Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

РОЗРОБКА ТА ДОСЛІДЖЕННЯ МЕТОДУ СТИСНЕННЯ ДАНИХ
(тема)

Виконав:
студент 2 курсу, групи ІНФм-21-1
Яременко Р.І.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник Машталір С.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2022 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Яременку Роману Івановичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка та дослідження методу стиснення даних

затверджена наказом по університету від 9 листопада 2022 року № 1469Ст

2. Термін подання студентом роботи до екзаменаційної комісії 26 листопада 2022 р.

3. Вихідні дані до роботи метод стиснення даних LZW, «Page Replacement» алгоритми, код змінної довжини, динамічний словник, мова програмування Java, середа розробки Intellij IDEA.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд методів стиснення даних.
2. Аналіз Page Replacement алгоритмів.
3. Програмна реалізація покращеного методу.
4. Тестування покращеного методу.
5. Огляд отриманих результатів та формування висновків.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) постановка задачі, алгоритми сімейства LZ, Page Replacement алгоритми, технології розробки.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Консультант з дотримання діючих стандартів та норм	Доцент Творошенко І.С.		

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	09.11.2022	
2	Аналіз завдання, підбір літератури	09.11.22-10.11.22	
3	Аналіз літератури з досліджуваної проблеми	11.11.22-12.11.22	
4	Аналіз технічних засобів	13.11.22-15.11.22	
5	Розробка методу побудови словника	16.11.22-17.11.22	
6	Програмна реалізація	18.11.22-19.11.22	
7	Оформлення пояснювальної записки	19.11.22-20.11.22	
8	Перевірка на плагіат	21.11.22-22.11.22	
9	Рецензування	22.11.22-23.11.22	
10	Підготовка презентації та доповіді	23.11.22-24.11.22	
11	Занесення роботи в електронний архів	24.11.22-25.11.22	
12	Попередній захист кваліфікаційної роботи	26.11.22	

Дата видачі завдання 9 листопада 2022 р.

Студент _____
(підпис)

Керівник роботи _____ проф._Машталір С.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 60 с., 7 табл., 18 рис., 51 джерело.

МЕТОД СТИСНЕННЯ ДАНИХ LZW, КОД ЗМІННОЇ ДОВЖИНИ, LOSSLESS, LOSSY, ДИНАМІНИЙ СЛОВНИК, LEAST RECENTLY USED, AGING REPLACEMENT.

Об’єктом дослідження є дані які потребують стиснення без втрат.

Метою дослідження є розробка та аналіз методу побудови словником для оптимізації процесу стиснення даних.

У результаті роботи був реалізований алгоритм LZW, та його вдосконалена версія. Модифікували процес формування словника алгоритмом LZW, використали код змінної довжини для оптимізації коефіцієнта стиснення алгоритму, також використали алгоритм з принципами так званих «Page Replacement Algorithms»: Least Recently Used та Aging Replacement.

DATA COMPRESSING METHOD LZW, VARIABLE LENGHT CODE, LOSSLESS, LOSSY, DYNAMIC DICTIONARY, LEAST RECENTLY USED, AGING REPLACEMENT.

The object of the research is data that requires lossless compression.

The purpose of the research is to develop and analyze a dictionary construction method to optimize the data compression process.

As a result of the work, the LZW algorithm and its improved version were implemented. We modified the dictionary formation process using the LZW algorithm, used a variable-length code to optimize the compression ratio of the algorithm, and also used an algorithm with the principles of the so-called «Page Replacement Algorithms»: Least Recently Used and Aging Replacement.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

LZ – Lempel-Ziv (алгоритм стиснення Лемпеля-Зіва)

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

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

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

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

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

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

BPC – Bits Per Character (біт на символ)

LRU – Least Recently Used (найменше використовуваний)

TTL – Time To Live (граничний період часу)

FIFO – First-In, First-Out (перший увійшов, перший вийшов)

NRU – Not Recently Used (нещодавно використовувався)

NFU – Not Frequently Used (використовується нечасто)

ВСТУП

Кількість даних, які зберігаються, обробляються та передаються, постійно зростає, кожного дня, хвилини, секунди. Цей процес збільшення даних змушує світ змінюватись також. Ці зміни включають:

- постійно зростаюча швидкість Інтернету та його трафіку;
- вибуховий розвиток мобільного зв’язку;
- постійне зростання якості відеозв’язку.

Стиснення даних є однією з основних технологій для кожного з цих аспектів мультимедійної революції. Розміщувати зображення, не кажучи вже про аудіо та відео, на вебсайтах було б непрактично, якби не алгоритми стиснення даних. Стільникові телефони не змогли б забезпечити зв’язок із зростаючою чіткістю, якби не стиснення. Поява цифрового телебачення була б неможлива без стиснення.

Саме тому сьогодні одна з найважливіших тем в обчислювальній техніці є стиснення даних. Шляхом стиснення даних зменшуємо обсяг пам’яті та навантаження на мережі [1].

Алгоритми стиснення зменшують розмір вхідних даних за допомогою різних підходів. Найпоширенішими є статистичний і словниковий підхід [2]. Незалежно від підходу, алгоритмів стиснення ділиться на два типи:

- з втратами;
- без втрат.

Актуальність дослідження полягає у тому що однією з труднощів у використанні алгоритму стиснення LZW для великих файлів даних є керування словником, оскільки розмір таблиці рядків часто перевищує розмір доступної пам’яті. Тут пропонується метод для побудови більш оптимального словника, покращили коефіцієнт стиснення, отриманий лише за допомогою LZW.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основні поняття

Розглянемо основні підходи, що застосовуються останні десятиліття для стиснення даних (у тому числі зображень [3] і відео).

Для подальшої розповіді нам знадобляться такі терміни:

– стиснення без втрат (lossless) − стиснення, при якому вихідні дані можуть бути точно відновлені після стиснення;

– стиснення з втратами (lossy) − стиснення, при якому відновлені після стиснення дані можуть незначно відрізнятися від вихідних даних;

– формула кількості інформації Шеннона;

– інформаційна ентропія − непередбачуваність появи будь-якого символу алфавіту.

Також необхідним буде дати далі декілька основних визначень в стисненні відео та зображень.

Стиснення зображення – це процес, застосований до графічного файлу для мінімізації його розміру в байтах без погіршення якості зображення нижче прийнятного порогу. Зменшивши розмір файлу, можна зберігати більше зображень на заданому об’ємі диска чи пам’яті. Зображення також вимагає меншої пропускної здатності під час передачі через Інтернет або завантаження з вебсторінки, що зменшує перевантаження мережі та пришвидшує доставку вмісту [4, 5].

Стиснення відео – це процес зменшення загальної кількості бітів, необхідних для представлення заданого зображення або відеопослідовності [6, 7]. Стиснення відео найчастіше виконується програмою з певним алгоритмом або формулою для визначення найкращого способу зменшення розміру даних. Алгоритми стиснення відео, такі як H.264/AVC або H.265/HEVC, зменшують необроблені дані вмісту в 1000 разів.

Відео сегментація − це процес поділу відеопослідовності на непересічні набори послідовних кадрів, однорідних за певними критеріями. У найпоширеніших типах сегментації відео поділяється на кадри, кадри або сцени [8, 9].

Кластеризації групує відеокадри та сегменти за схожістю кольорів, починаючи з кожного кадру у власному кластері та ітеративно поєднуючи два кластери, які створюють найменший об’єднаний кластер, поки не залишиться лише один кластер [10]. Рамки-члени, найближчі до центроїда, представляють кластери [11, 12].

1.2 Стиснення без втрат

Методи стиснення без втрат, як випливає з їх назви, не передбачають втрати інформації. Якщо дані були стиснуті без втрат, вихідні дані можна відновити саме зі стиснутих даних. Стиснення без втрат зазвичай використовується для програм, які не допускають будь-якої різниці між оригінальними та реконструйованими даними. Стиснення тексту є важливою сферою стиснення без втрат. Дуже важливо, щоб реконструкція була ідентичною оригінальному тексту, оскільки дуже невеликі відмінності можуть призвести до висловлювань з дуже різним значенням. Розгляньте речення «не надсилайте гроші» та «зараз надсилайте гроші». Подібний аргумент справедливий для комп’ютерних файлів і певних типів даних, таких як банківські записи. Якщо дані будь-якого типу мають бути оброблені або «покращені» пізніше, щоб отримати більше інформації, важливо зберегти цілісність. Наприклад, припустимо, що стиснули радіологічне зображення з втратами і різниця між реконструкцією Y та оригінальною V була візуально непомітною. Якщо це зображення пізніше було покращено, відмінності, які раніше не виявлялися, можуть спричинити появу артефактів, які можуть серйозно ввести рентгенолога в оману. Оскільки ціною такого роду нещасних

випадків може бути людське життя, має сенс бути дуже обережним із використанням схеми стиснення, яка генерує реконструкцію, яка відрізняється від оригіналу.

Дані, отримані із супутників, часто обробляються пізніше для отримання різних числових показників рослинності, вирубки лісів тощо. Якщо реконструйовані дані не ідентичні вихідним даним, обробка може призвести до «посилення» відмінностей. Можливо, неможливо буде повернутися назад і отримати ті самі дані знову. Тому недоцільно допускати будь-які відмінності в процесі стиснення.

Існує багато ситуацій, які вимагають стиснення, коли хочемо, щоб реконструкція була ідентичною оригіналу. Існує також ряд ситуацій, у яких можна послабити цю вимогу, щоб отримати більше стиснення. У таких ситуаціях шукаємо методи стиснення з втратами [1].

ZIP-файли – популярний приклад стиснення без втрат. Зберігати інформацію у вигляді ZIP-файлів більш ефективно, при цьому коли ви розпаковуєте архів, там є вся оригінальна інформація. Це актуально для виконуваних файлів, оскільки після стиснення із втратами розпакована версія буде пошкоджена та непридатна для використання.

Інші розповсюджені формати без втрат − PNG для зображень та FLAC для аудіо. Формати відео без втрат трапляються рідко, тому що вони займають багато місця.

1.3 Стиснення з втратами

Методи стиснення з втратами передбачають певну втрату інформації, і дані, які були стиснуті з використанням методів із втратами, зазвичай неможливо відновити або реконструювати точно. В обмін на прийняття цього викривлення під час реконструкції можемо отримати набагато вищі коефіцієнти стиснення, ніж це можливо при стисненні без втрат [13, 14].

У багатьох програмах відсутність точної реконструкції не є проблемою. Наприклад, при зберіганні або передачі мови точне значення кожного зразка мови не є необхідним. Залежно від якості, необхідної для реконструйованого мовлення, можна допускати різну кількість втрати інформації про значення кожного зразка. Якщо якість реконструйованого мовлення має бути подібною до почутої по телефону, можна допустити значну втрату інформації. Однак, якщо реконструйоване мовлення має бути такої якості, яку можна почути на компакт-диску, кількість втрати інформації, яку можна допустити, буде набагато нижчою [15, 16].

Подібним чином, під час перегляду реконструкції відеоряду, той факт, що реконструкція відрізняється від оригіналу, як правило, не є важливим, якщо відмінності не призводять до неприємних артефактів. Таким чином, відео зазвичай стискається за допомогою стиснення з втратами [1].

Стискання із втратами відмінно підходить для більшості медіафайлів. Це дуже важливо для таких компаній як Spotify та Netflix, які постійно транслюють великі обсяги інформації. Максимальне зменшення розміру файлу при збереженні якості робить їхню роботу більш ефективною [17].

1.4 Аналіз алгоритмів сімейства LZ

Алгоритм Lempel Ziv (LZ) − це алгоритм стиснення даних без втрат. Це не один алгоритм, а ціле сімейство алгоритмів (рис. 1.1), що походить від двох алгоритмів, запропонованих Джейкобом Зівом і Абрахамом Лемпелем у їхніх знакових статтях у 1977 та 1978 роках. Стосовно інших алгоритмів час стиснення та розпакування однаково. У процесі кодування LZ77 одне посилання (трійка) передається для кількох вхідних символів і, отже, це дуже швидко. У цьому процесі декодування відбувається набагато швидше, ніж кодування, і це одна з важливих особливостей цього процесу.

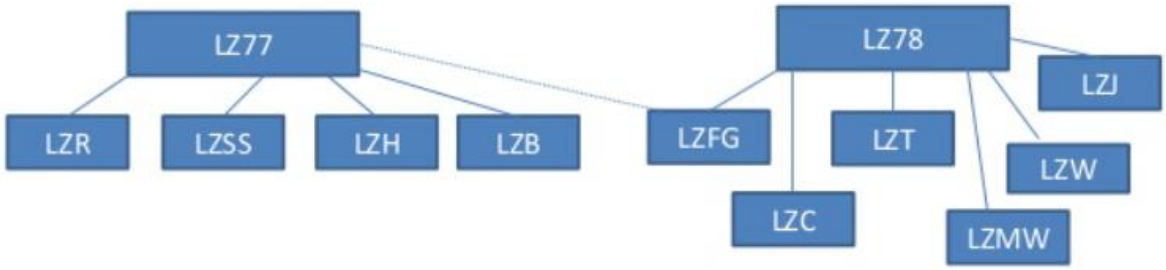

Рисунок 1.1 – Сімейство алгоритмів LZ

Однак у LZ77 більша частина часу стиснення LZ77 використовується для пошуку найдовшого збігу, тоді як декомпресія алгоритму LZ77 є швидкою, оскільки кожне посилання просто замінюється рядком, на який воно вказує.

Існує багато способів зробити схему LZ77 більш ефективною, і багато з удосконалень стосуються ефективного кодування за допомогою трійок. Існує кілька варіацій схеми LZ77, найвідомішими є LZSS, LZH і LZB. LZSS, який був опублікований Storer і Szymanksi, усуває вимогу обов'язкового включення наступного невідповідного символу в кожне кодове слово. Їх алгоритм використовує кодові слова фіксованої довжини, що складаються зі зміщення та довжини для позначення посилань. Вони пропонують включити додатковий біт (прапорець біта) на кожному кроці кодування, щоб вказати, чи представляє вихідний код пару (вказівник і довжина збігу) чи один символ. LZH – це схема, яка поєднує методики Зіва – Лемпеля та Хаффмана. Тут кодування виконується в два проходи. Перший, по суті, такий самий, як LZSS, тоді як другий використовує статистику, виміряну в першому, для кодування покажчиків і явних символів за допомогою кодування Хаффмана. LZB був опублікований Мохаммадом Баніказемі, використовує детальну схему для кодування посилань і довжини з різними розмірами. Незалежно від довжини фрази, яку він представляє, кожен покажчик LZSS має однаковий розмір. На практиці краще стиснення досягається за допомогою покажчиків різного розміру, оскільки певна довжина фрази зустрічається набагато частіше, ніж інші. LZB − це техніка, яка використовує різне кодування для

обох компонентів вказівника. LZB досягає кращого стиснення, ніж LZSS, і має додаткову перевагу, оскільки він менш чутливий до вибору параметрів.

1.4.1 Алгоритм LZ77

Джейкоб Зів і Абрахам Лемпель у 1977 році представили свою схему на основі словника для стиснення даних без втрат [2]. Сьогодні ця методика багатьом запам’ятовується іменем авторів і роком її впровадження. LZ77 використовує той факт, що слова та фрази в текстовому файлі можуть повторюватися [18]. Коли є повторення, вони можуть бути закодовані як вказівник на попереднє входження, при цьому вказівник супроводжується кількістю символів, які потрібно зіставити. Це дуже проста адаптивна схема, яка не потребує попереднього знання джерела та, здається, не вимагає припущень щодо характеристик джерела. У підході LZ77 словник є просто частиною попередньо закодованої послідовності. Кодер перевіряє вхідну послідовність через вікно, яке складається з двох частин: буфера пошуку, який містить частину нещодавно закодованої послідовності, і буфера перегляду, який містить наступну частину послідовності, яку потрібно закодувати. Алгоритм шукає вікно для найдовшого збігу з початком буфера прогнозу та виводить посилання (вказівник) на цей збіг. Можливо, що відповідності немає взагалі, тому вихідні дані не можуть містити лише покажчики. У LZ77 посилання завжди виводиться як трійка *o, l, c*, де *o* − зсув до збігу, *l* −довжина збігу, а *c* − наступний символ після збігу. Якщо збігу немає, алгоритм виводить нульовий покажчик (і зсув, і довжина збігу дорівнюють 0) і перший символ у буфері прогнозу. Значення зміщення для відповідності та довжини мають бути обмежені деякими максимальними константами. Крім того, ефективність стиснення LZ77 в основному залежить від цих значень. Зазвичай зміщення кодується на 12–16 бітах, тому воно обмежене від 0 до 65535 символів. Таким чином, немає необхідності

запам’ятовувати більше 65535 останніх використаних символів у ковзному вікні. Довжина збігу зазвичай кодується 8 бітами, що дає максимальну довжину збігу, що дорівнює 255 [19, 20].

1.4.2 Алгоритм LZ78

У 1978 році Джейкоб Зів і Абрахам Лемпель представили свою схему на основі словника [21], яка відома як LZ78. Це алгоритм стиснення на основі словника, який підтримує явний словник. Цей словник має бути створено як на стороні кодування, так і на стороні декодування, і вони повинні дотримуватися однакових правил, щоб гарантувати використання ідентичного словника. Кодові слова, виведені алгоритмом, складаються з двох елементів *i* та *c*, де *i* є індексом, що відноситься до найдовшої відповідної словникової статті та першого невідповідного символу. Окрім виведення кодового слова для зберігання/передачі, алгоритм також додає до словника пару індексу та символу. Коли символ, який ще не знайдено в словнику, кодове слово має значення індексу 0 і воно також додається до словника.

Алгоритм LZ78 має здатність захоплювати шаблони та зберігати їх нескінченно довго, але він також має серйозний недолік. Словник безмежно зростає. Існують різні способи обмеження розміру словника, найпростішим є припинити додавання записів і продовжити, як кодер статичного словника, або викинути словник і почати з нуля після досягнення певної кількості записів. Кодування, виконане LZ78, є швидким порівняно з LZ77, і це головна перевага стиснення на основі словника. Важлива властивість LZ77, яку зберігає алгоритм LZ78, полягає в тому, що декодування відбувається швидше, ніж кодування. Декомпресія в LZ78 відбувається швидше, ніж процес стиснення.

1.4.3 Модифіковані словникові алгоритми

LZW названо на честь його розробників А. Лемпеля та Дж. Зіва з пізнішими модифікаціями Террі А. Уелча. Це найкраща техніка для стиснення даних загального призначення завдяки своїй простоті та універсальності. Як правило, ви можете очікувати, що LZW стисне текст, виконуваний код і подібні файли даних приблизно до половини їх початкового розміру. LZW також добре працює, коли представлені надзвичайно надлишкові файли даних, такі як табличні числа, LZW є основою кількох утиліт персонального комп'ютера, які стверджують, що «подвоюють ємність вашого жорсткого диска». Якщо довжина кодового слова недостатньо велика, коди Лемпеля-Зіва також можуть повільно підвищуватися до прийнятної ефективності, короткочасно підтримувати хорошу продуктивність і не мати жодних переваг після вихідного коду та отриманих сигналів. Ступінь стиснення 5:1 є звичайним для цих випадків [22, 23].

LZSS − це техніка словникового кодування. На відміну від кодування Хаффмана, яке намагається зменшити середню кількість бітів, необхідних для представлення символу, LZSS намагається замінити рядок символів посиланням на розташування словника для того самого рядка. Передбачається, що словникове посилання має бути коротшим за рядок, який воно замінює. У випадку LZ77, попередника LZSS, це було не завжди так [24].

LZC є однофакторним нелінійним методом, який вимірює складність цього сигналу. Було встановлено, що він надійний з короткими наборами даних. Цей метод широко застосовувався в медичних дослідженнях, включаючи дослідження AD, хвороби Паркінсона і коми, а також багатьох інших. Щоб застосувати LZC, сигнал потрібно проаналізувати [25, 26].

LZAP (Lempel-Ziv All Prefixes) також є варіантом LZW, який додає більше записів шляхом додавання всіх префіксів до словника. Наприклад,

якщо спочатку прочитаємо «ABC» у нашому словнику, а потім «DEF»; «ABCD», «ABCDE», «ABCDEF» буде додано до нашого словника, а не лише «ABCD» у випадку LZW. Це означає, що в цілому отримуємо кращий коефіцієнт стиснення в більшості випадків [27].

LZMW (Lempel-Ziv-Miller-Wegman) − це варіант LZW, який покращує LZW двома способами [28]. Перший полягає в тому, що замість повторної ініціалізації словника, коли він заповнений, він видаляє найменш вживану фразу. Будь-який детермінований спосіб визначення найменш вживаної фрази буде працювати. Однак початкові 256 значень ASCII ніколи не слід видаляти. Друге покращення порівняно з LZW полягає в тому, що замість додавання найдовшого збігу на даний момент та першого невідомого символу додамо конкатенацію попереднього та поточного збігів до словника. Це означає, що замість того, щоб розширювати словникові статті лише буква за літерою, можем додавати цілі слова до словника. Це означає, що якщо стискаєте текст, словник часто складається зі слів або серії слів.

1.5 Аналіз Page Replacement алгоритмів

В операційній системі, яка використовує підкачку для керування пам’яттю, необхідний алгоритм заміни сторінки, щоб вирішити, яку сторінку потрібно замінити, коли надходить нова сторінка.

Помилка сторінки виникає, коли запущена програма звертається до сторінки пам’яті, яка відображена у віртуальному адресному просторі, але не завантажена у фізичну пам’ять. Оскільки фактична фізична пам’ять набагато менша за віртуальну, трапляються помилки сторінки. У разі помилки сторінки операційній системі, можливо, доведеться замінити одну з існуючих сторінок новою. Різні алгоритми заміни сторінок пропонують різні способи вирішувати, яку сторінку замінити. Метою всіх алгоритмів є зменшення кількості помилок сторінки.

Розглянемо існуючі алгоритми Page Replacement.

First In First Out (FIFO) – це найпростіший алгоритм заміни сторінок. У цьому алгоритмі операційна система відстежує всі сторінки пам’яті в черзі, найстаріша сторінка знаходиться на початку черги. Коли сторінку потрібно замінити, сторінка в першій частині черги вибирається для видалення.

Optimal Page replacement – це алгоритм у якому замінюються сторінки, які не використовуватимуться протягом найдовшого часу в майбутньому.

Least Recently Used (LRU) – це алгоритму у якому буде замінено сторінку, яка використовувалася найменше.

Most Recently Used (MRU) – це алгоритм у якому буде замінено сторінку, яка нещодавно використовувалася, також у цьому алгоритмі може виникнути аномалія Беладі.

Not Recently Used (NRU) – це алгоритм у який випадково видаляє сторінку з непорожнього класу з найменшим номером. У цьому алгоритмі мається на увазі, що краще видалити змінену сторінку, на яку не було посилання принаймні за один такт (зазвичай 20 мс), ніж чисту сторінку, яка активно використовується. Головна привабливість NRU полягає в тому, що він простий для розуміння, помірно ефективний у реалізації та забезпечує продуктивність, яка, звичайно, не є оптимальною, але може бути достатньою.

Second Chance Page Replacement – це проста модифікація FIFO, яка дозволяє уникнути проблеми викидання інтенсивно використовуваної сторінки, полягає в перевірці біта *R* найстарішої сторінки.

Якщо він дорівнює 0, сторінка є одночасно старою та невикористаною, тому її негайно замінюють. Якщо біт *R* дорівнює 1, біт очищається, сторінка поміщається в кінець списку сторінок, а час її завантаження оновлюється так, ніби вона щойно надійшла в пам’ять. Потім пошук продовжується.

Clock Page Replacement. Хоч «second chance» є розумним алгоритмом, він невиправдано неефективний, оскільки він постійно переміщує сторінки у своєму списку. Кращий підхід полягає в тому, щоб зберегти всі кадри

сторінок у кільцевому списку у формі годинника. Годинникова стрілка вказує на найстарішу сторінку.

Хоч обидва попередні алгоритми LRU в принципі реалізовані, небагато машин, якщо такі взагалі є, мають це обладнання, тому вони мало корисні для розробника операційної системи, який створює систему для машини, яка не має цього обладнання. Натомість потрібне рішення, яке можна реалізувати в програмному забезпеченні. Одна з можливостей називається алгоритмом NFU (Not Frequently Used).

Для цього потрібен програмний лічильник, пов’язаний із кожною сторінкою, спочатку нульовий. При кожному перериванні синхронізації операційна система сканує всі сторінки в пам’яті. Для кожної сторінки біт *R*, який дорівнює 0 або 1, додається до лічильника. По суті, лічильники є спробою відстежити, як часто посилаються на кожну сторінку. Коли виникає помилка сторінки, для заміни вибирається сторінка з найменшим лічильником.

У найчистішій формі підкачки сторінки процеси запускаються без жодної сторінки в пам’яті. Як тільки ЦП намагається отримати першу інструкцію, він отримує помилку сторінки, змушуючи операційну систему передавати сторінку, що містить першу інструкцію. Інші помилки сторінки для глобальних змінних і стека зазвичай слідують швидко. Через деякий час процес має більшість необхідних сторінок і починає працювати з відносно невеликою кількістю помилок. Ця стратегія називається «demand paging», оскільки сторінки завантажуються лише за запитом, а не заздалегідь.

Алгоритм основного робочого набору є громіздким, оскільки вся таблиця сторінок повинна скануватися при кожній помилці сторінки, доки не буде знайдено відповідний кандидат. Удосконалений алгоритм, який базується на алгоритмі годинника, але також використовує інформацію про робочий набір, називається WSClock (Carr and Hennessey). Завдяки простоті виконання і гарній продуктивності він широко використовується на практиці.

Загалом, два найкращі алгоритми – Aging та WSClock. Вони засновані на LRU і робочому наборі відповідно. Обидва забезпечують хорошу продуктивність підкачки і можуть бути ефективно реалізовані. Існує кілька інших алгоритмів, але ці два, мабуть, найважливіші на практиці.

1.6 Постановка задачі дослідження

Підсумовуючи вище сказане можемо сказати что дана проблема є актуальною так як зараз використовується дуже багато даних, та їх об’єм збільшується з кожним днем в геометричній прогресії. Однією з труднощів у використанні стиснення LZW для великих файлів даних є керування словником, оскільки розмір таблиці рядків часто перевищує розмір доступної пам’яті. Тут пропонується новий метод для побудови словника.

Об’єктом дослідження є дані які потребують стиснення без втрат.

Метою дослідження є розробка та аналіз методу побудови словником для оптимізації процесу стиснення даних.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз існуючих методів стиснення даних;

– провести аналіз сімейства алгоритмів LZ;

– провести аналіз існуючих методів побудови словником;

– розробити алгоритм побудови словником;

– реалізувати модифікований алгоритм LZW.

2 ПРИНЦИП РОБОТИ ТА АНАЛІЗ ХАРАКТЕРИСТИК СТИСНЕННЯ АЛГОРИТМІВ

2.1 Алгоритм LZW

Процес стиснення виглядає так. Послідовно зчитуються символи вхідного потоку та відбувається перевірка, чи існує у створеній таблиці рядків такий рядок. Якщо такий рядок існує, зчитується наступний символ, а якщо рядок не існує, в потік заноситься код попереднього знайденого рядка, рядок заноситься в таблицю, а пошук починається знову.

Наприклад, якщо стискають байтові дані (текст), рядків у таблиці виявиться 256 (від «0» до «255»). Якщо використовується 10-бітовий код, то під коди для рядків залишаються значення діапазоні від 256 до 1023. Нові рядки формують таблицю послідовно, тобто можна вважати індекс рядка її кодом.

Алгоритму декодування на вході потрібно лише закодований текст, оскільки він може відтворити відповідну таблицю перетворення безпосередньо закодованим текстом. Алгоритм генерує однозначно код, що декодується за рахунок того, що кожного разу, коли генерується новий код, новий рядок додається в таблицю рядків. LZW постійно перевіряє, чи є рядок вже відомим, і якщо так, виводить існуючий код без генерації нового. Таким чином, кожен рядок зберігатиметься в єдиному екземплярі і матиме свій унікальний номер. Отже, при дешифруванні при отриманні нового коду генерується новий рядок, а при отриманні вже відомого, рядок вилучається зі словника.

Просто за псевдокодом зрозуміти роботу алгоритму не дуже легко, тому розглянемо приклад стиснення та декодування зображення.

Спочатку створимо початковий словник окремих символів. У стандартному кодуванні ASCII є 256 різних символів, тому, щоб усі вони були коректно закодовані (якщо нам невідомо, які символи будуть присутні у вихідному

файлі, а які − ні), початковий розмір коду дорівнюватиме 8 біт. Якщо нам заздалегідь відомо, що у вихідному файлі буде менше різних символів, то цілком розумно зменшити кількість біт. Щоб ініціалізувати таблицю, встановимо відповідність коду 0 відповідного символу з кодом біт 00000000, тоді 1 відповідає символу з кодом 00000001, і т.д., до коду 255. Насправді вказали, що кожен код від 0 до 255 є. Більше в таблиці не буде інших кодів, які мають цю властивість. У міру зростання словника, розмір груп повинен зростати, щоб врахувати нові елементи. 8-бітові групи дають 256 можливих комбінації біт, тому, коли в словнику з'явиться 256 слово, алгоритм повинен перейти до 9-бітних груп. З появою 512-го слова відбудеться перехід до 10-бітових груп, що дає можливість запам'ятовувати вже 1024 слова і т.д. У нашому прикладі алгоритму заздалегідь відомо про те, що використовуватиметься всього 5 різних символів, отже, їх зберігання буде використовуватися мінімальна кількість біт, що дозволяє їх запам'ятати, тобто 3 (8 різних комбінацій).

2.1.1 Кодування

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

Розглянути логіку та як шаг за шагом вібдувається кодування можна на блок-схемі кодування алгоритму LZW що зображена на рисунку 2.1.

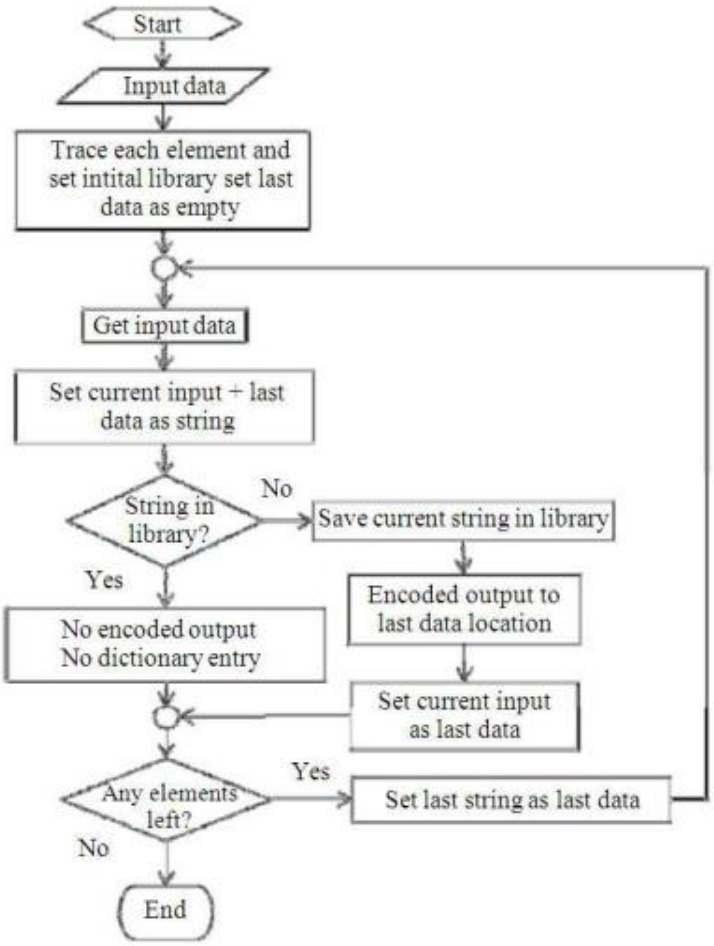

Рисунок 2.1 – Блок-схема кодування [29]

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

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

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

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

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

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

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

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

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

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

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

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

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

Отже, отримуємо закодоване повідомлення «0 1 0 2 5 0 3 9 8 6 4», що на 11 біт коротше. Для наглядності покрокові шаги представлені у таблиці 2.1.

Таблиця 2.1 – Кодування даних

Поточний рядок	**Поточний символ**	**Наступний символ**	**Вихід**		**Словник**
			Код	**Біти**	
1	**2**	**3**	**4**	**5**	**6**
ab	a	b	0	000	5:ab
ba	b	a	1	001	6:ba
ac	a	c	0	000	7:ac
ca	c	a	2	010	8:ca
ab	a	b	-	-	-:-
aba	b	a	5	101	9:aba
ad	a	d	0	000	10:ad
da	d	a	3	011	11:da
ab	a	b	-	-	-:-
aba	b	a	-	-	-:-

Продовження таблиці 2.1

1	2	3	4	5	6
abac	a	c	9	1001	12:abac
ca	c	a	-	-	-:-
cab	a	b	8	1000	13:cab
ba	b	a	-	-	-:-
bae	a	e	6	0110	14:bae
e	e	-	4	0100	-:-

2.1.2 Декодування

Блок-схему декодування алгоритму LZW представлено на рисунку 2.2.

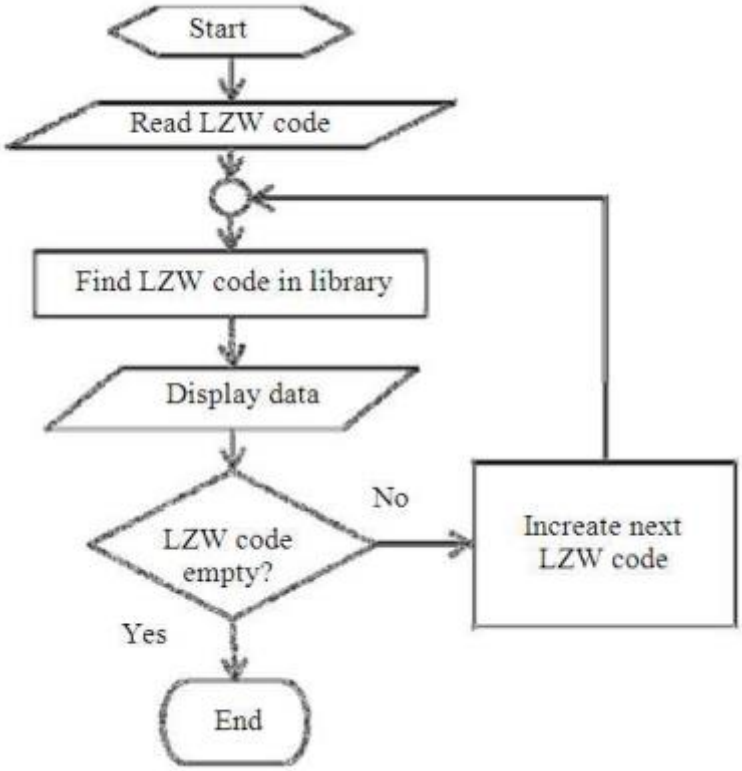

Рисунок 2.2 – Блок-схема декодування [29]

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

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

Таблиця 2.2 – Закодовані дані алгоритмом LZW

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

2.2 Практичне порівняння методів стиснення даних

У цьому розділі зосереджуємо нашу увагу на порівнянні продуктивності різних методів статистичного стиснення (кодування довжини серії, кодування Шеннона-Фано, кодування Хаффмана, адаптивне кодування Хаффмана та арифметичне кодування), алгоритмів сімейства LZ77 (LZSS, LZH і LZB) і алгоритми сімейства LZ78 (LZW і LZFG). Дослідження, проведені для оцінки ефективності будь-якого алгоритму стиснення, проводяться за двома важливими параметрами [30]. Один – це ступінь досягнутого стиснення, а інший – час, який витрачають алгоритми кодування та декодування. Кілька разів перевіряли практичну ефективність вищезазначених методів на файлах Кентерберійського корпусу та дізналися результати різних методів статистичного кодування та методів Лемпеля-Зіва, обраних для цього дослідження. Крім того, у наведених нижче таблицях представлено порівняльну роботу та ступінь стиснення.

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

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

алгоритмом Шеннона-Фано, а середнє значення бітів на символ становить 5,27.

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

Загальна продуктивність з точки зору середнього BPC згаданих вище методів статистичного кодування показана на рисунку 2.3.

Таблиця 2.3 – Порівняння BPC для різних методів статистичного стиснення

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

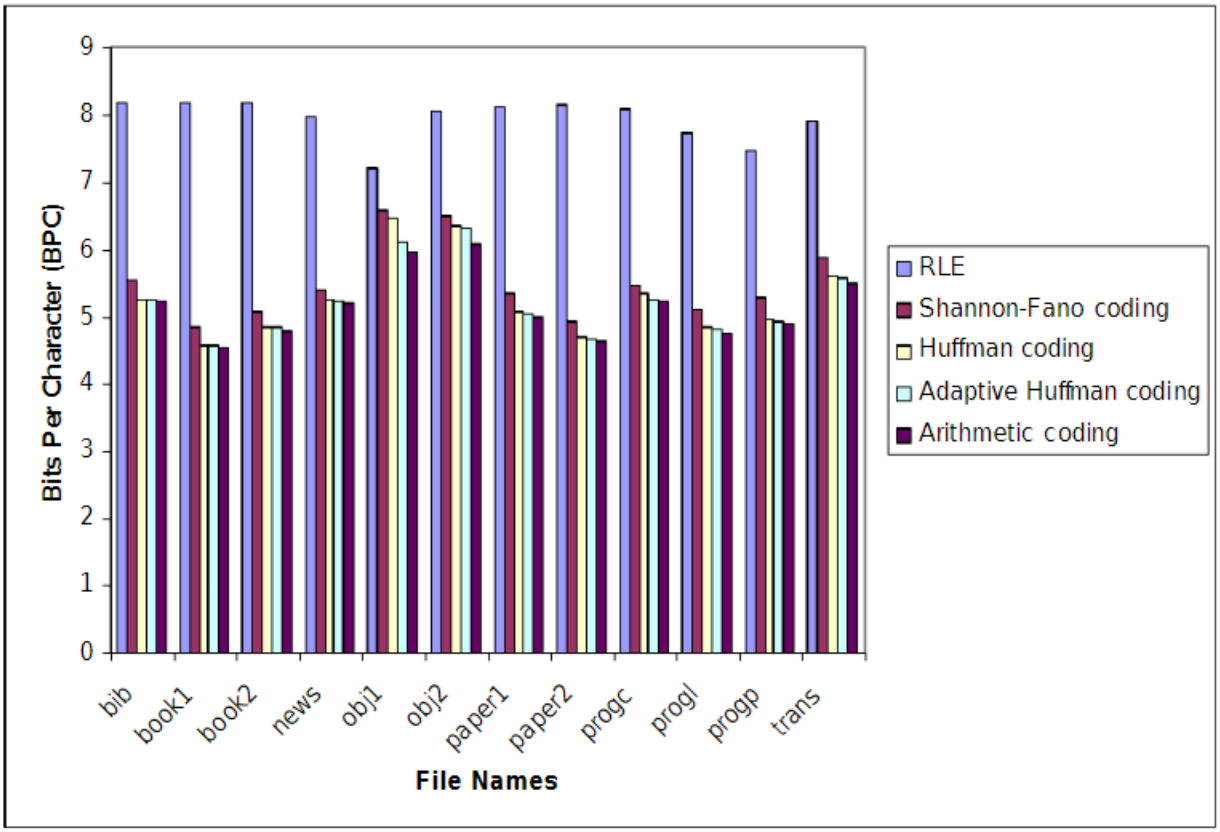

Рисунок 2.3 – Графік порівняння методів стиснення [10]

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

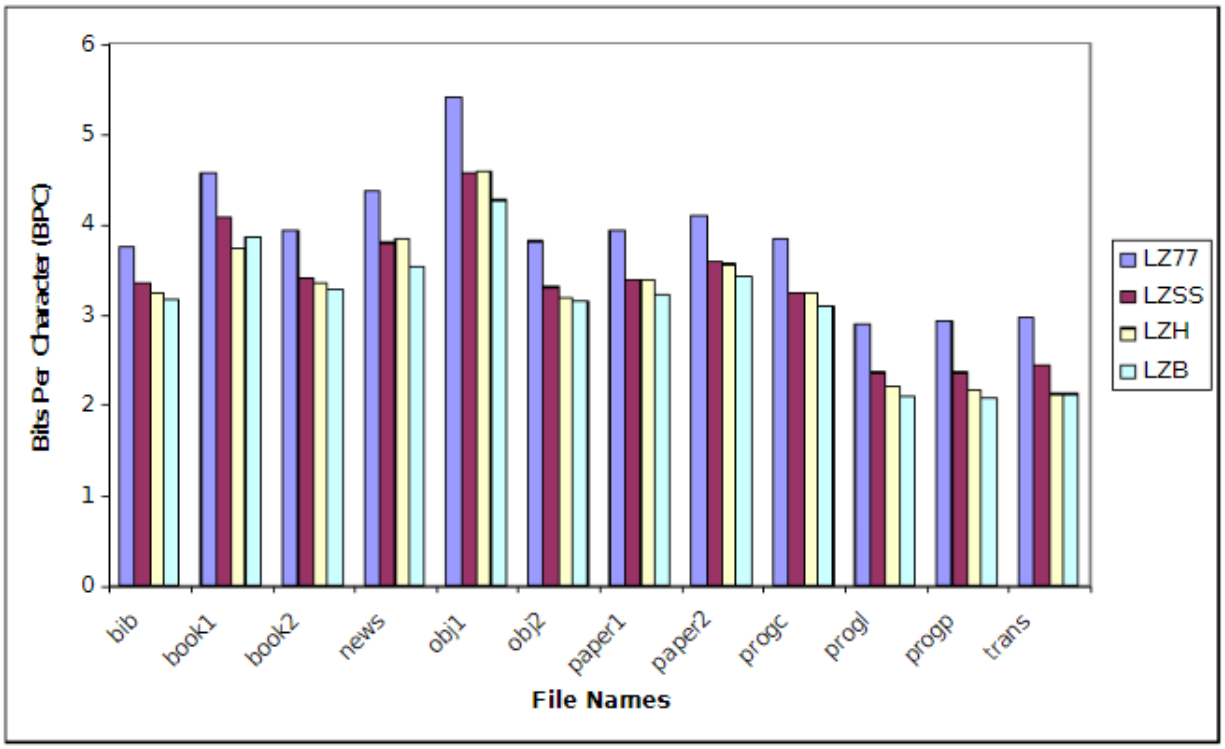

Рисунок 2.4 – Діаграма рівню стиснення для сімейства LZ77 [10]

Спробуємо зробити висновок за результатами, що були наведені на рисунку 2.4 про ефективність стиснення сімейства LZ77. Більшість файлів ASCII стискаються лише до половини початкового розміру, і в кожному файлі рівень стиснення є незмінним. Метод LZW, не маючи меж, приймає фрази, тому стиснення розширює файл «obj2» на 25%, що вважається недоліком цього підходу.

Крім того, з рисунка 2.5 очевидно, що продуктивність LZFG є найкращою серед цих методів, даючи середній BPC 2,89, що є дійсно значним. Серед сімейства LZ78 продуктивність LZFG є найкращою, оскільки схема, яку він використовує, ретельно відібрані коди для представлення покажчиків, що є найкращою схемою в сімействі LZ77. На рисунку 2.5 показано порівняння ступенів стиснення для сімейства LZ78 [19].

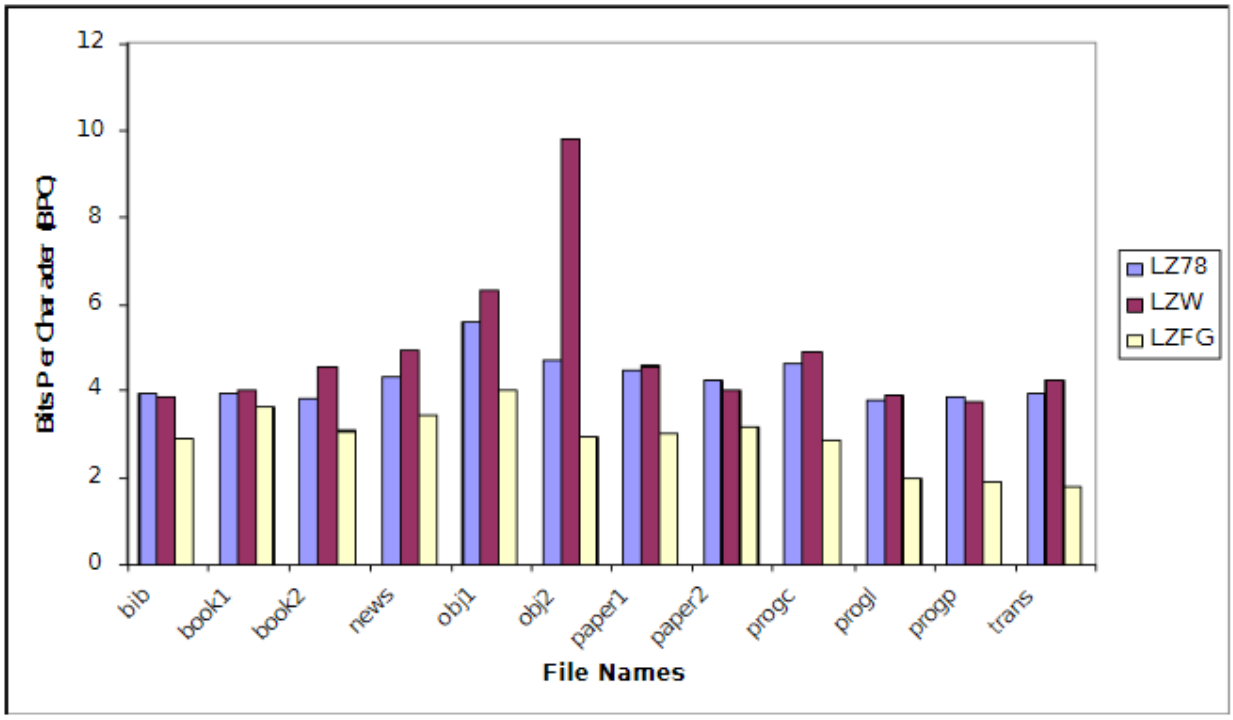

Рисунок 2.5 – Діаграма, що показує рівень стиснення для сімейства LZ78 [10]

2.3 Опис запропонованого алгоритму

Одним недоліком, який слід враховувати при реалізації алгоритму LZW, є постійно зростаюча таблиця рядків; у міру того, як аналізується більше даних, словник стає все більшим. Таблицею потрібно керувати, оскільки пам’ять комп’ютера обмежена. Два існуючі методи обробки постійно зростаючої таблиці рядків обговорюються нижче.

Table Freezing. Це метод, який використовується оригінальним алгоритмом LZW. Цей метод вибирає розмір таблиці рядків і не дозволяє таблиці перевищувати цей розмір. Замість цього він продовжує стиснення відповідно до замороженої таблиці. Це просто і легко, але воно погано працює з великими файлами [38].

Table Flushing. Цей метод періодично обчислює поточний ступінь стиснення. Коли таблиця заповнена і поточний коефіцієнт стиснення падає нижче деякого попередньо визначеного порогового значення, таблиця рядків очищується. Тобто, алгоритм відмовляється від поточної таблиці рядків і

створює нову під час стиснення решти вхідних даних. Промивання може позбутися від рідко використовуваних записів. Однак ця кардинальна операція також видаляє часто використовувані записи. Таким чином, це не покращує коефіцієнти стиснення для багатьох вхідних файлів.

Table Pruning. Пропонує обрізати рядкову таблицю. Коли таблиця рядків заповнюється і потрібен додатковий запис, замінюємо рідко використовуваний запис новим записом, і стиснення продовжується. Однак проблема вибору рідко використовуваного запису для скорочення є нетривіальною.

Запропонований алгоритм, застосований у цьому дослідженні, базується на алгоритмі LZW шляхом модифікації процесу формування словника та використання коду змінної довжини довжиною 8-12 біт [39]. Наприклад, існує послідовність символів, які потрібно закодувати, наприклад, wabba/bwabba/bwabba/bwabba/bwoo/bwoo/bwoo. Символ /b замість символу пробілу. Передбачається, що джерелом алфавіту є «/b, a, b, o, w», словник для початкових умов запропонованого нами алгоритму, наведеного в таблиці 2.4.

Таблиця 2.4 – Початкові умови словника запропонованого алгоритму

Індекс	Значення
1	/b
2	a
3	b
4	o
5	w

Кодеровщик спочатку розміщує вказівник prevDictionary у порожньому місці та шукає співпадіння «w» у словнику. Якщо його не було знайдено в словнику, то кодеровщик зчитує наступний символ і об’єднує його в співпадіння «wa». Якщо його не знайдено в словнику, кодеровщик повертає

індекс 5, додає співпадіння «wa» до словника з індексом 6, заповнює вказівник prevDictionary співпадінням «wa» та додає його до словника, якщо це співпадіння ще не зареєстровано. Далі пошук нового співпадіння почнеться з символу «a», а вказівник prevDictionary розміщується на символі «w», потім кодеровщик зчитує наступне співпадіння, яке є «b», і об’єднує його в «ab». Якщо цей співпадіння не знайдено в словнику, кодеровщик повертає індекс 2, додає співпадіння «ab» до словника з індексом 7, заповнює вказівник prevDictionary співпадінням «wab» і додає його до словника з індексом 8. Далі пошук нового співпадіння буде починатися з символу «b», а вказівник prevDictionary буде розміщено на символі «a». Цей процес триває, доки не буде закодована вся послідовність символів у джерелі даних. Процес формування словника для кодування наведено в таблиці 2.5.

Таблиця 2.5 – Формування в словнику для кодування запропонованого алгоритму

Індекс	**Значення**	**Індекс**	**Значення**
1	**2**	**3**	**4**
1	/b	18	/bwabb
2	a	19	ba/bw
3	b	20	bba/bw
4	o	21	wabba
5	w	22	/bwabba
6	wa	23	a/bwo
7	ab	24	ba/b
8	wab	25	oo
9	bb	26	woo
10	abb	27	o/b
11	ba	28	oo/b
12	bba	29	/bwo

Продовження таблиці 2.5

1	2	3	4
13	a/b	30	o/bwo
14	ba/b	31	oo/bw
15	/bw	32	woo/bw
16	a/bw	33	woo
17	wabb	34	/bwo

Після виконання процесу кодування для всіх символів, що містяться в алфавітному порядку, результат вихідної послідовності, згенерованої кодувальником, буде «5 2 3 3 2 1 8 14 17 16 4 4 15 28 5 25».

. Наступним процесом є виконання декодування, результатом вихідної послідовності, згенерованої кодувальником, є «5 2 3 3 2 1 8 14 17 16 4 4 15 28 5 25». Процес декодування зчитує послідовність символів, згенерованих під час процесу кодування, і формує той самий словник, що й процес кодування. Процес декодування зчитує послідовність символів результатів кодування як індекс у словнику.

Основною відмінністю запропонованого методу є те що додаємо додатковий вказівник який буде посилатися на попередній символ, коли наш алгоритм додав до словника нову послідовність.

На рисунках 2.6 і 2.7 показано процес стиснення запропонованого алгоритму.

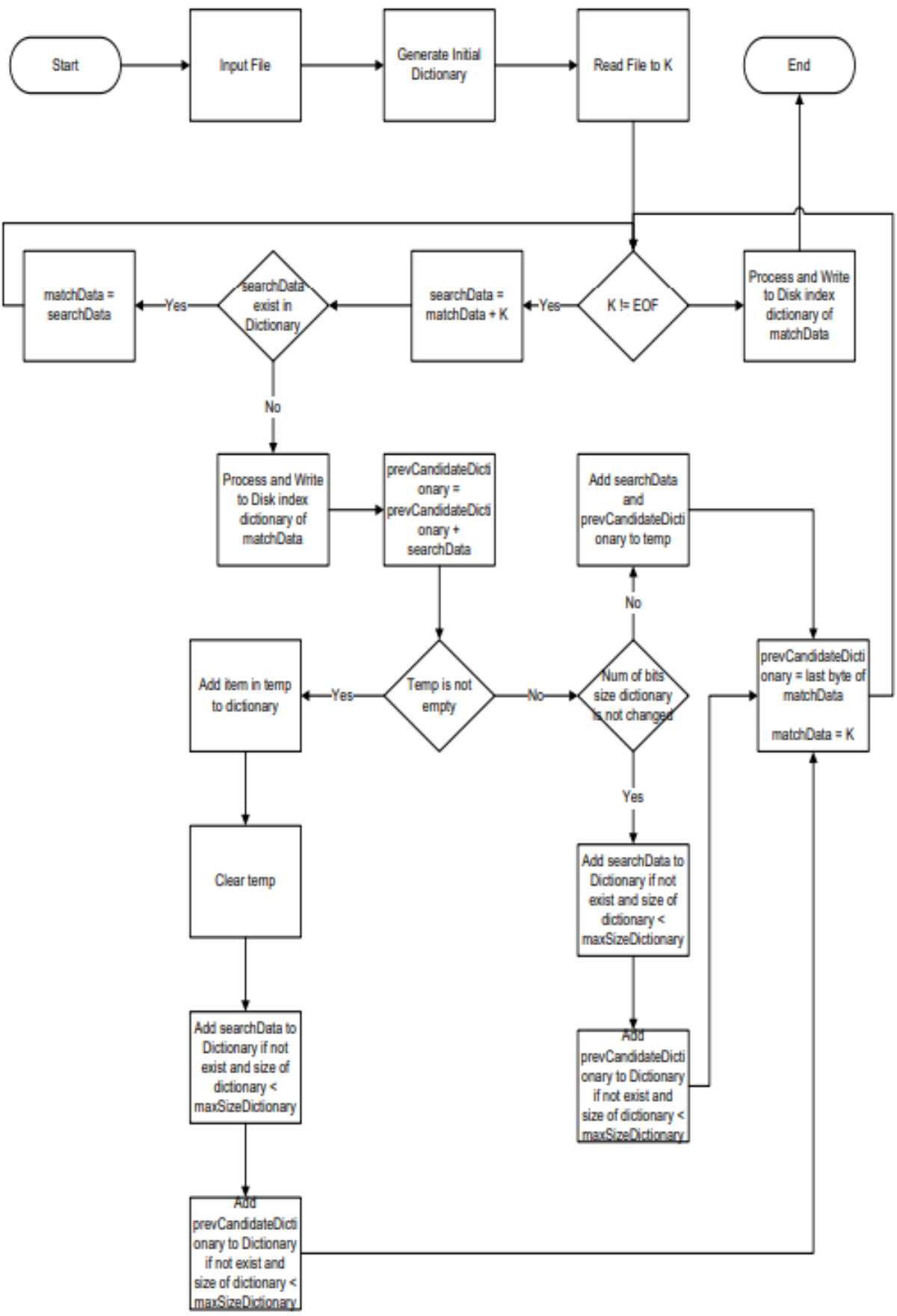

Рисунок 2.6 – Блок-схема запропонованого алгоритму стиснення

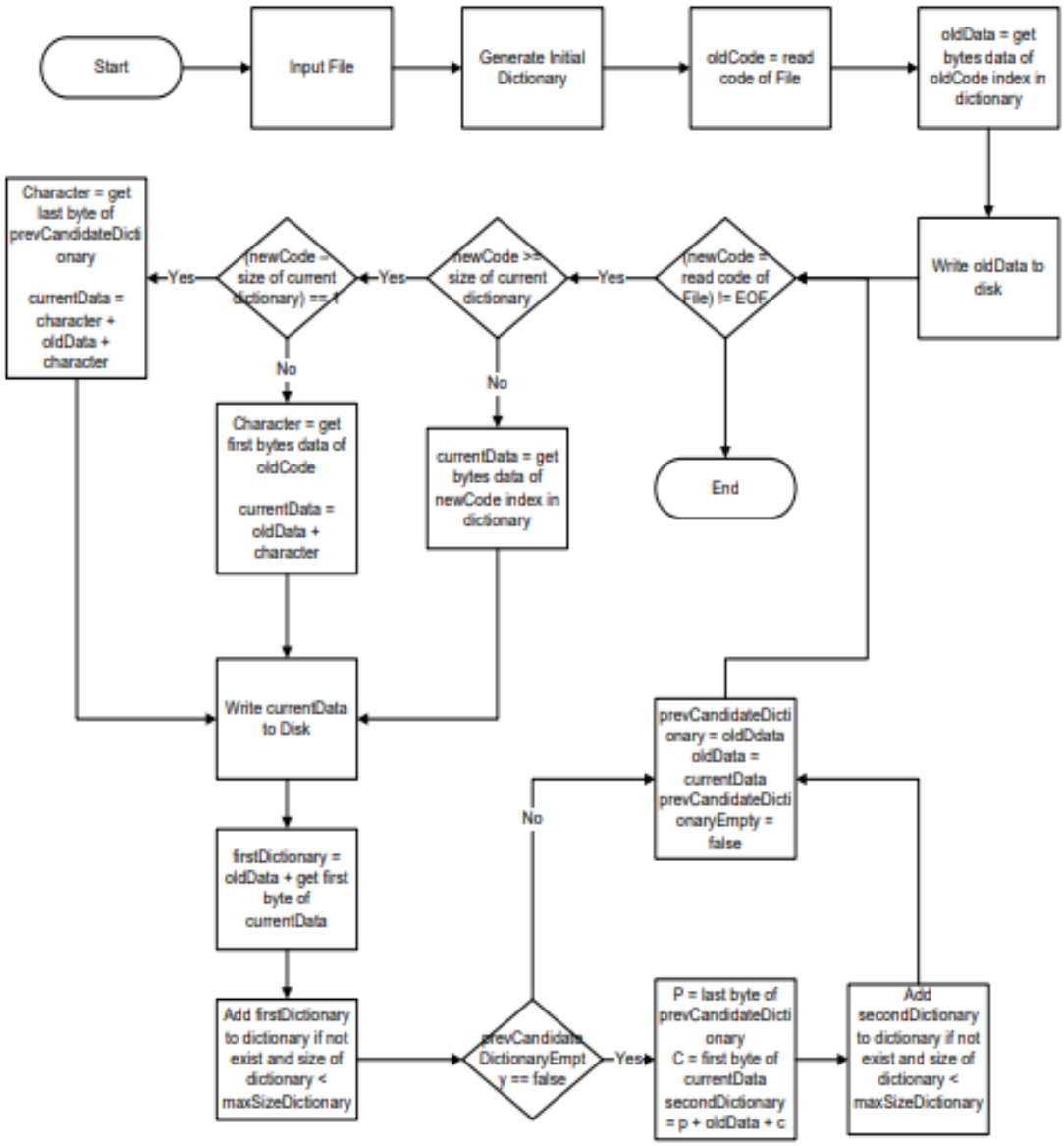

Рисунок 2.7 – Блок-схема запропонованого алгоритму декомпресії

Існує багато стратегій для вибору рідко використовуваних записів, проблема, подібна до вибору сторінок заміни для систем керування віртуальною пам’яттю. Тут наш запропонований алгоритм з принципами так званих «Page Replacement Algorithms»: Least Recently Used and Aging Replacement.

У LRU запис, до якого не було доступу протягом найдовшого часу, вибирається як запис для заміни. У даній реалізації використовуємо самоорганізований список, щоб вибрати запис, який використовувався найменше. Цей список містить індекс кожного запису таблиці рядків. Під час

стиснення кожного разу, коли здійснюється доступ до запису, відповідний індекс переміщується на початок списку. Якщо потрібен запис для заміни, він вибирається в кінці списку.

Окрім LRU, використовуєм алгоритм Aging Replacement для керування таблицею рядків. У цьому алгоритмі зберігаємо значення, яке називається часом життя (TTL) для кожного запису таблиці. Коли запис створюється, відповідний TTL ініціалізується деяким заздалегідь визначеним значенням. Періодично TTL знижується. Коли TTL стає нульовим, запис видаляється з таблиці рядків. Щоб доступ до таблиці, ближче до теперішнього часу, мав більший вплив, ніж доступ до таблиці давно, під час доступу до запису його TTL скидається до (поточне значення/2+початкове значення). Коли потрібен запис для заміни, буде вибрано невикористаний запис або запис із найменшим TTL.

Таким чином дана модифікація алгоритму LZW дозволяє краще створювати словник і за рахунок цього оптимізувати його побудову що є критичним для великих файлів, оскільки розмір таблиці рядків часто перевищує розмір доступної пам’яті.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ ЗАПРОПОНОВАНОГО МЕТОДУ

3.1 Вибір технологій для розробки

3.1.1 Мова програмування Java

Мова програмування Java спочатку була розроблена компанією Sun Microsystems, яка була ініційована Джеймсом Гослінгом і випущена в 1995 році як основний компонент платформи Java Sun Microsystems (Java 1.0 J2SE). Останнім випуском Java Standard Edition є Java SE 8. З розвитком Java та її широкою популярністю було створено кілька конфігурацій для різних типів платформ. Наприклад: J2EE для корпоративних програм, J2ME для мобільних програм [40]. Нові версії J2 були перейменовані на Java SE, Java EE та Java ME відповідно. Гарантовано, що Java буде писати один раз, працювати будь-де [41].

Java це:

– об’єктно-орієнтований: у Java все є об’єктом. Java може бути легко розширена, оскільки вона заснована на об’єктній моделі;

– незалежність від платформи: на відміну від багатьох інших мов програмування, включаючи C і C++, коли компілюється Java, вона компілюється не в машину, що залежить від платформи, а в незалежний від платформи байт-код. Цей байт-код розповсюджується в Інтернеті та інтерпретується віртуальною машиною (JVM) незалежно від того, на якій платформі він працює;

– простота: Java створена для легкого вивчення. Якщо ви розумієте основну концепцію ООП Java, освоїти її буде легко;

– безпечність: завдяки функції безпеки Java дозволяє розробляти системи, вільні від вірусів і несанкціонованого доступу. Методи автентифікації засновані на шифруванні з відкритим ключем;

– архітектурно нейтральний: компілятор Java генерує нейтральний щодо архітектури об’єктний формат файлу, який робить скомпільований код виконуваним на багатьох процесорах із наявністю системи середовища виконання Java;

– портативність: нейтральність до архітектури та відсутність залежних від реалізації аспектів специфікації робить Java портативною. Компілятор у Java написаний на ANSI C із чіткими межами переносимості, що є підмножиною POSIX;

– надійність: Java докладає зусиль для усунення ситуацій, які можуть викликати помилки, наголошуючи головним чином на перевірці помилок під час компіляції та перевірці під час виконання;

– багатопотоковість: за допомогою багатопотокової функції Java можна писати програми, які можуть виконувати багато завдань одночасно. Ця функція дозволяє розробникам створювати інтерактивні програми, які можуть працювати безперебійно;

– інтерпретація: байт-код Java транслюється на льоту у власні машинні інструкції та ніде не зберігається. Процес розробки є більш швидким і аналітичним, оскільки зв’язування є поетапним і легким процесом;

– висока продуктивність: завдяки використанню компіляторів Just-In-Time Java забезпечує високу продуктивність [42].

3.1.2 Середа розробки Intellij IDEA

IntelliJ IDEA − це інтелектуальна контекстно-залежна IDE для роботи з Java та іншими мовами JVM, такими як Kotlin, Scala та Groovy, у різноманітних програмах [43]. Крім того, IntelliJ IDEA Ultimate може допомогти вам розробити повний стек вебзастосунків завдяки своїм потужним інтегрованим інструментам, підтримці JavaScript і пов’язаних

технологій, а також розширеній підтримці популярних фреймворків, таких як Spring, Spring Boot, Jakarta EE, Micronaut, Quarkus, Helidon. Крім того, ви можете розширити IntelliJ IDEA за допомогою безкоштовних плагінів, розроблених JetBrains, що дозволить вам працювати з іншими мовами програмування, включаючи Go, Python, SQL, Ruby та PHP.

Intellij IDEA доступна у двох версіях:

– Community Edition;

– Ultimate Edition.

Порівняння між Community та Ultimate Edition зображено на рисунку 3.1.

Comparison Index	Community Edition	Ultimate Edition
License	Open-Source and free	Commercial
Language Support	Java, Kotlin, Groovy, Scala, Perl, Python, XML, Go	HTML, XHTML, CSS, Java, php, Python, Kotlin, Perl, Scala, Go, SQL, Ruby, JavaScript, Groovy etc.
Technology & Frameworks Support	Android, Ant, JavaFX, Junit, TestingNG, Gradle	Android, Ant, JavaFX, Junit, TestingNG, Gradle, Node.js, Spring, Struts, EjB, Django, OSGi etc.
Detecting Duplicate	Supported	Not Supported
Software Versioning and Revision Control	CVS, Git, GitHub	ClearCase, Perforce, CVS, Git, GitHub
Deployment	Docker, Docker Compose (via a plugin)	Docker, Docker Compose (via a plugin), Tomcat, Glassfish, Jboss, WebLogic, Jetty, Virgo etc.

Рисунок 3.1 – Порівняльна таблиця між версіями Community та Ultimate edition [23]

Перша версія IntelliJ IDEA була випущена в січні 2001 року. Це була перша Java IDE, яка має розширену навігацію по коду та можливості рефакторингу коду. У 2010 році він був визнаний найкращим доступним інструментом програмування для Java. У 2014 році Google випустила IDE з відкритим кодом для Android, яка також базується на IntelliJ IDEA [44].

Intellij має дуже багато крутих функції, які допомагають розробникам розробляти свій код у швидкому середовищі (рис. 3.2).

Рисунок 3.2 – Функціонал Intellij IDEA

Деякі з чудових функцій IntelliJ IDEA:

– Smart Code Completion;
– аналіз коду на льоту;
– розширений рефакторинг;
– виявлення дублікатів;

- перевірка та швидкі виправлення;
- ярлики для всього;
- термінал;
- навігація та пошук;
- підтримка інструментів і фреймворків;
- сервер застосунків.

Кожен аспект IntelliJ IDEA розроблено для максимального підвищення продуктивності розробника. Разом інтелектуальна допомога в кодуванні та ергономічний дизайн роблять розробку не тільки продуктивною, але й приємною.

Deep intelligence. Після того, як IntelliJ IDEA проіндексує ваш вихідний код, він пропонує надзвичайно швидкий і інтелектуальний досвід, надаючи відповідні пропозиції в будь-якому контексті: миттєве та розумне завершення коду, оперативний аналіз коду та надійні інструменти рефакторингу.

Out-of-the-box experience. Критично важливі інструменти, такі як інтегровані системи контролю версій і широкий спектр підтримуваних мов і фреймворків, – усі під рукою – жодних плагінів не включено.

Smart code completion. Хоч базове завершення пропонує назви класів, методів, полів і ключових слів у межах видимості, розумне завершення пропонує лише ті типи, які очікуються в поточному контексті.

3.1.3 Система контролю версій Git

На сьогоднішній день найпоширенішою сучасною системою контролю версій у світі є Git [45]. Git − це зрілий проект із відкритим кодом, який активно підтримується, спочатку розроблений у 2005 році Лінусом Торвальдсом, відомим творцем ядра операційної системи Linux. Приголомшлива кількість проектів програмного забезпечення покладаються

на Git для контролю версій, включаючи комерційні проекти, а також проекти з відкритим кодом. Розробники, які працювали з Git, добре представлені в пулі доступних розробників програмного забезпечення, і він добре працює в широкому діапазоні операційних систем та IDE (інтегрованих середовищ розробки).

Маючи розподілену архітектуру, Git є прикладом DVCS (отже, розподіленої системи керування версіями). Замість того, щоб мати лише одне місце для повної історії версій програмного забезпечення, як зазвичай у колись популярних системах контролю версій, таких як CVS або Subversion (також відомі як SVN), у Git робоча копія коду кожного розробника також є репозиторієм який може містити повну історію всіх змін [46].

Крім розповсюдження, Git було розроблено з урахуванням продуктивності, безпеки та гнучкості.

Необроблені характеристики продуктивності Git дуже сильні в порівнянні з багатьма альтернативами. Внесення нових змін, розгалуження, злиття та порівняння попередніх версій оптимізовано для продуктивності. Алгоритми, реалізовані в Git, використовують переваги глибоких знань про загальні атрибути реальних дерев файлів вихідного коду, як вони зазвичай змінюються з часом і які шаблони доступу.

На відміну від деяких програм контролю версій, Git не вводить в оману імена файлів, коли визначає, яким має бути сховище та історія версій дерева файлів, замість цього Git зосереджується на самому вмісті файлу. Зрештою, файли вихідного коду часто перейменовують, розділяють і змінюють порядок. Об’єктний формат файлів сховища Git використовує комбінацію дельта-кодування (зберігання відмінностей у вмісті), стиснення та явно зберігає вміст каталогу та об’єкти метаданих версії.

Git було розроблено з цілісністю керованого вихідного коду як головного пріоритету. Вміст файлів, а також справжні зв’язки між файлами та каталогами, версіями, тегами та комітами, усі ці об’єкти в сховищі Git захищені за допомогою криптографічно захищеного алгоритму хешування

під назвою SHA1. Це захищає код і історію змін як від випадкових, так і зловмисних змін і забезпечує повне відстеження історії.

З Git ви можете бути впевнені, що у вас є автентична історія вмісту вашого вихідного коду.

Однією з ключових цілей дизайну Git є гнучкість. Git є гнучким у кількох аспектах: у підтримці різних типів нелінійних процесів розробки, у своїй ефективності як у малих, так і у великих проектах, а також у своїй сумісності з багатьма існуючими системами та протоколами.

Git був розроблений для підтримки розгалуження та тегування як громадян першого класу (на відміну від SVN), а операції, які впливають на розгалуження та теги (такі як об’єднання або повернення), також зберігаються як частина історії змін. Не всі системи контролю версій мають такий рівень відстеження [47].

3.1.4 Засіб автоматизації збірки проєктів Maven

Maven − це потужний інструмент управління проєктами, який базується на POM (об’єктна модель проекту). Він використовується для побудови проектів, залежностей і документації. Це спрощує процес збирання, як ANT. Але він занадто просунутий, ніж ANT(принцип роботи maven представлений на рисунку 3.3). Коротко кажучи, можемо сказати, що maven − це інструмент, який можна використовувати для створення та керування будь-яким проектом на основі Java. Maven полегшує повсякденну роботу розробників Java і загалом допомагає зрозуміти будь-який проект на основі Java [48, 49].

Maven виконує багато корисних завдань, наприклад:

- можем легко створити проект за допомогою maven;
- можем легко додати jar та інші залежності проекту за допомогою maven;

– Maven надає інформацію про проект (документ журналу, список залежностей, звіти про модульне тестування тощо);

– Maven дуже корисний для проекту під час оновлення центрального сховища JAR та інших залежностей;

– за допомогою Maven можемо створювати будь-яку кількість проектів у типи виводу, такі як JAR, WAR тощо, без жодних сценаріїв;

– використовуючи maven, можемо легко інтегрувати наш проект із системою керування джерелами (наприклад, Subversion або Git).

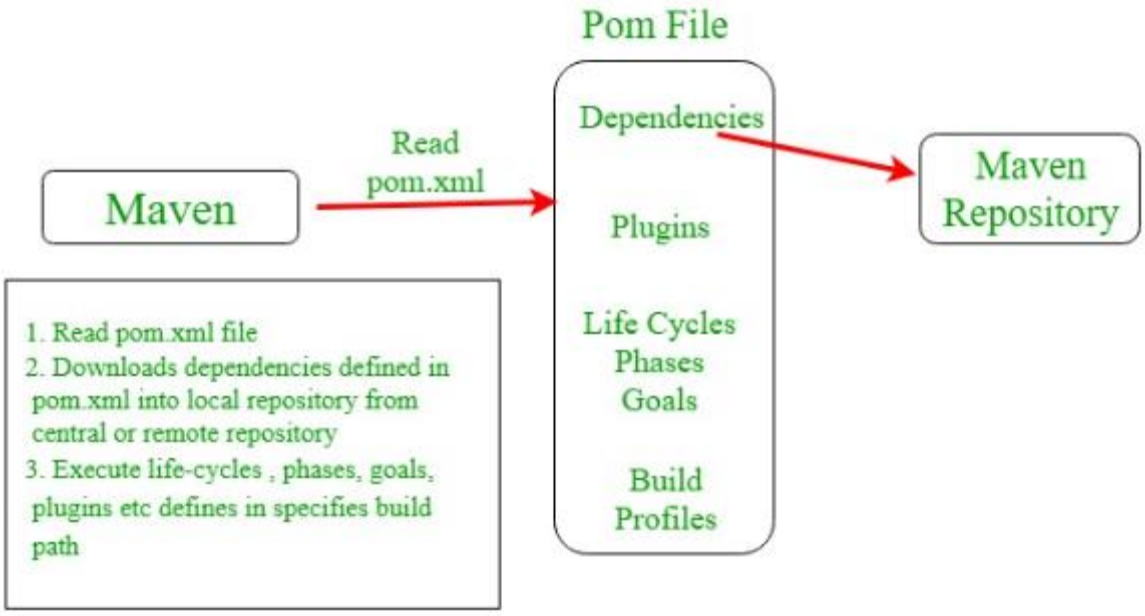

Рисунок 3.3 – Принцип роботи Maven

Основні концепції Maven:

– файли POM: файли об’єктної моделі проекту (POM) − це XML-файл, який містить інформацію, пов’язану з проектом, і інформацію про конфігурацію, таку як залежності, вихідний каталог, плагін, цілі тощо, які Maven використовує для створення проекту. Коли ви повинні виконати команду maven, ви надаєте maven файл POM для виконання команд. Maven читає файл pom.xml, щоб виконати його налаштування та операції;

– залежності та сховища: залежності − це зовнішні бібліотеки Java, необхідні для Project, а сховища − це каталоги упакованих файлів JAR.

Локальний репозиторій − це просто каталог на жорсткому диску вашого комп’ютера. Якщо залежності не знайдено в локальному репозиторії Maven, Maven завантажує їх із центрального репозиторію Maven і розміщує у вашому локальному репозиторії;

– життєві цикли збірки, фази та цілі: життєвий цикл збірки складається з послідовності етапів збірки, а кожна фаза збірки складається з послідовності цілей. Команда Maven − це назва життєвого циклу збірки, фази або цілі. Якщо запит на виконання життєвого циклу здійснюється за допомогою команди maven, усі фази побудови в цьому життєвому циклі також виконуються. Якщо запитано виконання фази збірки, усі фази збірки перед нею у визначеній послідовності також виконуються;

– профілі збірки: створюйте профілі набору значень конфігурації, які дозволяють створювати ваш проект з використанням різних конфігурацій. Наприклад, вам може знадобитися створити свій проект для локального комп’ютера для розробки та тестування. Щоб увімкнути різні збірки, ви можете додати різні профілі збірки до ваших файлів POM за допомогою його елементів профілів, які активуються різними способами;

– плагіни збірки: збірки що використовуються для досягнення певної мети. ви можете додати плагін до файлу POM. У Maven є кілька стандартних плагінів, які ви можете використовувати, і ви також можете реалізувати власні в Java.

3.2 Програмна реалізація покращеного методу

Для початку нам необхідно створити Maven проект для цього використаємо встроєну функції в Intellij IDEA, яка дозволяє нам вибрати необхідний архетип Maven проекту. Створений проект зображено на рисунку 3.4.

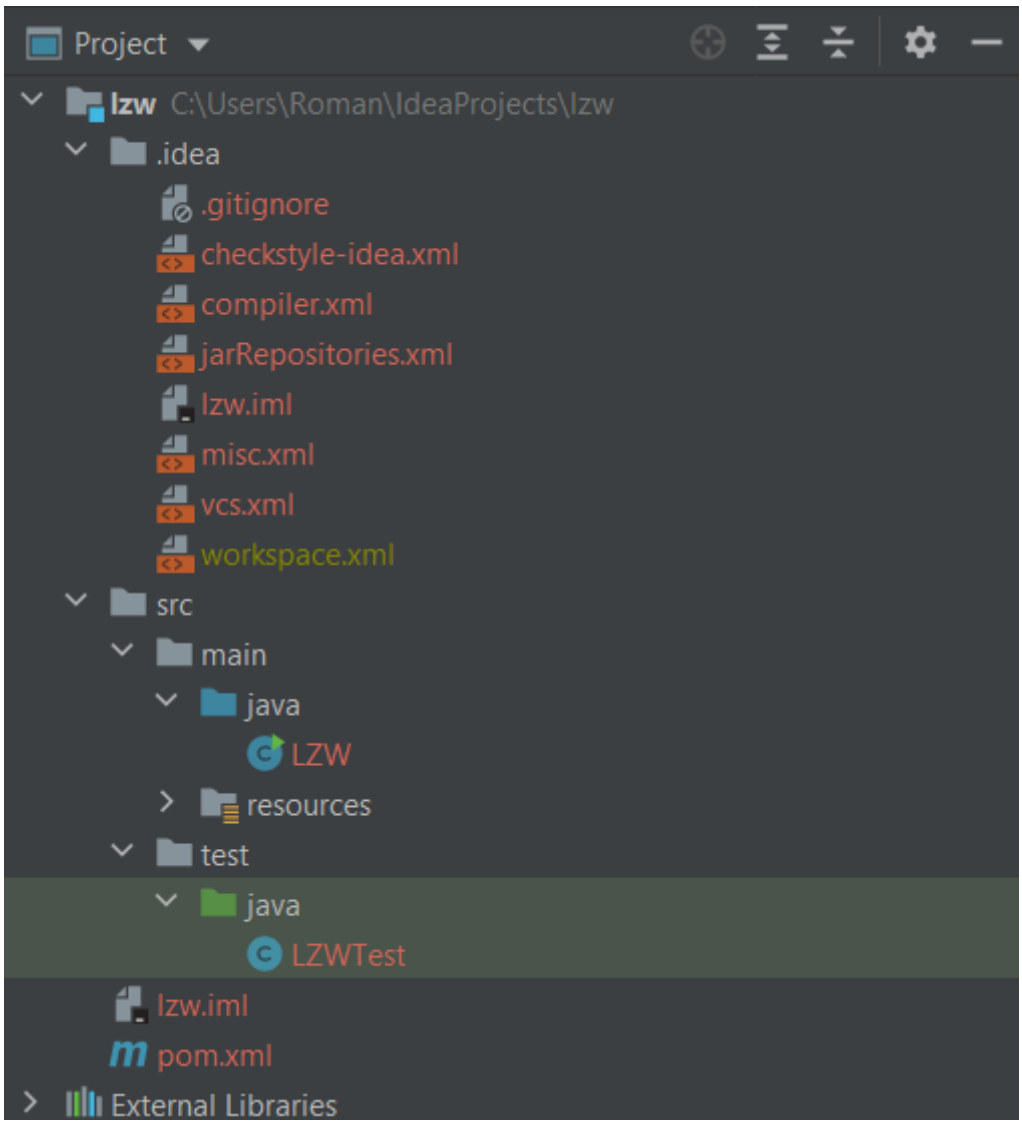

Рисунок 3.4 – Структура створеного проекту Maven

Розглянемо основні файли та директорії проекту:

– .idea – службова директорія для середи розробки Intellij IDEA;

– .gitignore – файл який дозволяє ігнорувати файли при заливанні даних в репозиторій(використовує ссистема контролю версіями git);

– pom.xml – файл який використовує Maven framework для підключення бібліотек, модулів та збірки проекту;

– lzw.iml – файл який зберігає інформацію про проект для Intellij IDEA.

Приступимо до розробки програми, та почнемо зі створення функції стиснення (рис. 3.5).

```
1    public static List<Integer> compress(String uncompressed) {
2        int dictSize = 256;
3        Map<String,Integer> dictionary = new HashMap<>();
4        for (int i = 0; i < 256; i++) {
5            dictionary.put(EMPTY + (char)i, i);
6        }
7
8        String w = EMPTY;
9        List<Integer> result = new ArrayList<>();
10       for (char c : uncompressed.toCharArray()) {
11           String wc = w + c;
12           if (dictionary.containsKey(wc))
13               w = wc;
14           else {
15               result.add(dictionary.get(w));
16               dictionary.put(wc, dictSize++);
17               w = EMPTY + c;
18           }
19       }
20
21       if (!w.equals(EMPTY)) {
22           result.add(dictionary.get(w));
23       }
24       return result;
25   }
```

Рисунок 3.5 – Код функції кодування даних

Спочатку на строчках 2-6, створюємо наш словник та заповняємо його базовими значеннями. На вході отримуємо строку з текстом. Далі проходимо циклом по всім символам та добавляємо у наш словник нові значення. Після цього повертаємо список наших закодованих значень.

Продовжимо розробку нашої програми з функції декодування (рис. 3.6).

```
1    public static String decompress(List<Integer> compressed) {
2        int dictSize = 256;
3        Map<Integer,String> dictionary = new HashMap<>();
4        for (int i = 0; i < 256; i++) {
5            dictionary.put(i, EMPTY + (char)i);
6        }
7
8        String w = EMPTY + (char)(int)compressed.remove(0);
9        StringBuffer result = new StringBuffer(w);
10       for (int k : compressed) {
11           String entry;
12           if (dictionary.containsKey(k)) {
13               entry = dictionary.get(k);
14           } else if (k == dictSize) {
15               entry = w + w.charAt(0);
16           } else {
17               throw new IllegalArgumentException("Bad compressed k: " + k);
18           }
19
20           result.append(entry);
21
22           dictionary.put(dictSize++, w + entry.charAt(0));
23
24           w = entry;
25       }
26       return result.toString();
27   }
```

Рисунок 3.6 – Код функції декодування даних

Спочатку на строчках 2-6, також створюємо наш словник та заповнюємо його базовими значеннями. На вході отримуємо список із закодованими символами. Далі проходимо циклом по всім закодованим символам та добавляємо у наш словник нові значення. Після цього повертаємо нашу строку в декодованому форматі.

Останньою функцією реалізації алгоритму LZW буде основна «main» функція для визову функцій що реалізували вище. Ця функція зображена на рисунку 3.7.

```
1    public static void main(String[] args) throws IOException {
2
3        String content = Files.readString(PATH);
4
5        List<Integer> compressed = compress(content);
6
7        String decompressed = decompress(compressed);
8
9    }
```

Рисунок 3.7 – Код основної функції стиснення даних

Тепер перейдемо до покращення алгоритму то використаємо код що написали вище для звичайного алгоритму LZW. З нового у нас тут з’явиться новий вказівник *prevDictionary* який буде посилатися на попередній символ.

Також для цього змінили наш цикл «foreach» на звичайний «fori» для того щоб можна було зручно брати і посилатися на необхідні символи.

Основна функція запуску залишилася без змін, тобто змінилась тільки логіка алгоритма, а так все працює як раніше (рис. 3.8). Реалізацію функціїї кодування та декодування покращенного алгоритму представлено на рисунках 3.9 та 3.10.

```
1    public static void main(String[] args) throws IOException {
2
3        String content = Files.readString(PATH);
4
5        List<Integer> compressed = compress(content);
6
7        String decompressed = decompress(compressed);
8
9    }
```

Рисунок 3.8 – Код основної функції покращеного алгоритму стиснення даних

```
public static List<Integer> compress(String uncompressed) {
    int dictSize = 256;
    Map<String,Integer> dictionary = new HashMap<>();
    for (int i = 0; i < 256; i++) {
        dictionary.put(EMPTY + (char)i, i);
    }

    String w = EMPTY;
    Integer prevDictionary = 0;
    List<Integer> result = new ArrayList<>();
    char[] charArray = uncompressed.toCharArray();
    for (int i = 0, charArrayLength = charArray.length; i < charArrayLength; i++) {
        char c = charArray[i];
        String wc = w + c;
        if (dictionary.containsKey(wc))
            w = wc;
        else {
            result.add(dictionary.get(w));
            dictionary.put(wc, dictSize++);
            w = EMPTY + c;
            prevDictionary = i;
            if (prevDictionary != 0) {
                String s = String.valueOf(charArray[prevDictionary]);
                dictionary.put(s + wc, dictSize++);
            }
        }
    }

    if (!w.equals(EMPTY)) {
        result.add(dictionary.get(w));
    }

    return result;
}
```

Рисунок 3.9 – Код функції кодування даних покращеного алгоритму

Також важно зазначити що новий вказівник *prevDictionary* який посилається на попередній символ буде збільшує розмір словника, але за допомогою цього можем збільшити ступінь стиснення. При декодуванні буде відбуватись зворотня операція як в стандартному алгоритмі LZW, тільки вже з додатковим вказівником prevDictionary.

```
1    public static String decompress(List<Integer> compressed) {
2        int dictSize = 256;
3        Map<Integer,String> dictionary = new HashMap<>();
4        for (int i = 0; i < 256; i++) {
5            dictionary.put(i, EMPTY + (char)i);
6        }
7
8        String w = EMPTY + (char)(int)compressed.remove(0);
9        StringBuffer result = new StringBuffer(w);
10       Integer prevDictionary = 0;
11       for (int i = 0, compressedSize = compressed.size(); i < compressedSize; i++) {
12           int k = compressed.get(i);
13           String entry;
14           if (dictionary.containsKey(k)) {
15               entry = dictionary.get(k);
16           } else if (k == dictSize) {
17               entry = w + w.charAt(0);
18               prevDictionary = i;
19           } else {
20               throw new IllegalArgumentException("Bad compressed k: " + k);
21           }
22
23           result.append(entry);
24
25           dictionary.put(dictSize++, w + entry.charAt(0));
26           Integer integer = compressed.get(prevDictionary);
27           String preSymbol = dictionary.get(integer);
28           dictionary.put(dictSize++, preSymbol + w + entry.charAt(0));
29
30           w = entry;
31       }
32
33       return result.toString();
34   }
```

Рисунок 3.10 – Код функції кодування даних покращеного алгоритму

Реалізація ідеї щодо алгоритмів Least Recently Used та Aging Replacement дещо ускладнена головним чином через представлення та керування таблицею рядків [50]. Щоб прискорити процес пошуку в таблиці рядків, для реалізації таблиці рядків використовується техніка подвійного хешування. Щоб досягти хорошої продуктивності хеш-таблиці, розмір хеш-таблиці на 25% перевищує необхідний розмір таблиці рядків. Через хешування видалити або замінити запис таблиці рядків неможливо безпосередньо. Щоб замінити запис, нам потрібно позначити запис як

видалений і використати невикористаний запис для нового запису. Через це нам потрібно очистити позначені записи до того, як хеш-таблиця заповниться. Для цього нам потрібно періодично відтворювати хеш-таблицю. Крім того, якщо для вибору рідко використовуваних записів використовується алгоритм LRU, для реалізації самоорганізованого списку додається пов'язаний список. Якщо використовується алгоритм застарілої заміни, додається купа, щоб прискорити процес пошуку запису з найменшим TTL.

Максимальний розмір таблиці рядків визначає кількість бітів, необхідних для представлення кодового слова, тобто індексу таблиці рядків. Чим більший розмір, тим більша кількість бітів знадобиться для представлення індексу. Щоб стиснути невеликий файл, менша таблиця призводить до меншого стисненого файлу. Щоб стиснути великий файл, менша таблиця містить менше рядків і, таким чином, менше шансів використовувати індекс для кодування довгого рядка символів і, таким чином, зменшити ефективність стиснення. Алгоритми зменшують розмір стисненого файлу за допомогою таблиць змінної довжини. Максимальна кількість біт, яку можна зберегти, становить 3840. Для великих файлів з мільйонами байт це несуттєво. Щоб повністю використовувати всі можливі комбінації бітів стисненого кодового слова, розмір таблиці рядків є ступенем 2. Після експериментів із різними розмірами таблиці в діапазоні від 212 до 222 виявили, що таблиця розміром 2^{16}, тобто 65536, добре працює з великі текстові файли.

Хеш-таблицю потрібно створити заново, перш ніж вона заповниться. Однак, якщо таблицю відтворюють занадто часто, швидкість програми значно знижується. Крім того, згідно з нашими спостереженнями, різна довжина періоду призводить до різного ступеня стиснення. Згідно з нашим дослідженням, для таблиці розміром 65536 оптимальним періодом для відновлення таблиці рядків є стиснення 4096 рядків.

Повторне створення хеш-таблиці є трудомістким процесом, під час якого необхідно отримати доступ до кожного запису таблиці. Щоб зменшити вплив на швидкість керування хеш-таблицею, поєднуємо завдання відтворення хеш-таблиці із завданням зменшення TTL. Тобто відновлення хеш-таблиці та зменшення TTL виконуються одночасно.

Декомпресія є простим завданням відносно стиснення. Оскільки немає необхідності шукати в таблиці рядків, техніка хешування не потрібна, і тому немає потреби періодично відтворювати хеш-таблицю. Однак для LZW/старіння та LZW/LRU все ще потрібна купа або список, що самоорганізується. Метою включення купи або списку, що самоорганізується, є синхронізація таблиці рядків декомпресії з таблицею рядків стиснення, щоб дві таблиці використовували однакову послідовність записів заміни.

3.3 Тестування покращеного методу

Щоб оцінити ефективність методів, протестуємо ці методи за допомогою тестових файлів із вебсайту Canterbury Corpus (http://corpus.canterbury.ac.nz). Кентерберійський корпус є еталоном, який дозволяє дослідникам оцінювати методи стиснення без втрат. Результати предаслені в таблицях 3.1 та 3.2. Три тестові файли E.coli, bible.txt і world192.txt знаходяться у великій колекції корпусів Кентерберійського корпусу. У цих експериментах використовували таблиці рядків розміром 65536, період відтворення хеш-таблиці 4096 і початкове значення TTL 64.

Таблиця 3.1 – Розміри стиснених файлів

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

Таблиця 3.2 – Коефіцієнти стиснення файлів

	E.coli	bible.txt	world192.txt
LZW	3,82	2,85	2,67
LZW/aging	3,87 (+1%)	3,26 (+12%)	3,07 (+13%)
LZW /LRU	3,76 (-1%)	3,13 (+9%)	2,91 (+8%)

Окрім тестових файлів із Кентерберійського корпусу, також було протестувано дані методи з іншими текстовими файлами. Тести на стиснення цих файлів дали такі результати:

– LZW/Aging працює краще, ніж LZW/LRU у 90% випадків;

– LZW/Aging може покращити ступінь стиснення LZW на 10-15% для 90% перевірених файлів.

Попередні тести даних методів із файлами відео та зображень також дали обнадійливі результати. Оригінальний LZW постійно збільшує файли відео та зображень приблизно на 25%. Дана система LZW/aging може постійно зменшувати кількість файлів відео та зображень на 1%. Іншими словами, LZW/старіння може підвищити коефіцієнт стиснення на 26% для великих файлів відео або зображень порівняно з оригінальним LZW.

ВИСНОВКИ

У рамках кваліфікаційної роботи був розроблений і реалізований метод покращення побудови словника алгоритму стиснення LZW.

Описано вдосконалення стиснення даних LZW, яке використовує методи скорочення таблиці. За допомогою більш ефективного керування динамічним словником можна досягти кращого коефіцієнта стиснення. Зокрема, показно, що LZW/Aging може значно покращити ступінь стиснення для більшості великих файлів. Відповідно до експериментів визначено чотири фактори, які мають вирішальне значення для ступеня стиснення LZW/Aging та LZW/LRU. Такими факторами є розмір таблиці рядків, період повторного створення хеш-таблиці, інтервал зменшення TTL і початкове значення TTL. Необхідно провести подальшу роботу, щоб охарактеризувати комбінаторні ефекти цих факторів і визначити їх оптимальні комбінації. Незважаючи на те, що алгоритм старіння забезпечив значне покращення порівняно лише зі стисненням LZW, слід вивчити додаткові алгоритми заміни.

Результати дослідження апробовано у вигляді тез [51] у рамках XXXVII міжнародної науково-практичної конференції «Modern ways of solving the latest problems in science», 20–23 вересня 2022 року, Варна, Болгарія.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

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

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

3. Винарский, В. Я., Машталир, С. В., Сакало, Е. С., & Щербинин, К. С. (2009). Робастные алгоритмы самообучения карты Кохонена в задаче обработки изображений.

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

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

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

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

8. Mashtalir, S., & Mashtalir, V. (2020). Spatio-temporal video segmentation. In Advances in Spatio-Temporal Segmentation of Visual Data (pp. 161-210). Springer, Cham.

9. Mashtalir, S., & Mashtalir, V. (2016, August). Sequential temporal video segmentation via spatial image partitions. In 2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP) (pp. 239-242). IEEE.

10. Bodyanskiy, Y. V., Tyshchenko, O. K., & Mashtalir, S. V. (2019, June). Fuzzy clustering high-dimensional data using information weighting. In International Conference on Artificial Intelligence and Soft Computing (pp. 385-395). Springer, Cham.

11. Bodyanskiy, Y., Shafronenko, A., & Mashtalir, S. (2019, May). Online Robust Fuzzy Clustering of Data with Omissions Using Similarity Measure of Special Type. In International Scientific Conference “Intellectual Systems of Decision Making and Problem of Computational Intelligence” (pp. 637-646). Springer, Cham.

12. Mashtalir, S. V., Stolbovyi, M. I., & Yakovlev, S. V. (2019). Clustering video sequences by the method of harmonic k-means. Cybernetics and Systems Analysis, 55(2), 200-206.

13. Власенко, Н. В., Канунников, А. С., & Машталир, С. В. (2013). Компрессирование описания визуальных объектов путем фильтрации его компонент по критерию стабильности. Bulletin of National Technical University" KhPI". Series: System Analysis, Control and Information Technologies, (2 (976)), 91-100.

14. Машталир, С. В., & Сакало, Е. С. (2008). Адаптивное нейросетевое сжатие сигналов большой размерности на основе взвешенного информационного критерия.

15. Винарский, В. Я., Машталир, С. В., Сакало, Е. С., & Щербинин, К. С. (2009). Робастные алгоритмы самообучения карты Кохонена в задаче обработки изображений.

16. Bodyanskiy, Y., Grimm, P., Mashtalir, S., & Vinarski, V. (2010, July). Fast training of neural networks for image compression. In Industrial Conference on Data Mining (pp. 165-173). Springer, Berlin, Heidelberg.

17. Ведмедь, А. Г., Машталир, С. В., & Сакало, Е. С. (2010). Восстановление изображений с использованием анализа главных и независимых компонент. Системи обробки інформації, (6), 66-72.

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

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

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

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

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

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

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

25. Simons, S., Abasolo, D., & Hughes, M. (2015). Investigation of Alzheimer's disease EEG frequency components with Lempel-Ziv complexity. In 6th European Conference of the International Federation for Medical and Biological Engineering (pp. 46-49). Springer, Cham.

26. Aboy, M., Hornero, R., Abásolo, D., & Álvarez, D. (2006). Interpretation of the Lempel-Ziv complexity measure in the context of biomedical signal analysis. IEEE transactions on biomedical engineering, 53(11), 2282-2288.

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

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

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

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

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

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

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

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

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

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

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

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

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

40. Java Tutorial. URL: https://www.tutorialspoint.com/java/java_tutorial.pdf (дата звернення 14.10.2022).

41. TheServerSide Java Defenition. URL: https://www.theserverside.com/definition/Java (дата звернення 14.10.2022).

42. Офіційна документація Java SE 17. URL: https://docs.oracle.com/en/java/javase/17/ (дата звернення 14.10.2022).

43. Офіційний сайт Jetbrains. URL: https://www.jetbrains.com/idea/features/ (дата звернення 15.10.2022).

44. IntelliJ IDEA Tutorial. URL: https://www.javatpoint.com/intellij-idea-tutorial (дата звернення 15.10.2022).

45. What is Git and Why Should You Use it? URL: https://www.nobledesktop.com/learn/git/what-is-git (дата звернення 16.10.2022).

46. Офіційна документація git. URL: https://git-scm.com/docs (дата звернення 17.10.2022).

47. What is Git. URL: https://www.atlassian.com/git/tutorials/what-is-git (дата звернення 17.10.2022).

48. Офіційна документація Maven. URL: https://maven.apache.org/what-is-maven.html (дата звернення 18.10.2022).

49. Introduction to Apache Maven | A build automation tool for Java projects. URL: https://www.geeksforgeeks.org/introduction-apache-maven-build-automation-tool-java-projects/ (дата звернення 18.10.2022).

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

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