Санкт-Петербургский государственный университет

Программная инженерия
Системное программирование

Глазырин Антон Георгиевич

Универсальный механизм первичного поиска повторов в тексте для пакета Duplicate Finder

Отчет по производственной (преддипломной) практике

Научный руководитель:
доц. каф. СП, к.ф.-м.н. Луцив Д. В.

Санкт-Петербург
2023

Оглавление

Введение

Документация является неотъемлемой частью большинства крупных проектов. При этом с ростом и развитием продукта она становится все объемнее и запутаннее: на первых этапах жизненного цикла проекта документация может отсутствовать совсем, однако чем дольше проект находится в фазе реализации, тем больше функционала и особенностей нужно документировать. Неудивительно, что со временем на поддержку документации будет тратится все больше сил и времени, что затрудняет сопровождение проекта.

В целом документацию можно разделить на два типа: описательную и справочную [25]. Описательная документация нужна для общего ознакомления пользователя с продуктом: она не содержит много деталей работы продукта или технических подробностей, и в основном представлена естественным языком. Справочная, наоборот, ориентирована на пользователей, которые уже разбираются в предметной области, и просто хотят найти какую-либо специфичную информацию. Поэтому, справочная документация обычно является сводкой существующего функционала: список методов, интерфейсов и т.п., и часто представлена исходным кодом. На практике, в одном документе могут быть фрагменты обоих типов, то есть документация является сочетанием естественного языка и исходного кода.

Одним из наиболее влиятельных факторов усложнения ведения документации является наличие большого количества повторов. Они не только могут сильно раздувать общий объем, но также из-за них усложняется сохранение целостности — одно и то же изменения приходится вносить несколько раз в разные части документации, что легко может привести к ошибкам и пропускам. Исследования также показывают, что повторы могут существовать практически в любом документе, и их наличие не зависит от конкретики продукта [9, 26]. Кроме того, в этих работах подтверждается, что повторы действительно оказывают негативное влияние на общее качество документации.

Сам по себе поиск повторов практически всегда является некото-

рой подзадачей для достижения более осмысленной цели, поэтому он применяется в совершенно различных областях [1]. В сфере разработки ПО наибольшее внимание уделяется поиску клонов в исходном коде [31]. Инструменты, основанные на этом подходе, предоставляют широкий спектр возможностей, от помощи с рефакторингом, до анализа недостатков глобальной архитектуры.

Однако, что касается более частной задачи поиска повторов в документации, данной теме уделяется не слишком много внимания, а если она и затрагивается, то лишь поверхностно. Существующие инструменты для работы с исходным кодом полагаются на достаточно специфичные особенности, такие как жесткая структура программы, семантика языков и т.п. [11, 37, 20]. Такие методы плохо подходят для работы с документацией из-за того, что она содержит не только фрагменты исходного кода, но и фрагменты на естественном языке.

Для решения задачи поиска повторов в документации был создан исследовательский прототип — Dupilcate Finder Toolkit [38], основной целью которого является работа с документацией. Данный инструмент представляет собой университетскую разработку, которая делалась многими людьми на протяжении большого временного промежутка, из-за чего проект состоит из множества составных частей, что сильно усложняет поддержку и дальнейшее развитие.

Особенно плохо дела обстоят как раз с компонентами, непосредственно отвечающими за поиск повторов [6, 39, 36]: они написаны на разных языках, некоторые из них являются внешними закрытыми разработками, они не разделяют потоков данных, повторяют некоторые этапы анализа несколько раз и в целом работают изолированно друг от друга. Для улучшения работы с пакетом было бы разумно разработать единую компоненту, целиком реализующую весь необходимый функционал.

1. Постановка задачи

Целью данной работы является разработка и реализация унифицированной подсистемы поиска точных и неточных повторов для Duplicate Finder Toolkit и ее интеграция с заменой существующих компонент.

1. Анализ предметной области — поиск повторов в текстах различных видов.
2. Определение проблем поиска повторов в DuplicateFinder и требований к новому механизму.
3. Проектирование конвейера механизма поиска повторов.
4. Разработка алгоритмов точного и неточного поиска.
5. Реализация инструмента и его интеграция в DuplicateFinder.
6. Проведение тестирования разработанного инструмента.

2. Обзор предметной области

2.1. Поиск повтров в различных сферах

Поиск повторов применяется во многих областях для достижения разнообразных целей [1]. При этом непосредственно сам поиск практически никогда не является конечной целью. Найденные повторы могут быть использованы, например, как входные данные для различных инструментов, или для проведения некоторого анализа. Благодаря тому, что задача поиска повторов является достаточно самостоятельной и обособленной, существует множество инструментов для ее решения, часто заточенных под свою конкретную область применения.

Неточное сравнение широко распространено в задачах, где необходимо вычислять схожесть текстов. Например, поиск неточных повторов лежит в основе большинства инструментов проверки на плагиат. В работе [15] рассматривается метод вычисления схожести двух документов путем сравнения текстовых фрагментов. Для сравнения используется подход min-hash [23]. Авторы инструмента [34] продолжили эту тему в своей работе. Вместо непосредственного сравнения фрагментов они предложили группировать их при помощи подхода bottom-k sketches [12]. Данный метод позволяет не только определять схожие фрагменты в двух документах, но и оптимально выбирать документ, с которым будет производиться сравнение. Еще одним распространенным подходом, основанном на хешировании, является sim-hash [10]. Он позволяет получить "отпечатки" (fingerprints) текстов и использовать их для быстрого получения приблизительной оценки схожести. Такой метод помогает оптимизировать обработку больших объемов данных, как, например, показано в работе [18].

Достаточно широко распространены средства для поиска повторов в программном коде, так как обнаружение дублированных участков кода помогает избежать ряда проблем и открывает возможности для улучшения системы [31, 30]. Кроме того, наличие большого количество работ по этой теме можно объяснить и тем, что согласно исследованиям

[3, 4] около 5–10% кода в больших проектах является дублированным, что составляет достаточно значительную часть. Одним из основных подходов для обнаружения дубликатов является анализ синтаксического дерева программы. Такие инструменты как [37, 11] позволяют находить повторы при помощи сравнивания частей AST, являющихся похожими структурными единицами. Еще одним широко используемым средством являются суффиксные деревья [16]. Так как использование суффиксного дерева подразумевает работу уже с простым текстом, этот подход можно применять одновременно с использованием AST, например, как описано в [20].

Поиск повторов также имеет применения на этапе сопровождения ПО. В различных работах поднимается проблема большого количества одинаковых или схожих отчетов об ошибках. Особенно острой эта проблема является для крупных IT компаний. В работе [32] описывается фреймворк для нахождения похожих отчетов компании BlackBerry, который использует популярную библиотеку Apache Lucene [2] для неточного поиска. Авторы инструмента [5] используют подход на основе вычисления наибольшей общей подстроки из токенов [17] для обнаружения схожих отчетов в репозитории ошибок Firefox.

Различные подходы для определения схожести фрагментов теста нередко применяются в NLP[1] [24]. Особый интерес вызывает модель N-грамм [8] — представление текста в виде множества кортежей из последовательных элементов, в качестве которых обычно берут слова. Сравнивая такие множества у двух различных фрагментов можно судить о степени их сходства. Также при работе с естественным языком очень популярным средством являются нейросети [7]. Они позволяют определять смысловую нагрузку текста вне зависимости от синтаксической составляющей, таким образом можно сравнивать семантическую схожесть двух фрагментов, такой подход рассматривается в работе [27]. Частным случаем нейронных сетей являются специальные модели, такие как [14], которые позволяют преобразовать слова в векторное представление, отображающее семантические особенности.

[1]Natural Language Processing — обработка естественного языка

2.2. Поиск повторов в документации

Несмотря на широкое распространение и применение методов поиска повторов, задаче поиска повторов в документации уделяется мало внимания. Существует ряд исследований [9, 26], которые показывают, что в среднем около 10–15% документации составляют дублированные фрагменты, и что из-за этого могут возникать проблемы, схожие с дублированием в исходном коде — увеличение размера документации, усложнение сопровождения и т.п. Однако, хотя эти и другие работы направлены на изучение повторов в документации, они не описывают конкретные подходы для ее улучшения, которые можно применить на практике.

Для решения этой задачи в диссертации [38] был разработан подход к улучшению документации на основе поиска повторов, и реализован соответствующий инструмент — Duplicate Finder Toolkit. Для поиска точных повторов в нем используется средство для поиска клонов в ПО CloneMiner [6], затем путем комбинирования этих точных повторов определяются неточные повторы. Однако, основной целью работы являлись разработка и алгоритма компоновки неточных повторов подхода по улучшению документации, поэтому для первоначального поиска повторов использовались уже существующие инструменты.

С течением времени проект расширялся дополнительными модулями, в том числе и для поиска повторов. В работе [39] описывается метод поиска неточных повторов: текст равномерно разбивается на фрагменты одинакового размера, которые затем сравниваются между собой. Для сравнения у фрагментов вычисляются хеши и редакционные расстояния. Затем схожие фрагменты объединяются в группы повторов. Другой подход к поиску неточных повторов используется в работе [13]: текст разбивается на предложения, и для них вычисляются множества N-грамм. Затем предложения, имеющие схожие множества объединяются в группы. Алгоритм объединения был далее усовершенствован в работе [36].

3. Требования к механизму поиска

В данной главе приводится анализ актуальных проблем поиска повторов в Duplicate Finder и определяются основные требования, которым должен соответствовать новый механизм.

3.1. Анализ Duplicate Finder

Прежде всего Duplicate Finder — это инструмент для улучшения документации. Он поддерживает различные сценарии работы и процесс обработки документа включает в себя множество этапов, одним из которых является поиск повторов. Схема работы Duplicate Finder приведена на рисунке 1.

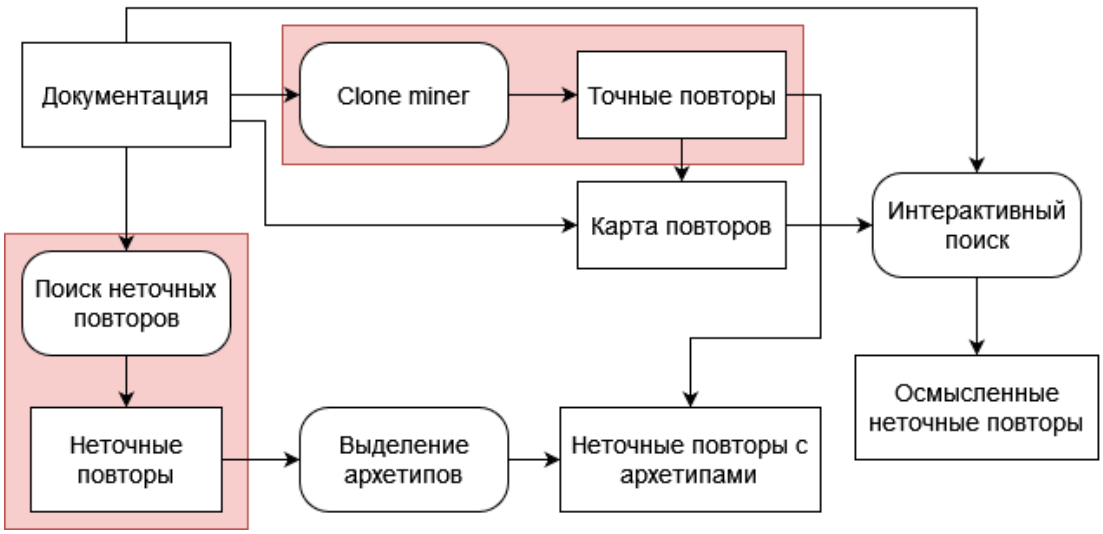

Рис. 1: Схема работы Duplicate Finder. Красным отмечены компоненты, которые планируется заменить

Сам поиск повторов делится на две категории — поиск точных и неточных повторов. В Duplicate Finder для этого используются внешние инструменты: CloneMiner [6] для точных повторов, FuzzyRepetitions [39] и NgrammSearch [36] для неточных. Механизм выглядит следующим образом: на этапе поиска повторов нужный инструмент вызывается при помощи CLI, потом результаты его работы читаются из файла, конвертируются в общий формат, и затем происходит переход на следующий этап. Далее рассмотрим недостатки такого подхода.

Основной проблемой является неоднородность компонентов для поиска. Во-первых, они написаны на разных языках — Python, C++, C#, что влечет за собой необходимость в дополнительных внешних зависимостях и эмуляторах. Во-вторых они абсолютно изолированы друг от друга, из-за чего возникает ряд проблем: весь процесс поиска каждый раз происходит с нуля, отсутствует возможность хранить какие-либо промежуточные состояния и т.п.

Кроме того, каждый инструмент является отдельным проектом, в связи с чем возникает проблема поддержки — у каждого инструмента свои авторы, свой жизненный цикл. Особенно стоит отметить проблему с CloneMiner. Это единственный инструмент для поиска точных повторов и от него зависит много функционала Duplicate Finder, однако он является закрытой разработкой и от него есть только готовые бинарники для Windows, что, сильно мешает развитию проекта в целом. Также ни один из этих проектов больше не развивается и не поддерживается.

Проблемой также является ограниченность интерфейсов этих инструментов. Каждый из них предоставляет возможность передавать набор параметров, однако их разнообразие достаточно ограниченно. Кроме того, все интерфейсы отличаются между собой и даже одни и те же настройки передаются по-разному, что вноси дополнительные неудобства в работу.

В таблице 1 приведена краткая сводка об основных инструментах для поиска повторов в Duplicate Finder.

Название	Тип повторов	Язык	Тип лицензии
CloneMiner	Точные	C++	Закрытая
FuzzyRepetitions	Неточные	C#	Открытая
NgrammSearch	Неточные	Python	Открытая

Таблица 1: Инструменты поиска повторов

3.2. Определение требований

Новый механизм поиска призван устранить обозначенные выше проблемы. Для его разработки необходимо определить основные требования, реализация которых позволить добиться этой цели. Для нового инструмента поиска повторов были сформулированы следующие функциональные и нефункциональные требования:

1. Инструмент должен быть написан на языке Python для бесшовной интеграции в Duplicate Finder.
2. Должна существовать возможность поиска как точных так и неточных повторов, при этом для них должен быть единый интерфейс.
3. Процесс поиска должен быть универсальным и различаться только в основном применяемом алгоритме.
4. Инструмент должен иметь как API для использования в качестве библиотеки, так и CLI для внешнего использования.
5. Должен предоставляться обширный набор параметров для настройки поиска.

4. Проектирование механизма поиска

В данной главе подробно описываются логика построения основного конвеера механизма поиска, его этапы, а также алгоритмы для поиска точных и неточных повторов.

4.1. Предобработка текста

Сперва стоит обратить внимание на то, что основной целью механизма будет поиск повторов в документации. Документация может быть описательной или справочной, однако в обоих случаях она обычно представляет собой комбинацию текста на естественном языке и исходного кода, хоть и в различных пропорциях. Из-за этой особенности по сути ее можно воспринимать как простой текст, так как в ней отсутствует строгая структура программы. Это значит, что главным фокусом будет нахождение текстовых повторов.

При проектировании механизма в первую очередь нужно подумать про то, какие шаги приведут к наиболее оптимальному результату поиска. Так как найденные фрагменты в конечном итоге нужны пользователю для улучшения документации, чем более семантически осмысленными они будут — тем лучше. Это, означает, что для улучшения результатов необходимо сначала каким-то образом обработать текст. В данной ситуации хорошо подойдут методы NLP [24], часто используемые в машинном обучении [28], например, в нейронный сетях [7, 27], для работы с естественными языками. Далее перечислим основные подходы, которые можно применить при предобработке текста в механизме поиска.

Фильтрация спецсимволов. Тексты часто содержат большое количество специальных символов. Для естественного языка они в основном представлены пунктуацией, для исходного кода — различные управляющие символы языка, такие как скобки и кавычки. Хотя такие символы и могут иметь некоторый семантический смысл, они выполняют вспомогательную роль и плохо отражают содержание. Кроме того, они сильно мешать поиску повторов, так как вносят значитель-

ные помехи. Поэтому первым шагом необходимо очистить текст от всех спецсимволов.

Удаление стоп слов. В естественных языках присутствует множество вспомогательных слов — предлоги, частицы, артикли и т.п., которые также не относятся к содержанию. Они называются стоп слова. Такие слова часто встречаются в тексте и распределены достаточно равномерно, из-за чего они по сути являются шумом, поэтому удаление стоп слов при обработке текста является часто применяемой практикой в машинном обучении. Данный подход будет использоваться в качестве второго шага обработки.

Для поиска точных повторов предобработка включает только два шага, обозначенные выше, так как следующие шаги подразумевают трансформацию слов. Это позволяет улучшить поиск неточных повторов, однако в такой ситуации теряется смысл точного поиска.

Лемматизация. Так как нашей целью является поиск содержательных повторов, хорошей идеей будет избавиться от различных грамматических особенностей естественных языков. Это особенно хорошо применимо к документации, учитывая, что в исходном коде также нередко встречаются разнообразные обычные слова (названия переменных, функций, комментарии и т.п.). В этом может помочь такой процесс как лемматизация — приведение слова к его нормальной форме. В результате применения этого подхода в тексте будет больше одинаковых слов, что увеличит качество находимых повторов.

Стемминг. Еще один подход, который по большей части выполняет ту же самую роль, что и лемматизация — это стемминг. Данный процесс представляет собой выделение из слова его основу, что позволяет игнорировать грамматические особенности языка. Как и лемматизация, способствует более хорошему поиску повторов.

Также очень важную роль играет способ представления обработанного текста, так как от этого зависит, какие методы можно будет использовать. Практически везде текст разбивается на токены, которые потом используются в качестве минимальной единицей обработки. Однако, в разных инструментах используются различные структуры для

представления текста. Самые распространенные — это дерево токенов и массив токенов. Первый вариант обычно используется при работе со структурированными документами, такими как исходный код, второй — при работе с естественными текстами, что больше подходит к нашей задаче. Таким образом, результатом этапа предобработки текста будет являться представление текста в виде массива токенов.

4.2. Поиск точных повторов

Поиск точных повторов подразумевает нахождение фрагментов текста, которые абсолютно идентичны друг другу. В Duplicate Finder для этого использовался CloneMiner [6] — инструмент для поиска клонов в ПО. Особенностью данного инструмента является подход, основанный на последовательности токенов: в отличие от многих других средств, которые используют синтаксическое дерево программы [11], CloneMiner работает с программой как с простым текстом, поэтому его можно применять вне области его предназначения.

Идею алгоритма для поиска точных повторов можно позаимствовать у CloneMiner. Хотя в работе [6] отсутствует полноценное описание алгоритма, лежащего в основе инструмента, а его реализация является закрытым проектом, авторы упоминают основные применяемые подходы. В частности, для поиска повторов вместо часто используемого суффиксного дерева [16] используется суффиксный массив [22]. Так как документация представляет собой простой текст, такой подход хорошо подойдет для нашей задачи. Изначально использование суффиксного массива подразумевает работу со строками, однако несложно обобщить работу с ним для токенов. Так как в качестве повторов нас интересуют фрагменты текста, будем рассматривать весь документ как строку, а каждый токен как ее символ. Тогда, построив суффиксный массив, можно будет сразу находить целые фрагменты текста, являющиеся точными повторами.

Суффиксный массив представляет собой последовательность индексов суффиксов строки, упорядоченных в лексиграфическом порядке.

Можно заметить, что в таком случае суффиксы, начинающиеся с одинаковых подстрок, будут находиться рядом в суффиксном массиве. Так как массив содержит все суффиксы строки, таким образом можно найти все точные повторы в тексте, кроме того, они будут сразу сгруппированы. Для упрощения сравнения рядом стоящих суффиксов можно использовать вспомогательную структуру — LCP[2] массив [21], которой содержит информацию о том, какая часть соседних суффиксов совпадает. Примеры этих двух структур приведены на рис. 2

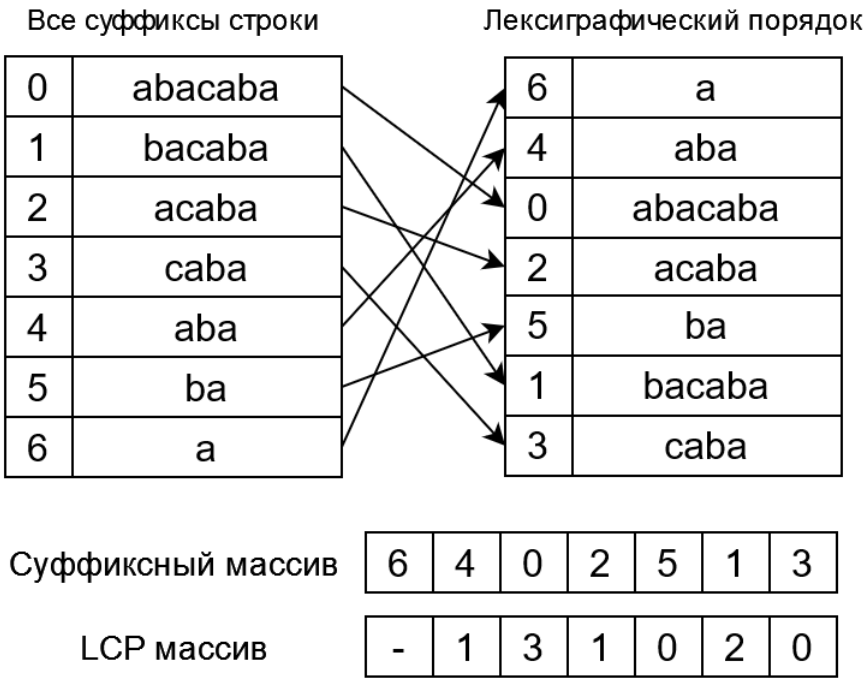

Рис. 2: Пример суффиксного и LCP массивов

Описание алгоритма

Пусть документ представлен в виде последовательности токенов $D = [t_0, ..., t_{n-1}]$. Построим для него суффиксный массив $SA = [sa_0, ..., sa_{n-1}]$. Для этого можно воспользоваться Skew-алгоритмом [19], который позволяет построить суффиксный массив за $O(n)$ используя поразрядную сортировку (Radix sort). Такой подход хорошо подходит для нашей цели, так как минимальной единицей обработки является токен, если быть точнее — его id, и используемый алфавит будет достаточно маленького размера. Затем построим LCP массив $L = [l_0, ..., l_{n-1}]$ на ос-

[2]Longest Common Prefix — массив наибольших общих префиксов

нове суффиксного, что можно также сделать за $O(n)$ используя идею, описанную в [21].

После этого будем итерироваться по суффиксному массиву. Определим параметр m как минимальный размер дубликатов. Для каждого суффикса $s_i, i \in [1, n-1]$ значение l_i показывает сколько общих токенов он имеет с s_{i-1}, и пока $l_i \geq m$ все эти фрагменты являются повторами одной группы. Когда определены все суффиксы текущей группы $[s_j, ..., s_k]$, нужно проверить, являются ли суффиксами других более длинных суффиксов. Для этого каждый суффикс расширяется пока они имеют одинаковый токен слева $t_{s_j-1} = ... = t_{s_k-1}$. В конце алгоритма из суффиксов вырезается совпадающий префикс и создается группа точных повторов, а соответствующие токены помечаются как использованные, и далее пропускаются. Таким образом, каждый токен может быть обработан максимум один раз, из-за чего сложность алгоритма будет $O(n)$ от количества токенов, хотя и с достаточно большой константой.

4.3. Поиск неточных повторов

Понятие неточных повторов можно определить различными способами. В рамках данной работы, учитывая используемые алгоритмы поиска, под группой неточных повторов будем понимать набор текстовых фрагментов $G = (g_1, ..., g_n)$ такой, что для каждой пары $g_i, g_j \in G$ и некоторой функции схожести f_{sim} выполняется неравенство $f_{sim}(g_i, g_j) \leq T$, где $T \geq 0$ — заранее определенный параметр. Таким образом, какие фрагменты будут считаться повторами в основном зависит от выбора функции, а строгость отбора — от параметра.

4.3.1. Алгоритм на основе редакционного расстояния

Одним из распространенных способов сравнения строк является редакционное расстояние или расстояние Левинштайна — минимальное количество операций удаления, вставки и замены символов, которое необходимо для трансформирования одной строки в другую. Этот под-

ход лежит в основе инструмента [39]. Далее рассмотрим алгоритм подробнее.

Сначала весь текст, представляющий собой последовательность токенов, равномерно разбивается на фрагменты одинакового размера. Затем все фрагменты попарно сравниваются, и если два каких-либо фрагмента достаточно похожи — они сохраняются как пара повторов. Основной функцией схожести как раз и является редакционное расстояние, исчисляемое как разница в токенах между фрагментами. Так как вычисление редакционного расстояния процесс достаточно трудоемкий, для оптимизации фрагменты сначала сравниваются по их хешам, которые рассчитываются заранее для каждого фрагмента. Эта оценка является достаточно грубой, но помогает отсеять сильно отличающиеся пары. Граничные значения схожести являются внешними параметрами алгоритма. На последнем этапе пары повторов объединяются в группы и затем проводится расширение повторов в этих групп за счет объединения с соседними фрагментами и удаления пересечений между группами.

Хотя основную концепцию данного подхода можно оставить, у оригинального инструмента есть ряд недостатков, решив которые можно улучшить алгоритм поиска:

- могут обрабатываться только документы в формате XML, что сильно сужает область применения
- группы составляются простым перебором пар, что неэффективно и предоставляет мало возможностей для дальнейшей обработки
- используемый подход хеширования достаточно ограничен и дает слишком неточную оценку

Описание алгоритма

Так как предобработкой текста и балансировкой групп занимаются отдельные компоненты, сосредоточим внимание на основной части алгоритма поиска. Будем рассматривать входной документ D как последовательность из n токенов $D = [t_0, ..., t_{n-1}]$. Определим размер фраг-

ментов $0 < m < n$ и равномерно разобьем текст на $k = n/m$ фрагментов этого размера (если $nmodm \neq 0$, тогда дополним текст с конца пустыми токенами). В результате получим набор фрагментов $[g_0, ...g_{k-1}]$, где $g_i = [t_{i*m}, ..., t_{i*(m+1)-1}]$.

После разбиения для каждого фрагмента вычислим его отпечаток в виде 32-битного хеша, используя подход sim-hash [10]. Для этого посчитаем хеш каждого токена при помощи алгоритма MD5 [35]. Выбор алгоритма хеширования обусловлен тем, что для наших целей криптографическая стойкость роли не играет, а при этом MD5 вычисляется быстрее популярной альтернативы SHA256 [29]. Далее для фрагмента g_i определим результирующих хеш h_i на основе хешей токенов: возьмем последние 32 бита каждого из них, и для каждой позиции, если более половины битов равны одному, то в результат запишем 1, иначе 0. Укороченный пример хеширования приведен на рис. 3.

Hash 1	1	1	0	1	1	0	1	1
Hash 2	1	1	0	0	0	1	1	0
Hash 3	0	1	1	0	1	0	0	1
Result	1	1	0	0	1	0	1	1

Рис. 3: Пример вычисления итогового хеша

Затем происходит попарное сравнение. Если разница в хешах фрагментов g_i, g_j, определенная как количество единиц в числе $h_i \oplus h_j$, не превышает определенной границы, то вычисляется редакционное расстояние между фрагментами. Для этого применяется алгоритм Укконена [33] для приблизительного сравнения строк (approximate string matching). Данный подход позволяет оптимально дать ответ на вопрос: больше ли редакционное расстояние между элементами, чем заданная граница, что мы и хотим выяснить. Так как размеры фрагментов одинаковы, а максимальное количество допустимых операций константно, сложность вычисления будет $O(m)$. Если в итоге фрагменты успешно проходят оба этапа сравнения, то они записываются, как пара повторов. В общей сложности этот этап замет не больше $O(k^2 * m) = O(\frac{n^2}{m})$

После нахождения всех пар повторов, необходимо объединить их в группы. Можно заметить, что по сути эти пары образуют неориентированный граф, где вершинами являются фрагменты, а ребра отображают их сходство. Графы очень широко распространены и имеют множество применений в разных областях, и, соответственно, существует большое количество различных алгоритмов и подходов для работы с графами, поэтому такое представление является очень выгодным. Составление групп повторов происходит путем поиска компонент связности полученного графа. Как более строгий, но медленный вариант можно использовать, например, поиск всех клик[3] графа. Так как поиск компонент связности даже в худшем случае не будет превышать $O(c^2)$, в целом сложность алгоритма можно оценить как $O(\frac{n^2}{m})$ от количества токенов и размера фрагментов.

4.3.2. Алгоритм на основе N-грамм

Еще один распространенный подход при работе с текстами — это N-граммы [8]. Построив множества N-грамм для двух фрагментов, можно затем получить оценку их схожести на основе сравнения этих множеств. На рисунке 4 приведен пример таких множеств для различных N. На этом принципе основан алгоритм поиска, описанный в статье [13]. Улучшенная версия данного алгоритма приводится в работе [36].

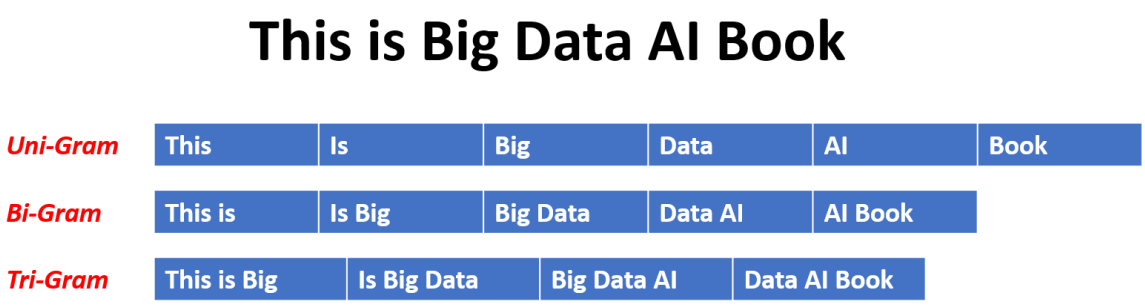

Рис. 4: Пример множеств N-грам

Отдельно рассматривать этот алгоритма смысла нет, так как описанная ниже итоговая версия по большей части совпадает с ним, однако

[3]Клика — максимальный полный подграф

стоит отметить некоторые недостатки, которые можно устранить для дальнейшего его улучшения:

- алгоритм содержит большое количество последовательных пересечений с множествами одних и тех же фрагментов, однако так как пересечение множеств — операция ассоциативная, достаточно вычислить пересечение группы один раз
- в работе описывается только работа с 3-граммами, однако алгоритм можно обобщить для любых N

Описание алгоритма

Разобьем входной документ на набор предложений: $D = [s_0, \ldots, s_{n-1}]$. Для каждого предложения s_i вычислим множество N-грам N_i. Изначально определим каждое предложение в отдельную группу повторов G_i, и для каждой такой группы будем поддерживать множество N-грам, которое будет являться пересечением таких множеств каждого предложения этой группы $N_{G_i} = \cap N_j, N_j \in G_i$. Затем будем пытаться добавить каждое предложение к какой-либо группе: для s_i и группы $G_k, k \in [0 \ldots i-1, i+1 \ldots n-1]$ вычислим мощность пересечения его множества N-грам с множеством группы-кандидата $overlap_k = |N_i \cap N_{G_k}|$. Затем выберем группу с максимальным пересечением $j = \underset{k}{\operatorname{argmax}}(overlap_k)$ и если $overlap_j > T$, где T - заранее заданная граница, то добавляем предложение s_i к группе G_j, обновляя $N_{G_j} = N_{G_j} \cap N_i$. После итерации по всем предложениям, результатом работы алгоритма будет набор групп неточных повторов.

Рассмотрим сложность алгоритма. Так как количество предложений линейно зависит от общего количества токенов, дальнейшие расчеты будем проводить относительно предложений. Также стоит отметить, что предложение состоит из константного количества токенов, как, соответственно и множество его N-грам. Пересечение двух множеств можно вычислить за $O(\min(k_i, k_j))$, где k_i, k_j - размеры множеств. Итерация по всем предложениям займет $O(n)$, для каждого предложения нужно пересечь его множества с каждой группой. Само пересечение в нашей ситуации выполняется за $O(1)$. В худшем случае, если

в тексте нет повторов, то количество групп будет всегда равно n. На практике, хоть количество групп и будет уменьшаться, повторы покрывают не очень большую часть текста. Таким образом, сложность алгоритма можно оценить как $O(n^2)$ от количества токенов.

4.4. Балансировка групп повторов

Результатом работы алгоритмов поиска является набор групп повторов. Так как основная задача инструмента — это поиск простых текстовых повторов, дальнейшие анализ или обработка полученных групп фрагментов с точки зрения семантики не требуется. Однако можно попробовать улучшить эти группы: во-первых — избавиться от возможных нежелательных последствий, которые могут возникнуть в результате работы алгоритма, а во-вторых — попробовать, манипулируя фрагментами, изменить сами группы, чтобы увеличить их осмысленность. Далее подробно рассмотрим конкретные шаги для достижения этих целей.

Фильтрация

Первым и последним шагом этого процесса является фильтрация незначимых групп. В результате работы некоторых алгоритмов, среди групп повторов могут быть пустые или содержащие только один фрагмент. Такие группы не представляют никакого интереса и их нельзя использовать для дальнейшей обработки, поэтому их стоит просто удалить. Особенно много таких групп может возникнуть в результате работы следующего шага из-за перестановок фрагментов между группами.

Балансировка

В зависимости от особенностей исходного текста и алгоритма поиска, повторы могут быть объединены в группы не лучшим образом. Для оценки качества разбиения нужно ввести некоторую метрику. Рассмотрим основные параметры, которые влияют на качество группы повторов:

- Количество повторов в группе — чем больше какой-либо фраг-

мент повторяется в тексте, тем больше интереса он представляет

- Длина повторов в группе — чем длиннее найденные фрагменты, тем больше шанс, что они будут содержать более осмысленную информацию

На основе этих параметров можно ввести простую функцию значимости группы:

$$f_{sig} = n * m^2 \tag{1}$$

где n - количество повторов, m - средняя длина.

Рассмотрим две группы повторов, а точнее как расположены их фрагменты в тексте. В общем случае может быть 3 варианта их взаимного расположения:

1. Фрагменты двух групп не находятся рядом (рис. 5). В таком случае ничего сделать не получится.

2. Некоторые фрагменты двух групп являются соседями в тексте (рис. 6). Теоретически соседние фрагменты можно объединить в один более длинный. Однако, так как группы пересекаются лишь частично, нельзя гарантировать, что такое объединение не нарушит логической целостности групп. Таким образом, в этом случае также не стоит изменять группы.

3. Все фрагменты одной группы находятся рядом с фрагментами другой. В таком случае можно попробовать объединить эти фрагменты, удалив их из другой группы как показано на рисунке 7. Для этого воспользуемся функцией значимости: будем проводить эту операцию, если после нее увеличится значимость групп: $f_{sig}(g_i^1) + f_{sig}(g_j^1) < f_{sig}(g_i^2) + f_{sig}(g_j^2)$. Частный случай — если в группах одинаковые количества повторов, тогда можно их сразу склеить.

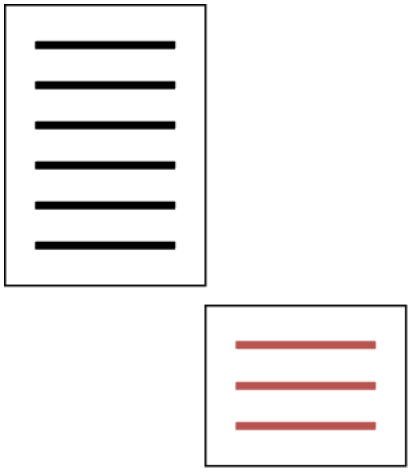

Рис. 5: Группы повторов, не имеющие соседних фрагментов

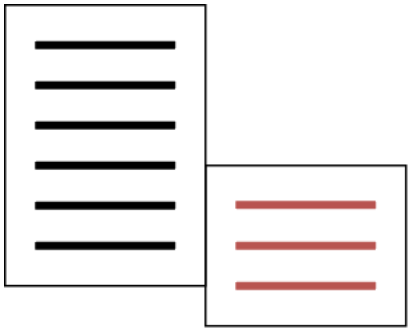

Рис. 6: Группы повторов, содержащие некоторые соседние фрагменты

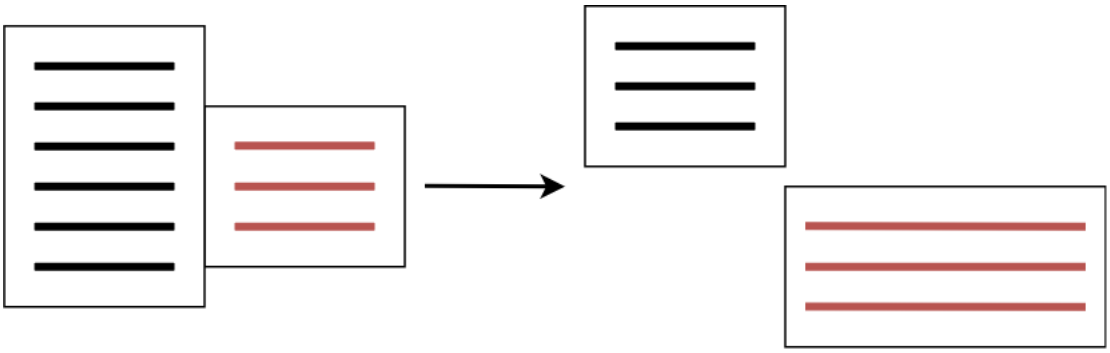

Рис. 7: Все фрагменты одной группы повторов являются соседями другой

5. Особенности реализации

5.1. Используемые технологии

Для реализации был использован язык Python, чтобы добиться максимальной совместимости с DuplicateFinder. Среди множества инструментов для работы с естественным языком, ведущей платформой является NLTK(Natural Language Toolkit), представляющая собой набор библиотек для достижения различных целей. В рамках данный работы для предобработки текста использовались следующие технологии:

- Различные токенизаторы для разбиения текста — LineTokenizer, WordPunctTokenizer, SentTokenizer
- Реализация алгоритма стемминга — PorterStemmer
- Пакет для лемматизации — WordNetLemmatizer

5.2. Архитектура

Основной задачей при разработке архитектуры было разбиение этапов конвейера на отдельные компоненты и поддержка низкой связности этих компонент. Такой подход позволит легко добавлять в инструмент новые методы токенизации и алгоритмы поиска. Кроме того, он предоставляет широкие возможности для манипулирования всем процессом поиска: для каждого этапа можно выбирать одну из существующих компонент, подстраивая инструмент под конкретную задачу. Схема архитектуры представлена ниже на рисунке 8.

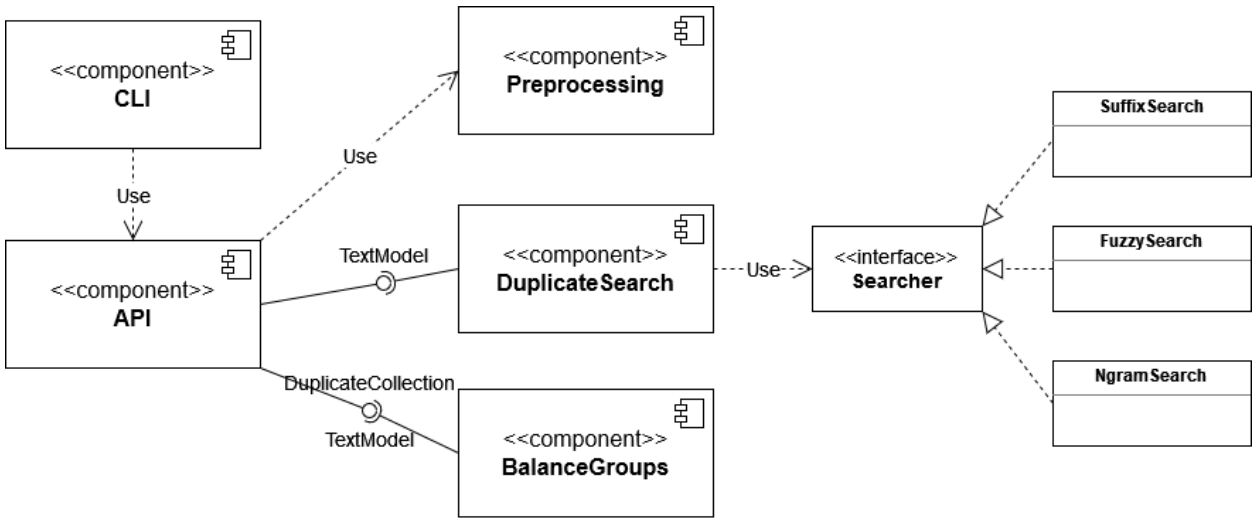

Рис. 8: Архитектура механизма поиска

6. Тестирование

Для тестирования инструмента были выбраны документации ряда крупных проектов: Blender, GIMP, PostgreSQL, Zend Framework, Apache Subversion. Информация о документах приведена в таблице 2.

Документ	Тип проекта	Тип документации	Объем, Кб	Кол-во токенов
The GIMP user manual	Графический редактор	Рук-во пользователя	1320	164035
PostgreSQL manual	СУБД	Рук-во пользователя	727	72728
Subversion Book	Система управления версиями	Рук-во по администрированию	1110	110270
Zend Framework V1 guide	PHP фреймворк	Рук-во программиста	1649	164035
Blender Manual	3D редактор	Рук-во пользователя	2684	285418

Таблица 2: Результаты точного поиска

Для этих документов был проведен поиск точных и неточных повторов, и собрана статистика по найденным группам. Характеристиками,

представляющими наибольший интерес являются количество групп повторов и их размер, а также длина фрагментов. Результаты приведены ниже в таблицах 3 и 4.

Документ	Группы повторов	Средний размер группы	Средняя длина повтора	Покрытие документа
GIMP Manual	400	2.57	14.59	11%
PostgreSQL Manual	289	2.30	16.31	14%
Subversion book	218	2.17	17.27	7%
Zend Framework guide	557	2.44	16.58	13
Blender Manual	587	2.33	19.14	9%

Таблица 3: Результаты точного поиска

Документ	Группы повторов	Средний размер группы	Средняя длина повтора	Покрытие документа
GIMP Manual	574	2.65	13.64	15%
PostgreSQL Manual	464	2.66	17.17	25%
Subversion book	282	2.27	18.93	10%
Zend Framework guide	522	2.32	22.96	16%
Blender Manual	1393	2.48	14.22	16%

Таблица 4: Результаты неточного поиска

Как видно по результатам работы, разработанный инструмент хорошо находит как точные так и неточные повторы. Также стоит отметить, что практические результаты хорошо отображают теоретические ожидания: среди документаций совершенно различных проектов, около 10% документа является дублированными фрагментами.

Заключение

В ходе данной работы были получены следующие результаты.

1. Проанализированы основные подходы и средства, которые используются в существующих инструментах для поиска повторов: синтаксические и суффиксные деревья, нейронные сети, обработка естественного языка, алгоритмы хеширования, N-граммы.

2. Выявлены следующие основные требования к новому механизму поиска: объединение точного и неточного поиска, унификация процесса поиска, высокая степень настраиваемости.

3. Спроектирован конвейер для механизма поиска, состоящий из трех основных этапов: предобработка текста, применение алгоритмов поиска повторов, балансировка групп повторов.

4. Разработаны алгоритмы для точного и неточного поиска повторов на основе использованных в Duplicate Finder инструментов.

5. Выполнена реализация инструмента на языке Python с использованием пакета NLTK для предобработки текста, исходный код доступен по ссылке `https://github.com/IceWind2/TextDuplicateSearch`; проведена интеграция с Duplicate Finder.

6. Проведено тестирование инструмента на корпусе документов, по результатам работы собрана статистика и проведен ее анализ.

Список литературы

[1] Alsulami Bassma S, Abulkhair Maysoon F, and Eassa Fathy E. Near duplicate document detection survey // International Journal of Computer Science and Communications Networks. — 2012. — Vol. 2, no. 2. — P. 147–151.

[2] Białecki Andrzej, Muir Robert, Ingersoll Grant, and Imagination Lucid. Apache lucene 4 // SIGIR 2012 workshop on open source information retrieval. — 2012. — P. 17.

[3] Laguë Bruno, Proulx Daniel, Mayrand Jean, Merlo Ettore M, and Hudepohl John. Assessing the benefits of incorporating function clone detection in a development process // 1997 Proceedings International Conference on Software Maintenance / IEEE. — 1997. — P. 314–321.

[4] Baker Brenda S. On finding duplication and near-duplication in large software systems // Proceedings of 2nd Working Conference on Reverse Engineering / IEEE. — 1995. — P. 86–95.

[5] Banerjee Sean, Cukic Bojan, and Adjeroh Donald. Automated duplicate bug report classification using subsequence matching // 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering / IEEE. — 2012. — P. 74–81.

[6] Basit Hamid Abdul and Jarzabek Stan. Efficient token based clone detection with flexible tokenization // Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. — 2007. — P. 513–516.

[7] Bishop Chris M. Neural networks and their applications // Review of scientific instruments. — 1994. — Vol. 65, no. 6. — P. 1803–1832.

[8] Broder Andrei Z. On the resemblance and containment of documents // Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171) / IEEE. — 1997. — P. 21–29.

[9] Juergens Elmar, Deissenboeck Florian, Feilkas Martin, Hummel Benjamin, Schaetz Bernhard, Wagner Stefan, Domann Christoph, and Streit Jonathan. Can clone detection support quality assessments of requirements specifications? // Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. — 2010. — P. 79–88.

[10] Charikar Moses S. Similarity estimation techniques from rounding algorithms // Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. — 2002. — P. 380–388.

[11] Baxter Ira D, Yahin Andrew, Moura Leonardo, Sant'Anna Marcelo, and Bier Lorraine. Clone detection using abstract syntax trees // Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272) / IEEE. — 1998. — P. 368–377.

[12] Cohen Edith and Kaplan Haim. Summarizing data using bottom-k sketches // Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. — 2007. — P. 225–234.

[13] Kanteev LD, Luciv DV, Koznov DV, Smirnov MN, et al. Discovering near duplicate text in software documentation // Труды Института системного программирования РАН. — 2017. — Vol. 29, no. 4. — P. 303–314.

[14] Mikolov Tomas, Sutskever Ilya, Chen Kai, Corrado Greg S, and Dean Jeff. Distributed representations of words and phrases and their compositionality // Advances in neural information processing systems. — 2013. — Vol. 26.

[15] Feng Weiqi and Deng Dong. Allign: Aligning all-pair near-duplicate passages in long texts // Proceedings of the 2021 International Conference on Management of Data. — 2021. — P. 541–553.

[16] Gusfield Dan. Algorithms on stings, trees, and sequences: Computer science and computational biology // Acm Sigact News. — 1997. — Vol. 28, no. 4. — P. 41–60.

[17] Hirschberg Daniel S. A linear space algorithm for computing maximal common subsequences // Communications of the ACM. — 1975. — Vol. 18, no. 6. — P. 341–343.

[18] Ho Phuc-Tran, Kim Hee-Sun, and Kim Sung-Ryul. Application of sim-hash algorithm and big data analysis in spam email detection system // Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems. — 2014. — P. 242–246.

[19] Kärkkäinen Juha and Sanders Peter. Simple linear work suffix array construction // Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30 / Springer. — 2003. — P. 943–955.

[20] Koschke Rainer, Falke Raimar, and Frenzel Pierre. Clone detection using abstract syntax suffix trees // 2006 13th Working Conference on Reverse Engineering / IEEE. — 2006. — P. 253–262.

[21] Landau Gad M, Kasai Toru, Lee Gunho, Arimura Hiroki, Arikawa Setsuo, and Park Kunsoo. Linear-time longest-common-prefix computation in suffix arrays and its applications // Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings 12 / Springer. — 2001. — P. 181–192.

[22] Manber Udi and Myers Gene. Suffix arrays: a new method for on-line string searches // siam Journal on Computing. — 1993. — Vol. 22, no. 5. — P. 935–948.

[23] Broder Andrei Z, Charikar Moses, Frieze Alan M, and Mitzenmacher Michael. Min-wise independent permutations // Proceedings of the thirtieth annual ACM symposium on Theory of computing. — 1998. — P. 327–336.

[24] Nadkarni Prakash M, Ohno-Machado Lucila, and Chapman Wendy W. Natural language processing: an introduction // Journal of the American Medical Informatics Association. — 2011. — Vol. 18, no. 5. — P. 544–551.

[25] Parnas David Lorge. Precise documentation: The key to better software // The future of software engineering. — Springer, 2010. — P. 125–148.

[26] Porubän Jaroslav et al. Preliminary report on empirical study of repeated fragments in internal documentation // 2016 Federated Conference on Computer Science and Information Systems (FedCSIS) / IEEE. — 2016. — P. 1573–1576.

[27] Prabowo Damar Adi and Herwanto Guntur Budi. Duplicate question detection in question answer website using convolutional neural network // 2019 5th International conference on science and technology (ICST) / IEEE. — 2019. — Vol. 1. — P. 1–6.

[28] Vijayarani S, Ilamathi Ms J, Nithya Ms, et al. Preprocessing techniques for text mining-an overview // International Journal of Computer Science & Communication Networks. — 2015. — Vol. 5, no. 1. — P. 7–16.

[29] Rachmawati Dian, Tarigan JT, and Ginting ABC. A comparative study of Message Digest 5 (MD5) and SHA256 algorithm // Journal of Physics: Conference Series / IOP Publishing. — 2018. — Vol. 978. — P. 012116.

[30] Rattan Dhavleesh, Bhatia Rajesh, and Singh Maninder. Software clone detection: A systematic review // Information and Software Technology. — 2013. — Vol. 55, no. 7. — P. 1165–1199.

[31] Roy Chanchal Kumar and Cordy James R. A survey on software clone detection research // Queen's School of Computing TR. — 2007. — Vol. 541, no. 115. — P. 64–68.

[32] Amoui Mehdi, Kaushik Nilam, Al-Dabbagh Abraham, Tahvildari Ladan, Li Shimin, and Liu Weining. Search-based duplicate defect detection: An industrial experience // 2013 10th Working Conference on Mining Software Repositories (MSR) / IEEE. — 2013. — P. 173–182.

[33] Ukkonen Esko. Algorithms for approximate string matching // Information and control. — 1985. — Vol. 64, no. 1-3. — P. 100–118.

[34] Wang Zhizhi, Zuo Chaoji, and Deng Dong. TxtAlign: Efficient Near-Duplicate Text Alignment Search via Bottom-k Sketches for Plagiarism Detection // Proceedings of the 2022 International Conference on Management of Data. — 2022. — P. 1146–1159.

[35] Yong-Xia Zhao and Ge Zhen. MD5 research // 2010 second international conference on multimedia and information technology / IEEE. — 2010. — Vol. 2. — P. 271–273.

[36] Горгулов Павел Олегович. Модель N-грамм для поиска нечетких повторов в «плоских» текстах : Выпускная квалификационная работа бакалавра ; Санкт-Петербургский государственный университет. — 2018.

[37] Зельцер НГ. Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге // Труды Института системного программирования РАН. — 2013. — Vol. 25. — P. 39–50.

[38] Луцив Дмитрий Вадимович. Поиск неточных повторов в документации программного обеспечения : Диссертация на соискание научно степени кандидата физико-математических наук ; Санкт-Петербургский государственный университет. — 2018.

[39] Столпнер Лев Артемович. Обнаружение нечётких повторов в форматированных текстах : Выпускная квалификационная работа бакалавра ; Санкт-Петербургский государственный университет. — 2016.