Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

КЛЕЩ КИРИЛО ОЛЕГОВИЧ

УДК 004.02

ДИСЕРТАЦІЯ

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ З ВИКОРИСТАННЯМ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ

122 – Комп’ютерні науки
Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____К.О. Клещ

Науковий керівник: д.т.н., проф. Петренко Анатолій Іванович

Київ – 2024

АНОТАЦІЯ

Клещ К.О. Підвищення ефективності алгоритмів нечіткого пошуку з використанням таблиці подібності символів. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки». – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», 2024.

Дисертаційне дослідження присвячене підвищенню швидкодії алгоритмів і розробці методу нечіткого пошуку з використанням таблиці подібності символів, для пошуку найбільш релевантних документів до пошукової фрази. В роботі створено метод нечіткого пошуку, який складається з 9 послідовних кроків та потрібен для швидкого пошуку співпадінь на великому наборі текстових даних. За допомогою цього методу була створена система нечіткого пошуку, яка дозволила вирішити задачу пошуку найрелевантніших документів до пошукової фрази з набору таких документів.

Розроблений метод нечіткого пошуку комбінує переваги алгоритмів на основі детермінованих скінченних автоматів та алгоритмів на основі динамічного програмування для підрахунку відстані Дамерау-Левенштейна. Така комбінація дозволила впровадити таблицю подібності символів оптимальним чином. В рамках роботи запропоновано підхід та спосіб створення таблиці подібності символів та розроблено приклад такої таблиці для символів з англійського алфавіту, що дозволило знаходити міру подібності поміж двома символами з константною асимптотикою та перетворювати поточний символ в його базовий аналог. Для фільтрування й сортування документів було розроблено метод оцінювання відповідності текстових даних до пошукової фрази на основі метрики, яка одночасно враховує кількість знайдених і незнайдених символів та кількість знайдених і незнайдених слів. Алгоритм Дамерау-Левенштейна дозволяє знаходити відстань редагування поміж двома словами, враховуючи помилки наступних типів: заміна, видалення, додавання

та транспозиція символів. В рамках роботи була запропонована модифікація цього алгоритму за допомогою використання таблиці подібності для більш точної оцінки відстані редагування між двома словами.

На основі розробленого методу була створена система нечіткого пошуку, яка дозволить знаходити шукані результати швидше та підвищить релевантність отриманих результатів шляхом їхнього сортування відповідно до значень розробленої метрики подібності тестових даних. Також у роботі було досліджено, проаналізовано та надано рекомендації, яким чином можна інтегрувати особливості та потужності таблиці подібності символів з алгоритмом нечіткого пошуку Дамерау-Левенштейна. Дослідження алгоритмів нечіткого пошуку в тексті є важливою темою в галузі обробки тексту та інформаційного пошуку. Це обумовлено зростаючим обсягом текстової інформації і ймовірністю помилок через вплив людського фактору при написанні тексту та створенні текстового контенту. Нечіткий пошук використовує алгоритми для пошуку даних в тексті, які приблизно відповідають шаблону. Це досягається шляхом зіставлення та порівняння рядків або ключових слів, які можуть бути схожими між собою, але не ідентичними.

У **першому розділі** дисертаційної роботи були розглянуті та проаналізовані різні алгоритми нечіткого пошуку. Такі як: алгоритм Дамерау-Левенштейна, алгоритм N-грам, алгоритм Джаро-Вінклера, алгоритм Bitap, звичайний алгоритм Левенштейна, алгоритм SoundE та алгоритми на основі скінченних автоматів. У ролі оптимального алгоритму пошуку відстані редагування між двома словами було обрано алгоритм Дамерау-Левенштейна, бо він дозволяє впровадити таблицю подібності оптимальним чином. Також були розглянуті алгоритми на основі скінченних автоматів, а саме: автомат на основі префіксного дерева, автомат на основі таблиці та автомат на основі хешування. Перші два виявились неефективними через певні недоліки, а останній виявився оптимальним та найбільш універсальним з точки зору швидкодії роботи та часу побудови, а також об’єму витраченої пам’яті.

У **другому розділі** дисертаційної роботи було розроблено та покроково описано метод нечіткого пошуку, який дозволяє знаходити найрелевантніші документи до пошукової фрази.

Також були розглянуті переваги та недоліки застосування таблиці подібності символів, підходи та способи її побудови. Було створено приклад таблиці подібності для символів з англійської мови за допомогою групування символів у JSON файлі. Використання таблиці подібності покращує отримані результати, особливо при використанні мов зі спеціальними символами. Це дозволяє знаходити набагато більше релевантних результатів, проте швидкодія алгоритму може зменшитись. За допомогою використання такої таблиці підвищується релевантність відповідних документів навіть при наявності орфографічних помилок, скорочень, слів-синонімів або інших форм неточностей у запиті. Підхід із використанням таблиці подібності символів може бути використаний у системах перевірки орфографії та автоматичної корекції, системах автозавершення та автодоповнення, а також у реалізації функцій з виявлення дублікатів даних та плагіату.

У **третьому розділі** дисертаційної роботи проведено аналіз коректності результатів та ефективності алгоритмів нечіткого пошуку з використанням таблиці і без, а також алгоритму пошуку точного збігу. Було проведено перевірку всіх 4 алгоритмів нечіткого пошуку на основі скінченних автоматів на коректність роботи, а також розроблено та проаналізовано тести продуктивності для різних вхідних даних. Для перевірки коректності було розроблено програму, що використовує словник слів та порівнює редагувальну відстань, обчислену поточним рішенням, із редагувальною відстанню, обчисленою за допомогою готового бібліотечного рішення.

Для перевірки продуктивності було проведено тестування, що визначає час побудови автомата та час перевірки слова для кожного з рішень. Також була реалізована система нечіткого пошуку на основі запропонованого методу та впроваджена у веб-додаток для пошуку найбільш релевантних документів.

Отримані результати можуть бути корисними та використані в різних галузях, де потрібно ефективно та швидко проводити нечіткий пошук у великих обсягах даних, наприклад, при пошуку подібних документів у пошукових системах або при автокорекції помилок.

В рамках роботи було: реалізовано метод нечіткого пошуку, який комбінує переваги алгоритмів на основі детермінованих скінченних автоматів та алгоритмів на основі динамічного програмування для підрахунку відстані Дамерау-Левенштейна; запропоновано технологію створення таблиці подібності символів у розроблений метод та створено приклад такої таблиці для символів з англійського алфавіту, що дозволило знаходити міру подібності двох символів із константною асимптотикою та перетворювати поточний символ в його базовий аналог; модифіковано метод оцінювання відстані редагування між двома словами за допомогою використання таблиці подібності в алгоритмі Дамерау-Левенштейна; запропоновано метод оцінювання відповідності текстових даних до пошукової фрази на основі метрики, яка одночасно враховує кількість знайдених/незнайдених слів та символів.

Для реалізації декількох кроків розробленого методу було запропоновано технологію до створення таблиці подібності символів, яка дозволить враховувати можливу семантичну подібність символів в словах. Оцінюючи подібність на основі різних критеріїв, таких як: форма, контекст і фонетика, таблиця подібності символів дозволила модифікувати алгоритм нечіткого пошуку, який може ранжувати відповідні результати навіть за наявності великої кількості неспівпадінь unicode значень схожих символів. Це, в свою чергу, збільшило кількість релевантних результатів на 10 %, в залежності від довжини пошукової фрази.

Також було впроваджено реалізований метод у систему для пошуку найбільш релевантних електронних листів, документів або текстів у середовищі для резервного копіювання.

Ключові слова: нечіткий пошук, таблиця подібності символів, алгоритм Дамерау-Левенштейна, скінченний автомат, метрика подібності, обробка текстових даних, модель, об’єкт, аналіз, експертна система, нечітка логіка, пошук за зразком, пошук даних, пошук відповідностей, бенчмарки.

ABSTRACT

Kyrylo Kleshch. Improving the efficiency of fuzzy search algorithms with the usage of symbols similarity table. – Qualification scientific work as manuscript.

Doctor of Philosophy dissertation under 122 «Computer Science» specialty. – National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute," Kyiv, 2024.

The dissertation research is dedicated to improving the speed of the algorithms and developing a fuzzy search method with the usage of symbols similarity table to find the most relevant documents for a search phrase. The work presents a fuzzy search method consisting of 9 sequential steps necessary for quickly finding matches in a large set of text data. Using this method, a fuzzy search system was created and it can solve the problem of finding the most relevant documents for a search phrase from a set of such documents.

The developed fuzzy search method combines the advantages of the algorithms based on deterministic finite automata and algorithms based on dynamic programming for calculating the Damerau-Levenshtein distance. This combination allowed to implement the symbols similarity table optimally. The work proposed an approach and method for creating a symbols similarity table, so the example of this table for symbols from the English alphabet was created, allowing for the measurement of similarity between two symbols with the constant asymptotics and transforming the current symbol into its base analog. For filtering and sorting documents, a method for evaluating the correspondence of text data to a search phrase based on a metric was developed, which simultaneously considers the number of found and unfound symbols and the number of found and unfound words. The Damerau-Levenshtein algorithm allows to find the editing distance between two words, considering the errors of the following types: substitution, deletion, addition, and transposition of symbols. In the course of the work, a modification of this algorithm was proposed using a similarity table for a more accurate estimation of the editing distance between two words.

The developed method allowed to create a fuzzy search system that would help to find the desired results faster and increase the relevance of the obtained results by sorting them according to the values of the developed similarity metric of test data. Additionally, the work investigated, analyzed, and provided recommendations on how to integrate the features and capabilities of the symbols similarity table with the Damerau-Levenshtein fuzzy search algorithm. Research on fuzzy search algorithms in text is an important topic in the field of text processing and information retrieval. The reason is the increasing volume of textual information and the likelihood of errors due to the influence of human factors in writing text and creating textual content. Fuzzy search uses algorithms to search for data in text that approximately match the pattern. This is achieved by comparing and matching strings or keywords that may be similar but not identical.

In the **first chapter** of the dissertation, various fuzzy search algorithms were considered and analyzed, such as the Damerau-Levenshtein algorithm, the N-gram algorithm, the Jaro-Winkler algorithm, the Bitap algorithm, the standard Levenshtein algorithm, the SoundEx algorithm, and algorithms based on finite automata. The Damerau-Levenshtein algorithm was chosen as the optimal algorithm for searching the editing distance between two words because it allows for an optimal implementation of the similarity table. Algorithms based on finite automata were also considered, namely: an automaton based on a prefix tree, an automaton based on a table, and an automaton based on hashing. The first two options were found to be inefficient due to certain drawbacks, while the latter proved to be optimal and the most versatile in terms of the operation speed, construction time, and memory consumption.

In the **second chapter** of the dissertation, a fuzzy search method was developed and described step-by-step, allowing for finding the most relevant documents for a search phrase.

The advantages and disadvantages of using a symbols similarity table, approaches, and methods of its construction were also discussed. An example of a symbols similarity table for symbols from the English language was created using symbols grouping in a JSON

file. The usage of the symbols similarity table improves the obtained results, especially when using languages with special symbols. This allows for finding significantly more relevant results, although the speed of the algorithm may decrease. Using this table enhances the relevance of the corresponding documents even in the presence of spelling mistakes, abbreviations, synonyms, or other inaccuracies in the query. The approach using a symbols similarity table can be used in spelling check and automatic correction systems, auto-completion and auto-suggestion systems, as well as in implementing functions for detecting data duplicates and plagiarism.

In the **third chapter** of the dissertation, an analysis of the correctness of results and the efficiency of fuzzy search algorithms with and without using a table, as well as the exact match search algorithm, was conducted. All four fuzzy search algorithms based on finite automata were tested for correctness, and performance tests were developed and analyzed for various input data. To verify correctness, a program was developed and it used a word dictionary and compared the editing distance calculated by the current solution with the editing distance calculated using a ready-made library solution.

To test performance, a testing was conducted to determine the time it takes to construct the automaton and the time it takes to check a word for each of the solutions. Additionally, a fuzzy search system based on the developed method was implemented and integrated into a web application for searching the most relevant documents.

The obtained results can be useful and applicable in various fields where the efficient and fast fuzzy search is required in large volumes of data, such as searching for similar documents in search systems or for an error auto-correction.

In the course of the work, the following was accomplished: the implementation of a fuzzy search method that combines the advantages of algorithms based on deterministic finite automata and algorithms based on dynamic programming for calculating the Damerau-Levenshtein distance; a proposal of a technology for creating a symbols similarity table in the developed method and creation of an example table for symbols from the English

alphabet, enabling the measurement of similarity between two symbols with constant asymptotics and transforming the current symbol into its base analog; the modification of the method for estimating the editing distance between two words using the similarity table in the Damerau-Levenshtein algorithm; a proposal of a method for evaluating the correspondence of text data to a search phrase based on a metric that simultaneously considers the number of found/unfound words and symbols.

For the implementation of several steps of the developed method, a technology was proposed for creating a symbols similarity table that allows for considering possible semantic similarity of the symbols in words. Assessing similarity based on various criteria such as form, context, and phonetics, the symbols similarity table allowed for modifying the fuzzy search algorithm, which can rank relevant results even in the presence of a large number of mismatches of Unicode values of similar symbols. This, in its turn, increased the number of relevant results by 10%, depending on the length of the search phrase.

Additionally, the implemented method was integrated into a system for searching the most relevant emails, documents, or texts in a backup environment.

Keywords: fuzzy search, symbols similarity table, Damerau-Levenshtein algorithm, finite automaton, similarity metric, text data processing, model, object, analysis, expert system, fuzzy logic, pattern matching, data search, match search, benchmarks.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Статті у наукових фахових виданнях України

[1] Kleshch, K., & Shablii, V. (2023). Comparison of fuzzy search algorithms based on Damerau-Levenshtein automata on large data. Technology Audit and Production Reserves, 4(2(72), 27–32. https://doi.org/10.15587/2706-5448.2023.286382.

[2] Клещ, К. О., & Царьов, М. О. (2023). МОДИФІКАЦІЯ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ ДЛЯ ВИКОРИСТАННЯ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ. Таврійський науковий вісник. Серія: Технічні науки, (3), 21-28. https://doi.org/10.32782/tnv-tech.2023.3.3.

[3] Kleshch, K. (2024). Development of fuzzy search method for creating an efficient information search system in text data. Technology Audit and Production Reserves, 1 (2 (75)), 20–24. doi: https://doi.org/10.15587/2706-5448.2024.298425.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

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

NLP (Natural Language Processing) – галузь комп'ютерних наук, яка займається задачами комп'ютерного аналізу та синтезу природної мови.

UNICODE (Universal Coded Character Set) – стандарт кодування символів та літер, який визначає уніфіковану універсальну систему для представлення тексту.

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

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

Асимптотична складність – поняття, яке використовується для аналізу алгоритмів для визначення темпу зростання просторової складності або часу алгоритму.

ДСА – детермінований скінченний автомат.

НСА – недетермінований скінченний автомат.

Експертна система (ЕС) – інтелектуальна система, призначена для розв’язання слабо формалізованих задач, на основі накопиченого в базі знань досвіду роботи експертів.

GoogleTest (Google C++ Testing Framework або gtest) – популярна бібліотека для автоматичного та автоматизованого тестування програм мовою програмування C++.

IDE (Integrated development environment) – інтегроване середовище розробки, що надає розробникам зручні інструменти для написання, налагодження, тестування й керування програмним кодом.

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

Google Benchmark – популярна бібліотека для вимірювання та аналізу продуктивності коду мовою програмування C++.

ПЗ – програмне забезпечення.

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

ВСТУП

Актуальність теми. У сучасному світі великі обсяги текстових даних є важливою складовою у багатьох галузях, таких як: торгівля, медицина, наука, економіка та інформаційні технології. Нечіткий пошук дозволяє ефективно знаходити інформацію навіть у випадках, коли в ній є неточності, помилки або вона є неповною. Однак, пошук потрібної інформації у таких великих масивах текстових даних може бути трудомістким і затратним по часу. Отже, нечіткий пошук допомагає зробити пошук більш гнучким та ефективним, особливо у випадках, коли точний пошук може бути ускладнений через різноманітність варіантів та помилки вводу.

Тема нечіткого пошуку широко висвітлена в різних наукових та практичних роботах. Деякі з найвідоміших досліджень та публікацій на цю тему включають:

- “Approximate string matching” від N. Gonzalo, в роботі виконаний аналіз алгоритмів нечіткого пошуку та розглянуті різні підходи пошуку інформації та їх застосування [1].
- “Fuzzy Sets and Systems” від Lotfi A. Zadeh, в роботі представлена теорія нечітких множин, яка є основою для багатьох методів нечіткого пошуку та інших алгоритмів обробки нечітких даних [2].
- “Information Retrieval” від Christopher D. Manning, в роботі розглянуті різні аспекти пошукових систем [3].
- “Fuzzy Matching Algorithms for Record Linkage in Big Data” від Peter Christen, робота досліджує застосування алгоритмів нечіткого пошуку для з'єднання записів великих наборів даних [4].
- “A Fast and Flexible Algorithm for Solving the All-Pairs Similarity Problem in Big Data” від Anh Dinh, робота пропонує швидкий алгоритм для вирішення проблеми пошуку схожих пар у великих наборах даних, який може бути застосований у нечіткому пошуку [5].

- “Efficient and Effective Duplicate Detection in Hierarchical Data” від Markus Endres, в роботі розглянуті методи нечіткого пошуку для виявлення дублікатів у ієрархічних даних [6].

Найновіші дослідження у сфері пошуку релевантних документів можна знайти у свіжих наукових статтях та конференційних доповідях. “A Deep Learning Approach for Document Relevance Ranking” [7] – в статті розглянуто використання глибокого навчання для покращення ранжування релевантних документів. “A Query-Focused Multi-View Learning Approach for Document Relevance Ranking” [8] – у цій роботі висвітлено підхід до ранжування релевантних документів з використанням багатовидового навчання. “Neural Information Retrieval: A Literature Review” [9] – цей огляд літератури присвячений застосуванню нейромереж у пошуку інформації, включаючи ранжування релевантних документів. “Interactive Deep Learning for Document Relevance Ranking” [10] – у цій праці досліджується інтерактивне глибоке навчання для ранжування релевантних документів з використанням зворотного зв'язку користувача.

Зв’язок роботи з науковими програмами, планами, темами. Наукові дослідження виконувались в рамках тематичного плану науково-дослідних робіт кафедри системного проектування. Результати дисертації використані в проектах НН ІПСА з підтримки та супроводження грід-центру засвідчення сертифікатів користувачів і грід-сайтів національної грід-інфраструктури: НДР № 2299/20 (номер держреєстрації 0120U103046), НДР № 2302/21 (номер держреєстрації 0121U110624), НДР № 2307/22 (номер держреєстрації 0122U002655), які виконувались згідно Програми інформатизації НАН України на 2020 – 2024 р.

На основі отриманих теоретичних і практичних результатів дисертаційного дослідження створено методичне забезпечення дисципліни «Алгоритми і структури даних», яка впроваджена у навчальному процесі кафедри системного проектування.

Мета і завдання дослідження. Метою дисертаційного дослідження є підвищення швидкодії та релевантності результатів алгоритмів нечіткого пошуку, шляхом розробки методу з використанням таблиці подібності символів, для пошуку найбільш релевантних документів до пошукової фрази.

Для досягнення поставленої мети необхідно розв’язати наступні завдання:

- дослідити існуючі методи нечіткого пошуку, виокремити ті з них, які можна застосувати разом із таблицею подібності символів;
- проаналізувати недоліки методів нечіткого пошуку та запропонувати варіанти покращення їхньої швидкодії;
- вибрати найбільш підходящий метод нечіткого пошуку слова в тексті для розв'язуваної задачі;
- модифікувати вибраний метод для підтримки таблиці подібності символів;
- розробити таблицю подібності для символів з англійського алфавіту;
- розробити ефективну з точки зору швидкодії, технологію пошуку речення або набору слів в тексті;
- провести тестування та обчислювальні експерименти;
- впровадити розроблену систему у середовище для зберігання резервних копій даних;

Об’єкт дослідження. Об’єктом дисертаційного дослідження є процеси інформаційного пошуку в наборі текстових даних.

Предмет дослідження. Предметом дисертаційного дослідження є методи та структури даних для пошуку текстової інформації в наборі текстових даних.

Методи дослідження. В дослідженні використовувались методи теорії прийняття рішень для експертного оцінювання ранжування результатів, методи ООП для реалізації системи нечіткого пошуку, методи теорії автоматів для побудови скінченних автоматів за пошуковим словом, методи нечіткої логіки для реалізації

шкали вимірювання ознак, методи інформаційного пошуку для знаходження подібності текстових даних.

Наукова новизна отриманих результатів.

1. *Вперше розроблено* метод нечіткого пошуку, який комбінує переваги алгоритмів на основі детермінованих скінченних автоматів та алгоритмів на основі динамічного програмування для підрахунку відстані Дамерау-Левенштейна. Така комбінація дозволила впровадити таблицю подібності символів оптимальним чином.

2. *Удосконалено технологію* створення таблиці подібності символів, яка відрізняється від існуючих структурою, що дозволило знаходити міру подібності двох символів із константною асимптотикою та перетворювати поточний символ в його базовий аналог для більшої кількості категорій символів;

3. *Набув подальшого розвитку* метод оцінювання відстані редагування між двома словами в алгоритмі Дамерау-Левенштейна з використанням таблиці подібності, що підвищило точність для слів з різних мов;

4. *Набув подальшого розвитку* метод оцінювання відповідності текстових даних до пошукової фрази на основі метрики, яка одночасно враховує кількість знайдених/незнайдених слів та символів, що покращує релевантність отриманих резулутатів для довгих пошукових фраз.

Практичне значення отриманих результатів. Практичне значення результатів, отриманих у ході дисертаційного дослідження, зводиться до наступного переліку:

1. Реалізовано систему нечіткого пошуку на основі розробленого методу, яка дозволяє знаходити текстові документи швидше та підвищує релевантність отриманих результатів, шляхом їхнього сортування відповідно до значень запропонованої метрики оцінювання релевантності текстових даних до пошукової фрази;

2. Розроблено методику реалізації таблиці подібності символів, за допомогою якої можлива розробка подібної таблиці із символів будь-якої мови для використання в суміжних областях обробки текстових даних;
3. Впроваджено систему нечіткого пошуку в середовище для резервного копіювання файлів для пошуку найбільш релевантних електронних листів, документів або текстів.

Особистий внесок здобувача. Дисертація є результатом самостійних наукових досліджень, в яких вкладено авторський підхід у розробку методу та програмних засобів для нечіткого пошуку найрелевантніших документів до пошукової фрази. Наукові положення та основні результати, які містяться в дисертації, отримані здобувачем самостійно у процесі науково-дослідницької роботи.

Апробація матеріалів дисертації. Основні положення та отримані наукові результати, що викладені в даній дисертаційній роботі, пройшли апробацію в наступних публікаціях.

Публікації.

Статті у наукових фахових виданнях України

[1] Kleshch, K., & Shablii, V. (2023). Comparison of fuzzy search algorithms based on Damerau-Levenshtein automata on large data. Technology Audit and Production Reserves, 4(2(72), 27–32. https://doi.org/10.15587/2706-5448.2023.286382.

[2] Клещ, К. О., & Царьов, М. О. (2023). МОДИФІКАЦІЯ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ ДЛЯ ВИКОРИСТАННЯ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ. Таврійський науковий вісник. Серія: Технічні науки, (3), 21-28. https://doi.org/10.32782/tnv-tech.2023.3.3.

[3] Kleshch, K. (2024). Development of fuzzy search method for creating an efficient information search system in text data. Technology Audit and Production Reserves, 1 (2 (75)), 20–24. doi: https://doi.org/10.15587/2706-5448.2024.298425.

Структура та обсяг дисертації. Дисертація складається із анотації, вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг дисертації становить 137 сторінок, у тому числі: 123 сторінки основного тексту, 50 рисунків, 4 таблиці, список використаних джерел із 126 найменувань на 13 сторінках.

РОЗДІЛ 1. АНАЛІЗ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ

Інформаційний пошук – процес пошуку, отримання та аналізу інформації для вирішення конкретної проблеми чи розв’язку задачі. Це може бути пошук інформації в інтернеті, базі даних, бібліотеці або будь-якому іншому джерелі з метою здобуття знань, вирішення проблеми чи прийняття рішення [11]. Інформаційний пошук може включати такі етапи, як формулювання запиту, вибір джерел інформації, аналіз та оцінка знайденої інформації.

Інформаційний пошук допомагає вирішувати різноманітні задачі, пов'язані з пошуком, аналізом та використанням інформації. До основних задач інформаційного пошуку відносяться:

- Пошук інформації, знаходження потрібної інформації серед великої кількості даних або джерел [12].
- Оцінювання інформації, аналіз та оцінювання знайденої інформації на предмет її достовірності, актуальності та значущості.
- Фільтрація інформації, відбір та відсіювання невідповідної або неактуальної інформації [13].
- Організація інформації, структурування та каталогізація інформації для зручного доступу та подальшого використання.
- Пошук патернів і зв'язків, виявлення закономірностей, зв'язків та патернів у великих обсягах даних.
- Пошук відомостей для прийняття рішень, збір та аналіз інформації для підтримки прийняття рішень в різних сферах діяльності.
- Моніторинг та аналіз трендів, виявлення та аналіз трендів, нових напрямків розвитку на основі зібраної інформації [4].
- Розробка стратегій комунікації, вибір та використання найефективніших засобів комунікації на основі дослідження ринку та аудиторії.

Основна ідея нечіткого пошуку полягає в тому, щоб знаходити рядки, які наближено відповідають заданому пошуковому слову, яке також називається рядком-шаблоном. На відміну від чіткого пошуку, де потрібно знайти точний збіг з шаблоном, нечіткий пошук дозволяє допускати деяку кількість помилок або неточностей між заданим і знайденим пошуковим словом. При цьому кожному співпадінню надається числова характеристика – порядок схожості знайденого рядка до шаблону. Цей вид пошуку широко використовується в пошукових системах, оскільки дозволяє отримувати близькі результати, навіть без точних збігів у запиті. Таким чином, нечіткий пошук підвищує користувацький досвід та надає більш точні та коректні результати пошуку [3].

Також, нечіткий пошук є важливим у сфері комп'ютерного зору. Системи оптичного розпізнавання символів часто стикаються із такими проблемами, як: шум, артефакти та різні шрифти, що можуть призводити до помилок у розпізнаванні тексту. Використання методів нечіткого пошуку допомагає покращити якість розпізнавання слів і виправити помилки. Алгоритми нечіткого пошуку не є новими та застосовуються вже доволі давно. Проте останнім часом суттєво збільшився об’єм інформації який необхідно та можна зберігати в різних сховищах даних. Тому важливо зосередитися на швидкості виконання та оптимізації алгоритму, навіть якщо це призведе до деякого зниження точності [14]. Одним із ключових понять у темі нечіткого пошуку є редагувальна відстань, яка використовується для визначення схожості двох рядків. Редагувальна відстань визначається як мінімальна кількість операцій перетворення над одним рядком, щоб він став ідентичним іншому.

Широко використовуваним типом редагувальної відстані є відстань Левенштейна. Вона визначається як мінімальна кількість операцій (вставки, видалення та заміни символів), необхідних для перетворення одного рядка на інший. Для більш точного врахування помилок користувача при введенні тексту використовується модифікована відстань Левенштейна, відома як відстань Дамерау-

Левенштейна [15]. Ця відстань враховує додаткову операцію – транспозицію, яка означає перестановку місцями двох символів. Один із найпоширеніших методів для обчислення відстані Левенштейна – це алгоритм, що був розроблений Робертом Вагнером та Міхаелем Фішером [4]. Основна ідея полягає в застосуванні динамічного програмування для обчислення редагувальної відстані.

Крім цього, існують альтернативні підходи до методів на основі динамічного програмування, наприклад, використання скінченних автоматів. Ці автомати приймають всі рядки, що відрізняються від заданого шаблону не більше, ніж на певну відстань [16]. В роботі розглянуті найпопулярніші алгоритми нечіткого пошуку, які використовують різний принципи роботи, та вибрано оптимальний з них для пошуку релевантних документів до пошукової фрази.

1.1 Класичні алгоритми пошуку редагувальної відстані

Обсяг електронної інформації в світі зростає кожного року приблизно на 30 %, відповідно зростання кількості доступних текстових даних ускладнює процес управління та пошуку інформації в цих даних [1]. Оскільки доступ до необхідної інформації має величезне значення, навички ефективного пошуку в обширних об'ємах даних стають дуже важливими. При роботі із текстовими даними, такими як: документи, веб-сторінки, електронні листи, презентації, електронні повідомлення, таблиці, тощо, критично важливо мати можливість швидко та точно знаходити потрібну інформацію та потрібні документи з переліку.

Однак традиційні алгоритми чіткого пошуку часто виявляються неефективними, особливо на великих масивах даних, або коли пошукова фраза або текстові дані містять такі неточності, як орфографічні чи друкарські помилки. У таких випадках класичні алгоритми, які покладаються на точні співпадіння, можуть не відповідати потребам користувачів [17]. Через це алгоритми нечіткого пошуку в тексті мають таку цінність. Дослідження алгоритмів нечіткого пошуку в тексті є важливим напрямком у сфері інформаційного пошуку та обробки тексту [18]. Технологія нечіткого пошуку

використовується для пошуку в інформаційних та текстових базах, забезпечуючи збіги для шаблону, навіть за наявності помилок або невизначеностей серед даних [19].

Існує багато алгоритмів пошуку, наприклад, такі як:

- алгоритм Дамерау-Левенштейна;
- алгоритм N-грам;
- алгоритм Джаро-Вінклера;
- алгоритм Bitap;
- звичайний алгоритм Левенштейна;
- алгоритм SoundEx;
- алгоритми на основі скінченних автоматів.

Всі ці алгоритми можуть бути корисні за умови пошуку приблизних відповідностей для заданого запиту в тексті, навіть якщо точної відповіді не існує. Розглянемо, деякі з найпопулярніших алгоритмів нечіткого пошуку, а також їхні переваги і недоліки.

1.1.1 Алгоритм Левенштейна

Алгоритм Левенштейна, також відомий як відстань редагування Левенштейна, показує кількість елементарних операцій зміни двома словами. Ця величина також використовується як параметр для перевірки подібності двох слів. Відстань Левенштейна між двома словами – найменша кількість односимвольних змін: вставок, видалень або замін, необхідних для перетворення одного слова на інше. Вона отримала назву на честь Володимира Левенштейна, радянського математика, який вивчав цю відстань у 1965 році [20]. Динамічне програмування передбачає першочергове розв’язання подібних задач меншого розміру, щоб потім застосувати цей метод до конкретної задачі більшої розмірності. Тому в цьому алгоритмі задача була розділена на 2 кроки, щоб з'ясувати відмінності та знайти спосіб розв'язання відповідно до цієї схеми [3].

Процес Левенштейна складається з таких частин: утворення матриці, перевірки літер у слові та присвоєння значення кожній комірці згідно з логікою алгоритму. На першому кроці потрібно створити матрицю, в якій слова будуть розміщені в рядках і стовпцях, незалежно від того, чи співпадає кількість літер. Спочатку розміщуються початкові значення у словах відповідно до їхнього порядку від початку до кінця. Після кожної літери значення збільшується на одиницю, за такої умови в обох словах клітинки поряд з ними заповнюються значеннями у зростаючому порядку. Тому в матриці є додаткова клітинка, що має значення 0, оскільки у стовпці або рядку немає слів які не містять жодної літери [21]. Порівняння починається з початку двох слів і продовжується до кінця матриці, тобто до останнього елемента на діагоналі. На початку вибирається початкова комірка із нульовим значенням у лівому верхньому куті, і для цієї комірки не передбачено літери. Після переходу до наступної комірки потрібно одна за одною застосувати деякі переходи відповідно до алгоритму.

Для кожної комірки матриці порівнюємо літери у початкових словах: якщо порівнювані літери рівні, то потрібно присвоїти поточному значенню комірки значення попередньої діагональної комірки. В іншому випадку, якщо порівнювані літери не співпадають, то значення збільшується на 1 у трьох комірках навколо неї: зліва, зверху і по діагоналі зверху-ліворуч [22]. Після збільшення вибирається найменше значення серед цих результатів як нове значення поточної комірки. Цей алгоритм застосовуються до всіх порожніх комірок поступово, від початкових до кінцевих, за формулою (1.1).

$$(i,j) = min \begin{cases} 0 \quad if\ i = j = 0, \\ d_{a,b}(i-1,j) + 1 \quad if\ i > 0, \\ d_{a,b}(i,j-1) + 1 \quad if\ j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \quad if\ i,j > 0, \end{cases} \tag{1.1}$$

Кожний перехід у тексті відповідає вставці або видаленню, які визначаються на другому кроці. Вартість кожної операції, як правило, дорівнює одиниці. Діагональний перехід коштує або одиницю, або нуль, в залежності від того, чи співпадають два

символи у рядку і стовпці. Другий крок – визначити, які саме операції потрібно виконати, щоб зробити обидва рядки однаковими. Для цього процесу потрібно завершити матрицю Левенштейна на основі вищезазначеної процедури. У цій матриці потрібно зосередитися на останньому елементі, який знаходиться у нижньому правому куті всієї сітки [23]. Наприклад, якщо довжина слів становить n та m, то цю першу вибрану комірку можна назвати n-тою та m-тою клітиною в матриці. Алгоритм Левенштейна це вдалий вибір, коли необхідно знайти редагувальну відстань між двома словами, проте він є доволі повільним, коли таких пар для порівняння є дуже багато.

1.1.2 Алгоритм Дамерау-Левенштейна

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

Відстань Дамерау-Левенштейна – міра подібності двох послідовностей символів або рядків. Загалом, дана відстань між двома рядками обчислюється як найменша кількість операцій вставки, видалення, заміни і транспозиції (перестановки двох сусідніх символів), необхідних для перетворення вхідної послідовності у вихідну [1].

Відстань Дамерау-Левенштейна є модифікацією класичної відстані Левенштейна, до якої, на додачу до трьох стандартних операцій, Дамерау додав операцію транспозиції [24]. Свою пропозицію він пояснив у статті, де зазначив, що під час дослідження орфографічних помилок для пошукової системи понад 80% були результатом помилки, яка належала саме до одного з чотирьох наведених типів, таких як: вставка, видалення, заміна і транспозиція.

Для розрахунку відстані Дамерау–Левенштейна найчастіше застосовують алгоритм на основі динамічного програмування, де використовується матриця

розміром (n + 1) * (m + 1), де n і m - довжини порівнюваних рядків. Окрім цього, вартість операцій видалення, заміни, вставки і транспозиції вважається однаковою і дорівнює одиниці, проте у процесі реалізації алгоритму це значення може бути змінене [25]. Щоб розрахувати дану метрику між двома рядками a і b, потрібно сконструювати матрицю, і заповнити кожну комірку матриці, використавши наступне рекурсивне рівняння (1.2).

$$d_{a,b}(i,j) = min\begin{cases} 0 \quad if\ i = j = 0, \\ d_{a,b}(i-1,j) + 1 \quad if\ i > 0, \\ d_{a,b}(i,j-1) + 1 \quad if\ j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \quad if\ i,j > 0, \\ d_{a,b}(i-2,j-2) + 1_{(a_i \neq b_j)}\ if\ i,j > 1\ and\ a_i = b_{j-1}\ and\ a_{i-1} = b_j, \end{cases} \tag{1.2}$$

де $1_{(a_i \neq b_j)}$ – це індикаторна функція, яка дорівнює нулю, коли $a_i = b_j$, і дорівнює одиниці у протилежному випадку.

Кожен рекурсивний виклик відповідає одному з випадків, відповідно до відстані Дамерау–Левенштейна:

- $d_{a,b}(i-1,j) + 1$ це видалення символа (від a в b);
- $d_{a,b}(i,j-1) + 1$ це вставка символа (від a в b);
- $d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)}$ це точне порівняння двох символів, які можуть збігатись або бути різними;
- $d_{a,b}(i-2,j-2) + 1_{(a_i \neq b_j)}$ це транспозиція між двома послідовними символами.

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

Якщо користувач задає певне порогове значення для метрики Дамерау–Левенштейна, то порівнюючи всі слова із вхідного тексту із заданим шаблоном, можна отримати результат нечіткого пошуку, який буде виключати всі слова, у яких відстань Дамерау–Левенштейна буде більшою за порогову.

Один із недоліків використання лише відстані Дамерау-Левенштейна для пошуку тексту полягає в тому, що цей алгоритм не враховує семантику або контекст

інформації [26]. Це може призводити до отримання неточних результатів, оскільки відстань Левенштейна вимірює лише кількість операцій (вставок, видалень, замін) для перетворення одного рядка на інший і не враховує сенсу слів або зв'язку між ними.

1.1.3 Алгоритм Джаро-Вінклера

Даний алгоритм є доволі схожим на алгоритм Дамерау–Левенштейна, проте для порівняння послідовностей тексту використовується інша метрика, а саме відстань Джаро–Вінклера. Відстань Джаро–Вінклера — рядкова метрика, яка показує відстань редагування поміж двома послідовностями символів. Значення відстані варіюється від 0 до 1, де нуль означає точну відповідність, а одиниця – відсутність подібності [27-29]. Дана метрика фактично визначена у термінах подібності Джаро-Вінклера, тому відстань набуває інверсивного значення, тобто відстань дорівнює одиниці мінус подібність.

Дана метрика була винайдена Вільямом Е. Вінклером у 1990 році як розширення подібності Джаро. Подібність Джаро-Вінклера враховує як рівність відповідних символів або літер, так і порядок цих символів у рядках [30]. В першу чергу, для визначення подібності Джаро–Вінклера необхідно розрахувати подібність Джаро, яка визначається за формулою (1.3)

$$sim_j = \begin{cases} 0, & if\ m = 0, \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right), & if\ m \neq 0, \end{cases} \quad (1.3)$$

де:

- m - кількість співпадінь символів, два символи від s_1 і s_2 вважаються точним співпадінням лише в тому випадку, якщо вони мають однакове значення відповідно до UNICODE кодування і розташовані не далі, ніж $\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor - 1$ один від одного;
- $|s_1|, |s_2|$ - довжини першого та другого рядка відповідно;

– t - це кількість транспозицій: Обчислюється як кількість відповідних символів, але у зворотному порядку послідовності і поділена на 2 [31].

Окрім подібності Джаро, для розрахунку подібності Джаро–Вінклера, використовується також оцінка префікса p, завдяки чому оцінки рядків є більш сприятливими, адже вони із самого початку відповідають заданій довжині префікса l. Якщо дано два рядки s_1 і s_2, то подібність Джаро–Вінклера sim_w розраховується за формулою (1.4):

$$sim_w = sim_j + lp\left(1 - sim_j\right) \quad (1.4)$$

де:

– sim_j – подібність Джаро для рядків s_1 і s_2;

– l – довжина загального префіксу на початку рядка, але не більше 4 символів;

– p є усталеним коефіцієнтом масштабування того, наскільки оцінка коригується за наявність загальних префіксів. p не повинен перевищувати 0,25 (тобто 1/4, причому 4 – це максимально можлива довжина префікса, що розглядається), інакше подібність може стати більшою за одиницю. Стандартним значенням цієї константи у роботі Вінклера є p = 0.1 [32].

Відстань Джаро-Вінклера є доволі ефективною для порівняння коротких і середніх за довжиною рядків, таких як імена або адреси, і вона зазвичай використовується в задачах пов’язування записів і дедуплікації даних. Проте даний алгоритм є зовсім неефективним для порівняння довгих рядків, оскільки за умови збільшення довжин вхідного пошукового слова, його продуктивність значно погіршується.

1.1.4 Алгоритм N-грам

В усіх алгоритмах нечіткого пошуку, які розглядались в попередніх розділах, була розглянута відмінність рядків, де порівнювався кожен символ окремо. Один зі способів розширити це поняття – дозвіл порівнювати одночасно декілька символів або, так звані, N-грами [33].

Відповідно, алгоритм N-грам – алгоритм зіставлення рядків, який потрібен для порівняння двох рядків і визначення міри їх подібності. Для цього потрібно розділити кожен рядок на підрядки довжиною N, де N – додатне ціле число, а потім порівнювати частоти входжень N-грам в обидва текстові рядки та обчислювати показник подібності на основі схожості або перекриття цих частот. Одиницею порівняння n-грам може бути послідовність символів довільної довжини, від одиниці, коли порівнюються набори символів, з яких складається слово, до довжини найкоротшого слова в тексті, що буде вважитись точним порівнянням слів. Чим більша довжина n-грами, тим більш жорсткі умови відбору подібних слів, і тим менша неточність такого методу [34-36].

У цього методу є один суттєвий недолік: одна помилка всередині слова спотворює декілька N-грам, що негативно впливає на результат порівняння [4]. Метод N-грам являється єдиним із розглянутих алгоритмів, який може визначити подібними пару слів, де одне складається із відповідних наборів символів, а інше – теж із них, але в іншому порядку. Прикладом таких пар можуть бути слова «графолог» і «логограф» чи наприклад, «логотип» і «типологія». Ця властивість даного методу вважається вагомим недоліком у порівнянні з іншими алгоритмами [2].

Загальна схема алгоритму подібності N-грам:

1. Передобробка даних:
   - Розбивка тексту на окремі слова або символи, залежно від необхідного рівня деталізації.
   - Вилучення стоп-слів, наприклад, таких поширених слів, як: "the", "is" і т. д., та проведення необхідної нормалізації та очищення тексту.
2. Генерація n-грам:
   - Створення всіх можливих N-грам, тобто послідовностей з n слів або символів для кожного тексту.
3. Створення індексів:

- Індекс n-грам створюється для зберігання поточних n-грам та їхніх відповідних позицій у вихідних рядках. Цей індекс дозволяє ефективно порівнювати та шукати n-грами під час пошуку відповідностей [37].

4. Обчислення частот:
   - Підрахунок кількості входжень кожної унікальної n-грами з рядка запиту в кожному вихідному рядку. Ці частоти необхідно зберігати у структурах даних, таких як словники або частотні вектори [38].
5. Обчислення подібності:
   - Для підрахунку показника подібності між двома текстами на основі частот n-грам, можна використовувати різні варіації показників, такі як: подібність Жаккара, коефіцієнт Дайса або косинусна подібність. Ці показники подібності зазвичай включають порівняння перетину та об'єднання множин n-грамів, а також використання векторних обчислень.
6. Формування результату:
   - Отриману оцінку подібності потрібно використати для ранжування або для порівняння подібності декількох текстів.
   - Вищий показник подібності вказує на більший відсоток перекриття частот n-грам, а отже, тексти більш схожі.

Алгоритм N-грам є швидким та ефективним методом для порівняння подібності текстів серед великих обсягів текстових даних [39]. Даний алгоритм може використовуватися для визначення плагіату та гарно поєднується з іншими алгоритмами зіставлення, наприклад, його можна використати для зіставлення записів даних і дедуплікації.

1.1.5 Алгоритм Bitap

Алгоритм Bitap, також відомий як алгоритм зсуву чи алгоритм Бези-Йетса-Гоннета – алгоритм для наближеного порівняння і зіставлення рядків. Даний алгоритм перевіряє, чи містить заданий вхідний текст підрядок, який приблизно дорівнює

заданому шаблону, де орієнтовна рівність визначається як відстань Левенштейна: якщо шаблон і підрядок знаходяться на заданій відстані k один від одного, тоді алгоритм вважає їх достатньо подібними [5].

Вперше ідею даного алгоритму запропонували Рікардо Беза-Йетс і Гастон Гоннет у відповідній статті у 1992 році. Проте дана версія алгоритму перевіряла лише заміну символів і, по суті, обчислювала лише відстань Хемінга [40]. Але пізніше Сунь Ву та Уді Манбер запропонували та впровадили модифікацію даного алгоритму для розрахунку відстані Левенштейна, тобто додали підтримку операцій вставки і видалення у запропонований раніше алгоритм. Суть алгоритму полягає у побудові шаблону, який потім буде порівняний з більшою частиною тексту для того, щоб знайти приблизні збіги. Основна ідея алгоритму полягає у використанні побітових операцій зсуву для швидкого порівняння шаблону із текстом [41]. Нехай S0 дорівнює “ababc” є шаблоном або пошуковим словом, який потрібно порівняти із рядком S1 = “abdababababc”, тоді загальна схема алгоритму Bitap буде мати наступний вигляд [6]:

1. Попередня обробка на початку алгоритму:
   - Визначити всі унікальні символи у відповідних рядках. Для S0 і S1 це будуть символи ‘a’, ‘b’, ‘c’, ‘d’.
   - Далі необхідно побудувати бітову маску для кожного символу, для цього потрібно перевернути шаблон S0 та для бітової маски певного символу – розмістити нуль там, де він присутній у шаблоні, в іншому випадку потрібно розмістити одиницю. Таким чином, для символу ‘a’, оскільки він зустрічається в 3-ій та 5-ій позиції шаблону, то на зазначених позиціях потрібно поставити 0, а в інших позиціях – 1. Приклад такої бітової маски зображено на рис. 1.1.

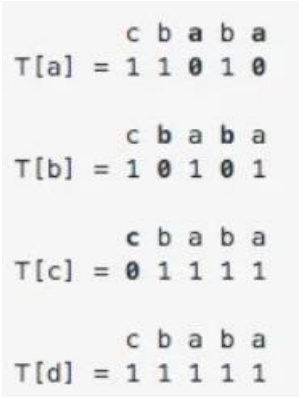

Рисунок 1.1 – Приклад бітових масок для символів ‘a’, ‘b’, ‘c’, ‘d’

2. Отримати початковий стан бітового шаблону довжини |S0|, де кожен біт якого дорівнює одиниці. В запропонованому прикладі це буде S=11111.
3. Проітерувати за кожним символом рядка S1, де необхідно знайти патерн, та виконати наступні дії:
   - Зробити зсув нуля в найменший біт шаблону S. У прикладі це буде 5-ий біт справа.
   - Провести побітову операцію логічного “або”: S = S | T[x].

Наприклад, для першого символу S1, ‘a’:

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

Для всього вхідного рядка S1, результат алгоритму bitap зображений на рис. 1.2.

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 1.2 – Результат алгоритму bitap для шаблону “ababc” та пошукового рядку “abdababababc”

Для того, аби виявити повний або частковий збіг шаблону із підрядком вхідного тексту, необхідно, щоб під час ітерацій за символами тексту нуль досяг найвищого біта у бітовому шаблоні стану S [42]. У такому разі результуючий підрядок буде

закінчуватися на символі, за яким шаблон досяг стану з нулем під час ітерації. У прикладі вище це останній символ ‘c’ [6]. Для виявлення нечіткого збігу необхідно стежити за нульовим бітом у різних позиціях стану S. Наприклад, якщо нуль знаходиться на 4 позиції, то було знайдено 4 з 5 символів у тій же послідовності, що й у S0. Так само для нуля і в усіх інших місцях [6].

Перевагою цього алгоритму у порівнянні з іншими методами пошуку є використання побітових операцій, що робить його дуже швидким. Додатковим плюсом алгоритму є те, що він легкий в реалізації, проте доволі складний для розуміння.

1.1.6 Алгоритм SoundEx

Soundex – представник фонетичних методів, які реалізують пошук подібних слів на основі їхнього звучання. Даний алгоритм перетворює слова у стандартизований код на основі їхньої вимови, а в процесі пошуку порівнюються саме ці фонетичні коди. Даний алгоритм розроблено на початку 20-го століття, і з тих пір він набув широкої популярності у різних сферах [43]. Наприклад, він використовується в генеалогії для дослідження сімейної історії, а саме: допомагає знайти альтернативні або дуже схожі варіанти прізвищ. Це може бути корисним у випадках, коли з часом варіанти написання прізвищ змінюються, проте звучання залишається таким же [44].

Алгоритм складається з наступних кроків:

1. Спочатку необхідно видалити всі початкові і кінцеві пропуски та перетворити літери слова в один регістр: маленькі чи великі.
2. Після цього потрібно зберегти першу літеру слова і відкинути всі наступні входження таких літер англійського алфавіту: 'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Надати числовий код кожній літері, що залишилася, виходячи з її звучання:
   - Літерам 'B', 'F', 'P', та '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'єднати першу літеру слова із числовими кодами, які були отримані у попередньому кроці [45].
5. Видалити послідовні входження одного й того ж самого номера, за виключенням першого входження.
6. Якщо отриманий код буде мати менше чотирьох символів, необхідно додати до нього зліва нулі. Якщо він буде мати більше чотирьох символів, потрібно обрізати його й обмежити чотирма.
7. Остаточний код Soundex це отримана чотирисимвольна послідовність [44-46].

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

1.2 Алгоритми нечіткого пошуку на основі скінченних автоматів

У попередньому пункті були розглянуті класичні алгоритми нечіткого пошуку, які гарно підходять для розв’язання задачі пошуку редагувальної відстані між двома словами. Проте для розв’язання задачі пошуку шаблону в тексті ситуація трохи інша, через те, що пошукове слово буде одним і тим самим для всіх слів із текстового набору. Найпростіший підхід для пошуку підходящих слів у тексті це використання алгоритму для розрахунку відстані Дамерау-Левенштейна напряму. Для кожного слова із текстових даних потрібно порахувати відстань редагування до заданого користувачем рядку та повернути всі слова, що мають найменшу редагувальну відстань. Проте такий підхід не є оптимальним, оскільки підходящими є результати із редагувальною відстанню до якогось числа помилок, а за найпростішого підходу кожен раз обчислюється повна відстань. Отже, поточна задача – знайти способи, що

дозволяють порівнювати рядки між собою швидше, ніж класичний підхід із розрахунком відстані Дамерау-Левенштейна [47].

Перспективними є підходи на основі детермінованих скінченних автоматів, що приймають всі слова до якоїсь редагувальної відстані відносно заданого патерна. Крім цього, вони дозволяють здійснювати ефективний пошук не тільки для онлайн варіанту, а й для офлайн пошуку [11]. Для поточної задачі ідеально підходять алгоритми на основі скінченних автоматів, адже такі алгоритми будують всі можливі патерни слів, які відрізняються від пошукового слова не більше, ніж на задану редагувальну відстань. Приклад роботи такого автомата зображено на рис. 1.3, де як шаблон обрано пошукове слово “algorithm”, максимальна редагувальна відстань якого становить 2, а ‘?’ – довільний символ.

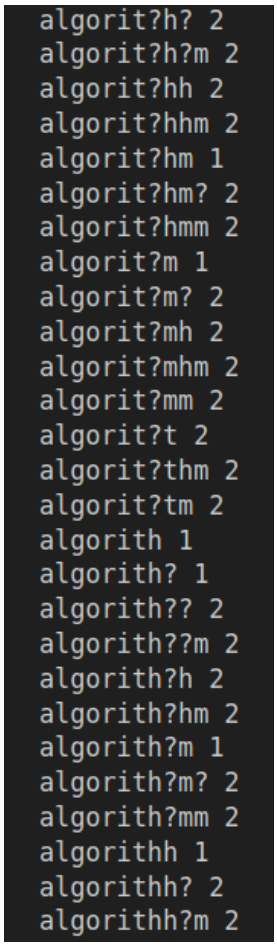

Рисунок 1.3 – Приклад роботи автомата

1.2.1 Скінченні автомати

Скінченним автоматом M зазвичай називають кортеж, який складається з 5 елементів: $\langle Q, q_0, A, \Sigma, \delta \rangle$, де

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

Скінченний автомат називається детермінованим або ДСА, якщо δ – функція з $Q \times \Sigma$ в Q, інакше він має назву недетермінований або НСА [48]. Скінченний автомат розпочинає свою роботу у стані q_0 і почергово зчитує символи вхідного рядка один за одним. Якщо автомат зараз знаходиться у стані q та зчитує вхідний символ ‘a’, то він робить перехід зі стану q у стан $\delta(q, a)$. Якщо поточний стан є елементом A, то автомат приймає зчитаний рядок на поточному символі. Рядок, який був не прийнятий автоматом, називається відхиленим, інакше рядок визначається прийнятим [49].

Скінченний автомат – математична модель обчислювального процесу, який може знаходитися в одному з кінцевих чисел станів. Автомат починає роботу в певному початковому стані, а потім, приймаючи послідовність вхідних символів із вхідного рядка, переходить зі стану в стан згідно з правилами переходу. У кожному стані автомат може виробляти одну або кілька дій: видаляти, змінювати або додавати символи до вхідної послідовності. Таким чином, автомат може розпізнавати або приймати рядки, які задовольняють певні умови. Скінченні автомати використовуються в багатьох областях, включаючи теорію мов програмування, компіляцію, шаблонний пошук, обробку природної мови та інші [50]. Вони є основним інструментом у вивченні формальних мов та автоматів. Детермінований скінченний автомат має один і тільки один перехід для кожного стану та вхідного символу, що робить його визначеним та передбачуваним. Недетермінований

скінченний автомат може мати кілька переходів для одного стану та вхідного символу або може мати переходи без вхідних символів, що робить його більш гнучким та потужним, але водночас складнішим у реалізації [51, 52]. Скінченний автомат може бути зображений графічно у вигляді таблиці переходів або діаграми станів. Діаграма станів дає інтуїтивне представлення станів, переходів між станами та положення кінцевих станів, тоді як таблиця переходів містить детальну інформацію про кожен стан та можливі переходи між ними. На рис 1.4 зображено приклад автомата, який приймає рядки, що завершуються непарною кількістю символів “a” у вигляді діаграми станів. Стани автомата позначені колами, а початковий стан позначений як 0. Подвійні кола відповідають за кінцеві стани автомата, а переходи між станами показані за допомогою стрілок [49].

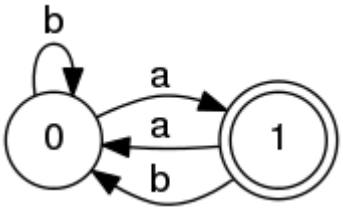

Рисунок 1.4 – Зображення автомата у вигляді діаграми станів

В таблиці 1.1 зображено той самий автомат, але за допомогою таблиці переходів.

Таблиця 1.1 – Приклад автомата у вигляді таблиці переходів

Стан	Символ	
	a	b
0	1	0
1	0	0

Згідно з визначенням, автоматом Левенштейна для рядка w та максимальної відстані редагування d називається автомат, який приймає усі можливі рядки для відстані Левенштейна до w не більше, ніж d. Скінченний автомат Левенштейна (Levenshtein automaton) – автомат, який використовується для пошуку всіх можливих

рядків, які мають відстань редагування не більше заданого числа d від заданого рядка w. Він дозволяє знаходити всі можливі редагувальні відстані між двома рядками без фактичного обчислення цих відстаней, що робить його ефективним для задач, де потрібно шукати рядки, близькі до заданого шаблону, або таких задач як пошук орфографічних помилок або апроксимація тексту.

Такий автомат побудований на основі детермінованого скінченного автомата, де стани відповідають певним позиціям у вхідному рядку, а переходи відбуваються на основі символів вхідного рядка та можливих редагувальних операцій [53]. Під час побудови автомата для кожного стану створюються переходи в інші стани, які відповідають можливим операціям над вхідним рядком. Для знаходження всіх можливих рядків із відстанню редагування не більше d, автомат рекурсивно просувається по вхідному рядку, дозволяючи вставку, видалення та заміну символів у межах відстані d. Таким чином, автомат знаходить всі можливі шляхи редагування, які ведуть до рядків із відстанню редагування не більше d від заданого рядка w.

1.2.2 Детермінізація скінченних автоматів

Основна відмінність між ДСА і НСА полягає в поведінці під час переходів. ДСА мають унікальний перехід між станами для кожного вхідного символу, тоді як НСА можуть мати декілька переходів або ε-перехід для заданого вхідного символу. ε-переходами називаються переходи, які зчитують пустий рядок. Тобто, переходячи до стану, що має ε-переходи, автомат опиняється одночасно у станах, на які вказують ці переходи [54]. Автомат, зображений на рис. 1.4, є детермінованим, адже для кожного стану та символу він має унікальний перехід, тому має тільки єдиний поточний стан. Перетворення з НСА в еквівалентний ДСА можливе за допомогою алгоритмів детермінізації.

Детермінізація скінченного автомата – процес перетворення недетермінованого скінченного автомата в еквівалентний йому детермінований скінченний автомат [55]. Цей процес дозволяє спростити автомат, зменшити його складність та зробити його

реалізацію більш ефективною. Основна ідея детермінізації полягає в тому, що для кожного стану і символу вхідного алфавіту в НСА створюється новий стан в ДСА, який представляє всі можливі переходи з цього стану за цим символом [56-59]. Це робить автомат детермінованим, оскільки для кожного стану і символу існує тільки один можливий перехід. Детермінізація може призвести до збільшення кількості станів у автоматі, особливо якщо в початковому НСА було багато недетермінованих переходів. Проте вона робить автомат більш прозорим та простим у реалізації. Детермінізація недетермінованого скінченного автомата може бути виконана за допомогою кількох алгоритмів.

Алгоритм підмножинної конструкції є одним з найпростіших способів детермінізації НСА. Для кожної підмножини станів НСА створюється стан в ДСА, який представляє цю підмножину. Початковий стан ДСА відповідає початковому стану НСА, а кінцеві стани ДСА відповідають підмножині станів НСА, які містять принаймні один стан закінчення [60].

Алгоритм перетину станів працює за принципом перебору всіх можливих комбінацій станів НСА для кожного вхідного символу, що призводить до створення нового стану ДСА. Якщо новий стан ДСА є переходом з одного стану НСА в інший, то він вважається прийнятим станом ДСА [61].

Алгоритм підтримки імплікантів використовує поняття імплікантів для пошуку еквівалентного ДСА. Він аналізує всі можливі комбінації станів НСА для кожного вхідного символу та вибирає найменшу множину станів, яка може бути використана для створення DFA.

Всі ці алгоритми дозволяють перетворити недетермінований скінченний автомат у детермінований, що спрощує його аналіз та реалізацію. Покроковий опис кожного із розглянутих алгоритмів:

- Підмножинна конструкція:

a. По-перше, створюється початковий стан ДСА, який відповідає початковому стану НСА [62];
b. Далі, для кожного стану ДСА і кожного можливого вхідного символу обчислюється множина станів НСА, в які можна потрапити з цього стану за допомогою даного символу;
c. Ці множини станів НСА стають новими станами ДСА;
d. Якщо стан ДСА містить принаймні один стан закінчення НСА, то він також є станом закінчення ДСА [63].

- Перетин станів:
  a. Для кожного стану ДСА і кожного можливого вхідного символу обчислюється множина станів НСА, в які можна потрапити з цього стану за допомогою даного символу [60];
  b. Ці множини станів НСА перетинаються, щоб отримати новий стан ДСА;
  c. Якщо отриманий стан не існує в ДСА, то він додається як новий стан;
  d. Процес повторюється для кожного нового стану ДСА, поки не будуть визначені всі можливі переходи.
- Підтримка імплікантів [64]:
  a. Аналізується кожний стан ДСА і кожен вхідний символ, щоб знайти множину станів НСА, в які можна потрапити з цього стану за допомогою поточного символу;
  b. Далі розглядаються усі можливі комбінації цих множин станів НСА для кожного вхідного символу;
  c. Вибирається найменша множина станів, яка покриває всі можливі переходи з ДСА;
  d. Ця множина станів стає новим станом ДСА, а процес повторюється для кожного нового стану ДСА.

Хоча ДСА і НСА відрізняються поведінкою переходів, вони еквівалентні між собою з точки зору розпізнавання мови. Будь-яка мова, що приймається ДСА, також буде прийнята ДСА і навпаки [65]. Недетерміновані скінченні автомати часто використовуються в теоретичних обговореннях і для побудови регулярних виразів, тоді як ДСА частіше використовуються у практичних реалізаціях завдяки їхній ефективності та визначеності.

1.2.3 Автомат на основі префіксного дерева

Скінченний автомат на основі префіксного дерева, також відомий як trie, є структурою даних, яка використовується для зберігання і пошуку множини рядків чи слів з певного алфавіту [66]. Trie може бути використаний для реалізації пошукових структур даних та алгоритмів, таких як: автозаповнення, пошук слова у словнику, перевірка правопису та багато іншого. Основна ідея цієї структури полягає в тому, щоб зберігати слова таким чином, щоб кожна літера слова була представлена як окремий вузол у дереві. Кожне слово розглядається як шлях від кореня до листка дерева. Якщо слово складається з більше ніж однієї літери, то шлях від кореня до листка буде містити послідовність вузлів, які відповідають кожній літері слова. Однією з важливих переваг автомату на основі префіксного дерева є те, що він дозволяє ефективно виконувати операції вставки, видалення та пошуку за час O(m), де m – довжина шуканого слова. Також він може забезпечити швидкий доступ до всіх слів, що починаються з певного префіксу, що робить його корисним для задач автозаповнення та пошуку [67]. Проте trie може мати деякі недоліки, зокрема, велика кількість пам'яті, що вимагається для зберігання словників із великою кількістю слів. Також, якщо алфавіт дуже великий, то trie може стати дуже глибоким і займати багато місця в пам'яті. Однак, ці недоліки можуть бути зменшені за допомогою методів стиснення, таких як компактне trie або бітові trie.

Реалізація даного алгоритму починається з побудови недетермінованого автомата у вигляді префіксного дерева. Щоб побудувати автомат, який би приймав

слова, що відрізняються від пошукового слова не більше, ніж на задану відстань, потрібно перебрати всі можливі варіанти операцій над шаблоном, для яких сумарна вартість є менше або рівною за максимально дозволену [68].

Покроковий опис алгоритму:

1. Поки черга не є порожньою, беремо перший елемент з черги і обробляємо його. Кожен елемент із черги буде відповідати комбінації поточного символу з пошукового слова та залишкової редагувальної відстані;
2. Якщо індекс поточного стану дорівнює довжині слова, потрібно відмітити поточний вузол як кінцевий стан;
3. За умови, що сумарна оцінка із вартістю вставки не перевищує максимальну допустиму відстань, необхідно створити новий вузол у дереві із символом вставки та додати новий стан до черги;
4. Створювати нові стани із різними діями: вставка, видалення, транспозиція та заміна символів. У випадку, коли сумарна поточна штрафна сума не перевищує максимально можливу відстань редагування, потрібно додати такий стан до черги та створити його у префіксному дереві.

На рис. 1.5 зображено приклад та результат побудови НСА для вхідного шаблону “ab” із максимальною редагувальною відстанню 1. За допомогою кола позначено стани, які є вузлами дерева, а подвійним колом зображено кінцеві стани автомата. Символом “?” позначено довільний символ, включаючи “a” та “b”, початковий стан має нульовий індекс [69, 70]. Цей автомат приймає будь-які слова, які мають редагувальну відстань до шаблону “ab” менше або рівну одиниці. Наприклад, для вхідного слова “ba” редагувальна відстань дорівнює 1, оскільки однієї транспозиції достатньо, для того щоб перетворити його на слово “ab”. Під час зчитування першого символу “b” у автомата є переходи в стани, які позначені індексами 10 та 1. Після цього, у процесі зчитування “a” можливі переходи до станів 11 та 2. Стан 11 вважається фінальним, тому слово приймається автоматом. Для

обчислення поточної відстані під час побудови автомата необхідно в кожному вузлі дерева зберігати величину, яка дорівнює сумі відстаней попередніх виконаних операцій. Їх необхідно виконати для того, щоб дійти до цього вузла. Ця величина і є редагувальною відстанню послідовності [71].

Оскільки прохід вздовж НСА є затратним процесом, наступним кроком є його детермінізація, що дозволить забезпечити швидку перевірку слів. Мета детермінізації полягає в тому, щоб кожен стан автомата мав лише один перехід для кожного символу [5]. Для досягнення цього, потрібно для кожного стану об’єднати універсальний перехід з іншими можливими його переходами. Завдяки цьому автомат буде уникати необхідності переходити в декілька станів одночасно. Після детермінізації результат ДСА зображено на рис. 1.6, де символ “?” відповідає за всі символи, окрім тих, для яких вже існує перехід із цього стану. Наприклад, для стану 0, символ “?” відповідає за всі символи, окрім символів “a” та “b” [72].

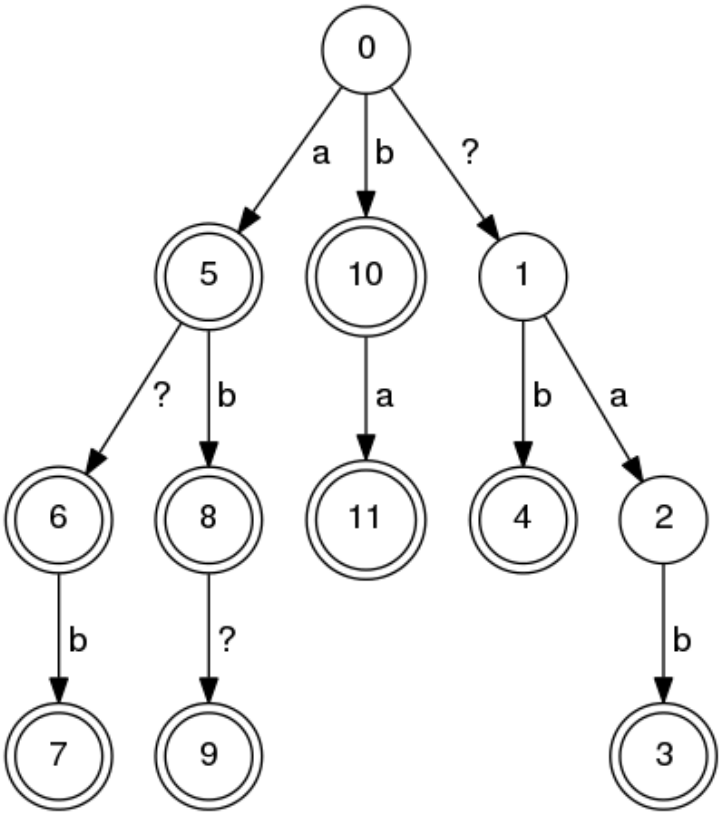

Рисунок 1.5 – НСА для шаблону “ab” та редагувальної відстані 1

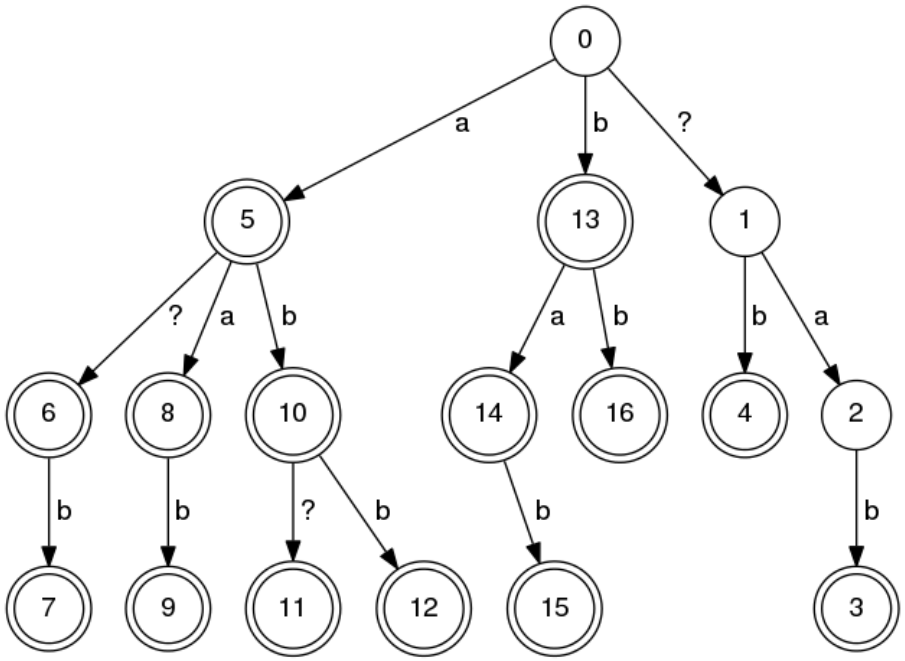

Рисунок 1.6 – ДСА для шаблону “ab” та редагувальної відстані 1

Перевірка слів автоматом є доволі простою. У циклі потрібно проітерувати кожен символ вхідного слова, паралельно оновлюючи поточний стан автомата. У разі потрапляння на неіснуючий перехід, ітерація завершується, оскільки вхідне слово має редагувальну відстань до шаблону більшу за максимально дозволену. У такому випадку автомат відхиляє слово, а в протилежному – приймає. Більше того, завдяки такий реалізації, автомат, окрім булевої відмітки про прийняття, також повертає і саму редагувальну відстань між шаблоном та вхідним словом.

Основним недоліком такого рішення є велика кількість станів для довгого пошукового слова і редагувальної відстані >= 3, відповідно буде мати місце довга побудова і великі затрати пам’яті.

1.2.4 Автомат на основі хешування

Автомат на основі хешування – структура даних, яка використовує хеш-таблицю для зберігання та швидкого пошуку ключів. У контексті скінченних автоматів, автомат на основі хешування може бути використаний для зберігання станів та переходів між станами. Основна ідея полягає в тому, що кожен стан автомата

може бути представлений як ключ у хеш-таблиці, де значення відповідає переходам із цього стану до інших станів [73]. Хеш-таблиця дозволяє швидко знаходити переходи для кожного стану без необхідності переглядати всю таблицю. Перевагою автомата на основі хешування є швидкість пошуку, особливо для великих автоматів із великою кількістю станів та переходів. Також, використання хешування дозволяє зменшити кількість пам'яті, потрібної для зберігання автомата, оскільки не треба зберігати всі можливі переходи в окремих структурах даних. Однак, недоліком цього підходу може бути втрата порядку переходів, оскільки хеш-таблиці не зберігають дані у впорядкованому вигляді. Це може стати проблемою у випадках, коли порядок переходів має значення, наприклад, у випадку автоматів, що моделюють послідовності дій [74].

Хешування станів скінченного автомата є методом зменшення кількості станів автомата за допомогою хеш-функцій. Цей процес дозволяє зменшити кількість пам'яті, необхідної для зберігання станів автомата, що особливо корисно для складних автоматів із великою кількістю станів. Основна ідея полягає в тому, щоб замінити кожен стан автомата за допомогою його хеша, тобто унікальним числовим значенням, що відповідає цьому стану [72]. Однак, за хешування станів важливо враховувати, що можуть виникати конфлікти, коли два різних стани автомата мають однаковий хеш, що в свою чергу може призвести до некоректної роботи автомата. Тому важливо вибрати ефективні хеш-функції, які мінімізують ймовірність колізій. Хешування станів скінченного автомата може бути корисним у випадках, коли автомат має велику кількість станів і обробляє складні задачі, такі як обробка природної мови або обробка даних у реальному часі.

У цьому рішенні немає прив’язки до структури префіксного дерева. Нехай ціна за кожну операцію, будь-то видалення, транспозиція, вставка або заміна є однаковою та дорівнює 1, що дозволяє побудувати більш структурований автомат, який пришвидшує побудову та детермінізацію НСА. Кожен стан автомата відповідає

певній конфігурації із поточної кількості оброблених символів у шаблоні та кількості застосованих у цьому випадку операцій редагування. Кожен перехід між станами відповідає конкретній операції. Стани, які повністю обробили заданий шаблон є фінальними [75]. На рис. 1.7 зображено HCA для шаблону “ab” та максимальної редагувальної відстані 1. Перше число у назві стану показує кількість оброблених символів шаблону, а друге число відповідає поточній редагувальній відстані. Якщо символ “*” позначає переходи, що приймають будь-який символ, а “ε” позначає нульові переходи, тоді початковий стан буде відповідати комбінації символів “0 0” [7]. Стан “0*0” відповідає операції транспозиції, що може статись із першими двома символами [76].

Програмна реалізація побудови такого HCA є доволі простою та зрозумілою. Алгоритм створює стан для кожного можливого типу помилки та кожної потенційної кількості помилок у відповідній позиції у шаблоні та додає переходи між ними. Для транспозиції додається новий проміжний допоміжний стан, що зчитує спочатку наступний символ, а після цього – поточний символ на даній ітерації. Для побудови ДСА потрібно виконати процедуру детермінізації.

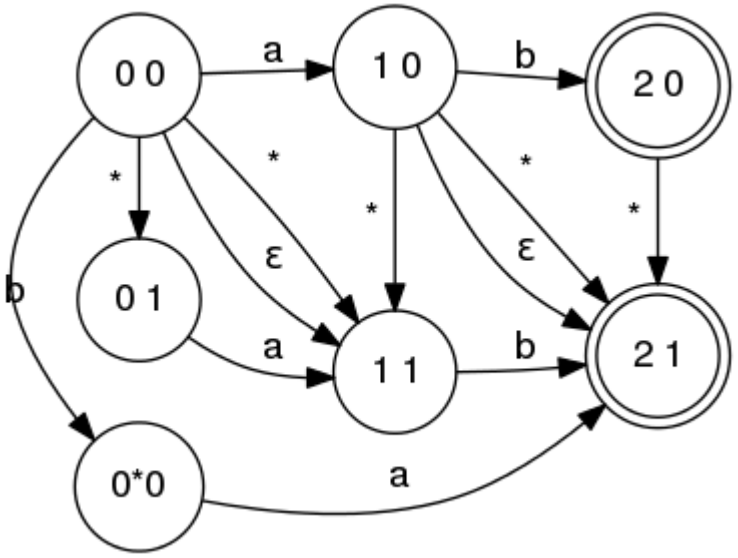

Рисунок 1.7 – HCA HashAutomaton для шаблону “ab”

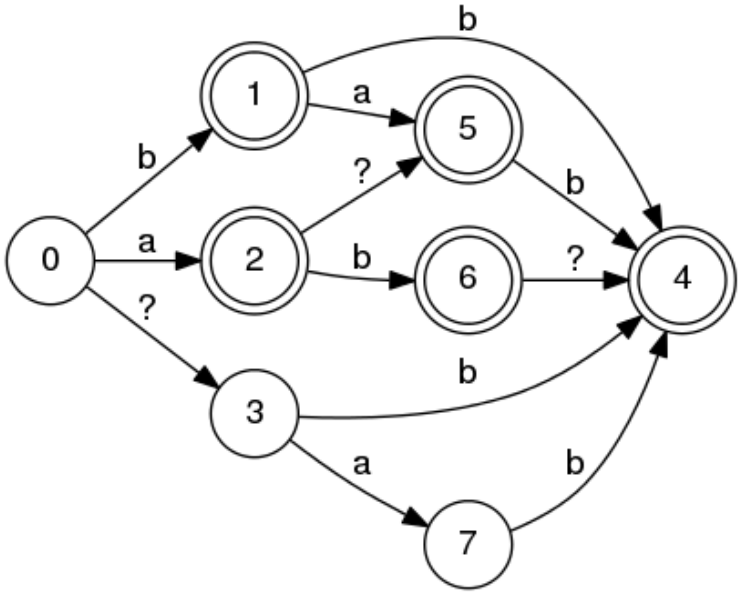

Рисунок 1.8 – ДСА HashAutomaton для шаблону “ab”

Далі необхідно побудувати детермінований скінченний автомат на основі НСА, який було сформовано на попередньому кроці. ДСА є значно зручнішим та ефективнішим у процесу пошуку та перевірки слів. Для побудови ДСА потрібно перейти за всіма доступними переходами НСА, створюючи нові стани в ДСА для кожної унікальної комбінації станів НСА. Після завершення процедури детермінізації НСА, автомат готовий до перевірки слів [77]. Перевірка слів для нього доволі схожа на перевірку в автоматі на основі префіксного дерева, що була розглянута до цього.

Запропонований автомат доволі схожий за принципом роботи на автомат Левенштейна, проте на виході, окрім булевого значення, повертає й редагувальну відстань між словами, а також підтримує операцію транспозиції символів.

1.2.5 Автомат на основі таблиці переходів

Автомат на основі таблиці переходів – тип скінченного автомата, де всі можливі переходи між станами задаються у вигляді таблиці. Ця таблиця містить всі можливі комбінації вхідних символів і станів, а також вказує, в який стан автомат повинен перейти за певних умов. До важливих аспектів автомата на основі таблиці переходів можна віднести його простоту та прозорість. У таблиці чітко вказані всі можливі

переходи, що полегшує розуміння логіки автомата та пришвидшує налагодження його роботи [78]. Проте автомат на основі таблиці переходів може стати неефективним у випадку великої кількості станів або символів вхідного алфавіту. Таблиця може стати дуже великою і вимагати багато пам'яті для зберігання, особливо у разі, коли є багато недосяжних станів або символів. Також варто мати на увазі, що таблиця переходів повинна бути досить динамічною, щоб враховувати такі зміни в автоматі, як додавання нових станів або переходів. Один з підходів до вирішення цієї проблеми – використання структур даних, що дозволяють проводити динамічну зміну розміру таблиці у вигляді хеш-таблиці або списків. Загалом, автомат на основі таблиці переходів є простим і зрозумілим способом представлення скінченного автомата, але він може бути неефективним у випадку великої складності автомата [79, 80].

Таблиця переходів у скінченному автоматі це структура даних, яка визначає, як автомат повинен змінювати свій поточний стан відповідно до вхідного символу. Вона містить інформацію про всі можливі переходи з одного стану в інший за умови зустрічі із різними символами вхідного алфавіту. Основні елементи таблиці переходів включають в себе:

1) Стани: кожен рядок таблиці відповідає одному стану автомата. Кожен стовпчик відповідає символу вхідного алфавіту;
2) Переходи: елементи таблиці, такі як стани та символи, визначають, в який стан можливо перейти у разі потрапляння на вхід відповідного символу у певному стані;
3) Початковий стан: один зі станів визначається як початковий, з якого починається обробка вхідної послідовності [81];
4) Приймаючий стан: деякі стани визначають як приймаючі, що характеризує кінцевий стан автомата. Якщо автомат досягає приймаючого стану після обробки вхідної послідовності, це означає, що вона задовольняє правила автомата.

Таблиця переходів може бути представлена у вигляді двовимірного масиву, де кожен елемент [i, j] відповідає переходу зі стану ‘i’ за входом ‘j’. Також її можна представити у вигляді словника, де ключ це стан, а значення – словник із символів вхідного алфавіту на нові стани. Таблиця переходів є ключовою частиною визначення скінченного автомата і визначає його поведінку відносно вхідних символів.

За основу взято приклад автомата, який розроблений у [8], в ролі модифікації наведеної реалізації можна додати операцію транспозиції unicode символів. HCA, що лежить в основі табличного автомату, нічим не відрізняється від HCA на основі хешування. Основна перевага TableAutomaton – відсутність необхідності явної детермінізації автомата. Маючи шаблон та максимальну відстань редагування, автомат одразу може перейти до перевірки слів. Для прикладу, результат детермінізації HCA в HashAutomaton для таких слів як “free” та “tree” буде ідентичним, але буде відрізнятись від ДСА для слова “pain” або “soon”. Якщо змінити слова, давши кожному унікальному символу свою цифру, враховуючи порядок, буде очевидним зв’язок між цими наборами слів. Нехай: free = 1233 = tree, pain = 1234, soon = 1223. Тобто набір унікальних станів HCA, що буде отримано з одного стану, залежить тільки від символу, який перевіряється на поточному кроці, та входжень цього символу в шаблон, починаючи зі зсуву, на якому цей стан знаходиться [82].

Для відображення цього факту вводиться поняття характеристичного вектору, який представляє собою бітову маску, де значення в позиції дорівнює одиниці тільки тоді, коли символ шаблону в цій позиції дорівнює символу в текстовому слові. Приклад для шаблону “free” та символу “e” представлено на рис. 1.9.

```
e
 f  r  e  e
<0, 0, 1, 1>
```

Рисунок 1.9 – Характеристичний вектор для шаблону “free” та символу “e”

Характеристичний вектор для шаблону – вектор, який містить інформацію про певний шаблон. У векторі можуть бути присутні різні параметри чи властивості, які характеризують шаблон і дозволяють відрізняти його від інших об'єктів або шаблонів. [83]. У контексті скінченних автоматів характеристичний вектор для шаблону може включати в себе інформацію про стани, вхідні символи та переходи, які характеризують шаблон та визначають його поведінку. Цей вектор може бути використаний для порівняння шаблону з іншими об'єктами або шаблонами, а також для визначення, чи відповідає даний об'єкт шаблону. Характеристичний вектор для шаблону дозволяє зручно представляти та обробляти інформацію про шаблон, яка може бути корисною для виконання різних завдань, пов'язаних із використанням шаблонів у скінченних автоматах, таких як: розпізнавання шаблонів, порівняння зразків та інші аналітичні або обчислювальні операції.

Для переходу між станами автомата важливі тільки характеристичні вектори довжини 2*d + 1, де d – максимально допустима редагувальна відстань. Враховуючи це, необхідно наперед розрахувати всі можливі переходи та всі потенційні стани для довільного шаблону на початку програми. Алгоритм схожий на детермінізацію НСА, але замість символів шаблону, для такої процедури необхідно використати характеристичні вектори та зберегти всі унікальні конфігурації станів, отримані в процесі, у таблицю переходів [84].

Варто зауважити, що такий підхід показує гарні результати тільки для невеликих значень максимальної редагувальної відстані, оскільки розмір таблиці переходів росте експоненційно з її значенням, через те, що існує 2^{2d+1} унікальних значень характеристичного вектора, де d – максимальна відстань редагування[7]. Під час перевірки слова для кожного поточного символу обчислюється характеристичний вектор до заданого шаблону, де в залежності від його поточного стану та значення вибирається наступний стан автомата із таблиці всіх можливих переходів. Відповідно до закінчення всіх вхідних символів необхідно перевірити, чи є поточний стан

фінальним. На рис. 1.10 зображено приклад таблиці переходів для вхідного шаблону “ab”.

$\chi(x, x_{i+1}x_{i+2}x_{i+3})$	A_i	B_i	C_i	D_i	E_i
$\langle 0,0,0\rangle$	C_i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\langle 1,0,0\rangle$	A_{i+1}	B_{i+1}	B_{i+1}	B_{i+1}	B_{i+1}
$\langle 0,1,0\rangle$	E_i	$\emptyset$	B_{i+2}	$\emptyset$	B_{i+2}
$\langle 0,0,1\rangle$	C_i	$\emptyset$	$\emptyset$	B_{i+3}	B_{i+3}
$\langle 1,1,0\rangle$	A_{i+1}	B_{i+1}	C_{i+1}	B_{i+1}	C_{i+1}
$\langle 1,0,1\rangle$	A_{i+1}	B_{i+1}	B_{i+1}	D_{i+1}	D_{i+1}
$\langle 0,1,1\rangle$	E_i	$\emptyset$	B_{i+2}	B_{i+3}	C_{i+2}
$\langle 1,1,1\rangle$	A_{i+1}	B_{i+1}	C_{i+1}	D_{i+1}	E_{i+1}

Рисунок 1.10 – Приклад таблиці переходів для шаблону “ab”

1.3 Висновки до розділу

У цьому розділі було детально розглянуто, описано та виокремлено переваги та недоліки різних алгоритмів нечіткого пошуку. Всі алгоритми нечіткого пошуку були умовно розділені на 2 категорії: класичні алгоритми та алгоритми на основі скінченних автоматів. У процесі дослідження було виявлено, що класичні алгоритми гарно підходять для пошуку подібності двох конкретних рядків. Натомість скінченні автомати краще підходять для пошуку одного конкретного пошукового слова в тексті.

При порівнянні алгоритмів нечіткого пошуку виявилося, що кожен з них має свої переваги та обмеження, які слід враховувати, вибираючи його для конкретного застосування.

У кожного з цих алгоритмів є свої сильні та слабкі сторони, і вибір конкретного алгоритму повинен залежати від конкретних вимог та обмежень застосування. Нечіткий пошук на основі автоматів може бути відмінним варіантом для деяких завдань, порівняно з іншими алгоритмами нечіткого пошуку. Особливості алгоритмів на основі автоматів:

1) Ефективність обробки. Автомати можуть бути ефективними для обробки великої кількості даних, оскільки вони можуть бути побудовані для швидкого пошуку підрядків у тексті. Це може бути особливо корисно для пошуку великих складових текстів або для розпізнавання патернів у потоках даних.
2) Гнучкість. Автомати можуть бути налаштовані для різних видів нечіткого пошуку, включаючи пошук заміни, вставки, видалення та транспозиції.
3) Пам'ять та швидкість. Зазвичай автомати вимагають менше пам'яті для зберігання правил пошуку, порівняно з іншими алгоритмами, такими як, наприклад, алгоритм Левенштейна.
4) Складність реалізації. Хоча автомати можуть бути потужними для нечіткого пошуку, їхню реалізацію може бути складніше порівнювати з іншими алгоритмами. Вони вимагають побудови та оптимізації автомату для конкретної задачі.

Автомат на основі дерева є найбільш загальним рішенням, скінченний автомат якого має вигляд префіксного дерева та дозволяє мати різну ціну за кожну операцію редагування. Автомат на основі хешування робить припущення про рівність ціни за кожну операцію, що дозволяє побудувати більш структурований та простий скінченний автомат, хешуючи значення його станів. Проте детермінізація цього автомату є складнішою за детермінізацію автомата на основі дерева. Табличний автомат працює з автоматом схожим на автомат на основі хешування. Проте потрібно наперед задати таблицю всіх можливих переходів для всіх редагувальних відстаней на основі входжень символу до шаблону. Це дозволяє йому пропустити етап детермінізації автомата та одразу перейти до перевірки слова. Найбільш універсальним є автомат на основі хешування, адже префіксний автомат неоптимально працює із довгим пошуковим словом, а табличний автомат є швидким лише для невеликої максимальної відстані редагування.

РОЗДІЛ 2. МЕТОД ПОШУКУ РЕЛЕВАНТНИХ ОБ’ЄКТІВ ДО ПОШУКОВОЇ ФРАЗИ

2.1 Пошук релевантних об’єктів

Пошук релевантних об'єктів – процес знаходження таких об'єктів, як: документи, веб-сторінки, зображення тощо, які є найбільш відповідними до певної пошукової фрази або запиту користувача [85]. Ця задача є ключовою для багатьох інформаційних систем і пошукових рушіїв, оскільки дозволяє користувачам швидко знаходити потрібну інформацію серед великого обсягу даних. Основні етапи задачі пошуку релевантних об'єктів включають:

1. Збір та індексація даних. Спочатку необхідно зібрати та індексувати всі дані, за якими буде відбуватись пошук. Це може включати в себе індексацію веб-сторінок, документів, баз даних тощо;
2. Токенізація індексу. Текстові дані розбиваються на окремі токени, це можуть бути слова або фрази, щоб покращити точність пошуку [86];
3. Відбір кандидатів. Пошук систематично вибирає об'єкти, які можуть бути релевантними до пошукового запиту. Це включає в себе використання ключових слів або інших критеріїв для відбору об'єктів;
4. Оцінка релевантності. Для кожного обраного об'єкта обчислюється рівень релевантності до пошукового запиту. До цього належить порівняння токенів у запиті з токенами в об'єкті та використання таких метрик, як відстань Левенштейна, для визначення ступеня відповідності;
5. Представлення результатів. Найбільш релевантні об'єкти представляються користувачеві у вигляді списку або в іншій формі, яка дозволяє швидко оцінити їх релевантність та вибрати потрібний [87];
6. Навігація та фільтрація. Користувач повинен маги можливість навігації серед результатів пошуку, використовуючи фільтри та інші інструменти для звуження результатів до більш конкретних об'єктів.

Пошук релевантних об'єктів є складною задачею, яка вимагає використання різних методів та алгоритмів для досягнення точних та швидких результатів.

2.1.1 Постановка задачі

Поставлена задача у дослідженні була розглянута у наступному вигляді. Нехай X – множина документів, які знаходяться у середовищі для зберігання даних D, яке представлено на рис. 2.1. Документи можуть бути різних типів, таких як: doc, email, pptx, txt, pdf, html, image, cpp і т. д., головне, щоб кожен документ містив текстові дані у своїй назві або контенті. S – пошукова фраза, яку задає користувач для пошуку, також вона називається шаблоном. Шаблон може складатись з одного слова, декількох слів, які утворюють речення чи фразу, або набору незалежних слів. Пошукова фраза може бути задана будь-яким unicode символом.

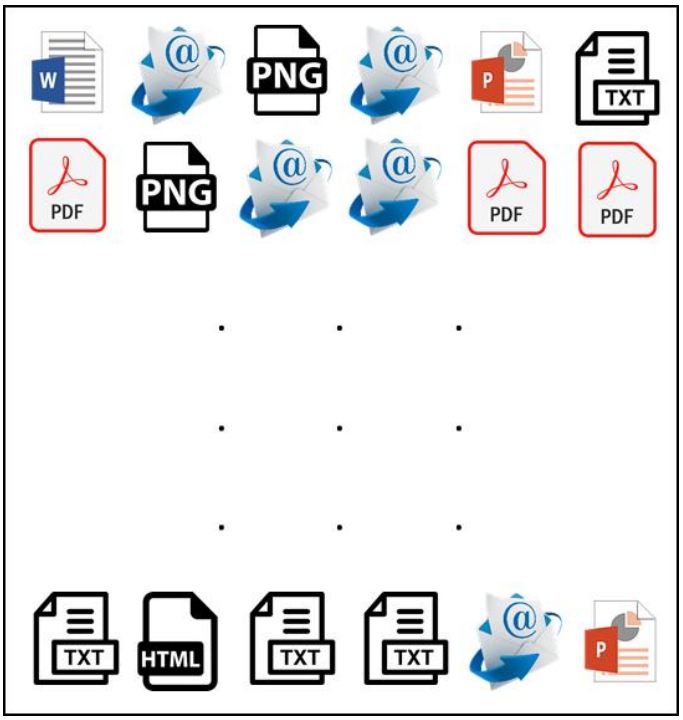

Рисунок 2.1 – Середовище для зберігання даних

Тоді $Y \in X$ – підмножина документів з X, які будуть найбільш релевантними до пошукової фрази S і відсортовані в порядку зменшення релевантності відповідно до метрики подібності текстових даних.

Середовище для зберігання даних – інфраструктура або система, яка призначена для зберігання, організації та управління даними. Це може бути фізичне обладнання, таке як сервери та сховища даних, або програмне забезпечення, таке як бази даних або хмарні сервіси зберігання даних. Середовище для зберігання даних забезпечує надійність, доступність та безпеку даних, а також може надавати інші функції, такі як: резервне копіювання, реплікація та шифрування. Вибір конкретного середовища для зберігання даних залежить від потреб користувачів, обсягу даних та інших факторів. Середовище для зберігання даних може бути реалізоване у формі різних систем та технологій. Одним з найпоширеніших типів є бази даних, які можуть бути реляційними, нереляційними або гібридними. Реляційні бази даних, такі як MySQL, PostgreSQL або Oracle, використовуються для зберігання даних у вигляді таблиць зі зв'язками між ними. Нереляційні бази даних, такі як MongoDB або Cassandra, призначені для зберігання даних у вигляді документів, ключ-значення або колонок [87]. Гібридні бази даних поєднують у собі реляційні та нереляційні підходи для забезпечення більш гнучкого зберігання даних.

Крім баз даних, для зберігання даних також використовуються файлові системи, хмарні сервіси зберігання даних, такі як Amazon S3 або Google Cloud Storage, а також спеціалізовані системи для зберігання великих обсягів даних, такі як Hadoop або Spark. Вибір конкретного середовища для зберігання даних залежить від різних факторів, а саме: обсяг даних, тип даних, доступність, надійність, швидкість доступу та вимоги до безпеки [88-90]. Деякі організації використовують комбінацію різних середовищ для зберігання даних, щоб відповідати своїм унікальним потребам.

Множина документів – колекція документів, які зазвичай організовані разом за допомогою певної системи або структури. Ця множина може включати текстові документи, зображення, відео, аудіофайли та інші типи даних. У контексті пошуку інформації множина документів може бути використана для пошуку релевантних документів відповідно до певного запиту чи критерію. Множина документів може

бути збережена у різних середовищах, а саме: бази даних, файлові системи або хмарні сервіси зберігання даних. Вона може бути опрацьована за допомогою різних алгоритмів для пошуку, аналізу та обробки даних залежно від конкретної задачі.

Популярні формати документів, які містять текстові дані, включають:

1) Microsoft Word (DOCX, DOC). Стандартний формат для текстових документів, який підтримує форматування тексту, вставку медіафайлів та інші функції [91];
2) PDF (Portable Document Format). Універсальний формат, який зберігає форматування та макет документа незалежно від операційної системи або програмного забезпечення;
3) TXT (Plain Text). Простий текстовий формат без форматування, часто використовується для зберігання чистого тексту без зайвих елементів;
4) CSV (Comma-Separated Values). Формат, в якому дані відокремлюються комами, часто використовується для зберігання табличних даних [90];
5) HTML (Hypertext Markup Language). Формат, в якому зберігаються дані для відображення веб-сторінок, включаючи текст, зображення та інші елементи;
6) JSON (JavaScript Object Notation). Легкий формат обміну даними, який використовується для зберігання структурованих даних, включаючи текст;
7) XML (Extensible Markup Language): Універсальний формат для представлення структурованих даних, включаючи текст та інші елементи.

Microsoft Word формат використовується для зберігання текстових документів, створених у програмі Microsoft Word. DOCX це більш сучасна версія, яка підтримує форматування тексту, таблиці, зображення та інші об'єкти [92]. DOC – старіший формат, який також підтримує багато функцій, але є менш сумісним із новішими версіями програми Word. PDF формат використовується для зберігання документів і дозволяє зберігати макет та форматування документа незмінними, незалежно від того, який програмний засіб використовується для перегляду або друку. Формат TXT є найпростішим текстовим форматом, де кожен символ представляється одним байтом.

Він не підтримує форматування або вкладені об'єкти, ідеально підходить для зберігання простого тексту без зайвих елементів [93]. CSV – формат для зберігання табличних даних, де кожен рядок представляє собою запис, а кожне поле в рядку розділяється комою. CSV дозволяє легко обмінюватися даними між програмами. HTML це мова розмітки, яка використовується для створення веб-сторінок. Файли HTML містять текст, зображення, відео та інші елементи, які браузер відображає у вигляді веб-сторінки. JSON – легкий формат обміну даними, який використовується для зберігання структурованих даних у вигляді пар "ключ-значення". Він часто використовується для обміну даними між веб-додатками. XML – універсальний формат для представлення структурованих даних. Він дозволяє визначати власні теги та структуру даних, що робить його корисним для різних типів даних і додатків.

До менш популярних текстових форматів можна віднести:

RTF – формат, який підтримує форматування тексту, таке як жирний, курсив, вирівнювання та інші елементи. Це універсальний формат, який може бути відкритий багатьма редакторами тексту [94].

ODT – формат, який використовується у вільному офісному пакеті LibreOffice та інших програмах. Він підтримує багато функцій форматування тексту та відкритий стандарт.

EPUB – формат для електронних книг, який підтримує різні функції форматування та ілюстрації. Він широко використовується для зберігання та читання книг на електронних пристроях.

FB2 – формат для зберігання літературних текстів, який підтримує форматування, метадані та інші функції для зручного читання електронних книг.

TeX та LaTeX – системи верстки тексту, які використовуються для створення наукових і технічних документів. Вони дозволяють точно контролювати вигляд та форматування документів.

Markdown – простий формат розмітки, який дозволяє легко створювати форматований текст, що можна перетворити в HTML або інші формати.

Ці формати мають свої властивості і використовуються у різних сферах, від документації до обміну даними в мережі. Всі вони потенційно можуть бути джерелом текстових даних та мають бути оброблені у пошуковій системі.

2.1.2 Розробка методу пошуку релевантних об'єктів до пошукової фрази

Розроблено метод нечіткого пошуку, який складається з 9 кроків та комбінує переваги різних алгоритмів, як на основі детермінованих скінченних автоматів, так і на основі алгоритмів динамічного програмування для підрахунку відстані Дамерау-Левенштейна. Ці алгоритми можуть бути реалізовані мовою програмування C++ та впроваджені у систему нечіткого пошуку документів у середовищі резервного копіювання даних. Схематичне зображення та покроковий опис методу зображено на рис. 2.2.

Для реалізації запропонованого методу потрібно виконати наступні кроки:

- Отримати набір текстових даних із вхідних документів. Розділити текст на слова;
- Перевести кожен символ з набору текстових даних в його базовий аналог за допомогою таблиці подібності;
- Перевести пошукову фразу в її базовий аналог за допомогою таблиці подібності;
- Побудувати детермінований скінченний автомат для кожного слова із пошукової фрази;
- За допомогою ДСА знайти попередню відстань редагування між кожним словом в пошуковій фразі і текстовому наборі;
- За допомогою власноруч модифікованого алгоритму Дамерау-Левенштейна уточнити відстань редагування;

- Поставити у відповідність словам з пошукової фрази, слова з текстового набору;
- Порахувати числову метрику релевантності текстових даних до пошукової фрази;
- Відсортувати та відфільтрувати вхідні документи відповідно до отриманої характеристики.

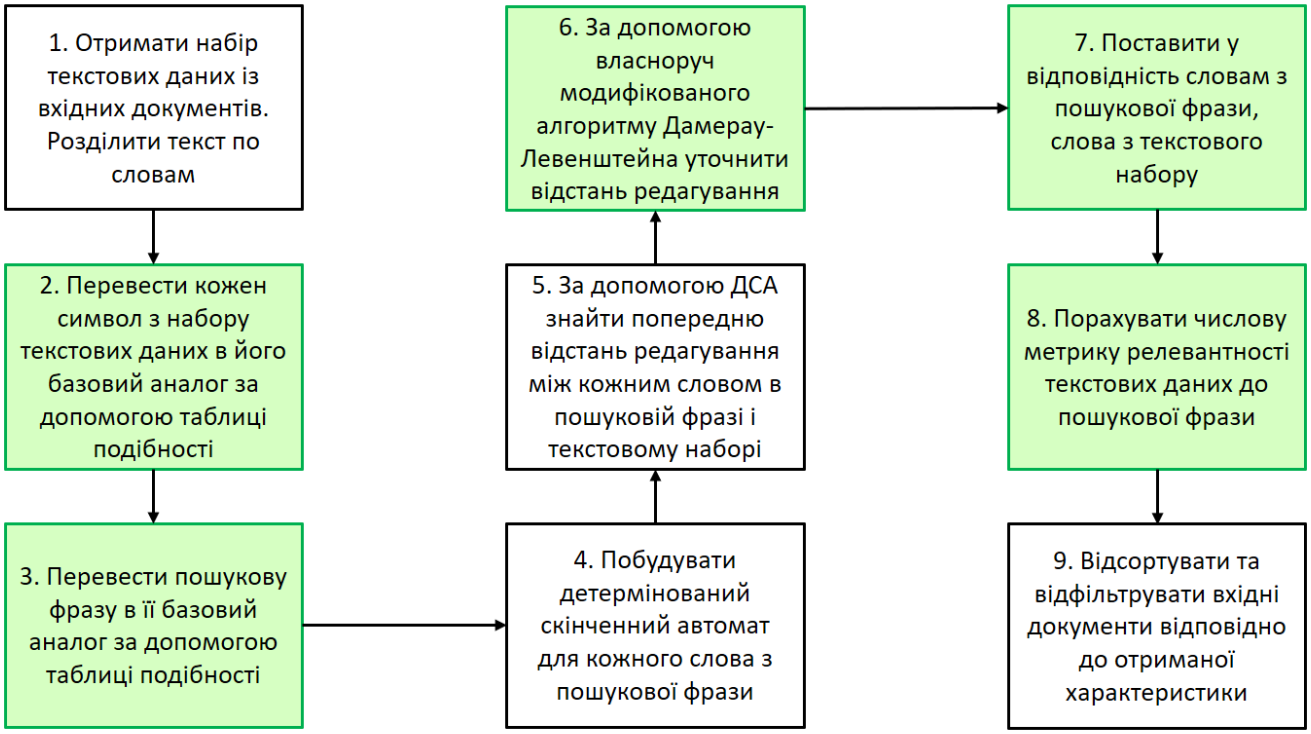

Рисунок 2.2 – Покроковий опис методу

На першому кроці потрібно отримати текст із вхідних документів. Якщо вхідним документом вже є текстовий файл чи веб-сторінка, то попередня обробка даних не потрібна, можна використати необроблені дані. У випадку, якщо на вхід системі надходять картинки чи презентації, то як текстові дані можна використати назву файлу. Після цього необхідно розділити отриманий текст на слова, використавши роздільні символи або пробіли. Другий та третій крок запропонованого методу виконуються за допомогою використання таблиці подібності. На четвертому

кроці необхідно побудувати детермінований скінченний автомат на основі хешування для шаблону або кожного шуканого слова із пошукової фрази. Після проходження п’ятого кроку запропонованого методу, переважна більшість слів із текстового набору буде відфільтрована, адже вони матимуть відстань редагування більшу, ніж максимально допустима підходяща редагувальна відстань. Проте залишаться слова, які потенційно можуть бути релевантними, саме для таких слів потрібно уточнити відстань редагування за допомогою власноруч модифікованого алгоритму Дамерау-Левенштейна із використанням таблиці подібності.

Наукова новизна або певні модифікації були зроблені на етапах методу під номерами 2, 3, 6, 7, 8. Саме ці блоки відмічені зеленим кольором на рисунку вище.

2.2 Методи визначення подібності символів

Існує кілька методів пошуку подібності символів, які можна використовувати в алгоритмах нечіткого пошуку. До них можна віднести розглянуті нижче методи. Порівняння кожного конкретного символу із кожним іншим символом: цей підхід полягає у порівнянні кожного символу в одному рядку із відповідним символом у другому рядку [95]. Якщо символи співпадають, то це зараховується як збіг. Цей підхід досить простий, але може бути витратним з точки зору обчислень для великих текстів.

Наступний підхід це використання хеш-функцій. Хеш-функції використовуються для створення унікальних ідентифікаторів або хешів для кожного символу або групи символів у рядку. Потім можна порівняти хеші замість порівняння символів безпосередньо. Цей метод може прискорити процес порівняння [96].

Використання матриць подібності: такі матриці застосовуються для збереження інформації про те, наскільки два символи подібні один до одного. Це дозволяє використовувати більш складні алгоритми для пошуку подібності, наприклад, алгоритми динамічного програмування.

Використання автоматів зі змінною довжиною вхідних даних (VFA): автомати VFA можуть бути використані для пошуку шаблонів у тексті, де довжина шаблону не відома наперед. Вони дозволяють ефективно шукати подібності в тексті із різною довжиною шаблонів. Ці методи можуть бути використані окремо або у поєднанні для покращення результатів пошуку подібності символів у тексті [97].

2.2.1 Таблиця подібності символів

Таблиці подібності символів використовуються для визначення ступеня подібності між символами або об'єктами. Ці таблиці можуть бути представлені у вигляді матриці, де кожен елемент показує ступінь подібності між двома символами. Таблиці подібності символів можуть бути використані у різних областях, а саме: обробка природної мови, біоінформатика, комп'ютерний зір та інші. Основні характеристики таблиць подібності символів включають:

- Числове представлення. Кожен елемент у таблиці подібності символів містить числове значення, яке вказує ступінь подібності між двома символами. Ці значення можуть бути задані вручну або визначатися автоматично на основі статистичних даних [103];
- Символьна подібність. Таблиці подібності символів дозволяють визначити ступінь подібності між символами або буквами. Наприклад, у біоінформатиці такі таблиці можуть використовуватися для визначення ступеня подібності між амінокислотами або нуклеотидами;
- Використання в алгоритмах. Таблиці подібності символів широко використовуються в алгоритмах пошуку подібності для пришвидшення процесу порівняння. Наприклад, в алгоритмі Дамерау-Левенштейна використовується таблиця подібності символів для визначення ступеня редагування, необхідного для перетворення одного рядка в інший;

- ❖ Застосування у машинному навчанні. У машинному навчанні таблиці подібності символів можуть використовуватися для розробки моделей класифікації або кластеризації на основі ступеня подібності між вхідними об'єктами [104].

Таблиці подібності символів є важливим інструментом для аналізу та порівняння символів або об'єктів у різних областях обробки даних.

Таблиця подібності символів, відома також як таблиця відображення символів, є інструментом, що дозволяє визначити подібність між символами різних мов або просто візуально подібними символами. Вона є важливим компонентом для транслітерації, транскрипції або конвертації текстів між різними системами письма. Така таблиця допомагає знаходити еквівалентні символи із різних алфавітів, що є важливим при обробці багатомовних систем. Наприклад, вона може бути корисною при перетворенні слова або фрази, написаної однією мовою, на іншу або допомогти під час пошуку відповідності символів чи слів для обробки даних, особливо для нечіткого пошуку в тексті, часто у випадках, коли текст написаний однією мовою, а пошуковий запит – іншою.

Таблиця подібності символів включає в себе символи з різних алфавітів, таких як: латиниця, грецька, кирилиця, а також символи з діакритичними знаками, що використовуються в різних мовах. Крім того, вона може включати спеціальні символи, наприклад, символи нуклеотидів або музичних нот, що використовуються у відповідних областях [98]. Таблиця подібності символів може відрізнятися залежно від конкретної задачі і методу імплементації. Деякі таблиці можуть фокусуватися на конкретних парах мов, тоді як інші можуть охоплювати широкий спектр. Вона також може бути спрямована на візуально подібні символи або включати в себе можливі друкарські помилки. Загальною структурою таблиці подібності символів є матриця, де кожен рядок і стовпець представляють символ, а в комірках матриці зберігається оцінка подібності між відповідними символами.

Оцінка подібності вказує на ступінь схожості між символами з точки зору їхньої форми, звуку тощо. Значення в таблиці зазвичай варіюються від 0 до 1, де 1 означає відсутність подібності, а 0 означає ідеальний збіг. Таблиця має бути симетричною, оскільки подібність між символами A і B така ж, як подібність між символами B і A [103]. Приклад базової таблиці подібності зображений на рис. 2. 3. Таблиці подібності символів є корисним інструментом для розробників програмного забезпечення, лінгвістів, перекладачів та інших фахівців, які працюють з обробкою багатомовного тексту або перетворенням символів.

	a	A	ά	b
a	0	0.25	0.2	1
A	0.25	0	0.35	1
ά	0.2	0.35	0	1
b	1	1	1	0

Рисунок 2.3 – Приклад таблиці подібності

2.2.2 Структура таблиці подібності символів

Таблицю подібності можна представити різними способами, в залежності від потреб конкретного алгоритму або задачі. Як приклад існують наступні варіанти:

- Двовимірний масив. Найпростіший спосіб представлення таблиці подібності – двовимірний масив, де рядки і стовпці відповідають символам, а кожен елемент масиву містить значення подібності або заміни між двома символами [103].
- Хеш-таблиця. Таблицю подібності можна представити у вигляді хеш-таблиці, де ключами є пари символів, а значеннями - їх подібність або заміна.

- Матриця. Таблицю подібності можна представити у вигляді матриці, де рядки і стовпці відповідають символам, а кожен елемент матриці містить значення подібності або заміни між двома символами. Відмінність від двовимірного масиву в тому, що матриця може бути представлена у вигляді окремого класу з методами для доступу до значень і зручного використання [99].
- Перелік списків або словники. Таблицю подібності також можна представити за допомогою переліку списків або словників, де ключами є символи або їх комбінації, а значеннями – відповідні значення подібності або заміни.
- Таблиця інцидентності. Цей підхід є аналогічним двовимірному масиву, але кожен запис представлений окремим об'єктом, який містить інформацію про рядок, стовпець і значення подібності [100].
- Тривимірний масив. Для великих алфавітів можна використовувати тривимірний масив, де третій розмір представляє різні значення подібності або заміни для кожної пари символів.
- Дві матриці для вставки та видалення. Деякі алгоритми можуть використовувати дві окремі матриці для вставки та видалення символів, що дозволяє різними способами впливати на вагу цих операцій.
- Рядок збереження. У випадках, коли таблиця подібності має особливу структуру (наприклад, деякі значення можуть бути виведені з інших), можна використовувати більш ефективні методи зберігання, такі як структура даних "Рядок збереження".

Класична структура таблиці подібності на основі матриці не підійшла через погану швидкодію. Така таблиця виявилась не оптимальною з точки зору обсягів пам’яті для її зберігання, а також асимптотики перевірки подібності між двома символами [101]. Тому в рамках роботи була запропонована та створена інша таблиця

подібності, структура якої зображена на рис. 2.4. Дана таблиця складається із двох словників. В першому словнику ключем є символ, для якого потрібно знайти подібні йому символи, а значенням – відповідно другий словник. У другому словнику ключем є символи, подібні до ключа з першого словника, а значенням є вартість їхньої подібності. Дана структура зберігає властивість симетричності, незважаючи на те, що це призведе до зберігання дублікатів даних. Асимптотична складність знаходження символу у даній структурі буде амортизовано константною O(1) [103].

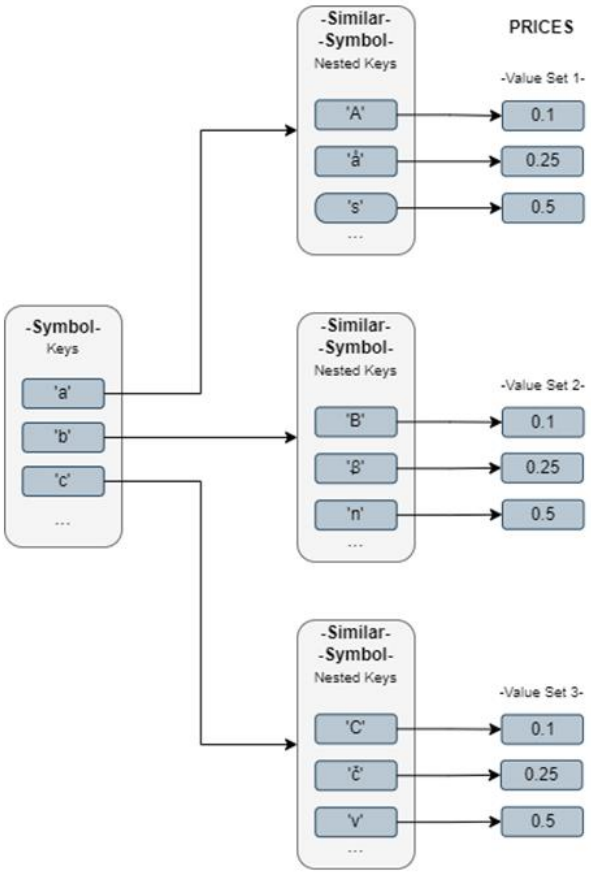

Рисунок 2.4 – Структура таблиці подібності символів

Для прикладу розглянемо деякі схожі символи до англійської літери “a”:

1. “A” – та сама літера, але велика
2. “ä” – схожий символ, проте з іншої мови

3. “s” – літера поруч на клавіатурі

Запропонована таблиця має 2 основні функції:

1) Приймає на вхід 2 символи, і повертає міру їхньої подібності від 0 до 1.

   Наприклад:

   - sim (a, b) = 1
   - sim (a, A) = 0.2
   - sim (a, ä) = 0.25

2) Приймає символ, і повертає базовий для нього

   Наприклад:

   - base (a) = a
   - base (ä) = a
   - base (A) = a
   - base (ß) = b

Обидві функції будуть працювати із константною асимптотикою. В роботі пропонується спосіб заповнення таблиці у вигляді груп у JSON файлі, приклад якого зображено на рис. 2.5.

```
<group>
    <basesymbol>a</basesymbol>

    <in-sequence-cost>0.2</in-sequence-cost>
    <cross-sequence-cost>0.25</cross-sequence-cost>

    <sequence>a,å,à,á,ä,ã,â,æ</sequence>
    <sequence_upper>A,Å,À,Á,Ä,Ã,Â,Æ</sequence_upper>
</group>
```

Рисунок 2.5 – Групування символів таблиці подібності

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

1) Семантично подібні символи

Категорія буде складатись із семантично схожих символів із різних мов до символів з англійського алфавіту. Для побудови таблиці подібності символів із різних мов було використане джерело [113] та мова програмування Python для ефективного аналізу електронного ресурсу. Після аналізу даних можна створити таблицю подібності та зберегти її у форматі JSON. У цьому форматі дані легко серіалізуються та десеріалізуються, що полегшує подальшу обробку та використання цієї таблиці. Приклад цієї категорії зображено на рис. 2.6.

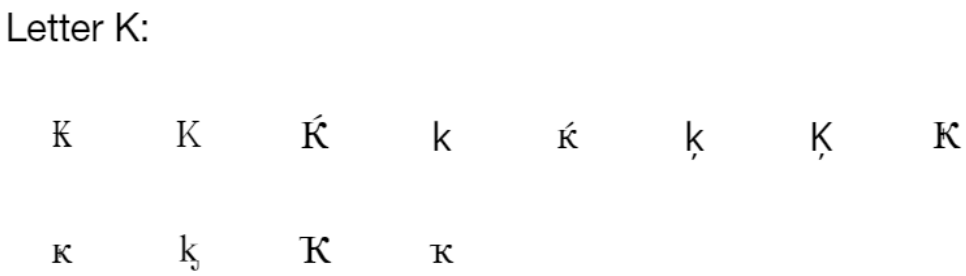

Рисунок 2.6 – Семантично подібні символи [113]

2) Символи, які поруч на клавіатурі

Дана категорія подібних символів буде містити всі можливі друкарські помилки для символів англійського алфавіту. Кожна група подібних символів у даній категорії буде складатися з поточного символу та всіх можливих помилок, таких як оточуючі символи на клавіатурі [103]. Приклад такої категорії зображено на рис. 2.7.

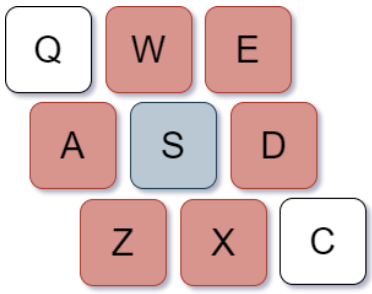

Рисунок 2.7 – Символи, які поруч на клавіатурі

3) Візуально подібні символи

Цю категорію символів необхідно заповнювати вручну, заздалегідь записавши у таблицю всі візуально подібні символи на думку розробника. До візуально подібних можна, наприклад, віднести наступну пару символів “1”, “l” (одиниця та англійська маленька літера “л”), приклад категорії можна побачити на рис. 2.8.

Для візуально подібних символів можна розглядати, наприклад, літери, цифри або символи, які мають схожий або ідентичний вигляд, але мають різні кодування або семантику. Наприклад, літера "O" та цифра "0", які у деяких шрифтах можуть виглядати дуже схоже. Додавання таких пар до таблиці дозволить виявляти та коригувати помилки, пов'язані із плутаниною між ними.

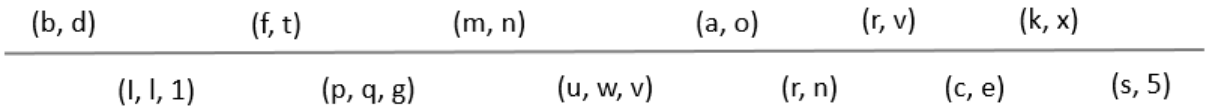

Рисунок 2.8 – Візуально подібні символи

2 та 3 крок запропонованого методу виконуються за допомогою таблиці подібності. Нехай пошукова фраза S = “Každý” (кожен), тоді S(base) = “kazdy”.

2.2.3 Модифікований алгоритм Дамерау-Левенштейна

Як вже було згадано раніше, алгоритм Дамерау-Левенштейна є одним із найпопулярніших підходів для розв’язання задачі пошуку в тексті. Його основна ідея полягає в ітеруванні за текстовими даними чи базою даних, розраховуючи при цьому відстань Дамерау-Левенштейна між пошуковим словом і кожним словом із тексту.

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

1. Створити матрицю *d[][]* розміром *(m+1) * (n+1)*, де *m* і *n* – довжина рядків, між якими потрібно знайти редагувальну відстань.

2. Нехай ціна вставки, заміни, видалення і транспозиції однакові і дорівнюють одиниці.
3. Потрібно проініціалізувати перший рядок і перший стовпець матриці значеннями *0, 1, 2, …, n і 0, 1, 2, …, m*, відповідно.
4. Визначити відстань редагування поточного рядка для кожної комірки матриці:
   – Оскільки в даному алгоритмі також використовується таблиця подібності символів, для початку необхідно визначити подібність символів, які знаходяться у двох рядках відповідно, позначимо її *compareCharCost()*. Вона може набувати різних значень: *0* – якщо символи однакові, певному значенню *X (0 < X < 1)*, яке можна отримати з таблиці подібності символів, або *1* – якщо, згідно таблиці подібності, ці два символи не можна порівняти.
   – Визначити поточну 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 $a_i = b_{j-1}$ та $a_{i-1} = b_j$, то також необхідно врахувати можливу транспозицію символів: *d[i][j] = min(d[i][j], d[i-2][j-2] + 1)* [104].

Підсумовуючи, модифікація формули (1.1) буде мати вигляд:

$$d_{a,b}(i,j) = min\begin{cases} 0 \quad if\ i = j = 0, \\ d_{a,b}(i-1,j) + 1 \quad if\ i > 0, \\ d_{a,b}(i,j-1) + 1 \quad if\ j > 0, \\ d_{a,b}(i-1,j-1) + compareCharCost() \quad if\ i,j > 0, \\ d_{a,b}(i-2,j-2) + 1\ if\ i,j > 1\ and\ a_i = b_{j-1}\ and\ a_{i-1} = b_j, \end{cases} \tag{2.1}$$

5. У результаті, відстань редагування для двох рядків буде дорівнювати *d[m][n]* – число, яке розташовується у нижній правій комірці матриці.

Задавши певне порогове значення та порівнявши всі слова із вхідного тексту із пошуковим шаблоном, використовуючи модифіковану відстань Дамерау-

Левенштейна, можна отримати результати пошуку в тексті із використанням таблиці подібності. Він буде містити всі слова, для яких відстань редагування менша за порогову.

Модифікація алгоритму Дамерау-Левенштейна полягає в тому, що для розрахунку відстані редагування також додається застосування таблиці подібності символів [106]. Завдяки цьому, можна більш ретельно контролювати порівняння подібності символів, дозволяючи алгоритму обробляти випадки, коли певні символи семантично чи візуально схожі на інші [104]. У випадку заміни символу, міра подібності двох символів визначається за допомогою таблиці подібності, у формулі це відображено за допомогою функції *compareCharCost()*, яка зображена на рис. 2. 9.

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

Рисунок 2.9 – Формула для підрахунку модифікованої відстані Дамерау-Левенштейна

2.3 Попередня обробка текстових даних

Попередня обробка текстових даних є важливим етапом аналізу тексту, який включає в себе ряд дій для підготовки тексту до подальшого аналізу. Попередня обробка тексту включає в себе такі основні кроки:

- Токенізація, тобто розбивка тексту на окремі слова або токени.
- Видалення стоп-слів, вилучення незначущих слів, які не несуть корисної інформації для аналізу [105].
- Лематизація або стемінг, тобто приведення слів до їхніх базових форм (лем) або коренів (стем), щоб зменшити розмірність даних і врахувати різні форми одного слова як одне.

- Вилучення спеціальних символів і пунктуації, а також зайвих символів, які не несуть семантичного навантаження, наприклад, коми або крапки.
- Нормалізація регістру, приведення усіх слів до одного регістру, наприклад, до нижнього [107].
- Видалення зайвих пробілів, вилучення зайвих символів, керування форматуванням.
- Вилучення чисел, якщо числа не є важливими для аналізу.

Ці кроки допомагають покращити якість подальшого аналізу тексту, зменшуючи вплив незначущих елементів і забезпечуючи більш точний та зрозумілий результат.

Символи, подібні до англійських літер, є символами з інших алфавітів або мов, які можуть мати схожість з англійськими літерами на основі їхнього зовнішнього вигляду, звучання або значення. Ці символи можуть використовуватися для покращення розпізнавання тексту, зокрема для автоматичного перекладу, пошуку або обробки тексту. Наприклад, у латинському алфавіті є буква “i”, яка виглядає дуже схоже до літери “і” українського алфавіту. Ці символи можуть бути співставлені за допомогою таблиці відповідності, де для кожної літери одного алфавіту вказується відповідна літера іншого алфавіту. Такі символи є важливими для багатьох областей, де використовується обробка тексту з різних мов або для пошуку інформації у багатомовних документах. Отже, використання символів, подібних до англійських літер, може допомогти покращити якість обробки тексту та забезпечити краще співставлення слів із різних мов [108-110].

2.3.1 Типи помилок у текстових даних

Помилки в текстових даних можуть виникати з різних причин і проявлятися у різних формах. Деякі з найпоширеніших помилок включають в себе: орфографічні помилки, наприклад, неправильне написання слова, граматичні помилки, наприклад, невірне вживання частин мови, помилки використання символів, такі як неправильне введення символів на клавіатурі, а також недоліки у форматуванні тексту, наприклад,

невірне вирівнювання або розмітка. Ці помилки можуть виникати в результаті ручного введення тексту, автоматичного генерування тексту, оптичного розпізнавання символів OCR або перетворення мови на текст STT. Обробка цих помилок важлива для багатьох задач, а саме: пошукові системи, автоматичне розпізнавання мови, машинний переклад, автоматичне виправлення помилок у тексті та інші.

Помилки у документах можуть бути різного характеру і виникають з різних причин. Ось деякі типові помилки, які можна знайти в документах:

- Орфографічні помилки: неправильно написані слова або вживання неправильних слів [111].
- Граматичні помилки: некоректне вживання частин мови, неправильна конструкція речень.
- Пунктуаційні помилки: неправильне розставлення ком, крапок, тире тощо.
- Лексичні помилки: використання невідповідних слів або термінів для позначення понять.
- Стилістичні помилки: некоректний стиль письма або відхилення від загальноприйнятих норм.
- Технічні помилки: помилки у форматуванні, розміщенні тексту, неправильне використання шрифтів тощо [112].
- Фактичні помилки: неправильна інформація, помилкові факти або дані.
- Помилки перекладу: невірний або незрозумілий переклад тексту.

Ці помилки можуть бути виявлені та виправлені різними способами, включаючи: ручну перевірку, використання редакторських програм, автоматичне виправлення помилок у тексті за допомогою програмного забезпечення.

Також користувачі часто допускають помилки у процесі введення текстових даних. Ці помилки можуть включати в себе: друкарські помилки, неправильні слова, невірний регістр літер тощо. Ці помилки можуть ускладнювати пошук і аналіз тексту,

особливо в автоматизованих системах, які опрацьовують великі обсяги даних. Для вирішення цієї проблеми можна використовувати методи автоматичної обробки тексту, такі як: виправлення друкарських помилок, нормалізація тексту, фільтрація стоп-слів тощо. Також можна використовувати алгоритми нечіткого пошуку, які дозволяють знаходити схожі слова навіть за наявності помилок у них.

Крім того, можна використовувати методи машинного навчання для автоматичного виявлення та виправлення помилок у тексті. Ці методи можуть використовувати моделі, навчені на великому обсязі текстових даних, щоб автоматично визначати правильне слово або скоригувати неправильне слово у тексті. Загалом, обробка помилок у тексті є важливою складовою частиною багатьох застосувань обробки текстів, таких як: пошукові системи, автоматичні коректори, системи відповідей на запитання та інші.

Помилки в архівних даних можуть бути серйозними, оскільки вони можуть призвести до втрати або пошкодження важливої інформації [115]. Серед них можна виділити пошкодження файлів, бо вони можуть бути пошкоджені під час зберігання або передачі через мережу, втрату даних, адже деякі дані можуть бути втрачені через помилки в архіві або недостатній рівень захисту даних. Некоректна структура даних може призвести до проблем із доступом до інформації. Неправильна індексація може спричинити неможливість знаходження потрібних даних. Несумісність між форматами даних може призвести до втрати інформації або неможливості її коректного відображення. Невірні атрибути файлів можуть призвести до недоступності чи неправильного відображення даних. Недостатній рівень безпеки може загрожувати втратою даних або несанкціонованим доступом до них.

Для запобігання цим помилкам важливо використовувати надійне програмне забезпечення для архівування та зберігання даних, регулярно робити резервні копії і перевіряти їх на наявність помилок [116]. Також важливо використовувати

стандартизовані формати для зберігання даних і дотримуватися рекомендацій із безпеки даних.

2.3.2 Токенізація тексту

Швидкий метод матчінгу токенів у тексті може бути реалізований за допомогою алгоритму Боєра-Мура або алгоритму Кнута-Морріса-Пратта (KMP). Ці алгоритми дозволяють ефективно шукати входження певного слова або шаблону у тексті, використовуючи підходи, що враховують характеристики шуканого слова для прискорення пошуку. Алгоритм Боєра-Мура використовує принцип "прискореного зіткнення" для зменшення кількості порівнянь між шуканим словом та текстом. Він спочатку створює таблицю зміщень для кожного символу у шуканому слові, що вказує на те, на скільки можна змістити шукане слово вправо, якщо зіткнення виявлено. Цей підхід дозволяє пропускати порівняння для символів, які вже були виявлені раніше [117]. Алгоритм KMP використовує префіксну функцію для знаходження взаємозв'язку між префіксами та суфіксами шуканого слова. Ця функція дозволяє швидко визначати, на якому місці у шуканому слові починати новий пошук за умови невідповідності. Обидва алгоритми відомі своєю ефективністю для швидкого пошуку великих об'ємів тексту на входження певного слова або шаблону [118].

Нормалізована метрика подібності слів використовується для вимірювання схожості між двома словами, враховуючи їхню довжину. Ця метрика корисна для порівняння слів, які можуть мати різну довжину, і дозволяє зробити об'єктивні порівняння, незалежно від довжини слів. Один зі способів нормалізації метрики подібності слів – використання відношення довжин слів до їхньої відстані Левенштейна. Нормалізована метрика може бути визначена як:

P_{word}(a, b) = 1 - d / max(len(a), len(b)),

де d – модифікована відстань Дамерау-Левенштейна, len(x) – довжина слова x. Для однакових слів метрика покаже 1, для повністю різних – 0. Цей підхід дозволяє

нормалізувати вимірювання подібності так, щоб воно не залежало від довжини порівнюваних слів, що дозволяє зробити більш точне порівняння.

Токенізація тексту – процес розбиття тексту на окремі частини, які називаються токенами. Токени можуть бути словами, числами, символами пунктуації або іншими одиницями, в залежності від вимог конкретної задачі аналізу тексту. Токенізація є важливим етапом підготовки тексту для подальшого аналізу, такого як: пошук ключових слів, аналіз настроїв тощо. Існує кілька підходів до токенізації тексту, таких як:

- Розбиття на слова: текст розбивається на окремі слова за пробілами або іншими розділовими символами [119].
- Розбиття на речення: текст розбивається на речення за символами кінця речення (крапка, знак питання, знак оклику тощо).
- Розбиття на символи: текст розбивається на окремі символи, які можуть бути корисні для аналізу, якщо потрібно розглядати текст на дуже малих рівнях.

Токенізація може бути складною задачею, особливо для складних мов або текстів із великою кількістю спеціальних символів. У деяких випадках можуть використовуватися спеціалізовані алгоритми або бібліотеки для досягнення кращої якості токенізації. До алгоритмів токенізації можна віднести наступні:

Токенізація за пробілами (Whitespace Tokenization) розбиває текст на токени за допомогою пробілів або інших пробільних символів (таких як табуляція, символ нового рядка). Наприклад, речення "Це речення для токенізації." буде розбите на токени "Це", "речення", "для", "токенізації.".

Токенізація за розділовими знаками (Punctuation Tokenization) розбиває текст на токени за допомогою розділових знаків, таких як: коми, крапки, знаки питання тощо. Наприклад, речення “Це, речення для токенізації?” буде розбите на токени “Це”, “,”, “речення”, “для”, “токенізації”, “?”.

Токенізація за допомогою регулярних виразів (Regular Expression Tokenization) використовує регулярні вирази для визначення правил розбиття тексту на токени. Наприклад, за допомогою регулярного виразу \b\w+\b можна розбити текст на слова.

Токенізація за допомогою граматики (Grammar-based Tokenization) використовує граматичні правила для розбиття тексту на токени. Наприклад, в англійській мові речення можна розбити на слова, використовуючи граматичні правила про пробіли та розділові знаки. Токенізація за допомогою машинного навчання (Machine Learning Tokenization) використовує алгоритми машинного навчання, такі як навчання з учителем або без учителя, для визначення оптимального розбиття тексту на токени [120].

У роботі пропонується спосіб токенізації за допомогою використання спеціальних розділових символів.

2.3.3 Модуль стоп-слів

Модуль стоп-слів – компонент аналізу тексту, який використовується для відсіювання неважливих слів, що зазвичай не несуть значення для аналізу або пошуку. Ці слова, як правило, є дуже поширеними та загальними, такими як "і", "в", "на", "до" та інші. Модуль стоп-слів може бути складений зі списку таких слів, набраний вручну або сформований із готових списків, розроблених для конкретної мови або завдання. Під час аналізу тексту модуль стоп-слів використовує цей список для фільтрації слів, що можуть бути проігноровані під час аналізу тексту [121].

Модулі стоп-слів є важливою складовою для ефективного аналізу тексту, оскільки вони допомагають зменшити обсяг оброблюваної інформації, спрощуючи задачу аналізу. Стоп-слова це ті слова, які вважаються неінформативними у контексті обробки тексту і часто вилучаються з нього перед аналізом або іншою обробкою. Ці слова включають розповсюджені службові слова, а також слова, які не несуть інформації про зміст тексту. Наприклад, в англійській мові до стоп-слів можуть відноситися "the", "and", "or", "but", "is", "of", "a" і так далі. Використання стоп-слів

допомагає зменшити обсяг тексту, який оброблюється, і фокусуватися на більш значущих словах. Це може поліпшити якість аналізу тексту, так як видаляються неважливі слова, які можуть заважати виявленню суттєвої інформації.

Алгоритми визначення стоп-слів зазвичай ґрунтуються на статистичних методах або списках слів, які можуть бути створені вручну або автоматично. Деякі алгоритми використовують підхід "частотного списку", де слова, які зустрічаються найчастіше у текстах, вважаються стоп-словами. Інші методи можуть використовувати лінгвістичні правила або машинне навчання для визначення стоп-слів [122].

Частотний аналіз стоп-слів являє собою процес визначення частоти використання кожного слова в наборі даних та ідентифікації слів, які є стоп-словами. Під час частотного аналізу тексту зазвичай виконують такі кроки для визначення стоп-слів:

- Токенізація тексту: розбивка тексту на окремі слова або токени;
- Підрахунок частоти: підрахунок кількості входжень кожного слова у текст [123];
- Сортування за частотою: сортування слів за їхньою частотою вживання від найбільш вживаного до найменш;
- Вибір стоп-слів: визначення порогового значення частоти, яке визначає, які слова вважаються стоп-словами. Часто це найбільш поширені слова, які відфільтровуються із тексту;
- Фільтрація тексту: вилучення стоп-слів із тексту.

Частотний аналіз стоп-слів допомагає покращити якість текстового аналізу, зменшуючи вплив надмірно вживаних слів, які не несуть значущої інформації.

2.4 Висновки до розділу

У цьому розділі було розглянуто задачу пошуку релевантних документів до пошукової фрази. Відповідно, пошук релевантних документів до певної пошукової

фрази – процес знаходження документів, які найбільш точно відповідають запиту користувача. Для цього використовуються різноманітні техніки обробки тексту та аналізу даних. Для розв’язання цієї задачі в роботі розроблено метод нечіткого пошуку, який складається з 9 кроків і поєднує в собі переваги існуючих методів, які базуються на використанні скінченних детермінованих автоматів та методів динамічного програмування для пошуку відстані редагування.

Було розглянуто принцип та підходи до побудови таблиці подібності символів, а також саму таблицю подібності символів, що є важливим інструментом для різноманітних областей, таких як: комп'ютерні науки, лінгвістика, переклад та дизайн шрифтів. Таблиця дозволяє знаходити еквівалентні символи з різних мов, ще її можна використовувати для транслітерації, транскрипції та конвертації між різними системами письма. Також таблиця використовується для пошуку відповідностей символів для обробки даних, зокрема для нечіткого пошуку в тексті.

У роботі пропонується структура таблиці подібності на основі двох словників, адже саме така структура дозволяє виконати основні операції із константною асимптотикою. Також важливим аспектом таблиці подібності є врахування різних видів подібності, таких як семантична, візуальна та фонетична. Це дозволяє використовувати таблицю для різних цілей, від пошуку відповідностей між символами до аналізу та обробки текстових даних.

В результаті аналізу встановлено, що використання таблиці подібності символів дозволяє значно покращити точність алгоритму Дамерау-Левенштейна. Даний модифікований алгоритм особливо необхідний у випадках, коли вхідний текст, в якому відбувається пошук, написаний однією мовою, а користувач буде робити запит іншою, через це він може в пошуковому запиті замінювати символи однієї мови на семантично схожі з іншої.

Запропоновано метод реалізації таблиці подібності символів, яка в свою чергу, дозволила враховувати можливу семантичну подібність символів у словах. Оцінюючи

подібності на основі різних критеріїв, таких як: форма, контекст і фонетика, таблиця подібності символів дозволила модифікувати алгоритм нечіткого пошуку, який може ранжувати відповідні результати навіть за наявності великої кількості неспівпадінь unicode значень схожих символів.

РОЗДІЛ 3. ВЕРИФІКАЦІЯ ОДЕРЖАНИХ РЕЗУЛЬТАТІВ

Верифікація одержаних результатів нечіткого пошуку є важливою складовою процесу пошуку інформації. Основна мета верифікації полягає в підтвердженні достовірності, точності і релевантності отриманих результатів. Деякі ключові аспекти верифікації результатів включають:

- Порівняння з оригінальним джерелом, результати нечіткого пошуку можуть бути порівняні з оригінальними джерелами або документами для перевірки відповідності.
- Перевірка відповідності контексту, важливо враховувати контекст пошукового запиту та результатів для забезпечення їх відповідності.
- Аналіз релевантності, результати можуть бути оцінені на релевантність до поставленого питання або завдання.
- Перевірка даних, важливо перевірити дані, які були використані для нечіткого пошуку, на їхню достовірність та актуальність.
- Контрольовані експерименти, необхідно виконати контрольовані експерименти для перевірки ефективності і точності алгоритмів нечіткого пошуку.
- Перевірка результатів аналітики, важливо перевіряти результати аналізу на правильність [123].
- Залучення експертів, експертне оцінювання може бути використане для верифікації результатів, особливо в складних випадках.

3.1 Порівняння скінченних автоматів нечіткого пошуку

На цьому етапі необхідно обрати алгоритм нечіткого пошуку, який швидко знайде всі потрібні слова із текстового набору, близькі до пошукового слова або шаблону, тобто з редагувальною відстанню <=d_max. Для поточної задачі ідеально підходять алгоритми на основі скінченних автоматів. Адже такі алгоритми будують всі можливі патерни слів, які відрізняються від пошукового слова не більше, ніж на

зазначену редагувальну відстань. На практиці тестувались 3 різні рішення на основі скінченних автоматів [102].

*3.1.1 Тестування продуктивності автомат*ів

Наступним нашим кроком є аналіз та написання тестів продуктивності для кожного із представлених рішень скінченних автоматів. А саме автомата на основі префіксного дерева, автомата на основі хешування та автомата на основі таблиці переходів. Потрібно провести тести, що визначать час побудови автомата та час перевірки слова для кожного з рішень. У тестах розраховується час для двох варіантів перевірок, тобто для співпадінь та неспівпадінь, або ж слів, що не приймаються автоматами. Також у тестах вимірюється максимальне використання пам’яті під час побудови автоматів.

Фінальний час вимірюється на загальному тесті, що імітує умови завдання, а саме: побудову автомата та перевірку великої кількості рядків один з одним. Також відбувається порівняння із часом, який витрачається тестом зі звичайним алгоритмом Дамерау-Левенштейна.

Для проведення та створення тестів було використано фреймворк google::benchmark [114]. Він допомагає обчислювати час, затрачений на виконання, і дозволяє швидше провести запуск тестів продуктивності. Google Benchmark – фреймворк для проведення надійних та повторюваних вимірювань швидкодії програмного забезпечення у середовищі C++. Він надає зручний інтерфейс для написання тестів для вимірювання продуктивності різних частин програми та порівняння їхньої швидкості. Основні функції та можливості Google Benchmark включають:

- Зручний синтаксис для оголошення тестів швидкодії.
- Автоматичне виведення результатів вимірювань, включаючи середні значення, середньоквадратичне відхилення та інші статистичні показники.

- Підтримка налаштування кількості ітерацій для кожного тесту [114].
- Можливість запускати тести у різних режимах виконання, таких як: одноразовий запуск тестів, множинний запуск тестів для отримання середніх значень та інші.
- Підтримка обмеження вимірювань до певного часу або кількості ітерацій.

Загалом, Google Benchmark допомагає розробникам ефективно оцінювати швидкодію свого коду, що дозволяє їм прийняти кращі рішення для оптимізації програм та вибору оптимальних алгоритмів [114].

Тести були представлені у наступному вигляді. У тесті, в якому вимірюється час та максимально затрачена пам’ять на створення автомату, формується автомат для шаблонів різної довжини та різної максимальної редагувальної відстані. Для побудови використовується такий набір слів: “a”, “an”, “cat”, “dogs”, “apple”, “banana”, “compute”, “language”, “algorithm”, “innovation”, “engineering”, “intelligence”, “complimentary”, “infrastructure”, “characteristics”.

Для перевірки слів було створено два окремих тести: перевірка слів, які приймаються автоматом, та перевірка слів, які не приймаються. Це дозволило проаналізувати два можливі окремі випадки перевірки слів та їхній час виконання окремо один від одного. У тесті спочатку завчасно створюється автомат, що перевіряється, та проводиться перевірка або на початковому слові, або на слові, що відрізняється від нього. Загальний тест проводить побудову автомата та перевірку на великій кількості різних слів [102].

Умови запуску тестування в Google Benchmark повинні бути налаштовані таким чином, щоб забезпечити надійні та чіткі результати. Деякі коректні умови включають також чітко визначені цілі тестування, тому перед початком тестування потрібно визначити чіткі цілі та очікувані результати. Це допоможе уникнути невиправданих очікувань та невірної інтерпретації результатів. Також потрібно переконатись, що умови тестування стабільні та незмінні протягом усього тесту. Це важливо для

отримання порівнянних результатів. Під час тестування виконується достатня кількість повторень кожного тесту, щоб отримати статистично значущі результати. Зазвичай рекомендується виконувати кожен тест кілька разів і обчислювати середнє значення та інші статистичні показники. Важливо уникати впливу фонових процесів, щоб тести виконувались на комп'ютері, який не має інших активних процесів, які можуть впливати на результати тестування. Також необхідна правильна конфігурація обладнання та програмного забезпечення, щоб було перевірено, що вони налаштовані відповідно до умов вимірювання швидкодії. Потрібно простежити, щоб тести були запущені в умовах, коли система не перевантажена і не використовується для інших завдань, що можуть впливати на швидкість виконання.

Тести виконано на обладнанні, характеристики якого вказано в таблицях 3.1 та 3.2.

Таблиця 3.1 – Характеристики центрального процесору

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	39 bits physical, 48 bits virtual
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	142
Model name:	Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
Stepping:	12
CPU MHz:	2100.000
CPU max MHz:	4200,0000
CPU min MHz:	400,0000
BogoMIPS:	4199.88
Virtualization:	VT-x
L1d cache:	128 KiB

```
L1i cache:                            128 KiB
L2 cache:                             1 MiB
L3 cache:                             6 MiB
```

Таблиця 3.2 – Характеристики оперативної пам’яті

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

Вимірювання затрачених часу та пам’яті на побудову були проведені тільки для HashAutomaton та TreeAutomaton, оскільки TableAutomaton не потребує процесу побудови. На рис. 3.1 зображено графіки залежності часу побудови від довжини пошукового слова для різних значень максимальної редагувальної відстані. Для автомата на основі префіксного дерева вдалось побудувати тільки екземпляри для відстані до 3, оскільки більші редагувальні відстані потребують довгої побудови та займають багато пам’яті [102].

На рис. 3.2 можна побачити графіки залежності максимального об’єму витраченої пам’яті від довжини шаблону. Зі збільшенням максимальної редагувальної відстані затрачений об’єм оперативної пам’яті зростає. Час та пам’ять для автомата на основі префіксного дерева зростають дуже швидко.

На рис. 3.3 відображено графіки, що порівнюють затрачений час та пам’ять для TreeAutomaton та HashAutomaton між собою. Для кожного типу автомата розмір вхідного шаблону змінюється від 1 до 15, а максимально допустима відстань редагування знаходиться в межах до 6. Час, у мілісекундах, для створення автомата збільшується разом зі збільшенням розміру шаблону та максимальної відстані редагування. Це пояснюється тим, що довші шаблони та більші значення відстані редагування вимагають більше обчислювальних ресурсів.

Порівнюючи автомати, HashAutomaton потребує менше часу для створення, порівняно з автоматом на основі дерева, для однакового розміру шаблону та відстані

редагування. Це пов’язано з основною складністю їхніх внутрішніх структур і способом їхньої реалізації.

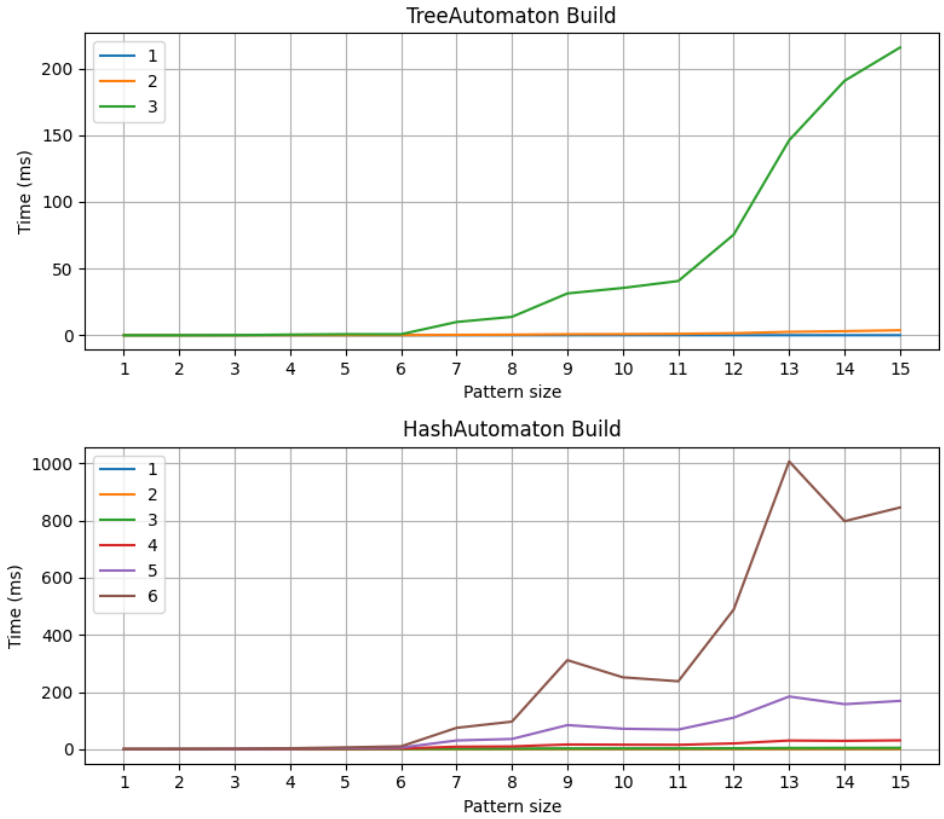

Рисунок 3.1 – Графік залежності часу побудови від довжини шаблону

Є декілька цікавих аномалій або відхилень, наприклад, перехід від розміру шаблону 6 до розміру шаблону 7 є досить швидким. А у випадку автомата на основі хешування із розміру шаблону від 9 до 11, час залишається доволі постійним. Це може бути пов’язано із певними характеристиками алгоритму проходження вздовж автомату чи специфікою даних. Дійсно, використані шаблони для розмірів 6 та 7 виглядають наступним чином: "banana", "compute". Слово “banana” має більшу кількість повторюваних символів, ніж слово “compute”, це також може впливати на процес детермінізації автомату та кінцеву кількість станів і переходів між цими

станами.

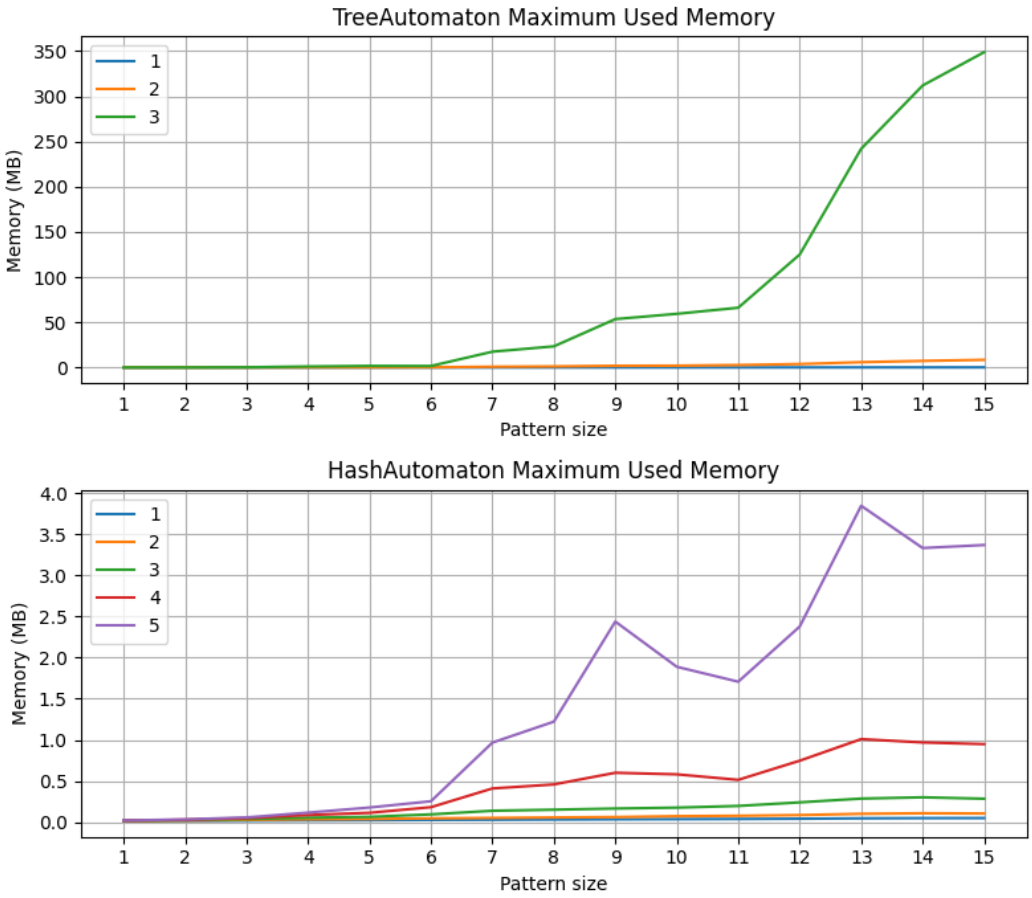

Рисунок 3.2 – Графік залежності максимального об’єму витраченої пам’яті від довжини шаблону

Слова, які були використані для шаблонів розмірами 9, 10, 11 виглядають так: “algorithm”, “innovation”, “engineering”. Вони також містили в собі декілька однакових символів та подвоєнь символів, чим і пояснюється відхилення від зростання часу побудови автомата зі збільшенням розміру вхідного шаблону.

Схожі результати були отримані при порівнянні об’єму затраченої пам’яті для двох рішень. HashAutomaton використав значно менше пам’яті під час побудови, ніж TreeAutomaton, це пов’язано із більшою структуризацією НСА, що в результаті дає меншу кількість станів у ДСА. Також у TreeAutomaton використана структура префіксного дерева, що потребує більшого об’єму виділеної пам’яті.

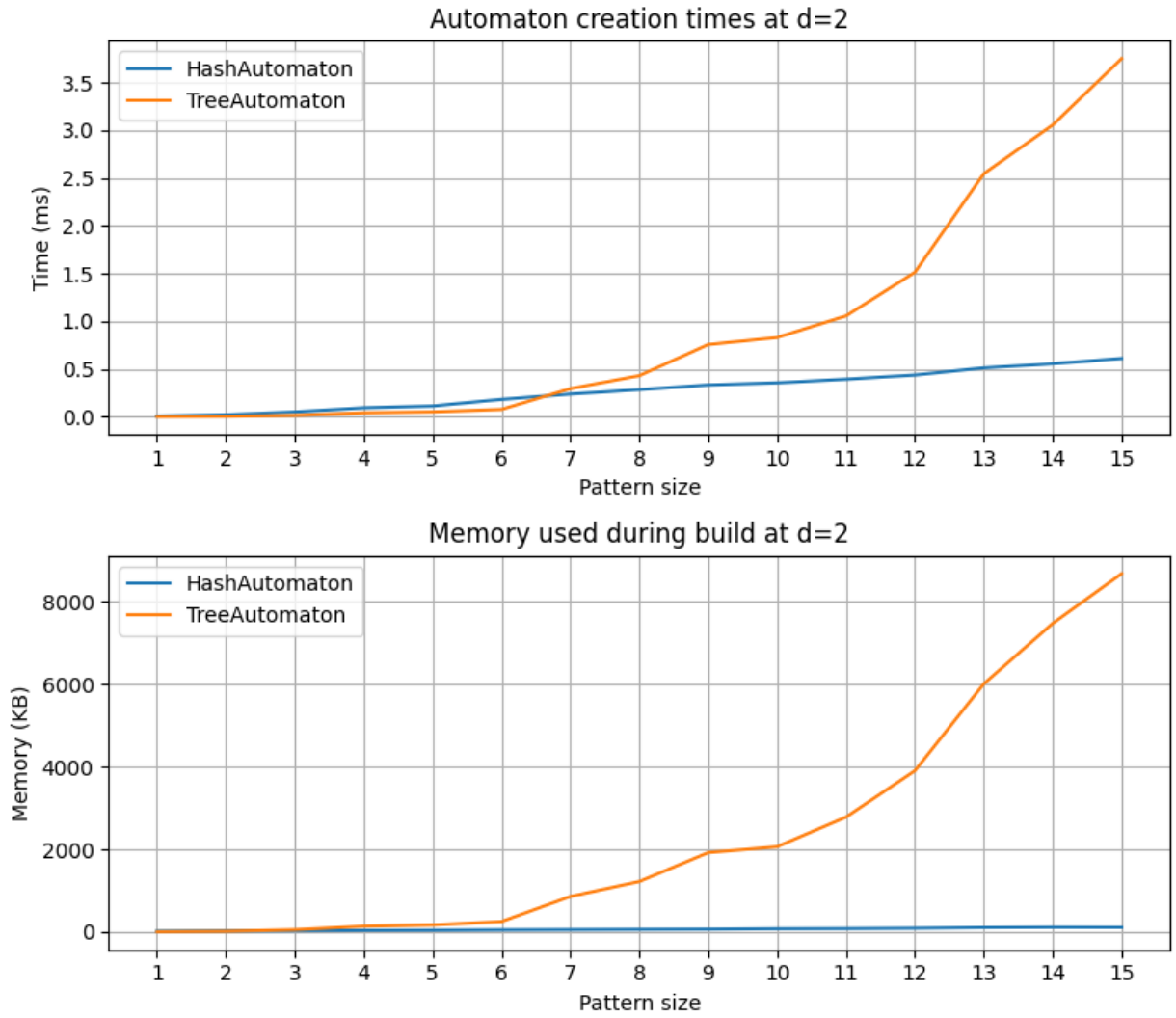

Рисунок 3.3 – Графіки порівняння побудови HashAutomaton та TreeAutomaton для максимальної редагувальної відстані 2

3.1.2 Загальне тестування різних типів автоматів

На рис. 3.4 зображені графіки, що відображають час перевірки слова автоматом. Загальна тенденція є наступною: час перевірки для слів, які приймаються автоматом є більшим, ніж час, затрачений на перевірку слів, які були відхилені автоматом. Це пов’язано з тим, що за умови відхилення послідовності, автомат може припинити виконання десь на середині слова, не зчитуючи послідовність до самого кінця.

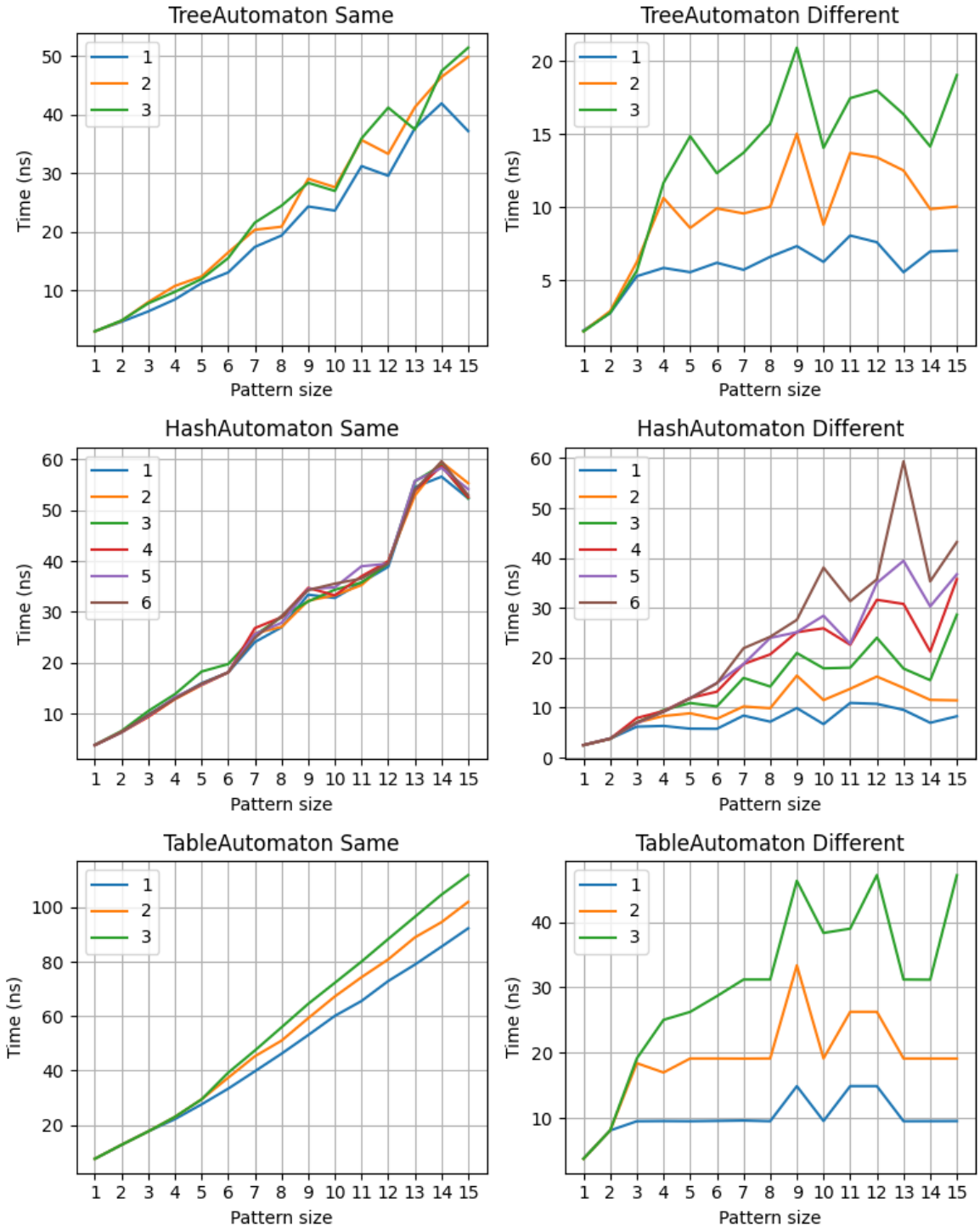

Рисунок 3.4 – Графіки залежності часу перевірки слова від довжини шаблону

Зі збільшенням розміру пошукового слова загальний час також має тенденцію до збільшення. На показники у TableAutomaton та TreeAutomaton помітно впливає максимально дозволена відстань редагування [102]. У першому випадку це пов’язано

з необхідністю розраховувати та обчислювати характеристичний вектор для кожного символу в рядку. У другому випадку це пов’язано із використанням структури даних асоціативного масиву або контейнеру на основі дерев, що асимптотично має логарифмічний час основних операцій, проте за невеликої кількості елементів працює швидше, ніж аналогічний контейнер у мові C++ на основі хеш-таблиці [104].

На рис. 3.5 зображено графіки, що відображають порівняння часу виконання рішень для максимальної редагувальної відстані два. Також до графіку перевірки однакових із шаблоном слів додано час виконання стандартного алгоритму Дамерау-Левенштейна. За результатами графіку видно, що TreeAutomaton та HashAutomaton працюють приблизно однаково, із невеликою перевагою TreeAutomaton з точки зору затраченого часу. Проте TableAutomaton має значно більший час за всіма напрямками та показниками. Це може бути пов’язано з тим, що для виконання перевірки слова потрібно розраховувати характеристичний вектор для кожного символу. Це може бути виправлено шляхом оптимізації на основі SIMD інструкцій процесора, через це скоротиться час його розрахунку, результати можна побачити на рис. 3.6 [102].

Перевірка рядків, що приймаються автоматом тепер не буде залежати від редагувальної відстані, це відбувається шляхом розрахунку характеристичного вектору за декілька інструкцій процесора. Водночас перевірка слів, які були відхилені автоматом, напряму залежить від редагувальної відстані. Через збільшення відстані редагування, в автоматі має пройти більше станів перш, ніж слово відкидається повністю. Відбувається пришвидшення для слів, які не схожі на шаблон, аж в два рази, до значень часу подібних до TreeAutomaton. Проте треба мати на увазі, що ця оптимізація має сенс тільки тоді, коли центральний процесор, на якому запускається алгоритм, підтримує використані інструкції. Також ця оптимізація працює коректно тільки для відносно малих значень редагувальної відстані.

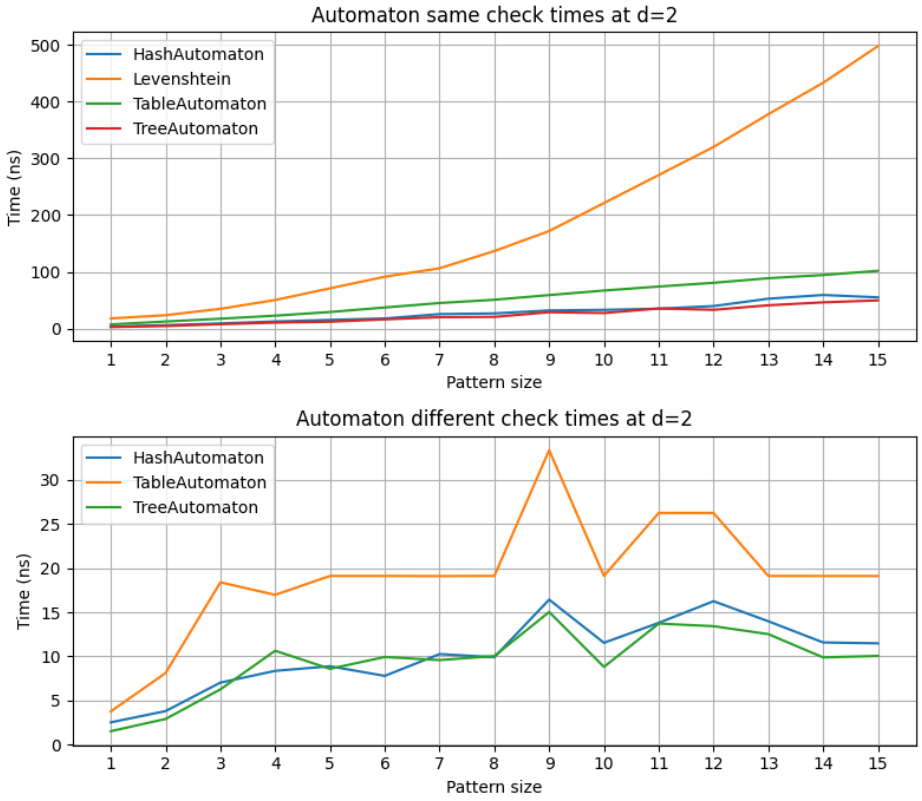

Рисунок 3.5 – Графік порівняння часу перевірки слова для редагувальної відстані 2

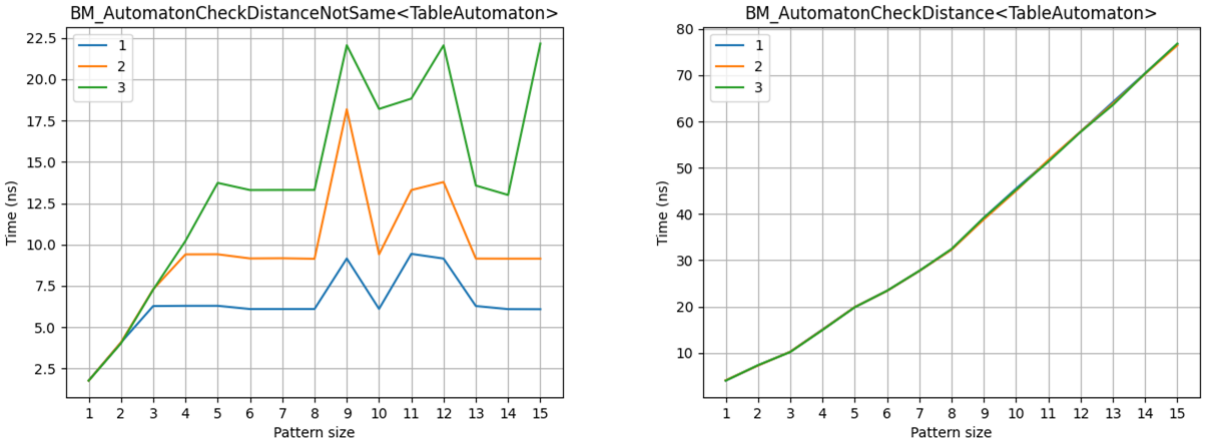

Рисунок 3.6 – Результат оптимізації на основі SIMD інструкцій

3.1.3 Вибір оптимального скінченного автомата на основі тестування

На практиці тестувались вище розглянуті рішення на основі скінченних автоматів. Автомат на основі префіксного дерева реалізує ідею побудови скінченного автомата у вигляді префіксного дерева. Він приймає всі слова, які відрізняються від пошукової фрази не більше, ніж на задану відстань редагування. Водночас враховуючи операції вставки, видалення, транспозиції та заміни. Основним недоліком цього автомата є велика кількість станів для довгого пошукового слова і редагувальної відстані більше 3. Відповідно, це довга побудова і великі затрати пам’яті. Через наведені недоліки, було вирішено відмовитись від використання цього автомата в запропонованому методі.

Другий підхід реалізує автомат на основі хешування і відповідно немає прив’язки до структури префіксного дерева. Нехай вартість кожної операції, чи то видалення, вставка, транспозиція або заміна, однакова і дорівнює одиниці, що дозволяє побудувати більш структурований автомат, сприяючи прискоренню побудови та детермінізації НСА. Кожен стан автомата відповідає певній конфігурації, яка включає кількість оброблених символів у шаблоні та кількість операцій редагування, використаних в процесі. Кожен перехід між станами відповідає певній операції. За принципом роботи, запропонований автомат доволі схожий на автомат Левенштейна, проте на виході, окрім булевого значення, повертає і редагувальну відстань між словами, а також підтримує операцію транспозиції символів. Серед недоліків можна відмітити складний підбір хеш-функції для хешування станів та довгий процес детермінізації. Проте ці недоліки не є критичними і саме цей підхід пропонується як пріоритетний для розв’язання поставленої задачі.

Логіка побудови автомата на основі таблиці переходів нічим не відрізняється від логіки побудови автомата на основі хешування. Основна перевага використання таблиці переходів – відсутність необхідності явної детермінізації [124]. Маючи

шаблон та максимальну редагувальну відстань, автомат одразу може приступити до перевірки слів. Під час аналізу кожного символу слова обчислюється характеристичний вектор відносно заданого шаблону. Залежно від значення цього вектору та поточного стану, визначається наступний стан автомата, який вибирається з таблиці всіх можливих переходів. Після обробки всіх вхідних символів проводиться перевірка, чи є поточний стан фінальним. Цей підхід є практичним тільки для невеликих значень максимальної редагувальної відстані, оскільки розмір таблиці росте експоненційно [102].

Результат виконання загального тесту відображено на рис. 3.7. Цей тест було проведено на словнику, що складається із 370 тисяч англійських слів. Кожне з рішень працює швидше, ніж простий підхід із використанням алгоритму Дамерау-Левенштейна. Відсутність явної побудови автомата не надає значної переваги TableAutomaton, оскільки побудова автоматів відбувається тільки один раз на початку.

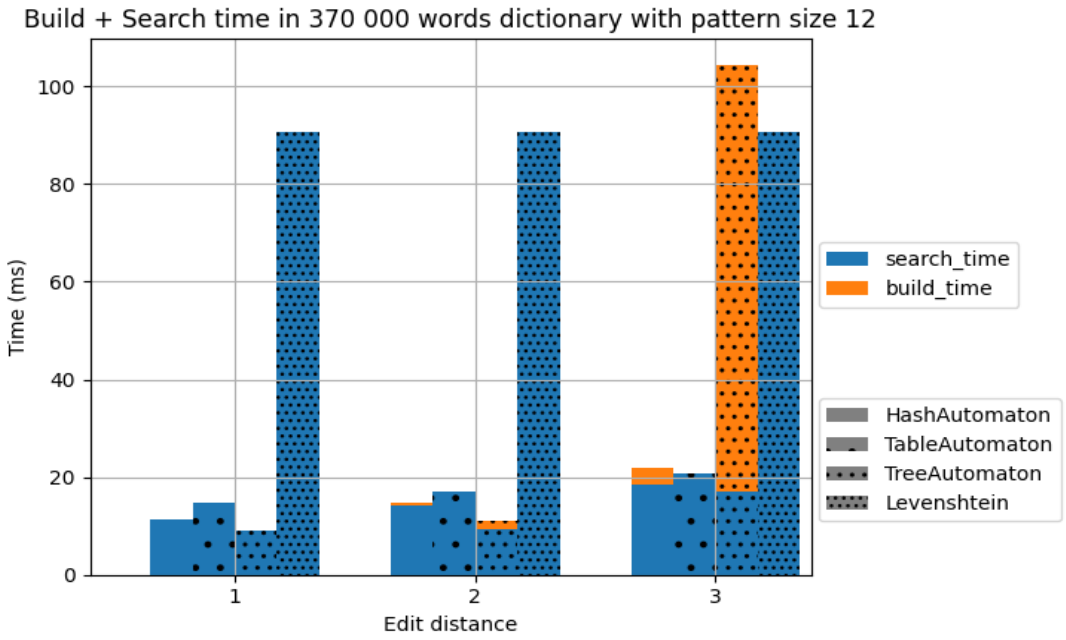

Рисунок 3.7— Результати загального тесту

В результаті дослідження було сформовано рекомендації для розробки алгоритмів нечіткого пошуку на основі скінченних автоматів. За умови малих допустимих відстаней редагування 1 або 2 варто використати TreeAutomaton. Для середніх довжин пошукових слів найкраще підійде TableAutomaton, а за довгих пошукових слів – HashAutomaton. Саме автомат на основі хешування виявився найбільш універсальним і саме він рекомендуються до впровадження. Варто зазначити, що побудова автомата має сенс за умови розміру текстових даних більше 5000 слів, в іншому випадку краще використати простий алгоритм Дамерау-Левенштейна.

3.2 Аналіз результатів використання таблиці подібності символів

У сучасному світі програмне забезпечення стає все складнішим і більш громіздким у реалізації. Більшість програм складаються з багатьох компонентів, які повинні взаємодіяти між собою. Навіть невелика помилка в одному з компонентів може призвести до недоліків або краху всієї системи. Саме тому, тестування програмного забезпечення є надзвичайно важливою частиною розробки програм, що допомагає виявити і усунути помилки перед запуском продукту на ринку. Тому тестування програмного забезпечення грає одну з перших ролей у життєвому циклі розробки ПЗ.

Для проведення тестування розроблених алгоритмів було обрано одну з найпопулярніших бібліотек GoogleTest. GoogleTest – потужна бібліотека для модульного тестування в C++. Вона дозволяє створювати тестові набори, що перевіряють правильність роботи окремих частин програмного коду. GoogleTest надає різні типи асертів для перевірки умов, а також можливості для організації тестів у тестові набори та тестові функції.

Бібліотека також підтримує фікстури, які дозволяють встановлювати початковий стан перед виконанням кожного тесту та видаляти ресурси після його завершення. Це дозволяє писати більш структурований та підтримуваний код тестів.

Однією з особливостей GoogleTest є його швидкодія. Він розроблений з урахуванням ефективності, що дозволяє виконувати велику кількість тестів швидко. GoogleTest також інтегрується з іншими інструментами для тестування. Тестування алгоритму нечіткого пошуку можна розбити на дві підзадачі: тестування таблиці подібності та тестування обчислення відстані редагування. Тест фікстура – об’єкт в GoogleTest, який дозволяє написати декілька або більше тестів, які працюватимуть на подібних даних. Тобто фікстури дозволяють використовувати ту саму конфігурацію об’єктів для кількох різних тестів під час тестування [114].

Спочатку необхідно створити об’єкт самої таблиці подібності, як для випадку тестування таблиці подібності, так і для тестування обчислення відстані редагування. Тому аби розпочати тестування спочатку потрібно створити фікстуру таблиці подібності.

3.2.1 Тестування методів з використанням таблиці подібності символів

Для тестування таблиці подібності символів було створено два різних типи тестів:

1. Тестування внутрішніх даних таблиці подібності.

У даному тесті, завдяки бібліотеці random, випадковим чином вибирається 30 символів. Далі для кожного символу необхідно проітеруватись за його подібними символами і перевірити, чи є вони в таблиці подібності, та чи є серед подібних символ, який тестується. Завдяки цьому тесту перевіряється цілісність таблиці подібності, а точніше, правильність її побудови. Також потрібно перевірити, чи виконується властивість симетричності таблиці подібності.

2. Тестування функції, яка розраховує міру подібності двох символів.

У даному тесті спочатку потрібно перевірити всі можливі випадки, які можна отримати із функції для чотирьох різних символів.

Наприклад, для символу ‘а’ необхідно розглянути п’ять можливих випадків:

- Символи ‘a’ і ‘q’: у даному випадку потрібно перевірити можливу друкарську помилку. Функція повинна повернути міру подібності 0.5;
- Символи ‘a’ і ‘A’: перевіряється випадок, коли букви мають різний регістр. Функція повинна повернути міру подібності 0.1;
- Символи ‘a’ і ‘l’: у даному випадку перевіряються символи, які не мають ніякої схожості. Функція повинна повернути Null, тобто відсутність подібності.
- Символи ‘a’ і ‘å’: перевіряється випадок, коли символи мають семантичну схожість. Функція повинна повернути міру подібності 0.25;
- Символи ‘a’ і ‘o’: перевіряється випадок, коли символи є візуально подібними. Функція повинна повернути міру подібності 0.6.

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

Всі чотири тести пройшли успішно. А отже, завдяки таким результатам можна впевнитися, що розроблений алгоритм працює вірно. Перед виконанням алгоритму нечіткого пошуку із використанням таблиці подібності, необхідно створити саму таблицю подібності, цей процес також вимагає певних затрат часу та ресурсів, тому її створення необхідно протестувати [103].

Сьогодні існує безліч різних варіантів реалізації алгоритмів нечіткого пошуку тексту. Всі вони відрізняються не лише критеріями для оцінювання результатів пошуку чи різними підходами до реалізації алгоритмів, а й швидкістю та точністю виконання пошуку. Швидкість і точність алгоритму впливають на велику кількість аспектів користування додатками. По-перше, ці два критерії безпосередньо впливають на комфорт користувача під час взаємодії із програмою. Користувачі очікують, що програми будуть швидкими та ефективними. Вони не мають бажання

довго чекати, поки програма буде оброблювати їхні запити, якщо існує альтернатива, яка робить це швидше. Тому сьогодні на конкурентному ринку продуктивність програмного забезпечення відіграє ключову роль. Контроль продуктивності програми дозволяє розробникам створити більш конкурентоздатні продукти програмного забезпечення.

Для тестування продуктивності розроблених алгоритмів у цьому пункті було обрано одну з найпопулярніших бібліотек – Google Benchmark. Google Benchmark це open-source бібліотека, розроблена Google, тобто бібліотека з відкритим кодом, для порівняльного аналізу коду мовою C++. Головна мета цієї бібліотеки: надати засоби для вимірювання продуктивності коду C++ та дозволити розробникам точно аналізувати та порівнювати час виконання для різних фрагментів коду або функцій. Бібліотека Google Benchmark розроблена таким чином, щоб бути простою у використанні та водночас забезпечувати надійні та точні результати. Саме тому було обрано цю бібліотеку [114].

Для тестування продуктивності було створимо бенчмарк тест. Дана функція 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. Результати такого тестування наведені в таблиці 3.3.

Таблиця 3.3 – Результати тестування швидкодії створення таблиці подібності

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

Враховуючи ці результати можна вважати, що за 264 спроби створити таблицю подібності середній час показує 2.67 ms.

3.2.2 Аналіз таблиці подібності в алгоритмі Дамерау-Левенштейна

Для перевірки швидкості роботи алгоритмів нечіткого пошуку з використанням таблиці і без її використання було розроблено два бенчмарки. Вони виконують схожу

роботу, окрім запуску роботи самого алгоритму. Перший бенчмарк виконує нечіткий пошук із використанням таблиці подібності, а інший – без її використання. Для прикладу було розглянуто, яким чином працюють дані бенчмарки, для тесту на основі *BenchmarkWithTable*.

У даному бенчмарку відбувається ітерування за об’єктом *State*, в якому зберігаються конфігурації середовища, де будуть виконані бенчмарки. Тобто ітерування відбувається за різними можливими параметрами середовища, в цьому випадку це різні пошукові запити та різні тексти, в яких проводиться пошук. Далі ітерування відбувається за тестовими словами, порівнюючи їх із заданим пошуковим запитом, в результаті буде порахована відстань редагування та визначена кількість слів, в яких дана відстань буде меншою за порогову. *BenchmarkWithoutTable* буде відрізнятися лише тим, що для розрахунку відстані редагування таблиця подібності символів не буде врахована. Фрагмент такого результату наведено на рис. 3.8.

```
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 3.8 – Фрагмент результату порівняння швидкодії алгоритмів

Для зручності представлення та порівняння результатів тестування швидкодії та точності самих алгоритмів зобразимо їх, використовуючи графіки. Для цього результати тестування були збережені в JSON файл, а потім сформовані у вигляді графіків та гістограм за використання засобів мови Python і бібліотеки matplotlib. Наступним кроком тестування є тести кількості отриманих результатів з алгоритмами нечіткого пошуку із використанням таблиці подібності, без використання таблиці та алгоритму чіткого пошуку точного співпадіння [104].

Розглянемо приклад: нехай пошуковою фразою буде слово *“Každý”*, а словом із текстового набору даних буде *“kazdy”*, тоді треба знайти відстань редагування між цими словами класичним алгоритмом Дамерау-Левенштейна і модифікованим, результати роботи зображені на рис. 3.9 та рис. 3.10, відповідно.

		K	a	ž	d	ý
	0	1	2	3	4	5
k	1	1	2	3	4	5
a	2	2	1	2	3	4
z	3	3	2	3	3	4
d	4	4	3	3	2	3
y	5	5	4	4	3	3

Рисунок 3.9 – Підрахунок відстані редагування класичним алгоритмом

		K	a	ž	d	ý
	0	1	2	3	4	5
k	1	0.25	1.25	2.25	3.25	4.25
a	2	1.25	0.25	1.25	2.25	3.25
z	3	2.25	1.25	0.45	1.45	2.45
d	4	3.25	2.25	1.45	0.45	1.45
y	5	4.25	3.25	2.45	1.45	0.65

Рисунок 3.10 – Підрахунок відстані редагування модифікованим алгоритмом

У прикладі наочно видно, що модифікований алгоритм знайде набагато більше релевантних результатів шляхом використання таблиці подібності символів. Якщо вважати максимально допустиму відстань редагування як половину пошукового слова округлену вниз, у наведеному прикладі це буде 2, то без використання таблиці подібності знайдене слово T не буде підходити [125].

Для тестування швидкодії роботи алгоритмів нечіткого пошуку із використанням таблиці і без було розроблено два тести:

1. Перевірка різних пошукових запитів.

У даному тесті було підготовано текстовий набір, який перекладений на три різні мови, а саме: англійську, німецьку та чеську. Також було обрано по три пошукові запити на кожен переклад. Кожен пошуковий запит був шість разів модифікований. Для прикладу, чеський пошуковий запит був модифікований таким чином, щоб були використані тільки літери з англійського алфавіту, рис. 3.11. Далі проводився нечіткий пошук із використанням таблиці і без для кожного запиту та відповідного тексту.

```
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 3.11 – Фрагмент лістингу коду для тестування швидкодії алгоритму

2. Поступова зміна алфавіту.

Основним завданням цього тесту є перевірка, як будуть працювати алгоритми за умови, якщо пошуковий запит написаний однією мовою, а текст іншою.

В даному тесті потрібно виконати наступні кроки:

- Зчитати всі слова з тестового файлу за допомогою функції *getWords()*. Серед цих тестових слів буде відбуватись пошук.
- Вибрати випадковим чином серед тестових даних слово, яке буде вважатись пошуковим запитом. За цю операцію відповідає функція *getRandomWord()*.

- Створити список *shuffledAlphabet*, який буде містити англійський алфавіт та буде випадковим чином перемішаний. Цю операцію виконує функція *getShufflAlphabet()*.
- Створити цикл, в якому буде ітерування за кількістю змінених символів алфавіту від 1 до 10.
  - У циклі потрібно обрати певний символ із *shuffledAlphabet*, для нього, за допомогою функції getRandomSimChar(), випадковим чином вибрати семантично подібний символ із таблиці подібності. Далі в тестових словах замінити всі входження символу на відповідний йому.
  - Створити вкладений цикл, в якому буде ітерування за різними пороговими значеннями подібності для алгоритму нечіткого пошуку. Цей поріг визначає, яка максимальна відстань редагування може бути між двома словами в конкретному тесті. Ітерування відбувається від 0 до 3.5 з кроком 0.5. У даному циклі запускаються бенчмарки *BenchmarkWithTable* і *BenchmarkWithoutTable*, які й будуть тестувати швидкодію в заданому середовищі.

Для тестування правильності роботи таблиці подібності, заздалегідь потрібно прописати значення подібності для всіх пар символів, які тестуються. Для тестування підрахунку відстані редагування також було розроблено тест, який перевіряє роботу алгоритму Дамерау-Левенштейна з таблицею подібності. У даному тесті важливо перевірити можливі випадки пар слів для обчислення їхньої відстані редагування [103].

Для перевірки коректності роботи функції всі розрахунки відстані Дамерау-Левенштейна було проведено вручну та звірено з результатами роботи функції. Для прикладу пара слів “anthropology” та “anthopolögy”. У другому слові є дві помилки: перша це пропуск символу “r”, друга – заміна літери “o” на схожу літеру з іншої мови. У результаті, вартість перетворення другого слова на перше дорівнює 1.25, що і

повинна повернути функція. Для цих тестів, в ролі вхідних даних, було обрано книгу “Гаррі Поттер і філософський камінь” у перекладі на три різні мови: англійську, німецьку та чеську. А також, в ролі вхідних тестових даних було обрано файл розміром 370 000 різних випадкових слів. Результати тестів відображені у вигляді гістограм, в яких виконується нечіткий пошук для різних пошукових запитів та різних мов, та зображені на рис. 3.12 – рис. 3.20.

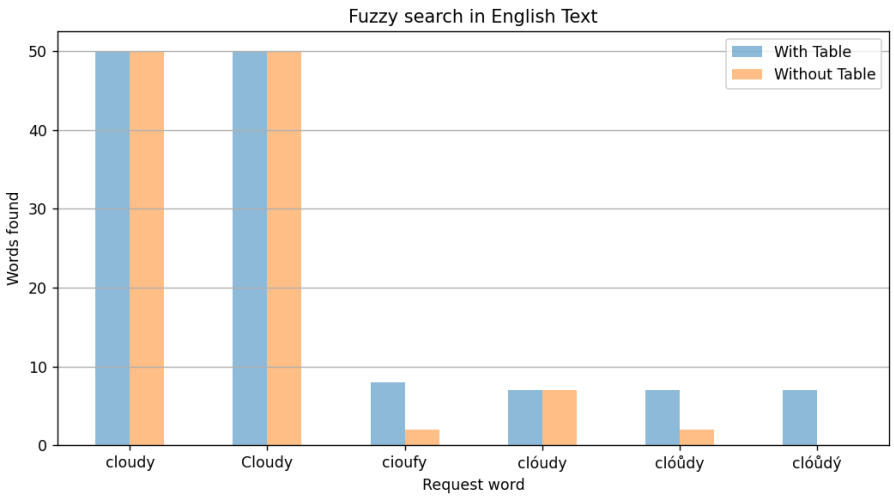

Рисунок 3.12 – Гістограма результатів пошуку в англійському тексті, пошуковий запит: cloudy

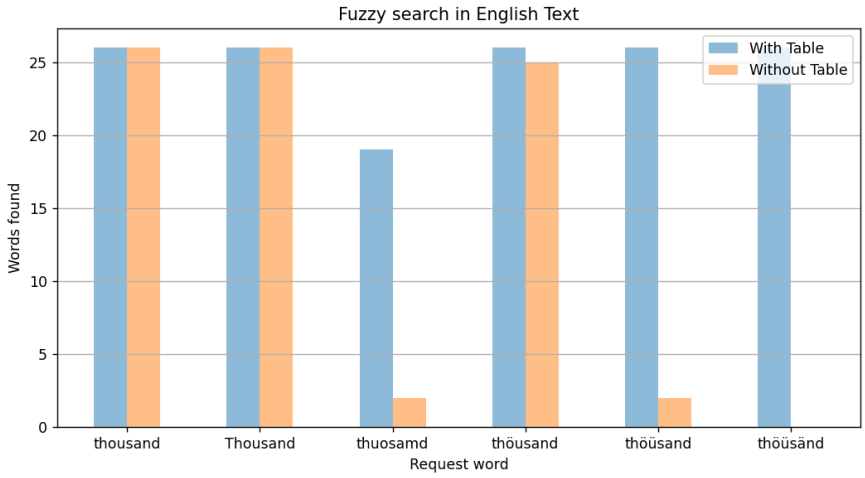

Рисунок 3.13 – Гістограма результатів пошуку в англійському тексті, пошуковий запит: thousand

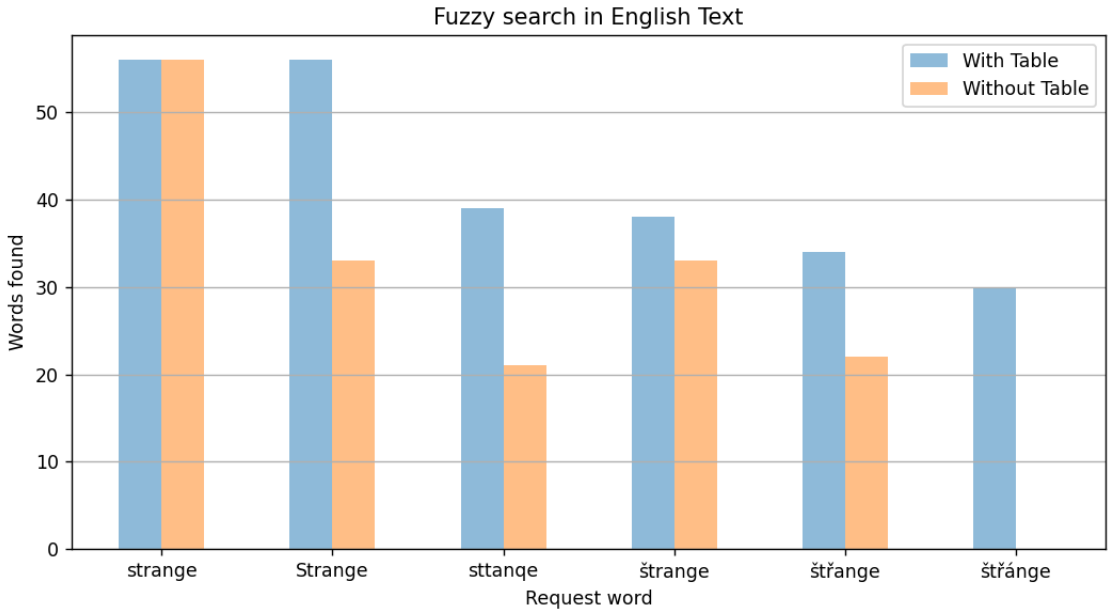

Рисунок 3.14 – Гістограма результатів пошуку в англійському тексті, пошуковий запит: strange

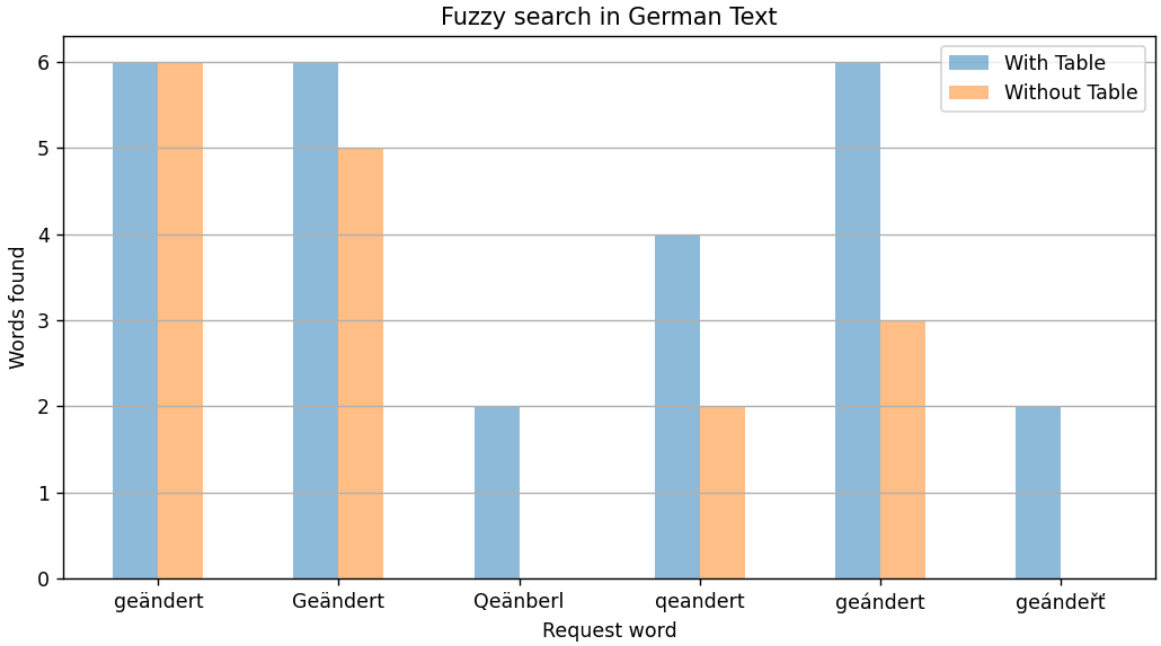

Рисунок 3.15 – Гістограма результатів пошуку в німецькому тексті, пошуковий запит: geändert

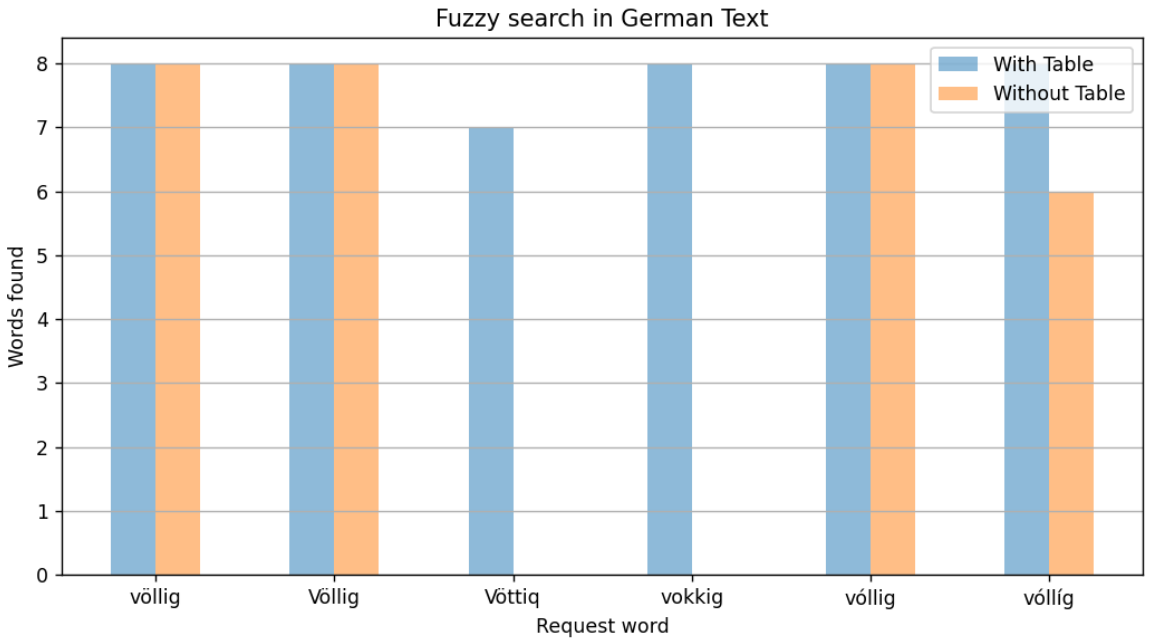

Рисунок 3.16 – Гістограма результатів пошуку в німецькому тексті, пошуковий запит: völlig

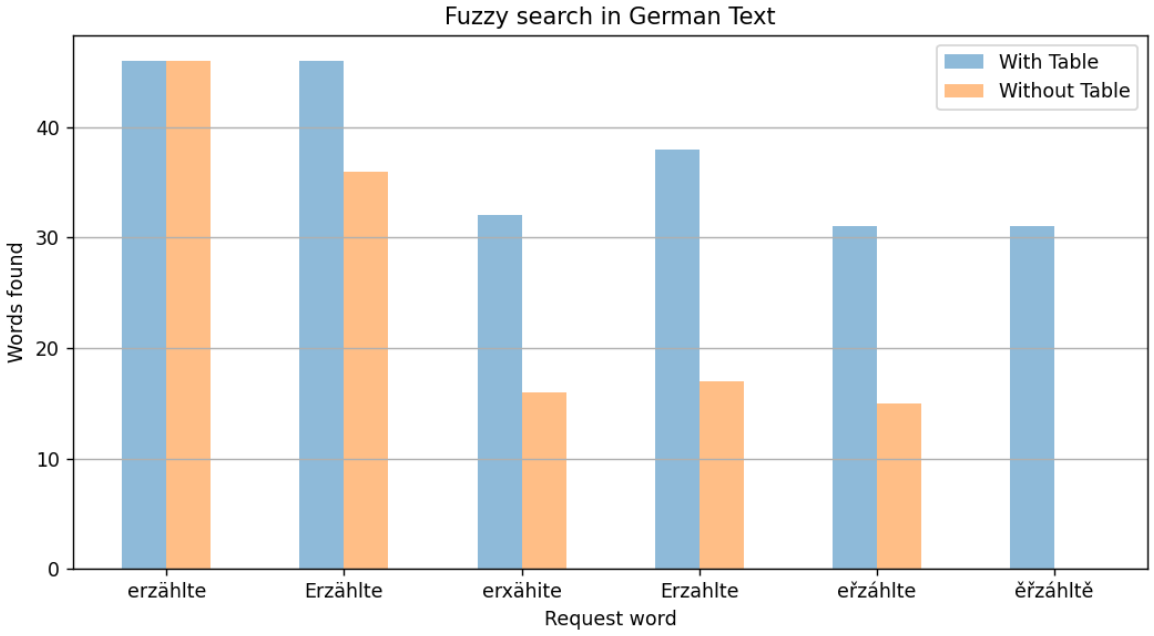

Рисунок 3.17 – Гістограма результатів пошуку в німецькому тексті, пошуковий запит: erzählte

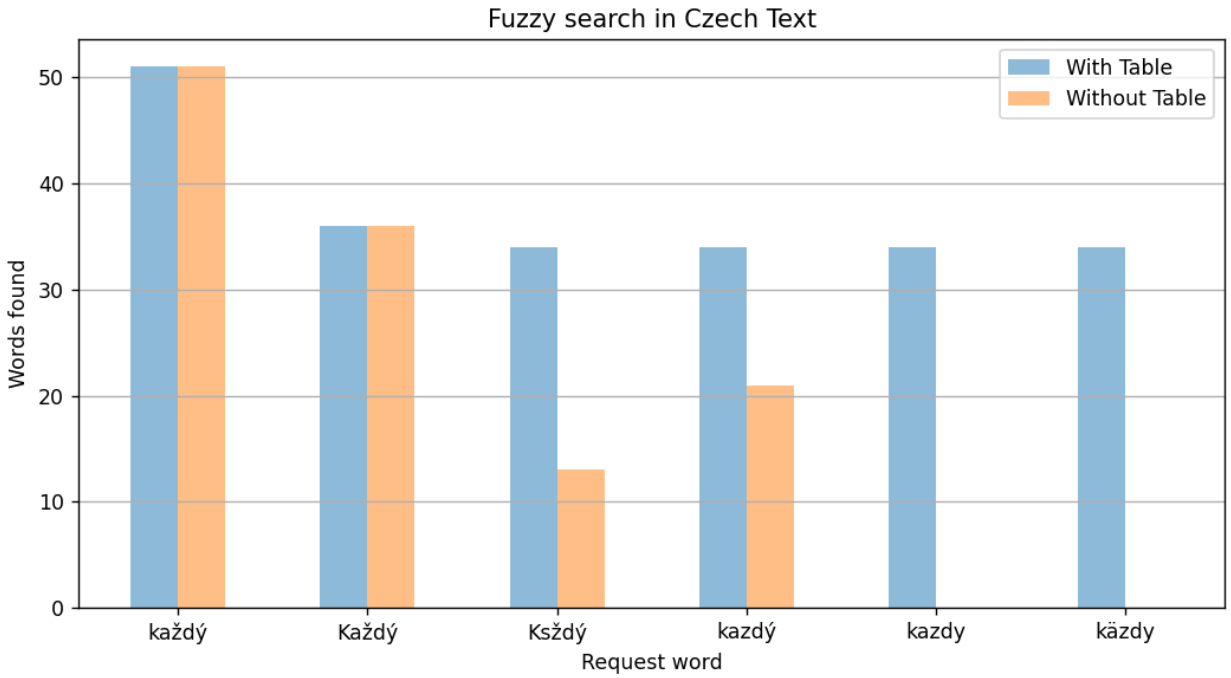

Рисунок 3.18 – Гістограма результатів пошуку в чеському тексті, пошуковий запит: každý

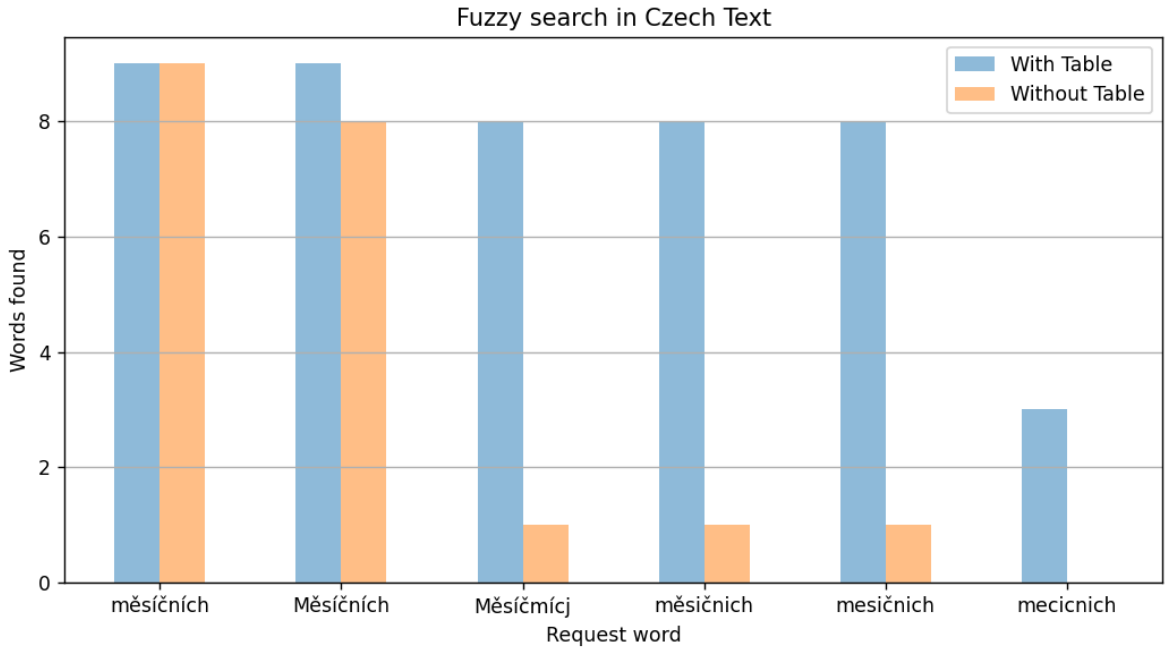

Рисунок 3.19 – Гістограма результатів пошуку в чеському тексті, пошуковий запит: měsíčních

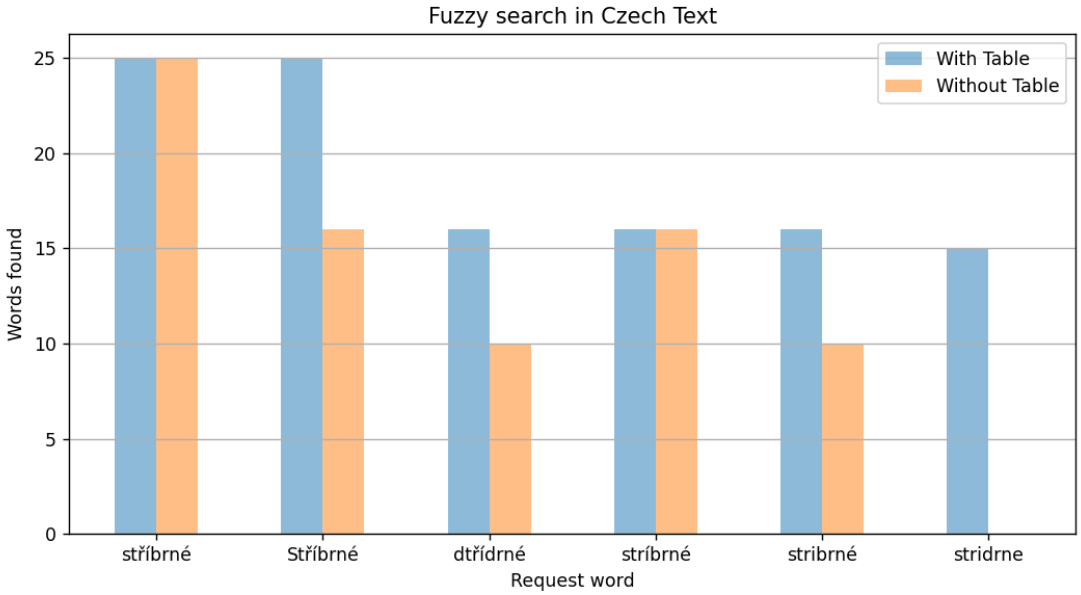

Рисунок 3.20 – Гістограма результатів пошуку в чеському тексті, пошуковий запит: stříbrné

По осі ікс, у даних гістограмах зображені різні пошукові запити, а по осі у – кількість знайдених слів. Аналізуючи дані гістограми, можна побачити, що для слова, яке не має жодної модифікації або написане без помилок, обидва алгоритми показують однаковий результат. Проте, коли пошуковий запит змінюється чи користувач допускає більше помилок, нечіткий пошук із використанням таблиці показує значно кращий результат, ніж алгоритм без таблиці. Також можна стверджувати, що за умови збільшення кількості змінених символів у слові, зменшується кількість слів, які були знайдені [103].

На рис. 3.21 та рис. 3.22 наведено графіки тестів, в яких алфавіт поступово змінюється. По x-осі на даних графіках представлена кількість змінених символів, а по осі y – час пошуку. У кожному окремому графіку порівнюється швидкодія з різним порогом пошуку від 0 до 3.5. Варто зауважити, що нечіткий пошук із використанням таблиці подібності в середньому виконується повільніше на 5%. Проте це компенсується більшою кількістю отриманих результатів. Порівнюючи кількість знайдених слів, можна зробити висновок, що алгоритм із використанням таблиці

подібності показує кращі результати. Чим більший поріг відстані редагування, тим більшу кількість слів було знайдено.

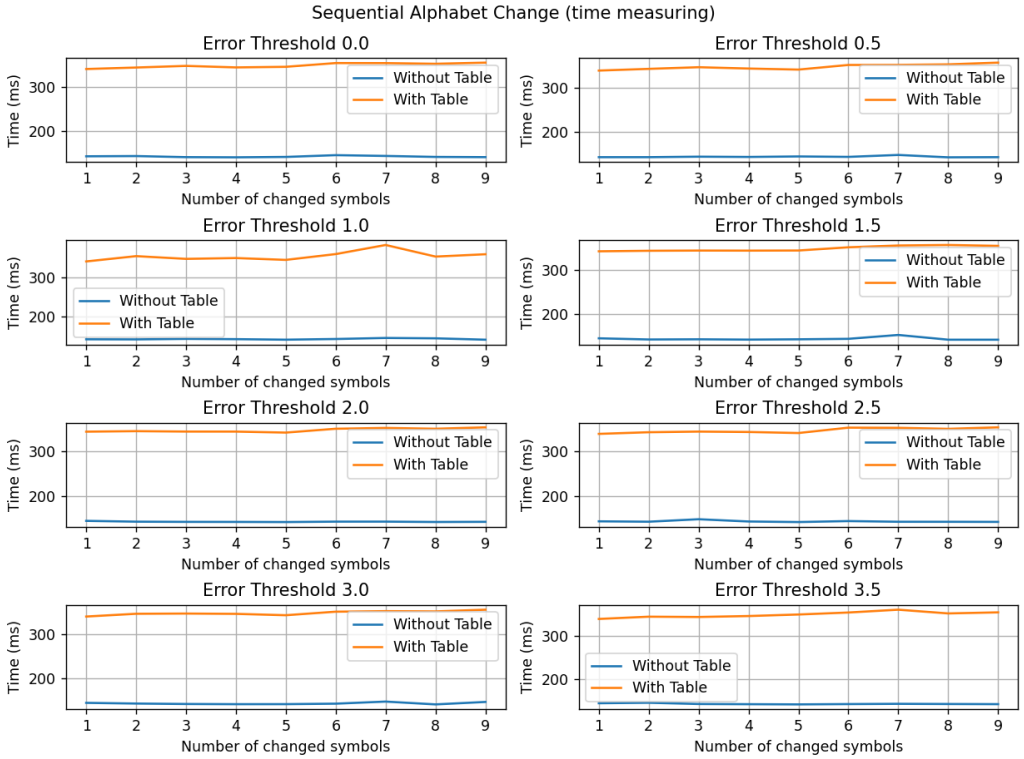

Рисунок 3.21 – Графік порівняння швидкодії зі зміною алфавіту

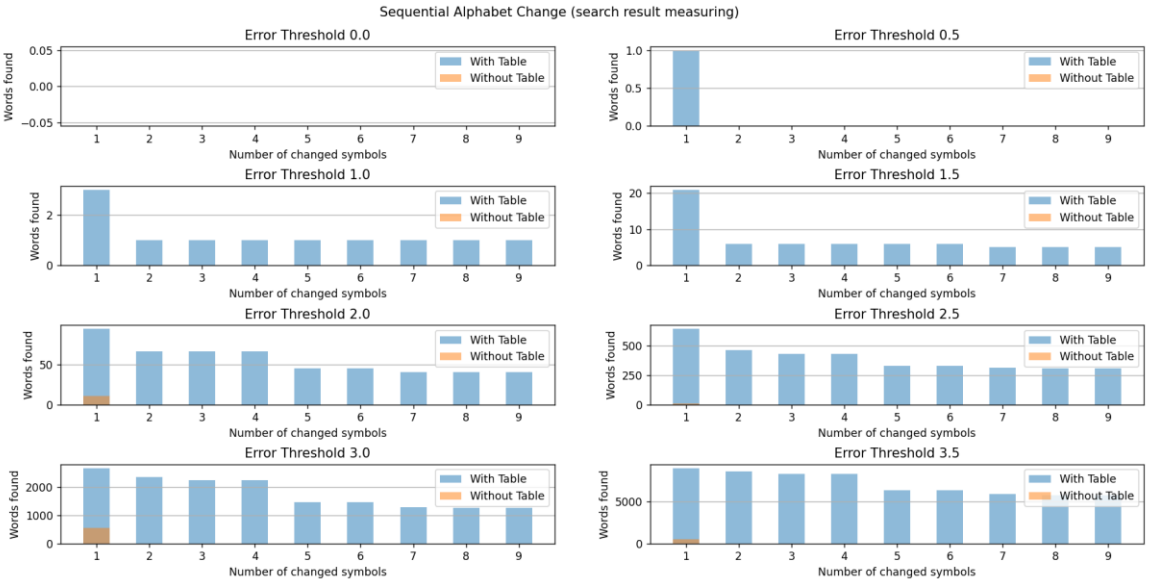

Рисунок 3.22 – Гістограма порівняння результатів зі зміною алфавіту

Як фінальний тест була перевірена кількість результатів, які повертають різні алгоритми пошуку: Дамерау-Левенштейна з таблицею, Дамерау-Левенштейна без таблиці і звичайний чіткий алгоритм пошуку. Для цього було підготовано великий набір текстових даних, перекладений на 3 різні мови, а саме: англійську, німецьку та чеську. У тесті необхідно дивитись на кількість результатів для різних пошукових слів із різними типами помилок. Результати такого тестування зображено для чеської мови на рис. 3.23, для німецької мови – на рис. 3.24, та для англійської мови – на рис. 3.25.

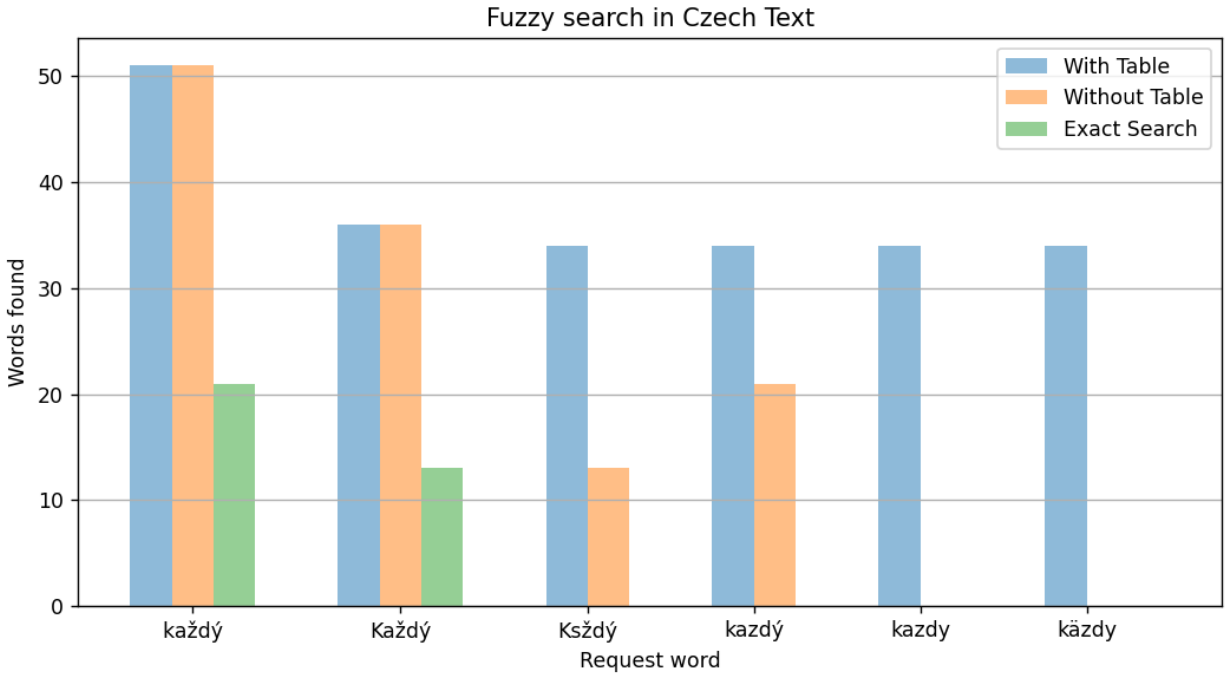

Рисунок 3.23– Результати фінального тестування для чеської мови

Тест швидкодії показав, що використання таблиці подібності зменшує продуктивність алгоритму на 10-20%, в залежності від довжини пошукового слова. Це пояснюється тим, що необхідно робити додаткові запити для таблиці подібності для кожної пари символів. Проте значно більша кількість знайдених релевантних результатів компенсує цей недолік [104].

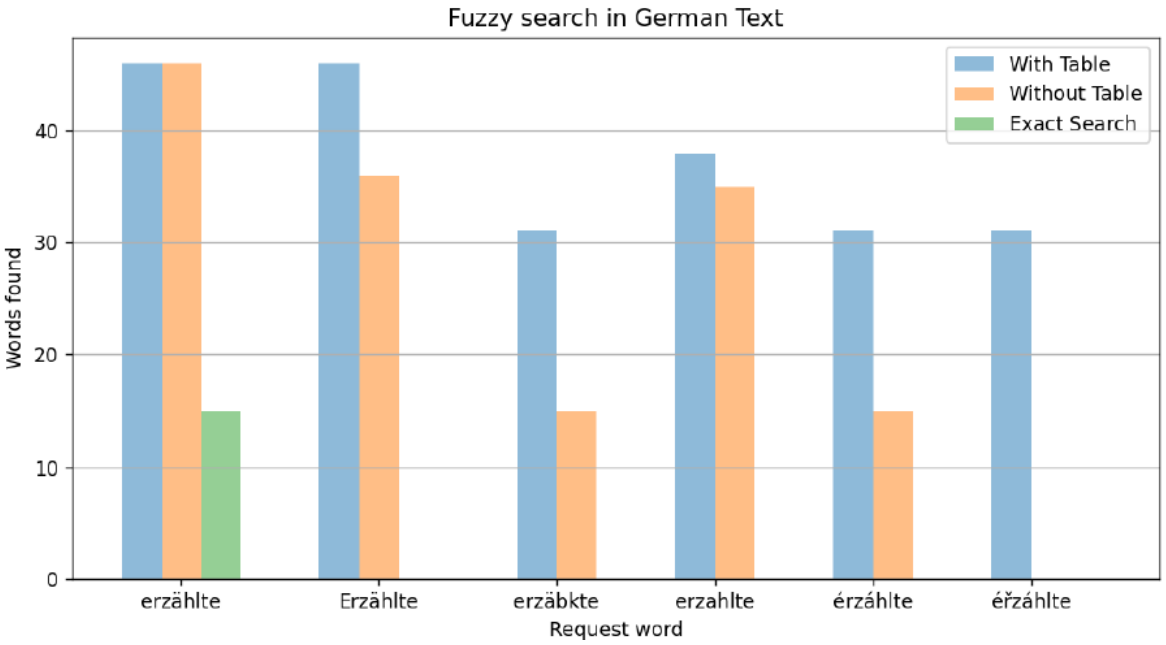

Рисунок 3.24– Результати фінального тестування для німецької мови

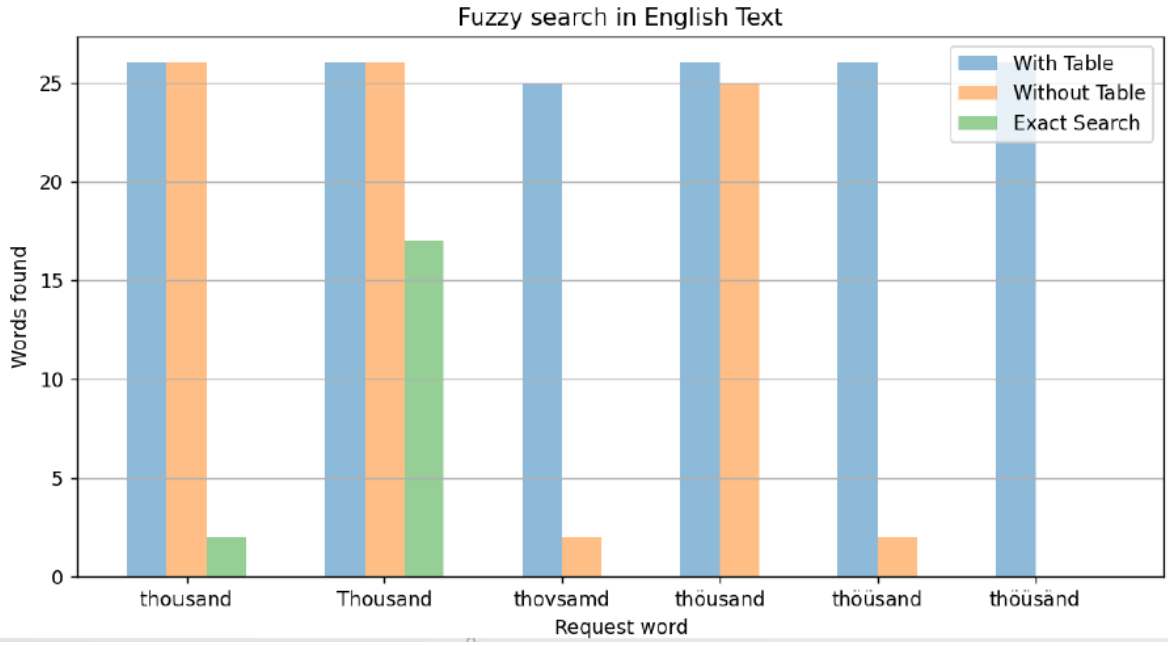

Рисунок 3.25– Результати фінального тестування для англійської мови

3.3 Реалізація системи нечіткого пошуку

Після проходження 6-ти кроків запропонованого методу, на сьомому кроці необхідно кожному слову із шаблона поставити у відповідність кожне слово з тексту, на рис. 3.26 слова із шаблону позначені синім кольором, а слова з тексту – червоним.

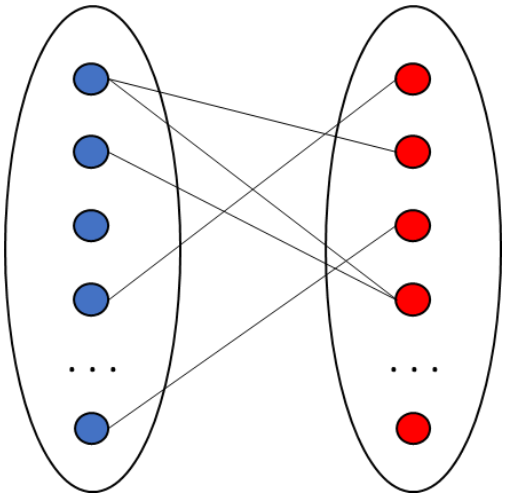

Рисунок 3.26 – Пошук відповідностей між шаблоном і текстом

Введемо метрику подібності слів: P_{word}(a, b) = 1 – d / max(len(a), len(b)), де d – модифікована відстань Дамерау-Левенштейна, len(x) – довжина слова x. Для однакових слів метрика покаже 1, для повністю різних – 0.

Для кожного слова з пошукової фрази рахуємо метрику P_{word} із кожним словом із тексту. Можливі ситуації:

- для поточного слова немає відповідності в тексті, відповідно, поточне слово не знайдено;
- для поточного слова є однозначна відповідність у тексті, відповідно, поточне слово знайдено, слово в тексті промарковане як використане;
- для поточного слова є декілька відповідностей у тексті, відповідно, поточне слово знайдено, слово в тексті з більшою метрикою P_{word} промарковане як використане [125].

Введемо метрику подібності фраз. Варто зауважити: ситуація, коли до пошукового слова з пошукової фрази немає відповідності в тексті, набагато гірша за ситуацію, коли не всі слова з тексту промарковані як використані [9].

$P_{phrase} = \frac{(\sum_{Sword} P_{word} * len(word))}{len(phrase)} - \frac{(\sum_{Tword}(1 - P_{word}) * len(word))}{len(text) * tFactor} * penaltyCoef$, де tFactor та penaltyCoef – балансуючі коефіцієнти. Метрика може набути значень від 0 до 1, де 1 – найкраще співпадіння, а 0 – найгірше.

У випадку завеликого пенальті, метрика може стати навіть від'ємною, в цьому випадку вважаємо її рівною нулю.

Введемо метрику релевантності фраз: $R_{phrase} = (1 - P_{phrase})$ * maxScore, метрика може набувати значень від 0 – найкраще співпадіння, до maxScore – найгірше співпадіння, в роботі пропонується значення maxScore = 100'000.

Для покращення точності розрахунку потрібно ввести модуль стоп-слів, детально розглянутий у розділі 2.3.3. Цей модуль буде ігнорувати певні слова в довгій пошуковій фразі та в довгому тексті. Наприклад, слова з англійської мови, які не мають сенсу для пошуку: "a", "the", "in", "at", "by", "to", проте не можна ігнорувати наступні короткі слова: "my", "you", "no", "yes".

3.3.1 Метод експертного оцінювання

На основі запропонованих метрик необхідно порахувати релевантність кожного документа до пошукової фрази, а потім відсортувати і відфільтрувати лише ті, які мають значення метрики менше порогового значення. Для підбору балансуючих коефіцієнтів використовувався метод експертної оцінки [125].

Метод експертного оцінювання – підхід до оцінювання та прийняття рішень, який використовує експертні знання та досвід для отримання обґрунтованих висновків. Цей метод широко застосовується в різних галузях, включаючи науку, технології, бізнес, управління та інші. Основні принципи методу експертного оцінювання включають такі елементи, як: використання експертних знань,

систематизацію та аналіз, врахування невизначеності та застосування статистичних методів. Деякі випадки використання статистичних методів можуть бути корисними для аналізу результатів та визначення ступеня впевненості у висновках. Експерти, які мають досвід у відповідній галузі, надають свої оцінки та рекомендації щодо питань, які вивчаються. Оцінки експертів систематизуються та аналізуються для отримання усереднених результатів та визначення ключових висновків [126]. Метод експертного оцінювання дозволяє отримати вагомий внесок у складних ситуаціях, де немає однозначних відповідей, а також допомагає приймати обґрунтовані рішення на основі професійного досвіду та знань [104].

Декільком експертам-користувачам пропонувався власноруч створений набір тестів, де кожен тест складався з пошукової фрази (searchPhrase) та набору текстових даних (testPhraseVector). Їхньою задачею було розташувати набір текстів у порядку зменшення релевантності до пошукової фрази, відкинувши нерелевантні, на їхню думку, екземпляри. На основі думки експертів, було сформовано порядок фраз для кожного тесту, який вважається правдивим (groundTruth). Система балансувалась таким чином, щоб для всіх тестів отримувати результат тотожний правдивому. Приклад такого тесту зображено на рис. 3. 27.

```
searchPhrase = "Ostemad";
testPhraseVector = {
    "Jeg har lyst til en ostemad",
    "Der er for få ostemadder i min tærte",
    "Mit luftskib er fuldt af ål",
    "ostemad", "Ostemad", "Østemåd",
    "THORBJØRN", "Ostemad med sennep",
    "Ostemad. 27,00 DKK. Ostemad antal." };

groundTruth = {
    "Ostemad", "ostemad", "Østemåd",
    "Ostemad. 27,00 DKK. Ostemad antal.",
    "Ostemad med sennep", "Jeg har lyst til en ostemad",
    "Der er for få ostemadder i min tærte" };
```

Рисунок 3.27 – Приклад тесту на основі експертної оцінки

На вхід пошуковій системі надходить слово-шаблон та набір текстових даних. Система визначає найбільш релевантні пошукові результати, сортує їх у порядку зменшення релевантності та відкидає непідходящі результати [104].

3.3.2 Результат роботи системи нечіткого пошуку

Для прикладу розглянемо пошукову фразу “His eyes functioned fine” і набір вхідних текстових даних:

– “human eye”
– “cognitive functions”
– “his book”
– “fine-tuning”
– “the eyes”
– “fine movements”
– “eye functions”
– “absent or non-functioning”
– “eye to see fine detail”
– “his spiritual eyes”

Результат роботи системи для таких вхідних наведено на рис. 3.28, відповідно до цього, вище розташовані найбільш релевантні результати, які мають більше співпадінь.

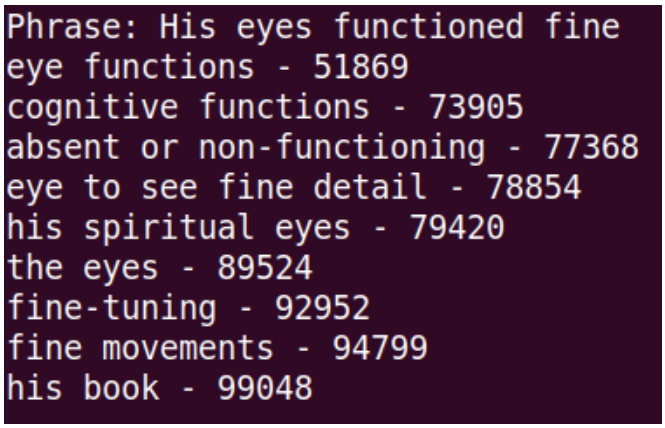

Рисунок 3.28 – Результат роботи системи

Як другий приклад роботи системи розглянемо пошукову фразу “Pingo”, та набір вхідних тестових даних, який складається з наступних десяти фраз:

– “Skingo”
– “Flamingo”
– “Ringo”
– “Ostemad”
– “pngo”
– “pingi”
– “pingen”
– “Pingen”
– “ring”
– “street”

Результат пошуку зображено на рис. 3.29.

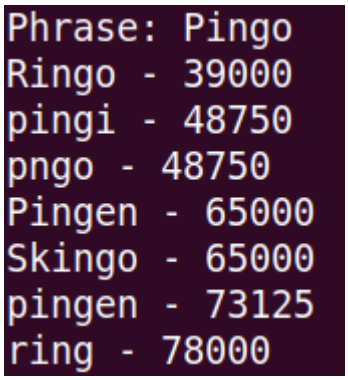

Рисунок 3.29 –Приклад роботи системи

Наприкінці результати потрібно відсортувати та відфільтрувати згідно значень фінальної метрики релевантності. Як алгоритм сортування пропонується стандартний алгоритм бібліотеки STL std::sort().

Додаток для зберігання резервних копій є важливим інструментом для підтримання безпеки та захисту даних користувача. Такі додатки дозволяють автоматично або вручну створювати та зберігати копії важливих файлів і папок, що дозволяє відновлювати дані у разі втрати або пошкодження оригінальних файлів.

Додатки для зберігання резервних копій можуть бути розроблені для різних платформ (Windows, macOS, Linux, мобільні операційні системи) та мати різні функції, в залежності від потреб користувача. Запропонована система була успішно реалізована мовою програмування C++ та впроваджена у веб-додаток для зберігання резервних копій даних. Завдяки системі, середній час пошуку релевантних результатів зменшився на 5 %, проте для пошукових фраз різної довжини та різної кількості слів у фразі, результат може відрізнятись. Чим довше пошукова фраза, тим менший виграш у часі. Кількість релевантних результатів зросла на 10 %, проте ця оцінка дуже залежить від вхідних текстових даних та частоти невимушених орфографічних помилок у користувача. Приклади роботи такого додатку зображені на рис. 3.30 та 3.31.

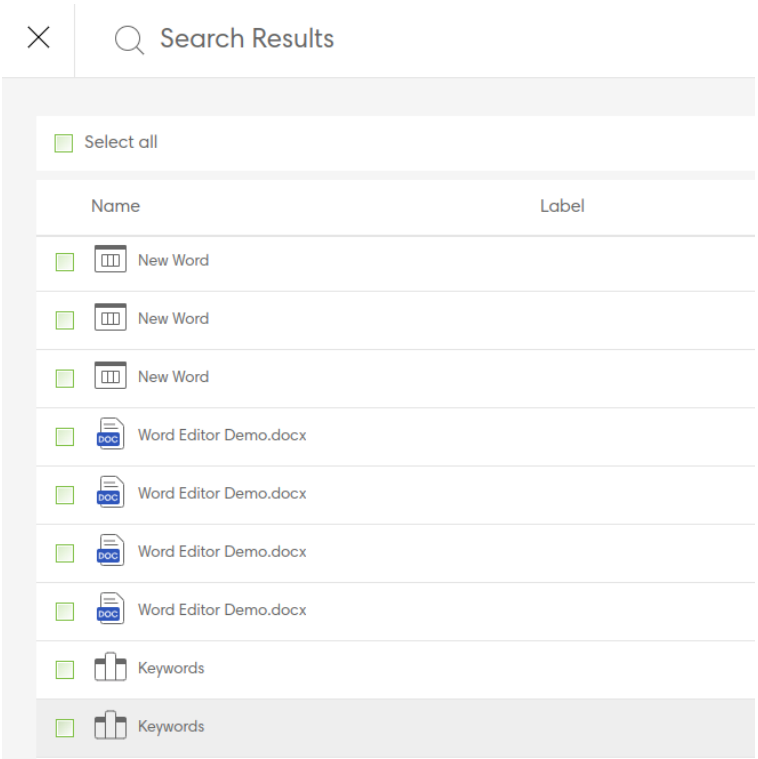

Рисунок 3.30 – Приклад роботи додатку для слова “word”

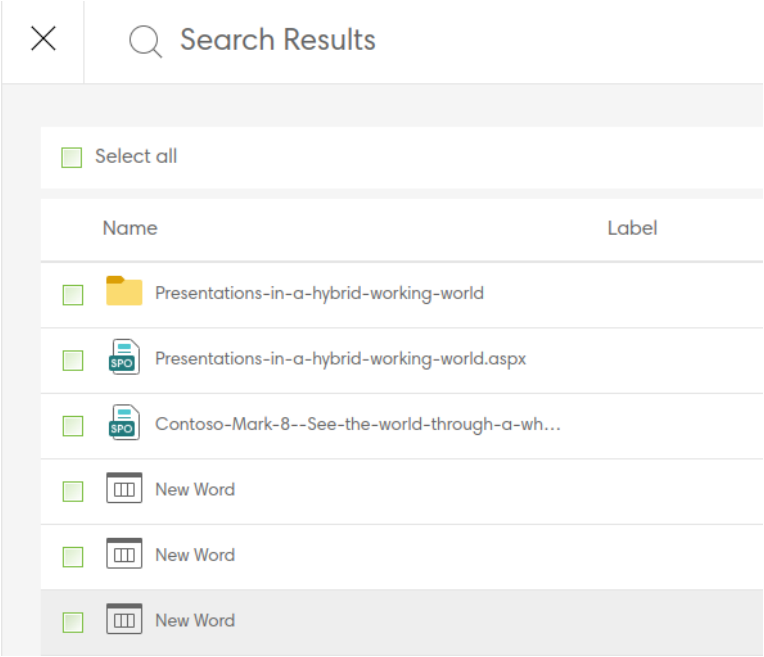

Рисунок 3.31 – Приклад роботи додатку для слова “world”

3.4 Висновки до розділу

У цьому розділі було проведено порівняння алгоритмів нечіткого пошуку на основі відстані Дамерау-Левенштейна. Між собою порівнювались звичайний алгоритм Дамерау-Левенштейна та 3 різні алгоритми на основі скінченних автоматів. Спочатку було розроблено та проаналізовано декілька варіантів програмної реалізації, включаючи автомат на основі префіксного дерева, автомат на основі хешування та автомат на основі таблиці переходів. Проведено декілька тестів для перевірки коректності реалізації та оцінки продуктивності, що включають: побудову автомата, перевірку слова та загальне тестування, що перевіряє швидкодію автоматів в порівнянні до тривіального підходу.

На основі аналізу проведених тестів, було виявлено, що всі розроблені рішення на основі скінчених автоматів працюють швидше, ніж тривіальний підхід з використанням алгоритму Дамерау-Левенштейна для пошукових текстових даних об’ємом більше за 5000 слів. Це пояснюється тим, що додатковий час витрачається на побудову автоматів, а звичайний підхід одразу починає пошук слів уникаючи

побудови. На словнику з 370000 англомовних слів та максимальних відстанях редагування від 1 до 3 кожен автомат показав результати швидші приблизно в 5-9 разів.

HashAutomaton рекомендується для загального використання, оскільки об’єм затраченої пам’яті значно менший для великих значень редагувальної відстані, а швидкодія перевірки слів майже не відрізняється від TreeAutomaton. Цей автомат виявився найбільш універсальним для розв’язання поточної задачі.

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

TableAutomaton рекомендується для випадків, коли пошукове слово часто змінюється, оскільки цей тип автомата не потребує побудови та детермінізації для шаблону та невеликих значень редагувальної відстані. Основним обмеженням швидкодії цього автомата є необхідність розрахунку характеристичного вектору для кожного вхідного символу, що може бути прискорене на 5-10% за допомогою використання SIMD інструкцій для його обчислення. Проте це рішення не є універсальним, оскільки потребує використання центрального процесору з підтримкою таких інструкцій.

Також було модифіковано алгоритм Дамерау-Левенштейна, додавши до розрахунку відстані редагування між словами, можливість отримати міру схожості символів з таблиці подібності. Для реалізації такого підходу було створено саму таблицю подібності та модифіковано алгоритм Дамерау-Левенштейна.

В ролі фінального тестування між собою порівнювались 3 різні алгоритми пошуку з точки зору швидкодії та кількості знайдених релевантних результатів. А саме: алгоритм Дамерау-Левенштейна з таблицею подібності, без таблиці та класичний алгоритм чіткого пошуку. Аналіз було здійснено на великому наборі тестових даних 3 різними мовами. Перший алгоритм показує значно кращі результати

нечіткого пошуку за знаходженням релевантних результатів, проте другий виконує пошук в середньому на 10-20% швидше. При цьому алгоритм чіткого пошуку показав низькі результати при внесенні певних помилок в пошукові слова чи в текст.

Загалом, нечіткий пошук з використанням таблиці подібності слід використовувати, коли для користувача важливіші результати пошуку. А пошук без таблиці слід використовувати, коли користувачу важлива швидкість обчислення. Водночас чіткий пошук найкраще підходить для задачі пошуку зразка в тексті при відсутності помилок різних типів. Для вибору найбільш підходящих документів, була запропонована метрика оцінки релевантності текстових даних відповідно до пошукової фрази. На основі тестування за допомогою експертного оцінювання розраховані оптимальні коефіцієнти у відповідних формулах. Також запропонована система була впроваджена в додаток резервного копіювання об’єктів.

ВИСНОВКИ

Жоден з існуючих алгоритмів нечіткого пошуку не враховує можливу семантичну подібність символів в словах. Таблиця подібності символів виявилась ефективним інстументом для вирішення цієї проблеми, фіксуючи можливі семантичні та інші види подібності між символами. Оцінюючи подібності на основі різних критеріїв, таких як: форма, контекст і фонетика, таблиця подібності символів дозволила модифікувати алгоритм нечіткого пошуку, який може ідентифікувати та ранжувати відповідні результати навіть за наявності помилок у шаблоні чи тексті, в якому відбувається пошук.

У рамках дослідження був модифікований алгоритм Дамерау-Левенштейна, додавши до розрахунку відстані редагування між словами, можливість отримати міру схожості символів з таблиці подібності. Для реалізації такого підходу було реалізовано саму таблицю подібності та створено модифікований алгоритм Дамерау-Левенштейна.

При проведенні тестування між собою порівнювались 3 різні алгоритми пошуку з точки зору швидкодії та кількості отриманих релевантних результатів. А саме: алгоритм Дамерау-Левенштейна з таблицею подібності, без таблиці та класичний алгоритм чіткого пошуку. Аналіз було здійснено на великому наборі тестових даних трьома різними мовами. Перший алгоритм показав значно кращі результати пошуку з точки зору знаходженням релевантних результатів, проте другий виконує пошук даних в середньому на 10-20% швидше. При цьому алгоритм чіткого пошуку показав низькі результати при внесенні певних типів помилок в пошукові слова чи в текст.

На основі проведених експериментів, можна дійти висновку, що нечіткий пошук з використанням таблиці подібності слід використовувати, коли для користувача більш важливі результати пошуку та їх релевантність. Пошук без таблиці слід використовувати, коли користувачу важлива швидкість обчислення. Водночас

чіткий пошук найкраще підходить для задачі пошуку зразка в тексті при відсутності помилок різних типів.

Було проведено порівняння алгоритмів нечіткого пошуку на основі відстані Дамерау-Левенштейна. Між собою порівнювався звичайний алгоритм Дамерау-Левенштейна та 3 алгоритми на основі скінченних автоматів. Було розроблено та проаналізовано декілька варіантів програмної реалізації, включаючи автомат на основі префіксного дерева, автомат на основі хешування та автомат на основі таблиці. Проведено тестування для перевірки коректності реалізації та оцінювання продуктивності, що включає: побудову автомата, перевірку слова та загальне тестування, що перевіряє швидкодію автоматів в порівнянні з тривіальним підходом. Спираючись на проведені тести та аналіз рішень можна зробити висновок, що всі розроблені рішення на основі скінченних автоматів працюють швидше, ніж простий підхід з використанням алгоритму Дамерау-Левенштейна для пошукових текстових даних більше за 5000 слів. Це пояснюється тим, що додатковий час витрачається на побудову автоматів, а стандартний підхід одразу починає пошук. На словнику з 370 тисяч слів та максимальних відстанях редагування від 1 до 3 кожен автомат показав результати швидші в 5–9 разів. HashAutomaton рекомендується для загального використання, оскільки об’єм використаної пам’яті, значно менший для великих значень редагувальної відстані, а швидкодія перевірки слів майже не відрізняється від TreeAutomaton, відповідно цей автомат виявився найбільш універсальним. TreeAutomaton рекомендується для випадків, коли потрібна різна вартість редагувальних операцій, або для відстані редагування в межах від 1 до 2. На малих значеннях редагувальної відстані автомат показав найбільшу швидкодію. TableAutomaton рекомендується для випадків, коли пошукове слово часто змінюється, оскільки автомат не потребує побудови та детермінізації для будь-якого шаблону та невеликих значень редагувальної відстані. Основним обмеженням швидкодії цього автомата є необхідність розрахунку характеристичного вектору для кожного вхідного

символу, що може бути прискорене на 5–10 % за допомогою використання SIMD інструкцій. Проте це рішення не є універсальним, оскільки потребує використання центрального процесору з підтримкою таких інструкцій.

Основні результати дослідження:

- Запропоновано метод нечіткого пошуку, який складається з 9 кроків і поєднує в собі переваги існуючих методів, які базуються на використанні скінченних детермінованих автоматів та методів динамічного програмування для пошуку відстані редагування. Основна ідея методу заключається в тому, що спочатку необхідно перевести пошукову фразу та вхідні текстові набори в їх базові аналоги, використавши таблицю подібності символів. За допомогою ДСА визначити ті слова з тексту, які мають відстань редагування до пошукових слів менше заданого значення. Лише для цих слів з тексту визначити уточнену відстань редагування за допомогою модифікованого алгоритму Дамерау-Левенштейна. За допомогою запропонованих метрик подібності слів, подібності фраз, релевантності фрази порахувати релевантність кожної фрази із текстового набору до шаблону й обрати з них найбільш відповідні.
- Запропоновано підхід до створення таблиці подібності символів, яка дозволить враховувати можливу семантичну подібність символів в словах. Оцінюючи подібності на основі різних критеріїв, таких як: форма, контекст і фонетика, таблиця подібності символів дозволила модифікувати алгоритм нечіткого пошуку, який може ранжувати відповідні результати навіть за наявності великої кількості неспівпадінь unicode значень схожих символів.
- Модифіковано алгоритм Дамерау-Левенштейна, додано до розрахунку відстані редагування між словами, можливість отримати міру схожості символів з таблиці подібності. Для реалізації такого підходу було створено саму таблицю подібності та реалізовано модифікований алгоритм Дамерау-Левенштейна.

- Досліджено різні алгоритми, які базуються на основі скінченних автоматів. А саме: TreeAutomaton, TableAutomaton, HashAutomaton, де перші два виявились неефективними через певні недоліки, а останній виявився оптимальним та найбільш універсальним з точки зору швидкості роботи та часу побудови, а також об’єму витраченої пам’яті.
- Запропоновано метрику оцінки релевантності текстових даних відповідно до пошукової фрази. На основі тестування за допомогою експертної оцінки, розраховані оптимальні коефіцієнти у відповідних формулах.
- Розроблена система впроваджена у веб-додаток для зберігання резервних копій даних.

Запропоновано метод нечіткого пошуку для знаходження найбільш релевантних документів відповідно до пошукової фрази. Основна ідея методу полягає в тому, що спочатку необхідно перевести пошукову фразу та вхідні текстові набори в їхні базові аналоги за допомогою таблиці подібності символів. За допомогою ДСА визначити ті слова з тексту, які мають відстань редагування до шаблону менше заданого значення. Лише для цих слів з тексту визначити уточнену відстань редагування за допомогою модифікованого алгоритму Дамерау-Левенштейна.

Для реалізації декількох кроків методу було запропоновано підхід до створення таблиці подібності символів, яка дозволить враховувати можливу семантичну подібність символів в словах. Оцінюючи подібність на основі різних критеріїв, таких як: форма, контекст і фонетика, таблиця подібності символів дозволила модифікувати алгоритм нечіткого пошуку, який може ранжувати відповідні результати навіть за наявності великої кількості неспівпадінь unicode значень схожих символів. Це, в свою чергу, збільшило кількість релевантних результатів на 10 %, в залежності від довжини пошукової фрази.

Як перспектива 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. Або спробувати застосувати комбінацію методів на основі алгоритмів Дамерау-Левенштейна та методів машинного навчання. Одним з загальних підходів є використання моделей класифікації або регресії для оцінки релевантності документів до пошукового запиту.

Такий підхід дозволить автоматизувати процес пошуку релевантних документів та покращити точність в порівнянні з традиційними методами, такими як ключові слова або фільтрація.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Gonzalo N. A guided tour to approximate string matching / Navarro Gonzalo. Association for Computing Machinery. – 2001.

[2] LOTFI A. ZADEH (1990) FUZZY SETS AND SYSTEMS, International Journal of General Systems, 17:2-3, 129-138, DOI: 10.1080/03081079008935104

[3] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008.

[4] Vatsalan, Dinusha & Sehili, Ziad & Christen, Peter & Rahm, Erhard. (2017). Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges. DOI: 10.1007/978-3-319-49340-4_25.

[5] Awekar, Amit & Samatova, Nagiza. (2009). Fast Matching for All Pairs Similarity Search. 295-300. DOI: 10.1109/WI-IAT.2009.52.

[6] Dominik M., Endres and Johannes E., Schindelin. A new metric for probability distributions. IEEE Trans. Inform. Theory, 49(7):1858–1860, 2003.

[7] Liu, L., Zhang, L., Wang, X., & Ma, S. (2022). "A Deep Learning Approach for Document Relevance Ranking." Information Retrieval Journal, 1-24.

[8] Zeng, Q., Zhang, J., & Lu, X. (2021). "A Query-Focused Multi-View Learning Approach for Document Relevance Ranking." Knowledge-Based Systems, 235, 106842.

[9] Yu, L., Wu, F., Wang, X., & Chen, T. (2021). "Neural Information Retrieval: A Literature Review." arXiv preprint arXiv:2107.03232.

[10] Zhao, X., Liu, Q., & Yang, Q. (2021). "Interactive Deep Learning for Document Relevance Ranking." Information Sciences, 561, 155-169.

[11] Zhang, Y., Zhang, J., Liu, Y., & Gao, M. (2020). "Efficient and Effective Document Relevance Ranking with Dual-Level Attention Based Convolutional Neural Networks." Information Processing & Management, 57(2), 102067.

[12] Schulz K. Fast string correction with Levenshtein automata / K. Schulz, S.

Mihov. // International Journal on Document Analysis and Recognition. – 2002.

[13] Boytsov L. Indexing methods for approximate dictionary searching: Comparative analysis / Leonid Boytsov. // Journal of Experimental Algorithmics. – 2011.

[14] Damerau–Levenshtein distance https://www.geeksforgeeks.org/damerau-levenshtein-distance/

[15] Snášel, V., Keprt, A., Abraham, A., Hassanien, A.E. (2009). Approximate String Matching by Fuzzy Automata. In: Cyran, K.A., Kozielski, S., Peters, J.F., Stańczyk, U., Wakulicz-Deja, A. (eds) Man-Machine Interactions. Advances in Intelligent and Soft Computing, vol 59. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-00563-3_29

[16] Baeza-Yates R. A faster algorithm for approximate string matching / R. Baeza-Yates, G. Navarro. // Springer, Berlin, Heidelberg. – 1996.

[17] Dr. V. Ramaswamy, Girijamma. H. A., An extension of Myhill Nerode theorem for fuzzy automata, Advances in Fuzzy Mathematics, Research India Publications, ISSN 0973-533X Volume 4, Number 1 (2009).

[18] Deformed fuzzy automata for correcting imperfect strings of fuzzy symbols, J.R. Garitagoitia; J.R.G. de Mendivil; J. Echanobe; J.J. Astrain; F. Farina https://doi.org/10.1109/TFUZZ.2003.812682

[19] Introduction to Algorithms, fourth edition / [T. Cormen, C. Leiserson, R. Rivest та ін.]., 2022. – 1312 с.

[20] Mihov S. Fast Approximate Search in Large Dictionaries / S. Mihov, K. Schulz. // Computational Linguistics. – 2004.

[21] J. P. Carvalho and L. Coheur, "Introducing UWS - A fuzzy based word similarity function with good discrimination capability: Preliminary results," *2013 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, Hyderabad, India, 2013, pp. 1-8.

[22] Yu, M., Li, G., Deng, D. et al. String similarity search and join: a survey. Front. Comput. Sci. 10, 399–417 (2016).

[23] J. Wang, G. Li and J. Fe, "Fast-join: An efficient method for fuzzy token matching based string similarity join," 2011 IEEE 27th International Conference on Data Engineering, Hannover, Germany, 2019, pp. 458-469, doi: 10.1109/ICDE.2011.5767865

[24] A. V. Aho and T. G. Peterson, "A minimum distance error-correcting parser for context-free languages", SIAM J. Comput., vol. 1, pp. 305-312, Dec. 1972.

[25] Computer Algorithms. String Pattern Matching Strategies, CA, Los Alamitos:IEEE Computer Society Press, 1994.

[26] Syntactic and Structural Pattern Recognition: Theory and Applications, Singapore:World Scientific, 1990.

[27] H. Bunke and J. Csirik, "Parametric string edit distance and its application to pattern recognition", IEEE Trans. Syst. Man Cybern., vol. 25, pp. 202-206, Jan. 1995.

[28] D. Dubois and H. Prade, Fuzzy Sets and Systems: Theory and Applications, New York:Academic, 1980.

[29] D. Dutta, "On fuzzy fication fuzzy language and multicategory fuzzy classifier", Proc. IEEE 27th Int. Conf. Cybernetics Society, pp. 591-595, 1977-Sept.

[30] J. Echanobe, J. R. González de Mendívil, J. R. Garitagoitia and C. F. Alastruey, "Deformed systems for contextual postprocessing", Fuzzy Sets Syst., vol. 96, pp. 335-341, 1998.

[31] K. S. Fu, Syntactic Methods in Pattern Recognition, New York:Academic, 1974.

[32] "Error-correcting parsing for syntactic pattern recognition" in Data-Structures Computer Graphics and Pattern Recognition, New York:Academic, 1976.

[33] K. S. Fu, Syntactic Pattern Recognition and Applications, NJ, Upper Saddle River:Prentice-Hall, 1982.

[34] J. R. González de Mendívil, J. R. Garitagoitia, J. J. Astrain and J. Echanobe, "Fuzzy automata for imperfect string matching", Proc. Estylf 2000, pp. 141-145, 2000-Sept.

[35] P. A. V. Hall and G. R. Dowling, "Approximate string matching", ACM Comput. Surveys, vol. 12, no. 4, pp. 381-402, Dec. 1980.

[36] J. Hopcroft and J. Ullman, Introduction to Automata Theory Languages and Computation, MA, Reading:Addison-Wesley, 1979.

[37] J. J. Hull, S. N. Srihari and R. Choudhari, "An integrated algorithm for text recognition: Comparison with a cascaded algorithm", IEEE Trans. Pattern Anal. Mach. Intell., vol. PAMI-5, pp. 384-395, Jul. 1983.

[38] M. Inui. The recognition of imperfect strings generated by fuzzy context sensitive grammars, Fuzzy Sets Syst., vol. 62, pp. 21-29, 1994.

[39] G. J. Klir and B. Yuan, Fuzzy Sets and Fuzzy Logic: Theory and Applications, NJ, Upper Saddle River:Prentice-Hall, 1995.

[40] K. Kukich, Techniques for automatically correcting words in text, ACM Comput. Surveys, vol. 24, no. 4, pp. 377-439, Dec. 1992.

[41] E. T. Lee and L. A. Zadeh, Note on fuzzy languages, Inform. Sci., vol. 1, pp. 421-434, 1969.

[42] A. Marzal and E. Vidal, "Computation of normalized edit distance and applications", IEEE Trans. Pattern Anal. Machine Intell., vol. 15, pp. 926-932, Sept. 1993.

[43] M. Mizumoto, J. Toyoda and K. Tanaka, Fuzzy languages, Syst. Comput. Control, vol. 1, no. 3, pp. 36-43, 1970.

[44] M. Mizumoto, Pictorial representations of fuzzy connectives Part I: Cases of T-norms T-conorms and averaging operators, Fuzzy Sets Syst., vol. 31, pp. 217-242, 1989.

[45] C. V. Negoita and D. A. Ralescu, Application of Fuzzy Sets to System Analysis,

MA, Boston:Birkhauser, 1975.

[46] B. Oommen, Constrained string editing, Inform. Sci., vol. 40, pp. 267-284, 1986.

[47] B. J. Oommen and R. L. Kashyap, "A formal theory for optimal and information theoretic syntactic pattern recognition", Pattern Recogn., vol. 31, no. 8, pp. 1157-1177, 1998.

[48] R. Reina, J. R. González de Mendívil and J. R. Garitagoitia, Improved character recognition system based on a neural network incorporating the context via fuzzy automata, Proc. 2nd Int. Conf. Fuzzy Logic Neural Networks, pp. 1143-1146, 1992-Jul.

[49] E. S. Ristad and P. N. Yianilos, Learning string-edit distance, IEEE Trans. Pattern Anal. Machine Intell., vol. 20, pp. 522-531, May 1998.

[50] D. Sankoff and J. B. Kruskal, Time Warps String Edits and Macromolecules: The Theory and Practice of Sequence Comparison, MA, Reading:Addison-Wesley, 1983.

[51] M. Schneider, H. Lim and W. Shoaff, The utilization of fuzzy sets in the recognition of imperfect strings in Fuzzy Sets Syst., vol. 49, pp. 331-337, 1992.

[52] M. G. Thomason, Finite fuzzy automata regular fuzzy languages and pattern recognitio", Pattern Recogn., vol. 5, pp. 383-390, 1973.

[53] A. J. Viterbi, Error bounds for convolutional codes and asymptotically optimum decoding algorithm, IEEE Trans. Inform. Theory, vol. IT-13, pp. 260-269, Apr. 1967.

[54] R. A. Wagner and M. J. Fischer, The string-to-string correction problem, J. ACM, vol. 21, no. 1, pp. 168-173, Jan. 1974.

[55] T. Thongtan and T. Phienthrakul, Sentiment Classification Using Document Embeddings Trained with Cosine Similarity, *ACL 2019 - 57th Annu. Meet. Assoc. Comput. Linguist. Proc. Student Res. Work.*, pp. 407–414, 2019.

[56] Baeza-Yates, R. 2004. A fast set intersection algorithm for sorted sequences. In

Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 3109, 400-408.

[57] Baeza-Yates, R. and Navarro, G. 1998. Fast approximate string matching in a dictionary. In Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE'98). Springer, 14-22.

[58] Baeza-Yates, R. A. and Gonnet, G. H. 1999. A fast algorithm on average for all-against-all sequence matching. In Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware (SPIRE'99). IEEE, 16.

[59] Baeza-Yates, R. A., Hurtado, C. A., and Mendoza, M. 2004. Query recommendation using query logs in search engines. In Proceedings of the International Workshop on Clustering Information over the Web (ClustWeb'04). Lecture Notes in Computer Science Series, vol. 3268, Springer, 588-596.

[60] Bayardo, R. J., Ma, Y., and Srikant, R. 2007. Scaling up all pairs similarity search. In Proceedings of the 16th International Conference on World Wide Web (WWW'07). ACM, New York, 131-140.

[61] Belazzougui, D. 2009. Faster and space-optimal edit distance “1” dictionary. In Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM'09). Springer, 154-167.

[62] Boytsov, L. 2011. Indexing methods for approximate dictionary searching: Comparative analysis. J. Exp. Algorithmics 16.

[63] Bratley, P. and Choueka, Y. 1982. Processing truncated terms in document retrieval systems. Inf. Process. Manage. 18, 5, 257-266.

[64] Cao, H., Jiang, D., Pei, J., He, Q., Liao, Z., Chen, E., and Li, H. 2008. Context-aware query suggestion by mining click-through and session data. In Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08). ACM, New York, 875-883.

[65] Celikik, M. and Bast, H. 2009. Fast error-tolerant search on very large texts. In Proceedings of the Symposium of Applied Computing (SAC'09). ACM, New York, 1724-1731.

[66] Chaudhuri, S., Ganti, V., and Kaushik, R. 2006. A primitive operator for similarity joins in data cleaning. In Proceedings of the 22nd International Conference on Data Engineering (ICDE'06). IEEE, 5.

[67] Chaudhuri, S. and Kaushik, R. 2009. Extending autocompletion to tolerate errors. In Proceedings of the 35th International Conference on Management of Data (SIGMOD'09). ACM, New York, 707-718.

[68] Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. 2001. Searching in metric spaces. ACM Comp. Surv. 33, 3, 273-321.

[69] Cole, R., Gottlieb, L.-A., and Lewenstein, M. 2004. Dictionary matching and indexing with errors and don't cares. In Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04). ACM, New York, 91-100.

[70] D'Amore, R. J. and Mah, C. P. 1985. One-time complete indexing of text: theory and practice. In Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'85). ACM, New York, 155-164.

[71] Du, M. W. and Chang, S. C. 1994. An approach to designing very fast approximate string matching algorithms. IEEE Trans. Knowl. Data Eng. 6, 4, 620-633.

[72] Figueroa, K., Chávez, E., Navarro, G., and Paredes, R. 2006. On the least cost for proximity searching in metric spaces. In Proceedings of the 5th Workshop on Efficient and Experimental Algorithms (WEA'06). Springer, 279-290.

[73] Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., and Srivastava, D. 2001. Approximate string joins in a database (almost) for free. In Proceedings of the 27th International Conference on Very Large Data Bases

(VLDB'01). Morgan Kaufmann Publishers Inc., San Francisco, CA, 491-500.

[74] James, E. B. and Partridge, D. P. 1973. Adaptive correction of program statements. Comm. ACM 16, 1, 27-37.

[75] Ji, S., Li, G., Li, C., and Feng, J. 2009. Efficient interactive fuzzy keyword search. In Proceedings of the 18th International Conference on World Wide Web (WWW'09). ACM, New York, 371-380.

[76] Jokinen, P. and Ukkonen, E. 1991. Two algorithms for approximate string matching in static texts. In Proceedings of the. 2nd Annual Symposium on Mathematical Foundations of Computer Science. P. Jokinen and E. Ukkonen, Eds., Lecture Notes in Computer Science, vol. 520, 240-248.

[77] Kahveci, T. and Singh, A. K. 2001. Efficient index structures for string databases. In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01). Morgan Kaufmann Publishers Inc., San Francisco, CA, 351-360.

[78] Kim, Y., Seo, J., and Croft, W. B. 2011. Automatic boolean query suggestion for professional search. In Proceedings of the 34th International ACM SIGIR CONFERENCE on Research and DEVELOPMENT in Information (SIGIR'11). ACM, New York, 825-834.

[79] Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. 10, 8, 707-710.

[80] Li, C., Lu, J., and Lu, Y. 2008. Efficient merging and filtering algorithms for approximate string searches. In Proceedings of the 24th International Conference on Data Engineering (ICDE'08). IEEE, 257-266.

[81] Li, G., Wang, J., Li, C., and Feng, J. 2012. Supporting efficient top-k queries in type-ahead search. In Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'12). ACM, New York, 355-364.

[82] Lund, C. and Yannakakis, M. 1993. On the hardness of approximating

minimization problems. In Proceedings of the 25th Annual ACM Symposium on Theory of computing (STOC'93). ACM, New York, 286-293.

[83] Mihov, S. and Schulz, K. U. 2004. Fast approximate search in large dictionaries. Comput. Linguistics 30, 451-477.

[84] Mor, M. and Fraenkel, A. S. 1982. A hash code method for detecting and correcting spelling errors. Comm. ACM 25, 12, 935-938.

[85] Myers, E. W. 1994. A sublinear algorithm for approximate keyword searching. Algorithmica V12, 4, 345--374.

[86] Myers, G. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM 46, 1-13.

[87] Navarro, G., Baeza-Yates, R., Sutinen, E., and Tarhio, J. 2000. Indexing methods for approximate string matching. IEEE Data Eng. Bull. 24, 2001.

[88] Navarro, G. and Salmela, L. 2009. Indexing variable length substrings for exact and approximate matching. In Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE'09). Springer, 214--221.

[89] Needleman, S. B. and Wunsch, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. 48, 3, 443-453.

[90] Russo, L. M. S., Navarro, G., Oliveira, A. L., and Morales, P. 2009. Approximate string matching with compressed indexes. Algorithms 2, 3, 1105-1136.

[91] Sankoff, D. 1972. Matching sequences under deletion-insertion constraints. Proc. Nat. Acad. Sci. 69, 4-6.

[92] Schulz, K. U. and Mihov, S. 2002. Fast string correction with Levenshtein automata. Int. J. Doc. Anal. Recogn. 5, 1, 67-85.

[93] Sellers, P. H. 1974. On the theory and computation of evolutionary distances. SIAM J. Appl. Math. 26, 4, 787-793.

[94] Shi, F. and Mefford, C. 2005. A new indexing method for approximate search

in text databases. In Proceedings of the the 5th International Conference on Computer and Information Technology (CIT'05). IEEE Computer Society, Los Alamitos, CA, 70-76.

[95] Sutinen, E. and Tarhio, J. 1995. On using q-gram locations in approximate string matching. In Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95). Springer, 327-340.

[96] Sutinen, E. and Tarhio, J. 1996. Filtration with q-samples in approximate string matching. In Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96). Springer, 50-63.

[97] Ukkonen, E. 1983. Algorithms for approximate string matching. Info. Control 64, 1--3, 100-118.

[98] Ukkonen, E. 1993. Approximate string-matching over suffix trees. In Proceedings of the Conference on Advanced Information Systems Engineering. Lecture Notes in Computer Science, vol. 684, 228-242.

[99] Wu, S. and Manber, U. 1992. Fast text searching allowing errors. Comm. ACM 35, 10, 83-91.

[100] R. Guha, R. McCool, and E. Miller, “Semantic Search,” in Proceedings of the 12th international conference on World Wide Web, 2003, pp. 700-709.

[101] T. Tran, D.M. Herzig, and G. Ladwig, “SemSearchPro – Using semantics throughout the search process,” Web Semantics: Science, Services and Agents on the World Wide Web, vol. 9, no. 4, pp. 349-364, Dec. 2011.

[102] Kleshch, K., & Shablii, V. (2023). Comparison of fuzzy search algorithms based on Damerau-Levenshtein automata on large data. *Technology Audit and Production Reserves*, *4*(2(72), 27–32. https://doi.org/10.15587/2706-5448.2023.286382

[103] Клещ, К. О., & Царьов, М. О. (2023). МОДИФІКАЦІЯ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ ДЛЯ ВИКОРИСТАННЯ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ. *Таврійський науковий вісник. Серія: Технічні науки*, (3), 21-28.

https://doi.org/10.32782/tnv-tech.2023.3.3

[104] Kleshch, K. (2024). Development of fuzzy search method for creating an efficient information search system in text data. Technology Audit and Production Reserves, 1 (2 (75)), 20–24. doi: https://doi.org/10.15587/2706-5448.2024.298425.

[105] М. В. Михайлова. Порівняння алгоритмів нечіткого пошуку в текстах українською мовою: Радіоелектроніка, інформатика, управління, 2007, 80 с.

[106] Відстань Дамерау-Левенштейна [Електронний ресурс]. https://www.geeksforgeeks.org/damerau-levenshtein-distance/

[107] Fred J. Damerau. A Technique for Computer Detection and Correction of Spelling Errors : Communications of the ACM, 1964, с. 171 – 176.

[108] J. P. Carvalho and L. Coheur, "Introducing UWS - A fuzzy based word similarity function with good discrimination capability: Preliminary results," *2013 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, Hyderabad, India, 2013, pp. 1-8.

[109] H. Ayeldeen, A. E. Hassanien and A. A. Fahmy, "Lexical similarity using fuzzy Euclidean distance," *2014 International Conference on Engineering and Technology (ICET)*, Cairo, Egypt, 2014, pp. 1-6.

[110] Mihov S. Fast Approximate Search in Large Dictionaries / S. Mihov, K. Schulz. // Computational Linguistics. – 2004.

[111] Yu, M., Li, G., Deng, D. et al. String similarity search and join: a survey. Front. Comput. Sci. 10, 399–417 (2016).

[112] Вступ в алгоритми, 4 видання / [Т.Кормен, Р.Рівест]., 2022. – 1312 с.

[113] Fancy Letters [Електронний ресурс]. https://symbl.cc/en/collections/fancy-letters/

[114] Посібник користувача Google Benchmar [Електронний ресурс]. https://github.com/google/benchmark/blob/main/docs/user_guide.md#runtime-and-reporting-considerations

[115] Boytsov, L. (2011). Indexing methods for approximate dictionary searching. ACM *Journal of Experimental Algorithmics, 16.* doi: https://doi.org/10.1145/1963190.1963191

[116] Z. Liu and Y. Zhang, “Research and Design of E-commerce Semantic Search,” 2010 3rd International Conference on Information Management, Innovation Management and Industrial Engineering, pp. 332-334, Nov. 2010.

[117] C. Lv, T. Kobayashi, K. Agusa, K. Wu, and Q. Zhu, “Image Semantic Search Engine,” 2009 First International Workshop on Database Technology and Applications, pp. 156-159, Apr. 2009.

[118] D. Tümer, M.A. Shah, and Y. Bitirim, “An Empirical Evaluation on Semantic Search Performance of Keyword-Based and Semantic Search Engines: Google, Yahoo, Msn and Hakia,” 2009 Fourth International Conference on Internet Monitoring and Protection, pp. 51-55, 2009.

[119] E. Nyamsuren and H. Choi, “Building a semantic model of a textual document for efficient search and retrieval,” in 11th International Conference onAdvanced Communication Technology, 2009, pp. 298-302.

[120] S. Chatvichienchai and K. Tanaka, “An Effective Document Search Technique by Semantic Relationship Approach,” 2009 Fourth International Conference on Computer Sciences and Convergence Information Technology, pp. 53-58, 2009.

[121] L.-F. Lai, C.-C. Wu, P.-Y. Lin, and L.-T. Huang, “Developing a fuzzy search engine based on fuzzy ontology and semantic search,” 2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011), pp. 2684-2689, Jun. 2011.

[122] R. Li, K. Wen, Z. Lu, X. Sun, and Z. Wang, “An improved semantic search model based on hybrid fuzzy description logic,” 2006 Japan-China Joint Workshop on Frontier of Computer Science and Technology, pp. 139-146, Nov. 2006.

[123] F. P. Romero, J.A. Olivas, J. De Mata, and C. Carmen, “BUDI: Architecture for Fuzzy Search in Documental Repositories,” vol. 16, pp. 71-85, 2009.

[124] Annunziato, M., Pizzuti, S.: Adaptive Parameterization of Evolutionary Algorithms Driven by Reproduction and Competition, ESIT, Aachen, Germany (2000)

[125] Eshelman, L.J.: The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination. In: Rawlins, G. (ed.) Foundations of Genetic Algorithms, pp. 265–283. Morgan Kaufmann, San Francisco (1991)

[126] E.-L. Silva-Ramirez, J.-F. Cabrera-Sánchez. Co-active neuro-fuzzy inference system model as single imputation approach for non-monotone pattern of missing data. Neural Comput. Appl., 33 (2021), pp. 8981-9004

ДОДАТОК 1. СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ ТА ВІДОМОСТІ ПРО АПРОБАЦІЮ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЇ

Статті у наукових фахових виданнях України

[1] Kleshch, K., & Shablii, V. (2023). Comparison of fuzzy search algorithms based on Damerau-Levenshtein automata on large data. Technology Audit and Production Reserves, 4(2(72), 27–32. https://doi.org/10.15587/2706-5448.2023.286382.

[2] Клещ, К. О., & Царьов, М. О. (2023). МОДИФІКАЦІЯ АЛГОРИТМІВ НЕЧІТКОГО ПОШУКУ ДЛЯ ВИКОРИСТАННЯ ТАБЛИЦІ ПОДІБНОСТІ СИМВОЛІВ. Таврійський науковий вісник. Серія: Технічні науки, (3), 21-28. https://doi.org/10.32782/tnv-tech.2023.3.3.

[3] Kleshch, K. (2024). Development of fuzzy search method for creating an efficient information search system in text data. Technology Audit and Production Reserves, 1 (2 (75)), 20–24. doi: https://doi.org/10.15587/2706-5448.2024.298425.