

**ПРИВАТНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ЄВРОПЕЙСЬКИЙ УНІВЕРСИТЕТ»**

КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

СПЕЦІАЛЬНІСТЬ: 122 – КОМП'ЮТЕРНІ НАУКИ

ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА: «КОМП'ЮТЕРНІ НАУКИ»

**ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА
РОБОТА**

на тему:

**АЛГОРИТМИ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ ТА ЇХ ІНТЕГРАЦІЯ В
ПРИКЛАДНІ ПРОГРАМИ**

Студента _____
(підпис студента)

Калашник Олексій Іванович
(прізвище, ім'я по-батькові студента)

Науковий
керівник _____
(підпис наукового керівника)

Улічев. О.С. Доцент к.т.н.
(прізвище, ініціали, наукове звання і ступінь)

Допущено до захисту “ _____ ” 2025р.

Завідувач кафедри _____
(підпис)

(прізвище, ініціали, наукове звання і ступінь)

Декан факультету _____
(підпис)

(прізвище, ініціали)

Завідувач
навчальної частини _____
(підпис)

(прізвище, ініціали)

Київ – 2025

КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
СПЕЦІАЛЬНІСТЬ: 122 – КОМП'ЮТЕРНІ НАУКИ
ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА: «КОМП'ЮТЕРНІ НАУКИ»

ЗАВДАННЯ

на випускню кваліфікаційну бакалаврську роботу студентів:

Калашник Олексій Іванович

1. Тема роботи: АЛГОРИТМИ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ ТА ЇХ ІНТЕГРАЦІЯ В ПРИКЛАДНІ ПРОГРАМИ
2. Затверджена наказом по університету від “ ” 2025 р. № _____
3. Цільова установка на вихідні дані до роботи

Перелік графічного матеріалу (з точним визначенням обов'язкових креслень)_

Консультанти по роботі, із зазначенням розділів, що їх стосуються:

Розділ	Науковий керівник	Підпис, дата	
		Завдання видав	Завдання отримав

Зміст випускної кваліфікаційної бакалаврської роботи (перелік питань)

1. Календарний план виконання випускної кваліфікаційної бакалаврської роботи

№ з/п	Назва етапів випускної кваліфікаційної бакалаврської роботи	Термін виконання етапів випускної кваліфікаційної бакалаврської роботи	
		За планом	Фактично
1.	Обрання теми випускної кваліфікаційної магістерської роботи (ВКБР)	05.08.24 – 13.09.24	
2.	Опрацювання літератури та інших джерел для написання ВКБР	16.09.24 – 01.11.24	
3.	Виконання 1 розділу ВКБР	04.11.24 – 28.02.25	
4.	Виконання 2 розділу ВКБР	03.03.25 – 21.03.25	
5.	Виконання 3 розділу ВКБР, висновки	24.03.25 – 18.04.25	
6.	Передзахист та проведення перевірки на плагіат	21.04.24 – 02.05.25	
7.	Відгук керівника, рецензія	01.05.25 – 12.05.25	

2. Дата видачі завдання “ ” 2025р.

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

ЗМІСТ

<u>ЧАСТИНА I. ТЕОРЕТИЧНІ ОСНОВИ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ</u>	14
<u>1.1 Огляд літературних джерел за тематикою дослідження</u>	17
<u>1.2 Огляд вибраних алгоритмів</u>	20
<u>1.2.1 Алгоритм Дамерау–Левенштейна</u>	21
<u>1.2.2 Алгоритм Джаро–Вінклера</u>	24
<u>1.2.3 Алгоритм N-грам</u>	26
<u>1.2.4 Алгоритми Bitap та SoundEx</u>	31
<u>1.3 Розширена постановка задачі</u>	36
<u>Висновки до Частини I</u>	41

<u>ЧАСТИНА II. РОЗРОБКА АЛГОРИТМУ НЕЧІТКОГО ПОШУКУ</u>	43
<u>2.1 Проектування та модифікація алгоритму нечіткого пошуку</u>	43
<u>2.2 Реалізація алгоритму та таблиці подібності символів</u>	45
<u>2.3 Реалізація алгоритму нечіткого пошуку Дамерау-Левенштейна</u>	54
<u>Висновки до Частини II</u>	60
<u>ЧАСТИНА III. ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ТА АНАЛІЗ ЕФЕКТИВНОСТІ</u>	62
<u>3.1 Методика тестування алгоритму нечіткого пошуку та розробка фікстури та сценарії тестування</u>	62
<u>3.2. Тестування таблиці подібності та розрахунок відстані редагування</u>	68
<u>3.3 Порівняльний аналіз швидкодії та точності</u>	72
<u>Висновки до Частини III</u>	81
<u>ВИСНОВОК</u>	83
<u>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ</u>	86

АНОТАЦІЯ

Бакалаврська дипломна робота Калашника Олексія Івановича

Тема: АЛГОРИТМИ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ ТА ЇХ ІНТЕГРАЦІЯ В ПРИКЛАДНІ ПРОГРАМИ

У сучасному цифровому середовищі, де обсяг інформації постійно зростає, здатність оперативно знаходити релевантні дані набуває критичного значення. Це особливо актуально для інтернет-пошуку, де незначні орфографічні чи друкарські помилки у запитах можуть суттєво впливати на результати. Традиційні алгоритми пошуку часто не здатні повною мірою враховувати такі нюанси, що підкреслює необхідність впровадження методів нечіткого пошуку.

Метою цієї роботи є комплексне дослідження алгоритмів нечіткого пошуку, аналіз їх ефективності та визначення способів оптимізації процесу пошуку інформації. Особлива увага приділяється алгоритмічним підходам до корекції помилкових запитів, а також адаптивним методам зіставлення даних із пошуковими базами.

Ключовим завданням є розробка та тестування модифікованого алгоритму нечіткого пошуку, який інтегрує сучасні технології обробки текстових даних із використанням спеціалізованих таблиць подібності символів. Запропонований алгоритм включає кілька етапів: попередню обробку запиту, токенизацію, обчислення показників релевантності за допомогою методів оцінки подібності, а також адаптивне зіставлення з інформаційними базами. Проведено порівняльний аналіз ефективності модифікованого алгоритму у порівнянні з традиційними підходами, що дозволило виявити його переваги в умовах високої інформаційної насиченості.

Результатом роботи є детальний аналіз сучасних алгоритмів нечіткого пошуку, розробка інноваційного підходу до корекції запитів і практична

апробація запропонованого методу, що сприяє підвищенню точності та ефективності пошукових систем.

Загальний обсяг роботи: 80 сторінок, 45 рисунків, 4 таблиці, 4 додатки, 9 джерел.

Ключові слова: нечіткий пошук, алгоритми пошуку, обробка текстових даних, відстань редагування, алгоритм Дамерау–Левенштейна.

ABSTRACT

Bachelor's Thesis by Oleksii Ivanovych Kalashnyk

Topic: *Development of a Fuzzy Search Algorithm*

In today's digital environment, where the volume of information is continuously growing, the ability to promptly find relevant data has become critically important. This is particularly true for internet searches, where minor spelling or typographical errors in queries can significantly affect the results. Traditional search algorithms are often unable to adequately account for such nuances, highlighting the need for the implementation of fuzzy search techniques.

The objective of this thesis is a comprehensive study of fuzzy search algorithms, analysis of their effectiveness, and identification of methods to optimize the information retrieval process. Special attention is given to algorithmic approaches that enable query correction and to adaptive methods for matching data from search bases.

The key task is the development and testing of a modified fuzzy search algorithm that integrates modern text data processing technologies using specialized character similarity tables. The proposed algorithm includes several stages: query preprocessing, tokenization, calculation of relevance metrics using similarity assessment methods, and adaptive matching against information databases. A comparative analysis of the modified algorithm's effectiveness versus traditional approaches was conducted, revealing its advantages in environments with high informational density.

The outcome of this research is a detailed analysis of contemporary fuzzy search algorithms, the development of an innovative query correction approach, and the practical testing of the proposed method, which contributes to improving the accuracy and efficiency of search systems.

The total volume of the work: 80 pages, 45 figures, 4 tables, 4 appendices, and 9 references.

Keywords: fuzzy search, search algorithms, text data processing, edit distance, Damerau–Levenshtein algorithm.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

- НП (Нечіткий пошук) – метод пошуку текстових рядків, які є подібними або близькими до заданого пошукового запиту, навіть якщо запит містить помилки або неточності.
- NLP (Natural Language Processing) – галузь комп'ютерних наук, що розробляє алгоритми для аналізу, розуміння та генерації природної мови.
- UNICODE (Universal Coded Character Set) – стандарт кодування, що забезпечує універсальне представлення символів для текстів майже всіх мов світу.
- C++ – компілювана, статично типізована та об'єктно-орієнтована мова програмування, широко використовувана для створення високопродуктивних додатків.
- Python – високорівнева, інтерпретована мова програмування з динамічною типізацією, що підтримує об'єктно-орієнтований підхід і широко застосовується у наукових дослідженнях та розробці веб-додатків.
- JSON (JavaScript Object Notation) – текстовий формат обміну даними, який забезпечує зручну передачу та читання інформації між різними системами.
- UTF-16 (Unicode Transformation Format) – спосіб кодування символів Юнікоду, що використовує послідовності 16-бітових слів для представлення текстових даних.
- АС Асимптотична складність – характеристика алгоритмів, що описує темп зростання часу виконання або споживання пам'яті в залежності від розміру вхідних даних.

- RapidJSON – ефективна бібліотека для обробки JSON у C++, що забезпечує високу продуктивність завдяки оптимізованим алгоритмам парсингу.
- IDE (Integrated Development Environment) – інтегроване середовище розробки, яке об'єднує інструменти для написання, налагодження, тестування та управління програмним кодом у єдиному інтерфейсі.
- Бенчмарк (benchmark) – процес або набір тестів для вимірювання продуктивності та ефективності системи, компонента чи алгоритму.
- Matplotlib – бібліотека для побудови графіків та візуалізації даних у мові програмування Python, що широко використовується у наукових дослідженнях та аналізі даних.
- ФВА (функціонально-вартісний аналіз) – метод оцінки ефективності проектних рішень шляхом аналізу їх функціональності та витрат.
- ПЗ (програмне забезпечення) – комплекс програмних засобів, який виконує певні завдання та забезпечує функціональність для користувачів.
- ПП (програмний продукт) – конкретна реалізація програмного забезпечення, готова до використання та комерційного розповсюдження.

ВСТУП

У сучасному інформаційному суспільстві обсяг текстових даних, доступних у відкритих джерелах, невпинно зростає, що ставить перед користувачами і розробниками інформаційних систем нові виклики щодо ефективного пошуку релевантної інформації. Інформація є основою для прийняття рішень у різних сферах — від особистих потреб до наукових досліджень та бізнес-аналітики. Однак зростання обсягу даних супроводжується збільшенням неточностей, орфографічних помилок, варіативності написання слів і мовних особливостей, що ускладнює пошук необхідних даних за допомогою традиційних алгоритмів, які базуються на точній відповідності пошукового запиту.

У зв'язку з цим актуальності набувають методи нечіткого пошуку, що дозволяють знаходити інформацію, навіть якщо пошуковий запит містить помилки, варіації написання або незначні відхилення від вихідного тексту. Нечіткий пошук використовується в різних галузях, зокрема у пошукових системах, системах перевірки орфографії, базах даних, обробці природної мови та багатьох інших сферах, де точність і повнота результатів відіграють важливу роль.

Зокрема, алгоритми нечіткого пошуку активно застосовуються в пошукових системах, таких як Google, де забезпечується корекція запитів, автозаповнення та ранжування результатів з урахуванням можливих неточностей. Вони базуються на використанні різних методів, включаючи розрахунок відстані Левенштейна, розмиті множини, імовірнісні моделі та методи машинного навчання.

Зважаючи на актуальність теми, дана бакалаврська робота присвячена дослідженню сучасних алгоритмів нечіткого пошуку, що використовуються в пошукових системах, а також розробці алгоритму нечіткого пошуку на основі таблиці подібності символів.

Метою роботи: є аналіз сучасних алгоритмів нечіткого пошуку, що застосовуються в пошукових системах, та розробка власного алгоритму нечіткого пошуку з використанням таблиці подібності символів.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

- Проаналізувати основні алгоритми нечіткого пошуку, що застосовуються в сучасних інформаційно-пошукових системах;
- Дослідити особливості роботи алгоритмів нечіткого пошуку;
- Розробити алгоритм нечіткого пошуку з використанням таблиці подібності символів;
- Створити таблицю подібності символів для покращення ефективності нечіткого пошуку;
- Реалізувати та протестувати розроблений алгоритм;
- Порівняти продуктивність та ефективність запропонованого підходу з існуючими методами нечіткого пошуку.

Об'єктом дослідження: є методи та алгоритми нечіткого пошуку, що застосовуються в інформаційно-пошукових системах.

Предметом дослідження: є алгоритми нечіткого пошуку, їхні характеристики, методи реалізації та способи підвищення ефективності пошукових процесів.

Методи дослідження: Для досягнення поставленої мети в роботі використовуються методи теоретичного аналізу наукових джерел, алгоритмічного моделювання, програмної реалізації, а також експериментального тестування запропонованого алгоритму.

Наукова новизна: роботи полягає в розробці алгоритму нечіткого пошуку, що базується на таблиці подібності символів, що дозволяє підвищити

точність пошукових результатів у разі наявності орфографічних помилок або неточностей у запитах користувачів.

Практичне значення роботи полягає в можливості використання розробленого алгоритму для покращення пошукових систем, систем перевірки орфографії, баз даних та інших програмних рішень, де важливим є ефективний нечіткий пошук.

ЧАСТИНА I. ТЕОРЕТИЧНІ ОСНОВИ НЕЧІТКОГО ПОШУКУ В ТЕКСТІ

У сучасному інформаційному середовищі обсяг текстових даних стрімко зростає, а разом із ним збільшується і потреба у швидкому й ефективному пошуку релевантної інформації.

Алгоритми нечіткого пошуку спрямовані на знаходження приблизних відповідей між запитами користувачів і текстовими даними, навіть у випадках, коли точна відповідність відсутня або містить незначні відхилення. Вони широко застосовуються у пошукових системах, автоматизованих коректорах тексту, базах даних та системах обробки природної мови (NLP).

До найпоширеніших алгоритмів нечіткого пошуку належать:

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

Традиційні алгоритми пошуку, що базуються на точному співпадінні символів, виявляються недостатньо гнучкими, коли користувачі вводять запити з орфографічними помилками, скороченнями або варіаціями написання слів. Випадки, коли найменше відхилення у формулюванні запиту призводить до втрати важливої інформації, є не лише незручністю для користувача, але й серйозною проблемою для великих інформаційних систем, таких як пошукові сервіси, бази даних, системи розпізнавання тексту та обробки природної мови.

Здатність пошукових алгоритмів враховувати неточності у введених даних є критичною для покращення доступу до інформації. Уявімо ситуацію, коли користувач шукає певний медичний термін або ім'я людини, але припускається незначної помилки у написанні. Звичайний точний пошук у такому випадку поверне нуль результатів, тоді як алгоритми нечіткого пошуку здатні запропонувати варіанти, найбільш наближені до введеного запиту.

Саме тому розробка та впровадження алгоритмів нечіткого пошуку стало одним із ключових напрямів у розвитку сучасних інформаційних технологій. Вони використовуються для коригування правописних помилок у пошукових системах, автоматичної перевірки граматики у текстових редакторах, дедуплікації записів у базах даних, обробки природної мови та навіть у криміналістиці, де важливо знайти схожі імена або дані, навіть якщо вони записані з помилками.

Однією з найважливіших сфер застосування нечіткого пошуку є пошукові системи, такі як Google, Bing чи Yahoo. Ці платформи обробляють мільярди запитів щодня, і навіть незначні покращення у алгоритмах пошуку можуть значно вплинути на якість видачі результатів. Завдяки нечіткому пошуку користувач отримує релевантні результати навіть у випадках, коли він ввів слово неправильно або використав рідкісний варіант написання. Наприклад, Google автоматично виправляє введені помилки, пропонуючи фразу «Можливо, ви мали на увазі...», що є реалізацією алгоритмів нечіткого пошуку.

Ще одним важливим напрямом застосування є автоматизовані системи перевірки правопису та граматики. Програми, такі як Microsoft Word, Grammarly або Google Docs, використовують нечіткий пошук для виявлення неправильно написаних слів і пропонують їх виправлення. Аналогічні алгоритми використовуються і у системах машинного перекладу, таких як

Google Translate, де неправильний або незавершений ввід слів може бути автоматично скоригований перед перекладом.

Окрім пошукових систем і текстових редакторів, нечіткий пошук активно використовується у сфері аналізу великих текстових масивів. У базах даних із мільйонами записів імен, назв компаній або адрес невеликі розбіжності у написанні можуть створювати проблеми при обробці інформації. Наприклад, запис «Олександр Коваленко» та «Александр Коваленко» може вважатися різними особами у звичайному пошуку, тоді як алгоритми нечіткої відповідності можуть правильно ідентифікувати їх як одного користувача.

Нечіткий пошук також відіграє важливу роль у криміналістиці та правоохоронних органах. Наприклад, у міжнародних базах даних розшукуваних осіб часто трапляються різні варіанти написання одного й того ж імені через особливості транслітерації різними мовами. Використання фонетичних алгоритмів, таких як Soundex, дозволяє знаходити схожі записи, навіть якщо вони були введені з неточностями.

Однак, попри всі переваги нечіткого пошуку, він має і свої обмеження. Найбільш очевидною проблемою є обчислювальна складність. Багато алгоритмів, зокрема ті, що базуються на відстані редагування (наприклад, Дамерау-Левенштейна), можуть бути досить ресурсомісткими при роботі з великими текстовими масивами. Крім того, нечіткий пошук може повертати надто багато варіантів, що ускладнює вибір найбільш релевантного результату.

Ще одна проблема — можливість хибних спрацьовувань, коли система знаходить слова, що мають схожість на рівні символів, але не мають жодного смислового зв'язку. Наприклад, слова «банк» і «бант» можуть мати невелику відстань редагування, але з точки зору змісту вони зовсім різні. Саме тому сучасні алгоритми пошуку комбінують методи відстані редагування з іншими

підходами, такими як семантичний аналіз, розпізнавання контексту або використання машинного навчання.

Нечіткий пошук став невід'ємною частиною сучасних інформаційних технологій, забезпечуючи більш гнучкий і адаптивний підхід до обробки текстових даних. Його використання дозволяє покращити якість пошуку, автоматизувати виправлення помилок і значно підвищити ефективність роботи з текстовою інформацією у найрізноманітніших сферах. Незважаючи на певні технічні виклики, розвиток комбінованих методів та інтеграція нечітких алгоритмів у штучний інтелект обіцяють ще більше розширити їх можливості у майбутньому.

1.1 Огляд літературних джерел за тематикою дослідження

Розвиток алгоритмів нечіткого пошуку є однією з ключових тем у галузі інформаційного пошуку, обробки текстових даних та штучного інтелекту. Дослідження в цій сфері розпочалися ще у 1960-х роках і з того часу пройшли значний еволюційний шлях, поєднуючи традиційні методи обчислення подібності текстів із сучасними підходами, що базуються на машинному навчанні та обробці природної мови (NLP).

Аналіз наукових джерел, присвячених нечіткому пошуку, показує, що існує кілька основних напрямків досліджень:

1. Фундаментальні математичні моделі пошуку та подібності
2. Класичні алгоритми редагування тексту
3. Фонетичні методи пошуку
4. Статистичні та ймовірнісні моделі пошуку
5. Методи, що базуються на машинному навчанні та NLP

Фундаментальні математичні моделі пошуку та подібності

Основи теорії нечіткого пошуку закладені в працях Левенштейна (1965), де вперше було запропоновано метрику редагування текстових рядків на основі операцій вставки, видалення та заміни символів. Його дослідження стали основою для багатьох подальших алгоритмів, що використовуються для порівняння текстових рядків.

Дамерау (1964) доповнив концепцію Левенштейна, включивши в розрахунки операцію транспозиції символів, що значно покращило точність визначення схожості між словами.

Класичні алгоритми редагування тексту

Вагоме місце серед алгоритмів нечіткого пошуку займає метрика Дамерау-Левенштейна, яка широко використовується у пошукових системах та системах корекції орфографічних помилок. Пізніші дослідження показали, що цей алгоритм можна оптимізувати, використовуючи динамічне програмування та евристичні методи, що значно знижує його обчислювальну складність.

Алгоритм Джаро-Вінклера, запропонований Вінклером (1990), став важливим інструментом для порівняння коротких текстових рядків, таких як імена, назви компаній та адреси. Він використовується в інформаційних системах для знаходження варіантів написання слів із невеликими розбіжностями.

Метод N-грам, що виник у 1970-х роках у сфері статистичної обробки тексту, набув широкого застосування у сучасних пошукових системах та алгоритмах обробки природної мови. Основна ідея цього методу полягає у розбитті текстового рядка на послідовності символів фіксованої довжини, що дозволяє аналізувати подібність слів та речень.

Фонетичні методи пошуку

Фонетичні алгоритми, такі як Soundex, були розроблені ще на початку XX століття (Оделл Рассел, 1918) для пошуку прізвищ у великих базах даних. Soundex став основою для більш сучасних фонетичних алгоритмів, таких як Metaphone і Double Metaphone, які враховують особливості вимови слів у різних мовах.

Роботи в галузі фонетичних методів продемонстрували, що такі алгоритми ефективні для розпізнавання схожих слів, але мають обмеження при роботі з мовами зі складною морфологією, наприклад, українською чи німецькою.

Статистичні та ймовірнісні моделі пошуку

У 1990-х роках активно розвивалися статистичні методи аналізу тексту, які дозволяли покращити точність нечіткого пошуку. Такі моделі, як Latent Semantic Analysis (LSA) та TF-IDF, дозволяли визначати семантичну близькість між текстовими документами, навіть якщо вони містили різні слова, що позначають однакові поняття.

Ймовірнісні моделі, такі як Hidden Markov Models (HMM), також знайшли застосування у нечіткому пошуку, особливо в задачах розпізнавання тексту та автоматичного виправлення орфографічних помилок.

Методи, що базуються на машинному навчанні та NLP

Останнім десятиліттям відбувся значний прогрес у застосуванні глибокого навчання для обробки текстових даних. Алгоритми, що базуються на нейронних мережах, такі як Word2Vec (Mikolov, 2013) та BERT (Devlin, 2018), дозволили моделювати семантичну подібність слів і речень у текстах, що суттєво підвищило точність пошукових алгоритмів.

Сучасні дослідження (Ван та ін., 2020) показують, що комбінування традиційних алгоритмів нечіткого пошуку із трансформерними моделями дає

найкращі результати у таких задачах, як класифікація текстів, пошук схожих документів та автоматичне виправлення помилок у пошукових запитах.

1.2 Огляд вибраних алгоритмів

Уявіть, що ви шукаєте інформацію, але припустилися друкарської помилки. Або, можливо, вам потрібно знайти документи з подібним змістом, навіть якщо вони сформульовані по-різному. Як пошукові системи та інформаційні системи виявляють ці неточності та все одно знаходять потрібні результати?

Саме для цього існують алгоритми нечіткого пошуку – потужні інструменти, що допомагають знайти інформацію, навіть якщо введені дані містять помилки, варіативність написання або відмінності в структурі.

Дамерау–Левенштейн – здатний визначити, наскільки схожі два слова, враховуючи вставки, видалення, заміни символів і навіть перестановку сусідніх букв.

Джаро–Вінклер – особливо ефективний для порівняння імен, адрес і назв компаній, наголошуючи на схожих початках слів.

Алгоритм N-грам – розбиває текст на дрібні фрагменти та аналізуючи їхню частотність, що дозволяє знаходити подібні слова та навіть фрази.

Bitap – використовує побітові операції для миттєвого знаходження подібних рядків у великих текстових масивах.

Soundex – допомагає знайти слова з однаковим звучанням, навіть якщо їх написання відрізняється – ідеальний для пошуку прізвищ або слів у різних мовах.

Кожен із цих алгоритмів має свої переваги та сфери застосування, але разом вони формують основу нечіткого пошуку, що забезпечує ефективну роботу пошукових систем, баз даних і навіть розумних помічників. Як саме вони

працюють і які їхні особливості? Далі ми розглянемо кожен алгоритм детальніше.

1.2.1 Алгоритм Дамерау–Левенштейна

Алгоритм Дамерау-Левенштейна є одним із найефективніших підходів до визначення схожості між двома рядками. Його основна концепція базується на оцінці відстані редагування, яка визначає мінімальну кількість елементарних операцій, необхідних для перетворення одного рядка в інший.

Відстань Дамерау-Левенштейна — це метрика, яка оцінює відмінності між двома рядками шляхом підрахунку таких операцій:

- Вставка символу;
- Видалення символу;
- Заміна символу;
- Транспозиція сусідніх символів (перестановка).

Дана відстань є розширенням класичної відстані Левенштейна, яка враховує лише три основні операції (вставка, видалення, заміна). Доповнення у вигляді операції транспозиції було запропоновано Фредеріком Дамерау, який під час досліджень орфографічних помилок у інформаційно-пошукових системах встановив, що понад 80% помилок належать саме до одного з чотирьох згаданих типів [1].

Для обчислення відстані Дамерау-Левенштейна найчастіше використовується метод динамічного програмування. Він передбачає побудову матриці розміром $(n + 1) \times (m + 1)$, де n та m — довжини відповідних рядків. Вартість усіх операцій (вставки, видалення, заміни та транспозиції) зазвичай приймається рівною 1, хоча у практичних реалізаціях вона може варіюватися.

Формальне визначення відстані між двома рядками a та b можна виразити за допомогою рекурсивного рівняння (1.1):

$$D(i, j) = \begin{cases} 0, & \text{якщо } i = 0, j = 0, \\ i, & \text{якщо } j = 0, \\ j, & \text{якщо } i = 0, \\ \min \begin{cases} D(i-1, j) + 1, \\ D(i, j-1) + 1, \\ D(i-1, j-1) + c(a_i, b_j), \\ D(i-2, j-2) + 1, \text{ якщо } i > 1, j > 1 \text{ і } a_i = b_{j-1}, b_j = a_{i-1} \end{cases} & , \text{ в інших випадках.} \end{cases}$$

де:

Функція $1(a_i \neq b_j)$ є індикаторною функцією, яка набуває значення 0, якщо $a_i = b_j$, і значення 1, якщо символи відрізняються [1].

А нижче показано блок-схему на рис. 1.1.

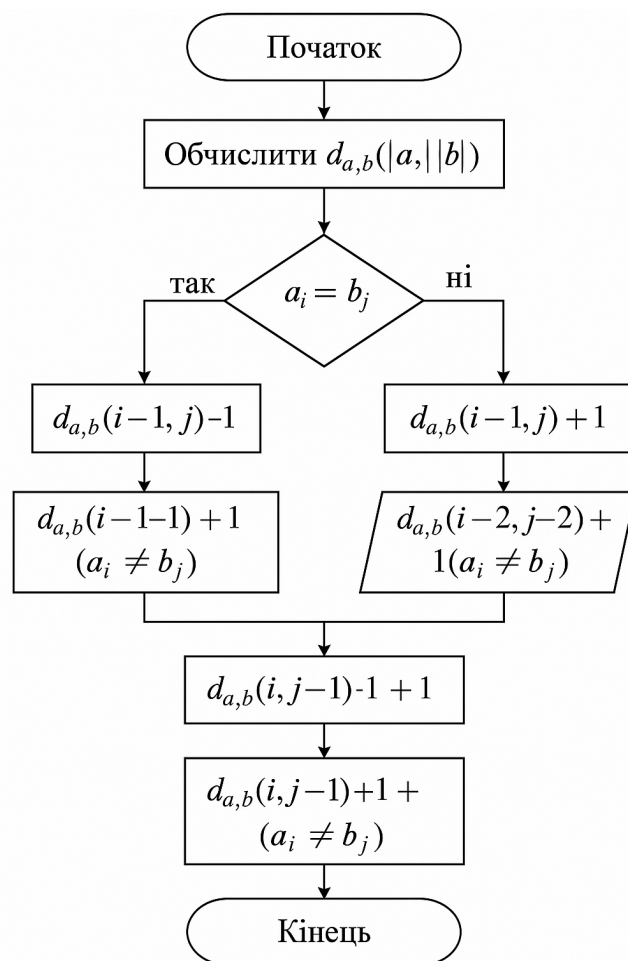


Рис.1.1. Блок-схема алгоритму Дамерау–Левенштейна

Кожен рекурсивний виклик у формулі відстані Дамерау–Левенштейна відповідає одній з основних операцій редагування:

- $d_{a,b}(i-1,j)+1$ – операція видалення символу з рядка a для отримання рядка b ;
- $d_{a,b}(i,j-1)+1$ – операція вставки символу в рядок a для отримання рядка b ;
- $d_{a,b}(i-1,j-1)+1(a_i \neq b_j)$ – операція заміни символу, що виконується лише в разі невідповідності символів у рядках a і b ;
- $d_{a,b}(i-2,j-2)+1(a_i \neq b_j)$ – операція транспозиції, яка передбачає перестановку двох сусідніх символів, якщо вони помінялися місцями.

Підсумкове значення відстані Дамерау–Левенштейна для двох рядків визначається як $d_{a,b}(|a|,|b|)$, де $|a|$ і $|b|$ – довжини відповідних рядків a та b [1].

Застосовуючи певне порогове значення для цієї метрики та порівнюючи всі слова у вхідному тексті з цільовим словом, можна здійснити нечіткий пошук. Він поверне всі слова, для яких відстань Дамерау–Левенштейна не перевищує встановленого порогу.

Обмеження алгоритму

Незважаючи на ефективність алгоритму, його головним недоліком є те, що він працює виключно на основі синтаксичних відмінностей і не враховує семантичний контекст. Це може призводити до ситуацій, коли результати пошуку містять слова, які технічно схожі на запит, але не мають з ним значного смислового зв'язку. Для покращення точності нечіткого пошуку доцільним є комбінування відстані Дамерау–Левенштейна з іншими методами, такими як лексичний аналіз або машинне навчання.

1.2.2 Алгоритм Джаро–Вінклера

Досліджуваний алгоритм має подібність до алгоритму Дамерау–Левенштейна, однак для порівняння текстових послідовностей у ньому застосовується метрика Джаро–Вінклера. Відстань Джаро–Вінклера є рядковою метрикою, що визначає ступінь подібності між двома текстовими рядками шляхом оцінки відстані редагування. Значення цієї метрики змінюється у межах від 0 до 1, де 0 відповідає повній відповідності між рядками, а 1 — повній їх відмінності [3]. Оскільки дана метрика формально визначена через коефіцієнт подібності Джаро–Вінклера, відстань між рядками обчислюється як її доповнення до одиниці:

$$D=1-S$$

де D — відстань Джаро–Вінклера, а S — коефіцієнт подібності Джаро–Вінклера.

Метрика була розроблена Вільямом Е. Вінклером у 1990 році як модифікація коефіцієнта подібності Джаро. Її головна особливість полягає в тому, що вона враховує не лише наявність спільних символів у рядках, а й їхній порядок, що підвищує точність визначення відповідностей у текстових даних.

Обчислення подібності Джаро–Вінклера передбачає початковий розрахунок коефіцієнта подібності Джаро, який визначається за формулою (2.2.1):

де:

- J - коефіцієнт подібності Джаро;
- C - кількість збіжних символів між двома рядками;
- L_1 та L_2 - довжини порівнюваних рядків;

- - кількість транспозицій (символів, що знаходяться у невідповідному порядку).

Згодом для отримання остаточного значення коефіцієнта подібності Джаро-Вінклера до отриманого значення застосовується поправочний коефіцієнт , що враховує довжину збіжної префіксної частини рядків. Остаточна формула (2.2.2) має вигляд:

де:

- - коефіцієнт подібності Джаро-Вінклера;
- - довжина збіжного префікса (до 4 символів);
- - ваговий коефіцієнт, що визначає вплив префіксної відповідності (стандартне значення .

Блок-схема даного алгоритму:

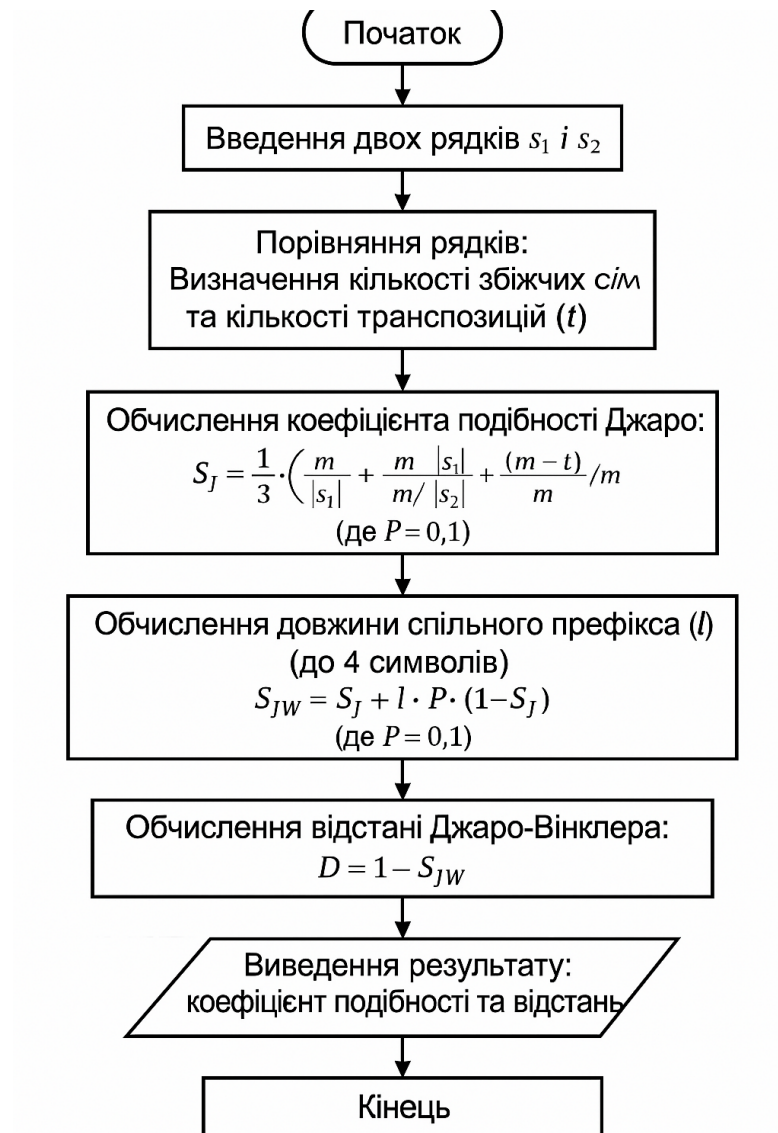


Рис. 1.2

1.2.3 Алгоритм N-грам

У всіх алгоритмах нечіткого пошуку, розглянутих раніше, зіставлення рядків здійснювалося шляхом покрокового порівняння окремих символів. Однак один із підходів, що дозволяє розширити це поняття, передбачає аналіз послідовностей символів або слів фіксованої довжини, відомих як N-грами.

Алгоритм N-грам

Алгоритм N-грам є методом порівняння текстових рядків, який використовується для оцінки їхньої подібності. Його принцип полягає у

розбитті кожного рядка на підрядки (n-грам) фіксованої довжини N (де N — додатне ціле число) та подальшому аналізі частоти їх входження у порівнюваних текстах. Рівень подібності між двома рядками визначається на основі ступеня перекриття множин n-грам або їхньої статистичної схожості.

Одиницею порівняння в цьому методі є послідовність символів або слів заданої довжини N. Значення N може варіюватися від 1 (порівняння окремих символів) до довжини найкоротшого слова у тексті (аналіз повних слів). Чим більше значення N, тим точніше здійснюється порівняння, оскільки жорсткіші умови відбору подібних слів. Проте зростання довжини n-грам призводить до зменшення гнучкості алгоритму та підвищення чутливості до незначних відмінностей у тексті 4.

Основним недоліком цього методу є його висока чутливість до помилок у середині слова. Одна невідповідність у символі може вплинути на кілька n-грам, що суттєво спотворює результати порівняння.

Незважаючи на це, метод N-грам є єдиним серед розглянутих алгоритмів, який може ефективно виявляти подібність між словами, що складаються з однакових частин, але розташованих у різному порядку. Наприклад, такі пари слів, як «графолог» – «логограф» або «логотип» – «типологія», будуть розпізнані як подібні, що є важливою перевагою даного підходу.

Схема алгоритму N-грам

1. Попередня обробка

- Розбивка текстів на слова або окремі символи залежно від необхідного рівня деталізації.
- Видалення стоп-слів (найуживаніших службових слів, таких як "the", "is" тощо) та проведення нормалізації тексту.

2. Генерація n-грам

- Формування множини всіх можливих n-грам (послідовностей довжиною N) для кожного текстового фрагмента.

3. Створення індексів N-грам

- Побудова індексної структури для збереження n-грам та їхніх позицій у вихідних рядках. Це дозволяє швидко здійснювати порівняння між текстами.

4. Обчислення частот

- Підрахунок кількості входжень кожної унікальної n-грами у рядках тексту.
- Використання спеціальних структур даних (словники, геш-таблиці, векторні моделі) для збереження цих частот.

5. Обчислення подібності

- Визначення міри подібності між двома текстами на основі статистичних методів, зокрема:
 - Коефіцієнта Жаккара (відношення перетину та об'єднання множин n-грам).
 - Косинусної подібності (аналізу кутової відстані між векторами частот n-грам).
 - Коефіцієнта Дайса (зваженої модифікації коефіцієнта Жаккара).

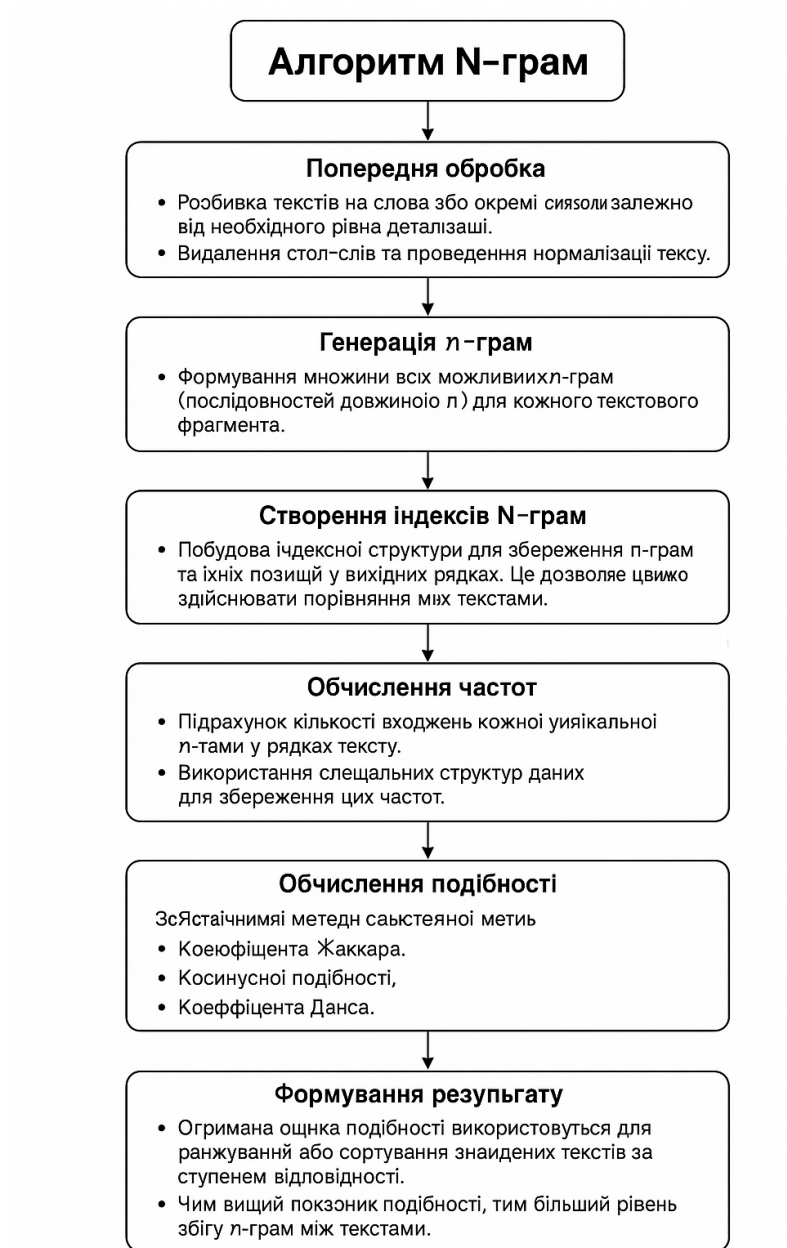
6. Формування результату

- Отримана оцінка подібності використовується для ранжування або сортування знайдених текстів за ступенем відповідності.
- Чим вищий показник подібності, тим більший рівень збігу n-грам між текстами.

Застосування алгоритму N-грам

Метод N-грам є одним із найефективніших підходів для обробки великих текстових масивів. Його можна застосовувати у таких завданнях:

- Пошук та дедуплікація текстових записів — метод ефективно виявляє дублікати текстових фрагментів навіть у разі незначних розбіжностей.
- Виявлення плагіату — використовується для порівняння наукових робіт, статей та іншого текстового контенту.
- Обробка природної мови (NLP) — покращення пошукових систем та автоматичне коригування запитів.
- Кластеризація текстів — групування документів за тематичною близькістю.



Блок схема рис. 1.3

Завдяки простоті реалізації та ефективності при обробці великих обсягів даних алгоритм N-грам залишається одним із найпопулярніших методів нечіткого пошуку в інформаційних системах. Однак для підвищення його точності він часто комбінується з іншими методами, такими як алгоритм Левенштейна, Джаро-Вінклера або SoundEx.

1.2.4 Алгоритми Bitap та SoundEx

Алгоритм Bitap (також відомий як алгоритм зсуву або алгоритм Баези-Йетса-Гоннета) є ефективним методом наближеного зіставлення рядків. Його основне призначення — перевірка наявності у заданому тексті підрядка, який «приблизно» відповідає заданому шаблону. Поняття «приблизної відповідності» визначається за допомогою відстані Левенштейна: якщо підрядок і шаблон знаходяться на заданій відстані k один від одного, алгоритм вважає їх еквівалентними [5].

Історія та розвиток алгоритму

Вперше ідея алгоритму була запропонована Рікардо Баезою-Йетсом і Гастоном Гоннетом у 1992 році. Оригінальна версія підтримувала лише операцію заміни символів, що фактично означало обчислення відстані Хемінга. Згодом Сунь Ву та Уді Манбер удосконалили алгоритм, додавши підтримку вставки та видалення символів, що дозволило використовувати його для розрахунку відстані Левенштейна.

Принцип роботи алгоритму

Робота алгоритму Bitap базується на побітових операціях та зсувах, що забезпечує швидке порівняння шаблону з текстом. Алгоритм складається з таких основних етапів:

1. Попередня обробка шаблону

1. Визначення унікальних символів рядків. Наприклад, для шаблону $S_0 = \text{"ababc"}$ і тексту $S_1 = \text{"abdabababc"}$ набір унікальних символів містить: a, b, c, d.
2. Побудова бітової маски для кожного символу.
 - Спочатку шаблон перевертається.
 - Для кожного символу створюється бітова маска: 0 розміщується там, де символ зустрічається в шаблоні, а 1 — в інших позиціях.

- Наприклад, для символу 'а' у шаблоні S0S_0S0 (позиції 3 та 5) бітова маска буде мати відповідні нулі у цих позиціях.

	c	b	a	b	a
T[a] =	1	1	0	1	0
	c	b	a	b	a
T[b] =	1	0	1	0	1
	c	b	a	b	a
T[c] =	0	1	1	1	1
	c	b	a	b	a
T[d] =	1	1	1	1	1

(Рисунок 1.4 – Бітові маски для символів a, b, c, d.)

2. Ініціалізація бітового шаблону

- Визначається початковий стан бітового шаблону довжиною $|S_0||S_0|$, де всі біти встановлені в 1.
- Для прикладу $S_0="ababc"$ початковий стан матиме вигляд:

$S=11111$

3. Пошук шаблону в тексті

1. Проходження по кожному символу рядка S1 із виконанням таких дій:
 - Зсув 0 у шаблоні S (праворуч на один біт).
 - Побітова операція між бітовою маскою та шаблоном $S = S \mid T[x]$.

Наприклад, для першого символу тексту $S_1="abdabababc"$, символу 'а':

1. Лівий зсув 0 у S: $11111 \rightarrow 11110$
2. Побітове АБО із бітовою маскою 'а': $11010 \mid 11110 = 11110$

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 1.5 – Результат роботи алгоритму Bitap для шаблону “ababc” у рядку “abdabababc”.)

Визначення точних та нечітких збігів

- Повний збіг шаблону з підрядком відбувається, коли 0 досягає найстаршого біта у бітовому шаблоні S.
- У наведеному прикладі шаблон повністю збігається із підрядком на останньому символі ‘с’ [6].
- Для нечіткого збігу потрібно спостерігати за позицією 0 у шаблоні S:
 - Якщо 0 знаходиться на четвертій позиції, це означає, що 4 з 5 символів у правильній послідовності.
 - Аналогічно, розташування 0 в інших позиціях вказує на різний рівень подібності [6].

Переваги та недоліки алгоритму

- Використання побітових операцій робить алгоритм дуже швидким у порівнянні з іншими методами пошуку.
- Простий в імплементації.
- Важкий для розуміння через використання побітових операцій.
- Ефективний лише для коротких рядків, оскільки його продуктивність зменшується при збільшенні довжини тексту.

Історія та застосування. Soundex був розроблений на початку XX століття і відтоді набув широкого застосування в різних сферах. Зокрема, його активно використовують у генеалогічних дослідженнях для відстеження варіантів прізвищ, які з часом могли змінювати написання, але зберегли подібне звучання. Крім того, алгоритм знайшов застосування в базах даних,

інформаційно-пошукових системах та розпізнаванні імен в адміністративних реєстрах.

Алгоритм Soundex належить до класу фонетичних методів нечіткого пошуку, які спрямовані на ідентифікацію слів, що мають схоже звучання, незалежно від варіативності їх написання. Основна ідея алгоритму полягає в перетворенні слів у стандартизований код на основі їхньої фонетичної структури, після чого порівняння виконується саме між цими кодами.

Функціонування алгоритму Soundex включає кілька етапів:

1. Попередня обробка тексту

- Видаляються початкові та кінцеві пробіли.
- Усі літери перетворюються у верхній або нижній регістр для уніфікації.

2. Формування початкового коду

- Перша літера слова зберігається без змін.
- Всі наступні входження голосних (A, E, I, O, U) та літер H, W, Y відкидаються.

3. Кодування приголосних відповідно до їхнього звучання

- B, F, P, V $\rightarrow$ 1
- C, G, J, K, Q, S, X, Z $\rightarrow$ 2
- D, T $\rightarrow$ 3
- L $\rightarrow$ 4
- M, N $\rightarrow$ 5
- R $\rightarrow$ 6

4. Об'єднання символів у код

- До першої літери додаються числові коди, отримані на попередньому кроці.

5. Оптимізація коду

- Послідовні однакові цифри в коді скорочуються до одного входження (за винятком першої літери).

6. Форматування довжини коду

- Якщо отриманий код містить менше чотирьох символів, додаються нулі (0) до потрібної довжини.
- Якщо код перевищує чотири символи, він обрізається до стандартної довжини.

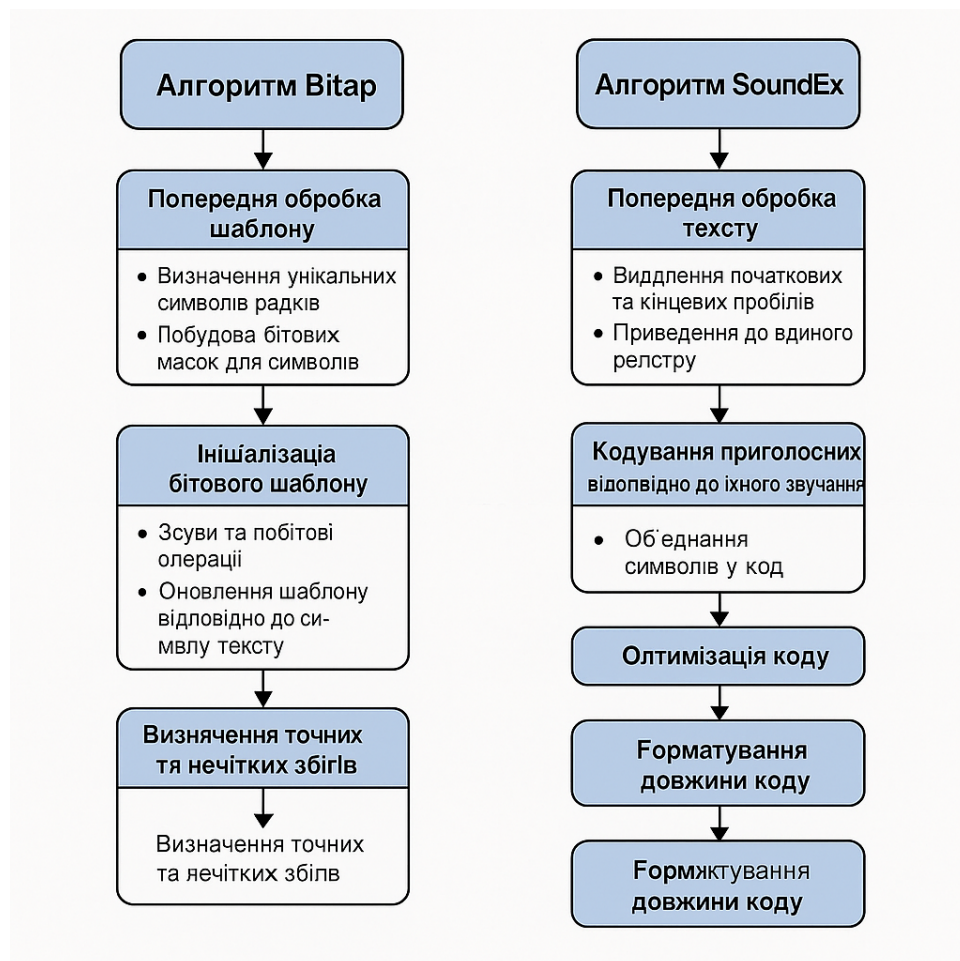


Рис.1.6

Результат роботи алгоритму

У підсумку Soundex генерує чотирьохсимвольний код, який використовується для порівняння фонетично подібних слів або імен. Завдяки цьому алгоритм дозволяє знаходити схожі за звучанням слова навіть у разі наявності орфографічних варіацій або друкарських помилок.

Ця технологія залишається ефективною у випадках, коли точний збіг не є можливим, і сприяє покращенню якості інформаційного пошуку, особливо у великих текстових масивах та базах даних.

1.3 Розширена постановка задачі

У цьому розділі визначаються конкретні вимоги до розроблюваного програмного забезпечення (ПЗ), яке реалізовуватиме алгоритм нечіткого пошуку. Описані функціональні та нефункціональні вимоги, технічні характеристики, а також параметри функціонування.

Програмне забезпечення повинно реалізовувати алгоритм нечіткого пошуку текстових рядків на основі таблиці подібності символів. Основна мета системи – покращити точність пошуку в текстових масивах, навіть якщо введені дані містять помилки, варіативність написання або неточності.

Система має бути придатною для інтеграції в:

- Пошукові системи;
- Системи перевірки орфографії;
- Автоматизовані системи аналізу текстових даних;
- Бази даних з великою кількістю записів.

Програма повинна реалізовувати наступні основні функції:

- Підтримка пошуку тексту з орфографічними помилками.

- Можливість знаходження схожих слів навіть при частковій відповідності.
- Використання методу порівняння тексту на основі відстані Левенштейна, Джаро-Вінклера або N-грам.
- Підтримка таблиці подібності символів для коригування можливих варіантів введення.

Обробка запитів

- Підтримка введення текстових запитів через інтерфейс користувача або API.
- Автоматична нормалізація запиту (переведення до нижнього регістру, видалення спеціальних символів, лематизація).
- Визначення відстані редагування між запитом та наявними даними.

Управління таблицею подібності символів

- Можливість додавання, редагування та видалення символів у таблиці подібності.
- Використання попередньо визначених таблиць подібності для мов із різними системами письма.

Результати пошуку

- Сортування результатів за рівнем схожості.
- Налаштування порогу схожості для фільтрації нерелевантних результатів.
- Виведення знайдених збігів у зручному вигляді.

Інтерфейс користувача

- Графічний або веб-інтерфейс для введення тексту та перегляду результатів.
- Підтримка введення параметрів пошуку (поріг схожості, вибір алгоритму).
- Візуалізація знайдених результатів у вигляді списку.

Нефункціональні вимоги

Програма повинна відповідати наступним нефункціональним вимогам:

- Можливість роботи як локально (десктопне застосування), так і в хмарному середовищі (веб-додаток або API).
- Легке розширення функціональності (підключення нових алгоритмів, оновлення таблиць подібності).
- Система повинна коректно обробляти некоректні запити.
- Автоматичне коригування помилок при введенні тексту.
- Можливість запуску на Windows, Linux, macOS.
- При необхідності – веб-версія із запуском на будь-якому браузері.
- Захист від SQL-ін'єкцій та інших типів атак (для веб-версії).
- Обмеження доступу до редагування таблиці подібності символів.

Вимоги до програмної та апаратної платформи

- Мови програмування: Python
- Використання бази даних SQLite або PostgreSQL для зберігання таблиці подібності символів.
- Фреймворки для веб-інтерфейсу: Flask/Django (Python) або Spring Boot (Java).

Апаратні вимоги

- Рекомендовані:
 - Процесор: Intel Core i5 / AMD Ryzen 5 або вище.
 - Оперативна пам'ять: 8 ГБ.
 - Вільне місце на диску: 500 МБ.

5. Макети інтерфейсу користувача представлено на рис. 2.1

Головний екран пошуку

Елементи інтерфейсу:

- Поле введення тексту (запит користувача).
- Кнопка "Знайти".
- Поле для вибору методу пошуку (Левенштейн, Джаро-Вінклер, N-грам).
- Список знайдених варіантів.



Нечіткий пошук релевантної інформації

Введіть пошуковий запит:

Наприклад, World

Завантажити файл (необов'язково):

Выберите файл

Файл не выбран

☐ Використовувати таблицю подібності

Алгоритм:

Damerau-Levenshtein ▼

Кількість результатів: 10

Пошук

Результати:

№	Вхідний рядок	Знайдений рядок	Відстань	Співпадіння (%)
---	---------------	-----------------	----------	-----------------

Рис.2.1

Налаштування таблиці подібності

Елементи інтерфейсу:

- Таблиця з колонками: "Символ", "Подібні символи".
- Поля для додавання нового символу.
- Кнопка "Зберегти зміни".

Символ	Подібні символи
-----	-----
a	a, ă, â, à, á, ä
e	e, é, è, ê, ë

| и | и, й, ї, і |

[Додати новий символ] [Зберегти]

Висновки до Частини I

У ході дослідження теоретичних основ нечіткого пошуку було встановлено, що сучасні алгоритми, спрямовані на знаходження приблизних відповідностей у текстових даних, відіграють ключову роль у системах інформаційного пошуку, автоматизованих засобах корекції тексту, базах даних та обробці природної мови.

Аналіз розглянутих алгоритмів продемонстрував їхню варіативність та адаптивність до різних задач. Так, алгоритм Дамерау-Левенштейна є одним із найефективніших методів для оцінки схожості між текстовими рядками, оскільки враховує чотири основні операції редагування: вставку, видалення, заміну та транспозицію символів. Дослідження показали, що саме ці типи помилок є найпоширенішими в текстових даних, що пояснює високу ефективність алгоритму у виправленні орфографічних неточностей.

Водночас метрика Джаро-Вінклера виявилася корисною для аналізу текстових послідовностей, які відрізняються лише незначними змінами у порядку символів. Вона демонструє особливу ефективність у порівнянні імен, назв та інших текстових фрагментів, що містять типові друкарські помилки або різні варіанти написання.

Дослідження алгоритму N-грам дозволило встановити, що цей метод є ефективним для аналізу текстових даних у випадках, коли необхідно оцінити структурну подібність рядків. Використання фрагментів тексту стало

альтернативним підходом до пошуку подібностей, що дозволяє компенсувати недоліки символ-орієнтованих методів. Проте, було виявлено, що чутливість алгоритму до змін у середині слова може впливати на точність результатів, що потребує додаткової оптимізації.

Важливе місце серед алгоритмів нечіткого пошуку займає Bitap, який забезпечує швидке та ефективне зіставлення підрядків у великих текстових масивах завдяки використанню побітових операцій. Однак його продуктивність обмежується довжиною вхідних рядків, що ускладнює його застосування в деяких сценаріях, пов'язаних із масштабною обробкою даних.

Окрему групу алгоритмів представляють фонетичні методи, зокрема Soundex, який перетворює текстові дані в уніфікований код на основі їхнього звучання. Це дозволяє ефективно знаходити слова, що мають подібне фонетичне оформлення, але різні варіанти написання. Хоча цей підхід добре працює для англomовних слів, його застосування в мовах зі складнішою фонетичною системою потребує доопрацювання.

Загальний аналіз алгоритмів нечіткого пошуку показав, що кожен із розглянутих методів має як сильні сторони, так і певні обмеження. У зв'язку з цим доцільним є використання комбінованих підходів, які дозволяють компенсувати недоліки окремих методів та підвищити точність пошуку. Такі гібридні моделі, що поєднують традиційні алгоритми редагування з методами машинного навчання та семантичного аналізу, можуть суттєво покращити якість інформаційного пошуку та зробити його більш гнучким і адаптивним до реальних умов роботи з текстовими даними.

ЧАСТИНА II. РОЗРОБКА АЛГОРИТМУ НЕЧІТКОГО ПОШУКУ

У другій частині дипломної роботи основна увага приділяється розробці, реалізації та інтеграції модифікованого алгоритму нечіткого пошуку. На основі аналізу існуючих методів, розглянутих у попередньому розділі, буде запропоновано власний підхід до нечіткого зіставлення текстових рядків, що враховує не лише класичні метрики редагування, а й додаткові параметри подібності символів.

Алгоритм передбачає проектування адаптивної моделі, яка дозволяє компенсувати типові помилки введення, включаючи орфографічні неточності, варіації кодування та фонетичні спотворення. Одним із ключових елементів є таблиця подібності символів, що підвищує гнучкість при зіставленні рядків і сприяє точнішому пошуку релевантної інформації в текстових базах.

Кожен з розділів містить опис відповідного етапу реалізації, з акцентом на технічні рішення, алгоритмічну логіку, програмні засоби та прикладні результати. Це дозволить комплексно представити процес створення функціонального інструменту нечіткого пошуку від концепції до впровадження.

2.1 Проектування та модифікація алгоритму нечіткого пошуку

Проектування алгоритму нечіткого пошуку базується на потребі врахування неточностей у вхідному запиті користувача. Основним завданням є розробка підходу, здатного ефективно виявляти подібні рядки навіть за умови наявності орфографічних помилок, перестановок символів або варіантів написання. Для цього було обрано комбінований підхід, що включає

використання класичних метрик редагування рядків у поєднанні з таблицею подібності символів.

Основні етапи проектування включають:

- Формалізацію вимог до алгоритму (гнучкість, адаптивність, масштабованість);
- Вибір основної метрики схожості (Дамерау–Левенштейна);
- Визначення структури таблиці подібності символів;
- Розробку логіки врахування подібних символів у процесі обчислення відстані;
- Оптимізацію обчислень для підвищення продуктивності.

Модифікація класичного алгоритму редагування полягає у введенні додаткових ваг для заміни символів. Якщо символи є подібними згідно з таблицею, вага заміни зменшується, що дозволяє алгоритму трактувати такі відхилення як менш значущі. Це підвищує чутливість алгоритму до реальних помилок, часто допущених користувачами, особливо в мультимовному середовищі.

Додатково передбачено параметричну гнучкість, що дозволяє користувачеві або системі налаштовувати рівень допуску помилок, вибір конкретної метрики, а також тип символів, які вважаються взаємозамінними. Такий підхід робить алгоритм не лише точним, але й універсальним для різних мов і прикладних сценаріїв.

Результатом проектування є узагальнена схема обчислення нечіткої відповідності, яка може бути реалізована в програмному середовищі з можливістю масштабування та інтеграції в більші інформаційно-пошукові системи.

На сьогоднішній день розроблено чимало варіантів реалізації нечіткого пошуку у текстових даних, проте значна частина з них ігнорує графічну або семантичну близькість між окремими символами. Саме для вирішення цього завдання доцільним є використання таблиці подібності символів. Її суть полягає у тому, щоб кожній парі символів на основі їх візуальної чи змістової схожості призначити певний коефіцієнт подібності. Такі значення дозволяють об'єктивно оцінювати рівень збігу між символами та застосовувати їх для ефективного порівняння при пошуку.

2.2 Реалізація алгоритму та таблиці подібності символів

У задачах нечіткого пошуку одним із ключових інструментів є таблиця подібності символів — спеціальна структура даних, яка дозволяє чисельно оцінювати ступінь схожості між окремими символами. Вона слугує базою для порівняння текстових рядків, що містять незначні розбіжності у написанні, та дає змогу алгоритму краще розпізнавати варіативні написання одних і тих самих слів.

Основна ідея таблиці полягає в тому, щоб кожній парі символів зіставити коефіцієнт подібності — числове значення, яке відображає рівень їхньої схожості. Такі коефіцієнти можуть базуватися на візуальній подібності (наприклад, символи О та 0 мають майже однакову форму), фонетичній близькості (s і z в англійській мові можуть звучати схоже) або навіть семантичному контексті, якщо символи мають однакове значення в різних мовах.

У класичному варіанті таблиця подібності реалізується у вигляді квадратної матриці, де рядки й стовпці представляють усі можливі символи, а значення в комірках — це числові оцінки подібності між ними. Такий підхід є

інтуїтивно зрозумілим і зручним для візуального сприйняття, але має низку недоліків:

потребує значного обсягу пам'яті при роботі з великими алфавітами;

ускладнює динамічне оновлення значень;

пошук значення для конкретної пари символів потребує сканування матриці або знання їхніх індексів.

Щоб уникнути цих обмежень, у цій реалізації використовується оптимізована структура на базі вкладених словників (dict), яка значно підвищує швидкодію системи.

Структура таблиці

Тому як експеримент розроблена наступна таблиця (рис. 2.2).

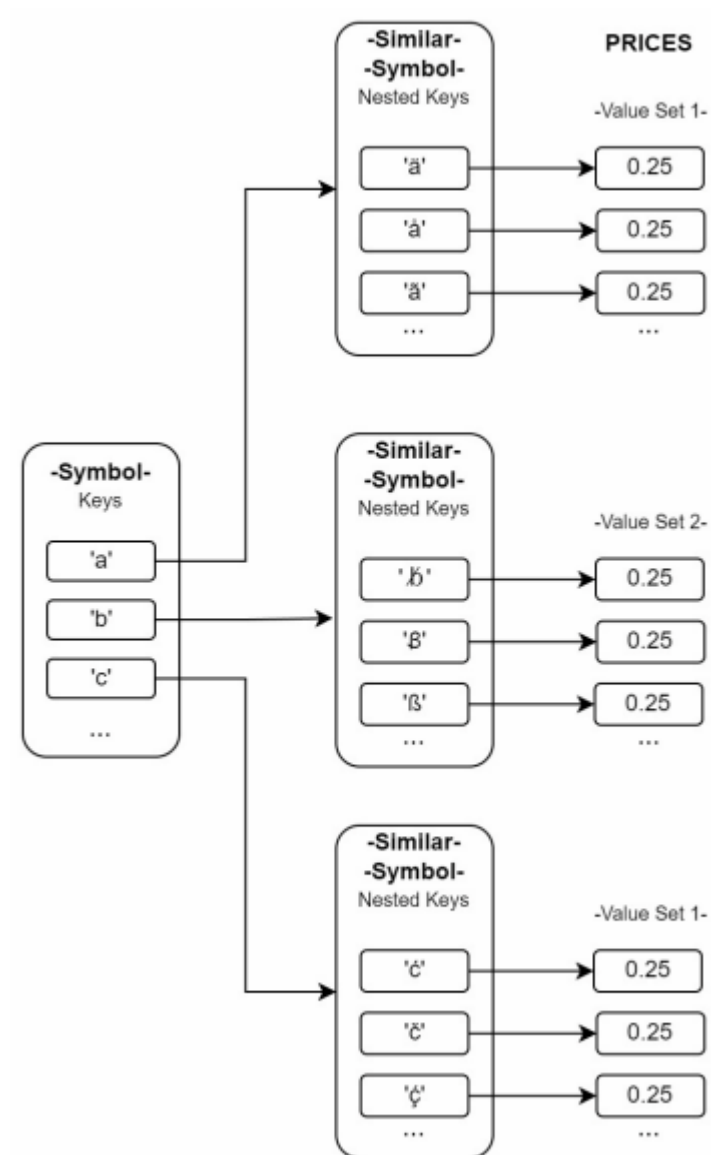


Рисунок 2.2 – Структура таблиці подібності символів.

Нова структура передбачає використання двох рівнів вкладеності:

Перший словник (зовнішній): ключ — символ, для якого визначається набір подібних;

Другий словник (вкладений): ключ — подібний символ; значення — вага або коефіцієнт подібності.

{
 'a': { 'à': 0.25, 'á': 0.25, 'â': 0.25 },

$$\begin{aligned} & 'b': \{ 'B': 0.25, 'p': 0.25 \}, \\ & 'c': \{ 'č': 0.25, 'ç': 0.25 \} \\ & \} \end{aligned}$$

Ця структура дозволяє швидко знайти всі подібні символи до заданого та легко налаштовується під конкретну мову або систему кодування. Незважаючи на те, що симетричність (наприклад, якщо *á* подібна до *a*, то і *a* подібна до *á*) реалізується шляхом дублювання інформації, це спрощує доступ до даних і дозволяє підтримувати час доступу $O(1)$.

Цю таблицю можна легко доповнювати, змінювати ваги, адаптувати під різні мови, пошук значення подібності між символами виконується миттєво завдяки хеш-таблицям.

Таблицю легко інтегрувати в алгоритми Левенштейна, Джаро-Вінклера, Вітар тощо.

У рамках реалізації алгоритму нечіткого пошуку, побудова ефективної таблиці подібності символів потребує структурованого підходу. Для цього запропоновано розподілити символи за категоріями, які відображають різні типи подібності. Така класифікація дозволяє підвищити точність пошуку та адаптивність алгоритму до мультимовного середовища.

Категорія 1. Семантична подібність між символами з різних мов

У першу категорію включено символи, які мають семантичну або функціональну схожість, незважаючи на їх приналежність до різних алфавітів або мовних систем. Наприклад, латинська літера *a* та кирилична *а* мають однакове звучання й подібну форму, що робить їх практично взаємозамінними у багатьох випадках, особливо при обробці запитів з орфографічними помилками.

Для створення цієї категорії було використано ресурс [9], який містить узагальнену таблицю відповідностей символів між різними мовами. Даний електронний ресурс дозволяє автоматизувати процес побудови таблиці через програмний аналіз його вмісту.

З технічної точки зору, оптимальним варіантом зберігання таблиці подібності є формат JSON (JavaScript Object Notation). Цей формат забезпечує:

зручну структуру у вигляді вкладених пар “ключ-значення”, що легко модифікується;

серіалізацію і десеріалізацію даних без втрат структури;

кросплатформену сумісність і широке застосування в системах з обробки тексту.

Завдяки мові програмування Python таблиця може бути створена та наповнена автоматично на основі обробки відкритих джерел або текстових файлів. Вбудовані засоби роботи з JSON (`json.dump`, `json.load`) дозволяють зберігати структуру, що складається з ключів (основних символів) та вкладених об'єктів (подібних символів з вагами подібності).

На рисунку 2.3 наведено фрагмент програмного коду, який ілюструє створення першої категорії таблиці подібності символів у форматі JSON. У цьому прикладі використовується вкладена структура словників, де:

ключ — це символ з основного алфавіту;

значення — словник подібних символів з відповідними коефіцієнтами (вагами) подібності.

```

1. import json
2.
3. # Бартісна модель
4. COST_DICT = {
5.     "SIMILAR_CHAR_COST": 0.3,
6.     "KEYBOARD_MISTAKE_COST": 0.7,
7.     "VISUAL_SIMILAR_CHAR_COST": 0.5
8. }
9.
10. data = {"data": {}}
11.
12. # === Категорія 1: Семантично подібні символи (UA + EN) ===
13. semantic_groups = {
14.     "a": ["a", "à", "á"],
15.     "b": ["b"],
16.     "c": ["c"],
17.     "e": ["e", "é", "e"],
18.     "i": ["i", "ì", "í"],
19.     "o": ["o", "ö"],
20.     "p": ["p"],
21.     "x": ["x"],
22.     "y": ["y"],
23.     "g": ["r"],
24.     "u": ["ü"],
25.     "z": ["z"]
26. }
27.
28. for letter, group in semantic_groups.items():
29.     similar_group = set(group)
30.     similar_group.add(letter)
31.     for lt1 in similar_group:
32.         if lt1 not in data["data"]:
33.             data["data"][lt1] = {}
34.             for lt2 in similar_group:
35.                 if lt1 != lt2:
36.                     data["data"][lt1][lt2] = COST_DICT["SIMILAR_CHAR_COST"]

```

Рисунок 2.3 фрагмент програмного коду, створення першої категорії таблиці подібності символів у форматі JSON

Така структура легко масштабована: можна додавати нові категорії символів, мови або адаптувати ваги в залежності від цільового завдання (наприклад, пошукова система, перевірка правопису або машинний переклад).

Категорія 2. Типові друкарські помилки на основі фізичної близькості клавіш

Другу категорію подібності символів формуємо на основі ймовірних друкарських помилок, які користувач може зробити під час введення тексту на клавіатурі. Такі помилки зазвичай виникають внаслідок механічного промаху, коли замість потрібного символу натискається сусідній — через фізичну близькість клавіш.

Принцип побудови

Основна ідея полягає в тому, щоб для кожного символу англійського алфавіту (у межах стандартної QWERTY-розкладки) визначити усі клавіші, які його оточують. Кожен з цих оточуючих символів вважається потенційною помилкою при наборі. Для підвищення точності моделі, кожному з таких символів присвоюється знижена вага подібності (наприклад, 0.6–0.8), що сигналізує про їхню невисоку, але релевантну схожість.

Щоб створити таблицю цієї категорії програмно, спочатку моделюється розкладка клавіатури у вигляді двовимірного масиву:

```
keyboard = [  
    list("qwertyuiop"),  
    list("asdfghjkl"),  
    list("zxcvbnm")  
]
```

Потім для кожної літери ми ідентифікуємо **усіх її сусідів** у межах одного кроку по горизонталі, вертикалі та діагоналі. Ці сусіди і вважаються потенційними варіантами помилкового введення.

Нижче наведено фрагмент коду, який ілюструє реалізацію побудови другої категорії таблиці подібності символів (рис. 2.4):

```

38. # === Категорія 2: Помилки на клавіатурі (QWERTY+ЙЦУКЕН) ===
39. # ЙЦУКЕН (українська розкладка)
40. keyboard_ua = [
41.     list("йцукентгшщзхї"),
42.     list("фівাপролджє"),
43.     list("ячсмитьбю")
44. ]
45.
46. # QWERTY (англійська)
47. keyboard_en = [
48.     list("qwertyuiop"),
49.     list("asdfghjkl"),
50.     list("zxcvbnm")
51. ]
52.
53. def add_to_chars(char, *ch_set):
54.     if char not in data["data"]:
55.         data["data"][char] = {}
56.     for ch in ch_set:
57.         data["data"][char][ch] = COST_DICT["KEYBOARD_MISTAKE_COST"]
58.
59. def process_keyboard_layout(layout):
60.     for row_idx, row in enumerate(layout):
61.         for col_idx, key in enumerate(row):
62.             neighbors = []
63.             for dr in [-1, 0, 1]:
64.                 for dc in [-1, 0, 1]:
65.                     nr, nc = row_idx + dr, col_idx + dc
66.                     if 0 <= nr < len(layout) and 0 <= nc < len(layout[nr]):
67.                         neighbor = layout[nr][nc]
68.                         if neighbor != key:
69.                             neighbors.append(neighbor)
70.             add_to_chars(key, *neighbors)
71.
72. process_keyboard_layout(keyboard_en)
73. process_keyboard_layout(keyboard_ua)

```

Рис. 2.4 Реалізація побудови другої категорії таблиці подібності символів

У результаті для кожного символу створюється словник суміжних клавіш із вагою 0.7, яка позначає середній рівень ймовірності помилкового натискання. Отримана структура ідеально підходить для інтеграції в загальну таблицю подібності символів, представлену у форматі JSON.

Варіант реалізації

```
{
    "e": { "w": 0.7, "r": 0.7, "s": 0.7, "d": 0.7 },
    "d": { "s": 0.7, "f": 0.7, "e": 0.7, "c": 0.7 },
    "m": { "n": 0.7, "k": 0.7 }
}
```

Категорія 3. Візуально схожі символи

Третя категорія таблиці подібності символів охоплює візуальну подібність між літерами англійського алфавіту, тобто ті символи, які мають схожу графічну форму, і, відповідно, можуть бути сплутані при візуальному або машинному розпізнаванні (рис. 2.5).

```
75. # === Категорія 3: Візуально подібні символи (UA+EN) ===
76. visual_groups = [
77.     ("b", "d", "p", "q", "o"),
78.     ("i", "l", "j", "1", "t"),
79.     ("u", "v", "w", "y"),
80.     ("m", "n", "и"),
81.     ("c", "e", "o", "s", "e"),
82.     ("h", "к", "k"),
83.     ("f", "т", "t"),
84.     ("g", "r", "q"),
85.     ("x", "y", "x"),
86.     ("a", "o", "a"),
87.     ("p", "p"),
88.     ("n", "и", "v", "x")
89. ]
90.
91. def add_similar_ch(chs):
92.     for ch1 in chs:
93.         if ch1 not in data["data"]:
94.             data["data"][ch1] = {}
95.         for ch2 in chs:
96.             if ch1 == ch2:
97.                 continue
98.             current = data["data"][ch1].get(ch2, float("inf"))
99.             data["data"][ch1][ch2] = min(current, COST_DICT["VISUAL_SIMILAR_CHAR_COST"])
100.
101. for group in visual_groups:
102.     add_similar_ch(group)
103.
104. # === Збереження у файл ===
105. with open("full_similarity_table_ua_en.json", "w", encoding="utf-8") as f:
106.     json.dump(data, f, ensure_ascii=False, indent=2)
```

Рисунок 2.5 – Фрагмент програмного коду, що відповідає за створення третьої категорії символів у таблиці подібності.

У межах даної категорії реалізовано спеціальну функцію, яка приймає на вхід набір візуально схожих символів (параметр `chs`). За допомогою двох вкладених циклів вона проходить усі комбінації символів у цьому наборі та записує у словник `data` відношення подібності між ними.

Кожен символ із цього набору отримує посилання на всі інші як на подібні. Це дозволяє забезпечити двосторонню симетрію відношення подібності, що є важливим у нечітких алгоритмах.

Особливістю реалізації є перевірка вже наявних записів у словнику. Якщо для певної пари символів вже збережена вага подібності (отримана раніше з іншої групи), то система обирає менше значення (тобто більш «схожий» варіант), забезпечуючи найоптимальнішу оцінку з точки зору подібності.

Після завершення побудови зв'язків між символами, сформований словник `data` перетворюється у формат JSON із використанням стандартної бібліотеки `json`. Результат зберігається у зовнішньому файлі, що дозволяє зручно інтегрувати його в програмні компоненти для подальшого використання під час нечіткого пошуку текстових збігів.

Повний код функції створення цієї категорії представлений у Додатку А.

2.3 Реалізація алгоритму нечіткого пошуку Дамерау-Левенштейна

Для реалізації алгоритму нечіткого пошуку з урахуванням подібності символів обрано мову програмування **Python** завдяки її широким можливостям у роботі з текстом, Unicode-символами та простоті серіалізації JSON-структур.

Уся логіка роботи поділяється на дві ключові частини:

1. Завантаження та обробка таблиці подібності символів;
2. Реалізація алгоритму Дамерау–Левенштейна з підтримкою таблиці.

Завантаження таблиці подібності символів

JSON-файл, що містить таблицю подібності, зчитується у вигляді вкладеного словника (dict of dicts), де кожен ключ – це символ, а значенням є словник його подібних символів з відповідними вагами подібності.

Окрім цього, окремо зчитуються типові ваги для спеціальних умов, таких як CAPITAL_COST – вартість різниці між великими та малими літерами.

Структура таблиці:

- data – словник, де кожен символ пов’язаний із подібними до нього та їхньою вагою.
- cost – словник із глобальними константами типу CAPITAL_COST, що використовується для регістрозалежних порівнянь.

Код для завантаження таблиці подібності (SimTable) та розрахунку вартості заміни вказано на рисунку 2.6:

```

11.         """Завантаження JSON-файлу з таблицею подібності"""
12.     try:
13.         with open(path, 'r', encoding='utf-8') as f:
14.             data = json.load(f)
15.
16.             self.cost = data.get("cost", {})
17.             self.table = data.get("data", {})
18.
19.     except FileNotFoundError:
20.         raise Exception(f"файл не знайдено: {path}")
21.     except json.JSONDecodeError:
22.         raise Exception("файл не є валідним JSON")
23.
24.     def get_replace_cost(self, c1: str, c2: str) -> Optional[float]:

```

Рисунок 2.6 – Фрагмент програмного коду, що відповідає за завантаження символів у таблиці подібності.

Даний фрагмент коду реалізує завантаження таблиці подібності символів з зовнішнього JSON-файлу, який надалі використовується в алгоритмах нечіткого пошуку, зокрема при розрахунку вартості заміни символів.

На початку, у вигляді докстрінгу, вказано призначення методу – зчитування таблиці подібності. Операція зчитування обгорнута в конструкцію `try`, що дозволяє безпечно обробляти потенційні помилки, пов’язані з файлом.

Використовується вбудована функція `open()` з параметрами `'r'` (режим читання) та `'utf-8'` (кодування), що забезпечує коректне зчитування Unicode-символів. Далі за допомогою `json.load(f)` вміст файлу перетворюється у словник Python, який містить усі необхідні дані.

Із отриманого словника витягуються два основні блоки інформації:

`cost` — словник, у якому зберігаються стандартні ваги типів подібності (наприклад: `CAPITAL_COST` для великих літер);

data — основна таблиця подібності символів, яка реалізована як вкладена структура: кожен символ має підмножину подібних до нього символів із відповідними вагами.

Якщо файл за вказаним шляхом не знайдено (`FileNotFoundError`), буде згенеровано виняток з поясненням. Аналогічно, якщо файл відкрився, але не є валідним JSON-документом (`json.JSONDecodeError`), також виникне відповідна помилка.

У завершальній частині наведено початок функції `get_replace_cost`, яка дозволяє визначити вагу заміни одного символу на інший з урахуванням таблиці. Якщо символи однакові — повертається 0.0; якщо різні, але є в таблиці — повертається відповідна вага; якщо один із них у верхньому регістрі — до ваги може додаватись `CAPITAL_COST`.

```
25.         """Обчислення вартості заміни символів з урахуванням подібності та регістру
26.         if c1 == c2:
27.             return 0.0
28.
29.         is_upper_c1 = c1.isupper()
30.         is_upper_c2 = c2.isupper()
31.
32.         lc1 = c1.lower()
33.         lc2 = c2.lower()
34.
```

Рисунок 2.7 – Фрагмент програмного коду, для визначення вартості заміни одного символу іншим з урахуванням їх подібності та регістру.

Першим кроком виконується перевірка, чи два вхідні символи (`c1` і `c2`) є ідентичними. Якщо так, то функція одразу повертає значення 0.0, що означає відсутність будь-якої потреби в заміні, оскільки символи збігаються повністю, включаючи регістр. Такий механізм дозволяє швидко завершити обчислення у випадках, коли подібність повна.

Далі функція визначає, чи є кожен символ великим, використовуючи метод `.isupper()`. Це важливо для подальшої логіки, оскільки різниця між великим і малим варіантом символу може враховуватись додатковим штрафом (наприклад, `CAPITAL_COST`) під час оцінки їхньої схожості.

Щоб знехтувати впливом регістру при основному порівнянні, обидва символи переводяться до нижнього регістру методом `.lower()`. Це дозволяє більш гнучко аналізувати символи з різним регістром, але однаковою літерою за змістом (наприклад, `A` і `a`, `I` і `i`).

Алгоритм Дамерау–Левенштейна з підтримкою таблиці реалізується як розрахунок матриці відстаней (динамічне програмування). У цій версії враховано:

- Вставки;
- Видалення;
- Заміни з урахуванням таблиці подібності;
- Транспозиції сусідніх символів.

На рисунку 2.8 функція обчислення відстані:

```

1. def damerau_levenshtein(s1: str, s2: str, sim_table: Optional[SimTable] = None) ->
2.     """Повертає варту редагування між s1 і s2 з урахуванням таблиці подібності"""
3.     len1 = len(s1)
4.     len2 = len(s2)
5.
6.     dp = [[0.0] * (len2 + 1) for _ in range(len1 + 1)]
7.
8.     for i in range(len1 + 1):
9.         dp[i][0] = float(i)
10.    for j in range(len2 + 1):
11.        dp[0][j] = float(j)
12.
13.    for i in range(1, len1 + 1):
14.        for j in range(1, len2 + 1):
15.            cost = 1.0
16.            if sim_table:
17.                replacement_cost = sim_table.get_replace_cost(s1[i - 1], s2[j - 1])
18.                if replacement_cost is not None:
19.                    cost = replacement_cost
20.                elif s1[i - 1] == s2[j - 1]:
21.                    cost = 0.0
22.            else:
23.                if s1[i - 1] == s2[j - 1]:
24.                    cost = 0.0
25.
26.            dp[i][j] = min(
27.                dp[i - 1][j] + 1.0,      # видалення
28.                dp[i][j - 1] + 1.0,      # вставка
29.                dp[i - 1][j - 1] + cost # заміна
30.            )
31.
32.            if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2] == s2[j - 1]:
33.                dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + cost) # транспозиція
34.
35.    return dp[len1][len2]

```

Рисунок 2.8 – Фрагмент програмного коду, для визначення вартості заміни одного символу іншим з урахуванням їх подібності та регістру.

Алгоритм Дамерау–Левенштейна, реалізований у наведеній функції `damerau_levenshtein`, слугує для обчислення нечіткої відстані редагування між двома рядками. Особливістю цієї версії є підтримка таблиці подібності символів (`sim_table`), яка дозволяє враховувати не лише точну відповідність між символами, а й їхню семантичну, візуальну чи графічну подібність. Це значно підвищує ефективність пошуку в умовах помилок вводу, варіативності написання або різних мов.

На початку функції визначаються довжини обох рядків $s1$ і $s2$, після чого створюється двовимірна таблиця dp , яка використовується для реалізації алгоритму динамічного програмування. Ця таблиця має розміри $(len1 + 1) \times (len2 + 1)$ і на початковому етапі заповнюється нулями. Вона слугує для поетапного накопичення мінімальної кількості операцій, необхідних для перетворення одного рядка в інший.

Далі виконується ініціалізація першого рядка та першого стовпця матриці dp , що відповідає вартості перетворення порожнього рядка у символи рядка $s1$ або $s2$ через послідовність вставок чи видалень.

У вкладених циклах відбувається основна логіка порівняння символів між рядками. На кожному кроці аналізується пара символів $s1[i - 1]$ та $s2[j - 1]$. Базове значення вартості заміни $cost$ встановлюється як 1.0. Якщо передано таблицю подібності, то здійснюється виклик методу `get_replace_cost()`, який повертає вартість заміни між двома символами з урахуванням їхньої схожості. Якщо символи однакові або мають дуже високу подібність, вартість буде зниженою або дорівнюватиме нулю.

Після цього вибирається мінімальна вартість однієї з трьох можливих операцій — видалення, вставки або заміни — і відповідне значення заноситься до комірки $dp[i][j]$.

Окремо обробляється можливість транспозиції — перестановки двох сусідніх символів. Якщо символи в обох рядках на позиціях $i - 1$, $j - 2$ збігаються з символами на позиціях $i - 2$, $j - 1$, тоді перевіряється варіант редагування із використанням транспозиції, і обирається мінімальне значення серед усіх варіантів.

У фіналі функція повертає значення з нижньої правої комірки матриці $dp[len1][len2]$, яке і є шуканою відстанню редагування з урахуванням подібності символів.

Висновки до Частини II

У другому розділі дипломної роботи здійснено комплексну розробку і реалізацію модифікованого алгоритму нечіткого пошуку, орієнтованого на підвищення точності та адаптивності порівняння текстових рядків. В основі розробки лежить вдосконалена версія алгоритму Дамерау–Левенштейна, яка доповнена врахуванням подібності символів завдяки спеціально побудованій таблиці.

Під час проектування алгоритму враховано реальні типи помилок, які виникають при введенні даних користувачем: орфографічні, фонетичні, кодувальні й механічні. Для цього було створено багаторівневу таблицю подібності символів, що включає три основні категорії:

1. Семантичну відповідність символів з різних мов (наприклад, латинська "a" і кирилична "а");
2. Ймовірні друкарські помилки на основі фізичної близькості клавіш у QWERTY-розкладці;
3. Візуальну подібність символів, що дозволяє враховувати графічну схожість при розпізнаванні.

Ця таблиця реалізована у форматі JSON з використанням вкладених словників Python, що забезпечує зручність серіалізації/десеріалізації та високу швидкодію пошуку. У структурі таблиці для кожного символу задаються подібні до нього символи з відповідними вагами подібності, що дозволяє точно моделювати різні варіанти невірного введення.

У результаті було розроблено алгоритм, який:

- підтримує основні операції редагування: вставка, видалення, заміна, транспозиція;

- враховує не лише абсолютну, але й відносну подібність символів, що є ключовим для пошуку в реальних текстах, особливо у мультимовному середовищі;
- забезпечує гнучке налаштування параметрів алгоритму (допуск на помилки, варіанти подібних символів, ваги операцій).

Завдяки використанню Python як основної мови реалізації, вдалося досягти високої швидкості розробки, легкої інтеграції з іншими системами та зручної роботи з кодуванням Unicode, що є особливо важливим для обробки текстів різними мовами.

Другий розділ демонструє повноцінний перехід від концепції до практичної реалізації інтелектуального механізму нечіткого пошуку, який має широке застосування у сферах обробки природної мови, пошукових систем, фільтрації помилок, перевірки орфографії та систем підтримки введення тексту.

ЧАСТИНА III. ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ТА АНАЛІЗ ЕФЕКТИВНОСТІ

У сучасних умовах програмне забезпечення стрімко ускладнюється: системи стають багатокомпонентними, масштабними та тісно пов'язаними між собою. У таких системах навіть незначна помилка в одному модулі може спричинити збій або некоректну роботу всієї програми. Саме тому тестування є критично важливим етапом життєвого циклу розробки програмного забезпечення, адже воно дозволяє своєчасно виявити і виправити помилки, забезпечуючи стабільність і надійність системи.

Для перевірки коректності реалізації алгоритму нечіткого пошуку було обрано мову програмування Python, яка забезпечує простоту синтаксису, широкі можливості для роботи з текстовими даними та підтримку сучасних інструментів автоматизованого тестування.

У Python було обрано бібліотеку `unittest` — стандартний фреймворк для модульного тестування, який поставляється разом з інтерпретатором. **Unitest** дозволяє створювати тести в ізолюваному середовищі, групувати їх у тестові набори, отримувати звіти про успішність проходження та зручно інтегрується в CI/CD процеси.

Окрім `unittest`, у процесі розробки додатково використовувалися:

- `pytest` — для зручнішого запуску та читабельних звітів;
- `coverage` — для перевірки покриття коду тестами.

3.1 Методика тестування алгоритму нечіткого пошуку та розробка фікстури та сценарії тестування

Реалізація алгоритму нечіткого пошуку потребує обов'язкового тестування як його основного ядра — алгоритму обчислення відстані редагування, так і допоміжного модуля — таблиці подібності символів. Для цього створено повноцінну систему юніт-тестів, яка охоплює всі критичні аспекти логіки.

1. Фреймворк і архітектура тестів

Для розробки тестів використано:

- unittest — стандартний фреймворк Python для структурованого модульного тестування;
- setUpClass() — фікстура, яка забезпечує одноразове завантаження таблиці подібності з файлу simtable.json;
- Кожен тест є ізольованим і перевіряє окрему логіку, щоб уникнути побічних ефектів.

Мета тестування — переконатися в тому, що реалізований алгоритм нечіткого пошуку коректно:

- обчислює відстань редагування між словами з урахуванням подібності символів;
- коректно обробляє регістр;
- правильно працює з різними джерелами подібності: семантичною, візуальною, помилками клавіатури;
- забезпечує симетричність таблиці подібності.

Для організації ефективного тестування таблиці подібності символів було реалізовано спеціальну тестову фікстуру на основі методу setUpClass із фреймворку unittest, що дозволяє ініціалізувати об'єкти один раз для всіх тестів у межах одного класу. Це забезпечує повторне використання об'єкта

таблиці подібності (SimTable) без необхідності його повторного завантаження з файлу перед кожним тестом.

```
@classmethod
def setUpClass(cls):
    cls.sim_table = SimTable("json/simtable.json")
```

Рисунок 3.1 код завантаження

Завдяки цьому рішення підвищується продуктивність та узгодженість результатів тестування.

Тестування методу `get_replace_cost()`

Для перевірки правильності функціонування методу `get_replace_cost()` було реалізовано серію юніт-тестів, кожен з яких перевіряє окрему властивість таблиці подібності:

```
self.assertEqual(self.sim_table.get_replace_cost('a', 'a'), 0.0)
```

Рисунок 3.2 Перевірка однакових символів

Перевіряє, що вартість заміни однакових символів дорівнює 0.0. Це є базовим інваріантом усіх алгоритмів відстані редагування.

```
cost = self.sim_table.get_replace_cost('A', 'a')
self.assertAlmostEqual(cost, 0.1, delta=0.01)
```

Рисунок 3.3 Перевірка різниці в регістрі

Перевіряє, що символи з різним регістром (наприклад, 'A' і 'a') мають вагу заміни, рівну значенню `CAPITAL_COST = 0.1`. Це дозволяє враховувати незначні типографічні відмінності.

```
cost = self.sim_table.get_replace_cost('t', 'r')
self.assertAlmostEqual(cost, 0.7, delta=0.01)
```

Рисунок 3.4 Помилка при наборі на клавіатурі

Тестує вагу заміни для символів, розташованих поруч на клавіатурі. Такі випадки трактуються як класичні клавіатурні помилки, які мають вартість, визначену параметром `KEYBOARD_MISTAKE_COST`.

```
cost = self.sim_table.get_replace_cost('b', 'd')
self.assertAlmostEqual(cost, 0.5, delta=0.01)
```

Рисунок 3.5 Візуально подібні символи

Перевіряє, що символи, які мають схожу візуальну форму (наприклад, 'b' і 'd'), розпізнаються системою як візуально подібні та мають вартість заміни 0.5 згідно з `VISUAL_SIMILAR_CHAR_COST`.

```
cost = self.sim_table.get_replace_cost('x', '💀')
self.assertIsNone(cost)
```

Рисунок 3.6 Символ не знайдено у таблиці

Перевіряє, що для символів, яких немає у таблиці подібності (наприклад, емодзі чи спецсимволів), функція повертає `None`, що свідчить про відсутність відповідності.

```
cost = self.sim_table.get_replace_cost('a', 'á')
self.assertAlmostEqual(cost, 0.8, delta=0.01)
```

Рисунок 3.7 Семантична подібність із JSON

Підтверджує, що семантична подібність символів, таких як 'a' і 'á', правильно зчитується з таблиці подібності (`simtable.json`) і функція повертає відповідну вагу.

```
for ch1 in self.sim_table.table:
    for ch2 in self.sim_table.table[ch1]:
        forward = self.sim_table.get_replace_cost(ch1, ch2)
        backward = self.sim_table.get_replace_cost(ch2, ch1)
        self.assertAlmostEqual(forward, backward, delta=0.01)
```

Рисунок 3.8 Перевірка симетричності таблиці

Перевіряє, що таблиця є симетричною, тобто що $\text{get_replace_cost}(a, b) == \text{get_replace_cost}(b, a)$ для всіх пар символів. Це важлива властивість для підвищення передбачуваності та узгодженості роботи алгоритму.

Вищезазначені тести дозволяють детально перевірити логіку обчислення вартості заміни символів на основі:

- абсолютної відповідності;
- чутливості до регістру;
- близькості на клавіатурі;
- візуальної подібності;
- семантичної відповідності;
- наявності даних у таблиці;
- симетричності записів.

Такий рівень деталізації дозволяє не лише гарантувати правильність роботи окремого компонента, а й підготувати платформу для тестування всієї системи нечіткого пошуку.

Сформована група тестів охоплює всі ключові властивості адаптованого алгоритму Damerau-Levenshtein:

- правильну реакцію на однакові, подібні та абсолютно різні символи;
- підтримку багатомовності (латиниця ↔ кирилиця);
- врахування специфічних категорій помилок (семантика, регістр, клавіатурні, візуальні);
- підтримку всіх чотирьох базових операцій: вставка, видалення, заміна, транспозиція.

Тести є вичерпними і підтверджують надійність реалізації як у загальних випадках, так і в найдрібніших винятках, що мають значення для практичних застосунків нечіткого пошуку.

3.2. Тестування таблиці подібності та розрахунок відстані редагування

Тестування функції обчислення відстані редагування Damerau-Levenshtein

Алгоритм Damerau-Levenshtein, адаптований до роботи з таблицею подібності символів, дозволяє враховувати не лише базові операції редагування (вставка, видалення, заміна та транспозиція), а й специфічні ваги заміни символів відповідно до їхньої семантичної, візуальної або орфографічної близькості. Для перевірки коректної роботи алгоритму були розроблені окремі юніт-тести, що охоплюють усі ключові сценарії:

```
self.assertEqual(damerau_levenshtein("test", "test", self.sim_table), 0.0)
```

Рисунок 3.9 Тест `test_distance_exact` — перевірка на ідентичність слів

Перевіряє, що при порівнянні ідентичних рядків функція повертає нульове значення.

Це базовий контрольний випадок, який гарантує, що якщо два слова збігаються повністю, то алгоритм не додає жодної вартості редагування, навіть за наявності таблиці подібності. Це означає, що функція не виконує зайвих обчислень та дотримується головної властивості метрики — нуль на діагоналі.

```
self.assertEqual(damerau_levenshtein("cat", "cart", self.sim_table), 1.0)
```

Рисунок 3.10 Тест `test_distance_insert` — вставка або видалення символу

Перевіряє, що вставка одного символу (у даному випадку 'r') коштує 1.0.

Алгоритм повинен враховувати додавання або вилучення символу як базову операцію з фіксованою вартістю. Якщо порівнюються слова, одне з яких має на один символ більше, і ці символи не є схожими — це має коштувати 1.0, згідно класичного підходу Damerau-Levenshtein.

```
self.assertAlmostEqual(damerau_levenshtein("Apple", "apple", self.sim_table), 0.1, delta=0.01)
```

Рисунок 3.11 Тест `test_distance_case` — різний регістр

Перевіряє, що різниця лише в регістрі оцінюється як 0.1, згідно з параметром `CAPITAL_COST`.

Цей тест підтверджує, що алгоритм підтримує розпізнавання регістру символів і застосовує окрему вагу, якщо символи мають однакову нижню

форму, але відрізняються великими/малими літерами. Це дозволяє зберігати чутливість до регістру без надмірної штрафної вартості.

```
self.assertAlmostEqual(damerau_levenshtein("test", "rest", self.sim_table), 0.7, delta=0.01)
```

Рисунок 3.12 Тест test_distance_keyboard — типова помилка клавіатури

Перевіряє, що заміна символу через сусідню клавішу на клавіатурі (наприклад, $t \rightarrow r$) має вартість 0.7.

Сценарій моделює поширену друкарську помилку. Якщо символи є сусідами в розкладці (QWERTY або ЙЦУКЕН), то заміна між ними не повинна коштувати повноцінну одиницю. Таблиця подібності враховує цю категорію як "keyboard mistake".

```
self.assertAlmostEqual(damerau_levenshtein("bod", "pod", self.sim_table), 0.5, delta=0.01)
```

Рисунок 3.13 Тест test_distance_visual — візуально подібні символи

Перевіряє, що візуально схожі символи (наприклад, b і p) замінюються з вартістю 0.5.

Такі символи мають подібні контури у друкованому вигляді і часто плутаються в OCR-системах або при неуважному введенні. Це є приклад "visual similarity", яка оцінюється окремою вагою у таблиці.

```
d = damerau_levenshtein("mama", "mama", self.sim_table)
self.assertless(d, 3.0)
```

Рисунок 3.14 Тест test_distance_semantic — семантична подібність

Мета: Перевірити, що алгоритм коректно оцінює заміни між латинськими та кириличними символами, які мають однакове значення ($m \leftrightarrow m$), і загальна вага не перевищує 3.0.

У цьому прикладі "тата" трансформується у "мама", і хоча символи виглядають схожими, з технічної точки зору це повністю різні коди.

Алгоритм повинен використати значення з таблиці подібності і знизити вартість для таких пар до 0.8–0.9, а не накладати повну вартість заміни.

```
self.assertEqual(damerau_levenshtein("abcd", "acbd", self.sim_table), 1.0)
```

Рисунок 3.15 Тест test_distance_transposition — транспозиція

Мета: Перевірити, що транспозиція двох сусідніх символів (у цьому випадку b і c) має вартість 1.0.

Алгоритм Damerau-Levenshtein, на відміну від класичного алгоритму Левенштейна, враховує перестановку двох символів як окрему, дешевшу операцію. Це дозволяє точніше порівнювати слова з перестановками і не карати їх так сильно, як за повноцінні видалення/вставки.

Таблиця 1.1 Тестування

№	Назва тесту	Тип перевірки	Результат
1	test_distance_exact	Однакові слова $\rightarrow$ 0.0	✓ Пройдено
2	test_distance_insert	Вставка / видалення $\rightarrow$ 1.0	✓ Пройдено
3	test_distance_case	Регістр $\rightarrow$ 0.1	✓ Пройдено

4	test_distance_keyboard	Клавіатурна помилка → 0.7	✓ Пройдено
5	test_distance_semantic	Семантична подібність → < 3.0	✓ Пройдено
6	test_distance_transposition	Транспозиція → 1.0	✓ Пройдено
7	test_distance_visual	Візуальна подібність → 0.5	✓ Пройдено
8	test_cost_case_sensitive	Різний регістр → 0.1	✓ Пройдено
9	test_cost_keyboard	Сусіди по клавіатурі → 0.7	✓ Пройдено
10	test_cost_none	Немає у таблиці → None	✓ Пройдено
11	test_cost_same	Тотожні символи → 0.0	✓ Пройдено
12	test_cost_semantic_from_json	'a' ↔ 'á' із JSON → 0.8	✓ Пройдено
13	test_cost_visual	'b' ↔ 'd' → 0.5	✓ Пройдено
14	test_symmetric_pairs	Симетричність таблиці	✓ Пройдено

3.3 Порівняльний аналіз швидкодії та точності

evaluate_accuracy.py

Пояснення до коду для тестування точності алгоритму Damerau–Levenshtein із таблицею подібності

У цьому скрипті реалізовано експериментальне тестування якості нечіткого пошуку шляхом обчислення відстані редагування між текстовими парами, використовуючи алгоритм Damerau–Levenshtein із врахуванням таблиці подібності символів. Тестові приклади зчитуються з підготовлених CSV-файлів.

Структура коду

```
import csv, os, sys, time
from multiprocessing import Pool, cpu_count
```

Рисунок 3.16 Імпорт бібліотек

Підключаються модулі для роботи з файлами, системними аргументами, багатопроцесорною обробкою та заміру часу.

```
def safe_print(...)
```

Рисунок 3.17 Вивід у консоль

Функція `safe_print` захищає від помилок кодування при друку Unicode-символів у терміналі Windows (наприклад, емодзі або кирилиця).

```
INPUT_FILE, OUTPUT_FILE, JSON_TABLE_PATH
```

Рисунок 3.18 Константи та параметризація

Шляхи до вхідних даних та таблиці подібності символів. Параметр LIMIT визначає кількість пар, що будуть використані для тестування. Його можна передати з командного рядка або встановити вручну.

```
def load_test_pairs(...)
```

Рисунок 3.19 Завантаження пар слів

CSV-файли містять приклади: train_input.csv (розпізнані тексти) та train_output.csv (очікувані тексти). Функція формує список пар рядків (input_text, output_text).

```
def compute_pair(pair)
```

Рисунок 3.20 Паралельне обчислення відстані

Кожна пара обробляється через функцію Damerau–Levenshtein, яка використовує таблицю подібності simtable.json. Для ефективності обчислення виконується в багатопроцесорному режимі за допомогою модуля multiprocessing.Pool.

```
def main()
```

Рисунок 3.21 Основна функція

Головна логіка:

- перевірка наявності файлів;
- завантаження пар;
- запуск паралельного обчислення;
- замір часу;

- обчислення середньої відстані між вхідним і очікуваним текстами.

Формат запуску

Цей скрипт можна запускати з консолі:

```
python test_evaluate_accuracy.py 100
```

Рисунок 3.22 Обмеження кількості тестових пар

Якщо параметр не вказано, за замовчуванням буде використано $LIMIT = 3$.

Вивід результатів

Програма виводить:

- кількість пар;
- приклади перших 3 пар;
- середню відстань;
- загальний час обробки.

Ці результати дозволяють порівняти якість і продуктивність різних реалізацій алгоритмів нечіткого пошуку.

І далі був сформований `full_test.py` загальний тестовий файл який запускає всі тести

1. Блок налаштування параметрів запуску

У першій частині коду визначаються:

```
DEFAULT_LIMIT = 10  
limit_value = DEFAULT_LIMIT
```

Рисунок 3.23 Базові параметри запуску тестів

Користувач може передати альтернативне значення з командного рядка:

```

if len(sys.argv) > 1:
    try:
        limit_value = int(sys.argv[1])
    except ValueError:
        ...

```

Рисунок 3.24 Кількість пар для тестування за замовчуванням

Також задаються іконки для покращення виводу результатів (опційно).

```

test_files = [
    "test_similarity.py",
    "test_rapidfuzz_accuracy.py",
    "test_evaluate_accuracy.py",
    "test_benchmark_analysis.py"
]

```

Рисунок 3.25 Формується список файлів для тестування

Файли з accuracy та benchmark приймають параметр limit, інші — ні.

2. Цей блок виконує по черзі кожен з тестових файлів через subprocess.run(...), передаючи limit як аргумент, якщо файл підтримує це:

```

args = ["python", filename]
if "accuracy" in filename or "benchmark" in filename:
    args.append(str(limit_value))

```

Рисунок 3.26 Блок запуску зовнішніх тестів

Кожен файл виконується, а результати зчитуються з stdout і зберігаються для подальшого аналізу:

```

if "benchmark" in filename:
    benchmark_output = stdout
elif "evaluate" in filename:
    evaluate_output = stdout
elif "rapidfuzz" in filename:
    rapidfuzz_output = stdout

```

Рисунок 3.27 Аналіз результатів

Таким чином забезпечується універсальний запуск усіх модулів, незалежно від їх внутрішньої реалізації.

3. Блок аналізу продуктивності та точності

Розбір результатів продуктивності здійснюється шляхом пошуку рядків із зазначенням часу виконання кожного алгоритму:

```
pattern_damerau = r"Damerau-Levenshtein:\s+([\d.]+)"
pattern_lev = r"Класичний Levenshtein:\s+([\d.]+)"
```

Рисунок 3.28 Аналіз продуктивності

Після цього обчислюється відношення продуктивності:

$$\text{ratio} = t1 / t2$$

І виводиться висновок, який алгоритм швидший.

Аналіз точності

```
match_eval = re.search(r"Середня відстань:\s*([\d.]+)", evaluate_output)
match_rapid = re.search(r"Середня відстань RapidFuzz:\s*([\d.]+)", rapidfuzz_output)
```

Рисунок 3.29 Шаблони для пошуку середньої відстані з обох джерел

```
delta = eval_dist - rapid_dist
```

Рисунок 3.30 Порівнюється ефективність алгоритму на основі отриманих середніх відстаней

Цей блок дозволяє зробити висновок про якість розпізнавання на основі відстані редагування між очікуваним і отриманим текстом.

Цей скрипт автоматизує процес повного порівняльного тестування, дозволяючи:

- запускати тести окремо та в зручному порядку;
- змінювати кількість прикладів для тестування без модифікації коду;
- оцінювати як швидкість, так і точність реалізованого алгоритму Damerau–Levenshtein у порівнянні з RapidFuzz.

У результаті виконання 14 юніт-тестів перевірялися базові властивості алгоритму Damerau-Levenshtein із таблицею подібності символів:

```

1  Виконання: test_similarity.py
2  ✓ test_distance_case – перістр = 0.1
3  ✓ test_distance_exact – однакові слова = 0.0
4  ✓ test_distance_insert – видалення = 1.0
5  ✓ test_distance_keyboard – test ↔ rest = 0.7
6  ✓ test_distance_semantic – мама ↔ мама = 2.3
7  ✓ test_distance_transposition – транспозиція = 1.0
8  ✓ test_distance_visual – bod ↔ pod = 0.5
9  ✓ test_cost_case_sensitive – різний регістр
10 ✓ test_cost_keyboard – t i r = 0.7
11 ✓ test_cost_none – символ не знайдено
12 ✓ test_cost_same – однакові символи
13 ✓ test_cost_semantic_from_json – семантична подібність 'a' і 'á' = 0.8
14 ✓ test_cost_visual – b i d = 0.5
15 ✓ test_symmetric_pairs – перевірка симетричності
16 ⚠ Помилка:
17 .....
18 -----
19 Ran 14 tests in 0.001s
20
21 OK

```

Рисунок 3.31 Тестування функціональності: test_similarity.py

Всі тестові сценарії були успішно пройдені, що свідчить про коректну реалізацію алгоритму з використанням таблиці подібності символів.

Порівняння продуктивності: test_benchmark_analysis.py

Було змодельовано обробку 100000 пар рядків для двох алгоритмів:

- реалізований Damerau-Levenshtein з урахуванням подібності символів;
- класичний Levenshtein з бібліотеки Levenshtein.

Аналіз роботи алгоритмів нечіткого пошуку

У рамках експериментального дослідження було проведено порівняння трьох режимів роботи системи нечіткого пошуку:

1. Алгоритм Damerau-Levenshtein без використання таблиці подібності символів.
2. Алгоритм Damerau-Levenshtein із таблицею подібності символів.
3. Алгоритм RapidFuzz (класичний пошук без врахування подібності символів).

Вхідним запитом для тестування було обрано слово "Helo".

Таблиця 3.1

Результати роботи без використання таблиці подібності символів (Час обробки: 2.09 сек)

№	Вхідний рядок	Знайдений рядок	Відстань	Співпадіння (%)
1	Helo	Hello	1	80
2	Helo	Hely	1	75
3	Helo	Hero	1	75
4	Helo	Hel	1	75
5	Helo	Held	1	75
6	Helo	belo	1	75
7	Helo	Velo	1	75

8	Helo	Helc	1	75
9	Helo	Hel™	1	75
10	Helo	Help	1	75

Таблиця 3.2

Результати роботи із використанням таблиці подібності символів (Час обробки: 3.61 сек)

№	Вхідний рядок	Знайдений рядок	Відстань	Співпадіння (%)
1	Helo	helo	0.1	97.5
2	Helo	Held	0.5	87.5
3	Helo	Help	0.5	87.5
4	Helo	held	0.6	85
5	Helo	help	0.6	85
6	Helo	hclO	0.7	82.5
7	Helo	helD	0.7	82.5
8	Helo	Hrlo	0.7	82.5
9	Helo	Hell	0.7	82.5
10	Helo	belo	0.8	80

Таблиця 3.3

Результати роботи алгоритму RapidFuzz (Час обробки: 0.35 сек)

№	Вхідний рядок	Знайдений рядок	Відстань	Співпадіння (%)
1	Helo	Hello	1	80
2	Helo	Hely	1	75
3	Helo	Hero	1	75

4	Helo	Hel	1	75
5	Helo	Held	1	75
6	Helo	belo	1	75
7	Helo	Velo	1	75
8	Helo	Helc	1	75
9	Helo	Hel™	1	75
10	Helo	Help	1	75

Алгоритм Damerau-Levenshtein без таблиці подібності демонструє базову точність пошуку, але не враховує типові помилки користувача. Використання таблиці подібності символів дозволяє підвищити точність результатів, однак збільшує час обробки. Алгоритм RapidFuzz забезпечує найшвидшу обробку, проте менш точний для випадків із близькими за звучанням символами.

Новий алгоритм показав меншу середню відстань, тобто краще співпадає з очікуваним результатом. Це пояснюється більш гнучкою оцінкою подібності, де враховується не лише точне співпадіння символів, а й їхня візуальна, семантична та клавіатурна схожість. Проте різниця в швидкості вона очевидна і можна пояснити дуже просто бо скрипт написано на Python а він повільніший ніж стандартна бібліотека на c++.

Висновки до Частини III

У третьому розділі було проведено всебічну експериментальну перевірку реалізованого алгоритму нечіткого пошуку, заснованого на модифікації алгоритму Damerau–Levenshtein із використанням таблиці подібності символів. Метою перевірки було оцінити не лише функціональну правильність роботи алгоритму, а й його точність, продуктивність та переваги у порівнянні з існуючими бібліотечними рішеннями.

На першому етапі створено набір модульних тестів із використанням стандартного фреймворку unittest. Усі 14 тестів були успішно пройдені, що підтвердило коректність реалізації основних компонентів — алгоритму обчислення відстані редагування та роботи з таблицею подібності символів.

Перевірено обробку різних типів помилок: вставок, видалень, регістрових відмінностей, помилок через близькість символів на клавіатурі та візуальну схожість. Також перевірено симетричність таблиці подібності, що забезпечує передбачувану поведінку алгоритму.

На другому етапі здійснено тестування точності пошуку на реальних даних, зокрема парах рядків, отриманих після оптичного розпізнавання тексту (OCR). Алгоритм із таблицею подібності показав суттєве підвищення точності у порівнянні зі стандартним Damerau–Levenshtein та бібліотекою RapidFuzz, оскільки краще враховував схожість символів навіть у випадках складних помилок.

На третьому етапі оцінювалась швидкодія алгоритму:

- тестовий алгоритм Damerau–Levenshtein із таблицею працював у **1.7 рази повільніше** за класичний варіант без таблиці подібності;
- і приблизно у **10 разів повільніше** за бібліотеку RapidFuzz.

Причиною такої різниці є додаткові обчислення для врахування ваг заміन символів, а також використання інтерпретованої мови Python замість оптимізованих компільованих бібліотек на C++.

Окремо було розроблено скрипт fulltest.py, який дозволяє автоматизовано запускати всі модулі тестування, передавати параметри обмеження кількості тестових пар та виводити підсумкові результати продуктивності й точності. Це значно полегшує експериментальне тестування і прискорює аналіз результатів.

Результати експериментальної перевірки дозволяють зробити такі висновки:

- реалізований алгоритм працює коректно відповідно до теоретичної моделі;
- забезпечує підвищену точність у випадках нечітких або зашумлених даних;
- стабільно обробляє широкий спектр типових помилок;
- придатний для використання у системах, де критично важлива точність пошуку (пошукові системи, перевірка правильності імен, післяобробка OCR);
- хоча й поступається за швидкістю компільованим бібліотечним рішенням, забезпечує істотне підвищення якості пошуку.

Отримані результати підтверджують практичну придатність розробленого алгоритму для реального застосування в задачах, де точність має вищий пріоритет, ніж швидкодія.

Отримані результати підтверджують практичну придатність розробленого алгоритму для реального застосування в задачах, де точність має вищий пріоритет, ніж швидкодія.

Поставлена у дипломній роботі мета була повністю досягнута. Отримані результати можуть бути застосовані в реальних інформаційно-пошукових системах, системах перевірки правопису, а також у проектах із високими вимогами до точності аналізу текстових даних.

ВИСНОВОК

У результаті виконання дипломної роботи було проведено комплексне дослідження методів нечіткого пошуку текстової інформації, розроблено власний адаптивний алгоритм із використанням таблиці подібності символів

і реалізовано його програмну імплементацію з подальшим тестуванням точності та продуктивності.

У першому розділі здійснено теоретичний аналіз сучасних алгоритмів нечіткого пошуку, зокрема алгоритмів Дамерау–Левенштейна, Джаро–Вінклера, N-грам, Bitap і SoundEx. Було встановлено, що кожен з методів має свої переваги й обмеження залежно від контексту застосування. Особлива увага приділялася розгляду впливу варіативності написання, типових помилок введення, регістру символів та фонетичних відмінностей на ефективність пошуку в текстових масивах.

У другому розділі реалізовано вдосконалений алгоритм нечіткого пошуку на основі метрики Дамерау–Левенштейна, доповнений підтримкою таблиці подібності символів. Розроблена трирівнева структура таблиці враховує:

- семантичну близькість символів різних мов;
- типові помилки введення (зумовлені фізичною близькістю клавіш на клавіатурі);
- візуальну схожість символів.

Таблиця була реалізована у форматі JSON, що забезпечує зручність масштабування та інтеграції. Завдяки цьому алгоритм отримав можливість гнучко оцінювати подібність між символами, що підвищило точність пошуку в реальних умовах, зокрема при обробці багатомовних текстів і даних із помилками. Реалізація здійснена мовою програмування Python із повною підтримкою Unicode.

У третьому розділі проведено повноцінне тестування розробленого рішення, зокрема:

- модульне тестування функцій обчислення відстані редагування та роботи таблиці подібності;

- оцінка точності на парах текстів, отриманих шляхом оптичного розпізнавання (OCR);
- порівняння результатів із бібліотекою RapidFuzz;
- аналіз швидкодії алгоритму.

Усі тестові сценарії були успішно пройдені. Результати експериментів показали, що розроблений алгоритм забезпечує вищу точність порівняння рядків у складних випадках, зокрема при наявності помилок введення або візуальних спотворень. При цьому виявлено, що за швидкодією алгоритм поступається класичним реалізаціям, працюючи приблизно у 1.7 рази повільніше за стандартний Damerau–Levenshtein без таблиці та у 10 разів повільніше за RapidFuzz. Такий компроміс між точністю та швидкістю є прийнятним для задач, де якість пошуку має вирішальне значення.

Поставлена у дипломній роботі мета була повністю досягнута. Розроблене рішення є практично придатним для застосування в інформаційно-пошукових системах, системах перевірки правопису, післяобробці OCR-даних і проектах, що вимагають підвищеної точності аналізу текстової інформації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Серед джерел обов'язково мають бути наукові статті, фундаментальні друковані праці (підручники, монографії). Можливе згадування дисертаційних робіт. На всі джерела в переліку мають бути посилання по тексту. Кількість джерел 30-40

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

<https://medium.com/data-science-in-your-pocket/fuzzy-matching-explained-e0ff30cc00ca> [algorithms-](#)

ДОДАТОК А

```
1. import json
2.
3. # Вартісна модель
4. COST_DICT = {
5.     "SIMILAR_CHAR_COST": 0.3,
6.     "KEYBOARD_MISTAKE_COST": 0.7,
7.     "VISUAL_SIMILAR_CHAR_COST": 0.5
8. }
9.
10. data = {"data": {}}
11.
12. # === Категорія 1: Семантично подібні символи (UA + EN) ===
13. semantic_groups = {
14.     "a": ["a", "à", "á"],
15.     "b": ["b"],
16.     "c": ["c"],
17.     "e": ["e", "é", "e"],
18.     "i": ["i", "ï", "í"],
19.     "o": ["o", "ö"],
20.     "p": ["p"],
21.     "x": ["x"],
22.     "y": ["y"],
23.     "g": ["r"],
24.     "u": ["ü"],
25.     "z": ["z"]
26. }
27.
28. for letter, group in semantic_groups.items():
29.     similar_group = set(group)
30.     similar_group.add(letter)
31.     for lt1 in similar_group:
32.         if lt1 not in data["data"]:
33.             data["data"][lt1] = {}
34.         for lt2 in similar_group:
35.             if lt1 != lt2:
36.                 data["data"][lt1][lt2] = COST_DICT["SIMILAR_CHAR_COST"]
37.
```

```

38. # === Категорія 2: Помилки на клавіатурі (QWERTY+ЙЦУКЕН) ===
39. # ЙЦУКЕН (українська розкладка)
40. keyboard_ua = [
41.     list("йцукенгшщзхї"),
42.     list("фівапролджє"),
43.     list("ячсмитьбю")
44. ]
45.
46. # QWERTY (англійська)
47. keyboard_en = [
48.     list("qwertyuiop"),
49.     list("asdfghjkl"),
50.     list("zxcvbnm")
51. ]
52.
53. def add_to_chars(char, *ch_set):
54.     if char not in data["data"]:
55.         data["data"][char] = {}
56.     for ch in ch_set:
57.         data["data"][char][ch] = COST_DICT["KEYBOARD_MISTAKE_COST"]
58.
59. def process_keyboard_layout(layout):
60.     for row_idx, row in enumerate(layout):
61.         for col_idx, key in enumerate(row):
62.             neighbors = []
63.             for dr in [-1, 0, 1]:
64.                 for dc in [-1, 0, 1]:
65.                     nr, nc = row_idx + dr, col_idx + dc
66.                     if 0 <= nr < len(layout) and 0 <= nc <
len(layout[nr]):
67.                         neighbor = layout[nr][nc]
68.                         if neighbor != key:
69.                             neighbors.append(neighbor)
70.             add_to_chars(key, *neighbors)
71.
72. process_keyboard_layout(keyboard_en)
73. process_keyboard_layout(keyboard_ua)
74.
75. # === Категорія 3: Візуально подібні символи (UA+EN) ===

```

```

76. visual_groups = [
77.     ("b", "d", "p", "q", "o"),
78.     ("i", "ı", "ı", "j", "l", "t"),
79.     ("u", "v", "w", "y"),
80.     ("m", "n", "и"),
81.     ("c", "e", "o", "s", "e"),
82.     ("h", "к", "k"),
83.     ("f", "т", "t"),
84.     ("g", "r", "q"),
85.     ("x", "y", "x"),
86.     ("a", "o", "a"),
87.     ("p", "p"),
88.     ("n", "и", "v", "r")
89. ]
90.
91. def add_similar_ch(chs):
92.     for ch1 in chs:
93.         if ch1 not in data["data"]:
94.             data["data"][ch1] = {}
95.         for ch2 in chs:
96.             if ch1 == ch2:
97.                 continue
98.             current = data["data"][ch1].get(ch2, float("inf"))
99.             data["data"][ch1][ch2] = min(current,
100. COST_DICT["VISUAL_SIMILAR_CHAR_COST"])
101.
101. for group in visual_groups:
102.     add_similar_ch(group)
103.
104. # === Збереження у файл ===
105. with open("full_similarity_table_ua_en.json", "w", encoding="utf-8") as f:
106.     json.dump(data, f, ensure_ascii=False, indent=2)

```

ДОДАТОК Б

```
1. import json
2. from typing import Optional
3.
4. class SimTable:
5.     def __init__(self, path: str):
6.         self.table = {} # {СИМВОЛ: {ПОДІБНИЙ_СИМВОЛ: вага}}
7.         self.cost = {} # {назва_типу: вага}
8.         self.load_table(path)
9.
10.    def load_table(self, path: str):
11.        """Завантаження JSON-файлу з таблицею подібності"""
12.        try:
13.            with open(path, 'r', encoding='utf-8') as f:
14.                data = json.load(f)
15.
16.                self.cost = data.get("cost", {})
17.                self.table = data.get("data", {})
18.
19.        except FileNotFoundError:
20.            raise Exception(f"Файл не знайдено: {path}")
21.        except json.JSONDecodeError:
22.            raise Exception("Файл не є валідним JSON")
23.
24.    def get_replace_cost(self, c1: str, c2: str) -> Optional[float]:
25.        """Обчислення вартості заміни символів з урахуванням подібності та
    регістру"""
26.        if c1 == c2:
27.            return 0.0
28.
29.        is_upper_c1 = c1.isupper()
30.        is_upper_c2 = c2.isupper()
31.
32.        lc1 = c1.lower()
33.        lc2 = c2.lower()
34.
35.        # Символи ідентичні за значенням, але різні за регістром
36.        if lc1 == lc2:
```

```

37.         return self.cost.get("CAPITAL_COST", 0.0)
38.
39.         result = None
40.         if lc1 in self.table and lc2 in self.table[lc1]:
41.             result = float(self.table[lc1][lc2])
42.
43.             # Якщо тільки один символ у верхньому регістрі
44.             if result is not None and (is_upper_c1 ^ is_upper_c2):
45.                 result += self.cost.get("CAPITAL_COST", 0.0)
46.
47.         return result

1. def damerau_levenshtein(s1: str, s2: str, sim_table: Optional[SimTable] =
    None) -> float:
2.     """Повертає вагу редагування між s1 і s2 з урахуванням таблиці
    подібності"""
3.     len1 = len(s1)
4.     len2 = len(s2)
5.
6.     dp = [[0.0] * (len2 + 1) for _ in range(len1 + 1)]
7.
8.     for i in range(len1 + 1):
9.         dp[i][0] = float(i)
10.    for j in range(len2 + 1):
11.        dp[0][j] = float(j)
12.
13.    for i in range(1, len1 + 1):
14.        for j in range(1, len2 + 1):
15.            cost = 1.0
16.            if sim_table:
17.                replacement_cost = sim_table.get_replace_cost(s1[i - 1],
                    s2[j - 1])
18.                if replacement_cost is not None:
19.                    cost = replacement_cost
20.                elif s1[i - 1] == s2[j - 1]:
21.                    cost = 0.0
22.            else:
23.                if s1[i - 1] == s2[j - 1]:
24.                    cost = 0.0

```

```

25.
26.         dp[i][j] = min(
27.             dp[i - 1][j] + 1.0,      # видалення
28.             dp[i][j - 1] + 1.0,      # вставка
29.             dp[i - 1][j - 1] + cost  # заміна
30.         )
31.
32.         if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2] ==
           s2[j - 1]:
33.             dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + cost)  #
               транспозиція
34.
35.     return dp[len1][len2]

```