

УДК 004.4'4, 004.272

АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ И ПРОВЕРКА ЗАДАЧ ПО ПРОГРАММНОЙ РЕАЛИЗАЦИИ И ТЕСТИРОВАНИЮ МАТРИЧНЫХ ПРЕОБРАЗОВАНИЙ НА ОСНОВЕ МУТАЦИОННОГО АНАЛИЗА

Горчаков А. В., к.т.н., доцент кафедры корпоративных информационных систем Института информационных технологий, Федеральное государственное бюджетное образовательное учреждение высшего образования «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78), e-mail: gorchakov@mirea.ru

Советов П. Н., к.т.н., доцент кафедры корпоративных информационных систем Института информационных технологий, Федеральное государственное бюджетное образовательное учреждение высшего образования «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78), e-mail: sovetov@mirea.ru

Актуальность работы обусловлена необходимостью развития профессиональных навыков разработчиков программного обеспечения (ПО) в области практической реализации матричных преобразований, лежащих в основе современных алгоритмов интеллектуального анализа данных и машинного обучения, а также навыков в области тестирования ПО для раннего выявления дефектов и ошибок в нём в соответствии рекомендациям, закреплённым в ГОСТ Р 56920–2024 и ГОСТ Р 56939–2024. Целью работы является автоматизация порождения задач на тему программной реализации и тестирования матричных преобразований с поддержкой автоматической проверки их программных решений. Научная новизна предлагаемого метода заключается в совместном использовании: системы переписывания термов с ограничениями, учитывающими специфику матричных преобразований; мутационного анализа на основе заранее заготовленных модифицированных программ для оценки качества реализованных тестирующих функций; интерпретации деревьев выражений с полной и пониженной точностью матричных операций для генерации кроссплатформенных наборов тестов. Использование предложенного метода в учебном процессе позволило освободить преподавателей от необходимости проверки 12269 программ, присланных студентами за семестр. Применение мутационного анализа как отдельного этапа автоматической проверки программ обеспечило повышение устойчивости задач к их решению большими языковыми моделями с методом zero-shot до 80% и повысило число отклонённых программ на 33,5% от общего числа проверенных программ по сравнению с алгоритмом автоматической проверки без мутационного анализа. В результате работы учащихся над повышением качества тестирующих функций в отклонённых программах среднее число выполнений оператора assert в принятых программах увеличилось в 2,24 раза. При этом время генерации индивидуальных задач для более 1400 учащихся с 5 тестами для автоматической проверки и 3 модифицированными программами для мутационного анализа, выполненной при помощи программного средства, реализованного на языке Python, не превышает нескольких минут, а время проверки и мутационного анализа одной программы в «песочнице», реализованной в виде Docker-контейнера с изоляцией, составляет в среднем 8,4 секунды. Полученные результаты могут использоваться в рамках учебных дисциплин с большим числом студентов, в массовых онлайн-курсах, при проведении соревнований по программированию.

Ключевые слова: алгоритмы генерации учебных задач, онлайн-образование, линейная алгебра, мутационный анализ, мутационное тестирование, модульное тестирование, программирование, numpy

AUTOMATED GENERATION AND ASSESSMENT OF EXERCISES ON SOFTWARE IMPLEMENTATION AND TESTING OF MATRIX TRANSFORMATIONS USING MUTATION ANALYSIS

Gorchakov A. V., Cand. Sci. (Eng.), Associate Professor of the ERP Systems Department, Institute of Information Technologies, Federal State Budget Educational Institution of Higher Education "MIREA – Russian Technological University" (78 Vernadskogo pr., Moscow, 119454 Russia), e-mail: gorchakov@mirea.ru

Sovietov P. N., Cand. Sci. (Eng.), Associate Professor of the ERP Systems Department, Institute of Information Technologies, Federal State Budget Educational Institution of Higher Education "MIREA – Russian Technological University" (78 Vernadskogo pr., Moscow, 119454 Russia), e-mail: sovetov@mirea.ru

Matrix transformations form the foundation of modern data mining and machine learning algorithms. The relevance of the study stems from the necessity to enhance the skills of novice programmers in software implementation of matrix transformations, as well as to strengthen their software testing skills for the early

detection of defects and errors, in compliance with the guidelines established in GOST R 56920–2024 and GOST R 56939–2024. The aim of the research is the automation of the generation and assessment process of exercises on the software implementation and testing of matrix transformations. The scientific novelty of the method proposed in this paper consists in the synergistic combination of the following components: a term rewriting system with constraints tailored to the specific nature of matrix transformations; mutation analysis based on prepared programs with injected mutations used to evaluate the quality of students' testing functions; interpretation of expression trees with both full and reduced precision of matrix operations to generate cross-platform test suites. The application of the proposed method in the educational process freed programming instructors from the burden of manually checking 12269 programs submitted by students over the course of a single semester. Incorporating mutation analysis as a distinct stage of the automated assessment pipeline has improved the robustness of exercises against solutions generated by large language models using the zero-shot approach, achieving an 80% resistance rate. Furthermore, it increased the absolute rejection rate of programs by 33.5% when compared to the automated assessment algorithm operating without mutation analysis. As a result of students' work on enhancing the quality of their testing functions in initially rejected programs, the average number of assert statement executions in accepted programs increased by a factor of 2.24. The time required to generate unique exercises for over 1400 students using a software tool implemented in Python does not exceed several minutes, every exercise consisted of 5 tests for automated assessment and 3 modified programs for mutation analysis. The average time consumed for checking and mutation analysis of a single program within the sandbox environment, implemented as an isolated Docker container, is 8.4 seconds. The obtained results can be used in **academic courses, in open online courses, and in programming competitions.**

Keywords: algorithms for generating educational tasks, online education, linear algebra, mutation analysis, mutation testing, unit testing, programming, numpy

Введение

Основу алгоритмического и программного обеспечения (ПО) современных вычислительных систем, создаваемых для задач высокопроизводительной обработки данных и машинного обучения, составляют операции над векторами, матрицами и тензорами. В частности, тензорные операции используются в нейросетевых алгоритмах [1], в алгоритмах нелинейного снижения размерности [2] и в алгоритмах решения систем линейных уравнений. При этом сложность современных вычислительных задач и необходимость работы с вещественным типом данных предъявляет жёсткие требования к надёжности программного кода, реализующего указанные методы и алгоритмы. В этой связи критически важным становится применение модульного тестирования непосредственно в процессе разработки ПО для раннего выявления и устранения, например, накапливающихся ошибок округления.

Главной целью тестирования ПО, как отмечено в ГОСТ Р 56920 – 2024, является выявление дефектов и ошибок в ПО с целью их исправления для эффективного использования ПО по назначению в конкретной системе. Одним из распространенных типовых подходов к тестированию ПО является автоматизированное тестирование и, в частности, модульное тестирование [3]. Модульное тестирование позволяет автоматизировать процесс проверки модулей ПО на соответствие требованиям, зафиксированным в виде набора контрольных примеров разработчиком ПО или специалистом в области тестирования [4]. В ГОСТ Р 56939 – 2024 закреплены общие требования к содержанию и порядку выполнения работ, связанных с созданием безопасного ПО и устранением выявленных недостатков ПО, включающие проведение фаззинг-тестирования вместо использования составленного вручную набора контрольных примеров. Фаззинг-тестирование основано на передаче программе автоматически сгенерированных случайных структурированных входных данных с целью обнаружения недостатков в ПО [5, 6].

Для оценки качества тестирования модулей ПО используют тестовое покрытие [5], [7]. В ГОСТ Р 56921 – 2016 тестовое покрытие определяется как степень, выраженная в процентах, в которой элементы тестового покрытия были проверены контрольными примерами. Элементами тестового покрытия могут быть функции, строки программного кода, операторы, однако наиболее строгой метрикой является метрика тестового покрытия ветвей графа потока управления программы (англ. branch coverage) [8]. Полное тестовое покрытие ветвей графа потока управления гарантирует достижимость всех линейных участков графа и выполнение всех операторов, расположенных в его линейных участках. Однако ошибки, зависящие от данных, поступающих на вход операторам в программе, могут остаться не обнаруженными даже при 100%-м тестовом покрытии ветвей набором контрольных примеров. С целью преодоления этой проблемы всё чаще применяются подходы к оценке качества тестов на основе мутационного анализа [9, 10]. Мутационный анализ – это метод оценки качества набора контрольных примеров, основанный на внедрении в тестируемый модуль ПО случайных изменений (мутаций). Если заданный набор контрольных примеров оказался неспособен обнаружить внедрённые мутации, он рассматривается как недостаточный [11]. Одной из широко используемых в мутационном анализе метрик

является отношение числа обнаруженных набором контрольных примеров мутаций к общему числу мутаций, внедрённых в тестируемый модуль ПО [12], выраженное в процентах.

Вследствие увеличения численности студентов в высших учебных заведениях на специальностях, связанных с программной инженерией, и в связи с участвовавшими случаями заимствования решений учебных задач, широкое распространение получили модели и средства автоматической генерации заданий для учебных дисциплин [13]. Помимо классических методов генерации заданий на основе шаблонов [14] и порождающих грамматик [15], всё чаще используются методы генерации заданий на основе больших языковых моделей (Large Language Models, LLM). Однако, как отмечено в [13], критическим недостатком подходов на основе LLM является вероятностная природа LLM и, следовательно, возможность генерации результатов с искажениями, что требует обязательной проверки результатов генерации преподавателем и затрудняет применение LLM при построении сборников задач для потоковых дисциплин.

Известно о подходах и о технических решениях, используемых преподавателями для обучения студентов модульному тестированию ПО. Так, в [16] для оценки качества написанных учащимися модульных тестов предлагается использовать метрику тестового покрытия, а в [17] предлагается оценивать, найдёт ли написанный одним студентом тест ошибки в модулях ПО, написанных другими студентами курса и решающих одну и ту же задачу. В [18] рассматриваются вопросы использования мутационного анализа для оценки качества модульных тестов, написанных учащимися, при этом показывается, что существующие открытые инструменты для мутационного анализа внедряют ошибки в тестируемые модули ПО не хуже, чем преподаватели, вручную составляющие наборы ошибочных программ для оценки качества модульных тестов.

Однако, открытые инструменты для мутационного анализа внедряют синтаксические мутации в тестируемые модули ПО случайным образом, что может привести к порождению программ с мутациями, результат выполнения которых не отличается от результата выполнения тестируемого модуля ПО, для которого учащийся пишет модульные тесты [19]. При этом отклонение эквивалентных оригинальному модулю ПО программ с мутациями во время автоматической проверки кода, написанного студентом, и повторное внедрение синтаксических мутаций в тестируемый модуль ПО требует значительных вычислительных затрат [20]. Кроме того, открытые инструменты для мутационного анализа ограничены синтаксическими преобразованиями программ и не внедряют предметно-ориентированные мутации, такие как, например, добавление или удаление унарной операции транспонирования матрицы или вектора.

Для устранения указанного противоречия предлагается метод автоматической генерации и проверки задач по программированию, научная новизна которого состоит в совместном использовании: системы переписывания термов с ограничениями, учитывающими специфику матричных преобразований при генерации деревьев выражений; мутационного анализа для оценки качества реализованных учащимися тестирующих функций на основе заранее заготовленных модифицированных программ, результат выполнения которых отличается от результата выполнения тестируемого модуля ПО; интерпретации деревьев выражений с полной и пониженной точностью операций для генерации кроссплатформенных наборов тестов и программ с мутациями, не являющихся семантически эквивалентными оригинальной программе, реализованной учащимся.

Архитектура генератора задач

Предлагаемый в данной работе метод основан на подходе к автоматической генерации задач по программированию, предложенном в [21] и использованном в [22] и [23] для автоматизации процесса генерации индивидуальных учебных задач из области программной инженерии с поддержкой автоматической проверки решений. В задачах, сгенерированных в соответствии с предлагаемым в данной работе методом, учащимся необходимо реализовать две функции: функцию «main» с программной реализацией индивидуального варианта формулы, составленной из матричных операций, а также функцию «test», выполняющую автоматическое тестирование реализованной учащимся функции «main», при этом для оценки качества набора контрольных примеров, реализованного в функции «test», используется мутационный анализ, основанный на заранее заготовленных версиях «main» с мутациями, гарантированно изменяющими результат выполнения функции.

Пример сгенерированной постановки задачи показан на рис. 1.

На рис. 2 показана высокоуровневая схема метода, позволяющего автоматизировать генерацию и проверку задач на тему программной реализации и тестирования матричных преобразований с использованием мутационного анализа для оценки качества тестов, написанных учащимися (см. пример задачи на рис. 1).

Реализовать функцию `main`, принимающую на вход матрицы:

$$\text{main}(D, S, X, Y) = \det((S^2 - 3S^{-1}) \odot (D \cdot (X \odot Y))).$$

В решении необходимо использовать библиотеку `numpy`.

Реализовать в функции `test` автоматическое тестирование функции `main`.

Пример результата вычислений:

```
main(array([[0.57, 0.67, 0.1], [0.33, 0.14, 0.42]]),
      array([[1.0, 0.0], [0.0, 1.0]]),
      array([[0.21, 0.33], [0.62, 0.92], [0.67, 0.72]]),
      array([[0.7, 0.44], [0.26, 0.15], [0.91, 0.01]])) ≈ 7.10e-02
```

Рисунок 1 – Пример сгенерированной задачи

Как показано на рис. 2, процесс генерации задачи начинается с построения формулы, содержащей матричные операции, в виде дерева выражения G , причем индивидуальный номер студента используется в качестве зерна генератора псевдослучайных чисел (ГПСЧ). В случае, если дерево выражения G соответствует ограничениям, осуществляется попытка генерации множества тестов $\text{tests}(G)$ для обеспечения возможности автоматической проверки корректности реализации функции «`main`». В случае, если множество тестов $\text{tests}(G)$ соответствует ограничениям, генерируется множество программ с мутациями $\text{mut}(G)$ для мутационного анализа реализованной студентом функции «`test`». В случае неудачи предпринимается попытка сгенерировать новое дерево G .

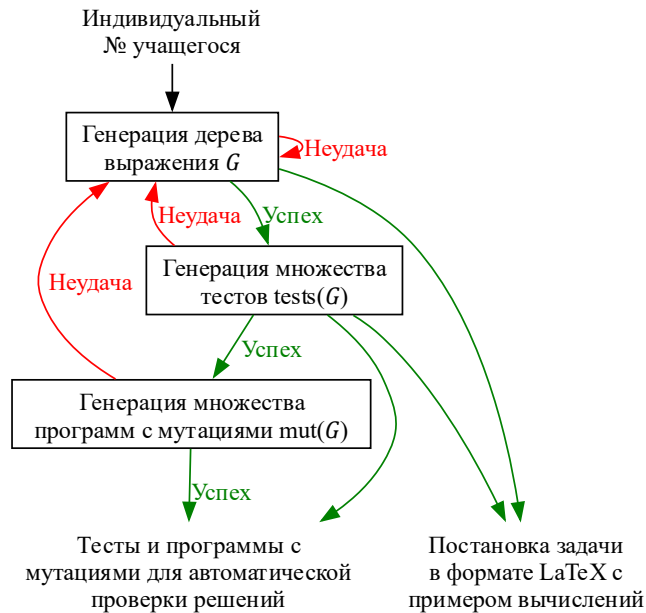


Рисунок 2 – Схема предложенного метода автоматической генерации задач

Генерация деревьев выражений, описывающих матричные преобразования

Для порождения выражений в генераторе задач используется система переписывания термов (СПТ), состоящая из переписывающих правил с поставленными им в соответствие вероятностями применения правил и ограничениями. В рассматриваемой СПТ термы могут быть составлены из матричных символов из множества M и из функциональных символов из множества F .

Каждый матричный символ $v \in M$ представлен тройкой

$$v = (t, n, m), \quad (1)$$

где $t \in \Sigma$ – символ из алфавита Σ , обозначающий имя матричного символа v ; $n, m \in \mathbb{N}$ – число строк и число столбцов в множестве матриц, соответствующих матричному символу v .

Множество функциональных символов F определяется как

$$F = F_h \cup F_f, \quad (2)$$

$$F_h = \{b/2, o/2, u/1, s/1\}, \quad (3)$$

$$F_f = \{+/2, -/2, \odot/2, \cdot/2, ^{-1}/1, ^p/1, k \cdot /1, ^T/1, \det/1, \text{tr}/1, \|\cdot\|_F/1\}, \quad (4)$$

где F_h (3) – множество вспомогательных функциональных символов, арности которых указаны справа от знака «/»; F_f (4) – множество функциональных символов, причём бинарные символы «+», «−», « $\odot$ », « $\cdot$ » соответствуют

бинарным операциям суммы матриц, разности матриц, поэлементного произведения (произведения Адамара) и матричного произведения; унарные символы « $^{-1}$ », « ^{p} », « $k \cdot$ », « ^{T} » соответствуют унарным операциям вычисления обратной матрицы, возведения матрицы в степень $p \in \mathbb{N}$, поэлементного умножения матрицы на заданную константу $k \in \mathbb{N}$, транспонирования матрицы; унарные символы « $\det$ », « tr », « $\|\cdot\|_F$ » соответствуют унарным операциям вычисления определителя матрицы, следа матрицы, нормы Фробениуса.

СПТ состоит из набора переписывающих правил, перечисленных в табл. 1. Числа, поставленные в соответствие правилам с одинаковой левой частью, соответствуют вероятностям выбора разных вариантов этих правил. Выбранное по табл. 1 правило применяется только при условии удовлетворения соответствующего правилу ограничения, в противном случае СПТ выбирает новое правило до успешного переписывания.

Таблица 1 – Вероятностные переписывающие правила и их ограничения для порождения выражений

№	Правило	P	Ограничение
1	$b(\alpha, \beta) \rightarrow \alpha + \beta.$	0.3	$\dim(\alpha) = \dim(\beta).$
2	$b(\alpha, \beta) \rightarrow \alpha - \beta.$	0.3	$\dim(\alpha) = \dim(\beta).$
3	$b(\alpha, \beta) \rightarrow \alpha \odot \beta.$	0.3	$\dim(\alpha) = \dim(\beta).$
4	$b(\alpha, \beta) \rightarrow \alpha \cdot \beta.$	0.1	$\exists n, m, l \in \mathbb{N}(\dim(\alpha) = (n, m) \wedge \dim(\beta) = (m, l)).$
5	$u(\alpha) \rightarrow \alpha^{-1}.$	0.3	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)).$
6	$u(\alpha) \rightarrow \alpha^p.$	0.2	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)) \wedge p \in \{2, \dots, 9\}.$
7	$u(\alpha) \rightarrow k \cdot \alpha.$	0.1	$k \in \{2, \dots, 9\}.$
8	$u(\alpha) \rightarrow \alpha^T.$	0.2	T.
9	$u(\alpha) \rightarrow \alpha.$	0.2	T.
10	$o(\alpha, \beta) \rightarrow b(\alpha, \beta).$	0.3	T.
11	$o(\alpha, \beta) \rightarrow b(\beta, \alpha).$	0.3	T.
12	$o(\alpha, \beta) \rightarrow u(\alpha).$	0.2	T.
13	$o(\alpha, \beta) \rightarrow u(\beta).$	0.2	T.
14	$s(\alpha) \rightarrow \det(\alpha).$	0.4	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)).$
15	$s(\alpha) \rightarrow \text{tr}(\alpha).$	0.4	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)).$
16	$s(\alpha) \rightarrow \ \alpha\ .$	0.2	T.

В правилах, перечисленных в табл. 1, $\alpha, \beta \in F_f \cup M$ – термы, каждый терм – это либо матричный символ из множества M (1), либо функциональный символ из F_f (4); значения $p, k \in \mathbb{N}$ в правилах 6, 7 в табл. 1 выбираются случайно из множества $\{2, \dots, 9\}$ в момент применения правил; T – ограничение, которое всегда удовлетворено.

Отображение $\dim$, используемое в ограничениях правил в табл. 1, ставит в соответствие заданному терму пару из числа строк и столбцов в соответствующей терму матрице по правилам, перечисленным в табл. 2.

Таблица 2 – Отображение $\dim$ для вычисления размера матрицы

№	Правило	Ограничение
1	$\dim(\alpha + \beta) \rightarrow (n, m).$	$\exists n, m \in \mathbb{N}(\dim(\alpha) = \dim(\beta) = (n, m)).$
2	$\dim(\alpha - \beta) \rightarrow (n, m).$	$\exists n, m \in \mathbb{N}(\dim(\alpha) = \dim(\beta) = (n, m)).$
3	$\dim(\alpha \odot \beta) \rightarrow (n, m).$	$\exists n, m \in \mathbb{N}(\dim(\alpha) = \dim(\beta) = (n, m)).$
4	$\dim(\alpha \cdot \beta) \rightarrow (n, l).$	$\exists n, m, l \in \mathbb{N}(\dim(\alpha) = (n, m) \wedge \dim(\beta) = (m, l)).$
5	$\dim(\alpha^{-1}) \rightarrow (n, n).$	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)).$
6	$\dim(\alpha^p) \rightarrow (n, n).$	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)) \wedge p \in \mathbb{N}.$
7	$\dim(k \cdot \alpha) \rightarrow (n, m).$	$\exists n, m \in \mathbb{N}(\dim(\alpha) = (n, m)) \wedge k \in \mathbb{N}.$
8	$\dim(\alpha^T) \rightarrow (m, n).$	$\exists n, m \in \mathbb{N}(\dim(\alpha) = (n, m)).$
9	$\dim(\det(\alpha)) \rightarrow (1, 1).$	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)).$
10	$\dim(\text{tr}(\alpha)) \rightarrow (1, 1).$	$\exists n \in \mathbb{N}(\dim(\alpha) = (n, n)).$
11	$\dim(\ \alpha\) \rightarrow (1, 1).$	$\exists n, m \in \mathbb{N}(\dim(\alpha) = (n, m)).$
12	$\dim(v) \rightarrow (n, m).$	$\exists n, m \in \mathbb{N}(v = (t, n, m) \wedge v \in M), \text{ см. формулу (1).}$

Генерация такого дерева выражения с матричными операциями, в котором размеры матриц оказываются совместимыми друг с другом при применении операций, определяемых функциональными символами из множества F (2), выполняется по алгоритму, псевдокод которого показан на рис. 3 а, с использованием переписывающих правил, перечисленных в табл. 1.

1: функция $\text{expr}(M, \eta)$	1: функция $\text{concat}(E)$
2: $E = \emptyset$	2: пока $ E > 1$ выполнять
3: пока $ E \leq \eta$ выполнять	3: $\alpha, \beta \leftarrow \text{pop}(E), \text{pop}(E)$
4: $E \leftarrow E \cup \{\text{concat}(M)\}$	4: $E \leftarrow E \cup \{o(\alpha, \beta)\}$
5: возвратить $s(\text{concat}(E))$	5: возвратить $\text{pop}(E)$
(а)	(б)

Рисунок 3 – Псевдокод: а) алгоритма генерации дерева выражения, описывающего матричные преобразования, б) вспомогательного алгоритма, объединяющего термы из входного множества с использованием переписывающих правил из СПТ, перечисленных в табл. 1

На вход функции expr (см. рис. 3, а) подаётся множество матричных символов M и параметр сложности дерева выражения $\eta \in \mathbb{N}$.

Множество M инициализируется как

$$M = \{(t_0, n_0, n_1), (t_1, n_0, n_0), (t_2, n_0, n_1), (t_3, n_1, n_0), (t_4, n_1, n_1)\}, \quad (5)$$

где t_i – имя i -го матричного символа, $i \in \{0, \dots, 4\}$, имя t_i выбирается случайно из множества заглавных букв английского алфавита, значения $n_0, n_1 \in \mathbb{N}$ выбираются случайно из множества $\{1, \dots, 5\}$, $n_0 \neq n_1$.

Элементы множества M удовлетворяют следующему ограничению:

$$\bigwedge_{0 \leq i < j \leq 4} (t_i \neq t_j), \quad (6)$$

где $t_i, t_j \in \Sigma$ – имена матричных символов (1) из M (5).

Функция expr (см. рис. 3, а) объединяет матричные символы из M η раз с использованием вспомогательной функции concat , после чего объединённые термы добавляются в множество термов E (строки 3–4 на рис. 3, а). После выполнения η итераций термы из сформированного множества E объединяются функцией concat в единый терм (строка 5 на рис. 3, а), к которому применяется одно из переписывающих правил с термом s в левой части (правила 14–16 в табл. 1) для получения на выходе терма, описывающего вычисление скаляра.

Функция concat (см. рис. 3, б) принимает на вход множество термов для объединения E . На каждой итерации функцией pop из множества E удаляются 2 случайно выбранных терма α и β (строка 3 на рис. 3, б), к ним применяется одно из переписывающих правил с термом o в левой части (правила 10–13 в табл. 1), после чего полученный в результате объединения α и β терм добавляется обратно в множество E . Итеративный процесс объединения термов продолжается, пока в E не останется единственного терма, который будет возвращён из функции (строка 5 на рис. 3, б). Использование множеств в функциях expr и concat позволяет избавиться от дублирующихся подвыражений, составленных из термов, так как множества не содержат дубликатов.

Для демонстрации примера работы функции concat обозначим матричный символ (t, n, m) (1), соответствующий матрице с именем t , числом строк n и числом столбцов m , как $t_{n \times m}$.

Тогда входное множество M_0 , инициализированное с ГПСЧ по правилу (5) и переданное на вход функции concat , может изменяться следующим образом:

$$M_0 = \{A_{2 \times 3}, B_{2 \times 2}, C_{2 \times 3}, D_{3 \times 2}, K_{3 \times 3}\}, \quad (7)$$

$$M_1 = \{B_{2 \times 2}, D_{3 \times 2}, K_{3 \times 3}, A_{2 \times 3} + C_{2 \times 3}\}, \quad (8)$$

$$M_2 = \{D_{3 \times 2}, K_{3 \times 3}, (A_{2 \times 3} + C_{2 \times 3})^T\}, \quad (9)$$

$$M_3 = \{K_{3 \times 3}, (A_{2 \times 3} + C_{2 \times 3})^T \odot D_{3 \times 2}\}, \quad (10)$$

$$M_4 = \{5(A_{2 \times 3} + C_{2 \times 3})^T \odot D_{3 \times 2}\}, \quad (11)$$

где i в именах вида M_i соответствует номеру итерации цикла на строках 2–4 в функции concat (см. рис. 3, б).

Множество M_1 (8) представляет собой множество M_0 (7), матричные символы $A_{2 \times 3}$ и $C_{2 \times 3}$ в котором были объединены в один терм бинарным символом «+» в результате применения правил 10 и 1 из табл. 1. Множество M_2 (9) получено в результате применения правил 12 и 8 из табл. 1 к термам $A_{2 \times 3} + C_{2 \times 3}$ и $B_{2 \times 2}$. Поскольку правило 12 из табл. 1 отбрасывает правый операнд, матричный символ $B_{2 \times 2}$ в результирующее выражение $(A_{2 \times 3} + C_{2 \times 3})^T$ в (9) не включается. При построении множества M_3 (10) использовались правила 10 и 3 из табл. 1, а при построении множества M_4 (11) – правила 12 и 7. Поскольку M_4 содержит единственный терм, критерий останова на строке 2 функции concat выполняется и алгоритм завершает работу, извлекая этот терм из M_4 и возвращая его.

Дерево выражения G , сформированное функцией expr (см. рис. 3, а), принимается только в том случае, если оно соответствует следующим ограничениям:

$$|\text{ops}(G) \cap \{-1, ^T\}| > 0, \quad (12)$$

$$|\text{ops}(G) \cap \{+, -, k \cdot, ^p\}| \geq \mu, \quad (13)$$

$$\exists (t, n, m) \in \text{vars}(G) (n = m), \quad (14)$$

$$|\text{vars}(G)| > 2, \quad (15)$$

где ops – отображение, ставящее в соответствие дереву G множество функциональных символов, использующихся в G , в (12) и (13) $\text{ops}(G) \subseteq F_f$ (4); vars – отображение, ставящее в соответствие G множество матричных символов, использующихся в G , в (14) и (15) $\text{vars}(G) \subseteq M$ (1); t – имя матричного символа; $n, m \in \mathbb{N}$ – число строк и столбцов матричного символа.

Ограничение (12) позволяет убедиться в наличии в задаче, которой соответствует дерево G , функциональных символов, описывающих операции линейной алгебры, таким как вычисление обратной матрицы и транспонирование. Ограничения (13) и (14) необходимы для последующего мутационного анализа программ, решающих задачу, соответствующую G : (13) позволяет убедиться в наличии в G как минимум μ функциональных символов, соответствующих операциям, применение которых не меняет размер матрицы, а (14) позволяет убедиться в наличии хотя бы одного матричного символа, описывающего квадратную матрицу. Наконец, ограничение (15) позволяет удостовериться, что выражение G не является тривиальным.

При трансляции сгенерированного дерева выражения G в текст постановки задачи в формате LaTeX, подобный указанному на рис. 1, а также при генерации набора тестов для автоматической проверки решений учащихся (см. рис. 2) формируются кортежи из входных матриц X только для матричных символов, присутствующих в G и принадлежащих $\text{vars}(G) \subseteq M$ (5), таким образом гарантируется отсутствие избыточных имён в параметрах функции, которую необходимо реализовать.

Генерация тестов для автоматической проверки текстов программ

На следующем этапе формирования постановки задачи для дерева выражения G генерируются тесты в виде множества кортежей, каждый из которых включает кортеж входных матриц X и результат вычисления значения дерева G для кортежа входных матриц X для обеспечения возможности автоматической проверки программ в дальнейшем. Схема, иллюстрирующая принцип работы генератора тестов, показана на рис. 4.

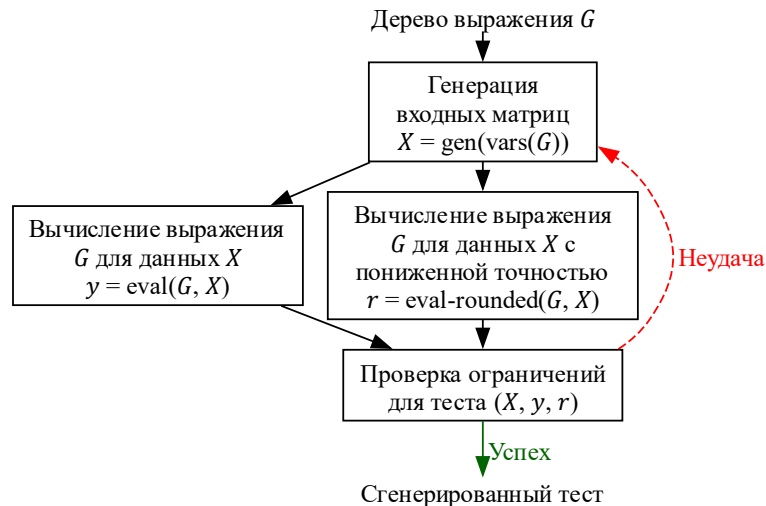


Рисунок 4 – Схема работы генератора тестов, обеспечивающего возможность автоматической проверки программных решений задач на платформах с различающимися реализациями матричных операций

Как показано на рис. 4, генерация теста начинается с формирования кортежа входных матриц X функцией gen , в кортеж X помещаются матрицы, размеры которых указаны в матричных символах $\text{vars}(G) \subseteq M$ (5), присутствующих в G , матрицы в X упорядочены по именам соответствующих им символов. Матрицы для X формируются путём заполнения их ячеек случайными значениями из промежутка $[0,1] \subset \mathbb{R}$.

После этого дважды выполняется интерпретация дерева выражения G для одного и того же кортежа входных матриц X – в первом случае для вычисления результата $y \in \mathbb{R}$ применяется интерпретатор eval , использующий операции из библиотеки с реализацией матричных преобразований, например, NumPy [24] или PyTorch [25], а во втором случае для вычисления результата $r \in \mathbb{R}$ применяется интерпретатор eval-rounded , в нём результаты выполнения всех библиотечных операций округляются до 10^{-6} .

Поскольку в дереве G могут присутствовать функциональные символы, определяющие такие операции, как обращение матриц, матричное умножение и вычисление определителя (см. табл. 1), ячейки входных матриц содержат вещественные числа, а предпочтительный способ программного решения задачи предполагает использование библиотеки с реализацией матричных преобразований, на платформах с различающимися

операционными системами и аппаратными архитектурами вычисление одного и того же дерева G с одними и теми же входными матрицами X может приводить к существенным различиям в результатах вычислений.

Для исключения влияния различий в реализациях библиотек с реализацией матричных преобразований для разных платформ принимаются только такие деревья G и множества тестов $\text{tests}(G)$, для которых удовлетворено следующее ограничение:

$$\forall (X, y, r) \in \text{tests}(G) (|y - r| < 10^{-6} \wedge 10^{-5} < |y| < 10^5), \quad (16)$$

где X – кортеж из входных матриц, соответствующих матричным символам из $\text{vars}(G) \subseteq M(5)$, присутствующим в дереве G , $|X| = |\text{vars}(G)|$; $y \in \mathbb{R}$ – результат вычисления дерева выражения G для кортежа входных матриц X с использованием библиотеки с реализацией матричных преобразований; $r \in \mathbb{R}$ – результат вычисления значения дерева выражения G для кортежа входных матриц X с пониженной точностью матричных операций.

Таким образом, как указано в (16) и на рис. 4, предложенный метод порождает только такие деревья выражений G и наборы тестов $\text{tests}(G)$, в которых различия между значениями y и r достаточно малы, что обеспечивает воспроизводимость результатов преобразований матриц на разных платформах.

Мутационный анализ программ

В генерируемых задачах студенту предлагается не только реализовать функцию «main», вычисляющую значение выражения G на основе формулы, соответствующей G , но и функцию «test», тестирующую программную реализацию функции «main».

С целью того, чтобы не засчитывать тривиальные программные реализации функции «test», на третьем этапе на основе дерева выражения G и множества тестов $\text{tests}(G)$ формируется множество $\text{mut}(G)$ из μ модифицированных версий дерева выражения G путём внесения случайных изменений (мутаций) в G на основе переписывающих правил 1–4 из табл. 3, правила 1–4 из табл. 3 выбираются равновероятно.

Таблица 3 – Мутации деревьев выражений

№	Правило	Ограничение
1	$\alpha + \beta \rightarrow \alpha - \beta.$	T.
2	$\alpha - \beta \rightarrow \alpha + \beta.$	T.
3	$k \cdot \alpha \rightarrow (k + 1) \cdot \alpha.$	$k, k + 1 \in \mathbb{N}.$
4	$\alpha^p \rightarrow \alpha^{(p+1)}.$	$p, p + 1 \in \mathbb{N}.$
5	$\alpha \rightarrow \alpha^T.$	$\exists n \in \mathbb{N} (\dim(\alpha) = (n, n) \wedge \alpha \in M).$

Модифицированные версии G' дерева G используются при автоматической проверке программных решений задачи путём мутационного анализа функции «test», реализованной студентом. Приемлемая реализация функции «test» должна выполняться без ошибок при проверке корректной реализации функции «main» на основе дерева G , но завершаться ошибкой тестирования `AssertionError` при проверке некорректной реализации функции «main», полученной путём трансляции $G' \in \text{mut}(G)$ в текст программы.

В связи с указанными требованиями в предложенном методе генерации задач принимаются только такие деревья выражений G , множества тестов $\text{tests}(G)$ и множества деревьев с мутациями $\text{mut}(G)$, для которых удовлетворяется следующее ограничение:

$$\forall G' \in \text{mut}(G) \forall (X, y, r) \in \text{tests}(G) (|y - \text{eval}(G', X)| > 10^{-6}), \quad (17)$$

где $y \in \mathbb{R}$ – результат вычисления дерева выражения G для кортежа входных матриц X ; eval – функция, вычисляющая результат для модифицированного по табл. 3 дерева G' и кортежа входных матриц X .

Как указано в (17), принимаются только те деревья выражений G' с мутациями, результат вычисления которых значительно отличается от результата вычисления исходного дерева G для входных данных X .

Примеры модифицированных версий дерева выражения G из множества $\text{mut}(G)$, транслированных в тексты программ на языке Python, показаны на рис. 5 в формате git diff, где знаком «–» обозначены удалённые из оригинальной программы строки кода, а знаком «+» – строки, добавленные в программу.

С целью того, чтобы для корректной реализации функции «test» одного примера входных и выходных данных, приведённого в постановке задачи, было недостаточно, в множество тестов $\text{tests}(G)$ добавляется дополнительный тест, в котором квадратные входные матрицы заменяются на единичные матрицы. Именно этот тест приводится в качестве примера входных и выходных данных в постановке задачи, наличие в дереве G матричных символов, соответствующих квадратным матрицам, гарантируется ограничением (14).

Поскольку

$$I = I^T, \quad (18)$$

где I – единичная матрица, в множество модифицированных версий $\text{mut}(G)$ дерева выражения G также добавляется случайным образом изменённое по правилу 5 из табл. 3 дерево выражения G , что позволяет отклонять на этапе мутационного анализа те программные решения задач, в которых тестирование преобразований матриц осуществляется на единственном примере входных и выходных данных с единичными матрицами, затрудняя тем самым решение задачи без использования интерпретатора или компилятора языка программирования для отладки программы на компьютере учащегося или с использованием LLM.

<pre>def main(f, m, s): t1 = f.T t2 = np.linalg.matrix_power(s, 3) - t3 = m - t1 + t3 = m + t1 t4 = t3 @ t2 t5 = 2 * t4 return np.linalg.norm(t5)</pre> (а)	<pre>def main(f, m, s): t1 = f.T - t2 = np.linalg.matrix_power(s, 3) + t2 = np.linalg.matrix_power(s.T, 3) t3 = m - t1 t4 = t3 @ t2 t5 = 2 * t4 return np.linalg.norm(t5)</pre> (б)
--	--

Рисунок 5 – Примеры случайных изменений (мутаций), внесённых в текст программы на языке Python: а) по правилам 1–4 в табл. 3; б) по правилу 5 в табл. 3

При успешной генерации множества $\text{mut}(G)$, соответствующего ограничению (17), модифицированные по правилам из табл. 3 версии G' дерева G сохраняются на диск для обеспечения возможности автоматической оценки качества функций «test», реализованных учащимися.

Автоматическая проверка текстов программ

Автоматическая проверка текста программы p учащегося, решающего задачу, заданную деревом выражения G , выполняется по алгоритму, псевдокод которого показан на рис. 6, а.

<pre>1: функция check(p, G, η) 2: для $(X, y, r) \in \text{tests}(G)$ выполнять 3: $o, e \leftarrow \text{sandbox}(X, p)$ 4: если $e \neq \perp$ то 5: вернуть e 6: если $o - y > 10^{-2}$ то 7: вернуть o 8: вернуть $\text{analyze}(p, G) \leq \eta$</pre> (а)	<pre>1: функция analyze(p, G) 2: $s \leftarrow 0$ 3: для $G' \in \text{mut}(G)$ выполнять 4: $k \leftarrow 1$ 5: для $(X, y, r) \in \text{tests}(G)$ выполнять 6: $o, e \leftarrow \text{sandbox}(X, p \parallel \text{compile}(G'))$ 7: если $e \neq \perp$ то 8: $k \leftarrow 0$ 9: $s \leftarrow s + k$ 10: вернуть $s \text{mut}(G) ^{-1}$</pre> (б)
---	--

Рисунок 6 – Псевдокод: а) алгоритма проверки текста программы p , б) алгоритма мутационного анализа текста программы p с использованием множества тестов и множества текстов программ с мутациями

Как показано на рис. 6, сначала выполняется проверка корректности работы программы p на сформированном ранее наборе тестов $\text{tests}(G)$. Безопасное выполнение программы p осуществляется в «песочнице» при помощи функции `sandbox`, принимающей на вход кортеж из матриц X и текст программы p , на практике «песочница» может быть реализована как Docker-контейнер с изоляцией `gVisor` [26]. Функция `sandbox` на рис. 6 вычисляет результат выполнения программы p при входных данных X , а также запускает функцию с именем «test», которая должна присутствовать в программе p . В случае, если выполнение программы p завершено с ошибкой e (строки 4–5 на рис. 6) или если ожидаемое значение y значительно отличается от фактического результата o (строки 6–7), функция `check` завершает работу и возвращает сведения об ошибке. Иначе запускается мутационный анализ p по алгоритму, псевдокод которого показан на рис. 6, б.

Функция `analyze` (см. рис. 6, б) подсчитывает относительное число программ с мутациями (см. рис. 5), которые не были обнаружены реализованной студентом функцией «test». Для этого модифицированное дерево выражения $G' \in \text{mut}(G)$ транслируется в текст программы функцией «`compile`» и конкатенируется с текстом программы p (см. строку 6), после чего полученная программа, содержащая функцию «`main`» с мутацией и функцию «`test`», запускается в «песочнице». Если в результате выполнения программы с мутацией возникло

исключение (см. строку 7), считается, что функция «test» обнаружила мутацию и отклонила функцию «main» с внедрённой ошибкой.

Параметр η в функции check (см. рис. 6, а) определяет долю программ с мутациями, которые должны быть обнаружены функцией «test» в результате мутационного анализа.

Вычислительная сложность алгоритма check равна $O(|\text{tests}(G)|(1 + |\text{mut}(G)|))$.

Оценка производительности генератора и времени автоматической проверки

Представленный метод генерации задач (см. рис. 2) и алгоритм автоматической проверки их решений на основе мутационного анализа (см. рис. 6, а) был реализован на языке программирования Python версии 3.12 и использован в системе «Цифровой ассистент преподавателя» (ЦАП). Система ЦАП применяется в рамках курса «Программирование на языке Питон» Института информационных технологий (ИИТ) РТУ МИРЭА для автоматизации рутинной деятельности преподавателей, связанной с созданием сборников индивидуальных задач, проверкой программ и определением способов решения задач [26].

Значения параметров разработанного генератора, таких как число тестов $|\text{tests}(G)|$ и число программ с мутациями $|\text{mut}(G)|$, выбирались на основе оценок производительности программной реализации метода генерации задач (см. рис. 2) и оценок времени работы алгоритма автоматической проверки (см. рис. 6), показанных на рис. 7. Замеры производительности и времени выполнялись на компьютере со следующими характеристиками: Intel Core i3-1115G4 @ 3.00 GHz, 8 GB RAM, SSD 256 GB PCIe-4, Windows 11, Python 3.12.0. Для получения достоверных оценок замеры производительности и времени повторялись 30 раз, на рис. 7 приведены округлённые вниз средние значения числа генерируемых задач в минуту и времени, затрачиваемого на проверку программ.

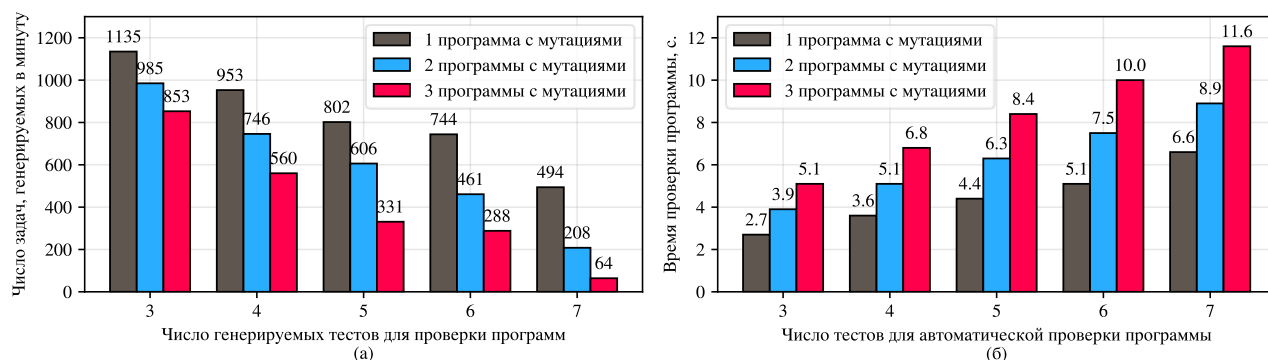
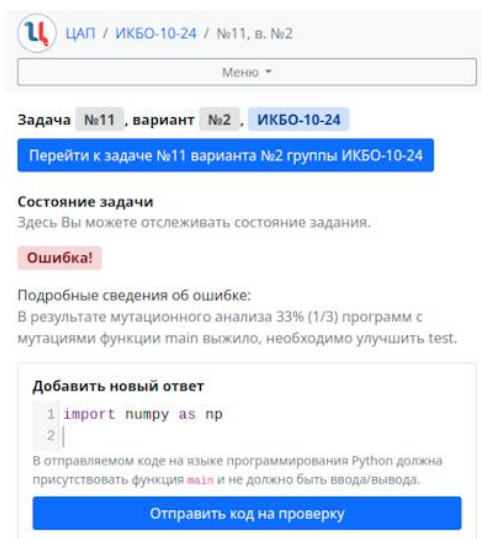


Рисунок 7 – Оценки: а) производительности программной реализации предложенного метода генерации задач; б) времени, затрачиваемого на проверку и мутационный анализ программ, для разного числа тестов и разного числа программ с мутациями

Как показано на рис. 7, б, с числом тестов $|\text{tests}(G)|$, равным 5, и числом программ с мутациями $|\text{mut}(G)|$, равным 3, среднее время автоматической проверки программы по алгоритму, псевдокод которого показан на рис. 6, а, не превышает 10 секунд, а время генерации индивидуальных вариантов задач для более 1400 студентов курса не превышало нескольких минут (см. рис. 7, а). Эти значения параметров использовались в системе ЦАП [26] в весеннем семестре 2025–2026 учебного года. Пользовательский интерфейс системы ЦАП с выводом сообщения о низком качестве тестирующей функции, сформированного на основе результатов мутационного анализа, показан на рис. 8, а. Пример текста программы, решающей один из сгенерированных вариантов задач, но содержащей некачественную реализацию функции «test», показан на рис. 8, б.

Как показано на рис. 8, а, после получения сообщения об отклонении программы по результатам её мутационного анализа учащийся имеет возможность исправить текст программы и повторно отправить его на проверку. Тестирующая функция в программе на языке Python, показанной на рис. 8, б, оказывается неспособна обнаружить, например, удаление операции транспонирования из правой части присваивания на строке 4 или удаление операции возведения матрицы в степень на строке 5, поскольку в функции «test» на вход функции «main» подаются только единичные матрицы. Вследствие этого функция «test» считается некачественной и отклоняется при её проверке по алгоритму, псевдокод которого показан на рис. 6, б.



(а)

```

1  import numpy as np
2
3  def main(a, c, k):
4      t1 = k.T
5      t2 = np.linalg.matrix_power(c, 6)
6      t3 = t1 + t2
7      t4 = np.linalg.inv(k)
8      t5 = c - t4
9      t6 = t5 + a
10     t7 = np.linalg.inv(t6)
11     t8 = t3 - t7
12     t9 = np.linalg.det(t8)
13     return t9
14
15 def test():
16     eye = np.eye(2)
17     res = main(eye, eye, eye)
18     assert np.isclose(res, 1.0)
    
```

(б)

Рисунок 8 – а) Пользовательский интерфейс системы ЦАП [26] с выводом результатов мутационного анализа; б) пример текста программы с некачественной реализацией функции «test»

Оценка эффекта применения мутационного анализа при проверке программ

Число отклонённых и принятых ЦАП программ из 12 269 программ, решающих индивидуальные задачи на тему программной реализации и тестирования матричных преобразований, с использованием мутационного анализа функции «test» и без него представлено в табл. 4.

Основные причины отклонения программ системой ЦАП показаны на рис. 9.

Таблица 4 – Число отклонённых и принятых программ с мутационным анализом и без него

Метрика	Отклонено программ	Принято программ
Результаты проверки без мутационного анализа	7414 (60,4%)	4855 (39,6%)
Результаты проверки с мутационным анализом	11530 (94,0%)	739 (6,0%)
Эффект применения мутационного анализа	+4116 (33,5%)	–4116 (33,5%)

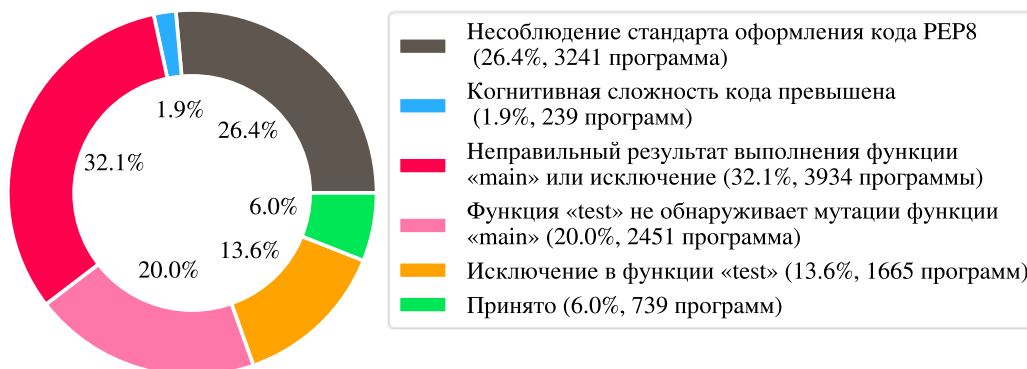


Рисунок 9 – Причины отклонения программ, решающих индивидуальные задачи на тему программной реализации и тестирования матричных преобразований, выявленные по результатам их проверки ЦАП

Как показано на рис. 9, значительная часть программ, отправленных на проверку системе ЦАП, была отклонена из-за несоблюдения стандарта оформления кода PEP8 (3241 шт.) и из-за ошибок в реализации функции «main» (3934 шт.), приводящих к исключениям или к неправильным результатам выполнения этой функции. Малая часть программ (239 шт.) была отклонена из-за превышения порогового значения метрики когнитивной сложности кода, функции в программах из этой категории содержали слишком много вложенных циклов, условных операторов и других управляющих конструкций [27].

Без использования мутационного анализа единственным требованием к функции «test» в программе p , отправленной студентом на проверку системе ЦАП, являлось отсутствие исключений при запуске функции, таким образом она могла быть, например, пустой. Однако, наличие мутационного анализа повысило требования к качеству функции «test». Это привело к тому, что при её реализации учащиеся нередко допускали ошибки,

приводящие, например, к возникновению исключения `AssertionError` при запуске программы и тестов в песочнице (см. строку 3 на рис. 6, а) ещё до внедрения случайных изменений в функцию «main» и, следовательно, до мутационного анализа функции «test» (см. рис. 6, б). Так, 1665 программ (см. рис. 9) было отклонено по причине возникновения исключения в функции «test» при запуске программы *p* в песочнице (см. строки 3–5 на рис. 6, а). При этом 2451 программа (см. рис. 9) была отклонена из-за того, что функция «test» успешно завершала работу даже при наличии случайных изменений, искажающих результат выполнения функции «main», внесённых в эту функцию по алгоритму, показанному на рис. 6, б.

Как показано в табл. 4, мутационный анализ функции «test» позволил повысить на 33,5% от общего числа проверенных программ число отклонённых программ с контрольными примерами, не позволяющими обнаружить намеренно внедрённые в функцию «main» изменения, приводящие к её некорректной работе. Отсутствие ограничений на число попыток сдачи работ в системе ЦАП позволило студентам провести работу над повышением качества тестирующей функции, заключающуюся в улучшении набора контрольных примеров в функции «test». В результате 739 программ было принято системой ЦАП.

Для численной оценки различий между 739 принятыми программами и 2451 отклонёнными программами с функциями «test», завершающими выполнение без исключений, но не обнаруживающими мутации функции «main» (см. рис. 9), было подсчитано число вызовов оператора `assert` при выполнении функции «test» при помощи стандартного средства динамического анализа кода `sys.settrace`, доступного в интерпретаторе CPython. Вызов `sys.settrace` позволяет зарегистрировать функцию трассировки [28], которая вызывается перед исполнением каждой строки анализируемого кода на языке Python. Вызовом оператора `assert` считался как вызов стандартного оператора `assert`, так и вызов его специализированных вариантов, таких как `np.testing.assert_allclose` из библиотеки NumPy [24]. Оператор `assert` широко используется в модульном тестировании программ на языке Python и позволяет проверять в тестах ограничения для выходных данных при заданных входных данных, завершая работу с исключением `AssertionError` в случае, если ограничения не удовлетворены.

Характеристики распределений числа вызовов оператора `assert` для 739 принятых программ и для 2451 отклонённой программы показано в табл. 5. Распределения числа вызовов оператора `assert` после удаления выбросов показаны на рис. 10 в виде «ящиков с усами».

Таблица 5 – Характеристики распределений числа вызовов оператора `assert`

Распределение числа вызовов <code>assert</code>	Медиана	Среднее	СКО	Минимум	Максимум
Для 2451 отклонённой программы	1	2,5	5,4	0	161
Для 739 принятых программ	3	5,6	12,7	1	201

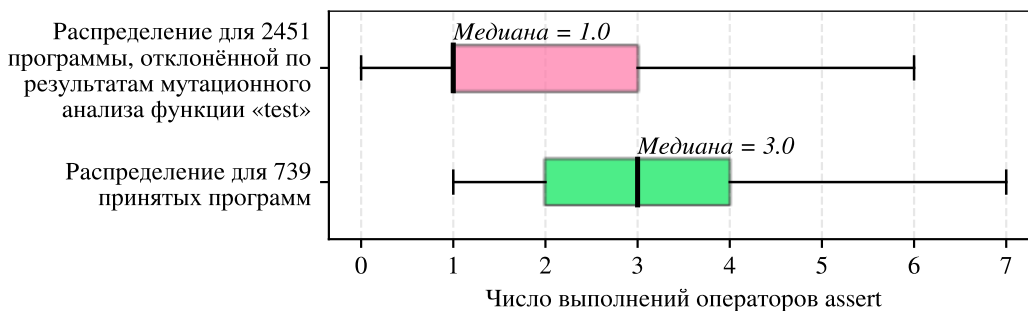


Рисунок 10 – Распределение числа вызовов оператора `assert` при выполнении функции «test» в принятых программах и в программах, отклонённых по результатам мутационного анализа

На рис. 10 показано распределение значений ω , соответствующих количеству выполнений оператора `assert`, для которых удовлетворяется следующее ограничение:

$$\omega \leq Q_3 + 1.5 \cdot \Delta Q, \quad (19)$$

$$\Delta Q = Q_3 - Q_1, \quad (20)$$

где Q_1 и Q_3 – первый и третий квартили, а ΔQ – межквартильный размах, ограничение (19) позволяет исключить из визуализации выбросы, жирной линией на рис. 10 выделено медианное значение для распределений.

Как показано на рис. 10 и в табл. 5, в отклонённых программах с корректной реализацией матричных преобразований в функции «main», но с некачественной с точки зрения мутационного анализа функцией «test» в большинстве случаев содержался единственный вызов оператора `assert` с контрольным примером, взятым из раздела «Пример результата вычислений», приведённого в постановке задачи (см. рис. 1). При улучшении

функции «test» большинство учащихся увеличивало число контрольных примеров до 3, хотя для получения зачёта по задаче в функцию «test» было достаточно поместить единственный контрольный пример без подаваемых на вход единичных или симметричных матриц (см. правило мутации 5 в табл. 3).

Рост значения среднеквадратичного отклонения (СКО) числа вызовов оператора `assert` в принятых программах по сравнению с отклонёнными связан с тем, что в систему ЦАП были присланы, в том числе, решения, в которых число вызовов `assert` превышало 200. В таких решениях функция «main» тестировалась в цикле на случайно сгенерированных входных матрицах. В среднем после работы учащихся над повышением качества тестирующей функции число вызовов оператора `assert` увеличилось в 2,24 раза, с 2,5 до 5,6 (см. табл. 5), а медианное число вызовов увеличилось в 3 раза.

Оценка разрешимости задач LLM с подходом zero-shot

Развитие и доступность современных LLM приводит к их массовому использованию при выполнении и сдаче практических работ учебных дисциплин. В новых условиях хорошей задачей считается та задача, для получения готового решения которой недостаточно отправить её постановку в LLM в режиме zero-shot, то есть без дополнительного контекста, который может содержать описание процедуры решения задачи, пример решения другого варианта или иные инструкции для LLM, направленные на получение корректного решения.

Для оценки разрешимости сгенерированных по предложенному методу задач на вход LLM подавалась только постановка задачи (см. пример на рис. 1) без дополнительного контекста, содержащего пояснения к решению или пример решения. Сгенерированный LLM текст программы затем подавался на вход средству автоматического оформления кода по стандарту PEP8 black, оформленный по стандарту PEP8 программный код загружался в систему ЦАП на проверку.

Постановки 10 задач, сгенерированных в соответствии с предложенным методом, подавались на вход таким LLM, как Yandex GPT 5.1 Pro, DeepSeek V4-Pro в режиме DeepThink, Qwen 3.7-Plus в режиме Reasoning. Проверка текстов программ осуществлялась двумя способами – без мутационного анализа тестирующей функции и с мутационным анализом. Полученные оценки разрешимости задач LLM с подходом zero-shot приведены в табл. 6.

Таблица 6 – Оценки разрешимости задач LLM при использовании подхода zero-shot

LLM	Yandex GPT 5.1 Pro	DeepSeek V4-Pro	Qwen 3.7-Plus
Принято без мутационного анализа	80%	90%	90%
Принято с мутационным анализом	10%	10%	20%
Выигрыш от мутационного анализа	–70%	–80%	–70%

Как показано в табл. 6, без использования мутационного анализа при проверке решений задачи оказываются решёнными LLM с подходом zero-shot в 80–90% случаев. При этом внедрение мутационного анализа позволяет снизить число решённых LLM задач на 70–80% вплоть до 10–20% в том случае, если запрос к LLM не дополняется контекстом, содержащим примеры решения или инструкции учащегося, способствующие генерации набора контрольных примеров, обнаруживающих все мутации.

Заключение

Основным научным результатом работы является новый метод генерации задач на тему программной реализации и тестирования матричных преобразований, отличающийся возможностью автоматической проверки программ и использованием мутационного анализа для оценки качества тестирующей функции, реализованной учащимся. Использование метода позволило освободить преподавателей курса «Программирование на языке Питон» ИИТ РТУ МИРЭА от необходимости проверки 12269 программ вручную (см. табл. 4) и высвободить время на разбор творческих задач по программированию на очных практических занятиях.

Генерация уникальных задач для более 1400 студентов курса с 5 тестами и 3 программами с мутациями занимает всего несколько минут (см. рис. 7), что делает возможным применение метода в массовых открытых онлайн-курсах, в рамках дисциплин высших учебных заведений с большим числом студентов и при проведении соревнований и олимпиад по программированию. Время автоматической проверки программы при том же числе тестов и программ с мутациями в песочнице, реализованной в виде Docker-контейнера с изоляцией gVisor [26], составляет 8,4 секунды, так как мутационный анализ программы по алгоритму, псевдокод которого показан на рис. 6, б, предполагает перезапуск тестов для каждой программы с мутациями. В связи с этим в системе ЦАП использовался асинхронный подход к проверке программ на основе очереди сообщений [26].

Применение мутационного анализа как отдельного этапа при автоматической проверке программных решений задач позволило повысить устойчивость задач к их решению при помощи LLM с подходом zero-shot на 70–80% (см. табл. 6) и привело к отклонению 4116 программ (33,5% от общего числа проверенных программ) с

некачественной реализацией тестирующей функции (см. табл. 4). Исходя из полученных результатов использование мутационного анализа представляется целесообразным даже в том случае, если учащиеся имеют возможность решать задачи с LLM, с целью мотивации учащихся к изучению основ промпт-инженерии и модульного тестирования для формирования корректных запросов к LLM. Среднее число выполнений оператора `assert` по результатам динамического анализа тестирующих функций увеличилось в 2,24 раза в принятых программах по сравнению с программами, отклонёнными по причине наличия не обнаруженных программ с мутациями, что свидетельствует о повышении качества реализованных учащимися тестирующих функций.

Список литературы

1. Абрамова Е. С. Сравнение машины экстремального обучения и метода обратного распространения ошибки при решении задачи распознавания физической активности человека // *Известия Тульского государственного университета. Технические науки.* – 2023. – Вып. 2. – С. 117–124.
2. Проневич О. Б., Клокова А. П. Анализ UMAP–метода снижения размерности исходных данных в машинном обучении для прогнозирования отказов в локомотивном комплексе // *Надежность.* – 2022. – Т. 22, № 4. – С. 53–62.
3. Шишин К. А., Муравьев И. В., Куликов Е. К. Об автоматической генерации модульных тестов для Java-приложений, использующих фреймворк Spring // *Труды Института системного программирования РАН.* – 2024. – Т. 36, вып. 2. – С. 59–72.
4. Neves L., Campos O., Santos R., Magalhaes C., Santos I., Souza Santos R. de. Elevating Software Quality in Agile Environments: The Role of Testing Professionals in Unit Testing / *Proceedings of the 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* – IEEE, 2024. – pp. 293–296.
5. Межуев Т. П., Парыгина Д. А., Куц Д. О. Направленный фаззинг Rust и Go с помощью LibAFL-DiFuzz // *Труды Института системного программирования РАН.* – 2026. – Т. 38, вып. 2. – С. 69–82.
6. Райкин Г. Р., Пелевин М. С., Ицыксон В. М. Алгоритм выбора семантических мутаций в фаззинге по принципу серого ящика // *Труды Института системного программирования РАН.* – 2025. – Т. 37, вып. 2. – С. 97–114.
7. Иванов А. А., Довгальок П. М., Фурсова Н. И. Обзор существующих методов анализа полносистемного покрытия кода // *Труды Института системного программирования РАН.* – Институт системного программирования им. ВП Иванникова Российской академии наук, 2025. – Т. 37, вып. 6 (4). – С. 187–200.
8. Altmayer Pizzorno J., Berger E. D. Slipcover: Near Zero-Overhead Code Coverage for Python / *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* – 2023. – pp. 1195–1206.
9. Mishra D. B., Acharya B., Rath D., Gerogiannis V. C., Kanavos A. A Novel Real Coded Genetic Algorithm for Software Mutation Testing // *Symmetry.* – MDPI, 2022. – Vol. 14, No. 8. – P. 1525.
10. Dakhel A. M., Nikanjam A., Majdinasab V., Khomh F., Desmarais M. C. Effective Test Generation Using Pre-Trained Large Language Models and Mutation Testing // *Information and Software Technology.* – Elsevier, 2024. – Vol. 171. – P. 107468.
11. Смирнов А. В. Мутационное тестирование программного обеспечения // *Известия СПбГЭТУ ЛЭТИ.* – 2015. – № 3. – С. 28–31.
12. Kaufman S. J., Featherman R., Alvin J., Kurtz B., Ammann P., Just R. Prioritizing mutants to guide mutation testing / *Proceedings of the 44th International Conference on Software Engineering.* – 2022. – P. 1743–1754.
13. Сычев О. А., Прокудин А. А., Денисов М. Е. Автоматическая генерация заданий по результатам анализа использования их банка в интеллектуальной обучающей системе // *Программные продукты и системы.* – 2024. – Т. 37, № 2. – С. 201–212.
14. Кишкин С. А. Совершенствование учебно-методического комплекса по дисциплине "Практические задачи оптимизации" // *E-Scio.* – 2022. – № 7 (70). – С. 278–284.
15. Valls-Vargas J., Zhu J., Ontañón S. Graph grammar-based controllable generation of puzzles for a learning game about parallel programming / *Proceedings of the 12th International Conference on the Foundations of Digital Games.* – 2017. – pp. 1–10.
16. Edwards S. H. Improving student performance by evaluating how well students test their own programs // *Journal on Educational Resources in Computing (JERIC).* – ACM New York, NY, USA, 2003. – Vol. 3, № 3. – pp. 1–24.
17. Edwards S. H., Shams Z. Comparing test quality measures for assessing student-written tests / *Companion Proceedings of the 36th International Conference on Software Engineering.* – 2014. – pp. 354–363.

18. Perretta J., DeOrio A., Guha A., Bell J. On the Use of Mutation Analysis for Evaluating Student Test Suite Quality / *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. – 2022. – pp. 263–275.
19. Laurent T., Papadakis M., Kintis M., Henard C., Le Traon Y., Ventresque A. Assessing and Improving the Mutation Testing Practice of PIT / *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. – IEEE, 2017. – pp. 430–435.
20. Ермаков А. Д., Евтушенко Н. В. Метод синтеза тестов с гарантированной полнотой по модели расширенного автомата // *Моделирование и анализ информационных систем*. – 2016. – Т. 23, № 6. – С. 729–740.
21. Sovietov P. N. Automatic generation of programming exercises / *Proceedings of the 2021 1st international conference on technology enhanced learning in higher education (TELE)*. – IEEE, 2021. – pp. 111–114.
22. Горчаков А. В., Фандеев И. И. Методика автоматической генерации задач на тему анализа низкоуровневых представлений кода / *Актуальные проблемы прикладной математики, информатики и механики: сборник материалов Международной научной конференции Актуальные проблемы прикладной математики, информатики и механики (г. Воронеж, 1–3 декабря 2025 г.)*. – Воронеж, 2026. – С. 4–10.
23. Советов П. Н., Горчаков А. В. Модели и алгоритмы автоматической генерации учебных задач по работе с системой git и разрешению зависимостей между пакетами // *Электронный научный журнал «ИТ-Стандарт»*. – 2025. – Т. 3. – С. 47–57.
24. Harris C. R., Millman K. J., Walt S. J. van der, Gommers R., Virtanen P., Cournapeau D., Wieser E., Taylor J., Berg S., Smith N. J., Kern R., Picus M., Hoyer S., Kerkwijk M. H. van, Brett M., Haldane A., Río J. F. del, Wiebe M., Peterson P., Gérard-Marchant P., Sheppard K., Reddy T., Weckesser W., Abbasi H., Gohlke C., Oliphant T. E. Array programming with NumPy // *Nature*. – Springer Science; Business Media LLC, 2020. – Vol. 585, No. 7825. – pp. 357–362.
25. Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., Killeen T., Lin Z., Gimelshein N., Antiga L., Desmaison A., Kopf A., Yang E., DeVito Z., Raison M., Tejani A., Chilamkurthy S., Steiner B., Fang L., Bai J., Chintala S. PyTorch: An Imperative Style, High-Performance Deep Learning Library // *Advances in Neural Information Processing Systems*. – 2019. – Vol. 32. – pp. 8024–8035.
26. Горчаков А. В., Демидова Л. А., Советов П. Н. Интеллектуальный учёт учебных достижений в системе цифровой ассистент преподавателя // *International Journal of Open Information Technologies*. – 2023. – Т. 11, № 4. – С. 106–115.
27. Campbell G. A. Cognitive Complexity: An Overview and Evaluation / *Proceedings of the 2018 international conference on technical debt*. – 2018. – pp. 57–58.
28. Barollet T., Guillon C., Selva M., Broquedis F., Bouchez-Tichadou F., Rastello F. EasyTracker: A Python Library for Controlling and Inspecting Program Execution / *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. – IEEE, 2024. – pp. 359–372.

References

1. Abramova E. S. Comparison of extreme learning machine and backpropagation method in the human activity recognition // *Izvestiya Tula State University* – 2023. – No. 2. – pp. 117–124.
2. Pronevich O.B., Klokova A.P. Analysis of UMAP, the method for reducing the dimensionality of initial data in machine learning for the purpose of failure prediction in a motive power service // *Dependability*. – 2022. – Vol. 22, No. 4. – pp. 53–62.
3. Shishin K. A., Muravev I. V., Kulikov E. K. On the Automated Unit Tests Generation for Java Applications using Spring Framework // *Trudy ISP RAN/Proc. ISP RAS*. – 2024. – Vol. 36, Iss. 2. – pp. 59–72.
4. Neves L., Campos O., Santos R., Magalhaes C., Santos I., Souza Santos R. de. Elevating Software Quality in Agile Environments: The Role of Testing Professionals in Unit Testing / *Proceedings of the 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. – IEEE, 2024. – pp. 293–296.
5. Mezhev T. P., Parygina D. A., Kutz D. O. Rust and Go Directed Fuzzing with LibAFL-DiFuzz // *Trudy ISP RAN/Proc. ISP RAS*. – 2026. – Vol. 38, Iss. 2. – pp. 69–82.
6. Raykin G. R., Pelevin M. S., Itsyson V. M. Semantic mutation strategy in grey-box fuzzing // *Trudy ISP RAN/Proc. ISP RAS*. – 2025. – Vol. 37, Iss. 2. – pp. 97–114.
7. Ivanov A. A., Dovgalyuk P. M., Fursova N. I. Review of existing methods for analyzing full-system code coverage // *Trudy ISP RAN/Proc. ISP RAS*. – 2025. – Vol. 37, Iss. 6 (4). – pp. 187–200.

8. Altmayer Pizzorno J., Berger E. D. Slipcover: Near Zero-Overhead Code Coverage for Python / *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. – 2023. – pp. 1195–1206.
9. Mishra D. B., Acharya B., Rath D., Gerogiannis V. C., Kanavos A. A Novel Real Coded Genetic Algorithm for Software Mutation Testing // *Symmetry*. – MDPI, 2022. – Vol. 14, No. 8. – P. 1525.
10. Dakhel A. M., Nikanjam A., Majdinasab V., Khomh F., Desmarais M. C. Effective Test Generation Using Pre-Trained Large Language Models and Mutation Testing // *Information and Software Technology*. – Elsevier, 2024. – Vol. 171. – P. 107468.
11. Smirnov A. V. Mutation testing of software // *Proceedings of Saint Petersburg Electrotechnical University*. – 2015. – No. 3. – pp. 28–31.
12. Kaufman S. J., Featherman R., Alvin J., Kurtz B., Ammann P., Just R. Prioritizing mutants to guide mutation testing / *Proceedings of the 44th International Conference on Software Engineering*. – 2022. – pp. 1743–1754.
13. Sychev O. A., Prokudin A. A., Denisov M. E. Intelligent tutoring system with automatic assignment generation according to the results of using an assignment bank // *Software & Systems*. – 2024. – Vol. 37, No. 2. – pp. 201–212.
14. Kishkin S. A. Sovershenstvovaniye uchebno-metodicheskogo kompleksa po distsipline "Prakticheskiye zadachi optimizatsii" // *E-Scio*. – 2022. – No. 7 (70). – pp. 278–284.
15. Valls-Vargas J., Zhu J., Ontañón S. Graph grammar-based controllable generation of puzzles for a learning game about parallel programming / *Proceedings of the 12th International Conference on the Foundations of Digital Games*. – 2017. – pp. 1–10.
16. Edwards S. H. Improving student performance by evaluating how well students test their own programs // *Journal on Educational Resources in Computing (JERIC)*. – ACM New York, NY, USA, 2003. – Vol. 3, № 3. – pp. 1–24.
17. Edwards S. H., Shams Z. Comparing test quality measures for assessing student-written tests / *Companion Proceedings of the 36th International Conference on Software Engineering*. – 2014. – pp. 354–363.
18. Perretta J., DeOrio A., Guha A., Bell J. On the Use of Mutation Analysis for Evaluating Student Test Suite Quality / *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. – 2022. – pp. 263–275.
19. Laurent T., Papadakis M., Kintis M., Henard C., Le Traon Y., Ventresque A. Assessing and Improving the Mutation Testing Practice of PIT / *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. – IEEE, 2017. – pp. 430–435.
20. Ermakov A. D., Yevtushenko N. V. Deriving Test Suites with the Guaranteed Fault Coverage for Extended Finite State Machines // *Modeling and Analysis of Information Systems*. – 2016. – Vol. 23, No. 6. – pp. 729–740.
21. Sovietov P. N. Automatic generation of programming exercises / *Proceedings of the 2021 1st international conference on technology enhanced learning in higher education (TELE)*. – IEEE, 2021. – pp. 111–114.
22. Gorchakov A. V., Fandeev I. I. Metodika avtomaticheskoy generatsii zadach na temu analiza nizkourovnevnykh predstavleniy koda / *Aktualnye problemy prikladnoy matematiki, informatiki i mekhaniki: Sbornik materialov Mezhdunarodnoy nauchnoy konferentsii Aktualnye problemy prikladnoy matematiki, informatiki i mekhaniki (Russia, Voronezh, 1–3 December 2025)*. – Voronezh, 2026. – pp. 4–10.
23. Sovietov P. N., Gorchakov A. V. Models and Algorithms for Automatic Generation of Educational Problems on Git System Usage and Package Dependency Resolution // *Electronic Scientific Journal "IT-Standard"*. – 2025. – №. 3. – pp. 47–57.
24. Harris C. R., Millman K. J., Walt S. J. van der, Gommers R., Virtanen P., Cournapeau D., Wieser E., Taylor J., Berg S., Smith N. J., Kern R., Picus M., Hoyer S., Kerkwijk M. H. van, Brett M., Haldane A., Río J. F. del, Wiebe M., Peterson P., Gérard-Marchant P., Sheppard K., Reddy T., Weckesser W., Abbasi H., Gohlke C., Oliphant T. E. Array programming with NumPy // *Nature*. – Springer Science; Business Media LLC, 2020. – Vol. 585, No. 7825. – pp. 357–362.
25. Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., Killeen T., Lin Z., Gimelshein N., Antiga L., Desmaison A., Kopf A., Yang E., DeVito Z., Raison M., Tejani A., Chilamkurthy S., Steiner B., Fang L., Bai J., Chintala S. PyTorch: An Imperative Style, High-Performance Deep Learning Library // *Advances in Neural Information Processing Systems*. – 2019. – Vol. 32. – pp. 8024–8035.
26. Gorchakov A. V., Demidova L. A., Sovietov P. N. Intelligent Accounting of Educational Achievements in the Digital Teaching Assistant System // *International Journal of Open Information Technologies*. – 2023. – Vol. 11, No. 4. – pp. 106–115.
27. Campbell G. A. Cognitive Complexity: An Overview and Evaluation / *Proceedings of the 2018 international conference on technical debt*. – 2018. – pp. 57–58.

28. Barollet T., Guillon C., Selva M., Broquedis F., Bouchez-Tichadou F., Rastello F. EasyTracker: A Python Library for Controlling and Inspecting Program Execution / Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). – IEEE, 2024. – pp. 359–372.