Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем

Системное программирование

Столпнер Лев Артемович

Обнаружение нечётких повторов в форматированных текстах

Бакалаврская работа

Научный руководитель:
ст. преп. Луцив Д. В.

Рецензент:
Главный конструктор ЗАО "Ланит-Терком" Вояковская Н.Н.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY
Software and Administration of Information Systems

Software Engineering

Stolpner Lev

Detection of fuzzy repetitions in formatted texts

Bachelor's Thesis

Scientific supervisor:
senior lecturer Dmitry Luciv

Reviewer:
Prime Designer Lanit-Tercom CJSC Natalia Voyakovskaya

Saint-Petersburg
2016

Оглавление

Введение

В ходе работы в последнее время нам все чаще приходится сталкиваться с книгами в электронном формате, материалами на веб-сайтах, промышленной документацией для программных продуктов, и многими другими электронными источниками информации. В частности, сложное программное обеспечение обычно сопровождается большим количеством подробной документации, которая содержит много сложноструктурированной информации.

Одной из важных особенностей электронной документации является наличие повторяющихся фрагментов текста, что значительно усложняет написание документа, так как в случае отсутствия дополнительных инструментов процесс его разработки и сопровождения может оказаться весьма трудоёмким и отнять немало времени. Также, если не отслеживать наличие повторов в тексте, становится вероятным понижение качества документа. Поэтому важной задачей становится упрощение и частичная автоматизация процесса поиска и рефакторинга таких повторов. Для её выполнения существуют такие решения, как DocLine[1][2][3] — проект, целью которого является создание инструмента для разработки и сопровождения электронной документации со значительными повторами в тексте. DocLine работает с достаточно популярным форматом DocBook[4] и его расширением DRL[5] и предоставляет возможности для нахождения имеющихся в тексте повторов и удобной работы с ними.

В большинстве случаев существенное количество повторов не являются дословными. Существующий и реализованный в DocLine на данный момент алгоритм[6] позволяет искать в тексте точные и вариативные повторы с одним фрагментом текста в середине. Алгоритм комбинирует находящиеся рядом точно повторяющиеся фрагменты текста. Последние ищутся при помощи реализованного с использованием инструмента Clone Miner[7][8] алгоритма на основе суффиксных массивов[9]. Clone Miner предназначен, в первую очередь, для поиска повторов в исходных тестах программ. Изменения при копировании

текста программ обычно представлены небольшим количеством достаточно крупных локализованных правок. Для текстов на естественных языках характерны как содержательные, так и стилистические правки после копирования, которые приводят к большому количеству мелких изменений. В случае текстов, таким образом, мы сталкиваемся со следующими ограничениями подхода[10]: возможен поиск нечётких клонов только с одной вариативной частью, рассматриваются кандидаты клонов только с промежутком в середине клона (не в начале или конце). Соответственно, возникает необходимость в функциональности, позволяющей быстро находить значительные нечёткие повторы в тексте, и которая не имела бы описанных выше ограничений.

Так как работа направлена на нахождение нечётких повторов в тексте документации, естественно при создании алгоритма было использовать существующие методы для нечёткого поиска. В ходе дипломной работы был использован и модифицирован для нечёткого сравнения алгоритм Рабина-Карпа[11] с использованием хеширования по сигнатуре[12][13] для быстрого сравнения фрагментов текста, а так же улучшенный алгоритм вычисления редакционного расстояния[14] между двумя фрагментами текста.

Постановка задачи

Целью данной дипломной работы является создание алгоритма для поиска нечётких повторов, не имеющего ограничений предыдущего подхода и с большей вариативностью. Для достижения этой цели в рамках работы были сформулированы следующие задачи:

1. Изучить средства нечёткого поиска и сопоставления текстов, научные работы и алгоритмы в области нечёткого поиска.

2. Предложить и реализовать новый алгоритм с использованием методов нечёткого поиска.

3. Провести эксперименты и тестирование алгоритма.

1. Обзор

1.1. Средства нечёткого поиска и сопоставления текстов

1.1.1. Технология DocLine

DocLine – это технология, позволяющая создавать и сопровождать документацию семейств программных продуктов с возможностью вариативного использования повторяющихся частей. Для представления документации DocLine использует XML-язык разметки DRL. DRL является расширением DocBook, основными изменениями по сравнению с последним являются новые конструкции для вариативного повторного использования: каталог элементов и информационный элемент. DocBook, в свою очередь, представляет собой инструмент, изначально предназначенный для создания объёмной технической документации со сложной структурой.

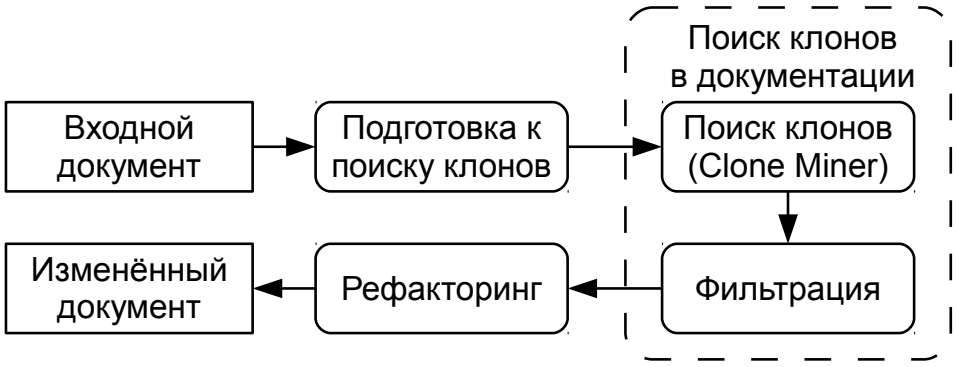

Рис. 1: Поиск клонов в DocLine

Для решения данной задачи в DocLine используется инструмент Documentation Refactoring Toolkit[15], имеющий графический интерфейс и позволяющий выбирать необходимый для редактирования файл, задавать определенные параметры и за некоторое время находить нечёткие повторяющиеся фрагменты документации в этом файле.

На Рис. 1 изображен алгоритм работы с документацией в DocLine: на вход подается файл, пользователь задает минимальную длину кло-

на, способ проверки и восстановления разметки, другие дополнительные опции, затем производится удаление разметки, токенизация. Клоны в тексте ищутся с помощью CloneMiner, затем осуществляется их фильтрация, после этого пользователь проводит их рефакторинг.

На данный момент алгоритм позволяет искать в тексте вариативные повторы, комбинируя четкие клоны, найденные CloneMiner, и обладает двумя существенными ограничениями: нечёткие клоны имеют только одну различающуюся часть, и эта часть может находиться только в середине клона, что сужает количество и вариативность находимых нечётких повторов.

1.1.2. Обнаружение клонов в естественных языках

Похожих целей пытаются достичь различные средства для обнаружения повторов в естественных языках. Существует много различных детекторов повторов[16][17][18][19], но в большинстве случаев они представляют собой инструменты для стилистического анализа, а не для нахождения крупных нечётких повторов. В основном, с помощью поиска четких клонов они выдают информацию об использовании конкретных слов, словосочетаний, речевых оборотов или предложений, и предоставляют статистику по количеству найденных совпадений.

1.1.3. Обнаружение клонов в программном обеспечении

Также можно пытаться находить нечёткие повторы в тексте с помощью детекторов клонов в программном обеспечении[20][21][22][23], как это делается в текущей версии DocLine с использованием инструмента CloneMiner, который ищет точные повторы. При работе с кодом в большом проекте нахождение клонов становится очень важной задачей, так как несколько исследований показало, что примерно 20-30%[24][25] кода больших программных систем повторяются. Такие клоны обычно делят на два типа[8]: простые и структурные. При поиске повторяющихся фрагментов в коде используются различные подходы[11]: текстовый, лексический, синтаксический и семантический. Однако, основным

направлением работы таких средств является поиск клонов в языках программирования, а не естественных языках, что представляет собой абсолютно другую задачу, так как в поиске дубликатов кода продуктивнее использовать более высокоуровневые подходы, чем текстовый.

1.1.4. Обнаружение плагиата в научных работах

Инструменты для нахождения плагиата[26][27][28][29][30] в научных работах также решают схожие задачи. Основное их предназначение – поиск повторяющихся речевых конструкций и предложений, иногда – с использованием семантического анализа для учета синонимов, однако в целом такие средства не приспособлены для нахождения объемных вариативных повторяющихся фрагментов с множеством семантически не обусловленных различий между клонами.

1.2. Алгоритмы нечёткого поиска

При поиске нечётких повторов в документации естественно рассмотреть различные алгоритмы нечёткого поиска, позволяющие сопоставлять фрагменты текста не точно, а с каким-то количеством несовпадений между ними.

Одним из важных шагов в нечётком поиске является выбор функции похожести строк[31]. Выбор подходящей функции непосредственно влияет на качество и скорость поиска. Хорошая функция определения близости слов позволяет учитывать разные типы искажений, такие как удаления, вставки, замены и транспозиции соседних символов. Одной из первых предложенных мер близости слов является функция вычисления расстояния Левенштейна[32], или редакционного расстояния. Расстояние Левенштейна равняется минимальному количеству операций редактирования, позволяющих преобразовать одну строку в другую, а именно операции удаления, вставки и замены одного символа. Модификация этого расстояния редактирования была предложена Дамерау[33], и включает в множество операций редактирования транспозицию символов. Алгоритмы вычисления редакционного

расстояния[34][35][36][37][38] изучаются уже давно и являются основой многих других алгоритмов нечёткого поиска.

В целом все виды нечёткого поиска и сравнения строк можно разбить на две основные категории[39]: с предварительной индексацией текста (offline), и без предварительной индексации (online). Методы с индексацией обычно подразумевают подсчет индекса от текста и его хранение, что занимает дополнительную память, но может значительно уменьшить время работы алгоритма непосредственно при сравнении фрагментов.

1.2.1. Алгоритмы с индексацией

В области алгоритмов с индексацией существует множество различных подходов, таких как хеширование по сигнатуре[12][13], n-граммная индексация[40][41][42], основанная на индексации отрезков фиксированной длины, kd-деревья[43] и rd-деревья[44], алгоритмы поиска в метрических пространствах[45], trie-деревья[46], алгоритмы локально устойчивого хеширования[47][48], sketching-алгоритмы[49][50]. Основной проблемой таких алгоритмов являются существенные затраты по памяти и времени для построения индекса. Так как в нашей задаче фактически требуется обнаружение клонов в реальном времени, и исходный документ может меняться довольно часто, то предварительная индексация целого документа не является целесообразной и будет естественнее использовать алгоритмы без индексации.

1.2.2. Алгоритмы без индексации

Среди online-алгоритмов было рассмотрено множество модификаций классических алгоритмов, таких как алгоритм Рабина-Карпа[11], Кнута-Морриса-Пратта[51] или Бойера-Мура[52]. Также для нечёткого сравнения используются алгоритмы, основанные на методе динамического программирования Вагнера-Фишера[34], алгоритм bitap[53] и его модификация от Wu-Manber[54]. Основной трудностью при использовании методов без индексации стало бы то, что при прямом использовании

этих методов (при сравнении фрагментов текста разного размера между собой) конечная сложность алгоритма становилась бы квадратичной и его работа была бы малопродуктивной.

1.3. Используемые технологии и программные средства

1.3.1. Платформа .NET и язык программирования C#

Для работы была выбрана платформа .NET и язык программирования C#, так как они позволяют удобно работать с документами в формате XML, предоставляют достаточно гибкие возможности для разработки алгоритма, его реализации и распараллеливания, а также большой комплект библиотек, упрощающих работу по нормализации текста.

1.3.2. Библиотека лемматизации

Подключаемая библиотека LemmaGen[55][56], доступная при разработке на платформе .NET, предназначена для работы по лемматизации текста, что является важным моментом при поиске повторяющихся фрагментов, так как без её использования многие повторяющиеся фрагменты были бы не учтены из-за того, что слова в разной форме считались бы не совпадающими. Библиотека позволяет разработчику самому генерировать свои лемматизаторы для разных языков, и также предоставляет набор уже сгенерированных для нескольких базовых языков.

1.3.3. Библиотека стемминга

Также подключаемая при работе в .NET библиотека StemmersNet[57] предоставляет функционал для быстрого стемминга слов в тексте, используя стеммер Портера[58][59]. Использование этой библиотеки позволяет слова в разных частях речи приводить к

общему основанию, также расширяя количество находимых при поиске повторяющихся фрагментов.

2. Предварительная обработка текста

2.1. Управление форматированием

Первым этапом при поиске повторяющихся фрагментов в XML-документе является работа с форматированием текста. Задача состоит в том, чтобы перевести документ из исходного формата в текстовый, это позволило бы избежать некоторых сложностей: во-первых, попадание XML-разметки в находимые алгоритмом повторяющиеся фрагменты и необходимость последующей фильтрации этой разметки, во-вторых, ненахождение алгоритмом существующих клонов из-за разницы в окружающей их разметке. С помощью предоставляемой платформой .NET библиотеки[60] была реализована возможность перехода от XML-формата к текстовому.

2.2. Нормализация

Следующим шагом для улучшения результатов алгоритма является нормализация[61] текста, полученного после изменения форматирования. При поиске клонов в ненормализованном тексте многие похожие элементы могут оказаться невыявленными из-за того, что слова в них могут находиться в разной форме, иметь различные окончания или корень. Чтобы избавиться от этих факторов, необходимо предварительно произвести нормализацию текста.

Первой ступенью нормализации текста является лемматизация[55][56][62]. Лемматизация - процесс группирования различных форм слова и приведения их к одной базовой форме - лемме. Суть процесса состоит в определении леммы для каждого слова отдельно, и, затем, замене всех слов исходного текста на определенные леммы. Этот шаг позволит увеличить количество находимых нечётких повторов, так как теперь алгоритм становится менее восприимчив к особенностям языка.

Второй шаг в нормализации текста - стемминг[63]. Этот процесс представляет более простой и быстрый алгоритм по обработке текста,

заключающийся в сокращении исходного слова до его основы, которая довольно часто не является самостоятельным значимым словом. Цель такой обработки схожа с целью лемматизации, однако алгоритм основывается лишь на некотором наборе правил по отсечению суффиксов слов, основывающихся на особенностях конкретного языка. В данном алгоритме используется стемминг Портера[58][59][64].

Таблица 1: Пример работы лемматизации и стемминга

Исходный текст	”Lemmatization is the process of grouping together the different inflected forms of a word”
Лемматизация	”lemmatization be the process of group together the different inflect form of a word”
Стемминг	”lemmat be the process of group togeth the differ inflect form of a word”

3. Алгоритм поиска нечётких повторов

Теперь необходимо искать нечёткие повторы в предобработанном с помощью лемматизации и стемминга тексте. Фактически эта задача состоит в сравнении полученного текста с самим собой, и нахождению таким образом неточно повторяющихся фрагментов с помощью алгоритмов нечёткого поиска. Изначально размер клонов неизвестен и может варьироваться, вследствие чего при прямолинейной реализации этого подхода пришлось бы сравнить между собой все фрагменты текста всех размеров. Сравнение между собой всех фрагментов одного размера имеет сложность $O(n^2/t^2)$, где n - количество слов в тексте, t - размер фрагмента в словах, так как количество фрагментов равняется n/t, и относительно него сложность алгоритма квадратичная. Посчитаем общую сложность алгоритма без учета стоимости сравнения двух фрагментов текста между собой: $\sum_{t=1}^{n} n^2/t^2 = n^2 * (1+1/2+...+1/n)^2 = O(n^2)$. Следовательно, вне зависимости от выбора алгоритма сравнения этих фрагментов, такой подход был бы крайне трудоемким и непродуктивным.

3.1. Построение алфавита

Прежде всего было решено построить по нормализованному тексту алфавит, символами которого будут являться слова, встречающиеся в тексте, и затем хранить текст в символах нового алфавита. Это преобразование предоставляет несколько преимуществ: уменьшение объема памяти, необходимой для хранения текста, а также ускорение дальнейшей работы алгоритма с текстом, так как появляется возможность работать не со строковыми значениями слов, а с их позицией в созданном алфавите, что позволит сравнивать фрагменты быстрее. После получения результатов работы алгоритма нечёткого поиска достаточно будет использовать алфавит, чтобы вернуться обратно к текстовому представлению. Однако, это преобразование так же не дает значительного прироста в производительности алгоритма. Поэтому было решено изменить способ сравнения фрагментов.

3.2. Фрагментация текста

Первый шаг предлагаемого алгоритма - изначально делать предположение о минимальном интересующем пользователя размере клона. Поскольку речь идет о работе с документацией большого размера, можно считать, что нечёткие повторы размером в несколько слов вряд ли будут являться основным объектом поиска и рефакторинга и могут быть легко найдены точным поиском, следовательно, можно задать ограничение на минимальный размер повторяющегося фрагмента.

Теперь идею о сравнении текста с самим собой можно модифицировать, используя предыдущее утверждение. Следующим шагом алгоритма будет разбиение текста на фрагменты заданного размера. Затем будет проводиться их нечёткое сравнение друг с другом. Так как в этом случае размер фрагмента уже зафиксирован, значительная часть сравнений отбрасывается. Однако, нечёткие клоны могут иметь намного больший размер, чем один такой фрагмент, вследствие чего одним из последующих этапов работы алгоритма будет расширение найденных клонов. При таком подходе сложность алгоритма без учета сложности сравнения фрагментов составит $O(n^2/t^2)$, где n - количество слов в тексте, t - минимальный размер клона.

3.3. Сравнение фрагментов

Следующим важным шагом алгоритма является использование идеи алгоритма Рабина-Карпа[11] об использовании хеширования для ускорения сравнения фрагментов. Но, поскольку требуется искать неточные совпадения между фрагментами, необходимо использовать перцептивное хеширование, которое будет выдавать близкие значения на похожих фрагментах, и позволит отбрасывать заведомо непохожие фрагменты.

3.3.1. Хеширование

Было решено модифицировать и использовать хеш-функцию, применяющуюся в хешировании по сигнатуре[13]. Оригинальная хеш-функция сопоставляла бы каждому фрагменту вектор размера m, называемый сигнатурой. Каждому i-ому элементу данного вектора соответствует набор символов алфавита, и если фрагмент содержит один из этих символов, то i-ый элемент будет равняться единице, и нулю в противном случае. Однако, в данном случае такое применение функции нецелесообразно, так как при значительных размерах фрагмента их сигнатуры всегда будут иметь близкие значения даже для абсолютно непохожих фрагментов. Также неэффективно было бы пытаться модифицировать функцию, используя вместо наборов символов алфавита наборы слов - символов нового алфавита, имеющего намного больший размер, что привело бы к одной из двух проблем: при попытке сохранить точность хеширования и увеличить размер хранимого вектора затраты по памяти на хранение хеша были бы слишком существенны, а при попытке сохранить размер вектора и значительно увеличить размер наборов символов, соответствующих значению элементов в нем, сильно пострадала бы точность хеширования.

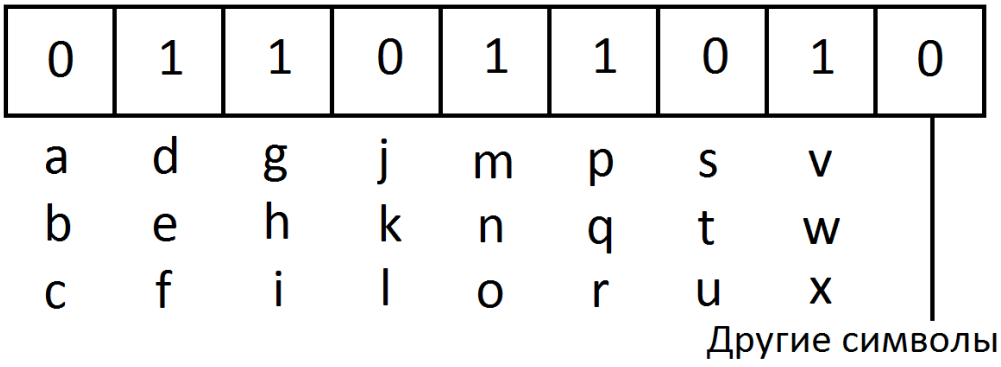

Рис. 2: Сигнатурная хеш-функция

Поэтому была предложена сигнатурная хеш-функция, наборами значений для элементов вектора в которой являлись бы символы языка документации, но теперь i-ый элемент вектора будет равняться единице, только если символ из набора является первым символом одного

из слов, встречающихся во фрагменте. Это позволит сохранить затраты по памяти на прежнем уровне, не потеряв при этом точность. В таком случае значением хеш-функции фрагмента будет являться число $H(w) = \sum_{i=0}^{m-1} 2^i * sign(w)_{i+1}$, где w - фрагмент, sign(w) - вектор-сигнатура размера m по первым символам слов во фрагменте. Как указано на Рис. 2, наиболее подходящим оказался вектор с количеством элементом $m = 9$.

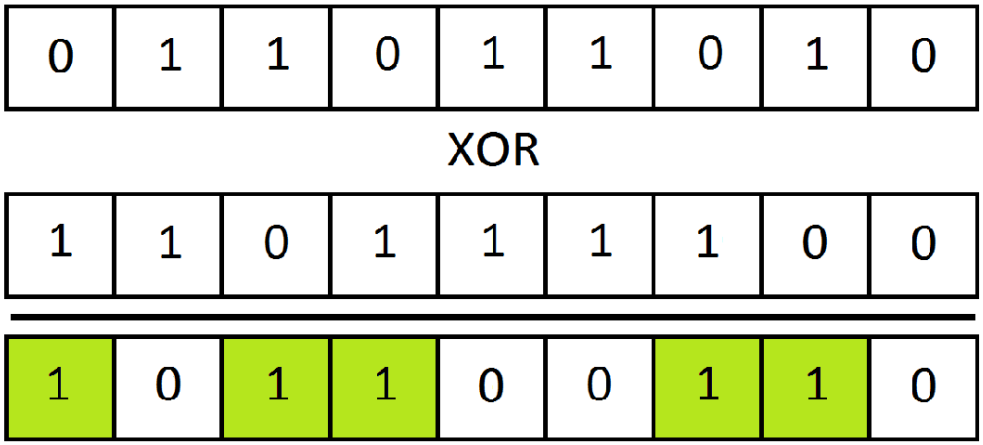

Рис. 3: Сравнение хеш-значений фрагментов

В целом сравнение фрагментов друг с другом теперь состоит из двух шагов. Сначала сравниваются их значения хеш-функций. На Рис. 3 представлена схема быстрого сравнения значений хеш-функции разных фрагментов. Было решено воспользоваться тем, что изначально значение представлялось в виде вектора из нулей и единиц. Значения хеш-функции двух фрагментов переводятся в двоичное представление, и побитово используется операция исключающего "ИЛИ". В побитовом представлении числа, полученного данным преобразованием, количество единичных битов будет обозначать разницу между значениями хеш-функций двух фрагментов. В случае, если эта величина не превышает определенное пороговое значение, задаваемое при запуске алгоритма, то производится второй шаг сравнения.

3.3.2. Вычисление редакционного расстояния

Как показано на Рис. 4, непосредственное сравнение фрагментов производится с помощью вычисления редакционного расстояния между ними, то есть, количества операций вставки, замены и удаления для получения одного фрагмента из другого. Существует множество подходов к его вычислению, в этом алгоритме основными ограничениями при выборе определенного подхода будет являться то, что он не должен требовать значительного использования дополнительной памяти, ускорять сравнение за счет предварительных вычислений и работать менее, чем за квадратичное время, которое предоставляет алгоритм динамического программирования, предложенный Вагнером и Фишером. В итоге, наиболее подходящей оказалась модификация Укконеном метода динамического программирования[14] со сложностью $O(k * t)$, где k - максимально допустимое количество операций редактирования для получения одного фрагмента из другого, а t - количество слов во фрагменте. Основной идеей его алгоритма заключается введение понятия графа зависимостей, преимущество которого состоит в том, что можно не вычислять многие лишние значения в матрице подсчета редакционного расстояния. Конечная сложность алгоритма с использованием подхода Укконена - $O(n^2 * k/t)$, где n - количество слов в тексте, k $\leq$ t.

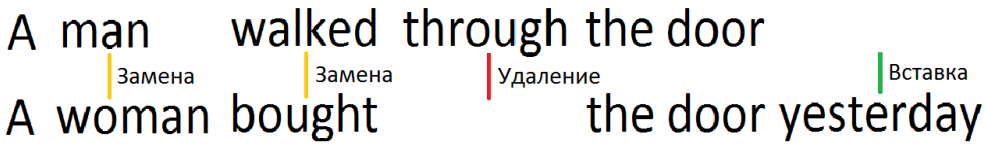

Рис. 4: Редакционное расстояние между фрагментами

Изначально был также реализован и апробирован классический алгоритм Вагнера-Фишера[34], который оказался слишком трудоемким, так как сложность самого сопоставления фрагментов в нем $O(t^2)$, где t-размер фрагмента, и общая сложность алгоритма составляла бы $O(n^2)$. Также испробованный алгоритм вычисления расстояния Хемминга[65] между двумя фрагментами представлял лучшую асимптотическую сложность алгоритма $O(n^2/t)$, однако, в конечном счете, плохо подхо-

дит для используемой в алгоритме фрагментации, так как не учитывает значительное количество нечётких повторов.

3.4. Группировка и расширение клонов

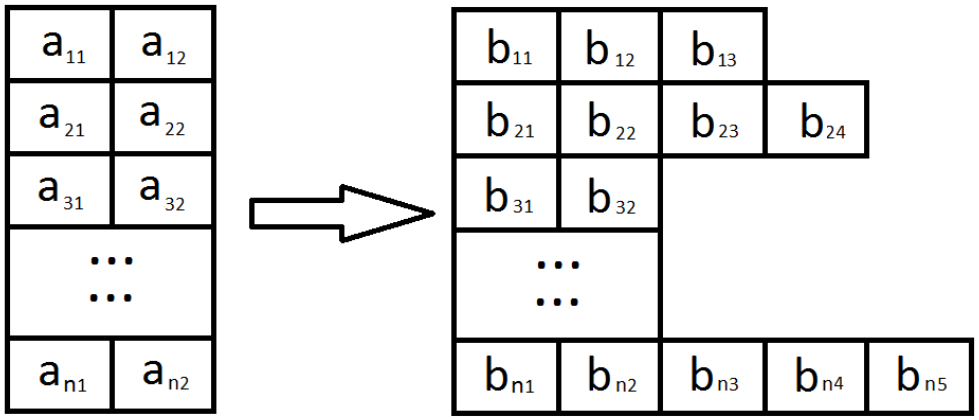

Рис. 5: Группирование схожих фрагментов

После того, как были найдены все пары повторяющихся фрагментов, необходимо провести объединение всех результатов, объединив похожие клоны в группы большего размера, как указано на Рис. 5.

Затем необходимо провести расширение клонов в группах от минимального размера клона до максимально возможного, как это изображено на Рис. 6, где m - номер группы клонов, a i, j, k - номера различных клонов в группе. Однако, если осуществлять расширение отдельно для каждой из полученных групп, проводя постепенное сравнение соседних фрагментов клонов в группе с последующим их присоединением к имеющимся фрагментам, в результате работы алгоритма получится множество групп клонов со значительными пересечениями. Это будет являться следствием того, что после сравнения фрагментов текста друг с другом уже были найдены все возможные совпадающие элементы, и добавление в группы новых лишь приведет к увеличению пересечений между ними.

Следовательно, необходимо произвести сравнение полученных групп клонов между собой и расширение клонов путем их соединения и последующего отбрасывания пересечений между группами. При об-

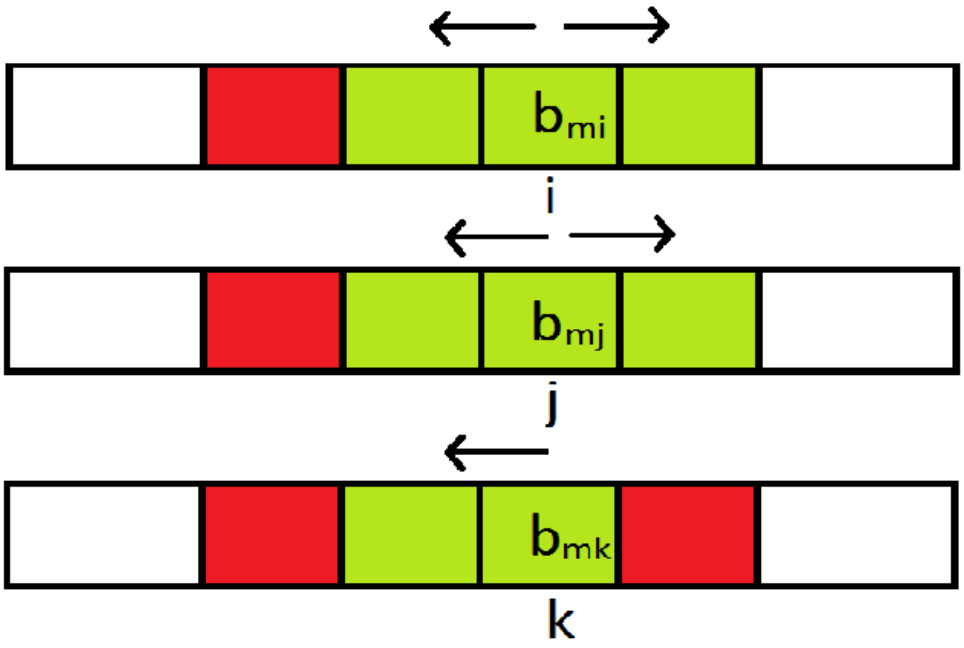

Рис. 6: Расширение групп клонов

наружении пересечения между клонами различных групп необходимо произвести удаление одного из них. Для определения того, какой из них должен быть исключен, было решено ввести функцию значимости клона, и клон, который который принимает на ней меньшее значение, удаляется из своей группы. Была выбрана функция $m * n^2$, где m– количество клонов в группе, к которой он принадлежит, а n – размер самого клона в количестве фрагментов, из которых он состоит. Полученные таким путем расширенные клоны будут представлять собой конечные результаты работы алгоритма. Заключительным этапом его работы является пересчет координат найденных клонов относительно исходного XML-документа, и их визуализация.

4. Особенности реализации

Алгоритм подразумевает возможность настройки и изменения некоторых параметров пользователем. Он может настроить такие параметры, как размер фрагментов, на которые разбивается текст, максимально допустимое редакционное расстояние между двумя фрагментами, а также порог близости двух значений хеш-функции для разных фрагментов.

Также реализация алгоритма предусматривает возможность распараллеливания на нескольких этапах работы алгоритма: нормализации текста, вычислении хешей фрагментов, сравнении фрагментов текста и их расширении, что позволит еще ускорить работу алгоритма.

5. Эксперименты

5.1. Формирование тестовых данных

Для проверки корректности работы алгоритма, а также с целью нахождения его ограничений, было сформировано несколько наборов тестовых данных:

1) Набор данных для проверки корректности алгоритма, состоящий из нескольких небольших XML-текстов

2) Данные для формирования статистики, получаемые при запуске алгоритма на документации крупного проекта

5.1.1. Данные для проверки корректности

Данные для тестов состоят из нескольких небольших фрагментов текста, содержащих повторяющиеся элементы. Проведение данных тестов призвано продемонстрировать работу нового алгоритма на различных типах клонов, в частности, доказать, что удалось избежать ограничений алгоритма, используемого в DocLine, и что новый алгоритм расширил диапазон распознаваемых клонов за счет увеличения их вариативности.

В качестве тестовых данных было решено использовать следующие фрагменты XML-документов:

1) Фрагмент, текстовая часть которого представляет собой два идентичных предложения, с целью выяснить, может ли алгоритм находить точные повторы в тексте

2) Фрагмент, представляющий собой пример клона, распознаваемого алгоритмом поиска в DocLine, с двумя точными клонами в начале и в конце, между которыми находится вариативная часть

3) Несколько фрагментов, содержащих точные клоны с двумя и более вариативными частями между ними

4) Фрагмент текста, позволяющий проверить возможность алгоритма находить несколько групп клонов

5.1.2. Данные для формирования статистики

Данные для формирования статистики были сформированы для получения количественной оценки эффективности алгоритма при различных входных параметрах, таких как минимальный размер клона и максимальное редакционное расстояние при сопоставлении двух фрагментов. В качестве документации была использована документация ядра популярной операционной системы Linux[66].

5.2. Схема экспериментов

Тестирование алгоритма проводилось в два этапа.

На первом этапе проверялась работа алгоритма на наборе тестов для проверки его корректности с конкретно заданными параметрами. Условием успешного прохождения теста являлось нахождение алгоритмом всех предполагаемых нечётких повторов, и отсутствие побочных результатов с фрагментами, не являющимися клонами.

На втором этапе проверялась работа алгоритма на документации в формате XML при нескольких наборах изначальных параметров, и замерялось несколько количественных характеристик: количество найденных групп клонов, средний размер группы и среднее количество токенов в клоне безотносительно к тому, в какой группе он находится.

5.3. Анализ результатов

5.3.1. Проверка корректности алгоритма

Было проведено 4 ключевых теста, успешное прохождение которых означает, что алгоритм позволяет находить как точные, так и нечёткие повторы разной степени вариативности, и позволяет преодолеть ограничения алгоритма, используемого в DocLine.

1) Проверка нахождения алгоритмом точных клонов:

<para>All work and no play makes Jack a dull boy.</para>

<para>All work and no play makes Jack a dull boy.</para>

В данном фрагменте алгоритм в качестве результата выдает 1 группу клонов длиной 10 токенов.

2) Проверка нахождения алгоритмом нечётких повторов, находимых алгоритмом поиска клонов DocLine:

<para>Just an identical part of a bit different small phrases.</para>

<para>Just an identical part of absolutely same small phrases.</para>

В данном фрагменте алгоритм в качестве результата также выдает 1 группу клонов с длиной клона равной 10.

3) Проверка нахождения алгоритмом нечётких повторов с двумя и более вариативными частями:

<para>Two paragraphs with two different parts in text.</para>

<para>Two phrases with few different parts of text.</para>

В данном фрагменте алгоритм в качестве результата найдет 1 группу клонов одинаковой длины.

<para>Two a bit different paragraphs. Algorithm should recognize this phrases as a clone</para>

<para>Two a little similar paragraphs. Algorithm can recognize that this phrases has a clone</para>

В этом примере в качестве результата также будет представлена 1 группа из двух клонов, полученная из двух предложений.

4) Проверка нахождения алгоритмом нескольких групп нечётких повторов:

<para>There are few same parts, middle is just different and has nothing in common, but the end is the same.</para>

<para>There are few same parts, part between beginning and the end is completely diverse, but the end is the same.</para>

В данном тесте алгоритм при относительно небольшом входном параметре размера фрагмента не сможет объединить два повторяющихся фрагмента в начале и в конце предложения. Поэтому в качестве результата будет выдано две группы клонов с длиной клона 5.

5.3.2. Формирование численных оценок и статистики

В Таблице 2 наглядно представлены количественные результаты работы алгоритма при 8 наборах входных параметров: n - размер минимального размера клона, задаваемый при фрагментации текста, k - максимальное редакционное расстояние между фрагментами при их сравнении друг с другом. Все тесты проведены на одном экземпляре документации ядра Linux объёмом 912KB. Результаты сравнивались по трём параметрам:

- количество найденных групп клонов,

- средний размер группы (среднее количество клонов),

- средняя длина клона в токенах (среднее количество слов в клоне по всем группам)

Таблица 2: Результаты работы алгоритма на документации Linux

Параметры	Количество групп	Размер группы	Средняя длина клона
n=5, k=2	1044	3	12
n=10, k=5	100	3	44
n=20, k=5	31	6	28
n=20, k=10	65	5	28
n=40, k=15	15	8	42
n=40, k=20	20	8	43
n=50, k=15	9	7	61
n=50, k=25	10	12	50

Из Таблицы 2 видно, как сильно меняются результирующие показатели в зависимости от входных параметров. При увеличении минимального размера клона количество групп постепенно уменьшается, при этом повышается средний размер группы клонов и, соответственно, средняя длина клонов так же увеличивается. При росте параметра редакционного расстояния относительно размера фрагмента можно наблюдать существенное увеличения числа находимых групп, причем размер групп в среднем остается прежним. За исключением примера с параметрами n=10, k=5, видно, что средняя длина клона обычно незначительно превышает размер фрагмента, однако именно в этом тесте расширение клонов помогло сильно увеличить размеры находимых по-

второв.

В целом, можно сделать вывод, что алгоритм является достаточно гибким для нахождения клонов различных размеров, позволяет работать как с небольшими, так и с крупными экземплярами XML-документации и находить существенное количество повторяющихся фрагментов. Самостоятельная настройка и регулировка пользователем указанных выше параметров алгоритма перед его запуском предоставляет возможность постепенного уточнения и балансировки необходимых результатов.

Заключение

В рамках данной работы были достигнуты следующие результаты:

1. Изучены средства нечёткого поиска и сопоставления текстов, научные работы и алгоритмы в области нечёткого поиска.

2. Предложен новый алгоритм с использованием модификаций алгоритма Рабина-Карпа, хеширования по сигнатуре и алгоритма вычисления редакционного расстояния. Алгоритм реализован.

3. Проведены эксперименты и тестирование алгоритма.

Список литературы

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

[2] Minchin L. Romanovsky K., Koznov D. *Refactoring the Documentation of Software Product Lines.* Lecture Notes in Computer Science. 2011. Vol. 4980 c. 158-170, 2011.

[3] Кознов Д.В. Шутак А.В. Смирнов М.Н., Смажевский М.А. *Поиск клонов при рефакторинге технической документации.* Компьютерные инструменты в образовании. 2012. № 4. С. 30-40., 2012.

[4] Walsh N. Muellner L. *DocBook: The Definitive Guide.* O'Reilly Media, 2003.

[5] Романовский К.Ю. Кознов Д.В. *Язык DRL для проектирования и разработки документации семейства программных продуктов.* Вестник С.-Петербург. ун-та. Сер. 10.2007. Вып. 4. С. 110-122., 2007.

[6] М.Н. Смирнов Д.В.Луцив К.Ю. Романовский Д.В.Кознов Х.А. Басúт, О.Е.Ли. *Метод поиска повторяющихся фрагментов текста в технической документации.* Научно-технический вестник информационных технологий, механики и оптики./ Вып. 4 (92), СПб НИУ ИТМО 2014. С. 106-114., 2014.

[7] Jarzabek S. Basit H. A. *A Data Mining Approach for Detecting Higher-Level Clones in Software.* IEEE Transactions on Software engineering, Vol. 35, No. 4, 2009., 2009.

[8] Jarzabek S. Basit H. A. *Detecting higher-level similarity patterns in programs.* ACM Sigsoft Software engineering notes, 2005.

[9] G. Myers U. Manber. *Suffix arrays: A new method for on-line string*

searches. Department of Computer Science University of Arizona Tucson, 1989.

[10] Hamid Abdul Basit Ouh Eng Lieh Mikhail Smirnov Dmitrij Koznov, Dmitry Luciv. *Clone Detection in Reuse of Software Technical Documentation.* 10th Ershov Informatics Conference (PSI '15), 2015.

[11] S. Rochimah R. D. Dewandono, F. A. Saputra. *Clone detection using Rabin-Karp parallel algorythm.* Department of Informatics, InstitutTeknologiSepuluhNopember, Surabaya, 60111, 2013.

[12] Л. М. Бойцов. *Классификация и экспериментальное исследование современных алгоритмов нечеткого словарного поиска.* The 6th Russian Conference on Digital Libraries (RCDL'04), 2004.

[13] 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.

[14] E. Ukkonen. *Algorithms for approximate string matching.* Department of Computer Science, University of Helsinki, 1985.

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

[16] Repetition detector 2, 2015. http://www.repetition-detector.com/.

[17] Textanz, 2011. http://www.textanz.com/.

[18] Prowritingaid, 2016. https://prowritingaid.com/.

[19] Text analyzer. http://www.online-utility.org/text/analyzer.jsp.

[20] Clone digger, 2016. http://clonedigger.sourceforge.net/.

[21] Clone doctor, 2016. http://www.semdesigns.com/products/clone/.

[22] iclones, 2016. http://www.softwareclones.org/iclones.php.

[23] Clone detection, 2016. http://patterninsight.com/products/clone-detection/.

[24] Merlo E Mayrand J., Leblanc C. *Experiment on the automatic detection of function clones in a software system using metrics.* International Conference on Software Maintenance, 1996.

[25] B. S. Baker. *On finding duplication and near-duplication in large software systems.* 2nd Working Conference on Reverse Engineering, 1995.

[26] Plagiarisma, 2016. http://plagiarisma.net/.

[27] Quetext, 2016. http://www.quetext.com/.

[28] Viper plagiarism scanner, 2016. http://www.scanmyessay.com/plagiarism/.

[29] Paper rater, 2016. http://www.paperrater.com/.

[30] Plagium, 2016. http://www.plagium.com/.

[31] Wikipedia. Edit distance. Wikipedia, free encyclopedia, 2016. https://en.wikipedia.org/wiki/Edit_distance.

[32] Wikipedia. Levenshtein distance. Wikipedia, free encyclopedia, 2016. https://en.wikipedia.org/wiki/Levenshtein_distance.

[33] Wikipedia. Damerau–levenshtein distance. Wikipedia, free encyclopedia, 2016.

[34] Fischer M. Wagner R. *The String-to-String Correction Problem.* 1974.

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

[36] Ukkonen E. *Algorithms for Approximate String Matching.* Information and Control 64, 1985.

[37] Ukkonen E. *Finding approximate patterns in strings.* Journal of algorithms 6, 1985.

[38] Myers E. *An O(ND) Difference Algorithm and Its Variations.* Department of Computer Science, University of Arizona, 1986.

[39] Бойцов Л.М. *Классификация и экспериментальное исследование современных алгоритмов нечеткого словарного поиска.* The 6th Russian Conference on Digital Libraries, 2004.

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

[41] L. Boytsov. *Indexing Methods for Approximate Dictionary Searching: Comparative Analysis.* ACM Journal of Experimental ALgorithmics, 2011.

[42] J.S. Culpepper M. Petri. *Efficient Indexing Algorithms for Approximate Pattern Matching in Text.* ADCS'12 Proceedings of the Seventeenth Australasian Document Computing Symposium, 2012.

[43] J.L. Bentley. *Multidimensional Binary Search Trees Used for Associative Searching.* Communication of the ACM, 1975.

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

[45] 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.

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

[47] M. Charikar. *Similarity Estimation Techniques from Rounding Algorithms.* STOC'02 Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, 2002.

[48] The simhash algorithm, 2009. http://matpalm.com/resemblance/simhash/.

[49] A sketching algorithm, 2009. http://matpalm.com/resemblance/sketching/.

[50] The simhash algorithm, 2010. http://lkozma.net/blog/sketching-data-structures/.

[51] V. Pratt D. Knuth, J. Morris. *Fast pattern matching in strings.* SIAM Journal on Computing, 1977.

[52] E. Ukkonen J. Tarhio. *Approximate Boyer-Moore string matching.* SIAM Journal on Computing, 1993.

[53] G. Gonnet R. Baeza-Yates. *A new approach to text searching.* Communications of the ACM, 1992.

[54] U. Manber S. Wu. *Fast text searching: allowing errors.* Communications of the ACM, 1992.

[55] Lemmagen, 2010. http://lemmatise.ijs.si/.

[56] T. Erjavec N. Lavrac M. Jursic, I. Mozetic. *LemmaGen: Multilingual Lemmatisation with Induced Ripple-Down Rules.* Journal of Universal Computer Science, 2010.

[57] Stemmersnet, 2012. https://stemmersnet.codeplex.com/.

[58] M. Porter. *An algorithm for suffix stripping.* Readings in information retrieval, 1997.

[59] Porter stemming algorithm, 2006. http://tartarus.org/martin/PorterStemmer

[60] System.xml namespace, 2016. https://msdn.microsoft.com/en-us/library/system.xml(v=vs.110).aspx.

[61] Stemming and lemmatization, 2009. http://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html.

[62] Lemmatisation, 2015. https://en.wikipedia.org/wiki/Lemmatisation.

[63] Stemming, 2016. https://en.wikipedia.org/wiki/Stemming.

[64] P. Willett. *The Porter stemming algorithm: then and now.* Program: electronic library and information systems, 2006.

[65] E. Porat A. Amir, M. Lewenstein. *Faster algorithms for string matching with k mismatches.* Journal of Algorithms 50, 2000.

[66] The linux kernel archives, 2016. https://www.kernel.org/.