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: «Дослідження алгоритмів нечіткого пошуку в текстових даних з використанням таблиці подібності символів»

Виконав:
студент IV курсу, групи ДА-91
Царьов Максим Олександрович _____

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
Царьову Максиму Олександровичу

1. Тема роботи «Дослідження алгоритмів нечіткого пошуку в текстових даних з використанням таблиці подібності символів», керівник роботи Клещ Кирило Олегович, асистент, затверджені наказом по університету від «30»травня 2023 р. № 2065-с.
2. Термін подання студентом роботи – 10 червня 2023 р.
3. Вихідні дані до роботи:

Дослідження алгоритмів нечіткого пошуку.

Реалізація таблиці подібності символів.

Реалізація алгоритму нечіткого пошуку з використанням таблиці подібності символів.

4. Зміст роботи:
    1. Вступ.

2. Постановка задачі проєктування.
3. Дослідження сучасних алгоритмів нечіткого пошуку в тексті.
4. Дослідження алгоритму нечіткого пошуку в тексті з використанням таблиці подібності.
5. Реалізація алгоритму нечіткого пошуку в тексті з використанням таблиці подібності.
6. Тестування алгоритму нечіткого пошуку в тексті.
7. Порівняння швидкодії алгоритмів нечіткого пошуку в тексті з використанням таблиці подібності символів і без.
8. Функціонально-вартісний аналіз програмного забезпечення.
9. Висновки.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо):
    1. Презентація до захисту роботи.
6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Економічний	Рощина Н. В., к. е. н., доцент		

7. 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	Аналіз сучасних алгоритмів нечіткого пошуку в тексті	30.03.2023	
3	Дослідження алгоритму нечіткого пошуку в тексті з використанням таблиці подібності	15.04.2023	

4	Реалізація алгоритму нечіткого пошуку з використанням таблиці подібності	30.04.2023	
5	Тестування розробленого алгоритму	10.05.2023	
6	Порівняння швидкодії алгоритмів нечіткого пошуку в тексті з використанням таблиці подібності символів і без	20.05.2023	
7	Проведення функціонально-вартісного аналізу програмного забезпечення	30.05.2023	
8	Оформлення дипломної роботи	15.06.2023	

Студент Царьов М.О.

Керівник Клещ К. О.

АНОТАЦІЯ

бакалаврської дипломної роботи Царьова Максима Олександровича на тему «Дослідження алгоритмів нечіткого пошуку в текстових даних з використанням таблиці подібності символів»

У сучасному світі, насиченому величезним обсягом інформації, здатність ефективно знаходити в тексті релевантну та точну інформацію в тексті є надзвичайно важливою. Саме тому пошукові алгоритми є надзвичайно важливими інструментами, які дозволяють нам орієнтуватися в цьому величезному обсязі доступної інформації. Проте традиційні алгоритми не справляються з випадками, коли текст чи пошуковий запит містить орфографічні помилки чи друкарські помилки. У таких випадках, щоби швидко знайти відповідні результати нам допомагають алгоритми нечіткого пошуку.

Тому метою дипломної роботи є дослідження сучасних алгоритмів нечіткого пошуку в тексті, реалізація нечіткого пошуку в тексті з використанням таблиці подібності символів та створення самої таблиці.

Передусім найважливішою частиною дипломної роботи є саме розробка алгоритму нечіткого пошуку з використанням таблиці подібності. Реалізований алгоритм має включати декілька етапів, які включають попередню обробку таблиці подібності символів, токенізацію вхідного тексту, обчислення балів подібності та виконання нечіткого зіставлення. Також для успішної роботи алгоритму було проведено його тестування та проведено порівняння швидкодії алгоритму з використанням таблиці подібності символів та без.

Результатами роботи є дослідження сучасних алгоритмів нечіткого пошуку, розроблена таблиця подібності символів та реалізований алгоритм нечіткого пошуку з використанням таблиці подібності.

Загальний обсяг роботи 100 с., 35 рис., 6 таблиць, 4 додатки, 11 джерела.

Ключові слова: нечіткий пошук, таблиця подібності символів, алгоритм Дамерау-Левенштейна, обробка текстових даних, відстань редагування.

ABSTRACT

to the Tsarov Maxim bachelor's degree thesis on the topic

«Research on fuzzy search algorithms in textual data using symbol similarity table»

In today’s world, filled with a huge amount of information, the ability to effectively find relevant and accurate information in a text is extremely important. That is why search algorithms are extremely important tools that allow us to operate with this huge amount of available information. However, traditional algorithms do not handle cases where the text or search query contains misspellings or typographical errors. In such cases, fuzzy search algorithms help us to quickly find relevant results.

Therefore, the aim of the thesis is to research on modern algorithms for fuzzy text search, to implement fuzzy search algorithm using a symbol similarity table and to create the table itself.

First of all, the most important part of the thesis is the development of a fuzzy search algorithm using a symbol similarity table. The algorithm's implementation should involve several steps, including preprocessing the symbol similarity table, tokenizing input string, calculating similarity scores, and performing fuzzy matching. Also, for the successful operation of the algorithm, it was tested and was performed a comparison of the speeds of the algorithms with and without the use of the symbol similarity table.

The results of the work are a research on modern fuzzy search algorithms, a developed symbol similarity table and an implemented fuzzy search algorithm using a symbol similarity table.

The total volume of work is 100 pages, 35 figures, 6 tables, 4 appendices, 11 sources.

Keywords: fuzzy search, symbol similarity table, Damerau-Levenstein algorithm, text processing, edit distance.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Нечіткий пошук – це пошук рядків, подібних або близьких до пошукового запиту.

NLP (Natural Language Processing) – це галузь комп'ютерної науки, яка займається проблемами комп'ютерного аналізу та синтезу природної мови.

UNICODE (Universal Coded Character Set) – це стандарт кодування символів, який визначає універсальну систему для представлення тексту майже всіх писемностей світу.

C++ – це компілювана, статично типізована, об'єктно-орієнтована мова програмування.

Python – це високорівнева, інтерпретована, об'єктно-орієнтована мова програмування зі строгою динамічною типізацією.

JSON (JavaScript Object Notation) – це текстовий формат обміну даними.

UTF-16 (Unicode Transformation Format) – один із способів кодування символів з Юнікоду у вигляді послідовності 16-бітових слів.

Асимптотична складність – це поняття, яке використовується в аналізі алгоритмів для визначення темпу зростання часу або просторової складності алгоритму при збільшенні розміру вхідних даних.

RapidJSON – це швидка, загальнопризначена бібліотека для обробки JSON в C++.

GoogleTest (Google C++ Testing Framework або gtest) – це популярна бібліотека для автоматизованого тестування програм на мові C++.

GoogleTest фікстура – це механізм, який дозволяє підготувати спільний контекст або стан для набору тестів.

IDE (Integrated development environment) – це інтегроване середовище розробки, що надає розробникам зручність інструментів для написання, тестування, налагодження і керування програмним кодом.

Google Benchmark - це бібліотека для вимірювання та аналізу продуктивності коду в мові програмування C++.

Бенчмарк (benchmark) – це процес вимірювання продуктивності або ефективності системи, компонента або алгоритму.

Matplotlib – це бібліотека для створення графіків та візуалізації даних в мові програмування Python.

ФВА – функціонально-вартісний аналіз.

ПЗ – програмне забезпечення.

ПП – програмний продукт.

ВСТУП

У сучасному світі інформація відіграє вирішальну роль у нашому житті. Інформація служить нам основою для особистих знань, завдяки яким ми формуємо наші ідеї та погляди, приймаємо обґрунтовані рішення та вирішуємо проблеми. Від особистого розвитку до наукових досягнень важливість інформації важко переоцінити.

Тому, враховуючи вагомість інформації, здатність до ефективного пошуку у великих масивах даних стає надзвичайно важливою для досягнення успіху у різних сферах. Працюючи з текстовими даними, такими як документи, веб-сторінки, електронні повідомлення та інше, критично важливо мати можливість швидко і точно знаходити необхідну інформацію.

Однак, традиційні алгоритми пошуку можуть бути неефективними, особливо у випадках, коли ми стикаємося з нечіткими запитами, з певними помилками при введені пошукових запитів або з певними неточностями, які зустрічаються в текстових даних. Наприклад, це можуть бути орфографічні, друкарські помилки. У таких випадках традиційні алгоритми пошуку, що базуються на точних відповідностях, не здатні задовольнити потреби користувачів. Саме тому алгоритми нечіткого пошуку в тексті стають надзвичайно важливими.

Дослідження алгоритмів нечіткого пошуку в тексті є одним з важливих напрямків дослідження в області інформаційного пошуку та обробки тексту. Нечіткий пошук – це технологія, яка використовується в інформаційному та текстовому пошуку, щоб знайти збіги для даного запиту або шаблону, навіть якщо є помилки або невизначеності в даних, в яких проводиться пошук.

Нечіткий пошук у тексті використовують в багатьох різних галузях, де точність традиційного пошуку може бути обмеженою. Наприклад, може використовуватися в таких сферах:

– Інформаційний пошук: у пошукових системах нечіткий пошук дозволяє користувачам знаходити релевантні документи навіть при

наявності орфографічних помилок або інших форм неточностей у запит;

- Перевірка орфографії та автовиправлення: нечіткий пошук використовується в системах перевірки орфографії, щоб запропонувати виправлення неправильно написаних слів;
- Дедуплікація даних: нечіткий пошук може бути корисний для виявлення та усунення дублікатів записів у базах даних. Порівнюючи текстовий вміст записів, алгоритми нечіткої відповідності можуть виявляти подібні або майже ідентичні записи, зменшуючи надмірність і покращуючи якість даних;
- Автопропозиції та автозавершення: нечіткий пошук може використовуватися в програмах, в яких, коли користувачі вводять текст, вони надають пропозиції для завершення рядка тексту;
- Обробка природньої мови (Natural Language Processing, NLP): в NLP нечіткий пошук може використовуватися для різних завдань, таких як розпізнавання об’єктів, обробка запитів, виправлення орфографії, кластеризація та багато інших завдань, в яких необхідно враховувати семантичні чи синтаксичні відхилення у тексті.

Нечіткий пошук може використовувати широкий спектр критеріїв для оцінки результатів. Тому існує досить багато різноманітних алгоритмів та підходів, спрямованих на вирішення задачі нечіткого пошуку в тексті. Вони базуються на різних принципах та методах, таких як розподільні моделі, імовірнісні методи, статистичні аналізи, машинне навчання та інші.

Одним з таких рішень є алгоритм нечіткого пошуку в тексті з використання таблиці подібностей символів. Поєднавши потужність алгоритму нечіткого пошуку з ретельно розробленою таблицею подібності символів, даний метод пропонує підвищену точність, ефективність і універсальність у системах пошуку інформації.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА МОЖЛИВИХ РІШЕНЬ.

Метою даної дипломної роботи є дослідження сучасних алгоритмів нечіткого пошуку, а також розробка алгоритму нечіткого пошуку з використанням таблиці подібності символів.

Для досягнення цієї мети необхідно вирішити наступні задачі:

- Аналіз сучасних алгоритмів нечіткого пошуку;
- Дослідження алгоритму нечіткого пошуку з використанням таблиці подібності символів;
- Реалізація алгоритму нечіткого пошуку з використанням таблиці подібності символів;
- Розробка таблиці подібності символів;
- Тестування розробленого алгоритму;
- Порівняння швидкодії роботи алгоритму нечіткого пошуку з використанням таблиці подібності символів і без.

2. АНАЛІЗ СУЧАСНИХ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ.

Алгоритми нечіткого пошуку розроблені для отримання приблизних відповідностей для заданого запиту в тексті, навіть якщо точної відповіді не існує. Розглянемо, деякі з найпопулярніших алгоритмів нечіткого пошуку:

- Алгоритм Дамерау-Левенштейна;
- Алгоритм Джаро-Вінклера;
- Алгоритм N-грам;
- Алгоритм Bitap;
- Алгоритм SoundEx.

2.1. Алгоритм Дамерау-Левенштейна.

Алгоритм Дамерау-Левенштейна є один із найбільш часто використовуваних підходів. Основна ідея нечіткого пошуку полягає у вимірюванні відстані редагування кожного рядка (слова) із вихідного тексту та заданим рядком, використовуючи метрику Дамерау-Левенштейна.

Відстань Дамерау-Левенштейна – це міра відмінності двох послідовностей символів (рядків). Загалом, дана відстань між двома рядками обчислюється як мінімальна кількість операцій вставки, видалення, заміни і транспозиції (перестановки двох сусідніх символів) необхідних для перетворення однієї послідовності в іншу [1].

Відстань Дамерау-Левенштейна є модифікацією класичною відстані Левенштейна, до якої в додаток трьох класичних операцій Дамерау додав операцію транспозиції. Свою пропозицію він пояснив у статті [2], в якій він заявив, що під час дослідження орфографічних помилок для інформаційно-пошукової системи понад 80% були результатом помилки, яка належала саме до одного з чотирьох типів (вставка, видалення, заміна і транспозиція) [1].

Для розрахунку відстані Дамерау–Левенштейна найчастіше застосовують алгоритм динамічного програмування, в якому використовується матриця розміром (n + 1) * (m + 1), де n і m - довжини порівнюваних рядків. Окрім цього вартість операцій вилучення, заміни, вставки і транспозиції вважається однаковою і дорівнює 1, проте при реалізації алгоритму її можна замінити. Щоби розрахувати дану метрику між двома рядками a і b, потрібно сконструювати матрицю, використовуючи рекурсивне рівняння (2.1.1).

$$d_{a,b}(i,j) = \\ min\begin{cases} 0 \quad if\ i = j = 0, \\ d_{a,b}(i-1,j) + 1 \quad if\ i > 0, \\ d_{a,b}(i,j-1) + 1 \quad if\ j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \quad if\ i,j > 0, \\ d_{a,b}(i-2,j-2) + 1_{(a_i \neq b_j)}\ if\ i,j > 1\ and\ a_i = b_{j-1}\ and\ a_{i-1} = b_j, \end{cases} \tag{2.1.1}$$

де $1_{(a_i \neq b_j)}$ – це індикаторна функція, яка дорівнює 0, коли $a_i = b_j$, і дорівнює 1 в протилежному випадку [1].

Кожен рекурсивний виклик відповідає одному з випадків, охоплених відстанню Дамерау–Левенштейна:

- $d_{a,b}(i-1,j) + 1$ відповідає видаленню символа (від a в b);
- $d_{a,b}(i,j-1) + 1$ відповідає вставці символа (від a в b);
- $d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)}$ відповідає за збіг символів або їх невідповідність, залежно від того чи відповідні символи однакові;
- $d_{a,b}(i-2,j-2) + 1_{(a_i \neq b_j)}$ відповідає за транспозицію між двома послідовними символами.

В результаті відстань Дамерау–Левенштейна є значенням функції $d_{a,b}(|a|,|b|)$, де $|a|$ і $|b|$ – це довжини відповідних рядків a і b [1].

Задавши певне порогове значення для метрики Дамерау–Левенштейна, та порівнявши всі слова з вхідного тексту із заданим словом, можна отримати результат нечіткого пошуку, який буде включати всі слова, в яких відстань Дамерау–Левенштейна, менша за порогову.

Недоліком, використання лише відстані Дамерау–Левенштейна для пошуку в тексті є те, що даний алгоритм не враховує семантику або контексту тексту, що може призводити до неточних результатів.

2.2. Алгоритм Джаро-Вінклера.

Даний алгоритм є подібним до алгоритму Дамерау–Левенштейна, проте при порівняні послідовностей тексту використовується метрика Джаро–Вінклера. Відстань Джаро–Вінклера — це рядкова метрика, яка вимірює відстань редагування між двома послідовностями. Значення відстані варіюється від 0 до 1, де 0 означає точну відповідність, а 1 означає відсутність подібності [3]. Дана метрика фактично визначена в термінах подібності Джаро–Вінклера, тому відстань визначається як інверсія цього значення (відстань = 1 - подібність).

Дана метрика була розроблена Вільямом Е. Вінклером у 1990 році як розширення подібності Джаро. Подібність Джаро-Вінклера враховує як рівність відповідних символів, так і порядок цих символів у рядках.

Для визначення подібності Джаро–Вінклера необхідно спочатку розрахувати подібність Джаро, яка для двох рядків визначається за формулою (2.2.1)

$$sim_j = \begin{cases} 0, & if\ m = 0, \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right), & if\ m \neq 0, \end{cases} \quad (2.2.1)$$

де:

- m - кількість співпадінь символів:
  - Два символи від s_1 і s_2 відповідно, вважаються співпадінням лише в тому випадку, якщо вони однакові і розташовані не далі, ніж $\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor - 1$ один від одного;
- $|s_1|, |s_2|$ - довжина першого та другого рядка відповідно;
- t - кількість транспозицій:

- Обчислюється як кількість відповідних символів (але в іншому порядку послідовності), поділена на 2 [3].

Окрім подібності Джаро, для того, щоб розрахувати подібність Джаро–Вінклера, використовується також оцінка префікса p, завдяки чому ми отримуємо більш сприятливі оцінки рядкам, які з самого початку відповідають заданій довжині префікса l. Наприклад дано два рядки s_1 і s_2. Подібність Джаро–Вінклера sim_w обчислюється наступним чином [3]:

$$sim_w = sim_j + lp(1 - sim_j) \quad (2.2.2)$$

де:

- sim_j – подібність Джаро для рядків s_1 і s_2;
- l – довжина загального префіксу на початку рядка — максимум до 4 символів;
- p є сталим коефіцієнтом масштабування того, наскільки оцінка коригується вгору за наявність загальних префіксів. p не повинен перевищувати 0,25 (тобто 1/4, причому 4 - це максимальна довжина префікса, що розглядається), інакше подібність може стати більшою за 1. Стандартним значенням цієї константи у роботі Вінклера є p = 0.1 [3].

Відстань Джаро-Вінклера є особливо ефективною для порівняння коротких і середніх рядків, таких як імена або адреси, і вона зазвичай використовується в задачах зв’язування записів і дедуплікації даних. Проте даний алгоритм є неефективним при порівнянні довгих рядків, оскільки при збільшенні довжин, його продуктивність значно погіршується.

2.3. Алгоритм N-грам.

У всіх алгоритмах нечіткого пошуку, які ми розглянули раніше, в них ми розглядали відмінність рядків, порівнюючи кожен символ окремо. Один із

способів розшири це поняття – це дозволи порівнювати одночасно декілька символів, або так звані N-грами.

Алгоритм N-грам – це алгоритм зіставлення рядків, який використовується для порівняння двох рядків і визначення їх подібності. Для цього потрібно розділити кожен рядок на підрядки довжиною N (де N – додатне ціле число), а потім алгоритм порівнює частоти входжень N-грам у обох текстах та обчислює показник подібності на основі перекриття або схожості цих частот.

Одиницею порівняння (n-грам) може бути послідовність символів будь-якої довжини, від 1 (порівняння наборів символів, з яких складається слово) до довжини найкоротшого слова в тексті (точне порівняння слів). Чим більша довжина n-грами, тим жорсткіші умови відбору подібних слів, і тим менша “нечіткість” метода [4].

Цьому методу притаманний один істотний недолік: одна помилка в середині слова спотворює n N-грам, що згубно впливає на результат порівняння [4].

Метод N-грам є єдиний з розглянутих алгоритмів, який може визначити як подібні пару слів, одне з яких складається з двох основ, а інше – з них же, але в іншому порядку. Прикладом таких пар можуть бути «графолог» і «логограф» чи «логотип» і «типологія». Ця властивість даного методу вважається важливою перевагою, серед інших алгоритмів [4].

Загальна схема алгоритму подібності N-грам:

1. Попередня обробка:
   - Розбивка текстів на окремі слова або символи, залежно від бажаного рівня деталізації.
   - Вилучення стоп-слів (поширених слів, таких як "the", "is" і т. д.) та проведення необхідної нормалізації або очищення тексту.
2. Генерація n-грам:
   - Створення всіх можливих N-грам (послідовностей з n слів або символів) для кожного тексту.

3. Створення індексів N-грам:
   - Індекс N-грам створюється для зберігання n-грам та їхніх відповідних позицій у вихідних рядках. Цей індекс дозволяє ефективно шукати та порівнювати N-грами під час процесу зіставлення.
4. Обчислення частот:
   - Підрахунок кількості входжень кожного унікального n-граму з рядка запиту в кожному вихідному рядку. Ці частоти зберігаються у структурах даних, таких як словники або вектори частот.
5. Обчислення подібності:
   - Обчислення показника подібності між двома текстами на основі частот n-грам. Можна використовувати різні показники подібності, такі як подібність Жаккара, косинусна подібність або коефіцієнт Дайса.
   - Ці показники подібності, як правило, включають порівняння перетину та об'єднання множин n-грамів або використання векторних обчислень.
6. Формування результату:
   - Отриману оцінку подібності можна використовувати для ранжування або порівняння подібності кількох текстів.
   - Вищий показник подібності вказує на більше перекриття частот n-грамів, а отже тексти більш схожі.

Алгоритм N-грам є швидким та ефективним методом, для порівняння великих обсягів текстових даних. Даний алгоритм може використовуватися для виявлення плагіату та, поєднавши з іншими алгоритмами зіставлення, його можна використовувати для зіставлення записів даних і дедуплікації.

2.4. Алгоритм Bitap.

Алгоритм Bitap (також відомий як алгоритм зсуву чи алгоритм Баези-Йетса-Гоннета) – це алгоритм для наближеного зіставлення рядків. Даний алгоритм перевіряє, чи містить заданий текст підрядок, який «приблизно дорівнює» заданому шаблону, де приблизна рівність визначається в термінах відстані Левенштейна – якщо підрядок і шаблон знаходяться на заданій відстані k один від одного, тоді алгоритм вважає їх рівноправними [5].

Вперше ідею даного алгоритму запропонували Рікардо Баеза-Йетс і Гастон Гоннет, опублікувавши відповідну статтю в 1992 році. Проте дана версія алгоритму перевіряли лише заміну символів і фактично обчислює відстань Хемінга. Але пізніше Сунь Ву та Уді Манбер запропонували модифікацію даного алгоритму для розрахунку відстані Левенштейна, тобто добавили підтримку операцій вставка і видалення.

Робота алгоритму заключається в побудові шаблону, який потім буде порівнюватися з більшою частиною тексту, щоб знайти приблизні збіги. Ключова ідея алгоритму полягає у використанні побітових операцій і побітового зсуву для швидкого порівняння шаблону з текстом.

Нехай S0=“ababc” є шаблоном, який потрібно зіставити з рядком S1 = “abdababababc”, тоді загальна схема алгоритму Bitap буде мати вигляд [6]:

1. Попередня обробка:
   - Визначити всі унікальні символи рядків. Для S0 і S1 це a, b, c, d.
   - Побудувати бітову маску для кожного символу. Для цього нам необхідно перевернути шаблон S0 і для бітової маски певного символу потрібно розмістити 0 там, де він зустрічається в шаблоні, в іншому випадку – 1. Так для символу ‘a’, оскільки він зустрічається на 3-ій і 5-ій позиції шаблону, то там ми ставимо 0, а в інших місцях – 1.

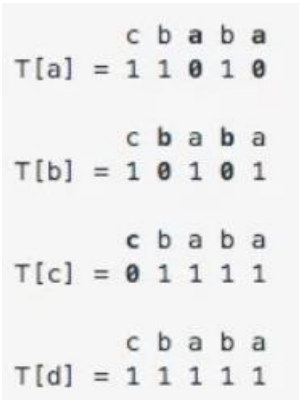

Рисунок 2.4.1 – Бітові маски для символів a, b, c, d.

2. Визначити початковий стан бітового шаблону довжини |S0|, кожен біт якого є 1. У нашому випадку це буде S=11111.
3. Проітерувати по кожному символу рядка S1, в якому ми шукаємо патерн, та виконуємо наступні дії:
   - Робимо зсув 0 в найменший біт шаблону S. В нашому випадку це 5-ий біт справа.
   - Проводимо побітову операцію: S = S | T[x].

Наприклад, для 1-ого символу S1, ‘a’:

1. Лівий зсув 0 в S: 11111 →11110;
2. 11010 | 11110=11110

Для всього вхідного рядка S1, результат bitap алгоритму буде мати вигляд [6] (рис. 2.4.2).

text :	a	b	d	a	b	a	b	a	b	c
T[x] :	11010	10101	11111	11010	10101	11010	10101	11010	10101	01111
state :	11110	11101	11111	11110	11101	11010	10101	11010	10101	01111

Рисунок 2.4.2 – Результат bitap алгоритму для шаблону “ababc” і рядку “abdababababc”, в якому здійснюється пошук.

Для того, щоб виявити повний збіг шаблона із підрядком вхідного текста, необхідно, щоб під час ітерації по тексту 0 досяг найвищого біта в бітовому шаблоні стану S. В цьому випадку результуючий підрядок буде

закінчуватися символом, під час ітерації якого шаблон стану досяг 0. Так в нашому прикладі це останній символ ‘c’ [6].

Для виявлення нечіткого збігу, необхідно спостерігати за 0 у різних позиціях стану S. Якщо 0 знаходиться на 4 місці, ми знайшли 4/5 символів у тій же послідовності, що й у S0. Так само для 0 в інших місцях [6].

Перевагою цього алгоритму перед іншими методами є використання побітових операцій. Це робить його дуже швидким. Також іще одним плюсом алгоритму є те, що його легко імплементувати, проте він є досить складним для розуміння.

2.4. Алгоритм SoundEx.

Soundex є представником фонетичних методів, які полягають в пошуку подібних слів на основі їх звучання. Даний алгоритм полягає в перетворенні слів у стандартизований код на основі їхньої вимови. При пошуку порівнюються саме ці фонетичні коди. Він був розроблений на початку 20-го століття і з тих пір широко використовується у різних сферах. Наприклад, у генеалогії для дослідження сімейної історії, а саме допомагає знайти альтернативні варіанти прізвищ, у випадках, коли з часом варіанти написання прізвищ видозмінюються, проте звучання залишається таким же.

Алгоритм працює наступним чином:

1. Спочатку потрібно видалити всі початкові і кінцеві пропуски і перетворити літери слова у один регістр: маленькі чи великі.
2. Збережіть першу літеру слова і відкиньте всі наступні входження таких літер: 'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'
3. Надайте числовий код кожній літерам, що залишилися, виходячи з їх звучання:
   - Літерам 'B', 'F', 'P', і 'V' надається номер 1;
   - Літерам 'C', 'G', 'J', 'K', 'Q', 'S', 'X', 'Z' - номер 2;
   - Літерам 'D' і 'T' - номер 3;

- Літері 'L' – 4;
- Літерам 'M' і 'N' – 5;
- Літері 'R' – 6.

4. Об'єднайте першу літеру слова з числовими кодами, отриманими на попередньому кроці.
5. Видаліть послідовні входження одного й того ж номера, за винятком першого входження.
6. Якщо отриманий код має менше чотирьох символів, додайте до нього зліва нулі. Якщо він має більше чотирьох символів, обріжте його.
7. Остаточний код Soundex - це отримана чотирьохсимвольна строка.

В результаті порівнюючи коди Soundex для різних слів або імен, можна ідентифікувати ті, які мають подібну вимову, навіть якщо вони мають різні варіанти написання.

2.5. Висновки до розділу.

Нечіткий пошук є важливим інструментом для ефективного пошуку інформації в сучасному світі. Існує безліч критеріїв для оцінки результатів. Тому існує досить багато різноманітних алгоритмів та підходів, спрямованих на вирішення цієї задачі. У кожного з цих алгоритмів є свої переваги й недоліки, що зумовлює їх застосування в різних сферах, користуючись їхніми позитивними сторонами. Так, наприклад алгоритм Дамерау-Левенштейна в основному використовується в програмах, де виявлення та виправлення друкарських помилок або орфографічних помилок є критично важливими. Алгоритм N-gram сьогодні в основному використовується в NLP. Алгоритм Bitap знаходить своє використання в завданнях приблизного зіставлення рядків і розпізнавання шаблонів. Алгоритм SoundEx в основному застосовують в системах фонетичного зіставлення та пошуку.

3. АЛГОРИТМ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ З ВИКОРИСТАННЯМ ТАБЛИЦІ ПОДІБНОСТІ.

Алгоритм нечіткого пошуку в тексті з використанням таблиці подібності є модифікацією алгоритму Дамерау-Левенштейна, в якому для порівняння символів також використовується таблиця подібності символів. Даний метод зазвичай використовується для пошуку приблизних відповідностей для заданого пошукового запиту в тексті чи базі даних, враховуючи можливі друкарські помилки чи орфографічні помилки. Даний алгоритм враховує як і візуальну, семантичну подібність окремих символів, так і операції вставки, видалення, заміни і транспозиції для вимірювання загальної подібності між рядками.

Загальна схема алгоритму нечіткого пошуку з використанням таблиці подібності має наступний вигляд:

1. Попередня обробка:
   - Створити таблиці подібності символів, яка визначає оцінку подібності між кожною парою символів на основі їхньої візуальної чи семантичної подібності. Наприклад, символи ‘а’ та ‘ä’ будуть мати вищу оцінку схожості порівняно з ‘а’ та ‘е’.
2. Обробка запиту:
   - Отримати рядок запиту.
   - Провести ітерацію по тексту або базі даних, щоб порівняти кожен рядок (запис) із пошуковим запитом.
3. Нечітке порівняння рядків:
   - Розрахувати відстань Дамерау-Левенштейна: для цього застосувати алгоритм димічного програмування, а саме побудувати матрицю розміром (n + 1) * (m + 1), де n і m - довжини порівнюваних рядків. Кожному елементу матриці присвоюється ціна операцій (вставка, видалення, заміна і транспозиція), які необхідно виконати, щоб відповідні рядки були рівні.

- Застосувати таблицю подібності символів: для кожної пари символів, які порівнюються під час обчислення відстані, потрібно знайти оцінку їхньої подібності, використовуючи при цьому таблицю подібності символів. Отриману оцінку необхідно врахувати при розрахунку кожного елемента матриці.
- Порівняти відстані для фільтрації всіх збігів: необхідно порівняти обчислені відстані з порогом подібності. Якщо відстань нижче порогового значення, то це збіг, а отже потрібно додати відповідний запис у результати пошуку.

4. Результат:
   - Надати результати пошуку: повернути рядки, що збігаються з пошуковим записом, із пов’язаною інформацією, а саме відстанню подібності цих рядків.

Завдяки таблиці подібності символів, ми можемо більш детально контролювати порівняння подібності символів, дозволяючи алгоритму обробляти випадки, коли певні символи більш візуально або семантично схожі, ніж інші.

3.1. Модифікація алгоритму Дамерау-Левенштейна.

Як вже згадувалося раніше алгоритм Дамерау-Левенштейна є один із найпопулярніших підходів для нечіткого пошуку в тексті. Його основна ідея полягає в ітерації по тексті чи базі даних, розраховуючи при цьому відстань Дамерау-Левенштейна (відстань редагування) між пошуковим запитом і кожним рядком чи словом тексту.

Модифікація алгоритму Дамерау-Левенштейна проявляється в тому, що для розрахунку відстані редагування під час порівняння символів, також застосовується таблиця подібності символів. Розрахунок відстанні редагування завдяки динамічного програмування буде мати наступні кроки[8]:

1. Створити матрицю (наприклад *d[][]*) розміром *(m+1) * (n+1)*, де *m* і *n* – це довжина рядків [8].
2. Будемо вважати, що ціна вставки, видалення, заміни і транспозиції однакові і дорівнюють 1.
3. Проініціалізувати перший рядок і стовпець матриці, відповідно, *0, 1, 2, ..., n і 0, 1, 2, ..., m*.
4. Визначити відстань редагування рядка для кожної клітинки матриці:
   – Оскільки в даному алгоритмі також використовується таблиця подібності символів, необхідно спочатку визначити подібність символів, які розташовуються відповідно у двох рядках, позначимо її *compareCharCost*. Вона буде дорівнювати *0* - якщо символи однакові, певному значенню *j (0 < j < 1)*, яке можна отримати з таблиці подібності символів, або *1* – якщо, використовуючи таблицю, дані символи не можна порівняти.
   – Визначити відстань редагування: *d[i][j] = min(d[i-1][j-1] + compareCharCost, d[i-1][j] + 1, d[i][j-1] + 1)*.
   – Якщо виконується умова $a_i = b_{j-1}$ і $a_{i-1} = b_j$, то також необхідно врахувати можливу транспозицію символів: *d[i][j] = min(d[i][j], d[i-2][j-2] + 1)* [8].

Підсумовуючи модифікація формули (2.1.1) буде мати вигляд:

$$d_{a,b}(i,j) = min\begin{cases} 0 \;\; if\; i = j = 0, \\ d_{a,b}(i-1,j) + 1 \;\; if\; i > 0, \\ d_{a,b}(i,j-1) + 1 \;\; if\; j > 0, \\ d_{a,b}(i-1,j-1) + compareCharCost \;\; if\; i,j > 0, \\ d_{a,b}(i-2,j-2) + 1 \; if\; i,j > 1 \; and \; a_i = b_{j-1} \; and \; a_{i-1} = b_j, \end{cases} \quad (3.1.1)$$

5. В результаті відстань редагування двох рядків буде дорівнювати *d[m][n]* (число, яке розташовується в нижній правій клітинці матриці) [8].

Задавши певне порогове значення та порівнявши всі слова з вхідного тексту із пошуковим запитом, використовуючи модифіковану відстань Дамерау-Левенштейна, можна отримати результати нечіткого пошуку в тексті

з використанням таблиці подібності. Він буде включати всі слова, яких відстань редагування, менша за порогову.

3.2. Таблиця подібності символів.

Таблиця подібності символів, також відома як таблиця відображення символів, є довідковою таблицею, яка чисельно визначає подібність між символами. Завдяки ній можна визначити еквівалентні символи з різних мов, чи просто візуально подібні символи.

Мета таблиці подібності символів полягає у тому, що вона є важливим інструментом при транслітерації, транскрипції або конвертації між різними системами письма. Вона дозволяє комп'ютерним системам знайти подібні символи у випадках, коли маємо справу з багатомовними системами. Наприклад, вона може допомогти перетворити слово або фразу, написану однією мовою на іншу, або допомогти в пошуку відповідності символів для обробки даних, особливо в нечіткому пошуку в тексті (наприклад у випадках, коли текст написаний на одній мові, а пошуковий запит на іншій).

В залежності від конкретної програми таблиця подібності символів може включати різні символи. Зазвичай це символи з різних алфавітів, таких як латиниця, грецька, кирилиця та багато інших чи символи з діакритичними знаками, які використовуються в різних мовах. Взагалі таблиця також може містити спеціальні символи для певного домену, такі як символи нуклеотидів чи символи музичних нот.

Таблиці подібності символів можуть сильно варіюватися залежно від конкретної задачі поставленої перед неї і від методу її імплементації. Деякі таблиці фокусуються на певній парі мов, тоді як інші можуть охоплювати ширший спектр. Також таблиця може фокусуватися лише на візуально подібні символи однієї мови або включати в себе можливі друкарські помилки.

Найбільш поширеною структурою таблиці подібності символів є матриця, де кожен рядок і стовпець представляють символ, а в комірках

матриці зберігається оцінка подібності між парами відповідних символів. Оцінка подібності може вказувати на те, наскільки тісно пов’язані два символи з точки зору їхньої форми, звуку чи інших відповідних характеристик.

Значення в таблиці подібності зазвичай варіюються від 0 до 1, де 1 означає відсутність подібності, а 0 означає ідеальний збіг. Таблиця має бути симетричною, оскільки подібність між символом A і символом B така ж, як подібність між символом B і символом A.

Загалом таблиці подібності символів служать цінним довідником не лише для розробників програмного забезпечення, а й для лінгвістів, перекладачів, дизайнерів шрифтів та інших професій, які працюють із завданнями обробки багатомовного тексту або перетворення символів.

3.3. Висновки до розділу.

Підсумовуючи, можна сказати, що, використовуючи таблиці подібності символів, можна значно покращити точність алгоритму Дамерау-Левенштейна. Даний модифікований алгоритм особливо необхідний у випадках, коли текст, в якому відбувається пошук написаний однією мовою, а користувач робить запит іншою, і тому він в пошуковому запиті може замінювати символи однієї мови на семантично схожі іншої.

4. РЕАЛІЗАЦІЯ АЛГОРИТМУ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ З ВИКОРИСТАННЯМ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ.

Сьогодні існує безліч різних реалізацій нечіткого пошуку в тексті, проте більшість з них не враховує графічну чи семантичну подібність символів. Саме в цій проблемі нам допоможуть можливості нечіткого пошуку з використанням таблиці подібності символів. Концепція таблиці подібності символів полягає в присвоєнні балів подібності парам символів на основі їхньої схожості. Ці показники фіксують ступінь подібності між символами, що дає нам змогу кількісно визначити їхню подібність і виконати їх порівняння.

4.1. Реалізація таблиці подібності символів.

Таблиця подібності символів є довідковою таблицею, яка чисельно визначає подібність двох символів. Дана таблиця надає набір символів разом із пов’язаними з ними вагами або оцінками, які базуються на основі схожості даних символів. Ці оцінки можуть бути використаними для нечіткого пошуку, щоби визначити рівень подібності символів і на основі цього формувати результат пошуку, який буде включати різні варіації рядків.

4.1.1. Структура таблиці подібності символів.

Найбільш поширеною структурою таблиці подібності символів є матриця, де кожен рядок і стовпець представляють символ, а в комірках матриці зберігається оцінка подібності між парами відповідних символів. Оцінка подібності може вказувати на те, наскільки тісно пов’язані два символи з точки зору їхньої форми, звуку чи інших відповідних характеристик. Така таблиця має бути симетричною, оскільки подібність між символом A і символом B така ж, як подібність між символом B і символом A. Проте

недоліком даної структури є складність пошуку комірки, яка містить вагу подібності для певних символів.

Тому була вигадана наступна структура, для реалізації таблиці подібності символів (рис. 4.1.1.1).

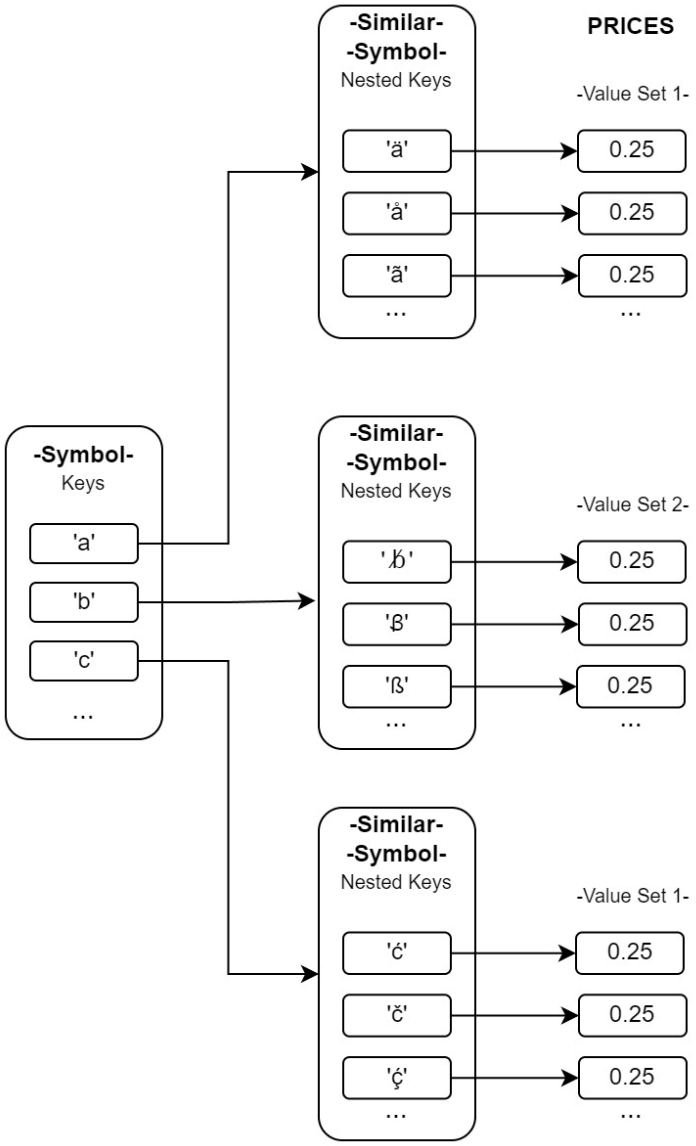

Рисунок 4.1.1.1. – Структура таблиці подібності символів.

Дана структура складається з двох словників. В першому словнику ключом є символ, для якого ми будемо шукати йому подібні, а значенням – другий словник. В другому словнику ключом є подібні символи до ключу з першого словника, а значенням вага їх подібності. Дана структура буде зберігати властивість симетричності, не дивлячись на те, що це призведе до

зберігання дублікатів даних. Асимптотична складність знаходження символу у дані структурі буде константною O(1).

4.1.2. Реалізація таблиці подібності символів.

Реалізована таблиця подібності буде складатися з трьох різних категорій подібності символів.

1. Перша категорія:

В дану категорію будуть входити семантично схожі символи з різних мов. За основу було взято джерело [9]. Оскільки це електронний ресурс, то найлегше та найефективніше його проаналізувати, для побудови таблиці подібності, буде за допомогою мови програмування Python. Саму таблицю подібності будемо зберігати в форматі JSON, оскільки завдяки цьому ми зможемо легко серіалізувати дані таблиці, а потім также легко десеріалізувати, щоби застосувати в нечіткому пошуку.

Наведемо фрагмент лістингу програми для створення першої категорії таблиці (рис. 4.1.2.1).

```
# SIMILAR CHARS WITHOUT CAPITAL
for sec in sym_table:
    h = sec.find('h3')
    letter = h.text[-2].lower()

    similar_group = set(letter)
    sim_letters = sec.find_all('a')
    for sl in sim_letters:
        tag = sl['data-template']
        data_dict = json.loads(tag)
        sym = data_dict["symbol"]

        if sym.islower():
            similar_group.add(sym)
        else:
            similar_group.add(sym.lower())

    for lt1 in similar_group:
        if lt1 not in data["data"].keys():
            data["data"][lt1] = dict()

        for lt2 in similar_group:
            if lt1 != lt2:
                data["data"][lt1][lt2] = COST_DICT["SIMILAR_CHAR_COST"]
```

Рисунок 4.1.2.1. – Фрагмент лістингу програми для створення першої категорії символів в таблиці подібності.

В даному фрагменті ми аналізуємо джерело [9]. Спочатку ми проходимося по усім спискам груп символів. Заголовком даних списків є символ англійського алфавіту, а вмістом в даних блоках є подібні до нього символи. Так для кожного символу ми знаходимо йому подібні і добавляємо їх всі в множину *similar_group*. Далі ми проходимо по цій множині використовуючи вкладений цикл і в результуючому словнику зберігаємо подібні символи відповідно структурі таблиці подібності (symbol→similar symbol→price). Тобто в результаті будуть зберігатися всі символи з *similar_group*, які в свою чергу в якості подібних йому символів будуть зберігати всі інші символи з *similar_group*. Завдяки цьому ми зберігаємо властивість симетричності таблиці подібності.

2. Друга категорія:

Дана категорія подібних символів буде містити можливі друкарські помилки для англійського алфавіту. Кожна група подібних символів в даній категорії буде складатися з символу та всіх можливих помилок, а саме всіх оточуючих символів на клавіатурі. Наведемо приклад, як це буде реалізовано для першого рядка клавіатури (рис. 4.1.2.2).

```
# KEYBOARD MISTAKE
keyboard = [["q", "w", "e", "r", "t", "y", "u", "i", "o", "p"],
            ["a", "s", "d", "f", "g", "h", "j", "k", "l"],
            ["z", "x", "c", "v", "b", "n", "m"]]


def add_to_chars(char, *ch_set):
    for ch in ch_set:
        data["data"][char][ch] = COST_DICT["KEYBOARD_MISTAKE_COST"]


for ind, value in enumerate(keyboard[0]):
    data_set = []
    if ind == 0:
        data_set.extend((keyboard[0][ind + 1], keyboard[1][ind]))
```

```
    elif ind == len(keyboard[0]) - 1:
        data_set.extend((keyboard[0][ind - 1], keyboard[1][ind - 1]))

    else:
        data_set.extend((keyboard[0][ind - 1], keyboard[0][ind + 1],
                         keyboard[1][ind - 1], keyboard[1][ind]))
    add_to_chars(value, *data_set)
```

Рисунок 4.1.2.2. – Фрагмент лістингу програми для створення другої категорії символів в таблиці подібності.

В даному фрагменті ми створюємо масив *keyboard*, який імітує розташування всіх символів на клавіатурі. Далі ми проходимося по першому рядку, знаходимо всі оточуючі клавіши та формуємо нову групу подібних символів, яку додаємо до результату. Для пройдених символів додаємо в результуючий словник нові подібні символи та їхню вагу схожості. Таким же чином проходимося для інших двох рядків.

3. Третя категорія:

Третя категорія подібних символів складається з візуально подібних символів англійкого алфавіту (рис. 4.1.2.3).

```
# VISUAL SIMILAR CHAR
def add_similar_ch(chs):
    for ch1 in chs:
        for ch2 in chs:
            if ch1 == ch2:
                continue
            if ch2 not in data["data"][ch1].keys():
                data["data"][ch1][ch2] = COST_DICT["VISUAL_SIMILAR_CHAR_COST"]
            elif COST_DICT["VISUAL_SIMILAR_CHAR_COST"] < data["data"][ch1][ch2]:
                data["data"][ch1][ch2] = COST_DICT["VISUAL_SIMILAR_CHAR_COST"]


add_similar_ch(("g", "q"))
```

Рисунок 4.1.2.3. – Фрагмент лістингу програми для створення третьої категорії символів в таблиці подібності.

В даному фрагменті було створено функцію, яка приймає в параметри масив подібних символів, *chs*. Далі завдяки вкладеному циклу ми проходимося по списку двічі, і зберігаємо в результуючому словнику всі символи, які в свою, в якості подібних символів, зберігають всі інші символи цього ж

словника. Якщо в результуючому словнику вже була певна пара подібних символів, яка може знову зустрітися в *chs*, то ми зберігаємо меншу вагу подібності.

Далі ми результуючий словник *data* перетворюємо в об’єкт JSON за допомогою бібліотеки json і зберігаємо його в файлі. Тепер таблиця подібності символів готова для застосування в нечіткому пошуку. Повний лістниг коду наявний в Додатку А.

4.2. Реалізація алгоритму нечіткого пошуку.

Алгоритм нечіткого пошуку в тексті з використанням таблиці подібності базується на алгоритмі Дамерау-Левенштейна та таблиці подібності символів. Для його реалізації було обрано мову C++, оскільки вони має зручні вбудовані засоби для роботи з символами UTF-16.

Реалізацію алгоритму можна поділити на дві складові:

1. Обробка таблиці подібності символів:

Спочатку потрібно завантажити JSON файл, який містить нашу таблицю подібності символів. Для цього використаємо бібліотеку RapidJSON. Таблиця буде зберігатися в двох структурах даних (рис. 4.2.1).

```
std::unordered_map<wchar_t, std::unordered_map<wchar_t, float>> table;
std::unordered_map<std::string, float> cost;
```

Рисунок 4.2.1. – Структури даних для зберігання таблиці.

Перша структура *table* – це unordered_map, ключом, якого є символ, а значення інша unordered_map, яка зберігає подібні символи та вагу подібності. Дана структура є головною і зберігає всі дані таблиці. Друга структура cost – це unordered_map, яка зберігає найменування типу подібності та відповідну вагу подібності (вона необхідна нам для майбутнього тестування таблиці).

Наведемо приклад, як ми завантажуємо дані таблиці (рис. 4.2.2).

```
std::ifstream ifs( Str: path,  Mode: std::ios::binary);
if (ifs.fail()) {...}

rapidjson::IStreamWrapper isw( &: ifs);
rapidjson::AutoUTFInputStream<unsigned , rapidjson::IStreamWrapper> eis( &: isw);

WDocument document;
document.ParseStream<0, rapidjson::AutoUTF<unsigned> >( &: eis);
if (!document.IsObject()){...}

//PARSE DATA
for (auto key :GenericMemberIterator<...>::GenericMemberIterator<...>  = document.MemberBegin(); key != document.MemberEnd(); key++){
    if(key->name == L"data"){
        for (auto wch :GenericMemberIterator<...>::GenericMemberIterator<...>  = key->value.MemberBegin(); wch != key->value.MemberEnd(); wch++){
            for (auto simWch :GenericMemberIterator<...>::GenericMemberIterator<...>  = wch->value.MemberBegin(); simWch != wch->value.MemberEnd(); simWch++){
                auto valueWch :UTF16LE::wchar_t  = wch->name.GetString()[0];
                auto valueSimWch :UTF16LE::wchar_t  = simWch->name.GetString()[0];
                auto price :float  = simWch->value.GetFloat();

                table[valueWch][valueSimWch] = price;
            }
        }
        break;
    }
}
```

Рисунок 4.2.2. – Фрагмент лістингу програми для обробки даних таблиці.

В даному фрагменті ми спочатку відкриваємо JSON файл, та створюємо об’єкт, який буде читати дані з цього файлу. Далі ми проходимося по всьому файлу, і шукаємо об’єкт, ключ якого є “data”. Далі ми проходимося по цьому об’єкту. Ключами об’єкту будуть символи, а значенням – подібні йому символи та ваги. Далі ми зберігаємо відповідні дані в *table*. Повний лістинг коду наявний в Додатку Б.

2. Реалізація алгоритму Дамерау-Левенштейна:

Наведемо лістинг коду даного алгоритму (рис. 4.2.3).

```
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,  j: 0) = i;

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

// building matrix
for (size_t i = 1; i < aLen; ++i)
{
    for (size_t j = 1; j < bLen; ++j)
    {
        float compareCharCost = compareWchar( c1: a[i - 1],  c2: b[j - 1], table);

        // replace, insert, delete
        matr.at(i, j) = std::min( llist: {matr.at( i: i - 1,  j: j - 1) + compareCharCost,
                                  matr.at(i,  j: j - 1) + insertCost,
                                  matr.at( i: 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: i - 2,  j: j - 2) + transpositionCost);
    }
}

return  matr.at( i: aLen - 1,  j: bLen - 1);
```

Рисунок 4.2.3. – Реалізація алгоритму Дамерау-Левенштейна.

В даному коді ми реалізовуємо всі кроки, які були описані в розділі 3, а саме будуємо матрицю розміром (aLen + 1) * (bLen + 1), де aLen і bLen - довжини порівнюваних рядків. Далі заповнюємо матрицю використавши описану в попередньому розділі функцію (3.1.1). І в результаті повертаємо відстань редагування між двома рядками, яка розташована в нижній правій комірки матриці.

4.3. Висновки до розділу.

Отже, в ході реалізації алгоритму нечіткого пошуку з використанням таблиці подібності символів, спочатку було створено саму таблицю. Таблиця включає в собі три можливих різновидів подібності: семантична подібність, візуальна подібність та можлива друкарська помилка. Далі було реалізовано функцію для завантаження таблиці в пам’ять і було реалізовано алгоритм, для розрахунку модифікованої відстані Дамерау-Левенштейна за формулою (3.1.1).

5. ТЕСТУВАННЯ АЛГОРИТМУ НЕЧІТКОГО ПОШУКУ.

У сучасному світі різне програмне забезпечення стає все більш складнішим в реалізації, більш громіздким. Більшість сучасних програм складаються з багатьох різних компонентів, які повинні взаємодіяти один з одним, і певна помилка в одному компоненті може призвести до краху всієї системи. Тому тестування програмного забезпечення грає першочергову роль в життєвому циклі розробки ПЗ.

Для проведення тестування розробленого алгоритму було обрано одну з найпопулярніших бібліотек GoogleTest. GoogleTest – це фреймворк тестування для мови C++, який розроблений командою Testing Technology з урахуванням конкретних вимог і обмежень Google [10]. Дана бібліотека надає великий набір різних функцій і утиліт, які значно допомагають в написанні та організації тестів. Також GoogleTest легко інтегрується з різними системами збірки та IDE, що значно полегшує включення тестування в робочий процес розробки. Ще одним плюсом даного фреймворку є його докладні звіти, щодо результатів тестування. Вони включають підсумки всіх тестів, повідомлення про помилки та інформацію про час виконання [10].

Тестування алгоритму нечіткого пошуку можна розбить на дві категорії: тестування таблиці подібності та тестування обчислення відстані редагування.

5.1. Створення фікстури для тестів.

Тест фікстура – це об’єкт в GoogleTest, який дозволяє нам написати два або більше тестів, які будуть працювати на подібних даних. Тобто фікстури дозволяють нам використовувати ту саму конфігурацію об’єктів для кількох різних тестів [10].

В нашому випадку, що для тестування таблиці подібності, що для тестування обчислення відстані редагування необхідно спочатку створити

об’єкт самої таблиці. Тому щоби розпочати тестування спочатку потрібно створити фікстуру таблиці подібності (рис. 5.1.1).

```
#include "gtest/gtest.h"
#include "../src/simtable.h"

class TableFixture : public testing::Test {
protected:
    void SetUp() override {
        const std::string pathToFile = "..\\..\\simtable.json";
        st_ = new SimTable( path: pathToFile);
    }
    SimTable* st_;
};
```

Рисунок 5.1.1. – Фрагмент лістингу коду для створення фікстури.

В даному коді ми створюємо фікстуру, яка буде запускатися перед нашими майбутніми тестами. В даній фікстурі ми лише створюємо об’єкт класу SimTable, який в свою чергу завантажує таблицю подібності з файлу, обробляє її і зберігає в пам’яті.

5.2. Тестування таблиці подібності символів.

Для тестування таблиці подібності символів було створено два різних тести:

1. Тестування внутрішніх даних таблиці подібності (рис. 5.2.1).

```
TEST_F(TableFixture, Table){
    std::random_device rd;
    std::mt19937  generator( Xx0: rd());
    std::uniform_int_distribution<int> distribution( Min0: 0,   Max0: st_->table.size()-1);

    std::set<size_t> ids;
    while(ids.size() < 30)
        ids.insert( Val: distribution( &: generator));

    size_t iter = 0;
    for (const auto& pair :const pair<...> &  : st_->table){
        if (ids.find( Keyval: iter) != ids.end()){
            for(const auto& simWch :const pair<...> &  : pair.second){
                auto simWchIter :iterator<...>  = st_->table.find( Keyval: simWch.first);
                EXPECT_NE(simWchIter, st_->table.end());
                EXPECT_NE(simWchIter->second.find(pair.first), simWchIter->second.end());
            }
        }
        iter++;
    }
}
```

Рисунок 5.2.1. – Фрагмент лістингу коду для тестування таблиці подібності.

В даному тесті ми, завдяки бібліотеці random, випадковим чином обираємо 30 символів. Далі для кожного символу ми проходимося по його подібним символам і перевіряємо чи є вони в таблиці подібності і чи є в них серед подібних – символ, який ми тестуємо. Завдяки цьому тесту в першу чергу ми хочемо перевірити цілісність таблиці подібності, а саме чи правильно вона створилася. Також ми перевіряємо чи виконується властивість симетричності таблиці подібності.

2. Тестування функції, яка використовуючи таблицю подібності, повертає вагу подібності двох символів (рис. 5.2.2.).

```
TEST_F(TableFixture, getReplaceCost){
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter;

// TEST SYMBOL A

    EXPECT_EQ(st_->getReplaceCost(L'a', L'q').value(), 0.5);

    auto testAA :float = st_->getReplaceCost( c1: L'A',  c2: L'a').value();
    EXPECT_TRUE(0.09 < testAA and testAA < 0.11);

    EXPECT_EQ(st_->getReplaceCost(L'a', L'l'), std::nullopt);

    EXPECT_EQ(st_->getReplaceCost(L'a', converter.from_bytes("á")[0]).value(), 0.25);

    auto testAO :float = st_->getReplaceCost( c1: L'a',  c2: L'o').value();
    EXPECT_TRUE(0.59 < testAO and testAO < 0.61);


// ANOTHER SYMBOLS

    EXPECT_EQ(st_->getReplaceCost(L'b', converter.from_bytes("β")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'c', converter.from_bytes("ç")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'd', converter.from_bytes("đ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'e', converter.from_bytes("é")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'f', converter.from_bytes("f")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'h', converter.from_bytes("ħ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'i', converter.from_bytes("i")[0]).value(), 0.25);
```

Рисунок 5.2.2. – Фрагмент лістингу коду для тестування функції, яка порівнює символи.

В даному тесті ми спочатку перевіряємо усі можливі випадки, які можна отримати з функції, яку ми тестуємо, для чотирьох різних символів. На рис. 5.2.2. наявний фрагмент лістингу лише для символу ‘a’, проте також було протестовано символи ‘g’, ‘n’, ‘r’.

Розглянемо на прикладі символу ‘a’. Ми розглядаємо п’ять можливих випадків:

1. Символи ‘a’ і ‘q’: в даному випадку ми перевіряємо можливу друкарську помилку. Функція повинна повернуть вагу подібності 0.5;
2. Символи ‘a’ і ‘A’: перевіряє випадок, коли букви мають різний регістр. Функція повинна повернуть вагу подібності 0.1. Оскільки числа з плаваючою комою, не можуть точно зберегти число 0.1, то його не можна порівняти оператором ==, тому щоби протестувати дане число, ми перевіряємо чи лежить воно в межах [0.09, 0.11];
3. Символи ‘a’ і ‘l’: в даному випадку ми перевіряємо символи , які не мають ніякої схожості. Функція повинна повернуть Null.
4. Символи ‘a’ і ‘å’: перевіряє випадок, коли символи маю семантичну схожість. Функція повинна повернуть вагу подібності 0.25;
5. Символи ‘a’ і ‘o’: перевіряє випадок, коли символи візуально подібні. Функція повинна повернуть вагу подібності 0.6. Оскільки дане число також не можна порівняти, то ми перевіримо чи лежить воно в межах [0.59, 0.61].

Далі також було протестовано для всіх можливих літер англійського алфавіту найважливішу категорію подібності символів, а саме семантичну подібність. Для кожного символу ми беремо його семантично подібний символ і перевіряємо чи для них функція поверне 0.25 .

5.3. Тестування обчислення відстані редагування.

Для тестування обчислення відстані редагування також було розроблено два тести. Перший перевіряємо роботу алгоритму Дамерау-Левенштейна без таблиці подібності, а другий – з таблицею. Розглянемо другий тест (рис. 5.3.1).

```
TEST_F(TableFixture, editDistanceWithTable){
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter;
    float test;

    EXPECT_EQ(editDistance(L"aah", L"aah", *st_), 0.);
    EXPECT_EQ(editDistance(L"", L"", *st_), 0.);
    EXPECT_EQ(editDistance(L"Open", L"", *st_), 4.);
    EXPECT_EQ(editDistance(L"", L"River", *st_), 5.);

    // KEYBOARD MISTAKE

    EXPECT_EQ(editDistance(L"algorithm", L"algoritjm", *st_), .5);
    EXPECT_EQ(editDistance(L"synchronixe", L"synchronize", *st_), .5);
    EXPECT_EQ(editDistance(L"difficult", L"diffidult", *st_), .5);
    EXPECT_EQ(editDistance(L"resiluent", L"resilient", *st_), .5);
    EXPECT_EQ(editDistance(L"alliance", L"allisnce", *st_), .5);

    // VISUAL SIMILAR CHAR

    test = editDistance( a: L"necessary",  b: L"recessary",  table: *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);
    test = editDistance( a: L"opporlunity",  b: L"opportunity",  table: *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);
```

Рисунок 5.3.1. – Фрагмент лістингу коду для тестування обчислення відстані редагування з використанням таблиці подібності.

В даному тесті ми перевіряємо можливі випадки пар слів для обчислення їх відстані редагування. Ми розглядаємо на прикладах слів всі можливі випадки помилок по окремості та випадки, коли помилки можуть комбінуватися. Для перевірки правильності роботи функції, всі розрахунки відстані Дамерау-Левенштейна було проведено вручну та звірено з результатами функції. В рис. 5.3.1 наведено лише фрагмент лістингу теста. Повний лістинг наявний в Додатку В.

Наведемо приклад можливих пар слів для пояснення, як проводилися розрахунки. Так наприклад пара слів “anthropology” та “amthopology”. В другому слові ми маємо дві помилки. Перша це друкарська помилка: “n” замінена на “m” (добавляємо до відстані редагування 0.5). Друга помилка це

пропуск символу “r” (добавляємо 1). В результаті, ціна перетворення другого слова на перше дорівнює 1.5, що і повинна повернуть функція.

5.4. Результати тестування.

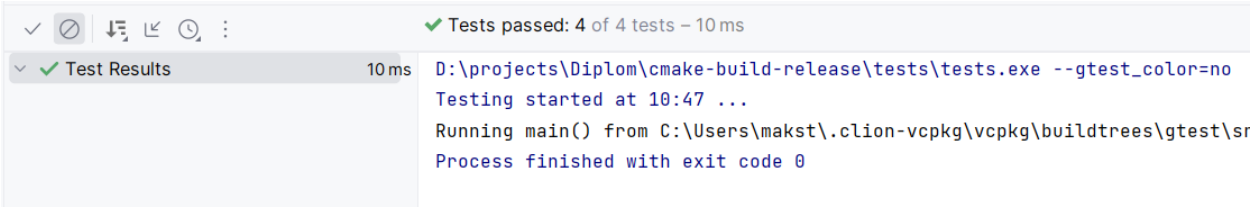

Рисунок 5.4.1 – Результати тестування.

Як видно з рис. 5.4.1, усі чотири тести пройшли успішно. Отже, завдяки даним результатам ми можемо впевнитися, що розроблений алгоритм працює вірно.

Повний лістинг коду для тестування алгоритму нечіткого пошуку і таблиці подібності наявний в Додатку В.

5.5. Висновки до розділу.

Отже, завдяки тестуванню програмного забезпечення, ми можемо знайти будь-які проблеми чи дефекти в написаному коді, що значно покращує якість кінцевого продукту. Тому було написано 4 різних тести, які перевіряли правильність роботи різних аспектів алгоритму нечіткого пошуку з використанням таблиці подібності символів. Усі чотири тести програма пройшла успішно, а отже алгоритм було реалізовано вірно.

6. ПОРІВНЯННЯ ШВИДКОДІЇ ТА ТОЧНОСТІ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ З ВИКОРИСТАННЯМ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ І БЕЗ.

Сьогодні існує безліч різних реалізацій алгоритмів нечіткого пошуку тексту. Вони всі відрізняються не лише різними критеріями для оцінки результатів нечіткого пошуку чи різними підходами до реалізації алгоритму, а й швидкістю і точністю виконання пошуку.

Швидкість та точність алгоритму впливають на безліч аспектів користування додатків. Перш за все ці два критерії безпосередньо впливають на задоволеність користувача у взаємодії з програмою. Користувачі очікують, що програми будуть швидкими та ефективними. Вони не будуть довго очікувати поки програма буде вирішувати їх запити, якщо існує альтернатива, яка робить це швидше. Тому на сьогоднішньому конкурентному ринку продуктивність програмного забезпечення може відіграти ключову роль. Контроль продуктивності програми дозволяє розробникам створювати більш конкурентоспроможні продукти.

Тому протестуємо швидкість виконання алгоритму нечіткого пошуку з використанням таблиці подібності та без таблиці. Та порівняємо їх швидкість та точність виконання.

Для тестування продуктивності розробленого алгоритму було обрано одну з найпопулярніших бібліотек Google Benchmark. Google Benchmark – це open-source бібліотека для порівняльного аналізу коду C+, розроблена Google. Головна мета даної бібліотеки – це надання засобів для вимірювання продуктивності коду C++, дозволяючи розробникам точно аналізувати та порівнювати час виконання різних функцій або фрагментів коду. Google Benchmark розроблений таким чином, щоб він був простим у використанні та водночас забезпечував надійні та точні результати [11]. Саме тому було обрано дану бібліотеку.

6.1. Тестування швидкодії створення таблиці подібності.

Оскільки перед виконанням алгоритму нечіткого пошуку з використанням таблиці подібності, необхідно створити саму таблицю, а цей процес також вимагає певного часу та ресурсів, тому її створення необхідно протестувати. Для цього створимо наступну бенчмарк функцію (рис. 6.1.1).

```
#include "benchmark/benchmark.h"
#include "../src/simtable.h"

static void BM_TABLE(benchmark::State& state) {
    for (auto _ :Value : state) {
        benchmark::DoNotOptimize( value: SimTable( path: "..\\..\\simtable.json"));
    }
}
```

Рисунок 6.1.1 – Фрагмент лістингу коду для тестування створення таблиці подібності символів.

Дана функцію полягає лише в тому, щоб створити об’єкт класу SimTable. Конструктор даного об’єкту проаналізує JSON файл, який містить дані таблиці подібності, та створить саму таблицю.

6.2. Бенчмарки для перевірки швидкодії алгоритмів нечіткого пошуку.

Для перевірки швидкості роботи алгоритмів нечіткого пошуку з використанням таблиці і без було розроблено два бенчмарки. Вони виконують ідентичну роботу, окрім запуску роботи самого алгоритму. Один бенчмарк виконує нечіткий пошук з використанням таблиці подібності, а інший – без. Тому пояснимо як працюють дані бенчмарки тільки на прикладі *BenchmarkWithTable* (рис. 6.2.1).

```
void BenchmarkWithTable(benchmark::State& state, const InputData& data, const SimTable& table) {
    for (auto _ :Value : state) {
        size_t countMatch = 0;
        auto errorThreshold :double = state.range( pos: 0) + 0.1;
        for(const auto& word :const wstring& : *data.words){
            auto result :float = editDistance( a: data.requestWord, b: word, table);
            benchmark::DoNotOptimize( &: result);

            if (result < errorThreshold)
                countMatch++;
        }
        state.counters["MATCHES"] = countMatch;
    }
}
```

Рисунок 6.2.1 – Бенчмарк для тестування швидкодії алгоритму нечіткого пошуку з використанням таблиці подібності.

В даному бенчмарку ми спочатку ітеруємося по об’єкту *State*, в якому зберігається конфігурації середовища, в якому будуть виконуватися бенчмарки. Тобто ми ітеруємося по різним можливим параметрам середовища, в нашому випадку це різні пошукові запити і тексти, в яких проводиться пошук. Далі ми ітеруємося по тестовим словам, порівнюємо їх із заданим пошуковим запитом, отримуємо відстань редагування та підраховуємо кількість слів, в яких дана відстань буде менша за порогову.

BenchmarkWithoutTable буде відрізнятися лише тим, що при розрахунку відстані редагування, не буде враховуватися таблиця подібності символів.

6.3. Тестування швидкодії алгоритмів нечіткого пошуку.

Для тестування швидкодії роботи алгоритмів нечіткого пошуку з використанням таблиці і без було вигадано два тести:

1. **Перевірка різних пошукових запитів.**

В даному тесті було підготовлено текстовий набір, який перекладений на три різні мови, а саме англійську, німецьку та чеську. Далі було вибрано по три пошукових запити на кожен переклад. Кожен пошуковий запит було шість разів модифіковано. Так, наприклад, чеський пошуковий запит було модифіковано так, як дане слово було б написано, використовуючи тільки

анлійський алфавіт (рис. 6.3.1). Далі ми проводимо нечіткий пошук з використанням таблиці і без для кожного запиту і відповідного тексту. Повний лістинг коду наявний в Додатку Г.

```
data.emplace_back( requestWord: converter.from_bytes( Ptr: "měsíčních"),  words: czText,  name: "Czech_Text");
data.emplace_back( requestWord: converter.from_bytes( Ptr: "Měsíčních"),  words: czText,  name: "Czech_Text");
data.emplace_back( requestWord: converter.from_bytes( Ptr: "Měsíčmícj"),  words: czText,  name: "Czech_Text");
data.emplace_back( requestWord: converter.from_bytes( Ptr: "měsičnich"),  words: czText,  name: "Czech_Text");
data.emplace_back( requestWord: converter.from_bytes( Ptr: "mesičnich"),  words: czText,  name: "Czech_Text");
data.emplace_back( requestWord: converter.from_bytes( Ptr: "mecicnich"),  words: czText,  name: "Czech_Text");
```

Рисунок 6.3.1. – Фрагмент лістингу коду для тестування швидкодії алгоритму нечіткого пошуку, використовуючи різні пошукові запити.

2. Поступова зміна алфавіту.

Основною метою даного тесту є перевірка, як будуть працювати алгоритми за умови, що пошуковий запит написаний на одній мові, а текст на іншій. Розглянемо фрагмент коду, який тестує алгоритми в даному середовищі (рис. 6.3.1).

```
//    TEST ALPHABET CHANGE

const std::string pathToTestWords = (R"(..\..\benchmarks\words_test.txt)");

InputData dataAlp;
dataAlp.words = std::make_shared<std::vector<std::wstring>>(getWords( path: pathToTestWords));
dataAlp.requestWord = getRandomWord( words: *dataAlp.words);

auto shuffledAlphabet :vector<wchar_t>  = getShuffledAlphabet( word: dataAlp.requestWord);

for(auto amountOfSymbolsToReplace = 1; amountOfSymbolsToReplace < 10; amountOfSymbolsToReplace++){
    auto oldSym :wchar_t  = shuffledAlphabet[amountOfSymbolsToReplace-1];
    auto newSym :wchar_t  = table.getRandomSimChar( wch: oldSym);
    changeSymbols( &: *dataAlp.words,  oldWch: oldSym,  newWch: newSym);
    for(auto errorThreshold = 0; errorThreshold <= 350; errorThreshold += 50){
        benchmark::RegisterBenchmark( name: "ALBenchmarkWithoutTable",  fn: [dataAlp](benchmark::State& state) -> void {
            BenchmarkWithoutTable( &: state,  data: dataAlp);
        })
        ->Args( args: {amountOfSymbolsToReplace, errorThreshold})
        ->Unit( unit: benchmark::kMillisecond)
        ->Iterations( n: 1);

        benchmark::RegisterBenchmark( name: "ALBenchmarkWithTable",  fn: [dataAlp, table](benchmark::State& state) -> void {
            BenchmarkWithTable( &: state,  data: dataAlp, table);
        })
        ->Args( args: {amountOfSymbolsToReplace, errorThreshold})
        ->Unit( unit: benchmark::kMillisecond)
        ->Iterations( n: 1);
    }
}
```

Рисунок 6.3.1. – Фрагмент лістингу коду для тестування швидкодії алгоритму нечіткого пошуку при поступовій зміні алфавіту.

В даному коді ми виконуємо наступні кроки:

1. Зчитуємо всі слова з тестового файлу за допомогою функції *getWords()*. Серед цих тестових слів ми і будемо проводити пошук.
2. Вибираємо випадковим чином серед тестових даних слово, яке будемо вважати пошуковим запитом. Цю операцію виконує функція *getRandomWord()*.
3. Створюємо список *shuffledAlphabet*, який буде містити англійський алфавіт. Алфавіт буде випадковим чином перемішаний. Цю операцію виконує функція *getShuffledAlphabet()*.
4. Створюємо цикл, в якому ми ітеруємося по кількості змінених символів алфавіту (від 1 до 10).
    4.1. В циклі ми беремо певний символ з *shuffledAlphabet*, для нього ми за допомоги функції table.getRandomSimChar() випадковим чином обираємо семантично подібний символ з таблиці подібності. Далі в тестових словах ми замінюємо всі входження символу на його подібний символ.
    4.2. Створюємо вкладений цикл, в якому ми ітеруємося по різним порогам подібності для алгоритму нечіткого пошуку. Цей поріг визначає, яка може бути максимальна відстань редагування між двома словами. Ми ітеруємося від 0 до 3.5 з кроком 0.5 . В даному циклі ми запускаємо бенчмарки *BenchmarkWithTable* і *BenchmarkWithoutTable*, які й будуть тестувати швидкодію в заданому середовищі.

Повний лістинг коду з реалізаціями всіх функцій доступний в Додатку Г.

6.4. Результати тестування швидкодії алгоритмів.

Для першого тесту в якості вхідних даних було обрано книгу «Гаррі Поттер і філософський камінь» в трьох перекладах: англійський, німецікий та чеський.

Для другого тесту в якості вхідних тестових даних було обрано файл розміром 370 000 різних випадкових слів. Наведемо фрагмент результату, який Google Benchmark, нам надає (рис. 6.4.1).

```
Run on (12 X 3000.38 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x6)
  L1 Instruction 32 KiB (x6)
  L2 Unified 512 KiB (x6)
  L3 Unified 4096 KiB (x2)
-------------------------------------------------------------------------------
Benchmark                                     Time             CPU   Iterations
-------------------------------------------------------------------------------
BM_TABLE                                2.67 ms         2.66 ms          264
ALBenchmarkWithoutTable/1/0/iterations:1     143 ms         93.8 ms            1 MATCHES=0
ALBenchmarkWithTable/1/0/iterations:1        372 ms          344 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/50/iterations:1    142 ms         93.8 ms            1 MATCHES=0
ALBenchmarkWithTable/1/50/iterations:1       373 ms          359 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/100/iterations:1   142 ms          109 ms            1 MATCHES=0
ALBenchmarkWithTable/1/100/iterations:1      380 ms          344 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/150/iterations:1   143 ms          109 ms            1 MATCHES=0
ALBenchmarkWithTable/1/150/iterations:1      370 ms          328 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/200/iterations:1   142 ms          109 ms            1 MATCHES=0
ALBenchmarkWithTable/1/200/iterations:1      372 ms          344 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/250/iterations:1   140 ms          141 ms            1 MATCHES=0
ALBenchmarkWithTable/1/250/iterations:1      381 ms          344 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/300/iterations:1   142 ms          125 ms            1 MATCHES=0
ALBenchmarkWithTable/1/300/iterations:1      371 ms          359 ms            1 MATCHES=1
ALBenchmarkWithoutTable/1/350/iterations:1   143 ms          125 ms            1 MATCHES=0
ALBenchmarkWithTable/1/350/iterations:1      369 ms          344 ms            1 MATCHES=1
ALBenchmarkWithoutTable/2/0/iterations:1     146 ms          125 ms            1 MATCHES=0
ALBenchmarkWithTable/2/0/iterations:1        374 ms          375 ms            1 MATCHES=124
```

Рисунок 6.4.1. – Фрагмент результату порівняння швидкодії алгоритмів.

Результати тестування швидкодії створення таблиці подібності наведемо в таблиці 6.4.1.

Таблиця 6.4.1. – Результати тестування швидкодії створення таблиці подібності.

Benchmark	**Time**	**CPU**	**Iterations**
BM_Table	2.67 ms	2.66 ms	264

Дивлячись на таблицю 6.4.1 можна сказати, що за 264 спроби створити таблицю подібності середній час показує 2.67 ms.

Для зручності представлення і порівняння результатів тестування швидкодії та точності самих алгоритмів, зобразимо їх, використовуючи графіки. Для цього результати тестування збережемо в JSON файл, а потім, використовуючи засоби мови Python і бібліотеки matplotlib, зобразимо їх у графіках та гістограмах.

Наведемо гістограми тесту, в якому ми виконуємо нечіткий пошук, використовуючи різні пошукові запити (рис. 6.4.2 – рис. 6.4.10).

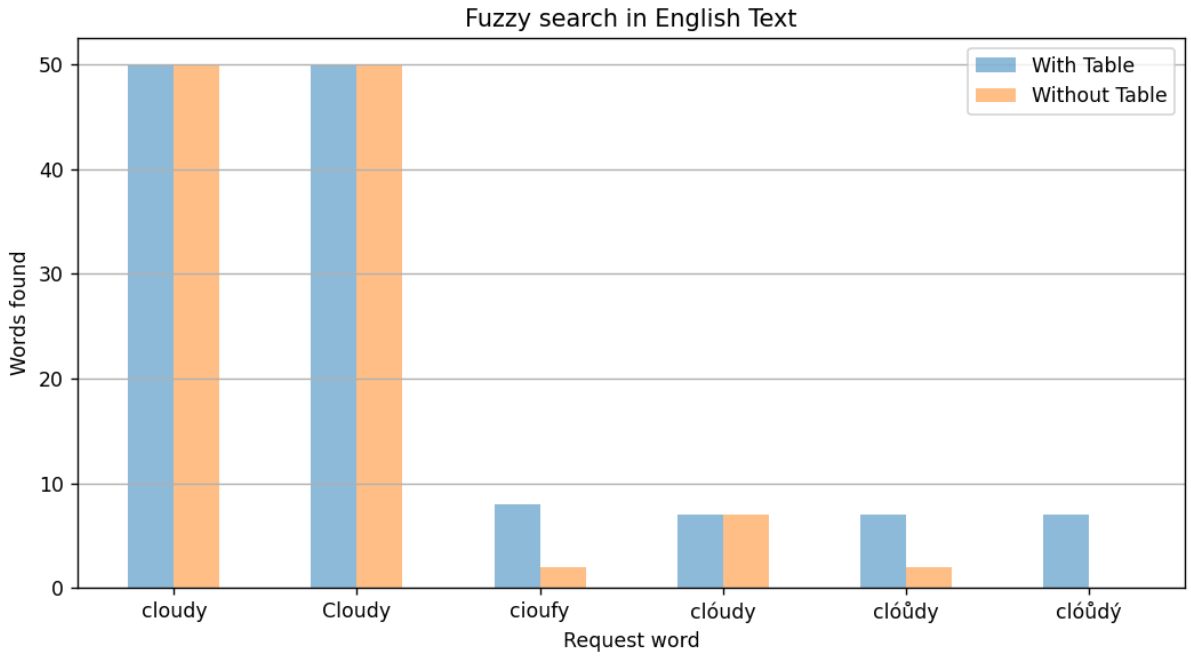

Рисунок 6.4.2. – Гістограма результатів пошуку алгоритмів в англійському тексті (пошуковий запит: cloudy).

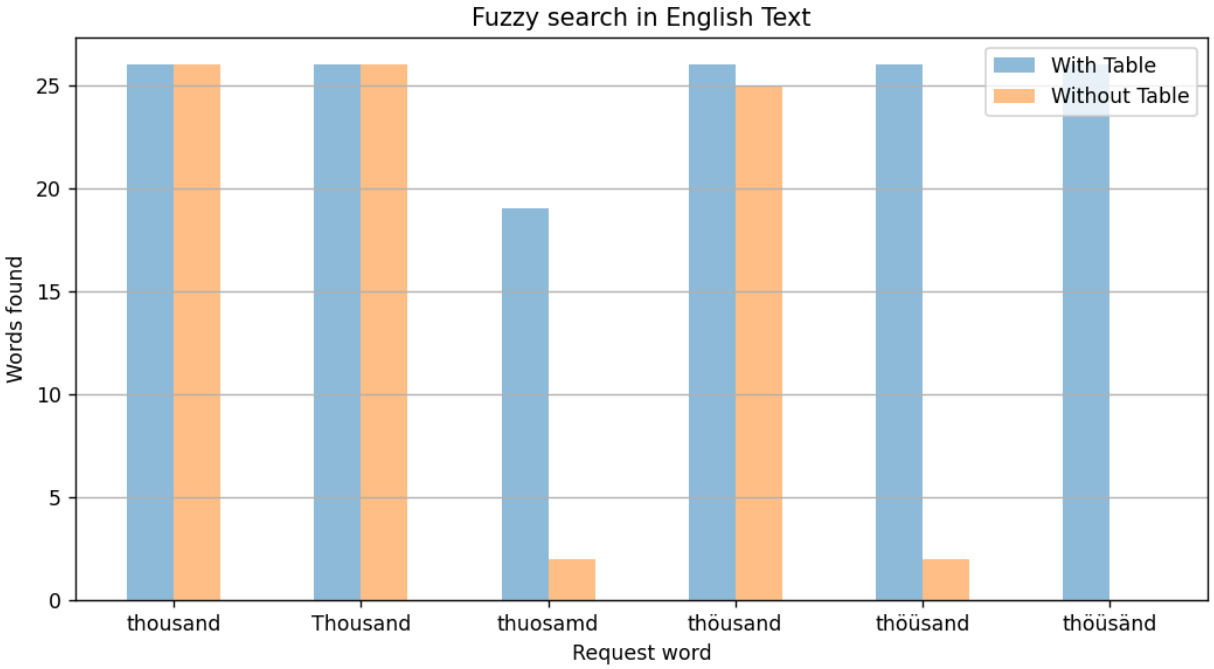

Рисунок 6.4.3. – Гістограма результатів пошуку алгоритмів в англійському тексті (пошуковий запит: thousand).

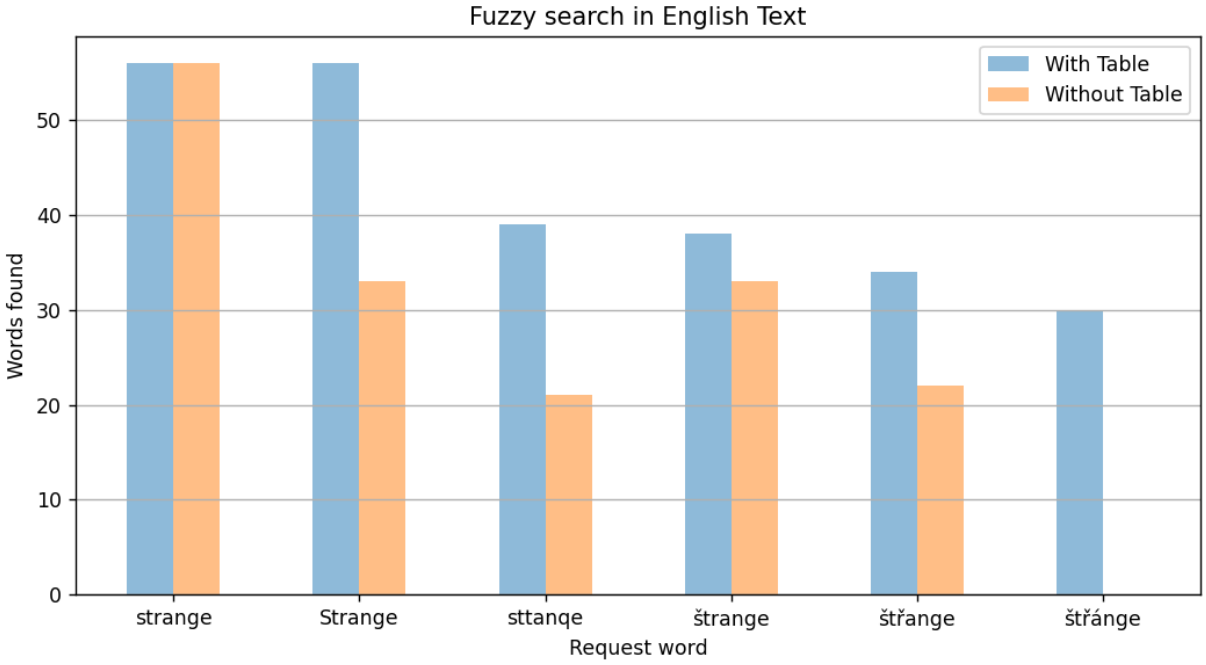

Рисунок 6.4.4. – Гістограма результатів пошуку алгоритмів в англійському тексті (пошуковий запит: strange).

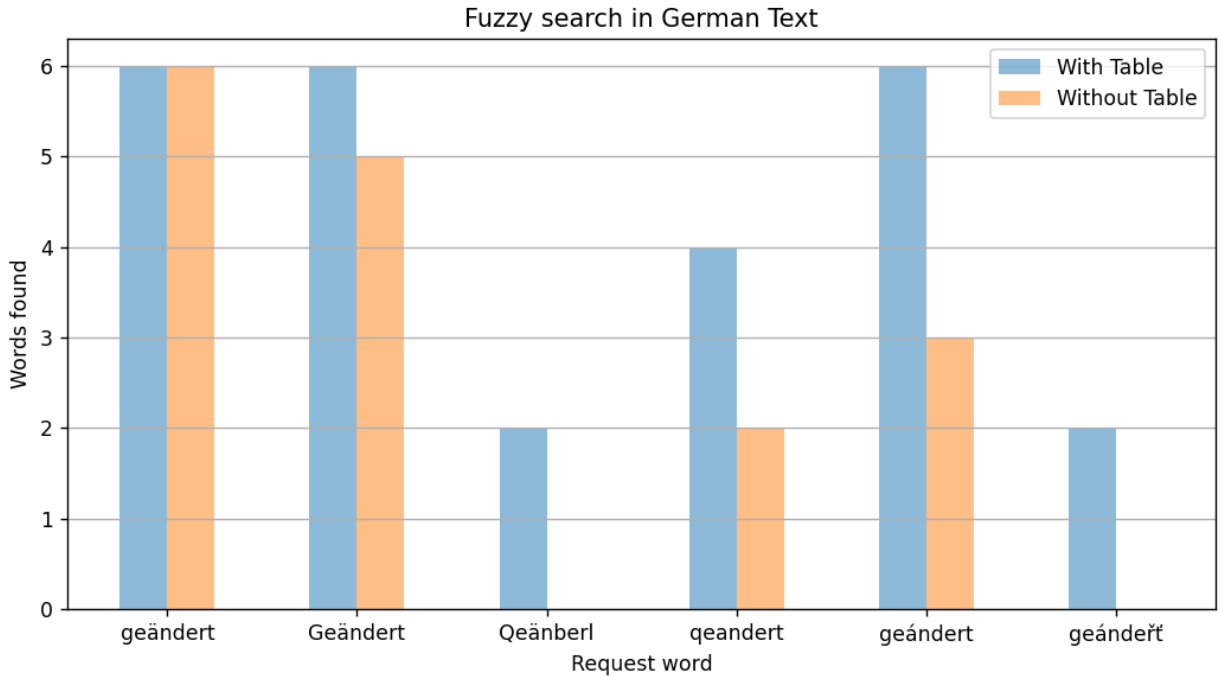

Рисунок 6.4.5. – Гістограма результатів пошуку алгоритмів в німецькому тексті (пошуковий запит: geändert).

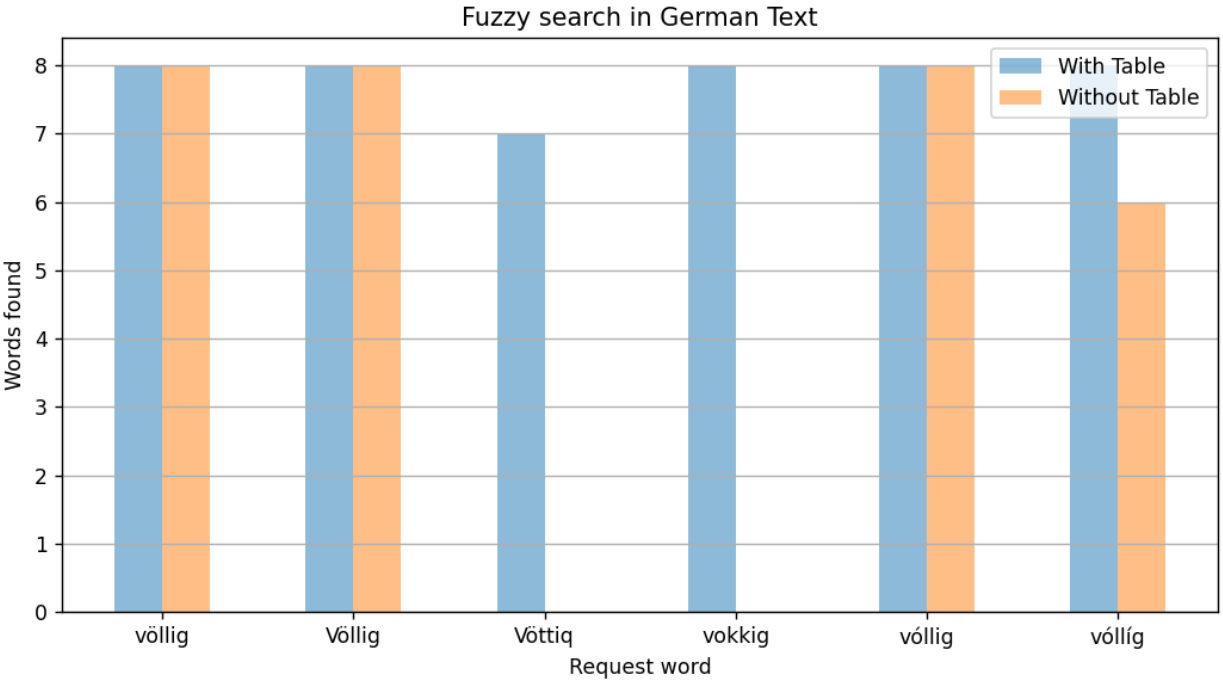

Рисунок 6.4.6. – Гістограма результатів пошуку алгоритмів в німецькому тексті (пошуковий запит: völlig).

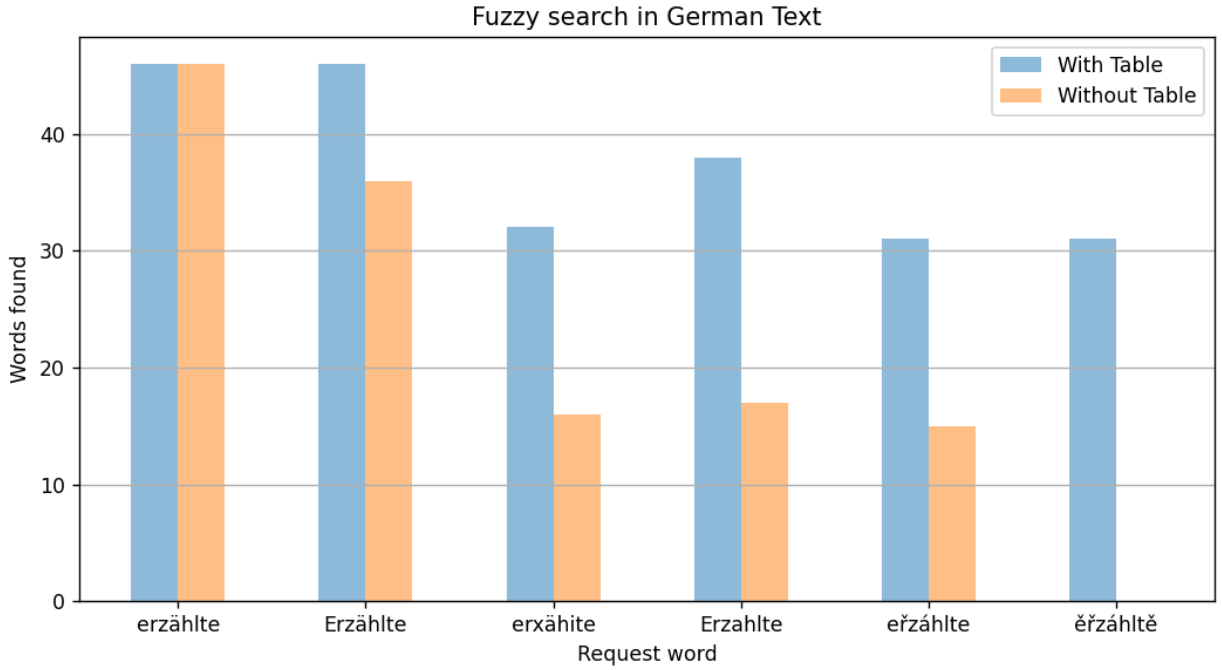

Рисунок 6.4.7. – Гістограма результатів пошуку алгоритмів в німецькому тексті (пошуковий запит: erzählte).

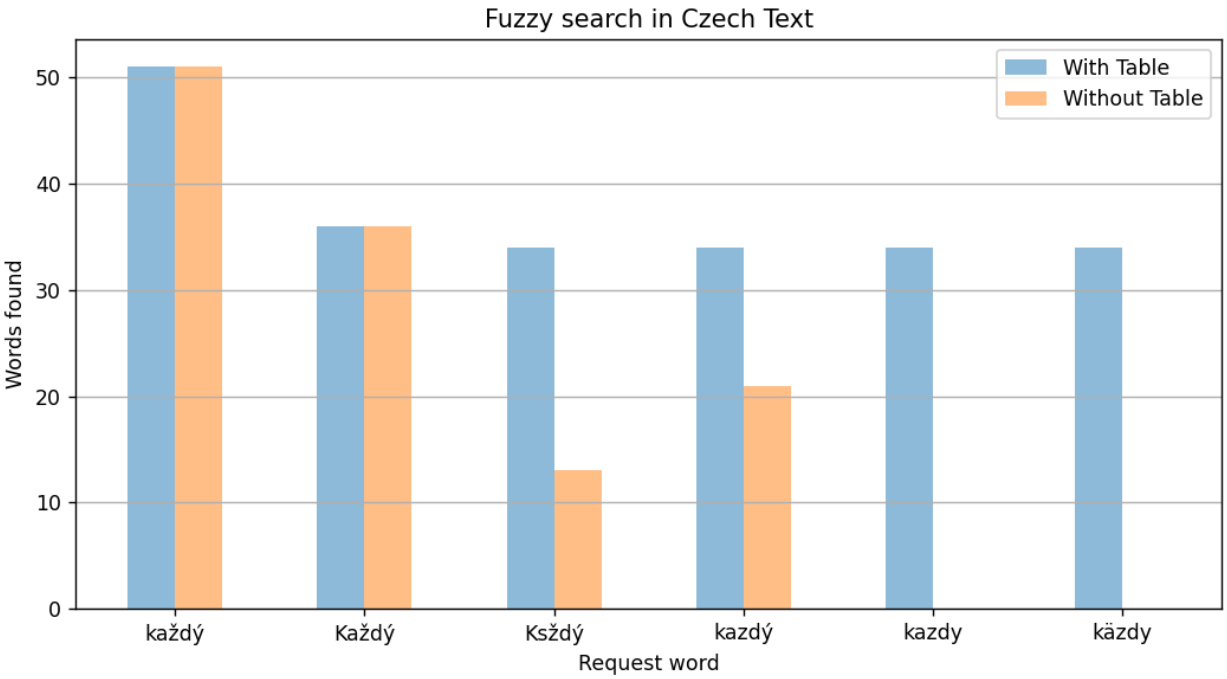

Рисунок 6.4.8. – Гістограма результатів пошуку алгоритмів в чеському тексті (пошуковий запит: každý).

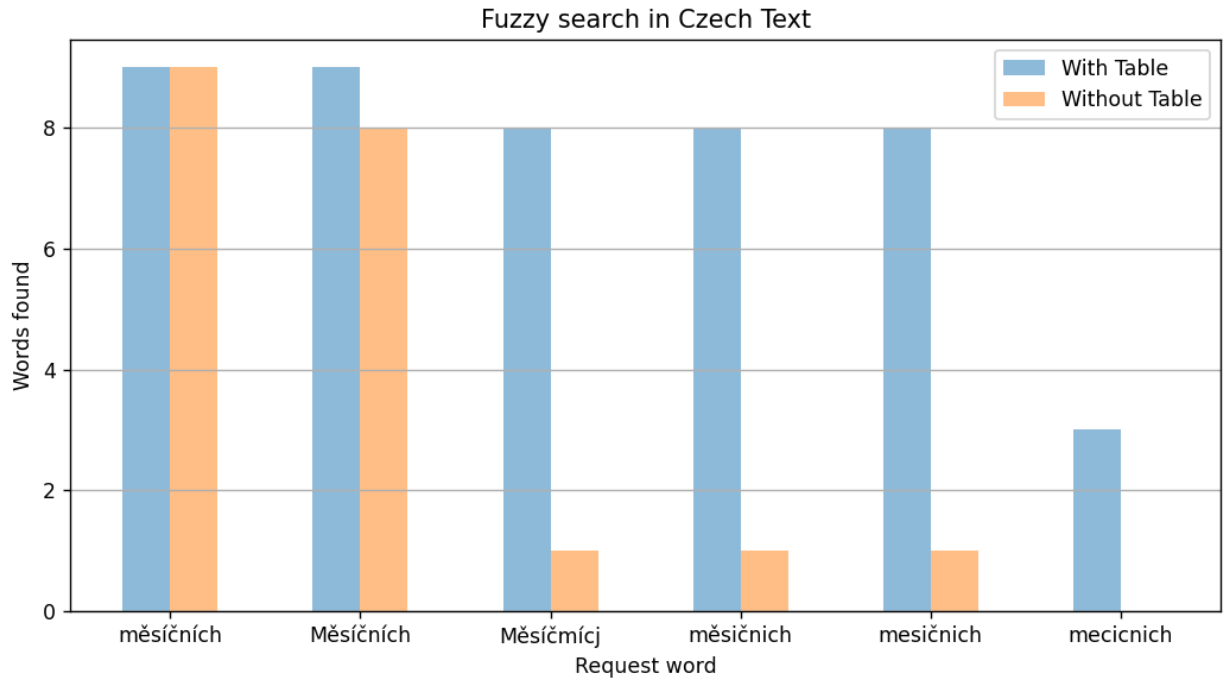

Рисунок 6.4.9. – Гістограма результатів пошуку алгоритмів в чеському тексті (пошуковий запит: měsíčních).

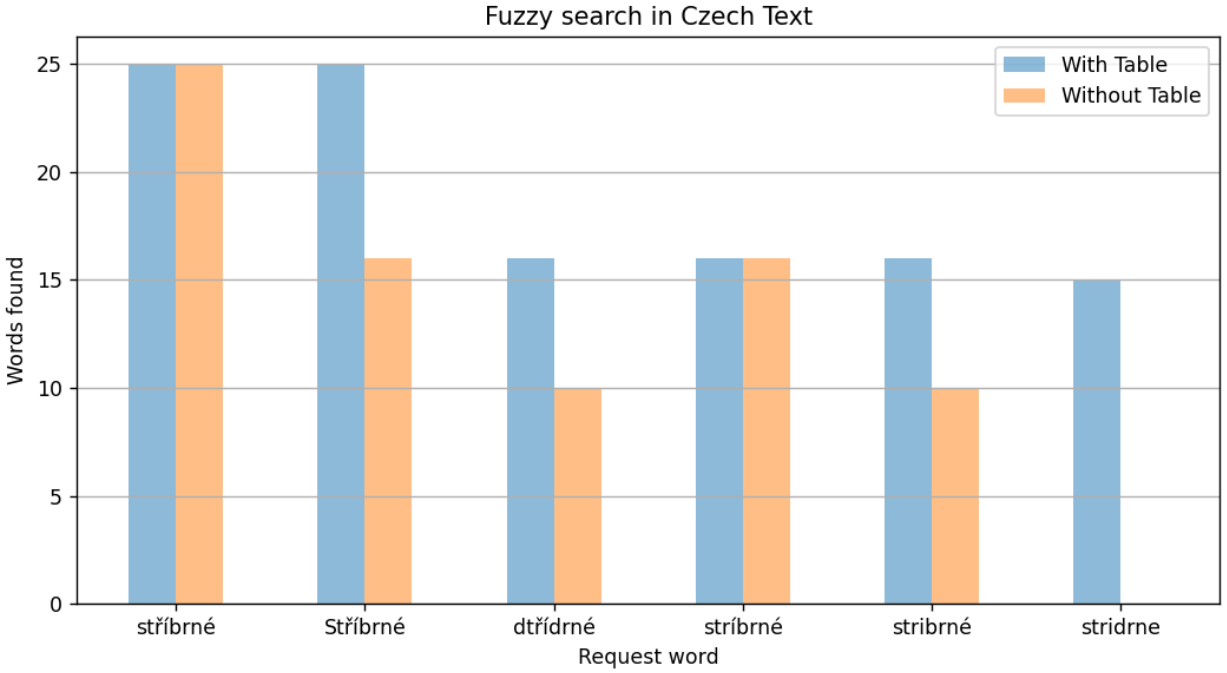

Рисунок 6.4.10. – Гістограма результатів пошуку алгоритмів в чеському тексті (пошуковий запит: stříbrné).

По x-осі в даних гістограмах зображені різні пошукові запити, а по y – кількість знайдених слів. Дивлячись на дані гістограми, можна дійти висновку,

що при слові, яке не має жодної модифікації, обидва алгоритми показують однаковий результат. Проте, коли ми змінюємо пошуковий запит, нечіткий пошук з використанням таблиці показує значно кращий результат чим алгоритм без таблиці. Також можна сказати, що чим більше змінених символів в слові, тим меншу кількість слів можна знайти.

Наведемо графіки тесту, який поступово змінює алфавіт (рис. 6.4.11 та рис. 6.4.12).

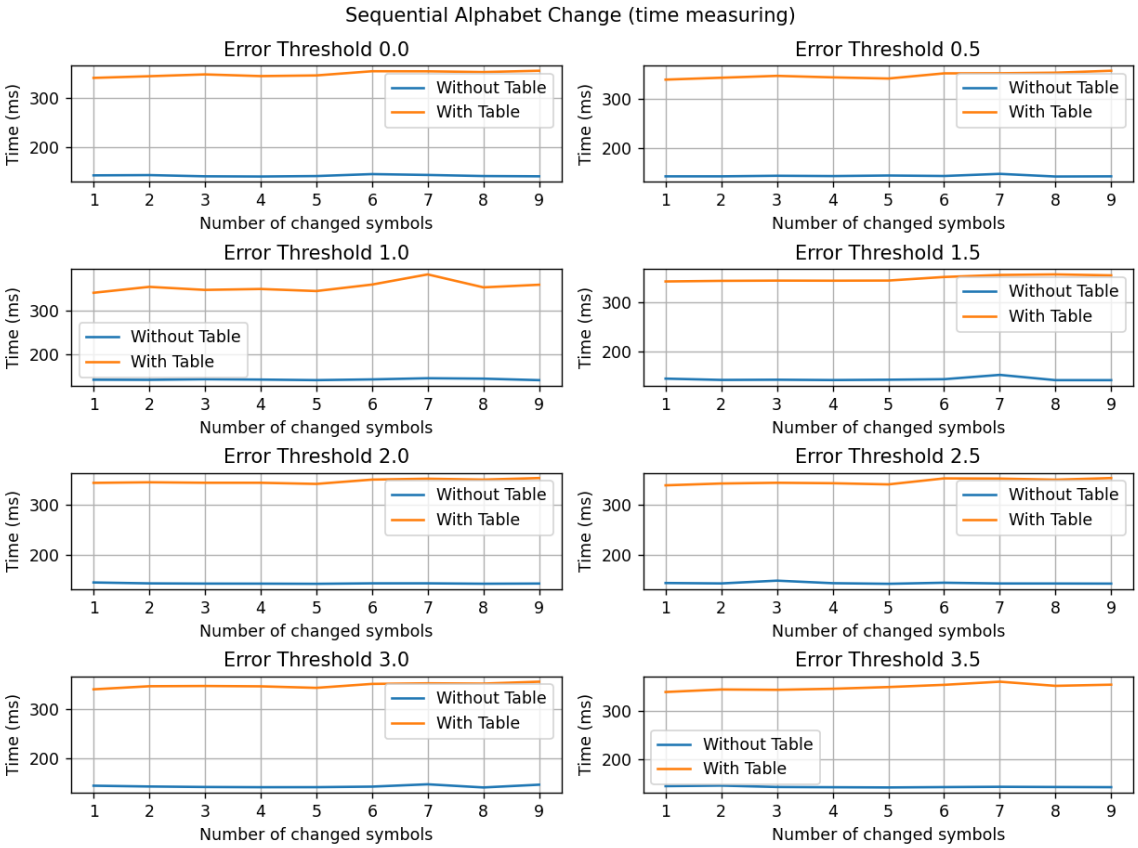

Рисунок 6.4.11. – Графік порівняння швидкодії алгоритмів, поступово змінюючи алфавіт тексту для пошуку.

По x-осі в даних графіках представлена кількість змінених символів, а по y – час пошуку. В кожному окремому графіку ми порівнюємо швидкодію з різним порогом пошуку від 0 до 3.5. Дивлячись на рис. 6.4.2, видно що швидкість алгоритмів не залежать від порогового значення чи вмісту тексту, в якому ведеться пошук. Також видно, що нечіткий пошук, використовуючи алгоритм без таблиці подібності, в середньому в 2.5 раз виконується швидше ніж з таблицею.

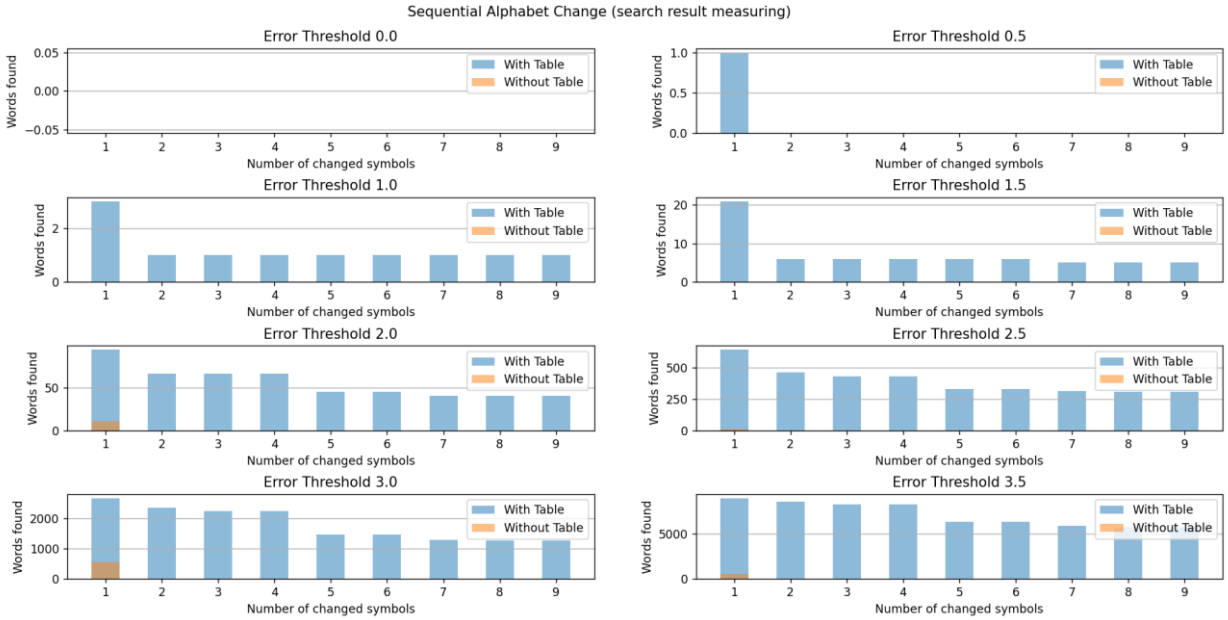

Рисунок 6.4.12. – Гістограма порівняння результатів пошуку алгоритмів, поступово змінюючи алфавіт тексту для пошуку.

По x-осі в даних графіках представлена кількість змінених символів, а по y – кількість знайдених слів. Порівнюючи кількість знайдених слів, можна сказати, що алгоритм з використанням таблиці подібності показує кращі результати. Чим більший поріг відстані редагування, тим більшу кількість слів ми можемо знайти. Проте збільшуючи даний показник, ми зменшуємо саму точність пошуку. Чим більший поріг, тим менш подібні слова ми можемо отримати в результаті.

6.5. Висновки до розділу.

Сьогодні швидкість виконання програмного забезпечення являється важливим критерієм, який впливає на задоволеність роботи користувача з ПЗ. Проте не меншу роль відіграє і точність програми. Порівнюючи ці два алгоритми, можна дійти висновку, що алгоритм нечіткого пошуку з використанням таблиці подібності показує кращу точність пошуку. Тому його варто використовувати, коли для користувача важливіші результати пошуку. А нечіткий пошук без таблиці показує кращі результати в швидкодії, тому його слід використовувати, коли користувачу важлива швидкість пошуку.

7. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ.

У даному розділі проведена оцінка основних функціональних та вартісних характеристик програмного продукту, а саме нечіткий пошук в тексті з використанням таблиці подібності символів.

Також в даному дослідженні нижче показано аналіз різних варіантів реалізації модулю з метою забезпечення найбільш коректної та оптимальної стратегії вибору, яка враховує економічні фактори та характеристики продукту, що впливають на продуктивність роботи і сумісність з апаратним забезпеченням. Для досягнення цієї мети було використано метод функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) є методологією, що дозволяє оцінити фактичну вартість продукту або послуги, незалежно від внутрішньої організаційної структури компанії. Основна мета ФВА полягає в ефективному розподілі ресурсів, які виділяються на виробництво продукції або надання послуг, між прямими та непрямими витратами. У даному контексті, ФВА використовується для аналізу функцій програмного продукту та ідентифікації всіх зв'язаних з цим функціональних витрат. Для проведення аналізу використовується економічна, технічна та конструкторська інформація.

Алгоритм функціонально-вартісного аналізу має такі кроки:

1. Спочатку визначається повна послідовність функцій, які необхідні для виробництва продукту. Потім ці функції розподіляються на дві групи: ті, що впливають на вартість продукту, і ті, що не мають такого впливу. Також проводиться оптимізація послідовності шляхом скорочення кроків, які не впливають на вартість і, відповідно, на витрати.
2. Для кожної функції визначається повний обсяг витрат (річних) та кількість робочих часів.

3. Для кожної функції визначається кількісна характеристика джерел витрат
4. На основі визначених оцінок, проводиться кінцевий розрахунок витрат на виробництво продукту

7.1. Постановка задачі проектування.

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки системи нечіткого пошуку в тексті з використанням таблиці подібності символів. 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 в тексті.

До продукту були визначені наступні технічні вимоги:

- можливість виконання на персональних комп’ютерах із стандартною конфігурацією;
- висока швидкість обробки даних та відповідь користувачеві у реальному часі;
- зручність та простота взаємодії;
- можливість зручного налаштування, масштабування та обслуговування;
- мінімальні витрати на впровадження програмного продукту.

7.2. Обґрунтування функцій програмного продукту.

Головна функція F0 – розробка програмного продукту , який вирішує задачу нечіткого пошуку в тексті з використанням таблиці подібності символів. Виходячи з конкретної мети, можна виділити наступні основні функції ПП:

– F_1 – вибір мови програмування;
– F_2 – вибір середовища розробки;
– F_3 – вибір формату файлу, для збереження таблиці подібності символів.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

a) C++\Python.

Функція F_2:

a) Clion\PyCharm;
b) Visual Studio Code.

Функція F_3:

a) JSON;
b) Створення свого формату.

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 7.2.1).

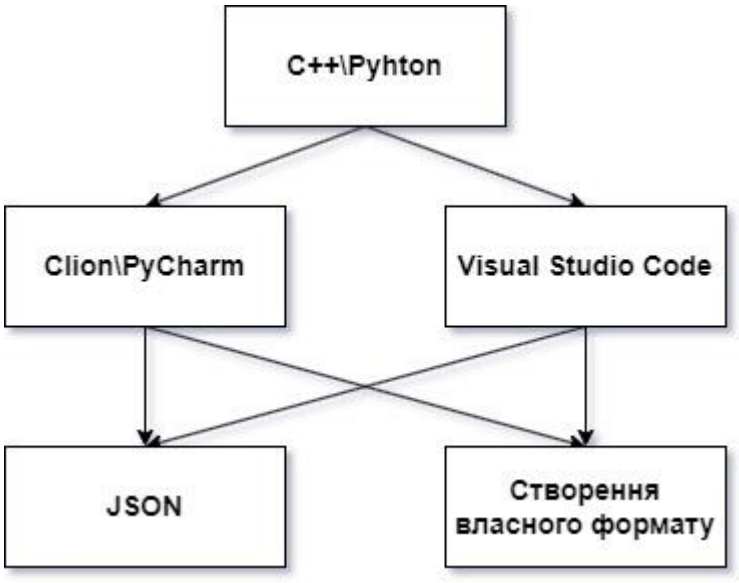

Рисунок 7.2.1. – Морфологічна карта варіантів реалізації функцій.

Побудуємо позитивно-негативну матрицю в таблиці 7.2.1, використовуючи морфологічну карту.

Таблиця 7.2.1 – Позитивно-негативна матриця варіантів основних функцій.

Функції	Варіанти реалізації		Переваги	Недоліки
F_1	A	C++	Швидкодія роботи, можливість керувати пам’яттю	Складність розробки продукту
		Python	Легкість використання, велика кількість бібліотек	Швидкодія роботи
F_2	A	Clion	Великий функціонал для розробки мовою C\C++, підтримка GIT, розширена підтримка C++	Вимагає потужне апаратне забезпечення
		PyCharm	Широка підтримка різних фреймворків, потужні інструменти аналізу коду, широка база документації	Вимагає потужне апаратне забезпечення
	B	Visual Studio Code	Підтримує багато мов програмування, легка у використанні, підтримує інтеграцію з багатьма інструментами розробки	Відсутність деяких вбудованих функцій, для яких необхідно сторонні розширення, менша підтримка мов програмування

Продовження таблиці 7.2.1.

F_3	A	JSON	Загально прийнята реалізація	Відсутність типізації, відсутність коментарів, обмежена підтримка бінарних даних
	B	Власний формат	Можна самому організувати дані, як це необхідно програмі	Необхідність виділення ресурсів на написання програми для побудови всього необхідного в задачі, можлива надмірна об’ємність даних

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

Функція F_1:

Оскільки мова програмування C++ має вбудовані засоби для роботи з символами UTF-16, а для парсингу сайтів загальноприйнятою мовою є Python, тому при розробці програмного коду існує лише один варіант вибору мов програмування.

Функція F_2:

Оскільки при розробці програмного коду, використання повноцінних IDE надасть більшу підтримку для заданих мов програмування, то для спрощення роботи по написанню коду варіант B має бути відкинутий.

Функція F_3:

Програма допускає обрання обох варіантів. Можливо використати варіанти A чи B.

Таким чином, будемо розглядати варіанти реалізації ПП:

F_1 (a) – F_2(a) – F_3(a);

F_1 (a) – F_2(a) – F_3(b).

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

7.3. Обґрунтування системи параметрів програмного продукту.

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

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

- X1 – швидкодія пошуку (кількість слів, які обробив пошуковий механізм за певний час);
- X2 – об’єм пам’яті для обробки даних під час виконання програми;
- X3 – точність пошуку в текстових даних;
- X4 – 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 7.3.1.

Таблиця 7.3.1 – Основні параметри програмного продукту.

Назва параметра	Умовні позначення	Одиниці виміру	Значення параметру		
			гірші	середні	кращі
Швидкодія пошуку	X1	Кількість слів/мс	800	950	1200
Об’єм пам’яті	X2	Мб	30	20	15
Точність пошуку	X3	%	70	85	100
Потенційний об’єм програмного коду	X4	Кількість рядків коду	1500	1300	1000

За даними таб. 7.3.2 будуються графічні характеристики параметрів (див. рис. 7.3.1–7.3.4).

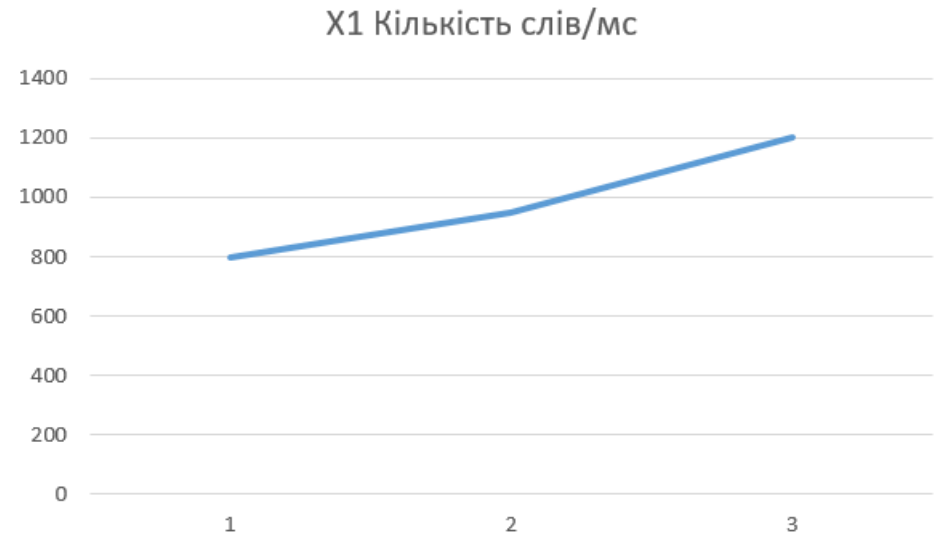

Рисунок 7.3.1 – Швидкодія пошуку.

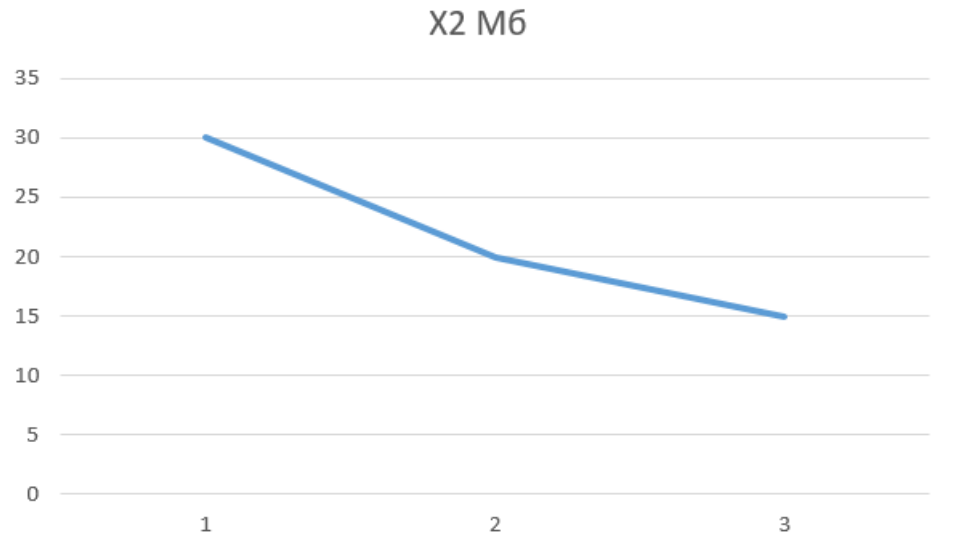

Рисунок 7.3.2 – Об’єм пам’яті для обробки даних.

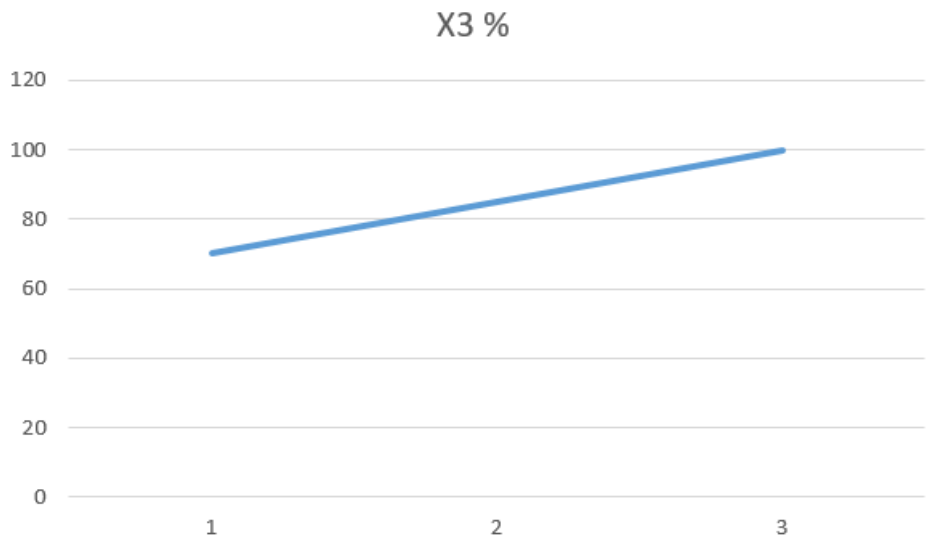

Рисунок 7.3.3 – Точність нечіткого пошуку.

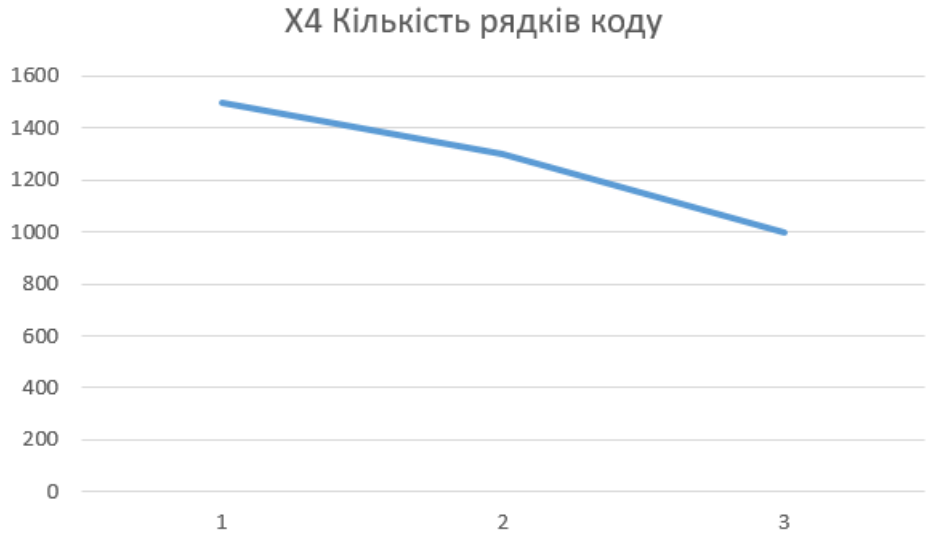

Рисунок 7.3.4 – Потенційний об’єм програмного коду.

7.4. 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 нечіткому пошуку в тексті.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості 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 7.4.1.

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхи-лення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія пошуку	Кількість слів/мс	4	3	3	4	4	3	4	25	7.5	56.25
X2	Об’єм пам’яті	Мб	2	1	2	2	2	1	3	13	-4.5	20.25
X3	Точність пошуку	%	3	4	4	3	3	4	2	23	5.5	30.25
X4	Потенційний об’єм програмного коду	Кількість рядків коду	1	2	1	1	1	2	1	9	-8.5	72.25
Разом			10	10	10	10	10	10	10	70	0	179

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

1. сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70, \qquad (7.4.1)$$

де N – число експертів, n – кількість параметрів;

2. середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 17.5 \qquad (7.4.2)$$

3. відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \qquad (7.4.3)$$

4. загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 179. \qquad (7.4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3-n)} = \frac{12*179}{7^2(4^3-4)} = 0.73 > W_k = 0.67. \quad (7.4.5)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 7.4.2.

Таблиця 7.4.2. – Попарне порівняння параметрів.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	>	>	>	>	>	>	>	1.5
X1 і X3	>	<	<	>	>	<	>	>	1.5
X1 і X4	>	>	>	>	>	>	>	>	1.5
X2 і X3	<	<	<	<	<	<	>	<	0.5
X2 і X4	>	<	>	>	>	<	>	>	1.5
X3 і X4	>	>	>	>	>	>	>	>	1.5

Числове значення, що визначає ступінь переваги і–го параметра над j–тим, *aij* визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases} \qquad (7.4.6)$$

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

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

$$K_{\text{вi}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \qquad (7.4.7)$$

$$b_i = \sum_{i=1}^{N} a_{ij} \qquad (7.4.8)$$

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:

$$K_{\text{вi}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'} \qquad (7.4.9)$$

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j \qquad (7.4.10)$$

Як видно з таблиці 7.4.3, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 7.4.3 – Розрахунок вагомості параметрів.

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер.	
	X1	X2	X3	X4	b_i	$K_{вi}$	b_i^1	$K_{вi}^1$	b_i^2	$K_{вi}^2$
X1	1	1.5	1.5	1.5	5.5	0.34	21.25	0.36	77.88	0.36
X2	0.5	1	0.5	1.5	3.5	0.22	12.25	0.21	44.88	0.21
X3	0.5	1.5	1	1.5	4.5	0.28	16.25	0.28	59.13	0.27
X4	0.5	0.5	0.5	1	2.5	0.16	9.25	0.16	34.13	0.16
Всього:					16	1	59	1	216	1

7.5. Аналіз рівня якості варіантів реалізації функцій.

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів X2 (об’єм пам’яті), X3 (точність пошуку) та X4 (потенційний об’єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра X1 (Швидкодія пошуку) обрано не найгіршим.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 7.5.1):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j}, \qquad (7.5.1)$$

де n – кількість параметрів;

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

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

Таблиця 7.5.1 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП.

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	A	X1	1000	24	0.36	8.64
F2	A	X2	30	8	0.21	1.68
F3	A	X3	1100	16	0.27	4.32
	B	X4	1500	10	0.16	1.6

За даними з таблиці 7.5.1 за формулою:

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}], \qquad (7.5.2)$$

Визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 8.64 + 1.68 + 4.32 = 14.64;$$

$$K_{K2} = 8.64 + 1.68 + 1.6 = 11.92;$$

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

7.6. 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;

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

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

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

Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_М \cdot K_{СТ} \cdot K_{СТ.М}, \qquad (7.6.1)$$

де T_P – трудомісткість розробки ПП;

$K_П$ – поправочний коефіцієнт;

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

$K_М$ – коефіцієнт рівня мови програмування;

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

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

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P =33 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.6. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $K_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ}$ = 0.85. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 33 * 1.6 * 0.85 = 44.88 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $T_Р$ =27 людино-днів, $К_П$ =0.75, $К_{СК}$ = 1, $К_{СТ}$ =0.8:

$$T_2 = 27 * 0.75 * 0.8 = 16.2 \text{ людино-днів.}$$

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

$$T_I = (44.88 + 16.2 + 5.77 + 16.2) * 8 = 664.4 \text{ людино-днів.}$$

$$T_{II} = (44.88 + 16.2 + 6.26 + 16.2) * 8 = 668.32 \text{ людино-днів.}$$

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 57400 грн., один аналітик в області даних з окладом 63000. Визначимо середню зарплату за годину за формулою:

$$С_ч = \frac{М}{T_m * t} \text{ грн.,} \quad (7.6.2)$$

де М – місячний оклад працівників;

T_m – кількість робочих днів в тиждень;

t – кількість робочих годин в день.

$$С_ч = \frac{57400+57400+63000}{3*21*8} = 352.78 \text{ грн.} \quad (7.6.3)$$

Тоді, розрахуємо заробітну плату за формулою:

$$С_{ЗП} = С_ч * T_i * К_Д, \quad (7.6.4)$$

де $С_ч$ – величина погодинної оплати праці програміста;

T_i – трудомісткість відповідного завдання;

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

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 352.78 * 664.4 * 1.2 = 281\,264.43$ грн.

II. $С_{ЗП} = 352.78 * 668.32 * 1.2 = 282\,923.92$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} * 0.22 = 281\,264.43 * 0.22 = 61\,878.17$ грн.

II. $С_{ВІД} = С_{ЗП} * 0.22 = 282\,923.92 * 0.22 = 62\,243.26$ грн.

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

Так як одна ЕОМ обслуговує одного програміста з окладом 57400 грн., з коефіцієнтом зайнятості 0.2 то для однієї машини отримаємо:

$$С_г = 12 * М * К_З = 12 * 57400 * 0.2 = 137\ 760 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_г * (1 + К_З) = 137\ 760 * (1 + 0.2) = 165\ 312 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} * 0.22 = 165\ 312 * 0.22 = 36\ 368.64 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 75000 грн.

$$С_А = К_{ТМ} * К_А * Ц_{ПР} = 1.35 * 0.25 * 75\ 000 = 25\ 312.5 \text{ грн.}$$

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:

$$С_Р = К_{ТМ} * Ц_{ПР} * К_Р = 1.35 * 75\ 000 * 0.08 = 8\ 100 \text{ грн.}$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) * t_з * К_В = (365 - 104 - 12 - 16) * 8 * 0.85 = 1\ 584.4 \text{ години,}$$

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;

$t_з$ – кількість робочих годин в день;

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

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} * N_С * К_З * Ц_{ЕН} = 1\ 584.4 * 0.25 * 0.7 * 4.79 = 1\ 328.12 \text{ грн.}$$

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:

$$С_Н = Ц_{ПР} * 0.67 = 75\,000 * 0.67 = 50\,250,\ грн$$

Тоді річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н, \qquad (7.6.5)$$

$$С_{ЕКС} = 165\,312 + 36\,368.64 + 25\,312.5 + 8\,100 + 1\,328.12 + 50\,250 = 286\,671.26 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М-Г} = С_{ЕКС}/Т_{ЕФ} = 286\,671.26/1\,584.4 = 180.93 \text{ грн/год.}$$

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

$$С_М = С_{М-Г} * Т, \qquad (7.6.6)$$

I. $С_М = 180.93 * 664.4 = 120\,209.89$ грн.

II. $С_М = 180.93 * 668.32 = 120\,919.13$ грн.

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} * 0.67, \qquad (7.6.7)$$

I. $С_Н = 281\,264.43 * 0.67 = 188\,447.17$ грн.

II. $С_Н = 282\,923.92 * 0.67 = 189.559.03$ грн.

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_М + С_Н, \qquad (7.6.8)$$

I. $С_{ПП} = 281\,264.43 + 61\,878.17 + 120\,209.89 + 188\,447.17 = 651\,799.66$ грн.

II. $С_{ПП} = 282\,923.92 + 62\,243.26 + 120\,919.13 + 189.559.03 = 655\,645.34$ грн.

7.6. Вибір кращого варіанту ПП техніко-економічного рівня.

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj}/С_{Фj}, \qquad (7.6.1)$$

$$К_{ТЕР1} = 14.64/651\,799.66 = 2.246 * 10^{-5},$$

$$К_{ТЕР2} = 11.92/655\,645.34 = 1.818 * 10^{-5},$$

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1}=2.246*10^{-5}$.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, 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 $К_{ТЕР1}=2.246*10^{-5}$.

Цей варіант реалізації програмного продукту має такі параметри:

- Вибір мов програмування C++ та Python;
- Вибір середовища розробки CLion та PyCharm;
- Вибір формату файлу для збереження таблиці подібності символів JSON.

Даний варіант виконання програмного комплексу дає користувачу зручний інтерфейс, непоганий функціонал і велику швидкодію ПП.

7.7. Висновки до розділу.

У даному розділі проведено функціонально-вартісний аналіз програмного продукту. Також було визначено параметри які характеризують ПП і проведено оцінку його основних функцій функції . Проведено експертне оцінювання параметрів та аналіз якості варіантів реалізації функцій. Також проведено економічний аналіз варіантів розробки.

На основі проведеного аналізу було визначено, кращий варіант реалізації програмного продукту з вищим показником техніко-економічного рівня якості, а саме перший варіант.

ВИСНОВКИ

В ході виконання дипломної роботи було ретельно досліджено поняття нечіткого пошуку в тексті, а саме було досліджено існуючі підходи нечіткого пошуку. Дані алгоритми дозволяють нам вирішити проблеми, пов’язані з пошуком і отриманням інформації у великих обсягах текстових даних, де можливі орфографічні помилки часто заважають традиційним методам пошуку. Було розглянуто, як працюють сучасні алгоритми нечіткого пошуку, їх переваги та недоліки. Завдяки цим перевагам, кожен алгоритм знайшов своє застосування в різних галузях інформаційного пошуку.

Проте жоден з розглянутих алгоритмів не враховує можливу певну семантичну подібність символів в словах. Таблиця подібності символів виявилася ефективним рішенням для вирішення даної проблеми, фіксуючи можливі семантичні та інші подібності між символами. Оцінюючи подібності на основі різних критеріїв, таких як форма, контекст і фонетика, таблиця подібності символів дозволила нам модифікувати алгоритм нечіткого пошуку, який зміг би ідентифікувати та ранжувати відповідні результати навіть за наявності помилок у вхідному запиті чи тексту, в якому відбувається пошук.

Тому згідно до аналізу алгоритму Дамерау-Левенштейна його було модифіковано, додавши до розрахунку відстані редагування між словами, можливість отримати оцінку схожості символів з таблиці подібності. Даний метод було реалізовано. Для цього було створено саму таблицю подібності і реалізовано модифікований алгоритм Дамерау-Левенштейна. Оскільки тестування програмного забезпечення відіграє важливу роль в життєвому циклі розробки ПЗ, даний метод було протестовано.

Крім того було оцінено та проаналізовано продуктивність модифікованого алгоритму нечіткого пошуку. Аналіз було здійснено на великом обсязі текстових наборів даних, де вміст даних було модифіковано різними способами. Швидкодію та точність алгоритму з використанням таблиці подібності було порівнято з нечітким пошуком без використання таблиці. Перший алгоритм показує значно кращі результати нечіткого

пошуку, проте другий виконує пошук на такому самому наборі даних в середньому в 2.5 раз швидше.

Загалом порівнюючи ці два алгоритми можна дійти висновку, що нечіткий пошук з використанням таблиці подібності слід використовувати, коли для користувача більш важливі результати пошуку. А неточний пошук без таблиці слід використовувати, коли користувачу важлива саме швидкість обчислення

В якості перспективи розвитку алгоритму нечіткого пошуку в тексті з використанням таблиці подібності можна створити функціонал індексування текстових даних, наприклад, використовуючи BK-дерева. Це дозволить збільшити швидкодію нечіткого пошуку з використанням таблиці.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Damerau-Levenshtein distance [Електронний ресурс]. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance
2. Fred J. Damerau. A Technique for Computer Detection and Correction of Spelling Errors : Communications of the ACM, 1964, с. 171 – 176. [Електронний ресурс]. — Режим доступу до ресурсу: https://dl.acm.org/doi/pdf/10.1145/363958.363994
3. Подібність Джаро — Вінклера [Електронний ресурс]. — Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D1%96%D0%B1%D0%BD%D1%96%D1%81%D1%82%D1%8C_%D0%94%D0%B6%D0%B0%D1%80%D0%BE_%E2%80%94_%D0%92%D1%96%D0%BD%D0%BA%D0%BB%D0%B5%D1%80%D0%B0
4. М. В. Михайлова. ПОРІВНЯННЯ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ В ТЕКСТАХ УКРАЇНСЬКОЮ МОВОЮ : Радіоелектроніка, інформатика, управління, 2007, с. 80. [Електронний ресурс]. — Режим доступу до ресурсу: https://cyberleninka.ru/article/n/porivnyannya-algoritmiv-nechitkogo-poshuku-v-tekstah-ukrayinskoyu-movoyu/viewer
5. Bitap algorithm [Електронний ресурс]. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Bitap_algorithm
6. Fuzzy match algorithms explained [Електронний ресурс]. — Режим доступу до ресурсу: https://medium.com/data-science-in-your-pocket/fuzzy-matching-algorithms-explained-e0ff30cc00ca
7. Soundex [Електронний ресурс]. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Soundex

8. Damerau–Levenshtein distance [Електронний ресурс]. — Режим доступу до ресурсу: https://www.geeksforgeeks.org/damerau-levenshtein-distance/
9. Fancy Letters [Електронний ресурс]. — Режим доступу до ресурсу: https://symbl.cc/en/collections/fancy-letters/
10. GoogleTest Primer [Електронний ресурс]. — Режим доступу до ресурсу: http://google.github.io/googletest/primer.html
11. Google Benchmark User Guide [Електронний ресурс]. — Режим доступу до ресурсу: https://github.com/google/benchmark/blob/main/docs/user_guide.md#runtime-and-reporting-considerations

ДОДАТОК А

Лістинг коду для створення таблиці подібності.

Створення категорії симентично подібних символів:

```
for sec in sym_table:
    h = sec.find('h3')
    letter = h.text[-2].lower()

    similar_group = set(letter)
    sim_letters = sec.find_all('a')
    for sl in sim_letters:
        tag = sl['data-template']
        data_dict = json.loads(tag)
        sym = data_dict["symbol"]

        # correction of input data
        if letter == 'b' and sym == 'Ъ':
            continue
        if letter == 'n' and sym == 'ń':
            continue
        if sym == 'İ':
            continue

        if sym.islower():
            similar_group.add(sym)
        else:
            similar_group.add(sym.lower())

    for lt1 in similar_group:
        if lt1 not in data["data"].keys():
            data["data"][lt1] = dict()

        for lt2 in similar_group:
            if lt1 != lt2:
                data["data"][lt1][lt2] = COST_DICT["SIMILAR_CHAR_COST"]
```

Створення категорії можливих друкарських помилок:

```
# KEYBOARD MISTAKE
keyboard = [["q", "w", "e", "r", "t", "y", "u", "i", "o", "p"],
            ["a", "s", "d", "f", "g", "h", "j", "k", "l"],
            ["z", "x", "c", "v", "b", "n", "m"]]


def add_to_chars(char, *ch_set):
    for ch in ch_set:
        data["data"][char][ch] = COST_DICT["KEYBOARD_MISTAKE_COST"]


for ind, value in enumerate(keyboard[0]):
    data_set = []
    if ind == 0:
        data_set.extend((keyboard[0][ind + 1], keyboard[1][ind]))

    elif ind == len(keyboard[0]) - 1:
        data_set.extend((keyboard[0][ind - 1], keyboard[1][ind - 1]))

    else:
        data_set.extend((keyboard[0][ind - 1], keyboard[0][ind + 1],
```

```
                        keyboard[1][ind - 1], keyboard[1][ind]))
    add_to_chars(value, *data_set)

for ind, value in enumerate(keyboard[1]):
    data_set = []
    if ind == 0:
        data_set.extend((keyboard[0][ind], keyboard[0][ind + 1],
                         keyboard[1][ind + 1], keyboard[2][ind]))

    elif ind == len(keyboard[1]) - 1:
        data_set.extend((keyboard[0][ind], keyboard[0][ind + 1],
                         keyboard[1][ind - 1]))

    elif ind == len(keyboard[1]) - 2:
        data_set.extend((keyboard[0][ind], keyboard[0][ind + 1],
                         keyboard[1][ind - 1], keyboard[1][ind + 1],
                         keyboard[2][ind - 1]))

    else:
        data_set.extend((keyboard[0][ind], keyboard[0][ind + 1],
                         keyboard[1][ind - 1], keyboard[1][ind + 1],
                         keyboard[2][ind - 1], keyboard[2][ind]))
    add_to_chars(value, *data_set)

for ind, value in enumerate(keyboard[2]):
    data_set = []
    if ind == 0:
        data_set.extend((keyboard[1][ind], keyboard[1][ind + 1],
                         keyboard[2][ind + 1]))

    elif ind == len(keyboard[2]) - 1:
        data_set.extend((keyboard[1][ind], keyboard[1][ind + 1],
                         keyboard[2][ind - 1]))

    else:
        data_set.extend((keyboard[1][ind], keyboard[1][ind + 1],
                         keyboard[2][ind - 1], keyboard[2][ind + 1]))
    add_to_chars(value, *data_set)
```

Створення категорії візуально подібних символів:

```
# VISUAL SIMILAR CHAR
def add_similar_ch(chs):
    for ch1 in chs:
        for ch2 in chs:
            if ch1 == ch2:
                continue
            if ch2 not in data["data"][ch1].keys():
                data["data"][ch1][ch2] =
COST_DICT["VISUAL_SIMILAR_CHAR_COST"]
            elif COST_DICT["VISUAL_SIMILAR_CHAR_COST"] <
data["data"][ch1][ch2]:
                data["data"][ch1][ch2] =
COST_DICT["VISUAL_SIMILAR_CHAR_COST"]


add_similar_ch(("b", "d", "p", "q", "o"))
add_similar_ch(("i", "j", "l", "t"))
add_similar_ch(("u", "v", "w"))
add_similar_ch(("m", "n"))
```

```
add_similar_ch(("c", "e", "o", "s"))
add_similar_ch(("h", "k"))
add_similar_ch(("f", "t"))
add_similar_ch(("g", "q"))
add_similar_ch(("x", "y"))
add_similar_ch(("a", "o"))
add_similar_ch(("k", "x"))
add_similar_ch(("r", "n", "v"))
```

ДОДАТОК Б

Лістинг коду десереалізації таблиці подібності для використання під час нечіткого пошуку:

```
#include "simtable.h"

#include "rapidjson/document.h"
#include "rapidjson/filereadstream.h"
#include "rapidjson/istreamwrapper.h"
#include <fstream>
#include <iostream>
#include <cwctype>
#include <random>

typedef rapidjson::GenericDocument<rapidjson::UTF16LE<> >
WDocument;

SimTable::SimTable(const std::string& path)
{
    std::ifstream ifs(path, std::ios::binary);
    if (ifs.fail()) {
        throw std::ifstream::failure("Could not open file: " +
std::string(path));
    }

    rapidjson::IStreamWrapper isw(ifs);
    rapidjson::AutoUTFInputStream<unsigned ,
rapidjson::IStreamWrapper> eis(isw);

    WDocument document;
    document.ParseStream<0, rapidjson::AutoUTF<unsigned> >(eis);
    if (!document.IsObject()){
        throw std::runtime_error("Opened file is not JSON");
    }

    //PARSE COST
    for (auto key = document.MemberBegin(); key !=
document.MemberEnd(); key++) {
        if (key->name == L"cost") {
            for(auto costName = key->value.MemberBegin();
costName != key->value.MemberEnd(); costName++){
                std::wstring wstr = costName->name.GetString();
                std::string str(wstr.begin(), wstr.end());
                cost[str] = costName->value.GetFloat();
            }
            break;
        }
    }
```

```
    //PARSE DATA
    for (auto key = document.MemberBegin(); key !=
document.MemberEnd(); key++){
        if(key->name == L"data"){
            for (auto wch = key->value.MemberBegin(); wch !=
key->value.MemberEnd(); wch++){
                for (auto simWch = wch->value.MemberBegin();
simWch != wch->value.MemberEnd(); simWch++){
                    auto valueWch = wch->name.GetString()[0];
                    auto valueSimWch = simWch-
>name.GetString()[0];
                    auto price = simWch->value.GetFloat();

                    table[valueWch][valueSimWch] = price;
                }
            }
            break;
        }
    }
}
```

Лістинг коду для отримання ваги подібності символів:

```
std::optional<float> SimTable::getReplaceCost(wchar_t c1,
wchar_t c2) const
{
    if (c1 == c2){
        return 0.;
    }

    std::optional<float> result;

    bool isUpperC1 = std::iswupper(c1);
    bool isUpperC2 = std::iswupper(c2);


    wchar_t lowerC1 = isUpperC1 ? std::towlower(c1) : c1;
    wchar_t lowerC2 = isUpperC2 ? std::towlower(c2) : c2;
    if (lowerC1 == lowerC2){
        return cost.at("CAPITAL_COST");
    }

    auto simChars = table.find(lowerC1);
    if(simChars != table.end()){
        auto resPrice = simChars->second.find(lowerC2);
        if (resPrice != simChars->second.end()){
            result = resPrice->second;
        }
    }

    if ((isUpperC1 xor isUpperC2) and result.has_value())
        result.emplace(result.value() +
```

```
cost.at("CAPITAL_COST"));

    return result;
}
```

ДОДАТОК В

Лістинг коду для тестування

table_fixture.cpp:

```
#include "gtest/gtest.h"
#include "../src/simtable.h"

class TableFixture : public testing::Test {
protected:
    void SetUp() override {
        const std::string pathToFile = "..\\..\\simtable.json";
        st_ = new SimTable(pathToFile);
    }
    SimTable* st_;
};
```

simtable_unittest.cpp:

```
#include <random>
#include <codecvt>
#include "gtest/gtest.h"
#include "table_fixture.cpp"


TEST_F(TableFixture, Table){
    std::random_device rd;
    std::mt19937  generator(rd());
    std::uniform_int_distribution<int> distribution(0,  st_-
>table.size()-1);

    std::set<size_t> ids;
    while(ids.size() < 30)
        ids.insert(distribution(generator));

    size_t iter = 0;
    for (const auto& pair : st_->table){
        if (ids.find(iter) != ids.end()){
            for(const auto& simWch : pair.second){
                auto simWchIter = st_->table.find(simWch.first);
                EXPECT_NE(simWchIter, st_->table.end());
                EXPECT_NE(simWchIter->second.find(pair.first),
simWchIter->second.end());
            }
        }
        iter++;
```

```
    }
}

TEST_F(TableFixture, getReplaceCost){
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
converter;

// TEST SYMBOL A

    EXPECT_EQ(st_->getReplaceCost(L'a', L'q').value(), 0.5);

    auto testAA = st_->getReplaceCost(L'A', L'a').value();
    EXPECT_TRUE(0.09 < testAA and testAA < 0.11);

    EXPECT_EQ(st_->getReplaceCost(L'a', L'l'), std::nullopt);

    EXPECT_EQ(st_->getReplaceCost(L'a',
converter.from_bytes("á")[0]).value(), 0.25);

    auto testAO = st_->getReplaceCost(L'a', L'o').value();
    EXPECT_TRUE(0.59 < testAO and testAO < 0.61);


// TEST SYMBOL G

    EXPECT_EQ(st_->getReplaceCost(L'g', L't').value(), 0.5);

    auto testGG = st_->getReplaceCost(L'g', L'G').value();
    EXPECT_TRUE(0.09 < testGG and testGG < 0.11);

    EXPECT_EQ(st_->getReplaceCost(L'g', L'z'), std::nullopt);

    EXPECT_EQ(st_->getReplaceCost(L'g',
converter.from_bytes("𝑔")[0]).value(), 0.25);

    auto testGQ = st_->getReplaceCost(L'g', L'q').value();
    EXPECT_TRUE(0.59 < testGQ and testGQ < 0.61);


// TEST SYMBOL N

    EXPECT_EQ(st_->getReplaceCost(L'n', L'h').value(), 0.5);

    auto testNN = st_->getReplaceCost(L'N', L'n').value();
    EXPECT_TRUE(0.09 < testNN and testNN < 0.11);

    EXPECT_EQ(st_->getReplaceCost(L'n', L'w'), std::nullopt);

    EXPECT_EQ(st_->getReplaceCost(L'n',
converter.from_bytes("ŋ")[0]).value(), 0.25);

    auto testNM = st_->getReplaceCost(L'n', L'v').value();;
```

```
    EXPECT_TRUE(0.59 < testNM and testNM < 0.61);


// TEST SYMBOL R

    EXPECT_EQ(st_->getReplaceCost(L'r', L'e').value(), 0.5);

    auto testRR = st_->getReplaceCost(L'r', L'R').value();
    EXPECT_TRUE(0.09 < testRR and testRR < 0.11);

    EXPECT_EQ(st_->getReplaceCost(L'r', L'x'), std::nullopt);

    EXPECT_EQ(st_->getReplaceCost(L'r',
converter.from_bytes("ŕ")[0]).value(), 0.25);

    auto testRV = st_->getReplaceCost(L'r', L'v').value();
    EXPECT_TRUE(0.59 < testRV and testRV < 0.61);

// ANOTHER SYMBOLS

    EXPECT_EQ(st_->getReplaceCost(L'b',
converter.from_bytes("β")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'c',
converter.from_bytes("ç")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'd',
converter.from_bytes("đ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'e',
converter.from_bytes("ė")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'f',
converter.from_bytes("f́")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'h',
converter.from_bytes("ħ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'i',
converter.from_bytes("ì")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'j',
converter.from_bytes("j̇")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'k',
converter.from_bytes("ќ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'l',
converter.from_bytes("l̥")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'm',
converter.from_bytes("ḿ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'o',
converter.from_bytes("ô")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'p',
converter.from_bytes("p̄")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'q',
converter.from_bytes("q̇")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L's',
converter.from_bytes("ş")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L't',
converter.from_bytes("ţ")[0]).value(), 0.25);
```

```
    EXPECT_EQ(st_->getReplaceCost(L'u',
converter.from_bytes("ü")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'v',
converter.from_bytes("ṽ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'w',
converter.from_bytes("ẅ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'x',
converter.from_bytes("ẋ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'y',
converter.from_bytes("ɤ")[0]).value(), 0.25);
    EXPECT_EQ(st_->getReplaceCost(L'z',
converter.from_bytes("ẓ")[0]).value(), 0.25);
}
```

editdistance_unittest.cpp:

```
#include <codecvt>
#include "gtest/gtest.h"
#include "table_fixture.cpp"
#include "../src/tools.h"


TEST(editDistance, editDistanceWithoutTable) {
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
converter;

    EXPECT_EQ(editDistance(L"", L""), 0.);
    EXPECT_EQ(editDistance(L"open", L""), 4.);
    EXPECT_EQ(editDistance(L"", L"river"), 5.);
    EXPECT_EQ(editDistance(L"elephant", L"eephlant"), 2);
    EXPECT_EQ(editDistance(L"sunday", L"monday"), 2.);
    EXPECT_EQ(editDistance(L"stack", L"steak"), 2.);
    EXPECT_EQ(editDistance(L"Cat", L"car"), 2.);
    EXPECT_EQ(editDistance(L"lake", L"lame"), 1.);
    EXPECT_EQ(editDistance(L"write", L"right"), 4.);
    EXPECT_EQ(editDistance(L"house", L"horses"), 2.);
    EXPECT_EQ(editDistance(L"chair", L"choir"), 1.);
    EXPECT_EQ(editDistance(L"rain", L"train"), 1.);
    EXPECT_EQ(editDistance(L"open", L"opne"), 1.);
    EXPECT_EQ(editDistance(L"chair", L"chiar"), 1.);
    EXPECT_EQ(editDistance(L"examples", L"examlpe"), 2.);
    EXPECT_EQ(editDistance(L"algorithm", L"algorihm"), 1.);
    EXPECT_EQ(editDistance(L"elephant", L"eelphant"), 1.);
    EXPECT_EQ(editDistance(L"mississippi", L"missippi"), 3.);
    EXPECT_EQ(editDistance(L"elephant", L"relevant"), 3.);
    EXPECT_EQ(editDistance(L"hypothesis", L"hypocrisy"), 5.);
    EXPECT_EQ(editDistance(L"phenomenon", L"penitentiary"), 8.);
    EXPECT_EQ(editDistance(L"exaggerate", L"extravagant"), 8.);
    EXPECT_EQ(editDistance(L"necessity", L"negotiable"), 8.);
    EXPECT_EQ(editDistance(L"sufficient",
```

```
converter.from_bytes("insufficient")), 3.);
    EXPECT_EQ(editDistance(L"ambiguous",
converter.from_bytes("ambivalence")), 8.);
    EXPECT_EQ(editDistance(converter.from_bytes("accommodate"),
L"accomplish"), 7.);
    EXPECT_EQ(editDistance(L"disastrous",
converter.from_bytes("spontaneous")), 9.);
    EXPECT_EQ(editDistance(converter.from_bytes("exponential"),
L"experimental"), 6.);
}


TEST_F(TableFixture, editDistanceWithTable){
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
converter;
    float test;

    EXPECT_EQ(editDistance(L"aah", L"aah", *st_), 0.);
    EXPECT_EQ(editDistance(L"", L"", *st_), 0.);
    EXPECT_EQ(editDistance(L"Open", L"", *st_), 4.);
    EXPECT_EQ(editDistance(L"", L"River", *st_), 5.);

    // KEYBOARD MISTAKE

    EXPECT_EQ(editDistance(L"algorithm", L"algoritjm", *st_),
.5);
    EXPECT_EQ(editDistance(L"synchronixe", L"synchronize",
*st_), .5);
    EXPECT_EQ(editDistance(L"difficult", L"diffidult", *st_),
.5);
    EXPECT_EQ(editDistance(L"resiluent", L"resilient", *st_),
.5);
    EXPECT_EQ(editDistance(L"alliance", L"allisnce", *st_), .5);

    // VISUAL SIMILAR CHAR

    test = editDistance(L"necessary", L"recessary", *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);
    test = editDistance(L"opporlunity", L"opportunity", *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);
    test = editDistance(L"catastrophe", L"catactrophe", *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);
    test = editDistance(L"relevont", L"relevant", *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);
    test = editDistance(L"renovate", L"renonate", *st_);
    EXPECT_TRUE(0.59 < test and test < 0.61);

    // SIMILAR CHARS

    EXPECT_EQ(editDistance(L"white",
converter.from_bytes("whîte"), *st_), .25);
    EXPECT_EQ(editDistance( converter.from_bytes("brilliânt"),
```

```
L"brilliant", *st_), .25);
    EXPECT_EQ(editDistance(L"agile",
converter.from_bytes("agiḽe"), *st_), .25);
    EXPECT_EQ(editDistance( converter.from_bytes("anaŕchy"),
L"anarchy", *st_), .25);
    EXPECT_EQ(editDistance(L"elephant",
converter.from_bytes("elephant̩"), *st_), .25);

    // CAPITAL

    test = editDistance(L"Frog", L"frog", *st_);
    EXPECT_TRUE(0.09 < test and test < 0.11);
    test = editDistance(L"water", L"Water", *st_);
    EXPECT_TRUE(0.09 < test and test < 0.11);
    test = editDistance(L"Horse", L"horse", *st_);
    EXPECT_TRUE(0.09 < test and test < 0.11);

    // DIFFERENT COMBINATION

    test = editDistance(L"Illustrious", L"illustriuos", *st_);
    EXPECT_TRUE(1.09 < test and test < 1.11);

    EXPECT_EQ(editDistance(L"anthropology", L"amthopology",
*st_), 1.5);

    test = editDistance(L"Galaxy",
converter.from_bytes("gaɬaxy"), *st_);
    EXPECT_TRUE(0.34 < test and test < 0.36);

    test = editDistance(L"Whispre", L"Vhisper", *st_);
    EXPECT_TRUE(1.59 < test and test < 1.61);

    test = editDistance(L"Thunder", L"yhunlder", *st_);
    EXPECT_TRUE(1.59 < test and test < 1.61);

    test = editDistance(L"Raimcoat",
converter.from_bytes("raincaoɪ"), *st_);
    EXPECT_TRUE(1.84 < test and test < 1.86);

    test = editDistance(L"Flrbfty", L"Firefly", *st_);
    EXPECT_TRUE(2.19 < test and test < 2.21);

    test = editDistance(L"Velvet",
converter.from_bytes("welvĕt"), *st_);
    EXPECT_TRUE(0.94 < test and test < 0.96);

    test = editDistance(L"Castle",
converter.from_bytes("çasltc"), *st_);
    EXPECT_TRUE(1.94 < test and test < 1.96);

    test = editDistance(L"Carousel", L"caraucel", *st_);
    EXPECT_TRUE(1.29 < test and test < 1.31);
```

```
    EXPECT_EQ(editDistance(converter.from_bytes("Sailíng"),
L"Ssiling", *st_), 0.75);

    EXPECT_EQ(editDistance(L"Luaghtr", L"Laughter", *st_), 2.);

    test = editDistance(L"Puzzle",
converter.from_bytes("ƿussle"), *st_);
    EXPECT_TRUE(1.35 < test and test < 1.36);

    test = editDistance(L"Mavshnalow", L"marshmallow", *st_);
    EXPECT_TRUE(2.19 < test and test < 2.21);

    test = editDistance(L"Serenade",
converter.from_bytes("sěrěnadě"), *st_);
    EXPECT_TRUE(0.84 < test and test < 0.86);

    test = editDistance(L"wamdevlust", L"Wanderlust", *st_);
    EXPECT_TRUE(1.19 < test and test < 1.21);

    test = editDistance(L"Slardust", L"Star", *st_);
    EXPECT_TRUE(4.59 < test and test < 4.61);

    test = editDistance(converter.from_bytes("Prělěvanr"),
L"Relevant", *st_);
    EXPECT_TRUE(2.09 < test and test < 2.11);

    test = editDistance(L"Accomplish", L"accamptish", *st_);
    EXPECT_TRUE(1.29 < test and test < 1.31);

    test = editDistance(L"Eccentric",
converter.from_bytes("essentrīs"), *st_);
    EXPECT_TRUE(2.14 < test and test < 2.16);
}
```

ДОДАТОК Г

Лістинг коду для тестування швидкодії та точності алгоритмів нечіткого пошуку з використанням таблиці подібності символів і без:

table_benchmark.cpp:

```
#include "benchmark/benchmark.h"
#include "../src/simtable.h"

static void BM_TABLE(benchmark::State& state) {
    for (auto _ : state) {

benchmark::DoNotOptimize(SimTable("..\\..\\simtable.json"));
    }
}
```

fuzzy_search_benchmark.cpp:

```
#include "fuzzy_search_benchmark.h"
#include "../src/tools.h"

#include <fstream>
#include <sstream>
#include <random>
#include <codecvt>
#include <algorithm>

void BenchmarkWithoutTable(benchmark::State& state, const
InputData& data) {
    for (auto _ : state) {
        size_t countMatch = 0;
        auto errorThreshold = state.range(1) + 0.01;
        for(const auto& word : data.words){
            auto result = editDistance(data.requestWord, word);
            benchmark::DoNotOptimize(result);

            if (result < errorThreshold)
                countMatch++;

        }
        state.counters["MATCHES"] = countMatch;
    }
}

void BenchmarkWithTable(benchmark::State& state, const
InputData& data, const SimTable& table) {
```

```
    for (auto _ : state) {
        size_t countMatch = 0;
        auto errorThreshold = state.range(1) + 0.01;
        for(const auto& word : data.words){
            auto result = editDistance(data.requestWord, word,
table);
            benchmark::DoNotOptimize(result);

            if (result < errorThreshold)
                countMatch++;
        }
        state.counters["MATCHES"] = countMatch;
    }
}
```

benchmark_util.cpp:

```
#include "benchmark_util.h"

#include <iostream>
#include <random>
#include <algorithm>
#include <fstream>
#include <unordered_map>
#include <codecvt>

std::vector<std::wstring> getWords(const std::string& path){
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
converter;

    std::ifstream words(path);
    if(!words.is_open())
        std::cerr << "Failed to open the file: " << "words.txt"
<< std::endl;

    std::vector<std::wstring> result;

    std::string line;
    while (getline(words, line))
        result.emplace_back(converter.from_bytes(line));

    return result;
}

std::vector<wchar_t> getShuffledAlphabet(){
    std::random_device rd;
    std::mt19937  generator(rd());

    std::vector<wchar_t> englishSymbols;
    for (auto i = 97; i < 123; ++i)
```

```
            englishSymbols.emplace_back(static_cast<wchar_t>(i));

    std::shuffle(englishSymbols.begin(), englishSymbols.end(),
generator);

    return englishSymbols;
}

std::vector<wchar_t> getShuffledAlphabet(std::wstring word){
    std::random_device rd;
    std::mt19937  generator(rd());

    std::set<wchar_t> alBegin;
    for(auto wch : word)
        alBegin.insert(wch);

    std::vector<wchar_t> englishSymbols;
    for (auto i = 97; i < 123; ++i) {
        auto wch = static_cast<wchar_t>(i);
        if(alBegin.find(wch) == alBegin.end())

englishSymbols.emplace_back(static_cast<wchar_t>(i));
    }

    std::shuffle(englishSymbols.begin(), englishSymbols.end(),
generator);

    englishSymbols.insert(englishSymbols.begin(),
alBegin.begin(), alBegin.end());

    return englishSymbols;
}

void changeSymbols(std::vector<std::wstring>& words, wchar_t
oldWch, wchar_t newWch){
    for(auto& word : words){
        std::replace(word.begin(), word.end(), oldWch, newWch);
    }
}

std::wstring getRandomWord(std::vector<std::wstring> words){
    std::random_device rd;
    std::mt19937  generator(rd());
    std::uniform_int_distribution<int> distribution(0,
words.size()-1);
    return words[distribution(generator)];
}

std::wstring getRandomWord(std::vector<std::wstring> words, int
len){
    std::vector<int> temp;

    for (size_t i = 0; i < words.size(); i++)
```

```
        if (words[i].size() == len)
            temp.emplace_back(i);

    std::random_device rd;
    std::mt19937  generator(rd());
    std::uniform_int_distribution<int> distribution(0,
temp.size()-1);
    return words[temp[distribution(generator)]];
}
```

benchmark_main.cpp:

```
#include "table_benchmark.cpp"
#include "benchmark_util.h"
#include "fuzzy_search_benchmark.h"

#include <fstream>
#include <codecvt>
#include <iostream>
#include <string>
#include <cctype>
#include <cwctype>

std::wstring strip(const std::wstring& input) {
    if(input.length() == 0)
        return input;

    std::size_t start = 0;
    std::size_t end = input.length() - 1;

    while (start <= end && std::iswspace(input[start])) {
        start++;
    }

    while (end >= start && std::iswspace(input[end])) {
        end--;
    }

    return input.substr(start, end - start + 1);
}

std::wstring removePunctuation(const std::wstring& input) {
    std::wstring result;
    for (auto c : input) {
        if (!std::iswpunct(c)) {
            result += c;
        }
    }
    return result;
}
```

```
std::vector<std::wstring> readFile(const std::string& path){
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
converter;

    std::ifstream words(path);
    if(!words.is_open())
        std::cerr << "Failed to open the file: " << "words.txt"
<< std::endl;

    std::vector<std::wstring> result;

    std::string line;
    while (getline(words, line)) {
        std::wstringstream
ss(strip(converter.from_bytes(line)));
        std::wstring word;
        while (getline(ss, word, L' ')){
            result.emplace_back(removePunctuation(word));
        }
    }
    words.close();
    return result;
}

int getErrorThreshold(int len){
    return len/3;
}

int main(int argc, char** argv) {
    std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>>
converter;

    benchmark::Initialize(&argc, argv);

    BENCHMARK(BM_TABLE)->Unit(benchmark::kMillisecond);

    const std::string pathToSimTable = "..\\..\\simtable.json";

    SimTable table(pathToSimTable);

    //    TEST ALPHABET CHANGE

    const std::string pathToTestWords =
(R"(..\..\benchmarks\words_test.txt)");

    InputData dataAlp;
    dataAlp.words =
std::make_shared<std::vector<std::wstring>>(getWords(pathToTestW
ords));
    dataAlp.requestWord = getRandomWord(*dataAlp.words);

    auto shuffledAlphabet =
getShuffledAlphabet(dataAlp.requestWord);
```

```
    for(auto amountOfSymbolsToReplace = 1;
amountOfSymbolsToReplace < 10; amountOfSymbolsToReplace++){
        auto oldSym = shuffledAlphabet[amountOfSymbolsToReplace-
1];
        auto newSym = table.getRandomSimChar(oldSym);
        changeSymbols(*dataAlp.words, oldSym, newSym);
        for(auto errorThreshold = 0; errorThreshold <= 350;
errorThreshold += 50){

benchmark::RegisterBenchmark("ALBenchmarkWithoutTable",
[dataAlp](benchmark::State& state){
                BenchmarkWithoutTable(state, dataAlp);
            })
            ->Args({amountOfSymbolsToReplace, errorThreshold})
            ->Unit(benchmark::kMillisecond)
            ->Iterations(1);

            benchmark::RegisterBenchmark("ALBenchmarkWithTable",
[dataAlp, table](benchmark::State& state){
                BenchmarkWithTable(state, dataAlp, table);
            })
            ->Args({amountOfSymbolsToReplace, errorThreshold})
            ->Unit(benchmark::kMillisecond)
            ->Iterations(1);
        }
    }


    const std::string pathToEnglishText =
(R"(..\..\test_text\HP-English.txt)");
    const std::string pathToGermanText = (R"(..\..\test_text\HP-
German.txt)");
    const std::string pathToCzechText = (R"(..\..\test_text\HP-
Czech.txt)");

    const auto& engText =
std::make_shared<std::vector<std::wstring>>(readFile(pathToEngli
shText));
    const auto& gerText =
std::make_shared<std::vector<std::wstring>>(readFile(pathToGerma
nText));
    const auto& czText =
std::make_shared<std::vector<std::wstring>>(readFile(pathToCzech
Text));

    std::vector<InputData> data;
    data.emplace_back(converter.from_bytes("cloudy"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("Cloudy"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("cioufy"), engText,
```

```
"English_Text");
    data.emplace_back(converter.from_bytes("clóudy"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("clóůdy"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("clóůdý"), engText,
"English_Text");

    data.emplace_back(converter.from_bytes("thousand"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("Thousand"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("thuosamd"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("thöusand"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("thöüsand"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("thöüsänd"), engText,
"English_Text");

    data.emplace_back(converter.from_bytes("strange"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("Strange"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("sttanqe"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("štrange"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("štřange"), engText,
"English_Text");
    data.emplace_back(converter.from_bytes("štřánge"), engText,
"English_Text");

    data.emplace_back(converter.from_bytes("geändert"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("Geändert"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("Qeänberl"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("qeandert"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("geándert"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("geándeřť"), gerText,
"German_Text");

    data.emplace_back(converter.from_bytes("völlig"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("Völlig"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("Vöttiq"), gerText,
"German_Text");
```

```
    data.emplace_back(converter.from_bytes("vokkig"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("vóllig"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("vóllíg"), gerText,
"German_Text");

    data.emplace_back(converter.from_bytes("erzählte"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("Erzählte"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("erxähite"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("Erzahlte"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("eřzáhlte"), gerText,
"German_Text");
    data.emplace_back(converter.from_bytes("ěřzáhltě"), gerText,
"German_Text");

    data.emplace_back(converter.from_bytes("každý"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("Každý"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("Kšždý"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("kazdý"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("kazdy"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("käzdy"), czText,
"Czech_Text");

    data.emplace_back(converter.from_bytes("měsíčních"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("Měsíčních"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("Měsíčmícj"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("měsičnich"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("mesičnich"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("mecicnich"), czText,
"Czech_Text");

    data.emplace_back(converter.from_bytes("stříbrné"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("Stříbrné"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("dtřídrné"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("stríbrné"), czText,
```

```
"Czech_Text");
    data.emplace_back(converter.from_bytes("stribrné"), czText,
"Czech_Text");
    data.emplace_back(converter.from_bytes("stridrne"), czText,
"Czech_Text");



    for (const auto& inputData : data) {
        benchmark::RegisterBenchmark("HPBenchmarkWithoutTable",
[inputData](benchmark::State &state) {
            BenchmarkWithoutTable(state, inputData);
        })
                -
>Args({getErrorThreshold(inputData.requestWord.length())})
                ->Unit(benchmark::kMillisecond)
                ->Iterations(1);

        benchmark::RegisterBenchmark("HPBenchmarkWithTable",
[inputData, table](benchmark::State &state) {
            BenchmarkWithTable(state, inputData, table);
        })
                -
>Args({getErrorThreshold(inputData.requestWord.length())})
                ->Unit(benchmark::kMillisecond)
                ->Iterations(1);
    }
        benchmark::RunSpecifiedBenchmarks();
        benchmark::Shutdown();

    std::wofstream file(R"(..\..\test_text\res_additional.txt)",
std::ios::binary | std::ios::trunc);
    const unsigned long MaxCode = 0x10ffff;
    const std::codecvt_mode Mode = std::generate_header;
    std::locale utf16_locale_data(file.getloc(), new
std::codecvt_utf16<wchar_t, MaxCode, Mode>);
    file.imbue(utf16_locale_data);
    if (!file.is_open())
        std::cerr << "Failed to create the file: " <<
R"(..\..\test_text\res_additional.txt)" << std::endl;

    for(const auto& inputData : data){
        file << inputData.requestWord << converter.from_bytes("
") << converter.from_bytes(inputData.name) <<
converter.from_bytes('\n');
    }
    file.close();
}
```