Санкт-Петербургский государственный университет

Программная инженерия

Глазырин Антон Георгиевич

Универсальный механизм первичного поиска повторов в тексте для пакета Duplicate Finder

Отчет по производственной (преддипломной) практике

Научный руководитель:

доцент, кафедра системного программирования,

к.ф.-м.н. Луцив Дмитрий Вадимович

Санкт-Петербург

2023

Содержание

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[1]. 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 [27, 28]. Кроме того, в этих работах подтверждается, что повторы действительно оказывают негативное влияние на общее качество документации.

Сам по себе поиск повторов практически всегда является некоторой подзадачей для достижения более осмысленной цели, поэтому он применяется в совершенно различных областях[2]. В сфере разработки ПО наибольшее внимание уделяется поиску клонов в исходном коде[9]. 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.[13, 14, 16]. 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[29], 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[30, 31, 32]: 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 выпускной работы является разработка и реализация унифицированной подсистемы поиска точных и неточных повторов для Duplicate Finder Toolkit и ее интеграция с заменой существующих компонент. Для достижения этой цели в рамках работы были сформулированы следующие задачи.

1. Анализ предметной области - применение поиска повторов в различных сферах, поиск повторов в документации в частности.

2. Выявление недостатков поиска повторов в DuplicateFinder и определение требований к новому механизму.

3. Определение этапов процесса поиска повторов и их содержимое: идеи, подходы, алгоритмы, проектирование архитектуры инструмента на основе этого процесса.

4. Реализация инструмента и его интеграция в DuplicateFinder.

5. Проведение тестирования разработанного инструмента.

2 Обзор

2.1 Поиск повтров в различных сферах

Поиск повторов применяется во многих областях для достижения разнообразных целей [2]. 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 [3] рассматривается метод вычисления схожести двух документов путем сравнения текстовых фрагментов. Для сравнения используется подход min-hash [4]. Авторы инструмента [7] продолжили эту тему в своей работе. Вместо непосредственного сравнения фрагментов они предложили группировать их при помощи подхода bottom-k sketches [8]. 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[6]. 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 [5].

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 [9, 10]. Кроме того, наличие большого количество работ по этой теме можно объяснить и тем,

что согласно исследованиям [11, 12] 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 [14, 13] позволяют находить повторы при помощи сравнивания частей AST, являющихся похожими структурными единицами. Еще одним широко используемым средством являются суффиксные деревья [15]. Так как использование суффиксного дерева подразумевает работу уже с простым текстом, этот подход можно применять одновременно с использованием AST, например, как описано в [16].

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 [17] описывается фреймворк для нахождения похожих отчетов компании BlackBerry, который использует популярную библиотеку Apache Lucene [18] для неточного поиска. Авторы инструмента [19] используют подход на основе вычисления наибольшей общей подстроки из токенов [20] для обнаружения схожих отчетов в репозитории ошибок Firefox.

Различные подходы для определения схожести фрагментов теста нередко применяются в NLP[1] [22]. Особый интерес вызывает модель n-грамм [21] - 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 [24]. Они позволяют определять смысловую нагрузку текста вне зависимости от синтаксической составляющей, таким образом можно сравнивать семантическую схожесть двух фрагментов, такой подход рассматривается в работе [25]. Частным случаем нейронных сетей являются специальные модели, такие как [26], ко-

[1]Natural Language Processing - обработка естественного языка

торые позволяют преобразовать слова в векторное представление, отображающее семантические особенности.

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 [27, 28], 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 работе [29] 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 [30], 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 [31] 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 [32]: текст разбивается на предложения, и для них вычисляются множества N-грамм. Затем предложения, имеющие схожие множества объединяются в группы. Ал-

горитм объединения был далее усовершенствован в работе [33].

3 Требования к механизму поиска

В данной главе приводится анализ недостатков существующего механизма поиска в Duplicate Finder и определяются основные задачи, которые призван решать 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.

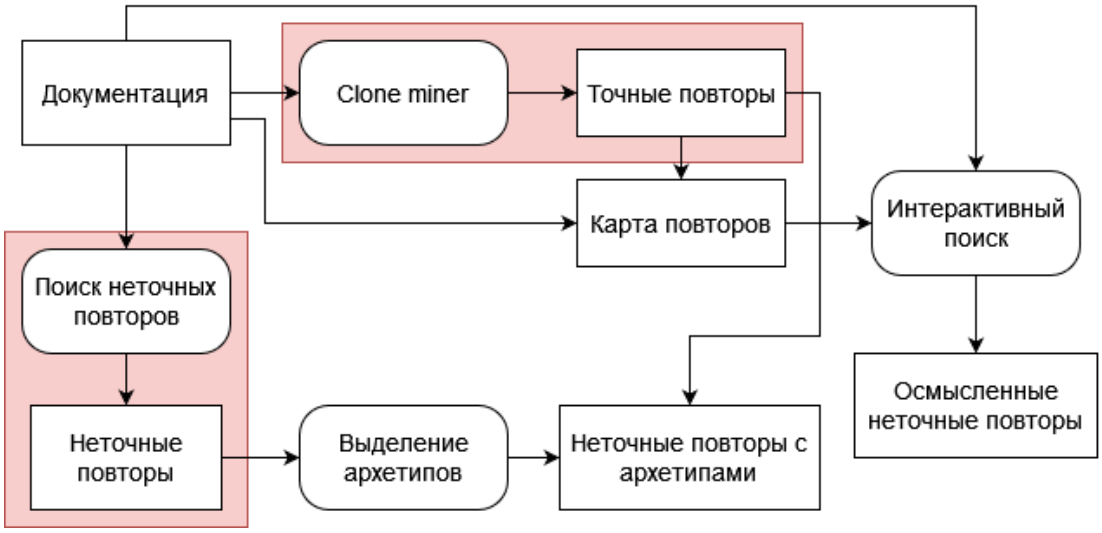

Рис. 1: Схема работы Duplicate Finder. Красным отмечены компоненты, которые планируется заменить

Сам поиск повторов делится на две категории - поиск точных и неточных повторов. В Duplicate Finder для этого используются внешние инструменты: CloneMiner [30] для точных повторов, FuzzyRepetitions [31] и NgrammSearch [33] 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. На данный момент ни один из этих проектов не поддерживается. 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, очевидно, сильно мешает развитию проекта в целом.

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**	**Поддержка**
CloneMiner	Точные	C++	Закрытая	Отсутствует
FuzzyRepetitions	Неточные	C#	Открытая	Отсутствует
NgrammSearch	Неточные	Python	Открытая	Отсутствует

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 [22], часто используемые в машинном обучении [23], например, в нейронный сетях [24, 25], 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 значительные 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 то-

кены - некоторые последовательности символов, которые будут в дальнейшем являться 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[30] - 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[13], 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 [30] 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 [15] используется суффиксный массив [34]. 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 [35], которой содержит информацию о том, какая часть соседних суффиксов совпадает. Примеры этих двух структур приведены на рис. 2

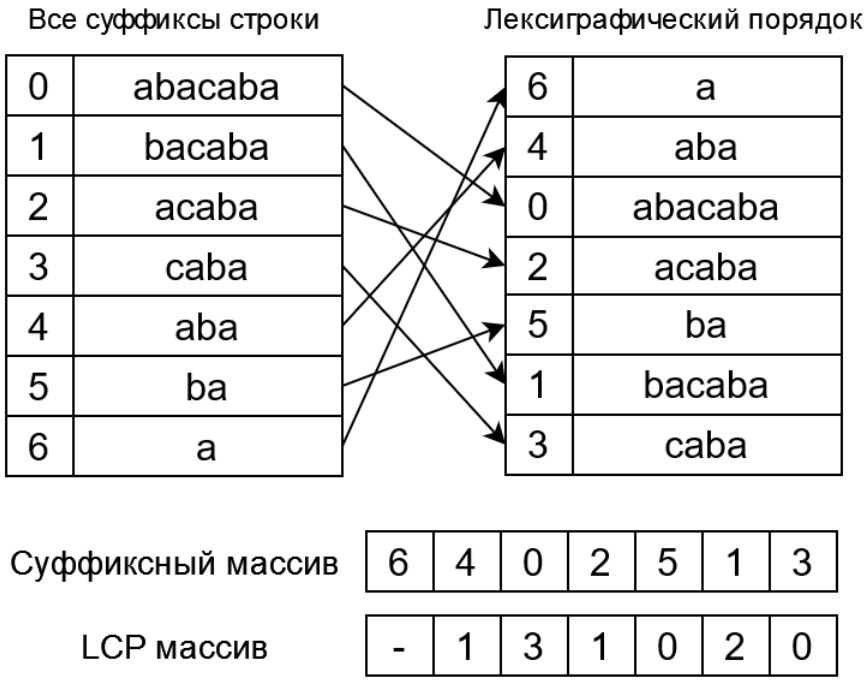

Рис. 2: Пример суффиксного и LCP массивов

Описание алгоритма

[2]Longest Common Prefix - массив наибольших общих префиксов

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 [36], 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 основе суффиксного, что можно также сделать за $O(n)$ используя идею, описанную в [35].

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 $O(n)$, 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 подход лежит в основе инструмента [31]. 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 фрагментов $0 < m < n$ и равномерно разобьем текст на $k = n/m$ фрагментов этого размера (если $n \mod m \neq 0$, 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[6]. Для этого посчитаем хеш каждого токена при помощи алгоритма MD5[40]. Выбор алгоритма хеширования обусловлен тем, что для наших целей криптографическая стойкость роли не играет, а при этом MD5 вычисляется быстрее популярной альтернативы SHA256[41]. 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 ре-

дакционное расстояние между фрагментами. Для этого применяется алгоритм Укконена [37] 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 $O(\frac{k^2}{m})$

4.3.2 Алгоритм на основе N-грамм

Еще один популярный подход для работы с текстами - это N-граммы.

Данный алгоритм основан на идее из работы [33]. Суть идеи заключается в следующем: текст разбивается на предложения, для каждого предложения вычисляется множество N-грамм. Затем предложения объединяются в группы по такому принципу: если множество N-грамм предложения-кандидата хотя бы на x% пересекается

[3]Клика - максимальный полный подграф

с каждым предложением группы, то это предложение добавляется в эту группу.

4.4 Балансировка групп повторов

После того, как найдены основные группы повторов, можно сказать, что цель механизма поиска достигнута. Однако, можно попробовать улучшить полученные результаты. Будем пытаться сбалансировать две группы повторов путем склеивания повторов и удаления их из группы. Для того, чтобы определять, когда это надо, а когда нет, будем руководствоваться, во-первых, метриками качества(например, blow-up[27]), и следить, чтобы они не ухудшались при балансировании, а во-вторых введем некоторую функцию значимости, и если у результирующих групп значимость больше, чем у изначальных, тогда балансируем. В результате группы будут более осмысленные.

5 Тестирование

Для тестирования инструмента были выбраны документации нескольких проектов: GIMP Manual, PostgreSQL Manual, Zend Framework Manual, Subversion book. Для этих документов был проведен поиск, и собрана статистика по найденным группам повторов. Результаты приведены ниже в таблицах 2 и 3.

	GIMP	PostgreSQL	Apache Subversion	Zend Framework
Токены	132554	72728	110270	164035
Группы повторов	400	289	218	557
Средний размер группы	2.57	2.30	2.17	2.44
Средняя длина повтора	14.59	16.31	17.27	16.58
Покрытие документа	11%	14%	7%	13%

Таблица 2: Результаты точного поиска

	GIMP	PostgreSQL	Apache Subversion	Zend Framework
Токены	132554	72728	110270	164035
Группы повторов	574	464	282	522
Средний размер группы	2.65	2.66	2.27	2.32
Средняя длина повтора	13.64	17.17 2	18.93	22.96
Покрытие документа	15%	25%	10%	16%

Таблица 3: Результаты неточного поиска

6 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. Для устранения недостатков DuplicateFinder сформулированы следующие основные требования к новому механизму: 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, объединение групп повторов.

4. Выполнена реализация инструмента на языке Python с использованием пакета NLTK для предобработки текста; проведена интеграция с Duplicate Finder с полной заменой соответствующих модулей.

5. Проведено тестирование инструмента на корпусе документов и по результатам работы собрана статистика.

Список литературы

[1] Parnas, D.L. Precise Documentation: The Key to Better Software / D.L. Parnas // The Future of Software Engineering. — 2011. — P. 125–148.

[2] https://www.researchgate.net/profile/Fathy-Eassa-2/publication/266005488_Near_Duplicate_Document_Detection_Survey/links/Duplicate-Document-Detection-Survey.pdf

[3] https://dl.acm.org/doi/abs/10.1145/3448016.3457548

[4] https://dl.acm.org/doi/pdf/10.1145/276698.276781

[5] https://dl.acm.org/doi/abs/10.1145/2663761.2664221

[6] https://www.cs.princeton.edu/courses/archive/spring04/cos598B/bib/CharikarEs

[7] https://dl.acm.org/doi/abs/10.1145/3514221.3526178

[8] https://dl.acm.org/doi/abs/10.1145/1281100.1281133

[9] https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf

[10] Rattan, D. Software clone detection: A systematic review / D. Rattan, R. Bhatia, M. Singh // Information and Software Technology. — 2013. — 55 (7). — P. 1165–1199.

[11] B. Lague, D. Proulx, E. Merlo, J. Mayrand, J. Hudepohl, Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, International Conference on Software Maintenance 1997, IEEE.

[12] Brenda Baker, On Finding Duplication and Near-Duplication in Large Software Systems, Working Conference on Reverse Engineering 1995, IEEE.

[13] http://facebook.comwww.semanticdesigns.com/Company/Publications/ICSM98

[14] Зельцер, Н.Г. / Н.Г. Зельцер // Труды института системного про- граммирования РАН. — 2013. — Т. 25. — С. 39–50.

[15] Gusfield, D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, NY, 1997

[16] https://ieeexplore.ieee.org/abstract/document/4023995

[17] https://stargroup.uwaterloo.ca/ mamouika/papers/pdf/MSR13_preprint.pdf

[18] https://lucene.apache.org/

[19] https://ieeexplore.ieee.org/abstract/document/6375640

[20] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. Commun. ACM 18, 6 (June 1975)

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

[22] https://academic.oup.com/jamia/article/18/5/544/829676

[23] Vijayarani, S., Ilamathi, M. J., Nithya, M. Preprocessing Techniques for Text Mining - An Overview // International Journal of Computer Science & Communication Networks. 2015. Vol 5, no. 1. P.7–16

[24] https://pubs.aip.org/aip/rsi/article/65/6/1803/682910/Neural-networks-and-their-applicationsNeural

[25] https://ieeexplore.ieee.org/abstract/document/9166343

[26] 105. Mikolov, T. Distributed representations of words and phrases and their compo-sitionality / T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean // Proceed- ings of Advances in neural information processing systems. — 2013. — P. 3111–3119.

[27] Juergens, E., Can clone detection support quality assessments of requirements specifications? / E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, J. Streit // Proceedings of ACM/IEEE 32ndInternational Conference on Software Engineering. — Vol. 2. — 2010. — P. 79–88.

[28] https://ieeexplore.ieee.org/abstract/document/7733462

[29] https://docline.github.io/pdf/phd-theses/2018.luciv.pdf

[30] Basit, H.A. Efficient Token Based Clone Detection with Flexible Tokenization / H. A. Basit, S. Jarzabek // Proceedings of the 6th Joint Meeting on European Soft-ware Engineering Conference and the ACM SIGSOFT Symposium on the Foun-dations of Software Engineering: Companion Papers. — 2007. — P. 513–516.

[31] https://docline.github.io/pdf/graduate-theses/

[32] https://docline.github.io/pdf/articles/kanteev.kostykov.luciv.koznov.smirnov.20

[33] https://docline.github.io/pdf/graduate-theses/2018.Gorgulov_Pavel.pdf

[34] Manber, U., Myers, G. Suffix Arrays: A New Method for On-Line String Searches // Proc. First Ann. ACM-SIAM Symp. Discrete Algorithm. 1990. P.319–327.

[35] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications // Combinatorial Pattern Matching. 2001. P.181–192.

[36] https://link.springer.com/chapter/10.1007/3-540-45061-0_73

[37] https://www.sciencedirect.com/science/article/pii/S0019995885800462

[38] https://dl.acm.org/doi/abs/10.1145/2833157.2833162

[39] LLVM

[40] https://ieeexplore.ieee.org/abstract/document/5474379/citations#citations

[41] https://iopscience.iop.org/article/10.1088/1742-6596/978/1/012116/meta