Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Лебедева А. В.

# Статическая приоритезация поискового робота.

Дипломная работа

Научный руководитель: Романенко А. А.

Санкт-Петербург
2013

# Содержание

# Введение

Объем информации в интернете в последнее время очень быстро растет: на данный момент число сайтов исчисляется исчисляется десятками миллиардов, а число веб-страниц — десятками биллионов. Кроме того, информация в интернете быстро изменяется, веб-страницы обновляются все чаще. В связи с данными особенностями развития интернета поиск необходимой информации является очень сложной задачей для пользователя, для упрощения которой были созданы поисковые системы.

Поисковый робот является частью поисковой системы, необходимой для ее корректного функционирования. Его задачей является перебор страниц интернета с целью их загрузки и сохранения в базу данных системы для последующих выдач в пользовательских поисковых запросах. Чтобы удовлетворять требованиям пользователей, поисковая система должна хранить важные веб-страницы, то есть востребованные пользователями. Для чего необходима стратегия приоритезации поискового робота, которая представляет собой комбинацию двух стратегий: отбора веб-страниц и повторного скачивания.

Для определения важности страниц необходим алгоритм ранжирования. Существует два основных подхода к ранжированию веб-страниц. Первый подход основан на анализе веб-графа. Среди алгоритмов, использующих данный подход, алгоритм PageRank, который успешно применяет поисковая система Google, а также HITS. Вычисление подобных рангов является достаточно ресурсоемкой операцией. Второй подход основан на анализе текстовых особенностей веб-страницы, особенностей ее URL-адреса и анализе ее содержимого. К алгоритмам, использующим данный подход относят Okapi BM25, Tf–idf, обучение ранжированию.

Задача стратегии отбора состоит в том, чтобы поисковый робот выбирал важные веб-страницы для скачивания в первую очередь, для чего необходима эвристическая оценка важности. Исходя из метода получения такой оценки, стратегии разделяют на три категории: использующие информацию о текущем обходе (обход в ширину, OPIC, FICA и т.д.), использующие информацию из предыдущих обходов и располагающие полной информацией.

Кроме того, поисковая система должна располагать актуальной информацией о состоянии веб-страниц, для чего поисковый робот должен повторно скачивать страницы, уже имеющиеся в базе поисковой системы. Здесь возникает вопрос: как часто повторно скачивать веб-страницы, и в каком порядке. Для чего необходима стратегия повторного скачивания.

В данной работе описывается разработка стратегии приоритезации поискового робота, базирующейся на методике обучения ранжированию с использованием статических особенностей URL-адресов веб-страниц.

# Глава 1. Обзор предметной области

В данной главе приведено описание предметной области. Описаны существующие подходы к алгоритмам ранжирования веб-страниц и стратегии приоритезации поискового робота.

## 1.1. Основные понятия

### 1.1.1. Интернет как граф

Одной из формальных моделей, с помощью которой можно представить интернет, является его представление в виде направленного графа. Вершинами в этом графе являются веб-страницы, а ребрами — ссылки. Из вершины $u$ есть ребро в вершину $v$, если на веб-странице, соответвующей узлу $u$ есть ссылка на веб-страницу, соответствующей вершине $v$. Заметим, что данный граф является динамическим, поскольку веб-страницы часто изменяются, какие-то из них удаляются, а также создаются новые.

### 1.1.2. Поисковая система

Поисковая система — это программный комплекс, предназначенный для помощи пользователю осуществления поиска информации в интернете. Основные компоненты поисковой системы представлены на рисунке 1.1.

Когда пользователь осуществляет информационный запрос, поисковая система формирует упорядоченное подмножество веб-страниц, хранящихся в ее индексе. Таким образом, качество работы поисковой системы во многом зависит от содержимого индекса, формирование которого осуществляется индексатором из множества веб-страниц коллекции, полученной в процессе обхода интернета, потому очень важно, чтобы в коллекции содержались важные веб-страницы, то есть веб-страницы, востребованные пользователем. Алгоритмы определения важности веб-страниц описаны в

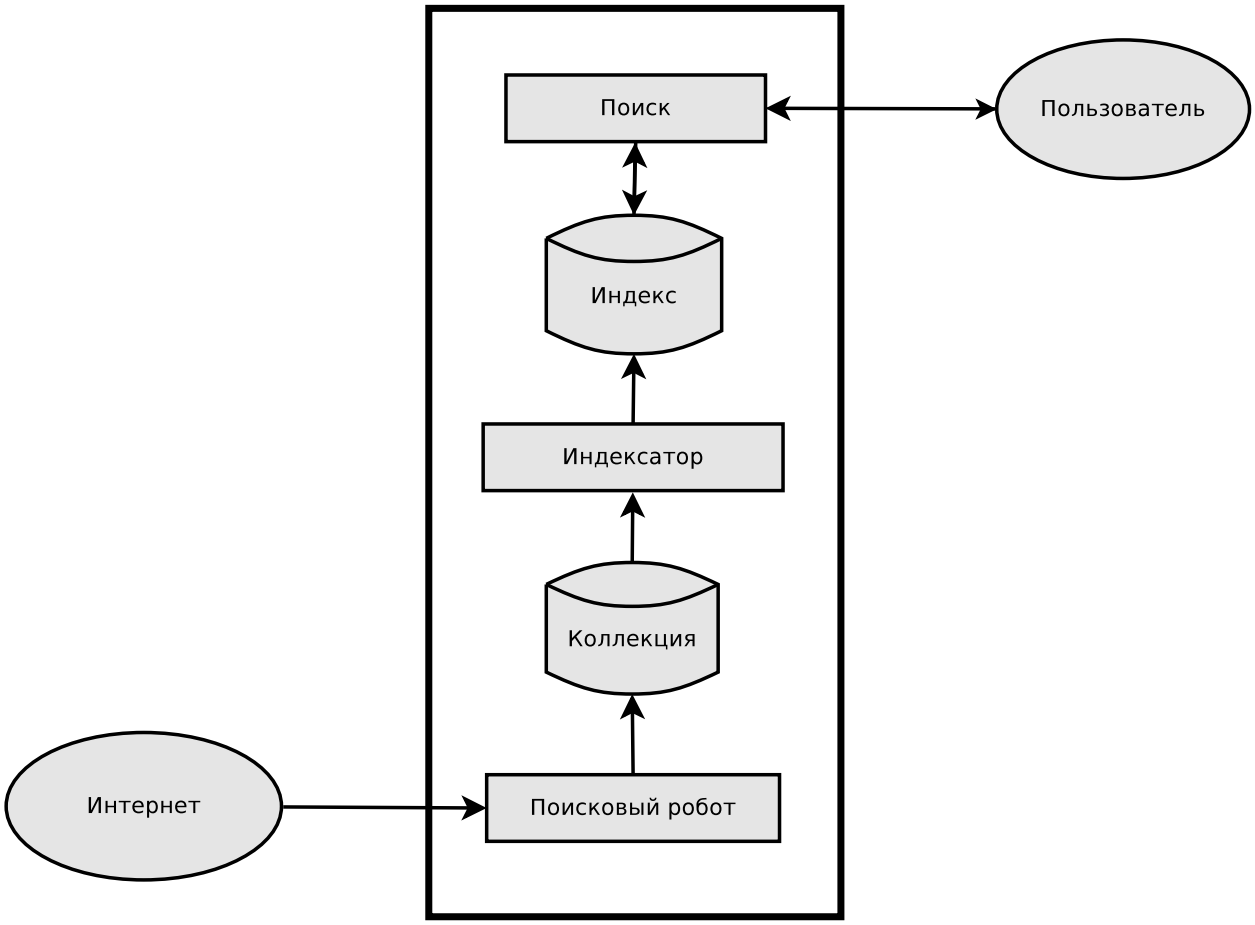


Рис. 1.1. Основные компоненты поисковой системы

разделе 1.2. Коллекцию, в свою очередь, формирует поисковый робот. Подробное описание работы которого приведено в разделе 1.3. Таким образом, поисковый робот в процессе обхода должен производить оценку страниц, и скачивать лишь важные.

## 1.2. Оценка важности веб-страниц

Оценка важности веб-страницы называется рангом. Процесс получения оценки важности страницы называется ранжированием. Рассмотрим существующие методы, осуществляющие ранжирование.

### 1.2.1. Алгоритмы ранжирования, использующие свойства веб-графа

Данная категория алгоритмов использует представление интернета в виде графа, описанное в разделе 1.1.1.

#### PageRank

Метод разработан Сергеем Брином и Ларри Пейджом, основателями поисковой системы Google. Первая статья, описывающая этот алгоритм ранжирования была опубликована в 1998 году [1]. Основная идея метода заключается в том, что страницы, чаще посещаемые в ходе случайного обхода веб-графа, имеют большую важность. Пусть имеется $N$ веб-страниц. Обход начинается с веб-страницы $p$, и в каждый момент времени с вероятностью $\alpha$ осуществляется переход по случайной ссылке, имеющейся на текущей странице. Также с вероятностью $1-\alpha$ мы переходим на случайную страницу. Итоговая формула для вычисления PageRank имеет следующий вид:

$$PageRank(p) = \frac{1-\alpha}{N} + \alpha * \sum_{p' \in In(p)} \frac{PageRank(p')}{|Out(p')|}$$

где $In(p)$ — множество страниц, содержащих ссылку на $p$, $Out(p')$ — множество страниц, на которые ссылается $p'$, а $\alpha$ — фиксированный параметр $(0 < \alpha < 1)$.

Ранжирование с использованием этого алгоритма является достаточно ресурсоемкой операцией, и требует знания о всей структуре веб-графа, что делает применение данного подхода на практике достаточно затруднительным.

#### HITS

Hyperlink-Induced Topic Search (HITS) — метод, разработанный в 1999 году Джоном Клейнбергом и описанный в работе. В отличие от

*PageRank*, является запросо-зависимым алгоритмом ранжирования, то есть полученные в результате анализа ссылок оценки веб-страниц зависят от конкретного поискового запроса. Базовая идея алгоритма заключается в том, что определенные веб-страницы являются *посредниками* (*hubs*), то есть используются как каталоги, указывающие на другие страницы, являющиеся авторитетными источниками (*authorities*) или *авторами* [2], которые содержат необходимую пользователю информацию. Тогда страница, указывающая на большое количество хороших *авторов*, является хорошим *посредником*, и наоборот, страница, на которую ссылается много хороших *посредников*, является хорошим *автором*. Основываясь на этом, для каждой страницы вычисляется две оценки: оценка авторитетности и посредническая оценка.

Для начала работы алгоритма необходимо получить множество наиболее релевантных страниц для текущего поискового запроса, называемое *корневым множеством* (*root set*). На его основе формируется *базовое множество* (*base set*), получаемое в результате добавления к *корневому множеству* всех веб-страниц, связанных с ним ссылками, как исходящими, так и входящими. Все вычисления проводятся на подграфе, сформированном *базовым множеством*. Основной алгоритм выполняет ряд итераций, на каждой из которых происходит обновление оценок для каждой вершины подграфа. Обновление посреднической оценки выполняется посредством суммирования оценок авторитетности каждой из вершин, на которые указывает текущая. В свою очередь, обновление авторитетной оценки осуществляется путем суммировния посреднических оценок вершин, ссылающихся на текущую. Окончательные оценки формируются после выполнения большого числа итераций алгоритма.

### 1.2.2. Обучение ранжированию

Алгоритмы данной категории используют методы машинного обучения для ранжирования документов. По обучающей выборке из элементов с заданным на них частичным порядком строится ранжирующая модель, которая затем используется для ранжирования новых данных. Как правило, необходимо отсортировать документы, отвечающие некоторому поисковому запросу. Элемент представляет собой пару: документ-запрос, для которой строится числовой вектор ранжирующих признаков. Признаки можно разделить на три группы:

- *Запросо-независимые.* Зависят только от самого документа, а не от запроса. Например, *PageRank* или длина документа.
- *Признаки запроса.* Зависят только от запроса. Например, число слов в запросе.
- *Запросо-зависимые.* Зависят как от документа, так и от запроса. Например, ранг *HITS*, мера *TF-IDF* соответствия запросу и т.д.

В статье [3] рассмотрены существующие подходы к обучению ранжировнию и разделены на следующие категории в соответствии с форматом входных данных и функцией потерь:

- *Поточечные подходы.* Каждому элементу сопоставляется численная оценка. Задача обучения — по паре запрос-документ предсказать ее оценку. К ним относятся: *OPRF, SLR, Pranking.*
- *Попарные подходы.* На вход подаются два документа и необходимо определить, какой из них лучше. Среди них алгоритмы: *RankSVM, RankBoost, RankNet, FRank* и т.д.
- *Списочные подходы.* На вход подается сразу весь набор документов, соответвующих данному запросу, а на выходе должна получиться их перестановка. Среди них: *ListNet, AdaRank, SoftRank, BayesRank* и т.д.

## 1.3. Поисковый робот

### 1.3.1. Общее описание

Одним из важнейших условий функционирования поисковой системы является осуществление обхода интернета (web crawling). Его производит поисковый робот (web crawler or spider). Принцип его работы заключается в следующем. Робот хранит очередь URL-адресов веб-страниц, которые ему необходимо обойти. На момент начала обхода в ней находится некоторое исходное множество адресов. На каждой итерации робот извлекает из очереди следующий URL-адрес и скачивает веб-страницу, которая ему соответствует. Затем полученная страница обрабатывается, из нее извлекается содержимое и ссылки. Содержимое передается индексатору для последующей обработки и добавления страницы в базу поисковой системы. А URL-адреса страниц, содержащихся в ссылках, извлеченных с текущей страницы, добавляются в очередь на скачивание.

В книге [4] сформулированы основные свойства поискового робота. Они разделены на две категории: обязательные и желательные.

### 1.3.2. Обязательные свойства

Обязательными свойствами поискового робота являются:

1. Устойчивость.
2. Вежливость.

Остановимся на каждом из них подробнее.

#### Устойчивость

Число страниц в интернете потециально может быть бесконечно, поскольку некоторые веб-сервера генерируют динамические веб-страницы по запросу на их создание, которым, в частности, является переход по ссылке.

Таким образом, поисковый робот может зациклиться на определенном хосте, бесконечно скачивая вновь сгенерированные динамические страницы. Такой механизм называется ловушкой для роботов (spider trap). Поисковый робот должен быть устойчив к подобного рода явлениям.

#### Вежливость

Для сайтов существуют явные и неявные правила поведения поискового робота при работе с ними. Эти правила регулируют частоту обращения к хосту, а также возможность доступа к его содержимому. Многие сайты имеют в корне специальный файл *robots.txt*, содержащий желательные правила поведения. Поисковый робот должен соблюдать их.

### 1.3.3. Желательные свойства

Желательными свойствами поискового робота являются:

1. Распределенность.
2. Масштабируемость.
3. Производительность и эффективность.
4. Качество.
5. Свежесть.
6. Расширяемость.

Остановимся на каждом из них подробнее.

#### Распределенность

Подразумевает возможность функционирования поискового робота одновременно на многих машинах.

### Масштабируемость

Подразумевает возможность увеличения эффективности поискового робота при увеличении ресурсов. Таких как увеличение числа машин или расширения полосы пропускания.

### Производительность и эффективность

Подразумевает эффективное использование поисковым роботом доступных ему ресурсов.

### Качество

Желательно, чтобы поисковый робот при скачивании веб-страниц отдавал предпочтение более качественным. Для выполнения этого свойства необходима стратегия приоритезации страниц. Существующие стратегии подробнее рассмотрены в разделе 1.4.

### Свежесть

Для хранения свежей информации о веб-страницах в базе поисковой системы необходимо, чтобы поисковый робот обходил ранее скачанные страницы с частотой, соответствующей частоте их обновления. Способы выполнения этого свойства также рассмотрены в разделе 1.4.

### Расширяемость

Поисковый робот должен быть разработан так, чтобы имелась возможность расширения для решения новых задач. Таких как добавление новых протоколов передачи данных или новых форматов данных.

## 1.4. Алгоритм приоритезации поискового робота

Под приоритезацией подразумевается введение полного или частичного порядка на множестве веб-страниц, которое поисковый робот должен обработать. Это необходимо для того, чтобы поисковая система хранила в базе страницы, содержащие информацию, которая необходима пользователю, осуществляющему поисковый запрос. Скачивание качественных веб-страниц осуществляется за счет применения стратегии отбора, а наличие свежей информации в базе данных поисковой системы осуществляется за счет применения той или иной стратегии повторного обхода. Объединение этих двух стратегий составляет стратегию приоритезации поискового робота.

### 1.4.1. Стратегии отбора

Стратегия отбора определяет, в каком порядке поисковый робот должен обрабатывать веб-страницы, URL-адреса которых были добавлены в очередь на скачивание. Для этого необходимо осуществить эвристическую оценку важности для каждой веб-страницы, являющейся кандидатом на скачивание. В работе [5] стратегии отбора разделены на три категории, описанные ниже.

#### Стратегии, не использующие дополнительную информацию

Стратегии, отнесенные в данную категорию, используют информацию, полученную в процессе текущего обхода, и никакую более.

**Обход в ширину**. При использовании данной стратегии поисковый робот скачивает новые страницы, осуществляя обход в ширину веб-графа, начиная с главных страниц стартового множества. То есть веб-страница будет скачана тем раньше, чем раньше ссылка на нее была извлечена с

какой-либо скачанной веб-страницы. В работе [6] показано, что в проведенных экспериментах поисковый робот, использующий данную стратегию, выбирал важные страницы первыми.

**OPIC**. Данный метод описан в работе [7]. Этот метод также базируется на представлении веба как графа, описанному в разделе 1.1.1. В начале обхода каждой странице присваивается одинаковая стоимость. После загрузки страницы ее стоимость поровну распределяется между веб-страницами, на которые она ссылается. Более высокий приоритет имеют страницы с высокой текущей стоимостью. Вычисление стоимости аппроксимирует вычисление *PageRank*.

**Количество обратных ссылок**. Данная стратегия была описана в работе [8]. В этом алгоритме первыми скачиваются страницы, имеющие наибольшее число входящих ссылок.

**FICA**. Метод подробно описан в статье [9]. Алгоритм базируется на методике обучения с подкреплением, где для каждого узла веб-графа вычисляется ранг на основании расстояния между страницами.

### Стратегии, использующие информацию, полученную ранее

Стратегии данной категории используют информацию о рангах страниц, полученную в результате предыдущих обходов. При этом поисковый робот начинает обход со страниц, имеющих более высокий ранг, а ранги для страниц, найденных впервые, рассчитываются следующими способами:

- Запрашивая информацию о рангах у «оракула», располгающего полной информацией о веб-графе;
- Равновероятно выбирая одно из значений рангов страниц, полученных в предыдущих обходах;
- Назначаются нулем, то есть сначала скачиваются все страницы, известные ранее, а затем — новые;

- Как значение ранга страницы, указывающей на нее, деленное на число ее исходящих ссылок.

По результатам экспериментов в работе [5], стратегии данной категории преимущественно превосходят стратегии, не использующие дополнительной информации. При этом лучшим является метод, запрашивающий информацию о рангах у «оракула».

#### Стратегия с полной информацией

К данной категории относится одна стратегия *Omniscient*, использование которой подразумевает наличие «оракула», который вычисляет актуальный ранг для каждой веб-страницы. Для того, чтобы выбрать страницы из очереди на скачивание, поисковый робот осуществляет запрос к «оракулу», после чего выбирает страницы с наибольшим рангом. Данная стратегия быстрее всего находит важные страницы.

### 1.4.2. Стратегии повторного обхода

Стратегия повторного обхода определяет какие из уже скачанных страниц следует скачать еще раз. Оба типа стратегий описаны в работе [10]:

- *С одинаковой частотой.* Данная стратегия подразумевает, что веб-страницы будут скачиваться с одинаковой частотой, при этом их важность никак не будет учитываться.
- *С разными частотами.* При использовании данного типа стратегии, веб-страницы скачиваются с разными частотами. Например, с частотами, пропорциональными частотам изменения страниц.

# Глава 2. Предлагаемый метод

В ходе работы был разработан и реализован программный модуль, реализующий поискового робота. В разделе 2.1 описана предлагаемая стратегия приоритезации, а в разделе 2.2 описаны основные этапы алгоритма работы поискового робота.

## 2.1. Стратегия приоритезации

Основная идея стретегии приоритезации заключается в объединении стратегии отбора и стратегии повторного обхода посредством введения единого значения $VRank$ (visit rank). Вычисление такого ранга для веб-страницы $p$ осуществляется по следующей формуле:

$$VRank(p) = QRank(p) \cdot RVRank(p)$$

где $QRank(p)$ — эвристическая важность страницы, то есть ранг, характеризующий ее качество (quality rank), а $RVRank(p)$ — ранг, характеризующий необходимость повторно скачивать веб-страницу $p$ (re-visit rank).

Как было сказано в разделе 1.4.1, стратегии отбора, располагающие информацией о рангах веб-страниц, находят важные страницы раньше, чем стратегии, использующие информацию, полученную только в процессе текущего обхода. Потому для вычисления $QRank$ предлагается использовать «оракула», который сообщал бы ранги веб-страниц, найденных во время обхода. Поскольку информацией о полном веб-графе мы не располагаем, будет использовать метод машинного обучения, который можно отнести к категории поточечных подходов обучения ранжированию, описание которой есть в разделе 1.2.2. Суть метода заключается в использовании нейронной сети, задача которой — по URL-адресу веб-страницы получить число, имеющее смысл эвристической оценки важности страницы. Для чего на основании URL-адреса строится вектор статических особенностей, список

которых представлен в разделе 2.3. Нейронная сеть принимает на вход этот вектор и вычисляет ранг веб-страницы.

Для вычисления $QRank$ предлагается два метода. Первый базируется исключительно на статических особенностях URL-адреса, и для каждой веб-страницы $QRank$ вычисляется один раз при добавлении адреса в коллекцию. Второй также использует свойства веб-графа, а именно, после обработки скачананной страницы, ее $QRank$ распределяется между страницами, ссылки на которые она содержит. Таким образом, при добавлении нового URL-адреса $QRank$ считается при помощи нейронной сети, а затем он может изменятся в процессе дальнейшей работы.

Метод вычисления $RVRank$ основан на следующей идее. Заново обходить веб-страницы следует с частотой, пропорциональной частоте их изменения, потому для каждой страницы необходимо определить период ее изменения. Это осуществляется посредством итеративного приближения, то есть уточнения значения периода измения страницы при каждом ее скачивании. Кроме того, следует учитывать, что существуют динамические страницы, которые генерируются заново при переходе по URL-адресу. Динамические страницы не следует скачивать бесконечно часто. Это осуществляется, во-первых, за счет вероятного небольшого значения $QRank$ таких страниц, а во-вторых, путем введения нижней границы для значения периода изменения. Таким образом, при повторном скачивании страницы, необходимо проверить, была ли она изменена с момента предыдущего посещения и, если да, уменьшить предполгаемый период изменения (если он не достиг нижней границы), в противном же случае — увеличить.

## 2.2. Этапы алгоритма

Основные компоненты разработанного поискового робота представлены на рисунке 2.2.

Алгоритм включает в себя два этапа: подготовительный и основной.

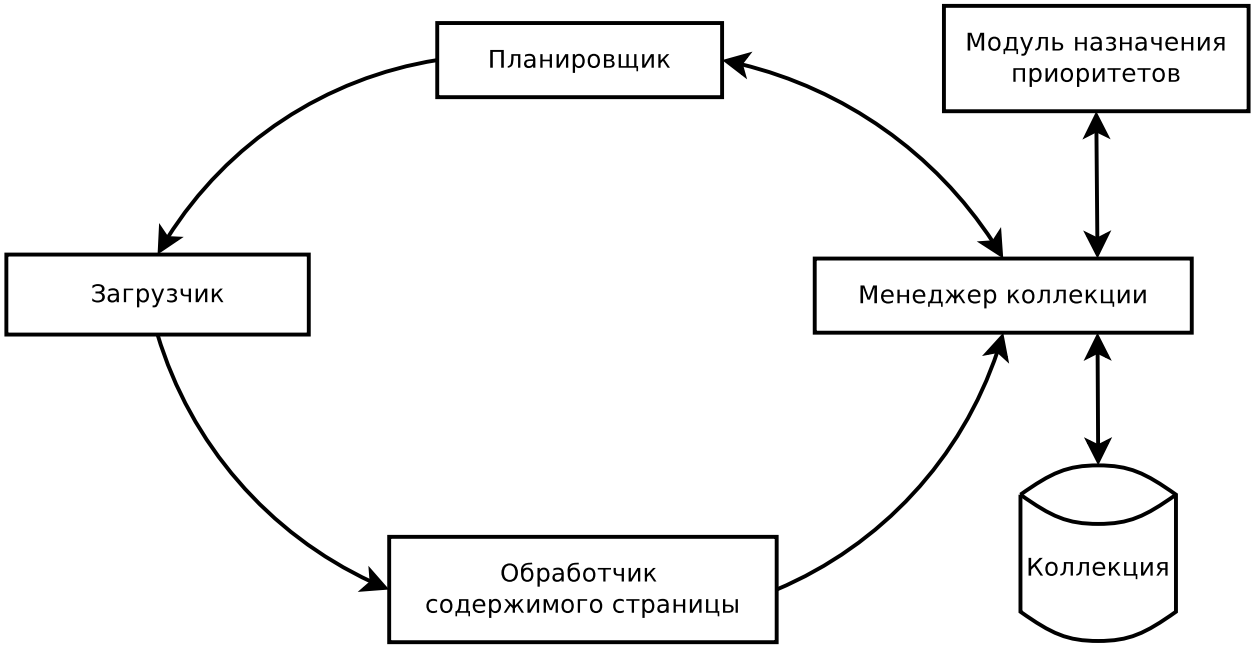


Рис. 2.1. Основные программные компоненты поискового робота

На подготовительном этапе происходит инициализация модуля назначения приоритетов, а на основном — непосредственно обход интернета. Рассмотрим подробнее каждый из этапов.

### Подготовительный этап

На подготовительном этапе производится обучение нейронной сети, которая входит в модуль назначения приоритетов. В качестве обучающего множества используется набор URL-адресов веб-страниц с известными рангами (например, $PageRank$). Затем в коллекцию добавляется стартовое множество URL-адресов, и для каждого из них вычисляется $QRank$.

### Основной этап

Схема работы алгоритма на основном этапе представлена на рисунке **??**. Работа происходит циклически, и каждом шаге цикла обрабатывается $K$ страниц из коллекции, имеющих наибольший $VRank$.

Цикл начинается с того, что планировщик запрашивает у менедже-

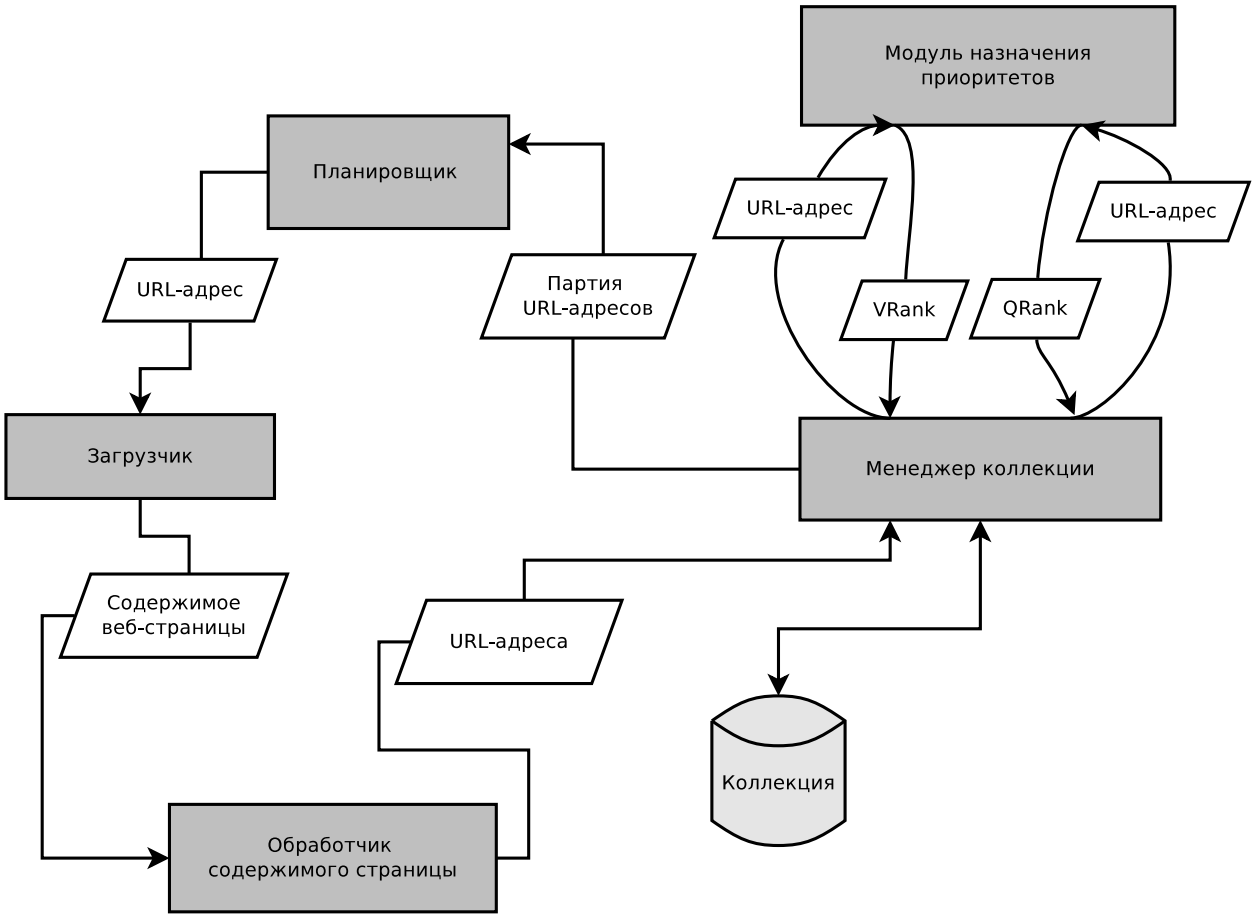


Рис. 2.2. Схема работы алгоритма на основном этапе

ра коллекции следующую партию адресов веб-страниц. После чего происходит подсчет $VRank$ для каждой страницы из коллекции посредством запросов менеджера коллекции к модулю приоритезации. Затем менеджер коллекции формирует партию, содержащую $K$ адресов веб-страниц, имеющих наибольший $VRank$ и отдает ее планировщику. После чего планировщик последовательно передает URL-адреса для дальнейшей обработки.

Обработка начинается с загрузки веб-страницы, получаемой по текущему URL-адресу. Следует помнить, что при загрузке следует учитывать правила вежливости, описанные в разделе 1.3, и в любой момент времени с каждым хостом должно быть установлено не более одного соединения. После загрузки следует проверить, обновилась ли страница с момента предыдущего посещения, чтобы осуществить пересчет предполагаемого периода

обновления для текущей страницы. После чего обработчик содержимого страницы извлекает ссылки с загруженной веб-страницы, нормализует их и передает полученые URL-адреса менеджеру коллекции. После чего модулем приоритезации рассчитываются $QRank$ для каждого URL-адреса, извлеченного с текущей страницы.

Когда все $K$ страниц из текущей партии будут обработаны, алгоритм перейдет к следующему шагу цикла.

## 2.3. Статические особенности URL-адреса

Для вычисления $QRank$ веб-страницы были использованы следующие статические особенности ее URL-адреса. Они были разделены на четыре группы.

### Особенности длины

В эту группу включены особенности, характеризующие длины компонент URL-адреса. Такие как:

- Суммарное число символов URL-адреса;
- Число символов схемы URL-адреса;
- Число символов хоста;
- Число символов пути;
- Число символов запроса.

### Орфографические особенности

В эту группу включены особенности, характеризующие орфографические свойства URL-адреса и его компонент. Такие как:

- Суммарное число цифр, входящих в URL-адрес;
- Суммарное число заглавных букв, входящих в URL-адрес;

- Число цифр, содержащихся в хосте;
- Число цифр, содержащихся в пути URL-адреса;
- Число цифр, содержащихся в запросе URL-адреса;
- Число заглавных букв, содержащихся в пути URL-адреса;
- Число заглавных букв, содержащихся в запросе URL-адреса.

### Количественные особенности термов

В данную группу включены особенности, характезующие количество термов в URL-адресе и его компонентах. Разделение на термы осуществлялось посредством разбиения адреса по символом, не являющимися буквами или цифрами. После чего выделялись следующие особенности:

- Суммарное число термов URL-адреса;
- Число термов хоста;
- Число термов пути;
- Число термов запроса.

### Словарные особенности термов

В эту группу включены компоненты, характеризующие частоту реальных слов в компонентах URL-адреса. А именно:

- Частота реальных слов среди термов, входящих в путь URL-адреса;
- Частота реальных слов среди термов, входящих в запрос.

# Глава 3. Экспериментальные результаты

Предлагаемый метод был реализован в виде программной системы на языке *Java* и протестирован.

## 3.1. Реализация

При реализации предлагаемого метода были реализованы следующие программные компоненты:

- Обертка для URL-адреса;
- Менеджер хоста;
- Модуль вежливости;
- Правило;
- Планировщик;
- Задание;
- Загрузчик веб-страниц;
- Обработчик содержимого страницы;
- Модули назначения приоритетов:
  - Стратегия обхода в ширину;
  - Стратегия, при которой для скачивания выбирается случайная страница из коллекции;
  - Использующий нейронную сеть;
  - Использующиуй нейронную сеть и свойства веб-графа;
- Контроллер приоритезации, использующей нейронную сеть;
- Компонент для извлечения особенностей URL-адреса страницы;
- Компонент, содержащий нейронную сеть;

- Словарный модуль;
- Менеджер коллекции.

UML-диаграмма классов представлена на рисунке 3.1. Рассмотрим детали реализации подробнее.

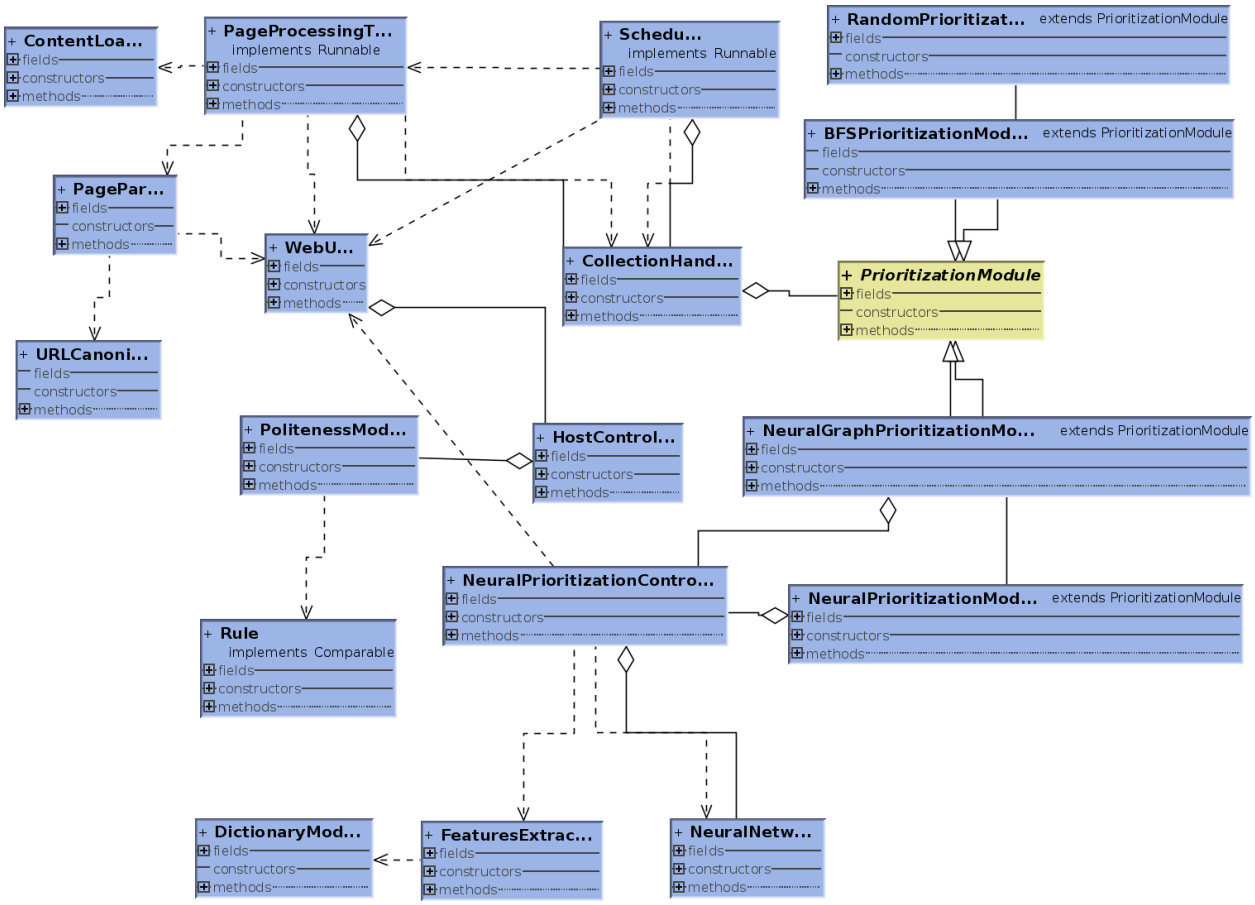


Рис. 3.1. Диаграмма классов

Основным объектом, с которым происходит работа в программе, является обертка для URL-адреса. Она реализована классом *WebURL*. Каждый экземпляр такого класса содержит мета-информацию о веб-странице, ее *QRank*, время последнего посещения, предполагаемый период обновления, ссылку на менеджер хоста и т.д.

Менеджер хоста реализован классом *HostController* и предназначен для выполнения правил вежливости для определенного хоста. В каждый момент времени с хостом должно быть установлено только одно соедине-

ние. Это реализовано за счет синхронизирования загрузок веб-страницы с одного хоста посредством использования поля *lock* менеджера хоста, являющегося экземпляром класса *Lock*. Кроме того, экземляр класса хранит на модуль вежливости, который определяет правила доступа к страницам хоста. Также менеджер хранит количество страниц с текущего хоста, имеющихся в коллекции.

Модуль вежливости предназначен для соблюдения правил вежливости, описанных в файле *robots.txt*, содержащемуся в корневом каталоге многих сайтов. Модуль вежливости реализован классом *PolitenessModule*, содержащий множество (экземляр класса *TreeSet*) правил, являющихся экземплярами класса *Rule*. Правило хранит текстовый шаблон (*Pattern*), удовлетворяющий набору каталогов сайта, и правило доступа (доступно или нет). Порядок на множестве введен для быстрого определения доступности того или иного URL-адреса.

Планировщик реализован классом *Scheduler*. Планировщик на каждом шаге цикла основного этапа алгоритма формирует $K$ заданий, являющихся экземплярами исполняемого (*Runnuble*) класса *PageProceccingTask*. Эти задания передаются исполнителю (экземпляр класса *ExecutorService*). Задание состоит из следующих инструкций для последовательной обработки. Загрузка, осуществляемая загрузчиком веб-страниц. Обработка содержимого и извлечение ссылок со страницы, осуществляемые обаботчиком содержимого страницы. И наконец отправка менеджеру коллекции для дальнейшей обработки. Обработка заданий осуществляется в несколько потоков.

Модуль назначения приоритетов является экземпляром класса, наследующего абстрактный класс *PrioritizationModule*. Разработаны два метода приоритезации: первый использует только нейронную сеть, а второй — нейронную сеть и анализ веб-графа. Реализованы они соотвественно классами *NeuralPrioritizationModule NeuralGraphPrioritizationModule*, исходный

код которых приведен в приложениях: А.1 и А.2. Кроме того, для сравнения с разработанными алгоритмами приоритезации были реализованы следующие стратегии: стратегия отбора методом обхода в ширину, реализованная классом *BFSPrioritizationModule*, и стратегия, при которой страница для скачивания выбирается из коллекции случайным образом, реализованная классом *RandomPrioritizationModule*.

Компонент для извлечения особенностей URL-адреса реализован классом *FeaturesExtractor*. Исходный программный код класса приведен в приложении А.3. Для каждой из особенностей адреса компонент реализует его численное представление в интервале от 0 до 1 посредством нормировки. Для извлечения словарных особенностей термов конструируется словарный модуль, использующий слова английского и русского языков из словарей с сайта [11]. Причем слова русского языка были заменены на их транслитерационный вид.

Нейронная сеть была реализована с использованием библиотеки *Encog* [12]. Сеть содержит три слоя нейронов. Входной слой содержит число нейронов, соответствующее числу особенностей URL-адреса веб-страницы (максимальное число особенностей — 16). Слой имеет сигмоидную активирующую функцию. Число нейронов в скрытом слое зависит от числа нейронов входного слоя. А именно — число нейронов скрытого слоя в три раза больше. Скрытый слой так же имеет сигмоидную активирующую функцию. Выходной слой содержит один нейрон и сигмоидную активирующую функцию. Для обучения использовался метод быстрого распространения (*Quickprop*), являющийся улучшением метода обратного распространения ошибки. Подробно он описан в статье [13]. Данная конфигурация сети была подобрана в ходе тестирования на множестве URL-адресов веб-страниц с известными $PageRank$, полученными с использованием *google toolbar*. Именно с такими параметрами отклонение вычисленного ранга от реального $PageRank$ было минимальным.

Менеджер коллекции реализован классом *CollectionHandler*. Он содержит ссылку на модуль приоритезации, посредством которого вычисляются ранги для URL-адресов веб-страниц, передаваемых в коллекцию. Коллекция представлена экзампляром класса *ConcurrentHashMap*, где ключом является строка, а значением — *WebURL*. Выбрана потокобезопасная реализация ассоциативного массива, поскольку добавление адресов происходит в несколько потоков. Также менеджер коллекции содержит множество менеджеров хостов для того, чтобы контролировать количество добавляемых в коллекцию страниц. Кроме того, менеджер коллекции осуществляет формирования пакета веб-страниц для обработки по запросу планировщика. Формирование осуществляется путем подсчета $VRank$ для каждой страницы коллекции и выбора $K$ страниц с наибольшим таким рангом. Для эффективной реализации такой сортировки был реализован компаратор *ValueComparator*, осуществляющий сортировку в *Map* по значению. В процессе формирования партии веб-страниц контролируется, чтобы она не содержала более определенного в планировщике числа страниц с одного сайта. Это необходимо для того, чтобы поисковый робот не зацикливался на одном сайте, что привело бы к замедлению работы.

## 3.2. Параметры запусков

Для сравнения работы различных стратегий приоритезации были произведены тестовые запуски поискового робота на сайтах, принадлежищих домену *.ru*. Стартовое множество URL-адресов состояло из главных страниц ста популярных сайтов, принадлежащих тому же домену. Максимальное число обрабатываемых сайтов составляло 5000, при этом с каждого сайта обрабатывалось не более 500 страниц. Размер партии страниц, обрабатываемых на одной итерации цикла составляло $K = 1000$, причем максимальное число страниц с одного сайта в ней составяло 50. Обработка

страниц происходила в 20 параллельных потоков.

Нейронная сеть была обучена на случайной выборке размера 50000 URL-адресов с домена *.ru*, для которых были известны их $PageRank$, полученные посредством использования веб-сервиса *Google Toolbar*, за счет осуществления запросов к $toolbarqueries.google.com$. Данный сервис по URL-адресу страницы возвращает целое число из отрезка от $-1$ до 10, являющееся $GooglePageRank$ данной страницы, то есть оценкой важности страницы по мнению *Google*, которое вследствие линейной нормировки приводилось к действительному числу из отрезка от 0 до 1. Для оценки качества работы обученной нейронной сети проводилась ее проверка на случайной выбрке URL-адресов с известными $GooglePageRank$ размера 200000, принадлежащих домену *.ru*.

## 3.3. Результаты

Значения $PageRank$ URL-адресов страниц, вычисленные нейронной сетью, отличаются от реальных значений *Google PageRank* в среднем на 14%.

Были протестированы стратегии приоритезации, реализующие два предложенных метода, стратегия, использующая обход в ширину, а также стратегия, при которой страницы из очереди выбирались случайным образом с равными вероятностями. В процессе обхода интернета при каждом запуске в коллецию добавлялось около 200 тысяч URL-адресов, из которых были скачаны 25 тысяч страниц с адресами, имеющими наибольший приоритет. Для всех скачанных в процессе запусков страниц был получен их $PageRank$ с использованием сервиса *Google Toolbar*, являющийся показателем их качества. На рисунке 3.2 приведен график завимости среднего $PageRank$ страниц от числа скачанных для каждой из протестированных стратегий приортезации.

Соответственно, стратегия тем лучше, чем выше $PageRank$ стра-

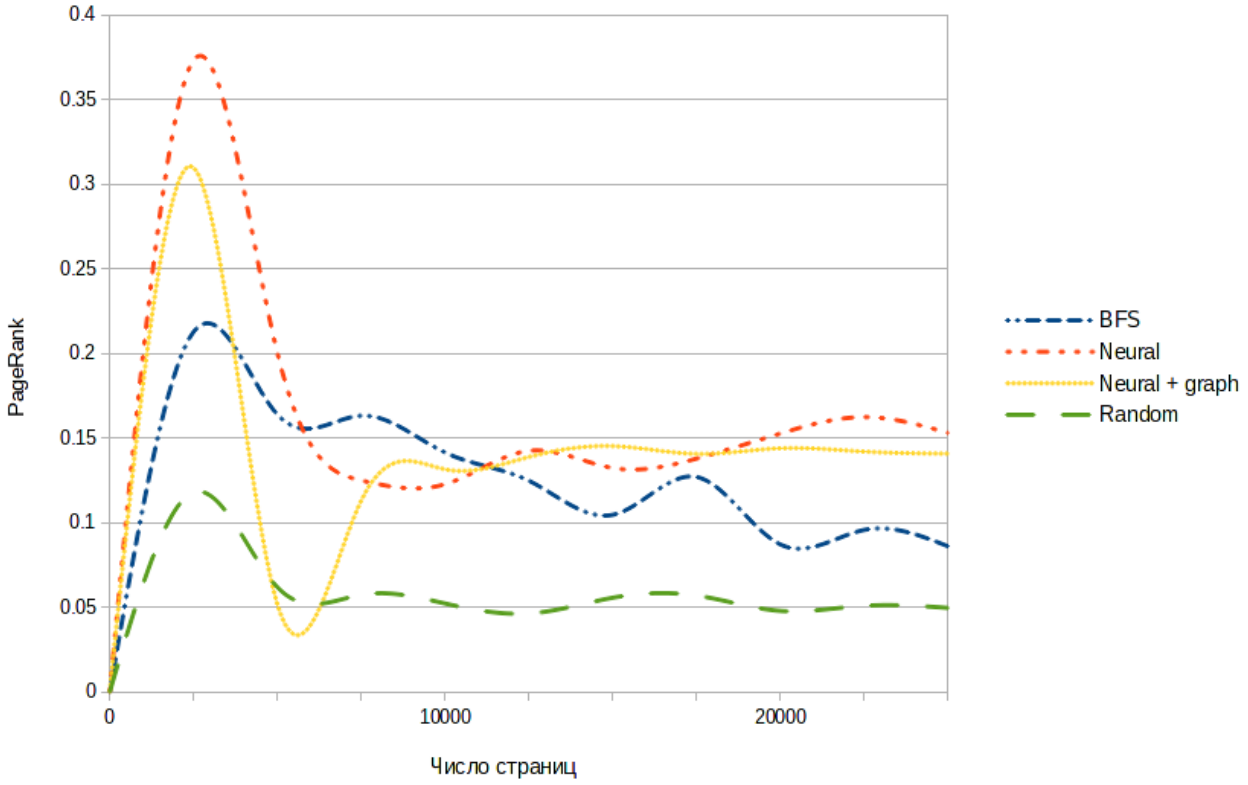


Рис. 3.2. График зависимости среднего *PageRank* страниц от скачанного числа страниц. *BFS* — стратегия обхода в ширину. *Neural* — стратегия приоритезации, использующая нейронную сеть. *Neural + Graph* — стратегия приоритезации использующая нейронную сеть и веб-граф. *Random* — стратегия приоритезации, случайно выбирающая из коллекции страницу для скачивания.

ниц, скачанных в процессе обхода интернета поисковым роботом, использующим ее, то есть, чем выше линия на графике.

## 3.4. Выводы

Полученные результаты экспериментального сравнения позволяют сделать следующие выводы:

1. Стратегия приоритезации, использующая метод обхода в ширину, превосходит стратегию, случайно выбирающую страницы.
2. Предложенные в работе стратегии приоритезации поискового робота, превосходят стандартную стратегию обхода в ширину.

3. Результаты обхода интернета с использованием двух предложенных стратегий отличаются незначительно.

# Заключение

В работе получены следующие результаты

- Предложен метод ранжирования веб-страниц, использующий нейронную сеть, основанный на использовании статических особенностей URL-адресов страниц.
- Разработаны две стратегии приоритезации поискового робота, использующие предложенный метод ранжирования.
- Реализован поисковый робот с целью сравнения стратегий приоритезации.
- Были произведены запуски поискового робота, использующего различные стратегии приоритезации.

Полученные результаты позволяют сделать вывод, что разработанный метод ранжирования позволяет достаточно точно определять значения ранга веб-страниц по их URL-адресу. При этом, поисковый робот, использующий стратегию приоритезации, базирующуюся на данном методе, более эффективен, чем стандартный, осуществляющий обход в ширину, поскольку выбирает для скачивания более важные веб-страницы.

# Источники


[1] *Brin S., Page L.* The anatomy of a large-scale hypertextual web search engine // *Comput. Netw. ISDN Syst.* 1998. Vol. 30, no. 1-7. Pp. 107–117. http://dx.doi.org/10.1016/S0169-7552(98)00110-X.

[2] *Wikipedia.* Алгоритм HITS — Wikipedia, The Free Encyclopedia. 2013. [Online; accessed 15-June-2013].

[3] *Liu T.-Y.* Learning to rank for information retrieval // *Found. Trends Inf. Retr.* 2009. Vol. 3, no. 3. Pp. 225–331. http://dx.doi.org/10.1561/1500000016.

[4] *Manning C. D., Raghavan P., Schütze H.* Introduction to Information Retrieval. New York, NY, USA: Cambridge University Press, 2008.

[5] *Castillo C.* Effective web crawling // *SIGIR Forum.* 2005. Vol. 39, no. 1. Pp. 55–56. http://doi.acm.org/10.1145/1067268.1067287.

[6] *Najork M., Wiener J. L.* Breadth-first crawling yields high-quality pages // Proceedings of the 10th international conference on World Wide Web. WWW '01. New York, NY, USA: ACM, 2001. Pp. 114–118. http://doi.acm.org/10.1145/371920.371965.

[7] *Abiteboul S., Preda M., Cobena G.* Adaptive on-line page importance computation // Proceedings of the 12th international conference on World Wide Web. WWW '03. New York, NY, USA: ACM, 2003. Pp. 280–290. http://doi.acm.org/10.1145/775152.775192.

[8] *Cho J., Garcia-Molina H., Page L.* Efficient crawling through url ordering // *Comput. Netw. ISDN Syst.* 1998. Vol. 30, no. 1-7. Pp. 161–172. http://dx.doi.org/10.1016/S0169-7552(98)00108-1.

[9] *Bidoki A. M. Z., Yazdani N., Ghodsnia P.* Fica: A novel intelligent crawling algorithm based on reinforcement learning // *Web Intelli. and Agent Sys.* 2009. Vol. 7, no. 4. Pp. 363–373. http://dx.doi.org/10.3233/WIA-2009-0174.

[10] *Cho J., Garcia-Molina H.* The evolution of the web and implications for an incremental crawler // Proceedings of the 26th International Conference on Very Large Data Bases. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Pp. 200–209. http://dl.acm.org/citation.cfm?id=645926.671679.

[11] WinEdt. [Online; accessed 15-April-2013]. http://www.winedt.org/.

[12] Encog Machine Learning Framework. http://www.heatonresearch.com/encog.

[13] *Fahlman S. E.* An empirical study of learning speed in back-propagation networks: Tech. rep.: 1988.

# Приложение А

## Программный код

### А.1. Программный код модуля модуля назначения приоритетов, использующего нейронную сеть

```
1  package ru.ifmo.mailru.priority;
2
3  import ru.ifmo.mailru.core.Page;
4  import ru.ifmo.mailru.core.WebURL;
5  import ru.ifmo.mailru.features.NeuralPrioritizationController;
6
7  import java.io.FileNotFoundException;
8
9  /**
10  * @author Anastasia Lebedeva
11  */
12 public class NeuralPrioritizationModule extends PrioritizationModule {
13     private NeuralPrioritizationController neuralPrioritizationController
           ;
14
15     public NeuralPrioritizationModule() throws FileNotFoundException {
16         neuralPrioritizationController = new
               NeuralPrioritizationController();
17         int positiveSize = neuralPrioritizationController.
               positiveExamples.size();
18         int negativeSize = neuralPrioritizationController.
               negativeExamples.size();
19         neuralPrioritizationController.train(positiveSize, negativeSize);
20     }
21
22     @Override
23     public void setQualityRanks(WebURL url, Page parentPage) {
24         Double computedRank = neuralPrioritizationController.computeRank(
               url);
25         url.setQualityRank(computedRank);
26     }
27
28     @Override
29     public void resetQualityRanks(WebURL url, Page parentPage) {
30         //To change body of implemented methods use File | Settings |
               File Templates.
31     }
```

```
}
```

Листинг А.1. Модуль назначения приоритетов, использующий нейронную сеть

## А.2. Программный код модуля модуля назначения приоритетов, использующего нейронную сеть и свойства веб-графа

```
package ru.ifmo.mailru.priority;

import ru.ifmo.mailru.core.Page;
import ru.ifmo.mailru.core.WebURL;
import ru.ifmo.mailru.features.NeuralPrioritizationController;

import java.io.FileNotFoundException;

/**
 * @author Anastasia Lebedeva
 */
public class NeuralGraphPrioritizationModule extends PrioritizationModule
    {
    private NeuralPrioritizationController neuralPrioritizationController
        ;

    public NeuralGraphPrioritizationModule() throws FileNotFoundException
        {
        neuralPrioritizationController = new
            NeuralPrioritizationController();
        int positiveSize = neuralPrioritizationController.
            positiveExamples.size();
        int negativeSize = neuralPrioritizationController.
            negativeExamples.size();
        neuralPrioritizationController.train(positiveSize, negativeSize);
    }


    @Override
    public void setQualityRanks(WebURL url, Page parentPage) {
        double addition = 0;
        if (parentPage != null) {
            addition = parentPage.getUrl().getQualityRank() / parentPage.
                getOutLinks().size();
        }
        Double computedRank = neuralPrioritizationController.computeRank(
            url);
        url.incrementQualityRank(computedRank + addition);
    }

    @Override
    public void resetQualityRanks(WebURL url, Page parentPage) {
        double addition = parentPage.getUrl().getQualityRank() /
            parentPage.getOutLinks().size();
        url.incrementQualityRank(addition);
    }
}
```

Листинг А.2. Модуль назначения приоритетов, использующий нейронную сеть и свойства веб-графа

## А.3. Программный код модуля выделения статических особенностей URL-адреса

```
package ru.ifmo.mailru.features;

import ru.ifmo.mailru.core.WebURL;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class FeaturesExtractor {
  private List<Double> features;
  private String[] components;
  private final int COMPONENT_NUMBER = 5;
    private static double[][] MAXIMUMS;

    /**
     * row 0: length features
     * row 1: orthographic features (digits)
     * row 2: orthographic features (capital case)
     * row 3: number of term
     * row 4: dictionary features
     */

    {
        MAXIMUMS = new double[COMPONENT_NUMBER][4];
        for (int j = 0; j < 3; j++) {
            for (int i = 3; i < COMPONENT_NUMBER; i++) {
                MAXIMUMS[i][j] = 300;
            }
        }
        for (int i = 0; i < 3; i++) {
            MAXIMUMS[0][i] = 650;
            MAXIMUMS[1][i] = 5;
            MAXIMUMS[2][i] = 50;
        }
        for (int i = 3; i < COMPONENT_NUMBER; i++) {
            MAXIMUMS[i][3] = 100;
        }
        MAXIMUMS[0][3] = 300;
        MAXIMUMS[1][3] = 1;
        MAXIMUMS[2][3] = 10;
    }

    public FeaturesExtractor(String url) throws URISyntaxException {
        this(new WebURL(new URI(url)));
    }

  public FeaturesExtractor(WebURL url) {
    components = new String[COMPONENT_NUMBER];
        URI uri = url.getUri();
        components[0] = uri.toString();
    components[1] = uri.getScheme();
    components[2] = uri.getHost();
```

```
56      components[3] = uri.getPath();
        components[4] = uri.getQuery();
58          for (int i = 1; i < COMPONENT_NUMBER; i++) {
                if (components[i] == null) {
60                  components[i] = "";
                }
62          }
        features = new ArrayList<>();
64  }

66  private void extractLengthFeatures() {
      for (int i = 0; i < COMPONENT_NUMBER; i++) {
68      features.add(components[i].length() / MAXIMUMS[i][0]);
      }
70          String[] path = components[3].split("/");
            features.add((double) path.length / 20);
72  }

74  private void extractOrthographicFeatures() {
      Pattern digit = Pattern.compile("\\d+");
76          double commonCount = 0;
      for (int i = 2; i < COMPONENT_NUMBER; i++) {
78      Matcher m = digit.matcher(components[i]);
        double count = 0;
80      while (m.find()) {
          count += m.group().length();
82      }
                commonCount += count;
84      features.add(count / MAXIMUMS[i][1]);
      }
86          features.add(commonCount / MAXIMUMS[0][1]);
      Pattern capitalCase = Pattern.compile("[A-Z]+");
88          commonCount = 0;
      for (int i = 3; i < COMPONENT_NUMBER; i++) {
90      Matcher m = capitalCase.matcher(components[i]);
        double count = 0;
92      while (m.find()) {
          count += m.group().length();
94      }
                commonCount += count;
96      features.add(count / MAXIMUMS[i][2]);
      }
98          features.add(commonCount / MAXIMUMS[0][2]);
  }
100
      private void extractTermFeature() {
102         List<String[]> terms = extractTerms();
            extractCountTerm(terms);
104         extractWordFrequency(terms);
      }
106
      private void extractWordFrequency(List<String[]> terms) {
108         for (int i = 3; i < COMPONENT_NUMBER; i++) {
                if (terms.get(i).length == 0) {
110                 features.add(1.0);
                    continue;
112             }
                double words = 0;
114             for (String s : terms.get(i)) {
                    if (DictionaryModule.isWord(s)) {
```

```
116                         words++;
                        }
118                 }
                    features.add(words / terms.get(i).length);
120             }
        }
122
        private void extractCountTerm(List<String[]> terms) {
124             for (int i = 0; i < COMPONENT_NUMBER; i++) {
                    features.add(terms.get(i).length / MAXIMUMS[i][3]);
126             }
        }
128
        private List<String[]> extractTerms() {
130             List<String[]> terms = new ArrayList<>(COMPONENT_NUMBER);
                for (int i = 0; i < COMPONENT_NUMBER; i++) {
132                 terms.add(DictionaryModule.splitIntoTokens(components[i]));
                }
134             return terms;
        }
136

138     public Double[] buildVector() {
          extractLengthFeatures();
140       extractOrthographicFeatures();
              extractTermFeature();
142       Double[] res = new Double[features.size()];
          return features.toArray(res);
144     }
    }
```

Листинг А.3. Модуль выделения статических особенностей URL-адреса