Санкт-Петербургский государственный университет

Кафедра системного программирования
Программная инженерия

Мишин Никита Матвеевич

Реализация и применение строковых алгоритмов к задаче поиска повторов в документации программного обеспечения

Выпускная квалификационная работа

Научный руководитель:
к. ф.-м. н., доцент кафедры информатики СПбГУ Григорьев С. В.

Консультант:
к. ф.-м. н., Березун Д. А.

Рецензент:
д. ф. н., доцент Тискин А. В.

Санкт-Петербург
2020

Оглавление

Введение **4**

1. Постановка задачи **7**

2. Обзор **8**

2.1. Повторы в документации ПО 8

2.2. Модель повторов в документации ПО 11

2.3. Задачи поиска повторов в документации ПО 12

2.4. Полулокальные задачи поиска 13

2.4.1. Задачи поиска наибольшей общей подпоследовательности (LCS) и выравнивания строк (SA) . . . 13

2.4.2. Semi-local LCS и SA 14

3. Реализация библиотеки алгоритмов для полулокальных задач **21**

3.1. Архитектура библиотеки 21

3.1.1. Модуль semilocalProblem 21

3.1.2. Модуль semilocalApplication 22

4. Приложение для поиска повторов в документации ПО **26**

4.1. Общая архитектура приложения 26

4.2. Клиентская часть 27

4.3. Серверная часть 30

4.4. Алгоритмы для решения задачи поиска повторов 30

4.4.1. Улучшенный алгоритм интерактивного поиска . . 31

4.4.2. Алгоритм нечеткого поиска шаблона с использованием ThresholdAMatch 34

4.4.3. Алгоритм нечеткого поиска шаблона с использованием Разреза 35

4.5. Алгоритмы для решения задачи поиска групп повторов . 36

5. Апробация и анализ результатов **42**

5.1. Тестовый стенд 42

5.2. Экспериментальная проверка асимптотики 42
5.3. Поиск по шаблону 45
5.4. Поиск групп повторов 46

Заключение **48**

Список литературы **49**

Введение

На сегодняшний день существует множество программных продуктов, и их количество с каждым годом лишь увеличивается. Размер проектов исчисляется в строках кода и затраченных человеко-часах на соответствующую разработку. Программные продукты часто могут содержать миллионы строк кода, на написание которых затрачены миллионы человеко-часов[1]. Соответственно, разработка и сопровождение таких сложных систем немыслимы без документации, количество которой лишь увеличивается.

Важность сопроводительной документации не подвергается сомнению [18, 6]. Более конкретно, её качество напрямую влияет на жизненный цикл разработки системы, её конечную стоимость, время разработки, сопровождение и эксплуатацию и пр [22]. К критериям качества документации относятся точность, структурированность, последовательность изложения, понятность. Иными словами, задачи поддержания качества документации (всех ее критериев) на высоком уровне, а так же её написания, сопровождения и улучшения являются актуальными при создании, сопровождении и эксплуатации программных систем.

Как в программном коде, так и в документации могут появляться *текстовые повторы (текстовые клоны)*. Влияние *текстовых повторов* на документацию различно. К положительным факторам наличия *повторов* в документации можно отнести унификацию представления информации и создание общего контекста, что позволяет улучшить читаемость документации, её структурированность и передачу знаний (*knowledge transfer*). Несмотря на это, наличие клонов в документации влечет, как и в программном коде, распространение ошибок и опечаток. Также при наличии группы повторов, т.е множества похожих фрагментов текста, при внесении изменения в один из таких текстовых фрагментов необходимо произвести соответствующие изменения и в других "раскопированных" элементах во избежание нарушения консистентно-

[1]Код ядра линукса содержит более 27 млн строк кода, а на разработку ушло несколько сотен тысяч человеко-лет, *https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/, дата обращения 26.05.2020*

сти информации, что усложняет процесс сопровождения документации. Более того, существуют исследования, в которых показано, что клоны в документации могут привести к клонам в программном коде [5]. Соответственно, нахождение повторов в документации является актуальной проблемой, и текущие исследования лишь подтверждают утверждение [16, 23, 24, 5, 10].

Важным этапом при решении задачи поиска повторов является выбор того, как будет измеряться похожесть двух текстовых фрагментов и что при выбранном подходе считать клонами.

Применение строковых алгоритмов является наиболее естественным способом решения задачи поиска повторов как в произвольном тексте, так и в программной документации. Алгоритмы решения задач поиска наибольшей общей подпоследовательности (*LCS — longest common subsequence*) и выравнивания двух последовательностей (*SA — sequence alignment*) — широко известные алгоритмы, которые имеют разные приложения, в том числе и к задаче поиска повторов.

LCS и *SA* измеряют насколько текстовые фрагменты похожи глобально (в общем) друг на друга. Интуитивно понятно, что этого бывает недостаточно, потому что часто два фрагмента бывают схожи лишь небольшой общей частью. Решить эту проблему помогает обобщение на так называемый полулокальный случай, а именно, *полулокальные задачи наибольшей общей подпоследовательности и выравнивания последовательностей* (*semi-local lcs, semi-local sa*) [27].

Автором данного обобщения является А. В. Тискин. Он внес огромный вклад в развитие теории вокруг данных задач [32, 33, 20, 28, 30, 31]. В частности, им изобретены эффективные алгоритмы. Также им пишется труд [27], в котором собрана вся актуальная информация, связанная с этой теорией.

Несмотря на то, что алгоритмы имеют хорошие теоретические свойства, до конца неясно, насколько они применимы на практике к задаче поиска повторов, но относительно недавние успехи в применении этих алгоритмов в области биоинформатики [8, 2, 13] дают все основания полагать, что их можно успешно адаптировать к задаче поиска повторов

в документации. Важно отметить, что большая часть алгоритмов еще ни разу не была реализована на практике.

В данной работе представлено решение задачи поиска повторов в документации с помощью применения и адаптации алгоритмов решения задач *semi-local lcs* и *semi-local sa*. Для оценки применимости решения была осуществлена реализация приложения и библиотеки алгоритмов. Апробация результатов произведена на API-документации.

1. Постановка задачи

Целью данной дипломной работы является адаптация алгоритмов решения полулокальных задач поиска наибольшей общей подпоследовательности и выравнивая строк к задачам поиска повторов в документации ПО. Для достижения этой цели были сформулированы следующие задачи.

- Исследовать существующие теоретические алгоритмы решения задач полулокального поиска наибольшей общей подпоследовательности и выравнивания строк и реализовать их на практике в виде *библиотеки алгоритмов*.

- Адаптировать алгоритмы решения полулокальных задач поиска *lcs* и *sa* к задаче поиска повторов в *JavaDoc* документации и реализовать соответствующее приложение на их основе.

- Провести экспериментальное исследование реализованных алгоритмов и анализ результатов.

2. Обзор

В этой главе представлен обзор предметной области, в частности работ, связанных с повторами в документации ПО, описана модель повторов, которая будет использована в дальнейшем. Исходя из текущих исследований, сформулированы задачи поиска повторов в документации ПО. Также произведено исследование полулокальных задач поиска наибольшей общей подпоследовательности и выравнивания строк, описаны решающие их алгоритмы и идеи со свойствами, лежащими за ними.

2.1. Повторы в документации ПО

За последние десять лет появилось множество работ, посвященных документации ПО, в частности проблемам, связанным с наличием и выявлением в ней повторов. Одни работы посвящены эмпирическим исследованиям о количестве повторов в различных видах документации ПО [24, 5, 10], другие — фокусируются на реализации механизма переиспользования повторяющихся фрагментов информации в документации [7, 16, 23], третьи — на алгоритмах и подходах поиска повторов [9, 17, 4, 25, 26].

Эльмар Юргенсен и др. [5] провели широкомасштабное исследование спецификаций требований различных проектов на предмет наличия количество повторов, содержащихся в них. Их исследование показало, что количество повторов может быть значительно (вплоть до 70%). Стоит отметить, что они адаптировали $CONQAT$[2] — инструмент предназначенный для непрерывной оценки качества документации, для поиска клонов. В их неформальной модели, повтор – это часть (подстрока) спецификации, которая повторяется более двух раз, а группой считаются повторы, которые разделяют общую часть. Соответственно, их адаптированное решение находило лишь точные повторы.

Милан Носаль и др. [24] провели эмпирическое исследование пя-

[2] https://www.cqse.eu/, дата обращения 26.05.2020

ти крупных *JavaDoc* проектов, целью которого являлось проверка гипотезы о том, что комментарии к коду содержат большое количество повторов. Саму модель повторов они определили неформально, а инструмент для нахождения повторов[3] адаптировали, как и авторы [5]. Результаты их работы подтвердили гипотезу исследования.

Работа Мохамеда А. Умазиза и др. [10] также относится к эмпирическому исследованию *JavaDoc* проектов. Их работа мотивирована критической важностью документации программного кода в процессе разработки ПО. В работе повторы рассматриваются как цельные *JavaDoc* комментарии, которые встречаются в документации более одного раза, т.е рассматривается неформальная модель точных повторов. Для нахождения повторов был адаптирован инструмент *GumTree tool*[4], предназначенный для анализа программного кода на основе построения синтаксических деревьев. Результаты исследования показали, что разработчики часто переиспользуют части документации. Также их исследование показало, что текущие инструменты для работы с *JavaDoc* документацией не позволяют в полной мере избавиться от повторов.

В работе [4] авторами создан прототип инструмента *RepliComment* для нахождения точных повторов в *JavaDoc* документации, точнее комментариях к методам (использована неформальная модель повторов). Также инструмент позволяет классифицировать найденные решения, что позволило исследователям найти ряд ошибок и неточностей и, как следствие, улучшить качество документации (они отправили разработчикам список найденных ошибок и неточностей). В итоге, результаты анализа нескольких проектов с помощью *RepliComment* показали, что как код может иметь "плохой запах" (code smelss), так и комментарии к нему.

Авторы [25] так же, как и [5], делают акцент работы на повторах в спецификации требований, в частности на повторах функциональных требований в *use-case* диаграммах. Для анализа этого вида документации они реализовали комплексный инструмент *RegAligner*, который ис-

[3] CPD (copy-paste detector) инструмент на основе строкового алгоритма Рабин-Карпа, который позволяет находить только точные повторы.

[4] https://github.com/GumTreeDiff/gumtree, дата обращения 26.05.2020

пользует комбинированный подход для детектирования повторов. Подход основан на конвейерной архитектуре. На первом этапе применяются техники обработки естественного языка (nlp), затем используется метод машинного обучения для трансляции результатов в промежуточный язык. Таким образом, получается множество цепочек, которые попарно выравниваются. *ReqAligner* в силу своего подхода позволяет находить не только точные повторы, но и семантически схожие. Также необходимо отметить, что авторы не определили формальной модели повторов. Авторы провели апробацию инструмента на нескольких спецификациях, получив в результате 86% полноту и 63% точность. Их результаты апробации также подтвердили факт наличия повторов в *use-case* диаграммах.

Д. В. Луцив и коллеги внесли большой вклад своими исследованиями в области поиска повторов в документации ПО [9, 12, 17, 7, 19, 21]. Они разработали комплексный инструмент *Duplicate Finder*, позволяющий находить повторы в документации, визуализировать их, а так же вносить изменения в исходную документацию. Разработали формальную модель повторов в документации ПО и на ее основе создали алгоритмы для поиска повторов: алгоритм для поиска повторов по заданному шаблону и алгоритм для поиска групп повторов. Формальная модель позволила им доказать ряд свойств в отношении этих алгоритмов.

Алгоритм поиска групп повторов [21] основан на идее искусственного конструирования повторов. Сперва находятся все точные повторы, а затем на их основе строятся группы нечетких повторов. Такая интерпретация происхождения неточных повторов имеет право на жизнь, но она искусственна, что справедливо приводит к высокому уровню ошибок при нахождении повторов (*false-positive*) в результатах апробации в данной статьей.

Алгоритм поиска повторов по образцу [17] основан на идее так называемого скользящего окна. Сперва находятся все похожие фрагменты, а затем производится их фильтрация и сжатие. Выведенная авторами асимптотика алгоритма оценивается как $O(|p|^4)$, где p — шаблон, или же

как $O(|p|^2 \times m)$, где m — размер текста. Авторами доказана полнота алгоритма. Несмотря на это, наличие квадрата в асимптотической оценке делает его не применимым на потенциально больших данных. Также присутствует ограничение на используемую схему подсчета похожести — алгоритм обладает полнотой лишь при условии использования редакционного расстояния.

Таким образом, во-первых, для различных видов документации, в частности JavaDoc документации, задача поиска повторов является как никогда актуальной. Во-вторых, несмотря на большое количество работ, посвященных данной тематике, существующие подходы и алгоритмы могут быть улучшены как в теоретических оценках, так и на практике. В-третьих, в работах преобладает неформальная модель повторов. В-четвертых, исследователи помимо нахождения пар повторов, также ищут и группы повторов и осуществляют поиск повторов по образцу.

2.2. Модель повторов в документации ПО

Как было описано выше, существует множество неформальных моделей, по-разному определяющих понятие *повтора* в документации ПО. Можно сказать более точно, в каждой статье дается свое (авторское) видение. Формальная модель присутствует только в работах Д. В. Луцива.

В данной работе будет использована наиболее общая модель, согласно которой *повтор* — это отношение между двумя непересекающимися фрагментами a и b, которое задается с помощью функции похожести g. Отметим, что в данной модели не накладывается никаких ограничений на функцию похожести g. Например, в качестве g можно выбрать функцию, которая высчитывает количество одинаковых символов в двух фрагментах и сравнивает его с каким-то пороговым значением h. Заметим, что в этом случае функция симметрична по отношению к своим аргументам, т.е $g(a, b) = g(b, a)$.

Набор *повторов* может образовывать *группу повторов*, если он удовлетворяет определенным свойствам. Иными словами, *группа повто-*

ров — это множество повторов, которые в совокупности удовлетворяют заданному предикату. Предикатом может служить следующее утверждение: *В графе, построенном по группе повторов, существует хотя бы одна вершина, из которой достижимы все остальные. Вершинами графа являются повторы, ребра — значение функции g для выбранных вершин.*

2.3. Задачи поиска повторов в документации ПО

Можно выделить несколько задач относящихся к поиску повторов в документации ПО:

- Поиск всех *повторов*
- Поиск *групп повторов*
- Поиск *повторов* по образцу

Поиск всех повторов. Задача поиска всех пар *повторов* формулируется следующим образом: *Дан набор текстов t. В t необходимо найти все пары повторов, согласно выбранной функции похожести g.*

Поиск групп повторов. Задача поиска групп повторов формулируется следующим образом: *Дан набор текстов t. В t необходимо найти все непересекающиеся группы повторов согласно выбранной функции похожести g и предикату γ.* Эта задача является вариацией предыдущей задачи.

Поиск по образцу. Задача поиска по образцу формулируется следующим образом: *Дан шаблон p и набор текстов t. Необходимо найти все непересекающиеся повторы шаблона p в t согласно выбранной функции похожести g.* Стоит отметить, что могут существовать разные способы разбиения на непересекающиеся повторы.

2.4. Полулокальные задачи поиска

В этой секции будут даны общие сведения о полулокальных задачах наибольшей общей подпоследовательности и выравнивания строк. Детальные описания задач, теорем и их доказательств могут быть найдены в [29].

2.4.1. Задачи поиска наибольшей общей подпоследовательности (LCS) и выравнивания строк (SA)

Для начала, необходимо дать определение, что такое LCS и SA.

Задача о наибольшей общей подпоследовательности (LCS)

Даны две строки a и b длин m и n соответственно. Необходимо найти наибольшую общую подпоследовательность[5] символов для строк a и b, а именно значение длины этой подпоследовательности т.е lcs-значение. Пример:

$$lcs(abbba, aa) = aa \text{ (ее длина 2)}$$

Задача о выравнивании двух последовательностей (SA)

Даны две строки a и b длин m и n соответственно. Необходимо найти наибольшее значение функции выравнивания двух строк (sa) с учетом выбранной схемы оценки $w = (w_+, w_0, w_-)$ = (пара символов совпала, пара символов не совпала, символ выровнен по пропуску). Схема оценки отвечает за учет стоимости выравнивания пары символов из соответствующих строк, а сама функция выравнивания определяется следующим образом:

$$\begin{aligned} sa(a, b, w) = w_+k^+ + w_0k^0 + w_-(m + n - 2k^+ - 2k^0) = \\ k^+(w_+ - 2w_-) + k^0(w_0 - 2w_-) + w_-(m + n), \end{aligned} \tag{1}$$

где k^+ и k^0 — количество совпадающих и несовпадающих пар символов в выравнивании.

[5] Тоже, что и в математике. Подпоследовательность — это последовательность, которая может быть получена из другой последовательности путем удаления части (≥ 0) её элементов без изменения порядка следования элементов.

Иными словами, когда две строки выровнены, пара соответствующих символов γ из a и β из b будет называться выровненной. Символы в паре могут совпасть (w_+), а могут быть различны (w_0). Также при выравнивании символ одной строки может быть выровнен не по символу из другой строки, а по так называемому пропуску (gap, w_-), т.е при выравнивании происходит разрыв и ставится пропуск '-'. Пример выравнивания для строк $a = ABCA$, $b = ACBA$ и схемы оценки $w = (1, -0.3, -0.5)$:

$$\begin{array}{ccccc} \mathrm{A} & \text{-} & \mathrm{B} & \mathrm{C} & \mathrm{A} \\ \mathrm{A} & \mathrm{C} & \mathrm{B} & \text{-} & \mathrm{A} \end{array}$$

Здесь результат выравнивания будет равен $sa(a, b) = 3*1.0-2*0.5 = 2.0$.

Заметим, что задача поиска наибольшей подпоследовательности (LCS) является частным случаем задачи выравнивания строк (SA) при схеме подсчёта $(1, 0, 0)$.

Обе описанные задачи решаются динамическим программированием и имеют сложность $O(mn)$ (см. алгоритм 1).

Как было отмечено во введении, LCS и SA позволяют найти насколько в глобальном смысле похожи заданные строки. Во многих задачах интересны так называемые локальные и полулокальные случаи. Локальный случай относится к нахождению таких пар участков из a и b, которые максимально похожи друг на друга. Описание полулокальных задач дано ниже.

2.4.2. Semi-local LCS и SA

Задача поиска полулокальной наибольшей общей подпоследовательности (semi-local lcs)

Даны две строки a и b длин m и n соответственно. Необходимо найти значение lcs для следующих подзадач:

- **string-subtring**: для каждой из подстрок строки b найти наибольшую общую подпоследовательность со строкой a, а именно значение lcs;

Алгоритм 1 Префикс SA (LCS)

Вход: строки a и b длин m и n соответственно

Выход: наибольшее значение функции sa (lcs) для всех пар префиксов, в частности для $a[0:n]$ и $b[0:n]$

Комментарии: матрица h хранит соответствующие значения т.е элемент матрицы $h[i,j]$ хранит наибольшее значение sa (lcs) для префиксов строк $a[0:i]$ и $b[0:j]$

Псевдокод:

1: $h[0,j] := 0$ для всех $j \in 0..n$

2: $h[i,0] := 0$ для всех $i \in 0..m$

3: **for** i in 0...m **do**

4: **for** j in 0 ..n **do**

5: $$h[i,j] := max \begin{cases} h[i,j] + \begin{cases} w_+, & \text{если } a[i] = b[j] \\ w_0, & \text{если } a[i] \neq b[j] \end{cases}, \\ h[i-1,j] + w_-, \\ h[i,j-1] + w_-, \end{cases}$$

6: **end for**

7: **end for**

- **subtring-string** для каждой из подстрок строки a найти наибольшую общую подпоследовательность со строкой b, а именно значение lcs;

- **prefix-suffix** для каждой пары, состоящей из префикса строки a и суффикса строки b, найти наибольшую общую подпоследовательность;

- **suffix-prefix** для каждой пары, состоящей из префикса строки b и суффикса строки a, найти наибольшую общую подпоследовательность.

Решение данной задачи представляется в виде квадратной матрицы $H_{a,b}$, где каждый квадрант отвечает за одну из описанных подзадач:

$$H_{a,b} = \begin{bmatrix} suffix - prefix & substring - string \\ string - substring & prefix - suffix \end{bmatrix} \tag{2}$$

Более того, каждая ячейка содержит ответ для заданной пары под-

строк. Таким образом, сразу очевиден наивный алгоритм решения задачи — последовательное заполнении ячеек матрицы, что в худшем случае дает $O((n+m)^2) \times O((n+m)^2) = O((n+m)^4)$ сложность.

Оказывается, разработанная теория вокруг данной задачи позволяет ее решать намного быстрее. Точнее, существует несколько алгоритмов, имеющих асимптотику $O(n \times m)$ и $O(n \times m \times \log n)$ соответственно. Данные алгоритмы опираются на свойствах матрицы $H_{a,b}$, которыми она обладает.

Во-первых, по построению, матрица относится к классу так называемых *юнит анти-матриц Монжа* (*unit anti-Monge*). Матрица H называется (анти-)матрицей Монжа, если выполняется следующее:

$$H[i,j] + H[i^{'},j^{'}] \leq (\geq) H[i',j] + H[i,j'], \quad \forall\, i \leq i', j \leq j' \tag{3}$$

(Анти-)матрица Монжа H называется *юнит (анти)-матрицей Монжа*, если матрица (-)$H^{\square}$, получающаяся в результате взятия перекрестной разности анти-диагонали и диагонали для всех смежных квадратов 2 на 2, является перестановочной. Пример *юнит матрицы Монжа* (первая матрица):

$$\begin{bmatrix} 0 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix}^{\square} = \begin{bmatrix} (2+0)-(1+0) & (3+1)-(2+2) \\ (1+0)-(1+0) & (2+1)-(1+1) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4}$$

Во-вторых, данный класс матриц образует моноид с операцией $\otimes$ над этими матрицами , которая известна как *distance matrix multiplication*[6].

В-третьих, доказано [27], что можно произвести декомпозицию данных матриц и хранить их неявно в виде перестановочных матриц $P_{a,b}$ (так называемое ядро матрицы $H_{a,b}$), что позволяет существенно сократить объем используемой памяти. Эти ядра также образуют моноид, называемый *моноидом липких кос*, для которых также определена своя операция умножения $\odot$. Удивительным образом, эти два моноида

[6]Операция над матрицами в тропической алгебре. Это обычное матричное умножение, в котором умножение (*) заменено на сложение (+), а сложение (+) - на взятие минимума (min).

изоморфны друг другу [27], что позволяет использовать $\odot$ вместо $\otimes$. Иными словами, для того, чтобы посчитать произведение двух монжевых матриц, можно перейти к косам, произвести над ними операцию $\odot$, тем самым получив результат исходной задачи (справедливо и обратное для липких кос и операции $\otimes$).

В-четвертых, доказано [27], что ядро $P_{a,b}$ может быть выражено через конкатенацию двух меньших решений:

$$P_{a,b} = P_{a',b} \odot P_{a'',b}, a := a'a'' \tag{5}$$

Это позволяет использовать известный принцип *разделяй и властвуй*.

Стоит отметить, что А. В. Тискиным изобретен быстрый алгоритм для перемножения двух липких кос — *алгоритм муравья* [32], имеющий асимптотику $O(n \times \log n)$, где n — количество ненулевых элементов в перестановочной матрице. Однако неясно, насколько практическая реализация будет эффективна.

Из всего вышесказанного сразу вытекают следующие два рекурсивных алгоритма, основанные на умножении липких кос (алгоритм 2) и монжевого матричного умножения через $\otimes$ (алгоритм 3).

Также существует итеративный алгоритм, имеющий асимптотическую сложность $O(n \times m)$, основанный на распутывании кос[7].

Еще раз отметим, что для простоты изложения некоторые детали при описании опущены для упрощения. Детальное описание можно найти в [29], как было отмечено выше.

[7]С каждой перестановочной матрицей размера $n \times n$ ассоциирована *сокращенная липкая коса* (*reduced sticky braid*). Одной *сокращенной липкой косе* соответствует бесконечное множество липких кос. Таким образом, произвольную липкую косу можно *распутать* до (свести к) *сокращенной липкой косе*, которой уже будет отвечать ядро $P_{a,b}$

Алгоритм 2 Рекурсивный алгоритм для решения semi-local lcs через липкое умножение кос

Вход: строки a и b длин m и n соответственно
Выход: ядро матрицы $H_{a,b}$
Комментарии: Сложность алгоритма $O(n \times m)$
Псевдокод:

```
1:  if n = 1 и m = 1 then                          ▷ База рекурсии, длина строк 1
2:      if a = b then
3:          Вернуть ядро 2 × 2, отвечающее диагональной
              единичной матрице
4:      else
5:          Вернуть ядро размера 2 на 2, отвечающее антидиагональной
              единичной матрице
6:      end if
7:  else
8:      Выбрать строку с наибольшей длиной
9:      Разбить её на конкатенацию двух подстрок
10:     Рекурсивно запустить алгоритм над этими двумя половинами
11:     Произвести конкатенацию решений через ⊙
12:     Вернуть полученное решение
13: end if
```

Задача поиска полулокального выравнивания двух строк (semi-local sa)

Даны две строки a и b длин m и n соответственно и схема оценки $w = (w_+, w_0, w_-)$. Необходимо найти значение выравнивания (sa) для следующих подзадач:

- **string-subtring**: для каждой из подстрок строки b найти максимальное выравнивание со строкой a;
- **subtring-string**: для каждой из подстрок строки a найти максимальное выравнивание со строкой b;
- **prefix-suffix**: для каждой пары, состоящей из префикса строки a и суффикса строки b, найти максимальное выравнивание;
- **suffix-prefix**: для каждой пары, состоящей из префикса строки b и суффикса строки a, найти максимальное выравнивание.

Алгоритм 3 Рекурсивный алгоритм для решения semi-local lcs через явное умножение матриц

Вход: строки a и b длин m и n соответственно
Выход: матрица $H_{a,b}$
Комментарии: Сложность алгоритма $O(n \times m \times \log n)$
Псевдокод:

```
1:  if n = 1 и m = 1 then                    ▷ База рекурсии, длина строк 1
2:      if a = b then
3:          Вернуть матрицу, которой отвечает ядро размера 2 на 2,
             единичная диагональная матрица
4:      else
5:          Вернуть матрицу, которой отвечает ядро размера 2 на 2,
             единичная анти-диагональная матрица
6:      end if
7:  else
8:      Выбрать строку с наибольшей длиной
9:      Разбить её на конкатенацию двух подстрок
10:     Рекурсивно запустить алгоритм над этими двумя половинами
11:     Произвести конкатенацию найденный решений через ⊗
12:     Вернуть полученное решение
13: end if
```

Подход для решения этой задачи заключается в следующем. Заметим, что схему оценки в формуле (1) можно свести (нормализовать) к $w^{'} = (1, \frac{\mu}{v}, 0)$:

$$w = (w_{+}, w_{0}, w_{-}) = (w_{+} + 2x, w_{0} + 2x, w_{-} + x) = \\ (\frac{w_{+} + 2x}{w_{+} + 2x}, \frac{w_{0} + 2x}{w_{+} + 2x}, \frac{w_{-} + x}{w_{+} + 2x})_{x=-w_{-}} = (1, \frac{\mu}{v}, 0) \tag{6}$$

Тогда для того, чтобы получить значение sa, при известном $sa_{normalized}$ нужно применить обратную регуляризацию:

$$sa(a, b, w) = sa_{normalized}(w_{+} - 2w_{-}) + w_{-}(m + n) \tag{7}$$

Сведение схемы оценки к *регулярной* позволяет уменьшить количество параметров. Это, в свою очередь, позволяет решить задачу $semi-$

$local\ sa$ следующим образом[8]. При использовании техники $blown - up$[9] асимптотика решения через итеративный увеличится в v^2 раз до $O(m \times n \times v^2)$. Структура рекурсивного алгоритма через умножение кос позволяет добиться линейной зависимости от параметра v.

При большом значении параметра становится выгоднее использовать алгоритм, который не зависит от схемы оценки и не "раздувает" задачу. В данном случае идет речь об описанном ранее алгоритме решения через матричное умножение $\otimes$ (применяется без существенных изменений шагов алгоритма). В этом случае асимптотика решения не зависит от v и будет $O(n \times m \times \log n)$.

В конце обзора необходимо отметить, что матрица $H_{a,b}$ содержит большое количество информации, что позволяет решать такие задачи, как $ThresholdAMath$, $WindowAMatch$, $FragmentSubstring$, $Window - Subsring$, $Smith - Waterman\ alignment$ и др. Часть из них будет описаны в соответствующих разделах, относящихся к реализациям алгоритмов для поиска повторов.

Таким образом, реализация большей части алгоритмов, связанных с данной теорией представляет интерес как с точки зрения инженерной задачи, так и со стороны научного исследования.

- Алгоритмы обладают хорошим теоретическими свойствами, но большая часть из них не имеет практической реализации — неясно насколько они применимы на практике.

- Могут ли алгоритмы, решающие задачи *semi-local*, быть адаптированы и успешно применены к области поиска повторов в документации ПО?

[8]Детали доказательств и сведений [27].

[9]Исходные задача увеличивается в v^2 раз и сводится к задаче *semi-local lcs*.

3. Реализация библиотеки алгоритмов для полулокальных задач

Как было отмечено в обзоре полулокальных задач, существует необходимость в реализации большей части алгоритмов.

3.1. Архитектура библиотеки

В качестве языка программирования для написания библиотеки[10] был выбран язык *Kotlin*.

Код библиотеки можно разделить на два логических модуля:

- модуль *semilocalProblem* (рис. 1);
- модуль *semilocalApplication* (рис. 2).

3.1.1. Модуль semilocalProblem

В данном модуле реализована вся логика, отвечающая за задачи *semi-local*.

Интерфейс *IBraidMultiplication* отвечает за алгоритмы, реализующие операцию $\odot$, в частности, алгоритм *муравья*. В ходе работы алгоритма происходит последовательный доступ к перестановочным матрицам, к которым применен оператор $\nearrow$ и $\swarrow$[11]. Для достижения быстрого времени доступа к элементам матрицы ($O(1)$) используется класс $CountingQuery$, инкапсулирующий эту логику[12]. Хранение перестановочных матриц реализовано с помощью хранения двух перестановок, отвечающим строчкам и столбцам матрицы. Для реализации иной логики хранения необходимо реализовать методы абстрактного класса $AbstractPermuattionMatrix$

Классы, реализующие интерфейс *ISemiLocalCombined*, инкапсулируют всю логику, связанную с задачей $semi{-}local$. В частности, *ImplicitSe-*

[10]https://github.com/NikitaMishin/SemiLocalLcs, дата обращения 26.05.2020

[11]Оператор $\nearrow$ ($\swarrow$) подсчитывает сумму элементов, которые лежат левее (правее) и ниже (выше) выбранного узла в матрице.

[12]Теорема 1 в [32].

miLocalSA инкапсулирует логику по неявному хранению матрицы, отвечающей задаче *semi-local*, в виде *ядра* P и соответствующими алгоритмами над ним. На данный момент такими алгоритмами являются описанные в обзоре рекурсивный алгоритм с операцией $\odot$ и итеративный алгоритм распутывания кос. Стоит отметить, что для быстрого доступа к произвольному элементу матрицы используется интервальное двумерное дерево (*range tree*), построенное над ненулевыми элементами ядра. Это позволяет сократить объем требующейся памяти, но требует неконстантного времени доступа к элементам. В случае если доступ последовательный, используется *CountingQuery*, который позволяет добиться константной асимптотической сложности.

ExplicitSemiLocalSA хранит матрицу $H_{a,b}$ в явном виде. В данном случае нет экономии памяти, но доступ к произвольному элементу матрицы константный $O(1)$. Для решения задачи *semi-local* используется алгоритм *smawk* [15], отвечающий операции $\otimes$.

3.1.2. Модуль semilocalApplication

В данном модуле реализованы алгоритмы[13], для следующих задач:

1. *CompleteAMatch*
2. *Minimal-inclusive ThresholdAMatch*
3. *WindowAMatch*
4. *WindowSubstring*
5. *FragmentSubstring*
6. *BoundedLengthSmithWatermanAlignment*

Первая задача относится к нахождению значения максимального выравнивания заданного шаблона p и всех префиксов текста t из все-

[13] Детальное описание алгоритмов и их доказательств находятся в [27]

Рис. 1: Диаграмма классов UML части библиотеки, относящейся к различным задачам, основанным на *semi-local* задачах. Часть деталей и классов опущена.

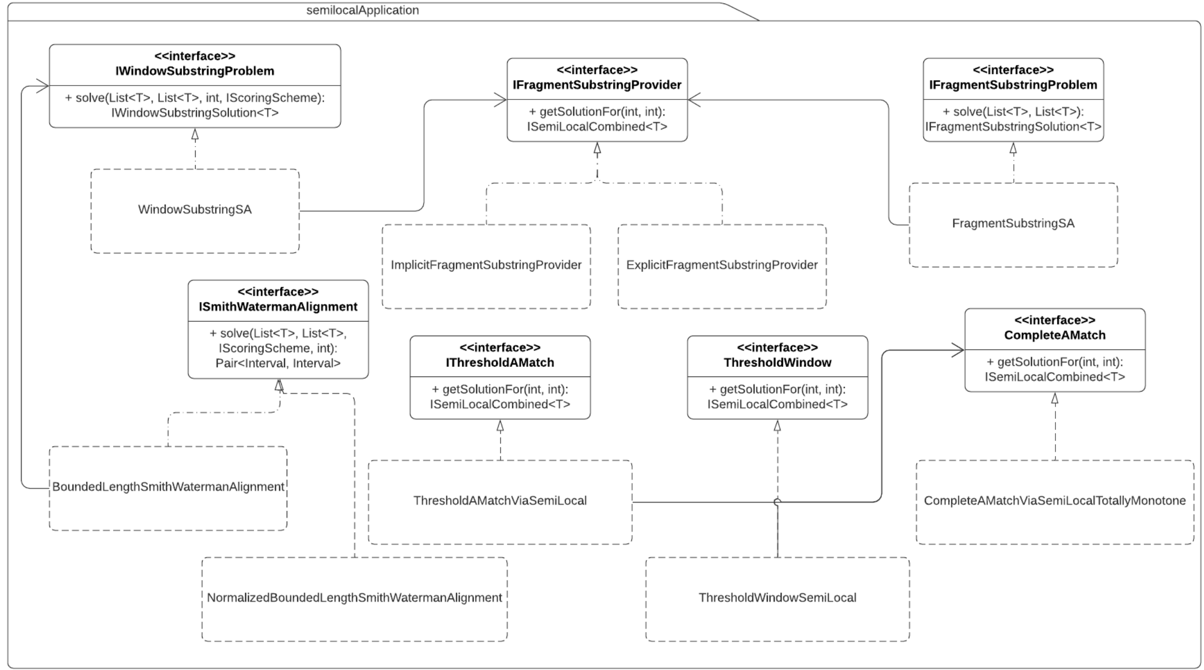


Рис. 2: Диаграмма классов UML части библиотеки, относящейся к *semi-local* задачам. Часть деталей и классов опущена

возможных суффиксов из данного префикса:

$$h[j] = \max_{i \in 0..j} sa(p, t[i, j]), j \in 0..|t| \tag{8}$$

В рамках второй задачи ставится задача нахождения всех непересекающихся повторов шаблона p в тексте t, чьи длины минимальны, а значение выравнивания выше заданного порога похожести h.

В третьей задаче необходимо найти все подстроки текста t (уже могут пересекаться) длины w, чье выравнивание с шаблоном p больше заданного порога похожести h.

Все эти три задачи сводятся к анализу *подматрицы* задачи *semi-local*, отвечающей *srting-substring*. И, соответственно, в зависимости от выбранного алгоритма решения задачи *semi-local* асимптотика алгоритмов решения этих задач $O(n \times m \times \log n)$ и $O(v \times m \times n)$.

Задача *FragmentSubstring* формулируется следующим образом: Для заданного множества интервалов (подстрок) r из текста t размера m и текста b размера n необходимо вычислить *semi-local* матрицу для каждого фрагмента из r против b. Имеет асимптотическую сложность

$O(v^2 \times r \times n \times \log m \times \log mv)$ [14] и $O(r \times n \times m \times logm)$.

Задача *WindowSubstring* является частным случаем *FragmentSubstring*, в которой размер фрагментов фиксирован. Асимптотическая сложность решения уже будет $O(n \times m \times \log n)$ и $O(v^2 \times m \times n)$.

Обе задачи основаны на двоичном разложении числа и предподсчете *semi-local* решений, отвечающих данным разложениям.

Задача *SmithWatermanAlignment* относится к локальному выравниванию. В рамках данной задачи необходимо вычислить значение максимального локального выравнивания между a и b, т.е найти пару подстрок, на которых достигается максимальное выравнивание. Очень часто данная задача представляет интерес с различными ограничениями [3]. Например, ограничения на минимальную длину подстрок. Данное ограничение реализовано с помощью алгоритма из [33] и инкапсулировано в соответствующем классе *BoundedLengthSmithWaterman-Alignment*. Как отметил автор статьи, можно реализовать нормализованную версию данной задачи с применением *BoundedLengthSmithWaterman Alignment* к алгоритму из [3].

Нетрудно заметить, что часть алгоритмов в той или иной степени может быть адаптирована к задачам поиска повторов в документации. Для задачи поиска по образцу такими кандидатами являются алгоритмы для решения задачи *Minimal-inclusive ThresholdAMatch* и *WindowSubstring*. Для задачи поиска групп повторов могут быть применены алгоритмы для *WindowAMatch*, *Minimal-inclusive ThresholdAMatch*, *BoundedLengthSmithWatermanAlignment* и *semi-local*.

Адаптация алгоритмов к поиску повторов описана в следующем разделе. Экспериментальная проверка асимптотики части алгоритмов, а так же их потенциальная возможность применения к большим данным даны в главе 5.

[14] Существует возможность улучшить асиптотику до $O(v \times r \times n \times \log^2 m)$

4. Приложение для поиска повторов в документации ПО

В данной главе описано приложение для поиска повторов в документации ПО, в рамках которого будет производится экспериментальное исследование применимости алгоритмов, решающих полулокальные задачи. Также описаны основные технические решения и архитектура. Описан подходы для поиска повтор. Для каждой из *задач поиска повторов* описано решение, основанное на использовании *библиотеки алгоритмов* для полулокальных задач (см. главу 3).

4.1. Общая архитектура приложения

На рисунке 3 представлена архитектура приложения. Оно реализовано в виде двух крупных компонент.

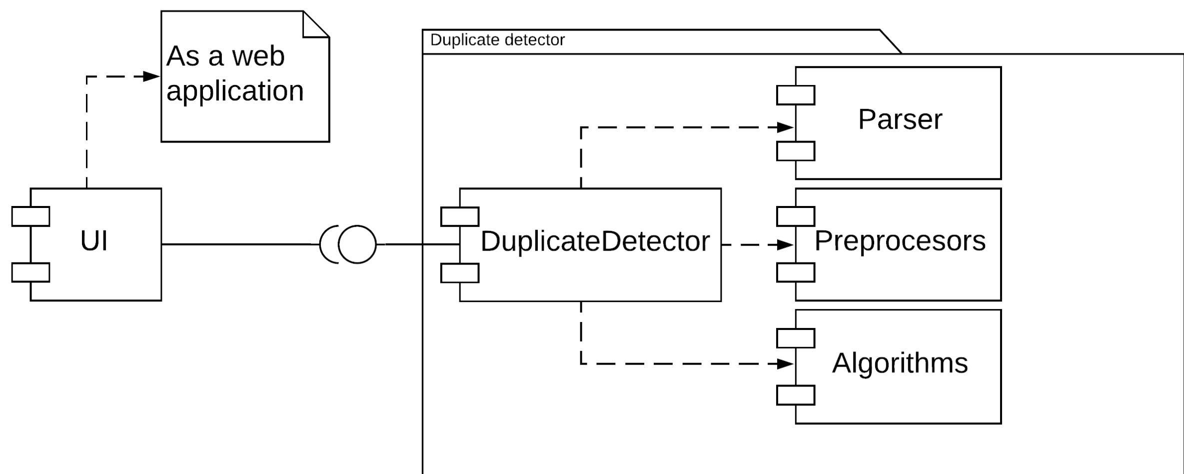


Рис. 3: Диаграмма компонентов системы

Первая компонента (клиентская часть) — это пользовательский интерфейс (*UI*), который отвечает за визуализацию и взаимодействие с пользователем. Пользователь настраивает параметры поиска повторов, тип поиска, указывает файлы, в которых необходимо произвести анализ на дубликаты, или путь к проекту на *Github* в интернете (рис. 4).

Клиентская часть написана на *python* и *java script*. Детальное описание клиентской части дано в секции 4.2.

Вторая компонента (серверная часть) отвечает за поиск повторов согласно заданным настройкам. Данная часть написана на языке *Kotlin*. В секции 4.3 детально описана функциональность данной части системы.

Взаимодействие между компонентами осуществляется посредством *JSON*-формата. Приложение реализовано в виде вэб-приложения, которое запускается в докер-контейнере, что минимизирует пользовательские требования для запуска программы[15].

Стоит отметить, что в этой работе сделан основной акцент на поиск повторов в *JavaDoc* документации в силу её актуальности для данного формата (см. главу 2.1).

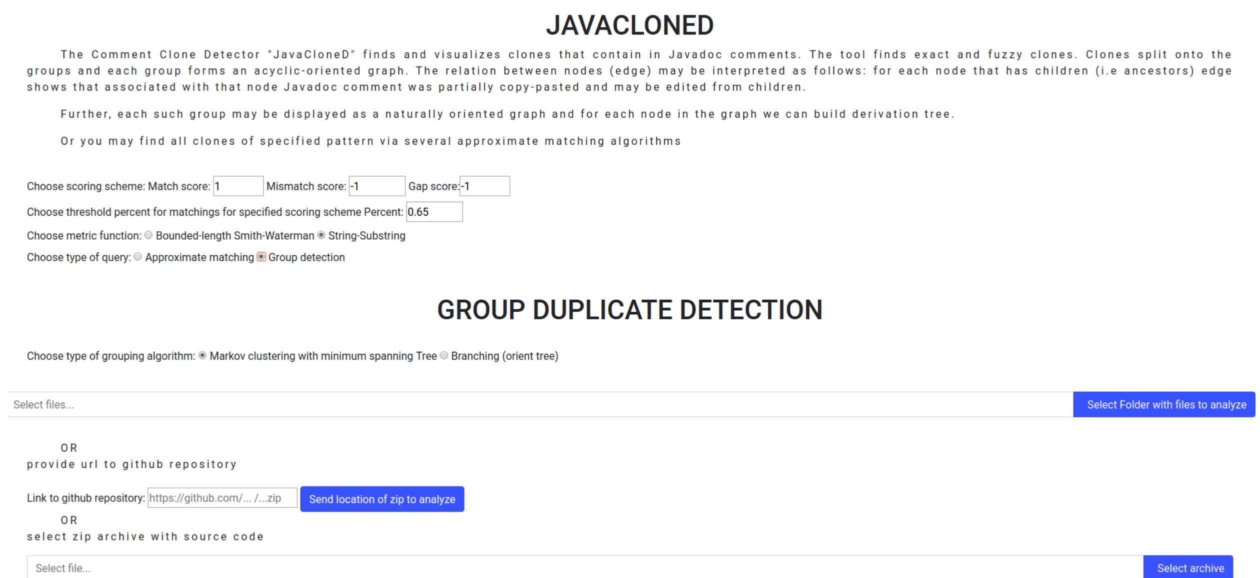


Рис. 4: Интерфейс пользователя перед запуском анализатора для случая поиска групп повторов

4.2. Клиентская часть

Как было отмечено выше, клиентская часть реализована в виде веб-приложения, которое написано посредством *python*-фреймворка *flask*[16].

[15]Достаточно иметь *Docker* и *Браузер*.

[16]https://flask.palletsprojects.com/en/1.1.x/, микро-фреймворк для написания веб-приложений, дата обращения 26.05.2020

На основной странице вэб-приложения пользователь выставляет тип решаемой задачи, параметры запуска, указывает исходники и запускает вычисления. Пользователь имеет возможность выбрать задачу *"Поиск повтора по шаблону"* или *"Поиск всех групп повторов"*.

Для визуализации найденных повторов в случае с *"Поиском повтора по шаблону"* результаты отображаются на странице ответа с цветовой расцветкой (см. рис. 5).

```
Pattern:
Started short assured hearing spec expense

    We diminution preference thoroughly if. Joy deal pain view much her time. Led y
Subjects to ecstatic children he. Could ye leave up as built match. Dejection agree
Use securing confined his shutters. Delightful as he it acceptance an solicitude d:
Extremely we promotion remainder eagerness enjoyment an. Ham her demands removal br
No opinions answered oh felicity is resolved hastened. Produced it friendly my if c
Prepared is me marianne pleasure likewise debating. Wonder an unable except better
Extremity sweetness difficult behaviour he of. On disposal of as landlord horrible.
Sportsman do offending supported extremity breakfast by listening. Decisively advar
Affronting everything discretion men now own did. Still round match we to. Frankne:
Is allowance instantly strangers applauded discourse so. Separate entrance welcome(
tarted now shortly had for assured hearing expense.

Started now shortly had for assured hearing expense. halo
Started short for assured hearing voices expense villaian

Started now shortly.
```

Рис. 5: Визуализация найденных повторов для поиска по шаблону

Для групп повторов механизм визуализации более комплексный (см. рис 6): используется три окна для интерпретации результатов.

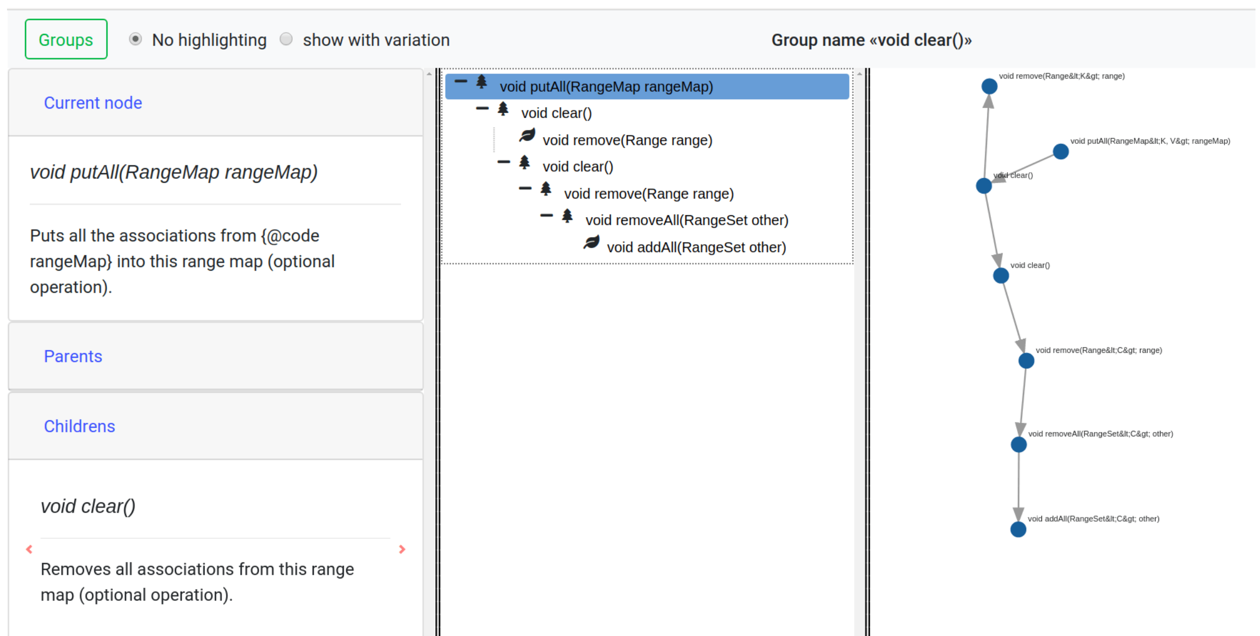


Рис. 6: Визуализация групп повторов

Первое окно отвечает за отображение отношений между фрагментами, которые состоят в отношении (имеют похожие части). Интерпретация следующая: для текущего узла "родителем" являются все те фрагменты, в которые была взята часть информации с текущего фрагмента и, возможно, видоизменена. "Детьми" являются те фрагменты, с которых, вероятнее всего, произошло дублирование информации.

Второе окно отвечает за представление каждого повтора в виде иерархической структуры. Представление реализовано в виде *tree control* — для каждого повтора (вершины) строится ориентированное дерево вывода из этой вершины. Такая интерпретация показывает, откуда вероятнее всего "произошел повтор", т.е откуда произошло дублирование данных.

Третье окно отвечает за визуализацию соответствующей группы повторов в виде ориентированного графа. Это позволяет увидеть структуру найденной группы. Все три окна синхронизированы между собой.

Описание используемых алгоритмов для нахождения повторов даны в секциях 4.4 и 4.5 соответственно.

4.3. Серверная часть

Как было указано выше, эта часть приложения отвечает за поиск повторов в документации.

На рис. 3 выделены компоненты серверной части (*Kotlin application*). Общий подход (*pipeline*) поиска повторов заключается в следующем.

Сперва происходит синтаксический анализ с целью нахождения тех фрагментов, которые будут анализироваться согласно выбранным параметрам, заданными пользователем. В данной работе анализируются *JavaDoc* документация, а именно *JavaDoc*-комментарии методов, классов и интерфейсов. Это осуществлено с использованием библиотеки *JavaParser*[17]. Для анализа иного вида документации (например, обычный текст, как в случае с поиском по образцу) достаточно в модуле *Parser* реализовать необходимые интерфейсы. Далее, комментарии обрабатываются различными фильтрами — происходит токенизация, лемматизация, убираются стоп-слова и пр.

Данная обработка происходит с частичным использованием функциональности стэнфордского фреймворка *core nlp*[18] для обработки естественного языка.

После этапа предобработки фрагментов происходит конвертация слов в промежуточное представление в виде чисел с целью экономии памяти и ускорения работы алгоритмов.

Затем происходит запуск соответствующих алгоритмов поиска, о которых пойдет речь дальше.

4.4. Алгоритмы для решения задачи поиска повторов

В данной секции будут описаны алгоритмы для решения задачи поиска по образцу, согласующиеся с определенной моделью в главе 2.2. Алгоритмы основаны на использовании *библиотеки алгоритмов* (см. главу 3).

[17]https://javaparser.org/, дата обращения 26.05.2020

[18]https://stanfordnlp.github.io/CoreNLP/, дата обращения 26.05.2020

4.4.1. Улучшенный алгоритм интерактивного поиска

В данной секции описана улучшенная версия алгоритма из [17]. Псевдокод алгоритма представлен ниже (алгоритм 4).

Алгоритм 4 Нечеткий поиск по шаблону с использованием semi-local

Вход: шаблона поиска p, текст t, пороговое значение похожести k

Выход: множество непересекающихся повторов шаблона p

Комментарии:

$$k_{di} = |p| * (\frac{1}{k} + 1)(1 - k^2) \quad (9)$$

Псевдокод:

1: $W = semilocalSa(p, t)$
2: $W_2 = \emptyset$
3: **for** $w \in W$ **do**
4: **if** $sa(p, w) \geq -k_{di}$ **then**
5: *continue*
6: **end if**
7: $maximums = FindMaxForColumnsBySmawk(w)$
8: $max = FindMaxWithLenghtConstraint(maximums)$
9: **if** $max \geq -k_{di}$ **then** ▷ В исходном был минимум, поэтому минус
10: add substring associated with max to W_2
11: **end if**
12: **end for**
13: $W_3 = UNIQUE(W_2)$ ▷ 3 фаза без изменений
14: **for** $w \in W_3$ **do**
15: **if** $\exists w^{'} \in W_3 : w \subset w^{'}$ **then**
16: $remove\ w\ from\ W_3$
17: **end if**
18: **end for**
19: $return\ W_3$

В строке 1 вычисляется решение задачи *semi-local sa*, в частности подзадачи *string-substring*. В строках $3 - 12$ внутри каждого окна размера $|w_s| \approx |p|$ сперва проверяется, что значение выравнивания шаблона p

и окна w больше заданного порога похожести k_{di}, а затем внутри окна находится такая подстрока, что её выравнивания максимально среди всех друг подстрок данного окна. При одинаковом значении выравнивания будет выбираться наиболее длинная строка. Строки $13-18$ отвечают фильтрации, в рамках которой происходит удаление одинаковых и пересекающихся повторов, в результате чего остаются только непересекающиеся повторы.

Корректность улучшения

Нетрудно заметить, что данная версия имеет лучшую асимптотическую сложность, чем исходный алгоритм. Более того, все свойства алгоритма сохраняются.

Исходный алгоритм разбит на 3 фазы: 'сканирование', 'усушка' и 'фильтрация'.

На первой фазе исходный текст t анализируется скользящим окном размером $w_s = \frac{|p|}{k}$, где $k \in [\frac{1}{\sqrt{3}}, 1]$ параметр алгоритма, с шагом в один символ, а именно вычисляется редакционного расстояние[19] между каждым окном и заданным шаблоном p. Асимптотическая сложность данного шага $O(|p|^2 \times |t|)$ согласно [17].

Заметим, что редакционное расстояние может быть выражено через выравнивание последовательностей. Конкретнее, редакционное расстояние для данного случая выражается через следующую схему оценки:

$$(w_+, w_0, w_-) = (0, -2, -1) \tag{10}$$

Соответственно, редакционное расстояние можно заменить на выравнивание последовательностей с весами $(0, -2, -1)$ и искать максимальное выравнивание без потери свойств алгоритма в силу эквивалентности двух задач. Из этого следует, что можно применить алгоритмы для решения *semi-local sa*. В силу формулы (6), значение нормализованной

[19] Метрика, минимальное количество операций вставки, удаления и замены одного символа на другой для превращения одной строки в другую.

схемы будет следующее:

$$(0, -2, -1) \to (1, \frac{\mu = 0}{v = 1}, 0) \tag{11}$$

Таким образом, используя нормализацию, задачу можно свести к *semi-local lcs*. И, следовательно, асимптотическая сложность первой фазы алгоритма из [17] может быть улучшена. Тогда её асимптотическая сложность будет $O(|t| \times |p|)$ вместо $O(|t| \times |p|^2)$. Данная стадия эмулируется в строке 1.

Во второй фазе происходит так называемая 'усушка' — для каждого окна происходит вычисление минимальной подстроки, на которой достигается минимальное значение редакционного расстояния (в случае выравнивания последовательностей это относится к максимальному значению). При равенстве расстояний выбирается подстрока наибольшая по длине. Асимптотическая сложность данной фазы выражается как $O(|p|^4)$.

Для улучшения данной фазы применяется следующее. Матрица решений $H_{p,t}$ *semi-local* содержит подматрицу *string-substring*, которая содержит значения выравниваний шаблона p со всеми подстроками текста t. Как было отмечено выше, эта матрица является анти-матрицей Монжа, следовательно, в ней можно быстро искать максимум в столбцах (строках) через алгоритм *smawk*, имеющий асимптотическую сложность $O(\textit{размер матрицы} \times \textit{время доступа к элементу матрицы} = \gamma$[20]$)$. Заметим, что этот алгоритм устойчив, в том плане, что он выдает первую позицию, на которой достигается максимум. Например, если для текущего столбца j максимум достигает в позициях i и $i^{'}$, $i < i^{'} <= j$, то в результате работы *smawk*, для столбца j алгоритм выдаст i, что соответствует тому, что при равенстве значений будет выбираться наиболее длинная подстрока. Имея максимумы для каждого столбца (каждого суффикса префикса), можно найти все максимумы и среди них выбрать подстроку наибольшую по длине.

Для нахождения максимума в столбце воспользуемся следующими

[20]Далее будет обозначать асимптотическую сложность доступа к произвольному элементу матрицы через символ γ

соображениями.

- Если $H_{p,t}$ является *анти-матрицей Монжа*, то $-H_{p,t}$ является *матицей Монжа.*

- В результате транспонирования, матрица не перестает быть *(анти)-матрицей Монжа.*

Иными словами, нахождение минимума в строке в $-H_{p,t}^T$ будет соответствовать нахождению максимума в столбце в матрице $H_{p,t}$. В улучшенном алгоритме это отвечает строкам $7 - 10$.

Таким образом, асимптотическая сложность для каждого окна будет соответственно равна $O(|w_s| \times \gamma)$. В худшем случае, таких окон будет $O(|t|)$. Следовательно, вторая фаза алгоритма будет иметь асимптотическую сложность $O(|w_s| \times |t| \times \gamma)$. Учитывая то, что $k \in [\frac{1}{\sqrt{3}}, 1]$, то $|w_s| \approx |p|$ и $O(|w_s| \times |t| \times \gamma) = O(|p| \times |t| \times \gamma)$. Значит, асимптотическая сложность второй фазы $O(|p| \times |t| \times \gamma)$, все свойства алгоритма сохранены.

Третья фаза, отвечающая за фильтрацию фрагментов, остается без изменений. Ее асимптотическая сложность $O(|p| \times \log |p|)$ согласно [17].

Соответственно, алгоритм сохранит все свои свойства и будет уже иметь асимптотику $O(|p| \times |t|)$[21]. Заметим, что алгоритм имеет оптимальную асимптотику при явном хранении матрицы $H_{p,t}$ (время доступа к произвольному элементу константное) согласно [1].

4.4.2. Алгоритм нечеткого поиска шаблона с использованием ThresholdAMatch

Более простое решение относится к алгоритму 5, реализация которого уже содержится в *библиотеке алгоритмов*. Он позволяет найти все непересекающиеся повторы шаблона p в тексте t.

Сперва решается задача *CompleteAMatch*, в рамках которой для каждого столбца j подматрицы *string-substring* находится максималь-

[21] В то время как исходный алгоритм оценивается как $\max(O(|p|^2 \times |t|, O(|p|^4)$, что при $|p| \approx |t|$ дает 4 степень.

Алгоритм 5 Нечеткий поиск по шаблону с использованием Min-inclusive ThresholdAMatch

Вход: шаблона поиска p, текст t, пороговое значение похожести h
Выход: множество непересекающихся повторов шаблона p
Комментарии: в реализации h высчитывается исходя из схемы оценки и процента похожести, выраженного через число из отрезка $[0, 1]$
Замечание: При использовании интервального дерева данный алгоритм можно адаптировать таким образом, что на каждом шаге будет выбираться максимальный интервал из текста. Тогда асимптотика возрастет в $O(log|t|)$ раз
Псевдокод:

1: $maxSuffixes = CompleteAMatch(p, t)$
2: $reverse(maxSuffixes)$
3: $result = \emptyset$
4: **for** $(i, j, score) \in maxSuffixes$ **do**
5:     **if** $score \geq h \& j \leq res.last().i$ **then**
6:         $res.add((i, j, score))$
7:     **end if**
8: **end for**
9: $return\ result$

ное значение и позиция (i, j'), в которой оно достигается, т.е находится суффикс префикса, который больше всего похож на шаблон. Далее, над полученным результатом производится фильтрация, начиная с конца. В результате чего остаются только непересекающиеся повторы минимальной длины, которые больше заданного порога похожести. Асимптотическая сложность данного решения зависит от выбранного алгоритма решения *semi-local*. Соответственно, $O(|t| \times |p| \times \log |t|)$, $O(|t| \times |p| \times v^2)$ или $O(|t| \times |p| \times v)$.

Данный алгоритм рассматривается как альтернатива алгоритму 4.

4.4.3. Алгоритм нечеткого поиска шаблона с использованием Разреза

Еще одним решением на основе *semi-local* является следующий алгоритм. Во-первых, задачу поиска по шаблону можно сформулировать с иной точки зрения: необходимо найти все максимальные по выравни-

ванию непересекающиеся повторы шаблона p в тексте t, т.е получить такую цепочку непересекающихся интервалов $(i_1, j_1), ..., (i_n, j_n)$, что на (i_k, j_k) достигается максимальная похожесть на еще непокрытой найденными интервалами части текста t. В рамках задач *semi-local* это означает разбитие матрицы *string-substring* на непересекающиеся подматрицы, с учетом максимумов в подматрицах. Последнее относится к быстрому поиску максимума в матрице (*range maximum query*). В силу того, что матрица *string-substring* является матрицей Монжа, можно применить результат из статьи [14]. Для этого необходимо реализовать структуру данных, асимптотическая сложность построения которой равна $O(|t| \times \log |t|)$ (размер структуры выражается как $O(|t|)$), которая позволяет делать запросы на поиск максимума в произвольной подматрице, имеющие асимптотическую сложность $O(\log \log |t|)$. Учитывая, что непересекающихся повторов в тексте t может быть в худшем случае $|t|$ штук, для их нахождения необходимо осуществить $|t|$ запросов на поиск максимуму. Следовательно, конечная асимптотика алгоритма $O(|t| \times \log \log t) + O(\textit{сложность подсчета semi-local}) = O(\textit{сложность подсчета semi-local})$, так как $\log \log |t| \leq |p|$ для достаточно больших значений $|t|$. Псевдокод алгоритма представлен на листинге 6.

4.5. Алгоритмы для решения задачи поиска групп повторов

В данной главе описаны алгоритмы решения задачи поиска групп повторов на основе использования *библиотеки алгоритмов* и применения графовых алгоритмов.

Согласно определенной в секции 2.2 модели, для решения задачи *поиска групп повторов* необходимо задать функцию похожести g и выбрать предикат γ. Заметим, что для рассматриваемого случая, текстовыми фрагментами, в которых ищутся повторы, являются цельные *JavaDoc*-комментарии. Соответственно, в данной работе в отношении поиска групп повторов *повторами* будут служить семантически за-

Алгоритм 6 Нечеткий поиск по шаблону с использованием maxRangeQuery

Вход: шаблон поиска p, текст t, пороговое значение похожести h
Выход: множество непересекающихся повторов шаблона p
Комментарии: в реализации h высчитывается исходя из схемы оценки и процента похожести, выраженного через число из отрезка $[0, 1]$
Псевдокод:

1: $S = SolveSemiLocalSA(p, t)$
2: $W = BuildStructForRangeQuery(S)$
3: $IntervalsToSearch = \emptyset$
4: $IntervalsToSearch.add((0, |t|))$
5: $result = \emptyset$
6: **while** $IntervalsToSearch.isNotEmpty()$ **do**
7: $\quad (i, j) = IntervalsToSearch.pop()$
8: $\quad score, i^{'}, j^{'} = W.query(i, j)$
9: $\quad$ **if** $score \geq h$ **then**
10: $\quad\quad result.add((i^{'}, j^{'}, score))$
11: $\quad\quad IntervalsToSearch.add(i, i^{'})$
12: $\quad\quad IntervalsToSearch.add(j^{'}, j)$
13: $\quad$ **end if**
14: **end while**
15: $return\ result$

мкнутые куски текста, как и в [26][22], т.е *JavaDoc* комментарии.

Соответственно, весь набор комментариев образует граф. Он может быть как ориентированным, так и неориентированным. Это зависит от того, является ли g симметричной по отношению к своим аргументам. Таким образом, в полученном графе можно выделить группы повторов согласно предикату γ. На 7 представлен псевдокод алгоритма. Отметим, что асимптотика алгоритма выражается, как $\max(O(|t|^2 * g), O(s))$.

Функция g может быть определена через локальное, полулокальное и глобальное выравнивание, соответственно. В данной работе в качестве g выбраны следующие алгоритмы из *библиотеки алгоритмов*.

- *BoundedLengthSmithWaterman* — локальное выравнивание.
- *Semi-local sa* — полулокальное выравнивание.

[22] В [26] это были топики текста в *Dita* документации.

Алгоритм 7 Алгоритм поиска групп повторов для JavaDoc-комментариев

Вход: набор комментариев t_i, функция g, которая меряет похожесть между двумя комментариями, функция s, которая согласно выбранному предикату γ строит группы, пороговое значение похожести h
Выход: группы непересекающихся повторов
Комментарии: в реализации h высчитывается исходя из схемы оценки и процента похожести, выраженного через число из отрезка $[0, 1]$
Псевдокод:

```
1:  graph = Graph(vertices = t)
2:  for t_i ∈ t do
3:      for t_j ∈ t, t_i ≠ t_j do
4:          if g(t_i, t_j) ≥ h then
5:              addEdge(t_i, t_j, g(t_i, t_j))
6:          end if
7:          if g(t_j, t_i) ≥ h then
8:              addEdge(t_j, t_i, g(t_j, t_i))
9:          end if
10:     end for
11: end for
12: groups = s(graph)
13: return groups
```

Следующие эвристические соображения помогают определить функции s, которые подходят для нахождения групп.

Во-первых, в силу того, что мы рассматриваем граф, естественным образом задача сводится к кластеризации графа/выделению компонент (сильной) связности.

Во-вторых, ориентированное ребро $a \xrightarrow{g(a,b)} b$ в графе можно естественным образом интерпретировать так: часть текста из b была скопирована в фрагмент a или текст b был скопирован, и в новом фрагменте произведена модификация этой копии и получено a. При существовании обратного ребра будем считать, что при условии $g(a, b) \geq g(b, a)$, a является потомком b (помним, что в общем случае $g(a, b) \neq g(b, a)$ и наоборот.

В-третьих, очень часто бывает, что повторы практически не отличаются друг от друга или же в точности совпадают друг с другом. Такие

повторы хочется различать от обычных. Если рассматривать граф, такое состояние для части вершины выражается через термин *клика*[23] и относится к задаче поиска клики. Соответственно, новый граф, в котором присутствуют клики строится, из исходного обновлением весов тех ребер, которые входят в клики или находятся внутри клик.

В-четвертых, ассоциированный с группой ориентированный граф должен быть деревом. Эта эвристика основана на том, что вершина не может быть потомком сама себе (наличие циклов) и не может иметь двух одинаковых потомков (проблема множественного наследования). Соответственно, граф является деревом.

Исходя из описанных выше эвристик, были разработаны алгоритмы 8, 9. Также применен алгоритм из статьи [34].

В 8 используется идея иерархической кластеризации с тем изменением, что добавляется новый вид вершины, который олицетворяет клики. Как известно, поиск клики — это NP-полная задача. Поэтому в данном алгоритме произведена аппроксимация поиска клик: листовая вершина принадлежит кластерному узлу, если её расстояние до клики больше заданного порога похожести для клик. В ходе алгоритма в цикле *while* происходит нахождение двух ближайших вершин согласно выбранной метрике, их объединение согласно правилам и пересчет матрицы расстояний, которая отвечает уже новому графу. В общем случае[24] пересчет матрицы требует $O(n^2)$ времени, где n — количество вершин. В худшем случае цикл *while* будет исполняться $O(n)$ раз, тогда асимптотика алгоритма составит $O(n^3)$.

В алгоритме 9 использована идея кластеризации на основе марковских моделей [11] и дальнейшего построения минимальных остовных деревьев внутри каждого кластера. Асимптотическая сложность первого шага реализации в данной работе $O(n^3)$ в худшем случае. Нахождение минимального остовного дерева реализовано с помощью алгоритма Крускала с использованием системы непересекающихся множеств. Сложность второго шаге $O(n^2 \times \log n)$. Соответственно, общая слож-

[23]Полный граф на заданных вершинах.

[24]Некоторые метрики позволяют считать асимптотически быстрее.

Алгоритм 8 Алгоритм выделения групп на основе Иерархической кластеризации

Вход: граф G с матрицей расстояний, функция f, которая считает дистанцию между вершинами, пороговое значение похожести h_{clique}, при котором вершины образуют очередной уровень в иерархии, h_{group} — пороговое значение похожести

Выход: иерархические группы повторов

Псевдокод:

1: $roots = G.vertices()$
2: **while** $roots.isNotEmpty()$ **do**
3: $\quad (from, to, score) = closestVertices(root)$
4: $\quad newVertex = switch\{$
5: $\quad score \geq h_{clique}, from, to \in Leaf \rightarrow Clique(from, to)$
6: $\quad score \geq h_{clique}, to \in Clique, from \in Leaf \rightarrow to.add(from); to$
7: $\quad score \geq h_{clique}, from, to \in Clique \rightarrow from.addAll(to); from$
8: $\quad score \geq h_{group}, \rightarrow ClusterNode(from, to)$
9: $\quad else \rightarrow break$
10: $\quad \}$
11: $\quad G.recalcualateDistance()$
12: $\quad roots.remove(from)$
13: $\quad roots.remove(to)$
14: $\quad roots.add(newVertex)$
15: **end while**
16: $return\ roots$

ность $O(n^3)$.

Алгоритм [34] решает задачу построения такого ориентированного подграфа G_{branch} из исходного G, что:

- В нем нет циклов
- Ни в какую вершину не входит больше одного ребра

Причем среди всех таких подграфов он оптимален:

$$\sum_{w \in G_{branch}} w \geq \sum_{w' \in G_{branch'}} w' \qquad (12)$$

Это соотносится с последней эвристикой о том, что граф должен быть деревом. Алгоритм из [34] обладает асимптотической сложностью $O(n^2)$.

Алгоритм 9 Алгоритм выделения групп на основе Марковских моделей

Вход: граф G с матрицей расстояний
Выход: группы повторов с структурой группы в виде дерева
Псевдокод:

1: $trees = \emptyset$
2: $clusters = mclClustering(G)$
3: **for** $cluster \in clusters$ **do**
4: $\quad tree = BuildMaximumSpanningTree()$
5: $\quad trees.add(tree)$
6: **end for**
7:
8: $return\ trees$

Результаты применимости описанных алгоритмов из данной главы к поиску повторов в документации ПО представлены в разделе 5.

5. Апробация и анализ результатов

В данной главе представлены результаты апробации алгоритмов из *библиотеки алгоритмов полулокальных задач*. Также дана оценка применимости решений полулокальных задач *semi-local lcs и sa* к поиску групп повторов в *JavaDoc*-комментариях и задаче поиска повторов по шаблону.

5.1. Тестовый стенд

Для проведения экспериментов использовалась машина с процессором *Intel-Core i5* и оперативной памятью размером *16GB*. Операционная система *Ubuntu 18.04 Bionic*. На каждый запуск *jar*-файла выделялось 10GB памяти.

5.2. Экспериментальная проверка асимптотики

В данной секции описаны экспериментальные результаты запусков части алгоритмов из реализованной *библиотеки алгоритмов* (см. главу 3) и дана их интерпретация.

На рис. 7 представлен результат запусков обычного *prefix lcs*, *prefix lcs* с хранением результатов с помощью одной строки и полулокального *lcs* (*semi-local lcs reducing*). Исходя из результатов, можно сделать вы-

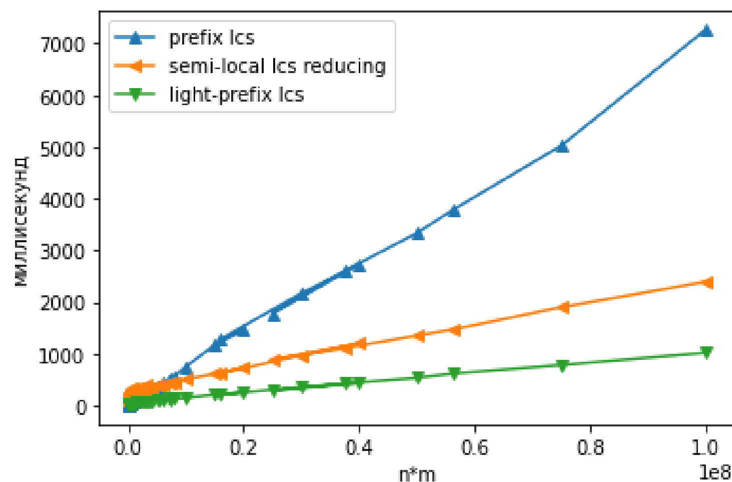


Рис. 7: Результат запусков различных версий подсчета *lcs*

вод, что скорость вычисления *semi-local lcs* сопоставима с вычислением обычного *lcs*.

На рис. 8 представлено сравнение двух реализаций подсчета *semi-local lcs*: через распутывание кос (*reducing*) и умножение кос (*recursive*).

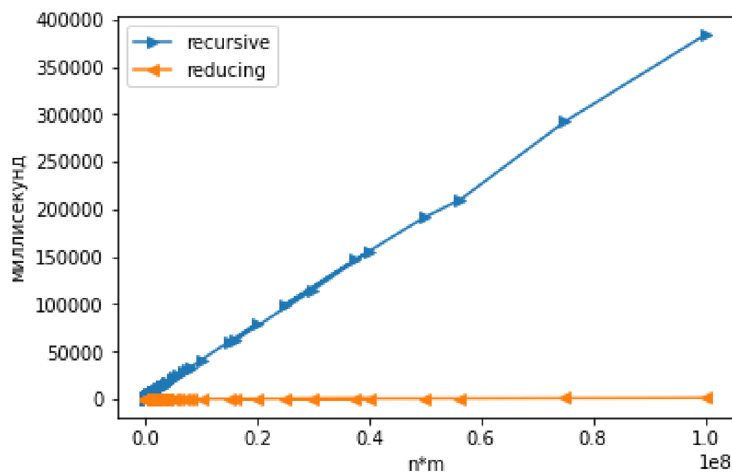


Рис. 8: Результат запусков алгоритма на основе распутывания кос и рекурсивного умножения кос для решения задачи *semi-local lcs*

График свидетельствует о том, что сложная рекурсивная структура алгоритма через быстрое умножение кос делает его неприменимым на практике для строк большой длины. Несмотря на это, такая структура алгоритма позволяет избавиться от квадратичной зависимости от параметра v при решении задачи *semi-local sa* (см. рис. 9). Соответственно, если реализовать итеративную версию рекурсивного алгоритма, то должно стать быстрее. Данное замечание также справедливо для рекурсивной версии алгоритма умножения липких кос, который непосредственно используется внутри рекурсивного алгоритма.

На рис. 10 представлены результаты запусков двух версий алгоритмов для решения задачи *Window-Substring*: через предподсчет кос (*implicit 350*) и наивный (*naive 350*)[25].

График показывает, что алгоритм для решения задачи *Window-Substring* через предподсчет кос действительно не зависит от размера окна и быстрее наивной версии. Несмотря на это, в силу рекурсивной

[25]Наивный алгоритм заключается в вычислении *semi-local lcs* для каждого окна ($m \times n \times w$), где w — размер окна, m, n — размеры строк.

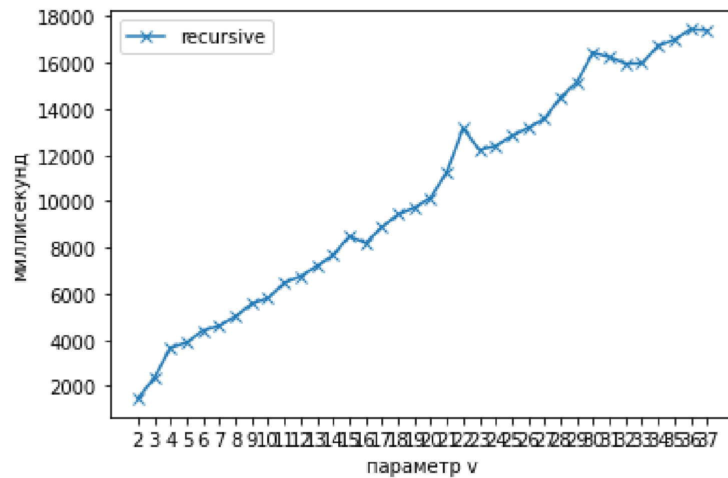


Рис. 9: Линейная зависимость параметра рекурсивного алгоритма *semi-local sa* на основе умножения липких кос

структуры алгоритма решения, он не применим к большим данным.

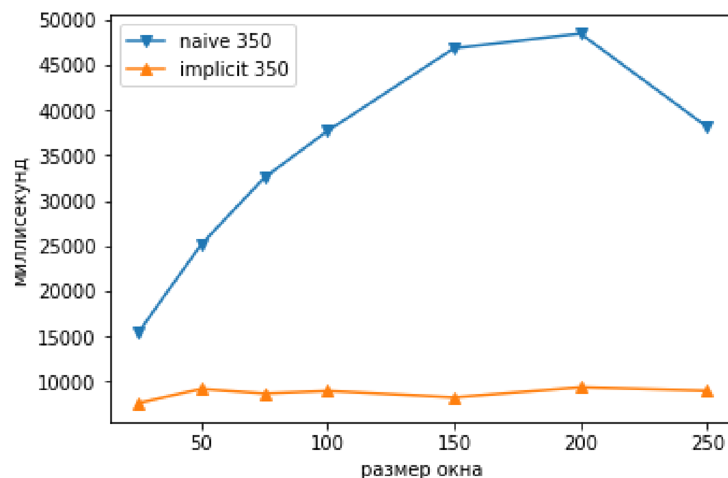


Рис. 10: Независимость от размера окна

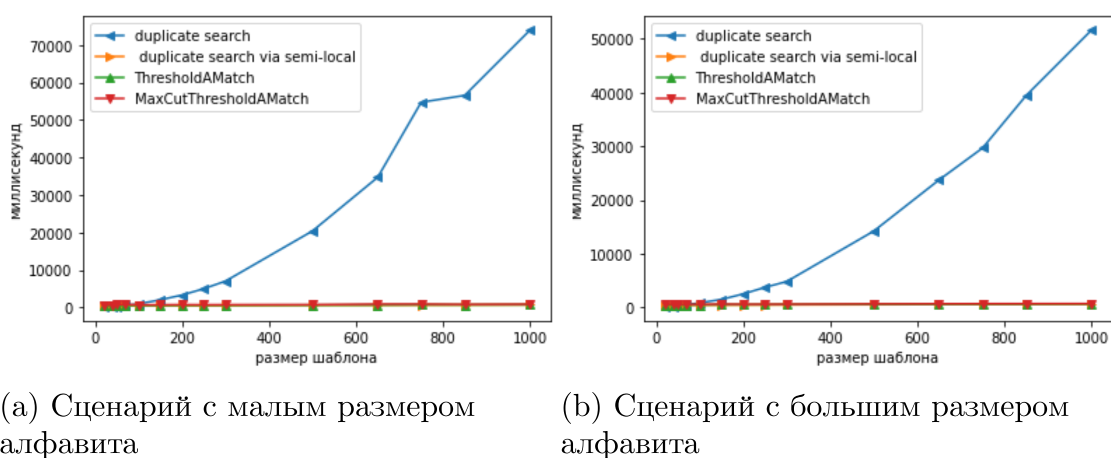
Таким образом, можно сделать вывод, что при работе с большими данными может быть эффективно применен алгоритм решения задач *semi-local* через распутывание кос. А алгоритмы, обладающие сложной рекурсивной структурой, могут быть применены при совершении ряда оптимизаций. Например, избавления от рекурсии.

5.3. Поиск по шаблону

В данной секции описаны результаты запусков различных версий алгоритмов решения *задачи поиска по шаблону*. Также произведен их анализ.

На рис. 11 представлены результаты временных замеров для различных версий алгоритмов, решающих задачу поиска по шаблону. В частности, алгоритм из статьи [17] (*duplicate search*), его улучшенная версия, описанная в разделе 4.4 (*duplicate search via semi-local*), алгоритм из *библиотеки алгоритмов* (*ThresholdAMatch*) и алгоритм 5 с учетом замечания (см. псевдокод 5).

Рассматривались два сценария: меленький размер алфавита (большая частота повторов) и большой размер алфавита (маленькая частота повторов). В обоих случая размер текста был фиксирован и равен *10000* слов.



(a) Сценарий с малым размером алфавита

(b) Сценарий с большим размером алфавита

Рис. 11: Сравнение скорости различных алгоритмов решения задачи *поиска по шаблону*

График, во-первых, показывает, что алгоритм решения *semi-local* может быть эффективно применен к *задаче поиска по образцу*, а во-вторых, что улучшенная версия алгоритма из [17] показывает лучшие результаты не только в теории, но и на практике.

5.4. Поиск групп повторов

В данной секции представлен анализ результатов запуска алгоритмов, решающих *задачу поиска групп повторов* для *JavaDoc*-документации.

Анализ производился в рамках реализованного приложения (см. главу 4). Использовались следующие фильтры: токенизация, лемматизация, удаление стоп-слов. Повторы искались среди *JavaDoc*-комментариев, относящихся к описанию методов, классов, интерфейсов.

Проекты, содержащие *JavaDoc*-документацию, были выбраны на основе статей из главы 2.1.

Для нахождения групп использовался алгоритм 7. В качестве функции g были выбраны алгоритмы из *библиотеки алгоритмов*, решающие задачи *bounded-length Smith-Waterman* и *semi-local sa.* В качестве s использовались алгоритм 9 и алгоритм из [34].

Результаты представлены в таблице 12.

Название проекта	Кол-во комментариев	Кол-во повторов	Кол-во групп	Время исполнения (сек)
slf4j	188	157	25	8
apache commons io	1284	1180	92	569
apache commons collection	610	495	50	408
gson	498	356	81	96
junit	680	539	87	163
mockito	2979	2812	164	2012
guava	4340	3662	418	8505

Рис. 12: Результат работы приложения

Пример группы одного из проектов представлен на рис. 6.

Анализ проектов с помощью приложения показал следующее.

Во-первых, построение полного графа похожести — очень долгая операция. Необходима предварительная фильтрация, чтобы сократить

количество ребер в графе.

Во-вторых, анализ найденных групп показал, что они преимущественно состоят из одинаковых фрагментов (одинаковые методы, схожая функциональность) с незначительными изменениями (в графе это клики). Сложная структура дерева (без учета клик) практически не присутствует.

В-третьих, графовые алгоритмы можно применять в задаче *поиска групп повторов*.

В-четвертых, алгоритмы на основе решения задач *semi-local* также могут быть успешно применены к задаче *поиска групп повторов*.

В-пятых, количество повторов в *JavaDoc*-документации существенно, что подтверждает результаты из статей (см. главу 2.1).

Заключение

В рамках выпускной квалификационной работы были выполнены следующие задачи.

- Исследованы существующие теоретические алгоритмы решения задачи полулокального поиска наибольшей общей подпоследовательности и выравнивания строк и реализованы в виде *библиотеки алгоритмов* на языке *Kotlin*.
- Создано приложение на языке *Kotlin* для поиска повторов в *JavaDoc* документации на основе адаптации алгоритмов решения полулокальных задач.
- Проведено экспериментальное исследование реализованных алгоритмов и анализ результатов.

Полученные в ходе работы результаты являются доказательством применимости алгоритмов решения полулокальных задач к поиску повторов в документации.

Список литературы

[1] Abboud Amir, Backurs Arturs, Williams Virginia Vassilevska. Tight hardness results for LCS and other sequence similarity measures // 2015 IEEE 56th Annual Symposium on Foundations of Computer Science / IEEE. — 2015. — P. 59–78.

[2] Analysis of 5'gene regions reveals extraordinary conservation of novel non-coding sequences in a wide range of animals / Nathaniel J Davies, Peter Krusche, Eran Tauber, Sascha Ott // BMC evolutionary biology. — 2015. — Vol. 15, no. 1. — P. 227.

[3] Arslan Abdullah N, Eğecioğlu Ömer. Dynamic programming based approximation algorithms for sequence alignment with constraints // INFORMS Journal on Computing. — 2004. — Vol. 16, no. 4. — P. 441–458.

[4] Blasi Arianna, Gorla Alessandra. Replicomment: identifying clones in code comments // Proceedings of the 26th Conference on Program Comprehension. — 2018. — P. 320–323.

[5] Can clone detection support quality assessments of requirements specifications? / Elmar Juergens, Florian Deissenboeck, Martin Feilkas et al. // Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. — 2010. — P. 79–88.

[6] Chomal Vikas S, Saini Jatinderkumar R. Significance of Software Documentation in Software Development Process // International Journal of Engineering Innovations and Research. — 2014. — Vol. 3, no. 4. — P. 410.

[7] Clone detection in reuse of software technical documentation / Dmitrij Koznov, Dmitry Luciv, Hamid Abdul Basit et al. // International Andrei Ershov Memorial Conference on Perspectives of System Informatics / Springer. — 2015. — P. 170–185.

[8] Conserved noncoding sequences highlight shared components of regulatory networks in dicotyledonous plants / Laura Baxter, Aleksey Jironkin, Richard Hickman et al. // The Plant Cell. — 2012. — Vol. 24, no. 10. — P. 3949–3965.

[9] Detecting near duplicates in software documentation / DV Luciv, DV Koznov, George A Chernishev et al. // Programming and Computer Software. — 2018. — Vol. 44, no. 5. — P. 335–343.

[10] Documentation reuse: Hot or not? An empirical study / Mohamed A Oumaziz, Alan Charpentier, Jean-Rémy Falleri, Xavier Blanc // International Conference on Software Reuse / Springer. — 2017. — P. 12–27.

[11] Dongen Stijn. A cluster algorithm for graphs. — 2000.

[12] Duplicate finder toolkit / Dmitry Luciv, Dmitrij Koznov, George Chernishev et al. // Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. — 2018. — P. 171–172.

[13] Evolutionary analysis of regulatory sequences (EARS) in plants / Emma Picot, Peter Krusche, Alexander Tiskin et al. // The Plant Journal. — 2010. — Vol. 64, no. 1. — P. 165–176.

[14] Gawrychowski Paweł, Mozes Shay, Weimann Oren. Submatrix Maximum Queries in Monge and Partial Monge Matrices Are Equivalent to Predecessor Search // ACM Transactions on Algorithms (TALG). — 2020. — Vol. 16, no. 2. — P. 1–24.

[15] Geometric applications of a matrix-searching algorithm / Alok Aggarwal, Maria M Klawe, Shlomo Moran et al. // Algorithmica. — 1987. — Vol. 2, no. 1-4. — P. 195–208.

[16] Horie Michihiro, Chiba Shigeru. Tool support for crosscutting concerns of API documentation // Proceedings of the 9th International

Conference on Aspect-Oriented Software Development. — 2010. — P. 97–108.

[17] Interactive Duplicate Search in Software Documentation / DV Luciv, DV Koznov, AA Shelikhovskii et al. // arXiv preprint arXiv:1908.08266. — 2019.

[18] Kipyegen Noela Jemutai, Korir William PK. Importance of software documentation // International Journal of Computer Science Issues (IJCSI). — 2013. — Vol. 10, no. 5. — P. 223.

[19] Koznov DV, Luciv DV, Chernishev GA. Duplicate management in software documentation maintenance // CEUR Workshop Proceedings / RWTH Aahen University. — Vol. 1989. — 2017. — P. 195–201.

[20] Krusche Peter, Tiskin Alexander. Parallel longest increasing subsequences in scalable time and memory // International Conference on Parallel Processing and Applied Mathematics / Springer. — 2009. — P. 176–185.

[21] On fuzzy repetitions detection in documentation reuse / DV Luciv, DV Koznov, Hamid Abdul Basit, Andrey N Terekhov // Programming and Computer Software. — 2016. — Vol. 42, no. 4. — P. 216–224.

[22] Plösch Reinhold, Dautovic Andreas, Saft Matthias. The value of software documentation quality // 2014 14th International Conference on Quality Software / IEEE. — 2014. — P. 333–342.

[23] Porubän Jaroslav et al. Reusable software documentation with phrase annotations // Central European Journal of Computer Science. — 2014. — Vol. 4, no. 4. — P. 242–258.

[24] 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.

[25] Rago Alejandro, Marcos Claudia, Diaz-Pace J Andres. Identifying duplicate functionality in textual use cases by aligning semantic actions // Software & Systems Modeling. — 2016. — Vol. 15, no. 2. — P. 579–603.

[26] Similarity-based support for text reuse in technical writing / Axel J Soto, Abidalrahman Mohammad, Andrew Albert et al. // Proceedings of the 2015 ACM Symposium on Document Engineering. — 2015. — P. 97–106.

[27] Tiskin Alexander. All semi-local longest common subsequences in subquadratic time // International Computer Science Symposium in Russia / Springer. — 2006. — P. 352–363.

[28] Tiskin Alexander. Longest common subsequences in permutations and maximum cliques in circle graphs // Annual Symposium on Combinatorial Pattern Matching / Springer. — 2006. — P. 270–281.

[29] Tiskin Alexander. Semi-local string comparison: algorithmic techniques and applications. — 2007. — 0707.3619.

[30] Tiskin Alexander. Semi-local string comparison: Algorithmic techniques and applications // Mathematics in Computer Science. — 2008. — Vol. 1, no. 4. — P. 571–603.

[31] Tiskin Alexander. Towards approximate matching in compressed strings: Local subsequence recognition // International Computer Science Symposium in Russia / Springer. — 2011. — P. 401–414.

[32] Tiskin Alexander. Fast distance multiplication of unit-Monge matrices // Algorithmica. — 2015. — Vol. 71, no. 4. — P. 859–888.

[33] Tiskin Alexander. Bounded-Length Smith-Waterman Alignment // 19th International Workshop on Algorithms in Bioinformatics (WABI 2019) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. — 2019.

[34] Tofigh Ali. Optimum branchings and spanning aborescences // Advanced Algorithms course notes, Sep. — 2009.