НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування

До захисту допущено:
Завідувач кафедри
_____ Вадим МУХІН
«___»_____2023 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою

“Інтелектуальні сервіс-орієнтовані розподілені обчислювання”

зі спеціальності 122 "Комп'ютерні науки"

на тему: «Пришвидшення алгоритмів нечіткого пошуку на великих масивах даних»

Виконав:
студент IV курсу, групи ДА-91
Шаблій Володимир Сергійович _____

Керівник:
асистент
Клещ Кирило Олегович _____

Консультант з економічного розділу:
Доцент, к.е.н.
Рощина Надія Василівна _____

Рецензент:
Доцент, к.т.н.
Шаповалова Світлана Ігорівна _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2023

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування

Рівень вищої освіти: перший (бакалаврський)

Спеціальність: 122 «Комп'ютерні науки»

Освітньо-професійна програма:

«Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Вадим МУХІН
«___»_____2023 р.

ЗАВДАННЯ
на дипломну роботу студенту
Шаблія Володимира Сергійовича

1. Тема роботи «Пришвидшення алгоритмів нечіткого пошуку на великих масивах даних», керівник роботи Клещ Кирило Олегович, асистент, затверджені наказом по університету від 30 травня 2023 р. №2065-с
2. Термін подання студентом роботи – 15 червня 2023 р.
3. Вихідні дані до роботи:
    1) “Algorithms” Robert Sedgewick, Kevin Wayne;
    2) “Introduction to Algorithms” Thomas H. Cormen
4. Зміст роботи:
    1. Вступ.

2. Огляд предметної області та алгоритмів нечіткого пошуку
3. Програмна реалізація рішень на основі автомату Левенштейна
4. Перевірка та аналіз рішень
5. Функціонально-вартісний аналіз програмного продукту
6. Висновки

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

1. Презентація до захисту роботи.

6. Дата видачі завдання 31.01.2023.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	31.01.2023	
2	Огляд задачі нечіткого пошуку в тексті та областей її застосування	01.04.2023	
3	Дослідження підходів та алгоритмів нечіткого пошуку в тексті	05.04.2023	
4	Порівняння та аналіз існуючих алгоритмів	11.04.2023	
5	Розробка алгоритму на основі автомату Левенштейна	10.05.2023	
6	Порівняння та аналіз отриманих результатів	21.05.2023	
7	Проведення функціонально-вартісного аналізу програмного забезпечення	31.05.2023	
8	Оформлення дипломної роботи	10.06.2023	

Студент Шаблій В. С.

Керівник Клещ К. О.

АНОТАЦІЯ

до дипломної роботи Шаблія Володимира Сергійовича на тему
«Пришвидшення алгоритмів нечіткого пошуку на великих масивах даних»

Результатом дипломної роботи є реалізація мовою програмування C++ алгоритмів нечіткого пошуку на основі скінченних автоматів, що працюють в декілька разів швидше та ефективніше за підхід з використанням тривіального алгоритму розрахунку відстані Дамерау-Левенштейна.

Дипломна робота також включає в себе огляд застосувань та існуючі алгоритми нечіткого пошуку. Проведено тестування коректності, швидкодії та проаналізовано переваги та недоліки розроблених рішень.

Загальний обсяг роботи 90 с., 17 рис., 7 таблиць, 2 додатки, 12 джерел.

Ключові слова: нечіткий пошук, автомат Левенштейна, редагувальна відстань, алгоритм Дамерау-Левенштейна.

ANNOTATION

bachelor's thesis of Volodymyr Shabllii Serhiyovych on “Speeding up fuzzy search algorithms on large data sets”

The result of the thesis is the implementation of fuzzy search algorithms based on finite state machines in C++, which are several times faster and more efficient than the approach using the trivial Damerau-Levenstein distance calculation algorithm.

The thesis also includes a review of applications and existing fuzzy search algorithms. Correctness and performance tests are performed and the advantages and disadvantages of the developed solutions are analyzed.

The total volume of the work is 90 pages, 17 figures, 7 tables, 2 appendices, 12 sources.

Keywords: fuzzy search, Levenshtein automaton, edit distance, Damerau-Levenshtein algorithm.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

НСА – недетермінований скінченний автомат

ДСА – детермінований скінченний автомат

Unicode – стандарт кодування представлення символів усіх писемностей світу та спеціальних символів

ФВА – функціонально-вартісний аналіз.

ПЗ – програмне забезпечення.

ПП – програмний продукт.

C++ – це компільована, статично типізована мова програмування.

Python – це високорівнева, інтерпретована мова програмування із динамічною типізацією.

ВСТУП

В сучасному інформаційному суспільстві великі масиви даних є невід'ємною складовою багатьох сфер діяльності людини, таких як торгівля, медицина, наука, економіка та інші. Нечіткий пошук надає можливість ефективно знаходити потрібну інформацію в масивах даних, які можуть містити неточності, помилки або неповну інформацію. Однак, пошук інформації в таких великих обсягах даних може бути трудомістким і затратним.

Метою цієї дипломної роботи є дослідження та вдосконалення алгоритмів нечіткого пошуку з метою підвищення швидкодії та ефективності пошуку на великих масивів даних.

Буде проведено огляд існуючих алгоритмів нечіткого пошуку: у рамках дипломної роботи будуть проаналізовані основні алгоритми нечіткого пошуку, їх переваги та недоліки. Будуть розглянуті алгоритми, такі як алгоритм Левенштейна, алгоритм Дамерау-Левенштейна, алгоритм Bitap та інші.

Буде розроблено методи пришвидшення нечіткого пошуку і проведено дослідження та оцінку результатів: розроблені методи будуть проаналізовані та порівняні з існуючими алгоритмами нечіткого пошуку. Будуть проведені експерименти на реальних або симульованих даних для оцінки швидкодії, точності та ефективності розроблених методів.

Очікуваними результатами дипломної роботи є вдосконалення алгоритмів нечіткого пошуку на великих масивах даних з метою підвищення швидкодії та ефективності пошуку. Це може мати важливе значення для практичних застосувань, де швидкий і точний пошук є ключовим фактором.

Отже, дослідження алгоритмів нечіткого пошуку та розробка методів пришвидшення є актуальними завданнями, які допоможуть вирішити проблему швидкодії пошуку в великих масивах даних. Результати цієї дипломної роботи можуть бути корисними для використання в різних

галузях, де потрібно швидко та ефективно проводити нечіткий пошук у великих обсягах даних.

1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ

В даному розділі буде проведено огляд предметної області нечіткого пошуку. Також буде розглянуто області застосування, існуючі підходи та алгоритми вирішення задачі нечіткого пошуку.

Основною ідеєю нечіткого пошуку, на відміну від чіткого, полягає у визначенні рядків, що наближено відповідають заданому рядку-шаблону. У випадку чіткого пошуку, необхідно визначити місцезнаходження, присутність або відсутність рядку, що точно відповідає заданому шаблону. При нечіткому пошуці ж допускається деяка кількість помилок, або неточність між заданим, шуканим рядком. При цьому, практичним є, кожному співпадінню надати чисельну характеристику - ступінь схожості знайденого рядка до шаблону.

1.1. Області застосування

Нечіткий пошук можна часто зустріти у багатьох практичних областях науки та техніки, де допускається виникнення помилок, або є необхідність визначення схожих, не обов’язково точно співпадаючих, на задану послідовностей [1].

1.1.1 Автокорекція помилок при введенні тексту користувачем

Однією з областей застосування нечіткого пошуку є автокорекція помилок в тексті. Завдання автокорекції помилок є дуже давнім, та, можливо, одним з найперших практичних застосувань нечіткого пошуку [1].

Дуже часто при введенні слова можна допустити помилку, маючи при цьому словник допустимих слів можна оцінити чи є це слово в цьому словнику, у випадку якщо його там не має, за допомогою нечіткого пошуку можна знайти найбільш схожі варіанти з словника та запропонувати їх користувачу. Більшість сучасних текстових редакторів підтримують цю функцію (рис. 1.1).

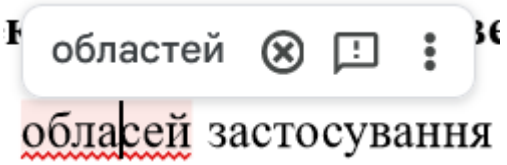

Рисунок 1.1 – Приклад автокорекції помилок в текстовому редакторі Google Docs

1.1.3 Пошукові системи

Нечіткий пошук також може бути застосований в пошукових системах, оскільки при пошуці є бажаним отримати близькі результати навіть за відсутності точних співпадінь.

Нечіткий пошук є важливою функцією для пошукових систем, оскільки допомагає покращити користувацький досвід і забезпечити більш точні результати пошуку, навіть при неповній або неточній інформації в запиті користувача.

У контексті пошукових систем техніки нечіткого пошуку можна використовувати для порівняння наборів слів або символів у рядку запиту та індексованих рядках для пошуку приблизних збігів. Часто використовуються комбінації методів, щоб підвищити точність зіставлення нечітких рядків.

1.1.4 Комп’ютерний зір

Застосування нечіткого пошуку також можна знайти в області комп'ютерного зору. Системи оптичного розпізнавання символів часто стикаються з проблемами через шум, артефакти, варіації шрифтів та інші фактори, які можуть внести помилки в розпізнаний текст. Доволі не рідкими помилками в оптичному розпізнаванні символів згідно до [2] є подвоєння та, навпаки, злиття двох символів в один. Для покращення якості розпізнавання слів можна скористатись методами нечіткого пошуку для коригування помилок.

Системи оптичного розпізнавання символів часто використовують словники або списки слів для підвищення точності розпізнавання. Порівнюючи розпізнаний текст зі словником, методи нечіткого пошуку можуть виправляти розпізнані слова, які не збігаються з жодним словом у словнику. Цей підхід корисний для обробки неправильно написаних або неправильно розпізнаних слів.

1.1.5 Обчислювальна біологія

Підходи та алгоритми нечіткого пошуку можна застосувати в області обчислювальної біології для оцінки міри схожості біологічних послідовностей, при аналізі послідовностей ДНК, РНК та білків, для анотації та пошуку біологічних послідовностей.

Оскільки послідовності ДНК та білків можна представити у вигляді тексту на маленькому алфавіті(A, C, G, T), для них можна використовувати алгоритми що працюють з текстовими даними, проте алгоритми чіткого пошуку шаблонів знаходять дуже мало застосувань в обчислювальній біології, оскільки геноми мають складну структуру і можуть містити мутації, втрати або вставки в генетичних послідовностях.

1.2 Основні поняття, підходи та алгоритми нечіткого пошуку

1.2.1 Редагувальна відстань

Одним з важливих понять нечіткого пошуку для оцінки схожості двох рядків є поняття редагувальної відстані(англ. edit distance), яка визначається як мінімальна необхідна кількість операцій перетворення над одним рядком, щоб він відповідав іншому. Найбільш поширеними операціями перетворення є видалення, вставка, заміна, та перестановка символів. Кожен набір операцій перетворення для обчислення редагувальної відстані має свої різні застосування.

Наприклад, редагувальна відстань між двома рядками, що розраховується як мінімальна кількість заміни символів одного радка щоб перетворити його у другий, відома як відстань Хеммінга, знаходить своє застосування у сфері виявлення та корекції помилок [7].

Найбільш відомим типом редагувальної відстані, що часто застосовується та обчислюється в алгоритмах нечіткого пошуку є відстань Левенштейна. Вона визначається як мінімальна кількість операцій (вставка, видалення та заміна символу), необхідних для перетворення одного рядка на інший.

Модифікація відстані Левенштейна, що більш точно відповідає помилкам користувача при наборі текста є відстань Дамерау-Левенштейна. Вона складається з операцій вставки, видалення, заміни символу та транспозиції двох символів.

Наведемо декілька прикладів. Відстань Левенштейна між словами “free” та “tree” дорівнює 1, що також дорівнює відстані Дамерау-Левенштейна та відстані Хеммінга між цими словами, оскільки достатньо однієї заміни “f” на “t”. Відстань Хеммінга між словами, що мають різні довжини неможливо порахувати, оскільки рахуються тільки заміни одного символу на інший. Відстань Левенштейна між словами “rte” та “tree” дорівнює 3, оскільки можна провести 2 заміни символів та одну вставку символу “e”, проте відстань Дамерау-Левенштейна для цих слів дорівнює 2, оскільки можна зробити транспозицію символів “r” та “t”, щоб отримати “tre” та додати 1 символ “e”.

Також, окрім редагувальної відстані, можна використовувати інші міри подібності для порівняння та знаходження схожих рядків. Наприклад, можна застосовувати відстані, засновані на N-грамах [10]. N-грама це послідовність з n елементів. В нашому випадку, елементами є символи. Наприклад 2-грами для слова “tree” виглядають так: “tr”, “re”, “ee”.

Поєднуючи N-грами з мірами подібності множин, наприклад, подібністю Жаккара, можна отримати міру подібності для рядків, основна ідея якої полягає в тому, що схожі між собою рядки будуть мати спільні підрядки.

1.2.2 Алгоритми нечіткого пошуку

Згідно до [3], методи пошуку в тексті можна поділити на онлайн, та офлайн методи. Для онлайн методів неможливо заздалегідь опрацювати текст, або проіндексувати його. Онлайн методи включають в себе алгоритми, що працюють з послідовностями рядків напряму. Проте швидкодія онлайн методів обмежена необхідністю послідовно обробляти кожен рядок. Офлайн методи працюють з попередньо обробленими послідовностями рядків, наприклад, з префіксними деревами, або великими попередньо обробленими словниками, що дозволяє не обмежувати швидкодію на кількості рядків.

Стандартним алгоритмом для розрахунку відстані Левенштейна, є алгоритм розроблений Робертом Вагнером та Міхаелем Фішером [8]. Цей алгоритм використовує підхід відомий як динамічне програмування для знаходження найкоротшого шляху редагування, необхідного для перетворення одного рядка в інший.

Також алгоритм може бути легко модифікований для інших редагувальних відстаней, або щоб дозволити обраховувати відстань Левенштейна, де ціна кожної операції може бути різною, наприклад, відстань між словами що мають схожі, або ті, що знаходяться поряд на клавіатурі, символи буде менша ніж відстань між словами, де відбувається заміна не пов’язаних між собою символів.

Альтернативою алгоритмів на основі динамічного програмування є підходи, що використовують скінченні автомати, що приймають всі рядки які відрізняються від заданого шаблону не більше ніж на задану відстань [1].

Одним з доволі цікавих підходів для пошуку рядків в тексті на основі відстані Левенштейна, теорія якого також базується на скінченних автоматах, є алгоритм під назвою Bitap [9]. Проте застосування обмежується тільки онлайн пошуком та менш гнучке до модифікацій на кшталт перестановки слів або таблиці ціни заміни слів.

1.3 Постановка задачі та вибір способу вирішення

Задача яку ми вирішуємо – існує великий набір текстових даних, що постійно оновлюється та в цілому є змінним. Користувач хоче здійснити пошук слова у цьому наборі файлів. Слово може бути терміном, яке користувач може не знати як правильно писати. Також користувач може допустити помилку при наборі тексту на клавіатурі. Тому у разі якщо не має точного співпадіння слова, необхідно до результатів пошуку додати слова які найбільш схожі на введене слово.

Оскільки ми працюємо з помилками, що виникають при друкуванні тексту на клавіатурі, найдоречніше всього використовувати відстань Дамерау-Левенштейна для оцінки схожості слів. Також ми б хотіли дозволити використання таблиць подібності символів для того, щоб символи, які знаходяться поряд на клавіатурі мали б більшу схожість, та з’являлись в результатах вище, за не пов’язані символи. Також таблиця подібності може бути застосована для unicode символів, щоб користувач міг вводити схожі символи, а отримувати результати з правильними і вони показувались вище. Тому підтримка таблиці подібності є важливою для нашої задачі.

Найпростіший підхід - використання алгоритму для розрахунку відстані Дамерау-Левенштейна напряму. Для кожного слова з текстових даних розраховуємо відстань до введеного користувачем рядку та повертаємо всі слова що мають найменшу відстань. Проте такий підхід не є оптимальним, оскільки нас цікавлять результати з редагувальною відстанню до якогось числа помилок, а ми кожен раз обраховуємо повну відстань. Отже

наша задача - розробити способи, що дозволяють порівнювати рядки між собою швидше ніж простий підхід з розрахунком відстані Дамерау-Левенштейна.

Перспективними є підходи на основі скінченних автоматів, що розпізнають всі слова до якоїсь редагувальної відстані до заданого паттерну. Також вони дозволяють проводити ефективний пошук не тільки для онлайн пошуку, а також для офлайн [11]. Отже, далі ми будемо розглядати програмні реалізації на основі скінченних автоматів.

1.4 Висновки до розділу

В цьому розділі було проведено огляд предметної області, та алгоритмів нечіткого пошуку. Розглянуто існуючі області застосування нечіткого пошуку, такі, як, наприклад, застосування в області обчислювальної біології, комп’ютерного зору, пошукових систем, та автокорекції помилок.

В розділі також були розглянуті існуючі підходи до нечіткого пошуку. Було розглянуто різні методи та алгоритми засновані на поняті редагувальної відстані.

Також в цьому розділі було приділено увагу постановці задачі яку ми будемо подалі вирішувати. Наша задача полягає в нечіткому пошуці в великому обсязі текстових даних, а саме в реалізації рішення, що буде порівнювати рядки швидше ніж простий підхід з розрахунком відстані Дамерау-Левенштейна.

2. ПРОГРАМНА РЕАЛІЗАЦІЯ РІШЕНЬ НА ОСНОВІ АВТОМАТУ ЛЕВЕНШТЕЙНА

У цьому розділі ми розробимо рішення на основі автомату Левенштейна, що можуть бути використані для нечіткого пошуку. Почнемо з більш детального розгляду теорії автоматів.

2.1 Теорія скінченних автоматів

2.1.1 Визначення

Згідно до [12], скінченним автоматом M, називають кортеж з 5 елементів: $\langle Q, q_0, A, \Sigma, \delta \rangle$, де

- Q – скінченна множина станів
- $q_0 \in Q$ – початковий стан
- $A \subset Q$ – множина приймаючих станів
- Σ – скінченний вхідний алфавіт
- δ – відношення $Q \times \Sigma \times Q$, що називається відношенням переходів M

Скінченний автомат називається детермінованим (ДСА), якщо δ – функція з $Q \times \Sigma$ в Q. Інакше, він називається недетермінованим (НСА).

Скінченний автомат починає свою роботу в стані q_0 і зчитує символи вхідного рядка один за одним. Якщо автомат знаходиться в стані q та зчитує вхідний символ a, він робить перехід з стану q в стан $\delta(q, a)$. Якщо поточний стан є елементом A, автомат приймає зчитаний рядок на цьому символі. Рядок, що не прийнятий автоматом, називається відхиленим.

Скінченний автомат може бути зображений графічно у вигляді діаграми станів або таблиці переходів. Діаграма станів надає інтуїтивне представлення станів, переходів і кінечних станів, тоді як таблиця переходів містить детальну інформацію про кожен стан і можливі переходи. На рис 2.1

зображено автомат що приймає рядки які завершуються на непарну кількість символів “a” у вигляді діаграми станів.

Колом позначено стани. Початковий стан – 0. Подвійне коло відповідає за кінцеві стани автомату. Переходи позначені стрілками.

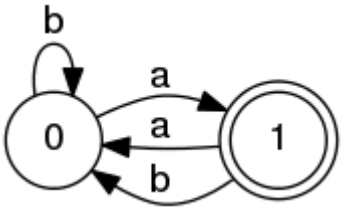

Рисунок 2.1 – Зображення автомату у вигляді діаграми станів

У таблиці 2.1 зображено той самий автомат, але у вигляді таблиці переходів.

Таблиця 2.1 – Зображення автомату у вигляді таблиці переходів

Стан	Символ	
	a	b
0	1	0
1	0	0

Згідно до [2], автоматом Левенштейна для рядка w та максимальної редагувальної відстані d називається автомат, що приймає усі можливі рядки чия відстань Левенштейна до w не більше ніж d.

2.1.2 Різниця між ДСА та НСА

Основна відмінність між ДСА і НСА полягає в поведінці переходу. ДСА мають унікальний перехід для кожного вхідного символу, тоді як НСА можуть мати кілька переходів або ε-переходів для одного вхідного символу. ε-переходами називаються переходи, що зчитують пустий рядок, тобто переходячи до стану, що має ε-переходи, автомат опиняється одночасно і в станах в які ведуть ці переходи.

Автомат зображений на рис. 2.1 є детермінованим, оскільки для кожного стану та символу має унікальний перехід, а отже має тільки єдиний поточний стан.

Хоча ДСА і НСА відрізняються поведінкою переходу, вони еквівалентні з точки зору розпізнавання мови. Будь-яка мова, що приймається НСА, також може бути прийнята ДСА, і навпаки.

НСА часто використовуються в теоретичних обговореннях і побудові регулярних виразів, тоді як ДСА використовуються в практичних реалізаціях завдяки їх детермінізму та ефективності.

Перетворення з НСА на еквівалентний ДСА можливе за допомогою алгоритмів детермінізації НСА.

2.2 Програмна реалізація

Далі буде наведено три програмні реалізації рішень на основі автомату Левенштейна, описано їх внутрішню роботу та різницю між ними. Реалізації були розроблені на мові програмування C++. Кожну з реалізацій можна розділити на два основні етапи - побудова автомату, та перевірка слів за допомогою нього.

З повним лістингом програмного коду для кожного з рішень можна ознайомитись у додатку А.

2.2.1 TreeAutomaton

Побудова починається з побудови не детерміністичного автомата який має вигляд дерева. Щоб побудувати автомат який приймав би слова що відрізняються від шаблону не більше ніж на задану відстань ми перебираємо усі можливі варіанти операцій над шаблоном, сумарна ціна яких є меншою за максимально дозволену. Лістинг алгоритму побудови НСА можна побачити нижче. Алгоритм працює наступним чином:

1. Поки черга не порожня, алгоритм продовжує вибирати перший елемент з черги та обробляти його. Кожен елемент з черги відповідає за комбінацію поточного символу(індекс) з шаблону, та редагувальної відстані що ще може бути використана
2. Якщо індекс стану дорівнює довжині слова, алгоритм перетворює поточний вузол на кінцевий стан.
3. За умови, що сумарний бал плюс вартість вставки не перевищує максимальну допустиму відстань, алгоритм створює новий вузол в дереві з символом вставки та додає новий стан до черги.
4. Алгоритм спробує створити нові стани з різними діями: вставка, видалення, транспозиція та заміна символів. Кожний новий стан, який підходить за умови, що він не перевищує максимальну допустиму відстань, додається до черги для подальшої обробки.

На рис 2.2.2 можна побачити приклад та результат побудови НСА для шаблону “ab” з максимальною редагувальною відстанню 1. Колом позначено стани (вузли дерева). Подвійне коло відповідає за кінцеві стани автомату. Символом “?” позначено будь-який символ, включаючи “a” та “b”. Початковий стан - 0.

Такий автомат приймає будь які слова, що мають редагувальну відстань до шаблону “ab” рівну 1. Наприклад, розглянемо вхідне слово “ba”. Воно має відстань 1, оскільки однієї транспозиції достатньо щоб перетворити його на шаблон “ab”. При зчитуванні першого символу “b” ми опиняємось у станах позначених як 10 та 1. Далі, при зчитуванні “a” ми переходимо до станів 11, та 2. Один з них (11), є приймаючим, отже слово прийняте автоматом.

Для обрахунку відстані, при побудові автомату ми зберігаємо в кожному вузлі дерева величину, що дорівнює сумі відстаней попередніх виконаних операцій для того щоб дійти до цього вузла. Це і є редагувальною відстанню послідовності

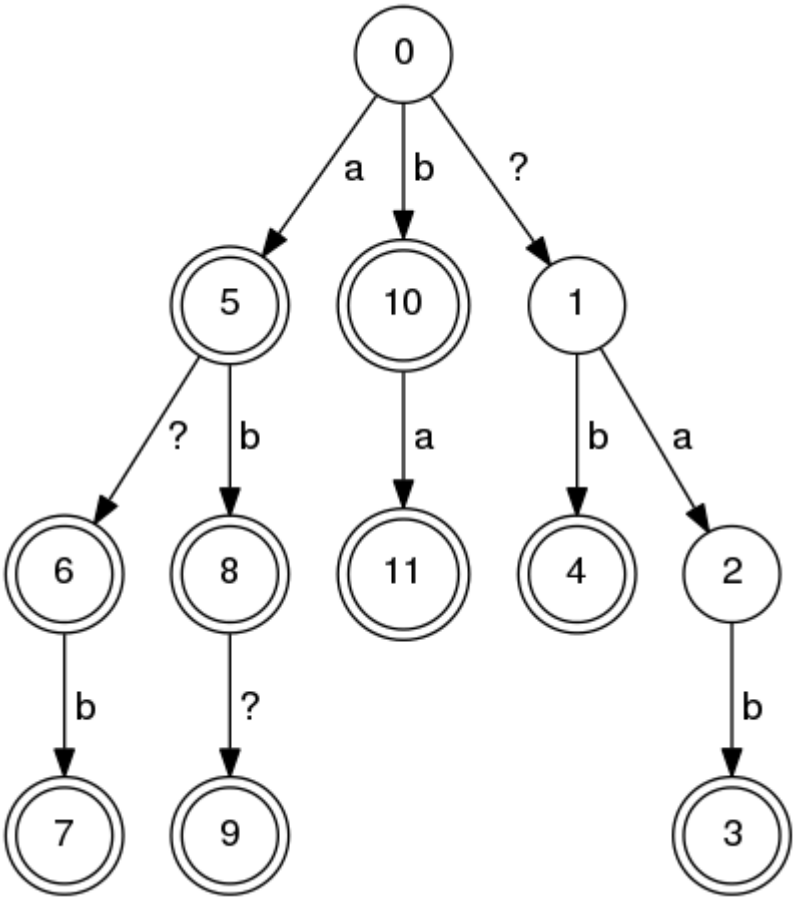

Рисунок 2.2.2 – НСА для шаблону “ab” та максимальної редагувальної відстані 1

Оскільки симуляція НСА є затратним процесом, наступним кроком ми проводимо детермінізацію НСА, для того щоб ми могли швидко перевіряти слова.

Для цього нам потрібно, щоб кожен стан автомату мав тільки один перехід для кожного символу. Для цього ми “об’єднуємо” для кожного стану універсальний перехід з іншими його переходами. Тим самим прибираючи необхідність переходити в два стани одночасно. Нижче можна побачити лістинг алгоритму детермінізації НСА на основі дерева:

Результат детермінізації НСА з рис 2.2.2 можна побачити на рис. 2.2.3. Тепер символ “?” відповідає за усі символи, окрім тих, для яких вже є переходи з цього стану. Наприклад, для стану 0, “?” відповідає за усі символи, окрім “a” та “b”.

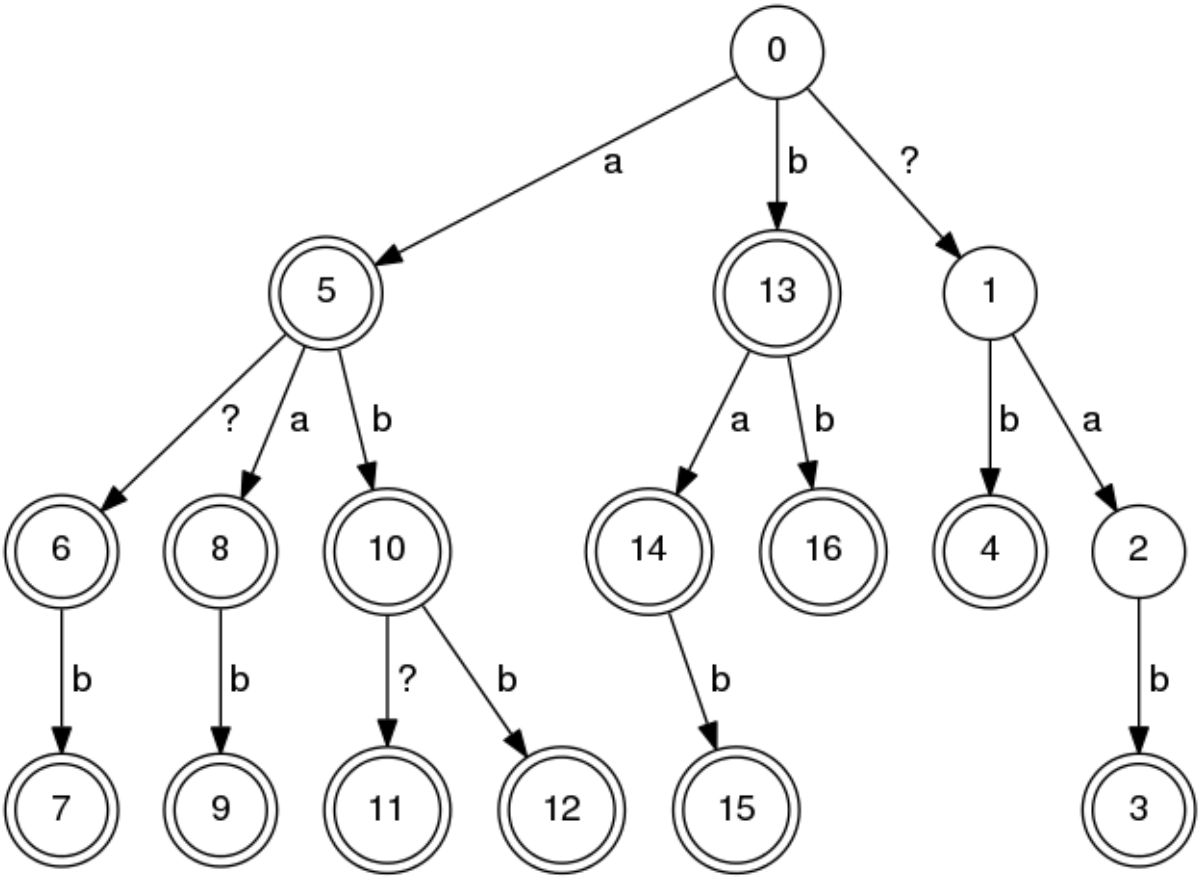

Рисунок 2.2.3 – ДСА для шаблону “ab” та максимальної редагувальної відстані 1

Тепер, при зчитуванні першого символу “b” ми опиняємось тільки у стані 13, після чого, при зчитуванні символу “a” – у стані 14.

Перевірка слів автоматом є дуже простою. Ми проходимо кожен символ вхідного слова паралельно оновлюючи поточний стан у якому знаходимось. У разі якщо натрапляємо на неіснуючий перехід, ітерація завершується, оскільки вхідне слово має редагувальну відстань до шаблону більшу за максимально дозволену, ми повертаємо, що слово не приймається автоматом. Лістинг алгоритму перевірки слова в TreeAutomaton можна побачити нижче:

```
std::pair<bool, float> TreeAutomaton::editDistance(
	std::wstring_view word) const {
	std::reference_wrapper<Node> currentNode = *m_root;
	for (auto c : word) {
	auto nextChild = currentNode.get().children.find(c);
	if (nextChild != currentNode.get().children.end()) {  // exist
		currentNode = nextChild->second;
```

```
        } else {
                auto universalChild =
                                currentNode.get().children.find(m_universalSymbol);
                if (universalChild != currentNode.get().children.end()) {  // exist
                        currentNode = universalChild->second;
                } else {
                        return {false, maxScore};
                }
        }
        }

        return currentNode.get().finalState;
}
```

2.2.2 HashAutomaton

В цьому рішені ми не опираємось на структуру дерева, та робимо припущення, що ціна за кожну операцію, будь то видалення, вставка, транспозиція або заміна є однаковою, що дозволяє нам побудувати більш структурований автомат, що пришвидшує побудову НСА та детермінізацію НСА.

Кожен стан автомату відповідає деякій конфігурації з кількості оброблених символів шаблону та кількості застосованих при цьому операцій редагування. Кожен перехід між станами відповідає якійсь операції. Стани, що повністю обробили шаблон, є фінальними.

Розглянемо НСА більш детально на прикладі шаблону “ab” та максимальної відстанні 1 (рис 2.2.4). Перше число в назві стану відповідає кількості оброблених символів шаблону. Друге число відповідає редагувальній відстані. “*” позначені переходи, що приймають будь який символ, “ε” позначені нульові переходи. Початковий стан – “0 0”. Стан “0*0” відповідає операції транспозиції, що може статись на першому символі.

Розглянемо роботу автомату на прикладі вхідного слова “b”. Так, через нульовий перехід, з самого початку ми опиняємось у станах “0 0” та “1 1”. При зчитуванні символу “b” з стану “0 0” ми переходимо у стани “0*0”, “0 1” та “1 1”, зі стану “1 1” ми переходимо у стан “2 1”. Вхідні символи

закінчились, а отже ми опинились у станах “0*0”, “0 1”, “1 1” та “2 1”. Стан “2 1” є фінальним, а отже автомат приймає слово “b”

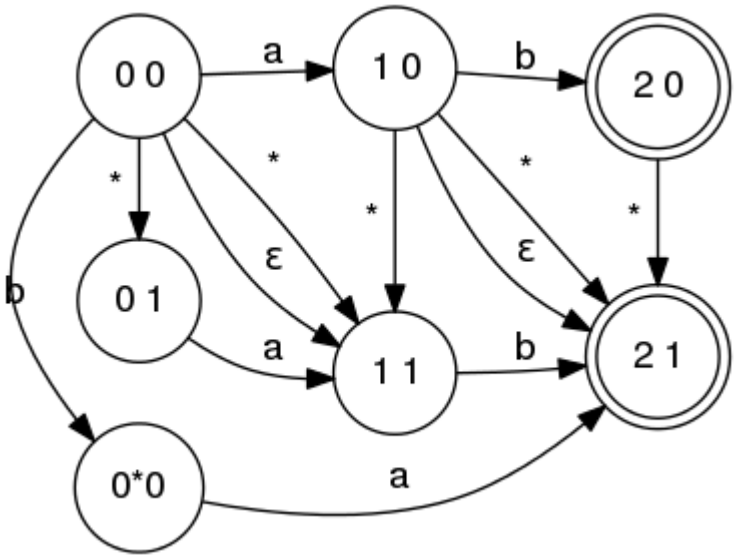

Рисунок 2.2.4 – HCA HashAutomaton для шаблону “ab” та максимальної редагувальної відстані 1

Програмна реалізація побудови такого HCA є доволі простою та зрозумілою. Лістинг алгоритму побудови HCA в HashAutomaton можна побачити нижче. Алгоритм створює стан для кожної можливої кількості помилок та позиції в шаблоні і додає переходи між ними.

Для транспозиції додається новий допоміжний стан, що зчитує спочатку наступний символ, а потім той, що ми розглядаємо в даний момент

```
void HashAutomaton::buildNFA() {
	uint32_t wordSize = m_word.size();

	for (uint32_t i = 0; i < wordSize; ++i) {
		for (uint32_t e = 0; e <= m_maxDistance; ++e) {
			// Correct Character
			State src = {i, e};
			State dest = {i + 1, e};
			addTransition(src, m_word[i], dest);

			if (e < m_maxDistance) {
				// Insertion
				State ins = {i, e + 1};
				addTransition(src, ANY, ins);
```

```
                        // Deletion
                        State del = {i + 1, e + 1};
                        addTransition(src, EPS, del);

                        // Substitution
                        addTransition(src, ANY, del);

                        // Transposition
                        if (i < wordSize - 1) {
                                State transpose = {i + 2, e + 1};
                                State extraState = {i + shift, e + shift};
                                addTransition(src, m_word[i + 1], extraState);
                                addTransition(extraState, m_word[i], transpose);
                        }
                }
        }
    }

    for (uint32_t e = 0; e < m_maxDistance; ++e) {
            State src = {wordSize, e};
            State dest = {wordSize, e + 1};
            addTransition(src, ANY, dest);
    }
}
```

Далі ми будуємо ДСА, який значно зручніший та ефективніший для процесу перевірки слів. Щоб побудувати ДСА, ми проходимо по усім переходам НСА, при цьому створюючи нові стани в ДСА для кожної унікальної комбінації станів НСА, яку ми отримуємо в результаті переходів. Лістинг алгоритму детермінізації можна знайти в додатку Б.

Після завершення детермінізації НСА, автомат готовий до перевірки слів. Перевірка слів доволі схожа на перевірку слів в автоматі на основі префіксного дерева, що ми розглядали до цього. Ітеруємось по кожному символу рядка, що перевіряємо, паралельно оновлюючи поточний стан автомату. Алгоритм перевірки слів в HashAutomaton можна побачити нижче:

```
std::pair<bool, float> HashAutomaton::editDistance(
    std::wstring_view word) const {
    size_t key = 0;
    for (auto c : word) {
    const auto& curVec = m_dfaVec[key];
```

```
        auto index = getAlphabetIndex(m_alphabet, c);
        if (index != -1) {
                auto nextKey = curVec[index];
                if (nextKey != -1) {
                key = nextKey;
                continue;
                }
        }

        auto nextKey = curVec[m_defIndex];
        if (nextKey != -1) {
                key = nextKey;
                continue;
        }

        return {false, maxScore};
        }

        auto curScore = m_finalStates[key];
        if (curScore != maxScore) {
        return {true, curScore};
        }

        return {false, maxScore};
}
```

2.2.3 TableAutomaton

Створимо власну реалізацію засновану на ідеї з [2], модифікуємо її для підтримки операцій транспозицій та unicode символів.

НСА, що лежить в основі TableAutomaton нічим не відрізняється від НСА в HashAutomaton. Тут ми знову робимо припущення, що ціна кожної операції є однаковою для всіх. Основна перевага TableAutomaton – відсутність необхідності явної побудови автомату, маючи шаблон та максимальну редагувальну відстань ми одразу можемо приступити до перевірки слів.

Основним спостереженням, що лежить в основі TableAutomaton є те, що результат детермінізації НСА в HashAutomaton для таких слів як “free” та “tree” буде однаковим, але буде відрізнятись від ДСА для слова “pain”, або “soon”.

Якщо ми перепишемо ці слова, давши кожному унікальному символу свою цифру по порядку, ми побачимо зв’язок між цими словами більш детальніше. Ми отримуємо: free = 1233 = tree, pain = 1234, soon = 1223. Тобто, набір унікальних станів НСА, що можна отримати з одного стану залежить тільки від символу, що зараз перевіряється та входжень цього символу у шаблон починаючи з зсуву на якому цей стан знаходиться.

Для відображення цього факту в [2] вводиться поняття характеристичного вектору, який є бітовою маскою, де значення в позиції дорівнює 1 тільки тоді, коли символ шаблону на цій позиції дорівнює символу який ми перевіряємо. Приклад для шаблону “free” та символу “e” можна побачити на рис. 2.2.5

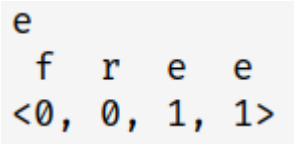

Рисунок 2.2.5 – Характеристичний вектор для шаблону “free” та символу “e”

Проте, для переходу між станами нас цікавлять характеристичні вектори обмеженої довжини, оскільки далекі входження символу не мають впливу на перехід, оскільки редагувальна відстань операцій на які вони мають вплив, є більшою за максимально допустиму. А саме, нас цікавлять тільки характеристичні вектори довжини 2*d + 1, де d - максимальна редагувальна відстань.

Отже, оскільки побудований нами детерміністичний скінченний автомат залежить лише від максимальної редагувальної відстані та усіх можливих значень характеристичного вектору, який має обмежену довжину, ми можемо наперед розрахувати всі можливі переходи та усі можливі стани для будь якого шаблону. Що ми й робимо на початку запуску самої програми.

Алгоритм схожий на детермінізацію НСА, але замість символів шаблону ми використовуємо характеристичні вектори, та зберігаємо отримані при цьому унікальні конфігурації станів у таблицю переходів.

Обчислення нових станів в залежності від характеристичного вектору відбувається за алгоритмом нижче. В ньому ми для стану ДСА обчислюємо нові стани НСА що містяться в ньому після переходу. Після чого ми видаляємо зайві стани, які дублюються, або ведуть до не оптимального рішення. У разі якщо новий набір станів НСА є унікальним, ми додаємо його до таблиці як новий стан ДСА.

```
constexpr State next(const CharacteristicValue& ch, int d) const {
	State result{};
	int min_i = minimal_boundary().i;

	for (const auto& position : positions) {
		int ch_offset = position.i - min_i;
		int k = d - position.e + 1;

		CharacteristicValue new_ch = (ch >> ch_offset) & ((1 << k) - 1);

		for (const auto& new_position :
			elementary_transition(position, new_ch, d)) {
			result.positions.push_back(new_position);
		}
	}

	result.clean();

	return result;
}
```

Функція `elementary_transition` розраховує для стану НСА набір станів НСА до яких можна потрапити з поточного за наданим характеристичним вектором.

Побудова таблиці дозволяє нам повністю уникнути необхідність побудови детерміністичного скінченного автомату для будь якого заданого шаблону. Проте варто зауважити, що такий підхід є практичним тільки для невеликих значень максимальної редагувальної відстані, оскільки розмір

таблиці росте експоненційно з її значенням, оскільки існує 2^{2d+1} унікальних значень характеристичного вектора, де d – максимальна відстань.

Після того як таблиця всіх можливих переходів побудована ми можемо перейти до перевірки слів. Для цього для кожного символу обчислюється характеристичний вектор до заданого шаблону та в залежності від його значення та поточного стану обирається наступний стан автомата з таблиці всіх переходів. По закінченню всіх вхідних символів ми перевіряємо чи є стан фінальним. Лістинг алгоритму можна побачити нижче:

```
std::pair<bool, float> TableAutomaton::editDistance(
	std::wstring_view pattern) const {
	StateIndex current = 0;
	int offset = 0;

	const auto& precomputed = precomputed_tables[m_maxDistance];

	for (const auto& c : pattern) {
	auto ch = characteristic(m_word, c, offset, m_maxDistance);
	const auto& [shift, new_state] =
		precomputed.transitions_table.at(current, ch);

	if (new_state == precomputed.final) {
		return {false, 0};
	}

	offset += shift;
	current = new_state;
	}

	return is_accepting(precomputed.index[current], m_word.size(),
			m_maxDistance, offset);
}
```

2.3 Висновки до розділу

В цьому розділі ми ознайомились з теорію автоматів та розглянули програмну реалізацію трьох різних рішень нашої задачі на її основі. Три представлених рішення, TreeAutomaton, HashAutomaton та TableAutomaton, дозволяють нам ефективно перевіряти подібність слів до заданого шаблону, що вирішує поставлену перед нами задачу. Кожне з рішень має різний підхід.

TreeAutomaton є найбільш загальним рішенням, скінченний автомат якого має вигляд дерева та дозволяє мати різну ціну за кожну операцію редагування.

HashAutomaton робить припущення про рівність ціни за кожну операцію, що дозволяє йому побудувати більш структурований та простий скінченний автомат. Проте детермінізація цього автомату є більш складною за детермінізацію в TreeAutomaton.

TableAutomaton працює з автоматом схожим до HashAutomaton. Проте будує наперед таблицю всіх можливих переходів для малих редагувальних відстаней на основі входжень символу до шаблону, що дозволяє йому пропустити етап побудови автомату для шаблону що перевіряється.

3. ПЕРЕВІРКА ТА АНАЛІЗ РІШЕНЬ

В цьому розділі буде проведено перевірку рішень на коректність, розроблено та проаналізовано тести продуктивності. Ми почнемо з перевірки розроблених рішень на коректність порівняно з простим підходом. Після чого перейдемо до тестів продуктивності та проведемо аналіз їх результатів, на основі чого зробимо висновки про ефективність та практичність кожного з розроблених рішень.

Лістинг програмного коду, що був використаний для проведення тестування можна знайти в додатку Б.

3.1 Перевірка на коректність

Для перевірки рішень на коректність розробимо програму, що буде використовувати словник слів та порівнювати редагувальну відстань обчислену одним із рішень до редагувальної відстані обчисленої за допомогою алгоритму Дамерау-Левенштейна.

Розроблена нами програма перевірки коректності рішень виглядає наступним чином:

```
template <typename T>
void testCalculateScore(const std::vector<std::wstring>& wordsVector) {
	std::random_device rd;
	std::mt19937 gen(rd());
	std::uniform_int_distribution<> randomIndex(0, wordsVector.size() - 1);
	std::uniform_int_distribution<> randomError(0, 3);

	const int searchPhrasesAmount = 100;

	for (int i = 0; i < searchPhrasesAmount; i++) {
	int searchIndex = randomIndex(gen);
	const float maxAllowedError = randomError(gen);
	std::wstring searchPhrase = wordsVector[searchIndex];
	T automaton(searchPhrase, maxAllowedError);

	for (const auto& word : wordsVector) {
		const auto& dist1 = editDistance(searchPhrase, word);
		const auto& dist2 = automaton.editDistance(word);
```

```
            if (dist1 <= maxAllowedError) {
            assert(dist1 == dist2.second);
            } else {
            assert(dist2.first == false);
            }
        }
        }
}

void correctnessTest() {
        std::ifstream file("words.txt");

        std::vector<std::wstring> wordsVector;
        std::string word;
        while (file >> word) {
        wordsVector.emplace_back(string2_w(word));
        }

        testCalculateScore<tree::TreeAutomaton>(wordsVector);
        testCalculateScore<hash::HashAutomaton>(wordsVector);
        testCalculateScore<TableAutomaton>(wordsVector);
}
```

Принцип роботи простий. Ми зчитуємо словник слів та запускаємо тест для кожного з рішень. Всередині тесту ми випадковим чином обираємо редагувальну відстань, шаблон та слова для перевіки. Далі ми обраховуємо для слів редагувальну відстань за допомогою алгоритму Дамерау-Левенштейна та порівнюємо її до результату виконання нашого рішення.

У разі якщо отримані результати не співпадають программа завершується та повідомляє про неспівпадіння, що означає, що наше рішення не є корректним. Звичайне завершення нашої програми означає, що програма не знайшла неспівпадінь, а отже наші рішення відпрацювали корректно.

Було успішно проведено запуск даного тесту на словнику, що містить у собі 370 тисяч англійських слів.

3.2 Тест продуктивності

Наступним нашим кроком є написання та аналіз тестів продуктивності для кожного з представлених рішень. Ми проведемо тести що визначають час

побудови автомату та час перевірки слова для кожного з рішень. Ми розрахуємо час для двох варіантів перевірок, для співпадінь та неспівпадінь, тобто слів, що не приймаються автоматами. Також виміряємо максимальне використання пам’яті під час побудови автоматів.

Розробимо та виміряємо час на загальному тесті, що імітує умови нашого завдання, а саме, побудову автомату та перевірку великої кількості рядків і порівняємо з часом який такий тест займає зі звичайним алгоритмом Дамерау-Левенштейна.

Для проведення та створення тестів ми скористались фреймворком google::benchmark[4]. Він звільняє нас від необхідності самому обчислювати затрачений на виконання час та дозволяє нам займатись тільки написанням самих тестів продуктивності.

Розглянемо тести більш детально.

В тесті, в якому вимірюється час та максимальна пам’ять затрачена на створення автомату, ми створюємо автомат для шаблонів різної довжини та максимальної редагувальної відстані.

```
template <class Automaton>
static void BM_AutomatonCreation(benchmark::State& state) {
	int n = state.range(0);
	float maxError = static_cast<float>(state.range(1));

	for (auto _ : state) Automaton automaton(words[n], maxError);
}
```

Для побудови ми використовуємо такий набір слів:

```
const static std::vector<std::wstring> words = {
	L"",
	L"a",
	L"an",
	L"cat",
	L"dogs",
	L"apple",
	L"banana",
	L"compute",
	L"language",
	L"algorithm",
	L"innovation",
```

```
        L"engineering",
        L"intelligence",
        L"complimentary",
        L"infrastructure",
        L"characteristics",
};
```

Для перевірки слів ми створили два окремих тести: перевірка слів, що приймаються автоматом, та перевірка слів, що не приймаються. Це дозволяє нам проаналізувати два можливих окремих випадки перевірки слів та їх час виконання окремо. В тесті ми спочатку завчасно створюємо автомат, що перевіряється, та проводимо перевірку або на слові на якому побудований цей автомат, або на слові що відрізняється від нього

```
template <class Automaton>
static void BM_AutomatonCheckDistance(benchmark::State& state) {
        int n = state.range(0);
        float maxError = static_cast<float>(state.range(1));
        const auto& word = words[n];
        Automaton automaton(word, maxError);

        for (auto _ : state) automaton.editDistance(word);
}

template <class Automaton>
static void BM_AutomatonCheckDistanceNotSame(benchmark::State& state) {
        int n = state.range(0);
        float maxError = static_cast<float>(state.range(1));
        const auto& word = words[n];
        const auto& otherWord = words[n - 1];
        Automaton automaton(word, maxError);

        for (auto _ : state) automaton.editDistance(otherWord);
}
```

Загальний тест проводить побудову автомату та перевірку на великій кількості різних слів, що зчитуються з великого словника з 370000 англійських слів. Також до загального тесту додано реалізацію на основі алгоритм обчислення відстані Дамерау-Левенштейна, що дозволяє нам в подальшому порівняти наші підходи до тривіального підходу з розрахунком відстані.

Отже, переходимо до запуску тестів. Згідно до [5], для коректного виконання тестів на продуктивність необхідно впевнитись у виконанні наступних умов:

1. Виконувати тест продуктивності багато разів щоб зменшити вплив шуму на його результати
2. Максимально зменшити навантаження на машині на якій запускаються тести. Завершити виконання багатьох процесів в операційній системі перед запуском
3. Вимкнути варіативність швидкості центрального процесору

Тести виконано на наступному обладнанні:

Центральний процесор:

```
Architecture:           x86_64
  CPU op-mode(s):       32-bit, 64-bit
  Address sizes:        48 bits physical, 48 bits virtual
  Byte Order:           Little Endian
CPU(s):                 12
  On-line CPU(s) list:   0-11
Vendor ID:              AuthenticAMD
  Model name:           AMD Ryzen 5 4600H with Radeon Graphics
      CPU family:       23
      Model:            96
      Thread(s) per core:   2
      Core(s) per socket:   6
      Socket(s):        1
      Stepping:         1
      CPU max MHz:      3000.0000
      CPU min MHz:      1400.0000
      BogoMIPS:         5990.01
Caches (sum of all):
  L1d:                  192 KiB (6 instances)
  L1i:                  192 KiB (6 instances)
  L2:                   3 MiB (6 instances)
  L3:                   8 MiB (2 instances)
```

Операційна система: Linux

Пам’ять:

```
Size: 8x2 GB
Type: DDR4
Speed: 3200 MT/s
```

3.2.1 Побудова автомату

Вимірювання часу та пам’яті на побудову були проведені тільки для TreeAutomaton та HashAutomaton, оскільки TableAutomaton не потребує побудови. На рис. 3.1 зображено графіки залежності часу побудови від довжини шаблону для різних значень максимальної редагувальної відстані. Для TreeAutomaton вдалось побудувати тільки автомати для відстані до 3, оскільки більші відстані потребують довгої побудови та займають багато пам’яті.

На рис. 3.2 зображено графіки залежності максимального об’єму використаної пам’яті від довжини шаблону. З збільшенням максимальної редагувальної відстані необхідний об’єм пам’яті зростає. Час та пам’ять для TreeAutomaton зростають дуже швидко.

На рис. 3.3 можна побачити графіки що порівнюють час та пам’ять для TreeAutomaton та HashAutomaton між собою.

Для кожного типу автомата розмір шаблону змінюється від 1 до 15, а максимальна відстань редагування лежить в межах до 6. Час (у мілісекундах) для створення автомата збільшується зі збільшенням розміру шаблону та максимальної відстані редагування. Це має сенс, оскільки більші шаблони та вищі відстані редагування вимагають більше обчислень.

Порівнюючи, HashAutomaton потребує меньше часу для створення порівняно з TreeAutomaton для того самого розміру шаблону та відстані редагування. Це пов’язано з основною складністю їх структур автоматів і способом їх реалізації.

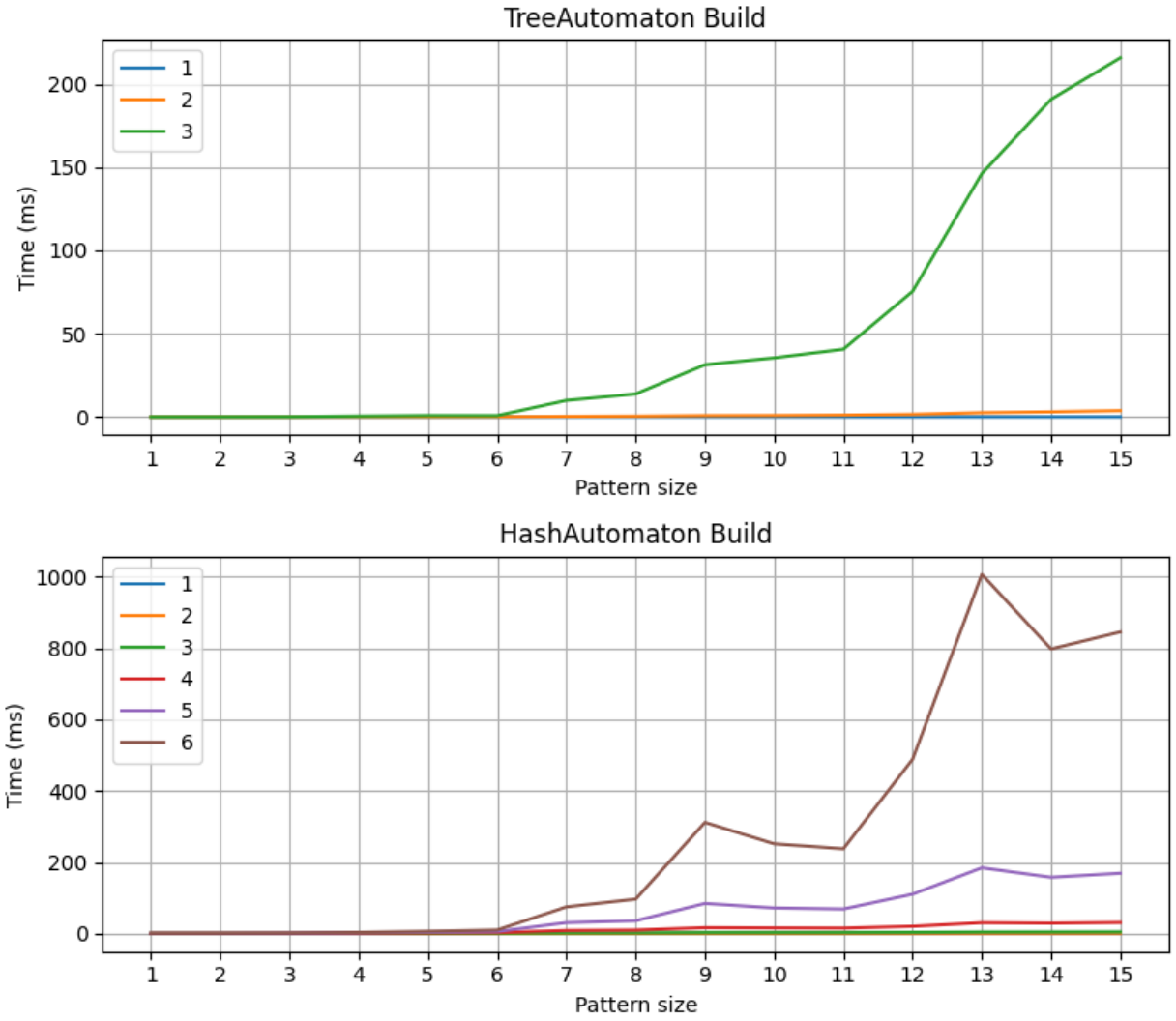

Рисунок 3.1 – Графіки залежності часу побудови від довжини шаблону

Є кілька цікавих відхилень, наприклад, перехід від розміру шаблону 6 до розміру шаблону 7 є доволі швидким. А у випадку HashAutomaton із розміром шаблону 9 до 11, час залишається досить постійним. Це може бути пов’язано з певними характеристиками алгоритму чи даних. Дійсно, використані шаблони для розмірів 6 та 7 виглядають так: "banana", "compute". Слово “banana” має більшу кількість символів, що повторюються ніж слово “compute”, це може впливати на процес детермінізації автомату та кінцеву кількість станів та переходів між ними.

Слова, що використані для шаблонів розмірами 9, 10, 11 виглядають так: "algorithm", "innovation", "engineering". Вони також містять в собі

декілька однакових символів та подвоєнь, чим пояснюється відхилення від зростання часу побудови автомату з збільшенням розміру шаблону.

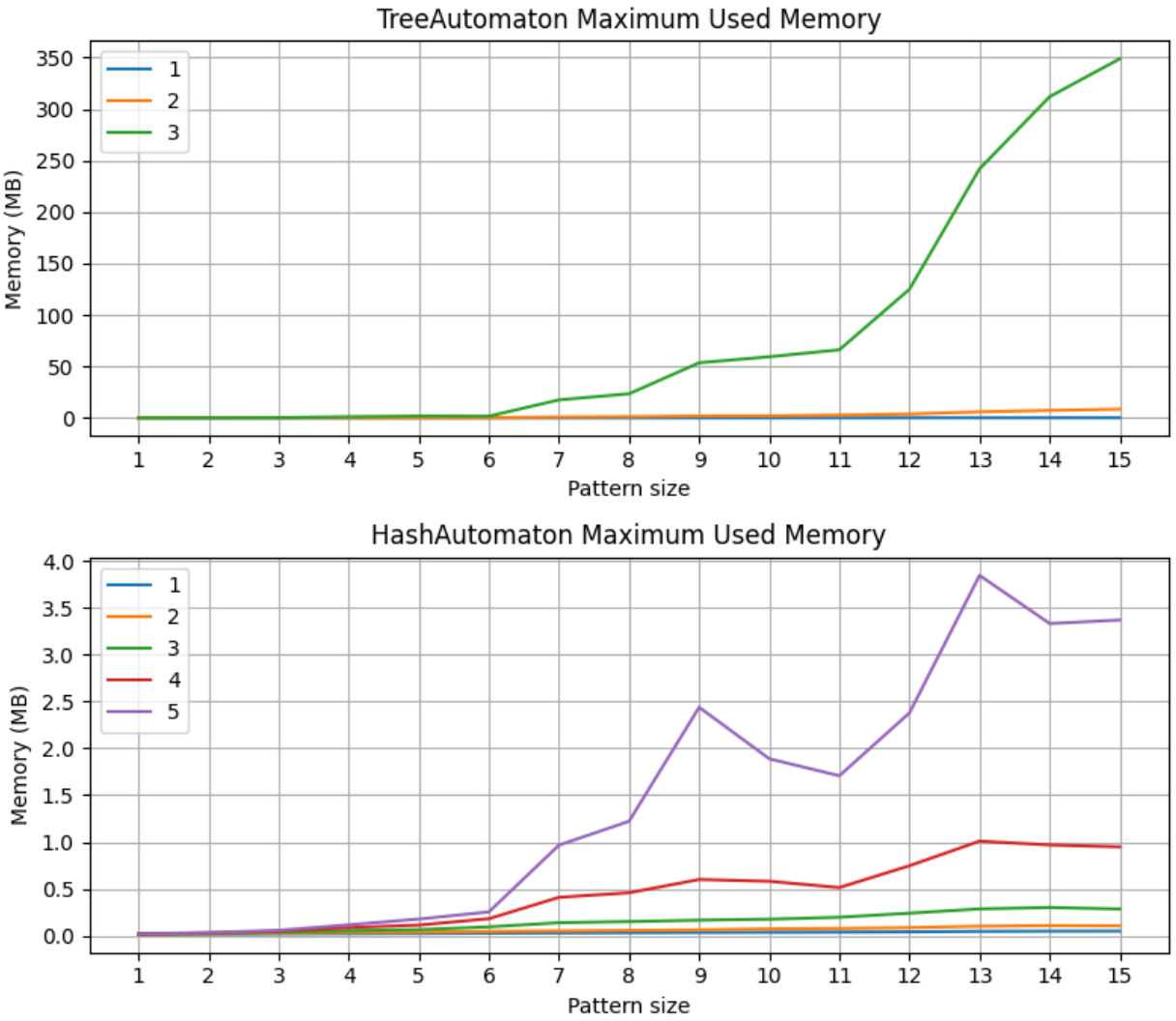

Рисунок 3.2 – Графіки залежності максимального об’єму використаної пам’яті під час побудови від довжини шаблону

Схожі результати маємо і при порівнянні об’єму використаної пам’яті для двох рішень. HashAutomaton використовує значно менше пам’яті при побудові автомата ніж TreeAutomaton, це пов’язано з більш структурованим NCA що в результаті дає меншу кількість станів ДСА. Також в TreeAutomaton використана структура даних дерева, що потребує більшу кількість виділень пам’яті

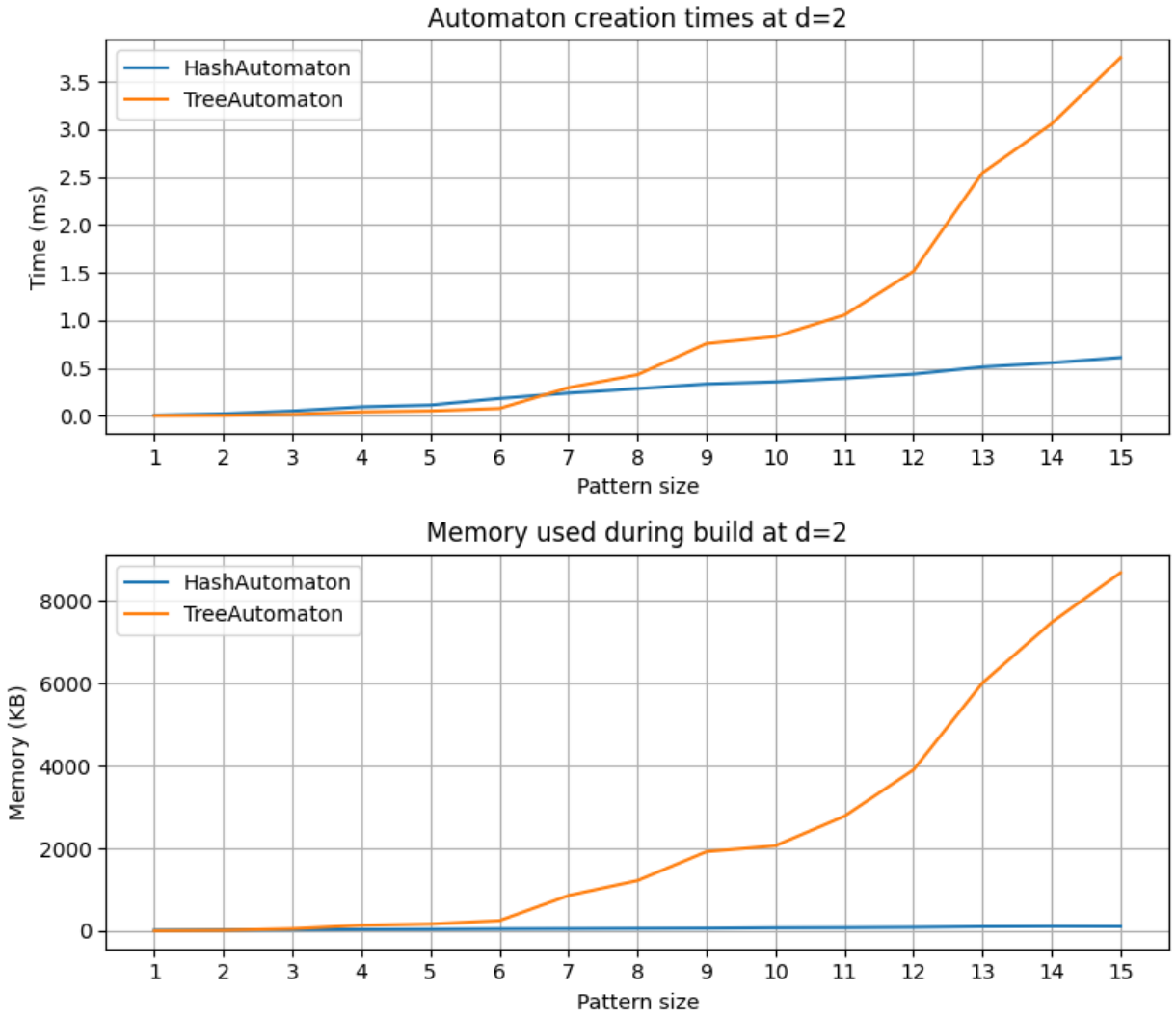

Рисунок 3.3 – Графіки порівняння побудови TreeAutomaton та HashAutomaton для максимальної редагувальної відстані 2

3.2.2 Перевірка слова

На рис. 3.4 можна побачити графіки, що відображають час перевірки слова. Загально можна побачити, що час перевірки для слів що приймаються автоматом є більшим ніж час перевірки слів, що не приймаються. Це пов’язано з тим, що з зчитуванням послідовності, що не приймається, автомат може припинити виконання на самому початку не зчитуючи послідовність до кінця.

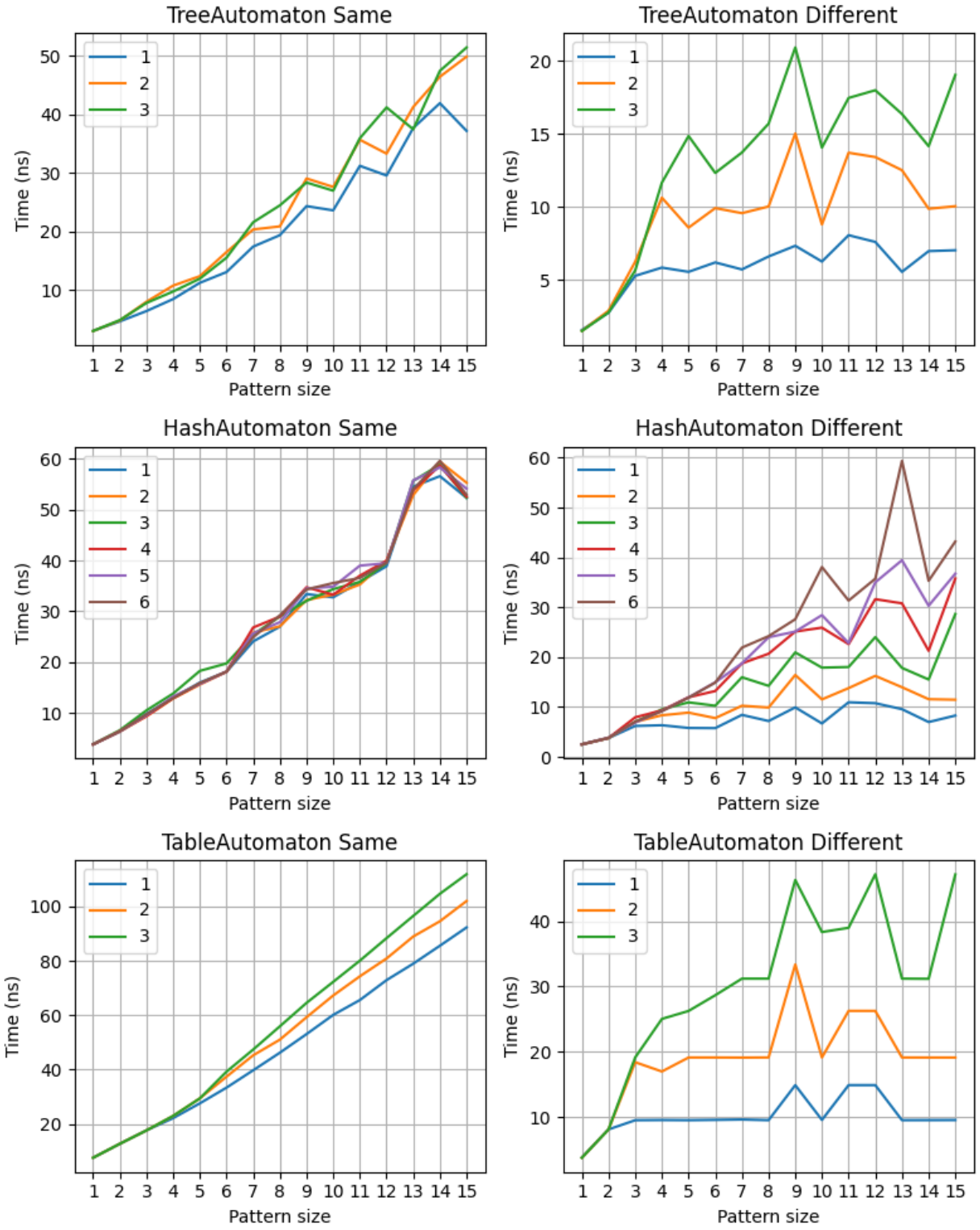

Рисунок 3.4 – Графіки залежності часу перевірки слова від довжини шаблону

Зі збільшенням розміру шаблону час також має тенденцію до збільшення. На час для TreeAutomaton та TableAutomaton помітно впливає максимальна відстань редагування. В першому випадку це пов’язано з

використанням структури даних асоціативного контейнеру на основі дерев, що асимптотично має логарифмічний час пошуку, проте при малій кількості елементів працює швидше ніж аналогічний контейнер в мові C++ на основі хешування[6]. В другому випадку це пов’язано з необхідністю розраховувати характеристичний вектор для кожного символу рядка.

На рис. 3.5 зображено графіки, що порівнюють час виконання рішень для максимальної редагувальної відстані 2. Також до графіку з перевіркою однакових до шаблону слів додано час виконання звичайного алгоритму Дамерау-Левенштейна.

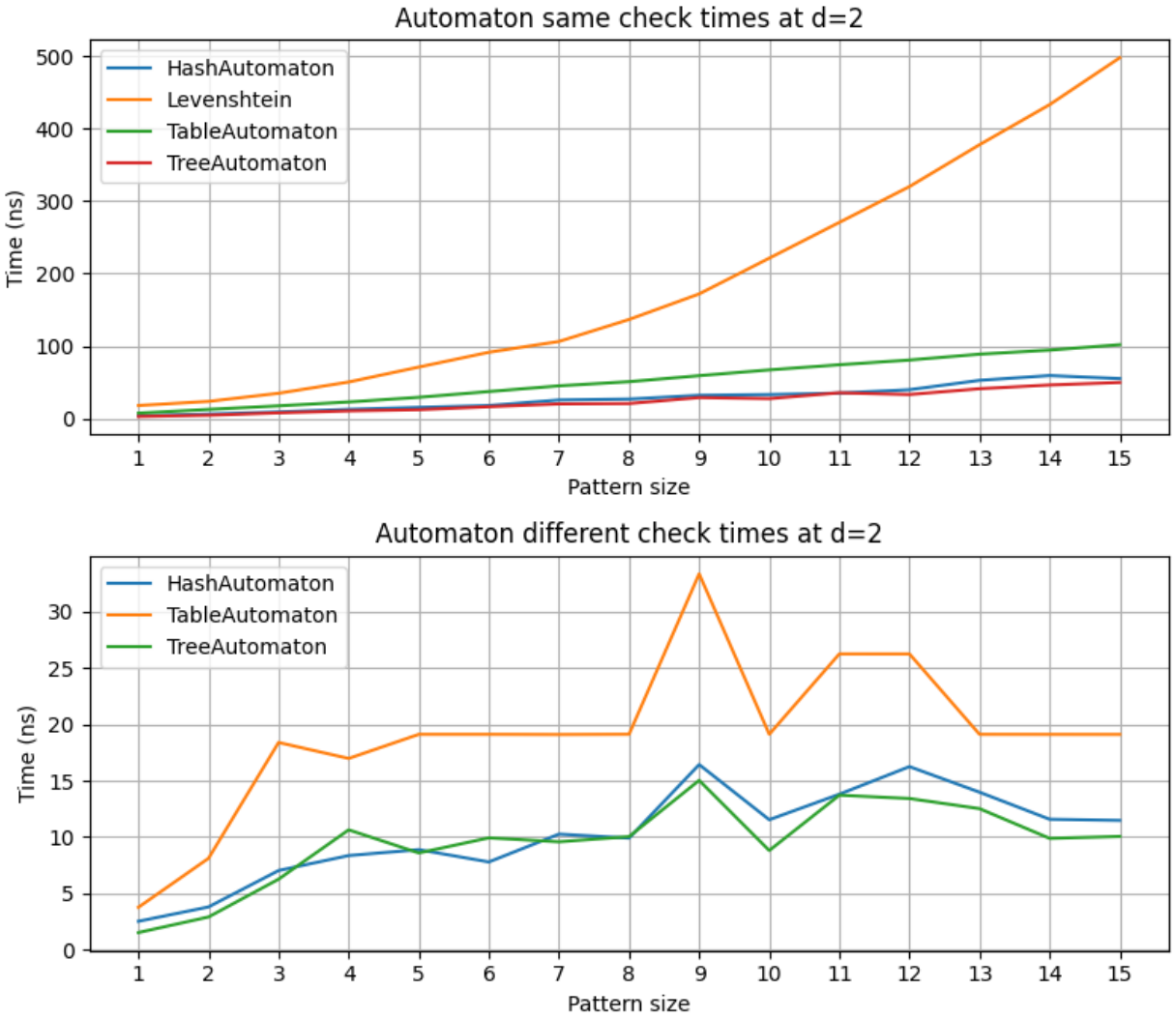

Рисунок 3.5 – Порівняння часу перевірки слова для редагувальної відстані 2

Дивлячись на графік, стає очевидним, що TreeAutomaton і HashAutomaton працюють приблизно однаково, з не значною перевагою TreeAutomaton з точки зору витраченого часу. Проте TableAutomaton, має значно вищі часи за всіма напрямками.

Це може бути пов’язано з тим, що для перевірки слова для кожного символу необхідно розраховувати характеристичний вектор. Ми можемо спробувати зробити оптимізацію на основі SIMD інструкцій процессора, чим скоротимо час його розрахунку:

```
inline CharacteristicValue characteristic(const std::wstring& word, wchar_t c,
                                          int offset, int d) {
       int w = word.size();
       int k = std::min(2 * d + 1, w - offset);
       CharacteristicValue result = 0;

       __m256i chunk = _mm256_loadu_si256((__m256i*)(word.c_str() + offset));
       __m256i compareResult = _mm256_cmpeq_epi32(chunk, _mm256_set1_epi32(c));
       result = _mm256_movemask_ps((__m256)compareResult);
       result = result & ((1 << k) - 1);

       return result;
}
```

Результати можна побачити на рис. 3.6. Перевірка рядків що приймаються автоматом тепер не залежить від редагувальної відстані, це сталось оскільки ми розраховуємо характеристичний вектор за декілька інструкцій процесора. При цьому перевірка слів, що не приймаються залежить від редагувальної відстані через те, що з її збільшенням має пройти більше станів перш ніж слово відкидається повністю. Бачимо пришвидшення для слів, що не схожі на шаблон аж в два рази, до значень часу близьких до TreeAutomaton

Проте треба мати на увазі, що ця оптимізація працює тільки якщо процессор на якому це запускається підтримує використані нами інструкції. Також ця оптимізація працює корректно тільки для малих значень редагувальної відстані.

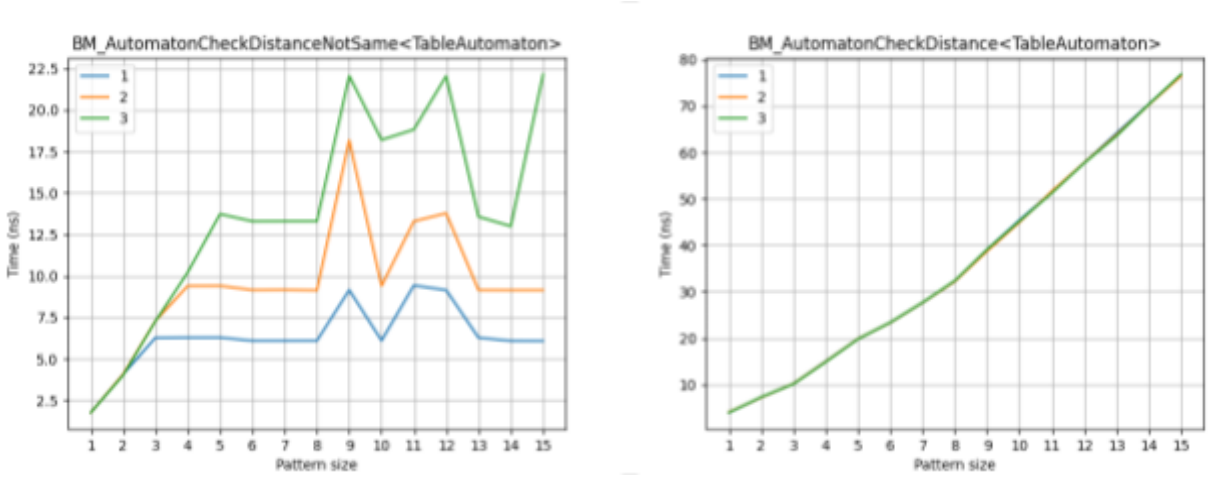

Рисунок 3.6 – Результат SIMD оптимізації для TableAutomaton

3.2.3 Загальний тест

Результат виконання загального тесту можна побачити на рис. 3.7. Цей тест було проведено на словнику що складається з 370 000 англійських слів.

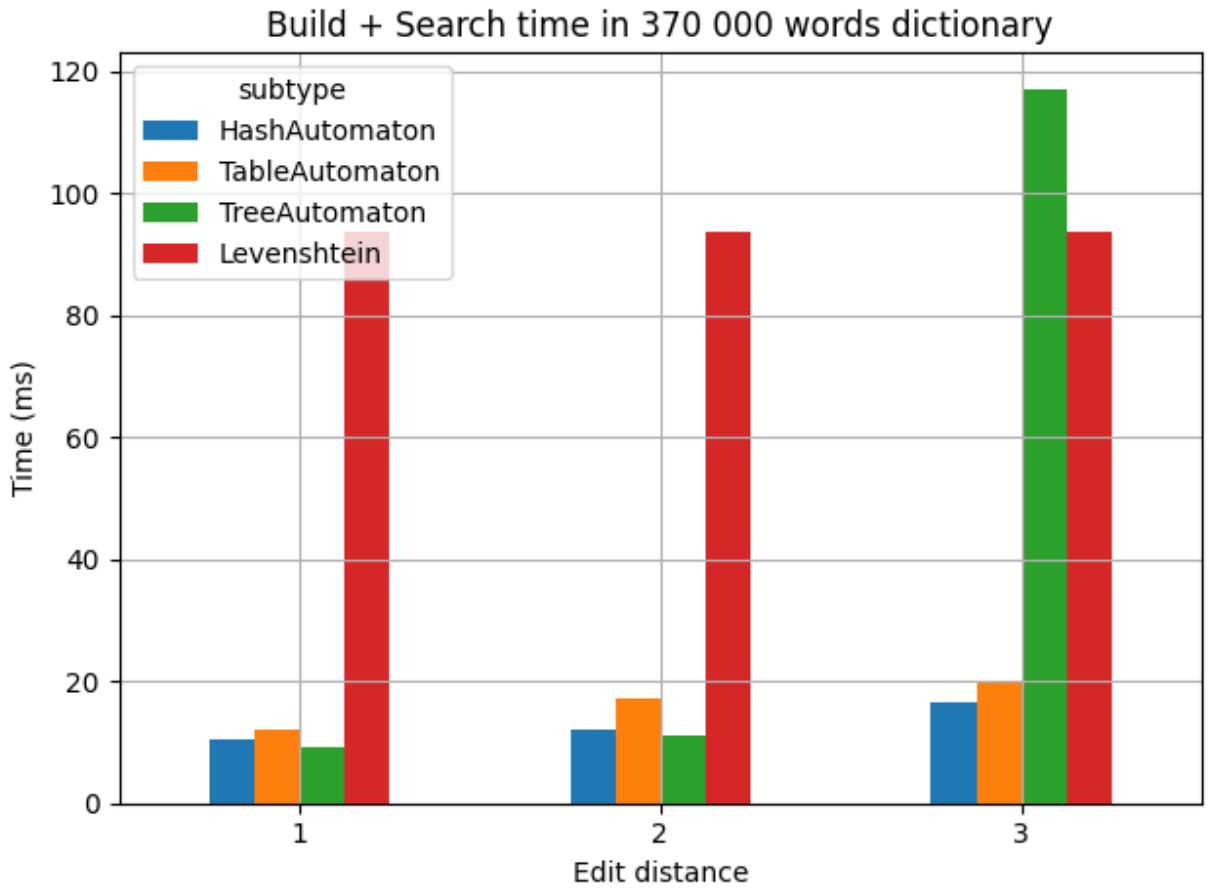

Рисунок 3.7 – Результати загального тесту

Кожне з рішень працює швидше ніж простий підхід з використанням алгоритму Дамерау-Левенштейна. Проте помітно, що для редагувальної відстані 3 час побудови TreeAutomaton починає займати велику кількість часу. Але для відстані 1 та 2 він працює найшвидше за інші рішення. Відсутність необхідності побудови автомата не надає значної переваги TableAutomaton оскільки побудова автоматів відбувається тільки один раз на початку.

3.3 Висновки до розділу

В розділі було проведено перевірку кожного з рішень на коректність. Також проведено та проаналізовано тести продуктивності. Можемо зробити наступні висновки:

1. HashAutomaton та TreeAutomaton демонструють подібну продуктивність часу виконання під час перевірки слів, але HashAutomaton витрачає менше часу та пам’яті на створення, що робить його більш ефективним рішенням для широкого застосування.
2. На відміну від HashAutomaton та TreeAutomaton, TableAutomaton має значно довший час перевірки слів. Проте, при частих змінах пошукового слова, TableAutomaton є ефективним рішенням, оскільки не вимагає побудови для малих значень редагувальної відстані.
3. TreeAutomaton витрачає значно більше часу та пам'яті на побудову, ніж інші рішення. Проте, він дозволяє мати різну ціну для кожної редагувальної операції, що може бути корисним в певних ситуаціях.
4. Усі розроблені рішення працюють швидше, ніж простий підхід з використанням алгоритму Дамерау-Левенштейна, що демонструє їхню ефективність та потенціал для подальшого застосування.
5. Додатково, всі розроблені рішення володіють здатністю швидко відкидати слова, що не відповідають критеріям, що підвищує їх ефективність.

Враховуючи ці висновки, рекомендується використовувати HashAutomaton для загального використання, TreeAutomaton для ситуацій, коли потрібна різна ціна для редагувальних операцій, та TableAutomaton для ситуацій, коли пошукове слово часто змінюється.

4. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

В даному розділі буде проведено оцінювання основних характеристик для майбутнього програмного продукту, що вирішує поставлену нами задачу нечіткого пошуку.

Програмний продукт, що є результатом вирішення задачі, може бути використаний на будь-якому персональному комп’ютері, що базується на будь-якій операційній системі для проведення нечіткого пошуку у купі файлів.

Також в даному дослідженні показано різні варіанти реалізації для забезпечення найбільш коректної та оптимальної стратегії вибору, що має вплив на економічні фактори та сумісність з майбутнім програмним продуктом. Для цього застосовувався апарат функціонально-вартісного аналізу.

Для проведення аналізу використовується економічна, технічна та конструкторська інформація. Метою функціонально-вартісного аналізу є, визначення кращих способів виконання функцій при мінімальних витратах. Цей аналіз дозволяє ідентифікувати критичні функції, які варто зберегти, оптимізувати або замінити, а також виявити можливості зниження витрат без втрати якості продукту.

Алгоритм функціонально-вартісного аналізу включає в себе визначення послідовності етапів розробки продукту, визначення послідовності функцій, які є необхідними для реалізації програмного засобу, а саму – усі можливі функції, які не мають впливу та мають вплив на вартість продукту. Визначення повних витрат (річних) та кількості робочих часів, визначення джерел витрат та кінцевий розрахунок вартості програмного продукту.

4.1 Постановка задачі проектування

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки системи прогнозу стійкості фінансових показників. Оскільки рішення стосовно проектування та реалізації компонентів, що розробляється, впливають на всю систему, кожна окрема підсистема має її задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для нечіткого пошуку у наборі файлів.

Технічні вимоги до програмного продукту є наступні:

- функціонування на персональних комп'ютерах із стандартним набором компонентів;
- зручність та зрозумілість для користувача;
- швидкість обробки даних та доступ до інформації в реальному часі;
- можливість зручного масштабування та обслуговування;
- мінімальні витрати на впровадження програмного продукту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка можливого програмного продукту, яка дозволяє аналізувати різні характеристики, що безпосередньо впливають на стійкість підприємства. Беручи за основу цю функцію, можна виділити наступні:

F_1 – вибір мови програмування.

F_2 – вибір способу представлення результатів.

F_3 – вибір середовища розробки.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

а) Python.

б) C++.

Функція F_2.

а) Інтерфейс командного рядку.

б) Графічний інтерфейс.

Функція F_3:

а) Emacs.

б) Visual Studio Code.

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1).

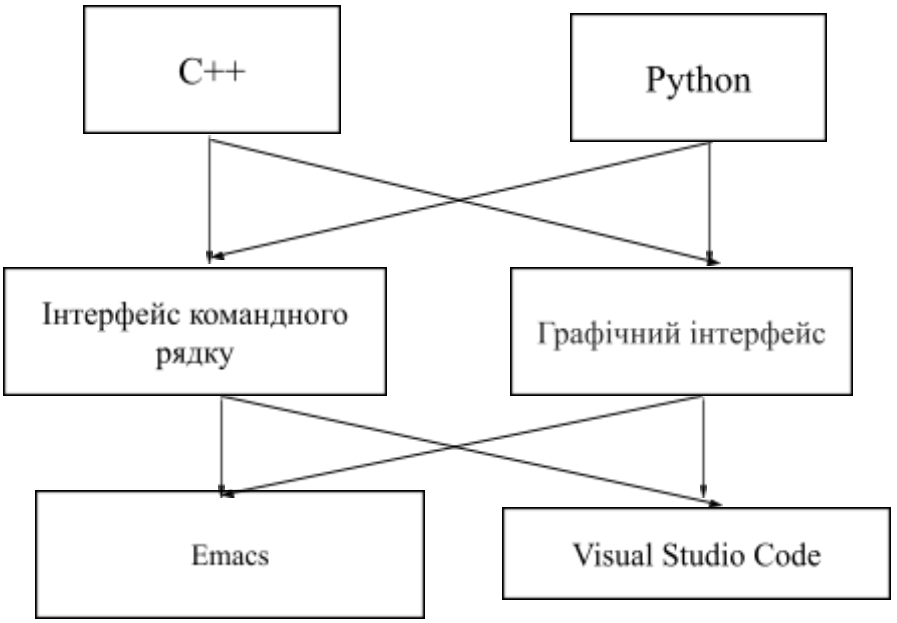

Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображає множину всіх можливих варіантів основних функцій. Позитивно-негативна матриця показана в таблиці 4.1.

Таблиця 4.1 - Позитивно-негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки

F_1	*А*	Швидкодія, великий обсяг бібліотек	На написання коду необхідно мати базові навички та вміння
	Б	Популярна мова, багатий набір вбудованих бібліотек, швидкість розробки	Швидкодія, необхідність встановлення середовища виконання
F_2	*А*	Легкість при написанні	Страждає інтуїтивність використання
	Б	Інтуїтивність при використанні	Складність написання, необхідність використання сторонніх бібліотек
F_3	*А*	Масштабованість, та великий набір розширень, можливість запуску без графічного інтерфейсу	Не популярний, складний для новачків
	Б	Загальновідомий, простий у використанні, кросплатформенний	Не інтуїтивні гарячі клавіші

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F_1:

Перевагу надаємо швидкодії та загальнодоступності. Тому для створення коду з високою швидкодією обираємо варіант А, а варіант Б має бути відкинутий.

Функція F_2:

Жоден з варіантів не суперечить функціоналу програми, а отже допускаємо обрання обох варіантів. Можливо використати варіанти А або Б.

Функція F_3:

Зважаючи на складність у використанні варіанту А, обираємо варіант Б.

Таким чином, будемо розглядати такий варіанти реалізації ПП:

F_1а – F_2а – F_3а

F_1а – F_2б – F_3а

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.3 Обґрунтування системи параметрів програмного продукту

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- *X1* – час виконання програмного забезпечення;
- *X2* – об’єм пам’яті необхідний для роботи програмного забезпечення;
- *X3* – потенційний об’єм програмного коду.

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту, як показано у таблиці 4.2.

Таблиця 4.2 - Основні параметри програмного продукту

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Потенційний час виконання коду	X1	мс	600	200	10
Об’єм пам’яті	X2	МБ	700	200	20
Потенційний об’єм програмного коду	X3	кількість рядків коду	3500	2000	500

За даними таблиці 4.3 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.4.

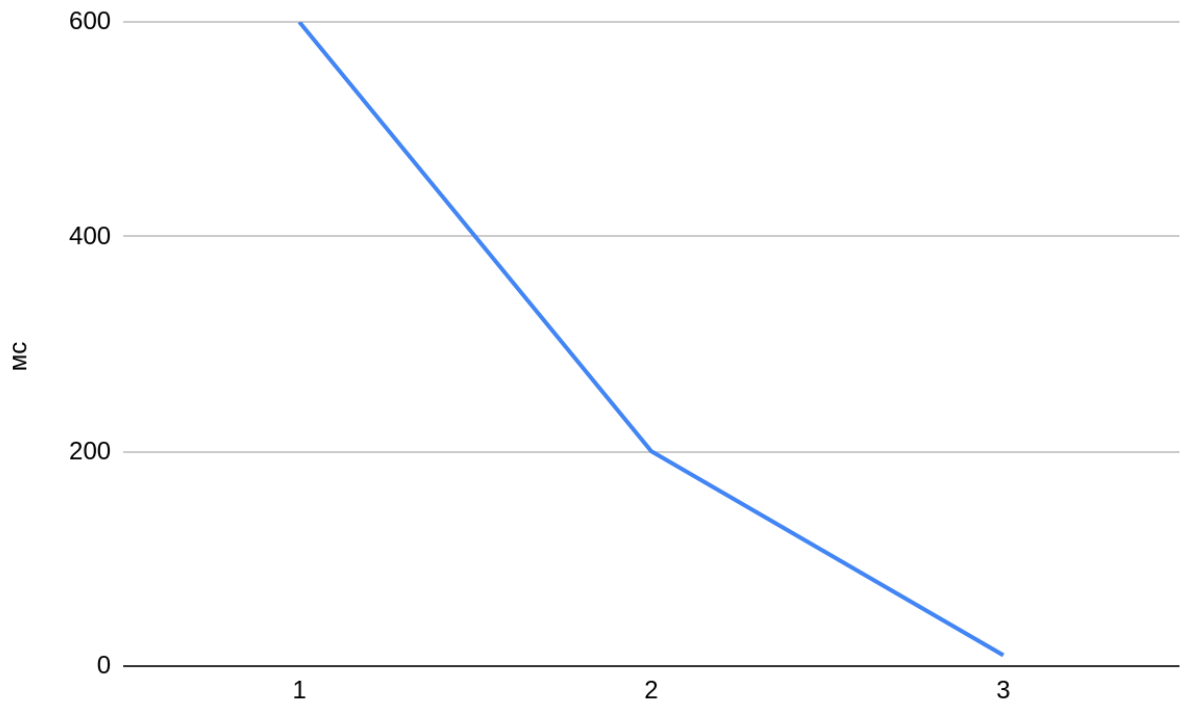

Рисунок 4.2 – X1, швидкодія мови програмування

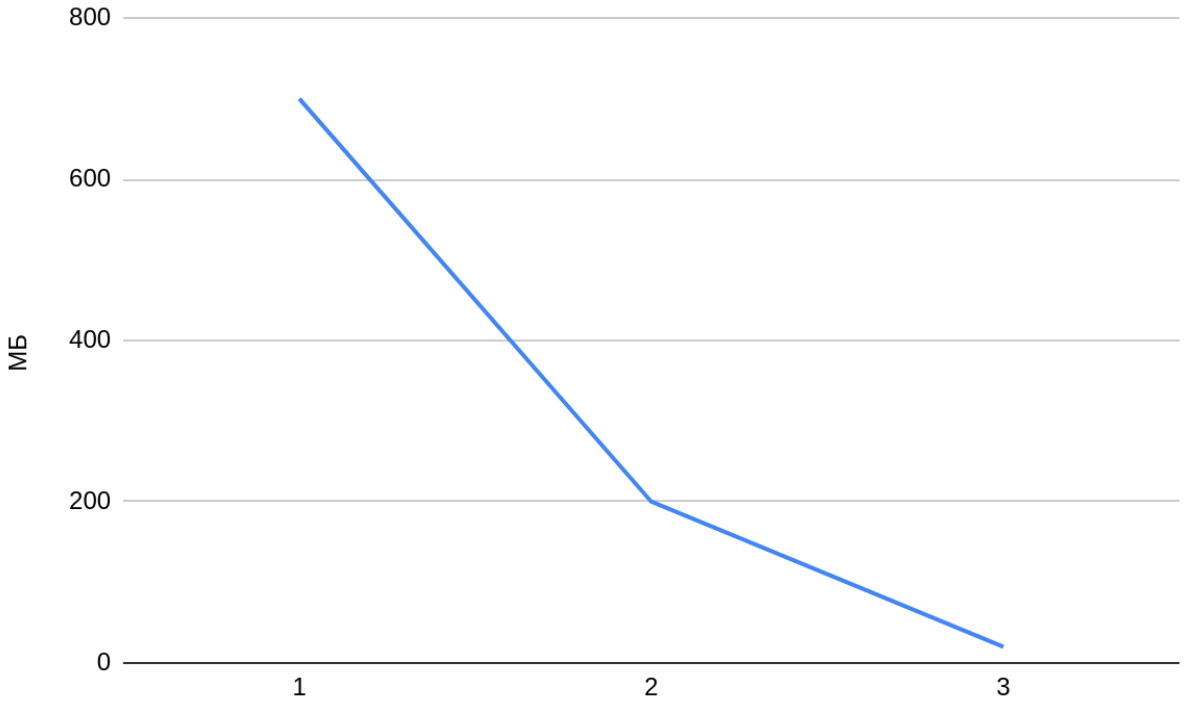

Рисунок 4.3 – X2, об’єм пам’яті

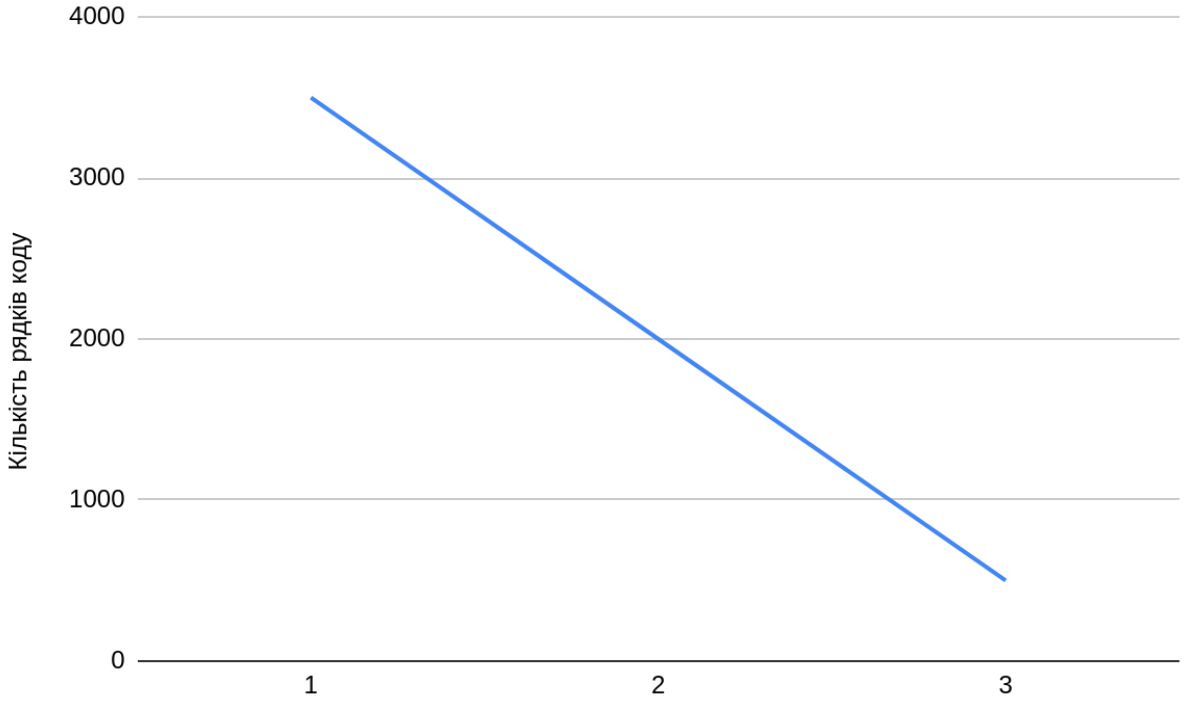

Рисунок 4.4 – X4, потенційний об’єм програмного коду

4.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 5 людей. Визначення коефіцієнтів значущості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Таблиця 4.3 - Результати ранжування параметрів

Позначення параметра	Одиниці виміру	Ранг параметра за оцінкою експерта					Сума рангів R_i	Відхилення Δ_i	Δ_i^2
		1	2	3	4	5			
X1	Оп/мс	3	2	3	2	3	13	3	9
X2	МБ	2	3	2	3	2	12	2	4
X3	Кількість	1	1	1	1	1	5	-5	25
Разом		6	6	6	6	6	30	0	38

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів та загальна сума рангів обчислюється таким чином:

$$R_i = \sum_{j=1}^{N} r_{ij} = 54, \ \#(4.1)$$

де N – число експертів, r_{ij} – ранг і-го параметра, визначений j-м експертом.

б) середня сума рангів, де n – кількість параметрів:

$$T = \frac{1}{n} R_i = 10 \ \#(4.2)$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \ \#(4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 38. \ \#(4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2\left(n^3 - n\right)} = \frac{12 \cdot 38}{5^2\left(3^3 - 3\right)} = 0,76 > W_k = 0,67. \ \#(4.5)$$

Ранжування можна вважати достовірним. Нормативний коефіцієнт узгодженості дорівнює 0,67, а знайдений коефіцієнт узгодженості – 0.76, що перевищує нормативний.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати заносимо у таблицю 4.4.

Таблиця 4.4 - Попарне порівняння параметрів.

Параметри	Експерти					Кінцева оцінка	Числове значення
	1	2	3	4	5		
X1 і X2	>	<	>	<	>	>	1.5
X1 і X3	>	>	>	>	>	>	1.5
X2 і X3	>	>	>	>	>	>	1.5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \{1.5 \text{ при } X_i > X_j \; 1.0 \text{ при } X_i = X_j \; 0.5 \text{ при } X_i < X_j . \#(4.6)$$

З отриманих числових оцінок переваги складемо матрицю $A=\| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \#(4.7)$$

$$b_i = \sum_{i=1}^{N} a_{ij} \#(4.8)$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{ві}} = \frac{b_i^{'}}{\sum_{i=1}^{n} b_i^{'}}, \ \#(4.9)$$

$$b_i^{'} = \sum_{i=1}^{N} a_{ij} b_j \#(4.10)$$

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 - Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j			Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1	1,5	1,5	4	0.44	16	0.55	64	0,64
X2	0,5	1	1,5	3	0.33	9	0,31	27	0,27
X3	0,5	0,5	1	2	0,23	4	0,14	8	0,9
Всього:				9	1	29	1	99	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X2* (Об'єм пам'яті), *X1* (час виконання програми) та *X3* (потенційний об'єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{\text{в}i,j} B_{i,j}, \ \#(4.11)$$

де n – кількість параметрів;

$K_{вi}$ – коефіцієнт вагомості *i*–го параметра;

B_i – оцінка *i*–го параметра в балах.

Таблиця 4.6 - Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Пара метри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	А	X1	10	3	0,64	1.92
F2	А	X2	20	2	0,27	0.81
	Б	X2	200	3	0,27	0.54
F3	А	X3	500	3	0,9	2.7

За даними з таблиці 4.6 за формулою:

$$K_K = K_{\text{ТУ}}\left[F_{1k}\right] + K_{\text{ТУ}}\left[F_{2k}\right] + \ldots + K_{\text{ТУ}}\left[F_{zk}\right], \ \#(4.12)$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 1.92 + 0.81 + 2.7 = 5.43;$$

$$K_{K2} = 1.92 + 0.54 + 2.7 = 5.16 .$$

Як видно з розрахунків, кращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

Третій варіант включає в себе додаткове завдання розробки графічного інтерфейсу.

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б, завдання 3 до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3; завданні 3 до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних, для завдання 3 використовується довідкова інформація.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як:

$$T_О = T_Р \cdot К_П \cdot К_{СК} \cdot К_М \cdot К_{СТ} \cdot К_{СТ.М}, \#(4.13)$$

де $T_Р$ – трудомісткість розробки ПП;

$К_П$ – поправочний коефіцієнт;

$К_{СК}$ – коефіцієнт на складність вхідної інформації;

$К_М$ – коефіцієнт рівня мови програмування;

$К_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$К_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру ступеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_Р$ = 25 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $К_П$ = 1.5. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $К_{СК}$ = 1. Оскільки при

розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ}$ = 0.9. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$Т_1 = 25 \cdot 1.5 \cdot 0.9 = 33.75 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $Т_Р$ = 35 людино-днів, $К_П$ =0.9, $К_{СК}$ = 1, $К_{СТ}$ =0.8:

$$Т_2 = 35 \cdot 1.5 \cdot 0.8 = 42 \text{ людино-днів.}$$

Для третього завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $Т_Р$ = 10 людино-днів, $К_П$ =0.9, $К_{СК}$ = 1, $К_{СТ}$ =0.8:

$$Т_3 = 10 \cdot 1.5 \cdot 0.8 = 12 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість

Загальна трудомісткість дорівнює:

$$Т_I = (33.75 + 42) \cdot 8 = 606 \text{ людино-годин.}$$
$$Т_{II} = (33.75 + 42 + 12) \cdot 8 = 702 \text{ людино-годин.}$$

В розробці беруть участь два програмісти з окладом 16500 грн., один аналітик в області даних з окладом 17000. Визначимо середню зарплату за годину за формулою:

$$СЧ = \frac{М}{T_m \cdot t} грн., \#(4.14)$$

де М – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$СЧ = \frac{16500+16500+17000}{3 \cdot 21 \cdot 8} = 99.20 \text{ грн. } \#(4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$СЗП = С_ч \cdot Т_i \cdot КД, \#(4.16)$$

де $С_ч$– величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 99.20 \cdot 606 \cdot 1.2 = 72142.85$ грн.

II. $С_{ЗП} = 99.20 \cdot 702 \cdot 1.2 = 83566.08$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 72142.85 \cdot 0.22 = 15871.42$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 83566.08 \cdot 0.22 = 18384.53$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 16500 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 16500 \cdot 0.2 = 39600 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 39600 \cdot (1 + 0.2) = 47520 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 47520 \cdot 0,22 = 10454.4 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 30000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.4 \cdot 0.12 \cdot 30000 = 5040 \text{ грн.,}$$

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$– річна норма амортизації;

$Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.4 \cdot 30000 \cdot 0.08 = 3360 \text{ грн.,}$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} =(Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 12 – 16) \cdot 8 \cdot 0.35 =$$

$$= \ 627.2 \ \text{години},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

$К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 627.2 \ \cdot 0.2 \cdot 0.3 \cdot 3.99 = 150.2 \ \text{грн.},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$– коефіцієнтом зайнятості приладу;

$Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 30000 \cdot 0.67 = 20100 \quad \text{грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н, \#(4.17)$$

$$С_{ЕКС} = 47520 + 10454.4 + 5040 + 3360 + 150.2 + 20100 = 86624.6 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{М\text{-}Г} = C_{ЕКС} / T_{ЕФ} = 86624.6 \ / \ 627.2 \ = 138.11 \quad \text{грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, складає:

$$C_М = C_{М-Г} \cdot T, \#(4.18)$$

I. $C_М = 138.11 \cdot 606 = 83694.66$ грн.

II. $C_М = 138.11 \cdot 702 = 96953.22$ грн.

Накладні витрати складають 67% від заробітної плати:

$$C_Н = C_{ЗП} \cdot 0,67, \#(4.19)$$

I. $C_Н = 72142.85 \cdot 0.67 = 48335.7$ грн.

II. $C_Н = 83566.08 \cdot 0.67 = 55989.27$ грн.

Отже, вартість розробки ПП становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_М + C_Н, \#(4.20)$$

I. $C_{ПП} = 72142.85 + 15871.42 + 83694.66 + 48335.7 = 220044.63$ грн.

II. $C_{ПП} = 83566.08 + 18384.53 + 96953.22 + 55989.27 = 254893.1$ грн.

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР_j} = К_{К_j} / С_{Ф_j}, \#(4.21)$$

$$К_{ТЕР1} = 5.43 \ / 220044.63 = 2{,}4676 \cdot 10^{-5},$$

$$К_{ТЕР2} = 5.16 \ / 254893.1 = 2{,}0243 \cdot 10^{-5},$$

Після виконання функціонально-вартісного аналізу програмного комплексу що розробляється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 2{,}4676 \cdot 10^{-5}$.

Цей варіант реалізації програмного продукту має такі параметри:

- Вибір мови програмування – C++;
- Реалізація інтерфейсу користувача у вигляді інтерфейсу командного рядку;
- Використання середовища розробки Visual Studio Code

Даний варіант виконання програмного комплексу дає користувачу зручний інтерфейс, швидку реалізацію програми та доступний функціонал для роботи.

4.8 Висновки до розділу

В даній частині було проведено повний функціонально-вартісний аналіз програмного продукту. Також було знайдено оцінку основних функцій програмного продукту.

В результаті виконання функціонально-вартісного аналізу програмного комплексу що розробляється, було визначено та проведено оцінку основних функцій програмного продукту, а також знайдено параметри, які його характеризують.

На основі аналізу вибрано варіант реалізації програмного продукту.

ВИСНОВКИ

У даній дипломній роботі було проведено огляд областей застосування нечіткого пошуку, включаючи автокорекцію помилок при введенні тексту користувачем, пошукові системи, комп'ютерний зір, та обчислювальну біологію. Розглянуто основні поняття, підходи, та алгоритми нечіткого пошуку, з акцентом на редагувальну відстань. Було обрано варіант прискорення нечіткого пошуку за допомогою скінченних автоматів, що дозволяє швидко обробляти рядки до заданої максимальної відстані.

Далі було розглянуто теорію скінченних автоматів і приступлено до програмної реалізації рішень на основі автомату Левенштейна. Розроблено та описано декілька варіантів програмної реалізації, включаючи TreeAutomaton, HashAutomaton та TableAutomaton.

Також виконано ретельну перевірку та аналіз рішень. Проведено тести для перевірки коректності реалізації та оцінки продуктивності, що включає побудову автомату, перевірку слова та загальні тести, що перевіряють швидкодію автоматів в порівнянні до тривіального підходу.

Спираючись на проведені тести та аналіз рішень можна зробити висновки, що всі розроблені рішення працюють швидше, ніж простий підхід з використанням алгоритму Дамерау-Левенштейна. На словнику з 370 000 слів та максимальних відстанях від 1 до 3 кожен автомат показав результати швидші в від 5 до 9 разів.

HashAutomaton рекомендується для загального використання, оскільки об’єм пам’яті, що використовується, значно меньше для високих значень редагувальної відстані ніж у інших рішень, а швидкодія перевірки слів не значно відрізняється від TreeAutomaton.

TreeAutomaton рекомендується для ситуацій, коли потрібна різна ціна для редагувальних операцій, або для редагувальної відстані в межах від 1 до 2. На малих значеннях редагувальної відстані він показує найбільшу швидкодію.

TableAutomaton рекомендується для ситуацій, коли пошукове слово часто змінюється, оскільки не потребує побудови та детермінізації для будь-якого шаблону та невеликих значень редагувальної відстані. Основним обмеженням швидкодії цього автомату є необхідність розрахунку характеристичного вектору для кожного вхідного символу, що може бути вдвічі прискорене за допомогою використання SIMD інструкцій для його обчислення, що наближає його швидкодію до HashAutomaton. Проте це рішення не є універсальним, оскільки потребує використання центрального процесору з підтримкою таких інструкцій.

У рамках функціонально-вартісного аналізу програмного продукту було обґрунтовано функції та систему параметрів програмного продукту, проведено аналіз експертного оцінювання параметрів, аналіз рівня якості варіантів реалізації функцій, економічний аналіз варіантів розробки ПП, та визначено кращий варіант програмного продукту з техніко-економічного погляду.

ПЕРЕЛІК ПОСИЛАНЬ

1. Gonzalo N. A guided tour to approximate string matching / Navarro Gonzalo. // Association for Computing Machinery. – 2001.
2. Schulz K. Fast string correction with Levenshtein automata / K. Schulz, S. Mihov. // International Journal on Document Analysis and Recognition. – 2002.
3. Boytsov L. Indexing methods for approximate dictionary searching: Comparative analysis / Leonid Boytsov. // Journal of Experimental Algorithmics. – 2011.
4. Google Benchmark [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/google/benchmark.
5. Benchmarking tips [Електронний ресурс] – Режим доступу до ресурсу: https://llvm.org/docs/Benchmarking.html.
6. Better Than unordered_map [Електронний ресурс]. – 2001. – Режим доступу до ресурсу: https://medium.com/applied/gist-better-than-unordered-map-1ad07b0a81b7.
7. Hamming R. W. Error detecting and error correcting codes / Richard Wesley Hamming. // The Bell System Technical Journal. – 1950.
8. Wagner R. A. The String-to-String Correction Problem / R. A. Wagner, M. J. Fischer. // Association for Computing Machinery. – 1974.
9. Baeza-Yates R. A faster algorithm for approximate string matching / R. Baeza-Yates, G. Navarro. // Springer, Berlin, Heidelberg. – 1996.
10. Ukkonen E. Approximate string-matching and the q-gram distance / Esko Ukkonen. // Springer, New York, NY. – 1993.
11. Mihov S. Fast Approximate Search in Large Dictionaries / S. Mihov, K. Schulz. // Computational Linguistics. – 2004.
12. Introduction to Algorithms, fourth edition / [T. Cormen, C. Leiserson, R. Rivest та ін.]., 2022. – 1312 с.

ДОДАТОК А

Лістинг коду кожного з реалізованих автоматів для нечіткого пошуку

tree_automaton.cpp

```
#include "tree_automaton.hpp"

#include <cmath>
#include <fstream>

namespace {
const float insertCost = 1.f;
const float deleteCost = 1.f;
const float replaceCost = 1.f;
const float transpositionCost = 1.f;
}  // namespace

std::pair<bool, float> TreeAutomaton::editDistance(
	std::wstring_view word) const {
	std::reference_wrapper<Node> currentNode = *m_root;
	for (auto c : word) {
	auto nextChild = currentNode.get().children.find(c);
	if (nextChild != currentNode.get().children.end()) {  // exist
		currentNode = nextChild->second;
	} else {
		auto universalChild =
		currentNode.get().children.find(m_universalSymbol);
		if (universalChild != currentNode.get().children.end()) {  // exist
		currentNode = universalChild->second;
		} else {
		return {false, maxScore};
		}
	}
	}

	return currentNode.get().finalState;
}

void TreeAutomaton::buildNFA() {
	while (!m_queue.empty()) {
	auto currentState = m_queue.front();
	m_queue.pop();
	auto& currentNode = currentState.curPtr;

	if (currentState.index == m_word.size()) {
		currentNode.finalState = {
		true,
		std::min(currentNode.finalState.second, currentState.score)};

		// try to insert extra symbols
		if (currentState.score + insertCost <= m_maxDistance) {
```

```
                //insert insertion symbols
                continue;
        }

        // 1. equal char
        auto& newNode = nextNode(currentNode, m_word[currentState.index]);
        m_queue.emplace(currentState.index + 1, currentState.score, newNode);

        // 2. insert
        if (currentState.score + insertCost <= m_maxDistance) {
                auto& newNode = nextNode(currentNode, m_universalSymbol);
                m_queue.emplace(currentState.index, currentState.score + insertCost,
                                newNode);
        }

        // 3. delete
        if (currentState.score + deleteCost <= m_maxDistance) {
                m_queue.emplace(currentState.index + 1,
                                currentState.score + deleteCost, currentNode);
        }

        // 4. transpose
        if (currentState.score + transpositionCost <= m_maxDistance &&
                currentState.index < m_word.size() - 1) {
                auto& newNode =
                nextNode(currentNode, m_word[currentState.index + 1]);
                auto& nextNewNode = nextNode(newNode, m_word[currentState.index]);
                m_queue.emplace(currentState.index + 2,
                                currentState.score + transpositionCost,
                                nextNewNode);
        }

        // 5. replace
        if (currentState.score + replaceCost <= m_maxDistance) {
                auto& universalNode = nextNode(currentNode, m_universalSymbol);
                m_queue.emplace(currentState.index + 1,
                                currentState.score + replaceCost, universalNode);
        }
        }
}

void TreeAutomaton::convertToDFA(Node& currentNode) {
        auto universalChild = currentNode.children.find(m_universalSymbol);
        bool isExist = universalChild != currentNode.children.end();
        for (auto& state : currentNode.children) {
        if (isExist && state.first != m_universalSymbol) {
                for (auto& universalNode : universalChild->second.children) {
                recursiveInsertNode(state.second, universalNode.first,
                                    universalNode.second);
                }

                if (universalChild->second.finalState.first) {
```

```
                // set final, update edit distance
                }
        }

        convertToDFA(state.second);
        }
}
```

hash_automaton.cpp

```
#include "hash_automaton.hpp"

#include <algorithm>
#include <cassert>
#include <queue>

namespace {
const size_t shift = 1007;
const float maxScore = 1000000.f;
const wchar_t ANY = L"?"[0];
const wchar_t EPS = L"#"[0];
}  // namespace

namespace hash {
HashAutomaton::HashAutomaton(const std::wstring& word, float dist)
        : m_word(word), m_maxDistance(dist), m_startState({0, 0}), m_curIndex(0) {
        m_statesMap.reserve(128);
        m_indexMap.reserve(128);
        m_dfaVec.reserve(128);
        m_finalStates.reserve(128);

        createAlphabet(m_word);  // O(n log n)
        buildNFA();               // O(nd)
        convertToDFA();

        m_defIndex = getAlphabetIndex(m_alphabet, ANY);  // O(log n)
}

void HashAutomaton::convertToDFA() {
        std::queue<std::set<State>> frontier;
        frontier.push(expand({m_startState}));

        while (!frontier.empty()) {
        auto current = std::move(frontier.front());
        frontier.pop();

        const auto& inputs = getInputs(current);
        const auto& curStateIdx = getStateIndex(current);

        for (auto input : inputs) {
                if (input == EPS) continue;
```

```
                const auto& newState = nextState(current, input);
                const auto& newStateIdx = getStateIndex(newState);

                if (newStateIdx.first) {  // state has not been processed yet
                frontier.push(newState);

                const auto& isFinal = isFinalState(newState, m_word.length());
                if (isFinal.first) {
                        m_finalStates[newStateIdx.second] = isFinal.second;
                }
                }

                auto index = getAlphabetIndex(m_alphabet, input);
                assert(index != -1);

                m_dfaVec[curStateIdx.second][index] = newStateIdx.second;
        }
        }
}

void HashAutomaton::addTransition(const State& src, wchar_t input,
                                  const State& dest) {
        m_statesMap[src][input].insert(dest);
}

std::pair<bool, float> HashAutomaton::editDistance(
        std::wstring_view word) const {
        size_t key = 0;
        for (auto c : word) {
        const auto& curVec = m_dfaVec[key];

        auto index = getAlphabetIndex(m_alphabet, c);
        if (index != -1) {
                auto nextKey = curVec[index];
                if (nextKey != -1) {
                key = nextKey;
                continue;
                }
        }

        auto nextKey = curVec[m_defIndex];
        if (nextKey != -1) {
                key = nextKey;
                continue;
        }

        return {false, maxScore};
        }

        auto curScore = m_finalStates[key];
        if (curScore != maxScore) {
```

```
        return {true, curScore};
        }

        return {false, maxScore};
}

}  // end of namespace hash
```

table_automaton.hpp

```
#ifndef TABLE_AUTOMATON_H_
#define TABLE_AUTOMATON_H_

#include <algorithm>
#include <array>
#include <cmath>
#include <map>
#include <memory>
#include <optional>
#include <queue>
#include <stack>
#include <string>
#include <string_view>
#include <unordered_map>
#include <unordered_set>

#include "tools.hpp"

class State;
class Position;

using StateIndex = int64_t;
using CharacteristicValue = uint64_t;
using Table = std::vector<std::vector<std::tuple<int, StateIndex>>>;

inline CharacteristicValue characteristic(const std::wstring& word, wchar_t c,
                                          int offset, int d) {
        int w = word.size();
        int k = std::min(2 * d + 1, w - offset);
        CharacteristicValue result = 0;
        for (int i = 0; i < k; ++i) {
        result |= (static_cast<int>(word[i + offset] == c) << i);
        }
        return result;
}

struct Position {
        int i;
        int e;
        bool transposition = false;
        auto operator<=>(const Position& rhs) const {
```

```
        if (i < rhs.i) return -1;
        if (i > rhs.i) return 1;
        if (transposition < rhs.transposition) return -1;
        if (transposition > rhs.transposition) return 1;
        if (e < rhs.e) return -1;
        if (e > rhs.e) return 1;
        return 0;
        }
        bool operator==(const Position&) const = default;
};

constexpr bool subsumes(Position a, Position b) {
        if (b.transposition || a.transposition) return false;
        return a.e < b.e && (std::abs(b.i - a.i) <= b.e - a.e);
}

constexpr State elementary_transition(Position start,
                                      const CharacteristicValue& ch, int n);
struct State {
        std::vector<Position> positions;

        constexpr State(std::initializer_list<Position> p)
        : positions(std::move(p)) {
        std::sort(positions.begin(), positions.end());
        }

        constexpr Position minimal_boundary() const {
        if (positions.size())
                return positions[0];
        else
                return {0, 0};
        }

        constexpr std::vector<Position>::const_iterator begin() const {
        return positions.begin();
        }
        constexpr std::vector<Position>::const_iterator end() const {
        return positions.end();
        }

        constexpr bool operator==(const State&) const = default;
        constexpr void clean() {
        std::vector<Position> not_subsumed;
        for (int i = 0; i < positions.size(); ++i) {
                const auto& i_pos = positions[i];
                bool is_subsumed = false;
                for (int j = 0; j < positions.size(); ++j) {
                if (i == j) continue;

                const auto& j_pos = positions[j];
                if ((i_pos == j_pos && j < i) || subsumes(j_pos, i_pos)) {
                        is_subsumed = true;
```

```
                        break;
                }
                }
                if (!is_subsumed) {
                not_subsumed.push_back(i_pos);
                }
        }

        positions = std::move(not_subsumed);
        }

        constexpr State next(const CharacteristicValue& ch, int d) const {
        State result{};
        int min_i = minimal_boundary().i;

        for (const auto& position : positions) {
                int ch_offset = position.i - min_i;
                int k = d - position.e + 1;

                CharacteristicValue new_ch = (ch >> ch_offset) & ((1 << k) - 1);

                for (const auto& new_position :
                elementary_transition(position, new_ch, d)) {
                result.positions.push_back(new_position);
                }
        }

        result.clean();

        return result;
        }
};

constexpr std::optional<int> first_true(const CharacteristicValue& ch) {
        int ffs = std::countr_zero(ch);

        if (ffs == 64) {
        return std::nullopt;
        }
        return ffs + 1;
}

constexpr State elementary_transition(Position start,
                                      const CharacteristicValue& ch, int n) {
        const int i = start.i;
        const int e = start.e;

        if (e < n) {
        if (start.transposition) {
                if (ch & 1) {
                return State({{i + 2, e + 1}});
                } else {
```

```
                return State{};
                }
        }
        if (ch & 1) {
                // Right move. Equals
                return State({{i + 1, e}});
        } else {
                auto j = first_true(ch);
                if (j) {
                // Up(insert), Up-right(substitute), Up-right jump(delete)
                return State{{i, e + 1},
                             {i + 1, e + 1},
                             {i + *j, e + *j - 1},
                             {i + *j - 2, e + *j - 2, true}};
                } else {
                return State{{i, e + 1}, {i + 1, e + 1}};
                }
        }
        } else {
        if (start.transposition) {
                return State{};
        }
        if (ch & 1) {
                return State({{i + 1, e}});
        } else {
                return State{};
        }
        }
}

inline bool is_accepting(Position position, int w, int n, int offset) {
        if (position.transposition) return false;
        if ((position.i + offset) > w) return false;
        return w - (position.i + offset) <= n - position.e;
}

inline std::pair<bool, float> is_accepting(const State& state, int w, int n,
                                           int offset) {
        int min_d = n;
        bool found = false;
        for (const auto& position : state) {
        if (is_accepting(position, w, n, offset)) {
                found = true;
                if (position.e + w - (position.i + offset) < min_d)
                min_d = position.e + w - (position.i + offset);
        }
        }

        return {found, min_d};
}

struct Precomputed {
```

```
        MMatrix<std::tuple<int, StateIndex>> transitions_table;
        std::vector<State> index;
        StateIndex final;
};

constexpr Precomputed build_table(int d);

class TableAutomaton {
    public:
        TableAutomaton(std::wstring word, float maxDist)
        : m_word(std::move(word)), m_maxDistance(std::ceil(maxDist)) {}
        ~TableAutomaton() = default;

        std::pair<bool, float> editDistance(std::wstring_view pattern) const;

    private:
        int m_maxDistance;
        std::wstring m_word;
};
#endif  // TABLE_AUTOMATON_H_
```

table_automaton.cpp

```
#include "table_automaton.hpp"

#include <bit>

constexpr std::pair<int, State> normalize(const State& state) {
        int offset = state.minimal_boundary().i;
        State copy = state;
        for (auto& pos : copy.positions) {
        pos.i -= offset;
        }

        return {offset, copy};
}

constexpr Precomputed build_table(int d) {
        Table table;
        std::vector<State> index;
        StateIndex final = 0;
        std::vector<StateIndex> to_check;

        const CharacteristicValue characteristic_upper_bound =
        (1 << (2 * d + 1)) - 1;

        State initial{{0, 0}};

        index.push_back(initial);
        table.emplace_back();
        to_check.push_back(0);
```

```
        while (!to_check.empty()) {
        StateIndex current_index = to_check.back();
        to_check.pop_back();

        State current_state = index[current_index];

        for (CharacteristicValue ch = 0; ch <= characteristic_upper_bound;
              ++ch) {
              auto [offset, new_state] = normalize(current_state.next(ch, d));

              auto new_index =
              std::distance(index.begin(),
                           std::find(index.begin(), index.end(), new_state));

              table[current_index].push_back({offset, new_index});

              if (new_index == index.size()) {
              index.push_back(new_state);
              table.emplace_back();
              to_check.push_back(new_index);
              if (new_state.positions.empty()) {
                    final = new_index;
              }
              }
        }
        }

        MMatrix<std::tuple<int, StateIndex>> matrix(table.size(), table[0].size());

        for (int i = 0; i < table.size(); ++i) {
        for (int j = 0; j < table[i].size(); ++j) {
              matrix.at(i, j) = table[i][j];
        }
        }
        return {matrix, index, final};
}

std::array precomputed_tables{build_table(0), build_table(1), build_table(2),
                           build_table(3)};

std::pair<bool, float> TableAutomaton::editDistance(
        std::wstring_view pattern) const {
        StateIndex current = 0;
        int offset = 0;

        const auto& precomputed = precomputed_tables[m_maxDistance];

        for (const auto& c : pattern) {
        auto ch = characteristic(m_word, c, offset, m_maxDistance);
        const auto& [shift, new_state] =
              precomputed.transitions_table.at(current, ch);
```

```
        if (new_state == precomputed.final) {
              return {false, 0};
        }

        offset += shift;
        current = new_state;
        }

        return is_accepting(precomputed.index[current], m_word.size(),
                            m_maxDistance, offset);
}
```

ДОДАТОК Б

main_correctness_test.cpp

```cpp
#undef NDEBUG
#include <cassert>
#define NDEBUG

#include <chrono>
#include <codecvt>
#include <fstream>
#include <iostream>
#include <locale>
#include <random>

#include "hash_automaton.hpp"
#include "table_automaton.hpp"
#include "tools.hpp"
#include "tree_automaton.hpp"

using namespace std::chrono;

static std::wstring_convert<std::codecvt_utf8<wchar_t>, wchar_t> ucs4conv;
std::wstring string2_w(const char* first, const char* last) {
	return ucs4conv.from_bytes(first, last);
}

std::wstring string2_w(const std::string& s) {
	return string2_w(s.c_str(), s.c_str() + s.size());
}

template <typename T>
void testCalculateScore(const std::vector<std::wstring>& wordsVector) {
	std::random_device rd;
	std::mt19937 gen(rd());
	std::uniform_int_distribution<> randomIndex(0, wordsVector.size() - 1);
	std::uniform_int_distribution<> randomError(0, 3);

	const int searchPhrasesAmount = 100;

	for (int i = 0; i < searchPhrasesAmount; i++) {
	int searchIndex = randomIndex(gen);
	const float maxAllowedError = randomError(gen);
	std::wstring searchPhrase = wordsVector[searchIndex];
	T automaton(searchPhrase, maxAllowedError);

	for (const auto& word : wordsVector) {
		const auto& dist1 = editDistance(searchPhrase, word);
		const auto& dist2 = automaton.editDistance(word);

		// if (dist2.second > 0)
		//     std::cout << dist1 << " " << dist2.second << std::endl;
```

```
                if (dist1 <= maxAllowedError) {
                std::wcout << searchPhrase << " " << word << std::endl;
                std::cout << dist1 << " " << dist2.second << std::endl;
                assert(dist1 == dist2.second);
                } else {
                assert(dist2.first == false);
                // assert(dist2.second == maxScore);
                }
        }
        }
}

void correctnessTest() {
        std::ifstream file("words.txt");

        std::vector<std::wstring> wordsVector;
        std::string word;
        while (file >> word) {
        wordsVector.emplace_back(string2_w(word));
        }

        testCalculateScore<TreeAutomaton>(wordsVector);
        testCalculateScore<HashAutomaton>(wordsVector);
        testCalculateScore<TableAutomaton>(wordsVector);
}

int main(int argc, char* argv[]) {
        correctnessTest();


        return 0;
}
```

tools.hpp

```
#ifndef TOOLS_HPP
#define TOOLS_HPP

#include <string>
#include <string_view>
#include <vector>

template <typename T>
struct MMatrix {
        constexpr MMatrix() : row(0), col(0), innerVec(std::vector<T>()) {}

        constexpr MMatrix(size_t aLen, size_t bLen)
        : row(aLen), col(bLen), innerVec(std::vector<T>(row * col)) {}

        constexpr T& at(size_t i, size_t j) { return innerVec[i * col + j]; }
```

```
        constexpr const T& at(size_t i, size_t j) const {
        return innerVec[i * col + j];
        }

    private:
        const size_t row;
        const size_t col;
        std::vector<T> innerVec;
};

float editDistance(std::wstring_view a, std::wstring_view b);
std::pair<bool, float> maxEditDistance(std::wstring_view a, std::wstring_view b,
                                       float d);
int SearchString(std::wstring_view text, std::wstring_view pattern, int k);
#endif /* TOOLS_HPP */
```

tools.cpp

```
#include "tools.hpp"

#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>

namespace {
const float insertCost = 1.f;
const float deleteCost = 1.f;
const float replaceCost = 1.f;
const float transpositionCost = 1.f;

template <typename T>
struct Matrix {
        constexpr Matrix(size_t aLen, size_t bLen)
        : row(aLen), col(bLen), innerVec(std::vector<T>(row * col, 100.f)) {}

        constexpr T& at(size_t i, size_t j) { return innerVec[i * col + j]; }

    private:
        const size_t row;
        const size_t col;
        std::vector<T> innerVec;
};
}  // namespace

float editDistance(std::wstring_view a, std::wstring_view b) {
        const size_t aLen = a.length() + 1;
        const size_t bLen = b.length() + 1;

        Matrix<float> matr(aLen, bLen);
```

```
	for (size_t i = 0; i < aLen; ++i) matr.at(i, 0) = i;

	for (size_t j = 0; j < bLen; ++j) matr.at(0, j) = j;

	// building matrix
	for (size_t i = 1; i < aLen; ++i) {
	for (size_t j = 1; j < bLen; ++j) {
		float compareCharCost = a[i - 1] == b[j - 1] ? 0.f : replaceCost;

		// replace, insert, delete
		matr.at(i, j) = std::min({matr.at(i - 1, j - 1) + compareCharCost,
					matr.at(i, j - 1) + insertCost,
					matr.at(i - 1, j) + deleteCost});

		// transposition
		if (i > 1 && j > 1 &&
		(a[i - 1] == b[j - 2] && a[i - 2] == b[j - 1]))
		matr.at(i, j) = std::min(
			matr.at(i, j), matr.at(i - 2, j - 2) + transpositionCost);
	}
	}

	return matr.at(aLen - 1, bLen - 1);
}
```

benchmark_automaton.cpp

```
#include <benchmark/benchmark.h>

#include <codecvt>
#include <fstream>
#include <iostream>
#include <locale>
#include <random>

#include "hash_automaton.hpp"
#include "table_automaton.hpp"
#include "tools.hpp"
#include "tree_automaton.hpp"

using namespace tree;
using namespace hash;

const static std::vector<std::wstring> words = {
	L"",
	L"a",
	L"an",
	L"cat",
	L"dogs",
	L"apple",
	L"banana",
```

```
        L"compute",
        L"language",
        L"algorithm",
        L"innovation",
        L"engineering",
        L"intelligence",
        L"complimentary",
        L"infrastructure",
        L"characteristics",
};

const int MAX_EDIT_RANGE = 3;
const int MAX_HASH_EDIT_RANGE = 6;

template <class Automaton>
static void BM_AutomatonCreation(benchmark::State& state) {
        int n = state.range(0);
        float maxError = static_cast<float>(state.range(1));

        for (auto _ : state) Automaton automaton(words[n], maxError);
}
BENCHMARK(BM_AutomatonCreation<TreeAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                       benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonCreation<HashAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                       benchmark::CreateDenseRange(1, MAX_HASH_EDIT_RANGE, 1)});

template <class Automaton>
static void BM_AutomatonCheckDistance(benchmark::State& state) {
        int n = state.range(0);
        float maxError = static_cast<float>(state.range(1));
        const auto& word = words[n];
        Automaton automaton(word, maxError);

        for (auto _ : state) automaton.editDistance(word);
}
BENCHMARK(BM_AutomatonCheckDistance<TreeAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                       benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonCheckDistance<HashAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                       benchmark::CreateDenseRange(1, MAX_HASH_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonCheckDistance<TableAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                       benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});

template <class Automaton>
static void BM_AutomatonCheckDistanceNotSame(benchmark::State& state) {
        int n = state.range(0);
        float maxError = static_cast<float>(state.range(1));
        const auto& word = words[n];
```

```
	const auto& otherWord = words[n - 1];
	Automaton automaton(word, maxError);

	for (auto _ : state) automaton.editDistance(otherWord);
}
BENCHMARK(BM_AutomatonCheckDistanceNotSame<TreeAutomaton>)
	->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
			benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonCheckDistanceNotSame<HashAutomaton>)
	->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
			benchmark::CreateDenseRange(1, MAX_HASH_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonCheckDistanceNotSame<TableAutomaton>)
	->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
			benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});

static void BM_LevenshteinCheckDistance(benchmark::State& state) {
	int n = state.range(0);
	const auto& word = words[n];

	for (auto _ : state) editDistance(word, word);
}
BENCHMARK(BM_LevenshteinCheckDistance)->DenseRange(1, 15, 1);

static void BM_LevenshteinCheckDistanceNotSame(benchmark::State& state) {
	int n = state.range(0);
	const auto& word = words[n];
	const auto& otherWord = words[n - 1];

	for (auto _ : state) editDistance(word, otherWord);
}
BENCHMARK(BM_LevenshteinCheckDistanceNotSame)->DenseRange(1, 15, 1);

static std::wstring_convert<std::codecvt_utf8<wchar_t>, wchar_t> ucs4conv;
std::wstring string2_w(const char* first, const char* last) {
	return ucs4conv.from_bytes(first, last);
}

std::wstring string2_w(const std::string& s) {
	return string2_w(s.c_str(), s.c_str() + s.size());
}

const int seed = 42;
template <class Automaton>
static void BM_AutomatonMany(benchmark::State& state) {
	std::ifstream file("words.txt");

	std::vector<std::wstring> wordsVector;
	std::string word;
	while (file >> word) {
	wordsVector.emplace_back(string2_w(word));
	}
```

```
	const int testPhrasesAmount = wordsVector.size();

	std::mt19937 gen(seed);
	std::uniform_int_distribution<> int_distr(0, wordsVector.size() - 1);

	const int searchPhrasesAmount = state.range(0);
	const float maxAllowedError = static_cast<float>(state.range(1));

	for (auto _ : state) {
	for (int i = 0; i < searchPhrasesAmount; i++) {
		int searchIndex = int_distr(gen);
		const int beginT =
		std::floor((static_cast<float>(i) * testPhrasesAmount) /
				searchPhrasesAmount);
		const int endT =
		std::ceil((static_cast<float>(i + 1) * testPhrasesAmount) /
				searchPhrasesAmount);
		const std::wstring& searchPhrase = wordsVector[searchIndex];
		Automaton automaton(searchPhrase, maxAllowedError);

		for (int j = beginT; j < endT; j++) {
		benchmark::DoNotOptimize(
			automaton.editDistance(wordsVector[j]));
		}
	}
	}
}
BENCHMARK(BM_AutomatonMany<TreeAutomaton>)
	->ArgsProduct({benchmark::CreateRange(1, 100, 10),
			benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonMany<HashAutomaton>)
	->ArgsProduct({benchmark::CreateRange(1, 100, 10),
			benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonMany<TableAutomaton>)
	->ArgsProduct({benchmark::CreateRange(1, 100, 10),
			benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});

static void BM_LevenshteinMany(benchmark::State& state) {
	std::ifstream file("words.txt");

	std::vector<std::wstring> wordsVector;
	std::string word;
	while (file >> word) {
	wordsVector.emplace_back(string2_w(word));
	}
	const int testPhrasesAmount = wordsVector.size();

	std::mt19937 gen(seed);
	std::uniform_int_distribution<> int_distr(0, wordsVector.size() - 1);

	const int searchPhrasesAmount = state.range(0);
```

```
        for (auto _ : state) {
        for (int i = 0; i < searchPhrasesAmount; i++) {
                int searchIndex = int_distr(gen);
                const int beginT =
                std::floor((static_cast<float>(i) * testPhrasesAmount) /
                                searchPhrasesAmount);
                const int endT =
                std::ceil((static_cast<float>(i + 1) * testPhrasesAmount) /
                                searchPhrasesAmount);
                const std::wstring& searchPhrase = wordsVector[searchIndex];
                for (int j = beginT; j < endT; j++) {
                benchmark::DoNotOptimize(
                        editDistance(searchPhrase, wordsVector[j]));
                }
        }
        }
}
BENCHMARK(BM_LevenshteinMany)
        ->ArgsProduct({benchmark::CreateRange(1, 1000, 10)});

BENCHMARK_MAIN();
```

benchmark_memory.cpp

```
#include <benchmark/benchmark.h>
#include <malloc.h>

#include <codecvt>
#include <fstream>
#include <iostream>
#include <locale>
#include <random>

#include "hash_automaton.hpp"
#include "tools.hpp"
#include "tree_automaton.hpp"

using namespace tree;
using namespace hash;
long long int numAllocations = 0;
long long int numBytesAllocated = 0;
long long int initial_bytes = 0;
long long int max_used = 0;
void* operator new(std::size_t sz) {
        if (sz == 0)
        ++sz;  // avoid std::malloc(0) which may return nullptr on success

        if (void* ptr = std::malloc(sz)) {
        numAllocations++;
        numBytesAllocated += malloc_usable_size(ptr);
        if (numBytesAllocated - initial_bytes > max_used) {
```

```
            max_used = numBytesAllocated - initial_bytes;
        }
        return ptr;
        }

        throw std::bad_alloc{};  // required by [new.delete.single]/3
}

// no inline, required by [replacement.functions]/3
void* operator new[](std::size_t sz) {
        if (sz == 0)
        ++sz;  // avoid std::malloc(0) which may return nullptr on success

        if (void* ptr = std::malloc(sz)) {
        numAllocations++;
        if (numBytesAllocated - initial_bytes > max_used) {
            max_used = numBytesAllocated - initial_bytes;
        }
        numBytesAllocated += malloc_usable_size(ptr);
        return ptr;
        }

        throw std::bad_alloc{};  // required by [new.delete.single]/3
}
void operator delete(void* ptr) noexcept {
        numBytesAllocated -= malloc_usable_size(ptr);
        std::free(ptr);
}

void operator delete(void* ptr, std::size_t size) noexcept {
        numBytesAllocated -= malloc_usable_size(ptr);
        std::free(ptr);
}

void operator delete[](void* ptr) noexcept {
        numBytesAllocated -= malloc_usable_size(ptr);
        std::free(ptr);
}

void operator delete[](void* ptr, std::size_t size) noexcept {
        numBytesAllocated -= malloc_usable_size(ptr);
        std::free(ptr);
}

class CustomMemoryManager : public benchmark::MemoryManager {
   public:
        int64_t num_allocs;
        int64_t whole;

        void Start() BENCHMARK_OVERRIDE {
        num_allocs = numAllocations;
        whole = numBytesAllocated;
```

```
        initial_bytes = numBytesAllocated;
        max_used = 0;
        }

        void Stop(Result* result) BENCHMARK_OVERRIDE {
        result->num_allocs = numAllocations - num_allocs;
        result->max_bytes_used = max_used;
        }
};

const static std::vector<std::wstring> words = {
        L"",
        L"a",
        L"an",
        L"cat",
        L"dogs",
        L"apple",
        L"banana",
        L"compute",
        L"language",
        L"algorithm",
        L"innovation",
        L"engineering",
        L"intelligence",
        L"complimentary",
        L"infrastructure",
        L"characteristics",
};

template <class Automaton>
static void BM_AutomatonMemory(benchmark::State& state) {
        int n = state.range(0);
        float maxError = static_cast<float>(state.range(1));

        std::unique_ptr<Automaton> a;
        for (auto _ : state) {
        a = std::make_unique<Automaton>(words[n], maxError);
        }
}

const int MAX_EDIT_RANGE = 3;

BENCHMARK(BM_AutomatonMemory<TreeAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                        benchmark::CreateDenseRange(1, MAX_EDIT_RANGE, 1)});
BENCHMARK(BM_AutomatonMemory<HashAutomaton>)
        ->ArgsProduct({benchmark::CreateDenseRange(1, 15, 1),
                        benchmark::CreateDenseRange(1, 5, 1)});

std::unique_ptr<CustomMemoryManager> mm(new CustomMemoryManager());

int main(int argc, char** argv) {
```

```
        ::benchmark::RegisterMemoryManager(mm.get());
        ::benchmark::Initialize(&argc, argv);
        ::benchmark::RunSpecifiedBenchmarks();
        ::benchmark::RegisterMemoryManager(nullptr);
}
```