекурсивно з кореня дерева до листя, алгоритм Хаффмана будує бінарне дерево від листя до кореня. Листя дерева відповідає символам алфавіту, а внутрішні вузли створюються як сукупності двох вузлів із найменшою частотою появи, забезпечуючи мінімальну середню довжину кодового слова [7].

Алгоритм Хаффмана починається з підрахунку частоти кожного символу у вихідному повідомленні. Далі формується список вузлів, де кожен вузол містить символ і його частоту. На кожному кроці алгоритму вибираються два вузли з мінімальною частотою, створюється новий вузол-батько, частота якого дорівнює сумі частот двох дочірніх вузлів, і додається в список вузлів. Один із дочірніх вузлів позначається бітовим кодом «0», інший – «1». Процес повторюється до тих пір, поки не залишиться один вузол, який стає коренем бінарного дерева. Таким чином, для кожного символу формується унікальне префіксне кодове слово, яке дозволяє однозначне декодування бітового потоку [2, 7, 8].

Формально, алгоритм кодування Хаффмана можна подати наступними кроками:

1) Обчислити частоти зустрічання всіх символів у повідомленні.

2) Створити для кожного символу вузол із зазначенням його частоти.

3) Поки кількість вузлів більше одного:

а) вибрати два вузли з мінімальними частотами;

б) створити новий вузол з частотою, що дорівнює сумі обраних вузлів;

в) об’єднати два вузли як дочірні цього нового вузла; присвоїти одному з дочірніх вузлів код «0», іншому – «1»;

г) видалити обрані вузли зі списку і додати новий вузол.

Коди символів формуються шляхом проходу від кореня до листя, при цьому кожен лівий перехід відповідає «0», правий – «1».

Практична реалізація алгоритму передбачає кілька ключових аспектів. По-перше, оскільки кодові слова мають змінну довжину, їх потрібно ефективно пакувати у байтовий потік для зберігання або передачі. Зазвичай використовується буфер байта, куди поступово записуються біти кодів символів. По-друге, щоб декодер міг відновити оригінальні дані, необхідно передати або таблицю частот, або відновлювальну структуру дерева. У найпростішому варіанті у заголовку стисненого файлу зберігають масив частот символів (наприклад, 256 32-бітових значень для байтового алфавіту), що дозволяє детерміновано відтворити дерево кодування. По-третє, оскільки довжина бітового потоку не завжди кратна 8, необхідно доповнювати останній байт нульовими бітами або зберігати інформацію про кількість заповнюваних бітів (padding), щоб декодер міг правильно відновити оригінальне повідомлення.

Для ілюстрації розглянемо приклад. Нехай вхідне повідомлення містить символи з частотами: A:45, B:13, C:12, D:16, E:9, F:5. Спочатку створюємо вузли для кожного символу. Далі обираємо два вузли з найменшою частотою: F(5) і E(9), об’єднуємо у вузол з частотою 14. Потім наступні два вузли з найменшою частотою: C(12) і B(13) → 25. Повторюємо процедуру, об’єднуючи вузли 14 і D(16) → 30, і вузли 25 та A(45) → 70. Нарешті об’єднуємо вузли 30 і 70 → 100, отримавши корінь дерева. Формуючи коди від кореня до листя, отримуємо A=0, B=101, C=100, D=111, E=1101, F=1100 (рис. 2.1). Середня довжина коду $L \approx 2.24$ біт/символ, що близько до ентропії $H \approx 2.22$ біт/символ. Цей приклад демонструє ефективність алгоритму Хаффмана та його здатність мінімізувати середню довжину кодового слова порівняно зі статичним методом Шеннона–Фано.

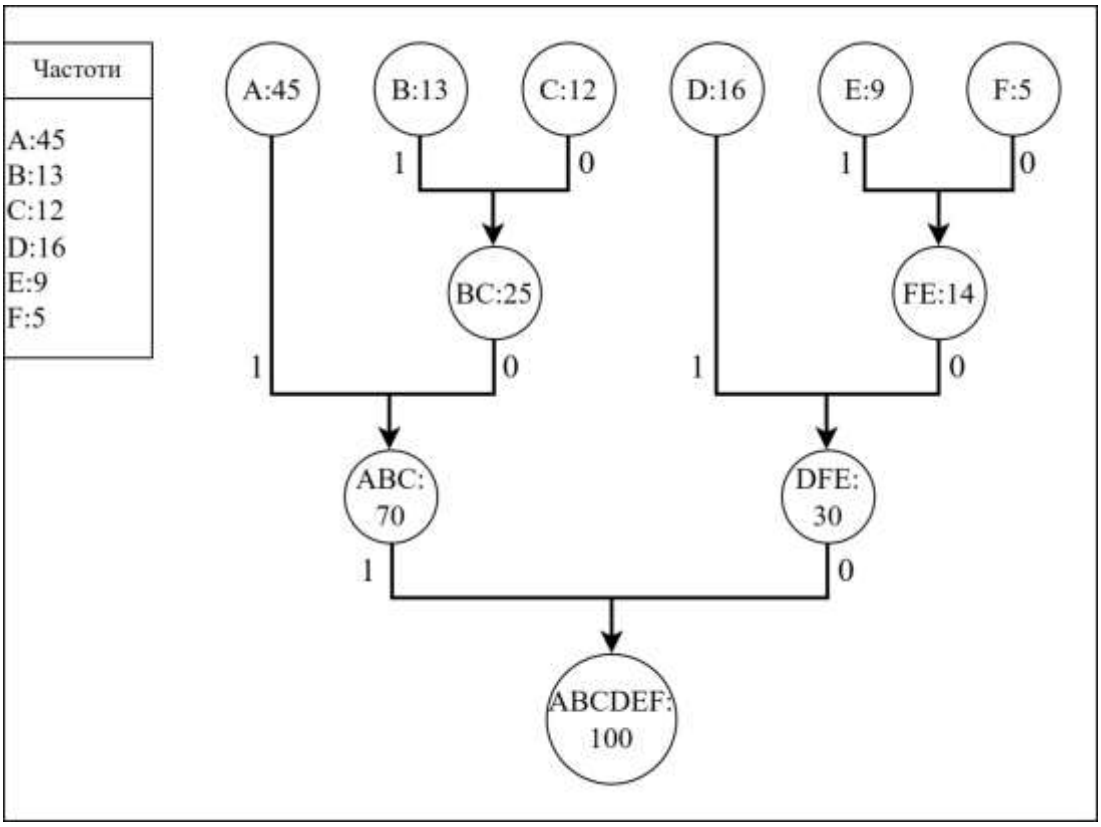

Рис. 2.1. Приклад кодування алгоритмом Хаффмана

Нижче наведено концептуальну реалізацію алгоритму Хаффмана на мові C#.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

class HuffmanCompressor
{
    // Клас вузла дерева Хаффмана
    class Node
    {
        public byte? Symbol;          // Символ (байт)
        public int Frequency;         // Частота
        public Node Left;             // Ліва гілка
        public Node Right;            // Права гілка
    }

    // --- Основна функція ---
    public static string CompressFileHuffman(string inputFile)
    {
        // 1. Зчитування файлу
        byte[] data = File.ReadAllBytes(inputFile);
```

```
        // 2. Підрахунок частот
        Dictionary<byte, int> freq = new Dictionary<byte,
int>();
        foreach (var b in data)
        {
            if (!freq.ContainsKey(b)) freq[b] = 0;
            freq[b]++;
        }

        // 3. Побудова дерева Хаффмана
        List<Node> nodes = freq.Select(f => new Node { Symbol =
f.Key, Frequency = f.Value }).ToList();

        while (nodes.Count > 1)
        {
            var ordered = nodes.OrderBy(n =>
n.Frequency).ToList();
            Node left = ordered[0];
            Node right = ordered[1];

            Node parent = new Node
            {
                Symbol = null,
                Frequency = left.Frequency + right.Frequency,
                Left = left,
                Right = right
            };

            nodes.Remove(left);
            nodes.Remove(right);
            nodes.Add(parent);
        }

        Node root = nodes.First();

        // 4. Створення таблиці кодів
        Dictionary<byte, string> codeTable = new
Dictionary<byte, string>();
        BuildCodeTable(root, "", codeTable);

        // 5. Кодування даних
        StringBuilder encodedBits = new StringBuilder();
        foreach (var b in data)
            encodedBits.Append(codeTable[b]);

        // 6. Додавання "паддінгу" (доповнення нулями)
        int padding = 8 - (encodedBits.Length % 8);
        if (padding != 8)
            encodedBits.Append('0', padding);

        // 7. Перетворення бітів у байти
        byte[] compressedBytes = new byte[encodedBits.Length /
8];
```

```
        for (int i = 0; i < compressedBytes.Length; i++)
        {
            string byteStr = encodedBits.ToString(i * 8, 8);
            compressedBytes[i] = Convert.ToByte(byteStr, 2);
        }

        // 8. Формування нового імені файлу
        string directory = Path.GetDirectoryName(inputFile);
        string                    fileName                    =
Path.GetFileNameWithoutExtension(inputFile);
        string extension = Path.GetExtension(inputFile);
        string     outputFile     =     Path.Combine(directory,
$"{fileName}_huffman{extension}");

        // 9. Записуємо результат у новий файл
        using    (var    fs    =    new    FileStream(outputFile,
FileMode.Create))
        {
            fs.WriteByte((byte)padding);   //   перший   байт   -
кількість доданих бітів
            fs.Write(compressedBytes,                          0,
compressedBytes.Length);
        }

        // 10. Інформація про результат
        Console.WriteLine($"Оригінальний    файл:    {inputFile}
({data.Length} байт)");
        Console.WriteLine($"Стиснутий     файл:     {outputFile}
({compressedBytes.Length + 1} байт)");

        return outputFile;
    }

    // --- Рекурсивне створення таблиці кодів ---
    private static void BuildCodeTable(Node node, string code,
Dictionary<byte, string> table)
    {
        if (node.Symbol != null)
        {
            table[node.Symbol.Value] = code;
        }
        else
        {
            BuildCodeTable(node.Left, code + "0", table);
            BuildCodeTable(node.Right, code + "1", table);
        }
    }

    // --- Точка входу для тестування ---
    static void Main()
    {
        string file = "example.txt";
        CompressFileHuffman(file);
```

```
    }
}
```

Клас HuffmanCoder включає внутрішній вузол HuffmanNode, метод BuildTree для побудови бінарного дерева на основі частот символів, метод GenerateCodes для отримання таблиці кодів символ→рядок бітів, метод Encode для кодування байтового масиву у стиснений потік із заголовком (довжина оригінального масиву, масив частот, padding), та метод Decode для відновлення оригінального масиву з бінарного потоку. Реалізація передбачає ефективну обробку padding, створення префіксного дерева декодування та перевірку цілісності даних при декодуванні.

2.3. Алгоритм кодування LZ77

Алгоритм LZ77, запропонований Абрагамом Лемпелем та Джейкобом Зівом у 1977 році, є одним із базових методів стиснення даних без втрат. Основна ідея алгоритму полягає у кодуванні повторюваних послідовностей символів повідомлення за допомогою словника, що реалізується у вигляді ковзного буфера. Стандартний алгоритм LZ77 представляє повторювану послідовність у вигляді трьох значень: відступу (offset), довжини збігу (length of match) та окремого символу (next symbol) [14].

Під час проходу повідомлення для кожної підпослідовності символів $x[i]..x[j]$, де i, j – індекси символів, алгоритм перевіряє наявність такої послідовності у буфері [14, 15]. У разі знаходження збігу перевіряється наявність розширеної послідовності $x[i]..x[j+1]$, і так процес продовжується доти, поки для наступного символу збіг не буде знайдено. Коли збіг припиняється, визначається відступ ссс від кінця ковзного буфера до початку найближчої послідовності, що збігається з $x[i]..x[j]$, а також довжина збігу l. Після цього в кінець буфера записується послідовність

$x[i]..x[j+1]$, а до списку збігів додаються три значення: *c*, *l* та символ $x[j+1]$.

Якщо збіг не знайдено, значення *c* та *l* вважаються рівними нулю. Така ситуація можлива лише у випадку, коли підпослідовність складається з одного символу, тобто *i=j*. У цьому випадку символ безпосередньо записується в кінець буфера. Алгоритм повторюється циклічно до моменту, поки не буде оброблено всі символи повідомлення. Варто зазначити, що якщо частина послідовності збігається з послідовністю у буфері, а інша частина міститься на початку повідомлення, її дозволяється не розривати. При декодуванні така нерозривна послідовність відновлюється точно так само, як і розірвана, що забезпечує правильність відновлення даних [14-16].

Декодування даних, стиснених алгоритмом LZ77, виконується у зворотному порядку. Для кожного триплету $(c, l, x[j])$ у буфер записується послідовність символів, що починається на *c*-му елементі від кінця буфера та продовжується на *l* символів, після чого дописується символ *x*[*j*]. У разі, коли *l>c*, послідовність продовжується циклічно з урахуванням щойно доданих символів до буфера, що дозволяє коректно відновлювати довгі повторювані підрядки, навіть якщо вони перекривають раніше закодовані сегменти [16].

Розглянемо процес роботи алгоритму LZ77 на прикладі рядка "abcabdabdabf". Припустимо, що розмір ковзного вікна (буфера) становить 5 символів.

На початковому етапі буфер порожній, тому для першого символу "a" збігів не знайдено. Відповідно, до вихідного потоку записується триплет (0, 0, "a"), а символ додається до буфера. Аналогічні дії виконуються для наступних символів "b" і "c", утворюючи (0, 0, "b") та (0, 0, "c"). Після цього в буфері міститься послідовність ("a", "b", "c").

Коли алгоритм досягає четвертого та п’ятого символів "a" і "b", виявляється, що в буфері вже є така послідовність. Визначається відстань до початку збігу – c = 3 символи від кінця буфера, а довжина збігу дорівнює l = 2. Наступним символом після збігу є "d", тому до закодованого виходу

записується триплет (3, 2, "d"). Після цього до буфера додаються символи "a", "b", "d", у результаті чого він набуває вигляду ("a", "b", "c", "a", "b", "d").

Далі, для сьомого символа "a", алгоритм знаходить частковий збіг довжиною три символи, починаючи з третьої позиції з кінця буфера. Однак, з урахуванням принципу циклічного повторення, цей збіг можна подовжити до п’яти символів, тобто "abdab". Наступним символом є "f", який не має збігів у буфері. Таким чином, формується триплет (3, 5, "f"), який завершує процес кодування.

Отже, після проходження всього повідомлення отримаємо послідовність:

$$\{(0,0,"a"),(0,0,"b"),(0,0,"c"),(3,2,"d"),(3,5,"f")\}.$$

Для подальшого збереження кожен триплет кодується у вигляді набору байтів. Відстань (offset) зазвичай представлена типом ushort (2 байти), довжина збігу (length) – також 2 байти, а символ (symbol) – одним байтом. Таким чином, кожен триплет займає чотири байти, що дозволяє зменшити обсяг вихідних даних за рахунок усунення надлишкових фрагментів.

На рис. 2.2 наведено схематичне зображення етапів кодування повідомлення за алгоритмом LZ77. На ньому показано, як формуються триплети (offset, length, symbol) і як відбувається поступове оновлення ковзного буфера, що забезпечує можливість відновлення початкового тексту під час декодування.

Процес декодування у методі LZ77 здійснюється у напрямку, зворотному до кодування, але з використанням спільного буфера, який поступово заповнюється відновленими символами. На початку роботи декодер обробляє перші три триплети, у яких значення зсуву c і довжини збігу l дорівнюють нулю. Це означає, що дані символи не мають відповідності у буфері, тому вони безпосередньо записуються у вихідну послідовність. Після цього в буфері послідовно накопичуються символи "a", "b" та "c".

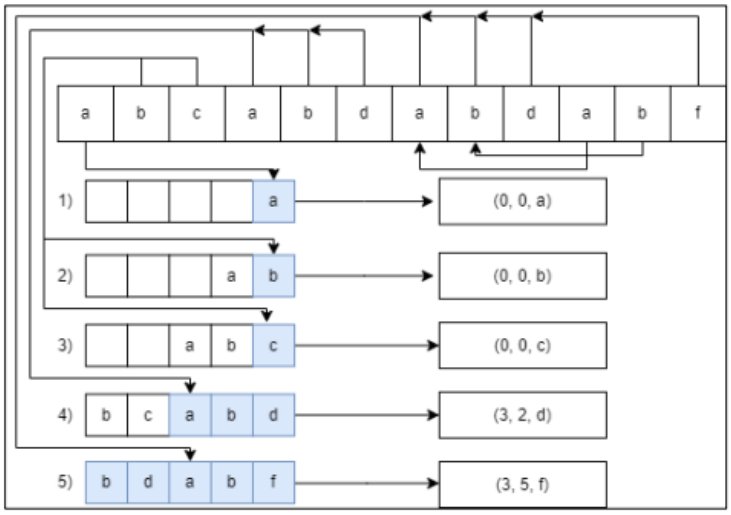

Рис. 2.2. Процес кодування методом LZ77

Під час обробки четвертого триплета (3, 2, "d") декодер звертається до третього символу з кінця буфера, тобто "a", і копіює два символи поспіль – "a" та "b". Після цього до відновленої послідовності додається символ "d". Таким чином, після четвертого кроку отримуємо проміжний результат "abcabd".

Для п’ятого триплета (3, 5, "f") процес відтворення є дещо складнішим. Згідно з вказаними параметрами, з третьої позиції від кінця буфера починається копіювання п’яти символів. При цьому, коли кількість копій перевищує розмір поточного буфера, копіювання здійснюється циклічно – нещодавно додані символи одразу стають доступними для подальшого використання. У результаті формується послідовність "abdab", після якої додається символ "f".

Зібравши всі відновлені частини, отримаємо остаточне повідомлення "abcabdabdabf", яке повністю збігається з початковим вхідним рядком до стиснення. Цей приклад демонструє, що алгоритм LZ77 є оборотним і забезпечує точне відтворення оригінальних даних без втрат.

На рис. 2.2 показано схематичне зображення процесу декодування, де проілюстровано, як відбувається послідовне копіювання підрядків із буфера

та додавання нових символів, що забезпечує повну реконструкцію початкового повідомлення.

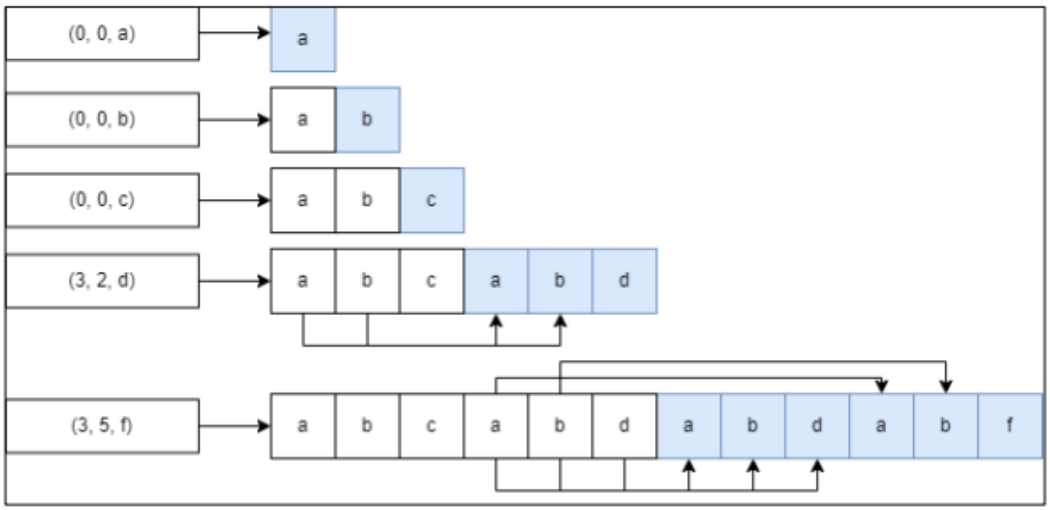

Рис. 2.3. Процес декодування методом LZ77

Нижче, представлена функціональна реалізація методу LZ77, що забезпечує стиснення та відновлення довільних типів файлів шляхом аналізу послідовності байтів у вхідному потоці. Алгоритм базується на принципі пошуку повторюваних підрядків у межах ковзного вікна фіксованого розміру. Для кожного повтору формується трійка параметрів (offset, length, nextByte), що описує зміщення до початку знайденої підпослідовності, її довжину та наступний символ, який не входить до повтору.

Під час кодування вхідний файл зчитується у вигляді байтового масиву за допомогою методу File.ReadAllBytes(). У процесі ітерації здійснюється пошук найдовшого збігу між поточним сегментом даних та попередньою частиною буфера. Результат кодування записується до вихідного файлу у вигляді послідовності трійок, що містять два числові параметри (зміщення та довжину) і байт наступного символу.

Декодування здійснюється у зворотному порядку: для кожного запису (offset, length, nextByte) зчитується відповідна кількість байтів із уже відновленого буфера, після чого додається новий символ. Завдяки цьому відтворюється початкова послідовність даних без втрати інформації.

```
using System;
using System.Collections.Generic;
using System.IO;

class LZ77
{
    const int WindowSize = 4096;
    const int LookaheadBufferSize = 18;
    // Функція для кодування LZ77
    public static List<(int offset, int length, byte next)>
Encode(byte[] input)
    {
        List<(int, int, byte)> encoded = new List<(int, int,
byte)>();
        int i = 0;
        while (i < input.Length)
        {
            int matchLength = 0;
            int matchDistance = 0;
            int startWindow = Math.Max(0, i - WindowSize);
            int   maxLength   =   Math.Min(LookaheadBufferSize,
input.Length - i);
            for (int j = startWindow; j < i; j++)
            {
                int length = 0;
                while (length < maxLength &&
                       input[j + length] == input[i + length])
                {
                    length++;
                    if (j + length >= i) break;
                }
                if (length > matchLength)
                {
                    matchLength = length;
                    matchDistance = i - j;
                }
            }
            byte nextByte = (i + matchLength < input.Length) ?
input[i + matchLength] : (byte)0;
            encoded.Add((matchDistance, matchLength, nextByte));
            i += matchLength + 1;
        }
        return encoded;
    }
    // Функція для декодування LZ77
    public static byte[] Decode(List<(int offset, int length,
byte next)> encoded)
    {
        List<byte> output = new List<byte>();
        foreach (var (offset, length, next) in encoded)
        {
            int start = output.Count - offset;
            for (int i = 0; i < length; i++)
```

```
            {
                output.Add(output[start + i]);
            }
            output.Add(next);
        }
        return output.ToArray();
    }
    public static void CompressFile(string inputPath, string
outputPath)
    {
        byte[] data = File.ReadAllBytes(inputPath);
        var encoded = Encode(data);
        using (BinaryWriter writer = new
BinaryWriter(File.Open(outputPath, FileMode.Create)))
        {
            foreach (var (offset, length, next) in encoded)
            {
                writer.Write((ushort)offset);
                writer.Write((byte)length);
                writer.Write(next);
            }
        }
        Console.WriteLine($"Файл {inputPath} стиснуто методом
LZ77 → {outputPath}");
    }
    public static void DecompressFile(string inputPath, string
outputPath)
    {
        List<(int, int, byte)> encoded = new List<(int, int,
byte)>();
        using (BinaryReader reader = new
BinaryReader(File.Open(inputPath, FileMode.Open)))
        {
            while (reader.BaseStream.Position <
reader.BaseStream.Length)
            {
                int offset = reader.ReadUInt16();
                int length = reader.ReadByte();
                byte next = reader.ReadByte();
                encoded.Add((offset, length, next));
            }
        }
        byte[] decoded = Decode(encoded);
        File.WriteAllBytes(outputPath, decoded);
        Console.WriteLine($"Файл {inputPath} розпаковано методом
LZ77 → {outputPath}");
    }
    static void Main()
    {
        string input = "test.txt";
        string compressed = "test.lz77";
        string output = "decoded.txt";
        CompressFile(input, compressed);
```

```
            DecompressFile(compressed, output);
        }
    }
```

Особливістю реалізації є незалежність від типу вхідного файлу – алгоритм оперує лише байтами, що дозволяє застосовувати його як до текстових (.txt), так і до графічних (*.bmp, .png, .jpg) чи двійкових (.bin, .exe) даних.

Метод LZ77 є ефективним для даних, які містять значну кількість повторюваних фрагментів, однак його ефективність зменшується при обробці вже стиснутих форматів (JPEG, ZIP тощо), де відсутня статистична надлишковість. Таким чином, алгоритм LZ77 широко застосовується у різних системах стиснення, включаючи формати ZIP, GZIP та PNG, завдяки простоті реалізації та високій ефективності при обробці текстових, бінарних та мультимедійних даних. Його головною перевагою є здатність досягати значного ступеня стиснення при збереженні всіх вихідних даних без втрат, що робить його універсальним інструментом у сучасних комп’ютерних системах обробки інформації [14].

2.4. Алгоритм кодування LZW

Алгоритм кодування LZW (Lempel–Ziv–Welch) є вдосконаленням попередніх методів стиснення, таких як LZ77 і LZ78, і базується на ідеї побудови динамічного словника під час процесу кодування. Основна перевага цього підходу полягає в тому, що той самий словник використовується як для стиснення, так і для декодування, тому немає необхідності передавати його разом із закодованими даними. На початковому етапі створюється словник, до якого заносяться всі можливі символи, що можуть зустрічатися у вхідному потоці. Наприклад, для роботи з латинським алфавітом у словнику будуть усі 26 літер від A до Z, а також

символи пробілу, розділових знаків і кінець рядка. Кожен із них має свій унікальний числовий код [15-18].

Процес кодування відбувається таким чином. Із вхідного потоку зчитується перший символ, який ініціалізується як поточна фраза W. Потім зчитується наступний символ K і перевіряється, чи існує послідовність WK у словнику. Якщо така послідовність уже є, тоді фраза W розширюється – тепер вона дорівнює WK, і зчитується наступний символ. Якщо ж послідовність WK відсутня у словнику, то код для W записується до вихідного потоку, нова послідовність WK додається до словника з новим унікальним кодом, а фраза W приймає значення K. Цей цикл повторюється доти, доки не буде оброблено весь вхідний потік символів. Завдяки такому підходу алгоритм поступово накопичує інформацію про повторювані послідовності та замінює їх коротшими числовими кодами.

Розглянемо приклад роботи алгоритму на рядку «ANANASBANANA». На початку словник містить усі однолітерні символи – A, B, N, S тощо. Під час зчитування символів створюються нові фрази AN і NA, яких немає у словнику, і вони додаються туди з новими індексами. Далі, при повторній появі цих фраз у тексті, замість букв записуються їхні коди зі словника. У результаті алгоритм замінює повторювані послідовності на коротші коди, завдяки чому досягається стиснення (рис. 2.4). Якщо вихідний рядок містив 12 символів, то після стиснення кількість кодів може становити, наприклад, лише 8. Таким чином, коефіцієнт стиснення дорівнює 12/8 = 1,5, що свідчить про зменшення обсягу даних без втрати інформації.

Перевага методу LZW полягає в тому, що ефективність стиснення зростає зі збільшенням довжини рядка, оскільки словник збагачується новими, довшими послідовностями. Проте слід зазначити, що розмір словника також збільшується, а отже, з часом може знадобитися більше бітів для кодування кожного нового запису. Деякі реалізації алгоритму передбачають можливість очищення словника, видаляючи фрази, що рідко використовуються, аби звільнити місце для нових комбінацій символів.

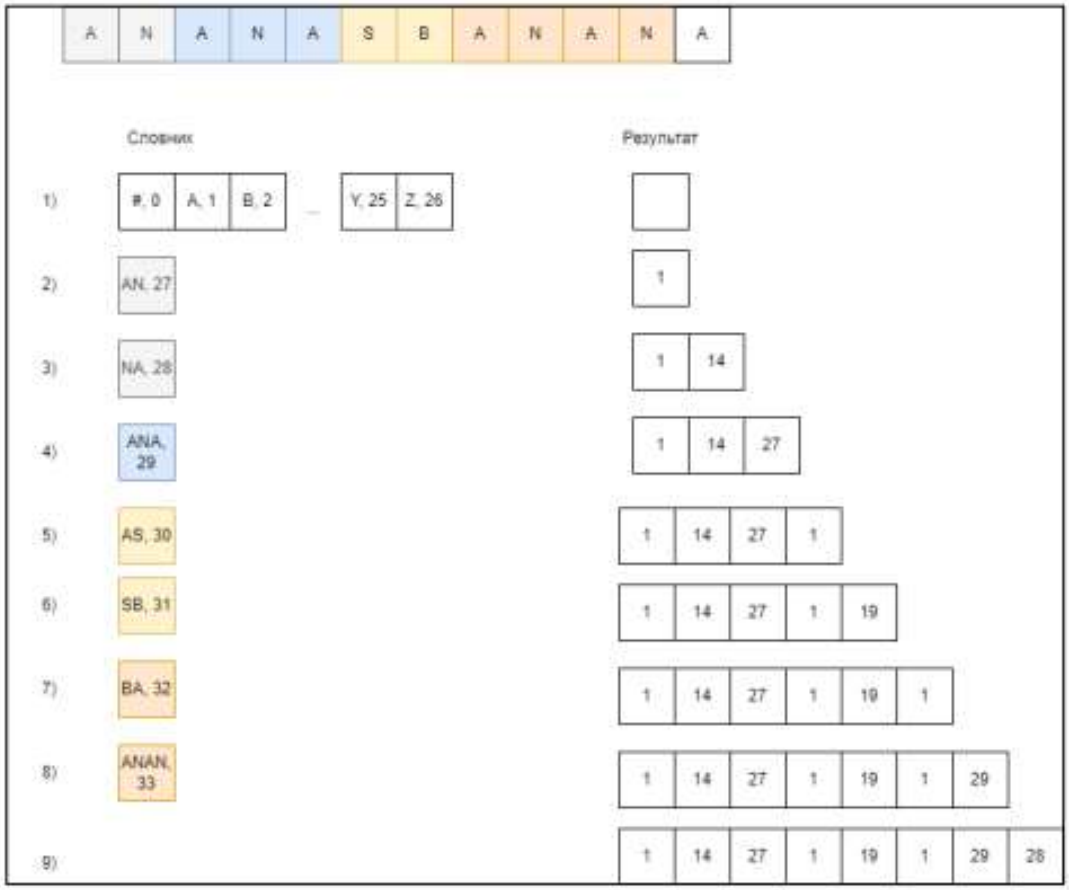

Рис. 2.4. Стиснення даних алгоритмом LZW

Розпакування даних, закодованих методом LZW, відбувається за аналогічним принципом (рис. 2.5). Для декодування спочатку також створюється словник із усіма можливими однолітерними символами. Далі послідовно зчитуються коди зі стисненого потоку, і за кожним кодом визначається відповідна фраза зі словника. Якщо фраза для певного коду ще не визначена, алгоритм використовує попередню відому фразу, до якої додається перший символ поточної фрази. Таким чином, під час декодування словник відтворюється у тій самій послідовності, що і під час стиснення, забезпечуючи точне відновлення початкового повідомлення.

Алгоритм LZW вважається одним із найефективніших безвтратних методів стиснення, особливо для текстових даних або даних із повторюваними структурами. Він широко використовується у форматах

зображень, таких як GIF і TIFF, а також у деяких архівних форматах. Для прикладу, у форматі GIF алгоритм LZW дозволяє суттєво зменшити розмір графічних файлів без погіршення якості зображення, завдяки великій кількості повторюваних кольорових послідовностей [18].

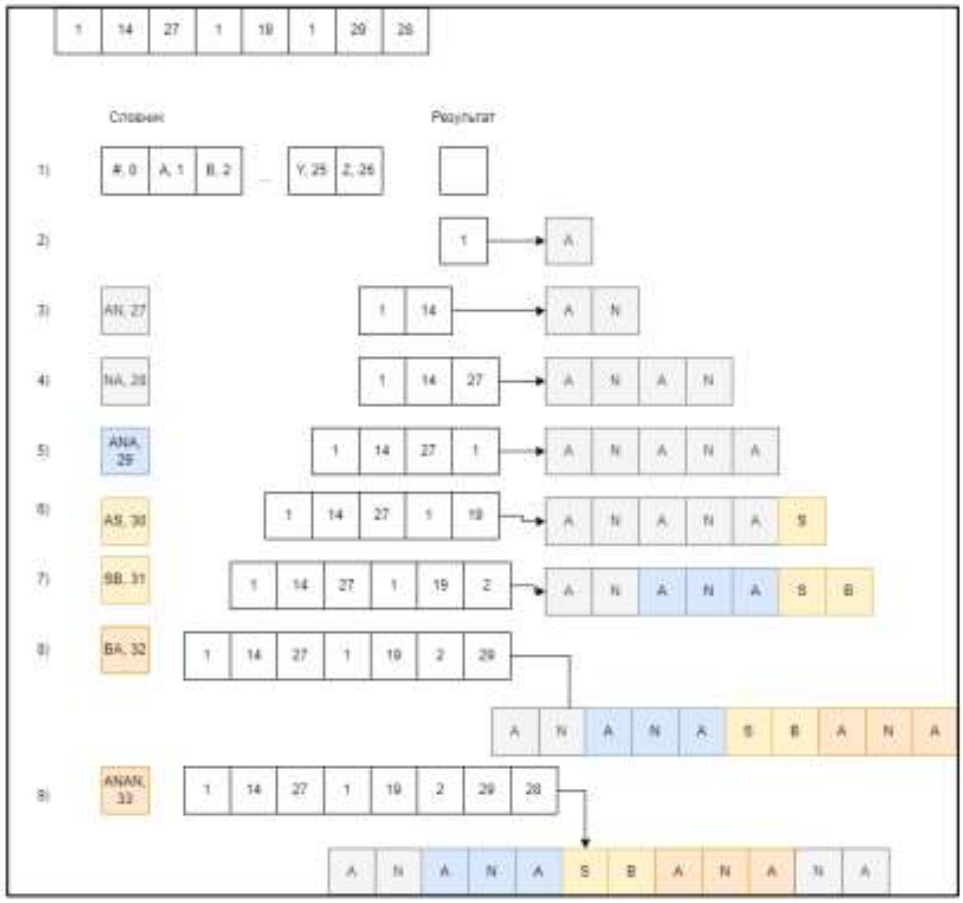

Рис. 2.5. Розпакування даних алгоритмом LZW

У порівнянні з алгоритмом LZ77, метод LZW не використовує «ковзне вікно», а натомість покладається на централізований словник, що спрощує реалізацію і прискорює процес стиснення. Завдяки жадібній стратегії зчитування символів і поступовому розширенню словника, алгоритм працює швидко та забезпечує прийнятну ефективність навіть на потокових даних. Проте у деяких випадках, наприклад при обробці випадкових або вже стиснутих даних (як у форматах ZIP чи JPEG), LZW може не дати

позитивного ефекту, а іноді навіть збільшити розмір файлу через необхідність зберігати нові записи у словнику [16, 18].

Важливим аспектом практичної реалізації LZW є управління розміром словника та бітовою довжиною кодів. На початку кожен символ кодується фіксованою кількістю бітів, але зі зростанням словника потрібно більше бітів для представлення нових кодів. Наприклад, при початкових 256 записах у словнику достатньо 8 бітів, але при подальшому розширенні може знадобитися 9, 10 або більше. Для цього реалізації LZW часто використовують адаптивне розширення коду, де довжина бітів збільшується автоматично при досягненні певної кількості записів.

Стиснення методом LZW є детермінованим, тобто однакові вхідні дані завжди дають однаковий вихідний потік, що забезпечує високу стабільність і точність результатів. Крім того, алгоритм не потребує попереднього аналізу вхідних даних, що робить його придатним для роботи в реальному часі. Його простота та надійність зумовили широке застосування в інформаційних системах, засобах архівації, передачі даних і мультимедіа.

Розроблена програма реалізує алгоритм стиснення та відновлення даних за методом LZW і призначена для дослідження ефективності цього алгоритму при роботі з файлами різних форматів. Програмне забезпечення написано мовою програмування C#, що забезпечує високу продуктивність, керування пам’яттю та підтримку об’єктно-орієнтованого підходу. У розробці використано стандартні бібліотеки .NET, що дозволяють працювати безпосередньо з байтовими потоками, файлами та структурами даних, необхідними для реалізації словникового методу кодування.

Основна ідея роботи програми полягає у побудові динамічного словника, який формується під час обробки вхідних даних. На етапі стиснення програма зчитує вхідний файл побайтово, створюючи первинний словник, який містить усі можливі однобайтові комбінації, тобто символи з кодами від 0 до 255. У процесі кодування алгоритм поступово об’єднує символи у довші фрази, додаючи їх до словника за умови, що така комбінація

ще не зустрічалася. Кожна фраза замінюється на свій унікальний код, який записується у вихідний файл. Таким чином, вихідні дані перетворюються на послідовність числових кодів, що представляють повторювані фрагменти у стислому вигляді.

Під час декодування програма виконує обернену процедуру. Вона ініціалізує словник аналогічно до процесу стиснення, зчитує закодовані коди з файлу та поступово відтворює оригінальні фрази. У момент, коли програма зустрічає новий код, вона визначає його зміст, використовуючи попередньо збережені комбінації у словнику. Якщо код відсутній у словнику, алгоритм формує його, додаючи перший символ попередньої фрази до вже відомої послідовності. Завдяки такому підходу декодування виконується без втрат інформації, що гарантує точне відновлення початкових даних.

Нижче наведено лістинг коду алгоритму LZW.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace LZWCompression
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = System.Text.Encoding.UTF8;
            Console.WriteLine("=== Алгоритм стиснення LZW ===");
            Console.Write("Введіть шлях до файлу для стиснення:
");
            string inputPath = Console.ReadLine();
            if (!File.Exists(inputPath))
            {
                Console.WriteLine("Файл не знайдено!");
                return;
            }
            string compressedPath = inputPath + ".lzw";
            string decompressedPath = inputPath + "_decoded" +
Path.GetExtension(inputPath);
            Console.WriteLine("\n Стиснення...");
            LZWEncode(inputPath, compressedPath);
            Console.WriteLine($" Файл стиснуто:
{compressedPath}");
            Console.WriteLine("\n Відновлення...");
            LZWDecode(compressedPath, decompressedPath);
```

```
            Console.WriteLine($" Файл відновлено:
{decompressedPath}");
        }
        /// <summary>
        /// Стиснення файлу методом LZW
        /// </summary>
        static void LZWEncode(string inputPath, string
outputPath)
        {
            byte[] inputBytes = File.ReadAllBytes(inputPath);
            Dictionary<string, int> dictionary = new
Dictionary<string, int>();
            for (int i = 0; i < 256; i++)
                dictionary.Add(((char)i).ToString(), i);
            string w = string.Empty;
            List<int> compressed = new List<int>();
            int dictSize = 256;
            foreach (byte b in inputBytes)
            {
                char c = (char)b;
                string wc = w + c;
                if (dictionary.ContainsKey(wc))
                {
                    w = wc;
                }
                else
                {
                    compressed.Add(dictionary[w]);
                    dictionary[wc] = dictSize++;
                    w = c.ToString();
                }
            }
            // Додаємо код останньої фрази
            if (!string.IsNullOrEmpty(w))
                compressed.Add(dictionary[w]);
            // Запис стиснених даних у файл
            using (BinaryWriter writer = new
BinaryWriter(File.Open(outputPath, FileMode.Create)))
            {
                foreach (int code in compressed)
                    writer.Write(code);
            }
        }
        /// <summary>
        /// Розпакування файлу методом LZW
        /// </summary>
        static void LZWDecode(string inputPath, string
outputPath)
        {
            List<int> compressed = new List<int>();
            using (BinaryReader reader = new
BinaryReader(File.Open(inputPath, FileMode.Open)))
            {
```

```
                while (reader.BaseStream.Position <
reader.BaseStream.Length)
                    compressed.Add(reader.ReadInt32());
            }
            // Ініціалізація словника
            Dictionary<int, string> dictionary = new
Dictionary<int, string>();
            for (int i = 0; i < 256; i++)
                dictionary[i] = ((char)i).ToString();
            string w = dictionary[compressed[0]];
            compressed.RemoveAt(0);
            List<byte> outputBytes = new
List<byte>(System.Text.Encoding.UTF8.GetBytes(w));
            int dictSize = 256;
            foreach (int k in compressed)
            {
                string entry;
                if (dictionary.ContainsKey(k))
                    entry = dictionary[k];
                else if (k == dictSize)
                    entry = w + w[0];
                else
                    throw new Exception("Помилка декодування:
невідомий код.");

outputBytes.AddRange(System.Text.Encoding.UTF8.GetBytes(entry));
                dictionary[dictSize++] = w + entry[0];
                w = entry;
            }
            File.WriteAllBytes(outputPath,
outputBytes.ToArray());
        }
    }
}
```

Реалізація алгоритму використовує структури даних типу Dictionary, які забезпечують швидкий доступ до елементів за ключем, що є критично важливим для роботи LZW. На етапі кодування словник має тип Dictionary<string, int>, де ключем виступає рядок символів, а значенням – його числовий код. Під час декодування використовується зворотна структура Dictionary<int, string>, що дозволяє відновлювати рядки за їх кодами. Це дає змогу досягти високої швидкодії та ефективності обробки великих обсягів даних.

Програма побудована у вигляді консольного застосунку, який взаємодіє з користувачем через введення та виведення текстових

повідомлень. Користувачеві пропонується вказати шлях до файлу, який потрібно стиснути. Після завершення процесу стиснення створюється новий файл із розширенням .lzw, що містить послідовність закодованих чисел. Далі програма автоматично виконує декодування, зчитуючи створений файл і генеруючи новий, відновлений файл із початковим розширенням та додаванням позначки _decoded до назви. Такий підхід дозволяє перевірити точність роботи алгоритму шляхом порівняння розміру та вмісту вихідного та відновленого файлів.

Програмна реалізація орієнтована на роботу з будь-якими типами файлів, оскільки обробка виконується на рівні байтів, а не символів. Це означає, що алгоритм може бути застосований як до текстових файлів (наприклад, .txt або .docx), так і до графічних (.bmp, .png), звукових (.wav, .mp3), відео (.avi), виконуваних (.exe) чи інших двійкових форматів. Така універсальність робить програму придатною для дослідження різних типів даних і порівняння ефективності стиснення залежно від структури вхідної інформації.

Робота алгоритму LZW має певні особливості. Ефективність стиснення зростає при збільшенні довжини вхідних даних, оскільки словник поступово збагачується новими комбінаціями символів, що дозволяє замінювати повторювані послідовності коротшими кодами. Для коротких або випадкових послідовностей, а також для файлів, які вже пройшли стиснення іншими алгоритмами (наприклад, JPEG або ZIP), коефіцієнт стиснення може бути незначним або навіть негативним. Однак при роботі з файлами, які містять багато повторюваних шаблонів, зокрема текстами або бітовими зображеннями, метод LZW демонструє високі показники ефективності [25].

Загалом, алгоритм LZW можна вважати компромісом між простотою реалізації, швидкодією та ефективністю. Його головна ідея полягає в поступовому навчанні словника під час обробки даних, що дозволяє стискати навіть ті послідовності, які спочатку не були передбачені. Саме завдяки цьому принципу LZW став одним із найвідоміших і найпоширеніших методів

безвтратного кодування, що залишається актуальним навіть у сучасних системах обробки та зберігання інформації.

2.5. Алгоритм кодування RLE

Алгоритм кодування довжин серій Run-Length Encoding (RLE) є одним із найпростіших і водночас найстаріших методів стиснення даних без втрат, який застосовується для зменшення обсягу інформації, що містить послідовності однакових елементів. Основна ідея цього методу полягає в тому, щоб замінити серію повторюваних символів або байтів одним екземпляром символу та числом, яке вказує кількість його повторів. Такий підхід особливо ефективний у випадках, коли в потоці даних є великі однорідні ділянки – наприклад, у растрових зображеннях з великими одноколірними областями або у простих текстових файлах, де часто повторюються однакові символи [22].

Математично принцип роботи RLE можна описати так. Нехай вхідна послідовність даних подається як

$$S = \{s_1, s_2, s_3, \ldots, s_n\},$$

де s_i – це елементи (символи або байти) даних. Якщо в потоці є підпослідовність $s_i = s_{i+1} = \ldots = s_{i+k-1}$, тобто k однакових елементів поспіль, то замість збереження кожного з них у закодованому потоці записується пара (s_i, k), де s_i – сам символ, а k – кількість його повторів. Таким чином, замість k байтів зберігається лише два елементи – символ і лічильник, що значно скорочує обсяг інформації за умови великих серій повторів.

Наприклад, нехай маємо послідовність символів

AAAABBCCCCCAA,

яка складається з повторів. Після кодування за допомогою RLE отримаємо структуру:

(4A)(2B)(5C)(2A).

У такому випадку замість 13 символів ми зберігаємо лише 8 (4 числа і 4 символи). Коефіцієнт стиснення визначається як

$$K = \frac{n}{m},$$

де n – довжина вихідного повідомлення, m – довжина закодованого повідомлення. Якщо $K > 1$, алгоритм забезпечує стиснення даних.

Попри свою простоту, RLE має як переваги, так і обмеження. Основна перевага полягає в низькій обчислювальній складності: кодування та декодування відбувається лінійно за кількістю символів $O(n)$, що робить алгоритм надзвичайно швидким і придатним для реалізації навіть у вбудованих системах. Проте ефективність стиснення значною мірою залежить від структури даних. Якщо символи змінюються часто, тобто серії мають малу довжину, то обсяг стисненого файлу може навіть перевищити початковий. Наприклад, послідовність

ABCDEFG

буде закодована як

(1A)(1B)(1C)(1D)(1E)(1F)(1G),

що збільшить кількість даних майже вдвічі, адже до кожного символу додається числовий маркер довжини.

Для підвищення ефективності алгоритму RLE у практичних реалізаціях використовують різні модифікації. Одна з них полягає у використанні спеціального маркерного байта, який вказує на початок серії. Наприклад, якщо маркером є символ ESC, тоді послідовність ESC ESC ESC ESC можна закодувати як ESC 4 ESC. Це дозволяє уникнути плутанини між даними та службовими символами. Ще один підхід полягає у використанні позитивних чисел для позначення довжин повторів і від’ємних – для послідовностей унікальних символів. Тоді, наприклад, рядок

BBBAACDDDD

можна подати як

3B2A1C4D,

або, у більш розширеній нотації з від’ємними індексами:

$$-1C3B2A4D,$$

що полегшує розпізнавання одноразових символів.

Формально процес кодування можна записати як послідовність операцій. Для кожного символу s_i перевіряється умова $i = s_{i+1}$. Якщо вона виконується, лічильник r збільшується на одиницю, і перевірка триває, доки символи збігаються. Коли послідовність обривається або досягає кінця, записується пара (s_i, r), після чого лічильник обнуляється. Декодування здійснюється у зворотному порядку – кожна пара розгортається у відповідну кількість однакових символів:

$$(s_i, r) \to s_i s_i s_i \ldots s_i \ (r \ разів).$$

З точки зору математичної ефективності, оптимальність RLE можна оцінити за формулою відношення обсягу закодованого повідомлення до початкового:

$$E = \frac{b_c}{b_o} = \frac{k + 2r}{n},$$

де b_c – кількість байтів після кодування, b_o – кількість байтів у вихідному потоці, k – кількість унікальних серій, r – кількість повторів. Якщо $E<1$, алгоритм є ефективним.

На практиці алгоритм RLE активно використовується у форматах зображень, де є великі області однакового кольору. Наприклад, у старих графічних форматах BMP, PCX або TIFF застосовувалася саме ця схема. Для двійкових зображень, де пікселі можуть мати лише значення 0 або 1, RLE дає особливо високі результати. Якщо, наприклад, маємо рядок бінарного зображення:

$$11111111000000000000011111,$$

то його можна закодувати як

$$(8,1)(14,0)(5,1),$$

що в декілька разів скорочує кількість необхідних байтів для зберігання.

Однак у загальному випадку ефективність RLE обмежується тим, що довжини серій можуть бути короткими. Тому в сучасних архіваторах і кодеках його зазвичай поєднують з іншими методами, наприклад, з алгоритмами Хаффмана або LZ77. У таких комбінаціях RLE виступає як попередній етап, що усуває надлишкові послідовності, після чого отриманий потік додатково стискається ентропійними методами.

Попри появу значно складніших методів, RLE залишається важливою частиною теорії стиснення даних. Його простота, лінійна складність і можливість апаратної реалізації роблять його незамінним у випадках, коли потрібна швидкість, а не максимальний коефіцієнт стиснення. У цифрових пристроях з обмеженими ресурсами, таких як принтери, сканери, мікроконтролери або сенсорні вузли Інтернету речей, саме RLE часто використовується для зберігання графічних або телеметричних даних. Таким чином, алгоритм кодування довжин серій є фундаментальним прикладом того, як прості математичні ідеї можуть забезпечити ефективне представлення інформації у практичних системах з обмеженими можливостями обчислення й пам’яті.

Нижче наведено реалізацію алгоритму RLE на мові C#.

```
using System;
using System.IO;
using System.Collections.Generic;

namespace RLE_Compression
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("===  RLE  Compression  Algorithm
===");

            Console.Write("Введіть шлях до файлу для стиснення:
");
            string inputPath = Console.ReadLine();

            if (!File.Exists(inputPath))
            {
                Console.WriteLine("Файл не знайдено!");
                return;
```

```
            }

            string compressedPath = inputPath + ".rle";
            string decompressedPath = inputPath + ".decoded";

            // Кодування файлу (стиснення)
            byte[] inputBytes = File.ReadAllBytes(inputPath);
            byte[] compressed = RLEEncode(inputBytes);
            File.WriteAllBytes(compressedPath, compressed);
            Console.WriteLine($"Файл                    стиснено:
{compressedPath}");

            // Декодування файлу (розпакування)
            byte[] decompressed = RLEDecode(compressed);
            File.WriteAllBytes(decompressedPath, decompressed);
            Console.WriteLine($"Файл                  відновлено:
{decompressedPath}");
        }

        // --- Алгоритм RLE ---
        public static byte[] RLEEncode(byte[] data)
        {
            List<byte> encoded = new List<byte>();

            for (int i = 0; i < data.Length; i++)
            {
                byte current = data[i];
                int count = 1;

                // Підрахунок кількості повторів одного байта
                while (i + 1 < data.Length && data[i + 1] ==
current && count < 255)
                {
                    count++;
                    i++;
                }

                encoded.Add((byte)count);
                encoded.Add(current);
            }

            return encoded.ToArray();
        }

        // --- Зворотне декодування ---
        public static byte[] RLEDecode(byte[] data)
        {
            List<byte> decoded = new List<byte>();

            for (int i = 0; i < data.Length; i += 2)
            {
                int count = data[i];
                byte value = data[i + 1];
```

```
                for (int j = 0; j < count; j++)
                {
                    decoded.Add(value);
                }
            }

            return decoded.ToArray();
        }
    }
}
```

Функція RLEEncode здійснює побайтове зчитування вхідного файлу, підраховуючи кількість повторів кожного байта. Оскільки один байт може зберігати значення від 0 до 255, максимально можлива довжина серії обмежена числом 255. Після підрахунку кожної серії у вихідний масив записується пара значень – кількість повторів та сам байт.

Функція RLEDecode виконує зворотну операцію – вона зчитує кожну пару байтів, де перше число відповідає кількості повторів, а друге – значенню байта, що повторюється. На основі цих даних функція відтворює початкову послідовність, розгортаючи закодовані серії до їхньої первісної форми.

Оскільки програма працює з байтами, використовуючи методи File.ReadAllBytes() та File.WriteAllBytes(), вона може обробляти будь-які типи файлів – від звичайних текстових (.txt) до зображень (.bmp, .png, .jpg) чи двійкових (.exe, .zip). Це робить алгоритм універсальним і незалежним від формату даних, адже він стискає не вміст файлу як текст, а його байтове представлення.

Водночас алгоритм має певні обмеження. Його ефективність помітна лише тоді, коли у файлі присутні повторювані байти, тобто серії однакових символів. Якщо ж дані є випадковими або вже стисненими іншими методами (наприклад, у форматах JPG, ZIP або MP3), застосування RLE може навіть збільшити розмір файлу. Це пояснюється тим, що для кожного байта додається службова інформація – лічильник повторів, що вносить додаткові байти до загального обсягу даних. Тому RLE найкраще підходить для файлів

із великою кількістю однорідних ділянок, наприклад для чорно-білих зображень, бітових масок або текстових даних із довгими послідовностями однакових символів.

2.4. Висновки до розділу 2

У другому розділі, в ході проведеного дослідження було детально розглянуто принципи роботи та особливості реалізації основних алгоритмів стиснення даних без втрат – Шеннона–Фано, Хаффмана, LZ77, LZW та RLE. Для кожного методу проаналізовано теоретичні основи, механізми побудови кодів, способи формування таблиць або словників, а також характерні переваги й недоліки в контексті їх практичного застосування у комп'ютерних системах.

Метод Шеннона–Фано ґрунтується на поділі множини символів за ймовірностями їх появи, що дозволяє отримати коди змінної довжини. Разом з тим, його ефективність обмежена через відсутність гарантованої оптимальності побудови коду.

Метод Хаффмана, на відміну від Шеннона–Фано, забезпечує оптимальне кодування, оскільки базується на побудові бінарного дерева частот. Його практичне застосування широко поширене у форматах архівації та мультимедійного стиснення (наприклад, JPEG, MP3), що підтверджує універсальність та ефективність даного підходу.

Алгоритми сімейства Lempel–Ziv, зокрема LZ77 і LZW, реалізують словникові принципи стиснення, що дозволяють уникати надлишкового зберігання повторюваних послідовностей. LZ77 працює з ковзним вікном і посиланнями на попередні підрядки, тоді як LZW формує динамічний словник без необхідності передавання таблиці кодів. Ці методи характеризуються високим ступенем стиснення для даних із частими повтореннями та ефективно застосовуються у форматах GIF, ZIP, PDF тощо.

Алгоритм RLE реалізує найпростіший підхід до стиснення шляхом заміни послідовностей повторюваних символів на їх числове представлення. Він є ефективним для даних із великими ділянками однорідної інформації, наприклад, для монохромних зображень або схемних структур, проте малоефективний для випадкових або сильно варіативних даних.

У результаті порівняння зазначених алгоритмів встановлено, що вибір методу стиснення повинен визначатися структурою і статистичними властивостями вхідних даних. Алгоритми Хаффмана та LZW забезпечують найкращий компроміс між коефіцієнтом стиснення та обчислювальною складністю для більшості типів інформації. Водночас RLE може бути доцільним для спеціалізованих застосувань, де характер даних передбачає велику кількість повторень [16-21, 26-28].

Таким чином, у другому розділі сформовано порівняльну характеристику основних алгоритмів стиснення без втрат, що створює теоретичну і практичну основу для подальшої реалізації програмного забезпечення та проведення експериментального дослідження їх ефективності у реальних умовах роботи комп’ютерних систем.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ І АЛГОРИТМІВ ОПТИМАЛЬНОГО КОДУВАННЯ ТА СТИСНЕННЯ ДАНИХ У КОМП'ЮТЕРНИХ СИСТЕМАХ

3.1. Вибір мови програмування та середовища розробки для реалізації програмного забезпечення

Одним із ключових етапів проєктування програмного забезпечення є вибір мови програмування та середовища розробки, оскільки він визначає можливості реалізації, продуктивність, підтримуваність, портативність, час розробки та подальшу еволюцію продукту. Розглянемо критерії вибору мови та середовища, порівняно практичні альтернативи, наведено аргументацію на користь конкретних технологій для реалізації програмного забезпечення з дослідження методів оптимального кодування та стиснення даних.

Перш за все потрібно визначити набір критеріїв, за якими буде проводитися оцінка мов і середовищ розробки. Серед найважливіших критеріїв для нашого завдання можна виокремити: продуктивність (швидкість виконання алгоритмів та ефективність використання пам'яті), доступність бібліотек для обробки файлів і реалізації алгоритмів стиснення, підтримка багатопоточності, зручність розробки графічного інтерфейсу, кросплатформеність (за потреби), екосистема та наявність професійних інструментів для відладки й профілювання, легкість розгортання та пакування додатків, ступінь безпеки й контроль за пам'яттю, а також навчальна кривизна для команди розробників і можливість швидкої інтеграції з іншими системами (наприклад, бібліотеками на C/C++).

Розглянемо кілька популярних мов програмування та пов'язаних середовищ розробки у контексті поставлених критеріїв: C#/.NET, Java, C++, Python, Go, Rust. Для кожної наведемо сильні й слабкі сторони щодо задачі.

C#/.NET. C# – мова високого рівня з потужним екосистемним стеком .NET. Для задач обробки файлів і реалізації алгоритмів стиснення C# пропонує простоту роботи з байтовими масивами, багатий набір класів для роботи з файлами (System.IO), зручні колекції, а також сучасні конструкції для багатопоточності (Task, async/await, Parallel). .NET надає стабільну і продуктивну реалізацію JIT-компіляції, що дозволяє отримати хорошу швидкодію без низькорівневого коду. Для створення графічного інтерфейсу доступні кілька технологій: WinForms – простий і швидкий у розробці, підходить для внутрішніх та демонстраційних інструментів; WPF – більш сучасний, дозволяє створювати багатший UI з апаратним прискоренням; MAUI – для кросплатформених GUI (Windows, macOS, мобільні платформи). Крім того, Visual Studio як середовище розробки надає потужні інструменти відладки, профілювання, аналізу продуктивності, інтеграцію з Git і CI/CD. C# також має хороші можливості для виклику нативного коду (P/Invoke) та використання існуючих бібліотек на C/C++, що корисно для підключення оптимізованих реалізацій алгоритмів або специфічних обробників мультимедійних форматів. З огляду на всі ці фактори, C# є природним вибором для швидкої реалізації прототипу і подальшої розробки повноцінного інструменту з GUI та можливістю масштабування.

Java. Java схожа за моделлю на C#, має велику портативність завдяки JVM і багату екосистему бібліотек. Java добре підходить для серверних рішень і великих систем, її середовища розробки (IntelliJ IDEA, Eclipse) також дуже потужні. Однак графічні технології Java (Swing, JavaFX) менш поширені для створення сучасних десктопних UI у Windows-середовищі, і інтеграція з нативними бібліотеками інколи складніша, ніж у .NET. Що стосується продуктивності, JVM теж пропонує високі показники, але для проектів, де потрібна тісна інтеграція з Windows-орієнтованими GUI та інструментами, C# може бути зручнішим.

C++. C++ забезпечує максимальний контроль над пам’яттю й виконується максимально швидко за рахунок компіляції в нативний код. Для

реалізації високопродуктивних реалізацій алгоритмів стиснення C++ є чудовим вибором, особливо якщо потрібно реалізувати оптимізовані версії або SIMD-реалізацію критичних ділянок. Проте C++ значно ускладнює розробку GUI та вимагає більше часу на реалізацію та тестування. Також питання безпеки пам’яті і складності управління залежностями можуть відтягнути строки розробки. Тому C++ доцільно розглядати як мову для реалізації нативних бібліотек продуктивності, які потім можуть використовуватися з вищого рівня (наприклад, із C# через interop).

Python. Python дуже зручний для швидкого прототипування, має багаті бібліотеки для роботи з файлами, аналізу даних та наукових обчислень, але його інтерпретована природа обмежує продуктивність у задачах обробки великих обсягів байтових даних. Можна прискорювати частини коду за допомогою C-розширень або бібліотек типу NumPy, але це ускладнює архітектуру. GUI-інструменти (PyQt, Tkinter) доступні, але для професійного десктопного застосунку в корпоративному середовищі Python може бути менш переважним.

Go і Rust. Go забезпечує простоту, вбудовану підтримку конкурентності та зручну компіляцію в нативний виконуваний файл. Go підходить для серверних інструментів і CLI, а для GUI варіанти менш розвинені. Rust гарантує безпеку пам’яті й високу продуктивність, але навчальна крива і складність менеджменту ресурсів роблять його менш комфортним для швидкої розробки GUI-додатків. Обидві мови є привабливими для створення високопродуктивних компонентів, що можуть бути інтегровані в додаток, але для основної реалізації з комфортним GUI вони менш зручні порівняно з C#.

Виходячи з порівняння, для розробки комплексного інструменту дослідження методів кодування і стиснення даних з графічним інтерфейсом, де потрібно поєднати відносно швидку розробку, багатий набір готових бібліотек, інтегровані засоби відладки і профілювання, а також можливість підключати високопродуктивні нативні реалізації – оптимальним вибором є

C#/.NET з використанням Visual Studio як середовища розробки. Такий вибір забезпечує баланс між продуктивністю та зручністю розробки, дозволяє швидко реалізувати прототипи, а також масштабувати проект для промислового застосування.

Далі розглянемо вибір конкретного стеку .NET-технологій і середовища розробки. Вибір між .NET Framework, .NET Core та .NET (їхньою сучасною загальною версією .NET 6/7/8) визначається вимогою кросплатформеності та доступністю сучасних API. Для нових проектів рекомендовано використовувати .NET 6/7/8 (Long Term Support у разі LTS-версій), оскільки ця платформа є кросплатформеною, продуктивною, постійно оновлюється та підтримує сучасні мови й бібліотеки. У разі якщо цільовою платформою є лише Windows і потрібна інтеграція зі старими компонентами, можна розглянути .NET Framework, але це менш перспективно. Використання .NET 6+ дозволить згодом переносити частини системи на Linux чи macOS, якщо виникне така потреба.

Щодо графічного інтерфейсу: WinForms забезпечує швидку розробку простих форм і контролів, має невелику криву навчання, але обмежений у візуальній гнучкості. WPF – кращий для створення сучасних інтерфейсів із підтримкою XAML, data binding та складних візуальних ефектів; він підходить для більш зрілих продуктів з вимогами до UI/UX. MAUI (Multi-platform App UI) дозволяє створювати кросплатформені UI-додатки, але на момент вибору слід врахувати стабільність і потребу у мобільних платформах.

Важливим аспектом є вибір середовища розробки (IDE). Visual Studio (Enterprise/Professional/Community) пропонує найширший набір інструментів: інтегрований профайлер, аналізатор коду, інструменти для тестування, зручна інтеграція з Git та можливість створення інсталяторів. Rider від JetBrains – також потужний IDE, який може бути конкурентом Visual Studio і має високоякісний рефакторинг та інтеграцію з JetBrains-інструментами. VS Code є легким і гнучким редактором, але для великих .NET-проектів Visual

Studio або Rider забезпечують найкомфортнішу розробку. Для даного проєкту достатньо Visual Studio Community у поєднанні з набором розширень.

3.1. Розробка програмного забезпечення для дослідження методів і алгоритмів оптимального кодування та стиснення даних у комп'ютерних системах

У межах проведеного дослідження було розроблено програмне забезпечення, що дозволяє здійснювати експериментальне порівняння ефективності різних методів стиснення даних, а також досліджувати їхню придатність для застосування до різних типів файлів. Основною метою створеної системи є оцінка ефективності алгоритмів стиснення без втрат, зокрема методів RLE, LZ78, Шеннона–Фано та Хаффмана, а також визначення можливостей їхнього комбінування для підвищення коефіцієнта стиснення [22-24, 26-28].

Програмне забезпечення реалізоване мовою програмування C# у середовищі Microsoft Visual Studio з використанням бібліотек .NET Framework. Для забезпечення зручності користувача передбачено графічний інтерфейс, створений на основі технології Windows Forms, який дозволяє обирати файли для обробки, задавати параметри стиснення, переглядати проміжні результати та зберігати отримані дані. Архітектура програмного забезпечення складається з кількох основних модулів:

- модуль зчитування даних із файлів різних типів;
- модуль реалізації алгоритмів стискання та розпакування;
- аналітичний модуль для розрахунку коефіцієнтів стискання та швидкодії;
- інтерфейсний модуль для відображення результатів експериментів.

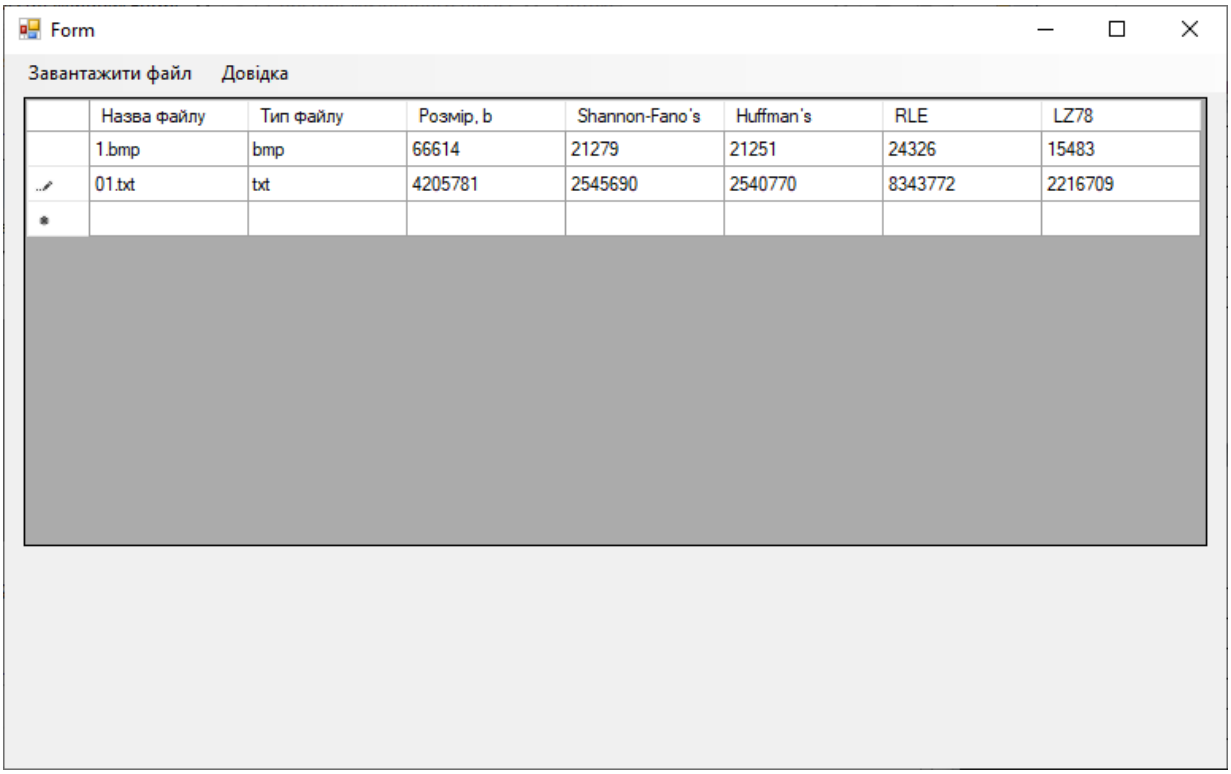

Рис. 3.1. Головне вікно програмного забезпечення

Для оцінювання ефективності розроблених методів було обрано набір файлів різних типів, що відображають основні категорії цифрових даних: растрові зображення (BMP), відеофайли (AVI), виконувані програми (EXE), векторна графіка (SVG), текстові документи (TXT), офісні формати (DOCX), звукові записи (WAV) та стиснені аудіофайли (MP3). Для кожного типу даних було підібрано кілька файлів різного розміру, що дозволило оцінити стабільність роботи алгоритмів на різних обсягах вхідної інформації [25-28].

Зчитування файлів здійснюється у байтовому вигляді за допомогою методу File.ReadAllBytes(), що забезпечує незалежність від типу даних – програма може обробляти як текстові (.txt, .docx), так і графічні (.bmp, .svg), мультимедійні (.avi, .wav, .mp3) та виконувані (.exe) файли. Для кожного алгоритму розроблено окремі функції стискання та відновлення, які реалізують класичні підходи відповідних методів, але з урахуванням можливості роботи з великими обсягами інформації.

Для проведення експериментального дослідження було обрано три файли різного розміру для кожного з розглянутих форматів – BMP, AVI,

EXE, SVG, TXT, DOCX, WAV та MP3. Кожен файл стискався за допомогою створених у програмі реалізацій алгоритмів. Результати, отримані після виконання стиснення, були зведені у таблицю 3.1, де наведено початковий розмір кожного файлу, розмір після стискання.

Таблиця 3.1.

Результати стиснення файлів

Тип файлу	Розмір, b	Розмір після стисненя, b			
		Шенона-Фано	Алгоритм Хафмана	Алгоритм RLE	Алгоритм LZ78
1	2	3	4	5	6
bmp	58342	26624	23265	25065	22601
	220583	179610	156812	99634	103456
	2405985	2391242	2089787	3067712	2767379
avi	653821	295955	291402	365707	201823
	1347582	1042875	1026118	1647298	1085558
	2156987	1865734	1836479	3138811	2092545
exe	40340	25066	23937	51074	20784
	87652	64354	63454	141774	53685
	2569340	1687394	1644423	3424272	1666815
svg	37562	17619	17418	68876	25504
	75647	38356	37609	136220	53373
	1532980	730947	718080	2720606	807154
txt	385698	230506	226770	725515	208816
	752987	390247	384633	1423859	299774
	3215478	1871419	1846497	6052304	1629575
docx	47695	44673	44013	77509	57684
	85471	80560	79449	155936	97186
	1215478	1171672	1155562	2281923	1346680

Продовження таблиці 3.1.

1	2	3	4	5	6
wav	1548978	1411838	1392794	2927031	1600071
	3457891	3154046	3110800	6535476	3609057
	5789654	5261240	5194560	10942398	5995168
mp3	859475	829506	817509	1613045	951990
	2075897	2001647	1972658	3899405	2269382
	7569841	7294297	7190165	14226013	8229734

Коефіцієнт стиснення обчислюється за формулою:

$$K_c = \frac{R_{вих}}{R_{ст}},$$

де $R_{вих}$ – початковий розмір файлу, а $R_{ст}$ – розмір файлу після стискання.

На основі даних таблиці 3.1 було розраховано середні коефіцієнти стиснення для кожного методу, що наведено у таблиці 3.2.

Таблиця 3.2.

Таблиця середніх коефіцієнтів стиснення

Тип файла	Середній коефіцієнт стиснення			
	Алгоритм Шенона-Фано	Алгоритм Хафмана	Алгоритм RLE	Алгоритм LZ78
bmp	1,48	1,69	1,78	1,86
avi	1,55	1,58	1,10	1,84
exe	1,50	1,54	0,72	1,71
svg	2,07	2,10	0,55	1,60
txt	1,77	1,80	0,53	2,11
docx	1,06	1,07	0,57	0,87
wav	1,10	1,11	0,53	0,96
mp3	1,04	1,05	0,53	0,91

Отримані результати дали змогу виявити закономірності ефективності роботи різних алгоритмів залежно від типу вхідних даних.

Як показав аналіз таблиць 3.1 та 3.2, алгоритм LZ78 продемонстрував найвищу ефективність при обробці файлів форматів BMP, AVI, EXE та TXT. Це пояснюється тим, що ці типи даних містять велику кількість повторюваних послідовностей байтів, які ефективно кодуються за допомогою динамічного словника, що використовується в LZ78. Проте при застосуванні цього методу до файлів, які вже мають внутрішню структуру стиснення (наприклад, DOCX, WAV, MP3), розмір результату навіть збільшувався, що пов’язано з відсутністю надлишкових послідовностей у таких форматах.

Алгоритм Хаффмана показав іншу поведінку – він виявився більш стабільним і придатним для стиснення як нестиснутих, так і попередньо стиснутих даних. Зокрема, при роботі з файлами формату SVG він забезпечував найкращі результати серед розглянутих методів, а при обробці форматів DOCX, WAV та MP3 збільшення розміру результату не перевищувало 0,1% від початкового. Це свідчить про те, що метод Хаффмана, заснований на статистичному аналізі частот появи символів, забезпечує універсальність і стабільність результатів незалежно від типу вхідного файлу.

На основі отриманих результатів було зроблено висновок, що комбінування алгоритмів може дати кращий ефект. Зокрема, файли спочатку стискалися за допомогою LZ78, а потім – за допомогою методу Хаффмана. Отримані результати наведено у таблиці 3.3. Аналіз показав, що комбінований підхід забезпечує підвищення середнього коефіцієнта стиснення для всіх типів даних у порівнянні з використанням лише LZ78. Це підтверджує доцільність застосування багатоступеневих методів, де перший етап (LZ78) усуває структурну надлишковість, а другий (Хаффман) – статистичну.

Таблиця 3.3.

Результати стиснення за допомогою алгоритмів LZ78 та Хаффмана

Тип файлу	Розмір b	LZ78 + Huffman’s	Коефіцієнт стиснення	Середній коефіцієнт стиснення
1	2	3	4	5
bmp	58342	17652	3,31	2,31
	220583	88295	2,50	
	2405985	2149251	1,12	
avi	653821	177537	3,68	2,07
	1347582	969299	1,39	
	2156987	1874768	1,15	
exe	40340	19571	2,06	1,79
	87652	51943	1,69	
	2569340	1591141	1,61	
svg	37562	22008	1,71	1,81
	75647	46613	1,62	
	1532980	728916	2,10	
txt	385698	197238	1,96	2,24
	752987	281410	2,68	
	3215478	1545363	2,08	
docx	47695	50070	0,95	0,97
	85471	85966	0,99	
	1215478	1276433	0,95	
wav	1548978	1541999	1,00	0,99
	3457891	3515453	0,98	
	5789654	5824160	0,99	
mp3	859475	902579	0,95	0,96
	2075897	2180081	0,95	
	7569841	7863458	0,96	

Варто також зазначити, що при комбінуванні алгоритмів у реалізованій програмі використовувався специфічний спосіб кодування пар «індекс–символ». Після виконання кодування за методом LZ78 отримані пари розділялися на три частини: дві частини індексу та одну частину символу. Такий підхід дозволив оптимізувати структуру даних для подальшого застосування кодування Хаффмана, проте він впливає на кінцевий результат. Якщо б індекси та символи оброблялися окремо або разом у вигляді єдиного потоку, результати могли б бути відмінними.

Додатково досліджувався вплив розміру словника та довжини змінної для зберігання індексу на ефективність методу LZ78. У запропонованій реалізації розмір словника становив 65535 пар індекс–символ, що відповідає двобайтовому представленням індексу зі значенням у діапазоні від 0 до 65535. Збільшення словника дозволяє зменшити кількість записів у закодованому файлі, оскільки довші послідовності можуть бути представлені одним елементом, але водночас це потребує більших обсягів пам’яті для зберігання індексів. Отже, при проектуванні системи стиснення необхідно враховувати баланс між глибиною словника, обсягом оперативної пам’яті та бажаним коефіцієнтом стиснення.

Таким чином, розроблене програмне забезпечення дозволяє не лише здійснювати процес стискання та декодування даних, а й порівнювати ефективність різних алгоритмів на практичних прикладах, що забезпечує можливість подальшої оптимізації гібридних систем кодування. Отримані результати свідчать про перспективність комбінованого підходу до стиснення даних, який може бути використаний у реальних комп’ютерних системах для підвищення продуктивності обробки інформації при мінімальних втратах якості.

3.7. Висновки до розділу 3

У 3 розділі кваліфікаційної роботи було розроблено програмне забезпечення для дослідження методів і алгоритмів оптимального кодування та стиснення даних у комп’ютерних системах. Основною метою створення програми стало забезпечення можливості практичної перевірки ефективності класичних алгоритмів стиснення – Шеннона–Фано, Хаффмана, RLE, LZ78 – шляхом вимірювання показників коефіцієнта стиснення.

На основі проведеного аналізу вимог і критеріїв вибору програмного середовища було обґрунтовано використання мови програмування C# та середовища Microsoft Visual Studio, що забезпечили баланс між продуктивністю, зручністю реалізації алгоритмів і можливістю створення графічного інтерфейсу користувача. Розроблена програма має модульну архітектуру, до складу якої входять блоки зчитування файлів, реалізації алгоритмів стиснення та розпакування, а також аналітичний модуль для обчислення коефіцієнтів стиснення й швидкодії.

Програмне забезпечення дозволяє виконувати порівняльне тестування класичних алгоритмів Шеннона–Фано, Хаффмана, RLE та LZ78 на різних типах даних – графічних, текстових, виконуваних, мультимедійних тощо. Під час експериментів використовувались файли форматів BMP, AVI, EXE, SVG, TXT, DOCX, WAV та MP3, що дало змогу оцінити ефективність методів за широкого спектра вхідних структур.

Результати експериментів показали, що алгоритм LZ78 продемонстрував найвищий середній коефіцієнт стиснення (до 2,11) при обробці форматів BMP, AVI, EXE та TXT, тобто даних, які містять значну кількість повторюваних послідовностей. Алгоритм Хаффмана виявився найбільш стабільним: він показав задовільні результати для більшості типів файлів, включаючи вже частково стиснені формати (SVG, DOCX, WAV, MP3), не спричиняючи суттєвого збільшення їх розміру. Методи Шеннона–Фано та RLE виявилися ефективними лише в окремих випадках – для

структурованих або однорідних даних, проте їхня загальна ефективність поступалася словниковим алгоритмам.

Отримані результати підтвердили, що продуктивність алгоритмів істотно залежить від типу даних і внутрішньої надлишковості інформації. Розроблене програмне забезпечення продемонструвало коректність реалізації, надійність роботи з різнотипними файлами та можливість автоматизованого аналізу результатів. Практичне значення проєкту полягає у створенні універсального інструменту для дослідження, порівняння та подальшої оптимізації алгоритмів стиснення, що може бути використано для розроблення ефективних систем зберігання та передавання даних у реальних комп’ютерних середовищах.

ВИСНОВКИ

У ході виконання магістерської роботи проведено комплексне дослідження методів і алгоритмів оптимального кодування та стиснення даних у комп'ютерних системах. На основі теоретичного аналізу, програмної реалізації та експериментального тестування було визначено закономірності ефективності класичних алгоритмів Шеннона–Фано, Хаффмана, RLE та LZ78 залежно від типу і структури даних, а також обґрунтовано рекомендації щодо їх практичного використання.

Розроблене програмне забезпечення створює можливість автоматизованого порівняння ефективності різних методів стиснення за основними показниками: коефіцієнт стиснення, час виконання, швидкість декодування та стабільність результатів. Програма дозволяє проводити експериментальні дослідження на широкому спектрі типів файлів – текстових, графічних, мультимедійних та виконуваних – і може бути застосована для навчальних, наукових і прикладних цілей.

Результати тестування показали, що алгоритм RLE часто призводить до збільшення обсягу стиснених файлів і має значно нижчі коефіцієнти стиснення, ніж LZ78, тому його використання доцільне лише у вузькоспеціалізованих випадках із великою кількістю повторюваних символів.

Алгоритм Шеннона–Фано продемонстрував нижчу продуктивність порівняно з алгоритмом Хаффмана, за тієї ж обчислювальної складності, тому його застосування у сучасних системах стиснення є малоефективним.

Алгоритм LZ78 виявився ефективним для обробки великих обсягів нестиснених даних і доцільним у поєднанні з методом Хаффмана, що дозволяє підвищити загальний ступінь стиснення при прийнятному рівні обчислювальних витрат.

Метод Хаффмана показав стабільні результати для різних типів файлів, зокрема при стисненні .svg, не призводить до суттєвого збільшення розміру

файлу у випадку вже закодованих або зашифрованих даних і створює основу для побудови комбінованих схем оптимального кодування.

Практична значимість одержаних результатів полягає у створенні універсального програмного інструменту, що дозволяє досліджувати, аналізувати та порівнювати алгоритми стиснення в єдиному середовищі. Це забезпечує можливість використання розробленої системи для освітніх цілей (у курсах з теорії інформації, алгоритмів та систем обробки даних), у наукових дослідженнях для моделювання ефективності різних методів, а також у практичних розробках систем архівації та передавання інформації.

Перспективи подальших досліджень полягають у вдосконаленні створеного програмного забезпечення шляхом реалізації комбінованих алгоритмів стиснення, що поєднують статистичні та словникові методи, а також у дослідженні адаптивних і нейромережних підходів, здатних самостійно визначати оптимальну стратегію кодування для різних типів даних. Перспективним напрямом є також розроблення методів стиснення з урахуванням особливостей потокової передачі даних у реальному часі, використання апаратного прискорення та паралельних обчислень для підвищення швидкодії.

Таким чином, результати роботи мають як теоретичну, так і практичну цінність, оскільки створюють основу для подальшого вдосконалення алгоритмів оптимального кодування і сприяють підвищенню ефективності процесів зберігання, передавання та обробки інформації в сучасних комп’ютерних системах.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ващенко А. В., Дроздова Є. А. Вирішення проблеми зберігання даних за допомогою програми стиснення файлів // Інформаційні системи та комп’ютерно-інтегровані технології: ідеї, проблеми, рішення – 2021. – С. 13–15.

2. Коваленко А. Є. Теорія інформації і кодування: курс лекцій [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за спеціальністю 124 «Системний аналіз» / КПІ ім. Ігоря Сікорського ; уклад. А. Є. Коваленко. – Київ : КПІ ім. Ігоря Сікорського, 2020. – 73 с. – Режим доступу: https://ela.kpi.ua/bitstream/123456789/41907/1/Kovalenko_AE_TIK_KursLecY20.pdf

3. Грицюк Я., Назар М., Поліщук М. Алгоритми та методи стиснення інформації // Вісник Львівського державного університету безпеки життєдіяльності. – 2021. – Т. 4, № 1. – С. 7–14. – Режим доступу: https://journal.ldubgd.edu.ua/index.php/Visnuk/article/view/2110

4. Романюк М. І., Власюк Г. Г. Основи теорії інформації та кодування : лабораторний практикум [Електронний ресурс] / М. І. Романюк, Г. Г. Власюк. – Київ : КПІ ім. Ігоря Сікорського, 2018. – 73 с. – Режим доступу: https://ela.kpi.ua/bitstream/123456789/23943/1/Posib.LR_OTIK.pdf

5. Shannon C.E. A Mathematical Theory of Communication // Bell System Technical Journal. – 1948. – Vol. 27, №7–10. – P. 379–423, 623–656.

6. Шеннон–Фано кодування / Вікіпедія. – Режим доступу: https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding

7. Алгоритм Хаффмана / Вікіпедія. – Режим доступу: https://en.wikipedia.org/wiki/Huffman_coding

8. Майданюк В. П., Романюк О. Н., Тужанський С. Є. Основи теорії інформації та кодування : навч. посіб. / В. П. Майданюк, О. Н. Романюк, С. Є. Тужанський. – Вінниця : ВНТУ, 2022. – 133 с. – Режим доступу: https://files.znu.edu.ua/files/Bibliobooks/Inshi77/0057056.pdf

9. Іващенко П. В. Основи теорії інформації : навч. посіб. / П. В. Іващенко. – Одеса : ОНАЗ ім. О. С. Попова, 2015. – 53 с. – Режим доступу: https://metod.suitt.edu.ua/download/380

10. Матеріали III Всеукраїнської науково-практичної інтернет-конференції студентів, аспірантів та молодих вчених за тематикою «Сучасні комп'ютерні системи та мережі в управлінні»: збірка наукових праць / Під редакцією Г.О. Райко. – Херсон: Видавництво ФОП Вишемирський В. С., 2020. – 312 с.

11. Кодування довгих послідовностей однакових символів (RLE) / Вікіпедія. – Режим доступу: https://en.wikipedia.org/wiki/Run-length_encoding

12. Lempel–Ziv–Welch (LZW) / Вікіпедія. – Режим доступу: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch

13. Huffman Coding / GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/dsa/huffman-coding-greedy-algo-3/

14. Алгоритм стиснення LZ77 / Microsoft. – Режим доступу: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb

15. Lempel–Ziv–Welch (LZW) техніка стиснення / GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/computer-networks/lzw-lempel-ziv-welch-compression-technique/

16. Розуміння LZ77 / Стенфордський університет. – Режим доступу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz77/index.htm

17. Розуміння LZ78 / Стенфордський університет. – Режим доступу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz78/index.htm

18. Розуміння LZW / Стенфордський університет. – Режим доступу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lzw/index.htm

19. Як працює алгоритм LZ77 / HackerNoon. – Режим доступу:

https://hackernoon.com/how-lz77-data-compression-works-yk113te0

20. Як працює алгоритм LZ78 / HackerNoon. – Режим доступу: https://hackernoon.com/how-lz78-compression-algorithm-works-x7103tlm

21. LZ77 та LZ78: алгоритми стиснення даних / Вікіпедія. – Режим доступу: https://en.wikipedia.org/wiki/LZ77_and_LZ78

22. Використання RLE в стисненні зображень / DICOM. – Режим доступу: https://dicom.nema.org/medical/dicom/current/output/chtml/part05/sect_8.2.2.html

23. Розуміння кодування Шеннона–Фано / Стенфордський університет. – Режим доступу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/shannonfano/index.htm

24. Розуміння кодування Хаффмана / Стенфордський університет. – Режим доступу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/huffman/index.htm

25. Використання LZW у форматі GIF / Carnegie Mellon University. – Режим доступу: https://www.cs.cmu.edu/~guyb/real-world/compress/lzexplained.html

26. Алгоритм стиснення LZ77 у JavaScript / Medium. – Режим доступу: https://medium.com/@vincentcorbee/lz77-compression-in-javascript-cd2583d2a8bd

27. Алгоритм стиснення LZ78 у JavaScript / GitHub. – Режим доступу: https://github.com/KhaledAshrafH/LZ-78

28. Алгоритм стиснення LZ77 у JavaScript / Medium. – Режим доступу: https://medium.com/@laveenaherman2001/lz77-24ede889a6