МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами

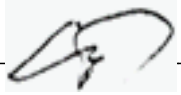
Кафедра програмного забезпечення автоматизованих систем

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма «Інженерія програмного забезпечення»

«Допущений до захисту»

Завідувач кафедри

_  _ проф. Приходько С.Б.

«___» _____ 2025 р.

КВАЛІФІКАЦІЙНАРОБОТА

на здобуття ступеня вищої освіти «бакалавр»

на тему: ***Розробка програмного забезпечення повнотекстового пошуку з багатопоточними алгоритмами***

Виконав: студент групи 4151

Mushko Мишко І.Л.

(підпис, ПІБ)

Керівник роботи:

доцент, кандидат фізико-математичних наук, доцент

(посада, науковий ступень вчене звання)

Латанська Л.О.

(підпис, ПІБ)

Миколаїв – 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами
Кафедра програмного забезпечення автоматизованих систем
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
Гарант освітньої програми
_____ доц. Макарова Л.М.
(підпис)
«_08_» ___04___ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «бакалавр»

Студенту Мишку Івану Леонідовичу
(Прізвище, ім'я, по батькові)

1. Тема роботи: Розробка програмного забезпечення повнотекстового пошуку з багатопоточними алгоритмами

Керівник роботи Латанська Людмила Олексіївна

Затверджені наказом ректора №153 від 08.04.2025 р.

2. Термін подання роботи: 02.06.2025 р.

3. Вихідні дані по роботі:

4. Перелік питань, що належать до розробки (найменування розділів)
Титульний аркуш; Завдання на кваліфікаційну роботу; Анотація (українською та англійською мовами); Зміст; Перелік умовних позначень, символів, одиниць та термінів (при необхідності); Вступ; Аналіз практичної задачі інженерії програмного забезпечення; Проєкт програмного забезпечення; Результати розробки програмного забезпечення; Розділ з охорони праці; Висновки; Список використаних джерел; Додатки (технічне завдання, текст програми, опис програми, інструкція користувача, програма та методика випробувань програмного забезпечення).

5. Перелік презентаційних матеріалів 1 Титульний слайд, 2 Мета та завдання кваліфікаційної роботи, 3 Актуальність та обґрунтування розробки, 4 Аналіз предметної області, 5 Аналіз вимог (діаграма варіантів використання), 6 Архітектура ПЗ: Структура веб-застосунку (діаграма компонентів), 7 Архітектура ПЗ: Фізичне розміщення компонентів (діаграма розгортання), 8 Ескізний проєкт: Деталізація основного сценарію (діаграма діяльності), 9 Ескізний проєкт: Проєктування інтерфейсу (прототипи), 10 Технічний проєкт: Статична структура веб-застосунку (діаграма класів), 11 Технічний проєкт: Структура бази даних (діаграма «сутність-зв'язок»), 12 Робочий проєкт: Вибір засобів розробки, 13 Результати розробки: Джерело «Текстові файли», 14 Результати розробки: Джерело «База даних PostgreSQL», 15 Результати розробки: Джерело «Веб-пошук», 16 Результати розробки: Аналіз результатів продуктивності, 17 Аналіз факторів та заходи з охорони праці, 18 Підготовлений комплект супровідної документації, 19 Висновки, 20 Заключний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	Гурець Н.В., ст. викладач		

7. Дата видачі завдання 08.04.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

Номер	Назва етапів роботи	Терміни виконання	Примітка
1.	Підготовка розділу ВСТУП	31.03.2025	ПП*
2.	Аналіз практичної задачі інженерії ПЗ	07.04.2025	ПП*
3.	Аналіз вимог до ПЗ	14.04.2025	ПП*
4.	Розробка архітектури ПЗ	21.04.2025	
5.	Розробка проекту ПЗ	05.05.2025	
6.	Реалізація та тестування ПЗ	12.05.2025	
7.	Підготовка розділу Результати розробки ПЗ	16.05.2025	
8.	Підготовка розділу з охорони праці	19.05.2025	ПП*
9.	Підготовка розділу ВИСНОВКИ	23.05.2025	
10.	Оформлення списку використаних джерел та додатків	26.05.2025	
11.	Подання на кафедру ПЗАС тексту остаточного варіанту роботи, підписаного її керівником, у роздрукованому та електронному форматі разом із заявами щодо самостійності виконання роботи та ідентичності друкованої та електронної версій роботи (Додатки 1 і 2 «Порядку здійснення заходів з перевірки робіт на наявність текстових збігів/ідентичності/схожості із використанням програмно-технічних засобів», який введений в дію наказом ректора НУК за №20 від 20.01.2020 р.)	02.06.2025	не пізніше, ніж за 14 діб до захисту (згідно п.4.1 зазначеного Порядку
12.	Підготовка презентації та доповіді	06.06.2025	
13.	Попередній захист роботи на засіданні кафедри ПЗАС	09.06.2025	
14.	Подання на кафедру ПЗАС електронних версій наступних документів у форматі pdf: кваліфікаційної роботи; файлу-опису кваліфікаційної роботи (згідно Додатку до наказу ректора НУК за №287-уч від 19.05.2020 р.); презентації доповіді, а також Авторського договору з додатком	16.06.2025	

* - за результатами переддипломної практики (ПП), яка була з 03.02.2025 до 23.03.2025 р.

Студент Myshko (підпис) Мишко І. Л. (ПІБ)

Керівник роботи (підпис) Латанська Л.О. (ПІБ)

АНОТАЦІЯ

Кваліфікаційна робота присвячена дослідженню, проєктуванню та розробці програмного забезпечення «SearchMaster» для повнотекстового пошуку з підтримкою багатопоточних алгоритмів та функціональністю порівняльного аналізу їх ефективності.

У роботі проведено аналіз існуючих алгоритмів повнотекстового пошуку (зокрема, Кнута-Морріса-Пратта, Бойєра-Мура, Рабіна-Карпа та інших) та аналогічних програмних рішень, на основі якого сформульовано вимоги та розроблено технічне завдання. Спроєктовано та реалізовано програмну систему «SearchMaster» з використанням мови програмування Python та вебфреймворку FastAPI. Система має модульну архітектуру, що включає реалізацію 10 різних алгоритмів пошуку, механізми паралельної обробки даних для файлових систем та баз даних PostgreSQL, а також інтерактивний веб-інтерфейс.

Проведено комплексне тестування розробленого ПЗ, включаючи функціональне тестування та тестування продуктивності на різних наборах даних (локальні файли, база даних, веб-пошук). Результати порівняльного аналізу показали, що варіації алгоритму Бойєра-Мура (Horspool, Sunday) є найефективнішими для локальних джерел даних, а паралельна обробка дозволяє досягти значного прискорення (до 4-5 разів). Також у роботі розглянуто аспекти охорони праці при розробці програмного забезпечення.

Кваліфікаційна робота викладена на 159 сторінках друкованого тексту, містить 21 рисунок, 22 таблиці, список використаних джерел з 14 найменувань та 5 додатків.

ABSTRACT

This qualification thesis is dedicated to the research, design, and development of the «SearchMaster» software for full-text search, featuring support for multi-threaded algorithms and the functionality for a comparative analysis of their effectiveness.

The thesis analyzes existing full-text search algorithms (including Knuth-Morris-Pratt, Boyer-Moore, Rabin-Karp, and others) and analogous software solutions, which formed the basis for formulating requirements and developing a technical specification. A software system, «SearchMaster», was designed and implemented using the Python programming language and the FastAPI web framework. The system features a modular architecture, which includes the implementation of 10 different search algorithms, mechanisms for parallel data processing for file systems and PostgreSQL databases, as well as an interactive web interface.

Comprehensive testing of the developed software was conducted, including functional and performance testing across various datasets (local files, database, web search). The results of the comparative analysis demonstrated that variations of the Boyer-Moore algorithm (Horspool, Sunday) are the most effective for local data sources, while parallel processing provides a significant speedup (up to 4-5 times). The thesis also addresses occupational safety aspects in software development.

The qualification thesis is presented on 159 pages of printed text, contains 21 figures, 22 tables, a list of references with 14 titles, and 5 appendices.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві обсяги даних зростають експоненційно, що ставить нові виклики перед системами пошуку та обробки інформації. Ефективний повнотекстовий пошук є критично важливим компонентом багатьох програмних систем. Існує велика кількість алгоритмів пошуку рядків (наприклад, Бойєра-Мура, Кнута-Морріса-Пратта, Рабіна-Карпа та інші), кожен з яких має свої переваги та недоліки залежно від характеристик даних та специфіки задачі. Вибір оптимального алгоритму є нетривіальною задачею інженерії програмного забезпечення.

Аналіз існуючих рішень, таких як потужні пошукові системи (Google, Bing) та спеціалізовані бібліотеки (Apache Lucene, Elasticsearch), показав, що вони, хоч і ефективні для кінцевого користувача чи індексації великих обсягів даних, мають недоліки з точки зору дослідницької задачі порівняння базових алгоритмів. Часто вони не дозволяють легко порівнювати різні алгоритми на необроблених даних (файли, БД, веб-контент) та керувати паралельною обробкою в однакових умовах, що ускладнює аналіз ефективності саме багатопоточних реалізацій. Це обґрунтовує необхідність розробки власного програмного забезпечення – інструменту, що дозволить проводити такі порівняльні дослідження.

Актуальність теми кваліфікаційної роботи полягає у необхідності створення гнучкого та ефективного інструменту для порівняльного аналізу алгоритмів повнотекстового пошуку, що дозволить розробникам обирати оптимальні рішення для своїх задач, а дослідникам – вивчати поведінку алгоритмів у контрольованому середовищі.

Метою кваліфікаційної роботи є розв’язання практичної задачі інженерії програмного забезпечення, а саме розробка програмного забезпечення повнотекстового пошуку з багатопоточними алгоритмами «SearchMaster» та з можливістю їх порівняння.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- Проаналізувати предметну область, зокрема існуючі алгоритми повнотекстового пошуку, їхні теоретичні основи та практичні особливості.
- Проаналізувати існуючі аналоги програмних систем та бібліотек для повнотекстового пошуку, щоб обґрунтувати необхідність розробки власного інструменту.
- Провести аналіз вимог до програмного забезпечення, що включає:
  - Розробку моделі варіантів використання для ідентифікації акторів та основних функцій системи.
  - Розробку детальних специфікацій для кожного варіанту використання.
- Розробити архітектуру програмного забезпечення, визначивши ключові компоненти, їхні взаємозв'язки та фізичне розміщення на вузлах інфраструктури.
- Розробити ескізний проєкт, що включає діаграми діяльності для візуалізації бізнес-логіки та прототипи користувацького інтерфейсу.
- Розробити технічний проєкт, що включає детальне проєктування статичної структури системи за допомогою діаграми класів та логічної моделі бази даних за допомогою ER-діаграми.
- Розробити робочий проєкт, що включає:
  - Обґрунтування вибору технологічного стеку.
  - Реалізацію програмного коду всіх модулів системи «SearchMaster».
- Провести демонстрацію роботи та аналіз продуктивності розробленого програмного забезпечення на різних наборах даних та з різними параметрами.
- Розробити повний комплект програмної документації, включаючи технічне завдання, опис програми, програму та методику випробувань, а також керівництво користувача.
- Проаналізувати потенційні шкідливі та небезпечні фактори при роботі з ПК та визначити заходи щодо забезпечення безпечних умов праці.

Об’єктом роботи є програмне забезпечення «SearchMaster», призначене для виконання повнотекстового пошуку з використанням різних алгоритмів, їх порівняння та аналізу продуктивності в умовах багатопоточної обробки даних.

Практична значущість роботи полягає у створенні готового до використання інструменту, який може застосовуватися в освітніх цілях для вивчення алгоритмів, у дослідницьких цілях для їх порівняння, а також як допоміжний засіб для розробників при виборі алгоритму для реальних проєктів.

Апробація результатів досліджень. Результати дослідження були представлені на науковій конференції «Когнітивні дослідження: результати, виклики та перспективи» (24 травня 2024 року, м. Київ) у доповіді на тему «Удосконалення та аналіз алгоритмів повнотекстового пошуку». Також, наукова робота на тему «Вдосконалення пошукового механізму у сховищі текстових документів за допомогою удосконаленого алгоритму Кнута-Моріса-Пратта» здобула перемогу у І турі Всеукраїнського конкурсу студентських наукових робіт з галузей знань і спеціальностей у 2023/2024 навчальному році, що проводився у Київському національному університеті імені Тараса Шевченка.

Публікації. Основні положення та результати кваліфікаційної роботи опубліковано у одній науковій праці – статті у журналі «Control Systems and Computers» [1].

1 АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПЗ ТА ПОСТАНОВКА ЗАДАЧІ НА РОЗРОБКУ ПЗ ПОВНОТЕКСТОВОГО ПОШУКУ З БАГАТОПОТОЧНИМИ АЛГОРИТМАМИ

1.1 Аналіз алгоритмів повнотекстового пошуку

Повнотекстовий пошук є фундаментальною операцією в багатьох галузях інформаційних технологій. Від ефективності пошукових алгоритмів залежить швидкість доступу до інформації, релевантність результатів та загальна продуктивність систем, що їх використовують. Існує значна кількість алгоритмів пошуку рядків, розроблених протягом десятиліть, починаючи від найпростішого «наївного» (Brute Force) [2] до більш складних та оптимізованих, таких як Кнута-Морріса-Пратта (KMP) [3, 4, 5], Бойєра-Мура (BM) [6], Рабіна-Карпа (RK) [6], Бітап (Bitap) [7, 8], Хорспула (Horspool) [6], Сандея (Sunday) та інших [6, 8].

Кожен алгоритм має свої теоретичні оцінки складності та практичні особливості поведінки. Наприклад:

- Наївний алгоритм: Простий у реалізації, але потенційно повільний (складність $O(n \cdot m)$, де n є довжиною тексту, а m відповідає за довжину шаблону).
- KMP: Має лінійну складність $O(n+m)$, ефективний для текстів з повторюваними патернами, використовує префікс-функцію.
- Boyer-Moore: Часто найшвидший на практиці для великих алфавітів та довгих шаблонів, використовує евристики «поганого символу» та «хорошого суфікса». Складність у гіршому випадку $O(n \cdot m)$, але на практиці близька до $O(n/m)$.
- Rabin-Karp: Використовує хешування, ефективний для пошуку множини шаблонів, середня складність $O(n+m)$, але залежить від якості хеш-функції та можливих колізій.
- Bitap: Ефективний для коротких шаблонів та невеликих алфавітів, використовує бітові операції, добре підходить для пошуку з помилками (не реалізовано в даній роботі, але є потенціалом).

- Horspool, Sunday, Two-Way, Simon: Різні варіації та оптимізації, що намагаються покращити швидкість зсуву шаблону по тексту.

Практична ефективність алгоритму залежить від багатьох факторів:

- Характеристики даних: Розмір тексту, розмір шаблону, розмір та природа алфавіту (текст, двійкові дані), наявність повторюваних послідовностей.
- Джерело даних: Пошук у файлах на диску, у базі даних чи у веб-контенті має різну латентність доступу до даних.
- Апаратне забезпечення: Можливості паралельної обробки (кількість ядер процесора).

Таким чином, вибір «найкращого» алгоритму не є однозначним. Розробникам часто потрібно експериментально визначати, який алгоритм буде найефективнішим для їх конкретної задачі та середовища. Це створює практичну потребу в інструменті, який дозволяє легко проводити такі порівняльні дослідження. Задача ускладнюється необхідністю врахування паралелізму, оскільки сучасні системи часто обробляють великі обсяги даних, і розпаралелювання пошуку може суттєво прискорити процес [9].

1.2 Аналіз існуючих аналогів повнотекстового пошуку та обґрунтування необхідності розробки

Існує певна кількість програмних продуктів та бібліотек, що реалізують алгоритми повнотекстового пошуку. Розглянемо деякі категорії:

- Пошукові системи загального призначення (Google, Bing, DuckDuckGo): Ці системи використовують надзвичайно складні, оптимізовані та часто пропрієтарні алгоритми, що включають не тільки пошук рядків, але й ранжування, індексацію, обробку природної мови тощо. Вони не призначені для порівняння базових алгоритмів пошуку рядків у контрольованому середовищі [10].
- Бібліотеки повнотекстового пошуку (Apache Lucene, Elasticsearch, Solr, Whoosh): Ці потужні інструменти надають готові рішення для індексації та пошуку

у великих колекціях документів. Вони використовують складні індекси (наприклад, інвертовані індекси) та оптимізовані алгоритми, але зазвичай не надають простого інтерфейсу для прямого порівняння базових алгоритмів пошуку рядків на «сирих» даних чи з різними джерелами «на льоту» [11].

Наприклад, Apache Solr надає адміністративний інтерфейс для управління індексами та виконання запитів, але не для прямого порівняння базових алгоритмів пошуку рядків (рис 1.1).

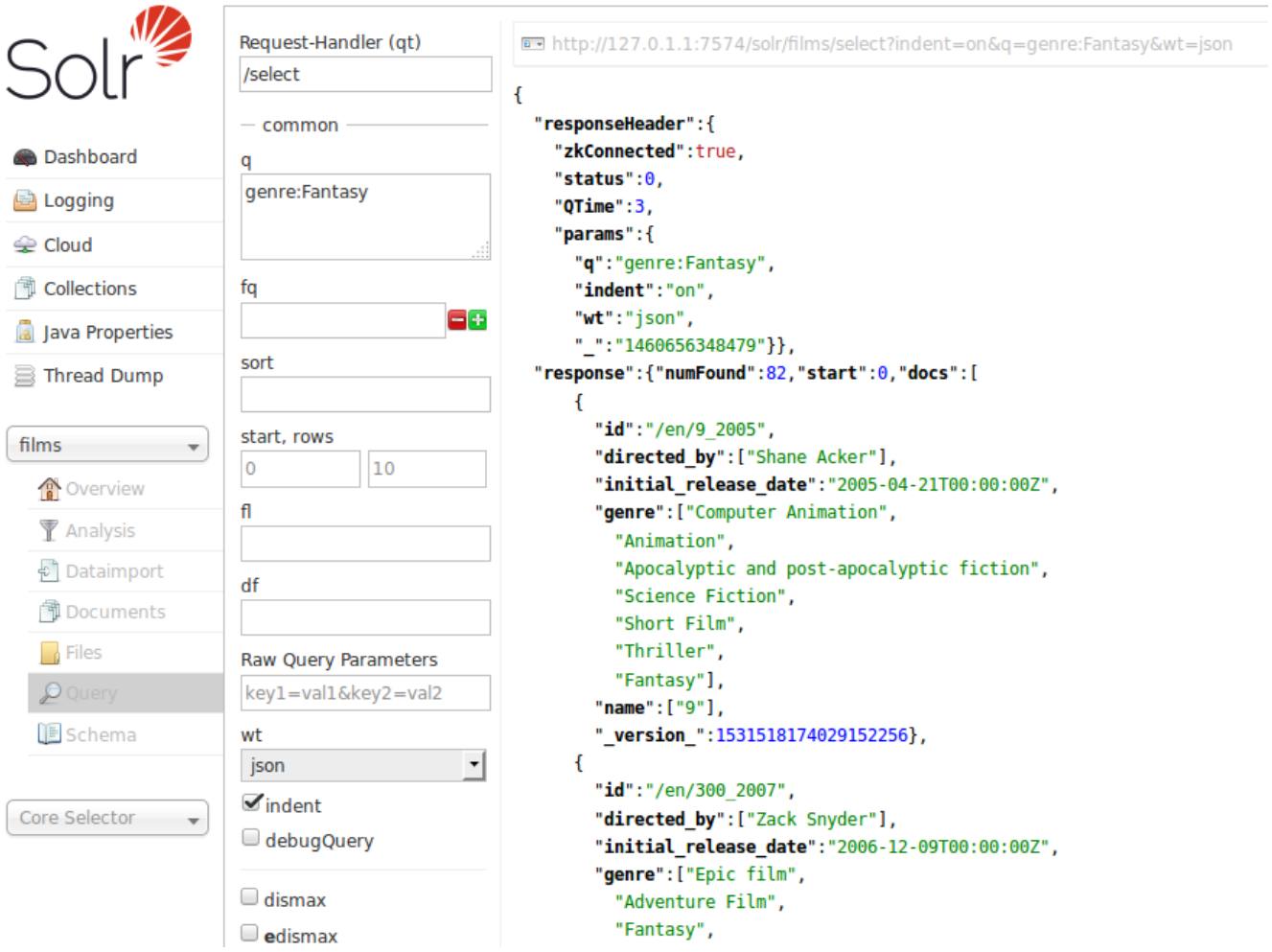


Рисунок 1.1 – Приклад інтерфейсу Apache Solr Admin UI [11]

Elasticsearch часто використовується з Kibana для візуалізації та аналізу даних, включаючи результати пошуку, але знову ж таки, фокус не на порівнянні самих алгоритмів пошуку (рис. 1.2).

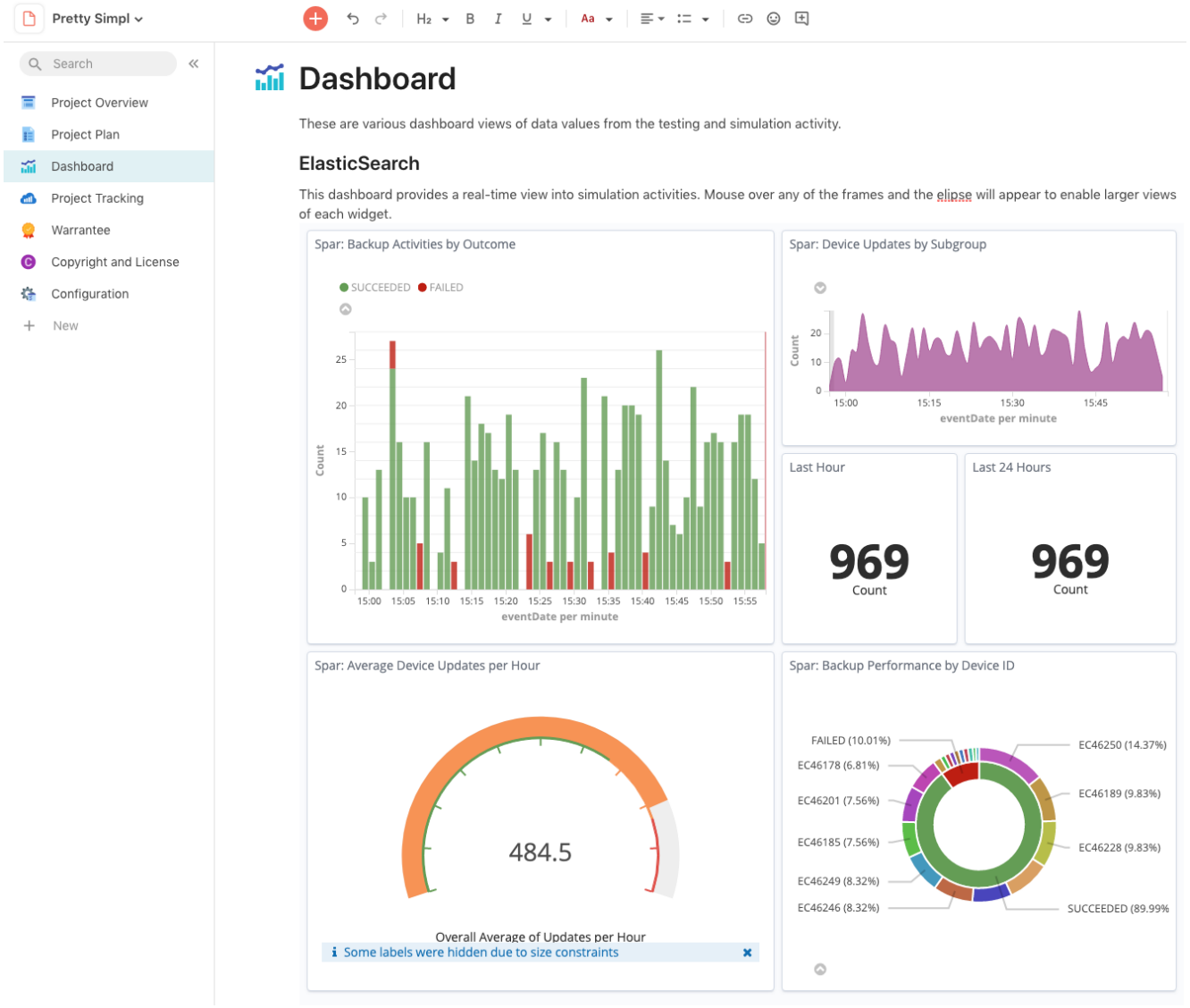


Рисунок 1.2 – Приклад панелі інструментів Kibana для Elasticsearch [12]

– Стандартні бібліотеки мов програмування: Багато мов програмування (включаючи Python) мають вбудовані функції або стандартні модулі для пошуку рядків (наприклад, str.find(), re в Python). Вони зазвичай реалізують один або кілька оптимізованих алгоритмів, але не надають можливості вибору чи порівняння різних підходів.

– Академічні та дослідницькі інструменти: Існують різні реалізації та тестові стенди, створені в рамках наукових досліджень [2-8]. Однак вони часто є специфічними для конкретного дослідження, можуть бути складними у використанні або не мати зручного інтерфейсу та підтримки різних джерел даних чи паралелізму.

Обґрунтування необхідності розробки власного ПЗ:

Проаналізувавши існуючі рішення, можна зробити висновок, що бракує універсального, гнучкого та легкого у використанні інструменту, який би задовольняв наступним вимогам:

- Реалізація широкого спектру (10 у даному випадку) класичних та сучасних алгоритмів пошуку рядків.
- Можливість пошуку у різних типах джерел даних (файли, БД, веб).
- Підтримка багатопоточної обробки з можливістю гнучкого налаштування кількості потоків/процесів для дослідження впливу паралелізму на продуктивність.
- Надання зручного веб-інтерфейсу для введення запитів, вибору параметрів та візуалізації результатів.
- Наявність функції прямого порівняння продуктивності алгоритмів за часом виконання та кількістю знайдених результатів для конкретного запиту та джерела.

Розробка ПЗ «SearchMaster» спрямована на заповнення цієї ніші, надаючи платформу для освітніх, дослідницьких та практичних цілей.

1.3 Постановка задачі на розробку ПЗ «SearchMaster»

Виходячи з аналізу алгоритмів повнотекстового пошуку та існуючих аналогів, формулюється наступна постановка задачі:

Потрібно розробити програмне забезпечення «SearchMaster» – веб-додаток, призначений для виконання, вимірювання та порівняння продуктивності різних алгоритмів повнотекстового пошуку рядків на різних джерелах даних з використанням паралельної обробки.

Вимоги до ПЗ «SearchMaster»:

Вимоги до складу виконуваних функцій:

- Пошук рядка: Система повинна дозволяти користувачеві вводити рядок (шаблон) для пошуку.

- Вибір джерела даних: Користувач повинен мати можливість обрати джерело даних для пошуку з наступних варіантів:
  - Локальні текстові файли (колекція файлів, що зберігаються на сервері).
  - База даних PostgreSQL (таблиця з текстовими документами).
  - Веб-пошук (використання зовнішньої пошукової системи, наприклад, Bing, для отримання URL та контенту сторінок).
- Вибір алгоритму пошуку: Користувач повинен мати можливість обрати один з 10 реалізованих алгоритмів для виконання пошуку: Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt (KMP), Bitap, Sunday, Naive (Brute Force), Horspool, Boyer-Moore-Horspool (BMH), Two-Way, Simon.
- Налаштування паралелізму: Користувач повинен мати можливість вказати кількість потоків/процесів (від 1 до 32) для виконання паралельного пошуку (для джерел «Файли» та «БД»).
- Виконання пошуку: Система повинна виконувати пошук обраним алгоритмом у вибраному джерелі з заданою кількістю потоків.
- Відображення результатів:
  - Для джерел «Файли» та «БД»: відображати список назв/ідентифікаторів знайдених документів.
  - Для джерела «Веб»: відображати список знайдених URL з коротким описом (як у пошуковій видачі).
  - Відображати час виконання пошуку.
  - Відображати інформацію про використаний алгоритм, джерело та кількість потоків.
- Порівняння алгоритмів: Система повинна надавати функцію порівняння продуктивності усіх реалізованих алгоритмів для заданого пошукового запиту, джерела даних та кількості потоків. Результати порівняння повинні включати:
  - Назву алгоритму.

- Час виконання (сек).
- Кількість знайдених результатів.
- Візуальне представлення відносної продуктивності (у % від найшвидшого).
- Визначення найшвидшого алгоритму для даного тесту. Таблиця має бути відсортована за зростанням часу виконання.

– Паралельна обробка: Система повинна використовувати модуль multiprocessing Python для паралельного виконання пошуку по окремих файлах або записах БД, якщо обрана кількість потоків > 1.

– Генерація даних: Система повинна передбачати використання скриптів (txt_generator.py, db_generator.py) для генерації та завантаження тестових даних.

Вимоги до складу й параметрів технічних засобів:

Для роботи ПЗ необхідний наступний склад технічних засобів з мінімальними/рекомендованими параметрами:

– Серверна частина (мінімальні):
- CPU: 2-ядерний процесор, 2.0 ГГц.
- RAM: 4 ГБ.
- HDD/SSD: 20 ГБ вільного місця (для ОС, ПЗ, БД та тестових даних).

– Серверна частина (рекомендовані для ефективного паралелізму та великих даних):
- CPU: 4+ ядерний процесор, 2.5+ ГГц.
- RAM: 8 ГБ+.
- SSD: 50 ГБ+ вільного місця.

– Клієнтська частина:
- Будь-який персональний комп'ютер, ноутбук або планшет, здатний запустити сучасний веб-браузер та підключитися до мережі, де розміщено сервер. Специфічних вимог до CPU/RAM клієнта немає, окрім тих, що потрібні для роботи браузера.

- Пристрої вводу-виводу: клавіатура, маніпулятор типу «миша», монітор.

Вимоги до інформаційної та програмної сумісності:

- Серверна частина:
  - Операційна система: Linux (рекомендовано), macOS або Windows, що підтримує Python 3.11+ та PostgreSQL 12+.
  - Інтерпретатор Python: Версія 3.11 або новіша.
  - СУБД: PostgreSQL версії 12 або новішої.
  - Необхідні бібліотеки Python: fastapi, uvicorn[standard], SQLAlchemy, psycopg2-binary, requests, beautifulsoup4, python-multipart.
- Клієнтська частина:
  - Сучасний веб-браузер з підтримкою HTML5, CSS3 та JavaScript (наприклад, Google Chrome, Mozilla Firefox, Microsoft Edge останніх версій).
- Взаємодія та формати:
  - Взаємодія з БД: Через стандартний інтерфейс SQLAlchemy та драйвер psycopg2.
  - Взаємодія з веб-джерелом (Bing API): Через стандартні HTTP-запити (бібліотека requests).
  - Формат текстових файлів для пошуку: UTF-8.
  - Формат файлу мапінгу текстів: texts.csv (CSV з роздільником ';').
  - Формат даних у БД: Таблиця text_documents з полями id (Integer, PK), title (String), content (Text), source_type (String).
  - Веб-інтерфейс: HTML, CSS, JavaScript.

2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ «SEARCHMASTER»

2.1 Аналіз вимог до програмного забезпечення «SearchMaster»

Аналіз вимог є фундаментальним етапом життєвого циклу розробки програмного забезпечення, що дозволяє визначити функціональні межі системи та очікування зацікавлених сторін. Для формалізації функціональних вимог до системи «SearchMaster» було застосовано метод моделювання варіантів використання (Use Case modeling). Цей підхід дозволяє описати поведінку системи з точки зору її користувачів, ідентифікуючи ключових акторів та їхні цілі.

2.1.1 Розробка моделі варіантів використання

Першим кроком у побудові моделі є ідентифікація акторів – зовнішніх сутностей, що взаємодіють із системою. Для програмного забезпечення «SearchMaster» було виявлено два основних актори, ролі та обов'язки яких описано в таблиці 2.1.

Таблиця 2.1 – Опис виявлених акторів системи

Актор	Опис ролі
Користувач (Дослідник)	Основний актор системи, який взаємодіє з веб-інтерфейсом для виконання текстового пошуку, обрання алгоритмів пошуку та джерел даних, аналізу результатів та порівняння продуктивності різних алгоритмів
Адміністратор системи	Вторинний актор, відповідальний за адміністрування системи: керування базою даних текстів, генерацію тестових даних, налаштування алгоритмів, моніторинг продуктивності та забезпечення функціонування системи

Для візуалізації взаємодії між акторами та функціональними можливостями системи була розроблена діаграма варіантів використання (рисунок 2.1). Вона демонструє межі системи «SearchMaster» та основні прецеденти, доступні кожному актору.

Рисунок 2.1 – Діаграма варіантів використання вебзастосунку SearchMaster

На основі розробленої діаграми було сформовано перелік основних прецедентів системи. Короткий опис кожного варіанту використання наведено в таблиці 2.2.

Таблиця 2.2 – Виявлені прецеденти системи

№	Назва варіанту використання	Короткий опис
1	2	3
1	Виконати пошук тексту	Користувач вводить текст для пошуку та отримує результати
2	Обрати алгоритм пошуку	Користувач обирає один або кілька алгоритмів для тестування
3	Обрати джерело даних	Користувач обирає джерело для пошуку (файли, БД, веб)

Кінець таблиці 2.2

1	2	3
4	Переглянути результати пошуку	Користувач переглядає знайдені входження та метрики
5	Порівняти продуктивність алгоритмів	Користувач аналізує час виконання різних алгоритмів
6	Керувати базою даних текстів	Адміністратор додає, редагує та видаляє тексти в базі даних
7	Генерувати тестові дані	Адміністратор створює тестові набори текстів для перевірки алгоритмів
8	Налаштовувати алгоритми	Адміністратор конфігурує параметри алгоритмів пошуку
9	Переглядати статистику використання	Адміністратор моніторить продуктивність системи та статистику запитів

Таким чином, розроблені варіанти використання та їхні короткі описи, представлені в таблиці 2.2, формують основний набір функціональних вимог до системи «SearchMaster».

2.1.2 Розробка специфікацій варіантів використання

Після ідентифікації акторів та основних прецедентів необхідно деталізувати кожен варіант використання. Специфікація прецеденту дозволяє формалізувати вимоги, описуючи послідовність дій, передумови та постумови для успішного виконання функції. Такий детальний опис є важливим для розробників та тестувальників, оскільки він усуває неоднозначність у трактуванні функціональності системи.

Нижче наведено специфікації для усіх ключових варіантів використання: «Виконати пошук тексту», що є основною функцією для Користувача, та «Керувати базою даних текстів», що є важливою адміністративною функцією.

У таблиці 2.3 детально описано прецедент «Виконати пошук тексту», включаючи основний та альтернативні потоки подій.

Таблиця 2.3 – Специфікація варіанту використання «Виконати пошук тексту»

Характеристика	Опис
Короткий опис	Користувач вводить рядок для пошуку, обирає алгоритм та джерело даних для отримання результатів з вимірюванням продуктивності
Дійові особи	Користувач (Дослідник)
Передумови	Додаток запущено, джерела даних доступні, обрано принаймні один алгоритм
Основний потік подій	1. Користувач відкриває головну сторінку 2. Вводить текст для пошуку у поле вводу 3. Обирає один або кілька алгоритмів для тестування 4. Обирає джерело даних (текстові файли, база даних PostgreSQL, веб-пошук) 5. Встановлює кількість потоків для паралельного виконання 6. Натискає кнопку "Пошук" 7. Система виконує пошук за допомогою обраних алгоритмів 8. Система вимірює час виконання для кожного алгоритму 9. Система відображає результати: знайдені входження, позиції, витрачений час
Альтернативний потік подій	4а. Рядок не знайдено – відображається повідомлення "Не знайдено" 6а. Помилка у джерелі даних – відображається повідомлення про помилку 7а. Перевищено час очікування – операція скасовується
Постумови	Результати пошуку та порівняння продуктивності відображені на сторінці результатів з детальною інформацією про кожний алгоритм

У таблиці 2.4 наведено специфікацію для прецеденту «Керувати базою даних текстів», який є критично важливим для підготовки тестових середовищ та адміністрування системи.

Таблиця 2.4 – Специфікація варіанту використання «Керувати базою даних текстів»

Характеристика	Опис
Короткий опис	Адміністратор керує колекцією текстових документів у базі даних, включаючи додавання нових текстів, редагування існуючих та генерацію тестових наборів даних
Дійові особи	Адміністратор системи
Передумови	База даних PostgreSQL налаштована та доступна, адміністратор має права доступу до системи
Основний потік подій	1. Адміністратор запускає утиліти адміністрування (db_generator.py, txt_generator.py) 2. Обирає тип операції: створення нових документів, видалення існуючих або генерація тестових даних 3. Встановлює параметри генерації: кількість документів, розмір тексту, ключові слова для пошуку 4. Система виконує операції з базою даних 5. Система генерує звіт про виконані операції 6. Адміністратор переглядає статистику оновленої бази даних
Альтернативний потік подій	3а. Помилка підключення до бази даних – відображається повідомлення про помилку 4а. Недостатньо місця для нових документів – система попереджає про обмеження 5а. Перевищено час виконання операції – операція скасовується з частковим результатом
Постумови	База даних текстів оновлена відповідно до заданих параметрів, система готова для проведення тестів пошуку

Окрім керування даними, Адміністратор системи відповідає за конфігурацію та розширення самого ядра системи – набору алгоритмів. Ця функціональність є ключовою для підтримки гнучкості та актуальності програмного забезпечення. Специфікація варіанту використання «Налаштовувати алгоритми» описана в таблиці 2.5.

Таблиця 2.5 – Специфікація варіанту використання «Налаштовувати алгоритми»

Характеристика	Опис
1	2
Короткий опис	Адміністратор керує конфігурацією алгоритмів пошуку, включаючи додавання нових алгоритмів, налаштування параметрів виконання та оптимізацію продуктивності системи
Дійові особи	Адміністратор системи
Передумови	Система SearchMaster запущена, адміністратор має доступ до конфігураційних файлів та коду алгоритмів
Основний потік подій	1. Адміністратор відкриває конфігураційні файли системи (main.py, algorithms/) 2. Переглядає поточний список доступних алгоритмів (algorithm_classes, algorithm_names) 3. Обирає тип операції: додавання нового алгоритму, модифікація існуючого або налаштування параметрів 4. При додаванні нового алгоритму: створює клас-нащадок BaseAlgorithm, реалізує метод search() 5. Оновлює словники algorithm_classes та algorithm_names 6. Налаштовує параметри паралельності (thread_count, ParallelAlgorithm) 7. Тестує новий/модифікований алгоритм на тестових даних 8. Переглядає статистику продуктивності для оптимізації параметрів 9. Зберігає зміни та перезапускає систему

Кінець таблиці 2.5

1	2
Альтернативний потік подій	4а. Помилка в реалізації алгоритму – відображається повідомлення про синтаксичну помилку 6а. Неправильні параметри паралельності – система попереджає про обмеження (1-32 потоки) 7а. Алгоритм працює некоректно – повернення до етапу налаштування 8а. Низька продуктивність – рекомендації щодо оптимізації
Постумови	Система оновлена з новими/модифікованими алгоритмами, конфігурація збережена, алгоритми доступні для користувачів через веб-інтерфейс

Після виконання пошуку користувач взаємодіє з результатами, що описано у специфікації прецеденту «Переглянути результати пошуку» (таблиця 2.6).

Таблиця 2.6 – Специфікація варіанту використання «Переглянути результати пошуку»

Характеристика	Опис
1	2
Короткий опис	Користувач обирає один або кілька алгоритмів пошуку для тестування з доступного переліку реалізованих алгоритмів
Дійові особи	Користувач (Дослідник)
Передумови	Веб-інтерфейс завантажений, доступний список алгоритмів (Boyer-Moore, Rabin-Karp, KMP, Bitap, Sunday, Naive, Horspool, BMH, Two-Way, Simon)

Кінець таблиці 2.6

1	2
Основний потік подій	1. Система відображає dropdown список доступних алгоритмів 2. Користувач переглядає список з назвами алгоритмів 3. Обирає потрібний алгоритм з випадаючого меню 4. Система збереже вибір алгоритму в локальній змінній 5. Відображається обраний алгоритм у формі пошуку
Альтернативний потік подій	3а. Алгоритм недоступний через помилку ініціалізації – відображається повідомлення про помилку 4а. Помилка збереження вибору – використовується алгоритм за замовчуванням (Boyer-Moore)
Постумови	Обраний алгоритм встановлений як активний для виконання пошуку

Важливою дослідницькою функцією системи є порівняння продуктивності, яке розширює базовий перегляд результатів. Цей процес деталізовано у таблиці 2.7.

Таблиця 2.7 – Специфікація варіанту використання «Порівняти продуктивність алгоритмів»

Характеристика	Опис
1	2
Короткий опис	Користувач обирає джерело даних для виконання пошуку: текстові файли, база даних PostgreSQL або веб-пошук
Дійові особи	Користувач (Дослідник)
Передумови	Система запущена, доступні джерела даних: файли (texts/), база даних (PostgreSQL), веб-сервіс

Кінець таблиці 2.7

1	2
Основний потік подій	1. Система відображає радіо-кнопки або dropdown для вибору джерела 2. Користувач переглядає доступні опції: "Text Files", "PostgreSQL Database", "Web Search" 3. Обирає потрібне джерело даних 4. Система перевіряє доступність обраного джерела 5. Ініціалізується відповідний SearchSource (FileSource, DBSource, WebSource) 6. Відображається підтвердження вибору джерела
Альтернативний потік подій	4a. Джерело недоступне (відсутні файли, помилка з'єднання з БД, немає інтернету) – повідомлення про помилку 5a. Помилка ініціалізації джерела – повернення до вибору за замовчуванням (файли)
Постумови	Обране джерело даних активне та готове для виконання пошуку

Для забезпечення якісного тестування та адміністрування системи передбачено низку функцій для Адміністратора. У таблиці 2.8 наведено специфікацію для прецеденту «Керувати базою даних текстів».

Таблиця 2.8 – Специфікація варіанту використання «Керувати базою даних текстів»

Характеристика	Опис
1	2
Короткий опис	Користувач переглядає детальні результати пошуку, включаючи знайдені входження, їх позиції, час виконання та метрики продуктивності

Кінець таблиці 2.8

1	2
Дійові особи	Користувач (Дослідник)
Передумови	Пошук виконано успішно, є результати для відображення
Основний потік подій	1. Система перенаправляє на сторінку результатів (search_result.html) 2. Відображається пошуковий запит та обрані параметри 3. Показуються знайдені тексти з виділенням шуканих фрагментів 4. Відображається час виконання алгоритму 5. Показується кількість знайдених входжень 6. Користувач може переглядати повний текст документів 7. Доступна можливість повернутися до нового пошуку
Альтернативний потік подій	2а. Нічого не знайдено – відображається повідомлення "Не знайдено" 3а. Помилка у форматуванні результатів – показується сирий текст 6а. Великий обсяг результатів – реалізується пагінація
Постумови	Користувач ознайомлений з результатами пошуку та може прийняти рішення про наступні дії

Прецедент «Керувати базою даних текстів» включає в себе генерацію тестових даних, що є окремим важливим підпроцесом, описаним у таблиці 2.9.

Таблиця 2.9 – Специфікація варіанту використання «Генерувати тестові дані»

Характеристика	Опис
Короткий опис	Користувач запускає порівняльний тест всіх доступних алгоритмів для аналізу їх продуктивності на однакових даних
Дійові особи	Користувач (Дослідник)
Передумови	Введено пошуковий запит, система має доступ до endpoint /performance
Основний потік подій	1. Користувач натискає кнопку "Compare Algorithms Performance" 2. Система відображає індикатор завантаження "Testing algorithms..." 3. Виконується послідовний запуск усіх 10 алгоритмів з однаковими параметрами 4. Вимірюється час виконання кожного алгоритму (execution_time) 5. Збираються метрики: кількість результатів, успішність виконання 6. Результати сортуються за часом виконання (найшвидший спочатку) 7. Відображається таблиця порівняння з графічними індикаторами продуктивності 8. Показуються рекомендації щодо оптимального алгоритму
Альтернативний потік подій	3а. Помилка в одному з алгоритмів – пропускається з позначкою "Error" 4а. Перевищено timeout – алгоритм позначається як "Timeout" 7а. Забагато результатів для відображення – показуються топ-5 алгоритмів
Постумови	Користувач має повну картину порівняльної продуктивності алгоритмів для прийняття рішень

Окрім керування даними, Адміністратор системи відповідає за конфігурацію та розширення самого ядра системи – набору алгоритмів. Ця функціональність є ключовою для підтримки гнучкості та актуальності програмного забезпечення. Специфікація варіанту використання «Налаштовувати алгоритми» описана в таблиці 2.10.

Таблиця 2.10 – Специфікація варіанту використання «Налаштовувати алгоритми»

Характеристика	Опис
Короткий опис	Адміністратор створює тестові набори текстових документів з контрольованими параметрами для якісного тестування алгоритмів пошуку
Дійові особи	Адміністратор системи
Передумови	Доступні скрипти генерації (txt_generator.py), система має права на створення файлів
Основний потік подій	1. Адміністратор запускає скрипт txt_generator.py 2. Встановлює параметри генерації: кількість файлів, розмір тексту, мову 3. Визначає ключові слова для включення в тексти (для перевірки пошуку) 4. Обирає тип контенту: випадковий текст, література, технічні тексти 5. Система генерує файли в директорії texts/ 6. Оновлюється файл texts.csv з відповідностями ID-назва 7. Створюється звіт про згенеровані файли 8. Виконується тестовий пошук на згенерованих даних
Альтернативний потік подій	5а. Недостатньо місця на диску – повідомлення про помилку 6а. Помилка запису в texts.csv – ручне оновлення файлу 8а. Тестовий пошук не знаходить ключові слова – перегенерація даних
Постумови	Створено новий набір тестових текстів, система готова для проведення експериментів

Нарешті, для моніторингу та оптимізації роботи системи Адміністратор має можливість переглядати статистику її використання, що деталізовано в таблиці 2.11.

Таблиця 2.11 – Специфікація варіанту використання «Переглядати статистику використання»

Характеристика	Опис
Короткий опис	Адміністратор моніторить статистику використання системи, включаючи популярні запити, продуктивність алгоритмів та навантаження
Дійові особи	Адміністратор системи
Передумови	Система збирає метрики використання, доступ до логів та статистичних даних
Основний потік подій	1. Адміністратор відкриває інтерфейс статистики або переглядає лог-файли 2. Аналізує метрики використання: кількість запитів, популярні алгоритми 3. Переглядає статистику продуктивності: середній час виконання по алгоритмах 4. Вивчає частоту використання різних джерел даних 5. Аналізує найпопулярніші пошукові запити 6. Переглядає технічні метрики: використання потоків, навантаження системи 7. Генерує звіти для оптимізації конфігурації системи
Альтернативний потік подій	2а. Недостатньо даних для аналізу – повідомлення про необхідність більше часу збору 3а. Аномальні показники продуктивності – рекомендації щодо діагностики 7а. Помилка генерації звіту – експорт сирих даних
Постумови	Адміністратор має актуальну інформацію для прийняття рішень щодо оптимізації та розвитку системи

Таким чином, деталізовані специфікації всіх ключових варіантів використання, представлені в таблицях 2.3-2.11, утворюють повний набір функціональних вимог, що є фундаментальною основою для подальшого етапу проєктування архітектури системи «SearchMaster» та її реалізації. Вони забезпечують чітке розуміння функціоналу як для розробників, так і для тестувальників, сприяючи послідовній та ефективній розробці.

2.2 Розробка архітектури програмного забезпечення «SearchMaster»

Розробка архітектури є ключовим етапом проєктування, на якому визначається загальна структура системи, її основні компоненти та взаємозв'язки між ними. Правильно обрана архітектура забезпечує надійність, масштабованість, гнучкість та легкість супроводу програмного продукту. Для системи «SearchMaster» було обрано модульну архітектуру, що дозволяє чітко розділити відповідальність між різними частинами застосунку.

На рисунку 2.2 представлена діаграма компонентів, яка візуалізує логічну структуру вебзастосунку. Вона ілюструє, як контролер (Controller) обробляє HTTP-запити, взаємодіє з шаром представлення (View Layer) та статичними файлами, а також використовує модельний шар (Model Layer) для доступу до бізнес-логіки та даних. Модельний шар, у свою чергу, складається з компонентів логіки пошуку (Search Logic), джерел даних (Data Sources) та моделей бази даних (Database Models), які взаємодіють із зовнішніми системами.

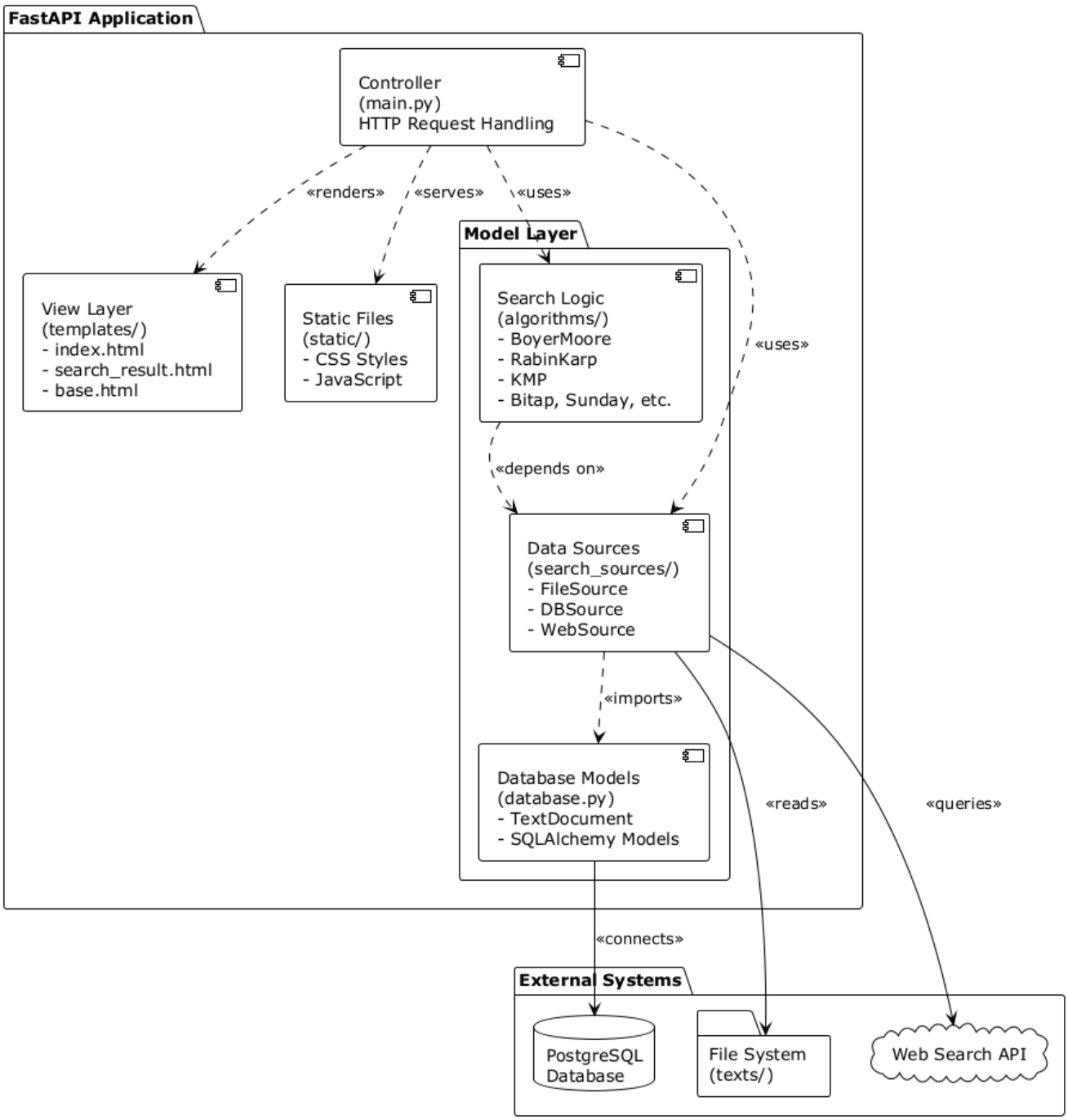


Рисунок 2.2 – Діаграма компонентів вебзастосунку SearchMaster

Для розуміння фізичного розміщення компонентів системи була розроблена діаграма розгортання (рисунок 2.3). Вона показує, як програмні артефакти розподілені по вузлах інфраструктури. Клієнтський вузол містить веб-браузер, серверний вузол – основний FastAPI додаток, а вузли даних включають файлове сховище та базу даних PostgreSQL. Також показано взаємодію із зовнішніми веб-сервісами.

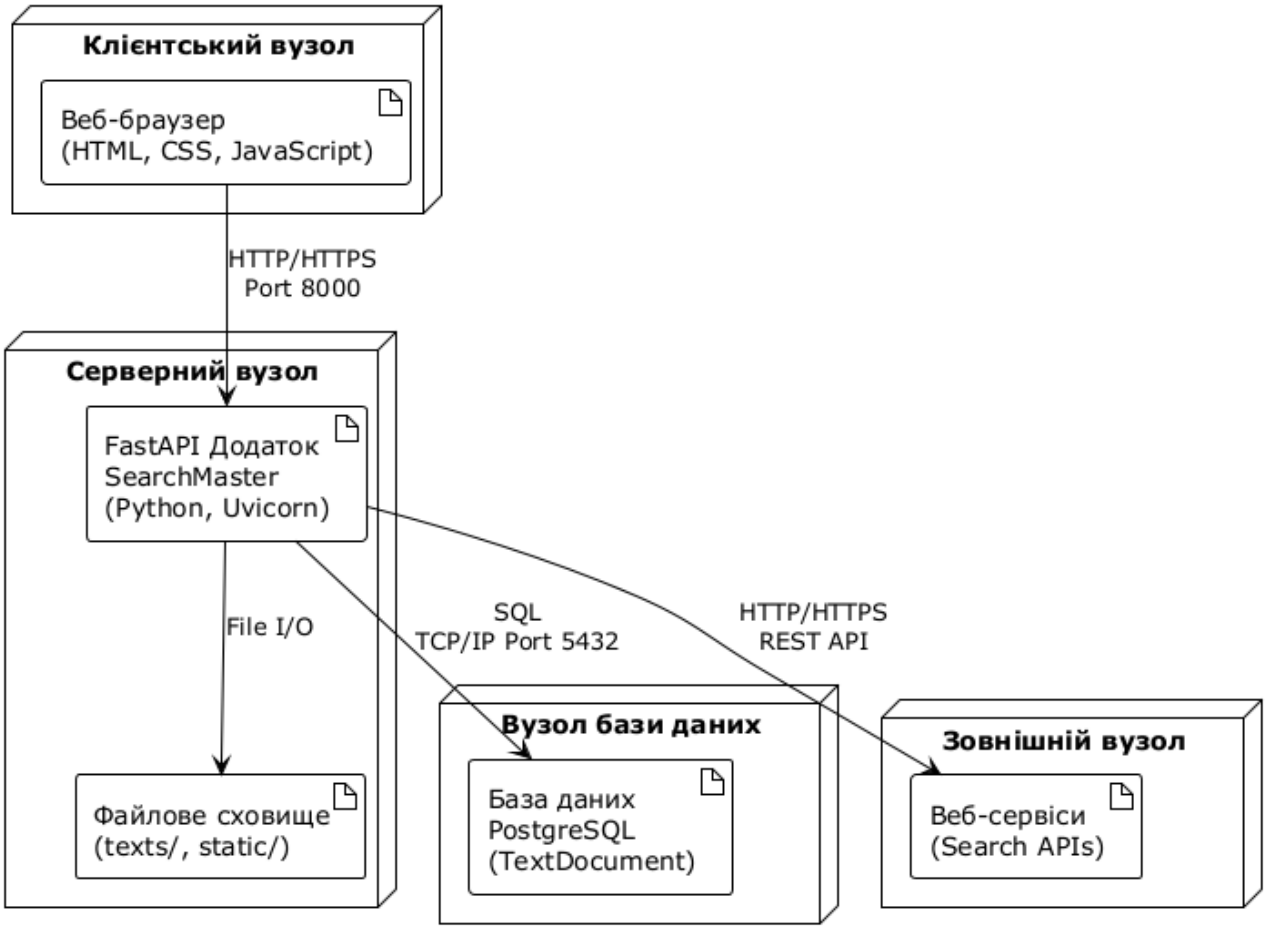


Рисунок 2.3 – Діаграма розгортання вебзастосунку SearchMaster

Таким чином, обрана архітектура розгортання забезпечує гнучкість системи, дозволяючи масштабувати кожен компонент незалежно та інтегрувати зовнішні джерела даних для розширення її функціональності. Це також підкреслює розподілений характер застосунку, оптимізований для ефективного виконання пошукових запитів та керування даними.

2.3 Проєкт програмного забезпечення «SearchMaster»

На основі розробленої архітектури було створено проєкт програмного забезпечення, що включає ескізний, технічний та робочий етапи.

2.3.1 Ескізний проєкт

Ескізний проєкт фокусується на візуалізації поведінки системи та розробці прототипів користувацького інтерфейсу.

Для деталізації основного сценарію взаємодії користувача з системою – виконання пошуку – була створена діаграма діяльності (рисунок 2.4). Вона покроково ілюструє логіку процесу: від відображення сторінки пошуку та отримання даних від користувача до валідації, ініціалізації компонентів, виконання пошуку та відображення результатів.

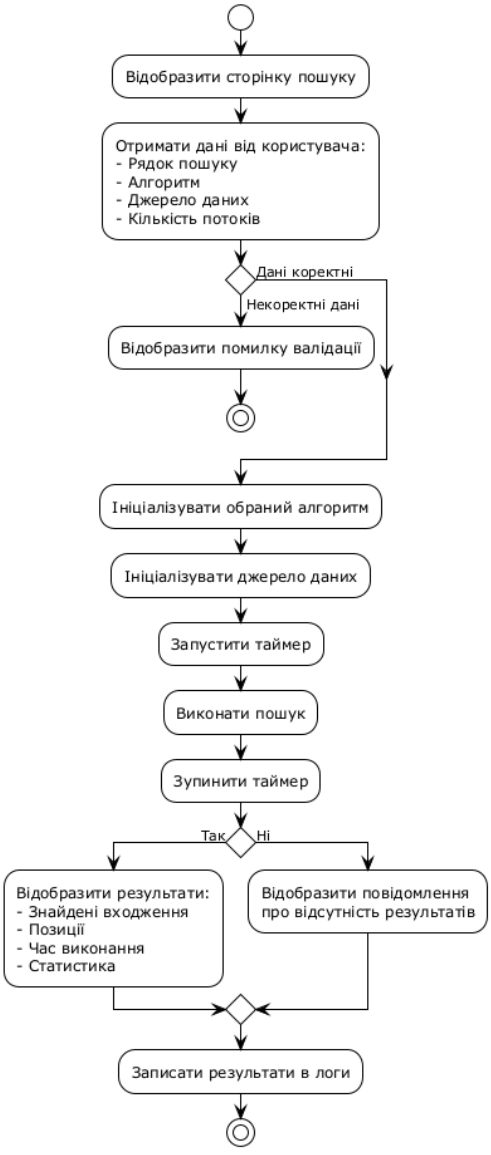


Рисунок 2.4 – Діаграма діяльності варіанту використання «Виконати пошук тексту»

Для візуалізації користувацького досвіду та узгодження вимог до інтерфейсу було розроблено прототипи основних екранів. На рисунку 2.5 зображено головний інтерфейс системи «SearchMaster», який містить поле для введення запиту, селектори для вибору параметрів та кнопки для запуску пошуку та порівняння алгоритмів.

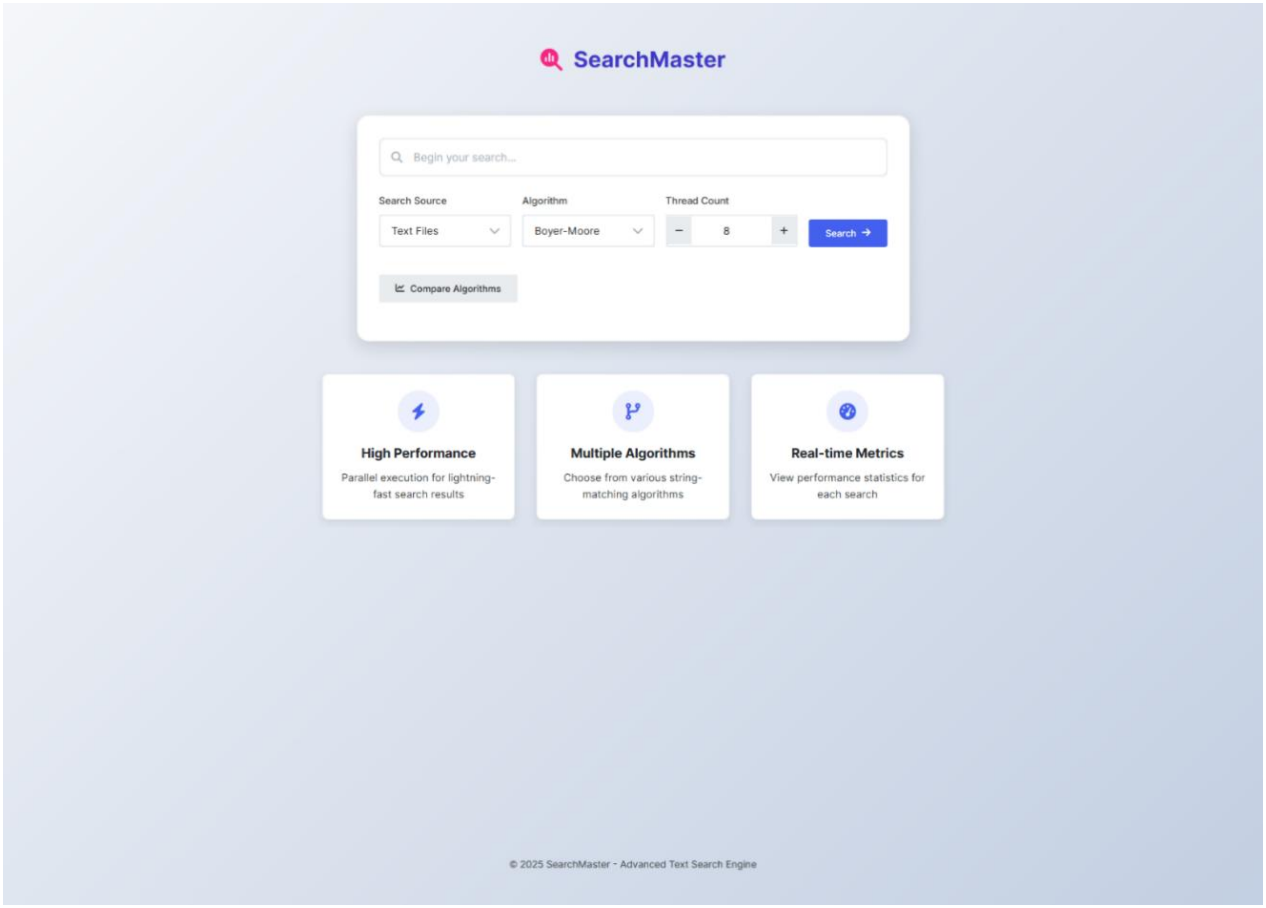


Рисунок 2.5 – Головний інтерфейс SearchMaster

На рисунку 2.6 показано випадаючий список для вибору джерела даних, що дозволяє користувачеві легко перемикатися між пошуком у файлах, базі даних або в Інтернеті.

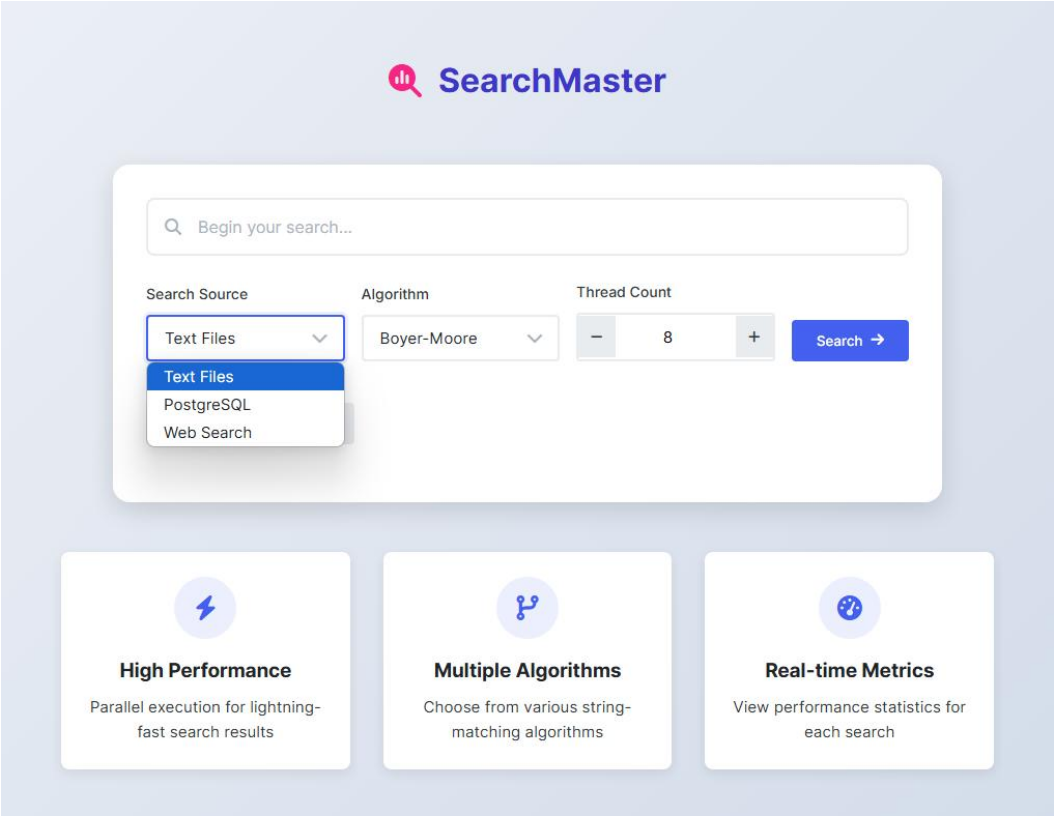


Рисунок 2.6 – Вибір джерела даних

На рисунку 2.7 продемонстровано вибір алгоритму пошуку з випадаючого списку, де представлені всі 10 реалізованих у системі алгоритмів.

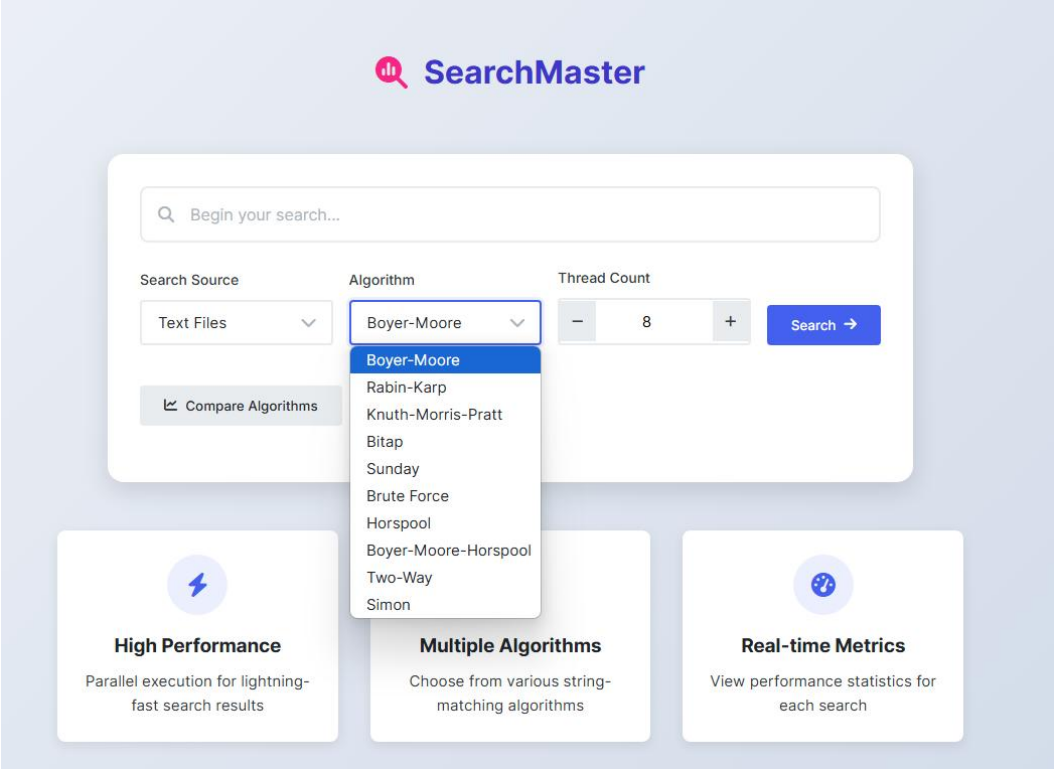


Рисунок 2.7 – Вибір алгоритму пошуку

Ці прототипи інтерфейсу (Рисунки 2.5-2.7) демонструють інтуїтивно зрозуміле керування основними параметрами пошуку, забезпечуючи зручність взаємодії користувача з системою перед виконанням запиту. Вони підтверджують, що функціональні вимоги до інтерфейсу відображені в ескізному проєкті.

2.3.2 Технічний проєкт

Технічний проєкт деталізує внутрішню структуру системи, описуючи статичні елементи, такі як класи та їхні взаємозв'язки, а також структуру даних, що зберігаються. Цей етап є основою для подальшої реалізації програмного коду.

Для представлення статичної структури системи «SearchMaster» була розроблена діаграма класів (рисунок 2.8). Вона ілюструє основні класи, їхні атрибути, методи та зв'язки між ними. Центральними елементами є абстрактні класи BaseAlgorithm та BaseSearchSource, від яких успадковуються конкретні реалізації алгоритмів (напр., BoyerMoore, KMP) та джерел даних (FileSource, DBSource, WebSource). Клас SearchController виступає як точка входу для обробки запитів, а ParallelAlgorithm слугує обгорткою для реалізації багатопоточності.

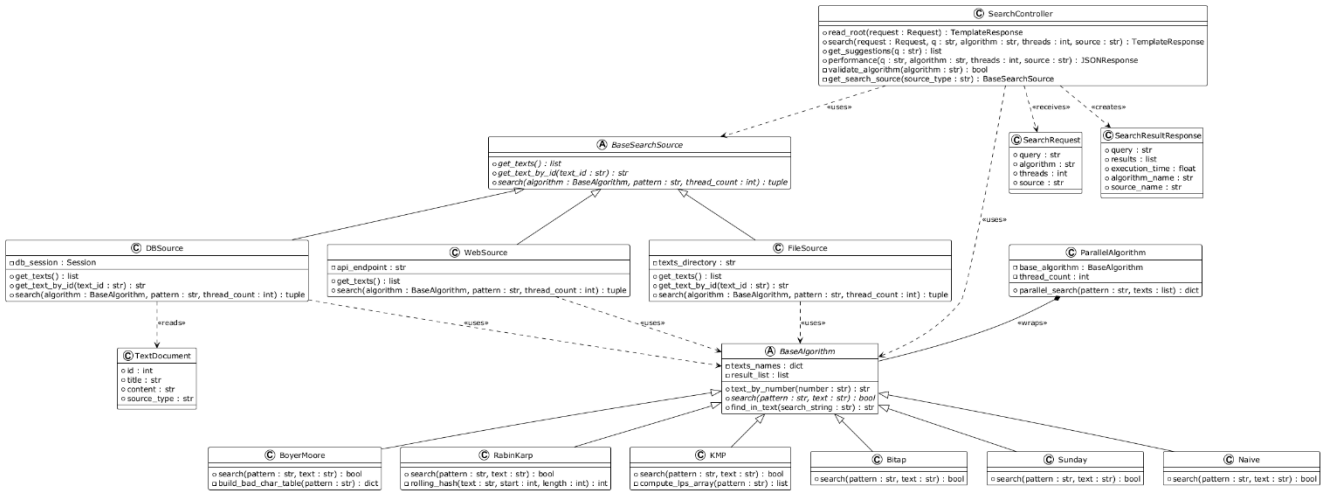


Рисунок 2.8 – Діаграма класів вебзастосунку SearchMaster

Нижче наведено детальні специфікації для класів, зображених на діаграмі.

Базові абстрактні класи

Клас BaseAlgorithm

- Призначення: Базовий абстрактний клас для всіх алгоритмів пошуку, забезпечує спільну функціональність та інтерфейс
- Атрибути:
  - texts_names: dict – словник для зберігання відповідностей між номерами текстів та їх назвами/шляхами
  - result_list: list – список результатів пошуку, містить назви файлів де знайдено збіги
- Методи:
  - text_by_number(number: str): str – завантажує та повертає вміст текстового файлу за його номером
  - search(pattern: str, text: str): bool – абстрактний метод для реалізації конкретного алгоритму пошуку
  - find_in_text(search_string: str): str – виконує пошук у всіх доступних текстах та формує результуючий рядок

Клас BaseSearchSource

- Призначення: Абстрактний базовий клас для всіх джерел даних, визначає єдиний інтерфейс доступу до різних типів джерел
- Атрибути: Немає власних атрибутів (абстрактний клас)
- Методи:
  - get_texts(): list – абстрактний метод для отримання списку всіх доступних текстів у форматі (id, title, content)
  - get_text_by_id(text_id: str): str – абстрактний метод для отримання конкретного тексту за його ідентифікатором
  - search(algorithm, pattern, thread_count): tuple – абстрактний метод для виконання пошуку з використанням заданого алгоритму

Конкретні алгоритми пошуку

Клас BoyerMoore

- Призначення: Реалізація алгоритму Boyer-Moore для ефективного пошуку підрядків
- Атрибути: Успадковує від BaseAlgorithm
- Методи:
  - search(pattern: str, text: str): bool – реалізує алгоритм Boyer-Moore з використанням таблиці поганих символів
  - build_bad_char_table(pattern: str): dict – будує таблицю зсувів для символів патерну

Клас RabinKarp

- Призначення: Реалізація алгоритму Rabin-Karp з використанням rolling hash для швидкого пошуку
- Атрибути: Успадковує від BaseAlgorithm
- Методи:
  - search(pattern: str, text: str): bool – реалізує алгоритм Rabin-Karp з хешуванням
  - rolling_hash(text: str, start: int, length: int): int – обчислює rolling hash для підрядка

Клас KMP

- Призначення: Реалізація алгоритму Knuth-Morris-Pratt для лінійного пошуку без повторних порівнянь
- Атрибути: Успадковує від BaseAlgorithm
- Методи:
  - search(pattern: str, text: str): bool – реалізує алгоритм KMP з використанням LPS масиву
  - compute_lps_array(pattern: str): list – обчислює масив longest proper prefix suffix

Клас Bitap

- Призначення: Реалізація алгоритму Bitap (Baeza-Yates-Gonnet) з бітовими операціями
- Атрибути: Успадковує від BaseAlgorithm
- Методи:
  - search(pattern: str, text: str): bool – реалізує алгоритм Bitap з бітовими масками

Клас Sunday

- Призначення: Реалізація алгоритму Sunday – модифікації Boyer-Moore з оптимізованими зсувами
- Атрибути: Успадковує від BaseAlgorithm
- Методи:
  - search(pattern: str, text: str): bool – реалізує алгоритм Sunday

Клас Naive

- Призначення: Реалізація наївного (brute force) алгоритму пошуку для порівняння продуктивності
- Атрибути: Успадковує від BaseAlgorithm
- Методи:
  - search(pattern: str, text: str): bool – реалізує простий перебір всіх позицій

Допоміжні класи

Клас ParallelAlgorithm

- Призначення: Обгортка для виконання будь-якого алгоритму в паралельному режимі з використанням багатопотоковості
- Атрибути:
  - base_algorithm: BaseAlgorithm – екземпляр алгоритму для паралельного виконання

- thread_count: int – кількість потоків для паралельної обробки (1-32)

- Методи:
  - parallel_search(pattern: str, texts: list): dict – виконує паралельний пошук по списку текстів

Джерела даних

Клас FileSource

- Призначення: Реалізація джерела даних для роботи з локальними текстовими файлами
- Атрибути:
  - texts_directory: str – шлях до директорії з текстовими файлами
- Методи:
  - get_texts(): list – завантажує всі тексти з директорії texts/ та файлу texts.csv
  - get_text_by_id(text_id: str): str – читає конкретний файл за ідентифікатором
  - search(algorithm, pattern, thread_count): tuple – виконує пошук у файлах з використанням ParallelAlgorithm

Клас DBSource

- Призначення: Реалізація джерела даних для роботи з базою даних PostgreSQL
- Атрибути:
  - db_session: Session – сесія SQLAlchemy для роботи з базою даних
- Методи:
  - get_texts(): list – отримує всі TextDocument записи з бази даних
  - get_text_by_id(text_id: str): str – отримує конкретний документ за ID

- search(algorithm, pattern, thread_count): tuple – виконує паралельний пошук по документах БД

Клас WebSource

- Призначення: Реалізація джерела даних для веб-пошуку через зовнішні API (Bing Search)
- Атрибути:
  - api_endpoint: str – URL ендпойнту для веб-пошуку
- Методи:
  - get_texts(): list – для веб-пошуку повертає порожній список (тексти отримуються динамічно)
  - search(algorithm, pattern, thread_count): tuple – виконує веб-пошук та обробляє HTML результати

Контролер та моделі даних

Клас SearchController

- Призначення: Основний контролер FastAPI для обробки HTTP запитів та координації роботи всіх компонентів
- Атрибути: Немає постійних атрибутів (stateless контролер)
- Методи:
  - read_root(request: Request): TemplateResponse – відображає головну сторінку з формою пошуку
  - search(request, q, algorithm, threads, source): TemplateResponse – виконує основний пошук та відображає результати
  - get_suggestions(q: str): list – повертає автодоповнення для пошукових запитів
  - performance(q, algorithm, threads, source): JSONResponse – виконує тест продуктивності конкретного алгоритму

- validate_algorithm(algorithm: str): bool – перевіряє валідність обраного алгоритму
- get_search_source(source_type: str): BaseSearchSource – створює відповідний екземпляр джерела даних

Клас TextDocument

- Призначення: ORM модель для зберігання текстових документів у базі даних PostgreSQL
- Атрибути:
  - id: int – унікальний ідентифікатор документа (primary key)
  - title: str – назва документа для відображення в результатах
  - content: str – повний текстовий вміст документа для пошуку
  - source_type: str – тип джерела даних (за замовчуванням "database")

Клас SearchRequest

- Призначення: Pydantic модель для валідації вхідних параметрів пошукових запитів
- Атрибути:
  - query: str – пошуковий запит користувача
  - algorithm: str – назва обраного алгоритму
  - threads: int – кількість потоків для виконання
  - source: str – тип джерела даних

Клас SearchResultResponse

- Призначення: Pydantic модель для структурованого повернення результатів пошуку
- Атрибути:
  - query: str – оригінальний пошуковий запит

- results: list – список знайдених результатів
- execution_time: float – час виконання пошуку в секундах
- algorithm_name: str – повна назва використаного алгоритму
- source_name: str – назва джерела даних для відображення

Для проєктування структури бази даних, яка використовується для зберігання текстових документів та результатів пошуку, була створена діаграма «сутність-зв'язок» (ER-діаграма), наведена на рисунку 2.9. Вона визначає ключові сутності: Algorithm (алгоритм), SearchSource (джерело даних), TextDocument (текстовий документ), SearchResult (результат пошуку) та SearchSession (сесія пошуку), а також атрибути та зв'язки між ними. Ця модель є основою для створення таблиць у реляційній базі даних PostgreSQL.

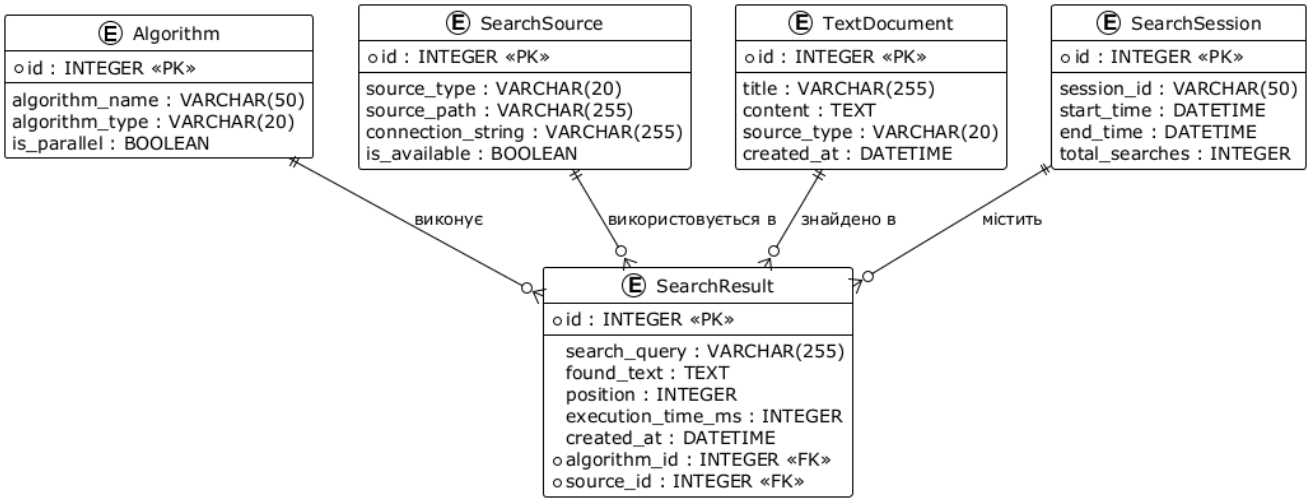


Рисунок 2.9 – Діаграма «сутність-зв'язок» вебзастосунку SearchMaster

Діаграма відображає структуру бази даних системи SearchMaster з п'ятьма основними сутностями та їх взаємозв'язками.

– Центральна сутність SearchResult містить лише два зовнішні ключі (algorithm_id та source_id), що обумовлено архітектурними рішеннями системи:

– Зв'язок з Algorithm (1:N): Кожний результат пошуку обов'язково використовує конкретний алгоритм (Boyer-Moore, KMP, Rabin-Karp тощо)

- Зв'язок з SearchSource (1:N): Кожний пошук виконується в конкретному джерелі даних (файли, база даних, веб)
- Зв'язок з TextDocument (логічний): Прямий FK відсутній, оскільки один пошук може знайти збіги у декількох документах одночасно. Результати зберігаються як агрегований текст у полі found_text
- Зв'язок з SearchSession (1:N): Декілька пошуків можуть належати одній користувацькій сесії для групування пов'язаних запитів

Така архітектура забезпечує гнучкість системи, дозволяючи одному результату пошуку описувати знахідки у багатьох документах без необхідності створення додаткових таблиць зв'язків Many-to-Many, що спрощує структуру БД та підвищує продуктивність запитів.

2.3.3 Робочий проєкт

Робочий проєкт включає обґрунтування вибору технологічного стеку та детальний опис реалізації основних програмних модулів системи.

Обґрунтування вибору засобів розробки

Вибір технологічного стеку є фундаментальним етапом проєктування, який визначає архітектурні можливості, продуктивність, масштабованість та надійність програмного забезпечення. Для проєкту «SearchMaster» вибір технологій здійснювався на основі ретельного аналізу функціональних та нефункціональних вимог.

Вибір технологічного стеку є фундаментальним етапом проєктування, який визначає архітектурні можливості, продуктивність, масштабованість, надійність та вартість розробки і підтримки програмного забезпечення. Для проєкту «SearchMaster», метою якого є створення гнучкого та ефективного інструменту для порівняльного аналізу алгоритмів повнотекстового пошуку, вибір технологій здійснювався на основі ретельного аналізу функціональних та нефункціональних вимог, викладених у Технічному завданні (Додаток А). Ключовими критеріями

були: висока продуктивність обчислень (особливо для CPU-bound алгоритмів пошуку), можливість реалізації паралельної обробки, гнучкість у роботі з різними джерелами даних (файли, БД, веб), швидкість розробки та можливість створення інтерактивного веб-інтерфейсу.

Мовою програмування для реалізації серверної частини (бекенду) було обрано Python версії 3.11+ [13]. Цей вибір обґрунтований кількома факторами. По-перше, Python має надзвичайно багату стандартну бібліотеку та величезну екосистему сторонніх пакетів, що значно спрощує реалізацію різноманітних завдань: від роботи з текстом та рядками (що є основою проєкту) до взаємодії з базами даних, мережевих запитів та створення веб-сервісів. По-друге, читабельний та лаконічний синтаксис Python дозволяє прискорити процес розробки та полегшує подальшу підтримку коду. По-третє, Python має потужні та добре документовані засоби для реалізації паралельних та асинхронних обчислень, зокрема стандартний модуль multiprocessing, який є критично важливим для виконання вимоги щодо дослідження впливу паралелізму на продуктивність пошуку [9]. Хоча Python є інтерпретованою мовою, сучасні реалізації та використання оптимізованих бібліотек (наприклад, написаних на C) дозволяють досягти високої продуктивності для багатьох задач, а для CPU-bound операцій, як пошук рядків, multiprocessing дозволяє ефективно задіяти всі ядра процесора.

Для створення веб-сервера та API було обрано фреймворк FastAPI [14]. Це сучасний, високопродуктивний Python-фреймворк, що базується на стандартах ASGI (Asynchronous Server Gateway Interface). Його ключовими перевагами, що зумовили вибір, є: виняткова швидкість роботи, порівнянна з фреймворками на компільованих мовах (Node.js, Go), що важливо для забезпечення швидкого відгуку системи; вбудована підтримка асинхронного програмування (async/await), що дозволяє ефективно обробляти операції введення-виведення (наприклад, очікування відповіді від БД чи зовнішнього веб-сервісу) без блокування основного потоку; автоматична валідація даних запитів та відповідей на основі стандартних підказок типів Python (завдяки інтеграції з Pydantic), що підвищує надійність коду; автоматична генерація інтерактивної документації API (Swagger UI та ReDoc), що

значно спрощує тестування та використання API. Простота та мінімалістичність FastAPI також сприяють швидкій розробці.

Для запуску FastAPI-додатку необхідний ASGI-сервер. Було обрано Uvicorn, оскільки він є рекомендованим сервером для FastAPI, відомий своєю високою продуктивністю та повною підтримкою асинхронних можливостей стандарту ASGI.

Як система управління базами даних для зберігання текстових документів (одне з джерел даних) було обрано PostgreSQL версії 12+ [15]. Це потужна, об'єктно-реляційна СУБД з відкритим кодом, що має репутацію надійної, стабільної та функціональної системи. PostgreSQL ефективно працює з великими обсягами текстових даних завдяки типу TEXT та вбудованим можливостям повнотекстового пошуку (хоча в «SearchMaster» пошук реалізується на рівні додатку для порівняння базових алгоритмів). Важливими перевагами є відповідність стандартам SQL, активна спільнота розробників та добра підтримка з боку Python через надійні драйвери та ORM.

Для взаємодії з PostgreSQL з Python-коду було обрано бібліотеку SQLAlchemy. Це найпопулярніший та найпотужніший інструмент для роботи з реляційними БД у Python. SQLAlchemy надає як високорівневий ORM (Object-Relational Mapper), що дозволяє працювати з таблицями та записами як з об'єктами Python, так і низькорівневий Core API для побудови SQL-запитів. Використання ORM значно спрощує операції CRUD (Create, Read, Update, Delete), визначення схеми даних (через класи-моделі) та управління сесіями бази даних, абстрагуючи розробника від написання «сирого» SQL.

Для реалізації вимоги щодо дослідження впливу паралелізму на пошук у файлах та БД було використано стандартний модуль Python multiprocessing [9]. На відміну від модуля threading, який в CPython обмежений Global Interpreter Lock (GIL) і не дає справжнього паралелізму для CPU-bound задач, multiprocessing створює окремі процеси ОС, кожен зі своїм інтерпретатором Python та пам'яттю. Це дозволяє повною мірою використовувати потужності багатоядерних процесорів для таких обчислювально інтенсивних завдань, як сканування великих текстів.

Інтерфейс multiprocessing.Pool з методами map/starmap надає зручний спосіб для розподілу великої кількості незалежних завдань (пошук в окремому файлі/записі БД) між пулом робочих процесів.

Для генерації HTML-сторінок на сервері використовується шаблонізатор Jinja2. Він є стандартним вибором для багатьох Python веб-фреймворків, включаючи FastAPI (через fastapi.templating). Jinja2 надає потужні можливості, такі як наслідування шаблонів, макроси, фільтри, умовні конструкції та цикли, що дозволяє створювати складні та динамічні веб-сторінки, відокремлюючи логіку представлення від основної логіки додатку.

Фронтенд-частина реалізована з використанням стандартних веб-технологій: HTML5 для структури сторінок, CSS3 для візуального оформлення та адаптивності, та JavaScript (Vanilla JS) для реалізації інтерактивності. Враховуючи відносну простоту користувацького інтерфейсу (одна основна сторінка з формами та динамічним блоком результатів порівняння), використання складних фронтенд-фреймворків (React, Vue, Angular) було визнано надлишковим. Vanilla JS використовується для обробки подій (натискання кнопок), виконання асинхронних запитів до бекенду (за допомогою Fetch API) для отримання результатів порівняння та динамічного оновлення DOM (відображення таблиці порівняння, індикаторів завантаження, toast-повідомлень) без перезавантаження сторінки.

Для реалізації функції веб-пошуку необхідні інструменти для взаємодії з зовнішніми веб-ресурсами. Бібліотека Requests була обрана для виконання HTTP-запитів до пошукової системи Bing. Це де-факто стандарт для HTTP-клієнтів у Python завдяки своєму простому та елегантному API. Для обробки HTML-відповідей від Bing та вилучення необхідної інформації (URL, заголовки, сніпети) використовується бібліотека BeautifulSoup4. Вона відома своєю здатністю парсити навіть некоректно сформований HTML та надає зручні методи для навігації по DOM-дереву та пошуку елементів за допомогою CSS-селекторів.

Для управління вихідним кодом проєкту використовується система контролю версій Git. Це стандарт індустрії, що забезпечує надійне відстеження

змін, можливість розгалуження та злиття коду, а також полегшує командну роботу та інтеграцію з платформами типу GitHub чи GitLab.

Як основне середовище розробки було обрано Visual Studio Code (VS Code). Це безкоштовний, потужний та розширюваний редактор коду з відмінною підтримкою Python (включаючи автодоповнення, лінтинг, налагодження), інтеграцією з Git, вбудованим терміналом та великою кількістю корисних розширень, що значно підвищує продуктивність розробника.

Отже, вибір кожного компонента технологічного стеку був ретельно обґрунтований з огляду на специфічні вимоги проєкту «SearchMaster», що дозволило створити функціональну, продуктивну та гнучку систему для досягнення поставленої мети.

Опис реалізації ключових модулів

Розробка ПЗ «SearchMaster» здійснювалася на основі модульної архітектури, що забезпечує гнучкість, розширюваність та легкість супроводу. Нижче наведено опис структури проєкту та деталі реалізації ключових компонентів.

Загальна структура проєкту:

Проєкт має наступну структуру директорій та файлів:

```
├── algorithms/              # Модулі алгоритмів пошуку
│   ├── base.py              # Базовий клас BaseAlgorithm
│   ├── parallel/            # Модуль паралелізації
│   │   └── base.py          # Клас ParallelAlgorithm
│   ├── boyer_moore.py       # Реалізація Boyer-Moore
│   ├── rabin_karp.py        # Реалізація Rabin-Karp
│   ├── kmp.py               # Реалізація Knuth-Morris-Pratt
│   ├── bitap.py             # Реалізація Bitap
│   ├── sunday.py            # Реалізація Sunday
│   ├── naive.py             # Реалізація Naive (Brute Force)
│   ├── horspool.py          # Реалізація Horspool
```

```
│   ├── bmh.py              # Реалізація Boyer-Moore-Horspool
│   ├── two_way.py          # Реалізація Two-Way
│   └── simon.py            # Реалізація Simon
├── search_sources/         # Модулі джерел даних
│   ├── base.py             # Базовий клас BaseSearchSource
│   ├── file_source.py      # Реалізація для файлів
│   ├── db_source.py        # Реалізація для БД PostgreSQL
│   └── web_source.py       # Реалізація для веб-пошуку
├── scripts/                # Допоміжні скрипти
│   ├── db_generator.py     # Скрипт генерації даних для БД
│   └── txt_generator.py    # Скрипт генерації текстових файлів
├── static/                 # Статичні файли (CSS, JS, зображення)
│   ├── css/
│   │   └── style.css
│   └── js/
│       └── script.js
├── templates/              # HTML-шаблони (Jinja2)
│   ├── index.html          # Головна сторінка
│   └── search_result.html  # Сторінка результатів пошуку
├── texts/                  # Директорія для згенерованих текстів
├── database.py             # Налаштування БД та модель SQLAlchemy
├── main.py                 # Головний файл додатку FastAPI
└── texts.csv               # Файл мапінгу ID текстів на шляхи
```

Бекенд: FastAPI додаток (main.py):

Це центральний файл, що визначає логіку веб-сервера та API.

- Ініціалізація FastAPI: Створюється екземпляр app = FastAPI().
- Монтування статичних файлів: Директорія static/ робиться доступною за шляхом /static.

- Налаштування шаблонів: Ініціалізується Jinja2Templates для роботи з директорією templates/.
- Словники класів та назв: Створюються словники (algorithm_classes, algorithm_names, source_classes, source_names) для зручного доступу до реалізацій алгоритмів/джерел за їхніми рядковими ключами, що надходять із запитів.
- Фабрична функція get_search_source(): Повертає екземпляр потрібного класу джерела даних за ключем.
- Ендпоінти (маршрути):
  - @app.get("/"): Обробляє запити до головної сторінки, рендерить шаблон index.html, передаючи йому списки доступних алгоритмів та джерел для заповнення випадаючих списків.
  - @app.get("/item"): Обробляє запит на виконання пошуку. Приймає параметри запиту (q, algorithm, threads, source) через Query параметри URL. Використовує get_search_source() для отримання об'єкта джерела, створює екземпляр обраного алгоритму, викликає метод search() джерела даних, вимірює час виконання. Рендерить шаблон search_result.html, передаючи йому результати пошуку та метадані. Обробляє можливі винятки під час пошуку.
  - @app.get("/performance"): Обробляє асинхронний запит від JavaScript для тестування продуктивності *одного* алгоритму. Приймає аналогічні параметри. Виконує пошук, вимірює час. Повертає результат у форматі JSON, що містить назву алгоритму, час виконання, кількість потоків, кількість знайдених результатів та флаг наявності результатів. Цей ендпоінт викликається багаторазово з фронтенду для порівняння всіх алгоритмів.
- Запуск сервера: Блок if __name__ == "__main__": використовує uvicorn.run() для запуску додатку (переважно для локальної розробки).

 # main.py (ключові фрагменти)

```
# ... (імпорти) ...

app = FastAPI()
# ... (монтування статики, шаблони, словники класів) ...

def get_search_source(source_key: str):
    # ... (реалізація) ...

@app.get("/", response_class=HTMLResponse)
async def read_root(request: Request):
    # ... (рендеринг index.html) ...

@app.get("/item", response_class=HTMLResponse)
async def search(
    request: Request,
    q: str = Query(..., min_length=1),
    algorithm: str = Query("boyer_moore", enum=list(algorithm_classes.keys())),
    threads: int = Query(8, ge=1, le=32),
    source: str = Query("files", enum=list(source_classes.keys()))
):
    start_time = time.time()
    # ... (отримання джерела, алгоритму, виклик search(), обробка помилок) ...
    execution_time = time.time() - start_time
    # ... (рендеринг search_result.html) ...

@app.get("/performance")
async def performance(
    # ... (параметри запиту) ...
):
    start_time = time.time()
```

```
    # ... (отримання джерела, алгоритму, виклик search(), обробка помилок) ...
    execution_time = time.time() - start_time
    # ... (повернення JSON з результатами) ...

# ... (запуск uvicorn) ...
```

Модулі алгоритмів (algorithms/):

Кожен файл реалізує один алгоритм як клас, що успадковує BaseAlgorithm з algorithms/base.py.

- BaseAlgorithm: Визначає спільний конструктор (__init__), властивості для доступу до __texts_names та __result_list, метод text_by_number (приклад реалізації для файлів) та абстрактний метод search(self, pattern, text), який *має бути* перевизначений у нащадках.
- Конкретні алгоритми (напр., kmp.py): Містять специфічну логіку.
  - *Препроцесинг:* Часто реалізується як приватний метод (напр., _compute_lps_array у KMP, _preprocess_bad_char у BM), який викликається на початку методу search або в конструкторі, якщо препроцесинг не залежить від тексту.
  - *Метод search():* Реалізує основну логіку сканування тексту та порівняння з шаблоном відповідно до правил алгоритму. Повертає True при першому знайденому входженні (для оптимізації, оскільки нам потрібен лише факт наявності для порівняння) або після повного сканування, якщо не знайдено.

```
# algorithms/kmp.py (фрагмент)
from .base import BaseAlgorithm

class KMP(BaseAlgorithm):
    def _compute_lps_array(self, pattern):
        # ... (обчислення LPS масиву) ...
        pass
```

```
def search(self, pattern: str, text: str) -> bool:
    m = len(pattern)
    n = len(text)
    # ... (перевірки на порожні рядки) ...
    lps = self._compute_lps_array(pattern) # Препроцесинг
    i = 0  # індекс для text
    j = 0  # індекс для pattern
    while i < n:
        # ... (основна логіка порівняння та зсувів КМР) ...
        if j == m:
            return True # Знайдено
    return False # Не знайдено
```

Модуль паралелізації (algorithms/parallel/base.py):

Клас ParallelAlgorithm слугує обгорткою для паралельного виконання будь-якого базового алгоритму.

- Конструктор: Приймає екземпляр базового алгоритму (algorithm_instance).
- Метод find_in_text_parallel(): Метод, що буде виконуватись у кожному окремому процесі. Його конкретна реалізація залежить від джерела даних (має отримати текст за ID/номером та викликати search() базового алгоритму).
- Метод find_in_text(): Основний метод, що викликається ззовні.
  - Визначає кількість процесів (num_processes).
  - Завантажує список ідентифікаторів текстів (з texts.csv або БД).
  - Готує список завдань (tasks) для пулу процесів.
  - Створює multiprocessing.Pool(processes=num_processes).
  - Використовує pool.starmap або pool.map для запуску find_in_text_parallel для кожного завдання.
  - Збирає результати та форматує вихідний рядок.

```
# algorithms/parallel/base.py (концептуально)
from multiprocessing import Pool, cpu_count
import csv
# ...

class ParallelAlgorithm:
    def __init__(self, algorithm_instance):
        self.algorithm = algorithm_instance
        # ... (ініціалізація texts_names, result_list) ...

    def find_in_text_parallel(self, task_data):
        # Цей метод має бути реалізований або адаптований
        # для конкретного джерела даних у відповідному класі джерела
        # Приклад: отримати текст за ID, викликати self.algorithm.search
        raise NotImplementedError

    def find_in_text(self, search_string, thread_count=8):
        num_processes = max(1, min(thread_count, cpu_count(), 32))
        # ... (завантажити список ID/шляхів у self.texts_names) ...
        tasks = # ... (сформувати список завдань для процесів) ...

        with Pool(processes=num_processes) as pool:
            raw_results = pool.map(self.find_in_text_parallel, tasks) # або starmap

        self.result_list = [r for r in raw_results if r is not None]
        # ... (форматування результату) ...
```

Модулі джерел даних (search_sources/):

Абстрагують роботу з різними місцями зберігання текстів.

- BaseSearchSource: Визначає інтерфейс з методами get_texts() та search().
- FileSearchSource: Реалізує search(), викликаючи паралельний механізм для обробки файлів, шляхи до яких беруться з texts.csv.
- DatabaseSearchSource: Реалізує search(), отримуючи список ID документів з БД, а потім паралельно обробляючи кожен запис (кожен процес завантажує свій контент з БД).
- WebSearchSource: Реалізує search(), виконуючи запит до Bing, парсячи результати та повертаючи їх у вигляді HTML. Паралелізм тут не застосовується до базових алгоритмів пошуку рядків.

Модуль бази даних (database.py):

Використовує SQLAlchemy для визначення структури таблиці та підключення до PostgreSQL.

```
# database.py
from sqlalchemy import create_engine, Column, Integer, String, Text
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, Session

# --- Параметри підключення (краще виносити в конфігурацію/змінні оточення) ---
DB_NAME = "search_text_db"
DB_USER = "postgres"
DB_PASSWORD = "password" # У реальному проекті - безпечніше зберігання
DB_HOST = "localhost"
DB_PORT = "5432"
DATABASE_URL = f"postgresql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"
# ----------------------------------------------------------------------
```

```
engine = create_engine(DATABASE_URL)
SessionFactory = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# --- Модель даних ---
class TextDocument(Base):
    __tablename__ = "text_documents"
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, index=True)
    content = Column(Text)
    source_type = Column(String, default="database") # Може бути корисним

# --- Функція для створення сесії ---
def get_session_factory():
    return SessionFactory

# --- Функція для ініціалізації БД (створення таблиці) ---
def init_db():
    Base.metadata.create_all(bind=engine)
```

Шаблони та статика (templates/, static/):

- templates/index.html: Містить основну HTML-розмітку головної сторінки: форму з полями вводу, селекторами, кнопками "Search" та "Compare Algorithms", а також порожній контейнер (div) для відображення результатів порівняння. Використовує Jinja2 для динамічного заповнення списків алгоритмів та джерел.
- templates/search_result.html: Шаблон для відображення результатів звичайного пошуку. Виводить пошуковий запит, знайдені результати (список або HTML), час виконання та використані параметри.

- static/css/style.css: Містить CSS-правила для оформлення всіх елементів інтерфейсу, включаючи таблицю порівняння, смуги продуктивності, повідомлення тощо.
- static/js/script.js: Містить JavaScript-код для обробки подій (натискання кнопок), виконання асинхронних запитів до /performance за допомогою Fetch API, обробки JSON-відповідей, динамічної генерації HTML-таблиці порівняння та відображення toast-повідомлень.

Скрипти генерації даних (scripts/):

- txt_generator.py: Генерує вказану кількість текстових файлів у директорії texts/. Кожен файл містить випадковий текст заданої довжини та випадковий набір пошукових термінів зі списку search_terms. Також створює/перезаписує файл texts.csv з мапінгом Text {i} на згенерований шлях до файлу.
- db_generator.py: Підключається до PostgreSQL, викликає init_db() для створення таблиці (якщо її немає), очищує таблицю text_documents та заповнює її вказаною кількістю записів з випадковим текстом та пошуковими термінами. Використовує модель TextDocument та сесії SQLAlchemy.

Ця модульна структура та реалізація забезпечують виконання всіх поставлених функціональних вимог та створюють основу для гнучкої та розширюваної системи порівняння алгоритмів пошуку.

Тестування функціональності

Було проведено ручне тестування всіх основних функцій системи:

Пошук: Перевірено коректність пошуку для всіх 10 алгоритмів на всіх 3 джерелах даних з різними пошуковими запитами (існуючі та неіснуючі слова/фрази).

Вибір параметрів: Перевірено роботу випадаючих списків для джерел та алгоритмів, а також поля для введення кількості потоків.

Паралелізм: Перевірено зміну часу виконання при зміні кількості потоків (1, 4, 8).

Порівняння алгоритмів: Перевірено коректність відображення таблиці порівняння, сортування за часом, визначення найшвидшого алгоритму та відповідність кількості знайдених результатів для всіх алгоритмів.

Інтерфейс: Перевірено відгук інтерфейсу, відображення результатів та повідомлень на різних розмірах екрану.

Усі основні функції працюють коректно згідно з вимогами.

3 РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Результати створення програмного забезпечення

У результаті виконання кваліфікаційної роботи було розроблено програмне забезпечення «SearchMaster» – веб-додаток для дослідження та порівняння алгоритмів повнотекстового пошуку. Для досягнення поставленої мети було виконано наступні задачі:

- Аналіз та проєктування: В межах Розділу 1 було проаналізовано предметну область, існуючі рішення та сформульовано вимоги до системи. В межах Розділу 2 було розроблено архітектуру (діаграми компонентів та розгортання), ескізний проєкт (діаграми варіантів використання, діяльності, прототипи інтерфейсу) та технічний проєкт (діаграми класів та «сутність-зв'язок»).
- Реалізація: В межах підрозділу 2.3.3 було реалізовано робочий проєкт, що включає 10 алгоритмів пошуку, модулі для роботи з трьома типами джерел даних (файли, БД, веб), механізм паралельної обробки та інтерактивний веб-інтерфейс. Загальний розмір програмного коду проєкту становить ~3000 рядків коду.
- Документація: Було розроблено повний комплект супровідної документації, що включає:
    - Додаток А: Технічне завдання на розробку ПЗ.
    - Додаток Б: Опис програмного забезпечення.
    - Додаток В: Програма та методика випробувань.
    - Додаток Г: Керівництво користувача (оператора).
    - Додаток Д: Лістинги ключових програмних модулів.

3.2 Аналіз функціонування програмного забезпечення

Для демонстрації функціональності та аналізу продуктивності було використано функцію «Compare Algorithms». Було проведено серію експериментів для пошукового запиту «dsds» на різних джерелах даних та з різною кількістю потоків.

А. Джерело: Текстові файли

Було згенеровано 500 текстових файлів обсягом близько 20000 слів кожен.

Результати тестування для різної кількості потоків (8, 4 та 1) представлені графічно на рисунках 3.1-3.3 та узагальнені в таблицях 3.1-3.3.

Таблиця 3.1 – Результати порівняння алгоритмів (Джерело: Файли, Запит: «dsds», Потоки: 8)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Horspool	0.446058	63	100.00%
Sunday	0.447483	63	99.68%
Boyer-Moore-Horspool	0.460363	63	96.89%
Simon	0.511038	63	87.28%
Two-Way	0.585813	63	76.14%
Boyer-Moore	0.680684	63	65.53%
Brute Force	0.910100	63	49.01%
Knuth-Morris-Pratt	1.012676	63	44.05%
Bitap	1.400858	63	31.84%
Rabin-Karp	1.787325	63	24.96%

(Дані взято з Рис. 3.1)

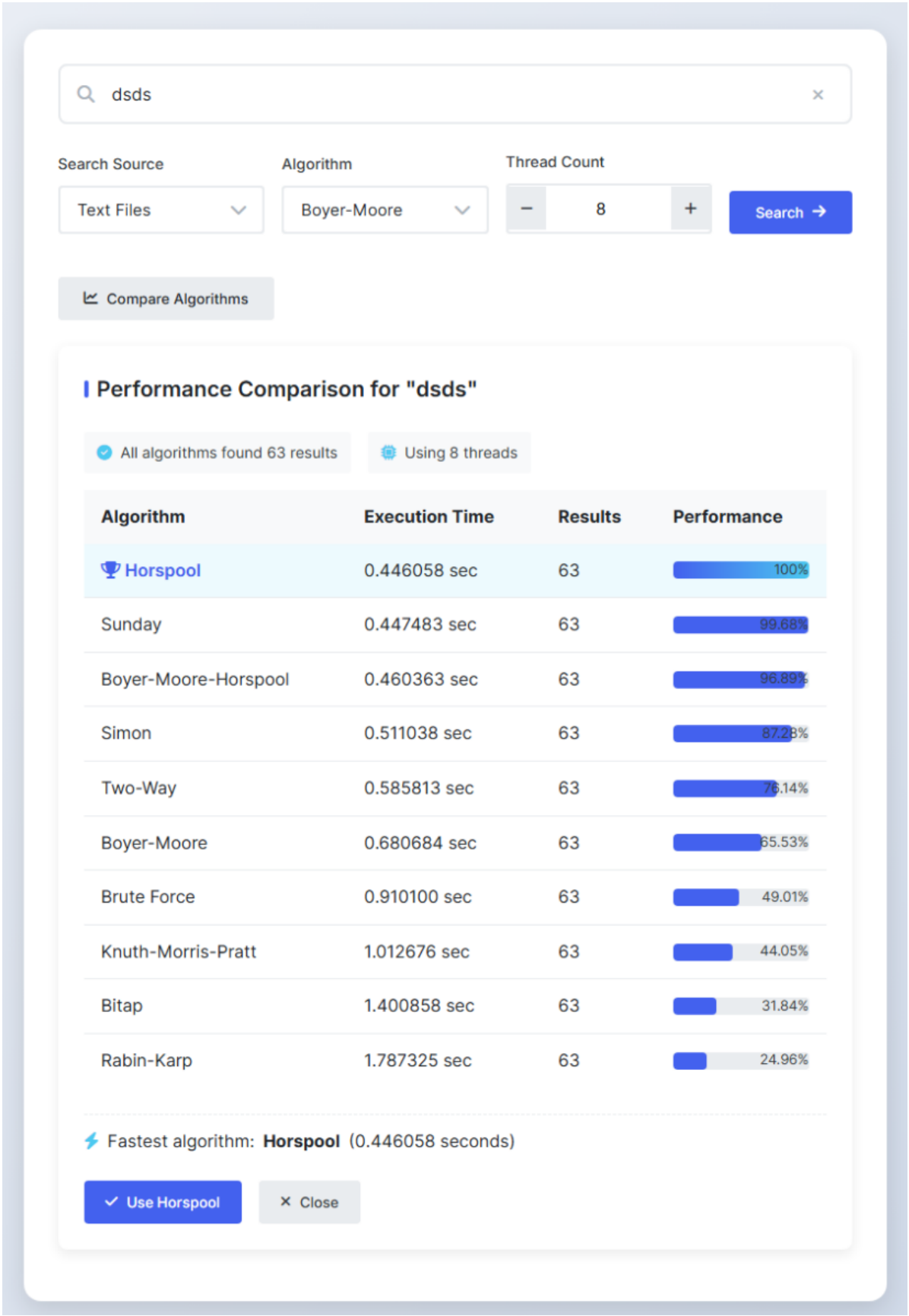


Рисунок 3.1 – Порівняння продуктивності (Файли, 8 потоків)

Таблиця 3.2 – Результати порівняння алгоритмів (Джерело: Файли, Запит: «dsds», Потоки: 4)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Sunday	0.607253	63	100.00%
Horspool	0.627400	63	96.79%
Boyer-Moore-Horspool	0.669226	63	90.74%
Simon	0.724777	63	83.78%
Two-Way	0.820366	63	74.02%
Boyer-Moore	0.908859	63	66.81%
Brute Force	1.355637	63	44.79%
Knuth-Morris-Pratt	1.505674	63	40.33%
Bitap	2.098248	63	28.94%
Rabin-Karp	2.621925	63	23.16%

(Дані взято з Рис. 3.2)

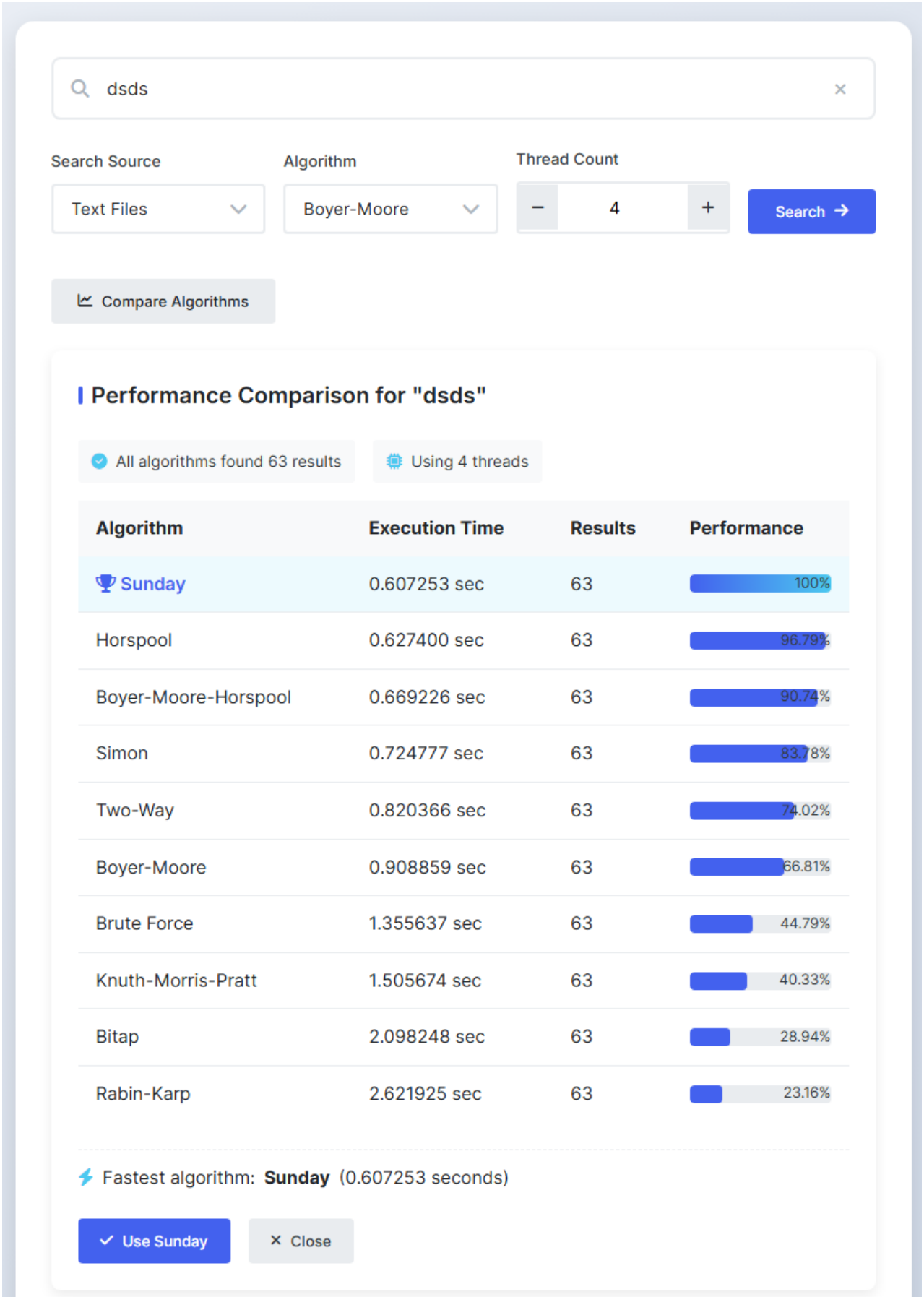


Рисунок 3.2 – Порівняння продуктивності (Файли, 4 потоки)

Таблиця 3.3 – Результати порівняння алгоритмів (Джерело: Файли, Запит: «dsds», Потоки: 1)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Sunday	1.976464	63	100.00%
Horspool	2.068602	63	95.55%
Boyer-Moore-Horspool	2.191977	63	90.17%
Simon	2.387648	63	82.78%
Two-Way	2.704947	63	73.07%
Boyer-Moore	2.818998	63	70.11%
Brute Force	4.641138	63	42.59%
Knuth-Morris-Pratt	5.207134	63	37.96%
Bitap	7.361513	63	26.85%
Rabin-Karp	9.710563	63	20.35%

(Дані взято з Рис. 3.3)

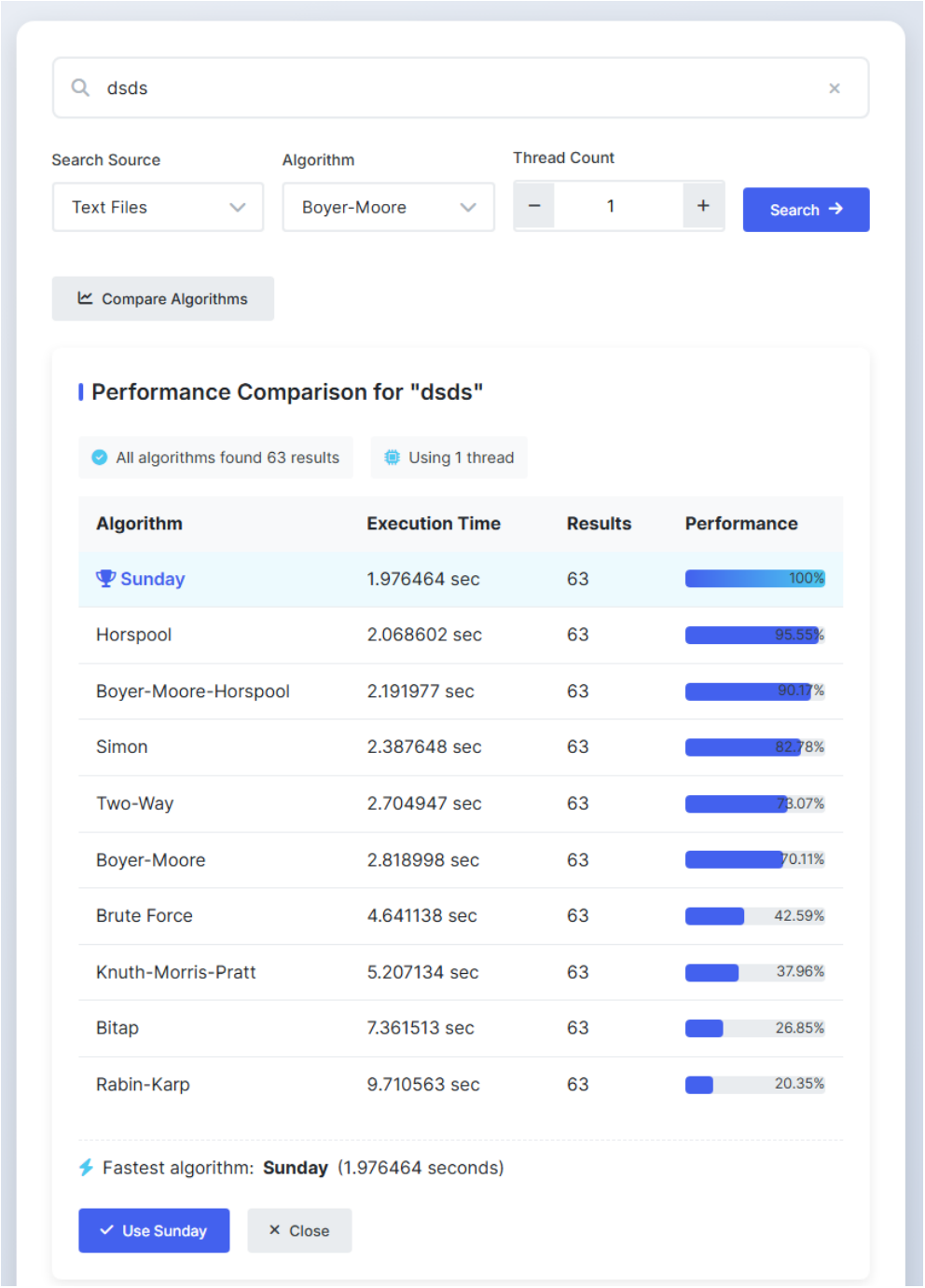


Рисунок 3.3 – Порівняння продуктивності (Файли, 1 потік)

Аналіз (Файли):

Найшвидшими алгоритмами для пошуку у файлах стабільно є Horspool та Sunday.

Зменшення кількості потоків з 8 до 1 призводить до значного (приблизно в 4-5 разів) збільшення часу виконання для всіх алгоритмів. Це наочно демонструє

високу ефективність багатопоточної обробки даних при роботі з файловою системою.

Алгоритми KMP, Bitap та Rabin-Karp показують найгіршу продуктивність у цьому сценарії.

Б. Джерело: База даних PostgreSQL

Було згенеровано 100 записів у БД, кожен з текстом обсягом близько 20000 слів.

Результати тестування для різної кількості потоків (8, 4 та 1) представлені графічно на рисунках 3.4-3.6 та узагальнені в таблицях 3.4-3.6.

Таблиця 3.4 – Результати порівняння алгоритмів (Джерело: БД, Запит: «dsds», Потоки: 8)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Boyer-Moore-Horspool	1.100839	16	100.00%
Horspool	1.102684	16	99.83%
Two-Way	1.115533	16	98.68%
Sunday	1.129843	16	97.43%
Simon	1.136212	16	96.89%
Boyer-Moore	1.195110	16	92.11%
Brute Force	1.196418	16	92.01%
Knuth-Morris-Pratt	1.220725	16	90.18%
Bitap	1.302689	16	84.51%
Rabin-Karp	1.393346	16	79.01%

(Дані взято з Рис. 3.4)

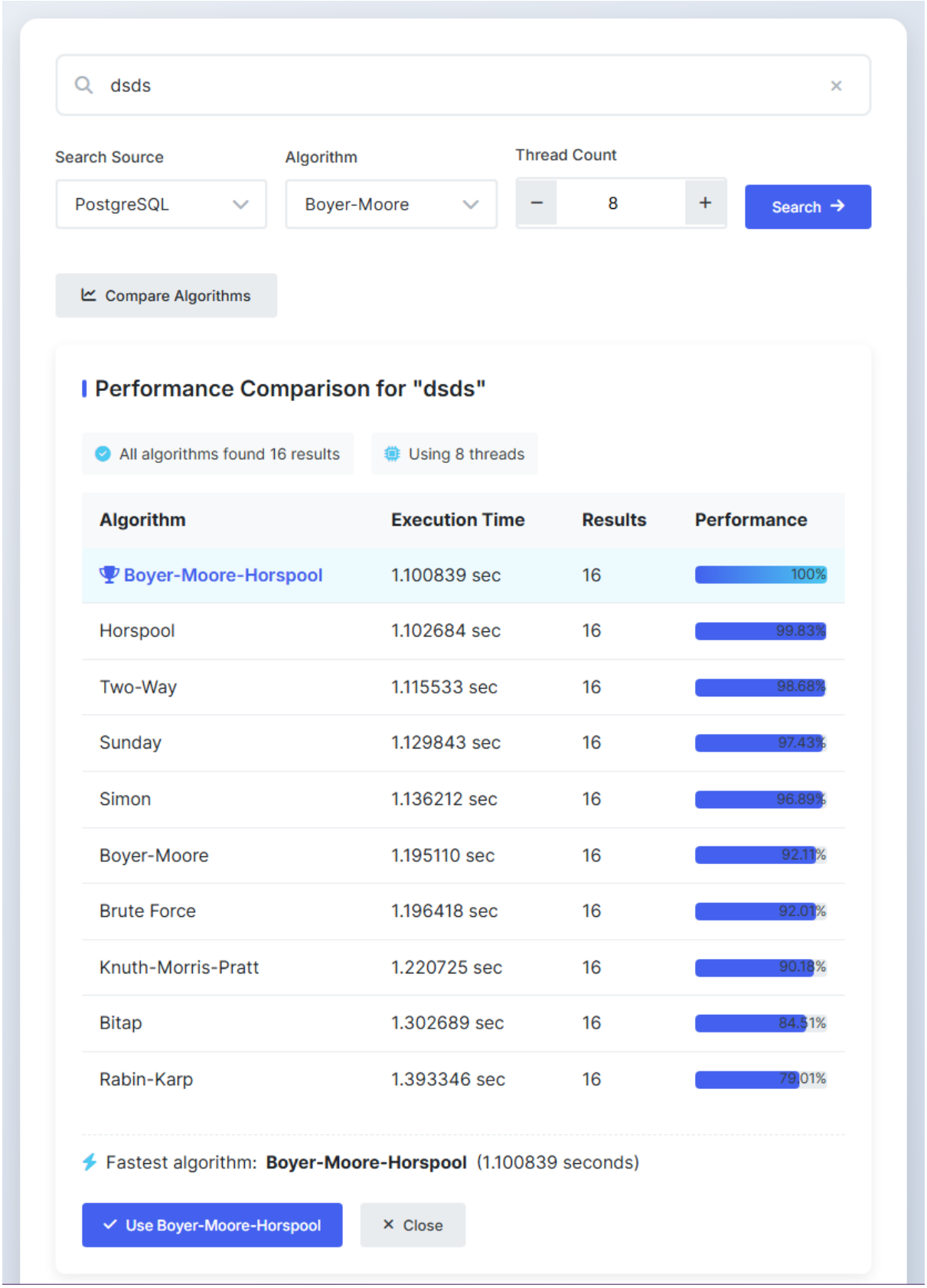


Рисунок 3.4 – Порівняння продуктивності (БД, 8 потоків)

Таблиця 3.5 – Результати порівняння алгоритмів (Джерело: БД, Запит: «dsds», Потоки: 4)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Sunday	0.942664	16	100.00%
Boyer-Moore-Horspool	0.959379	16	98.26%
Horspool	0.962634	16	97.93%
Simon	0.984827	16	95.72%
Boyer-Moore	1.011353	16	93.21%
Brute Force	1.089231	16	86.54%
Two-Way	1.120433	16	84.13%
Knuth-Morris-Pratt	1.148638	16	82.07%
Bitap	1.298684	16	72.59%
Rabin-Karp	1.541340	16	61.16%

(Дані взято з Рис. 3.5)

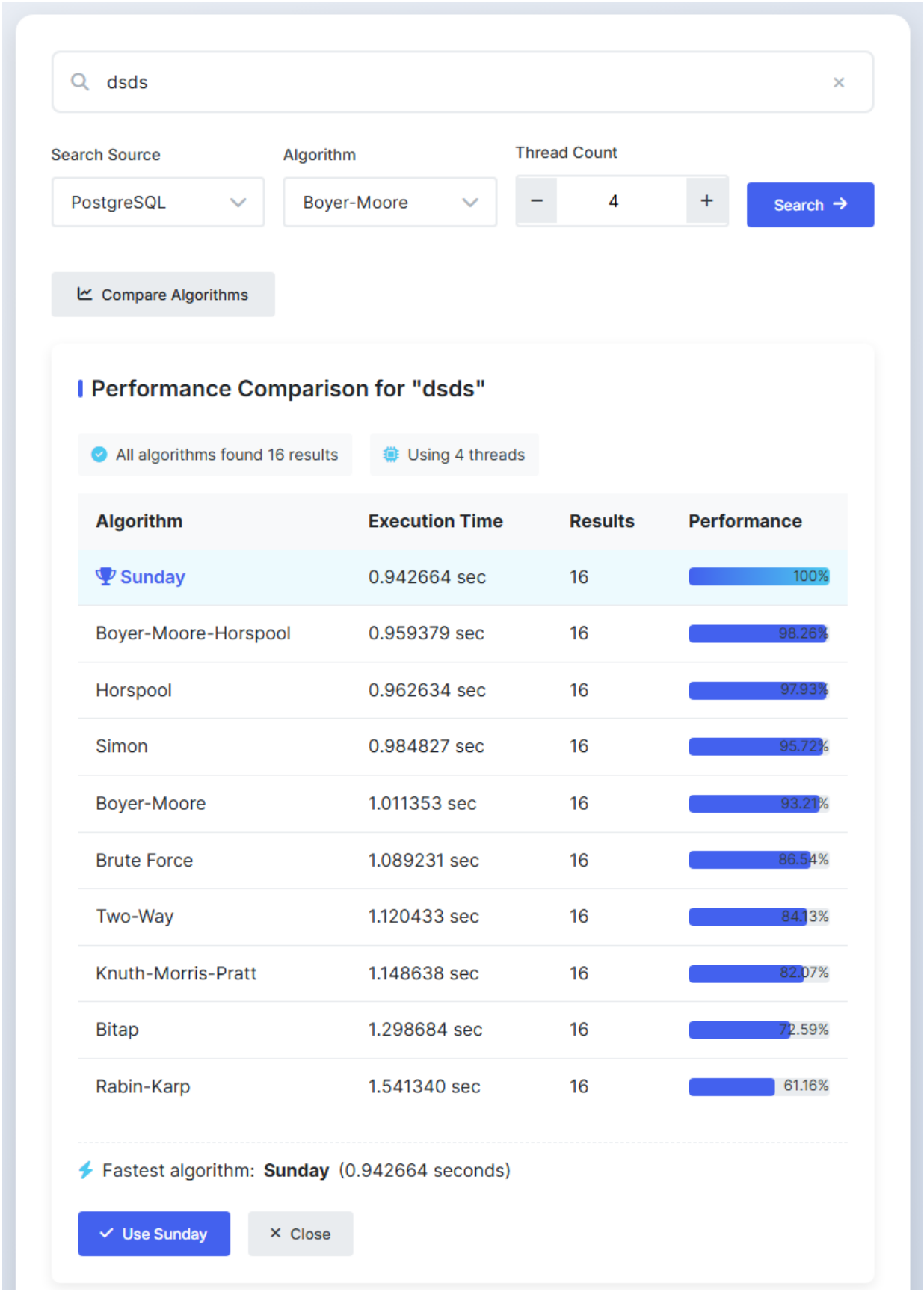


Рисунок 3.5 – Порівняння продуктивності (БД, 4 потоки)

Таблиця 3.6 – Результати порівняння алгоритмів (Джерело: БД, Запит: «dsds», Потоки: 1)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Sunday	1.095444	16	100.00%
Boyer-Moore-Horspool	1.120064	16	97.80%
Simon	1.169553	16	93.66%
Boyer-Moore	1.250294	16	87.61%
Two-Way	1.262139	16	86.79%
Horspool	1.324280	16	82.72%
Brute Force	1.575930	16	69.51%
Knuth-Morris-Pratt	1.716092	16	63.83%
Bitap	2.121756	16	51.63%
Rabin-Karp	2.622796	16	41.77%

(Дані взято з Рис. 3.6)

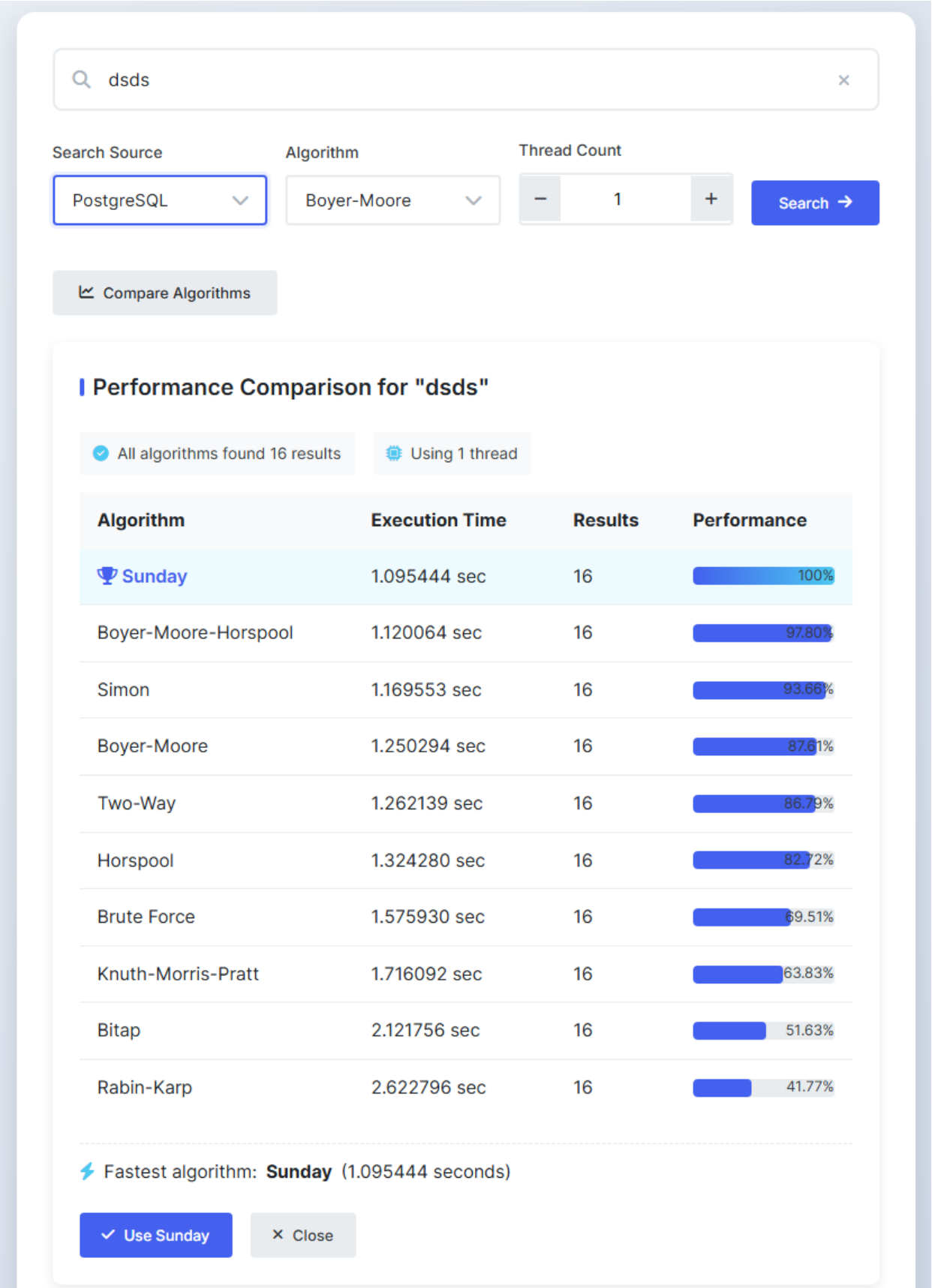


Рисунок 3.6 – Порівняння продуктивності (БД, 1 потік)

Аналіз (БД):

При пошуку в БД найшвидшими виявляються алгоритми Sunday та Boyer-Moore-Horspool.

Вплив багатопоточної архітектури на пошук у базі даних менш виражений, ніж при роботі з файлами (зменшення часу приблизно в 1.1-1.5 рази при переході від 1 до 8 потоків). Це може бути пов'язано з накладними витратами на взаємодію з БД та обмеженнями самої БД.

Загальний час виконання пошуку в БД вищий, ніж у файлах, що очікувано через додаткові операції з базою даних.

Bitap та Rabin-Karp знову показують найнижчу продуктивність.

В. Джерело: Веб-пошук

Веб-пошук використовує зовнішню систему (Bing) для отримання URL, тому порівняння базових алгоритмів пошуку рядків у цьому режимі стосується часу обробки запиту до Bing та парсингу результатів, а не сканування контенту сторінок (хоча це можна додати). Надані зображення показують результати порівняння, ймовірно, часу, потрібного для отримання та базової обробки результатів від Bing, де сам пошук рядка виконується вже на стороні Bing. Однак, якщо припустити, що порівнюється швидкість пошуку *після* завантаження контенту 5 знайдених сторінок, аналіз буде іншим. Оскільки точна методологія веб-тестування не описана, наведемо дані з зображень.

Результати порівняння продуктивності для різної кількості потоків (8, 4 та 1) представлені графічно на рисунках 3.8-3.10 та узагальнені в таблицях 3.7-3.9.

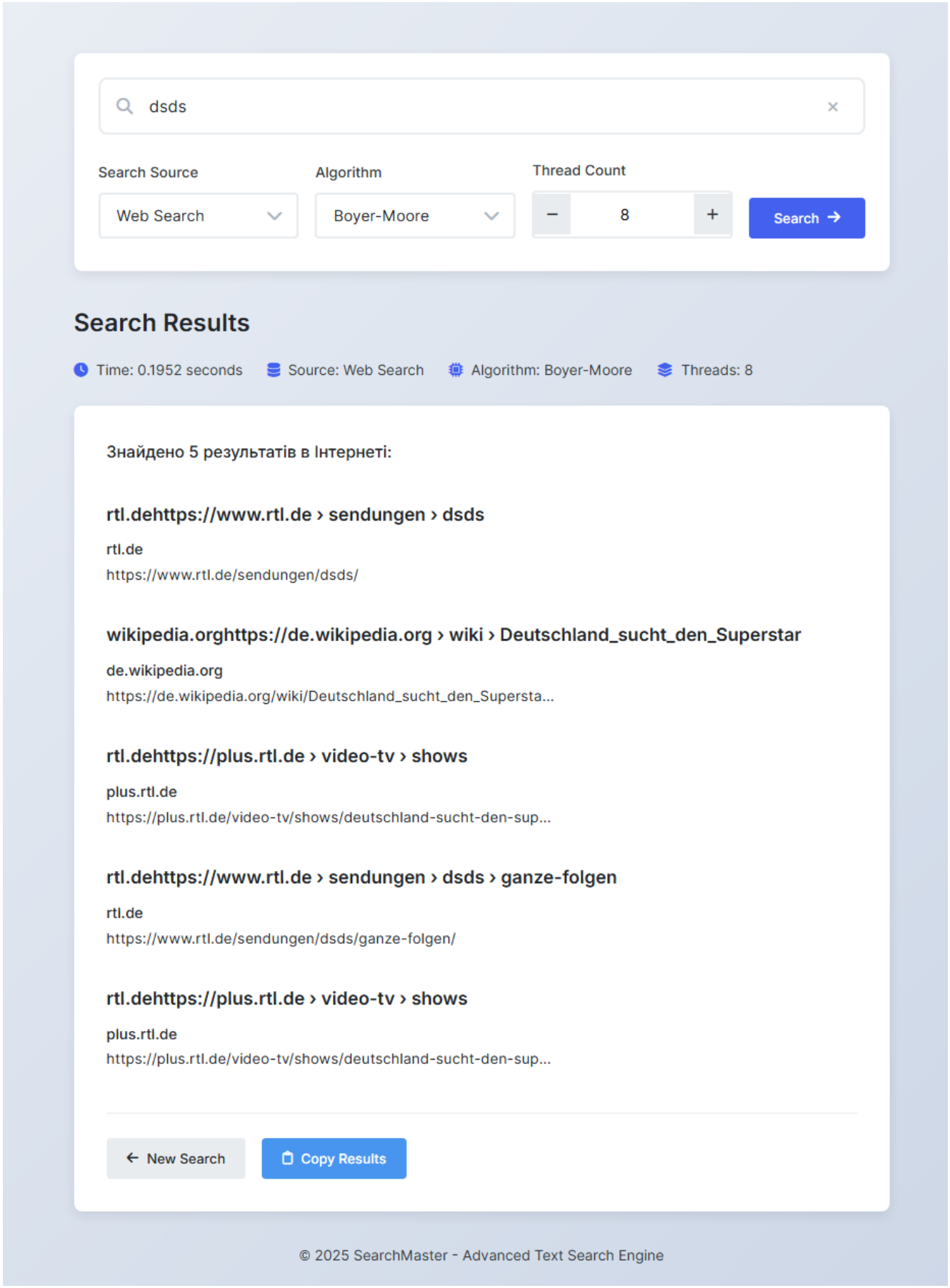


Рисунок 3.7 – Результати веб-пошуку

Таблиця 3.7 – Результати порівняння алгоритмів (Джерело: Веб, Запит: «dsds», Потоки: 8)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Brute Force	0.082677	5	100.00%
Sunday	0.082801	5	99.85%
Knuth-Morris-Pratt	0.128951	5	64.12%
Two-Way	0.132867	5	62.23%
Simon	0.141127	5	58.58%
Boyer-Moore	0.145174	5	56.95%
Boyer-Moore-Horspool	0.151222	5	54.67%
Horspool	0.155341	5	53.22%
Bitap	0.168617	5	49.03%
Rabin-Karp	0.171714	5	48.15%

(Дані взято з Рис. 3.8)

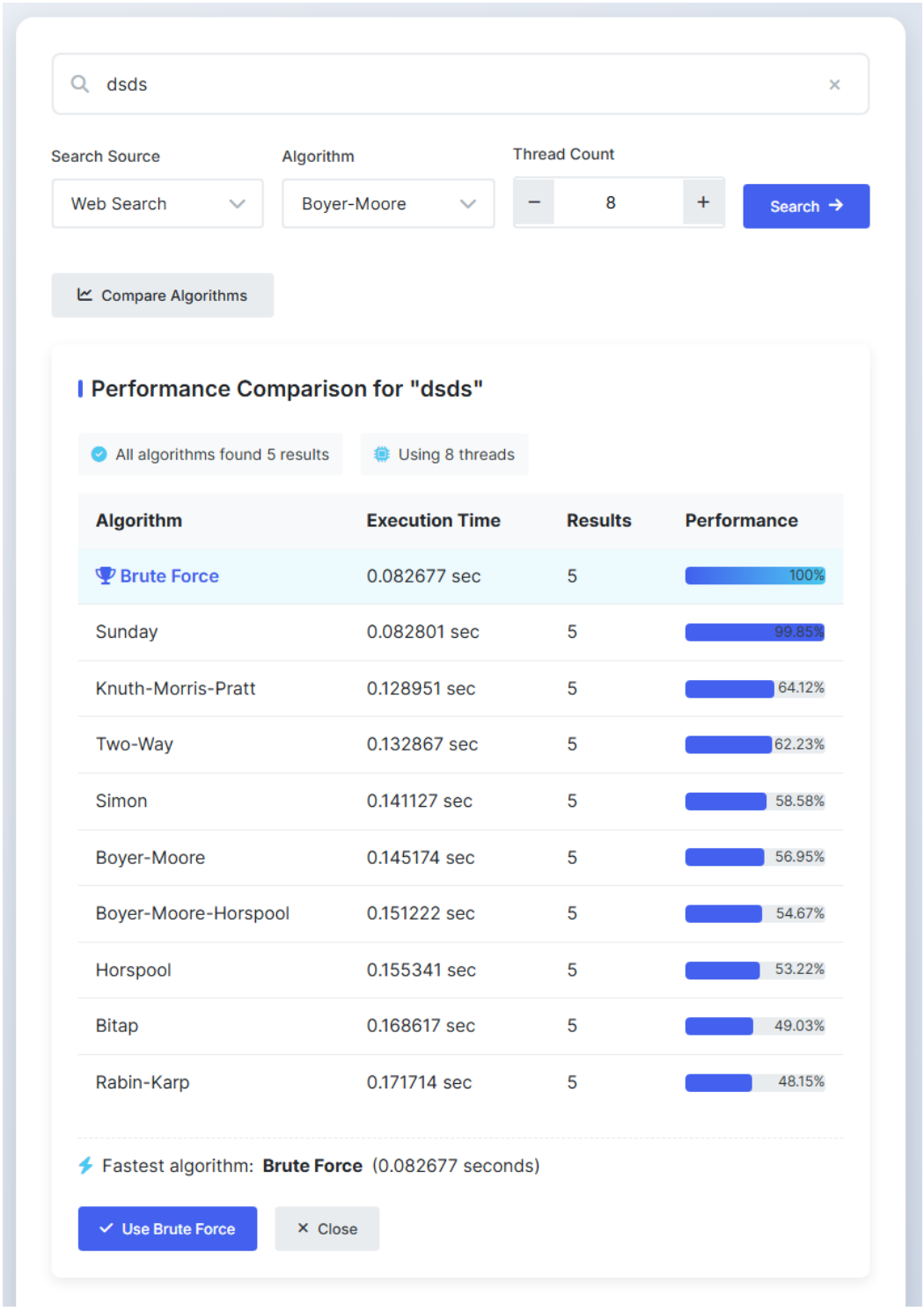


Рисунок 3.8 – Порівняння продуктивності (Веб, 8 потоків)

Таблиця 3.8 – Результати порівняння алгоритмів (Джерело: Веб, Запит: «dsds», Потоки: 4)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Boyer-Moore	0.084262	5	100.00%
Boyer-Moore-Horspool	0.086873	5	97.00%
Rabin-Karp	0.140078	5	60.15%
Two-Way	0.140082	5	60.15%
Knuth-Morris-Pratt	0.151529	5	55.61%
Bitap	0.156515	5	53.84%
Brute Force	0.158313	5	53.22%
Horspool	0.175044	5	48.14%
Sunday	0.250382	5	33.65%
Simon	0.260991	5	32.29%

(Дані взято з Рис. 3.9)

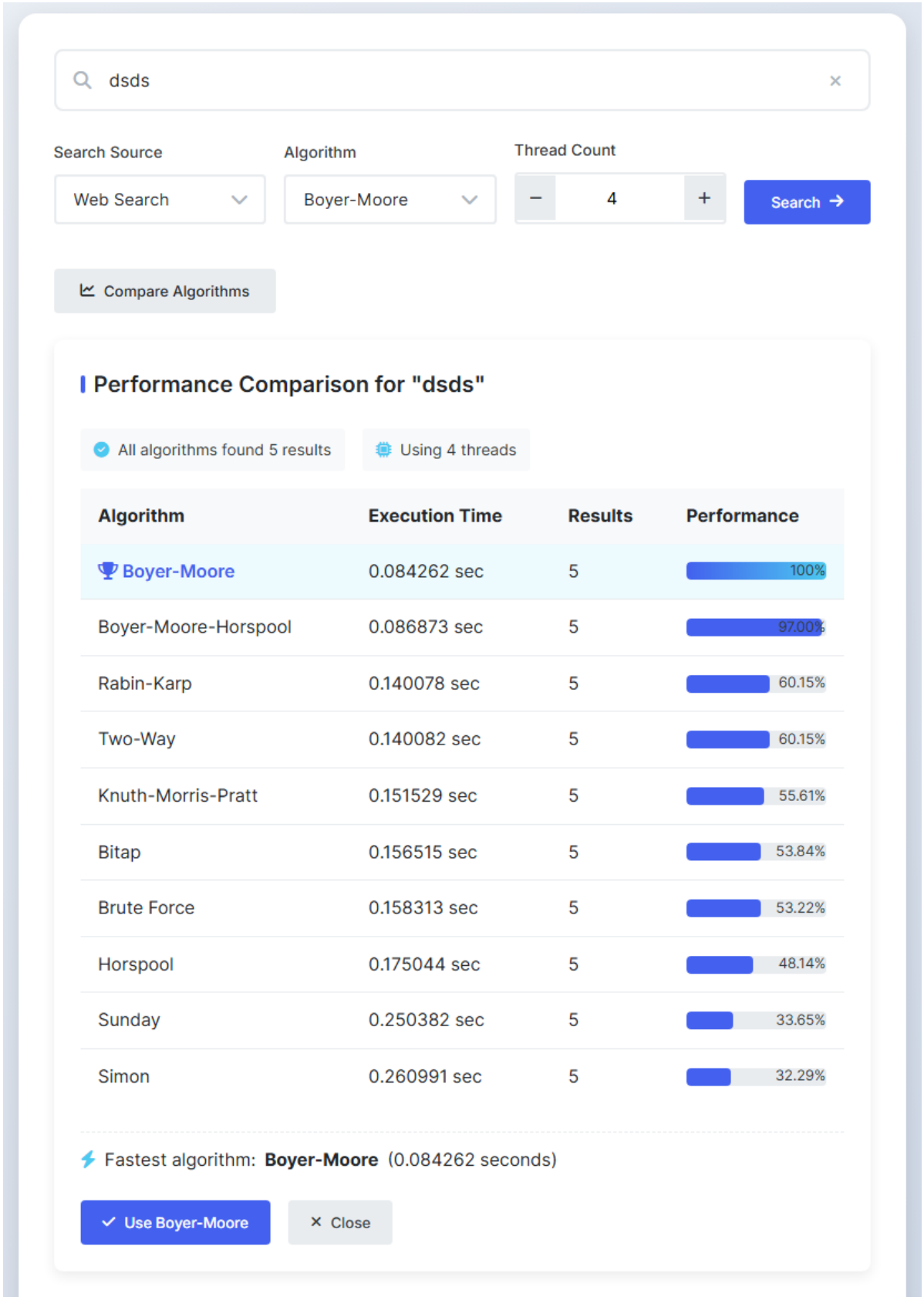


Рисунок 3.9 – Порівняння продуктивності (Веб, 4 потоки)

Таблиця 3.9 – Результати порівняння алгоритмів (Джерело: Веб, Запит: «dsds», Потоки: 1)

Алгоритм	Час виконання (сек)	Знайдено результатів	Продуктивність (відносна)
Two-Way	0.064555	5	100.00%
Sunday	0.081785	5	78.93%
Knuth-Morris-Pratt	0.089577	5	72.07%
Simon	0.091241	5	70.75%
Brute Force	0.130047	5	49.64%
Boyer-Moore-Horspool	0.133567	5	48.33%
Horspool	0.151121	5	42.72%
Rabin-Karp	0.152050	5	42.46%
Boyer-Moore	0.153521	5	42.05%
Bitap	0.225850	5	28.58%

(Дані взято з Рис. 3.10)

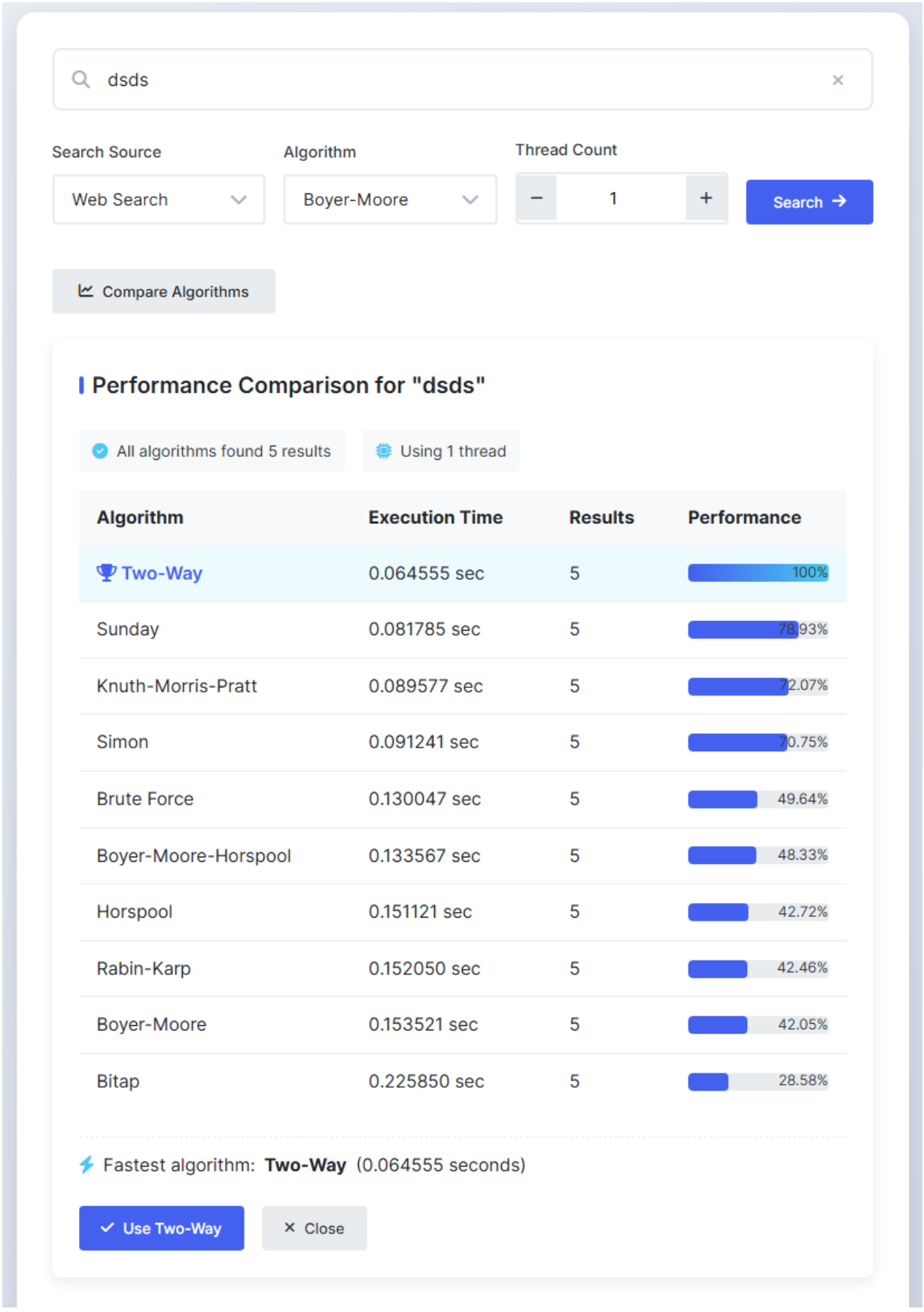


Рисунок 3.10 – Порівняння продуктивності (Веб, 1 потік)

Аналіз (Веб):

Результати для веб-пошуку показують значно менший час виконання, оскільки основна робота (пошук та індексація) виконується зовнішньою системою Bing. Виміряний час, ймовірно, відображає час запиту до Bing API, отримання та парсингу результатів, а також, можливо, дуже швидкий пошук у невеликому обсязі отриманих даних (заголовки, сніпети).

Лідерство різних алгоритмів (Brute Force, Boyer-Moore, Two-Way) при різній кількості потоків може бути артефактом вимірювання дуже коротких проміжків часу або вказувати на те, що для дуже малих текстів (сніпетів) різниця між алгоритмами мінімальна, і накладні витрати на ініціалізацію складніших алгоритмів стають помітними.

Вплив паралелізму тут не показовий для самих алгоритмів пошуку рядків, а скоріше для паралельної обробки результатів від Bing (якщо вона реалізована).

Продуктивність алгоритмів суттєво залежить від джерела даних та використання багатопоточної обробки. Для локальних файлів та БД найефективнішими виявились варіації алгоритмів Horspool, Sunday та Boyer-Moore. Реалізовані багатопоточні алгоритми дають значний приріст швидкості, особливо при роботі з файловою системою. Система «SearchMaster» дозволяє наочно демонструвати ці відмінності та допомагає обрати оптимальний підхід для конкретної задачі.

4 ОХОРОНА ПРАЦІ

4.1 Аналіз шкідливих та небезпечних факторів при роботі з ПК

Розробка та використання програмного забезпечення «SearchMaster», як і будь-яка робота, пов'язана з тривалим використанням персонального комп'ютера (ПК) та відеодисплейних терміналів (ВДТ), супроводжується низкою потенційно шкідливих та небезпечних виробничих факторів (табл 4.1). Згідно з Законом України «Про охорону праці» [16] та відповідними санітарними нормами, необхідно враховувати ці фактори для забезпечення безпечних умов праці.

Основні шкідливі та небезпечні фактори при роботі з ПК:

- Фізичні фактори:
  - Електромагнітне випромінювання: Неіонізуюче випромінювання від монітора, системного блоку та іншого обладнання. Сучасні монітори мають значно нижчі рівні випромінювання, але фактор залишається.
  - Недостатня або надмірна освітленість: Неправильно організоване освітлення робочого місця (відблиски на екрані, занадто яскраве або тьмяне світло) може призвести до зорової втоми.
  - Підвищений рівень шуму: Шум від роботи системного блоку, принтера чи іншого офісного обладнання.
  - Неоптимальні параметри мікроклімату: Невідповідна температура, вологість та швидкість руху повітря у робочому приміщенні.
  - Небезпека ураження електричним струмом: Можлива при несправності обладнання або порушенні правил експлуатації.
- Психофізіологічні фактори:
  - Зорове напруження: Тривала робота з монітором, необхідність розрізняти дрібні символи, фіксація погляду на екрані.

- Розумове напруження: Висока концентрація уваги, обробка великих обсягів інформації, монотонність роботи (при тестуванні, аналізі даних).
- Емоційне напруження: Відповідальність за результат, можливі збої в роботі ПЗ, необхідність дотримання термінів.
- Статичне навантаження та незручна робоча поза: Тривале сидіння в одній позі може призвести до проблем з опорно-руховим апаратом (остеохондроз, сколіоз).
- Монотонність праці: Виконання однотипних операцій під час розробки, тестування чи аналізу результатів.

Таблиця 4.1 – Шкідливі та небезпечні виробничі фактори при роботі з ПК

Найменування фактору	Можливі джерела його виникнення	Характер дії
1	2	3
Небезпека ураження електричним струмом	Мережа живлення, несправне обладнання	Небезпечний
Електромагнітне випромінювання	Монітор, системний блок, периферійні пристрої	Шкідливий
Несприятлива освітленість	Недостатнє/надмірне штучне або природне освітлення, відблиски	Шкідливий
Підвищений рівень шуму	Системний блок, принтер, вентиляція	Шкідливий
Неоптимальний мікроклімат	Система опалення/кондиціонування, недостатня вентиляція	Шкідливий
Зорове напруження	Тривала робота з монітором, дрібний текст/графіка	Шкідливий

Кінець таблиці 4.1

1	2	3
Розумове/емоційне напруження	Складність задач, відповідальність, дедлайни, монотонність	Шкідливий
Статичне навантаження/незручна поза	Тривале сидіння, неправильно організоване робоче місце	Шкідливий

Тривала та інтенсивна робота за комп'ютером без дотримання норм охорони праці може стати джерелом професійних захворювань, таких як синдром комп'ютерного зору, захворювання опорно-рухового апарату, стрес та перевтома.

4.2 Заходи щодо забезпечення безпечних умов праці

Для мінімізації впливу шкідливих та небезпечних факторів при розробці та експлуатації ПЗ «SearchMaster» необхідно дотримуватись наступних заходів:

- Вимоги до робочого місця:
  - Площа та об'єм: Робоче місце повинно мати достатню площу (не менше 6 м²) та об'єм (не менше 20 м³).
  - Розташування обладнання: Монітор слід розташовувати на відстані 60-70 см від очей, трохи нижче рівня очей. Клавіатура та миша повинні знаходитись на зручній висоті, що дозволяє тримати передпліччя паралельно підлозі.
  - Крісло: Використовувати ергономічне крісло з можливістю регулювання висоти сидіння, нахилу спинки та висоти підлокітників.
  - Освітлення: Забезпечити раціональне освітлення робочого місця, уникати відблисків на екрані. Рекомендовано комбіноване освітлення (загальне та місцеве).
- Вимоги до обладнання:

- Використовувати сучасні монітори з низьким рівнем випромінювання та високою чіткістю зображення.
- Регулярно проводити технічне обслуговування обладнання, перевіряти справність кабелів та заземлення.

– Режим праці та відпочинку:
- Дотримуватись регламентованих перерв протягом робочого дня (наприклад, 10-15 хвилин кожні 1-1.5 години роботи).
- Під час перерв виконувати вправи для очей та фізичні вправи для зняття статичного напруження.

– Мікроклімат:
- Підтримувати оптимальні параметри температури (22-24°C в холодний період, 23-25°C в теплий), вологості (40-60%) та швидкості руху повітря (до 0.1 м/с).
- Регулярно провітрювати приміщення.

– Електробезпека:
- Використовувати тільки справне обладнання.
- Не торкатися оголених проводів та роз'ємів.
- Використовувати джерела безперебійного живлення для запобігання втраті даних та пошкодженню обладнання при перепадах напруги.

Дотримання цих заходів дозволить створити безпечні та комфортні умови праці, знизити ризик виникнення професійних захворювань та підвищити продуктивність роботи при розробці та використанні ПЗ «SearchMaster».

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було успішно досягнуто поставленої мети – розроблено програмне забезпечення «SearchMaster» для повнотекстового пошуку, що надає функціональність для порівняльного аналізу ефективності різних багатопоточних алгоритмів.

У процесі роботи було вирішено наступні завдання:

- Проаналізовано предметну область, зокрема теоретичні основи повнотекстового пошуку та існуючі алгоритми, такі як Бойєр-Мур, Кнут-Морріс-Пратт, Рабін-Карп та інші. Аналіз існуючих аналогічних рішень (пошукових систем та бібліотек) підтвердив необхідність створення гнучкого інструменту для дослідницьких та освітніх цілей.
- Сформульовано детальні функціональні та нефункціональні вимоги до системи «SearchMaster» та розроблено технічне завдання, яке стало основою для проєктування.
- Спроєктовано архітектуру програмного забезпечення на основі сучасного технологічного стеку (Python, FastAPI, SQLAlchemy, PostgreSQL, Jinja2, JavaScript), що забезпечує модульність, розширюваність та високу продуктивність.
- Реалізовано програмний продукт «SearchMaster», який включає 10 алгоритмів пошуку, підтримує роботу з трьома типами джерел даних (локальні файли, база даних, веб-пошук) та дозволяє налаштовувати кількість потоків для паралельної обробки.
- Проведено комплексне тестування розробленого програмного забезпечення. Функціональне тестування підтвердило коректність роботи всіх компонентів системи. Тестування продуктивності виявило ключові закономірності:
    - Для пошуку у великих текстових файлах та базах даних найшвидшими стабільно виявляються варіації алгоритму Бойєра-Мура (Horspool, Sunday).

- Використання паралельної обробки даних дає значний приріст продуктивності, скорочуючи час виконання пошуку в 4-5 разів при переході від одного до восьми потоків, особливо при роботі з файловою системою.
- Алгоритми КМП, Bitap та Рабін-Карп показали нижчу продуктивність у проведених тестах, що підкреслює важливість вибору алгоритму залежно від конкретної задачі та характеристик даних.

– Розроблено повний комплект програмної документації, включаючи технічне завдання, опис програми, програму та методику випробувань, а також керівництво користувача.

– Проаналізовано питання охорони праці, пов’язані з розробкою та використанням програмного забезпечення, та визначено заходи для створення безпечних умов праці.

Таким чином, кваліфікаційна робота охопила повний цикл розробки програмного забезпечення: від аналізу вимог та предметної області до реалізації, тестування та документування. Створена система «SearchMaster» є завершеним програмним продуктом, що може бути використаний в освітніх та дослідницьких цілях для демонстрації, вивчення та порівняння алгоритмів повнотекстового пошуку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dukhnovska K.K., Myshko I.L. Analysis and Comparison of Full-Text Search Algorithms C. 45-52 // Системи управління, навігації та зв'язку. 2024. URL: http://usim.org.ua/?page_id=17118&lang=uk (дата звернення: 18.02.2025).

2. Sulaiman H. E. Use the Brute_Force Pattern Matching Algorithm for Misuse Intrusion Detection System // AL-Rafidain Journal of Computer Sciences and Mathematics. 2019. Vol. 13, No. 1. P. 68-85. URL: https://doaj.org/article/6c40c705b1fd4263955b4fc066b9155d (дата звернення: 18.02.2025).

3. Abikoye O. C., Abubakar A., Dokoro A. H., Akande O. N., Kayode A. A. A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm // EURASIP Journal on Information Security. 2020. P. 1-14. URL: https://doaj.org/article/27372f01abe34aea9ad7f516f9739556 (дата звернення: 03.03.2025).

4. Shapira D., Daptardar A. Adapting the Knuth–Morris–Pratt algorithm for pattern matching in Huffman encoded texts // Information processing & management. 2006. Vol. 42, No. 2. P. 429-439. URL: https://www.sciencedirect.com/science/article/abs/pii/S0306457305000191 (дата звернення: 19.03.2025).

5. Rytter W. On maximal suffixes and constant-space linear-time versions of KMP algorithm // Theoretical computer science. 2003. Vol. 299, No. 1-3. P. 763-774. URL: https://www.sciencedirect.com/science/article/pii/S030439750200590X (дата звернення: 28.02.2025).

6. Zhang Z. Review on String-Matching Algorithm // SHS Web of Conferences. 2022. Vol. 144. P. 03018. URL: https://doaj.org/article/c78bdfe402274e139c26295b12694388 (дата звернення: 10.02.2025).

7. Hendrian D., Ueki Y., Narisawa K., Yoshinaka R., Shinohara A. Permuted pattern matching algorithms on multi-track strings // Algorithms. 2019. Vol. 12, No. 4. P. 73. URL: https://doaj.org/article/4a62dbb11cc14fd0ba9e2be7fa549cc2 (дата звернення: 15.03.2025).

8. Hendrian D., Ueki Y., Narisawa K., Yoshinaka R., Shinohara A. Permuted pattern matching algorithms on multi-track strings // Algorithms. 2019. Vol. 12, No. 4. P. 73. URL: https://www.mdpi.com/1999-4893/12/4/73 (дата звернення: 08.02.2025).

9. Python Multiprocessing | Official Documentation. URL: https://docs.python.org/3/library/multiprocessing.html (дата звернення: 25.02.2025).

10. Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press. URL: https://nlp.stanford.edu/IR-book/information-retrieval-book.html (дата звернення: 14.03.2025).

11. Apache Lucene - Core | Official Website. URL: https://lucene.apache.org/core/ (дата звернення: 15.03.2025).

12. Kibana: Explore, Visualize, Discover Data | Elastic. URL: https://www.elastic.co/kibana/ (дата звернення: 15.03.2025).

13. Python Programming Language | Official Website. URL: https://www.python.org/ (дата звернення: 11.03.2025).

14. FastAPI | Official Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 05.03.2025).

15. PostgreSQL – The world’s most advanced open source database | Official Website. URL: https://www.postgresql.org/ (дата звернення: 20.02.2025).

16. Закон України «Про охорону праці» від 21.11.2002 № 229-IV. URL: https://zakon.rada.gov.ua/laws/main/2694-12#Text (дата звернення: 17.03.2025).

ДОДАТОК А – Технічне завдання на розробку ПЗ «SearchMaster»

Вступ

Повна назва розроблюваного проекту: Розробка програмного забезпечення повнотекстового пошуку з багатопоточними алгоритмами

Галузь застосування: Дослідження алгоритмів повнотекстового пошуку, порівняльний аналіз їх продуктивності, освітні цілі в галузі інженерії програмного забезпечення та алгоритмів, допоміжний інструмент при розробці пошукових підсистем.

1 Підстави для розробки

Підставою для розробки програмного забезпечення (ПЗ) «SearchMaster» є наказ на затвердження тем кваліфікаційних робіт освітнього ступеня «Бакалавр».

2 Призначення розробки

2.1 Функціональне призначення

Функціональним призначенням програмного забезпечення є надання користувачам інструментарію для дослідження та порівняння алгоритмів повнотекстового пошуку, а саме – забезпечення можливості оперативного виконання пошуку заданого рядка (шаблону) в текстових даних, вибору джерела даних для пошуку (локальні текстові файли, база даних PostgreSQL, результати веб-пошуку), вибору одного з 10 реалізованих алгоритмів повнотекстового пошуку (Boyer-Moore, Rabin-Karp, KMP, Bitap, Sunday, Naive, Horspool, BMH, Two-Way, Simon), налаштування кількості паралельних потоків/процесів (від 1 до 32) для пошуку в файлах та базі даних, вимірювання та відображення часу виконання пошукових операцій, виконання порівняльного тестування всіх реалізованих алгоритмів для заданих умов, а також відображення результатів пошуку та порівняння алгоритмів у зручному вигляді (списки, таблиці, візуальні індикатори).

2.2 Експлуатаційне призначення

Експлуатаційним призначенням програмного забезпечення є полегшення проведення дослідницьких, інженерних та освітніх завдань, пов'язаних з алгоритмами повнотекстового пошуку, а саме – спрощення процесу вибору оптимальних алгоритмів пошуку для розробників ПЗ шляхом надання інструменту для обґрунтованого порівняння, забезпечення дослідників контрольованим середовищем для тестування та аналізу продуктивності алгоритмів, покращення якості навчального процесу для студентів шляхом надання практичного інструменту для вивчення алгоритмів та методів оцінки їх ефективності, а також забезпечення наочної візуалізації відмінностей у продуктивності алгоритмів залежно від умов задачі.

3 Вимоги до програмного забезпечення

3.1 Вимоги до функціональних характеристик

3.1.1 Вимоги до складу виконуваних функцій

В рамках розробки ПЗ «SearchMaster» необхідно виконати наступні завдання:

- Пошук рядка: Система повинна дозволяти користувачеві вводити рядок (шаблон) для пошуку.
- Вибір джерела даних: Користувач повинен мати можливість обрати джерело даних для пошуку з наступних варіантів:
  - Локальні текстові файли (колекція файлів, що зберігаються на сервері).
  - База даних PostgreSQL (таблиця з текстовими документами).
  - Веб-пошук (використання зовнішньої пошукової системи, наприклад, Bing, для отримання URL та контенту сторінок).
- Вибір алгоритму пошуку: Користувач повинен мати можливість обрати один з 10 реалізованих алгоритмів для виконання пошуку: Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt (KMP), Bitap, Sunday, Naive (Brute Force), Horspool, Boyer-Moore-Horspool (BMH), Two-Way, Simon.

- Налаштування паралелізму: Користувач повинен мати можливість вказати кількість потоків/процесів (від 1 до 32) для виконання паралельного пошуку (для джерел «Файли» та «БД»).
- Виконання пошуку: Система повинна виконувати пошук обраним алгоритмом у вибраному джерелі з заданою кількістю потоків.
- Відображення результатів:
  - Для джерел «Файли» та «БД»: відображати список назв/ідентифікаторів знайдених документів.
  - Для джерела «Веб»: відображати список знайдених URL з коротким описом (як у пошуковій видачі).
  - Відображати час виконання пошуку.
  - Відображати інформацію про використаний алгоритм, джерело та кількість потоків.
- Порівняння алгоритмів: Система повинна надавати функцію порівняння продуктивності усіх реалізованих алгоритмів для заданого пошукового запиту, джерела даних та кількості потоків. Результати порівняння повинні включати:
  - Назву алгоритму.
  - Час виконання (сек).
  - Кількість знайдених результатів.
  - Візуальне представлення відносної продуктивності (у % від найшвидшого).
  - Визначення найшвидшого алгоритму для даного тесту. Таблиця має бути відсортована за зростанням часу виконання.
- Паралельна обробка: Система повинна використовувати модуль multiprocessing Python для паралельного виконання пошуку по окремих файлах або записах БД, якщо обрана кількість потоків > 1.
- Генерація даних: Система повинна передбачати використання скриптів (txt_generator.py, db_generator.py) для генерації та завантаження тестових даних.

3.1.2 Вимоги до організації вхідних та вихідних даних

- Вхідні дані:
  - Рядок-шаблон для пошуку (вводиться користувачем).
  - Вибір джерела даних (зі списку).
  - Вибір алгоритму пошуку (зі списку).
  - Кількість потоків/процесів (число від 1 до 32, вводиться користувачем).
  - Тестові дані: текстові файли у форматі UTF-8; файл texts.csv (CSV з роздільником ';') для мапінгу файлів; дані в таблиці text_documents БД PostgreSQL (структура: id (Integer, PK), title (String), content (Text), source_type (String)).
  - Вихідні дані:
  - Результати пошуку: список назв/ідентифікаторів файлів/записів БД або список URL з описом.
  - Час виконання пошуку (числове значення).
  - Таблиця порівняння алгоритмів (структуровані дані: назва, час, кількість результатів, відносна продуктивність).
  - Інформаційні повідомлення для користувача (успіх, помилка).
  - HTML-сторінки веб-інтерфейсу.

3.2 Вимоги до надійності

3.2.1 Вимоги до забезпечення надійності функціонування програмного забезпечення

- ПЗ повинно коректно обробляти помилки, пов'язані з доступом до джерел даних (відсутність файлу, помилка підключення до БД, помилка мережі при веб-пошуку).
- ПЗ повинно обробляти некоректні вхідні дані від користувача (наприклад, нечислове значення для кількості потоків).

- Збій при обробці одного файлу/запису БД в паралельному режимі не повинен призводити до зупинки обробки інших файлів/записів або до краху всієї системи.
- ПЗ повинно інформувати користувача про виникнення помилок через відповідні повідомлення в інтерфейсі.

3.2.2 Час відновлення після відмови

У разі некритичної відмови (наприклад, помилка обробки одного запиту) система повинна продовжувати роботу. У разі критичної відмови (наприклад, падіння веб-сервера) відновлення працездатності здійснюється шляхом перезапуску серверного процесу вручну або за допомогою засобів управління процесами операційної системи. Спеціальних вимог до автоматичного відновлення не висувається.

3.2.3 Вимоги до контролю вхідної та вихідної інформації

- ПЗ повинно виконувати базову валідацію вхідних даних від користувача (наприклад, перевірка діапазону для кількості потоків).
- ПЗ повинно забезпечувати відповідність виведених результатів пошуку та порівняння фактично отриманим даним.
- Необхідно забезпечити коректне відображення часу виконання операцій.

3.3 Вимоги до умов експлуатації

3.3.1 Кліматичні умови експлуатації

ПЗ призначене для експлуатації на стандартному комп'ютерному обладнанні (сервер, клієнтські ПК) в умовах, що відповідають вимогам експлуатації даного обладнання (стандартні офісні умови). Спеціальних вимог до кліматичних умов не висувається.

3.3.2 Вимоги до чисельності та кваліфікації користувачів

- Чисельність: Система розрахована на одночасну роботу невеликої кількості користувачів (типово для освітніх та дослідницьких цілей).
- Кваліфікація: Передбачається, що користувачі мають базові навички роботи з комп'ютером та веб-браузером. Для ефективного використання (інтерпретації результатів) бажані базові знання про алгоритми пошуку рядків. Спеціальної підготовки для роботи з інтерфейсом не потрібно.

3.4 Вимоги до складу і параметрів технічних засобів

- Сервер:
  - Процесор: Багатоядерний (рекомендовано 4+ ядер).
  - Оперативна пам'ять: Мінімум 4 ГБ (рекомендовано 8 ГБ+).
  - Дисковий простір: Достатній для ОС, ПЗ, бібліотек, тестових даних та БД.
- Клієнт:
  - ПК, ноутбук або планшет з доступом до мережі.
  - Сучасний веб-браузер.

3.5 Вимоги до інформаційної та програмної сумісності

- Серверна частина:
  - Операційна система: Linux (рекомендовано), macOS або Windows, що підтримує Python 3.11+ та PostgreSQL 12+.
  - Інтерпретатор Python: Версія 3.11 або новіша.
  - СУБД: PostgreSQL версії 12 або новішої.
  - Необхідні бібліотеки Python: fastapi, uvicorn[standard], SQLAlchemy, psycopg2-binary, requests, beautifulsoup4, python-multipart.
- Клієнтська частина:

- Сучасний веб-браузер з підтримкою HTML5, CSS3 та JavaScript (наприклад, Google Chrome, Mozilla Firefox, Microsoft Edge останніх версій).

- Взаємодія та формати:
  - Взаємодія з БД: Через стандартний інтерфейс SQLAlchemy та драйвер psycopg2.
  - Взаємодія з веб-джерелом (Bing API): Через стандартні HTTP-запити (бібліотека requests).
  - Формат текстових файлів для пошуку: UTF-8.
  - Формат файлу мапінгу текстів: texts.csv (CSV з роздільником ';').
  - Формат даних у БД: Таблиця text_documents з полями id (Integer, PK), title (String), content (Text), source_type (String).
  - Веб-інтерфейс: HTML, CSS, JavaScript.

3.6 Вимоги до маркування та пакування

Вимоги до маркування та пакування не висуваються, оскільки ПЗ розповсюджується у вигляді вихідного коду або розгортається на сервері.

3.7 Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не висуваються. Зберігання здійснюється на носіях інформації сервера та системи контролю версій.

4 Вимоги до програмної документації

Склад програмної документації повинен включати:

- Технічне завдання;
- Програму та методику випробувань;
- Інструкцію користувача;
- Опис програмного забезпечення;
- Текст програми

5 Техніко-економічні показники

У рамках кваліфікаційної роботи розрахунок техніко-економічних показників не проводився.

6 Стадії та етапи розробки

Розробка ПЗ «SearchMaster» включає наступні стадії та етапи, представлені в таблиці А.1.

Таблиця А.1 – Стадії та етапи розробки програмного забезпечення

Етап роботи	Дата початку	Дата завершення
1. Аналіз практичної задачі та існуючих аналогів	12.02.2025	19.02.2025
2. Формулювання вимог та розробка Технічного завдання (ТЗ)	20.02.2025	28.02.2025
3. Вибір технологічного стеку та проєктування архітектури ПЗ	01.03.2025	08.03.2025
4. Проєктування інтерфейсу користувача (UI)	09.03.2025	15.03.2025
5. Реалізація базових класів та алгоритмів пошуку	16.03.2025	26.03.2025
6. Реалізація модулів джерел даних (Файли, БД, Веб) та паралелізму	27.03.2025	06.04.2025
7. Реалізація API (FastAPI) та інтерфейсу користувача (HTML/CSS/JS)	07.04.2025	17.04.2025
8. Інтеграція модулів та тестування	18.04.2025	25.04.2025
9. Підготовка документації	25.04.2025	30.04.2025

Представлений графік відображає систематичний підхід до реалізації проєкту, який охоплює всі ключові етапи від початкового аналізу до підготовки

фінальної документації. Такий структурований план розробки забезпечує ефективне управління процесом та своєчасне досягнення поставлених цілей.

7 Порядок контролю та приймання

Контроль за розробкою ПЗ здійснюється шляхом періодичних консультацій та перевірки проміжних результатів.

Приймання ПЗ здійснюється на основі:

- Перевірки відповідності реалізованого функціоналу вимогам даного Технічного завдання.
- Демонстрації працездатності ПЗ: виконання пошуку з різними параметрами, виконання порівняльного тестування для різних джерел даних та кількості потоків.
- Перевірки наявності та відповідності програмної документації.

У разі знаходження помилок під час приймання програмного виробу складається акт про знайдені помилки, який підписується представниками замовника (керівника практики від університету) і розробника (студента) і затверджуються керівником організації - замовника та організації - розробника. Розробник повинен на протязі не більше ніж 2 тижнів виправити зазначені помилки і оповістити замовника про повторне проведення прийому.

ДОДАТОК Б – Опис програмного забезпечення «SearchMaster»

1 Загальні відомості

- Позначення та найменування програми: Програмне забезпечення «SearchMaster» (далі – ПЗ «SearchMaster»). Повна назва: Програмне забезпечення повнотекстового пошуку з багатопоточними алгоритмами.
- Програмне забезпечення, необхідне для функціонування програми:
  - Серверна частина:
    - Операційна система: Linux (рекомендовано), macOS або Windows, що підтримує Python 3.11+ та PostgreSQL 12+.
    - Інтерпретатор Python: Версія 3.11 або новіша.
    - Система управління базами даних (СУБД): PostgreSQL версії 12 або новішої (якщо використовується джерело даних «БД»).
    - Веб-сервер ASGI: Uvicorn.
    - Необхідні бібліотеки Python: FastAPI, uvicorn[standard], SQLAlchemy, psycopg2-binary, requests, beautifulsoup4, python-multipart, Jinja2.
  - Клієнтська частина:
    - Сучасний веб-браузер з підтримкою HTML5, CSS3 та JavaScript (наприклад, Google Chrome, Mozilla Firefox, Microsoft Edge останніх версій).
- Мови програмування, на яких написана програма:
  - Серверна частина (бекенд): Python.
  - Клієнтська частина (фронтенд): HTML, CSS, JavaScript.

2 Функціональне призначення

- Призначення програми: ПЗ «SearchMaster» призначене для надання користувачам інструментарію для дослідження та порівняння ефективності різних алгоритмів повнотекстового пошуку рядків. Програма забезпечує:
    - Оперативне виконання пошуку заданого рядка (шаблону) в текстових даних.
    - Можливість вибору джерела даних для пошуку:
        - Локальні текстові файли (колекція файлів, що зберігаються на сервері).
        - База даних PostgreSQL (таблиця з текстовими документами).
        - Веб-пошук (використання зовнішньої пошукової системи Bing для отримання URL та контенту сторінок).
    - Вибір одного з 10 реалізованих алгоритмів повнотекстового пошуку: Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt (KMP), Bitap, Sunday, Naive (Brute Force), Horspool, Boyer-Moore-Horspool (BMH), Two-Way, Simon.
    - Налаштування кількості паралельних потоків/процесів (від 1 до 32) для виконання пошуку в джерелах «Файли» та «БД».
    - Вимірювання та відображення часу виконання пошукових операцій.
    - Виконання порівняльного тестування всіх реалізованих алгоритмів для заданих умов (пошуковий запит, джерело даних, кількість потоків).
    - Відображення результатів пошуку (списки знайдених файлів/документів/URL) та результатів порівняння алгоритмів (таблиці, візуальні індикатори продуктивності) у зручному для користувача вигляді через веб-інтерфейс. ПЗ може використовуватися

в освітніх цілях для вивчення алгоритмів пошуку, в дослідницьких цілях для аналізу їх продуктивності, а також як допоміжний інструмент при розробці пошукових підсистем.

- Відомості про функціональні обмеження при використанні:
  - Кількість паралельних потоків/процесів для пошуку в файлах та БД обмежена діапазоном від 1 до 32.
  - Функціональність веб-пошуку залежить від доступності та політик використання зовнішнього Bing Search API.
  - Максимальний обсяг даних, що може бути ефективно оброблений, залежить від апаратних ресурсів сервера (CPU, RAM, швидкість дискової підсистеми).
  - Програма не призначена для індексації та постійного зберігання великих обсягів даних, як це роблять повноцінні пошукові системи; фокус зроблено на прямому порівнянні алгоритмів на “сирих” даних.

3 Опис логічної структури

- Алгоритм програми (загальний): Робота ПЗ «SearchMaster» відбувається за наступним сценарієм:
  - Користувач взаємодіє з веб-інтерфейсом, доступним через браузер.
  - На головній сторінці користувач вводить пошуковий рядок (шаблон), обирає джерело даних (текстові файли, база даних PostgreSQL або веб-пошук), обирає один з 10 доступних алгоритмів пошуку та вказує бажану кількість потоків для паралельної обробки (для файлів та БД).

- При натисканні кнопки «Search» (для одиночного пошуку) або «Compare Algorithms» (для порівняння всіх алгоритмів) формується HTTP GET-запит до серверної частини (FastAPI додаток).
- Серверна частина отримує параметри запиту, валідує їх.
- На основі параметра source обирається відповідний клас-обробник джерела даних (наприклад, FileSearchSource, DatabaseSearchSource або WebSearchSource).
- На основі параметра algorithm обирається відповідний клас-реалізація алгоритму пошуку.
- Для джерел «Файли» та «БД», якщо вказана кількість потоків більша за 1, використовується модуль multiprocessing Python для створення пулу процесів та паралельного виконання пошуку в окремих файлах або записах БД.
- Обраний алгоритм виконує пошук заданого шаблону у вказаному джерелі даних. Вимірюється час виконання операції пошуку.
- Результати пошуку (список знайдених файлів/документів/URL, інформація про час виконання, використаний алгоритм та кількість потоків) повертаються на клієнтську частину.
- У випадку одиночного пошуку (ендпоінт /item), сервер рендерить HTML-сторінку search_result.html з результатами.
- У випадку запиту на порівняння алгоритмів (ендпоінт /performance, викликається асинхронно для кожного алгоритму), сервер повертає JSON-відповідь з даними про продуктивність конкретного алгоритму. Клієнтський JavaScript агрегує ці дані та динамічно формує таблицю порівняння на головній сторінці.
- Користувацький інтерфейс відображає результати пошуку або таблицю порівняння продуктивності алгоритмів.

– Методи, що використовуються:

- Алгоритми повнотекстового пошуку: Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt (KMP), Bitap, Sunday, Naive (Brute Force), Horspool, Boyer-Moore-Horspool (BMH), Two-Way, Simon. Кожен алгоритм реалізований як окремий клас.
- Методи паралельної обробки: Використання класу multiprocessing.Pool зі стандартної бібліотеки Python для створення пулу робочих процесів та розподілу завдань пошуку (в окремих файлах або записах БД) між ними за допомогою методу pool.map або pool.starmap.
- Методи взаємодії з базою даних: Використання SQLAlchemy ORM для визначення моделі даних (TextDocument), створення сесій, виконання запитів на вибірку даних з таблиці text_documents в PostgreSQL.
- Методи веб-взаємодії:
- Виконання HTTP GET-запитів до Bing Search API за допомогою бібліотеки requests.
- Парсинг HTML-відповідей від Bing API для вилучення URL, заголовків та коротких описів сторінок за допомогою бібліотеки BeautifulSoup4.
- Методи обробки HTTP-запитів на сервері: Використання FastAPI для визначення маршрутів (ендпоінтів), обробки параметрів запиту, валідації даних та формування HTTP-відповідей (HTML або JSON).
- Методи роботи з шаблонами: Використання шаблонізатора Jinja2 для генерації динамічних HTML-сторінок на сервері.
- Методи взаємодії на клієнті: Використання JavaScript (Fetch API) для виконання асинхронних запитів до сервера, обробки JSON-відповідей та динамічного оновлення DOM (Document Object Model) для відображення результатів.

– Структура програми з описом функцій складових частин і зв’язку між ними: Проєкт має модульну архітектуру, що складається з наступних ключових компонентів:

- main.py (Основний модуль FastAPI додатку):
  - Ініціалізує FastAPI додаток.
  - Визначає API ендпоінти:
    - / (GET): Рендерить головну сторінку index.html, передаючи списки доступних алгоритмів та джерел.
    - /item (GET): Обробляє запит на виконання одиночного пошуку, викликає відповідні модулі джерел та алгоритмів, вимірює час, рендерить сторінку результатів search_result.html.
    - /performance (GET): Обробляє асинхронний запит для тестування продуктивності одного алгоритму, повертає результат у JSON.
  - Монтує директорію зі статичними файлами (static/).
  - Налаштовує шаблонізатор Jinja2 для роботи з директорією templates/.
  - Містить фабричну функцію get_search_source() для отримання екземпляра класу джерела даних.
  - Координує взаємодію між користувацькими запитами, модулями алгоритмів та модулями джерел даних.
- Директорія algorithms/ (Модулі алгоритмів пошуку):
  - base.py: Містить базовий клас BaseAlgorithm, що визначає спільний інтерфейс для всіх алгоритмів пошуку (метод search).
  - Окремі файли для кожного алгоритму (наприклад, kmp.py, boyer_moore.py): Кожен файл містить клас, що успадковує BaseAlgorithm та реалізує специфічну логіку відповідного

алгоритму пошуку, включаючи необхідні функції препроцесингу.

- Директорія search_sources/ (Модулі джерел даних):
  - base.py: Містить базовий клас BaseSearchSource, що визначає спільний інтерфейс для роботи з різними джерелами даних (метод search).
  - file_source.py: Реалізує логіку пошуку в локальних текстових файлах. Зчитує шляхи до файлів з texts.csv. Реалізує паралельну обробку файлів за допомогою multiprocessing.Pool, де кожен процес обробляє один або декілька файлів.
  - db_source.py: Реалізує логіку пошуку в базі даних PostgreSQL. Взаємодіє з БД через SQLAlchemy для отримання текстового контенту. Реалізує паралельну обробку записів БД.
  - web_source.py: Реалізує логіку веб-пошуку. Виконує HTTP-запити до Bing API за допомогою бібліотеки requests, парсить HTML-відповідь за допомогою BeautifulSoup4 для вилучення URL та сніпетів.
- database.py (Модуль бази даних):
  - Налаштовує підключення до PostgreSQL за допомогою SQLAlchemy.
  - Визначає модель даних TextDocument для таблиці text_documents.
  - Надає функції для створення сесії БД (get_session_factory()) та ініціалізації таблиць (init_db()).
- Директорія scripts/ (Допоміжні скрипти):
  - txt_generator.py: Скрипт для генерації тестових текстових файлів та файлу мапінгу texts.csv.
  - db_generator.py: Скрипт для генерації тестових даних та заповнення таблиці text_documents в PostgreSQL.

- Директорія templates/ (HTML-шаблони):
  - index.html: Шаблон головної сторінки з формою пошуку, селекторами та контейнером для результатів порівняння.
  - search_result.html: Шаблон сторінки для відображення результатів одиночного пошуку.
- Директорія static/ (Статичні файли):
  - css/style.css: CSS-стилі для оформлення веб-інтерфейсу.
  - js/script.js: JavaScript-код для обробки подій на клієнті, виконання асинхронних запитів до ендпоінта /performance, обробки JSON-відповідей та динамічного оновлення HTML-сторінки (відображення таблиці порівняння, повідомлень).
- texts.csv (Файл мапінгу текстів): CSV-файл, що містить відповідність між ідентифікаторами текстів та шляхами до файлів у директорії texts/ (використовується FileSearchSource).

Зв’язки між компонентами:

- Користувач взаємодіє з index.html (через браузер).
- script.js обробляє дії користувача та надсилає запити до main.py.
- main.py викликає методи класів з search_sources/ та algorithms/.
- Класи з search_sources/ можуть взаємодіяти з database.py (для DatabaseSearchSource), читати texts.csv (для FileSearchSource), або використовувати requests та BeautifulSoup4 (для WebSearchSource).
- main.py використовує шаблони з templates/ для генерації HTML-відповідей.

– Зв’язок програми з іншими програмами:

- СУБД PostgreSQL: ПЗ «SearchMaster» взаємодіє з сервером PostgreSQL для читання текстових документів, якщо обрано джерело даних «БД». Взаємодія відбувається через драйвер psycopg2-binary та бібліотеку SQLAlchemy.

- Bing Search API: Для реалізації функції веб-пошуку ПЗ «SearchMaster» надсилає HTTP-запити до зовнішнього сервісу Bing Search API та обробляє отримані відповіді.
- Веб-сервер Uvicorn: FastAPI додаток main.py запускається та працює під управлінням ASGI-сервера Uvicorn, який обробляє мережеві з'єднання та передає HTTP-запити до додатку.
- Веб-браузер користувача: Клієнтська частина ПЗ «SearchMaster» виконується у веб-браузері користувача, який надсилає запити до сервера та відображає отримані дані.

4 Необхідні технічні засоби

Для роботи ПЗ «SearchMaster» необхідні наступні технічні засоби:

- Серверна частина:
  - Мінімальні вимоги:
    - Процесор (CPU): 2-ядерний, з тактовою частотою 2.0 ГГц.
    - Оперативна пам'ять (RAM): 4 ГБ.
    - Дисковий простір: 20 ГБ вільного місця (для операційної системи, ПЗ, бази даних та тестових даних).
  - Рекомендовані вимоги (для ефективного паралелізму та роботи з великими даними):
    - Процесор (CPU): 4 або більше ядер, з тактовою частотою 2.5 ГГц або вище.
    - Оперативна пам'ять (RAM): 8 ГБ або більше.
    - Дисковий простір: 50 ГБ або більше вільного місця на SSD-накопичувачі.
- Клієнтська частина:

- Будь-який персональний комп’ютер, ноутбук або планшет, здатний запустити сучасний веб-браузер.
- Підключення до мережі (локальної або Інтернет) для доступу до сервера, на якому розгорнуто ПЗ «SearchMaster».
- Пристрої вводу-виводу: клавіатура, маніпулятор типу «миша» (або сенсорний екран), монітор.

5 Виклик і завантаження

- Спосіб виклику програми з відповідного носія даних:
  - Серверна частина: Програмний код серверної частини розміщується на серверному комп’ютері. Запуск серверного додатку здійснюється з командного рядка за допомогою ASGI-сервера Uvicorn. Наприклад, команда для запуску може виглядати так:
    - uvicorn main:app --host 0.0.0.0 --port 8000

    Для розробки може використовуватися команда з автоматичним перезавантаженням:
    - uvicorn main:app --reload
  - Клієнтська частина: Доступ до ПЗ «SearchMaster» здійснюється користувачем через будь-який сучасний веб-браузер. Для цього користувач повинен ввести у адресному рядку браузера URL-адресу, за якою запущено серверний додаток (наприклад, http://<IP-адреса_сервера>:8000 або http://localhost:8000 при локальному запуску).
- Вхідні точки в програму: Основними вхідними точками для взаємодії з програмою є HTTP ендпоінти, визначені у файлі main.py FastAPI додатку:

- GET /: Головна сторінка програми. При зверненні до цього ендпоінту користувачеві відображається інтерфейс index.html для введення параметрів пошуку.
- GET /item: Ендпоінт для виконання одиночного пошуку. Приймає параметри: q (пошуковий запит), algorithm (обраний алгоритм), threads (кількість потоків), source (джерело даних). Повертає HTML-сторінку search_result.html з результатами пошуку.
- GET /performance: Ендпоінт для асинхронного тестування продуктивності одного алгоритму. Приймає ті ж параметри, що й /item. Повертає JSON-об’єкт з даними про час виконання, кількість знайдених результатів та іншу метаінформацію. Цей ендпоінт викликається багаторазово з клієнтського JavaScript для порівняння всіх алгоритмів.

6 Вхідні дані

- Характер, організація і попередня підготовка вхідних даних:
  - Пошуковий запит (рядок-шаблон): Текстовий рядок, що вводиться користувачем безпосередньо у відповідне поле веб-інтерфейсу. Спеціальної попередньої підготовки не потребує.
  - Вибір джерела даних: Обирається користувачем з випадаючого списку на веб-інтерфейсі. Доступні варіанти: «Файли», «БД PostgreSQL», «Веб-пошук».
  - Вибір алгоритму пошуку: Обирається користувачем з випадаючого списку на веб-інтерфейсі. Доступні 10 реалізованих алгоритмів.

- Кількість потоків/процесів: Ціле число, що вводиться користувачем у відповідне поле (або обирається за допомогою кнопок +/-) на веб-інтерфейсі. Діапазон значень від 1 до 32.
- Тестові дані для джерела «Файли»: Колекція текстових файлів, що зберігаються на сервері у директорії texts/. Ці файли генеруються заздалегідь за допомогою скрипта scripts/txt_generator.py. Також використовується файл texts.csv (генерується тим же скриптом), що містить мапінг ідентифікаторів текстів на шляхи до відповідних файлів.
- Тестові дані для джерела «БД PostgreSQL»: Записи в таблиці text_documents бази даних PostgreSQL. Ці записи генеруються заздалегідь за допомогою скрипта scripts/db_generator.py.

– Формат, описання та спосіб кодування вхідних даних:

- Пошуковий запит (рядок-шаблон): Текстовий рядок. Кодування UTF-8. Передається як параметр URL-запиту.
- Вибір джерела даних: Рядковий ідентифікатор (наприклад, “files”, “db”, “web”). Передається як параметр URL-запиту.
- Вибір алгоритму пошуку: Рядковий ідентифікатор (наприклад, “boyer_moore”, “kmp”). Передається як параметр URL-запиту.
- Кількість потоків/процесів: Ціле число. Передається як параметр URL-запиту.
- Текстові файли (джерело «Файли»): Файли у форматі простого тексту. Кодування UTF-8.
- Файл texts.csv: Файл формату CSV (Comma-Separated Values) з крапкою з комою (;) як роздільником. Кодування UTF-8. Структура: TextID;FilePath.
- Дані в таблиці text_documents (джерело «БД PostgreSQL»):
  - id: INTEGER, первинний ключ.
  - title: VARCHAR або TEXT.

- content: TEXT (містить текстовий контент для пошуку).
- source_type: VARCHAR (наприклад, “database”).

Кодування даних у БД – UTF-8.

7 Вихідні дані

- Характер і організація вихідних даних:
  - Результати одиночного пошуку (при запиті до /item):
    - Для джерел «Файли» та «БД»: Список назв/ідентифікаторів файлів або документів, у яких знайдено шаблон.
    - Для джерела «Веб-пошук»: Список URL знайдених веб-сторінок разом з їх заголовками та короткими описами (сніпетами).
    - Метадані пошуку: використаний пошуковий запит, обраний алгоритм, джерело даних, кількість потоків, загальний час виконання пошуку.
  - Результати порівняння алгоритмів (при запиті до /performance та агрегації на клієнті):
    - Таблиця, що містить для кожного з 10 алгоритмів: назву алгоритму, час виконання (в секундах), кількість знайдених результатів (має бути однаковою для всіх алгоритмів при коректній роботі), відносну продуктивність (у відсотках від найшвидшого алгоритму).
    - Визначення найшвидшого алгоритму для даного тесту.
  - Інформаційні повідомлення для користувача: Текстові повідомлення про статус операцій (наприклад, “Пошук завершено”, “Помилка під час пошуку”, “Результати порівняння готові”).
- Формат, опис і спосіб кодування вихідних даних:
  - Для ендпоінта /item (одиночний пошук):

- HTML-сторінка, згенерована за допомогою шаблону search_result.html. Кодування UTF-8. Візуально представляє список знайдених результатів та метадані пошуку.

- Для ендпоінта /performance (тестування продуктивності одного алгоритму):
  - JSON-об’єкт. Кодування UTF-8. Структура:

```
{
 "algorithm_name": "string", // Назва алгоритму
 "execution_time": "float", // Час виконання в секундах
 "thread_count": "integer", // Використана кількість потоків
 "results_found_count": "integer", // Кількість знайдених входжень/документів
 "has_results": "boolean" // Чи були знайдені результати
}
```

- Відображення на клієнтській стороні (в браузері):
  - Результати одиночного пошуку: Відображаються як частина HTML-сторінки.
  - Результати порівняння алгоритмів: Динамічно генеруються JavaScript у вигляді HTML-таблиці на головній сторінці (index.html). Таблиця містить стовпці: “Алгоритм”, “Час виконання (сек)”, “Знайдено результатів”, “Продуктивність” (з візуальною смугою).
  - Інформаційні повідомлення: Можуть відображатися як текстові елементи на сторінці або у вигляді спливаючих toast-повідомлень.
  - Все текстове відображення використовує кодування UTF-8.

ДОДАТОК В – Програма та методика випробувань ПЗ «SearchMaster»

1 Об’єкт випробувань

- Найменування випробуваної програми: Програмне забезпечення «SearchMaster» (далі – ПЗ «SearchMaster»).
- Область застосування випробуваної програми:
  - Дослідження алгоритмів повнотекстового пошуку.
  - Порівняльний аналіз продуктивності різних алгоритмів пошуку рядків.
  - Освітні цілі в галузі інженерії програмного забезпечення та алгоритмів.
  - Допоміжний інструмент при розробці пошукових підсистем.
- Позначення випробуваної програми: ПЗ «SearchMaster».

2 Мета випробувань

Метою проведення випробувань ПЗ «SearchMaster» є:

- Перевірка відповідності розробленого програмного забезпечення функціональним та нефункціональним вимогам, викладеним у Технічному завданні.
- Підтвердження коректної роботи всіх реалізованих алгоритмів повнотекстового пошуку (Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt, Bitap, Sunday, Naive, Horspool, Boyer-Moore-Horspool, Two-Way, Simon).
- Валідація функціональності вибору джерел даних (локальні файли, база даних PostgreSQL, веб-пошук через Bing API).
- Перевірка коректності реалізації паралельної обробки даних для джерел «Файли» та «БД» та оцінка її впливу на продуктивність.

- Забезпечення того, що користувацький інтерфейс є інтуїтивно зрозумілим, коректно відображає параметри пошуку, результати та дані порівняння алгоритмів.
- Оцінка та порівняння продуктивності різних алгоритмів пошуку за різних умов (тип джерела даних, кількість потоків, характеристики пошукового запиту).
- Виявлення та документування можливих помилок, дефектів або невідповідностей у роботі програмного забезпечення.

3 Вимоги до програми

Під час випробувань перевірці підлягають наступні вимоги до ПЗ «SearchMaster», задані в Технічному завданні:

- Функціональні вимоги:
  - Можливість введення користувачем рядка (шаблону) для пошуку.
  - Можливість вибору користувачем одного з трьох джерел даних: локальні текстові файли, база даних PostgreSQL, веб-пошук.
  - Можливість вибору користувачем одного з 10 реалізованих алгоритмів повнотекстового пошуку.
  - Можливість налаштування користувачем кількості паралельних потоків/процесів (від 1 до 32) для виконання пошуку в джерелах «Файли» та «БД».
  - Коректне виконання пошуку обраним алгоритмом у вибраному джерелі даних із заданою кількістю потоків.
  - Коректне відображення результатів пошуку:
    - Для джерел «Файли» та «БД»: список назв/ідентифікаторів знайдених документів.

- Для джерела «Веб»: список знайдених URL з коротким описом.

- Коректне відображення часу виконання пошуку.
- Коректне відображення інформації про використаний алгоритм, джерело даних та кількість потоків.
- Наявність функції порівняння продуктивності всіх реалізованих алгоритмів для заданого пошукового запиту, джерела даних та кількості потоків.
- Коректне відображення результатів порівняння у вигляді таблиці, що включає: назву алгоритму, час виконання, кількість знайдених результатів, відносну продуктивність.
- Коректне визначення та індикація найшвидшого алгоритму для даного тесту порівняння.
- Коректна робота скриптів генерації тестових даних (txt_generator.py, db_generator.py).

– Вимоги до надійності:

- Коректна обробка помилок, пов’язаних з доступом до джерел даних (відсутність файлу, помилка підключення до БД, помилка мережі при веб-пошуку).
- Коректна обробка некоректних вхідних даних від користувача (наприклад, нечислове значення для кількості потоків).
- Збій при обробці одного файлу/запису БД в паралельному режимі не повинен призводити до зупинки обробки інших файлів/записів або до краху всієї системи.
- Інформування користувача про виникнення помилок через відповідні повідомлення в інтерфейсі.

– Вимоги до інтерфейсу користувача:

- Інтуїтивно зрозумілий та зручний веб-інтерфейс для введення параметрів пошуку та перегляду результатів.

- Чітке та однозначне відображення результатів пошуку та порівняння алгоритмів.

4 Вимоги до програмної документації

- Склад програмної документації, що пропонується на випробування:
  - Технічне завдання на розробку ПЗ «SearchMaster».
  - Опис програмного забезпечення ПЗ «SearchMaster».
  - Програма та методика випробувань ПЗ «SearchMaster».
  - Керівництво користувача (оператора) ПЗ «SearchMaster».
  - Текст програми.
- Спеціальні вимоги: Спеціальні вимоги до програмної документації, окрім її повноти, коректності та відповідності стандартам, у Технічному завданні не вказані.

5 Засоби і порядок випробувань

- Технічні засоби, що використовуються під час випробувань:
  - Серверна частина:
    - Процесор: Intel Core i5 (або аналогічний) 4+ ядер, 2.5+ ГГц.
    - Оперативна пам’ять: 8 ГБ DDR4.
    - Накопичувач: SSD 256 ГБ (з достатнім вільним місцем для ОС, ПЗ, БД та тестових даних).
    - Операційна система: Linux (наприклад, Ubuntu 22.04 LTS) або Windows 10/11.
  - Клієнтська частина:

- Персональний комп’ютер або ноутбук з підключенням до мережі (для доступу до сервера).
- Сучасний веб-браузер: Google Chrome (остання версія), Mozilla Firefox (остання версія).

– Програмні засоби, що використовуються під час випробувань:

- Серверна частина:
  - Інтерпретатор Python версії 3.11.
  - СУБД PostgreSQL версії 14.
  - Веб-сервер Uvicorn.
  - Фреймворк FastAPI.
  - Бібліотеки Python: SQLAlchemy, psycopg2-binary, requests, beautifulsoup4, python-multipart, Jinja2.
- Клієнтська частина:
  - Веб-браузер (Google Chrome, Mozilla Firefox) з інструментами розробника (для перегляду консолі, мережевих запитів).
- Інструментальні засоби:
  - Середовище розробки: Visual Studio Code (для перегляду коду та логів).
  - Система контролю версій: Git.
  - Скрипти для генерації тестових даних: txt_generator.py, db_generator.py.

– Порядок проведення випробувань:

- Підготовчий етап:
  - Розгортання ПЗ «SearchMaster» на сервері згідно з інструкціями з встановлення.
  - Налаштування та запуск сервера СУБД PostgreSQL.

- Створення бази даних та таблиць за допомогою скрипта database.py (init_db()).
- Генерація тестових даних:
  - Запуск scripts/txt_generator.py для створення 500 текстових файлів (обсягом приблизно 20000 слів кожен) у директорії texts/ та файлу texts.csv.
  - Запуск scripts/db_generator.py для заповнення таблиці text_documents у PostgreSQL 100 записами (з текстом обсягом приблизно 20000 слів кожен).
- Перевірка доступності веб-інтерфейсу ПЗ «SearchMaster» через браузер.

- Основний етап (виконання тестів):
  - Проведення функціонального тестування згідно з описаними методами та тест-кейсами (див. розділ 6).
  - Проведення тестування продуктивності та порівняння алгоритмів згідно з описаними сценаріями (див. розділ 6).
  - Проведення тестування обробки помилок та стійкості.
  - Фіксація результатів кожного тест-кейсу: опис кроків, очікуваний результат, фактичний результат, статус (пройдено/не пройдено), коментарі (за потреби).
- Завершальний етап:
  - Аналіз отриманих результатів тестування.
  - Порівняння фактичних результатів з очікуваними.
  - Виявлення та документування всіх невідповідностей, помилок та дефектів.
  - Складання звіту про результати випробувань.

6 Методи випробувань

Випробування ПЗ «SearchMaster» проводяться з використанням наступних методів:

- Метод функціонального тестування (ручне тестування): Цей метод передбачає перевірку відповідності реалізованих функцій ПЗ вимогам, зазначеним у Технічному завданні, шляхом безпосередньої взаємодії з користувацьким інтерфейсом.

  - Тест-кейс 1: Перевірка базової функціональності пошуку.
  - Опис: Перевірка коректності пошуку для існуючих та неіснуючих пошукових запитів для кожної комбінації з 10 алгоритмів та 3 джерел даних.
  - Кроки:
    - Відкрити головну сторінку ПЗ.
    - Обрати джерело даних (Файли/БД/Веб).
    - Обрати алгоритм пошуку.
    - Встановити кількість потоків (наприклад, 8).
    - Ввести пошуковий запит, який гарантовано існує у вибраному джерелі (наприклад, “dsds” для згенерованих даних).
    - Натиснути кнопку «Search».
    - Перевірити: коректність відображення списку знайдених документів/URL, відображення часу виконання, назви алгоритму, джерела та кількості потоків.
    - Повторити кроки 5-7, ввівши пошуковий запит, який гарантовано відсутній у джерелі. Перевірити відображення повідомлення про відсутність результатів.
    - Повторити для всіх 10 алгоритмів та 3 джерел даних.
  - Очікуваний результат: ПЗ коректно знаходить та відображає результати для існуючих запитів та повідомляє про їх

відсутність для неіснуючих. Метадані пошуку відображаються правильно.

- Тест-кейс 2: Перевірка вибору параметрів пошуку.
  - Опис: Перевірка коректної роботи елементів керування для вибору джерела даних, алгоритму та кількості потоків.
  - Кроки:
    - Послідовно обрати кожне значення у випадаючому списку «Search Source».
    - Послідовно обрати кожне значення у випадаючому списку «Algorithm».
    - Ввести різні значення у поле «Thread Count» (наприклад, 1, 8, 32, а також некоректні: 0, 33, текстове значення).
  - Очікуваний результат: Усі вибори реєструються коректно. Некоректні значення для кількості потоків обробляються (наприклад, ігноруються, встановлюється значення за замовчуванням або відображається помилка валідації).
- Тест-кейс 3: Перевірка функціональності паралелізму.
  - Опис: Перевірка зміни часу виконання при зміні кількості потоків для джерел «Файли» та «БД».
  - Кроки:
    - Обрати джерело «Файли» та один з швидких алгоритмів (наприклад, Horspool).
    - Виконати пошук з пошуковим запитом “dsds” при кількості потоків 1. Зафіксувати час.
    - Повторити пошук з тим же запитом при кількості потоків 4. Зафіксувати час.
    - Повторити пошук з тим же запитом при кількості потоків 8. Зафіксувати час.

- Порівняти отримані часи.
- Повторити кроки 1-5 для джерела «БД».

- Очікуваний результат: Час виконання зменшується при збільшенні кількості потоків (особливо помітно для файлів).

- Тест-кейс 4: Перевірка функціональності порівняння алгоритмів.
  - Опис: Перевірка коректності роботи функції «Compare Algorithms».
  - Кроки:
    - Обрати джерело даних (наприклад, «Файли»).
    - Встановити кількість потоків (наприклад, 8).
    - Ввести пошуковий запит (наприклад, “dsds”).
    - Натиснути кнопку «Compare Algorithms».
  - Очікуваний результат: Відображається таблиця порівняння для всіх 10 алгоритмів, що містить: назву алгоритму, час виконання, кількість знайдених результатів (має бути однаковою для всіх алгоритмів, що знайшли запит), відносну продуктивність. Найшвидший алгоритм коректно визначений та підсвічений.
- Тест-кейс 5: Перевірка веб-пошуку.
  - Опис: Перевірка отримання результатів від Bing API.
  - Кроки:
    - Обрати джерело «Веб-пошук».
    - Ввести пошуковий запит (наприклад, “Test query”).
    - Натиснути кнопку «Search».
  - Очікуваний результат: Відображається список URL та описів, отриманих від Bing. Відображається час виконання.

– Метод тестування продуктивності (на основі функції «Compare Algorithms»): Цей метод використовується для кількісної оцінки та порівняння

швидкодії різних алгоритмів за різних умов. Результати фіксуються з таблиць, що генеруються ПЗ.

- Сценарій 1: Джерело – Текстові файли.
  - Дані: 500 текстових файлів, ~20000 слів кожен. Пошуковий запит: «dsds».
  - Підтест 1.1: Кількість потоків = 8. Зафіксувати дані з таблиці порівняння.
  - Підтест 1.2: Кількість потоків = 4. Зафіксувати дані.
  - Підтест 1.3: Кількість потоків = 1. Зафіксувати дані.
  - Очікуваний результат: Дані відповідають або близькі до наведених у звіті. Horspool та Sunday показують високу продуктивність.
- Сценарій 2: Джерело – База даних PostgreSQL.
  - Дані: 100 записів у БД, ~20000 слів кожен. Пошуковий запит: «dsds».
  - Підтест 2.1: Кількість потоків = 8. Зафіксувати дані.
  - Підтест 2.2: Кількість потоків = 4. Зафіксувати дані.
  - Підтест 2.3: Кількість потоків = 1. Зафіксувати дані.
  - Очікуваний результат: Дані відповідають або близькі до наведених у звіті. Sunday та Boyer-Moore-Horspool показують високу продуктивність.
- Сценарій 3: Джерело – Веб-пошук.
  - Пошуковий запит: «dsds».
  - Підтест 3.1: Кількість потоків = 8. Зафіксувати дані.
  - Підтест 3.2: Кількість потоків = 4. Зафіксувати дані.
  - Підтест 3.3: Кількість потоків = 1. Зафіксувати дані.
  - Очікуваний результат: Дані відповідають або близькі до наведених у звіті. Час виконання значно менший.

- Метод тестування обробки помилок та стійкості:
  - Тест-кейс 6: Некоректний пошуковий запит.
    - Опис: Перевірка реакції системи на порожній пошуковий запит.
    - Кроки: Залишити поле пошукового запиту порожнім, натиснути «Search» або «Compare Algorithms».
    - Очікуваний результат: Система відображає повідомлення про необхідність ввести запит або не виконує пошук, не виникає помилок на сервері. (FastAPI валідація min_length=1 для q).
  - Тест-кейс 7: Недоступність джерела даних.
    - Опис: Перевірка реакції системи на недоступність файлів або БД.
    - Кроки:
      - Тимчасово перейменувати/видалити файл texts.csv. Спробувати виконати пошук у джерелі «Файли».
      - Зупинити сервер PostgreSQL. Спробувати виконати пошук у джерелі «БД».
    - Очікуваний результат: Система відображає користувачеві інформативне повідомлення про помилку доступу до джерела даних та не завершує роботу аварійно.
  - Тест-кейс 8: Проблеми з мережею для веб-пошуку.
    - Опис: Перевірка реакції системи на відсутність інтернет-з’єднання при веб-пошуку.
    - Кроки: Відключити інтернет-з’єднання на сервері. Спробувати виконати веб-пошук.
    - Очікуваний результат: Система відображає повідомлення про помилку мережі або неможливість з’єднатися з Bing API.

ДОДАТОК Г – Керівництво користувача (оператора) ПЗ «SearchMaster»

1 Призначення програми

- Функціональне призначення програми: Програмне забезпечення «SearchMaster» (далі – ПЗ «SearchMaster») призначене для надання користувачам інструментарію для дослідження та порівняння ефективності різних алгоритмів повнотекстового пошуку рядків. Програма дозволяє виконувати пошук заданого рядка (шаблону) в текстових даних з різних джерел (локальні файли, база даних PostgreSQL, веб-пошук), використовуючи один з 10 реалізованих алгоритмів (Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt (KMP), Bitap, Sunday, Naive (Brute Force), Horspool, Boyer-Moore-Horspool (BMH), Two-Way, Simon). Ключовою особливістю є можливість налаштування кількості паралельних потоків/процесів для пошуку та виконання порівняльного тестування всіх алгоритмів для заданих умов з візуалізацією результатів.
- Експлуатаційне призначення програми: ПЗ «SearchMaster» використовується для:
    - Освітніх цілей: Демонстрація роботи та порівняння характеристик різних алгоритмів повнотекстового пошуку студентам та особам, що вивчають комп’ютерні науки та інженерію програмного забезпечення.
    - Дослідницьких цілей: Проведення експериментального аналізу продуктивності алгоритмів пошуку на різних наборах даних та з різними параметрами паралелізації.
    - Допоміжний інструмент для розробників: Надання можливості швидко оцінити та обрати найбільш ефективний алгоритм пошуку для конкретної задачі при розробці пошукових підсистем у програмних продуктах.
- Склад функцій:

- Введення пошукового запиту: Користувач може ввести текстовий рядок (шаблон) для пошуку.
- Вибір джерела даних: Користувач може обрати одне з трьох джерел для пошуку:
    - Текстові файли (локально на сервері).
    - База даних PostgreSQL.
    - Веб-пошук (через Bing API).
- Вибір алгоритму пошуку: Користувач може обрати один з 10 реалізованих алгоритмів повнотекстового пошуку.
- Налаштування паралелізму: Користувач може вказати кількість потоків/процесів (від 1 до 32) для виконання пошуку в джерелах «Файли» та «БД».
- Виконання одиночного пошуку: Запуск пошуку з обраними параметрами та відображення результатів (знайдені документи/URL, час виконання, використаний алгоритм).
- Порівняння продуктивності алгоритмів: Запуск тестування всіх 10 алгоритмів для поточних налаштувань (запит, джерело, потоки) та відображення результатів у вигляді порівняльної таблиці (час виконання, кількість знайдених результатів, відносна продуктивність).
- Відображення результатів: Представлення результатів пошуку та порівняння у зручному для користувача вигляді через веб-інтерфейс.

2 Умови виконання програми

- Мінімальний склад апаратних засобів:
    - Серверна частина (де виконується основна логіка ПЗ):
        - Процесор (CPU): 2-ядерний, з тактовою частотою 2.0 ГГц.
        - Оперативна пам’ять (RAM): 4 ГБ.

- Дисковий простір: 20 ГБ вільного місця.

- Клієнтська частина (де користувач взаємодіє з інтерфейсом):
  - Будь-який персональний комп’ютер, ноутбук або планшет, здатний запустити сучасний веб-браузер.
  - Підключення до мережі для доступу до сервера.
  - Пристрої вводу-виводу: клавіатура, маніпулятор типу «миша» (або сенсорний екран), монітор.

– Мінімальний склад програмних засобів:

- Серверна частина:
  - Операційна система: Linux, macOS або Windows, що підтримує Python 3.11+ та PostgreSQL 12+.
  - Інтерпретатор Python: Версія 3.11 або новіша.
  - СУБД: PostgreSQL версії 12 або новішої (якщо використовується джерело даних «БД»).
  - Веб-сервер ASGI: Uvicorn.
  - Необхідні бібліотеки Python: FastAPI, uvicorn[standard], SQLAlchemy, psycopg2-binary, requests, beautifulsoup4, python-multipart, Jinja2.
- Клієнтська частина:
  - Сучасний веб-браузер з підтримкою HTML5, CSS3 та JavaScript (наприклад, Google Chrome, Mozilla Firefox, Microsoft Edge останніх версій).

– Вимоги до персоналу (користувачів):

- Користувачі повинні мати базові навички роботи з персональним комп’ютером та веб-браузером.
- Для ефективного використання програми та інтерпретації результатів порівняння алгоритмів бажані базові знання про алгоритми пошуку рядків.

- Спеціальної підготовки для роботи з інтерфейсом ПЗ «SearchMaster» не потрібно.
- Для адміністрування серверної частини (запуск, зупинка сервера, налаштування БД, генерація тестових даних) потрібні навички роботи з командним рядком та базові знання адміністрування відповідної ОС та СУБД PostgreSQL.

3 Виконання програми

- Завантаження та запуск програми:
  - Серверна частина:
    - Переконайтеся, що всі необхідні програмні компоненти (Python, PostgreSQL, бібліотеки) встановлені на сервері.
    - Переконайтеся, що сервер СУБД PostgreSQL запущений та налаштований (створена база даних та користувач, якщо це необхідно).
    - За потреби згенеруйте тестові дані за допомогою скриптів scripts/txt_generator.py та scripts/db_generator.py.
    - Перейдіть у кореневу директорію проєкту ПЗ «SearchMaster» через командний рядок.
    - Запустіть веб-сервер Uvicorn з FastAPI додатком. Типова команда для запуску:

      uvicorn main:app --host 0.0.0.0 --port 8000

      (де 0.0.0.0 дозволяє доступ з будь-якої мережевої адреси, а 8000 – порт, який прослуховується).
  - Клієнтська частина:
    - Відкрийте сучасний веб-браузер на комп’ютері користувача.

- В адресному рядку браузера введіть URL-адресу, за якою запущено серверний додаток ПЗ «SearchMaster». Наприклад, якщо сервер запущено локально на порту 8000, адреса буде http://localhost:8000. Якщо сервер знаходиться в мережі, вкажіть його IP-адресу або доменне ім’я та порт.
- Після завантаження сторінки відобразиться головний інтерфейс ПЗ «SearchMaster» (рис. 2.5).

– Виконання програми (робота з інтерфейсом): Головний інтерфейс ПЗ «SearchMaster» (рис 2.5) надає наступні можливості:

- Введення пошукового запиту:
  - У текстове поле з підказкою «Begin your search…» введіть рядок (шаблон), який необхідно знайти.
- Вибір джерела даних (Search Source):
  - Натисніть на випадаючий список під міткою «Search Source».
  - Оберіть одне з доступних джерел:
    - Text Files: Пошук буде здійснюватися у колекції текстових файлів на сервері.
    - PostgreSQL: Пошук буде здійснюватися у записах бази даних PostgreSQL.
    - Web Search: Пошук буде здійснюватися в Інтернеті за допомогою Bing API (будуть повернуті URL та сніпети).
- Вибір алгоритму пошуку (Algorithm):
  - Натисніть на випадаючий список під міткою «Algorithm».
  - Оберіть один з 10 доступних алгоритмів: Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt, Bitap, Sunday, Brute Force, Horspool, Boyer-Moore-Horspool, Two-Way, Simon.
- Налаштування кількості потоків (Thread Count):
  - Це поле активне для джерел «Text Files» та «PostgreSQL».

- Введіть бажану кількість потоків/процесів для паралельного пошуку (ціле число від 1 до 32) або скористайтеся кнопками «-» та «+» для зміни значення.

- Виконання одиночного пошуку:
  - Після введення всіх параметрів натисніть кнопку «Search →».
  - Система виконає пошук. Результати будуть відображені на новій сторінці (або в оновленій частині поточної сторінки).
  - На сторінці результатів буде вказано: час виконання пошуку, використане джерело, алгоритм, кількість потоків та список знайдених документів/URL.
  - Зі сторінки результатів можна повернутися до нового пошуку («← New Search») або скопіювати результати («❐ Copy Results»).
- Порівняння продуктивності алгоритмів:
  - Введіть пошуковий запит, оберіть джерело даних та встановіть кількість потоків.
  - Натисніть кнопку «Compare Algorithms».
  - Система послідовно виконає пошук для кожного з 10 алгоритмів з обраними параметрами.
  - Після завершення тестування під формою пошуку динамічно з’явиться таблиця «Performance Comparison for “ваш_запит”».
  - У таблиці буде відображено:
    - Назва алгоритму.
    - Час виконання (Execution Time) у секундах.
    - Кількість знайдених результатів (Results).

- Відносна продуктивність (Performance) у вигляді відсоткової смуги.

- Під таблицею буде вказано найшвидший алгоритм для даного тесту та його час виконання.

- Ви можете обрати найшвидший алгоритм для подальшого використання, натиснувши кнопку «✓ Use [Назва_алгоритму]», або закрити результати порівняння кнопкою «✕ Close».

– Завершення роботи програми:

- Клієнтська частина:

  - Для завершення роботи з ПЗ «SearchMaster» користувачеві достатньо закрити вкладку або вікно веб-браузера, в якому відкрито інтерфейс програми.

- Серверна частина:

  - Для зупинки серверного додатку ПЗ «SearchMaster» необхідно зупинити процес Uvicorn, який був запущений у командному рядку. Зазвичай це робиться натисканням комбінації клавіш Ctrl+C у вікні терміналу, де запущено сервер.

ДОДАТОК Д – Лістинги ключових програмних модулів

Д.1 Файл algorithms/parallel/__init__.py

```
from algorithms.parallel.base import ParallelAlgorithm
```

Д.2 Файл algorithms/parallel/base.py

```
import csv
from multiprocessing import Pool


class ParallelAlgorithm:
    def __init__(self, algorithm_instance):
        """
        Takes an instance of any search algorithm and runs it in parallel

        Args:
            algorithm_instance: An instance of a search algorithm class
        """
        self.algorithm = algorithm_instance
        self.__texts_names = {}
        self.__result_list = []

    @property
    def texts_names(self):
        return self.__texts_names

    @texts_names.setter
    def texts_names(self, texts_names):
        self.__texts_names = texts_names
        self.algorithm.texts_names = texts_names

    @property
    def result_list(self):
        return self.__result_list

    @result_list.setter
    def result_list(self, result_list):
        self.__result_list = result_list
        self.algorithm.result_list = result_list
```

```
    def find_in_text_parallel(self, text_number, search_string):
        text = self.algorithm.text_by_number(str(text_number))
        result = self.algorithm.search(search_string.lower(), text)
        if result:
            return self.texts_names[f"Text {text_number}"]
        return None

    def find_in_text(self, search_string, thread_count=8):
        # Ensure thread count is within reasonable bounds
        thread_count = max(1, min(32, thread_count))

        self.texts_names = {}

        with open("texts.csv", mode="r", encoding="UTF-8") as inp:
            reader = csv.reader(inp, delimiter=";")
            for rows in reader:
                self.texts_names.update({rows[0]: rows[1]})

        # Use the specified thread count
        with Pool(thread_count) as p:
            self.result_list = p.starmap(
                self.find_in_text_parallel,
                [(i, search_string) for i in range(len(self.texts_names))],
            )
        self.result_list = [result for result in self.result_list if result is not None]

        if self.result_list:
            return "Знайдено в текстах: " + ", ".join(self.result_list)
        else:
            return "Не знайдено"
```

Д.3 Файл algorithms/__init__.py

```
from algorithms.rabin_karp import Algorithm as RabinKarp
from algorithms.kmp import Algorithm as KMP
from algorithms.boyer_moore import Algorithm as BoyerMoore
from algorithms.bitap import Algorithm as Bitap
from algorithms.sunday import Algorithm as Sunday
from algorithms.naive import Algorithm as Naive
from algorithms.horspool import Algorithm as Horspool
from algorithms.bmh import Algorithm as BMH
```

```
from algorithms.two_way import Algorithm as TwoWay
from algorithms.simon import Algorithm as Simon
```

Д.4 Файл algorithms/base.py

```
import csv


class BaseAlgorithm:
    def __init__(self):
        self.__texts_names = {}
        self.__result_list = []

    @property
    def texts_names(self):
        return self.__texts_names

    @texts_names.setter
    def texts_names(self, texts_names):
        self.__texts_names = texts_names

    @property
    def result_list(self):
        return self.__result_list

    @result_list.setter
    def result_list(self, result_list):
        self.__result_list = result_list

    def text_by_number(self, number):
        with open(f"texts/{str(number)}.txt", encoding="UTF-8") as file:
            text = file.read()
            return text.lower()

    def search(self, pattern, text):
        """
        Implements the specific search algorithm.
        Must be overridden by subclasses.
        """
        raise NotImplementedError("Subclasses must implement search method")

    def find_in_text(self, search_string):
        self.texts_names = {}
```

```
        with open("texts.csv", mode="r", encoding="UTF-8") as inp:
            reader = csv.reader(inp, delimiter=";")
            for rows in reader:
                self.texts_names.update({rows[0]: rows[1]})

        self.result_list = []
        for i in range(len(self.texts_names)):
            result = {}
            result[f"Text {i}"] = self.search(
                search_string.lower(), self.text_by_number(str(i))
            )
            if result[f"Text {i}"]:
                self.result_list.append(self.texts_names[f"Text {i}"])

        if self.result_list:
            return "Знайдено в текстах: " + ", ".join(self.result_list)
        else:
            return "Не знайдено"
```

Д.5 Файл algorithms/boyer_moore.py

```
from algorithms.base import BaseAlgorithm


class Algorithm(BaseAlgorithm):
    def search(self, pattern, text):
        """Boyer-Moore string search algorithm"""
        pattern_length = len(pattern)
        text_length = len(text)

        if pattern_length == 0:
            return True
        if pattern_length > text_length:
            return False

        # Bad character rule: last occurrence of each character in pattern
        last = {}
        for i in range(pattern_length):
            last[pattern[i]] = i

        i = pattern_length - 1  # index for text
        k = pattern_length - 1  # index for pattern (from right)
```

```
    while i < text_length:
        if pattern[k] == text[i]:
            if k == 0: # Matched all characters
                return True
            i -= 1
            k -= 1
        else: # Mismatch
            # Shift based on bad character rule
            # l is the index of text[i] in pattern, or -1 if not present
            l = last.get(text[i], -1)
            # Shift is pattern_length - min(k, 1 + l)
            # but simpler: shift by k - l if l < k, else by 1 (good suffix simplified)
            # More standard Boyer-Moore shift:
            # shift = max(1, k - l)
            # i += shift
            # k = pattern_length - 1 # Reset pattern index

            # Corrected shift logic for basic Boyer-Moore (bad char only)
            # The shift is determined by the character in the text that caused the mismatch (text[i])
            # or the character aligned with the rightmost character of the pattern if the mismatch
            # occurred earlier.
            # We align the pattern such that text[i] (mismatched char) aligns with its last
            # occurrence in the pattern, or shift by pattern_length if text[i] is not in pattern.

            # The current alignment starts at text[i-k ... i ... i + (pattern_length - 1 - k)]
            # The rightmost character of the current window in text is text[i + (pattern_length - 1 - k)]
            # Let's use the standard approach:
            # i is the right end of the current alignment window in text
            # j is the index in pattern, from right to left (pattern_length - 1 down to 0)

            # Re-aligning the loop for clarity (standard Boyer-Moore)
            # s is the starting position of the pattern in the text
            s = 0
            while s <= text_length - pattern_length:
                j = pattern_length - 1 # Start comparing from the end of the pattern
                while j >= 0 and pattern[j] == text[s + j]:
                    j -= 1 # Matched, move left

                if j < 0: # Pattern found
                    return True
```

```
            else:
                # Mismatch at text[s+j]
                # Shift based on bad character rule:
                # Align text[s+j] with its last occurrence in pattern[0...j-1]
                # Or shift by j - last.get(text[s+j], -1)
                # If text[s+j] is not in pattern, last.get returns -1, shift is j - (-1) = j+1
                # If text[s+j] is pattern[x], shift is j-x
                # The shift value is how much 's' should increase.
                # The character in text that caused mismatch is text[s+j]
                # We want to align this character with its last occurrence in pattern.
                # If pattern[k] = text[s+j], we shift so s_new + k = s+j.
                # s_new = s + j - k. Shift amount = j-k.
                # last.get(text[s+j], -1) gives k.
                # So shift amount is j - last.get(text[s+j], -1)
                # However, the shift must be at least 1.
                # And the shift is applied to 's'.
                # shift = max(1, j - last.get(text[s+j], -1))
                # s += shift

                # Simpler: shift based on text[s + pattern_length - 1] if comparing from r
ight
                # Or text[s+j] if mismatch at j
                # Shift based on the character in text that is aligned with the rightmost c
har of pattern
                # This is text[s + pattern_length - 1]
                # shift_char_in_text = text[s + pattern_length - 1]
                # s += max(1, pattern_length - 1 - last.get(shift_char_in_text, -1))

                # Correct shift for bad character rule:
                # The character text[s+j] caused the mismatch.
                # We want to shift the pattern so that an occurrence of text[s+j] in the pa
ttern
                # aligns with text[s+j].
                # The last occurrence of text[s+j] in the pattern is at index last.get(text[s
+j], -1).
                # We want this index (k_pat = last.get(text[s+j], -1)) to align with j (inde
x in pattern).
                # So, shift by j - k_pat.
                # However, the shift is applied to s.
                # s_new = s + (j - last.get(text[s+j], -1))
                # But this is not standard. Standard: shift s by char_in_text_aligned_with
_pattern_end

                # Shift based on text[s + j] (the character that mismatched)
```

```
                # The position of this character in pattern is last.get(text[s+j], -1)
                # We want to shift s so that this character in pattern aligns with text[s+j]
                # s_new + last.get(text[s+j], -1) = s + j
                # s_new - s = j - last.get(text[s+j], -1)
                # shift = j - last.get(text[s+j], -1)
                # s += max(1, shift)

                # Standard bad character shift:
                # Shift the pattern so that the character text[s+j] aligns with its
                # rightmost occurrence in pattern[0...j-1].
                # If text[s+j] is not in pattern[0...j-1], shift pattern past text[s+j].
                # shift = j - last.get(text[s+j], -1)
                # s += max(1, shift)

                # Let's use the common implementation:
                # Shift based on the character in text aligned with the last char of pattern
                # This is text[s + pattern_length - 1]
                # If mismatch occurs at pattern[j] != text[s+j]
                # Shift is max(1, j - last.get(text[s+j], -1))

                # Shift based on text[s+j]
                char_shift = j - last.get(text[s+j], -1)
                s += max(1, char_shift)
            return False # Should be outside the while s loop if not found
        return False # If loop finishes without returning True
```

Д.6 Файл algorithms/kmp.py

```
from algorithms.base import BaseAlgorithm


class Algorithm(BaseAlgorithm):
    def search(self, pattern, text):
        """Knuth-Morris-Pratt string search algorithm"""
        m = len(pattern)
        n = len(text)

        if m == 0:
            return True
        if m > n:
            return False

        # Build LPS (Longest Proper Prefix which is also Suffix) array
```

```
        lps = [0] * m
        length = 0  # length of the previous longest prefix suffix
        i = 1
        while i < m:
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1

        # KMP search
        i = 0  # index for text
        j = 0  # index for pattern
        while i < n:
            if pattern[j] == text[i]:
                i += 1
                j += 1

            if j == m: # Pattern found
                return True
                # To find all occurrences:
                # print(f"Found pattern at index {i - j}")
                # j = lps[j - 1]
            elif i < n and pattern[j] != text[i]: # Mismatch after j matches
                if j != 0:
                    j = lps[j - 1]
                else:
                    i += 1
        return False
```

Д.7 Файл algorithms/naive.py

```
from algorithms.base import BaseAlgorithm


class Algorithm(BaseAlgorithm):
    def search(self, pattern, text):
        """Naive brute force string search algorithm"""
```

```
        m = len(pattern)
        n = len(text)

        # Edge cases
        if m == 0:
            return True # Empty pattern is found everywhere
        if m > n:
            return False # Pattern longer than text

        # Try all possible starting positions of pattern in text
        for i in range(n - m + 1): # Iterate from 0 to n-m
            j = 0
            # Check if pattern matches at current position text[i...i+m-1]
            while j < m and pattern[j] == text[i + j]:
                j += 1

            if j == m: # All characters of pattern matched
                return True

        return False  # Pattern not found after checking all positions
```

Д.8 Файл algorithms/rabin_karp.py

```
from algorithms.base import BaseAlgorithm


class Algorithm(BaseAlgorithm):
    def search(self, pattern, text):
        """Rabin-Karp string search algorithm"""
        m = len(pattern)
        n = len(text)

        if m == 0:
            return True
        if m > n:
            return False

        # Using Python's built-in hash for simplicity.
        # For a robust implementation, a custom rolling hash with a large prime is needed.
        pattern_hash = hash(pattern)

        for i in range(n - m + 1):
            current_slice = text[i: i + m]
```

```
            # In a real Rabin-Karp, we'd use a rolling hash.
            # Here, we re-hash each slice for simplicity, then compare strings if hashes matc
h.
            if hash(current_slice) == pattern_hash:
                if current_slice == pattern: # Verify to avoid collisions
                    return True
        return False
```

Д.9 Файл search_sources/base.py

```
class BaseSearchSource:
    """Base class for all search sources"""

    def get_texts(self):
        """
        Returns a list of all available texts from this source.
        Each text should be a tuple of (id, title, content)
        """
        raise NotImplementedError("Subclasses must implement get_texts method")

    def get_text_by_id(self, text_id):
        """Returns content of a specific text by its ID"""
        raise NotImplementedError("Subclasses must implement get_text_by_id method")

    def search(self, algorithm, pattern, thread_count=8):
        """
        Performs search using the provided algorithm
        Returns a result string and matching document IDs (list of integers or strings)
        """
        raise NotImplementedError("Subclasses must implement search method")
```

Д.10 Файл search_sources/db_source.py

```
from search_sources.base import BaseSearchSource
from database import get_session_factory, TextDocument
from multiprocessing import Pool
import time


class DatabaseSearchSource(BaseSearchSource):
    """Search implementation for PostgreSQL database"""

    def get_texts(self):
```

```
        """Get all available texts from database"""
        SessionLocal = get_session_factory()
        db = SessionLocal()
        try:
            # Fetch id, title, and content for all documents
            texts_data = db.query(TextDocument.id, TextDocument.title, TextDocument.co
ntent).all()
            # Convert to list of tuples: (id, title, content)
            texts = [(doc.id, doc.title, doc.content) for doc in texts_data]
            return texts
        finally:
            db.close()

    def get_text_by_id(self, text_id):
        """Get a specific text by ID from database"""
        SessionLocal = get_session_factory()
        db = SessionLocal()
        try:
            doc = db.query(TextDocument).filter(TextDocument.id == text_id).first()
            return doc.content.lower() if doc else ""
        finally:
            db.close()

    def _search_in_doc(self, doc_data, algorithm_class_name, pattern):
        """
        Search in a single document.
        algorithm_class_name is used to instantiate the algorithm in the child process.
        """
        # Dynamically import and instantiate the algorithm
        # This is necessary because algorithm instances might not be picklable
        # or to ensure each process has its own instance.
        # For simplicity, if algorithm_instance is picklable, it can be passed directly.
        # Here, we assume algorithm_instance is passed and is picklable.
        # If not, one would pass algorithm_class and instantiate it here.

        # Assuming algorithm_instance is passed directly if picklable
        # For multiprocessing, it's often better to pass data and re-create objects if they ar
e complex
        # or hold unpicklable state (like DB connections).
        # However, our algorithms are generally stateless or have simple state.

        # Let's assume 'algorithm' passed to search() is an instance.
        # If it's not picklable, this approach needs adjustment.
        # For now, we assume it is.
```

```
        doc_id, title, content = doc_data
        # The algorithm instance is passed directly to starmap, so it's available here.
        # We need to ensure the algorithm instance is picklable for multiprocessing.
        # Most of our algorithms should be.

        # Re-instantiate algorithm in each process to be safe
        # This requires passing the class or class name and then instantiating.
        # For now, let's assume the passed algorithm instance is used.
        # This part might need refinement based on pickling behavior of algorithm instanc
es.

        # If algorithm is an instance:
        result = algorithm_class_name.search(pattern.lower(), content.lower()) # Use the p
assed instance
        return (doc_id, title) if result else None

    def search(self, algorithm_instance, pattern, thread_count=8):
        """Perform parallel search in database texts"""
        texts_data_list = self.get_texts() # List of (id, title, content)

        if not texts_data_list:
            return "Не знайдено (база даних порожня або тексти відсутні)", []

        # Ensure thread count is reasonable
        thread_count = max(1, min(32, thread_count))

        # Prepare arguments for starmap: (doc_data_tuple, algorithm_instance, pattern_st
ring)
        # The algorithm_instance should be picklable.
        args_for_starmap = [(doc_data, algorithm_instance, pattern) for doc_data in texts_
data_list]

        with Pool(processes=thread_count) as pool:
            # Each process will call _search_in_doc with one item from args_for_starmap
            results = pool.starmap(self._search_in_doc, args_for_starmap)

        found_docs = [result for result in results if result is not None]

        if found_docs:
            doc_titles = [title for _, title in found_docs]
            doc_ids = [doc_id for doc_id, _ in found_docs] # Collect IDs
            return "Знайдено в текстах бази даних: " + ", ".join(doc_titles), doc_ids
```

```
        else:
            return "Не знайдено у базі даних", []
```

Д.11 Файл search_sources/file_source.py

```
import csv
import glob
import os
from search_sources.base import BaseSearchSource
from algorithms.parallel.base import ParallelAlgorithm # Assuming this is designed for file paths


class FileSearchSource(BaseSearchSource):
    """Search implementation for local text files"""

    def __init__(self):
        self.texts_dir = "texts"
        self.csv_file_path = "texts.csv"
        self.text_mappings = self._load_text_mappings()

    def _load_text_mappings(self):
        mappings = {}
        if not os.path.exists(self.csv_file_path):
            print(f"Warning: {self.csv_file_path} not found. File search may not work correctly.")
            return mappings

        with open(self.csv_file_path, mode="r", encoding="UTF-8") as inp:
            reader = csv.reader(inp, delimiter=";")
            for rows in reader:
                if len(rows) == 2:
                    # rows[0] is like "Text 0", rows[1] is "texts/0.txt"
                    # We want to extract the ID (0) from "Text 0" or use the filename part
                    try:
                        # text_id = int(rows[0].split(" ")[1]) # Assumes "Text <id>" format
                        # Or, derive ID from filename if consistent
                        file_id_str = os.path.splitext(os.path.basename(rows[1]))[0]
                        text_id = int(file_id_str)
                        mappings[text_id] = {"name": rows[0], "path": rows[1]}
                    except (IndexError, ValueError) as e:
                        print(f"Warning: Could not parse row in {self.csv_file_path}: {rows}. Error: {e}")
        return mappings
```

```
    def get_texts(self):
        """Get all available text files based on mappings"""
        texts = []
        for text_id, info in self.text_mappings.items():
            try:
                with open(info["path"], "r", encoding="utf-8") as f:
                    content = f.read()
                    texts.append((text_id, info["name"], content)) # (id, title/name, content)
            except FileNotFoundError:
                print(f"Warning: File not found {info['path']}")
            except Exception as e:
                print(f"Warning: Error reading file {info['path']}: {e}")
        return texts

    def get_text_by_id(self, text_id_str): # text_id is expected as string from ParallelAlgo
rithm
        """Get a specific text file by ID (e.g., "0", "1")"""
        try:
            text_id_int = int(text_id_str)
            if text_id_int in self.text_mappings:
                file_path = self.text_mappings[text_id_int]["path"]
                with open(file_path, "r", encoding="UTF-8") as file:
                    return file.read().lower()
            else:
                print(f"Warning: Text ID {text_id_str} not found in mappings.")
                return ""
        except ValueError:
            print(f"Warning: Invalid text ID format: {text_id_str}")
            return ""
        except FileNotFoundError:
            # This case should ideally be caught by mappings check, but good to have
            print(f"Warning: File for text ID {text_id_str} not found.")
            return ""


    def search(self, algorithm_instance, pattern, thread_count=8):
        """Perform parallel search in text files"""
        # ParallelAlgorithm expects the algorithm_instance to have a text_by_number met
hod
        # and to handle its own texts_names and result_list internally if needed by its find_
in_text_parallel

        # We need to adapt ParallelAlgorithm or how it's used here.
```

```
        # ParallelAlgorithm's find_in_text_parallel calls algorithm.text_by_number(str(tex
t_number))
        # So, the passed algorithm_instance needs that method. Our BaseAlgorithm has it.

        # The ParallelAlgorithm.find_in_text method loads texts.csv itself.
        # It then calls algorithm_instance.search() via its find_in_text_parallel.

        parallel_algo_wrapper = ParallelAlgorithm(algorithm_instance)
        # The find_in_text method of ParallelAlgorithm will use the text_by_number from
algorithm_instance
        # which in turn uses the file paths like "texts/0.txt".
        # It also populates its own texts_names from texts.csv.

        result_string = parallel_algo_wrapper.find_in_text(pattern, thread_count)

        matching_ids = []
        if "Не знайдено" not in result_string:
            # parallel_algo_wrapper.result_list contains file paths like "texts/0.txt"
            for file_path_in_result in parallel_algo_wrapper.result_list:
                try:
                    # Extract the ID (number part of the filename)
                    doc_id_str = os.path.splitext(os.path.basename(file_path_in_result))[0]
                    matching_ids.append(int(doc_id_str))
                except ValueError:
                    print(f"Warning: Could not parse ID from result path: {file_path_in_result}
")

        return result_string, matching_ids
```

Д.12 Файл templates/base.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.
0/css/all.min.css">
    <link rel="stylesheet" href="{{ url_for('static', path='/css/style.css') }}">
    <title>{% block title %}SearchMaster - Розширений Пошуковий Рушій{% endblo
ck %}</title>
    {% block extra_head %}{% endblock %}
</head>
<body>
```

```
<div class="app-container {% block container_class %}{% endblock %}">
  <header>
    <div class="logo">
      {% block logo %}
      <a href="{{ url_for('read_root') }}">
        <i class="fa-solid fa-magnifying-glass-chart"></i>
        <h1>SearchMaster</h1>
      </a>
      {% endblock %}
    </div>
  </header>

  <main>
    {% block search_form %}
    <div class="search-container {% block search_container_class %}{% endblock
%}">
      <form class="search-form" action="{{ url_for('search') }}" method="GET">
        <div class="search-box">
          <i class="fa-solid fa-magnifying-glass search-icon"></i>
          <input type="text" name="q" id="search-input" placeholder="Почніть
пошук..." autocomplete="off" value="{{ query|default('') }}">
          <button type="button" id="clear-search" class="clear-search-btn {% if n
ot query %}hidden{% endif %}">
            <i class="fa-solid fa-times"></i>
          </button>
        </div>

        <div class="search-options">
          <div class="option-group">
            <label for="search-source">Джерело Пошуку</label>
            <div class="custom-select">
              <select name="source" id="search-source">
                <option value="files" {% if current_source == 'files' or not curr
ent_source %}selected{% endif %}>Текстові Файли</option>
                <option value="database" {% if current_source == 'database' %
}selected{% endif %}>База Даних PostgreSQL</option>
                <option value="web" {% if current_source == 'web' %}selected
{% endif %}>Веб-Пошук</option>
              </select>
              <i class="fa-solid fa-chevron-down"></i>
            </div>
          </div>
          <div class="option-group">
            <label for="algorithm">Алгоритм</label>
```

```
                    <div class="custom-select">
                        <select name="algorithm" id="algorithm">
                            {% for key, name_val in algorithms.items() %}
                                <option value="{{ key }}" {% if key == current_algorithm %}selected{% endif %}>{{ name_val }}</option>
                            {% endfor %}
                        </select>
                        <i class="fa-solid fa-chevron-down"></i>
                    </div>
                </div>
                <div class="option-group">
                    <label for="threads">Кількість Потоків</label>
                    <div class="thread-selector">
                        <button type="button" class="thread-btn" id="thread-minus" aria-label="Зменшити кількість потоків">
                            <i class="fa-solid fa-minus"></i>
                        </button>
                        <input type="number" name="threads" id="threads" value="{{ thread_count|default(8) }}" min="1" max="32">
                        <button type="button" class="thread-btn" id="thread-plus" aria-label="Збільшити кількість потоків">
                            <i class="fa-solid fa-plus"></i>
                        </button>
                    </div>
                </div>
                <button type="submit" class="search-button">
                    <span>Пошук</span>
                    <i class="fa-solid fa-arrow-right"></i>
                </button>
            </div>
        </form>

        <ul class="suggestions" id="suggestions-list"></ul>

        {% block advanced_options %}{% endblock %}
    </div>
    {% endblock %}

    {% block content %}{% endblock %}
</main>

<footer>
    <p>&copy; 2025 SearchMaster - Розширений Текстовий Пошуковий Рушій. Студент Мишко І.Л., група 4151.</p>
```

```
      </footer>
    </div>

    <script src="{{ url_for('static', path='/js/script.js') }}"></script>
    {% block extra_scripts %}{% endblock %}
</body>
</html>
```

Д.13 Файл templates/index.html

```
{% extends "base.html" %} {% block title %}SearchMaster - Головна{% endblock %}
{% block advanced_options %}
<div class="advanced-options">
  <button id="compare-algorithms" type="button" class="compare-button">
    <i class="fa-solid fa-chart-line"></i>
    Порівняти Алгоритми
  </button>
  <div id="comparison-results" class="comparison-results">
    <!-- Результати порівняння будуть завантажені сюди -->
  </div>
</div>
{% endblock %} {% block content %}
<div class="features">
  <div class="feature">
    <i class="fa-solid fa-bolt"></i>
    <h3>Висока Продуктивність</h3>
    <p>Паралельне виконання для блискавичних результатів пошуку.</p>
  </div>
  <div class="feature">
    <i class="fa-solid fa-code-branch"></i>
    <h3>Різноманіття Алгоритмів</h3>
    <p>Обирайте з поміж різних алгоритмів пошуку рядків.</p>
  </div>
  <div class="feature">
    <i class="fa-solid fa-gauge-high"></i>
    <h3>Метрики в Реальному Часі</h3>
    <p>Переглядайте статистику продуктивності для кожного пошуку.</p>
  </div>
</div>
{% endblock %}
```

Д.14 Файл templates/search_result.html

```
{% extends "base.html" %} {% block title %}Результати Пошуку - {{ query }}{%
endblock %} {% block container_class %}results-page{% endblock %} {% block
search_form %} {# Включаємо форму пошуку і на сторінці результатів, але робим
о
її компактною #} {{ super() }} {% endblock %} {% block search_container_class
%}compact{% endblock %} {% block content %}
<div class="results-section">
  <div class="results-header">
    <h2>Результати пошуку для: "{{ query }}"</h2>
    <div class="performance-metrics">
      <div class="metric">
        <i class="fa-solid fa-clock"></i>
        <span>Час: {{ "%.6f"|format(execution_time) }} секунд</span>
      </div>
      <div class="metric">
        <i class="fa-solid fa-database"></i>
        <span>Джерело: {{ source_display_name }}</span>
      </div>
      <div class="metric">
        <i class="fa-solid fa-microchip"></i>
        <span>Алгоритм: {{ algorithm_display_name }}</span>
      </div>
      <div class="metric">
        <i class="fa-solid fa-layer-group"></i>
        <span>Потоки: {{ threads }}</span>
      </div>
    </div>
  </div>

  <div class="result-card">
    <div class="result-content">
      {% if source == 'web' %} {{ result|safe }} {% else %}
      <p>{{ result }}</p>
      {% endif %}
    </div>

    <div class="result-actions">
      <a href="{{ url_for('read_root') }}" id="new-search" class="button">
        <i class="fa-solid fa-arrow-left"></i>
        Новий Пошук
      </a>
      <button id="copy-results" type="button">
```

```
      <i class="fa-regular fa-clipboard"></i>
      Копіювати Результати
     </button>
    </div>
  </div>
</div>
{% endblock %} {% block extra_scripts %}
<script>
  // Скрипт для кнопки копіювання залишається в static/js/script.js
  // Переконуємося, що він правильно обробляє контент
</script>
{% endblock %}
```

Д.15 Файл database.py

```
import os
from sqlalchemy import create_engine, Column, Integer, String, Text
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
import psycopg2 # For initial database creation check

# Database connection parameters from environment variables or defaults
DB_NAME = os.environ.get("DB_NAME", "search_text_db")
DB_USER = os.environ.get("DB_USER", "postgres")
DB_PASSWORD = os.environ.get("DB_PASSWORD", "password") # Ensure this is s
ecure
DB_HOST = os.environ.get("DB_HOST", "localhost")
DB_PORT = os.environ.get("DB_PORT", "5432")

# Database URL for SQLAlchemy
DATABASE_URL = f"postgresql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{
DB_PORT}/{DB_NAME}"

# Base for declarative class definitions
Base = declarative_base()


# Define TextDocument model (table schema)
class TextDocument(Base):
    __tablename__ = "text_documents"

    id = Column(Integer, primary_key=True, index=True, autoincrement=True)
    title = Column(String, index=True, nullable=False)
```

```
    content = Column(Text, nullable=False)
    source_type = Column(String, default="database", nullable=False) # e.g., 'database', '
file_import'


def create_database_if_not_exists():
    """Creates the database in PostgreSQL if it doesn't already exist."""
    try:
        # Connect to the default 'postgres' database or any existing database to run CREA
TE DATABASE
        conn_params_no_db = {
            "user": DB_USER,
            "password": DB_PASSWORD,
            "host": DB_HOST,
            "port": DB_PORT,
            "dbname": "postgres" # Or your default maintenance DB
        }
        conn = psycopg2.connect(**conn_params_no_db)
        conn.autocommit = True # Required for CREATE DATABASE to work outside a tr
ansaction block
        cursor = conn.cursor()

        cursor.execute(f"SELECT 1 FROM pg_database WHERE datname = %s", (DB_N
AME,))
        exists = cursor.fetchone()

        if not exists:
            print(f"Database '{DB_NAME}' does not exist. Creating...")
            # It's important to ensure DB_NAME is properly sanitized if it were dynamic.
            # Here it's from config, so it's safer.
            cursor.execute(f"CREATE DATABASE {DB_NAME}") # Use SQL, not f-string if
possible for safety
            print(f"Database '{DB_NAME}' created successfully.")
        else:
            print(f"Database '{DB_NAME}' already exists.")

        cursor.close()
        conn.close()
    except psycopg2.Error as e:
        print(f"Error connecting to PostgreSQL or creating database: {e}")
        # Depending on the error, you might want to exit or handle differently
        # For example, if DB_USER/DB_PASSWORD is wrong, table creation will also fai
l.
    except Exception as e:
```

```
        print(f"An unexpected error occurred during database creation check: {e}")


# SQLAlchemy engine
_engine = None
def get_engine():
    global _engine
    if _engine is None:
        _engine = create_engine(DATABASE_URL)
    return _engine

# SQLAlchemy session factory
_SessionFactory = None
def get_session_factory():
    global _SessionFactory
    if _SessionFactory is None:
        _SessionFactory = sessionmaker(autocommit=False, autoflush=False, bind=get_en
gine())
    return _SessionFactory


def init_db():
    """Initializes the database: creates the database if not exists, then creates tables."""
    print("Initializing database...")
    create_database_if_not_exists() # Step 1: Ensure database exists

    # Step 2: Create tables using SQLAlchemy metadata
    # This connects to the specific DB_NAME which should now exist.
    try:
        engine = get_engine()
        print(f"Attempting to create tables in database '{DB_NAME}'...")
        Base.metadata.create_all(bind=engine)
        print("Database tables checked/created successfully.")
    except Exception as e:
        # This might happen if the DB_NAME couldn't be connected to,
        # e.g. if create_database_if_not_exists failed silently or permissions are wrong.
        print(f"Error creating database tables with SQLAlchemy: {e}")
        print("Please ensure the database exists and connection parameters are correct.")
        # raise # Optionally re-raise to halt application startup


# Dependency for FastAPI to get a DB session
def get_db():
    SessionLocal = get_session_factory()
```

```
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# If you run this file directly, it can try to initialize the DB
if __name__ == "__main__":
    print("Running database initialization directly...")
    init_db()
    print("Database initialization process finished.")
```

Д.16 Файл main.py

```
import time
import glob
import threading
import uvicorn
import os

from pathlib import Path

from fastapi import FastAPI, Request, Depends, HTTPException
from fastapi.templating import Jinja2Templates
from fastapi.responses import JSONResponse, HTMLResponse
from fastapi.staticfiles import StaticFiles
from sqlalchemy.orm import Session

from algorithms import (
    BoyerMoore, RabinKarp, KMP, Bitap, Sunday,
    Naive, Horspool, BMH, TwoWay, Simon,
)
# Assuming ParallelAlgorithm is correctly imported if used directly by main
# from algorithms.parallel.base import ParallelAlgorithm
from search_sources import get_search_source, FileSearchSource, DatabaseSearchSou
rce, WebSearchSource
from database import get_db, init_db # init_db for startup

app = FastAPI(title="SearchMaster API", version="1.0.0")

# Mount static files (CSS, JS)
app.mount(
    "/static",
```

```
    StaticFiles(directory=Path(__file__).parent.absolute() / "static"),
    name="static",
)

# Setup Jinja2 templates
templates_dir = Path(__file__).parent.absolute() / "templates"
templates = Jinja2Templates(directory=str(templates_dir))

# Algorithm mapping (key to class and display name)
ALGORITHMS_CONFIG = {
    "boyer_moore": {"class": BoyerMoore, "name": "Boyer-Moore"},
    "rabin_karp": {"class": RabinKarp, "name": "Rabin-Karp"},
    "kmp": {"class": KMP, "name": "Knuth-Morris-Pratt"},
    "bitap": {"class": Bitap, "name": "Bitap"},
    "sunday": {"class": Sunday, "name": "Sunday"},
    "naive": {"class": Naive, "name": "Brute Force (Naive)"},
    "horspool": {"class": Horspool, "name": "Horspool"},
    "bmh": {"class": BMH, "name": "Boyer-Moore-Horspool"},
    "two_way": {"class": TwoWay, "name": "Two-Way"},
    "simon": {"class": Simon, "name": "Simon"},
}

# Search source display names
SOURCE_DISPLAY_NAMES = {
    "files": "Текстові Файли",
    "database": "База Даних PostgreSQL",
    "web": "Веб-Пошук",
}

# Global cache for suggestions (simple implementation)
suggestions_cache = []
SUGGESTIONS_UPDATE_INTERVAL = 1200  # seconds (20 minutes)

def update_suggestions_cache():
    """Updates the global suggestions cache from file sources."""
    global suggestions_cache
    temp_suggestions = set()

    # Using FileSearchSource to get text content for suggestions
    # This assumes FileSearchSource().get_texts() is efficient enough or can be adapted
    try:
        file_source_instance = FileSearchSource() # Create an instance
        # get_texts returns list of (id, name, content)
        all_texts_data = file_source_instance.get_texts()
```

```
        for _, _, content in all_texts_data:
            words = content.lower().split()
            # Add words longer than 3 chars, limit per text to avoid bias
            temp_suggestions.update(w for w in words if len(w) > 3 and w.isalpha())
            if len(temp_suggestions) > 5000: # Overall limit for cache size
                break
    except Exception as e:
        print(f"Error updating suggestions cache: {e}")

    suggestions_cache = sorted(list(temp_suggestions))
    print(f"Suggestions cache updated with {len(suggestions_cache)} unique words.")

    # Schedule next update
    threading.Timer(SUGGESTIONS_UPDATE_INTERVAL, update_suggestions_cach
e).start()


@app.on_event("startup")
async def startup_event():
    print("Application startup...")
    try:
        init_db() # Ensure database and tables are created
        print("Database initialization complete.")
        # Initial population of suggestions cache
        # Run in a separate thread to not block startup
        threading.Thread(target=update_suggestions_cache, daemon=True).start()
    except Exception as e:
        print(f"Error during startup initialization: {e}")
        # Depending on severity, you might want to stop the app
        # For now, just log it.


@app.get("/", response_class=HTMLResponse)
async def read_root(request: Request):
    """Serves the main search page."""
    return templates.TemplateResponse(
        "index.html",
        {
            "request": request,
            "algorithms": {k: v["name"] for k, v in ALGORITHMS_CONFIG.items()},
            "sources": SOURCE_DISPLAY_NAMES, # Pass display names for sources
            "current_algorithm": "boyer_moore", # Default selected algorithm
            "current_source": "files", # Default selected source
```

```
            "thread_count": 8, # Default thread count
        },
    )

@app.get("/item", response_class=HTMLResponse)
async def search_item(
    request: Request,
    q: str,
    algorithm: str = "boyer_moore",
    threads: int = 8,
    source: str = "files",
    db: Session = Depends(get_db) # Inject DB session if needed by a source
):
    """Handles search requests and displays results."""
    if not q:
        # Redirect to home or show an error if query is empty
        return templates.TemplateResponse("index.html", {"request": request, "error": "П
ошуковий запит не може бути порожнім.", "algorithms": {k: v["name"] for k, v in A
LGORITHMS_CONFIG.items()}, "sources": SOURCE_DISPLAY_NAMES, "current_
algorithm": algorithm, "current_source": source, "thread_count": threads})

    algo_config = ALGORITHMS_CONFIG.get(algorithm)
    if not algo_config:
        raise HTTPException(status_code=400, detail="Invalid algorithm selected")

    algo_instance = algo_config["class"]()

    # Ensure threads are within a sane range
    thread_count = max(1, min(32, threads))

    search_source_instance = get_search_source(source, db_session=db) # Pass db sessio
n
    if not search_source_instance:
        raise HTTPException(status_code=400, detail="Invalid search source selected")

    start_time = time.time()
    result_text, found_ids = search_source_instance.search(algo_instance, q, thread_coun
t)
    execution_time = time.time() - start_time

    return templates.TemplateResponse(
        "search_result.html",
        {
            "request": request,
```

```
            "query": q,
            "result": result_text, # This might be HTML for web search
            "execution_time": execution_time,
            "algorithm_display_name": algo_config["name"],
            "source_display_name": SOURCE_DISPLAY_NAMES.get(source, "Невідоме
Джерело"),
            "threads": thread_count,
            # For search form on results page:
            "algorithms": {k: v["name"] for k, v in ALGORITHMS_CONFIG.items()},
            "sources": SOURCE_DISPLAY_NAMES,
            "current_algorithm": algorithm,
            "current_source": source,
            "source": source, # for conditional rendering in template if needed
        },
    )

@app.get("/search", response_class=JSONResponse)
async def get_search_suggestions(q: str):
    """Provides search suggestions based on the cached word list."""
    if not q or len(q) < 2: # Minimum query length for suggestions
        return []

    query_lower = q.lower()
    # Simple substring matching for suggestions
    # More advanced would use prefix matching, Levenshtein distance, etc.
    matched_suggestions = [s for s in suggestions_cache if query_lower in s]

    return matched_suggestions[:10] # Return top 10 matches


@app.get("/performance", response_class=JSONResponse)
async def get_algorithm_performance(
    q: str,
    algorithm: str,
    threads: int = 8,
    source: str = "files",
    db: Session = Depends(get_db) # Inject DB session
):
    """Endpoint to test performance of a single algorithm."""
    if not q:
        raise HTTPException(status_code=400, detail="Query parameter 'q' is required.")

    algo_config = ALGORITHMS_CONFIG.get(algorithm)
    if not algo_config:
```

```
        raise HTTPException(status_code=400, detail=f"Invalid algorithm: {algorithm}")

    algo_instance = algo_config["class"]()
    thread_count = max(1, min(32, threads))
    search_source_instance = get_search_source(source, db_session=db)

    if not search_source_instance:
        raise HTTPException(status_code=400, detail=f"Invalid source: {source}")

    start_time = time.time()
    result_text, found_ids = search_source_instance.search(algo_instance, q, thread_coun
t)
    execution_time = time.time() - start_time

    return {
        "algorithm": algorithm,
        "display_name": algo_config["name"],
        "source": source,
        "source_display_name": SOURCE_DISPLAY_NAMES.get(source, "Unknown"),
        "execution_time": execution_time,
        "threads": thread_count,
        "has_results": "Не знайдено" not in result_text if result_text else False,
        "result_count": len(found_ids) if found_ids else 0,
    }

# To run the application: uvicorn main:app --reload
if __name__ == "__main__":
    # This part is for development only. For production, use Gunicorn with Uvicorn work
ers.
    print("Starting Uvicorn server for development...")
    uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)
```