В. Д. Струнин, Е. Е. Бовыкина, Е. А. Павлова
Тюменский государственный университет, г. Тюмень
УДК 004.4

РАЗРАБОТКА АЛГОРИТМА ПРОВЕРКИ ПРОГРАММНЫХ КОДОВ СТУДЕНТОВ ПО ДИСЦИПЛИНЕ «ПРОГРАММИРОВАНИЕ И ОСНОВЫ АЛГОРИТМИЗАЦИИ»

Аннотация. Статья посвящена разработке алгоритма для автоматизированной проверки программных кодов студентов на языке программирования Python. Акцент сделан на комбинированном подходе, включающем автоматизированное тестирование stdin/stdout, структурный анализ через абстрактное синтаксическое дерево (AST), мутационное тестирование и генерацию референсных решений с помощью ChatGPT-4o.

Ключевые слова: автоматизированная проверка; AST-анализ; мутационное тестирование; автоматизированная генерация тестов; анализ программного кода.

Введение. С развитием цифровых технологий автоматизация процессов обучения становится одним из ключевых направлений модернизации образования. Особенно актуальной эта тенденция является в контексте преподавания программирования, где проверка студенческих решений отнимает значительное количество времени и требует высокой вовлеченности преподавателя. В высших учебных заведениях, в том числе и в Тюменском государственном университете (ТюмГУ), преподаватели часто проверяют студенческие работы вручную — через демонстрацию студентами решений на занятиях или загрузку файлов на LMS-платформу. Такой подход ограничен во времени, не всегда позволяет предоставить быструю обратную связь, а также увеличивает нагрузку на преподавателя, снижая объективность оценки.

За последние годы было предложено множество решений, направленных на автоматизацию проверки программных заданий. Распространенным является метод автоматического тестирования с использованием заранее подготовленных входных и выходных данных [1]. Однако такие решения не учитывают оригинальность и логику выполнения кода, ограничиваясь проверкой соответствия заданному результату. Современные платформы для обучения программированию, такие как Codeforces, LeetCode, Codewars в основном применяют только этот метод, но это лишь проверяет соответствие выходных данных заданным требованиям. В научной литературе и практических исследованиях рассматриваются более продвинутые методы анализа студенческого кода. Например, использование AST (Abstract Syntax Tree) позволяет сравнивать структуру программ и выявлять схожесть или различия решений независимо от незначительных изменений в коде [2-3]. Метод эталонного кода и модели ошибок, предложенный в Массачусетском технологическом институте, автоматически предлагает исправления для типичных ошибок студентов и формирует пояснения к ним [4]. Использование языковых моделей, таких как GPT-3.5, также демонстрирует потенциал в анализе кода и генерации обратной связи, хотя требует адаптации под специфику учебных заданий [5]. Дополнительно перспективным направлением является мутационное тестирование, ориентированное на типичные ошибки обучающихся и позволяющее не только проверять корректность, но и объяснять причины ошибок [6].

Таким образом, подход к автоматизированной проверке программных решений должен сочетать несколько методов: структурный анализ, оценку логики, сравнение с референсным решением и генерацию объяснений ошибок.

Цель работы — разработка алгоритма, обеспечивающего автоматизированную проверку программных решений студентов на соответствие заданию и установленным ограничениям, с возможностью генерации комментариев и указаний на возможные ошибки. Для достижения цели решаются следующие задачи:

1. Формирование базы размеченных данных на основе программных кодов студентов из отчетов по дисциплине «Программирование и основы алгоритмизации 2023-2024» для тестирования алгоритмов проверки кода.

2. Формирование базы референсных кодов с помощью языковой модели (ChatGPT-4o)

3. Разработка алгоритма проверки программного кода студентов с генерацией тестов, поиском исправления ошибок и сравнением структуры с референсным кодом

Ограничения алгоритма.

- Поддержка проверки кода только на языке программирования Python.
- Не должны учитываться несущественные различия с референсным кодом.
- Генерация референсного решения с использованием запроса в большую языковую модель (LLM) разрешена только для текстовых заданий.
- Алгоритм тестирования программного кода должен основываться исключительно на проверке соответствия.

Разработка алгоритма автоматизированной проверки. Алгоритм проверки кодов студентов включает несколько модулей, каждый из которых решает конкретную задачу: генерация тестов, анализ результатов, структурное сравнение кода и поиск ошибок.

1. Тестирование stdin/stdout с генерацией тестов

На основе референсного решения и заданных ограничений формируются тестовые наборы данных, содержащие входные и выходные параметры.

Математическая постановка задачи автоматизированной генерации тестовых наборов:

Дано:

1. $T = \{t_1, t_2, \ldots, t_n\}$ — множество заданий в текстовом формате.

2. $B_i = \{b_{i1}, b_{i2}, \ldots, b_{im}\}$ — множество ограничений входных данных задания t_i.

Требуется: для каждого задания t_i из множества T сформировать множество тестовых наборов $H_i = \{h_{i1}, h_{i2}, \ldots, h_{ig}\}$, удовлетворяющих ограничениям B_i. Формирование тестовых наборов происходит с помощью генерации случайных чисел с последующей проверкой ограничений B_i.

2. Сравнение структуры кода

Проверка заключается в сравнении узлов AST референсного решения с узлами AST кода студента с применением метода TSED (Tree Similarity Edit Distance) [7], чтобы определить соответствие структуры исходному заданию. Если схожесть превышает порог в 70%, код считается корректным; в противном случае назначается метка для дополнительной проверки преподавателем. Метод TSED использует обход дерева для извлечения всех узлов из AST. На основе узлов двух деревьев формируется матрица стоимостей, где каждая стоимость отражает степень различия между узлами. Затем применяется алгоритм минимального назначения для сопоставления узлов с минимальной суммарной стоимостью, добавляются штрафы за несовпадающие узлы и отсутствие важных (в контексте задачи) конструкций, а также поощрения за их наличие. Итоговая стоимость нормализуется и преобразуется в процент сходства между структурами кода.

Математическая постановка задачи сравнения структуры программных кодов:

Дано:

1. $T = \{t_1, t_2, ..., t_n\}$ — множество заданий в текстовом формате.

2. $C_i = \{c_{i1}, c_{i2}, ..., c_{ik}\}$ — множество решений задания t_i, отправленных студентами (программный код на языке Python в текстовом формате).

3. $R = \{r_1, r_2, ..., r_n\}$ — множество корректных решений (референсных кодов на языке Python в текстовом формате) заданий из множества T.

Требуется: для каждого решения задания t_i получить множество $S_i = \{s_{i1}, s_{i2}, ..., s_{ik}\}$, где s_{ij} — процент структурного сходства программного кода c_{ij} и референсного кода r_i.

Вычисление процента структурного сходства s_{ij}:

$$s_{ij}(c_{ij}, r_i) = 100 * \left(1 - \frac{d(c_{ij}, r_i)}{\max\left(n(c_{ij}), n(r_i)\right)}\right) \quad (1)$$

где

- $d(c_{ij}, r_i)$ — минимальное количество действий для преобразования AST (Abstract Syntax Tree) кода c_{ij} в AST кода r_i.
- $n(c_{ij})$ — количество узлов AST кода c_{ij}.
- $n(r_i)$ — количество узлов AST кода r_i.

Вычисление минимального кол-ва действий $d(c_{ij}, r_i)$:

Метод TSED использует обход дерева (ast.walk) для извлечения всех узлов из AST. На основе узлов двух деревьев формируется матрица стоимостей, где каждая стоимость отражает степень различия между узлами (например, разные типы узлов имеют максимальную стоимость).

$$Cost\left(u(c_{ij})_p, u(r_i)_m\right) == \begin{cases} 1, & \text{типы узлов различны} \\ 0.5 * |w(u(c_{ij})_p - w(u(r_i)_m)|, & \text{типы узлов совпадают} \\ 0, & \text{узлы идентичны} \end{cases} \quad (2)$$

где

- $u(c_{ij})_p$, $u(r_i)_m$ — узлы из двух деревьев (AST кода c_{ij} и AST кода r_i), которые сравниваются.
- 1 — максимальная стоимость, которая присваивается, если узлы имеют разные типы. Это означает, что они считаются полностью несоответствующими.
- $= \begin{cases} 1, & \text{типы узлов различны} \\ 0.5 * |w(u(c_{ij})_p - w(u(r_i)_m)|, & \text{типы узлов совпадают} \\ 0, & \text{узлы идентичны} \end{cases}$ — половина абсолютной разницы между весами узлов $u(c_{ij})_p$ и $u(r_i)_m$. Применяется, если типы узлов совпадают, но их свойства отличаются. Эта часть формулы оценивает степень различий в их свойствах.
- 0 — стоимость равна нулю, если узлы идентичны по типу и свойствам.

После формирования матрицы стоимостей с помощью алгоритма оптимального назначения рассчитывается $d(c_{ij}, r_i)$. Дополнительно итоговое значение $d(c_{ij}, r_i)$ корректируется с учетом следующих факторов:

- Штрафы: начисляются за наличие важных конструкций в одном AST и их отсутствие в другом.
- Поощрения: применяются, если важные конструкции присутствуют в обеих структурах.

3. Мутационное тестирование

Метод основан на генерации мутантов — измененных версий кода, которые проходят те же тесты, что и исходный код. Если мутант успешно проходит тесты, это указывает на потенциальную ошибку в месте изменения [8].

Математическая постановка задачи мутационного тестирования:

Дано:

- $T = \{t_1, t_2, ..., t_n\}$ — множество заданий в текстовом формате.
- $C_i = \{c_{i1}, c_{i2}, ..., c_{ik}\}$ — множество решений задания t_i, отправленных студентами (программный код на языке Python в текстовом формате).
- $M = \{m_1, m_2, ..., m_z\}$ — множество возможных модификаций для программного кода (например, замена операторов)
- $H = \{h_1, h_2, ..., h_{ig}\}$ — множество тестовых наборов задания t_i.

Требуется: для каждого решения c_{ig} получить множество программных кодов $F_{ij} = \{f_{ij1}, f_{ij2}, ..., f_{ijz}\}$, содержащее все модификации множества M, причем каждый элемент множества F_{ij} должен успешно проходить все тестовые наборы множества H_i.

4. Генерация референсного кода

Для генерации референсного кода используется языковая модель ChatGPT-4o. При добавлении задания запрос с подробным prompt отправляется через OpenAI API. Prompt содержит такие элементы как: роль модели, цель, ограничения, контекст задачи, описание задания (формулировка).

Сгенерированный референсный код используется для генерации входных и выходных данных для автоматизированного тестирования и служит основой для AST-сравнения.

Данные для оценки точности работы алгоритма. Для оценки алгоритма проверки был собран датасет из 3362 студенческих отчетов (6579 программных кодов на языке Python) студентов первого курса по дисциплине «Программирование и основы алгоритмизации» за 2023-2024 год. Коды извлекались автоматически с помощью разработанного парсера. Каждый код проверялся автоматически и привязывался к конкретному заданию. Дополнительно анализировались типичные ошибки студентов для формирования кейсов мутационного тестирования.

Результаты и обсуждение. В ходе работы разработан алгоритм, обеспечивающий: проверку программных кодов студентов с предоставлением обратной связи

Для анализа работы алгоритма было отобрано 2009 программных кодов, содержащих решение заданий из трех практических работ в соотношении 944 правильных и 1065 неправильных. Из отобранных неправильных программных кодов 596 содержали ошибки в логике

работы программы, 469 содержали использование некорректных методов в контексте решения задания.

Из фактически верных 944 кодов алгоритм проверки верно классифицировал 867, что соответствует 91%. Неверно классифицированы 77 кодов, что составляет 9%. Из числа фактически неверных кодов, верно классифицированы 1043 программных кода студентов, что составляет 98%. Неверно классифицированы 2% решений. В табл. 1 представлена матрица ошибок классификации всех отобранных программных решений студентов.

Таблица 1

Матрица ошибок

Матрица ошибок		*фактическое*	
		Positive	*Negative*
Предположение	Positive	867	22
	Negative	77	1043

Алгоритм проверки в большинстве случаев верно определяет использование некорректных методов решения задач, опираясь на заданный контекст практической работы. В таких случаях итоговый процент сходства снижается до значения менее 40% при установленном пороге допустимого сходства в 70%.

Важно отметить, что алгоритм чувствителен к порядку входных данных: он ожидает, что в студенческом коде они будут использоваться в том же порядке и виде, как это указано в ограничениях на генерацию тестовых данных. Иногда это приводит к получению обратных значений — например, противоположных логических результатов (True вместо False) или чисел с неверным знаком. В результате такие решения ошибочно классифицируются как неверные, даже если логика работы кода правильная.

Кроме того, алгоритм строго учитывает формат вывода. Например, если в студенческом решении используется округление, и оно не совпадает с ожидаемым форматом, программа также считается некорректной, несмотря на правильность расчетов.

Классификация решений при помощи алгоритма автоматизированной проверки показало, что средняя точность классификации составляет 95%.

Точность проверки по выбранным практическим работам представлена в табл. 2.

Таблица 2

Метрики работы алгоритма

Тема	*Математические функции и выражения*		*Оператор условия*		*Оператор цикла while*	
	Positive	*Negative*	*Positive*	*Negative*	*Positive*	*Negative*
Accuracy	0.90		0.99		0.97	
Precision	0.99	0.83	0.99	1.0	1.0	0.92
Recall	0.85	0.99	0.99	1.0	0.95	1.0
F1	0.92	0.90	0.99	1.0	0.98	0.96

Ошибки проверки чаще возникают в работах по «Математическим функциям и выражениям» из-за вариативности решений и короткого кода, что искажает сравнение AST. Лучшие результаты — в заданиях на «Условия» и «Циклы while», где код более однообразен.

Точность проверки зависит от корректности референсного решения и ограничений, заданных преподавателем. Ошибки в эталонном коде (например, несоответствие требованиям) ведут к ложному занижению оценки студенческих работ. Аналогично, некорректные тестовые данные могут помешать проверке даже верных решений.

Мутационное тестирование эффективно лишь на 54% из-за разнообразия ошибок. Данный метод не позволяет обнаружить ошибки, допущенные по невнимательности (например, опечатки), и требует много времени при увеличении вариаций, что особенно заметно в задачах, где используются циклы.

Заключение. Разработан алгоритм, который позволяет проверять студенческие программы на Python. Алгоритм проверки объединил несколько подходов: генерацию тестов, структурное сравнение кода через AST, мутационное тестирование и возможности больших языковых моделей для создания референсных решений. Такой комплексный подход алгоритма позволил добиться точности проверки в 95%.

Проверка точности работы на реальных студенческих решениях показала, что сервис может определять ошибки в логике, использованные некорректных методов и инструментов.

Кроме того, в процессе были выявлены важные замечания: например, что на результат влияет порядок ввода данных или даже лишний пробел в выводе. Такие детали открыли новые идеи для доработки — более гибкая настройки проверок и улучшения кейсов типичных ошибок.

Планируется добавить и протестировать задания из других практических работ дисциплины «Программирование и основы алгоритмизации».

СПИСОК ЛИТЕРАТУРЫ

1. Алексеев Ю. Е., Куров А. В. Автоматизация тестирования студенческих программ //Инженерный журнал: наука и инновации. — 2013. — № 6 (18). — С. 5.
2. Wen W. et al. Code similarity detection using ast and textual information //International Journal of Performability Engineering. — 2019. — Т. 15. — № 10. — С. 2683.
3. Lee G. et al. Review of code similarity and plagiarism detection research studies //Applied Sciences. — 2023. — Т. 13. — № 20. — С. 11358.
4. Singh R. Automated Feedback Generation for Introductory Programming Assignments / R. Singh, S. Gulwani, A. Solar-Lezama. — URL: https://arxiv.org/abs/1204.1751 (date of the application: 16.11.2012). — Text: electronic.
5. Мальцев В. М., Назаров И. А., Павлова Е. А. Исследование применимости генеративной нейронной сети в качестве классификатора решений заданий по программированию // Математическое и информационное моделирование: материалы Всероссийской конференции молодых ученых. — Вып. 22. — Тюмень: ТюмГУ-Press, 2024.
6. Prasad S. et al. Conceptual Mutation Testing for Student Programming Misconceptions //arXiv preprint arXiv:2401.00021. — 2023.
7. Song Y., Lothritz, C., Tang, D., Bissyandé, T. F., Klein, J. Revisiting Code Similarity Evaluation with Abstract Syntax Tree Edit Distance., 2024, arXiv:2404.08817v2. URL: https://arxiv.org/abs/2404.08817v2 (дата обращения: 06.03.2025).
8. Rezaalipour M., Furia C. A. An Empirical Study of Fault Localization in Python Programs., 2024, arXiv:2305.19834v3. URL: https://arxiv.org/abs/2305.19834v3 (дата обращения: 06.03.2025).