DOI 10.37882/2223-2966.2024.05.34

УЛУЧШЕНИЕ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЗА СЧЕТ УСКОРЕНИЯ МУТАЦИОННОГО ТЕСТИРОВАНИЯ

IMPROVING SOFTWARE QUALITY BY SPEEDING UP MUTATION TESTING

A. Filimonov

Summary. Mutation testing has a long history that starts back in the 1970s. The process of mutation analysis has undergone many improvements. But despite this, there are still areas for development. The number of publications is growing year after year, which shows the interest of system engineers in this topic and its relevance. Since mutation testing can be divided into several steps, most research focuses on optimizing a specific step rather than the whole process. This paper will focus on test cases and the implementation of an algorithm for their effective use in mutation testing. The algorithm being developed aims to show that selecting relevant test cases can reduce the resource cost of performing automated mutation testing. The reason for implementing such an algorithm is that testing has always been an important step in the development process, with written tests not always being able to validate the quality of the software. In order to improve the quality of testing, mutation testing is used. Conducting mutation testing is a resource-intensive process; the new approach is supposed to save the resources and make the process available to developers. The result of this work is a test case selection method based on commits and test coverage and a software tool module for mutation testing. To achieve the desired result, a systematic analysis of research work in this direction has been done. Also, test case selection algorithms have been developed, based on them the module for mutation testing tool has been implemented and tested on open source projects to confirm the applicability.

Keywords: testing quality, mutant selection, test suite reduction, test coverage, git commits.

Филимонов Артём Александрович
Аспирант, Российский экономический университет им. Г.В. Плеханова
pe4enko111@rambler.ru

Аннотация. Мутационное тестирование имеет достаточно длинную историю развития, которая началась еще в 1970-х годах. Процесс мутационного анализа претерпел большое количество улучшений. Но, несмотря на это, все еще остаются области для развития. Количество публикаций растет из года в год, что говорит об интересе системных инженеров к этой теме и ее актуальности. Поскольку мутационное тестирование можно разбить на несколько этапов, большинство исследований посвящено оптимизации конкретного этапа, а не всего процесса. Данная работа будет посвящена тестовым случаям и реализации алгоритма для их эффективного использования в мутационном тестировании. Разрабатываемый алгоритм призван показать, что выбор релевантных тестовых случаев может снизить затраты ресурсов на выполнение автоматизированного мутационного тестирования. Причина внедрения такого алгоритма заключается в том, что тестирование всегда было важным этапом процесса разработки, при этом написанные тесты не всегда могут подтвердить качество программного обеспечения. Для того, чтобы улучшить качество тестирования используется мутационное тестирование. Проведение мутационного тестирования — ресурсоемкий процесс; новый подход позволит сократить затрачиваемые ресурсы и сделать процесс доступным для разработчиков. Результатом работы является метод выбора тестовых случаев, основанный на коммитах и тестовом покрытии, и модуль программного инструмент для мутационного тестирования. Для достижения желаемого результата был проведен систематический анализ исследовательских работ в данном направлении. Также, были разработаны алгоритмы выбора тестовых случаев, на их основе реализован модуль для инструмента мутационного тестирования и проверен на открытых проектах для подтверждения применимости.

Ключевые слова: качество тестирования, выбор мутантов, сокращение тестового набора, покрытие тестов, git коммиты.

Введение

Процесс разработки программного обеспечения делится на несколько этапов, каждый из которых имеет решающее значение для успешного продукта. Фаза тестирования отвечает за обеспечение качества. Разработчикам важно обеспечить качественное тестирование, чтобы устранить ошибки на ранних стадиях. Возникает вопрос — как определить, что написанные тестовые случаи адекватны в выявлении ошибок? Существует несколько видов тестирования, которые используются для разных целей. На первый взгляд покрывающего тестирования должно быть достаточно для определения качества, так как оно позволяет определить, какие участки кода тестируются. Но на самом деле его недостаточно для 100-процентного покрытия тестов. Покрывающее тестирование не охватывает синтаксические аспекты кода, где еще могут проявиться ошибки. Для оценки актуальности и адекватности тестового набора можно использовать мутационное тестирование. Мутационное тестирование — это процесс использования мутационного анализа для количественной оценки сильных сторон тестового набора. Целью мутационного анализа является генерация семантических вариантов программ путем автоматического изменения синтаксиса программы и сравнения с оригинальной программой [1].

В настоящее время большинство исследований в области мутационного тестирования посвящено оптимизации процесса тестирования. В [1], [2] подробно описано

мутационное тестирование и его применение. Современный процесс мутационного тестирования включает 10 шагов, которые изображены на рис. 1. Процесс начинается с выбора мутантов. Мутант — программа, содержащая дефекты в коде, полученные путем изменения синтаксиса в соответствии с определенными правилами. Этот этап предполагает два шага: выбор мутантных операторов и сокращение мутантов. Выбор операторов зависит от различных условий, таких как тип языка программирования, приложения или ошибки. Когда мутантные операторы выбраны, необходимо выбрать стратегию сокращения мутантов, так как после генерации мутантов их количество может быть огромным и приводить к увеличению затрат на их выполнение. Кроме того, не все мутанты будут валидными из-за проблем с компиляцией. Создание мутантов также является изученной темой с рядом предложенных подходов для ее оптимизации. На следующем этапе необходимо удалить эквивалентные и избыточные мутанты, поскольку они искажают результаты мутационного тестирования, либо завышая, либо занижая достигнутый уровень покрытия [1]. При необходимости на основе мутантов могут быть сгенерированы тестовые случаи. Далее мутанты выполняются. В [1] предлагается два основных сценария выполнения мутантов. Выполнять мутанта до тех пор, пока он не погибнет, или выполнять для всех тестовых случаев и строить матрицу мутантов. В зависимости от количества мутантов, погибших во время выполнения, вычисляется балл мутации. На шаге 7 лишние тестовые случаи могут быть исключены из тестирования, а оставшиеся — расставлены по приоритетам. На самом деле существует всего несколько исследований на эту тему, но предполагается, что сокращение числа тестовых случаев может повысить эффективность процесса мутационного тестирования при сохранении качества тестов. На следующем этапе необходимо определить, является ли полученная оценка мутации удовлетворительной или нет. Пороговое значение приемлемой оценки мутации должен определить человек, так как универсального подхода для разных случаев не существует. Если было решено, что результаты неудовлетворительны, шаги 4–7 повторяются. На последнем шаге программа должна быть проверена на корректность поведения. Если есть какие-либо ошибки, их следует устранить и повторить весь процесс.

Цель исследования

Внимание в данной работе направлено на этап сокращения и оптимизации тестов. Причиной этого является малое количество исследований данного этапа по сравнению с другими. Предполагается, что оптимизация этапа сокращения тестового набора может иметь значительный влияние на снижение стоимости мутационного тестирования. Основным фактором, влияющим на стоимость мутационного тестирования, является количество тестируемых мутантов. Если, например, выполнение тестового набора на оригинальной программе потребляет N ресурсов, то выполнение тех же тестов при тех же условиях на мутантах будет стоить $N \times M$ ресурсов, где M — количество мутантов. Очевидно, что мутанты можно выполнять параллельно, что сокращает время выполнения, но при этом увеличивается потребление

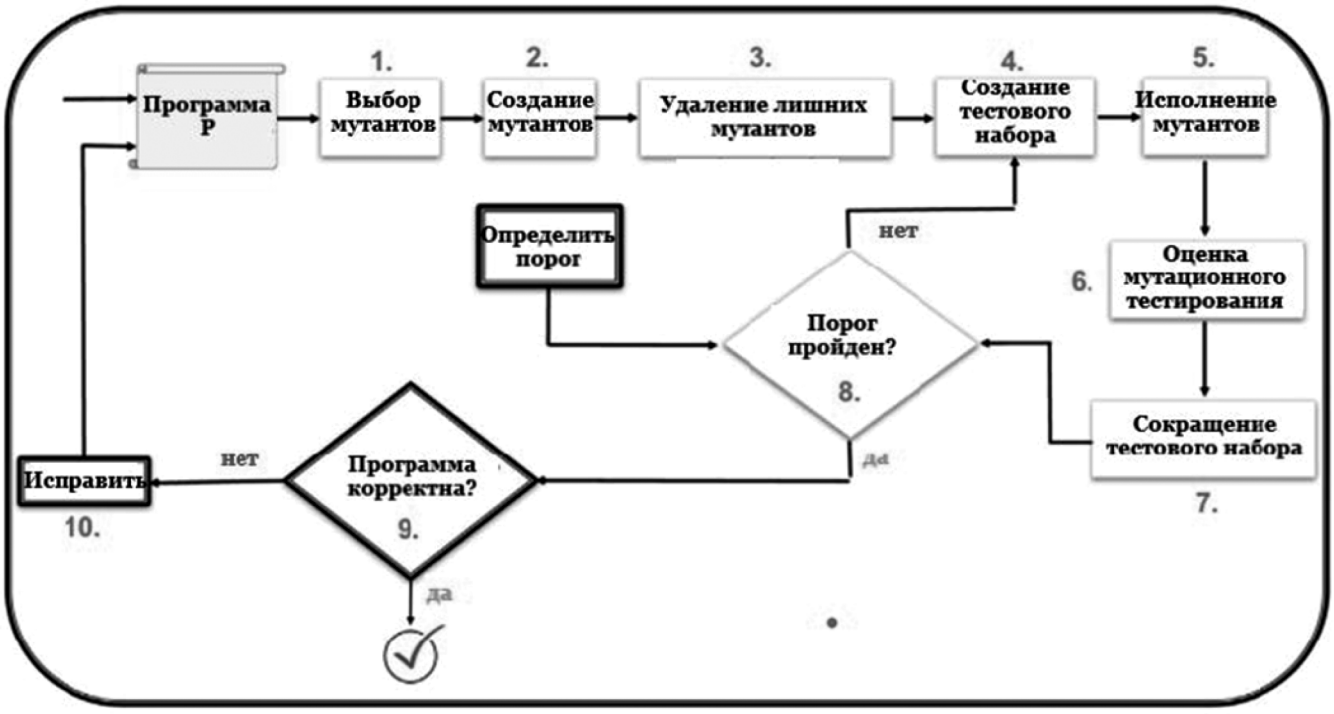

Рис. 1. Современное мутационное тестирование [1]

ресурсов процессора и памяти. Выигрыш от этого метода невелик. Другой подход заключается в уменьшении количества мутантов за счет удаления лишних. В качестве альтернативы этому подходу можно попробовать уменьшить количество тестовых случаев. Поскольку стоимость тестирования зависит от размера тестового набора, количество ресурсов будет меньше, если мы выполним меньше тестов. Вместе с этим возникает проблема локализации неисправностей. Когда мы уменьшаем набор тестов, есть вероятность, что некоторые ошибки не будут обнаружены во время тестирования. С мутационным тестированием все немного иначе, поскольку каждое выполнение мутанта подразумевает поиск дефектов в конкретном месте, а не во всей программе. Поэтому не обязательно выполнять все тесты, а только те, которые потенциально могут убить мутанта.

Современный процесс разработки стремится к применению автоматизации на всех своих этапах. Разработчики программного обеспечения заменяют ручное тестирование автоматизированным выполнением тестов. В этих условиях сложно интегрировать ресурсоемкое мутационное тестирование в процесс разработки. Весь процесс CI/CD для крупных проектов может занимать несколько часов, а если в него интегрировать мутационное тестирование, то время выполнения значительно увеличится. Один из возможных способов преодоления этой проблемы — проводить мутационное тестирование не на всей программе, а на ее измененной части. Для отслеживания состояния программы мы можем использовать историю коммитов проекта. При сравнении двух состояний программы выявляются различающиеся файлы, которые затем используются для мутационного тестирования. Поскольку нет необходимости запускать весь набор тестов на измененных файлах, необходимо выбрать соответствующие тестовые случаи. Выбор тестовых случаев может быть основан на информации о покрытии кода. Это традиционный сценарий сокращения тестового набора, хотя в некоторых инструментах мутационного тестирования он отсутствует. Полученная информация о тестах, убивающих мутантов, должна использоваться для определения приоритета их выполнения. Такой подход может быть использован для автоматизации мутационного тестирования.

Основная часть нового метода основана на информации о коммитах и поэтому требует интеграции с Git. Каждый Git-проект состоит из нескольких обязательных компонентов, которые формируют структуру версионного программного обеспечения. Коммиты — один из этих компонентов, и во временной шкале проекта они играют ключевую роль для сборки. Коммиты можно сравнить со снимками программного обеспечения, которые создаются в процессе разработки. На временной шкале Git-проекта может быть создано множество коммитов, что позволяет проследить историю разработки программного обеспечения с самого начала.

Таким образом, целью работы является метод сокращения количества тестовых случаев при автоматизированном выполнении мутационного тестирования. Объектом работы является мутационное тестирование, а предметом исследования — информация о коммитах и тестовом покрытии измененной программы. Для достижения цели работы необходимо решить следующие задачи:

1. Систематический анализ в области сокращения тестовых наборов.
2. Разработка метода и математической модели.
3. Разработка алгоритма и реализация в виде программного обеспечения.
4. Разработка эксперимента и метрик, используемых для валидации метода.
5. Проведение эксперимента на реальных проектах.
6. Сделать вывод о применимости метода.

Анализ проблемы

Предлагаемый в работе метод реализован на основе традиционного подхода к сокращению тестовых наборов. Традиционный метод включает в себя две оптимизации. Во-первых, тестовые случаи, которые не достигают мутированного оператора в исходной программе, не должны выполняться. Во-вторых, нет необходимости выполнять тестовые случаи, когда мутант был убит. Этот подход будет использоваться для сравнения результатов, полученных после внедрения нового подхода, основанного на информации о коммитах.

Сегодня каждый программист знаком с инструментами git. Они используются для управления версиями кода и его распространения. Тесная интеграция с другими инструментами DevOps дает разработчикам возможность автоматизировать некоторые процессы, одним из которых является автоматизация тестирования. Поскольку разработка — это непрерывный процесс, изменения вносятся постоянно. Изменения, вносимые разработчиками, можно отследить по истории коммитов. Коммиты хранят информацию о файлах и их содержимом. Затем содержимое сравнивается, и мы получаем разницу в программных инструкциях. Основываясь на изменениях в строках кода, мы можем сократить и приоритизировать тестовые случаи, которые должны быть выполнены во время мутационного тестирования.

Предлагается отслеживать историю мутационного тестирования и хранить результаты в специальном файле в репозитории проекта. Это должно помочь сделать повторное выполнение тестового набора более эффективным, благодаря имеющейся информации о результатах тестирования и примененных изменениях. Резуль-

таты предыдущего тестирования анализируются вместе с новыми коммитами, добавленными в историю. Файл должен содержать ID каждого мутанта, который когда-либо генерировался для проекта, и результаты этой мутации: был ли мутант убит или выжил. Если мутант был убит, то в результатах сохраняется соответствующее имя теста. При появлении новых мутантов или получении новых результатов тестирования файл обновляется. Предложенный метод поддерживает итеративный процесс разработки программного обеспечения и поэтому применим для использования в практике CI/CD.

Среди большого количества инструментов для мутационного тестирования есть и такие, которые обладают функциями для итеративного использования. В PIT [3] эти возможности заявлены как экспериментальные, но могут быть включены пользователем. Еще один инструмент — Mutmut, который сохраняет результаты в файл и повторно запускает тесты только тогда, когда изменение применяется к тестовому файлу. Mutmut не учитывает цель и условия изменения, необоснованно выполняя весь тестовый набор. Но инструмент запоминает уцелевшие мутанты, которые становятся единственной целью для проверки. Это очень простой подход, который упускает оптимизацию и приводит к появлению необнаруженных ошибок.

Новый подход учитывает изменения, которые были внесены в программный код, а также изменения в тестовом наборе. Эта информация используется для обнаружения пар «мутант — тест», которые можно сократить. Когда мы создаем и тестируем наш продукт итеративно, некоторые части программы могут меняться, некоторые остаются неизменными. Мы не хотим тратить ресурсы на тесты, которые дают один и тот же результат в течение нескольких запусков. В то же время не стоит забывать, что изменения могут повлиять на обнаружение мутантов как положительно, так и отрицательно.

Метод сокращения количества тестов зависит от мутантов, в частности от того, как они были сгенерированы. В этой работе мутанты будут генерироваться случайным образом по всей кодовой базе с учетом покрытия тестами. Одна из основных причин — сокращение времени мутационного тестирования.

Согласно итерационному подходу к разработке, код или набор тестов должен меняться от версии к версии. На самом деле, мутационное тестирование может выполняться по разным сценариям, и в данной работе выделяется 4 из них: ничего не изменилось, тестовый набор был изменен, код программы был изменен, код программы и тестовый набор были изменены. Все сценарии будут учтены в алгоритме.

Математическая модель

Для разработки правильного решения проблемы сокращения тестовых случаев необходимо описать ее в математических терминах. Предположим, что у нас есть набор мутантов $M = \{m_0 \dots m_i\}$ и тестовый набор $TS = \{t_0 \dots t_j\}$. После проведения мутационного тестирования мы получаем матрицу (1), где $a_{ij} = (m_i, t_j)$ — результат выполнения пары мутант — тест со значениями {0, 1}. Значение результата 1 означает, что тест успешно обнаружил мутанта, а 0 — что мутант выжил. Мы можем сократить набор пар мутант — тест (m_i, t_j), которые должны быть выполнены, заранее определив их результаты и подставив их в матрицу.

$$\begin{matrix} a_{11} & \dots & a_{1j} \\ \vdots & \ddots & \vdots \\ a_{i1} & \dots & a_{ij} \end{matrix} \quad (1)$$

В результате применения нового метода мы должны получить таблицу с уже заданными значениями для нескольких ячеек такую, как таблицу 1. В традиционном мутационном тестировании каждый запуск начинается с пустой таблицы, так как мы не сохраняем результаты предыдущих запусков и, следовательно, не можем предсказать новый. Предлагаемый подход предполагает, что результаты выполнения теста должны быть идентичны предыдущим результатам, если никакие изменения не повлияли на его выполнение. Таким образом, мы переносим некоторые значения предыдущего запуска в новый, который будет основан на таблице. Сохранение результатов тестирования зависит от изменений, которые были применены к новой версии проекта.

Таблица 1.

Предопределенные результаты мутационного тестирования

	t_0	t_1	t_2	t_3	t_4
m_0			1		
m_1		0		0	
m_2					
m_3	0	0	0	0	0

Набор тестов убивает мутанта только в том случае, когда результаты выполнения теста на мутировавшей программе $Tres_m$ отличаются от результатов на оригинальной $Tres_s$: $Tres_s \oplus Tres_m = True$. Рассмотрим мутантный оператор $m := f$, который соответствует строке программного кода. После завершения фазы генерации мутантов мы получаем набор мутантов $M = \{m_0 \dots m_i\}$. Описываемый подход к сокращению тестов основывается на покрытии кода тестами. $Cov(t)$ обозначает функцию поиска покрытия строк для теста; $Cov : t \rightarrow \{F_i'\}$,, где $F' = \{f_n\}_{n \in \{1, \dots size(F)\}}$ представляет собой набор строк

файла, покрываемых тестом. Когда для каждого теста определено покрытие, мы начинаем формировать сокращенный набор тестов для каждого мутанта. Тесты выбираются в соответствии с включением мутантного оператора в тестовое покрытие (2).

$$TS_i' = \left\{ m_i \cap Cov\left(t_j\right) \right\}_{j \in \{0,\ldots size(TS)\}} \quad (2)$$

Последним реализуемым свойством является то, что выполнение тестового набора останавливается после того, как мутант был убит. Данный подход дает значительное снижение затрат при сохранении оценки мутации. Количество сэкономленных ресурсов соответствует (3).

$$\sum_{i=0}^{size(M)} size(TS \setminus TS_i') \quad (3)$$

Предлагаемый алгоритм

В этом разделе предложен улучшенный алгоритм на основе традиционного. В него добавлены шаги, которые, как ожидается, приведут к ускорению всего процесса тестирования мутаций. Эти шаги выделены красным цветом на рис. 2. Три из этих шагов находятся на предварительном этапе, а «вычисление мутационной оценки» является практически последним шагом.

Полученные результаты

Изучив существующие инструменты тестирования, было решено сделать модуль для одного из них. Среди большого количества публичных репозиториев Github для мутационного тестирования были найдены 4, реализованные на Python: MutPy [4], Mutatest [5], Cosmic Ray [6] и Mutmut [7]. Все они были протестированы на применимость предлагаемого метода, и только один из них был выбран. Основная функциональность этих инструментов практически одинакова и похожа на традиционный алгоритм мутационного тестирования. Их отличает ряд новых возможностей и способ реализации глубокого мутационного процесса. Исходя из сравнения возможностей инструментов был выбран Mutmut для доработки в виде модуля.

После разработки программного обеспечения его необходимо было протестировать. Проект, который используется для определения корректности работы программы, был выбран еще до начала разработки. В широком спектре общедоступных Python-проектов искались репозитории, которые поддерживаются командой, имеют сложившуюся структуру проекта и хорошую тестовую базу. Проект Flask [8] имеет широкую известность и хорошую репутацию в сообществе разработчиков. Он соответсвует вышеперечисленным требованиям и поэтому был выбран для тестирования модифицированной программы Mutmut.

Программа Mutmut изначально тестировалась с помощью ручных тестов, а также модифицированных версий, существующих автотестов. В основном автотесты запускались для проверки новых возможностей программы, таких как генерация ограниченного числа мутантов или использование покрытия для выбора тестов. Далее программа Mutmut запускалась на проекте Flask и результаты сравнивались с оригинальной программой Mutmut.

Когда тестирование закончилось и было установлено, что программа корректно работает на Flask, можно ставить эксперимент. Основная цель эксперимента — проверить корректность предложенного метода сокращения тестовых случаев. Поскольку мы рассчитываем сократить количество выполняемых тестовых случаев, а также количество генерируемых мутантов при сохранении оценки мутации, результаты эксперимента должны показать, сохранится ли он при многочисленных запусках или нет.

Так как необходимо протестировать результаты между версиями, то мы делим историю коммитов на 6 частей. Первый коммит для проверки — это 100 коммитов от последнего коммита ветки (HEAD~100). Другие — HEAD~80, HEAD~60, HEAD~40, HEAD~20 и HEAD~0, который является последним коммитом мастер-ветви.

Результаты мутационного тестирования на проекте Flask приведены в таблице 2. Для каждой ревизии мы собрали оценку мутации (MS), время выполнения (ET) и количество созданных мутантов (CM). Как мы видим, во всех ревизиях, кроме HEAD~0, оценка мутации модифицированного Mutmut отличается от оценки мутации оригинального Mutmut не более чем на 2 %. Согласно [9], такое отклонение описывается 95 % примеров. Авторы также доказывают, что мутационная оценка образца может иметь абсолютную ошибку в 7 %, что является нормальным. Таким образом, оценка мутации HEAD~0 также удовлетворяет критерию приемлемости. Чтобы убедиться в том, что отклонение существует всегда, программа была протестирована повторно. Вновь полученная оценка мутации составила 66,5 %, что близко к оценке мутации исходной программы. Предполагается, что отклонение может быть вызвано неудачной выборкой. Среди прочих результатов можно отметить, что модифицированная программа завершилась быстрее оригинальной, за исключением первой ревизии (HEAD~100). Одним из факторов, влияющих на скорость, является уменьшение количества пар мутант — тестовый случай. Количество создаваемых мутантов меньше размера выборки, что дает выигрыш в эффективности.

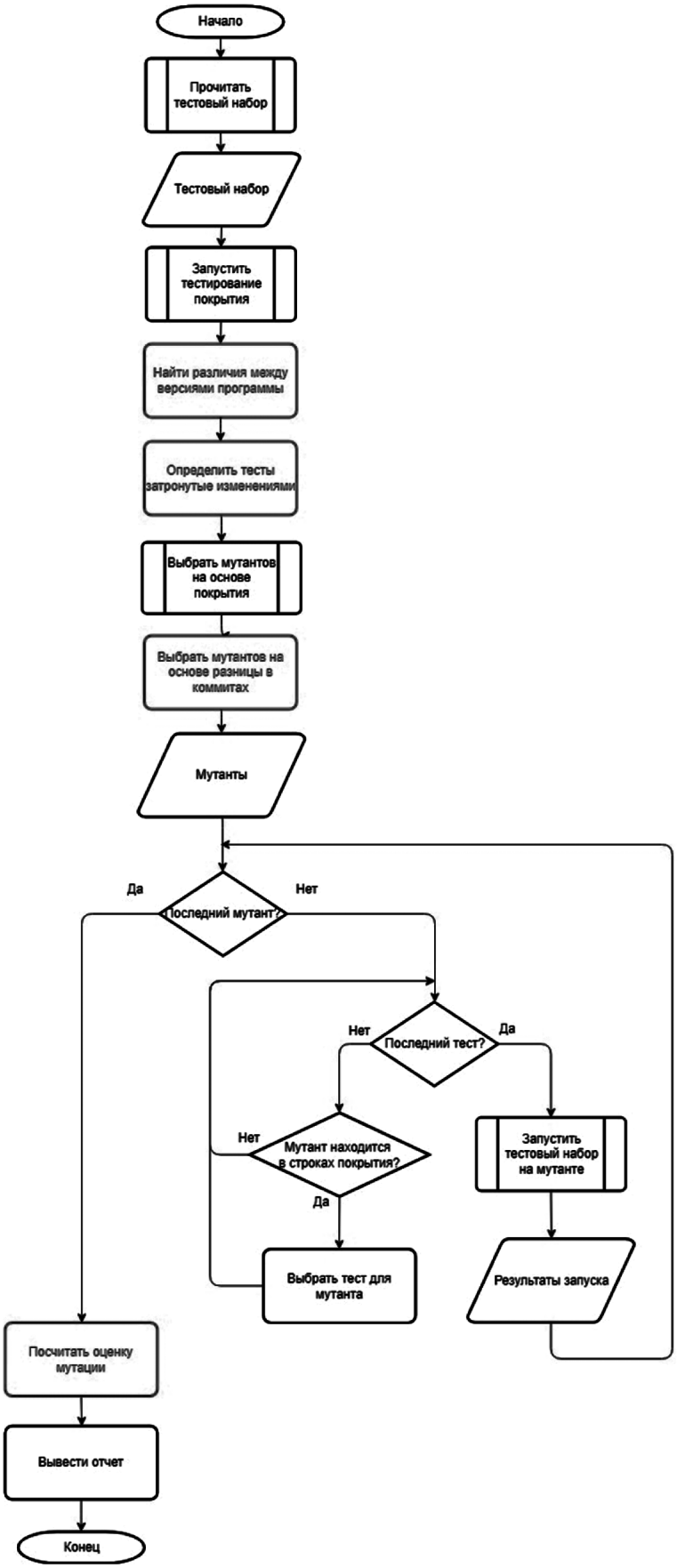

Рис. 2. Улучшенный алгоритм для мутационного тестирования

Таблица 2.

Мутационное тестирование на проекте Flask

Ревизия	HEAD~100			HEAD~80			HEAD~60		
Критерий	MS	ET	CM	MS	ET	CM	MS	ET	CM
Оригинальный Mutmut	63,75 %	13m8s	400	61,25 %	10m2s	400	62,75 %	9m15s	400
Модифицированный Mutmut	63,75 %	13m8s	400	62,00 %	9m16s	293	61,25 %	6m20s	218
Ревизия	HEAD~40			HEAD~20			HEAD~0		
Критерий	MS	ET	CM	MS	ET	CM	MS	ET	CM
Оригинальный Mutmut	67,00 %	8m28s	400	65,00 %	9m15s	400	67,00 %	8m34s	400
Модифицированный Mutmut	65,00 %	7m16s	288	63,75 %	1m43s	29	61,50 %	3m53s	169

Заключение

Все цели, поставленные в начале работы, были успешно выполнены:

1. Проведено систематический анализ исследование в области сокращения тестовых наборов.
2. Разработан метод сокращения пар мутант — тест.
3. Разработана математическая модель, описывающая метод.
4. Разработан алгоритм и реализован в виде программного обеспечения.
5. Разработан эксперимент и метрики, используемые для валидации метода.
6. Проведен эксперимент на реальных проектах.

Эксперимент по проверке применимости предложенного метода показал хорошие результаты, выбранный проект демонстрирует, что можно сократить набор мутантов, проверяемых между итерациями, предсказав рез часть набора мутантов. Мы берем случайные мутанты из выборки, и размер выборки не всегда задается правильно. Мутационное тестирование — дорогостоящая процедура, и перед ее проведением тестировщикам следует установить корректное количество используемых мутантов, которое будет использоваться вместо полного набора мутантов. Описанные в работе исследования показали, что выборка может быть достаточно мала, что было подтверждено и в данной работе.

ЛИТЕРАТУРА

1. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. and Harman, M., 2019. Mutation Testing Advances: An Analysis and Survey. Advances in Computers, pp.275–378.
2. Jia, Y. and Harman, M., 2011. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, 37(5), pp.649–678.
3. PIT (no date) Incremental analysis. Режим доступа: https://pitest.org/quickstart/incremental_analysis/ (дата обращения: 20 февраля 2024).
4. Mutpy (no date) Mutpy/Mutpy: MutPy is a mutation testing tool for python 3.x source code, GitHub. Режим доступа: https://github.com/mutpy/mutpy (дата обращения: 15 февраля 2024).
5. EvanKepner (no date) Evankepner/mutatest: Python mutation testing: Test your tests! safely run mutation trials without source code modifications and see what will get past your test suite., GitHub. Режим доступа: https://github.com/EvanKepner/mutatest (дата обращения: 15 февраля 2024).
6. Sixty-North (no date) Sixty-north/cosmic-ray: Mutation testing for python, GitHub. Режим доступа: https://github.com/sixty-north/cosmic-ray (дата обращения: 15 февраля 2024).
7. Boxed (no date) Boxed/Mutmut: Mutation Testing System, GitHub. Режим доступа: https://github.com/boxed/mutmut (дата обращения: 15 февраля 2024).
8. Pallets (no date) Pallets/flask: The Python Micro Framework for building web applications., GitHub. Режим доступа: https://github.com/pallets/flask (дата обращения: 10 февраля 2024).
9. R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, A. Groce, How hard does mutation analysis have to be, anyway? in: 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2–5, 2015, 2015, pp, 216–227.

Журнал «Современная наука: актуальные проблемы теории и практики»