Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів та алгоритмів компресії даних
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-1

Жихарського П.О.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Каук В.І.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

**ЗАВДАННЯ**
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Жихарському Петру Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Аналіз методів та алгоритмів компресії даних»
Затверджена наказом по університету від 29.03. 2024р. № 250 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 17.06.2024
3. Вихідні дані до роботи опис досліджуваних методів та алгоритмів, мови програмування C#, технології .NET 8.0, середовища розробки Visual Studio 2022
4. Перелік питань, що потрібно опрацювати в роботі
аналіз та порівняння існуючих методів та алгоритмів компресії даних, вибір підходящих інструментів для дослідження, написання програмних рішень, проведення експериментів та аналіз отриманих результатів

# КАЛЕНДАРНИЙ ПЛАН

| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз предметної галузі та постановка задачі | 23.01 – 05.02.24 | *виконано* |
| 2 | Аналіз та вибір алгоритмів для дослідження | 05.02 – 20.02.24 | *виконано* |
| 3 | Аналіз та моделювання предметної області | 20.02 – 01.03.24 | *виконано* |
| 4 | Планування експериментів | 01.03 – 10.03.24 | *виконано* |
| 5 | Програмна реалізація кожного з обраних для дослідження алгоритмів | 10.03 – 20.03.24 | *виконано* |
| 6 | Експериментальні дослідження | 20.03 – 20.04.24 | *виконано* |
| 7 | Аналіз результатів експериментальних досліджень та розробка рекомендацій | 20.04 – 01.05.24 | *виконано* |
| 8 | Написання та оформлення статті та тез доповіді | 01.05 – 10.05.24 | *виконано* |
| 9 | Підготовка пояснювальної записки | 10.05 – 31.05.24 | *виконано* |
| 10 | Підготовка презентації та доповіді | 31.05 – 05.06.24 | *виконано* |
| 11 | Нормоконтроль | 05.06 – 08.06.24 | *виконано* |
| 12 | Рецензування | 08.06 – 12.06.24 | *виконано* |
| 13 | Занесення диплома в електронний архів | 12.05.2024 | *виконано* |
| 14 | Попередній захист | 13.05.2024 | *виконано* |
| 15 | Допуск до захисту у зав. кафедри | 15.06.2024 | *виконано* |

Дата видачі завдання 23 січня 2024р.

Студент _____ Жихарський П.О.
(підпис)

Керівник кваліфікаційної роботи _____ доц. Каук В.І.
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 73 с., 13 рис., 8 табл., 13 джерел.

АЛГОРИТМИ, ДОСЛІДЖЕННЯ, МЕТОДИ, ПІДХОДИ, КОМПРЕСІЯ .NET, C#

Об’єктом дослідження є методи та алгоритми компресії даних.

Метою роботи є проведення дослідження продуктивності методів та алгоритмів компресії даних.

У результаті кваліфікаційної роботи було розроблено консольний застосунок для проведення дослідження методів та алгоритмів компресії даних.

COMPRESION ALGORITHMS, .NET, C#

The object of research is data compression methods and algorithms.

The purpose of the study is to investigate the performance of data compression methods and algorithms.

As a result of the qualification work, a console application was developed to study data compression methods and algorithms.

Заява щодо самостійного виконання роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, Жихарський Петро Олександрович, студент гр. ІПЗм-22-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

# ВСТУП

З поширенням у сучасному світі цифрових носіїв та мережі Інтернет, дедалі частіше постає питання про зберігання та передачу даних у цифровому форматі. Потреби користувачів зростають швидше, ніж розвиваються апаратні технології зберігання і передавання даних, тому все частіше постає питання програмної оптимізації цих процесів. Незважаючи на безперервно зростаючий накопичувальний обсяг інформаційних носіїв, деколи потрібно зберегти велику кількість даних на сховищі невеликої ємності. Компресія даних має на увазі під собою зменшення розміру інформації, що розглядається, завдяки використанню того факту, що здебільшого дані не є випадковим набором біт, а підкоряються певному закону. Інакше кажучи, використовується той факт, що дані є або залежними випадковими величинами, або випадковими величинами, що підкоряються певній, нерівномірній функції розподілу[1].

Характерною особливістю більшості типів даних є їхня надмірність. Ступінь надмірності даних залежить від типу даних. Наприклад, для відеоданих ступінь надмірності в кілька разів більший ніж для графічних даних, а ступінь надмірності графічних даних, своєю чергою, більший ніж ступінь надмірності текстових даних. Іншим фактором, що впливає на ступінь надмірності, є прийнята система кодування. Прикладом систем кодування можуть бути звичайні мови спілкування, які є ні чим іншим, як системами кодування понять та ідей для висловлення думок. Так, встановлено, що кодування текстових даних за допомогою засобів української мови дає в середньому надмірність на 20-25% більшу, ніж кодування аналогічних даних засобами англійської мови[2].

Для людини надмірність даних часто пов'язана з якістю інформації, оскільки надмірність, як правило, покращує зрозумілість і сприйняття інформації. Однак, коли йдеться про зберігання та передачу інформації засобами комп'ютерної техніки, то надмірність відіграє негативну роль, оскільки вона призводить до зростання вартості зберігання та передачі інформації. Особливо актуальною ця проблема стає в разі обробки величезних обсягів інформації за

незначних обсягів носіїв даних. У зв'язку з цим, постійно виникає проблема зменшення надмірності або компресії даних. Якщо методи компресії даних застосовуються до готових файлів, то часто замість терміна "стиснення даних" вживають термін "архівація даних", стислий варіант даних називають архівом, а програмні засоби, що реалізують методи стиснення, називаються архіваторами.

Зберігання та передача інформації обходяться учасникам інформаційного процесу недешево. Знаючи вартість носія і його місткість, неважко підрахувати, у що обходиться зберігання одиниці інформації, а знаючи пропускну здатність каналу зв'язку і вартість його оренди, можна визначити витрати на передачу одиниці інформації. Отримані результати зазвичай становлять цілком значущі величини як для корпоративних, так і для індивідуальних користувачів. У зв'язку з цим регулярно виникає необхідність стискати дані перед тим, як розміщувати їх в архівах або передавати каналами зв'язку. Відповідно, існує і зворотна необхідність відновлення даних з попередньо ущільнених архівів. Під час експлуатації комп'ютера з найрізноманітніших причин можливі псування і втрата інформації на магнітних дисках. Це може статися через фізичне псування магнітного диска, неправильне коригування або випадкове знищення файлів, руйнування інформації комп'ютерним вірусом тощо. Для того щоб зменшити втрати в таких ситуаціях, слід мати архівні копії використовуваних файлів і систематично оновлювати копії змінюваних файлів[3].

Актуальність роботи з дослідження методів та алгоритмів компресії даних зумовлена кількома ключовими факторами. У сучасному світі обсяги генерованих і збережених даних зростають експоненціально. Big Data, IoT, мультимедійні дані та інші джерела вимагають ефективних методів зберігання і передачі інформації. Без ефективної компресії витрати на зберігання і передачу даних можуть стати непідйомними. Пристрої з обмеженими обчислювальними ресурсами  потребують методів, що дають змогу мінімізувати використання пам'яті та пропускної здатності мережі. Ефективні алгоритми компресії допомагають оптимізувати використання цих ресурсів.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

## 1.1 Аналітичний огляд

Компресія даних - це процес за допомогою якого, можна значно зменшити об’єм даних. Це можна зробити шляхом видалення надлишкової інформації або використання більш ефективних методів представлення даних. Компресія даних допомагає зберігати набагато більшу кількість даних на тих самих об’ємах пам’яті, ніж зберігання даних у їх первинному виді. Також зменшення об’єму даних допомагає пришвидшити передачу через мережу інтернет. Це важливо для ефективної передачі даних через Інтернет або інші мережі, особливо при обмеженій пропускній здатності.

Основа ідея компресії даних базується на пошук та видаленні надмірності, яка присутня у вихідних даних. Повторення блоків у тексті, наприклад слів або цілих речень, є простішим прикладом надмірності даних. Надмірність такого роду можна усунути заміною повторюваної послідовності посиланням на вже закодований фрагмент із вказання його довжини. Ще один вид надмірності даних полягає у тому, що деякі значення в стисливих даних зустрічаються частіше за інші. Зменшення обсягу даних досягається шляхом заміни часто використовуваних даних короткими кодовими словами, а рідкісних даних – довгими кодовими словами (ентропійне кодування)[4].

Усі методи компресії даних поділити на два типи:

- компресія без втрат;
- компресія з втратами.

Алгоритми стиснення даних без втрат дозволяють повністю відновити вихідні дані, стиснення з втратами дозволяє відновлювати дані з викривленнями, які зазвичай не важливі з точки зору подальшого використання відновлених даних.[5]

Алгоритми компресії даних без втрат зазвичай використовують де важливо зберегти кожен біт інформації, таких як зберігання текстів, програмного коду, баз даних текстових даних, комп'ютерних програм, рідше - для скорочення

обсягу аудіо- та відеоданих, цифрових фотографій або інших критичних даних, у випадках, коли спотворення даних є неприпустимими або небажаним процесом. При створенні архівів, де збереження оригінальної структури і вмісту файлів є критичним, використовуються методи компресії без втрат. Такі архіви можуть містити в собі безліч різнорідних файлів і структур даних. Також такий вид алгоритмів застосовується для прискорення передавання даних мережею, особливо за умови обмеженої пропускної здатності. Це знижує час передачі та зменшує навантаження на мережу без шкоди для цілісності даних.

Алгоритми компресії даних з втратами володіють значно більшою, ніж компресія без втрат, ефективністю, зазвичай застосовується для скорочення об'єму аудіо- та відеоданих і цифрових фотографій у тих випадках, коли таке скорочення є пріоритетним, а повна відповідність вихідних і відновлених даних не потрібна. Наприклад, формат JPEG для зображень, MP3 для аудіо та MPEG для відео використовують методи компресії з втратами для скорочення розмірів файлів, що робить їх більш придатними для зберігання і передачі. У мережевих умовах, де пропускна здатність обмежена, компресія з втратами важливе для забезпечення плавного передавання відео. Це дає змогу зменшити об'єм даних, що передаються мережею, що особливо цінно під час потокового передавання відео та організації відеоконференцій.[5]

## 1.2 Характеристики алгоритмів компресії даних

Основною характеристикою алгоритму компресії є коефіцієнт компресії. Він визначається як відношення об'єму вихідних нестиснутих даних до об'єму стиснутих, тобто: $k=So/Sc$, де k - коефіцієнт компресії, So - об'єм вихідних даних, а Sc - об'єм стиснутих. Таким чином, чим вищий коефіцієнт компресії, тим алгоритм ефективніший [6]. Слід зазначити:

- якщо $k = 1$, то алгоритм не здійснює компресії даних, тобто вихідні дані рівні за об'ємом до вхідних даних;

- якщо k < 1, то алгоритм створює дані більшого розміру, ніж нестиснене, тобто здійснює шкідливу роботу.[7]

Випадок, коли k < 1, повністю можливий у контексті компресії даних. Неможливо створити алгоритм компресії без втрат, який для будь-яких даних завжди формував би вихідні дані із меншою або рівною довжиною. Це обґрунтовано тим, що кількість можливих різних повідомлень довжиною n біт дорівнює 2n. Згідно з цим, кількість різних повідомлень із довжиною, меншою або рівною n (включаючи хоча б одне повідомлення меншої довжини), буде меншою за 2n. Таким чином, неможливо чітко відображати всі вихідні повідомлення на їх стиснені відповідники: або деякі вихідні повідомлення не матимуть стисненого представлення, або кілька вихідних повідомлень можуть відповідати одному й тому ж стисненому варіанту, що призводить до невизначеності. Навіть у випадку, коли алгоритм компресії збільшує розмір вихідних даних, можна легко забезпечити те, щоб їх обсяг гарантовано не збільшився більше, ніж на 1 біт .Тобто зробити так, щоб навіть у найгіршому випадку мала місце нерівність:

$$k = So \,/\, (So + 1)$$

Робиться це так: якщо обсяг стиснутих даних менший за обсяг вихідних, повертаються стислі дані шляхом додавання до них "1", інакше повертаємо вихідні дані, додавши до них "0". Коефіцієнт компресії може бути як постійним (деякі алгоритми компресії звуку, зображення тощо, наприклад A-закон, μ-закон, ADPCM, усічене блокове кодування), так і змінним. У другому випадку він може бути визначений або для кожного конкретного повідомлення, або оцінений за деякими критеріями:

- середній (зазвичай за деяким тестовим набором даних);
- максимальний (випадок найкращої компресії);
- мінімальний (випадок найгіршої компресії).

Коефіцієнт компресії з втратами при цьому сильно залежить від допустимої похибки компресії або якості, яка зазвичай виступає як параметр алгоритму. У загальному випадку постійний коефіцієнт компресії здатні забезпечити тільки методи компресії даних із втратами[8].

Основним критерієм, який визначає відмінність між алгоритмами компресії, є наявність або відсутність втрат, яка була описана вище. Узагальнено взято, алгоритми компресії без втрат є універсальними в тому сенсі, що їх можна застосовувати для будь-яких типів даних без обмежень, в той час як застосування компресії з втратами повинно бути обґрунтоване. Деякі типи даних категорично не допускають спотворень, оскільки зміни в них обов'язково ведуть до зміни семантики. Серед таких даних можна вказати символічні дані, зміна яких автоматично впливає на їхню семантику, такі як програми та їхні вихідні тексти, двійкові масиви і т. д. Життєво важливі дані також належать до цієї категорії, оскільки будь-які зміни в них можуть викликати критичні помилки, наприклад, дані, отримані з медичної апаратури чи контрольних пристроїв літальних чи космічних апаратів. Додатково, також існують дані, які піддаються багаторазовій компресії та відновленню під час багатоетапного оброблення графічних, звукових і відеоданих.

Різні алгоритми можуть вимагати різної кількості ресурсів обчислювальної системи, на яких вони реалізовані:

- оперативної пам'яті (під проміжні дані);
- постійної пам'яті (під код програми та константи);
- процесорного часу.

Загалом, вимоги до алгоритмів залежать від їх складності. Зазвичай спостерігається загальна тенденція: чим більш ефективний та універсальний алгоритм, тим вищі вимоги до обчислювальних ресурсів для його виконання. Проте у конкретних випадках прості та компактні алгоритми можуть виявитися не менш ефективними, ніж їх складні та універсальні аналоги [9].

Системні вимоги визначають споживчі якості алгоритмів: чим менше вимогливий алгоритм, тим легше йому реалізуватися на простій, компактній, надійній та більш доступній системі.

Оскільки алгоритми компресії та відновлення діють як в парі, важливо збалансувати системні вимоги для обох. Часто можна ускладнивши один алгоритм, значно спрощуючи інший. Таким чином, існують три можливих варіанти:

1. Алгоритм компресії даних потребує більших ресурсів обчислення, ніж алгоритм відновлення.

2. Алгоритми компресії і відновлення вимагають приблизно рівних ресурсів обчислення.

3. Алгоритм компресії даних потребує меньше ресурсів обчислення, ніж алгоритм відновлення.

Алгоритм компресії вимагає значно більше обчислювальних ресурсів порівняно з алгоритмом відновлення. Це типове співвідношення, особливо в ситуаціях, коли одноразово стиснуті дані мають бути використані повторно. Це часто спостерігається в цифрових аудіо- та відеопрогравачах.

Алгоритми компресії і відновлення можуть вимагати приблизно однакові обчислювальні ресурси. Це особливо актуально в контексті лінійного зв'язку, коли процес компресії і відновлення відбувається одноразово на обох кінцях зв'язку, як, наприклад, у цифровій телефонії.

Алгоритм стиснення вимагає значно менше ресурсів, ніж алгоритм відновлення. Така ситуація є типовою для випадків, коли процедура стиснення реалізована за допомогою простого, часто портативного пристрою, для якого обсяг доступних ресурсів є критично важливим. Прикладами можуть бути космічний апарат або велика розподілена мережа датчиків. Також в цю категорію можуть входити дані, розпакування яких потрібно лише у дуже обмеженому відсотку випадків, наприклад, записи відеоспостереження[10].

## 1.3 Постановка задачі

Метою цього дослідження є порівняння певних алгоритмів компресії даних. В рамках дослідження потрібно вирішити наступні завдання:

- проаналізувати та обрати алгоритми для компресії даних;
- проаналізувати існуючі підходи до вимірювання швидкодії алгоритмів компресії;
- визначити метрики, які будуть використані для проведення експерименту та подальшого оцінювання;
- визначити функціональні вимоги для консольного застосунку, який буде використано для проведення експерименту, та розробити його;
- виміряти та підрахувати значення обраних метрик для кожного з алгоритмів;
- надати рекомендації щодо використання кожного з методів.

# 2 ІСНУЮЧІ МЕТОДИ ТА АЛГОРИТМИ

## 2.1 Алгоритм Хаффмана

Алгоритм Хаффмана є одним із найвідоміших і найбільш широко використовуваних алгоритмів компресії даних без втрат. Він був розроблений Девідом Гаффманом у 1952 році і став потужним інструментом для стиснення текстових даних, а також інших типів даних з нерівномірним розподілом символів.

Основна концепція кодування Хаффмана базується на частоті появи символів у послідовності. Символи, що зустрічаються частіше, отримують короткі коди, тоді як символи, що зустрічаються рідше, отримують довші коди. Це рішення зумовлене бажанням мінімізувати використання місця для зберігання інформації про найчастіші символи після опрацювання всього введення. Таким чином, коди для символів, що зустрічаються найчастіше, займають менше простору (порівняно з їхнім поданням в оригіналі), тоді як для рідкісних символів дозволяється використовувати довші коди, довжиною таких символів можна знехтувати, бо вони й так рідко зустрічаються і сильно вплинути на підсумковий розмір не зможуть [11].

При використанні кодів різної довжини для кожного окремого символу виникає проблема однозначного декодування набору бітів. Треба якось відокремлювати один символ від іншого. Для розуміння проблеми давайте розглянем приклад із наступний рядок “aabacdab”. Рядок містить 8 символів, якщо використовувати кодування фіксованої довжини, для зберігання цього рядка потрібно 64 біти. Однак, з огляду на частоту символів "a", "b", "c" і "d", що відповідно дорівнює 4, 2, 1, 1, ми можемо представити цю послідовність більш компактно. Наприклад, можна закодувати символ "a" одним бітом (0), символ "b" - двома бітами (11), а символи "c" і "d" - трьома бітами (100 і 011 відповідно). У результаті в нас вийде наступна таблиця кодування:

Таблиця 1 - Таблиця кодування символів

| | |
|---|---|
| a | 0 |
| b | 11 |
| c | 100 |
| d | 11 |

Таким чином, послідовність "aabacdab" буде закодовано у вигляді 00110100011011, де кожен символ замінено відповідним бітовим кодом (0 для "a", 11 для "b", 100 для "c", 011 для "d"). Однак виникає основна проблема під час декодування цього рядка. При спробі декодування послідовності 00110100011011 виникає неоднозначність, тому що її можна інтерпретувати таким чином:

- 0 | 011 | 0 | 100 | 011 | 0 | 11  adacdab
- 0 | 0 | 11 | 0 | 100 | 0 | 11 | 011  aabacabd
- 0 | 011 | 0 | 100 | 0 | 11 | 0 | 11  adacabab

Для запобігання цій невизначеності необхідно забезпечити дотримання префіксного правила в нашому кодуванні. Префіксне правило гарантує, що кожен код може бути декодований єдиним унікальним чином. Це правило передбачає, що жоден код не є префіксом іншого. Під кодом тут розуміють біти, що використовуються для представлення певного символу. У наведеному вище прикладі префіксне правило порушується, оскільки 0 є префіксом 011. Дотримання префіксного правила забезпечує однозначне декодування (і навпаки), що є ключовим аспектом цього підходу.

Розглянемо попередній приклад з іншого боку. Цього разу ми присвоїмо коди символам "a", "b", "c" і "d" таким чином, щоб вони відповідали префіксному правилу.

Таблиця 2 - Таблиця кодування з використанням префіксного правила

| | |
|---|---|
| a | 0 |
| b | 10 |
| c | 110 |
| d | 111 |

Використовуючи вищеописане кодування, послідовність "aabacdab" буде представлена як 00100100011010, де кожен символ має код (0 для "a", 10 для "b", 100 для "c", 011 для "d"). Це кодування забезпечує однозначне декодування, дозволяючи відновити вихідний рядок "aabacdab" із послідовності 00100100011010.

Будувати коди із використанням префіксного правила можна за допомогою бінарного дерева. У бінарному дереві кожен вузол може бути або кінцевим, являючи собою лист (кінцевий вузол), або внутрішнім вузлом. Спочатку всі вузли розглядаються як листя (кінцеві вузли), що представляють окремі символи та їхні ваги (частоту появи). Внутрішні вузли містять інформацію про вагу символу і посилання на два вузли-спадкоємці. Згідно із загальною угодою, використання біта "0" вказує на проходження по лівій гілці, а біт "1" - по правій. У повному дереві Хаффмана існує N листків і N-1 внутрішніх вузлів. Ефективним методом при побудові дерева Хаффмана є відкидання невикористовуваних символів, що сприяє отриманню оптимальних кодів змінної довжини.

Для побудови дерева Хаффмана використовується черга з пріоритетами, де вузлу з найменшою частотою буде присвоєно вищий пріоритет. Нижче описано кроки побудови:

1) Створіть вузол-лист для кожного символу і додайте їх у чергу з пріоритетами.
2) Поки в черзі більше одного аркуша, робимо таке:

а) Видаліть два вузли з найвищим пріоритетом (з найнижчою частотою) з черги;

б) Створіть новий внутрішній вузол, де ці два вузли будуть спадкоємцями, а частота появи дорівнюватиме сумі частот цих двох вузлів.

в) Додайте новий вузол у чергу пріоритетів.

3) Єдиний вузол, що залишився, буде кореневим, на цьому побудова дерева закінчиться.

Припустімо, ми маємо текст, що містить тільки символи "a", "b", "c", "d" і "e", з відповідними частотами зустрічальності: 15, 7, 6, 6 і 5. Нижче подано графічні ілюстрації, що відображають етапи виконання алгоритму.

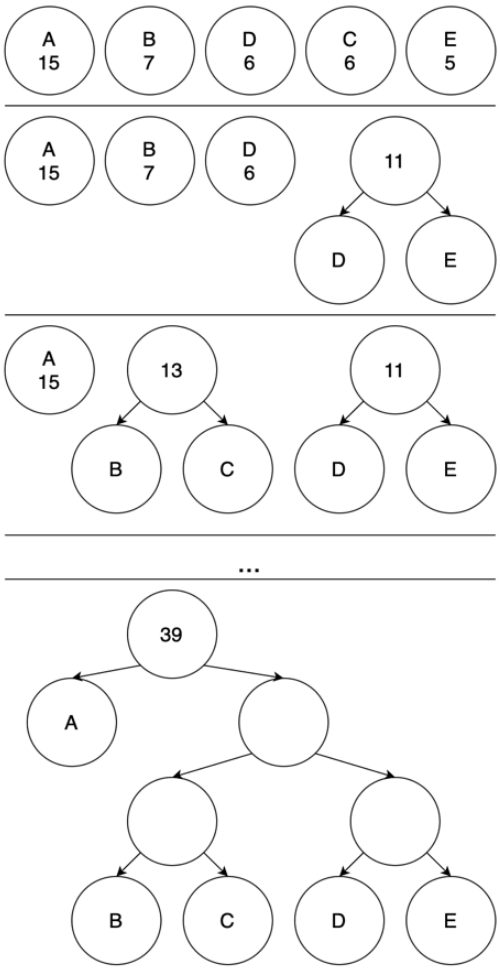


Рисунок 2.1 - Алгоритм побудування дерева Хаффмана (ввиконано самостійно)

Шлях від кореня до будь-якого кінцевого вузла зберігатиме оптимальний префіксний код (також відомий, як код Хаффмана), що відповідає символу, пов'язаному з цим кінцевим вузлом. Згідно із загальною угодою, використання біта "0" вказує на проходження по лівій гілці, а біт "1" - по правій. Склавши усі номери переходів по дереву Хаффмана до певного символу і буде собою являти код цього символу із використанням префіксного правила

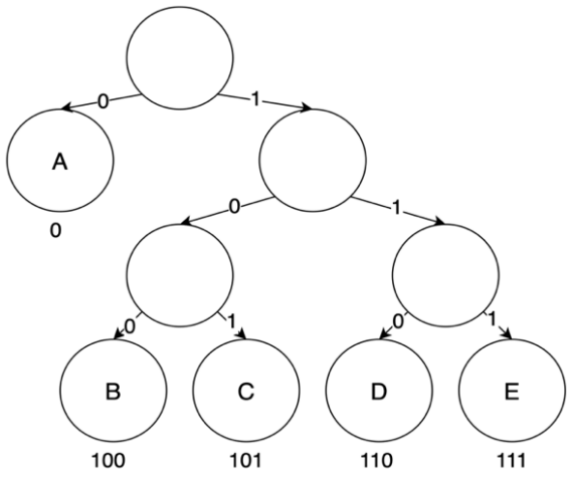


Рисунок 2.2 - Дерево Хаффмана із кодами символів (ввиконано самостійно)

## 2.2 Арифметичне кодування

Основна ідея алгоритму полягає у тому, що кожному символу або послідовності символів присвоюється певне число на відрізку [0, 1]. Увесь символьний потік представлений одним числом, яке лежить в інтервалі, що відповідає кодованому символу або послідовності символів. Для прикладу можна взяти просту строку тексту:

abracabra

З початку необхідно порахувати частотність появи символів і записати їх у таблицю у вигляді коефіцієнтів, що позначають імовірність їхньої появи. Тільки ще наприкінці тексту ми додамо деякий символ, що буде позначати кінець рядка, навіщо це потрібно, стане зрозумілим у процесі декодування. Частотна таблиця для нашої строки наведена у таблиці 3.

Таблиця 3 - Таблиця частотність появи символів у вигляді коефіцієнтів

| | |
|---|---|
| a | 0.4 |
| b | 0.2 |
| r | 0.2 |
| c | 0.1 |
| END | 0.1 |

Наступним кроком необхідно відміряти робочий відрізок від нуля до одного й розбити його на відрізки, прямо пропорційні нашим імовірностям, тобто перший відрізок відповідає літері A і складає 40% від довжини. Другий відрізок відповідає літері B с 20% і так далі для усіх інших літер вихідного тексту (Рисунок 3.3).

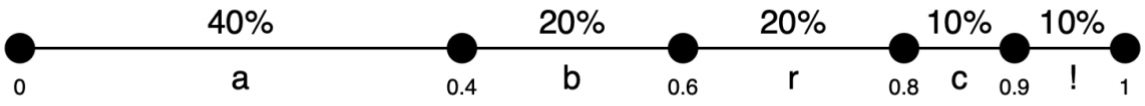


Рисунок 2.3 - Робочий відрізок із імовірностями (ввиконано самостійно)

Далі для того щоб закодувати наш текст, беремо найперший символ і йому відповідає перший відрізок, який тепер замість основного стає нашим робочим відрізком, на якому так само, у відсотковому співвідношенні, необхідно розмістити всі наші точки (Рисунок 3.4).

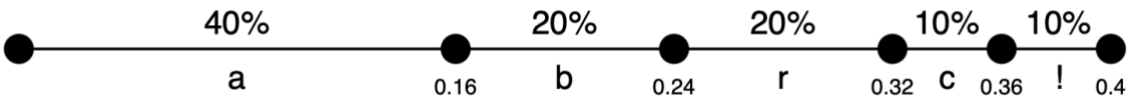


Рисунок 2.4 – Робочий відрізок із імовірностями (ввиконано самостійно)

Другий символ B відповідає другому відрізку, тож тепер той стає робочим відрізком, на якому ми знову розставляємо крапки (Рисунок 2.5).

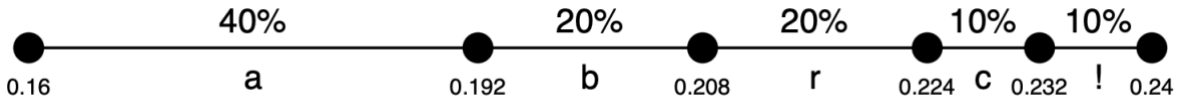


Рисунок 2.5 – Робочий відрізок із імовірностями (ввиконано самостійно)

Подібні кроки необхідно повторювати доки ми не добираємося до останнього символу рядка, до цього моменту в нас залишиться лише один відрізок із двома крапками, тож стислою інформацією стане дробова частина числа між двома крапками, які можна записати мінімальною кількістю знаків (Рисунок 2.6) Виходить наше закодоване число – 0.2132569.

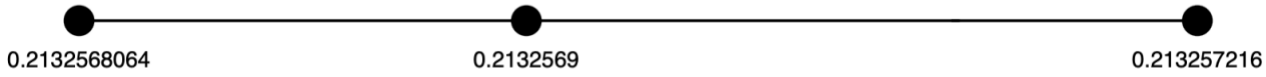


Рисунок 2.6 – Робочий відрізок із імовірностями (ввиконано самостійно)

Тепер для того, щоб із цього числа відновити вихідну інформацію, нам треба разом зі стислою інформацією зберігати і таблицю ймовірностей, яку буде використовувати декодер для побудови відрізків точно за таким самим алгоритмом дії. Спочатку потрібно розбити робочій відрізок на п'ять частин, згідно збереженої таблиці, і дивимося, до якої частини потрапляє наше стисле число. Воно потрапляє до першого відрізка - значить, перша буква вихідного тексту A (Рисунок 2.7).

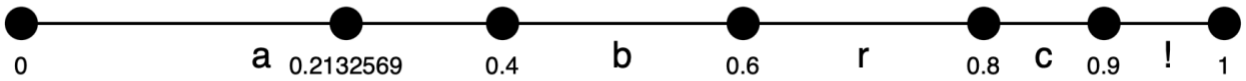


Рисунок 2.7 – Робочий відрізок із імовірностями (ввиконано самостійно)

Тепер знову як робочий відрізок вибирають той, у якому опинилася наша точка, знову ділимо його на п'ять частин і знову дивимося, куди потрапило наше число, куди воно потрапить, та буква й буде другою літерою вихідного повідомлення, а щоб декодер зрозумів, у який момент створення відрізків йому треба зупинитися, ми додали спеціальний символ, що позначає кінець рядка.

## 2.3 LZ77

У 70-х роках починають активно створюватися словникові методи стиснення, які взагалі не використовують статистику частотності символів і не використовують жодних кодів змінної довжини, натомість вони кодують цілі послідовності символів, що раніше зустрічались у словнику, а як стислі дані записують лише мітки на зміщення збігів у цьому словнику. Уперше ідею стиснення даних з використанням словника в 1977 році здійснили два ізраїльські математики Якоб Зів і Абрахам Лемпель. Алгоритм було названо за першими літерами їхніх прізвищ - LZ77. Він вирізняється своєю ефективністю, а також тим, що він водночас простий у розумінні та потребує складної реалізації.

Цей метод включає в себе використання "ковзного" словника розміром V від 2 до 32 кілобайт. Якщо наступний вхідний рядок s тексту збігається з рядком зі словника, його замінюють покажчиком ptr на цей рядок у формі ptr=<префікс, відстань, довжина>, де префікс дорівнює 1. Після цього поточна позиція у вихідному тексті pos зміщується на довжину символів уперед. Тут поняття словника і рядка трохи відрізняється від звичайного: рядком вважається будь-яка безперервна послідовність символів (літер), починаючи з певної позиції в словнику або тексті, довжиною $|s| \leq s_{max}$. Зазвичай $s_{max}$ вибирається в межах від 16 до 256 байт; словник являє собою |V| байт стискуваного тексту, що

передують поточному символу, що кодується, у позиції pos. Тому він названий "ковзаючим" словником, оскільки він, ніби ковзає вздовж тексту від його початку до кінця, оновлюючи найактуальнішу інформацію про його зміст. Оскільки перед початком процесу компресії перед першим символом нічого немає, словник спочатку порожній (або заповнений певним символом).

Словник містить будь-які рядки довжиною не більше smax, що починаються з будь-якої позиції в словнику. Таким чином, у словнику завжди присутній |V|·smax слів. У разі відсутності рядка s у словнику генерується код chr у форматі chr=<префікс, символ>, де префікс дорівнює 0, а символ представляє поточний символ вихідного тексту (який перебуває в позиції pos). Префікс, як видно, необхідний для розрізнення кодів ptr і chr. Цей префікс вносить додаткову інформацію в алгоритм кодування LZ77, що може призвести до збільшення надмірності.

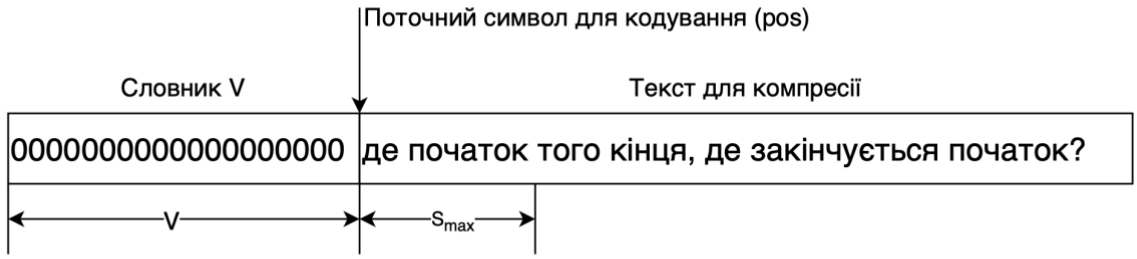


Рисунок 2.8 - візуалізація роботи алгоритма LZ77 (ввиконано самостійно)

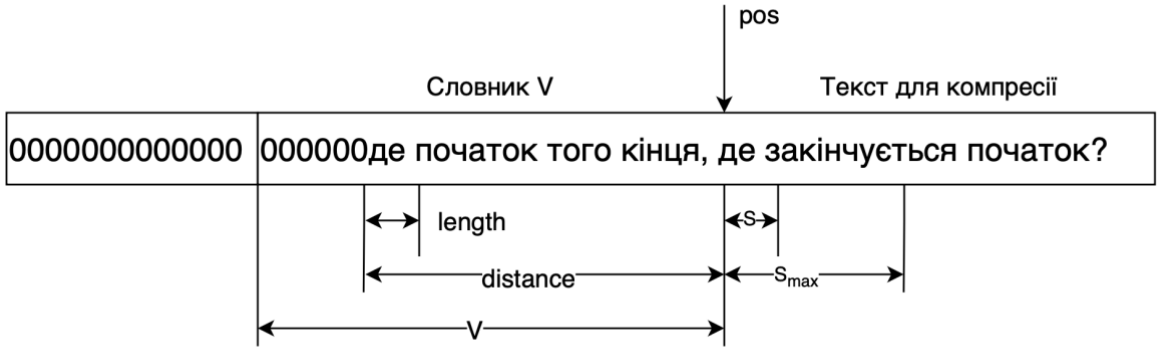


Рисунок 2.9 – візуалізація роботи алгоритма LZ77 на 24-му символі (ввиконано самостійно)

На рисунку 3.9 представлено зображення словника і закодованого тексту на 24-му символі. У цей момент алгоритм виявив, що слово "де" збігається з аналогічним словом у словнику завдовжки 3 байти, яке знаходиться на відстані 23 байти від позиції pos. У результаті буде сформовано код ptr=<1,23,3>. Тепер визначимо довжину коду ptr.

Якщо розмір коду ptr перевищує значення length у байтах, це призведе до зайвої надмірності. У зв'язку з цим в алгоритмі вводиться поріг (threshold), який зазвичай становить 2-3 байти. Якщо довжина рядка, що збігається з поточним, менша за цей поріг, його не піддають кодуванню (або, точніше, кодують явним чином, символ за символом, з використанням chr-кодів).

Використання множинних двозв'язних списків є гідною альтернативою послідовному пошуку, даючи змогу прискорити процес більш ніж у 10 разів. Хоча вона не досягає такої високої швидкості, як бінарні дерева, реалізація цього методу набагато простіша. Основна ідея полягає у створенні 256 (або більше) двозв'язних списків, у кожному з яких містяться слова, що починаються з однієї й тієї самої літери (або 1.5-2 літери). Очевидно, що пошук наступного слова виконується тільки в тому списку, де слова починаються з тієї самої літери, що й перша буква поточного символу.

Зворотне відновлення тексту відбувається набагато швидше і простіше. Оскільки на момент декодування кожного символу весь попередній текст уже відомий, декодувальнику не потрібно передавати словник разом з упакованими даними. Декомпресор будує словник за тим самим алгоритмом, що й компресор, використовуючи вже отримані символи. Якщо декодувальник виявляє (за префіксом), що надійшов chr-код, він просто копіює символ у вихідний потік і в словник, потім пересуваючи словник на одну позицію після тексту. У разі надходження ptr-коду декодувальник копіює length символів, починаючи з позиції distance у словнику. Важливо зазначити, що distance відраховується від кінця словника до його початку.

## 2.4 Алгоритм RLE

Алгоритм RLE (Run Length Encoding, кодування довжин серій) являє собою швидкий, простий і легко зрозумілий метод стиснення даних, який іноді виявляється дуже ефективним. Його суть полягає в тому, що будь-яка послідовність вхідних символів, що повторюються, замінюється набором із трьох вихідних символів: перший байт префікса вказує на те, що виявлено послідовність, що повторюється; другий байт визначає довжину цієї послідовності, а третій символ є вхідним символом, що повторюється, сам по собі - <префікс, довжина, символ>.

Наприклад: нехай є (шістнадцятиричний) текст із 20 байт вигляду:

05 05 05 05 05 01 01 03 03 03 03 03 03 05 03 FF FF FF FF

Виберемо як префікс байт FF. Тоді на виході архіватора ми отримаємо послідовність:

06 05 FF 02 01 FF 06 03 FF 01 05 FF 01 03 FF 04 FF

Довжина цієї послідовності становить 18 байт, що свідчить про певний рівень стиснення. Однак варто зазначити, що під час кодування певних символів відбувається збільшення розміру вихідного коду (наприклад, з 01 01 виходить FF 02 01). Очевидно, що безглуздо кодувати окремі символи або ті, що повторюються двічі (чи тричі) - їх краще записати явно. У результаті виходить нова послідовність, яка має довжину лише 13 байт. Досягнутий ступінь компресії: 13/20 = 65%.

06 05 01 01 FF 06 03 05 03 FF 04 FF

Легко помітити, що можливий збіг префікса з одним із вхідних символів. У таких випадках вхідний символ може бути замінений своїм "префіксним" представленням, наприклад, це може бути еквівалентно FF 01 FF замість використання одного байта. Таким чином, правильний вибір префікса відіграє ключову роль в ефективності самого алгоритму стиснення. Якщо у вихідному тексті часто зустрічаються поодинокі символи FF, то розмір вихідного тексту

може навіть перевищити розмір вхідного. У загальному випадку рекомендується вибирати як префікс найрідкісніший символ вхідного алфавіту.

Можна зробити наступний крок у збільшенні ступеня стиснення, об'єднавши префікс і довжину в один байт. Припустимо, префікс представлений числом F0...FF, де друга цифра визначає довжину (length) в діапазоні від 0 до 15. У результаті вихідний код буде двобайтним, але за рахунок цього ми обмежуємо представлення довжини від 255 символів до 15, що також звужує вибір префікса. Таким чином, вихідний текст для цього прикладу матиме наступну форму. Довжина-10 байт, ступінь стиснення-50%.

05 F2 01 F6 03 05 03 F4 FF.

Далі, оскільки ми домовилися не кодувати послідовності довжиною від 0 до 3, зручно використовувати код довжини (length) зі зсувом на три, де 00 представляє 3, 0F - 18, FF - 258, що дає змогу ефективно упаковувати довші ланцюжки за один раз.

Якщо поодинокі символи зустрічаються досить рідко, може бути корисною модифікація алгоритму RLE без використання префікса, використовуючи тільки <length, symbol>. У цьому разі поодинокі символи також обов'язково мають бути закодовані в префіксній формі, щоб декодувальник міг їх відрізнити у вихідному потоці. Приклад такої модифікації подано наступним чином. При цьому довжина становить 12 байт, а ступінь стиснення - 60%.

06 05 02 01 06 03 01 05 01 03 04 FF

Можливий варіант алгоритму, за якого замість довжини (length) кодується позиція щодо початку тексту (distance) першого символу, що відрізняється від попереднього. Для цього прикладу це призведе до вихідного рядка.

01 05 07 01 09 03 0F 05 10 03 11 FF.

## 2.5 Метод адаптивної компресії даних

Адаптивний метод, виходячи зі своєї назви, може адаптуватися під різні набори даних, навіть у межах одного файлу. Головною особливістю методу є використання особливостей усіх методом компресії раніше розглянутих у цій роботі. Кожен із цих методів має переваги над іншими тільки в певних умовах і на певних властивостях даних. У таблиці 4 наведено всі раніше розглянуті методи компресії даних разом з їхніми умовами найвищої ефективності.

Таблиця 4 – Умови максимальної ефективності для різних методів компресії даних

| Метод компресії даних | Умова максимальної ефективності |
|---|---|
| Алгоритм Хаффмана | добре підходить для даних з нерівномірним розподілом символів |
| LZ77 | ефективний для стиснення даних з повторюваними блоками |
| RLE | добре працює для даних з довгими серіями однакових символів |
| Арифметичне кодування | ефективне для даних з відомим і добре модельованим розподілом символів |

Виходячи з переваг кожного з методів, необхідно застосовувати їх тільки на блоки дані, де вони будуть максимально ефективні. З цією метою можна розбити вихідний файл на блоки рівної довжини і провести аналіз кожного з отриманих блоків. Кожен із блоків має бути перевірений на такі властивості:

- Таблиця частотності символів;
- Кількість і довжина повторюваних блоків або біт, що повторюються;
- Кількість і довжина серій однакових символів або біт.

На основі отриманих даних можна вибрати для кожного з блоків певний метод компресії, який буде максимально ефективний у кожному конкретному блоці.

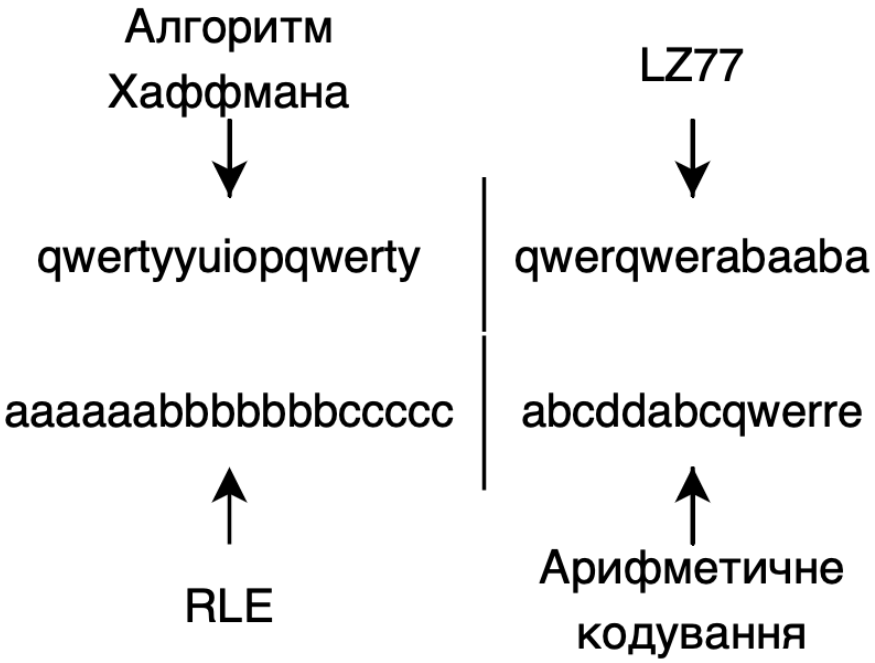


Рисунок 2.10 – Приклад розподілу методів компресії даних у адаптивному алгоритмі (виконано самостійно)

Для подальшого декодування інформації необхідно зробити позначки де і який метод компресії було використано, а також межі всіх створених нами блоків. Для спрощення блоки можна взяти рівної довжини. Дані для декодування необхідно записати на початку файлу. Першим буде записано розмір блоку та їхню кількість. Далі будуть записані методи компресії, які були використані для кодування інформації в послідовності блоків. Тобто якщо закодованих блоків вийшло 10, то на початку файлу буде записано список із 10 методів компресії даних.

Декодування буде проводитися за допомогою метаданих, які записані на початку файлу. Необхідно буде прочитати список методів декодування і згідно з цим списком проводити декодування блоків один за одним.

Швидкість роботи адаптивного методу під час кодування може бути значно меншою внаслідок того, що потрібно провести досить великий аналіз перед безпосередньо кодуванням файлу. Швидкість самого кодування файлу не буде сильно відрізнятися від інших методів, також як і швидкість декодування. Це відбувається через те, що ми використовуємо ті самі методи компресії даних.

## 2.6 Визначення методики проведення дослідження

Варто враховувати, що успішність будь-якого застосунку значною мірою залежить від його продуктивності. Це стосується не лише дуже критичних моментів для застосунку, а й на усіх етапах його розробки. В процесі проектування має особливо важливе значення саме оцінка програмного забезпечення, оскільки вона надає змогу попередньо оцінити продуктивність застосунку та врахувати можливі ризики проєкту й етапів розроблення. Це також дає розробникам можливість порівняти необхідні витрати та майбутню вартість проєкту. Таким чином, під час проектування програмного забезпечення слід приділяти увагу продуктивності на всіх етапах розроблення, а не лише в критичних моментах. Попередня оцінка програмного забезпечення допомагає знизити ризики проекту і підвищити його шанси на успіх.

Швидкість роботи системи може легко погіршитися, якщо використовуються невідповідні бібліотеки, не тільки для управління станом. Крім того, метрики та їхні граничні значення зазвичай визначають індивідуально для кожного проєкту, якщо такі передбачені. Тому важливо ретельно підходити до вибору бібліотек та інших інструментів з метою забезпечення максимальної продуктивності програми. Крім того, слід заздалегідь встановлювати метрики та граничні значення, щоб забезпечити ефективну роботу застосунку та підтримувати його високу швидкість.

Для виміру швидкодії можна використати потужну .NET бібліотеку - BenchmarkDotNet, яка призначена для проведення бенчмарків, тобто вимірювання продуктивності коду в різних умовах. Ця бібліотека спрощує процес створення тестових наборів, визначення параметрів тестування, проведення автоматичних тестів і отримання звітів із результатами у різних форматах. Використання BenchmarkDotNet дозволяє здійснювати порівняльний аналіз різних варіантів реалізації алгоритмів, що сприяє знаходженню оптимального варіанту для конкретного випадку використання.

BenchmarkDotNet надає наступну інформацію для оцінки продуктивності:

- середнє значення часу виконання для кожного методу;
- мінімальне і максимальне значення часу виконання методу;
- середнє значення часу виконання для одного виклику методу;
- графік, який ілюструє розподіл викликів методів у часі;
- обсяг використаної пам'яті під час виконання методу;
- статистичні параметри, такі як середнє значення, медіана, стандартне відхилення та коефіцієнт варіації для кожного методу.

Ці критерії сприяють проведенню ретельного аналізу ефективності програмного коду та виявленню можливостей для його оптимізації. Варто зазначити що окрім показників бенчмаркігку будуть враховуватися інші критерії відносно компресії даних, такі як:

- коофіцент компресії;
- швидкість компресії та декомпресії;
- адаптивність.

Однак було враховано метрики, створені BenchmarkDotNet [12]. Передбачалося, що найбільший вплив на розмір вхідних даних матиме метод скорочення посилань.

Для проведення дослідження та порівняння алгоритмів компресії даних були визначені наступні показники:

- коефіцієнт компресії даних;
- швидкість компресії даних;
- швидкість декомпресії даних;
- адаптивність;
- кількість пам'яті, використаної під час виконання методу.

Також існують деякі обмеження у використанні певних алгоритмів компресії, оскільки певні алгоритми краще працюють на специфічних наборах даних. Ефективність алгоритмів компресії залежить від типу даних, які піддаються компресії. Різні алгоритми компресії можуть виявляти різну ефективність залежно від структури даних і характеристик. Деякі загальні тенденції впливу типів даних на ефективність стиснення включають:

- ділянки, що повторюються: Якщо в тексті часто повторюються однакові фрагменти або символи, алгоритми стиснення, такі як Lempel-Ziv, можуть ефективно справлятися з усуненням повторів;
- специфічні структури даних: Деякі алгоритми стиснення можуть бути більш ефективними для певних структур даних. Наприклад, алгоритми, засновані на словнику, можуть добре працювати з текстами, що містять багато повторюваних слів;
- текстові дані з високою ентропією: Якщо текст містить високий ступінь випадковості або ентропії, наприклад, коли кожен символ зустрічається приблизно з однаковою ймовірністю, то стиснення може бути менш ефективним;
- Розмір словника: В алгоритмах стиснення на основі словника ефективність може залежати від розміру доступного словника. Великі словники можуть бути більш ефективними для обробки певних типів даних.

# 3 СТВОРЕННЯ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ ДОСЛІДЖЕННЯ

## 3.1 Функціональні вимоги

Для дослідження різних методів компресії даних на швидкодію та ефективність, було створено консольний застосунок на платформі .NET 8 з використанням мови програмування C#[13].

Розроблений консольний застосунок має відповідати наступним функціональним умовам:

- програма дозволяє ввести шлях до файлу;
- програма дозволяє вибрати компресію або декомпресію для вибраного файлу;
- програма дозволяє вибрати алгоритм для компресії файлу;
- програма обробляє вибраний файл;
- програма відображає статистичні дані щодо процесу компресії або декомпресії.

Згідно з наведеними функціональними вимогами, тестовий застосунок повинен мати можливість працювати з різними алгоритмами, які користувач може обрати для скорочення посилання. Реалізація цих вимог за допомогою різних методів компресії даних дозволяє проводити порівняльний аналіз без будь-яких припущень або змін у вихідних даних, оскільки експеримент проводиться на тих самих даних. Це забезпечує отримання точних і порівняльних результатів.

Згідно з вищезазначеними функціональними вимогами, у додатку не буде присутніх сутностей. Тестова програмна реалізація зосереджена виключно на компресію та декомпресію даних. Основний функціонал описано нижче:

- вибір алгоритму компресії даних;
- компресія або декомпресія даних;
- відображення статистичних даних.

Презентаційна частина програми має бути створена на платформі .NET з використанням мови програмування C# і залишатися незмінною протягом всього дослідження. Алгоритми компресії даних мають бути реалізовані окремо за допомогою обраних бібліотек, які також розроблені на базі .NET.

## 3.2 Розробка інтерфейсу програмного застосунку

Для створення користувацького інтерфейсу було використано фреймворк ASP.NET Core на базі .NET 8. Цей фреймворк пропонує підхід до формування інтерфейсу як композицію з різних компонентів. ASP.NET Core є кросплатформовим фреймворком з відкритим вихідним кодом, розробленим для створення веб-додатків на платформі .NET спільно зі спільнотою Microsoft. Його висока продуктивність, що перевищує ASP.NET, робить його дуже популярним. ASP.NET Core має модульну структуру і підтримує операційні системи, такі як Windows, Linux та macOS. Тестовий додаток можна запустити на локальній машині у будь-якому терміналі.

Презентаційна частина буде являти собою консольний застосунок. Виходячи з вимог до програмного забезпечення, консольного додатку буде достатньо для проведення усіх необхідних експериментів та отримання результатів. Такий підхід значно пришвидшує процес розробки додатку, адже не потрібно витрачати час на будування графічного інтерфейсу, який не є необхідним для проведення дослідження.

Для взаємодії із консольним застосунком було використано бібліотеку Spectre. Ця бібліотека дозволяє розробляти консольні застосунки надаючи інструменти для будування команд та їх параметрів. Також ця бібліотека надає безліч різноманітних графічних елементів, які можуть відображатися у вікні терміналу. Головна ціль Spectre це зменшення обсягу робіт, що потрібні для побудування взаємодії користувача та консольного застосунку та скороченні часу, необхідного для опису документації застосунку.

Запуск застосунку без жодної команди передбачає відображення списку команд, які підтримую консольний застосунок. На рисунку 4.1 зображено початкове меню застосунку.

```
❯ ./CompressApp.Console
USAGE:
    CompressApp.Console.dll [OPTIONS] <COMMAND>

OPTIONS:
    -h, --help       Prints help information
    -v, --version    Prints version information

COMMANDS:
    compress      Compress given file
    decompress    Decompress given file
```

Рисунок 3.1 – головне меню консольного застосунку

Кожна команда має свій набір параметрів для роботи. Доступ до параметрів команд можна отримати при введені самої команди та класичного параметру -h (--help). Після цього у термінали з’являться усі параметри для обраної команди. На рисунку 4.2 зображено приклад набору параметрів для одної із команд.

```
❯ ./CompressApp.Console compress -h
DESCRIPTION:
Compress given file

USAGE:
    CompressApp.Console.dll compress [OPTIONS]

OPTIONS:
    -h, --help       Prints help information
    -v, --version    Prints version information
    -s               Show statistic data
    -p               Path of source file
    -P               Path of end file
    -a               Algorithm for compression (A - arithmetic coding, H -
                     huffman, L - LZ77, R - RLE, M - adaptive)
```

Рисунок 3.2 – набор параметрів для команди compress

Після створення консольного застосунку можна визначити головні компоненти системи. Таким чином, ми можемо виділити наступні компоненти:

- IcompressAlgorithm;
- CompressCommand;
- DecompressCommand;
- Huffman;
- LZ77;
- RLE;
- Adaptive;
- ArithmeticCoding.

На діаграмі класів можна побачити, що інтерфейс ICompressAlgorithm має два методи для компресії та декомпресії даних. Метод компресії даних на вхід отримую строку, яку необхідно закодувати та шлях до файлу куди треба зберегти вихідний результат роботи алгоритму. Алгоритм має сам записати результат до файлу, тому що кожен з них має свою унікальну структуру файлу. Метод декопресії має лише один параметр, це шлях до файлу де знаходиться дані для декомпресії. Цей метод також читає файл самостійно, адже кожен файл має свою унікальну структуру.

Створення інтерфейсів є ключовою складовою розробки програмного забезпечення, оскільки це забезпечує більшу гнучкість та спрощує обслуговування коду. На діаграмі класів видно, що інтерфейс ICompressAlgorithm має 5 реалізацій. Це алгоритми компресії даних, які розглядалися раніше, а саме:

- RLE;
- Huffman;
- LZ77;
- Adaptive;
- ArithmeticCoding.

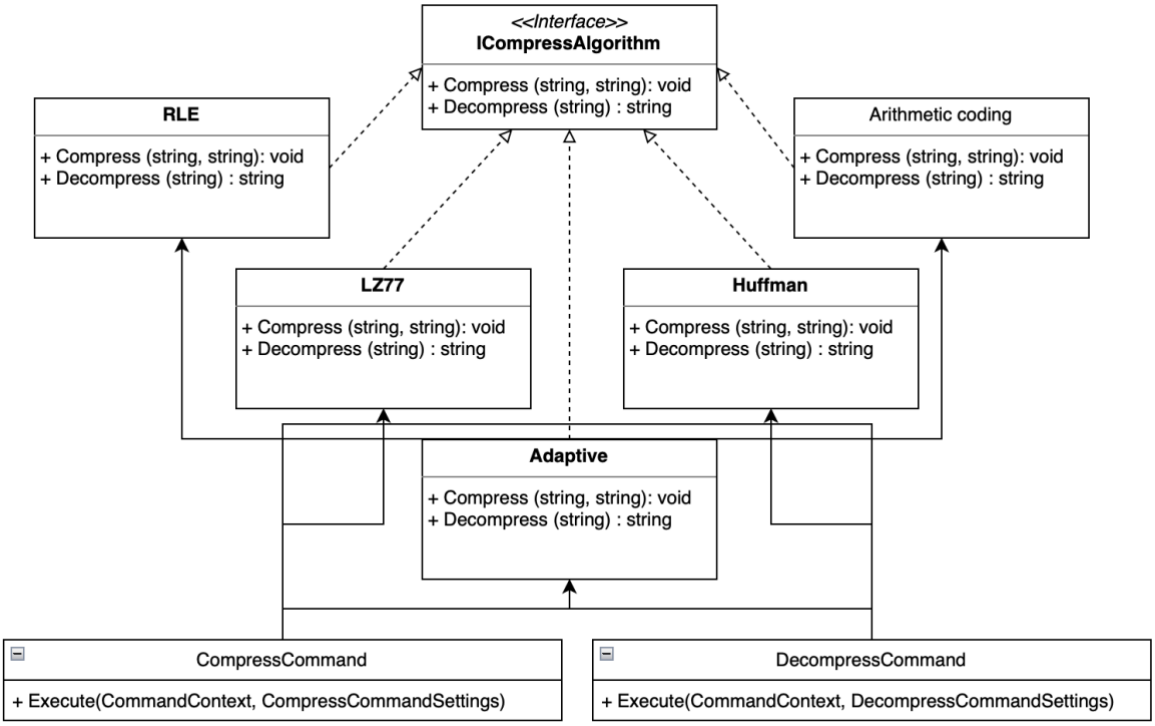


Рисунок 3.3 – діаграма класів консольного застосунку (ввиконано самостійно)

Використання інтерфейсів дозволяє досягти слабкої зв'язності між різними компонентами програми. Завдяки ним, класи можуть взаємодіяти, обмінюючись повідомленнями, не знаючи деталей реалізації один одного. Це сприяє підвищенню гнучкості та можливостей модифікації системи, оскільки різні компоненти можуть замінюватися один одним без порушення роботи інших компонентів. Таким чином, використання інтерфейсів сприяє модульності програми і полегшує її розширення та підтримку.

У командах CompressCommand та DecompressCommand залежності від сервісів для компресії даних передаються через конструктор. Використання вбудованого контейнера ін'єкції залежностей дозволяє досягти слабкої зв'язності. Далі наведено приклад коду для реєстрації залежностей у розробленому консольному застосунку (див. додаток В).

```
    var registrations = new ServiceCollection();
registrations.AddSingleton<ICompressAlgorithm, Adaptive>();
registrations.AddSingleton<ICompressAlgorithm, Huffman>();
```

```
registrations.AddSingleton<ICompressAlgorithm, LZ77>();
registrations.AddSingleton<ICompressAlgorithm, RLE>();
registrations.AddSingleton<ICompressAlgorithm, ArithmeticCoding>();
```

ASP.NET Core підтримує патерн проектування програмного забезпечення з використанням ін'єкції залежностей (DI), яка є методом для реалізації Інверсії Керування між класами та їх залежностями. Реєстрація залежностей відбувається за допомогою методу AddSingleton, який контролює об'єкт залежності протягом життєвого циклу додатку. Метод AddSingleton створює лише один об'єкт протягом усього життєвого циклу додатку, тобто для кожного запиту на місцях, де залежність потребує реалізації певного сервісу, буде створено лише один об'єкт. Використання AddSingleton є корисним в тих випадках, коли потрібно обмежити алокацію об'єктів в оперативній пам'яті (кількість).

У програмі були реалізовані всі необхідні функції для успішного проведення дослідження.

## 3.3 Реалізація алгоритмів компресії даних

### 3.3.1 Реалізація алгоритму Хаффмана

Алгоритму Хаффмана починається із побудови бінарного дерева, в якому кожен вузол може бути кінцевим або внутрішнім вузлом. Кожен вузол має містити символ та його частоту появи у тексті. Бінарне дерево має будуватися за наступним алгоритмом:

1) Створити вузол для кожного символу та додати їх у чергу з пріоритетами.
2) Поки черга має більше одного елементу треба робите наступне:
   а) Видалити два вузли з найвищим пріоритетом
   б) Створити новий внутрішній вузол, де ці два вузли будуть спадкоємцями, а частота появи дорівнюватиме сумі частот цих двохвузлів.
   в) Додати новий вузол у чергу пріоритетів.
3) Останній вузол буде кореневим для дерева Хаффмана

```
//1
var initTrees = new List<BinaryTree>();
foreach (var (key, value) in frequencyTable)
{
    var node = new Node
    {
        Frequency = value,
        Letter = key
    };
    initTrees.Add(new BinaryTree(node));
}
var sortedTrees = initTrees.OrderBy(x => x.Root.Frequency).ToList();

//2
while (sortedTrees.Count != 1)
{
    //a
    var firstTree = sortedTrees[0];
    var secondTree = sortedTrees[1];
    sortedTrees.RemoveRange(0, 2);

    //b
    var newNode = new Node
    {
        Frequency = firstTree.Root.Frequency + secondTree.Root.Frequency,
        LeftChild = firstTree.Root,
        RightChild = secondTree.Root
    };
    var newTree = new BinaryTree(newNode);

    //c
    sortedTrees.Add(newTree);
    sortedTrees = sortedTrees.OrderBy(x => x.Root.Frequency).ToList();
}

var binaryTree = sortedTrees.First(); //3
return binaryTree;
```

Використовуючи побудоване дерево, ми можемо створити таблицю кодування для поточного файлу. Для цього потрібно пройти по усьому дереву та зібрати усі символи. Шлях від кореню до кожного конкретного символу буде являти собою оптимальний префіксний код (код Хаффмана). Згідно із загальною угодою, використання біта "0" вказує на проходження по лівій гілці, а біт "1" - по правій. Склавши усі номери переходів по дереву Хаффмана до певного символу і буде собою являти код цього символу із використанням префіксного правила. Обхід бінарного дерева зазвичай виконується за допомогою

рекурсивного методу. Далі наведено приклад рекурсивного методу для обходу дерева та будування нової таблиці кодування.

```
var table = new Dictionary<char, string>();
FillEncodingArray(tree.Root, "", "");
return table;

void FillEncodingArray(Node node, String codeBefore, String direction)
{
    if (node.RightChild == null && node.LeftChild == null)
    {
        table[node.Letter] = codeBefore + direction;
    }
    else
    {
        FillEncodingArray(node.LeftChild, codeBefore + direction, "0");
        FillEncodingArray(node.RightChild, codeBefore + direction, "1");
    }
}
```

Після отримання таблиці кодування залишається лише закодувати вихідний текст використовуючи отриману таблицю. Для цього необхідно пройти по усім символам вихідного тексту та замінити їх на префіксний код із таблиці кодування. Запис до файлу відбувається після перетворення двійкового коду на байти. Файл має містити таблицю кодування там сам закодований текст. Нижче наведено код для перетворення вихідного текст у закодований.

```
var encodedSb = new StringBuilder();
foreach (var c in str)
{
    encodedSb.Append(encodingTable[c]);
}

var binaryString = encodedSb.ToString();

byte[] bytes = new byte[(binaryString.Length + 7) / 8];
for (int i = 0; i < bytes.Length; i++)
{
    bytes[i] = Convert.ToByte(binaryString.Substring(i * 8, Math.Min(8,
binaryString.Length - i * 8)), 2);
}

return bytes;
```

Останнім кроком є запис отриманого масиву байт до файлу. Спочатку необхідно записати до файлу таблицю кодування, яка надалі буде використана для декодування байтів, та сам масив байтів.

### 3.3.2 Реалізація алгоритму арифметичного кодування

Арифметичне кодування починається з будування таблиці частотності. Далі за допомогою таблиці частотності треба порахувати коефіцієнти, що позначають імовірність появи символів у вхідному файлі, також треба додати символ кінця файлу. Далі наведено код для розрахування коефіцієнтів.

```
var ranges = new List<Tuple<char, BigDecimal, BigDecimal>>();
var range = max - min;
var low = min;
var high = min;
foreach (var kvp in frequencies)
{
    var symbol = kvp.Key;
    var frequency = (BigDecimal)kvp.Value;
    high += frequency / (BigDecimal) total * range;
    ranges.Add(new Tuple<char, BigDecimal, BigDecimal>(symbol, low,
high));
    low = high;
}

return ranges;
```

Для коректної роботи алгоритму арифметичного кодування стандартні типи C# не зовсім підходять. Цей алгоритм потребує зберігання великої кількості символів після коми. C# має дробовий тип decimal, який може зберігати лише 29 символів після коми. Цього може бути недостатньо для великих обсягів даних. Для цього алгоритму було розроблено новий тип BigDecimal. Цей тип підтримує необмежену кількість символів після коми. Така властивість досягається за допомогою вбудованого типу в C#, BigInteger. BigInteger дозволяє зберігати числа з необмеженою кількістю символів. BigDecimal містить у собі BigInterger у якості контейнера для зберігання даних та окреме поле для позиції коми у числі. Далі наведено частина класу BigDecimal.

```
public class BigDecimal
{
    private BigInteger integer;
    private BigInteger scale;
}
```

Для повноцінного використання розробленого типу необхідно було реалізувати усі базові операції, такі як: додавання, віднімання, множення, розподіл, порівняння та приведення до рядка для коректного відображення.

Кодування вхідного файлу робиться за допомогою раніше визначених коефіцієнтів. Беремо відрізок, який відповідає першому символу вхідного файлу. Тепер цей відрізок стає нашим робочим відрізком, на якому так само, у відсотковому співвідношенні, необхідно розмістити всі наші точки. Для кожного символу необхідно виконати перерахування відрізків. Після остання символи залишається лише один відрізко, на якому потрібно знайти число з найменшою кількістю знаків. Далі наведено приклад коду для кодування тексту за допомогою арифметичного кодування.

```
str += EndSymbol;
var frequencies = str.GroupBy(c => c).ToDictionary(g => g.Key, g =>
g.Count());
var ranges = RecalculateRanges(frequencies, str.Length, 0, 1);

foreach (var c in str)
{
    var rangeInfo = ranges.Find(r => r.Item1 == c);
    if (rangeInfo == null) throw new ArgumentException("Cannot find
range");

    ranges = RecalculateRanges(frequencies, str.Length, rangeInfo.Item2,
rangeInfo.Item3);
}

var value = FindNumberWithMinDigits(ranges.First().Item2,
ranges.Last().Item3);

WriteToFile(filename, str.Length, frequencies, value);
```

Після отримання числа необхідно записати його до файлу разом із частотною таблицю, адже вона знадобиться для подальшого декодування.

### 3.3.3 Реалізація алгоритму RLE

Реалізація алгоритму RLE є найпростішою з усіх наведених алгоритмів у цій роботи. Концепція цього алгоритму є дуже простою. Кодування тексту за допомогою алгоритму RLE не потребує первинного аналізу файлу. Робота починається з послідовної обробки символів та скорочення серії однакових символів. Далі наведено код реалізації даного алгоритму.

```
var tokens = new List<Token>();
byte count = 1;

for (var i = 1; i <= str.Length; i++)
{
    if (i == str.Length || str[i] != str[i - 1])
    {
        tokens.Add(new Token(str[i - 1], count));
        count = 1;
    }
    else
    {
        count++;
    }
}

return tokens;
```

Останнім кроком необхідно записати результат роботи алгоритму до файлу. На відміну від раніше розглянутих алгоритмів, не потрібно записувати таблиць або словників.

### 3.3.4 Реалізація алгоритму LZ77

Робота алгоритму базується на так званому вікні. Вікно має довільний розмір, який може задаватися параметром алгоритму та він може мати різну ефективність та час роботи. Алгоритм намагається знайти найдовше співпадіння у словнику та заміняє це співпадіння на посилання у словнику. Алгоритм має словнику наступного типу.

```
internal class Token
{
    public int Offset { get; set; }

    public int Length { get; set; }
```

```
    public char? NextChar { get; set; }
}
```

Елемент словника складається з позиції співпадіння у словнику, довжини співпадіння та наступного символу після співпадіння.

Після кодування частини тексту вікно рухається далі по тексту до його кінця. Останній елемент словника буде мати лише перші два елементи, offsest та length, адже наступного символу немає. Далі наведено код алгоритму кодування.

```
var compressedData = new List<Token>();

int index = 0;
while (index < input.Length)
{
    int matchLength = 0;
    int matchIndex = 0;

    for (int i = Math.Max(0, index - WindowSize); i < index; i++)
    {
        int j = 0;
        while (j < BufferSize && index + j < input.Length && input[i + j]
== input[index + j])
        {
            j++;
        }

        if (j > matchLength)
        {
            matchLength = j;
            matchIndex = i;
        }
    }

    if (matchLength > 0)
    {
        if (index + matchLength >= input.Length)
        {
            compressedData.Add(new Token(index - matchIndex, matchLength,
null));
        }
        else
        {
            compressedData.Add(new Token(index - matchIndex, matchLength,
input[index + matchLength]));
        }

        index += matchLength + 1;
    }
    else
    {
```

```
            compressedData.Add(new Token(0, 0, input[index]));
            index++;
        }
    }

    return compressedData;
```

Останнім кроком треба записати отриманий словник до файлу, зберігаючи цілісність структури словника для подальшого коректного декодування файлу.

### 3.3.5 Реалізація адаптивного алгоритму

Адаптивний алгоритм базується на основі попередніх 4 алгоритмів та використовує усі їх переваги. Для роботи алгоритму необхідно розділити вхідний файл на блоки рівної довжини. Розмір цих блоків можна задавати як параметр алгоритму, що буде впливати на ефективність та швидкість роботи алгоритму. Кожен з блоків треба проаналізувати на наступні характеристики для виявлення найефективнішого алгоритму для кожного блоку:

- нерівномірний розподіл символів;
- повтори блоків;
- серії однакових символів;
- модельований розподіл символів.

Після визначення характеристик кожного блока можна виконувати кодування блоку. Під час запису вже закодованих даних до нового файлу, необхідна вказати алгоритм за допомогою якого було виконано кодування. Тобто блок буде складатися з одного символу, який вказує алгоритм кодування, та безпосередньо закодованих даних. Далі наведено код кодування.

```
var strings = str.SplitIntoBlocks(BlockSize);

if (File.Exists(filename))
{
    File.Delete(filename);
}

using var stream = File.Open(filename, FileMode.Append);
foreach (var block in strings)
{
```

```
    var res = new List<Tuple<long, string>>();
    var algorithm = AnalyzeBlock(block);

    var binaryWriter = new BinaryWriter(stream);
    binaryWriter.Write(algorithm.ToString());
    binaryWriter.Write(best.Length);
    algorithm.encode(block, stream);
}
```

Декодування відбувається в зворотному порядку. Необхідно зчитати символ алгоритму, довжину закодованого блоку там сам блок закодованих даних. Залишаться тільки декодувати зчитаний блок відповідним алгоритмом.

# 4 ОПИС ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ

У роботі було проведено два типи вимірювань:

- Вимірювання коефіцієнту стискання;
- Вимірювання швидкодії.

Усі розрахунки були отримані за допомогою раніше розробленого консольного застосунку. Платформа застосунку - .NET 8. Вимірювання швидкої виконувалось за допомогою класу Stopwatch. Коефіцієнт стискання розраховувався за наступною формулою:

$$K = \frac{s_1}{s_2},$$

де, K – коефіцієнт стиснення

$S_1$ – початковий розмір файлу

$S_2$ – розмір файлу після його стискання

Для проведення дослідження були обрані наступні типи файлів:

- Документ із текстом у форматі txt;
- Документ із текстом у форматі docx;
- Зображення у форматі bmp;
- Зображення у форматі jpg.

Таблиця 5 – Вимірювання швидкості та коефіцієнту стискання алгоритму Хаффмана

| Назва файлу | Початковий розмір | Розмір після компресії | Коефіцієнт компресії | Час компресії | Час декомпресії |
|---|---|---|---|---|---|
| 1.txt | 663442b | 217140b | 0,327 | 27ms | 1989ms |
| 2.txt | 1105612b | 361310b | 0,326 | 42ms | 5076ms |

| | | | | | |
|---|---|---|---|---|---|
| 3.txt | 4133129b | 1371817b | 0,331 | 156ms | 19914ms |
| 1.doc | 216144b | 196021b | 0,906 | 189ms | 21270ms |
| 2.doc | 372732b | 310022b | 0,831 | 290ms | 36123ms |
| 3.doc | 1402567b | 1066562b | 0,760 | 1394ms | 178884ms |
| 1.bmp | 818058b | 729979b | 0,892 | 902ms | 101754ms |
| 2.bmp | 3275658b | 2721545b | 0,830 | 4898ms | 280431ms |
| 3.bmp | 7372938b | 5946655b | 0,806 | 13933ms | 483275ms |
| 1.jpg | 474441b | 380324b | 0,801 | 328ms | 29185ms |
| 2.jpg | 744692b | 577176b | 0,775 | 552ms | 43954ms |
| 3.jpg | 1372378b | 1029390b | 0,750 | 1153ms | 104584ms |

З результатів вимірювань видно що алгоритм Хаффмана є ефективним на усіх типах файлів. Алгоритм показав доволі високу швидкість обробки кожного з виду файлів на різних розмірах файлів. Але слід зазначити, що він потребує значно більше часу для відновлення даних після компресії.

Таблиця 6 – Вимірювання швидкості та коефіцієнту стискання алгоритму LZ77

| Назва файлу | Початковий розмір | Розмір після компресії | Коефіцієнт компресії | Час компресії | Час декомпресії |
|---|---|---|---|---|---|
| 1.txt | 663442b | 463119b | 0,698 | 161ms | 21ms |

| | | | | | |
|---|---|---|---|---|---|
| 2.txt | 1105612b | 808896b | 0,731 | 252ms | 32ms |
| 3.txt | 4133129b | 3034398b | 0,734 | 1050ms | 121ms |
| 1.doc | 216144b | 303718b | 1,405 | 124ms | 17ms |
| 2.doc | 372732b | 528100b | 1,416 | 220ms | 28ms |
| 3.doc | 1402567b | 1993783b | 1,421 | 802ms | 95ms |
| 1.bmp | 818058b | 1126186b | 1,376 | 419ms | 56ms |
| 2.bmp | 3275658b | 4354731b | 1,329 | 1422ms | 182ms |
| 3.bmp | 7372938b | 9694066b | 1,314 | 3293ms | 390ms |
| 1.jpg | 474441b | 665728b | 1,403 | 267ms | 36ms |
| 2.jpg | 744692b | 1037663b | 1,393 | 366ms | 47ms |
| 3.jpg | 1372378b | 1905057b | 1,388 | 688ms | 82ms |

Алгоритм LZ77 показав свою ефективність лише для файлів у форматі txt, в усіх інших випадках він не є ефективним. Коефіцієнт вище 1 означає, що файл після компресії став займати більше місця ніж до компресії. Також слід зазначити, що алгоритм LZ77 потребує більше часу на свою роботу у порівняння із алгоритмом Хаффмана, але це актуально тільки у випадках компресії звичайного тексту у форматі txt. В інших випадках, де алгоритм LZ77 показав свою неефективність він працював швидше. Також слід зазначити, що алгоритм LZ77 має дуже високу швидку декомпресії у порівнянні із алгоритмом Хаффмана. У всіх випадках він є на порядок швидше.

Таблиця 7 – Вимірювання швидкості та коефіцієнту стискання алгоритму RLE

| Назва файлу | Початковий розмір | Розмір після компресії | Коефіцієнт компресії | Час компресії | Час декомпресії |
|---|---|---|---|---|---|
| 1.txt | 663442b | 1092814b | 1,647 | 19ms | 32ms |
| 2.txt | 1105612b | 1809457b | 1,636 | 44ms | 52ms |
| 3.txt | 4133129b | 6807283b | 1,647 | 138ms | 217ms |
| 1.doc | 216144b | 500263b | 2,314 | 12ms | 12ms |
| 2.doc | 372732b | 865753b | 2,322 | 23ms | 24ms |
| 3.doc | 1402567b | 3278302b | 2,337 | 65ms | 92ms |
| 1.bmp | 818058b | 1890505b | 2,31 | 37ms | 52ms |
| 2.bmp | 3275658b | 7567690b | 2,31 | 159ms | 261ms |
| 3.bmp | 7372938b | 16966333b | 2,3 | 314ms | 585ms |
| 1.jpg | 474441b | 1124113b | 2,369 | 22ms | 37ms |
| 2.jpg | 744692b | 1755625b | 2,357 | 42ms | 46ms |
| 3.jpg | 1372378b | 3218170b | 2,344 | 64ms | 91ms |

Алгоритм RLE не показав жодного результату із коефіцієнтом менше 1. У всіх випадках файл після компресії ставав більше, майже в усіх випадках коефіцієнт перевищував 2, тобто файл став більше у 2 рази. Це сталося через надзвичайну простоту цього алгоритму, адже усі файлі містять у собі доволі широкий набор символів, які завжди чергуються, що не дає змогу алгоритму бути ефективним та зменшувати розмір файлів. Але слід зазначити, що алгоритм має дуже високу швидкість компресії та декомпресії даних.

Таблиця 8 – Вимірювання швидкості та коефіцієнту стискання алгоритму арифметичного кодування

| Назва файлу | Початковий розмір | Розмір після стискання | Коефіцієнт компресії | Час компресії |
|---|---|---|---|---|
| 1.txt | 663442b | 1047574b | 1,579 | 600min |

Алгоритм арифметичного кодування показав свою повну не ефективність на великих обсягах даних. Вимірювання було здійснено лише на одному файлі. Результат у 600 хвилин не дає можливості протестувати усі інші файли із тестового набору, адже це займе купу часу та ці результати не дадуть жодних результатів. Цей алгоритм працює ефективно лише на дуже малих обсягах даних, але для великих обсягів отримане дробове число стає настільки великим, що надалі уповільнює кодування наступних символів.

Таблиця 8 – Вимірювання швидкості та коефіцієнту стискання адаптивного алгоритму

| Назва файлу | Початковий розмір | Розмір після компресії | Коефіцієнт компресії | Час компресії | Час декомпресії |
|---|---|---|---|---|---|
| 1.txt | 663442b | 431532b | 0,65 | 206ms | 130ms |
| 2.txt | 1105612b | 724185b | 0,655 | 349ms | 189ms |
| 3.txt | 4133129b | 2734833b | 0,66 | 1187ms | 607ms |
| 1.doc | 216144b | 399907b | 1,85 | 239ms | 151ms |

| | | | | | |
|---|---|---|---|---|---|
| 2.doc | 372732b | 684951b | 1,837 | 382ms | 207ms |
| 3.doc | 1402567b | 2578841b | 1,836 | 1332ms | 840ms |
| 1.bmp | 818058b | 1388699b | 1,697 | 754ms | 409ms |
| 2.bmp | 3275658b | 5597928b | 1,7 | 2823ms | 1445ms |
| 3.bmp | 7372938b | 12851990b | 1,743 | 6983ms | 4404ms |
| 1.jpg | 474441b | 848591b | 1,788 | 457ms | 248ms |
| 2.jpg | 744692b | 1323973b | 1,777 | 696ms | 439ms |
| 3.jpg | 1372378b | 2420543b | 1,763 | 1231ms | 667ms |

Адаптивний алгоритм має схожі результати з алгоритмом LZ77. Тобто він ефективен лише для текстових файлів у форматі txt. В інших випадках коефіцієнт компресії більше 1, що означає збільшення розміру після компресії даним алгоритмом.

# ВИСНОВКИ

У ході виконання кваліфікаційної роботи було виконано аналіз предметної області та досліджена швидкодія методів компресії даних в залежності від різних вхідних даних за допомогою платформи .NET.

Для цього було:

– проаналізовано та обрано алгоритми для компресії даних;

– визначено функціональні вимоги та розроблено консольний застосунок, який було використано в цілях проведення практичної частини експерименту;

– виміряно та підраховано значення обраних метрик для кожного з методів компресії даних за допомогою розробленого застосунку;

– порівняно та проаналізовано отриманні дані;

– зроблено висновки відносно описаних результатів дослідження.

Під час дослідження було проведено ряд тестів та вимірювань для оцінки впливу кожного методу компресії даних на швидкодію. Отримані результати дозволили зробити висновки про те, який метод дає найбільшу швидкість обробки даних, а також про те, які чинники впливають на ефективність цих методів.

У результаті дослідження можна зробити висновок, що оптимальним методом компресії даних є алгоритм Хаффмана, з точки зору універсальності та коефіцієнту компресії. В той же час він має один суттєвий недолік, це швидкість декомпресії. Вона є дуже низька у порівняні із швидкістю компресії.

Також можна виділити алгоритм LZ77, який має трохи меншу швидкість роботи ніж алгоритм Хаффмана та значно високу швидкість декомпресії, але має певну кількість недоліків, які не дають йому змогу конкурувати із алгоритмом Хаффмана. Перший і основний недолік це ефективність на лише на текстових файлах, інші типи файлів показали коефіцієнт компресії більше 1. Другий недолік це невисокий коефіцієнт компресії для текстових файлів у порівнянні із алгоритмом Хаффмана.

Інші методи компресії даних неможливо застосовувати через їх високий коефіцієнт компресії або низку швидкодію. Для того, щоб покращіти результати цих алгоритмів потрібні додаткові дослідження. Для цього можна розглянути використання більш складних варінтів реалізації цих алгоритмів. Необхідно порівняти різні методи та їх комбінації, щоб отримати більш точне уявлення про швидкість та ефективність методів компресії. Також, важливо розглянути використання різних мов програмування та фреймворків для розробки. Це може допомогти у покращенні швидкодії та ефективності роботи алгоритмів.

За результатами дослідження опубліковано тези «Data Compression Algorithms» на 26-тій Міжнародній науковій та технічній конференції «Theoretical And Practical Aspects Of Modern Research» (див. додаток Д).

# СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Software agents for learning resources of digital library, DA Milashenko, Sergiy D Makovetskiy, Natalya S Lesna, Gesellschaft für Informatik eV. – 2003.

2. Hennadii Falatiuk. Investigation of Architecture and Technology Stack for e-Archive System / Hennadii Falatiuk, Mariya Shirokopetleva, Zoia Dudar, IEEE – 2019, DOI: 10.1109/PICST47496.2019.9061407

3. Pu I. M. Fundamental data compression. Elsevier Science & Technology Books, 2005.

4. Sayood K. Introduction to data compression. Elsevier Science & Technology Books, 2005.

5. Benton D. J. Compression & Encryption: Algorithms & Software. Independently Published, 2019.

6. IJIRST - International Journal for Innovative Research in Science and Technology. A Research Paper on Lossless Data Compression Techniques. *Academia.edu - Share research*. URL: https://www.academia.edu/35454069/A_Research_Paper_on_Lossless_Data_Compression_Techniques (дата звернення: 11.06.2024).

7. Gersho A. Vector Quantization and Signal Compression. Boston, MA : Springer US, 1992. 732 c.

8. Wayner P. Compression Algorithms for Real Programmers. Elsevier Science & Technology Books, 1999. 240 c.

9. Hemmendinger D. Data compression | Lossless & Lossy Algorithms. *Encyclopedia Britannica*. URL: https://www.britannica.com/technology/data-compression(дата звернення: 14.01.2024).

10. How Lossless Data Compression Works | Quanta Magazine. *Quanta Magazine*. URL: https://www.quantamagazine.org/how-lossless-data-compression-works-20230531/(дата звернення: 14.01.2024).

11. Huffman coding | greedy algo-3 - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/(дата звернення: 14.01.2024).

12. Home - BenchmarkDotNet Documentation. *Site not found · GitHub Pages*. URL: https://fransbouma.github.io/BenchmarkDotNet/index.htm(дата звернення: 14.01.2024).

13. Рихтер, Дж. Програмування на платформі .NET Framework / Дж. Рихтер. – М., 2003.