**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

Кафедра вычислительной математики

Касияник Алексей Леонидович

**УСКОРЕНИЕ СХОДИМОСТИ ПРОЦЕССОВ УСТАНОВЛЕНИЯ. ПЕРЕОБУСЛОВЛИВАНИЕ И ПОДАВЛЕНИЕ ОШИБКИ**

Дипломная работа  
студента 5 курса 5 группы

|  |
| --- |
| “Допустить к защите“  **Руководитель работы**  *Фалейчик Борис Викторович*  доцент кафедры выч. мат.,  кандидат физ.-мат. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  “\_\_\_” \_\_\_\_\_\_\_\_\_\_\_ 2015 г |

Минск 2015

**Белорусский государственный университет**

**Факультет прикладной математики и информатики**

Кафедра вычислительной математики

“**Утверждаю**”

Заведующий кафедрой

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_П.А. Мандрик

“\_\_\_”  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  2015 г.

**ЗАДАНИЕ**

ПО ПОДГОТОВКЕ КУРСОВОЙ РАБОТЫ

Студенту 5 курса Касиянику Алексею Леонидовичу (5 группа)

1. Тема УСКОРЕНИЕ СХОДИМОСТИ ПРОЦЕССОВ УСТАНОВЛЕНИЯ. ПЕРЕОБУСЛАВЛИВАНИЕ И ПОДАВЛЕНИЕ КОМПАНЕНТ

2. Срок сдачи студентом законченной работы \_\_ июня 2015 г.

3. **Исходные данные к работе**

* Численные методы
* Технические требования к электронным версиям отчетных документов.

Библиографические описания источников, рекомендуемых студентам к ознакомлению при выполнении работы (для изучения предметной части задания, как правило, достаточно ознакомиться с любой из перечисленных в начале списка книг):

1. *Pawlik, M.,* RTED: A Robust Algorithm for the Tree Edit Distance / M. Pawlik, N. Augsten // PVLDB 5(4):334–345, 2011.
2. *Moussiades, L.,* Pdetect: A clustering approach for detecting plagiarism in source code datasets / L. Moussiades, A. Vakali // The computer journal - 48(6), 2005.
3. *Lancaster, T.*, A comparison of source code plagiarism detection engines. / T. Lancaster, F. Culwin // Computer Science Education - 14:2:101 – 117, 2004.
4. *Joy, M.S.,* Plagiarism in Programming Assignments(1999). - Joy, M.S., Luck, M. IEEE Transactions on Education, 42(2). pp. 129-133. ISSN 0018-9359
5. *Левенштейн В.И.,* Двоичные коды с исправлением выпадений, вставок и замещений символов: Доклады Академий наук СССР / В.И. Левенштейн – М.: Проспект, 2009. – 861 с.

4. **Перечень вопросов подлежащих разработке или краткое содержание работы**

* Рассмотреть постановку задачи нахождения плагиата в исходных кодах программ.
* Рассмотреть основные алгоритмы нахождения плагиата.
* Спроектировать систему нахождения плагиата в исходных кодах программ.
* Реализовать алгоритмы нахождения плагиата в исходных кодах программ

5. **Дата выдачи задания** \_\_ сентября 2014 г.

6. **Календарный график** работы на весь период (с указанием этапов работы и   
сроков их выполнения)

* **сентябрь-октябрь** – изучение постановки задачи, исследование предметной области;
* **ноябрь-декабрь** – проектирование системы, алгоритмов;
* **январь-март** – реализация системы;
* **апрель-май** – отчет о проделанной работе;
* **июнь:**  защита.

**Руководитель** \_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Б.В. Фалейчик / …. февраля 2015 г.

**Задание принял к исполнению** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ .… сентября 2014 г.

(подпись студента)

АННОТАЦИЯ

Касияник А.Л. Ускорение сходимости процессов установления. Переобусловливание и подавление компонент: Дипломная работа / Минск: БГУ, 2015.– 22 с.

В дипломной работе исследуются различные способы ускорения сходимости в процессе решения жестких систем дифференциальных уравнений с помощью методов, основанных на процессах установления. В частности, рассмотрено использование переобусловливания и подавления компонент. На примере тестовой задачи проведен численный эксперимент для линейного случая.

АНАТАЦЫЯ

Касіянік А.Л. Паскарэнне збежнасці працэсаў усталявання. Пераабумоўленне і падаўленне кампанент: Дыпломная / Мінск: БДУ, 2015. – 41 с.

У дыпломная працы даследуюцца розныя падыходы паскарэння збежнасці ў працэсе рашэння жорсткіх сістэм дыферэнцыяльных ураўненняў з дапамогай метадаў, заснаваных на працэсах усталявання. У прыватнасці, разгледжана выкарыстанне пераабумоўлення і падаўлення кампанент. На прыкладзе тэставай задачы праведзены лікавы эксперымент для лінейнага выпадку.

ANNOTATION

Kasiyanik A. Convergence acceleration of steadying processes. The preconditioning and component suppression: Diploma project /Minsk: BSU, 2015. – 41 p.

In the present work considers the various approaches of convergence acceleration in the process of solving stiff systems of differential equations by methods based on iterative steadying processes. In particular considers using of the preconditioning and component suppression. There is numerical experiment for test problem in this work for linear system.

# РЕФЕРАТ

Курсовая работа, 22 с., 7 рис., 1 таблица, 9 источников.

***Ключевые слова:*** ПРИНЦИП УСТАНОВЛЕНИЯ, ЖЕСТКИЕ ЗАДАЧИ, МЕТОДЫ РУНГЕ-КУТТЫ, ПЕРЕОБУСЛАВЛИВАНИЕ, СПЕКТРАЛЬНЫЕ СВОЙСТВА, УСКОРЕНИЕ СХОДИМОСТИ, ПОДАВЛЕНИЕ КОМПОНЕНТ.

***Объектами исследования*** является численные методы решения жестких дифференциальных задач.

***Цель исследования:*** разработка ускоренных вычислительных алгоритмов для решения жестких дифференциальных задач на основе принципа установления.

***Методы исследования:*** методы численного анализа.

***Результаты:*** изучен алгоритм решения жестких дифференциальных задач, в основе которого лежит принцип установления; предложены способы ускорения сходимости в процессе решения; проведен численный эксперимент.

***Область применения:*** решение задач математической физики.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 8](#_Toc418430868)

[1 ПОСТАНОВКА ЗАДАЧИ 9](#_Toc418430869)

[2 ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ 10](#_Toc418430870)

[2.1 Компиляторы C, C++, Pascal 10](#_Toc418430871)

[2.2 Специфика поиска плагиата в программах 11](#_Toc418430872)

[2.2.1 Классификация методов поиска плагиата 11](#_Toc418430873)

[2.2.2 Атрибутные методы 11](#_Toc418430874)

[2.2.3 Структурные методы 11](#_Toc418430875)

[2.2.4 Нейросетевые методы 12](#_Toc418430876)

[2.3 Обзор существующих алгоритмов поиска плагиата 12](#_Toc418430877)

[2.3.1 Алгоритм Хескела 12](#_Toc418430878)

[2.3.2 Выравнивание строк 13](#_Toc418430879)

[2.3.3 Жадное строковое замощение 14](#_Toc418430880)

[2.3.4 Метод отпечатков 15](#_Toc418430881)

[2.4 Модели представления программ 16](#_Toc418430882)

[2.4.1 Представление программ в виде элемента n-мерного пространства 16](#_Toc418430883)

[2.4.2 Исходный код 16](#_Toc418430884)

[2.4.3 Токенизация 16](#_Toc418430885)

[2.5 Алгоритм вычисления расстояния между деревьями 17](#_Toc418430886)

[2.5.1 Рекурсивное решение 17](#_Toc418430887)

[2.5.2 Существующие алгоритмы 18](#_Toc418430888)

[2.5.3 Стратегии 19](#_Toc418430889)

[2.5.4 Общий алгоритм 20](#_Toc418430890)

[2.6 Средства распознавания кода 20](#_Toc418430891)

[2.7 Форма Бэкуса — Наура 22](#_Toc418430892)

[2.8 Самоорганизующаяся карта Кохонена 23](#_Toc418430893)

[2.8.1 Структура сети 23](#_Toc418430894)

[2.8.2 Работа сети 23](#_Toc418430895)

[2.8.3 Алгоритм 24](#_Toc418430896)

[2.9 Планирование задач в Spring 26](#_Toc418430897)

[3 ОБЗОР СУЩЕСТВУЮЩИХ СИСТЕМ ОБНАРУЖЕНИЯ ПЛАГИАТА 28](#_Toc418430898)

[3.1 Детектор JPlag 28](#_Toc418430899)

[3.2 Детектор MOSS 28](#_Toc418430900)

[3.3 Детектор SID 28](#_Toc418430901)

[3.4 Сравнительная характеристика детекторов 29](#_Toc418430902)

[4 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРИЛОЖЕНИЯ 30](#_Toc418430903)

[4.1 Структура базы данных 30](#_Toc418430904)

[4.2 Use-case диаграмма 31](#_Toc418430905)

[4.3 Порядок обработки программ 31](#_Toc418430906)

[4.4 Алгоритм тестирования программы 31](#_Toc418430907)

[4.5 Проверка на наличие плагиата 32](#_Toc418430908)

[4.5.1 Реализация атрибутного алгоритма 33](#_Toc418430909)

[4.5.2 Реализация структурного алгоритма RTED 33](#_Toc418430910)

[4.5.3 Реализация нейронной сети 34](#_Toc418430911)

[4.6 Получение древовидной структуры программы 35](#_Toc418430912)

[4.7 Организация тестирования по расписанию 35](#_Toc418430913)

[4.8 Визуализация результатов 36](#_Toc418430914)

[4.9 Описание основных классов системы 37](#_Toc418430915)

[4.9.1 Класс ProgramFilesUtil 37](#_Toc418430916)

[4.9.2 Класс TreeNode 38](#_Toc418430917)

[4.9.3 Интерфейс NodeDistance 38](#_Toc418430918)

[4.9.4 Класс TreeParser 38](#_Toc418430919)

[4.9.5 Класс TreeEditDistance 39](#_Toc418430920)

[4.9.6 Класс RTED 39](#_Toc418430921)

[4.9.7 Класс NeuralNode 39](#_Toc418430922)

[4.9.8 Класс NeuralAttributesExtractor 40](#_Toc418430923)

[ЗАКЛЮЧЕНИЕ 41](#_Toc418430924)

[ЛИТЕРАТУРА 42](#_Toc418430925)

ВВЕДЕНИЕ

Жесткие задачи исследуются примерно со второй половины 20 века. Однако и сейчас сформулировать точное определение жесткости проблематично. Наиболее прагматическая точка зрения вместе с тем была и исторически наиболее ранней (Кертисс и Хиршфельдер, 1952 год): *жесткие уравнения — это уравнения, для которых определенные неявные методы дают лучший результат, обычно несравненно более хороший, чем явные методы*. При этом определенную роль играют собственные значения матрицы Якоби, но важны и такие параметры, как размерность системы, гладкость решения или интервал интегрирования.

Более полным является определение данное Ламбертом: *если численный метод с ограниченной областью абсолютной устойчивости, примененный к системе с произвольными начальными условиями вынужден использовать на некотором интервале интегрирования величину шага, которая чрезмерно мала по отношению к гладкости точного решения на этом интервале, тогда говорят, что система является жесткой на этом интервале.*

Как известно, наиболее трудоёмким этапом численного интегрирования жёсткой системы (не)линейных обыкновенных дифференциальных уравнений (ОДУ) размерности n неявным методом является решение на каждом шаге системы (не)линейных уравнений, размерность которой пропорциональна n. В таком случае использование методов ньютоновского типа практически невозможно, а традиционные методы типа простой итерации либо не сходятся, либо сходятся очень медленно. В данной работе рассматриваются способы ускорения методов, основанных на процессах установления, которые применимы в указаной выше ситуации.

В работе исследованы два эффективных способа ускорения сходимости решения в процессе решения методами установления: переобусловливание и подавление компонент. Переобусловливание является классическим способом уменьшения «спектрального числа обусловленности»[1] матрицы решаемой задачи, которое существенным образом определяет свойства сходимости итерационного процесса. С помощью операции переобусловливания производится уменьшение числа обусловленности, что положительно влияет на скорость сходимости процесса.

Также было замечено, что компоненты ошибки, которые соответствуют малым собственным значениям, сходятся медленно. Поэтому предположительно подавление ошибки медленно сходящися компонент может дать существенную прибавку в скорости сходимости итерационного процесса. В результате исследования данного феномена был предложен прием ускорения итерационного процесса, который по аналогу со схожими алгоритмами из известных источников[][] именуется «подавлением компонент».

Исследованию описанных проблем, разработке вычислительного алгоритмов, которые бы основывались на идее установления и при этом превосходили в скорости известные алгоритмы и посвящается данная работа.

1. ЖЕСТКИЕ ЗАДАЧИ

Тема численного решения однородных дифференциальных уравнений не нова, и, возможно, несколько удивителен тот факт, что методы, разработанные еще в начале 20 века, до сих пор являются основой наиболее эффективных и распространенных подходов при решении ОДУ. За прошедшее время были достигнуты значительные продвижения в надежности и эффективности этих методов, и большинство существующих типичных научных задач могут быть решены достаточно легко и быстро. Тем не менее есть некоторый класс задач, с которыми классические методы справиться не могут. Такие задачи, называемые «жесткими», слишком важны, чтобы их игнорировать, и слишком трудны, чтобы их решить. Они слишком важны, чтобы их игнорировать, так как они возникают при решении важных физических задач. Они слишком затратные при решении, так как из-за присущей им большой размерности и сложности, классические методы становятся слабо применимы даже несмотря на многократное увеличение мощности современной вычислительной техники. Классические методы решения требуют так много шагов, что ошибки округления могут сделать полученное решение далеким от приемлемого[2]. В этом разделе мы и рассмотрим понятие «жесткости», а также те ключевые проблемы, которые возникают при их решении.

* 1. Явление жесткости

Задачи, называемые жесткими, весьма разнообразны, и дать математически строгое определение жесткости непросто. Поэтому в литературе можно встретить различные определения жесткости, отличающиеся степенью строгости. Сущность же явления жесткости состоит в том, что решение, которое необходимо вычислить, меняется медленно, однако в любой его окрестности существуют быстро затухающие возмущения. Характерное время затухания их называют пограничным слоем. Наличие таких возмущений затрудняет получение медленно меняющегося решения численным способом. При этом жесткими могут быть как скалярные дифференциальные уравнения, так и, что встречается особенно часто, системы обыкновенных дифференциальных уравнений.

Система обыкновенных дифференциальных уравнений вида

с постоянной (n x n) матрицей А называется жесткой, если

1. (т.е. задача устойчива);
2. Отношение ;
3. Промежуток интегрирования велик по сравнению с длиной погранслоя.

Число *S* иногда называют коэффициентом жесткости системы.

Поскольку система нелинейных обыкновенных дифференциальных уравнений вида

может быть в окрестности некоторого известного решения заменена линейной системой , где

– матрица Якоби системы,

то понятие жесткости для нелинейных систем может быть определено аналогично. Заметим, однако, что за пределами класса систем линейных обыкновенных дифференциальных уравнений с постоянной матрицей полагаться на спектр как на источник надежной информации о распространении погрешности уже нельзя.

* 1. Явление жесткости

Приведем вначале

некоторые примеры.

1. ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ
   1. Компиляторы C, C++, Pascal

В данном приложении было использовано 3 компилятора: Digital Mars(C++), Tiny C Compiler, Free Pascal Compiler.

Digital Mars

Digital Mars C/C++ — это компилятор C/C++ для ОС Win32, Win16, DOS32, DOS. Данный компилятор обладает высокой скоростью сборки и компиляции, мощной технологией оптимизации, HTML документацию, дизассемблер, множество библиотек, компилятор ресурсов, системой обновления через Интернет, графическим интерфейсом и многим другим.

Основные черты:

* поддержка C/C++
* высокая скорость сборки и компиляции
* мощная технология оптимизации с применением распределения регистров и планированием инструкций
* поддержка SGI’s Standard Template Library 3.3
* превосходная поддержка инструкций и расширенных инструкций Intel MMX, SSE, SSE2, AMD

Tiny C Compiler

Tiny C Compiler (англ. Tiny C Compiler — крошечный компилятор Си), или TinyCC, или TCC — компилятор C для платформы x86. Работает в ОС Linux и Microsoft Windows.

Особенности:

* От других распространённых компиляторов TCC отличается, прежде всего, тем, что может исполнять скомпилированную им программу, то есть выполнять функцию интерпретатора. Данное свойство позволяет использовать язык Си в качестве скриптового языка.
* Компактность. Исполняемый файл для процессоров x86, включающий препроцессор, компилятор, ассемблер и компоновщик, составляет всего около 100 КБ.
* Высокая скорость компиляции. Например, TCC примерно в 9 раз быстрее GCC.
* Возможность формировать код с контролем границ массивов, который можно беспрепятственно использовать вместе с обычным кодом.
* Возможность напрямую использовать любую динамическую библиотеку.
* Оптимизация кода ограничена вычислением константных выражений на этапе компиляции, заменой операций умножения и деления операциями сдвига, где это возможно, а также некоторыми другими действиями.

Free Pascal Compiler

Free Pascal (полное название Free Pascal Compiler, часто используется сокращение FPC) — свободно распространяемый компилятор языка программирования Pascal.

Особенности:

* Поддержка перегрузки арифметических операторов (+, -, \*, \*\*, /, div, mod), операторов сравнения (<, >, =, >=, <=) и оператора присваивания :=.
* Поддержка операторов присваивания с выполнением арифметической операции в стиле Си (+=, -=, \*=, /=).
* Наличие собственной системы сборки (fpcmake) и генератора документации (fpcdoc).
* Встроенный ассемблер по умолчанию использует синтаксис AT&T, синтаксис Intel включается отдельной директивой.
  1. Специфика поиска плагиата в программах

Обычно с программой сопоставляют так называемый исходный код и исполняемый код (а также объектный код как промежуточный этап).

Исходный код программы анализировать легче, поскольку в ней сохраняется больше характеристик свойственных конкретному автору (в основном это касается стилистических особенностей автора, которые при компиляции в основном утрачиваются). Тем не менее, и по исполняемому коду тоже можно искать, в нем хранится много индивидуальной информации (используемые алгоритмы, специфические ошибки, способ организации данных).

* + 1. Классификация методов поиска плагиата

Выделяют 2 основных подхода к оценке близости программ (и соответственно разработке алгоритмов поиска плагиата): атрибутный и структурный. Существуют также методы, включающих оба подхода.

* + 1. Атрибутные методы

Исторически первыми появились атрибутные методы. Смысл их заключался в численном выражении каких-то признаков (атрибутов) программы и сравнении полученных чисел для разных программ. Программы с близкими численными характеристиками атрибутов потенциально похожи. В простейшем случае можно использовать размер программы или количество переменных.

Можно комбинировать несколько признаков, чтобы программа была представлена не одним числом, а некоторым набором. Две программы могут считаться похожими, если соответствующие числа из их наборов совпадают или близки. Для описания программ были предложены такие атрибуты, как количество операторов или операндов, цикломатическая сложность и другие. Таким образом, оценка близости программ сводится к сравнению чисел или векторов, которые получаются путем несложного анализа непосредственно исходного кода. Основным недостатком атрибутных методов является то, что несвязанные между собой параметры программы плохо описывают ее в целом, и, при таком подходе, разные программы получают близкие характеристики. Атрибутные методы активно развивались в 80-х годах XX века, но постепенно отошли на второй план.

* + 1. Структурные методы

Другой, более современный и перспективный, подход состоит в сравнении программ с учетом их структуры. Эта процедура более сложная, чем сравнение численных выражений отдельных свойств программы. Структурные методы исследуют свойства программы не изолированно, а в контексте, устанавливает взаимосвязь различных характеристик, их совместное поведение.

Чтобы отбросить лишнюю информацию и выделить нужные зависимости, программу предварительно переводят в более компактное представление.

Недостатком структурных методов является их сложность и вычислительная трудоемкость. Классическим примером структурного подхода является построение дерева программы с последующим сравнением деревьев для разных программ. Трудоемкость такого метода кубическая, что не дает возможности эффективно его применять для большого числа достаточно длинных программ. Кроме того, структурные методы обычно опираются на синтаксис одного языка программирования. Адаптация метода для другого языка требует значительных усилий. Сложность реализации алгоритмов, сравнивающих структуру программы, является платой за их точность.

* + 1. Нейросетевые методы

Поиск плагиата можно свести к задаче классификации, когда набор программ требуется разбить на несколько классов, в каждом из которых будут находиться только похожие программы. Как известно, нейронные сети – это один из лучших инструментов для решения задачи классификации.

Нейронную сеть можно представить как черный ящик, на вход которому подается известная информация, а на выходе выдается информация, которую хотелось бы узнать. Каждый нейрон нейросети, как правило, связан со всеми нейронами предыдущего слоя обработки данных. Типичный формальный нейрон производит простейшую операцию - взвешивает значения своих входов со своими же локально хранимыми весами и производит над их суммой нелинейное преобразование. Из-за нелинейности функции активации всю нейросеть нельзя свести к одному нейрону и возможности нейросети существенно выше возможностей отдельных нейронов. Состояние сети характеризуется набором весов, которые меняются в процессе поступления входной информации. Данный процесс называется обучением сети. Принцип обучения нейросетей основан на минимизации эмпирической ошибки. Функция ошибки, оценивающая данную конфигурацию сети, задается извне - в зависимости от того, какую цель преследует обучение. Но далее сеть начинает постепенно модифицировать свою конфигурацию - состояние всех своих весов - таким образом, чтобы минимизи­ровать эту ошибку. В итоге, в процессе обучения сеть все лучше справляется с возложенной на нее задачей. Характерной особенностью нейросетей является их способность к обобщению, позволяющая обучать сеть на ничтожной доле всех возможных ситуаций, с которыми ей, может быть, придется столкнуться в процессе функционирования.

Различие в подходах при решении задачи поиска плагиата заключается, в основном, в использовании той или иной разновидности сети. Все нейросети можно разделить на два класса по методу обучения: обучение с учителем и обучение без учителя (самообучение).

При обучении с учителем реальные выходы сети сравниваются с эталонными, и процесс обучения сводится к задаче максимального приближения выходов сети к эталону.

Другой разновидностью нейронных сетей являются самоорганизующиеся карты. В данном виде нейросетей используется метод обучения без учителя. В этом случае сети предлагается самой найти скрытые закономерности в массиве данных и результат обучения зависит только от структуры входных данных. То есть классы не известны априори и определяются согласно схожести входной информации при обучении. После обучения на вход сети можно подавать новые примеры, и они будут отнесены к одному из классов или ни к одному. В последнем случае можно говорить, что появился новый класс. Один из видов самоорганизующихся карт - сети Кохонена.

* 1. Обзор существующих алгоритмов поиска плагиата
     1. Алгоритм Хескела

Пусть у нас есть две программы, представим их в виде строк токенов a и b соответственно. Одним из критериев сходства строк считается длина их наибольшей общей подпоследовательности. Мы всегда можем найти такой элемент строки ai, что НОП строк a0 = a|a|a|a|−1 . . . aia1 . . . ai−1 и b будет значительно меньше (максимум в два раза), чем НОП(a, b) (если НОП(a, b) > 1). Чтобы избежать этого явления можно воспользоваться алгоритмом сравнения строк Хескела. Он требует нескольких проходов, но работает за линейное время. Разобьем строки a и b на k-граммы (подстроки длины k). Найдем те k-граммы, которые встречаются в a и b только по одному разу. Для каждой такой пары проверим совпадают ли элементы строк, непосредственно лежащие над ними; если это так, то проведем ту же проверку и для них и так далее, пока несовпадение не будет найдено. Аналогично для строк, лежащих ниже соответствующих k-граммов. Получаем набор общих непересекающиеся подстрок a и b. Их общая длина может служить мерой схожести программ соответствующих a и b.

Достоинства:

1. Линейная трудоемкость алгоритма

Недостатки:

1. Возможность совпадения токенизированного представления программ, но отсутствия совпадения в исходных кодах программ.
2. Небольшое количество уникальных k-граммов в больших программах, соответственно многие совпадения, не содержащие в себе таких k-граммов будут проигнорированы.
3. Вставка в найденный блок или изменение на семантически эквивалентный оператора во многих случаях будет приводить к игнорированию той части блока, в которой не содержится уникальный k-грамм.
   * 1. Выравнивание строк

Пусть у нас есть две программы, представим их в виде строк токенов s и t соответственно (возможно различной длины). Теперь мы можем воспользоваться методом локального выравнивания строк, разработанным для определения схожести строк ДНК. Выравнивание двух строк получается с помощью вставки в них пробелов таким образом, чтобы их длины стали одинаковыми. Заметим, что существует большое количество различных выравниваний двух строк. Например, рассмотрим два выравнивания строк “masters” и “stars”:

masters masters

sta rs stars

Будем называть строки, полученные после выравнивания s и t, соответственно s0 и t0. Рассмотрим si и ti: стоимость их совпадения m, стоимость пропуска g, стоимость несовпадения d, где m, d и g — произвольные числа. Цена выравнивания — это сумма индивидуальных стоимостей всех пар s’i и t’i , наибольшее значение этой целевой функции для всех i и j (i ≤ j ≤ |s0| = |t0|) на строках s0[i..j] и t0[i..j] — величина выравнивания. Пусть m = 1, d = −1 и g = −2, тогда “rs/rs” и "sters/strars— блоки с наибольшей целевой функцией в первом и втором выравнивании соответственно. Цена выравниваний — 2 и 3. Она родственна редакционному расстоянию и используется для измерения расстояния между почти идентичными объектами, также успешно применяется для определения неточного соответствия ДНК в вычислительной биологии.

Оптимальное выравнивание между двумя строками — максимальное значение целевой функции среди всех выравниваний. Это значение может быть вычислено с помощью динамического программирования. Пусть даны две строки s и t, определим D(i, j) как оптимальное выравнивание между строками s[1..i] и t[1..j]. Мы ищем max1≤i≤|s|,1≤j≤|t| D(i, j).

Определим:

Следующее рекуррентное соотношение позволяет нам получить требуемое:

Граничные условия задаются соотношениями D(0, i) = 0 и D(j,0) = 0. Остальные элементы матрицы D могут быть вычислены при проходе слева направо и сверху вниз. Это возможно, потому что D(i, j) зависит только от D(i − 1, j − 1), D(i − 1, j) и D(i, j − 1). Время вычисления оптимального выравнивания составляет O(|s||t|); затраты памяти — O(max(|s|, |t|), потому что для вычисления нужны только две строки.

Применение этого алгоритма в нашем случае выглядит примерно так: получаем токенизированное представление p1 и p2 двух программ, делим вторую строку p2 на подстроки (секции), каждая из которых представляет модуль исходной программы. Для каждой секции и p1 получаем значение оптимального локального выравнивания. Это позволяет алгоритму корректно обрабатывать перестановки модулей исходной программы.

При построении оптимального локального выравнивания существуют некоторые тонкости. Игнорируются разделительные символы, содержание комментариев (то есть комментарию сопоставляется один код-токен), каждой переменной и функции вместо имени динамически ставится в соответствие некоторый код, каждому ключевому слову языка и специальному символу присваивается заранее определенный код, всем операндам также присваиваются коды. При построении оптимального локального выравнивания p1 и подстроки p2, несовпадениям и совпадениям разных токенов могут ставиться в соответствие разные цены. Например, при совпадении двух токенов-идентификаторов, мы имеем 2, другие совпадение 1; при несовпадении двух токенов-идентификаторов имеем 0, другие несовпадение -2.

* + 1. Жадное строковое замощение

Рассмотрим эвристический алгоритм получения жадного строкового замощения (The Greedy String Tiling). Он получает на вход две строки символов над определенным алфавитом (у нас это множество допустимых токенов), а на выходе дает набор их общих непересекающихся подстрок близкий к оптимальному. Подстроку, входящую в этот набор, мы будем называть тайлом (tile).

Пусть P и T — токенизированные представления сравниваемых программ.

**Определение.** *MinimumMatchLength* — минимальная длина наибольшего общего префикса строк Pp и Tt , при которых он учитывается алгоритмом.

**Определение.** Длина самого большого из пока найденных на текущей итерации алгоритма общих префиксов строк Pp и Tt обозначается *maxmatch*.

**Определение.** Набор тайлов мы будем обозначать как *tiles*.

**Определение**. Множество, содержащее кандидатов на попадание в набор тайлов, обозначается *matсhes*.

Алгоритм можно разделить на две фазы:

1. Ищутся наибольшие общие подстроки P и T, состоящие только из непомеченных элементов (вначале алгоритма все элементы непомечены). Для этого используются три вложенных цикла: первые пробегает по всем возможным Pp, второй по всем Tt, а третий находит наибольший общий префикс Pp и Tt. Далее существует три варианта в зависимости от соотношения величин maxmatch и найденного префикса:
2. если maxmatch меньше, то мы удаляем из списка общих подстрок matches все до этого добавленные и помещаем туда найденный префикс.
3. если maxmatch больше, то ничего не меняем.
4. если они равны, то добавляем наибольший общий префикс Pp и Tt к списку matches.
5. Проходим по списку, если текущий элемент списка — подстрока, не содержащая помеченных элементов, то помещаем ее в выходной набор tiles (теперь эту подстроку называют tile — отсюда и название алгоритма), помечаем все элементы рассматриваемой строки, входящие в P и T. Если длины строк в списке (maxmatch) больше чем MinimumMatchLength, то переходим к первой фазе.

Этот алгоритм использует две эвристики:

1. Вначале мы находим наибольшие общие подстроки, состоящие из непомеченных символов, потому что считаем, что длинные совпадения для нашей задачи наиболее содержательны. Понятно, что вследствие этой эвристики мы иногда можем не найти более оптимальное (в плане количества покрывающих подстрок) общее частичное покрытие, но это было бы нам абсолютно безразлично без следующей эвристики.
2. Если в алгоритме мы будем позволять учитывать слишком маленькие наибольшие общие префиксы, то случайные совпадения небольшого количества токенов будут влиять на определение схожести программ, чтобы избежать этого вводится константа MinimumMatchLength.
   * 1. Метод отпечатков

В этом алгоритме мы представляем токенизированную программу в виде набора отпечатков (меток, ﬁngerprints), так чтобы эти наборы для похожих программ пересекались. Этот метод позволяет организовать эффективную проверку по базе данных. Метод отпечатков можно представить в виде четырех нижеследующих шагов:

1. Последовательно хэшируем подстроки токенизированной программы P длины k (фиксированный параметр).
2. Выделяем некоторое подмножество их хэш-значений, хорошо характеризующее P. Проделываем те же шаги для токенизированных программ T1, T2 . . . Tn и помещаем их выбранные хэш-значения в хэш-таблицу.
3. С помощью хэш-таблицы (базы) получаем набор участков строки P, подозрительных на плагиат.
4. Анализируем полученные на предыдущем шаге данные и делаем выводы.

Очевидно, что содержательная часть этого алгоритмы выбор k и шаг 2. Что же задает число k? Оно назначает длину наименьшей подстроки, с которой может работать данный алгоритм. Если в двух программах есть общая подстрока длиной, например, два символа, то необязательно, что мы нашли плагиат – скорее всего это совпадение просто случайность и обусловленно малой мощностью нашего алфавита. То есть значение k может способствовать игнорированию “шума”, но при больших значениях k, мы можем проигнорировать настоящий случай плагиата, поэтому это число нужно выбирать с осторожностью.

Пусть наша хэш-функция – это h, а h1, h2 . . . h|P|−k+1 – последовательность значений хэш-функций, полученных на шаге 1. Рассмотрим некоторые возможные реализации шага 2:

* Наивный подход заключается в выборе каждого i-ого из n хэш-значений, но он неустойчив к модификациям кода. (Если мы добавим вначало файла один лишний символ, то получим совершенно другое подмножество хэш-значений после выполнения алгоритма.)
* Мы можем назначать метками p минимальных хеш-значений, их количество для всех документов будет постоянно. С помощью этого метода нельзя найти частичные копии, но он хорошо работает на файлах примерно одного размера, находит похожие файлы, может применяться для классификации документов.
* Манбер предложил выбирать в качестве меток только те хеш-значения, для которых hi ≡ 0 mod p, так останется только n/p меток (объем идентификационного набора для разных файлов будет отличаться, сами метки будут зависеть от содержимого файла). Однако, в этом случае расстояние между последовательно выбранными хеш-значениями не ограничено и может быть велико. В этом случае совпадения, оказавшиеся между метками, не будут учтены.

Метод просеивания (winnowing) не имеет этого недостатка. Алгоритм гарантирует, что если в двух файлах есть хотя бы одна достаточно длинная общая подстрока, то как минимум одна метка в их наборах совпадет.

* 1. Модели представления программ
     1. Представление программ в виде элемента n-мерного пространства

Ранние системы по обнаружению плагиата представляли программу, как точку в n-мерном пространстве натуральных чисел с нулем, i-я координата которой – количественная характеристика какого-либо свойства/атрибута программы. Например, средняя длина строки кода, количество объявленных и используемых переменных, средняя длина имен переменных количество операторов ветвления и т.д. Если точки двух программ лежат рядом, то одна из них предполагается плагиатом другой.

Обычно такие системы называют атрибутно-подсчетными (attribute-counting systems). Чтобы определить, насколько близко лежат точки двух программ, обычно подсчитывается метрика (например, евклидово расстояние) или определяется их корреляция.

Вычисляя *i*-ю характеристику на протяжении всей программы, мы получаем ее усредненное значение, поэтому теряем слишком много информации о структуре программы. У таких систем 2 недостатка: они выдают очень много ложных совпадений и при крайне поверхностном изменении кода не находят плагиат. Если мы используем только какую-то часть программы в нашей, то найти плагиат практически невозможно. В некоторых системах используются и более сложные характеристики, основанные на использовании графа потока управления (control-flow graph), но это более трудоемкие алгоритмы и, к тому же, небольшие изменения структуры программы могут вызвать сильные количественные изменения таких характеристик.

* + 1. Исходный код

Некоторые системы обнаружения плагиата рассматривают исходный код «как есть». Так поступают детекторы плагиата, которые работают с кодом так же, как и с обычными текстами. Но они крайне неэффективны, так как переименование функций и переменных или несущественные изменения в коде являются серьезными препятствиями для их работы. Иногда используется параметризованное представление кода. Например, имена функций и переменных заменяются при первой встрече в коде на 0, а при последующих на расстояние до предыдущей позиции. Обычно детекторы, основанные на этих двух представлениях, лучше находят плагиат, чем системы, подсчитывающие отличительные черты, а также способны находить плагиат в тех случаях, когда скопирована только часть программы.

* + 1. Токенизация

Основная идея токенизированного представления – сохранение существенных и игнорирование поверхностных (то есть легко модифицируемых) деталей кода программы. Процедура токенизации выглядит примерно так:

* Каждому оператору языка (кроме пустого) приписываем код, назначенный заранее для каждого класса операторов. Также коды можно приписывать блочным операторам (begin/end {}), подключениям библиотек и заголовочных файлов.
* Строим строку из полученных кодов, сохраняя порядок следования их в исходном коде программы. Один символ строки (токен) – код одного оператора.

Таким образом, мы автоматически игнорируем названия функций и переменных (классов, объектов и т. д.), разделительные символы, предотвращаем влияние мелких изменений кода программы.

Процесс токенизации и разбиение операторов на классы зависит от языка программирования, используемого в исходном коде. К одному классу операторов обычно относят те, которым соответствует один идентификатор языка программирования, все вызовы функций, вызовы методов классов, объявления переменных элементарных типов, объявление экземпляров классов и т. д.

* 1. Алгоритм вычисления расстояния между деревьями

Расстояние между двумя упорядоченными помеченными деревьями – минимальная по стоимости последовательность операция над узлами, которая трансформирует одно дерево в другое. Мы рассматриваем следующие 3 операции:

* **delete**  - удалить узел и присоединить все его дочерние узлы к родительскому узлу удаляемого узла
* **insert –** вставить узл между некоторым узлом и подпоследовательностью дочерних узлов этого узла
* **rename –** переименовать узел

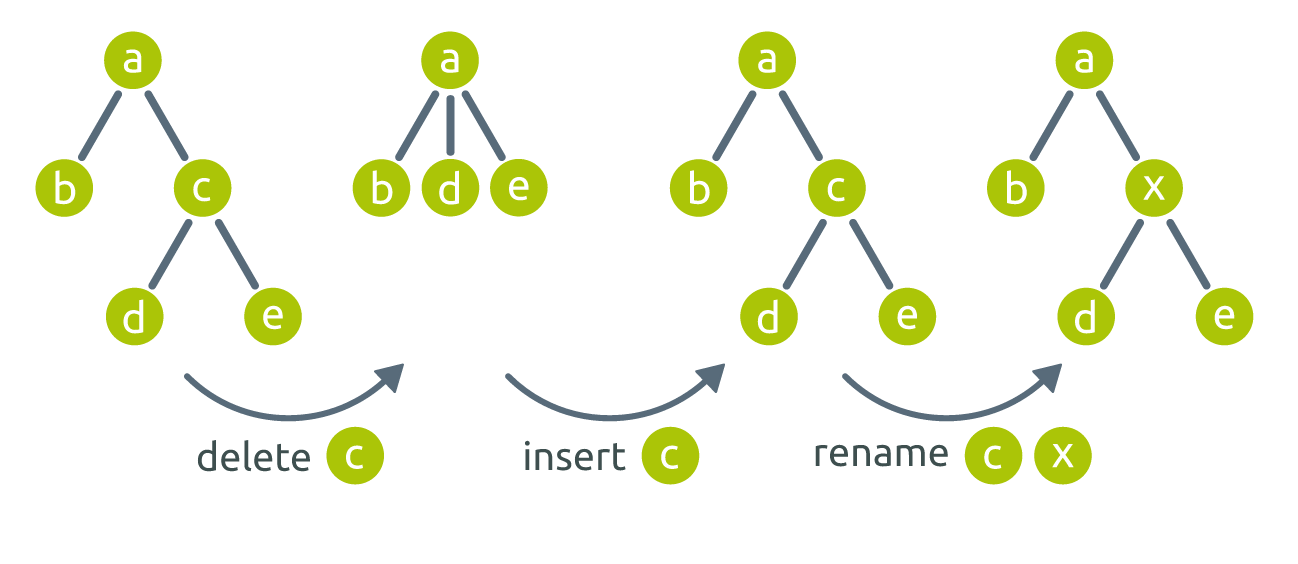


Рисунок 1 - Операции над узлами

Существует много различных способов трансформации одного дерева в другое. У каждой операции есть своя стоимость. Итоговая стоимость трансформации одного дерева в другое – сумма всех стоимостей в последовательности. Наша задача – найти такую последовательность, стоимость которой минимальна.

* + 1. Рекурсивное решение

Существует рекурсивное решение для разложения дерева на поддеревья. Расстояние между двумя лесами вычисляется за постоянное время на основе решений мелких подзадач. На каждом шаге рекурсии существуют два способа разложения: либо, удалив самый левый, либо самый правый корневой узел.

Расстояние между двумя лесами (голубой и зеленый) - минимальное расстояние из четырех небольших проблем. Расходы, связанные с операций редактирования, добавлены к соответствующим проблемам.

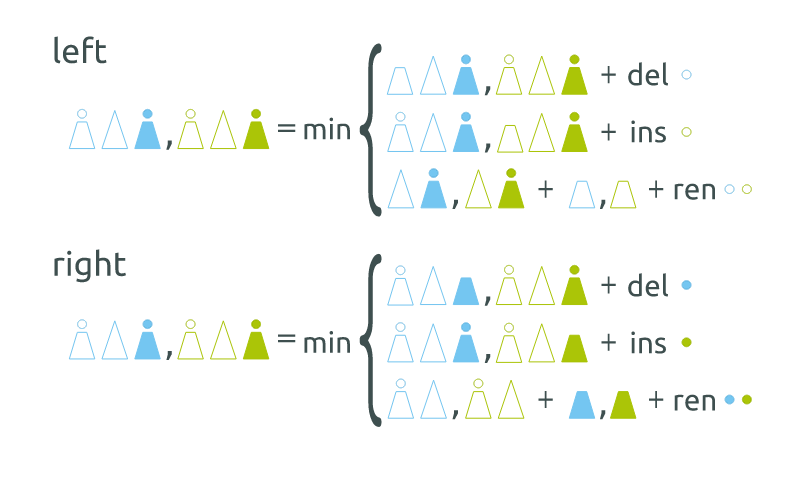


Рисунок 2 - Рекурсивное решение

Подзадачи, которые возникают в процессе вычисления расстояния редактирования дерева, называются **соответствующие подзадачи**. Их число, которое описывает и временную сложность, зависит от выбора между левой и правой стороной при каждом рекурсивном шаге. Алгоритмы вычисления расстояния стараются свести к минимуму количество соответствующих подзадач.

* + 1. Существующие алгоритмы

Простая реализация рекурсивного решения приводит к экспоненциальной сложности. «state-of-the-art» алгоритмы используют динамическое программирование для достижения полиномиального времени выполнения. Меньшие подзадачи запоминается и повторно используются для решения больших подзадач. Они используют так называемую стратегию разложения, чтобы определить выбор слева или справа на каждом рекурсивном шаге.

Несколько алгоритмов были предложены. Тай представил первое неэкспоненциальное решение в 1979 году. Его алгоритм работает за *O (m3n3)* по времени и памяти для деревьев с m и n узлами соответственно. Чжан и Шаша улучшили временную сложность *О (m2n2)*. Они всегда выбирают левое правило разложения. Более того, они используют тот факт, что не все возможные подзадачи необходимо вычислять. Клейн использует тяжелый (heavy) путь в одном из деревьев, чтобы направлять выбор между двумя рекурсивными решениями. Алгоритм Клейна работает за *O (n2mlog(m))* по аремени и памяти. Димэйн также использует тяжелые пути. Тем не менее, в отличие от Клейна, тяжелые пути используются в обоих деревьев. Это делает алгоритм более эффективен с *O (n2m (1 + log(mn)))* временной сложностью. Кроме того, аналогично Чжан и Шаша, Димэйн нашел хитроумный способ для разбиения на подзадачи и получить *O (MN)* затраты памяти.

Из-за временных и пространственных сложностей, двумя основными конкурентами являются алгоритмы Чжанг-Шаша и Димэйна. Первый эффективен для некоторых интересных форм деревьев, например, сбалансированных деревьев.Эффективность двух алгоритмов сильно зависит от формы дерева, и трудно выбрать между ними. Выбор неправильного алгоритма приводит к увеличению времени выполнения.

Надежный алгоритм (RTED – Robust Tree Edit Distance) был предложен Пауликом и Остеном. RTED не только независим от формы дерева, но для любого экземпляра задачи вычисляет наилучшую стратегию. Кроме того, во многих случаях RTED превосходит конкурентов. RTED работает в *O (N3)* по времени и *О (MN)* памяти.

* + 1. Стратегии

Выбор между левым и правым рекурсивным решением могут быть определены с помощью так называемых **путей разложения**. Они используют пути от корня до листеьв **(root-leaf path)** в дереве, чтобы разложить его на поддеревья и леса.

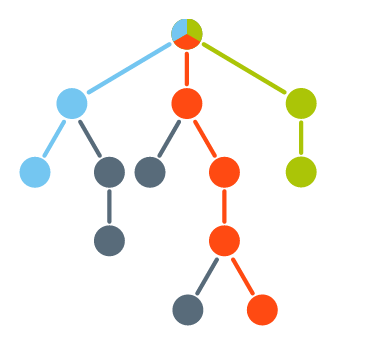


Рисунок 3 - Типы путей: левый, правый, тяжелый

Соответствующие поддеревья являются результатом устранения пути из дерева.

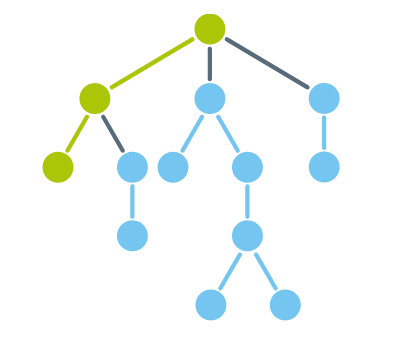


Рисунок 4 - Удалив зеленый путь, получим 3 поддерева

Соответствующие поддеревья могут быть получены в следующем порядке:

1. удалить корневой узел и остановиться, если узлов не осталось,
2. удалять крайний левый корневой узел, пока крайняя левый узел находится на пути
3. удалить правый узел, пока правый узел находится на пути
4. рекурсивно повторить процедуру в соответствующем поддереве.

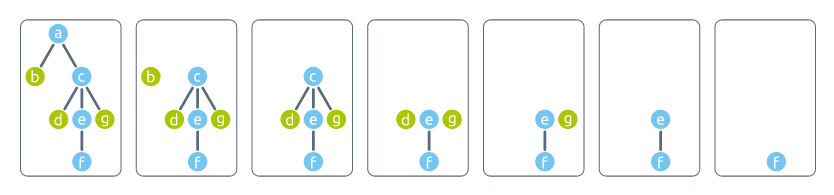


Рисунок 5 - Синий путь ведет к получению соотвествующих поддеревьев

* + 1. Общий алгоритм

Понятие **путь разложения** было использовано для вычисления расстояния в оптимальном *O(MN)* по памяти алгоритме. Кроме того, эти понятия позволяют разработать общий алгоритм для расстояния между деревьями, который для любой стратегии вычисляет расстояние, расходуя *O (MN)* памяти.

Функция пути (**single-path function**) вычисляет расстояние между двумя соответствующими поддеревьями в соответствии с выбранным путем. Для двух заданных деревьев, F и G, функция вычисляет расстояние между каждым поддеревом F, корень которого лежит в выбранном пути, и всеми поддеревьями G. Расстояния хранятся в матрице расстояний для последующего повторного использования. Расстояния между каждым поддеревом F (в зависимости от выбранного пути) и всеми поддеревьями G вычисляются заранее.

Пусть существует два дерева - F и G, мы можем вычислить расстояние между ними в два этапа. Во-первых, мы выбираем путь в одном из них, скажем, F, и вычисляем расстояния между соответствующими поддеревьями F и деревом G. Эти расстояния будут повторно в следующем шаге. Во-вторых, мы вычисляем расстояние между F и G в восходящем порядке вычисления расстояния между соответствующими лесами F и всеми соответствующими лесами группы G.

Общий алгоритм расстояния между деревьями использует стратегию пути для вычисления расстояния.

1. Для данной пары деревьев, (F, G), находим путь в стратегии путей.
2. Если путь в F - выполним следующие действия, в противном случае запустим алгоритм (G, F).

* Запустим алгоритм для каждого соответствующего поддерева F 'в F и дерева G.
* Вычислим функцию пути для (F, G) в зависимости от типа пути.
  1. Средства распознавания кода

Для начала рассмотрим базовые определения из теории построения компиляторов. Компиляторы работают со структурированными или формальными языками (formal languages). Это языки, для которых задан набор правил, описывающий все возможные конструкции этого языка. Этот набор правил называется грамматикой (grammar) языка. Все языки программирования (C, C++, Java, Python, Ruby, JavaScript и т.д.) и языки разметки (HTML, XML, JSON, TeX и т.д.) являются примерами формальных языков.

Согласно современной теории построения компиляторов анализ языка состоит из 2-х этапов:

* Лексический анализ (lexical analysis). На этом этапе лексер (lexer) разбивает поток символов (chars stream) на блоки, называемые токенами (tokens).
* Синтаксический анализ (parsing). На этом этапе парсер (parser) преобразует поток токенов (token stream), полученных от лексера в синтаксическое дерево или дерево разбора (parse tree).

**ANTLR** (от англ. ANother Tool for Language Recognition — «ещё одно средство распознавания языков») — генератор парсеров, позволяющий автоматически создавать программу-парсер (как и лексический анализатор) на одном из целевых языков программирования (C++, Java, C#, Python, Ruby) по описанию LL(\*)-грамматики на языке, близком к РБНФ. Позволяет конструировать компиляторы, интерпретаторы, трансляторы с различных формальных языков. Предоставляет удобные средства для восстановления после ошибок и сообщения о них. ANTLR — продолжение PCCTS (Purdue Compiler Construction Tool Set), который был разработан в 1989 г.

Основоположником проекта и его главным вдохновителем является проф. Теренс Парр (Terence Parr) из Университета Сан-Франциско. ANTLR — проект с открытым исходным кодом, версия 3.0 распространяется по лицензии BSD. Проект в настоящее время активно развивается.

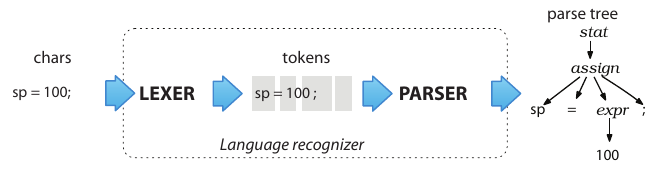


Рисунок 6 - Схема ANTLR

Нас интересуют 2 файла:

* **ANTLR Runtime**, распространяется с нашей программой, использующей ANTLR. Этот код нужен для работы с деревом разбора.
* **Complete ANTLR** (ANTLR Runtime + ANTLR Tools). ANTLR Tools нужен для генерации синтаксического анализатора и отладки.

Чтобы получить синтаксический анализатор некоторого языка, нам нужна грамматика этого языка, записанная в файле с расширением \*.g4 или \*.g(в зависимости от версии ANTLR – для версий 3 и ниже \*.g, для версии 4 \*.g4). Используя этот файл, ANTLR Tools сгенерирует файлы анализатора.

ANTLR имеет набор классов для поддержки семантического анализа. Он поддерживает 2 подхода:

* основанный на использовании слушателя (listener). Это аналог SAX-парсера для XML. В этом случае мы запускаем обходчик дерева, который при входе и выходе в каждый из узлов соответствующие методы.
* основанный на использовании посетителя (visitor). Это аналог DOM для XML. В этом случае мы сами вызываем соответствующие методы по мере необходимости.
  1. Форма Бэкуса — Наура

Форма Бэкуса — Наура (сокр. БНФ, Бэкуса — Наура форма) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик.

Используется для описания синтаксиса языков программирования, данных, протоколов (например, в документах RFC) и т. д. (причем как грамматики, так и регулярной лексики, поскольку регулярные грамматики являются подмножеством контекстно-свободных).

Расширенная форма Бэкуса — Наура (расширенная Бэкус — Наурова форма (РБНФ)) (англ. Extended Backus–Naur Form (EBNF)) — формальная система определения синтаксиса, в которой одни синтаксические категории последовательно определяются через другие. Используется для описания контекстно-свободных формальных грамматик. Предложена Никлаусом Виртом. Является расширенной переработкой форм Бэкуса-Наура, отличается от БНФ более «ёмкими» конструкциями, позволяющими при той же выразительной способности упростить и сократить в объёме описание.

Описание грамматики в РБНФ представляет собой набор правил, определяющих отношения между терминальными символами (терминалами) и нетерминальными символами (нетерминалами).

* **Терминальные** символы — это минимальные элементы грамматики, не имеющие собственной грамматической структуры. В РБНФ терминальные символы — это либо предопределённые идентификаторы (имена, считающиеся заданными для данного описания грамматики), либо цепочки — последовательности символов в кавычках или апострофах.
* **Нетерминальные** символы — это элементы грамматики, имеющие собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики. В РБНФ каждый нетерминальный символ имеет имя, которое представляет собой строку символов.

Пример конструкции:

Синтаксис = { СинтОператор }.

СинтОператор = идентификатор "=" СинтВыражение ".".

СинтВыражение = СинТерм {"|" СинТерм}.

СинТерм = СинтФактор { СинтФактор }.

СинтФактор = идентификатор | цепочка

| "(" СинтВыражение ")" | "[" СинтВыражение "]"

| "{" СинтВыражение "}".

К достоинствам РБНФ можно отнести простоту, достаточную мощность и наглядность, что делает его удобным для выполнения описаний, предназначенных не только для автоматической интерпретации, но и для чтения людьми. К недостаткам РБНФ можно отнести тот факт, что они описывают грамматическую структуру формального языка без учёта контекстных зависимостей, а значит, при наличии таких зависимостей РБНФ-описание оказывается неполным, и некоторые правила синтаксиса описываемого языка приходится излагать в обычной текстовой форме. Это приводит к тому, что текст, точно соответствующий РБНФ-грамматике, может, тем не менее, оказаться синтаксически некорректным.

* 1. Самоорганизующаяся карта Кохонена

**Самоорганизующаяся карта Кохонена** — нейронная сеть с обучением без учителя, выполняющая задачу визуализации и кластеризации. Является методом проецирования многомерного пространства в пространство с более низкой размерностью (чаще всего, двумерное), применяется также для решения задач моделирования, прогнозирования и др.

Самоорганизующиеся карты Кохонена используются для решения таких задач, как моделирование, прогнозирование, выявление наборов независимых признаков, сжатие информации, а также для поиска закономерностей в больших массивах данных. Наиболее часто описываемый алгоритм применяется для кластеризации данных

* + 1. Структура сети

Самоорганизующаяся карта состоит из компонентов, называемых узлами или нейронами. Их количество задаётся аналитиком. Каждый из узлов описывается двумя векторами. Первый — т. н. вектор веса *m*, имеющий такую же размерность, что и входные данные. Второй — вектор r, представляющий собой координаты узла на карте.

Изначально известна размерность входных данных, по ней некоторым образом строится первоначальный вариант карты. В процессе обучения векторы веса узлов приближаются к входным данным. Для каждого наблюдения (семпла) выбирается наиболее похожий по вектору веса узел, и значение его вектора веса приближается к наблюдению. Также к наблюдению приближаются векторы веса нескольких узлов, расположенных рядом, таким образом, если в множестве входных данных два наблюдения были схожи, на карте им будут соответствовать близкие узлы. Циклический процесс обучения, перебирающий входные данные, заканчивается по достижении картой допустимой (заранее заданной аналитиком) погрешности, или по совершении заданного количества итераций.

В результате работы алгоритма получаются следующие карты:

* **карта входов нейронов** — визуализирует внутреннюю структуру входных данных путем подстройки весов нейронов карты.
* **карта выходов нейронов** — визуализирует модель взаимного расположения входных примеров. Очерченные области на карте представляют собой кластеры, состоящие из нейронов со схожими значениями выходов.
* **специальные** карты — это карта кластеров, полученных в результате применения алгоритма самоорганизующейся карты Кохонена, а также другие карты, которые их характеризуют
  + 1. Работа сети
* Инициализация карты, то есть первоначальное задание векторов веса для узлов.
* Цикл:

1. Выбор следующего наблюдения (вектора из множества входных данных).
2. Нахождение для него лучшей единицы соответствия (best matching unit, BMU, или Winner) — узла на карте, вектор веса которого меньше всего отличается от наблюдения (в метрике, задаваемой аналитиком, чаще всего, евклидовой).
3. Определение количества соседей BMU и обучение — изменение векторов веса BMU и его соседей с целью их приближения к наблюдению.
4. Определение ошибки карты.
   * 1. Алгоритм

* **Инициализация**

Наиболее распространены три способа задания первоначальных весов узлов:

1. Задание всех координат случайными числами.
2. Присваивание вектору веса значение случайного наблюдения из входных данных.
3. Выбор векторов веса из линейного пространства, натянутого на главные компоненты набора входных данных.

* **Цикл**

Пусть t — номер итерации (инициализация соответствует номеру 0).

1. Выбрать произвольное наблюдение x(t) из множества входных данных.
2. Найти расстояния от него до векторов веса всех узлов карты и определить ближайший по весу узел . Это — BMU или Winner. Условие на :

для любого , где — вектор веса узла Если находится несколько узлов, удовлетворяющих условию, BMU выбирается случайным образом среди них.

1. Определить с помощью функции h (функции соседства) соседей и изменение их векторов веса.

* **Задание h**

Функция определяет «меру соседства» узлов и и изменение векторов веса. Она должна постепенно уточнять их значения, сначала у большего количества узлов и сильнее, потом у меньшего и слабее. Часто в качестве функции соседства используется гауссовская функция:

где 0 < < 1 — обучающий сомножитель, монотонно убывающий с каждой последующей итерацией (то есть определяющий приближение значения векторов веса BMU и его соседей к наблюдению; чем больше шаг, тем меньше уточнение);

— координаты узлов и на карте;

— сомножитель, уменьшающий количество соседей с итерациями, монотонно убывает.

Параметры и их характер убывания задаются аналитиком.

Более простой способ задания функции соседства:

,

если находится в окрестности заранее заданного аналитиком радиуса, и 0 в противном случае.

Функция равна для BMU и уменьшается с удалением от BMU.

* Изменение векторов веса

Изменить вектор веса по формуле:

Таким образом, вектора веса всех узлов, являющихся соседями BMU, приближаются к рассматриваемому наблюдению.

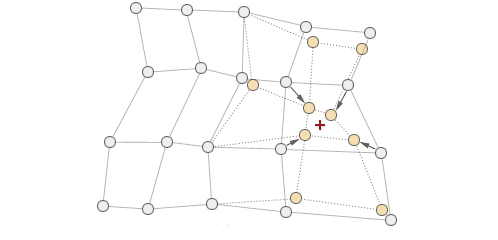


Рисунок 7- Приближение узлов к рассматриваемому наблюдению

**Окрестность** представляет собой несколько нейронов, которые окружают нейрон-победитель. Сначала к окрестности принадлежит большое число нейронов, далее ее размер постепенно уменьшается. Сеть формирует топологическую структуру, в которой похожие примеры образуют группы примеров, близко находящиеся на топологической карте.

Полученную карту можно использовать как средство визуализации при анализе данных. В результате обучения карта Кохонена классифицирует входные примеры на кластеры (группы схожих примеров) и визуально отображает многомерные входные данные на плоскости нейронов.

Уникальность метода самоорганизующихся карт состоит в преобразовании n-мерного пространства в двухмерное. Применение двухмерных сеток связано с тем, что существует проблема отображения пространственных структур большей размерности.

Имея такое представление данных, можно визуально определить наличие или отсутствие взаимосвязи во входных данных.

* Вычисление ошибки карты

Например, как среднее арифметическое расстояний между наблюдениями и векторами веса соответствующих им BMU:

где N — количество элементов набора входных данных.

* 1. Планирование задач в Spring

Планирование задач в основном состоит из трех частей: задачи (представляющей собой порцию бизнес-логики, которая должна запускаться в определенное время или на регулярной основе), триггера (указывающего условие, при удовлетворении которого задача должна выполняться) и планировщика (который запускает задачу на основе информации, полученной от триггера).

Для планирования задач в Spring доступны три варианта.

* Поддержка JDK-объекта Timer. При планировании задач в Spring поддерживается объект Timer из JDK.
* Интеграция с Quartz. Платформа Spring интегрирована с Quartz Scheduler
* Абстракция TaskScheduler, встроенная в Spring.

В Spring-абстракции TaskScheduler имеются три главных участника.

* Интерфейс Trigger. Предоставляет поддержку для определения механизма запуска. В Spring доступны две реализации Trigger. Класс CronTrigger поддерживает запуск на основе выражения cron, а класс PeriodicTrigger — запуск на основе начальной задержки и затем фиксированного интервала.
* Задача. Задача — это порция бизнес-логики, запуск которой необходимо запланировать. В Spring задача может быть указана как метод внутри любого бина Spring. Описание задачи включает следующие параметры:
* **ref**. Объект из контекста bean, метод которого будет выполняться по расписанию;
* **method**. Имя метода объекта. Именно данный метод будет вызываться по расписанию;
* **fixed-delay**\*. Запуск осуществляется через определенный интервал времени (в миллисекундах). При этом отсчета ведется после окончания выполнения последней задачи;
* **fixed-rate**\*. Запуск осуществляется через определенный интервал времени (в миллисекундах). При этом отсчет ведется после начала запуска метода;
* **cron**\*. Параметры запуска задаются в нотации cron.

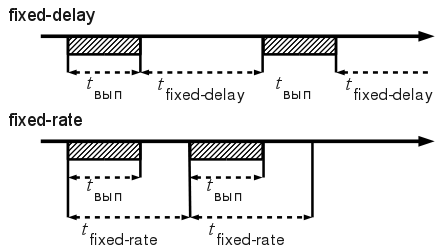


Рисунок 8 - Разница между fixed-delay и fixed-rate

\*Можно указать один из способов времени запуска: fixed-delay | fixed-rate | cron.

* Интерфейс TaskScheduler. Предоставляет поддержку для планирования задач. В Spring доступны три класса реализации интерфейса TaskScheduler: TimerManagerTaskScheduler, ConcurrentTaskScheduler и ThreadPoolTaskScheduler.

На рисунке показаны отношения между интерфейсом Trigger, интерфейсом TaskScheduler и задачей (которая реализует интерфейс java.lang.Runnable).

Планировать задачи с применением абстракции TaskScheduler из Spring можно двумя способами. Один из них предусматривает использование пространства имен task в XML-конфигурации Spring, а другой — аннотаций.

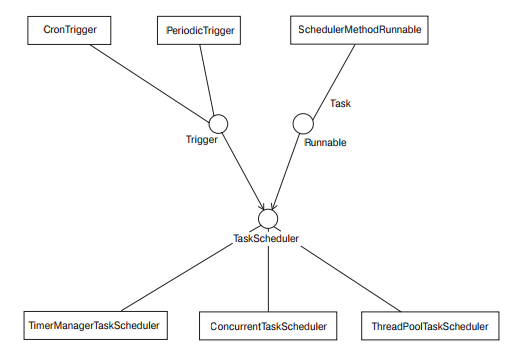


Рисунок 9 - Отношения между триггером, задачей и планировщиком

1. ОБЗОР СУЩЕСТВУЮЩИХ СИСТЕМ ОБНАРУЖЕНИЯ ПЛАГИАТА
   1. Детектор JPlag

JPlag разработан Guido Malpohl в 1996 году в Univercity of Karlsruhe. Изначально JPlag предназначался для нахождения плагиата среди упражнений студентов, на них же он и тестировался. Детектор поддерживает языки: Java, С, С++, Scheme.

Алгоритм работает в 2 этапа:

1) Все программы, которые надо сравнить, синтаксически анализируются и преобразуются в последовательность токенов.

2) Эти токены попарно сравниваются, для каждой пары определяется "похожесть". Это делается алгоритмом Greedy String Tiling. В завершении JPlag выводит в виде HTML страниц найденные сходные подстроки - для возможности дальнейшего анализа.

**Эффективность**. В техническом отчете утверждается, что, по результатам тестирования 12 различных наборов Java программ, JPlag находит весь плагиат за очень маленьким исключением. Время выполнения при этом меньше чем одна минута на запрос из 100 программ, по несколько сотен строк каждая.

Доступность пользователю. Надо заполнить форму по адресу <https://swt.ira.uka.de:2222/RequestForm.j>. После заполнения формы на указанный e-mail приходит письмо, в котором просят подтвердить желание зарегестрироваться. В письме указана ссылка, по которой нужно зайти для подтверждения. После этого введенные в форме данные уходят на рассмотрение к администратору JPlag. Если он сочтет нужным, он разрешит регистрацию. Результат рассмотрения приходит в следующем письме.

* 1. Детектор MOSS

MOSS (Measure Of Software Similarity). Разработан в 1994 году Алексом Эйкеном. Поддерживает языки: C, C++, C#, Java, Python, Visual Basic, Javascrpt, FORTRAN, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, Matlab.

Полного описания алгоритма MOSS нет. Основные идеи: используется метод «просеивания» (winnowing) для построения идентификационных меток (fingerprints). Разработчик утверждает, что алгоритм, на котором основан MOSS, модифицирован против всех известных автору алгоритмов обмана детектора.

Для регистрации необходимо послать письмо на адрес  [moss@moss.stanford.edu](mailto:%22moss@moss.stanford.edu%22) с любым полем subject и 2 строчками:

registeruser   
 mail *username@domain*

После этого мы получим скрипт на языке Perl, который содержит UserId, с помощью которого мы и получает доступ к системе.

* 1. Детектор SID

SID (Software Integrity System). Поддерживаемые языки: Java, C++.

Алгоритм основан на использовании Колмогоровской сложности и алгоритма TokenCompress. Работает в 2 этапа:

1. Исходный код синтаксически анализируется и преобразуется в последовательность токенов.
2. Для всех программ попарно вычисляется d(x, y) – мера общей информации. Для этого используется алгоритм TokenCompress. После этого все пары сортирутся по их d и выводятся.

Авторы SID утверждают, что обмануть систему очень сложно.

Доступен в online режиме. Нужно заполнить простую форму с логином, паролем, адресом электронной почты. На оставленный e-mail будут приходить результаты сравнения файлов на предмет плагиата.

* 1. Сравнительная характеристика детекторов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Детектор | Программные требования | Представление результатов | Автоматизация | Представление результатов | Языки |
| JPlag | запрос, разре­шение админи­стратора | JRE, web-браузер | нет | окно с фрей­мами | Java, С, С++, Scheme |
| MOSS | письмо с запросом | Perl, web-браузер | да | окно с фрей­мами | C, C++, C#, Java, Python, Visual Basic, Javascrpt, FORTRAN, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, Matlab |
| STD | заполнить форму | zip-архиватор, web-браузер | да | окно с фрей­мами | Java, C++ |

Таблица 1 - Сравнительная характеристика

Если ограничиться проверкой небольших программ, состоящих из одного файла, то все детекторы показывают требуемый результат. В более сложных случаях, максимально возможное сходство программ 10-20%, когда, полученный порог сходства для плагиата не ниже 30%, то есть при массовой проверке с большой базой данных мы получим большое количество ложных совпадений, которые необходимо проверить, чтобы не пропустить настоящий случай плагиата.

1. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРИЛОЖЕНИЯ
   1. Структура базы данных

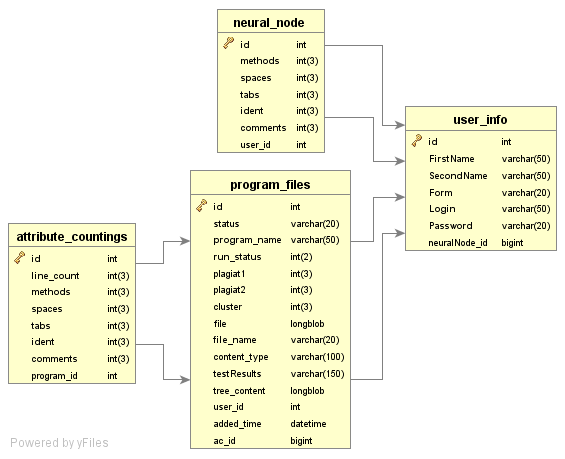


Рисунок 10 - Структура базы данных

* 1. Use-case диаграмма



Рисунок 11 - Use-case диаграмма

* 1. Порядок обработки программ

1. Студент загружает исходный код программы, программа получает статус **0 (UPLOADED)**;
2. Производится проверка программы на соответствие тестам, после этого программа получает статус равный **1 (TESTED)**;
3. Производится проверка программы на плагиат, статус программы равен **2(CHECKED\_PLAGIARISM)**;
4. Программа отдается на обработку нейронной сети для определения автора данной программы, статус меняется на **3 (EXEXUTED\_NEURAL\_NETWORK)**;
5. \*В случае возникновения ошибок на одном из четырех этапов программе ставится статус равный **4 (FAILED).**
   1. Алгоритм тестирования программы

Для проверки программы на соответствие тестам производятся следующие действия:

1. На жестком диске создается папка с <username\_номер\_лабораторной>. В эту папку помещается входной файл.
2. Компилятор пытается обработать входной файл и сгенерировать exe-файл.
3. В случае неудачи захватываются сообщения компилятора и выдаются пользователю.
4. В противном случае проверяется сгенерированный exe-файл. На жестком диске хранятся входные данные и правильные ответы. По очереди отдаем на проверку каждый тест.
5. Результаты тестов записываются в файл и сохраняются в базу данных.
6. Есть ограничение на время прохождения теста. Используется класс com.google.common.util.concurrent.TimeLimiter.

* Создан интерфейс Checker и его имплементация TimeChecker, содержащий единственный метод int compileFile(Process process);
* Создается proxy-объект, указывается максимальное время работы метода
* Вызывается метод: proxy.compileFile(…);
* Если время работы метода больше указанного значения, выбрасывается UncheckedTimeoutException – тест помечается как failed
* В противном случае, если время не превышено и ответ правильный, тест помечается как passed;

1. После прохождения всех тестов папка с входным файлом и сгенерированным exe-файлом удаляется с жесткого диска.
   1. Проверка на наличие плагиата

Проверка на наличие плагиата проводится в 3 этапа:

* на первом этапе используется атрибутный алгоритм, который выделяет в программе определенные характеристики и стоит n-мерный вектор и сравнивает этот вектор с уже существующими;
* на втором этапе, если первый алгоритм не нашел плагиат, происходит проверка с помощью структурного алгоритма;
* на третьем этапе, используя результаты первых двух этапов, программа подается на обработку сети Кохонена, определяется предполагаемый автор программы.

Общая схема поиска плагиата (Рисунок 12):

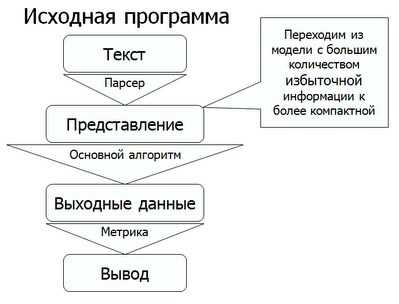


Рисунок 12 - Общая схема поиска плагиата

* + 1. Реализация атрибутного алгоритма

В качество атрибутов для отслеживания были выбраны следующие:

* Количество объявлений переменных
* Количество методов
* Количество вывозов методов
* Количество операций +-\*/
* Количество присваиваний
* Количество циклов
* Количество условных операторов

В результате генерируем вектор, i-я координата которого, отвечает за определенный атрибут.

Расстояние между двумя атрибутами v1[i], v2[i] - модуль их разности:

Степень похожести 2 векторов определяем по формуле:

Очевидно, такой алгоритм неэффективен для обнаружения плагиата в исходном коде программы, и используется при первоначальной проверке. Если 2 программы, реализующие одну и ту же задачу, будут близки при проверке атрибутным методом, то, скорее всего, имеет место плагиат. В противном случае они будут подвергнуты проверке другим, более сложным методом.

* + 1. Реализация структурного алгоритма RTED

В данной работе был реализован алгоритм RTED (Robust Tree Edit Distance), описанный в пункте 2.3. Напомним, что для каждой операции (удаление, вставка, переименование) должна быть определена стоимость. В данной работе стоимость определяется следующим образом:

* **удаление** – стоимость равна **1**
* **вставка** – стоимость равна **1**
* **переименование** – стоимость зависит от типа сравниваемых узлов

Всевозможные типы узлов разбиты на группы. Например, условные операторы(if, switch), операторы цикла (while, for), арифметические операторы (+-\*/) и т. д... Если узлы находятся в разных группах – стоимость переименование равна **1,** в противном случае она равна – *exp(-Diff(node1, node2))*, где *Diff* – функция, которая вычисляет расстояние Левенштейна между двумя строками. (т. е. количество операций вставок, удаления, изменения слов, чтобы из 1 строки получилась 2).

Для этого, в property-файлах (отдельных для каждого языка программирования) хранятся коды для каждой операции.

Пример:

**FORMAL\_PARAM\_LIST=133**

**FORMAL\_PARAM\_STD\_DECL=134**

**BLOCK\_SCOPE=117**

**FUNCTION\_METHOD\_DECL=136**

**VOID\_METHOD\_DECL=163**

**RETURN=88**

Далее, все эти операции объединены в группы по типу операции. Например, IF и SWITCH объединены в одну группу под названием CONDITION.

Если у двух операций разные группы, то стоимость переименования будет равна 1, если одинаковые (следовательно, возможна преднамеренная замена одного оператора другим для сокрытия плагиата) – стоимость будет меньше 1, что, очевидно, улучшает результаты.

* + 1. Реализация нейронной сети

1. Выбор атрибутов нейронной сети.

Для того чтобы подать программу на обработку нейронной сети, ее надо преобразовать к соответствующему виду. В данном приложении используется *n*-мерное представление программы. На данном этапе обработки программы необходимо окончательно определить автора данной программы. Поэтому в качестве атрибутов мы выбираем «стилистические» особенности автора, а именно:

* Количество пробелов
* Количество табуляций
* Количество методов
* Средняя длина используемых идентификаторов
* Средняя длина комментариев
* Количество «магических» чисел (численных констант) и др.

а также используется степень плагиата программы, вычисленная на первом и втором этапах.

1. Инициализация карты.

Количество классов, на которые должны быть разбиты входящие данные – количество учеников (студентов). При добавлении нового студента в систему ему ставится в соответствии свой класс. При первой загрузке файла в систему данным студентом вектор, соответствующий этому файлу, принимается в качестве первоначального веса для данного класса.

1. Вычисление расстояния между узлами.

,

1. Функция соседства

,

где 0 < < 1 — обучающий сомножитель, в нашем случае

* 1. Получение древовидной структуры программы

Для дальнейших вычислений степени заимствования кода с помощью структурного алгоритма, необходимо получить программу в виде дерева. Для построения дерева была использована технология ANTLR. С помощью ANTLR можно сгенерировать парсер кода, который преобразует его в дерево. В данной системе допустимые языки программирования – С++, Java, Pascal. Таким образом, с помощью соотвествующих грамматик для данных языков были построены парсеры кода.

* Java.g – файл грамматики языка Java.
* Cpp.g – файл грамматики языка Java.
* Pascal.g – файл грамматики языка Java.

Для каждой грамматики сгенерируем лексер и парсер:

$ java -cp antlr-3.5.1.jar org.antlr.Tool \

-o src/com/by/iason/plagiat/model \

-package com.by.iason.plagiat.model \

-listener \

grammars/<имя\_файла>.g

В результате сгенерировались следующие файлы:

\*Lexer.java - лексер;

\*Parser.java - парсер;

\*.tokens, \*Lexer.tokens -- служебные файлы со списком токенов.

Эти файлы затем мы просто добавляем в проект. Пример получения дерева для языка Java.

ANTLRInputStream in = new ANTLRInputStream(bytes);

JavaLexer lexer = new JavaLexer(in);

CommonTokenStream tokens = new CommonTokenStream(lexer);

JavaParser parser = new JavaParser(tokens);

CommonTree tree =

(CommonTree) parser.compilationUnit().getTree();

В итоге мы получаем объект ***CommonTree***, который хранит структуру программы.

* 1. Организация тестирования по расписанию

Задача решается с использованием фреймворка Spring Scheduler, входящего в состав Spring Framework.

Созданы 3 класса с задачами:

* **TestsRunner** – запускает проверку программ на соответствие тестам
* **PlagiarismRunner** – запускает проверку программы на наличие плагиата аттрибутным и структурным методами
* **NeuralNetworkRunner** – запускает нейронную сеть

Для каждой задачи определено время, когда она запускается, с помощью выражения cron.

* 1. Визуализация результатов

На рисунке 9 представлен наглядный отчет о тестировании.

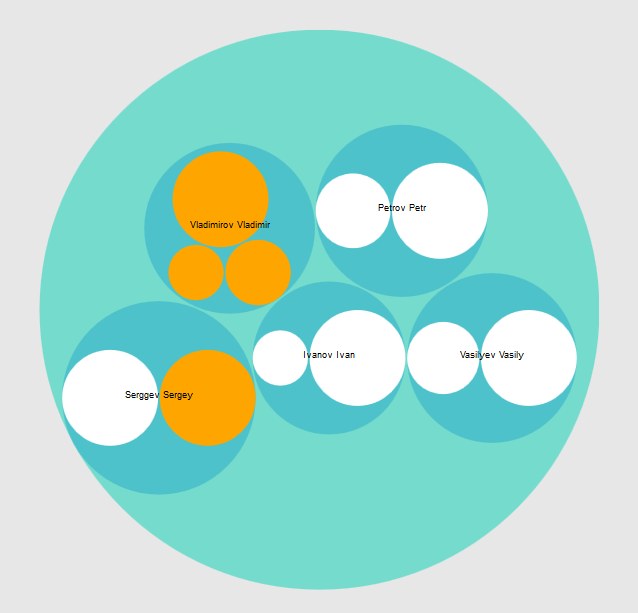


Рисунок 13 - Визуализация результатов поиска плагиата

Чем больше радиус круга – тем больше степень плагиата, вычисленная на первых двух этапах. Если круг белый, то по стилистическим особенностям, описанным в пункте 3.2.3, программа принадлежит автору. Оранжевые круги – подозрительные на плагиат программы.

Кликнув на определенного студента, увидим его программы (рисунок 13).

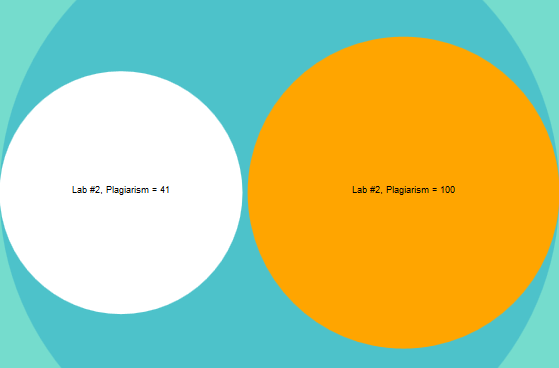


Рисунок 14 - Список программ студента

* 1. Описание основных классов системы

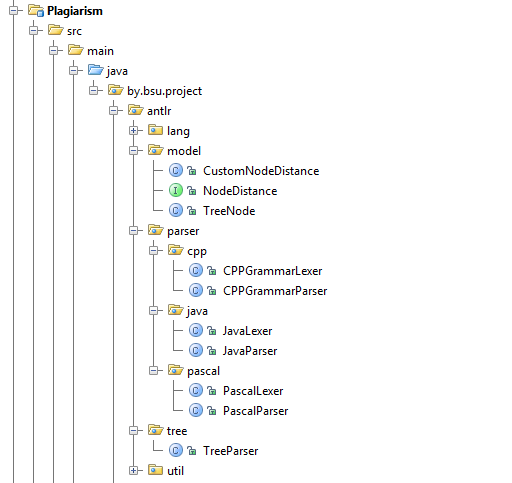


Рисунок 15 - Структура классов

* + 1. Класс ProgramFilesUtil

Реализует проверку программы на первом этапе (соответствие тестам)

Методы класса ProgramFilesUtil:

* **ProgramFilesUtil(MultipartFile file, UserInfoEntity user, String programName)** – конструктор с параметрами
* **boolean compile(String cmd, String postfix)**

**cmd** – строка, содержащая параметры командной строки в зависимости от расширения поступающего файла

**postfix** – расширение файла

* **getDigitalMarsMessages(InputStream in)**

**getFreePascalMessages(InputStream in)**

**getTinyCCompilerMessages(InputStream in)**

**in** – входной поток проекта

возвращают сообщения компиляторов

* **boolean checkAllInputFiles() -** проверяет загруженный файл на соответствие заготовленным тестам, хранящимся на жестком диске
* **boolean compareFiles()** – проверяет выходной файл программы “out.txt” на соответствию файлу с правильными ответами
* **byte[] getTestResults()** – возвращает массив байт, содержащий информацию о прохождении тестов.

Шаблон #№теста: passed/failed

* + 1. Класс TreeNode

Для хранения узла дерева был создан класс **TreeNode**, который содержит следующую информацию:

* **Long id** - идентификатор узла
* **Long parentId** - идентификатор предка
* **String name** - имя узла
* **Operation operation** - тип узла
* **List<TreeNode> children** – список дочерних узлов

Целое дерево представлено списком TreeNode – List<TreeNode>.

Для того, чтобы для каждой новой проверки не запускать парсеры для программ, для которых уже проверки проводились, мы храним содержимое листа. Это производится с помощью сериализации. Поэтому класс TreeNode реализует интерфейс Serializable так же в нем есть 2 метода для сериализации/десериализации списка узлов:

* **List<TreeNode> getTree(byte data[])**
* **byte[] getBytes(List<TreeNode> list)**
  + 1. Интерфейс NodeDistance

Интерфейс NodeDistance определяет 3 операции, возможные над узлами:

* delete
* insert
* rename

Класс CustomNodeDistance – его реализация.

Как было сказано выше, стоимости операций вставок и удалений равны **1.**

Стоимость операции переименования зависит от типа узлов.

Для этого создан enum Type: **CLASS, VAR, METHOD, IDENT, ASSIGN, PLUS, MINUS, MULTIPLY, DIVIDE, NUMBER\_LITERAL, STRING\_LITERAL, BOOL\_LITERAL** и т. д. ...

Для каждого узла определен тип. Таким образом, если тип узлов не совпадает(например, объявление переменной и условный оператор), стоимость переименования равна **1,** если совпадает (например, for и while) – вычисляется расстояние Левеншштейна между строчками кода, соответствующим данным узлам

В данном примере расстояние буде находится между следующими строками:

* **for (i = 0; i<n; i++)**
* **while(i<n).**

Очевидно, расстояние будет больше нуля. А следовательно, и стоимость, которая вычисляется по формуле будет меньше 1.

* + 1. Класс TreeParser

С помощью сгенерированных парсеров мы можем получить объект класса CommonTree, описывающий древовидную структуры нашей программы. Однако, использовать CommonTree для сравнения неудобно:

* Для дальнейшего использования в алгоритме нахождения расстояния между леревьями узлы должны быть пронумерованы
* CommonTree содержит много «бесполезных» узлов, которые нам неинтересны при проверке на плагиат. Например, комментарии, подключение библииотек и т. д., которые, тем не менее, увеличивают количество узлов в дереве и усложняют задачу.
* Для трех парсеров (java, c++, pascal) названия некоторых узлов не совпадают, что не подходит для сравнеий программ написанных на разных языках

Для решения этих проблем был создан класс TreeParser.

Основные методы:

* **List<TreeNode> getTree(byte[] bytes)** – получить список узлов программы
  + **bytes** – код программы
* **TreeParser(Lang lang)** – конструктор
  + **lang** – язык, на котором написана программа
* **TreeParser(Lang lang, boolean checkIdentifiers)**
  + **lang** – язык, на котором написана программа
  + **checkIdentifiers** – указать, учитывать ли при проверки имена переменных, методов, классов и др.
    1. Класс TreeEditDistance

Класс TreeEditDistance реализует алгоритм нахождения расстояния между деревьями, описанный в пункте 2.3.

Основные методы:

* **TreeEditDistance(NodeDistance nd, List<TreeNode> t1, List<TreeNode> t2)**
  + **nd** – объект, определяющий операции изменения узлов
  + **t1, t2** - деревья для сравнения
* **double calculate()** – вычисление расстояния между деревьями
  + 1. Класс RTED

Реализует RTED-алгоритм (нахождение минимального расстояния между двумя деревьями).

Основные методы:

* **public double nonNormalizedTreeDist(List<TreeNode> t1, List<TreeNode> t2)** – рассчитывает расстояние между деревьями
* **public void init(List<TreeNode> t1, List<TreeNode> t2)** – инициализация; каждое дерево представлено как список его вершин
* **public void computeOptimalStrategy()** – рассчитывает оптимальную стратегию для пары деревьев
* **private double computeDistUsingStrArray()** – рассчитывает расстояние между двумя деревьями на основе заранее просчитанной стратегии.
  + 1. Класс NeuralNode

Создан для описания узла нейронной сети.

Атрибуты класса:

* **private Integer methods –** среднее количество методов
* **private Integer spaces –** среднее количество пробелов
* **private Integer tabs –** среднее количество табуляций
* **private Integer ident -** средняя длина идентификатора
* **private Integer comments –** средняя длина комментариев
* **private Integer magicNumbers –** среднее количество магических чисел
  + 1. Класс NeuralAttributesExtractor

Создан для генерации n-мерного вектора, поступающего на вход сети.

* **public NeuralAttributesExtractor (byte[] source)–** конструктор

**sourse** – исходный файл программы

* **public Integer getSpaces()**
* **public Integer getTabs()**
* **public Integer getMethods()**
* **public Integer getIdentifiers()**
* **public Integer getMagicNumbers()**
* **public Integer getComments()**

ЗАКЛЮЧЕНИЕ

В данной работе были изучены следующие предметные области:

* алгоритмы поиска плагиата
* средства распознавания языков
* способы представления программ
* критерии оценки программ

В данной работе были реализованы 2 метода для сравнения: атрибутный и структурный. Атрибутные методы подходят только для первоначальной проверки и дают плохие результаты, но зато быстро работают и просты в реализации. Структурные методы дают хорошие результаты, но работают значительно дольше. Поэтому в данной системе файл сначала проверяют атрибутным методом, и если он не прошел, то велика вероятность плагиата. В противном случае файл идет дальше на проверку структурным методом.

В результате данной работы была реализована автоматизированная тестирующая система с возможность поиска плагиата в исходных кодах программ.

ЛИТЕРАТУРА

1. *Pawlik, M.,* RTED: A Robust Algorithm for the Tree Edit Distance / M. Pawlik, N. Augsten // PVLDB 5(4):334–345, 2011.
2. *Moussiades, L.,* Pdetect: A clustering approach for detecting plagiarism in source code datasets / L. Moussiades, A. Vakali // The computer journal - 48(6), 2005.
3. *Lancaster, T.*, A comparison of source code plagiarism detection engines. / T. Lancaster, F. Culwin // Computer Science Education - 14:2:101 – 117, 2004.
4. *Joy, M.S.,* Plagiarism in Programming Assignments(1999). - Joy, M.S., Luck, M. IEEE Transactions on Education, 42(2). pp. 129-133. ISSN 0018-9359
5. *Левенштейн В.И.,* Двоичные коды с исправлением выпадений, вставок и замещений символов: Доклады Академий наук СССР / В.И. Левенштейн – М.: Проспект, 2009. – 861 с.
6. Определение плагиата – Википедия [Электронный ресурс]. – Режим доступа: <http://ru.wikipedia.org/wiki/Определение_плагитата>, 19.05.14, свободный – Загл. с экрана.
7. Расстояние Левенштейна – Википедия [Электронный ресурс]. – Режим доступа: [http://ru.wikipedia.org/wiki/Расстояние\_Левенштейна](http://ru.wikipedia.org/wiki/Расстояние_Левенштейна%20) , 19.05.14, свободный – Загл. с экрана.
8. ANTLR – Википедия [Электронный ресурс]. – Режим доступа: <http://ru.wikipedia.org/wiki/ANTLR>, 09.02.15, свободный – Загл. с экрана.
9. Самоорганизующаяся карта Кохонена – Википедия [Электронный ресурс]. – Режим доступа: [http://ru.wikipedia.org/wiki/ Самоорганизующаяся\_карта\_Кохонена](http://ru.wikipedia.org/wiki/%20Самоорганизующаяся_карта_Кохонена), 01.05.15, свободный – Загл. с экрана.