Санкт-Петербургский государственный университет

Программная инженерия
Системное программирование

Кантеев Леонид Дмитриевич

Поиск неточных повторов в программной документации

Курсовая работа

 Научный руководитель:

д.т.н., доцент Кознов Д. В.

Санкт-Петербург

2018

[**Введение**](#_1fob9te) **2**

[**Постановка задачи**](#_3znysh7) **3**

[**1. Обзор**](#_2et92p0) **4**

[1.1. Технология DocLine](#_tyjcwt) 4

[1.2. Обнаружение клонов в естественных языках](#_3dy6vkm) 5

[1.3. Подходы к поиску нечётких повторов](#_1t3h5sf) 5

[1.4. N-граммная индексация](#_7dr1kwktmj4u) 6

[1.5. Библиотека NLTK](#_nbwk3wrfqoeb) 6

[1.6. Утилита Pandoc](#_m93jjcd7qnud) 6

[1.7. Библиотека Pandoc-filters](#_7xgneu9k4y2k) 6

[1.8. Существующий алгоритм на N-граммах](#_3rdcrjn) 7

[**2. Метод поиска неточных повторов**](#_26in1rg) **8**

[2.1. Предлагаемые улучшения](#_lnxbz9) 8

[2.2. Общее описание алгоритма](#_35nkun2) 8

[2.2.1. Утилита Pandoc](#_x294ys1hcvvh) 9

[2.2.2. Библиотека Pandoc-filters](#_n3wu8qlwe2qj) 9

[2.2.3. Библиотека NLTK](#_r17wnpnjkbda) 10

[2.3. Поиск повторов](#_1ksv4uv) 10

[2.4. Особенности реализации](#_44sinio) 12

[**3. Эксперименты**](#_2jxsxqh) **13**

[3.1. Поиск неточных повторов](#_z337ya) 13

[3.2. Производительность](#_3j2qqm3) 14

[**Заключение**](#_1y810tw) **15**

[**Список литературы**](#_2xcytpi) **16**

#

# Введение

В современных программных проектах создаётся значительное количество текстовой документации, которая нуждается в сопровождении и поддержке. Одной из подзадач является поиск дублированного текста: ведь при разработке программной документации часто используется приём copy/paste, и результаты его применения не отслеживается должным образом [[2]](#3whwml4), [[3]](#2bn6wsx), [[4]](#qsh70q).

Существуют различные виды программной документации [[1]](#1ci93xb); для некоторых из них дублированный текст нежелателен, для других дублирование необходимо, но в любом случае дубликаты увеличивают сложность документации и усложняют процесс её поддержки. Ситуация усугубляется “неточными повторами” – дублированным текстом, фрагменты которого незначительно отличается друг от друга в зависимости от детализации, контекста и т. д. И если не отслеживать наличие повторов в документации, то снижается её качество. Поэтому важной задачей становится упрощение и автоматизация процесса поиска и рефакторинга таких повторов.

Наиболее популярным методом обнаружения дубликатов в программной документации является техника software clone detection [[5]](#3as4poj). Однако, этот подход обнаруживает только точные дубликаты. Существуют решения, использующие эту технику и осуществляющие с её помощью поиск неточных повторов, такие как DocLine [[6]](#1pxezwc), [[7]](#49x2ik5). Но данный способ имеет ограничения, например, большое количество ложных срабатываний.

Существует множество способов определять похожесть текстов. Одним из таких способов является сравнение N-грам [[9]](#147n2zr). В [8] был представлен алгоритм, основанный на N-граммах для поиска неточных повторов в документации ПО. Однако у него имеются существенные ограничения: работа с повторами, совпадающими с один предложением, большое количество ложных срабатываний и проблемы с производительностью.

# Постановка задачи

Целью данной курсовой работы является доработка алгоритма для поиска нечётких повторов на основе N-грам текстов. Для достижения этой цели в рамках работы были сформулированы следующие задачи.

1. Изучить существующие алгоритмы поиска нечётких повторов.
2. Улучшить существующий новый алгоритм нечёткого поиска на основе N-грамм.
3. Провести эксперименты и тестирование алгоритма.
4. Интегрировать алгоритм в Duplicate Finder.

#

# 1. Обзор

Проблема дублирования текста в программной документации активно изучается на данный момент. E.Juergens [[2]](#3whwml4) с коллегами анализируют избыточность в документах с требованиями. M.Horie [[14]](#1hmsyys) рассматривает проблему фрагментации дубликатов в Java API-документации. A.Wingkvist с коллегами [[15]](#41mghml) пишут в контексте управления процессом сопровождения документацией программных проектов. Однако проблема обнаружения неточных повторов остается открытой.

Ещё в конце 90-х годов P.Bassett ввёл понятие архетипа (общая часть повторяющегося контента) и дельты (вариативная часть) [[16]](#2grqrue). С помощью этой концепции S.Jarzabek разработал метод повторного использования программного обеспечения на основе XML [[17]](#vx1227). Д.Кознов и К.Романовский [[7]](#49x2ik5) применили идеи P.Bassett и S.Jarzabek к программному обеспечению для повторного использования документации, включая рефакторинг документации. Однако, эти исследования не разрешили проблему обнаружения неточных повторов в документах.

## 1.1. Технология DocLine

DocLine – это проект, посвящённый разработке документации семейств программных продуктов с возможностью вариативного использования повторяющихся частей [[7]](#49x2ik5). Для решения задачи преобразования документации в DocLine используется инструмент Documentation Refactoring Toolkit [[18]](#3fwokq0), имеющий графический интерфейс и позволяющий выбирать необходимый для редактирования файл, задавать определённые параметры и находить нечёткие повторы в этом файле.

Имеется также ещё одни инструмент под названием Duplicate Finder, предназначенный для поиска неточных повторов в документации ПО. Создано два алгоритма неточных повторов: алгоритм компоновки точных повторов и алгоритм поиска по образцу. Алгоритм компоновки комбинирует точные клоны, найденные инструментом CloneMiner (реализует software clone detection). Он обладает существенными проблемами, выдавая большое количество ложно-положительных срабатываний и даже в том случае, когда он находит неточные повторы правильно, он часто нарушает их границы и группирует их не оптимальным способом. Алгоритм поиска по образцу находит повторы, основанные на предложенном образце. Оба алгоритма показали неудовлетворительное качество найденных повторов на тестовых примерах, что говорит о необходимости создания нового решения.

## 1.2. Обнаружение клонов в естественных языках

Существуют различные средства для обнаружения повторов в естественных языках [[19]](#1v1yuxt), [[20]](#4f1mdlm). Однако в большинстве случаев они представляют собой инструменты для стилистического анализа, а не для нахождения всех нечётких повторов. В основном, с помощью поиска четких дубликатов они выдают информацию об использовании конкретных слов, словосочетаний, речевых оборотов или предложений, и выдают отчёт по количеству найденных совпадений.

## 1.3. Подходы к поиску нечётких повторов

Одним из самых важных подходов к нечёткому поиску является поиск похожих строк. Выбор подходящей функции сравнения текстов непосредственно влияет на качество и скорость поиска. Одной из самых популярный мер близости текстов является функция вычисления расстояния Левенштейна [[21]](#2u6wntf), или редакционного расстояния. Расстояние Левенштейна равняется минимальному количеству операций редактирования, позволяющих преобразовать одну строку в другую. Алгоритмы вычисления редакционного расстояния [[22]](#19c6y18) изучаются уже давно и являются основой многих других алгоритмов нечёткого поиска. Однако, существует множество других подходов, таких как хеширование по сигнатуре [[23]](#3tbugp1), n-граммная индексация [[24]](#28h4qwu), kd-деревья и rd-деревья [[25]](#nmf14n), алгоритмы поиска в метрических пространствах [[26]](#37m2jsg), декартовы деревья [[27]](#1mrcu09).

## 1.4. N-граммная индексация

N-грамма - это последовательность из n подряд идущих слов в тексте. Разбив текст на N-граммы можно достаточно быстро определять похожесть его частей. N-граммы успешно применяются для выявления плагиата, категоризации текста. Кроме того, их можно использовать для создания функций, которые позволяют получать знания из текстовых данных. Используя N-граммы, можно эффективно найти кандидатов, чтобы заменить слова с ошибками правописания.

## 1.5. Библиотека NLTK

 Библиотека NLTK предоставляет большие возможности в анализе и обработке текстов на натуральных языках. С её помощью можно делать стемминг слов - сокращение исходного слова до его основы, которая довольно часто не является самостоятельным значимым словом, выделять N-граммы, убирать пунктуацию и делить текст на предложения.

## 1.6. Утилита Pandoc

Pandoc — универсальная утилита для работы с текстовыми форматами. Основная сфера применения — форматирование математических и технических текстов. Также она способна изменять формат текстовых файлов без потери содержимого.

## 1.7. Библиотека Pandoc-filters

Pandoc-filters - это библиотека, позволяющая работать с деревом синтаксического разбора текста. С её помощью можно анализировать разметку текста, удалять или изменять нужные фрагменты текста.

## 1.8. Существующий алгоритм на N-граммах

Ранее была опубликована статья, в которой было произведено тестирование ранней версии алгоритма  [[8]](#2p2csry). Алгоритм позволяет искать неточные повторы одиночных предложений в программной документации. В нем используется N-граммная модель сравнения предложений. Однако у алгоритма есть существенные недостатки – множество ложно-положительных срабатываний и медленное время работы.

# 2. Поиск неточных повторов на основе N-грамм

## 2.1. Предлагаемые улучшения существующего алгоритма

Самой главное проблемой ранней версии алгоритма является то, что он мог находить повторы лишь в одиночных предложениях. Поэтому основным улучшением алгоритма должен стать на поиск по нескольким предложениям. Предлагается сделать это за счет расширения уже найденных групп на новые предложения. С множеством ложно-положительных групп в результатах алгоритма предлагается бороться усилением критерия попадания предложения в группу. Также нужно изменить порядок обхода предложений для уменьшения времени работы алгоритма.

## 2.2. Общее описание алгоритма

На [рис.1](#ljxnzk4dyhw2) представлена схема работы нашего подхода. Исходный текст конвертируется с помощью утилиты pandoc в дерево синтаксического разбора текста. Далее, с помощью pandoc-filters обрабатывается дерево синтаксического разбора и на выходе получается текст, в котором убраны мешающие работе алгоритма части. Далее текст анализируется с помощью библиотеки NLTK, которая выдает структуры данных с информацией для алгоритма. Это подается на вход алгоритму, который выдает готовые к просмотру группы повторов.



Рис.1 Алгоритм

### 2.2.1. Утилита Pandoc

Первой задачей при поиске повторов является преобразование исходного документа в дерево синтаксического разбора. Это позволяет более точно понять в каких частях текста нужно искать повторы, а в каких нет.

### 2.2.2. Библиотека Pandoc-filters

Получив дерево синтаксического разбора, мы можем убрать ненужные для поиска фрагменты текста и отформатировать его так, чтобы было удобно с ним работать. В нашем случае применяются следующие правила обработки дерева синтаксического разбора:

* убрать заголовки;
* убрать блоки с кодом;
* убрать ссылки;
* все списки превратить в параграфы с текстом;
* убрать все выделения в тексте;
* убрать весь html из текста.

### 2.2.3. Библиотека NLTK

Было решено использовать библиотеку NLTK для парсинга текста и построения нужных алгоритму структур данных. Библиотека выполняет следующие действия.

1. Поделить на предложения.
2. Убрать пунктуацию.
3. Убрать не несущие смысла слова.
4. Обрезать слова стеммером.
5. Выделить в предложениях 3-граммы.
6. Создать структуры данных, нужные для поиска неточных повторов.

## 2.3. Поиск повторов

Алгоритм идет по всем предложениям и пытается построить группу основываясь на критерии и, если это получилось сделать, пытается расширить получившуюся группу путем добавления к каждому фрагменту следующего за ним предложения в тексте и повторной проверки критерия. Все предложения, которые после этой операции находятся в группе, больше не используются в алгоритме, а группа добавляется к ответу. Таким образом, в результате работы алгоритма получается набор групп, которые и являются группами неточных повторов этого текста.



**Листинг 1. Поиск повторов**

Опишем использованные  в [листинге 1](https://docs.google.com/document/d/1uBL1p2eha6uXNrD9VKpmY1g_NRWarq_BAnkxiQ2tpt4/edit#bookmark=id.aakaxr4eovz0) обозначения.

* **𝑓𝑟**∈**𝑆𝐷** — фрагменты текста такие, что каждый из них содержит одно или несколько целых предложений.
* **𝑔𝑟** — множество фрагментов текста, например,  **𝑔𝑟1=**∅ или **𝑔𝑟2={𝑓𝑟1,𝑓𝑟2}**.

Опишем использованные  в листинге 1 функции и процедуры.

* 𝐼𝑛𝑣𝑜𝑙𝑣𝑒𝑑  — возвращает истину, если фрагмент или его часть входит в какую-то группу.
* 𝑁𝑔𝑟𝑎𝑚 — возвращает множество N-грам фрагмента.
* 𝑂𝑣𝑒𝑟𝑙𝑎𝑝 — возвращает коэффициент пересечения 2 фрагментов по N-граммам.
* 𝑆𝑢𝑖𝑡𝑎𝑏𝑙𝑒 — возвращает истину, если фрагмент 𝑓𝑟 подходит для включения в группу 𝑔𝑟: либо группа пустая, либо коэффициент пересечения фрагмента со всеми фрагментами группы больше 0.5. Это так называемый “критерий попадания фрагмента в группу”.
* 𝐸𝑥𝑝𝑎𝑛𝑑 — возвращает фрагмент  𝑓𝑟′, включающий все предложения  𝑓𝑟 + одно, непосредственно следующее за  𝑓𝑟.
* 𝐴𝑝𝑝𝑒𝑛𝑑 — добавляет фрагмент 𝑓𝑟 к группе 𝑔𝑟.
* 𝐴𝑝𝑝𝑒𝑛𝑑  — добавляет группу 𝑔𝑟 к множеству 𝑔𝑟𝑝𝑠.

Теперь опишем детально представленный в листинге 1 алгоритм.

* **Строки 3-22** – основной цикл алгоритма, который ищет группы неточных повторов.
* **Строки 8-10 –** цикл, в котором формируется новая группа для расширения.
* **Строки 11-21 –** цикл, в котором расширяется новая группа
* **Строки 13-17 –** цикл, в котором расширяется фрагмент, и если выполняется критерий, то расширенный фрагмент добавляется в группу
* **Строки 18-21 –** если в группе осталось больше 2 фрагментов, то можно продолжить фазу расширения, иначе её надо остановить
* **Строка 22 –**  добавить получившуюся расширенную группу в ответ.
* **Строки 23 –**  исключить из ответа группы, в которых содержится по 1 фрагменту.

## 2.4. Особенности реализации

Алгоритм реализован на языке python, поскольку он позволяет использовать множество библиотек, в том числе NLTK. Также при использовании алгоритма можно варьировать минимальный процент пересечения фрагментов для попадания в группу и изменять язык документа (в пределах языков, реализованных в библиотеки NLTK).

# 3. Эксперименты

 Было решено проводить апробацию алгоритма на тех же документах, что и для предыдущей версии алгоритма [[8]](#2p2csry) для того, чтобы сравнить результаты.

Тестирование алгоритма проводилось на англоязычной документации 4 открытых проектов, а также на русскоязычном документе одного закрытого коммерческой проекта:

* Linux Kernel documentation (LKD), 892 кб [10];
* Zend Framework documentation (Zend), 2924 кб [11];
* DocBook 4 Definitive Guide (DocBook), 686 кб [12];
* Version Control with Subversion (SVN), 1810 кб [13];
* Commercial project user guide (CProj), 164 кб.

## 3.1. Поиск неточных повторов

Ниже представлена [таблица 1](#l68d416yzxut), в которой подсчитан процент переиспользования – это суммарное количество символов в неточных повторах, разделённое на количество символов во всём тексте. Также, с помощью ручного разбора был выявлен процент нерелевантных повторов – повторов без семантического смысла.



Таблица 1.

Для документов разного размера количество групп повторов варьируется от нескольких десятков до нескольких сотен в зависимости от размера и характера документа, поэтому можно сказать, что алгоритм обнаруживает значительное количество близких дубликатов, и большинство из них имеют смысл.

Если сравнивать с результатами, которые представлены в статье [[8]](#2p2csry), то можно заметить, что процент переиспользования почти не изменился, процент нерелевантных повторов упал, процент осмысленного переиспользования вырос, а вот процент релевантных повторов значительно вырос. Это связано в первую очередь с тем, что алгоритму больше не поступает на вход разметка. Также существенную часть увеличения качества взяло на себя улучшение критерия вхождения в группу повторов.

## 3.2. Производительность

Тестирование производительности алгоритма проводилось на тех же документах и на компьютере с Intel i5-2400, 3.10GHz, ОЗУ 4 Гб, Windows 10.



Таблица 2.

На т[аблице 2](#397epjw48so1) можно заметить, что время работы хорошо коррелирует с размером документов, а также с количеством повторов в них - чем меньше процент переиспользования, тем дольше работает алгоритм. Это связано с тем, что если предложение быстро вошло в какую-либо группу, то оно больше не будет обрабатываться алгоритмом, а в ином случае может просматриваться множество раз. Таким образом, чем больше повторов в тексте, тем быстрее будет работать алгоритм.

Если сравнивать с проверкой производительности в статье [[8]](#2p2csry), то заметно сильное увеличение скорости работы. Выигрыш в производительности обеспечен улучшением алгоритма и переписыванием основного цикла.

# Заключение

В рамках данной работы были достигнуты следующие результаты.

1. Изучены средства нечёткого поиска, научные работы и алгоритмы в области нечёткого поиска.
2. Улучшен алгоритм поиска нечетких повторов на основе модели n-грамм:
	* алгоритм расширен на поиск повторов, состоящих не из одного предложения;
	* добавлен анализ разметки
3. Проведены эксперименты и тестирование алгоритма:
	* увеличение процента осмысленных повторов;
	* уменьшение процента нерелевантных повторов;
	* улучшение производительности.

В связи с однозначным улучшением качества повторов после добавления анализа разметки, дальнейшая работа над алгоритмом будет производится в сторону улучшения качества анализа разметки.

# Список литературы

[1] Parnas D. L. Precise Documentation: The Key To Better Software. Nanz S. (ed.) The Future of Software Engineering, Springer, 2011.

[2] Juergens E. et al. Can clone detection support quality assessments of requirements specifications? Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering, 2010, vol. 2, pp. 79–88.

[3] Horie M., Chiba S. Tool support for crosscutting concerns of API documentation. Proceedings of 9th International Conference on Aspect-Oriented Software Development, 2010, pp. 97–108.

[4] Rago A., Marcos C., Diaz-Pace J.A. Identifying duplicate functionality in textual use cases by aligning semantic actions. International Journal on Software and Systems Modeling, vol. 15, issue 2, 2016, pp. 579–603.

[5] Akhin, M., Itsykson, V. Clone Detection: Why, What and How? Proceedings of CEESECR’10, 2010, pp. 36–42.

[6] Lutsiv D.V., Koznov D.V., Basit H.A., Lieh O.E., Smirnov M.N., Romanovsky K.Yu. An approach for clone detection in documentation reuse. Nauchno-tehnicheskij vestnik informacionnyh tehnologij, mehaniki i optiki [Scientific and Technical Journal of Information Technologies, Mechanics and Optics] vol. 92, issue 4, 2014, pp. 106–114 (in Russian).

[7] Романовский К.Ю., Кознов Д.В. DocLine: метод разработки документации семейств программных продуктов. Программирование №4, Санкт-Петербург, Старый Петергоф, Университетский пр., 28, 2008.

[8] Kanteev L.D., Kostyukov Yu.O., Luciv D.V., Koznov D.V., Smirnov M.N. Discovering Near Duplicate Text in Software Documentation. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 303-314.

[9] Wagner S., Fernández D.M. Analysing Text in Software Projects. Preprint, 2016. URL: https://arxiv.org/abs/1612.00164

[10] Linux Kernel Documentation, snapshot on Dec 11, 2013. URL: https://github.com/torvalds/linux/tree/master/Documentation/DocBook/

[11] Zend PHP Framework documentation, snapshot on Apr 24, 2015. URL: https://github.com/zendframework/zf1/tree/master/documentation

[12] DocBook Definitive Guide, snapshot on Apr 24, 2015. URL: http://sourceforge.net/p/docbook/code/HEAD/tree/trunk/defguide/en/

[13] SVN Book, snapshot on Apr 24, 2015. URL: http://sourceforge.net/p/svnbook/source/HEAD/tree/trunk/en/book/

[14] Horie M., Chiba S. Tool support for crosscutting concerns of API documentation. Proceedings of 9th International Conference on Aspect-Oriented Software Development, 2010, pp. 97–108.

[15] Wingkvist A., Ericsson M., Lincke R., Löwe W. A Metrics-Based Approach to Technical Documentation Quality. Proceedings of 7th International Conference on the Quality of Information and Communications Technology, 2010, pp. 476–481.

[16] Bassett P. Framing software reuse – lessons from real world. Prentice Hall, 1996.

[17] Jarzabek S., Bassett P., Zhang H., Zhang W. XVCL: XML-based Variant Configuration Language. Proceedings of 25th International Conference on Software Engineering, 2003, pp. 810–811.

[18] Луцив Д.В. Documentation refactoring toolkit, 2015. <https://goo.gl/vpEBj2>.

[19] Prowritingaid, 2016. <https://prowritingaid.com/>.

[20] Repetition detector 2, 2015. <http://www.repetition-detector.com/>

[21] Wikipedia. Levenshtein distance. Wikipedia, free encyclopedia, 2016. https://en.wikipedia.org/wiki/Levenshtein\_distance.

[22] Paterson M. Masek W. A faster algorithm computing string edit distances. Journal of computer and system sciences, 1980.

[23] Л. М. Бойцов. Поиск по сходству в документальных базах данных: хеширование по сигнатуре — оптимальное соотношение скорости поиска, простоты реализации и объема индексного файла. Программист. № 1, 2001.

[24] Ukkonen E. Approximate string-matching with q-grams and maximal matches. Theoretical Computer Science 92, 1992.

[25] A. Pfeffer J.M. Hellerstein. The RD-tree: an index structure for sets. Technical Report of University of Wisconsin.

[26] P.N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, 1993.

[27] T.H. Merrett H. Shang. Tries for approximate string matching. IEEE Transactions on Knowledge and Data Engineering, 1995.