ПРОГРАММИРОВАНИЕ, 2019, № 6, с. 55–66

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.413

ИНТЕРАКТИВНЫЙ ПОИСК НЕТОЧНЫХ ПОВТОРОВ В ДОКУМЕНТАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

© 2019 г. Д. В. Луцив[a,*], Д. В. Кознов[a,], А. А. Шелиховский[a,***], К. Ю. Романовский[a,****], Г. А. Чернышев[a,*****], А. Н. Терехов[a,******], Д. А. Григорьев[a,*******], А. Н. Смирнова[a,********], Д. В. Боровков[a,*********], А. И. Васенина[a,**********], Е. Н. Шеметова[a,**********]**

[a] Санкт-Петербургский государственный университет
199034 Санкт-Петербург, Университетская набережная, д. 7–9, Россия
**E-mail: d.lutsiv@spbu.ru*
***E-mail: d.koznov@spbu.ru*
****E-mail: tshel231@gmail.com*
*****E-mail: k.romanovsky@spbu.ru*
******E-mail: g.chernyshevspbu.ru*
*******E-mail: a.terekhov@spbu.ru*
********E-mail: d.a.grigoriev@spbu.ru*
*********E-mail: anna.en.smirnova@gmail.com*
**********E-mail: danila_yumsh@mail.ru*
**********E-mail: saibrog@yandex.ru*
***********E-mail: katyacyfra@gmail.com*

Поступила в редакцию 28.06.2018 г.
После доработки 05.09.2018 г.
Принята к публикации 05.09.2018 г.

Различные элементы ПО – классы, методы, требования, тесты и пр. – часто имеют сходную функциональность, что, в частности, влечет наличие повторов в документации, которая описывает эти элементы. Но неконтролируемые повторы, создаваемые с помощью приема copy/paste, затрудняют сопровождение и поддержку документации. Поэтому задача поиска повторов в уже существующей документации ПО оказывается важной, ее решение позволяет применять плановое повторное использование (reuse), создавать и использовать шаблоны для унификации и автоматической генерации документации. В данной статье мы представляем интерактивную методику поиска неточных повторов, позволяющую привлечь пользователя для реализации семантически-значимого поиска. Методика включает в себя новое формальное определение неточных повторов, алгоритм поиска по образцу и доказательство полноты алгоритма. Также представлены результаты экспериментов на наборе документов из промышленных проектов.

DOI: 10.1134/S0132347419060049

1. ВВЕДЕНИЕ

О проблемах с качеством документации ПО писали еще в 70-е годы [1] и продолжают писать в настоящее время [2]. Вместе с тем документация, как и ПО, становится все сложнее, требует все больше ресурсов для разработки и сопровождения.

При создании и модификации документов общепринятым является использование приема copy/paste: один и тот же фрагмент текста многократно копируется и потом модифицируется и дополняется. Этот прием опирается на тот факт, что многие элементы ПО (software features), описываемые в документации, повторно используют одну и ту же функциональность – это справедливо для требований, элементов пользовательского интерфейса, а также для программного кода, тестов и т.д. Но при отсутствии специальных средств раскопированные фрагменты текста создают дополнительные трудности при сопровождении документации, поскольку требуют трудоемкой синхронизации при изменении соответствующих свойств ПО. И если повторному использованию

ПО посвящены многочисленные исследования, часть из которых воплотилась в промышленных технологиях (см. обзоры [3–5]), то вопрос повторного использования документации ПО остается, главным образом, лишь предметом академических исследований [6–10]. Следует также отметить, что остро стоит вопрос унификации документации – коль скоро имеется большое количество сходной информации, то целесообразно, чтобы она была представлена одинаково. Таким образом, поиск и исследование повторов являются важной задачей при наладке повторного использования и унификации документации ПО.

Повторы в документации ПО активно исследуются в последнее десятилетие [6, 11–17]. В то же время отсутствуют специальные средства для поиска повторов в документации ПО – обычно с этой целью применяются различные инструменты текстового поиска. Однако такие средства не удается применить для обнаружения неточных повторов – текстовых фрагментов, которые имеют значительную общую часть и некоторые различия. В связи с этим мы создали инструмент Duplicate Finder [18, 19]. В [16] мы представили алгоритм поиска неточных повторов, который ищет такие повторы как комбинации точных. Однако качество выдачи данного алгоритма оказалось низким в силу игнорирования семантики при поиске повторов.

В данной статье мы представляем интерактивную методику поиска неточных повторов. Для привнесения семантического критерия в процесс поиска мы привлекаем пользователя. Кратко, процесс поиска организован так. Сначала для документа, в котором нужно найти неточные повторы, автоматически, с использованием инструмента Clone Miner [20], строится карта точных повторов. Потом, основываясь на этой карте, а также на гипотезе, что скопление точных повторов в некотором месте документа означает высокую вероятность наличия там же неточного повтора, составленного из этих точных, пользователь выбирает фрагмент текста, состоящий из самых часто используемых точных фрагментов, замыкает его до семантически значимого (то есть расширяет его границы, делая его полным описанием некоторого свойства ПО) и использует далее в качестве шаблона для поиска. Результат поиска пользователь также может редактировать – отсеивать ложноположительные срабатывания (false positives) из выдачи алгоритма, а также придавать элементам выдачи семантическую завершенность, делая каждый из них в отдельности больше или меньше посредством расширения/сужения его границ в документе. При этом мы оставляем в стороне точный ответ на вопрос о том, что такое семантически значимый фрагмент текста, полностью возлагая ответственность за это на пользователя.

Статья организована следующим образом. В разделе 2 делается обзор работ, сходных с данным исследованием, в разделе 3 описываются использованные в нашем исследовании подходы, методы и технологии, в разделе 4 излагается интерактивная методика поиска неточных повторов, в разделе 5 дается новое формальное определение неточного повтора, в разделе 6 приводится алгоритм поиска неточных повторов по образцу, а также формулируется критерий полноты поиска по образцу и доказывается полнота предложенного алгоритма. В разделе 7 приведены оценки сложности алгоритма, а в разделе 8 представлены результаты экспериментального исследования.

2. ОБЗОР

Повторы в документации ПО активно исследуются на протяжении последнего десятилетия.

В [6] рассматриваются повторы в API-документации Java-проектов, расширяя JavaDoc средствами повторного использования. В [12] этот подход расширяется неточными повторами. Подобно подходам [8–10], для определения вариативных фрагментов повторов применяется параметризация. Но в работе не затрагивается задача обнаружения неточных повторов, а сами неточные повторы определены неформально.

В [13] исследованы точные повторы во встроенной документации для ряда проектов с открытым исходным кодом, но неточные повторы в этой работе не рассматривались.

В [14] повторы в документации ПО использовались для оценки качества документации, неточные повторы не рассматривались.

В статье [11] исследовались повторы в описаниях требований: было проанализировано 28 промышленных документов, найденные повторы были отфильтрованы и классифицированы вручную. В статье обсуждался вопрос о том, что означают найденные повторы(в частности, были выделены повторы, соответствующие дублированию в коде). Также исследовался вопрос о влиянии избыточности на скорость чтения и понимания текстов. В статье не рассмотриваются другие виды документации программного обеспечения, а также неточные повторы, хотя указывается на их наличие в реальных документах.

В [7] анализируется API-документация нескольких известных открытых проектов, найденные повторы классифицируются. В статье рас-

сматривается вопрос повторного использования документации. Однако неточные повторы в работе не рассматриваются, хотя и указывается на то, что они часто встречаются и важны на практике.

В [21] ищутся неточные повторы в текстовых описаниях вариантов использования (use cases) с использованием методов анализа естественных языков. Однако рассмотрен весьма специфический способ описания требований, редко применяемый на практике. Кроме того, неочевидно, как использовать данный метод для других видов документации ПО.

Следует отметить, что изученные нами существующие подходы, кроме [21], используют токен-ориентированные (token-based) средства анализа клонов в программном коде, что не позволяет решить проблему поиска неточных повторов. Однако отмечается наличие и важность таких повторов при анализе избыточности документации, а также при организации повторного использования [7, 11. 12].

Предложенные в данной работе результаты существенно опираются на поиск по образцу. Данная задача является известной и многократно решалась различными способами в разных контекстах. Остановимся на кратком обзоре этой задачи в рамках текстового поиска.

Алгоритм, описанный в [22], позволяет эффективно искать неточные вхождения образца в текст, но требует достаточно трудоемкого предварительного анализа образца. В [23] описан метод, позволяющий искать неточные вхождения образца в текст с применением средств информационного поиска; следует отметить, что данный метод требует трудоемкого предварительного анализа входных данных (документа и образца). Алгоритм [22] ориентирован на работу с неизменным образцом, а подход [23] оперирует с неизменным текстом документа, но ни то, ни другое не характерно для нашей задачи.

Алгоритмы, представленные в [22, 24–26], являются эффективными, но достаточно сложны, что затрудняет их использование и модификацию, а также доказательство их формальных свойств.

В [25, 26] описаны алгоритмы, которые позволяют искать текстовые фрагменты, для которых расстояние Левенштейна [27] не превосходит заданного. Однако следует отметить высокую сложность вычисления расстояния Левенштейна, что, в силу наших экспериментов, делает неперспективным данный подход для задач поиска повторов в документации ПО. Более полный обзор алгоритмов поиска неточных вхождений текстовых образцов можно найти в [28].

Пользуясь идеями, предложенными в данной области, мы создали собственный алгоритм поиска по образцу, руководствуясь следующими соображениями:

(i) нам требовалось, чтобы алгоритм выполнял поиск в соответствии с нашим определением неточных повторов;

(ii) мы хотели формально доказать ряд свойств алгоритма;

(iii) требовалось, чтобы алгоритм имел приемлемое для нашей задачи быстродействие (с учетом того, что алгоритм должен работать в интерактивном режиме);

(iv) необходимо, чтобы он выдавал минимальное количество ложных срабатываний.

3. ИСПОЛЬЗОВАННЫЕ ПОДХОДЫ, МЕТОДЫ И ТЕХНОЛОГИИ

3.1. Редакционное расстояние

Для определения степени близости (схожести) двух фрагментов текста (текстовых строк) используется *редакционное расстояние* (edit distance) [27] – количество строковых операций, с помощью которых можно получить одну строку из другой: чем меньше операций требуется, тем более похожи друг на друга строки [29]. Различные определения редакционных расстояний отличаются набором допустимых операций. В данной работе мы использовали *редакционное расстояние по наибольшей общей подпоследовательности* (longest common subsequence distance) [30, 31], использующее только операции вставки и удаления символа, по причине хорошего соответствия описанной ниже модели неточных повторов и, следовательно, удобства дальнейших построений и доказательств. В [29] показано, что редакционное расстояние по наибольшей общей подпоследовательности обладает метрическими свойствами. В дальнейшем будем обозначать редакционное расстояние по наибольшей подпоследовательности между двумя строками s_1 и s_2 как $d(s_1, s_2)$.

Вычисление редакционного расстояния является трудоемкой задачей; сложность вычисления выбранного нами алгоритма вычисления расстояния по наибольшей подпоследовательности [32] в среднем случае оценивается как $\mathbb{O}(|s_1| * |s_2|)$, где s_1 и s_2 – соответствующие строки, причем в работе [33] показано, что невозможно создать алгоритм, у которого сложность в худшем случае была бы ниже. В данной работе для вычисления редакционного расстояния используется библиотека difflib [34], которая входит в стандартную поставку

Рис. 1. Схема методики.

Python (ее части, критичные по производительности, реализованы на языке C).

3.2. Обнаружение точных повторов и Clone Miner

Мы использовали поиск точных повторов для того, чтобы создать карту повторов, на которой пользователь смог бы выбрать шаблон (образец) для поиска. С этой целью мы использовали инструмент поиска клонов в исходном коде Clone Miner [20]. Clone Miner – инструмент поиска клонов на основе токенов текста, а не с помощью построения дерева анализа программы. Токен – это отдельное слово (последовательность символов) в документе, отделенное от соседних слов разделителями следующих видов: “.”, “,”, “ ”, “(”, “)” и т.д. Например, фрагмент текста “FM registers” состоит из двух токенов. Clone Miner рассматривает текст как упорядоченную коллекцию токенов и находит повторяющиеся фрагменты (клоны) при помощи алгоритмов, основанных на суффиксных деревьях [35]. Выбор Clone Miner был обусловлен тем, что он просто интегрируетсяс другими инструментами при помощи интерфейса командной строки и поддерживает русский язык.

4. МЕТОДИКА

Основная задача методики – обеспечить учет семантики информации при поиске повторов. с помощью интерактивности, т.е. взаимодействия с пользователем. Схема методики представлена на рис. 1. Опишем методику подробнее.

Генерация карты повторов. При помощи инструмента englishClone Miner [20] в документе находятся все группы точных повторов. Каждому токену (слову) t в документе ставится в соответствие цвет в интервале от красного до белого в цветовой модели RGB: $color(t) = (h(t)/T_m) * R + (1 - (h(t))/T_m) * W$, где $R = [1,0,0]$, $W = [1,1,1]$, $h(t)$ – максимальная по мощности группа точных повторов, включающая данный токен (далее – температура токена), T_m – максимальная мощность группы точных повторов в данном документе (максимальная температура в документе)[1]. Чем ближе цвет токена к красному, тем он “теплее”. Данная метафора называется тепловой картой (englishheat map) [36], пример приведен на рис. 2.

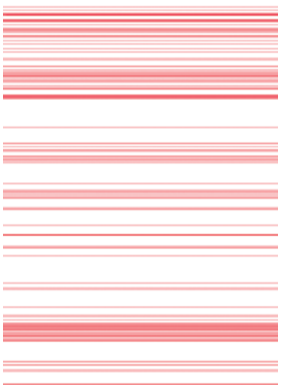

Рис. 2. Карта повторов.

Карта повторов позволяет пользователю увидеть в виде красных областей, кратко, без деталей, наиболее вероятные вхождения неточных повторов в документе. Поскольку токены, которые входят в максимально красные области, повторяются примерно одно и то же количество раз, то велика вероятность того, что вместе они образуют осмысленный неточный повтор.

Выбор образца для поиска. Пользователь находит на карте повторов (рис. 2) самый красный (самый “теплый”) участок и увеличивает его, просматривая соответствующий текст, и выбирает фрагмент (образец) для дальнейшего поиска (см.рис. 3). При этом он руководствуется не только цветом выбранного текста, но также стремится к тому, чтобы образец описывал целостное свойство ПО, т.е. к семантической замкнутости. Для соблюдения семантической замкнутости можно взять часть текста, имеющую белый цвет, так и не взять часть текста, имеющую красный цвет. Рассмотрим пример. Руководствуясь информацией с рис. 3 мы выбираем следующий фрагмент:

```
To alter the owner, you must also be a
direct or indirect member of the new
owning role, and that role must have
CREATE privilege on the table's schema.
(These restrictions enforce that alte-
```

[1] При этом мы рассматриваем только те группы, которые состоят из фрагментов длиной более четырех токенов, поскольку, согласно нашим экспериментам, именно такое ограничение позволяет отсеять большое количество ложноположительных срабатываний (false positives) [15].

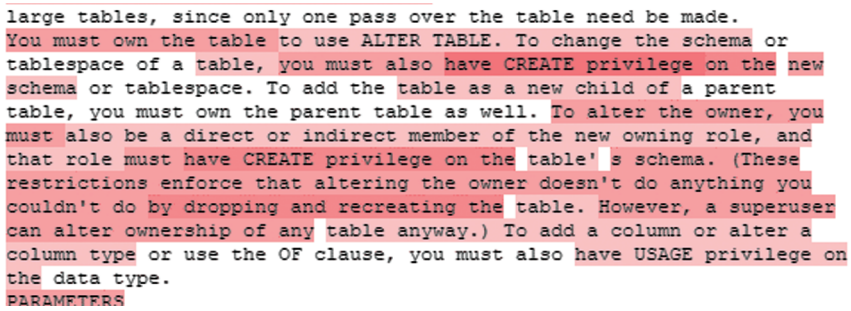

Рис. 3. Выбор образца для поиска (руководство по диалекту SQL СУБД PostgreSQL).

```
ring the owner doesn't do anything you
couldn't do by dropping and recreating
the table. However, a superuser can
alter ownership of any table anyway.)
```

Данный фрагмент описывает целостное свойство ПО, связанное с администрированием СУБД PostgreSQL: для того, чтобы разработчик мог сменить владельца таблицы базы данных, ему требуется обладать специфическими правами или быть администратором.

Поиск неточных повторов. Для выделенного фрагмента пользователь выбирает меру близости — число от $1/\sqrt{3}$ до 1 и запускает алгоритм поиска по образцу[2].

Формирование группы неточных повторов. Выдачей алгоритма является набор текстовых фрагментов — кандидатов в нечеткие повторы заданного образца. Пользователь имеет возможность редактировать ее “вручную”. Во-первых, он может удалять элементы выдачи, похожие на образец лишь синтаксически, но не по смыслу. Во-вторых, он может изменять границы любого элемента выдачи с тем, чтобы гарантировать семантическую замкнутость каждого из них.

5. ОПРЕДЕЛЕНИЕ ГРУППЫ НЕТОЧНЫХ ПОВТОРОВ

В данном разделе мы обобщим определение группы неточных повторов, предложенное нами ранее в [16, 37]. В отличие от предыдущего определения, здесь мы используем для задания меры близости параметр, а не константу, и позволяем варьировать текст в начале и в конце фрагмента. Документ, в котором ищутся повторы, обозначим D и будем рассматривать его как конечную последовательность символов; длину документа в символах обозначим как $\text{length}(D)$.

Определение 1. *Определим* ***текстовый фрагмент*** *как вхождение в документ D некоторой строки символов.*

Таким образом, каждому текстовому фрагменту g документа D соответствует, кроме строки символов, еще целочисленный интервал $[b, e]$, где b — это координата (смещение от начала документа в символах) первого символа фрагмента, e — координата последнего. Для текстового фрагмента g документа D будем писать $g \in D$. Функцию, выдающую по текстовому фрагменту g его целочисленный интервал, обозначим $[g]$. Функцию, выдающую по текстовому фрагменту g его строку символов, обозначим $\text{str}(g)$. С помощью $b(g)$ и $e(g)$ будем обозначать координату начала и конца g. Функцию, выдающую по g его длину, обозначим и определим так: $|g| = 1 + e(g) - b(g)$. Введем двухместный предикат $Before(g_1, g_2)$, который является истинным тогда и только тогда, когда $e(g_1) < b(g_2)$.

Определение 2. ***Группа неточных повторов.*** *Пусть у нас имеется набор текстовых фрагментов $g_1, \ldots, g_M$ документа D. Будем называть этот набор группой неточных повторов с мерой близости $k \in (1/\sqrt{3}, 1]$ (или просто группой неточных повторов), если выполнены следующие условия.*

1. $\forall i \in \{1, \ldots, M-1\}$ *выполнено* $\text{Before}(g_i, g_{i+1})$

2. Существует упорядоченный набор строк $(I_1, \ldots, I_N)$ такой, что имеется вхождение этого набора в каждый текстовый фрагмент, то есть $\forall j \in \{1, \ldots, M\}\ \forall i \in \{1, \ldots, N\} I_i \subset \text{str}(g_j)$ *и выполнено* $\forall i \in \{1, \ldots, N-1\} \text{Before}(I_i^j, I_{i+1}^j)$, *где I_i^j — вхождение I_i в g_j, а также выполнено следующее условие*:

$$\forall j \in \{1, \ldots, M\} \frac{\sum_{i=1}^{N} |I_i|}{|g_j|} \geq k$$

[2] Значение $1/\sqrt{3} \approx 0.577$ было выбрано из-за удобства последующих доказательств; при этом, исходя из наших экспериментов, мы сделали вывод о том, что если мера близости меньше, чем 1/2, то часто в документе содержится много случайных совпадений с такой мерой близости; выбранная нами нижняя граница незначительно превосходит 1/2.

Набор строк $(I_1,\ldots,I_N)$ назовем *архетипом* данной группы. Легко показать, что приведенное выше определение обобщает определение из [16, 37]. Если у нас имеется G – группа неточных повторов, то обозначим с помощью $|G|$ количество элементов в этой группе.

Определение 3. *Пусть у нас имеется некоторый текстовый фрагмент p документа D, то есть $p \in D$. Пусть имеется также $g \in D$. Будем говорить, что g является* ***неточным повтором*** *p с точностью k, если и p вместе образуют группу неточных повторов с точностью k в смысле определения* 2.

6. АЛГОРИТМ ПОИСКА НЕТОЧНЫХ ПОВТОРОВ ПО ОБРАЗЦУ

6.1. Описание алгоритма

Алгоритм 1: Поиск неточных повторов по образцу

Входные данные: D – документ,

p – образец, k – мера близости

Результат: R

// Фаза 1 (сканирование)

$W_1 \leftarrow \varnothing$

for $\forall w_1 : w_1 \in D \wedge |w_1| = L_w$ **do**

  if $\mathrm{d}_{di}(w_1, p) \le k_{di}$ **then**

    add w_1 to W_1

// Фаза 2 (“усушка”)

$W_2 \leftarrow \varnothing$

for $w \in W_1$ **do**

  $w_2' \leftarrow w$

  for $l \in I$ **do**

    for $\forall w_2 : w_2 \subseteq w \wedge |w_2| = l$ **do**

      if Compare(w_2, w_2', p) **then**

        $w_2' \leftarrow w_2$

  add w_2' to W_2

// Фаза 3 (фильтрация)

$W_3 \leftarrow \mathrm{Unique}(W_2)$

for $w_3 \in W_3$ **do**

  if $\exists w_3' \in W_3 : w_3 \subset w_3'$ **then**

    remove w_3 from W_3

$R \leftarrow W_3$

Алгоритм поиска вхождений образца p в документ D разбит на три фазы. На **фазе 1 (сканирование)** текст документа D сканируется окном w длины $L_w = \frac{|p|}{k}$ с шагом в один символ[3]; текстовый фрагмент, соответствующий текущему положению окна, сравнивается с образцом p по редакционному расстоянию, и если они близки, т.е. $d(p,w) \le k_{di}$, то этот фрагмент сохраняется в множестве W_1. Пороговое значение k_{di} при этом определяется следующим образом:

$$k_{\mathrm{di}} = |p|\left(\frac{1}{k} + 1\right)(1 - k^2). \qquad (1)$$

Его выбор будет обоснован ниже.

На **фазе 2 (“усушка”)** производится поиск наибольшего фрагмента текста внутри каждого элемента из множества W_1, максимально похожего на образец p. Таким образом происходит “усушка” элементов из W_1, то есть у них уменьшается длина. Это целесообразно, поскольку окно (а значит и все элементы W_1) имеет максимально возможный размер, который может иметь неточный повтор фрагмента p (см. лемму. 1). При “усушке” для каждого $w_2 \in W_1$ последовательно перебираются все фрагменты, входящие в него, от фрагментов длины $|p| * k$ до фрагментов длины $\frac{|p|}{k}$. Из них выбирается ближайший к образцу по редакционному расстоянию, а в случае нескольких таких – обладающий максимальной длиной. Результатом работы этой фазы является множество W_2. На **фазе 3 (фильтрация)** из множества W_2 удаляются одинаковые элементы (они появляются на предыдущей фазе в силу того, что в W_1 могут попадать текстовые фрагменты, отличающиеся друг от друга на сдвиг окна в несколько символов), а также удаляются элементы, полностью входящие в другие элементы W_2. Итогом этой фазы является множество W_3, которое оказывается результатом действия алгоритма, то есть множеством R.

Опишем используемые в алгоритме 1 вспомогательные функции. Функция Compare используется на фазе 2 для того, чтобы выяснить, какой из двух фрагментов текста является максимально близким к образцу p в смысле редакционного расстояния; если расстояния от обоих фрагментов до образца равны, предпочтение отдается фрагменту с большей длиной:

[3] Здесь и далее для более экономных выкладок не выполняется округление длин интервалов до целых. Тем не менее, все рассуждения и доказательства могут быть повторены с округлением.

$$\text{Compare}(w_1, w_2, p) = \begin{cases} true & d(w_1, p) < d(w_2, p) \\ false & d(w_1, p) > d(w_2, p) \\ |w_1| > |w_2| & d(w_1, p) = d(w_2, p) \end{cases}$$

Функция Unique получает набор текстовых фрагментов и делает так, чтобы каждый фрагмент входил в набор ровно один раз.

6.2. Полнота алгоритма

Критерий полноты для алгоритма поиска неточных повторов по заданному образцу определим следующим образом. Пусть для произвольного D и $p \in D$, а также для R – некоторой выдачи алгоритма и для любой группы неточных повторов G фрагмента p с точностью k (см. опр. 2) истинно следующее условие:

$$\forall g \in G \quad \exists w \in R: |g \cap w| \geq O_{\min}(k), \qquad (2)$$

где $O_{\min}(k) = \frac{|p|}{2}\left(3k - \frac{1}{k}\right)$. Смысл данного критерия заключается в том, что для любого фрагмента документа D, являющегося неточным повтором образца p, множество R будет содержать текстовый фрагмент, который существенно пересекает данный неточный повтор, позволяя пользователю легко узнать этот повтор в выдаче R. Процент пересечения ограничен снизу функцией $O_{\min}(k)$. $O_{\min}\left(\frac{1}{\sqrt{3}}\right) = 0$, при бóльших k $O_{\min}(k) > 0$, так как эта функция возрастает по k (ее производная равна $\frac{|p|}{2}\left(3 + \frac{1}{k^2}\right)$ и, очевидно, положительна при всех допустимых $\frac{1}{\sqrt{3}} < k \leq 1$). На практике наилучшие результаты достигаются для $k \geq 0.77$: при таких k $O_{\min}(k) > \frac{|p|}{2}$, то есть все элементы R "цепляют" неточные повторы, минимум, на половину длины образца. Отметим, что нижняя оценка в виде $O_{\min}(k)$ пессимистична: результаты экспериментов показывают большее пересечение выдачи алгоритма и неточных повторов в документе. Теперь перейдем к доказательству полноты предложенного алгоритма. В начале докажем несколько вспомогательных утверждений.

Лемма 1. *Пусть G – группа неточных повторов фрагмента p с точностью k. Тогда для $\forall g_1, g_2 \in G$ справедливо $k \leq \frac{|g_1|}{|g_2|} \leq \frac{1}{k}$.*

Доказательство. Пусть A – архетип группы G. Тогда $k|g_1| \leq |A|$ и $k|g_2| \leq |A|$. Так как $A \subset \text{str}(g_1)$ и $A \subset \text{str}(g_2)$, имеем: $k|g_1| \leq |A| \leq |g_1|$ и $k|g_2| \leq |A| \leq |g_2|$. Следовательно, $k|g_1| \leq |g_1|$ и $k|g_2| \leq |g_2|$. Разделив неравенства на $|g_2|$ и $k|g_2|$ соответственно, легко получаем требуемое. □

Лемма 2. *Пусть G – группа неточных повторов фрагмента p с точностью k. Тогда $\forall g \in G$ справедливо следующее*: $d(g, p) \leq (1 - k^2)|p|$.

Доказательство. Поскольку p и g принадлежат одной группе неточных повторов, то они имеют один и тот же архетип и могут быть представлены следующим образом:

$$p = v_0^p I_1 v_1^p I_2 \ldots v_{N-1}^p I_N v_N^p,$$
$$g = v_0^g I_1 v_1^g I_2 \ldots v_{N-1}^g I_N v_N^g,$$

где $I_1, I_2, \ldots, I_N$ является архетипом группы G, $v_0^p, v_1^p, \ldots, v_N^p$ – вариативной частью p, $v_0^g, v_1^g, \ldots, v_N^g$ – вариативной частью g.

Введем следующие обозначения: $v^p = v_0^p v_1^p, \ldots, v_N^p$, $v^g = v_0^g v_1^g, \ldots, v_N^g$, $A = I_1 I_2 \ldots I_N$. Тогда в силу (2) имеем $|A|/|p| \geq k \Rightarrow |p| - |v^p| \geq |p|k \Rightarrow |p| - |p|k \geq |v^p| \Rightarrow$ $\Rightarrow |p|(1 - k) \geq v^p$, и, аналогично $|g|(1 - k) \geq v^g$. При этом g из p можно получить заменой v_i^p на v_i^g, то есть $d(g, p) \leq |v^g| + |v^p| \leq (1 - k)(|p| + |g|)$. В силу леммы 1 имеем $|g| \leq k|p|$. Тогда $d(g, p) \leq (1 - k)(1 + k)|p| = (1 - k^2)|p|$. □

Лемма 3. *Для любых $p \in D$, $k \in (1/\sqrt{3}, 1]$ и G группы неточных повторов фрагмента p с точностью k (опр. 3) в отношении результата работы фазы 1 выполняется критерий полноты 2.*

Доказательство. Как указывалось выше, для редакционного расстояния по наибольшей подпоследовательности справедливо неравенство треугольника: $d(fr, p) \leq d(fr, g) + d(g, p)$. Согласно лемме 2, $d(g, p) \leq |p|(1 - k^2)$. Также известно, что $g \subseteq fr$. Следовательно, поскольку g можно получить из fr отбрасыванием всех символов, которые принадлежат $fr \backslash g$, справедливо $d(fr, g) \leq |fr| - |g|$. Но так как $|fr| = \frac{|p|}{k}$ и, согласно лемме 1, $|g| \geq k|p|$, то справедливо следующее: $|fr| - |g| \leq \frac{1}{k}|p| - k|p|$. Следовательно, $d(fr, p) \leq |p|\left(\frac{1}{k} - k + 1 - k^2\right) = |p|\left(1 + \frac{1}{k}\right)(1 - k^2)$. Очевидно, что при сканировании документа на фазе 1 найдется такое положение окна, что fr попадает в окно. Тогда в силу (1) $fr \in W_1$. Таким образом, справедливо следующее:

$$\forall g \in G : \left(|fr| = \frac{|p|}{k}, g \subseteq fr \right) \Rightarrow fr \in W_1.$$

Поскольку для любого неточного повтора найдется элемент из W_1, который не просто пересекает этот повтор, а целиком его содержит, то критерий 2 выполнен для множества W_1, если его взять в качестве множества R. □

Лемма 4. *Для любых $p \in D$, $k \in (1/\sqrt{3}, 1]$ и G группы неточных повторов фрагмента p с точностью k (опр. 3) в отношении результата работы фазы 2 выполняется критерий полноты 2.*

Доказательство. Формального доказательства мы не приводим в силу его объемности. Идея доказательства заключается в рассмотрении самой пессимистичной ситуации, при которой элементы $w_1 \in W_1$ во время "усушки" по длине до $k|p|$. Рассмотрение частных случаев (когда элемент "прижат" к началу или концу w_1, или находится в центре) позволяет убедиться в том, что утверждение леммы справедливо. □

Лемма 5. *Для любых $p \in D$, $k \in (1/\sqrt{3}, 1]$ и G* группы неточных повторов фрагмента p с точностью k (*опр.* 3) *в отношении результата работы фазы* 3 *выполняется критерий полноты 2.*

Доказательство. Во время третьей фазы алгоритма из W_2 лишь удаляются элементы, интервалы которых содержатся в интервалах других элементов. Очевидно, что если W_2 соответствовало критерию 2, то для W_3 это соответствие также сохранится. □

Теорема 1. *Для любых фрагмента $p \in D$, $k \in (1/\sqrt{3}, 1]$, соответствующей выдачи алгоритма R и для любой G группы неточных повторов фрагмента p с точностью k выполняется критерий полноты.*

Доказательство. Соответствие критерию 2 выдачи фаз 1–3 алгоритма было доказано в леммах 3, 4, 5. □

6.3. Оптимизация алгоритма

Представленный алгоритм оказался неудовлетворительным в смысле производительности: на документах размером около 2 Мб несколько Мб и при поиске образцов размером больше 100 символов время работы составляло более 1 часа. Более того, алгоритм выдавал большое количество ложноположительных срабатываний – в выдачу по несколько раз попадали одни и те же фрагменты текста, незначительно сдвинутые относительно друг друга. В связи с этим был предложен ряд следующих оптимизаций.

Оптимизация 1 применяется на фазе 1 (сканирование) и позволяет минимизировать количество вычислений d, тем самым существенно понижая время работы алгоритма. Идея оптимизации взята из известного алгоритма Бойера-Мура, предназначенного для поиска точных вхождений образца в строку [38]: при сканировании окна по документу результаты выполняется проверка того, на сколько можно сместить окно без риска пропустить искомый результат. Таким образом, если во время сканирования для положения окна w справедливо $d(w, p) > k_{\text{di}} + 1$, то будем сдвигать окно на $(d(w, p) - k_{\text{di}})/2$ символов вправо, в противном случае сдвигаем на один символ.

Оптимизация 2 применяется на фазе 2 ("усушка") и также позволяет минимизировать количество вычислений d. Подход аналогичен используемому в предыдущей оптимизации. Во время "усушки" текстового фрагмента w_1 при каждом положении окна w_2' обновляем (при необходимости) наименьшее значение $d(p, w_2')$ (обозначим его $d_{\min}$). Если для данного положения окна w_2' справедливо $d(p, w_2') > d_{\min} + 1$, сдвинем далее окно не на один, а на $(d(p, w_2') - d_{\min})/2$ символов вправо, в противном случае сдвинем окно на один символ вправо. Значение $d_{\min}$ обновляется в начале каждого цикла, соответствующего следующему значению длины сканирующего окна.

Оптимизация 3 применяется на фазе 3 (фильтрация) и позволяет уменьшить мощность множества W_3, которое с этой целью разбивается на максимальные, транзитивно замкнутые по пересечению, подмножества. Далее, для каждого такого подмножества выбирается фрагмент w_3 с наименьшим значением $d(w_3, p)$, а в случае наличия нескольких таких – фрагмент с наибольшей длиной, то есть наилучший из перекрывающихся определяется с использованием определенной выше функции Compare. Остальные фрагменты из множества W_3 удаляются.

Оптимизация 4 применяется на фазе 3 и производит расширение всех текстовых фрагментов W_3 до целых слов. То есть начало и конец текстового фрагмента могут нарушать границу слова, и в этом случае границы текстового фрагмента соответственно "раздвигаются" с тем, чтобы включить в себя эти слова целиком. Это позволяет уменьшить количество ложноположительных срабатываний в выдаче алгоритма.

Оптимизация 5 применяется на фазах 1 и 2 и заключается в повторном использовании d для одних и тех же строк и распараллеливании "усушки" элементов множества W_1.

Покажем теперь, как эти оптимизации влияют на полноту алгоритма.

Теорема 2. *Оптимизации* 1, 2, 4, 5 *сохраняют свойство полноты алгоритма*.

Доказательство. Пусть у нас есть строки, каждая из которых является конкатенации двух строк: $s_1 = ab$ и $s_2 = bc$, причем $|a| = |c|$, а $d(s_1, s_2) = d$. В таком случае легко показать, что $|a| + |c| \geq d$. Пользуясь этим, легко доказать полноту оптимизации 1. Полнота оптимизации 2 доказывается аналогично. Полнота оптимизацией 4 не вызывает сомнений, т.к. лишь расширяет элементы выдачи. Наконец, оптимизация 5 полна, т.к., фактически, затрагивает только реализацию алгоритма, не меняя его общей схемы. □

Замечание 1. *Возможны ситуации, когда оптимизация* 3 *нарушает критерий полноты, однако проведенные нами эксперименты показали, что на практике количество таких ситуаций крайне незначительно*.

7. СЛОЖНОСТЬ АЛГОРИТМА ПОИСКА ПО ОБРАЗЦУ

В качестве существенных переменных, влияющих на быстродействие алгоритма, выделим длину документа $|D|$, длину образца $|p|$, значение k и мощность группы неточных повторов образца $|G_p|$. Проведем оценку сложности алгоритма по этим параметрам.

Средняя сложность вычисления d (имеется в виду используемое нами редакционное расстояние) составляет $\mathbb{O}(|p|^2)$ [32]. Средняя сложность фазы 1 алгоритма, таким образом, пропорциональна $|p|^2$ и $|D|$. На фазе 2 для каждого фрагмента $w_1 \in W_1$ выполняется перебор всех его внутренних фрагментов. Легко показать, что их количество пропорционально $|p|$. Кроме того, мощность множества W_1 пропорциональна $|G_p|$ и k_{di}. В итоге сложность фазы 2 составляет $\mathbb{O}(|G_p|)$ и $\mathbb{O}(|p|^4)$. Операции фазы 3 имеют сложность $\mathbb{O}(|W_2| * \log|W_2|)$, но поскольку $|W_2| = |W_1|$, сложность фазы 3 составляет $\mathbb{O}(|G_p| * \log|G_p|)$ и $\mathbb{O}(|p| * \log|p|)$. Оптимизации 1 и 2 в среднем влекут "пропуски" при переборе, размер которых пропорционален k_{di} (и значит $|p|$), делая оценку сложности фаз 1 и 2 $\mathbb{O}(|p|)$ и $\mathbb{O}(|p|^3)$ соответственно. Таким образом, при k = const время работы алгоритма в среднем можно оценить следующим образом: $\mathbb{O}(|D|)$, $\mathbb{O}(|p|^3))$ и $\mathbb{O}(|G_p| * \log|G_p|)$.

Теорема 3. *Сложность работы алгоритма при фиксированных D и p оценивается в среднем как* $\mathbb{O}(1/k^4)$.

Доказательство этой теоремы опустим в виду его громоздкости.

8. ЭКСПЕРИМЕНТЫ

Теоретических оценок сложности недостаточно для определения реального быстродействия предложенного алгоритма. Дело в том, что эти оценки делались по выделенным существенным параметрам алгоритма независимо (с целью упрощения доказательств), в то время как реальная сложность может зависеть от их комбинаций. Еще одним аргументом в пользу необходимости экспериментов является тот факт, что теоретические оценки не дают представления о реальных интервалах изменения существенных параметров. Наконец, помимо быстродействия необходимо также оценить и другие свойства алгоритма.

Мы провели эксперименты, отвечающие на следующие вопросы: (i) каково быстродействие алгоритма поиска по образцу на реалистичных данных и (ii) какова величина реальных выдач алгоритма. Первый вопрос важен потому, что алгоритм применяется пользователем в интерактивном режиме, поэтому время его работы не должно превышать нескольких минут. Относительно выдачи алгоритма нами было доказано, что для конкретного образца она содержит все имеющиеся в документе его нечеткие повторы. Но не ясно следующее: во-первых, сколько таких повторов бывает в документации ПО вообще, во-вторых, сколько алгоритм выдает ложноположительных срабатываний. Если один из этих параметров или оба вместе становятся причиной больших выдач (более 100 текстовых фрагментов), то их будет сложно анализировать человеку.

Для экспериментов были взяты 19 документов промышленных проектов на русском и английском языках, описанных в [16]. Эксперименты проводились на компьютере со следующими характеристиками: Intel Core i7 2600, 3.4 ГГц, RAM 16 Гб. Документы были конвертированы утилитой Pandoc [39] в формат "плоского текста" (кодировка UTF-8) и после конвертации имели объем от 0.04 Мб до 2.5 Мб (в среднем – 0.75 Мб). Мы склонны считать, что этот диапазон значений является реалистичным для параметра $|D|$. Но следует отметить, что для уточнения границ данного параметра следует провести более представительную выборку для разных типов документации.

Эксперименты были организованы следующим образом. На каждом из исследуемых документов алгоритм запускался для образцов с длиной от 50 до 1000 символов с шагом в 50 символов. Текстовый фрагмент в 1000 символов составляет приблизительно 0.25 страницы docx-формата, т.е. это большой фрагмент, а, исходя из наших экспе-

риментов, размеры повторов, как правило, существенно меньше. Значение меры близости k мы итерировали от 0.6 до 1 с шагом 0.1 для каждого выбранного образца в каждом из рассматриваемых документов. Образец выбирался следующим образом. Имея фиксированное значение длины образца, мы, следуя нашей методике, выбирали самый "теплый" участок в документе, имеющий такую длину, вычисляя его автоматически как фрагмент, на котором следующее выражение принимает максимальное значение: $\sum_{t\in fr} h(t)$, где t – это токен, принадлежащий фрагменту fr, $h(t)$ – его температура, а сумма берется по всем токенам фрагмента, включая возможно неполные крайне левый и крайне правый токены.

Анализируя данные, полученные в ходе экспериментов и отвечая на вопрос о реалистичном быстродействии алгоритма, мы установили следующее: в 38% случаев алгоритм выполнялся менее чем за 5 секунд, в 78% случаев – менее, чем за 30 секунд, в 90% случаев – менее, чем за 2 минуты. Это вполне приемлемое быстродействие для интерактивной работы.

Были получены следующие экспериментальные данные о величине выдач предложенного алгоритма: 84% выдач включают менее 100 элементов, 5% выдач – от 100 до 200 элементов, 5.6% – от 200 до 600 элементов, 5.4% – от 600 до 1000 элементов. Из этих данных, а также в силу Теоремы 1, можно утверждать, что количество повторов в группах в подавляющем большинстве случаев невелико и не превосходит 100 элементов.

9. ЗАКЛЮЧЕНИЕ

В данной работе представлена методика интерактивного поиска неточных повторов в документации ПО. Методика решает вопрос семантической осмысленности (meaningfulness) неточных повторов с помощью привлечения пользователя. Ему в помощь для того, чтобы он мог определять наиболее вероятные вхождения неточных повторов, строится тепловая карта документа на основе точных повторов. Также в рамках методики создан алгоритм поиска неточных повторов по образцу, представлены оптимизации алгоритма. Доказана полнота алгоритма в том смысле, что все имеющиеся в тексте неточные повторы данного образца попадают в выдачу алгоритма, точнее существенно пересекаются с определенными элементами этой выдачи, и поэтому пользователь может их легко идентифицировать и исправить их границы, включив в выдачу целиком. В работе также представлены оценки сложности алгоритма, проведены эксперименты. Полученные результаты позволяют утверждать, что группы повторов в документации ПО, в основном, не превышают 100 элементов, а сам алгоритм имеет быстродействие, приемлемое для практических нужд.

В дальнейшем с помощью данного алгоритма и предложенной модели экспериментов планируется детально исследовать различные виды документации ПО (прежде всего – API-документацию). Также планируется более тщательно исследовать поведение алгоритма при различной динамике входных параметров (длина образца и мера близости). Наконец, планируется перейти к автоматическим способам выделения семантически осмысленных повторов с применением машинного обучения. Также необходимо выполнить обстоятельный анализ различных видов неточных повторов для разных видов документации ПО. Кроме этого, интересными направлениями исследований может быть интеграция повторного использования документации с автоматизированной разработкой тестов [40, 41] и визуализация структуры повторов с помощью онтологий [42].

10. БЛАГОДАРНОСТИ

Работа выполнена при частичной поддержке РФФИ, грант № 16-01-00304.

СПИСОК ЛИТЕРАТУРЫ

1. *Brooks F.P.* The Mythical Man-Month: Essays on Software Engineering / F. P. Brooks. Addison-Wesley, 1975.
2. *Parnas D.L.* Precise Documentation: The Key to Better Software / D. L. Parnas // The Future of Software Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. P. 125–148.
3. *Bassett P.G.* Framing Software Reuse: Lessons from the Real World / P. G. Bassett. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.
4. *Jarzabek S.* Research journey towards industrial application of reuse technique / S. Jarzabek, U. Pettersson // ICSE. 2006. P. 608–611.
5. *Irshad M.* A systematic literature review of software requirements reuse approaches / M. Irshad, K. Petersen, S.M. Poulding // Information & Software Technology. 2018. V. 93. P. 223–245.
6. Horie M. Tool Support for Crosscutting Concerns of API Documentation / M. Horie, S. Chiba // Proceedings of the 9th International Conference on Aspect-Oriented Software Development. New York, NY, USA: ACM, 2010. P. 97–108.
7. *Oumaziz M.A.* Documentation Reuse: Hot or Not? An Empirical Study / M.A. Oumaziz, A. Charpentier, J.-R. Falleri, X. Blanc // Mastering Scale and Complexity in Software Reuse: 16th International Conference on

Software Reuse. Springer International Publishing, 2017. P. 12–27.

8. *Koznov D.V.* DocLine: A method for software product lines documentation development / D.V. Koznov, K.Yu. Romanovsky // Programming and Computer Software. 2008. V. 34. № 4. P. 216–224.
9. *Romanovsky K.* Refactoring the Documentation of Software Product Lines / K. Romanovsky, D. Koznov, L. Minchin // Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2011. V. 4980. P. 158–170.
10. *Jarzabek S.* Documentation Reuse: Managing Similar Documents / S. Jarzabek, D. Dan // FedCSIS. 2017. P. 1325–1334.
11. *Juergens E.* Can clone detection support quality assessments of requirements specifications? / E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S.Wagner, C. Domann, J. Streit // Proceedings of ACM/IEEE 32nd International Conference on Software Engineering. V. 2. 2010. P. 79–88.
12. *Nosál' M.* Reusable software documentation with phrase annotations / M. Nos'al', J. Porubän // Central European Journal of Computer Science. 2014. V. 4. № 4. P. 242–258.
13. *Nosál' M.* Preliminary report on empirical study of repeated fragments in internal documentation / M. Nosál', J. Porubän // Proceedings of Federated Conference on Computer Science and Information Systems. 2016. P. 1573–1576.
14. *Wingkvist A.* Analysis and visualization of information quality of technical documentation / A. Wingkvist, W. Lowe, M. Ericsson, R. Lincke // Proceedings of the 4th European Conference on Information Management and Evaluation. 2010. P. 388–396.
15. *Koznov D.* Clone Detection in Reuse of Software Technical Documentation / D. Koznov, D. Luciv, H.A. Basit, O.E. Lieh, M. Smirnov // International Andrei Ershov Memorial Conference on Perspectives of System Informatics (2015). Springer Nature, 2016. V. 9609 of Lecture Notes in Computer Science. P. 170–185.
16. *Luciv D.V.* Detecting Near Duplicates In Software Documentation / D.V. Luciv, D.V. Koznov, G.A. Chernishev, A.N. Terekhov, K.Yu. Romanovsky, D.A. Grigoriev // Programming and Computer Software. 2018. V. 44. № 5.
17. Koznov D.V. Duplicate management in software documentation maintenance / D.V. Koznov, D.V. Luciv, G.A. Chernishev // Proceedings of V International conference Actual problems of system and software engineering. V. 1989. CEUR Workshop Proceedings, 2017. P. 195–201.
18. *Duplicate Finder.* URL: http://www.math.spbu.ru/user/kromanovsky/docline/index.html.
19. *Luciv D.V.* Poster: Duplicate Finder Toolkit / D.V. Luciv, D.V. Koznov, G.A. Chernishev, H.A. Basit, K.Yu. Romanovsky, A.N. Terekhov // Proceedings of the International Conference on Software Engineering (ICSE 2018). 2018. P. 171–172.
20. *Basit H.A.* Efficient Token Based Clone Detection with Flexible Tokenization / H.A. Basit, S.J. Puglisi, W.F. Smyth, A. Turpin, S. Jarzabek // Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. New York, NY, USA: 2007. P. 513–516.
21. *Rago A.* Identifying duplicate functionality in textual use cases by aligning semantic actions / A. Rago, C. Marcos, J.A. Diaz-Pace // Software & Systems Modeling. 2016. V. 15. № 2. P. 579–603.
22. *Ukkonen E.* Finding approximate patterns in strings / E. Ukkonen // Journal of Algorithms. 1985. V. 6. № 1. P. 132–137.
23. *Broder A.Z.* On the resemblance and containment of documents / A. Z. Broder // Compression and Complexity of Sequences 1997. Proceedings. IEEE, 1997. P. 21–29.
24. *Wu S.* Fast Text Searching: Allowing Errors / S. Wu, U. Manber // Commun. ACM. 1992. V. 35. № 10. P. 83–91.
25. *Landau G.M.* Fast string matching with k differences / G. M. Landau, U. Vishkin // Journal of Computer and System Sciences. 1988. V. 37. № 1. P. 63–78.
26. *Myers G.*A Fast Bit-vector Algorithm for Approximate String Matching Based on Dynamic Programming / G. Myers // J. ACM. 1999. V. 46. № 3. P. 395–415.
27. *Levenshtein V.* Binary codes capable of correcting spurious insertions and deletions of ones / V. Levenshtein // Problems of Information Transmission. 1965. V. 1. P. 8–17.
28. *Smyth W.* Computing Patterns in Strings / W. Smyth. Addison-Wesley, 2003. P. 423.
29. *Gusfield, D.* Algorithms on Strings, Trees, and Sequences / D. Gusfield. Cambridge University Press, 1997.
30. *Bergroth L.A* survey of longest common subsequence algorithms / L. Bergroth, H. Hakonen, T. Raita // String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on. 2000. P. 39–48.
31. *Leskovec J.* Mining of massive datasets / J. Leskovec, A. Rajaraman, J. D. Ullman. Cambridge university press, 2014.
32. *Ratcliff J.W.* Pattern Matching: The Gestalt Approach / J. W. Ratcliff, D. E. Metzener // Dr. Dobb's Journal. 1988. V. 13. № 7. P. 46–72.
33. *Abboud A.* Tight hardness results for LCS and other sequence similarity measures / A. Abboud, A. Backurs, V.V. Williams // Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on. 2015. P. 59–78.

34. Python DiffLib module. URL: https://docs.python.org/3/library/difflib.html.

35. *Abouelhoda M.I.* Replacing Suffix Trees with Enhanced Suffix Arrays / M. I. Abouelhoda, S. Kurtz, E. Ohlebusch // J. of Discrete Algorithms. 2004. V. 2. № 1. P. 53–86.

36. *Špakov O.* Visualization of eye gaze data using heat maps / O. Špakov, D. Miniotas // Elektronika ir elektrotechnika. 2007. P. 55–58.

37. *Luciv D.V.* Detecting Near Duplicates in Software Documentation. 2017. arXiv : 1711.04705.

38. *Boyer Robert S.* A fast string searching algorithm / Robert S. Boyer, J. Strother Moore // Communications of the ACM. 1977. V. 20. № 10. P. 762–772.

39. Pandoc: a universal document converter. URL: https://pandoc. org/.

40. Drobintsev P. D. A formal approach to test scenarios generation based on guides / P. D. Drobintsev, V. P. Kotlyarov, A. A. Letichevsky // Automatic Control and Computer Sciences. 2014. Dec. V. 48. № 7. P. 415–423.

41. *Pakulin N.V.* Model-based testing of Internet Mail Protocols / N.V. Pakulin, A.N. Tugaenko // Proceedings of the Institute for System Programming. 2011. V. 20. P. 125–141.

42. *Gavrilova T.* Diagrammatic Knowledge Modeling for Managers – Ontology-based Approach / T. Gavrilova, D. Kudryavtsev // Proceedings of the International Conference on Knowledge Engineering and Ontology Development. 2011. P. 386–389.