Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: Головенко А. В.

Группа: М8О-412Б-22

Преподаватель: Кухтичев А. А.

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2025

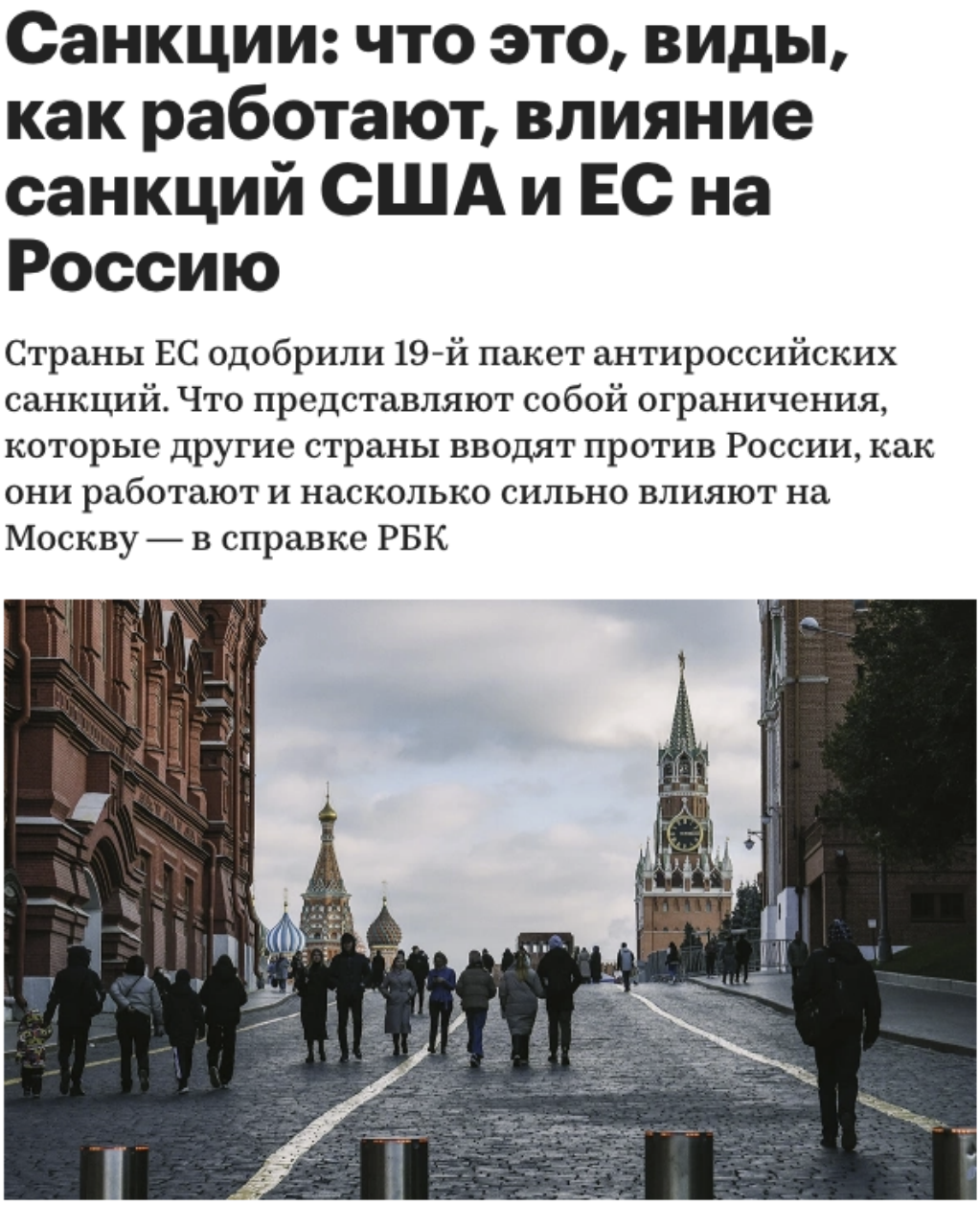
Содержание

Цель работы

- Составить корпус документов
- Произвести обкачку документов
- Произвести очистку документов: выделить текст статей
- Произвести токенизацию текстов
- Составить булев индекс и реализовать булев поиск по документам
- Составить обратный индекс для TF-IDF и реализовать поиск с использованием TF-IDF для ранжирования

Описание данных

В качестве темы для корпуса документов были выбраны источники связанные с мировыми новостями и политикой: rbc.ru и dw.com.



Санкции: что это, виды, как работают, влияние санкций США и ЕС на Россию

Страны ЕС одобрили 19-й пакет антироссийских санкций. Что представляют собой ограничения, которые другие страны вводят против России, как они работают и насколько сильно влияют на Москву — в справке РБК

Рис. 1: Пример статьи на rbc.ru

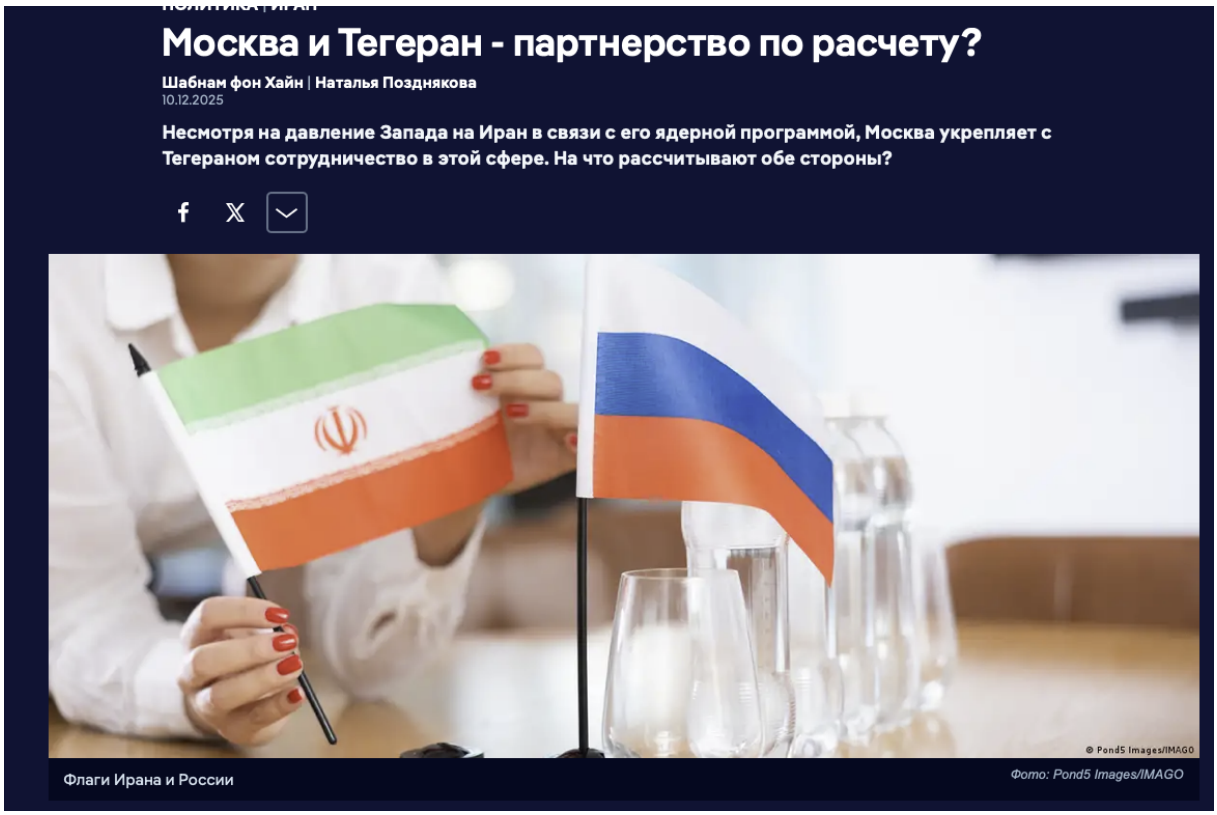


Рис. 2: Пример статьи на dw.com

Для составления корпуса документов был написан вспомогательный скрипт на языке Python, который проходит по sitemap.xml обоих ресурсов и составляет список url, которые я потом буду скачивать в базу данных. В результате составления корпуса документов был получен корпус размером 99654 документов.

Обкачка документов

Для сбора текстового корпуса был разработан специализированный асинхронный краулер (crawler) на языке Python. Архитектура приложения построена на базе библиотеки `asyncio`, что позволяет эффективно утилизировать сетевые ресурсы при большом количестве одновременных соединений.

Конфигурация системы задается через YAML-файл, определяющий следующие группы параметров:

- **Параметры БД:** хост, порт и имя коллекции MongoDB;
- **Логика обхода:** уровень конкурентности (concurrency), задержка между запросами (politeness delay), целевое количество документов и User-Agent;
- **Источники (Sources):** список корневых URL и разрешенных доменов (allow-list) для каждого новостного ресурса (в данном случае RBC и DW).

Для хранения данных о документах и управления очередью задач используется СУБД MongoDB (с асинхронным драйвером `motor`). В отличие от решений с внешними брокерами сообщений (например, Redis), в данной реализации очередь задач интегрирована непосредственно в коллекцию документов через систему статусов (`pending`, `done`, `error`) и временных меток (`next_check`). Это позволило упростить архитектуру и развертывание приложения.

Для получения HTML-страниц используется библиотека `aiohttp`. Парсинг и извлечение ссылок осуществляются с помощью `BeautifulSoup` (lxml). В базу данных записываются следующие данные: нормализованный URL, исходный HTML-код, MD5-хеш контента, название источника и время скачивания.

Ключевые особенности реализованного алгоритма:

1. **Bootstrap (Холодный старт).** Реализован механизм автоматического поиска точек входа. При старте краулер анализирует `robots.txt`,

карты сайта (`sitemap.xml`) и RSS-ленты источников для быстрого наполнения очереди актуальными ссылками.

2. **Дедупликация.** Перед сохранением вычисляется MD5-хеш контента (`content_hash`). Если при повторном заходе хеш совпадает с сохраненным в базе, тело страницы не перезаписывается, что экономит дисковое пространство и ресурсы БД.

3. **Балансировка нагрузки.** Реализован алгоритм «честной выборки» (fair fetching). При запросе новой пачки задач из БД ссылки выбираются равномерно от каждого источника, чтобы предотвратить «забивание» канала одним быстрым сайтом.

4. **Фильтрация.** Применяется строгая валидация доменов и игнорирование бинарных файлов (.pdf, .jpg, .zip и т.д.) на этапе извлечения ссылок.

Целевой объем базы данных ограничен параметром конфигурации `max_documents` (100 000 документов). При достижении этого лимита сбор новых ссылок прекращается.

Парсинг и токенизация страниц

Парсинг и токенизация осуществлялись на языке C++. Взаимодействие с базой данных MongoDB было реализовано через класс-обертку `MongoConnector`, использующий официальный драйвер `mongocxx`. Процесс обработки документов состоит из последовательных этапов: извлечение чистого текста, токенизация и лемматизация.

Для разбора HTML-страниц был разработан класс `HtmlParser`. В его основе лежит библиотека **Google Gumbo**, которая строит DOM-дерево из исходного HTML-кода. Метод `getCleanText` рекурсивно обходит узлы дерева, извлекая содержимое текстовых узлов (`GUMBO_NODE_TEXT`) и игнорируя служебные теги, такие как `<script>` и `<style>`. Это позволяет получить чистый текст статьи без технического мусора.

Полученный текст передается в класс `Tokenizer`. Особенностью реализации является работа с кодировками: исходная строка UTF-8 преобразуется в UTF-32 (`std::wstring`) для корректной посимвольной обработки кириллицы. В процессе прохода по строке выполняется:

- **Приведение к нижнему регистру:** Реализовано вручную путем смещения кодов символов для латиницы и кириллицы (диапазоны Unicode), а также отдельная обработка буквы «ё».
- **Выделение токенов:** Токенами считаются последовательности букв и цифр. Знаки препинания и специальные символы служат разделителями.

Для уменьшения размерности словаря и улучшения качества поиска применяется стемминг. Класс `Lemmatizer` использует библиотеку **libstemmer** (алгоритм Snowball) для приведения слов к их псевдо-основе.

Итоговый конвейер обработки в `main.cpp` выглядит следующим образом: документ загружается из MongoDB → очищается от HTML-тегов → разбивается на токены → применяется стемминг → добавляется в инвертированный индекс.

Проверка закона Ципфа

В процессе индексации была собрана статистика частотности термов. Для проверки эмпирического закона Ципфа данные были экспортированы в CSV-файл, термы отсортированы по убыванию частоты (рангу), и построен график в двойной логарифмической шкале (log-log plot).

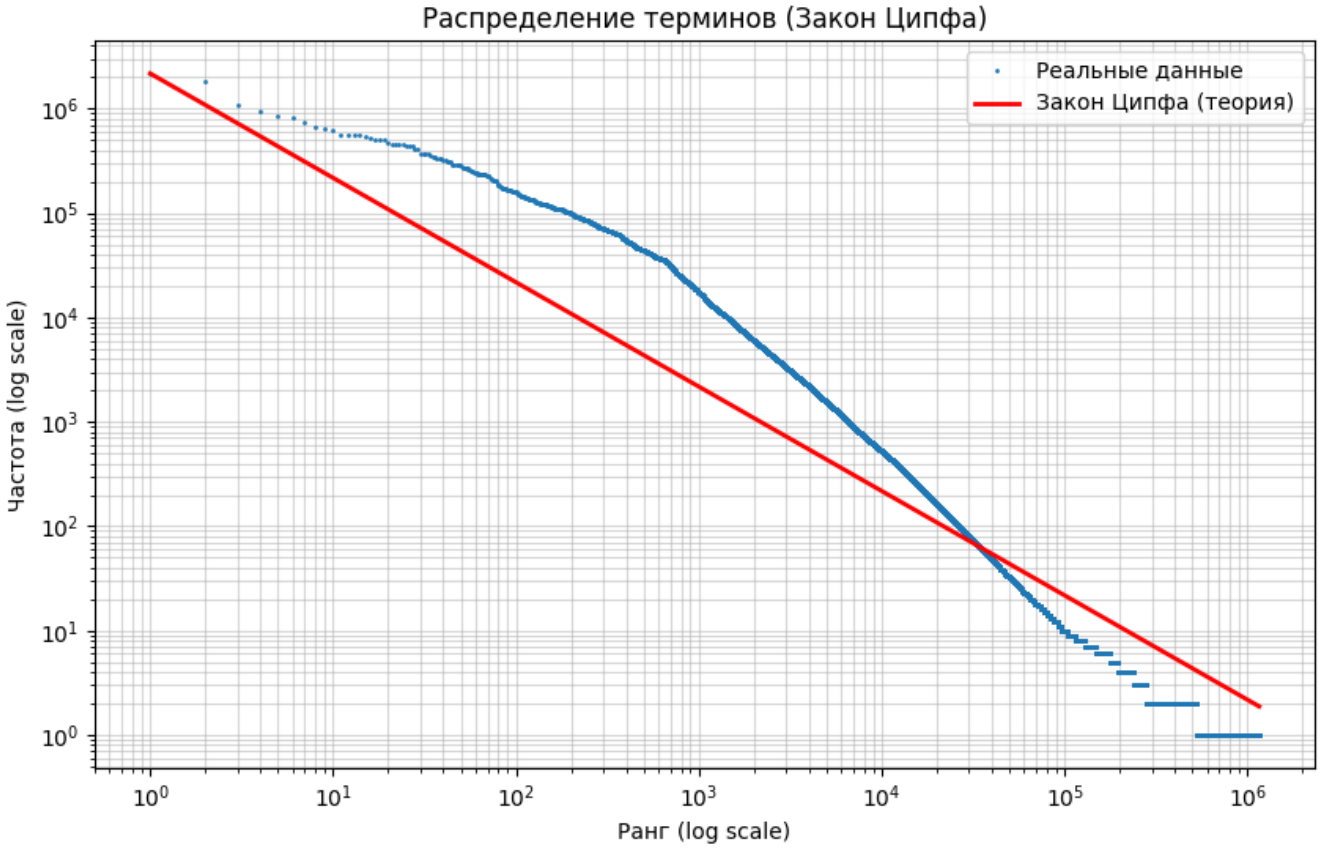


Рис. 3: График зависимости частоты терма от его ранга (log-log шкала)

Как видно из графика (рис. 3), распределение слов в собранном корпусе аппроксимируется прямой линией с отрицательным наклоном, что подтверждает соответствие собранных данных степенному закону распределения (закону Ципфа). Небольшие отклонения в «хвосте» графика объясняются наличием редких уникальных токенов (чисел, специфических аббревиатур), которые встречаются в корпусе единожды.

Булев индекс и Булев поиск

При разработке поисковой системы первоочередной задачей была реализация ранжирующего поиска (TF-IDF) и соответствующего обратного индекса. Приступая к реализации Булева поиска, было принято архитектурное решение не дублировать ресурсоемкий процесс парсинга исходных документов из базы данных. Вместо этого реализован механизм конвертации уже построенного основного индекса в формат Булева индекса.

В рамках проекта был реализован класс `BooleanIndex`. В отличие от наивных реализаций с линейным поиском, для хранения словаря в оперативной памяти используется собственная реализация хеш-таблицы (`HashMap`), что обеспечивает среднюю асимптотическую сложность поиска $O(1)$.

Построение индекса

Поскольку основной индекс уже содержит всю необходимую информацию о токенах, построение булева индекса происходит путем экспорта данных (метод `exportToBooleanIndex`):

1. Программа загружает в память уже готовый обратный индекс (`index.bin`), иначе сначала строится обратный индекс.
2. Происходит итерация по всем терминам основного индекса.
3. Для каждого термина берется список документов (Posting List).
4. Из списка удаляется информация о частоте (TF), остаются только идентификаторы документов (DocID).
5. Полученные данные записываются в бинарный файл булева индекса.

Такой подход позволил сократить время подготовки данных, исключив повторную токенизацию и лемматизацию текстов.

Сериализация и хранение

Для сохранения индекса на диск был разработан упрощенный бинарный формат сериализации. Данные записываются в файл `boolean_index.bin` в следующей последовательности:

- Общее количество документов (метаданные);
- Количество уникальных термов в словаре;
- Для каждого терма:
  - Длина строки (слова) в байтах и само слово (терм);
  - Количество документов, в которых встречается терм;
  - Массив идентификаторов документов (`uint32_t`).

Общий объем индекса существенно меньше основного, так как не хранятся частоты. При старте программы в режиме булева поиска этот файл загружается в `HashMap`.

Алгоритм поиска

Поиск по булеву индексу реализован в классе `QueryParser` и поддерживает сложные запросы с операторами `AND`, `OR`, `NOT` и скобками. Процесс выполнения запроса состоит из следующих этапов:

1. **Лексический анализ.** Поисковый запрос пользователя разбивается на токены двух типов: операторы (И, ИЛИ, НЕ, скобки) и термы (слова). Слова подвергаются токенизации и лемматизации (стеммингу).
2. **Преобразование в ОПЗ.** Полученная инфиксная запись запроса преобразуется в Обратную Польскую Запись (RPN) с помощью алгоритма сортировочной станции (Shunting-yard algorithm). Это позволяет корректно учитывать приоритет операторов и вложенность скобок.
3. **Вычисление результата.** Выражение в RPN вычисляется с использованием стека:

- Для каждого слова из индекса извлекается список документов (Posting List).
- Операторы `AND` и `OR` выполняют пересечение (`set_intersection`) и объединение (`set_union`) списков соответственно.
- Оператор `NOT` выполняет инверсию множества относительно полного списка всех документов в коллекции.

Результатом вычисления является итоговый список идентификаторов документов, удовлетворяющих всему булеву выражению.

Рис. 4: Пример выполнения булева поиска

TF-IDF и сжатие индекса

Для реализации ранжированного поиска необходимо хранить не просто факт наличия слова в документе, но и количество его вхождений. Поэтому структура обратного индекса была модифицирована: теперь для каждого термина хранится список структур `Posting`, содержащих `docId` и `termFrequency` (TF).

Сжатие индекса

Учитывая, что индекс хранит огромные массивы целых чисел (ID документов), было реализовано специализированное сжатие, направленное на эффективную упаковку целочисленных последовательностей. В отличие от универсальных алгоритмов (вроде zlib), использованный подход позволяет распаковывать данные намного быстрее.

В классе `Compression` реализован двухступенчатый алгоритм сжатия:

1. **Delta Encoding (Дельта-кодирование).** Поскольку документы индексируются последовательно, списки `docId` всегда отсортированы по возрастанию. Вместо хранения больших чисел сохраняются разности между соседними элементами:

$$\Delta_i = docId_i - docId_{i-1}$$

   Это позволяет превратить последовательность больших чисел (например, 100500, 100505, 100510) в последовательность очень маленьких (100500, 5, 5).

2. **VarByte (Variable Byte) Compression.** Полученные малые числа кодируются переменным количеством байт. Каждое число занимает от 1 до 5 байт. В каждом байте 7 бит используются для данных, а старший 8-й бит служит флагом продолжения. Это позволяет записывать часто встречающиеся малые дельты всего в 1 байт.

Благодаря этому подходу удалось существенно сократить требования к дисковому пространству без потери производительности при поиске. Сжатый индекс по итогу весит 159.1 Мб. Без сжатия файл индекса стоит 484.1 Мб.

Ранжирование (Scorer)

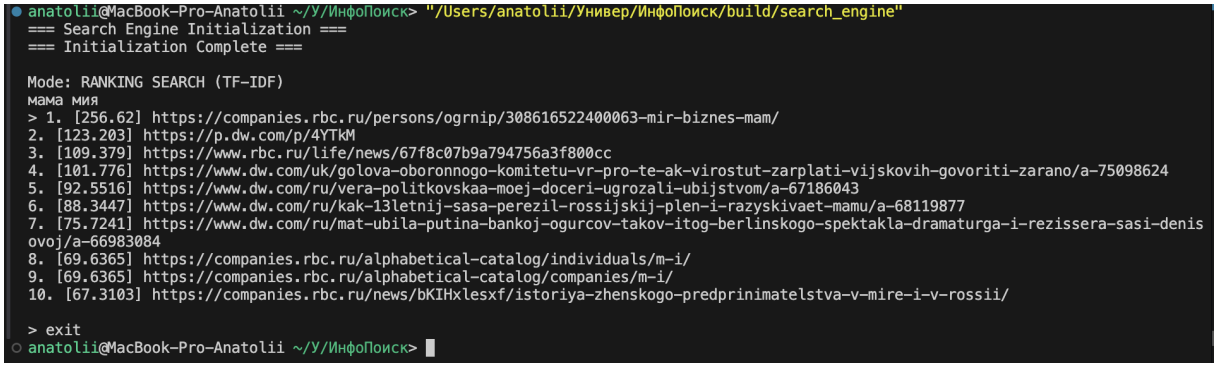


Рис. 5: Пример ранжированного поиска с использованием TF-IDF

Для сортировки результатов по релевантности был реализован класс `Scorer`. Оценка релевантности документа d запросу q вычисляется как сумма весов входящих в него терминов:

$$Score(q, d) = \sum_{t \in q} TF(t, d) \cdot IDF(t)$$

В текущей реализации используются следующие формулы:

$$TF(t, d) = count(t, d)$$

$$IDF(t) = \ln\left(\frac{N}{df(t)}\right)$$

где:

- $count(t, d)$ — количество вхождений термина t в документ d;
- N — общее количество документов в коллекции;
- $df(t)$ (document frequency) — количество документов, содержащих термин t.

Алгоритм поиска выглядит следующим образом:

1. Для каждого слова из запроса извлекается список постигов.
2. Вычисляется вес IDF для текущего слова.
3. Происходит итерация по списку документов. Для каждого документа вычисляется вклад текущего слова ($TF \times IDF$) и добавляется в аккумулятор баллов (используется хеш-таблица `docScores`).
4. После обработки всех слов запроса, результаты преобразуются в массив и сортируются по убыванию итогового балла.

Вывод

В ходе выполнения работы изучены такие важные аспекты информационного поиска как: токенизация, стемминг, индексация, булев поиск и поиск с применением ранжирования по TF-IDF. Создан поисковый движок, позволяющий искать новостные статьи о мировых событиях и политике, сохраняя высокую скорость обработки запросов и релевантность выдачи. Весь код находится в репозитории на Github: https://github.com/TheAnatolii/InfoSearch/tree/main