ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка та оптимізація алгоритмів стиснення даних з метою підвищення пропускної спроможності мережі»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерна інженерія
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олексій Погоржевський
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. КІД-41
Олексій Погоржевський
Ім'я, ПРІЗВИЩЕ
Керівник: Ярослав Торошанко
Ім'я, ПРІЗВИЩЕ
Рецензент: _____
Ім'я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерної інженерії

Ступінь вищої освіти бакалавр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедрою Наталія Лащевська
Ім'я, ПРІЗВИЩЕ
«___»_____2024р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Погоржевський Олексій Романович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка та оптимізація алгоритмів стиснення даних з метою підвищення пропускної спроможності мережі.

керівник кваліфікаційної роботи Ярослав Торошанко, к.т.н., с.н.с.
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від « 27 » лютого 2024 р. № 36.

2. Строк подання кваліфікаційної роботи 10 травня 2024 р.

3. Вихідні дані до кваліфікаційної

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1.
   2.
   3.

5. Перелік ілюстративного матеріалу:

6. Дата видачі завдання « 13 » лютого 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Збір даних	26.02-22.03.2024	Виконано
2	Обробка матеріалу	23.03-05.04.2024	Виконано
3	Розробка теоретичної частини	06.04-10.04.2024	Виконано
4	Розробка практичної частини	11.04-30.04.2024	Виконано
5	Висновки за результатами аналізу	01.05-03.05.2024	Виконано
6	Оформлення документу	04.05-08.05.2024	Виконано
7	Розробка презентації	09.05-11.05.2024	Виконано
8	Здача в деканат	27.05-31.05.2024	Виконано

Здобувач вищої освіти _____ _____
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ _____
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 40 стор., 12 рис., 0 табл., 19 джерел.

Мета роботи – дослідження різних основоположних алгороитмів стиснення даних з метою зменшення їх об'єму для передачі в мережі

Об'єкт дослідження – розбір роботи різних алгоритмів стиснення даних

Предмет дослідження – виконання одного з алгоритмів стиснення даних на мові програмування Python та можливості його оптимізування.

Короткий зміст роботи:

Розділ 1: Історія створення алгоритмів стиснення даних

Цей розділ присвячений історичному огляду розвитку методів стиснення даних. Розглядаються основні етапи еволюції алгоритмів стиснення, починаючи від ранніх методів до сучасних підходів. Аналізуються важливі досягнення в цій області та внесок відомих алгоритмів у розвиток технології стиснення даних.

Розділ 2: Програми створені на базі алгоритмів стиснення даних

У цьому розділі проводиться огляд популярних програм та утиліт, що використовують алгоритми стиснення даних. Описуються особливості різних програм та їх порівняння за ефективністю. Розглядається вплив стиснення даних на продуктивність мережі, а також наводяться практичні приклади використання таких програм в реальних умовах.

Розділ 3: Проста програмна реалізація алгоритму LZW використовуючи мову програмування Python

Цей розділ фокусується на алгоритмі LZW (Lempel-Ziv-Welch). Описуються основні переваги та недоліки цього алгоритму. Детально розглядаються кроки реалізації алгоритму LZW на мові програмування Python, включаючи підготовку середовища розробки та необхідних інструментів. Після реалізації проводиться тестування та оптимізація алгоритму. Розділ завершується аналізом ефективності реалізованого алгоритму та можливими напрямками для його покращення.

ABSTRACT

The text part of the qualification work for the bachelor's degree: 40 pages, 12 figures, 0 tables, 19 sources.

Purpose - to study the various basic data compression algorithms in order to reduce the amount of data to be transmitted in the network

Object of research - analysis of the work of various data compression algorithms

The subject of research - the implementation of one of the data compression algorithms in the Python programming language and the possibilities of its optimization.

Summary of the work:

Section 1: History of data compression algorithms

This chapter is devoted to a historical overview of the development of data compression methods. The main stages of the evolution of compression algorithms are considered, starting from early methods to modern approaches. Important achievements in this field and the contribution of well-known algorithms to the development of data compression technology are analyzed.

Section 2: Programs based on data compression algorithms

This section provides an overview of popular programs and utilities that use data compression algorithms. It describes the features of different programs and compares their efficiency. The impact of data compression on network performance is discussed, and practical examples of using such programs in real-world conditions are provided.

Chapter 3: A simple software implementation of the LZW algorithm using the Python programming language

This chapter focuses on the LZW (Lempel-Ziv-Welch) algorithm. The main advantages and disadvantages of this algorithm are described. The steps for implementing the LZW algorithm in Python are discussed in detail, including preparing the development environment and the necessary tools. After implementation, the algorithm is tested and optimized. The chapter concludes with an analysis of the effectiveness of the implemented algorithm and possible areas for its improvement.

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

ПОДАННЯ
ГОЛОВІ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня бакалавра

Направляється здобувач _____до захисту кваліфікаційної роботи
(прізвище та ініціали)
за спеціальністю_____
(код, найменування спеціальності)
освітньо-професійної програми _____
(назва)
на тему: «_____».

Кваліфікаційна робота і рецензія додаються.

Директор ННІІТ _____ _____
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Висновок керівника кваліфікаційної роботи

Здобувач_____
_____
_____
_____

Все це дозволяє оцінити виконану кваліфікаційну роботу здобувача_____ на оцінку «_____» та присвоїти йому кваліфікацію _____.

Керівник кваліфікаційної роботи _____ _____
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

«_____»_____2024р.

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач _____ допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедрою _____ _____ _____
(назва) *(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну бакалаврську роботу

здобувача вищої освіти Погоржевський Олексій Романович
(прізвище, ім'я, по батькові)

на тему «Розробка та оптимізація алгоритмів стиснення даних з метою підвищення пропускної спроможності мережі»

Актуальність.

Пропускна здатність мережі визначає максимальний обсяг даних, який може бути переданий через мережу за певний час. Ця характеристика стає ключовою у високовитратних мережах, де значні обсяги даних передаються на великі відстані. Підвищення пропускної здатності мережі сприятиме ефективнішому використанню інфраструктури та забезпечить задоволення потреб користувачів у швидкій та безперервній передачі даних.

Позитивні сторони.

1. У бакалаврській роботі було розглянуті різні алгоритми стискання даних та їх практичне застосування в реальному житті.
2. В ході підготовки та написання бакалаврської роботи студентом було розглянуто та використано велику кількість програм для стискання даних що допомогли виконати роботу.

Недоліки.

1. В бакалаврській роботі розглянуто замало можливостей для оптимізації алгоритму.

Відзначені зауваження не впливають на загальну позитивну оцінку кваліфікаційної бакалаврської роботи.

Висновок: *кваліфікаційна бакалаврська робота заслуговує оцінку «ДОБРЕ», а здобувач Погоржевський О.Р. заслуговує присвоєння кваліфікації: бакалавр.*

Рецензент: _____ _____
(підпис) *Ім'я, ПРІЗВИЩЕ*

ЗМІСТ

ВСТУП 9
РОЗДІЛ 1. ІСТОРІЯ СТВОРЕННЯ АЛГОРИТМІВ СТИСНЕННЯ ДАНИХ 11
1.1 Алгоритм Шеннона-Фано 11
1.2 Кодування Хаффмана 12
1.3 Кодування Лемпела-Зіва 15
1.4 Алгоритм Лемпела-Зіва-Велча 18
1.5 DEFLATE 21
1.6 RLE 25
РОЗДІЛ 2. ПРОГРАМИ СТВОРЕНІ НА БАЗІ АЛГОРИТМІВ СТИСНЕННЯ ДАНИХ 28
2.1 UNIX Compress 28
2.2 GZIP 30
2.3 PKZIP 32
2.4 bzip2 34
РОЗДІЛ 3. ПРОСТА ПРОГРМНА РЕАЛІІЗАЦІЯ АЛГОРИТМУ LZW ВИКОРИСТОВУЮЧИ МОВУ ПРОГРАМУВАННЯ PYTHON 41
3.1 Алгоритм LZW, опис та виконання коду 41
3.2 Оптимізація створеного алгоритму 46
ВИСНОВОК 52
СПИСОК ДЖЕРЕЛ 53
Додаток А 55

ВСТУП

У сучасному цифровому світі, що неперервно та стрімко розвивається, обсяги передаваних у мережах даних постійно зростають. Цей тренд обумовлений багатьма чинниками, включаючи збільшення кількості веб-контенту, потоку відео та аудіо матеріалів, а також іншої цифрової інформації, яка генерується та обмінюється між користувачами щодня. Проте, на фоні цього зростає й вимога до ефективності передачі даних, що виражається у підвищенні пропускної здатності мережі.

Пропускна здатність мережі визначає максимальний обсяг даних, який може бути переданий через мережу за певний час. Ця характеристика стає ключовою у високовитратних мережах, де значні обсяги даних передаються на великі відстані. Підвищення пропускної здатності мережі сприятиме ефективнішому використанню інфраструктури та забезпечить задоволення потреб користувачів у швидкій та безперервній передачі даних.

Одним з методів підвищення пропускної здатності мережі є вдосконалення алгоритмів стиснення даних. Стискання даних полягає у зменшенні їх обсягу шляхом вилучення зайвої інформації або представлення даних у більш компактному форматі. Ефективні методи стискання можуть значно зменшити обсяг передаваних даних, що, у свою чергу, призводить до підвищення пропускної здатності мережі та поліпшення її продуктивності.

Проте, зростання пропускної здатності мережі вимагає комплексного підходу. Крім вдосконалення алгоритмів стиснення даних, необхідно також розвивати інфраструктуру мережі, використовувати новітні технології передачі даних та управління трафіком. Цей комплексний підхід дозволить ефективно відповісти на зростаючі вимоги до пропускної здатності мережі в сучасному інформаційному середовищі.

Зараз ми стикаємося з різноманітними викликами, пов'язаними із зростанням обсягів передаваних даних. Одним із таких викликів є висока динаміка змін інтернет-трафіку. За останні роки споживання веб-контенту, такого як стрімінг

відео та аудіо, соціальні мережі та інші цифрові платформи, стрімко зростає. Це призводить до збільшення обсягів передаваних даних і створює тиск на пропускну здатність мережі.

РОЗДІЛ 1. ІСТОРІЯ СТВОРЕННЯ АЛГОРИТМІВ СТИСНЕННЯ ДАНИХ

1.1 Алгоритм Шеннона-Фано

Раннім та найвідомішим всьому людству алгоритмом стиснення даних є азбука Морзе винайдена в 1838 році для використання в телеграфії і заснована на використанні коротших кодових слів для таких літер, як "e" і "t", які є більш поширеними в англійській мові. Більш сучасні роботи зі стискання даних розпочалися у кінці 1940-х років із започаткування та розвитком теорії інформації. У 1949 році два вчених Клод Шеннон та Роберт Фано винайшли систематичний спосіб призначення кодових слів на основі ймовірностей блоків. Пізніше Девід Хаффман знайшов більш оптимальний метод для цього способу який потім отримав назву кодування Хаффмана.

Систематичний спосіб призначення кодових слів на основі ймовірностей блоків Шенона і Фано (алгоритм Шеннона-Фано) є спорідненим між двома їхніми методами для побудови префіксного коду на основі набору символів та їхніх ймовірностей які були або оцінені, або виміряні. Метод Шеннона вибирає код префікса, де вихідний символ це задана довжина кодового слова. Натомість, метод Фано ділить вихідні символи на набори 0 та 1 з імовірністю близькою до ½, потім ці набори ділять на дві частини поки кожен з них не міститиме лише один символ.

Основні етапи кодування Шеннона-Фано:

а) вхідне повідомлення записується в таблицю частоти повторень кожного символу і всі частоти сумуються між собою (таким чином утворюється коріння так званого дерева кодування);

б) символи поділяють навпіл на приблизно рівні частоти;

в) в двох утворених гілках продовжується операція розділення частот символів;

г) отриманим частинам призначаються відповідні двійкові цифри в префіксному коді (для гілок що йдуть першими (з лівої сторони) призначається код 1, а для гілок які йдуть другими (з правої сторони) призначається код 0).

Приклад роботи алгоритму Шеннона-Фано зображено на рисунку 1.1.

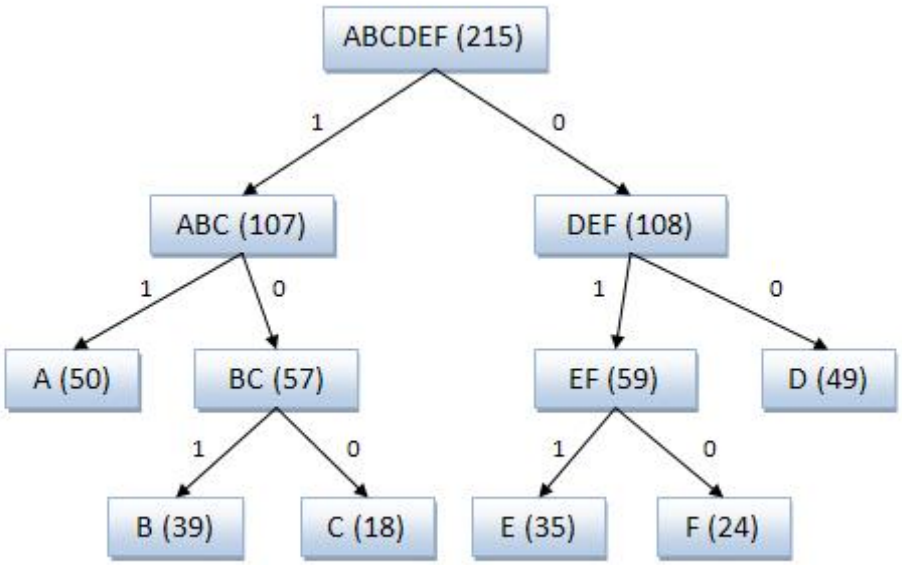


Рисунок 1.1 Просте дерево виконання алгоритму Шеннона-Фано

На кроці ділення алфавіту існує неоднозначність, так як різниця сумарних частот може бути однакова для двох варіантів поділу (враховуючи, що всі символи початкового алфавіту мають частоту більше нуля).

1.2 Кодування Хаффмана

Девід Хаффман був першопроходьцем у сфері теорії інформації. Навчаючись у Массачусетському технологічному інституті та будучи аспірантом при написанні курсової роботи створив алгоритм префіксного кодування з мінімальною надмірністю також відомий як код Хаффмана, або алгоритм Хаффмана. Його робота було опублікована в статті 1952 року «A method for the Construction of Minimum-Redundancy Codes». Алгоритм Хаффмана активно використовується і у наш час. На його основі було створено велику кількість більш просунутих алгоритмів стиснення даних які в наші часи дають нам можливості архівування, стиснення якості зображення для швидшої передачі та інше.

Алгоритм Хаффмана також називають адаптивним жадібним алгоритмом. Поняття адаптивний жадібний алгоритм пояснюється як простий і прямолінійний алгоритм який приймає найкраще рішення виходячи з наявних даних на кожному етапі і не зважаючи на можливі наслідки отримує оптимальний розв’язок.

На відмінну від кодування Шеннона-Фано, алгоритм Хаффмана завжди є оптимальним для вторинних алфавітів з більш ніж двома символами.

Основна ідея алгоритму полягає в тому, що знаючи частоту появи символів у повідомленні, можна описати процедуру побудови кодів змінної довжини так - символам з більшою частотою ставляться у відповідність коротші коди, а символам з найменшою частотою навпаки.

Кодування алгоритмом Хаффмана має властивість префіксності (ні одне кодове слово не може бути префіксом іншого), це дає можливість декодувати отримане закодоване повідомлення.

Даний алгоритм має два основні етапи:

а) розробка так званого дерева;

б) відображення код-символ на основі розробленого дерева.

Класичний алгоритм Хаффмана отримує вхідний алфавіт з частотою повторень символів у повідомленні. Потім використовуючи дані цієї таблиці будується дерево кодування яке має починатися з листа дерева і рухатися вгору до кореня, збираючи біти, які потрібно передати. Натомість, декодеру не має значення, яку довжину має код, який він декодує, оскільки він просто рухається від кореня до листа дерева кодування, вибираючи по одному біту з вхідного потоку.

Основною проблемою алгоритмів Хаффмана є переповнення.

Основні етапи кодування Хаффмана:

а) маючи вхідний алфавіт з частотою повторень кожного символу повідомлення записує їх у таблицю і використовує частоту повторень в якості пріоритету;

б) обирає два вузли з найменшими пріоритетами та поєднує їх у один вузол, а сума частот двох з’єднаних вузлів буде частотою отриманого вузла (тобто маючи вузли з частотою 1 та 2 отримаємо один вузол з частотою 3);

в) отриманий вузол переміщається по таблиці пріоритетів в залежності від отриманого номеру.

Подальша послідовність дій виконується повторно відповідно наведеним вище пунктам поки вся послідовність не дійде до коріння дерева (останнього символу з найбільшою частотою). Ілюстрація роботи алгоритму показана на рисунку 1.2.

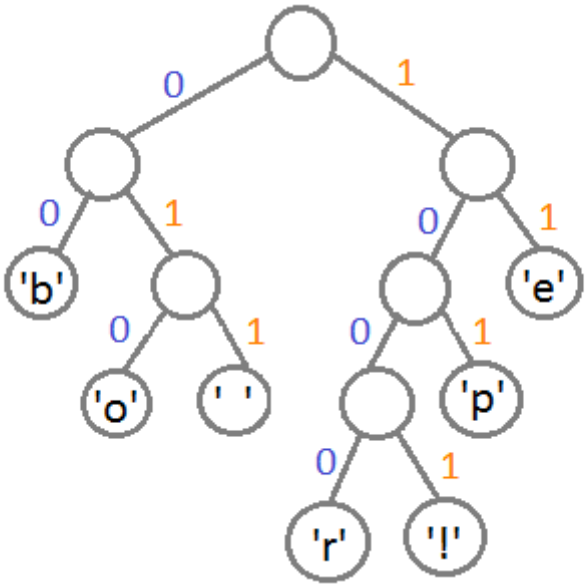


Рисунок 1.2 Ілюстрація роботи алгоритму Хаффмана

Окрім класичного кодування існує ще:

а) адаптивне кодування - передбачає динамічне обчислення ймовірностей на основі останніх фактичних частот у послідовності вихідних символів і зміну структури дерева кодування відповідно до оновлених оцінок ймовірностей. На практиці використовується рідко, оскільки вартість оновлення дерева робить його повільнішим за оптимізоване адаптивне арифметичне кодування, яке є більш гнучким і має краще стиснення;

б) шаблон алгоритму Хаффмана - дозволяє використовувати будь-який тип ваг (вартість, частоти, пари ваг, нечислові ваги) і один з багатьох методів комбінування (не тільки додавання);

в) обмежене за довжиною кодування Хаффмана - це варіант, де метою все ще є досягнення мінімальної зваженої довжини шляху, але є додаткове обмеження, що довжина кожного кодового слова має бути меншою за задану константу;

г) кодування Хаффмана з нерівною вартістю літер - літери кодуючого алфавіту можуть мати нерівномірну довжину, що зумовлено характеристиками середовища передачі;

д) оптимальні алфавітні бінарні дерева - у стандартній задачі кодування Хаффмана передбачається, що будь-якому вхідному символу може відповідати будь-яке кодове слово тоді як в алфавітному варіанті алфавітний порядок входів і виходів повинен бути ідентичним;

е) канонічний код Хаффмана – код отриманий в результаті числового впорядкування вхідних даних.

1.3 Кодування Лемпела-Зіва

Наприкінці 1970-х років, коли онлайн зберігання текстових файлів стало поширеним, почали розроблятися програми стиснення програмного забезпечення майже всі засновані на адаптивному кодуванні Хаффмана. У травні 1977 році Авраам Лемпель у співтоваристві з Яковом Зівом створили та представили універсальний алгоритм стиснення даних без втрат LZ77 та LZ78.

Початком для створення нового алгоритму відмінного від відомого алгоритму Хаффмана стала запропонована ідея формування так званого словника послідовностей входження повідомлення. Стиснення даних відбувалося за рахунок заміни схожих вхідних даних даними які є в створеному словнику.

Використовуючи алгоритм Лемпела-Зіва було створено велику кількість можливих алгоритмів стиснення даних, але всі створені алгоритми мали один з трьох теоретичних підходів до зменшення даних в повідомленні. Перший підхід включає в себе зміну вмісту даних, другий - зміну структури даних, а третій - одночасну зміну як структури, так і вмісту даних.

Якщо під час стиснення даних змінюється їх вміст, то цей метод стиснення є незворотнім, тобто при відновленні даних з архіву не можна повністю відновити інформацію. Такі методи часто називаються методами стиснення з контрольованими втратами інформації. Очевидно, що такі методи можна

використовувати лише для даних, де втрата частини вмісту не призводить до суттєвого спотворення інформації, наприклад, відео, аудіо та графічні дані. Методи стиснення з контрольованими втратами інформації забезпечують великий ступінь стиснення, але їх не можна застосовувати до текстових даних.

Прикладами таких методів можуть бути:

а) JPEG (Joint Photographic Experts Group) для графічних даних;

б) MPG - для відеоданих;

в) MP3 - для аудіоданих.

Якщо під час стиснення даних змінюється лише їх структура, то цей метод стиснення є зворотнім. Під час такого кодування можна повністю відновити інформацію з архіву. Зворотні методи стиснення можна застосовувати до будь-яких типів даних, але вони надають менший ступінь стиснення порівняно з незворотніми методами стиснення.

Приклади форматів стиснення без втрати інформації включають:

а) GIF та TIFF - для графічних даних;

б) AVI - для відеоданих;

в) ZIP, ARJ, RAR, CAB, LH - для різних типів даних.

LZ77 використовує вже переглянуту частину повідомлення як словник. Щоб досягти стиснення, він намагається замінити вхідний фрагмент повідомлення на фрагмент який вже відомий вмісту словника.

В якості моделі даних LZ77 використовує вікно, що ковзає за повідомленням, розділене на дві нерівні частини. Перша, велика за розміром, включає вже переглянуту частину повідомлення, друга - набагато менша і є буфером, що містить ще не закодовані символи вхідного потоку.

Зазвичай розмір вікна становить кілька кілобайтів. Буфер набагато менше, зазвичай трохи більше ста байтів. Алгоритм намагається знайти у словнику фрагмент, що збігається із вмістом буфера. Ілюстрація роботи показана на рисунку 1.3.

Position	Input: 'AACAACABCABAAAC'		Output Pair
	Lookup Array	Encoded Array	(offset, length)
0	∅	A ACAACABCABAAAC	(0, 'A')
1	A	A CAACABCABAAAC	(1, 1)
2	AA	C AACABCABAAAC	(0, 'C')
3	AAC	AACA BCABAAAC	(3, 4)
7	AACAACA	B CABAAAC	(0, 'B')
8	AACAACAB	CAB AAAC	(3, 3)
11	AACAACABCAB	AA AC	(11, 2)
13	AACAACABCABAA	AC	(12, 2)

Рисунок 1.3 Ілюстрацію роботи алгоритму LZ77

Як і всі попередні алгоритми, цей також має певні проблеми з розв'язанням задач і основна проблема алгоритму LZ77 є його швидкодія. Вона залежить від того яким саме способом буде виконуватися пошук збігів в так званому словнику.

Якщо шукати збіг повним перебором всіх можливих варіантів, очевидно, що стиснення буде дуже повільним. Причому при збільшенні розмірів вікна для підвищення степеню стиснення, швидкість роботи буде пропорційно зменшуватися. Для декодера це не має значення, оскільки при декодуванні не здійснюється пошук збігу.

Швидкодія і кодера, і декодера залежить від того, як реалізовано "ковзання" вікна за вмістом повідомлення. Раціонально було б для роботи з вікном користуватися кільцевим буфером, організованим на фізично суцільній ділянці пам'яті, а не реальним зсувом вікна. Хоча для підтримки кільцевого буферу необхідні додаткові витрати на збереження цілісності індексів у ньому, загалом це дає дуже істотний виграш тому що відсутнє постійне зрушення великого блоку пам'яті.

Крім проблем зі швидкодією, у алгоритму LZ77 виникають проблеми із самим стиском. Вони з'являються, коли кодер не може знайти збіг у словнику і видає стандартний 3-компонентний код, намагаючись закодувати один символ. Це буде займати багато місця і значно знижувати показники продуктивності алгоритму.

1.4 Алгоритм Лемпела-Зіва-Велча

Алгоритм був опублікований вченим Террі Велчем у 1984 році як більш вдосконалена версія алгоритму LZ78. Алгоритм простий у реалізації та має потенціал для дуже високої пропускної здатності в апаратних реалізаціях. Це алгоритм утиліти стиснення файлів Unix compress і використовується у форматі зображень GIF.

Сценарій, описаний у статті Велча 1984 року, кодує послідовності 8-бітових даних 12-бітними кодами фіксованої довжини. Коди від 0 до 255 представляють 1-символьні послідовності, що складаються з відповідного 8-бітового символу, а коди від 256 до 4095 створюються у словнику для послідовностей, що зустрічаються в даних під час їх кодування. На кожному етапі стиснення вхідні байти збираються в послідовність до тих пір, поки наступний символ не утворить послідовність, яка ще не має коду в словнику. Код послідовності (без цього символу) додається до вихідних даних, а новий код (для послідовності з цим символом) додається до словника.

Словник ініціалізовано так, щоб він містив односимвольні рядки, які відповідають усім можливим вхідним символам (і нічого іншого, крім кодів очищення та зупинки, якщо вони використовуються). Алгоритм працює, скануючи вхідний рядок на наявність послідовно довших підрядків, поки не знайде такий, якого немає у словнику. Коли такий рядок знайдено, індекс рядка без останнього символу (тобто найдовшого підрядка зі словника) витягується зі словника і надсилається на вивід, а новий рядок (включно з останнім символом) додається до словника з наступним доступним кодом. Останній введений символ використовується як наступна точка відліку для пошуку підрядків.

Таким чином, послідовно довші рядки реєструються в словнику і стають доступними для подальшого кодування як окремі вихідні значення. Алгоритм найкраще працює на даних з повторюваними шаблонами, тому початкові частини повідомлення стискаються незначно. Однак, коли повідомлення зростає, коефіцієнт стиснення асимптотично прагне до максимуму (тобто коефіцієнт

стиснення покращується по зростаючій кривій, а не лінійно, наближаючись до теоретичного максимуму за обмежений проміжок часу, а не протягом нескінченного часу).

Алгоритм декодування працює зчитуючи значення з закодованого входу і виводячи відповідний рядок зі словника. Однак повний словник не потрібен, лише початковий словник, який містить односимвольні рядки (і який зазвичай жорстко кодується в програмі, а не надсилається разом із закодованими даними). Замість цього повний словник перебудовується під час процесу декодування наступним чином: після декодування значення і виведення рядка, декодер об'єднує його з першим символом наступного декодованого рядка (або з першим символом поточного рядка, якщо наступний рядок не може бути декодований; оскільки якщо наступне значення невідоме, то це має бути значення, додане до словника на цій ітерації, і тому його перший символ збігається з першим символом поточного рядка), і оновлює словник новим рядком. Потім декодер переходить до наступного входу (який вже було прочитано на попередній ітерації) і обробляє його, як і раніше, і так до тих пір, поки не вичерпає вхідний потік.

Пояснюючи простими словами процес кодування починається з ініціалізації словника в який одразу вноситься початковий алфавіт A, B, C, D, E, F и так далі аж до Z. Після ініціалізації словника починається кодування з першого символу і якщо цей символ є в словнику на вихід подається його код з словника, наприклад, якщо це символ A то на вихід виводиться 0, тобто його код в словнику.

Далі дещо складніше. Так як символ A вже є в словнику ми залишаємо його і до нього додається наступний символ, наприклад, B. Тоді наш запис має вигляд AB і так як B йшла наступною, на вихід ми подаємо її код в словнику, тобто 1, а комбінацію AB записуємо у словник як нову комбінацію і призначаємо їй новий код в словнику. Якщо нам знову зустрічається комбінація AB, наприклад, якщо після нашої B йшла A і за нею одразу B, наша комбінація для кодування буде виглядати так: спочатку отримуємо нову комбінацію BA і записуємо її у словник та додаємо їй новий код у словнику, на наступному кроці зустрічаємо знову AB яке вже є в словнику тому на вихід подаємо код який вже записаний у словнику и ця

комбінація залишається на наступний крок і до неї додається наступний символ, наприклад C. Отримуємо нову комбінацію ABC і записуємо її у словник. Схожі дії виконуються на всіх етапах кодування повідомлення алгоритмом LZW.

Декодування відбувається зворотнім шляхом. На вхід отримуємо закодоване повідомлення, для початку роботи декодування знову ініціалізується стандартний словник з односимвольним алфавітом. Далі кожен код який є в послідовності на вході проходить через алфавіт і підставляється символ співпадаючий даному коду. Приклад виконання алгоритму LZW видно на рисунку 1.4.

INPUT SEQUENCE:ACGACCACCCGACAGGT						
SLNo:	CHAR	STR + CHAR	IN DICT	ADD TO DICT	STRING	O/P
1	A	A			A	
2	C	AC	N	5=AC	C	1
3	G	CG	N	6=CG	G	3
4	A	GA	N	7=GA	A	2
5	C	AC=5	Y		AC	
6	C	ACC	N	8=ACC	C	5
7	A	CA	N	9=CA	A	3
8	C	AC=5	Y		AC	
9	C	ACC=8	Y		ACC	
10	C	ACCC	N	10=ACCC	C	8

Рисунок 1.4 Ілюстрація виконання алгоритму LZW

Відмінністю покращеної версії алгоритму Велча від алгоритму Лемпела-Зіва є те, що при відправці закодованого повідомлення словник кодування не передається разом з пакетом повідомлення, він формується знову в точці куди буде доставлений пакет повідомлення.

1.5 DEFLATE

У комп'ютерних технологіях Deflate (стилізований під DEFLATE, а також називається Flate) - це формат стиснення даних без втрат, який використовує комбінацію кодування LZ77 і Huffman. Його розробив Філ Кац для версії 2 свого інструменту архівації PKZIP. Пізніше Deflate було описано в RFC 1951 (1996).

Кац також розробив оригінальний алгоритм, який використовується для побудови потоків Deflate. Цей алгоритм був запатентований як патент США 5,051,745 і закріплений за компанією PKWARE, Inc. Як зазначено в документі RFC, алгоритм створення файлів Deflate вважався таким, що може бути реалізований у спосіб, не захищений патентами. Це призвело до його широкого використання - наприклад, у стиснутих файлах gzip і файлах зображень PNG, на додаток до формату ZIP, для якого Кац спочатку його розробив. З тих пір термін дії патенту закінчився.

На етапі стиснення саме кодувальник обирає кількість часу, витраченого на пошук відповідних рядків. Реалізація стандарту zlib/gzip дозволяє користувачеві вибирати зі змінної шкали ймовірний рівень стиснення порівняно зі швидкістю кодування. Параметри варіюються від 0 (не намагатися стискати, просто зберігати без стиснення) до 9, що відповідає максимальним можливостям еталонної реалізації у zlib/gzip.

Було створено інші кодувальники Deflate, які також створюють сумісний бітовий потік, що може бути розпакований будь-яким існуючим декодером Deflate. Різні реалізації, ймовірно, призведуть до варіацій кінцевого закодованого бітового потоку. Зазвичай, у не-zlib версіях кодерів основна увага приділяється створенню більш ефективно стисненого і меншого за розміром кодованого потоку.

Реалізації Deflate є у вільному доступі багатьма мовами. Програми, написані на C, зазвичай використовують бібліотеку zlib (за ліцензією zlib). Програми на Borland Pascal (і сумісних мовах) можуть використовувати paszlib. Програми на C++ можуть скористатися покращеною бібліотекою Deflate у 7-Zip. І Java, і .NET Framework пропонують підтримку Deflate у своїх бібліотеках (відповідно,

java.util.zip і System.IO.Compression). Програми на Ada можуть використовувати Zip-Ada (чистий) або ZLib-Ada.

Якщо в стиснутих блоках виявляється повторювана серія байтів (повторюваний рядок), то замість неї вставляється зворотне посилання, яке посилається на попереднє місце розташування цього ідентичного рядка. Закодований збіг з попереднім рядком складається з 8-бітної довжини (3-258 байт) і 15-бітної відстані (1-32,768 байт) до початку повторюваного рядка. Відносні зворотні посилання можуть бути зроблені на будь-яку кількість блоків, якщо відстань знаходиться в межах останніх 32 Кбайт декодованих нестиснених даних (так зване «ковзаюче вікно»).

Якщо відстань менша за довжину, дублікат накладається сам на себе, вказуючи на повторення. Наприклад, послідовність з 10 однакових байтів може бути закодована як один байт, за яким слідує дублікат довжиною 9, починаючи з попереднього байта.

Другий етап стиснення полягає в заміні часто використовуваних символів більш короткими представленнями, а менш часто використовуваних символів - більш довгими представленнями. Використовується метод кодування Хаффмана, який створює непрефіксоване дерево інтервалів, що не перекриваються, де довжина кожної послідовності обернено пропорційна логарифму ймовірності того, що цей символ потрібно буде кодувати. Чим більша ймовірність того, що символ буде закодовано, тим коротшою буде його бітова послідовність.

Створюється дерево, яке містить місце для 288 символів:

0-255: представляють літеральні байти/символи 0-255.

256: кінець блоку - зупинити обробку, якщо це останній блок, інакше почати обробку наступного блоку.

257-285: у поєднанні з додатковими бітами, довжина збігу 3-258 байт.

286, 287: не використовуються, зарезервовані та заборонені, але все ще є частиною дерева.

За кодом довжини збігу завжди слідує код відстані. На основі прочитаного коду відстані можуть бути прочитані додаткові «зайві» біти для отримання остаточної відстані. Дерево відстаней містить місце для 32 символів:

0-3: відстані 1-4

4-5: відстані 5-8, 1 додатковий біт

6-7: відстані 9-16, 2 додаткові біти

8-9: відстані 17-32, 3 додаткові біти

...

26-27: відстані 8,193-16,384, 12 додаткових бітів

28-29: відстані 16 385-32 768, 13 додаткових бітів

30-31: не використовуються, зарезервовані та незаконні, але все ще є частиною дерева.

Пошук у попередньому тексті підрядків, що повторюються, є найбільш обчислювально затратною частиною алгоритму DEFLATE і операцією, на яку впливають налаштування рівня стиснення.

Обидва коди (дерево довжини/літералів з 288 символів і дерево відстаней з 32 символів) кодуються як канонічні коди Хаффмана із зазначенням довжини біта коду для кожного символу. Довжина бітів сама по собі кодується довжиною прогону для отримання якомога компактнішого представлення. Як альтернатива включенню деревовидного представлення, опція «статичне дерево» надає стандартні фіксовані дерева Хаффмана. Розмір стиснення за допомогою статичних дерев можна обчислити, використовуючи ту ж саму статистику (кількість повторень кожного символу), що і для створення динамічних дерев, тому компресор може легко вибрати те, що є меншим.

Deflate64, визначений PKWARE, є пропрієтарним варіантом Deflate. Це принципово той самий алгоритм. Змінилося лише збільшення розміру словника з 32 КБ до 64 КБ, розширення кодів відстані до 16 біт, щоб вони могли адресувати діапазон 64 КБ, і коду довжини, який розширено до 16 біт, щоб він міг визначати довжини від трьох до 65 538 байт. Це призводить до того, що Deflate64 має довший час стиснення і потенційно дещо вищий ступінь стиснення, ніж Deflate. Декілька

вільних та/або відкритих проектів підтримують Deflate64, наприклад, 7-Zip, тоді як інші, наприклад, zlib, не підтримують його через власницьку природу процедури і дуже скромний приріст продуктивності порівняно з Deflate.

Реалізовані кодери на алгоритмі Deflate:

а) PKZIP: перша реалізація, яку було створено Філом Кацем у рамках PKZip;

б) zlib: стандартна еталонна реалізація, прийнята у багатьох програмах завдяки відкритому вихідному коду та дозвільній ліцензії. Див. розділ Zlib § Форки для отримання інформації про високопродуктивні форки;

в) Crypto++: містить загальнодоступну реалізацію на C++, спрямовану на зменшення потенційних вразливостей безпеки. Автор, Вей Дай (Wei Dai), стверджує: «Цей код менш розумний, але, сподіваємось, більш зрозумілий і підтримуваний [ніж zlib]»;

г) 7-Zip: написана Ігорем Павловим на C++, ця версія вільно ліцензована і досягає вищого стиснення, ніж zlib, за рахунок використання процесора. Має можливість використовувати формат зберігання DEFLATE64;

д) PuTTY 'sshzlib.c': окрема реалізація під ліцензією MIT від Simon Tatham, має повну можливість декодування, але підтримує лише створення статичних дерев;

е) libflate: частина Plan 9 від Bell Labs, реалізує стиснення зі здуттям;

є) Hyperbac: використовує власну пропрієтарну бібліотеку стиснення (на C++ та Асемблері) з можливістю реалізації формату зберігання DEFLATE64;

ж) Zopfli: реалізація на C під ліцензією Apache від Google; досягає вищого стиснення за рахунок використання процесора. ZopfliPNG: варіація Zopfli для роботи з файлами у форматі PNG;

з) igzip: кодер, написаний на мові асемблера x86, випущений компанією Intel під ліцензією MIT. У 3 рази швидший за zlib -1. Корисний для стиснення геномних даних;

и) libdeflate: бібліотека для швидкого стиснення та розпакування на основі методу DEFLATE для всього буфера. Libdeflate добре оптимізовано, особливо для процесорів x86.

AdvanceCOMP використовує версії Deflate з вищим ступенем стиснення у 7-Zip, libdeflate та Zopfli, щоб уможливити повторне стиснення файлів gzip, PNG, MNG та ZIP з можливістю отримання файлів меншого розміру, ніж zlib може досягти на максимальних налаштуваннях.

1.6 RLE

Кодування довжин серій (англ. Run-length encoding, RLE) або Кодування повторів - простий алгоритм стиснення даних, який оперує серіями даних, тобто послідовностями, в яких один і той же символ зустрічається кілька разів поспіль. При кодуванні рядок однакових символів, що становлять серію, замінюється рядком, який містить сам повторюваний символ і кількість його повторів.

Схеми кодування довжини відрізка (RLE) використовувалися для передачі аналогових телевізійних сигналів ще у 1967 році. У 1983 році компанія Hitachi запатентувала кодування довжини кадру. RLE особливо добре підходить для растрових зображень на основі палітри (які використовують відносно мало кольорів), таких як комп'ютерні іконки, і було популярним методом стиснення зображень на ранніх онлайн-сервісах, таких як CompuServe, до появи більш складних форматів, таких як GIF. Він не дуже добре працює на зображеннях з безперервним тоном (які використовують дуже багато кольорів), таких як фотографії, хоча JPEG використовує його на коефіцієнтах, які залишаються після перетворення і квантування блоків зображень.

Поширеними форматами для кодованих даних, що виконуються, є Truevision TGA, PackBits (від Apple, використовується в MacPaint), PCX та ILBM. Міжнародний союз електрозв'язку також описує стандарт кодування кольору для факсимільних апаратів, відомий як T.45. Цей стандарт кольорового кодування факсимільних повідомлень, який разом з іншими методами включено до модифікованого кодування Хаффмана[джерело не вказано], є відносно ефективним, оскільки більшість документів, що надсилаються факсом, містять переважно білий простір, зрідка з вкрапленнями чорного.

Процес кодування полягає у перетворенні вхідних даних у стислий формат шляхом ідентифікації та підрахунку послідовних входжень кожного символу. Алгоритм складається з наступних кроків:

а) Ініціалізація порожнього рядка результату;

б) Прохід по вхідним даним;

в) Для кожного символу в даних підрахувати кількість його послідовних входжень;

г) Додати кількість входжень та символ до рядка результату.

Процес декодування полягає у відновленні вихідних даних із закодованого формату шляхом повторення символів відповідно до їх кількості. Алгоритм виглядає наступним чином:

а) Ініціалізація порожнього рядка результату;

б) Прохід по закодованим даним;

в) Для кожної пари символ-лічильник повторення кількості символів;

г) Додавання цих символів до рядка результату.

Розглянемо екран, що містить звичайний чорний текст на суцільному білому тлі. На порожньому просторі буде багато довгих відрізків білих пікселів, а в тексті - багато коротких відрізків чорних пікселів. Гіпотетичний рядок розгортки, де B позначає чорний піксель, а W - білий, може виглядати наступним чином:
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWW
WWWWWWWWWWWWWWBWWWWWWWWWWWWW

За допомогою алгоритму стиснення даних з кодуванням довжини прогону (run-length encoding, RLE), застосованого до наведеного вище гіпотетичного рядка сканування, він може бути відображений наступним чином: 12W1B12W3B24W1B14W.

Це можна інтерпретувати як послідовність з дванадцяти W, однієї B, дванадцяти W, трьох B і т.д., і представляє оригінальні 67 символів лише у 18. Хоча фактичний формат, який використовується для зберігання зображень, як правило, двійковий, а не ASCII-символи, принцип залишається тим самим. Цим методом можна стискати навіть двійкові файли даних; специфікації форматів файлів часто

диктують повторювані байти у файлах як проміжки для заповнення. Однак новіші методи стиснення, такі як DEFLATE, часто використовують алгоритми на основі LZ77, узагальнення кодування довжини прогону, яке може використовувати переваги прогонів рядків символів (наприклад, BWWBWWBWBWWBWW).

Кодування довжини циклу може бути виражено різними способами, щоб врахувати властивості даних, а також додаткові алгоритми стиснення. Наприклад, один з популярних методів кодує довжину циклу лише для циклів з двох або більше символів, використовуючи символ «escape» для ідентифікації циклів або використовуючи сам символ як escape, таким чином, щоразу, коли символ з'являється двічі, він позначає цикл. У попередньому прикладі це дасть наступне: WW12BWW12BB3WW24BWW14

Це можна інтерпретувати як цикл з дванадцяти W, одного B, цикл з дванадцяти W, цикл з трьох B і т.д. У даних, де прогони відбуваються рідше, це може значно покращити ступінь стиснення.

Інша справа - застосування додаткових алгоритмів стиснення. Навіть після вилучення прогонів частота різних символів може бути великою, що дозволяє проводити подальше стиснення; однак, якщо довжини прогонів записані у файлі в тих місцях, де відбувалися прогони, присутність цих чисел перериває нормальний потік і ускладнює стиснення. Щоб подолати цю проблему, деякі кодувальники довжини прогонів відокремлюють дані та символи екранування від довжини прогонів, щоб їх можна було обробляти незалежно. Для наведених у прикладі даних це призведе до отримання двох виходів: рядка «WWBWWBBWWBWW» і чисел (12,12,3,24,14).

РОЗДІЛ 2. ПРОГРАМИ СТВОРЕНІ НА БАЗІ АЛГОРИТМІВ СТИСНЕННЯ ДАНИХ

2.1 UNIX Compress

Алгоритм LZW, що використовується в компресії, був запатентований дослідницьким центром Сперрі в 1983 році, а згодом Террі Велч опублікував статтю про алгоритм в IEEE в 1984 році, але не зазначив, що він подав заявку на патент на алгоритм. Спенсер Томас з Університету штату Юта взяв цю статтю і реалізував стиснення у 1984 році, не знаючи, що на алгоритм LZW подано заявку на патент компанією Unisys. Формат зображень GIF також використовує LZW-стиснення, і пізніше компанія Unisys вимагала ліцензійний платіж за реалізацію GIF.

Пізніше Джозеф М. Орост очолив команду і разом з Спенсером Томасом та іншими і створив «остаточну» версію (4.0) стиснення і опублікував її як вільне програмне забезпечення в групі USENET «net.sources» у 1985 році.

Патент США 4,558,302 було видано у 1985 році, і саме тому compress не можна було використовувати без сплати ліцензійного платежу компанії Unisys.

compress недолюблювали певні групи користувачів, оскільки він використовує алгоритм LZW, який був захищений патентом Unisys - через це утиліти gzip і bzip2 стали популярнішими в операційних системах на основі Linux завдяки своїм альтернативним алгоритмам, а також кращому стисненню файлів. Однак compress зберіг свою присутність в системах Unix і BSD, а команди compress і uncompress також були перенесені в операційну систему IBM і.

Compress - програма для стиснення у командній оболонці Unix, заснована на алгоритмі стиснення LZW. Порівняно з найшвидшим варіантом gzip, compress працює трохи повільніше при стисненні, трохи швидше при розпакуванні і має значно нижчий ступінь стиснення. Для стиснення даних Премії Хаттера (Премія Хаттера - це грошова премія, що фінансується Маркусом Хаттером і

нагороджується за покращення стиснення даних у конкретному текстовому файлі розміром 1 Гб англійською мовою з метою заохочення досліджень у галузі штучного інтелекту) використовується 1,8 МБ пам'яті, що трохи більше, ніж при найповільнішому використанні gzip.

Утиліта uncompress відновить файли до початкового стану після того, як вони були стиснуті за допомогою утиліти compress. Якщо не вказано жодних файлів, стандартний ввід буде розпаковано до стандартного виводу.

У наступній версії специфікацій POSIX та Single Unix планується додати підтримку алгоритму DEFLATE (алгоритм який працює за допомогою комбінації алгоритму Хаффмана та LZ77), який використовується у форматі gzip, у цих утилітах.

Файли, стиснуті за допомогою compress, зазвичай отримують розширення «.Z» (за зразком попередньої програми pack, яка використовувала розширення «.z»). Більшість tar-програм передаватимуть свої дані за допомогою compress, якщо у командному рядку вказати параметр «-Z». Програма tar сама по собі не стискає а просто зберігає декілька файлів у одному стрічковому архіві.

Файли можна повернути до початкового стану за допомогою decompress. Звичайною дією розпакування є не лише створення нестиснутої копії файлу, але й відновлення мітки часу та інших атрибутів стиснутого файлу.

Для файлів, створених за допомогою compress в інших системах, decompress підтримує стиснення від 9 до 16 біт.

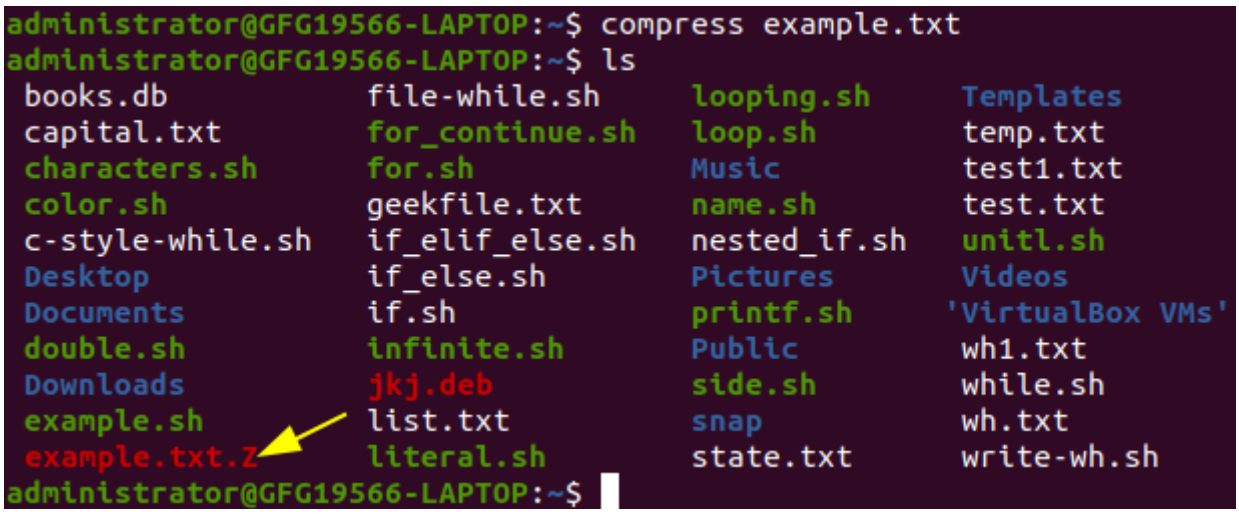


Рисунок 2.1 Виконання утиліти compress в терміналі Linux

З рисунку 2.1 видно що файл після виконання отримав розширення example.xls.Z що вказує на те що файл було заархівовано.

2.2 GZIP

gzip - це формат файлів і програмне забезпечення, що використовується для стиснення і розпакування файлів. Програма була створена Жаном-Лу Гейлі та Марком Адлером як безкоштовна заміна програмі compress, що використовувалася в ранніх системах Unix, і призначалася для використання GNU (це велика колекція безкоштовних програм (385 пакетів станом на вересень 2023 року), які можна використовувати як операційну систему або використовувати частинами з іншими операційними системами) звідки і походить літера «g» в gzip. Версію 0.1 було вперше публічно випущено 31 жовтня 1992 року, а версію 1.0 - у лютому 1993 року.

Розпакування формату gzip може бути реалізовано як потоковий алгоритм, що є важливою функцією для веб-протоколів, обміну даними та додатків ETL (у стандартних трубах).

gzip базується на алгоритмі DEFLATE, який є комбінацією LZ77 та кодування Хаффмана. DEFLATE був розроблений як заміна LZW та інших алгоритмів стиснення даних, обтяжених патентами, які на той час обмежували можливості використання утиліти compress та інших популярних архіваторів.

«gzip» часто також використовується для позначення формату файлів gzip, тобто:

а) 10-байтовий заголовок, що містить магічне число (1f 8b), метод стиснення (08 для DEFLATE), 1 байт прапорів заголовка, 4-байтну мітку часу, прапори стиснення та ідентифікатор операційної системи;

б) необов'язкові додаткові заголовки, дозволені прапорами заголовків, зокрема оригінальне ім'я файлу, поле коментаря, поле «extra» та нижня половина контрольної суми CRC-32 для секції заголовка.;

в) тіло, що містить стиснене до рівня DEFLATE корисне навантаження;

г) 8-байтовий трейлер, що містить контрольну суму CRC-32 і довжину вихідних нестиснутих даних за модулем 232.

gzip можна комбінувати з програмою tar для стиснення декількох файлів. Хоча формат gzip також дозволяє об'єднувати декілька таких потоків (gzip-файли просто розпаковуються об'єднаними так, ніби вони були одним файлом), gzip зазвичай використовується для стиснення лише одного файлу. Стиснуті архіви зазвичай створюються шляхом збирання колекцій файлів в один tar-архів (також званий tarball), а потім стискання цього архіву за допомогою gzip. Кінцевий стиснутий файл зазвичай має розширення .tar.gz або .tgz.

gzip не слід плутати з форматом архіву ZIP, який також використовує DEFLATE. Формат ZIP дозволяє зберігати колекції файлів без зовнішнього архіватора, але є менш компактним, ніж стиснуті tar-архіви з тими самими даними, оскільки він стискає файли окремо і не може скористатися перевагами надлишковості між файлами (суцільне стиснення). Формат файлів gzip також не слід плутати з форматом файлів утиліти compress, заснованої на LZW, з розширенням .Z; однак утиліта gunzip здатна розпаковувати файли .Z.

Було написано різні реалізації програми. Найбільш відомою є реалізація проекту GNU з використанням кодування Lempel-Ziv (LZ77). Версія gzip для OpenBSD - це, власне, програма стиснення, до якої було додано підтримку формату gzip у OpenBSD 3.4. Літера «g» у цій конкретній версії означає gratis. У FreeBSD, DragonFly BSD і NetBSD замість версії GNU використовується ліцензована BSD реалізація; насправді це інтерфейс командного рядка для zlib, призначений для сумісності з опціями реалізацій GNU. Ці реалізації походять з NetBSD і підтримують розпакування bzip2 та формат пакунків Unix.

Альтернативною програмою стиснення, яка забезпечує на 3-8% краще стиснення, є Zopfli. Вона досягає gzip-сумісного стиснення за допомогою більш вичерпних алгоритмів, але за рахунок збільшення часу стиснення. Це не впливає на час розпакування.

pigz, написаний Марком Адлером, сумісний з gzip і прискорює стиснення, використовуючи всі доступні ядра і потоки процесора.

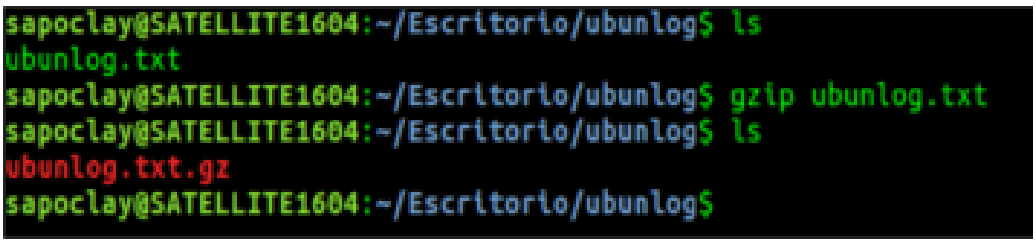


Рисунок 2.2 Архівування файлу за допомогою утиліти gzip

Рисунок 2.3 Робота утиліти gzip з розархівування файлу

2.3 PKZIP

PKZIP - це комп'ютерна програма для архівації файлів, яка відома тим, що запровадила популярний формат ZIP. Вперше PKZIP було представлено для MS-DOS на IBM-PC сумісній платформі у 1989 році. З того часу були випущені версії для ряду інших архітектур і операційних систем. Спочатку PKZIP був написаний Філом Катцем і продавався його компанією PKWARE, Inc, починаючи з 1986 року. Компанія носить його ініціали: «PK».

До 1970-х років програми для архівації файлів поширювалися як стандартні утиліти з операційними системами. До них належать утиліти для Unix - ar, shar і tar. Ці утиліти були розроблені для того, щоб зібрати кілька окремих файлів в один архівний файл для полегшення копіювання та розповсюдження. За бажанням ці архіви можна було пропустити через утиліту-компресор потоку, наприклад, compress та інші.

У 1980-х роках з'явилися й інші архіватори, зокрема ARC від System Enhancement Associates, Inc. (SEA), ZOO Рахула Десі, DWC Діна В. Купера, LHarc Харухіко Окомури і Харуясу Йошизакі та ARJ, що розшифровується як «Archived by Robert Jung».

Вперше про розробку PKZIP було оголошено у файлі SOFTDEV.DOC з пакунку PKPAK 3.61, де йшлося про розробку нової, ще неназваної програми стиснення. Оголошення було зроблено після судового процесу між SEA та PKWARE, Inc. Хоча SEA виграла позов, вона програла війну за стиснення, оскільки користувачі перейшли на PKZIP як на кращий компресор. На чолі з деякими системними адміністраторами BBS, які відмовлялися приймати або пропонувати файли, стиснуті у форматі .ARC, користувачі почали перепаковувати всі старі архіви, які зберігалися у форматі .ARC, у файли .ZIP.

Перша версія була випущена в 1989 році як інструмент командного рядка DOS, що розповсюджувався за моделлю умовно-безкоштовного програмного забезпечення з реєстраційним внеском 25 доларів США (47 доларів США з інструкцією).

Щоб забезпечити сумісність формату ZIP, Філ Кац опублікував оригінальну специфікацію формату .ZIP у файлі документації APPNOTE.TXT. PKWARE продовжує підтримувати цей документ і періодично публікує його оновлення. Спочатку специфікація постачалася лише з зареєстрованими версіями PKZIP, але згодом вона стала доступною на сайті PKWARE.

Специфікація має власний номер версії, який не обов'язково відповідає номерам версій PKZIP, особливо для PKZIP 6 або пізніших версій. У різний час PKWARE додає попередні функції, які дозволяють продуктам PKZIP розпаковувати архіви з використанням розширених можливостей, але продукти PKZIP, які створюють такі архіви, не будуть доступні до наступного великого випуску.

Хоча свого часу ZIP-архіви, що використовують методи стиснення PKZIP 1.0, були популярними, зараз вони рідкісні, і багато утиліт для розархівування, наприклад, 7-Zip, здатні читати і записувати деякі інші формати архівів.

2.4 bzip2

bzip2 - це безкоштовна програма для стиснення файлів з відкритим вихідним кодом, яка використовує алгоритм Берроуза-Вілера. Вона стискає лише окремі файли і не є архіватором файлів. Вона покладається на окремі зовнішні утиліти для виконання таких завдань, як робота з кількома файлами, шифрування та розбиття архіву на частини.

bzip2 був вперше випущений у 1996 році Джуліаном Сьюардом. Він стискає більшість файлів ефективніше, ніж старіші алгоритми стиснення LZW і Deflate, але працює повільніше. bzip2 особливо ефективний для текстових даних, а розпакування відбувається відносно швидко. Алгоритм використовує кілька рівнів методів стиснення, таких як кодування довжини виконання (RLE), перетворення Берроуза-Вілера (BWT), перетворення зсуву до початку (MTF) і кодування Хаффмана. bzip2 стискає дані блоками розміром від 100 до 900 кБ і використовує перетворення Берроуза-Вілера для перетворення послідовностей символів, що часто повторюються, у рядки з однакових літер. Потім застосовується перетворення зсуву до початку та кодування Хаффмана. Ефективність стиснення є асиметричною, причому декомпресія відбувається швидше, ніж стиснення.

Алгоритм пройшов через кілька супровідників з моменту його початкового випуску, з червня 2021 року супровідником є Міка Снайдер (Micah Snyder). Алгоритм зазнав деяких модифікацій, таких як pbzip2, який використовує багатопотоковість для підвищення швидкості стиснення на комп'ютерах з декількома процесорами і багатоядерних комп'ютерах.

bzip2 підходить для використання в додатках для роботи з великими даними з кластерними обчислювальними фреймворками, такими як Hadoop і Apache Spark, оскільки стиснутий блок може бути розпакований без необхідності обробки попередніх блоків.

Звичайний процес стиснення і розпакування з використанням bzip2 включає наступні кроки.

Стиснення:

а) Блочне сортування Хаффмана (BWT):

1) Ділимо вхідні дані на блоки;

2) Кожен блок перетворюється за допомогою BWT, щоб змінити порядок символів.

б) Перетворення зсуву до початку (MTF):

1) Кодуємо кожний блок, використовуючи MTF, що замінює символи на їх індекс у таблиці символів.

в) RLE кодування довжини прогону (Run-Length Encoding):

1) Застосовуємо RLE до результатів MTF, щоб стиснути послідовності однакових символів.

г) Кодування Хаффмана:

1) Кодуємо результати RLE, використовуючи таблиці Хаффмана.

д) Запис додаткової інформації:

1) Записуємо додаткову інформацію, таку як вибір таблиць Хаффмана, унарне кодування тощо.

Розпакування:

а) Відновлення додаткової інформації:

1) Розпаковуємо додаткову інформацію, яка включає вибір таблиць Хаффмана.

б) Розпакування Хаффмана:

1) Розпаковуємо дані, використовуючи відповідні таблиці Хаффмана.

в) RLE декодування довжини прогону:

1) Розпаковуємо результати Хаффмана, використовуючи RLE.

г) Відновлення зсуву до початку (MTF):

1) Відновлюємо дані, використовуючи MTF.

д) Відновлення блочного сортування Хаффмана (BWT):

1) Відновлюємо початковий порядок символів у кожному блоку.

Цей процес дозволяє стиснути та розпакувати дані з використанням методів стиснення, які оптимально пристосовані для різноманітних типів даних.

bzip2 використовує декілька рівнів стиснення, які накладаються один на одного, у наступному порядку під час стиснення і у зворотному порядку під час розпакування:

а) кодування довжини циклу (RLE) початкових даних;

б) перетворення Берроуза-Вілера (BWT), або блокове сортування;

в) перетворення зсуву до початку (MTF);

г) кодування довжини прогону (RLE) результату MTF;

д) кодування Хаффмана;

е) вибір між декількома таблицями Хаффмана;

ґ) унарне кодування вибору таблиці Хаффмана з основою 1;

і) дельта-кодування (Δ) довжини бітів коду Хаффмана;

ї) розріджений бітовий масив, що показує, які символи використовуються.

Будь-яка послідовність від 4 до 255 послідовних символів, що повторюються, замінюється першими 4 символами і довжиною повтору від 0 до 251. Таким чином, послідовність AAAAAAABBBCCCD замінюється на AAAA\3BBBB\0CCCD, де \3 і \0 позначають значення байтів 3 і 0 відповідно. Послідовності символів завжди перетворюються після 4-х послідовних символів, навіть якщо довжина послідовності дорівнює нулю, щоб зробити перетворення оборотним.

У гіршому випадку це може призвести до розширення на 1.25, а у кращому - до зменшення до <0.02. Хоча специфікація теоретично дозволяє кодувати рядки довжиною 256-259, еталонний кодер не дасть такого результату.

Автор bzip2 заявив, що крок RLE був історичною помилкою і мав на меті лише захистити оригінальну реалізацію BWT від патологічних випадків.

Перетворення Берроуза-Вілера - це оборотне сортування блоків, яке лежить в основі bzip2. Блок є повністю автономним, вхідний та вихідний буфери залишаються однакового розміру - у bzip2 робоча межа для цього етапу становить 900 кБ. Для сортування блоку створюється (умовна) матриця, в якій рядок і містить весь буфер, повернутий так, щоб почати з і-го символу. Після обертання рядки матриці сортуються в алфавітному (числовому) порядку. Зберігається 24-бітний вказівник, що позначає початкову позицію для того, щоб блок не був перетворений.

На практиці немає необхідності будувати повну матрицю; замість цього сортування виконується з використанням вказівників для кожної позиції в буфері. Вихідним буфером є останній стовпчик матриці; він містить весь буфер, але впорядкований таким чином, що він, ймовірно, містить великі пробіги однакових символів.

Перетворення зсуву на початок знову не змінює розмір оброблюваного блоку. Кожен символ, що використовується у документі, розміщується у масиві. Коли символ обробляється, він замінюється на його місцезнаходження (індекс) у масиві, і цей символ переміщується на початок масиву. Це призводить до того, що символи, які часто повторюються, замінюються нульовими символами (довгі рядки будь-якого довільного символу стають рядками нульових символів), тоді як інші символи переставляються відповідно до їхньої локальної частоти.

Багато «природних» даних містять однакові символи, які повторюються в обмеженому діапазоні (текст - гарний приклад). Оскільки перетворення MTF присвоює низькі значення символам, які часто повторюються, це призводить до того, що потік даних містить багато символів у діапазоні низьких цілих чисел, багато з яких є ідентичними (різні вхідні символи, що повторюються, можуть фактично відображатися в один і той самий вихідний символ). Такі дані можуть бути дуже ефективно закодовані будь-яким старим методом стиснення.

Довгі рядки нулів на виході перетворення зсуву вперед (які походять від повторюваних символів на виході BWT) замінюються послідовністю двох спеціальних кодів, RUNA і RUNB, які представляють довжину прогону у вигляді двійкового числа. Фактичні нулі ніколи не кодуються на виході; одиночний нуль стає RUNA. (Цей крок фактично виконується одночасно з MTF; щоразу, коли MTF видає нуль, він збільшує лічильник, щоб потім кодувати RUNA та RUNB).

Довгі рядки нулів на виході перетворення зсуву вперед (які походять від повторюваних символів на виході BWT) замінюються послідовністю двох спеціальних кодів, RUNA і RUNB, які представляють довжину прогону у вигляді двійкового числа. Фактичні нулі ніколи не кодуються на виході; одиночний нуль стає RUNA. (Цей крок фактично виконується одночасно з MTF; щоразу, коли MTF видає нуль, він збільшує лічильник, щоб потім кодувати RUNA та RUNB).

Послідовність 0, 0, 0, 0, 0, 0, 1 буде представлена як RUNA, RUNB, 1; RUNA, RUNB представляє значення 5, як описано нижче. Виконання коду завершується при досягненні іншого нормального символу. Цей процес RLE є більш гнучким, ніж початковий крок RLE, оскільки він може кодувати довільні цілі числа (на практиці це зазвичай обмежується розміром блоку, так що на цьому кроці не кодується прогін довжиною більше 900000 байт). Довжина циклу кодується таким чином: присвоюючи першому біту значення 1, другому - 2, третьому - 4 і т.д. у послідовності, множимо кожне значення у місці RUNB на 2 і додаємо всі отримані значення (як для RUNA, так і для RUNB) разом. Це схоже на двійкову нумерацію з основою 2. Таким чином, послідовність RUNA, RUNB дає значення $(1 + 2 \times 2) = 5$.

Цей процес замінює символи фіксованої довжини в діапазоні 0-258 на коди змінної довжини залежно від частоти використання. Більш часто використовувані коди стають коротшими (2-3 біти), в той час як рідкісні коди можуть мати довжину до 20 біт. Коди підбираються ретельно, щоб жодну послідовність бітів не можна було переплутати з іншим кодом.

Особливо цікавим є код кінця потоку. Якщо в нестиснених даних використовується n різних байтів (символів), то код Хаффмана складатиметься з двох RLE-кодів (RUNA і RUNB), n - 1 символьних кодів і одного коду кінця потоку. Завдяки комбінованому результату кодування БЧХ і RLE на попередніх двох кроках, ніколи немає необхідності явно посилатися на перший символ у таблиці БЧХ (у звичайному БЧХ він був би нульовим), таким чином зберігаючи один символ для маркера кінця потоку (і пояснюючи, чому в дереві Хаффмана кодується тільки n - 1 символ). У крайньому випадку, коли в нестиснених даних використовується лише один символ, у дереві Хаффмана взагалі не буде кодів символів, а весь блок складатиметься з RUNA і RUNB (неявно повторюючи один байт) і маркера кінця потоку зі значенням 2.

0: RUNA,

1: RUNB,

2–257: byte values 0–255,

258: end of stream, finish processing (could be as low as 2).

Кілька однакових за розміром таблиць Хаффмана можна використовувати з блоком, якщо вигода від їх використання перевищує витрати на включення додаткової таблиці. Може бути присутнім щонайменше 2 і до 6 таблиць, причому найбільш підходяща таблиця вибирається повторно перед обробкою кожних 50 символів. Перевага цього методу полягає в тому, що він забезпечує дуже чутливу динаміку Хаффмана без необхідності постійно додавати нові таблиці, як це було б потрібно в методі DEFLATE. Кодування довжини прогону на попередньому кроці призначене для того, щоб подбати про коди, зворотна ймовірність використання яких вища, ніж у найкоротшого коду Хаффмана, що використовується.

Якщо використовується декілька таблиць Хаффмана, вибір кожної таблиці (пронумерованої від 0 до 5) здійснюється зі списку за допомогою біта з нульовим закінченням довжиною від 1 до 6 біт. Вибір відбувається у список таблиць MTF. Використання цієї функції призводить до максимального розширення близько 1.015, але зазвичай менше. Це розширення, ймовірно, значно перекривається перевагою вибору більш відповідних таблиць Хаффмана, а загальний випадок продовження використання тієї самої таблиці Хаффмана представлено у вигляді одного біта. Замість унарного кодування, фактично це екстремальна форма дерева Хаффмана, де кожен код має половину ймовірності попереднього коду.

Довжина бітів коду Хаффмана необхідна для реконструкції кожної з використовуваних канонічних таблиць Хаффмана. Кожна довжина біта зберігається як закодована різниця з довжиною біта попереднього коду. Нульовий біт (0) означає, що попередня довжина біта повинна бути продубльована для поточного коду, тоді як одиничний біт (1) означає, що наступний біт повинен бути прочитаний і довжина біта збільшена або зменшена на основі цього значення. У загальному випадку використовується один біт на символ у таблиці, і в найгіршому випадку - перехід від довжини 1 до довжини 20 - потребуватиме приблизно 37 бітів. В результаті більш раннього кодування MTF довжина коду починалася з 2-3 біт (дуже часто використовувані коди) і поступово збільшувалася, що означає, що дельта-формат є досить ефективним, вимагаючи близько 300 біт (38 байт) на повну таблицю Хаффмана.

Бітове зображення використовується для того, щоб показати, які символи використовуються всередині блоку і повинні бути включені в дерева Хаффмана. Двійкові дані, швидше за все, використовують всі 256 символів, які можна представити в байті, тоді як текстові дані можуть використовувати лише невелику підмножину доступних значень, можливо, охоплюючи діапазон ASCII між 32 і 126. Зберігання 256 нульових бітів було б неефективним, якби вони здебільшого не використовувались. Використовується розріджений метод: 256 символів розбиваються на 16 діапазонів, і тільки якщо символи використовуються в межах цього блоку, включається 16-бітний масив. Наявність кожного з цих 16 діапазонів позначається додатковим 16-бітним бітовим масивом спереду. Загалом бітове зображення використовує від 32 до 272 біт пам'яті (4-34 байти). Для контрасту, алгоритм DEFLATE покаже відсутність символів, кодуючи їх як такі, що мають нульову бітову довжину з кодуванням довжини прогону і додатковим кодуванням Хаффмана.

Рисунок 2.4 Архівування файлу за допомогою утиліти bzip2

РОЗДІЛ 3. ПРОСТА ПРОГРМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ LZW ВИКОРИСТОВУЮЧИ МОВУ ПРОГРАМУВАННЯ PYTHON

3.1 Алгоритм LZW, опис та виконання коду

```
import time

start_time = time.time()

def compress(message):
    dictionary = {chr(i): i for i in range(256)}
    current_code = 256
    result = []
    current_string = ""

    for char in message:
        new_string = current_string + char
        if new_string in dictionary:
            current_string = new_string
        else:
            result.append(dictionary[current_string])
            dictionary[new_string] = current_code
            current_code += 1
            current_string = char

    if current_string in dictionary:
        result.append(dictionary[current_string])
```

```
    return result

def decompress(compressed):
    dictionary = {i: chr(i) for i in range(256)}
    current_code = 256
    result = []
    previous_code = compressed[0]
    previous_string = dictionary[previous_code]
    result.append(previous_string)

    for code in compressed[1:]:
        if code in dictionary:
            entry = dictionary[code]
        elif code == current_code:
            entry = previous_string + previous_string[0]
        else:
            raise ValueError("Bad compressed code")

        result.append(entry)

        dictionary[current_code] = previous_string + entry[0]
        current_code += 1
        previous_string = entry

    return ''.join(result)

# Приклад використання:
input_message = input("Enter message: ")

compressed_message = compress(input_message)
```

```
decompressed_message = decompress(compressed_message)

end_time = time.time()

print("Compressed:", compressed_message)
print("Decompressed:", decompressed_message)
print("Total execution time:", end_time - start_time, "seconds")
```

Даний код наглядно показує реалізацію алгоритму LZW та виконання архівування та розархівування вхідного повідомлення.

Функція compress приймає вхідне повідомлення і стискайте його за алгоритмом LZW. Як вже було описано в першому розділі для початку роботи ініціалізується словник де кожному символу ASCII від 0 до 255 відповідає його код і вписується у вихідний архів. В коді цю операцію реалізує команда dictionary = {chr(i): i for i in range(256)}. Після ініціалізації словнику ми вказуємо наступний доступний код для нового слова в словнику використовуючи команду current_code = 256

Команда result = [] це список кодів стиснутого повідомлення які ми отримаємо після стискання повідомлення. Далі переходимо до наступного символу який потрібно обробити командою current_string = "".

Далі ініціалізуємо функцію for char in message. За допомогою неї проходимося по кожному символу вхідного повідомлення. Далі формуємо новий рядок, додавши поточний символ до поточного рядку. Після додавання рядку перевіряємо чи є новий рядок в словнику.

Додаємо умову if new_string in dictionary та else if. Якщо є в словнику, оновлюємо поточний рядок, а якщо ні додаємо код поточного рядку до результату, а потім додаємо новий рядок до словнику з наступний доступним кодом. Після занесення нового рядку в словник збільшуємо поточний доступний код на один, так як поточний код вже присвоєно новому рядку в словнику, виконуємо це командою

```
dictionary[new_string] = current_code
```

current_code += 1

current_string = char

Додаємо останній код поточного рядка до результату і повертаємо список кодів стиснутого повідомлення.

Процес декодування відбувається зворотнім шляхом, але для нього також ініціалізується словник де кожному коду від 0 до 255 відповідає символ ASCII.

Так як код реалізований одразу для компресії та декомпресії, щоб код був більш адаптивний в нього додана можливість ввести вхідний алфавіт користувачем. Для прикладу, повідомлення: «fhesdfghrstghjrsgrjnedtyjdfgbhsrtysro;iujghaediujfkgbn;slekrthnfgblaernmgeoabrvbvoi qenroignv;azeolirthnbslkna;oierb». Після його опрацювання кодом, ми отримали зашифрований код який виглядає так: «[102, 104, 101, 115, 100, 102, 103, 104, 114, 115, 116, 262, 106, 264, 103, 114, 106, 110, 101, 100, 116, 121, 106, 260, 103, 98, 104, 115, 114, 276, 283, 111, 59, 105, 117, 106, 262, 97, 274, 289, 106, 102, 107, 280, 110, 59, 115, 108, 101, 107, 284, 104, 110, 261, 98, 108, 293, 114, 110, 109, 103, 101, 111, 97, 98, 114, 118, 98, 118, 111, 105, 113, 101, 110, 114, 325, 103, 110, 118, 59, 97, 122, 317, 108, 105, 306, 110, 98, 302, 107, 110, 97, 59, 325, 101, 114, 98]»

Нижче на рисунку 3.1 схематично показано робота всього коду.

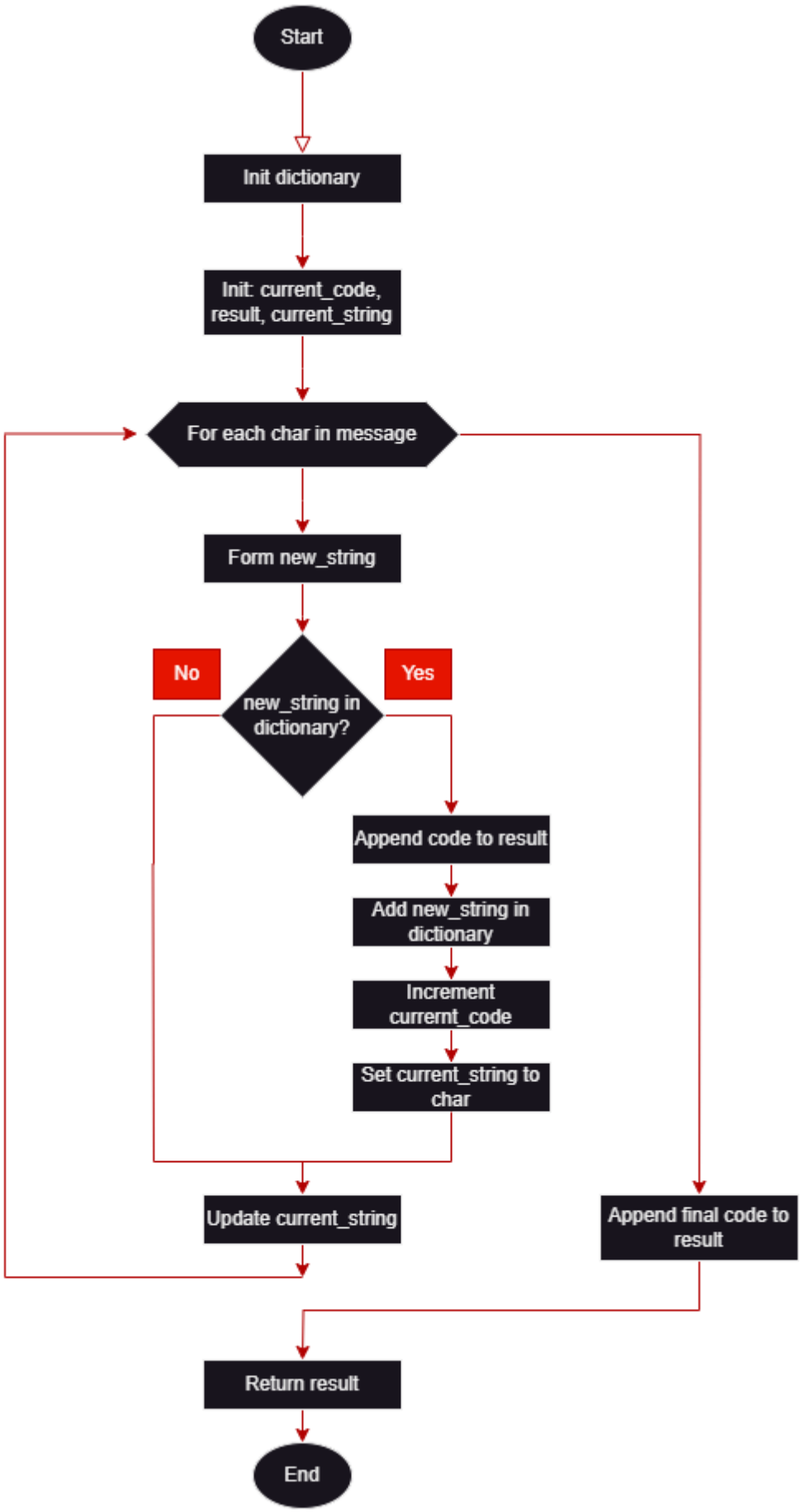


Рисунок 3.1 Блок схема виконання алгоритму

3.2 Оптимізація створеного алгоритму

```
import time

start_time = time.time()

# Оптимізовані функції compress і decompress

def compress(message):
    dictionary = {chr(i): i for i in range(256)}
    current_code = 256
    result = []
    current_string = ""
    max_dict_size = 4096

    for char in message:
        new_string = current_string + char
        if new_string in dictionary:
            current_string = new_string
        else:
            result.append(dictionary[current_string])
            if len(dictionary) < max_dict_size:
                dictionary[new_string] = current_code
                current_code += 1
            current_string = char

    if current_string in dictionary:
        result.append(dictionary[current_string])

    return result
```

```
def decompress(compressed):
    dictionary = {i: chr(i) for i in range(256)}
    current_code = 256
    result = []
    previous_code = compressed[0]
    previous_string = dictionary[previous_code]
    result.append(previous_string)
    max_dict_size = 4096

    for code in compressed[1:]:
        if code in dictionary:
            entry = dictionary[code]
        elif code == current_code:
            entry = previous_string + previous_string[0]
        else:
            raise ValueError("Bad compressed code")

        result.append(entry)

        if len(dictionary) < max_dict_size:
            dictionary[current_code] = previous_string + entry[0]
            current_code += 1

        previous_string = entry

    return ''.join(result)

# Приклад використання
input_message = input("Enter message: ")
```

```
compressed_message = compress(input_message)
decompressed_message = decompress(compressed_message)

end_time = time.time()

print("Compressed:", compressed_message)
print("Decompressed:", decompressed_message)
print("Total execution time:", end_time - start_time, "seconds")
```

Оптимізація коду LZW включає кілька покращень для підвищення ефективності та обмеження розміру словника. Ось детальний опис змін:

а) Ініціалізація словника та змінних;

1) Додана змінна max_dict_size, що встановлює максимальний розмір словника (4096 записів);

б) Основний цикл (функція compress);

1) Заміна перевірки наявності рядка в словнику та додавання нового рядка;

Перевірка, чи новий рядок new_string є в словнику. Якщо new_string є в словнику, оновлюється current_string.

Якщо new_string немає в словнику: Додається код current_string до списку результатів result. Перевіряється розмір словника: якщо розмір менший за max_dict_size, додається новий рядок до словника, і current_code збільшується на 1. Якщо розмір словника досяг max_dict_size, новий рядок не додається. current_string встановлюється на char.

в) Основний цикл (функція decompress);

1) Заміна перевірки наявності коду в словнику та додавання нового рядка;

Для кожного коду code у стисненому повідомленні: Якщо code є у словнику, entry встановлюється на рядок, який відповідає цьому коду. Якщо code дорівнює

current_code, entry встановлюється на previous_string + перший символ previous_string. В іншому випадку виникає помилка. Додається entry до результату. Перевіряється розмір словника: якщо розмір менший за max_dict_size, додається новий запис до словника з ключем current_code та значенням previous_string + перший символ entry, і current_code збільшується на 1. previous_string встановлюється на entry.

г) Вимірювання часу виконання коду - додано заміри часу до початку та кінця виконання скрипту, щоб визначити загальний час виконання.

Для перевірки оптимізації коду можемо порівняти час виконання оптимізованого коду та неоптимізованого (Рисунки 3.2 та 3.3).

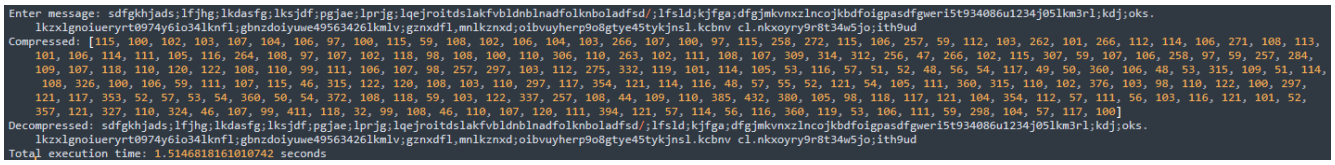


Рисунок 3.2 Час виконання неоптимізованого коду

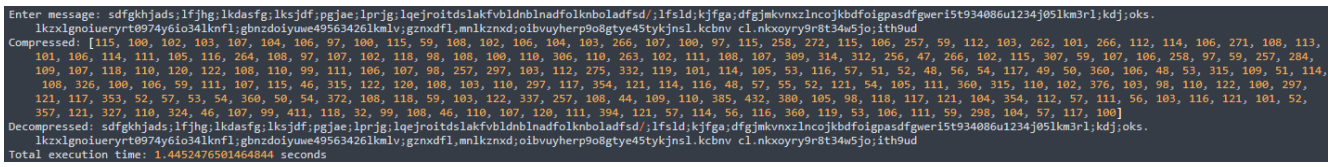


Рисунок 3.3 Час виконання оптимізованого коду

Час виконання обох кодів майже однаковий але невелика різниця все ж є: оптимізований код – 1.44, а неоптимізований – 1.51.Це тому що дані які потрібно було шифрувати були представлені в малому обсязі. Враховуючи що наш словник обмежений розміром у 12біт, при його переповненні існує декілька варіантів продовження виконання алгоритму, окрім того який вказаний в представленому алгоримті:

а) Ігнорування нових кодів - алгоритм може продовжувати працювати, але ігнорувати створення нових кодів, коли словник вже максимально заповнений. Це може призвести до зниження ступеня стиснення, але зберігає працездатність алгоритму;

б) Використання більшого ліміту - збільшення ліміту на розмір словника може допомогти зберегти більше унікальних символів без очищення словника. Проте це може призвести до збільшення використання пам'яті;

в) Самостійне видалення менш використаних записів - алгоритм може видаляти менш використані записи зі словника, щоб звільнити місце для нових. Це може зменшити кількість доступних кодів, але зберегти стиснення;

г) Динамічне зменшення ліміту - ліміт може бути зменшений динамічно в залежності від контексту, наприклад, на основі статистики використання словника. Це може допомогти у збереженні ефективності алгоритму при мінімізації використання пам'яті.

Для подальшої оптимізації можемо додати до коду можливість читання будь-яких символів, а не тільки таблицю ASCII, тобто використати UNICODE. Крім цього можна ще:

а) Використання хеш-таблиці для словника;

Замість використання словника Python, можна застосувати спеціалізовану хеш-таблицю, яка забезпечить більш швидкий доступ до елементів.

б) Динамічне збільшення розміру коду;

Початково використовувати менший розмір коду, а потім динамічно збільшувати його при необхідності.

в) Використання побітових операцій;

Для стиснення та декомпресії можна використовувати побітові операції, щоб зменшити кількість операцій введення/виведення та оптимізувати роботу з пам'яттю.

г) Застосування багатопотоковості;

Для великих обсягів даних можна використовувати багатопотокову обробку, щоб паралельно стиснути різні частини даних.

д) Використання індексів замість рядків;

Замість зберігання рядків у словнику можна зберігати індекси, що зменшить обсяг пам'яті, що використовується.

е) Попереднє аналізування даних.

Попередній аналіз даних може допомогти виявити повторювані шаблони та ефективніше використовувати словник.

Алгоритм виконання оптимізованого коду проілюстровано на рисунку 3.4.

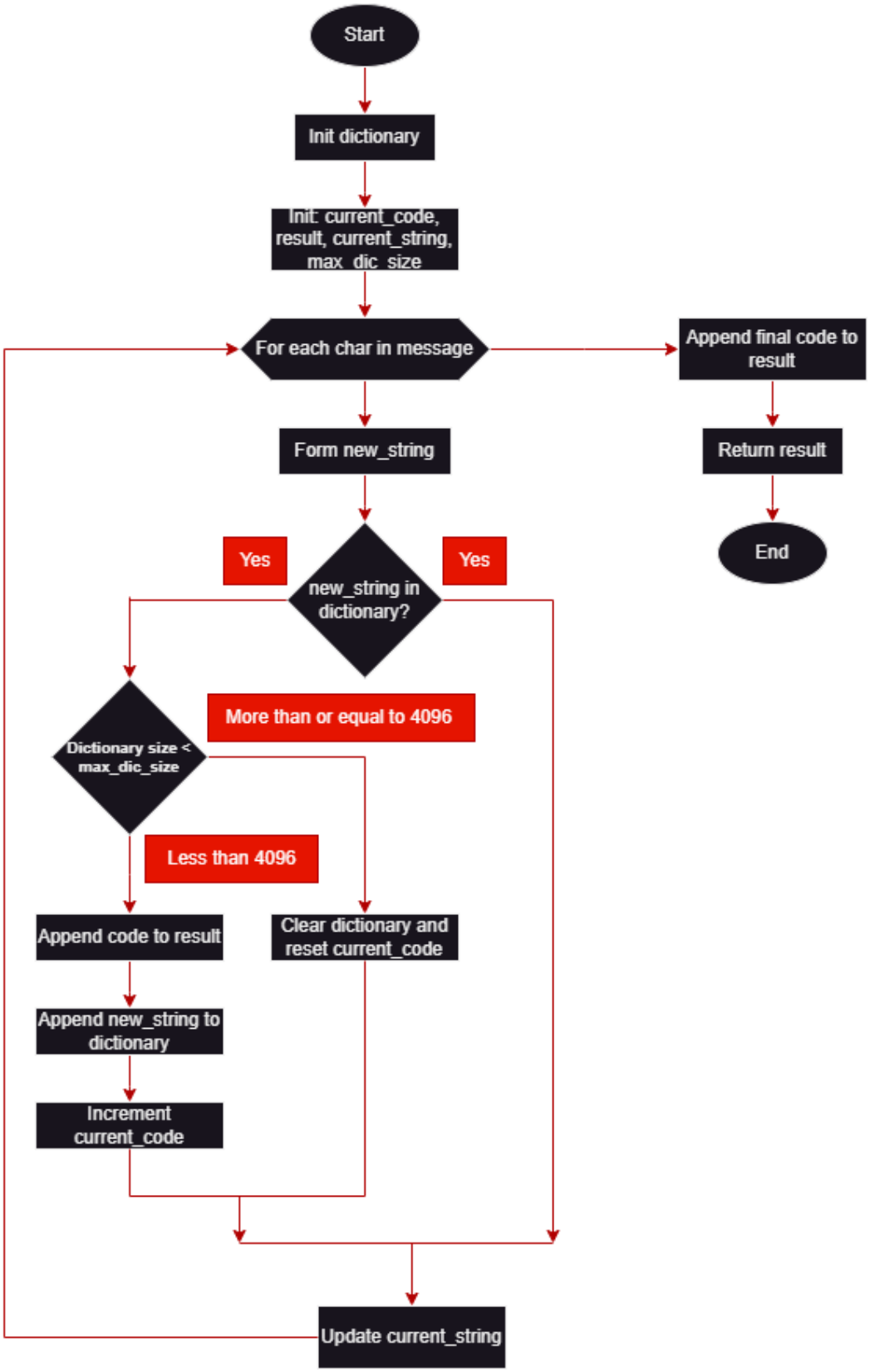


Рисунок 3.4 Блок схема виконання оптимізованого алгоритму

ВИСНОВОК

Ретельно вивчивши тему моєї дипломної роботи, я прослідкував історичний шлях розвитку алгоритмів стиснення даних, а також їхні модифікації та принципи роботи. Цей глибокий аналіз дозволив мені отримати глибше розуміння важливості операції архівування для ефективної передачі файлів в мережі. Крім того, я досліджував сучасні тенденції в цій галузі, що включають в себе розробку нових методів стиснення та їхню імплементацію в програмних засобах. Мій досвід та знання, набуті в процесі дипломного дослідження, стали невід'ємною складовою моєї професійної підготовки, що забезпечує мені широкі можливості в подальшій науковій та практичній діяльності в цій області.

У результаті моїх досліджень я зрозумів, що архівування та стиснення даних не лише є технічними аспектами, але й мають велике значення з точки зору економії ресурсів, збереження пропускної здатності мережі та забезпечення безпеки даних. Мій дипломний досвід надав мені глибоке розуміння цих питань і підготував мене до викликів сучасного цифрового світу.

СПИСОК ДЖЕРЕЛ

1. History [of data compression] [Електронний ресурс] – Режим доступу до ресурсу: https://www.wolframscience.com/reference/notes/1069b/.

2. Claude Shannon [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Claude_Shannon.

3. Robert Fano [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Robert_Fano.

4. Fano coding [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding.

5. Алгоритм Шеннона — Фано [Електронний ресурс] – Режим доступу до ресурсу: bit.ly/3wSk5lr.

6. Девід Гаффман [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D0%B2%D1%96%D0%B4_%D0%93%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD.

7. Код Гаффмана [Електронний ресурс] – Режим доступу до ресурсу: bit.ly/4bCalKZ.

8. Алгоритм Хаффмана на пальцях [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/articles/144200/.

9. LZ77 і LZ78 [Електронний ресурс] – Режим доступу до ресурсу: https://bit.ly/3X4PwDE.

10. Алгоритми LZW, LZ77 та LZ78 [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/articles/132683/.

11. Авраам Лемпель [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%90%D0%B2%D1%80%D0%B0%D0%B0%D0%BC_%D0%9B%D0%B5%D0%BC%D0%BF%D0%B5%D0%BB%D1%8C.

12. Яков Зів [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%AF%D0%BA%D0%BE%D0%B2_%D0%97%D1%96%D0%B2.

13. Алгоритм Лемпела-Зіва-Велча [Електронний ресурс] – Режим доступу до ресурсу: https://studfile.net/preview/6889806/page:6/.

14. compress (software) [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Compress_(software)#Special_output_format.

15. bzip2 [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Bzip2.

16. PKZIP [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/PKZIP.

17. Gzip [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Gzip.

18. Deflate [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Deflate.

19. Run-length encoding [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Run-length_encoding.

20. RLE [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/RLE.

Додаток А

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут Інформаційних технологій
Кафедра Комп'ютерної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

на тему: «Розробка та оптимізація алгоритмів стиснення даних з метою підвищення пропускної спроможності мережі»

Виконав студент групи КІД-41
Погоржевський Олексій Романович

Керівник кваліфікаційної роботи:
Торошанко Ярослав Іванович, кандидат технічних наук, доцент

Київ – 2024

Мета роботи – дослідження різних основоположних алгоритмів стиснення даних з метою зменшення їх об'єму для передачі в мережі

Об'єкт дослідження – розбір роботи різних алгоритмів стиснення даних

Предмет дослідження – виконання одного з алгоритмів стиснення даних на мові програмування Python та можливості його оптимізування.

Актуальність обраної теми

Пропускна здатність мережі визначає максимальний обсяг даних, який може бути переданий через мережу за певний час. Ця характеристика стає ключовою у високовитратних мережах, де значні обсяги даних передаються на великі відстані. Підвищення пропускної здатності мережі сприятиме ефективнішому використанню інфраструктури та забезпечить задоволення потреб користувачів у швидкій та безперервній передачі даних.

Алгоритм Шеннона-Фано

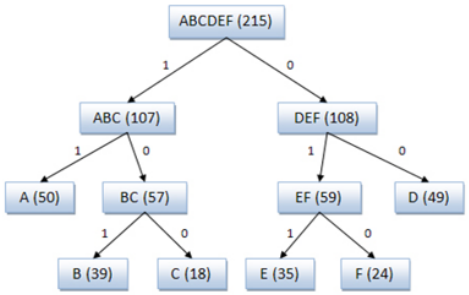
Основні етапи кодування Шеннона-Фано:

а) вхідне повідомлення записується в таблицю частоти повторень кожного символу і всі частоти сумуються між собою (таким чином утворюється коріння так званого дерева кодування);

б) символи поділяють навпіл на приблизно рівні частоти;

в) в двох утворених гілках продовжується операція розділення частот символів доки в кінцевих гілках не отримаємо 1 або 0;

г) отриманим частинам призначаються відповідні двійкові цифри в префіксному коді (для гілок що йдуть першими (з лівої сторони) призначається код 1, а для гілок які йдуть другими (з правої сторони) призначається код 0).



Алгоритм Хаффмана

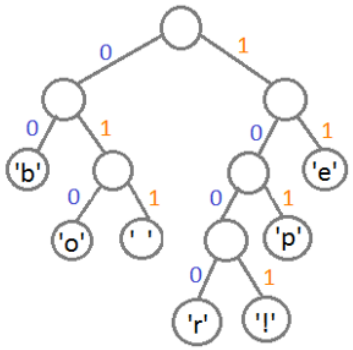
Основні етапи кодування Хаффмана:

а) маючи вхідний алфавіт з частотою повторень кожного символу повідомлення записує їх у таблицю і використовує частоту повторень в якості пріоритету;

б) обирає два вузли з найменшими пріоритетами та поєднує їх у один вузол, а сума частот двох з'єднаних вузлів буде частотою отриманого вузла (тобто маючи вузли з частотою 1 та 2 отримаємо один вузол з частотою 3);

в) отриманий вузол переміщається по таблиці пріоритетів в залежності від отриманого номеру.

Подальша послідовність дій виконується повторно відповідно наведеним вище пунктам поки вся послідовність не дійде до коріння дерева (останнього символу з найбільшою частотою).



Алгоритм Лемпеля-Зіва

LZ77 використовує вже переглянуту частину повідомлення як словник. Щоб досягти стиснення, він намагається замінити вхідний фрагмент повідомлення на фрагмент який вже відомий вмісту словника.

В якості моделі даних LZ77 використовує вікно, що ковзає за повідомленням, розділене на дві нерівні частини. Перша, велика за розміром, включає вже переглянуту частину повідомлення, друга - набагато менша і є буфером, що містить ще не закодовані символи вхідного потоку.

Position	Input: 'AACAACABCABAAAC'		Output Pair
	Lookup Array	Encoded Array	(offset, length)
0	∅	A ACAACABCABAAAC	(0, 'A')
1	A	A CAACABCABAAAC	(1, 1)
2	AA	C AACABCABAAAC	(0, 'C')
3	AAC	AACA BCABAAAC	(3, 4)
7	AACAACA	B CABAAAC	(0, 'B')
8	AACAACAB	CAB AAAC	(3, 3)
11	AACAACABCAB	AA AC	(11, 2)
13	AACAACABCABAA	AC	(12, 2)

Алгоритм Лемпеля-Зіва-Велча

Процес кодування починається з ініціалізації словника в який одразу вноситься початковий алфавіт A, B, C, D, E, F и так далі аж до Z. Після ініціалізації словника починається кодування з першого символу і якщо цей символ є в словнику на вихід подається його код з словника, наприклад, якщо це символ A то на вихід виводиться 0, тобто його код в словнику.

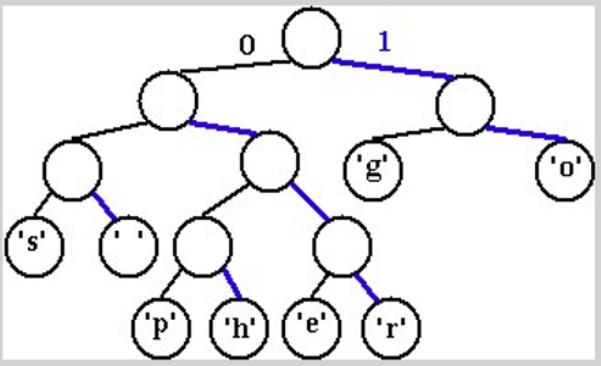
INPUT SEQUENCE:ACGACCACCCGACAGGT						
SLNo:	CHAR	STR + CHAR	IN DICT	ADD TO DICT	STRING	O/P
1	A	A			A	
2	C	AC	N	5=AC	C	1
3	G	CG	N	6=CG	G	3
4	A	GA	N	7=GA	A	2
5	C	AC=5	Y		AC	
6	C	ACC	N	8=ACC	C	5
7	A	CA	N	9=CA	A	3
8	C	AC=5	Y		AC	
9	C	ACC=8	Y		ACC	
10	C	ACCC	N	10=ACCC	C	8

7

Алгоритм Deflate

Алгоритм Deflate — це комбінація двох методів стиснення: кодування Лемпеля-Зіва (LZ77) і кодування Хаффмана. Вхідне повідомлення стискається за спочатку за допомогою LZ77 а потім ще більше стискається використовуючи алгоритм Хаффмана

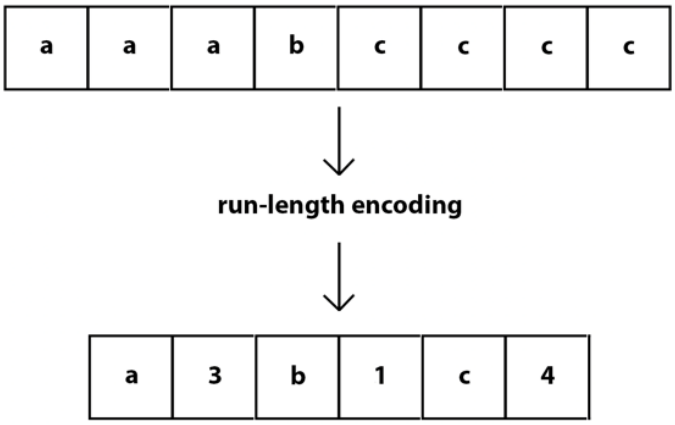
char	binary
'g'	10
'o'	11
'p'	0100
'h'	0101
'e'	0110
'r'	0111
's'	000
' '	001



8

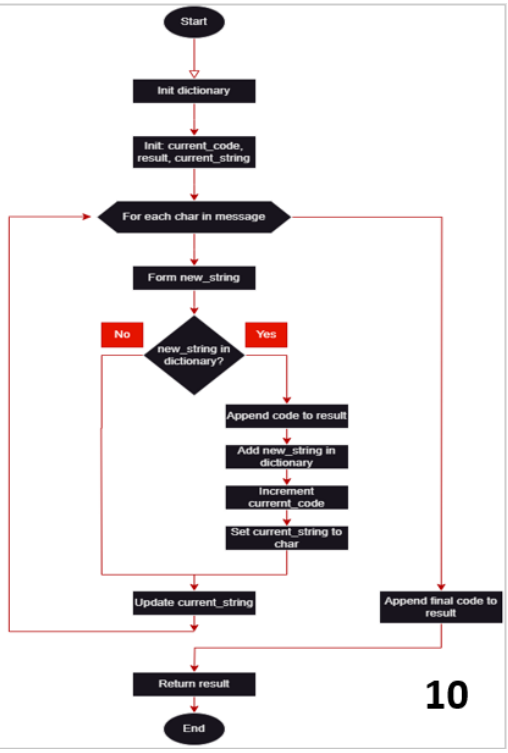
Алгоритм RLE (Run-Length Encoding)

RLE — це простий метод стиснення даних, який використовується для зменшення розміру файлів з послідовностями повторюваних символів. Використовується для фотографій та картинок

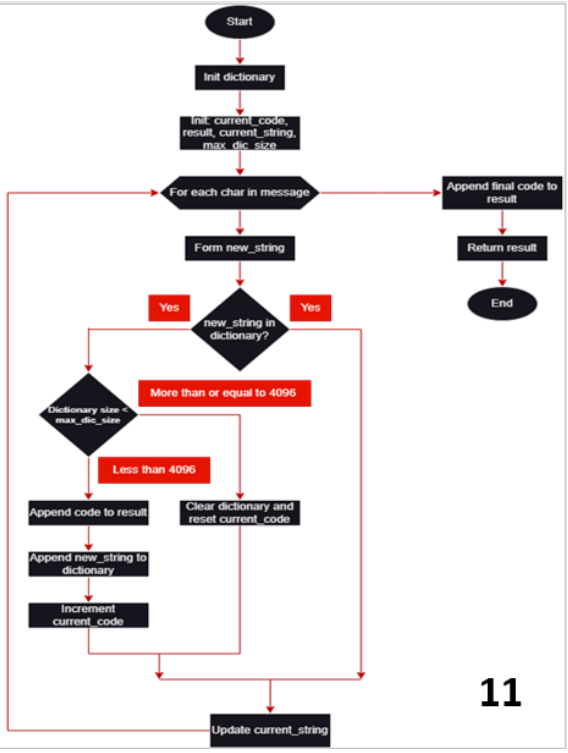


Реалізація Алгоритму LZW засобами мови програмування Python

Блок-схема показує просту реалізацію алгоритму який може зашифрувати повідомлення використовуючи таблицю значень ASCII з якої і формує свій словник.



Для оптимізації алгоритму виконано обмеження словника до 12 бітів та додана можливість попереднього перегляду повідомлення для зменшення навантаження на звертання до словника



Для перевірки оптимізації коду було виконано порівняння часу виконання оптимізованого коду та неоптимізованого.

Неоптимізований впорався з шифрування повідомлення за 1.51 а оптимізований, за 1.44. Враховуючи відносно невелике повідомлення яке зашифровувалося різниця велика. Для більш масштабних повідомлень різниця буде набагато більшою

```
Enter message: sdfgkhjads;lfjhg;lkdasfg;lksjdf;pgjae;lprjg;lqejroitdslakfvbldnblnadfolknboladfsd/;lfsld;kjfga;dfgjmkvnxzlncojkbdfoigpasdfgweri5t934086u1234j05lkm3rl;kdj;oks.
    lkzxlgnoiueryrt0974y6io34lknfl;gbnzdoiyuwe495634261kmlv;gznxdfl,mnlkznxd;oibvuyherp9o8gtye45tykjnsl.kcbnv cl.nkxoyry9r8t34w5jo;ith9ud
Compressed: [115, 100, 102, 103, 107, 104, 106, 97, 100, 115, 59, 108, 102, 106, 104, 103, 266, 107, 100, 97, 115, 258, 272, 115, 106, 257, 59, 112, 103, 262, 101, 266, 112, 114, 106, 271, 108, 113,
    101, 106, 114, 111, 105, 116, 264, 108, 97, 107, 102, 118, 98, 108, 100, 110, 306, 110, 263, 102, 111, 108, 107, 309, 314, 312, 256, 47, 266, 102, 115, 307, 59, 107, 106, 258, 97, 59, 257, 284,
    109, 107, 118, 110, 120, 122, 108, 110, 99, 111, 106, 107, 98, 257, 297, 103, 112, 275, 332, 119, 101, 114, 105, 53, 116, 57, 51, 52, 48, 56, 54, 117, 49, 50, 360, 106, 48, 53, 315, 109, 51, 114,
     108, 326, 100, 106, 59, 111, 107, 115, 46, 315, 122, 120, 108, 103, 110, 297, 117, 354, 121, 114, 116, 48, 57, 55, 52, 121, 54, 105, 111, 360, 315, 110, 102, 376, 103, 98, 110, 122, 100, 297,
    121, 117, 353, 52, 57, 53, 54, 360, 50, 54, 372, 108, 118, 59, 103, 122, 337, 257, 108, 44, 109, 110, 385, 432, 380, 105, 98, 118, 117, 121, 104, 354, 112, 57, 111, 56, 103, 116, 121, 101, 52,
    357, 121, 327, 110, 324, 46, 107, 99, 411, 118, 32, 99, 108, 46, 110, 107, 120, 111, 394, 121, 57, 114, 56, 116, 360, 119, 53, 106, 111, 59, 298, 104, 57, 117, 100]
Decompressed: sdfgkhjads;lfjhg;lkdasfg;lksjdf;pgjae;lprjg;lqejroitdslakfvbldnblnadfolknboladfsd/;lfsld;kjfga;dfgjmkvnxzlncojkbdfoigpasdfgweri5t934086u1234j05lkm3rl;kdj;oks.
    lkzxlgnoiueryrt0974y6io34lknfl;gbnzdoiyuwe495634261kmlv;gznxdfl,mnlkznxd;oibvuyherp9o8gtye45tykjnsl.kcbnv cl.nkxoyry9r8t34w5jo;ith9ud
Total execution time: 1.5146818161010742 seconds
```

```
Enter message: sdfgkhjads;lfjhg;lkdasfg;lksjdf;pgjae;lprjg;lqejroitdslakfvbldnblnadfolknboladfsd/;lfsld;kjfga;dfgjmkvnxzlncojkbdfoigpasdfgweri5t934086u1234j05lkm3rl;kdj;oks.
    lkzxlgnoiueryrt0974y6io34lknfl;gbnzdoiyuwe495634261kmlv;gznxdfl,mnlkznxd;oibvuyherp9o8gtye45tykjnsl.kcbnv cl.nkxoyry9r8t34w5jo;ith9ud
Compressed: [115, 100, 102, 103, 107, 104, 106, 97, 100, 115, 59, 108, 102, 106, 104, 103, 266, 107, 100, 97, 115, 258, 272, 115, 106, 257, 59, 112, 103, 262, 101, 266, 112, 114, 106, 271, 108, 113,
    101, 106, 114, 111, 105, 116, 264, 108, 97, 107, 102, 118, 98, 108, 100, 110, 306, 110, 263, 102, 111, 108, 107, 309, 314, 312, 256, 47, 266, 102, 115, 307, 59, 107, 106, 258, 97, 59, 257, 284,
    109, 107, 118, 110, 120, 122, 108, 110, 99, 111, 106, 107, 98, 257, 297, 103, 112, 275, 332, 119, 101, 114, 105, 53, 116, 57, 51, 52, 48, 56, 54, 117, 49, 50, 360, 106, 48, 53, 315, 109, 51, 114,
     108, 326, 100, 106, 59, 111, 107, 115, 46, 315, 122, 120, 108, 103, 110, 297, 117, 354, 121, 114, 116, 48, 57, 55, 52, 121, 54, 105, 111, 360, 315, 110, 102, 376, 103, 98, 110, 122, 100, 297,
    121, 117, 353, 52, 57, 53, 54, 360, 50, 54, 372, 108, 118, 59, 103, 122, 337, 257, 108, 44, 109, 110, 385, 432, 380, 105, 98, 118, 117, 121, 104, 354, 112, 57, 111, 56, 103, 116, 121, 101, 52,
    357, 121, 327, 110, 324, 46, 107, 99, 411, 118, 32, 99, 108, 46, 110, 107, 120, 111, 394, 121, 57, 114, 56, 116, 360, 119, 53, 106, 111, 59, 298, 104, 57, 117, 100]
Decompressed: sdfgkhjads;lfjhg;lkdasfg;lksjdf;pgjae;lprjg;lqejroitdslakfvbldnblnadfolknboladfsd/;lfsld;kjfga;dfgjmkvnxzlncojkbdfoigpasdfgweri5t934086u1234j05lkm3rl;kdj;oks.
    lkzxlgnoiueryrt0974y6io34lknfl;gbnzdoiyuwe495634261kmlv;gznxdfl,mnlkznxd;oibvuyherp9o8gtye45tykjnsl.kcbnv cl.nkxoyry9r8t34w5jo;ith9ud
Total execution time: 1.4452476501464844 seconds
```

Окрім цього для перевірки була додана можливість моніторингу пам’яті. В даному випадку використання даних відрізняється не дуже сильно тому що повідомлення невелике

Line #	Mem usage	Increment
4	22.2 MiB	22.2 MiB
5		
6	22.2 MiB	0.0 MiB
7	22.2 MiB	0.0 MiB
8	22.2 MiB	0.0 MiB
9	22.2 MiB	0.0 MiB
10		
11	22.2 MiB	0.0 MiB
12	22.2 MiB	0.0 MiB
13	22.2 MiB	0.0 MiB
14	22.2 MiB	0.0 MiB
15		
16	22.2 MiB	0.0 MiB
17	22.2 MiB	0.0 MiB
18	22.2 MiB	0.0 MiB
19	22.2 MiB	0.0 MiB
20		
21	22.2 MiB	0.0 MiB
22	22.2 MiB	0.0 MiB
23		
24	22.2 MiB	0.0 MiB

Line #	Mem usage	Increment
4	22.1 MiB	22.1 MiB
5		
6	22.1 MiB	0.0 MiB
7	22.1 MiB	0.0 MiB
8	22.1 MiB	0.0 MiB
9	22.1 MiB	0.0 MiB
10	22.1 MiB	0.0 MiB
11		
12	22.1 MiB	0.0 MiB
13	22.1 MiB	0.0 MiB
14	22.1 MiB	0.0 MiB
15	22.1 MiB	0.0 MiB
16		
17	22.1 MiB	0.0 MiB
18	22.1 MiB	0.0 MiB
19	22.1 MiB	0.0 MiB
20	22.1 MiB	0.0 MiB
21	22.1 MiB	0.0 MiB
22		
23	22.1 MiB	0.0 MiB
24	22.1 MiB	0.0 MiB
25		
26	22.1 MiB	0.0 MiB

Для подальшої оптимізації можемо додати до коду можливість читання буль-яких символів, а не тільки таблицю ASCII, тобто використати UNICODE. Крім цього можна ще:
а) Використання хеш-таблиці для словника;
б) Динамічне збільшення розміру коду;
в) Використання побітових операцій;
г) Застосування багатопотоковості;
д) Використання індексів замість рядків;
е) Попереднє аналізування даних.

Висновки

Одним з методів підвищення пропускної здатності мережі є вдосконалення алгоритмів стиснення даних. Стискання даних полягає у зменшенні їх обсягу шляхом вилучення зайвої інформації або представлення даних у більш компактному форматі. Ефективні методи стискання можуть значно зменшити обсяг передаваних даних, що, у свою чергу, призводить до підвищення пропускної здатності мережі та поліпшення її продуктивності.

В даній роботі були розглянуті алгоритми стиснення даних. На мові програмування Python був реалізований код виконання алгоритму LZW на основі вхідного повідомлення від користувача та можливості для його оптимізації.