МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра Прикладная математика и информатика
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Информационные системы и технологии корпоративного управления
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему «Моделирование процесса автоматической проверки заданий по программированию при онлайн обучении»

Студент А. О. Кузовенков
(И.О. Фамилия) (личная подпись)

Научный руководитель к. т. н., доцент, А. Б. Кузьмичев
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

Содержание

Введение

Онлайн-образование, несмотря на свою относительную молодость, в настоящее время набирает всё большую популярность, так как благодаря нему можно получать знания, находясь практически в любом месте, и стоимость такого обучения значительно ниже. Как и при обычном обучении, получая образование дистанционно, необходимо выполнять различные работы, которые впоследствии буду проверены и оценены. Так как обучение производится дистанционно и нужно проверять достаточно большое количество различных работ учащихся, образовательному учреждению просто необходимо использовать системы автоматической проверки работ.

В рамках данной темы будет рассматриваться автоматическая проверка задач по программированию при дистанционном обучении в Тольяттинском государственном университете.

Объектом исследования являются методы и технологии автоматической проверки заданий при онлайн обучении.

Предметом исследования является процесс автоматической проверки задач по программированию при онлайн обучении.

Целью данного исследования является создание модели процесса автоматической проверки задач по программированию при онлайн обучении.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить критерии оценки работы студента;
- выявить основные положения, концепции системы автоматической проверки задач по программированию при онлайн обучении;
- изучить методы и технологии проверки исходного кода программы студента;
- изучить методы и технологии проверки работы программы студента и выдаваемых ею результатов;

– создать модель процесса автоматической проверки задач по программированию при онлайн обучении.

Для достижения поставленной цели, исследование будет проводится методом абстрагирования.

Научная новизна исследования определяется поставленной научной целью и задачами, имеющими теоретическую и практическую значимость, и выражается в расширении научных представлений о статическом анализе исходного кода и автоматической оценке программного решения. Новизна заключается в сделанных на основании проведенного исследования научных предложениях и выводах, в том числе: выделены критерии оценки программного решения; обоснован выбор методов и способов, решающих проблемы автоматической проверки программных решений; разработаны алгоритмы проверки исходного кода программного решения.

На защиту выносятся следующие основные положения, являющиеся новыми или содержащие элементы новизны:

– разработана модель процесса автоматической проверки задач по программированию;
– для определения процента оригинальности исходного кода в процесс автоматической проверки был внедрён метод разбиения на токены;
– внедрено использование виртуальных машин для безопасной проверки работы программного решения, а также дополнен алгоритм его тестирования;
– разработан алгоритм проверки стиля исходного кода, использующий авторскую систему штрафных баллов.

Теоретическая значимость работы заключается в том, что полученные результаты оказали влияние на область статического анализа исходного кода программ.

Практическая значимость – итоги исследования могут быть применены для создания системы автоматической проверки программных решений при онлайн обучении.

Основные теоретические положения, выводы и научно-практические рекомендации, которые изложены в диссертации, получили отражение в 2 статьях.

Результаты исследования были обсуждены на конференциях:

– Международная научно-техническая конференция «Перспективные информационные технологии» (ПИТ-2020);
– VI Международная научно-практическая конференция (школа-семинар) молодых ученых «Прикладная математика и информатика: современные исследования в области естественных и технических наук»;
– IV Всероссийская научная конференция с международным участием «Информационные технологии в моделировании и управлении: подходы, методы, решения»;
– VII Международная научно-практическая конференция (школа-семинар) молодых ученых «Прикладная математика и информатика: современные исследования в области естественных и технических наук».

Диссертационное исследование состоит из введения, 4 разделов, заключения и библиографии (30 наименований). Работа изложена на 74 страницах, содержит 23 рисунка и 15 таблиц.

1 Анализ возможности автоматической проверки задач по программированию

1.1 Основные критерии оценки программных решений студентов

Система автоматической проверки задач по программированию – это система, которая оценивает программные решения учащихся. Как же должен выглядеть процесс автоматической проверки в подобных системах? Прежде чем ответить на данный вопрос, сначала выделим основные критерии, на основании которых необходимо производить оценку программы учащегося (смотреть таблицу 1).

Таблица 1 – Критерии оценки программы учащегося

Критерий	Описание
Корректность исходного кода программы	Это некий балл, который показывает, на сколько исходный код программы соответствует требованиям, указанным в задании. Довольно часто присланная на проверку программа может выдавать правильные выходные данные, но при этом иметь исходный код, несоответствующий заданию. Значение данного критерия зависит также от результатов компиляции (или проверки интерпретатором) исходного кода программы (время компиляции, наличие предупреждений и т. д.).
Безопасность программы	Это показатель, который может принимать значения «0», если программа представляет опасность для системы автоматической проверки, или «1», если присланная программа безопасна. Под безопасностью программы понимается безопасность её исходного кода и процесса выполнения. Иногда умышленно или нет исходный код программы может содержать опасные вставки, способные нарушить работу системы или даже полностью её остановить. Так, например, исходный код может содержать ассемблерные включения, которые способны вызывать потенциально опасные функции для работы с системой на низком уровне. Также опасной может быть директива процессору, которая разворачивает код в бесконечную последовательность, из-за чего компилятор может попросту зависнуть [30]. Допустим, что в исходном коде нет опасных вставок, однако вывести из строя систему автоматической проверки могут даже разрешённые команды. Например, умышленно или нет студент может прислать программу, которая во время выполнения войдёт в бесконечный цикл и зависнет или же, работая с файловой системой, удалит важные для системы файлы и каталоги.

Продолжение таблицы 1

Критерий	Описание
Оригинальность исходного кода программы	Это процентный показатель, демонстрирующий объем исходного кода программы, написанного самостоятельно. Не исключено, что некоторые студенты (например, те, которые плохо усваивают материал), чтобы выполнить задание, будут использовать в качестве своего решения чужой исходный код. Однако исходный код, как и реферат, школьное сочинение и т. п. каждый человек пишет по-своему [24].
Корректность работы программы	Это некий балл, который показывает, на сколько правильные данные выдаёт программа. Также на этот балл оказывает влияние количество ресурсов, которые использует программа во время выполнения.

Таким образом в данном подразделе были выделены 4 основных критерия оценки программных решений учащихся.

1.2 Используемые методы проверки программных решений студентов

Теперь рассмотрим, как в соответствии с каждым критерием работают современные системы автоматической проверки. Начнём с оценки корректности исходного кода. В системе Ejudge она осуществляется следующим образом. Вначале системой запускается проверяющая подпрограмма для стиля оформления исходного кода. Проверяющая программа для стиля оформления исходного кода получает в качестве аргумента командной строки имя файла с исходным текстом программы. Проверяющая программа может выводить диагностику и на стандартный вывод, и на стандартный поток ошибок. Проверяющая программа должна завершиться с кодом завершения 0 в случае успешной проверки, либо с кодом завершения 1 или 4 в случае нарушения ограничений стиля программы [6].

Проверяющая программа задается с помощью конфигурационной переменной style_checker_cmd раздела описания задачи, либо с помощью

конфигурационной переменной style_checker_cmd раздела описания языкового процессора [11].

Переменные окружения для проверяющей программы задаются с помощью конфигурационной переменной style_checker_env раздела описания задачи, либо с помощью конфигурационной переменной style_checker_env раздела описания языкового процессора [4].

Программа выполняет некоторые простые проверки текста программы на Си-подобных языках:

- длина строки текста не превосходит установленного ограничения;
- в тексте программы не встречаются байты 0-31 и 127 кроме байтов \r, \n, \t;
- (если задана соответствующая опция) в тексте программы не используется байт \t;
- размер отступа в начале каждой строки кратен установленному значению.

Параметры программы задаются с помощью переменных окружения, перечисленных ниже:

- EJ_MAX_LINE_LENGTH=LENGTH (максимальная длина строки в байтах, по умолчанию равная 120);
- EJ_DISABLE_TABS=BOOL (запретить/разрешить использование TAB, по умолчанию равно 1, т. е. запрещено);
- EJ_BASE_INDENT=INDENT (размер отступа, по умолчанию равен 4).

После проверки стиля исходного кода происходит компиляция программы (или проверка интерпретатором). Если программа не скомпилировалась (или интерпретатор сообщает об ошибках), то такая программа оценивается минимальным количеством баллов, и проверка прекращается [26].

Что касается остальных систем автоматической проверки, то они не уделяют должного внимания проверке исходного кода программы и

ограничиваются лишь результатами его компиляции (или проверки интерпретатором).

При проверке оригинальности исходного кода системы Ejudge и DOMjudge используют стандартное средство diff для простого сравнения содержимого двух фалов. В этих системах проверка оригинальности исходного кода не входит в итоговую оценку и является опциональной функцией [18].

В системе T-BMSTU не используют какое-то конкретное средство для проверки оригинальности исходного кода. Вместо этого сервер обнаружения некорректных заимствований выполняет интеграцию системы T-BMSTU с внешним программным обеспечением, которое ведет поиск некорректных заимствований. Сервер работает на том же компьютере, что и web-сервер системы, и реализован в виде скрипта на языке Ruby. Он запускается по расписанию и осуществляет выемку решений из базы данных. Если с момента последнего запуска сервера в базе данных появились новые решения некоторой задачи, то все решения этой задачи передаются на вход установленным в системе программам поиска некорректных зависимостей. Отчеты программ поиска некорректных зависимостей сохраняются в каталоге файловой системы, который синхронизирован через Dropbox с компьютером преподавателя, проверяющего решения [29].

В настоящее время сервер обнаружения некорректных зависимостей интегрирован с сервисом MOSS, предоставляемым Стэндфордским университетом, и с программой обнаружения плагиата Platypus [27].

Оценку корректности работы программы все системы автоматической проверки осуществляют при помощи набора тестов (смотреть рисунок 1). Входные данные и эталонный ответ теста могут быть представлены текстом или файлом. Процесс формирования тестов возможен двумя способами:

– с помощью программы динамического генерирования тестовых данных. Используя описанный администратором алгоритм, такая

программа автоматически генерирует входные и выходные данные в соответствии с условиями задания;

– вручную. В данном случае администратор самостоятельно создаёт набор тестов.

Вне зависимости от способа формирования тестов, необходимо чтобы полученные тесты покрывали все возможные пути выполнения тестируемой программы [8].

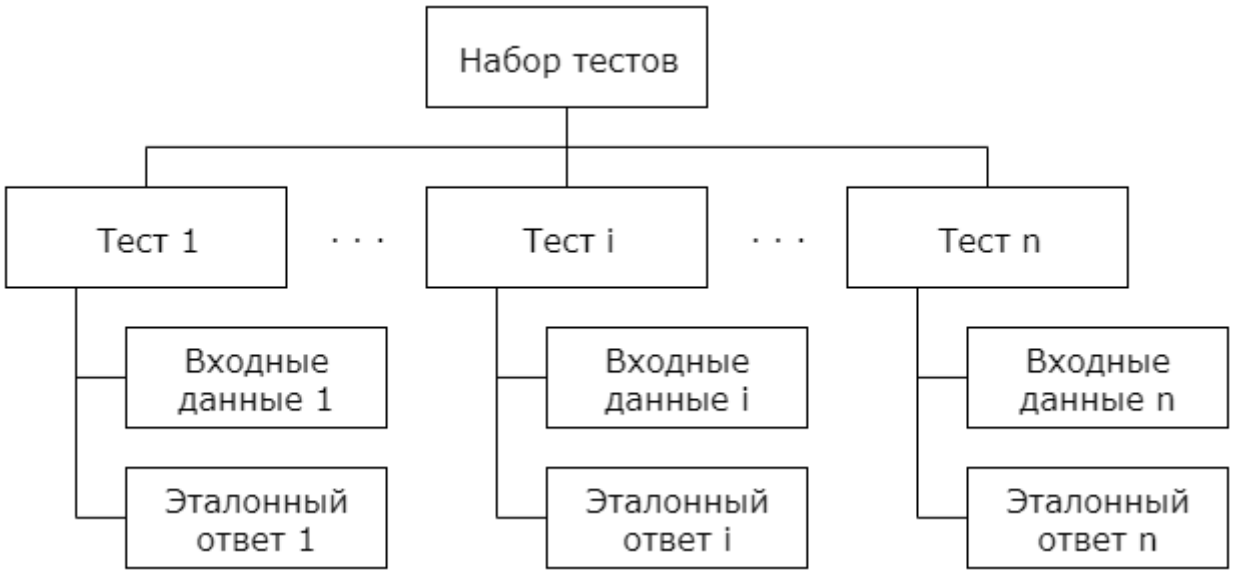

Рисунок 1 – Набор тестов

Само тестирование осуществляется по алгоритму, изображённому на рисунке 2. Сравнение выходных данных программы с эталонным ответом теста в основном представляет собой посимвольное сравнение двух строк (с пробелами). В конце тестирования программы подводятся итоги: определяется, какие тесты программа прошла, а какие нет и в зависимости от этого учащемуся начисляется определённое количество баллов.

Что касается безопасности, то большинство современных систем автоматической проверки используют интегрированные системы изолированного запуска приложений и контроля над ними, которые обычно называют «песочницами». Песочница – это некий механизм для безопасного

исполнения программ, который позволяет контролировать набор ресурсов, предназначенный для исполнения программы, а также доступ к определённым элементам системы. Например, с помощью песочницы можно сделать так, чтобы программа, присланная на проверку, имела определённое количество доступной оперативной памяти, могла работать только с каталогом, из которого была запущена и не имела доступа в интернет [25].

Рисунок 2 – Общий алгоритм тестирования программы

Таким образом в данном подразделе были рассмотрены методы, которые используют современные системы, для автоматической проверки программных решений студентов.

1.3 Постановка задачи на исследование реализации автоматической проверки программных решений

Обобщим результаты проведённого ранее исследования. В таблице 2 представлены результаты анализа существующих систем автоматической проверки программных решений в соответствии с выделенными критериями.

Таблица 2 – Результаты анализа современных систем автоматической проверки

Название системы	Корректность исходного кода программы	Оригинальность исходного кода программы	Корректность работы программы	Безопасность программы
Ejudge	используется инструмент, проверяющий стиль кода; в остальном код проверяется посредством его компиляции (или интерпретатором)	простое сравнение содержимого двух файлов (diff)	проверка правильности работы программы осуществляется при помощи заранее подготовленных тестов; контролируется время работы и количество использованной памяти	используются интегрированные системы изолированного запуска приложений и контроля над ними (песочницы)
DOMjudge	код проверяется посредством его компиляции (или интерпретатором)	простое сравнение содержимого двух файлов (diff)	проверка правильности работы программы осуществляется при помощи заранее подготовленных тестов	устанавливается минимальное окружение с помощью chroot и посредством cgroup ограничиваются ресурсы (средства ОС Linux)

Продолжение таблицы 2

Название системы	Корректность исходного кода программы	Оригинальность исходного кода программы	Корректность работы программы	Безопасность программы
PC^2	код проверяется посредством его компиляции (или интерпретатором)	отсутствует	проверка правильности работы программы осуществляется при помощи заранее подготовленных тестов	отсутствует
Contester	код проверяется посредством его компиляции (или интерпретатором)	отсутствует	проверка правильности работы программы осуществляется при помощи заранее подготовленных тестов; контролируется время работы и количество использованной памяти	контролируются запрещённые библиотеки и подключаемые модули; программа выполняется в защищенном изолированном окружении
T-BMSTU	код проверяется посредством его компиляции (или интерпретатором)	используется сервер обнаружения некорректных заимствований, который выполняет интеграцию системы с внешним программным обеспечением, которое ведет поиск некорректных заимствований	проверка правильности работы программы осуществляется при помощи заранее подготовленных тестов; контролируется время работы и количество использованной памяти	выполняется в защищенном изолированном окружении; запущенное решение не имеет доступа ни к сетевым интерфейсам, ни к файловой системе сервера; решение, написанное на небезопасном с точки зрения работы с памятью языке (например, C, C++ и Pascal), дополнительно запускается в отладчике valgrind

Из таблицы 2 видно, что для оценки программы, существующие системы выполняют практически одинаковые действия. Таким образом общий алгоритм работы систем автоматической проверки можно представить в виде блок-схемы, изображённой на рисунке 3. Вначале исходный код студента проходит предварительную проверку. Если исходный код не удовлетворяет требованиям системы, то проверка прекращается, и решение студента оценивается минимальным количеством баллов. В противном случае система подготавливает защищённое изолированное окружение, чтобы безопасно скомпилировать (или проверить интерпретатором) исходный код решения студента и протестировать полученную программу. В конце на основании предыдущих этапов формируются результаты проверки решения [28].

Рассмотрим общую архитектуру системы автоматической проверки (смотреть рисунок 4). Из диаграммы компонентов видно, что система имеет несложную клиент-серверную архитектуру. С помощью клиентского приложения студенты и преподаватели взаимодействуют с системой (студент осуществляет отправку исходного кода своей программы, преподаватель просматривает результаты, добавляет новые задания и т. д.). В системе автоматической проверки модуль управления отвечает за работу с пользователями и контролирует процессы тестирования. Модуль тестирования осуществляет проверку решений студентов и отклоняет неправильные решения. Для обеспечения эффективной проверки обычно работает несколько модулей тестирования.

Таким образом на основе анализа, проведённого выше, и рассмотренных критериев оценки программных решений можно выделить следующие актуальные задачи, которые должна решать автоматическая проверка программных решений:

– оценивать исходный код программы. Опираясь на требования, указанные в задаче, необходимо проверять то, насколько грамотно написано решение студента;

– проверять оригинальность исходного кода. Необходимо определять, студент сам написал код или скопировал его у своих одногруппников;
– обеспечить безопасность проверки программного решения студента. Необходимо чтобы проверяемый код не мог исказить результаты проверки или нарушить работу системы автоматической оценки;
– оценивать корректность работы программы. Необходимо проверять, работает ли программа согласно требованиям, представленным в задаче.

Рисунок 3 – Общий алгоритм работы системы автоматической проверки

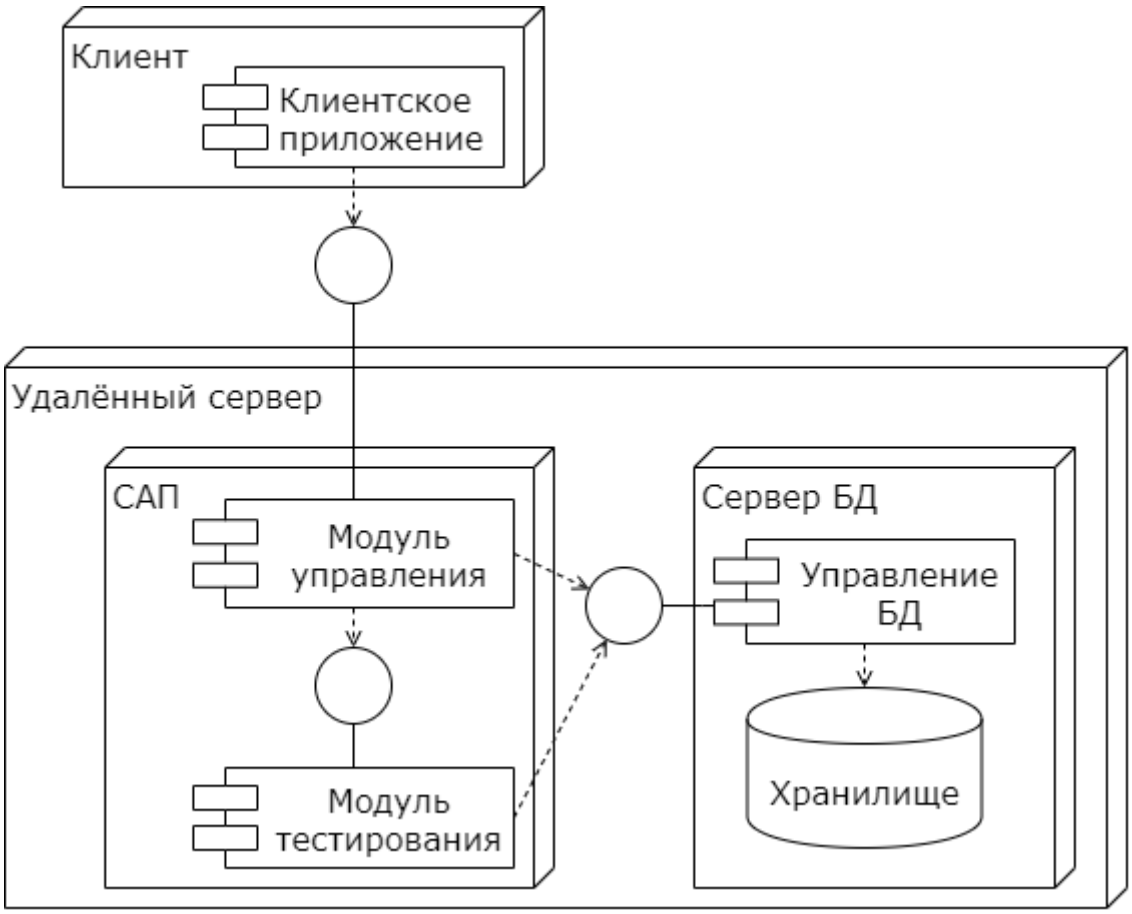

Рисунок 4 – Диаграмма компонентов

В данном подразделе было сделано обобщение, проведённого ранее исследования работы существующих систем автоматической проверки программных решений, составлен общий алгоритм их работы, рассмотрена общая архитектура проверяющих систем, а также сформулированы задачи, которые они должны решать.

В первом разделе были рассмотрены критерии оценки программного решения студента, и как в соответствии с ними работают современные системы автоматической проверки. Сделано обобщение проведённого исследования и сформулированы задачи, которые должна решать система автоматической проверки программных решений.

2 Анализ существующих решений проблем автоматической проверки задач по программированию

2.1 Способы проверки корректности исходного кода программы

В соответствии с задачами, поставленными выше, проведём анализ способов проверки исходного кода (смотреть таблицу 3).

Таблица 3 – Способы проверки исходного кода

Название способа	Описание способа
Компиляция (проверка интерпретатором)	Это основной способ, позволяющий проверить корректность исходного кода программы. Основываясь на результатах компиляции (или проверки интерпретатором) исходного кода программы, можно сразу определить, стоит ли проверять программу дальше. Так если компилятор (или интерпретатор) обнаружит синтаксическую ошибку в исходном коде программы, то дальше можно уже не проводить никаких проверок, поскольку такую программу просто невозможно запустить. Помимо выявления синтаксических ошибок компилятор (или интерпретатор) способен выдавать предупреждения. Предупреждения – это сообщения о подозрительных местах в программе. Их можно проигнорировать и продолжить проверять программу. Однако впоследствии эти предупреждения могут оказаться программными ошибками, способными нарушить нормальную работу программы. Поэтому предупреждения компилятора (или интерпретатора) тоже стоит учитывать при оценке корректности исходного кода программы данным способом.
Проверка стиля	Данный способ не поможет выявить сложные ошибки в исходном коде программы, однако он позволит взглянуть на код с точки зрения его оформления. В реальных проектах очень часто с одним и тем же исходным кодом работает сразу несколько человек. Поэтому очень важно чтобы программист придерживался единого стиля. Это облегчит чтение, поддержку и слияние кода для остальных программистов. Также к этому способу стоит отнести проверку размера файлов с исходным кодом. Это позволит отсеять слишком большие решения. Так, например, ученик может просчитать очень трудоёмкое решение на своём компьютере и поместить результаты в один большой словарь в исходном коде. А само решение будет выглядеть как read(a), write(result[a]). Таким образом, проверив размер исходного кода, можно будет сразу отклонить подобные решения [19].

Компиляция (проверка интерпретатором) – это обязательный этап любой автоматической оценки программного решения. Без успешной компиляции исходного кода, остальные его проверки просто теряют смысл, поскольку такой код невозможно выполнить. Для выставления программному решению более точной оценки необходимо анализировать вывод компилятора (интерпретатора), учитывать все предупреждения, которые он выдаёт. Однако все компиляторы выводят предупреждения и ошибки в разных форматах. Это существенно затрудняет автоматический анализ результатов компиляции (проверки интерпретатором) [3].

Очевидно, что успешной компиляции исходного кода недостаточно, чтобы считать его полностью корректным. Код программы должен быть читаемым и эффективным. Для этого должна осуществляться проверка его стиля (смотреть рисунок 5).

Рисунок 5 – Общий алгоритм проверки стиля исходного кода

Анализируя код на данном этапе, можно найти серьёзные ошибки в программном решении студента. Например, это может быть использование в

коде потенциально опасных функций или запрещённых условием задания паттернов. В таком случае можно сразу прекратить дальнейшую проверку решения. Это сэкономит время и вычислительные ресурсы. Ошибки в оформлении кода (например, большое количество строк в одном функциональном блоке, неполные операторы и т. п.) также должны влиять на итоговую оценку [10]. Ниже в таблице 4 приведены ещё примеры ошибок, которые необходимо выявлять на данном этапе.

Таблица 4 – Примеры дефектов исходного кода

Дефект	Признаки наличия	Способ обнаружения
Ошибки обработчика (Handler Errors)	Выброс исключения, у которого нет обработчика.	Создание списка всех выбросов исключений и списка всех обработчиков. Сравнение двух списков.
Присваивание вместо сравнения (Assigning instead of Comparing)	Наличие «=» вместо «==» в блоке if, где сравниваются лишь переменные.	Проверка заданного списка регулярных выражений для содержимого заданных типов блоков.
Неверная инициализация (Improper Initialization)	Отсутствие инициализации значений объектов (например, строк) перед их использованием.	Поиск объявлений объектов, которые требуют инициализации перед своим использованием (например, статический массив строки). Поиск первого использования объекта в вызовах и операциях (например, правая часть в присваивании; функции, использующие их в качестве входных значений), при отсутствии их инициализации (левая часть в присваивании, функции, использующие их в виде выходных значений).
Остаточный отладочный код (Leftover Debug Code)	Наличие отладочного кода.	Поиск присутствия вызовов функций из списка отладочных. Поиск подозрительных правил в названия отладочных функций (ключевые слова по debug).
Разыменование нулевых указателей (NULL Pointer Dereference)	Вызов операций по освобождению памяти для указателей, которые равны NULL.	Поиск вызовов функций, возвращающих указатель (на которую может повлиять пользователь), при отсутствии проверки ее значения следом за этим.

Продолжение таблицы 4

Дефект	Признаки наличия	Способ обнаружения
Нереализованная или неподдерживаемая функция в GUI (Unimplemented of unsupported feature in UI)	Наличие скрытых элементов GUI, а также некорректностей в обработчиках событий.	Поиск наличия «disabled=true», «hidden=true» и других подобных свойств в файле GUI. Поиск отсутствия кода в функции-обработчике.

В данном подразделе были рассмотрены и проанализированы два способа проверки корректности исходного кода программы. Было выяснено, что необходимо комбинировать два этих способа, чтобы грамотно оценить исходный код присланной студентом программы.

2.2 Методы проверки оригинальности исходного кода программы

Теперь проанализируем методы проверки оригинальности исходного кода (смотреть таблицу 5), которые позволяют выявить сходства между двумя решениями.

Таблица 5 – Методы проверки оригинальности исходного кода

Название метода	Описание метода
Простое сравнение двух строк	При данном методе исходный код программы представляется в виде одной символьной строки. Далее в основном решается задача нахождения наибольшей общей подпоследовательности. В результате становится видно: какие части исходного кода совпадают, а какие – нет. Данный метод позволяет определить заимствования вне зависимости от языка программирования, на котором был написан исходный код проверяемой программы. Также метод довольно просто реализовать. Однако метод не учитывает структуру и синтаксические особенности исходного кода программы. Следовательно, данный метод можно легко обойти. Например, студенту достаточно всего лишь переименовать переменные, функции, классы и т. п., и метод уже не сможет достаточно корректно выявить плагиат.

Продолжение таблицы 5

Название метода	Описание метода
Использование метрик	Данный метод позволяет представить исходный код программы в виде точки, находящейся в n-мерном пространстве, где каждая координата этой точки соответствует определённой характеристике исходного кода программы. Затем находится расстояние между такими точками. Соответственно можно считать, что чем ближе такие точки расположены, тем больше заимствований в сравниваемых программах. В отличие от простого сравнения строк, данный метод студент уже не сможет обойти посредством простых переименований. Если в целом достаточно грамотно выбрать измеряемые характеристики исходного кода, то можно будет довольно эффективно выявлять плагиат. Однако данный метод имеет свои недостатки: он в некоторой степени зависит от языка программирования, на котором написан исходный код проверяемой программы, а также достаточно часто ложно срабатывает (особенно при проверки небольших программ) и, наоборот, не всегда срабатывает в случаях, когда студент позаимствовал только часть кода [12].
Разбиение на токены	При данном методе каждому типу элементов исходного кода программы присваивается свой уникальный идентификатор, который называется токеном. Затем каждый конкретный элемент исходного кода заменяется на соответствующий его типу идентификатор. Из полученных токенов строится последовательность с сохранением порядка их следования в исходном коде программы. Далее полученные последовательности сравнивают между собой. В зависимости от количества совпадающих фрагментов определяют, как много исходного кода студент позаимствовал из чужого решения. Как и предыдущий метод, разбиение на токены игнорирует несущественные характеристики исходного кода программы. Однако данный метод, в отличие от использования метрик, не делает ошибок в случаях, когда была позаимствована только часть исходного кода. Даже перестановка позаимствованной части не сможет обмануть данный метод. Из недостатков можно выделить следующие: метод всё также зависит от конкретного языка программирования, на котором была написана программа, и может ложно срабатывать при проверки небольших программ [22].

Продолжение таблицы 5

Название метода	Описание метода
Построение синтаксического дерева	Данный метод предполагает построение абстрактного синтаксического дерева на основе исходного кода. Полученное дерево отображает логику проверяемой программы. Для выявления заимствований осуществляется поиск совпадающих поддеревьев. Как и два предыдущих метода, синтаксическое дерево позволяет уйти от рассмотрения несущественных характеристик исходного кода программы. К плюсам данного метода можно также отнести то, что процедура сравнения синтаксических деревьев сведена к анализу числовых матриц, что в свою очередь не требует больших вычислительных мощностей. Из минусов стоит отметить то, что метод может оказаться недостаточно эффективным при поиске намеренных заимствований в случае их видоизменений, например, при перестановке строк или операндов местами. Также данный метод зависит от языка программирования, на котором был написан исходный код проверяемой программы, причём круг языков программирования, с которыми может работать данный метод, ограничен. Помимо этого, возможны ложные срабатывания при проверки небольших программ [20].

При проверке исходного кода на плагиат под простым сравнением двух строк в основном подразумевается использование утилиты сравнения файлов diff. Эта программа выводит построчно различия между двумя файлами (смотреть рисунок 6). Данный метод не зависит от языка программирования, однако его легко обмануть путём переименования переменных или функций. Поэтому простое сравнение строк больше подойдёт для ручной проверки оригинальности, чем автоматической.

Метод, основанный на использовании метрик, тоже не сможет должным образом обнаружить плагиат. Он выдаст некорректные результаты, если студент полностью скопирует чужое решение и добавит к нему свой код. Причиной этому служит существенное различие между замерами характеристик исходного кода (например, количество циклов, функций, переменных, условий в решениях будет отличаться). Следовательно, такое решение будет считаться достаточно оригинальным, поскольку расстояние

между точками, в виде которых представляются решения, будет большое. Очевидно, что такой результат проверки оригинальности неверный. Также данный метод может посчитать плагиатом решение, которое, например, имеет одинаковое количество циклов вместе с другим решением. Однако на самом деле эти решения могут быть оригинальны, поскольку одни и те же циклы можно реализовать разными способами.

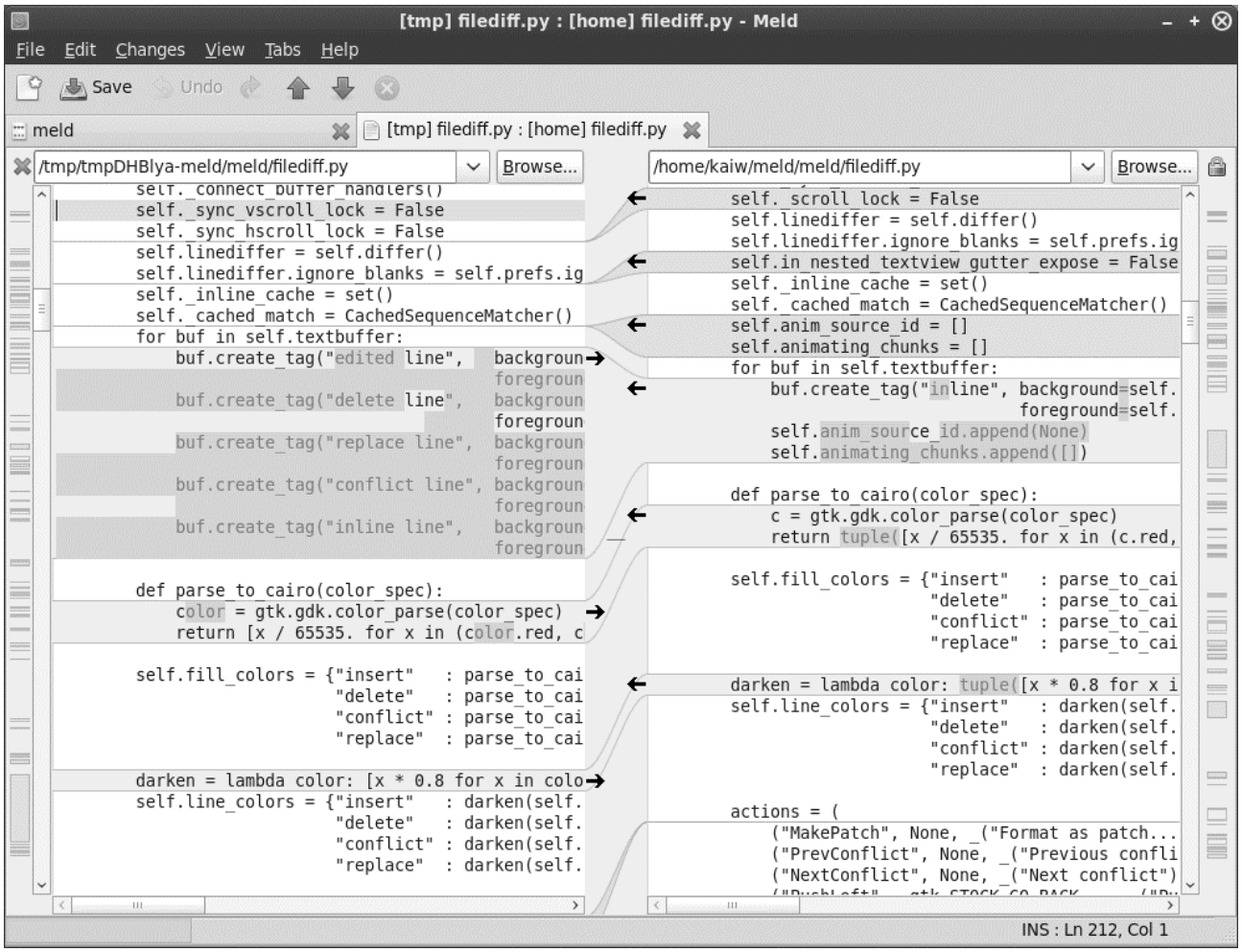

Рисунок 6 – Пример работы утилиты diff

При разбиении на токены, весь исходный код двух решений преобразуется в две последовательности идентификаторов в соответствии с типом каждого элемента. Затем применяется один из алгоритмов строкового сравнения для поиска совпадающих фрагментов кода. Например, алгоритм W-

shingling, разработанный для поиска копий и дубликатов рассматриваемого текста в веб-документе, работает следующим образом:

- выполняется канонизация текста. Оригинальный текст приводится к единой нормальной форме. Текст очищается от предлогов, союзов, знаков препинания, HTML тегов, и прочих лишних элементов, которые не должны участвовать в сравнении. В большинстве случаев также предлагается удалять из текста прилагательные, так как они не несут смысловой нагрузки. Также на этапе канонизации текста можно приводить существительные к именительному падежу, единственному числу, либо оставлять от них только корни [7];
- далее текст разбивается на шинглы, где шингл – это выделенная из текста подпоследовательность, состоящая из двух или более слов. Выборка происходит внахлест с определённым заранее шагом;
- производится вычисление хэшей шинглов: для каждого шингла из составленного ранее набора рассчитывается значение контрольной суммы через разные функции (например, SHA1, MD5, CRC32 и т. п.);
- полученные наборы контрольных сумм сравниваются друг с другом: ищутся значения, которые присутствуют в двух наборах одновременно. В конце выдаётся процент совпадений [9].

Другим примером может служить алгоритм Вагнера-Фишера. С помощью данного алгоритма вычисляется кратчайшее расстояние Левенштейна, которое отражает сходство между двумя строками. Похожесть исходной и целевой строки измеряется как количество замен, вставки и удаления, необходимых для преобразования одной строки в другую. Расстояние Левенштейна увеличивается в зависимости от количества преобразований, необходимых для превращения одной строки в другую. Если исходная строка – «лом», а целевая – «ком», то расстояние Левенштейна будет равно 1. Это означает, что необходимо сделать одну замену, чтобы преобразовать одну строку в другую [15].

Метод разбиения на токены частично зависит от языка программирования и может выдавать не очень корректные результаты при проверке небольших решений, однако он лишён всех остальных недостатков других методов.

Последний метод предполагает построение синтаксического дерева (смотреть рисунок 7). Для этого необходим алгоритм, который будет анализировать и преобразовывать код подобно компилятору или интерпретатору. Следовательно, данный метод очень сильно зависит от языка программирования, поскольку нужно большое количество отдельных алгоритмов, которые будут строить синтаксическое дерево, что довольно неудобно и неэффективно.

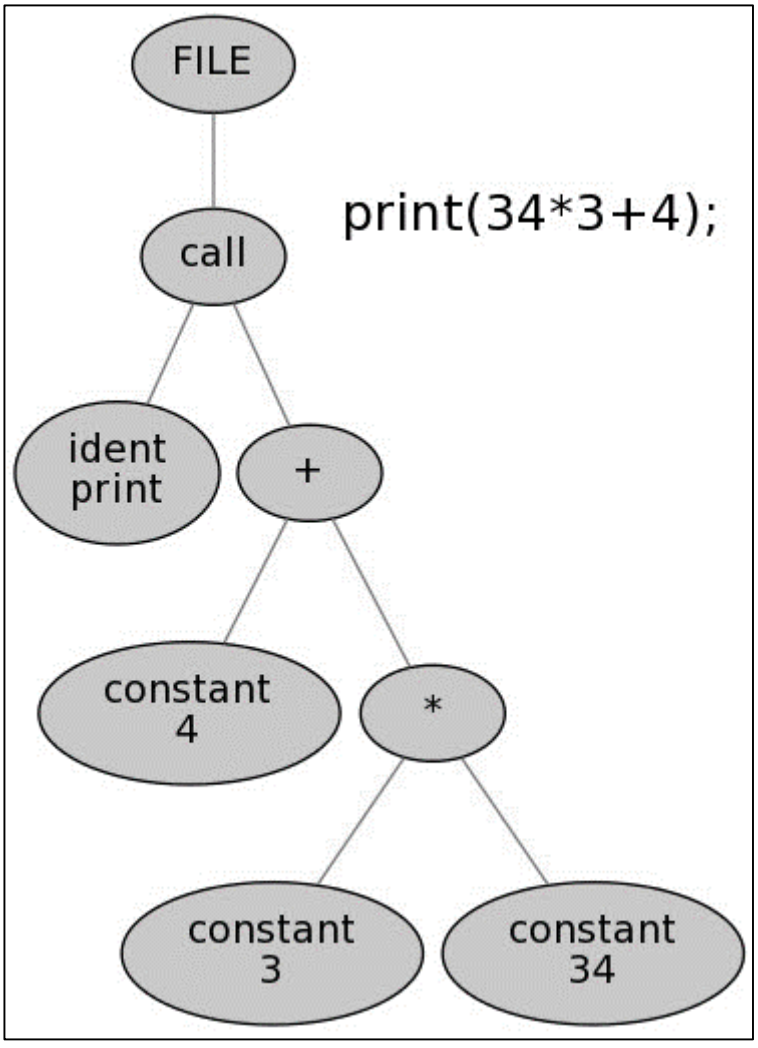

Рисунок 7 – Пример синтаксического дерева

Таким образом, можно сделать вывод о том, что самым эффективным методом автоматической проверки оригинальности исходного кода решений является метод разбиения на токены, поскольку он меньше всех зависит от конкретного языка программирования, совершает значительно меньше ошибок и выявляет типичные попытки плагиата.

В данном подразделе были рассмотрены и проанализированы методы проверки оригинальности исходного кода. В результате анализа был выбран метод разбиения на токены

2.3 Способы проверки корректности работы программы

В настоящее время для того чтобы проверить работу готовой программы, используют заранее подготовленный набор тестов (смотреть рисунок 1). Ниже в таблице 6 представлены способы, которые проверяют корректность работы программы, используя подобный набор тестов.

Таблица 6 – Способы проверки корректности работы программы

Название способа	Описание способа
Использование метрик	При данном способе для сравнения выходных данных программы с эталонным ответом используются метрики. Суть использования метрик описана в подразделе 2.2. В данном случае оценивается расстояние между выводом программы и эталонным ответом. Чем меньше расстояние, тем выше оценка. Таким образом, повышается точность оценивания. Однако данный способ имеет существенный недостаток. Использование метрик зависит от формата ответа. Чтобы выделить и измерить характеристики, необходимо знать, в каком виде представляется ответ. Поскольку каждое задание требует выводить данные по-своему, приходится каждый раз заново анализировать формата ответа.
Измерение потребляемых ресурсов	Данный способ не проверяет правильность данных выдаваемых программой при каждом тесте, однако он позволяет проверить программу с точки зрения её эффективности. Для этого производятся замеры времени её выполнения и количества потребляемой памяти в соответствии с входными данными, описанными в тесте. Если программа выполняется достаточно долго и/или потребляет слишком много памяти, то оценка за текущий тест снижается.

Продолжение таблицы 6

Название способа	Описание способа
Простое сравнение	Данный способ был приведён ранее в качестве примера (смотреть рисунки 1 и 2). Его суть заключается в сравнении выходных данных программы с эталонным ответом теста. Иначе говоря, производится посимвольное сравнение двух строк (с пробелами), и, если строки не совпадают, то считается, что текущий тест программа не прошла, осуществляется переход к следующему тесту. Таким образом, возможны только две оценки при прохождении каждого теста: зачтено или не зачтено. В конце тестирования анализируют количество пройденных и заваленных тестов. Данный способ имеет два недостатка. Нет промежуточных оценок, из-за чего падает точность оценивания. Для того чтобы покрыть все возможные пути выполнения тестируемой программы, необходимо создавать огромное количество тестов. Например, программа раскладывает число 10 на множители и выводит строку «5, 2». Если был создан только один тест, в котором идёт сравнение со строкой «2, 5», то такая программа будет считаться неправильной, несмотря на то, что ответ «5, 2» тоже правильный. Именно из-за таких ситуаций и возникает необходимость в создании огромного количества тестов под каждую вариацию одного ответа.
Использование регулярных выражений	При данном способе выходные данные программы сопоставляются с регулярным выражением. Регулярные выражения представляют собой сильный инструмент для поиска строк, проверки их на соответствие какому-либо шаблону и другой подобной работы. Строго говоря, регулярные выражения – это специальный язык для описания шаблонов строк. Таким образом, если вывод программы совпадает с описанным в тесте шаблоном, то считается, что программа прошла текущий тест, осуществляется переход к следующему тесту. Как и в случае с первым способом, возможны только две оценки при прохождении каждого теста: зачтено или не зачтено. Соответственно данный способ имеет тот же недостаток – меньшая точность оценивания. Также стоит отметить, что использование регулярных выражений требует знания достаточно сложных правил их составления. Однако данный способ имеет одно важное преимущество: он позволяет избежать создания огромного количества тестов. Например, программа в качестве ответа выводит слово «минута». Пусть слова «Минута» и «минуты» тоже являются правильными ответами. Тогда чтобы охватить все три варианта одного и того же ответа, достаточно создать только один тест с регулярным выражением «[Мм]инут[аы]».

Из таблицы видно, что использовать метрики при проверке вывода программы довольно непросто: необходимо всегда подстраиваться под формат вывода программы. Поэтому данный способ неэффективен. Если использовать регулярные выражения, то можно значительно уменьшить количество тестов в наборе. Однако далеко не все преподаватели знают

правила составления регулярных выражений. В добавок придумывание регулярного выражения отнимает достаточно времени. Недостатки двух этих способов довольно серьёзны, а их достоинства не сильно выделяются, поскольку рассматривается проверка студенческих работ, которые в большинстве своём довольно небольшие и простые. Поэтому простого посимвольного сравнения вывода программы с эталонным ответом будет достаточно. А для получения более точной оценки программы желательно проводить ещё и замер потребляемых ресурсов. При таком подходе, тесты довольно просто составлять, а также снижается вычислительная нагрузка.

В данном подразделе были рассмотрены и проанализированы способы проверки корректности работы программы. Было принято решение о том, что для проверки достаточно просто сравнивать вывод программы с эталонным ответом, параллельно замеряя время её выполнения и потребляемую память.

2.4 Способы безопасного выполнения программы

Рассмотрим способы безопасного выполнения программы (смотреть таблицу 7), которые позволяют предотвратить негативные воздействия на процесс оценки программного решения.

Таблица 7 – Способы безопасного выполнения программы

Название способа	Описание способа
Выявление потенциально опасного кода	Данный способ поможет отклонить потенциально опасные программы. Его суть заключается в том, чтобы проверить исходный код программы на наличие опасных участков кода, которые в дальнейшем способны оказать негативное воздействие на компилятор, интерпретатор, среду выполнения и т. д. Таким образом, если в программе обнаруживается потенциально опасный код (например, ассемблерные вставки), то такая программа оценивается меньшим количеством баллов или вовсе снимается с проверки. Данный способ не требует больших вычислительных мощностей и позволяет выявить угрозы на начальных этапах проверки. Недостатки: достаточно большое количество опасностей невозможно обнаружить, сканируя только исходный код, способ зависит от конкретного языка программирования.

Продолжение таблицы 7

Название способа	Описание способа
Использование виртуальных машин	Виртуальная машина – это виртуальный компьютер со всеми виртуальными устройствами и виртуальным жёстким диском, на который и устанавливается новая независимая операционная система. Виртуальная машина полностью изолирована от реального компьютера, хотя и может иметь доступ к его диску и периферийным устройствам. Таким образом, благодаря этим особенностям, становится возможным тестирование любой программы без особой угрозы для системы автоматической проверки. Компиляция (проверка интерпретатором) и выполнение тестируемой программы будут проводится внутри виртуальной машины. Следовательно, любые изменения, которые могут возникнуть в ходе этих процессов, отразятся только на виртуальной машине. И в случае если виртуальной машине был нанесён какой-либо ущерб или она была полностью выведена из строя, то её можно создать заново. Однако процесс создания новой виртуальной машины требует достаточно немало времени. Да и в целом при использовании данного способа появляются дополнительные расходы на эмуляцию виртуального оборудования и запуск операционной системы, поддержка и администрирование необходимого окружения для работы проверяемой программы. Также при разворачивании большого количества виртуальных машин на сервере объем занимаемого ими места на жёстком диске будет только расти, т.к. для каждой из них требуется место, как минимум, для операционной системы и драйверов для виртуальных устройств.
Использование контейнеров	Контейнер – это изолированный приемник команд, предназначенный для запуска приложения в операционной системе сервера. Контейнеры реализуются поверх ядра операционной системы узла и содержат только приложения, некоторые API-интерфейсы и службы операционной системы, работающие в пользовательском режиме. Таким образом, при помощи контейнеров, как и в случае с виртуальными машинами, можно в изолированном окружении безопасно скомпилировать (проверить интерпретатором) и выполнить тестируемую программу. У данного способа есть важные преимущества. Контейнеры занимают меньше места, так как они, в отличие от виртуальных машин, переиспользуют большее количество общих ресурсов хост-системы. Кроме того, контейнер значительно легче разворачивать, масштабировать. Контейнер даёт более эффективный механизм инкапсуляции приложений, обеспечивая необходимые интерфейсы хост-системы. Данная возможность позволяет контейнерам разделить ядро системы, где каждый из контейнеров работает как отдельный процесс основной операционной системы, у которого есть своё собственное виртуальное адресное пространство, таким образом данные, принадлежащие разным областям памяти, не могут быть изменены. Однако у контейнеров есть недостаток: поскольку они предоставляют упрощенную изоляцию от узла и других контейнеров, отсутствует настолько надежная граница безопасности, как в случае виртуальных машин.

Продолжение таблицы 7

Название способа	Описание способа
Предварительное сканирование исполняемых файлов	Суть данного способа заключается в сканировании исполняемой программы с целью выявления в ней опасных сигнатур (уникальных последовательностей байтов). Если опасность была обнаружена, то такая программа оценивается меньшим количеством баллов или вовсе не допускается к последующей проверке. Данный способ тоже не требует больших вычислительных мощностей. Из недостатков можно отметить следующие: необходимость постоянного обновления баз данных известных сигнатур, неспособность обнаружить новые угрозы, невозможность выявления угроз, способных навредить процессу компиляции. Также стоит добавить, что данный способ, как и сканирование исходного кода, не поможет обнаружить большое количество немало важных угроз.

Из таблицы видно, что для эффективного выявления потенциально опасного кода необходимо постоянно обновлять базы данных, в которых описаны опасные фрагменты. Это же касается и предварительного сканирования исполняемых файлов. Таким образом неизвестные угрозы способны нанести ущерб системе автоматической проверки. Этого существенного недостатка лишены виртуальные машины и контейнеры. Для того чтобы сделать окончательный выбор, проведём детальное сравнение виртуальной машины и контейнера на примере Oracle VM VirtualBox и Docker (смотреть таблицу 8).

Таблица 8 – Сравнение виртуальной машины и контейнера

Критерий	Виртуальная машина	Контейнер
Изоляция	Обеспечивает полную изоляцию от операционной системы узла и других виртуальных машин. Это полезно, когда важна строгая граница безопасности, например, для разделения приложений от конкурирующих компаний на одном сервере или в кластере.	Обычно предоставляет упрощенную изоляцию от узла и других контейнеров, но не предоставляет настолько надежную границу безопасности, как в случае виртуальных машин.
Совместимость с гостевой системой	Работает практически с любой операционной системой в виртуальной машине.	Работает на той же версии операционной системы, что и узел.

Продолжение таблицы 8

Критерий	Виртуальная машина	Контейнер
Развертывание	Развертывание отдельных виртуальных машин с помощью центра администрирования Windows или диспетчера Hyper-V; развертывание нескольких виртуальных машин с помощью PowerShell или System Center Virtual Machine Manager.	Развертывание отдельных контейнеров с помощью Docker с использованием командной строки; развертывание нескольких контейнеров с помощью Orchestrator, например, службы Azure Kubernetes.
Операционная система	Содержит полноценную операционную систему, включая ядро, поэтому требует больше системных ресурсов (ЦП, памяти и хранилища).	Запускает часть операционной системы в пользовательском режиме и ее можно адаптировать, чтобы она содержала только необходимые службы для приложения, что позволит использовать меньше системных ресурсов.
Обновления и исправления для операционной системы	Загрузка и установка обновлений операционной системы на каждой виртуальной машине. Для установки новой версии операционной системы требуется обновить, а зачастую и создать полностью новую виртуальную машину. Это может занять много времени.	Обновление или исправление файлов операционной системы в контейнере выполняется точно так же, однако происходит это намного быстрее из-за простоты запуска нового контейнера.
Постоянное хранилище	Использует виртуальный жесткий диск (VHD) для локального хранилища для одной виртуальной машины или общий файловый ресурс SMB для совместно используемого несколькими серверами хранилища	Использует диски Azure для локального хранилища для одного узла или службы файлов Azure (общие ресурсы SMB) для совместно используемого несколькими узлами или серверами хранилища.
Балансировка нагрузки	Балансировка нагрузки виртуальной машины перемещает выполняющиеся виртуальные машины на другие серверы в отказоустойчивом кластере.	Сами контейнеры не перемещаются. Вместо этого Orchestrator может автоматически запускать или прекращать работу контейнеров на узлах кластера для управления изменениями нагрузки и доступности.

Продолжение таблицы 8

Критерий	Виртуальная машина	Контейнер
Отказоустойчивость	Виртуальные машины могут выполнить отработку отказа на другой сервер в кластере с перезапуском операционной системы виртуальной машины на новом сервере.	В случае сбоя узла кластера все контейнеры, работающие на нем, быстро пересоздаются Orchestrator на другом узле кластера.
Сеть	Использует виртуальные сетевые адаптеры.	Использует изолированное представление виртуального сетевого адаптера, предоставляя меньшую виртуализацию: брандмауэр узла используется контейнерами совместно – при этом используется меньше ресурсов.

Из таблицы видно, что контейнеры, в сравнении с виртуальными машинами, создаются и разворачиваются значительно быстрее и потребляют гораздо меньше ресурсов. Это очень важно, поскольку тестирование решений студентов будет проводиться довольно часто. Однако контейнер работает на той же версии операционной системы, что и узел. Это говорит о том, что, например, контейнеры Windows будут запускаться только на Windows, а контейнеры Linux только на Linux. Из-за этого станет невозможным одновременное выполнение двух программ, написанных под разные операционные системы. Также контейнеры, как уже писалось ранее, не предоставляют настолько надёжную границу безопасности. Таким образом, на основании проведённого анализа был сделан выбор в пользу виртуальных машин Oracle VM VirtualBox, поскольку безопасность и кроссплатформенность превыше всего [14].

2.5 Пути решения задач автоматической проверки программных решений

Таким образом, на основании проведённого анализа, были предложены следующие пути решения задач автоматической проверки программных решений:

– перед тем как скомпилировать (или проверить интерпретатором) присланную программу, необходимо сначала проверить размер файла с исходным кодом. Это делается для того, чтобы отсеять слишком большие решения. Так, например, ученик может просчитать очень трудоёмкое решение на своём компьютере и поместить результаты в один большой словарь в исходном коде. А само решение будет выглядеть как read(a), write(result[a]). Таким образом, проверив размер исходного кода, можно будет сразу отклонить подобные решения. Далее нужно проверить исходный код на наличие ассемблерных вставок. Это можно сделать с помощью программы noasm. Такая проверка позволит избежать компиляции и последующего выполнения потенциально опасного кода. Затем необходимо проверить стиль исходного кода. Такая проверка нужна по причине того, что в реальных проектах очень часто с одним и тем же исходным кодом работает сразу несколько человек. Поэтому очень важно чтобы программист придерживался единого стиля. Это облегчит чтение, поддержку и слияние кода для остальных программистов. В конце должны осуществляться проверки, которые непосредственно связаны с условиями задачи. Например, условиями задачи запрещено использовать функцию sin. Проверив исходный код можно будет определить, использовал ли студент данную функцию или нет [17];
– на плагиат имеет смысл проверять только достаточно большие программные решения (приблизительно от 20 килобайт), поскольку маленькую программу (например, реализацию алгоритма Base64)

довольно трудно сделать оригинальной. Для определения процента оригинальности присланного исходного кода предполагается использовать метод разбиения на токены. Данный метод почти не зависит от языка программирования и позволяет обнаружить типичные попытки плагиата (перестановки фрагментов кода, переименование переменных и т. д.) [13];

– чтобы безопасно проверить исходный код решения, его необходимо компилировать (проверять интерпретатором) и выполнять строго внутри виртуальной машины. Также необходимо изолировать остальные данные от проверяемого решения. Благодаря такому подходу потенциально вредоносный код не сможет навредить автоматической проверке или исказить её результаты. Кроме того, благодаря виртуальным машинам, можно будет проверять решения, написанные для разных операционных систем;
– для оценки работы программы студента предлагается использовать набор тестов. Вывод программы посимвольно сравнивается с эталонным ответом теста. Помимо этого, необходимо производить замер потребляемой программой памяти и времени её выполнения. В зависимости от количества пройденных тестов выставляется оценка работе программы. Такой подход недостаточно гибкий (производится прямое сравнение строк), однако он имеет одно важное для преподавателя преимущество: в виду своей простоты подобный набор тестов довольно легко и быстро создать.

В данном подразделе были предложены возможные пути решения задач автоматической проверки программных решений.

Во втором разделе в соответствии с поставленными ранее задачами были рассмотрены и проанализированы способы и методы автоматической проверки программных решений. Исходя из результатов анализа, были предложены пути решения задач автоматической проверки программных решений.

3 Разработка модели процесса автоматической проверки задач по программированию

3.1 Общая модель процесса проверки программного решения

Предполагается, что с проектируемой системой автоматической проверки будут работать следующие актёры:

- студент – человек, который проходит какие-либо обучающие курсы, связанные с программированием, пишет программный код.
- преподаватель – человек, который ведёт курсы (обучает студентов) по программированию, выдаёт задания студентам для закрепления материала.

Основным процессом, протекающим в системе, является процесс автоматической проверки программного решения (исходного кода) студента (смотреть рисунок 8). Для оценки решения будет использоваться сто балльная шкала.

Для того чтобы добавить решение в систему автоматической проверки, студенту необходимо в специальной форме выбрать сборщик проекта (или просто название компилятора/интерпретатора), параметры его запуска, а также прикрепить сам проект (или только файлы с исходным кодом).

После этого проверяется суммарный размер всех файлов с исходным кодом решения. Если их размер больше максимума, который установил преподаватель при создании задания, то такое решение сразу отклоняется и оценивается в 0 баллов, а в системе сохраняется соответствующий отчёт. Таким образом, мы отсеиваем нецелесообразно большие решения (например, вшитый в код огромный набор пар ключ-значение).

Если размер файлов с исходным кодом меньше или равен максимуму, то начинается загрузка решения на сервер системы автоматической проверки. По окончании загрузки содержимое всех файлов с исходным кодом объединяется в один текст (содержимое одного файла присоединяется к концу содержимого

предыдущего файла). Стоит отметить, что исходные файлы при этом не удаляются.

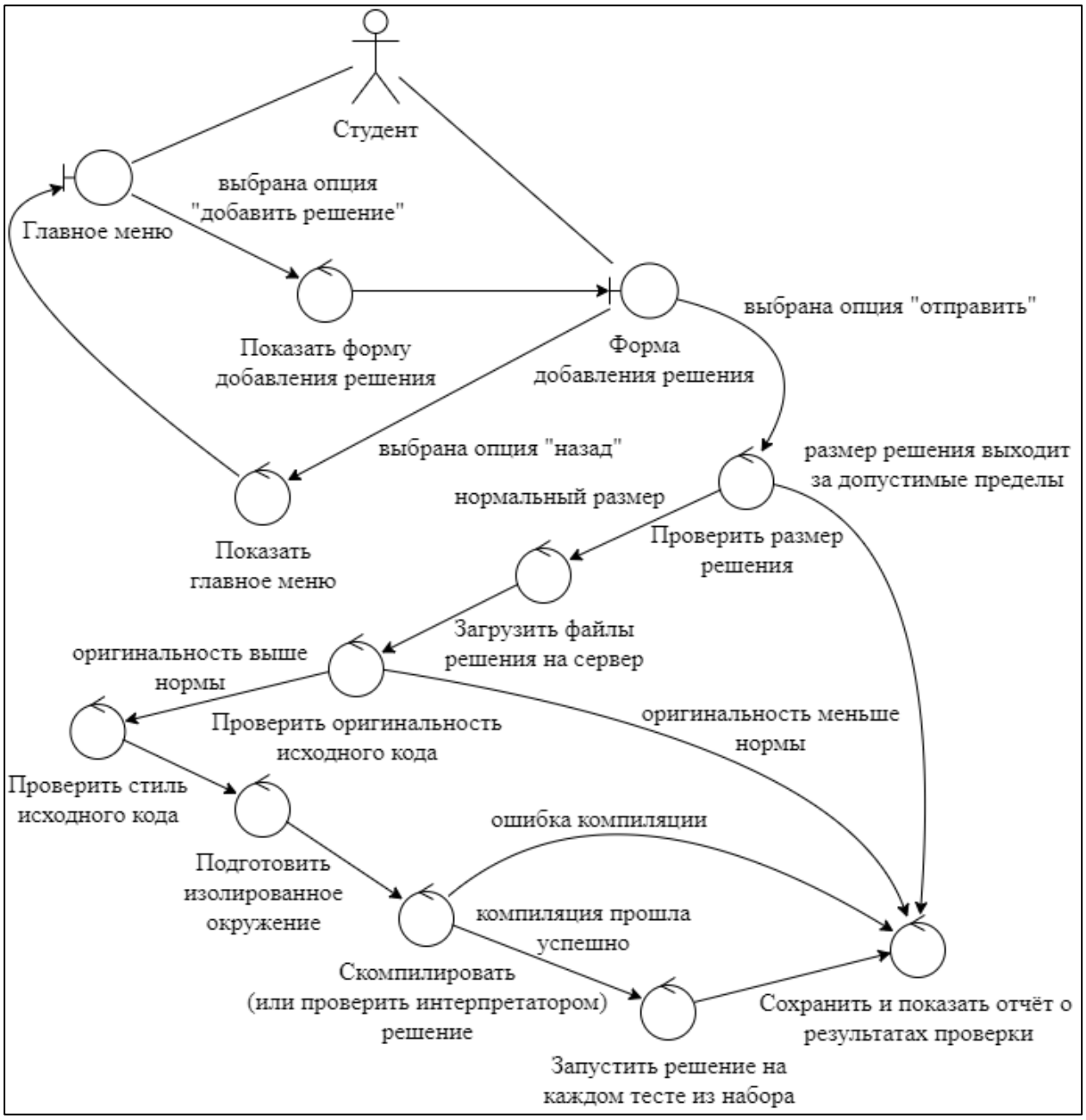

Рисунок 8 – Диаграмма пригодности «Проверка программного решения»

Таким образом, в данном подразделе была представлена модель процесса проверки программного решения в общем виде, а также описано то, как начинается проверка. Далее рассмотрим основные этапы проверки более подробно.

3.2 Проверка оригинальности исходного кода программного решения

Сначала необходимо определить, какие различия между двумя решениями так или иначе будут считаться плагиатом:

– добавление/удаление несущественных символов (пробелов, знаков табуляции и т. п.);
– изменение имён и замена типов у переменных и функций;
– добавление новых элементов к полностью скопированному решению.

Во всех остальных случаях считается, что решение студента в той или иной степени оригинально. Стоит также отметить, что если проверяемое решение отличается от остальных языком программирования, на котором оно написано, то такое решение считается полностью уникальным и не нуждается в выявлении плагиата. Чтобы определить степень оригинальности, предлагается использовать следующие данные и алгоритм (смотреть таблицу 9 и рисунок 9).

Таблица 9 – Исходные данные для выявления плагиата

Обозначение	Тип	Описание
P_{limit}	Вещественное неотрицательное число (до 100 включительно)	Максимально допустимый процент плагиата.
W	Вещественное неотрицательное число (до 100 включительно)	Доля проверки исходного кода в процентах. Эта доля указывает на то, насколько заимствования в исходном коде будут сильнее влиять на итоговый процент плагиата в сравнении с заимствованиями в комментариях.
S^*	Строка	Проверяемое решение в виде одного текста.
$S_1, \dots, S_n$	Массив строк	Все остальные решения студентов (их файлы с исходным кодом в виде единых текстов).

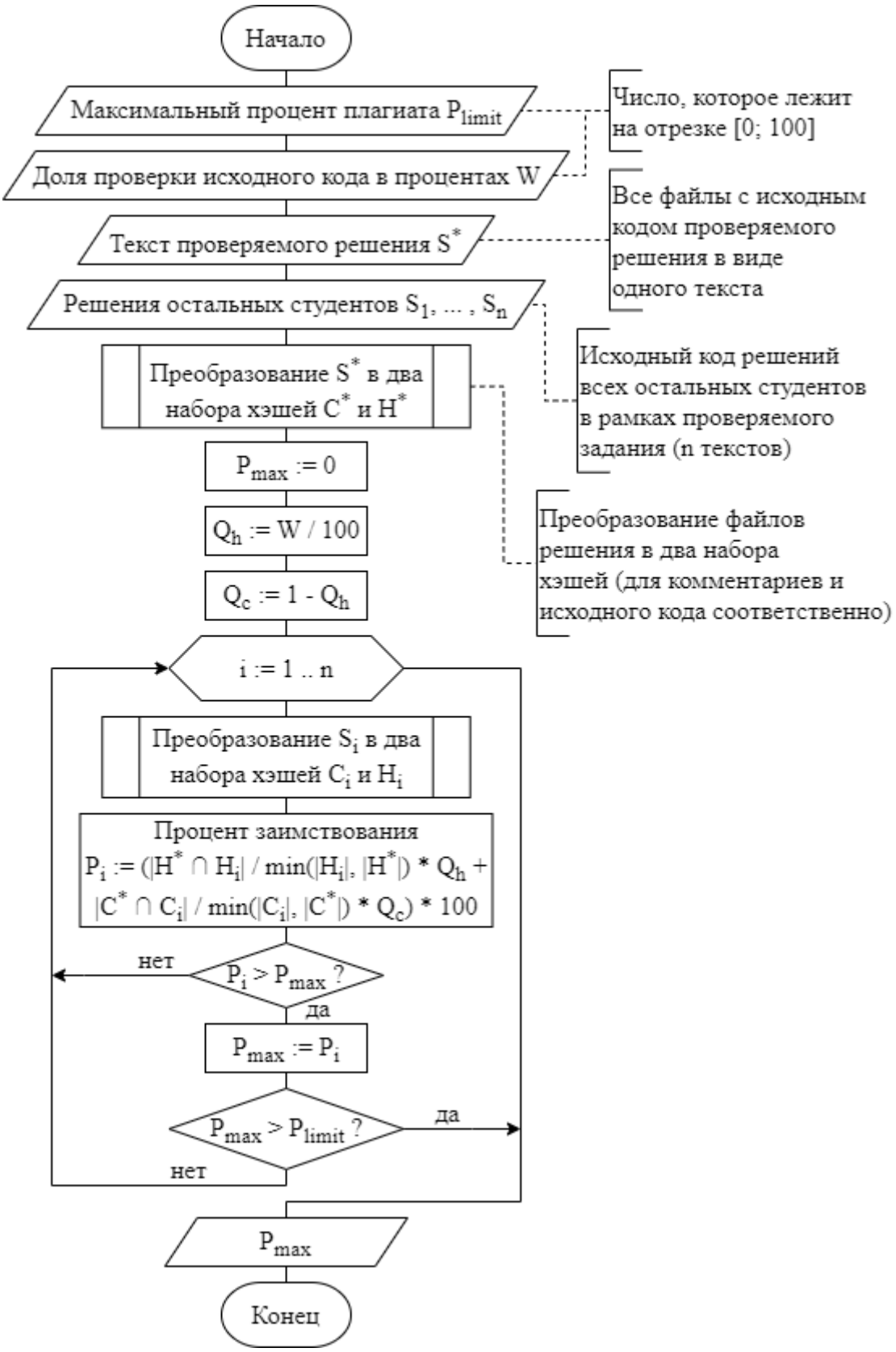

Рисунок 9 – Алгоритм выявления плагиата

Вначале текст проверяемого решения преобразуется в два набора контрольных сумм (смотреть рисунок 10). От текста с помощью регулярных выражений отделяются комментарии (остаётся только исходный код программы). Начинается параллельное преобразование исходного кода и комментариев. Происходит нормализация исходного кода:

– все несущественные части (например, символы перевода строк, точка с запятой и т. д.) заменяются пробелом или полностью убираются;
– каждая последовательность пробельных символов заменяется одним пробелом.

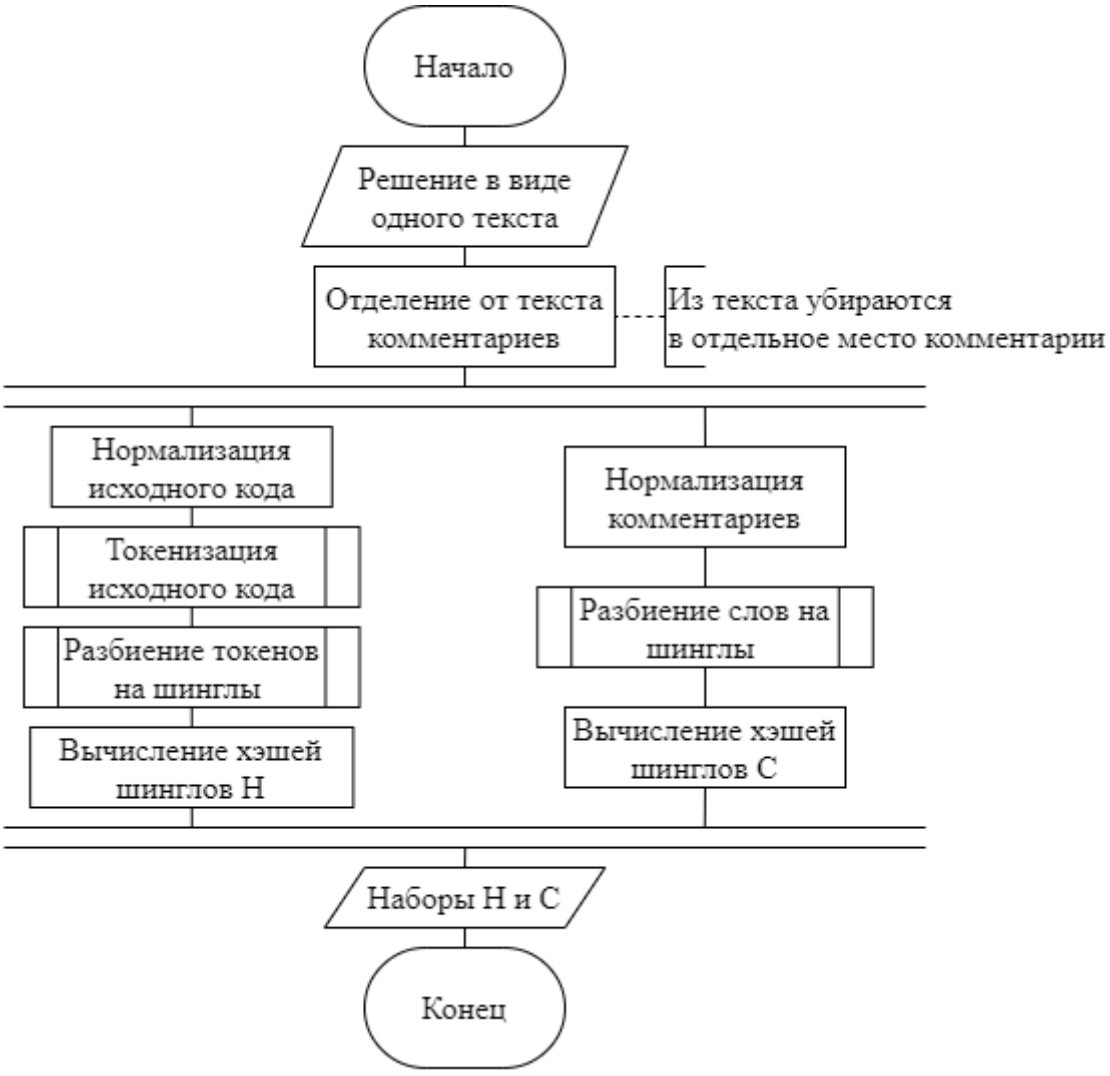

Рисунок 10 – Алгоритм преобразования текста решения

После этого обработанный исходный код при помощи набора регулярных выражений преобразуется в массив токенов (смотреть рисунок 11).

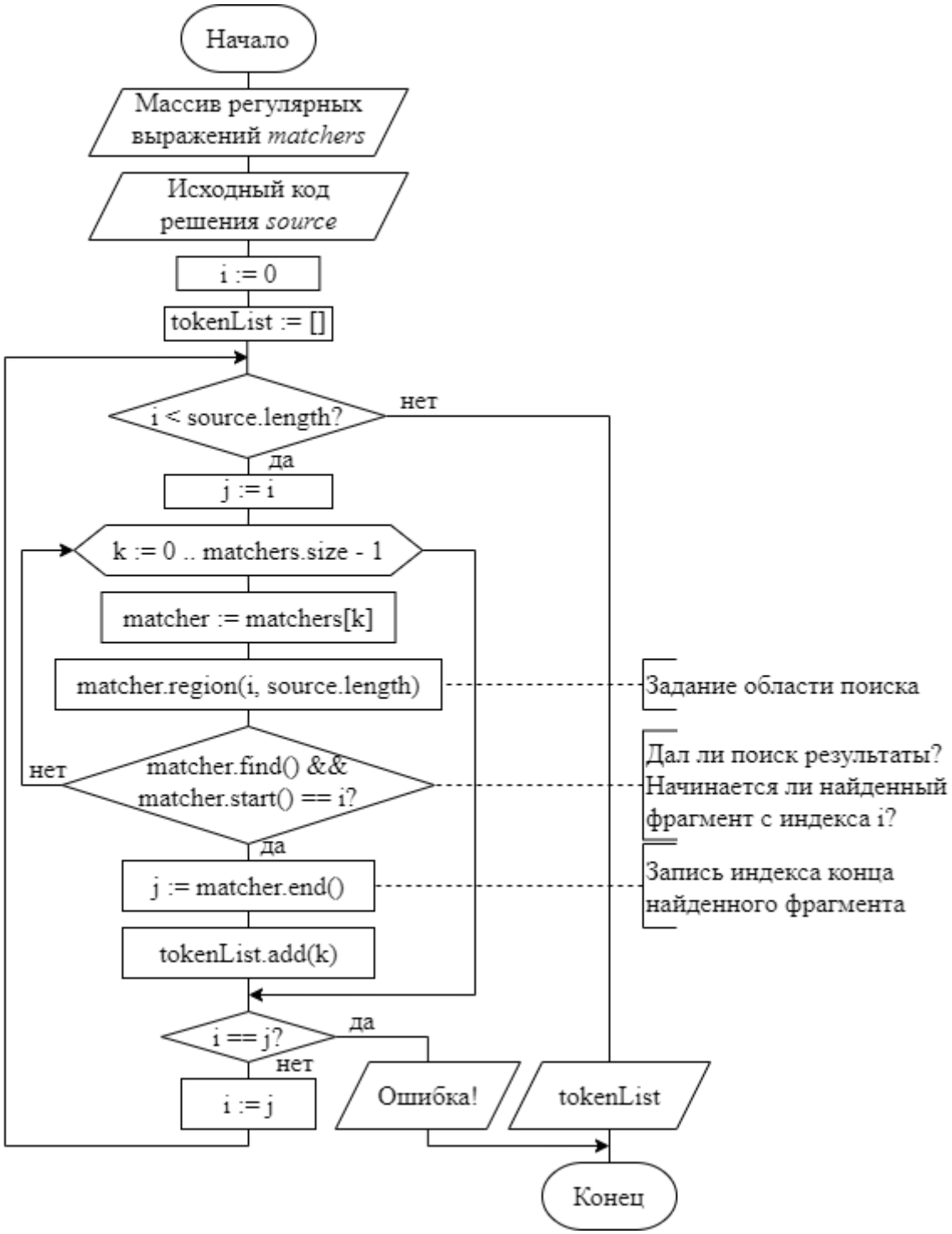

Рисунок 11 – Алгоритм токенизации исходного кода

Токен – это минимальная единица языка программирования, имеющая самостоятельный смысл. В данном случае каждый токен будет представляться в виде целого числа, соответствующего определённому виду этого токена (идентификатор, ключевое слово, знак операции, разделитель, константа и т. д.). Каждое регулярное выражение описывает один вид токена.

Из полученного набора токенов (массива целых чисел) составляются шинглы (группы токенов) одинакового размера (смотреть рисунок 12), причём выборка происходит внахлёст, а не встык (с фиксированным шагом).

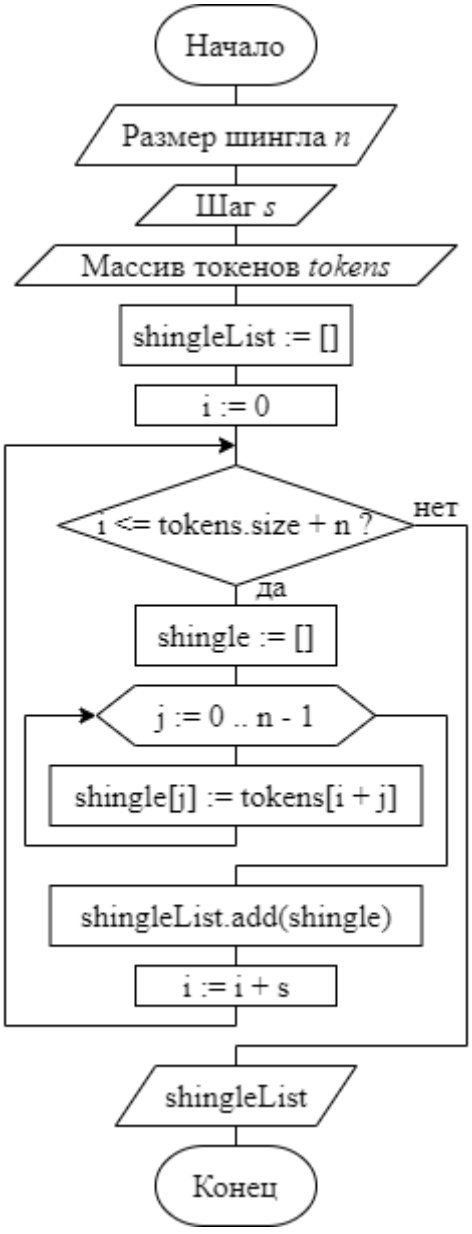

Рисунок 12 – Алгоритм создания набора шинглов

Теперь для каждого шингла находится контрольная сумма, которая вычисляется по формуле:

$$h = 31^n + \sum_{i=1}^{n} 31^{n-i} * a_i, \quad (1)$$

где h – контрольная сумма,

n – количество токенов (целых чисел) в шингле (массиве),

i – порядковый номер токена,

a – шингл (массив).

Для комментариев процесс почти аналогичен. Вначале выполняется нормализация комментариев:

- производится их очистка (или замена на пробел) от предлогов, союзов, знаков препинания и прочих символов;
- каждая последовательность пробельных символов заменяется одним пробелом.

После этого оставшиеся слова разбиваются на шинглы одинакового размера (группы по несколько слов каждая). Выборка всё так же осуществляется внахлёст. Для каждого шингла находится контрольная сумма, которая вычисляется по формуле:

$$c = \sum_{i=1}^{n} 31^{n-i} * \sum_{j=1}^{m} 31^{m-j} * s_{i,j}, \quad (2)$$

где h – контрольная сумма,

n – количество слов в шингле,

i – порядковый номер слова,

m – длина слова,

j – порядковый номер символа в слове,

s – массив слов.

Таким образом, в результате работы алгоритма преобразования, показанного на рисунке 10, получается два набора контрольных сумм (для комментариев и исходного кода). Возвращаемся обратно к основному алгоритму (смотреть рисунок 9). Дальше в цикле проверяемое решение сравнивается с остальными. Для этого решение другого студента так же предварительно преобразуется в два набора контрольных сумм. Находятся пересечения соответствующих наборов контрольных сумм, т. е. создаются два множества, которые состоят из совпадающих контрольных сумм для исходного кода и комментариев. Вычисляются размеры этих двух пересечений. Находятся размеры соответствующих исходных массивов с контрольными суммами, а затем среди них берутся минимальные размеры (отдельно для исходного кода и комментариев). Размеры двух пересечений делятся на соответствующие минимумы. Получается процент плагиата в исходном коде и процент плагиата в комментариях. Эти проценты умножаются на соответствующие коэффициенты, а затем складываются. Сумма умножается на 100.

В результате получается процент плагиата. Данный процент показывает, сколько в проверяемом решении содержится фрагментов из другого решения. Получается, если студент полностью скопирует всё решение другого студента и, не изменяя ничего, добавит к нему что-то новое, то данный алгоритм всё равно при сравнении выдаст почти 100%. Перестановка фрагментов скопированного решения соответственно тоже не сильно уменьшит процент плагиата.

Процент плагиата получен, и если он больше максимума, то осуществляется выход из цикла и вывод процента. В противном случае производится сравнение со следующим решением. Если максимум так и не был превышен, то в конце осуществляется вывод наибольшего процента, полученного в ходе сравнений.

Из 100 вычитается итоговый процент плагиата. Получается процент оригинальности проверяемого решения. Если он меньше значения, которое

установил преподаватель при составлении задания, то проверяемое решение отклоняется и оценивается в 0 баллов, а в системе автоматической проверки сохраняется соответствующий отчёт.

В данном подразделе был рассмотрен алгоритм выявления плагиата в исходном коде решения, а также представлен алгоритм преобразования решения в два набора контрольных сумм, которые необходимы для проверки.

3.3 Оценка стиля исходного кода программного решения

Под проверкой стиля исходного кода понимается оценка следующих характеристик кода:

- количество непустых строк внутри каждого функционального блока (функции, класса, метода и т. д.);
- уровень вложенности функциональных блоков;
- размер отступа вначале каждой строки;
- длина строки;
- количество запрещённых символов, которые были использованы при написании кода;
- количество некорректных выражений в коде.

Перед началом проверки стиля исходного кода, создаётся пустой набор штрафных баллов. Штрафной балл – это действительное число в диапазоне от 0, не включая его, до 100 включительно. Каждый штрафной балл уменьшает итоговый балл согласно формуле:

$$S_i = S_{i-1} - S_{i-1} * \frac{P_i}{100}, \quad (3)$$

где S – итоговый балл,

i – порядковый номер штрафного балла,

P – набор штрафных баллов.

Данная формула применяется последовательно для каждого штрафного балла из набора. Получается, чем меньше итоговый балл, тем больше нужно штрафных баллов, чтобы уменьшить его. И, наоборот, если у студента довольно высокий итоговый балл, то даже несколько штрафных баллов могут заметно его уменьшить.

Для самой проверки стиля исходного кода требуются исходные данные, которые представлены в таблице 10.

Таблица 10 – Исходные данные для проверки стиля исходного кода

Название	Тип	Описание	Пример
blockStart	Символ	Символ начала функционального блока.	‘{’
blockEnd	Символ	Символ конца функционального блока.	‘}’
blockSizes	Массив целых положительных чисел	Максимально допустимое количество непустых строк внутри функционального блока (класса, метода, цикла и т. д.) для каждого уровня.	[500, 500, 50]
blockSizesPenalty	Массив вещественных положительных чисел (до 100.0 включительно)	Список штрафов за превышение максимумов из массива blockSizes. Размер blockSizesPenalty должен быть равен размеру blockSizes.	[1.0, 2.0, 3.0]
maxNestingDepth	Целое положительное число	Максимальная вложенность функциональных блоков.	7
maxNestingDepthPenalty	Вещественное положительное число (до 100.0 включительно)	Штраф за превышение максимальной вложенности.	4.0
indentPenalty	Вещественное положительное число (до 100.0 включительно)	Штраф за нарушение размера отступа вначале каждой строки (размер отступа должен быть больше или равен уровню функционального блока)	3.8

Продолжение таблицы 10

Название	Тип	Описание	Пример
maxBytesInLine	Целое положительное число	Максимально допустимое количество символов в одной строке.	120
maxBytesInLinePenalty	Вещественное положительное число (до 100.0 включительно)	Штраф за превышение maxBytesInLine.	3.5
badSymbols	Массив символов	Набор запрещённых символов.	[‘a’, ‘b’, ‘c’]
badSymbolsPenalty	Массив вещественных положительных чисел (до 100.0 включительно)	Штрафы за использование соответствующих символов из badSymbols.	[5.0, 40.8]
badExpressions	Массив регулярных выражений (строк)	Набор регулярных выражений, которые описывают запрещённые конструкции (например, имена некоторых функций).	[“\.function”, “val [а-яА-Я_0-9]+”]
badExpressionsPenalty	Массив вещественных положительных чисел (до 100.0 включительно)	Штрафы за использование конструкций, описанных в badExpressions.	[80.2, 50.0, 20.8]
solutionFiles	Массив файлов (строк)	Файлы с исходным кодом решения (без комментариев)	[“task1.java”, “task2.java”]

Выполняется посимвольная обработка каждого файла решения (определяются функциональные блоки, их размер, подсчитываются пробельные символы вначале каждой непустой строки и т. д.) согласно диаграмме состояний, изображённой на рисунке 13. Если в результате обработки были обнаружены нарушения ограничений, описанных выше (был превышен какой-либо максимум или найден запрещённый символ), то в набор штрафных баллов помещаются соответствующие этим нарушениям значения.

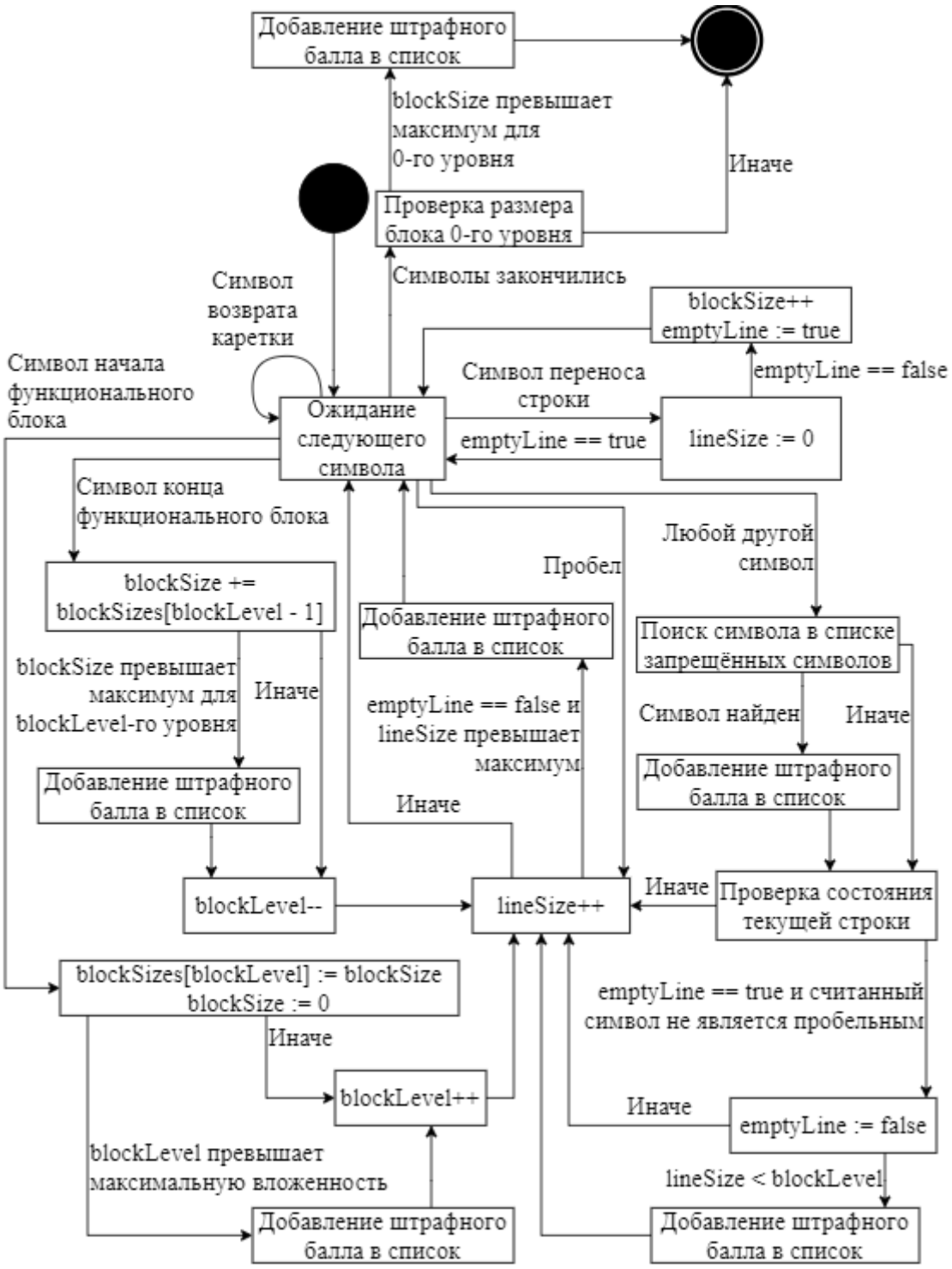

Рисунок 13 – Диаграмма состояний «Посимвольная проверка стиля»

Дополнительно файлы с исходным кодом решения проверяются при помощи регулярных выражений (смотреть рисунок 14): ищутся фрагменты, которые удовлетворяют регулярным выражениям из набора. Если такие фрагменты были найдены, то в набор штрафных баллов помещаются значения

в соответствии с запрещёнными конструкциями (например, имена некоторых функций), которые были использованы.

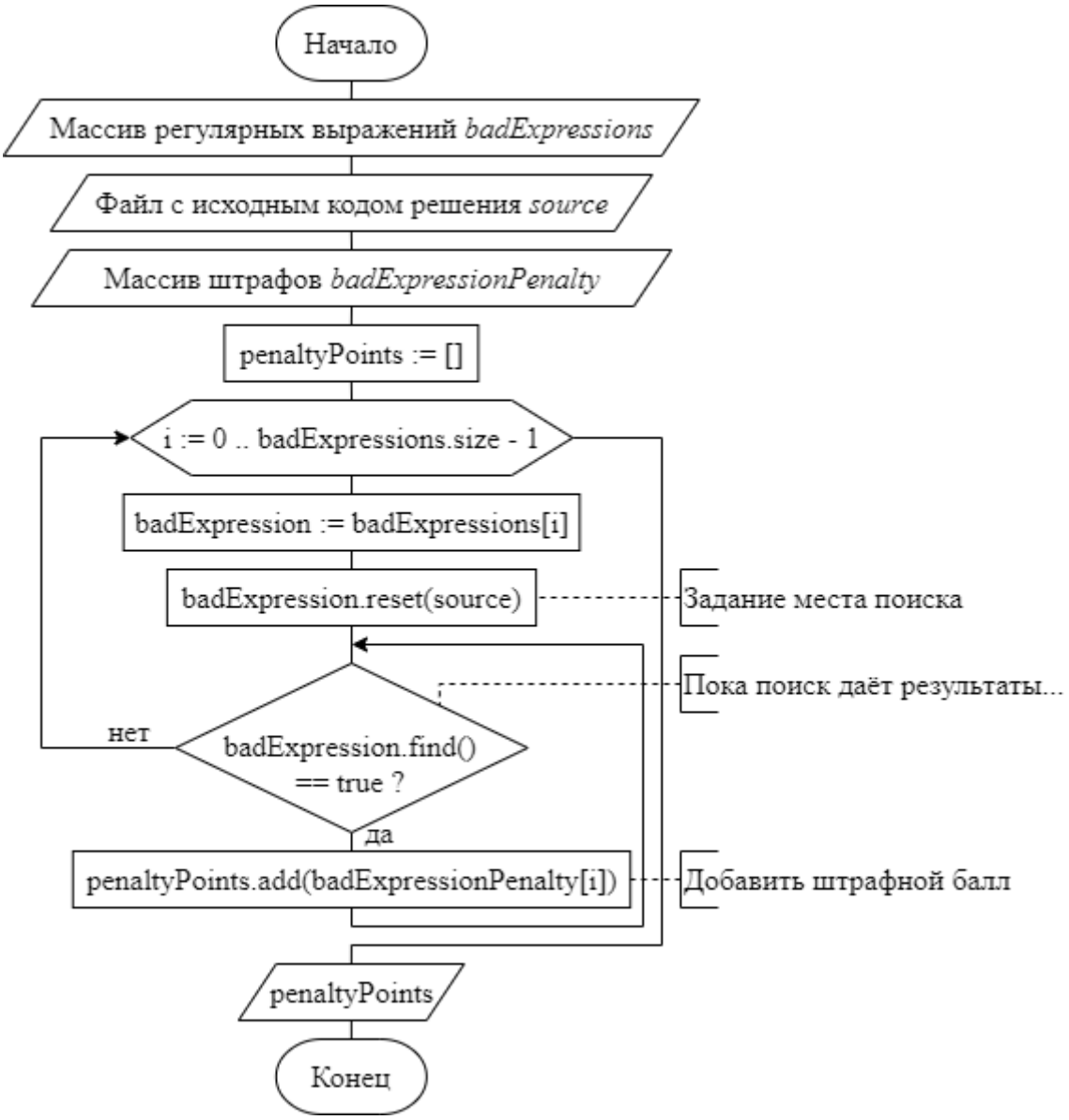

Рисунок 14 – Алгоритм поиска запрещённых выражений

В конце проверки стиля исходного кода все полученные штрафные баллы добавляются в общий список, который будет использоваться на последнем этапе проверки решения студента для уменьшения итогового балла. Стоит отметь, что если в списке штрафных баллов присутствует значение 100, то дальнейшая проверка прекращается, решение оценивается в

0 баллов, а в системе автоматической проверки сохраняется соответствующий отчёт.

Таким образом, в данном подразделе дано определение стиля исходного кода, описана система штрафных баллов и показан алгоритм оценки стиля исходного кода, который её использует.

3.4 Оценка корректности работы программы

Прежде чем приступать к дальнейшей проверке решения студента, системе автоматической проверки необходимо сначала выполнить следующие действия (смотреть рисунок 15).

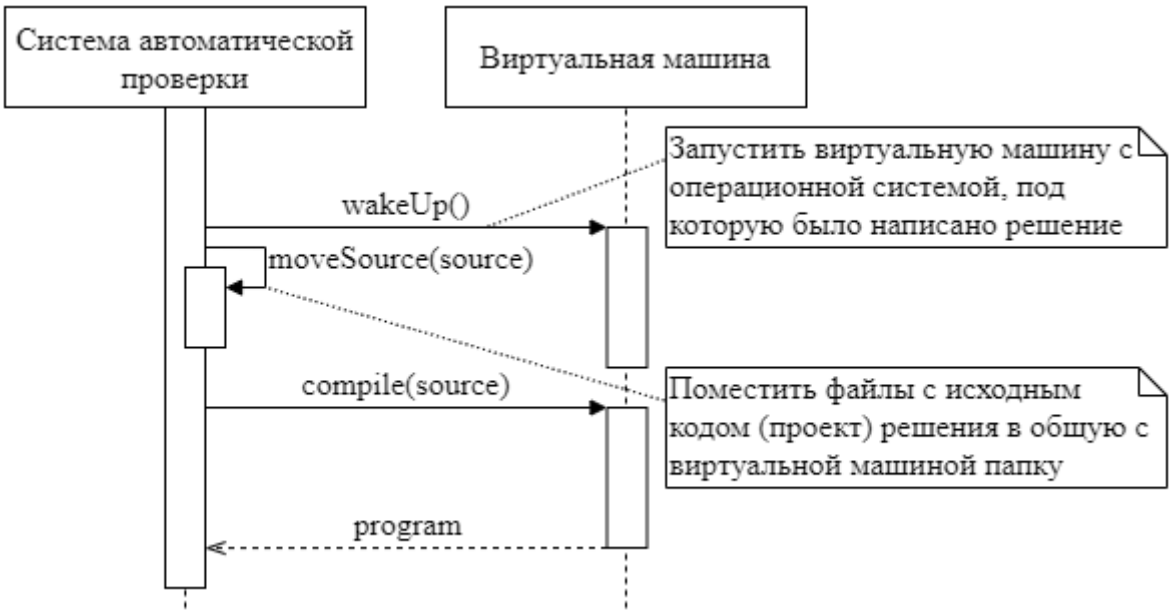

Рисунок 15 – Подготовка к проверке работы программы

Система автоматической проверки на виртуальной машине запускает процесс компиляции (или проверку интерпретатором) исходного кода решения. В случае возникновения ошибки во время этого процесса дальнейшая проверка прекращается, решение оценивается в 0 баллов, а в

системе сохраняется соответствующий отчёт. Под ошибкой в данном случае подразумевается появление хотя бы одного из следующих событий:

– превышено максимально допустимое время;
– превышен лимит оперативной памяти;
– ошибка компиляции (обнаружена синтаксическая ошибка);
– аварийное завершение работы компилятора/интерпретатора.

Если компиляция прошла успешно, то начинается оценка корректности работы полученной программы. Запуск проверяемой программы система автоматической проверки будет осуществлять всё так же внутри виртуальной машины, в которой проводилась компиляция.

Проверка программы осуществляется при помощи набора тестов, где каждый тест включает в себя данные, которые представлены в таблице 11.

Таблица 11 – Компоненты одного теста

Название	Тип	Описание	Пример
Исходные данные (input)	Строка	Это данные, которые будут подаваться на вход проверяемой программе в виде аргументов командной строки.	“10 data 5”
Входные файлы (input)	Массив файлов (строк)	Это файлы, пути к которым будут подаваться на вход проверяемой программе вместе с исходными данными (сначала идут исходные данные, затем через пробел записываются пути к файлам).	[“task/key.txt”, “task/data.txt”]
Эталонный ответ (output)	Строка	Это данные, которые должна выдавать проверяемая программа на стандартный поток вывода.	“max 8 min 7”
Ожидаемые файлы (output)	Массив файлов (строк)	Это файлы с определённым содержимым, которые должна создать проверяемая программа при выполнении на соответствующих аргументах командной строки.	[“task/enc.txt”, “task/res.txt”]

Продолжение таблицы 11

Название	Тип	Описание	Пример
Вес теста (weight)	Целое положительное число	Чем выше вес, тем больше баллов принесёт студенту успешное и правильное выполнение программы на соответствующем наборе аргументов.	2
Минимальное время (minTime)	Вещественное неотрицательное число, а также единица измерения времени (строка)	Минимальное время выполнения проверяемой программы на соответствующих аргументах командной строки.	1s 800ms
Максимальное время (maxTime)	Вещественное неотрицательное число, а также единица измерения времени (строка)	Максимальное время выполнения проверяемой программы на соответствующих аргументах командной строки.	5.2s
Минимальная память (minMemory)	Вещественное неотрицательное число, а также единица измерения информации (строка)	Минимальное количество оперативной памяти, которое должна использовать проверяемая программа.	90.5Mb
Максимальная память (maxMemory)	Вещественное неотрицательное число, а также единица измерения информации (строка)	Максимальное количество оперативной памяти, которое может использовать проверяемая программа.	1Gb 450Mb

Согласно выбранной шкале за успешное прохождение всех тестов студент может максимально набрать 100 баллов. Далее на рисунке 16 показано, как проводится тестирование программы на наборе тестов, описанном выше. На вход подаётся набор тестов, описанных выше, исполняемый файл проверяемой программы, а также лимиты памяти и времени. Затем суммируются веса всех тестов. Дальше в цикле проверяемой программе подаются на вход в виде аргументов командой строки исходные данные каждого теста, включая его входные файлы. Поверяемая программа начинает свою работу, и если она не выдаёт результат в установленный временной промежуток, то её работа принудительно останавливается, за этот тест студент получает 0 баллов, осуществляется переход к следующему тесту.

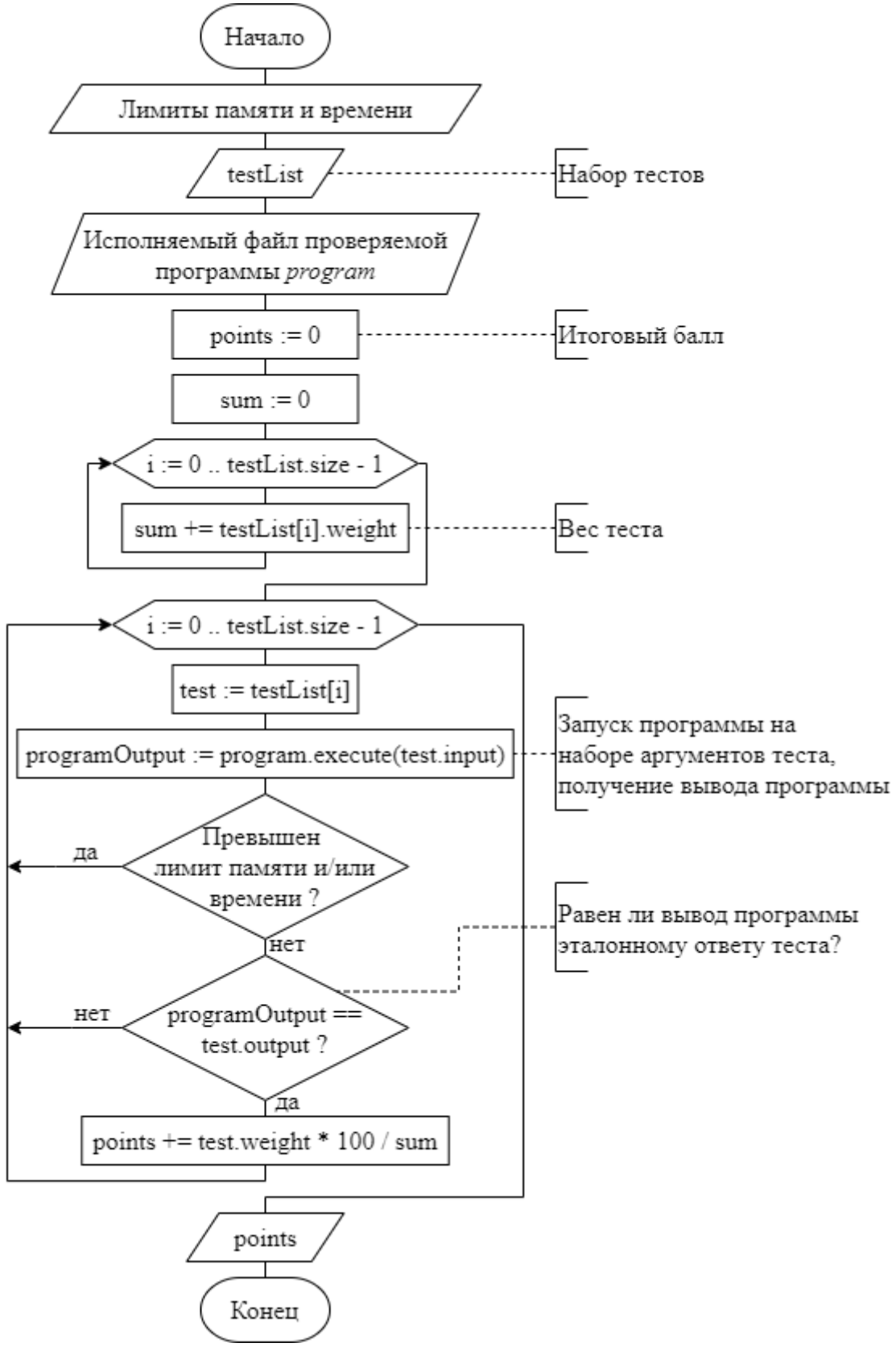

Рисунок 16 – Алгоритм тестирования программы

Аналогично для лимитов памяти. Если же лимиты времени и памяти не были нарушены, то содержимое стандартного потока вывода (вывод проверяемой программы) сравнивается с эталонным ответом, который указан в тесте. Помимо этого, созданные проверяемой программой файлы сопоставляются с файлами, определёнными в тесте. Если вывод программы и созданные ею файлы строго равны соответствующим данным теста, то к итоговому баллу прибавляется значение, которое получается путём деления веса текущего теста на сумму весов и умножения на 100. После прохождения всех тестов осуществляется вывод итогового балла.

К полученному баллу применяется формула 3 последовательно для каждого штрафного балла из набора. В результате получается итоговый балл с учётом всех штрафов, который и является итоговой оценкой всего программного решения, присланного студентом. В конце в системе автоматической проверки сохраняется отчёт о результатах каждого этапа проверки.

Таким образом в данном подразделе была рассмотрена подготовка к тестированию, определены данные теста, описан сам алгоритм тестирования программы и объяснено, как формируется итоговая оценка программного решения.

В третьем разделе показана модель процесса проверки программного решения в общем виде. Рассмотрены этапы проверки программного решения, которые служат своеобразным барьером. Осуществляется проверка размера и оригинальности решения, а также преобразование в удобный для анализа вид. Отсеиваются различные решения, которые по своей сути являются попытками студентов обмануть систему автоматической оценки, чтобы получить высокий балл. Были рассмотрены этапы проверки программного решения, на которых формируется итоговая оценка: проверка стиля исходного кода программы, а также его компиляция и тестирование исполняемого файла в изолированном окружении.

4 Оценка разработанной модели процесса автоматической проверки задач по программированию

4.1 Критерии оценки полученной модели процесса автоматической проверки программных решений

Чтобы оценить разработанную модель процесса автоматической проверки программных решений, необходимо выделить критерии её оценки (смотреть таблицу 12).

Таблица 12 – Критерии оценки модели автоматической проверки программ

Номер	Критерий	Описание
К1	Безопасность процесса автоматической проверки	Балл, который определяет степень защищённости процесса автоматической проверки программных решений от вредоносного кода, способного исказить итоговую оценку или нарушить работу системы автоматической оценки.
К2	Гибкость методов оценки программного решения	Балл, который показывает, как много типов задач по программированию можно автоматически проверить с помощью данной модели.
К3	Степень независимости от языка программирования	Некоторый балл, который показывает, насколько широк ряд языков программирования, с которыми может работать данная модель автоматической проверки программных решений. Сюда также входит то, как мало используется различных алгоритмов для работы с поддерживаемыми моделью языками программирования, насколько сильно алгоритмы модели абстрагированы от конкретного языка программирования.
К4	Степень вычислительной нагрузки	Некоторый балл, который показывает, насколько низки затраты мощностей вычислительной техники на работу данной модели автоматической проверки программных решений.
К5	Объективность итоговой оценки программного решения	Балл, который показывает, как много характеристик программного решения принимается во внимание при автоматической проверке.

Оценка будет производиться по пятибалльной шкале. Баллы по критериям будут выставляться относительно работы существующих систем автоматической проверки программных решений.

В данном подразделе были выделены и описаны критерии оценки модели автоматической проверки задач по программированию, определены правила оценивания.

4.2 Исходные данные для проверки программных решений

Рассмотрим работу созданной модели автоматической проверки программных решений на примере небольшого задания «сортировка вставками». Текст задания выглядит следующим образом: «Реализуйте на языке программирования Java алгоритм сортировки вставками. На вход программе в качестве аргументов командной строки будут подаваться вещественные числа. Необходимо вывести на стандартный поток вывода отсортированный массив (те же вещественные числа через пробел в порядке возрастания).». Определим, какие данные необходимы, чтобы осуществить автоматическую проверку программного решения.

Согласно модели сначала необходимо выполнить проверку размера файлов с исходным кодом. Так как задание небольшое, ограничим размер файлов до 4 килобайт.

Далее идёт проверка оригинальности. Необходимо отделить исходный код от комментариев. Для этого воспользуемся регулярным выражением «/*{1,2}(.*)*/|//([^\n]*)» (фрагменты единого текста, которые удовлетворяют этому регулярному выражению, будут помещены отдельно от оставшихся фрагментов).

Нормализацию будем осуществлять с помощью регулярного выражения «[()\[\]{},;\s&&[^]]» (будем заменять все фрагменты кода, которые удовлетворяют этому регулярному выражению на пробел, а множественные пробелы заменять одним).

Для токенизации понадобиться набор регулярных выражений (смотреть рисунок 17).

```
[
  "(abstract|default|extends|final|implements|native|
private|protected|public|static|strictfp|synchronized|
throws|transient|volatile) ",
  "assert ",
  "(boolean|byte|char|double|float|int|long|short|void) ",
  "(break|continue) ",
  "(if|else|case|switch) ",
  "(try|catch|finally|throw) ",
  "(enum|class) ",
  "(while|do|for) ",
  "(package|import) ",
  "instanceof ",
  "new ",
  "return ",
  "[?:] ?",

  "(true|false) ?",
  "null ?",
  "super ?",
  "this ?",

  "[+\\-*/%] ?",
  "= ?",
  "(\\+\\+|--|[+\\-*/%]=) ?",
  "([&|^]|>>|<<)= ?",
  "(&&|\\|\\||!) ?",
  "(==|!=|[<>]=?) ?",
  "(>{2,3}|<<|[&|^~]) ?",

  "('.*'|\"[^\"]*\") ?",
  "\\.[_a-zA-Z*][_a-zA-Z0-9]* ?",
  "[_a-zA-Z][_a-zA-Z0-9]* ?",
  "-?[0-9]*\\.?[0-9]*[fd]? ?"
]
```

Рисунок 17 – Типы токенов в Java (JSON)

В представленном наборе регулярных выражений каждая строка описывает один тип токена. Например, третье регулярное выражение представляет собой описание примитивных типов языка Java. Таким образом, в соответствии с этим регулярным выражением и его порядковым номером в наборе все названия примитивных типов в исходном коде решения будут заменяться на целое число 2, которое и является токеном.

Пусть размер шингла на соответствующем этапе разбиения будет равен 10 для исходного кода и 3 для комментариев, а шаг – 1.

Необходимо также задать долю проверки исходного кода и минимальный процент оригинальности. Предположим их равными 80% и 50% соответственно, так как проверяемые решения будут небольшого размера.

Для проверки стиля исходного кода воспользуемся данными, которые представлены в таблице 13.

Таблица 13 – Входные данные для проверки стиля исходного кода

Название	Значение	Комментарий
blockStart	‘{’	Начало функции, метода и т. д.
blockEnd	‘}’	Конец функции, метода и т. д.
blockSizes	[50, 50, 10]	Максимальные размеры файла, класса и функции соответственно.
blockSizesPenalty	[5.0, 5.0, 10.0]	Штрафы за нарушение размеров функциональных блоков.
maxNestingDepth	5	Максимальная вложенность функциональных блоков.
maxNestingDepthPenalty	10.0	Штраф за нарушение вложенности.
indentPenalty	15.0	Штраф за нарушение размера отступа вначале строки.
maxBytesInLine	60	Максимальная длина одной строки (небольшая для примера).
maxBytesInLinePenalty	10.0	Штраф за превышение максимальной длины строки.
badSymbols	[‘\t’]	Запрет символа табуляции.
badSymbolsPenalty	[1.0]	Штраф за использование одного символа табуляции.
badExpressions	[“for\(*;”]	У «for» обязательно должно быть начало.
badExpressionsPenalty	[12.0]	Штраф за неполный «for».

Для оценки корректности работы программы будем использовать набор тестов, который представлен в таблице 14.

Таблица 14 – Набор тестов для оценки программного решения

Исходные данные	Эталонный ответ	Вес теста	Ограничения времени	Ограничения памяти
1 5 3 8 4 7	1 3 4 5 7 8	1	От 0ms до 1ms	От 0Mbyte до 500Mbyte
1 9 2 4 8 3 7 5 6	1 2 3 4 5 6 7 8 9	2	От 0ms до 1ms	От 0Mbyte до 500Mbyte
0 1 0 2 3 8 5 5 0	0 0 0 1 2 3 5 5 8	3	От 0ms до 1ms	От 0Mbyte до 500Mbyte
-1 -9 2 1 8 -2 -1	-9 -2 -1 -1 1 2 8	4	От 0ms до 1ms	От 0Mbyte до 500Mbyte
1.1 0.09 3.5 0 1.5	0 0.09 1.1 1.5 3.5	4	От 0ms до 1ms	От 0Mbyte до 500Mbyte

Необходимые данные для работы модели процесса автоматической проверки подготовлены. Теперь рассмотрим два программных решения, которые будут оцениваться (смотреть рисунки 18 и 19).

```
1   public class Main2 {
2       // Главная функция
3       public static void main(String[] args) {
4           int[] array = new int[args.length];
5           int i;
6           for (i = 0; i < args.length; i++)
7   array[i] = Integer.parseInt(args[i]); // Считывание массива
8           // Сортировка массива
9           for (i = 1; i < array.length; i++) {
10              int current = array[i];
11              int j = i - 1;
12              while(j >= 0 && array[j] > current) {
13                  array[j + 1] = array[j];
14                  j--;
15              }
16              array[j + 1] = current;
17          }
18          // Вывод массива
19          System.out.print(array[0]);
20          for (i = 1; i < array.length; i++)
21  System.out.format(" %d", array[i]);
22      }
23  }
```

Рисунок 18 – Второе программное решение

```java
public class Main1 {
    // Главная функция
    public static void main(String[] args) {
        int[] forSort = convert(args);
        insertionSort(forSort);
        System.out.print(forSort[0]);
        for (int i = 1; i < forSort.length; i++) {
            System.out.print(' ');
            System.out.print(forSort[i]);
        }
    }
    // Преобразование в массив целых чисел
    public static int[] convert(String[] source) {
        int size = source.length;
        int[] result = new int[size];
        for (int i = 0; i < size; i++)
            result[i] = Integer.parseInt(source[i]);
        return result;
    }
    // Сортировка вставками
    public static void insertionSort(int[] array) {
        for (int i = 1; i < array.length; i++) {
            int current = array[i];
            int j = i- 1;
            for(;j >= 0 && current < array[j]; j--)
                array[j + 1] = array[j];
            array[j + 1] = current;
        }
    }
}
```

Рисунок 19 – Первое программное решение

Эти решения выглядят по-разному, однако они выполняют одинаковые действия: сортируют массив чисел с помощью алгоритма вставками.

В данном подразделе был рассмотрен пример задачи по программированию, а также два возможных её решения на языке программирования Java. Также были представлены данные, которые необходимы алгоритмам системы автоматической проверки для того, чтобы проверить и оценить эти решения.

4.3 Тестирование разработанной модели процесса автоматической проверки программных решений

Начнём с проверки размеров файлов с исходным кодом данных решений. Их размер меньше 4 килобайт, значит можно продолжать проверку.

Далее необходимо отделить исходный код от комментариев и нормализовать его. В результате получатся следующие данные (смотреть рисунок 20).

```
Решение 1
public class Main1 public static void main String args int
forSort = convert args insertionSort forSort System.out.print
forSort 0 for int i = 1 i < forSort.length i++
System.out.print ' ' System.out.print forSort i public static
int convert String source int size = source.length int result
= new int size for int i = 0 i < size i++ result i =
Integer.parseInt source i return result public static void
insertionSort int array for int i = 1 i < array.length i++
int current = array i int j = i - 1 for j >= 0 && current <
array j j-- array j + 1 = array j array j + 1 = current
Решение 2
public class Main2 public static void main String args int
array = new int args.length int i for i = 0 i < args.length
i++ array i = Integer.parseInt args i for i = 1 i <
array.length i++ int current = array i int j = i - 1 while j
>= 0 && array j > current array j + 1 = array j j-- array j +
1 = current System.out.print array 0 for i = 1 i <
array.length i++ System.out.format " %d" array i
```

Рисунок 20 – Нормализованные решения (без комментариев)

Нормализировав комментарии, получим два текста, которые состоят только из слов (без знаков препинания). Дальше производится токенизация исходного кода. Получаются наборы целых чисел, которые изображены на рисунке 21.

```
Решение 1
0, 6, 26, 0, 0, 2, 26, 26, 26, 2, 26, 18, 26, 26, 26, 26, 26,
25, 25, 26, 27, 7, 2, 26, 18, 27, 26, 22, 26, 25, 26, 17, 17,
26, 25, 25, 24, 26, 25, 25, 26, 26, 0, 0, 2, 26, 26, 26, 2,
26, 18, 26, 25, 2, 26, 18, 10, 2, 26, 7, 2, 26, 18, 27, 26,
22, 26, 26, 17, 17, 26, 26, 18, 26, 25, 26, 26, 11, 26, 0, 0,
2, 26, 2, 26, 7, 2, 26, 18, 27, 26, 22, 26, 25, 26, 17, 17,
2, 26, 18, 26, 26, 2, 26, 18, 26, 17, 27, 7, 26, 22, 27, 21,
26, 22, 26, 26, 26, 17, 17, 26, 26, 17, 27, 18, 26, 26, 26,
26, 17, 27, 18, 26
Решение 2
0, 6, 26, 0, 0, 2, 26, 26, 26, 2, 26, 18, 10, 2, 26, 25, 2,
26, 7, 26, 18, 27, 26, 22, 26, 25, 26, 17, 17, 26, 26, 18,
26, 25, 26, 26, 7, 26, 18, 27, 26, 22, 26, 25, 26, 17, 17, 2,
26, 18, 26, 26, 2, 26, 18, 26, 17, 27, 7, 26, 22, 27, 21, 26,
26, 22, 26, 26, 26, 17, 27, 18, 26, 26, 26, 17, 17, 26, 26,
17, 27, 18, 26, 26, 25, 25, 26, 27, 7, 26, 18, 27, 26, 22,
26, 25, 26, 17, 17, 26, 25, 25, 24, 26, 26
```

Рисунок 21 – Токенизированный исходный код решений

Полученные токены объединяются в шинглы и находятся их контрольные суммы (смотреть рисунок 22).

```
Решение 1
-1685700393, 1651828613, -67181047, -237584135, -701207645,
2106332729, -1755293471, -1029461727, -3514144, 1736089857,
-347920615, -350576150, -1923606739, -1952205763, 1456191813,
-257665739, -1847642290, 402692180, -353367349, 1978591032,
-1243159127, -536214888, -1675068780, 1280344029, -
1413979739, -529379612, 116113596, 1149582500, 229509314,
369882463, ...
Решение 2
-1685700393, 1651828613, -67181047, -237584151, -701208165,
2106316609, -1755793192, -1044953102, -483746768, -266219617,
2004995131, -424632065, 75627212, 1208053926, 462834704, -
986965038, -483068144, -422341910, 1637330991, 1280344029, -
1413979739, ...
```

Рисунок 22 – Фрагменты наборов контрольных сумм шинглов

Для комментариев тоже вычисляются контрольные суммы шинглов. Вычислив оригинальность по алгоритму, описанному в третьем разделе, получим 71,(6)%. Следовательно, решения оригинальны, если сравнивать их друг с другом. Можно приступать к проверке стиля. В результате получим набор штрафных баллов для каждого решения. Для первого решения получим набор, состоящий только из одного числа – 12. Это решение имеет неполный оператор «for». Для второго решения получим набор: 10, 10, 10. Здесь в решении нарушена максимальная длина строки и размеры функциональных блоков (класса и метода).

Далее идёт подготовка изолированного окружения: проекты решений по очереди помещаются в общую с виртуальной машиной директорию. Извне виртуальной машине отдаётся команда компиляции исходного кода каждого решения. Синтаксических ошибок в решениях нет. Компиляция прошла успешно. Начинается проверка на наборе тестов. Данные решения прошли все тесты из набора, кроме последнего, поскольку данные решения могут работать только с целыми числами. В результате работы алгоритма тестирования (смотреть рисунок 16) оба решения набирают приблизительно 71 балл. Теперь в соответствии с формулой 3 необходимо вычесть из полученных баллов штрафы из набора для каждого решения. В результате итоговый балл для первого решения приблизительно равен 63, а для второго – 52.

В данном подразделе были рассмотрены результаты тестирования двух программных решений на каждом этапе и в конце получены итоговые оценки по 100-бальной шкале.

4.4 Оценка и анализ результатов тестирования разработанной модели процесса автоматической проверки программных решений

На основании результатов, полученных выше, по выделенным критериям проведём оценку разработанной модели процесса автоматической проверки программных решений (смотреть таблицу 15).

Таблица 15 – Оценка разработанной модели

Номер	Критерий	Оценка	Пояснение
К1	Безопасность процесса автоматической проверки	5	Использование виртуальной машины – лучший способ обезопасить автоматическую проверку от потенциально опасного кода, так виртуальная машина обеспечивает полную изоляцию от операционной системы узла.
К2	Гибкость методов оценки программного решения	4	Используя для оценки исходного кода регулярные выражения, можно реализовать автоматическую проверку большого спектра задач. Например, можно запретить использование каких-либо конструкций, функций. Однако метод проверки работы программы довольно примитивен: он не позволяет сравнивать пары ввод-вывод программы между собой. Это, например, затрудняет реализацию задачи «разработка хеш-функции», где неважно, что именно будет выводить проверяемая программа, а необходимо проверять решение на наличие коллизий.
К3	Степень независимости от языка программирования	4	Благодаря использованию регулярных выражений при работе с исходным кодом, не нужно вмешиваться в алгоритмы модели. Для проверки оригинальности программного решения и оценки его исходного кода, написанного на определённом языке программирования, достаточно просто предоставить модели необходимый набор регулярных выражений. Однако алгоритм проверки стиля исходного ориентирован по большей части на Си-подобные языки.
К4	Степень вычислительной нагрузки	3	Проверка при помощи регулярных выражений и использование виртуальных машин снижает производительность процесса автоматической проверки.
К5	Объективность итоговой оценки программного решения	5	Разработанная модель позволяет производить автоматическую проверку программных решений по всем критериям, которые были рассмотрены в подразделе 1.1.

Таким образом, на основании таблицы 15 у разработанной модели можно выделить следующие достоинства:

– в отличии от большинства систем автоматической проверки программных решений, где изолированное окружение использует операционную систему узла, или проверка потенциально

вредоносного кода отсутствует, в разработанной модели используются полностью изолированные виртуальные машины. Таким образом, если и произойдёт сбой автоматической проверки, то только внутри виртуальной машины, работа которой впоследствии будет восстановлена;

– в современных системах автоматической проверки программных решений оценка исходного кода программы студента в основном ограничивается лишь результатами компиляции. Помимо этого, оригинальность решения часто тоже не проверяется. Разработанная модель лишена этих недостатков, работа студента проверяется по всем критериям (смотреть подраздел 1.1), из-за чего оценка получается полной и объективной;
– благодаря использованию в разработанной модели регулярных выражений, для проверки оригинальности программного решения и оценки его исходного кода не нужно добавлять сторонние программы, как это делается в большинстве современных систем автоматической проверки программных решений. Таким образом, разработанная модель почти не зависит от языка программирования.

У разработанной модели был обнаружен один важный недостаток. Более низкая производительность. Известно, что использование виртуальных машин требует значительных вычислительных ресурсов. Это же касается и применение регулярных выражений. На их парсинг и поиск текста по паттерну требуется достаточное количество времени, что тоже сказывается на производительности автоматической проверки программных решений.

В данном подразделе была произведена оценка разработанной модели процесса автоматической проверки программных решений по каждому критерию, а также выделены её достоинства и недостатки.

4.5 Возможные перспективы дальнейшего развития разработанной модели процесса автоматической проверки программных решений

Процессы создания эталонного решения и тестового набора тесно переплетены: для того чтобы создать правильный тестовый набор, необходимо правильное эталонное решение, а для отладки эталонного решения требуется полный тестовый набор. Полнота тестового набора (полнота покрытия его тестами всех классов эквивалентности, на которые разбивается пространство возможных решений) является существенным условием для определения правильности проверяемого решения (в т. ч. и эталонного) [5].

Зачастую автором эталонного решения и тестового набора является один и тот же человек, причём некоторые тесты из «рабочего» тестового набора, использованного при отладке эталонного решения, затем входят и в итоговый тестовый набор [2].

Если для задачи по программированию специфицирован формат её входных данных, то процесс создания проверочных тестовых наборов можно частично автоматизировать (смотреть рисунок 23). Генератором выходных данных служит программа, являющаяся авторским решением исходной задачи. Также не менее важную роль играют спецификации формата, которые чаще всего описываются в тексте задачи. Сюда же относятся описания ограничений и условий, которым должны удовлетворять решения. Для извлечения спецификаций из текста задачи необходим текстовый анализ, однако его результаты всё равно подлежат проверке человеком, поэтому желательно, чтобы «внутренний» вид спецификаций, используемый генератором тестовых наборов, не слишком отличался от их «внешнего» вида, предназначенного для людей. Представляется, что для частного случая олимпиадных задач автоматическое извлечение спецификаций не составит проблемы, поскольку для них давно уже выработаны и стали общепринятыми правила описания форматов входных и выходных данных [1].

Таким образом, автоматическая генерация тестового набора при её грамотной реализации позволит сократить время, затрачиваемое преподавателем на создание задания.

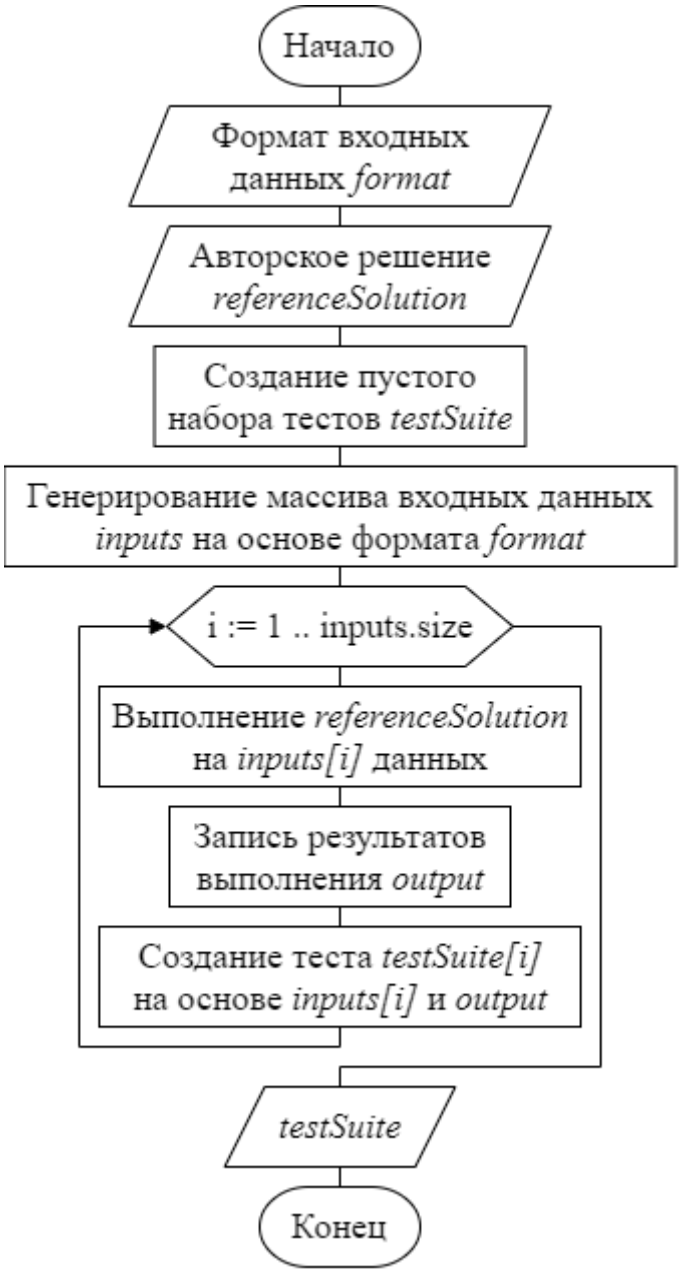

Рисунок 23 – Алгоритм генерации тестов

Также в будущем для автоматической проверки исходного кода программы студента можно будет применять алгоритмы машинного обучения. В настоящее время существует уже множество реализаций

статических анализаторов, основанных на или использующих машинное обучение, в том числе – глубокое обучение и NLP для обнаружения ошибок [21]. Такой подход позволит решить сразу несколько проблем:

- выявление ошибок, которые не видны компилятору (runtime errors). Например, можно будет сразу обнаружить участки кода, при исполнении которых возможна утечка памяти;
- проверка стиля исходного кода. Алгоритмы машинного обучения способны находить участки кода, которые не соответствуют общепринятым требованиям к их оформлению;
- предварительная проверка безопасности, посредством выявления потенциально опасных участков исходного кода;
- обнаружение неэффективного кода;
- выявление плагиата.

Таким образом, алгоритмы машинного обучения позволят давать более точную оценку программному решению студента. Также уменьшится время, затрачиваемое на автоматическую проверку исходного кода [23].

Немало важным улучшением полученной модели может стать выдача хорошо структурированного и понятного отчёта о результатах автоматической проверки, а не только оценки. Такой отчёт поможет студенту и преподавателю найти ошибку в решении. Стоит отметить, что отчёт не должен студенту прямо указывать на ошибку. Чтобы обучение было эффективным, студент должен уметь самостоятельно искать проблемы и ошибки в своём решении [16].

В четвёртом разделе были выделены критерии оценки модели процесса автоматической проверки программных решений, приведён пример задания по программированию, описаны данные, которые необходимы для проверки программных решений. Приведены два примера программных решений для этого задания. Поэтапно рассмотрена автоматическая проверка этих примеров, получены их итоговые оценки. В конце была произведена оценка разработанной модели, выделены её достоинства и недостатки, а также рассмотрены возможные пути её улучшения.

Заключение

В ходе работы были выделены основные критерии оценки программы студента: корректность исходного кода программы и её работы, оригинальность решения и безопасность его выполнения. Рассмотрены существующие системы автоматической проверки задач по программированию и то, как они оценивают решения учеников по выделенным критериям, описаны общие алгоритм работы и структура существующих систем. В результате были поставлены актуальные задачи, которые должна решать автоматическая проверка программных решений. В соответствии с поставленными задачами были исследованы различные методы и технологии как способы, которые можно использовать для автоматической проверки задач по программированию при дистанционном обучении. На основе проведённого анализа были предложены пути решения поставленных задач.

Был спроектирован процесс автоматической проверки программных решений. На ранних его этапах отсеиваются решения, которые являются попытками получения высокого балла обманным путём. В ходе основной проверки, используя разработанную систему штрафных баллов, оценивается стиль исходного кода решения, а также результаты выполнения программы на наборе тестов. В большинстве методов, которые используются для проверки, применяются наборы регулярных выражений, что позволяет масштабировать, расширять созданную модель, а выполнение внутри виртуальной машины позволяет обезопасить процесс от вредоносных программ и оценивать решения, написанные для разных операционных систем. В конце работы была произведена оценка разработанной модели по выделенным критериям, а также проанализированы её достоинства и недостатки.

Разработанная модель может быть применена для создания полноценной системы автоматической проверки задач по программированию при онлайн обучении.

Список используемой литературы и используемых источников

1. Аксенова Н. В., Диденко А. В. Эффективные методы и подходы для разработки электронного курса // Профессиональное образование в России и за рубежом. – 2016. – №2 (22). – С. 108-114.

2. Андреева, Т. А. Возможность автоматизации процесса генерирования тестовых наборов // Universum: технические науки. 2017. №8 (41). URL: https://cyberleninka.ru/article/n/vozmozhnost-avtomatizatsii-protsessa-generirovaniya-testovyh-naborov (дата обращения: 05.11.2019).

3. Бедердинова О. И., Бойцова Ю. А. Интегральная оценка качества программных средств // Arctic Evironmental Research. 2016. №2. URL: https://cyberleninka.ru/article/n/integralnaya-otsenka-kachestva-programmnyh-sredstv (дата обращения: 29.11.2019).

4. Блоховцова Г. Г., Волохатых А. С. Перспективы развития дистанционного образования. Преимущества и недостатки // Символ науки. 2016. №10-2. [Электронный ресурс]. URL: http://cyberleninka.ru/article/n/perspektivy-razvitiyadistantsionnogo-obrazovaniya-preimuschestva-i-nedostatki (дата обращения: 08.01.2018).

5. Введение в программные системы и их разработку [Электронный ресурс] : [учеб. пособие] / С. В. Назаров [и др.]. – 2-е изд., испр. – Москва : ИНТУИТ, 2016. – 649с. : ил.

6. Гладких И. Ю., Якушин А. В. Системы автоматизированного тестирования по программированию в образовательном пространстве // Современные проблемы науки и образования. – 2016. – № 3.; URL: http://www.science-education.ru/ru/article/view?id=24719 (дата обращения: 27.10.2019).

7. Голодок Д. А., Алексеев В. М. Преимущества дистанционного обучения // Инновационная наука. 2016. №11-2. URL: https://cyberleninka.ru/article/n/preimuschestva-distantsionnogo-obucheniya-1 (дата обращения: 03.12.2019).

8. Горчаков Л. В., Стась А. Н., Карташов Д. В. Обучение программированию с использованием системы Ejudge // Вестник ТГПУ. 2017. №9 (186). URL: https://cyberleninka.ru/article/n/obuchenie-programmirovaniyu-s-ispolzovaniem-sistemy-ejudge (дата обращения: 27.10.2019).

9. Евстропов, Г. О. Системы оценивания в задачах с автоматической проверкой на олимпиадах по программированию / Г. О. Евстропов // Информатика и образование. – 2016. – № 3 (272). – С. 65-67.

10. Князева Е. В., Попова Г. И. Мобильность педагогического образования на примере обучения языкам программирования // ИСОМ. 2016. №5-1. С. 167-170.

11. Лишманова Н. А., Пимичева М. А. Дистанционное обучение и его роль в современном мире // Научно-методический электронный журнал «Концепт». – 2016. – Т. 11. – С. 2216–2220. – URL: http://e-koncept.ru/2016/86472.htm.

12. Макиева, З. Д. Проектирование автоматизированной системы проверки олимпиадных заданий по программированию / З. Д. Макиева // Известия Кыргызского государственного технического университета им. И. Раззакова. – 2016. – Т. 38. – С. 54-61.

13. Мокрый, В. Ю. О преподавании дисциплины «Информатика» студентам гуманитарного вуза с помощью системы дистанционного обучения Moodle // Вестник ТГПУ. 2017. №9 (186). URL: https://cyberleninka.ru/article/n/o-prepodavanii-distsipliny-informatika-studentam-gumanitarnogo-vuza-s-pomoschyu-sistemy-distantsionnogo-obucheniya-moodle (дата обращения: 03.12.2019).

14. Петракова, Н. В. Информационные технологии дистанционного обучения // Сборник научных трудов института энергетики и природопользования. Брянск, 2017. С.171-174.

15. Привалов А. Н., Гладких И. Ю. Принципы построения и реализация системы автоматизированного тестирования решений задач по программированию // Известия ТулГУ. Технические науки. 2016. №2. URL:

https://cyberleninka.ru/article/n/printsipy-postroeniya-i-realizatsiya-sistemy-avtomatizirovannogo-testirovaniya-resheniy-zadach-po-programmirovaniyu (дата обращения: 27.10.2019).

16. Самощенко, Ю. Ю. Исследование эффективности автоматизированной проверки решений при проведении олимпиад по программированию / Ю. Ю. Самощенко // Молодой ученый. – 2016. – № 11. – С. 223-226.

17. Сверчкова, Г. В. Автоматизированная система проверки результатов олимпиады по программированию / Г. В. Сверчкова, Д. И. Кислицын // Сборник статей студ., аспирантов и магистр. «Информационные системы и технологии». – 2016. – С. 30-34.

18. Семенова З. В., Любич С. А., Кузнецов А. Г., Мальцев П. А. Сравнительная характеристика средств автоматизированной проверки правильности составления SQL-запросов // Вестник СибАДИ. 2017. №3 (55). URL: https://cyberleninka.ru/article/n/sravnitelnaya-harakteristika-sredstv-avtomatizirovannoy-proverki-pravilnosti-sostavleniya-sql-zaprosov (дата обращения: 05.11.2019).

19. Серегина, Е. А. Технология дистанционного обучения как способ получения высшего образования в России // ПНиО. 2018. №2 (32). URL: https://cyberleninka.ru/article/n/tehnologiya-distantsionnogo-obucheniya-kak-sposob-polucheniya-vysshego-obrazovaniya-v-rossii (дата обращения: 03.12.2019).

20. Симуни М. Л., Соловьев А. Ю., Шайтан В. И. Автоматизированная проверка задач в курсе «Функциональное программирование» // Современные информационные технологии и ИТ-образование. 2016. №3-1. URL: https://cyberleninka.ru/article/n/avtomatizirovannaya-proverka-zadach-v-kurse-funktsionalnoe-programmirovanie (дата обращения: 27.10.2019).

21. Слепцова Л. Н., Николаев Е. В. Внедрение системы дистанционного обучения Moodle для работы со студентами с индивидуальным графиком обучения как ключевое направление развития

образовательных учреждений профессионального образования // Kant. 2017. №4 (25). URL: https://cyberleninka.ru/article/n/vnedrenie-sistemy-distantsionnogo-obucheniya-moodle-dlya-raboty-so-studentami-s-individualnym-grafikom-obucheniya-kak-klyuchevoe (дата обращения: 03.12.2019).

22. Шагилова, Е. В. Архитектура и реализация программы онлайн-тестировщика для проверки решений задач по олимпиадному программированию // Вестник ДГТУ. 2018. №2. URL: https://cyberleninka.ru/article/n/arhitektura-i-realizatsiya-programmy-onlayn-testirovschika-dlya-proverki-resheniy-zadach-po-olimpiadnomu-programmirovaniyu (дата обращения: 05.11.2019).

23. Щанникова Е. А., Сараева Е. В. Дистанционное обучение как важная составляющая внедрения информационных технологий в образовательный процесс // Научно-методический электронный журнал «Концепт». – 2016. – Т. 32. – С. 233–237. – URL: http://e-koncept.ru/2016/56691.htm.

24. Якушин А. В., Гладких И. Ю. Выбор системы автоматизированного тестирования решений задач по программированию // International Journal of Open Information Technologies. 2016. №6. URL: https://cyberleninka.ru/article/n/vybor-sistemy-avtomatizirovannogo-testirovaniya-resheniy-zadach-po-programmirovaniyu (дата обращения: 27.10.2019).

25. Aldriye H., Alkhalaf A., Alkhalaf M. Automated Grading Systems for Programming Assignments: A Literature Review [Электронный ресурс] // International Journal of Advanced Computer Science and Applications. 2019. Vol. 10.3. PP. 215-222. URL: https://thesai.org/Downloads/Volume10No3/Paper_28-Automated_Grading_Systems.pdf (дата обращения: 03.12.2019).

26. Bennouar D. An Automatic Grading System Based on Dynamic Corpora [Электронный ресурс] // The International Arab Journal of Information Technology. 2017. Vol. 14.4A PP. 552-564. URL: https://pdfs.semanticscholar.org/f188/a42968741ca733178701766d1eebb9f0a410.pdf (дата обращения: 24.10.2019).

27. Earle C. B., Fredlund L-A., Hughes J. Automatic Grading of Programming Exercises using Property-Based Testing [Электронный ресурс] // Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education. 2016. PP. 47-52. URL: https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/automatic-grading.pdf (дата обращения: 24.10.2019).

28. Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. “Towards a Systematic Review of Automated Feedback Generation for Programming Exercises–Extended Version”. In: (2016).

29. Liu X., Wang S., Wang P., Wu D. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics [Электронный ресурс] // Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training. 2016. PP. 126-137. URL: https://faculty.ist.psu.edu/wu/papers/autograder.pdf (дата обращения: 24.10.2019).

30. Pedro Antonio Gutiérrez et al. “Ordinal regression methods: survey and experimental study”. In: IEEE Transactions on Knowledge and Data Engineering 28.1 (2016), pp. 127–146.