КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

ПЕРЕТВОРЕННЯ БАРРОУЗА – ВІЛЛЕРА ТА ЙОГО ЗАСТОСУВАННЯ

Виконав студент 4 курсу
Волохович Ігор Ігорович _____
(підпис)

Науковий керівник:
Професор, доктор фіз.-мат. наук
Анісімов Анатолій Васильович _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики

«____»_____202__р.

протокол №_____

Завідувач кафедри

Терещенко В.М _____
(підпис)

Київ – 2021

Реферат

Обсяг роботи 48 сторінок, 10 ілюстрацій, 50 джерел посилань.

ПЕРЕТВОРЕННЯ БАРРОУЗА-УІЛЛЕРА, МЕТОДИ СТИСНЕННЯ ДАНИХ, АЛГОРИТМ ЛЕМПЕЛЯ–ЗІВА–ВЕЛЧА, АЛГОРИТМ ХАФФМАНА

Об’єктом роботи є дослідження методів стиснення даних, перетворення Барроуза –Віллера[9] та доцільність його використання у методах стиснення.

Метою роботи є розробка алгоритмів стиснення даних та перетворення Барроуза–Віллера[9]. Модифікація одного з алгоритму та порівняння якість та час стиснення даних.

Методи розроблення: аналіз алгоритмів стискання, кодування методів стиснення, аналіз якості та часу стиснення алгоритмів та їх модифікацій, розробка загальної бібліотеки алгоритмів, розробка прикладів використання алгоритмів, розробка веб–застосунку для демонстрації можливостей бібліотеки.

Інструменти розроблення. Для розробки реалізації було використано інтерактивне середовище розробки JebBrains Rider[1]. Було використано мову C#[2]. Створено .NET Library[3] та використовуючи ASP.NET Framework [4] створено веб – застосунок. Розроблено систему логування часу та обрахування відсотку стиснення.

Результат роботи. Було досліджено методи стиснення даних та вже існуючі продукти стиснення на ринку. Розроблено декілька демонстраційних консольних програм та веб-застосунок. Також було запроваджено стандартну загальну бібліотеку методів, яка було викладена в мережу на правах open-source та доступна всім розробникам. Було показано використання бібліотеки на різних платформах. Було продемонстровано, що модифіковані перетворенням Барроуза – Віллера алгоритми стиснення дають значне

покращення ступеню стиснення на великих файлах. Також продемонстровано в деяких випадках нераціональність використання такого модифікованого алгоритму та його наслідки у стисненні, які виражаються у збільшенні розміру файлу. Також було розроблено та продемонстровано найоптимальніший модифікований алгоритм Хаффмана з використанням перетворення Барроуза – Віллера та перетворення «Переміщення до початку», яке дало значний приріст(близько 40%) ступеню стиснення та успішно конкурує з наявними на ринку алгоритмами.

Зміст

ВСТУП

Оцінка сучасного стану об’єкта дослідження. На сьогоднішній день, зі зростанням кількості користувачів та кількості даних в мережі інтернет гостро постає питання зберігання даних. Так як будь-яка людина розуміє, що просто зберігати файли не дуже правильно, люди замислюються щодо використання алгоритмів стиснення задля збереження дорогого та цінного місця на серверах, комп’ютерах, мережевих системах. Через це були придумані та розроблені різні алгоритми стиснення даних та допоміжні перетворення для збільшення якості стиснення. Наприклад великій продуктовій компанії Amazon не вигідно зберігати сотні мільйонів даних про користувачів, товари та історію у первісному вигляді. Це займає багато місця. Тому вони використовують алгоритми, які допомагають в такому же об’ємі пам’яті зберігати набагато більше файлів.

Актуальність роботи та підстави для її виконання. Застосування, розробка нових та модифікація існуючих алгоритмів стиснення має важливе значення зі збільшенням генерації інформації у сьогоднішній час. Задля раціонального використання пам’яті та попередження надлишкової інформаційної катастрофи, коли не вистачатиме пам’яті. Методи стиснення та модифіковані (наприклад з використанням алгоритму Барроуза–Віллера) допомагають зменшити розмір файлу тим самим зменшуючи витрати та попередження настання інформаційної катастрофи.

Мета й завдання роботи. Метою дипломної роботи є огляд алгоритму перетворення Барроуза–Віллера, модифікація алгоритму Лемпеля – Зіва – Велша використовуючи цей алгоритм та порівняльний аналіз результатів стиснення, які приведуть до висновку модифікування алгоритмів у такий спосіб. Також завданням є реалізація власне перетворення BWT(Барроуза – Віллера)[9], алгоритму стиснення LZW(Лемпеля – Зіва – Велша)[12], модифікація даного алгоритму використовуючи перетворення та реалізація алгоритму Huffman coding(алгоритм стиснення Хаффмана)[8]. Порівняння

даних результатів у розрізі часу виконання та результуючому розмірі файлу. Для досягнення цієї поставлено такі завдання:

- Огляд методу BWT[9].
- Огляд основних методів та алгоритмів стиснення.
- Провести роботу у дослідженні розробки бібліотек
- Розробити програмну реалізацію алгоритму BWT[9], LZW[12], LZW + BWT, Huffman coding[8] з можливістю використовувати у будь-якому проекті на базі мови C#[2].

Об'єкт, методи й засоби розроблення. Об’єктом порівняльного аналізу є стандартні та модифіковані алгоритми стиснення даних. Об’єктом розробки є декілька програмних реалізацій, які допомагають побачити та застосувати бажаний алгоритм стиснення та бібліотеку з алгоритмами, яка може бути використана та встановлена будь-яким розробником через NuGet[4] пакет у середовищі мови програмування C#[2]

Можливі сфери застосування. Дана реалізація доступна у вигляді розширюваної бібліотеки для вбудовування у будь – які додатки на розсуд розробника. Розроблені алгоритми допомагають зменшити розмір файлів не пошкоджуючи якість даного файлу.

РОЗДІЛ 1. ЗАГАЛЬНІ ВІДОМОСТІ ПРО СТИСНЕННЯ ДАНИХ

1.1 Що таке стиснення даних?

Стиснення даних – це зменшення кількості бітів, необхідних для репродукції даних. Такі алгоритми допомагають зберігати пам’ять, збільшити швидкість передачі даних та зменшити вартість накопичувачів та необхідну пропускну властивість інтернету для передачі. Іншими словами, це процес кодування, реструктуризації або модифікування даних для зменшення її розміру. Фундаментально, це включає повторне шифрування інформації, користуючись меншими бітами (шматками), ніж оригінальне представлення. Стиснення даних підлягає компромісу між кількістю пам’яті, яку займає та часом. Це широко застосовується в обчислювальних сервісах та рішеннях, особливо в комунікації даними, Стискання даних працює за допомогою декількох методів стиснення та програмних рішень, які використовують алгоритми стискання для зменшення обсягу даних. [20]

1.2 Чому стиснення даних так важливо?

Стиснення даних може значно зменшити обсяг пам'яті, займаний файлом. Наприклад, при співвідношенні стиснення 2:1, файл розміром 20 мегабайт (МБ) займає 10 МБ місця. В результаті стиснення, компанії витрачають менше грошей і менше часу на зберігання. Стиснення оптимізує продуктивність сховища резервних копій і нещодавно проявилося в скороченні обсягу даних первинного сховища. Стиснення буде важливим методом скорочення даних, оскільки кількість даних продовжує зростати експоненційно.

Практично будь-який тип файлів може бути стиснутий, але важливо дотримуватися рекомендацій при виборі того, які з них стискати. Наприклад, деякі файли вже можуть бути стиснуті, тому стиснення цих файлів не матиме істотного впливу або може випливти в більший розмір файлу.

1.3 Історія створення методів стиснення

Теоретичну базу стискання даних було розроблено Клодом Шеноном, який опублікував роботи по алгоритмічній теорії інформації(AIT) [13] для компресії без втрат та теорії швидкості спотворення[14] для компресії з втратами. Вони стали фундаментом започаткування стискання даних, які на сьогоднішній день досягли величезних здібностей та дуже часто застосовуються у машинному навчанні.

З народженням інтернету в 70 – х роках, взаємозв'язок між розміром файлів та швидкістю передачі даних став більш значущим. Видатні розуми всього світу боролися з цією проблемою роками, але безрезультатно допоки не з'явився у середині 80 – х роках універсальний алгоритм стиснення без втрат Лемпеля – Зіва – Велша (LZW)[12], який показав реальні переваги стискання. [21]

Метод LZW був першим широко вживаним алгоритмом стискання даних, який було розроблено на комп'ютерах та досі використовується сьогодні (у різноманітних варіаціях). Великий текст англійською мовою може бути стиснутий приблизно на 50% використовуючи LZW[6]. Але історія методів починає задовго до цього. Азбука Морзе, яка була створена у 1838 році, являється предком такого поняття як «стискання даних». Тобто, це є одним із найперших методів стискання. Таким алгоритмом, в якому найпоширеніші літери англійської мові, такі як «e» і «t» мають коротші азбуки Морзе.

Пізніше, в 1949 році, коли почали освоюватися мейнфрейми, вже відомий Клод Шеннон разом з Робертом Фано розробили кодування Шеннона – Фано[11]. Їх алгоритм призначав коди символам у заданих блоках базуючись на ймовірності появи символу. Ймовірність появи символу обернено пропорційна довжині коду, що призводить до більш короткого способу презентації даних[5].

Два роки потому, Девід Хаффман вивчав курс «Теорія Інформації» в Массачусетському Технологічному Інституті (MIT) та мав заняття, викладачем яких був Робертом Фано. Викладач запропонував класу вибір: написати курсову робочу або складати випускний іспит. Авжеж Хаффман виправ курсову, тема якої була «Пошук найбільш ефективного методу двійкового кодування». Попрацювавши кілька місяців та нічого не придумавши, Хаффман вже збирався викинути всю свою роботу й почати готуватися до випускного іспиту замість здачі курсової. Саме в цей час йому в голову прийшла ідея, просвітництво та він придумав дещо схожу на техніку Шеннона – Фано[11], але набагато ефективнішу. Ключова відмінність між кодуванням Хаффмана та зазначеним вище було те, що у алгоритмі Шеннона – Фано[11] дерево ймовірностей будувалося знизу вверх, створюючи неоптимальний результат, а в цьому алгоритмі – зверху вниз. [15]

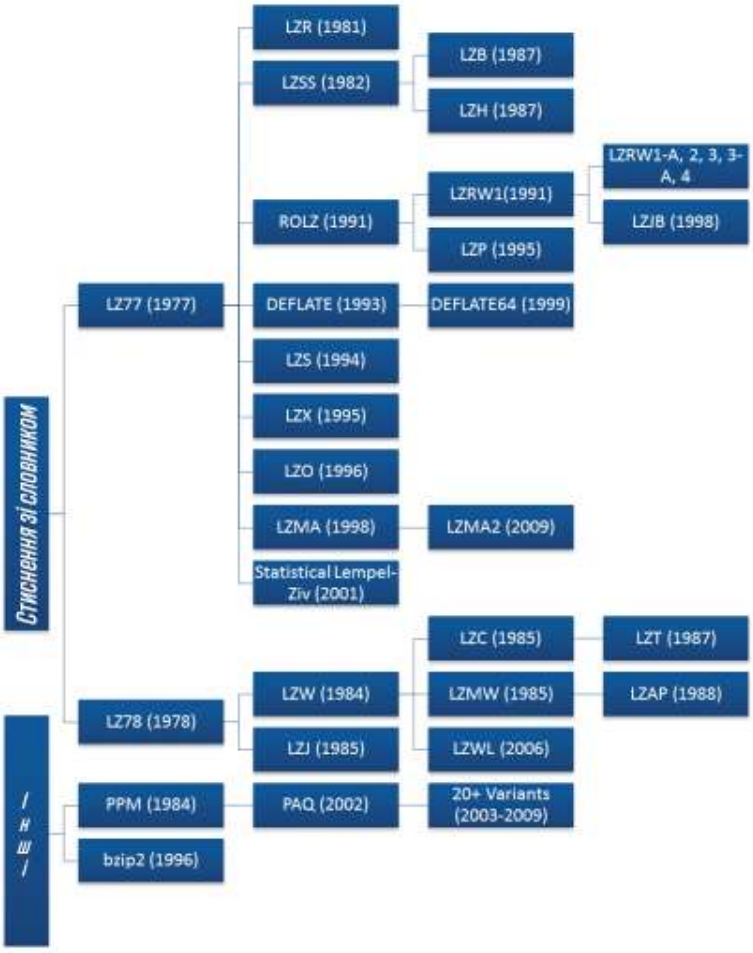


Рисунок 1. Ієрархія алгоритмів стиснення «без втрат»

Ранні імплементації алгоритмів Хаффмана[8] та Шеннона – Фано[11] були розроблені з використанням апаратних та жорстко закодованих (hard coded) кодів. Але коди Хаффмана до цього були статичними. Завдяки появі Інтернету в 1970 – х роках та сервісів, які надаюсь послуги онлайн – сховищ, розробники змогли вдосконалити алгоритм та реалізувати такий застосунок, який генерував коди Хаффмана динамічно застосовуючи вхідні дані як основу. Пізніше, в 1977 році, Авраам Лемпель та Якоб Зів опублікували свою інноваційний, новаторський алгоритм LZ77[6]. Це був перший алгоритм, який для стиснення використовує словник. Конкретніше, LZ77[6] часто використовував динамічний словник, який називається «ковзаючим вікном» (sliding window)[10]. У 1978 році цей же дует опублікував новий алгоритм LZ78[7], який теж використовує словник. На відміну від LZ77[6], цей алгоритм аналізує вхідні дані та генерує статичний словник, а не генерує його динамічно. [16]

1.4 Зростання дефляції

З моменту виходу алгоритмів дуету Лемпеля – Зіва, компанії та інші великі організації використовують алгоритми стиснення даних, оскільки їхні потреби в зберіганні та сховищі продовжують зростати і стиснення даних може повністю задовольнити їх потреби. Однак алгоритми стиснення даних не набули такої широкої популярності, допоки всесвітня мережа Інтернет не набула ваги та значущості в кінці 1980 – х років. Саме в ці роки виникла необхідність в стисненні даних. В ті часи мережева пропускна здатність була коштовною, обмеженою або і тим й іншим одночасно. Саме стиснення допомогло вирішити ці проблеми та закрити недоліки. З появою Всесвітньої павутини, коли люди почали між собою обмінюватися різноманітною інформацією, від текстів до відео, стискаючі алгоритми стали особливо важливими. Адже це допомагало зберігати дорогоцінний трафік та (зважаючи на маленькі швидкості) швидко завантажувати необхідні дані. Постало питання задоволення потреб в стисненні різних даних. Саме для цього було

розроблено кілька нових форматів файлів, що включають стиснення, включаючи ZIP, GIF, PNG.

Том Хендерсон опублікував та випустив перший комерційно успішний формат архівування під назвою ARC через свою компанію в 1985 році, System Enhancement Associates. ARC був особливо популярним серед товариства BBS, так як це була одна з перших програм, яка могла не тільки будувати, але й стискати файли. Також вона мала відкритий вихідний код. Формат ARC використовував модифікований алгоритм LZW. В цей час Філ Кац помітив шалену популярність формату ARC і вирішив поліпшити його, написавши процедури компресії та декомпресії мовою асемблера. Він опублікував PKARC та почав його продавати. Через копірайт, який було доведено, Кац був вимушений відректися від свого дітища через судові процеси. [17]

Так як він не міг більше продавати PKARC, в 1989 році він розробив модифіковану версію алгоритму, який зараз відомий як ZIP формат. Через використання запатентованого алгоритму LZW під капотом, було вирішено перейти на новий IMPLODE алгоритм. Формат було знову оновлено в 1993 році, коли Кац випустив PKZIP 2.0, який реалізовував DEFLATE алгоритм і деякі інші функції, наприклад поділ об’ємів. Ця версія формату ZIP і сьогодні повсюдно зустрічається, так як майже всі файли .zip слідують формату PKZIP 2.0, незважаючи на поважний вік.

GIF або Graphical Interchange Format, було розроблено компанією CompuServe аж в 1987 році щоб дозволити спільне використання растрових зображень без втрати даних (хоча формат обмежений 256 кольорами на кадр) при цьому істотно зменшивши розмір файлу, щоб забезпечити передачу по модемах віддаленого доступу (dialup modems). Однак, як і формат ZIP, GIF також базується на запатентованому алгоритмі LZW. Незважаючи на обтяженість патентами, компанія Unisys не змогла захистити та жорстко контролювати дотримання своїх патентів, щоб зупинити поширення формату.

Навіть зараз, більше 20 років потому, GIF залишається у використанні, особливо через його здатність анімації. [22]

Хоча GIF не вдалося зупинити, CompuServe шукала формат, який не є обтяженим патентами. І в 1994 році представила формат Portable Network Graphics (PNG). Як і ZIP, стандарт PNG використовує алгоритм стискання DEFLATE. Хоча DEFLATE був запатентований Філом Кацом, патент ніколи не застосовувався і, таким чином, PNG та інші формати, засновані на алгоритмі DEFLATE, уникають порушення патентів. Незважаючи на те, що LZW сильно піднявся у рейтингу в перші дні започаткування стискання даних, через суперечливий характер Unisys він більш – менш помер на користь більш швидшого та ефективнішого алгоритму DEFLATE. Зазначений у минулому реченні алгоритм в даний час є найбільш використовуваним у стисненні даних, оскільки він ніби швейцарський армійський ніж серед усіх алгоритмів.

Крім використання у форматах PNG і ZIP, алгоритм DEFLATE також дуже часто використовується в інших обчислювальних системах. Наприклад, формат файлу gzip (.gz) використовує DEFLATE, оскільки він по суті є версією Zip з відкритим кодом. Інші види використання DEFLATE включають HTTP, SSL та інші технології, призначені для ефективного стиснення даних по мережі.

1.5 Типи стиснення

Стиснення даних буває «без втрат»(lossless) та «з втратами»(lossy). Стискання «без втрат» допомагає відновлювати файл у його початковий стан без втрати жодного з бітів даних при розпакуванні. Таке стиснення – це типовий підхід до застосування з виконавчими файлами, а також текстовими та електронними таблицями, де втрата слів або чисел змінить інформацію. Стискання «без втрат»: видаляє біти, виявляючи статистичні надмірності. Саме через цю техніку ніяка інформація насправді не видаляється. Стиснення без втрат зазвичай має менший ступінь стиснення, що дозволяє не втратити

будь-які дані у файлі. Це дуже важливо тоді, коли необхідно підтримувати абсолютну якість, як у випадку з інформацією бази даних або професійними медіа файлами. Такі формати, як FLAC і PNG, пропонують варіанти стиснення без втрат.

Таке стиснення, як lossy, назавжди видаляє ті біти даних, які непотрібні через надлишковість. Вони непотрібні й не впливають на вихідний результат декомпресії. Можливо деякі з них навіть непомітні. Таке корисно для графіки, аудіо, відео та зображень, де видалення деяких бітів даних практично не впливає на представлення вмісту. Стискання «з втратами»: зменшує розмір за рахунок видалення непотрібної інформації та зменшення складності існуючих бітів інформації. Стиснення з втратами може забезпечити набагато вищі коефіцієнти стиснення через можливе погіршення якості файлів. JPEG пропонує варіанти стиснення з втратами, а MP3 і то заснований на стисненні з втратами.

Стиснення графічних зображень може бути «з втратами» або «без втрат». Формати файлів графічних зображень зазвичай призначені для стиснення інформації, оскільки файли, як правило, великі. JPEG – це формат файлу зображення, який підтримує стиснення зображень з втратами. Такі формати, як GIF і PNG, використовують стиснення без втрат. [18] [19]

1.6 Переваги та недоліки стискання даних

Основними перевагами стискання файлів є скорочення розміру файлу на носії, часу передачі даних і пропускної здатності зв'язку, а також економія витрат. Стиснутий файл вимагає меншого обсягу пам'яті, ніж нестиснений файл, і використання стиснення може привести до значного зниження витрат на дискові та/або твердотільні накопичувачі. Стиснутий файл також вимагає менше часу для передачі і споживає менше пропускної здатності мережі, ніж нестиснений файл.

Основним недоліком стиснення даних є вплив на продуктивність, що виникає в результаті використання ресурсів процесора і пам'яті для стиснення даних і виконання декомпресії. Багато компаній розробили і досі розробляють свої системи, щоб спробувати звести до мінімуму вплив інтенсивних обчислень, пов'язаних зі стисненням. Якщо стиснення виконується вбудовано, перед записом даних на диск, система може розвантажити або, іншими словами, вбити процес стиснення, щоб зберегти системні ресурси. Наприклад, IBM використовує окрему карту апаратного прискорення для обробки стиснення з деякими зі своїх корпоративних систем зберігання даних.

Якщо дані стискаються після запису на диск або після обробки, стиснення може виконуватися у фоновому режимі, щоб знизити вплив на продуктивність. Хоча стиснення після такої операції може скоротити час відгуку для кожного вводу - виводу (I/O), воно як і раніше споживає пам'ять, процесорні цикли і може вплинути на загальну кількість операцій вводу-виводу, з якими може впоратися система зберігання. Крім того, оскільки дані спочатку повинні бути записані на диск або флеш-накопичувач в нестисненому вигляді, економія фізичного сховища не така велика, як при вбудованому стисненні. [18]

1.7 Поточні програми архівації

Формат ZIP та інші формати, засновані на алгоритмі DEFLATE, були королями аж до середини 1990-х років, коли почали з'являтися нові та поліпшені формати. У 1993 році Євген Рошаль випустив свій архіватор, відомий як WinRAR, який використовує власний формат RAR. Остання версія RAR використовує комбінацію алгоритмів PPM і LZSS, але про ранні реалізації відомо небагато. RAR став стандартним форматом для обміну файлами через Інтернет, особливо при поширенні піратських носіїв. Реалізація перетворення Барроуза-Віллера з відкритим вихідним кодом під назвою bzip2 була представлена в 1996 році і швидко набрала популярність на платформі UNIX в порівнянні з форматом GZIP на основі DEFLATE.

Ще одна програма стиснення з відкритим вихідним кодом була випущена в 1999 році у форматі 7-Zip або .7z. 7-Zip може стати першим форматом, який кине виклик домінуванню ZIP і RAR через його, як правило, високий ступінь стискання, модульність та відкритості формату. Цей формат не обмежується використанням одного алгоритму стиснення, але замість цього може вибирати між алгоритмами bzip2, LZMA, LZMA2 і PPMd серед інших. Нарешті, на передовій архівного програмного забезпечення знаходяться формати PAQ. Перший формат PAQ був випущений Метом Махоні в 2002 році під назвою PAQ1. Цей алгоритм істотно покращує алгоритм PPM використовуючи метод, відомий як змішування контексту, який об'єднує дві або більше статистичних моделей для отримання кращого передбачення наступного символу, ніж будь-яка з інших моделей.

1.8 Майбутні розробки

Майбутнє ніколи не буває певним. Але, виходячи з поточних тенденцій, можна зробити деякі прогнози щодо того, що може статися в майбутньому при стисненні даних. Алгоритми змішування контексту, такі як PAQ і його варіанти, почали ставати дедалі популярнішими. І вони, як правило, досягають найвищих коефіцієнтів стиснення, але зазвичай працюють повільно. З експоненціальним збільшенням швидкості апаратного забезпечення відповідно до закону Мура, алгоритми змішування контексту, ймовірно, будуть процвітати, оскільки штрафи за швидкість нівелюються за рахунок більш швидкого обладнання і на виході отримуємо високий ступінь стиснення. Алгоритм, який PAQ прагнув поліпшити, називається прогнозом шляхом часткового зіставлення (PPM), також може обзавестися модифікаціями та нащадками. Нарешті, ланцюговий алгоритм Лемпеля – Зіва – Маркова (LZMA) незмінно демонструє відмінний компроміс між швидкістю компресії та високим ступенем стиснення. Ймовірно, в майбутньому цей алгоритм породить більше варіантів. LZMA може навіть стати "переможцем" серед всіх алгоритмів у подальшого розвитку, оскільки він вже був прийнятий на

озброєння багатьма конкуруючими форматами стиснення з тих пір, як його було введено з форматом 7-Zip. Ще одним потенційним розвитком є використання стиснення за допомогою перерахування під рядків (CSE), який є перспективним методом стиснення, який ще не бачив багатьох програмних реалізацій. У своїй наївній формі він працює аналогічно bzip2 і PPM, і дослідники працюють над підвищенням його ефективності. [23]

РОЗДІЛ 2. ОГЛЯД МЕТОДІВ СТИСНЕННЯ

2.1 Вступ

Для стиснення даних використовується безліч різних методів. Більшість методів стиснення не можуть стояти самі по собі, але повинні бути об'єднані разом, щоб сформувати алгоритм стиснення. Навіть ті, які можуть стояти окремо, часто більш ефективні, коли вони об'єднані з іншими методами стиснення. Більшість з цих методів підпадають під категорію ентропійного кодування. Але є й інші, такі як кодування довжини пробігу і перетворення Барроуза-Віллера, які також широко використовуються. Також в даному розділі розглянемо такі методи, як ентропійне кодування, алгоритм Шеннона – Фано, алгоритм Хаффмана та арифметичне кодування.

2.2 Кодування довжин серій (RLE)

Кодування довжин серій або Run-Length Encoding – це дуже простий метод стиснення, який замінює прогони двох або більше однакових символів числом, що представляє довжину прогону, за яким слід вихідний символ; окремі символи кодуються як прогони одиниці. RLE корисний для сильно надлишкових даних, індексованих зображень з великою кількістю пікселів одного кольору в рядку або в поєднанні з іншими методами стиснення, такими як перетворення Барроуза-Віллера.

Розглянемо приклад. Нехай у нас є чорно – біле зображення, яке складається з чорних або білих пікселів. Дано такий масив пікселів:

“WWWWBBBBBBBBBBBBBBWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWB”

Застосуємо до цього алгоритму RLE та продемонструємо вихідний масив та деяку статистику. Треба пам’ятати, що обсяг вихідного масиву може зрости якщо вхідний масив містить багато неповторюваних символів:

Вихідний масив: “4W14B34W1B”

Вхідна кількість символів: 53 символи

Вихідна кількість символів: 10 символів

Співвідношення: 81,1%

Проблему зростання кількості знаків досить легко вирішити. В цьому випадку алфавіт, який застосовувався для запису довжини серії розділяється навпіл. Наприклад розділити цілі числа на додатні, від'ємні та використати додатні для запису кількості повторюваних символів, а від'ємні – неоднакових.

2.3 Перетворення Барроуза – Віллера (BWT)

Перетворення Барроуза-Віллера – це метод стиснення, винайдений в 1994 році Девідом Віллером та Майклом Барроузом, який був його студентом. Цей алгоритм спрямований на оборотне перетворення блоку вхідних даних таким чином, щоб кількість запусків ідентичних символів було максимальним. Сам BWT не виконує жодних операцій стиснення, він просто перетворює вхідні дані таким чином, щоб їх можна було більш ефективно кодувати за допомогою кодування довжини виконання або іншого методу вторинного стиснення.

Алгоритм для BWT простий:

- Створити масив рядків.
- Згенерувати всі можливі повороти вхідного рядка, зберігаючи кожен з них в масиві.
- Відсортувати масив за алфавітом.
- Повернути останній стовпчик масиву. [24]

BWT зазвичай найкраще працює на довгих вхідних даних з великою кількістю однакових символів, які чергуються. Наведемо приклад алгоритму, що виконується на ідеальному вході. Зверніть увагу, що & - це кінець символу файлу:

Вхідні дані	Обертання	Альфа-Сортовані Обертання	Вихідні дані
НАНАНА&	НАНАНА&	АНАНА&**Н**	**ННН&ААА**
	&НАНАНА	АНА&НА**Н**	
	А&НАНАН	А&НАНА**Н**	
	НА&НАНА	НАНАНА**&**	
	АНА&НАН	НАНА&Н**А**	
	НАНА&НА	НА&НАН**А**	
	АНАНА&Н	&НАНАН**А**	

Таблиця 1. Приклад перетворення BWT

Через чергування однакових символів, виконання BWT на цьому вході генерує оптимальний результат, який інший алгоритм може додатково стиснути, наприклад RLE, який дасть "ЗН&ЗА". Хоча цей приклад дав оптимальний результат, він не дає оптимальних результатів для більшості реальних даних.

Ряд методів оптимізації можуть зробити цей алгоритм більш ефективним без зміни вихідних даних. Немає необхідності представляти таблицю ні в кодері, ні в декодері. У кодері кожен рядок таблиці може бути представлений одним покажчиком на рядки, а сортування виконується з використанням індексів. У декодері також немає необхідності зберігати таблицю, і насправді сортування взагалі не потрібно. У часі, пропорційному розміру алфавіту і довжині рядка, декодований рядок може генеруватися по одному символу за раз справа наліво. "Символ" в алгоритмі може бути байтом, бітом або будь-яким іншим зручним розміром. Можна також зробити зауваження, що математично закодований рядок може бути обчислений як проста модифікація масиву суфіксів, а масиви суфіксів можуть бути обчислені з лінійним часом і пам'яттю. BWT може бути визначений щодо масиву суфіксів SA тексту T як (індексування на основі 1):

$$BWT[i] = \begin{cases} T[SA[i]-1], \text{якщо } SA[i] > 0 \\ \$, \text{в іншому випадку} \end{cases}$$

Немає необхідності мати фактичний символ "EOF". Замість цього можна використовувати покажчик, який запам'ятовує, де в рядку був би "EOF", якби він існував. При такому підході вихідні дані BWT повинні включати як перетворений рядок, так і кінцеве значення покажчика. Зворотне перетворення потім стискає його назад до вихідного розміру: йому дається рядок, покажчик і повертається тільки рядок.

2.3.1 Динамічний алгоритм BWT

При редагуванні тексту його перетворення Барроуза-Віллера зміниться. Мікаель Салсон у своїй статті пропонує алгоритм, який виводить перетворення Барроуза–Віллера відредагованого тексту з вихідного тексту, виконуючи обмежену кількість локальних переупорядкувань у вихідному перетворенні Барроуза–Віллера. Це може бути швидше, ніж безпосередньо побудова перетворення трансформації Барроуза–Віллера відредагованого тексту. [26]

2.4 Ентропійне кодування (Entropy coding)

Ентропія при стисненні даних означає найменшу кількість бітів, необхідна в середньому для представлення символу або літералу. Базовий кодер ентропії поєднує в собі статистичну модель і кодер. Вхідний файл аналізується і використовується для створення статистичної моделі, яка складається з ймовірностей появи даного символу. Потім кодер буде використовувати статистичну модель, щоб визначити, які біти або байт-коди призначати кожному символу таким чином, щоб найбільш поширені символи мали найкоротші коди, а найменш поширені символи мали найдовші коди. [24]

Двома найпоширенішими методами ентропійного кодування є кодування Хаффмана та арифметичне кодування.[25] Якщо приблизні характеристики ентропії потоку даних відомі заздалегідь (особливо для стиснення сигналу),

може бути корисний простіший статичний код. Ці статичні коди включають універсальні коди (такі як гамма-кодування Еліаса або кодування Фібоначчі) і коди Голомба (такі як унарне кодування або кодування рису).

З 2014 року Компресори даних почали використовувати сімейство методів ентропійного кодування асиметричних числових систем, що дозволяє поєднувати ступінь стиснення арифметичного кодування з витратами на обробку, аналогічними кодування Хаффмана.

2.5 Алгоритм Шеннона – Фано (Shannon – Fano coding)

Це один з найбільш ранніх методів стиснення, винайдений в 1949 році Клодом Шенноном і Робертом Фано. Цей метод включає в себе створення двійкового дерева для представлення ймовірностей появи кожного символу. Символи впорядковані таким чином, що найбільш часто зустрічаються символи з'являються у верхній частині дерева, а найменш ймовірні – в нижній. Код для даного символу виходить шляхом пошуку його в дереві Шеннона – Фано і додавання до коду значення 0 або 1 для кожної взятої лівої або правої гілки відповідно. Наприклад, якщо" A "- це дві гілки зліва і одна справа, її код буде "0012". Кодування Шеннона – Фано не завжди дає оптимальні коди через те, що воно будує двійкове дерево знизу вгору. З цієї причини замість цього використовується кодування Хаффмана, оскільки воно генерує оптимальний код для будь-якого заданого введення.

Алгоритм генерації кодів Шеннона-Фано досить простий

- Проаналізувати вхідні дані, підрахувавши появу кожного символу.
- Визначити ймовірність кожного символу, використовуючи кількість символів.
- Відсортувати символи за ймовірністю, спочатку вибравши найбільш ймовірні.
- Створити кінцеві вузли для кожного символу.

- Розділити список на дві частини, зберігаючи ймовірність лівої гілки приблизно рівною ймовірності правої гілки.
- Додати 0 і 1 до кодів лівого і правого вузлів відповідно.
- Рекурсивно застосувати кроки 5 і 6 до лівого і правого під-дерева, поки кожен вузол не стане листом в дереві. [27]

2.6 Алгоритм Хаффмана (Huffman coding)

Кодування Хаффмана – це ще один варіант ентропійного кодування, який працює дуже схоже на кодування Шеннона – Фано, але двійкове дерево будується зверху вниз для отримання оптимального результату.

Алгоритм генерації кодів Хаффмана схожий своїми першими кроками з Шенноном – Фано:

1. Проаналізувати вхідні дані, підрахувавши появу кожного символу.
2. Визначити ймовірність кожного символу, використовуючи кількість символів.
3. Відсортувати символи за ймовірністю, спочатку вибравши найбільш ймовірні.
4. Створити кінцеві вузли для кожного символу, включаючи P, і додати їх в чергу.
5. Допоки (вузлів в черзі > 1)
   - Видалити з черги два вузли з найменшою ймовірністю.
   - Додати 0 і 1 до кодів лівого і правого вузлів відповідно.
   - Створити новий вузол зі значенням, рівним сумі ймовірностей вузлів.
   - Призначити перший вузол до лівої гілки, а другий вузол – до правої гілки.
   - Додати вузол в чергу
6. Останній вузол, що залишився в черзі, є коренем дерева Хаффмана. [28]

2.7 Арифметичне кодування (Arithmetic coding)

Цей метод було розроблено в 1979 році в IBM, яка вивчала методи стиснення даних для використання в своїх мейнфреймах. Арифметичне кодування, можливо, є найбільш оптимальним методом ентропійного кодування, якщо метою є найкраща ступінь стиснення, оскільки воно зазвичай досягає кращих результатів, ніж кодування Хаффмана. Однак це досить складно в порівнянні з іншими методами кодування. Замість того, щоб розбивати ймовірності символів на дерево, арифметичне кодування перетворює вхідні дані в одне раціональне число від 0 до 1, змінюючи базу і присвоюючи кожному унікальному символу одне значення від 0 до бази. Потім він далі перетворюється в двійкове число з фіксованою точкою, яке є закодованим результатом. Значення може бути декодовано в вихідний висновок, змінивши базу з двійковою назад на вихідну базу і замінивши значення символами, яким вони відповідають.

Загальний алгоритм обчислення арифметичного коду:

1. Обчислити кількість унікальних символів у вхідних даних. Це число являє собою базу b (наприклад, база 2 є двійковим) арифметичного коду.
2. Призначити значення від 0 до b кожному унікальному символу в порядку їх появи.
3. Використовуючи значення з кроку 2, замінити символи у вхідних даних їх кодами
4. Перетворити результат кроку 3 з бази b в досить довге двійкове число з фіксованою точкою, щоб зберегти точність.
5. Записати довжину вхідного рядка де – небудь в результаті, так як це необхідно для декодування. [29]

Наведемо приклад арифметичного кодування:

Дано масив " ABCDAABD”

1. Знайдено 4 унікальних символи у вхідних даних, отже, база = 4. Довжина = 8
2. Присвоєні значення символам: A=0, B=1, C=2, D=3
3. Замінено вхідні дані кодами: "0.012300134", де початковий 0 не є символом.
4. Перетворено " 0.012311234 "з бази 4 в базу 2:" 0.011011000001112”
5. Результат знайдено. Звертаємо увагу, що в результаті довжина вхідного сигналу дорівнює 8.

Припускаючи 8-бітові символи, вхідний сигнал має довжину 64 біта, в той час як його арифметичне кодування становить всього 15 біт, що призводить до відмінної ступеня стиснення у 24%. Цей приклад демонструє, як арифметичне кодування добре стискається при обмеженому наборі символів.

РОЗДІЛ 3. ОГЛЯД АЛГОРИТМІВ СТИСНЕННЯ

3.1 Алгоритми «Ковзаючого вікна» (Sliding window)

Алгоритм «ковзаючого вікна» – це один із способів, за допомогою якого програмісти можуть спрощувати свій код. Цей алгоритм робить саме те, що і означає; вікно формується над деякою частиною даних і це вікно може ковзати за даними, щоб захопити різні їх частини.

3.1.1 LZ77 та його модифікації

Опублікований в 1977 році, LZ77 – це алгоритм, з якого все почалося. У ньому вперше була введена концепція «ковзаючого вікна», яка призвела до значного поліпшення ступеня стиснення в порівнянні з більш примітивними алгоритмами. LZ77 підтримує словник, використовуючи трійки, що представляють зміщення, довжину виконання і символ відхилення. Зсув – це відстань від початку файлу, з якого починається дана фраза, а довжина виконання – це кількість символів, що пройшли через зміщення, які є частиною фрази. Символ відхилення (deviating character) – це просто вказівник на те, що була знайдена нова фраза і ця фраза дорівнює фразі від зміщення до фрази, «зміщення + довжина» плюс символ відхилення. Використовуваний словник динамічно змінюється в залежності від ковзного вікна в міру аналізу файлу. Наприклад, ковзне вікно може мати розмір 64 МБ, що означає, що словник буде містити записи для останніх 64 МБ вхідних даних.

Наведемо приклад алгоритму. Маємо на вході масив «abbadabba». Вихідний масив буле виглядати як «abb(0,1, ’d’)(0,3,’a’)» як наведено в таблиці 2 нижче:

Позиція	Символ	Вихід
0	a	a
1	b	b
2	b	b
3	a	(0,1,’d’)
4	d	
5	a	(0,3,’a’)
6	b	
7	b	
8	a	

Таблиця 2. Приклад роботи алгоритму LZ77

Хоча ця підстановка трохи більша ніж вхідні дані, вона зазвичай дає значно менший результат при довших вхідних даних. [6]

LZR – це модифікація LZ77, винайдена Майклом Роде в 1981 році. Алгоритм націлений на те, щоб бути лінійною тимчасовою альтернативою LZ77. Однак закодовані покажчики можуть вказувати на будь-який зсув у файлі. Це означає, що LZR споживає значний обсяг пам'яті. У поєднанні з його поганим ступенем стиснення (LZ77 часто перевершує) це поганий варіант. [30] [31]

Алгоритм LZX був розроблений в 1995 році Джонатаном Форбсом і Томі Поутаненом для комп'ютера Amiga. X в назві LZX не має особливого значення. Джонатан Форбс продав алгоритм Microsoft в 1996 році і почав працювати на них, де він був додатково вдосконалений для використання у

форматі Microsoft Cabinet (.CAB). Цей алгоритм також використовується Microsoft для стиснення стиснутих файлів довідки HTML (CHM), файлів формату зображень Windows (WIM) і аватару Xbox Live. [35]

Алгоритм Лемпеля–Зіва–Стака (LZS) був розроблений компанією Stac Electronics в 1994 році для використання в програмному забезпеченні для стиснення дисків. Це модифікація LZ77, яка розрізняє літеральні символи в парах виводу і довжини зміщення на додаток до видалення наступного зустрінутого символу. Алгоритм LZS функціонально найбільш схожий на алгоритм LZSS [37]

Статистичний алгоритм Лемпеля–Зіва був концепцією, створеною доктором Семом Квонгом і Ю Фан Хо в 2001 році. Основний принцип, полягає в тому, що статистичний аналіз даних може бути об'єднаний з алгоритмом варіанту LZ77 для подальшої оптимізації того, які коди зберігаються в словнику. [40]

3.1.2 LZSS та його модифікації

Алгоритм LZSS, або алгоритм Лемпеля – Зіва – Сторера – Шиманскі був вперше опублікований в 1982 році Джеймсом Сторером і Томасом Шиманскі. LZSS покращує LZ77 тим, що він може визначити чи зменшить заміна символу розмір файлу чи ні. Якщо зменшення розміру не буде досягнуто, вхідні дані будуть залишені в якості літералу в вихідних даних. В іншому випадку розділ введення замінюється парою (зміщення, довжина), де зміщення – це кількість байтів від початку введення, а довжина – це кількість символів для читання з цієї позиції.[32] Ще одне поліпшення в порівнянні з LZ77 пов'язане з усуненням "наступного символу" і використанням тільки пари зміщення довжини.

Наведемо простий приклад. Маємо на вході речення «these theses» яке після обробки алгоритмом вертає «these(0,6)s», що економить всього один байт, але досягає значної економії на великих вхідних даних.

Індекс	0	1	2	3	4	5	6	7	8	9	10	11	12
Символ		t	h	e	s	e		t	h	e	s	e	s
Заміна		t	h	e	s	e	(	0	,	6	)	s	

Таблиця 3. Приклад алгоритму LZSS

LZSS як і раніше використовується в багатьох популярних архівних форматах, найвідоміший з яких WinRAR. Він також іноді використовується для стиснення мережевих даних.

Алгоритм LZH був розроблений в 1987 році і розшифровується як Лемпеля – Зіва – Хаффмана. Це варіант LZSS, який використовує кодування Хаффмана для стиснення вказівники, що призводить до дещо кращого стиснення. Однак поліпшення, отримані за допомогою кодування Хаффмана незначні і стиснення не вартує зниженням продуктивності при використанні кодування Хаффмана. [31]

Алгоритм LZB також був розроблений в 1987 році Тімоті Беллом як варіант LZSS. Як і LZH, LZB також прагне зменшити розмір стисненого файлу, більш ефективно кодуючи вказівники LZSS. Це робиться шляхом поступового збільшення розміру вказівники у міру збільшення ковзного вікна. Він може досягти більш високого ступеня стиснення ніж LZSS і LZH, але він все ще досить повільний в порівнянні з LZSS через додатковий крок кодування для вказівників. [31]

LZO був розроблений Маркусом Оберхумером в 1996 році. Метою розробки було швидке стиснення і декомпресія. Алгоритм дозволяє регулювати рівні стиснення і вимагає тільки 64 Кілобайт додаткової пам'яті для найвищого рівня стиснення, в той час як для декомпресії потрібні тільки вхідні і вихідні буфери. LZO функціонує дуже схоже на алгоритм LZSS, але оптимізований для швидкості, а не для ступеня стиснення. [34]

3.1.1 ROLZ та його модифікації

ROLZ розшифровується як «Reduced Offset Lempel – Ziv» або алгоритм Лемпеля – Зіва зі зменшеним зсувом. Його мета полягає в тому, щоб поліпшити стиснення LZ77 обмеживши довжину зміщення, щоб зменшити обсяг даних, необхідних для кодування пари зміщення – довжина. Цей похідний від LZ77 алгоритм був вперше помічений 1991 року в алгоритмі Росса Вільямса LZRW4. Інші реалізації включають BALZ, QUAD і RZM. Високо оптимізований ROLZ може досягти майже тих же коефіцієнтів стиснення, що і LZMA; однак ROLZ страждає від нестачі популярності.

LZP розшифровується як «Lempel–Ziv+» або «Лемпеля – Зіва+». Це окремий випадок алгоритму ROLZ, де зміщення зменшено до 1. Існує кілька варіантів використання різних методів для досягнення або більш швидкої роботи, або кращого ступеня стиснення. LZW4 реалізує арифметичний кодер для досягнення найкращого ступеня стиснення за рахунок швидкості. [33]

3.1.1 LZRW та його модифікації

Рон Вільямс створив алгоритм LZRW1 у 1991 році, вперше представивши концепцію стиснення Лемпеля – Зіва зі зменшеним зміщенням. LZRW1 може досягати високих коефіцієнтів стиснення, залишаючись при цьому дуже швидким і ефективним. Рон Вільямс також створив кілька варіантів, що поліпшують LZRW1, таких як LZRW1-A, 2, 3, 3-A і 4. [34]

Джефф Бонвік створив свій алгоритм Лемпеля–Зіва–Джеффа Бонвіка (LZJB) в 1998 році для використання у файловій системі Solaris Z (ZFS). Розглядається як варіант алгоритму LZW, зокрема варіант LZRW1 який націлений на максимальну швидкість стиснення. Оскільки він використовується у файловій системі, швидкість особливо важлива для забезпечення того, щоб операції на диску не були обмежені алгоритмом стиснення.

3.1.1 LZMA та його модифікації

Алгоритм ланцюга Лемпеля–Зіва–Маркова був вперше опублікований в 1998 році з випуском архіватора 7-Zip для використання у форматі .7z. У більшості випадків він забезпечує краще стиснення, ніж bzip2, DEFLATE та інші алгоритми. LZMA використовує ланцюжок методів стиснення для досягнення своєї продуктивності. По–перше, для аналізу даних використовується модифікований алгоритм LZ77, який працює на побітовому рівні, а не на традиційному побайтовому рівні. Потім вихідні дані алгоритму LZ77 піддаються арифметичному кодуванню. Залежно від конкретної реалізації LZMA, може бути застосовано більше методів. В результаті значно поліпшується коефіцієнт стиснення в порівнянні з більшістю інших варіантів LZ, головним чином завдяки побітовому методу стиснення, а не побайтовому.[38]

LZMA2–це поступове поліпшення оригінального алгоритму LZMA, вперше представленого в 2009 році.[39] в оновленні програмного забезпечення для архівування 7-Zip. LZMA2 покращує можливості багатопоточності і, отже, продуктивність алгоритму LZMA. А також кращу обробку нестисливих даних, що призводить до дещо кращого стиснення.

3.1.2 DEFLATE та DEFLATE64

DEFLATE був винайдений Філом Кацом в 1993 році і сьогодні є основою для більшості завдань стиснення. Він просто об'єднує препроцесор LZ77 або LZSS з кодуванням Хаффмана на бекенді для досягнення помірно стислих результатів за короткий час.

DEFLATE64 – це власне розширення алгоритму DEFLATE, яке збільшує розмір словника до 64 Кілобайт (звідси і назва). Це дозволяє збільшити відстань в ковзному вікні. Він підвищує як продуктивність, так і ступінь стиснення в порівнянні з DEFLATE. Однак пропрієтарний характер DEFLATE64 та його скромні поліпшення в порівнянні з DEFLATE призвели

до обмеженого впровадження формату. Замість цього зазвичай використовуються алгоритми з відкритим кодом, такі як LZMA.

3.2 Алгоритми з використанням словника

3.2.1 LZ78, LZW та їх модифікації

LZ78 було створено дуетом Лемпелем і Зівом в 1978 році, звідси і абревіатура. Замість того, щоб використовувати ковзне вікно для створення словника, існує дві стратегії алгоритмізації:

1. Вхідні дані попередньо обробляються для створення словника з нескінченним обсягом вхідних даних.
2. Словник формується в міру аналізу файлу.

LZ78 використовує саме останню тактику. Розмір словника зазвичай обмежений кількома мегабайтами або всіма кодами до певної кількості байтів, наприклад 8. Це робиться для зниження вимог до пам'яті. Те, як алгоритм обробляє повний словник – це те, що відрізняє більшість алгоритмів типу LZ78 один від одного.[7]

При зчитуванні файлу, алгоритм LZ78 додає кожен знову зустрітий символ або рядок символів до словника. Для кожного символу у вхідних даних генерується словниковий запис у формі (індекс словника, невідомий символ); якщо символ вже є в словнику, то в словнику буде виконаний пошук під рядків поточного символу і наступних за ним символів. Індекс найдовшого під рядка використовується для індексу словника. Дані, на які вказує індекс словника, додаються до останнього символу невідомого під рядка. Якщо поточний символ невідомий, то індекс словника встановлюється рівним 0, щоб вказати, що це запис, який складається з одного символу. Записи зі структури даних формують структуру зв’язного листа. [31]

Наведемо приклад виконання алгоритму: на вході маємо рядок «abbadabbaabaad» який після завершення алгоритму буде рівнятися «(0,a)(0,b)(2,a)(0,d)(1,b)(3,a)(6,d)»

Вхід		a	b	ba	d	ab	baa	baad
Індекс	0	1	2	3	4	5	6	7
Вихід	NULL	(0,a)	(0,b)	(2,a)	(0,d)	(1,b)	(3,a)	(6,d)

Таблиця 4. Приклад обрахунку алгоритму LZ78

LZW – це алгоритм Лемпеля–Зіва–Велша, створений в 1984 році Террі Велшем. Це найбільш часто використовуваний похідний алгоритм сімейства LZ78, незважаючи на те, що він сильно обтяжений патентами. LZW покращує LZ78 аналогічно LZSS; він видаляє надлишкові символи у виводі і робить висновок повністю з вказівників. Він також включає кожен символ в словнику перед початком стиснення і використовує інші прийоми для поліпшення стиснення, такі як кодування останнього символу кожної нової фрази в якості першого символу наступної фрази. LZW зазвичай зустрічається у форматі обміну графікою, а також у ранніх специфікаціях формату ZIP та інших спеціалізованих додатках. LZW дуже швидкий, але забезпечує погане стиснення в порівнянні з більшістю нових алгоритмів і деякі алгоритми працюють швидше і забезпечують краще стиснення.

LZWL – це модифікація алгоритму LZW, створеного в 2006 році, який працює зі складами, а не з окремими символами. LZWL призначений для кращої роботи з певними наборами даних та з складами, які часто зустрічаються (такими як XML-дані). Цей тип алгоритму зазвичай використовується з препроцесором, який розкладає вхідні дані на склади. [43]

Матті Якобсон опублікував алгоритм LZJ в 1985 році[41], і це один з небагатьох алгоритмів LZ78, який відхиляється від LZW. Алгоритм працює, зберігаючи кожен унікальний рядок у вже оброблених вхідних даних до довільної максимальної довжини в словнику і присвоюючи кожному коду коди. Коли словник заповнений, всі записи, які відбулися тільки один раз, видаляються. [31]

Стиснення LZC або Lempel – Ziv – Compress це невелика модифікація алгоритму LZW, використовуваного в утиліті стиснення UNIX. Основна відмінність між LZC і LZW полягає в тому, що LZC контролює ступінь стиснення вихідного сигналу. Як тільки співвідношення перетинає певний поріг, словник відкидається і перебудовується. [31]

3.2.2 LZT та його модифікації

Алгоритм Лемпеля – Зіва – Тищера (Lempel – Ziv–Tischer) – це модифікація LZC, яка коли словник заповнений, видаляє найменш нещодавно використану фразу і замінює її новим записом. Є й інші додаткові поліпшення, але ні LC, ні LZT сьогодні не використовуються. [31]

Алгоритм LZMW, винайдений в 1984 році Віктором Міллером і Марком Вегманом, дуже схожий на LZT в тому, що він використовує найменш недавно використану стратегію заміни фраз. Однак замість того, щоб об'єднувати аналогічні записи в словнику, LZMW об'єднує останні дві закодовані фрази і зберігає результат у вигляді нового запису. В результаті розмір словника може збільшуватися досить швидко, і LRUs доводиться відкидати частіше. LZMW зазвичай забезпечує краще стиснення, ніж LZT, однак це ще один алгоритм, який не має великого сучасного застосування. [31]

LZAP був створений в 1988 році Джеймсом Сторером в якості модифікації алгоритму LZMW. AP розшифровується як "всі префікси" в тому, що замість того, щоб зберігати одну фразу в словнику кожну ітерацію, словник зберігає кожну перестановку. Наприклад, якщо остання фраза була «last», а поточна фраза – «next», словник буде зберігати «lastn», «lastne», «lastnex» та «lastnext». [42]

3.3 Алгоритми без використання словників

3.3.1 Передбачення щодо часткового збігу (PPM)

Передбачення шляхом часткового зіставлення – це метод статистичного моделювання, який використовує набір попередніх символів у вхідних даних, щоб передбачити, яким буде наступний символ для зменшення ентропії вихідних даних. Цей підхід відрізняється від словникових, оскільки PPM робить прогнози про те, яким буде наступний символ, а не намагається знайти наступні символи в словнику, щоб закодувати їх. PPM зазвичай поєднується з кодером під капотом, таким як арифметичне кодування або адаптивне кодування Хаффмана. [44] PPM або варіант, відомий як PPMd, реалізовані в багатьох форматах архівів, включаючи 7-Zip і RAR.

3.3.2 Bzip2

Bzip2 – це реалізація перетворення Барроуза–Віллера з відкритим вихідним кодом. Його принципи роботи прості, але вони забезпечують дуже хороший компроміс між швидкістю і ступенем стиснення, що робить формат bzip2 дуже популярним в середовищі UNIX. Спочатку до даних застосовується RLE. Потім застосовується перетворення Барроуза–Віллера. Після цього застосовується перетворення «переміщення вперед» з метою створення великої кількості ідентичних символів, що утворюють прогони для використання в ще одному кодері RLE. Нарешті, результат кодується алгоритмом Хаффмана й завертається у заголовок. [45]

3.1.1 PAQ

PAQ був створений Меттом Махоні в 2002 році як поліпшення старих алгоритмів сімейства PPM(d). Це робиться за допомогою революційної техніки названої змішуванням контексту. Змішування контексту – це коли кілька статистичних моделей (PPM – один з прикладів) розумно об'єднуються, щоб зробити кращі прогнози наступного символу, ніж будь–яка з моделей сама по собі. PAQ є одним з найбільш перспективних алгоритмів через його

надзвичайно високий ступінь стиснення і дуже активної розробки. З моменту його створення було створено більше 20 варіантів, причому деякі варіанти досягли рекордних коефіцієнтів стиснення. Найбільшим недоліком PAQ є його низька швидкість через використання декількох статистичних моделей для отримання найкращого ступеня стиснення. Однак, оскільки апаратне забезпечення постійно стає швидше, воно може стати стандартом майбутнього. [46] PAQ поступово впроваджується, і варіант під назвою PAQ8O, який забезпечує 64-розрядну підтримку і значне поліпшення швидкості, можна знайти в програмі PeaZip для Windows. Інші формати PAQ в основному доступні тільки в командному рядку.

РОЗДІЛ 4. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ І АЛГОРИТМІВ СТИСНЕННЯ

4.1 Перетворення Барроуза – Віллера

Весь код був написаний в асинхронній манері, що дозволяє ефективно використовувати ресурси машини й зменшити час, необхідний для виконання того чи іншого методу. Для цього було використано один із можливих рішень мови C#, а саме Task Parallel Library[47], що дозволяє системі вирішувати кількість потоків та безпечно видаляти необхідні.

```
13      public static async Task<byte[]> Transform(byte[] input)
14      {
15          var task = Task.Factory.StartNew(() =>
16          {
17              Console.WriteLine("Performing BWT");
18              var output = new byte[input.Length + 4];
19              var newInput = new short[input.Length + 1];
20
21              for (var i = 0; i < input.Length; i++)
22                  newInput[i] = (short) (input[i] + 1);
23
24              newInput[input.Length] = 0;
25              var suffixArray = SuffixArray.Construct(newInput);
26              var end = 0;
27              var outputInd = 0;
28              for (var i = 0; i < suffixArray.Length; i++)
29              {
30                  if (suffixArray[i] == 0)
31                  {
32                      end = i;
33                      continue;
34                  }
35
36                  output[outputInd] = (byte) (newInput[suffixArray[i] - 1] - 1);
37                  outputInd++;
38              }
39
40              var endByte = IntToByteArr(end);
41              endByte.CopyTo(output, input.Length);
42              Console.WriteLine("BWT Ended");
43              return output;
44          });
45
46          return await task;
47      }
```

Рисунок 2. Асинхронний алгоритм трансформації Барроуза–Віллера

Для методу трансформації вхідного масиву байтів методом Барроуза – Віллера було використано суфіксний масив замість суфіксного дерева. Для алгоритму нам треба впорядкований масив всіх суфіксів рядка, що забезпечується таким масивом. текстового файлу.h

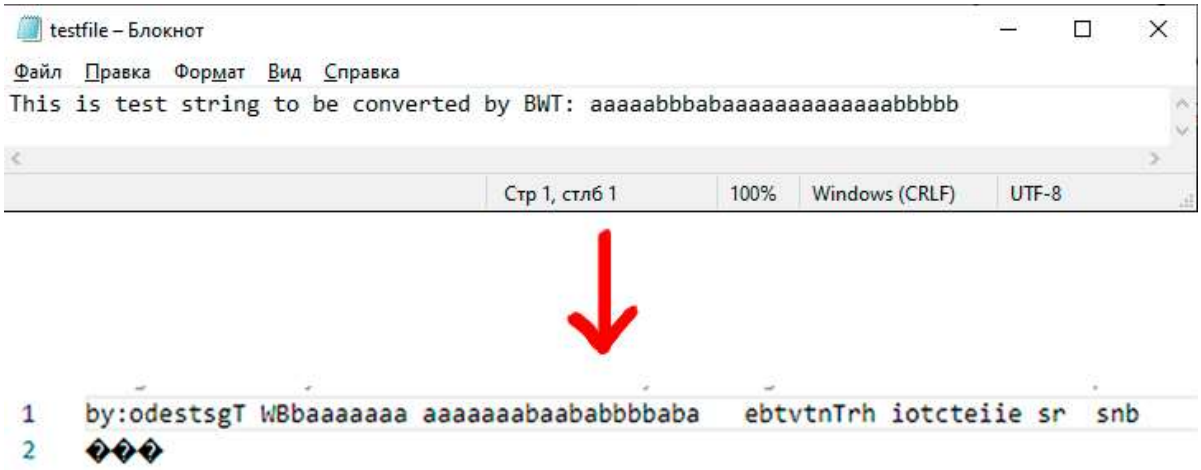


Рисунок 3. Результат перетворення Барроуза – Віллера

Також була запрограмована реалізація сортування масиву суфіксів на базі масиву рангів. Часова складність цього алгоритму становить $O(N)$. Розглянемо роботу алгоритму на базі невеликого прикладу (рис. 3). Як видно, алгоритм дійсно відтворює те, що очікується: під рядки, які повторюються утворюють на виході нові послідовності, в який однакові символи йдуть поспіль.

4.2 Стандартний метод LZW

Для реалізації методу LZW було використано напрацювання Марка Нельсона, які допомогли ефективніше використовувати ресурси та значно зменшити кількість часу та пам’яті, необхідної для обрахувань.[48]

Спочатку було визначено константні дані, які використовуються в ході зчитування та стискання даних а саме:

- Максимальна кількість дозволених для зчитування бітів (Max. Bits = 14).
- Кількість хеш бітів для знаходження коректного індексу в алгоритмі хешування (Max.Bits – 8).

- Дозволене максимальне значення, яке спирається на максимальну кількість дозволених бітів ((1 << Max. Bits) – 1).
- Максимальна кількість можливих кодів (Max. Value – 1)
- Розмір таблиці (Table Size = 18041).

Також відразу було ініціалізовано масиви кодів, префіксів та символів розмір який сягає розміру таблиці. Спочатку алгоритм вичищає усю таблицю, ставлячи у кожну комірку значення -1. Потім використовуючи побайтове зчитування файлу, допоки не дійдемо до кінця. Потім використовуючи функцію хешування знаходимо індекс префіксу + символу. Якщо не знайшли, то повертаємо -1 сигналізуючи про те, що місце вільне. Якщо ми маємо щось під цим індексом, то перезначаємо рядок елементом з цим індексом. Інакше вставляємо новий запис в таблицю та інкрементуємо індекс наступного коду для використання. Після цього виводимо дані як рядок.

Розглянемо приклад виконання у веб – застосунку:

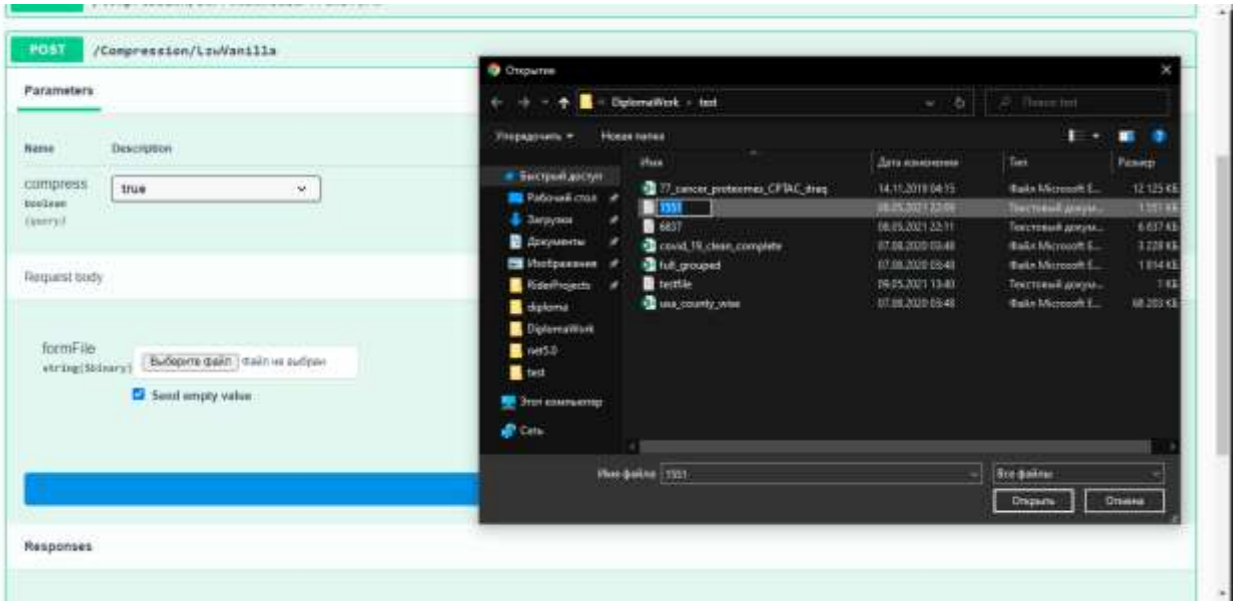


Рисунок 4. Застосування алгоритму LZW для файлу 1551.txt

Користувача зустрічає зручний інтерфейс, завдяки якому він має можливість вибрати файл, який необхідно стиснути та завантажити після стиснення його копію з меншим розміром. Після опрацювання, на сервері (така ж сама функція наявна у консольній десктопній версії) виводиться статистика по файлу, а саме: вхідний розмір файлу, вихідний розмір

стиснутого файлу, байтова різниця та ступінь стиснення.

```
info: BackCompression.Controllers.CompressionController[0]
      [09.05.2021 14:09:14]: Input file size: 1587902;
Output file size: 664207;
Difference: 923695
Compression rate: 58,170781320257795
```

Рисунок 5. Статистика після стиснення файлу 1551.txt

Як видно, завдяки простому стисненню LZW нам вдалося зменшити розмір файлу на 58%! Це досить значне зменшення знаючи про те, що це алгоритм «без втрат» та після декомпресії буде отримаю оригінальний файл з оригінальним розміром.

4.3 Метод LZW з перетворенням Барроуза-Віллера

Для реалізації цього методу було використано вже реалізовані методи. В першу чергу, файл проходить трансформацію через перетворення Барроуза – Віллера після чого завдяки алгоритму Лемпеля – Зіва – Велша стискається. Через використання перетворення BWT, алгоритм значно програє у часі стандартному LZW, але в деяких випадках стискає дещо краще. Вважаю, що недоцільно використовувати суто BWT з LZW.

```
info: BackCompression.Controllers.CompressionController[0]
      [09.05.2021 14:44:20] Lzw+Bwt Time: 20,646 ms
info: BackCompression.Controllers.CompressionController[0]
      [09.05.2021 14:44:20]: Input file size: 1587902;
Output file size: 952794;
Difference: 635108
Compression rate: 39,9966748577683
```

Рисунок 6. Результат роботи алгоритму LZW + BWT над файлом 1511.txt

Як видно, алгоритм програє стандартному, тому така зв’язка є недоцільною.

4.4 Метод Хаффмана (Huffman coding)

В реалізації методу Хаффмана було використано техніку Threads для більш гнучкої паралельності задач. Спочатку зчитується файл та будується адаптивне дерево Хаффмана. Зчитується побітово файл та далі все відбувається за класичним алгоритмом Хаффмана. Все відбувається згідно описаному в розділі 2 алгоритмі. Наведемо приклад стиснення методом Хаффмана на основі веб – застосунку. Інтерфейс такий самий, тому виведемо результат стиснення:

```
Compressed 1516596 out of 1587902
Original file CRC32 is 2041508C
info: BackCompression.Controllers.CompressionController[0]
      [09.05.2021 14:25:43] Huffman Time: 1,11 ms
info: BackCompression.Controllers.CompressionController[0]
      [09.05.2021 14:25:43]: Input file size: 1587902;
Output file size: 977329;
Difference: 610573
Compression rate: 38,45155431506478
```

Рисунок 7. Результат стиснення файлу 1551.txt методом Хаффмана

Як видно, в цьому випадку алгоритм Хаффмана стиснув дещо слабкіше ніж LZW на всього лиш 38,45%. Далі розглянемо модифікацію цього алгоритму.

4.5 Стиснення Барроуза – Віллера з MTF та методом Хаффмана

Для максимізації ступеню стиснення, варто використовувати перетворення Барроуза – Віллера разом з іншими методами, наприклад трансформації переміщення на початок (Move–to–Front coding). MTF – це кодування даних (зазвичай потік байтів), призначене для підвищення продуктивності методів стиснення ентропійного кодування. При ефективній реалізації він досить швидкий, та його переваги зазвичай виправдовують включення його в якості додаткового кроку в алгоритм стиснення даних. В такому випадку використання перетворення BWT є більш ефективним тому

що, як правило, легше стискати такий рядок, який складається з повторюваних символів за допомогою методів MTF чи RLE.

Для цього методу було використано саме MTF після якого відбувається стиснення методом Хаффмана, що дає приголомшливий результат, який конкурує навіть з методом архіватора BZip2!

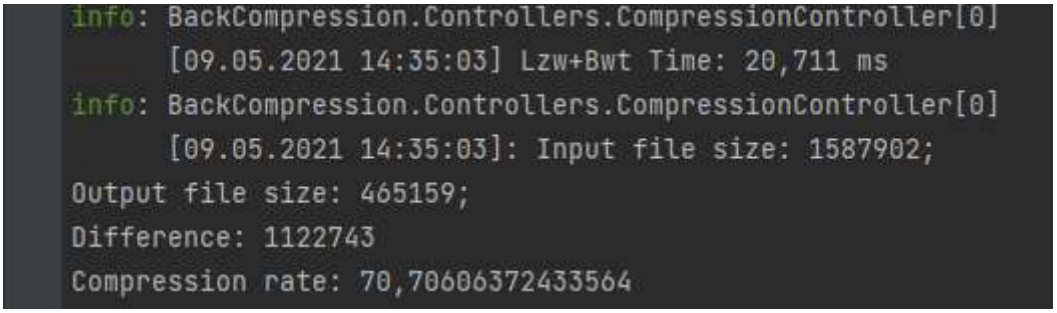


Рисунок 8. Результат стиснення файлу 1551.txt

Результат перевершує всі сподівання. В такий метод ми досягаємо стиснення файлу аж на 70 відсотків всього лиш за 20 секунд! Для додаткового коду слід звернутися до електронного джерела, де збережені мої напрацювання або дивитись додатки[50].

РОЗДІЛ 5. ПОРІВНЯННЯ РЕЗУЛЬТАТІВ СТИСКАННЯ

a. Порівняння розроблених методів між собою

Для порівняння розробленої бібліотеки з готовими рішеннями, було використано бібліотеку SharpZipLib, яка імплементує алгоритм стиснення bZip та який можливо використовувати будь – якому розробнику на правах відкритого вихідного коду[49]. Як видно з наведеного нижче графіку, метод компресії Барроуза – Віллера з MTF та методом Хаффмана демонструє прекрасний результат та конкурує з професійним методом з архіватора BZip2!

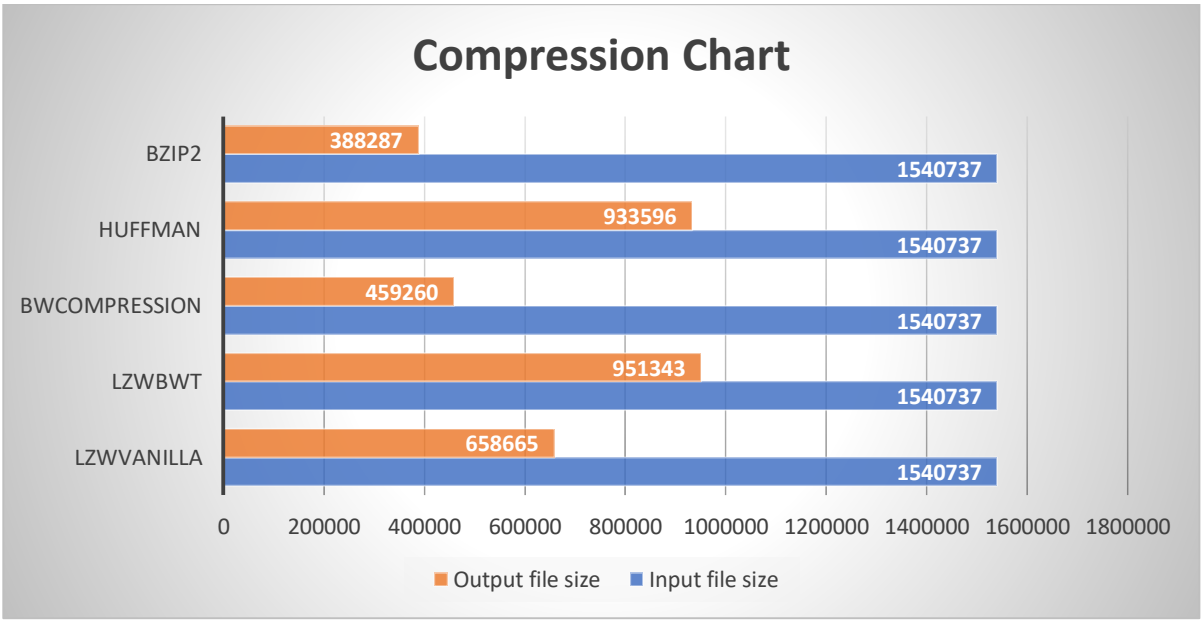


Рисунок 9. Графік стиснення файлу розміру 1551 МБ.

Наведемо відсоток стиснення для кожного з використаних методів:

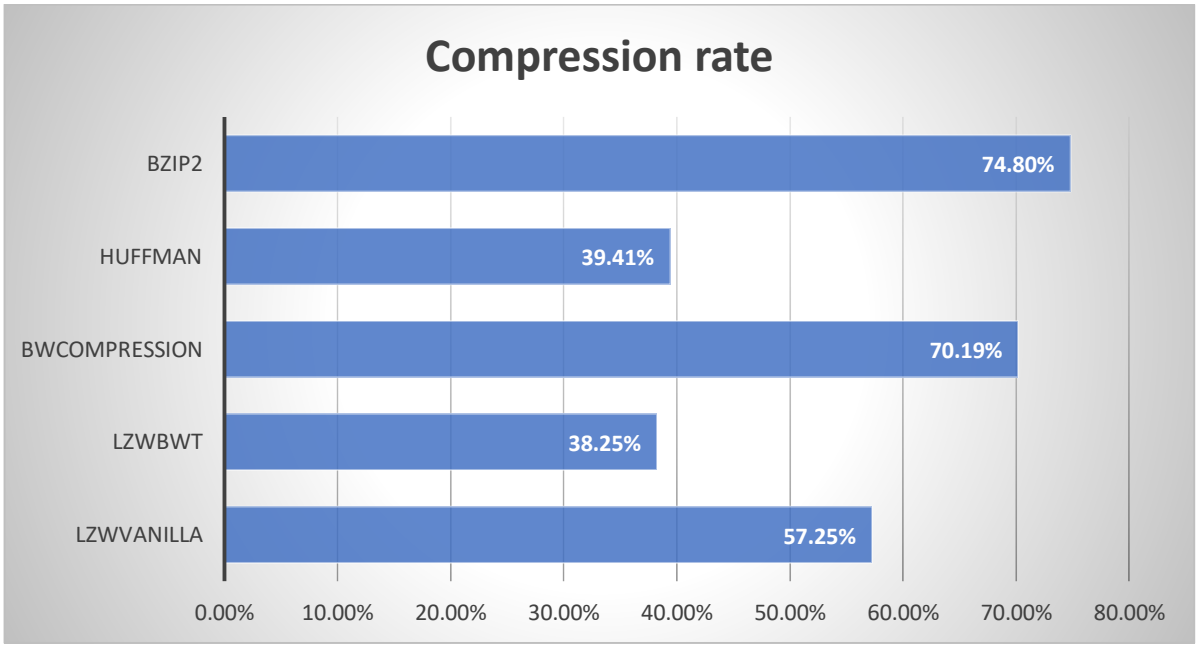


Рисунок 10. Відсоток стиснення файлу різними алгоритмами

З наведеного графіку видно, що алгоритм Хаффмана разом із LZW з BWT програють простому методу Лемпеля – Зіва – Велша у стисканні, але переможцями у цьому змаганні є власноруч написаний алгоритм стискання за допомогою перетворення Барроуза – Віллера з кодером MTF та методом Хаффмана. Додаткові досліди та статистика знаходяться у додатках.

ВИСНОВКИ

Отже, виконуючи цю роботу було опрацьовано історію та зародження стиснення даних. Оглянуто та висвітлено проблему стиснення даних, чому виникла та як великі компанії винайшли рішення проблеми. Також було оглянуто базові методи такі як:

- Кодування довжин серій (RLE).
- Перетворення Барроуза – Віллера (BWT).
- Ентропійне кодування (Entropy coding).
- Алгоритм Шеннона – Фано.
- Алгоритм Хаффмана.
- Арифметичне кодування.

Після цього було оглянуто основні алгоритми стиснення даних, їх розподіл та різниця між алгоритмами «ковзаючого вікна», безсловникових та словникових алгоритмів, а саме: LZ77, LZR, DEFLATE, DEFLATE64, LZSS, LZH, LZB, ROLZ, LZRW1, LZP, LZJB, LZS, LZX, LZO, LZMA, LZMA2, LZ78, LZW, LZC, LZT, LZMW, LZAP, LZWL, LZJ, PPM, BZIP2 та PAQ.

Завдяки розглянутим алгоритмам було розроблено бібліотеку з алгоритмами стиснення даних, а саме:

- Алгоритм Лемпеля – Зіва – Велша.
- Алгоритм Хаффмана
- Модифікований алгоритм Лемпеля – Зіва – Велша з перетворенням Барроуза – Віллера
- Перетворення Барроуза – Віллера
- Модифікований алгоритм Хаффмана з перетворенням Барроуза – Віллера та MTF алгоритмом стиснення.

та два додатки (веб – застосунок та консольний додаток) для огляду, перевірки та збору статистики. Було проведено статистичні дослідження стиснення та доцільності використовувати той чи інший алгоритм спираючись на час стиснення або відсоток стиснутих даних.

Модифікація методів зв’язкою перетворень Барроуза – Віллера та переміщенням на початок (MTF) дало значний приріст у ступені стиснення. Це чітко видно на заданих графіках. Наприклад, максимальна різниця між співвідношенням компресії для модифікованого алгоритму Хаффмана сягало 40 відсотків, що є досить гарним результатом. В деяких випадках було продемонстровано недоцільність використання модифікації алгоритму Лемпеля – Зіва – Велша з перетворенням Барроуза – Віллера без додаткових методів перетворення. Загалом, модифікація алгоритмів є доцільною та

допомагає зберегти дорогоцінне місце та мережевий трафік у випадку передачі даних за допомогою мережі Інтернет.

Отже, використання перетворення Барроуза – Віллера є доцільним разом з алгоритмами стиснення лише в тих випадках, коли воно застосовується разом з алгоритмами на кшталт RLE чи MTF. В цей спосіб ми отримаємо значний приріст у відсотку стиснення файлу, що є позитивним аспектом у розробці алгоритмів. Але на жаль це призводить до збільшення часу роботи алгоритму, але це нівелюється збільшенням потужності машини.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jeb Brains Rider [Електронний ресурс]. – Режим доступу до ресурсу: https://www.jetbrains.com/rider/.
2. C# programming language [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/C_Sharp_(programming_language).
3. .NET Library [Електронний ресурс] // Microsoft. – 2018. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/standard/library-guidance/.
4. ASP.NET Framework [Електронний ресурс] // Microsoft. – 2021. – Режим доступу до ресурсу: https://dotnet.microsoft.com/apps/aspnet.
5. Стівен В. Новий вид Науки / Вольфрам Стівен. – Іллінойс: Вольфрам Медіа, 2002. – 1069 с.
6. Lossless Data Compression: LZ77 [Електронний ресурс] – Режим доступу до ресурсу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz77/index.htm.
7. Lossless Data Compression: LZ78 [Електронний ресурс] – Режим доступу до ресурсу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz78/index.htm.
8. Huffman encoding [Електронний ресурс] – Режим доступу до ресурсу: http://homes.sice.indiana.edu/yye/lab/teaching/spring2014-C343/huffman.php.
9. Burrows-Wheeler Transform [Електронний ресурс] – Режим доступу до ресурсу: https://web.stanford.edu/class/cs262/presentations/lecture4.pdf.

10. Sliding Window Algorithm [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/cs/sliding-window-algorithm.
11. Lossless Data Compression: Shannon-Fano [Електронний ресурс] – Режим доступу до ресурсу: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/shannonfano/index.htm.
12. LZW Encoding [Електронний ресурс] – Режим доступу до ресурсу: http://www.cs.columbia.edu/~allen/S14/NOTES/lzw.pdf.
13. Grunwald P. D. Algorithmic Information Theory: дис. докт. філос. наук / Grunwald Peter D. – Амстердам, 2007. – 37 с.
14. Rate-Distortion Theory [Електронний ресурс] – Режим доступу до ресурсу: https://web.stanford.edu/class/ee368b/Handouts/04-RateDistortionTheory.pdf.
15. Huffman D. Encoding the “Neatness” of Ones and Zeroes / David Huffman. // Scientific American. – 1991. – С. 54–58.
16. Ziv J. Compression of individual sequences via variable-rate coding: 10.1109/TIT.197 / Ziv Jacob, 1978. – 530 с.
17. Phil Katz [Електронний ресурс]. – 2002. – Режим доступу до ресурсу: http://www.esva.net/~thom/philkatz.html.
18. Crocetti P. Data Compression [Електронний ресурс] / Paul Crocetti – Режим доступу до ресурсу: https://searchstorage.techtarget.com/definition/compression.
19. What is Data Compression? [Електронний ресурс] – Режим доступу до ресурсу: https://www.barracuda.com/glossary/data-compression.
20. Wayner P. Compression / Peter Wayner // Disappearing Cryptography (Third Edition) / Peter Wayner., 2009.
21. Mengyi Pu I. Dictionary-based compression / Ida Mengyi Pu // Fundamental Data Compression / Ida Mengyi Pu., 2006.
22. What about patents on data compression algorithms? [Електронний ресурс] // 2014 – Режим доступу до ресурсу: http://www.faqs.org/faqs/compression-faq/part1/section-7.html.

23. Iwata K. An Improvement in Lossless Data Compression via Substring Enumeration / Iwata K. – IEEE/ACIS 10th International Conference on Compute, 2011.
24. Wheeler D. J. A Block-sorting Lossless Data Compression Algorithm / D. J. Wheeler, M. Burrows. // SRC Research Report. – 1994. – №124. – С. 1–24.
25. Huffman D. A. A Method for the Construction of Minimum-Redundancy Code / David Huffman. // IEEE. – 1952. – №40. – С. 1098–1101.
26. A four-stage algorithm for updating a Burrows–Wheeler transform / M.Salson, T. Lecroq, M. Leonard, L. Mouchard. // String Algorithmics: Dedicated to Professor Maxime Crochemore on his 60th birthday. – 2009. – №410. – С. 4350–4359.
27. Shannon C. E. A Mathematical Theory of Communication / Shannon. // Bell System Technical Journal. – 1948. – №27. – С. 379–423.
28. Huffman D. A. A method for the construction of minimum-redundancy codes. / Huffman. // In Proceedings of the Institute of Electrical and Radio Engineers. – 1952. – №40. – С. 1098–1101.
29. Rissanen J. Arithmetic coding / J. Rissanen, G. G. Langdon. // IBM Journal of Research & Development. – 1979. – №23. – С. 149–162.
30. Rodeh. Linear algorithm for data compression via string matching / Rodeh, Pratt, Even. // ACM. – 1981. – №28. – С. 16–24.
31. Bell T. Modeling for Text Compression / T. Bell, I. Witten, J. Cleary. // ACM Computing Surveys. – 1989. – №4.
32. Storer J. Data compression via textual substitution / J. Storer, T. G. Szymanski. // ACM. – 1982. – №29. – С. 928–951.
33. Bloom C. LZP: a new data compression algorithm: 10.1109/DCC.199 / Bloom – Snowbird, 1996.
34. Williams R. LZRW1 Compression [Електронний ресурс] / Ron Williams – Режим доступу до ресурсу: http://www.ross.net/compression/.
35. Forbes J. LZX sold to Microsoft [Електронний ресурс] / Forbes – Режим доступу до ресурсу: http://www.linkedin.com/pub/jonathan-forbes/3/70a/a4b.

36. Oberhumer M. LZO information [Електронний ресурс] / Markus Oberhumer. – 2017. – Режим доступу до ресурсу: http://www.oberhumer.com/opensource/lzo/.
37. McGabe R. Data Compression Method - Adaptive Coding witih Sliding Window for Information Interchange / McGabe., 2000. – (Special Publication (NIST SP) - 500-245).
38. Lempel–Ziv–Markov chain algorithm [Електронний ресурс]. – 2011. – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm.
39. 7-Zip. Release LZMA2 [Електронний ресурс] // 9.04 beta. – 2009. – Режим доступу до ресурсу: https://www.7-zip.org/history.txt.
40. Kwong S. A Statistical Lempel-Ziv Compression Algorithm for Personal Digital Assistant (PDA) / S. Kwong, Y. F. Ho. // IEEE Transactions on Consumer Electronics. – 2001. – №47. – С. 154–162.
41. Jakobsson M. Compression of Character Strings by an Adaptive Dictionary / Jakobsson. // BIT Computer Science and Numerical Mathematics. – 1985. – №25.
42. Salomon D. Data Compression – The complete reference / David Salomon // Data Compression – The complete reference / David Salomon.. – (4th Edition). – С. 212.
43. Chernik K. Syllable-based Compression for XML Documents / K. Chernik, J. Lansky, L. Galambos. // Dateso. – 2006. – С. 21–31.
44. Cleary J. Data Compression Using Adaptive Coding and Partial String Matching / J. Cleary, I. Witten. // IEEE Transactions on Communications. – 1984. – С. 396–402.
45. Seward J. bzip2 and libbzip2 / Seward // bzip2 Manual / Seward., 2000.
46. Mahoney M. Adaptive Weighting of Context Models for Lossless Data Compression / Mahoney., 2002.

47. Task Parallel Library [Електронний ресурс] // Microsoft. – 2017. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl.
48. Nelson M. Data Compression Methods [Електронний ресурс] / Mark Nelson – Режим доступу до ресурсу: https://marknelson.us/.
49. SharpZipLib [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://github.com/icsharpcode/SharpZipLib.
50. Volokhovych I. Implementation of Compression Algorithms [Електронний ресурс] / Ihor Volokhovych – Режим доступу до ресурсу: https://github.com/antomys/DiplomaWork.

ДОДАТКИ

ДОДАТОК А

Метод стиснення LZW

```
public bool Compress(string pInputFileName, string pOutputFileName,
out string fileNamePath)
{
    Stream reader = null;
    Stream writer = null;

    try
    {
        Initialize();
        reader = new FileStream(pInputFileName, FileMode.Open);
        writer = new FileStream(pOutputFileName, FileMode.Create);
        var iNextCode = 256;

        for (var i = 0; i < TableSize; i++) //blank out table
            _iaCodeTable[i] = -1;

        var iString = reader.ReadByte();

        int iChar;
        while((iChar = reader.ReadByte()) != -1) //read until we reach
end of file
        {
            var iIndex = FindMatch(iString, iChar);

            if (_iaCodeTable[iIndex] != -1) //set string if we have
something at that index
                iString = _iaCodeTable[iIndex];
            else //insert new entry
            {
                if (iNextCode <= MaxCode) //otherwise we insert into
the tables
                {
                    _iaCodeTable[iIndex] = iNextCode++; //insert and
increment next code to use
                    _iaPrefixTable[iIndex] = iString;
                    _iaCharTable[iIndex] = (byte)iChar;
                }

                WriteCode(writer, iString); //output the data in the
string
                iString = iChar;
            }
        }

        WriteCode(writer, iString); //output last code
        WriteCode(writer, MaxValue); //output end of buffer
        WriteCode(writer, 0); //flush
    }
    catch(Exception ex)
    {
```

```
            Console.WriteLine(ex.StackTrace);
            writer?.Close();
            File.Delete(pOutputFileName);
            fileNamePath = string.Empty;
            return false;
        }
        finally
        {
            reader?.Close();
            writer?.Close();
        }

        fileNamePath = Path.GetFullPath(pOutputFileName);
        return true;
    }
```

Метод стиснення Хаффмана

```
public static void HuffConsole(string inputFile, string outputFile,
out string filePath) {
   try {
      Stream ifStream = new FileStream (inputFile, FileMode.Open,
FileAccess.Read);
      Stream ofStream = new FileStream (outputFile, FileMode.Create,
FileAccess.ReadWrite);
      var crc = new CrcCalc ();

      //Writing file signature
      ofStream.Write (sign, 0, sign.Length);
      //Padding for the header
      for (var i = 0; i < 12; i++)
         ofStream.WriteByte (0x0);

      var myCompressor = new Thread
      (o => Huff (ifStream, ofStream, val => crc.UpdateByte
((byte)val)));
      myCompressor.Start ();

      while (myCompressor.IsAlive) {
         Console.Write ("\rCompressed {0} out of {1}",
ifStream.Position, ifStream.Length);
         Thread.Sleep (100);
      }
      //Writing file length to the header
      ofStream.Seek (4, SeekOrigin.Begin);
      var buffer = BitConverter.GetBytes (ofStream.Length);
      ofStream.Write (buffer, 0, buffer.Length);

      //Writing file crc immediately after the length
      buffer = BitConverter.GetBytes (crc.GetCrc ());
      ofStream.Write (buffer, 0, buffer.Length);
      PrintHelper.Notify ("\nOriginal file CRC32 is {0:X8}\n",
crc.GetCrc ());
      ofStream.Close();
      ifStream.Close ();
      filePath = Path.GetFullPath(outputFile);
```

```
    }
    catch (Exception)
    {
       filePath = string.Empty;
                throw;
            }
}
```

Метод MTF:

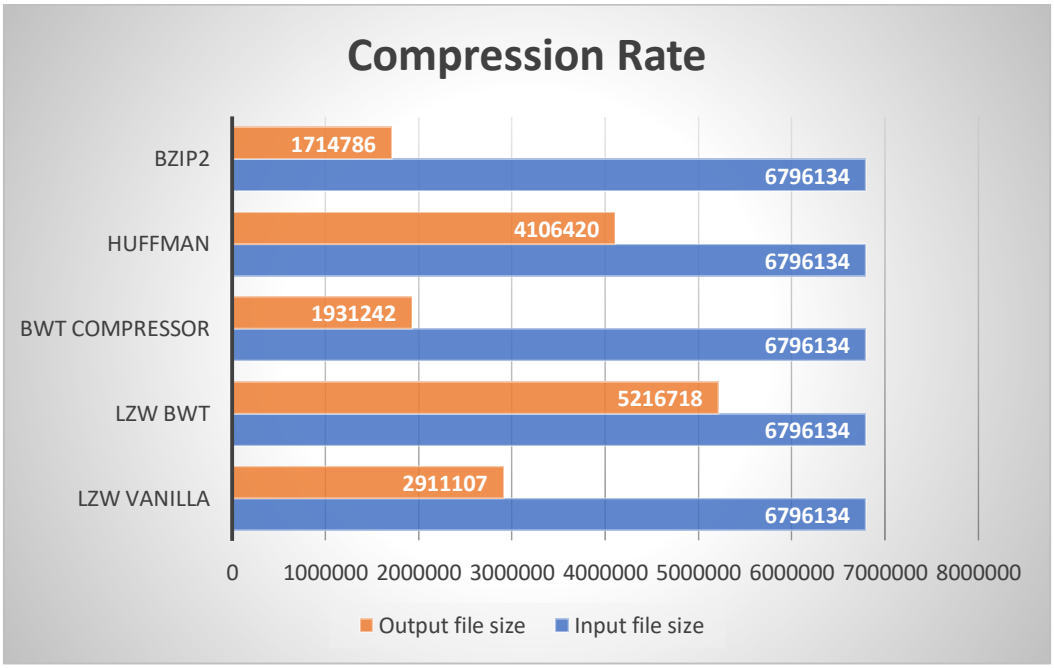
```
// complexity : O(256*N) -> O(N)
public static byte[] Encode(byte[] input)
{
    byte[] symbols = new byte[256];
    byte[] output = new byte[input.Length];

    for (int i = 0; i < 256; i++)
        symbols[i] = (byte)i;

    for (int i = 0; i < input.Length; i++)
    {
        int ind =MoveToFront(symbols, input[i]);
        output[i] = (byte)ind;
    }
    return output;
}

// complexity : O(256*N) -> O(N)
public static byte[] Decode(byte[] input)
{
    byte[] symbols = new byte[256];
    byte[] output = new byte[input.Length];

    for (int i = 0; i < 256; i++)
        symbols[i] = (byte)i;

    for (int i = 0; i < input.Length; i++)
    {
        int ind = (int)input[i];
        output[i] = symbols[ind];
        MoveToFront(symbols, ind);
    }
    return output;
}
```

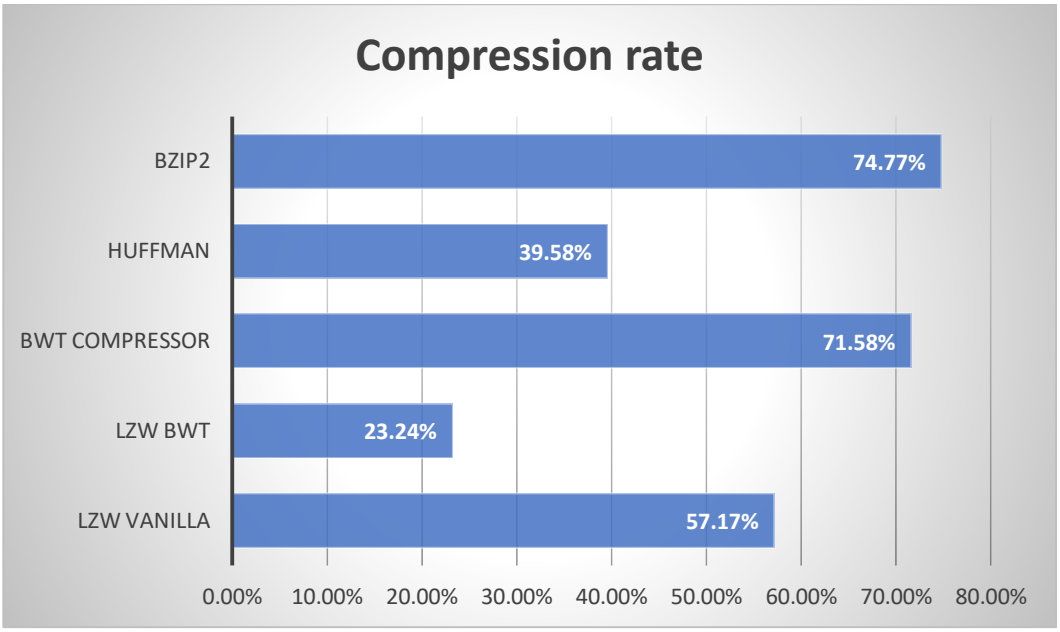
ДОДАТОК Б

Таблиці порівняння стиснення різними алгоритмами

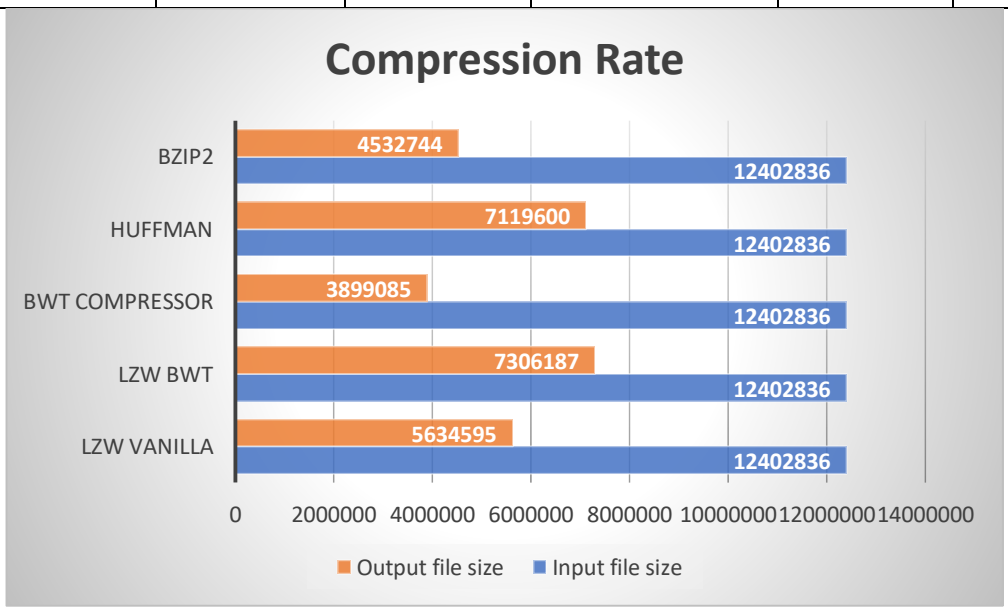
1551.txt					
	LZW Vanilla	LZW BWT	BWT Compressor	Huffman	BZip2
Input file size	1540737	1540737	1540737	1540737	1540737
Output file size	658665	951343	459260	933596	388287
Compression rate	57,25%	38,25%	70,19%	39,41%	74,80%

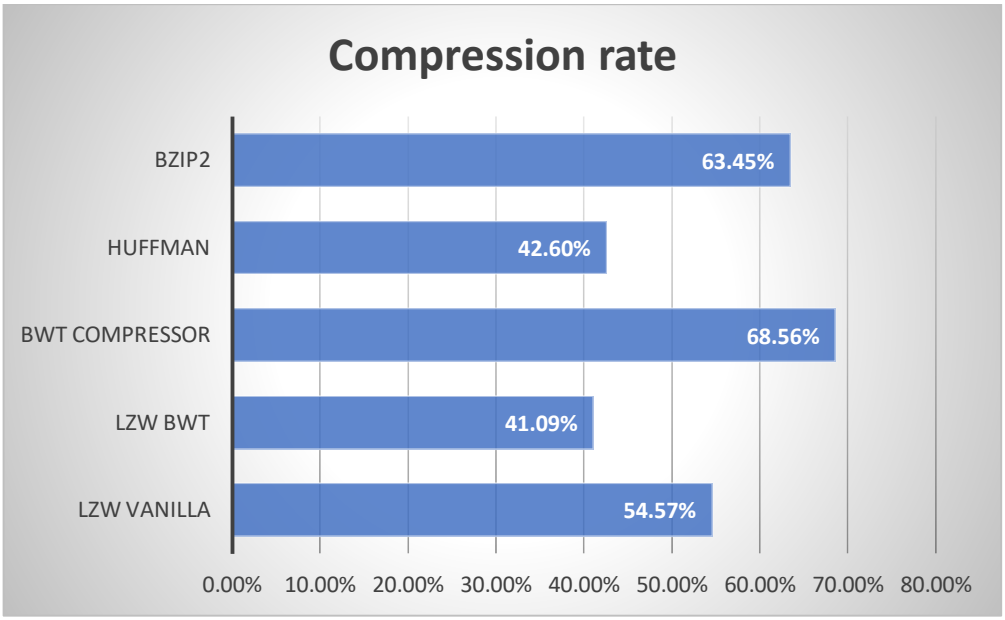
6837.txt					
	LZW Vanilla	LZW BWT	BWT Compressor	Huffman	BZip2
Input file size	6796134	6796134	6796134	6796134	6796134
Output file size	2911107	5216718	1931242	4106420	1714786
Compression rate	57,17%	23,24%	71,58%	39,58%	74,77%





77_cancer_proteomes_CPTAC_itraq.csv 1.lzw					
	LZW Vanilla	LZW BWT	BWT Compressor	Huffman	BZip2
Input file size	12402836	12402836	12402836	12402836	12402836
Output file size	5634595	7306187	3899085	7119600	4532744
Compression rate	54,57%	41,09%	68,56%	42,60%	63,45%




Compression rate
BZIP2
HUFFMAN
BWT COMPRESSOR
LZW BWT
LZW VANILLA
63.45%
42.60%
68.56%
41.09%
54.57%
0.00%
10.00%
20.00%
30.00%
40.00%
50.00%
60.00%
70.00%
80.00%