Основы реализации поисковой системы

В. В. Осокин, Р. Ф. Алимов, Р. Р. Хайдаров

Ставится задача построения поисковой системы вебстраниц. Основной задачей системы является выдача отсортированного списка результатов, удовлетворяющий заданному поисковому запросу пользователя. Система разделена на несколько частей: кроулер, индексер, ранжирование и поисковый интерфейс. Статья содержит детальное описание каждой из этих частей. Кроме того, по каждой части предоставлена реализация.

Ключевые слова: интернет, поисковая система, кроулер, индексер, ранжирование, HITS, PageRank, TF-IDF.

Введение

В данной работе будут рассмотрены поисковые системы. Их задачей является осуществление полнотекстового поиска среди вебстраниц (далее просто документы). Необходимость в них возникла сравнительно недавно, с появлением и очень стремительным развитием интернета в 90-х годах XX века.

При построении поисковых систем разработчики сталкиваются с очень большим числом сложностей, возникающих в следствие большого числа факторов. В первую очередь это большое количество документов в интернете. Разные источники приводят разные цифры, но все сходятся в том, что их число оценивается миллиардами. Необходимо учесть, что многие сайты показывают сгенерированные документы, которые меняются со временем. При этом отсутствует какая-либо структурированность содержимого документов. Доступность добавления информации в интернет приводит к тому, что качество документов может быть как высоким, так и низким, так как контент создается не профессиональными редакторами, а кем угодно. В следствие этого возникает задача построения поисковых систем,

способных за малое время находить документы, которые будут потенциально полезны пользователю.

В разделе «Структура базы данных, подключение к ней и стоп-слова» будет рассмотрена структура базы данных и некоторые подготовительные работы для реализации поисковой системы. Большинство поисковых систем включают в себя кроулер, индексер и поисковый интерфейс.

Кроулер представляет из себя сбор документов из интернета с целью дальнейшей работы с ними и сохранения связей, которые состоят из ссылок между ними. Кроулер подробнее будет рассмотрен в разделе «Кроулер».

Индексер — это подсистема для индексация документов с целью получения информации о том, какие слова содержатся в каких документах. Индексер будет рассмотрен подробнее в разделе «Индексер».

Поисковый интерфейс — веб-интерфейс поисковой системы, позволяющий вводить поисковые запросы и получать в ответ подходящие для них страницы (поисковую выдачу). Поисковый интерфейс будет разработан в разделе «Поисковый интерфейс».

В дальнейшем мы рассмотрим каждую подсистему подробнее и реализуем их. Не все документы, содержащие слова из поискового запроса, являются одинаково хорошими источниками информации для пользователей. Сергей Брин, сооснователь компании Google, сказал: «Мы пришли к выводу, что не все веб-страницы созданы равными. Люди — да, но не веб-страницы». Возникает задача сортировки документов, входящих в поисковую выдачу. Эта задача называется ранжированием (раздел «Ранжирование»). В данной работе мы рассмотрим и реализуем два типа ранжирования. Ссылочное ранжирование — это ранжирование документов на основе анализа ссылок между ними. Мы рассмотрим два алгоритма ссылочного ранжирования: HITS (раздел «Алгоритм ранжирования HITS») и PageRank (раздел «Алгоритм ранжирования PageRank»). Далее рассмотрим алгоритмы ранжирования на основе мер TF и IDF. В разделе «Подсчет мер TF и IDF» будет реализован алгоритм подсчета мер TF и IDF. Будут рассмотрены два алгоритма ранжирования документов по поисковым запросам: на основе расчета косинуса угла между векторами (раздел «Алгоритм ранжирования на основе подсчета косинусов углов между векторами») и Okapi BM25 (раздел «Алгоритм ранжирования Okapi BM25»).

Далее в разделе «Применение ранжирования в поисковой выдаче» будет рассмотрено применение алгоритмов ранжирования в генерации поисковой выдачи.

По итогам работы у нас будет реализованная нами поисковая система, которую мы сможем протестировать на реальных данных.

Постановка задачи и полученные результаты

Цель данной работы заключается в разработке поисковой системы и исследовании методов применения алгоритмов ранжирования для улучшения результатов поиска.

Требуется разработать поисковую систему, в которой будут реализованы следующие подсистемы: кроулер, индексер и поисковый интерфейс, и систему ранжирования, применимую к поисковой системе, с использованием алгоритмов ранжирования для оценки качества результатов поиска.

В ходе решения поставленных задач на языке программирования *php* создана поисковая система, в которой были реализованы следующие подсистемы: кроулер, индексер и поисковый интерфейс. Также создана система ранжирования, которая используется для сортировки документов в поисковой выдаче по мере убывания их полезности и информативности пользователю.

На рисунках 1 и 2 приведены результаты поиска слова «plane» без ранжирования [5] и с ранжированием [6] соответственно.

Рис. 1. Поиск слова «plane» без применения ранжирования.

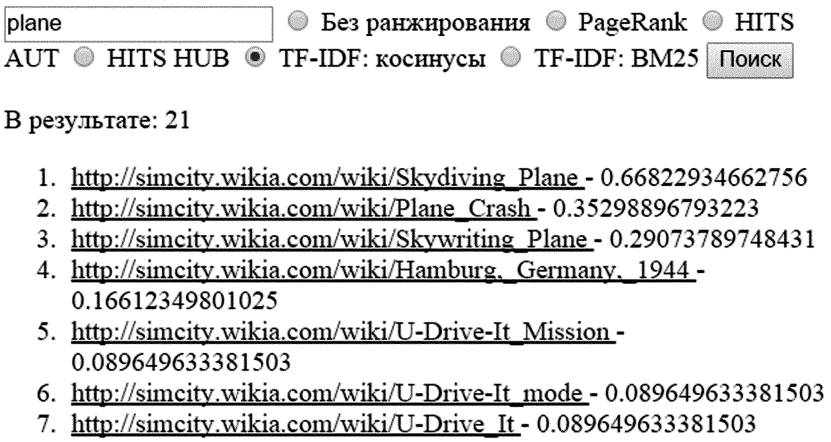

Рис. 2. Поиск слова «plane» с применением ранжирования.

Структура базы данных, подключение к ней и стоп-слова

Для реализации поисковой системы мы используем систему управления базами данных MySQL. Вначале необходимо разработать структуру базы данных. Назовем базу данных search_engine. Перечислим таблицы и из каких полей они состоят. Таблица page служит для хранения документов, которые проиндексированы. Таблица состоит из полей id — уникальный идентификатор документа, url — URL документа, name — название файла на локальном диске, checked — был ли этот документ обработан кроулером, p_rank — PageRank документа, h_auth — оценка авторитетности документа по алгоритму HITS, h_hub — оценка хабности документа по алгоритму HITS. Таблица link служит для хранения ссылок. В ней поле id — уникальный идентификатор ссылки, page_from — уникальный идентификатор документа, в котором находится ссылка, page_to — уникальный идентификатор документа, на который ведет ссылка. Таблица term служит для хранения термов, полученных со всех проиндексированных документов. Она состоит из полей id — уникальный идентификатор терма, term — основа слова, полученная стеммингом (см. раздел «Индексер»), idf — мера IDF данного терма среди

всех проиндексированных документов. Таблица term_page служит для связки между таблицами page и term. В ней содержатся записи слов и документов. В ней id — идентификатор записи, term_id — уникальный идентификатор терма, page_id — уникальный идентификатор документа, num — сколько раз встречается терм в данном документе, tf — мера TF терма в данном документе, tf_idf — мера TF-IDF данного терма в данном документе. Также в системе имеется таблица stop_word стоп-слов (см. описание далее в данном разделе), где id — уникальный идентификатор стоп-слова, term — основа стоп-слова.

Подключение к базе данных содержится в файле db.php [7]. Далее необходимо заполнить таблицу стоп-слов. Список стоп-слов можно легко найти в интернете. В данной работе поисковая система работает с английским языком и, соответственно, список стоп-слов берется для английского языка. Ранее было сказано, что работа будет вестись не с исходным видом слов, а с их основами, которые мы называем термами. Мы используем реализацию алгоритма Porter Stemmer, разработанную Ричардом Хайсом. Стеммер размещен в файле stemmer.php. Создадим файл stopword.php [8] и добавим в него файлы подключения к базе данных и стеммера. Создадим массив стоп-слов, так как их не так много. Очистим таблицу стоп-слов и в цикле сохраним терм от каждого слова в базе данных.

На данном этапе разработана база данных для поисковой системы, создан файл с подключением к базе данных, заполнена таблица стоп-слов английского языка.

Кроулер

Задачей кроулера является сбор документов из интернета с целью дальнейшей работы с ними и сохранения связей между документами, задаваемыми ссылками между ними. Интернет можно представить в виде ориентированного графа, где вершины — это документы, а ребра — это ссылки. Документ, в котором находится ссылка — это начальная вершина ребра, а документ, на который ведет ссылка — это конечная вершина ребра. Данный граф называется веб-графом. Алгоритм работы кроулера похож на обход графа в ширину.

Для работы кроулера необходимо вести список документов, которые ему необходимо посетить и которые он уже посетил. Назовем список непосещенных документов границей (вследствие того, что работа кроулера похожа на обход графа в ширину). В начале работы граница задается нами вручную и список документов состоит только из одного заданного нами документа. Посещение документа состоит из следующих шагов. Сначала мы получаем документ из интернета. Далее выделяем текст из содержимого документа и сохраняем его на локальный диск. После этого собираем ссылки в документе и для каждой из них сохраняем документ, на который ведет ссылка, как граничный, если он отсутствует в списке документов. Сохраняем ссылку, если еще такой нет в базе данных. Далее помечаем текущий документ как пройденный.

В реализации кроулера мы используем библиотеку PHP Simple HTML DOM Parser, которая позволяет легко работать с HTML страницами. Библиотека размещена в файл simple_html_dom.php.

До того, как начинать реализацию кроулера, мы создаем файл functions.php [9], в котором размещаются фукнции, используемые в различных подсистемах поисковой системы. Первой функцией в данном файле является функция getPath, используемая для получения пути к файлу на локальном диске по его названию. В данном случае файлы страниц находятся в директории crawl_data в корневой директории проекта.

Создадим файл crawler.php [10] и подключим к нему файл с подключением к базе данных, файл с общими функциями и библиотеку PHP Simple HTML DOM Parser. В данном примере мы загружаем документы с сайта http://simcity.wikia.com. Это сайт с данными об игре Sim City.

Реализуем функцию getLinks, которая возвращает массив ссылок, находящихся в документе, сохраненном на локальном диске. Используя библиотеку PHP Simple HTML DOM Parser и ее функцию file_get_html, получаем объект для работы с HTML страницей. Если эта функция вернула значение false, возвращаем null. Извлекаем все ссылки из документа. В дальнейшем ссылки нам понадобятся в алгоритмах ранжирования результатов поиска.

Реализуем функцию findPageByUrl, которая находит документ в таблице page базы данных по его URL. Функция делает запрос к таблице page и возвращает его результат.

Опишем функцию createNewPage, которая создает новый документ в базе данных. Вначале производим проверку документа с заданным URL в базе данных. Если документ с заданным URL уже существует в базе данных, возвращаем значение false. Иначе создаем документ в базе данных и возвращаем значение true.

Создадим функцию generateRandomString для генерации случайной строки. Она понадобится при сохранении документа на локальном диске. На входе функция получает необходимую длину строки и возвращает строку необходимой длины, состоящую из случайных символов.

Опишем функцию checkUrl получения HTTP кода при обращении по ссылке. Данная функция понадобится при загрузке документов на локальный диск. Если при обращении по данному URL ответ содержит код 200, то документ существует и его можно загрузить. Используя библиотеку curl, инициализируем HTTP запрос. Производим запрос, получаем заголовки и закрываем соединение. В конце возвращаем HTTP код.

Далее реализуем функцию loadPage, которая используется для сохранения документа на локальный диск по его URL. Если при обращении по данному URL возвращается статус, не равный 200, возвращаем значение null. Получаем объект библиотеки PHP Simple HTML DOM Parser для дальнейшей обработки документа. В документах с данными с сайта http://simcity.wikia.com нас интересует только содержимое элемента div с id равным mw-content-text. Очищаем и удаляем объект PHP Simple HTML DOM Parser чтобы освободить память. Удаляем скрипты JavaScript из документа. Удаляем все HTML теги, кроме тега a, из документа. Удаляем все внешние ссылки. Генерируем случайное название файла и сохраняем документ на локальный диск. Возвращаем название файла документа на локальном диске.

Напишем функцию createNewLink, которая используется для создания новой ссылки в базе данных. Если такая ссылка уже существует в базе данных возвращаем значение false. Сохраняем ссылку в базе данных и возвращаем значение true.

Перейдем к описанию собственно работы кроулера. Перед запуском кроулера необходимо очистить таблицы page и link. Далее задаем начальную границу. В данном случае она состоит из одного документа с URL http://simcity.wikia.com/wiki/SimCity. В цикле прорабатываем все документы, пока еще остаются граничные документы. По-

лучаем список ссылок в документе, сохраняем ссылки. Сохраняем в базе данных название файла на локальном диске и помечаем документ как пройденный. По результатам работы кроулера выводим на экран число загруженных документов.

Если задать в качестве начальной границы документ http://simcity.wikia.com/wiki/SimCity, то в базе данных на момент написания статьи сформировалось 335 документов и 333 из них загрузились на локальный диск. Два документа, как оказалось, загружаются с HTTP статусом 404 (не найдено).

Индексер

Задачей индексера является индексация документов с целью получения информации о том, какие слова содержатся в каких документах. При этом в базе данных в рамках данной статьи мы храним только основы слов. Их мы называем термами. Получение основы слова принято называть стеммингом. Также мы исключаем из рассмотрения предлоги, причастия, междометия, цифры, частицы и тому подобное, так как они встречаются в большей части документов и, соответственно, мало помогают выделению отличий между документами. Список этих слов мы называем стоп-словами и храним в базе данных их основы.

В своей работе индексер использует данные, подготовленные другой подсистемой поисковой системы — кроулером.

Для каждого документа, сохраненного в базе данных необходимо получить текст, сохраненный в файле на локальном диске, разбить текст по пробелам и получить список слов, находящихся в данном документе. Далее для каждого полученного слова необходимо получить основу слова при помощи стемминга. Если полученный терм является одним из стоп-слов, переходим к следующему слову. Если данного терма еще нет в базе данных, сохраняем его. Если нет записи о том, что данный терм находится в данном документе, создаем такую запись и увеличиваем счетчик числа вхождений данного терма в данный документ на единицу.

Добавим следующие функции в файл functions.php [9].

Первая функция — это findTerm. Данная функция находит терм в базе данных. Делается запрос к таблице term по ключевому терму $term и возвращается его результат.

Следующая функция isItStopWord проверяет, является ли терм стоп-словом. Стоп-слова ищем по переменной $term, которая подается на вход функции, в таблице стоп-слов stop_word.

Далее идет функция clearStr, которая оставляет в строке $str только буквы и пробелы.

Теперь перейдем к описанию функций собственно индексера. Создаем файл indexer.php [11] и подключаем к нему файл с подключением к базе данных, наши функции из functions.php [9], библиотеку PHP Simple HTML DOM Parser и стеммер.

Реализуем функцию getArrayWordsFromFile, возвращающую массив из слов, содержащихся в документе. По заданному пути $path проверяем существование файла. Вынимаем текст из документа. Вызываем ранее добавленную в файл functions.php [9] функцию clearStr. Возвращаем массив слов документа

Далее описываем фукнцию addTerms для добавления термов в базу данных. В цикле работаем с каждым словом. Переводим слово в нижний регистр и получаем терм стеммированием слова. Если терм является стоп-словом, переходим к следующему слову. Ищем терм в базе данных. Если нет терма в базе данных, добавляем его и записываем информацию о то, что данный терм содержится в данном документе. Иначе, если терм уже есть в базе данных, пытаемся загрузить информацию о том, что данный терм находится в данном документе. Если терма нет в данном документе, создаем новую запись и указываем, что терм встречается в данном документе один раз на данный момент. Иначе, если уже есть запись о том, что данный терм есть в данном документе, увеличиваем число вхождений терма в данном документе на единицу.

Все необходимые функции перечислены. Теперь перейдем к реализации запуска алгоритма. В первую очередь очищаем таблицы term и term_page. Загружаем данные о всех документах из базы данных. Индексируем каждый документ в цикле. Если значение $page равно null, переходим в начало цикла.

Получаем путь до файла документа и разбиваем его текст на слова. Если массив слов не пустой — добавляем термы и информацию об их содержании в данном документе в базу данных. После индекса-

ции всех документов, существующих на данный момент мы увидим, что число термов на наших страницах равно 4 389 (за исключением стоп-слов) и 30 790 записей о том, что некоторый терм находится в некотором документе.

Поисковый интерфейс

Поисковый интерфейс — это веб-интерфейс поисковой системы, с которой непосредственно взаимодействуют ее пользователи. Задачей поискового интерфейса является получение поисковых запросов от пользователей и генерация по ним поисковой выдачи.

Поисковый интерфейс состоит из двух экранов: экрана с пустой формой ввода поискового запроса, экрана с заполненной формой ранее введенным поисковым запросом и с отображением поисковой выдачи.

Форма ввода поискового запроса отправляет GET запрос в поисковую систему, в ответ на который она выдает сгенерированную поисковую выдачу. При отправке GET запроса с поисковым запросом выполняются следующие шаги. Полученный поисковый запрос разбивается по пустым символам на слова. Далее у каждого слова оставляем только основу с помощью стемминга (получаем список термов). Из полученного списка термов исключаем стоп-слова. Находим все документы, в которых встречаются все слова из запроса. Генерируем поисковую выдачу, которая состоит из ссылок на найденные документы.

Создаем файл search.php [12] и подключаем к нему файл с подключением к базе данных и стеммер.

Реализуем функцию deleteStopWords для удаления стоп-слов из массива термов. В аргумент функции передаем массив $arrayString поискового запроса от пользователя. В цикле проходим по каждому слову запроса. Далее к этому слову применяем стеммер, который возвращает основу слова. Основу слова проверяем является ли оно стоп-словом. Составялем запрос к таблице stop_word, который проверяет является ли данное слово стоп-словом. Если оно не содержится в таблице stop_word, добавляем его в массив $searchWords. Функция возвращает массив $searchWords не содержащий стоп-слова.

Далее реализуем функцию findPageIdsByWords для нахождения уникальных идентификаторов документов по заданному массиву термов $searchWords. В начале пробуем найти документы, содержащие все термы из массива. Если таких документов нет, пробуем найти документы, содержащие хоть один терм из массива.

Формируем данные для поисковой выдачи с помощью функции getResults. На вход функции подается массив $pagesIds, который содержит индентификаторы документов. По заданному массиву в таблице page находим данные документов. Создаем новый массив $resultLinks, который будет содержать найденные результаты. В цикле проходим по каждому элементу массива $pageIds и находим документ по его идентификатору. Добавляем в $resultLink найденный результат запроса. Возвращаем найденные документы, содержащиеся в массиве $resultLinks. Если после отправки пользователем поискового запроса в GET есть значение q, осуществляем поиск. Из строки запроса $_GET['q'] получаем массив $str, содержащий слова запроса. Удаляем стоп-слова из массива $str, используя функцию deleteStopWords, и записываем данные в массив $search_words. По заданному массиву слов $search_words находим индентификаторы документов, которые содержат эти слова, и используем функцию findPageIdsByWords, чтобы найти документы, содержащие данные слова. Если массив $pageIds, содежащий индентификаторы документов не пустой, вызываем функция getResults, которой передаем данный массив, и в дальнейшем выводим полученные данные на экран пользователя.

Далее выводим поисковую форму. В этой форме пользователь вводит поисковый запрос. Формируем поисковую выдачу. Здесь на экран пользователя выводим результаты поиска. Проверяем существование индекса q в массве $_GET. Если данный индекс есть, то был задан поисковый запрос. Проверяем массив $result_links на наличие результата поиска. Если результаты были получены, выводим на экран данные в виде ссылок на найденные документы. Иначе, если поиск не дал результатов, выводим пользователю соответствующее сообщение.

На данный момент наша поисковая система выводит форму для введения поискового запроса, по заполнению которой осуществляется поиск по ранее проиндексированным документам. Результатом поиска является список ссылок на проиндексированные документы.

Ранжирование

На данный момент наша поисковая система может генерировать поисковую выдачу, в которой найденные документы располагаются в совершенно непредсказуемом порядке. Возникает задача сортировки документов поисковой выдачи в порядке убывания их значимости для пользователя, который ищет информацию. Сортировка поисковой выдачи называется ранжированием.

В нашей поисковой системе мы реализовываем два вида алгоритмов ранжирования: алгоритмы ссылочного ранжирования и алгоритмы ранжирования на основе мер TF и IDF.

Алгоритмы ссылочного ранжирования рассчитывают ранги документов на основе ориентированного графа, где вершины — это документы, а ребра — это ссылки. Начальной вершиной ребра является документ, в котором находится ссылка, а конечной вершиной является документ, на который указывается ссылка. Мы рассмотрим два алгоритма ссылочного ранжирования — HITS и PageRank. Алгоритмы ссылочного ранжирования рассчитывают одно или несколько чисел для каждого документа. Данные числа называются рангами и рассчитываются на основе ориентированного графа, где вершины — это документы, а ребра — это ссылки. Начальной вершиной ребра является документ, в котором находится ссылка, а конечной вершиной является документ, на который указывается ссылка. Мы рассмотрим два алгоритма ссылочного ранжирования — HITS и PageRank.

Второй тип алгоритмов ранжирования использует меры TF и IDF для сравнения схожести документа и поискового запроса. TF — это мера важности терма для одного документа. IDF — это мера важности терма для всей коллекции документов. Для данного типа ранжирования рассмотрим два алгоритма. Первый алгоритм параметризует запросы и документы на основе термов, содержащихся в них, создавая векторы документов и запросов. Релевантность документа поисковому запросу определяется косинусом угла между вектором запроса и вектором документа — чем больше косинус угла, тем лучше документ подходит под данный поисковый запрос. Второй алгоритм, который мы рассмотрим и реализуем — это Okapi BM25, представляющий собой формулу, содержащую меры TF и IDF.

Алгоритм ранжирования HITS

Рассмотрим алгоритм ранжирования HITS (англ. Hyperlink Induced Topic Search), который был предложен Джоном Клейнбергом в 1998 году. Идея алгоритма заключается в том, что документ имеет две роли: документ «автор» и документ «посредник».

Документ, на который ссылаются многие другие документы должен быть хорошим «автором». В свою очередь документ, который указывает на многие другие, должен быть хорошим «посредником». При этом чем больше хороших «посредников» ссылаются на документ, тем лучшим «автором» он является и наоборот, если документ содержит в себе ссылки на хорошие «авторы», то он является хорошим «посредником». Основываясь на этом предположении, в алгоритме HITS для каждого документа итеративно рассчитываются две оценки: оценка авторитетности и посредническая оценка.

Оценку авторитетности будем далее называть авторитетностью документа, тогда как посредническую оценку будем называть хабностью документа.

Рассмотрим два типа обновления: правило обновления авторитетности и хаб-обновление. Правило обновления авторитетности: для любого документа p, имеем

$$auth(p) = \sum_{i=1}^{n} hub(i)$$

где n — общее количество документов, связанных с p и i — документ, связанный с p.

Следовательно, оценка авторитетности документа вычисляется как сумма значений оценок посреднических документов, которые указывают на этот документ.

Правило хаб-обновления: для любого документа p, имеем

$$hub(p) = \sum_{i=1}^{n} auth(i)$$

где n — общее количество документов, на которые указывает p и i — документ, на который указывает p. Следовательно, хабность документа вычисляется как сумма значений оценок авторитетности документов, на которых он ссылается.

Рассмотрим как вычисляются авторитетность и хабность документа. В начале ранжирования авторитетность и хабность каждого документа приравняем значению 1. Затем выполняются правила обновления авторитетности и хаб-обновления. Далее происходит нормализация значений путем деления каждой хабности на квадратный корень из суммы квадратов всех хабностей, и деления каждой оценки авторитетности на корень квадратный из суммы квадратов всех оценок авторитетности. И если необходимо повторяем шаги, начиная с правил обновления авторитетности и хаб-обновленя до того, как модуль разницы между предыдущим и текущим значением авторитетности и хабности каждого документа не будет меньше определенного числа, который мы назовем *epsilon*. Чем меньше число *epsilon*, тем больше итераций понадобится для вычисления авторитетности и хабности.

Далее рассмотрим реализацию алгоритма HITS. Создадим файл hits.php [13] и подключим к нему файл с подключением к базе данных.

Реализуем функцию getCountPages для подсчета количества документов. Составляем запрос к базе данных, который подситывает количество документов и возвращает найденный результат.

Далее реализуем функцию calculateAuth для подсчета авторитетности документов. В этой функции выполняется правило обновления авторитетности, которое описано выше. В качестве аргументов функции передаем ссылку на массив, содержащий авторитетность документов, в который будут записываться результаты подсчета правила обновления авторитетности. Кроме того, вторым аргументом функция получает текущую хабность документов. В цикле проходим по каждому документу и составляем запрос к таблице link, который возвращает документы, содержащие ссылки на текущий документ. Если данный запрос вернул не пустой результат, авторитетность документа вычисляется как сумма значений хабности документов, которые ссылаются на текущий документ. Иначе авторитетность документа приравниваем значению 0.

Реализуем фукнцию calculateHub для подсчета хабности документов. В этой функции подсчитываем правило хаб-обновления, которое описано выше. В качестве аргументов функции передаем ссылку на массив, содержащий хабность документов, в который будут записываться результаты подсчета правила хаб-обновления. Кроме того,

вторым аргументом функция получает текущую авторитетность документов. В цикле пробегаем по каждому документу и составляем запрос к таблице link, который возвращает документы, на которые ссылается текущий документ. Если данный запрос вернул не пустой результат, хабность документа вычисляется как сумма значений авторитетности документов, на которые ссылается текущий документ. Иначе хабность документа приравниваем значению 0.

Реализуем функцию conditionOfExit для проверки условия остановки алгоритма. В качестве аргументов функции передаем текущие и предыдущие значения авторитетности и хабности документов. В теле функции опишем цикл, который пробегает по текущему и предыдущему значению авторитетности каждого документа. Если хотя бы для одного документа оказалось, что модуль разницы между текущим и предыдущим значением авторитетности больше *epsilon*, итерации по нахождению значений автотиретности и хабности необходимо продолжать. В данном случае функция возвращает значение false. Далее цикл, который также пробегает по текущему и предыдущему значению хабности документа. Если хотя бы для одного документа оказалось, что модуль разницы между текущим и предыдущим значением хабности больше *epsilon*, итерации по нахождению значений автотиретности и хабности необходимо продолжать. В данном случае функция возвращает значение false. Если не оказалось документов, у которых предыдущее и текущее значение авторитетности или хабности отличается более, чем на *epsilon*, функция возвращает значение true.

Функция normalize используется для нормализации вектора. В аргумент фукнции передается массив вектора, который необходимо нормализовать. Для нормализации вектора необходимо каждую его компоменту поделить на длину вектора. Длина вектора равна квадратному корню суммы квадратов компонентов.

Далее производим начальную подготовку для запуска алгоритма: получаем количество документов, используя функцию getCountPages($dblink), и устанавливаем начальную авторитетность и хабность каждого документа в значение 1. В переменной $iterations будем хранить число итераций алгоритма. Выполняем подсчет авторитетности. Выполняем подсчет хабности. Нормализуем вектора авторитетности и хабности. Если выполняется условие остановки, выходим из цикла. Если условие остановки алгоритма не выполнилось,

то в цикле для каждого предыдущего значения авторитетности и хабности приравнивается текущее значение авторитетности и хабности соответственно. Иначе цикл прерывается. Увеличиваем количество итераций на единицу. Записываем для каждого документа результат расчета его авторитетности и хабности.

После остановки алгоритма можно увидеть, что для выбранного значения *epsilon* и для ранее проиндексированных документов производится шесть итераций обновления авторитетности и хабности.

Алгоритм ранжирования PageRank

PageRank — один из алгоритмов ссылочного ранжирования, разработанный Ларри Пейджем и Сергеем Брином в 1998 году. PageRank итеративно рассчитывает числовое значение, которое характеризует «важность» документов. Рассмотрим формулу посчета PageRank на определенном шаге.

$$PR(A) = \frac{1-d}{N} - d(\sum_{i=1}^{n} \frac{PR(B_i)}{N(B_i)}),$$

где $PR(A)$ — PageRank документа A, d — коэффициент затухания, который означает вероятность того, что пользователь, зашедший в документ, перейдет по одной из ссылок, содержащейся в этом документе, а не закроет браузер. Обычно его принимают равным 0.85, n — количество документов, ссылающихся на документ A, N — количество всех документов, B_i — i-документ, который ссылается на документ A, $N(B_i)$ — количество исходящих ссылок из документа B_i.

Как видно из формулы, каждый документ отдает равную долю своей важности документам, на которых он ссылается. Рассмотрим теперь как итеративно рассчитывается PageRank. Вначале каждому документу приравнивается значение PageRank, равное $1/N$. Далее на каждом шаге для каждого документа производим вычисление приведенной выше формулы. Вычисления заканчиваются, как только для каждого документа модуль разницы между текущим и предыдущим значением PageRank становится меньше заранее определенного числа *epsilon*. Чем меньше число epsilon, тем точнее будут значения PageRank каждого документа.

Далее рассмотрим реализацию расчета значений PageRank для ранее проиндексированных документов. Создадим файл pagerank.php [14] и включим в него файл с подключением к базе данных.

Установим коэффициент затухания и значение epsilon.

Реализуем функцию conditionOfExit для проверки условия остановки алгоритма. В качестве аргументов функции передаем текущие и предыдущие значения PageRank всех документов. В теле функции описываем цикл, который для каждого документа высчитывает модуль разницы между текущим и предыдущим значеним PageRank. Если хотя бы для одного документа оказалось, что модуль разницы между предыдущим и текущим значением PageRank больше *epsilon*, итерации по расчету PageRank необходимо продолжать. И в даном случае функция возвращает значение false. Если не оказалось документов, у которых предыдущее и текущее значение PageRank отличается более, чем на EPSILON, функция возвращает значение true.

Далее реализуем функцию getCountPages для подсчета количества документов. Составляется запрос к базе данных, который подсчитывает количество документов. Полученное значение является результатом работы функции.

Реализуем функцию getCountLinkOnPage для подсчета числа исходящих ссылок для всех документов. Все одинаковые ссылки в документе считаются за одну. В этой функции составляется запрос к таблице link, который для каждого документа подсчитывает количество исходящих ссылок. Результат работы функции возвращаются в виде массива, в котором ключи — это номера документов, а значения — это количество исходящих ссылок.

Затем делаем начальные приготовления для запуска итеративного алгоритма по подсчету значений PageRank. Подсчитываем общее число документов и создаем перменную $iteration для подсчета числа итераций алгоритма. Подсчитываем число уникальных ссылок с документов. Задаем начальное значение PageRank для каждого документа. Создаем переменную $prob, в которой будем хранить значение $(1 - d)/N$ формулы подсчета PageRank. В цикле считаем PageRank, пока не выполнится условие остановки. Выше в запросе к базе данных получаем массив индентификаторов документов, ссылающихся на i-ый документ, и записываем полученные данные в массив $dbLinkToPage. Далее производится вычисление PageRank для каж-

дого документа по формуле, которая была описана ранее. Полученное значение записываем в массив $curRank, который хранит PageRank для каждого документа. Проверяем условие остановки алгоритма. Если условие остановки алгоритма не выплонилось, в цикле для каждого документа записываем текущее значение PageRank как предыдущее. Увеличиваем число итераций на единицу и продолжаем работу цикла. Иначе, если условие остановки алгоритма выполнено, прерываем цикл и заканчиваем подсчет значений PageRank проиндексированных документов. Записываем посчитанные значения PageRank для каждого документа в базу данных и выводим число произведенных итераций на экран пользователя.

После того, как алгоритм выполнился, можно увидеть, что для выбранных значений коэффициента затухания и *epsilon* производится семь итераций.

Подсчет мер TF и IDF

TF (от англ. term frequency — частота слова) — это отношение числа вхождения терма к количеству всех термов документа. То есть чем чаще входит терм в документ — тем он важнее для данного документа.

$$tf = \frac{n_i}{\sum_k n_k}$$

где n_i — число вхождений терма в документ, а n_k — общее число термов в данном документе.

IDF (от англ. inverse document frequency — обратная документная частота) — инверсия частоты, с которой терм встречается во всех документах. То есть, если данный терм встречается практически в каждом документе, то он менее важен для поиска.

$$idf = log\frac{N}{df_i}$$

где N — количество всех документов, df_i — количество документов, в которых встречается терм f_i.

Таким образом, мера TF-IDF является произведением этих двух частот. То есть самый важный терм (с большим весом TF-IDF) — такой, который чаще всего встречается в одном документе, и реже во всех остальных.

$$tfidf = tf \cdot idf$$

Рассмотрим реализацию расчета мер TF и IDF в нашей поисковой системе. Создадим файл tf-idf.php [15] файл с подключением к базе данных.

Реализуем функцию getCountPages для подсчета количества документов. Составляем запрос к базе данных, который подситывает количество документов и возвращает найденный результат.

Затем итед реализация функции getCountTermPages, используемой для подсчета числа записей в таблице term_page. Составляем запрос к таблице и возвращаем количество записей в таблице.

Функция getCountWordsOnPage используется подсчета числа термов в документе. В качестве аргумента функции передается индентификатор документа, по которому в таблице term_page подсчитывается количество термов в документе и возвращается найденный результат.

Далее реализована функция getCountPagesOnWord для подсчета числа документов, в которые входит терм. В качестве аргумента функции передается индентификатор терма. По этому индентификатору терма в таблице term_page находятся все документы, которые содержат этот терм. Возвращается количество найденных документов.

Затем идет расчет мер TF и IDF в цикле. Получаем начальные данные количество документов $countPages, используя функцию getCountPages($dbLink) и число записей $countTermPages, используя функцию getCountTermPages($dbLink). В цикле для каждой записи из таблицы term_page будем подсчитывать меру TF-IDF. Рассчитываем меру TF по описанной ранее формуле. Получаем число вхождений терма в документ. Получаем общее число термов в документе. Вычисляем значение меры TF. Расчитываем меру IDF по описанной ранее формуле. Получаем количество документов, в которых встречается терм. Вычисляем значение меры IDF.

$$idf = log\frac{countPage}{countPagesOnWord}$$

Мера TF-IDF терма в документе является произведением мер TF и IDF.

$$tf\_idf = tf \cdot idf$$

Записываем результат в базу данных. Полученные значения мер TF и TF-IDF записываем в таблицу term_page. Полученное значение меры IDF записываем в таблицу term.

Алгоритм ранжирования на основе подсчета косинусов углов между векторами

Мера TF-IDF показывает насколько важным является терм, в содержащем его документе и во всей коллекции документов. С данной мерой удобно работать, потому что она представляет собой вещественное число. Для того, чтобы использовать меру TF-IDF для ранжирования, необходимо произвести параметризацию документов и поисковых запросов на ее основе. Удобным для работы является метод параметризации документов и запросов, представляющий собой составление векторов из мер TF-IDF термов, содержащихся в документе.

Рассмотрим метод составления векторов документов и векторов запросов. Размерность каждого вектора будет равна количеству термов в коллекции документов. Для начала рассмотрим составление вектора документа:

$$\vec{d_j} = (w_{1j}, w_{2j}, \ldots, w_{nj})$$

где $\vec{d_j}$ — векторное представление j-го документа, w_{ij} — вес i-го терма в j-м документе (если i-й терм не содержится в j-м документе, значение w_{ij} берем равным значению 0), n — общее количество различных термов во всех документах коллекции.

Далее рассматривается составление вектора поискового запроса:

$$\vec{q} = (r_1, r_2, \ldots, r_n)$$

где $\vec{q}$ — векторное представление запроса, r_i — булевый вес i-го терма — равен 1, если он встречается в запросе и 0 в противном случае, n — общее количество различных термов во всех документах коллекции.

Получив вектора запроса и документа можно вычислить меру сходства между двумя векторами с помощью измерения косинуса угла между ними, который будет показывать на сколько запрос и доку-

мент похожи. Косинус угла между вектором документа $\vec{d_j}$ и вектором запроса $\vec{q}$ рассчитывается по формуле:

$$cos(\vec{d_j}, \vec{q}) = \frac{\vec{d_j} \cdot \vec{q}}{\| \vec{d_j} \| \cdot \| \vec{q} \|}$$

где $\| \vec{d_j} \|$ и $\| \vec{q} \|$ — нормы векторов $\vec{d_j}$ и $\vec{q}$ соответственно, $\vec{d_j} \cdot \vec{q}$ — скалярное произведение векторов $\vec{d_j}$ и $\vec{q}$.

Отсюда можно сказать, что чем меньше угол между вектором документа и запроса, тем больше его косинус, а следовательно, поисковый запрос и документ более похожи.

Алгоритм будет реализован в разделе «Применение ранжирования в поисковой выдаче».

Алгоритм ранжирования Okapi BM25

Okapi BM25 — это алгоритм ранжирования на основе расчета одноименной формулы, в которой используются меры TF и IDF.

Рассмотрим формулу алгоритма ранжирования Okapi BM25:

$$score(D, Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgl})}$$

где Q — поисковый запрос.
D — документ из нашей коллекции.
q_i — термы (слова), которые содержатся в запросе Q.
$f(q_i, D)$ — частота терма q_i, то есть мера TF терма q_i в документе D.
$IDF(q_i)$ — обратная документная частота терма q_i, то есть мера IDF терма q_i .
$| D |$ — количество всех термов в документе D (или длина документа).
$avgl$ — средняя длина документа в коллекции.
k_1 и b — свободные коэффициенты, где $k_1 = 1.5$, $b = 0.75$.

Okapi BM25 — это одна формула из целого семейства формул. Варьируя зрачения $k1$ и b можно получить различные формулы из данного семейства и их результаты ранжирования будут немного отличаться. Подробное рассмотрение данного семейства формул выходит за рамки данной работы.

Алгоритм ранжирования Okapi BM25 будет реализован в разделе «Применение ранжирования в поисковой выдаче».

Применение ранжирования в поисковой выдаче

В разделе «Поисковый интрефейс» было рассмотрено подробное описание и задача поискового интерфейса нашей поисковой системы. В этом разделе будут применены и реализованы все типы алгоритмов ранжирования и выдача их результатов пользователю в поисковой выдаче. К нашему поисковому интферейсу добавится возможность выбора из числа алгоритмов ранжирования, которые были описаны ранее: ссылочные алгоритмы ранжирования HITS и PageRank; алгоритмы ранжирования на основе мер TF-IDF — алгоритм на основе расчета косинуса угла между векторами и алгоритм Okapi BM25.

Создадим файл searchrank.php [16], к которому подключим файл с подключением к базе данных и стеммер.

Определяем параметры k1 и b для алгоритма Okapi BM25.

Первою опишем функцию length для подсчета длины вектора. На входе функция получает вектор-массив, длину которого нужно вычислить. Длина вектора равна квадратному корню суммы квадратов компонентов. Пусть дан вектора $\vec{a}$, тогда длина вектора: $\mid \vec{a} \mid = \sqrt{a_1^2 + a_2^2 + \ldots + a_n^2}$

Далее опишем функцию product для подсчета скалярного произведения векторов. На входе функция получает два вектор-массива и возвращает скалярное произведение векторов. Для подсчета скалярного произведения векторов воспользуемся следующей формулой: $\vec{a} \cdot \vec{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n$, где $\vec{a}$ и $\vec{b}$ — векторы.

Функция buildVectorWords используется для построение вектора поискового запроса. Как было сказано ранее, $\vec{q} = (r_1, r_2, \ldots, r_n)$ где $\vec{q}$ — векторное представление запроса, r_i — булевый вес i-го терма — равен 1, если он встречается в запросе и 0 в противном случае, n — общее количество различных термов во всех документах коллекции. Получаем количество всех термов в коллекции и сохраняем его в переменную $countWords. Инициализируем начальный вектор-массив запроса и в цикле приравниваем значение 0. В цикле для каждого поискового терма в массиве $searchWords строим вектор запроса

$vectorWords. Значение компоненты вектора будет равно 1, если терм встречается в нашей коллекции термов. После всех действий функция возвращает вектор-массив запроса.

Подсчет числа термов в базе данных осуществляется с помощью функции getCountTerms. Составляем запрос к таблице term и возвращаем результат в виде количества найденных термов.

Построение векторов для страниц реализовано в функции buildVectorPages. Как было сказано ранее, $\vec{d_j} = (w_{1j}, w_{2j}, \ldots, w_{nj})$ где $\vec{d_j}$ — векторное представление j-го документа, w_{ij} — вес i-го терма в j-м документе (если i-й терм не содержится в j-м документе, значение w_{ij} берем равным значению 0), n — общее количество различных термов во всех документах коллекции. В качества аргумента функции передается массив $pageIds, содержащий индентификаторы документов, которые содержат термы из поискового запроса. В результате работы функции возвращается массив векторов документов. Инициализируем массив $vectorPages, в который далее запишем векторы документов. Получаем число термов в коллекции и записываем в переменную $termsCount. В цикле пробегаем по каждому документу массива $pageIds. Заполняем вектор-массив $vectorPages значениями 0 и устанавливаем размерность вектора равным$termsCount. По индентификаторам страниц из массива $pageIds составляем запрос к таблице term_page, который возвращает нам необходимые данные, такие как: идентификаторы термов, идентификаторы страниц и их меры TF-IDF. В цикле для каждой записи формируем $vectorPages, который представляет собой двумерный массив индентификаторов документов и индентификаторов термов. Значением двумерного массива $vectorPages будут веса термов в документах. Возвращаем заполненный двумерный массив $vectorPages.

Функция tf_idf содержит подсчет ранга документа по поисковому запросу на основе алгоритма ранжирования по косинусам углов между векторами документов и запросов. В качестве аргумента функции передается массив $searchWords термов поискового запроса и массив $pageIds индентификаторов документов, которые содержат термы из поискового запроса.

Создаем вектор запроса и векторы документов с помощью вызова ранее определенных функций. Далее инциализируем массив $tf_idf, который будет хранить значение косинуса угла между векторами за-

проса и документов. В цикле считаем косинус угла для вектора запроса и векторов документов. Создаем массив для перевода результатов расчета в более удобный вид, с дальнейшей его сортировкой по убыванию ранга.

Расчет рангов документов по поисковому запросу на основе алгоритма ранжирования Okapi BM25 реализован в функции bm25. В качестве аргумента функции передаем массив $searchWords термов поискового запроса и массив $pageIds индентификаторов документов, которые содержат термы из поискового запроса. Подсчитываем количество всех документов в таблице page. Составляем запрос в базе данных и полученные результаты записываем в переменную $countPages. Подсчитываем количество всех записей в таблице term_page. Составляем запрос в базе данных и полученные результаты записываем в переменную $countTermPage. Вычисляем среднюю длину документа в коллекции. Далее из таблицы term производим выборку термов поискового запроса. По результатам данного запроса идентификаторы термов записываем в массив $termsIds, а в массив $termsIDFs записываем метрики IDF соответствующих термов. Составляем запрос к таблице term_page. После получения данных применяется формула алгоритма Okapi BM25, рузультат которой записываем в массив $pageRank, состоящий из ключа — индентификатора документа, и значения формулы Okapi BM25. Создаем массив для перевода результатов расчета в более удобный вид, с дальнейшей его сортировкой по убыванию ранга.

Фукция deleteStopWords используется для удаления стоп-слов из массива. В качестве аргумента функции передаем массив поискового запроса пользователя, из которого удаляюся стоп-слова. Подробное описание функции было в разделе «Поисковый интерфейс».

findPageIdsByWords — это функция нахождения уникальных идентификаторов документов по заданному массиву термов. В начале пробуем найти документы, содержащие все термы из массива. Если таких документов нет, то пробуем найти документы, содержащие хоть один терм из массива.

Далее описана функция getResultsByPageRank, генерирующая поисковуя выдачу на основе алгоритма ранжирования PageRank. На вход функции подается массив индентификаторов документов $pageIds. Инициализируем массив $resultLinks и составляем запрос к таблице page, который по индентификаторам документов из мас-

сива $pageIds находит документы и производит сортировку результатов по весу PageRank. Найденные результаты записываем в массив $resultLinks и возвращаем в качестве результата вызова функции.

Поисковая выдача на основе алгоритма ранжирования HITS AUTH реализована в функции getResultsByHITSaut. Инициализируем массив $resultLinks и составляем запрос к таблице page, который по индентификаторам документов из массива $pageIds находит документы и производит сортировку результатов по авторитетности документов. Найденные результаты записываем в массив $resultLinks и возвращаем в качестве результата вызова функции.

А поисковая выдача на основе алгоритма ранжирования HITS HUB выполняется с помощью функции getResultsByHITShub. Инициализируем массив $resultLinks и составляем запрос к таблице page, который по индентификаторам документов из массива $pageIds находит документы и производит сортировку результатов по хабности страниц. Найденные результаты записываем в массив $resultLinks и возвращаем в качестве результата вызова функции.

Далее идет поисковая выдача на основе алгоритма ранжирования по косинусам углов между векторами запроса и документов, реализованная в функции getResultsByTFIDF. На вход функции подается массив индентификаторов документов $pageIds и массив поискового запроса $searchWords. Инициализируем массив $resultLinks и получаем данные с помощью функции tf_idf($pageIds, $searchWords, $dbLink), которая рассчитывает ранг документа по поисковому запросу на основе алгоритма ранжирования по косинусам углов между векторами запроса и документов. Результат перезаписываем в $pageIds, который содержит индентификаторы и ранги документов. Далее, для каждой записи из $pageIds в цикле составляем запрос к таблице page, который возвращает данные о документах. Найденные результаты записываем в массив $resultLinks, который возвращаем в качестве результата работы функции.

Поисковая выдача на основе алгоритма ранжирования Okapi BM25 реализована в функции getResultsByBM25. На вход функции подается массив индентификаторов страниц $pageIds и массив поскового запроса $searchWords. Инициализируем массив $resultLinks и получаем данные с помощью функции bm25($pageIds, $searchWords, $dbLink), которая подсчитвает ранги документов по поисковому запросу на основе алгоритма ранжирования Okapi BM25. Результат

перезаписываем в $pageIds, который содержит индентификаторы документов и ранг документа. Далее, для каждой записи из $pageIds в цикле составляется запрос к таблице page, который возвращает данные о документах. Найденные результаты записываем в массив $resultLinks, который возвращаем в качестве результата работы функции.

Поисковая выдача без алгоритма ранжирования производится функцией getResultsWithoutAlgorithm. На вход функции подается массив $pagesIds, который содержит индентификаторы документов. В цикле по каждому элементу массива $pageIds находим документы по идентификаторам и добавляем в новый массив $resultLinks, который возвращаем в качестве результата работы функции.

Далее идет алгоритм страницы поисковика с ранжированием результатов.

Если после отправки пользователем поискового запроса в GET есть значение q, то осуществляем поиск. Из строки запроса $_GET['q'] получаем массив $str слов запроса. Производим удаление стоп-слов из массива $str, используя функцию deleteStopWords, и записываем данные в массив $search_words. По заданному массиву слов $search_words находим индентификаторы документов, которые содержат эти слова, и воспользуемся функцией findPageIdsByWords, чтобы найти документы, содержащие данные слова. Если существуют документы, содержащие данные слова и передана информация о том, какой необходимо использовать алгоритм ранжирования, продолжаем работу по поиску документов. На данном этапе задача состоит в том, чтобы ранжировать найденные документы по тому или иному алгоритму.

Ранжируем документы на основе алгоритма PageRank. Ранжируем документы на основе алгоритма HITS по авторитетности документов. Ранжируем документы на основе алгоритма HITS по хабности документов. Ранжируем документы на основе алгоритма на основе подсчета косинусов углов между векторами поискового запроса и документов. Для сравнения результатов, была оставлена возможность поиска документов без применения алгоритмов ранжирования. Вывод страницы. Выводим форму поискового запроса. Выводим радиокнопки для выбора алгоритмов ранжирования на экране пользователя. Формируем поисковую выдачу. Здесь на экран пользователя выводим результаты поиска. Проверяем существование индекса q в

массве $_GET. Если данный индекс есть, то был задан поисковый запрос. Проверяем массив $result_links на наличие результата поиска. Если результаты были получены, выводим на экран пользователя данные ввиде ссылок на найденные документы. Иначе, если поиск не дал результатов, выводим пользователю соответствующее сообщение.

На данный момент наша поисковая система может ранжировать результаты поиска с помощью различных алгоритмов. При сравнении поисковой выдачи с ранжированием и без него, очевидно, что качество поиска стало намного лучше.

Заключение

В данной работе были рассмотрены и реализованы следующие подсистемы поисковых систем. Кроулер загружает содержимое документов и сохраняет связи между документами. Индексер разбивает текст документа на термы и создает индекс для дальнейшего поиска по нему. Поисковый интерфейс позволяет осуществить поиск информации среди загруженных и проиндексированных страниц.

Немаловажную часть работы составляют рассмотренные два вида алгоритмов ранжирования. Основной идеей ссылочных алгоритмов ранжирования является анализ связи между страницами, создаваемый ссылками. В статье реализованы алгоритмы ссылочного ранжирования HITS и PageRank. Реализованы алгоритмы ранжирования на основе мер TF и IDF страниц по поисковым запросам: алгоритм ранжирования на основе подсчета косинусов углов между векторами и Okapi BM25.

Реализованную поисковую систему мы смогли протестировать на реальных данных и увидели разницу в ранжировании в зависимости от примененных алгоритмов ранжирования.

Список литературы

[1] Manning C. D., Raghavan P., Schütze H. An introduction to Information Retrieval. — Cambridge University Press, 2008.

[2] Kleinberg J. M. Authoritative sources in a hyperlinked environment // Journal of the ACM. — 1999.

[3] Brin S., Page L. The Anatomy of a Large-Scale Hypertextual Web Search Engine // Computer Networks and ISDN Systems. — 1998.

[4] PHP Group [http://php.net].

[5] Поиск слова «plane» без применения ранжирования [http://search.dvinemnauku.ru/searchrank.php?q=plane &alg=NoRank].

[6] Поиск слова «plane» с применением ранжирования [http://search.dvinemnauku.ru/searchrank.php?q=plane &alg=TFIDF].

[7] Листинг файла db.php [http://search.dvinemnauku.ru/listing.html#connect-db].

[8] Листинг файла stopword.php [http://search.dvinemnauku.ru/listing.html#stopword].

[9] Листинг файла functions.php [http://search.dvinemnauku.ru/listing.html#functions].

[10] Листинг файла crawler.php [http://search.dvinemnauku.ru/listing.html#crawler].

[11] Листинг файла indexer.php [http://search.dvinemnauku.ru/listing.html#indexer].

[12] Листинг файла search.php [http://search.dvinemnauku.ru/listing.html#search].

[13] Листинг файла hits.php [http://search.dvinemnauku.ru/listing.html#algorithm-hits].

[14] Листинг файла pagerank.php [http://search.dvinemnauku.ru/listing.html#algorithm-pagerank].

[15] Листинг файла tf-idf.php [http://search.dvinemnauku.ru/listing.html#algorithm-tf-idf].

[16] Листинг файла searchrank.php [http://search.dvinemnauku.ru/listing.html#searchrank].