КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки

на тему:

Розробка програми компресії CSV-файлів

Виконав студент 4-го курсу
Гутаревич Олександр Сергійович

(підпис)

Науковий керівник:
доцент, кандидат технічних наук
Ткаченко Олексій Миколайович

(підпис)

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

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«____» _____ 202_ р., протокол № ___
Завідувач кафедри
М. С. Нікітченко _____
(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 53 сторінки, 13 ілюстрацій, 7 таблиць, 34 джерела посилань.

CSV ФОРМАТ, КОМПРЕСІЯ ФАЙЛІВ У ФОРМАТІ CSV, КОМПРЕСІЯ ДАНИХ, МЕТОД ЛЕМПЕЛЯ-ЗІВА-ВЕЛЧА, МЕТОД ХАФФМАНА, МЕТОДИ КОМПРЕСІЇ ДАНИХ БЕЗ ВТРАТ, ПЕРЕТВОРЕННЯ БЕРРОУЗА-ВІЛЕРА.

Об’єктом дослідження є CSV-формат. Предметом є методи та їх програмна реалізація для стиснення файлів у форматі CSV.

Метою кваліфікаційної роботи є розробка програми для компресії CSV-файлів, яка б враховувала специфіку файлів CSV формату.

Методи розробки: кодування довжин серій, метод Лемпеля-Зіва-Велча, метод Хаффмана, перетворення Берроуза-Вілера, методи алгоритмічної та об’єктно-орієнтованої декомпозиції, загальні методи (порівняння). Інструменти розробки: інтегроване середовище розробки QT Creator[1], призначене для створення крос-платформових застосунків з використанням бібліотеки QT framework [2], мова програмування C++.

Результат роботи: детально розглянуто специфіку формату CSV; виконано загальний огляд компресії даних; детально розглянуті деякі методи компресії без втрат; розроблено алгоритм, який враховує специфіку даних у форматі CSV в процесі їх компресії; розроблений алгоритм був реалізований у програмі, яка призначена для стиснення файлів у форматі CSV; розроблена програма була протестована на ряді файлів у форматі CSV; на основі даних отриманих в результаті тестування був зроблений висновок про доцільність використання програми з ціллю стиснення CSV-файлів.

Зміст

ВСТУП

Оцінка сучасного стану об’єкта розробки. CSV – це один з найбільш популярних на сьогодні форматів для збереження та обробки табличних даних. Для збереження або передачі даних у форматі CSV активно використовується компресія даних.

Для зменшення обсягу даних у форматі CSV, з ціллю їх подальшої передачі або збереження, на сьогодні активно використовують універсальні методи стиснення даних, які не враховують особливості файлів у форматі CSV, таким чином не забезпечують мінімізацію розміру даних.

На сьогодні, на ринку відсутнє рішення, яке б враховувало природу CSV-файлів при їх компресії та забезпечувало максимальну ефективність стиснення їх вмісту. Таким чином виникає задача розробки програми для стиснення CSV фалів, яка б враховувала природу даних, які вони зберігають, при виконанні їх компресії.

Актуальність роботи та підстави для її виконання. Так як CSV – популярний формат даних, який використовується в різноманітних сферах, то задача економії пам’яті для збереження або передачі таких даних є актуальною.

Виходячи з того, що наявні на ринку засоби для вирішення цієї задачі (WinZip[3], gzip[4], 7-zip[5] та ін.) є узагальненими та не враховують специфіку файлів у форматі CSV, або передбачають переформатування даних, то виникає потреба у створенні інструменту для стиснення CSV файлів, який враховував би їх природу та особливості в процесі компресії.

Мета й завдання роботи. Метою кваліфікаційної роботи є розробка програми для компресії CSV файлів, яка б враховувала специфіку CSV формату. Для досягнення цієї мети поставлено такі завдання:

- Теоретично вивчити формат даних CSV, його природу та специфіку.

- Розглянути основні базові підходи до стиснення даних.
- Розробити алгоритм стиснення файлів у форматі CSV, який враховує специфіку даних у цьому форматі.
- Реалізувати програму-компресор CSV-файлів, яка працює за допомогою розробленого алгоритму.

Об'єкт, методи й засоби розроблення. Об’єктом дослідження є CSV формат даних. Предметом є методи та їх програмна реалізація для стиснення файлів у форматі CSV.

Для розробки програмної реалізації було використано мову програмування C++. Ця мова надає практично повний доступ до контролю пам’яті, що сприяє продуктивності розроблених додатків та має потужну стандартну бібліотеку. Для розробки інтерфейсу був використаний крос-платформні інструментарій розробки програмного забезпечення QT framework [2].

Під час розробки продукту використовувалося інтегроване середовище розробки QT Creator[1], яке дозволяє створювати крос-платформні додатки мовою C++. Також, для відслідковування змін, під час розробки програми активно використовувалася система контролю версій git[6].

РОЗДІЛ 1. ФОРМАТ ЗБЕРЕЖЕННЯ ДАНИХ CSV

1.1 CSV формат збереження даних

Comma-separated values (більш відомий як CSV) – це надзвичайно популярний текстовий формат, який широко використовується для обробки та зберігання табличних даних. Цей формат є одним з найбільш широко використовуваних форматів для наукових та інженерних завдань. Окрім наукових розрахунків, він також широко використовується у галузі охорони здоров'я, виробництва та фінансів.

1.2 Стандартне означення формату CSV

CSV формат, на відміну від інших текстових форматів, які використовуються для обробки даних, таких як JSON та XML, не має єдиної специфікації. Це робить можливим існування великої кількості реалізацій цього формату, які залежать від специфіки задач, які вирішуються.

Хоч і єдиної стандартизованої специфікації не існує, проте практично всі наявні реалізації цього формату слідують правилам, описаним в стандарті RFC4180[7]:

1. Кожен запис знаходиться в окремому рядку, відокремленому розривом рядка (CRLF). Наприклад:

   aaa,bbb,ccc CRLF

   zzz,yyy,xxx CRLF

2. Останній запис у файлі може мати або не мати розриву рядка. Наприклад:

   aaa,bbb,ccc CRLF

   zzz,yyy,xxx

3. Перший рядок файлу може бути рядком заголовків, який оформлений так само, як і звичайні рядки-записи. Цей рядок заголовків містить імена, що відповідають полям у файлі, і повинен мати ту саму кількість полів, що і записи в решті файлу (наявність або відсутність рядка заголовка повинна бути вказана за допомогою необов'язкового параметра "header"). Наприклад:

field_name,field_name,field_name CRLF

aaa,bbb,ccc CRLF

zzz,yyy,xxx CRLF

4. У рядку заголовків та кожному рядку-записі може бути одне або кілька розділених комами полів. Кожен рядок файлу має містити однакову кількість полів. Пробіли вважаються частиною поля і їх не слід ігнорувати. Після останнього поля рядку кома не ставиться. Наприклад:

aaa,bbb,ccc

5. Кожне поле може включати або не включати подвійні лапки (") (деякі програми, такі як Microsoft Excel, взагалі не використовують подвійні лапки). Якщо поле не загорнуте в подвійні лапки, то воно не може містити подвійні лапки (символ "). Наприклад:

"aaa","bbb","ccc" CRLF

zzz,yyy,xxx

6. Поля, що містять розриви рядків (CRLF), подвійні лапки (") або коми (,), мають бути обгорнуті в подвійні лапки("). Наприклад:

"aaa","b CRLF bb","ccc" CRLF

zzz,yyy,xxx

7. Якщо поле огорнуте подвійними лапками, то подвійні лапки всередині поля мають бути екрановані попередніми лапками. Наприклад:

"aaa","b""bb","ccc"

ABNF граматика:

file = [header CRLF] record *(CRLF record) [CRLF]

header = name *(COMMA name)

record = field *(COMMA field)

name = field

field = (escaped / non-escaped)

escaped = DQUOTE *(TEXTDATA / COMMA / CR / LF / 2DQUOTE) DQUOTE

non-escaped = *TEXTDATA

1.3 Історія формату CSV

CSV формат існував ще до появи першого персонального комп’ютера. Компілятор IBM Fortran підтримував цей формат, коли він вийшов в реліз у 1972 році. Ввід / вивід спрямований на список, також відомий як "free form", (визначений у FORTRAN 77, затвердженому в 1978 р.) використовував коми або пробіли як роздільник та використовував огортання подвійними лапками значень, які містять кому або пробіл.

Термін comma-separated values (і відповідне скорочення CSV) був вперше використаний в документації до комп’ютера Osborne Executive в 1983 році. Перша спроба стандартизації цього формату відбулася в 2005 році, коли був розроблений стандарт RFC4180[7]. Пізніше, в 2013 році, деякі недоліки RFC4180[7] були усунені за рекомендацією W3C.

В 2014 році IETF опублікував RFC7111[8] з описом застосування фрагментів URI до файлів у форматі CSV. RFC7111[8] визначає, як можна вибрати діапазони рядків, стовпців і комірок з CSV-файла, використовуючи позиційні індекси.

У 2015 році W3C, з ціллю покращити формат CSV за допомогою формальної семантики, оприлюднив перші проекти рекомендацій щодо стандартів CSV-метаданих.

1.4 Переваги та недоліки формату CSV

Використання даних у форматі CSV має чисельні переваги, які зазвичай залежать від конкретної задачі та сфери застосування. Нижче наведено найбільш загальні переваги формату CSV:

- CSV легко створити. Дійсно, оскільки формат CSV є інтуїтивно зрозумілим, його легко створити. Насправді, це можна зробити за допомогою будь-якого текстового редактора (на відміну від формату XLS або XLSX).
- CSV-файли можна прочитати практично будь-яким текстовим редактором.
- Дані у форматі CSV легко піддаються аналізу. Парсер для CSV файлів набагато простіший у порівнянні з парсерами XML або JSON форматів.
- Формат CSV має достатньо просту схему, що робить його популярним для вирішення задач зі складними математичними обчисленнями.
- Маніпуляції з CSV файлами відбувається швидко. В силу простоти цього формату практично ніщо не сповільняє аналіз файлу, що призводить до швидкого читання та запису даних.
- Формат CSV компактний. Порівняно з іншими форматами, такими як XML або JSON, не потрібно дбати про теги.

Крім зазначених переваг CSV формат має також ряд недоліків:

- CSV дозволяє зберігати дані з декількома вимірами лише у формі флет-таблиці.
- Відсутня типізація стовпчиків, тобто в CSV файлах нема різниці між текстовими та числовими стовпцями.
- Слабка підтримка спеціальних символів.
- Відсутність єдиного стандарту.

- Не існує стандартного способу відображати бінарні дані.

Загалом, попри всі недоліки, формат CSV надзвичайно часто використовується для вирішення різноманітних задач в різних сферах. Наприклад, на популярній платформі для організації конкурсів з аналізу даних та машинного навчання, Kaggle[9] значна частина датасетів зберігається саме у форматі CSV.

1.5 Проблема стиснення CSV-файлів

Кожну секунду людство генерує мільярди байтів інформації, яку необхідно зберігати. За приблизними оцінками, щодня Facebook генерує 4 петабайта даних, Twitter – 12 терабайт даних, а WhatsApp – близько 3 петабайт даних [10]. Оскільки кількість інформації, яку необхідно зберігати, зростає з кожним роком, постає проблема економії пам’яті для збереження цих даних. Так як CSV – один з популярних форматів збереження табличних даних, які можуть займати достатньо багато пам’яті, то актуальною є проблема компресії файлів збережених в цьому форматі.

Зазвичай для стиснення файлів у форматі CSV використовують універсальні методи стиснення без втрат такі як: Deflate, LZMA, LZSS, та інші. Проте ці універсальні методи компресії не враховують специфіку табличної структури CSV файлу, що негативно відображається на ефективності стиснення.

Альтернативним рішенням проблеми стиснення CSV файлів є використання інших форматів для збереження табличних даних, таких як Parquet[34] та Avro[11]. Проте цей підхід має ряд недоліків, що робить його непопулярним на практиці.

По-перше, такі формати націлені на підвищення ефективності доступу до редагування та перегляду даних, а не на економію пам’яті. По-друге, конвертація даних до цих форматів вимагає поглиблені знання даних збережених у файлі, адже якісна конвертація можлива лише при заданні схеми

даних (з типами) вручну. Крім того, в багатьох випадках практичні задачі потребують використання саме CSV формату, а не інших, і постійні переходи від одного формату до іншого потребують значних системних ресурсів та можуть призводити до втрати даних.

Таким чином, наявні рішення проблеми стиснення файлів у форматі CSV є не ідеальними і ця проблема залишається актуальною. Для спроби вирішити цю проблему необхідно вивчити основні підходи до стиснення даних.

РОЗДІЛ 2. МЕТОДИ КОМПРЕСІЇ ДАНИХ

2.1 Історія компресії даних

Зародженням компресії даних можна вважати появу в 1838 р. азбуки Морзе (рис. 2.1), яка широко використовувалася в телеграфії. Вона дозволила зменшити об’єм даних, які треба передати, шляхом кодування найбільш вживаних символів, таких як E і T, короткими кодами , а рідко вживаних – довгими. Застосування азбуки Морзе значно пришвидшило передачу даних, адже зменшило їх розмір.

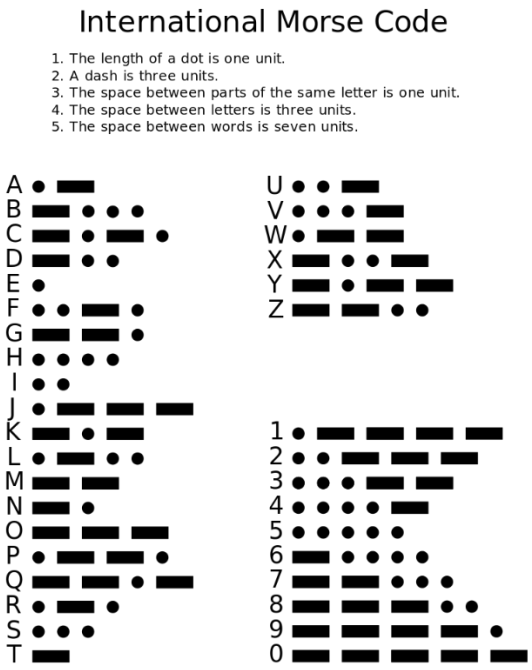

Рисунок 2.1 – Азбука Морзе

Стрімкий розвиток методів компресії даних розпочався в 1948 р. разом з виникненням Теорії інформації – розділом прикладної математики, що досліджує процеси збереження, перетворення і передачі інформації.

В 1949 році Клод Шеннон і Роберт Фано розробили свій алгоритм компресії даних, названий в їх честь, кодуванням Шеннона-Фано. Цей

алгоритм базується на присвоєнні одиницям даних (символам) кодів залежно від їх частоти входження в дані, які кодуються.

Девід Хаффман розвинув їх ідею і створив свій алгоритм, названий алгоритмом Хаффмана. Основна його відмінність від алгоритму Шеннона-Фано полягає в способі побудови дерева частот: в алгоритмі Шеннона-Фано воно будується знизу вверх, а алгоритм Хаффмана передбачає побудову дерева зверху вниз.

В 1977 році з'явився перший словниковий метод компресії – LZ77, розроблений Аврамом Лемпелем та Яковом Зівом. Для компресії він використовує динамічний словник, який називають «ковзаючим вікном». Основна ідея цього алгоритму полягає в заміні повторів підпослідовностей одиниць даних на посилання на позиції в даних, де ці підпослідовності вже зустрічалася.

В 1978 році Аврам Лемпель і Яков Зів подарували світу свій новий алгоритм компресії – LZ78. Як і LZ77 він є словниковим, проте на відміну від LZ77 він використовує статичний словник, який генерується в процесі кодування та декодування.

1984 року Террі Велч розробив новий алгоритм, що базується на LZ78, названий LZW. Він виявився зручним як для апаратної так і для програмної реалізації, що зробило його одним з найбільш часто використовуваних методів компресії даних.

Зі зростанням популярності збереження мультимедійної інформації (цифрових зображення, аудіо та відео) в кінці 1980-х років було розроблено купу форматів для компактного збереження цих даних. Компактність досягалася використанням різноманітних методів компресії та їх комбінацій. Наприклад, популярний формат зображень та анімацій GIF (Graphics Interchange Format) використовує раніше згаданий метод компресії LZW.

В 1990-ті роки і по сьогодні розвиток в цій сфері не зупиняється. Виникають все нові файлові формати, які реалізують різноманітні види і підходи до компресії даних. Таким чином були розроблені такі звичні для нас

формати як JPEG і PNG. Популярності набувають архіви – файли, які є композицією одного або декількох файлів та метаданих. Архіви використовуються для зібрання різних файлів разом для зручнішого перенесення, збереження або для їх стиснення з метою економії пам’яті. В цьому їм допомагають методи компресії без втрат, адже неточності у відновленні файлів архіву недопустимі. Найбільшим попитом користуються форматами архівів rar, gzip та zip.

2.2 Класифікація методів стиснення даних

Методи компресії даних поділяються на 2 класи: методи компресії без втрат (lossless) та з втратами (lossy). Методи компресії даних без втрат – це методи компресії даних, при використанні яких закодовані дані однозначно можуть бути відновлені з точністю до біта. Методи компресії з втратами, відповідно, не забезпечують однозначного відновлення даних, але забезпечують незначну для їх подальшого використання різницю відновлених (декомпресованих) даних з початковими.

Області застосування цих двох класів методів відрізняються, але обидва класи надзвичайно часто використовуються. Методи компресії з втратами зазвичай застосовуються для збереження чуттєвої інформації (зображень, звукових доріжок та відео), адже неточності в декомпресованих даних можуть бути непомітні для людини через особливості її органів сприйняття. Це показано на рисунку 2.2[17]: без збільшення масштабу важко відрізнити стиснуті з втратами зображення (справа) від оригіналу (зліва). Прикладом застосування стиснення з втратами є всім добре відомий формат зображень JPEG (використовує для компресії однойменний метод). Методи компресії без втрат застосовуються для даних, для яких точність декомпресії є вирішальною, наприклад, для текстових даних навіть незначні неточності можуть унеможливити сприйняття декомпресованої інформації. Прикладом застосування стиснення даних без втрат є утиліта gzip[4] (використовує для

компресії метод Deflate (комбінація LZ77 + метод Хаффмана)), яка активно використовується в UNIX-системах.

Рисунок 2.2 [17] – Зліва – оригінальне зображення, розмір = 112 кілобайт; Справа – це ж зображення, стиснене з втратами, розмір = 50 кілобайт.

Для стиснення даних збережених у форматі CSV, безглуздо розглядати методи стиснення з втратами, адже неточності в декомпресованому файлі можуть значно викривити дані, які в ньому зберігаються, що призведе до втрати цінності даних. Отже, в контексті стиснення CSV файлів варто розглядати лише методи компресії без втрат. Деякі з них розглянемо в наступних підрозділах.

2.3 Кодування довжин серій

Одним із самих простих та інтуїтивно зрозумілих методів стиснення даних без втрат – RLE (Run-Length encoding, укр. Кодування довжин серій). Компресія за допомогою цього методу виконується за рахунок заміни

послідовності однакових символів (вважаємо що символ – це одиниця даних(байт)) на кількість входжень символу в цю послідовність + сам символ. Наприклад, вхідні дані «AAABCC» будуть перетворені в «3A1B2C». Як бачимо, вдалося зекономити на збереженні одного символу.

Очевидно, що ефективність компресії цього методу дуже сильно залежить від природи вхідних даних: якщо в даних мало, або взагалі немає серій однакових символів, то скомпресовані дані будуть більшими за вхідні, що робить застосування стиснення методом RLE для цих даних абсолютно безглуздим. Наприклад, вхідні дані «ABCABCCCCC» будуть перетворені в «1A1B1C1A1B5C», тобто отримали скомпресовані дані в 1,2 рази більші за вхідні.

Частково цю проблему вирішує невелика модифікацію RLE. Вона полягає в заміні послідовності серій довжиною 1 (послідовність пар 1+симол) на довжину цієї послідовності та послідовність символів цих серій. В такій модифікації необхідно розрізняти довжини серій однакових символі та різних. Для цього пропонується кодувати довжину серії однакових символів невід'ємними числами, а довжину серії різних символів – від'ємними (по суті алфавіт кодів довжин розділяється на дві частини). Наприклад, вхідні дані «ABCABCCCCC» тепер будуть кодуватися наступним чином: «-5ABCAB5C», тобто після модифікації скомпресовані дані мають розмір на 20% менший за вхідні.

Декомпресія методом RLE відбувається за наступним алгоритмом:

0) Якщо досягли кінця фалу, то КІНЕЦЬ, інакше переходимо до 1).

1) X:= наступний код з файлу, переходмо до 2).

2) Якщо (X>0) , то переходимо до 3), інакше переходимо до 4).

3) C:= наступний символ з файлу; Повторюємо X раз: (записуємо C в вихідний файл); Переходимо до 0).

4) Повторити (-X) раз: (C:= наступний символ з файлу; записуємо C в вихідний файл); Перейти до 0).

2.4 Кодування Хаффмана

Метод компресії даних Хаффмана базується на присвоєнні кожному символу вхідних даних коду, довжина якого варіюється залежно від того як часто зустрічається символ у вхідних даних. Таким чином, символам, які частіше зустрічаються в даних, присвоюються коротші коди, а символам, які рідше зустрічаються – довші коди. Досягається це за рахунок побудови дерева Хаффмана.

2.4.1 Алгоритм побудови дерева Хаффмана

Для побудови бінарного дерева Хаффмана необхідно знати кількість входжень кожного символу в вхідний файл. Нехай маємо вектор пар: символ, кількість його входжень. Дерево будується за наступним алгоритмом:

1) Для кожного символу створюємо вузол-листок дерева з вагою рівною кількості входжень цього символу у файл, а значенням рівним символу та додаємо цей вузол-листок в контейнер вузлів; Переходимо до 2).
2) Відсортуємо вузли-листки в контейнері у порядку спадання їх ваги; Переходимо до 3).
3) Якщо контейнер містить один вузол, то КІНЕЦЬ (цей вузол і є коренем шуканого дерева), інакше переходимо до 4).
4) Вилучаємо з контейнера 2 останні (з найменшою вагою) вузли A1 і A2; створюємо B - батьківський вузол для A1 і A2: правий вказівник на A1, лівий на A2, вага(B) = вага(A1)+вага(A2); вставляємо B у контейнер не порушуючи порядок спадання ваги вузлів; переходимо до 3).

Нехай маємо наступний вектор пар (символ, кількість входжень): {(‘p’, 2); (‘r’, 2); (e, 2); (s, 4); (c, 1); (o, 1); (m, 1)}. Демонстрація роботи алгоритму побудови дерева наведена на рисунках 2.3-2.4.

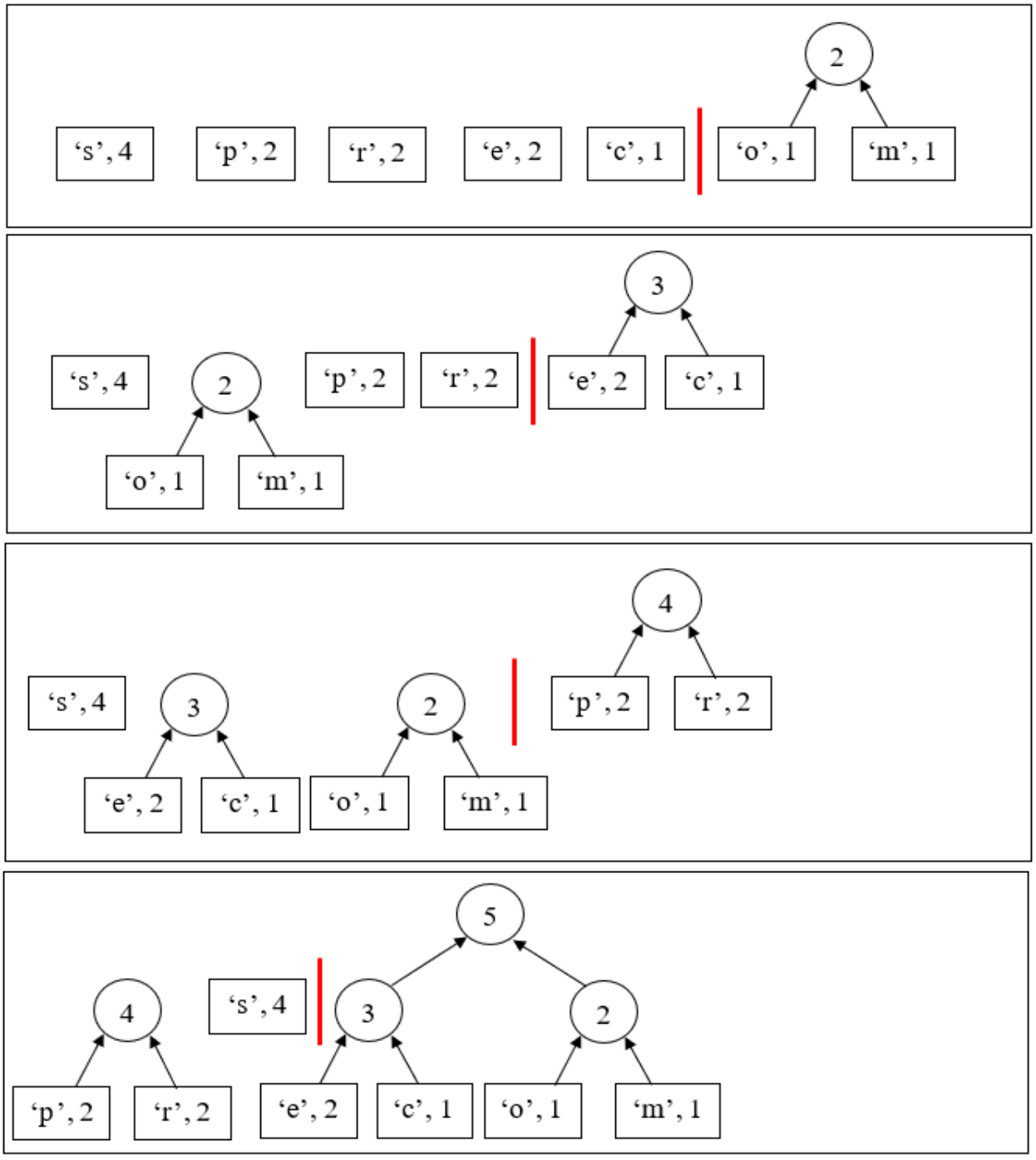

Рис. 2.3 - демонстрація роботи алгоритму побудови дерева Хаффмана частина 1

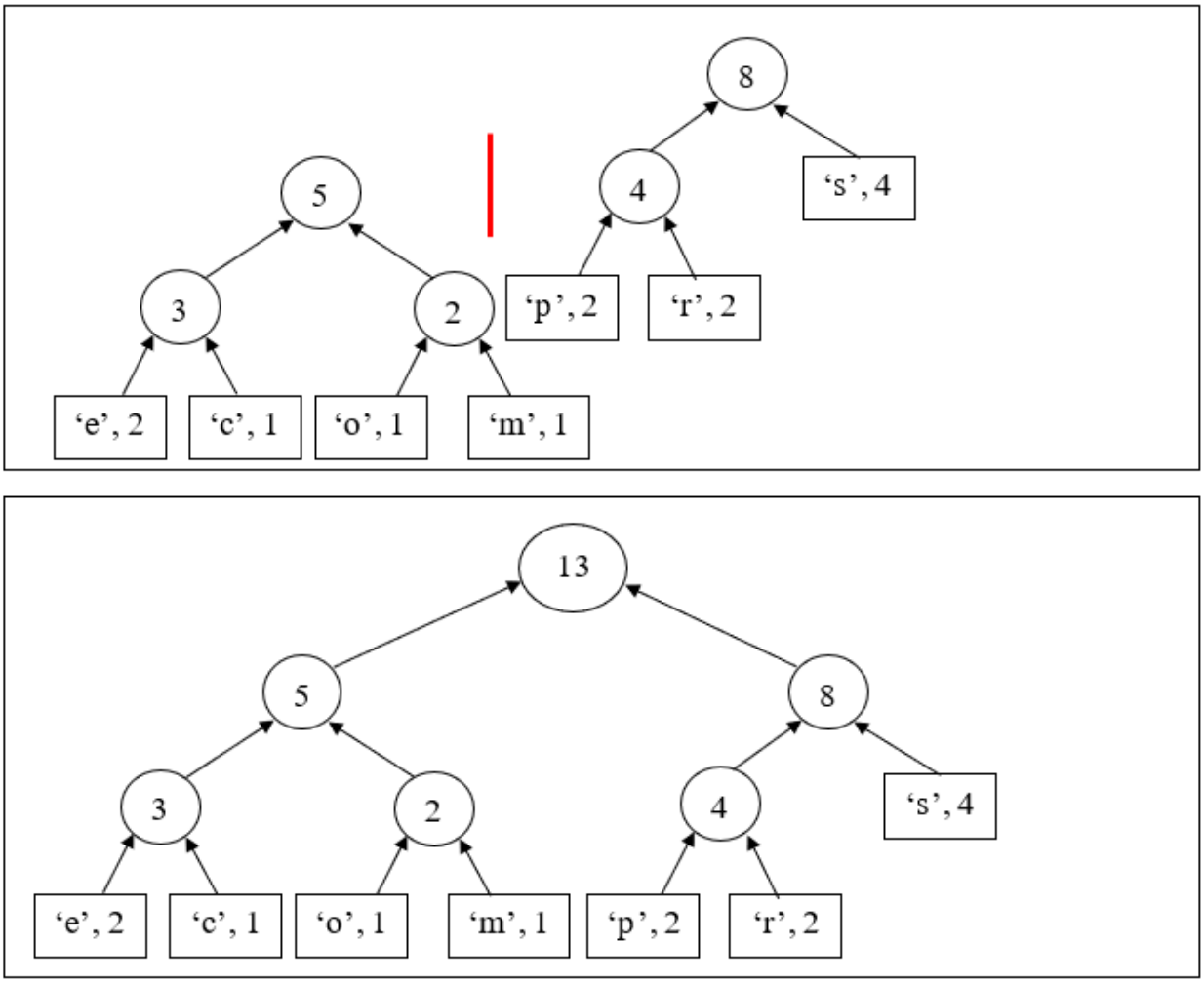

Рис. 2.4 - демонстрація роботи алгоритму побудови дерева Хаффмана частина 2

2.4.2 Використання дерева Хаффмана

Тепер розберемо як відбувається процес компресії та декомпресії даних з використанням дерева Хаффмана. Стиснення відбувається в два проходи по вхідному файлі. Перший прохід необхідний для визначення частоти входження кожного символу в файл, щоб побудувати саме дерево Хаффмана.

Код для символу на основі дерева Хаффмана знаходиться наступним чином: встановлюється вказівник на вузол-листок дерева, в якому зберігається цей же символ, код:=””. Переходимо з поточного до батьківського взула і якщо поточний вузол був для батьківського правою дитиною, то код:=’1’+код, а

якщо лівою, то код:=’0’+код. Коли дійдемо до кореня дерева, отримаємо шуканий код.

Знайшовши коди для всіх символів, виконуємо другий прохід по файлу і для кожного символу записуємо його код в вихідний файл. Таким чином вихідний файл буде містити закодовану інформацію. Розглянемо простий приклад:

Нехай на вхід подається “presscompress”. Знаходимо для нього вектор пар (символ, кількість його входжень): {(‘p’, 2); (‘r’, 2); (e, 2); (s, 4); (c, 1); (o, 1); (m, 1)}. Для таких даних уже розглядався процес побудови дерева Хаффмана, отримаємо дерево як на рисунку 2.5.

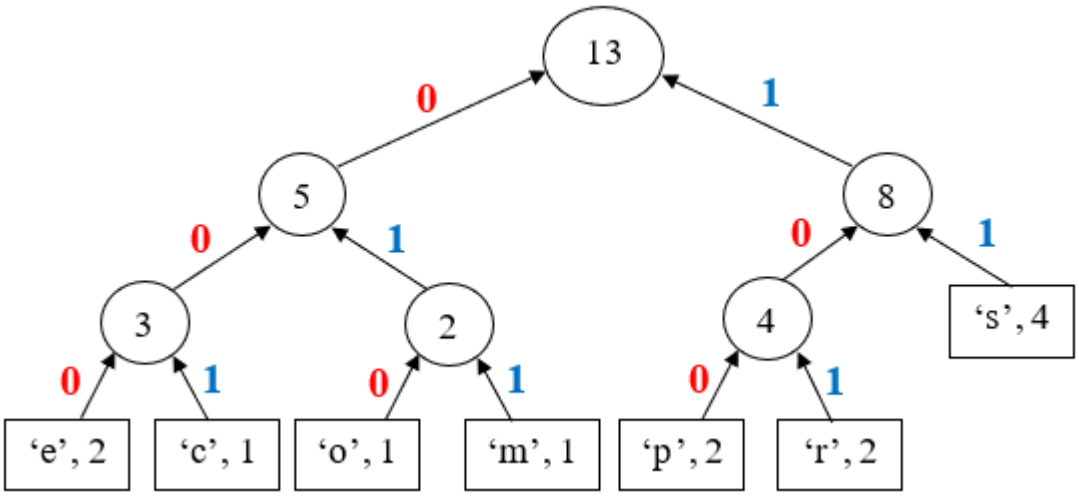

рис.2.5 – дерево Хаффмана для “presscompress”

Тепер, використовуючи описаний раніше підхід, знаходимо коди для символів: e=000; c=001; o=010; m=011; p=100; r=101; s=11. Тепер кодуємо вхідні дані “presscompress” вагою 13 байт (104 біт) як 10010100 01111001 01001110 01010001 111 у бітовому вигляді отримали 35 біт, тобто різниця між вхідними і стиснутими даними дійсно велика. Проте, це не кінцева вага, адже для декомпресії нам необхідно зберегти саме дерево, по якому відбувалося стиснення. Як альтернатива збереженню дерева, можна зберігати вектор пар, по якому легко відновити дерево для декомпресії даних.

Розглянемо як відбувається декомпресія. Для початку необхідно зчитати вектор пар та побудувати дерево Хаффмана по ньому. Тепер встановлюємо вказівник P на корінь дерева і починаємо, власне розкодування даних:

1) Якщо досягли кінця файлу, то КІНЕЦЬ; Інакше перейти до 2).
2) Читаємо наступний біт B; Переходимо до 3).
3) Якщо P вказує на вузол-листок, то

записуємо символ вузола-листка у вихідний файл;

P встановлюємо на корінь дерева;

Інакше:

Якщо B=1, то

Пересуваємо P на праву дитину вузла, на який вказує P;

Інакше

Пересуваємо P на ліву дитину вузла, на який вказує P;

Перейти до 1).

Таким чином, досягнувши кінця файлу з стиснутими даними, в вихідний файл буде записано декомпресовані дані. Вихідний файл буде містити точну копію даних, до стиснення.

2.5 Метод LZW

LZW (Lempel-Ziv-Welch) – один з словникових алгоритмів стиснення даних сімейства алгоритмів LZ. Як вже було сказано, він є модифікацією методу LZ78. Основна ідея алгоритму – замінити послідовності символів кодами, адже код займає менше місця чим послідовність символів.

2.5.1 Алгоритм стиснення

Для стиснення LZW використовується словник кодів для послідовності символів. Розмір кодів такого словника зазвичай встановлюється в діапазоні 9-16 біт. Автор алгоритму, Террі Велч, пропонує користуватися кодами

довжиною 12 біт. Початковий словник містить коди (0-255) для всіх можливих символів (послідовностей з одного символа). Стиснення досягається за рахунок присвоєння вільних у словнику кодів послідовностям символів. Причому цей процес відбувається так, що при декомпресії можливо відновити словник, знаючи лише послідовність кодів.

Алгоритм компресії методом LZW виглядає наступним чином:

1) Словник:=Початковий словник;
   Посл:="";
   Переходимо до 2).
2) Якщо досягли кінця файлу, то
   В вихідний файл записати код для Посл в Словнику;
   КІНЕЦЬ;
   Інакше переходимо до 3).
3) С:=наступний символ з файлу;
   переходимо до 4).
4) Якщо Словник містить (Посл+С), то Посл:=Посл+С;
   перейти до 2).
   Інакше перейти до 5).
5) В вихідний файл записати код для Посл в Словнику;
   Додати (Посл+С) в Словник (тобто надати (Посл+С) код);
   Посл:=С; Перейти до 2).

Наприклад, стиснемо вхідні дані "prepro" за цим алгоритмом. Цей процес демонструє таблиця 2.1. Отримали 112,114,101,256,111.

Табл. 2.1 - Демонстрація роботи компресії методом LZW. Частина 1

С	Посл.	Вихідний файл	Зміни в словник
p	{""}		INIT
r	{p}	=>112	Add(256, {pr})
e	{r}	=>114	Add(257, {re})

Табл. 2.1 - Демонстрація роботи компресії методом LZW. Частина 2

p	{e}	=>101	Add(258, {ep})
r	{p}		
o	{pr}	=>256	Add(259, {pro})
	{o}	=>111	

2.5.2 Проблема переповнення словника

Виникає питання: що робити коли словник переповниться? У цієї проблеми є 2 очевидних способи вирішення. Перший спосіб полягає у продовженні кодування не надаючи нові коди. Другий спосіб використовує спеціальний сигнал для перегрузки словника, тобто присвоєння поточному словнику початкового.

Основна перевага першого способу над другим полягає в тому, що в процесі перезаповнення словника спочатку кодування не ефективне, адже перегружений словник ще не містить кодів для послідовностей і потребує деякого об’єму оброблених даних для заповнення. Проте, порівняно з першим способом другий спосіб краще працює в ситуація, коли є такі послідовності, які не зустрічаються спочатку файлу, але дуже часто наявні в його кінці. Наприклад, при компресії тексту книги ім’я одного з ключових героїв не наявне в перших розділах, але часто з’являється в наступних. При використанні першого способу словник, може заповнитися протягом обробки перших розділів і не містити коду для імені цього персонажа, що негативно повпливає на розмір закодованих даних. При використанні ж другого способу, словник пергрузиться і відносоно швидко надасть імені персонажа код, що посприяє зменшенню розміру стиснутих даних.

Існують ще способи адаптивного оновлення словника, тобто при заповненні словника виконується присвоєння новим послідовностям

невживаних або маловживаних кодів. Такі способи значно ускладнюють алгоритм методу і є досить складними в реалізації.

2.5.3 Алгоритм декомпресії

Як і алгоритм стиснення, алгоритм декомпресії методу LZW є досить простим:

1) Словник = початковий словник;

   O=наступний код з вхідного файлу;

   C=Словник(O) (послідовність символів з словнику для коду O);

   Вивести C в вихідний файл;

   Перейти до 2).

2) Якщо досягли кінця файлу, то КІНЕЦЬ;

   Інакше перейти до 3).

3) H=наступний код з вхідного файлу;

   Якщо H не зайнятий в словнику, то

   C=Словник(O);

   C=C+T;

   Інакше C=Словник(H);

   Вивести C в вихідний файл;

   T=перший символ C;

   Додати (Словник(O)+T) в Словник;

   O=H;

   Перейти до 2).

Демонстрація роботи алгоритму для вхідних даних 112,114,101,256,111 наведена в таблиці 2.2. На виході отримали prepro – точну копію даних, які підлягли компресії.

Табл. 2.2 - Демонстрація роботи декомпресії методом LZW

Крок	O	C	H	T	Словник	Вихід
1	112	p			INIT	+p
2	112	r	114	r	Add(256, {pr})	+r
3	114	e	101	e	Add(257, {re})	+e
4	101	pr	256	p	Add(258, {ep})	+pr
5	256	o	111	o	Add(259, {pro})	+o

2.6 Перетворення Берроуза-Вілера

BWT(Burrows-Wheeler transform), або блочно-сортуюче стиснення – це техніка обробки даних, яка перетворює їх в більш зручну для компресії форму. Тобто, BWT – не метод компресії даних, а скоріше доповнення до інших методів компресії, яке здатне підготовити дані до компресії. Перетворення відбувається шляхом зміни порядку даних таким чином, щоб можна було однозначно відновити їх після перетворення і щоб вони краще підходили для компресії.

Нехай маємо послідовність символів, тобто рядок P. Власне перетворення відбувається наступним чином:

1) Знаходиться T - масив всіх циклічних перестановок рядка P.
2) Рядки в T відсортовуються у порядку зростання.
3) Послідовно, починаючи з першого, закінчуючи останнім, беруться рядки з T і в результуючий рядок записується останній символ кожного рядка.
4) Повертається результуючий рядок та номер початкового рядка в T після процесу сортування.

Наприклад, маємо рядок “prepro”, таблиця 2.3 демонструє як відбувається перетворення. На виході отримуємо номер початкового рядка =

3, вихідний рядок = “rroepp”, який на відміну від вхідного містить серії однакових символів.

Табл. 2.3 - Демонстрація перетворення Берроуза-Вілера.

Вхідний рядок	T	T (сорт.)	Вихідний рядок
prepro	prepro oprepr roprep proper epropr reprop	epropr oprepr prepro propre reprop roprep	rroepp (3)

Нехай маємо перетворений рядок P та номер початкового рядка K. Зворотнє перетворення відбувається наступним чином:

1) Створюється порожня таблиця символів T, яка не містить рядків та стовпчиків. Перейти на крок 2).

2) Виконати крок 3) N раз, де N – довжина P. Перейти на крок 4).

3) Додати зліва в таблицю T рядок P, як стовпчик. Після чого відсортувати рядки таблиці T у порядку зростання.

4) Повернути K-тий рядок таблиці T.

Для прикладу розглянемо зворотнє перетворення для рядка “rroepp” з номером початкового рядка = 3. Таблиця 2.4 демонструє як змінюється таблиця T в процесі зворотнього перетворення: T0 – початкова порожня таблиця, Ti (i∈{1..6}) – таблиця, до якої додали вхідний рядок як стовпчик (позначений червоним кольором), Tis – відсортована таблиця Ti.

Табл. 2.4 - Демонстрація зворотнього перетворення Берроуза-Вілера.

T0	T1	T1s	T2	T2s	T3	T3s	T4	T4s	T5	T5s	T6	T6s
	r	e	re	ep	rep	epr	repr	epro	repro	eprop	reprop	epropr
	r	o	ro	op	rop	opr	ropr	opre	ropre	oprep	roprep	oprepr
	o	p	op	pr	opr	pre	opre	prep	oprep	prepr	oprepr	prepro
	e	p	ep	pr	epr	pro	epro	prop	eprop	propr	epropr	proper
	p	r	pr	re	pre	rep	prep	repr	prepr	repro	prepro	reprop
	p	r	pr	ro	pro	rop	prop	ropr	propr	ropre	propre	roprep

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ

3.1 Попередня обробка файлу

Основним недоліком використання стандартних універсальних методів для стиснення CSV-файлів є те, що вони не враховують табличну структуру файлу.

Через те, що в форматі CSV табличні дані зберігаються рядками, а не стовпчиками, часто різнотипні дані зберігаються підряд, що негативно впливає на ефективність стиснення. Методи компресії розглядають файл, як потік байтів, і цей потік для CSV файлів можна зобразити як на рисунку 3.1.

Якщо стовпці містять різнотипні дані, наприклад, стовпець 1 містить число, стовпець 2 – текстове посилання, а стовпець 3 – дату, то з точки зору стиснення, файл матиме не надто вдалу ентропію (міра хаотичності даних), що негативно вплине на ефективність стиснення. Крім того, для різних типів даних краще підходять різні методи стиснення.

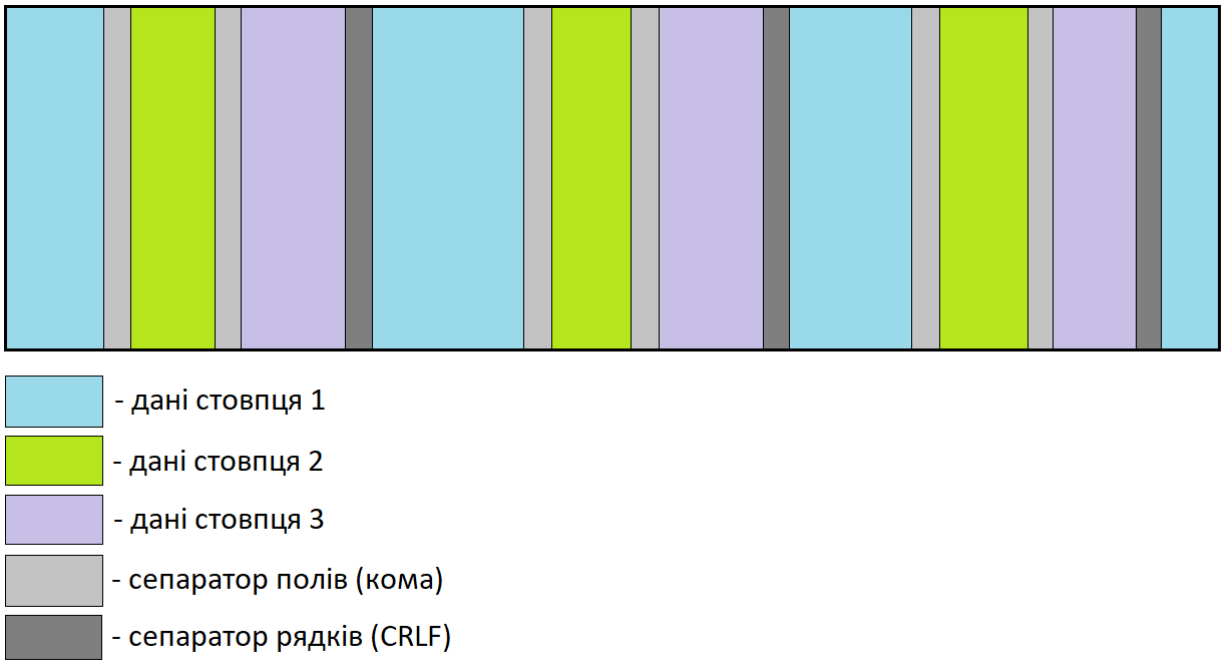

Рис. 3.1 – Зображення CSV файлу в пам’яті

В нашій розробці, для покращення ентропії даних (зменшення хаотичності) та заважаючи на табличну природу CSV формату, було вирішено використати колонко-орієнтований підхід. Тобто, до перетворення даних з ціллю стиснення, програма буде групувати їх по стовпчикам.

Проте, такий підхід потребує збереження значних обсягів даних в оперативній пам’яті, або запис їх в окремі файли, що не дуже мене влаштовує. Тому, групування буде відбуватися не по всьому файлу відразу, а по частинам: зчитується N рядків CSV файлу, дані групуються в стовпчики, а потім вже стискаються. Тоді дані в пам’яті можна зобразити як на рисунку 3.2.

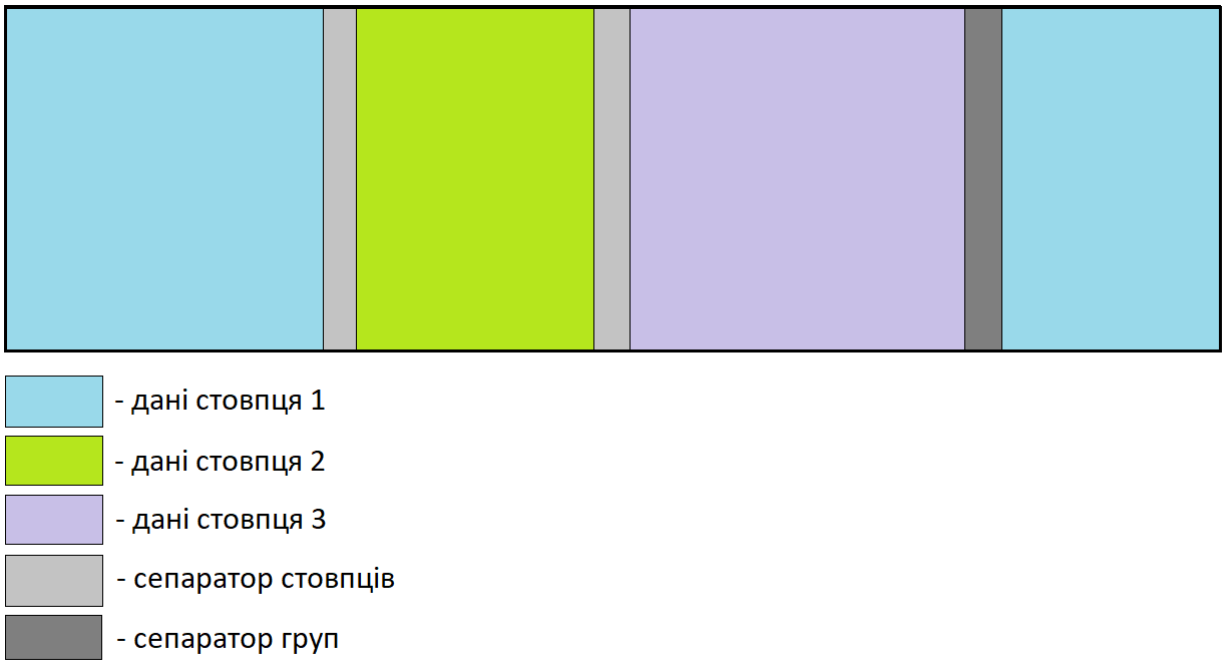

Рис. 3.2 – Зображення колонко-орієнтованих даних файлу в пам’яті.

Крім покращення ентропії даних, такий підхід дозволяє також використовувати різні підходи до стиснення даних для різних стовпчиків, що теж значно покращить ефективність стиснення.

3.2 Загальний алгоритм роботи програми

Основна ідея алгоритму програми полягає в тому, що для стиснення даних кожного зі стовпчиків файлу використовується певний метод стиснення, який найкраще підходить для його стиснення. Для цього необхідно проаналізувати вміст кожного стовпця, а вже потім виконувати його стиснення.

Загальний алгоритм роботи програми для стиснення файлу виглядає наступним чином:

1) Зчитати перший рядок вхідного CSV-файлу, який зберігає назви стовпчиків. Перейти до кроку 2.

2) Якщо не досягли кінця файлу, то КІНЕЦЬ, інакше зчитати M рядків поки файл не закінчиться, або M=N. Перейти до кроку 3.

3) Розбити отримані M рядків на колонки. Перейти до кроку 4.

4) Для кожної колонки проаналізувати її вміст та прийняти рішення яким методом краще всього стискати її вміст. Перейти до кроку 5.

5) Перетворити (стиснути) дані з кожної колонки відповідним методом. Перейти до кроку 6.

6) Для кожної колонки записати код метода та стиснуті (перетворені) дані в вихідний файл. Перейти до кроку 2.

Декомпресія файлу відбувається за наступним алгоритмом:

1) Зі стиснутого файлу зчитати назви стовпчиків. S – кількість стовпчиків. Перейти до кроку 2.

2) Якщо не досягли кінця файлу, то КІНЕЦЬ, інакше зчитати M – кількість рядків, яка закодована в поточному блоці. Перейти до кроку 3.

3) S раз зчитати код метода який відповідає методу стиснення, який кодує поточну колонку та зчитати дані, які кодують колонку. Перейти до кроку 4.

4) Декомпресувати зчитані дані відповідними методами декомпресії. Перейти до кроку 5.

5) Записати отримані декомпресовані дані в файл у форматі CSV (перетворити колонко-орієнтований файл в рядковий). Перейти до кроку 2.

Подані алгоритми є досить абстрактними та упускають чисельні деталі реалізації, тому розглядати їх варто лише для розуміння ідеї. Отже, алгоритм стискає табличні дані шляхом виділення груп (блоків) колонок певного розміру та застосування відповідних методів стиснення для кожної колонки групи окремо. Алгоритм декомпресії зчитує стиснуті групи колонок та застосовує відповідні методи декомпресії, таким чином відновлюючи файл.

3.3 Методи компресії колонок

Як зазначалося раніше, основна ідея програми полягає в застосуванні різних методів стиснення для різних стовпців таблиці, збереженої в CSV файлі. В програмі виділено 3 типи колонок, на основі належності до яких, буде прийматися рішення про застосування відповідного метода стискання:

1) Колонка зі значним числом повторів полів;
2) Колонка, яка містить числові дані;
3) Колонка, яка містить текстові дані;

Детально розглянемо яким чином визначається приналежність кожної колонки до одного з виділених типів та як відбувається стиснення кожного з типів колонок.

3.3.1 Стиснення колонок зі значним числом повторів

Часто на практиці виявляється, що стовпець таблиці містить поля з відносно невеликої множини значень і ці значення зустрічаються достатньо багато разів в одному стовпці. Наприклад, коли в таблиці дата розподілена по окремим стовпцям (окремі стовпці: день, місяць, рік), то кожний з цих стовпців містить значення з відносно невеликої множини (є 12 місяців, 31

день) і у випадку, коли таблиця містить велику кількість записів, одне і те ж значення (місць) повторюється в стовпчику багато разів.

В нашій розробці вирішено виділити такі стовпці в окремий тип, адже вони надзвичайно добре піддаються стисненню. Приналежність стовпця до типу колонок зі значним числом повторів відбувається шляхом побудови словника, де ключі – поля колонки, а значення – кількість повторів поля в колонці. Якщо розмір словника менший за кількість полів колонки в N разів, то відносимо колонку до цього типу.

Для стиснення колонок зі значним числом повторів був обраний метод кодування Хаффмана, який відрізняється від класичного тим, що замість символів ми оперуємо полями стовпця. Тобто стиснення відбувається в 3 етапи:

1) Побудова таблиці частоти входження значення у стовпець;

2) На основі побудованої таблиці частот побудова дерева Хаффмана;

3) Кодування колонки за допомогою побудованого дерева Хаффмана;

Таким чином, в результаті стиснення ми маємо зберегти таблицю частот появи поля в таблиці та самі коди Хаффмана, які дозволяють виконати декомпресію даних.

Декомпресія даних, стиснутих за допомогою цього методу, відбувається аналогічно алгоритму декомпресії методом Хаффмана, описаному в попередньому розділі.

При значній кількості повторів у стовпці такий підхід дозволяє дуже ефективно стиснути дані. Проте, при малій кількості повторів такий підхід призведе лише до зростання обсягу зкомпресованих даних порівняно з вхідними. Тому необхідно приділити увагу визначенню параметра N, який відіграє вирішальну роль у віднесенні стовпця до типу колонок зі значним числом повторів, так щоб компресія була ефективною.

Для визначення параметра N розглянемо невдалу для стиснення ситуацію: Нехай в колонці 64000 полів (розмір блока в програмі), середня довжина поля 2 символи, тоді для збереження цієї колонки в CSV файлі

витрачається 64000*(2+1)=192000 байт (2+1, оскільки необхідно зберігати сепаратор(кому)); Для збереження цієї колонки за допомогою описаного вище метода необхідно (64000/N)*(2+2) + 64000*([log_2(64000/N)]+1)/8 (таблиця частот + коди); Якщо встановити N=10, то (64000/N)*(2+2) + 64000*([log_2(64000/N)]+1)/8 = 25600 + 104000 = 126000 байт. Отже, при N=10 вдалося отримати стиснуті дані, менші вхідних приблизно на 30%.

Розглянута ситуація є одною з найбільш невдалих для стиснення і дозволяє оцінити мінімальне значення параметра N, яке дозволить завжди отримувати допустиму ефективність компресії.

На практиці, середня довжина поля більша, що значно підвищує ефективність стиснення таких колонок. Наприклад, є стовпчик таблиці, який зберігає імена людей (множина імен відносно невелика) та містить 1000 унікальних імен. Якщо припустити, що середня довжина імені = 5, а колонка містить 64000 полів, то розмір вхідних даних можна оцінити як 384000 байт, а розмір стиснутих даних вийде близько 71000 байт.

Отже, до колонок зі значною кількістю повторів будемо відносити ті стовпчики, в яких кількість унікальних полів стовпчика менша за кількість полів цього стовпчика більш ніж в 10 разів. Для стиснення стовпчиків цього типу будемо користуватися методом кодування Хаффмана для полів.

3.3.2 Стиснення колонок, які містять числові дані

Для стиснення колонок, які містять числові дані існує декілька підходів. Самим очевидним та нескладним підходом є конвертація числових даних в базові числові типи даних і запис даних в цих типах у файл. В теорії, цей підхід до стиснення має непогано працювати, адже в CSV фалі нема різниці між текстовими та числовими даними і вони зберігаються як текст, що не надто ефективно.

Таким чином, логічно виділити наступні типи:

1) unsigned short – беззнаковий цілочисельний тип [0,65534], займає 2 байти.

2) int – знаковий цілочисельний тип [$-2.14*10^9-1$; $+2.14*10^9$], займає 4 байти.

3) float – знаковий числовий тип з рухомою комою, 6-9 (частіше всього 7) чисел після коми, займає 4 байти.

4) double – знаковий числовий тип з рухомою комою, 15-18 (частіше всього 16) чисел після коми, займає 8 байт.

Одною з особливостей CSV файлів є наявність відсутності значення. Для позначення відсутності значення в цьому підході використовується мінімальне значення кожного з типів.

Під час реалізації цього підходу був помічений ряд недоліків. По-перше, в процесі переведення текстового числа з рухомою комою втрачається або викривлюється точність числа. Наприклад, число з 20 знаками після коми не зможе бути відновлено, декілька останніх чисел загубляться.

По-друге, під час збереження чисел з рухомою комою, які мають точність 2 знаки після коми, дані відновляться до чисел з 6 знаками після коми. Для виправлення цієї проблеми необхідно додатково витрачати цінну пам’ять на збереження додаткової інформації про точність стиснутого числа, або відмовитися від однозначного відновлення числових даних, що унеможливить компресію даних без втрат.

По-третє, такий спосіб в деяких випадках не є надто ефективним з точки зору стиснення. Наприклад, для збереження стовпчика (1000 полів) з числами у форматі dd.dd в CSV файлі витрачається 1000(5+1) байт (1 байт на сепаратор (кому)), а в стиснутому вигляді ці ж дані займають 1000*4 байт (не враховуючи пам’ять необхідну для збереження точності числа). Тобто компресія виявилася не надто ефективною (стиснуті дані займають на 40% менше пам’яті).

Отже, такий підхід має ряд недоліків, які значно ускладнюють реалізацію та можуть призвести до компресії даних з втратами у випадку типів з рухомою комою. Проте, для стиснення цілочисельних типів такий підхід цілком підходить. Крім того, він забезпечує непогану ефективність стиснення. Наприклад, колонка має 1000 полів та зберігає рік випуску студента, тоді ця колонка у вхідному CSV файлі займає 1000*(4+1) байт, а в зкомпресованому вигляді всього 1000*2 байт, що в 2.5 разів менше розміру не стиснутих даних.

Для стиснення числових даних з рухомою комою кращим виявився підхід, який стискає дані шляхом присвоєння цифрам та допоміжним символам кодів довжиною 4 біти. Кодування відбувається за таблицею 3.1.

Табл. 3.1 – Коди символів для кодування числових даних

Символ	Код	Символ	Код
0	0000	8	1000
1	0001	9	1001
2	0010	/	1010
3	0011	.	1011
4	0100	‘ ‘(прогалина)	1100
5	0101	:	1101
6	0110	,	1110
8	0111	-	1111

Так як цей підхід заміняє символи, які займають 8 біт, кодами, які займають 4 біти. Порівняно з попереднім підходом, такий підхід завжди гарантує непогану ефективність компресії (дані стовпця у вхідному файлі займають в 2 рази більше місця ніж в стиснутому). Крім того, зникають проблеми зі збереженням точності чисел з рухомою комою та однозначним їх відновленням (відбувається стиснення без втрат).

Крім цифр, коми (необхідна як “природний” для CSV файлів сепаратор) та крапки, таблиця кодування також містить деякі інші символи, які роблять можливим стиснення цим методом також колонок, які зберігають дату та час у числовому форматі, наприклад у форматі дд.мм.рррр гг:хх (20.04.2021 01:20).

Таким чином, в моїй розробці тип колонок, які містять числові дані був розбитий на 3 підтипи:

1) дані у форматі unsigned short;
2) дані у форматі int;
3) дані у загальному числовому форматі;

Для стиснення стовпців, які відносяться до першого або другого підтипу, доцільно використовувати перший підхід (конвертація числових даних в базові числові типи). Для стиснення стовпців, які відносяться до третього підтипу, доцільно використовувати другий підхід (кодування символів кодами довжини 4 біт).

3.3.3 Стиснення колонок, які містять текстові дані

До типу колонок, які містять текстові дані, можна віднести будь-яку колонку CSV файлу, адже він містить дані у форматі тексту. Проте, в контексті даної розробки, до типу колонок, які містять текстові дані, будемо відносити ті колонки, які не є колонками зі значним числом повторів та колонками, які зберігають числові дані.

Для стиснення таких колонок в розробці запропоновано 2 підходи. Перший підхід полягає у використанні методу стиснення Лемпеля-Зіва-Велча для стиснення даних колонки як потоку символів. Стиснення відбувається за рахунок кодування підпослідовностей байтів вхідних даних.

Такий підхід чудово підходить при наявності великої кількості повторів підпослідовностей символів у вхідних даних. Проте, коли у даних колонок невдала ентропія (дані хаотичні, підпослідовності відсутні), такий підхід може

призвести до зростання обсягу скомпресованих даних, порівняно з вхідними. Але, на практиці, випадки слабкої ентропії даних в межах однієї колонки – достатньо малоймовірна ситуація.

З ціллю зменшення хаотичності даних можливо користуватися перетворенням Берроуза-Вілера для передобробки даних до стиснення методом LZW. Проте, часто у випадку коли хаотичність даних низька, така передобробка може лише призвести до погіршення ентропії даних, що в свою чергу негативно повпливає на ефективність стиснення. Тому, через описану причину, в розробці така передобробка не використовується.

Вирішувати проблему переповнення словника будемо шляхом ігнорування переповнення, адже число полів колонки (в блоці) є досить невеликим та переповнення словника настає наприкінці процесу стиснення даних колонки. А у випадку перезавантаження словника ефективність стиснення тимчасово знижується, що негативно вплине на загальну ефективність стиснення файлу.

Другий підхід до стиснення даних колонок цього типу полягає у стисненні даних шляхом класичного кодування Хаффмана. Стиснення відбувається за рахунок присвоєння символам(байтам), які часто зустрічаються в даних колонки, більш коротких кодів.

Такий підхід вимагає попереднього аналізу частот входження символів в поля колонки шляхом побудови таблиці частот, яка необхідна для побудови дерева Хаффмана. Основним недоліком такого підходу є те, що для стиснення необхідно виконати 2 проходи по даним колонки (перший раз для побудови таблиці частот, другий для заміни символів на коди), що негативно вплине на продуктивність програми. Проте, розробка в першу чергу націлена на забезпечення високої ефективності стиснення, та перший підхід теж вимагає значних системних ресурсів, тому такий недолік не є надто важливим.

На відміну від першого, такий підхід менш чутливий до хаотичності даних, проте для даних з гарною ентропією він не показує таку ж хорошу ефективність стиснення, як перший підхід.

Отже, в силу того що обидва підходи є більш або менш ефективними в конкретних ситуаціях (залежить від специфіки даних збережених у файлі), в розробці реалізовано обидва підходи та користувачу надано можливість самому вирішувати який підхід варто використовувати для вирішення задачі стиснення конкретного CSV файлу.

3.4 Реалізація програми

Для реалізації програми-компресора файлів у форматі CSV була обрана мова програмування C++ (стандарту C++11). В процесі реалізації активно використовувалися такі структури даних: std::vector, std::map, std::queue, що на думку автора повинно сприяти швидкодії. Для протидії можливим витокам пам’яті (memory leak) в програмі використовуються розумні вказівники (smart pointers). Деякі фрагменти реалізації подані в додатку А-Д.

3.4.1 Класи, які описують логіку роботи програми

Загальний алгоритм стиснення був реалізований шляхом розробки низки класів, які реалізують логіку програми. Всі класи, реалізовані в програмі можна умовно розбити на 2 групи: класи, які використовуються для компресії файлу; класи, які використовуються для декомпресії файлу.

На рисунку 3.3 зображена діаграма класів, які використовуються для виконання компресії файлу. Розглянемо призначення кожного з класів на діаграмі:

- HuffLeafStr – клас, який описує листок дерева Хаффмана для рядкового типу.
- HuffLeafCh – клас, який описує листок дерева Хаффмана для символьного типу.
- RecordParser – статичний клас, який містить методи необхідні для обробки та формування записів у форматі CSV.

- ColumnCompressor – абстрактний клас, батьківський для всіх класів, які реалізують метод стиснення колонок та запис стиснутих даних.
- HuffmanSCoder – клас, успадкований від ColumnCompressor, призначений для стиснення колонок зі значним числом повторів полів, та запису стиснутих даних у файл.
- IntTypeCoder – клас, успадкований від ColumnCompressor, призначений для стиснення колонок, які містять числові дані у форматі int, та запису стиснутих даних у файл.
- UShortTypeCoder – клас, успадкований від ColumnCompressor, призначений для стиснення колонок, які містять числові дані у форматі unsigned short, та запису стиснутих даних у файл.
- LZWCoder – клас, успадкований від ColumnCompressor, призначений для стиснення колонок, які містять текстові дані за допомогою метода LZW, та запису стиснутих даних у файл.
- NumericCoder – клас, успадкований від ColumnCompressor, призначений для стиснення колонок, які містять числові дані у загальному числовому форматі, та запису стиснутих даних у файл.
- HuffmanKCoder – клас, успадкований від ColumnCompressor, призначений для стиснення колонок, які містять текстові дані за допомогою метода Хаффмана, та запису стиснутих даних у файл.
- ColumnAnalyzer – клас, призначений для аналізу вмісту колонки для прийняття рішення про приналежність даних колонки до певного типу (вибору метода для стиснення даних колонки) та збору даних (які збираються у процесі аналізу), необхідних для подальшого стиснення цієї колонки.
- CSVBlockCompressor – клас, який використовується для читання частини файлу, виконання перетворення даних в колонко-орієнтовані, аналізу виділених колонок (використовує об’єкти

класу ColumnAnalyzer), стиснення даних колонок та запису стиснутих даних (використовує об’єкти класів-нащадків класу ColumnCompressor).

- FileCompressor – клас, призначений для стиснення файлу по частинам, використовуючи об’єкти класу CSVBlockCompressor.

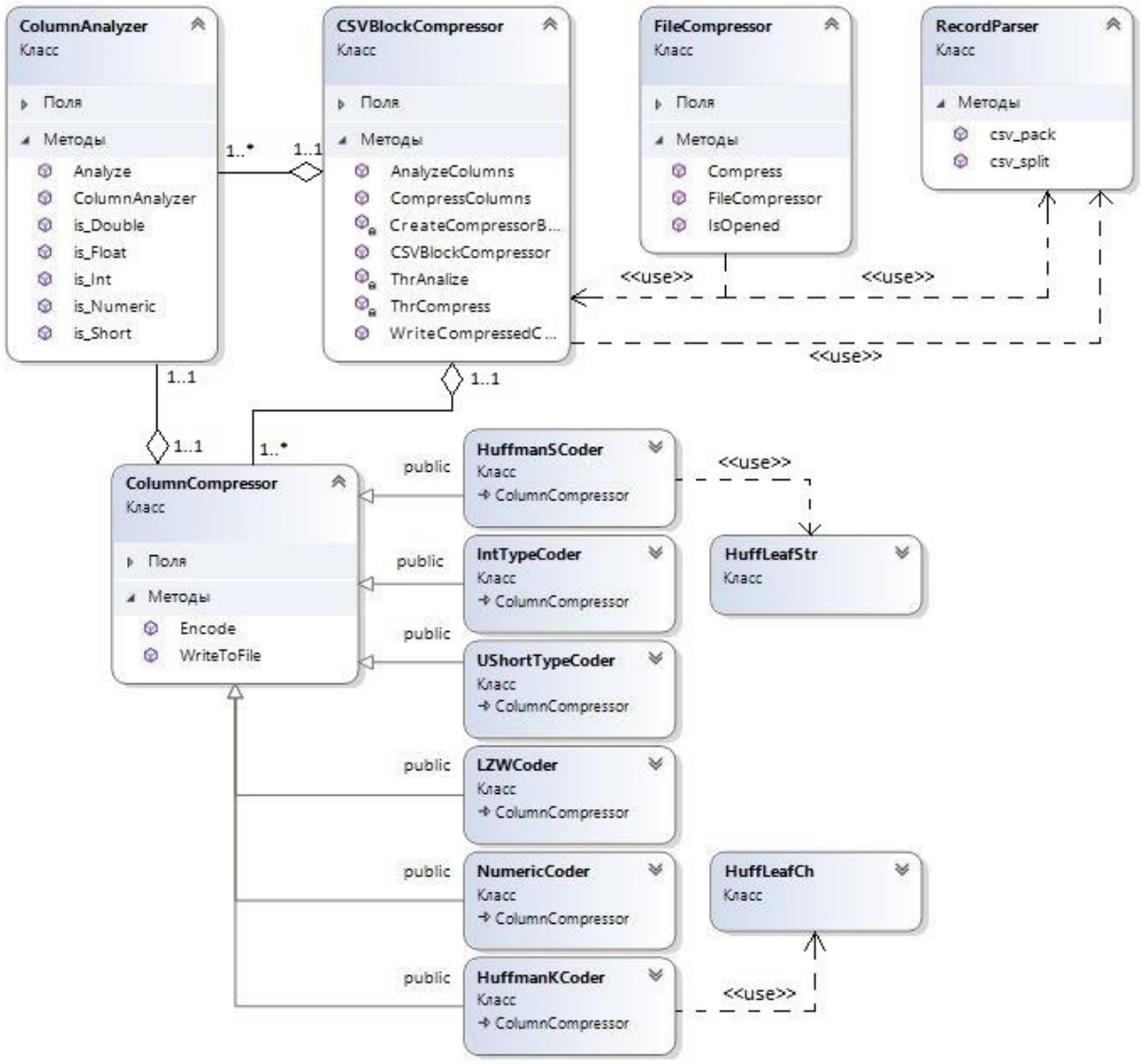

Рис. 3.3 – Діаграма класів, які використовуються для стиснення файлу.

На рисунку 3.4 зображена діаграма класів, які використовуються для виконання декомпресії файлу. Розглянемо призначення кожного з класів на діаграмі (класи HuffLeafStr, HuffLeafCh та RecordParser вже були розглянуті):

- ColumnDecompressor – абстрактний клас, батьківський для всіх класів, які реалізують зчитування стиснутих даних колонки та декомпресію зчитаних даних.
- HuffmanSDecoder – клас, успадкований від ColumnDecompressor, призначений для зчитування даних колонки стиснутих, використовуючи HuffmanSCoder, та виконання їх декомпресії.
- IntTypeDecoder – клас, успадкований від ColumnDecompressor, призначений для зчитування даних колонки стиснутих, використовуючи IntTypeCoder, та виконання їх декомпресії.
- UShortTypeDecoder – клас, успадкований від ColumnDecompressor, призначений для зчитування даних колонки стиснутих, використовуючи UShortTypeCoder, та виконання їх декомпресії.
- LZWDecoder – клас, успадкований від ColumnDecompressor, призначений для зчитування даних колонки стиснутих, використовуючи LZWCoder, та виконання їх декомпресії.
- NumericDecoder – клас, успадкований від ColumnDecompressor, призначений для зчитування даних колонки стиснутих, використовуючи NumericCoder, та виконання їх декомпресії.
- HuffmanKDecoder – клас, успадкований від ColumnDecompressor, призначений для зчитування даних колонки стиснутих, використовуючи HuffmanKCoder, та виконання їх декомпресії.
- CSVBlockDecompressor – клас, який використовується для зчитування блоку колонок, виконання їх декомпресії (використовує об’єкти класів-нащадків класу ColumnDecompressor) та запису відновлених даних у вихідний файл.
- FileDecompressor – клас, призначений для виконання декомпресії файлу по частинам, використовуючи об’єкти класу CSVBlockDecompressor.

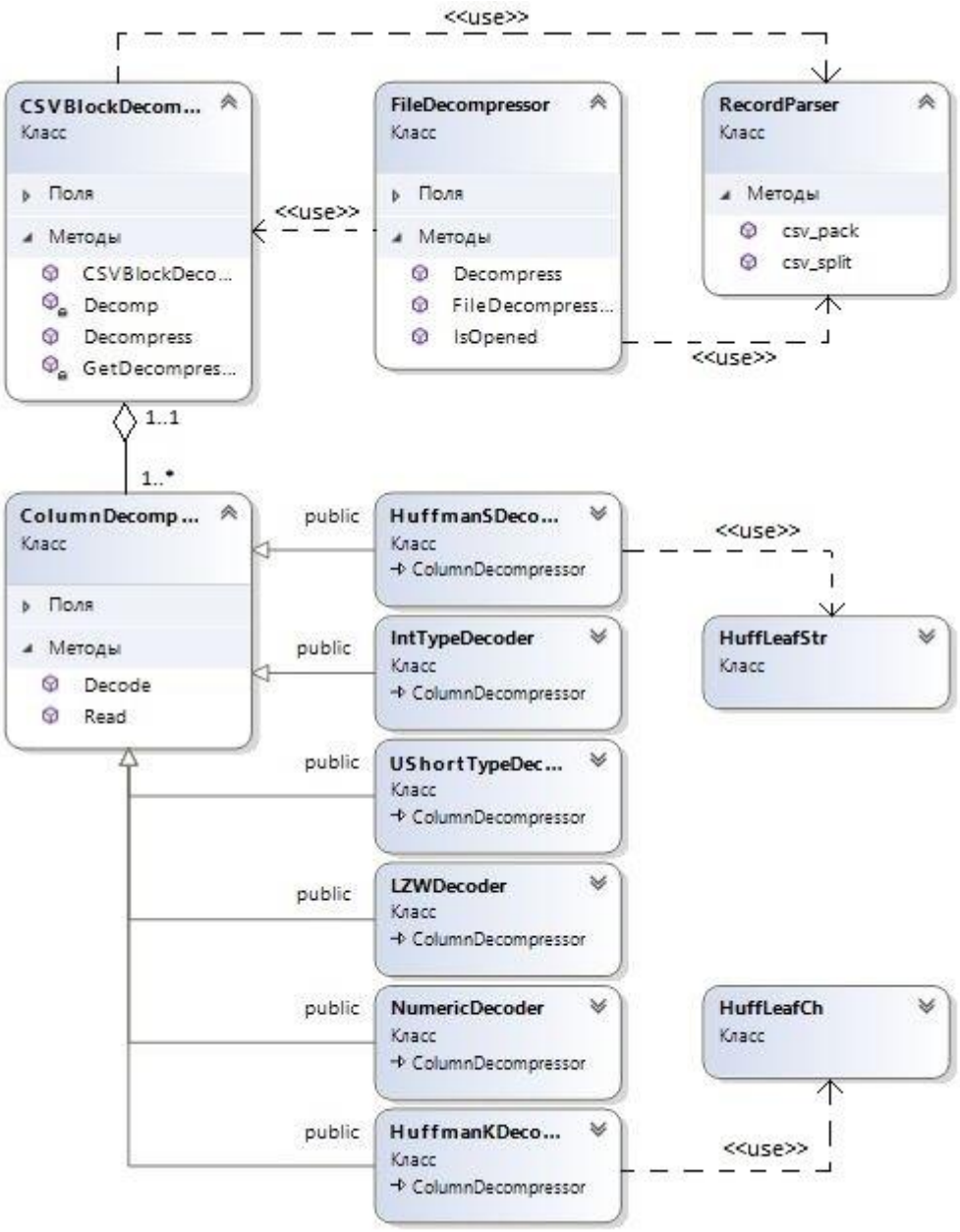

Рис. 3.4 – Діаграма класів, які використовуються для декомпресії файлу

3.4.2 Оптимізація шляхом паралельного виконання

Хоч і основна задача програми – забезпечити високу степінь стиснення, трішки уваги варто приділити швидкодії. З ціллю підвищення продуктивності програми, було прийнято рішення про використання багатопоточності для паралельного виконання деяких частин програми.

Зважаючи на реалізацію та архітектуру програми, існує декілька місць, де можна було б застосувати виконання програми на декількох потоках:

виконання аналізу даних колонки, стиснення даних колонки, декомпресія даних колонки.

Етап аналізу колонки, незалежно від її типу, проходить для всіх колонок приблизно з однаковою швидкістю (однак, не з однаковою), що робить цю частину перспективною для розпаралелювання. Тому, в розробці було реалізовано виконання аналізу колонок на відповідність одному з виділених типів на окремих потоках (для кожної колонки свій потік). Для реалізації було використано стандартний інструмент C++ std::thread

Натомість, етап стиснення або декомпресії даних колонок витрачає різну кількість системних ресурсів для колонок різних типів (наприклад, колонка, яка містить числові дані типу int буде стискатися та відновлюватися в сотні разів швидше, аніж колонка, яка містить текстові дані). Тому від виконання компресії (декомпресії) даних колонок було вирішено відмовитися.

3.4.3 Розробка інтерфейсу програми

Для розробки інтерфейсу користувача спочатку передбачалося використати технологію C++/CLI[23] та Windows Forms[24]. Проте, на жаль, компіляція у середовищі CLR не підтримує клас std::thread, який був використаний для реалізації часткового розпаралелення програми. Тому від використання цієї технології довелося відмовитися.

Інтерфейс програми був розроблений за допомогою інтегрованого середовища розробки QT Creator[1] та з використанням крос-платформного інструментарію розробки програмного забезпечення QT framework[2]. Розроблений інтерфейс показаний на рисунку 3.5. Інтерфейс є достатньо простим і інтуїтивно зрозумілим, не містить нічого зайвого, що мого б відвернути уваги від основного функціоналу.

З кодом програми можна ознайомитися на сторінці репозиторію[32], розміщеному на Github.

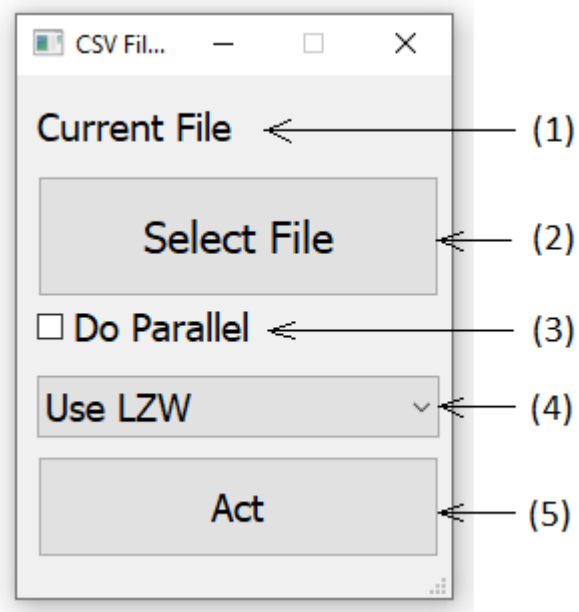

Рис. 3.5 – Інтерфейс програми.

3.5 Інструкція до використання програми

Для стиснення файлу у форматі CSV користувачу спочатку необхідно переконатися, що файл, який треба стиснути, дотримується правил стандарту RFC4180[]. У випадку недотримання цих правил, дані у файлі можуть бути відновлені з втратами інформації.

Якщо файл відповідає вимогам, то для його стиснення користувачу необхідно виконати ряд кроків:

1) Натиснути кнопку “Select File” ((2) на рисунку 3.5), в результаті чого виникне стандартне системне вікно вибору файлу з застосованим фільтром відображення папок та файлів у форматі CSV.

2) У відкритому стандартному системному вікні вибрати потрібний CSV файл. Після вибору файлу його назва буде відображена в текстовому полі ((1) на рисунку 3.5) та текст кнопки “Act” ((5) на рисунку 3.5) зміниться на “Compress”.

3) Встановити необхідні параметри за допомогою елементів (3) та (4) на рисунку 3.5 (чи застосовувати паралельне виконання аналізу колонок, та

який метод використовувати для стиснення колонок, які містять текстові дані).

4) Натиснути кнопку “Compress” ((5) на рисунку 3.5), після чого програма почне стиснення обраного файлу. Після завершення стиснення, з’явиться вікно як на рисунку 3.6, яке сповістить про успішне завершення стиснення, або про помилки, які не дали завершити процес стиснення (наприклад, невідповідність файлу вимогам).

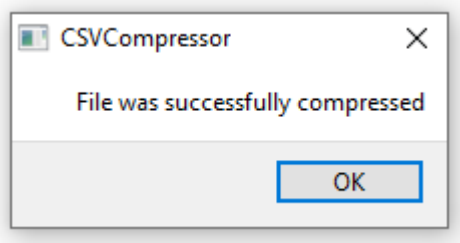

Рис. 3.6 – Сповіщення про успішне завершення стиснення.

5) У разі успішного стиснення, стиснуті дані будуть збережені у файл з розширенням .ccsv, з тою ж назвою та розташуванням що у вхідного CSV файлу.

Дані, стиснуті програмою можна відновити (виконати декомпресію) за допомогою виконання наступних кроків:

1) Натиснути кнопку “Select File” ((2) на рисунку 3.5), в результаті чого виникне стандартне системне вікно вибору файлу з застосованим фільтром відображення папок та файлів у форматі CSV. Для відображення файлів, які зберігають стиснуті дані, внизу вікна вибору файлу необхідно змінити поточний фільтр файлів на “Compressed csv (*.ccsv)” (див. рисунок 3.7).

2) У відкритому стандартному системному вікні вибрати потрібний CCSV файл. Після вибору файлу його назва буде відображена в текстовому полі ((1) на рисунку 3.5) та текст кнопки “Act” ((5) на рисунку 3.5) зміниться на “Decompress”.

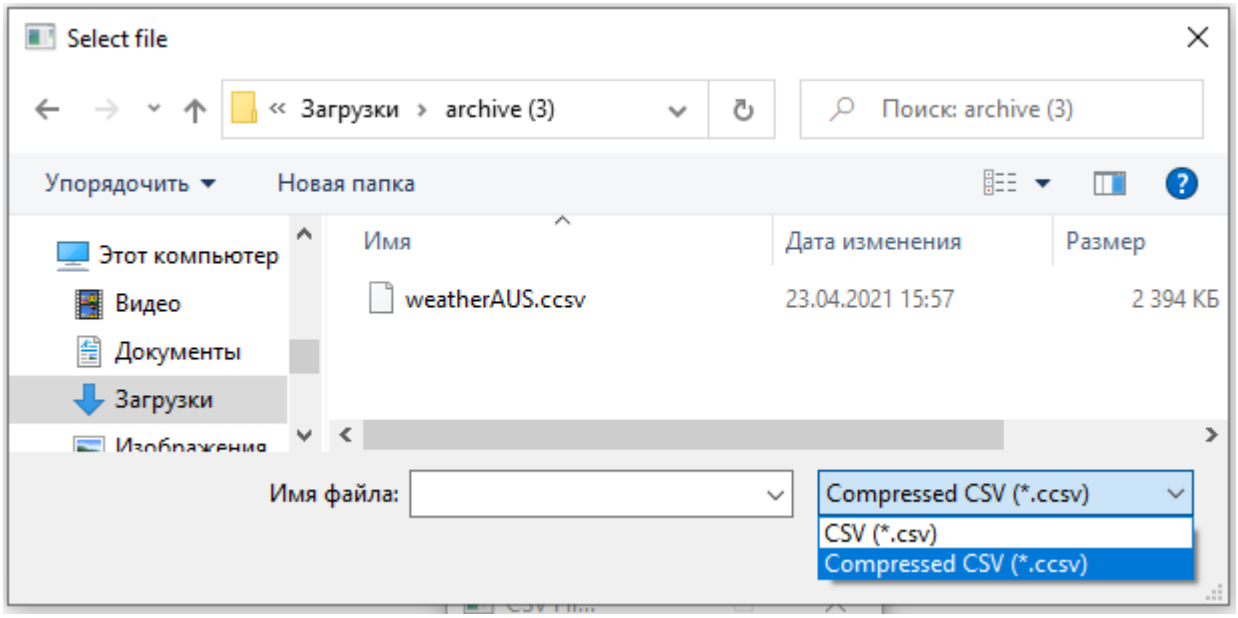

Рис. 3.7 – Системне вікно вибору файлу.

4) Натиснути кнопку “Decompress” ((5) на рисунку 3.5), після чого програма почне виконувати декомпресію обраного файлу. Після завершення декомпресії, з’явиться вікно як на рисунку 3.8, яке сповістить про успішне завершення, або про помилки, які не дали завершити процес декомпресії.

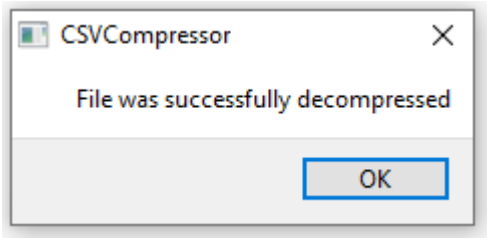

Рис. 3.8 – Сповіщення про успішне завершення декомпресії.

5) У разі успішної декомпресії, відновлені дані будуть збережені у файл з розширенням .csv, з тою ж назвою та розташуванням як у файлу, що містив стиснуті дані.

3.6 Тестування ефективності програми.

Для вимірювання ефективності стиснення CSV файлів за допомогою розробленої програми було проведено ряд тестів на різних даних. Результати ефективності стиснення даних використовуючи розроблену програму були порівнянні з результатами стиснення цих же даних за допомогою стандартного вбудованого в систему Windows 10 [25] ZIP-архіватора та популярного архіватора WinRar[34].

Для тестування програми на ефективність стиснення були обрані файли у форматі CSV, опис яких подано в таблиці 3.1.

Табл. 3.1 – Файли у форматі CSV, обрані для тестування програми.

Назва	Опис вмісту	Розмір	Джерело
steam-200k.csv	Дані про поведінку користувачів по відношенню до куплених ігор на платформі Steam.	8958 Кб	[28]
acs2015_census_tract_data.csv	Містить демографічні дані США за 2015 рік.	14448 Кб	[29]
Jan_2019_ontime.csv	Містить дані, зібрані урядом США про більш як 400 тисяч польотів протягом січня 2019 року.	76028 Кб	[31]
DataCoSupplyChainDataset.csv	Містить дані про доставку замовлень з різних країн.	95910 Кб	[30]
accidents_2012_to_2014.csv	Дані уряду Великобританії про автомобільні аварії за 2014-2016 роки.	133627 Кб	[26]
city_temperature.csv	Містить дані про середню температуру в різних містах за 2020 рік.	140600 Кб	[27]

Обрані для тестування файли були стиснуті за допомогою утиліти ZIP, використовуючи розроблену програму в режимі використання методу LZW та метод Хаффмана для стиснення колонок, які містять текстові дані. Результати тестування подані в таблиці 3.2 (розмір файлів зазначений в кілобайтах, колонки Prog (X) містять результати стиснення файлів за допомогою розробленої програми в режимі використання методу X).

Табл. 3.2 – Результати стиснення файлів

Файл	Розмір файлу до стиснення	Prog (LZW)	Prog (Huffman)	ZIP	RAR
steam-200k.csv	8958	1139	1147	1541	1149
acs2015_census_tract_data.csv	14448	4283	4 283	5301	4903
Jan_2019_ontime.csv	76028	10120	10120	11971	9061
DataCoSupplyChainDataset.csv	95910	13967	15175	18809	12775
accidents_2012_to_2014.csv	133627	15412	17480	17 552	15366
city_temperature.csv	140600	11042	11042	14410	12554

Спостерігається, що ефективність програми краща при використанні метода LZW (колонка Prog (LZW) таблиці 3.2) для стиснення колонок, які містять текстові дані. Деякі файли програма, використовуючи метод LZW та Хаффмана стиснула до однакового розміру. Це зумовлено тим, що ці файли не містили колонок, які були б віднесені до типу колонок, які містять текстові дані, і ці методи компресії у стисненні файлу не приймали участі.

Відповідно результатам тестування, розроблена програма показала кращу ефективність стиснення порівняно з утилітою ZIP. Але при цьому, порівняно з ZIP та RAR, розроблена програма має ряд недоліків. По-перше, розроблена програма працює лише з CSV файлами, які створені з дотриманням

правил стандарту RFC4180[7]. По-друге, розроблена програма витрачає значно більше часу та системних ресурсів, аніж ZIP чи RAR.

Виявилося, що для 3 з 6 тестових файлів RAR-архіватор показав трохи кращу ефективність стиснення, ніж розроблена програма. В основному це спостерігається для файлів, які мають значну кількість колонок, які містять текстові дані. Таким чином, ефективність програми може бути поліпшена шляхом покращення реалізації методів стиснення та застосуванням більш досконалих методів стиснення для колонок, які містять текстові дані.

Для ще більшого підвищення ефективності стиснення можна застосовувати один з універсальних методів компресії для компресії файлу, стиснутого розробленою програмою. Наприклад, зкомпресований розробленою програмою файл можна додатково стиснути одним з архіваторів.

В таблиці 3.3 подано відношення стиснення (відношення розміру вхідних даних до розміру скомпресованих даних), для тестових файлів. Загалом, результати, подані в табличці 3.3, демонструють ефективність обраного підходу до стиснення файлів у форматі CSV.

Табл. 3.3 – Відношення стиснення для файлів.

Файл	Prog (LZW)	Prog (Huffman)	ZIP	RAR
steam-200k.csv	786,47%	780,99%	581,31%	779,63%
acs2015_census_tract_data.csv	337,33%	337,33%	272,55%	294,67%
Jan_2019_ontime.csv	751,26%	751,26%	635,10%	839,06%
DataCoSupplyChainDataset.csv	686,69%	632,02%	509,91%	750,76%
accidents_2012_to_2014.csv	867,03%	764,45%	761,32%	869,62%
city_temperature.csv	1273,32%	1273,32%	975,71%	1119,96%

ВИСНОВКИ

В даній роботі було детально розглянуто формат CSV, його специфіку, основні переваги та недоліки. Був виконаний загальний огляд методів компресії даних. Детально були розглянуті такі методи стиснення даних без втрат як кодування довжин серій (RLE), метод Хаффмана, метод Лемпеля-Зіва-Велча (LZW) та перетворення Берроуза-Вілера (BWT).

Був розроблений алгоритм для стиснення файлів у форматі CSV, який враховує особливості даних, збережених в цьому форматі. Була реалізована програма-компресор CSV-файлів, яка працює на базі розробленого алгоритму.

Розроблена програма для компресії CSV-файлів була протестована на ряді даних. Результати тестування були порівняні з результатами стиснення цих самих даних, використовуючи вбудований в систему Windows 10 [25] ZIP-архіватор та архіватор WinRar[34].

В результаті тестування виявилося, що для всіх тестових даних розроблена програма забезпечила краще відношення стиснення ніж ZIP-архіватор. Для файлів, які мають значну кількість колонок, які містять текстові дані, краще відношення стиснення продемонстрував RAR-архіватор, проте для файлів, в яких кількість таких колонок незначна, розроблена програма виявилася більш ефективною.

Отже, результати тестування показують доцільність використання розробленої програми для стиснення файлів у форматі CSV. Максимальну ефективність стиснення програма демонструє для файлів, які мають незначну кількість колонок, які містять текстові дані.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. QT Creator [Електронний ресурс] – Режим доступу до ресурсу: https://www.qt.io/product/development-tools
2. QT framework [Електронний ресурс] – https://www.qt.io/product/framework
3. WinZip [Електронний ресурс] – https://www.winzip.com/win/en/
4. Gzip [Електронний ресурс] – https://www.gnu.org/software/gzip/
5. 7-zip [Електронний ресурс] – https://www.7-zip.org/
6. Git [Електронний ресурс] – https://git-scm.com/
7. RFC4180 [Електронний ресурс] – https://tools.ietf.org/html/rfc4180#section-1
8. RFC7111 [Електронний ресурс] – https://tools.ietf.org/html/rfc7111
9. Kaggle [Електронний ресурс] – https://www.kaggle.com/
10. How to compress CSV file efficiently [Електронний ресурс] – https://abhisheksingh007226.medium.com/how-to-compress-csv-file-efficiently-in-just-25-lines-of-code-4e974b60b2b8
11. Big Data File Formats Explained [Електронний ресурс] – https://towardsdatascience.com/big-data-file-formats-explained-dfaabe9e8b33
12. CSV FORMAT: HISTORY, ADVANTAGES AND WHY IT IS STILL POPULAR [Електронний ресурс] – https://bytescout.com/blog/csv-format-history-advantages.html#5
13. Big Data file formats [Електронний ресурс] – https://luminousmen.com/post/big-data-file-formats
14. The Basic Principles of Data Compression [Електронний ресурс] – Режим доступу до ресурсу: https://www.2brightsparks.com/resources/articles/data-compression.html
15. The History of Data Compression [Електронний ресурс] – Режим доступу до ресурсу: http://techmeup.net/history-data-compression-infographic/

16. Data compression - Undergraduate study in Computing and related programmes [Електронний ресурс] – Режим доступу до ресурсу: https://london.ac.uk/sites/default/files/study-guides/data-compression.pdf
17. Lossy vs Lossless Image Compression programmes [Електронний ресурс] – Режим доступу до ресурсу: https://flat-icons.com/lossy-vs-lossless/
18. АЛГОРИТМЫ СЖАТИЯ [Електронний ресурс] – Режим доступу до ресурсу: http://mf.grsu.by/UchProc/livak/po/comprsite/theory_contents.html
19. LZW (Lempel–Ziv–Welch) Compression technique [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/
20. Общие алгоритмы сжатия и кодирования [Електронний ресурс] – Режим доступу до ресурсу: http://algolist.ru/compress/standard/
21. Кодирование и сжатие информаци [Електронний ресурс] – Режим доступу до ресурсу: https://studfile.net/preview/1757716/
22. The Data Compression Guide [Електронний ресурс] – Режим доступу до ресурсу: https://sites.google.com/site/datacompressionguide/
23. C++/CLI [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/cpp/dotnet/dotnet-programming-with-cpp-cli-visual-cpp?view=vs-2019
24. Windows Forms [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/framework/winforms/
25. Windows [Електронний ресурс] – https://www.microsoft.com/uk-ua/windows
26. accidents_2012_to_2014.csv [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/daveianhickey/2000-16-traffic-flow-england-scotland-wales?select=accidents_2012_to_2014.csv
27. city_temperature.csv [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities
28. steam-200k.csv [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/tamber/steam-video-games

29. acs2015_census_tract_data.csv [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/muonneutrino/us-census-demographic-data?select=acs2015_census_tract_data.csv
30. DataCoSupplyChainDataset.csv [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/shashwatwork/dataco-smart-supply-chain-for-big-data-analysis
31. Jan_2019_ontime.csv [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/divyansh22/flight-delay-prediction?select=Jan_2019_ontime.csv
32. Git-репозиторій проекту [Електронний ресурс] – https://github.com/skanerOH/Diploma_and_Inerface
33. WinRar [Електронний ресурс] – https://www.win-rar.com/start.html?&L=4
34. Parquet [Електронний ресурс] – https://parquet.apache.org/

ДОДАТКИ

Додаток А

```
#include "LZWCoder.h"

LZWCoder::LZWCoder(const std::shared_ptr<ColumnAnalyzer> &colAnal) {
    this->colAnal = colAnal;
    res = std::vector<unsigned short>();
    InitDict();
}

void LZWCoder::InitDict()
{
    dict = std::map<std::string, unsigned short>();
    for (char i=CHAR_MIN; i!=CHAR_MAX; ++i)
    {
        dict[std::string(1, i)]=dict.size();
    }
}

std::string LZWCoder::GetStrByCode(unsigned short code)
{
    for (auto &i : dict)
    {
        if (i.second == code)
            return i.first;
    }
    return "";
}

void LZWCoder::Encode() {
    std::vector<char> chars = std::vector<char>();

    for (auto str = colAnal->columnValues->begin(); str != colAnal-
>columnValues->end(); ++str) {
        for (auto c = str->begin(); c != str->end(); ++c) {
            chars.emplace_back(*c);
        }
        chars.emplace_back(',');
    }

    chars.pop_back();
    std::string curr;
    std::string priv;
    for (unsigned int i = 0; i < chars.size(); ++i) {
        curr = std::string(1, chars.at(i));
        if (dict.count(priv + curr)) {
            priv += curr;
        } else {
            res.push_back(dict[priv]);
            if (dict.size() < USHRT_MAX - 1)
                dict[priv + curr] = dict.size() + 1;
            priv = curr;
        }
    }
    res.push_back(dict[priv]);
}

void LZWCoder::WriteToFile(std::ofstream &out) {

    //codesCount
```

```
    codesCount = res.size();
    out.write((char*)&codesCount, sizeof(unsigned int));

    //codes
    for (auto a = res.begin(); a!=res.end(); ++a)
    {
        out.write((char*)&(*a), sizeof(unsigned short));
    }
}


#include "LZWDecoder.h"

void LZWDecoder::InitDict() {
    dict = std::map<std::string, unsigned short>();
    for (char i=CHAR_MIN; i!=CHAR_MAX; ++i)
    {
        dict[std::string(1, i)]=dict.size();
    }
}

std::string LZWDecoder::GetStrByCode(unsigned short code) {
    for (auto &i : dict)
    {
        if (i.second == code)
            return i.first;
    }
    return "";
}

LZWDecoder::LZWDecoder(unsigned short vc) {
    codesVect = std::vector<unsigned short>();
    res = std::shared_ptr<std::vector<std::string>>(new
std::vector<std::string>());
    valuesCount = vc;
    InitDict();

}

void LZWDecoder::Read(std::ifstream &infile) {


    //codesCount
    infile.read((char*)&codesCount, sizeof(unsigned int));

    //codes vector
    unsigned short val;
    for (unsigned int i=0; i<codesCount; ++i)
    {
        infile.read((char*)&val, sizeof(unsigned short));
        codesVect.emplace_back(val);
    }

}


void LZWDecoder::Decode() {
    unsigned short int currCode;
    unsigned short int privCode;
    std::string currStr;
    std::string privStr;
    std::string S;
```

```
    std::vector<char> microbuff = std::vector<char>();

    privCode=codesVect.front();
    privStr=GetStrByCode(privCode);
    for (char k : privStr) {
        if (k==',')
        {
            std::string s="";
            for (char c : microbuff)
            {
                s+=c;
            }
            res->push_back(s);
            microbuff.clear();
        } else{
            microbuff.push_back(k);
        }
    }
    char C = privStr[0];

    for (auto currCode=codesVect.begin()+1;
currCode!=codesVect.end();++currCode)
    {
        privStr=GetStrByCode(privCode);
        currStr=GetStrByCode(*currCode);

        if (dict.count(currStr))
        {
            S=currStr;
        } else
        {
            S=privStr;
            S=S+std::string(1,C);
        }

        for (char k : S)
            if (k==',')
            {
                std::string s="";
                for (char c : microbuff)
                {
                    s+=c;
                }
                res->push_back(s);
                microbuff.clear();
            } else{
                microbuff.push_back(k);
            }
        C=S[0];
        if (dict.size()<USHRT_MAX-1)
            dict[privStr+std::string(1,C)]=dict.size()+1;
        privCode=*currCode;
    }
    std::string s="";
    for (char c : microbuff)
    {
        s+=c;
    }
    res->push_back(s);

    while (res->size()>valuesCount)
    {
        for (int i=0; i<res->size()-1; ++i)
```

```
            {
                std::string s = res->at(i);
                if (s[0]=='"' && s[s.size()-1]!='"')
                {
                    s.append(res->at(i+1));
                    res->at(i)=s;
                    res->erase(res->begin()+i+1);
                }
            }
        }

}
```

Додаток Б

```
#include "HuffmanS.h"

HuffLeafStr::HuffLeafStr(std::string value_t, HuffLeafStr* parent_t, int
freq_t)
{
    value=value_t;
    parent=parent_t;
    freq=freq_t;
    isLeaf=true;
    right= nullptr;
    left= nullptr;
}

HuffLeafStr::HuffLeafStr(HuffLeafStr* right_t, HuffLeafStr* left_t) {
    right = right_t;
    left = left_t;
    freq = right_t->freq + left_t->freq;
    value = "";
    isLeaf = false;
    parent = nullptr;
}

HuffLeafStr::~HuffLeafStr()
{
    delete left;
    delete right;
}

std::string HuffLeafStr::GetCode(std::string inp)
{
    if (this->isLeaf)
    {
        return "1";
    }
    if (left->isLeaf)
    {
        if (left->value.compare(inp)==0)
            return "1";
    }
    else
    {
        std::string v=left->GetCode(inp);
        if (v[0]!='n')
            return "1"+v;
    }

    if (right->isLeaf)
    {
        if (right->value.compare(inp)==0)
            return "0";
    }
    else
    {
        std::string v=right->GetCode(inp);
        if (v[0]!='n')
            return "0"+v;
    }
    return "n";
}
```

```
HuffmanSCoder::HuffmanSCoder(const std::shared_ptr<ColumnAnalyzer> &colAnal)
{
    this->colAnal = colAnal;
    res = std::vector<char>();
}

void HuffmanSCoder::Encode() {
    std::shared_ptr<HuffLeafStr>
TreeRoot=std::shared_ptr<HuffLeafStr>(CreateTree());
    std::map<std::string, std::string> encMap;

    std::vector<bool> codedBools;
    std::string codeS;
    for (int i=0; i<colAnal->columnValues->size(); ++i)
    {
        if (!encMap.count(colAnal->columnValues->at(i)))
            encMap[colAnal->columnValues->at(i)]=TreeRoot->GetCode(colAnal-
>columnValues->at(i));
        codeS=encMap[colAnal->columnValues->at(i)];
        for (int j=0; j<codeS.size(); ++j)
            codedBools.push_back(codeS[j]=='1');
        while (codedBools.size()>=8)
        {
            char s(0);
            bool bits[8];
            for (int i=0; i<8; ++i) {
                bits[i] = codedBools.front();
                codedBools.erase(codedBools.begin());
            }
            s = pack_byte(bits);
            res.push_back(s);
        }
    }
    if (codedBools.size()>0) {
        while (codedBools.size() < 8)
            codedBools.push_back(false);
        char s(0);
        bool bits[8];
        for (int i = 0; i < 8; ++i) {
            bits[i] = codedBools.front();
            codedBools.erase(codedBools.begin());
        }
        s = pack_byte(bits);
        res.push_back(s);
    }

}

HuffLeafStr *HuffmanSCoder::CreateTree() {
    {
        std::vector<HuffLeafStr*> huffVect;
        for (auto f=colAnal->strMap.begin(); f != colAnal->strMap.end(); ++f)
        {
            huffVect.push_back(new HuffLeafStr(f->first, nullptr, f-
>second));
        }

        while (huffVect.size()>1)
        {
            std::sort(huffVect.begin(), huffVect.end(), [](const HuffLeafStr*
lhs, const HuffLeafStr* rhs) {
                return lhs->freq>rhs->freq;});
            HuffLeafStr* rightLeaf=huffVect.back();
```

```
            huffVect.pop_back();
            HuffLeafStr* leftLeaf=huffVect.back();
            huffVect.pop_back();
            HuffLeafStr* fuzed= new HuffLeafStr(rightLeaf, leftLeaf);
            rightLeaf->parent=fuzed;
            leftLeaf->parent=fuzed;
            huffVect.push_back(fuzed);
        }

        return huffVect.at(0);
    }
}

char HuffmanSCoder::pack_byte(bool bits[8]) {
    char res(0);
    for (unsigned i(8); i--;)
    {
        res <<=1;
        res |= char(bits[i]);
    }
    return res;
}

void HuffmanSCoder::WriteToFile(std::ofstream &out) {

    //map size
    unsigned short mapSize = colAnal->strMap.size();
    out.write((char*)&mapSize, sizeof(unsigned short));

    //map
    for (auto item = colAnal->strMap.begin(); item!=colAnal-
>strMap.end();++item)
    {
        std::string s = item->first;
        do{
            if (s.size()<=254)
            {
                unsigned char sSize = s.size();
                out.write((char*)&sSize, sizeof(unsigned char));
                out.write((char*)(s.c_str()), sizeof(char)*sSize);
                s.clear();
            } else
            {
                unsigned char sSize = 255;
                out.write((char*)&sSize, sizeof(unsigned char));
                out.write((char*)(s.substr(0,sSize).c_str()),
sizeof(char)*sSize);
                s.erase(0,sSize);
            }
        }
        while (s.size()!=0);

        unsigned short freq = item->second;
        out.write((char*)&freq, sizeof(unsigned short));
    }

    //res size
    unsigned int ressize = res.size();
    out.write((char*)&ressize, sizeof(unsigned int));

    //codes
    unsigned int from=0;
    char* buff=new char[USHRT_MAX];
```

```
    while (from<res.size())
    {
        if (from+USHRT_MAX<res.size())
        {
            for (int i=0; i<USHRT_MAX; ++i)
            {
                buff[i]=res.at(from+i);
            }
            out.write(buff, sizeof(char)*USHRT_MAX);
            from+=USHRT_MAX;
        }
        else {
            for (int i=0; i<res.size()-from; ++i)
            {
                buff[i]=res.at(from+i);
            }
            out.write(buff, sizeof(char)*(res.size()-from));
            from=res.size();
        }
    }
}

HuffmanSDecoder::HuffmanSDecoder(const unsigned short & codesCount) {
    this->valuesCount = codesCount;
    res = std::shared_ptr<std::vector<std::string>>(new
std::vector<std::string>());
    freqMap = std::map<std::string, unsigned short>();
    codesVect = std::vector<char>();

}

void HuffmanSDecoder::Read(std::ifstream &infile) {
    //read freq map size
    unsigned short mapSize;
    infile.read((char*)&mapSize, sizeof(unsigned short));
    //read freq map
    for (unsigned short i = 0; i<mapSize; ++i)
    {
        std::string s;
        unsigned short freq;
        unsigned char c;
        do {
            infile.read((char *) &c, sizeof(unsigned char));
            char *buff = new char[c];
            infile.read(buff, sizeof(char) * c);
            for (unsigned char i=0; i<c; ++i)
                s.push_back(buff[i]);
            delete[] buff;
        }
        while (c==255);
        infile.read((char*)&freq, sizeof(unsigned short));
        freqMap[s] = freq;
    }
    //reading codes vect size
    unsigned int inpSize;
    infile.read((char*)&inpSize, sizeof(unsigned int));
    //reading codes
    char c;
    for (unsigned int i = 0; i<inpSize; ++i)
    {
        infile.read((char*)&c, sizeof(char));
        codesVect.push_back(c);
    }
```

```
}

void HuffmanSDecoder::Decode() {
    std::shared_ptr<HuffLeafStr>
TreeRoot=std::shared_ptr<HuffLeafStr>(CreateTree());
    HuffLeafStr* currLeaf=TreeRoot.get();

    if (freqMap.begin()->first.compare("1347")==0)
    {
        std::cout<<"cock";
    }

    if (freqMap.size()==1)
    {
        for (int i=0; i<valuesCount; ++i)
        {
            res->push_back(TreeRoot->value);
        }
    } else {
        std::deque<bool> boolsToDecode;
        for (auto c = codesVect.begin(); c != codesVect.end(); ++c) {
            for (int i = 0; i < 8; ++i)
                boolsToDecode.push_back((*c) & (1 << i));

            while (boolsToDecode.size() > 0) {
                if (currLeaf->isLeaf) {
                    res->push_back(currLeaf->value);
                    currLeaf = TreeRoot.get();
                } else {
                    if (boolsToDecode.front())
                        currLeaf = currLeaf->left;
                    else
                        currLeaf = currLeaf->right;
                    boolsToDecode.pop_front();
                }
            }
        }

        if (currLeaf->isLeaf) {
            res->push_back(currLeaf->value);
        }

        while (res->size() > valuesCount)
            res->pop_back();
    }
}

HuffLeafStr *HuffmanSDecoder::CreateTree() {
    std::vector<HuffLeafStr*> huffVect;
    for (auto f=freqMap.begin(); f != freqMap.end(); ++f)
    {
        huffVect.push_back(new HuffLeafStr(f->first, nullptr, f->second));
    }

    while (huffVect.size()>1)
    {
        std::sort(huffVect.begin(), huffVect.end(), [](const HuffLeafStr*
lhs, const HuffLeafStr* rhs) {
            return lhs->freq>rhs->freq;});
        HuffLeafStr* rightLeaf=huffVect.back();
        huffVect.pop_back();
        HuffLeafStr* leftLeaf=huffVect.back();
```

```
            huffVect.pop_back();
            HuffLeafStr* fuzed= new HuffLeafStr(rightLeaf, leftLeaf);
            rightLeaf->parent=fuzed;
            leftLeaf->parent=fuzed;
            huffVect.push_back(fuzed);
        }

        return huffVect.at(0);
    }
```

Додаток В

```
#include "HuffmanK.h"

HuffmanKCoder::HuffmanKCoder(const std::shared_ptr<ColumnAnalyzer> &colAnal)
{
    this->colAnal = colAnal;
    res = std::vector<char>();
    codesCount=0;
}

void HuffmanKCoder::Encode() {
    std::shared_ptr<HuffLeafCh>
TreeRoot=std::shared_ptr<HuffLeafCh>(CreateTree());
    std::map<char, std::string> encMap;


    std::vector<char> chars = std::vector<char>();

    for (auto str = colAnal->columnValues->begin(); str!=colAnal-
>columnValues->end(); ++str)
    {
        for (auto c=str->begin(); c!=str->end(); ++c)
        {
            chars.emplace_back(*c);
        }
        chars.emplace_back(',');
    }

    chars.pop_back();

    codesCount = chars.size();

    std::vector<bool> codedBools;
    std::string codeS;
    for (auto c = chars.begin(); c!=chars.end(); ++c)
    {
        if (!encMap.count(*c))
            encMap[*c]=TreeRoot->GetCode(*c);
        codeS=encMap[*c];
        for (int j=0; j<codeS.size(); ++j)
            codedBools.push_back(codeS[j]=='1');
        if (codedBools.size()>=8)
        {
            char s(0);
            bool bits[8];
            for (int i=0; i<8; ++i) {
                bits[i] = codedBools.front();
                codedBools.erase(codedBools.begin());
            }
            s = pack_byte(bits);
            res.push_back(s);
        }
    }
    if (codedBools.size()>0) {
        while (codedBools.size() < 8)
            codedBools.push_back(false);
        char s(0);
        bool bits[8];
        for (int i = 0; i < 8; ++i) {
            bits[i] = codedBools.front();
            codedBools.erase(codedBools.begin());
        }
```

```
            s = pack_byte(bits);
            res.push_back(s);
        }

    }

    HuffLeafCh *HuffmanKCoder::CreateTree() {
        {

            std::vector<HuffLeafCh*> huffVect;
            for (auto f=colAnal->charMap.begin(); f != colAnal->charMap.end();
    ++f)
            {
                huffVect.push_back(new HuffLeafCh(f->first, nullptr, f->second));
            }

            while (huffVect.size()>1)
            {
                std::sort(huffVect.begin(), huffVect.end(), [](const HuffLeafCh*
    lhs, const HuffLeafCh* rhs) {
                    return lhs->freq>rhs->freq;});
                HuffLeafCh* rightLeaf=huffVect.back();
                huffVect.pop_back();
                HuffLeafCh* leftLeaf=huffVect.back();
                huffVect.pop_back();
                HuffLeafCh* fuzed= new HuffLeafCh(rightLeaf, leftLeaf);
                rightLeaf->parent=fuzed;
                leftLeaf->parent=fuzed;
                huffVect.push_back(fuzed);
            }

            return huffVect.at(0);
        }
    }

    char HuffmanKCoder::pack_byte(bool bits[8]) {
        char res(0);
        for (unsigned i(8); i--;)
        {
            res <<=1;
            res |= char(bits[i]);
        }
        return res;
    }

    void HuffmanKCoder::WriteToFile(std::ofstream &out) {
        //dic size
        unsigned char dicSize = (unsigned char)colAnal->charMap.size();
        out.write((char*)&dicSize, sizeof(unsigned char));

        //dic
        for (auto i=colAnal->charMap.begin(); i!=colAnal->charMap.end(); ++i)
        {
            out.write((char*)&(i->first), sizeof(char));
            out.write((char*)&(i->second), sizeof(unsigned short));
        }

        //res size
        unsigned int ressize = res.size();
        out.write((char*)&ressize, sizeof(unsigned int));

        //codes
        unsigned int from=0;
```

```
    char* buff=new char[USHRT_MAX];
    while (from<res.size())
    {
        if (from+USHRT_MAX<res.size())
        {
            for (int i=0; i<USHRT_MAX; ++i)
            {
                buff[i]=res.at(from+i);
            }
            out.write(buff, sizeof(char)*USHRT_MAX);
            from+=USHRT_MAX;
        }
        else {
            for (int i=0; i<res.size()-from; ++i)
            {
                buff[i]=res.at(from+i);
            }
            out.write(buff, sizeof(char)*(res.size()-from));
            from=res.size();
        }
    }
}

HuffmanKDecoder::HuffmanKDecoder(unsigned short vc) {
    res = std::shared_ptr<std::vector<std::string>>(new
std::vector<std::string>());
    freqMap = std::map<char, unsigned short>();
    codesVect = std::vector<char>();
    valuesCount = vc;
    codesCount=0;

}

void HuffmanKDecoder::Read(std::ifstream &infile) {
    //dic size
    unsigned char dicSize;
    infile.read((char*)&dicSize, sizeof(unsigned char));
    //dic
    for (unsigned char i=0; i<dicSize; ++i)
    {
        char ch;
        unsigned short freq;
        infile.read((char*)&ch, sizeof(char));
        infile.read((char*)&freq, sizeof(unsigned short));
        freqMap[ch]=freq;
    }

    //reading codes vect size
    unsigned int inpSize;
    infile.read((char*)&inpSize, sizeof(unsigned int));
    //reading codes
    char c;
    for (unsigned int i = 0; i<inpSize; ++i)
    {
        infile.read((char*)&c, sizeof(char));
        codesVect.push_back(c);
    }

}

void HuffmanKDecoder::Decode() {
    std::shared_ptr<HuffLeafCh>
TreeRoot=std::shared_ptr<HuffLeafCh>(CreateTree());
```

```
    HuffLeafCh* currLeaf=TreeRoot.get();

    std::vector<char> microbuff = std::vector<char>();

    std::deque<bool> boolsToDecode;

    for(auto c=codesVect.begin(); c!=codesVect.end(); ++c)
    {
        for (int i=0; i<8; ++i)
            boolsToDecode.push_back((*c) & (1 << i));

        while (boolsToDecode.size()>0)
        {
            if (currLeaf->isLeaf) {
                if (currLeaf->value == ',')
                {
                    std::string s ="";
                    for (auto ch=microbuff.begin(); ch!=microbuff.end();
++ch)
                    {
                        s.push_back(*ch);
                    }
                    res->push_back(s);
                    microbuff.clear();
                }
                else
                    microbuff.push_back(currLeaf->value);
                currLeaf=TreeRoot.get();
            } else {
                if (boolsToDecode.front())
                    currLeaf = currLeaf->left;
                else
                    currLeaf = currLeaf->right;
                boolsToDecode.pop_front();
            }
        }
    }

    if (currLeaf->isLeaf) {
        if (currLeaf->value == ',') {
            std::string s = "";
            for (auto ch = microbuff.begin(); ch != microbuff.end(); ++ch) {
                s.push_back(*ch);
            }
            res->push_back(s);
            microbuff.clear();
            s ="";
            res->push_back(s);
        } else
            microbuff.push_back(currLeaf->value);
        currLeaf = TreeRoot.get();
    }

    if (microbuff.size()>0)
    {
        std::string s ="";
        for (auto ch=microbuff.begin(); ch!=microbuff.end(); ++ch)
        {
            s.push_back(*ch);
        }
        res->push_back(s);
        microbuff.clear();
    }
```

```
    bool ch = true;
    while (res->size()>valuesCount)
    {
        if (!ch)
        {
            while (res->size()>valuesCount)
            {
                res->pop_back();
            }
            break;
        }
        ch = false;
        for (int i=0; i<res->size()-1; ++i)
        {
            std::string s = res->at(i);
            if (s[0]=='"' && s[s.size()-1]!='"')
            {
                s.append(res->at(i+1));
                res->at(i)=s;
                res->erase(res->begin()+i+1);
                ch = true;
            }
        }
    }
}

HuffLeafCh *HuffmanKDecoder::CreateTree() {
    std::vector<HuffLeafCh*> huffVect;
    for (auto f=freqMap.begin(); f != freqMap.end(); ++f)
    {
        huffVect.push_back(new HuffLeafCh(f->first, nullptr, f->second));
    }

    while (huffVect.size()>1)
    {
        std::sort(huffVect.begin(), huffVect.end(), [](const HuffLeafCh* lhs,
const HuffLeafCh* rhs) {
            return lhs->freq>rhs->freq;});
        HuffLeafCh* rightLeaf=huffVect.back();
        huffVect.pop_back();
        HuffLeafCh* leftLeaf=huffVect.back();
        huffVect.pop_back();
        HuffLeafCh* fuzed= new HuffLeafCh(rightLeaf, leftLeaf);
        rightLeaf->parent=fuzed;
        leftLeaf->parent=fuzed;
        huffVect.push_back(fuzed);
    }

    return huffVect.at(0);
}
```

Додаток Г

```
#include "ColumnAnalyzer.h"

ColumnAnalyzer::ColumnAnalyzer(std::shared_ptr<std::vector<std::string>>
colPtr, const bool& usetypes, const bool& uselzw) {
    strMap = std::map<std::string, unsigned short int>();
    charMap = std::map<char, unsigned short>();
    columnValues = colPtr;
    ushortCol = std::vector<unsigned short>();
    intCol = std::vector<int>();
    floatCol = std::vector<float>();
    doubleCol = std::vector<double>();
    useTypes=usetypes;
    useLZW=uselzw;
}

char ColumnAnalyzer::Analyze() {
    bool moveToNext = false;
    //int overdrawVal = columnValues->size()>1000 ? columnValues->size()/50 :
columnValues->size()/10;
    int overdrawVal = columnValues->size()/10;
    for (auto val = columnValues->begin(); val!=columnValues->end(); ++val)
    {
        auto iter=strMap.find(*val);
        if (iter==strMap.end())
        {
            strMap[*val]=1;
        } else
        {
            if (strMap[*val]<UINT32_MAX) strMap[*val]++;
        }

        if (strMap.size()>overdrawVal) {
            moveToNext = true;
            break;
        }
    }
    if (!moveToNext)
        return 'h';

    moveToNext = false;

    if (useTypes) {

        for (auto val = columnValues->begin(); val != columnValues->end();
++val) {
            unsigned short num = 0;
            if (is_Short(*val, num)) {
                ushortCol.push_back(num);
            } else {
                moveToNext = true;
                break;
            }
        }
        if (!moveToNext)
            return 'u';

        ushortCol = std::vector<unsigned short>();
        moveToNext = false;

        for (auto val = columnValues->begin(); val != columnValues->end();
++val) {
```

```
            int num = 0;
            if (is_Int(*val, num)) {
                intCol.push_back(num);
            } else {
                moveToNext = true;
                break;
            }
        }
        if (!moveToNext) {
            return 'i';
        }

        intCol = std::vector<int>();
//        moveToNext = false;
//
//        for (auto val = columnValues->begin(); val != columnValues->end();
++val) {
//            float num = 0;
//            if (is_Float(*val, num)) {
//                floatCol.push_back(num);
//            } else {
//                moveToNext = true;
//                break;
//            }
//        }
//        if (!moveToNext)
//            return 'f';
//
//        floatCol = std::vector<float>();
//        moveToNext = false;
//
//        for (auto val = columnValues->begin(); val != columnValues->end();
++val) {
//            double num = 0;
//            if (is_Double(*val, num)) {
//                doubleCol.push_back(num);
//            } else {
//                moveToNext = true;
//                break;
//            }
//        }
//        if (!moveToNext)
//            return 'd';
//
//        doubleCol = std::vector<double>();
    }

    moveToNext = false;

    for (auto val = columnValues->begin(); val != columnValues->end(); ++val)
{
        if (!is_Numeric(*val))
        {
            moveToNext = true;
            break;
        }
    }
    if (!moveToNext) {
        return 'n';
    }

    if (useLZW)
        return 'l';
```

```
    for (auto i=columnValues->begin(); i!=columnValues->end(); ++i)
    {
        for (auto c=i->begin(); c!=i->end(); ++c)
        {
            if ((++charMap[*c])==USHRT_MAX-1)
                for (auto k=charMap.begin(); k!=charMap.end();++k)
                {
                    k->second=k->second/10+k->second%10;
                }
        }
        if ((++charMap[','])==USHRT_MAX-1)
            for (auto k=charMap.begin(); k!=charMap.end();++k)
            {
                k->second=k->second/10+k->second%10;
            }
    }
    return 'k';
}

bool ColumnAnalyzer::is_Float(const std::string& s, float& res)
{
    if (s.empty())
    {
        res = FLT_MIN;
        return true;
    }

    int before_del = 0;
    int after_del = 0;
    bool was_del = false;
    int signes =0;

    for (auto i=s.begin(); i!=s.end();++i)
    {
        if (std::isdigit(*i))
        {
            if (was_del)
                after_del++;
            else
                before_del++;
        } else
        if (*i=='-')
        {
            signes++;
        }
        else if (*i=='.')
        {
            was_del=true;
        } else
        {
            return false;
        }
    }

    if (signes>2)
        return false;
    if (before_del+after_del+(was_del ? 1 : 0)+signes!=s.size())
        return false;
    if (before_del>=37)
        return false;
    if (after_del>=9)
        return false;
```

```
    try
    {
        res = std::stof(s);
    }
    catch(...)
    {
        return false;
    }
    return true;
}

bool ColumnAnalyzer::is_Double(const std::string& s, double& res)
{
    if (s.empty())
    {
        res = DBL_MIN;
        return true;
    }

    int before_del = 0;
    int after_del = 0;
    bool was_del = false;
    int signes =0;

    for (auto i=s.begin(); i!=s.end();++i)
    {
        if (std::isdigit(*i))
        {
            if (was_del)
                after_del++;
            else
                before_del++;
        } else
        if (*i=='-')
        {
            signes++;
        }
        else if (*i=='.')
        {
            was_del=true;
        } else
        {
            return false;
        }
    }

    if (signes>2)
        return false;
    if (before_del+after_del+(was_del ? 1 : 0)+signes!=s.size())
        return false;
    if (before_del>=307)
        return false;
    if (after_del>=16)
        return false;
    try
    {
        res = std::stod(s);
    }
    catch(...)
    {
        return false;
    }
```

```
    return true;
}

bool ColumnAnalyzer::is_Int(const std::string& s, int& res)
{
    if (s.empty())
    {
        res = INT32_MIN;
        return true;
    }

    int digcount=0;
    int signes=0;

    for (auto i=s.begin(); i!=s.end();++i)
    {
        if (std::isdigit(*i))
        {
            digcount++;
        } else
        if (*i=='-')
        {
            signes++;
        } else
        {
            return false;
        }
    }

    if (digcount>9)
        return false;
    if (signes>1)
        return false;
    if (signes+digcount!=s.size())
        return false;

    try
    {
        res = std::stoi(s);
    }
    catch(...)
    {
        return false;
    }
    return true;
}

bool ColumnAnalyzer::is_Short(const std::string &s, unsigned short &res) {
    {
        if (s.empty())
        {
            res = USHRT_MAX;
            return true;
        }

        int digcount=0;

        for (auto i=s.begin(); i!=s.end();++i)
        {
            if (std::isdigit(*i))
            {
                digcount++;
            } else
```

```
            {
                return false;
            }
        }

        if (digcount>5)
            return false;

        int i;
        try
        {
            i = std::stoi(s);
        }
        catch(...)
        {
            return false;
        }

        if (i>USHRT_MAX-1)
            return false;

        res = (unsigned short) i;

        return true;
    }
}

bool ColumnAnalyzer::is_Numeric(const std::string &s) {
    for (auto i=s.begin(); i!=s.end();++i)
    {
        if (!(std::isdigit(*i) || *i==',' || *i=='.' || *i==':' || *i==' ' ||
*i=='/' || *i=='-'))
            return false;
    }

    return true;
}
```

Додаток Д

```
 void CSVBlockCompressor::AnalyzeColumns() {
    comprMethodsCodes = std::vector<char>();

    if (doParallel) {
        std::vector<std::thread> threadVector = std::vector<std::thread>();
        for (auto i = columnsVector.begin(); i != columnsVector.end(); ++i) {
            analizedColumns.push_back(std::shared_ptr<ColumnAnalyzer>(new
ColumnAnalyzer(*i, useTypes, useLZW)));
            comprMethodsCodes.push_back(' ');
        }

        for (int i = 0; i < columnsVector.size(); ++i) {
            threadVector.push_back(
                    std::thread(ThrAnalize, std::ref(analizedColumns.at(i)),
std::ref(comprMethodsCodes.at(i))));
        }

        for (auto i = threadVector.begin(); i != threadVector.end(); ++i) {
            i->join();
        }
    } else{
        for (auto i = columnsVector.begin(); i != columnsVector.end(); ++i) {
            analizedColumns.push_back(std::shared_ptr<ColumnAnalyzer>(new
ColumnAnalyzer(*i, useTypes, useLZW)));
            comprMethodsCodes.push_back(' ');
        }

        for (int i = 0; i < columnsVector.size(); ++i) {
            ThrAnalize(std::ref(analizedColumns.at(i)),
std::ref(comprMethodsCodes.at(i)));
        }

    }
}
```