Московский авиационный институт

(национальный исследовательский университет)

Институт информационные технологии и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Информационный поиск»

ПРОВЕРКА ОРФОГРАФИИ ПРИ ПОИСКЕ

Студент: Е.М. Стифеев
Преподаватель: А.А. Кухтичев
Группа: М8О-109М-21
Дата: 03.12.21
Оценка:
Подпись:

Москва, 2021

Оглавление

Задание

Необходимо реализовать проверку опечаток в поисковых запросах для системы, разработанной по итогам лабораторных работ по курсу «Информационного поиска» и «Обработки текстов на естественном языке».

1. Описание существующей поисковой системы

Корпус

Поисковая система обрабатывает запросы для корпуса документов, хранящегося на диске.

По итогам лабораторных работ по курсу с помощью веб-скрапинга по нескольким сайтам был получен корпус документов (доступен по ссылке https://cloud.mail.ru/public/ZfkX/gccM7hnDR), который имеет следующую структуру:

- films1.txt (94 Мб, 15000 документов, UTF-8)
- films2.txt (96 Мб, 15000 документов, UTF-8)
- films3.txt (184 Мб, 15000 документов, UTF-8)
- films4.txt (219 Мб, 15000 документов, UTF-8)
- films5.txt (322 Мб, 15000 документов, UTF-8)
- films6.txt (711 Мб, 15000 документов, UTF-8)
- films7.txt (823 Мб, 15000 документов, UTF-8)
- films8.txt (226 Мб, 15000 документов, UTF-8)
- films9.txt (67 Мб, 15000 документов, UTF-8)
- films10.txt (75 Мб, 15000 документов, UTF-8)
- films11.txt (99 Мб, 15000 документов, UTF-8)
- films12.txt (78 Мб, 15000 документов, UTF-8)
- films13.txt (41 Мб, 6109 документов, UTF-8)

$$\Sigma_{Gb} = 2{,}899\ \text{Gb}, \Sigma_{\text{docs}} = 186109$$

Получение одного документа зачастую включало проход по нескольким html-страницам и обработку динамически подгружаемых страниц, поэтому общее количество обработанных страниц было >800’000.

Всего была обкачено три сайта:

- https://www.kinopoisk.ru/
- https://shikimori.one/
- http://filmplace.ru/

В каждом файле *.txt документы хранятся следующим образом:

- 1 строка 1 документ {….}
- 2 строка 2 документ {….}
- *n* строка *n* документ {….}

Каждый документ снабжён прямой ссылкой на источник, откуда был скачен, и хранит только выделенный из html-кода текст в кодировке UTF-8. Например, 234 строка файла films1.txt выглядит так:

{"page_url": "https://www.kinopoisk.ru/media/article/1773537/", "title": "Артур Смольянинов: «Я сомневался, что смогу сыграть ангела»", "body": "2 января в российский прокат вышла романтическая комедия Веры Сторожевой „Мой парень — ангел“, главные роли в которой исполнили Артур Смольянинов и Анна Старшенбаум. Мы подготовили небольшой видеосюжет с участием создателей картины...Студентка Саша с большим трудом верит в чудеса. Ангелу Серафиму приходится приложить немало усилий, чтобы доказать ей, что ангелы существуют. Но он не учел одного: если девушка тебе поверит, она, скорее всего, тебя полюбит.\n\n\n\n\n\n\nАвтор: Дарико Цулая", "comments": ""}.

Индекс

Готовый индекс хранится в четырёх файлах (доступен по ссылке https://cloud.mail.ru/public/wynT/adagiBjh9):

- **docs_id.data** (42 Мб)

Файл служит для отображения индекса документа (doc_id) в его текстовое представление в файлах *.txt. Поддерживается переменная длина пути до файлов с документами.

Структура

n_docs					
offset[0]	offset[1]	…	offset[n_docs-1]	offset[n_docs]	
n_chars[0]	name[0]		doc_offset[0]	doc_size[0]	
n_chars[1]	name[1]		doc_offset[1]	doc_size[1]	
…	…		…	…	
n_chars[n_docs-1]	name[n_docs-1]		doc_offset[n_docs-1]	doc_size[n_docs-1]	

Описание полей

Название	Тип	Назначение
n_docs	uint	Число документов в корпусе
offset[0],…, offset[n_docs]	uint	Смещения в байтах до строк таблицы с описанием документов, расположенной ниже. Таким образом, если понадобится открыть документ с doc_id = 5, то можно будет сразу переместить головку диска (fseek) до offset[5], прочитать это поле и сразу сместиться до нужной строки в таблице на offset[5], чтобы попасть в начало n_chars[5]. offset имеет на один элемент больше чем нужно (offset[n_docs]), чтобы работала блочная индексация и слияние блоков
n_chars[0],…, n_chars[n_docs-1]	uint	Число символов wchar в абсолютном пути до файла, где хранится документ
name[0],…, name[n_docs-1]	*wchar	Абсолютный путь до файла *.txt в кодировке UTF-16, т.е. два байта на символ

doc_offset[0],… doc_size[n_docs-1]	uint	Смещение в байтах до начала документа в файле *.txt
doc_size[0],… doc_size[n_docs-1]	uint	Размер документа в байтах.

- **terms.data** (54 Мб)

Файл служит для хранения словаря с терминами и ссылок (смещений) на файл с словопозициями и координатами. Поддерживается переменная длина термина. Термины упорядочены в лексикографическом порядке.

Структура

n_terms		
n_chars[0]	term[0]	offset_post_list[0]
n_chars[1]	term[1]	offset_post_list[1]
…	…	…
n_chars[n_terms-1]	term[n_terms-1]	offset_post_list[n_terms-1]

Описание полей

Название	Тип	Назначение
n_terms	uint	Число терминов в корпусе/ число списков словопозиций
n_chars[0],…, n_chars[n_terms-1]	uint	Число символов в термине
term[0],…, term[n_terms-1]	*wchar_t	Термин в кодировке UTF-16
offset_post_list[0],…, offset_post_list[n_docs-1]	uint	Смещение в файле со словопозициями

- **postings_list.data** (2.68 Гб)

Файл служит для хранения словопозиций и координат терминов в документе. Слопозиции упорядочены по возрастанию идентификаторов документов.

Структура

n_terms						
n_docs[0]	doc…doc	freq…freq	offset…offset	begin…begin	end…end	begin…
n_docs[1]	doc…doc	freq…freq	offset…offset	begin…begin	end…end	begin…
…						

Описание полей

Название	Тип	Назначение
n_terms	uint	Число терминов в корпусе/ число списков словопозиций и координат
n_docs[0],…, n_docs [n_terms -1]	uint	Число словопозиций для конкретного терма
doc[i][0],…, doc[i][n_docs[i]-1], i = 0…n_terms-1	*int	Вектор идентификаторов документов, в которых встречается термин (слопозиции)
freq[i][0],…, freq [i][n_docs[i]-1], i = 0…n_terms-1	*int	Вектор частот вхождений терма в документы
offset[i][0],…, offset[i][n_docs[i]-1], i = 0…n_terms-1	*uint	Относительные смещения до координат. Таким образом, если понадобятся координаты i-терма в j-м документе, то сначала выполнится смещение до нужной строки в таблице *posting_list.data* с помощью смещений в словаре. Затем, зная число документов n_docs[i], можно быстро считать freq и offset, не читая остальные данные. Далее, с помощью offset выполняется смещение до блока, в котором находятся координаты begin…begin, end…end терма в документе. Их количество равно значению freq.

begin… begin	*int	Координаты начал термина в документе. Координаты измеряются в символах от начала документа
end…end	*int	Координаты концов термина в документе. Координаты измеряются в символах от начала документа

- **tf.data** (939 Мб)

Файл служит для быстрого получения компонент документа, как вектора в пространстве терминов. Он нужен для быстрого ранжирования на основе косинуса между вектором запроса и вектором документа. См. подробности в ЛР по ранжированию.

Структура

n_docs						
n_terms	ti[0]	…	ti[n_terms-1]	tw[0]	…	tw[n_terms-1]
n_terms	ti[0]	…	ti[n_terms-1]	tw[0]	…	tw[n_terms-1]
…						

Описание полей

Название	Тип	Назначение
n_docs	uint	Число документов в корпусе
n_terms[i] i=0…n_docs-1	*uint	Число терминов в i-м документе
ti[0]… ti[n_terms-1], ti[0]… ti[n_terms-1], …	*int	Вектор идентификаторов терминов
tw[0]… tw[n_terms-1], tw[0]… tw[n_terms-1], …	*double	Вектор весов. Вес соответствует идентификатору

Построение индекса для корпуса с учётом лемматизации терминов с помощью отечественной NLP-системы Natasha [1] занимает 4 часа при распараллеливании на 4 OMP-потока (больше не позволяет размер оперативной памяти) процессора Intel Core i7 9700K (3.6 GHz). Блочный

индекс (до слияния) доступен по ссылке https://cloud.mail.ru/public/F3Fe/eTEiUPHt6.

Поисковая система. Виды запросов

Система поддерживает несколько видов запросов, более подробно рассмотренных в соответствующих лабораторных работах:

1. **Булев поиск**

Синтаксис поисковых запросов:

- Пробел или два амперсанда, «&&», соответствуют логической операции «И».
- Две вертикальных «палочки», «||» – логическая операция «ИЛИ»
- Восклицательный знак, «!» – логическая операция «НЕТ»
- Могут использоваться скобки.

Парсер поисковых запросов устойчив к переменному числу пробелов, максимально толерантен к введённому поисковому запросу.

Примеры запросов:

- `[ московский && авиационный && институт ];`
- `[ (красный || желтый) автомобиль ];`
- `[ руки !ноги ].`

2. **Цитатный поиск**

Синтаксис этого элемента следующий:

- `[ «что где когда» ]` – кавычки, включают режим цитатного поиска для терминов внутри кавычек. Этому запросу удовлетворяют документы, содержащие в себе все термины что, где и когда, причём они должны встретиться внутри документа ровно в этой последовательности, без каких-либо вкраплений других терминов.

- `[ «что где когда» / 5 ]` – аналогично предыдущему пункту, но допускаются вкрапления других терминов так, чтобы расстояние от первого термина цитаты до последнего не превышало бы 5.

Новый элемент может комбинироваться с другими стандартными средствами булева поиска, например:

- `[ «что где когда» && друзь ]`
- `[ «что где когда» || квн ]`
- `[ «что где когда» && !«хрустальная сова» ]`

3. Нечёткий поиск

Если запрос содержит в себе только термины через пробелы, то он трактуется как нечёткий запрос, т.е. допускается неполное соответствие документа терминам запроса и т.п. Примеры запросов:

- `[ роза цветок ]`
- `[ московский авиационный институт ]`

Если запрос содержит в себе операторы булева поиска, то запрос трактуется как булев, т.е. соответствие должно быть строгим, но порядок выдачи должен определён ранжированием TF-IDF. Например:

- `[ роза && цветок ]`
- `[ московский && авиационный && институт ]`

Поисковая система. Интерфейс

Реализовано декстоп-приложение для ОС семейства Windows.

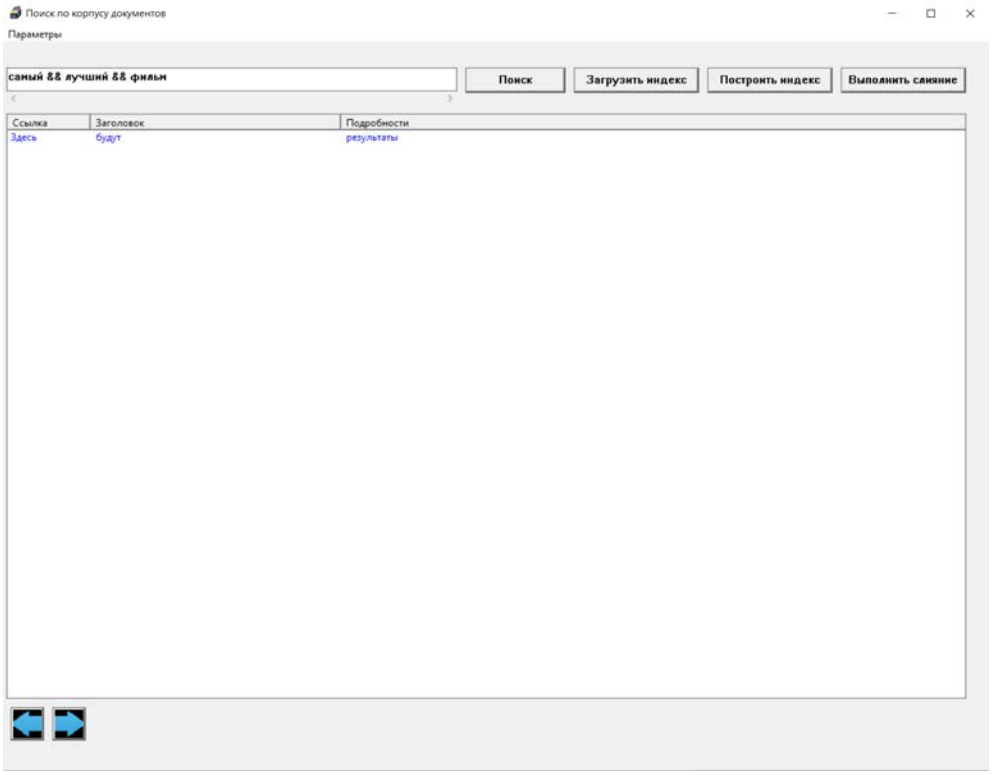


Рис. 1 – Стартовое графическое окно приложения

При запуске приложения пользователь видит два окна: графическое (с элементами управления) и консольное – для логирования.

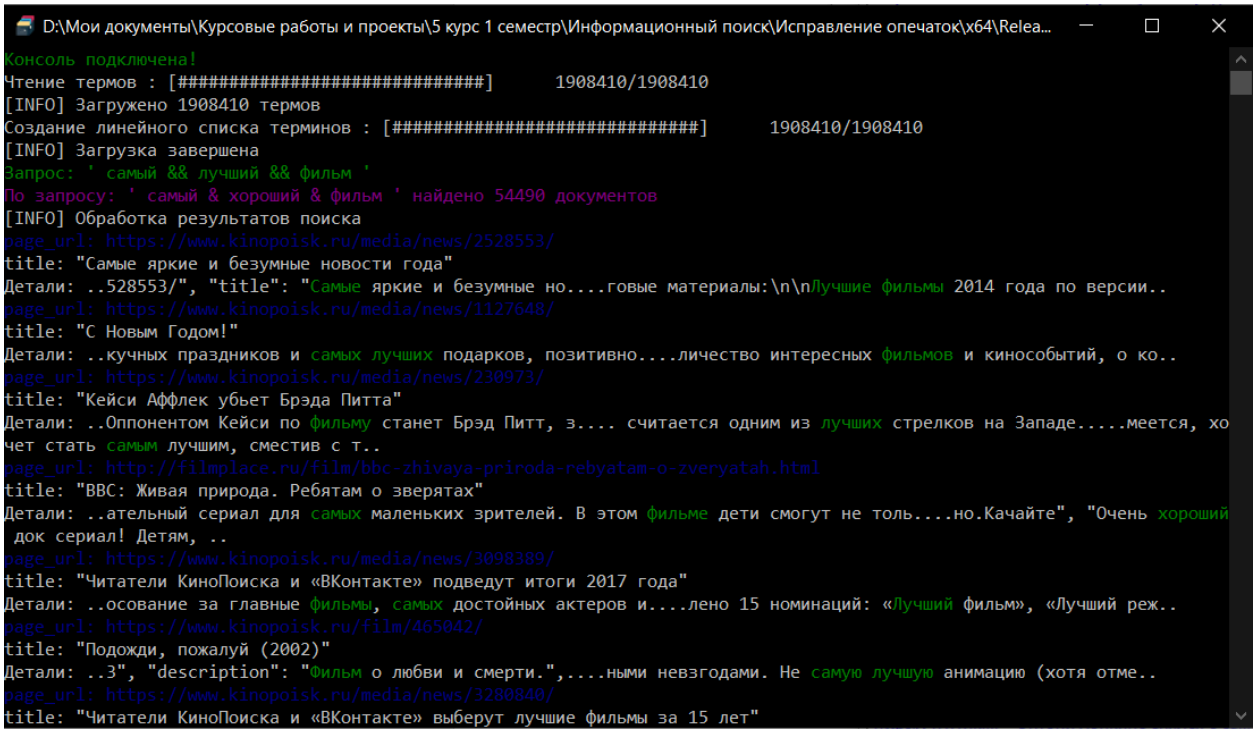


Рис. 2 – Консольное окно приложения

При взаимодействии пользователя с системой последняя через консольное окно ведёт оповещение, чем она «занята» в данный момент. Прежде делать запрос к корпусу необходимо загрузить индекс с помощью кнопки «Загрузить индекс» или, если индекс не создан, то необходимо создать блочный индекс и затем выполнить слияние с помощью соответствующих кнопок. Корпус должен храниться в директории на диске в формате, описанном в информации о корпусе (нумерация файлов необязательна – названия - произвольные). При этом один файл с документами считается системой одним блоком, которые обрабатываются параллельно, так что к выбору размера файла нужно подходить разумно.

В меню «параметры» настраивается количество потоков и пути до нужных директорий. При закрытии и запуске приложения система запоминает все пути, количество потоков, а также последний сделанный пользователем запрос.

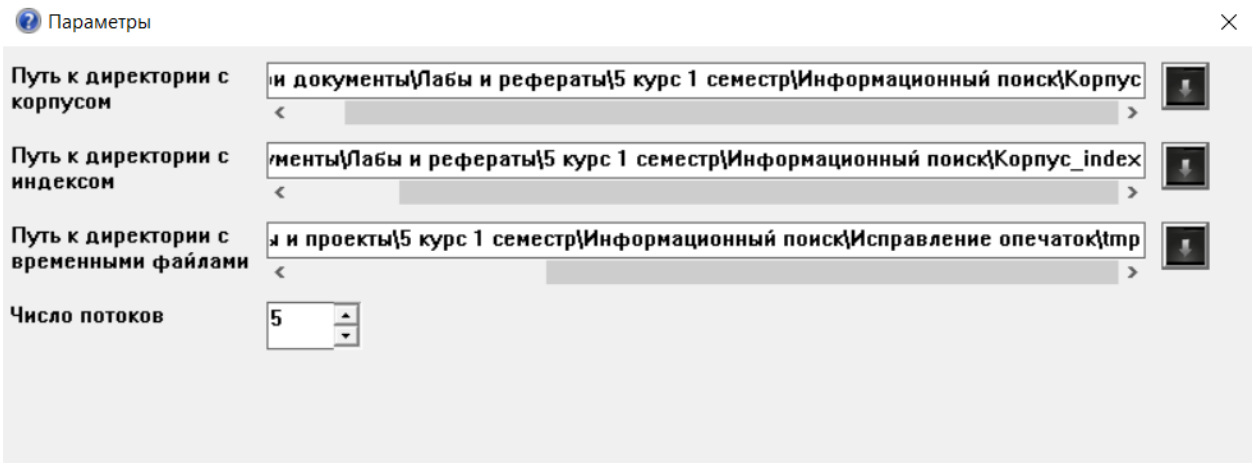


Рис. 3 – Выбор параметров

После загрузки индекса, можно делать запросы в формах, описанных в предыдущем разделе. Результаты сортируются по косинусному правилу TF-IDF.

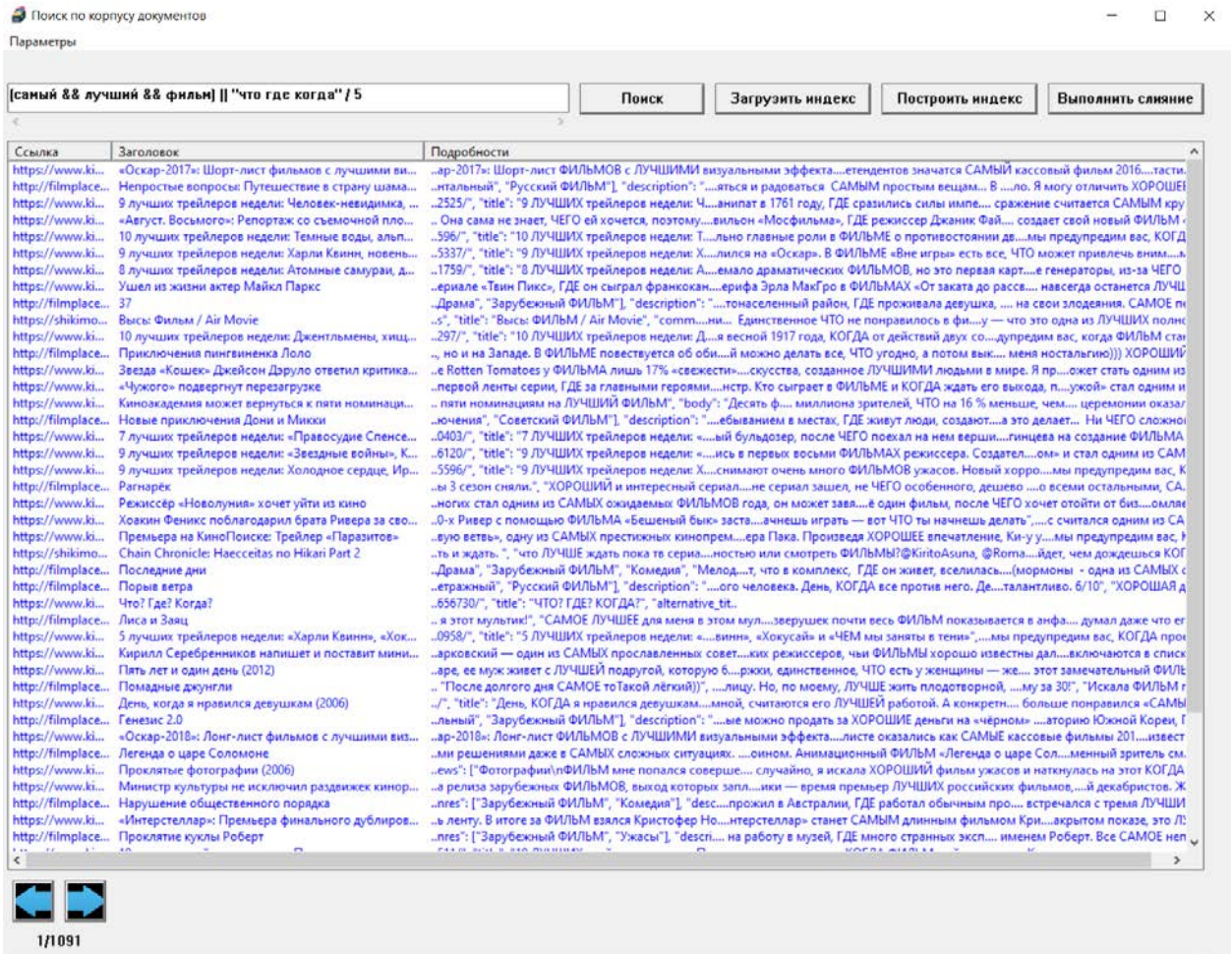


Рис. 4 – Поисковая выдача по комбинированному запросу (графическое окно)

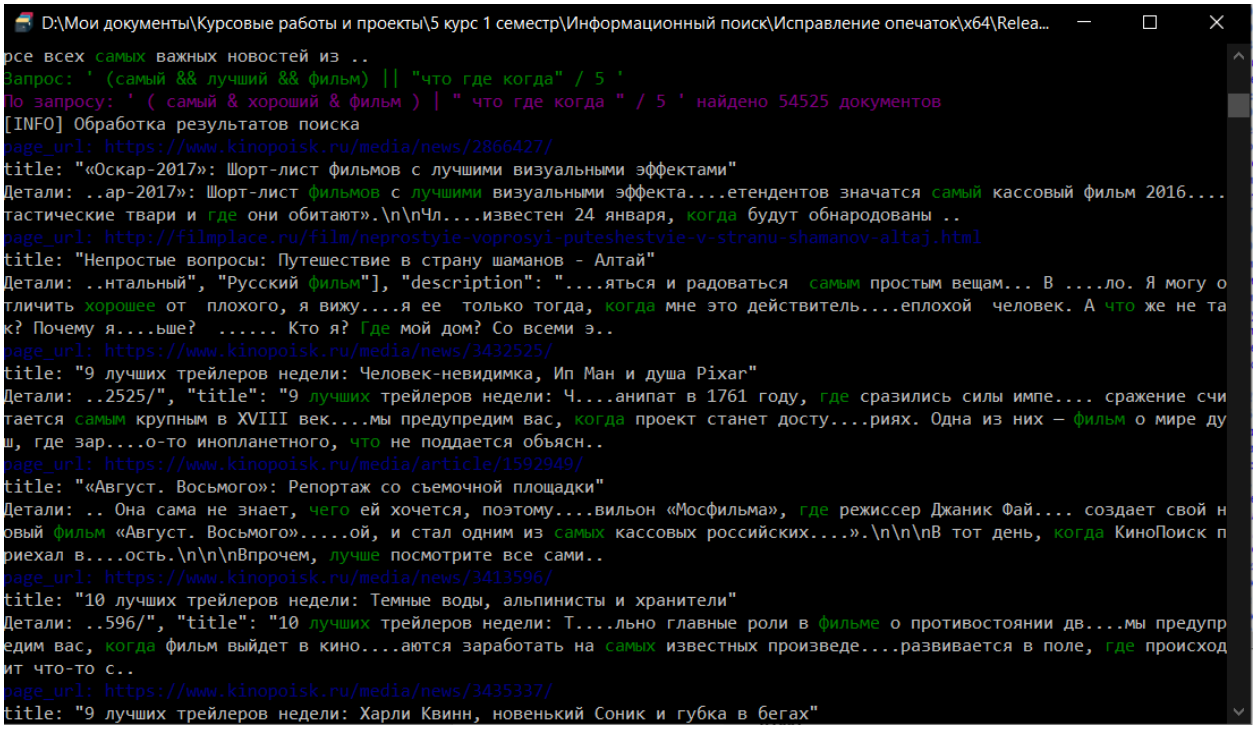


Рис. 5 – Поисковая выдача по комбинированному запросу (консольное окно)

По результатам запроса пользователь открывает ссылку в первом столбце в любом, привычном ему, браузере.

Также существует полностью консольная версия приложения, существующая в т.ч. для тестирования и измерения метрик поисковой системы. Ей на вход в качестве аргументов программы подаются на вход все вышеуказанные параметры, а также путь к данным для просчёта метрик (см. подробности в соответствующей ЛР).

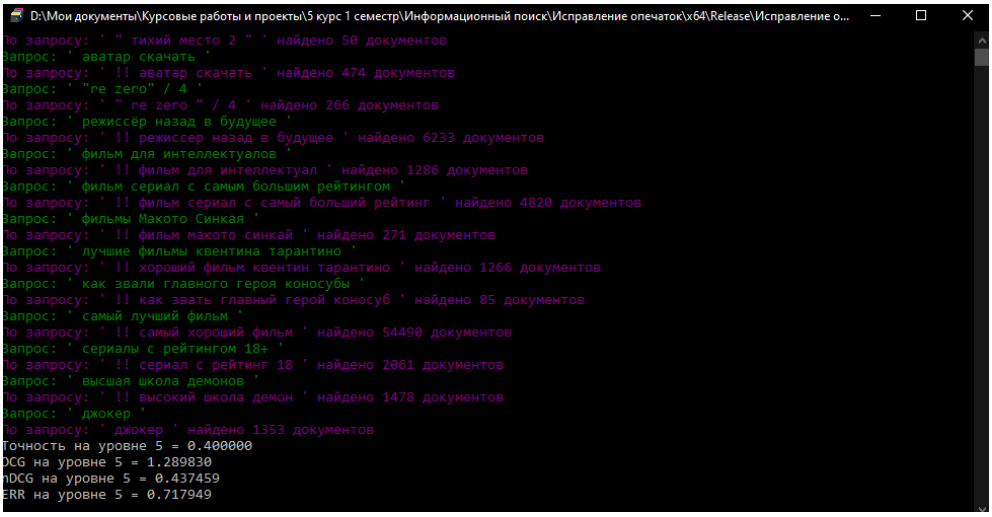


Рис. 6 Утилита тестирования системы

Целью курсового проекта является улучшение существующей системы путём добавление проверки орфографии.

2. Реализация проверки орфографии

Существует [3] две формы исправления ошибок: *исправление изолированного термина* (isolated-term correction) и *исправление с учётом контекста* (context-sensitive correction). При исправлении изолированного термина термины запроса исправляются по отдельности. Пример со словом `carot` иллюстрирует этот подход. Этот метод не позволяет, например, обнаружить, что запрос `flew form Heathrow` содержит искаженный термин `from`, поскольку каждый из терминов запроса по отдельности записан правильно.

По проблеме исправления изолированного термина существует два метода решения задачи: расстояние редактирования и перекрытие k-грамм.

Рассмотрим более подробно методы, основанные на расстоянии редактирования.

Расстояние Левенштейна

Расстояние редактирования между двумя строками символов s_1 и s_2 — это минимальное количество операций редактирования (edit operations), с помощью которых строку s_1 можно трансформировать в строку s_2. Операции редактирования, позволяющие это сделать, включают в себя следующие преобразования:

1) вставка символа в строку,
2) удаление символа из строки,
3) замена символа в строке другим символом.

При указанном наборе операций редактирования расстояние редактирования называется расстоянием Левенштейна (Levenshtein distance). Например, расстояние редактирования между словами `cat` и `dog` равно трем. Расстояние редактирования можно обобщить, если разным операциям редактирования присвоить разные веса. Например, операции замены символа `s` символом `p` можно присвоить более высокий вес, чем операции замены символа `s` символом `a` (так как буква `a` ближе к букве `s` на клавиатуре). Присвоение весов с учетом правдоподобности замены на практике оказалось очень эффективным [3]. Однако в нашем изложении мы будем придерживаться допущения, что все операции редактирования имеют один и тот же вес.

Хорошо известно, как вычислить расстояние редактирования между двумя строками за время $O(|s_1| \times |s_2|)$, где $|s_i|$ — длина строки s_i. Для этого используется алгоритм динамического программирования, представленный на рис. 7, где строки s_1 и s_2 представлены в виде массивов. Этот алгоритм заполняет все (целочисленные) ячейки матрицы m, размеры которой равны длинам соответствующих строк; после выполнения алгоритма элемент (i, j) содержит расстояние редактирования между строками, состоящими из первых i символов строки s_1 и первых j символов строки s_2. Основной шаг динамического программирования отображен в строках 8–10 (рис. 7), в которых вычисляется минимум трех величин с учетом замены символа в строке s_1, вставки символа в строку s_1 и вставки символа в строку s_2.

```
EditDistance(s1, s2)
 1  int m[|s1|, |s2|] = 0
 2  for i ← 1 to |s1|
 3  do m[i, 0] = i
 4  for j ← 1 to |s2|
 5  do m[0, j] = j
 6  for i ← 1 to |s1|
 7  do for j ← 1 to |s2|
 8        do m[i, j] = min{m[i–1,j–1] + if(s1[i] = s2[j]) then 0 else 1fi,
 9                         m[i–1,j] +1,
10                         m[i,j–1] + 1}
11  return m[|s1|, |s2|]
```

Рис. 7 – Алгоритм вычисления расстояния Левенштейна без учёта весов

Реализация алгоритма на C++, представлена алгоритмом 1.

Алгоритм 1 – реализация вычисления расстояние Левенштейна на C++

```
double LevenshteinDistance(const wstring &s1, const wstring &s2, vector<double> &work)
{
    int m = s1.size() + 1,
        n = s2.size() + 1;

    work.resize(m * n);
    vector<double> &M = work;

#define ij2i(i, j) ((i) * n + (j))

    M[ij2i(s1.size(), s2.size())] = 0;

    int i, j;

    for (i = 1; i <= s1.size(); i++)
    {
        M[ij2i(i, 0)] = i;
    }

    for (j = 1; j <= s2.size(); j++)
    {
        M[ij2i(0, j)] = j;
    }

    for (i = 1; i <= s1.size(); i++)
    {
        for (j = 1; j <= s2.size(); j++)
        {
            M[ij2i(i, j)] = min3(M[ij2i(i-1, j-1)] + (s1[i - 1] == s2[j - 1] ? 0 : 1),
                                 M[ij2i(i - 1, j)] + 1,
                                 M[ij2i(i, j - 1)] + 1);
        }
    }

    return M[ij2i(s1.size(), s2.size())];

#undef ij2i
}
```

Вектор work – рабочее пространство, позволяет экономить вызовы malloc'а или память в стеке. Матрица хранится по строкам в линеаризованном массиве (с доступом по макросу ij2i(i, j)) для ускорения доступа путём попадания крайних элементов в одну кеш-линию.

Расстоянии Дамерау-Левенштейна. Некорректный алгоритм

Расстояние Дамерау-Левенштейна (Damerau-Levenshtein distance) между двумя строками, состоящими из конечного числа символов — это минимальное число операций вставки, удаления, замены одного символа и транспозиции двух соседних символов, необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна, отличается от него добавлением операции перестановки. Дамерау показал, что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе [2], поэтому имеет смысл использовать именно эту метрику.

Существует несколько алгоритмов реализации вычисления этого расстояния: упрощённый и корректный алгоритмы.

Упрощённый алгоритм не решает задачу корректно, но бывает полезен на практике. Здесь и далее будем использовать следующие обозначения: S и T — строки, между которыми требуется найти расстояние Дамерау-Левенштейна; M и N — их длины соответственно.

Рассмотрим алгоритм, отличающийся от алгоритма поиска расстояния Левенштейна одной проверкой (храним матрицу D, где $D(i,j)$ — расстояние между префиксами строк: первыми i символами строки S и первыми j символами строки T). Рекуррентное соотношение имеет вид:

Ответ на задачу – $D(M,N)$, где

$$D(i,j) = \begin{cases} \min(A, D(i-2,j-2) + tCost), & \text{если } i>1, j>1, S[i]=T[j-1], S[i-1]=T[j] \\ A, & \text{иначе} \end{cases}$$

$$A = \begin{cases} 0, & \text{если } i=0, j=0 \\ i \cdot dCost, & \text{если } j=0, i>0 \\ j \cdot iCost, & \text{если } i=0, j>0 \\ D(i-1,j-1), & \text{если } S[i]=T[j] \\ \min(& \\ D(i,j-1)+iCost & \\ D(i-1,j)+dCost & \text{если } j>0, i>0, S[i]\neq T[j] \\ D(i-1,j-1)+rCost & \\ ), & \end{cases}$$

$dCost$ – стоимость удаления символа,

$iCost$ – стоимость вставки,

$rCost$ – стоимость удаления,

$tCost$ – стоимость перестановки символов.

Таким образом для получения ответа необходимо заполнить матрицу D, пользуясь рекуррентным соотношением. Сложность алгоритма: $O(M \cdot N)$. Затраты памяти: $O(M \cdot N)$.

Контрпример: $S = 'CA'$ и $T = 'ABC'$. Расстояние Дамерау-Левенштейна между строками равно 2 ($CA \to AC \to ABC$), однако функция приведённая выше возвратит 3. Дело в том, что использование этого упрощённого алгоритма накладывает ограничение: любая подстрока может быть редактирована не более одного раза. Поэтому переход $AC \to ABC$ невозможен, и последовательность действий такая: ($CA \to A \to AB \to ABC$). Упрощенный алгоритм Дамерау-Левенштейна не является метрикой, так как не выполняется правило треугольника: $DLD('CA','AC') + DLD('AC','ABC') \not\geq DLD('CA','ABC')$.

Несмотря на то, что условие многих практических задач в не предполагает многократного редактирования подстрок [2], в задаче исправления опечаток в запросах его недостаточно, ниже представлен более сложный алгоритм, который корректно решает задачу поиска расстояния Дамерау-Левенштейна.

Расстоянии Дамерау-Левенштейна. Корректный алгоритм

В основу алгоритма положена идея динамического программирования по префиксу. Будем хранить матрицу $D[0..M+1][0..N+1]$, где $D[i+1][j+1]$ — расстояние Дамерау-Левенштейна между префиксами строк S и T, длины префиксов — i и j соответственно. Для учёта транспозиции потребуется хранение следующей информации. Инвариант:

$lastPosition[x]$ — индекс последнего вхождения x в S,

$last$ — на i-й итерации внешнего цикла индекс последнего символа T: $T[last] = S[i]$.

Тогда если на очередной итерации внутреннего цикла положить: $i' = lastPosition[T[j]], j' = last$, то

$D(i,j) = \min(A, D(i',j') + (i - i' - 1) \cdot dCost + tCost + (j - j' - 1) \cdot iCost)$, где

$$A = \begin{cases} 0, & \text{если } i = 0, j = 0 \\ i \cdot dCost, & \text{если } j = 0, i > 0 \\ j \cdot iCost, & \text{если } i = 0, j > 0 \\ D(i-1, j-1), & \text{если } S[i] = T[j] \\ \min(& \\ D(i, j-1) + iCost & \\ D(i-1, j) + dCost & \text{если } j > 0, i > 0, S[i] \neq T[j] \\ D(i-1, j-1) + rCost & \\ ), & \end{cases}$$

Сложность алгоритма: $O(M \cdot N \cdot max(M, N))$. Затраты памяти: $O(M \cdot N)$. Однако скорость работы алгоритма может быть улучшена до $O(M \cdot N)$.

Псевдокод алгоритма изображён на рис. 8.

```
int DamerauLevenshteinDistance(S: char[1..M], T: char[1..N]; deleteCost, insertCost, replaceCost, transposeCost: int):
    // Обработка крайних случаев
    if (S == "")
        if (T == "")
            return 0
        else
            return N
    else if (T == "")
        return M
    D: int[0..M + 1][0..N + 1]   // Динамика
    INF = (M + N) * max(deleteCost, insertCost, replaceCost, transposeCost)  // Большая константа

    // База индукции
    D[0][0] = INF
    for i = 0 to M
        D[i + 1][1] = i * deleteCost
        D[i + 1][0] = INF
    for j = 0 to N
        D[1][j + 1] = j * insertCost
        D[0][j + 1] = INF

    lastPosition: int[0..количество различных символов в S и T]
    //для каждого элемента C алфавита задано значение lastPosition[C]

    foreach (char Letter in (S + T))
        lastPosition[Letter] = 0

    for i = 1 to M
        last = 0
        for j = 1 to N
            i' = lastPosition[T[j]]
            j' = last
            if S[i] == T[j]
                D[i + 1][j + 1] = D[i][j]
                last = j
            else
                D[i + 1][j + 1] = min(D[i][j] + replaceCost, D[i + 1][j] + insertCost, D[i][j + 1] + deleteCost)
            D[i + 1][j + 1] = min(D[i + 1][j + 1], D[i'][j'] + (i - i' - 1) · deleteCost + transposeCost + (j - j' - 1) · insertCost)
        lastPosition[S[i]] = i

    return D[M][N]
```

Рис. 8 – Корректный алгоритм вычисления расстояния Дамерау-Левенштейна

Реализация этого алгоритма на C++ представлена алгоритмом 2.

Алгоритм 2 – Реализация алгоритма вычисления расстояния Дамерау-Левенштейна на C++

```
double DamerauLevenshteinDistance(const wstring &S, const wstring &T,
                                  vector<double> &work, dict<wchar, int> iwork,
                                  double deleteCost = 1,
                                  double insertCost = 1,
                                  double replaceCost = 1,
                                  double transposeCost = 0.8)
{
    int M = S.size(),
        N = T.size();

    // Обработка крайних случаев
    if (S == L"")
        if (T == L"")
            return 0;
        else
            return N;
    else if (T == L"")
        return M;

    work.resize((M + 2) * (N + 2));
    double *D = work.data();

#define ij2i(i, j) ((i) * (N + 2) + (j))

    double INF = (M + N) * max4(deleteCost, insertCost, replaceCost, transposeCost);

    // База индукции
    D[ij2i(0, 0)] = INF;

    int i, j;
    for (i = 0; i <= M; i++)
    {
        D[ij2i(i + 1, 1)] = i * deleteCost;
        D[ij2i(i + 1, 0)] = INF;
        for (j = 0; j <= N; j++)
        {
            D[ij2i(1, j + 1)] = j * insertCost;
            D[ij2i(0, j + 1)] = INF;
        }
    }

    int diff = 0;
    dict<wchar, int> &lastPosition = iwork;
    lastPosition.clear();

    for (i = 0; i < S.size(); i++)
    {
        if (std::find(T.begin(), T.end(), S[i]) != T.end())
        {
            lastPosition[S[i]] = 0;
        }
    }

    int last,
        i_, j_;
    for (i = 1; i <= M; i++)
    {
```

```
            last = 0;
            for (j = 1; j <= N; j++)
            {
                i_ = lastPosition[T[j - 1]];
                j_ = last;
                if (S[i - 1] == T[j - 1])
                {
                    D[ij2i(i + 1,j + 1)] = D[ij2i(i, j)];
                    last = j;
                }
                else
                {
                    D[ij2i(i + 1, j + 1)] = min3(D[ij2i(i, j)] + replaceCost,
                                                 D[ij2i(i + 1, j)] + insertCost,
                                                 D[ij2i(i, j + 1)] + deleteCost);
                }
                D[ij2i(i + 1, j + 1)] = min(D[ij2i(i + 1, j + 1)],
                                            D[ij2i(i_, j_)] +
                                            (i - i_ - 1) * deleteCost +
                                            transposeCost +
                                            (j - j_ - 1) * insertCost);
            }
            lastPosition[S[i - 1]] = i;
        }

        return D[ij2i(M, N)];

#undef ij2i
}
```

Здесь work – рабочее пространство, iwork – реализация словаря. Например #define dict std::unordered_map.

Окончательный алгоритм

Рассмотрим общую задачу. Дан запрос (не важно в какой форме), состоящий из множества терминов $T = \{T_0, \dots, T_m\}$ и множество терминов в корпусе $\{S_0, \dots, S_n\}$, для каждого термина определена его документная частота df. Требуется исправить возможные опечатки в терминах.

Предлагается решать эту задачу по алгоритму, представленному ниже.

Алгоритм 3 – Исправление опечаток в запросе

```
def(T, S, Corpus) -> [множество изменённых термов T]
{
    // если найдено документов больше чем threash1
    if len(search(T, Corpus)) >= threash1
        return T
    for(i = 0; i < m; i++) // цикл по терминам из запроса
    {
        if(df(T[i]) < thresh2) // документная частота термина меньше порога
        {
            min_dist = INFINITY
            max_df = 0
            for(j = 0; j < n; j++) // цикл по всем терминам в словаре
            {
                dist = theta * DamerauLevenshteinDistance(T[i], S[j]) +
                    (1-theta) * (-log10(df(S[j]) / len(Corpus)))
                if(dist < min_dist)
                {
                    min_dist = dist
                    max_df = df(S[j])
                }
                else if(dist == min_dist)
                {
                    if(max_df < df(S[j]))
                    {
                        max_df = df(S[j])
```

```
                    }
                }
            }
        }
        T[i] = [термин с минимальным расстоянием min_dist и частотой
            max_df]
    }
    return T
}
```

В алгоритме 3 учтено не только расстояние до терма, но и его документная частота с помощью взвешенного соотношения:

$$dist(T,S) = \theta \cdot \text{DamerauLevenshteinDistance}(T,S) + (1-\theta)\left(-\log_{10}\frac{df(S)}{|Corpus|}\right)$$

В практическое реализации алгоритма 3 цикл по терминам в словаре можно распараллелить (реализовано) и/или применить эвристику того, что опечатка редко бывает в первом символе (таким образом, можно ограничиться только терминами, начинающимися на одну букву с термином запроса)

3. Исходный код

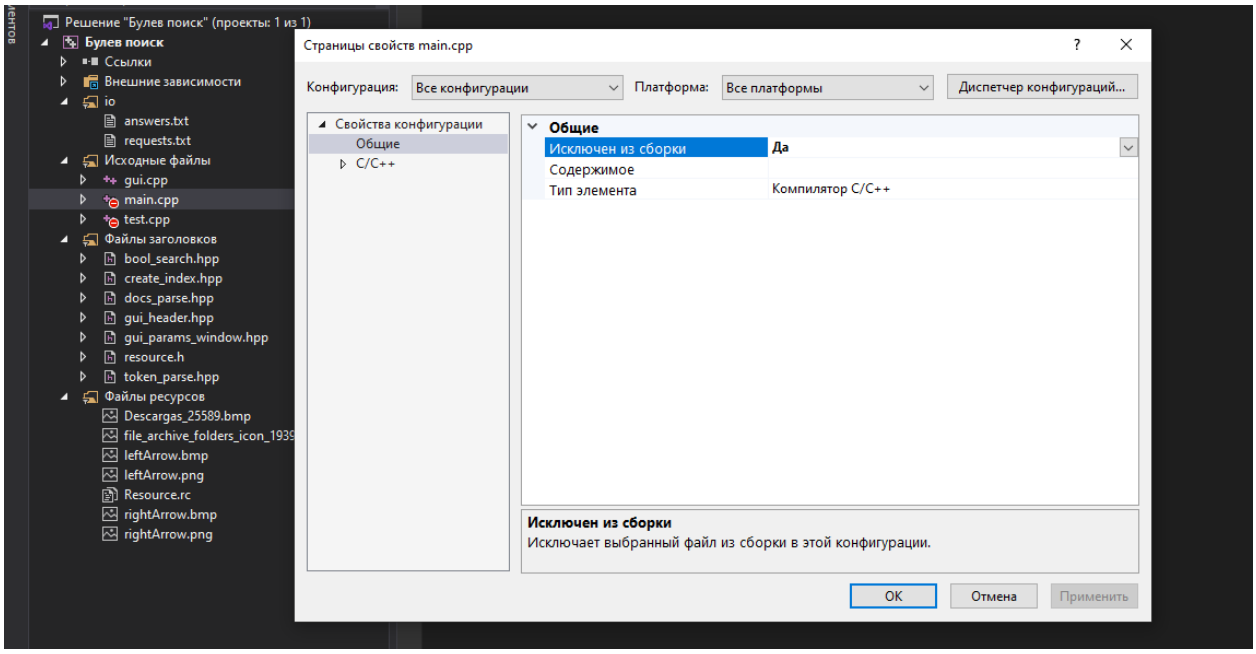
Структура проекта

- include
  - algebra.hpp (простейшие операции с векторами)
  - create_index.hpp (создание, чтение индекса)
  - defs.hpp (подключение внешних библиотек, макросы)
  - docs_parse.hpp (извлечение полей из корпуса)
  - gui_defs.hpp (подключение внешних библиотек, макросы, глобальные переменные)
  - gui_params_window.hpp (окно с выбором параметров)
  - resource.h (подключение изображений, иконок и прочего)
  - search.hpp (реализация всех видов поиска)
  - token_parse.hpp (функции для преобразования токенов в термы)
  - typos_correction.hpp (реализация исправления опечаток)
- python
  - lemmatizator.py (лемматизация документа)
  - lemmatizator_setup.py (компиляция lemmatizator.py в exe-файл)
  - request_parse.py (лемматизация запроса)
  - request_parse_setup.py (компиляция request_parse.py в exe-файл)
- io
  - anwers.txt
  - requests.txt
- src
  - gui.cpp (точка входа в оконный интерфейс)
  - main.cpp (точка входа в консольный интерфейс тестирования программы)
- resourses (файлы ресурсов для оконного приложения)

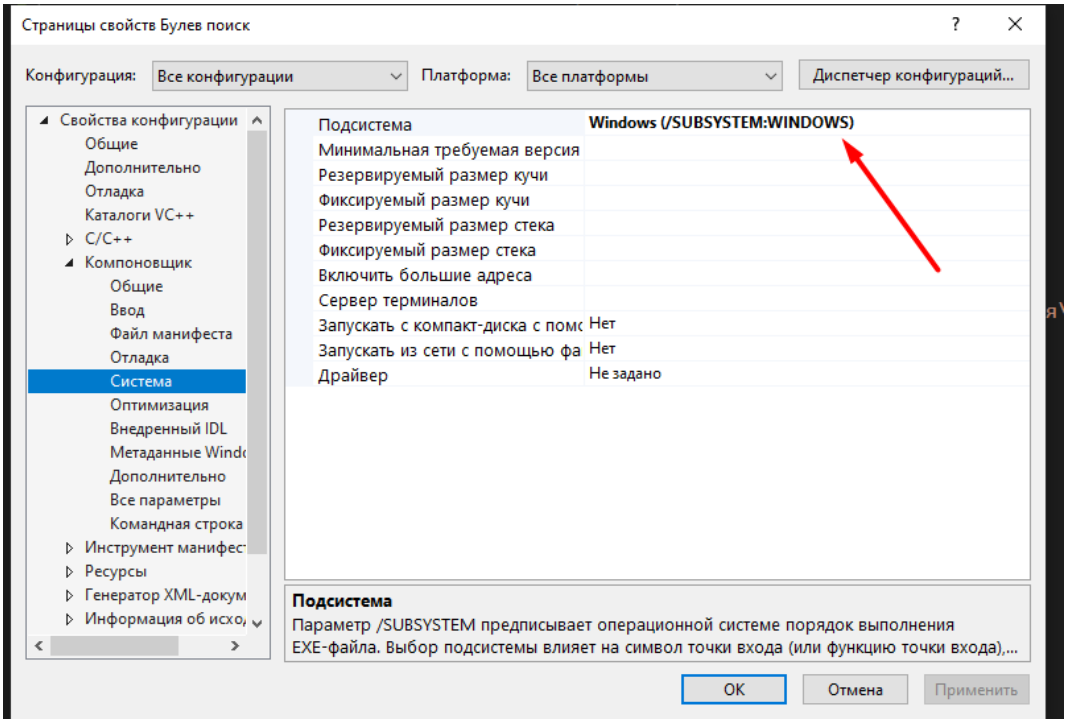
Проект был написан с помощью Microsoft Visual Studio 2019 эксклюзивно для ОС семейства Windows. Исходный код доступен по https://github.com/Stifeev/Information-retrieval/tree/main/Курсовой%20проект.

Запуск и сборка

Переключение между тремя точками входа осуществляется с помощью флага «исключить из сборки»:



Не забудь при переключении между консольными и оконными приложениями менять подсистему в настройке проекта:



Консольное приложение поддерживает флаги запуска:

- -i 'путь к корпусу'
- -o 'путь к индексу'
- -t 'путь к директории с блочным индексом'
- -m 'путь к директории с эталонами для метрик'
- -p кол-во_процессов_для_распараллеливания
- -create : создать блочный индекс
- -merge : выполнить слияние блочного индекса
- -clear : очистить папку с временными файлами после слияния
- -search : выполнить поиск
- -metric : высчитать метрики

Пример создания блочного индекса из корпуса:

```
$ ./prog.exe –p 4 –create -i "..\..\Корпус" -o -t "tmp"
```

Вывод

```
[INFO] Создание индекса для блоков

[INFO] Thread 0 processing block 1/13 : D:\Мои документы\Лабы и рефераты\5 курс 1 семестр\Информационный поиск\Корпус\films13.txt

[INFO] Thread 1 processing block 2/13 : D:\Мои документы\Лабы и рефераты\5 курс 1 семестр\Информационный поиск\Корпус\films9.txt

[INFO] Thread 2 processing block 3/13 : D:\Мои документы\Лабы и рефераты\5 курс 1 семестр\Информационный поиск\Корпус\films10.txt

[INFO] Thread 3 processing block 4/13 : D:\Мои документы\Лабы и рефераты\5 курс 1 семестр\Информационный поиск\Корпус\films12.txt

[INFO] Block 1 has 232216 terms

<…>

[INFO] Block 10 has 597482 terms

[INFO] Block 11 has 669497 terms

[INFO] Block 12 has 921883 terms

[INFO] Block 13 has 1383148 terms

[INFO] Создание очередей термов:   13 блок из   13

[INFO] Слияние docs_id:   13 блок из   13
```

```
[INFO] Слияние слопозиций термов
[INFO] Осталось термов:       0
[INFO] Очистка временных файлов
[INFO] Общее число термов в словаре = 2809203
Время выполнения = 145,5 sec, размер корпуса = 2,899 Gb, документов = 186109
Средняя скорость на документ = 0,782 ms
Средняя скорость на килобайт = 0,048 ms
```

Пример слияние блочного индекса:

```
$ ./prog.exe –p 4 –merge –clear –i "..\..\Корпус_index" -t "tmp"
```

Вывод

```
[INFO] Слияние блочного индекса
[INFO] Создание очередей термов:   13 блок из   13
[INFO] Слияние docs_id:   13 блок из   13
[INFO] Слияние слопозиций термов
[INFO] Осталось термов:            0
[INFO] Общее число термов в словаре = 1908410
Документов = 186109
[INFO] Время на слияние блочного индекса: 35 sec
[INFO] Вычисление статистики
Первый проход. Термов осталось:            0
Второй проход. Документов осталось:            0
[INFO] Вычисление статистики закончено
```

Пример просчёта метрик:

```
$ ./prog.exe –p 4 -metric -i "..\..\Корпус" -o "..\..\Корпус_index"
-m "..\..\Корпус_metric"
```

Вывод

```
Чтение термов : [##############################]      1908410/1908410
[INFO] Загружено 1908410 термов
Запрос: ' "тихое место 2" '
По запросу: ' " тихий место 2 " ' найдено 50 документов
Запрос: ' аватар скачать '
По запросу: ' !! аватар скачать ' найдено 474 документов
Запрос: ' "re zero" / 4 '
По запросу: ' " re zero " / 4 ' найдено 266 документов
Запрос: ' режиссёр назад в будущее '
По запросу: ' !! режиссер назад в будущее ' найдено 6233 документов
Запрос: ' фильм для интеллектуалов '
По запросу: ' !! фильм для интеллектуал ' найдено 1286 документов
Запрос: ' фильм сериал с самым большим рейтингом '
По запросу: ' !! фильм сериал с самый больший рейтинг ' найдено 4820
документов
Запрос: ' фильмы Макото Синкая '
По запросу: ' !! фильм макото синкай ' найдено 271 документов
Запрос: ' лучшие фильмы квентина тарантино '
По запросу: ' !! хороший фильм квентин тарантино ' найдено 1266
документов
Запрос: ' как звали главного героя коносубы '
По запросу: ' !! как звать главный герой коносуб ' найдено 85 документов
Запрос: ' самый лучший фильм '
По запросу: ' !! самый хороший фильм ' найдено 54490 документов
```

```
Запрос: ' сериалы с рейтингом 18+ '
По запросу: ' !! сериал с рейтинг 18 ' найдено 2061 документов
Запрос: ' высшая школа демонов '
По запросу: ' !! высокий школа демон ' найдено 1478 документов
Запрос: ' джокер '
По запросу: ' !! джокер ' найдено 1353 документов
Точность на уровне 30 = 0.241026
DCG на уровне 30 = 2.655877
nDCG на уровне 30 = 0.289893
ERR на уровне 30 = 0.732372
```

4. Выводы и результаты

$\theta = 0.7$, остальные параметры алгоритма подобраны таким образом, чтобы происходило автоисправление ($thresh_1 = 30, threash_2 = 30$).

Запрос	Исправленный запрос
самый && лучший && фиьлм	самый && лучший && фильм
красавый	красивый
висна	видно
весно	весна
пвлапопвап	полуподвал
друзь	друг
пачему	почему
полупокер	покупка
gjdfkgjkdfgjkdfjgdf	dfxgdfgdfgdfg
пекап	показ
юзернет	юзернейм
нормлвы	норма

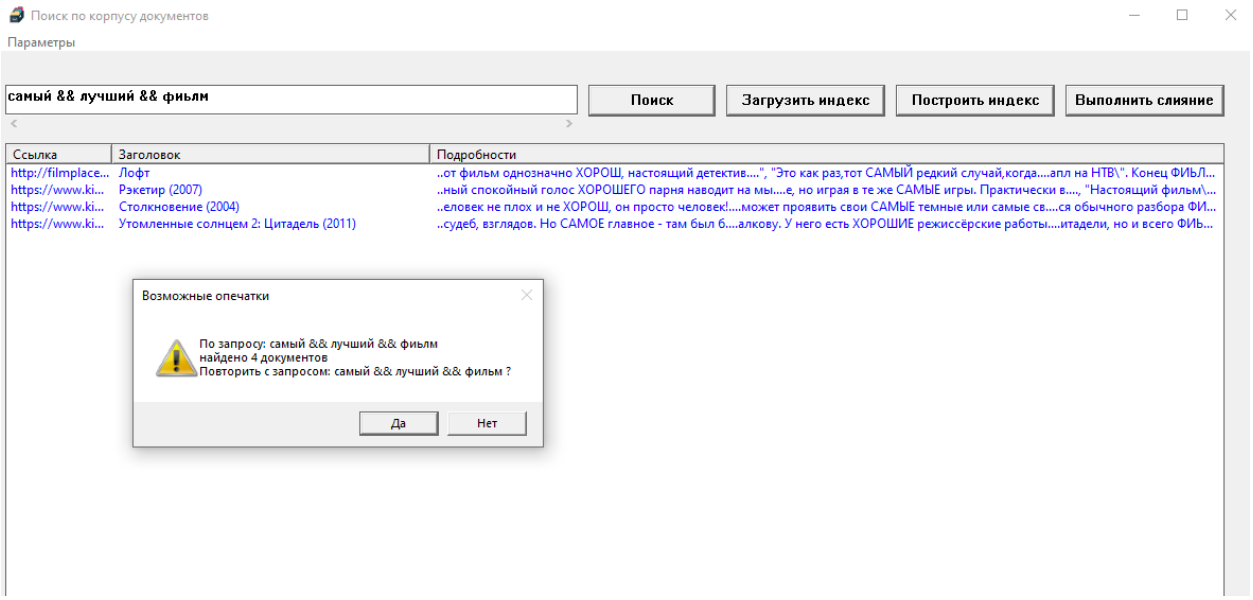


Рис. 9 – Реализация исправления опечаток в пользовательском интерфейсе (до исправления)

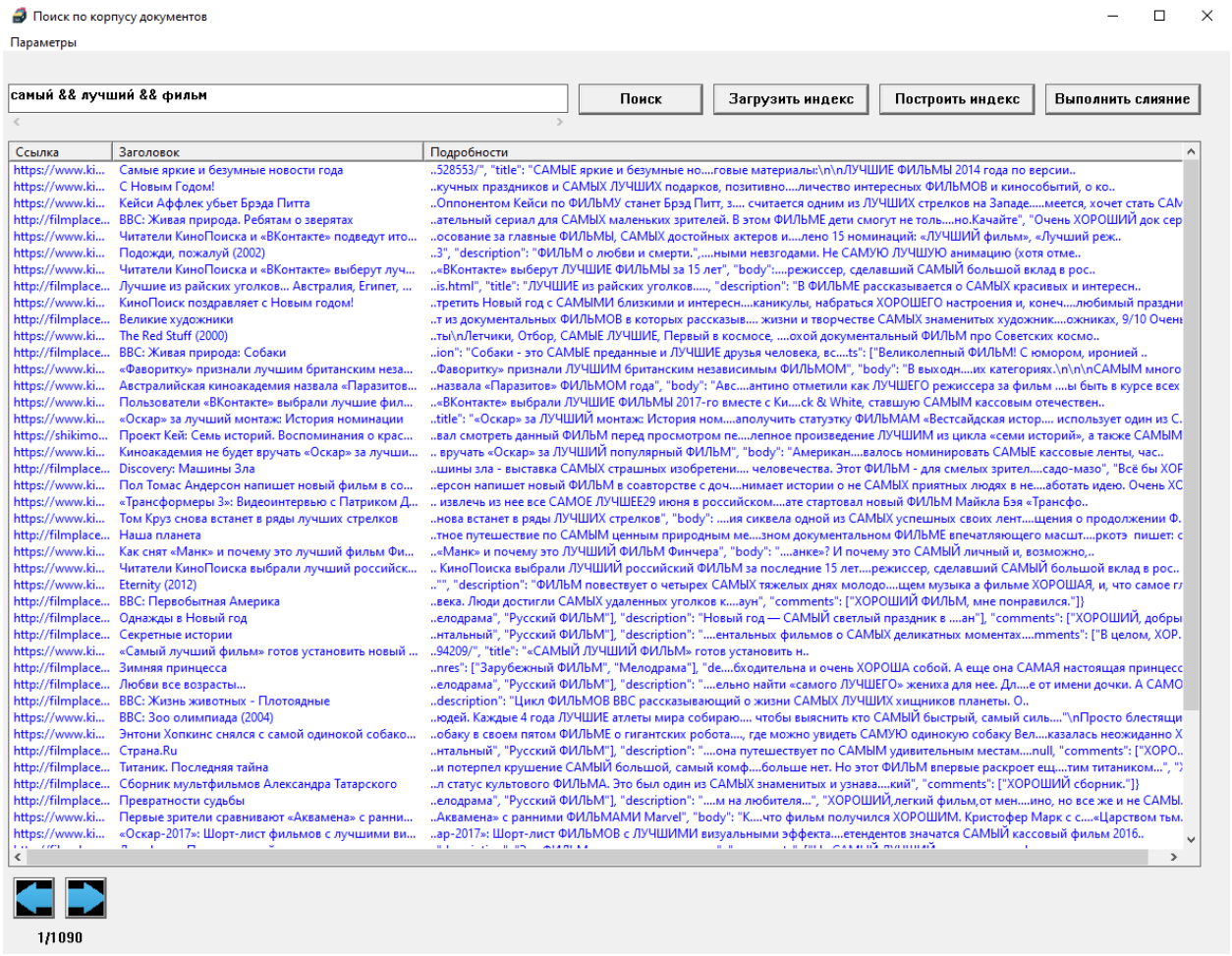


Рис. 10 Реализация исправления опечаток в пользовательском интерфейсе (после исправления)

По приведённым результатам можно сделать вывод, что автоисправление ошибок работает вполне корректно.

В ходе выполнения курсового проекта я научился выполнять автоисправление запросов к коллекции документов.

Литература

[1] https://natasha.github.io/

[2] https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_расстоянии_Дамерау-Левенштейна

[3] Кристофер Д.Маннинг, Прабхакар Рагхаван, Хайнрих Шютце. Введение в информационный поиск. 2020, изд. Вильямс.