Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: Д.В. Казарцев
Преподаватель: А.А. Кухтичев
Группа: М8О-410Б
Дата:
Оценка:
Подпись:

Москва, 2025

1 Описание

Целью лабораторных работ по курсу «Информационный поиск» является построение полноценной поисковой системы, ориентированной на тематический корпус документов — «Автомобильные статьи». Работа включает в себя полный цикл подготовки данных и реализации базовых компонентов поисковой инфраструктуры.

В рамках работы решаются следующие задачи:

- Сбор и нормализация корпуса из открытых источников (английская Wikipedia и auto.ru).
- Анализ структуры HTML-документов: выявление метаинформации, разметки, навигационных и служебных блоков.
- Извлечение чистого текстового содержимого с удалением рекламы, шаблонов и несемантических элементов.
- Реализация поискового робота (краулера) на языке Python 3 с поддержкой вежливого поведения и идемпотентности (повторная обкачка без дублирования).
- Разработка модулей предобработки текста на C++: токенизатор и стеммер, не использующие стандартную библиотеку шаблонов (STL) за исключением `std::string` и `std::vector`.
- Эмпирическая проверка статистических закономерностей языка — в частности, закона Ципфа.
- Построение бинарных прямого и обратного индексов.
- Реализация булевого поисковика с поддержкой логических операций `&&`, `||`, `!` и скобочной группировки.

В качестве источников выбраны:

- **Английская Wikipedia** — содержит энциклопедические статьи о моделях автомобилей, производителях, технологиях, исторических событиях. Контент структурирован, проверен сообществом, легко доступен через HTTP.
- **Auto.ru** — российский портал с обзорами, новостями, техническими характеристиками и аналитикой. Используется для расширения корпуса актуальными данными.

Краулер сохраняет документы в в MongoDB. Это упрощает последующую индексацию и делает систему автономной.

В результате сформирован корпус из **50 218 документов** со средним размером **7 800 символов**. Корпус прошёл полную очистку: удалены блоки `<div class="navbox»`, таблицы `<table class="infobox»`, скрипты, стили, сноски и другие элементы, не относящиеся к основному содержанию статьи.

2 Исходный код

Реализация состоит из двух частей: **сбор данных (Python)** и **обработка/поиск (C++)**. Все компоненты разработаны с акцентом на корректность, производительность и минимализм зависимостей.

Поисковый робот (`crawler.py`, `download.py`)

Робот состоит из двух модулей:

- `crawler.py` — рекурсивно обходит категории и ссылки в рамках заданного раздела Wikipedia (например, `/wiki/Category:Cars`), извлекая все внутренние ссылки на статьи.
- `download.py` — загружает HTML-страницы по списку URL, парсит их с помощью `BeautifulSoup`, удаляет нерелевантные элементы:
  - Все теги `<script>`, `<style>`;
  - Блоки с классом `navbox` (навигационные шаблоны);
  - Таблицы с классом `infobox`;
  - Сноски и примечания.

  Извлечённый текст сохраняется в файл `<doc_id>.txt`.

Метаданные сохраняются в `<doc_id>.meta.json` и включают:

- Нормализованный URL;
- Заголовок статьи;
- Значения HTTP-заголовков: `Last-Modified`, `ETag`;
- Хеш содержимого (SHA-1) для детектирования изменений.

Для обеспечения устойчивости к сбоям робот записывает состояние в `crawler_state.json`, содержащий последний обработанный URL и счётчик скачанных документов. При повторном запуске:

1. Выполняется HEAD-запрос к каждому URL;
2. Сравниваются `Last-Modified` и `ETag` с сохранёнными;
3. Документ перезагружается **только если изменился**;
4. Между запросами соблюдается задержка 1.5 секунды.

Это гарантирует корректность, вежливость и экономию ресурсов.

Токенизатор (`tokenizer.cpp`)

Алгоритм работает в четыре этапа:

1. **Чтение файла в память** в двоичном режиме (`std::ios::binary`) — чтобы избежать проблем с кодировкой (корпус в UTF-8).
2. **Приведение к нижнему регистру** только для латинских букв A–Z (функция `to_lower_ascii`). Нелатинские символы (включая кириллицу, если есть) остаются без изменений.
3. **Фильтрация**: символ считается допустимым, если он принадлежит множеству `[a-z0-9]`. Все остальные — разделители.
4. **Формирование токенов**: при встрече разделителя накопленная последовательность проверяется на длину; токены короче 2 символов отбрасываются.

Результат — вектор `std::string` токенов, готовых к стеммингу и индексации.

Стеммер (`stemmer.cpp`)

Реализован упрощённый алгоритм, вдохновлённый Портером, с итеративным применением правил:

1. Слово приводится к нижнему регистру (если ещё не сделано).
2. Выполняется цикл до стабилизации:
   - Удаляются суффиксы в порядке приоритета:
     - `ies` $\to$ `i`, `es` $\to$ (удаляется), `s` $\to$ (если слово длиннее 2 символов);
     - `ed`, `ing` — при наличии гласной до последней согласной;
     - `ly`, `ness`, `ful`;
     - `e` — если перед ним гласная и слово не оканчивается на `ee`.
   - После удаления `ed`/`ing` применяется правило *удвоения согласной*: если слово оканчивается на CVC (согласная-гласная-согласная) и последняя согласная — не `w, x, y`, то последняя буква дублируется при добавлении суффикса. При стемминге — наоборот: одна из дублирующих согласных удаляется.
3. Слова короче 3 символов не стеммируются.

Алгоритм не использует внешние библиотеки и работает за $O(k)$ для слова длины k.

Проверка закона Ципфа

После токенизации и стемминга всего корпуса строится частотный словарь. Слова сортируются по убыванию частоты $f(r)$, где r — ранг.
Закон Ципфа:

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

Для визуализации строится график в лог-лог масштабе:

- Ось X: $\log(r)$
- Ось Y: $\log(f(r))$

На график наносятся:

- Синие точки — эмпирические данные;
- Красная линия — теоретическая модель Ципфа ($C = 24$);
- Зелёная пунктирная линия — модель Мандельброта ($f(r) = C/(r+b)^a$, $a = 1.0$, $b = 0.5$).

Результаты:

- Наблюдается линейная зависимость с наклоном ≈ -1 — закон Ципфа подтверждён.
- Первые 10 слов составляют $\sim 15\%$ всех токенов.
- Топ-1000 слов — $\sim 50\%$ корпуса.

Это свидетельствует о естественности корпуса и пригодности для построения поисковой системы.

Булев индекс (`index.cpp`)

Строится два бинарных файла:

- **Обратный индекс** (`inverted_index.bin`): Вектор структур `TermRecord { std::string term; std::vector<int> doc_ids; }`. После построения — сортируется по `term` для бинарного поиска.

- **Прямой индекс** (`forward_index.bin`): Вектор `DocRecord { int doc_id; std::string title; std::string url; }`. Порядок документов совпадает с именами файлов в `corpus_en/`.

Сериализация — побайтовая, с явным указанием длин строк (например, сначала 4 байта — длина строки, затем сама строка). Это обеспечивает:

- Компактность (без избыточного текстового формата);
- Быструю десериализацию (прямое чтение в структуры);
- Платформонезависимость при соблюдении endianness.

Сложность построения: $O(N \cdot L + M \log M)$, где N — число документов, L — средняя длина, M — число уникальных терминов.

Булев поиск (`search.cpp`)

Поддерживается полный синтаксис:

- `car engine` → `car && engine`
- `car || truck`
- `car && !bike`
- `(car || truck) && engine`

Архитектура:

1. Загрузка индексов из бинарных файлов.
2. Лексический анализ: строка разбивается на токены — термины, операторы (`&&`, `||`, `!`), скобки.
3. Синтаксический анализ методом рекурсивного спуска:
   - `evaluate_expression()` — обрабатывает OR;
   - `evaluate_term()` — обрабатывает AND;
   - `evaluate_factor()` — обрабатывает NOT и скобки.
4. Операции над списками `doc_id`:
   - AND — пересечение (слияние двух отсортированных списков);

- OR — объединение (слияние с удалением дубликатов);
- NOT — вычитание из полного множества документов (реализовано через `std::set` для $O(\log N)$ проверки).

Производительность:

- Все списки `doc_id` отсортированы $\rightarrow$ AND/OR за $O(|A| + |B|)$;
- Среднее время поиска — < 1 мс на корпусе из 50 тыс. документов;
- Ввод/вывод: `stdin` $\rightarrow$ запрос, `stdout` $\rightarrow$ результаты, `stderr` $\rightarrow$ ошибки.

3 Выводы

Выполнив лабораторные работы по курсу «Информационный поиск», я получил целостное представление о том, как устроена поисковая система «с нуля» — от сбора данных до обработки запросов. На практике я реализовал все ключевые компоненты, которые в реальных поисковиках масштабируются до миллиардов документов, но в лабораторных условиях остаются полностью контролируемыми и понятными.

Во-первых, я освоил принципы работы поисковых роботов: научился корректно обходить веб-сайты, соблюдая правила вежливости, извлекать семантически значимый контент и удалять шум. Это дало понимание, насколько важна качественная предобработка данных — даже самый продвинутый ранжировщик бесполезен, если индекс построен на «грязных» текстах.

Во-вторых, я углубил знания в области лингвистической обработки текста. Реализация токенизатора и стеммера на C++ без использования сторонних библиотек (в том числе большей части STL) научила меня работать на низком уровне: управлять памятью, оптимизировать циклы и избегать скрытых накладных расходов. Я лучше понял, как устроены классические алгоритмы, такие как стеммер Портера, и почему они до сих пор актуальны.

В-третьих, проверка закона Ципфа не была просто «галочкой» — она показала, что собранный корпус действительно обладает статистическими свойствами естественного языка. Это важный момент: он подтверждает, что дальнейшие эксперименты (например, с TF-IDF или BM25) будут иметь смысл и давать предсказуемые результаты.

Наконец, реализация булева поиска с поддержкой скобок и логических операций продемонстрировала силу простых, но хорошо продуманных алгоритмов. Использование отсортированных списков `doc_id` и операций слияния позволило достичь времени отклика менее 1 мс даже на корпусе из 50 тыс. документов — это наглядный пример того, как правильная структура данных может заменить сложную логику.

Полученные навыки имеют прямое практическое применение: они лежат в основе не только веб-поиска, но и систем рекомендаций, анализа логов, внутреннего поиска в корпоративных базах знаний и даже некоторых компонентов ИИ-ассистентов. Особенно ценно, что теперь я понимаю не только «как использовать», но и «как устроено внутри» — а это ключевое отличие инженера от потребителя API.

Список литературы

[1] Маннинг, К. Д., Рагхаван, П., Шютце, Х. *Введение в информационный поиск* — М.: Издательский дом «Вильямс», 2011. — 528 с. ISBN 978-5-8459-1623-4 (рус.). Перевод с английского под ред. доктора физ.-мат. наук Д. А. Клюшиной.

[2] Porter, M. F. *An algorithm for suffix stripping* // Program. — 1980. — Vol. 14, No. 3. — P. 130–137.

[3] Zipf, G. K. *Human Behavior and the Principle of Least Effort* — Cambridge, MA: Addison-Wesley, 1949.

[4] Wikipedia contributors. *Zipf's law* // Wikipedia, The Free Encyclopedia. — URL: `https://en.wikipedia.org/wiki/Zipf%27s_law` (дата обращения: 30.12.2025).

[5] ГОСТ Р 7.0.5–2008. *Библиографическая ссылка. Общие требования и правила составления.* — Введ. 2008–07–01. — М.: Стандартинформ, 2008. — 33 с. URL: `http://www.ifap.ru/library/gost/7052008.pdf` (дата обращения: 30.12.2025).