Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы

по курсу «Информационный поиск»

Студент: И. С. Глушатов

Преподаватель: А. А. Кухтичев

Группа: М8О-407Б-19

Дата:

Оценка:

Подпись:

Москва, 2021

Лабораторная работа №1 «Добыча корпуса документов»

Необходимо подготовить корпус документов, который будет использован при выполнении остальных лабораторных работ:

- Скачать его к себе на компьютер. В отчёте нужно указать источник данных.
- Ознакомиться с ним, изучить его характеристики. Из чего состоит текст? Есть ли
- дополнительная метаинформация? Если разметка текста, какая она?
- Разбить на документы.
- Выделить текст.
- Найти существующие поисковики, которые уже можно использовать для поиска по выбранному набору документов (встроенный поиск Википедии, поиск Google с использованием ограничений на URL или на сайт). Если такого поиска найти невозможно, то использовать корпус для выполнения лабораторных работ нельзя!
- Привести несколько примеров запросов к существующим поисковикам, указать
- недостатки в полученной поисковой выдаче.
- В результатах работы должна быть указаны статистическая информация о корпусе:
- Размер «сырых» данных.
- Количество документов.
- Размер текста, выделенного из «сырых» данных.
- Средний размер документа, средний объём текста в документе.

1. Описание

Требуется выбрать источник документов для создания корпуса для последующих лабораторных работ.

Для своих лабораторных работ я выбрал сайт “Википедия”, так как для скачивания его статей есть собственный сервис, а также собственный поисковик.

Также для википедии есть хороший сервис PetScan, который позволяет находить статьи в википедии по тэгам. Я использовал рекурсивный поиск с заходом также на страницы, на которые переводят ссылки на в текущем документе. Язык был выбран русский, а формат выходного файла – csv.

Сначала я скачал таблицу с названиями статей и их URL с помощью этого сервиса. Затем написал программу на Питоне для скачивания всех этих документов, воспользовавшись сервисом Википедии: https://en.wikipedia.org/wiki/Special:Export.

Скачивание происходила долго, где-то 2-3 часа. Всего скачивалось порядка миллиона документов, однако в дальнейших лабораторных программа тестировалась только на 65 000 файлов из миллиона. В итоге я создал три различных по размеру корпуса, каждый из которых является подмножеством большего: из 966 857 файлов на 10.6Гб, 67 265 файлов на 1.5Гб и 349 файлов на 23.1 Мб. На последнем корпусе тестировался функционал, а на втором проверялась вся система.

2. Исходный код

Ниже приведена функция, осуществляющая скачивание самого корпуса документов. Для этого используется библиотека BeautifulSoup.

```
def create_corpus(filename, direct):
  count = 1
  with open(filename, newline='', encoding='utf-8') as csvfile:
    reader = csv.DictReader(csvfile, delimiter=',')

    pages = []

    for row in reader:
      if count % 1000 == 0:
        print(f"[{count}] done...")
      if count == 1000002:
        break
      if count > 537304:
        pages.append((row['title'], row['pageid'], row['touched']))

      if len(pages) == 35:

        url = 'https://ru.wikipedia.org/wiki/Служебная:Export?pages=' + "%0A".join(i[0] for i in pages)
        try:
          for page in pages: print(page)
          r = session.get(url)
          soup = BeautifulSoup(r.text, 'html.parser')
          l = soup.findAll('page')

          for i in range(len(l)):
            with open(direct + pages[i][1] + "_" + pages[i][2] + ".txt", "w", encoding='utf-8') as output:
              output.write(l[i].text)
        except Exception as inst:
          print(f"Bad request : {inst}")
        pages.clear()


      count += 1
```

3. Выводы

В данной лабораторной работе я вспомнил, как работать с HTPP запросами, подготовил корпус с документами для выполнения дальнейших лабораторных работ.

Лабораторная работа №3 «Булев индекс»

Требуется построить поисковый индекс, пригодный для булева поиска, по подготовленному в ЛР1 корпусу документов.
Требования к индексу:

- Самостоятельно разработанный, бинарный формат представления данных. Формат
- необходимо описать в отчёте, в побайтовом (или побитовом) представлении.
- Формат должен предполагать расширение, т.к. в следующих работах он будет меняться
- под требования новых лабораторных работ.
- Использование текстового представления или готовых баз данных не допускается.
- Кроме обратного индекса, должен быть создан «прямой» индекс, содержащий в себе как
- минимум заголовки документов и ссылки на них (понадобятся для выполнения ЛР4, при
- генерации страницы поисковой выдачи).
- Для термов должна быть как минимум понижена капитализация.
- В отчёте должно быть отмечено как минимум:
- Выбранное внутренне представление документов после токенизации.
- Выбранный метод сортировки, его достоинства и недостатки для задачи индексации.
- Среди результатов и выводов работы нужно указать:
- Количество термов.
- Средняя длина терма. Сравнить со средней длинной токена, вычисленной в ЛР1 по курсу
- ОТЕЯ. Объяснить причину отличий.
- Скорость индексации: общую, в расчёте на один документ, на килобайт текста.
- Оптимальна ли работа индексации? Что можно ускорить? Каким образом? Чем она
- ограниченна? Что произойдёт, если объём входных данных увеличится в 10 раз, в 100 раз,
- в 1000 раз?

1. Описание

Inverted_index.data						
Токен$_1$			…	**Токен$_n$**		
8 байт	8 байт	4 байт		8 байт	8 байт	4 байт
Хэш	Суммарная частота	Указатель на список документов		Хэш	Суммарная частота	Указатель на список документов

Inverted_index.data_ID							
Токен$_1$				…	**Токен$_n$**			
8 байт	8 байт	…	8 байт		8 байт	8 байт	…	8 байт
Количество документов	docID$_1$	…	docID$_{k1}$		Количество документов	docID$_1$	…	docID$_{kn}$

Выше представлены две таблицы с описанием формата обратного индекса. Первый файл является основным, каждое его поле содержит в себе хэш-представление слова, частоту его встречаемости во всём документе и указатель на место, по которому можно прочитать список документов, в котором это слово находится. Второй файл непосредственно хранит списки этих документов. Первый файл полностью будет считываться в оперативную память в hash-таблицу для быстрого поиска.

Straight_index.data					
Документ$_1$					
8 байт	8 байт	?	8 байт	4 байт	
docID$_1$	Размер заголовка	Заголовок	Количество слов	Указатель на текст	…

Straight_index.data_ID				
Документ$_1$				
8 байт	?		?	
Размер слова	Слово$_1$	…	Слово$_{k1}$	…

В двух последних табличках приведет формат бинарных данных для прямого индекса. По аналогии с обратным индексом первый файл полностью помещается в ОЗУ, а второй при необходимости используется в программе.

Далее описаны результаты для корпуса в 67к файлов.

В итоге размеры полученных индексов:

- Inverted_index.data – 30Кб
- Inverted_index.data_ID – 340Кб
- Straight_index.data – 3.5Кб
- Straight_index.data_ID – 1.8Гб

Количество получившихся термов 1.5 миллиона. Обработка заняла 12 минут, а средняя длина терма составила около 7 символов.

Так как итоговая реализация включает в себя доп. алгоритмы из следующих лабораторных, статистика может быть не объективна. Поэтому на данный момент среднее время обработки токена 2083ток/сек, что примерно 30Кб/сек – скорость токенизации.

Для сортировки использовалась сортировка слиянием на файлах. Её сложность O(nlogn) и она позволяет обрабатывать любой размер памяти, доступной на диске.

2. Исходный код

```
void tokenize(const std::string& name, std::set<TOKEN>& dictionary, size_t docID, FILE* straight_index,
FILE* straight_index_ID) {
  //size_t litva = hash_wstr(L"литва");

  std::wstring row;
  //std::wstring buffer = L"";

  FILE* input_file;
  OPENMACROS(fopen_s(&input_file, name.c_str(), "r, ccs=UTF-8"));

  wchar_t word[MAXSTRINGSIZE] = { L'\0' };
  size_t buffer_size = 0;
  wchar_t buffer[MAXSTRINGSIZE] = { L'\0' };

  /* TITLE READING */
  fgetws(word, MAXSTRINGSIZE, input_file);
  fgetws(word, MAXSTRINGSIZE, input_file);

  std::wstring title(word, wcslen(word) - 1);
  size_t title_size = title.size();
  long pointer = ftell(straight_index_ID);

  fwrite(&docID, sizeof(size_t), 1, straight_index);
  fwrite(&title_size, sizeof(size_t), 1, straight_index);
  fwrite(title.data(), sizeof(wchar_t), title_size, straight_index);


  size_t position = 0;
  while (fwscanf_s(input_file, L"%ls", word, MAXSTRINGSIZE) >= 0) {
    row = std::wstring(word);
    for (size_t i = 0; i < row.size(); i++) {
      wchar_t c = row[i];
      bool in_alpha = (alphabet.find(c) != alphabet.end());

      if (in_alpha) {
        buffer[buffer_size] = towlower(c);
        buffer_size++;
      }

      if ((!in_alpha or (i == row.size() - 1)) and buffer_size) {

        fwrite(&buffer_size, sizeof(size_t), 1, straight_index_ID);
        fwrite(buffer, sizeof(wchar_t), buffer_size, straight_index_ID);

        size_t hash = token_to_term(std::wstring(buffer, buffer_size));

        dictionary.insert({hash, docID, position});
```

```
            position++;
            buffer_size = 0;
        }
    }
  }

  fwrite(&position, sizeof(size_t), 1, straight_index);
  fwrite(&pointer, sizeof(long), 1, straight_index);

  fclose(input_file);
}

size_t process_files(const std::string& Path, const std::string& Out2, size_t& COUNT_OF_FILES) {
  std::set<TOKEN> dictionary;
  std::filesystem::path path(Path);
  std::filesystem::directory_iterator iter(path);
  std::string name;
  COUNT_OF_FILES = 0;
  size_t COUNT_OF_OUT_FILES = 0;
  FILE* file_with_tokens_c;


  FILE* straight_index, * straight_index_ID;
  OPENMACROS(fopen_s(&straight_index, Out2.c_str(), "wb, ccs=UNICODE"));
  OPENMACROS(fopen_s(&straight_index_ID, (Out2 + "_ID").c_str(), "wb, ccs=UNICODE"));

  while (!iter._At_end())
  {
    std::filesystem::directory_entry entry = *iter;
    name = entry.path().string();

    tokenize(name, dictionary, file_id(Path, name), straight_index, straight_index_ID);

    if (dictionary.size() > MAXPOCKETSIZE) {
      OPENMACROS(fopen_s(&file_with_tokens_c, (SORTINGPATH +
std::to_string(COUNT_OF_OUT_FILES) + ".data").c_str(), "wb, ccs=UNICODE"));
      for (auto& token : dictionary) {
        fwrite(&token, sizeof(TOKEN), 1, file_with_tokens_c);
      }
      fclose(file_with_tokens_c);
      dictionary.clear();
      COUNT_OF_OUT_FILES++;
    }

    COUNT_OF_FILES++;
    iter++;

    if (COUNT_OF_FILES % 1000 == 0) {
      std::cout << "[" << COUNT_OF_FILES << "] done...\n";
```

```
        }
    }

    fclose(straight_index_ID);
    fclose(straight_index);

    if (dictionary.size()) {
        OPENMACROS(fopen_s(&file_with_tokens_c, (SORTINGPATH +
std::to_string(COUNT_OF_OUT_FILES) + ".data").c_str(), "wb, ccs=UNICODE"));
        for (auto& token : dictionary) {
            fwrite(&token, sizeof(TOKEN), 1, file_with_tokens_c);
        }
        fclose(file_with_tokens_c);
        dictionary.clear();
        COUNT_OF_OUT_FILES++;
    }

    return COUNT_OF_OUT_FILES;
}

void merge_files(const std::string& path, size_t l, size_t r, size_t step = 0) {
    if (r == l) {
        if (std::rename((path + std::to_string(l) + ".data").c_str(),
            (path + "merge" + std::to_string(step) + ".data").c_str()))
        {
            std::perror("Error renaming");
            exit(-1);
        }
        return;
    }

    merge_files(path, l, (l + r) / 2, step * 2 + 1);
    merge_files(path, (l + r) / 2 + 1, r, step * 2 + 2);

    std::cout << "File " << l << " to " << r << " merging... on step " << step << std::endl;

    FILE* input1, * input2, * output;
    OPENMACROS(fopen_s(&input1, (path + "merge" + std::to_string(step * 2 + 1) + ".data").c_str(), "rb,
ccs=UNICODE"));
    OPENMACROS(fopen_s(&input2, (path + "merge" + std::to_string(step * 2 + 2) + ".data").c_str(), "rb,
ccs=UNICODE"));
    OPENMACROS(fopen_s(&output, (path + "merge" + std::to_string(step) + ".data").c_str(), "wb,
ccs=UNICODE"));

    TOKEN TI1, TI2;
    size_t n1, n2;

    n1 = fread(&TI1, sizeof(TOKEN), 1, input1);
    n2 = fread(&TI2, sizeof(TOKEN), 1, input2);
```

```
    do {
        if (n1 <= 0 and n2 <= 0) {
            break;
        }
        else if (n1 <= 0) {
            fwrite(&TI2, sizeof(TOKEN), 1, output);
            n2 = fread(&TI2, sizeof(TOKEN), 1, input2);
        }
        else if (n2 <= 0) {
            fwrite(&TI1, sizeof(TOKEN), 1, output);
            n1 = fread(&TI1, sizeof(TOKEN), 1, input1);
        }
        else {

            if (TI1 < TI2) {
                fwrite(&TI1, sizeof(TOKEN), 1, output);
                n1 = fread(&TI1, sizeof(TOKEN), 1, input1);
            }
            else {
                fwrite(&TI2, sizeof(TOKEN), 1, output);
                n2 = fread(&TI2, sizeof(TOKEN), 1, input2);
            }
        }

    } while (1);


    fclose(input1);
    fclose(input2);
    fclose(output);

    try {
        std::filesystem::remove((path + "merge" + std::to_string(step * 2 + 1) + ".data"));
        std::filesystem::remove((path + "merge" + std::to_string(step * 2 + 2) + ".data"));
    }
    catch (const std::filesystem::filesystem_error& err) {
        std::cout << "filesystem error: " << err.what() << '\n';
    }
}

void concatenate(const std::string& path, const std::string& out, const std::string& coords_out) {
    FILE *input, * output, *output_ID, * coords_output, * coords_output_ID, * coords_output_ID_POS;

    OPENMACROS(fopen_s(&input, path.c_str(), "rb, ccs=UNICODE"));
    OPENMACROS(fopen_s(&output, out.c_str(), "wb, ccs=UNICODE"));
    OPENMACROS(fopen_s(&output_ID, (out + "_ID").c_str(), "wb, ccs=UNICODE"));
    OPENMACROS(fopen_s(&coords_output, coords_out.c_str(), "wb, ccs=UNICODE"));
    OPENMACROS(fopen_s(&coords_output_ID, (coords_out + "_ID").c_str(), "wb, ccs=UNICODE"));
```

```
   OPENMACROS(fopen_s(&coords_output_ID_POS, (coords_out + "_ID_POS").c_str(), "wb,
ccs=UNICODE"));

   TOKEN last_TI, TI;
   size_t sum_freq = 0;
   std::set<size_t> fileIdes;
   std::vector<size_t> filePos;

   long prev_hash = -1;

   while (fread(&TI, sizeof(TOKEN), 1, input) > 0) {
      if (sum_freq == 0) {
         last_TI = TI;
         sum_freq++;
         fileIdes.insert(TI.docID);
         filePos.push_back(TI.pos);

         prev_hash = ftell(coords_output_ID);

         fwrite(&TI.hash, sizeof(size_t), 1, coords_output);
         fwrite(&prev_hash, sizeof(long), 1, coords_output);
         fwrite(&TI.hash, sizeof(size_t), 1, coords_output_ID);
      }
      else if (last_TI.hash == TI.hash and last_TI.docID == TI.docID) {
         sum_freq++;
         filePos.push_back(TI.pos);
      }
      else if (last_TI.hash == TI.hash) {
         sum_freq++;

         size_t filePos_size = filePos.size();
         long pointer_coords_output_ID_POS = ftell(coords_output_ID_POS);

         // COORDS FILE
         fwrite(&last_TI.docID, sizeof(size_t), 1, coords_output_ID);
         fwrite(&pointer_coords_output_ID_POS, sizeof(long), 1, coords_output_ID);

         fwrite(&filePos_size, sizeof(size_t), 1, coords_output_ID_POS);
         Compressor::compress(coords_output_ID_POS, filePos);
         //fwrite(filePos.data(), sizeof(size_t), filePos_size, coords_output_ID_POS);

         last_TI = TI;
         filePos.clear();
         fileIdes.insert(TI.docID);
         filePos.push_back(TI.pos);
      }
      else {
         size_t fileIdes_size = fileIdes.size();
         size_t filePos_size = filePos.size();
```

```
        long pointer_output_ID = ftell(output_ID);
        long pointer_coords_output_ID = ftell(coords_output_ID);
        long pointer_coords_output_ID_POS = ftell(coords_output_ID_POS);

        // INDEX FILE
        fwrite(&last_TI.hash, sizeof(size_t), 1, output);
        fwrite(&sum_freq, sizeof(size_t), 1, output);
        fwrite(&pointer_output_ID, sizeof(long), 1, output);

        fwrite(&fileIdes_size, sizeof(size_t), 1, output_ID);
        Compressor::compress(output_ID, fileIdes);
        /*for (auto& i : fileIdes) {
           fwrite(&i, sizeof(size_t), 1, output_ID);
        }*/

        // COORDS FILE
        fwrite(&last_TI.docID, sizeof(size_t), 1, coords_output_ID);
        fwrite(&pointer_coords_output_ID_POS, sizeof(long), 1, coords_output_ID);

        long temp = ftell(coords_output_ID);
        fseek(coords_output_ID, prev_hash, 0);
        fwrite(&fileIdes_size, sizeof(size_t), 1, coords_output_ID);
        fseek(coords_output_ID, temp, 0);

        fwrite(&filePos_size, sizeof(size_t), 1, coords_output_ID_POS);
        Compressor::compress(coords_output_ID_POS, filePos);
        //fwrite(filePos.data(), sizeof(size_t), filePos_size, coords_output_ID_POS);

        prev_hash = ftell(coords_output_ID);

        fwrite(&TI.hash, sizeof(size_t), 1, coords_output);
        fwrite(&prev_hash, sizeof(long), 1, coords_output);
        fwrite(&TI.hash, sizeof(size_t), 1, coords_output_ID);

        last_TI = TI;
        sum_freq = 1;
        fileIdes.clear();
        filePos.clear();
        fileIdes.insert(TI.docID);
        filePos.push_back(TI.pos);
    }
}

if (fileIdes.size()) {
    size_t fileIdes_size = fileIdes.size();
    size_t filePos_size = filePos.size();
    long pointer_output_ID = ftell(output_ID);
    long pointer_coords_output_ID = ftell(coords_output_ID);
    long pointer_coords_output_ID_POS = ftell(coords_output_ID_POS);
```

```
      // INDEX FILE
      fwrite(&last_TI.hash, sizeof(size_t), 1, output);
      fwrite(&sum_freq, sizeof(size_t), 1, output);
      fwrite(&pointer_output_ID, sizeof(long), 1, output);

      fwrite(&fileIdes_size, sizeof(size_t), 1, output_ID);
      Compressor::compress(output_ID, fileIdes);
      /*for (auto& i : fileIdes) {
        fwrite(&i, sizeof(size_t), 1, output_ID);
      }*/

      // COORDS FILE
      fwrite(&last_TI.docID, sizeof(size_t), 1, coords_output_ID);
      fwrite(&pointer_coords_output_ID_POS, sizeof(long), 1, coords_output_ID);

      long temp = ftell(coords_output_ID);
      fseek(coords_output_ID, prev_hash, 0);
      fwrite(&fileIdes_size, sizeof(size_t), 1, coords_output_ID);
      fseek(coords_output_ID, temp, 0);

      fwrite(&filePos_size, sizeof(size_t), 1, coords_output_ID_POS);
      Compressor::compress(coords_output_ID_POS, filePos);
      //fwrite(filePos.data(), sizeof(size_t), filePos_size, coords_output_ID_POS);

      fileIdes.clear();
      filePos.clear();
    }

    fclose(coords_output_ID_POS);
    fclose(coords_output_ID);
    fclose(coords_output);
    fclose(output_ID);
    fclose(output);
    fclose(input);
}
```

Токенизация разделена на несколько этапов. Функция `process_files` создает файлы с токенами и доп. информацией типа позиции и документа, где токен встретился. Функция `tokenize` обрабатывает один файл и создает по ходу прямой индекс. `merge_files` сортирует токены по хэшам, потом по документам, потом по позициям. `concatenate` занимается построением обратного индекса.

3. Выводы

В ходе данной лабораторной работе я разработал собственные форматы представления обратного и прямого индексов, написал сортировку слиянием для упорядочивания токен перед созданием обратного индекса. Построил обратный и прямой индекс. Для хэширования строк использовался встроенный в c++ алгоритм хэширования. Естественным образом могут возникать

коллизии в последующем поиске, поэтому этот недостаток несущественен в дальнейшей работе, однако в лучшем случае желательно однозначно представлять токены при хранении в обратном индексе. Скорость токенизации достаточно низкая, однако видимых способов его ускорить я не увидел, учитывая и так достаточно низкоуровневый способ обработки файлов. Сама программа получается сложно масштабируема, так как полноценная картина разработки программы для информационного поиска складывалась только по ходу работы и её объем был не ясен. К тому же были опасения, что слишком абстрактная программа начнет замедлять работу программы, не имея при этом и так достаточной скорости.

Лабораторная работа №4 «Булев поиск»

Нужно реализовать ввод поисковых запросов и их выполнение над индексом, получение поисковой выдачи.
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 должен быть устойчив к переменному числу пробелов, максимально толерантен к введённому поисковому запросу.
Примеры запросов:

- [московский авиационный институт]
- [(красный || желтый) автомобиль]
- [руки !ноги]

Для демонстрации работы поисковой системы должен быть реализован веб-сервис, реализующий базовую функциональность поиска из двух страниц:

- Начальная страница с формой ввода поискового запроса.
- Страница поисковой выдачи, содержащая в себе форму ввода поискового запроса, 50 результатов поиска в виде текстов заголовков документов и ссылок на эти документы, а также ссылку на получение следующих 50 результатов.

Так же должна быть реализована утилита командной строки, загружающая индекс и выполняющая поиск по нему для каждого запроса на отдельной строчке входного файла.
В отчёте должно быть отмечено:

- Скорость выполнения поисковых запросов.
- Примеры сложных поисковых запросов, вызывающих длительную работу.
- Каким образом тестировалась корректность поисковой выдачи.

1. Описание

Для реализации Булева поиска потребовалось написать собственный парсер, аналогичный парсеру арифметических выражений на основе перевода запроса в инфиксную форму с параллельным вычислением результата.

На данный момент программа работает в двух режимах – консольное приложение, которое запускается с флагом –console [обратный индекс] [прямой индекс] [координатный индекс]; и приложение-сервер, к которому можно присоединиться с помощью сокетов по определенному адресу, отправлять запросы и получать ответы, запускается с флагом --web и принимает те же аргументы. По дефолту программа запускается в консольном режиме.

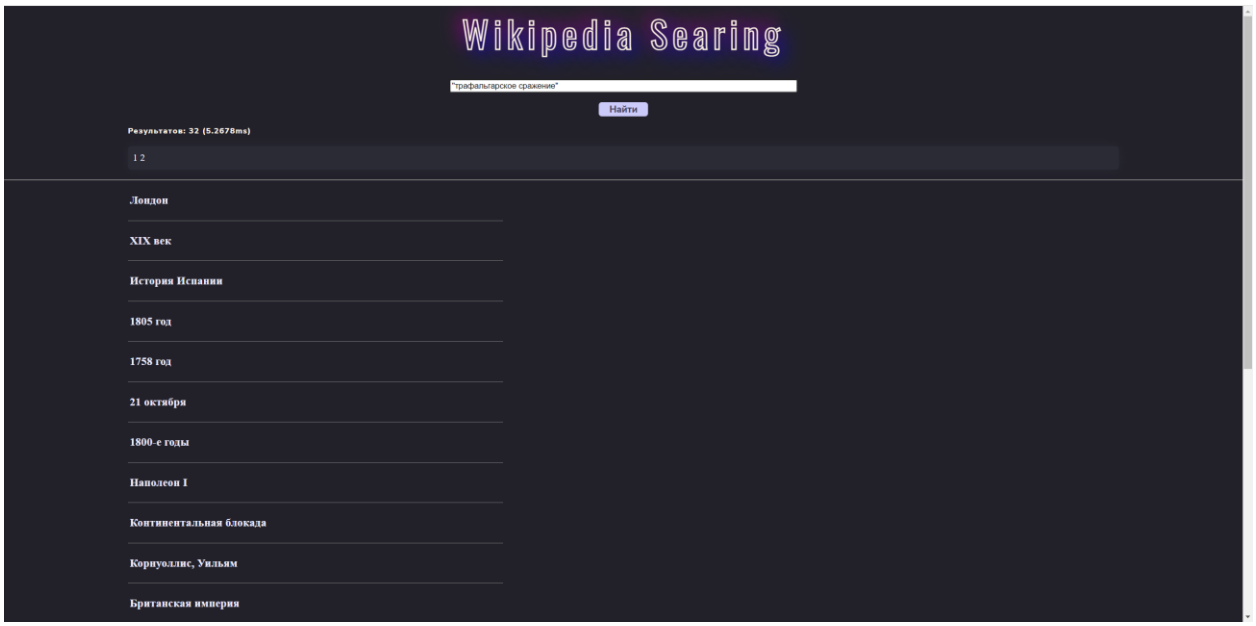
Сам булев поиск по очереди считывает символы и преобразует запрос в инфиксную запись. По ходу, если ситуация позволяет применить оператор И/ИЛИ, то вызывается соответствующая функция. Если оператор отсутствует, то считается, что пропущенный оператор является оператором И. Символы операторов & - И, | - ИЛИ. Скобки также допускаются для задания приоритетов операций.

Веб-сервис написан на Python с использованием фрэймворка Django. Программа подключается к серверу. Введенный в форме запрос отправляется по сокетам на сервер, дальше получает ответ и выводит 20 файлов. Данную константу можно изменить. По списку страниц можно получить другие результаты.

Тестирование проводилось на нескольких запросах. Ниже представлен отчёт по ним:

Запрос	Время (в ms)	Количество найденных файлов
русь	0.2753	2034
ленин	0.8437	1378
революция	0.3262	4436
война	0.6716	14604
победа	0.4739	2957
бунт	0.4671	654
русь & война	0.7741	977
ленин & революция	0.4458	622
(война | победа) & революция	1.2429	2877
война & бунт & русь	1.2017	56
победа | ленин | революция	0.8029	7569
(по | площади)	**2.3864**	**52163**
(из | под | тишка)	**4.3493**	**51480**
(из под в на)	**8.0795**	**27386**
(из | под | в | на)	**8.71**	**66152**

Из результатов видно, что обычные слова (их количество в запросе) мало влияют на скорость выдачи результата, однако стоп-слова, которые встречаются практически в каждом файле, увеличивают время поиска существенно. Корректность работы тестировалась вручную с помощью текстового поиска на странице википедии.
Вид веб-сервиса:



2. Исходный код

```
std::vector<size_t> parseBooling(std::wstring& query, const std::string& path,
        const std::string& straight_path, const std::string& coords_path,
        index_dictionary_type& dictionary, straight_dictionary_type& straight_dictionary,
        coords_dictionary_type& coords_dictionary)
{
        std::stack<ELEM> stack;
```

```
	std::vector<std::vector<size_t>> result;
	std::wstring buff = L"";

	bool last_token = false;
	bool citation = false;
	for (size_t i = 0; i <= query.size(); i++) {
		if (i == query.size()) { // Если достигли конца запроса
			if (buff.size()) {
				result.push_back(getDocs(path, dictionary, buff));
				buff = L"";
			}
			while (!stack.empty()) {
				ELEM temp = stack.top();
				stack.pop();

				if (result.size() >= 2) {
					std::vector<size_t> l, r;
					r = result[result.size() - 1];
					l = result[result.size() - 2];
					result.pop_back();
					result.pop_back();
					result.push_back(temp.type == TYPE::OPAND ? op_intersect(l, r)
: op_union(l, r));
				}
			}
		}
		else if (query[i] == L' ') { // Если встретился пробел
			if (buff.size()) {
				result.push_back(getDocs(path, dictionary, buff));
				buff = L"";
			}
		}
		else if (query[i] == L'(') { // Если встретилась открывающая скобка
			if (buff.size()) {
				result.push_back(getDocs(path, dictionary, buff));
				buff = L"";
			}
			if (last_token) { // & algo
				while (!stack.empty() and (stack.top().type == TYPE::OPAND or
stack.top().type == TYPE::OPOR)) {
					ELEM temp = stack.top();
					stack.pop();

					if (result.size() >= 2) {
						std::vector<size_t> l, r;
						r = result[result.size() - 1];
						l = result[result.size() - 2];
						result.pop_back();
						result.pop_back();
```

```
                                                result.push_back(temp.type == TYPE::OPAND ?
op_intersect(l, r) : op_union(l, r));
                                        }
                                }

                                stack.push({ TYPE::OPAND });
                        }

                        last_token = false;
                        stack.push({ TYPE::OPENBRACKET });
                }
                else if (query[i] == L'&' or query[i] == L'|') { // Если встретился оператор и/или
                        last_token = false;
                        if (buff.size()) {
                                result.push_back(getDocs(path, dictionary, buff));
                                buff = L"";
                        }

                        while (!stack.empty() and (stack.top().type == TYPE::OPAND or stack.top().type
== TYPE::OPOR)) {
                                ELEM temp = stack.top();
                                stack.pop();

                                if (result.size() >= 2) {
                                        std::vector<size_t> l, r;
                                        r = result[result.size() - 1];
                                        l = result[result.size() - 2];
                                        result.pop_back();
                                        result.pop_back();
                                        result.push_back(temp.type == TYPE::OPAND ? op_intersect(l, r)
: op_union(l, r));
                                }
                        }

                        stack.push({ query[i] == L'&' ? TYPE::OPAND : TYPE::OPOR });

                }
                else if (query[i] == L')') { // Если встретилась закрывающая скобка
                        last_token = true;
                        if (buff.size()) {
                                result.push_back(getDocs(path, dictionary, buff));
                                buff = L"";
                        }
                        while (stack.top().type != TYPE::OPENBRACKET) {
                                ELEM temp = stack.top();
                                stack.pop();

                                if (result.size() >= 2) {
                                        std::vector<size_t> l, r;
```

```
                                        r = result[result.size() - 1];
                                        l = result[result.size() - 2];
                                        result.pop_back();
                                        result.pop_back();
                                        result.push_back(temp.type == TYPE::OPAND ? op_intersect(l, r)
: op_union(l, r));
                                }
                        }
                        stack.pop();
                }
                else if (query[i] == L'\"') {
                        if (buff.size()) {
                                result.push_back(getDocs(path, dictionary, buff));
                                buff = L"";
                        }
                        if (last_token) { // & algo
                                while (!stack.empty() and (stack.top().type == TYPE::OPAND or
stack.top().type == TYPE::OPOR)) {
                                        ELEM temp = stack.top();
                                        stack.pop();

                                        if (result.size() >= 2) {
                                                std::vector<size_t> l, r;
                                                r = result[result.size() - 1];
                                                l = result[result.size() - 2];
                                                result.pop_back();
                                                result.pop_back();
                                                result.push_back(temp.type == TYPE::OPAND ?
op_intersect(l, r) : op_union(l, r));
                                        }
                                }

                                stack.push({ TYPE::OPAND });
                        }

                        i++;
                        while (query[i] != L'\"') {
                                buff += query[i];
                                i++;
                        }

                        if (buff.size()) {
                                result.push_back(parseCitation(path, coords_path, dictionary,
coords_dictionary, buff));
                                buff = L"";
                        }

                        last_token = true;
                }
```

```
                else { // Если встретилась буква/символ
                        if (last_token and (buff.size() == 0)) { // & algo
                                while (!stack.empty() and (stack.top().type == TYPE::OPAND or
stack.top().type == TYPE::OPOR)) {
                                        ELEM temp = stack.top();
                                        stack.pop();

                                        if (result.size() >= 2) {
                                                std::vector<size_t> l, r;
                                                r = result[result.size() - 1];
                                                l = result[result.size() - 2];
                                                result.pop_back();
                                                result.pop_back();
                                                result.push_back(temp.type == TYPE::OPAND ?
op_intersect(l, r) : op_union(l, r));
                                        }
                                }

                                stack.push({ TYPE::OPAND });
                        }

                        last_token = true;
                        buff += query[i];
                }
        }

        return result.size() == 1 ? result[0] : std::vector<size_t>();
}
```

Две первые функции, как понятно из названия реализуют алгоритмы пересечения и объединения списков документов. Последняя функция реализует разбор запроса, в ходе которого ищутся нужные документа.

3. Выводы

В ходе данной лабораторной работе я реализовал булев поиск с использованием индексных файлов, созданных в прошлой лабораторной работы. Среднее время поиска в 0.8ms я считаю отличным результатом. Из недостатков я могу выделить алгоритм пересечения и объединения для булева поиска. Возможно если не вытаскивать заранее списки документов, а делать это походу считывания из файла, то возможно еще на немного ускорить поиск, однако даже в таком виде я удовлетворен работой алгоритма. Самым сложным оказалось делать веб-сервис, так как пришлось разработать формат передачи данных по сокетам. Из минусов веб-сервиса можно выделить то, что программа передает сразу все найденные документы программе-клиенту, что заставляет передавать по сокету большой объем данных продолжительное количество времени. В дальнейшем последний запрос храниться на стороне клиента и не требует затрат при переходе по страницам, что можно назвать плюсом, хотя такое решение выглядит сомнительным. Однако я решил, что хранить запрос на сервере или заново производить поиск будет неправильно и затратно, поэтому я решил оставить такой вариант. Также в лабораторной работе я не реализовал отрицание.

Лабораторная работа №5 «Поиск цитат, координатный индекс»

В этом задании необходимо расширить язык запросов булева поиска новым элементом – поиском цитат. 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»]

Для реализации цитатного поиска нужно использовать координатный индекс, т. е. для каждого вхождения термина в документ построить и сохранить список позиций внутри документа, где этот термин встречался.

В отчёте нужно описать формат координатного индекса. Привести статистические данные:

- Размер получившегося индекса.
- Время построения индекса.
- Общее количество позиций. Среднее количество позиций на термин и на пару термин-документ.
- Скорость индексации (кб входных данных в секунду)
- Время выполнения поисковых запросов.
- Примеры долго выполняющихся запросов.

Кроме того, нужно привести примеры запросов и результаты их выполнения. В выводах должны быть указаны недостатки работы, приведены примеры их решения. Что можно сделать, чтобы ускорить «долгие» запросы?

1. Описание

<table>
<tr><td colspan="5">Coords_index.data</td></tr>
<tr><td colspan="2">Токен$_1$</td><td rowspan="3">…</td><td colspan="2">Токен$_n$</td></tr>
<tr><td>8 байт</td><td>4 байт</td><td>8 байт</td><td>4 байт</td></tr>
<tr><td>Хэш</td><td>Указатель на список документов</td><td>Хэш</td><td>Указатель на список документов</td></tr>
</table>

Coords_index.data_ID						
Токен$_1$						
8 байт	8 байт	4 байт		8 байт	4 байт	
Количество документов	docID$_1$	Указатель на список позиций	...	docID$_k$	Указатель на список позиций	...

Coords_index.data_ID_POS								
Токен$_1$								
Документ$_1$					**Документ$_m$**			
8 байт	8 байт		8 байт		8 байт	8 байт		8 байт
Количество позиций	pos$_1$	...	pos$_{k1}$	...	Количество позиций	pos$_1$	...	pos$_{k2}$

Выше представлен формат координатного индекса. Первый файл загружается в ОЗУ. Для цитатного поиска для каждого нужного слова вытаскивается список документов, где это слово встречается вместе с указателями, где хранятся позиции слова, встречаемого в определенном документе. Последний файл данные позиции непосредственно хранит. Информация из второго файла хранится в ОЗУ только в момент запроса, информация из третьего файла же вытаскивается во время пересечения и занимает O(const) память.

Алгоритм, учитывающий вкрапления других слов в цитатном поиске, был изменён и имеет другой синтаксис. Общий вид запросов: “слово1 [|k1] слово2 [|k2] … словоN”. Алгоритм не ищет цитату так, чтобы можно было указать расстояние именно между первым и последним словом. Вместо этого предлагается искать цитату с вкраплениями между каждой последовательной парой слов. Если расстояние не указано, то оно подразумевается равным единице. Т. е. запрос “кит |3 рыба блюдо” будет подразумевать, что между словами “кит” и “рыба” может быть до 2 других слова, а после “рыба” обязано идти слово “блюдо”. Цитата так же участвует в булевом поиске и может компоноваться с другими выражениями. В данном случае цитата представляется как отдельный терм с особым алгоритмом поиска документов, где она встречается.

Размеры файлов:

- Coords_index.data - 17.5Мб
- Coords_index.data_ID - 506.8Мб
- Coords_index.data_ID_POS - 1Гб

Количество получившихся позиций около 91.5 миллиона. Среднее кол позиций на термин – 61. На термин-документ – 2. Время построения индекса учесть не смог в силу того, что он строится параллельно с обратным индексом.

Тестирование цитатного поиска проводилось на нескольких запросах. Ниже представлен отчёт по ним:

Запрос	Время (в ms)	Количество найденных файлов
"советская республика"	68.8086	505
"географическое положение"	39.5157	939
"битва при Аустерлице"	**101.305**	**50**
"трафальгарское сражение"	4.884	32
"по площади"	**138.031**	**1389**
"война и мир"	**276.777**	**252**
"война |2 мир"	64.2987	263
"по |4 площади"	**135.347**	**1547**
"данные |20 участники"	24.462	10
"сражение на реке" & (Калка | Калке)	**159.712**	**7**
"индустриальная эпоха" | "серебряный век"	38.3371	149

Как видно, цитатный поиск обходится намного дороже по времени в сравнение с обычным булевским. Теперь в среднем поиск происходит за чуть меньше, чем 95ms. Цитаты, включающие в себя стоп-слова, сильно замедляют цитатный поиск, что связано с большим количеством позиций, где они встречаются, а следовательно, и более длительной работы алгоритма пересечения. Интересно также заметить, что запрос “война |2 мир” выполняется в 4.3 раза быстрее “война и мир”, однако количество найденных файлов не сильно отличается. Кажется, что выгоднее делать запросы с пропуском стоп-слов, практически не изменяя при этом результат вывода.

2. Исходный код

```
std::vector<CompareType> op_intersect_pro(const CORPUS& Corpus, size_t word1_hash, size_t word2_hash,
	std::vector<CompareType>& l, std::vector<CompareType>& r, size_t k)
{
	std::set<CompareType> result;

	size_t i = 0, j = 0;

	term_docs_type docs1;
	term_docs_type docs2;

	getDocPos(Corpus, word1_hash, docs1);
	getDocPos(Corpus, word2_hash, docs2);

	FILE* pp1, *pp2;
	OPENMACROS(fopen_s(&pp1, (Corpus.coords_path + "_ID_POS").c_str(), "rb, ccs=UNICODE"));
	OPENMACROS(fopen_s(&pp2, (Corpus.coords_path + "_ID_POS").c_str(), "rb, ccs=UNICODE"));

	while (i < l.size() && j < r.size()) {
		if (l[i] == r[j]) {

			size_t size1, size2;

			getPos(Corpus, pp1, docs1, l[i].doc, size1);
```

```
                        getPos(Corpus, pp2, docs2, r[j].doc, size2);

                        size_t ii = 0, jj = 0;
                        size_t pos_ii, pos_jj;

                        pos_ii = Compressor::read_pos(pp1);
                        pos_jj = Compressor::read_pos(pp2);

                        while (ii < size1) {
                                while (jj < size2) {
                                        if (0 < (pos_jj - pos_ii) and (pos_jj - pos_ii) <= k) {
                                                result.insert(l[i] + r[j]);
                                        }
                                        else {
                                                if (pos_jj > pos_ii)
                                                        break;
                                        }

                                        pos_jj += Compressor::read_pos(pp2);
                                        jj++;
                                }

                                pos_ii += Compressor::read_pos(pp1);
                                ii++;
                        }

                        i++;
                        j++;
                }
                else if (l[i] < r[j]) {
                        i++;
                }
                else {
                        j++;
                }
        }

        fclose(pp1);
        fclose(pp2);

        return std::vector<CompareType>(result.begin(), result.end());
}

std::vector<CompareType> parseCitation(std::wstring& citation, const CORPUS& Corpus)
{
        std::wstring last_word;
        std::stack<std::vector<CompareType>> result;
        std::wstring buff = L"";
```

```
	int i = 0;
	size_t k = 1;
	while (i <= citation.size()) {
		if ((i == citation.size()) or (citation[i] == ' ')) {
			if ((buff.size() != 0) and (result.size() == 0)) {
				last_word = buff;
				result.push(getDocs(Corpus, buff));
				buff = L"";
			}
			if (buff.size()) {
				std::vector<CompareType> a1 = result.top();
				result.pop();
				std::vector<CompareType> a2 = getDocs(Corpus, buff);
				size_t word1_hash = token_to_term(last_word);
				size_t word2_hash = token_to_term(buff);

				result.push(op_intersect_pro(Corpus, word1_hash, word2_hash, a1, a2,
k));
				k = 1;
				last_word = buff;
				buff = L"";
			}
		}
		else if (citation[i] == '|') {
			k = 0;
			i++;
			while ('0' <= citation[i] and citation[i] <= '9') {
				k *= 10;
				k += (size_t)(citation[i] - '0');
				i++;
			}
			i--;
		}
		else {
			buff += citation[i];
		}
		i++;
	}

	return result.size() == 1 ? result.top() : std::vector<CompareType>();
}
```

Из представленного кода первая функция выполняет алгоритм пересечения документов при цитатном поиске. Вторая функция выполняет разбор самой цитаты подобно булевому поиску, основанном на разборе арифметического выражения.

3. Выводы

В ходе данной лабораторной работе я разработал бинарный формат координатного индекса, построил его и использовал для цитатного поиска. Суммарный размер координатного индекса составил 1.56Гб, что не очень приятно. Составление координатного индекса увеличило время токенизации, так как потребовало частые перемещения файлового указателя вперед-назад. Время поиска цитат увеличилось. По созданному формату координатного индекса смысл обратного индекса пропадает, так как документы coords_index.data и coords_index_data_ID полностью включают в себя обратный индекс, плюс храниться дополнительный указатель на позиции слов. Также я изменил смысл вкрапления слов в цитате, однако кажется, что данная форма позволяет гибче задавать цитату и исключать при желании вкрапления между определенными словами. Проблема цитатного поиска в моей реализации также заключается в том, что полная читата в документе может не найтись, так как де-факто документ обязан включать только все пары рядом стоящих слов в цитате на определенном расстоянии.

Лабораторная работа №6 «Сжатие»

В этом задании необходимо применить алгоритмы сжатия к координатным блокам. Исследовать изменения в размерах частей индекса, влияние на скорость индексации и поиска.
В отчёте нужно указать:

- Выбранный метод сжатия. Привести побитовую схему хранения данных в индексе. Описать причины, по которым был выбран именно этот метод сжатия.
- Влияние сжатия на размер и скорость прохождения по координатным блокам всех терминов, редких терминов, терминов средней частотности и высокочастотных терминов.
- Обосновать, почему поиск после внедрения сжатия работает корректно. Как производилось тестирование?

1. Описание

В данной лабораторной в качестве алгоритма сжатия я выбрал variable byte code (VB-код), так как он простой, понятный и красивый. Для этого я реализовал отдельный класс Compressor со статическими функциями, которые принимают файловый указатель и выполняют сжатие, декомпрессию как целого списка чисел, так и каждого элемента по отдельности.

Inverted_index.data_ID				
Токен$_1$			**Токен$_n$**	
8 байт	?		8 байт	?
Количество документов	Сжатые данные (docID)	…	Количество документов	Сжатые данные (docID)

Coords_index.data_ID_POS					
Токен$_1$					
Документ$_1$			**Документ$_m$**		
8 байт	?		8 байт	?	
Количество позиций	Сжатые данные (pos)	…	Количество позиций	Сжатые данные (pos)	…

Выше представлены измененные форматы представления обратного и координатного индексов.

Текущий размер файлов:

- inverted_index.data_ID - 76Мб (вместо 341Мб)
- coords_index.data_ID_POS - 472Мб (вместо 1Гб)

Таким образом размер обратного индекса уменьшился в 4.5 раза, а координатного в 2 раза. Общий объем обратного уменьшился в 3.5 раза, а координатного в 1.5 раза. Скорость поиска при этом увеличилось, а скорость токенизации выросло на 2 минуты.

Тестирование поиска производилось вручную и в соответствии с результатами поиска без сжатия (совпадало ли количество полученных файлов или нет).

2. Исходный код

```
#include "Compressor.h"

size_t Compressor::s(size_t n) {
  return n == 0 ? 0 : 1 + s(n >> 1);
}

void Compressor::write_pos(FILE* file, size_t n) {
  char byte = 0;

  size_t sn = s(n);

  while (sn > 7) {
    byte |= (n >> ((sn - 1) / 7 * 7));
    fwrite(&byte, sizeof(char), 1, file);

    byte = 0;
    n &= (1ull << ((sn - 1) / 7 * 7)) - 1ull;
    sn -= 7;
  }

  byte |= 0x80ull | n;
  fwrite(&byte, sizeof(char), 1, file);
}

void Compressor::compress(FILE* file, const std::vector<size_t>& poses) {
  if (poses.size() == 0) {
    throw std::exception("poses array is empty");
  }

  size_t last_element = poses[0];

  write_pos(file, last_element);

  for (size_t i = 1; i < poses.size(); i++) {
    if (poses[i] <= last_element) {
      throw std::exception("poses array is not sorted");
    }

    size_t dif = poses[i] - last_element;
    write_pos(file, dif);
```

```
        last_element = poses[i];
    }
}

void Compressor::compress(FILE* file, const std::set<size_t>& poses) {
    if (poses.size() == 0) {
        throw std::exception("docs is empty");
    }

    auto it = poses.begin();
    size_t last_element = *it;

    write_pos(file, last_element);

    it++;
    //for (size_t i = 1; i < poses.size(); i++) {
    for (; it != poses.end(); it++) {
        if (*it <= last_element) {
            throw std::exception("docs is not sorted");
        }

        size_t dif = *it - last_element;
        write_pos(file, dif);

        last_element = *it;
    }
}

size_t Compressor::read_pos(FILE* file) {
    char byte;
    size_t res = 0;

    fread(&byte, sizeof(char), 1, file);
    res = byte & 0x7F;

    while ((byte & 0x80) == 0) {
        res <<= 7;
        fread(&byte, sizeof(char), 1, file);
        res |= (byte & 0x7F);
    }

    return res;
}

void Compressor::decompress(FILE* file, std::vector<size_t>& poses, size_t size) {
    size_t last_element;

    last_element = read_pos(file);
```

```
    poses.push_back(last_element);

    for (size_t i = 1; i < size; i++) {
        size_t dif = read_pos(file);

        poses.push_back(last_element + dif);
        last_element = last_element + dif;
    }
}
```

Выше представлены функции, реализованные в классе Compress. Compress и decompress выполняют одноименные функции, принимая файл и контейнер с последовательностью, откуда будет браться информация или куда записываться. Write_pos и Read_pos выполняют естественно запись и чтение числа в сжатом виде.

3. Выводы

В ходе данной лабораторной работе я реализовал класс, выполняющий сжатие возрастающих последовательностей чисел, использовав VB - код. Данный алгоритм мне очень понравился и оказался достаточно эффективным для сжатия индексов. Также данная лабораторная работа позволила сильно ускорить поиск в несколько раз, однако точные цифры я привести не могу за неимением старого кода, где сжатия не было.

Лабораторная работа №8 «Ранжирование TF-IDF»

Необходимо сделать ранжированный поиск на основании схемы ранжирования TF-IDF. Теперь, если запрос содержит в себе только термины через пробелы, то его надо трактовать как нечёткий запрос, т.е. допускать 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 надо трактовать как булев, т.е. соответствие должно быть строгим, но порядок выдачи должен быть определён ранжированием TF-IDF. Например:

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

В отчёте нужно привести несколько примеров выполнения запросов, как удачных, так и не удачных.

1. Описание

Нечёткий поиск я реализовал как объединение всех документов, в которых находятся слова из запроса. Т. е. он происходил аналогично булевому, где вместо дефолтного оператора И используется оператор ИЛИ.
Сначала документу ищутся в соответствии с запросом, потом они передаются в функцию range, где происходит ранжирование TF-IDF по формуле:

$$w(i) = \sum_{j=0}^{N_{query}} tf - idf(i,j),$$

$$tf - idf(i,j) = \frac{n_{ij}}{N_i} log_{10}\left(\frac{N}{d_j}\right),$$

где:

- N_{query} – количество термов в запросе - вычисляется после запроса
- i – номер документа из полученной выборки - итератор во время ранжирования
- N – количество документов в корпусе - размер hash-таблицы прямого индекса
- N_i – количество слов в i'ом документе - достается из прямого индекса
- n_{ij} – количество вхождений j'ого токена в i'ый документ – количество позиций из координатного индекса
- d_j – количество документов с j'ым словом - количество документов из координатного индекса

В таком случае у нас есть все данные для того, чтобы производить ранжирование.

Запрос	Время (в ms)	Количество найденных файлов
ледовое побоище	2.8567	136
по площади	**102.558**	**52163**
война и мир	**162.001**	**64770**
завоевание Константинополя	6.7162	1543
византийская империя	21.9516	10351
дания история	**56.1356**	**27704**
ледовое & побоище	0.5652	43

по & площади	**91.6875**	**5230**
война & и & мир	**154.927**	**4703**
завоевание & Константинополя	3.1141	79
византийская & империя	20.6612	602
дания & история	**45.6381**	**927**

Выше приведена таблица с тестовыми запросами. Видно, что время для некоторых запросов со стоп-словами достаточно большое, так как практически каждый документ их включает. В целом нечёткий поиск не сильно замедляет поиск, только на некоторых запросах показывая в разы превосходящие результаты по времени. В принципе это нормально, учитывая, что мы тратим время на сортировку.

2. Исходный код

```
std::vector<CompareType> get_indexes(const CORPUS& Corpus, size_t hash, long pointer) {
	FILE* coords_index_ID, * coords_index_ID_POS;
	OPENMACROS(fopen_s(&coords_index_ID, (Corpus.coords_path + "_ID").c_str(), "rb,
ccs=UNICODE"));
	OPENMACROS(fopen_s(&coords_index_ID_POS, (Corpus.coords_path + "_ID_POS").c_str(), "rb,
ccs=UNICODE"));
	fseek(coords_index_ID, pointer, 0);

	size_t N = Corpus.straight_dictionary.size(); // N
	size_t size; // dj
	fread(&size, sizeof(size_t), 1, coords_index_ID);

	std::vector<CompareType> result(size);

	for (size_t i = 0; i < size; i++) {
		size_t docID;
		long pointer_pos;
		fread(&docID, sizeof(size_t), 1, coords_index_ID);
		fread(&pointer_pos, sizeof(long), 1, coords_index_ID);

		size_t Ni = Corpus.straight_dictionary[docID].count_of_words; // Ni
		size_t nij; // nij

		fseek(coords_index_ID_POS, pointer_pos, 0);
		fread(&nij, sizeof(size_t), 1, coords_index_ID_POS);

		double rank = static_cast<double>(nij) / static_cast<double>(Ni) *
			log10(static_cast<double>(N) / static_cast<double>(size));
		result[i] = {docID, rank};
	}

	fclose(coords_index_ID);
	fclose(coords_index_ID_POS);

	return result;
```

```
}

void range(std::wstring& query, std::vector<CompareType>& res, const CORPUS& Corpus) {
	if (res.empty()) { return; }

	std::sort(res.begin(), res.end(), [](const CompareType& a, const CompareType& b) {return a.rank
> b.rank; });
}
```

Выше приведена функция get_indexes, которая возвращает список документов для каждого терма, при этом считая TF-IDF метрику для каждого документа. range просто сортирует список документов по TF-IDF метрике. Пришлось немного переписать методы объединения и пересечения, чтобы учесть TF-IDF.

3. Выводы

В ходе данной лабораторной работе я реализовал ранжирование TF-IDF и нечёткий поиск. Данные задачи немного ухудшили время поиска, однако это не критично, учитывая, что даже сложные запросы возвращаются за менее чем 400ms. Качество поиска естественным образом улучшилось, однако наверняка сказать сложно. Все равно нужные документы оказываются чаще не на первых страницах поиска, однако их наличие хотя бы в первой десятке уже радует.

Лабораторная работа №2 «Закон Ципфа»

Для своего корпуса необходимо построить график распределения терминов по частотностям в логарифмической шкале, наложить на этот график закон Ципфа. Объяснить причины расхождения. В качестве дополнительного задания, можно (но необязательно) подобрать константы для закона Мандельброта, наложить полученный график на график распределения терминов по частотностям. Привести выбранные константы.

1. Описание

Для построения графика я использовал обратный индекс, сделанный в 3-й лабораторной работе. Написал программу, которая выводит в файл отсортированную последовательность частот. Затем gnuplot считывает этот файл и рисует график.

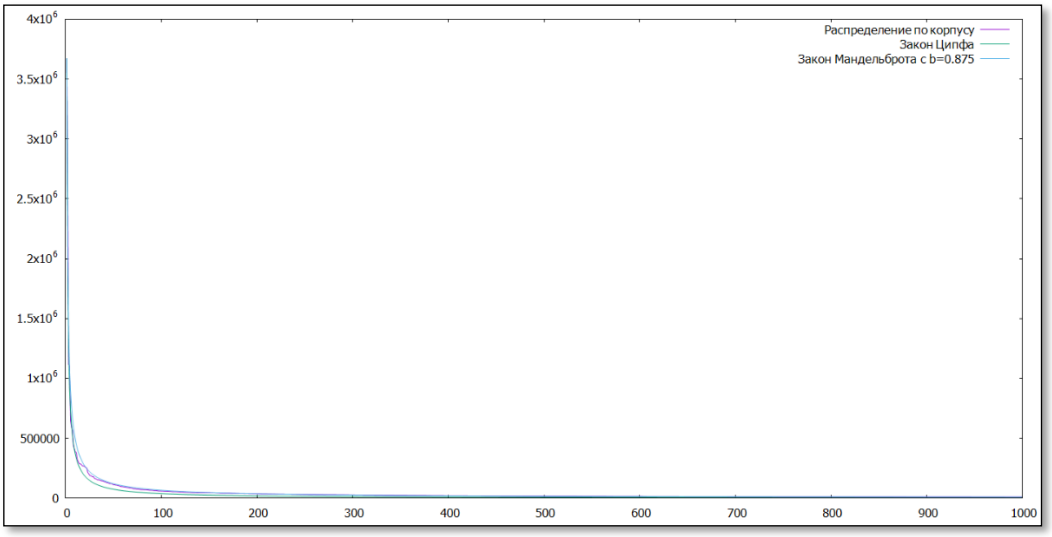


Рисунок 1. Распределение частот термов в отсортированном порядке

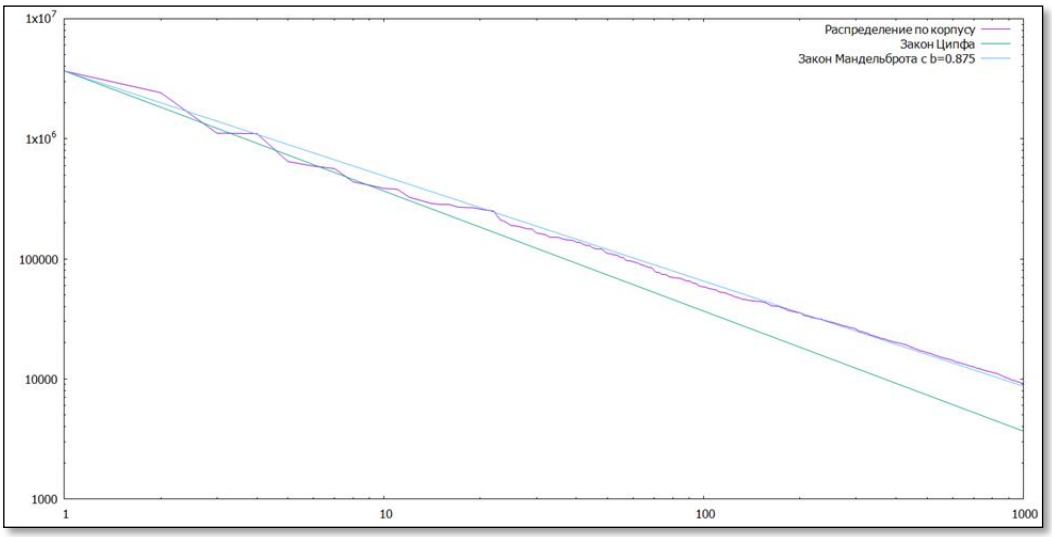


Рисунок 2. Распределение частот термов в отсортированном порядке в логарифмических координатах

Закон Ципфа утверждает, что зависимость частоты употребляемого слово обратно пропорционально зависит от ранга в отсортированном виде. Закон Мандельброта обобщает закон Ципфа, добавляя в формулу константы, при этом увеличивая гибкость.

В общем случае получается:

$$f(r) = f_{max}(r - v)^{-b}$$

В моём случае получается, что v = 0, b = 0.875. Данные значения очень близки к оригинальному закону Ципфа. И в принципе можно говорить, что корпус моих документов подчиняется этим законам.

2. Исходный код

```
int main()
{
	FILE* II, *out;
	OPENMACROS(fopen_s(&II, "C:/Users/Igor/Desktop/инфопоиск/inverted_index_light.data",
"rb, ccs=UNICODE")); //C:\Users\Igor\Desktop\инфопоиск\inverted_index_very_light.data
	OPENMACROS(fopen_s(&out, "out.txt", "w"));

	termInfo TI;

	vector<my_pair> v;
	while (fread(&TI, sizeof(termInfo), 1, II) > 0) {
		v.push_back({ TI.hash , TI.freq });
	}

	sort(v.begin(), v.end(), [](const my_pair& a, const my_pair& b) -> bool {return a.second >
b.second; });

	for (size_t i = 0; i < 1000; i += 1) {
		fprintf(out, "%llu %llu\n", i + 1, v[i].second);
		//fwrite(&v[i].second, sizeof(size_t), 1, out);
	}

	fclose(II);
	fclose(out);

  return 0;
}
```

Программа считывает данные из обратного индекса, сортирует их и выводит в файл.

plot 'C:\Users\Igor\source\repos\Contest\Contest\out.txt' using 1: 2 with line title "Распределение по корпусу", '' using 1:(3671835/$1) with line title "Закон Ципфа", '' using 1:(3671835*($1)**(-0.875)) with lines title "Закон Мандельброта с b=0.875" – Этой командой я выводил графики в GNUPlot.

3. Выводы

В ходе данной лабораторной работы я проанализировал свой корпус на удовлетворение его закону Ципфа и Мандельброта, а также подобрал для второго закона константы. В результате все получилось достоверно и хорошо.

Лабораторная работа №3 «Лемматизация»

Добавить в созданную поисковую систему (ЛР 1-8 по курсу «Информационный поиск») лемматизацию. В простейшем случае это просто поиск без учёта словоформ. В более сложном случае, можно давать бонус большего размера за точное совпадение слов. Лемматизацию можно добавлять на этапе индексации, можно на этапе выполнения поискового запроса.
В отчёте должна быть включена оценка качества поиска, после внедрения лемматизации. Стало ли лучше? Изучите запросы, где качество ухудшилось. Объясните причину ухудшения и как можно было бы улучшить качество поиска по этим запросам, не ухудшая остальные запросы?

1. Описание

Лемматизация – это процесс приведения слова к его нормальной форме. Для каждого языке данный процесс должен производиться по-разному. Направление, которое этим занимается, достаточно сложное, и даже продвинутые алгоритмы могут давать неточные результаты.
Для данной лабораторной работы я реализовал класс Lematization, который инкапсулирует процесс приведения слова к его начальной форме. В классе реализован статический метод normalization, который выполняет одноименную функцию. Сам алгоритм достаточно прост и не претендует на правильность. В простейшем случае в константных словарях храняться окончания для прилагательных и глаголов. В процессе токенизации помимо исходного слова в словарь также добавляется его форма без окончания. Таким образом для слов: “красивая”, “красивый”, “красивое”, “красив”, “красивым”, будет одна и та же приведенная форма – “красив”.
В процессе поиска для цитатного используется только его истинная форма; для булева и нечёткого же будут учитываться документы не только начального слова, но и документы с его нормальной формой. Они будут объединяться, а документы с нормальной формой получат меньший вес при поиске, нежели чем документы с введенным словом.

2. Исходный код

```
#include "Lematization.h"

namespace {
  static const std::unordered_set<std::wstring> ADJECTIVE = {
    L"ый", L"ому", L"ого", L"ым", L"ом",
    L"ий", L"ему", L"его", L"ою", L"ие",
    L"им", L"ей", L"ее",

    L"ое", L"о",

    L"ая", L"ой", L"ую", L"ою", L"а",

    L"ые", L"ых", L"ыми", L"ы",
  };

  static const std::unordered_set<std::wstring> VERB = {
    L"у", L"ишь", L"ит", L"им", L"ите",

    L"ят", L"я",
```

```
    L"ив", L"и", L"ить", L"ешь", L"ите",

    L"а",
  };
}

std::wstring Lematization::normalization(const std::wstring& word) {
  if (word.size() <= 3) {
    return word;
  }

  for (int k = 3; k > 0; k--) {
    if (ADJECTIVE.find(word.substr(word.size() - 1, k)) != ADJECTIVE.end()) {
      return word.substr(0, word.size() - k);
    }
  }

  for (int k = 3; k > 0; k--) {
    if (VERB.find(word.substr(word.size() - 1, k)) != VERB.end()) {
      return word.substr(0, word.size() - k);
    }
  }

  return word;
}
```

В коде представлена реализация метода, выполняющего нормализацию введенного слова. Он пытается отсекать как можно больший суффикс, и если он есть в словаре, то выводит перфикс, как нормализованную форму.

```
std::vector<CompareType> getDocs(const CORPUS& Corpus, std::wstring& buff, int mode = 1)
{
        std::vector<CompareType> result;

        size_t hash = token_to_term(buff);
        index_dictionary_type::iterator iter = Corpus.coords_dictionary.find(hash);
        if (iter != Corpus.coords_dictionary.end()) {
                result = get_indexes(Corpus, hash, iter->second);
        }

        if (mode == 1) {
                size_t normal_hash = token_to_term(Lematization::normalization(buff));
                index_dictionary_type::iterator normal_iter =
Corpus.coords_dictionary.find(normal_hash);
                if (normal_iter != Corpus.coords_dictionary.end()) {
                        result = op_union(result, get_indexes(Corpus, normal_hash, normal_iter-
>second, 0.1));
                }
        }
```

```
        return result;
}
```

Здесь приведен измененный метод getDocs, который учитывает для булевого и нечеткого поиска документы с нормализованной формой слова.

3. Выводы

В ходе данной лабораторной работы я добавил лемматизацию для алгоритмов токенизации и поиска. Количество найденных документов при булевом и нечётком поиске естественным образом увеличилось. Оценка качества поиска производилась вручную. В принципе кажется, что поиск чуть-чуть улучшился, однако, не имея чётких метрик, сложно говорить о реальных результатах. Скорость поиска особо не изменилась, а вот скорость токенизации возросло в полтора раза, дойдя до 16 минут. Размеры корпусов тоже изменились, но не сильно. Всё же реализованный алгоритм крайне прост. По-хорошему для лемматизации следует пользоваться проверенными алгоритмами с хорошей базой данных. Мой же алгоритм больше напоминает выборочный стемминг. В идеале для балансировки между скоростью и эффективностью лучше применять нейросети LSTM, GRU. В более простом случае можно использовать алгоритм Стеммера Портера, который реализует интеллектуальный стемминг.