Санкт–Петербургский государственный университет

РОГАЧЁВ Юрий Витальевич

Выпускная квалификационная работа

Выявление признаков плагиата в исходных кодах программных систем

Уровень образования: бакалавриат

Направление 01.03.02 «Прикладная математика и информатика»

Основная образовательная программа СВ.5005.2017 «Прикладная математика, фундаментальная информатика и программирование»

Профиль «Математическое и программное обеспечение вычислительных машин»

Научный руководитель:

д.ф. - м.н., профессор Утешев А. Ю.

Соруководитель научной работы:

ст. преп., Давыденко А. А.

Рецензент:

д.ф. - м.н., профессор Гришкин В. М.

Санкт-Петербург

2021 г.

Содержание

1 Введение

1.1 Актуальность задачи

Дублирование исходного кода часто возникает в процессе разработки программного обеспечения. Этот дублированный код часто называют клонами кода, а процесс - клонированием кода. Исследования показали что 20-50% современного программного обеспечения состоит из склонированого кода [1]. Считается, что клоны кода имеют отрицательное влияние на качество кода, потому что они делают поддержку кода сложнее, а так же из-за них сложнее находить и устранять программные ошибки [4][11]. Также когда клонированный код появляется в кодовой базе, он так же может быть признаком нелегального переиспользования кода. Такое нелегальное переиспользование кода в программном обеспечении с открытым исходным кодом, как правило, принимает форму нарушения лицензии. Исследования выделяют две основные причины по которым разработчики клонируют код. Первая причина состоит в том, что разработчики программного обеспечения с открытым исходным кодом переиспользуют код с целью не фокусироваться на некоторых тривиальных задачах и потому что часто ограничены во времени и ресурсах [16]. Второй причиной выделяют тот факт что существует множество заблуждений о природе лицензий программного обеспечения с открытым исходным кодом, и это проблема только усугубляется большим количеством существующих лицензий для программного обеспечения с открытым исходным, что ведет к тому, что разработчики не до конца понимают условия лицензий и разницу между ними [3]. Нарушение лицензии программного кода может быть причиной судебных разбирательств, поэтому в наличии инструментов поиска склонированного кода заинтересованы не только разработчики программного обеспечения, но и их работодатели поскольку именно они будут нести ответственность за нарушенные лицензии.

1.2 Обзор литературы

Ранние работы, затрагивающие поиск похожих фрагментов кода, использовали прямолинейные подходы для поиска возможных дубликатов кода в больших кодовых базах. В этих работах анализировались названия директорий и файлов и находились директории в которых дословно совпадает большое количество файлов с исходным кодом [7][8]. Такие подходы позволяют корректно идентифицировать точные копии проектов и хорошо расширяется на очень большие объемы данных, но такие методы невозможно применять для выявления плагиата в коде, потому что уровень похожести должен быть определен непосредственно для кода.

Одним из первых программных инструментов, позволяющим производить поиск клонированного кода на уровне функций, при этом расширяющийся на большие объемы данных является SourcererCC [14]. Этот инструмент позволял производить эффективный поиск клонов, используя анализ токенов, использующихся в исходном коде. Данное решение будет подробно рассмотрено далее в тексте, вместе с дальнейшими модификациями данного подхода. Несмотря на эффективность, подходы основанные на анализе токенов не позволяют надежно находить некоторые типы клонов, в частности они не позволяют находить структурные и семантические клоны.

Подходы для поиска дубликатов в коде с использованием структурных признаков, как правило, используют информацию, которую можно получить из синтаксического дерева и используют методы для поиска похожих поддеревьев [5], из-за чего они обладают как минимум квадратичной асимптотикой и плохо масштабируются на большие объемы кода.

Так же существуют подходы которые ищут дубликаты кода, используя семантическую информацию. Зачастую такие подходы используют алгоритмы машинного обучения. Для подходов основанных на машинном обучении крайне важным является способ извлечения признаков из кода. В качестве признаков может быть использован код в виде текста, синтаксические дере-

вья и даже байткод [13].

1.3 Определения

Далее приведены общепринятые определения[2][12][14], используемые в рамках данной работы

Фрагмент кода: Непрерывный сегмент исходного кода, определяемый как триплет (l, s, e), где l - файл с исходным кодом, s - номер строки на котором начинается данный фрагмент, e - номер строки на котором он заканчивается.

Пара клонов: Пара похожих фрагментов кода, определяемая как триплет (f_1, f_2, ϕ), где f_1, f_2 - фрагменты кода, а ϕ - тип клона.

Класс клонов: Множество похожих фрагментов кода, определяемая как кортеж $(f_1, f_2, \ldots, f_n, \phi)$. Каждая пара различных фрагментов является парой клонов: $(f_i, f_j, \phi), i, j \in 1..n, i \neq j$.

Блок кода: Последовательность операций, объявлений классов и переменных внутри фигурных скобок или иных ограничителей зон видимости.

Проект: Проект P является некоторым набором фрагментов кода, предположим что он представлен в виде коллекции блоков кода $P = \{B_1, \ldots, B_n\}$.

Клоны первого типа: Идентичные фрагменты кода, за исключением расположение пробелов и комментариев.

Клоны второго типа: Идентичные фрагменты кода, за исключением разницы в названиях переменных и литералов, в дополнение к разнице первого типа.

Клоны третьего типа: Синтаксически похожие фрагменты кода, которые отличаются на уровне операций. В фрагментах присутствуют операции которые добавлены, изменены и/или убраны по отношению друг к другу в дополнение к разнице первого и второго типа

Клоны четвертого типа: Синтаксически не похожие фрагменты кода, реализующие одинаковый функционал.

1.4 Постановка задачи

Обобщенно задача формулируется следующим образом.

1. Ввести метрику похожести f принимающую на вход два блока кода, и возвращающую неотрицательное вещественное число, отражающее степень похожести двух блоков кода.

2. Разработать инструмент, который используя данную метрику находит все пары блоков кода B_i, B_j в некотором проекте P такие, что $f(B_i, B_j) \geq \theta$, где θ заранее заданный порог.

Уточним постановку задачи, введя дополнительные определения. Предположим, что проект P представлен как набор блоков кода $P : B_1, \ldots, B_n$. В свою очередь блок кода B представлен как мешок токенов $B : T_1, \ldots, T_k$. Под **токеном** мы понимаем ключевое слово языка программирования, литерал или идентификатор. Строковые литералы разделены по пробелам.

Чтобы численно показать, что два блока кода являются клонами мы будем пользоваться метрикой похожести, которая будет измерять похожесть двух блоков кода друг на друга и возвращать неотрицательное число. Чем больше значение метрики, тем более похожи два фрагмента друг на друга. В результате блоки кода с метрикой похожести, большей чем некоторый заранее заданный порог, будут считаться клонами.

Формально: имея проект P, метрику похожести f, пороговое значение значение θ, цель найти все пары блоков кода $P.B_x$ и $P.B_y$, такие что: $f(P.B_x, P.B_y) \geq \lceil \theta * max(|P.B_x|, |P.B_y|) \rceil$

2 Наивный алгоритм

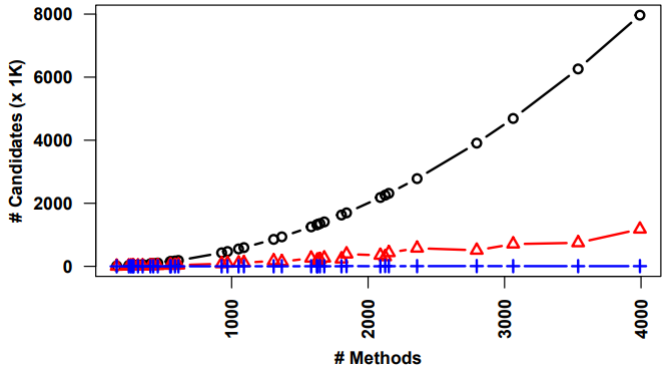
В качестве метрики похожести можно взять большое количество различных функций. Простейшей из них, пожалуй, будет является функция пересечения двух множеств. Она является неплохим выбором, поскольку интуитивно передает смысл пересечения двух блоков кода. Например, пусть нам даны два блока кода B_x и B_y, тогда пересечение этих двух блоков $O(B_x, B_y)$ будет вычислено как количество общих токенов в B_x и B_y.

$$O(B_x, B_y) = |B_x \cap B_y|$$

Чтобы найти все пары клонов в проекте можно проитерироваться по всем блокам кода в проекте и вычислить пересечение все пар блоков кода. Для данного блока кода все другие блоки кода с которыми осуществляется сравнение называются **блоками кандидатами** или сокращенно **кандидатами**.

Такой подход является простым и интуитивно понятным, но у него есть существенный недостаток - он плохо масштабируется на большое количество блоков кода, из-за того что он имеет асимптотику $O(n^2)$. На Рис. 1 представлен график зависимости количества кандидатов (по оси Y) от количества блоков кода (по оси X). Точки отмеченные символом ∘ показывают что количество кандидатов возрастает квадратично.

Рис. 1: Рост числа кандидатов в зависимости от количества блоков кода



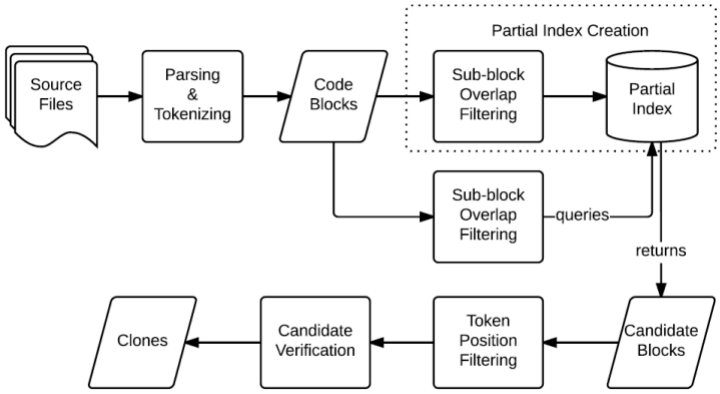
3 SourcererCC

В статье SourcererCC: Scaling Code Clone Detection to Big Code [14] был представлен один из первых эффективных алгоритмов поиска клонов в коде, основанный на анализе токенов. Далее приведен обзор данного подхода. Многие технические детали и детали реализации были опущены.

3.1 Обзор подхода

Обобщенно процедура поиска клонов, использованная в SourcererCC, представлена на Рис. 2. Глобально процедура состоит из двух шагов: (i) построение частичного индекса, (ii) поиск клонов.

Рис. 2: Процедура поиска клонов, использованная в SourcererCC



На этапе построения индекса выполняются следующие шаги:

1. Парсинг блоков кода из файлов с исходным кодом
2. Токенизация блоков кода
3. Из блоков кода происходит построение инвертированного индекса: отображение токенов к блокам кода которые содержат данный токен. Но в индекс попадают блоки не целиком, вместо этого используется фильтрующая эвристика из секции 3.2 и производиться построение частичного индекса на некотором подмножестве токенов из каждого блока.

На этапе поиска клонов выполняются следующие шаги:

1. Итерирование по всем блокам кода

2. Получение блоков кандидатов посредством запроса из индекса. На данном шаге используется фильтрующая эвристика (секция 3.2) позволяющая значительно сократить количество кандидатов. Она будет описана далее.

3. Получение верхних и нижних границ метрики похожести между запросом и кандидатами. На данном шаге используется еще одна фильтрующая эвристика (секция 3.3) использующая особый порядок токенов в блоке кода. Эта эвристика также будет описана далее.

4. Отсев кандидатов чья верхняя граница метрики меньше чем пороговое значение.

5. Кандидаты чья нижняя граница больше чем пороговое значение помечаются как клоны.

3.2 Фильтрация по пересечениям под-блоков

За следующими эвристиками стоит следующее интуитивное соображение: если два множества имеют большое пересечение, то даже их более маленькие подмножества будут пересекаться. Поскольку мы представляем блоки кода как мешки токенов (т.е. мультимножество) то мы можем расширить эту идею и на блоки кода. Формально это задается следующим свойством

Свойство 1.: Пусть даны два блока кода B_x и B_y состоящие из t токенов в некотором заранее определенном порядке, если $|B_x \cap B_y| \geq i$, тогда подблоки SB_x и SB_y, из B_x и B_y соответственно, должны иметь хотя бы один общий токен среди первых $t - i + 1$ токенов.

Чтобы понять что нам дает это свойство рассмотрим следующий пример. Пусть $B_x = \{\mathbf{a}, \mathbf{b}, c, d, e\}$ и $B_y = \{\mathbf{c}, \mathbf{d}, e, f, g\}$, тогда $t = 5$. Пусть $\theta = 0.8$,

тогда чтобы 2 блока длины t считались клонами они должны совпасть как минимум в 4 токенах ($i = 4$). По свойству 1 если среди $t - i + 1 = 2$ токенов первых токенов нет ни одного общего токена, то блоки кода B_x и B_y не могут быть клонами для заданного θ. То есть из-за этого свойства мы можем, посмотрев только на первые $t - i + 1$ токенов, заключить что блоки B_x и B_y не могут быть клонами.

Для выполнения этого свойства необходимо чтобы блоки кода были даны в некотором порядке. Несмотря на то что существует множество способов упорядочить токены, мы можем выделить самый эффективный метод упорядочивая. Оказывается, что вокабуляр языков программирования подчиняется закону Ципфа [10][18]. Это значит что существует небольшое количество часто встречаемых токенов и частота встречания токенов быстро убывает. Другими словами большинство блоков кода будут содержать некоторые часто встречаемые токены (например ключевые слова), но немногие они будут содержать мало редких токенов (например названия переменных) и они редко будут одинаковыми. Соответственно если упорядочить токены по частоте встречания, то подблоки и будут состоять из редких токенов. Другими словами такое упорядочивание уберет наибольшее количество ложно положительных кандидатов. На Рис. 1 точки обозначенные $\triangle$ показывают количество кандидатов после применения данной фильтрации.

3.3 Фильтрация по позициям токенов

Свойства 1 может быть неэффективным для некоторых блоков кода. Например блоки $B_x = \{\mathbf{a}, \mathbf{b}, c, d\}$ и $B_y = \{\mathbf{b}, \mathbf{c}, d, e, f\}$ по свойству 1 будут считаться кандидатами.

Можно заметить что используя позиции токенов можно дать верхнюю оценку на пересечение двух блоков кода B_x и B_y как сумму текущего количества совпадающих токенов и минимальному количеству токенов из B_x и B_y которые мы еще не успели обработать, т. е. $1 + min(2, 4) = 3$. Исполь-

зуя эту оценку можно отсеивать кандидаты которые точно не будут являться клонами. Формально это можно сформулировать в виде:

Свойство 2.: Пусть даны два упорядоченные блоки кода B_x и B_y и $\exists$ токен t в позиции i в B_x. Разделим B_x на две части: $B_x(first) = B_x[1 \dots (i-1)]$ и $B_x(second) = B_x[i \dots |B_x|]$. Теперь если $|B_x \cap B_y| \geq \lceil \theta * max(|B_x|, |B_y|) \rceil$, тогда $\forall t \in B_x \cap B_y, |B_x(first) \cap B_y(first)| + min(|B_x(second)|, |B_y(second|) \geq \lceil \theta * max(|B_x|, |B_y|) \rceil$

На Рис. 1 точки обозначенные $+$ показывают количество кандидатов после применения данной фильтрации. Таким образом, используя свойство 1 и 2 можно сократить количество сравнений с квадратической функции до почти линейной

3.4 Алгоритм поиска клонов

На рисунке 2 представлен алгоритм поиска клонов, используемый в SourcererCC. Далее приведен код на языке программирования python полного алгоритма поиска клонов.

```
# INPUT: B - массив блоков кода {b_1, ..., b_n}
#        GTP - глобальный словарь позиций токенов
#        theta - пороговое значение
# OUTPUT: Частичный индекс (I) B
def createPartialIndex(B, theta):
    I = {}
    for каждого блока кода b in B:
        b = Sort(b, GTP)
        tokensToBeIndexed = |b| - |theta * |b|| + 1

        for i = 1 : tokensToBeIndexed:
            t = b[i]
            I = I + (t, i)
```

```
    return I

# INPUT: B - массив блоков кода {b_1, ..., b_n}
#        I - частичный индекс созданный для B
#        theta - пороговое значение
# OUTPUT: все клоны из B
def detectClones(B, I, theta):
    for каждого блока кода  b in B:
        candidateSimilarityMap = {}
        b = Sort(b, GTP)
        querySubBlock = |b| - |theta * |b|| + 1

        for i = 1 : querySubBlock:
            t = b[i]

            for each (c, j) in I such that |c| > |theta * |b||:
                ct = |max(|c|, |b|) * theta|
                uBound = 1 + |min(|b| - i, |c| - j)|

                if candidateSimilarityMap[c] + uBound >= ct:
                    candidateSimilarityMap[c] += (1, j)
                else:
                    candidateSimilarityMap[c] = (0, 0)

            verifyCandidates(b, candidateSimilarityMap, ct)
    return cloneMap

# INPUT: B - массив блоков кода {b_1, ..., b_n}
#        I - частичный индекс созданный для B
#        theta - пороговое значение
```

```
# OUTPUT: все клоны из B
def verifyCandidates(b, candidateSimilarityMap, ct):
    for каждого c в (candidateSimilarityMap такого что
            candidateSimilarityMap[c] > 0):

        tokPos_c = Позиция последнего обработанного токена в c
        tokPos_b = Позиция последнего обработанного токена в b

        while tokPos_b < |b| and tokPos_c < |c|:
            if min(|b| - tokPos_b, |c| - tokPos_c) >= ct:
                if b[tokPos_b] == c[tokPos_c]:
                    candidateSimilarityMap[c] += 1
                else:
                    if GTP[b[topPos_b]] < GTP[c[tokPos_c]]:
                        tokPos_b++
                    else:
                        tokPos_c++
            else:
                break

        if candidateSimilarityMap[c] >= ct:
            cloneMap[b] += c
```

3.5 Особенности данного подхода

У данного алгоритма есть три важные особенности:

1. Асимптотика работы данного алгоритма в среднем почти линейна
2. Данный алгоритм не теряет точности при нахождении клонов 1 и 2 типа

3. В отличии от ранних подходов, основанных на анализе токенов, он способен находит клоны 3 типа. Это происходит из-за того что блоки кода представлены в виде мешков токенов, а они устойчивы к изменениям кода, свойственным для клонов 3-го типа.

4 Использование адаптивных префиксов в алгоритме SourcererCC

В статье Scalable Code Clone Detection and Search based on Adaptive Prefix Filtering [9] был усовершенствован алгоритм поиска клонов из SourcererCC. А именно была оптимизирована фильтрация по пересечениям под-боков путем введения фильтрации по адаптивным префиксам под-блоков.

4.1 Фильтрация по адаптивным префиксам под-блоков

Фильтрация по адаптивным префиксам под-блоков является расширением идеи фильтрации по пересечениям под-блоков из пункта 3.2. Идея состоит в том что мы будем вместо совпадения одного токена в блоках кода искать несколько совпадений в зависимости от длины этих блоков. Такой вариант совпадения будем называть $l-$ префиксом (тогда ранее мы рассматривали 1-префикс), где l - количество совпадений токенов которого мы хотим добиться. Фильтрация по адаптивным префиксам более агрессивно отсеивает кандидатов за счет большего количества сравнений на шаге фильтрации. Сформулируем формально свойство которое нам дает такая фильтрация.

Свойство 3: Пусть даны два блока кода B_x и B_y, каждый состоящий из t токенов в некотором заранее определенном порядке, если $|B_x \cap B_y| \geq i$, тогда подблоки SB_x и SB_y из B_x и B_y соответственно, каждый из которых имеет длину $t - i + l$ будут совпадать в как минимум l токенах.

Упомянем лемму, которая позволит нам использовать фильтрацию по адаптивным префиксам вместо обычной фильтрации по префиксам.

Лемма 1: Для любой пары блоков кода B_x и B_y, если $P_l(B_x) \cap P_l(B_y) < l$, тогда $|B_x \cap B_y| < \lceil \theta * max(|B_x|, |B_y|) \rceil$ [17].

Здесь $P_l(B_x)$ и $P_l(B_x)$ обозначают $l-$ префиксы блоков B_x и B_y, где каждый $l-$префикс состоит из первых $max(|B_x|, |B_y|) - \lceil \theta * max(|B_x|, |B_y|) \rceil + l$ элементов.

4.2 Использование фильтрации по адаптивным префиксам на практике

Структура данных инвертированный индекс часто используется на практике для получения совпадающих документов (в нашем случае блоков кода) используя некоторый токен в качестве запроса. Для фильтрации по префиксам необходимо построение всего одного инвертированного индекса. Вместо создания индекса для каждого отдельного документа инвертированный индекс строится для каждого токена и хранит все документы в которых встречается данный токен. Но для фильтрации по адаптивным префиксам необходимо построение такого индекса для каждой из $l-$ префикс схемы. Дельта инвертированный индекс можно использовать для избежания дублирования которое возникло бы если бы мы просто построили инвертированный индекс для каждой схемы.

Инвертированный индекс $I_l(e)$ хранит в себе все блоки кода чьи $l-$префиксы содержат токен e. Аналогично $I_{l+1}(e)$ хранит в себе все блоки кода чьи $l+1-$ префиксы содержат токен e и $I_l(e) \subseteq I_{l+1}(e)$. Дельта инвертированный индекс $\Delta I_{l+1}(e)$ - это структура данных которая хранит в себе только документы встречающиеся в $I_{l+1}(e)$ и не встречающиеся в $I_l(e)$. Тогда $\Delta I_1(e) = I_1(e)$ и мы создаем дельта инвертированные индексы $\Delta I_2(e), \Delta I_3(e), \ldots \Delta I_l(e)$ для $I_1(e), I_2(e), \ldots I_l(e)$

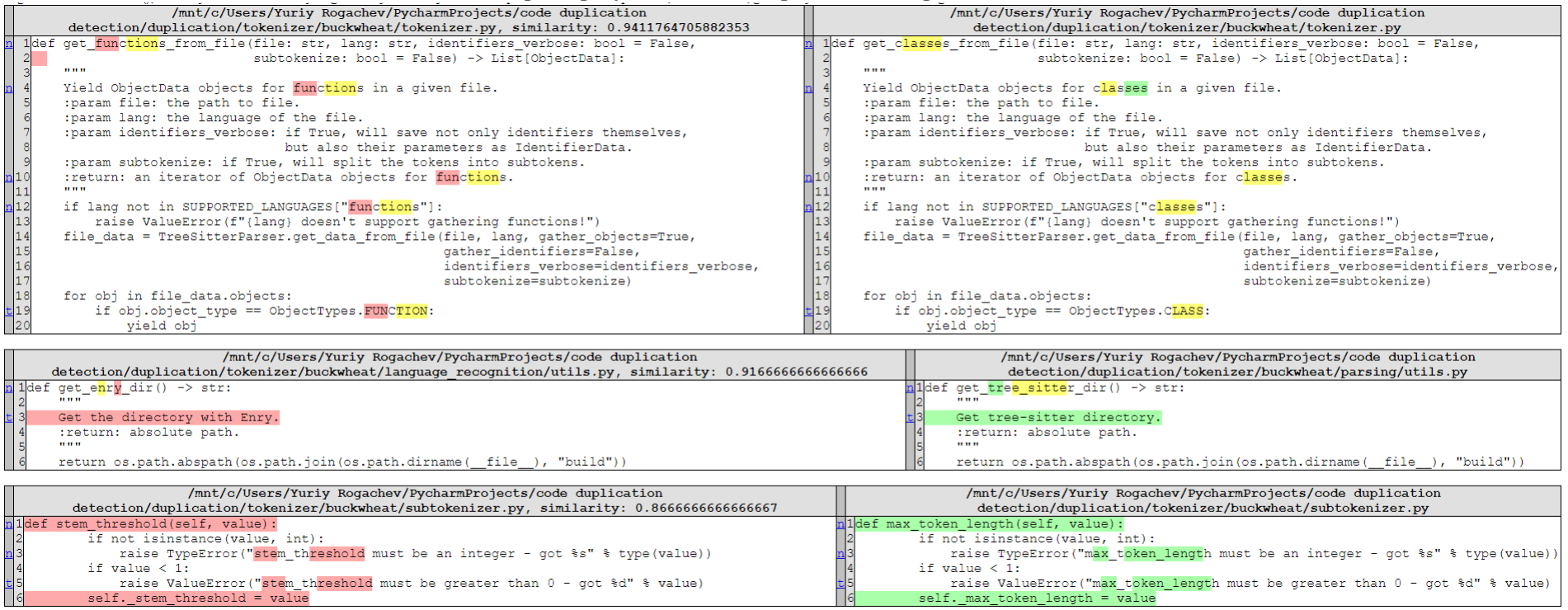
5 Практические результаты

В рамках практической части данной работы было реализовано консольное приложение и *python*-пакет для поиска дубликатов кода под названием *potator*. Данное приложение реализует вышеописанный алгоритм поиска с возможностью использования фильтрации по адаптивным префиксам, но с некоторыми отличиями от оригинальных алгоритмов. В данной секции будут описаны отличия от существующих подходов, приведен пример результата работы программы, а также поскольку *potator* доступен в виде *python*-пакета будут описаны основные возможности этого пакета, алгоритмы и модели данных реализованные в нем.

5.1 Возможности программного средства *potator*

Используя программное средство *potator* возможно быстро находить дубликаты кода в директориях с файлами с исходным кодом. Результат работы *potator* будет представлен в виде HTML файла в котором можно увидеть все дубликаты кода, а так же метрику похожести и различия между ними. Данное приложение имеет открытый исходный код. Его можно установить как *python*-пакет выполнив команду `pip install potator`. Исходный код доступен по ссылке https://github.com/otzhora/potator вместе с инструкциями по способам установки. Используя *potator* можно автоматизировать поиск дубликатов, например можно использовать его для проверки на наличие дублирования как один из шагов сборки проектов в которых используется практика непрерывной интеграции. Программное средство *potator* поставляется по лицензии MIT и имеет открытый исходный код, а значит существует возможность для других разработчиков легко расширять функционал данного пакета, например, добавляя в него новые алгоритмы поиска и переиспользовать исходный код *potator* без нарушения лицензии. Пример дубликатов которые находит *potator* представлены на рисунке 3. Сравнение с программным средством SourcererCC приведено в секции 5.5.

Рис. 3: Пример дубликатов которые находит *potator*



5.2 Отличия от вышеописанных алгоритмов

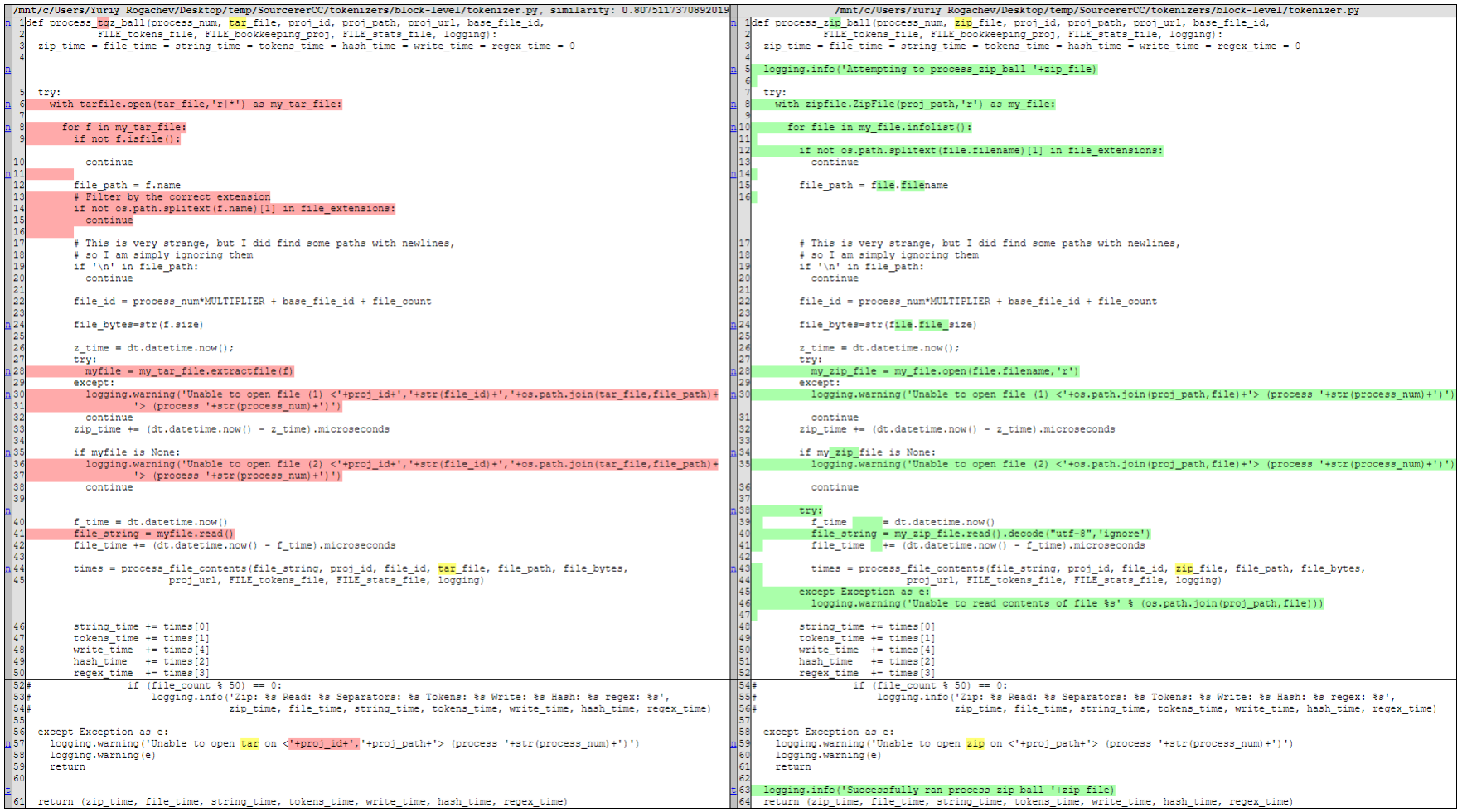
В своей реализации я использовал в качестве метрики похожести обобщенную меру Жаккара [15].

$$J(B_x, B_y) = \frac{|B_x \cap B_y|}{max(|B_x|, |B_y|)} = \frac{O(B_x, B_y)}{max(|B_x|, |B_y|)}$$

Она эквивалента пересечению и для нее выполняются все вышеописанные свойства, но формулы в них упрощаются.

В *potator* существует возможность осуществлять поиск не по самим токенам, а по субтокенам [6]. В процессе разработки программного обеспечения программисты часто называют свои переменные названиями состоящими из нескольких слов, например *SmithWatermanAnnealingProcessor*. В данном примере токеном будет являться названием переменной, а субтокенами слова из которых это название состоит, т.е *smith*, *waterman*, *annealing*, *processor*. Такое разбиение на токены позволяет находить соответствия не только используя токены целиком, но и их составные части, например если в проекте существует другая переменная с названием *HillerAndGreenAnnealingProcessor*, то разбив эти токены на субтокены мы сможем найти больше соответствий между ними. Таким образом используя данную модификацию можно получить более полное множество дубликатов. Алгоритм используемые для субтокенизации будет описан далее. На рисунке 4 представлен пример дубликатов кода который не находит SourcererCC, но находит potator благодаря использованию субтокенизации.

Рис. 4: Дубликаты которое находит *potator*, но не находит SourcererCC



/mnt/c/Users/Yuriy Rogachev/Desktop/temp/SourcererCC/tokenizers/block-level/tokenizer.py, similarity: 0.8075117370892019

```
def process_tgz_ball(process_num, tar_file, proj_id, proj_path, proj_url, base_file_id,
          FILE_tokens_file, FILE_bookkeeping_proj, FILE_stats_file, logging):
  zip_time = file_time = string_time = tokens_time = hash_time = write_time = regex_time = 0



  try:
    with tarfile.open(tar_file,'r|*') as my_tar_file:

      for f in my_tar_file:
        if not f.isfile():

          continue

        file_path = f.name
        # Filter by the correct extension
        if not os.path.splitext(f.name)[1] in file_extensions:
          continue

        # This is very strange, but I did find some paths with newlines,
        # so I am simply ignoring them
        if '\n' in file_path:
          continue

        file_id = process_num*MULTIPLIER + base_file_id + file_count

        file_bytes=str(f.size)

        z_time = dt.datetime.now();
        try:
          myfile = my_tar_file.extractfile(f)
        except:
          logging.warning('Unable to open file (1) <'+proj_id+','+str(file_id)+','+os.path.join(tar_file,file_path)+
                  '> (process '+str(process_num)+')')
          continue
        zip_time += (dt.datetime.now() - z_time).microseconds

        if myfile is None:
          logging.warning('Unable to open file (2) <'+proj_id+','+str(file_id)+','+os.path.join(tar_file,file_path)+
                  '> (process '+str(process_num)+')')
          continue


        f_time = dt.datetime.now()
        file_string = myfile.read()
        file_time += (dt.datetime.now() - f_time).microseconds

        times = process_file_contents(file_string, proj_id, file_id, tar_file, file_path, file_bytes,
                        proj_url, FILE_tokens_file, FILE_stats_file, logging)



        string_time += times[0]
        tokens_time += times[1]
        write_time  += times[4]
        hash_time   += times[2]
        regex_time  += times[3]
#               if (file_count % 50) == 0:
#                  logging.info('Zip: %s Read: %s Separators: %s Tokens: %s Write: %s Hash: %s regex: %s',
#                               zip_time, file_time, string_time, tokens_time, write_time, hash_time, regex_time)

  except Exception as e:
    logging.warning('Unable to open tar on <'+proj_id+','+proj_path+'> (process '+str(process_num)+')')
    logging.warning(e)
    return


  return (zip_time, file_time, string_time, tokens_time, write_time, hash_time, regex_time)
```

/mnt/c/Users/Yuriy Rogachev/Desktop/temp/SourcererCC/tokenizers/block-level/tokenizer.py

```
def process_zip_ball(process_num, zip_file, proj_id, proj_path, proj_url, base_file_id,
          FILE_tokens_file, FILE_bookkeeping_proj, FILE_stats_file, logging):
  zip_time = file_time = string_time = tokens_time = hash_time = write_time = regex_time = 0

  logging.info('Attempting to process_zip_ball '+zip_file)

  try:
    with zipfile.ZipFile(proj_path,'r') as my_file:

      for file in my_file.infolist():

        if not os.path.splitext(file.filename)[1] in file_extensions:
          continue

        file_path = file.filename




        # This is very strange, but I did find some paths with newlines,
        # so I am simply ignoring them
        if '\n' in file_path:
          continue

        file_id = process_num*MULTIPLIER + base_file_id + file_count

        file_bytes=str(file.file_size)

        z_time = dt.datetime.now();
        try:
          my_zip_file = my_file.open(file.filename,'r')
        except:
          logging.warning('Unable to open file (1) <'+os.path.join(proj_path,file)+'> (process '+str(process_num)+')')

          continue
        zip_time += (dt.datetime.now() - z_time).microseconds

        if my_zip_file is None:
          logging.warning('Unable to open file (2) <'+os.path.join(proj_path,file)+'> (process '+str(process_num)+')')

          continue

        try:
          f_time      = dt.datetime.now()
          file_string = my_zip_file.read().decode("utf-8",'ignore')
          file_time   += (dt.datetime.now() - f_time).microseconds

          times = process_file_contents(file_string, proj_id, file_id, zip_file, file_path, file_bytes,
                          proj_url, FILE_tokens_file, FILE_stats_file, logging)
        except Exception as e:
          logging.warning('Unable to read contents of file %s' % (os.path.join(proj_path,file)))

        string_time += times[0]
        tokens_time += times[1]
        write_time  += times[4]
        hash_time   += times[2]
        regex_time  += times[3]
#               if (file_count % 50) == 0:
#                  logging.info('Zip: %s Read: %s Separators: %s Tokens: %s Write: %s Hash: %s regex: %s',
#                               zip_time, file_time, string_time, tokens_time, write_time, hash_time, regex_time)

  except Exception as e:
    logging.warning('Unable to open zip on <'+proj_path+'> (process '+str(process_num)+')')
    logging.warning(e)
    return

  logging.info('Successfully ran process_zip_ball '+zip_file)
  return (zip_time, file_time, string_time, tokens_time, write_time, hash_time, regex_time)
```

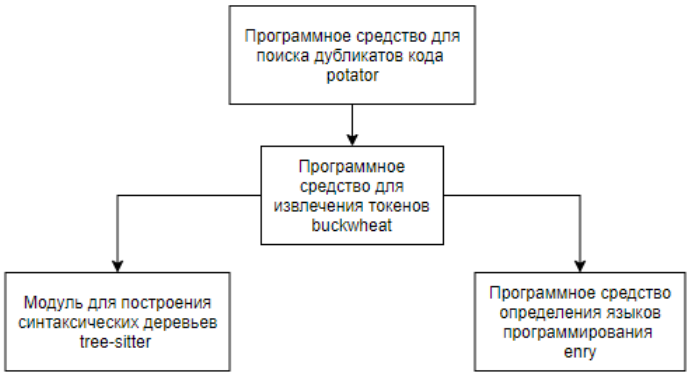
5.3 Описание моделей и абстракций используемых в пакете *potator*

Для извлечения токенов из директории с файлами с исходным кодом используется модифицированная версия программного пакета *buckwheat*. Этот пакет состоит из двух модулей: парсер и модуль для распознавания языков программирования в файлах с исходным кодом.

В основе модуля для распознавания языков программирования лежит программный пакет *enry*. *enry* написан на языке программирования *go* и состоит из большого числа эвиристик, позволяющих быстро и точно распознавать языки программирования. Поскольку enry является консольным приложением в buckwheat реализована функция которая представляет собой абстракцию над вызовом enry в консоле.

Парсер основан на модуле *tree-sitter*. *tree-sitter* позволяет строить синтаксические деревья для широкого спектра языков программирования. В buckwheat реализован класс, представляющий уровень абстракции над методами *tree-sitter*, которые, в частности, позволяют извлекать токены из вершин синтаксических деревьев которые строит *tree-sitter*. Программные зависимости *potator* представлены на рисунке 5.

Рис. 5: Программные зависимости *potator*



Для работы *tree-sitter*'у необходимо знать грамматики языков программирования на которых предстоит строить синтаксические деревья. Эти грамма-

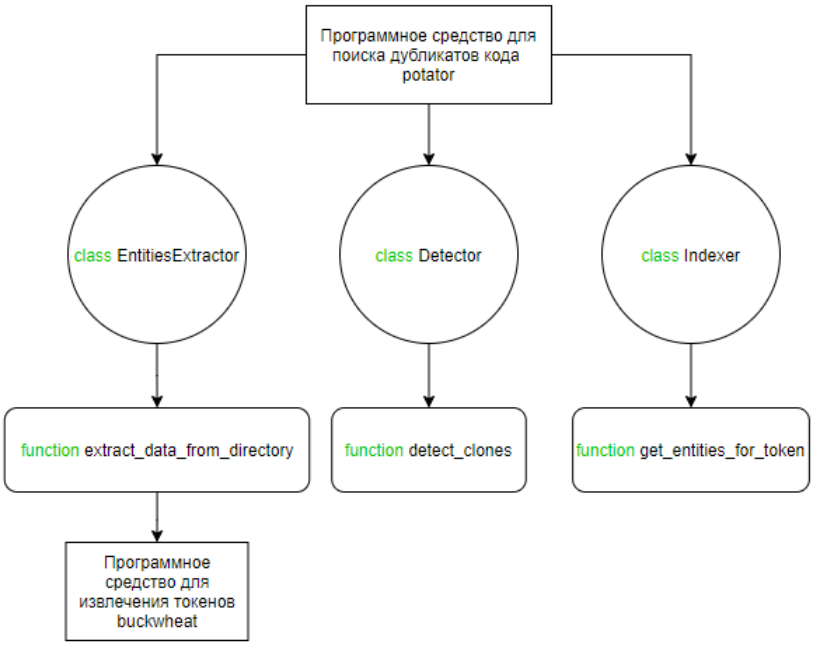
тики поставляются в виде отдельных пакетов и в *buckwheat* не было методов автоматической загрузки и настройки этих пакетов. Так же в *buckwheat* не было методов для автоматической сборки *enry*. То есть каждый раз перед его использованием требовалась ручная настройка. В данной работе использовалась модифицированная версия *buckwheat*. Модификация *buckwheat* состоит в добавлении в него методов для автоматической подготовке к работе и обновлении внутренних абстракций для работы с новейшими версиями *tree-sitter* и *enry*. Таким стало возможно использовать его внутри *python*-пакета.

В potator реализован класс представляющий еще один уровень абстракции над методами *buckwheat*. Этот класс называется *EntityExtractor* и его задача извлекать из директории информацию о файлах, извлекать из файлов функции и классы, а также извлекать из функций и классов токены, делить токены на субтокены и представлять все это в удобном для работы виде, а именно в качестве экземпляров класса *EntityData*. *EntityData* хранит в себе данные о том из какого файла были извлечены данные, язык программирования на котором написан этот файл, исходный код из которого были извлечены данные и сами токены (в зависимости от параметров они могут быть как самими токенами, так и субтокенами). Каждый экземпляр *EntityData* соответствует какой-то фунцкии или классу из файлов с исходным кодом. Основным методом этого класса является метод *extract_data_from_directory*. Этот метод принимает на вход директорию с файлами с исходным кодом, а затем для каждого фала распознает язык программирования и при помощи buckwheat осуществляет извлечение токенов с дальнейшей субтокенизацией. Этот метод возвращает массив файлов с соответствующими им языками программирования и массив *EntityData* для каждой функции и класса из файлов с исходным кодом.

В potator реализовано несколько алгоритмов поиска дубликатов исходного кода. Эти алгоритмы реализованы в классах *NaiveDetector* и *FilteringDetector*. Оба эти класса реализуют интерфейс *Detector*. Этот интерфейс требует от них обязательной реализации метода *detect*, который принимает два аргумен-

та: директорию с файлами с исходным кодом и пороговое значение для метрики похожести. Этот метод возвращает экземпляр *DetectionResult*, который хранит в себе пары дубликатов кода вместе с информацией о том какой алгоритм был использован. Внутри себя этот метод использует *EntityExtractor* для извлечения токенов из директории в удобном для работы виде. Алгоритм который использует *NaiveDetector* описан в секции 2, а *FilteringDetector* реализует алгоритм из SourcererCC с использованием фильтрации по адаптивному префиксу и принимает параметр l в качестве аргумента при создании экземпляра *FilteringDetector*. Для работы *FilteringDetector* необходимо построение дельта инвертированного индекса и в potator для работы с ними реализован класс *Indexer*. Его задача строить дельта инвертированные индексы для заданного параметра l и уметь выдавать все экземпляры EntityData чьи l – префиксы содержат некоторый токен. Упрощенно устройство potator показано на рисунке 6.

Рис. 6: Упрощенное устройство программного средства potator



5.4 Результаты работы программного средства *potator*

Поскольку *FilteringDetector* является лишь оптимизацией *NaiveDetector*, то дубликаты кода которые они находят в точности совпадают. Также если не применять субтокенизацию, то дубликаты кода которые находит potator в точности совпадают с дубликатами которые находит *SourcererCC*. Было экспериментально замечено, что использование субтокенизации увеличивает количество дубликатов которые находит potator на приблизительно 15%. Тестирование скорости работы проводилось на исходном коде *SourcererCC*. Исходный код *SourcererCC* состоит из 103 файлов с исходным кодом и содержит 9153 строчки кода. Следует заметить что *SourcererCC* тратит больше времени на построение индекса из-за того что рассчитан на работу на большом количестве ядер и обработку данных пакетами. В рамках данного тестирования пакетный режим не был утилизирован. Сравнение скорости работы приведено в таблице 1. Сравнение количества найденных дубликатов приведено в таблице 2

Название средства	Построение индекса	Поиск дубликатов	Всего
potator.NaiveDetector	0 мс	615.22 мс	615.22 мс
potator.FilteringDetector, l=1	19.47 мс	98.98 мс	118.45 мс
potator.FilteringDetector, l=2	19.98 мс	122.69 мс	142.67 мс
potator.FilteringDetector, l=3	20.14 мс	105.57 мс	125.71 мс
potator.FilteringDetector, l=5	21.05 мс	105.00 мс	121.05 мс
SourcererCC	204.25 мс	95.36 мс	299.61 мс

Таблица 1: Сравнение скорости работы средств для поиска дубликатов

Название средства	Количество найденных клонов
potator без использования субтокенизации	45
potator с использованием субтокенизации	49
SourcererCC	45

Таблица 2: Сравнение количества найденных дубликатов

5.5 Описания алгоритмов используемых в программном средстве *potator*

5.5.1 Описание алгоритма субтокенизации

Цель субтокенизации состоит в том чтобы обработать названия переменных в соответствии с общепринятыми соглашениями об именовании. Например для имени *class FooBarBaz* субтокенами будут являться *foo, bar, baz*. Для разбиения токена на субтокены используются регулярные выражения и субтокены которые имеют длину меньше 3 объединяются с предыдущим субтокеном. Реализация данного алгоритма на языке `python` может быть найдена в приложении 1.

5.5.2 Описание алгоритма используемого в *NaiveDetector*

Данный детектор для поиска дубликатов кода для каждой функции и класса осуществляет полный перебор всех возможных кандидатов и вычисляет обобщенную меру Жаккара и отсеивает те кандидаты чья мера похожести меньше порогового значения. Реализация данного алгоритма на языке `python` может быть найдена в приложении 2.

5.5.3 Описание алгоритма используемого для работы с дельта инвертированными индексами

Поскольку фильтрация по адаптивным префиксам является обобщением фильтрации по префиксам, то и алгоритм построения и работы с обратным индексом будет являться обобщением алгоритма из пункта 3.4. Различие состоит в том что при $l > 1$:

$$tokensToBeIndexed = tokens[|tokens| - \lceil \theta * |tokens| \rceil + l]$$

Реализация данного алгоритма на языке `python` может быть найдена в приложении 3.

5.5.4 Описание алгоритма используемого в *FilteringDetector*

Алгоритм который используется в *FilteringDetector* для поиска клонов во многом повторяет алгоритм из секции 3.4. Различия состоят в том, что в *FilteringDetector*'е используется фильтрация не по префиксам, а по адаптивным префиксам. Значит при построении `candidateSimilarityMap` он добовляет в нее значения из l дельта инвертированных индексов. Реализация данного алгоритма на языке `python` может быть найдена в приложении 4.

6 Заключение

В рамках данной работы было проведено исследование существующих методов поиска дубликатов кода, а также реализовано программное средство способное находить дубликаты кода в наборах файлов с исходным кодом *potator*. Данное программное средство находится в открытом доступе и имеет открытый исходный код. Исходный код может быть найден по ссылке https://github.com/otzhora/potator вместе с инструкциями по установке. Также было проведено сравнение результатов поиска и скорости работы со стандартным средством поиска дубликатов SourcererCC. Реализованное программное средство *potator* пригодно для использовании в реальной практике разработки программного обеспечения.

Список литературы

[1] Baker, B. S. On finding duplication and near-duplication in large software systems / B. S. Baker // Proceedings of the Second Working Conference on Reverse Engineering. — WCRE '95. — USA: IEEE Computer Society, 1995. — P. 86.

[2] Comparison and evaluation of clone detection tools / S. Bellon, R. Koschke, G. Antoniol et al. // IEEE Transactions on Software Engineering. — 2007. — Vol. 33, no. 9. — P. 577–591.

[3] Do software developers understand open source licenses? / D. A. Almeida, G. C. Murphy, G. Wilson, M. Hoye // 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). — 2017. — P. 1–11.

[4] Effects of cloned code on software maintainability: A replicated developer study / D. Chatterji, J. C. Carver, N. A. Kraft, J. Harder // 2013 20th Working Conference on Reverse Engineering (WCRE). — 2013. — P. 112–121.

[5] Mallaiah, S. Structural similarity detection using structure of control statements / Sudhamani Mallaiah, Lalitha Rangarajan // Procedia Computer Science. — 2015. — 12. — Vol. 46. — P. 892–899.

[6] Markovtsev, V. Topic modeling of public repositories at scale using names in source code. — 2017.

[7] Mockus, A. Large-scale code reuse in open source software / Audris Mockus // Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development. — FLOSS '07. — USA: IEEE Computer Society, 2007. — P. 7. — https://doi.org/10.1109/FLOSS.2007.10.

[8] Mockus, A. Amassing and indexing a large sample of version control systems: Towards the census of public source code history / A. Mockus // 2009

6th IEEE International Working Conference on Mining Software Repositories. — 2009. — P. 11–20.

[9] Nishi, M. Scalable code clone detection and search based on adaptive prefix filtering / Manziba Nishi, Kostadin Damevski // Journal of Systems and Software. — 2017. — 11. — Vol. 137.

[10] On the naturalness of software / Abram Hindle, Earl T. Barr, Zhendong Su et al. // Proceedings of the 34th International Conference on Software Engineering. — ICSE '12. — Zurich, Switzerland: IEEE Press, 2012. — P. 837–847.

[11] On the use of clone detection for identifying crosscutting concern code / Magiel Bruntink, Arie Deursen, R. Engelen, Tom Tourwe // Software Engineering, IEEE Transactions on. — 2005. — 11. — Vol. 31. — P. 804– 818.

[12] Roy, C. A survey on software clone detection research / Chanchal Roy, James Cordy // School of Computing TR 2007-541. — 2007. — 01.

[13] Sheneamer, A. A detection framework for semantic code clones and obfuscated code / Abdullah Sheneamer, S. Roy, J. Kalita // Expert Syst. Appl. — 2018. — Vol. 97. — P. 405–420.

[14] Sourcerercc: Scaling code clone detection to big code / Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko et al. — 2015.

[15] Svajlenko, J. Cloneworks: A fast and flexible large-scale near-miss clone detection tool / Jeffrey Svajlenko, Chanchal Roy. — 2017. — 05. — P. 177–179.

[16] von Krogh, G. Knowledge reuse in open source software: An exploratory study of 15 open source projects / G. von Krogh, S. Spaeth, S. Haefliger // Proceedings of the 38th Annual Hawaii International Conference on System Sciences. — 2005. — P. 198b–198b.

[17] Wang, J. Can we beat the prefix filtering? an adaptive framework for similarity join and search / Jiannan Wang, Guoliang Li, Jianhua Feng // Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. — SIGMOD '12. — New York, NY, USA: Association for Computing Machinery, 2012. — P. 85–96. — https://doi.org/10.1145/2213836.2213847.

[18] Zipf, G. K. Selected studies of the principle of relative frequency in language. / G. K. Zipf // Harvard Univ. Press. — 1932.

A Приложение 1. Листнг алгоритма субтокенизации

```
# INPUT: token -> токен из которого будут извлечены субтокены
# OUTPUT: subtokens -> массив субтокенов
NAME_BREAKUP_RE = re.compile(r"[^a-zA-Z]+")

prev_p = [""]

def ret(name):
    result = []
    r = name.lower()
    if len(name) >= 3:
        result.append(r)
        if prev_p[0]:
            result.append(prev_p[0] + r)
            prev_p[0] = ""
    else:
        prev_p[0] = r
    return result

def subtokenize(token):
    result = []
    for part in NAME_BREAKUP_RE.split(token):
        prev = part[0]
        pos = 0
        for i in 1 : len(part):
            this = part[i]
            if prev.islower() and this.isupper():
                result.extend(ret(part[pos:i]))
```

```
            pos = i
        elif prev.isupper() and this.islower():
            if 0 < i - 1 - pos <= 3:
                result.extend(ret(part[pos:i - 1]))
                pos = i - 1
            elif i - 1 > pos:
                result.extend(ret(part[pos:i]))
                pos = i
                prev = this
        last = part[pos:]
        if last:
            result.extend(ret(last))
    return result
```

В Приложение 2. Листинг алгоритма используемого в *NaiveDetector*

```
# INPUT: directory -> директория с файлами с исходным кодом
#        threshold -> пороговое значение поиска
# OUTPUT: DetectionResult ->
class NaiveDetector:
    def detectr(directory, threshold):
        files, files_data, entities \
            = EntitiesExtractor.extract_data_from_directory(directory)

        clones = []
        for entity in entities:
            for candidate in entities:
                if candidate == entity:
```

```
                    continue

                similarity = jaccard(entity.tokens, candidate.tokens)
                if similarity > threshold:
                    clones.append(CloneData(entity, \
                        candidate, similarity))

        return DetectionResult(clones, NaiveDetector)
```

C Приложение 3. Листинг алгоритма для работы с дельта инвертированными индексами

```
# INPUT: entities -> массив токенов для которого нужно получить
#                    границы
#        l_depth -> длина l-префикса
#        threshold -> пороговое значение
# OUTPUT: границы в которых необходими индексировать данный
#         набор токенов
def get_tokens_bounds(tokens, l_depth, threshold):
    if l_depth == 1:
        left_bound = 0
    else:
        left_bound = len(tokens) - ceil(threshold * len(tokens)) \
            + l_depth - 1

    right_bound = len(tokens) - ceil(thr * len(tokens)) + l_depth
    return left_bound, right_bound


class Index:
```

```
    def __init__(self, l_depth, tokens2entites):
        self.l_depth = l_depth
        self.tokens2entities = tokens2entities

def build_l_index(entities, l_depth, thr):
    tokens2entities = {}
    for entity in entities:
        tokens = entity.bag_of_tokens

        left_bound, right_bound = \
            get_tokens_bounds(tokens, l_depth, thr)
        tokensToBeIndexed = tokens[left_bound: right_bound]
        for token in tokensToBeIndexed:
            tokens2entities[token].append(entity)

    return Index(l_depth, tokens2entities)


class Indexer:
    def __init__(entities, max_l_depth, threshold):
        self._indexes = []

        for l_depth in range(1, self.max_l_depth + 1):
            self._indexes.append(build_l_index(entities, \
                                               l_depth, threshold))

    def get_entities_for_token(self, l, token):
        return self._indexes[l - 1].tokens2entities[token]
```

D Приложение 4. Листинг алгоритма используемого в *FilteringDetector*

```
class FilteringDetector:
    def detect(directory, threshold, l_depth):
        files, files_data, entities = EntitiesExtractor \
            .extract_data_from_directory(directory)

        # GTC - Global Token Counts, количество раз которое токен
        # встречается в файлах с исходным кодом
        gtc = sort_tokens_gtc(entities)
        indexer = Indexer(entities, l_depth, threshold)

        clones = []
        for entity in entities:
            candidates = set()

            for l_depth in range(1, l_depth + 1):
                left_bound, right_bound \
                    = get_tokens_bounds(tokens, l_depth, threshold)
                tokensToBeQuered = tokens[left_bound: right_bound]
                for token in tokensToBeQuered:
                    candidate_entities = \
                        indexer.get_entities_for_token(token, \
                                                       lang, l_depth))
                    for entity in candidate_entities:
                        candidates += {entity: l_depth}

            ct = |entity|*theta
            candidateSimilarityMap = verifyCandidates(token, \
```

```
                                                    candidates, \
                                                    ct)

        return DetectionResult(clones, FilteringDetector)
```