НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

"На правах рукопису"
УДК_____

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2023 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою
“Цифрові технології в енергетиці”
зі спеціальності 122 “Комп’ютерні

на тему Методи зменшення складності алгоритму для покращення пошуку в Big Data на основі модифікації алгоритму Карпа-Рабіна

Виконав (-ла): студент (-ка) _2__курсу, групи ТР-23мп
Сірик Олександр Олександрович _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник доц., к.т.н., Кузьменко Ігор Миколайович _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____
(підпис)

Київ - 2023

Національний технічний університет України
“Київський політехнічний інститут ім. Ігоря Сікорського”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ
Рівень вищої освіти – другий (магістерський)
Освітньо-професійна програма “Цифрові технології в енергетиці”
Спеціальність 122 “Комп’ютерні науки”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Наталія АУШЕВА
(підпис)
«____» _____ 2023р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ СТУДЕНТУ

СІРИКУ Олександру Олександровичу

(прізвище, ім’я, по батькові)

1. Тема дисертації Методи зменшення складності алгоритму для покращення пошуку в Big Data на основі модифікації алгоритму Карпа-Рабіна
науковий керівник дисертації
Кузьменко Ігор Миколайович, к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “06” листопада 2023 року №5152-с

2. Строк подання студентом дисертації 18 грудня 2023р

3. Вихідні дані до роботи: операційна система macOS, мова програмування Rust, система збірки проектів Cargo, середовище розробки Clion, Rust Rover, інструмент пошуку ripgrep

4. Перелік питань, які потрібно розробити: проаналізувати підходи та методи пошуку по невпорядкованому тексту, описати використані методи зменшення складності, розробити програмне забезпечення, провести тестування розробленої програми та скомпілювати бінарний застосунок

5. Орієнтований перелік ілюстративного матеріалу: схеми архітектури системи, діаграми класів, зразки інтерфейсу командного рядку, графіки порівняння продуктивності

6. Орієнтований перелік публікацій: Вісник Кременчуцького національного

університету імені Михайла Остроградського. – Кременчук: КрНУ, 2023. – Випуск 4

7. Консультанти розділів дисертації

8. Дата видачі завдання «24» жовтня *2023 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	01.08.2023	
2	Вивчення та аналіз задачі	01.08.2023–14.08.2023	
3	Збір інформації	15.08.2023–31.08.2023	
4	Аналіз вимог завдання, розробка методів і засобів розв’язання поставленої задачі	01.09.2023–30.09.2023	
5	Розробка та тестування програмного продукту	01.10.2023–14.10.2023	
6	Підготовка матеріалів магістерської роботи	15.10.2023–20.11.2023	
7	Захист програмного продукту	24.10.2023	
8	Передзахист	05.12.2023	
9	Захист		

Студент _____ Сірик О. О.

(підпис) (прізвище та ініціали)

Науковий керівник _____ Кузьменко І. М.

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Актуальність теми. В епоху інформаційних технологій та інтернету критично збільшуються обсяги обігу даних, особливо невпорядкованої текстової інформації, яку потрібно обробляти. Для того щоб відповідати викликам сучасності, необхідно оптимізувати алгоритми пошуку по таким даним для збільшення швидкодії та практичності цих алгоритмів.

Метою роботи є підвищення ефективності алгоритму Карпа-Рабіна, як одного з найбільш універсальних та загальновживаних алгоритмів у задачі пошуку підрядку в тексті.

Завдання дослідження:

— провести аналіз відомих підходів для оптимізації наївного сценарію пошуку в тексті;

— описати методи оптимізації алгоритму Карпа-Рабіна, які необхідно розробити, на основі проведеного аналізу;

— розробити програмне забезпечення яке реалізує описані методи та дозволяє здійснювати пошук по великим обсягам даних;

— провести тестування розробленої програми;

Об’єкт дослідження — пошук по невпорядкованих текстових даних великого обсягу.

Предмет дослідження — методи оптимізації алгоритму Карпа-Рабіна на основі паралельних обчислень та алгоритмів викрадення роботи.

Практична цінність полягає в досліджені та розробці програмного застосунку який здійснюватиме пошук по великим обсягам даних за допомогою алгоритму Карпа-Рабіна та використаних оптимізацій.

Ключові слова: алгоритми пошуку рядка, алгоритм Карпа-Рабіна, паралельні обчислення, викрадення роботи, Big Data.

ABSTRACT

Actuality of theme. In the era of information technology and the internet, the volumes of data circulation, especially unstructured textual information that needs to be processed, are critically increasing. To meet the challenges of modernity, it is necessary to optimize search algorithms for such data to increase their speed and practicality.

The aim of the work is to improve the efficiency of the Karp-Rabin algorithm, as one of the most universal and widely used algorithms in the substring search task in the text.

Objectives of the study:

— Conduct an analysis of known approaches for optimizing the naive scenario of searching in text;

— Describe the methods of optimizing the Karp-Rabin algorithm, which need to be developed, based on the conducted analysis;

— Develop software that implements the described methods and allows searching through large volumes of data;

— Conduct testing of the developed software;

The object of research searching by unstructured text-based data of large sizes.

The subject of research are methods of optimizing the Karp-Rabin algorithm based on parallel computing and work-stealing algorithms.

The practical value lies in the research and development of a software application that will perform searches through large volumes of data using the Karp-Rabin algorithm and the optimizations used.

Keywords: string search algorithms, Karp-Rabin algorithm, parallel computing, work stealing, Big Data.

ЗМІСТ

ВСТУП

Пошук рядків у текстових даних завжди був однією з найбільш фундаментальних проблем у сфері комп'ютерних наук. Ще з ранніх часів розвитку галузі, коли були розроблені перші алгоритми для обробки тексту, такі як алгоритми Кнута-Морріса-Пратта, Боєра-Мура та інші, ця область постійно розвивалася, намагаючись знайти найефективніші способи вирішення проблеми.

Алгоритм Карпа-Рабіна, запропонований у 1987 році, відіграє важливу роль у розвитку алгоритмів пошуку рядків у тексті. В ньому використовується хешування для швидкого виявлення потенційних співпадінь, що дозволяє йому з низькою асимптотичною складністю O(n + m), ефективно обробляти текст, навіть з наявністю "шуму" чи неточностей. Цей алгоритм має свої переваги, такі як спроможність ефективно обробляти великі об'єми текстових даних, але також існують і певні обмеження, зокрема, у плані часової складності у випадках, коли потрібно здійснити пошук великої кількості рядків у величезних текстових файлах.

У сучасному світі концепція Big Data стає все більш актуальною через колосальний об'єм даних, який генерується щодня. Пошук інформації в обширних наборах даних стає складнішим і більш затратним по часу. Тому оптимізація часової складності алгоритмів пошуку набуває величезного значення для підвищення продуктивності та ефективності обробки даних.

Концепція кільцевого хешування, яка є наріжним каменем у алгоритмі Карпа-Рабіна показує високу універсальність на найбільш різноманітних наборах текстової інформації. Головною перевагою підходу з хешуванням – є суттєве зменшення кількості ітерацій витрачених на перевірку співпадінь. Замість покрокової перевірки символів, здійснюється перевірка хеш значень шуканого рядку, таким чином нівелюється залежність на розмір шуканого підрядку. Висока універсальність такого підходу робить цей алгоритм одним з найкращих для пошуку по неструктурованим даним, які містять багато шуму та неточностей.

Завдяки кільцевому хешуванню, алгоритм Карпа-Рабіна показує найменшу математично можливу асимптотичну складність для пошуку по невпорядкованим даним (за умови використання хеш-функції яка не породжує колізій) але алгоритм не оптимізований для паралельного використання ресурсів процесору, що робить можливим проведення дослідження у цьому напрямку. Ідеальним сценарієм було б розбити виконання цього алгоритму на N паралельних частин, де N – це кількість логічних процесорів системи, що зменшило б час виконання цього алгоритму в N разів.

Результатом цієї роботи є створення застосунку, який буде здійснювати пошук по великих за обсягом даних, по файлах, розмір яких може перевищувати обсяг оперативної пам’яті комп’ютеру, при використанні буферизації та виконання пошуку за методами паралельного програмування. Це може значно покращити час виконання алгоритму, особливо у контексті Big Data, дозволяючи обробляти великі текстові файли, з меншою затратою часу.

Робота складається зі вступу, 5 розділів, висновків до розділів, загальних висновків, списку використаних джерел (41 джерел, у тому числі 41 – іноземною мовою). Загальний обсяг дисертації – 85 сторінок. Роботу проілюстровано 22 таблицями, 13 рисунками.

1 ЗАДАЧА ОПТИМІЗАЦІЇ ПОШУКУ В ТЕКСТІ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Задача пошуку в тексті виникла ще з появою перших комп’ютерів. З того часу було винайдено чимало рішень до неї, які мають як свої переваги, так і недоліки. Для кращого розуміння проблематики, та більш ефективного вибору методів зменшення складності алгоритму Карпа-Рабіна необхідно проаналізувати методи, які використовуються в інших алгоритмах.

1.1 Загальний огляд підходів до пошуку в тексті

Для того, щоб здійснювати пошук по текстовим даним, необхідно розуміти, як саме текст зберігається та обробляється на комп'ютері. Основою цього процесу є кодування символів. Кодування - це спосіб, за допомогою якого символи тексту (наприклад, букви, цифри, знаки пунктуації) перетворюються в дані, які може зберігати та обробляти комп'ютер. Наприклад, ASCII [1] - це популярна система кодування, яка використовується для представлення латинських символів. Вона включає в себе цифри, літери англійського алфавіту та деякі спеціальні символи. Для інших мов та символів використовуються більш складні системи кодування, наприклад, UTF-8 [2], який може представляти широкий спектр символів різних мов світу.

Коли текстовий файл відкривається на комп'ютері, він спочатку декодується з використанням відповідного кодування. Це означає, що байти даних (які є основною одиницею зберігання в комп'ютері) перетворюються назад у символи, які ми можемо читати. Цей процес є критично важливим, оскільки правильне декодування забезпечує, що текст відображається коректно і без спотворень.

Після того, як текст декодовано, можна починати процес пошуку підрядку. Це включає в себе аналіз тексту для знаходження відповідності запиту. Процес

пошуку полягає в перевірці, чи містить текст послідовність символів, яка збігається з шуканим підрядком. Це може бути простою перевіркою кожного символу в тексті на відповідність підрядку, але існують також більш складні методи, які оптимізують цей процес, роблячи його швидшим і ефективнішим, особливо для великих об'ємів тексту.

У загальному випадку важливо враховувати, що кодування шуканого рядку має збігатися з кодуванням збереженого тексту адже, врешті-решт, під час порівняння символів, порівнюються саме байти, і якщо кодування відрізняється, неможливо буде отримати коректний результат пошуку.

Важливо розуміти, що ефективність пошуку залежить від багатьох факторів, включаючи довжину тексту, характер підрядку, який шукається, та специфіку кодування символів. Ця задача є ключовою в областях, які вимагають швидкого доступу та обробки текстової інформації, наприклад, в пошукових системах, редакторах тексту та системах управління базами даних.

1.2 Огляд відомих алгоритмів пошуку в тексті

Для вирішення задачі пошуку в тексті, на цей момент, було розроблено чимало алгоритмів. Вони призначені для вирішення різних задач, від простого лінійного пошуку до більш складних сценаріїв, що включають великі об'єми даних або складні мовні структури.

Одні алгоритми пошуку спрямовані на оптимізацію пошуку по невпорядкованим даним, інші ж пропонують спеціальні структури даних для впорядкування тексту [3].

Основною метою цих алгоритмів є забезпечення швидкості та точності при знаходженні вказаних підрядків. Деякі з них мають оптимізації, які спрямовані на пошук по великим обсягам даних, зокрема, вони використовують спеціальні структури даних чи алгоритмічні прийоми для зменшення кількості операцій порівняння [4]. Це важливо, оскільки великі тексти можуть містити мільйони

символів, і простий лінійний пошук, який порівнює кожен символ із шуканим підрядком, стає надзвичайно неефективним.

1.2.1 Наївний алгоритм пошуку рядку

Наївна імплементація алгоритму пошуку підрядку в тексті – це один з найпростіших методів для знаходження одного рядка в іншому. Цей алгоритм не вимагає складної підготовки або передобчислення і працює досить прямолінійно.

Основна ідея полягає в перевірці кожної можливої позиції у тексті, щоб побачити, чи починається підрядок саме з цієї позиції. Алгоритм бере підрядок, який треба знайти, і порівнює його з кожною послідовністю символів у тексті, що має таку ж довжину, як і шуканий підрядок.

Спочатку алгоритм розглядає перші символи тексту та підрядку і порівнює їх. Якщо символи співпадають, алгоритм продовжує порівнювати наступні символи підрядку з відповідними символами у тексті. Це продовжується до тих пір, поки або не буде знайдено відповідність (всі символи підрядку співпадають з послідовністю символів у тексті), або не буде виявлено неспівпадіння. У випадку неспівпадіння алгоритм пересувається на один символ вправо у тексті і повторює процес.

Цей підхід означає, що кожен символ тексту може бути перевірений багато разів, особливо якщо шуканий підрядок має символи, які часто зустрічаються в тексті. Це може призвести до значної кількості порівнянь, особливо для довгих текстів та підрядків.

Щодо ефективності, наївний алгоритм не є найефективнішим, особливо для довгих текстів або коли підрядок часто зустрічається в тексті. Його час виконання може бути дуже тривалим у найгіршому випадку, особливо якщо підрядок майже, але не повністю, збігається з багатьма частинами тексту.

Отже, наївна імплементація алгоритму пошуку підрядку є дуже простою у розумінні та імплементації, але вона не є оптимальною з точки зору ефективності.

Вона добре підходить для невеликих текстів або коли висока швидкість обробки не є критичною.

1.2.2 Алгоритм Кнута-Морріса-Пратта

Алгоритм Кнута-Морріса-Пратта (КМП) [5] - це ефективний метод пошуку підрядка, розроблений Дональдом Кнутом, Воном Праттом та Джеймсом Моррісом. Цей алгоритм розв'язує основну проблему наївного алгоритму пошуку шляхом уникнення повторного перегляду символів у тексті, які вже були порівняні.

Основна ідея КМП полягає в тому, щоб обчислити так звану таблицю префіксів для підрядку, який потрібно знайти. Ця таблиця містить інформацію про те, наскільки далеко алгоритм повинен "зсунутися" після неспівпадіння символів. Завдяки цій таблиці алгоритм може уникати зайвих порівнянь.

Таблиця префіксів будується на основі аналізу самого підрядку і визначає, як далеко можна "зсунути" підрядок у випадку неспівпадіння символу. Наприклад, якщо підрядок має повторюваний шаблон на початку і в середині, таблиця префіксів дозволяє "перестрибувати" через вже перевірені частини підрядку, замість того щоб починати порівняння заново з кожним новим символом у тексті.

Під час виконання пошуку, КМП спершу порівнює символи з початку тексту і підрядку. Якщо символи співпадають, він продовжує рухатися далі. У разі неспівпадіння, замість повернення на один символ в тексті, алгоритм використовує таблицю префіксів, щоб визначити, наскільки далеко можна зсунути підрядок вперед. Це означає, що алгоритм може пропускати декілька символів у тексті, значно зменшуючи загальну кількість порівнянь.

КМП є значно ефективнішим за наївний алгоритм у випадках, коли підрядок містить повторювані шаблони, оскільки це дозволяє уникнути багатьох непотрібних порівнянь. Його часова складність в найгіршому випадку є лінійною відносно суми довжин тексту та підрядку, що робить його значно швидшим для довгих текстів або підрядків.

Загалом, алгоритм Кнута-Морріса-Пратта є високоефективним і широко використовуваним рішенням для задач пошуку підрядків, особливо у ситуаціях, де потрібна висока продуктивність обробки великих обсягів тексту.

1.2.3 Алгоритм Боєра-Мура

Алгоритм Боєра-Мура [6] є ще одним важливим методом для ефективного пошуку підрядків у тексті. Він був розроблений Бобом Боєром і Дж. Стротером Муром та вважається одним з найшвидших алгоритмів для пошуку підрядків, особливо коли розмір алфавіту (набору символів, з якого складається текст) великий.

Головна ідея алгоритму полягає в тому, щоб починати пошук не з першого символу підрядку, а з останнього. Це дає можливість значно зменшити кількість порівнянь у багатьох випадках, оскільки при виявленні неспівпадіння можна "перестрибувати" через більшу частину тексту.

Алгоритм Боєра-Мура використовує дві ключові ідеї: "правило поганого символу" (bad character rule) та "правило гарного суфіксу" (good suffix rule).

1. Правило поганого символу: Коли відбувається неспівпадіння між символом у тексті і відповідним символом у підрядку, алгоритм шукає останнє входження цього "поганого" символу в підрядку. Якщо символ відсутній у підрядку, підрядок можна перемістити відразу за позицію цього символу у тексті. Якщо ж символ присутній, підрядок зсувається так, щоб цей символ у підрядку вирівнявся з його останнім входженням у тексті.

2. Правило гарного суфіксу: Коли відбувається неспівпадіння, але декілька останніх символів підрядку співпадають з текстом, алгоритм використовує цю інформацію для зсуву підрядку вперед. Він шукає інше місце в підрядку, де ця ж послідовність символів (суфікс) зустрічається, і зсуває підрядок так, щоб цей суфікс вирівнявся зі своїм останнім входженням.

Для ефективної реалізації цих правил алгоритм Боєра-Мура використовує дві допоміжні таблиці: таблицю поганих символів і таблицю гарних суфіксів. Таблиця поганих символів містить інформацію про те, наскільки далеко можна зсунути підрядок у випадку виявлення поганого символу. Таблиця гарних суфіксів розраховує зсуви на основі виявлених співпадінь суфіксів підрядку з частинами тексту.

Завдяки цим двом правилам алгоритм Боєра-Мура здатен значно знижувати кількість необхідних порівнянь, особливо у випадках, коли підрядок є довгим або коли символи підрядку рідко зустрічаються у тексті. Це робить його особливо ефективним для пошуку великих підрядків у великих текстах.

Алгоритм Боєра-Мура відомий своєю високою ефективністю в багатьох практичних застосуваннях, зокрема в області обробки текстів та пошуку в базах даних. Однак, його реалізація є складнішою в порівнянні з наївним алгоритмом або навіть КМП, оскільки вимагає попереднього обчислення та зберігання додаткової інформації у вигляді таблиць поганих символів та гарних суфіксів.

1.2.4 Алгоритм Карпа-Рабіна

Алгоритм Карпа-Рабіна [7] є ще одним важливим методом для пошуку підрядків у тексті, розроблений Річардом Карпом і Майклом Рабіном. Цей алгоритм відрізняється від інших методів пошуку підрядків своїм використанням хеш-функцій для швидкого порівняння послідовностей символів, що робить його особливо ефективним в деяких сценаріях.

Основна ідея алгоритму полягає у використанні хеш-функції для обчислення числового значення (хешу) для шуканого підрядку та для послідовностей символів у тексті, які мають таку саму довжину. Якщо хеш підрядку співпадає з хешем послідовності символів у тексті, може відбутися порівняння на більш детальному рівні для перевірки, чи дійсно ці послідовності ідентичні.

Алгоритм складається з декількох ключових кроків:

1. Хешування підрядку: Спочатку обчислюється хеш для підрядку, який потрібно знайти.

2. Хешування тексту: Наступний крок - обчислення хешів для всіх послідовностей символів у тексті, які мають таку ж довжину, як і шуканий підрядок.

3. Порівняння хешів: Алгоритм порівнює хеш підрядку з хешами послідовностей символів у тексті. Якщо хеші співпадають, алгоритм проводить детальне порівняння відповідних послідовностей символів, щоб переконатися, що вони дійсно ідентичні.

4. Рух по тексту: Якщо хеші не співпадають, алгоритм "зсуває" вікно порівняння вперед по тексту і повторює процес.

Однією з ключових особливостей алгоритму Карпа-Рабіна є його використання "кільцевого хешу" (rolling hash) [8]. Це означає, що коли алгоритм зсувається вперед по тексту, хеш для наступної послідовності символів може бути швидко обчислений на основі попереднього хешу, без необхідності обчислення хешу з нуля. Це значно збільшує швидкість роботи алгоритму.

Однією з найбільш ефективних хеш-функцій для використання в кільцевому хешуванні є поліноміальна хеш-функція [9]. Ця функція обчислює хеш рядка як поліном, коефіцієнти якого відповідають символам рядка, трансформованим у їхні числові значення. Поліноміальна хеш-функція має кілька переваг, особливо у використанні з кільцевим хешем. По-перше, вона дозволяє легко обчислювати хеш для "зсунутого" рядка: коли алгоритм зсувається на один символ вперед у тексті, можна швидко вирахувати новий хеш, віднявши вагу видаленого символу та додавши вагу нового символу, що входить у вікно порівняння. По-друге, ця функція має тенденцію зменшувати кількість хеш-колізій, хоча вони все одно можливі.

Обчислення поліноміальної хеш-функції можна описати наступними кроками:

1. Перетворення символів у числа: Кожен символ у рядку перетворюється в числове значення. Наприклад, можна використовувати ASCII-код символу.

2. Використання основи для полінома: Вибирається "основа" полінома, яка є певним фіксованим числом (часто просте число). Основа використовується для створення різних ступенів кожного символу в рядку.

3. Обчислення полінома: Для кожного символу у рядку обчислюється його "вага" як добуток його числового значення на основу, піднесену до ступеня, що відповідає позиції символу в рядку. Наприклад, для першого символу ступінь буде 0, для другого - 1, і так далі.

4. Сумування: Обчислені ваги для всіх символів рядка сумуються, утворюючи кінцеве хеш-значення. Це значення представляє собою поліноміальний хеш рядка.

5. Модульне арифметичне скорочення: Часто хеш-значення береться по модулю великого простого числа, щоб уникнути занадто великих чисел і обмежити розмір хешу.

Алгоритм Карпа-Рабіна ефективний у випадках, коли потрібно знайти багато різних підрядків у одному і тому ж тексті, оскільки після первинного обчислення хешів для тексту, пошук кожного наступного підрядку вимагає лише порівняння хешів. Це робить його особливо корисним у таких застосуваннях, як пошукові системи або програмне забезпечення для обробки текстів.

Однак, існує також можливість "колізій" хешів, коли різні послідовності символів мають однаковий хеш. Це може призвести до помилкових позитивних результатів, тому алгоритм повинен виконувати детальне порівняння послідовностей у випадку співпадіння хешів, щоб переконатися у точності збігу.

1.2.5 Алгоритм Ахо-Корасік

Алгоритм Ахо-Корасік [10], розроблений Альфредом Ахо та Маргарет Корасік, є високоефективним алгоритмом для пошуку множини підрядків у тексті. Він концептуально відрізняється від інших алгоритмів пошуку підрядків, таких як Кнута-Морріса-Пратта або Боєра-Мура, оскільки здатен ефективно обробляти

велику кількість підрядків одночасно, а не фокусуватися на пошуку одного підрядка в тексті.

Основна ідея алгоритму Ахо-Корасік полягає у побудові спеціалізованої структури даних, званої "автоматом станів" або "трієм" [11], яка представляє всі підрядки, які потрібно знайти. Цей автомат дозволяє алгоритму ефективно сканувати текст, переходячи між станами залежно від вхідних символів, і швидко визначати, коли будь-який з шуканих підрядків зустрічається в тексті.

Процес роботи алгоритму Ахо-Корасік можна описати таким чином:

1. Побудова автомата станів [12]: Спочатку алгоритм створює трій з усіх підрядків, які потрібно знайти. Кожен шлях від кореня до листка цього дерева відповідає одному з підрядків. Такий підхід дозволяє легко перевіряти присутність багатьох підрядків одночасно.

2. Створення додаткових переходів у трій: Для оптимізації швидкості пошуку алгоритм створює додаткові переходи між вузлами трій. Це дозволяє уникати повернення до кореня дерева при кожному неспівпадінні символу і замість цього переходити до іншого відповідного вузла.

3. Сканування тексту: Під час сканування тексту алгоритм переходить від одного стану трій до іншого, залежно від поточного символу тексту. При кожному переході алгоритм перевіряє, чи досягнутий стан відповідає завершенню одного з підрядків. Якщо так, це означає, що підрядок було знайдено у тексті.

4. Визначення співпадінь: Коли алгоритм досягає стану, що відповідає кінцю одного з підрядків, він ідентифікує співпадіння, дозволяючи виявити всі місця в тексті, де зустрічається цей підрядок.

Головна перевага алгоритму Ахо-Корасік полягає у його здатності обробляти велику кількість підрядків одночасно, що робить його ідеальним для таких задач, як пошук багатьох шаблонів у тексті або аналіз біологічних послідовностей. Він особливо ефективний у випадках, коли необхідно одночасно шукати велику кількість різних підрядків, оскільки весь процес сканування виконується лише один раз, незалежно від кількості шуканих підрядків. Недоліком цього алгоритму є те, складність побудови такого автомату станів є доволі високою, і якщо вхідні дані

заздалегідь не були збережені і структуровані у такому вигляді, то на первинну побудову цього автомату станів буде витрачено більше часу, ніж пошук по невпорядкованим даним, що робить цей алгоритм більш програшним ніж вищезгадані для пошуку по саме невпорядкованим даним.

1.2.6 Алгоритм Коменц-Вальтер

Алгоритм Коменц-Вальтер [13], названий на честь його розробників, є вдосконаленою версією алгоритму Боєра-Мура для пошуку підрядку в тексті. Особливістю алгоритму Коменц-Вальтер є його здатність одночасно використовувати декілька евристик для визначення оптимального зсуву підрядку під час пошуку, що дозволяє зменшити кількість необхідних порівнянь і підвищити швидкість пошуку.

На відміну від алгоритму Боєра-Мура, який використовує правило поганого символу та правило гарного суфіксу, алгоритм Коменц-Вальтер розширює ці концепції, додаючи додаткові евристики. Він використовує розширену обробку поганих символів та вдосконалену обробку добрих суфіксів. Це дозволяє алгоритму здійснювати більш ефективний зсув підрядку при скануванні тексту.

У процесі пошуку Коменц-Вальтер аналізує підрядок та текст, визначаючи, наскільки далеко можна перемістити підрядок вперед після кожного неспівпадіння. Це здійснюється за допомогою обчислення зсувів на основі позиції поганого символу в підрядку та оцінки можливих добрих суфіксів. Алгоритм дозволяє уникати перевірки кожного символу тексту, замість цього фокусуючись на найбільш перспективних позиціях для виявлення збігу.

Однією з ключових особливостей алгоритму є його здатність одночасно враховувати кілька можливих зсувів і вибирати оптимальний з них. Це робить алгоритм Коменц-Вальтер особливо ефективним для пошуку підрядків, які часто зустрічаються в тексті, або для текстів з високою частотою повторюваності символів.

Алгоритм Коменц-Вальтер ефективний у випадках, коли потрібно швидко знайти підрядок у великому тексті, оскільки його оптимізації дозволяють значно знизити кількість порівнянь порівняно з іншими алгоритмами, такими як наївний метод пошуку або навіть алгоритм Боєра-Мура. Його ефективність особливо виражена у випадках з великими обсягами тексту, де необхідно мінімізувати час обробки.

1.3 Порівняння використаних методів оптимізації у контексті алгоритму Карпа-Рабіна

Наведені алгоритми пропонують ряд оптимізацій, що спрямовані на загальне зменшення кількості ітерацій для порівняння підрядку. Було б цікаво дослідити, чи можливо застосувати якісь із цих оптимізацій до алгоритму Карпа-Рабіна.

Як уже було згадано, основою алгоритму Карпа-Рабіна є кільцева хеш-функція, яка не перераховується для кожного символу під час ітерації пошуку, а зсуває своє вікно тільки на один символ за раз, таким чином асимптотична складність перерахування хеш-функції є константною, тобто O(1). Цей алгоритм є так званим числовим алгоритмом, де для порівняння символів здійснюється певне числове перетворення, на відміну від інших наведених алгоритмів.

Розглянемо таблицю префіксів підрядку, запропоновану в алгоритмів Кнута-Морріса-Пратта. Ця оптимізація сильно знижує кількість перевірок, особливо для текстів у яких багато входжень, які майже однакові, але не дорівнюють шуканому підрядку. Це дозволяє “перестрибувати” на кількість елементів, яка однакова у тексті та підрядку. На відміну від КМП, алгоритм Карпа-Рабіна не має такої оптимізації, тобто покроково будуть порівнюватися усі символи, навіть якщо тільки останній символ відрізняється, алгоритм Карпа-Рабіна продовжить свою роботу з наступного символу в тексті. У силу особливості, що це є числовим алгоритмом, оптимізація КМП не має сенсу в загальному випадку алгоритму Карпа-Рабіна, адже за умови використання надійної хеш-функції, текст взагалі не

буде порівнюватися посимвольно, тобто для побудови таблиць префіксів потрібно буде виділяти додаткові ітерації, проте оскільки під час перевірки на входження не порівнюються символи, неможливо буде застосувати цю таблицю. Єдиним місцем де теоретично має сенс ця оптимізація в алгоритмі Карпа-Рабіна – це під час фінальної, посимвольної перевірки рядків. Ця перевірка здійснюється тільки у двох випадках:

1. шуканий підрядок повністю збігається із входженням у тексті;
2. хеш-функція створила колізію і підрядок не збігається із текстом.

У першому варіанті, немає сенсу застосовувати таблицю суфіксів адже це фінальна перевірка і ітерація закінчиться після повного співпадіння. У другому, було б корисно зсунути вікно на довжину префіксу що збігається, але в силу того, що алгоритм Карпа-Рабіна працює з хешем, потрібно буде перерахувати хеш для рядку з символу на який ми зсунули. Щоб перерахувати цей хеш, потрібно або посимвольно пройтися по всім символам що були пропущені та за властивістю кільцевого хешу зсунути це вікно, або повністю перерахувати хеш з нуля. Обидва варіанти є неоптимальними і повністю нівелюють перевагу зсуву на довжину префіксу що збігається адже врешті-решт, алгоритму необхідно буде повернутися до пропущених символів.

Алгоритм Боєра-Мура пропонує цікаву оптимізацію – починати перевірку не з початку, а з кінця підрядку. По аналогії з КМП це дозволяє зсувати індекс ітерації зразу на, так званий, “гарний суфікс”. По причинам описаним вище, оптимізації нечислових алгоритмів, які спрямовані на “перестрибування” певної кількості символів не мають сенсу в алгоритмі Карпа-Рабіна адже в будь-якому випадку необхідно перерахувати значення хешу.

На відміну від алгоритмів наведених вище, алгоритм Ахо-Корасік, пропонує спеціальну структуру даних для впорядкування тексту. Навіть якщо знехтувати тим що мова йде саме про невпорядкований текст, використання алгоритму Карпа-Рабіна поруч з Ахо-Корасік є дуже сумнівною адже структура даних trie дозволяє знаходити рядок за значно менший час ніж алгоритм Карпа-Рабіна. Під час такого пошуку хешування не потрібне адже якщо при покроковій перевірці символів було

виявлено, що хоча б один символ не збігається – можна завершувати ітерацію тому що шуканого підрядку в цій структурі гарантовано немає.

Отже, порівнявши наведені оптимізації, можна прийти до висновку, що їх використання поруч з алгоритмом Карпа-Рабіна не приведе до більшої ефективності пошуку, а в деяких випадках, навпаки суттєво зменшить ефективність.

1.4 Огляд підходів до паралелізації алгоритму Карпа-Рабіна

Із попереднього розділу можна зробити висновок, що нечислові оптимізації не мають сенсу в алгоритмі Карпа-Рабіна. Числові ж оптимізації можна застосувати тільки до хеш-функції, проте поліноміальна кільцева хеш-функція вже оптимізована під найважливіші критерії [14]: вона майже не породжує колізій, та перерахунок зсуву на символ має константний час. Як показує практика, використання хешу розміром навіть у 32 біти достатньо, щоб у текстах розміром декілька десятків гігабайтів не було жодної колізії.

Суттєве підвищення ефективності алгоритму Карпа-Рабіна дала б можливість запускати його паралельно в багатьох потоках.

Розглянемо можливі сценарії паралелізації. Умовно їх можна розділити на 2 напрямки: паралелізації обчислення кільцевого хешу та паралелізацію безпосередньо пошуку [15].

Як уже згадувалося раніше, кільцевий хеш має стан (state), тобто для того щоб порахувати нове значення необхідно використати попереднє. Через таке обмеження, неможливо одночасно паралельно рахувати значення хешу без використання попередніх значень. Хеш функція та сам хеш можуть бути пораховані тільки послідовно, один за одним. Оскільки асимптотична складність такого обрахунку O(1), його немає сенсу паралельно обчислювати.

Паралелізація пошуку звучить як більш реалістичний сценарій оптимізації адже в ідеальному варіанті можна знизити загальний час виконання пошуку в t

разів, де t – це кількість логічних процесорів системи. Складність такого підходу полягає в тому, що стає неочевидно як в такому випадку розділяти вхідні дані. Уявімо що ми маємо справу з масивом байтів, тому що в кінцевому варіанті алгоритм Карпа-Рабіна працює саме з байтами. Якщо довільно розбити масив на t однакових частин, можемо опинитися в ситуації, коли шуканий підрядок знаходиться на перетині цих частин. Потік 1 знайде тільки першу частину підрядку, потік 2 знайде останню, і в сумі, пошук дасть хибний результат, що підрядку в тексті не існує, хоча він там є, просто опинився на перетині частин. Для того щоб уникнути такої ситуації, потрібно не просто розбивати вхідні дані на t однакових частин, а й додавати кількість байтів, що дорівнює кількості байтам шуканого підрядку в кінець кожної частини. Це збільшить загальний об'єм тексту в середньому на $(\mathrm{t} - 1) * \mathrm{m}$, де m – це довжина шуканого підрядку, проте нівелює хибні спрацювання під час паралельного виконання. Також у загальному випадку, довжина m не є настільки значною, щоб помітно вплинути на виконання алгоритму. Додавши цю кількість байтів в кінець гарантує що всі входження будуть знайдені, проте, іноді ці входження можуть бути знайдені 2 рази, у випадку якщо співпадіння наступної частини трапляється з першого ж байту. Для того щоб уникнути цього достатньо зменшити додану кількість байтів до $\mathrm{m} - 1$. Це гарантує те що всі входження в текст будуть знайдені і виключить потенційні дублікати.

1.4.1 Розбиття вхідних даних на менші частини

Із попереднього розділу випливає те що паралелізація шляхом розбиття вхідних даних є перспективним методом зменшення загального часу виконання. Хоч це і збільшить загальний розмір роботи n на $n + (t - 1) * (m - 1)$, де n – це загальний розмір тексту, для величезних обсягів у Big Data це не грає суттєвої ролі, адже це залежність не на весь об'єм тексту, а тільки на розмір шуканого підрядку. Проте така оптимізація може суттєво зменшити час виконання.

Оскільки мова йде про величезні обсяги даних, не можна просто розбити їх на частини та завантажити до пам’яті адже у загальному випадку, комп’ютер фізично не матиме такого об’єму оперативної пам’яті. Для того щоб вирішити цю проблему, можна використати підхід з буферизацією. Тобто виділити достатній розмір пам’яті під буфер і під час пошуку читати дані в цей буфер, по якому вже можна буде здійснювати пошук алгоритмом Карпа-Рабіна. Такий підхід чудово спрацює з паралельним виконанням адже для кожного потоку вже виділені загальні межі роботи, а завантаження байтів до буферу здійснюватиметься послідовно, тому це виключає так званий стан гонки “race condition”, коли паралельні потоки будуть звертатися до одних і тих самих даних.

1.4.2 Оптимізація паралелізації за допомогою алгоритму викрадення роботи

Описаний підхід із розбиттям даних на менші частини може значно покращити час виконання. Проте він не є ідеальним, у ньому присутні деякі недоліки, які можна усунути так званим алгоритмом викрадення роботи, який був описаний та використаний у ForkJoinPool, стандартній бібліотеці мови Java [16].

Розділивши загальну роботу на t потоків, можна опинитися в ситуації, коли ці частини не будуть однаковими, наприклад якщо довжина тексту не кратна кількості потоків. Також, унаслідок особливостей роботи планувальника операційних систем, виконання деяких потоків може бути перерване під час виконання пошуку, що спричинить затримку одних потоків у порівнянні з іншими. Усе це спричиняє те, що навіть за однакових обсягів розділених частин, виконання пошуку буде розподілятися не рівномірно між всіма потоками. Одні будуть закінчувати свою роботу швидше ніж інші, і, як наслідок, будуть простоювати до моменту завершення всіх потоків. Для того щоб уникнути цієї ситуації, можна надати змогу потокам, що швидше завершили свою роботу “допомагати” іншим потокам, які ще працюють. Для цього стратегія розбиття загальної роботи повинна

відрізнятися він банального ділення на кількість потоків. Як одиницю роботи можна використати буфери, описані в попередньому розділі. Необхідно умовно розділити вхідні дані на певний розмір буферу, розподілити ці буфери між потоками, і завантажувати розмічений діапазон до цього буферу.

Архітектурно можна виділити такі елементи:

1. Worker – Це черга FIFO або LIFO, яка належить одному потоку, але інші потоки можуть викрадати з нього роботу. Планувальники задач зазвичай створюють одну робочу чергу на поток;

2. Stealer – Примітив, що має доступ до черги роботи і доступний для всіх потоків. Будь який потік може звернутися до цього примітиву за новою роботою.

Описаний підхід дозволяє рівномірно розподіляти роботу між усіма потоками, що гарантує відсутність простоїв для всіх потоків. Викрадення роботи не потребує обчислювальних затрат, при правильно підібраним структурам, наприклад двостороннім чергам, асимптотична складність такої операції буде дорівнювати O(1). Єдиним недоліком такого підходу є те, що він значно складніший у реалізації за звичайне розділення роботи. Він потребує використання примітивів синхронізації та використання атомічних операцій, щоб унеможливити стан гонки та взаємне блокування, яке може стати реальною проблемою, коли викрадення роботи буде заблоковане для декількох потоків, і подальша робота алгоритму буде неможливою.

1.4.3 Оптимізація читання за допомогою асинхронного вводу/виводу

При великих обсягах даних, читання байтів в оперативну пам’ять займатиме суттєву кількість часу. У цей час робота алгоритму пошуку буде неможливою і потік буде простоювати та чекати доки буфер наповниться даними.

Для вирішення цієї проблеми, сучасні операційні системи мають примітиви, що здатні реалізовувати асинхронне введення/виведення. У випадку з читанням

даних до буферів, операційна система може повідомити, коли таке читання завершиться і можна буде продовжувати виконання роботи програми.

Це реалізовано в ядрі операційних систем наступними системними викликами:

1. select/poll: Ранні Unix системи впровадили системні виклики select та poll. Ці виклики дозволяють програмі моніторити декілька файлових дескрипторів, чекаючи, поки один або декілька з них стануть "готовими" для певного класу введення-виведення (наприклад, можливе введення). Обмеження цих API полягає в тому, що вони масштабуються лінійно з кількістю файлових дескрипторів.

2. epoll (Linux) [17]: Щоб подолати проблеми масштабування select та poll, Linux впровадив epoll, масштабовану систему сповіщення про події введення-виведення. epoll можна використовувати для моніторингу декількох файлових дескрипторів, щоб дізнатися, чи можливе введення-виведення на будь-якому з них. Це ефективніше для великої кількості файлових дескрипторів, оскільки використовує механізм зворотнього виклику замість опитування.

3. AIO (POSIX Asynchronous I/O): POSIX-сумісні системи також пропонують асинхронне введення-виведення через API AIO. Ці API дозволяють програмам подавати та контролювати асинхронні операції введення-виведення за допомогою підтримки ядра.

4. IOCP (I/O Completion Ports): Windows надає порти завершення введення-виведення, потужний механізм для виконання асинхронних операцій введення-виведення. IOCP дозволяє ефективно виконувати асинхронне введення-виведення, відокремлюючи подання запиту на введення-виведення від його завершення.

Програми можуть реалізувати так зване асинхронне середовище виконання (Asynchronous Runtime), яке буде зважати на описані системні виклики операційної системи. Це середовище має власні потоки виконання, які не блокуються під час операцій введення/виведення. Прикладами таких середовищ є libuv, написане мовою C++, та Tokio, написане мовою Rust.

Для задачі пошуку алгоритм Карпа-Рабіна цей підхід означає те, що можна запустити більше потоків, які будуть виконуватися одночасно, тому що асинхронне

середовище виконання буде призупиняти потоки що здійснюють асинхронне читання байтів у буфер. Під час цього, інші потоки будуть виконувати їхню безпосередню задачу – пошук підрядку. Таким чином, замість часу витраченого потоком на простій та очікування, поки операційна система заповнить буфер оперативної пам’яті, логічні процесори будуть завжди зайнять роботою.

Асинхронне програмування може зменшити загальний час виконання пошуку, проте на практиці використання асинхронного середовища виконання також потребує ресурсів, що впливатиме на роботу пошуку. Тому на практиці, така оптимізація може не показати позитивних результатів.

Висновки до розділу 1

У першому розділі описано предметну область пошуку в тексті. Розглянуто проблеми зв’язані з кодуванням тексту, та їх вирішення. Було розглянуто та проаналізовано основні алгоритми пошуку в тексті, включаючи алгоритм Карпа-Рабіна.

У цьому розділі було надано оцінку можливим методам зменшення складності. Було проаналізовано відомі оптимізації та розглянуто їх у контексті алгоритму Карпа-Рабіна. Зазначено відмінність числових алгоритмів пошуку по тексту, та нечислових алгоритмів. Визначено методи оптимізації які доцільно використовувати для пошуку алгоритмом Карпа-Рабіна.

У силу особливостей алгоритму Карпа-Рабіна, визначено напрямок оптимізації – паралельного виконання пошуку, та наведено аргументи в користь цього вибору.

На основі проведеного аналізу було обрано 2 основні методи оптимізації: паралелізація пошуку та розбиття вхідних даних на менші частини, та оптимізація за допомогою алгоритму викрадення роботи, які теоретично принесуть найбільший приріст у ефективності виконання пошуку.

2 ТЕХНОЛОГІЇ ТА ЗАСОБИ РЕАЛІЗАЦІЇ МЕТОДІВ ЗМЕНШЕННЯ СКЛАДНОСТІ АЛГОРИТМУ КАРПА-РАБІНА

Вибір технологій для імплементації алгоритму Карпа-Рабіна, та методів зменшення складності продиктований специфікою поставленої задачі. Для того щоб більш-менш об’єктивно виконати порівняння методів, сам пошук, та їх ефективність необхідно використати низькорівневу мов програмування, яка має прямий контроль над оперативною пам’яттю, без збирача сміття адже під час виконання переівірок ефективності, збирач сміття може повністю зупинити роботу програми на деякий час і почати звільняти оперативну пам’ять.

Під такі вимоги підходить небагато мов, найпопулярніші – це C, C++, та Rust. Оскільки дана робота сфокусована на паралельному виконанні, мови C, та C++ буде досить складно використовувати адже вони не взагалі захищають розробника від помилок що можуть спричинити вразливості доступу до пам’яті, такі як помилки сегментації, тощо. Вони також не дають ніяких гарантій щодо цілісності пам’яті під час паралельного виконання. На відміну від них, мова Rust є більш сучасною, та без втрати продуктивності надає всі вищеописані гарантії.

2.1 Мова програмування Rust

Мова програмування Rust надає велику увагу безпеці та ефективності, і системи Ownership та Borrow Checker є фундаментальними елементами, які визначають її дизайн [18]. Спробуємо розглянути їх детальніше.

Система Ownership в Rust описує, хто є "власником" даних у пам'яті та як ці дані можуть бути доступними іншим частинам програми. В Rust кожне значення має єдиного власника, і коли власник виходить з області видимості, значення

автоматично очищується. Це допомагає уникнути витоків пам'яті та забезпечити більш детерміноване управління пам'яттю.

Borrow checker — це частина компілятора Rust, яка наглядає за тим, як програма "позичає" посилання на дані. Позичення може бути здійснене двома способами: у вигляді незмінних посилань (immutable borrows), що дозволяє множині об'єктів мати доступ до даних тільки для читання, або у вигляді змінного посилання (mutable borrow), яке дозволяє одному об'єкту мати ексклюзивний доступ до даних для читання та модифікації. Borrow checker гарантує, що два змінних посилання не можуть існувати одночасно і що не може бути одночасного змінного та незмінного посилання на дані.

Така система допомагає уникнути гонок даних, забезпечуючи безпечний доступ до пам'яті без блокування та ускладнення, характерних для традиційних багатопоточних систем. Комбінація ownership та borrow checking у Rust сприяє написанню коду, який є не тільки безпечним, але і ефективним, з обмеженим використанням ресурсів та високою продуктивністю. Оскільки компілятор Rust детерміновано визначає в якому місці коду повинне відбутися очищення пам’яті, ми уникаємо використання збирача сміття, який час від часу повністю зупиняв би виконання програми для видалення з пам’яті структур даних, які більше не використовуються.

Мова Rust має дуже невеликі витрати ресурсів на підтримання вищеописаних правил, що робить її співставною по продуктивності з такими мовами як C та C++, в яких основним девізом є те, що розробник не “платить” за те, що не використовує.

Завдяки надійній системі типів мови Rust дуже просто створювати надійний та безпечний код. Вона має статичну типізацію, що означає, що типи даних перевіряються на етапі компіляції, не дозволяючи випадкових перетворень типів або роботи з невизначеними значеннями. Однією з ключових особливостей системи типів Rust є Enums та Match вирази. Enums дозволяють декларувати тип, який може мати декілька варіантів, і це часто використовується для представлення різних можливих станів. Match вирази, у свою чергу, дозволяють зручно та безпечно робити вибір на основі значення Enums, враховуючи всі можливі варіанти

та допомагаючи уникнути неочікуваних ситуацій. Generics та Traits у Rust також є важливою частиною системи типів. Generics дозволяють створювати загальні типи та функції, зберігаючи при цьому статичну типізацію. Traits визначають спільну поведінку для різних типів, дозволяючи використовувати поліморфізм та код, який легко адаптується до різних ситуацій. Система типів Rust грає вирішальну роль у забезпеченні безпечного управління пам'яттю та доступу до даних у багатопоточних програмах. Ownership та Borrow Checker дозволяють Rust гарантувати, що доступ до даних є коректним та безпечним, пропонуючи прості правила для управління життєвим циклом даних та доступом до них у багатопоточному середовищі. Ці механізми забезпечують, що в один момент часу або один потік може мати змінний доступ до даних, або декілька потоків можуть мати доступ тільки для читання, уникнення ситуацій гонки та небезпечних модифікацій даних. Таким чином, система типів у Rust, у поєднанні з концепціями ownership та borrowing, створює потужні засоби для написання надійних, ефективних та безпечних багатопоточних програм.

Завдяки таким її унікальним особливостям ця мова ідеально підходить для задач, пов’язаних з пошуком тексту, оптимізацією такого пошуку, та задачами з використанням багатопоточності.

2.1.1 Компілятор rustc

Компілятор rustc [19] для мови програмування Rust є ключовим інструментом, який перетворює високорівневий код Rust на машинний код, пристосований для всіх сучасних архітектур процесорів та операційних систем. Використовуючи LLVM як свій бекенд, rustc перетворює Rust код спочатку у проміжне представлення LLVM (IR), яке потім оптимізується і компілюється у машинний код. Цей підхід дозволяє rustc ефективно підтримувати широкий спектр архітектур, включаючи, але не обмежуючись, x86, ARM та MIPS.

Особливість rustc полягає у його здатності забезпечувати строгу безпеку пам'яті завдяки системі володіння та запозичення в Rust, яка уникає помилок зв'язаних з паралельним доступом до даних та неправильним керуванням пам'яттю, без потреби у збирачі сміття. rustc аналізує код на етапі компіляції, забезпечуючи відсутність гонок даних та інших помилок, які можуть виникнути при неправильному використанні пам'яті.

Тісна інтеграція з Cargo, системою управління пакетами і збірки Rust, є ще однією ключовою особливістю rustc. Cargo використовує rustc для компіляції проектів, управління залежностями та збірки пакетів, що значно спрощує процес розробки на Rust.

Компілятор rustc також підтримує розширення функціоналу через плагіни, так звані build scripts, що дозволяє гнучко та універсально налаштовувати процес компіляції. Підтримка атрибутів і макросів у Rust дозволяє rustc обробляти складні конструкції на етапі компіляції, надаючи можливості для метапрограмування та гнучкої модифікації поведінки програми.

2.1.2 Система збірки проекту Cargo

Інструмент управління пакетами та система збірки Cargo [20], для Rust, представляє собою значне вдосконалення порівняно з традиційними підходами до збірки в мовах C і C++. У цих мовах відсутність єдиної, стандартної системи збірки призводить до складнощів: розробники мусять вибирати між численними інструментами, як-от Make, CMake, або Ninja, кожен з яких має свої особливості, способи налаштування та управління залежностями.

В контексті Rust, Cargo вирішує ці проблеми, надаючи уніфіковане рішення для збірки проектів, управління залежностями та автоматизації різних завдань, пов'язаних з розробкою. Замість того, щоб розробники вручну управляли бібліотеками та залежностями, Cargo автоматизує цей процес, дозволяючи легко вказувати та управляти залежностями через файл конфігурації Cargo.toml. Це

виключає необхідність писати складні скрипти збірки, як це часто відбувається у C/C++. Така уніфікація досягається за допомогою підходу до структурування проекту. Розробники мають розміщувати файли в конкретних, заздалегідь продуманих локаціях, для того щоб Cargo міг їх використати. Наприклад уже згаданий Cargo.toml повинен лежати в корені проекту, в кожному проекті має бути папка src, в якій Cargo буде шукати файл main.rs, або lib.rs і буде трактувати його як вхідну точку в проект. По аналогії поруч з папкою src, може знаходитися папка test, в якій можливо розміщувати інтеграційні тести, які будуть запущені Cargo як окремі застосунки. Звичайні ж модульні тести, завдяки можливостям та синтаксису мови Rust можливо писати поруч зі звичайним кодом, і Cargo розуміє яка частина відповідає за вихідний код, а яка за тести. Cargo інтегрує функції тестування безпосередньо в процес збірки, забезпечуючи зручність та ефективність виконання тестів. Ця інтеграція сприяє кращій організації тестування та допомагає розробникам підтримувати високу якість коду. Cargo також полегшує публікацію та спільне використання бібліотек через централізоване сховище crates.io, що сприяє спільноті Rust і спрощує повторне використання коду.

У підсумку, Cargo вносить значний вклад у ефективність і спрощення розробки на Rust, усуваючи багато проблем, з якими стикаються розробники у мовах C і C++, та надає більш структурований та уніфікований підхід до збірки програмного забезпечення.

2.1.3 Інструмент статичного аналізу Clippy

Інструмент статичного аналізу Clippy [21], відіграє роль помічника в написанні більш чистого, ефективнішого та ідіоматичного коду. Він працює шляхом аналізу коду на наявність відомих шаблонів, які можуть бути поліпшені, вказуючи на типові помилки або надаючи рекомендації щодо їх виправлення. Clippy забезпечує великий набір перевірок, які включають стиль кодування,

оптимізацію використання, покращення читабельності та уникнення зайвих конструкцій.

Основна мета Clippy полягає в підвищенні якості коду Rust, забезпечуючи його відповідність передовим практикам і конвенціям. Цей інструмент виявляє не тільки синтаксичні або логічні помилки, а й нюанси, пов'язані з ефективністю, використанням типів, специфічними особливостями Rust, такими як системи ownership та borrow checker і навіть потенційні логічні помилки в коді.

Clippy також забезпечує гнучкість у налаштуванні. Є можливість налаштовування рівеню суворості перевірок, включати або виключати конкретні перевірки. Це дозволяє використовувати Clippy ефективно в різних проектах з різними вимогами до якості коду та стилістики.

Інтеграція Clippy в процес розробки Rust є частиною рекомендованих практик. Його використання не тільки сприяє кращому розумінню ідіом Rust, але й забезпечує вищу ступінь впевненості у якості та надійності коду. Інструмент є невід'ємною частиною екосистеми Rust, сприяючи високому рівню кодування та сприянню кращим практикам програмування.

2.1.4 Інструмент форматування rustfmt

Інструмент для форматування коду rustfmt [22], забезпечує уніфікацію стилю коду. Цей інструмент автоматично вирівнює код згідно з загальноприйнятими стандартами стилю Rust, що сприяє легшому читанню коду, його зрозумілості та підтримці. Інструмент rustfmt аналізує існуючий код і структурує його, змінюючи розташування елементів коду, таких як блоки, вирази, коментарі, щоб вони відповідали встановленим нормам форматування.

Особливість rustfmt полягає в тому, що він не тільки сприяє кращому візуальному представленню коду, але й підтримує послідовність стилів коду. Це дуже важливо в умовах спільної роботи, де різні розробники можуть мати свої переваги в стилі написання коду. Інструмент може бути налаштований через файл

конфігурації, що дозволяє адаптувати правила форматування під конкретні потреби проекту або команди.

Крім того, rustfmt легко інтегрується з іншими інструментами та середовищами розробки (наприклад Clion), що робить його зручним у використанні як частини регулярного процесу розробки. Використання rustfmt є частиною рекомендованих практик розробки на Rust, оскільки воно сприяє підвищенню якості коду та його стандартизації.

2.1.5 Мовний сервер Rust Analyzer

Інструмент Rust Analyzer [23] використовується для аналізу коду Rust, який використовує Language Server Protocol (LSP) для надання розширених можливостей редакторам коду. LSP – це стандартний протокол, який забезпечує зв'язок між редактором коду та сервером мови, дозволяючи редактору запитувати інформацію про код, таку як автодоповнення, підказки, визначення, та отримувати відповіді від сервера. Цей протокол дозволяє розробникам інтегрувати розширені можливості аналізу коду в будь-які редактори, які підтримують LSP, забезпечуючи більш гнучке та єдине середовище розробки.

Rust Analyzer використовує LSP для надання таких можливостей, як автоматичне завершення коду, навігація по коду, виявлення помилок та підказки з рефакторингу. Це робить його інструментом, який може адаптуватися до різних редакторів та середовищ, забезпечуючи розробникам Rust високоякісні інструменти для аналізу коду незалежно від їх вибору редактора.

Раніше Rust мав інший інструмент для цих цілей - Rust Language Server (RLS). RLS також був реалізацією LSP для Rust, але з часом Rust Analyzer виявився більш ефективним у багатьох аспектах. Rust Analyzer був розроблений з метою вирішення деяких обмежень і проблем, які були в RLS, таких як швидкість роботи та точність аналізу коду.

Однією з ключових переваг Rust Analyzer є його швидкість і точність, особливо в великих проектах, де RLS міг проявляти повільність і недостатню відповідність. Rust Analyzer використовує більш сучасні техніки та алгоритми для аналізу коду, забезпечуючи кращу підтримку сучасних функцій мови Rust і кращий досвід користування.

Завдяки своїй високій ефективності, точності та гнучкості, Rust Analyzer швидко став популярним у спільноті Rust, замінивши RLS як основний інструмент для аналізу коду в більшості проектів. Його впровадження значно підвищило ефективність розробки на Rust, забезпечуючи розробникам потужний інструмент для роботи з кодом.

2.2 Середовище розробки Clion

Середовище CLion [24] від JetBrains є одним з передових інтегрованих середовищ розробки (IDE), яке первісно було розроблено для C і C++, але завдяки розширеному плагіну Rust, також пропонує чудову підтримку для розробки на Rust. Плагін Rust для CLion розширює функціональність IDE, включаючи підтримку синтаксису Rust, підсвічування коду, автозавершення, перехід до оголошення та інші функції, які забезпечують більш продуктивну роботу та зручний досвід розробки. CLion відзначається глибокою інтеграцією з технологіями LLDB та LLVM, які є ключовими для розробки на мові програмування Rust.

CLion фокусується на забезпеченні розробників потужними інструментами для роботи з кодом та організації робочого процесу. IDE автоматизує багато рутинних завдань, які виникають під час розробки, таких як рефакторинг коду, аналіз коду на наявність помилок і потенційних проблем. Це допомагає зосередитися на логіці програми та алгоритмах, замість того, щоб витрачати час на механічні аспекти кодування.

LLDB є стандартним дебагером для Rust, і в CLion існує глибока інтеграція з цим дебагером. LLDB у CLion дозволяє розробникам ефективно відлажувати код,

використовуючи зручний графічний інтерфейс для таких операцій, як встановлення точок зупинки, перегляд значень змінних, стеку викликів, керування потоками виконання тощо. Глибока інтеграція з LLDB в CLion означає, що розробники можуть використовувати всі переваги дебагера прямо в IDE, отримуючи повний контроль над процесом виконання програми та можливість швидкого виявлення та усунення помилок і проблем у коді.

З іншого боку, LLVM є фундаментальною технологією для компіляції та оптимізації коду на Rust. LLVM є набором модульних та повторно використовуваних компіляторів та інструментальних засобів, які використовуються у ряді мов програмування, включаючи Rust. Через те, що CLion має вбудовану підтримку LLVM, розробники отримують доступ до ряду інструментів для аналізу, діагностики та оптимізації коду. Наприклад, CLion може використовувати засоби LLVM для аналізу профілювання, виявлення вузьких місць у продуктивності коду, а також застосування оптимізацій на рівні компілятора для підвищення ефективності виконання програм.

Таким чином, інтеграція з LLDB та LLVM у CLion дозволяє розробникам на Rust ефективно відлажувати, аналізувати та оптимізувати свій код, користуючись всіма перевагами цих технологій прямо з середовища розробки.

2.3 Інструмент порівняльного аналізу Hyperfine

Hyperfine [25] – це потужний і простий у використанні командний інструмент для вимірювання продуктивності, особливо корисний для бенчмаркінгу алгоритмів пошуку. Він призначений для забезпечення точних і надійних вимірювань часу виконання команд у командному рядку, що робить його ідеальним для оцінювання швидкості і ефективності різних скриптів або програм.

Основні особливості та можливості hyperfine включають:

1. Простота використання: Hyperfine має простий і інтуїтивний синтаксис. Необхідно лише ввести команду, яку потрібно тестувати, і hyperfine автоматично

здійснить вимірювання, виконуючи команду кілька разів для забезпечення точності результатів.

2. Точність вимірювань: Hyperfine автоматично визначає кількість необхідних повторів команди для забезпечення статистично значимих результатів. Це дозволяє уникнути випадкових варіацій у часі виконання, які можуть виникати через різні зовнішні фактори.

3. Порівняння продуктивності: За допомогою hyperfine можена легко порівняти продуктивність різних команд або алгоритмів. Це дозволяє ефективно визначити, який з підходів є швидшим або більш оптимізованим для конкретного випадку використання. Результати порівняння продемонстровані на рисунку 2.1.

4. Виведення результатів: Результати вимірювань можуть бути виведені у різних форматах, включаючи текст, Markdown або JSON. Це забезпечує гнучкість при інтеграції з іншими інструментами або для подальшого аналізу даних.

5. Параметризація команд: Hyperfine дозволяє параметризувати команди, що дозволяє автоматично тестувати різні варіанти однієї і тієї ж команди з різними вхідними даними або параметрами.

```
~/dev/rabinkarp/fork-ripgrep
→ hyperfine -i "rg Eligibility unknown as battery range tmp/dataset.txt"
Benchmark 1: rg Eligibility unknown as battery range tmp/dataset.txt
  Time (mean ± σ):      17.0 ms ±   0.9 ms    [User: 9.8 ms, System: 6.1 ms]
  Range (min … max):    15.3 ms …  19.2 ms    140 runs

  Warning: Ignoring non-zero exit code.

~/dev/rabinkarp/fork-ripgrep
→ hyperfine -i "./target/release/rg --fabian-matcher=rabin-karp --parallel-searcher=work
-stealing fabian-needle tmp/dataset.txt"
Benchmark 1: ./target/release/rg --fabian-matcher=rabin-karp --parallel-searcher=work-ste
aling fabian-needle tmp/dataset.txt
  Time (mean ± σ):      19.4 ms ±   0.3 ms    [User: 41.3 ms, System: 9.6 ms]
  Range (min … max):    18.9 ms …  20.6 ms    140 runs

  Warning: Ignoring non-zero exit code.
```

Рисунок 2.1 – Результат роботи Hyperfine

Завдяки цим особливостям, hyperfine є відмінним інструментом для оцінки та порівння продуктивність алгоритмів, скриптів або програм. Наприклад, при роботі з алгоритмами пошуку підрядків, hyperfine може бути використаний для точного вимірювання часу виконання кожного алгоритму на різних вхідних даних, дозволяючи тим самим ефективно.

2.4 Інструмент аналізу продуктивності Flamegraph

Інструмент для візуалізації продуктивності програм Flamegraph [26] дозволяє розробникам мови Rust (і не тільки) ефективно аналізувати і виявляти вузькі місця в програмному коді. Це особливо корисно при оптимізації продуктивності складних додатків, де зрозуміти, яка частина коду викликає затримки або перевантаження, може бути складно.

Flamegraph працює шляхом збору даних про виконання програми, зазвичай через профілювання або трасування. Ці дані містять інформацію про виклики функцій, час їх виконання та ієрархії викликів. З цієї інформації Flamegraph генерує "полум'яний графік", зображений на рисунку 3.1, - візуальне представлення стеку викликів [27], де кожна функція представлена блоком, розмір якого пропорційний часу виконання цієї функції.

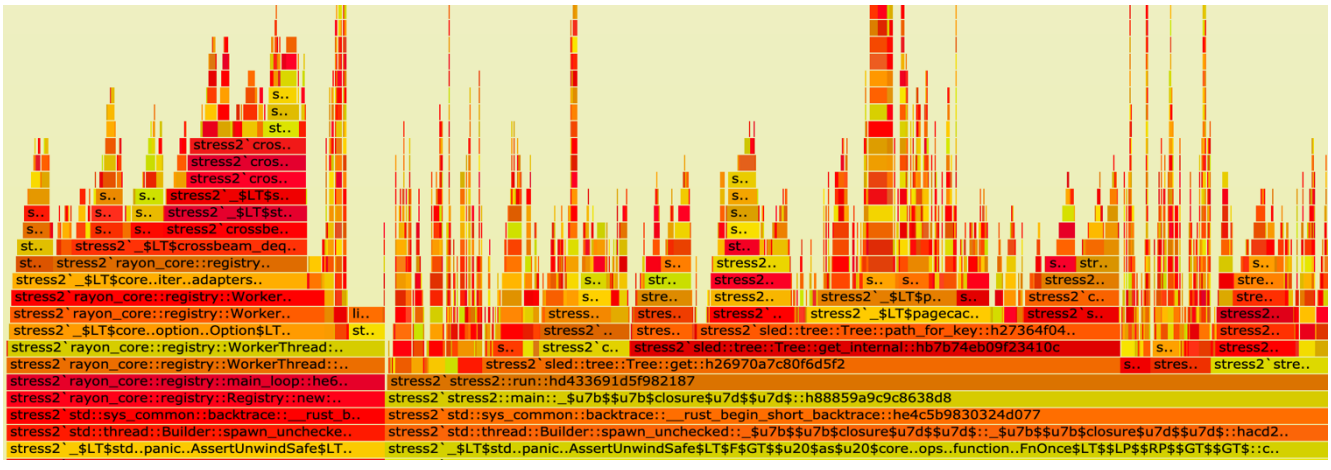

Рисунок 2.2 – Графік стеку викликів згенерований Flamegraph

Особливістю Flamegraph є те, що він дозволяє легко ідентифікувати функції, які займають найбільшу частину часу виконання. Блоки у Flamegraph розташовуються один над іншим, що відображає стек викликів: нижні блоки представляють функції, які були викликані першими, а верхні - ті, які були викликані пізніше. Це дозволяє розробникам швидко визначати, які частини коду споживають найбільше ресурсів.

Flamegraph особливо корисний у мові програмування Rust, оскільки Rust часто використовується для розробки додатків з високими вимогами до продуктивності. Rust має складну систему власності та запозичення, що може призвести до неочікуваної поведінки з точки зору продуктивності. За допомогою flamegraph розробники можуть точно визначити, де саме виникають затримки у виконанні програми, і оптимізувати ці ділянки коду.

Крім того, Flamegraph може виявити неефективні алгоритми або надмірні виклики функцій, які можуть бути прихованими при звичайному читанні коду. Це робить інструмент незамінним при рефакторингу та оптимізації існуючих програм, дозволяючи розробникам зосередитися на тих аспектах коду, які найбільше впливають на продуктивність.

Висновки до розділу 2

У цьому розділі було наведено інструменти та технології використані під час розробки, було обрано сучасну низькорівневу мову програмування Rust із повним набором супутніх технологій таких як Cargo, Clippy, Rustfmt, Rust Analyzer.

Було використане супутнє програмне забезпечення спрямоване на дослідження продуктивності та ефективності програмного коду. Була розглянута можливість написання модульних та інтеграційних тестів для перевірки коректності вихідного коду.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ МЕТОДІВ ЗМЕНШЕННЯ СКЛАДНОСТІ АЛГОРИТМУ КАРПА-РАБІНА

Для реалізацій методів зменшення складності алгоритму Карпа-Рабіна, в першу чергу необхідно реалізувати сам алгоритм перевірки символів, поліноміальну хеш-функцію, яка підтримує кільцеве додавання та віднімання символів.

Після реалізації алгоритму та запропонованих оптимізацій, необхідно розробити прикладний застосунок, для двох цілей: практичного використання розроблених оптимізацій та перевірки цих оптимізацій та вимірювання ефективності у реальних умовах.

3.1 Реалізація алгоритму Карпа-Рабіна та поліноміальної хеш-функції

Алгоритм Карпа-Рабіна реалізується у такий спосіб [28]: спершу обчислюється хеш-функція шуканого підрядку і хеш-функція першого вікна, що має довжину шуканого підрядку, в тексті. Використовуючи метод кільцевого хешування за допомогою поліноміальної хеш-функції, алгоритм здійснює ефективне обчислення хеш-функцій для наступних вікон у тексті.

Поліноміальна хеш-функція обчислюється наступним чином: кожен символ у підрядку (або вікні тексту) підноситься до ступеня, який відповідає його позиції, і потім множиться на константу (зазвичай просте число), і всі ці значення сумуються. Це значення потім береться за модулем іншого простого числа, щоб зменшити розмір хешу.

Кільцеве хешування дозволяє ефективно обчислити хеш-функцію для наступного вікна, використовуючи хеш-функцію попереднього вікна, без

необхідності обчислення хешу з нуля. Це досягається шляхом відняття значення, що відповідає старшому символу попереднього вікна, додаванням значення нового символу, і потім множенням результату на константу [29].

Щоб бути ефективним для вирішення проблеми зіставлення рядків, хеш функція повинна мати такі властивості:

1. ефективно обчислювана;

2. висока диференціація для рядків;

3. $hash(y[j+1..j+m])$ має легко обчислюватися з $hash(y[j..j+m-1])$ та $y[j+m]$: тобто $hash(y[j+1..j+m]) = rehash(y[j], y[j+m], hash(y[j..j+m-1])$.

Для слова w довжини m нехай $hash(w)$ буде визначено таким чином: $hash(w[0..m-1]) = (w[0] * 2^{m-1} + w[1] * 2^{m-2} + \cdots + w[m-1] * 2^0) \ mod \ q$, де q – зазвичай велике просте число. Тоді $rehash(a, b, h) = \big((h - a * 2^{m-1}) * 2 + b\big) \ mod \ q$.

Фаза попередньої обробки алгоритму Карпа-Рабіна полягає в обчисленні $hash(x)$. Цей хеш обчислюється тільки один раз перед початком виконання основного алгоритму та його асимптотична складність становить $O(m)$.

Під час фази пошуку достатньо порівняти $hash(x)$ із $hash(y[j..j+m-1])$ для $0 \leq j < n - m$. Якщо відповідність значень обчислених хешів знайдено, все одно необхідно посимвольно перевірити рівність $x = \mathrm{y[\,j \ldots j + m - 1]}$, оскільки можуть виникнути колізії хеш-функції: за однакових числових значень хешів можуть бути різні послідовності символів.

Часова складність фази пошуку алгоритму Карпа-Рабіна дорівнює $O(mn)$. Його очікувана кількість порівнянь текстових символів дорівнює $O(n+m)$. На практиці, використана поліноміальна хеш функція жодного разу не дала колізії, тому завжди спрацьовував найкращий сценарій.

Можна помітити, що піднесення до степеню перших символів рядку можуть спричинити переповнення типу $u32$, тобто 32-бітного типу який представляє ціле число, який використовувався для обчислення хешу. Наприклад при довжині підрядку $m = 64$, перший символ необхідно помножити на 2^{64}, що значно більше

ніж максимальне значення типу $u32$ в якого максимальне значення становить 4,294,967,295.

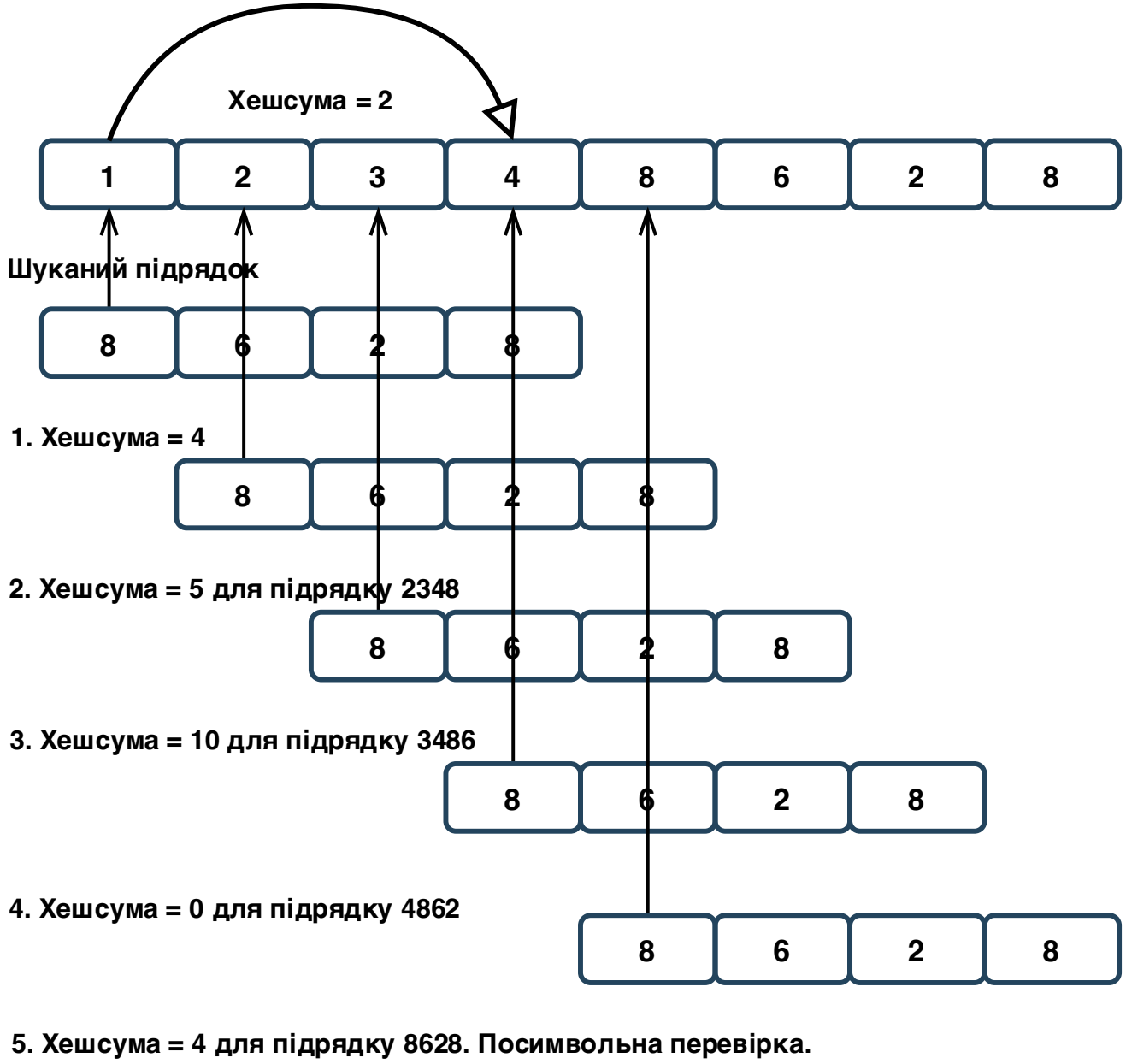

Рисунок 3.1 – Перерахунок вікна кільцевого хешу

Мова Rust пропонує спеціальні типи даних сімейства Wrapping, які дозволяють виконувати операції з переповненнями. Достатньо лише обгорнути тип u32 у тип Wrapping і можна буде безпечно використовувати переповнення типу.

Завдяки властивостям поліноміальної хеш-функції, переповнення не впливає на точність отриманого хешу.

Також, можна використати числову оптимізацію, яка можлива завдяки властивостям збереження чисел у двійковій системі числення: замість використання функції піднесення до степеню $pow(2, x)$ можна використати побітовий зсув на один розряд вправо, що еквівалентно множенню на 2. Така оптимізація значно збільшує ефективність адже зсув бітів набагато дешевший для процесору ніж функція піднесення до степеню.

На рисунку 3.1 зображено етапи виконання алгоритму Карпа-Рабіна. Для шуканого рядку 8628 обчислюється числове значення хешу, яке становить 4, тобто $hash(8628) = 4$. Для першого вікна в тексті також обчислюється значення хешу $hash(1234) = 2$. На етапі перевірки порівнюються тільки два числа 2 та 4. Оскільки значення обчисленого хешу не збігаються, обраховується наступне значення вікна тексту на основі попереднього хешу. Це триває доти, доки значення хешів підрядку та вікна тексту не будуть збігатися, тоді можна виконувати перевірку по символам.

3.2 Реалізація методу паралелізації та розбиття даних на менші частини

Порівняно простою, але надзвичайно ефективною оптимізацією є паралельне застосування алгоритму Карпа-Рабіна для читання та пошуку по файлу. В ідеальному варіанті, часову складність виконання можна буде зменшити до t разів, де t – це кількість логічних процесорів системи.

Наївна імплементація не потребує методів синхронізації між потоками, достатньо лише правильно розподілити між ними роботу. Необхідно виділити кожному потоку робочий буфер в який буде відбуватися читання з файлу, надати чергу з корисним навантаженням [30] і дочекатися моменту виконання всіх потоків.

Унаслідок того, що читання з файлу може відбуватися паралельно, не можна використовувати функції які рухають його курсор адже це може порушити цілісність даних. Замість цього, кожен потік має зберігати власний курсор на індекс байту в файлі, з якого він почав читати. Стандартна бібліотека мови Rust дає можливість читання файлу з указаним зсувом, тому таке рішення просто реалізувати.

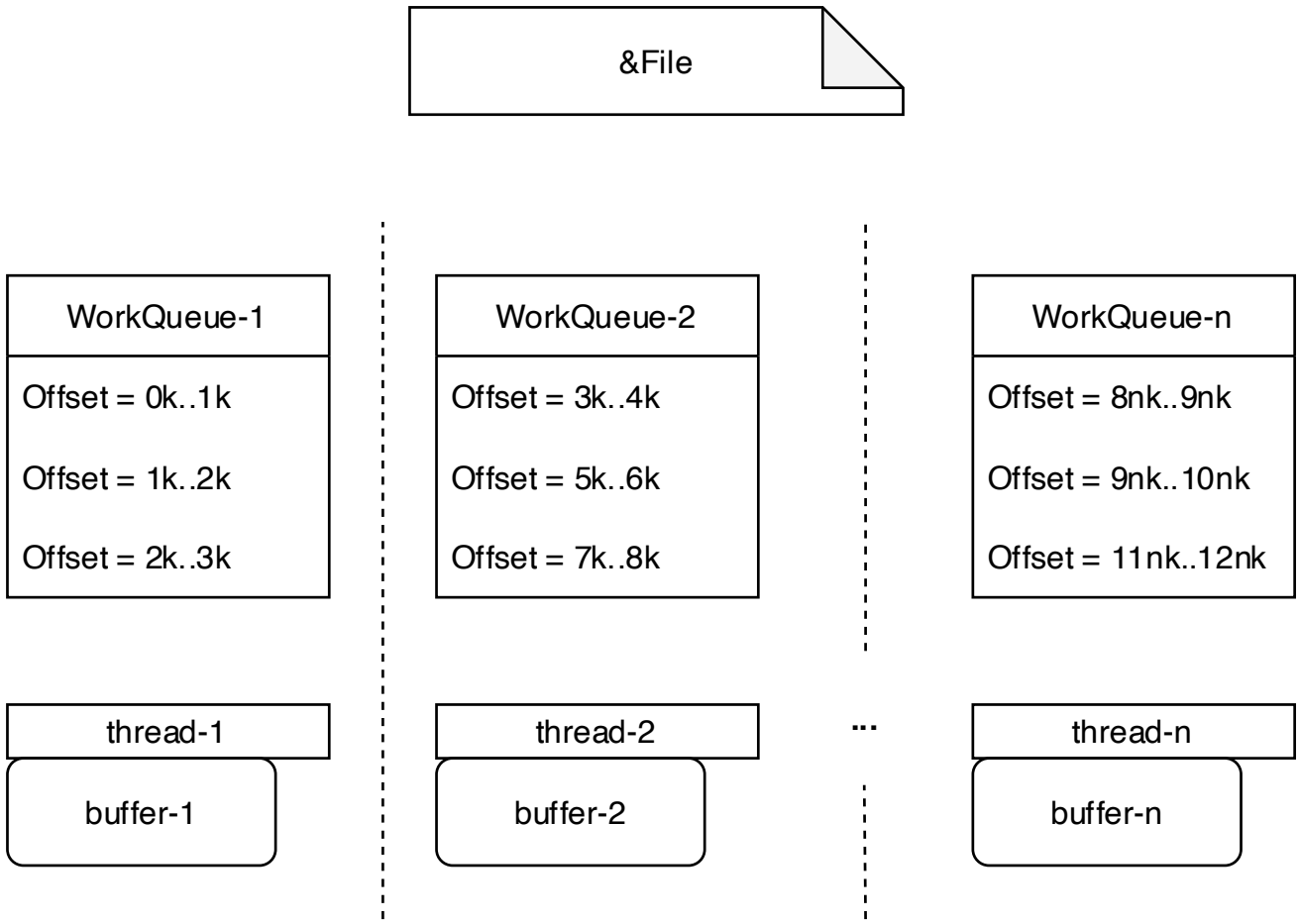

Рисунок 3.2 – Схематичне зображення розбиття даних на частини для паралельного виконання

На рисунку 3.2 зображено яким чином пам’ять розподіляється між потоками. Кожен буфер назначений під конкретний потік. Кожен потік має свою чергу, з якої він бере зсув в файлі, для читання. Важливо уточнити, що в черзі знаходяться не самі байти для пошуку, а тільки індекс початку та індекс кінця ділянки файлу,

таким чином значно економиться оперативна пам’ять для роботи програми. Використавши таку оптимізацію, кількість роботи виконувана одним логічним процесором зменшиться пропорційно кількості процесорів системи.

3.3 Реалізація методу викрадення роботи

Незважаючи на те, що наївна реалізація паралелізації показує значний приріст ефективності, вона все ще не є оптимальною. У ідеальних умовах (коли на одному процесорі запущена тільки одна програма), така реалізація спрацювала б чудово, проте на сучасних комп’ютерах такий сценарій є нереалістичним. У випадку коли планувальник операційної системи розподіляє ресурси між іншими програмами, виконання потоків що займаються пошуком може бути призупинене, таким чином одні потоки будуть виконувати свою роботу швидше за інші і це буде призводити до, що вони будуть простоювати без корисного навантаження [31].

Метод викрадення роботи покликаний уникнути цієї проблеми. Завдяки ньому загальна кількість роботи буде розподілятися більш рівномірно, під час виконання самої програми. Таким чином всі потоки будуть навантажені доти, доки існує ще невиконана робота.

Розглянемо реалізацію цього методу детальніше. До вже описаного методу паралелізації пропонується додаються додаткові сутності, завдяки яким можна реалізувати описану логіку без порушення цілісності виконання:

1. WorkQueue – двостороння, глобальна черга із загальним обсягом роботи;

2. Worker – структура що включає в себе локальну чергу потоку, буфер пам’яті та назначеного Stealer;

3. Stealer – це структура яка видима для інших Stealer що розподілені між потоками, та яка має доступ до глобальної черги WorkQueue. За замовчуванням, Worker звертається у локальну чергу та у випадку якщо там порожньо, звертається до назначеного Stealer.

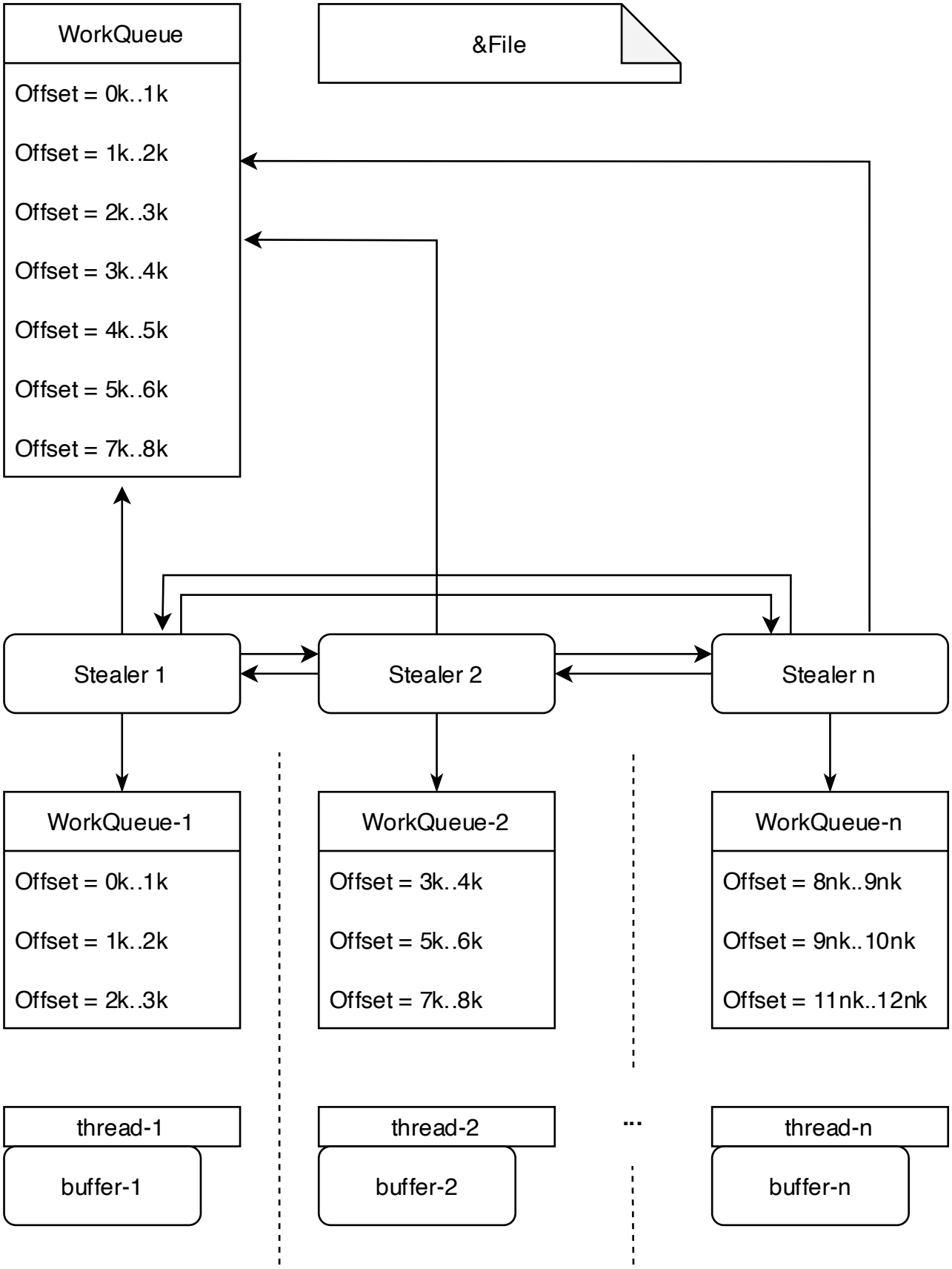

Рисунок 3.3 – Схематичне зображення викрадення роботи

Ключовим елементом такої архітектури є об’єкт Stealer, зображений на рисунку 3.3. Він відповідальний за наповнення черги потоку, до якого він назначений. У випадку коли потік виконує свою роботу та в локальній черзі немає нової роботи, він звертається до Stealer, який в свою чергу може звернутися до глобальної черги і взяти роботу звідти. Якщо ж глобальна черга теж порожня, Stealer має доступ до всіх Stealer інших потоків та намагається дістати нову роботу звідти. Якщо всі Stealer повідомляють що їх черга пуста – це означає що весь розмічений файл було оброблено та пошук можна завершити.

У чорновому варіанті реалізації такого алгоритму використовувалися наступні примітиви синхронізації [32]:

1. Arc – англійською “atomic reference counting type”, це тип, що дозволяє зробити певну ділянку оперативної пам’яті безпечною для передачі між іншими потоками. Значення залишається доступним для читання, але недоступним для зміни;

2. Mutex – англійською “mutual exclusion primitive”, це примітив, що дозволяє безпечно блокувати ресурс для його модифікації. При використанні м’ютексу ресурс завжди блокується як для читання, так і для запису;

3. RwLock – англійською “read-write lock” тип що схожий на Mutex, він теж блокує ресурс і запобігає взаємній модифікації, але на відміну від Mutex ресурс RwLock можна використати для читання без блокування. Це буває корисно, коли модифікації ресурсу відбуваються значно рідше, ніж його читання.

Під час рефакторингу та тестування було вирішено відмовитися від примітивів Mutex та RwLock і віддана перевага лінеаризованим типам даних. Замість блокування черги для операцій вставки та видалення, можна використати типи сімейства Atomic як вказівник на поточну позицію в черзі. При зверненні до черги достатньо лише зсунути значення вказівнику і почати читати дані з черги. Це не буде призводити до порушення цілісності черги адже інші потоки будуть читати вже з нової позиції вказівнику. Така, на перший погляд, незначна оптимізація дозволяє не тільки спростити читабельність вихідного коду шляхом прибирання типів-обгорток RwLock, та Mutex але і напряму впливає на продуктивність

програми. Це дозволяє зменшити кількість алокацій пам’яті та, що більш важливо, не блокує ресурс для читання та запису адже елементи черги залишаються в ній та не видаляються.

3.4 Побудова програмного застосунку

Для того щоб перевірити ефективність описаних методів та надати можливість кінцевим користувачам користуватися перевагами запропонованих оптимізацій, необхідно розробити застосунок командного рядку, який надасть інтерфейс взаємодії що дозволить здійснювати пошук по файлам.

3.4.1 Архітектура програмного застосунку

Однією з головних вимог до архітектури застосунку – є мати можливість запускати різні алгоритми та оптимізації без повторної компіляції. Це значно спростить взаємодію як з кодовою базою: не потрібно кожен раз шукати місце в програмі де вимкнути чи увімкнути ту чи іншу оптимізацію, так і розширить можливості самого застосунку.

У якості архітектурного каркасу був використаний застосунок командного рядку з відкритим вихідним кодом ripgrep [33]. Він бере на себе такі рутинні речі як парсинг аргументів командного рядку, виведення допоміжної інформації, логування та трасування [34] і дозволяє сфокусуватися на безпосередньому вирішенні задачі.

На діаграмі класів, зображеній на рисунку 3.4 можна прослідкувати залежності між головними компонентами системи. Оскільки у застосунку чітко визначений момент початку та закінчення роботи, вхідною точкою є функція main, у якій здійснюється парсинг аргументів та ініціалізація основних типів даних, в залежності від переданих аргументів.

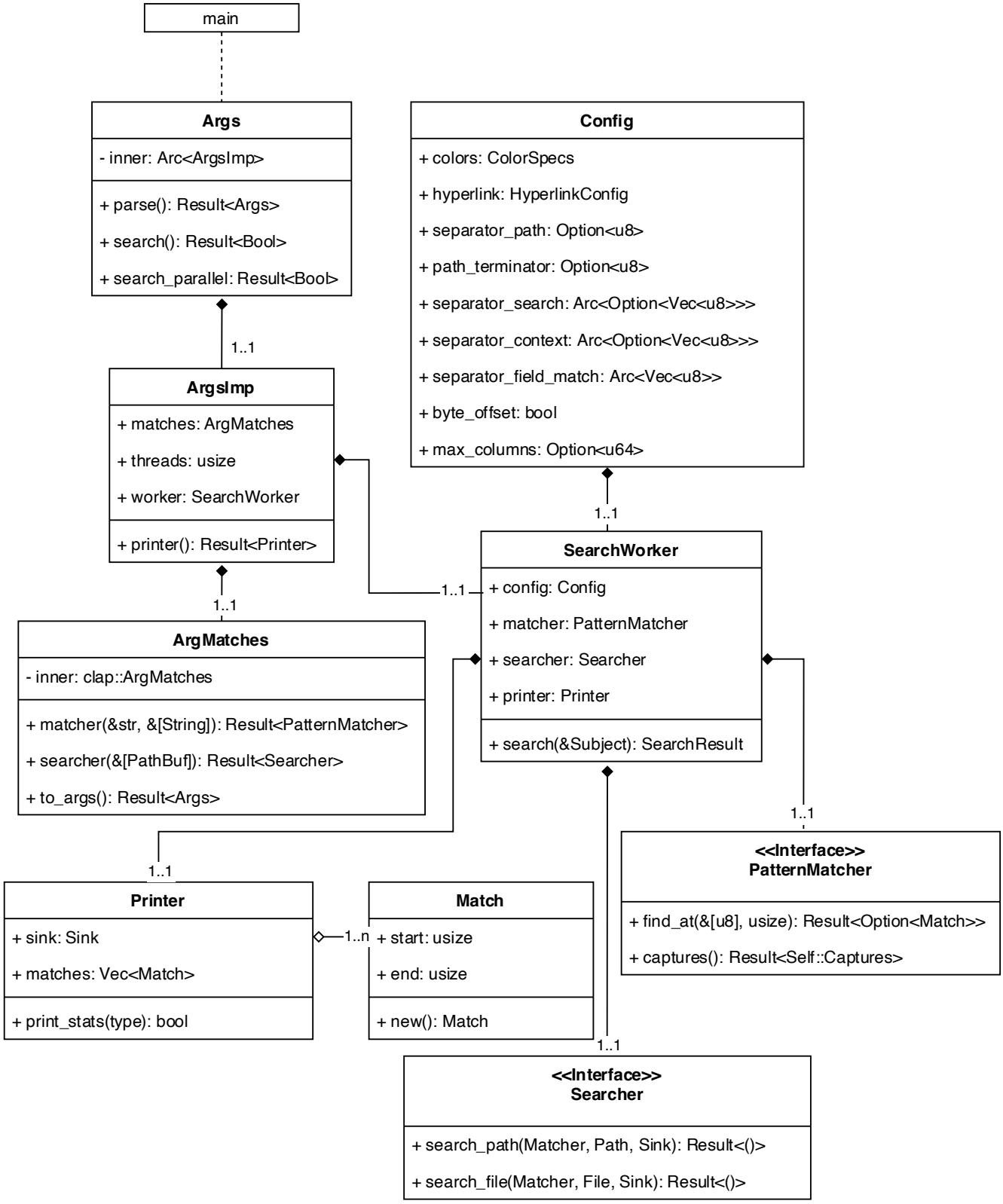

Рисунок 3.4 – Діаграма класів системи

Для того, щоб мати змогу під час роботи застосунку змінювати алгоритм пошуку та стратегію паралелізації, було використано параметричний поліморфізм

типів. Структури Searcher та PatternMatcher, зображені на рисунку 3.4 не є конкретними імплементаціями, а є інтерфейсами [35], що дозволяє підставляти замість них будь яку кількість імплементацій цих інтерфейсів.

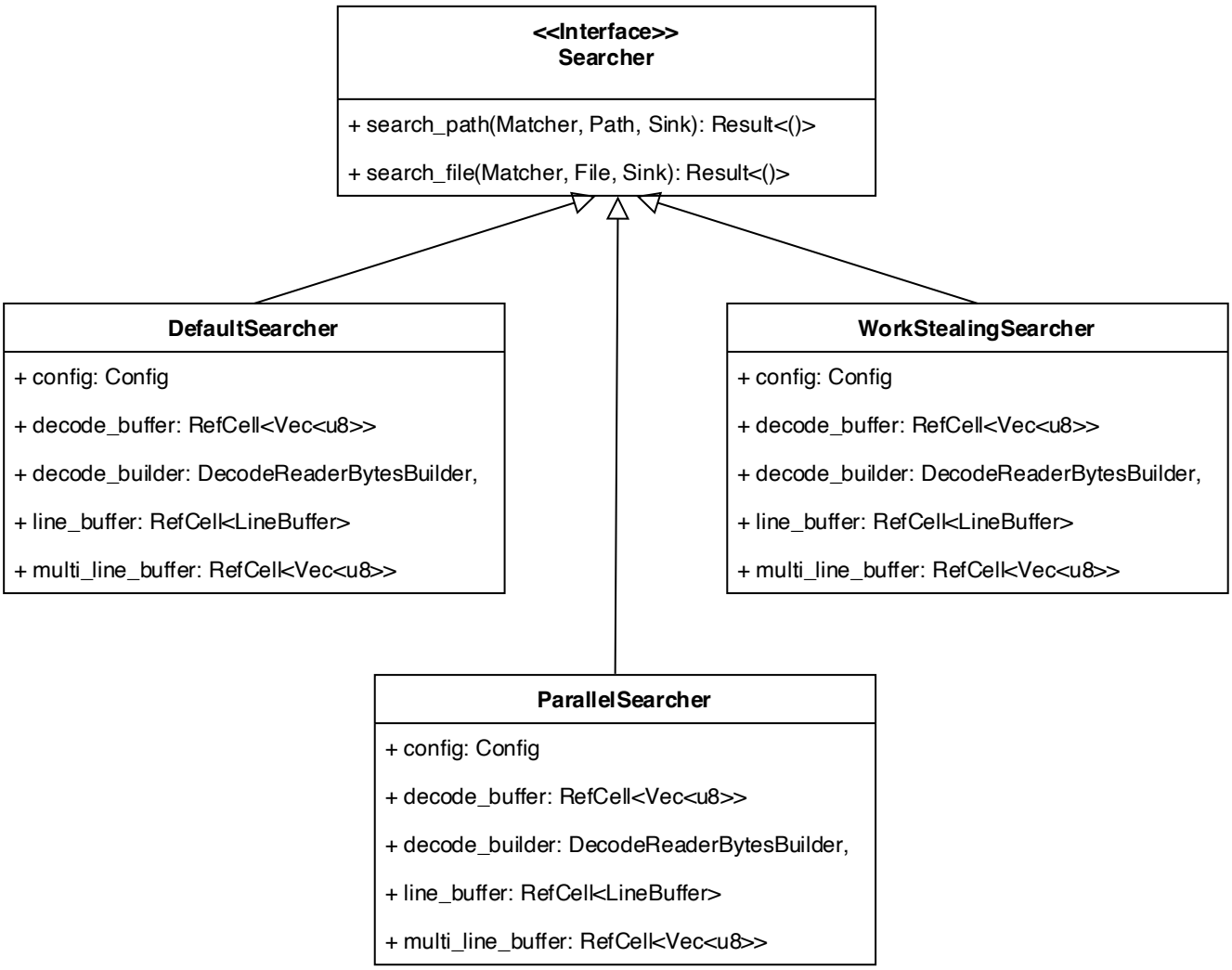

Рисунок 3.5 – Імплементації інтерфейсу Searcher

На рисунку 3.5 зображені всі імплементації інтерфейсу Searcher. Структура DefaultSearcher відповідальна за виконання пошуку за звичайною стратегією, без паралелізації виконання. Структура ParallelSearcher, як зрозуміло з назви додає можливість паралельного пошуку. І, насамкінець, структура WorkStealingSearcher використовує алгоритм викрадення роботи. Аргументи командного рядку визначають яка з конкретних імплементацій буде використана [36] в той чи інший момент запуску програми. Центральним компонентом системи є SearchWorker,

який поєднує імплементацію Searcher з імплементацією PatternMatcher, яка необхідна для Searcher, щоб здійснювати пошук.

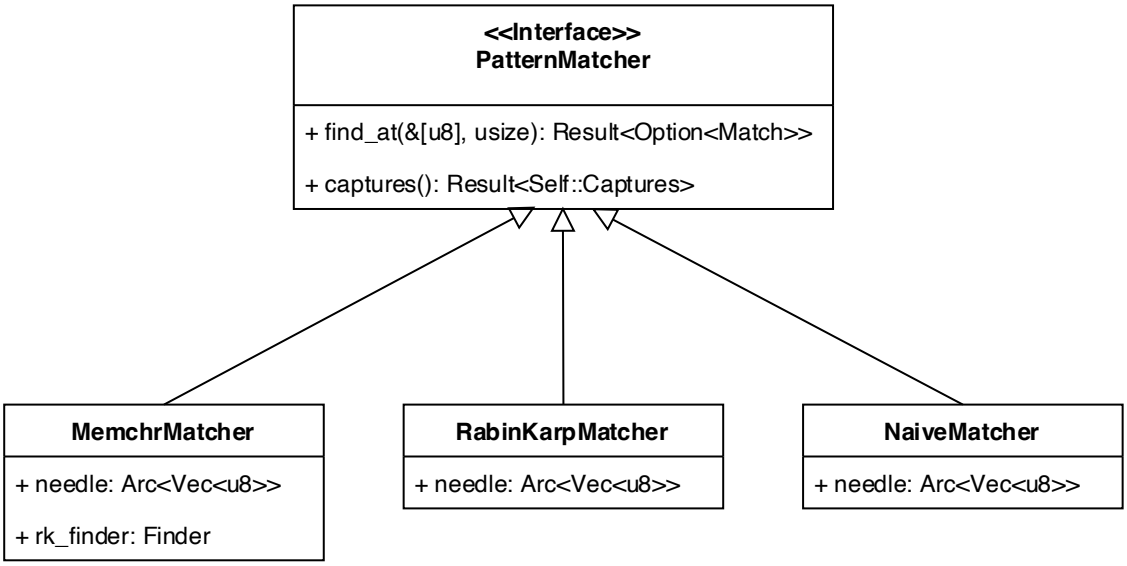

Рисунок 3.6 – Імплементації інтерфейсу PatternMatcher

На рисунку 3.6 зображено імплементації інтерфейсу PatternMatcher. Ці структури безпосередньо здійснюють пошук по масиву байтів, який їм надав Searcher. Структура RabinKarpMatcher – це імплементація алгоритму Карпа-Рабіна. Структура NaiveMatcher – це наївна імплементація пошуку, без будь яких додаткових оптимізацій. Вони всі створені з так званою “голкою”, тобто підрядком, який шукається в тексті. Стандартизований інтерфейс PatternMatcher дозволяє одночасно мати декілька імплементацій різних алгоритмів пошуку.

3.4.2 Функціонал програмного застосунку

Головною задачею розробленого застосунку є пошук по файлам. Він оптимізований здійснювати пошук як і по великій кількості файлів малого розміру, так і по невеликій кількості файлів великого розміру, що покриває більшість сценаріїв використання дослідників у Big Data. Завдяки рекурсивному обходу

директорії не потрібно вказувати конкретні файли по яким буде здійснюватися пошук, хоча така можливість також існує. Також можливо вказати файли по яким не потрібно здійснювати пошук. Підтримується автоматичне фільтрування файлів вказаних у файлі з назвою .gitignore. Застосунок може бути легко вбудований у інші програми завдяки підтримці виводу у формат даних JSON.

3.4.3 Інтерфейс командного рядку

Завдяки бібліотеці Clap можна легко додавати та обробляти нові аргументи командного рядку [38]. Ця бібліотека покриває повний обсяг взаємодії з парсингом, валідацією, виведенням допоміжної інформації для аргументів командного рядку.

Було розроблено допоміжні структури даних, які на основі текстової інформації, прочитаної за допомогою бібліотеки Clap, створюють усі структури, необхідні для повноцінної роботи програми. Таким чином, зважаючи на передані аргументи командного рядку, програма конфігурується під час виконання. Інтерфейс дозволяє вказувати кількість потоків, які може використовувати програма, алгоритм пошуку та методи оптимізації. Додано додатковий прапорець для виведення даних у формат JSON [37], замість звичайного тексту. Також є можливість вказати символ переносу рядка за допомогою спеціального аргументу.

3.4.4 Тестування програмного застосунку

На відміну від більшості мов програмування, мова Rust має підтримку тестування першого класу [39]. Модульні тести можна писати поруч із вихідним кодом, що робить процес тестування дуже зручним. Синтаксис мови дозволяє робити перевірки коректності, та компілятор може працювати в двох режимах: продуктивності та тестування, компілюючи окремо код тестів, що дозволяє не засмічувати бінарний файл цими ж тестами.

Переважна більшість розробленого функціоналу була протестована за допомогою модульних тестів, включаючи такі структури як RabinKarpMatcher, NaiveMatcher, ParallelSearcher, WorkStealingSearcher.

Були використані інструменти Flamegraph та Hyperfine для перевірки продуктивності програми та для знаходження вузьких місць [40]. Завдяки використанню цих застосунків було знайдено та виплавлено декілька помилок, які сповільнювали роботу програми. Також під час тестування використовувався підхід з логуванням та трасуванням, навіть у фінальному варіанті можна включити логи, що додають значну кількість допоміжної інформації [41] та переконатися у коректності роботи застосунку.

Висновки до розділу 3

У цьому розділі було описано реалізацію алгоритму Карпа-Рабіна, поліноміальної кільцевої хеш-функції, а також методів оптимізації цього алгоритму: розділення даних на менші частини та метод викрадення роботи.

Було наведено діаграму класів та архітектуру системи, розібрано ключові елементи системи, їх ролі та взаємодію. Описано взаємодію між інтерфейсами PatternMatcher та Searcher, а також між імплементаціями цих структур.

Також в розділі описано розробку інтерфейсу командного рядку та використані для цього інструменти. Було описано підхід до модульного тестування та тестування продуктивності застосунку.

У розділі було описано використаний підхід написання модульних тестів для перевірки коректності та надійності роботи програми. Під час тестування було знайдено та виправлено ряд логічних помилок, що спричиняли зменшення ефективності програми. Також під час профілювання та побудови графіків за допомогою Flamegraph було знайдено та оптимізовано найбільш затратні функції.

4 РЕЗУЛЬТАТИ РОБОТИ ТА ВЗАЄМОДІЯ КОРИСТУВАЧА З РОЗРОБЛЕНИМ ЗАСТОСУНКОМ

У цьому розділі описуються результати виконаної роботи, демонструється створений застосунок, який використовує запропоновані методи зменшення складності алгоритму Карпа-Рабіна. Також тут наведені результати та вимірювання ефективності розроблених методів, та проводиться аналіз отриманих даних.

4.1 Результати та вимірювання ефективності використаних методів

Для тестування та вимірювання ефективності були використані інструменти Flamegraph та Hyperfine.

Інструмент для трасування стеку виклику функцій Flamegraph використовувався для пошуку вузьких місць і проблем з продуктивністю. Завдяки ньому було знайдено та виправлено недостатньо оптимальне використання примітивів паралельного програмування і було прийнято рішення відмовитися від використання Mutex та RwLock, описаних у попередньому розділі.

Інструмент бенчмаркінгу Hyperfine використовувався для вимірювання ефективності розроблених методів. Як основний критерій оцінки брався час виконання пошуку. Hyperfine використовує методи статичного аналізу, зібраного на основі декількох запусків, що допомагає більш точно оцінити середній реальний час виконання. Також Hyperfine підтримує так звані запуски для розігріву, та очищення кешу процесора перед кожним окремим заміром, що виключає можливість відхилень та пришвидшення виконання кожного наступного заміру.

Заміри проводилися на 10-ядерному процесорі архітектури arm64 з використанням 8 високопродуктивних ядер з частотою 3228 МГц, та 2 енергоефективних ядер з частотою 2064 МГц. Обсяг RAM в системі становить 32

гігабайти. Читання файлу здійснювалося на SSD зі швидкість 5322 МБ/с. Здійснювався пошук по декільком файлам що містили невпорядкований та неструктурований текст середнім обсягом 30 ГБ. Усі заміри наведені для найгіршого сценарію, коли шуканий підрядок є останньою послідовністю байтів у текстовому файлі.

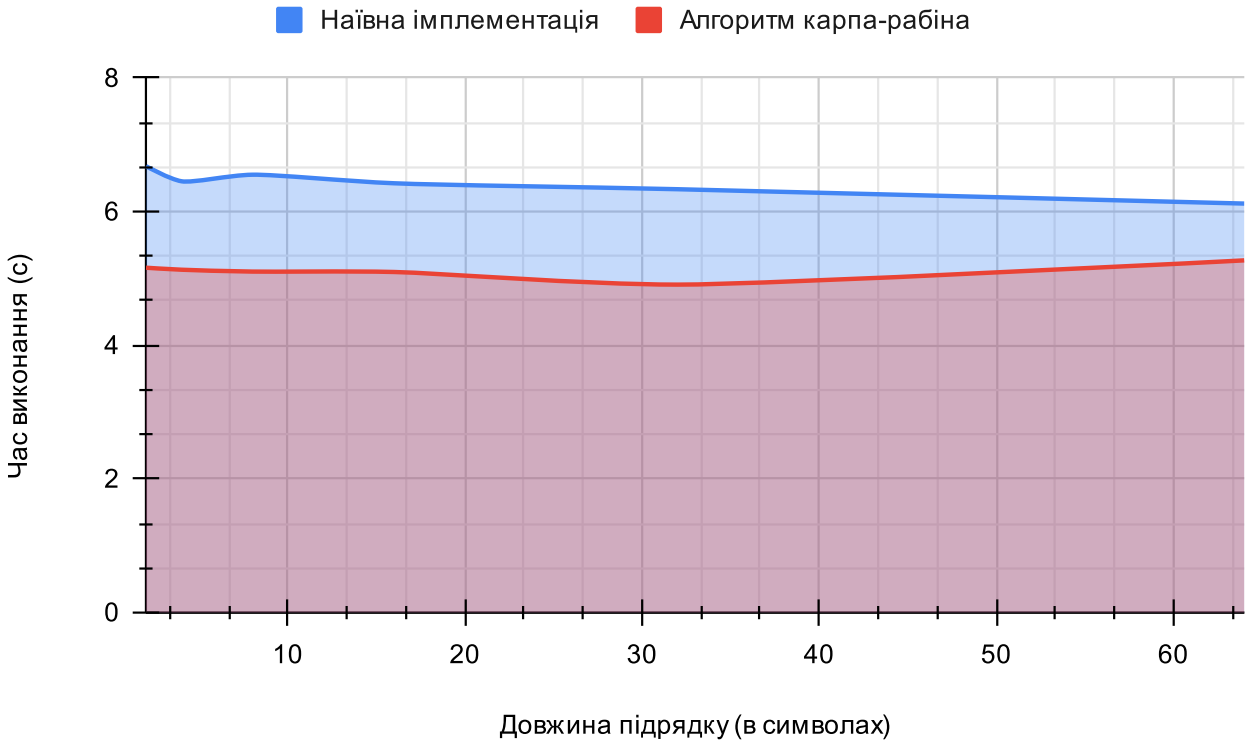

Рисунок 4.1 – Порівняння часу виконання алгоритму Карпа-Рабіна та наївної імплементації пошуку

На рисунку 4.1 зображено графік залежності часу виконання пошуку від кількості символів у підрядку. На графіку наведено алгоритм Карпа-Рабіна та пошук наївним способом. Можна переконатися, що пошук алгоритму Карпа-Рабіна в середньому працює швидше на всьому проміжку кількості символів. Однак, на практиці не було виявлено кореляції між збільшенням кількості символів та збільшенням розриву по часу виконання на користь алгоритму Карпа-Рабіна, як того очікувалося. Це може бути пов’язано з тим, що сучасні процесори дуже

швидко виконують порівняння набору байтів, особливо якщо вони розташовані поспіль, одне за одним.

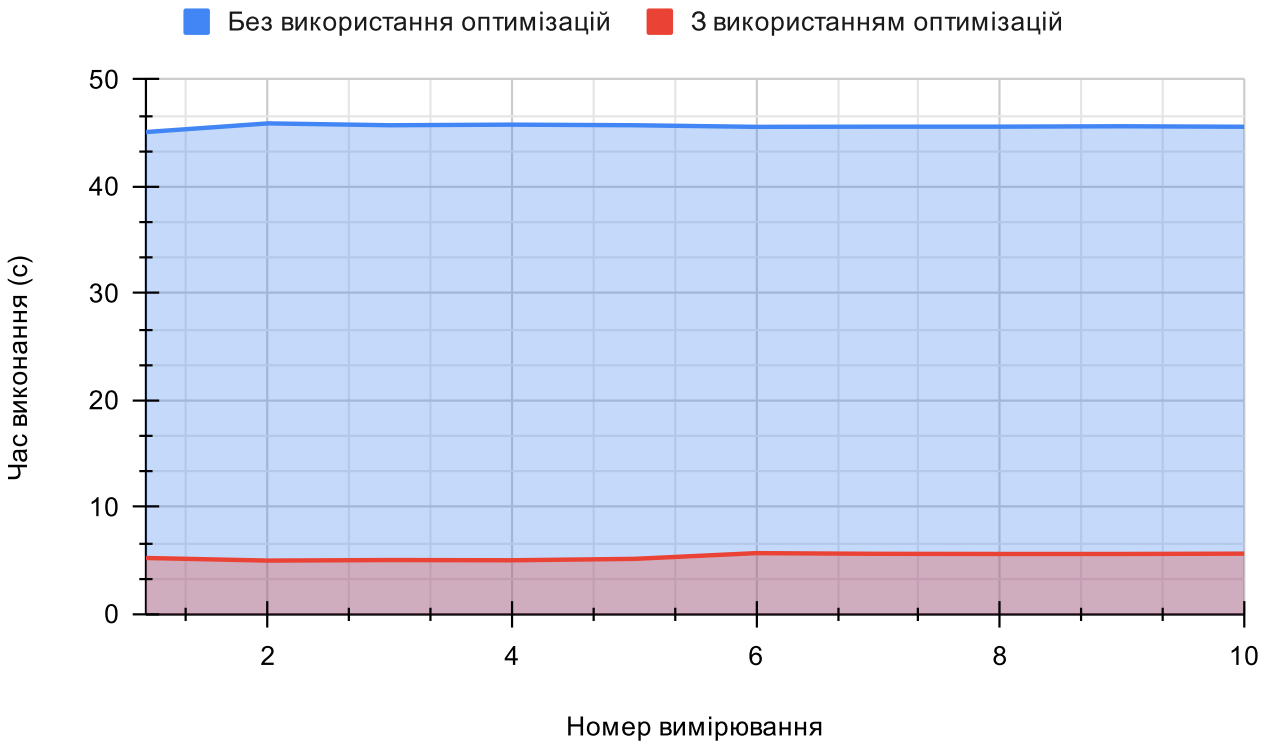

Рисунок 4.2 – Графік порівняння часу виконання алгоритму Карпа-Рабіна з та без застосування розроблених методів зменшення складності

На рисунку 4.2 наведено графік результату 10 вимірювань, що демонструють різницю в часі виконання алгоритму Карпа-Рабіна без використаних оптимізацій та з методами паралельного виконання та викрадення роботи. Середній час виконання алгоритму без оптимізацій становить 45.6409 секунд, а зі всіма оптимізаціями 5.3896 секунд, тобто приріст майже у 8.5 рази.

Важливо мати на увазі, що виконання алгоритму відбувалося на системі з десятьма логічними процесорами, тому приріст продуктивності на один логічний процесор можна оцінити як $0.85 * t$, де t – це кількість логічних процесорів використаних під час пошуку. Також, варто зазначити, що метод паралельного

виконання без викрадення роботи працює повільніше в середньому на 20%, ніж з викраденням роботи, тобто цей коефіцієнт без викрадення роботи становить 0.7.

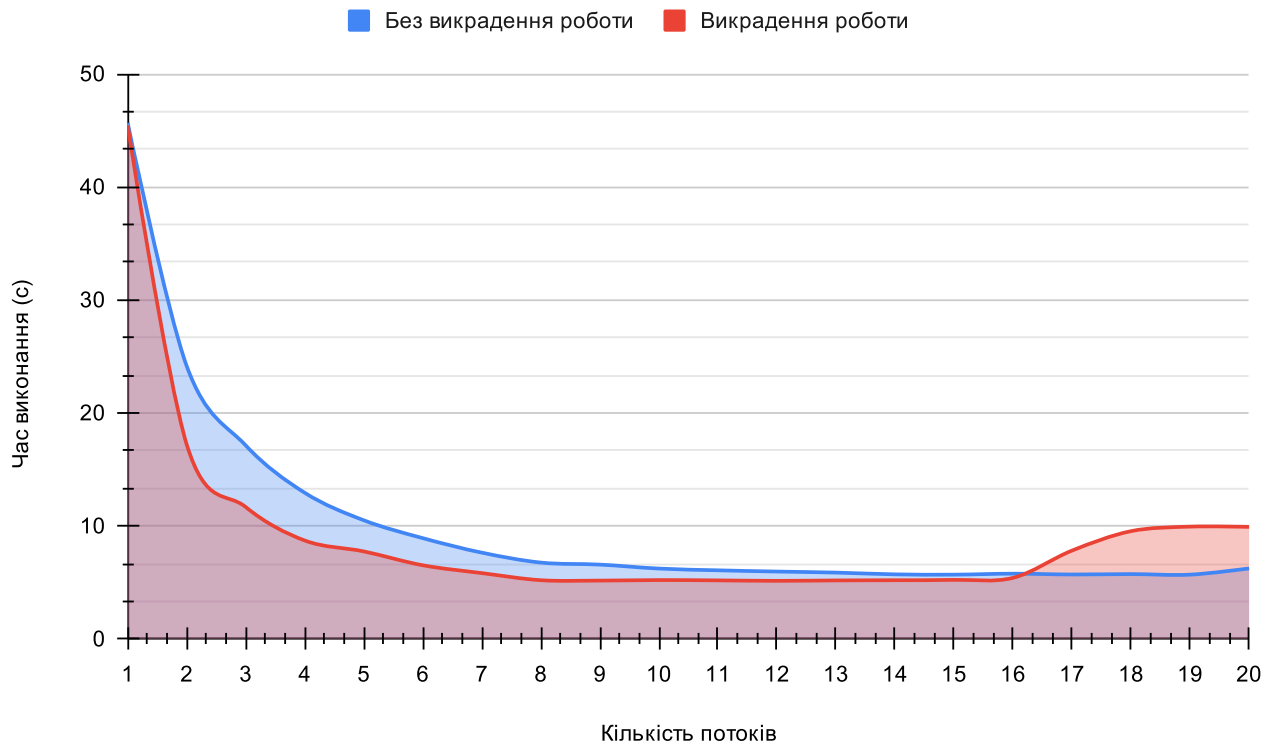

Рисунок 4.3 – Графік залежності часу виконання алгоритму від кількості потоків

На рисунку 4.3 зображено графік залежності часу виконання від кількості потоків використаних під час пошуку з викраденням роботи, та без викрадення роботи. Можна спостерігати ефект що на незначному показнику паралелізації, коли кількість потоків становить від 2 до 5, метод викрадення роботи показує найбільшу перевагу перед звичайним паралельним виконанням. Це дуже просто пояснити, справа в тому що чим менша кількість потоків буде використана, між якими розподіляється робота, тим більш нерівномірним буде цей розподіл, тому викрадення роботи, яке покликане нівелювати цю нерівномірність і показує значно кращі результати.

Із використанням кількості потоків що дорівнює кількості логічних процесорів системи, метод викрадення роботи показує перевагу в середньому в

20%, порівняно зі звичайним паралельним виконанням, що також є доволі непоганим показником приросту ефективності.

Цікавий ефект спостерігається коли використовується більша кількість потоків, ніж є логічних процесорів у системі. Метод викрадення роботи різко погіршує свій час виконання з 5.2 секунд до 9.9 секунд (майже в 2 рази), при тому як звичайне паралельне виконання майже не збільшує час виконання. Це можна пояснити тим, що логіка виконання викрадення роботи є значно складнішою ніж без неї, у випадку коли на одне ядро припадає декілька потоків що виконують пошук та намагаються викрадати роботу одне в одного, планувальник операційної системи менш ефективно розподіляє між ними ресурси процесору.

4.2 Робота користувача зі створеною програмою

У ході виконання роботи було розроблено програмний застосунок, за допомогою якого і були проведені виміри. Як основу було взято застосунок командного рядку з відкритим вихідним кодом ripgrep, у якому було використано його існуючий функціонал, та доповнено розробленими методами оптимізації за допомогою паралельного пошуку по файлу та пошуку за допомогою алгоритму Карпа-Рабіна.

Скомпілюваний бінарний файл застосунку не потребує спеціального встановлення на систему потенційних користувачів, достатньо лише його завантажити, відкрити Shell операційної системи та почати роботу. Для підвищення зручності, щоб мати змогу використовувати застосунок глобально в системі, а не тільки безпосередньо в директорії де знаходиться фізичний файл, можна додати шлях до застосунку в глобальну змінну PATH [42], яка підтримується сучасними операційними системами Windows, Mac та Linux. Проте цей крок не є обов’язковим.

Варто зазначити, що застосунок є крос-платформенним, тобто може бути запущений майже на всіх сучасних архітектурах процесорів, включаючи arm64, x86-64, x86-32, та на всіх сучасних операційних системах.

Для того, щоб почати роботу, достатньо розмістити файл у папці що лежить поруч, або глибше по ієрархії файлової системи, та ввести команду rg "$pattern$", де замість pattern, потрібно вказати шуканий підрядок. Застосунок автоматично просканує кожен файл на наявність вказаного підрядку. Також підтримується можливість вказати конкретний файл, по якому потрібно здійснювати пошук.

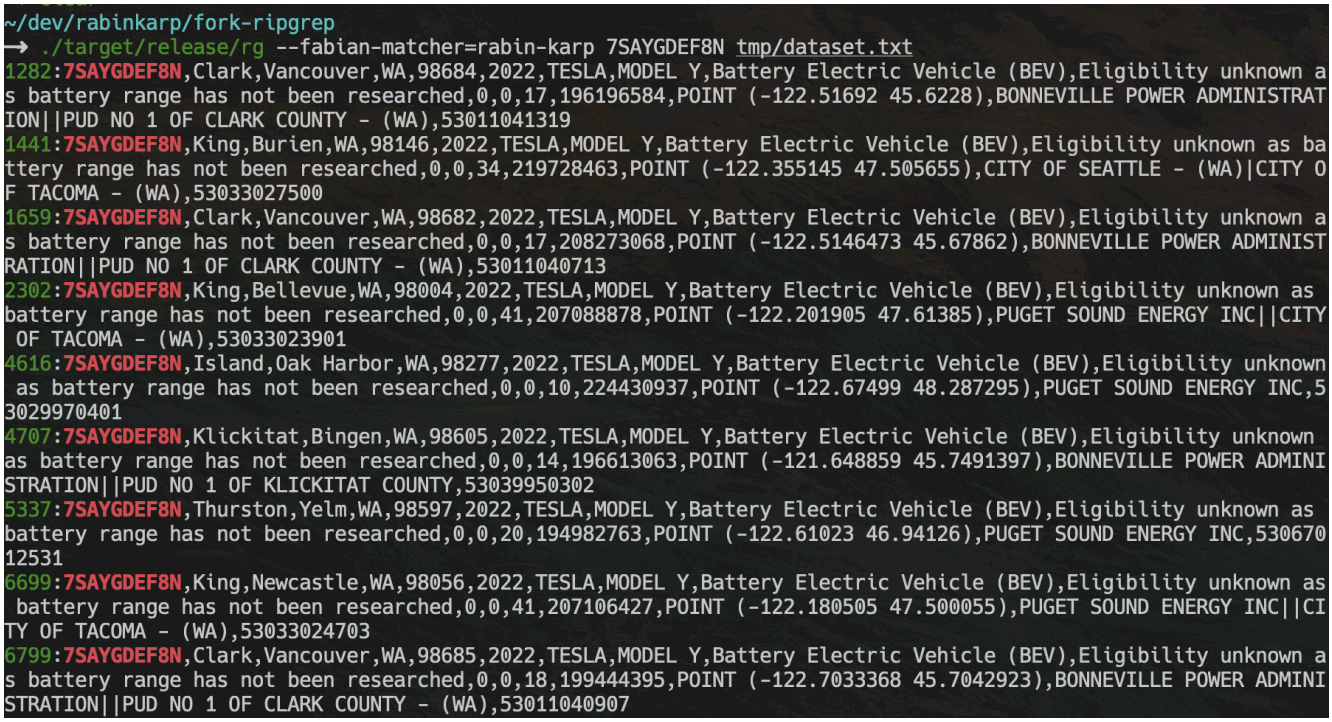

Рисунок 4.4 – Демонстрація результату пошуку

На рисунку 4.4 зображено приклад пошуку алгоритмом Рабіна-Карпа. Виведення тексту відбувається за допомогою функціоналу наявного в застосунку ripgrep, підтримується можливість вказування виведення кількості рядків до та після знайденого підрядку, підтримується увімкнення та вимкнення виведення з кольорами, що допомагає більш наглядно знайти шуканий підрядок. Також виведення інформації відбувається рядок за рядком та вказується номер рядку. Завдяки таким покращенням досвіду користування, застосунок є доволі зручним

для використання не тільки автоматизованим системам, а й живій людині, бо результати виводяться у форматі, що легко сприйняти без додаткової обробки.

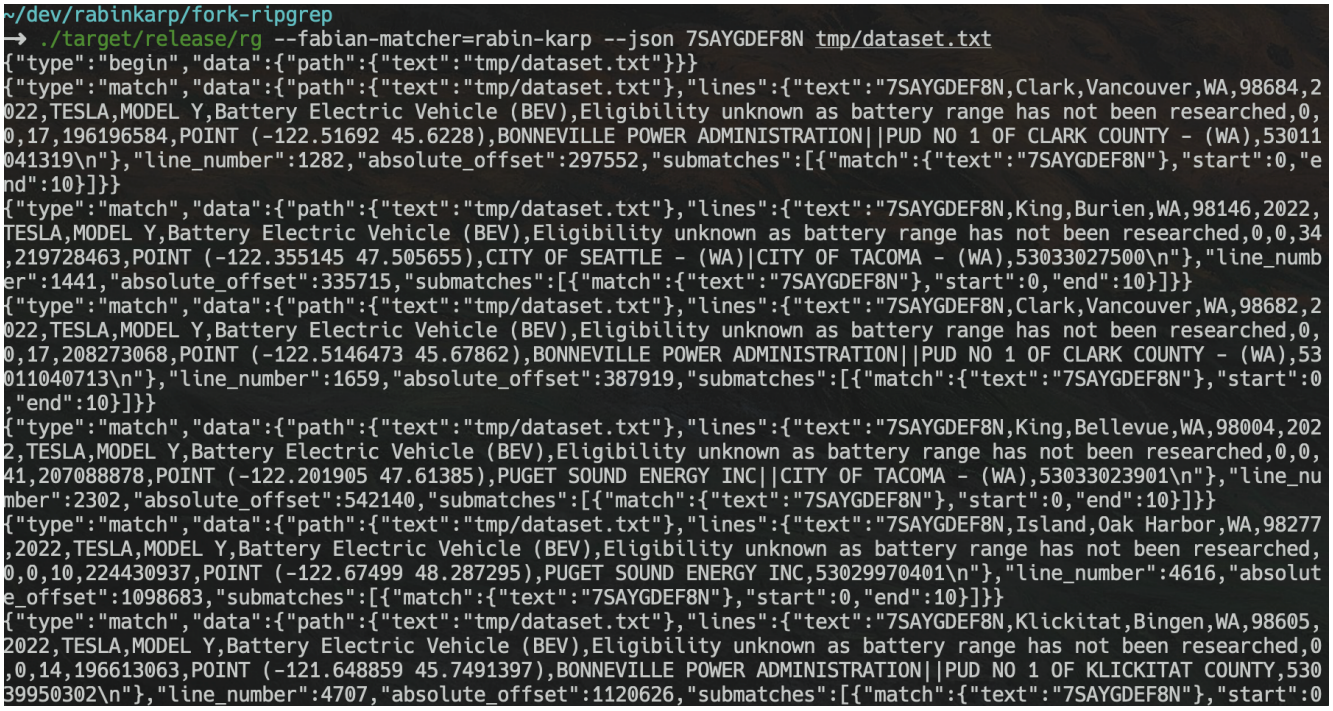

Рисунок 4.5 – Виведення результатів у форматі JSON

Застосунок також підтримує виведення результатів у форматі JSON, яке продемонстроване на рисунку 4.5, що робить його зручним для вбудування в інші програмні продукти.

4.3 Висновки до розділу 4

У цьому розділі було наведено результати проведеного дослідження. Було встановлено, що запропоновані методи зменшують часову складність алгоритму в $0.85 * t$ разів, де t – це кількість логічних процесорів системи.

Також у цьому розділі було продемонстровано роботу розробленого програмного застосунку у двох режимах виведення результатів: у тому, який підходить для ручного пошуку, та у форматі JSON.

5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

У межах цього розділу необхідно висвітлити маркетингові аспекти впровадження та розробки стартап-проекту. Необхідно описати ідею стартап-проекту, провести її технологічний аудит, проаналізувати ринкові можливості стартап-проекту, розробити ринкову стратегію та маркетингову програму продукту.

5.1 Опис ідеї стартап-проекту

У межах цього пункту необхідно проаналізувати зміст запропонованої ідеї, напрямків, до яких можна застосувати ці ідею, а також які вигоди може отримати кінцевий користувач програмного продукту.

Ці позиції подано у вигляді таблиці 5.1

Таблиця 5.1 — Загальні дані стартап-проекту

Зміст ідеї (що пропонується)	Напрямки застосування	Вигоди для користувача
Розробка системи (SaaS) з використанням хмарних технологій що надає API для збереження файлів та здійснення швидкого пошуку по ним.	1. Завантаження та збереження текстової інформації великих обсягів.	1. Відсутність необхідності налаштовувати інфраструктуру для зберігання файлів.
	2. Використання спроможностей хмарних обчислень для здійснення пошуку.	2. Можливість швидкого пошуку по завантаженим файлам.

У таблиці 5.2 проводиться порівняльний аналіз переваг ідеї, порівняно із пропозиціями конкурентів.

Таблиця 5.2. – Визначення характеристик ідеї проекту

Техніко-економічні характеристики ідеї	Концепція (стратегія) конкурентів			Слабкі (W), нейтральні (N) та сильні (S) сторони		
Назва продукту	Власний проект	Сервіс DropBox	Сервіс GoogleDrive	W	N	S
Наявність веб-інтерфейсу застосунку	Ні	Так	Так		+	
Наявність ліміту по використанню сховища	Відсутність ліміту	2 терабайти	2 терабайти			+
Можливість пошуку файлів за назвою	Так	Так	Так		+	
Можливість пошуку вмісту файлів	Так	Ні	Ні			+
Призначення	Спеціалізоване призначення для пошуку в Big Data	Загальне призначення	Загальне призначення			+

Визначений перелік характеристик та властивостей ідеї товару є підґрунтям для формування його конкурентоспроможності. Відсутність вузького спрямування конкурентів створює незакриту нішу для компаній які мають необхідність зберігати великі обсяги текстової інформації та швидко здійснювати пошук по їх вмісту. Такий функціонал надає конкурентну перевагу перед проектами що вже існують.

5.2 Технологічний аудит ідеї проекту

У цьому розділі проведено аудит технологій, за допомогою яких можна реалізувати ідею проекту. Технологічна здійсненність ідеї проекту представлена на таблиці 5.3.

Таблиця 5.3 — Технологічна здійсненність ідеї проекту

№ з/п	Ідея проекту	Технології реалізації	Наявність технологій	Доступність технологій
1	Графічний інтерфейс системи	Мова програмування Rust, фреймворк Tauri	Наявна	Умовна безкоштовно
2	Розширювана база даних	SQL, PostgreSQL	Наявна	Умовна безкоштовно
3	Реалізація API	Мова програмування Java	Наявна	Умовна безкоштовно

Кінець таблиці 5.3

4	Реалізація алгоритму пошуку	Мова програмування Rust, пакетний менеджер Cargo	Наявна	Умовна безкоштовно
5	Використання хмарних спроможностей датацентру	Використання API обраного хмарного провайдера	Наявна	Доступна за моделями підписки та використання ресурсів CPU

Проаналізувавши таблицю можна зробити висновок про можливість технологічної реалізації проекту. Витрати на використання необхідних технологій майже відсутні, за винятком використання сховища та ресурсів хмарного провайдера. За умови підняття seed раунду інвестицій, ідея стартапу має місце на існування.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Для того, щоб спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів необхідно визначити ринкові можливості, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту.

Аналіз ринкових можливостей можна розділити на декілька етапів.

Перший етап — аналіз попиту, він включає в себе: наявність попиту, обсяг, динаміку розвитку ринку. Ці показники представлено в таблиці 5.4.

Таблиця 5.4 — Попередня характеристика потенційного ринку стартап-проекту

№ з/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	2
2	Загальна потреба в продукції	Необхідна
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузі (або по ринку), %	50%

За результатом аналізу таблиці можна зробити висновок, що ринок є привабливим для входження. Незважаючи на те що найбільші гравці займають крупну долю цього ринку, наявність чіткого позиціонування під технологічне використання даного продукту дає можливість виграти конкурентну боротьбу для компаній, яким необхідна ця вузька технологічна ніша.

Другий етап — визначення потенційних груп клієнтів. Аналіз цього етапу представлено в таблиці 5.5. В результаті цього етапу формується перелік вимог до товару.

Таблиця 5.5 — Характеристика потенційних клієнтів стартап-проекту

№ з/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару

Кінець таблиці 5.5

1	Зберігання та пошук великих об’ємів тексту та можливість пошуку по ним	Компанії що досліджують великі обсяги текстової інформації та мають необхідність пошуку по ній	Компанії заключають довгострокові договори, нові клієнти надають перевагу пробному періоду	стабільність роботи; невисока ціна; випробувальний період; програма навчання; наявність документації; підтримка супроводу;

Після визначення потенційних груп клієнтів наступним етапом є аналіз ринкового середовища. До нього входять: фактори загроз і фактори можливостей.

Фактори подаються у вигляді таблиць з колонками:

1. Фактор;

2. Зміст загроз/можливостей;

3. Можлива реакція компанії.

Завдяки такому аналізу можна скоригувати напрямок розвитку компанії відповідно до наявних ризиків та можливостей.

Фактори загроз та можливостей представлено в таблицях 5.6 і 5.7 відповідно.

Таблиця 5.6 — Фактори загроз

№ з/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Обмеженість функцій	Обмеженість наявних функцій і відсутність деяких функцій, які мають конкуренти	Додавання нових функцій за потреби, доопрацювання наявних

Кінець таблиці 5.6

2	Поява нових конкурентів у представленій ніші	Можлива поява нових конкурентів, які можуть створити більш якісний продукт	Постійна розробка удосконалень, розширення асортименту
3	Економічний спад	Відсутність попиту на програмний продукт через економічну складову	Зменшення ціни; зміна цільової аудиторії.
4	Зниження репутації компанії	Можлива ситуація, коли конкуренти спроможуться на більший попит	Проведення рекламних та промо-акцій для програмного продукту

Таблиця 5.7 – Фактори можливостей

№ з/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Відсутність спеціалізованого хмарного сервісу для зберігання та пошуку великих обсягів тексту	Конкуренти не пропонують спеціалізованих інструментів для вирішення поставленої задачі	Компанія зможе економити на загальному функціоналі конкурентів, що не використовується

Кінець таблиці 5.7

2	Динамічність ринку попиту	З попитом на навчання великих мовних моделей (LLM), можливість зберігати та мати доступ до великих обсягів тексту набуває великої актуальності	Розробка сервісу що задовольняє попит

Наступним етапом є аналіз пропозиції, в таблиці 5.8 визначено загальні риси конкуренції на ринку.

Для проведення аналізу конкуренції використовуються два способи: ступеневий аналіз і аналіз конкуренції в галузі за М. Портером.

Аналіз за М. Портером являється більш детальним аналізом умов конкуренції.

Ступеневий аналіз конкуренції представлений у таблиці 5.8.

Таблиця 5.8 — Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії щоб бути конкурентоспроможними)
1. Вказати тип конкуренції монополія/олігополія/ монополістична/чиста	олігополія	Підтримка якості продукту, постійні нововведення, вдосконалення

Кінець таблиці 5.8

2. За рівнем конкурентної боротьби: локальний/національний	національний	Створити основу продукту так, щоб можна було переробити його для використання в інших країнах
3. За галузевою ознакою – міжгалузева/ внутрішньогалузева	міжгалузева	Постійне вдосконалення продукту, що не має прив'язки до сфери
4. Конкуренція за видами товарів: — товарно-родова — товарно-видова — між бажаннями	товарно-видова	Розробити програмний продукт, враховуючи недоліки конкурентів
5. За характером конкурентних переваг: цінова / нецінова	нецінова	Використання дешевших технологій ніж ті що використовують конкуренти, але лише якщо технології відповідають необхідним вимогам якості
6. За інтенсивністю: марочна/не марочна	марочна	Надання функцій, які не надають конкуренти

Аналіз конкуренції за Майклом Портером представлено у таблиці 5.9.

Таблиця 5.9 — Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
		Dassault Systems	Визначити фактори сили в постачальників	Контроль якості	Відсутні

Кінець таблиці 5.9

Висновки:	Інтенсивність конкурентної боротьби незначна	Є можливості виходу на ринок, бо існуючі рішення не надають потрібних переваг	Наявні усі можливості входу на ринок. Потенційні конкуренти не виявлені. Строки виходу на ринок невідомо	Клієнти вказують вимоги згідно з умовами експлуатації	Немає обмежень

Враховуючі результати ступеневого аналізу та аналізу за М. Портером можна зробити висновок про можливість роботи на ринку з огляду на конкурентну ситуацію.

На основі аналізу конкуренції за М. Портером, проведеного у таблиці 5.9, а також із врахуванням характеристик ідеї проекту в таблиці 5.2, вимог споживачів до товару в таблиці 5.5 та факторів маркетингового середовища в таблицях 5.6 і 5.7 визначимо та обґрунтуємо перелік факторів конкурентоспроможності в таблиці 5.10.

На основі аналізу конкуренції, з урахуванням характеристик ідеї проекту, вимог споживачів до товару та факторів маркетингового середовища визначається перелік факторів конкурентоспроможності. Аналіз проведено у таблиці 5.10.

Таблиця 5.10 — Обґрунтування факторів конкурентоспроможності

№ з/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Використання сучасних технологій	Використання сучасних технологій збільшить конкурентоспроможність продукту

Кінець таблиці 5.10

2	Пропозиція функцій, відсутніх у конкурентів	Зважаючи на відсутність функціоналу з пошуку вмісту файлів у конкурентів надає ринкову перевагу розробленому продукту. Наявність використаних оптимізацій для швидкодії застосунку дозволяє закріпитися на цьому ринку.

За визначеними факторами конкурентоспроможності проводиться аналіз сильних та слабких сторін стартап-проекту. Дані представлено у таблиці 5.11.

Таблиця 5.11 — Порівняльний аналіз сильних та слабких сторін проекту

№ з/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів конкурентів у порівняні з даним продуктом						
			-3	-2	-1	0	1	2	3
1	Використання сучасних технологій	15				+			
2	Пропозиція функцій, відсутніх у конкурентів	20			+				

Останнім етапом ринкового аналізу можливостей впровадження являється складання SWOT-аналізу (матриця аналізу сильних, англ. Strength та слабких, англ. Weak сторін, загроз, англ. Troubles та можливостей, англ.Opportunities) на основі виділених ринкових загроз та можливостей, а також сильних і слабких сторін проекту. SWOT аналіз є ключовим інструментом стратегічного планування в бізнесі, що допомагає організаціям оцінювати свої сильні та слабкі сторони, а також виявляти Можливості та Загрози у зовнішньому середовищі. Виявлення цих аспектів дозволяє компанії оптимізувати свою стратегію та підвищити ефективність.

Аналіз SWOT стартап-проекту проведено на таблиці 5.12

Таблиця 5.12 — SWOT-аналіз стартап-проекту

Сильні сторони: — Інноваційні технології	Слабкі сторони: — Затрати на розробку нового функціоналу
Можливості: — Вихід на нові ринки — Відсутність повноцінних альтернатив	Загрози: — Обмежена кількість клієнтів — Банкрутство

Після отримання SWOT-аналізу стартап-проекту, створюється альтернативи ринкової поведінки для виведення його на ринок та розраховується приблизний час для ринкової реалізації, відштовхуючись від проектів конкурентів, що також можуть бути виведені на ринок.

Обрані альтернативи аналізуються з огляду строків та ймовірності виконання.

Таблиця 5.13 — Альтернативи ринкового впровадження стартап-проекту

№ з/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Зміна цільового сектору	20%	500 год
2	Пошук нових споживачів	10%	500 год
3	Орієнтація поточної моделі на підприємницький ринок	70%	200 год

Альтернатива, яка найбільше підходить під поставлену задачу та являється більш простим, ймовірним та стислим варіантом — №3. Ймовірність отримання ресурсів 70%, терміни реалізації 200 годин.

5.4 Розроблення ринкової стратегії продукту

Розробка ринкової стратегії продукту поділяється на декілька етапів. Першим кроком є визначення стратегії охоплення ринку. Результатом цього етапу являється таблиця з описом цільових груп споживачів та їх критеріями.

Дані першого етапу розробки ринкової стратегії продукту занесені до таблиці 5.14.

Таблиця 5.14 — Вибір цільових груп потенційних споживачів

№ з/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачі в сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1	Потенційними споживачами є підприємства, які займаються виготовленням продукції	Висока	Вище середньої	Низька	Відсутність спеціалізованих конкурентів
Які цільові групи обрано: компанії, які мають потребу швидко шукати по тексту					

За результатами аналізу потенційних груп споживачів було обрано цільові групи, для яких буде запропоновано даний товар, та визначено стратегію

охоплення ринку — стратегію диференційованого маркетингу (компанія працює з декількома сегментами).

Наступним етапом є формування базової стратегії розвитку. Дані етапу занесені до таблиці 5.15.

Таблиця 5.15 — Визначення базової стратегії розвитку

№ з/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Орієнтація поточної моделі на підприємницький ринок	Диференціації	Можливість налаштувати систему під потреби користувача	Спеціалізації

Наступним кроком необхідно обрати стратегію конкурентної поведінки. Її результати представлено в таблиці 5.16.

Таблиця 5.16 — Визначення базової стратегії конкурентної поведінки

№ з/п	Чи є проект “першопрохідцем” на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1	Так	Шукати нових користувачів	Ні	Стратегія зняття конкурентної ніші

На основі обраних сегментів до постачальника та продукту та використовуючи обрану базову стратегію розвитку та стратегії конкурентної поведінки необхідно розробити стратегію позиціонування. Ця стратегія полягає у формулюванні ринкової позиції за допомогою яких споживачі можуть ідентифікувати торгівельну марку. Дані цього кроку представлені в таблиці 5.17.

Таблиця 5.17 — Визначення стратегії позиціонування

№ з/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Стабільність та швидкість роботи. Невисока ціна	Стратегія диференціації	Позиція на основі порівняння фірми з товарами конкурентів	Споживачі – підприємства які займаються виготовленням технічних виробів

Отже, робота стартап-проекту повинна бути спланована певним чином: за стратегією диференціації буде виконаний і поширюватись програмний продукт з унікальними якостями, дотримуючись у конкурентній поведінці стратегії виклику лідера, тобто випускається один тип товару для усіх користувачів.

5.5 Розроблення маркетингової програми проекту

Першим кроком створення маркетингової програми проектує формування маркетингової концепції товару. В результаті отримано таблицю 5.18, у якій підсумовано результати вже зробленого аналізу конкурентоспроможності товару.

Таблиця 5.18 — Визначення ключових переваг концепції потенційного товару

№ з/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Швидкий економічний ефект	Швидкий економічний ефект	Практична відсутність конкурентів
2	Оцінка якості продукту	Оцінка відносно стандартів	Висока точність оцінки програмної системи

Після визначення ключових переваг розроблюється трирівнева маркетингова модель товару. До складу цієї моделі входить: уточнення ідеї продукту, його фізичні складові та особливості. Ці дані показано в таблиці 6.19.

Таблиця 5.19 — Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові
I. Товар за задумом	Хмарний сервіс провайдер сховища та здійснення пошуку по файлам що містять текстову інформацію великих обсягів, необмеженою в кількості зайнятого місця.

Кінець таблиці 5.19

II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	1. Розмір програмного забезпечення	Нм	Тх
	2. Інтерфейс користувача	Нм	Тх
	3. Модульність	Нм	Вр
	Якість: стандарти, нормативи, тестування, тощо		
	Пакування: відсутнє		
	Марка: Сервіс пошуку тексту		
III. Товар із підкріпленням	До продажу: відсутнє Після продажу: персональна підтримка за додаткову платню		

М/Нм – монотонні або немонотонні;

Вр/Тх/Тл/Е/Ор – вартісні, технічні, технологічні, ергономічні або органолептичні (останній – для продуктів харчування).

Наступним етапом є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар. Дані представлені у таблиці 5.20.

Таблиця 5.20 — Визначення меж встановлення ціни

№ з/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів ціль групи споживачів	Верхня та нижня межа встановлення ціни на товар/послугу
1	10 – 20 $	Високий	Високий	8 – 17 $

Наступним кроком є визначення оптимальної системи збуту, в межах якого було прийняте рішення, таблиця 5.21.

Таблиця 5.21 — Формування системи збуту

№ з/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Клієнт отримує одразу повний доступ	Вихід на ринок з новим товаром	Кількість модулів	Збільшення ринкової частки підприємства

Останнім кроком маркетингової програми проекту є концепція маркетингових комунікацій, що базується на обрану основу позиціонування.

Таблиця 5.22 — Концепція маркетингових комунікацій

№ з/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Прямий маркетинг	особистий продаж; продаж за каталогами; та інші засоби ЗМІ (телевізійний маркетинг); Інтернет.	Доступність та об’єктивність інформації про фірму та товар Семінар Виставки	Дослідження ринку, Виявлення потенційних клієнтів	Пропозиція

Результатом підрозділу стала маркетингова програма, що включає в себе концепції товару, збуту, просування та попередній аналіз можливостей ціноутворення, спирається на цінності та потреби потенційних клієнтів,

конкурентні переваги ідеї, стан та динаміку ринкового середовища, в межах якого впроваджено проект, та відповідну обрану альтернативу ринкової поведінки.

Висновки до розділу 5

У цьому розділі було проведено аналіз розробки стартап-проекту та системи яка надасть змогу використовувати розроблені технології.

Побудова спеціалізованого сервісу для зберігання та пошуку текстової інформації є перспективним напрямком адже дозволить закрити необхідність компаніям-клієнтам створювати власні системи для вирішення цієї задачі. Таким чином продукт дозволить швидко продемонструвати економічний ефект для клієнта.

Оскільки головні гравці на ринку не фокусуються над вирішенням спеціалізованої задачі, а займаються розробкою функціоналу загального призначення, у сфері Big Data це відкриває нішу можливостей для розробки спеціалізованого функціоналу, чим продукт компанії і буде відрізнятися та приносити користь своїм клієнтам. Модель ведення бізнесу, коли кінцевими користувачами є інші бізнеси дозволяє сфокусуватися лише на корпоративному функціоналі, нівелюючи потребу в розробці функцій загального призначення.

Базова стратегія розвиту проекту це диференціації, тобто конкурентоспроможність формується через надання споживачеві потрібного товару. На основі ретельного вивчення аналогічних систем було розроблено декілька унікальних характеристик товару.

Перспективи впровадження з огляду на стан конкуренції, на потенційні групи користувачів, та на конкурентоспроможності проекту – прямі, і тільки доводять можливість впровадження, та не марну розробку створеного програмного продукту.

ВИСНОВКИ

У ході магістерської роботи було проведено дослідження проблематики пошуку по невпорядкованому тексту. Було досліджено методи зменшення складності що використовуються як у числових, так і нечислових алгоритмах.

На основі проведеного аналізу було визначено напрямок роботи, та прийнято рішення дослідити та розробити методи паралельних обчислень та методи викрадення роботи для зменшення складності пошуку алгоритмом Карпа-Рабіна що припадає на один логічний процесор.

Було проведено тестування розроблених методів зменшення складності, та виміряно їх ефективність у порівнянні з іншими методами. Підтверджено факт що алгоритм Карпа-Рабіна є більш ефективним алгоритмом у порівнянні з пошуком наївним алгоритмом. Встановлено, що розроблені методи зменшення складності пришвидшують алгоритм пошуку, та зменшують часову складність виконання на один логічний процесор у $0.85 * t$ разів, де t – загальна кількість процесорів системи.

Було розроблено та протестовано програмний застосунок що використовує розроблені методи зменшення складності. До застосунку було застосовано методи буферизації що дозволяють читання та пошук по величезних за обсягом текстових файлах, що дає змогу використовувати його для пошуку в Big Data. Гнучка архітектура застосунку дозволяє його вбудовувати в інші застосунки та використовувати разом з іншими інструментами, які використовують дослідники Big Data.

Програмний застосунок написано мовою Rust із використанням вбудованих можливостей стандартної бібліотеки по паралелізації та синхронізації потоку виконання. Застосунок є скомпільованим бінарним файлом, тому не потребує спеціального функціоналу з встановлення, та є крос-платформенним і працює на всіх сучасних операційних системах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Coded character sets: history and development: Веб-сайт. URL: https://archive.org/details/codedcharacterse00unse (дата звернення: 05.08.2023).
2. Gonnet G. , Baeza-yates R. Text Algorithms // Handbook of Algorithms and Data Structures in Pascal and C, 2nd Edition. Addison-Wesley Publishing Company, 1991. P. 251–288.
3. Crochemore M. Pattern matching and text compression algorithms // CRC Computer Science and Engineering Handbook. CRC Press Inc., Boca Raton, FL, 1996. P. 162-202.
4. Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt Fast Pattern Matching in Strings. SIAM Journal on Computing. 1977. Vol. 6. № 2 P. 323–350. DOI: https://epubs.siam.org/doi/10.1137/0206024
5. Robert S. Boyer, J. Strother Moore A fast string searching algorithm. Communications of the ACM. 1977. Vol. 2. № 10. P. 762–772. DOI: https://dl.acm.org/doi/10.1145/359842.359859
6. Karp R. , Rabin M. Efficient Randomized Pattern-Matching Algorithms // / IBM Journal of Research and Development. 1987. № 31. P. 249–260.
7. Кузьменко І. , Дацюк О. Алгоритми з використанням хешування. Алгоритм Карпа-Рабіна // Базові алгоритми та структури даних. Київ. С. 106–111.
8. Polynomials and the FFT // Introduction to Algorithms / Cormen T. et al. MIT Press, 1990. Topic 34. P. 853–885.
9. Alfred V. Aho, Margaret J. Corasick Efficient string matching: an aid to bibliographic search. Communications of the ACM. 1975. Vol. 18. № 6. P. 333–340. DOI: https://dl.acm.org/doi/10.1145/360825.360855
10. Aho A. Algorithms for finding patterns in strings // Handbook of Theoretical Computer Science, Volume A, Algorithms and complexity. Elsevier, Amsterdam. Topic 5. P. 255–300.
11. Bertrand Meyer Incremental String Matching. Information Processing Letters 21. 1985. P. 219–227. URL:

https://se.inf.ethz.ch/~meyer/publications/string/string_matching.pdf (дата звернення: 15.08.2023).

12. A String-Matching Algorithm: Веб-сайт. URL: https://web.archive.org/web/20171010223532/http://www.hs-albsig.de/studium/wirtschaftsinformatik/Documents/commentzwalterextab.pdf (дата звернення: 07.09.2023).
13. Crochemore M. , Hancart C. Pattern Matching in Strings // Algorithms and Theory of Computation Handbook. CRC Press Inc., Boca Raton, FL., 1999. Topic 11. P. 11–28.
14. Aiken, A. Optimal loop parallelization / A. Aiken, A. Nicolau // SIGPLAN Not. – 1988. – Vol. 23, no. 7. – P. 308–317.
15. A Java Fork/Join Framework // Proceedings of ACM Java Grande 2000 Conference. San Francisco, CA, USA, 2000. URL: http://gee.cs.oswego.edu/dl/papers/fj.pdf (дата звернення: 13.07.2023).
16. Epoll(7) — Linux manual page: Веб-сайт. URL: https://man7.org/linux/man-pages/man7/epoll.7.html (дата звернення: 19.09.2023).
17. The Rust Programming Language: Веб-сайт. URL: https://doc.rust-lang.org/book/ (дата звернення: 04.09.2023).
18. The rustc book: Веб-сайт. URL: https://doc.rust-lang.org/rustc/ (дата звернення: 01.09.2023).
19. The Cargo Book: Веб-сайт. URL: https://doc.rust-lang.org/cargo/ (дата звернення: 05.08.2023).
20. GitHub - rust-lang/rust-clippy: Веб-сайт. URL: https://github.com/rust-lang/rust-clippy (дата звернення: 13.09.2023).
21. Rustfmt: Веб-сайт. URL: https://rust-lang.github.io/rustfmt/?version=v1.6.0 (дата звернення: 20.09.2023).
22. Rust-analyzer: Веб-сайт. URL: https://rust-analyzer.github.io/ (дата звернення: 22.09.2023).
23. Learn CLion: Веб-сайт. URL: https://www.jetbrains.com/clion/learn/ (дата звернення: 30.09.2023).

24. Hyperfine - crates.io: Веб-сайт. URL: https://crates.io/crates/hyperfine (дата звернення: 05.10.2023).
25. Flame Graphs: [Веб-сайт]. URL: https://www.brendangregg.com/flamegraphs.html (дата звернення: 20.09.2023).
26. Perf WIki: [Веб-сайт]. URL: https://perf.wiki.kernel.org/index.php/Main_Page (дата звернення: 14.09.2023).
27. Rabin-Karp Algorithm for Pattern Searching: [Веб-сайт]. URL: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/ (дата звернення: 18.08.2023).
28. Sedgewick R. Chapter 19 // Algorithms. Addison-Wesley Publishing Company, 1988. P. 277-292.
29. Attiya G., Hamam Y. Two phase algorithm for load balancing in heterogeneous distributed systems //12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 2004. P. 434-439.
30. Blumofe, R. D., & Papadopoulos, D. (1998). The performance of work stealing in multiprogrammed environments. ACM sigmetrics performance evaluation review, 26(1), 266–267.
31. Gjengset J. Rust for Rustacenans: San Francisco: no starch press, 2022. 168 p.
32. GitHub - BurntSushi/ripgrep: Веб-сайт. URL: https://github.com/BurntSushi/ripgrep (дата звернення: 02.08.2023).
33. Crate log: Веб-сайт. URL: https://docs.rs/log/latest/log/ (дата звернення: 01.09.2023).
34. Abstract Factory: Веб-сайт. URL: https://refactoring.guru/design-patterns/abstract-factory (дата звернення: 31.08.2023).
35. The Rustonomicon: Веб-сайт. URL: https://doc.rust-lang.org/nomicon/ (дата звернення: 14.09.2023).
36. ECMA-404, 2nd edition, December 2017: Веб-сайт. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf (дата звернення: 15.09.2023).

37. Command line apps in Rust: Веб-сайт. URL: https://rust-cli.github.io/book/ (дата звернення: 19.09.2023).
38. Code complete test Mcconnell S. Code Complete. Washington, United States: Cisco Press, 1993. P. 553–571.
39. Code complete test Mcconnell S. Code Complete. Washington, United States: Cisco Press, 1993. P. 525–550.
40. Martin F. Refactoring 2nd Edition: Addison-Wesley Professional. 45 p.
41. Environment Variables: Веб-сайт. URL: https://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap08.html#tag_08_03 (дата звернення: 29.09.2023).