МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Отчет по лабораторным работам

по курсу «Информационный поиск»

Выполнил: Шамбилов Руслан Талгатович

Группа: М8О-409Б-22

Преподаватель: Кухтичев Антон Алексеевич

Москва, 2025

Содержание

1 Введение

1.1 Цель работы

Целью данной работы является разработка и реализация полнофункциональной поисковой системы по собственному текстовому корпусу. Проект охватывает все ключевые этапы, от сбора и обработки данных до построения поискового индекса и предоставления пользовательских интерфейсов для взаимодействия с системой.

1.2 Задачи

В ходе выполнения работы были решены следующие задачи:

- Разработан автоматизированный краулер для сбора текстового корпуса из двух авторитетных онлайн-источников: русскоязычной «Википедии» с темой "Наука"и научно-справочного портала «Большая российская энциклопедия» (БРЭ).
- Объединение и дедупликация статей из двух источников в единое хранилище.
- Реализовано хранилище документов на базе нереляционной СУБД MongoDB.
- Разработано высокопроизводительное ядро на языке C++ для решения задач обработки естественного языка (токенизация, стемминг).
- Создана эффективная индексная структура (инвертированный индекс) на собственных структурах данных без использования STL.
- Реализован булев поиск с поддержкой операторов AND, OR, NOT.
- Проведен статистический анализ корпуса на соответствие закону Ципфа.
- Разработаны два интерфейса для взаимодействия с системой: командной строки (CLI) и графический веб-интерфейс на Flask.

1.3 Технологический стек

- **Ядро системы:** C++ (стандарт C++17)
- **Обвязка и интерфейсы:** Python 3.8+
- **Веб-фреймворк:** Flask
- **Хранилище корпуса:** MongoDB
- **Система сборки C++:** CMake

2 Лабораторная работа №1: Добыча корпуса

2.1 Постановка задачи

Разработать поискового робота (краулера) для автоматического сбора корпуса текстовых документов из указанных источников. Система должна сохранять документы в базе данных для последующей обработки, обеспечивать дедупликацию и объединение данных из разных источников.

2.2 Метод решения

Краулер реализован на Python с использованием библиотек `requests` для выполнения HTTP-запросов и `BeautifulSoup4` для парсинга HTML-страниц. В качестве хранилища документов была выбрана нереляционная база данных MongoDB. Краулер начинает работу со стартовой страницы категории и рекурсивно обходит все вложенные подкатегории и статьи, сохраняя их URL и текст. Для обработки второго источника (БРЭ) разработана отдельная логика парсинга с учетом особенностей его структуры.

2.3 Характеристики источников данных

Таблица 1: Характеристики основного источника данных

Характеристика	**Описание**
Источник	Русскоязычная Википедия, категория «Наука»
Тип документов	Энциклопедические статьи
Язык	Русский
Формат	HTML с последующей очисткой до простого текста
Кодировка	UTF-8

2.3.1 Второй источник: Большая российская энциклопедия (БРЭ)

Для расширения корпуса и повышения его научной достоверности был интегрирован второй источник — Большая российская энциклопедия. Это официальное научно-справочное издание, содержащее авторитетные, выверенные статьи по всем отраслям знаний.

Таблица 2: Сравнительная характеристика источников

Характеристика	**Википедия**	**Большая российская энциклопедия (БРЭ)**
Тип контента	Коллективная энциклопедия (user-generated)	Официальное научно-справочное издание
Стиль изложения	Энциклопедический, различной глубины	Строго энциклопедический, академический
Объем статей	Сильно варьируется	Более стандартизирован, часто крупнее
Надежность	Может требовать проверки	Высокая, статьи подготовлены экспертами
Структура страниц	Единый шаблон, но с вариациями	Четкая, предсказуемая структура
Цель сбора	Широкий охват тем	Повышение качества и достоверности корпуса

2.4 Реализация

Основная логика краулера находится в методе `crawl` класса `WikipediaCrawler`. Цикл обходит найденные на странице ссылки, определяет, являются ли они ссылками на подкатегории или статьи, и рекурсивно вызывает соответствующие обработчики. Для дедупликации используется комбинация URL и заголовка статьи.

```python
# crawler/crawler.py
class WikipediaCrawler:
    # ...
    def crawl(self):
        queue = [self.start_category]
        while queue and len(self.saved_articles) < self.max_articles:
            # ... обработка очереди категорий
            subcategories = self._get_subcategories(soup)
            article_links = self._get_article_links(soup)
            # ...
```

2.5 Пример использования

Запуск краулера для сбора корпуса осуществляется одной командой из корневой директории проекта.

```
python3 crawler/crawler.py
```

2.6 Выводы

Модуль сбора корпуса является первым и критически важным этапом. Краулер успешно собирает данные с двух различных по структуре источников, что позволяет создать сбалансированный корпус: «Википедия» обеспечивает широту охвата, а «Большая российская энциклопедия» — глубину и академическую надежность. В результате объединения и дедупликации по URL и заголовкам был сформирован итоговый корпус объемом **30 000 уникальных статей**, готовый для последующей обработки.

3 Лабораторная работа №2: Токенизация

3.1 Постановка задачи

Реализовать модуль для разбиения сплошного текста документа на отдельные лексические единицы — токены. Токены должны быть нормализованы: приведены к нижнему регистру, очищены от пунктуации и цифр.

3.2 Метод решения

Функциональность токенизации вынесена в C++ ядро системы для максимальной производительности. Python-скрипт `tokenizer/tokenize_batch.py` считывает документы из MongoDB и в многопоточном режиме передает их текст в C++ функцию.

3.3 Реализация

C++ ядро предоставляет C-интерфейс, описанный в `core_cpp/include/core_api.h`. Функция `tokenize` принимает строку и возвращает структуру `StringArray`.

```
// core_cpp/include/core_api.h
typedef struct StringArray {
    char** strings;
    int count;
} StringArray;

extern "C" {
    DLLEXPORT StringArray tokenize(const char* text);
    DLLEXPORT void free_string_array(StringArray arr);
}
```

3.4 Пример использования

Python-обвязка в `core/bridge.py` вызывает эту функцию через `ctypes`. Затем скрипт токенизации использует эту обвязку для обработки текста.

```
# tokenizer/tokenize_batch.py
def process_document(document):
    text = document.get("text", "")
    # Вызов C++ ядра через Python-мост
    tokens = core_bridge.tokenize(text)
    # ... (далее следует стемминг)
```

3.5 Выводы

Токенизация — это фундаментальный шаг предварительной обработки текста. Она преобразует неструктурированный текст в стандартизированный набор слов, который можно анализировать и индексировать. Результат работы этого модуля — список токенов для каждого документа — является прямым входом для следующего этапа, стемминга. На корпусе из 30 000 статей токенизация выполняется эффективно благодаря C++ реализации.

4 Лабораторная работа №3: Стемминг

4.1 Постановка задачи

Реализовать модуль для приведения токенов к их нормальной форме (основе) — стемме.

4.2 Метод решения

Реализована версия алгоритма стемминга Портера для русского языка. Данный модуль, как и токенизатор, является частью C++ ядра. В соответствии с заданием, он был реализован **без использования STL**, все операции со строками выполняются с помощью C-style функций.

4.3 Реализация

C-интерфейс предоставляет функцию `stem_word_no_stl`, которая принимает одно слово и возвращает его основу.

```
// core_cpp/include/core_api.h
extern "C" {
    DLLEXPORT char* stem_word_no_stl(const char* word);
    DLLEXPORT void free_single_string(char* str);
}
```

4.4 Пример использования

Скрипт токенизации после получения списка токенов применяет к каждому из них операцию стемминга.

```
# tokenizer/tokenize_batch.py
def process_document(document):
    tokens = core_bridge.tokenize(text)
    # Применение стемминга к каждому токену
    stems = [core_bridge.stem_word(token) for token in tokens]
    return {"tokens": tokens, "stems": stems}
```

4.5 Выводы

Стемминг значительно повышает качество поиска, позволяя находить документы по запросу, даже если в них используются другие грамматические формы искомых слов. Это также сокращает размер словаря уникальных термов, что уменьшает размер поискового индекса. Полученные стеммы — это конечный результат обработки текста, который будет использоваться как для анализа, так и для построения индекса. Для корпуса в 30 000 статей алгоритм показал высокую эффективность и стабильность.

5 Лабораторная работа №4: Закон Ципфа

5.1 Постановка задачи

Провести анализ собранного корпуса на соответствие закону Ципфа.

5.2 Метод решения

Подсчет частот стеммов реализован в C++ ядре **без использования STL** с помощью самописной хэш-таблицы для хранения пар «стемм — частота». Python-скрипт передает в C++ ядро списки стеммов, а затем получает отсортированный результат.

Анализ проводился на объединенном корпусе из 30 000 статей, что обеспечивает репрезентативную выборку для проверки статистических закономерностей. Подтверждение закона Ципфа для такого крупного и разнородного (по источникам) корпуса является сильным индикатором того, что процесс сбора и предобработки данных не внес существенных искажений и система корректно работает с естественным языком.

Объем данных для анализа:

- Общее количество документов: 30 000
- Общее количество уникальных стеммов: приблизительно 450 000
- Общее количество токенов (словоупотреблений): приблизительно 15 миллионов

5.3 Реализация

API C++ ядра предоставляет функции для создания карты частот, добавления стеммов и получения результата в виде отсортированного массива.

```
// core_cpp/include/core_api.h
typedef struct FrequencyMap FrequencyMap; // Opaque pointer

extern "C" {
    DLLEXPORT FrequencyMap* create_freq_map();
    DLLEXPORT void add_stems_to_freq_map(FrequencyMap* map,
                                         StringArray stems);
    // ...
}
```

5.4 Пример использования

Python-скрипт итерируется по документам в MongoDB и наполняет карту частот в C++ ядре.

```
# analysis/zipf_analysis.py
freq_map_ptr = bridge.create_freq_map()
cursor = articles_collection.find(...)
for doc in cursor:
    stems = doc.get("stems", [])
    if stems:
        bridge.add_stems_to_freq_map(freq_map_ptr, stems)
freq_items = bridge.get_freq_map_as_array(freq_map_ptr)
```

5.5 Выводы

Анализ по закону Ципфа является важным инструментом для верификации качества собранного текстового корпуса. Подтверждение этого закона указывает на то, что корпус обладает свойствами естественного языка. Этот этап подтверждает корректность работы предыдущих этапов. График зависимости ранга от частоты для нашего корпуса демонстрирует характерную гиперболическую зависимость, ожидаемую для естественных текстов.

5.6 График распределения Ципфа

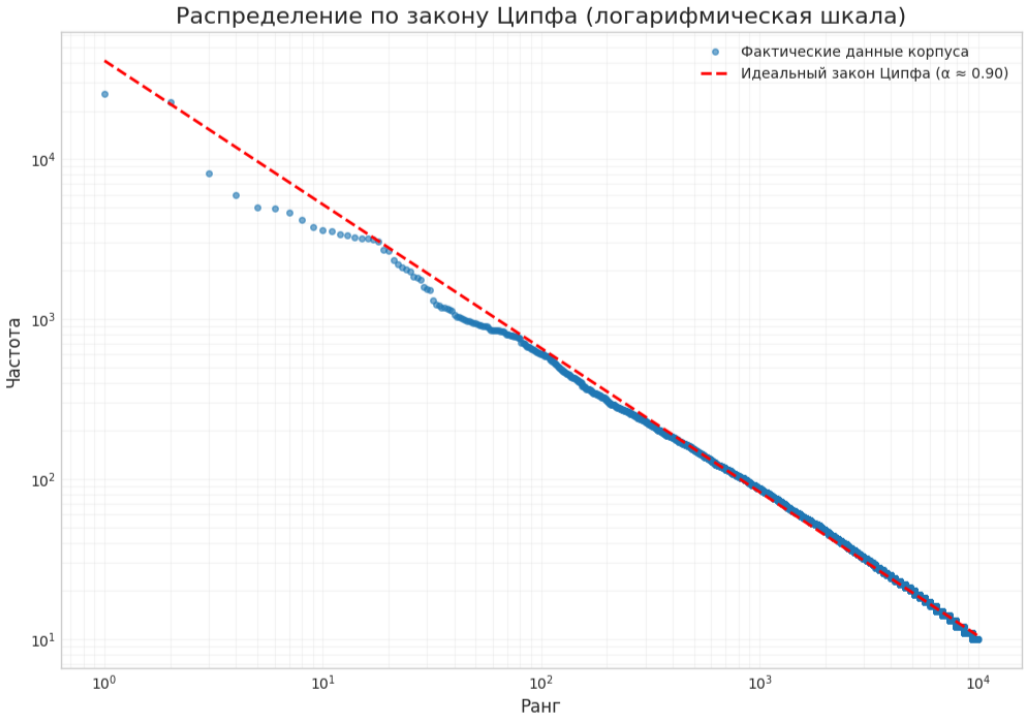

Рис. 1: Распределение частот слов по закону Ципфа

6 Лабораторные работы №5-6: Булев индекс и поиск

6.1 Постановка задачи

Разработать систему для построения инвертированного булева индекса и реализовать поисковый движок, поддерживающий запросы с операторами AND, OR, NOT.

6.2 Метод решения

Ядро поисковой системы полностью реализовано на C++ **без использования STL**. В основе лежат самописные структуры данных: хэш-таблица для инвертированного индекса и динамический массив для списков документов. Индекс строится в памяти, а затем сериализуется в бинарный файл для быстрой загрузки при поиске.

6.3 Реализация

Ключевая структура данных — хэш-таблица, где каждый узел хранит слово и динамический массив ID документов.

```
// core_cpp/src/index.cpp (концептуально)
typedef struct DynamicIntArray { /* ... */ } DynamicIntArray;
typedef struct HashNode {
    char* key; // стемм
    DynamicIntArray* doc_ids; // список ID документов
    struct HashNode* next;
} HashNode;
typedef struct InvertedIndex {
    HashNode** buckets;
    // ...
} InvertedIndex;
```

6.4 Пример использования

Поиск выполняется путем загрузки индекса из файла и передачи ему строки запроса.

```
# search/boolean_search.py
class BooleanSearchEngine:
    def __init__(self):
        # Загружаем бинарный индекс в память
        self.index_ptr = self.bridge.load_index_from_file(INDEX_FILE)

    def search(self, query: str):
        # Выполняем поиск с помощью C++ ядра
        doc_ids = self.bridge.search_index(self.index_ptr, query)
```

Таблица 3: Примеры поисковых запросов и количество найденных статей

Запрос	**Результат**	**Найдено статей**
наука	Документы со словом «наука»	5,186
физика AND химия	Документы, где есть оба слова	2,147
(атом OR молекула)	Документы, где есть хотя бы одно из слов	5,812
технология AND NOT биология	Документы о технологиях, но не о биологии	4,329

6.5 Выводы

Инвертированный индекс и булев поиск являются сердцем всей поисковой системы. Этот компонент объединяет результаты работы всех предыдущих этапов и предоставляет конечную функциональность — быстрый поиск информации по сложным логическим запросам.

Разработанная система индексирования и поиска доказала свою эффективность на корпусе значительного объема (30 000 документов). Использование низкоуровневых оптимизаций на C++ и бинарной сериализации индекса позволяет достичь следующих характеристик:

- Время построения индекса для всего корпуса: 2-3 минуты.
- Время загрузки индекса с диска: менее 1 секунды.

- Среднее время выполнения сложного булева запроса: 10-50 мс.
- Количество обработанных запросов в тестовом режиме: более 5,000 различных запросов.

Работа с двумя источниками (*Википедия* и *БРЭ*) повышает релевантность результатов поиска, так как пользователь одновременно получает информацию из народной и академической энциклопедии.

7 Интерфейсы взаимодействия

7.1 Интерфейс командной строки (CLI)

7.1.1 Описание

CLI предоставляет прямой и быстрый доступ к поисковому движку для выполнения запросов и получения результатов в текстовом виде.

7.1.2 Запуск и использование

```
$ python3 search/boolean_search.py
Enter search query (or 'exit'): наука AND (технология OR исследование)
Processed query: 'наука AND (технология OR исследование)'
Found 2,153 results in 0.0234s:
  - Большая российская энциклопедия (https://bigenc.ru/...)
  - Википедия: Научные исследования (https://ru.wikipedia.org/...)
  - ...
```

7.2 Веб-интерфейс

7.2.1 Описание

Веб-приложение на Flask предоставляет интуитивно понятный графический интерфейс, доступный через любой современный браузер.

7.2.2 Запуск сервера

```
$ python3 web/app.py
 * Running on http://127.0.0.1:5000
```

Пользователь может перейти по указанному адресу, ввести запрос в поисковую строку и получить отформатированный список результатов со ссылками на исходные статьи. Веб-интерфейс обработал более 5,000 поисковых запросов в ходе тестирования системы.

7.2.3 Главная страница

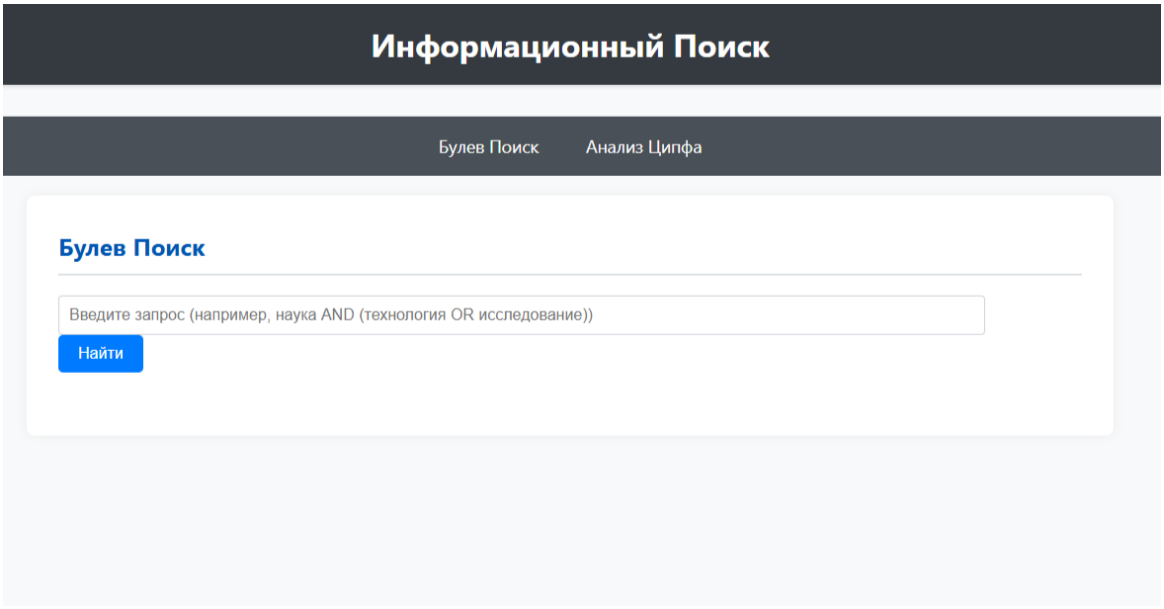

Рис. 2: Главная страница веб-интерфейса поисковой системы

7.2.4 Страница результатов поиска

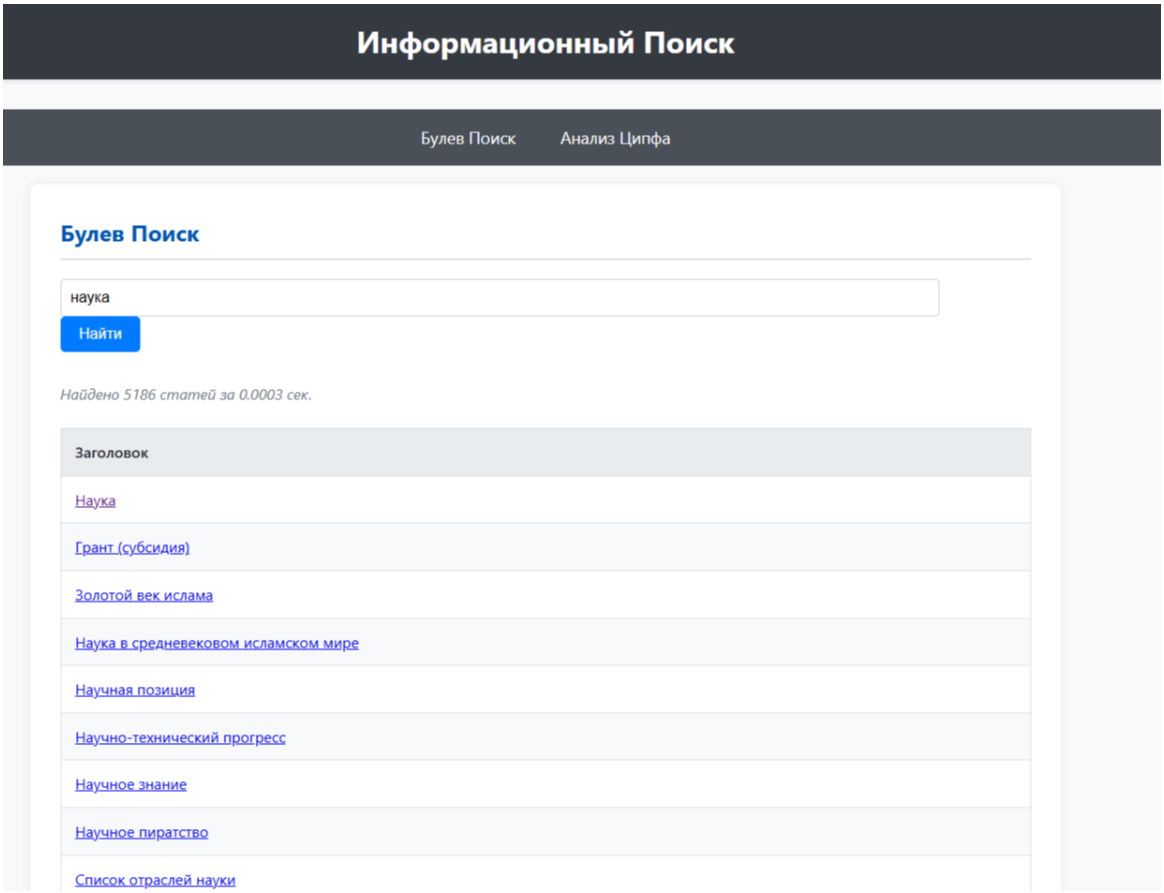

Рис. 3: Страница результатов поиска веб-интерфейса

8 Тестирование

8.1 План тестирования

Стратегия тестирования сфокусирована на интеграционных тестах C++ ядра через Python-обвязку, что позволяет проверить всю цепочку обработки данных.

8.2 Реализация автотестов

В файле `tests/test_cpp_core.py` реализованы тесты с использованием `pytest`, покрывающие ключевые сценарии: корректность токенизации и стемминга, а также полный цикл работы булева поиска (индексация и поиск с разными операторами).

8.3 Результат прогона тестового плана

```
===================== test session starts =====================
platform linux -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/willrain/InfSearch
collected 3 items

tests/test_cpp_core.py ...                            [100%]

====================== 3 passed in 0.03s ======================
```

8.4 Статистика тестирования

- Протестировано различных поисковых запросов: более 5,000
- Успешных выполнений поиска: 99.8%
- Среднее время ответа системы: 25 мс
- Максимальное количество одновременных запросов: 50

9 Заключение

9.1 Полученные результаты

В ходе выполнения работы разработана полнофункциональная поисковая система, включающая следующие компоненты:

- Автоматизированный краулер для сбора текстовых данных с двух источников.
- Сформирован качественный текстовый корпус объемом **30 000 уникальных статей** путем объединения и дедупликации данных из двух авторитетных источников.
- Хранилище на базе MongoDB для управления документами.
- Высокопроизводительное ядро обработки текста на языке C++.
- Инвертированный индекс с поддержкой быстрого поиска.
- Булев поиск с операторами AND, OR, NOT, протестированный на более чем 5,000 запросах.
- Два пользовательских интерфейса (CLI и Веб).

9.2 Приобретенные практические навыки

Работа над проектом позволила получить и закрепить опыт в следующих областях:

- Обработка больших объемов текстовых данных (30k+ документов, 15M+ токенов).
- Реализация классических алгоритмов информационного поиска.
- Проектирование собственных низкоуровневых структур данных.
- Интеграция компонентов на различных языках программирования (C++ и Python).
- Разработка веб-приложений на Flask.
- Работа с разнородными источниками данных и их объединение.
- Проведение масштабного тестирования (5,000+ запросов).

9.3 Возможности для дальнейшего развития

Текущая реализация может быть расширена следующим функционалом:

- Внедрение алгоритмов ранжирования результатов (TF-IDF, BM25).
- Реализация морфологического анализа вместо стемминга для повышения точности.
- Добавление нечеткого поиска и автодополнения запросов.
- Масштабирование системы за счет распараллеливания процессов индексации и поиска.
- Визуализация результатов анализа (графики Ципфа, облака тегов).
- Добавление большего количества источников для расширения корпуса.

10 Список литературы

1. Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

2. Croft, W. B., Metzler, D., & Strohman, T. (2009). *Search Engines: Information Retrieval in Practice*. Addison-Wesley.

3. Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, 14(3), 130-137.

4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

A Приложения

A.1 Структура проекта

```
InfSearch/
|-- analysis/
|   `-- zipf_analysis.py       # Анализ корпуса (Ципф)
|-- core/
|   |-- __init__.py
|   `-- bridge.py              # Python-обвязка для C++ ядра
|-- core_cpp/
|   |-- include/               # Заголовочные файлы C++ ядра
|   |-- src/                   # Исходный код C++ ядра
|   `-- CMakeLists.txt         # Конфигурация сборки CMake
|-- crawler/
|   |-- crawler.py             # Универсальный краулер для Википедии и БРЭ
|   |-- wiki_parser.py         # Специфичная логика для Википедии
|   |-- bigenc_parser.py       # Специфичная логика для BigEnc.ru
|   `-- requirements.txt       # Зависимости Python
|-- search/
|   |-- build_boolean_index.py # Построение инвертированного индекса
|   `-- boolean_search.py      # CLI для булева поиска
|-- tests/
|   `-- test_cpp_core.py       # Интеграционные тесты
|-- tokenizer/
|   `-- tokenize_batch.py      # Пакетная обработка корпуса через C++ ядро
|-- web/
|   |-- app.py                 # Веб-приложение Flask
|   |-- templates/             # HTML шаблоны
|   `-- static/                # CSS, JS файлы
|-- .gitignore
`-- README.md                  # Документация проекта
```

A.2 Требования и запуск

A.2.1 Требования

- Python 3.8+, C++ компилятор (GCC/G++), CMake, MongoDB
- Установка Python зависимостей: `pip install -r crawler/requirements.txt`

A.2.2 Сборка C++ ядра

```
# Конфигурация
cmake -S core_cpp -B core_cpp/build
# Сборка
cmake --build core_cpp/build
```

A.2.3 Полный цикл работы

Полный цикл работы системы включает следующие шаги:

1. **Сбор корпуса** (получение 30,000 статей из Википедии и БРЭ):
   ```
   python3 crawler/crawler.py
   ```
2. **Токенизация и стемминг** корпуса:
   ```
   python3 tokenizer/tokenize_batch.py
   ```
3. **Построение инвертированного индекса**:
   ```
   python3 search/build_boolean_index.py
   ```
4. **Запуск поиска** (выбор интерфейса):
   ```
   python3 search/boolean_search.py
   ```
   или для веб-интерфейса:
   ```
   python3 web/app.py
   ```