УДК 004.624 DOI: 10.17587/prin.8.490-503

В. А. Васенин, д-р физ.-мат. наук, проф., e-mail: vasenin@msu.ru, НИИ Механики МГУ имени М. В. Ломоносова, **М. Д. Дзабраев,** разработчик, e-mail: dzabraew@gmail.com, ИАС ИСТИНА, г. Москва

Методы и средства мониторинга публикаций в средствах массовой информации

Описан подход к решению востребованной на практике задачи извлечения данных с веб-сайтов в целях их дальнейшей обработки для тех или иных приложений. Изложено описание и детали реализации алгоритма, с помощью которого представляется возможным осуществить обход веб-страницы. Целью обхода веб-страницы является попадание во все возможные места на веб-странице и извлечение полезных данных. Обход веб-страницы осуществляется путем нажатия кнопок на веб-странице. Нажатие каждой кнопки способно либо загрузить новую веб-страницу, либо модифицировать существующую с помощью исполнения JavaScript. Алгоритм, описанный в настоящей статье, предназначен для реализации нажатия кнопок, которые изменяют текущую веб-страницу.

Ключевые слова: *извлечение данных, веб, readability, обход веб-сайта, обход веб-страницы, Javascript, Firefox*

Введение

Средства массовой информации с использованием современных инфокоммуникационных технологий перманентно генерируют, аккумулируют и хранят огромные объемы сведений, имеющих отношение к тем или иным субъектам политической и экономической, научной-технической и хозяйственной, социальной и других сфер деятельности. Эти сведения способствуют формированию отношения отдельных людей, а также больших социальных групп к этим субъектам, влияют на их имидж в обществе. Как следствие, сформированное отношение оказывает прямое или опосредованное воздействие на результаты деятельности перечисленных субъектов. С учетом отмеченных обстоятельств задача создания эффективных средств автоматизации процессов выявления текстов в средствах массовой информации (далее СМИ), имеющих отношение к отдельным субъектам перечисленных выше сфер деятельности в обществе, приобретает особую актуальность.

В работе [1] представлены подходы к решению этой задачи, макет программных средств ее решения и результаты анализа тональности публикаций в СМИ с использованием таких средств. Одним из существенных недостатков этого макета является отсутствие в нем механизмов эффективного перманентного мониторинга заранее определенного и достаточно широкого информационного пространства (множества веб-сайтов) для формирования исходной коллекции документов в целях ее последующего анализа. Это обстоятельство значительно сужало функциональные возможности представленного в работе [1] макета для комплексного решения поставленной выше задачи.

В настоящей статье изложены модельные представления, алгоритмы и реализующие их программные механизмы, которые выполняют перманентный мониторинг заданного множества веб-сайтов в целях обнаружения и извлечения содержащихся в них новых публикаций. Извлеченные публикации при этом сохраняются в специально созданную базу данных (далее БД) для последующего их анализа с той или иной целью. Совокупность программ, которые реализуют перечисленные выше функциональные возможности, далее будем именовать сервисом.

На настоящее время сервис в автоматическом режиме осуществляет извлечение публикаций с четырех веб-сайтов новостных агентства: ria.ru, lenta.ru, gazeta.ru, mk.ru. Количество новостных сообщений (публикаций), хранящихся на данный момент в БД, насчитывает 3 млн 200 тыс. Средствами этого сервиса осуществляется мониторинг публикаций на предмет содержания одного из заранее заданных названий организации с последующей классификацией публикаций на три класса тональности.

В следующем разделе представлено описание модели, на основе которой построен алгоритм, положенный в основу сервиса, осуществляющего извлечение публикаций из веб-сайтов.

Автоматическое извлечение данных из веб-сайтов

Веб-сайт представляет собой набор документов. Каждый документ обладает уникальным адресом — URL. Как правило, документ — это смесь текста на естественном языке и HTML-разметки. Практически всегда вместе с документом распространяются стили (CSS), с помощью которых определяется вид документа и программы на языке программирования JavaScript. Стили и JavaScript-программы создают разработчики веб-сайта.

С помощью JavaScript-кода страницы веб-сайта могут быть наделены динамическим (интерактивным) поведением. Примером такого поведения является отслеживание событий, которые порождаются пользователем. Можно запрограммировать реакцию веб-страницы на то или иное событие, которое является результатом действий пользователя. С помощью JavaScript-кода с содержанием HTML-документа можно сделать многое, вплоть до полного его удаления. Пример изменения документа с помощью JavaScript-кода представлен на рис. 1 (см. вторую сторону обложки). На данном рисунке выделена кнопка, при каждом нажатии которой в документ (веб-страницу) встраивается список ссылок на новостные сообщения.

Для извлечения данных с веб-сайта необходимо собрать коллекцию URL, которые указывают на обрабатываемый веб-сайт. При этом желательно суметь собрать как можно больше URL. Причина в том, что если какой-то URL оказывается пропущенным, то вместе с ним упущенным становится и документ, который соответствует данному URL.

Поскольку чрезвычайно важным является сбор как можно большего числа URL, то это вынуждает разработчика писать программы, которые не просто загружают HTML-документ по данному URL и извлекают из него все URL, но также поддерживают динамический режим взаимодействия с веб-сайтом, "нажимая" кнопки и генерируя различные события. Иными словами, программа, которая проводит сбор URL, должна уметь исполнять JavaScript-код. Причина в том, что каждое нажатие той или иной кнопки может встроить в документ набор URL, которые ранее не были обнаружены.

Например, если нажимать отмеченную на рис. 1 (см. вторую сторону обложки) кнопку, то последовательно в документ будут встроены все ссылки на все публикации для данного агентства. Если программа будет обеспечивать такое нажатие на указанную кнопку, то она гарантированно извлечет URL всех публикаций.

Можно попытаться не исполнять JavaScript-код, а поступить напрямую: загружать документ, извлекать из него все URL; добавлять каждый URL в БД; аналогично поступать для каждого URL, который хранится в БД. Однако, исполняя JavaScript-код, программа сможет собрать не меньшее количество URL, чем без исполнения JavaScript-кода.

Обход веб-страницы с исполнением JavaScript-кода

Рассмотрим неформальное описание модели, которая положена в основу алгоритма, реализующего автоматический обход веб-страницы. Этот алгоритм предписывает, как нужно обеспечивать нажатие кнопок на веб-странице, чтобы программа смогла извлечь наибольшее число URL. Таким образом, алгоритм преследует следующую цель: в результате нажатия кнопок программа должна попасть во все части веб-страницы, куда такая возможность предоставляется и, оказавшись в каждой из этих частей, извлечь интересующие данные. Алгоритм необходим, чтобы полностью или частично снять с программиста работу по написанию кода для осуществления нажатий кнопок. Перед формализованным изложением алгоритма рассмотрим пример, с помощью которого на вербальном уровне излагаются исходные посылки модели, используемой для его построения, на основе которой разработан этот алгоритм.

На рис. 2 (см. вторую сторону обложки) представлен пример двух кнопок, которые обведены красными овалами. При нажатии кнопки "выбрать по дате" исполняется JavaScript-код, который встраивает в документ прямоугольник "Архив новостей". Этот прямоугольник содержит кнопку "стрелка влево" и набор гиперссылок (URL) в виде календаря. При нажатии кнопки "стрелка влево" будет выполнен JavaScript-код, который обновит содержимое календаря путем удаления имеющихся гиперссылок и добавлением новых, имеющих отношение к предыдущему месяцу гиперссылок. При этом, если нажать "выбрать по дате", затем несколько раз нажимать "стрелка влево", а после этого нажать в любую точку документа, которая не принадлежит прямоугольнику "Архив новостей", то выполнится JavaScript-код, который удалит прямоугольник "Архив новостей". Вместе с прямоугольником будет удалена кнопка "стрелка влево". Возможна ситуация, когда кнопку "стрелка влево" можно было бы нажать еще раз и получить URL, который ранее не встречался. В связи с этим обстоятельством необходимо, чтобы программа запомнила то место, где была ненажатая кнопка "стрелка влево", и запомнила путь, по которому она попала в это место. В дальнейшем программа должна вновь пройти по пути, который был запомнен ранее, и нажать еще не нажатую кнопку.

Перед формальным описанием алгоритма следует подробнее остановиться на том, как устроен HTML-документ и каким образом разработчики веб-сайта встраивают JavaScript-код в такой документ.

Всякий HTML-документ представляет собой дерево HTML-тегов. Дерево HTML-тегов принято называть DOM-дерево. Пример DOM-дерева изображен на рис. 3.

Для понимания принципов работы алгоритма, который будет представлен далее, важно понимать, что такое кнопка в документе с точки зрения DOM-дерева. Кнопка, на которую пользователь мо-

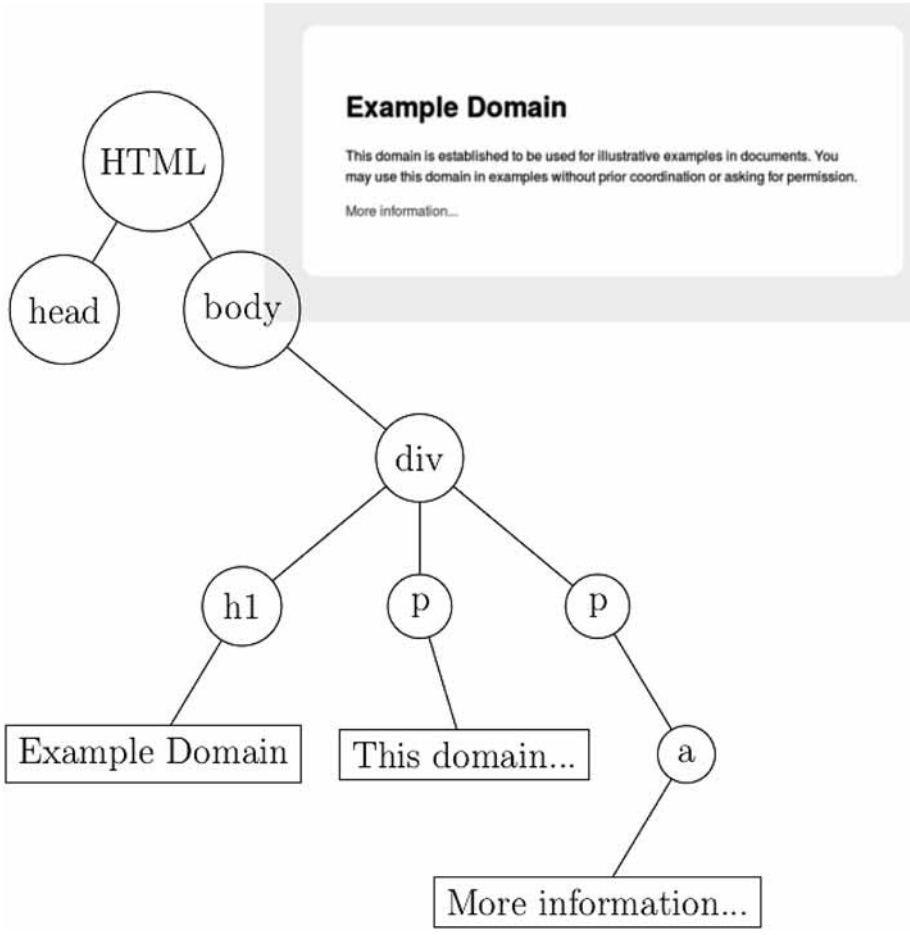

Рис. 3. Фрагмент веб-сайта example.com, который можно наблюдать через веб-браузер, и дерево HTML-тегов, соответствующее фрагменту

жет нажать, является узлом DOM-дерева. При этом отметим, что на этот узел разработчики веб-сайта прикрепили один или более обработчиков событий. Обработчик события представляет собой следующую пару: функция на языке JavaScript; тип события. Если на узел DOM-дерева было сгенерировано событие некоторого типа (обозначим название типа символом T), то будут выбраны все пары, у которых второй компонент совпадает с T, и будут исполнены все функции из первого компонента пары. В рамках алгоритма будет рассматриваться только событие click — нажатие левой кнопки мыши. Таким образом, кнопкой называется узел DOM-дерева, на котором существует хотя бы один обработчик события click.

Для работы алгоритма необходимо отличать одну кнопку от другой. В основу такого различия можно было бы положить следующий способ: каждой кнопке сопоставить путь в DOM-дереве от корня; если пути различаются, то и кнопки можно считать отличающимися. Вместе с тем такой способ не подходит, поскольку при нажатии какой-либо кнопки местоположение нажатой или другой кнопки в дереве может измениться. При повторной загрузке с сервера документа, он также может выглядеть немного иначе с точки зрения древовидной структуры. При этом одни и те же кнопки могут иметь различные пути от корня.

Наиболее корректным представляется следующее отношение равенства, которое неформально можно определить следующим образом: если две кнопки при нажатии на выходе реализуют один и тот же результат, то это одна и та же кнопка. Практическая реализация этого критерия равенства представляет большую сложность. Дело в том, что функции-обработчики, скорее всего, будут использовать на чтение и запись глобальные переменные, а также будут читать/изменять атрибуты DOM-дерева. Ввиду большой сложности программной реализации данного критерия равенства, в предлагаемом подходе этот критерий использоваться не будет.

Введем следующее отношение равенства, которое близко к отношению "делают одно и то же". Каждой кнопке ставятся в соответствие два параметра: текст внутри кнопки; список исходных текстов на JavaScript обработчиков событий. Отношение равенства выглядит следующим образом. Если у двух кнопок указанные параметры совпадают, то это одна и та же кнопка.

Для изложения алгоритма необходимо ввести определение состояния веб-страницы. После загрузки веб-страница имеет некоторое начальное состояние. После нажатия кнопки может ничего не происходить и, таким образом, страница остается в неизменном состоянии. Если после нажатия что-то изменилось, то веб-страница находится в другом состоянии. Например, на рис. 2 (см. вторую сторону обложки) можно выделить следующую цепочку состояний.

1. Состояние после загрузки страницы. Ни одна кнопка не была нажата.

2. После нажатия кнопки "выбрать по дате" в страницу встраивается прямоугольник "Архив новостей". Этот прямоугольник помимо прочего привнес новую кнопку и набор URL. При этом заметим, что встроенных URL до нажатия кнопки на странице не было.

3. После нажатия кнопки "стрелка влево" происходит обновление календаря в прямоугольнике. В процессе обновления календаря происходит удаление имеющихся URL и встраивание новых URL. В результате выполненных изменений состояние веб-страницы изменилось.

4. Если теперь нажать мышью вне прямоугольника, то прямоугольник "Архив новостей" пропадет,

и веб-страница окажется в исходном состоянии (как после загрузки страницы).

На веб-страницу можно смотреть, как на (конечный) автомат. В качестве объектов, которые подаются на его вход, можно рассматривать события нажатия кнопок. При получении объекта на вход состояние автомата меняется.

Выходом автомата является та веб-страница, которую видит пользователь.

Поскольку цель обхода веб-страницы заключается в извлечении URL, то целесообразно определять контекст страницы по набору URL и по набору кнопок. Причина в том, что нажатие любой кнопки может породить операцию встраивания новых URL в страницу. Таким образом, введем следующее определение.

Состоянием веб-страницы называется тройка (B, N, H), где $B = \{b_1,..., b_n\}$ представляет множество кнопок; N — число URL, которые встречаются в данном состоянии страницы. Пусть $h_1,..., h_N$ — множество всех URL на странице. Пусть f (*str*) — хэш-функция (на выходе целое число). Тогда третий параметр определяется соотношением $H = \sum_{i=1}^{N} f(h_i)$.

Изложим далее алгоритм обхода веб-страницы. В процессе обхода веб-страницы будет осуществляться построение графа состояний. Граф, который будет построен в результате обхода веб-страницы, выступает в роли хранителя кнопок. Если в результате нажатия кнопки веб-страница перешла в новое состояние, а в старом состоянии остались ненажатые кнопки, то необходимо запомнить информацию о ранее ненажатых кнопках, а также путь, по которому алгоритм пришел в вершину (состояние), чтобы вновь прийти в это состояние и нажать ненажатые кнопки. В графе каждая вершина будет соответствовать состоянию веб-страницы. Каждой вершине приписывается множество кнопок. Вершины соединяются ориентированными ребрами. Каждое ребро соответствует переходу из одного состояния в другое посредством нажатия кнопки. Каждому ребру при этом приписывается кнопка, с помощью которой был осуществлен переход.

Далее будет представлено формальное описание алгоритма, построенного на базе неформально представленной модели.

Формальное описание алгоритма

Шаг 1. Осуществляется загрузка страницы. После загрузки веб-страницы вычисляются параметры (B_1, N_1, H_1). Затем в граф добавляется корневая вершина. Корневой вершине приписываются вычисленные параметры. Корневой вершине назначается статус текущей вершины. Корневая вершина изображена на рис. 4.

Шаг 2. Если множество B_1 пусто, то обход закончен. Иначе выбирается кнопка $b \in B_1$, после чего она нажимается.

Шаг 3. После того, как нажатие кнопки завершилось, вычисляются параметры (B_2, N_2, H_2).

Шаг 4. Если (B_2, N_2, H_2) $\neq$ (B_1, N_1, H_1), то в граф состояний добавляется новая вершина. Новая вершина назначается текущей вершиной (рис. 5).

В противном случае реализуется петля (рис. 6).

Каждому ребру приписывается список пар (*b*, *hist*): кнопка; история нажатий. Первый элемент пары представляет кнопку, при нажатии которой был осуществлен переход по ребру. Второй параметр пары — история нажатий — является второстепенным, по-

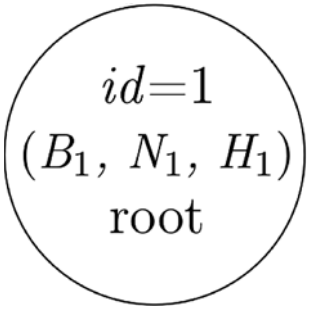

Рис. 4. Корневая вершина в графе состояний

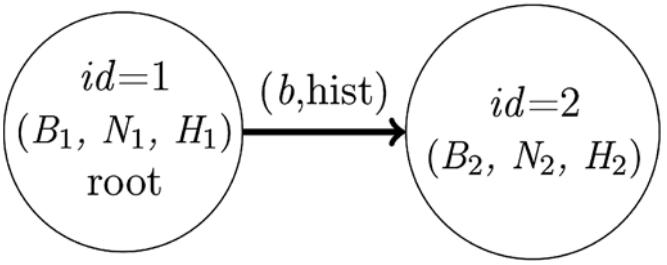

Рис. 5. Добавление в граф новой вершины

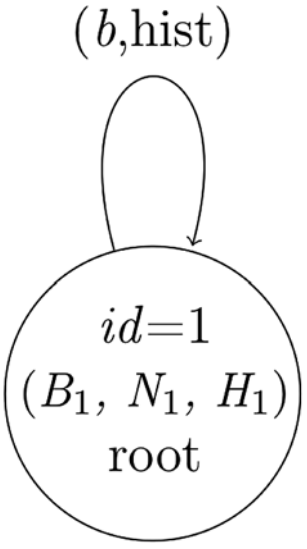

Рис. 6. Петля в графе состояний

этому не будет обсуждаться в настоящем разделе, ему посвящен раздел "История нажатий".

Случай добавления ребра и вершины для первого нажатия ничем не отличается от добавлений вершин и ребер при последующих нажатиях. В общем случае правило добавления вершины и ребра в граф состояний выглядит следующим образом. После нажатия кнопки вычисляются параметры (B, N, H). Далее в графе осуществляется поиск вершины с параметрами (B, N, H). Результату поиска будет соответствовать один из следующих трех пунктов.

- В графе не оказалось вершины с параметрами (B, N, H). В этом случае в граф добавляется вершина (обозначена $id = p$ на рис. 7, *а*) с параметрами (B, N, H). Текущая вершина соединяется ребром с добавленной вершиной. Ребру приписывается нажатая кнопка.
- В графе нашлась вершина A (обозначена $id = p$ на рис. 7, *б*) с параметрами (B, N, H), но не существует ребра из текущей вершины в вершину A. В этом случае добавляется ребро из текущей вершины в вершину A. Ребру приписывается нажатая кнопка.
- В графе нашлась вершина A (обозначена $id = p$ на рис. 7, *в*) с параметрами (B, N, H), причем существует ребро из текущей вершины в вершину A. В этом случае, если ребру уже приписана кнопка, которая была нажата, то никаких действий не предпринимается. Если данному ребру не приписана нажатая кнопка, то кнопка приписывается ребру.

Шаг 5. Для дальнейшего описания алгоритма необходимо дать более подробное описание операции переходов по графу. Для наглядности операция переходов проиллюстрирована на примере рис. 8 (см. третью сторону обложки). Текущая вершина обозначена синим цветом. Следует отметить, что при операциях нажатий кнопок для наглядности имеет смысл представлять веб-браузер с загруженной страницей. Затем необходимо представить, что на странице есть кнопка, а после нажатия этой кнопки страница изменилась, т. е. перешла в новое состояние. Новому состоянию будет соответствовать новая вершина графа и новое ребро, соединяющее старое состояние с новым. Представление о веб-браузере оправдано, поскольку реализация алгоритма основана на веб-браузере Firefox. Если алгоритму требуется перейти из вершины $id = 2$ в вершину $id = 1$, то сначала будет осуществлен переход по ребру (2, 3) посредством нажатия кнопки b_2. Затем будет осуществлен переход по ребру (3, 1), посредством нажатия кнопки b_3. Находясь в вершине $id = 1$, алгоритм может перейти вновь в вершину $id = 2$ посредством нажатия кнопки b_1 или b_4.

Особым случаем операции перехода является переход в корневую вершину графа с помощью перезагрузки страницы. Находясь в любой вершине, можно сделать перезагрузку (обновление) страницы в веб-браузере. В результате перезагрузки страницы алгоритм окажется в начальном состоянии. Следует отметить, что после обновления страницы и вычисления параметров (B, N, H), данные параметры могут не совпадать с параметрами, которые записаны в корневой вершине. Описанная ситуация будет обсуждена в разделе "История нажатий".

Шаг 6. Выбор кнопки для нажатия. Согласно алгоритму, выбор кнопки для нажатия осуществляется с помощью схемы, изображенной на рис. 9. Пусть в результате работы алгоритма был построен некоторый граф, и вершина V этого графа является текущей. Пусть B' представляет множество ненажатых кнопок в данной вершине. Рассмотрим самый верхний блок на рис. 9 и будем двигаться от него вниз. Если в текущей вершине множество B' не пусто, тогда из B' выбирается произвольная кнопка (переход по левой стрелке). Если в текущей вершине множество B' пусто (переход по правой стрелке), то осуществляется поиск пути из текущей вершины до любой другой, в которой B' не пусто. Если такой путь существует, то осуществляется переход по этому пути и выбирается кнопка из множества ненажатых кнопок новой вершины. Если из текущей вершины не существует пути до вершины, в которой есть хотя бы одна ненажатая кнопка, и текущая вершина является корневой вершиной, то процесс обхода считается завершенным. Если текущая вершина не является корневой, то осуществляется переход в корневую вершину и операции, изложенные в данном пункте, выполняются для корневой вершины.

Схематично работа алгоритма изображена на рис. 10.

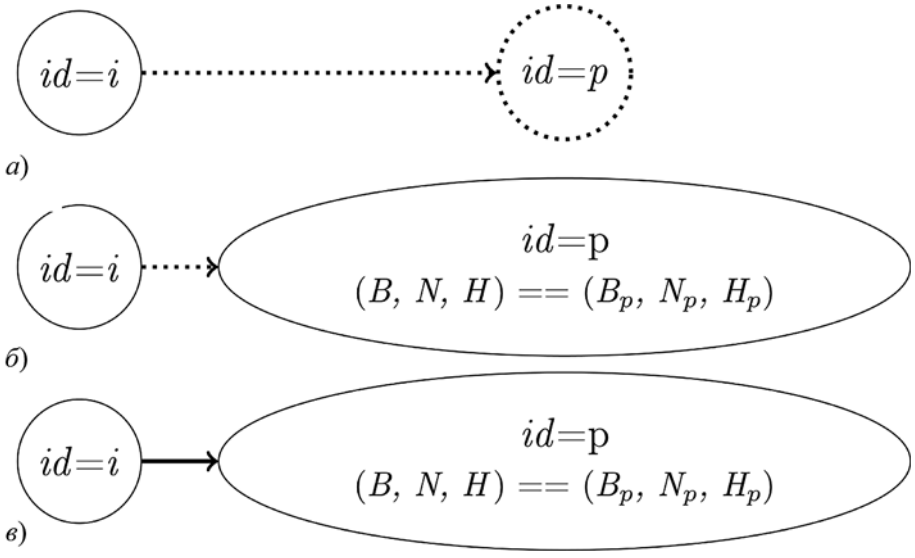

Рис. 7. Три ситуации, которые возникают после нажатия кнопки. Текущая вершина имеет метку $id = i$. Пунктирными линиями обозначены объекты, которые необходимо добавить в граф состояний

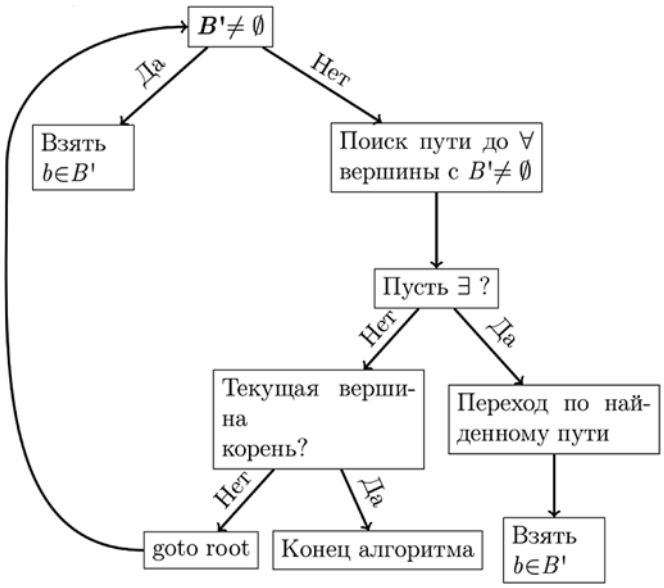

Рис. 9. Схема, описывающая правило выбора кнопки для нажатия

Рис. 10. Схема, описывающая правило выбора кнопки для нажатия

История нажатий

На рис. 11 (см. третью сторону обложки) изображена ситуация, при которой программа, реализующая алгоритм, дважды попала в вершину $id = i$. Находясь в первый раз в вершине $id = i$ и "нажав" кнопку b, программа оказалась в вершине $id = q$. Попав второй раз в вершину $id = i$ и "нажав" кнопку b, программа попала в другую вершину $id = p$. Если в данном случае добавить новое ребро, то получится, что из вершины $id = i$ будут выходить два ребра, которым приписана одна и так же кнопка. В этом случае возникает неоднозначность: если программа в третий раз попадет в вершину $id = i$, то куда будет сделан переход при нажатии кнопки b? Важной частью алгоритма является поиск пути до вершины, в которой есть ненажатая кнопка. В случае описанной неоднозначности становится затруднительно вычислять пути до вершины с ненажатой кнопкой.

Ситуация, изображенная на рис. 8 (см. третью сторону обложки), могла произойти в результате следующих трех причин.

1. Первая причина: на результат нажатия повлияло нажатие предыдущих кнопок. На рис. 8 (см. третью сторону обложки) переходу по ребру (i, q) предшествовала история нажатий кнопок [..., 4, 2], а переходу по ребру (i, p) — [..., 5, 2].

2. Вторая причина может заключаться в том, что веб-страница могла самопроизвольно измениться, например, мог встроиться новый URL. Причем в результате нажатия кнопки были проведены одни и те же действия, однако вследствие самопроизвольного изменения страницы, для алгоритма страница выглядит иначе с точки зрения параметров B, N, H. Под самопроизвольным изменением страницы имеется ввиду следующее. Изменения в веб-странице могут происходить в результате того, что программа "нажала" кнопку. Начиная с момента нажатия кнопки и до момента завершения ее нажатия считается, что страница меняется не самопроизвольно. Если страница изменяется вне указанного отрезка времени, то изменение страницы считается самопроизвольным.

3. Третья причина может заключаться в том, что со временем обработчик событий на кнопке начал работать по-другому. Такую функцию мог реализовать разработчик веб-сайта. Например, спустя 10 с после загрузки страницы функция начинает работать иначе. Пример самопроизвольного изменения изображен на рис. 12 (см. третью сторону обложки). С самопроизвольными изменениями состояния страницы можно бороться с помощью интерфейса MutationObserver. Этот интерфейс предписывается стандартом (https://dom.spec.whatwg.org/#mutationobserver, https://www.w3.org/TR/dom/#mutationobserver) и реализован в Firefox. Он позволяет наблюдать за любыми изменениями DOM-дерева. С помощью MutationObserver можно сделать так, чтобы при изменении DOM-дерева вызывались JavaScript-функции, которые написал разработчик алгоритма. Используя интерфейс MutationObserver, можно реализовать JavaScript-функцию, которая будет вызываться при изменения DOM-дерева. Данная функция должна пометить те узлы дерева, которые изменились самопроизвольно. В качестве меток, которые символизируют о том, что узел DOM-дерева самопроизвольно изменился, можно использовать атрибуты узлов в DOM-дереве. Если узел самопроизвольно изменился, то данному узлу добавляется заранее известный атрибут. Помеченные узлы должны игнорироваться при вычислении параметров B, N, H.

Опишем пример зацикливания алгоритма. Пусть будет иметь место следующее стечение обстоятельств. Пусть отмеченный на рис. 12 (см. третью сторону обложки) блок при каждом своем изменении будет содержать в себе ранее не встреченный на данной странице URL. Пусть страница содержит в себе кнопку, нажатие на которую не осуществляет каких-либо модификаций страницы. Как следствие, нажатие этой кнопки должно давать петлю в графе. Пусть блок изменяется самопроизвольно после нажатия кнопки, но до вычисления параметров B, N, H.

Последовательность действий, приводящая к зацикливанию, выглядит следующим образом:

1) нажатие кнопки b;
2) самопроизвольное изменение;
3) вычисление B, N, H;
4) нажатие кнопки b;
5) самопроизвольное изменение;
6) вычисление B, N, H;
7) ...

Граф состояний, соответствующий излагаемому примеру, отображен на рис. 13. Этот граф состояний представляет цепочку вершин, где вершина i связана выходящим ребром только с вершиной $i + 1$. Таким образом, получаем зацикливание.

Описанные выше проблемные вопросы удается разрешить, если использовать интерфейс MutationObserver. В случае использования интерфейса, после первого изменения блок будет игнорироваться. С учетом интерфейса MutationObserver можно считать, что вторая из представленных выше причин зацикливания не может привести к ситуации, изображенной на рис. 11. С первой и третьей причинами предлагается бороться следующим образом. Когда реализация алгоритма впервые оказывается в вершине $id = i$, после чего нажата кнопка b и реализация оказывается в вершине $id = q$, то ребру приписывается история длины ноль. Когда реализация алгоритма вновь попадает в вершину $id = i$, вновь нажимает кнопку b и оказывается в вершине $id = p$, то на ребрах удлиняется история. Пусть полная история нажатий кнопок имеет вид [..., 4, 2, b,..., 5, 2, b]. Если взять историю длины 1, то метки ребер будут одинаковыми и будут иметь значения (b, [2]). Если взять историю длины 2, то метки на ребрах становятся различными. Ребру (i, p) будет приписана метка (b, [5, 2]), а ребру (i, q) метка (b, [4, 2]).

Удлинение истории основывается на том, что результат нажатия кнопки определяется предыдущими нажатиями. Однако это предположение может быть неверно, поскольку могла измениться работа обработчика событий. Для этого предлагается ввести ограничение на максимальную длину удлинения истории. Пусть максимальная длина истории может равняться L, пусть была встречена неоднозначность, подобная описанной на рис. 11 (см. третью сторону обложки). Пусть процесс удлинения истории каждого ребра дошел до состояния, когда история каждого ребра имеет длину L и истории совпадают. В этом случае метка с кнопкой b должна быть удалена с того ребра, по которому не был сделан последний переход. В случае если $L = 1$, то в примере, представленном на рис. 11 (см. третью сторону обложки), должна быть удалена метка с ребра (i, q). Если на ребре не осталось других меток, то ребро должно быть удалено.

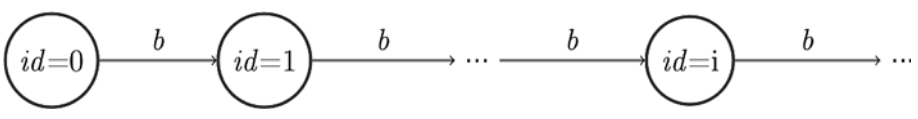

Рис. 13. Пример зацикливания

Использование историй усложняет операцию поиска пути до вершины. Рассмотрим операцию поиска пути. Пусть текущей вершиной в графе состояний является P_0. Пусть от начала работы программы и до момента попадания в вершину P_0 были нажаты кнопки b_{-N},..., b_0. Рассмотрим последовательность кнопок b_1,..., b_n. После нажатия каждой b_s текущей вершиной становится некоторая вершина P_s. Пусть из вершины P_n выходит ребро d, которому приписана история [q_1,..., q_m]. Если найдется такое j, для которого $b_{n-j}! = q_{m-j}$, то считается, что переход по ребру d невозможен и, как следствие, последовательность кнопок b_1,..., b_{n+1} не должна рассматриваться как путь для перехода в вершину с ненажатой кнопкой.

Следует также заметить, что при переходе к вершине с ненажатой кнопкой по некоторому пути очередной переход может привести в неожиданную (не предполагаемую) вершину. В этом случае, если необходимо, добавляется новое ребро и выполняется операция разрешения конфликтов с использованием удлинения истории. После этого необходимо заново запустить операцию поиска пути до вершины с ненажатой кнопкой (см. шаг 6. алгоритма).

Подробнее следует рассмотреть операции перехода в корень (перезагрузки страницы). При переходе в корень возможна одна из обозначенных далее трех ситуаций. Пусть страница была перезагружена и были вычислены значения B, N, H для страницы после перезагрузки. Пусть (B_{root}, N_{root}, H_{root}) представляют параметры корневой вершины.

1. $(B, N, H) = (B_{root}, N_{root}, H_{root})$. Новые параметры корня совпадают со старыми параметрами корня. В этом случае корень назначается текущей вершиной. Никаких изменений с перестройкой графа не осуществляется.

2. $(B, N, H) \neq (B_{root}, N_{root}, H_{root})$. При этом среди всех вершин графа не существует вершины, которой приписаны параметры (B, N, H). В этом случае старые параметры корня заменяются новыми параметрами.

3. $(B, N, H) \neq (B_{root}, N_{root}, H_{root})$, при этом нашлась вершина A, для которой $(B, N, H) = (B_A, N_A, H_A)$. В данном случае корневую вершину *root* необходимо удалить. После удаления корневой вершиной назначается вершина A. При этом необходимо осуществить перенос ребер со старого корня на новый. При переносе ребер могут возникнуть конфликтные ситуации.

Пусть существует вершина X и ребра (*root*, X) и (A, X). Пусть по обоим ребрам переход был сделан по кнопке b. Пусть на ребре (*root*, X) имеются истории h_1,..., h_n для кнопки *b*. Тогда все истории h_i должны быть перенесены на ребро (A, X). После переноса всех ребер с общей концевой вершиной переносятся ребра, не имеющие общих концевых вершин. Такие ребра переносятся по следующему правилу: исходная вершина *root* (*root*, Y) заменяется вершиной A. Входящие ребра также необходимо перенести. Для ребер с общей начальной вершиной (Z, *root*), (Z, A) необходимо перенести все истории с ребра (Z, *root*) на (Z, A). После переноса меток ребер, ребра (Z, *root*) удаляются. В случае с вершинами Y, такими что ребро (Y, *root*) существует, но ребра (Y, A) не существует, для всех ребер (Y, *root*) концевая вершина *Root* заменяется вершиной A.

Если при перезагрузке страницы пришлось обновлять параметры B, N, H в корне, то вероятно, у вершин, соседних с корнем, также придется обновить параметры. Если осуществлять обновление параметров только в корне, то может возникнуть ситуация, изображенная на рис. 14. Формально, вершины x^0 и x^1, $x \in [1—3]$ являются разными, поскольку у них различаются тройки параметров B, N, H. Однако до каждой из вершин x^0 и x^1, можно добраться из корневой вершины с помощью нажатия одной и той же последовательности кнопок.

Пример, представленный на рис. 14, иллюстрирует ситуацию, которая приводит к зацикливанию алгоритма. Предположим, что $b = b1 = b2$, а также в вершинах типа 2^n существует кнопка d (рис. 15).

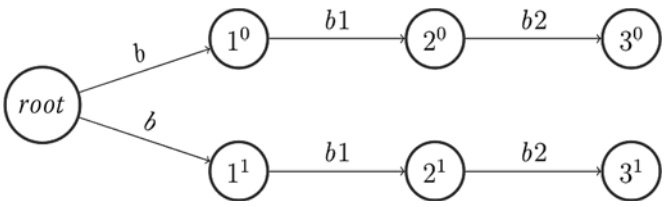

Рис. 14. Пример ситуации, при которой была выполнена операция обновления страницы, после чего последовало обновление параметров *B*, *N*, *H* в корне, но при этом не было обновления параметров в других вершинах графа

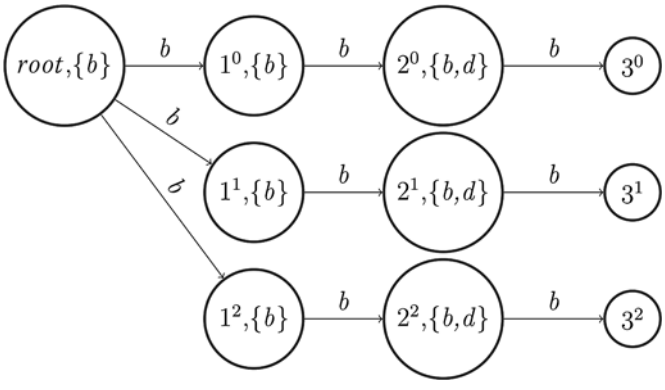

Рис. 15. Пример зацикливания алгоритма

Пусть программа, реализующая алгоритм, впервые загрузила страницу и "нажала" кнопку b три раза. В результате данной операции текущей вершиной окажется 3^0. В вершине 2^0 содержится кнопка d, которую программа должна "нажать". Для осуществления нажатия будет выполнена перезагрузка страницы, в результате чего текущей вершиной окажется корневая вершина. Предположим, что параметры корня и других вершин изменились. Программа, согласно алгоритму, проведет обновление параметров в вершине *root*, после чего попытается попасть в вершину 2^0. Однако при первом нажатии кнопки b программа попадает не в вершину 1^0, а в вершину 1^1. Произошел переход в непредполагаемую вершину. В этом случае запустится операция поиска вершины, в которой имеется ненажатая кнопка. В вершине 1^1 имеется кнопка b. Согласно алгоритму, будет проведено нажатие найденной кнопки. После первого нажатия в текущей вершине будет обнаружена еще одна кнопка b. Будет проведено еще одно нажатие кнопки b. В результате двух нажатий текущей вершиной окажется вершина 3^1, однако в вершине 3^1 отсутствуют кнопки. Как следствие, будет запущена операция поиска пути до ненажатой кнопки. В результате этой операции будет найден путь $root \to 1^1 \to 2^1$. Обновив страницу, программа попытается перейти по данному пути и вновь первый переход приведет в непредполагаемую вершину (не в 1^1, а в 1^2).

С учетом представленного примера (см. рис. 14, 15) предлагается следующее правило обновления параметров B, N, H в вершинах, соседних с корнем. Если при переходе текущей вершиной оказывается не ожидаемая вершина X, а вершина X', но при этом для перехода в вершину X программе потребовалось "нажать" последовательность кнопок B = [JUMP_ROOT, $b1$,..., bn] и, чтобы попасть в X', было выполнено нажатие такой же последовательности кнопок, то в этом случае предлагается правило: вместо добавления новой вершины и ребра выполняется обновление параметров B, N, H в вершине X, при этом текущей вершиной назначается вершина X.

Представленное выше соображение об обновлении параметров B, N, H в соседних с корнем вершинах исключает один из шаблонов зацикливания. Однако его использование не исключает всего набора ситуаций зацикливания. В связи с этим обстоятельством при реализации алгоритма на практике рекомендуется установить ограничение на максимальное число нажатий кнопок на странице.

Эвристика выбора кнопки

Пусть программа, реализующая алгоритм, находится в некоторой вершине графа и в этой вершине имеется множество ненажатых кнопок B'. Какую из кнопок следует выбирать так, чтобы обход страницы занял наименьшее время?

На рис. 2 (см. вторую сторону обложки) представлена кнопка "стрелка влево". Пусть программа N раз "нажала" кнопку "стрелка влево". Если алгоритм на-

жмет кнопку вне блока "Архив новостей", то данный блок исчезнет со страницы. В случае пропавшего блока, согласно алгоритму, программа в дальнейшем должна будет вернуться в то место, где кнопка "стрелка влево" была нажата N раз, и "нажать" ее еще один раз. В описанной ситуации появляется N лишних нажатий.

В связи с представленным примером предлагается, чтобы в программе сохранялся текст последней нажатой кнопки, и, при выборе очередной кнопки для нажатия среди кнопок множества B', будет выполняться поиск той, текст которой совпадает с сохраненным текстом. Если такая кнопка нашлась, то она выбирается для нажатия. Если такой кнопки не нашлось, то кнопки множества B' упорядочиваются по возрастанию редакторского расстояния до текста последней нажатой кнопки и выбирается кнопка с наименьшим расстоянием.

Разрастающиеся списки

На практике встречаются кнопки, которые можно нажать десятки тысяч раз. При этом, в результате очередного нажатия в веб-страницу будут встроены новые данные, а старые данные не будут удалены. Как следствие, веб-страница увеличивается с каждым нажатием кнопки, в результате чего веб-браузер начинает работать медленно. В этом случае не исключена ситуация, при которой закончится оперативная память. Пример такой кнопки приведен на рис. 1 (см. вторую сторону обложки). Нажатие этой кнопки встраивает в веб-страницу примерно 20 ссылок на публикации. Под ссылкой понимается гиперссылка на публикацию, картинка или какой-либо другой текст. При очередном нажатии старые ссылки не пропадают. Отмеченную кнопку можно нажать более 70 000 раз. Если не очищать веб-страницу, то после 70 000 нажатий в веб-странице окажется 1 400 000 ссылок. Такого количества будет достаточно, чтобы браузер начал работать медленно.

Для преодоления описанного затруднения необходимо в автоматическом режиме выделять части веб-страницы, которые можно удалить. В случае с примером относительно веб-страницы, изображенной на рис. 1, идеальным вариантом является усечение растущего списка ссылок. Следует отметить, что удаленный узел дерева может являться одной из кнопок, удалять которые нежелательно. Удаленный узел при этом может не являться кнопкой, однако может содержать в своих атрибутах какие-либо данные, необходимые для корректной работы той или иной кнопки. Удалять такие узлы тоже нежелательно.

Для разрешения вопросов, связанных с разрастающимися списками ссылок, применяется алгоритм поиска таких списков и их усечения. Представленный алгоритм никак не учитывает, является

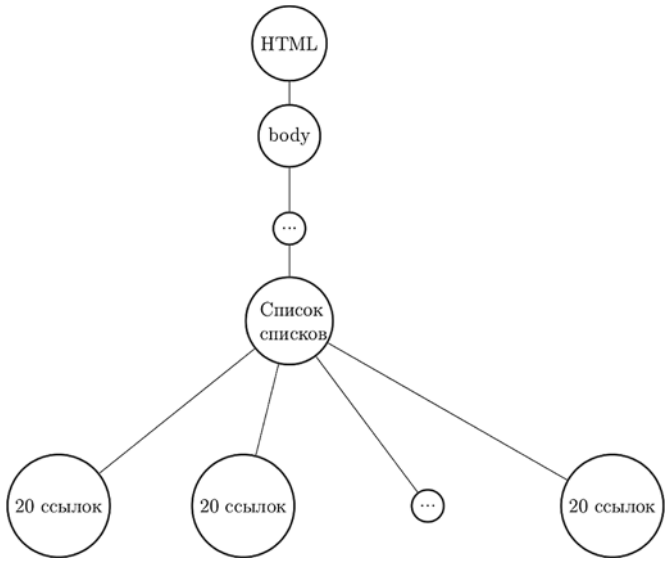

Рис. 16. DOM-дерево, соответствующее разрастающейся веб-странице

ли узел DOM-дерева кнопкой или нет. Однако на практике реализация алгоритма усечения списков, несмотря на свою простоту, не выбрасывает кнопки из дерева и усекает именно то, что необходимо.

Как правило, в результате нажатий кнопок веб-страница (DOM-дерево) разрастается не хаотичным образом, а в страницу (в DOM-дерево) встраиваются повторяющиеся поддеревья. В DOM-дереве, которое соответствует рис. 1 (см. вторую сторону обложки), это вершина U. При нажатии кнопки "загрузить еще" в потомки вершины U будут встраиваться списки из 20 ссылок на публикации. Схематично данная ситуация изображена на рис. 16.

HTML-разметка поддерева "список списков" приведена на рис. 17.

Первый элемент списка был добавлен при загрузке страницы, остальные элементы были добавлены в результате восьмикратного нажатия кнопки. Отличительной чертой списков является то обстоятельство, что у списка существует родитель, а непосредственные потомки родителя похожи друг на

```
<div class="b-list-normal">
  <div class="b-list-normal">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
  <div class="b-list" style="display: block;">20 ссылок</div>
</div>
```

Рис. 17. HTML-разметка поддерева "список списков"

друга. В рассмотренном примере родителем является элемент div с class="b-list-normal".

Пусть имеется функция F, которая получает на вход вершину дерева, а на выходе выдает число непосредственных потомков, которые похожи друг на друга. Применять эту функцию нужно следующим образом. После загрузки страницы данная функция применяется к каждому узлу дерева. Число, которое возвращает данная функция $F(X)$, приписывается узлу дерева X, обозначим это число в каждой вершине как N_0. После каждого нажатия кнопки должна вызываться функция F. Если для какого-то узла T выполняется $\frac{F(T)}{N_0} > \alpha$, то считается, что список увеличился достаточно сильно и его необходимо усекать. В этом случае все потомки T удаляются, однако сам T при этом не удаляется. В качестве константы α можно взять число два.

Определим функцию F. Пусть имеется функция $f(node)$, которая получает на вход вершину дерева и возвращает признаки вершины. Тогда по определению назовем вершину v похожей на своих соседей v_0, v_1, если $f(v) = f(v_0) = f(v_1)$. Если у вершины v не существует левого или правого соседа, то вершина v игнорируется. Пусть вершина T является родителем, $T_0,\ldots, T_n$ — непосредственные потомки T, а $F(T) = \sum_{i=1}^{n-1}\left[f(T_i) == f(T_{i-1}) \text{ and } f(T_i) == f(T_{i+1})\right]$.

Скобка $[f(x)]$ обозначает операцию, результатом которой является 1, если значение предиката внутри скобки *True*, и возвращает 0 в противном случае.

В качестве функции f можно взять функцию $f(node)$ = [имя_тега_node, имя_класса_node].

На этом завершается описание алгоритма, который предназначается для обхода веб-страницы. В результате обхода множества веб-страниц сформирована коллекция ссылок (URL) на публикации. Следующий раздел посвящен следующей задаче. Имеется URL, для которого известно, что в содержании веб-страницы содержится публикация, требуется отделить публикацию от всего остального, после чего она будет добавлена в БД.

Извлечение публикации

Сервис мониторинга публикаций предоставляет возможность выделения публикации следующими двумя способами: readability и "ручной".

Операция readabilty применяется к HTML-документам, в которых содержится статья. При выполнении данной операции из документа удаляется реклама, кнопки и меню на веб-сайте. Операция применяется в веб-браузерах и именуется "режим чтения". После удаления нежелательных элементов в документе должна остаться только статья.

Существует ряд реализаций readability. Известной реализацией является ARC90 [2]. На ее основе реализован режим чтения в Firefox [3] и в веб-браузере Safari. Реализация ARC90 и построенные на ее основе средства обрабатывают каждый HTML-документ независимо от других HTML-документов.

К недостаткам имеющихся на настоящее время реализаций можно отнести тот факт, что согласно алгоритму может быть допущена ошибка, в результате чего будет извлечена не статья. Например, если статья имеет небольшой размер, порядка двух предложений, то алгоритм может вместо статьи извлечь, например, информацию о копирайте, поскольку текст с копирайтом по объему больше, чем текст статьи. Другим недостатком является то обстоятельство, что угадав положение статьи верно, вместе со статьей, согласно алгоритму, может быть извлечен текст из меню веб-сайта и текст из кнопок.

К преимуществам readability можно отнести тот факт, что данный подход не требует настройки.

Другой способ выделения публикации — "ручной" — выделяет публикацию точно, не захватывая лишних частей веб-страницы. Однако данный метод требует ручной настройки. Обычно веб-страницы с публикациями обладают одним и тем же форматом. С точки зрения DOM-дерева фраза "обладают одним и тем же форматом" формулируется следующим образом. Пусть веб-страница содержит публикацию. Тогда существует вершина в DOM-дереве, в которой располагается статья. При этом путь от корня DOM-дерева до вершины с публикацией будет одинаков для всех веб-страниц. Как правило, на практике для одного веб-сайта встречается несколько типов веб-страниц, содержащих публикацию. Каждый тип обладает своим путем от корня до вершины с публикацией. На практике встречается от одного до четырех типов страниц.

Согласно данному подходу предлагается, чтобы программист потратил несколько минут на визуальный осмотр веб-сайта. После этого он должен выделить по одному представителю из разных типов веб-страниц и описать все пути от корня до вершины со статьей на языке XPATH. Пример пути от корня до вершины с публикацией на языке XPATH: //div[contains(@class,'b-inline-topics-box')]. Описанная процедура не является долгой и занимает порядка десяти минут.

С помощью алгоритма обхода веб-страницы и методов выделения публикации из веб-страницы было извлечено более трех миллионов публикаций с различных веб-сайтов. С помощью программного средства, описанного в работе [1], был осуществлен анализ публикаций на предмет эмоциональной окраски.

Программная реализация алгоритма и результаты тестовых испытаний

Функции сбора URL и выделения публикации из веб-страницы реализованы в рамках одной компьютерной программы contfetcher. Эта программа реализована на языке Python.

Программа contfetcher применяется для обработки одного веб-сайта. Для обработки нескольких веб-

сайтов необходимо запускать несколько экземпляров программы contfetcher. Для каждого веб-сайта в БД создаются свои экземпляры таблиц.

Программа contfetcher является многопроцессной. Существует главный процесс, который соединен коммуникационным каналом с СУБД и с процессами типа Worker. Главный процесс отвечает за запуск необходимого числа процессов типа Worker и за распределение заданий между процессами типа Worker. Под заданиями понимается, например, скачивание веб-страницы, которая соответствует заданному URL, и извлечение публикации из HTML-кода веб-страницы или обход веб-страницы, при котором будут извлекаться и сохраняться все встреченные URL. Если один из процессов типа Worker аварийно завершается, то главный процесс перезапустит процесс типа Worker. Визуальное представление компонентов программы дано на рис. 18.

В рамках программы contfetcher запускаются приложения. Каждое приложение можно включить и выключить. Для каждого включенного приложения при старте программы будут создаваться процессы типа Worker и при работе программы каждому процессу типа Worker будут высылаться задания для обработки. Если приложение выключено, то процессы типа Worker для данного приложения запускаться не будут.

В программе contfetcher существует три стандартных приложения.

1. Приложение traverse — обход веб-страницы с исполнением JavaScript. Данное приложение применяется для извлечения URL с веб-страницы, его имеет смысл применять на страницах, где для получения URL необходимо интерактивно взаимодействовать с веб-страницей. Примером такой страницы является https://ria.ru/lenta/. Для получения наиболее полной коллекции URL на этой странице необходимо нажимать кнопку "Загрузить еще". В рамках данного приложения веб-страница загружается с помощью веб-браузера Firefox. Следует отметить, что при использовании Firefox будет осуществляться ожидание загрузки всех CSS и JavaScript-файлов, в результате чего время загрузки страницы может занимать время, равное нескольким секундам. Для сравнения, при скачивании только HTML-кода веб-страницы, как правило, требуется время, равное десятым долям секунды.

2. Приложение readability — в процессе работы данного приложения из БД будут извлекаться URL, после чего для каждого URL будет осуществляться загрузка HTML-кода. Из HTML-кода в автоматическом режиме будет осуществляться извлечение публикации, название публикации, автор публикации, время публикации. Тело извлеченной статьи будет сохранено на жесткий диск. В БД будет сохранено название статьи, имя автора публикации, время публикации, путь до тела статьи на жестком диске и URL, из которого была извлечена публикация. Достоинством данного похода является то, что от пользователя не требуется никаких усилий для выделения статьи из HTML. К недостаткам можно отнести следующие:

- публикация может выделиться неточно, а именно — вместе с телом публикации могут быть захвачены элементы антуража веб-страницы (кнопки, меню и пр.);
- публикация может не выделиться, даже при условии, что на веб-странице она есть;
- публикация может быть выделена там, где ее нет, т. е. будет выделен "мусор".

Операция readability осуществляется с помощью веб-браузера Firefox. В Firefox данная операция называется режимом чтения.

3. Приложение grabber — данное приложение выполняет те же функции, что и readability, за исключением того, что правила для извлечения публикации описывает программист. Правила для извлечения

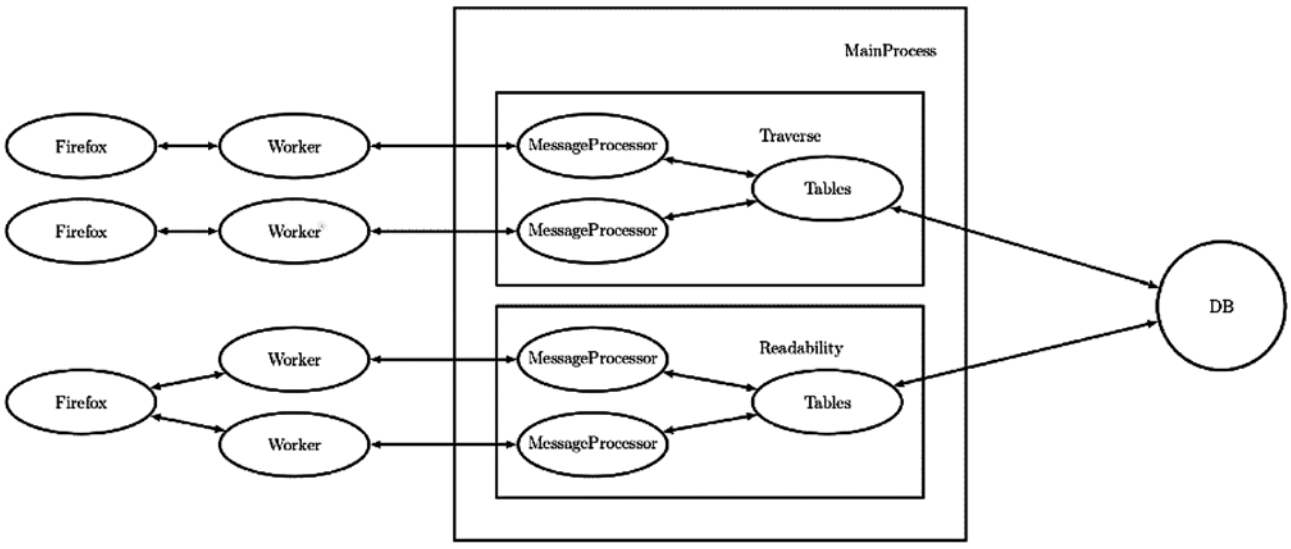

Рис. 18. Визуальное представление компонентов программы contfetcher

пишут на языке запросов к XML-документам — XPATH. На практике такие правила занимают несколько строк кода в конфигурационном файле. Использование данного похода предпочтительнее, чем readability, поскольку публикация будет извлекаться точно.

Кроме использования стандартных приложений для программы contfetcher существует возможность написания пользовательских приложений. При необходимости, функционал пользовательского приложения могут наследовать функции другого приложения.

Каждое приложение состоит из трех классов: MessageProcessor; Tables; Worker. Все три класса реализуются на языке Python. Перечисленные классы имеют следующее назначение.

1. Worker — данный класс запускается в рамках процесса типа Worker. Данный класс принимает задания на обработку от главного процесса. В процессе выполнения задания осуществляется загрузка вебстраницы и все необходимые операции, связанные с обработкой HTML или других данных, полученных от удаленного сервера. Результатом обработки задания является либо сообщение с полезными данными, либо сообщение с кодом ошибки.

2. Tables — в рамках данного класса должен быть написан код (SQL), с помощью которого будут созданы таблицы в БД. Через данный класс осуществляется также вставка данных в БД.

3. MessageProcessor — данный класс используется ядром программы contfetcher для отправления заданий в процессы типа Worker (в класс Worker) и для приема сообщений от процессов типа Worker. Данный класс должен вызывать методы класса Tables для вставки данных в БД.

На настоящее время сервис осуществляет мониторинг публикаций, в которых содержится название хотя бы одной из следующих организаций:

- МГУ имени М. В. Ломоносова;
- Институт динамики систем и теории управления имени В. М. Матросова СО РАН;
- Институт нефтехимического синтеза им. А. В. Топчиева Российской академии наук;
- Институт прикладной математики им. М. В. Келдыша;
- Институт цитологии РАН;

Упоминания научных организаций в СМИ

Организация	Число упоминаний			
	Отнесенных к положительному классу	Отнесенных к отрицательному классу	Отнесенных к нейтральному классу	Неоднозначных
МГУ имени М. В. Ломоносова	1959 (77,5)*	185 (7,3)	376 (14,8)	6 (0,2)
Институт динамики систем и теории управления имени В. М. Матросова СО РАН	1 (100,0)	0 (0)	0 (0)	0 (0)
Институт прикладной математики им. М. В. Келдыша	2 (100,0)	0 (0)	0 (0)	0 (0)
Институт цитологии РАН	3 (100,0)	0 (0)	0 (0)	0 (0)
Институт нефтехимического синтеза им. А. В. Топчиева Российской академии наук	2 (100,0)	0 (0)	0 (0)	0 (0)
Московский педагогический государственный университет	84 (88,4)	0 (0)	11 (11,5)	0 (0)
Российский государственный гуманитарный университет	180 (87,8)	0 (0)	25 (12,1)	0 (0)
ФГБНУ «Почвенный институт им. В. В. Докучаева»	0 (0)	0 (0)	0 (0)	0 (0)
ФГБНУ Институт машиноведения им. А. А. Благонравова РАН	1 (100,0)	0 (0)	0 (0)	0 (0)
ФГБНУ Научный центр неврологии	7 (100,0)	0 (0)	0 (0)	0 (0)
ФГБУН Физико-технический институт УрО РАН	0 (0)	0 (0)	0 (0)	0 (0)
Воронежский государственный университет	47 (87)	2 (3,7)	5 (9,2)	0 (0)
Полярный геофизический институт	0 (0)	0 (0)	0 (0)	0 (0)
Институт географии РАН	13 (81,25)	0 (0)	2 (12,5)	1 (6,25)
*В скобках указан процент упоминаний данного класса от общего числа упоминаний				

- Московский педагогический государственный университет;
- Российский государственный гуманитарный университет;
- ФГБНУ "Почвенный институт им. В. В. Докучаева";
- ФГБНУ Институт машиноведения им. А. А. Благонравова РАН;
- ФГБНУ Научный центр неврологии;
- ФГБУН Физико-технический институт УрО РАН;
- Воронежский государственный университет;
- Полярный геофизический институт;
- Институт географии РАН.

Сервис предоставляет возможность сформировать коллекцию публикаций, в которых встречается упоминание организации из приведенного списка, после чего провести автоматическую классификацию на три класса: "положительный", "отрицательный", "нейтральный". Если публикация относится к классу "положительный", то это означается, что она положительным образом влияет на формирование репутации организации. Отнесение к классу "отрицательный" означает, что публикация негативным образом сказывается на формирование репутации организации. Отнесение публикации к классу "нейтральный" означает, что она не влияет на формирование репутации.

В таблице приведены результаты мониторинга публикаций за 2016 г. Разъяснения требует столбец "неоднозначные". Это означает, что публикация влияет как положительно, так и отрицательно на формирование репутации. Метка "неоднозначно" не приписывается публикации при автоматической классификации публикаций. Данная метка существует, если эксперт вручную указал, что публикация улучшает и очерняет репутацию организации одновременно.

Заключение

В статье описан алгоритм, с помощью которого возможно реализовать обход веб-страницы путем нажатия кнопок, попадая при этом во всевозможные места веб-страницы. Изложенный алгоритм был реализован авторами и применяется в ИАС ИСТИНА для мониторинга веб-сайтов СМИ с целью извлечения публикаций.

Список литературы

1. **Васенин В. А., Роганов В. А., Дзабраев М. Д.** Методы автоматизированного анализа тональности текстов в средствах массовой информации // Программная инженерия. 2016. Т. 7, № 8. С. 360—372.
2. **ARC90,** readability. URL: http://code.google.com/p/arc90labs-readability.
3. **Mozilla** Firefox readability. URL: https://github.com/mozilla/readability

Methods and Means for Monitoring Publications in Mass Media

V. A. Vasenin, vasenin@msu.ru, Moscow State University, Moscow, 119234, Russian Federation,
M. D. Dzabraev, dzabraew@gmail.com, ISTINA Information System, Moscow, 119192, Russian Federation

Corresponding author:

Vasenin Valery A., Professor, Moscow State University, Moscow, 119192, Russian Federation,
E-mail: vasenin@msu.ru

Received on August 21, 2017
Accepted on September 07, 2017

Currently various applications encounter a task of extracting various data from the web sites. The first step to solve this task is extraction of all URLs from the web site being analyzed. Modern web sites are usually interactive, where interactivity means that the site can listen to events generated by user and respond to them. The most important event is clicking the left mouse button. To obtain the most complete collection of URLs one should click a certain sequence of buttons on the web page, which may cause a new block containing new URLs to be dynamically inserted. In other words, to obtain the most complete collection of URLs one should develop an algorithm that emulates actions of a user. This article presents model views, algorithms and software that emulates mouse clicks by a user. It also presents a business process model for algorithms and software using which one may automatically navigate within a page. Web pages may contain buttons of two types: clicking on the button either opens a new page, or modifies the current page by evaluating JavaScript. The navigating algorithm ignores the first type and only deals with the second type of buttons. The algorithm presented in this articles is intended to work with the following assumptions.

The implementation should automatically detect the buttons of second type on the page and automatically choose a sequence of buttons to be clicked on. The algorithm takes into account that as a consequence of clicking old buttons may disappear and new buttons may appear. If some button was present and then disappeared without being clicked, the memory of the implementation will contain a path using which the implementation will later come into state when this button was present and click it.

Keywords: *data extraction, web, readability, web-site traverse, web-page traverse, Javascript, Firefox*

For citation:

Vasenin V. A., Dzabraev M. D. Methods and Means for Monitoring Publications in Mass Media, *Programmnaya Ingeneria,* 2017, vol. 8, no.11, pp. 490—503.

DOI: 10.17587/prin.8.490-503

References

1. **Vasenin V. A., Roganov V. A., Dzabraev M. D.** Metodi avtomatizirovannogo analiza tonalnosti tekstov v sredstvah massovoi informacii (Methods of Automated Sentiment Analysis of Texts Published by Mass Media), *Programmnaya Ingeneria*, 2016, vol. 7, no. 8, pp. 360—372 (in Russian).

2. **ARC90** readability, available at: http://code.google.com/p/arc90labs-readability

3. **Mozilla** Firefox readability, available at: https://github.com/mozilla/readability