Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: Юрков Е.Ю.

Группа: М8О-412Б-22

Преподаватель: Кухтичев А. А.

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2025

Содержание

Цель работы

- Добыча корпуса документов
- Создание поискового робота
- Реализовать поисковый движок со следующими компонентами:
  - Токенизация
  - Лемматизация
  - Булев индекс
  - Булев поиск
  - Цитатный поиск
  - Сжатие индекса
  - Ускорение, прыжки по индексу
  - Ранжирование (TF-IDF)
  - Автотесты
  - Построение сниппетов
- Рассчитать график закона Ципфа для выбранного корпуса документов.

Описание данных

Для корпуса документов были выбраны следующие сайты: *habr.com*, *www.geeksforgeeks.com*. Они содержат различные статьи из темы ИТ на русском и английском языках.

Для обхода страниц сайтов использовались sitemap.xml: `https://habr.com/sitemap.xml` и `https://www.geeksforgeeks.org/sitemap_index_new.xml`.

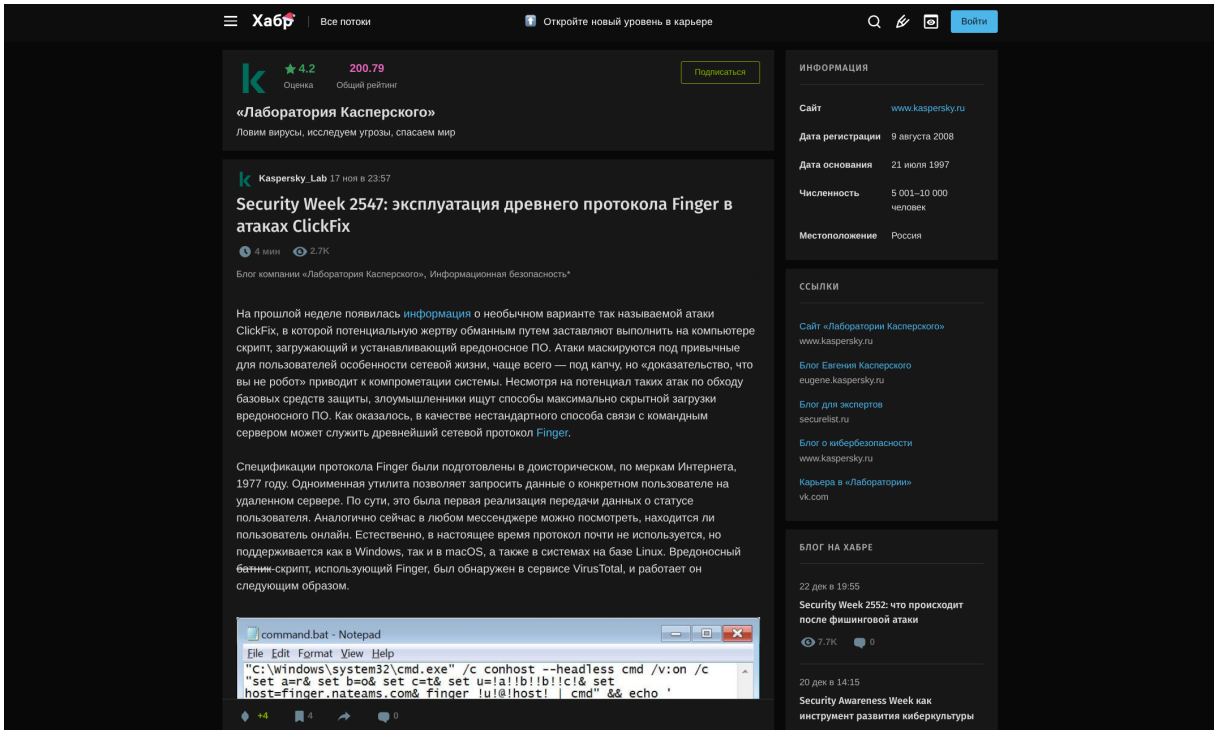

Рис. 1: Пример страницы с habr.com

Для тестирования поискового движка было скачано и обработано 50000 статей.

Обкачка документов

Поисковый робот был реализован на языке `Python`.

Был реализован класс `UrlFetcher`, который совершает обход sitemap и предоставляет url сайтов для обкачки в асинхронную очередь. Класс `PageDownloader` брал url из очереди и скачивал страницы. Обращение к страницам в сети интернет производилось с помощью библиотеки `aiohttp`.

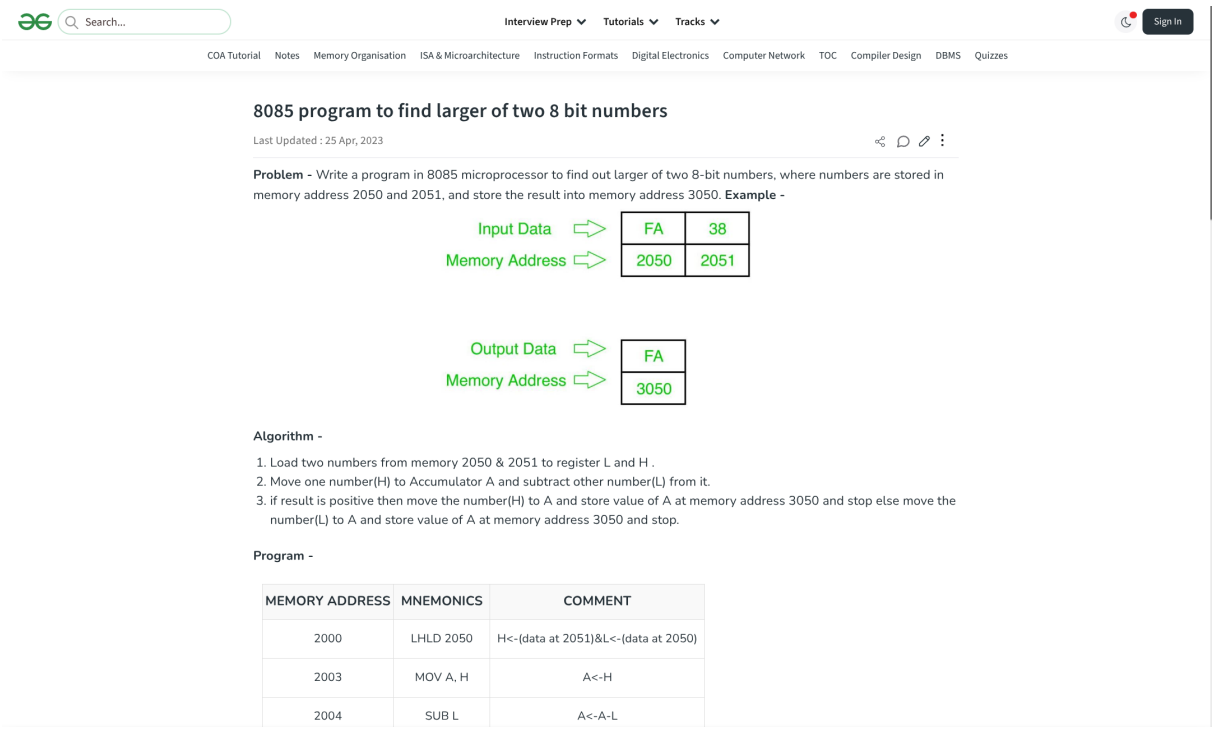

Рис. 2: Пример страницы с www.geeksforgeeks.com

Для парсинга статьи из html использовалась библиотека `BeautifulSoup`. Также был реализован класс `StorageManager`, который сохранял скачанные страницы в базу данных *Mongo*. Для взаимодействия с *MongoDB* использовалась библиотека `pymongo`.

Итоговый документ после скачивания в базе данных имел следующие поля:

- _id - ObjectID, генерируемый *MongoDB*;
- url - ссылка на страницу в интернете;
- site - источник, с которого взята страница (*habr.com* или *www.geeksforgeeks.com*)
- html - сырой html страницы, который использовался для парсинга.
- text - заголовок и текст извлечённый парсером (возможно пустой);
- created_at - время скачивания страницы;
- doc_id - идентификатор документа типа int32, который используется при построении индекса.

Также поисковый робот сохранял в отдельную коллекцию базы данных информацию об url последней скачанной страницы для каждого сайта, чтобы при повторном запуске начинать обкачку с неё и не повторяться.

Предобработка текста

Сам поисквый движок был написан на языке C++. Первой реализованной библиотекой была библиотека `linguistics`. В ней содеражатся классы `Tokenizer`, `Lemmatizer` и `Preprocessor`, который объединяет в себе предыдущие два.

Для токенизации есть несколько правил: для обработки слов, чисел и аббревиатур. Они определяют удовлетворяет ли текст, начинающийся с заданной позиции правилу и отделяют токен от текста. Разделение на правила нужно, из-за того, что не всегда правильно отделять токен по пробелам и знакам препинания. Например, аббревиатуры могут быть записаны подряд, а могут быть разделены точками хотя являются целым словом: СССР или С.С.С.Р., числа могут быть дробными и содержать запятую или точку: 0.25 или 0,25.

Для лемматизации было реализовано два класса: `MockLemmatizer`, который не производил лемматизацию, но использовался в тестах для создания `Preprocessor`, и `DictLemmatizer`, который производил словарную лемматизацию. Для словарной лемматизации был скачан словарь лемм *OpenCorpora* для русских слов и словарь *UniMorph* для английских слов. Для слов, которые не нашлись в словаре использовался простой алгоритм стемминга, который обрезает окончания слов. Для скачивания словарей написан скрипт `scripts/build_dicts.py`.

Для токенизации и стемминга были написаны тесты с использованием библиотеки *GoogleTest*.

Индекс и сжатие

Основная библиотека поискового движка - `engine`. В ней реализован индекс и обработчики запросов.

Индекс представлен классом `InvertedIndex`. Он хранит списки постингов соответствующие каждому слову. Списки хранятся в классе `HashTable`, который представляет собой хеш таблицу, которая в качестве ключа использует `std::string`, а в качестве значения - `PostingList`.

Для ранжирования в индексе для каждого документа добавляется количество вхождений слова (tf). А для поддержания цитатных запросов нужно хранить ещё и координаты этого слова в документе. Поэтому в `PostingList` хранятся следующие структуры:

- `doc_buffer`: `[(doc_id, coord_pos), ...]`;
- `coord_buffer`: `[(tf, [pos_0, pos_1, ...]), ...]`.

где `coord_pos` — позиция начала координатного списка в массиве `coord_buffer`, `tf` — размер массива координат или количество вхождений слова в документ, а `pos_i` — позиции слова в документе.

Для сжатия `PostingList` был реализован класс `CompressedPostingList`. В нём данные doc_buffer и coord_buffer сжаты с помощью V-Byte Encoding. После сжатие усложнился доступ к элементам списка, поэтому для булевого поиска и ранжированного поиска были реализованы классы итераторов: `DocIterator` для получения doc_id, tf и создания итератора по координатам и `CoordIterator` для итерации по позициям в документе (для цитатного поиска). Для ещё большего сжатия вместо самих doc_id храняться разности соседних документов, что также позволяет уменьшить размер doc_id и увеличить пользу V-Byte Encoding. Аналогичный подход применяется и к координатам в документе.

Для ускоренной итерации при пересечении двух `PostingList` используется `skip_buffer`, который хранит указатели для прыжков по индексу. Они хранят позиции части документов в `doc_buffer`. Чтобы таблица прыжков не занимала слишком много памяти (так как к ней не применяется сжатие) и достаточно ускоряла обход списка документов размеры прыжка определяются по следующей формуле: $skip\_size = \sqrt{N}$, где N - это общее количество документов в списке.

Для расчёта TF-IDF в классе `InvertedIndex` также храняться длины каждого документа, они используются для косинусной нормализации.

Чтобы не терять индекс после его построения был написан класс `FileIndexStorage`. Он позволяет скачивать и загружать индекс из бинарного файла.

На основе данных индекса была рассчитана зависимость частоты слов от их порядка в отсортированном по частоте списке.

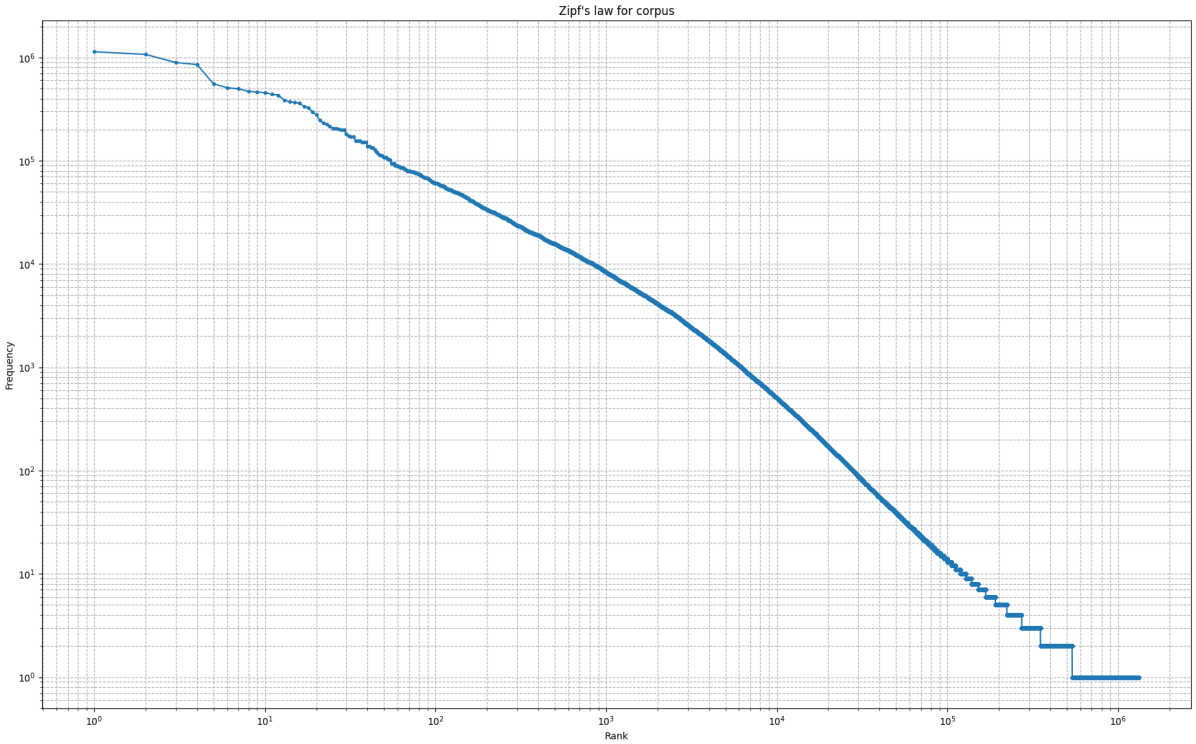

Рис. 3: График закона Ципфа

Закон Ципфа описывает частоту появления элементов в ранжированных списках. Он утверждает, что во многих наборах данных естественного языка частота элемента обратно пропорциональна его рангу.

Если упорядочить слова в тексте по убыванию частоты, то:

$$P(r) \frac{1}{r^{\alpha}}$$

где $P(r)$ - частота элемента с рангом r, r - ранг элемента, α - коэффициент, близкий к 1 для многих естественных распределений.

Визуальный анализ графика в двойных логарифмических координатах показывает выраженную линейную зависимость. Это подтверждает,

что распределение частот терминов в корпусе документов соответствует степенному закону и, в частности, закону Ципфа.

Обработка запросов

У поискового движка есть два режима: булевый поиск и ранжированный. Булевый используется при запуске с флагом `-boolean`. Ранжированный используется по умолчанию. В движке реализована поддержка цитатных (фразовых) запросов, которые учитывают относительный порядок слов при поиске. Для их использования нужно брать искомую фразу в кавычки.

Класс обработки булевых запросов называется `BoolQuery`, у него есть два основных метода: `Parse` и `Execute`. При парсинге булевого запроса строится дерево, где каждый `ASTNode` имеет один из следующих типов: Term, Phrase, And или Or. При исполнении запроса операции And и Or выполняются с помощью соответствующих методов `CompressedPostingList`, которые используют прыжки по индексу для ускорения операций.

Класс обработки ранжированных запросов `RankedQuery` имеет тот же интерфейс, что и `BoolQuery`. Для ранжирования используется TF-IDF.

В данной реализации используется векторная модель поиска с TF–IDF взвешиванием и косинусной мерой сходства между запросом и документами.

TF (term frequency). Для терма t применяется логарифмически сглаженная частота:

$$\mathrm{tf}(t,d) = \begin{cases} 1 + \log f_{t,d}, & f_{t,d} > 0, \\ 0, & f_{t,d} = 0, \end{cases}$$

где $f_{t,d}$ — число вхождений терма t в документе d. Аналогично определяется частота терма в запросе q:

$$\mathrm{tf}(t,q) = 1 + \log f_{t,q}.$$

IDF (inverse document frequency). Используется сглаженная версия обратной документной частоты:

$$\mathrm{idf}(t) = \log \frac{1+N}{1+\mathrm{df}(t)} + 1,$$

где N — общее число документов в коллекции, а $\mathrm{df}(t)$ — количество документов, содержащих терм t. Сглаживание предотвращает деление на ноль и уменьшает влияние редких выбросов.

Веса термов. Вес терма t в документе и в запросе определяется как:

$$w_{t,d} = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t), \qquad w_{t,q} = \mathrm{tf}(t,q) \cdot \mathrm{idf}(t).$$

Скалярное произведение. Ненормализованная оценка релевантности документа d запросу q вычисляется как скалярное произведение TF–IDF векторов:

$$\mathrm{score}(d,q) = \sum_{t \in q \cap d} w_{t,d} \cdot w_{t,q}.$$

Косинусная нормализация. Для устранения влияния длины документа и запроса применяется нормализация:

$$\|q\| = \sqrt{\sum_{t \in q} w_{t,q}^2}, \qquad \|d\| = \sqrt{\sum_{t \in d} w_{t,d}^2}.$$

Итоговый скор вычисляется по формуле косинусного сходства:

$$\mathrm{score}_{\mathrm{norm}}(d,q) = \frac{\mathrm{score}(d,q)}{\|d\| \cdot \|q\|}.$$

Документы сортируются по убыванию значения $\mathrm{score}_{\mathrm{norm}}(d,q)$.

Вывод

В ходе выполнения лабораторных работ по предмету «Информационный поиск» были последовательно изучены и реализованы ключевые этапы построения поисковой системы, начиная с добычи корпуса документов и разработки поискового робота и заканчивая созданием полноценного поискового движка. В рамках проекта были рассмотрены базовые и продвинутые методы обработки текстов, включая токенизацию и лемматизацию, а также построение булева индекса и реализацию различных видов поиска, таких как булев и цитатный.

Особое внимание было уделено вопросам эффективности хранения и обработки данных: реализованы механизмы сжатия индекса, ускорения поиска за счёт прыжков по индексным спискам и ранжирования документов на основе метрики TF-IDF. Дополнительно была выполнена разработка автотестов для проверки корректности работы поискового движка и реализовано построение сниппетов, повышающих удобство представления результатов поиска. Анализ распределения частот слов в корпусе документов с помощью графика закона Ципфа позволил на практике изучить статистические свойства естественного языка и их влияние на задачи информационного поиска.

Выполнение данного проекта позволило получить целостное представление о внутреннем устройстве поисковых систем и основных алгоритмах, применяемых в информационном поиске. Полученные навыки могут быть полезны при разработке поисковых сервисов, систем анализа текстов, рекомендательных систем и других приложений, работающих с большими объёмами текстовой информации.