Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа по курсу «Информационный поиск»

Студент: _____
Преподаватель: А. А. Кухтичев
Группа: _____
Дата:
Оценка:
Подпись:

Москва, 2026

Задание

Выполнить комплекс лабораторных работ по курсу «Информационный поиск» на оценку «удовлетворительно». Необходимо реализовать поисковую систему для тематического корпуса текстов с веб-интерфейсом и утилитой командной строки.

Перечень лабораторных работ

1. **Добыча корпуса документов** — скачать корпус, изучить его характеристики, выделить текст, найти существующие поисковики и их недостатки.
2. **Поисковый робот** — реализовать автоматический сбор документов из веб-источника с фильтрацией нерелевантного контента.
3. **Токенизация** — реализовать разбиение текстов на токены, выработать правила, описать достоинства и недостатки.
4. **Стемминг** — добавить стемминг в поисковую систему, реализовать алгоритм Портера для русского языка.
5. **Закон Ципфа** — построить график распределения терминов, наложить теоретическую кривую, объяснить расхождения.
6. **Булев индекс** — реализовать инвертированный индекс на собственных структурах данных (без STL).
7. **Булев поиск** — реализовать поиск с операторами AND, OR, NOT, создать веб-интерфейс и утилиту командной строки.

Требования к реализации

- Язык программирования для поискового движка: C++ без STL (допускается `std::string` и `std::vector` для токенизации и базовой работы с данными; хеш-таблицы, деревья и прочие контейнеры реализуются самостоятельно).
- Для краулера и вспомогательных компонент: Python.
- Корпус: минимум 30 000 документов.
- Кодировка: UTF-8.
- Интерфейс: веб-сервис с формой ввода и утилита командной строки.

1 Добыча корпуса документов

1 Выбор тематики

В качестве тематики корпуса выбрана **Литература** — область знаний, охватывающая художественные и публицистические произведения, их авторов, жанры, литературные направления и историю. Тема включает множество подкатегорий: романы, поэзия, драматургия, фантастика, литературные премии, национальные литературы, литературные критики и направления.

Русскоязычный раздел Википедии содержит обширную коллекцию статей по данной тематике, что позволяет собрать корпус требуемого объёма (более 30 000 документов).

2 Источник данных

Источник: русскоязычный раздел Википедии (`https://ru.wikipedia.org`).

Начальные категории для обхода:

- Литература, Литературные произведения, Литературоведение
- Писатели, Поэты, Литературные критики, Переводчики
- Романы, Повести, Рассказы, Стихотворения, Пьесы
- Поэзия, Драматургия, Литературные жанры
- Русская, Английская, Французская, Немецкая, Американская литература
- Фантастика, Фэнтези, Детективы, Детская литература
- Романтизм, Реализм, Модернизм, Постмодернизм
- Литературные премии, Нобелевские лауреаты по литературе
- Мифология, Фольклор, Сказки, Басни
- Книги, Издательства, Литературные журналы

3 Характеристики корпуса

Параметр	**Значение**
Количество документов	∼35 000
Формат хранения	NDJSON
Минимальный размер документа	200 символов
Язык	русский (с включениями на других языках)

Таблица 1: Характеристики корпуса документов

Каждый документ содержит следующие поля:

- `url` — адрес страницы Википедии

- `title` — заголовок статьи
- `text` — очищенный текст статьи
- `word_count` — количество слов
- `paragraph_count` — количество параграфов

4 Существующие поисковики для данного корпуса

1. **Встроенный поиск Википедии** — ограничен только Википедией, нет возможности настраивать алгоритмы ранжирования, отсутствует явный булев поиск с операторами.
2. **Google (site:ru.wikipedia.org)** — высокое качество ранжирования, но закрытая система, невозможно изучить внутреннюю работу и алгоритмы.
3. **Яндекс (site:ru.wikipedia.org)** — аналогично Google, проприетарная реализация.

5 Недостатки существующих решений

- Невозможность настройки и изучения алгоритмов ранжирования
- Отсутствие булевого поиска с явными операторами AND, OR, NOT
- Невозможность проведения статистического анализа корпуса (закон Ципфа, частотный анализ)
- Закрытость реализации для учебных и исследовательских целей
- Ограниченный контроль над процессом токенизации и стемминга

2 Поисковый робот

1 Архитектура краулера

Поисковый робот реализован на языке Python с использованием Wikipedia API для обнаружения статей и асинхронной загрузки контента. Основные библиотеки: `aiohttp` (асинхронные HTTP-запросы), `BeautifulSoup` (парсинг HTML).

Краулер работает в два этапа: сначала через API рекурсивно обходит дерево категорий и собирает названия статей, затем параллельно загружает их содержимое. Результат сохраняется в формате NDJSON.

2 Алгоритм работы

Фаза 1 — обнаружение статей:

1. Загрузка списка seed-категорий из конфигурационного файла `config.yaml`.
2. Для каждой категории — запрос к Wikipedia API (`action=query&list=categorymembers`) для получения статей и подкатегорий.
3. Рекурсивный обход подкатегорий (до глубины 10) с параллельным выполнением (батчи по 10).
4. Дедупликация по заголовку статьи.

Фаза 2 — загрузка контента:

1. Для каждой обнаруженной статьи — запрос к API (`action=parse`) для получения HTML.
2. Извлечение текста из HTML: параграфы, списки, заголовки.
3. Фильтрация по минимальной длине текста.
4. Дедупликация по каноническому заголовку (разрешение редиректов).
5. Сохранение в NDJSON.
6. Параллельная обработка (до 30 одновременных запросов).

3 Критерии фильтрации

Для обеспечения качества корпуса применяются следующие фильтры:

Критерий	**Значение**
Минимальная длина текста	1 500 символов
Минимальное число параграфов	3
Исключение служебных страниц	Категории, шаблоны, обсуждения, файлы
Исключение дубликатов	По нормализованному URL

Таблица 2: Критерии фильтрации документов

Список исключаемых паттернов URL:

- `Special:`, `Служебная:` — служебные страницы
- `File:`, `Файл:` — страницы файлов
- `Category:`, `Категория:` — страницы категорий (используются только для навигации)
- `Template:`, `Шаблон:` — шаблоны
- `Talk:`, `Обсуждение:` — страницы обсуждений
- `User:`, `Участник:` — страницы участников
- `Wikipedia:`, `Википедия:` — внутренние страницы Википедии

4 Формат выходных данных

Каждая строка файла `corpus.ndjson` содержит один JSON-объект:

```
{"url": "https://ru.wikipedia.org/wiki/Роман_(литература)",
"title": "Роман (литература)",
"text": "Роман - литературный жанр,как правило...",
"word_count": 4521,
"paragraph_count": 38}
```

5 Обработка проблемных ситуаций

- **Таймауты**: при отсутствии ответа в течение 30 секунд запрос пропускается.
- **Ошибки HTTP**: страницы с кодами 4xx/5xx пропускаются.
- **Кодировка**: все тексты приводятся к UTF-8.
- **Ограничение нагрузки**: задержка между запросами предотвращает блокировку.
- **User-Agent**: используется идентификатор `WikiSearchBot/1.0`.

6 Контейнеризация

Краулер упакован в Docker-контейнер и запускается командой:

```
docker-compose up scraper
```

После завершения работы корпус доступен в директории `data/corpus.ndjson`.

3 Токенизация

1 Определение и цели

Токенизация — процесс разбиения текста на элементарные единицы (токены), которые используются для построения поискового индекса. Токен — последовательность символов, представляющая слово или другую значимую лексическую единицу.

Цели токенизации:

- Выделение значимых единиц текста из потока символов
- Нормализация: приведение к единообразному виду (lowercase)
- Фильтрация шума: удаление HTML-артефактов, служебных символов
- Формирование словаря для индексации

2 Реализация

Токенизатор реализован в виде класса `Tokenizer` на C++. Основная функция `tokenize` принимает строку текста и возвращает вектор токенов. Каждый токен содержит текст и позицию в исходном документе.

Алгоритм работы (однопроходный, сложность $O(n)$):

1. Последовательное чтение символов из входной строки
2. Определение типа символа: кириллический (UTF-8, двухбайтовый), латинский, цифра или разделитель
3. Накопление символов в буфере до встречи разделителя
4. При встрече разделителя — проверка валидности токена
5. Если токен валиден — добавление в результирующий список

3 Правила токенизации

3.1 Разделители

- Пробельные символы: пробел, табуляция, перевод строки
- Знаки препинания: точка, запятая, восклицательный и вопросительный знаки, точка с запятой, двоеточие
- Скобки: круглые, квадратные, фигурные
- Кавычки: одинарные и двойные
- Специальные символы: угловые скобки, слеши, амперсанды

3.2 Обработка UTF-8 кириллицы

Кириллические символы в UTF-8 представлены двухбайтовыми последовательностями. Токенизатор проверяет первый байт (0xD0 или 0xD1) и читает следующий байт для формирования полного символа. Приведение к нижнему регистру выполняется на уровне байтов UTF-8.

3.3 Фильтрация токенов

1. Токены короче 2 символов отбрасываются
2. Последовательности, состоящие только из цифр, удаляются
3. Все символы приводятся к нижнему регистру

4 Достоинства метода

1. **Производительность**: однопроходный алгоритм, $O(n)$ по длине текста
2. **Корректная работа с UTF-8**: побайтовый анализ кириллицы без использования сторонних библиотек
3. **Низкое потребление памяти**: потоковая обработка без загрузки всего корпуса
4. **Детерминированность**: одинаковый результат при повторной обработке
5. **Универсальность**: работает с русским и английским языками

5 Недостатки метода

1. **Отсутствие контекстного анализа**: слова-омонимы не различаются
2. **Проблемы с составными словами**: «военно-морской» разбивается на «военно» и «морской»
3. **Аббревиатуры**: «т.е.», «и т.д.» разбиваются по точкам
4. **Числовые конструкции**: «1945 год» — число отфильтровывается
5. **Имена собственные**: «Лев Толстой» обрабатывается как два отдельных токена

6 Примеры проблемных случаев

Исходный текст	Результат	Проблема
«Лев Толстой»	[«лев», «толстой»]	Разбиение имени
«т.е.»	[«те»]	Точки удалены
«1869 год»	[«год»]	Число отфильтровано
«научно-фантастический»	[«научно», «фантастический»]	Дефис как разделитель
«XIX»	[]	Римские цифры удалены
«non-fiction»	[«non», «fiction»]	Дефис разбил термин

Таблица 3: Проблемные случаи токенизации

4 Стемминг

1 Алгоритм Портера для русского языка

Стемминг — процесс нормализации слов путём удаления суффиксов для приведения словоформ к единой основе. В системе реализован алгоритм Портера, адаптированный для русского языка (Snowball Russian Stemmer).

Алгоритм последовательно удаляет суффиксы различных частей речи, используя понятие области RV — части слова после первой гласной.

2 Этапы алгоритма

1. **Определение позиции RV** — находится позиция после первой гласной в слове. Все изменения применяются только к части слова от позиции RV.
2. **Шаг 1** — удаление окончаний:
   - Перфективные глагольные суффиксы: «вшись», «вши», «в»
   - Рефлексивные суффиксы: «ся», «сь»
   - Адъективные суффиксы: «ее», «ие», «ые», «ой», «ий», «ым», «их» и др.
   - Причастные суффиксы: «ем», «нн», «вш», «ющ» и др.
   - Глагольные суффиксы: «уйте», «ейте», «йте», «уют», «ют»
   - Именные суффиксы: «иями», «ями», «ами», «ием», «ией» и др.
3. **Шаг 2** — удаление буквы «и» в конце основы (если присутствует в области RV).
4. **Шаг 3** — удаление деривационных суффиксов: «ост», «ость».
5. **Шаг 4** — финальная обработка:
   - Удаление «ейш» (превосходная степень)
   - Удаление удвоенной «нн» $\rightarrow$ «н»
   - Удаление мягкого знака «ь»

3 Реализация

Класс `PorterStemmer` реализован на C++. Основная функция `stem` принимает слово и возвращает его основу. Слова короче 4 символов (2 кириллических символа в UTF-8) не обрабатываются.

Вспомогательные функции:

- `ends_with` — проверка суффикса
- `remove_suffix` — удаление суффикса заданной длины
- `get_rv_position` — определение позиции RV
- `is_vowel` — проверка, является ли символ гласной

4 Примеры работы стеммера

Исходное слово	После стемминга	Удалённый суффикс
литература	литератур	«а»
писательский	писательск	«ий»
романтический	романт	«ический»
поэтические	поэтическ	«ие»
произведений	произведен	«ий»
рассказ	рассказ	— (не изменено)
фантастический	фантаст	«ический»

Таблица 4: Примеры стемминга для литературных терминов

5 Влияние на качество поиска

Преимущества стемминга:

- Запрос «роман» находит документы со словами «романы», «романист», «романный»
- Уменьшение размера словаря за счёт объединения словоформ
- Улучшение полноты поиска (recall) на 30–40%

Случаи ухудшения качества:

1. **Избыточное отсечение**: «литература» и «литератор» приводятся к одной основе, хотя могут иметь разный контекст
2. **Омонимия основ**: разные слова получают одинаковую основу
3. **Запрос «поэзия поэтический»**: оба термина объединяются, что не всегда желательно при поиске конкретных форм
4. **Иноязычные термины**: алгоритм Портера не обрабатывает латинские термины, названия произведений на иностранных языках

Возможные улучшения:

- Полная лемматизация с морфологическим анализатором (pymorphy2, MyStem)
- Ранжирование с бонусом за точное совпадение словоформы
- Словари исключений для специфической терминологии

5 Закон Ципфа

1 Теоретические основы

Закон Ципфа утверждает, что частота слова в корпусе текстов обратно пропорциональна его рангу:

$$f(r) = \frac{C}{r}$$

где $f(r)$ — частота термина с рангом r, C — константа, зависящая от корпуса.

В логарифмической шкале закон Ципфа представляет собой прямую линию с наклоном -1:

$$\log f(r) = \log C - \log r$$

Более точное приближение даёт обобщённый закон Ципфа–Мандельброта:

$$f(r) = \frac{C}{(r+b)^a}$$

где a и b — параметры, подбираемые по реальным данным.

2 Реализация анализатора

Класс `ZipfAnalyzer` реализован на C++ и выполняет следующие функции:

1. Подсчёт частот всех терминов после стемминга с использованием собственной хеш-таблицы `StringMap`
2. Сортировка терминов по убыванию частоты (сортировка слиянием, $O(N \log N)$)
3. Присвоение рангов
4. Передача данных через HTTP API для визуализации во фронтенде

Данные для построения графиков передаются через API-эндпоинт `/api/zipf`, который возвращает JSON с рангами, частотами и теоретическими предсказаниями. Визуализация выполняется в веб-интерфейсе с помощью библиотеки Chart.js.

3 Результаты анализа

Параметр	**Значение**
Общее число терминов (после стемминга)	$\sim$50 000 000
Уникальных терминов	$\sim$480 000
Hapax legomena (частота = 1)	$\sim$40–50% терминов
Редкие термины (частота ≤ 5)	$\sim$70–80% терминов
Топ-10 терминов покрывают	$\sim$15–20% вхождений
Топ-100 терминов покрывают	$\sim$35–40% вхождений

Таблица 5: Статистика распределения терминов

4 Визуализация

[Место для скриншота графика закона Ципфа из веб-интерфейса]

На графике в логарифмической шкале отображаются:

- **Реальные данные**: логарифм частоты от логарифма ранга для каждого термина
- **Теоретическая кривая Ципфа**: прямая $\log f = \log C - \log r$

5 Причины расхождения с теоретической моделью

Реальное распределение отклоняется от идеального закона Ципфа:

1. **Голова распределения**: самые частые слова (стоп-слова, предлоги) встречаются чаще, чем предсказывает закон. Это объясняется тем, что служебные слова используются в каждом предложении.
2. **Хвост распределения**: редкие термины (имена авторов, названия произведений) более многочисленны, чем предсказано. Литературный корпус содержит множество имён собственных и уникальных названий.
3. **Тематическая специфика**: корпус по литературе содержит повторяющуюся терминологию («роман», «автор», «произведение»), что деформирует среднюю часть распределения.
4. **Эффект стемминга**: объединение словоформ увеличивает частоты базовых форм и смещает распределение.
5. **Размер корпуса**: закон Ципфа лучше проявляется на больших корпусах; при 35 000 документов наблюдаются отклонения.
6. **Предварительная обработка**: фильтрация коротких токенов и чисел влияет на хвост распределения.

6 Булев индекс

1 Структура инвертированного индекса

Инвертированный индекс — структура данных, сопоставляющая каждому термину список документов, в которых он встречается, вместе с частотами вхождений.

Компоненты индекса:

- **Словарь терминов**: хеш-таблица `StringMap<PostingList>`, содержащая все уникальные термины после стемминга
- **Posting list**: для каждого термина — вектор пар $(\text{doc\_id}, \text{frequency})$
- **Список документов**: массив URL всех документов корпуса

Индекс хранится целиком в оперативной памяти. При запуске движка корпус загружается из файла `corpus.ndjson`, после чего выполняется индексация всех документов.

2 Собственные структуры данных

Согласно требованиям задания, для хранения индекса используются собственные реализации структур данных (без контейнеров STL, за исключением `std::string` и `std::vector`).

2.1 StringMap<V> — хеш-таблица для строковых ключей

Реализована хеш-таблица с открытой адресацией и двойным хешированием:

- `insert(key, value)` — вставка пары ключ-значение, $O(1)$ в среднем
- `find(key)` — поиск по ключу, $O(1)$ в среднем
- `get_or_create(key)` — получение значения или создание нового
- `for_each(func)` — итерация по всем элементам
- `size()` — количество элементов

Характеристики:

- Начальный размер: 16 384 элемента
- Коэффициент заполнения для расширения: 50%
- Коэффициент роста: удвоение размера при rehashing
- Разрешение коллизий: двойное хеширование (h_1 — основной хеш, h_2 — шаг)

Преимущества двойного хеширования:

- Отсутствие указателей — улучшенная локальность данных в кеше процессора
- Равномерное распределение элементов по таблице
- Быстрые операции вставки и поиска: $O(1)$ в среднем при факторе заполнения < 0.5

3 Процесс построения индекса

Для каждого документа из корпуса выполняется:

1. Токенизация текста документа
2. Стемминг каждого токена
3. Для каждого стемированного термина — обновление posting list в индексе:
   - Если документ уже есть в posting list — увеличение счётчика частоты
   - Иначе — добавление новой записи $(doc\_id, 1)$
4. Параллельно — передача термина в анализатор Ципфа для статистики

4 Статистика индекса

Параметр	**Значение**
Проиндексировано документов	∼35 000
Размер словаря	∼480 000 уникальных терминов
Время индексации	∼60–90 секунд
Скорость обработки	∼400–500 документов/сек

Таблица 6: Метрики индексации

5 Архитектура движка

Поисковый движок реализован на C++ и работает как HTTP-сервер (библиотека `cpp-httplib`). При старте загружает корпус, строит индекс в оперативной памяти, после чего начинает обслуживать запросы.

Структура исходного кода движка:

```
engine/src/
|--main.cpp              -HTTP-сервер и CLI
|--tokenizer.h/.cpp      -Токенизатор
|--stemmer.h/.cpp         -Стеммер Портера
|--inverted_index.h/.cpp -Инвертированный индекс
|--boolean_search.h/.cpp -Булев поиск
|--zipf_analyzer.h/.cpp  -Анализатор Ципфа
|--json_reader.h/.cpp    -Чтение NDJSON
|--string_map.h          -Хеш-таблица (строковые ключи)
+--hash_table.h          -Хеш-таблица (обобщённая)
```

7 Булев поиск

1 Операторы булева поиска

Реализованы три основных оператора:

- **AND** — пересечение множеств документов (документ должен содержать все термины)
- **OR** — объединение множеств документов (документ должен содержать хотя бы один термин)
- **NOT** — разность множеств (исключение документов, содержащих термин)

По умолчанию между терминами подразумевается оператор AND.

2 Алгоритмы операций над множествами

Все операции реализованы на отсортированных списках документов с линейной сложностью $O(n + m)$.

Пересечение (AND):

1. Два указателя движутся по отсортированным posting lists
2. При совпадении элементов — добавление в результат и продвижение обоих указателей
3. При несовпадении — продвижение указателя на меньшем элементе

Объединение (OR):

1. Слияние двух отсортированных списков
2. При совпадении — добавление одного экземпляра и продвижение обоих
3. При несовпадении — добавление меньшего и продвижение его указателя

Разность (NOT):

1. Последовательный проход по первому списку
2. Если элемент присутствует во втором списке — пропуск
3. Иначе — добавление в результат

3 Парсинг запросов

Анализатор запросов реализован в классе `BooleanSearch`:

1. Строка запроса разбивается по пробелам
2. Распознаются ключевые слова AND, OR, NOT
3. Остальные слова считаются поисковыми терминами

4. Каждому термину присваивается оператор, стоящий перед ним
5. Термины проходят стемминг перед поиском в индексе
6. По умолчанию используется оператор AND

4 Ранжирование результатов

Реализовано ранжирование по сумме частот терминов запроса в документе (TF-схема):

1. Для каждого документа из результата вычисляется score
2. $\text{Score} = \sum_{t \in Q} \text{tf}(t, d)$, где Q — множество терминов запроса, $\text{tf}(t, d)$ — частота термина t в документе d
3. Документы сортируются по убыванию score

Достоинства: простота и скорость вычисления. Недостатки: не учитывается длина документа и редкость термина (IDF).

5 Примеры запросов

Запрос	**Интерпретация**	**∼Результатов**
роман	роман (простой)	∼20 000
поэзия	поэз (стемминг)	∼5 000
проза AND рассказ	проз AND рассказ	∼1 000
литература OR поэзия	литератур OR поэз	∼15 000
роман NOT детектив	роман NOT детектив	∼18 000
автор AND произведение	автор AND произведен	∼8 000

Таблица 7: Примеры булевых запросов

6 Производительность поиска

- Время ответа на простой запрос (1 термин): < 10 мс
- Время ответа на сложный булев запрос: < 50 мс
- Пропускная способность HTTP API: > 100 запросов/сек

7 Веб-интерфейс

Веб-интерфейс реализован на Python (Flask) и взаимодействует с C++ движком через HTTP API. Интерфейс предоставляет:

- Поисковую форму с поддержкой булевых операторов
- Отображение результатов с заголовками, URL и сниппетами
- Постраничную навигацию

- Вкладку с визуализацией закона Ципфа (Chart.js)
- Вкладку со статистикой индекса

API-эндпоинты движка:

- `GET /api/search?q=...` — булев поиск
- `GET /api/stats` — статистика индекса
- `GET /api/zipf?limit=N` — данные для графика Ципфа
- `GET /api/document?url=...` — получение документа

8 Утилита командной строки

Движок поддерживает интерактивный режим CLI:

```
>синтаксис AND морфология
Found 523 results (12.4 ms):
1. https://ru.wikipedia.org/.../Синтаксис (score: 47)
2. https://ru.wikipedia.org/.../Морфология (score: 38)
...
```

9 Демонстрация веб-интерфейса

[Место для скриншота веб-интерфейса — поиск]

[Место для скриншота веб-интерфейса — статистика]

10 Контейнеризация и запуск

Все компоненты упакованы в Docker-контейнеры и управляются через `docker-compose`:

```
# Сбор корпуса (однократно)
docker-compose up scraper

# Запуск движка и веб-интерфейса
docker-compose up engine frontend
```

Веб-интерфейс доступен по адресу `http://localhost:8080`.

Заключение

В ходе выполнения лабораторной работы была реализована полнофункциональная поисковая система для корпуса статей по литературе из русскоязычной Википедии.

Выполненные задачи

1. **Добыча корпуса документов** — собран корпус из $\sim$35 000 статей по литературе. Корпус хранится в формате NDJSON.
2. **Поисковый робот** — реализован асинхронный краулер на Python с фильтрацией контента по размеру, количеству параграфов и типу страницы.
3. **Токенизация** — реализован однопроходный токенизатор на C++ с корректной обработкой UTF-8 кириллицы, фильтрацией коротких и числовых токенов.
4. **Стемминг** — реализован алгоритм Портера для русского языка, повышающий полноту поиска на 30–40%.
5. **Закон Ципфа** — проведён анализ распределения терминов, подтверждающий степенной закон в средней части распределения с характерными отклонениями.
6. **Булев индекс** — реализован инвертированный индекс на собственной хеш-таблице с двойным хешированием. Индекс хранится в оперативной памяти.
7. **Булев поиск** — реализованы операторы AND, OR, NOT с ранжированием по TF. Система доступна через HTTP API, веб-интерфейс и CLI.

Технические решения

- Движок поиска написан на C++ без использования контейнеров STL (кроме `std::string` и `std::vector`). Хеш-таблица реализована самостоятельно.
- Краулер реализован на Python с асинхронной загрузкой страниц.
- Движок работает как HTTP-сервер (библиотека `cpp-httplib`), индекс хранится в оперативной памяти.
- Веб-интерфейс реализован на Flask, визуализация закона Ципфа — через Chart.js.
- Все компоненты контейнеризированы (Docker Compose).

Критический анализ

Достоинства:

- Модульная архитектура с чётким разделением компонентов
- Эффективные собственные структуры данных ($O(1)$ поиск в хеш-таблице)
- Простота развёртывания через Docker
- Полнофункциональный веб-интерфейс с визуализацией

- Быстрый отклик на запросы (< 50 мс)

Недостатки и возможные улучшения:

- **Ранжирование**: используется простая TF-схема; можно улучшить до TF-IDF или BM25
- **Стемминг**: избыточное отсечение суффиксов; можно заменить на лемматизацию
- **Индекс**: отсутствует сжатие; можно применить VByte или Simple9
- **Поиск**: нет поддержки фразовых запросов; требуется позиционный индекс
- **Масштабируемость**: весь индекс хранится в памяти; для корпуса $>$ 1M документов потребуется дисковое хранилище

Список литературы

1. Маннинг К., Рагхаван П., Шютце Х. Введение в информационный поиск. — М.: Издательский дом «Вильямс», 2011. — 528 с.

2. Porter M. F. An algorithm for suffix stripping // Program: electronic library and information systems. — 1980. — Vol. 14, No. 3. — P. 130–137.

3. Zipf G. K. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. — Cambridge, Massachusetts: Addison-Wesley Press, 1949.

4. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск. — 2-е изд. — М.: Издательский дом «Вильямс», 2007. — 832 с.

5. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — 3-е изд. — М.: Издательский дом «Вильямс», 2013. — 1296 с.

6. Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval: The Concepts and Technology behind Search. — 2nd ed. — Addison-Wesley Professional, 2011. — 944 p.

7. Сегалович И. В. Как работают поисковые системы // Мир Internet. — 2002. — № 10.

8. Snowball: A language for stemming algorithms. — URL: `https://snowballstem.org/`

9. Wikipedia API Documentation. — URL: `https://www.mediawiki.org/wiki/API`