Санкт-Петербургский государственный университет

Математико-механический факультет
Кафедра прикладной кибернетики

Шелиховский Анатолий Алексеевич

Быстрый поиск по образцу неточных повторов в текстовых документах

Бакалаврская работа

Научный руководитель:
д.т.н., профессор Кознов Д.В.

Рецензент:
ст. преп. Луцив Д.В.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY

Faculty of Mathematics and Mechanics
Department of Applied Cybernetics

Anatolii Shelikhovskii

Quick near duplicates search by the pattern in the software documentation

Bachelor's Thesis

Scientific supervisor:
Doctor of Engineering Sciences, professor Koznov Dmitry

Reviewer:
Senior Lecturer Luciv Dmitry

Saint-Petersburg
2018

Оглавление

Введение

Современная документация программного обеспечения (ПО) зачастую имеет такую же сложную структуру, как и само ПО. Программные проекты производят много текстовой информации, и анализ этих данных является действительно важной задачей для практики. Создание адекватной документации остается проблемой в течение многих лет.

Дублирование контента в документации неизбежно, оно возникает в следствие намеренного или случайного копирования информации. Существует ряд причин, по которым важно идентифицировать дублирование. Одной из них является поддержание качества документации и переиспользование повторяющихся фрагментов. В течение своего жизненного цикла документация накапливает большое число «близких дубликатов», то есть фрагментов текста, которые был скопированы из некоторого источника и впоследствии изменены. Некоторые повторы — это точные копии, и обычно их легко найти. Но если вы вовлечены в работу, связанную с жизненным циклом документа (например, при разработке проекта), то многие из ваших фрагментов текста будут копиями, которые в основном представляют собой разные версии одного и того же текста. В связи с этим актуальна задача поиска близких дубликатов в документации ПО.

Одним из способов решать эту задачу является выполнение поиска неточного вхождения образца в строку. Существует несколько различных подходов к решению данной проблемы [1, 2, 3, 4, 5]. При этом ввиду существует некоторая специфика для документации ПО, которая накладывает ограничения на реализацию («близость» строк определеяется в зависимости от введенной модели неточных повторов).

Для решения данной задачи предложен инструмент Duplicate Finder [6]. Это программный продукт для работы с неточными повторами в документации ПО, целью которого является разработка подхода, позволяющего решить проблемы со стандартизацией документации и одновременным рефакторингом повторяющейся информации. Также Duplicate

Finder позволяет находить неточные повторы в документации в двух режимах — автоматический и интерактивный. Автоматический режим позволяет предварительно анализировать наличие повторов в документе, их количество и вид. Интерактивный режим может использоваться для анализа и управления повторами, введенными пользователем вручную.

Интерактивный режим основывается на алгоритме поиска неточных повторов по образцу. Ранее был реализован алгоритм поиска по образцу [7], который выдавал результаты, приемлемые с точки зрения качества и производительности. Однако в рамках исследования было выявлено, что возможности его оптимизации не исчерпаны.

Постановка задачи

Цель работы заключалась в распараллеливании и оптимизации существующего алгоритма поиска неточных повторов по образцу в документации ПО, используемого в инструменте Duplicate Finder. В связи с этим были сформулированы следующие задачи.

1. Проведение экспериментов по измерению скорости и качества работы существующего алгоритма.

2. Проектирование оптимизаций: выяснение и описание возможностей по оптимизации существующего алгоритма, изучение технических средств реализации.

3. Реализация и интеграция оптимизаций в инструмент Duplicate Finder.

4. Обоснование полноты выполненных оптимизаций, выполнение экспериментов и анализ результатов.

1. Обзор

В этой главе рассматривается документация программного обеспечения, её виды и возникающие проблемы, обсудим существующие методы для решения задачи поиска повторов в документации. Также опишем модель неточных повторов, использующуюся в данной работе и существующий алгоритм поиска повторов по образцу в проекте Duplicate Finder. Обзор, представленный в этой главе, следует материалам, изложенным в [7].

1.1. Документация ПО, её виды и проблемы разработки

Документация ПО является значимым моментом в разработке компьютерных систем. Следует отметить, что проблемы с документацией для открытых проектов в настоящее время очевидно присутствуют. Наиболее ярким примером является документация проекта Linux Kernel [8]. Проблемы с отечественной документацией отмечались многими исследователями в 90-х и 2000-х годах: А.Н. Тереховым [9], А.А. Шалыто [10], В.В. Липаевым [11] и другими.

При этом люди, участвующие в проектах, обычно имеют нужную профессиональную подготовку, однако приложение академических знаний зачастую оказывается трудным и безуспешным. Следовательно, требуются вспомогательные исследования, и создание новейших подходов по управлению документацией программного обеспечения.

Рассмотрим некоторые виды документации ПО, описанные в различных классификациях в литературе. Известный исследователь в области программной инженерии И. Соммервил выделил два следующих вида документации: продукта и процесса [12]. В первый тип он включил руководство по установке предоставляемых сервисов, руководство пользователя, руководство по администрированию, руководство по началу работы. Документация процесса состоит из различных этапов разработки: отчетов, планов проекта, стандартов. Первый вид, в свою оче-

редь, включает в себя документацию системы (спецификация архитектуры, требований, функциональности, руководство по тестированию, сопровождению и листинги кода) и документацию пользователя.

Т. Лезбридж, исходя из результатов опросов разработчиков ПО, описывает такие виды документации [13]:

- спецификация системы;
- спецификация требований
- общая и детальная архитектура;
- архитектура низкого уровня;
- документация для тестирования.

Многие методологи выделяют также API—документацию [14]. В ней описываются интерфейсы модулей и библиотек: классы, функции, типы данных и константы, и др.

1.2. Анализ повторов

При редактировании документов зачастую копируется уже существующее описание, которое модифицируется и дополняется в другом контексте. Таких копирований обычно осуществляется немало, они по—разному редактируются, и поэтому в документации появляется большое число неточных повторов, которые являются вариациями одних и тех же данных. Помимо этого, в документации накапливаются определенные фрагменты текста, такие как вводные выражения, сходственные замечания и т.д.

Повторы в документации расцениваются по—разному различными исследователями. Одни считают [15], что повторы — это признак избыточности, они понижают качество документации ПО. С другой стороны, в [16] говорится, что наличие повтором способствует лучшему пониманию текста.

Рассмотрим некоторые работы, содержащие сведения о повторах в документации.

М. Хори [17] описывает инструмент CommentWeaver, позволяющий управлять повторами в Javadoc. В соответствии с результатами экспериментов с проверкой исходных кодов Eclipse, приблизительно до 20% документации Javadoc дублируется. Использование CommentWeaver уменьшает размер документации примерно на 10%. При этом в работе рассматриваются только точные повторы.

Я. Порубан и М. Носал [18] расширяют работу [17] благодаря рассмотрению неточных повторов. Они определяют неточные повторы как множество фрагментов текста, имеющих одинаковый семантический смысл. Инструмент, представленный составителями этой работы, предоставляет возможность добавлять унифицированные неточные повторы в Javadoc с помощью аннотаций в исходном коде на Java. Однако данное определение слишком неформально, и проблема поиска повторов не рассматривается.

Э. Юргенс и др. [16] анализирует точные повторы в различных спецификациях, используя программную утилиту Clone Detective [19]. Авторы исследовали 28 документов, рассматривая повторы длиной не менее 20 слов. По результатам авторы делают вывод, что в среднем повторяются 13,6% текста. Он также отмечает значимость неточных повторов, однако в своей работе не учитывают их.

Существует ещё некоторое количество работ, посвящённых анализу повторов, об этом более подробно можно прочитать в [7].

1.3. Средства и методы обнаружения повторов

Ранее были описаны различные подходы для обнаружения повторов в документации ПО. В данном разделе рассмотрим некоторые средства и методы поиска повторов в текстах, которые могут быть применимы к нашей проблеме.

Одной из самых известных моделей является статистическая модель N—грамм [4]. В ней текст рассматривается как множество токенов

(перекрывающихся текстовых фрагментов одной длины), называемых N—граммами. На основе похожести множества N—грамм модель предоставляет возможность делать некоторые заключения о синтаксической близости текстов.

Тематическое моделирование [5] позволяет разделять множество документов на классы, которые относятся к различной тематике. Детальнее можно исследовать текст, используя векторные представления слов с помощью различных инструментов, например, Google Word2Vec [20]. В программных библиотеках NLTK [21] и Texterra [22] реализовано немалое число алгоритмов по анализу текста на естественном языке. Сравнительный анализ нескольких этих и некоторых других систем можно посмотреть в [23].

Поиску повторов в текстах также посвящён ряд работ в рамках информационного поиска. Однако они не рассматривают документацию программного обеспечения.

1.4. Терминология модели неточных повторов

Рассмотрим некоторые важные определения из описания модели неточных повторов, которые используются в данной работе (более подробное описание можно найти в [7]).

Текстовый фрагмент

Текстовым фрагментом называется вхождение определенной строки в документ. То есть каждому текстовому фрагменту соответствует пара значений $[b,\ e]$, такая что b — координата в тексте первого символа фрагмента, а e — это координата последнего. Обозначается как g.

Отношение порядка

Before — предикат, который является истинным для двух текстовых фрагментов g_1 и g_2 тогда и только тогда, когда $e(g_1) < b(g_2)$.

Редакционное расстояние между строками

Редакционное расстояние между парой строк определяется как минимальное число операций редактирования, позволяющих преобразовать одну строку в другую [24]. Далее в работе под операцией редакти-

рование будем понимать вставку или удаление символа [25].

Группа неточных повторов с мерой близости k

Рассмотрим набор фрагментов текста $fr_1, fr_2, \ldots, fr_M$. Этот набор является группой неточных повторов с мерой близости k, если выполняются следующие условия:

1. $\forall\, i\; Before(fr_i, fr_{i+1})$.

2. Существует такой упорядоченный набор строк $I_1, \ldots, I_N$ такой, что имеется вхождение этого набора в каждый из текстовых фрагментов $fr_1, fr_2, \ldots, fr_M$. (Далее этот набор $\{I_1, \ldots, I_N\}$ будем называть архетипом)

3. Число k такое, что $\frac{1}{\sqrt{3}} < k \leq 1$, и при этом выполняется условие:

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

1.5. Алгоритм поиска неточных повторов, реализованный в инструменте Duplicate Finder

Алгоритм поиска по образцу, реализованный в инструменте Duplicate Finder и описанный в [7], состоит из нескольких стадий. На вход ему подается документ D, образец p для поиска и мера близости k.

1. Фаза «сканирования». На данной фазе текст документа D сканируется окном определенной длины и соответствующий ему фрагмент сравнивается (подсчитывается редакционное расстояние) с p. В случае их достаточной близости (определяется по мере близости k) этот фрагмент добавляется в результирующее множество S_1.

2. Фаза «усушки». Наиболее затратная стадия. Анализируются все подэлементы внутри каждого фрагемента из множества S_1 и выполняется поиск наиболее похожего на p, который добавляется в S_2. В результате на выходе данной фазы получаем множество S_2.

3. Фаза «фильтрации». Здесь происходит отбор значимых элементов из множества S_2, то есть отбрасываются идентичные фрагменты текста, а так же фрагменты, которые целиком содержатся в других. В конечном итоге получаем множество S_3, которое является результатом работы алгоритма.

Важной особенностью архитектуры вышеописанного алгоритма является его модульность. Благодаря этому есть возможность подменять отдельные его части (например, при использовании других определений редакционного расстояния).

2. Исследование текущего алгоритма и эксперименты

2.1. Тестовые данные

Формирование данных для анализа алгоритма является очень важным аспектом, потому что плохо спроектированные данные тестируют не все возможные сценарии. Тем самым результаты работы программы будут содержать не до конца подлинную информацию. Для экспериментов был взят тестовый набор, описанный в [7]. Он включает в себя 19 документов программного обеспечения (как на русском, так и на английском языках), представленных в табл. 1.

Первые 12 документов выбраны из проектов с открытым исходным кодом, остальные 7 — коммерческие. Файлы по каждому проекту были соединены в один, и далее в нём удалялась разметка и другая информация, не несущая смысла, т.е. оставался так называемый «плоский» текст.

2.2. Схема экспериментов

Проверка работы алгоритма осуществлялась на компьютере с 8 Гб оперативной памяти и 4—ядерным процессором Intel Core i5—7400T (частота 2,4 ГГц).

Был осуществлён следующий эксперимент: для всех значений меры близости из диапазона {0.57,0.67,...,0.97} (1 означает абсолютное сходство) проводился анализ вышеописанных документов из табл. 1. Для каждого из них были выбраны образцы различной длины для поиска (от 20 до 100 символов с шагом 10 символов). Таким образом всего алгоритм тестировался на 171 образцах. Следует заметить, что отбор каждого образца проходил вручную исходя из тепловой карты, построенной с помощью инструмента Duplicate Finder. Данный выбор основывается на предположении, что алгоритм будет работать дольше при большем числе повторений образца в документе.

Таблица 1: Список документов ПО для тестирования

№	Документ	Тип документа	Доступ
1	DocBook definitive guide	Справочник по языку	http://docbook.org/
2	Subversion Book	Рук—во по администрированию	https://subversion.apache.org/
3	Zend Framework V1 guide	Рук—во программиста	https://github.com/zendframework/zf1/tree/master/documentation
4	Linux Kernel Documentation	Рук—во программиста	https://github.com/torvalds/linux/tree/master/kernel
5	GNU Core Utils Manual	Рук—во пользователя	https://www.gnu.org/software/coreutils/coreutils.html
6	Postgre SQL Manual	Справочник по языку	https://www.postgresql.org/
7	The GIMP user manual	Рук—во пользователя	https://www.gimp.org/
8	Blender reference manual	API—документация	https://www.blender.org/
9	LibLDAP Manual	API—документация	http://www.openldap.org/
10	Eclipse SWT Reference	API—документация	http://git.eclipse.org/
11	Python Requests	Рук—во программиста	http://python—requests.org/
12	Qt Quick Reference	Рук—во программиста	http://wiki.qt.io/Developer_Guides
13	Документ 1	Рук—во пользователя	—
14	Документ 2	Рук—во пользователя	—
15	Документ 3	Спецификация архи—тектуры	—
16	Документ 4	Спецификация архи—тектуры	—
17	Документ 5	Спецификация архи—тектуры	—
18	Документ 6	Спецификация архи—тектуры	—
19	Документ 7	Спецификация архи—тектуры	—

Измерялись следующие две величины: время работы и число элементов в выдаче алгоритма для каждого образца. В результате были построены гистограммы зависимости каждого из параметров от длины исходного паттерна.

2.3. Анализ результатов

По результатам эксперимента было определено, что в среднем алгоритм работает около 6 секунд. Однако в некоторых случаях время становится в разы большим: если на вход подается документ крупного размера, либо образец большой длины (ближе к 100 символам). Худшим результатом по времени являются 32 секунды. В этом и состоит проблема текущего алгоритма.

Что касается качества выдаваемых данных, эмпирическим способом установлено, что программа в большинстве своём выдает существенные повторы, то есть корректные результаты. Только в незначительном числе случаев может возникнуть нарушения полноты выдачи. Такое происходит, когда есть некоторое число подряд идущих и при этом пересекающихся повторов, которые объединяются в один. Хотя возможен такой случай, что в действительности их должно быть два или более. Например, для случая «qwe qwe qwe qwe» — в выдаче программы останется только образец «qwe qwe».

3. Проектирование оптимизаций

3.1. Описание оптимизаций

Изначально предложенный алгоритм поиска по образцу работал неприемлемо долго, и его выдача была неприемлемо большой. С помощью применения различных оптимизаций удалось добиться удовлетворительных значений по обоим показателям.

Ниже представлен список оптимизаций, которые были реализованы:

• распараллеливание некоторых стадий алгоритма (фазы «усушки» и «сканирования»);

• группировка пересекающихся образцов, найденных на фазе «фильтрации»;

• применение статически компилируемого в машинный код диалекта языка Python Cython (далее для краткости будем называть его компилятором);

• оптимизация ряда критических с точки зрения производительности участков средствами самого Python без затрагивания алгоритма.

Остановимся более подробно на каждой оптимизации. Рассмотрим мотивацию к их применению, а также псевдокод с описанием произведенных изменений.

3.2. Распараллеливание некоторых стадий алгоритма

Следуя описанию алгоритма, на втором этапе происходит «усушка» найденных фрагментов, полученных в результате фазы сканирования текста. Это наиболее затратная стадия алгоритма, так как для каждого фрагмента необходимо проанализировать все отрезки меньшей длины не короче определенного значения и выбрать среди них наиболее подходящий (расстояние до искомого паттерна минимально). При этом каждый шаг происходит независимо от других, что привело к идее выполнения этих шагов параллельно. Таким образом первой оптимизацией является распараллеливание фазы «усушки».

Оптимизация заключается в следующем. Сначала всё множество фрагментов делится на несколько частей (максимально 8). Затем создается набор процессов, и список работ. Каждый процесс включает в себя выделенные части исходного множества и действие, которое необходимо с ним выполнить, т.е. «усушку». Далее все работы выполняются параллельно с помощью созданных процессов. В конечном итоге их результаты объединяется в одно единое множество.

Аналогичным образом с помощью набора процессов была распараллелена фаза «сканирования». Документ делится на части и каждая из них анализируется отдельно в своем потоке.

Формальное описание оптимизации представлено на листинге 1.

```
1  fit_candidates(pattern, similarity, candidates)
2  initialize_fit()
3  initialize_pool(max_processes = 8)
4  initialize_jobs()
5  for chunk in get_candidates_chunks(candidates, 8) do
6  |   result = do_async(process_chunk, (similarity, chunk))
7  |   append_to_job(result)
8  end
9  wait_for_job_finish()
10 close_pool()
11 return fit
12 get_candidates_chunks(candidates, number_of_processes)
13 chunk_size = get_size(candidates, number_of_processes)
14 start, end = 0, chunk_size
15 initialize_chunk_list()
16 while True do
17 |   add_chunk(start, end)
18 |   start, end = recount_boundaries()
19 |   stop_if_end()
20 end
21 return chunk_list
22 process_chunk(pattern, similarity, candidates_list)
23 initialize_fit()
24 for candidate in candidates_list do
25 |   append_to_fit(fit_candidate(similarity, candidate))
26 end
27 return fit
```

Листинг 1: Параллельная «усушка»

Опишем эту оптимизацию детально.

- Строки 2—4: инициализация искомого на фазе 2 множества фраг-

ментов, пула процессов (их максимально число равно 8) а также списка работ.

- Строки 5—8: параллельное выполнение работ с помощью пула потоков для каждой из частей множества.
- Строки 9—11: ожидание завершения всех работ, объединение результатов и возвращение результата фазы «усушки».
- Строки 13—15: инициализация списка, состоящего из частей исходного множества фрагментов и подсчет начальных границ этих частей.
- Строки 16—20: на данном этапе происходит заполнение списка и пересчет границ. При этом на каждой итерации проводится проверка на достижение конца исходного множества.
- Строки 23—26: для каждого кандидата происходит его «усушка».

3.3. Группировка пересекающихся образцов

На фазе «фильтрации» алгоритма поиска по образцу происходит фильтрация оставшихся после «усушки» фрагментов. Однако среди этих элементов есть некоторое число сильно пересекающиеся, которые в действительности располагаются в одном и том же месте в тексте. Поэтому алгоритм выдавал избыточные результаты.

В связи с этим была предложена следующая оптимизация. Рассматриваются группы пересекающихся фрагментов текста. В каждой из них находится отрезок, началом которого является начало самого левого фрагмента и концом — конец самого правого соответственно. Если его длина превышает максимально допустимую (удвоенная длина паттерна, умноженная на меру близости), то он делится пополам с учетом границ слов и для каждой части происходит «усушка». Тем самым к конечному множеству добавляется пара новых фрагментов вместо группы старых, что избавляет нас от избыточности.

Формальное описание представлено листинге 2.

```
1  join_overlapping_candidates(similarity, pattern, candidates)
2  fill_cluster_state(candidates)
3  something_changed = True
4  while something_changed do
5      something_changed = False
6      for pairs in cluster_state do
7          if intersects(zone1, zone2) then
8              candidates = sum(zone1, zone2)
9          end
10         begin, end = find_boundaries(zone1, zone2)
11         rewrite(cluster_state, begin, end)
12         something_changed = True
13     end
14     break
15 end
16 initialize(final_candidates)
17 for pairs in candidates do
18     begin, end = get_candidate_border()
19     if exceeded_max_length(begin, end) then
20         middle = get_word_right_space(document, (e - b) / 2)
21         if has_disjoint_elements(cluster_state[k], begin, end) then
22             left_cand= fit_candidate(begin, begin + middle)
23             right_cand = fit_candidate(begin + middle + 1, end)
24             fit_final_candidates(left_cand, right_cand)
25         end
26         else
27             add_one_final_candidates()
28         end
29         add_remaining_to_final_candidates()
30     end
31 end
32 return final_candidates
```

Листинг 2: Объединение пересекающихся повторов

Рассмотрим предлагаемую оптимизацию детальнее.

- Строка 2: инициализация ассоциативного массива cluster_state, в котором ключом является отрезок (то есть пара: начало и конец фрагмента текста), а в качестве значения — список найденных паттернов, содержащихся внутри этого отрезка.

- Строки 3—14: происходит заполнение массива cluster_state, описанного выше. итерация по cluster_state отвечает за вложенный

цикл, внутри которого рассматриваются все пары найденных фрагментов, В случае их пересечения (функция intersects) они перезаписываются в cluster_state

- Строки 16—25: происходит проход по всему множеству ключей(отрезков) в cluster_state. Если длина отрезка больше максимально допустимой (удвоенная длина паттерна, умноженная на меру близости), то проверяется значение функции has_disjoint_elements, которая определяет есть ли в соответствующем ключу списке фрагментов пересекающиеся. При положительном исходе рассчитывается середина отрезка с учётом границ слов, для правой и левой частей происходит «усушка» и добавление в искомое множество. В противном случае добавляется исходный фрагмент. При этом cluster_state очищается от уже просмотренных элементов

- Строки 26—32: final_candidates заполняется оставшимися фрагментами и возвращается окончательный результат алгоритма

3.4. Применение Cython

Данная оптимизация заключается в следующем. Было использовано расширение, с помощью которого код написанный на Python преобразуется в C-код для дальнейшей компиляции. Оно называется Cython. Достоинство данного способа состоит в том, что существующий код может выполняться практически со скоростью C добавлением некоторых статических объявлений типов и определенными поправками критических циклов.

Чтобы использовать Cython, понадобилось следующее.

1. Сохранить Python код с расширением .pyx, при этом добавив статическую типизацию к некоторым переменным для еще более быстрой работы.

2. Написать небольшой конфигурационный файл setup.py.

3. Сохранить Python код с расширением .pyx, при этом добавив аннотации для статической типизации к некоторым переменным для еще

более быстрой работы.

4. Написать небольшой конфигурационный файл setup.py. В нём необходимо указать название результирующего файла и имя файла, который будет скомпилирован в C-файл.

```
1 from distutils.core import setup
2 from Cython.Build import cythonize
3 setup(
4 name = 'Near duplicate finder',
5 ext_modules = cythonize("duplicates.pyx"))
6 return final_candidates
```

Листинг 3: Конфигурирование Cython

В результате получится готовый C-модуль, который можно в дальнейшем скомпилировать и импортировать в Python.

3.5. Оптимизация критических участков средствами Python

Рассмотрим оптимизации, выполняемые средствами языка Python с учётом свойств Cython. Они не затрагивают алгоритм и сводятся лишь к применению эффективных приёмов программирования.

Была проанализирована работа интерпретатора языка и выделены следующие особенности:

- цикл for выполняется быстрее while, это связано с тем, что в нём нет логических проверок;
- работа с глобальными переменными медленней работы с локальными;
- типы сравниваемых данных должны совпадать, иначе скорость падает;
- если функцию, вызывающуюся много раз подряд сначала сохранить в какой—либо переменной, а потом вызывать, то код становится более быстрым.

Все описанные пункты незначительно влияют на скорость работы, однако позволяют выиграть некоторое время. Поэтому они и были реализованы в рамках данной работы.

4. Реализация и интеграция

Алгоритм поиска неточных повторов по образцу был реализован на языке программирования Python. Он очень прост в освоении и предоставляет гибкие возможности для решения задач, связанных с обработкой текста.

В качестве модуля для распараллеливания алгоритма был выбран multiprocessing [26], который позволяет производить вычислениях одновременно на нескольких процессорах. Его важной особенностью является поддержка различных операционных систем, таких как Unix и Windows. Данный модуль также включает API, который не имеет аналогов в стандартной библиотеке языка Python threading. Ярким примером этого является класс Pool, который предлагает удобное средство для параллельного выполнения функций. Он поддерживает асинхронные результаты с тайм-аутами и обратными вызовами. Он и был использован в приведённых оптимизациях.

Cython — это компилятор Python [27]. Он работает следующим образом: из Cython кода генерируется C-код, который в свою очередь компилируется в обычный Python модуль. Его особенностью является то, что он может скомпилировать обычный код Python без изменений. Однако для критических точек кода часто бывает полезно добавлять объявления статического типа, поскольку они позволят генерировать более простой и быстрый C-код — иногда быстрее на порядки. Поэтому в рамках оптимизации с переходом на Cython была добавлена статическая типизация для функций и переменных (все типы C доступны для объявлений типов). Также благодаря Cython расспараллеливание стало более эффективным.

Для нахождения редакционного расстояния между строками применялся алгоритм из стандартной библиотеки difflib [28]. Его идея состоит в том, чтобы найти самую длинную общую подпоследовательность, которая не содержит «нежелательных» элементов. «Нежелательные» элементы - это такие, которые в некотором смысле неинтересны, например, пустые строки или пробелы. Стоит отметить, что применённая

оптимизация, связанная с использованием компилятора Cython, не повлияла на скорость вычисления данного расстояния, критические секции в difflib написаны на языке C.

Все описанные оптимизации были интегрированы в инструмент Duplicate Finder. На рис. 1 представлен интерактивный режим работы, в котором используется оптимизированный алгоритм.

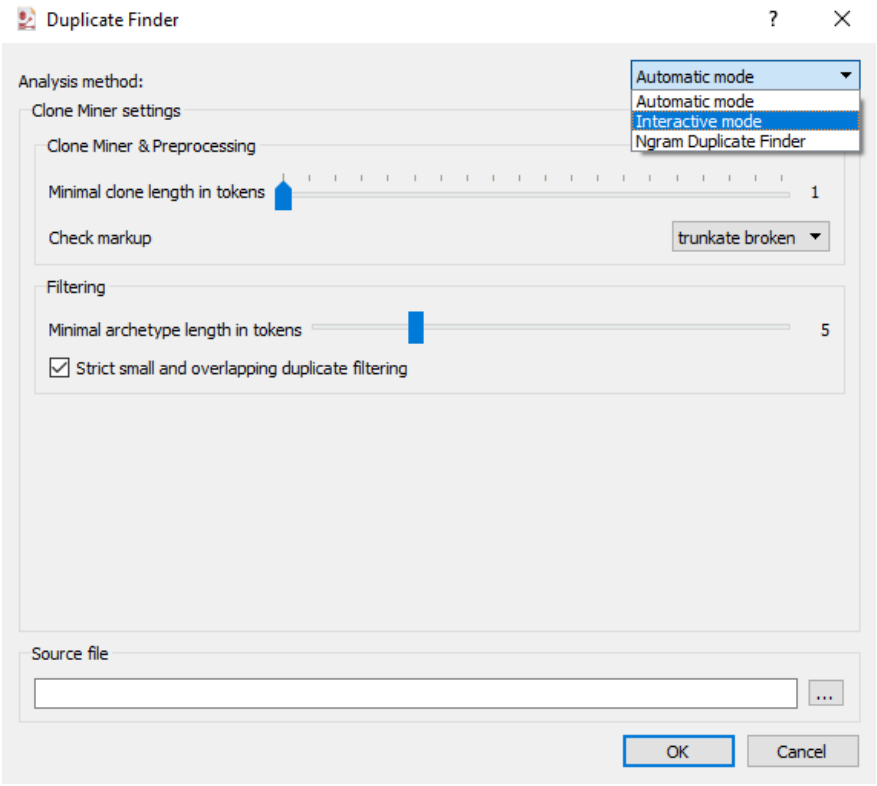

Рис. 1: Интерактивный режим

5. Обоснование полноты оптимизаций и эксперименты

Рассмотрим вопрос полноты оптимизаций. Оптимизации 1, 3, 4, приведённые в главе 3, не влияют на количество выдаваемых повторов, поскольку они улучшают только время работы за счёт распараллеливания и использования компилятора Cython. Оптимизация 2нарушает полноту лишь в незначительном числе случаев, когда несколько пересекающихся найденных повторов объединяются в один вместо того, чтобы оставить их отдельно (более подробно данный случай лписан в конце главы 2). Однако этот случай на практике встречается очень редко (как будет показано далее, число таких случаев составляет менее 1 процента)).

После реализации всех оптимизаций был проведён повторный эксперимент, идентичный описанному в главе 2. Результаты представлены на рис. 2 и 3.

Тестирование алгоритма было произведено для меры близости, равной 0,87 (это примерно 15%, так как, согласно, [29] вариативная часть обычно не превышает 15% от архетипа). Стоит заметить, что в проведённых экспериментах изменяемым параметром является длина образца поиска, а постоянный — мера близости. Это можно объяснить тем, что производительность исследуемого алгоритма значительно варьируется при изменении длины тестируемого образца (как можно заметить из рис. 3, время быстро возрастает). А так как мера близости может меняться лишь в малой степени, то это незначительно сказывается на времени работы алгоритма.

По результатам экспериментов можно выделить следующие особенности, касающиеся производительности алгоритма с оптимизациями.

1. При возрастании длины образца время работы алгоритма теперь растёт не так стремительно (как видно из рис.2, до оптимизации время среднее работы алгоритма выросло на 5 секунд, а после оптимизации на 1).

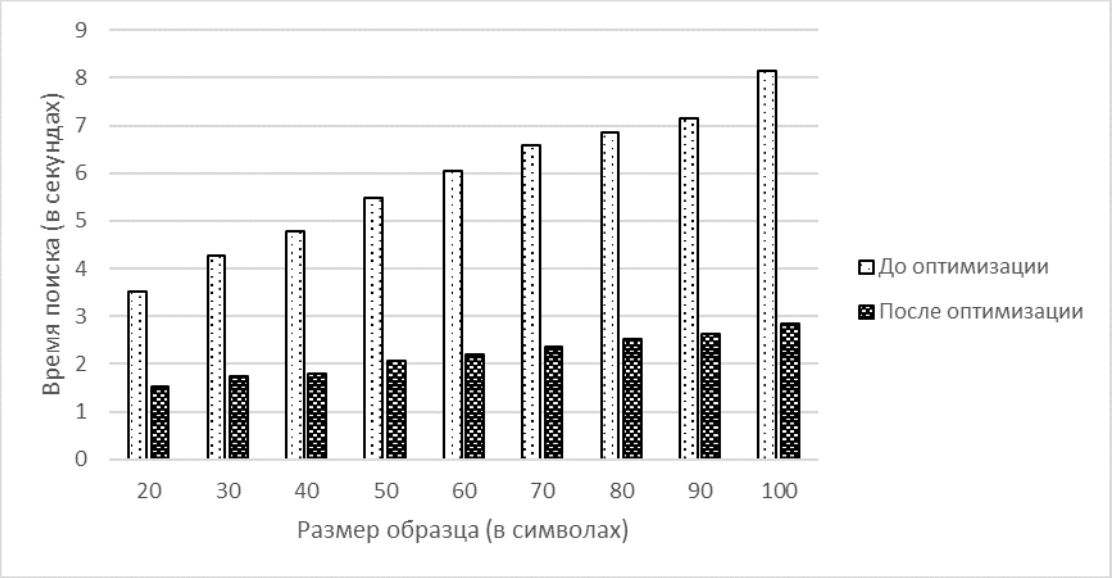

Рис. 2: Результаты оптимизации алгоритма

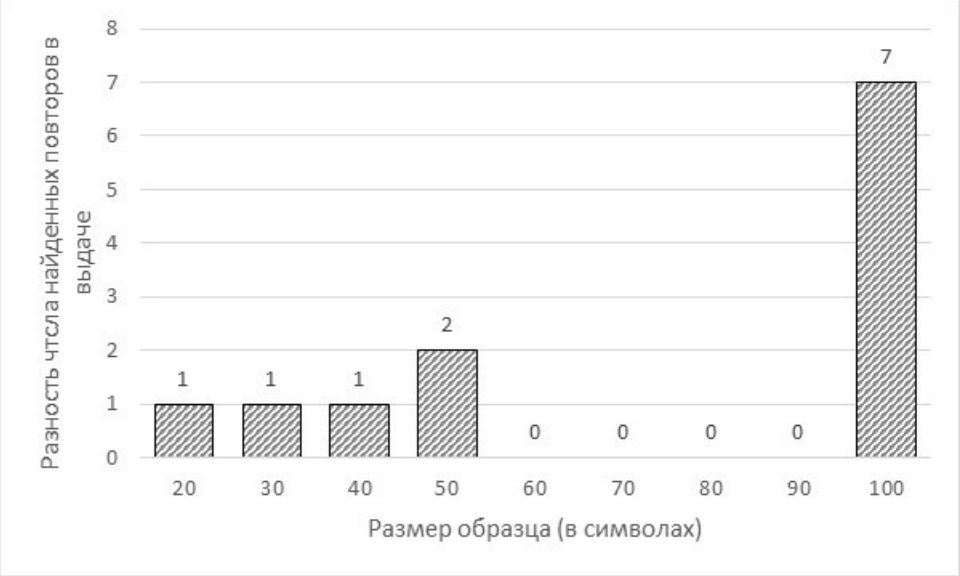

Рис. 3: Разность числа найденных групп

2. Для документации Blender (самый крупный документ) алгоритм на образце длины 100 символов стал работать 9 секунд по сравнению с 32 секундами для существующего алгоритма Это худший по производительности результат. Среднее же время работы оптимизированного алгоритма составило 2,2 секунды.

3. Чем больше было время работы для первоначального алгоритма, тем больший выигрыш по времени дали оптимизации (отношение

среднего времени работы алгоритма до и после оптимизаций возрастает ближке к правой части графика на рис. 1)

Также были проведены замеры на нескольких образцах большей длины (200-1000 символов). Для них оптимизации дали ещё больший выигрыш по времени. Например, для образца длиной 760 символов для документа LKD [8] время работы алгоритма сократилось с 52 до 16 секунд.

Что касается качества выдачи, то тут проведённая оптимизация №2 не внесла решающего вклада в алгоритм. На рис. 4 представлена разница между числом элементов в выдаче по всем 19 документам до и после этой оптимизации для каждого значения длины образца от 20 до 100 символов. Было выявлено, что суммарное число элементов в выдаче после оптимизации увеличилось всего лишь на 12 повторов (это составляет чуть более 0,1% от суммарной выдачи), что логично по причине малочисленности случаев, когда может нарушаться полнота алгоритма. Эти элементы были потеряны в выдаче алгоритма до оптимизации. Оптимизация в некоторых случаях ощутимо влияет на скорость работы. Поэтому было предложено сделать её опциональной для пользователя. В случае необходимости максимально точных результатов он может включить данную оптимизацию.

На основании выполненных экспериментов можно сделать вывод, что оптимизации улучшили скорость работы алгоритма примерно в 2,5 раза. Есть основания предполагать, что скорость можно ещё больше увеличить (особенно на крупных образцах и документах), если использовать 8-ядерный процессор (к сожалению, у автора рабоыт не было возможности это проверить). Можно заключить, что полученные результаты являются вполне удовлетворительными для применения на практике, потому что среднее время работы плгоритма поиска по образцу составляет всего 2,2 секунды.

Заключение

В ходе данной работы были получены следующие результаты.

1. Были проведены исследования существующего алгоритма на документации 19 открытых и коммерческих проектов с точки зрения качества и скорости работы и обнаружены недостатки алгоритма.

2. Выявлены и описаны возможные места алгоритма поиска по образцу, которые можно оптимизировать. Изучены технические средства реализации, такие как языки Python и Cython, библиотеки difflib и multiprocessing, инструмент Duplicate Finder.

3. Все предложенные оптимизации были реализованы и интегрированы в инструмент Duplicate Finder.

4. Выполнена апробация алгоритма поиска по образцу с оптимизациями.

   - Выполнено обоснование полноты алгоритма в соответствии с введённой моделью неточных повторов.
   - Выполнена вторая серия экспериментов, проведён анализ её результатов по сравнению с начальными измерениями. Установлено, что оптимизации улучшили скорость работы алгоритма примерно в 2,5 раза. Для некоторых случаев время работы уменьшилось значительно, например, с 32 секунд до 9 (Blender), с 14 до 4 (Zend).

5. Результаты данной дипломной работы вошли в статью «Интерактивная методика поиска неточных повторов в документации ПО», поданную в журнал «Программирование» (индексируется SCOPUS и Web Of Science).

Список литературы

[1] Koznov, D.V. Duplicate management in software documentation maintenance / D.V. Koznov, D.V. Luciv, G.A. Chernishev // CEUR Workshop Proceedings. — APSSE '17. — Proceedings of V International conference Actual problems of system and software engineering.

[2] Кантеев, Л.Д. Обнаружение неточно повторяющегося текста в документации программного обеспечения / Л.Д. Кантеев, Ю.О. Костюков, Д.В. Луцив, Д.В. Кознов, М.Н. Смирнов. — 2017. — Т. 29, № 2. — С. 303–314.

[3] Луцив, Д.В. Задача поиска нечетких повторов при организации повторного использования документации / Д.В. Луцив, Д.В. Кознов, Х.А. Басúт, А.Н. Терехов. — 2016. — Т. 4. — С. 39–49.

[4] Broder, A. On the Resemblance and Containment of Documents / A. Broder. — Proceedings of the Compression and Complexity of Sequences, 1997. — P. 21–29.

[5] Коршунов, А.В. Тематическое моделирование текстов на естественном языке / А.В. Коршунов, А.Г. Гомзин. — 2012. — Т. 23. — С. 215–242.

[6] Koznov, D.V. Detecting Near Duplicates in Software Documentation / D.V. Koznov, D.V. Luciv, G.A. Chernishev, A.N. Terekhov. — 2017.

[7] Луцив, Д.В. Поиск неточных повторов в документации программного обеспечения: Диссертация на соискание степени кандидата ф.-м. наук / СПБГУ. — 2018.

[8] Linux Kernel Documentation. — URL: `https://github.com/torvalds/linux/tree/master/kernel` (online; accessed: 20.01.2018).

[9] Терехов, А.Н. Технология программирования встроенных систем

реального времени: Диссертация на соискание степени доктора ф.-м. наук / ВЦ СО АН СССР. — 1991.

[10] Шалыто, А.А. Новая инициатива в программировании. Движение за открытую проектную документацию / А.А Шалыто. — 2003. — Т. 4. — С. 52–56.

[11] Липаев, В.В. Документирование сложных программных средств / В.В Липаев. — 2005.

[12] Sommerville, I. Software documentation. — URL: `http://www.literateprogramming.com/documentation.pdf` (online; accessed: 20.01.2018).

[13] Lethbridge, T.C. How software engineers use documentation: the state of the practice / T.C. Lethbridge, J. Singer, A. Forward. — 2003. — Vol. 26(1). — P. 35–39.

[14] Oumaziz, M.A. et al. Documentation Reuse: Hot or Not? An Empirical Study / M.A. et al. Oumaziz. — 2017. — P. 12–27.

[15] Wingkvist, A. Analysis and visualization of information quality of technical documentation / A. Wingkvist, W. Löwe, M. Ericsson, R. Lincke. — 2010. — P. 388–396.

[16] Juergens, E. Can clone detection support quality assessments of requirements specifications? / E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, Schaetz, S. Wagner, C. Domann, J. Streit. — 2010. — Vol. 2. — P. 79–88.

[17] Horie, M. Tool Support for Crosscutting Concerns of API Documentation / M. Horie, S. Chiba. — 2010. — P. 97–108.

[18] Nosál', M. Reusable software documentation with phrase annotations / M. Nosál', J. Porubän. — 2014. — Vol. 4(4). — P. 242–258.

[19] Juergens, E. CloneDetective — A workbench for clone detection research / E. Juergens, F. Deissenboeck, B. Hummel. — 2009. — P. 603–606.

[20] Mikolov, T. Distributed representations of words and phrases and their compositionality / T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, G.S. Dean. — 2013. — P. 3111–3119.

[21] Bird, S. NLTK: the natural language toolkit / S. Bird. — 2006. — P. 69–72.

[22] Турдаков, Д.Ю. Texterra: инфраструктура для анализа текстов / Д.Ю. Турдаков, Н.А. Астраханцев, Я.Р. Недумов, А.В. Коршунов. — 2014. — Т. 26(1). — С. 421–438.

[23] Choi, J.D. It Depends: Dependency Parser Comparison Using A Web-based Evaluation Tool / J.D. Choi, J.R. Tetreault, A. Stent. — 2015. — P. 387–396.

[24] Gusfield, D. Algorithms on Strings, Trees, and Sequences / D. Gusfield. — 1997. — P. 534.

[25] Bergroth, L. A survey of longest common subsequence algorithms / L. Bergroth, H. Hakonen, Raita T. — 2000. — P. 39–48.

[26] Python Multiprocessing module. — URL: `https://docs.python.org/3/library/multiprocessing.html` (online; accessed: 20.01.2018).

[27] Cython extension. — URL: `http://docs.cython.org/en/latest/src/tutorial/index.html` (online; accessed: 20.01.2018).

[28] Python DiffLib module. — URL: `https://docs.python.org/3/library/difflib.htm` (online; accessed: 20.01.2018).

[29] Bassett, P.G. The Theory and Practice of Adaptive Reuse / P.G Bassett. — 1997. — Vol. 22(3). — P. 2–9.